Cookbook

Contents

Chapter 1: Intel® VTune™ Profiler Performance Analysis Cookbook

Methodologies 4
Analyzing Uncore Perfmon Events for Use of Intel® Data Direct I/O Technology in Intel® Xeon® Processors (NEW) 4
Top-down Microarchitecture Analysis Method 20
OpenMP* Code Analysis Method 29
Custom Data Collection for Performance Analysis (NEW) 34
Software Optimization for Intel® GPUs (NEW) 42
Core Utilization in DPDK Apps 57
PCIe Traffic in DPDK Apps 62
DPDK Event Device Profiling 67
Effective Utilization of Intel® Data Direct I/O Technology 72
Compile a Portable Optimized Binary with the Latest Instruction Set 81
Configuration Recipes 87
Profiling OpenVINO™ Applications (NEW) 88
Profiling High Bandwidth Memory Performance on Intel® Xeon® CPU Max Series (NEW) 92
Profiling Windows* Applications for Hybrid CPU Platforms (NEW) 99
Viewing Analysis Results on a Web Browser (NEW) 107
Profiling Machine Learning Applications (NEW) 111
Profiling Single-Node Kubernetes* Applications (NEW) 118
Analyzing Hot Code Paths Using Flame Graphs (NEW) 127
Improving Hotspot Observability in a C++ Application Using Flame Graphs 133
Measuring Performance Impact of NUMA in Multi-Processor Systems 148
Profiling Games built with Unity* (NEW) 160
Profiling Games built with Unreal Engine* (NEW) 169
Profiling Java Applications as a Remote User (NEW) 173
Profiling JavaScript* Code in Node.js* 177
Analyzing CPU and FPGA (Intel® Arria® 10 GX) Interaction 180
Profiling a .NET* Core Application 185
Profiling Applications in Amazon Web Services* (AWS) EC2 Instances 191
Enabling Performance Profiling in GitLab* CI 195
Configuring a Hyper-V* Virtual Machine for Hardware-Based Hotspots Analysis 201
Profiling an Application for Performance Anomalies (NEW) 207
Profiling an OpenMP* Offload Application running on a GPU 213
Profiling a SYCL* Application running on a GPU 224
Profiling an FPGA-driven SYCL* Application 229
Profiling Hardware Without Intel Sampling Drivers 234
Profiling MPI Applications 241
Profiling Docker* Containers 250
Profiling a Remote Target Through a Proxy Server (NEW) 260
Profiling in an Apptainer* Container 262
Profiling Linux*, Android*, and QNX* System Boot Time 264
Using Intel® VTune™ Profiler Server with Visual Studio Code and Intel® DevCloud for oneAPI (NEW) 273
Using Intel® VTune™ Profiler Server in HPC Clusters 277

Using the Command-Line Interface to Analyze the Performance of a SYCL* Application running on a GPU (NEW) 283
Tuning Recipes 292
Cache-Related Latency Issues in Segmented Cache Environment 292
False Sharing 299
Frequent DRAM Accesses 305
Poor Port Utilization 310
Page Faults 319
Instruction Cache Misses 322
OS Thread Migration 328
OpenMP* Imbalance and Scheduling Overhead 331
Processor Cores Underutilization: OpenMP* Serial Time 338
Scheduling Overhead in an Intel® oneAPI Threading Building Blocks Application 344
PMDK Application Overhead 350
Notices and Disclaimers 354

Intel® VTune™ Profiler Performance Analysis Cookbook 1

This Cookbook introduces methodologies and use-case recipes to analyze the performance of your code with VTune Profiler, a tool that helps you identify ineffective algorithm and hardware usage and provides tuning advice.

- To download VTune Profiler or request product support, visit the product page.
- All recipes in this Cookbook are scalable. You can apply them to any version of VTune Profiler, or 2018 and newer versions of its predecessor, Intel® VTune™ Amplifier . Slight version-specific configuration changes are possible.
- Some recipes were translated to Simplified Chinese, and are available in the Simplified Chinese version of this Cookbook.
- Intel® VTune™ Profiler is a new renamed version of Intel® VTune™ Amplifier.

Documentation Formats for Intel® VTune™ Profiler
Starting with VTune Profiler 2023, you can access both online and offline (PDF) versions of the documentation by selecting the required version from the **Version** drop-down menu on top of the web page. To download the PDF version, click the **Download** button in the top right corner of the page.

Access documentation for VTune Profiler release versions prior to 2023 in these ways:

VTune Profiler Release Version	Available Documentation Formats	Links
2021 - 2022.4 releases	Online	Online documentation for VTune Profiler
	Offline	Downloadable documents for VTune Profiler
Versions older than 2021 release	Offline only	Available documentation downloads by product version: • Download Documentation for Intel Parallel Studio XE • Download Documentation for Intel System Studio

Methodologies

Start cooking your performance analysis. Understand tuning techniques, performance metrics and hardware solutions to collect statistics. Next, drill down to particular tuning or configuration recipes that feature Intel® VTune™ Profiler or its predecessor, Intel® VTune™ Amplifier.

Analyzing Uncore Perfmon Events for Use of Intel® Data Direct I/O Technology in Intel® Xeon® Processors (NEW)

Use Intel® VTune™ Profiler to understand the utilization efficiency of Intel® Data Direct I/O (Intel® DDIO) technology, a hardware feature of Intel® Xeon® processors.

Content Experts: Alexander Antonov, Yu Zhang

When you profile software applications for performance, an uncore event refers to the functions of the CPU that operate outside the core. These functions can relate to the operations of:

- the memory controller
- the UPI block
- I/O stacks

You can use Intel VTune Profiler to monitor the count of uncore hardware performance events. Together with other performance metrics from analysis runs, this information can help you dive deep into PCIe traffic and behavior.

This recipe demonstrates how you use uncore events, related to PCIe, to analyze DDIO and VT-d efficiency. This recipe uses the **Input and Output analysis type** in Intel VTune Profiler.

The I/O metrics and events described in this recipe are based on the use of the Intel® Xeon® server platforms code named Ice Lake. Since there can be different uncore events per platform, there may be some differences for the same metric calculation.

Ingredients

- **Application:** DPDK `testpmd` application
- **Analysis Tool:**Intel® VTune™ Profiler version 2023.2 or newer (Input and Output analysis)
- **Platforms:** Two Intel® Xeon® Platinum 8358 CPU @ 2.60GHz platforms connected through an Ethernet cable
- **Network Interface Card:** Intel® Corporation Ethernet Controller X710 for 10GBASE-T
- **Operating System:** Ubuntu 22.04.3 LTS

Directions

1. Run Input and Output Analysis
2. Analyze Results
3. Group Results into Views

Run Input and Output Analysis

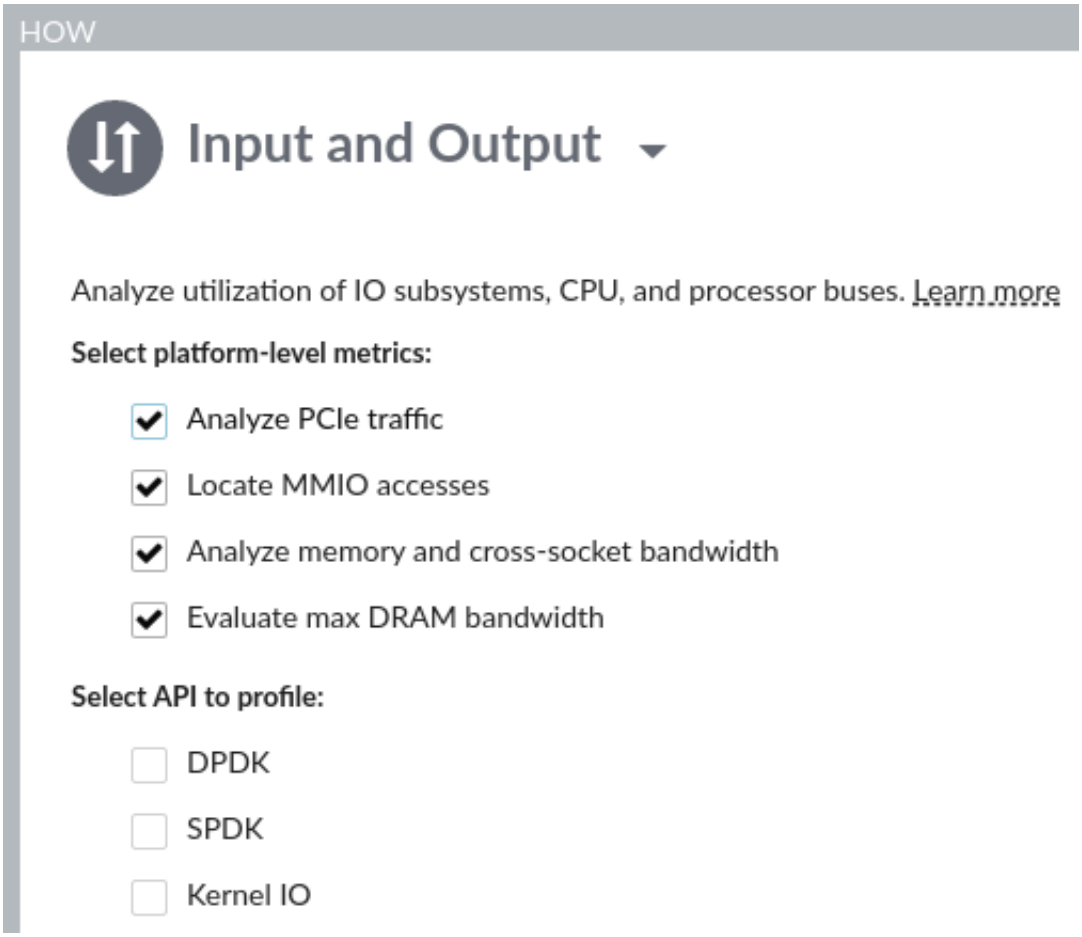
Before you begin:

- Make sure to use 1st or higher generations of Intel® Xeon® Scalable processors.
- Check to confirm that the sampling driver is loaded.
- Plan to collect data for at least 20 seconds.

This procedure uses the **Input and Output analysis** in VTune Profiler to collect Intel DDIO utilization efficiency metrics. When you run the analysis in driverless mode, there may be some limitations.

1. From the Analysis Tree in VTune Profiler, open the **Input and Output** analysis type.
2. In the **WHAT** pane, do one of the following:
   - Select **Launch Application** and specify the path to the application and any parameters.
   - Select **Attach to Process** and specify the PID

   If necessary, use the **Automatically stop collection after (sec)** option to control the collection time.
3. In the **HOW** pane, check the **Analyze PCIe traffic** checkbox to collect Intel DDIO utilization efficiency metrics.



4. Click the  **Start** button to run the analysis.

To run this analysis on the dpdk-testpmd benchmark from the command line, type:

```
# vtune -collect io -knob iommu=true -- dpdk-testpmd -l 32-35 -n 8 -- --mbcache=0 -mbuf-
size=65535 -total-num-mbufs=8192
```

Run this analysis on one of the available systems. On the other system, generate PCIe traffic:

```
# dpdk-testpmd -l 0-3 --  --forward-mode=txonly
```

Analyze Results

Once data collection is complete, download the results to your workstation. You can then open the results in the VTune Profiler GUI or VTune Profiler Web Server.

This recipe demonstrates the use of the latter.

Summary Window

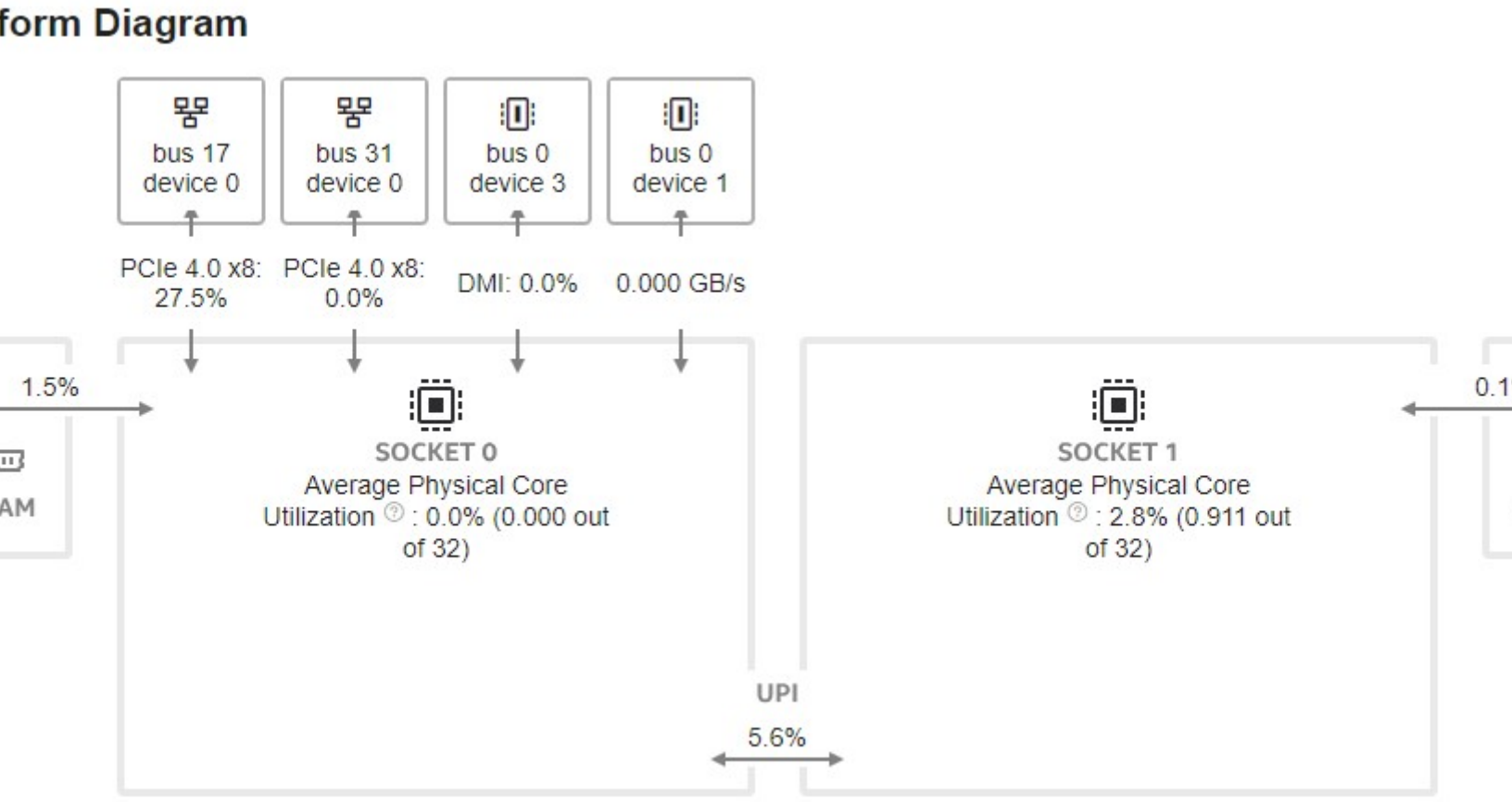
Start your examination of results in the Summary window, where you see a general overview of the execution of the application.



Platform Diagram

Next, look at the Platform Diagram. You see information about:

- System topology and physical core utilization
- Average DRAM use for each CPU socket package
- Average PCIe utilization for each PCIe device with known device capabilities. For unknown capabilities, the diagram displays the PCIe Average Bandwidth.
- Utilization metric for Intel® Ultra Path Interconnect (Intel UPI) cross-socket links



PCIe Traffic

Let us now look at metrics in the PCIe Traffic Summary section:



Inbound and Outbound PCIe Bandwidth

To understand how I/O traffic processes on the platform, look at the first level of PCIe bandwidth metrics.

PCIe traffic includes Inbound and Outbound traffic.

Inbound traffic: This type of traffic is initiated by I/O devices. These can be external or internal PCIe devices that read from or write to the system memory. Inbound traffic is processed through Intel Data Direct I/O (DDIO) Technology.

This technology can help improve the efficiency of I/O data processing for delivery and consumption of data from I/O devices. Instead of the DRAM, Intel® DDIO makes the L3 cache the primary destination and source of I/O data. By avoiding multiple read and write operations to the DRAM, Intel® DDIO achieves these benefits:

- Reduced inbound read and write latency
- Increased I/O bandwidth
- Reduced DRAM access
- Reduced power consumption

An Inbound Read operation has the I/O device reading from system memory.

An Inbound Write operation has the I/O device writing to the system memory.

Outbound traffic: This type of traffic is initiated by the CPU. These operations include reading from or writing to the memory or registers of I/O devices. Typically, the CPU accesses the device memory through the Memory-Mapped I/O (MMIO) space.

An Outbound Read operation has the CPU reading from I/O device registers.

An Outbound Write operation has the CPU writing to the I/O device registers.

The Integrated I/O (IIO) performance monitoring events are used to calculate PCIe bandwidth. IIO stacks manage traffic between the PCIe and Mesh domains. These tables list the set of uncore events that are used to count PCIe bandwidth by the channels (or parts) of the I/O stacks. The uncore events are mapped to the direction of the traffic which they count.

Direction	Read Uncore Events	Description
Inbound	UNC_IIO_DATA_REQ_OF_CPU.MEM_READ.PART[0-7]	Four bytes data request from system memory

Direction	Read Uncore Events	Description
Outbound	UNC_IIO_DATA_REQ_OF_CPU.CMPD.PART[0-7]	I/O device sends completion to CPU four bytes request

Direction	Write Uncore Events	Description
Inbound	UNC_IIO_DATA_REQ_OF_CPU.MEM_WRITE.PART[0-7]	Four bytes data request to system memory
Outbound	UNC_IIO_DATA_REQ_BY_CPU.MEM_WRITE.PART[0-7]	Four bytes data requested by CPU

To understand better the mapping of IIO performance events on the channels of I/O stacks, see the Uncore Performance Monitoring Reference Manual.

The events described in these tables increment once for every 4 bytes of data. To calculate the average PCIe bandwidth, multiply the event count by 4 and divide by the sample time.

Average PCIe Bandwidth (MB/s) = (Event count * 4 bytes / 10^6) / seconds sampled

When device capabilities are known and the maximum physical bandwidth can be calculated, the device link is attributed with the **PCIe Effective Link Utilization** metric. This metric is the ratio of bandwidth consumed on data transfers to the available physical bandwidth.

This metric does not account for protocol overhead (TLP headers, DLLPs, or physical encoding) and the metric reflects link utilization in terms of payloads. Thus, the **PCIe Effective Link Utilization** metric cannot reach 100%. However, this metric can inform about how far the link is from saturation. You can calculate the **PCIe Effective Link Utilization** metric this way:

PCIe Effective Utilization (%) = (Average PCIe Total Bandwidth (MB/s) / PCIe Max Bandwidth (MB/s)) * 100%

Average PCIe Total Bandwidth (MB/s) = (Average Inbound PCIe Bandwidth (MB/s) + Avg Outbound PCIe Bandwidth (MB/s))

Average Inbound/Outbound PCIe Bandwidth (MB/s) = (Average Read Bandwidth (MB/s) + Average Write Bandwidth (MS/s))

The calculation of PCIe Max Bandwidth is based on the `max_link_width` and `max_link_speed` sysfs attributes which are provided by the OS:

PCIe Max Bandwidth (MB/s) = Max Link Speed (GT/s per x1 link) * 1000 / 8 * Max Link Width * 2

L3 Hit/Miss Ratios for Inbound Requests

The L3 Hit/Miss Ratios for Inbound I/O requests are the ratios of requests made by I/O devices to the system memory that hit or miss the L3 cache. To learn more about using Intel® DDIO efficiently, see Effective Utilization of Intel® Data Direct I/O Technology in the Intel® VTune™ Profiler Performance Analysis Cookbook.

The Caching/Home Agent (CHA) and Integrated I/O Ring Port (IRP) Performance Monitoring Events are used to calculate the L3 Hit/Miss ratios.

The CHA is a distributed agent that maintains memory coherence between core caches, IIO stacks, and sockets.

The IIO Ring Port (IRP) maintains coherency for IIO traffic that targets coherent memory.

The following table describes the events which are used to calculate these metrics.

Traffic	Uncore Events	Alias	Description
Read	UNC_CHA_TOR_INSERTS.IO_MISS_PCIRDCUR	X	These are `PCI Read Current (PCIRdCurs)` requests issued by IO Devices that missed the LLC. These requests are full cache line read requests to get the most current data.
	UNC_I_FAF_INSERTS	Y	These are inbound read requests to coherent memory. They are received by the IRP and inserted into the `Fire and Forget (FAF)` queue.
Write	UNC_CHA_TOR_INSERTS.IO_MISS_ITOM	A	These are `Invalid to Modified (ItoM)` requests issued by IO Devices that missed the LLC. ItoM is an ownership request without data, that is used for full cache line writes.
	UNC_CHA_TOR_INSERTS.IO_MISS_ITOMCACHENEAR	B	The `ItoMCacheNears` indicates a partial write request from IO Devices that missed the LLC.
	UNC_I_TRANSACTIONS.WR_PREF	C	These are Inbound write (fast path) requests to coherent memory, that are received by the IRP. These requests result in write ownership requests issued by IRP to the mesh.
	UNC_I_MISC1.LOST_FWD	D	The snoop filter pulled away ownership before a write was committed

The aliases listed in the table above represent the corresponding traffic in the formulae below.

L3 Hit/Miss Ratios for Inbound PCIe Read Bandwidth	L3 Hit/Miss Ratios for Inbound PCIe Write Bandwidth
Inbound PCIe Read L3 Miss (%) = (X) / (Y)* 100%	Inbound PCIe Write L3 Miss (%) = ((A) + (B)) / ((C) + (D)) * 100%
Inbound PCIe Read L3 Hit (%) = 100% - Miss Ratio (%)	Inbound PCIe Write L3 Hit (%) = 100% - Miss Ratio (%)

Average Latency for Inbound Requests

The **Average Latency** metric for Inbound Read/Write requests gives the average amount of time spent by the platform on processing inbound read/write requests for a single cache line. The calculation of this metric is based on Little's Law:

Latency = Occupancy / Inserts

When the request is processed, a completion buffer holds items. Information about the occupancy of this buffer and the request arrival rate (inserts) enables the calculation of the average request processing time.

The table below describes uncore performance events which are used to calculate these metrics.

Traffic	Uncore Events	Alias	Description
Read	UNC_IIO_COMP_BUF_OCCUPANCY.CMPD	X	PCIe Completion Buffer Occupancy
	UNC_IIO_COMP_BUF_INSERTS.CMPD	Y	PCIe Completion Buffer Inserts of completions with data
Write	UNC_I_CACHE_TOTAL_OCCUPANCY.MEM	A	Total IRP occupancy of inbound read and write requests to coherent memory.
	UNC_I_FAF_OCCUPANCY	B	Occupancy of the IRP FAF queue
	UNC_I_TRANSACTIONS.WR_PREF	C	Inbound write (fast path) requests to coherent memory, received by the IRP resulting in write ownership requests issued by IRP to the mesh.

The aliases listed in the table above represent the corresponding traffic in the formulae below.

Inbound PCIe Read Latency (Ns)	Inbound PCIe Write Latency (Ns)
(X) / (Y) / IIO Frequency (GHz)	((A) - (B)) / (C) / IRP Frequency (GHz)

CPU/IO Conflicts for Inbound Write Requests

The CPU/IO Conflicts metric gives the ratio of Inbound I/O write requests that experienced contention for a cache line, between the IO controller and another agent on the CPU (like the core or another IO controller). These conflicts are caused by simultaneous access to the same cache line. Under certain conditions, these access attempts can cause the IO controller to lose ownership of the cache line. This forces the IO controller to reacquire the ownership of this cache line. Such issues can occur in applications that use the polling communication model, resulting in suboptimal throughput and latency.

The table below describes the uncore performance events which are used to calculate these metrics.

Uncore Events	Alias	Description
UNC_I_MISC1.LOST_FWD	A	Snoop pulled away ownership before a write was committed

Uncore Events	Alias	Description
UNC_I_TRANSACTIONS.WR_PREF	B	The Inbound write (fast path) requests to coherent memory that were received by the IRP. These requests result in write ownership requests issued by IRP to the mesh.

This formula is used to calculate the CPU/IO Conflicts metric:

Inbound PCIe Write CPU/IO Conflicts (%) = (A) / (B) * 100%

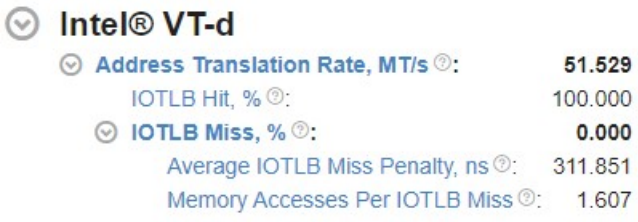
Intel® Virtualization Technology for Directed I/O (Intel® VT-d)

Intel® VT-d allows addresses in the incoming I/O device memory transactions to be remapped to different host addresses. To learn more about virtualization technology, see Intel® Virtualization Technology for Directed I/O Architecture Specification.

Intel VT-d enables the remapping of addresses for Inbound I/O requests. The Input and Output analysis type in Intel® VTune™ Profiler provides metrics related to Intel VT-d, that can help you analyze the efficiency with which workloads utilize Intel VT-d technology.

Understand Intel VT-d Metrics and Events

The IOMMU events from the IIO PMON block are used to calculate Intel VT-d metrics.



- **VT-d Address Translation Rate** : This is the top level Intel VT-d metric. This metric shows the total amount of I/O Translation Lookaside Buffer (IOTLB) Lookups in millions of translations per second. The IOTLB is an address translation cache in the remapping hardware unit. IOTLB caches the effective translations from virtual addresses used by devices, to host physical addresses. IOTLB lookups happen on address translation requests. The UNC_IIO_IOMMU0.FIRST_LOOKUPS event (which counts the first time a request looks up IOTLB) is used to calculate the Address Translation Rate metric using the following formula:

  Address Translation Rate (MT/s) = UNC_IIO_IOMMU0.FIRST_LOOKUPS / 10^6 / seconds sampled
- **IOTLB Hit/Miss Metrics :** These metrics reflect the ratios of address translation requests that hit or miss the IOTLB. The UNC_IIO_IOMMU0.CTXT_CACHE_LOOKUPS uncore event is used to calculate the OTLB Miss ratio. This event counts each time a transaction looks up root context cache. The formulae for Hit/ Miss ratios are:

  IOTLB Miss (%) = UNC_IIO_IOMMU0.CTXT_CACHE_LOOKUPS / UNC_IIO_IOMMU0.FIRST_LOOKUPS * 100%

  IOTLB Hit (%) = 100% - IOTLB Miss

For the group of IOTLB misses, there are two relevant metrics:

- **Average IOTLB Miss Penalty Metric :** This metric shows the average amount of time spent on handling an IOTLB miss. This metric includes the time to look up the context cache, intermediate page table caches and page table reads (page walks) on a miss, which turn into memory read requests. The table below describes the uncore performance events which are used to calculate these metrics.

Uncore Events	Alias	Description
UNC_IIO_PWT_OCCUPANCY	A	This event indicates the number of page walks that are outstanding at any point in time.
UNC_IIO_IOMMU0.CTXT_CACHE_LOOKUPS	B	This event counts each time a transaction looks up the root context cache.

Here is the formula used to calculate this metric:

Average IOTLB Miss Penalty (ns) = (**A**) / (**B**) / IIO Frequency (GHz)

- **Memory Access Per IOTLB Miss Metric** : This metric shows the average number of memory read requests (page walks) per IOTLB miss. To calculate this metric, an additional event (UNC_IIO_TXN_REQ_OF_CPU.MEM_READ.IOMMU1) is used. This event counts the number of 64B cache line requests that were initiated by the I/O device, including reads and writes.

  This metric is calculated by the following formula:

  Memory Access Per IOTLB Miss = UNC_IIO_TXN_REQ_OF_CPU.MEM_READ.IOMMU1 / UNC_IIO_IOMMU0.CTXT_CACHE_LOOKUPS

DRAM and UPI Bandwidth Histogram

This histogram plots the system bandwidth (in this example, DRAM and UPI) used by a certain value on the X axis against time on the Y axis. The histogram provides thresholds to categorize bandwidth utilization as High, Medium, or Low. To set the threshold, move sliders at the bottom of the histogram. Here are examples for both bandwidth domains:

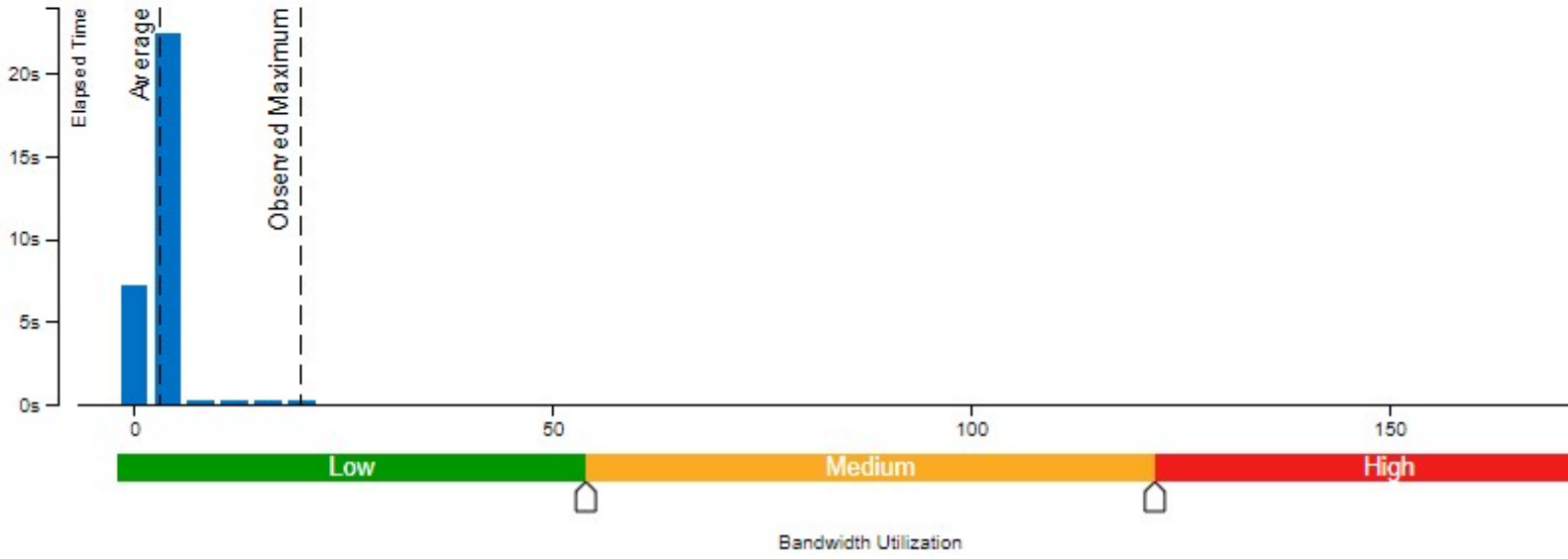
dwidth Utilization Histogram

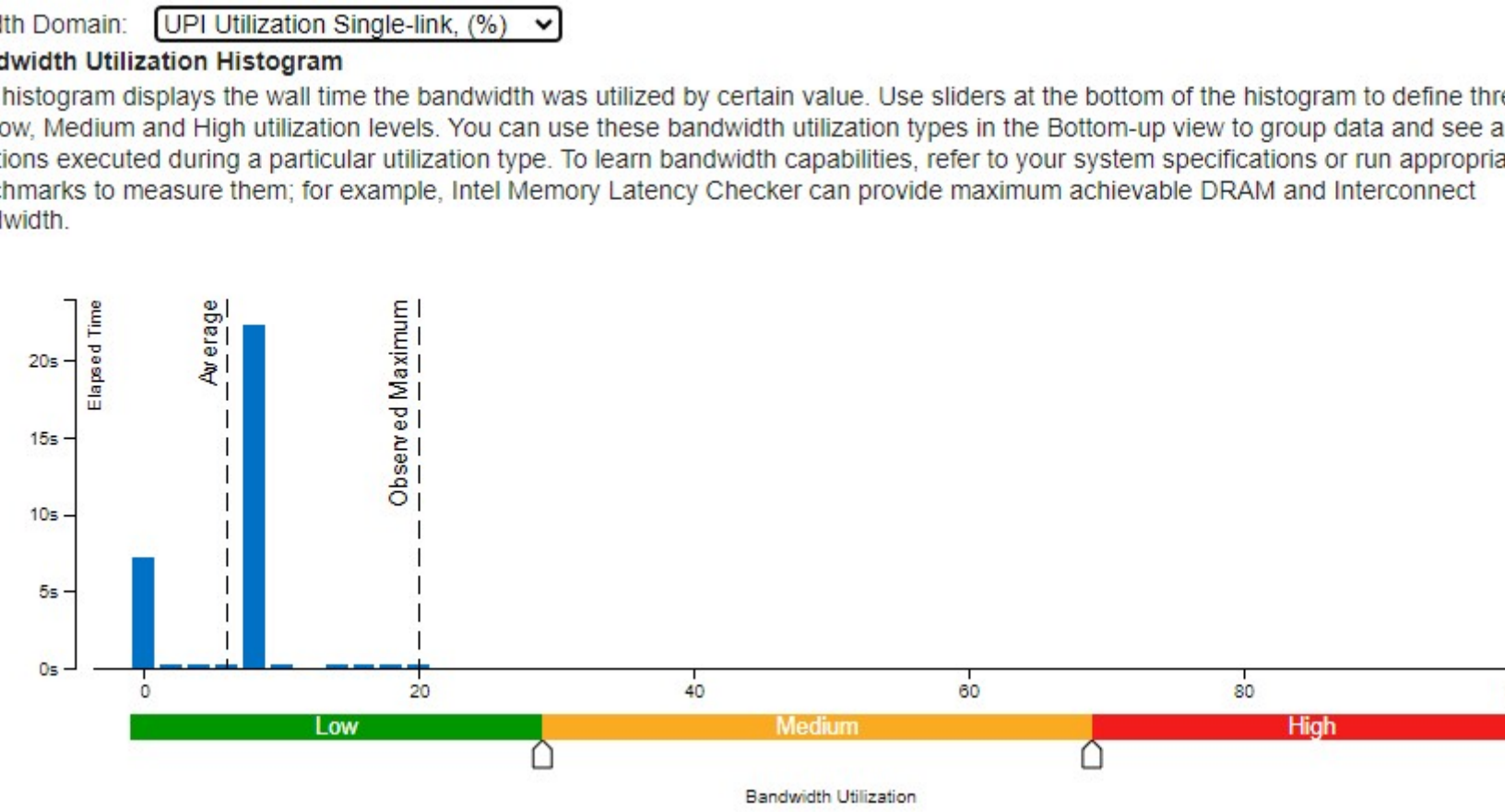
e bandwidth utilization over time using the histogram and identify memory objects or functions with maximum contribution to the high
idth utilization.

idth Domain: DRAM Single-Package, GB/sec

ndwidth Utilization Histogram

is histogram displays the wall time the bandwidth was utilized by certain value. Use sliders at the bottom of the histogram to define t
Low, Medium and High utilization levels. You can use these bandwidth utilization types in the Bottom-up view to group data and see
nctions executed during a particular utilization type. To learn bandwidth capabilities, refer to your system specifications or run approp
nchmarks to measure them; for example, Intel Memory Latency Checker can provide maximum achievable DRAM and Interconnect
ndwidth.





Understand DRAM Bandwidth and UPI Metrics and Events

DRAM Bandwidth and Utilization:

- **UNC_M_CAS_COUNT.RD** : This event counts the total number of DRAM Read Column Access Select (CAS) read commands on a per channel basis. CAS commands are issued to specify the address to read or write on the DRAM. This event increments for every 64 bytes. To calculate the total DRAM read data, use this formula:

  Data Read (GB) = UNC_M_CAS_COUNT.RD * 64 (bytes) / 10^9

  The average read bandwidth can be calculated by dividing on the sample time.

  Average Read Bandwidth (GB/s) = Data Read (GB) / Global Elapsed Time (s)
- **UNC_M_CAS_COUNT.WR** : This event counts the total number of DRAM Write CAS commands. This event increments for every 64 bytes. As for the read metric, the total DRAM write data is calculated this way:

  Data Written (GB) = UNC_M_CAS_COUNT.WR * 64 (bytes) / 10^9

  The average write bandwidth is calculated as:

  Average Write Bandwidth (GB/s) = Data Written (GB) / Global Elapsed Time (s)
- The DRAM average utilization can be calculated by dividing the average DRAM bandwidth by the maximum DRAM bandwidth. To calculate the maximum DRAM bandwidth, before you run the collection, select the **Evaluate max DRAM bandwidth** checkbox. A microbenchmark then starts before the collection to calculate the bandwidth.

  DRAM Utilization (%) = Average DRAM Bandwidth (GB/s) / Max Local Bandwidth (GB/s) * 100%

  Average DRAM Bandwidth (GB/s) = Average Write Bandwidth (GB/s) + Average Read Bandwidth (GB/s)

UPI Utilization:

The **UNC_UPI_TxL_FLITS.ALL_DATA** and **UNC_UPI_TxL_FLITS.NON_DATA** events indicate that valid flits sent for all data and non-data.

UPI Utilization (%) = (UNC_UPI_TxL_FLITS.ALL_DATA + UNC_UPI_TxL_FLITS.NON_DATA) / 3 / (UPI Active Clockticks * (5/6)) * 100%

UPI Active Clockticks = UNC_UPI_CLOCKTICKS – UNC_UPI_L1_POWER_CYCLES

Group Results into Views

The results of the Input and Output Analysis give us detailed information about I/O metrics. These metrics can be attributed to

- A group of PCIe devices
- PCIe metrics and performance uncore events
- I/O metrics over timeline

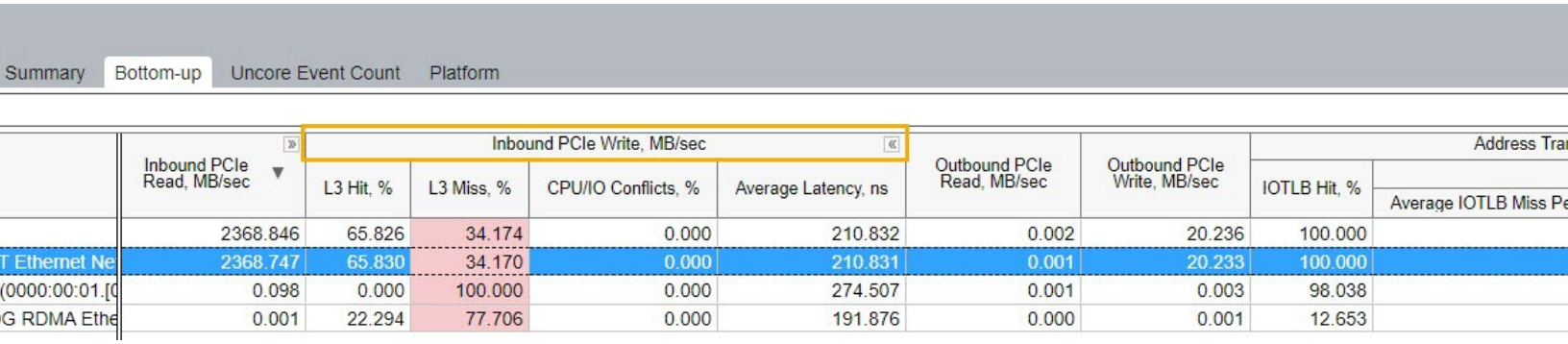
You can group these results in several views in these windows:

- Bottom-up window
- Uncore event count window
- Platform window

Bottom-up window:

When you use the **Package / IO Unit** grouping in the Bottom-up window, you get a per-device breakdown for:

- Inbound and Outbound Traffic
- Inbound request L3 hits and misses
- Average latencies
- CPU/IO Conflicts
- VT-d metrics



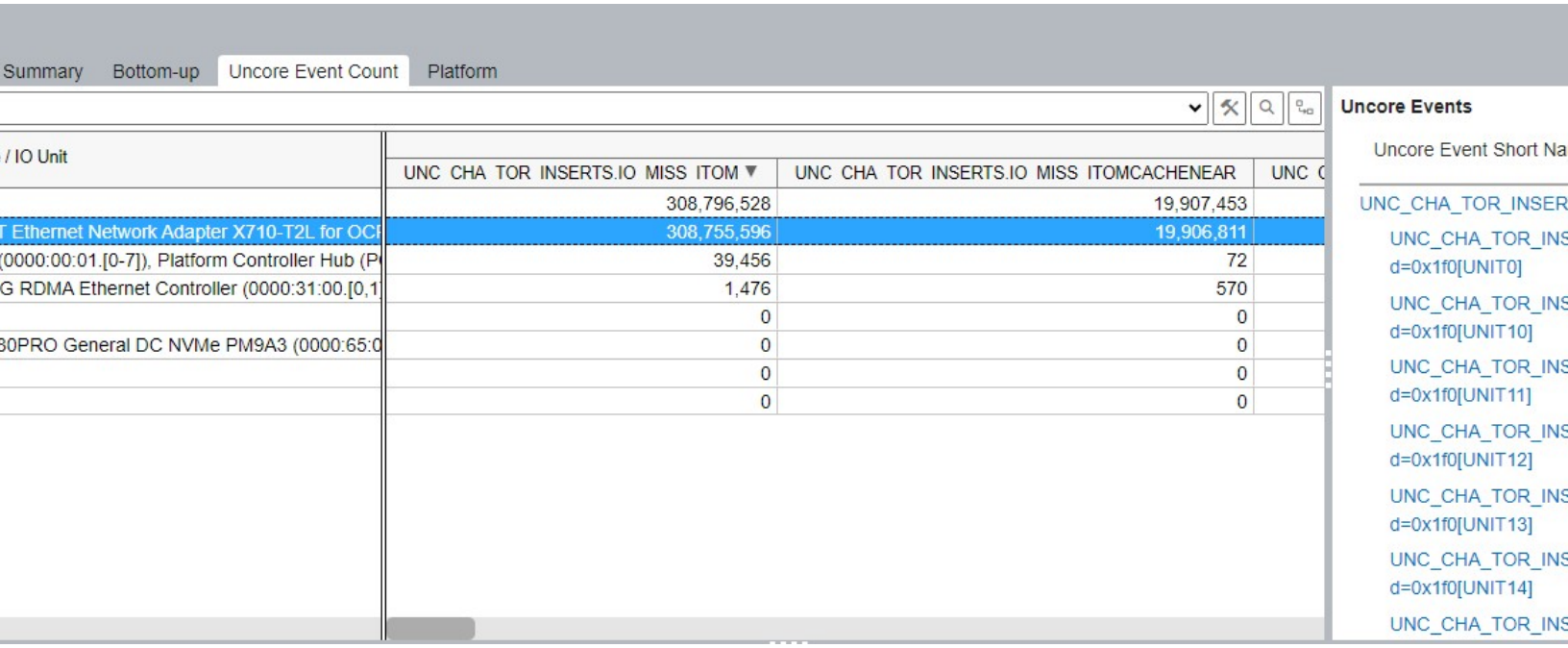
	Inbound PCIe Read, MB/sec	Inbound PCIe Write, MB/sec				Outbound PCIe Read, MB/sec	Outbound PCIe Write, MB/sec	Address Tra	
		L3 Hit, %	L3 Miss, %	CPU/IO Conflicts, %	Average Latency, ns			IOTLB Hit, %	Average IOTLB Miss Pe
	2368.846	65.826	34.174	0.000	210.832	0.002	20.236	100.000	
T Ethernet Ne	2368.747	65.830	34.170	0.000	210.831	0.001	20.233	100.000	
(0000:00:01.[0	0.098	0.000	100.000	0.000	274.507	0.001	0.003	98.038	
G RDMA Ethe	0.001	22.294	77.706	0.000	191.876	0.000	0.001	12.653	

To show or hide second level metrics, adjust the display option in the **Inbound PCIe Write** table.

Uncore Event Count Window:

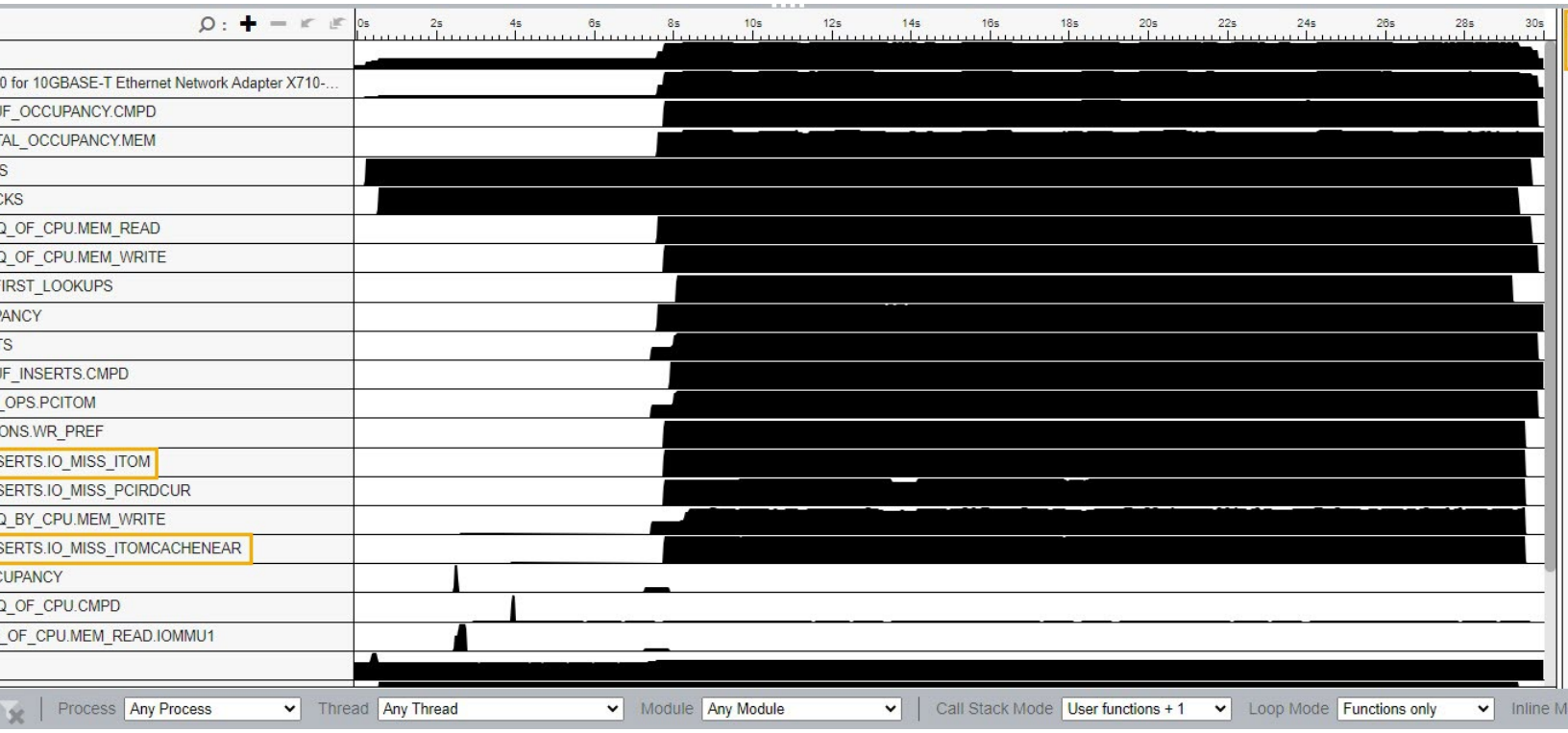
When use the **Package / IO Unit** grouping in the Uncore Event Count window, you can identify the count of uncore events per group of I/O device.

In this example, the **Uncore Events** section on the right contains performance events for the Ethernet Controller, which is the only device selected in the Package/IO Unit grouping.



You can visualize all uncore events in the timeline section.

To see uncore events that are attributed a group of PCIe devices on each socket, in the right corner of the timeline section, select the **Package/IO Unit/Uncore Event Short Name/Uncore Event Type** grouping.



Use the formulae described earlier in this recipe to understand the behavior of I/O devices.

For example, the **Inbound PCIe Write L3 Miss** metric depends on the number of ItoM requests that missed the L3 cache. In the figure above, there are 2 ItoM requests that missed the L3 cache.

You can visualize with raw data over time. Increase these events to understand better the time when the overall PCIe performance can be impacted.

To see the exact amount of hardware events at a certain instant, hover your mouse over the timeline.

To take a closer look at suspicious values, select the region and right click to zoom or filter in.

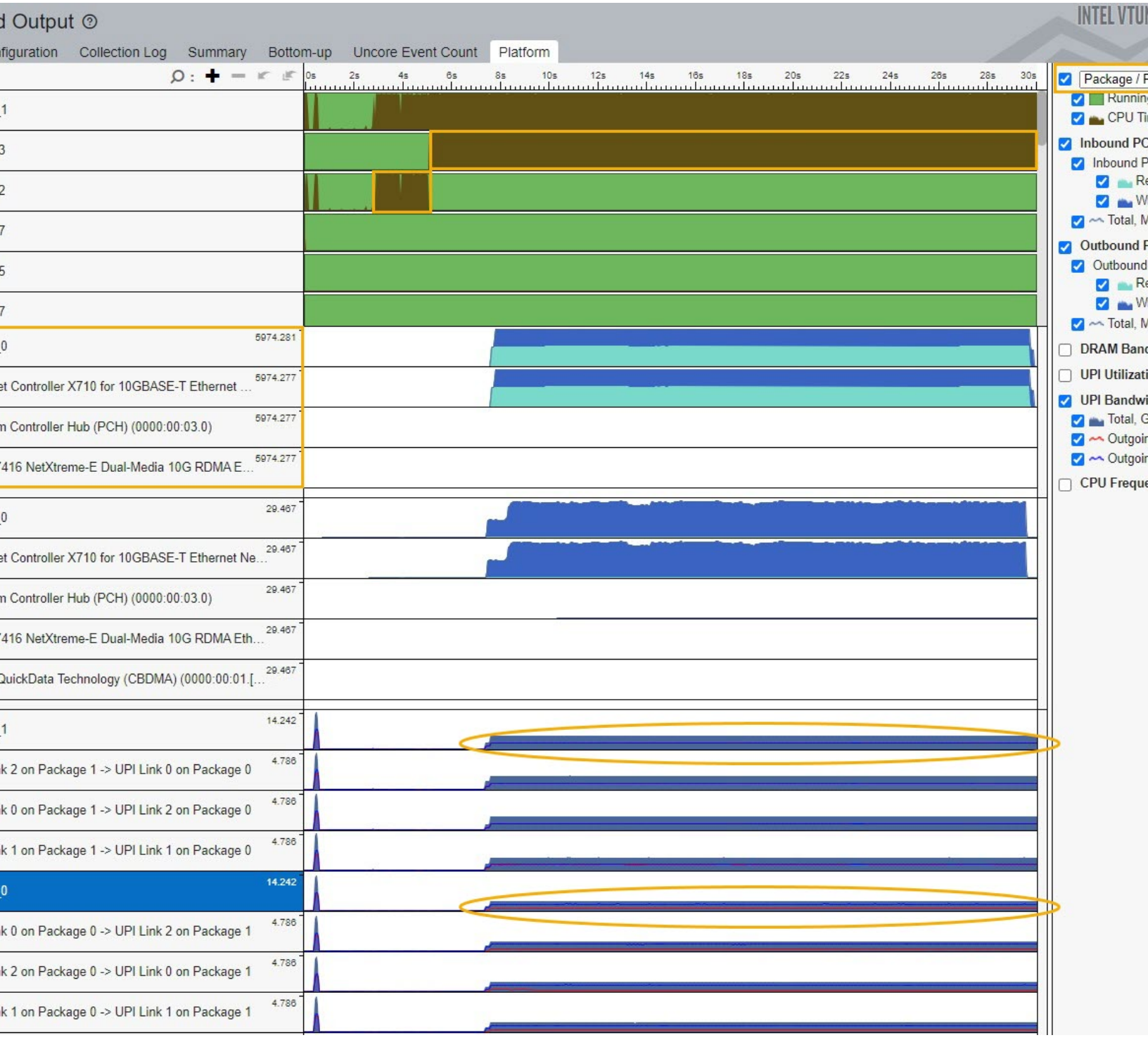
The I/O events that cannot be attributed to I/O stacks, like IMC and UPI events, are displayed in the **Unknown** device group.

Using the Uncore Event Count window can be extremely helpful when you run Custom Analyses which use custom uncore events.

Platform Window:

Use the Platform window to correlate PCIe traffic with the DRAM and UPI bandwidths. For example, when there is an L3 miss, data is read from the external DRAM.

Use the **Package/Physical Core** grouping in the Platform window to see the workload distribution on a core or thread. Use this information to further understand if there are issues related to NUMA.



Use the UPI traffic information provided by UPI links to identify if I/O performance issues are likely caused by inefficient remote socket accesses.

In the example above, the worker cores were set on remote socket. Setting the worker cores on the same socket as the PCIe device can reduce UPI traffic as well as L3 miss values.

In turn, these reductions can improve the overall PCIe performance.

You can discuss this recipe in the Analyzers forum.

See Also

Effective Utilization of Intel® Data Direct I/O Technology
DPDK Event Device Profiling
Core Utilization in DPDK Apps
PCIe Traffic in DPDK Apps
Utilizing the Intel® Xeon® Processor Scalable Family IIO Performance Monitoring Events
Lists of Intel Performance Monitoring Events

Top-down Microarchitecture Analysis Method

Use this recipe to know how an application is utilizing available hardware resources and how to make it take advantage of CPU microarchitectures. One way to obtain this knowledge is by using on-chip Performance Monitoring Units (PMUs).

NOTE This recipe is also available in Simplified Chinese.

PMUs are dedicated pieces of logic within a CPU core that count specific hardware events as they occur on the system. Examples of these events may be Cache Misses or Branch Mispredictions. These events can be observed and combined to create useful high-level metrics such as Cycles per Instruction (CPI).

A specific microarchitecture may make available hundreds of events through its PMU. However, it is frequently non-obvious to determine which events are useful in detecting and fixing specific performance issues. Often it requires an in-depth knowledge of both the microarchitecture design and PMU specifications to obtain useful information from raw event data. But you can benefit from using predefined events and metrics, and the top-down characterization method to convert the data into actionable information.

Explore the PMU analysis recipe to learn the methodology and how it is used in the Intel® VTune™ Profiler:

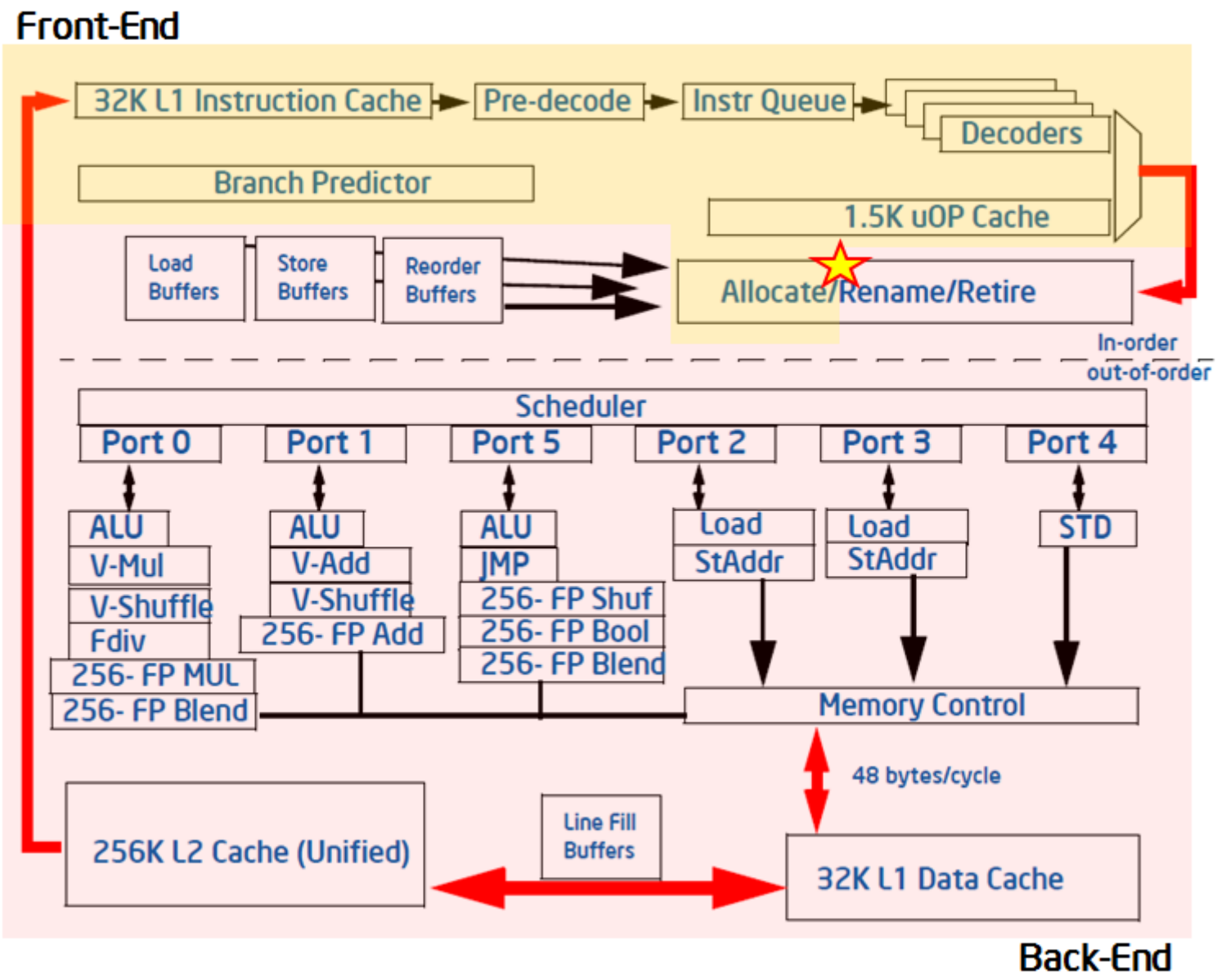
- INGREDIENTS:
  - Top-down Microarchitecture Analysis Method (TMA) overview
  - Top-Down Analysis Method with VTune Profiler
  - Microarchitectural Tuning Methodology
- DIRECTIONS:
  - Tune for the Back-End Bound Category
  - Tune for the Front-End Bound Category
  - Tune for the Bad Speculation Category
  - Tune for the Retiring Category
- Related Cookbook Recipes

Top-down Microarchitecture Analysis Method Overview

Modern CPUs employ pipelining as well as techniques like hardware threading, out-of-order execution and instruction-level parallelism to utilize resources as effectively as possible. In spite of this, some types of software patterns and algorithms still result in inefficiencies. For example, linked data structures are commonly used in software, but cause indirect addressing that can defeat hardware prefetchers. In many cases, this behavior can create bubbles of idleness in the pipeline while data is retrieved and there are no other instructions to execute. Linked data structures could be an appropriate solution to a software problem, but may result in inefficiencies. There are many other examples at the software level that have implications on the underlying CPU pipelines. The Top-down Microarchitecture Analysis Method based on the Top-Down Characterization methodology aims to provide an insight into whether you have made wise choices with your algorithms and data structures. See the *Intel® 64 and IA-32 Architectures Optimization Reference Manual, Appendix B.1* for more details on the Top-down Microarchitecture Analysis Method.

The Top-Down Characterization is a hierarchical organization of event-based metrics that identifies the dominant performance bottlenecks in an application. Its aim is to show, on average, how well the CPU's pipeline(s) were being utilized while running an application. Previous frameworks for interpreting events relied on accounting for CPU clockticks - determining what fraction of CPU's clockticks was spent on what type of operations (retrieving data due to L2 cache misses, for example). This framework instead is based on accounting for the pipeline's resources. To understand the Top-Down Characterization, explore a few microarchitectural concepts below, at a high level. Many of the details of the microarchitecture are abstracted in this framework, enabling you to use and understand it without being a hardware expert.

The pipeline of a modern high-performance CPU is quite complex. In the simplified view blow, the pipeline is divided conceptually into two halves, the Front-end and the Back-end. The Front-end is responsible for fetching the program code represented in architectural instructions and decoding them into one or more low-level hardware operations called *micro-ops* (uOps). The uOps are then fed to the Back-end in a process called *allocation*. Once allocated, the Back-end is responsible for monitoring when uOp's data operands are available and executing the uOp in an available execution unit. The completion of a uOp's execution is called *retirement*, and is where results of the uOp are committed to the architectural state (CPU registers or written back to memory). Usually, most uOps pass completely through the pipeline and retire, but sometimes speculatively fetched uOps may get cancelled before retirement – like in the case of mis-predicted branches.

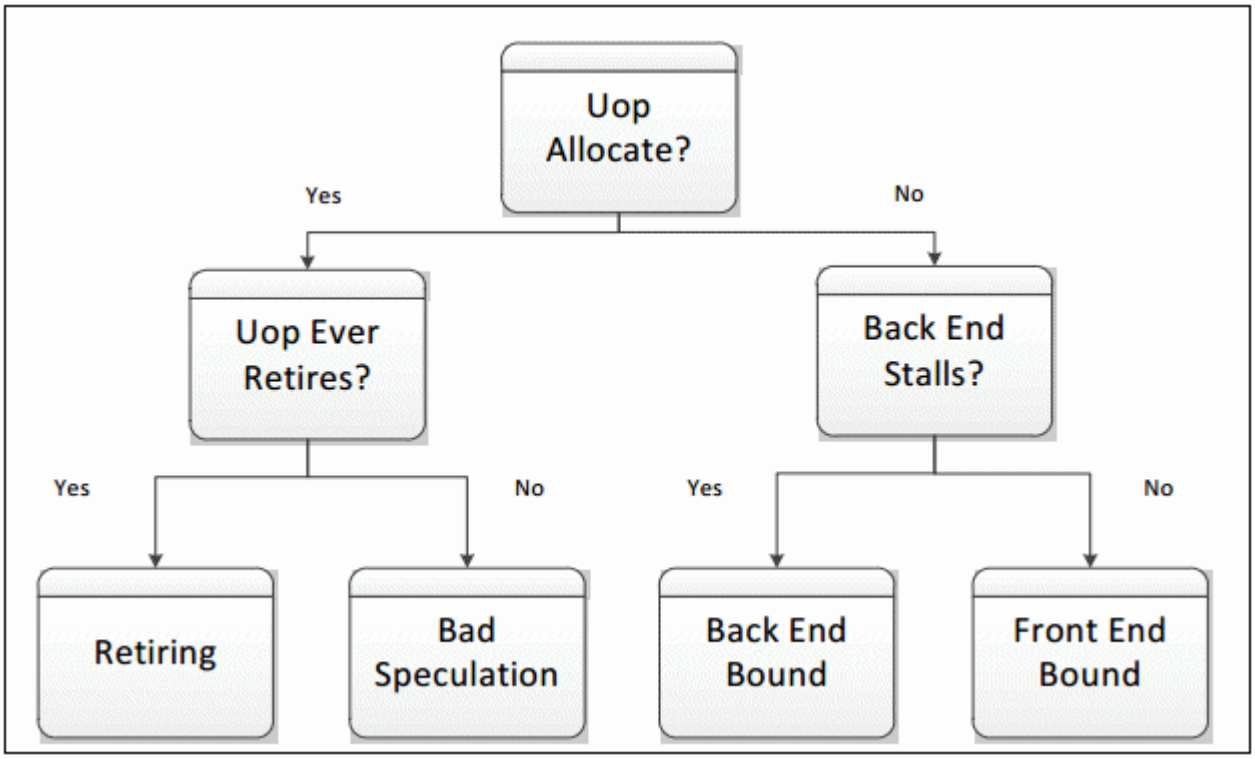


The Front-end of the pipeline on recent Intel microarchitectures can allocate four uOps per cycle, while the Back-end can retire four uOps per cycle. From these capabilities the abstract concept of a pipeline slot can be derived. A pipeline slot represents the hardware resources needed to process one uOp. The Top-Down Characterization assumes that for each CPU core, on each clock cycle, there are four pipeline slots available. It then uses specially designed PMU events to measure how well those pipeline slots were utilized. The status of the pipeline slots is taken at the allocation point (marked with a star in the figure above), where uOps leave the Front-end for the Back-end. Each pipeline slot available during an application's runtime will be classified into one of four categories based on the simplified pipeline view described above.

During any cycle, a pipeline slot can either be empty or filled with a uOp. If a slot is empty during one clock cycle, this is attributed to a stall. The next step needed to classify this pipeline slot is to determine whether the Front-end or the Back-end portion of the pipeline caused the stall. This is done using designated PMU events and formulas. The goal of the Top-Down Characterization is to identify dominant bottlenecks, hence the attribution of the stall to either the Front- or Back-end is a critical point of consideration. Generally, if the stall is caused by the Front-end's inability to fill the slot with a uOp, it will be classified as a Front-End Bound slot for this cycle, meaning, the performance was limited by some bottleneck under the Front-End Bound category. In the case where the Front-end has a uOp ready but cannot deliver it because the Back-end is not

ready to handle it, the empty pipeline slot will be classified as Back-End Bound. Back-end stalls are generally caused by the Back-end running out of some resource, for example, load buffers. However, if both the Front-end and the Back-end are stalled, then the slot will be classified as Back-End Bound. This is because, in that case, fixing the stall in the Front-end would most likely not help an application's performance. The Back-end is the blocking bottleneck, and it would need to be removed first before fixing issues in the Front-end would have any effect.

If the processor is not stalled then a pipeline slot will be filled with a uOp at the allocation point. In this case, the determining factor for how to classify the slot is whether the uOp eventually retires. If it does retire, the slot is classified as Retiring. If it does not, either because of incorrect branch predictions by the Front-end or a clearing event like a pipeline flush due to Self-Modifying-Code, the slot will be classified as Bad Speculation. These four categories make up the top level of the Top-Down Characterization. To characterize an application, each pipeline slot is classified into exactly one of these four categories:



The distribution of pipeline slots in these four categories is very useful. Although metrics based on events have been possible for many years, before this characterization there was no approach for identifying which possible performance issues were the most impactful. When performance metrics are placed into this framework, you can see which issues need to be tackled first. The events needed to classify pipeline slots into the four categories are available beginning with Intel® microarchitecture code name Sandy Bridge – which is used in the 2nd Generation Intel Core processor family and the Intel Xeon® processor E5 family. Subsequent microarchitectures may allow further decomposition of these high-level categories into more detailed performance metrics.

Top-Down Analysis Method with VTune Profiler

NOTE

- Starting with the 2020 release, Intel® VTune™ Amplifier has been renamed to Intel® VTune™ Profiler.
- Most recipes in the Intel® VTune™ Profiler Performance Analysis Cookbook are flexible. You can apply them to different versions of Intel® VTune™ Profiler. In some cases, minor adjustments may be required.
- Get the latest version of Intel® VTune™ Profiler:
  - From the Intel® VTune™ Profiler product page.
  - Download the latest standalone package from the Intel® oneAPI standalone components page.

Intel® VTune™ Profiler provides a Microarchitecture Exploration analysis type that is pre-configured to collect the events defined in the Top-Down Characterization starting with the Intel microarchitecture code name Ivy Bridge. Microarchitecture Exploration also collects the events required to calculate many other useful performance metrics. The results of a Microarchitecture Exploration analysis are displayed by default in the Microarchitecture Exploration viewpoint .

Microarchitecture Exploration results are displayed in hierarchical columns to reinforce the top-down nature of the characterization. The **Summary** window gives the percentage of pipeline slots in each category for the whole application. You can explore results in multiple ways. The most common way to explore results is to view metrics at the function level:

ing: Function / Call Stack

ion / Call Stack	Instructions Retired	CPI Rate	Front-End Bound	Bad Speculation	Back-End Bound		Ret
					Memory Bound	Core Bound	
e_out_impl	62,556,093,834	1.261	2.2%	7.4%	64.2%	8.4%	
esh_potential	17,836,026,754	3.589	3.0%	8.1%	73.2%	9.6%	
nal_bea_mpp	38,108,057,162	1.393	5.6%	24.3%	34.4%	21.0%	
late_tree	4,092,006,138	3.373	7.2%	11.5%	62.3%	11.8%	
_basket	12,246,018,369	1.037	20.7%	50.4%	3.8%	4.6%	
nal_iminus	5,324,007,986	2.148	7.1%	6.7%	55.0%	20.5%	
nal_net_simple:	266,000,399	2.466	17.4%	43.4%	13.1%	13.1%	

For each function, the fraction of pipeline slots in each category is shown. For example, the `price_out_impl` function, selected above, had 2.2% of its pipeline slots in the Front-End Bound category, 7.4% in Bad Speculation, 64.2% in Memory Bound, 8.4% in Core Bound, and 17.8% in the Retiring category. Each category can be expanded to view metrics underneath that category. Automatic highlighting is used to draw your attention to potential problem areas, in this case, to the high percentage of Memory Bound pipeline slots for `price_out_impl`.

Microarchitectural Tuning Methodology

When doing any performance tuning, it is important to focus on the top hotspots of the application. *Hotspots* are the functions taking the most CPU time. Focus on these spots will ensure that optimizations impact the overall application performance. VTune Profiler has a Hotspots analysis with two specific collection modes: user-mode sampling and hardware event-based sampling. Within the Microarchitecture Exploration viewpoint, hotspots can be identified by determining the functions or modules with the highest Clockticks

event counts, which measures the number of CPU clockticks. To obtain maximum benefit from microarchitectural tuning, ensure that algorithmic optimizations such as adding parallelism have already been applied. Generally system tuning is performed first, then application-level algorithm tuning, then architectural and microarchitectural tuning. This process is also referred to as "Top-Down", as in the Top-Down software tuning methodology. It, as well as other important aspects of performance tuning like workload selection, are described in the De-Mystifying Software Performance Optimization article.

1. Select a hotspot function (one with a large percentage of the application's total clockticks).
2. Evaluate the efficiency of that hotspot using the Top-Down Method and the guidelines given below.
3. If inefficient, drill down the category representing the primary bottleneck, and use the next levels of sub-bottlenecks to identify causes.
4. Optimize the issue(s). VTune Profiler tuning guides contain specific tuning suggestions for many of the underlying performance issues in each category.
5. Repeat until all significant hotspots have been evaluated.

VTune Profiler automatically highlights metric values in the GUI if they are outside a predefined threshold and occur in a hotspot. VTune Profiler classifies a function as a hotspot if greater than 5% of the total clockticks for an application accrued within it. Determining whether a given fraction of pipeline slots in a particular category constitutes a bottleneck can be workload-dependent, but some general guidelines are provided in the table below:

	Expected Range of Pipeline Slots in This Category, for a Hotspot in a Well-Tuned:		
Category	**Client/Desktop Application**	**Server/Database/ Distributed application**	**High Performance Computing (HPC) application**
Retiring	20-50%	10-30%	30-70%
Back-End Bound	20-40%	20-60%	20-40%
Front-End Bound	5-10%	10-25%	5-10%
Bad Speculation	5-10%	5-10%	1-5%

These thresholds are based on analysis of some workloads in labs at Intel. If the fraction of time spent in a category (other than Retiring) for a hotspot is on the high end or greater than the range indicated, an investigation might be useful. If this is true for more than one category, the category with the highest fraction of time should be investigated first. Note that it is expected that hotspots will have some fraction of time spent in each category, and that values within the normal range below may not indicate a problem.

The important thing to realize about the Top-Down Method is that you do not need to spend time optimizing issues in a category that is not identified as a bottleneck - doing so will likely not lead to a significant performance improvement.

Tune for the Back-End Bound Category

The majority of un-tuned applications will be Back-End Bound. Resolving Back-end issues is often about resolving sources of latency, which cause retirement to take longer than necessary. On the Intel microarchitecture code name Sandy Bridge, VTune Profiler has Back-End Bound metrics to find the sources of high latency. For example, the LLC Miss (Last-Level Cache Miss) metric identifies regions of code that need to access DRAM for data, and the Split Loads and Split Stores metrics point out memory access patterns that can harm performance. For more details on Intel microarchitecture code name Sandy Bridge metrics, see the *Tuning Guide*. Starting with Intel microarchitecture code name Ivy Bridge (which is used in the 3rd Generation Intel Core processor family), events are available to breakdown the Back-End Bound classification

into Memory Bound and Core Bound sub-metrics. A metric beneath the top 4 categories may use a domain other than the pipeline slots domain. Each metric will use the most appropriate domain based on underlying PMU events. For more details see the documentation for each metric or category.

The Memory and Core Bound sub-metrics are determined using events corresponding to the utilization of the execution units - as opposed to the allocation stage used in the top-level classifications. Therefore, the sum of these metrics will not necessarily match the Back-End Bound ratio determined at the top-level (though they correlate well).

Stalls in the Memory Bound category have causes related to the memory subsystem. For example, cache misses and memory accesses can cause Memory Bound stalls. Core Bound stalls are caused by a less-than-optimal use of the available execution units in the CPU during each cycle. For example, several multi-cycle divide instructions in a row competing for the divide units could cause Core Bound stalls. For this breakdown, slots are only classified as Core Bound if they are stalled AND there are no uncompleted memory accesses. For example, if there are pending loads, the cycle is classified as Memory Bound because the execution units are being starved while the loads have not returned data yet. PMU events were designed into the hardware to specifically allow this type of breakdown, which helps identify the true bottleneck in an application. The majority of Back-End Bound issues will fall into the Memory Bound category.

Most of the metrics under the Memory Bound category identify which level of the memory hierarchy from the L1 cache through the memory is the bottleneck. Again, the events used for this determination were carefully designed. Once the Back-end is stalled the metrics try to attribute the stalls of pending loads to a particular level of cache or to in-flight stores. If a hotspot is bound at a given level, it means that most of its data is being retrieved from that cache- or memory-level. Optimizations should focus on moving data closer to the core. Store Bound is also called out as a sub-category, which can indicate dependancies - such as when loads in the pipeline depend on prior stores. Under each of these categories, there are metrics that can identify specific application behaviors resulting in Memory Bound execution. For example, Loads Blocked by Store Forwarding and 4k Aliasing are metrics that flag behaviors that can cause an application to be L1 Bound.

Core Bound stalls are typically less common within Back-End Bound. These can occur when available computing resources are not sufficiently utilized and/or used without significant memory requirements. For example, a tight loop doing Floating Point (FP) arithmetic calculations on data that fits within cache. VTune Profiler provides some metrics to detect behaviors in this category. For example the Divider metric identifies cycles when divider hardware is heavily used and the Port Utilization metric identifies competition for discrete execution units.

on / Call Stack	Back-End Bound						
	Memory Bound					Core Bound	
	L1 Bound	L2 Bound	L3 Bound	DRAM Bound	Store Bound	Divider	Port Utiliza
e_out_impl	4.7%	4.7%	4.4%	44.9%	0.0%	0.0%	
esh_potential	1.7%	0.0%	4.6%	75.9%	0.0%	0.0%	
al_bea_mpp	0.0%	1.1%	11.7%	26.0%	0.0%	0.0%	
ate_tree	33.3%	16.5%	22.9%	4.1%	0.0%	0.0%	
_basket	11.4%	0.0%	0.0%	0.0%	0.0%	0.0%	

NOTE
Grayed out metric values indicate that the data collected for this metric is unreliable. This may happen, for example, if the number of samples collected for PMU events is too low. You may either ignore this data, or rerun the collection with the data collection time, sampling interval, or workload increased.

Tune for the Front-End Bound Category

The Front-End Bound category covers several other types of pipeline stalls. It is less common for the Front-end portion of the pipelines to become the application's bottleneck; however there are cases where the Front-end can contribute in a significant manner to machine stalls. For example, JITed code and interpreted code can cause Front-end stalls because the instruction stream is dynamically created without the benefit of compiler code layout in advance. Improving performance in the Front-End Bound category will generally relate to code layout (co-locating hot code) and compiler techniques. For example, branchy code or code with a large footprint may highlight the Front-End Bound category. Techniques like code size optimization and compiler profile-guided optimization (PGO) are likely to reduce stalls in many cases.

The Top-Down Method on Intel microarchitecture code name Ivy Bridge and beyond divides Front-End Bound stalls into 2 categories, Front-End Latency and Front-End Bandwidth. The Front-End Latency metric reports cycles in which no uops were issued by the Front-end in a cycle, while the Back-end was ready to consume them. Recall that the Front-end cluster can issue up to 4 uops per cycle. The Front-End Bandwidth metric reports cycles in which less than 4 uops were issued, representing an inefficient use of the Front-end's capability. Further metrics are identified below each of the categories.

Branch mispredictions, which are mostly accounted for in the Bad Speculation category, could also lead to inefficiencies in the Front-end as denoted by the Branch Resteers bottleneck metric underneath Front-End Latency starting in the Intel microarchitecture code name Ivy Bridge.

on / Call Stack	Front-End Bound							
	Front-End Latency						Front-End Ban	
	ICache Misses	ITLB ...	Branch Resteers	DSB ...	Length...	MS ...	Front-End Band...	Front-
e_out_impl	0.0%	0.0%	0.9%	0.0%	0.0%	0.1%	3.8%	
esh_potential	0.0%	0.0%	1.4%	0.2%	0.0%	0.1%	5.7%	
al_bea_mpp	0.0%	0.0%	2.9%	0.1%	0.0%	0.0%	12.2%	
ate_tree	0.0%	0.0%	2.2%	0.6%	0.0%	0.0%	7.2%	
_basket	0.0%	0.0%	11.4%	2.1%	0.0%	0.0%	31.4%	
al_iminus	0.0%	0.0%	1.0%	0.7%	0.0%	0.0%	9.0%	

VTune Profiler lists metrics that may identify causes of Front-End Bound code. If any of these categories shows up significantly in the results, dig deeper into the metrics to determine the causes and how to correct them. For example, the ITLB Overhead (Instruction Translation Lookaside Buffer Overhead) and ICache Miss (Instruction Cache miss) metrics may point out areas suffering from Front-End Bound execution. For tuning suggestions see the VTune Profiler tuning guides.

Tune for the Bad Speculation Category

The third top-level category, Bad Speculation, denotes when the pipeline is busy fetching and executing non-useful operations. Bad Speculation pipeline slots are slots wasted by issued uops that never retired or stalled while the machine recovers from an incorrect speculation. Bad Speculation is caused by branch mispredictions and machine clears and less commonly by cases like Self-Modifying-Code. Bad Speculation can be reduced through compiler techniques such as Profile-Guided Optimization (PGO), avoiding indirect branches, and eliminating error conditions that cause machine clears. Correcting Bad Speculation issues may also help decrease the number of Front-End Bound stalls. For specific tuning techniques refer to the VTune Profiler tuning guide appropriate for your microarchitecture.

Function / Call Stack	Bad Speculation		Back-End Bound
	Branch Mispredict	Machine Clears	
price_out_impl	7.4%	0.0%	72.7%
refresh_potential	8.0%	0.1%	82.8%
primal_bea_mpp	24.3%	0.0%	55.5%
update_tree	11.5%	0.0%	74.1%
sort_basket	50.4%	0.0%	8.4%
primal_iminus	6.7%	0.0%	75.4%
primal_net_simple	0.0%	43.4%	26.1%

Tune for the Retiring Category

The last category at the top level is Retiring. It denotes when the pipeline is busy with typically useful operations. Ideally an application would have as many slots classified in this category as possible. However, even regions of code with a large portion of their pipeline slots retiring may have room for improvement. One performance issue that will fall under the retiring category is heavy use of the micro-sequencer, which assists the Front-end by generating a long stream of uops to address a particular condition. In this case, although there are many retiring uops, some of them could have been avoided. For example, FP Assists that apply in the event of Denormals can often be reduced through compiler options (like DAZ or FTZ). Code generation choices can also help mitigate these issues - for more details see the VTune Profiler tuning guides. In the Intel microarchitecture code name Sandy Bridge, Assists are identified as a metric under the Retiring category. In the Intel microarchitecture code name Ivy Bridge and beyond, the pipeline slots in the ideal category of retirement are broken into a sub-category called General Retirement, and Microcode Sequencer uops are identified separately.

Function / Call Stack	Retiring				
	General Retirement				Microcode Sequencer
	FP Arithmetic			Other	Assists
	FP x87	FP Scalar	FP Vector		
price_out_impl	0.0%	0.0%	0.0%	100.0%	0.0%
refresh_potential	0.0%	0.0%	0.0%	100.0%	0.0%
primal_bea_mpp	0.0%	0.0%	0.0%	100.0%	0.0%
update_tree	0.0%	0.0%	0.0%	100.0%	0.0%
sort_basket	0.0%	0.0%	0.0%	100.0%	0.0%

If not already done, algorithmic tuning techniques like parallelization and vectorization can help improve the performance of code regions that fall into the retiring category.

Conclusion

The Top-Down Method and its availability in VTune Profiler represent a new direction for performance tuning using PMUs. Developer time invested in becoming familiar with this characterization will be worth the effort, since support for it is designed into recent PMUs and, where possible, the hierarchy is further expanded on future Intel microarchitectures. For example, the characterization was significantly expanded between Intel microarchitecture code name Sandy Bridge and Intel microarchitecture code name Ivy Bridge.

The goal of the Top-Down Method is to identify the dominant bottlenecks in an application performance. The goal of Microarchitecture Exploration analysis and visualization features in VTune Profiler is to give you actionable information for improving your applications. Together, these capabilities can significantly boost not only application performance, but also the productivity of your optimizations.

Related Cookbook Recipes

- Tuning Recipe: False Sharing
- Tuning Recipe: Frequent DRAM Accesses
- Tuning Recipe: Poor Port Utilization
- Tuning Recipe: Instruction Cache Misses

See Also

Microarchitecture Pipe

Microarchitecture Exploration View

Tuning Guides and Performance Analysis Papers
Clockticks vs Pipeline Slots Based Metrics

OpenMP* Code Analysis Method

This recipe introduces a flow to analyze CPU utilization of your OpenMP or hybrid OpenMP-MPI application and identify causes of possible inefficiencies.*

Content Expert: Rupak Roy

OpenMP is a fork-join parallel model, which starts with an OpenMP program running with a single master serial-code thread. When a parallel region is encountered, that thread forks into multiple threads, which then execute the parallel region. At the end of the parallel region, the threads join at a barrier, and then the master thread continues executing serial code. It is possible to write an OpenMP program more like an MPI program, where the master thread immediately forks to a parallel region and constructs such as *barrier* and *single* are used for work coordination. But it is far more common for an OpenMP program to consist of a sequence of parallel regions interspersed with serial code.

Ideally, parallelized applications have working threads doing useful work from the beginning to the end of execution, utilizing 100% of available CPU core processing time. In real life, useful CPU utilization is likely to be less when working threads are waiting, either actively spinning (for performance, expecting to have a short wait) or waiting passively, not consuming CPU. There are several major reasons why working threads wait, not doing useful work:

- **Execution of serial portions (outside of any parallel region)**: When the master thread is executing a serial region, the worker threads are in the OpenMP runtime waiting for the next parallel region.
- **Load imbalance:** When a thread finishes its part of workload in a parallel region, it waits at a barrier for the other threads to finish.
- **Not enough parallel work:** The number of loop iterations is less than the number of working threads so several threads from the team are waiting at the barrier not doing useful work at all.
- **Synchronization on locks:** When synchronization objects are used inside a parallel region, threads can wait on a lock release, contending with other threads for a shared resource.

Use VTune Profiler to understand how an application utilizes available CPUs and identify causes of CPU underutilization.

To analyze an OpenMP application with VTune Profiler:

1. Compile your code with recommended options.
2. Configure OpenMP regions analysis.
3. Explore application-level OpenMP metrics.
4. Identify serial code.
5. Estimate potential gain.
6. Understand limitations.

Compile Your Code with Recommended Options

To enable parallel regions and source analysis during compilation, do the following:

- To analyze OpenMP parallel regions, make sure to compile and run your code with the Intel® oneAPI DPC +/C++ Compiler version 2023.2.0 (or newer). If an obsolete version of the OpenMP runtime libraries is detected, VTune Profiler provides a warning message. In this case the collection results may be incomplete.

  To access the newest OpenMP analysis options described in the documentation, make sure you always use the latest version of the Intel compiler.
- On Linux*, to analyze an OpenMP application compiled with GCC*, make sure the GCC OpenMP library (`libgomp.so`) contains symbol information. To verify, search for `libgomp.so` and use the `nm` command to check symbols, for example:

```
nm libgomp.so.1.0.0
```

  If the library does not contain any symbols, either install/compile a new library with symbols or generate debug information for the library. For example, on Fedora* you can install GCC debug information from the `yum` repository:

```
yum install gcc-debuginfo.x86_64
```

Configure OpenMP Analysis

To enable OpenMP analysis for your target:

1. Click the

   (standalone GUI)/

   (Visual Studio IDE)**Configure Analysis** button on the Intel® VTune™ Profiler toolbar.

   The **Configure Analysis** window opens.
2. From **HOW** pane, click the

   Browse button and select an analysis type that supports OpenMP analysis: Threading, HPC Performance Characterization, Memory Access, or any Custom Analysis type.
3. Select the **Analyze OpenMP regions** option, if it is not pre-selected (see the **Details** section to confirm).
4. Click the

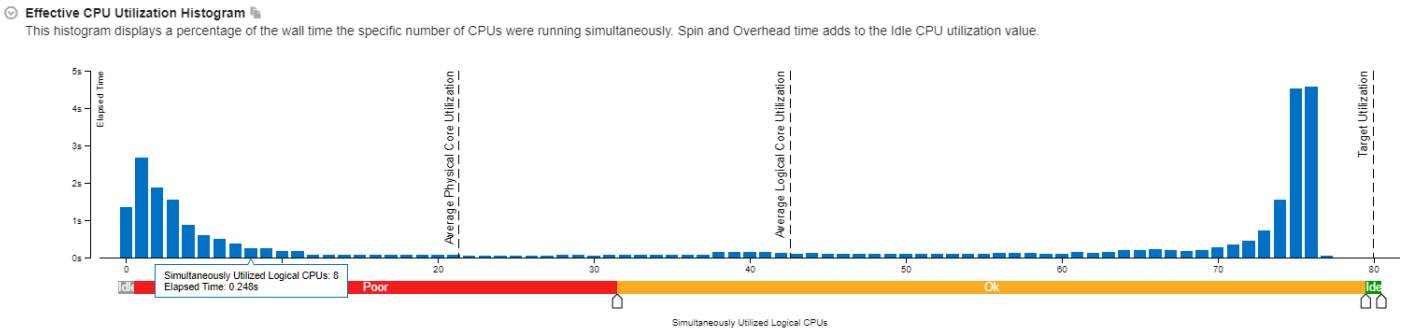
   Start button to run the analysis.

The OpenMP runtime library in the Intel Composer provides special markers for applications running under profiling that can be used by the VTune Profiler to decipher the statistics of OpenMP parallel regions and distinguish serial parts of the application code.

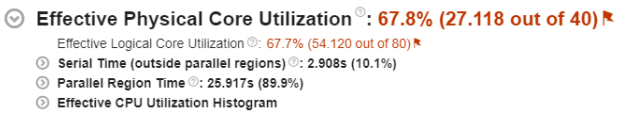
Explore Application-Level OpenMP Metrics

Start your analysis with understanding the CPU utilization of your analysis target. If you are using the HPC Performance Characterization viewpoint, focus on the **Effective Physical Core Utilization** section of the Summary window that shows the number of used logical and physical cores and estimates the efficiency (in percent) of this CPU utilization. Poor core utilization is flagged as a performance issue.

Other viewpoints provide the **CPU Utilization Histogram** that displays the Elapsed time of your application, broken down by CPU utilization levels. The histogram shows only useful utilization so the CPU cycles that were spent by the application burning CPU in spin loops (active wait) are not counted. You can adjust sliders from the default levels if you intentionally use a number of OpenMP working threads less than the number of available hardware threads.



If the bars are close to Ideal utilization, you might need to look deeper, at algorithm or microarchitecture tuning opportunities, to find performance improvements. If not, explore the **OpenMP Analysis** section of the **Summary** window for inefficiencies in parallelization of the application:



This section of the **Summary** window shows the Collection Time as well as the duration of serial (outside of any parallel region) and parallel portions of the program. If the serial portion is significant, consider options to minimize serial execution, either by introducing more parallelism or by doing algorithm or microarchitecture tuning for sections that seem unavoidably serial. For high thread-count machines, serial sections have a severe negative impact on potential scaling (Amdahl's Law) and should be minimized as much as possible.

Identify Serial Code

To analyze the serially executed code, expand the **Serial Time (outside parallel regions)** section of the **Summary** window and review the **Top Serial Hotspots (outside parallel regions)**. You can click a function name to be taken to that function in the **Bottom-up** window for more detail.

Effective Physical Core Utilization: 53.4% (21.360 out of 40)

Effective Logical Core Utilization: 53.3% (42.605 out of 80)

Serial Time (outside parallel regions): 2.284s (7.8%)

Top Serial Hotspots (outside parallel regions)

This section lists the loops and functions executed serially in the master thread outside of any OpenMP region and consuming the most CPU time. Improve overall application performance by optimizing or parallelizing these hotspot functions. Since the Serial Time metric includes the Wait time of the master thread, it may significantly exceed the aggregated CPU time in the table.

Function	Module	Serial CPU Time
[Loop at line 462 in checkSTREAMresults]	stream_avx512.bin	1.423s
func@0x86010	libittnotify_collector.so	0.005s

Estimate Potential Gain

To estimate the efficiency of CPU utilization in the parallel part of the code, use the Potential Gain metric. This metric estimates the difference in the Elapsed time between the actual measurement and an idealized execution of parallel regions, assuming perfectly balanced threads and zero overhead of the OpenMP runtime on work arrangement. Use this data to understand the maximum time that you may save by improving parallel execution.

The **Summary** window provides a detailed table listing the top five parallel regions with the highest Potential Gain metric values.

Top OpenMP Regions by Potential Gain

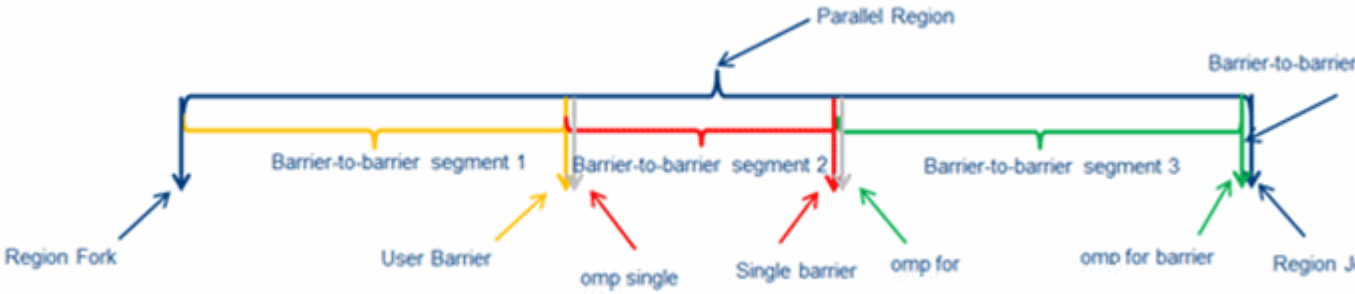
This section lists OpenMP regions with the highest potential for performance improvement. The Potential Gain metric shows the elapsed time that could be saved if the region was optimized to have no load imbalance assuming no runtime overhead.

OpenMP Region	OpenMP Potential Gain	(%)	OpenMP Region Time
mainompparallel:80@/nfs/site/home/rupakroy/openmp_analysis/memory-bandwidth-benchmarks/stream.c:343:344	1.361s	4.7%	7.296s
mainompparallel:80@/nfs/site/home/rupakroy/openmp_analysis/memory-bandwidth-benchmarks/stream.c:333:334	1.161s	4.0%	7.111s
mainompparallel:80@/nfs/site/home/rupakroy/openmp_analysis/memory-bandwidth-benchmarks/stream.c:323:324	1.017s	3.5%	5.496s
mainompparallel:80@/nfs/site/home/rupakroy/openmp_analysis/memory-bandwidth-benchmarks/stream.c:313:314	0.874s	3.0%	5.336s
mainompparallel:80@/nfs/site/home/rupakroy/openmp_analysis/memory-bandwidth-benchmarks/stream.c:286:287	0.075s	0.3%	0.113s
[Others]			0.566s

If Potential Gain for a region is significant, you can go deeper and select the link on a region name to navigate to the **Bottom-up** window employing the **/OpenMP Region/OpenMP Barrier-to-Barrier Segment/..** dominant grouping that provides detailed analysis of inefficiency metrics like Imbalance by barriers.

Intel OpenMP runtime from Intel Parallel Studio instruments barriers for the VTune Profiler. VTune Profiler introduces a notion of *barrier-to-barrier* OpenMP region segment that spans from a region fork point or previous barrier to the barrier that defines the segment.

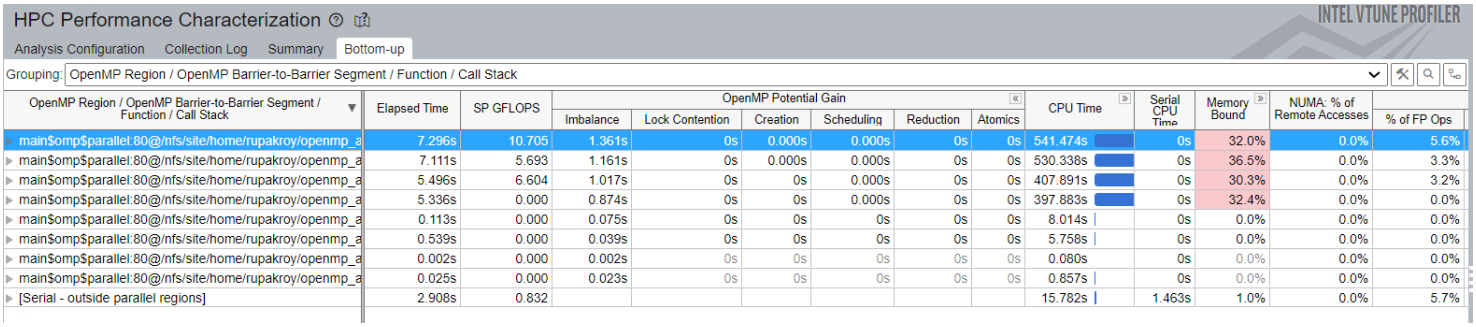
```
ma omp parallel

....
ma omp barrier
ma omp single

   ....

ma omp for

   ...
```



In the example above, there are four barrier-to-barrier segments defined as a `user` barrier, implicit `single` barrier, implicit `omp for` loop barrier and `region join` barrier.

For the cases when an OpenMP region contains multiple barriers either implicit with parallel loops or #pragma single sections, or explicit with user barriers, analyze the impact of a particular construct or a barrier to inefficiency metrics.

A barrier type is embedded to the segment name, for example: `loop`, `single`, `reduction`, and others. It also emits additional information for parallel loops with implicit barriers like loop scheduling, chunk size and min/max/average of the loop iteration counts that is useful to understand imbalance or scheduling overhead nature. The loop iteration count information is also helpful to identify problems with underutilization of worker threads with small number of iterations that can be a result of outer loop parallelization. Consider inner loop parallelization or "collapse" clause to saturate the working threads in this case.



OpenMP Region / OpenMP Barrier-to-Barrier Segment / Function / Call Stack	Elapsed Time	SP GFLOPS	OpenMP Potential Gain						CPU Time	Serial CPU Time	Memory Bound	NUMA: % of Remote Accesses	% of FP Ops
			Imbalance	Lock Contention	Creation	Scheduling	Reduction	Atomics					
mainompparallel:80@/nfs/site/home/rupakroy/openmp_a	7.296s	10.705	1.361s	0s	0.000s	0.000s	0s	0s	541.474s	0s	32.0%	0.0%	5.6%
mainompparallel:80@/nfs/site/home/rupakroy/openmp_a	7.111s	5.693	1.161s	0s	0.000s	0.000s	0s	0s	530.338s	0s	36.5%	0.0%	3.3%
mainompparallel:80@/nfs/site/home/rupakroy/openmp_a	5.496s	6.604	1.017s	0s	0s	0.000s	0s	0s	407.891s	0s	30.3%	0.0%	3.2%
mainompparallel:80@/nfs/site/home/rupakroy/openmp_a	5.336s	0.000	0.874s	0s	0s	0.000s	0s	0s	397.883s	0s	32.4%	0.0%	0.0%
mainompparallel:80@/nfs/site/home/rupakroy/openmp_a	0.113s	0.000	0.075s	0s	0s	0s	0s	0s	8.014s	0s	0.0%	0.0%	0.0%
mainompparallel:80@/nfs/site/home/rupakroy/openmp_a	0.539s	0.000	0.039s	0s	0s	0s	0s	0s	5.758s	0s	0.0%	0.0%	0.0%
mainompparallel:80@/nfs/site/home/rupakroy/openmp_a	0.002s	0.000	0.002s	0s	0s	0s	0s	0s	0.080s	0s	0.0%	0.0%	0.0%
mainompparallel:80@/nfs/site/home/rupakroy/openmp_a	0.025s	0.000	0.023s	0s	0s	0s	0s	0s	0.857s	0s	0.0%	0.0%	0.0%
[Serial - outside parallel regions]	2.908s	0.832							15.782s	1.463s	1.0%	0.0%	5.7%

Analyze the **Potential Gain** column data that shows a breakdown of Potential Gain in the region by representing the cost (in elapsed time) of the inefficiencies with a normalization by the number of OpenMP threads. Elapsed time cost helps decide whether you need to invest into addressing a particular type of inefficiency. VTune Profiler can recognize the following types of inefficiencies:

- **Imbalance**: threads are finishing their work in different time and waiting on a barrier. If imbalance time is significant, try dynamic type of scheduling. Intel OpenMP runtime library from Intel Parallel Studio Composer Edition reports precise imbalance numbers and the metrics do not depend on statistical accuracy as other inefficiencies that are calculated based on sampling.
- **Lock Contention**: threads are waiting on contended locks or "ordered" parallel loops. If the time of lock contention is significant, try to avoid synchronization inside a parallel construct with reduction operations, thread local storage usage, or less costly atomic operations for synchronization.
- **Creation**: overhead on a parallel work arrangement. If the time for parallel work arrangement is significant, try to make parallelism more coarse-grain by moving parallel regions to an outer loop.
- **Scheduling**: OpenMP runtime scheduler overhead on a parallel work assignment for working threads. If scheduling time is significant, which often happens for dynamic types of scheduling, you can use a "dynamic" schedule with a bigger chunk size or "guided" type of schedule.
- **Atomics**: OpenMP runtime overhead on performing atomic operations.
- **Reduction**: time spent on reduction operations.

To analyze the source of a performance-critical OpenMP parallel region, double-click the region identifier in the grid, sorted by the **OpenMP Region/..** grouping level. VTune Profiler opens the source view at the beginning of the selected OpenMP region in the pseudo function created by the Intel compiler.

> **NOTE**
> By default, the Intel compiler does not add a source file name to region names, so the `unknown` string shows up in the OpenMP parallel region name. To get the source file name in the region name, use the `-parallel-source-info=2` option during compilation.

Limitations

VTune Profiler supports the analysis of parallel OpenMP regions with the following limitations:

- Maximum number of supported lexical parallel regions is 512, which means that no region annotations will be emitted for regions whose scope is reached after 512 other parallel regions are encountered.
- Regions from nested parallelism are not supported. Only top-level items emit regions.
- VTune Profiler does not support static linkage of OpenMP libraries.

See Also

knob analyze-openmp=true vtune option
vtune option
MPI Code Analysis

Profiling MPI Applications

OpenMP* Imbalance and Scheduling Overhead

Processor Cores Underutilization: OpenMP Serial Time

Custom Data Collection for Performance Analysis (NEW)

Learn how to configure a data collector to inject custom data into an analysis by Intel® VTune™ Profiler. Get additional context on the collected data and insights for an enhanced analysis.

When you collect performance data with VTune Profiler, you can configure a custom data collector to interact with the collected data. You can then inject custom data once the collection activity has been completed.

In this recipe, we will see how you can do this.

Content expert: Jeffrey Reinemann

- INGREDIENTS
- DIRECTIONS:

  1. Understand the Data Collector
  2. Run Hotspots Analysis
  3. Use the Custom Data File
  4. Append Existing Custom Data
  5. Display Custom Interval Data

Ingredients

Here are the hardware and software tools you need for this recipe.

- **Application**:
  - `Mandelbrot`: The recipe uses this sample application. You can use any application of your choice.
  - `CustomCollector`: This batch file is used as a custom data collector. However, you can use any compiled program or Python script.
- **Analysis Tool**: VTune Profiler version 2023 or newer - Hotspots Analysis

Understand the Data Collector

The `CustomCollector` data collector batch file has the following code:

```
@Echo Off
Echo %AMPLXE_COLLECT_CMD%
if "%AMPLXE_COLLECT_CMD%" == "start" goto start
if "%AMPLXE_COLLECT_CMD%" == "stop" goto stop
Echo Invalid command
Exit 1

:start
Rem Start command in non-blocking mode
Rem echo start my_collector_command_to_start_collection "%AMPLXE_DATA_DIR%"\data_file-hostname-
```

```
%COMPUTERNAME%.csv
Start C:\Source\CustomCollectorWin\Debug\CustomCollectorWin.exe
Start C:\Source\CustomCollectorWin\Debug\CustomCollectorWin.exe 2
Start C:\Source\CustomCollectorWin\Debug\CustomCollectorWin.exe 4
Exit 0

:stop
Echo stop "%AMPLXE_DATA_DIR%\..\log\CustomCollector0PID.txt"
Echo|set /p="taskkill /PID " > "%TEMP%\stop0.bat
type "%AMPLXE_DATA_DIR%\..\log\CustomCollector0PID.txt" >> "%TEMP%"\stop0.bat
Call "%TEMP%"\stop0.bat
Exit 0
Echo stop "%AMPLXE_DATA_DIR%\..\log\CustomCollector2PID.txt"
Echo|set /p="taskkill /PID " > "%TEMP%\stop2.bat
type "%AMPLXE_DATA_DIR%\..\log\CustomCollector2PID.txt" >> "%TEMP%"\stop2.bat
Call "%TEMP%"\stop2.bat
Echo stop "%AMPLXE_DATA_DIR%\..\log\CustomCollector4PID.txt"
Echo|set /p="taskkill /PID " > "%TEMP%\stop4.bat
type "%AMPLXE_DATA_DIR%\..\log\CustomCollector4PID.txt" >> "%TEMP%"\stop4.bat
Call "%TEMP%"\stop4.bat
Exit 0
```

The collector uses several environment variables including `AMPLXE_COLLECT_CMD`, a collect command argument which specifies whether to start, stop, pause, or resume the collection. The batch file implements the start and stop options only.

Other environment variables include `AMPLXE_DATA_DIR`, the data directory path name where you write custom data which will be integrated into the analysis results.

When you run the `CustomCollector` batch file, multiple instances of the custom data collector execute and collect different metrics. Each instance provides two custom data variables for a total of six variables from the three instances.

Run Hotspots Analysis

1. Open VTune Profiler.
2. Click the **Configure Analysis** button to set up a new analysis.
3. Set these fields:
   - In the **WHERE** pane, select **Local Host**.
   - In the **HOW** pane, select Hotspots Analysis in the Algorithm group in the Analysis Tree.
   - In the **WHAT** pane, select an application, process ID, or system-wide collection. This example uses the `Mandelbrot` application.
   - Also in the **WHAT** pane, expand the **Advanced Options** section. Use the **Custom Collector** text box to navigate to the `CustomCollector` batch file.
4. Click the **Command Line** button

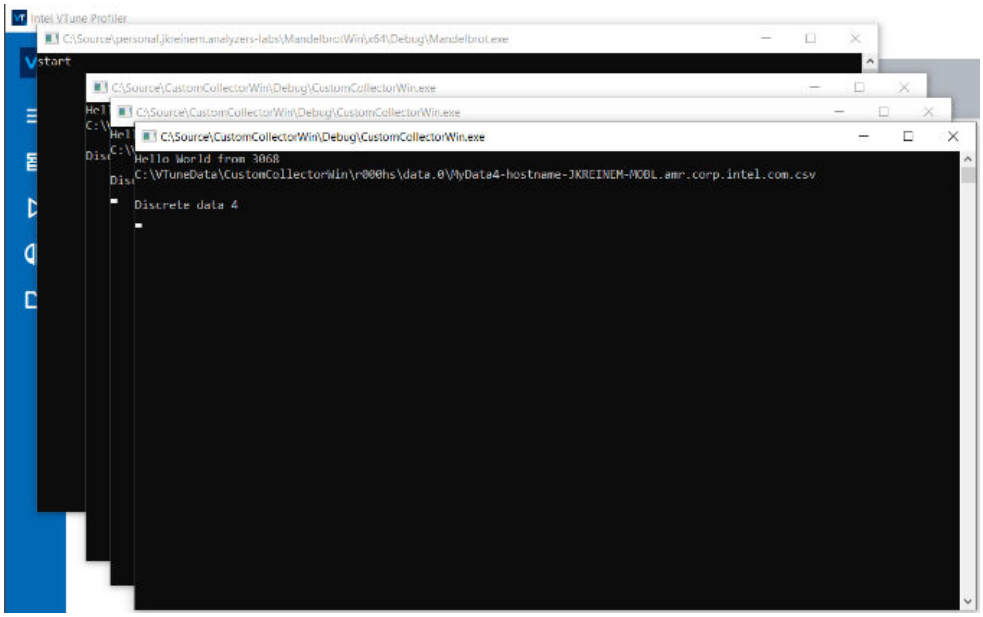
   and run this analysis from the command line.

   Command line:

   ```
   "C:\Program Files (x86)\Intel\oneAPI\vtune\latest\bin64\vtune" -collect hotspots -
   knob sampling-mode=hw -knob enable-characterization-insights=false -custom-
   collector=C:\Bats\CustomCollector.bat -finalization-mode=full --app-working-
   dir=C:\Source\personal.jkreinem.analyzers-labs\MandelbrotWin\x64\Debug --
   C:\Source\personal.jkreinem.analyzers-labs\MandelbrotWin\x64\Debug\Mandelbrot.exe
   ```

When the data collection begins, four output windows display:

- One instance for the `Mandelbrot` application
- Three instances for the `CustomCollector` data collector

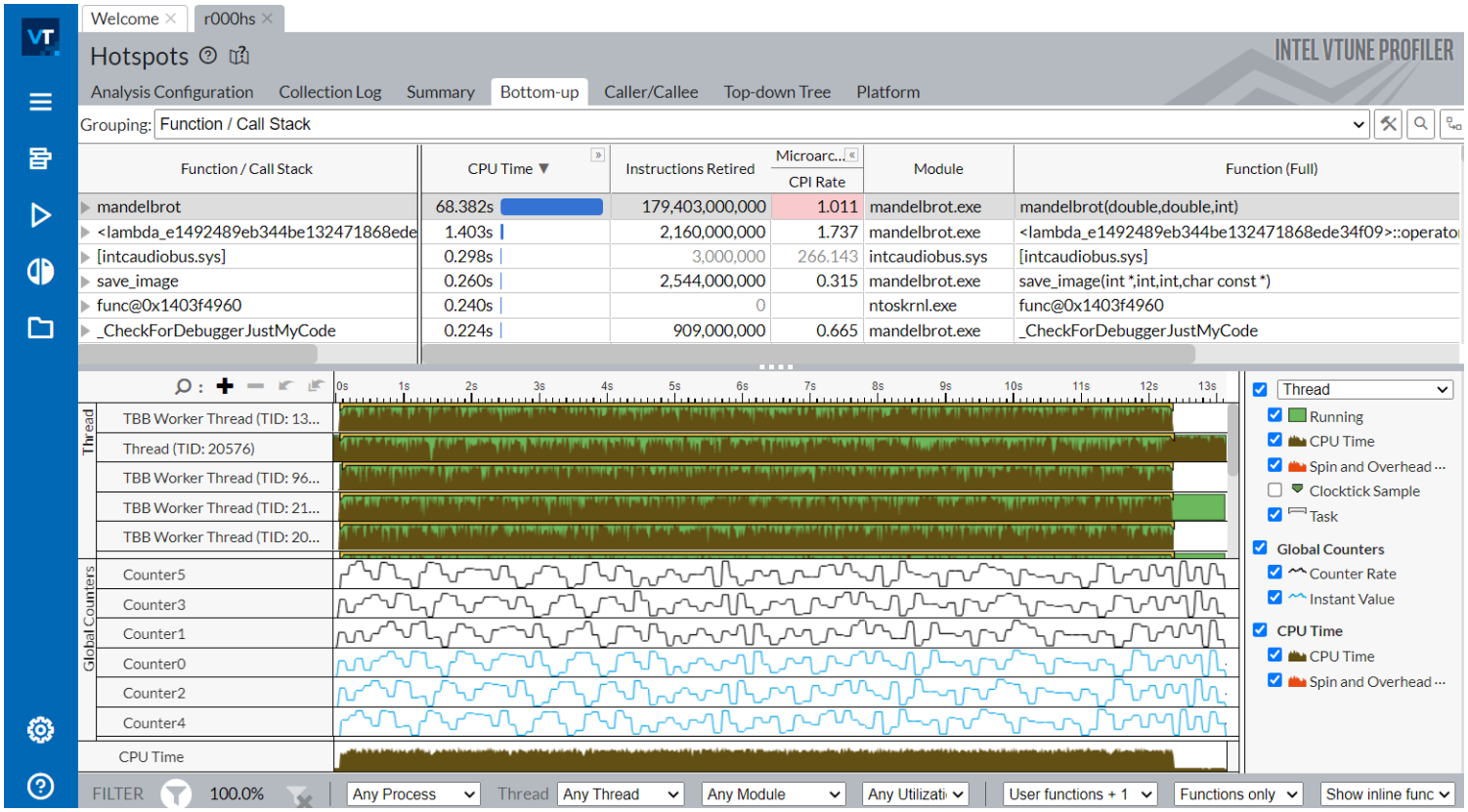


When the data collection finishes, VTune Profiler calls the custom data collector with the stop command in `AMPLXE_COLLECT_CMD`. The collector writes its custom data into the VTune Profiler results directory in `AMPLXE_DATA_DIR`.

VTune Profiler then integrates the custom data into the results during the finalization phase of the collection.

Once the results display, switch to the **Bottom-up** tab to see the custom data on the timeline. In this example,

- Custom data counters 0,2, and 4 display as Instant Values.
- Custom data counters 1,3, and 5 display as Counter Rate.



Use the Custom Data File

Let us now examine the custom data files that were created during our analysis.

Open the folder containing VTune Profiler results. Look in the data subfolder which contains all of the three custom data files.

The naming convention for custom data files is:

```
<user-defined string>-hostname-<host name of system under test>.csv
```

For example,

```
MyData-hostname-<host name of system under test>.csv
```

Understand the Custom Data File

The header of the custom data file indicates the name of the metric it captured and the corresponding data type. For example, the header in the custom data file below indicates that counter 0 is an Instant Value and counter 1 is a Counter Rate.

```
1  tsc.UTC,Counter0.INST,Counter1.COUNT,pid,tid
2  2022-08-31 22:07:54.161,30049,15024,,
3  2022-08-31 22:07:54.271,13587,21817,,
4  2022-08-31 22:07:54.378,4736,24185,,
5  2022-08-31 22:07:54.489,1437,24903,,
6  2022-08-31 22:07:54.599,10054,29930,,
7  2022-08-31 22:07:54.708,14780,37320,,
8  2022-08-31 22:07:54.816,8886,41763,,
```

Each line of data in the custom file must start with a time stamp to map it with other data collected at that time.

The following examples explain how you can use the custom data file in different ways:

- Generate a time stamp
- Write discrete data type information

- Format the path name
- Append existing custom data
- Display custom interval data

Generate Time Stamp

This example demonstrates how you can generate a time stamp for each line of data using `time` and `GetSystemTime` calls.

```
#define UTC_FORMAT "%Y-%m-%d %H:%M:%S"
char* get_utc_time()
{
    SYSTEMTIME    system_time;
    static time_t previous_local_time;
           time_t local_time;
    struct tm     gm_time;
           int    millisec;
           char   *time_fmt = NULL;

    time(&local_time);
    GetSystemTime(&system_time);
    millisec = system_time.wMilliseconds;
    if ((millisec == 0) && (local_time == previous_local_time))
        local_time++; // missed second rollover
    gmtime_s(&gm_time, &local_time);
    previous_local_time = local_time;

    time_fmt = (char *) malloc(128);
    if (time_fmt)
    {
        strftime(time_fmt, 128, UTC_FORMAT, &gm_time);
        sprintf_s(time_fmt, 128, "%s.%03d", time_fmt, millisec);
    }

    return time_fmt;
} // get_utc_time
```

Write Discrete Data Type Information

This example demonstrates how you can write the Instant Value and Counter Rate of a discrete data type to the custom data file.

```
// Discrete data format: tsc.[QPC|CLOCK_MONOTONIC_RAW|RDTSC|UTC],CounterName1.COUNT|
INST[,CounterName2.COUNT|INST],[pid],[tid]
#define DISCRETE_FORMAT_1 "%s,%llu,%llu,,\n"
#define DISCRETE_FORMAT_2 "%s,%llu,%llu,%lu,\n"
#define DISCRETE_HEADER "tsc.UTC,Counter%c.COUNT,Counter%c.COUNT,pid,tid\n"
void write_discrete_data(FILE *file_out, char *buffer, uint64_t tsc, uint64_t counter1, uint64_t
counter2, DWORD dPid, DWORD dTid)
{
    char   *utc_time     = get_utc_time();
    size_t bytes_written = 0;

    if (buffer && file_out && utc_time)
    { // buffer and file_out valid

        if ((dPid == 0) && (dTid == 0))
            sprintf_s(buffer, BUFFER_SIZE, DISCRETE_FORMAT_1, utc_time, counter1, counter2);
        else
            sprintf_s(buffer, BUFFER_SIZE, DISCRETE_FORMAT_2, utc_time, counter1, counter2,
```

```
dPid);

        bytes_written = fwrite(buffer, strlen(buffer), 1, file_out);
        fflush(file_out);
    }

    if (utc_time)
        free(utc_time);
} // write_discrete_data
```

Format Path Name

This example demonstrates how you can format the full path name to the custom data file (which will be created for a collection instance) using the environment variables of VTune Profiler.

```
 if (command)
        _dupenv_s(&command, &length, "AMPLXE_COLLECT_CMD");
    length = BUFFER_SIZE;
    if (datadir)
        _dupenv_s(&datadir, &length, "AMPLXE_DATA_DIR");
    if (filename && hostname)
    { // filename and hostname valid
        length = BUFFER_SIZE;
        _dupenv_s(&hostname, &length, "COMPUTERNAME");

        // log my process id so CustomCollector.bat can stop me
        sprintf_s(filename, BUFFER_SIZE, "%s\\..\\log\\CustomCollector%cPId.txt", datadir,
instance_id);
        printf("Data file: %s\n", filename);
        fopen_s(&file_out, filename, "wb");
        if (file_out)
        { // file_out valid
            sprintf_s(buffer, BUFFER_SIZE, "%u", my_pid);
            fwrite(buffer, strlen(buffer), 1, file_out);

            fclose(file_out);
            file_out = NULL;
        }

        // Note: appending .amr.corp.intel.com is unique to my test environment
        sprintf_s(filename, BUFFER_SIZE, "%s\\MyData%c-hostname-%s.amr.corp.intel.com.csv",
datadir, instance_id, hostname);
        std::cout << filename << std::endl;

        errnum = fopen_s(&file_out, filename, "wb");
```

Append Existing Custom Data

Suppose you have a custom data collector that runs independent of the VTune Profiler custom collector interface. If you want to include this data in your analysis results, you must make sure to run the independent data collector during the VTune Profiler analysis run. VTune Profiler includes only data with time stamps within its running time frame.

You must also ensure that the custom data file for the independent data collector adheres to the file name specifications and formatting rules seen earlier.

For example, this custom data file adds counters 6 and 7 to the results.

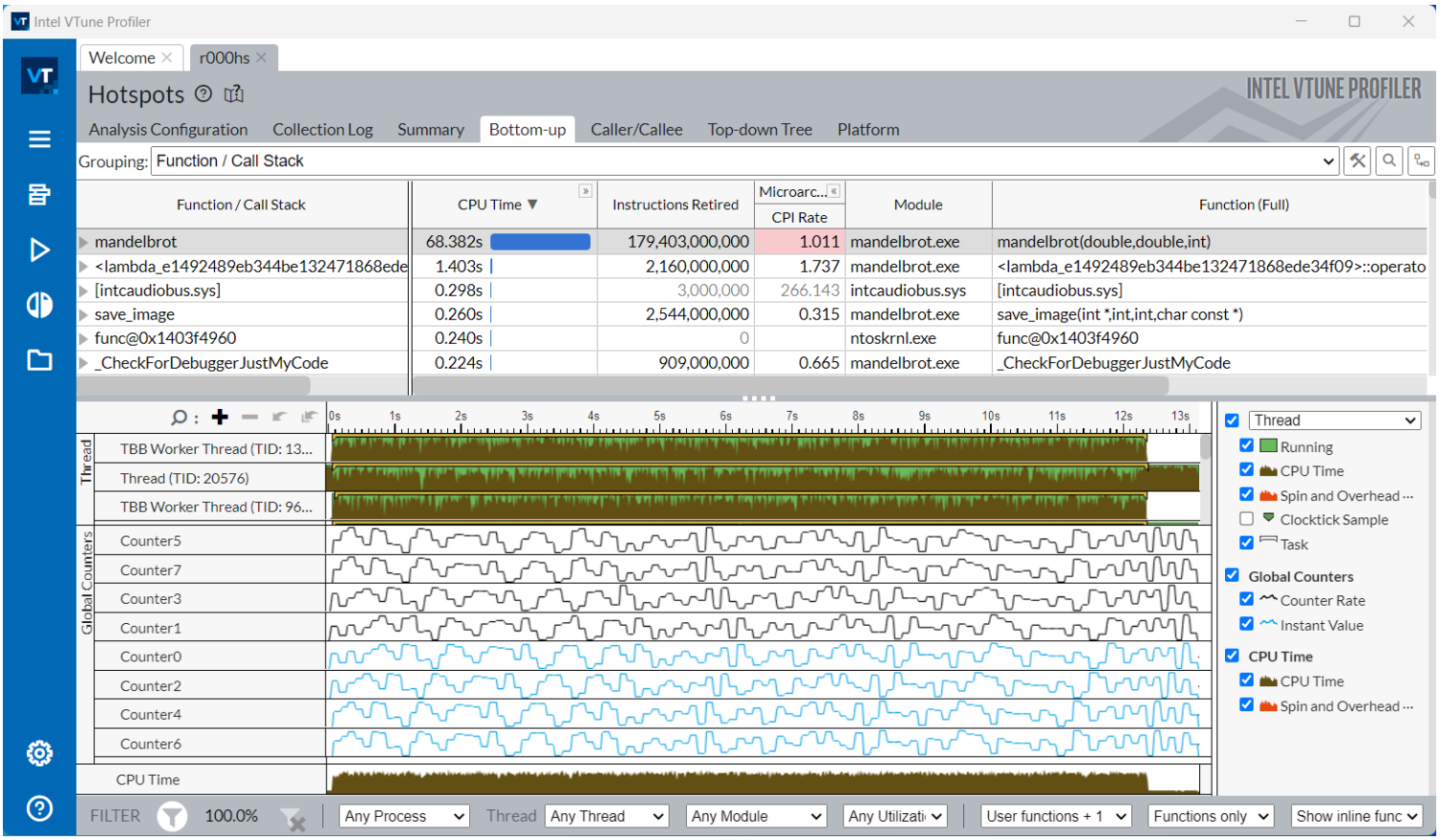
```
tsc.UTC,Counter6.INST,Counter7.COUNT,pid,tid
2022-08-31 22:07:54.193,10054,21783,,
2022-08-31 22:07:54.302,14780,53806,,
2022-08-31 22:07:54.410,8886,73059,,
2022-08-31 22:07:54.520,12970,101160,,
2022-08-31 22:07:54.630,24154,153493,,
```

Once the new custom data file is copied into the data subdirectory of VTune Profiler results,

1. In the VTune Profiler GUI, open the **Collection Log** tab.
2. Click the **Re-resolve** button (

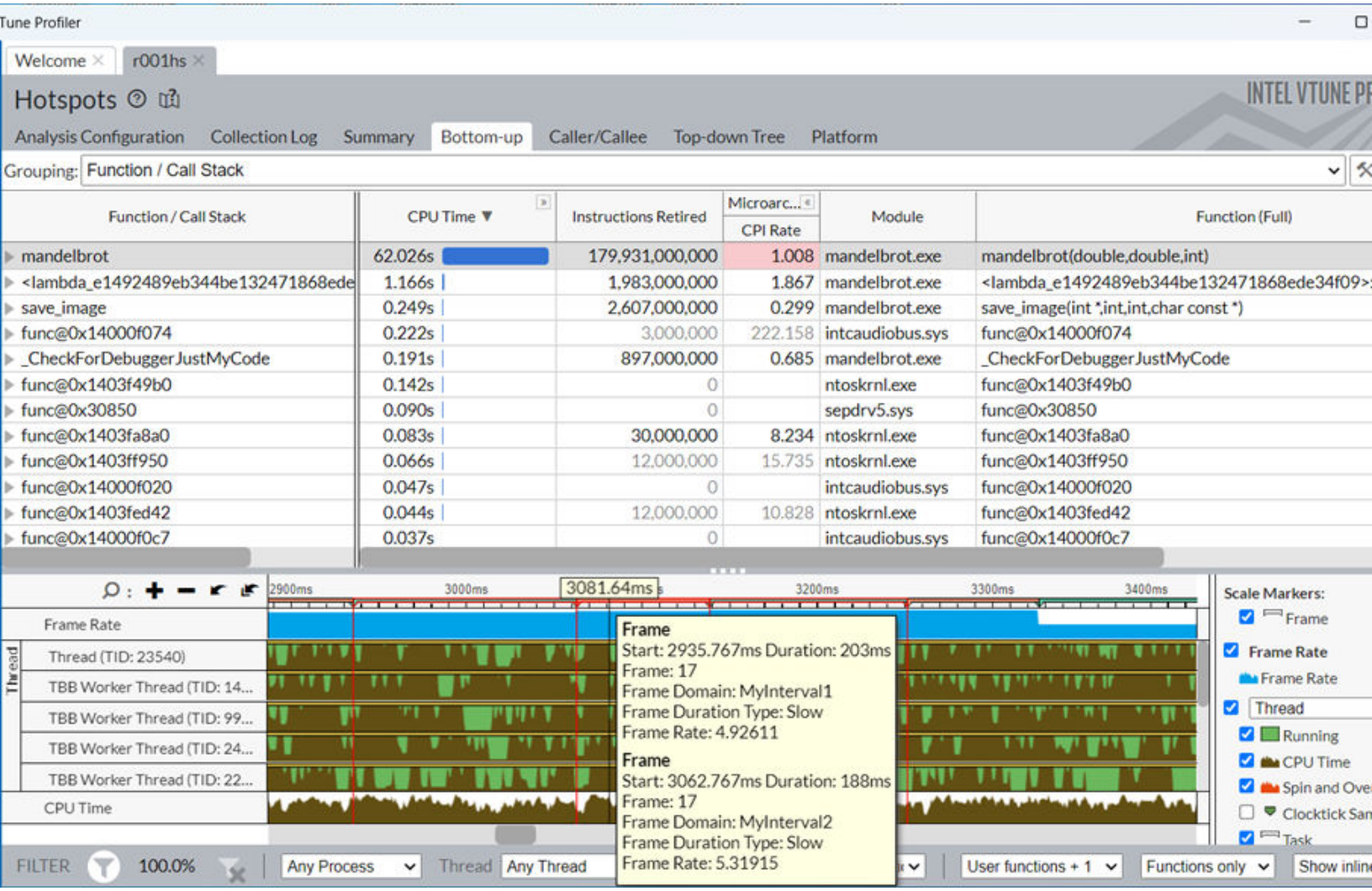
) to finalize the data again. This includes the new custom data file that has time stamps in the range of the collection time of VTune Profiler.
3. After finalization, switch to the **Bottom-up** tab and observe that counters 6 and 7 display in the data.



Display Custom Interval Data

When you collect custom Interval Data, after finalization of results, this information appears in the **Frame Rate** swim lane of the timeline.

In the following example, there are two overlapping custom interval data - `MyInterval1` and `MyInterval2`. In the pop-up window below, you can see the start time of each interval. Hover your mouse over the timeline to see all active intervals along with their start times and durations.



The custom interval data file must adhere to the same naming and formatting conventions explained previously:

- The file name follows the structure `<user-defined string>-hostname-<host name of system under test>.csv`.
- The header indicates the interval name, start, and stop times. The header may also include program and thread IDs.

```
1  name,start_tsc.UTC,end_tsc,pid,tid
2  MyInterval1,2022-08-31 23:17:17.735,2022-08-31 23:17:17.804,,
3  MyInterval2,2022-08-31 23:17:17.736,2022-08-31 23:17:17.913,,
4  MyInterval1,2022-08-31 23:17:17.804,2022-08-31 23:17:17.990,,
5  MyInterval2,2022-08-31 23:17:17.913,2022-08-31 23:17:18.097,,
```

This example demonstrates how you can write interval data to the custom data file.

```
// You can mix multiple intervals in the data. Starting one interval stops the previous interval.
// The format for the interval data is name,start_tsc.[QPC|CLOCK_MONOTONIC_RAW|RDTSC|
UTC],end_tsc,[pid],[tid]
#define INTERVAL_FORMAT "%s,%s,%s,,\n"
#define INTERVAL_FORMAT_2 "%s,%s,%s,%lu,\n"
#define INTERVAL_HEADER "name,start_tsc.UTC,end_tsc,pid,tid\n"
void write_interval_data(FILE* file_out, char* buffer, const char* name, char *start_utc, char
*end_utc, DWORD dPid, DWORD dTid)
{
```

```
    size_t bytes_written = 0;

    if (buffer && file_out)
    { // buffer and file_out valid
        if ((dPid != 0) || (dTid != 0))
            sprintf_s(buffer, BUFFER_SIZE, INTERVAL_FORMAT_2, name, start_utc, end_utc, dPid);
        else
            sprintf_s(buffer, BUFFER_SIZE, INTERVAL_FORMAT, name, start_utc, end_utc);
        bytes_written = fwrite(buffer, strlen(buffer), 1, file_out);
        fflush(file_out);
        if (bytes_written < 1)
            std::cout << "fwrite() failed\n";
    }
} // write_interval_data
```

See Also

Import External Data
Create a CSV File with External Data
`custom-collector` Reference Information

Software Optimization for Intel® GPUs (NEW)

Use Intel® VTune™ Profiler to estimate overhead when offloading onto an Intel GPU. Analyze the performance of computing tasks offloaded onto the GPU.

The increasing popularity of heterogeneous computing has led performance-conscious developers to discover that different types of workloads perform best on different hardware architectures. Intel provides many high-performance architectures including CPUs, GPUs, and FPGAs. This methodology describes how you use VTune Profiler to profile and optimize compute-intensive workloads offloaded onto Intel GPUs.

Understand Your Intel GPU

- **Employ parallelism:** Extracting superior performance from a workload-intensive GPU begins with an understanding of GPU architecture and functionality. A GPU employs a high level of parallelism with several smaller processing cores that work together. A GPU is well suited for workloads that can be split into tasks that run concurrently. Single-core serial performance on a GPU is much slower than on a CPU. Therefore, applications must take advantage of the massive parallelism available in a GPU.
- **Move data intelligently:** Using a GPU requires you to move data to and from the GPU, which can create overhead and impact performance. Marshall data intelligently to take advantage of temporal and spatial locality in the GPU. Using registers and caches to store data close together is important to get the best performance.
- **Use the offload model:** Although your GPU is available to handle the most significant parts of your workload, your CPU is still vital to perform other workload tasks. Use the GPU in an *offload* model, where you offload some portion of your workload onto the GPU(*target*) device. The GPU functions as an *accelerator* for those parts that perform best on the GPU. The CPU(*host*) executes the rest of the workload. Optimizing software performance in this context centers on two major tasks:
  - Optimal offload onto a GPU
  - Optimization for the GPU

NOTE This methodology focuses on the use of a general-purpose GPU (GPGPU) exclusively for computation. It covers these aspects of using GPUs in a computation model:

- What to offload
- How to offload
- How to write the GPGPU algorithm
- How to use GPU Offload Analysis in VTune Profiler to analyze GPU offload performance

The methodology does not address the use of Intel GPUs for graphics. For analysis of graphical applications, use the GPU Compute/Media Hotspots Analysis in VTune Profiler as well as Intel® Graphics Performance Analyzers (Intel® GPA).

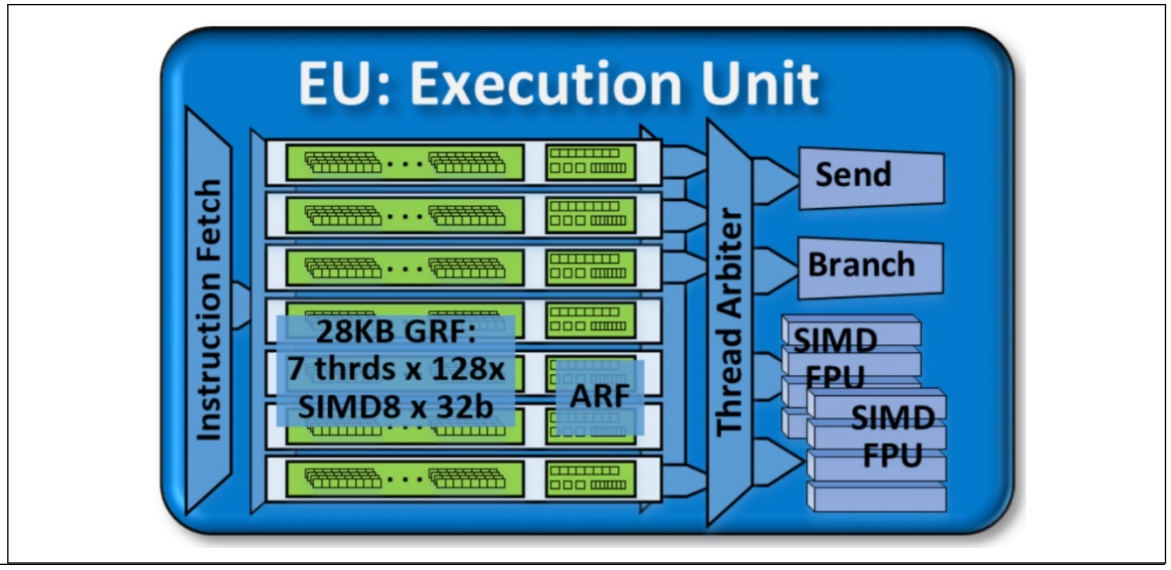
Intel GPU Architecture

NOTE Families of Intel® Xe graphics products starting with Intel® Arc™ Alchemist (formerly DG2) and newer generations feature GPU architecture terminology that shifts from legacy terms. For more information on the terminology changes and to understand their mapping with legacy content, see GPU Architecture Terminology for Intel® Xe Graphics.

Before we examine the GPU offload model, let us first examine the architecture of an Intel GPU, like the **Gen9 GT2 GPU**. This device is integrated into Intel® microarchitecture codenamed Skylake. You can program this GPU using high level languages like OpenCL* and SYCL*.

The Gen9 GT2 GPU has a single slice with 24 Execution Units (EUs). An Execution Unit is the foundational building block of GPU architecture.

The Execution Unit (EU) of a Gen9 GT2 GPU



An EU is a combination of simultaneous multi-threading (SMT) and fine-grained interleaved multi-threading (IMT). EUs are computing processors that drive multiple issue, Single Instruction Multiple Data Arithmetic Logic Units (SIMD ALUs). These SIMD ALUs are pipelined across multiple threads. SIMD ALUs are useful for high-throughput floating-point and integer computations. The fine-grain threaded nature of the EUs ensures continuous streams of instructions that are ready for execution. The IMT also hides the latency of longer operations like memory scatter/gather, sampler requests, or other system communications.

The thread arbiter dispatches several instructions in each cycle of operation. When these instructions do not propagate to functional units, there is a stall. The duration of a stall is measured by the number of execution cycles that passed in that state. This measure helps us estimate the efficiency of EUs. The **EU Array Stalled**

metric counts the number of cycles when the EU was stalled but at least a single thread was active. The stalling could happen when the EU was waiting for data from a memory subsystem. See GPU Metrics Reference (in the VTune Profiler User Guide) for more information on related GPU metrics.

The Importance of Efficient Scheduling

To use the full computing power of a massively parallel machine, you must provide all EUs in the GPU with enough calculations to execute. Therefore, EUs have more hardware threads than functional processing units. Having more hardware threads can cause an oversubscription of instructions that need to be executed, but this can also help hide stalls due to data that is waiting.

The scheduling of threads in this manner is an expensive operation. To make the scheduling efficient and cost-effective, it is important to keep all EUs busy as much as possible. Scheduling can be ineffective in these situations:

- The quantity of calculations is too small. Here, the scheduling overhead may be comparable to the time spent on completing useful calculations.
- The quantity of calculations is too large. In this case, work distribution between threads can be uneven. The entire occupancy of all EUs in the GPU will drop.

Use the **EU Threads Occupancy** metric to detect both of these situations. Low thread occupancy is a clear indicator of ineffective distribution of workloads between threads.

Another situation that is less common happens when there are no tasks for EUs for a certain time period. The EUs are then idle, and the idle state can impact occupancy negatively. Use the **EU Array Idle** metric to detect this situation.

SIMD Execution with Floating-Point Units

In an EU, the primary computation units are a pair of SIMD floating-point units (FPUs). These FPUs actually support both floating-point and integer computations. This table describes the SIMD execution capability of these FPUs.

Data Size	Data Type	Number of SIMD Operations
16-bit	Integer	8
16-bit	Floating point	8
32-bit	Integer	4
32-bit	Floating point	4

The **EU IPC Rate** metric is a good indicator of the saturation of the FPUs. For example, if two non-stalling threads saturate the floating-point compute throughput of the machine, the EU IPC Rate metric is 2. Typically, this metric is below its theoretical maximum value of 2.

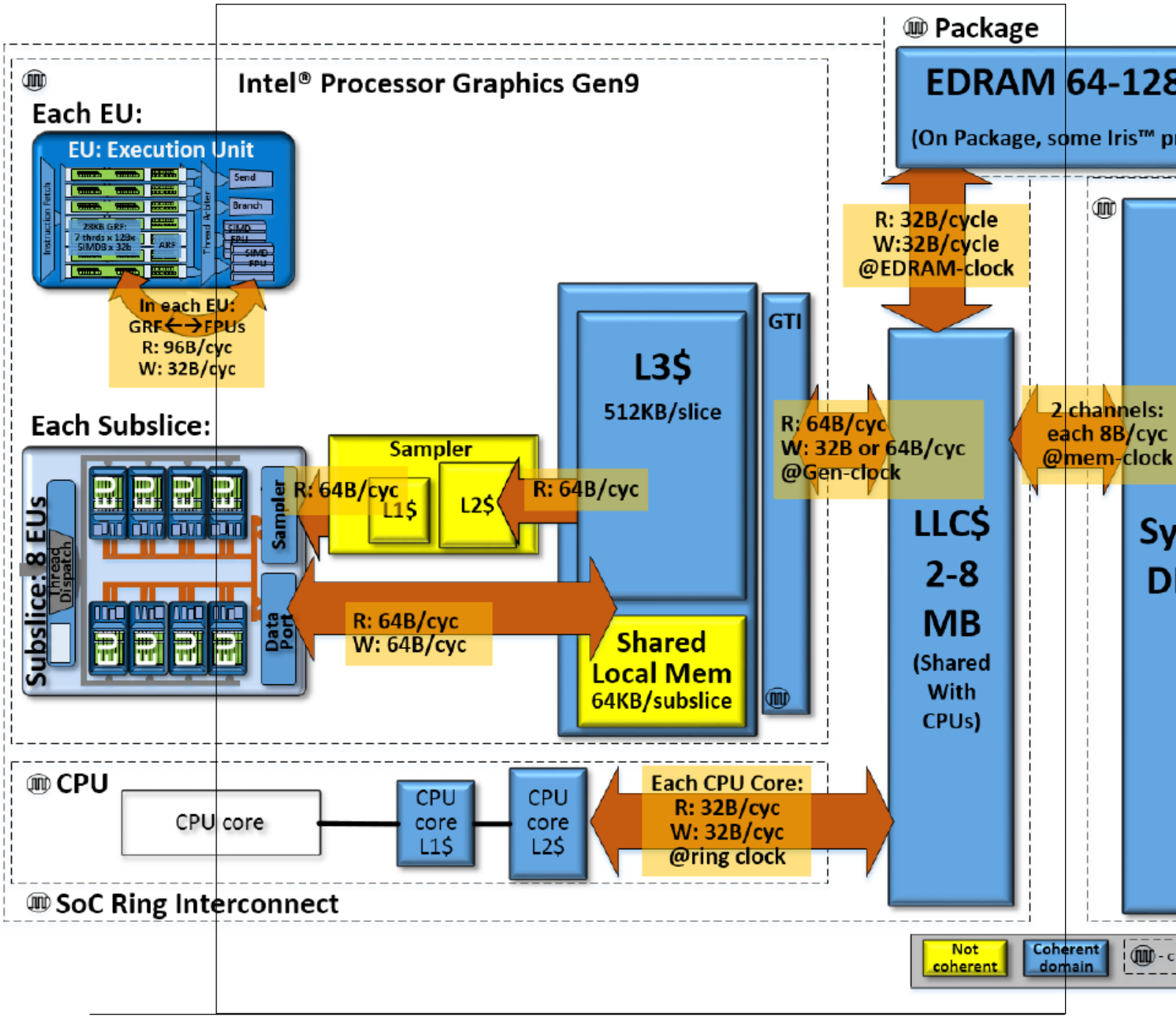
In the event that FPUs are saturated, but the data width is low, there is insufficient use of instruction level parallelism. In this case, look at the **SIMD Width** metric.

SIMD Width Value	Implication
Less than 4	See what is preventing the compiler from performing loop vectorization.
4 or higher	There is successful vectorization of instructions by the compiler. Removing data dependencies or applying loop unrolling techniques to the code can increase this value to 16 or 32, which is a good condition for data locality and cache re-use.

Memory Subsystem

The Gen9 GT2 GPU has a unique memory subsystem with a **Unified Memory Architecture**. It shares its physical memory with the CPU and employs the zero copy buffer transfer effectively. This feature can speed up data transfer between CPU and GPU, as illustrated below.

Memory hierarchy of Intel Processor Graphics Gen9 GT2 GPU at the SoC Level



EUs receive data from DRAM/LLC memories. They can take advantage of the reuse of data blocks that are cached in GPU L3 or the Shared Local Memory (SLM). Due to massive parallelism, when all EUs request data from memory, they can saturate the bandwidth capabilities of the memory sub-blocks.

Access to the local CPU caches is much faster than access to system memory. In an ideal situation, data access should happen from the local CPU caches as well. Similarly, the data read by EUs can remain in the L3 GPU cache. If reused, data access from the cache would be much faster than fetching data from main memory.

In the Gen9 memory architecture, each slice has access to its own L3 cache. Each slice also contains two sub-slices. Each sub-slice contains:

- A Local Thread Dispatcher
- An instruction cache
- A data port to L3
- Shared Local Memory (SLM)

You can control data locality by one of two methods:

- Particular consequent data access, which helps the hardware that stores the data in L3 cache.
- A special API to allocate local memory that is accessible for a work group and is served in SLM by hardware.

While access to data in L3 cache is very fast, the cache capacity itself is not very large. Traversing large arrays can make the cache useless as data may get evicted. The **L3 Cache Miss** metric indicates the amount of data access required to fetch data from memory behind GTI. Data blocking techniques can also help with reducing cache misses. For example, when you keep blocks for data fitted to an SLM, the Local Thread Dispatcher for a sub-slice can retain the highest level of data locality. You can use VTune Profiler to track SLM traffic and see information about the amount of data transferred as well as the transfer rate.

GPU Profiling Features in Intel® VTune™ Profiler

This methodology focuses on several key features in VTune Profiler that are tailored to support GPU analysis. The following workflow highlights these features:

1. Run the **GPU Offload Analysis** on your application.
   - Find out if your application is CPU or GPU bound.
   - Define GPU Utilization.
   - See if GPU EUs are stalled during execution.
   - Identify the computing tasks that were most responsible to keep the GPU busy. These tasks could be candidates for further analysis of GPU efficiency.
2. Collect a **GPU Compute/Media Hotspots** profile. Get a list of top computing tasks with metrics on:
   - Execution time
   - EU efficiency
   - Memory stalls
3. Use the **Memory Hierarchy Diagram** to work on the most inefficient computing tasks.
   - Analyze data transfer/bandwidth metrics.
   - Identify the memory/cache units that cause execution bottlenecks.
   - Make decisions on data access patterns in your algorithm based on GPU microarchitectural constraints.
4. Run the **Instructions Count** preset analysis on kernels.
   - Verify instruction sets and the selection of SIMD instructions generated by the compiler.
   - Leverage special compilation options and pragmas so the compiler generates more efficient instructions.
5. For large compute kernels, use the **Basic Block Latency** preset of the GPU Compute/Media Hotspots analysis.
   - Identify the code regions that are responsible for the greatest execution latency.
   - Explore the latency metrics against your source code lines through the **Source View**.
6. Use the **Memory Latency** preset to find memory access code that created significant execution stalls.

- Examine memory access details through assembly instructions in the **Assembly View**, which displays latencies against each individual instruction.
- Use known optimization techniques for GPUs to rearrange data access for a more memory-friendly pattern.

7. Repeat iterations of the **GPU Compute/Media Hotspots analysis** on your improved algorithm until you are satisfied with performance metrics.

Optimization Methodology When Offloading to Intel GPU

Heterogeneous applications are normally designed in a manner that the portion to be offloaded onto an accelerator is already identified. If you do not already know what code portions to offload, use Intel® Offload Advisor for this purpose as the decision can be a complex task.

This methodology assumes that you have already identified the code to be offloaded onto a GPU. We now focus on the best way to implement this offload on the host side.

Step 1: Examine Device Utilization

Your optimization methodology should distribute the time spent on algorithm execution by CPU cores and accelerator EUs effectively. Usage metrics on device utilization (**CPU Usage** and **GPU Usage** metrics) can help us determine this efficiency early on. Ideally, these values are 100% but if there are gaps or delays in the execution, use VTune Profiler to identify the locations in the application code where they occurred.

Step 2: Define Efficiency of Code Execution on the GPU

Let us look at the matrix sample application. This contains matrix-to-matrix multiplication operations over FP data with dense matrix `C = A B`.

For the sake of coding simplicity, `A`, `B` and `C` are square `n × n` matrices.

> **NOTE**
> For the sake of readability and compactness of representation, we apply many simplifications. The matrix multiplication types of benchmarks are well known, and many computing optimization methods are developed even for accelerators. We consider the analysis of algorithms instead of their synthesis.

```
for (size_t i = 0; i < w; i++)
        for (size_t j = 0; j < w; j++) {
            c[i][j] = T{};
            for (size_t k = 0; k < w; k++)
                c[i][j] += a[i][k] * b[k][j];
    }
```

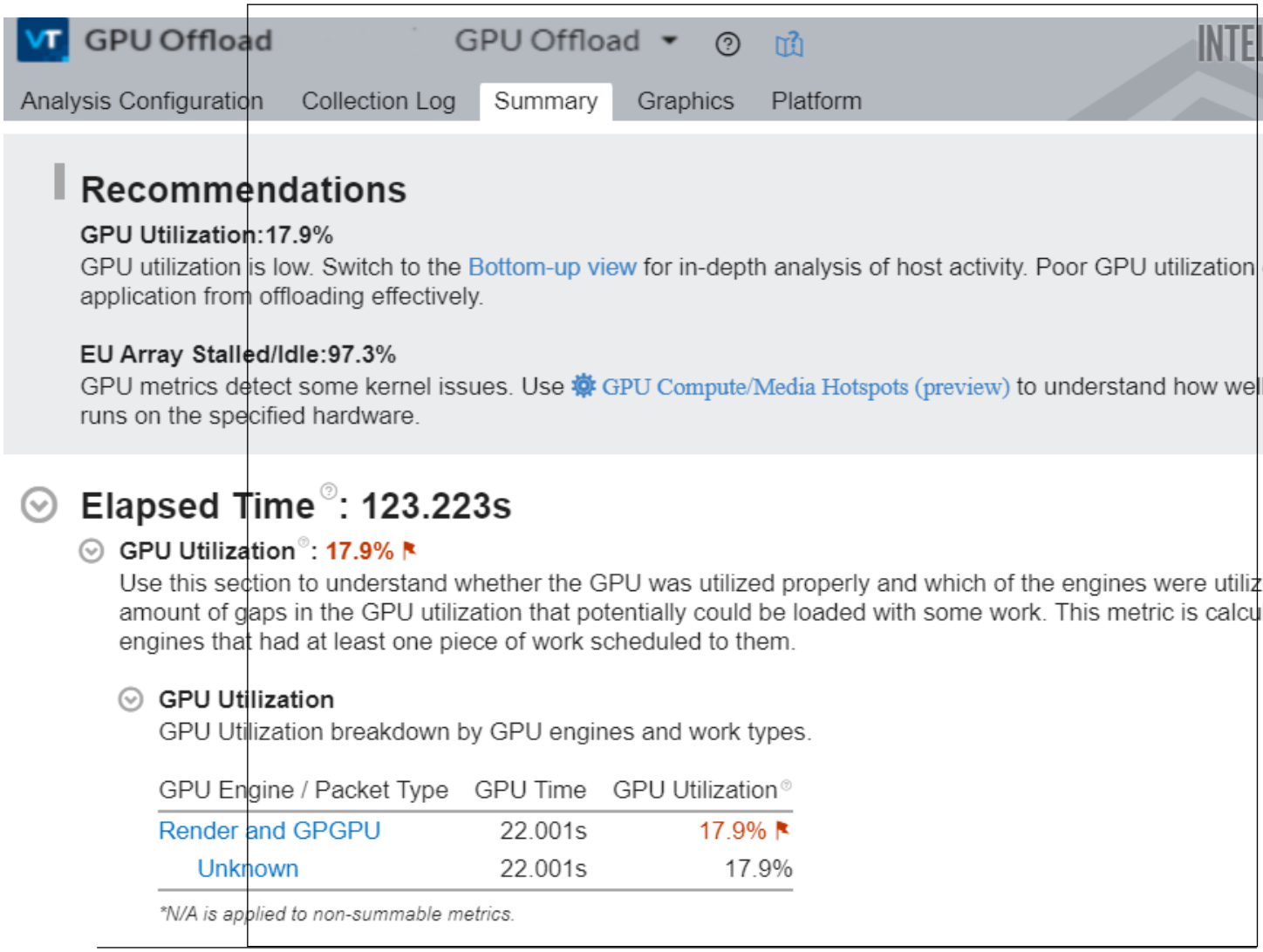
In this example, we look at a simplified C++ version of the matrix sample. This version has been stripped of details about kernel submission into a queue. The actual matrix sample is written to SYCL* standards and compiled with the Intel® oneAPI DPC++/C++ compiler.

Let us identify a portion of this code to offload onto an accelerator. Typically, the outermost look is a good candidate. However, in this example, the innermost loop could be a compute kernel. Also, the innermost loop in this snippet may not necessarily be the innermost loop in the sample either. Higher level library calls or third party functionality could mask an entire structure of computer iteration. Therefore, for the purpose of explaining this methodology, we choose to offload the innermost loop:

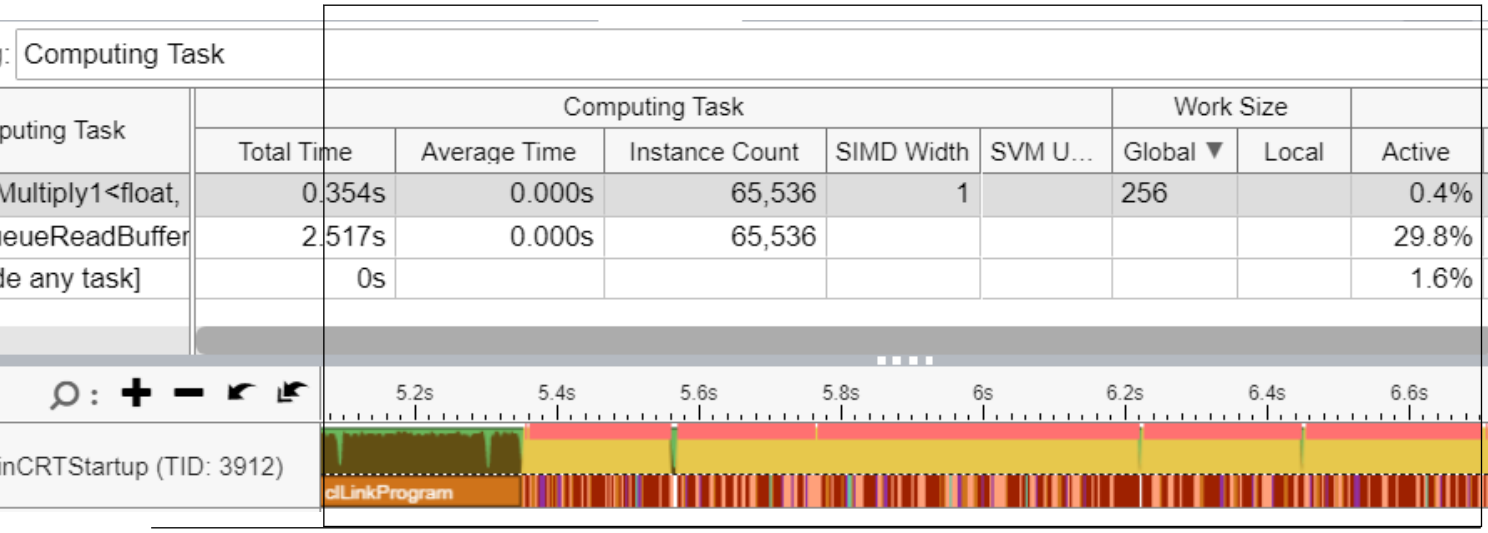
```
for (size_t k = 0; k < w; k++)
                c[i][j] += a[i][k] * b[k][j];
```

Step 3: Run GPU Offload Analysis

Use the GPU Offload Analysis in VTune Profiler to quickly identify the hottest computing tasks offloaded to a GPU. You can also clarify CPU activity when submitting these tasks. In the example below, we focus on a single active computing task. Therefore, we can ignore the CPU here. We use the GPU Offload analysis to collect information about computing task execution on the GPU.



Once the analysis is complete, the **Summary** window informs us about measurements of GPU Utilization and EU Stalls. Following the recommendations here, let us first examine host activity that could be responsible for low GPU utilization. We switch to the **Graphics** tab to open the **Bottom-up view**.



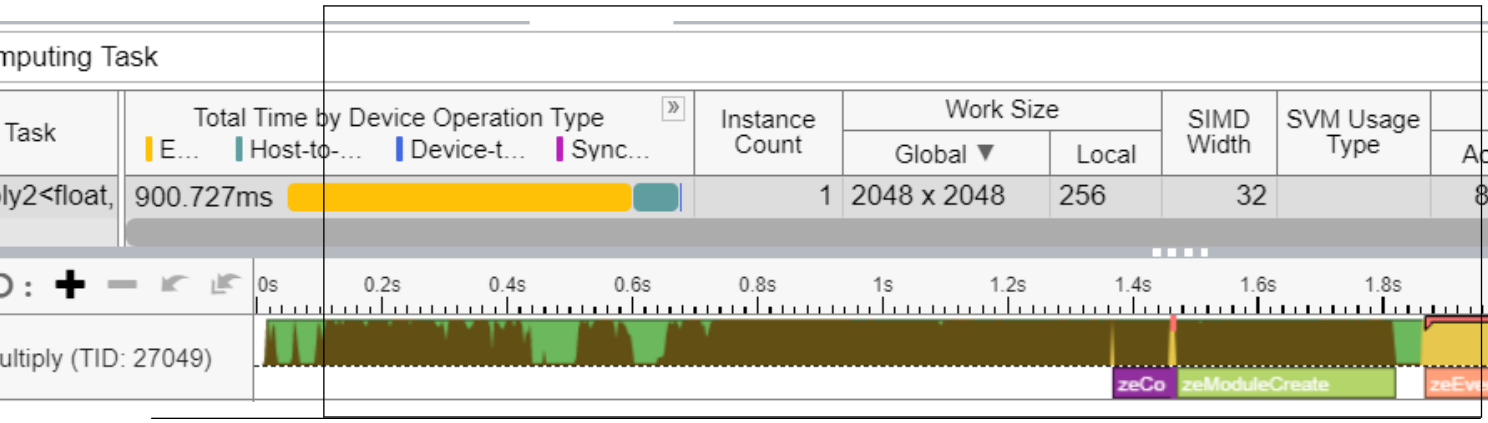
Look at the `matrixMultiply1` kernel results in the figure above.

In order to get the work completed in a reasonable amount of time, this version of the kernel uses 256X256 dimension matrices. The **Instance Count** column tells us that the kernel was invoked 65,536 times. Each instance was so small that the average time of the kernel was rounding off to zero seconds. The spectrum pattern in the timeline also indicates a rapid kernel invocation rate. In this case, most of the time is spent on creating small kernels. The **Idle** column in the **EU Array** section informs us that the EUs were idle for 92.6% of the time. Invoking too many short kernels is a key indicator of work inefficiency.

The **Work Size** section reveals that there was inefficiency in work distribution. Let us now offload the outer loop.

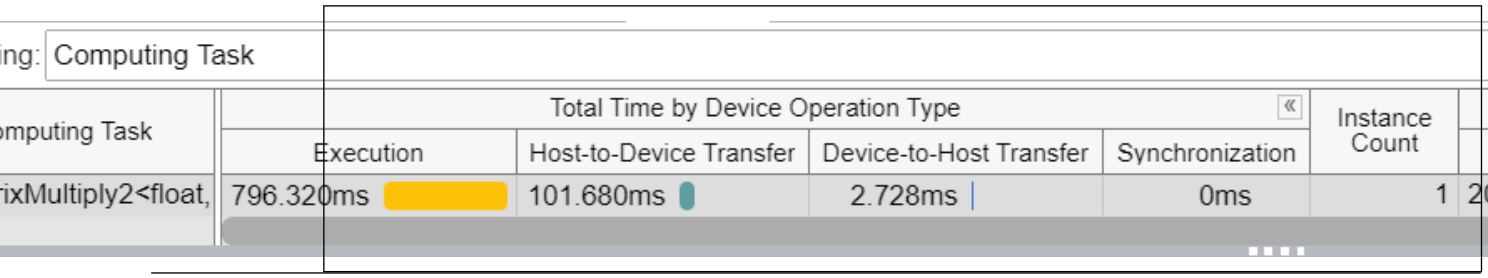
This action should give better performance by reducing the number of compute kernel instances to one (`matrixMultiply2`). The figure below shows a GPU Hotspots analysis for this improved version of the kernel. This version is also called a *Naïve implementation*.

GPU Hotspots Analysis for naive implementation of matrix multiply example



In this case, the size of the matrices was increased to 2048 X 2048 and the wall-clock performance was still more than 10x faster. The **EU Threads Occupancy** metric is high. This indicates that there is enough work available for the execution units.

Task time characterized by device operations



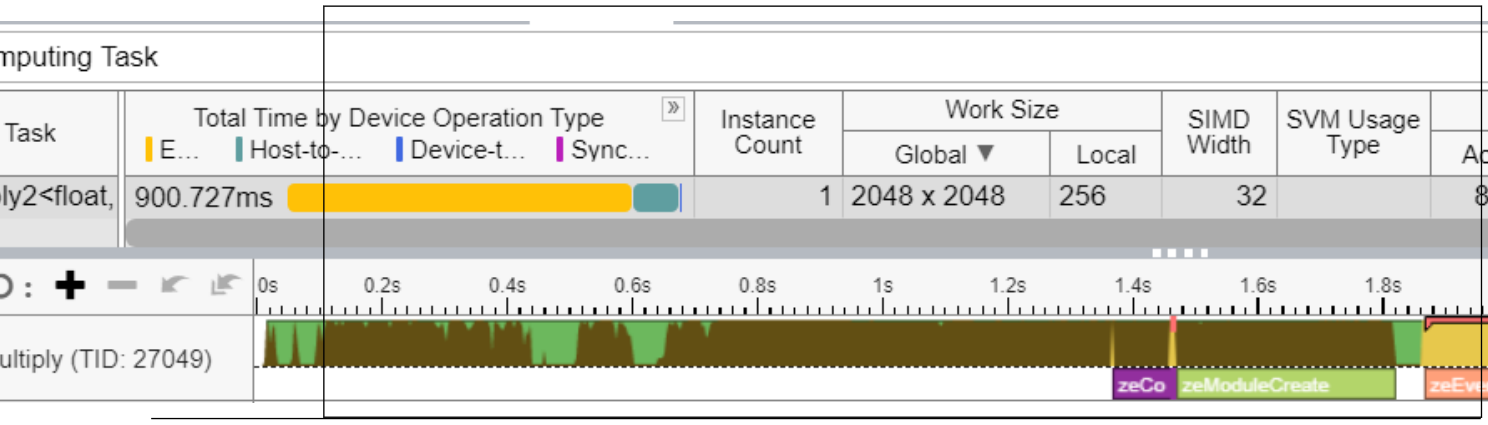
When we look at the timeline in the figure above, we observe a single computing task that took nearly 800 ms, versus data transfer that only took 100 ms. This ratio between executing data and transferring data is more desirable. Further improvements to the algorithm can result in greater improvements to this ratio.

Notice that the compiler generated the full length of SIMD instructions (SIMD Width=32). This arranged data access that resulted in the EUs being active for 86.8% of the time, as opposed to near zero in the previous run. This exercise demonstrates the importance of providing enough work within each invocation of a kernel.

Step 4: Run GPU Compute/Media Hotspots Analysis

The naive implementation of the matrix multiplication example is much faster than the initial version. But we can still expect improvements in performance.

VTune Profiler reported a high value for the **EU Threads Occupancy** metric (95.7%), which meant that the work was distributed properly among EUs. But the execution engine is still underutilized with the `matrixMultiply2` kernel. We deduce this from the **EU Array Stalled** metric, which is only 9.2%.



To investigate limiting factors for a kernel, let us run the GPU Compute/Media Hotspots analysis. This way, we can see detailed information about kernel execution in a GPU.

Our first step is to identify if the kernel is computed bound or memory bound.

The GPU Hotspots analysis has several predefined profiles or presets. You can use these presets to collect different metrics related to memory access and computing efficiency. To understand kernel execution better, we use the **Full Compute** preset. From the information in this preset, we see that EU FPUs were only active 63.5% of the time by executing the kernel `matrixMultiply2`.

FPU activity for the compute kernel

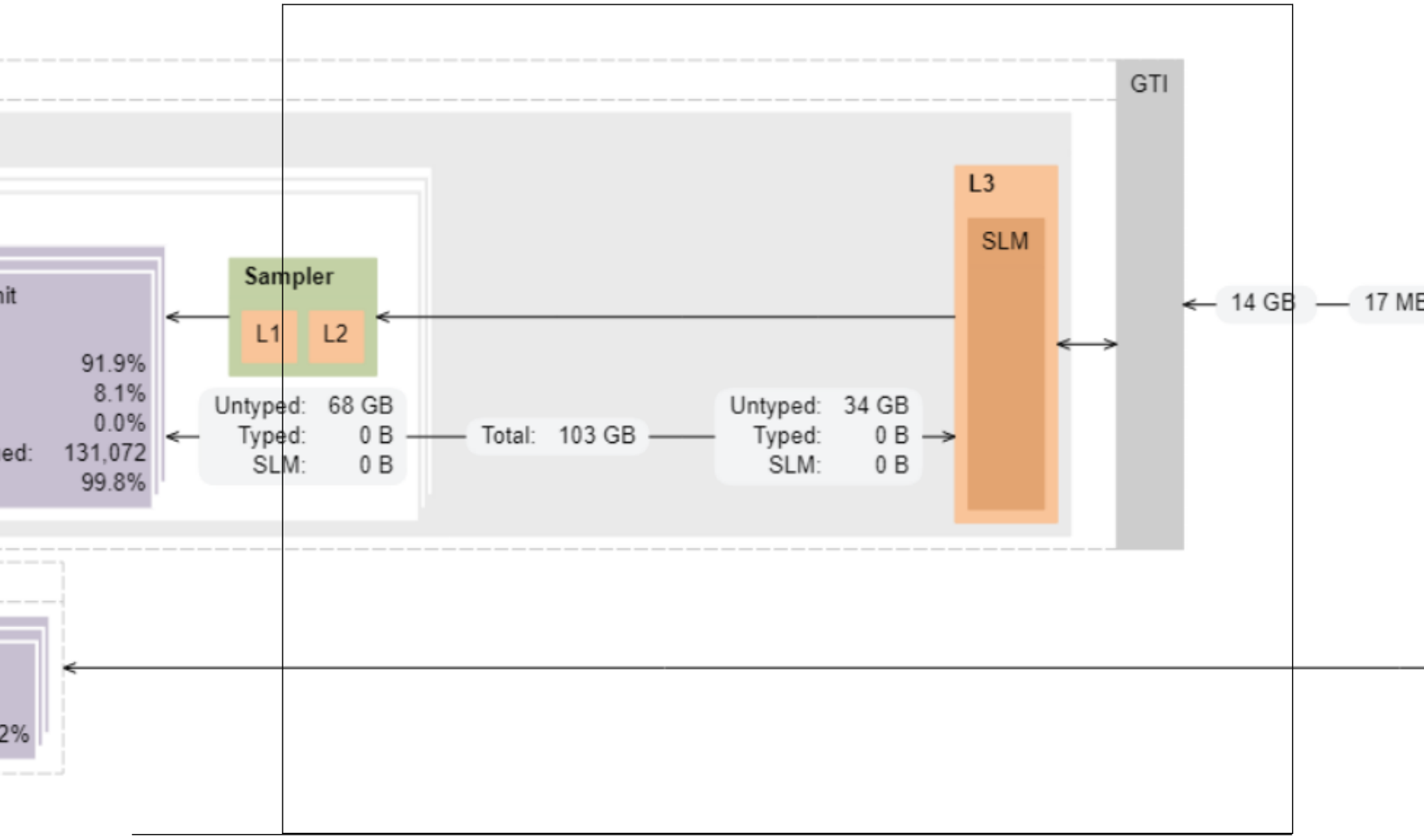
Computing Task	EU Instructions			L3 Bandwidth, GB/sec	Untyped Memory Bandwidth, GB/sec		Shared L
	IPC Rate	2 FPUs active	Send active		Read	Write	Read
matrixMultiply2<float,	1.705	63.5%	20.0%	132.179	88.105	44.074	0.000

Therefore, the kernel was *memory bound*, not compute bound.

Our next step is to examine the **Memory Hierarchy Diagram**. This diagram provides data transfer information between EUs and memory units. The information can help us define optimization steps in the code of the kernel.

When we select the **Overview** preset, the Memory Hierarchy Diagram displays values for the bandwidth of the links between memory units (like GPU L3 Cache, GTI Interface, LLC and DRAM) and EUs, as well as total data transferred between them.

Kernel data transfer in the GPU memory subsystem



Notice the overall amount of data transferred to EUs (~68 GB) and data brought from LLC/DRAM through the GTI interface (14 GB).

When you compare these data sizes to the size of each matrix data array (2048x2048x4=16MB), the transferred amount is enormously high. This condition makes execution ineffective due to access to global memory. We should address this issue with more efficient data access (a consequent data or unit stride access in array) and minimal access to the global memory.

Step 5: Additional Kernel Code Optimizations

Fetching data from global memory is a common performance limiting factor for GPUs. This problem is worsened in the case of discrete GPUs. Here, the PCIe bus introduces more bandwidth and latency limitations. A common but sub-optimal approach is to increase data locality and reuse. This is done by blocking matrix areas and completing multiply-add operations within the smaller blocks that fit into a cache memory that resides closer to execution units. You can implement this optimization by one of two ways:

- Allow the hardware to recognize frequently accessed data and preserve it in a cache automatically.
- Exercise more manual control over access to data blocks by placing the most used data in the Shared Local Memory.

Use care when implementing the latter as it can result in these conditions:

- Poor management of threads, as SLM access is limited to threads from its slice only.
- Slow data access in case the ratio of data reads/write is below a certain threshold.

> **NOTE** The impact of the read/write ratio on GPU performance can vary with GPU hardware. Therefore the read/write threshold value is subtle and depends on the GPU hardware. But an increase in the number of write operations increases the chances of performance slowdown.

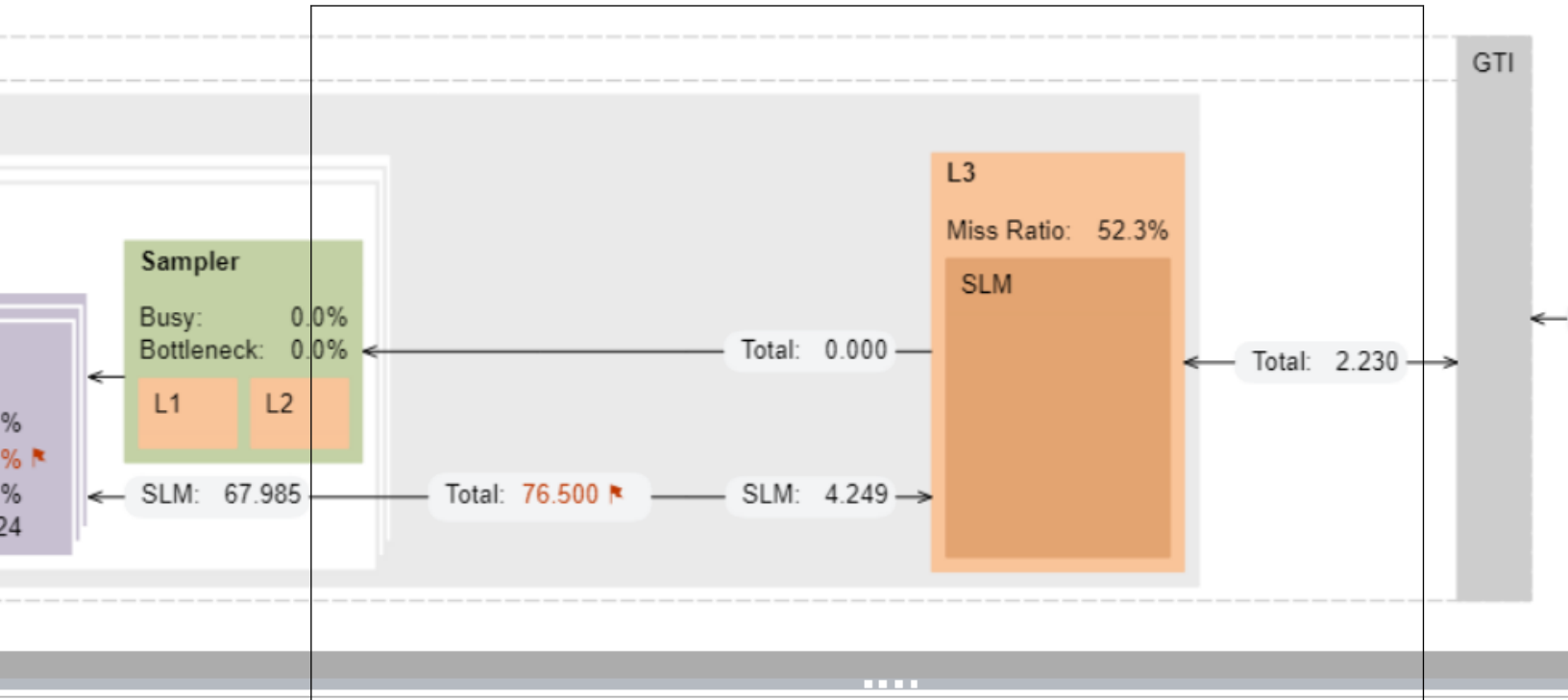
One approach to use SLM with the matrix multiplication algorithm is to split the global work set of matrices into blocks or tiles and perform dot product operations in the tiles separately. This action should decrease the number of global memory accesses as the entire tile should fit into the SLM area. Although this approach does not enable optimal access to data arrays, the access is much faster due to achieved data locality.

In the code snippet below, the pseudo code demonstrates the idea of data accesses to tiles in the local index space.

```
i, j // global idx
    for (size_t tidx = 0; tidx < TILE_COUNT; tidx++)
         ti, tj // local idx
         ai, aj, bi, bj // global to local idx
         ta[ti, tj] = a[ai, aj]
         tb[ti, tj] = b[ai, aj]

         for (size_t tk = 0; tk < TILE_SIZE; tk++)
                c[i][j] += ta[ti][tk] * tb[tk][tj];
     }
```

The implementation of the tiled multiplication significantly redistributes the data flow. An analysis of the `matrixMultiply4` kernel (see Memory Hierarchy diagram below) reveals some observations:

Tiled kernel data transfer in the Shared Local Memory (SLM)



EU Array			Computing Threads Started	L3 Sampler Bandwidth, GB/sec	L3 Bandwidth, GB/sec	L3 <-> GTI Total Bandwidth, GB/sec	Shared Loca
ctive	Stalled	Idle					Read
57.1%	42.9%	0.0%	259,424	0.000	155.891	4.544	138.540
61.7%	36.3%	2.1%	413,309	0.000	33.946	21.465	0.000

- The data volume coming from LLC via GTI interface is just ~2 GBs, most of which came from L3/SLM.
- The L3 Bandwidth metric (highlighted in the table above) reached 155GB/s, which is more than 70% of the maximum L3 bandwidth.
- 42% of EUs were still stalled.

From these observations, we can conclude that the algorithm execution is *still memory bound*, albeit with much faster cache memory. In total, the kernel now executes almost 5x faster than the naïve implementation we started with.

Next, let us look at the total time for computing tasks, as shown in the table below.

Timing for the tiled kernel

Grouping: Computing Task

Computing Task	Work Size		Computing Task			
	Global	Local	Total Time ▼	Average Time	Instance Count	SIMD Width
matrixMultiply4<float, (unsig	2048 x 2048	16 x 16	490.727ms	490.727ms	1	16
clEnqueueReadBufferRect			3.116ms	3.116ms	1	
[Outside any task]			0ms	0ms	0	

There are a few ways in which we can enable a faster implementation.

- **Organize high level data access in a more optimal way.**

  Using sub-groups for data distribution, we can leverage sub-slices of the GPU that access their own local memory.
- **Use a low-level optimization for specific GPU architecture and use optimized libraries like Intel® oneAPI Math Kernel Library (oneMKL).**

  These steps can help us achieve near maximum performance with the GPU. However, any GPU has its theoretical limit for performance that can be calculated using some known characteristics.

  For example, let us calculate the theoretical minimum time for algorithm execution in the Gen9 GPU. From the Gen9 GT2 GPU architecture parameters, we know that this GPU contains 24 EUs. Each EU has two FPUs (SIMD-4). Each FPU can perform two operations (MUL+ADD). With a max core frequency of 1.2 GHz, the maximum FP performance is:

  24 * 2 * 4 * 2 = 384 Flop/cycle (32b float)

  384 * 1.2 = 460.8 GFLOPS

  The number of FP operations of the naïve matrix multiplication implementation is $2*N^3$, which is approximately 17.2 GOPS when N=2048.

  Theoretically, if we were not limited by data access inefficiency and bandwidth constraints, the algorithm could be calculated in 17.2 / 460.8 = 0.037 sec or 37 ms. The VTune results revealed that the best time executed by the kernel was 490 ms, which is over 10x slower than the theoretical calculation time. We can therefore conclude that there is still room for performance improvement.

Scaling Performance

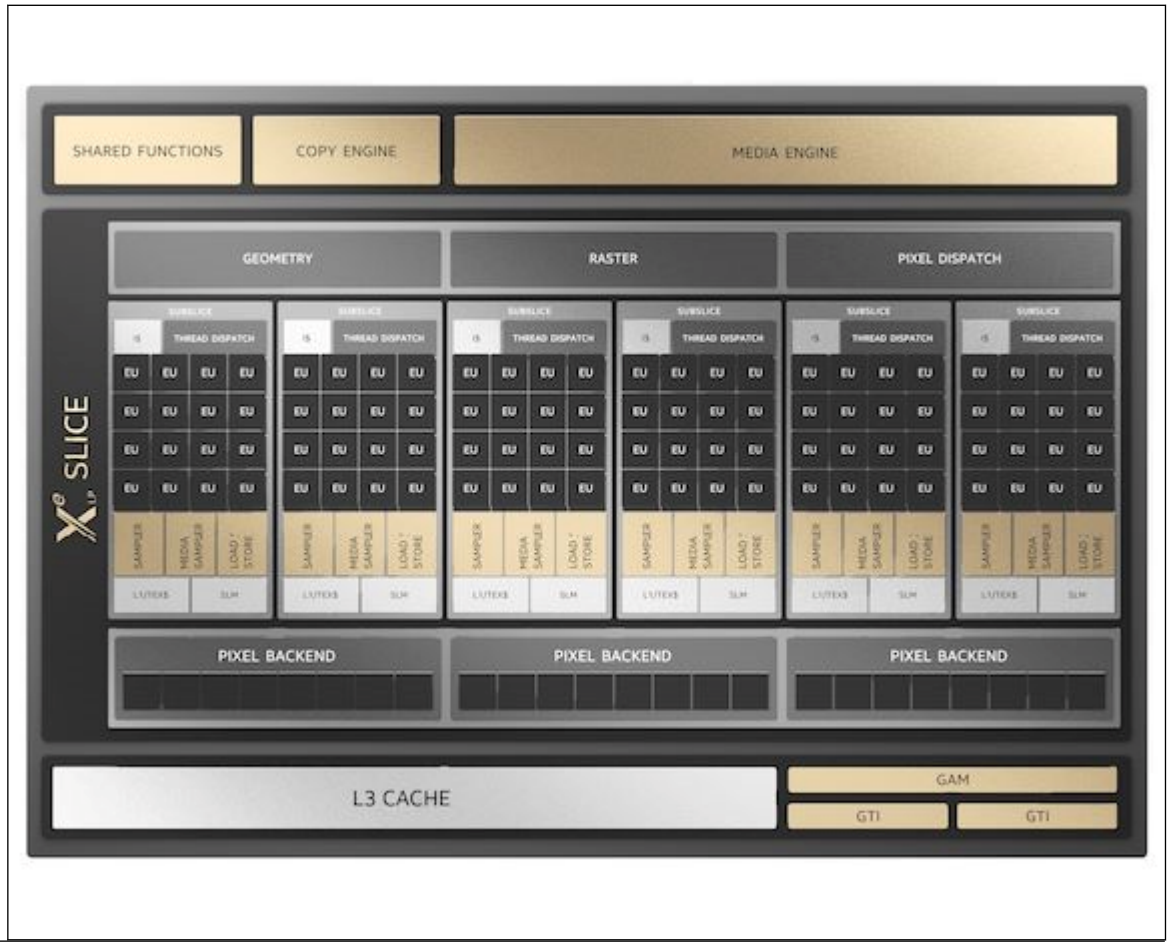
A highly parallel application, like the matrix multiplication sample, leverages the increased efficiency from the use of GPU resources. However, using additional compute resources should also increase performance, provided the scaling is not limited by memory bottlenecks.

In the Gen9 series of GPUs, there are GT3 and GT4 options, which contain 48 EUs and 72 EUs respectively. However, embedded GPUs have a fundamental limitation in area. This prevents us from adding more EUs for greater potential scaling, and bigger cache blocks for faster data access. Discrete GPUs are less limited by area or power constraints. If a system allows integration with a single, external GPU or with multiple GPUs, we could scale up accelerator performance.

However, remember that between the main CPU, its memory, and the GPU, there will be a communication interface (like a PCIe bus). This may have its own constrains on bandwidth, latency, and data coherency.

Let us look at an Intel® Iris® Xe MAX GPU, previously known as a PCIe discrete graphics card with the code name DG1.

High level view of the Intel® Iris® Xe MAX microarchitecture



An analysis of the same tiled kernel implementation gives us these results:

The tiled matrixMultiply kernel in GPU Hotspots results

Computing Task	Work Size		Computing Task			
	Global ▼	Local	Total Time	Average Time	Instance Count	SIMD Width
matrixMultiply4<	2048 x 2048	16 x 16	111.455ms	111.455ms	1	16
clEnqueueRead			159.791ms	159.791ms	1	
[Outside any tas			0ms			

The kernel execution is roughly 4x faster.

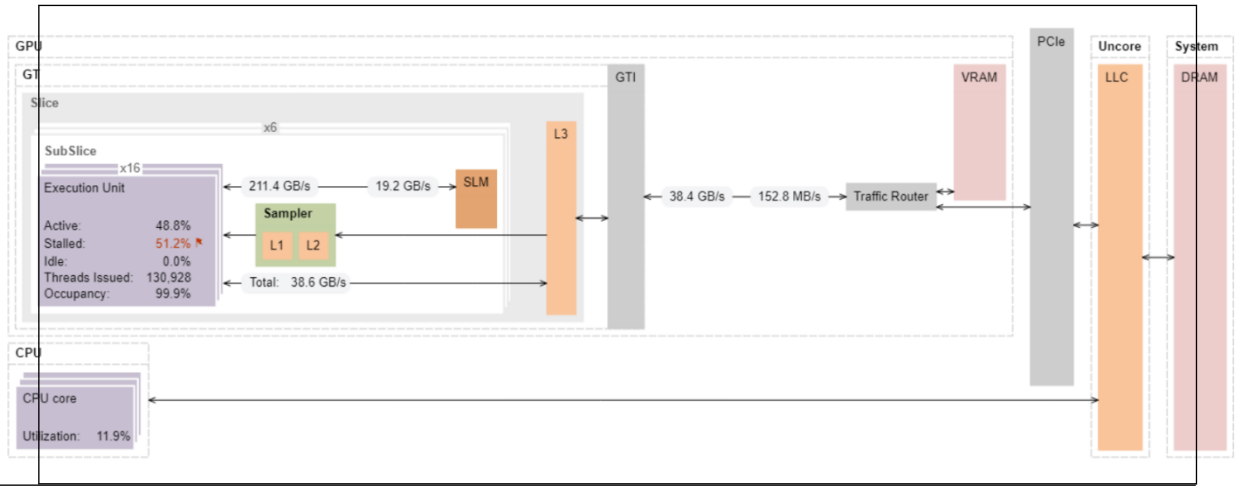
This is expected, as the Intel® Iris® Xᵉ MAX GPU has 96 EUs against the 24 EUs in the observed Gen9 GPU. However, we can notice in the table below that the EUs are still stalled 51% of the time during execution. This is quite likely due to the wait for data from memory (which is well known for general matrix algorithms). The question is, which one?

EU Array metrics for the tiled `matrixMultiply4` kernel

sk	EU Array			EU Threads Occupancy	Computing Threads Started	L3 Bandwidth, GB/sec	Shared Local Memory Bandwidth, GB/sec	
	Active	Stalled	Idle				Read	Write
<float,	48.8%	51.2%	0.0%	99.9%	130,928	6.5%	17.8%	19.220
dBuffer	0.3%	99.6%	0.0%	94.7%	261,730	0.2%	0.0%	0.033
ask]	0.0%	2.7%	97.3%	2.5%	49,710	0.0%	0.0%	0.000

If we switch the mode in the results grid to show the percentage of maximum bandwidth, we observe that the L3 and GPU memory bandwidth was far from the maximum, so they are not bottlenecks. Let us look at the Memory Hierarchy Diagram to get a better picture of data transfers.

Memory Hierarchy Diagram with data transfer metrics



Beyond the GTI interface, data comes from the VRAM or main DRAM. As we prepare matrix data on the CPU side, we know that data for matrix *a* and matrix *b* is transferred via PCIe to the GTI. The measured GTI bandwidth is a rough indication of the data rate required for PCIe interface. The measured data read rate is 38 GB/s at the GTI interface, while PCIe 3.0x16 has a theoretical maximum of only16GB/s one way. A reasonable conclusion is that we are limited to the PCIe bandwidth. To measure the data traffic on PCIe with VTune Profiler, we need a server platform, which has PCIe performance counters.

On a server-based setup, the bandwidth on the PCIe is much lower than bus limitations. So, we can conclude that:

- All data is being fetched from VRAM and the EU stalls. This may be defined by the latency of traveling data from video memory to EUs.
- Since the data traffic between EUs and L3 is the same as between GTI and the external traffic router, you can achieve additional performance optimizations using a better reuse of the L3 cache. For example, you can introduce second level of matrix tiling with blocks size that would fit to the L3 cache of each GPU slice.

Conclusion

Generally, in heterogeneous applications, once a certain workload is offloaded onto an accelerator, it is essential to provide enough computing tasks for massively parallel accelerator machines like a GPU.

- Improve the efficiency of the GPU by estimating the data transfer and task scheduling overhead for offloaded tasks.
- Use the **GPU Utilization** and **GPU Occupancy** metrics in the GPU Offload analysis of VTune Profiler to estimate the inefficiency of using a GPU.
- The performance of a computing task execution may be limited by several microarchitectural factors, like the lack of Execution Units or presence of bottlenecks in memory subsystems or interfaces. Run the **GPU Compute/Media Hotspots analysis** to identify these limitations. Highlight the bottlenecks on the GPU Memory Hierarchy Diagram along with detailed microarchitecture metrics for every computing task. For more complicated kernels, use the latency analysis to identify the most critical code inside a kernel.

See Also

The-Compute-Architecture-of-Intel-Processor-Graphics-Gen9-v1d0.pdf
Offload with Intel Advisor
VTune Profiler User Guide
Profiling a SYCL* application running on a GPU
Optimize Applications for Intel® GPUs with Intel® VTune™ Profiler
GPU Architecture Terminology for Intel® Xe Graphics
Using the Command-Line Interface to Analyze the Performance of a SYCL* Application running on a GPU
Intel oneAPI Math Kernel Library

Core Utilization in DPDK Apps

Explore metrics that characterize core utilization in terms of packet receiving in Data Plane Development Kit (DPDK)-based applications.*

In data plane applications where you require fast packet processing, the DPDK is supposed to poll a certain port for incoming packets in an infinite loop, pinned to a certain logical core. Such a polling model of packet retrieval poses the challenge of measuring effective core utilization. The CPU time on the core (where the polling loop is running) is always close to 100%, regardless of the many loop cycles when the DPDK runs idle. So, the CPU time cannot reflect how the core is utilized on the packet retrieval. However, for this polling model, the core utilization indicator might be **Rx Spin Time - % of wasted polling loop cycles**. Wasted Cycles are iterations during which the DPDK does not receive any packets.

Follow this recipe to analyze the efficiency of packet retrieval in a DPDK-based workload.

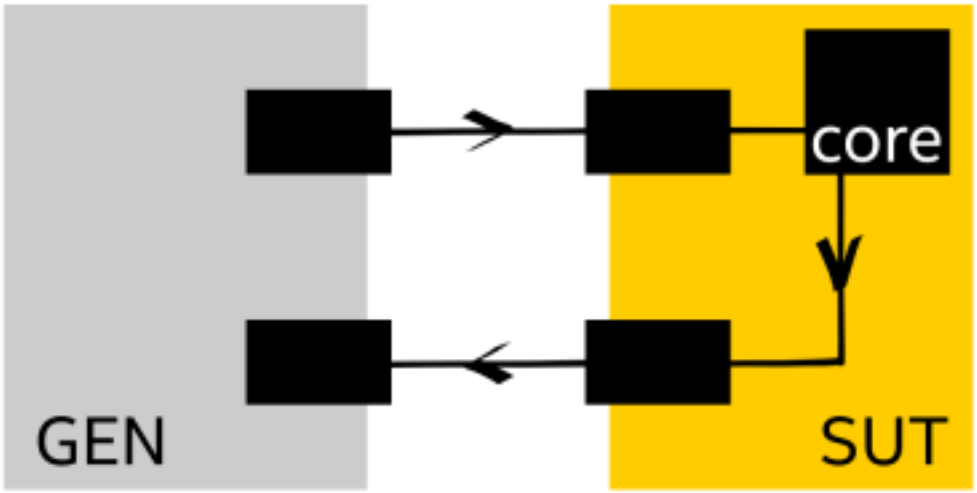
Content expert: Jeffrey Reinemann

1. INGREDIENTS
2. DIRECTIONS:
   a. Run Input and Output analysis
   b. Analyze core utilization with the DPDK Rx Spin Time metric
   c. Analyze packets retrieval with DPDK Rx Batch Statistics histogram
   d. Understand Rx operations and investigate Rx peaks

Ingredients

- **Application**: A DPDK `testpmd` application that runs on a single core and performs L2 forwarding. The application is compiled against DPDK with profiling support for VTune Profiler.
- **Tools**:
  - **DPDK with VTune Profiler profiling support enabled**: DPDK versions 18.11 (and newer) include profiling support for VTune Profiler. When using earlier versions, apply the attached patches (available for versions 17.11, 18.02, and 18.05). To enable profiling on the DPDK side, enable the VTune Profiler to attach to the DPDK polling cycle. For this, reconfigure and recompile the DPDK (and the target

application) with the `CONFIG_RTE_ETHDEV_RXTX_CALLBACKS` and `CONFIG_RTE_ETHDEV_PROFILE_WITH_VTUNE` flags enabled (located in the `config/common_base config` file).
  - **Intel® VTune™ Profiler**: Input and Output analysis
- **Operating system**: Test system that consists of the traffic generator (GEN in the picture below) providing 64-byte frames and packet receiver (SUT - system under test), connected via 40 GbE link. The SUT performs L2 forwarding of packets.

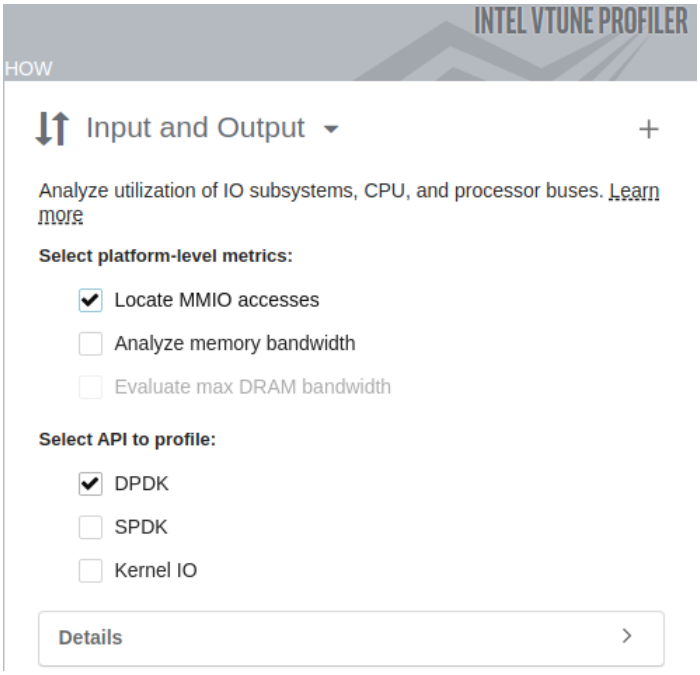


- **CPU**: Intel® Xeon® Platinum 8180 (38.5M Cache, 2.5 GHz, 28 cores)

Run Input and Output Analysis

To run DPDK analysis, select the and enable in the :

1. Open the VTune Profiler GUI.
2. From the Analysis Tree, select **Input and Output analysis**.
3. Under **Select API to profile**, select the **DPDK** option.



You can correlate API-specific metrics, such as DPDK Rx Spin Time, with the hardware events and hardware event-based metrics. For example, you can see the dependency between DPDK Rx Spin Time and PCIe bandwidth that can be collected when the **Locate MMIO accesses** option is enabled.

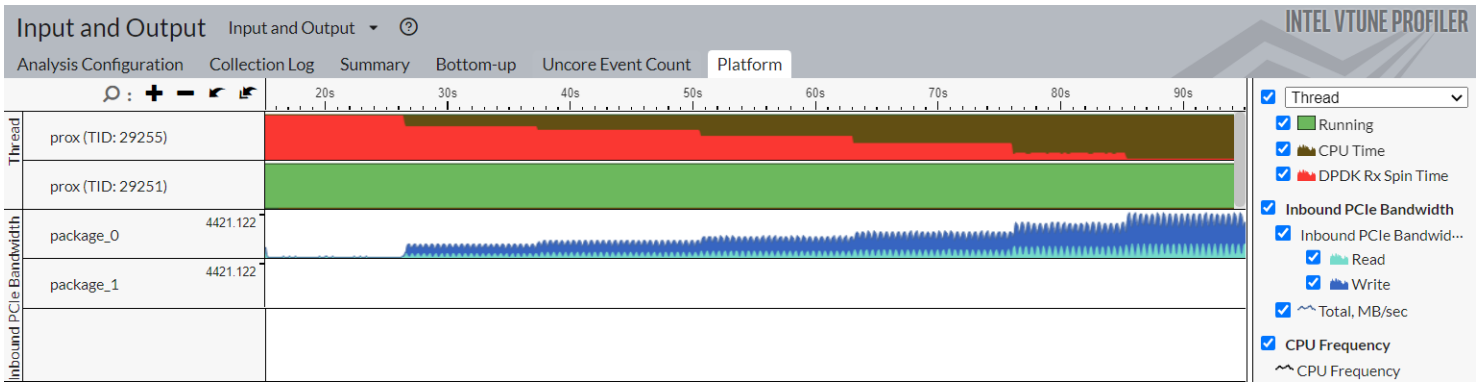
To run **Input and Output analysis** with PCIe bandwidth and DPDK metrics from command line, execute the following command as a root, which enables getting per-device PCIe bandwidth with human-readable names:

```
vtune-cl -collect io -knob kernel-stack=false -knob dpdk=true -knob mmio=true -knob collect-memory-bandwidth=false -knob dram-bandwidth-limits=false --target-process=testpmd
```

Analyze Core Utilization with the DPDK Rx Spin Time Metric

When the data is collected, start your analysis with the **Platform** tab and explore the **DPDK Rx Spin Time** overtime metric that refers to a thread. This metric shows (on a per-thread basis) a portion of `rte_eth_rx_burst(...)` function calls that return zero packets, which is identical to the fraction of polling loop iterations that provide no packets:

$$DPDK\ Rx\ Spin\ Time = \frac{num\ iterations\ that\ give\ 0\ packets}{total\ num\ iterations}$$



NOTE
The result demonstrated in this recipe is synthetic.

On the **Platform** view above, the **CPU Time** (brown) for the polling thread is always close to 100%. The **DPDK Rx Spin Time** (red) illustrates thread utilization in terms of packet retrieving. Hover the mouse over the charts to find values at each moment of time in the tooltip.

In this example, the traffic generator was automated to increase the traffic rate every two seconds by 5% of 40 Gbps and collect packet loss data. Overtime data written to a properly formatted `*.csv` file can be imported to a VTune Profiler project and visualized on its timeline.

By default, VTune Profiler cannot collect the **Packet Rate** and **Packet Loss** metrics displayed in the **Global Counters** section above. For this recipe, these metrics were collected separately and manually imported to the result collected by the VTune Profiler. As an alternative, you can use the custom collector feature in VTune Profiler to import a `csv` file with additional metrics. The custom collector is an additional process executed by VTune Profiler when the collection starts, stops, or pauses. You can use the custom collector to implement all the system automation and collect additional metrics. This makes the experiment reproducible and results valid for comparison, which is useful for consequential performance tuning.

At the bottom of the **Platform** view, you can see how the **Inbound PCIe Bandwidth** was changing over time. Since the analysis was run on the Intel microarchitecture code named Skylake with root privileges, PCIe Bandwidth is broken into PCIe devices with human readable names.

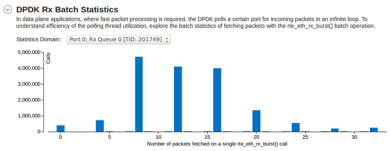
All metrics in the **Platform** view above are correlated. As the traffic generation rate grows, the **Inbound PCIe Bandwidth** increases and **DPDK Rx Spin Time** goes down. At some point, the test system gets overloaded and a non-zero **Packet Loss** value shows up.

NOTE
If a thread processes several Rx Queues, the **DPDK Rx Spin Time** metric will represent composite statistics.

Analyze Packets Retrieval with DPDK Rx Batch Statistics Histogram

DPDK uses the `rte_eth_rx_burst(...)` function to receive batches of packets from the NIC. It can retrieve any number of packets in the interval (0, `MAX_NB_PKTS`) , where `MAX_NB_PKTS` is a constant value (typically, 32). Hence, with the fixed **Rx Spin Time**, the core may process far different traffic, so **Rx Spin Time** does not represent a full picture.

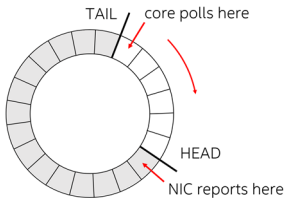
To see summary statistics for packet retrieving and get a full characterization of core utilization on Rx, switch to the **Summary** tab and explore the **DPDK Rx Batch Statistics** histogram:



This histogram represents statistics on receiving batch packets for the selected **Port / Rx Queue / TID** grouping. In this example, all the peaks show values multiple of 4. This is not a coincidence and the root cause investigation requires understanding the background of the packet receiving.

Understand Rx Operations and Investigate Rx Peaks

To receive packets, the working core communicates with the NIC through the Rx descriptors that are data structures keeping the information about the packet, such as its address, size, and so on. The Rx descriptors are joint into ring buffers called Rx Queues. In simple terms, the packet receiving is the race in the ring buffer, where the NIC fills in the Rx descriptors from the ring buffer **Head** and working core polls, processes and frees Rx descriptors coming from the **Tail**:



When the core frees Rx descriptors, it moves the Tail pointer forward. When the Tail reaches the Head, `rte_eth_rx_burst()` can return 0 packets. In the opposite case, when the Head reaches the Tail, there are no free Rx descriptors in the Rx Queue and packet loss may occur.

To deliver a new packet, the NIC reads the Rx descriptor in the Head of the Rx Queue and transfers the packet to the memory by the address specified by the core in the descriptor. Then, it has to write back the Rx descriptor to notify the core on the new packet arrival.

Intel® Ethernet Controller XL710, used in the recipe setup, supports 16 and 32 Byte Rx descriptors. Both are less than the cache line size, therefore the NIC has the descriptor write back policy denoting that NIC should coalesce writes by packing Rx descriptors into the integer number of cache lines to save PCIe bandwidth. Primarily, the XL710 writes back completed Rx descriptors when the following requirements are met:

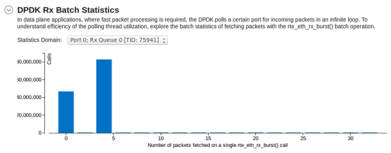
- **4 x 32 Byte** descriptors or **8 x 16 Byte** descriptors are completed.
- A descriptor is invalidated in the internal NIC cache.

Refer to the Intel Ethernet Controller X710/ XXV710/XL710 Datasheet for more details.

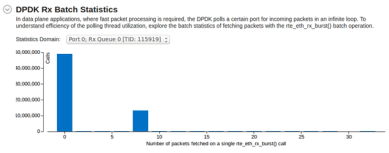
In this recipe, the system employed 32 Byte Rx descriptors. That is why most peaks of the **DPDK Rx Batch Histogram** mark values in multiple of 4.

DPDK allows toggling the Rx descriptor size. These **DPDK Rx Batch Histogram** diagrams describe the changes that happen when running `testpmd` with 32 and 16 Byte Rx descriptors under medium load:

- 32 Byte Rx descriptor: Most of `rte_eth_rx_burst()` calls receive 4 packets.



- 16 Byte Rx descriptor: Most of `rte_eth_rx_burst()` calls receive 8 packets.



See Also

PCIe Traffic in DPDK Apps This recipe introduces PCIe Bandwidth metrics used in Intel® VTune™ Profiler to explore the PCIe traffic for a packet forwarding DPDK-based workload.

Use a Custom Collector

Create a CSV File with External Data

External Data Import

PCIe Traffic in DPDK Apps

This recipe introduces PCIe Bandwidth metrics used in Intel® VTune™ Profiler to explore the PCIe traffic for a packet forwarding DPDK-based workload.

Contact Expert: Jeffrey Reinemann

Data plane applications running on systems with 10/40 GbE NICs usually highly utilize platform I/O capabilities, in particular, intensively consume the bandwidth of the PCIe link that is an interface between the CPU and Network Interface Card (NIC). For such workloads, it is critical to effectively utilize the PCIe link by keeping the balance between packet transfers and control communications. Understanding PCIe transfers helps locate and fix performance issues.

For detailed methodology of the PCIe performance analysis for DPDK-based workloads, see Benchmarking and Analysis of Software Data Planes.

In this recipe, you can explore the stages of packet forwarding with DPDK and theoretical estimations for PCIe bandwidth consumption. Then, you can compare the theoretical estimations with the data collected with Intel® VTune™ Profiler.

1. INGREDIENTS
2. DIRECTIONS:
   a. Understand Inbound/Outbound PCIe Bandwidth metrics.
   b. Configure and run Input and Output analysis.
   c. Understand PCIe transfers required for packet forwarding.
   d. Understand PCIe Traffic optimizations.
   e. Estimate PCIe Bandwidth consumption.
   f. Compare PCIe Bandwidth vs Packet Rate.

Ingredients

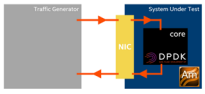
This section lists the hardware and software tools used for the recipe.

- **Application**: DPDK `testpmd` app running on one core and performing L2 forwarding. The application is compiled against DPDK with profiling enabled by Intel® VTune™ Profiler.
- **Performance analysis tools**:
  - Intel® VTune™ Profiler 2024 (or newer): Input and Output analysis.

  NOTE

  - Starting with the 2020 release, Intel® VTune™ Amplifier has been renamed to Intel® VTune™ Profiler.
  - Most recipes in the Intel® VTune™ Profiler Performance Analysis Cookbook are flexible. You can apply them to different versions of Intel® VTune™ Profiler. In some cases, minor adjustments may be required.
  - Get the latest version of Intel® VTune™ Profiler:
    - From the Intel® VTune™ Profiler product page.
    - Download the latest standalone package from the Intel® oneAPI standalone components page.

- **System setup**: a traffic generator and a system under test, where the `testpmd` app performs packet forwarding and where Intel® VTune™ Profiler collects performance data.



- **CPU**: Intel® Xeon® Platinum 8180 (38.5M Cache, 2.5 GHz, 28 cores)

Understand Inbound / Outbound PCIe Bandwidth Metrics

PCIe transfers may be initiated by both the PCIe device (for example, NIC) and the CPU. So, Intel® VTune™ Profiler distinguishes PCIe bandwidth metrics for the following bandwidth types:

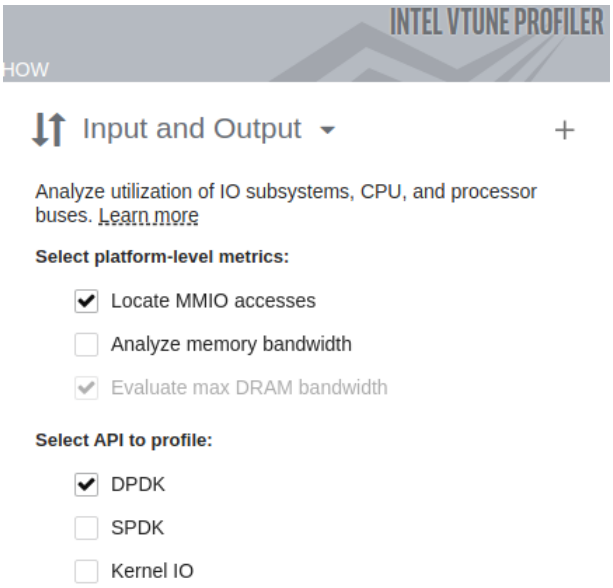
- **Inbound PCIe Bandwidth** caused by device transactions targeting the system memory
  - **Inbound Reads** show device reads from the memory
  - **Inbound Writes** show device writes to the memory
- **Outbound PCIe Bandwidth** caused by CPU transactions targeting device's MMIO space
  - **Outbound Reads** show CPU reads from device's MMIO space
  - **Outbound Writes** show CPU writes to the device's MMIO space

Configure and Run Input and Output Analysis

To collect **Inbound** and **Outbound PCIe Bandwidth** data, use the Input and Output analysis.

Run Analysis from GUI:

1. Create a new project in VTune Profiler.
2. In the **HOW** pane, select **Input and Output** analysis.
3. In **Select platform-level metrics**, select **Locate MMIO accesses**.
4. In **Select API to profile**, select **DPDK**.



5. Click the **Start** button.

Run Analysis from Command Line:

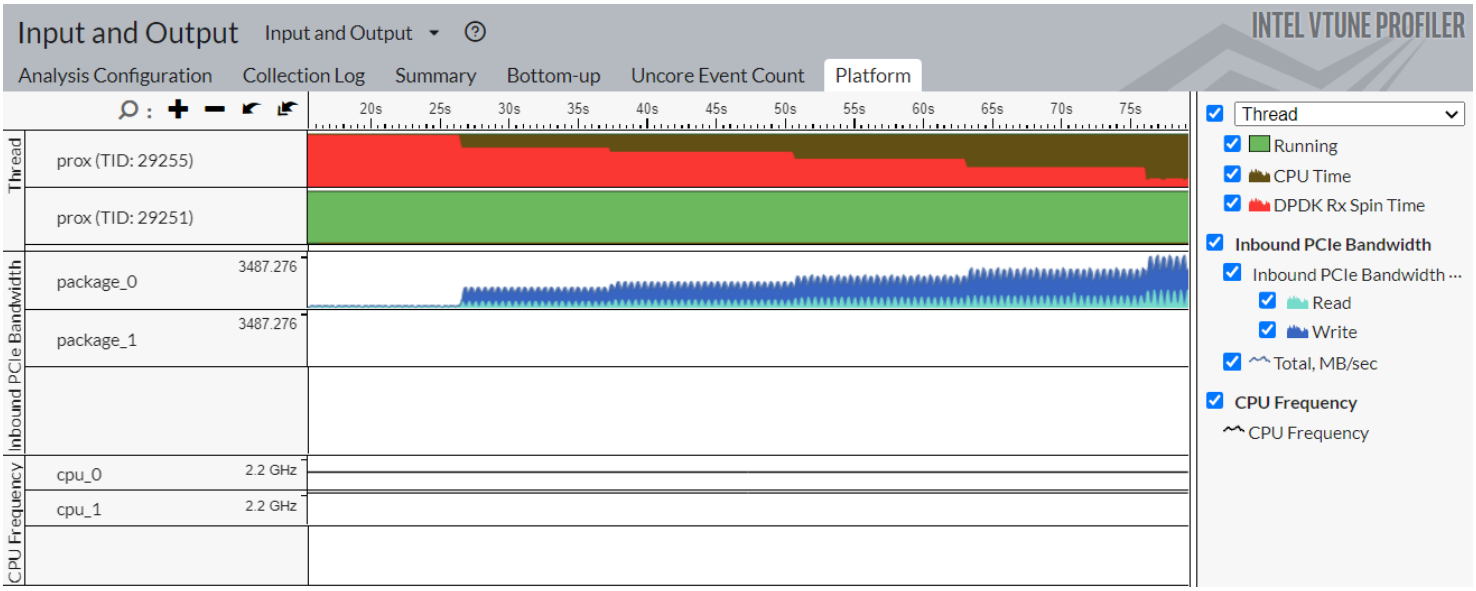
When running this analysis from the command line, use the `collect-pcie-bandwidth` knob. By default, this knob is set to `true`.

The following command starts the collection of PCIe Bandwidth data along with DPDK metrics:

```
vtune -collect io -knob kernel-stack=false -knob dpdk=true -knob mmio=true -knob collect-memory-bandwidth=false --target-process my_process
```

Once the results display in the Intel® VTune™ Profiler GUI, open the **Platform** tab. Focus on the **Inbound** and **Outbound PCIe Bandwidth** sections.

> **NOTE**
> Starting with server platforms based on the Intel microarchitecture code name Skylake, you can collect PCIe Bandwidth metrics per-device. You must have root privileges.



Understand PCIe Transfers Required for Packet Forwarding

Packet forwarding with DPDK implies receiving a packet (`rx_burst` DPDK routine) followed by transmitting the packet (`tx_burst`). The Core Utilization in DPDK Apps recipe describes details of packet receiving by means of Rx queue containing Rx descriptors. Packet transmitting with DPDK works similarly to packet receiving. To transmit packets, a working core employs Tx descriptors - the 16-Byte data structures that store a packet address, size, and other control information. The buffer of Tx descriptors is allocated by the core in the contiguous memory and is called Tx queue. Tx queue is handled as a ring buffer and is defined by its length, head, and tail. Packet transmitting from the Tx queue perspective is very similar to the packet receiving: the core prepares new Tx descriptors at the Tx queue tail, and the NIC processes them starting from the head.

For both Rx and Tx queues, the tail pointers are updated by the software to notify the hardware that new descriptors are available. The tail pointers are stored in the NIC registers that are mapped to the MMIO space. So, the tail pointers are updated through Outbound Writes (MMIO Writes). MMIO address space is uncacheable, so Outbound Writes, and especially Outbound Reads, are very expensive transactions and, therefore, such transfers should be minimized.

For packet forwarding, PCIe transactions go through the following workflow:

1. The core prepares the Rx queue and starts polling the Rx queue tail.
2. The NIC reads an Rx descriptor in the Rx queue head **(Inbound Read)**.
3. The NIC delivers the packet to the address specified in the Rx descriptor **(Inbound Write)**.
4. The NIC writes back the Rx descriptor to notify the core that the new packet arrived **(Inbound Write)**.
5. The core processes the packet.
6. The core frees the Rx descriptor and moves the Rx queue tail pointer **(Outbound Write)**.
7. The core updates the Tx descriptor in the Tx queue tail.
8. The core moves the Tx queue tail pointer **(Outbound Write)**.
9. The NIC reads the Tx descriptor **(Inbound Read)**.

10. The NIC reads the packet **(Inbound Read)**.
11. The NIC writes back the Tx descriptor to notify the core that the packet is transmitted and the Tx descriptor can be freed **(Inbound Write)**.

Understand PCIe Traffic Optimizations

To increase the maximum packet rate and reduce the latency, the DPDK uses the following optimizations:

- **No Outbound Reads.** No expensive Outbound Reads (MMIO Reads) are needed to understand Rx and Tx queues head position. Instead, the NIC writes back Rx and Tx descriptors to notify software that the head position moves.
- **Decreased Inbound Write Bandwidth related to the Tx descriptors.** Tx descriptor write back is required to notify the core where the Tx queue head is and which Tx descriptors can be reused. In case of packet receiving, it is critical to write back each Rx descriptor to notify the core about a new arrived packet as soon as possible. In packet transmitting, there is no need to write back each Tx descriptor. It is sufficient to notify the core about successful packet transmission periodically (for example, on every 32nd packet), which would mean that all previous packets are transmitted successfully too. The NIC writes back the Tx descriptor when the RS (Report Status) bit of the Tx descriptor is set. On the DPDK side, there is a *RS bit threshold*; its value defines how frequently the RS bit is set and thus how frequently the NIC writes back Tx descriptors. This optimization amortizes Inbound Writes related to the Tx descriptors.
- **Amortized Outbound Writes**. The DPDK performs packet receiving and transmitting in batches, and application updates tail pointers after a batch of packets has been processed. Some implementations of `rx_burst` use the *Rx free threshold*. This threshold enables setting the number of Rx descriptors processed before the app updates the Rx queue tail pointer (note that threshold becomes effective only when it is greater than the batch size). That way, the Outbound Writes are averaged among a number of packets.

Besides, at the platform level all the transactions are performed at the cache line granularity, so the hardware always tries to coalesce reads and writes to avoid partial cache line transfers.

Estimate PCIe Bandwidth Consumption

For the packet forwarding case with optimizations described above, you can apply the following equations estimating PCIe bandwidth consumption at the given packet rate:

$$Inbound_Write = Pkt_Rate \cdot \left(Rx_Desc_Size + Pkt_Size + \frac{Tx_Desc_Size}{rs_bit_threshold} \right)$$

$$Inbound_Read = Pkt_Rate \cdot \left(Rx_Desc_Size + Pkt_Size + Tx_Desc_Size \right)$$

$$Outbound_Write = Pkt_Rate \cdot \left(\frac{Rx_Tail_Ptr_Size}{\max(rx_free_threshold,\ rx_batch_size)} + \frac{Tx_Tail_Ptr_Size}{tx_batch_size} \right)$$

$$Outbound_Read = 0$$

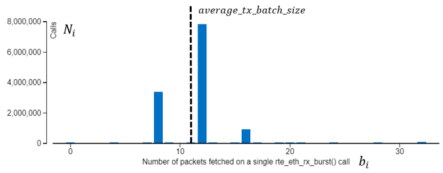
The equation for Outbound Write Bandwidth above works only when the packets are processed with the batches of the same size. This formula should be more accurate if packets are transmitted with batches of multiple sizes.

In the simple forwarding case, the core transmits all the received packets. The `testpmd` is an application designed within run-to-completion model, so you can assume that `tx_burst` operates with the same batches of packets as `rx_burst` does. In other words, the **Rx Batch Histogram** (see the Core Utilization in DPDK Apps recipe) reflects the statistics of both packet receiving and packet transmitting. Therefore, you can use the **Rx Batch Histogram** to estimate Outbound Write Bandwidth in a generic case.

Instead of the Tx batch size consider an "average" Tx batch size:

$$average_tx_batch_size = \sum_i b_i \frac{n_i}{n} = \frac{\sum_i b_i^2 N_i}{\sum_i b_i N_i}$$

where b_i is the batch size, N_i - the number of `rx_burst` calls with batch size b_i, $n_i = b_i N_i$ – the number of packets in the ith peak of the **Rx Batch Histogram** and $n = \sum_i b_i N_i$ is the total number of packets forwarded. The picture below illustrates this calculation. For this example, the batch histogram has 3 peaks with batch sizes of 8, 10 and 12, and the calculation provides an average batch size equal to 11.

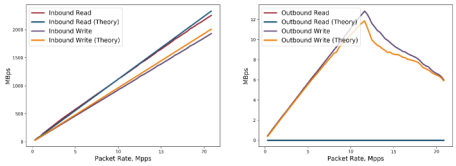


For simplicity, consider Rx free thresholds greater that maximal Rx batch size. Then, the final equation for Outbound Write bandwidth is the following:

$$Outbound_Write = Pkt_Rate \cdot \left(\frac{Rx_Tail_Ptr_Size}{rx_free_threshold} + \frac{Tx_Tail_Ptr_Size}{average_tx_batch_size} \right)$$

Compare Estimations vs. Analysis

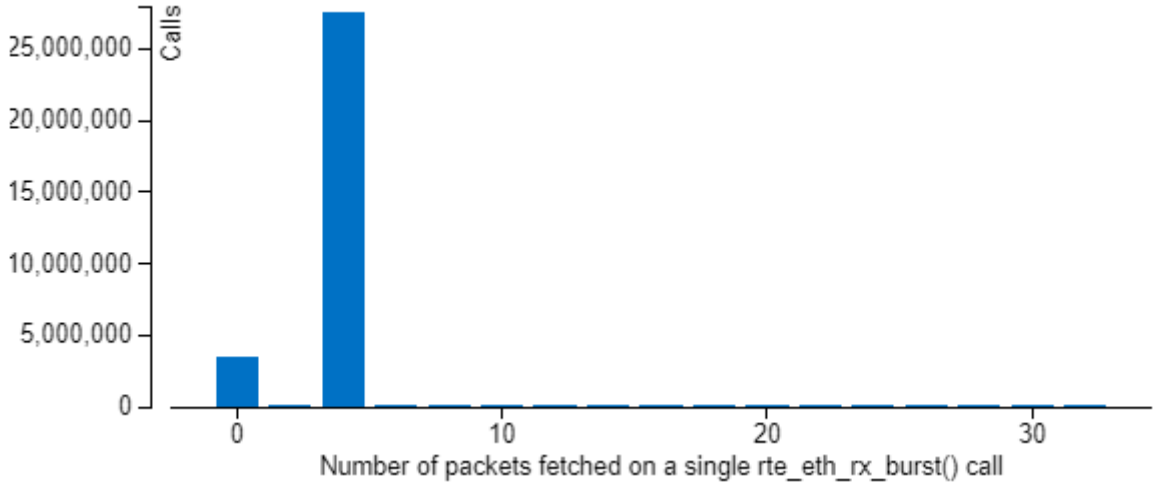
The charts illustrate a theoretical estimation for PCIe bandwidth and the PCIe bandwidth collected with Intel® VTune™ Profiler for the `testpmd` app configured as listed in the table below.



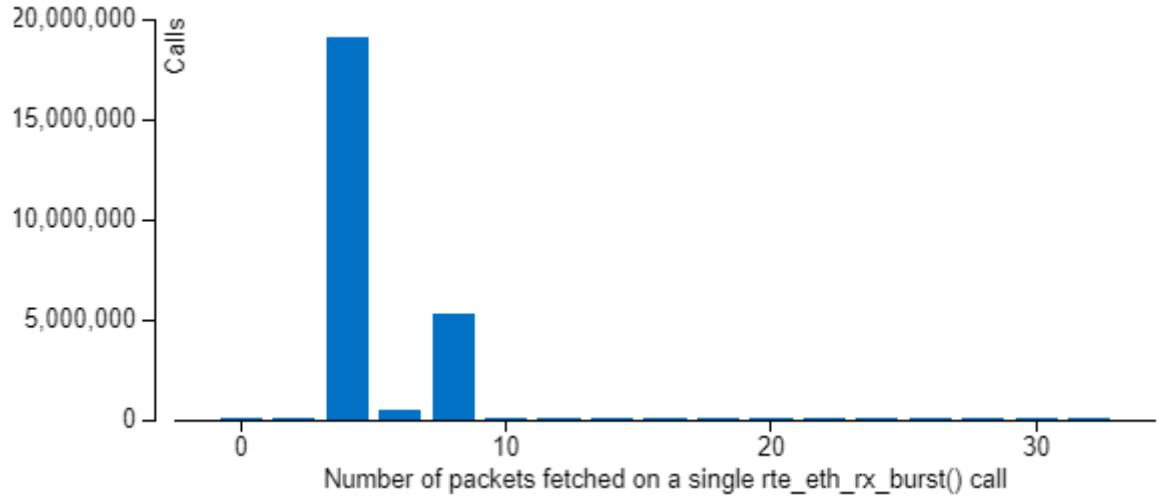
Packet Size, B	64
Rx Descriptor Size, B	32
RS Bit Threshold	32
Rx Free Threshold	32

The fracture in the middle of the **Outbound Write** dependency looks interesting. This drop is a consequence of the **Batch Histogram** modification caused by processing of the increased packet rate. Before this point, the **Batch Histogram** has only two peaks - with batch sizes 0 and 4, and at this point a new peak of 8 appears (see the histograms below). According to the equations listed above, this increases an average batch size and leads to reducing the **Outbound Write Bandwidth**.

For 10 Mpps:



For 13 Mpps:



In general, theoretical estimations look very close to the data reported by Intel® VTune™ Profiler, though there are some deviations that may be caused by effects that are not taken into account in equations.

The data plane application used in this recipe is already well-optimized; however, the demonstrated recipe is a solid starting point for I/O-centric performance analysis of real world application.

See Also

Toggling the 16/32 Byte Rx descriptor for i40e driver
Available thresholds and how to change them in testpmd
Benchmarking and Analysis of Software Data Planes

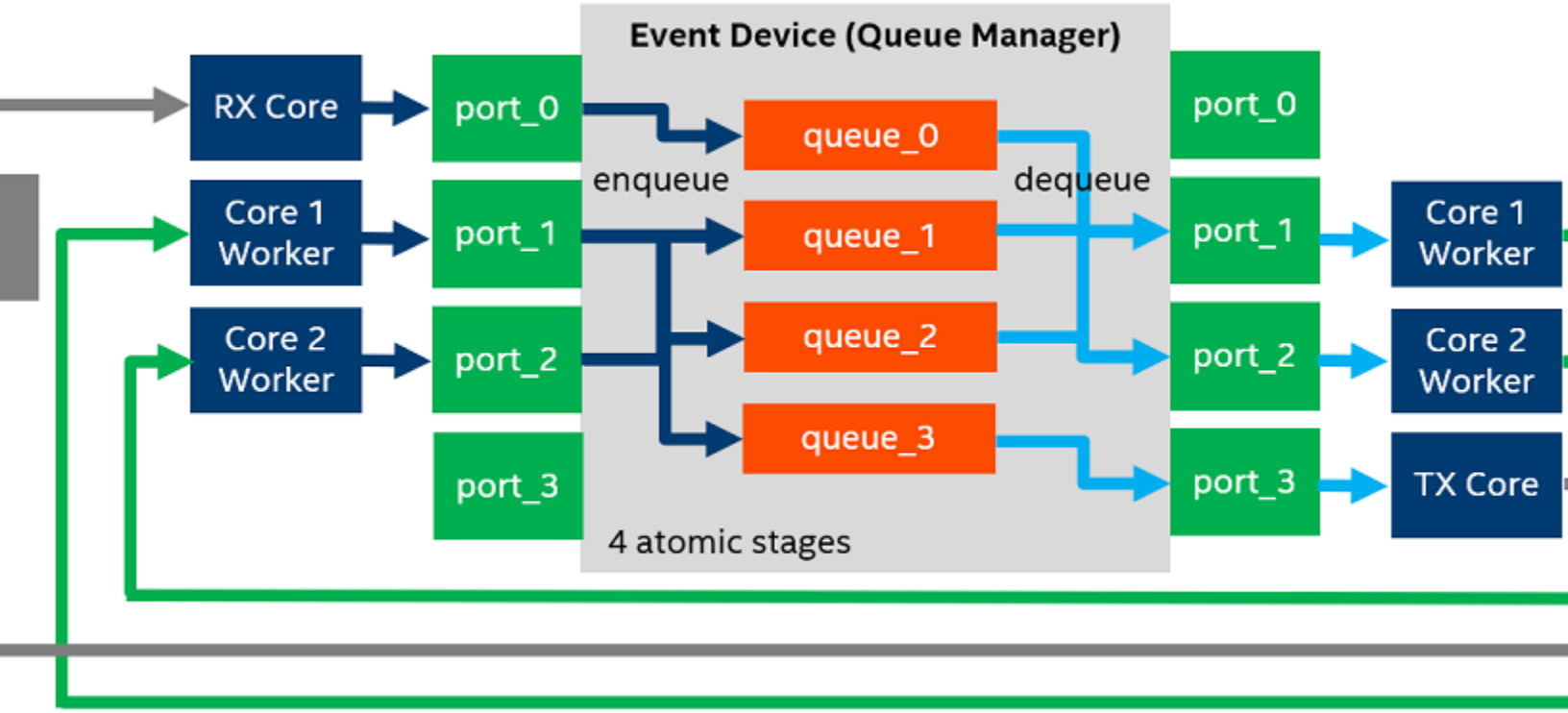
DPDK Event Device Profiling

Use Intel® VTune™ Profiler to analyze the efficiency of DPDK Event Device pipeline utilization in your DPDK-based application and identify issues, such as inhomogeneous load distribution and worker core underutilization.

Content Experts: Eugeny Parshutin, Jeffrey Reinemann

The Data Plane Development Kit (DPDK) is a framework that consists of libraries that accelerate packet processing workloads running on a wide variety of CPU architectures. One of these libraries is the `eventdev` library that enables you to improve system load balancing by using an event-based model in your application. An event-based approach suggests that the work that needs to be done by the system is presented as separate units called events. One common example of using the event-based programming model in DPDK is the network packet processing pipeline, where each packet plays the role of an event.

This figure gives an example of an `eventdev` pipeline configuration:



Here, each block represents the following unit:

- **Event Device** – device with event scheduling capability, implemented either in hardware or software.
- **Queue** – logical stage of the processing pipeline that contains events of different flows associated with scheduling types (atomic, ordered, or parallel).
- **Ports** – points of contact between cores and the `eventdev` library that are used to enqueue and dequeue events to and from `eventdev` queues.
- **Worker Cores** – CPU cores that are available to the application to perform work.
- **Rx Core** – CPU core that receives packets from the NIC.
- **Tx Core** – CPU core that transmits packets to the NIC.
- **NIC** – Network Interface Card.

This example demonstrates an Event Device that is configured to manage four atomic stages that are presented as four event queues:

- **queue_0** is dedicated to keep newly arrived packets. Only the Rx core enqueues packets (events) to this queue.
- **queue_1** and **queue_2** are dedicated to some type of event processing stage, such as setting destination address, cryptography processing, or compression. Worker cores perform these tasks and transfer packets between queues 0, 1, 2, and 3.
- **queue_3** is dedicated to keep packets that are ready to be transmitted. Only the Tx core dequeues packets from this queue.

The dequeue operation is performed using the `rte_event_dequeue_burst()` routine in an endless loop. Thus, worker cores continuously poll Event Device ports, looking for a batch of events to be processed. The batch size depends on overall load and performance of different stages. The maximum batch size is defined by the workload.

Per-worker dequeue statistics provided by Intel® VTune™ Profiler reveal the load balancing details and enable you to analyze pipeline configuration efficiency and identify pipeline bottlenecks.

This recipe defines the following steps to analyze the efficiency of the pipeline processing model in DPDK-based applications:

- Ingredients
- Directions
  - Run Input and Output analysis
  - Analyze load per stage
  - Analyze CPU utilization

Ingredients

This section lists the hardware and software tools used in this performance analysis scenario:

- **Application:** the DPDK `eventdev_pipeline` application demonstrates the usage of the `eventdev` API and shows how an application can configure a pipeline and assign a set of worker cores to perform event processing. The application is compiled with DPDK with VTune Profiler support enabled.
- **Tools:**
  - DPDK, compiled with VTune Profiler support enabled. To enable `eventdev` profiling on DPDK side, you need to apply a patch and recompile DPDK and the target DPDK application.

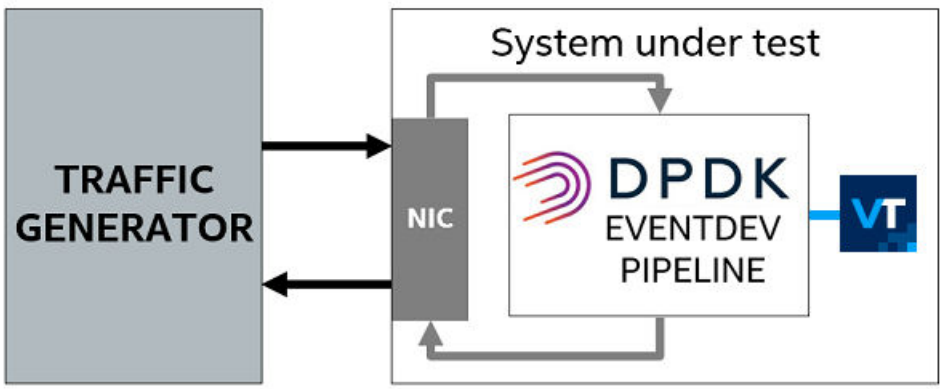
    Use the following patches:

    - For DPDK versions older than 20.11
    - For DPDK versions 20.11 and newer
  - Intel® VTune™ Profiler (version 2024 or newer): Input and Output analysis

NOTE

- Starting with the 2020 release, Intel® VTune™ Amplifier has been renamed to Intel® VTune™ Profiler.
- Most recipes in the Intel® VTune™ Profiler Performance Analysis Cookbook are flexible. You can apply them to different versions of Intel® VTune™ Profiler. In some cases, minor adjustments may be required.
- Get the latest version of Intel® VTune™ Profiler:
  - From the Intel® VTune™ Profiler product page.
  - Download the latest standalone package from the Intel® oneAPI standalone components page.

- **System Setup:**
  - **Traffic generator:** a system that generates traffic for the system being tested.
  - **System under test:** a system running the `eventdev_pipeline` application for packet (event) processing and VTune Profiler for performance data collection.



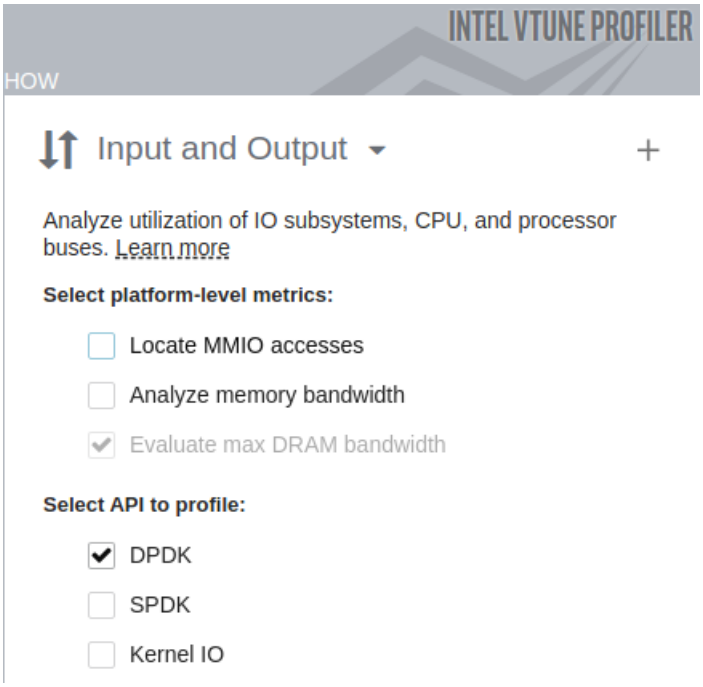
- **CPU:** Intel® Xeon® Platinum 8168 processor (formerly code named Skylake).
- **Operating System:** Linux* OS.

Run Input and Output Analysis

To collect DPDK `eventdev` dequeue statistics, run the **Input and Output** analysis in VTune Profiler.

Run Analysis from GUI:

1. Create a new project in VTune Profiler.
2. In the **HOW** pane, select **Input and Output** analysis
3. In **Select API to profile**, select **DPDK**



4. Click the **Start** button.

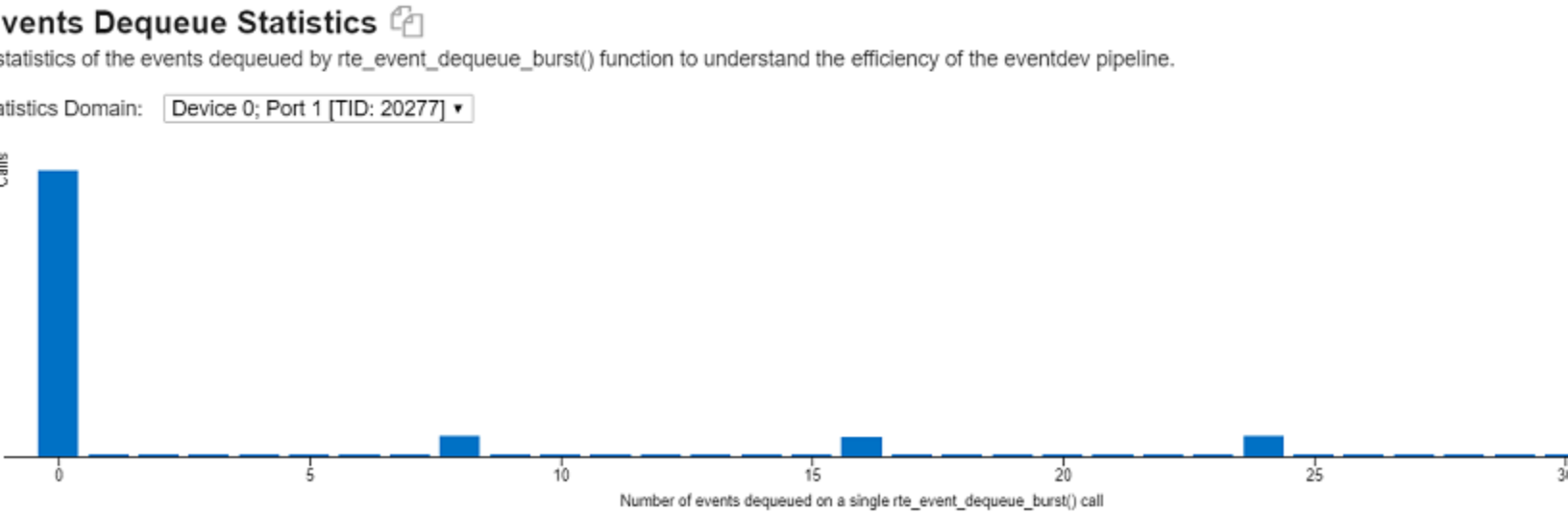
Run Analysis from Command Line:

Use this command:

```
vtune -collect io -knob kernel-stack=false -knob dpdk=true --target-process=eventdev_pipeline
```

Analyze Load per Stage

To get an overall characterization of DPDK `eventdev` pipeline utilization, start your investigation with the **Summary** tab and explore the **DPDK Events Dequeue Statistics** histogram:

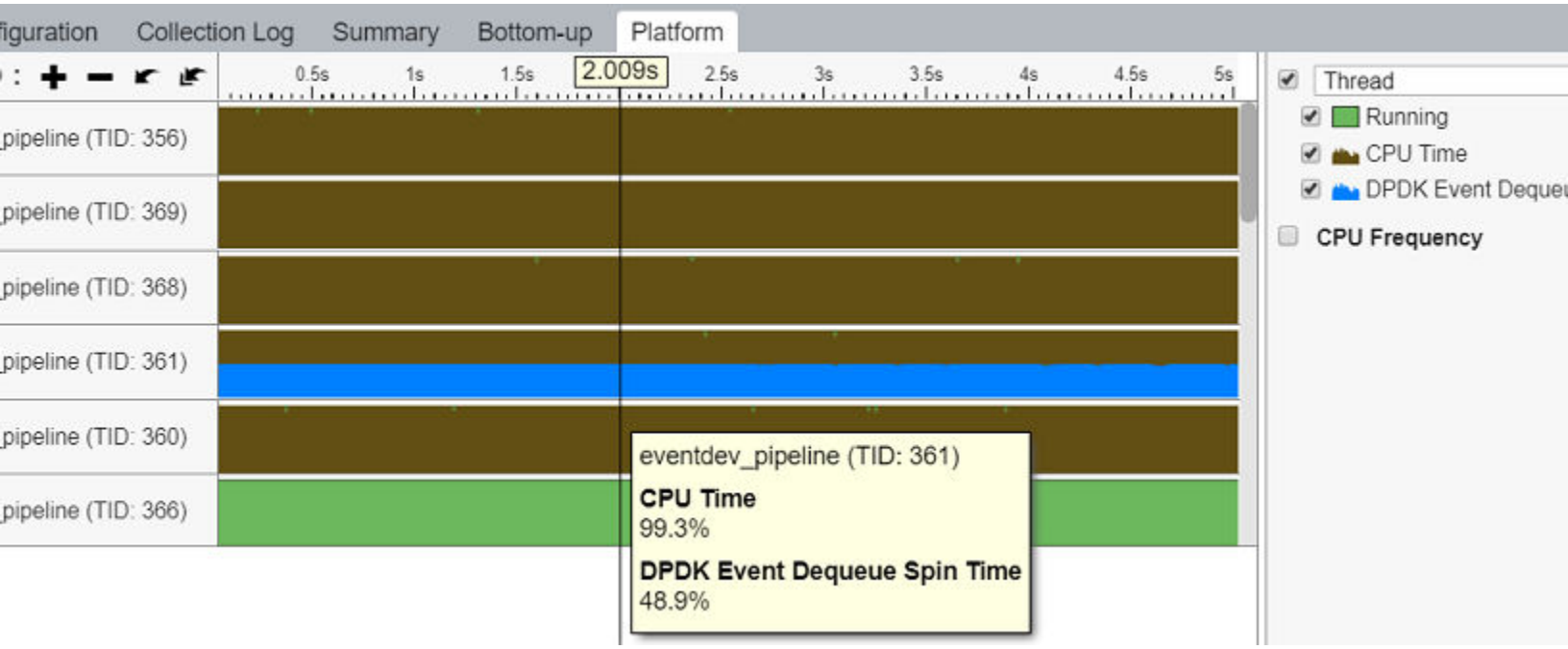


This histogram represents the statistics for the number of dequeued events for each `eventdev` port, that is, for each worker thread that polls the event device. Explore the different areas of the histogram to identify inhomogeneous load distribution, oversubscribed, or underutilized workers.

If you identify any imbalance in worker thread load distribution, try to reconfigure your pipeline to avoid this an re-run the analysis.

Analyze CPU Utilization

To understand the CPU utilization for workers performing event dequeue operations, navigate to the **Platform** tab and explore the **DPDK Event Dequeue Spin Time** overtime metric attributed to worker threads.



The **DPDK Event Dequeue Spin Time** per-thread metric shows the ratio of empty dequeue cycles, which is the ratio of `rte_event_dequeue_burst()` calls that returned zero events with respect to the total number of dequeue calls. Explore this metric to estimate worker thread load and to decide whether the application underutilizes cores or needs more resources.

See Also

Cookbook: Core Utilization in DPDK Apps
Cookbook: PCIe Traffic in DPDK Apps
DPDK Event Device library
Introduction to the Data Plane Development Kit (DPDK) Eventdev Library
Benchmarking and Analysis of Software Network Data Planes

Effective Utilization of Intel® Data Direct I/O Technology

Use Intel® VTune™ Profiler to understand the utilization efficiency of Intel® Data Direct I/O (Intel® DDIO) technology, a hardware feature of Intel® Xeon® processors.

Content Expert: Alexander Antonov, Eugeny Parshutin

Traditionally, inbound PCIe transactions target the main memory. Data movement from the I/O device to the consuming core requires multiple DRAM accesses. For I/O-intensive use cases like software data planes, this scheme becomes inapplicable.

For example, when a 100G NIC is fully utilized with 64B packets and 20B Ethernet overhead, a new packet arrives every 6.72 nanoseconds on average. If any component on the packet path takes longer than this time to process the individual packet, a packet loss occurs. For a core running at 3GHz, 6.72 nanoseconds only accounts for 20 clock cycles, while the DRAM latency is 5-10 times higher, on average. This is the main bottleneck of the traditional DMA approach.

The Intel® DDIO technology in Intel® Xeon® processors eliminates this bottleneck. Intel® DDIO technology allows PCIe devices to perform read and write operations directly to and from the L3 cache, or the last level cache (LLC). This places the incoming data as close to the cores as possible. With proper use of Intel DDIO technology, the core and I/O device interactions are completely served by using the L3 cache. This results in some benefits:

- There is no longer a need for DRAM accesses.
- There are low inbound read and write latencies that allow for high throughput.
- The DRAM bandwidth and power consumption are both reduced.

Though Intel® DDIO is a hardware feature that is always enabled and transparent to software, improper use can cause suboptimal performance. To optimize the use of Intel® DDIO, there are two main software tuning opportunities:

- **Topology configuration:** For a system with multiple sockets, it is critical that all of the following are on the same NUMA mode:
  - I/O device
  - Core that interfaces with the I/O device
  - Memory
- **L3 cache management:** Advanced tuning helps to optimize the usage of L3 cache by keeping the necessary data available in the L3 cache at the right time.

This recipe demonstrates how you use the Input and Output analysis in VTune Profiler to detect inefficiencies in the use of Intel® DDIO technology.

- Ingredients

Directions:

- Understand the architectural background
- Analyze Intel® DDIO traffic
- Understand typical examples
  - Remote socket accesses
  - Poor L3 cache management
- Key take-aways

Ingredients

- **System:** A two-socket 2nd Generation Intel® Xeon® Scalable processor-based system.
- **Application:** DPDK `testpmd` application configured to run on a single core and to perform packet forwarding using one port of one 40G network interface card attached to Socket one.
- **Performance Analysis Tool:**Intel® VTune™ Profiler version 2024 or newer (Input and Output analysis)

Understand the Architectural Background

In Intel® Xeon® Scalable processors, the L3 cache is a resource that is shared between all cores and all Integrated I/O controllers (IIO) within a single socket. Data transfers between the L3 cache and the cores are performed with cache line granularity (64B). When a PCIe device makes a request to the system memory, IIO translates this request into one or multiple cache line requests and issues them to the L3 cache on the local socket. A request to local L3 cache can happen in these ways:

- **Inbound PCIe write request**
  - **Inbound PCIe write L3 hit:** In the ideal case, an address targeted by a write request is already cached in the local L3. The cache line in the L3 is then overwritten with the new data.
  - **Inbound PCIe write L3 miss:** The scenario you want to avoid has an address that is targeted by the write request and is not cached in the local L3. In this case, a cache line is first evicted from an L3 way dedicated for I/O data. This could lead to DRAM write-back if the evicted line was in a dirty state. Then, in place of the evicted line, a new cache line is allocated. If the targeted cache line is cached remotely, cross-socket accesses through the Intel® Ultra Path Interconnect (Intel® UPI) are required to enforce coherency rules and to complete the cache line allocation. Finally, the cache line is updated with the new data.
- **Inbound PCIe read request**
  - **Inbound PCIe read L3 hit:** In the ideal case, an address targeted by a read request is cached in the local L3. The data is read and sent to the PCIe device.
  - **Inbound PCIe read L3 miss:** In the scenario you want to avoid, an address targeted by a read request is not cached the local L3. In this case, the data is read from the local DRAM or from the remote socket's memory subsystem. No local L3 allocation is performed.

For the 1st and 2nd Generation Intel® Xeon® Scalable processors, the **Input and Output analysis** in VTune Profiler provides Intel® DDIO utilization efficiency metrics including:

- **L3 hit/miss ratios**
- **Average latencies** of inbound PCIe reads and writes

This analysis also supports data breakdown by groups of PCIe devices. These groups are defined by M2PCIe units, which are the interfaces between the IIO controller and the mesh.

Analyze Intel DDIO Traffic

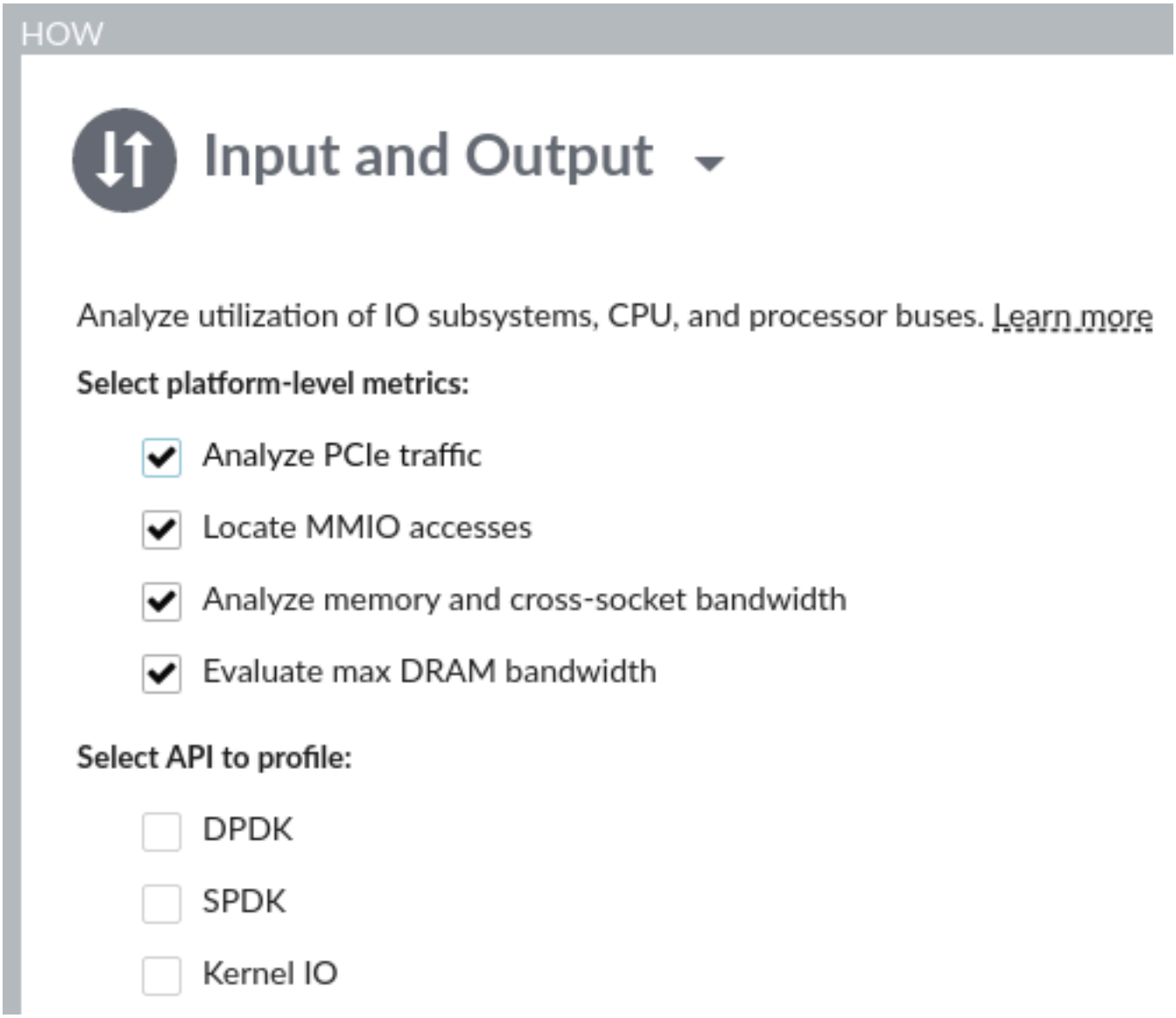
Before you begin:

- Make sure to use 1st or higher generations of Intel® Xeon® Scalable processors.
- Check to confirm that the sampling driver is loaded.
- Plan to collect data for at least 20 seconds.

This procedure uses the **Input and Output analysis** in VTune Profiler to collect Intel DDIO utilization efficiency metrics. When you run the analysis in driverless mode, there may be some limitations.

1. In the **WHAT** pane, do one of the following:
   - Select **Launch Application** and specify the path to the application and any parameters.
   - Select **Attach to Process** and specify the PID

   You can also use the **Automatically stop collection after (sec)** option to control the collection time.
2. In the **HOW** pane, select the **Input and Output** analysis.
3. Check the **Analyze PCIe traffic** checkbox to collect Intel DDIO utilization efficiency metrics.



4. Click the  **Start** button to run the analysis.

Understand Typical Examples

To understand typical inefficient usages of Intel DDIO technology and the capabilities of VTune Profiler that help to highlight them, run the DPDK `testpmd` application on a two-socket system equipped with 2nd Generation Intel® Xeon® Scalable processors. The application is configured to run on a single core and to perform packet forwarding using one port of a single 40G NIC attached to Socket 1.

The traffic generator injects 64B packets into the system with a packet rate that is much higher than what a single core can process. The optimization criterion is the system throughput - higher throughput indicates increased optimization . For this configuration, the experiment identifies the single core application throughput.

As a single baseline for two examples below, use a configuration where a core from Socket 1 performs packet forwarding. This configuration is called local, since the core and PCIe device reside on the same socket.

```
# ./testpmd -n 4 -l 24,25 -- -i
testpmd> set fwd mac retry
testpmd> start
```

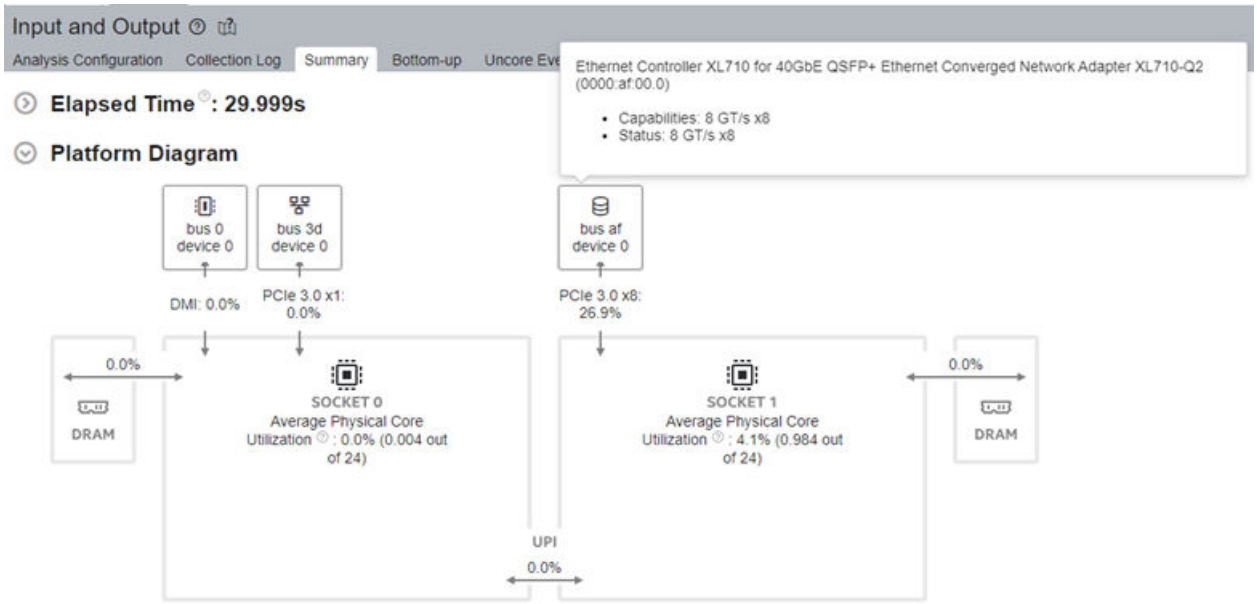
Use the `mac` forwarding mode. In this mode, a core changes the source and the destination Ethernet addresses of packets. Thus the forwarding core accesses packet descriptors and touches each packet.

Run the **Input and Output analysis**. Once the results display, look at the **Platform Diagram** in the **Summary** window.

The Platform Diagram shows the system topology and indicates an average utilization of hardware resources:

- Physical cores by computations of the workload being analyzed
- DRAM
- Intel® UPI and PCIe links

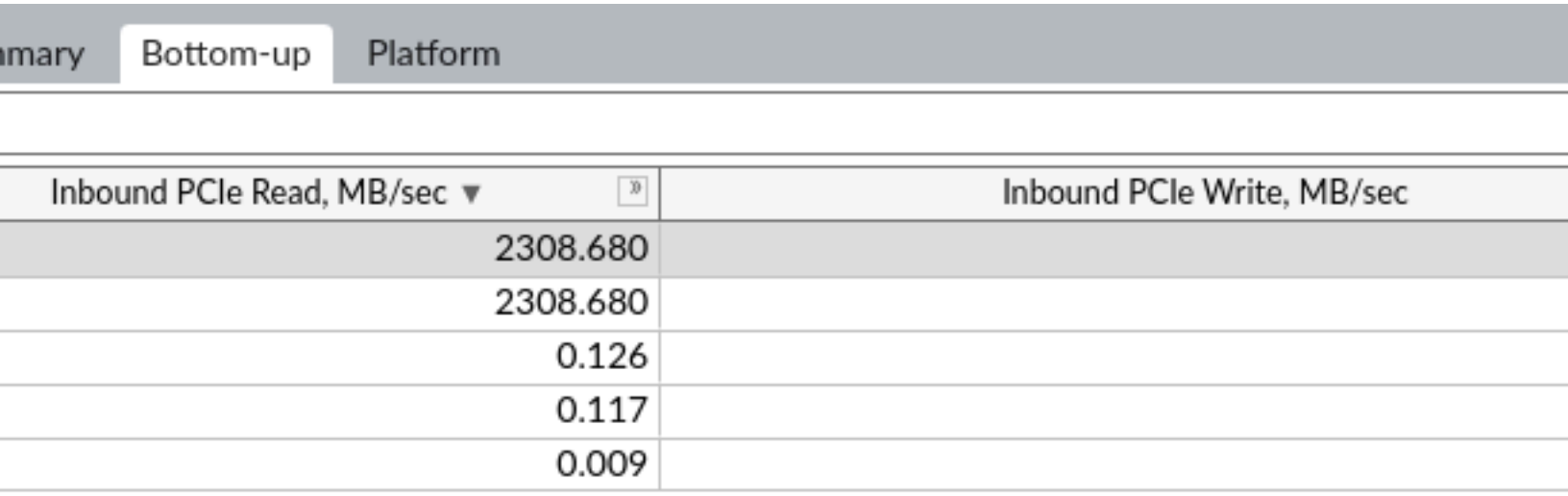
Note that the metric presented for PCIe devices shows the **Effective link utilization**. This value is calculated as the portion of the physical bandwidth consumed on transferring payloads. The overhead is not considered. For more information on this metric, see Input and Output analysis.



In the **Summary** tab, see the **PCIe Traffic Summary** section. You see the total inbound and outbound PCIe read and write traffic as first-level metrics. Intel DDIO utilization efficiency indicators appear as second-level metrics:

- **L3 Hit/Miss Ratios** represent a portion of inbound requests that hit/miss the L3 cache.
- **Average Latency** shows the average amount of time the platform spent on handling inbound requests for a cache line.
- **Core/IO Conflicts** metric shows the ratio of inbound writes that experience cache line contentions. When detected, VTune Profiler suggests a possible tuning direction.

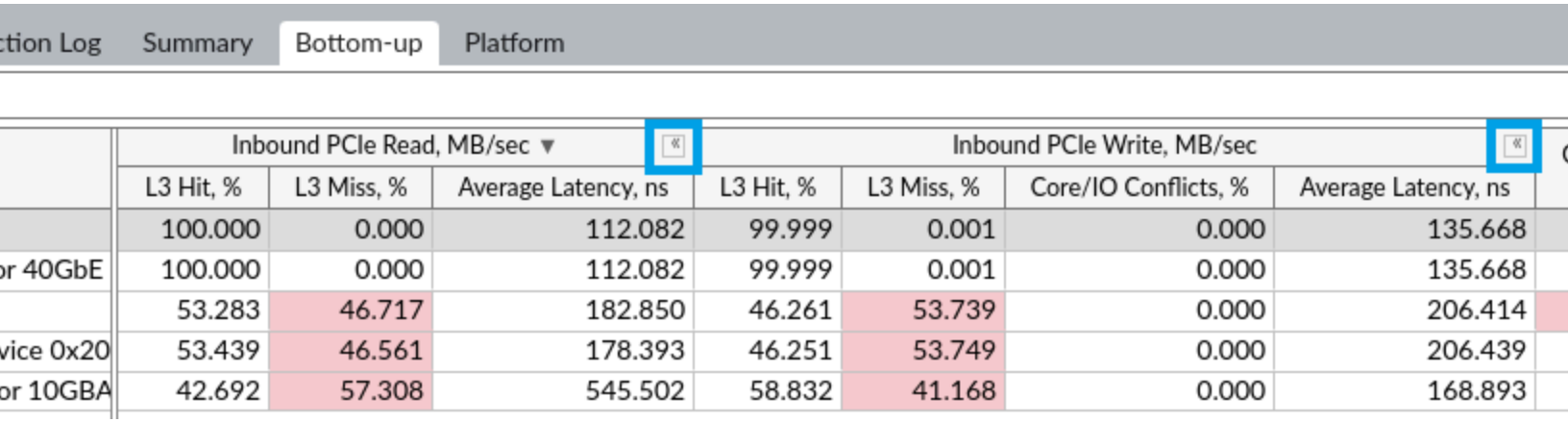
To see detailed IO metrics, click any metric in the **PCIe Traffic Summary** section to switch to the **Bottom-up** pane:



Inbound PCIe Read, MB/sec ▼	Inbound PCIe Write, MB/sec
2308.680	
2308.680	
0.126	
0.117	
0.009	

To see PCIe metrics, select the **Package/M2PCIe** grouping in the **Grouping** drop-down menu. This grouping breaks down the metrics by sockets. The M2PCIe blocks are named by the PCIe devices they serve. If M2PCIe manages more than one device, the device names are display in a comma-separated list. Hover over a cell to see all devices.

To see Intel DDIO utilization efficiency metrics, expand the second level. Click the **Expand** button on each column.



	Inbound PCIe Read, MB/sec ▼			Inbound PCIe Write, MB/sec			
	L3 Hit, %	L3 Miss, %	Average Latency, ns	L3 Hit, %	L3 Miss, %	Core/IO Conflicts, %	Average Latency, ns
	100.000	0.000	112.082	99.999	0.001	0.000	135.668
or 40GbE	100.000	0.000	112.082	99.999	0.001	0.000	135.668
	53.283	46.717	182.850	46.261	53.739	0.000	206.414
vice 0x20	53.439	46.561	178.393	46.251	53.749	0.000	206.439
or 10GBA	42.692	57.308	545.502	58.832	41.168	0.000	168.893

Check the **Platform** tab to see DRAM and Intel UPI bandwidth details.

Remote Socket Accesses

This first experiment demonstrates a suboptimal application topology that results in these conditions:

- High DDIO miss rate
- Higher DDIO latency
- Induced DRAM and Intel® Ultra Path Interconnect (Intel® UPI) traffic

Together, all these factors limit performance.

This is an example of a suboptimal topology that uses a remote configuration with the forwarding core and the NIC residing on distinct sockets:

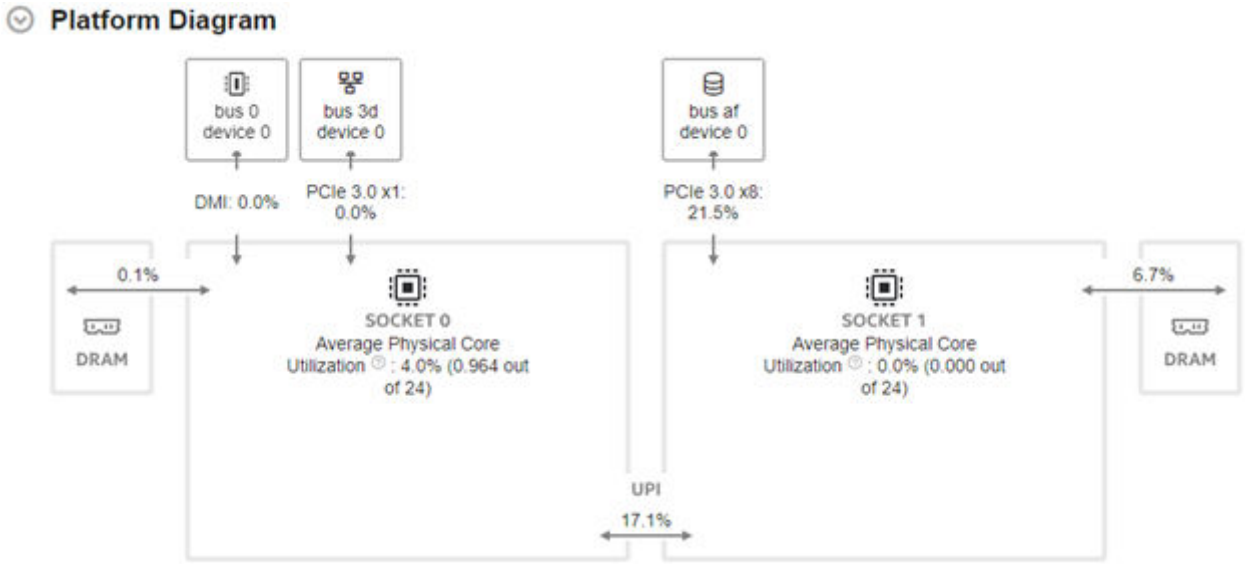
```
# ./testpmd -n 4 -l 0,1 -- -i
testpmd> set fwd mac retry
testpmd> start
```

Now, re-run the analysis in the **Attach to Process** mode, using the graphical interface or the command line:

```
# vtune -collect io --duration 20 --target-process testpmd
```

The **Platform Diagram** reveals issues with topology:

- Core utilization on a distinct socket with regard to NIC
- Non-zero DRAM and UPI utilization

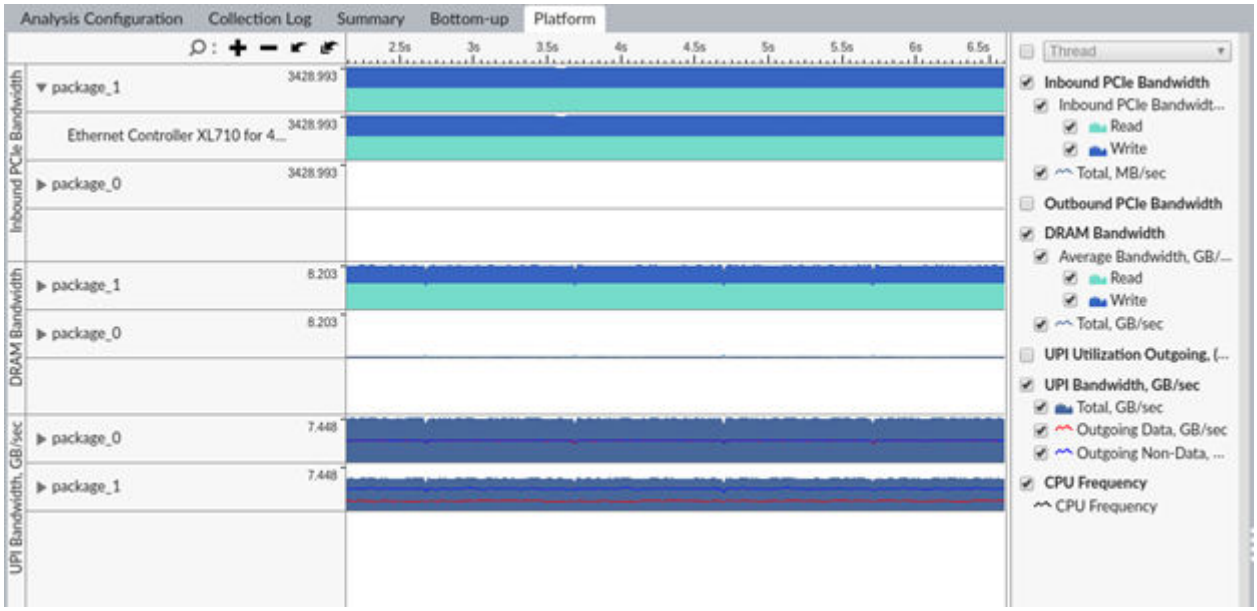


Examine the results:

Forwarding core ID	Throughput, Mpps	Inbound PCIe Read L3 Miss, %	Average Inbound PCIe Read Latency, ns	Inbound PCIe Write L3 Miss, %	Average Inbound PCIe Write Latency, ns
25	21.1	0	112	0	135
1	17.1	100	320	100	240

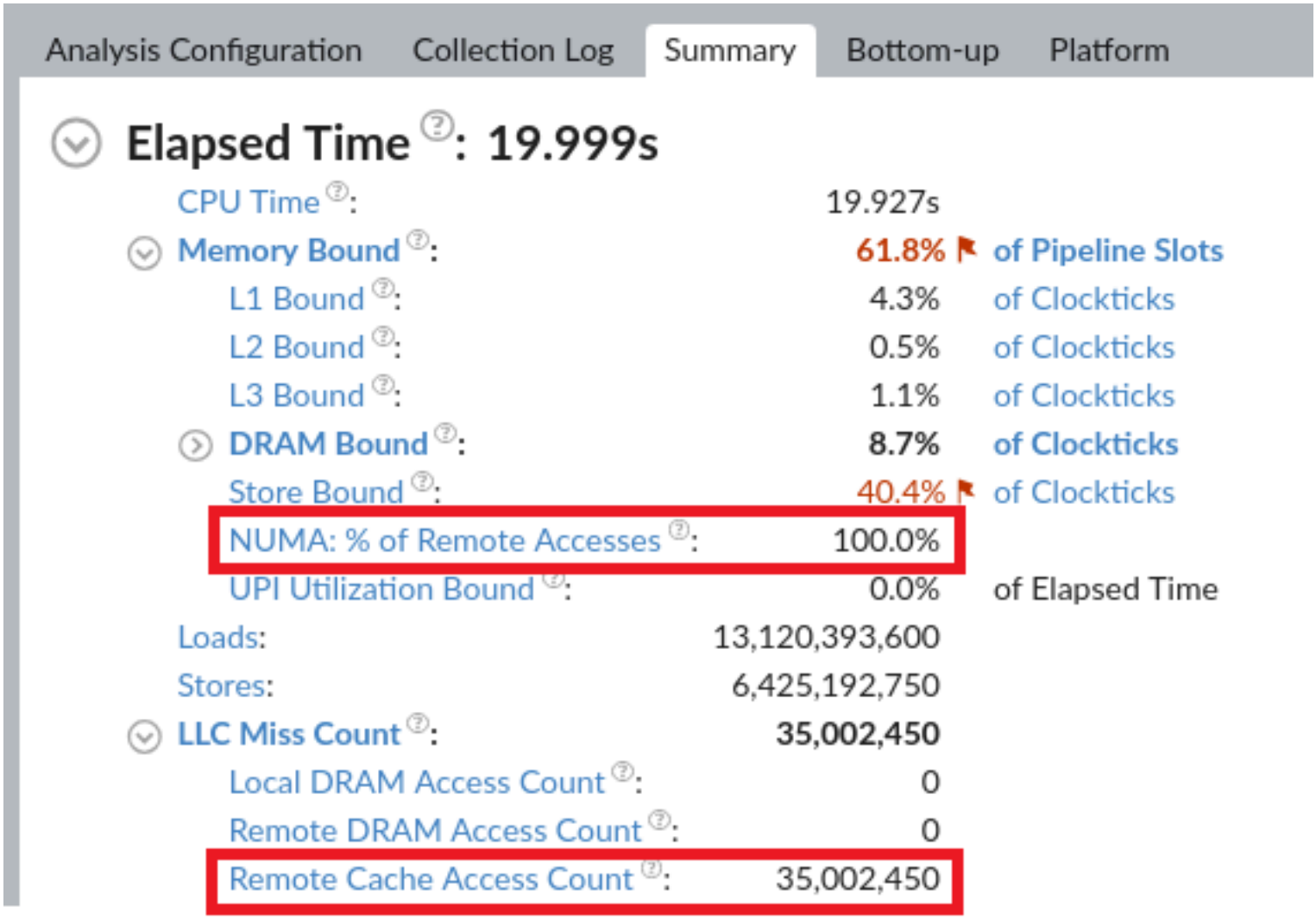
A configuration where the forwarding core and the NIC reside on distinct sockets demonstrates worse performance with a 100% L3 miss rate and higher latency for inbound PCIe requests.

To understand the implications of high miss rates in the remote case, switch to the **Platform** pane:



There is high UPI and DRAM bandwidth. To get a holistic view, analyze the configuration from the core perspective by using the **Memory Access** analysis:

```
# vtune -collect memory-access -knob dram-bandwidth-limits=false --duration 20 --target-process
testpmd
```



From the VTune Profiler results, we see in the remote configuration that CPU cores suffer from remote accesses. These are due to LLC misses that get resolved by taking the data from the remote L3 cache.

Navigate to the **Bottom-up** pane to determine which cores, processes, threads, or functions accessed the remote LLC:

nction / Call Stack

CPU Time	Memory Bound	Loads	Stores	LLC Miss Count	
				Local DRAM Access Count	Remote DRAM Acces
9.902s	61.8%	13,120,393,600	6,425,192,750	0	
9.902s	61.8%	13,120,393,600	6,425,192,750	0	
0.025s	0.0%	0	0	0	

You can see that all of the remote LLC accesses are induced by the `testpmd` application running on Core 1 from Socket 0.

Now you can recreate what is happening in the remote configuration. There is zero DRAM bandwidth on Socket 0. Therefore, all the memory consumed by the application and used for descriptor and packet rings, is allocated on Socket 1 locally to the NIC. When the forwarding core from Socket 0 accesses descriptors and packets, the core misses LLC on Socket 0.Snoop requests are sent to bring the data from Socket 1, which induces UPI traffic. If these requests find modified data in the LLC of Socket 1, a DRAM write-back occurs. This write-back contributes to the measured DRAM bandwidth.

When the device accesses the same locations again, it misses the L3 cache on Socket 1. This is because the data was last used by a core on Socket 0 and remains there. So the memory directory is accessed to determine the socket where the address could be cached. This action also contributes to the observed DRAM bandwidth. This time, the snoop requests travel from Socket 1 to Socket 0 to enforce coherency rules and to complete the I/O request.

As a result, you observe these values from the data collected by the **Input and Output** and **Memory Access** analyses:

Forwarding core ID	Throughput, Mpps	Inbound PCIe Read L3 Miss %	Average Inbound PCIe Read Latency, ns	Inbound PCIe Write L3 Miss %	Average Inbound PCIe Write Latency, ns	testpmd: LLC Miss Count	testpmd: Remote Cache Access Count	Total DRAM bandwidth (Socket 1), GB/s	Total UPI bandwidth, GB/s
25	21.1	0	112	0	135	0	0	0	0
1	17.1	100	320	100	240	35M	35M	8	12.6

In addition to lower throughput caused by higher request latencies, suboptimal application topology causes the system to waste DRAM bandwidth, UPI bandwidth, and platform power.

Poor L3 Cache Management

This second experiment introduces several serious performance issues where, even in the absence of remote socket accesses, the performance is limited. This is due to suboptimal management of I/O data in the LLC, as shown by DDIO misses.

To demonstrate an example using DPDK `testpmd`, disable the software-level caching of memory pools (DPDK Mempool Library) that are used as packet rings. Using this default caching mechanism, a core receives a new packet. For the data destination, the core uses a "warm" memory pool element, which most likely resides in hardware caches. Therefore, there are no L3 misses for Inbound PCIe reads and writes, even with packet ring size going above L3 capacity.

Run `testpmd` with `--mbcache=0` option to disable memory pools software caching:

```
# ./testpmd -n 4 -l 24,25 -- -i --mbcache=0
testpmd> set fwd mac retry
testpmd> start
```

Compare `testpmd` performance for the initial local configuration and the same configuration, but with disabled software caching of memory pools:

Mempool cache enabled	Throughput, Mpps	Inbound PCIe Read L3 Miss %	Average Inbound PCIe Read Latency, ns	Inbound PCIe Write L3 Miss %	Average Inbound PCIe Write Latency, ns	Total DRAM bandwidth (Socket 1), GB/s
Yes	21.1	0	112	0	135	0
No	20.2	0	115	54	178	4.7

When the application is running without memory pool caching optimization, a significant portion of inbound PCIe write requests misses the L3.

To forward one packet, the NIC and the core communicate through the packet descriptor and packet rings. On the data path, the NIC uses inbound PCIe writes to write a packet and update packet descriptors. To learn more, see PCIe Traffic in DPDK Apps.

Descriptor rings are always accessed by a core before I/O, so the probability of an I/O L3 miss on descriptor access is low. But the packet ring is accessed first by the NIC, so all the inbound PCIe write L3 misses are caused by the NIC writing packets to the packet ring at the Rx stage. At the same time, no inbound PCIe read L3 misses are observed, because DPDK follows a zero-copy policy for network data, and once the NIC attempts to take a packet and perform Tx, a packet is already cached.

In conclusion, when packet ring software caching is disabled, packet ring accesses should suffer from hardware cache misses. However, as this recipe demonstrates, you can understand what data was accessed by I/O, resulting in a DDIO miss.

The implications of inbound PCIe request L3 misses in this case are the DRAM bandwidth induced by write-backs from the L3, L3 allocations, and memory directory accesses.

Inferences

Intel® DDIO technology enables software to fully utilize high-speed I/O devices. However, software may not fully benefit from Intel DDIO due to a suboptimal application topology in NUMA systems and/or poor management of data in the L3 cache. This can lead to high L3 access latencies and unnecessary DRAM traffic. Analysis types in VTune Profiler like **Input and Output**, **Memory Access**, and **Microarchitecture Exploration** highlight these inefficiencies and help create a holistic picture from both the core and I/O perspectives.

While the problem of a wrong application topology has an obvious solution, developing an efficient L3 utilization scheme may be more complicated. Several approaches can help design such a scheme and increase performance:

- Choose buffer sizes that are less than LLC capacity.
- Recycle buffer elements.
- Use software prefetching of locations used by the device.
- Use L3 partitioning with Intel Cache Allocation Technology (CAT).

Discuss this recipe in the Analyzers forum.

See Also

Input and Output Analysis
Intel® Data Direct I/O Technology Brief
PCIe Traffic in DPDK Apps This recipe introduces PCIe Bandwidth metrics used in Intel® VTune™ Profiler to explore the PCIe traffic for a packet forwarding DPDK-based workload.
Intel® Xeon® Processor Scalable Family Technical Overview
Utilizing the Intel® Xeon® Processor Scalable Family IIO Performance Monitoring Events
Intel® Xeon® Processor Scalable Family Uncore Reference Manual
Optimize Memory Usage in Multithreaded Data Plane Development Kit (DPDK) Applications
Benchmarking and Analysis of Software Data Planes Whitepaper
What Every Programmer Should Know About Memory by Ulrich Drepper of Red Hat, Inc.

Compile a Portable Optimized Binary with the Latest Instruction Set

Learn how to compile a binary with the latest instruction set while maintaining portability.

Content expert: Jeffrey Reinemann

Modern Intel® processors support instruction set extensions like the different versions of Intel® Advanced Vector Extensions (Intel® AVX):

- AVX
- AVX2
- AVX-512

When you compile your application, consider these options based on the purpose of your application:

- **Generic binary:** Compile an application for the generic x86 instruction set. The application runs on all x86 processors, but may not utilize a newer processor to its full potential.
- **Native binary:** Compile an application for the specific processor. The application utilizes all features of the target processor but does run on older processors.
- **Portable binary:** Compile a portable optimized binary with multiple versions of functions. Each version is targeted for different processors using compiler options and function attributes. The resulting binary has the performance characteristics of an application compiled for a specific processor (native binary) and can run on older processors.

This recipe demonstrates how you can compile a portable binary with the performance characteristics of a native binary, while still maintaining portability of a generic binary. In this recipe, you compile both the generic and native binaries first to determine if the resulting performance improvement is large enough to justify the increase in binary size.

This recipe covers the Intel® C++ Compiler Classic and the GNU* Compiler Collection (GCC).

This recipe does not cover:

- Manual dispatching using the `CPUID` processor instruction
- Processor Targeting compiler options
- The target function attribute

Ingredients

This section lists the systems and tools used in the creation of this recipe:

- **Processor:** Intel® Core™ code named Skylake i7-6700 CPU @ 3.40GHz
- **Operating System:** Linux OS (Ubuntu 22.04.3 LTS with kernel version 6.2.0-35-generic)
- **Compilers:**
  - Intel® C++ Compiler Classic 2024.0
  - GCC version 11.4.0
- **Analysis Tool :**Intel® VTune™ Profiler version 2024.0 or newer

Sample Application

Save this code to a source file named `fma.c`:

```
// fma.c
#include <stdio.h>
#include <stdlib.h>

void init(float *a, float *b, float *c, int size)
{
    for (int i = 0; i < size; i++)
    {
        a[i] = (float) (i % 10);
        b[i] = a[i] * 1.1f;
        c[i] = a[i] * 1.2f;
    }
}

void my_fma(float *a, float *b, float *c, int size)
{
    for (int i = 0; i < size; i++)
    {
        c[i] += a[i]*b[i];
```

```
    }
}

#define ITERATIONS 10000000
#define SIZE 2048

int main()
{
    float *a = malloc(SIZE*sizeof(float));
    float *b = malloc(SIZE*sizeof(float));
    float *c = malloc(SIZE*sizeof(float));

    for (int i = 0; i < ITERATIONS; i++)
    {
        init(a, b, c, SIZE);
        my_fma(a, b, c, SIZE);
    }
    printf("%f", c[5]); // use the data

    free(a);
    free(b);
    free(c);
    return 0;
}
```

Compile Generic Optimized Binary

Compile the binary following the recommendations from VTune Profiler User Guide (recommendations for Windows).

Intel® DPC++/C++ Compiler

Compile the binary with debug information and `-O3` optimization level:

```
icx -g -O3 -debug inline-debug-info fma.c -o fma_generic
```

GNU Compiler Collection

Compile the binary with debug information and `-O2` optimization level:

```
gcc -g -O2 fma.c -o fma_generic_O2
```

To check if the code was vectorized, use the HPC Performance Characterization analysis in VTune Profiler:

```
vtune -c hpc-performance -r fma_generic_O2_hpc ./fma_generic_O2
```

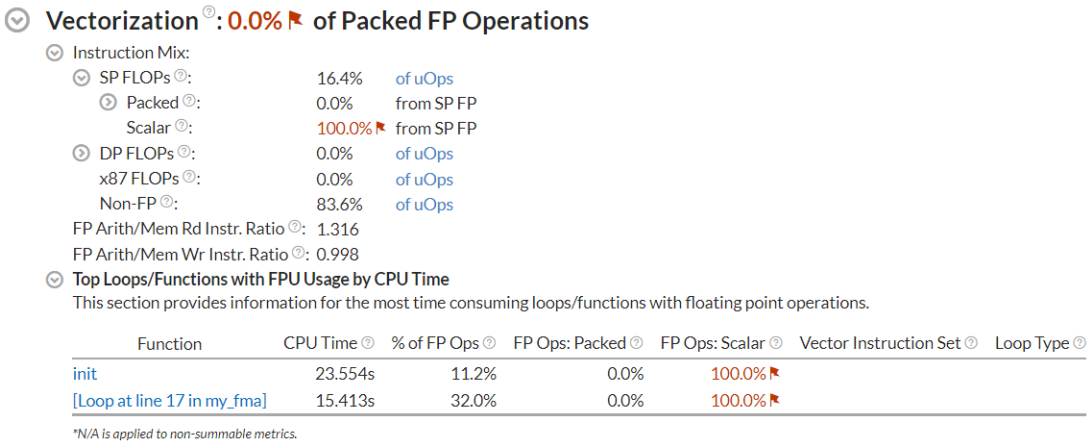
The output of this command includes information about vectorization:

```
Vectorization: 0.0% of Packed FP Operations
Instruction Mix
SP FLOPs: 16.4% of uOps
 Packed: 0.0% from SP FP
  128-bit: 0.0% from SP FP
  256-bit: 0.0% from SP FP
 Scalar: 100.0% from SP FP
```

Open the result in the VTune Profiler GUI:

```
vtune-gui fma_generic_O2_hpc
```

Once you open the analysis result, in the **Summary** tab, see the **Top Loops/Functions with FPU Usage by CPU Time** section :



Vectorization: 0.0% of Packed FP Operations

Instruction Mix:

SP FLOPs:	16.4%	of uOps
Packed:	0.0%	from SP FP
Scalar:	100.0%	from SP FP
DP FLOPs:	0.0%	of uOps
x87 FLOPs:	0.0%	of uOps
Non-FP:	83.6%	of uOps
FP Arith/Mem Rd Instr. Ratio:	1.316	
FP Arith/Mem Wr Instr. Ratio:	0.998	

Top Loops/Functions with FPU Usage by CPU Time

This section provides information for the most time consuming loops/functions with floating point operations.

Function	CPU Time	% of FP Ops	FP Ops: Packed	FP Ops: Scalar	Vector Instruction Set	Loop Type
init	23.554s	11.2%	0.0%	100.0%		
[Loop at line 17 in my_fma]	15.413s	32.0%	0.0%	100.0%		

*N/A is applied to non-summable metrics.

The fact that **FP Ops: Scalar** value equals 100% and that the **Vector Instruction Set** column is empty indicates that GCC does not vectorize the code at `-O2` optimization level.

Use `-O2 -ftree-vectorize` or `-O3` options to enable vectorization.

Compile the `fma_generic` binary with `-O3` optimization level:

```
gcc -g -O3 fma.c -o fma_generic
```

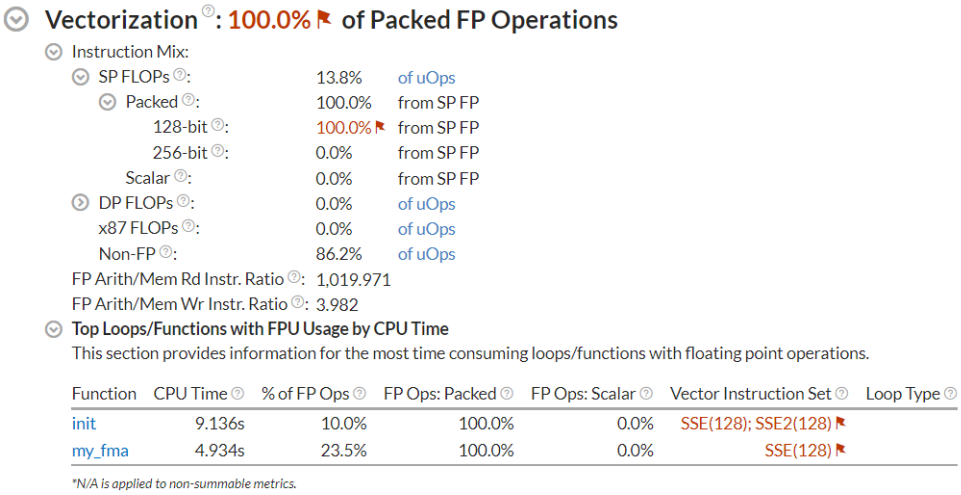
Collect the HPC Performance Characterization analysis data for the generic binary:

```
vtune -c hpc-performance -r fma_generic_hpc ./fma_generic
```

The output of this analysis includes the following information:

```
Vectorization: 100.0% of Packed FP Operations
Instruction Mix
SP FLOPs: 8.3% of uOps
    Packed: 100.0% from SP FP
        128-bit: 100.0% from SP FP
        256-bit: 0.0% from SP FP
 Scalar: 0.0% from SP FP
```

When you open the analysis result in the VTune Profiler GUI, you can find information about vectorization:



Vectorization: 100.0% of Packed FP Operations

Instruction Mix:

SP FLOPs:	13.8%	of uOps
Packed:	100.0%	from SP FP
128-bit:	100.0%	from SP FP
256-bit:	0.0%	from SP FP
Scalar:	0.0%	from SP FP
DP FLOPs:	0.0%	of uOps
x87 FLOPs:	0.0%	of uOps
Non-FP:	86.2%	of uOps
FP Arith/Mem Rd Instr. Ratio:	1,019.971	
FP Arith/Mem Wr Instr. Ratio:	3.982	

Top Loops/Functions with FPU Usage by CPU Time

This section provides information for the most time consuming loops/functions with floating point operations.

Function	CPU Time	% of FP Ops	FP Ops: Packed	FP Ops: Scalar	Vector Instruction Set	Loop Type
init	9.136s	10.0%	100.0%	0.0%	SSE(128); SSE2(128)	
my_fma	4.934s	23.5%	100.0%	0.0%	SSE(128)	

*N/A is applied to non-summable metrics.

Compile Native Binary

Compile native binary with the Intel® DPC++/C++ Compiler

The `-xHost` option instructs the compiler to generate instructions for the highest instruction set available on the processor performing the compilation. Alternatively, the `-x{Arch}` option, where `{Arch}` is the architecture codename, instructs the compiler to target processor features of a specific architecture.

Compile the `fma_native` binary with `-xHost` flag:

```
icx -g -O3 -debug inline-debug-info -xHost fma.c -o fma_native
```

Compile native binary with the GNU Compiler Collection

Compile the `fma_native` binary with `-march=native` flag:

```
gcc -g -O3 -march=native fma.c -o fma_native
```

If your processor supports the AVX-512 instruction set extension, consider experimenting with the `mprefer-vector-width=512` option.

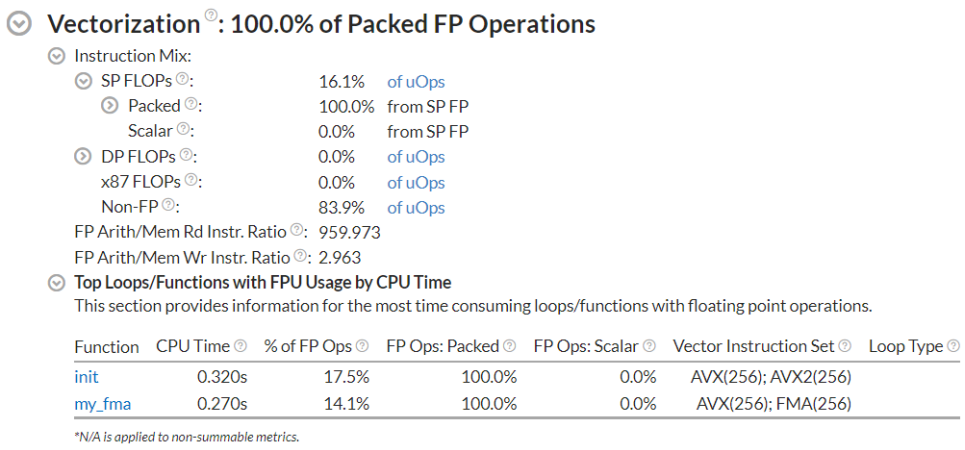
Next, collect HPC data for the native binary:

```
vtune -c hpc-performance -r fma_native_hpc ./fma_native
```

The output of this analysis includes the following information:

```
Vectorization: 100.0% of Packed FP Operations
Instruction Mix
  SP FLOPs: 14.2% of uOps
   Packed: 100.0% from SP FP
    128-bit: 0.0% from SP FP
    256-bit: 100.0% from SP FP
   Scalar: 0.0% from SP FP
```

When you open the analysis result in the VTune Profiler GUI, you can find information about vectorization:



Compare Generic and Native Binaries

To compare the HPC data collected for the generic and native binaries, run this command:

```
vtune-gui fma_generic_hpc fma_native_hpc
```

In the VTune Profiler GUI, switch to the **Bottom-Up** tab. Set **Loop Mode** to **Functions only**.

Switch to the **Summary** tab and scroll down to the **Top Loops/Functions with FPU Usage by CPU Time** section:

Vectorization: Not changed, 100.0% of Packed FP Operations

Instruction Mix:

SP FLOPs:	13.8% - 16.1% = -2.3%	of uOps
DP FLOPs:	Not changed, 0.0%	of uOps
x87 FLOPs:	Not changed, 0.0%	of uOps
Non-FP:	86.2% - 83.9% = 2.3%	of uOps
FP Arith/Mem Rd Instr. Ratio:	1,019.971 - 959.973 = 59.998	
FP Arith/Mem Wr Instr. Ratio:	3.982 - 2.963 = 1.019	

Top Loops/Functions with FPU Usage by CPU Time

This section provides information for the most time consuming loops/functions with floating point operations.

Function	CPU Time	% of FP Ops	FP Ops: Packed	FP Ops: Scalar	Vector Instruction Set	Loop Type
init	9.136s - 0.320s = 8.815s	10.0% - 17.5% = -7.5%	Not changed, 100.0%	Not changed, 0.0%	SSE(128); SSE2(128) | AVX(256); AVX2(256)	Not changed, undefined
my_fma	4.934s - 0.270s = 4.664s	23.5% - 14.1% = 9.5%	Not changed, 100.0%	Not changed, 0.0%	SSE(128) | AVX(256); FMA(256)	Not changed, undefined

*N/A is applied to non-summable metrics.

Observe the **CPU Time** and **Vector Instruction Set** columns.

Consider the performance difference between the generic and the native binary. Decide whether it makes sense to compile a portable binary with multiple code paths.

> **NOTE**
> This sample application was auto-vectorized by the compiler. To investigate vectorization opportunities in your application in depth, use Intel® Advisor.

Compile Portable Binary

If the comparison between the generic and native binary shows a performance improvement, for example, if the **CPU Time** was improved, consider compiling a portable binary.

Compile the portable binary with the Intel® DPC++/C++ Compiler

Use the `-ax` (`/Qax` for Windows) option to instruct the compiler to generate multiple feature-specific auto-dispatch code paths for Intel processors.

Compile the `fma_portable` binary with the `-ax` option:

```
icx -g -O3 -debug inline-debug-info -axCOMMON-AVX512,CORE-AVX2,AVX,SSE4.2,TREMONT,ICELAKE-SERVER
fma.c -o fma_portable
```

Refer to the `-ax` option help page for the list of supported architectures.

Compile the portable binary with the GNU Compiler Collection

Compare the results for generic and native binaries. If the **CPU Time** was improved and an additional **Vector Instruction Set** was utilized for a specific function in the native binary result, then add the `target_clones` attribute to this function.

If the function calls other functions, consider adding the `flatten` attribute to force inlining, since the `target_clones` attribute is not recursive.

Copy the contents of the `fma.c` source file to a new file, `fma_portable.c`, and add the `TARGET_CLONE` preprocessor macro:

```
#define TARGET_CLONES __attribute__((flatten,target_clones("default,sse4.2,avx,"\
    "avx2,avx512f,arch=skylake,arch=tremont,arch=skylake-avx512,"\
    "arch=cascadelake,arch=cooperlake,arch=tigerlake,arch=icelake-server")))
```

Refer to the x86 Options page of the GCC manual for the list of supported architectures.

Multiple versions of a function will increase the binary size. Consider the trade-off between performance improvement for each target and code size. Collecting and comparing VTune Profiler results enables you to make data-driven decisions to apply the `TARGET_CLONES` macro only to the functions that will run faster with new instructions.

Add the `TARGET_CLONES` macro before the `my_fma` function definition and `init` functions and save the changes to `fma_portable.c`:

```
TARGET_CLONES
void my_fma(float *a, float *b, float *c, const int size)
```

Compile the `fma_portable` binary:

```
gcc -g -O3 fma_portable.c -o fma_portable
```

Compare Portable and Native Binaries

To compare the performance of portable and optimized binaries, collect the HPC Performance Characterization data for the `fma_portable` binary:

```
vtune -c hpc-performance -r fma_portable_hpc ./fma_portable
```

The output of this analysis includes the following data:

```
Vectorization: 100.0% of Packed FP Operations
Instruction Mix
  SP FLOPs: 6.9% of uOps
   Packed: 100.0% from SP FP
    128-bit: 66.7% from SP FP
    256-bit: 33.3% from SP FP
   Scalar: 0.0% from SP FP
```

Open the comparison in VTune Profiler GUI:

```
vtune-gui fma_portable_hpc fma_native_hpc
```

Vectorization: Not changed, 100.0% of Packed FP Operations

Instruction Mix:

SP FLOPs:	Not changed, 16.1%	of uOps
DP FLOPs:	Not changed, 0.0%	of uOps
x87 FLOPs:	Not changed, 0.0%	of uOps
Non-FP:	Not changed, 83.9%	of uOps
FP Arith/Mem Rd Instr. Ratio:	undefined - 959.973 = -959.973	
FP Arith/Mem Wr Instr. Ratio:	Not changed, 2.963	

Top Loops/Functions with FPU Usage by CPU Time

This section provides information for the most time consuming loops/functions with floating point operations.

Function	CPU Time	% of FP Ops	FP Ops: Packed	FP Ops: Scalar	Vector Instruction Set	Loop Type
init	0.315s - 0.320s = -0.005s	16.6% - 17.5% = -1.0%	Not changed, 100.0%	Not changed, 0.0%	Not changed, AVX(256); AVX2(256)	Not changed, undefined
my_fma	0.273s - 0.270s = 0.003s	15.3% - 14.1% = 1.2%	Not changed, 100.0%	Not changed, 0.0%	Not changed, AVX(256); FMA(256)	Not changed, undefined

**N/A is applied to non-summable metrics.*

As a result, the portable binary uses the highest instruction set extension available and demonstrates optimal performance on the target system.

Configuration Recipes

Follow these recipes to configure your system and set up Intel® VTune™ Profiler or its predecessor, Intel® VTune™ Amplifier, for performance analysis in particular code environments.

Profiling OpenVINO™ Applications (NEW)

Learn how to use Intel® VTune™ Profiler to profile AI applications. This recipe uses a benchmark application in the OpenVINO™ toolkit.

Content Expert: Kali Uday Balleda, Kumar Sanatan

- INGREDIENTS
- DIRECTIONS:

1. Set up OpenVINO™.
2. Build the OpenVINO™ source.
3. Configure OpenVINO™ for Performance Analysis.
4. Profile CPU Hot Spots.
5. Profile GPU Hot Spots.
6. Profile NPUs.

Ingredients

Here are the hardware and software tools you need for this recipe.

- **Applications and Toolkits**:
  - If your OpenVINO application requires the use of Intel® oneAPI Deep Neural Network Library (oneDNN) or Intel® oneAPI Data Analytics Library (oneDAL), download the Intel® oneAPI Base Toolkit for Windows* (version 2024.1 or newer).
  - Intel® NPU Driver - Windows* (version 32.0.100.2381)
  - `Benchmark Tool` application. Get this application with the 2024.1 (or a newer) version of the OpenVINO™ toolkit. The toolkit contains Python and C++ versions of `benchmark_app`.
  - Intel® Distribution for Python* (version 2024.1)
  - CMake version 3.29.2 for Microsoft Windows*
  - Microsoft Visual Studio 2022 Community Edition (17.9.6)
  - Git for Windows
- **Analysis Tool**:VTune Profiler (version 2024.1 or newer)
- **CPU**: Intel® Core Ultra 7 Processor 165H (code named Meteor Lake)

Set up OpenVINO™

To set up OpenVINO™ from the package manager, install these applications in sequence:

1. If you use oneDNN or oneDAL libraries, install the Intel® oneAPI Base Toolkit. Follow these instructions.
2. Next, install Intel® Distribution for Python* (Intel Python3).
3. Set up CMake.
4. Next, set up Microsoft Visual Studio 2022.
5. Install Git for Windows.
6. At the command prompt, set the path to Intel Python3. In the folder where you downloaded Intel Distribution for Python, run `vars.bat` from the `env` subdirectory.
7. Install the latest version of the OpenVINO™ toolkit:

```
pip install OpenVINO™
```

Build the OpenVINO™ source

1. In Github, clone the OpenVINO™ repository:

```
git clone https://github.com/OpenVINO™toolkit/OpenVINO™.git
```

2. Open the OpenVINO™ folder:

```
cd OpenVINO™
```

3. Update the submodule:

```
git submodule update –init
```

4. Create a build directory:

```
mkdir build
```

5. Open the build directory:

```
cd build
```

6. Generate a Microsoft Visual Studio project with the OpenVINO source code. Make sure to enable the OpenVINO runtime with Python API. Also enable the ability to profile with the Instrumentation and Tracing Technology (ITT) API.

```
cmake -G “Visual Studio 17 2022” <path\to\OpenVINO™ source code> -DCMAKE_BUILD_TYPE=Release
-DENABLE_PYTHON=ON  -DPython3_EXECUTABLE= "C:\Users\sdp\Downloads\intelpython3\python.exe"
-DENABLE_PROFILING_ITT=ON
```

7. Open the `OpenVINO™.sln` solution in Microsoft Visual Studio 2022.

8. Build the solution file in release mode.

When the build is successful, you find a new `bin` directory that contains:

- Python libraries in `bin\intel64\Release\python`
- OpenVINO™ Plugins and DLLs in `bin\int64\Release\`

Configure OpenVINO™ for Performance Analysis

Before you run performance analyses on AI applications, make sure to configure OpenVINO™ first:

1. Configure the OpenVINO™ build with the Instrumentation and Tracing Technology API (ITT API). Set these environment variables:

```
set PYTHONPATH=<path to OpenVINO™ repository>\bin\intel64\Release\python;%PYTHONPATH%
set OpenVINO™_LIB_PATHS=<path to OpenVINO™ repository>\bin\intel64\Release;%OpenVINO™_LIB_PATH%
```

2. Run the setup script for Python:

```
C:\Users\sdp\intelpython3\env\vars.bat
```

3. Run the setup script for oneAPI tools:

```
C:\Program Files (x86)\Intel\<toolkit_version>\oneAPI-vars.bat
```

Understand the Sample Application:

Before you run an AI application on an Intel processor, you must convert the AI model into the OpenVINO™ IR format. To learn more about this conversion, use the `omz_converter` tool.

You can run the profiling analyses described here on any AI application of your choice. This recipe uses the Benchmark tool (available in the OpenVINO™ toolkit) to demonstrate how you run analyses with VTune Profiler. Use `benchmark_app` to calculate the throughput and run a latency analysis for your AI application.

To run `benchmark_app`, at the command prompt, type this command:

```
benchmark_app -m model.xml -d NPU -niter 1000
```

where:

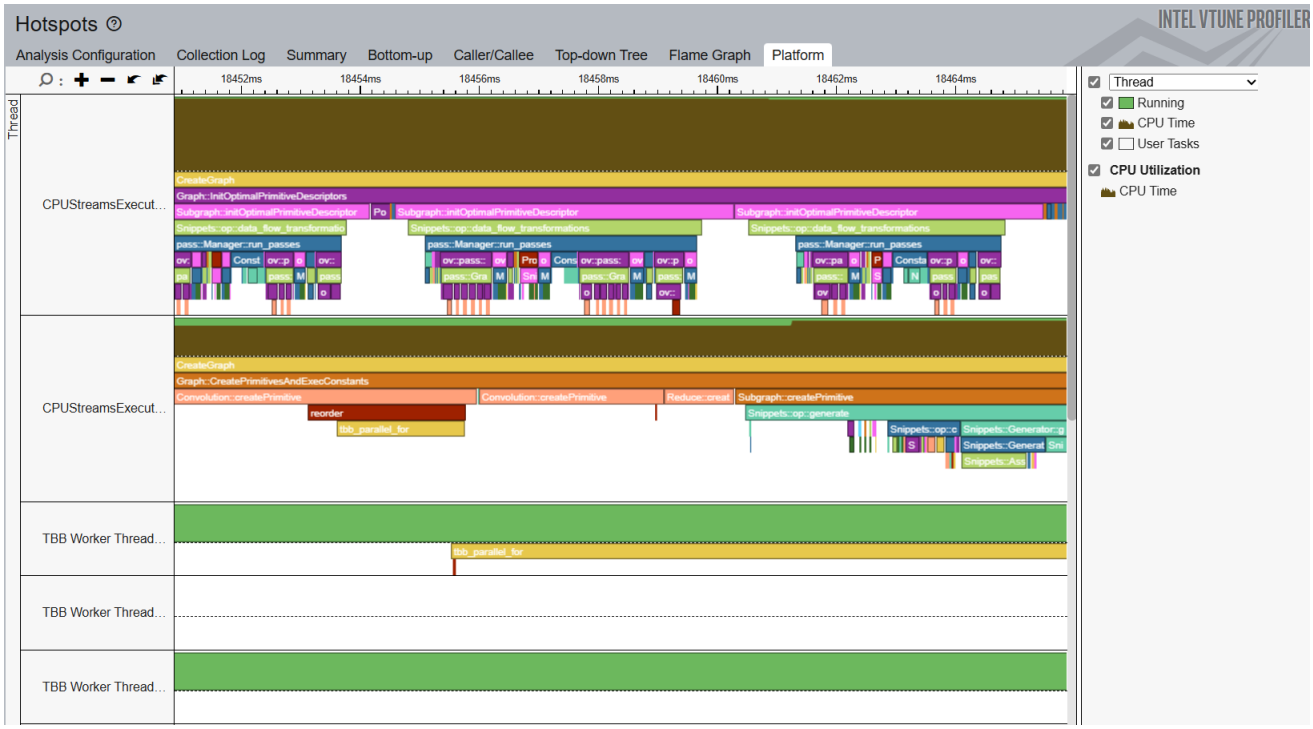
- `model.xml` is your AI application
- `d` is the target device (CPU/GPU/NPU)
- `niter` is the number of iterations

Profile CPU Hot Spots

To identify CPU bottlenecks, run the **Hotspots analysis** . In the command window, type:

```
vtune -c hotspots -r <path to VTune results directory> -- benchmark_app -m model.xml -d CPU -
niter 1000
```

Open the result of the analysis and identify CPU hot spots in the **Bottom-up** and **Platform** windows.

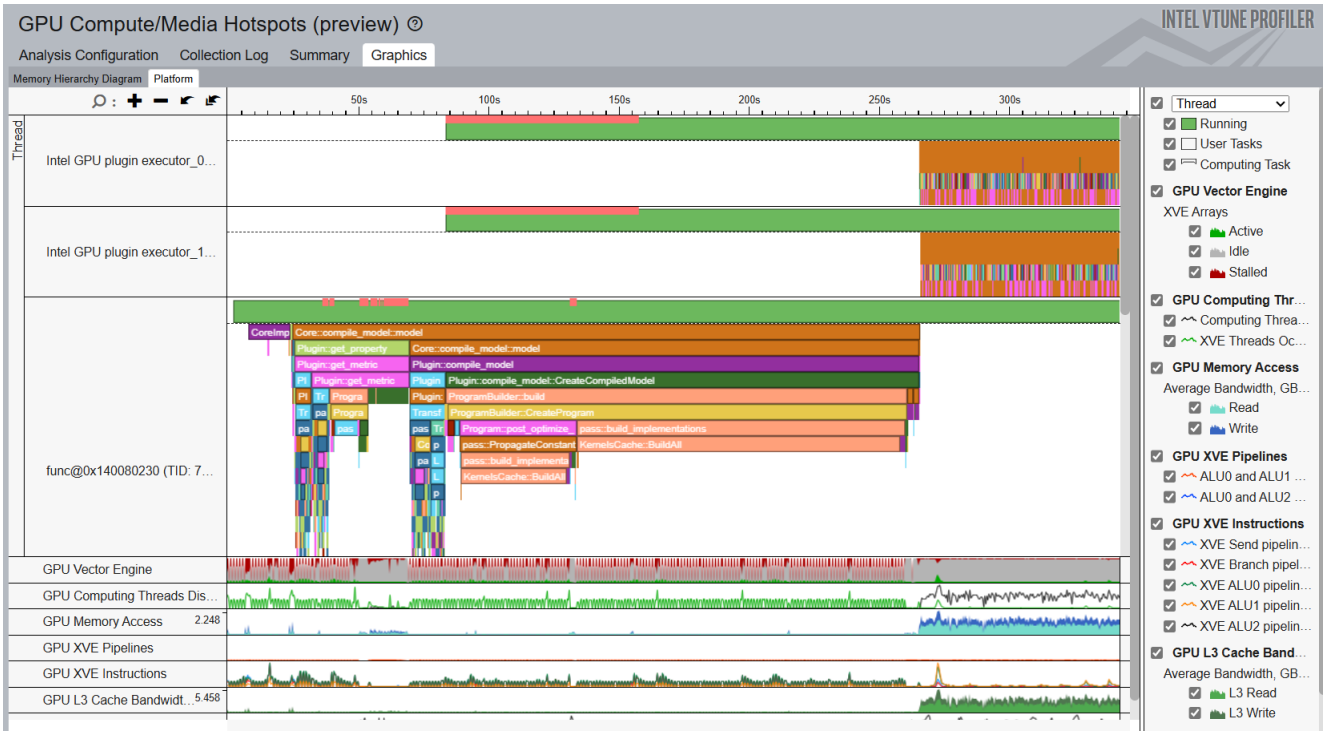


Profile GPU Hot Spots

To identify GPU bottlenecks, run the **GPU Compute/Media Hotspots analysis** . In the command window, type:

```
vtune -c gpu-hotspots -r <path to VTune results directory> -- benchmark_app -m model.xml -d GPU -
niter 1000
```

Open the result of the analysis and identify GPU hot spots in the **Graphics** window.

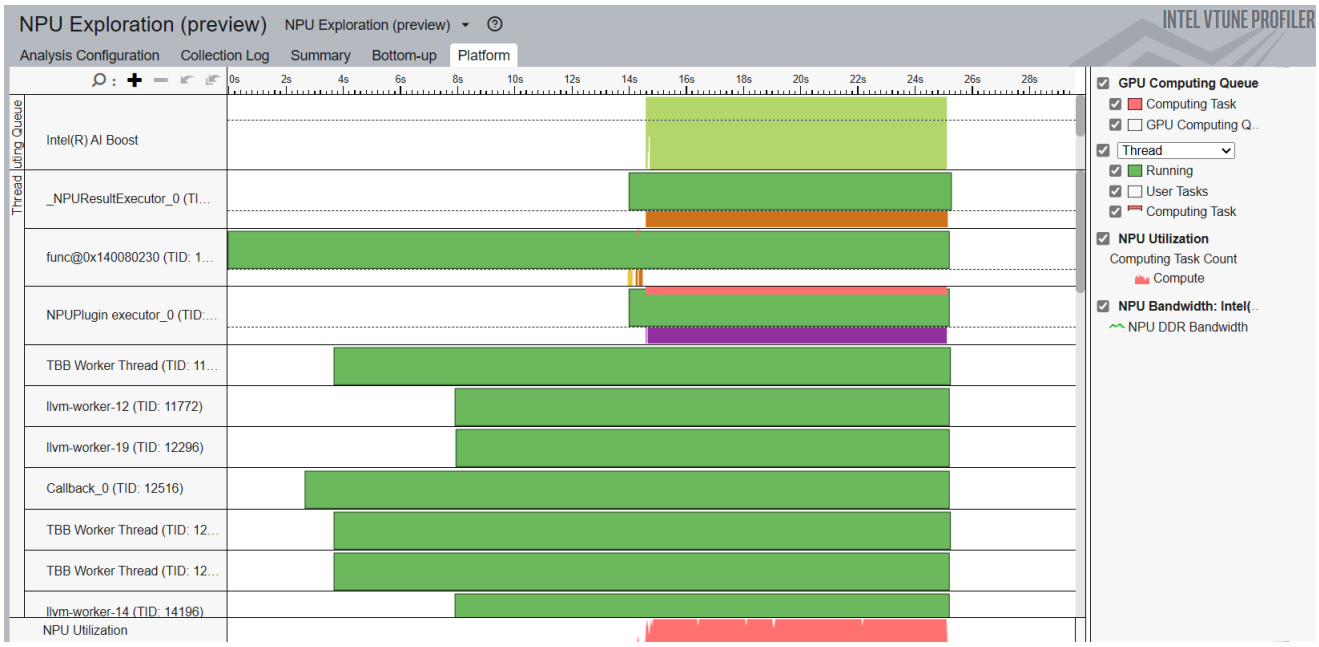


Profile NPUs

To profile NPUs, run the **NPU Exploration analysis** . In the command window, type:

```
vtune -c npu -r <path to VTune results directory> -- benchmark_app -m model.xml -d npu -niter
1000
```

Open the result of the analysis and understand NPU usage in the **Platform** window.



See Also

Intel® Optimization for TensorFlow*

Intel® Optimization for PyTorch*

Build OpenVINO™ Runtime for Windows systems
Hotspots Analysis for CPU Usage Issues
GPU Compute/Media Hotspots Analysis
NPU Exploration Analysis

Profiling High Bandwidth Memory Performance on Intel® Xeon® CPU Max Series (NEW)

Use Intel® VTune™ Profiler to profile memory-bound workloads in high performance computing (HPC) and artificial intelligence (AI) applications which utilize high bandwidth memory (HBM).

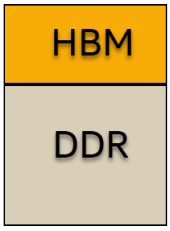
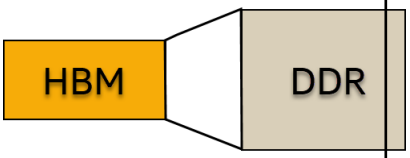
As HPC and AI applications grow increasingly complex, these memory-bound workloads are increasingly challenged by memory bandwidth. High bandwidth memory (HBM) technology in the Intel® Xeon® CPU Max Series of processors tackles the bandwidth challenge. This recipe describes how you use VTune Profiler to profile HBM performance in these memory-bound applications.

Content Experts: Vishnu Naikawadi, Min Yeol Lim, and Alexander Antonov

In this recipe, you use VTune Profiler to profile a memory-bound application on a system that has HBM memory.VTune Profiler displays HBM-specific performance metrics which can help you understand the usage of HBM memory by the workload. Thus, you can analyze the performance of the workload in the context of HBM memory.

Memory Modes in HBM

The Intel® Xeon® CPU Max Series of processors offers HBM in three memory modes:

	HBM Only	HBM Flat Mode	HBM Caching Mode
Memory Configuration	HBM Memory. No DRAM. 	Flat memory regions with HBM and DRAM 	HBM caches DRAM 
Workload Capacity	64 GB or less	64 GB or more	64 GB or more
Code Change	No code change.	Code change may be necessary to optimize performance.	No code change.
Usage	System boots and operates with HBM only.	Provides flexibility for applications that require large memory capacity.	Blend of HBM Only and HBM Flat Mode. Whole applications may fit in HBM cache. This mode blurs the line between cache and memory.

Switch HBM Modes

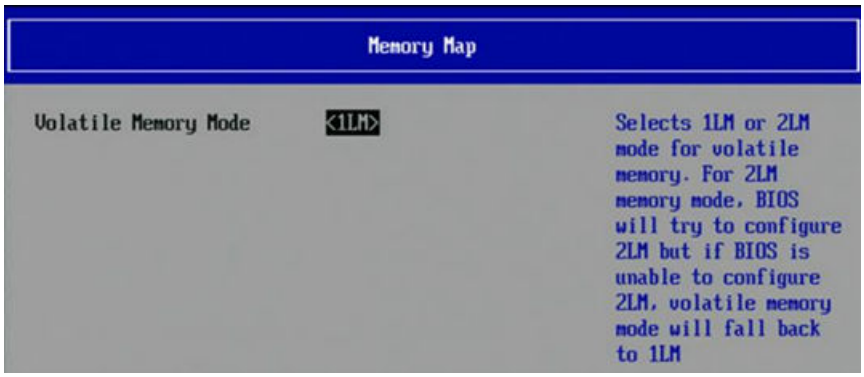
When you do not install DRAM, the processor operates in **HBM Only** mode. In this mode, HBM is the only memory available to the OS and all applications. The OS may see all of the installed HBM, while applications can only see what is exposed by the OS.

When you install DRAM, you can select different HBM memory modes by changing the BIOS memory mode configuration:

1. Open **EDKII Menu**.
2. In the **Socket Configuration** option, select **Memory Map**.
3. Open **Volatile Memory**.

> **NOTE** The UI path to change the BIOS configuration may vary depending on the BIOS running on your system.

4. Change the HBM mode:
   - To select the **HBM Flat** mode, select **1LM** (or 1-Level Mode). This mode exposes the HBM and DRAM memories to the software. Each memory is available as a separate address space (NUMA node).
   - To select the **HBM Cache** mode, select **2LM** (or 2-Level Mode). In this mode, only the DRAM address space is visible. HBM functions as a transparent memory-side cache for DRAM.



Depending on your BIOS, additional changes may be necessary. For more information on switching between memory modes, see the Intel® Xeon® CPU Max Series Configuration and Tuning Guide.

Ingredients

Here are the hardware and software tools you need for this recipe.

- **Application**: This recipe uses the STREAM benchmark.
- **Analysis Tools**:
  - VTune Profiler (version 2024.0 or newer)
  - `numactl` - Use this application to control NUMA policy for processes or shared memory.
- **CPU**: 4th Generation of Intel® Xeon® CPU Max Series processors (formerly code-named Sapphire Rapids HBM)
- **Operating System**: Linux* OS

System Configuration

This recipe uses a system with:

- 2-socket, 224 logical CPUs with Hyper-Threading
- 16 32GB DRAM DIMMs (8 DIMMs for each socket)
- **HBM Flat** mode with SNC4 enabled

As shown in the table below, the system used in this recipe has 8 NUMA nodes per socket:

	Socket 0	Socket 1
DRAM	Nodes 0,1,2,3	Nodes 4,5,6,7

	Socket 0	Socket 1
HBM	Nodes 8,9,10,11	Nodes 12,13,14,15

Directions

1. Run Memory Access Analysis
2. Analyze Results

Run Memory Access Analysis

In this recipe, you use VTune Profiler to run the Memory Access analysis type on the STREAM benchmark. You can run the VTune Profiler standalone application on the target system or use a web browser to access the GUI by running VTune Profiler Server.

This example uses VTune Profiler Server. To set up the server, on your target platform, run this command:

```
/opt/intel/oneapi/vtune/latest/bin64/vtune-backend --web-port <port_id> --allow-remote-access --
data-directory /home/stream/results --enable-server-profiling
```

Here:

- `--web-port` is the HTTP/HTTPS port for the web server UI and data APIs
- `--allow-remote-access` enables remote access through a web browser
- `--data-directory` is the root directory to store projects and results
- `--enable-server-profiling` enables the selection of the hosting server as the profiling target

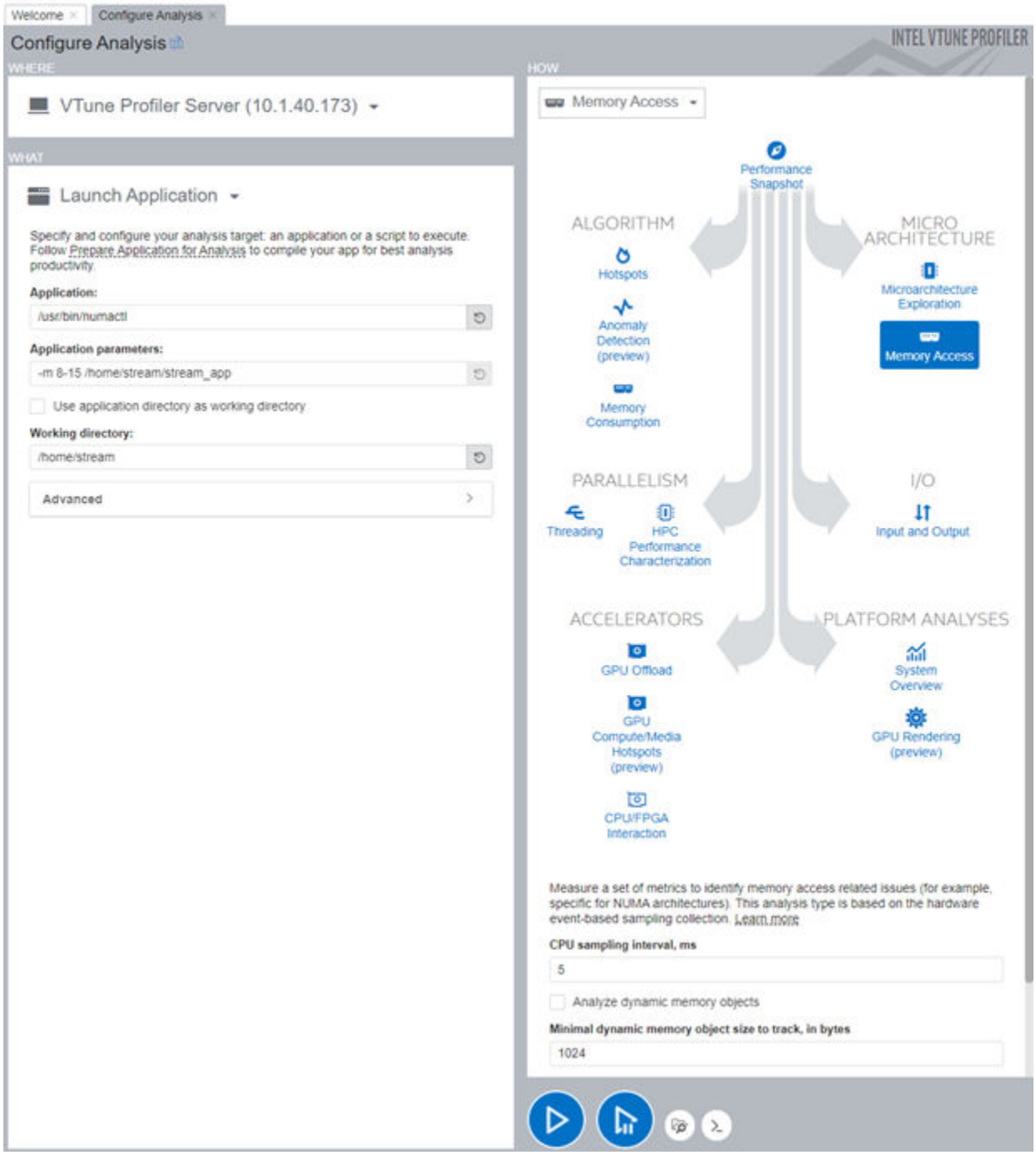
This command returns a token and a URL. Now you are ready to start the analysis.

This recipe describes how VTune Profiler profiles the STREAM application using only HBM NUMA nodes. For this specific system configuration, the analysis uses NUMA nodes 8-15.

1. Open the URL returned at the command prompt.
2. Set a password to use VTune Profiler Server.
3. From the Welcome screen, create a new project.
4. In the **Configure Analysis** window, set these options:

Pane	Option	Setting
WHERE	-	VTune Profiler Server
WHAT	Target	**Launch Application**
	Application	Path to the `numactl` application. **NOTE** Although STREAM is the actual application that gets profiled, you specify the `numactl` tool as the application in order to set NUMA affinity for the STREAM benchmark. You provide the benchmark in the **Application parameters** field instead.
	Application parameters	HBM NUMA nodes 8-15 Path to STREAM benchmark
	Working directory	Path to application directory
HOW	Analysis type	**Memory Access Analysis**

5. Click **Start** to run the analysis.



In this default configuration, VTune Profiler collects HBM bandwidth data in addition to DRAM bandwidth. Therefore, you do not require additional settings.

NOTE To run the Memory Access analysis from the command line, type:

```
vtune -collect memory-access --app-working-dir=/home/stream -- /usr/bin/numactl -m “8-15” /
home/stream/stream_app
```

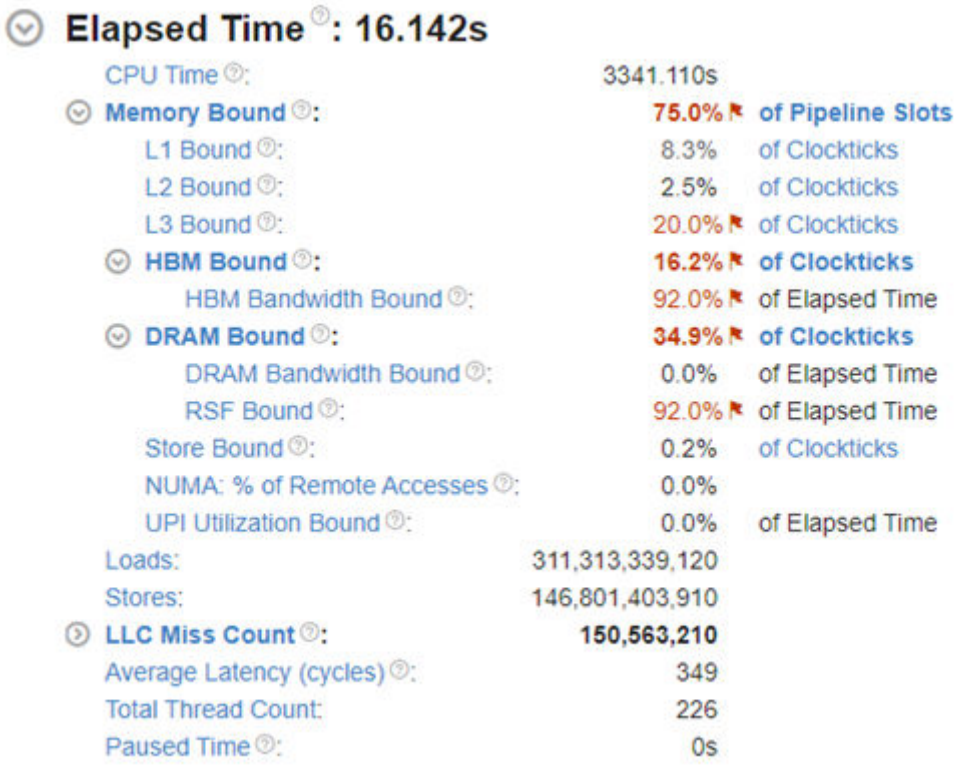
Analyze Results

Once data collection is complete and VTune Profiler displays the results, open the **Summary** window to see general information about the execution. The information is sorted into several sections.

Elapsed Time

This section contains the following statistics:

- Application execution per pipeline slots or clockticks
- **Total Elapsed Time** - This includes idle time
- **CPU Time** - This is the sum of CPU times of all threads and the Paused Time, which indicates the total time the application was paused (by commands from the GUI, CLI, or user API)
- **HBM Bound** - This metric estimates how often the CPU was stalled due to High Bandwidth Memory (HBM) accesses by loads. This metric is measured in CPU cycles or clockticks. Depending on the workload you choose, this metric may be less accurate in data collections in the **HBM Only** and **HBM Flat** modes.
- **HBM Bandwidth Bound** - This shows the percentage of **Elapsed Time** that used HBM bandwidth. This metric is measured in terms of elapsed time.

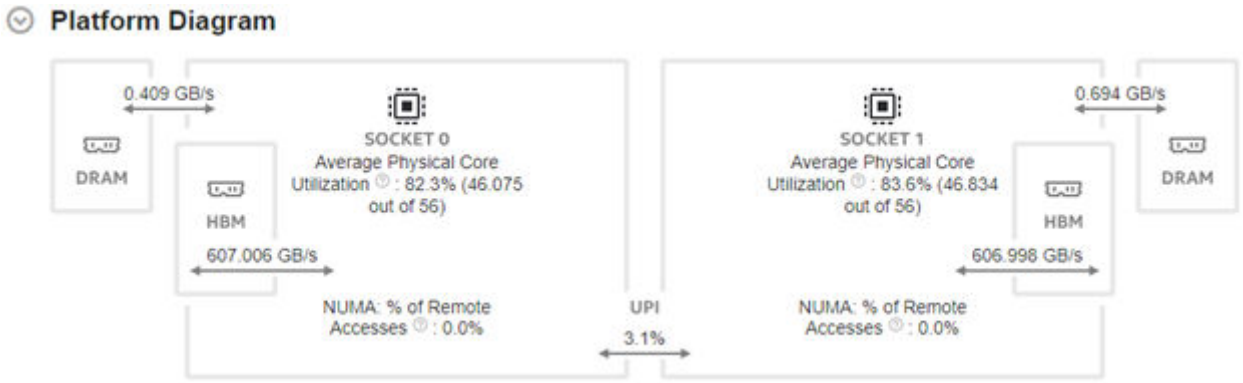


Platform Diagram

Next, see the Platform Diagram, which presents the following information:

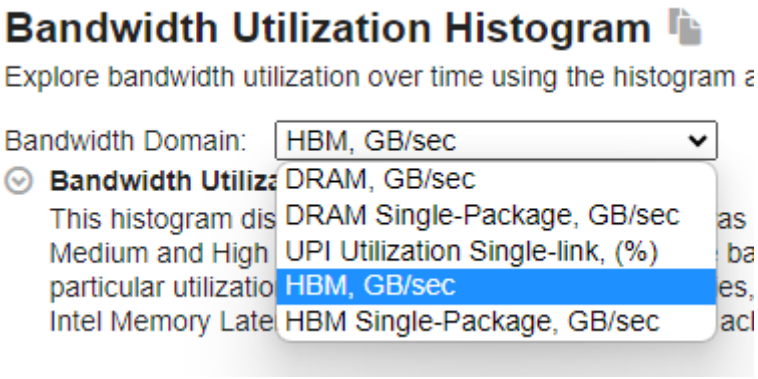
- System topology
- Average DRAM and HBM bandwidths for each package
- Utilization metrics for Intel® Ultra Path Interconnect (Intel® UPI) cross-socket links and physical cores

Suboptimal application topology can cause cross-socket traffic, which in turn can limit the overall performance of the application.



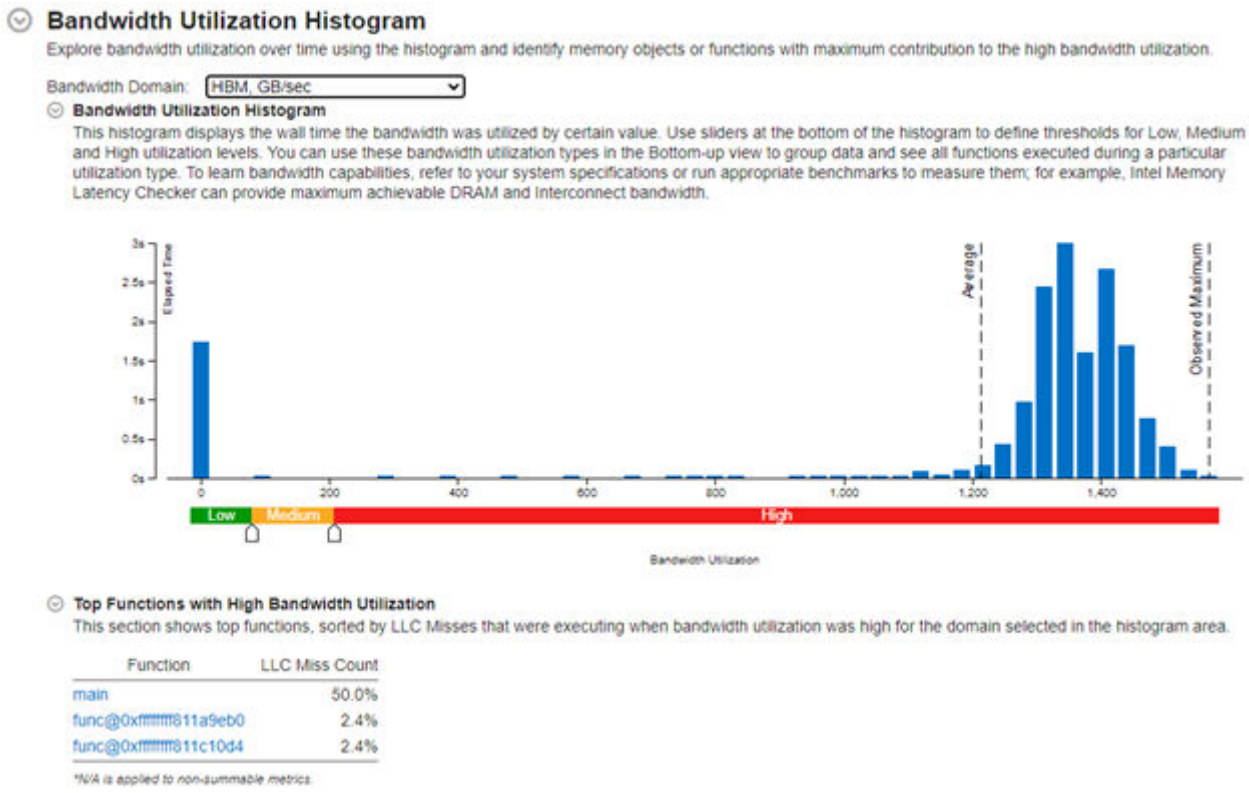
Bandwidth Utilization

In this section, observe the bandwidth utilization in different domains. In this example, the system uses DRAM, UPI, and HBM domains.



NOTE You can see per-socket bandwidth information for DRAM and HBM domains.

To see the overall HBM utilization across the entire system, in the **Bandwidth Domain** pulldown menu, select **HBM, GB/sec**. This information displays in a histogram of bandwidth utilization (GB/sec) vs the aggregated elapsed time (sec) for each bandwidth utilization group.



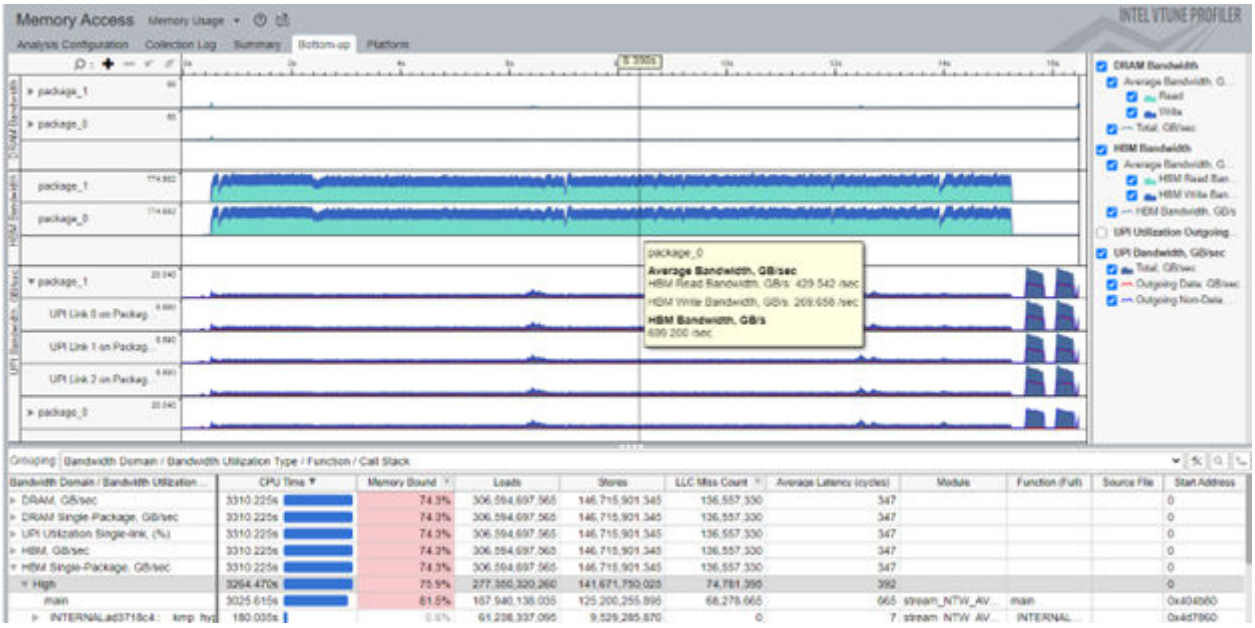
In this example, there is a high utilization of HBM with over 1200 GB/sec for the majority of the duration. This is because the STREAM benchmark is designed to maximize the use of memory bandwidth.

> **NOTE** You can observe the same bandwidth information in the **HPC Performance Characterization analysis** and the **Input and Output Analysis** as well. To do this, make sure to check the **Analyze memory bandwidth** option before running those analyses.

Timeline

Finally, switch to the **Bottom-up** window to observe the timeline. Here, you can examine the following bandwidths over time:

- DRAM bandwidth (broken down per channel)
- HBM bandwidth (broken down per package)
- Intel® UPI links (broken down per link)



Use this information to identify potential issues like misconfiguration which can lead to unnecessary UPI or DRAM bandwidth.

Hover your mouse on the graph to analyze specific parts and see the bandwidth at the selected instant of time.

In the **Grouping** pane, select the **Bandwidth Domain / Bandwidth Utilization / Type / Function / Call Stack** grouping. Use this grouping to identify functions with high utilization in the HBM bandwidth domain.

To further optimize the performance of your application, run these analyses:

- Memory Access analysis
- Input and Output analysis

Follow these analysis procedures to identify other performance issues.

This recipe describes how you measure performance when running the STREAM application in **HBM Flat** mode. To compare performance in the **HBM Caching** and **HBM Only** modes, switch the HBM mode and repeat the performance analysis. Find the mode with the shortest elapsed time. You can also compare DRAM and HBM bandwidths to look for higher overall bandwidth.

See Also

Enabling High-Bandwidth Memory for HPC and AI Applications for Next Gen Intel® Xeon® Processors

Intel® Xeon® CPU Max Series Configuration and Tuning Guide

HBM Benchmarks

Memory Bandwidth Benchmarks

Profiling Windows* Applications for Hybrid CPU Platforms (NEW)

Use this recipe to profile and view hybrid CPU utilization data for Windows-based applications.

A combination of performance cores (P-Cores) and efficient cores (E-Cores) empowers hybrid CPUs to tackle the processing demands of modern workloads, including computer games. Fine tuning your applications enables you to take full advantage of the processing capability of hybrid CPUs, especially if you are using the 12th Generation Intel® Core™ processor.

This recipe describes how you use VTune Profiler to profile and visualize performance data for Windows* applications that run on hybrid CPUs. This recipe highlights two examples with the `asteroids_d3d12` executable in the `HybridDetect` sample. For a description of this sample, see the Game Dev Guide for 12th Gen Intel® Core™ Processor.

Content expert: Jennifer DiMatteo

Ingredients

- **Application**: `asteroids_d3d12` executable in the `HybridDetect` sample
- **Tools**:
  - Intel® VTune™ Profiler version 2023 - Hotspots Analysis (using Hardware Event-based Sampling)
  - Microsoft* Visual Studio - For versions compatible with Intel® VTune™ Profiler, see Intel® VTune™ Profiler System Requirements.
- **CPU/GPU**: Intel® Core™ i7-12700H
- **Operating system**: Windows* 11 Enterprise

Directions

1. Build the Sample
2. Run Hotspots Analysis
3. Review Results
4. Adjust Scheduler Value and Repeat Analysis

Build the Sample in Microsoft* Visual Studio

1. Open the `HybridDetect` sample in Microsoft* Visual Studio.
2. Build the sample.

Run Hotspots Analysis

This procedure describes how you run the hotspots analysis in hardware event-based sampling mode using the standalone version of VTune Profiler. If you are using VTune Profiler integrated into Microsoft Visual Studio, in order to run hardware event-based sampling analysis, you must run Visual Studio as an administrator.

1. Open Intel® VTune™ Profiler and click **New Project** on the Welcome screen.
2. Specify a project name and a location for your project.
3. Click **Create Project**.
4. In the **WHERE** pane of the **Configure Analysis** window, select **Local Host**.
5. In the **WHAT** pane,
   - Fill in the **Application** field with the path to the `asteroids_d3d12` executable.
   - In the **Application parameters** field, enter `-scheduler 0`. This parameter ensures that each of the render and update tasks run on individual threads.
   - In the **Advanced** section, select **Automatically stop collection after** to 30 seconds. Also select **Analyze child processes**.
6. In the **HOW** pane, select the **Hotspots analysis type** and enable **Hardware Event-Based Sampling**.
7. Click **Start** to run the analysis.

The sample shows that the frame rate is variable at 103 frames per second (fps).

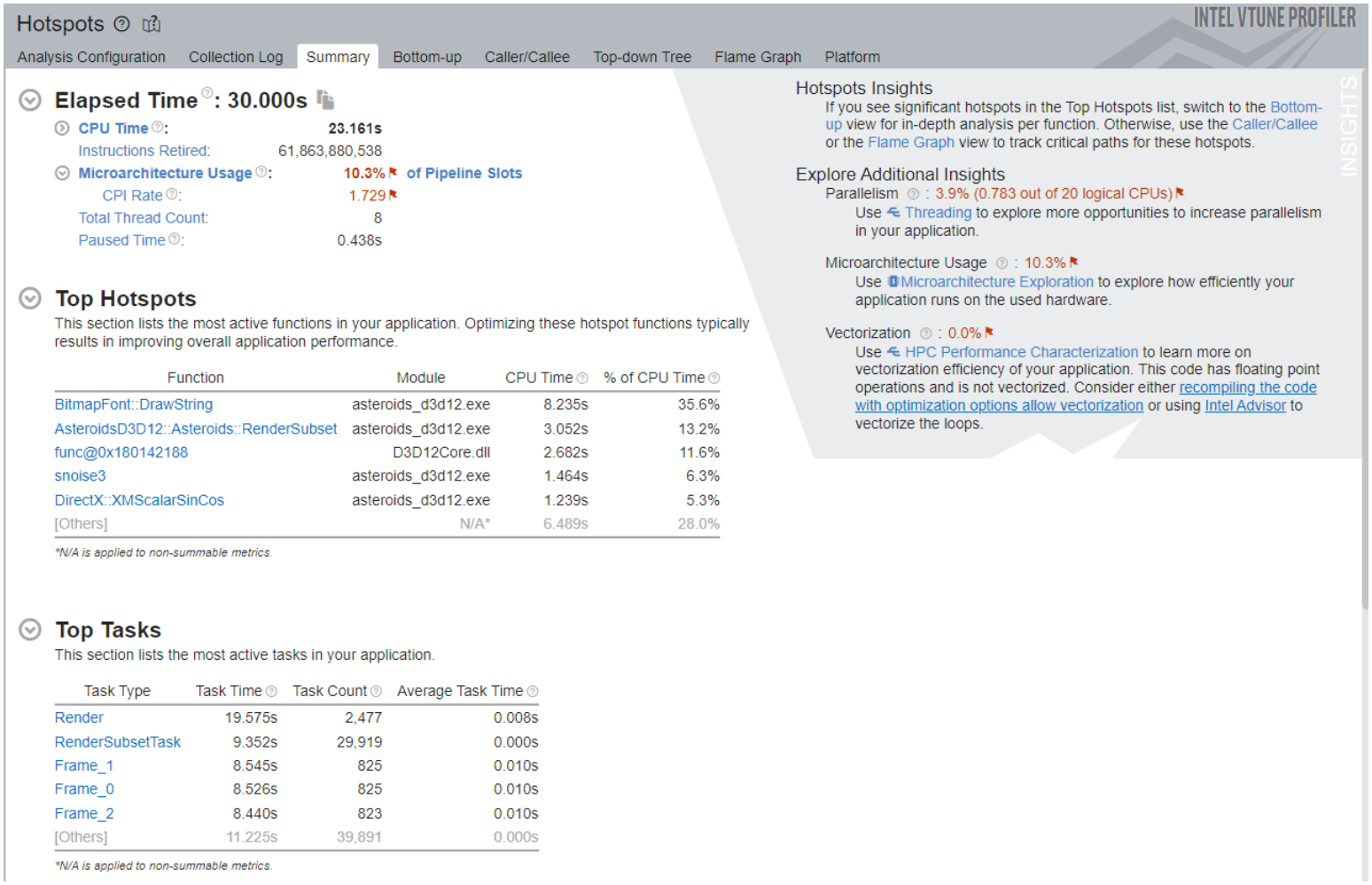


Review Results

After the data collection runs for about 30 seconds, Intel® VTune™ Profiler terminates the application and data collection. Finalizing the results may take a few minutes as Intel® VTune™ Profiler finds and resolves debug symbols.

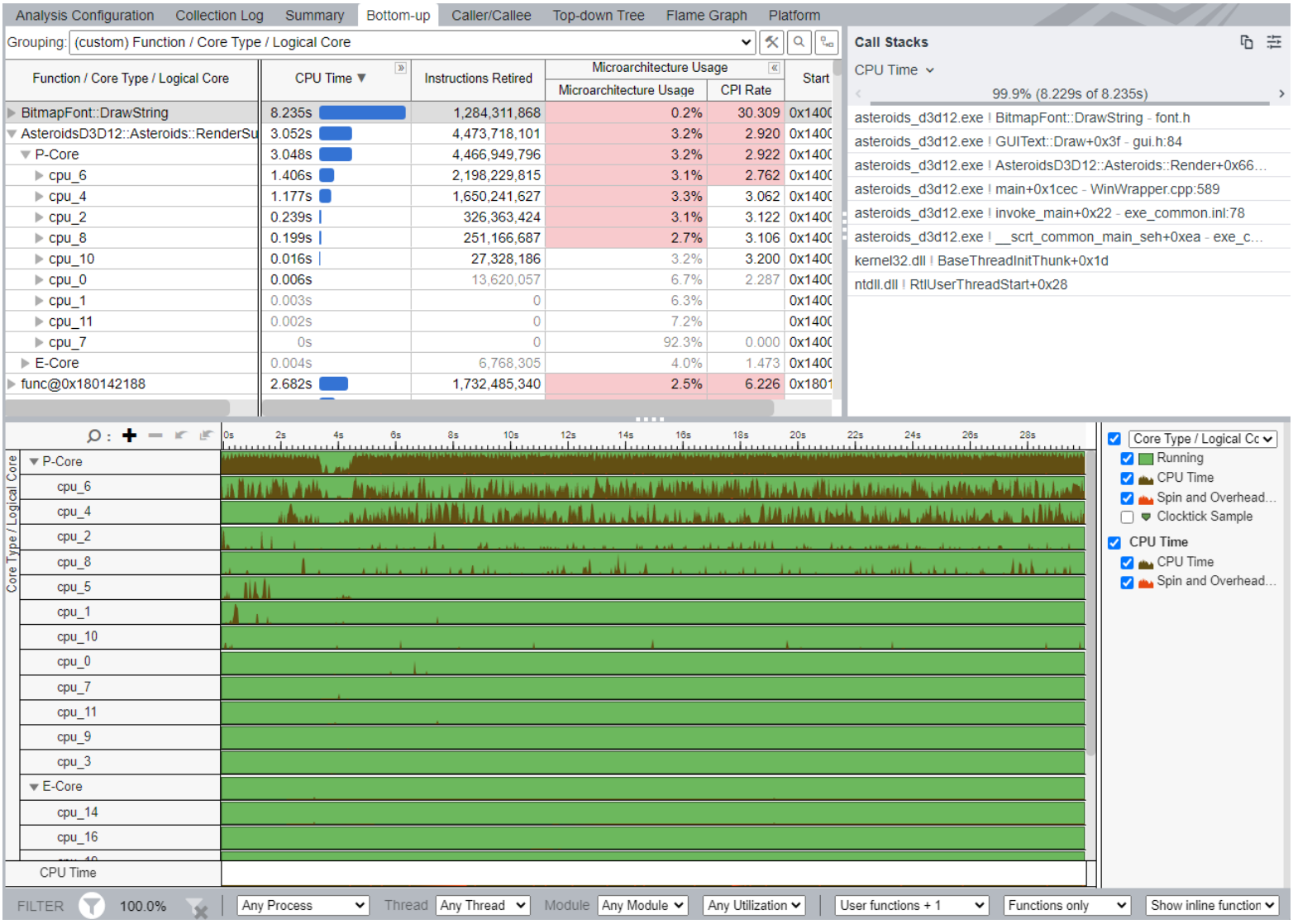
Once results have been finalized, the **Summary** tab displays information about:

- Elapsed time
- Top hotspots
- Top tasks
- Additional insights and guidance



In this example, the top task is `DrawString` with a high clock cycles per instruction (CPI) rate of 1.729. This means that the execution of instructions is slower than optimal. The thread count is eight, but actual parallelism in the executable is very low. You can infer this detail from the **Additional Insights** section in the upper right corner. The application is executing instructions very slowly and only on a single CPU.

Next, look at the **Bottom-up** window.



Customize Results by Core Type

To see how the application uses P-Cores and E-Cores, create a custom grouping. Click the tool icon next to the **Grouping** pulldown menu and group results by **Function/Core Type/Logical Core**.

Next, expand the `RenderSubset` function. You can see that this function ran on two logical P-Cores.

In this manner, you see from the timeline that the entire application used only two P-Cores and hardly any E-Cores. This implies that any execution that happened on E-Cores was too minimal for Intel® VTune™ Profiler to collect and use the data confidently.

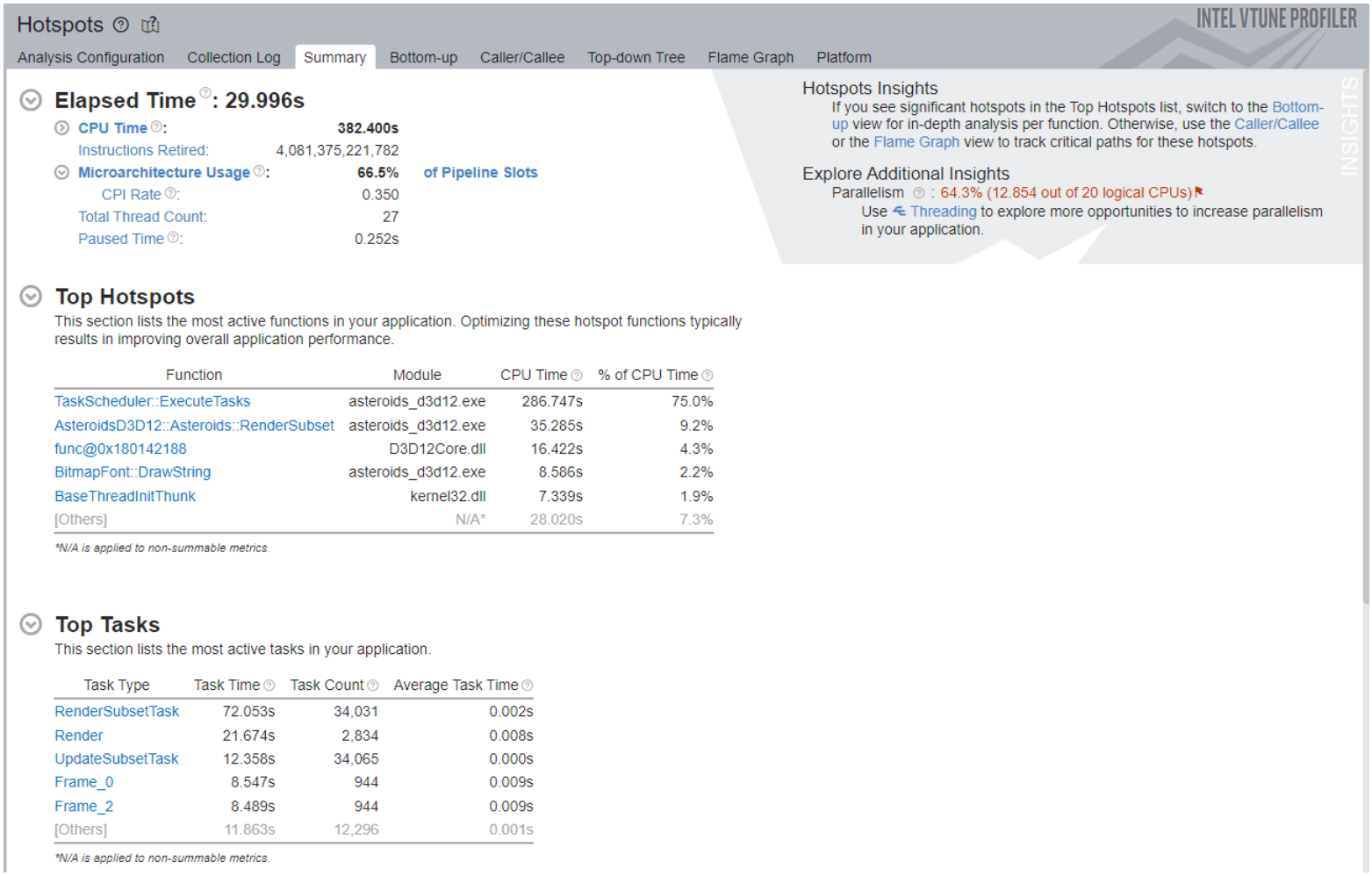
Adjust Scheduler Value and Repeat Analysis

Repeat the hotspots analysis, but this time, set **Application parameters** to `-scheduler 1`. This setting ensures that the number of render tasks is equal to the number of P-Cores. Also, there are eight update tasks which now run independent of the render tasks.

Once you repeat the analysis, the application shows that 12 tasks are running. The average frame rate is 119 fps.



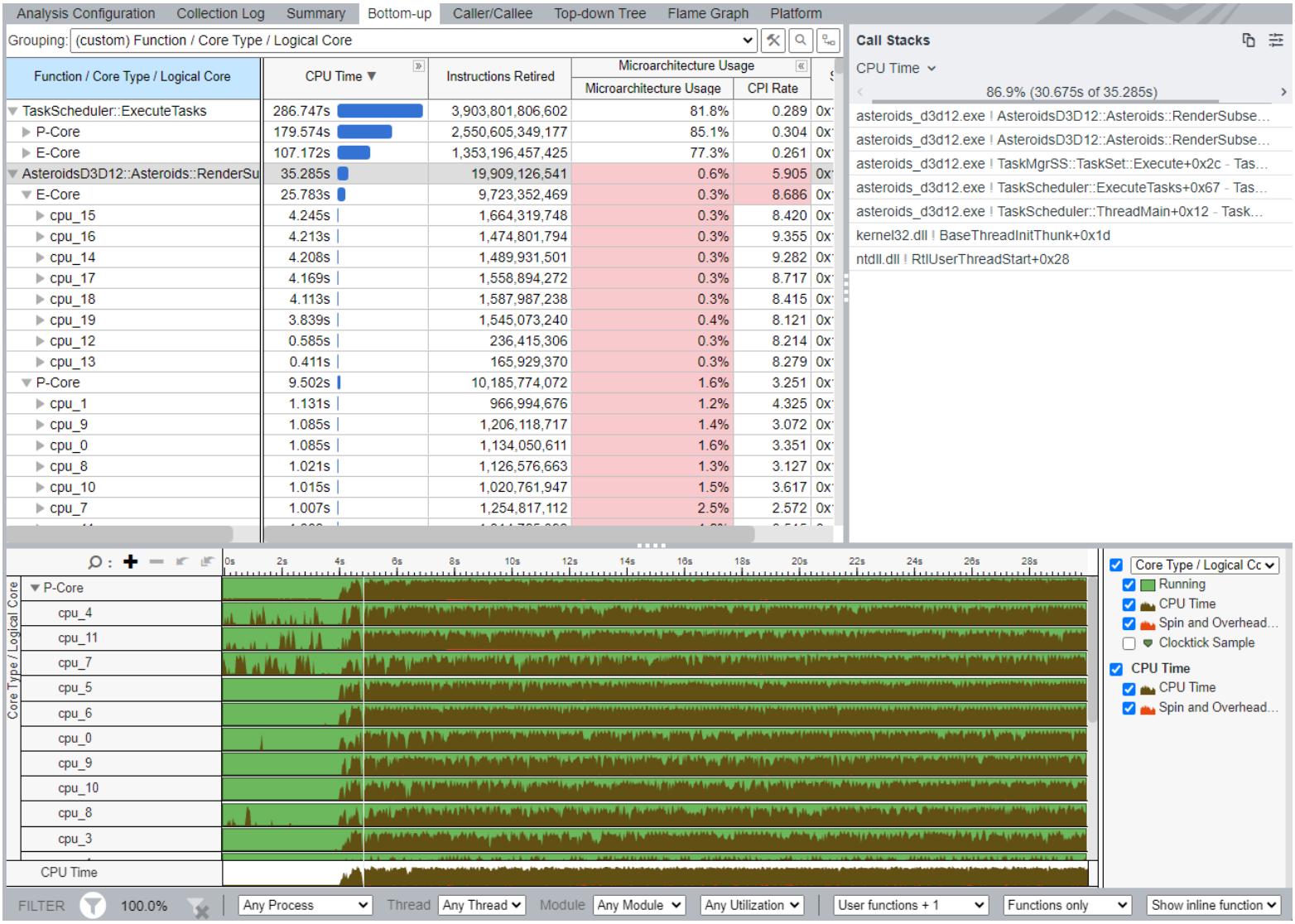
When the data collection completes, Intel® VTune™ Profiler finalizes results. The **Summary** window opens with this information:



The parallelism is now much higher. Instead of using one logical CPU, the application used over 12 CPUs.

This time, the top function is the `TaskScheduler`. The total thread count is 27. At 0.35, the CPI rate is much lower now. This may be because the scheduler had a very low CPI rate and accounted for most of the CPU time.

It is important to ensure that the render performance has actually improved, so that the results are not skewed by overhead. The **Bottom-up** window shows that the CPI rate of the render function improved.



However, the render function ran almost 20 million instructions this time, compared to 4.5 million instructions the previous time. This work was split almost evenly between P-Cores and E-Cores, although P-Cores ran the instructions 2x faster. Again, the timeline shows the overall utilization of cores.

The `HybridDetect` sample has several configurations which you can use to understand how to control CPU utilization on a hybrid platform. This recipe uses the default pre-compiler macros available in the sample. These macros give you control over the render and update threads, but they let the Intel Thread Detector determine the core type to be used.

As an additional exercise, force the render tasks to run on P-Cores and update tasks to run on E-Cores. To do this, enable the `ENABLE_RUNON` macro in `HybridDetect.h`. See how this change affects performance.

See Also

Hybrid CPU Analysis

Hotspots Analysis for CPU Usage Issues

Game Dev Guide for 12th Gen Intel® Core™ Processor

Game Tuning with Intel

Viewing Analysis Results on a Web Browser (NEW)

Use techniques in this recipe to view analysis results from Intel® VTune™ Profiler in a web browser, particularly on systems where you cannot install VTune Profiler to run the analyses.

Typically, you install VTune Profiler on a host machine and use it to profile an application target on the same machine. Sometimes the host and target operating systems may be different. Depending on the specific operating environment, you may not be able to open the VTune Profiler GUI on the host machine, with access limited to the command line only. If you use a macOS* system, you cannot run VTune Profiler for data collection or viewing, as the platform is not supported. When you are unable to use the VTune Profiler user interface, you can open analysis results in a web browser instead.

Use a web browser to view analysis results when:

- You need easy access. Multiple users across multiple systems can access the results.
- You need to manage a single repository. When you have a large volume of results, save them on a system with sufficient storage without having to copy the results to local systems.
- You want to manage fewer versions of VTune Profiler. You use a single viewer and avoid mismatches between the versions of VTune Profiler used for data collection and viewing.
- You want to improve the finalization of results. For better resolution of symbols, store the debug versions of binaries and sources in a single location.
- You want to view results without relying on the VTune Profiler UI. See analysis results on a Linux server without having to install VTune Profiler.

Content expert: Jennifer DiMatteo

Ingredients

- **Application**: `matrix` sample available with the installation of VTune Profiler
- **Tools**: Intel® VTune™ Profiler version 2023 or newer
- **Browser**: Google Chrome*
- **Operating system**: Windows* 11 Enterprise, Ubuntu 22.04

Directions

1. Configure Your System for Intel® VTune™ Profiler Web Server
2. Start Intel® VTune™ Profiler Web Server
3. Run a Collection

Configure Your System for Intel® VTune™ Profiler Web Server

To use Intel® VTune™ Profiler web server, choose a system that has:

- Plenty of storage. A single Intel® VTune™ Profiler result can be over 1 GB in size.
- An open web port for user access. You specify this port for use when you run Intel® VTune™ Profiler from the command line.

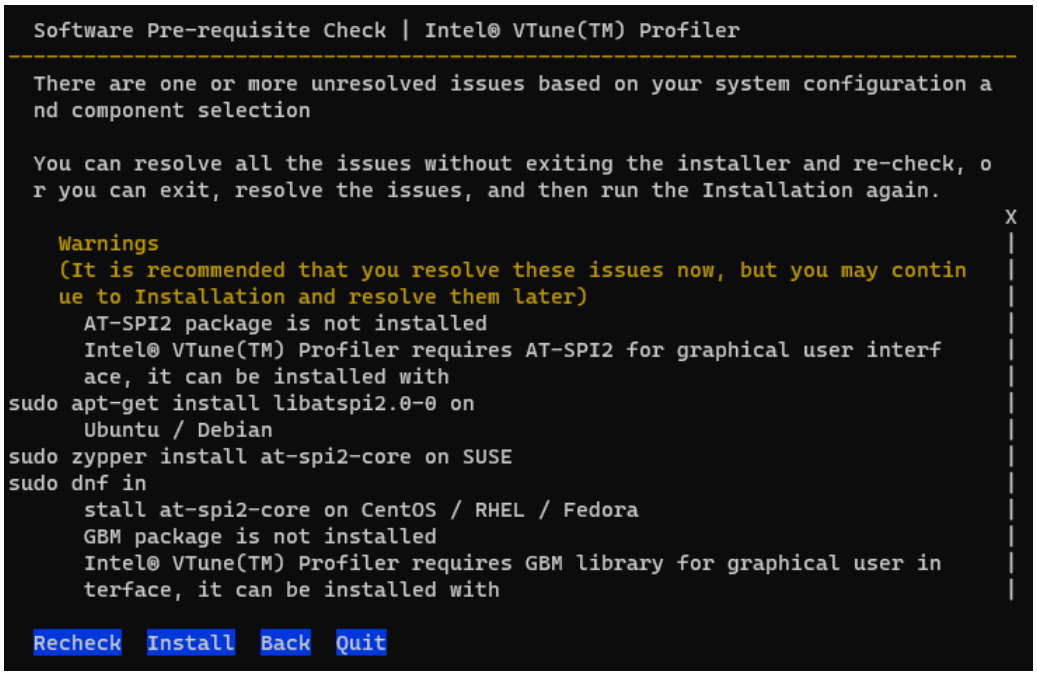
Additionally, when you profile remote systems, ensure these details:

- Ensure that the system you use to run the web server has the same operating system as the target system(s) used for data collection. These should be Linux or Windows machines. This is necessary for proper SSH communication between the server and the target. If you copy results manually into the server result directory, you can use results from any OS.
- Enable SSH communication between the web server and the target(s).

Start Intel® VTune™ Profiler Web Server

This example uses an installation of Intel® VTune™ Profiler on a Linux system.

1. Install Intel® VTune™ Profiler as root or administrator with default configurations. If you install on a Linux OS, you may encounter warnings about missing GUI libraries. You can ignore these warnings as the libraries are not required to run the web server.



2. Select a user who can run the web server. The credentials of this user are used to run the data collection, including the availability of hardware profiling and access to the target process. For example, to use the hardware driver of Intel® VTune™ Profiler, the user must have group access. The default group is vtune. For more information about using the hardware driver, see Sampling Drivers.
3. Start the Intel® VTune™ Profiler web server. Use the `vtune-backend` command. Configure these options:

Option	Purpose
`web-port`	This port must be accessible by remote connections. If this port is not specified, Intel® VTune™ Profiler chooses a random port that is available.
`data-directory`	Intel® VTune™ Profiler searches for results in this directory. By default, this is in the home directory of the user who started the web service. If you provide a custom directory, ensure that the user starting the web service has access to the custom directory.
`enable-server-profiling`	This option enables the web service to profile the server on which it runs.
`allow-remote-access`	Use this option to enable browser access from systems other than `localhost`.

Use these commands:

- Linux OS:

```
$ vtune-backend -web-port=8080 -data-directory=/mnt/vtune-results/ --enable-server-profiling -
allow-remote-access &
```

- Windows OS:

```
C:\Program Files (x86)\Intel\oneAPI\vtune\latest\bin64\vtune-backend --web-port=55012 --allow-
remote-access --enable-server-profiling
```

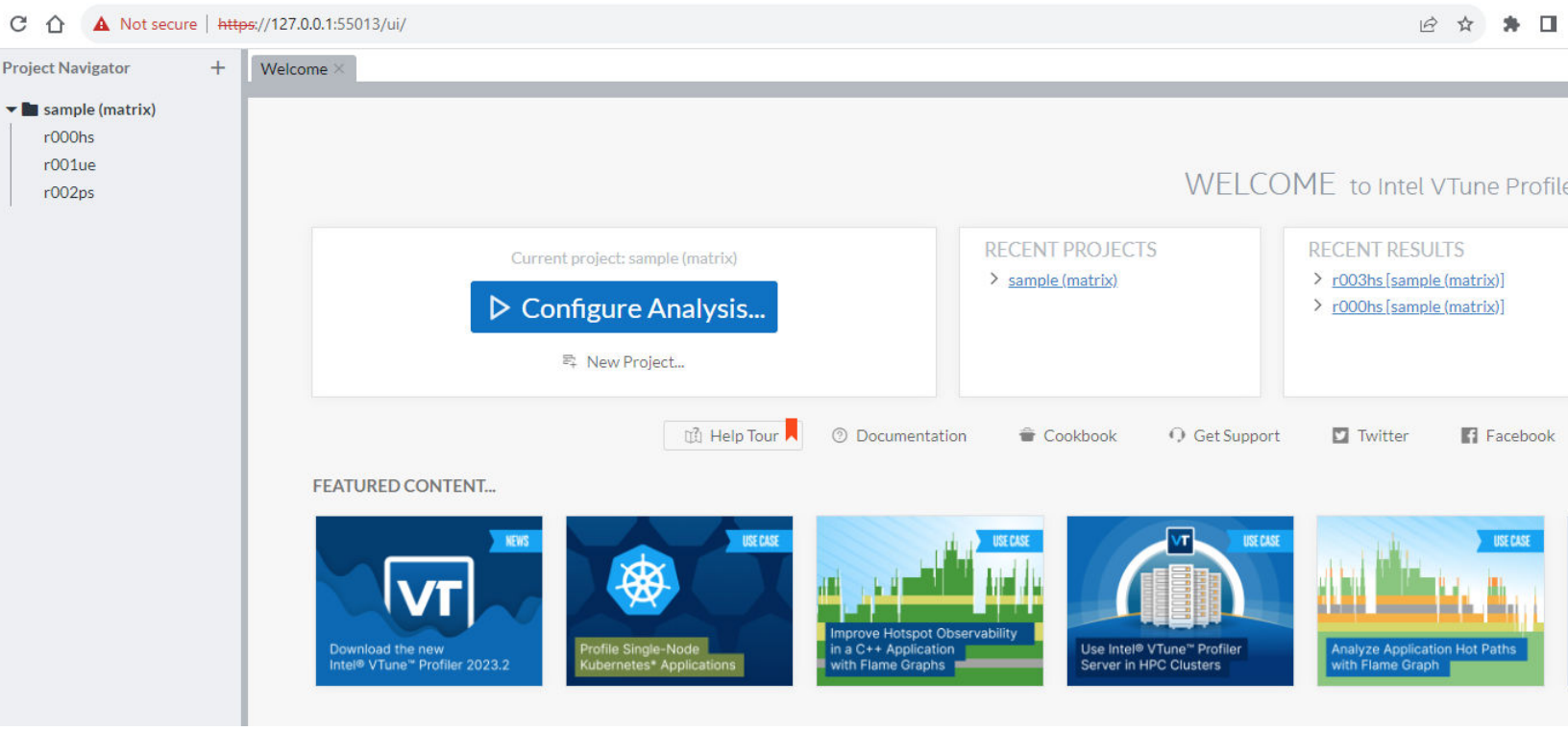
When the web service begins, a URL displays which you can paste into your local browser. If this is the first time you are starting web service on the system, the URL contains a one-time token you use to set a security passphrase.

```
PS C:\Program Files (x86)\Intel\oneAPI\vtune\2024.0\bin64> ./vtune-backend --web-port=55012 --allow-remote-
e-server-profiling
certificate was provided as a --tls-certificate command-line argument thus a self-signed certificate is ger
le secure HTTPS transport for the web server: C:\Users\dimatteo\AppData\Roaming\Intel\VTune\settings\certif
are.crt.
rofiler GUI is accessible via https://10.209.21.241:55012/?one-time-token=b71f9b87210ef7e91061c47d920d9e58
rofiler GUI is accessible via https://192.168.1.5:55012/?one-time-token=b71f9b87210ef7e91061c47d920d9e58
  VTune Profiler Agent will be connected to https://10.209.21.241:55012/?one-time-token=b71f9b87210ef7e9106
```

NOTE This procedure does not enable an official TLS certificate. When you connect to the server through a web browser, you may see warnings.

Run a Collection

The installation package of Intel® VTune™ Profiler includes a `matrix multiply` sample with results. If you use the default data directory, these results display in the user interface on the browser.

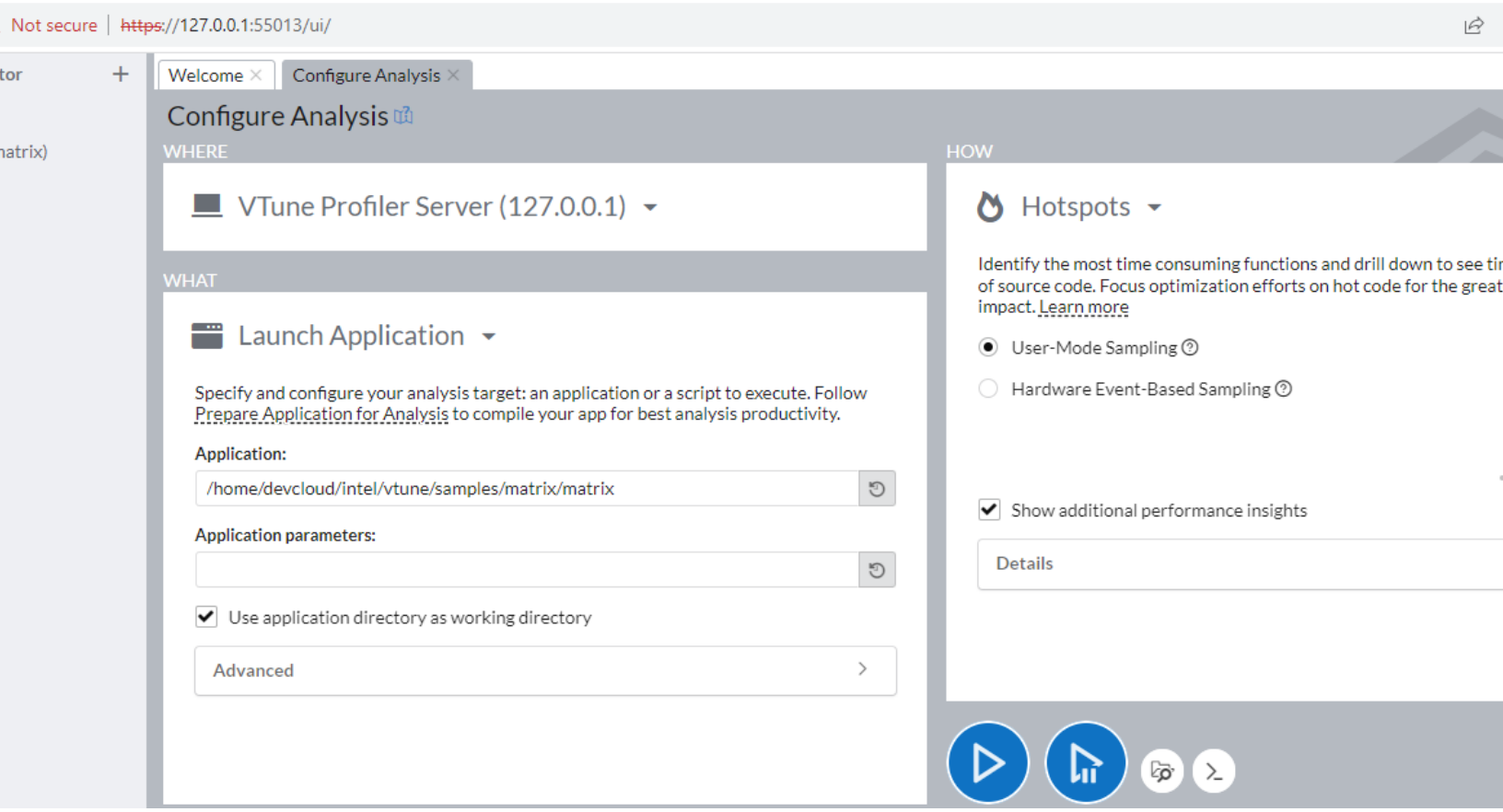


You can run a data collection in these ways:

Method	Notes
Run a collection on the web server.	Use the default settings in the WHERE pane.
Configure a remote system.	You must install the collection agent on the target. You can do this manually or automatically. For more information, see Deploy the Intel® VTune™ Profiler Agent .

Method	Notes
Run a collection on the target and copy results over.	Install Intel® VTune™ Profiler manually on the target OS. Run a data collection and then copy the results into the data directory of the server. This method is useful if remote collection is not possible due to SSH or permission issues. You can also use this method to view results without needing to run a new data collection.

This example demonstrates the first method to run a Hotspots analysis on the `matrix` sample on the web server. For this purpose, create a new project (called `new_test`) so you can see how results are structured on the server:



Once results have been collected, they are available in the `new_test` data directory:

```
devcloud@a4bf0190563d:~/intel/vtune/projects$ ls
 new_test  'sample (matrix)'
devcloud@a4bf0190563d:~/intel/vtune/projects$ cd new_test/
devcloud@a4bf0190563d:~/intel/vtune/projects/new_test$ ls
new_test.vtuneproj  r000hs
devcloud@a4bf0190563d:~/intel/vtune/projects/new_test$
```

When you collect or open Intel® VTune™ Profiler results this way, you do not install or copy anything onto your local system. Intel® VTune™ Profiler and the collected results exist only on the remote devcloud system.

See Also

Install Intel® VTune™ Profiler Server

Intel® VTune™ Profiler Web Server Interface

Profiling Machine Learning Applications (NEW)

Learn how to use Intel® VTune™ Profiler to profile Machine Learning (ML) workloads.

In our increasingly digital world powered by software and web-based applications, Machine Learning (ML) applications have become extremely popular. The ML community uses several Deep Learning (DL) frameworks like Tensorflow*, PyTorch*, and Keras* to solve real world problems.

However, understanding computational and memory bottlenecks in DL code like Python or C++ is challenging and often requires significant effort due to the presence of hierarchical layers and non-linear functions. Frameworks like Tensorflow* and PyTorch* provide native tools and APIs that enable the collection and analysis of performance metrics during different stages of Deep Learning Model development. But the scope of these profiling APIs and tools is quite limited. They do not provide deep insight at the hardware level to help you optimize different operators and functions in the Deep Learning models.

In this recipe, learn how you can use VTune Profiler to profile a Python workload and improve data collection with additional APIs.

Content Expert: Rupak Roy

- INGREDIENTS
- DIRECTIONS:

  1. Run VTune Profiler on a Python Application
  2. Include Intel® Instrumentation and Tracing Technology (ITT)-Python APIs
  3. Run Hotspots and Microarchitecture Exploration Analyses
  4. Add PyTorch* ITT APIs (for PyTorch Framework only)
  5. Run Hotspots Analysis with PyTorch ITT APIs

Ingredients

Here are the hardware and software tools you need for this recipe.

- **Application**: This recipe uses the `TensorFlow_HelloWorld.py` and `Intel_Extension_For_PyTorch_Hello_World.py` applications. Both of these code samples are implementations of a simple neural network with a convolution layer, a normalization layer, and a ReLU layer which can be trained and evaluated.
- **Analysis Tool**: User-mode sampling and tracing collection with VTune Profiler (version 2022 or newer)

  NOTE

  - Starting with the 2020 release, Intel® VTune™ Amplifier has been renamed to Intel® VTune™ Profiler.
  - Most recipes in the Intel® VTune™ Profiler Performance Analysis Cookbook are flexible. You can apply them to different versions of Intel® VTune™ Profiler. In some cases, minor adjustments may be required.
  - Get the latest version of Intel® VTune™ Profiler:
    - From the Intel® VTune™ Profiler product page.
    - Download the latest standalone package from the Intel® oneAPI standalone components page.

- **CPU**: 11th Gen Intel® Core(TM) i7-1165G7 @ 2.80GHz
- **Operating System**: Ubuntu Server 20.04.5 LTS

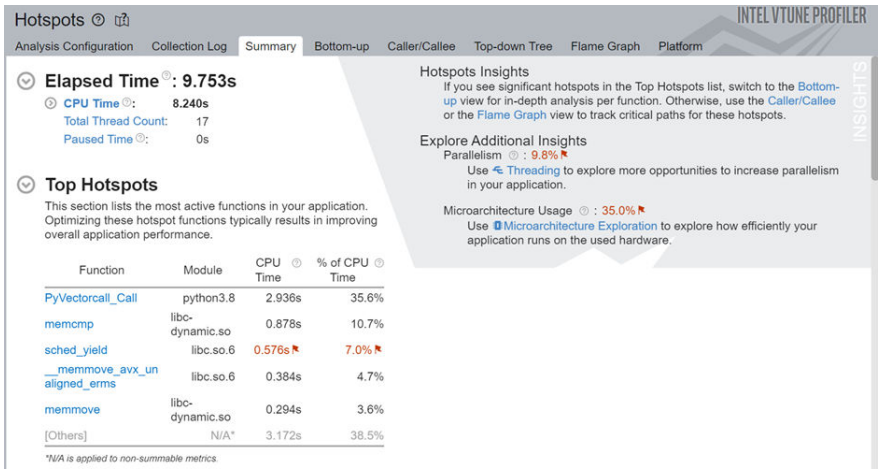
Run VTune Profiler on a Python Application

Let us start by running a Hotspots analysis on the `Intel_Extension_For_PyTorch_Hello_World.py` ML application, without any change to the code. This analysis is a good starting point to identify the most time-consuming regions in the code.

In the command line, type:

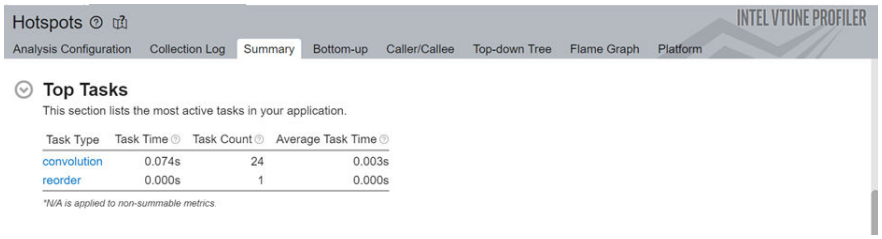
```
vtune -collect hotspots -knob sampling-mode=sw -knob enable-stack-collection=true -source-search-dir=path_to_src -search-dir /usr/bin/python3 -result-dir vtune_hotspots_results -- python3 Intel_Extension_For_PyTorch_Hello_World.py
```

Once the analysis completes, see the most active functions in the code by examining the **Top Hotspots** section in the **Summary** window.



In this case, we see that the `sched_yield` function consumes considerable CPU time. Excessive calls to this function can cause unnecessary context switches and result in a degradation of application performance.

Next, let us look at the top tasks in the application:



Here we can see that the convolution task consumes the most processing time for this code.

While you can dig deeper by switching to the Bottom-up window, it may be challenging to isolate the most interesting regions for optimization. This is particularly true for larger applications because there may be a lot of model operators and functions in every layer of the code. Therefore, we will now add Intel® Instrumentation and Tracing Technology (ITT) APIs to generate results that are easier to interpret.

Include ITT-Python APIs

Let us now add Python* bindings available in ITT-Python to the Intel® Instrumentation and Tracing Technology (ITT) APIs used by VTune Profiler. These bindings include user task labels to control data collection and some user task APIs (that can create and destroy task instances).

ITT-Python uses three types of APIs:

- Domain APIs
  - `domain_create(name)`

- Task APIs
  - `task_begin(domain, name)`
  - `task_end(domain)`
- Anomaly Detection APIs
  - `itt_pt_region_create(name)`
  - `itt_pt_region_begin(region)`
  - `itt_pt_region_end(region)`

The following example from `TensorFlow_HelloWorld.py` calls the Domain and Task APIs in ITT-Python:

```
itt.resume()
domain = itt.domain_create("Example.Domain.Global")
itt.task_begin(domain, "CreateTrainer")
for epoch in range(0, EPOCHNUM):
  for step in range(0, BS_TRAIN):
    x_batch = x_data[step*N:(step+1)*N, :, :, :]
    y_batch = y_data[step*N:(step+1)*N, :, :, :]
    s.run(train, feed_dict={x: x_batch, y: y_batch})
    '''Compute and print loss. We pass Tensors containing the predicted and    true values of y,
and the loss function returns a Tensor containing the loss.'''
    print(epoch, s.run(loss,feed_dict={x: x_batch, y: y_batch}))
itt.task_end(domain)
itt.pause()
```

Here is the sequence of operations:

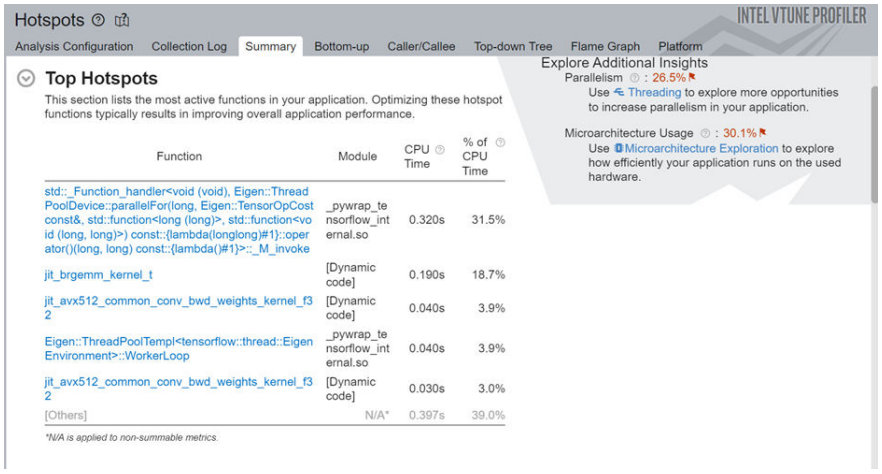
1. Use `itt.resume()` API to resume the profiling just before the loop begins to execute.
2. Create an ITT domain (like `Example.Domain.Global`) for a majority of the ITT API calls.
3. Use the `itt.task.begin()` API to start the task. Label the task as `CreateTrainer`. This label appears in profiling results.
4. Use `itt.task()` API to end the task.
5. Use `itt.pause()` API to pause data collection.

Run Hotspots and Microarchitecture Exploration Analyses

Once you have modified your code, run the Hotspots analysis on the modified code.

```
vtune -collect hotspots -start-paused -knob enable-stack-collection=true -knob sampling-mode=sw
-search-dir=/usr/bin/python3 -source-search-dir=path_to_src  -result-dir vtune_data -- python3
TensorFlow_HelloWorld.py
```

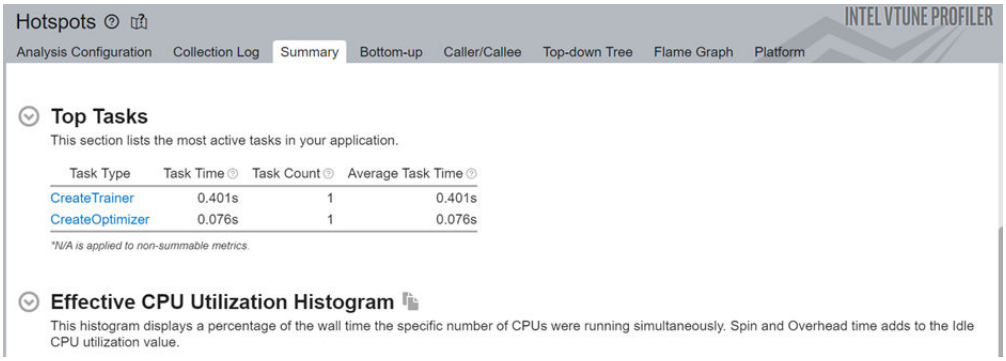
This command uses the `-start-paused` parameter to profile only those code regions marked by ITT-Python APIs. Let us look at the results of the new Hotspots analysis. The **Top Hotspots** section displays hotspots in the code regions marked by ITT-Python APIs.



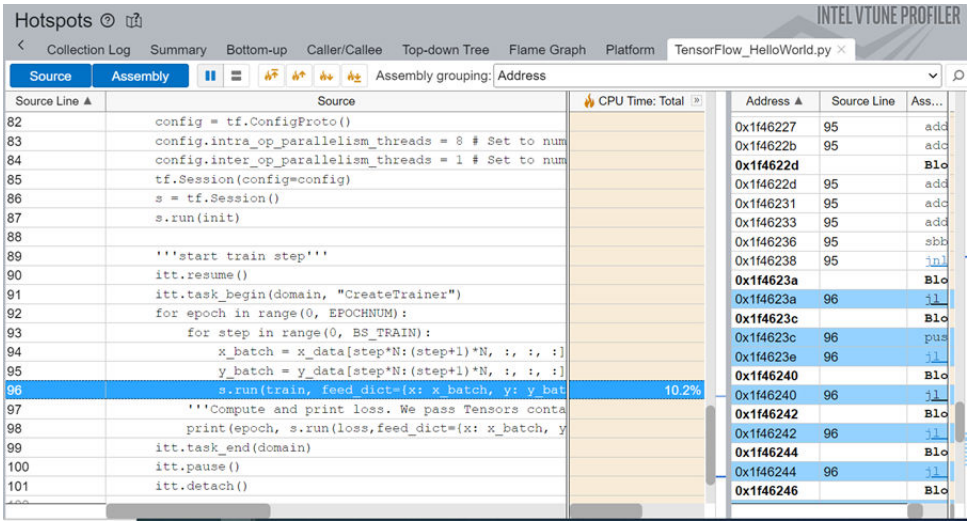
Examine the most time-consuming ML primitives in the target code region. Focus on these primitives first to improve optimization. Using the ITT-APIs helps you identify those hotspots quickly which are more pertinent to ML primitives.

Next, look at the top tasks targeted by the ITT-Python APIs. Since you can use these APIs to limit profiling results to specific code regions, the ITT logical tasks in this section display information including:

- CPU time
  - Effective time
  - Spin time
  - Overhead time
- CPU utilization time
- Source code line level analysis



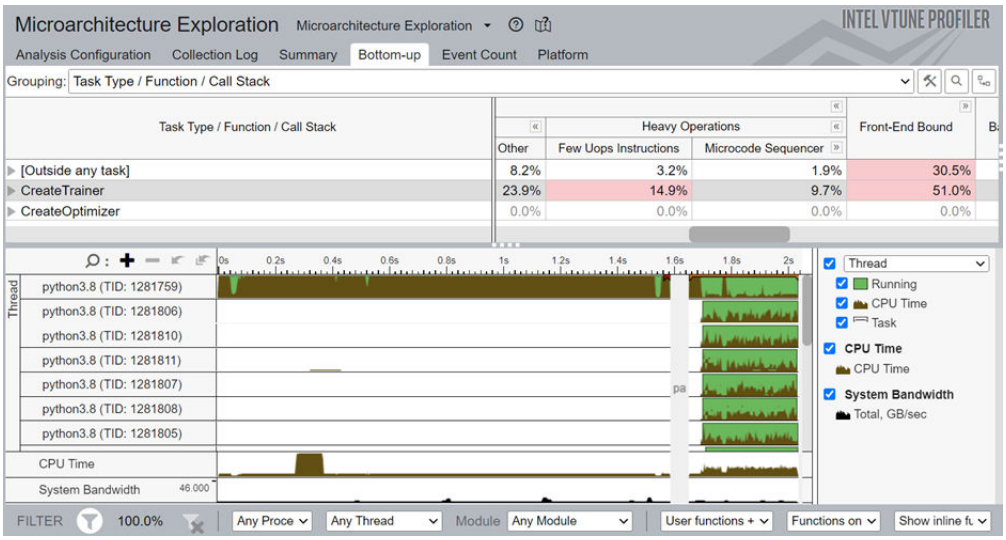
The source line level profiling of the ML code reveals the source line breakdown of CPU time. In this example, the code spends 10.2% of the total execution time to train the model.



To obtain a deeper understanding of application performance, let us now run the Microarchitecture Exploration analysis. In the command window, type:

```
vtune -collect uarch-exploration -knob collect-memory-bandwidth=true  -source-search-dir=path_to_src -search-dir /usr/bin/python3 -result-dir vtune_data_tf_uarch -- python3 TensorFlow_HelloWorld.py
```

Once the analysis completes, the **Bottom-up** window displays detailed profiling information for the tasks marked with ITT-Python APIs. We can see that the `CreateTrainer` task is front-end bound, which means that the front end is not supplying enough operations to the back end. Also, there is a high percentage of heavy-weight operations (those which need more than 2 µops).



To focus your analysis on a smaller block of code, right click on one of the `CreateTrainer` tasks and enable filtering.

Add PyTorch* ITT APIs (for PyTorch Framework only)

Just like ITT-Python APIs, you can also use PyTorch* ITT APIs with VTune Profiler. Use PyTorch ITT APIs to label the time span of individual PyTorch operators and get detailed analysis results for customized code regions. PyTorch 1.13 provides these versions of `torch.profiler.itt` APIs for use with VTune Profiler:

- `is_available()`
- `mark(msg)`

- `range_push(msg)`
- `range_pop()`

Let us see how these APIs are used in a code snippet from Intel_Extension_For_PyTorch_Hello_World.py.

```
itt.resume()
with torch.autograd.profiler.emit_itt():
        torch.profiler.itt.range_push('training')
        model.train()
        for batch_index, (data, y_ans) in enumerate(trainLoader):
            data = data.to(memory_format=torch.channels_last)
            optim.zero_grad()
            y = model(data)
            loss = crite(y, y_ans)
            loss.backward()
            optim.step()
        torch.profiler.itt.range_pop()
itt.pause()
```

The above example features this sequence of operations:

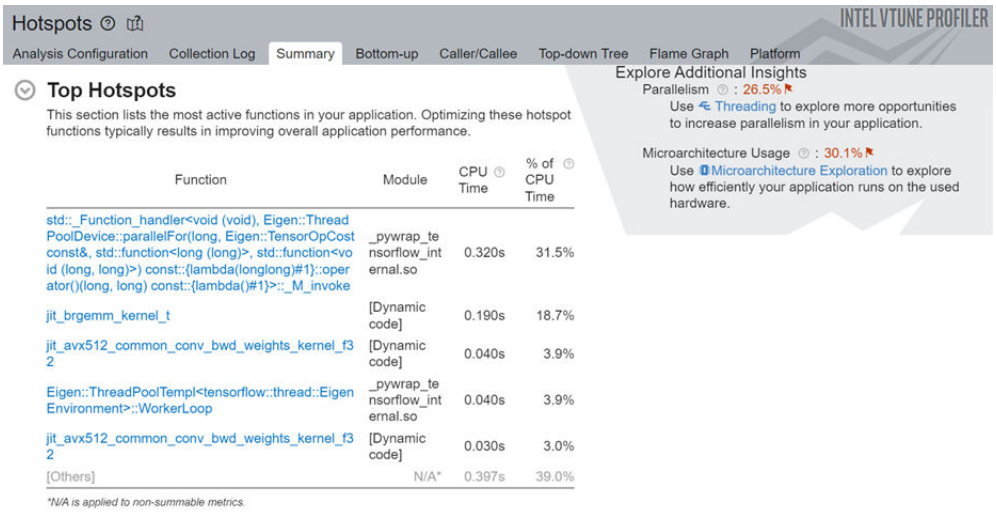
1. Use the `itt.resume()` API to resume the profiling just before the loop begins to execute.
2. Use the `torch.autograd.profiler.emit_itt()` API for a specific code region to be profiled.
3. Use the `range_push()` API to push a range onto a stack of nested range spans. Mark it with a message ('*training*').
4. Insert the code region of interest.
5. Use the `range_pop()` API to pop a range from the stack of nested range spans.

Run Hotspots Analysis with PyTorch ITT APIs

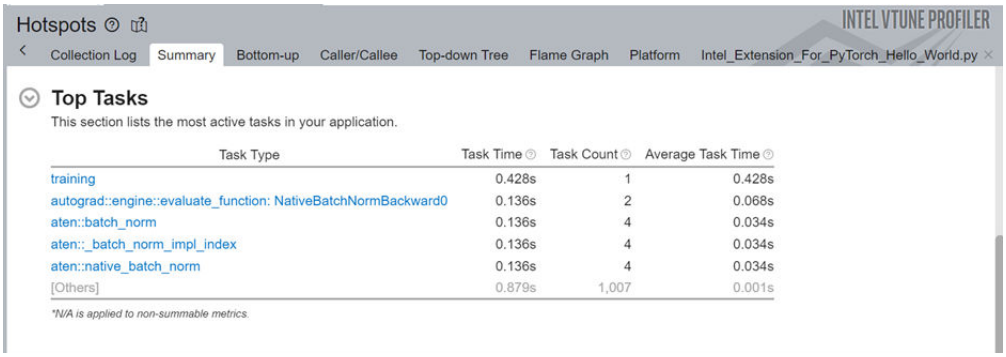
Let us now run the Hotspots analysis for the code modified with PyTorch ITT APIs. In the command window, type:

```
vtune -collect hotspots -start-paused -knob enable-stack-collection=true -knob sampling-mode=sw -
search-dir=/usr/bin/python3 -source-search-dir=path_to_src  -result-dir
vtune_data_torch_profiler_comb -- python3 Intel_Extension_For_PyTorch_Hello_World.py
```

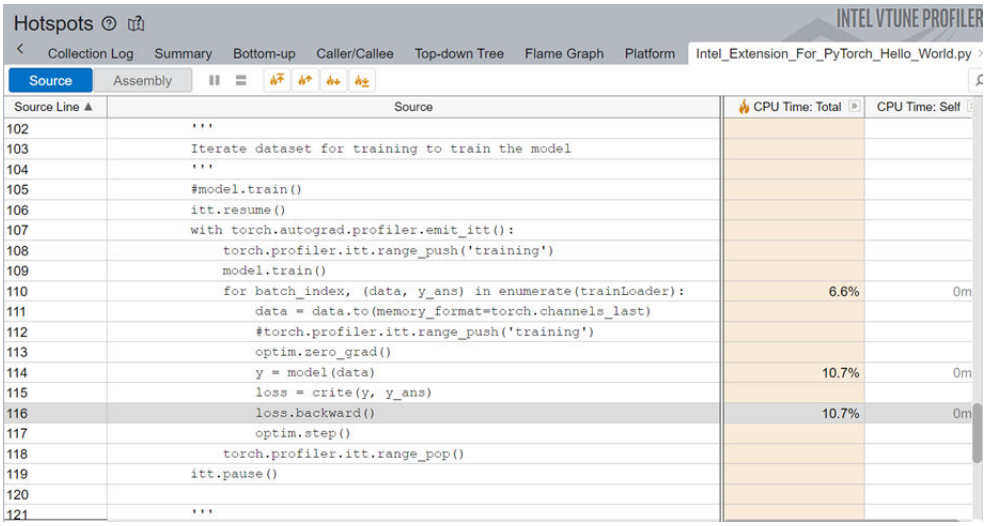
Here are the top hotspots in our code region of interest:



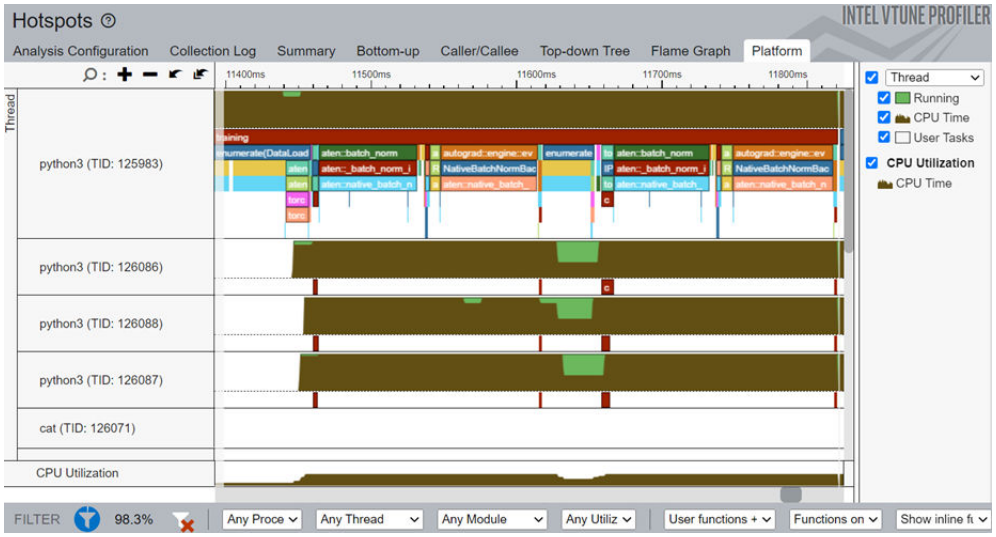
In the **Top Tasks** section of the **Summary**, we see the `training` task which was labeled using the ITT-API.



When we examine the source line profiling of the PyTorch code, we see that the code spends 10.7% of the total execution time in backpropagation.



Switch to the **Platform** window to see the timeline for the `training` task, which was marked using PyTorch ITT APIs.



In the timeline, the main thread is python3(TID:125983) and it contains several smaller threads. Operator names that start with `aten::batch_norm`, `aten::native_batch_norm`, `aten::batch_norm_i` are model operators.

From the Platform window, you can glean these details:

- CPU usage (for a specific time period) for every individual thread
- Start time
- Duration of user tasks and oneDNN primitives(Convolution, Reorder)
- Source lines for each task and primitive. Once the source file for a task/primitive is compiled with debug information, click on the task/primitive to see the source lines.
- Profiling results grouped by iteration number (when multiple iterations are available)

See Also

Intel® Optimization for TensorFlow*
Intel® Optimization for PyTorch*

Profiling Single-Node Kubernetes* Applications (NEW)

Learn how to use Intel® VTune™ Profiler to profile Kubernetes applications deployed in single-node environments.*

Content Expert: Alexey Kireev

Kubernetes* applications are popularly deployed in multi-node environments, where aspects like scalability and durability are important advantages. However there are other advantages when you deploy Kubernetes applications in single-node environments. You can expect a better experience in terms of deployment and management of containerized workloads, in addition to standard containerization features. You can use VTune Profiler to profile Kubernetes* applications in single-node environments and Kubernetes pods with multiple containers running simultaneously.

Follow this recipe to configure a single Kubernetes node and use VTune Profiler to analyze one or more pods running Docker* containers. This recipe employs the Java* code analysis capabilities of VTune Profiler.

- INGREDIENTS
- DIRECTIONS:

  1. Configure a Kubernetes Pod
  2. Run Hardware Event-Based Hotspots Analysis with VTune Profiler and Target in the Same Pod
  3. Run Hotspots Analysis in User-Mode Sampling on a Pod Target
  4. Run Profile System Analysis for Pods with Multiple Containers

Ingredients

Here are the hardware and software tools you need for this recipe:

- **Application**:
  - `MatrixMultiplication` - This is a Java application that is used as a demo. This application is not available for download.
  - `vtunedemo_fork` - This is a native application that is used as a demoThe application may not be available for download.
- **Tools**: VTune Profiler 2023 (or newer) - Hotspots analysis with Hardware Event-Based Sampling.
- **Container Orchestration System**: Kubernetes
- **Operating system**: Ubuntu* 22.04 based on Linux* kernel version 5.15 or newer
- **CPU**: Intel® microarchitecture code named Skylake or newer architecture

Configure a Kubernetes Pod

Prerequisite: Install a Kubernetes pod. Follow instructions at http://kubernetes.io.

Once you have a Kubernetes pod ready,

1. Modify the YAML configuration file for the pod. This example uses `pod-test` as the pod name and `test.yaml` for the configuration file.

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-test
  labels:
    app: pod-test
spec:
  containers:
  - name: pod-test-1
```

2. Enable a shared path between the host machine and the pod.

```
spec:
       volumes:
  - name: shared-path
    hostPath:
      path: /tmp/shared_path
      type: Directory
      containers:
   - name: pod-test-1
     volumeMounts:
     - name: shared-path
      mountPath: /tmp/test_application
```

   where:

   - `/tmp/shared_path` is a directory located on the host side
   - `/tmp/test_application` is a path located inside the pod

3. Configure the security context for the container. Enable privileged mode when profiling running pods. Type:

```
spec:
  containers:
    securityContext:
      privileged: true
```

   In order to profile the system with workloads running in pods, you must give access to the host PID namespace:

```
spec:
  hostPID: true
```

4. Apply the Kubernetes pod configuration file:

```
host> kubectl apply -f test.yaml
```

Run Hardware Event-Based Hotspots Analysis with VTune Profiler and Target in the Same Pod

In this procedure, let us run VTune Profiler and the workload in the same Kubernetes pod. We will then analyze the collected results on the host machine.

Prerequisite: Install VTune Profilersampling drivers for Linux targets or enable driverless collection.

1. Enable a shared path with VTune Profiler and the results directory between the node and the pod.

```
spec:
       volumes:
  - name: vtune-path
    hostPath:
      path: /opt/intel/oneapi/vtune
```

```
      type: Directory
        - name: vtune-results-path
    hostPath:
      path: /opt/vtune_results
      type: Directory
      containers:
  - name: pod-test-1
    volumeMounts:
    - name: vtune-path
     mountPath: /vtune
    - name: vtune-results-path
     mountPath: /tmp/vtune_results
```

where:

- `/opt/intel/oneapi/vtune` is the path to the installation directory (on the host) for VTune Profiler
- `/vtune` is a VTune Profiler path located inside the pod
- `/opt/vtune_results` is a writable location for VTune Profiler results on the host
- `/tmp/vtune_results` is a path to VTune Profiler results in the pod

2. Run Hotspots analysis in **Launch Application** mode. Use `analyze_mod` path in the results directory.

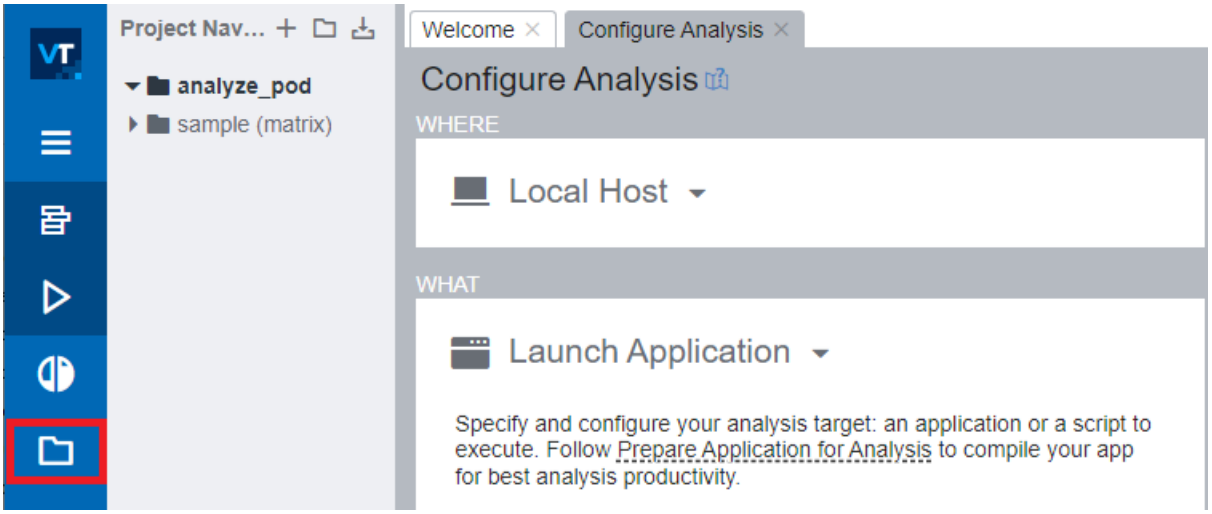
```
pod> cd /vtune/latest
pod> source vtune-vars.sh
pod> vtune -collect hotspots -knob sampling-mode=hw -knob enable-stack-collection=true --app-
working-dir=/var/local/jdk-19.0.2/bin -result-dir=/tmp/vtune_results/analyze_pod/r@@@{at} --
duration 30 -- /var/local/jdk-19.0.2/bin/java -cp /tmp/test_application/java_tests/
MatrixMultip_32bit/ MatrixMultiplication
```

> **NOTE** You can run User-mode and Hardware Event-based Hotspots analysis in both **Launch Application** and **Attach to Process** modes within the pod.

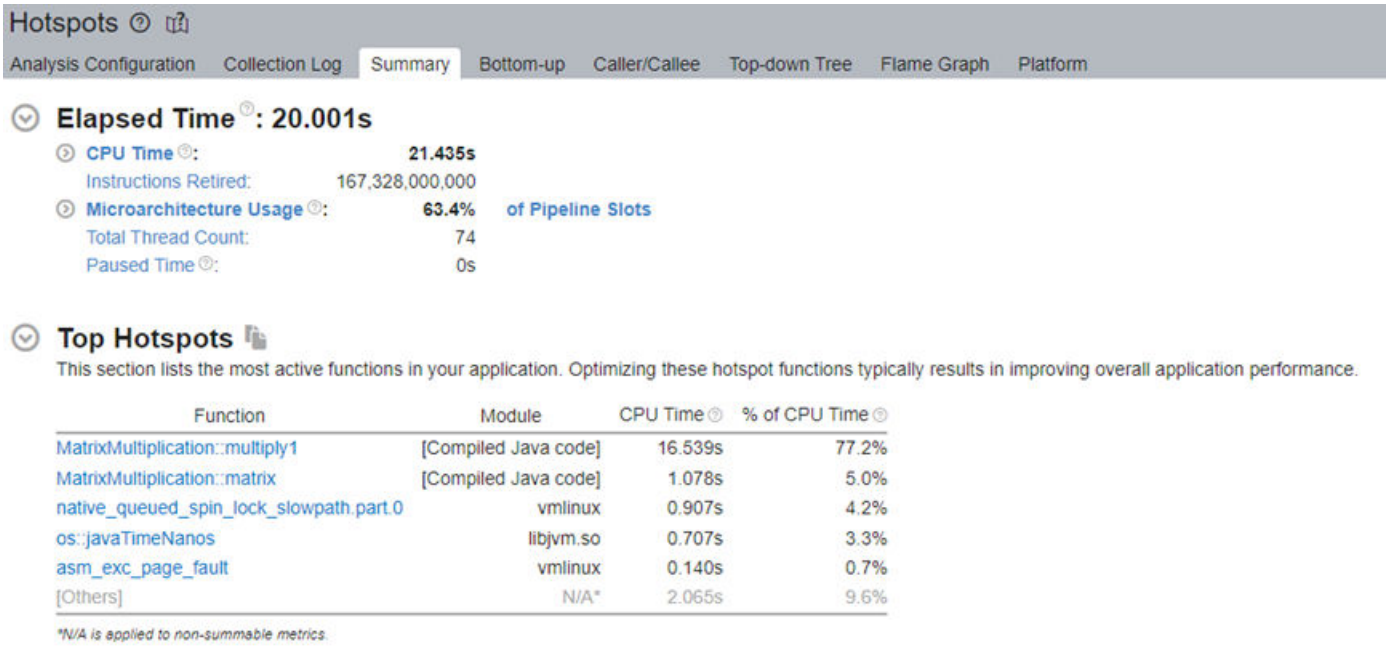
3. When the data collection completes, open the GUI on the VTune Profiler host machine.

```
host> vtune-gui
```

4. Create a project for the collected results. Let us call it `analyze_pod`.
5. Open the collected results. Click on the icon highlighted here:



6. Review the results in the **Summary** window of the Hotspots Analysis.



Run Hotspots Analysis in User-Mode Sampling on a Pod Target

Use VTune Profiler on the host machine to run a Hotspots analysis on a target in a Kubernetes pod.

1. Start VTune Profiler Server on the host machine. Type:

```
host> cd /opt/intel/oneapi/vtune/latest
host> source vtune-vars.sh
host> vtune-backend --allow-remote-access --web-port=50777 --enable-server-profiling &
```

   where:

   - `--allow-remote-access` enables remote access
   - `--web-port=50777` is the HTTP/HTTPS port for web UI and data APIs
   - `--enable-server-profiling` enables the user to select the hosting server as the profiling target
   - `&` runs the command in the background

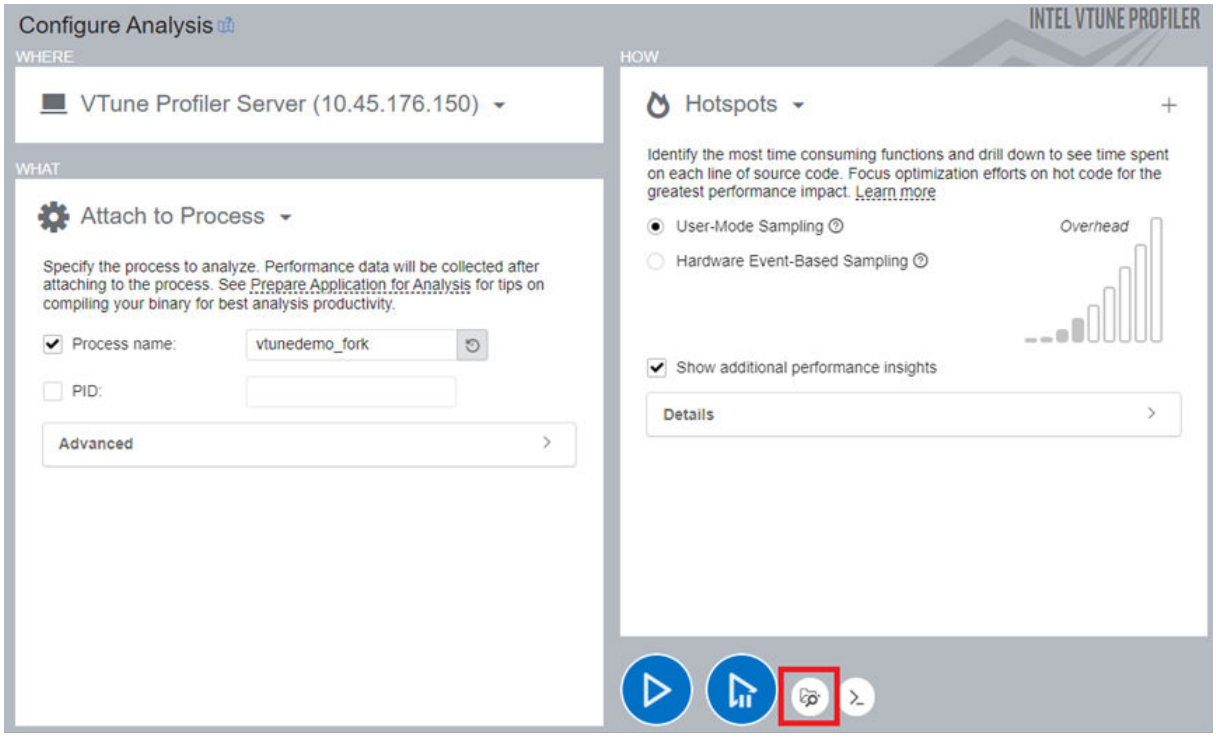
   The `vtune-backend` command returns a URL which you can open outside the container. For example:

```
Serving GUI at https://10.45.176.150:50777/?one-time-token=0ee4ec13b6c33fe416b49fcb273d43ac
```

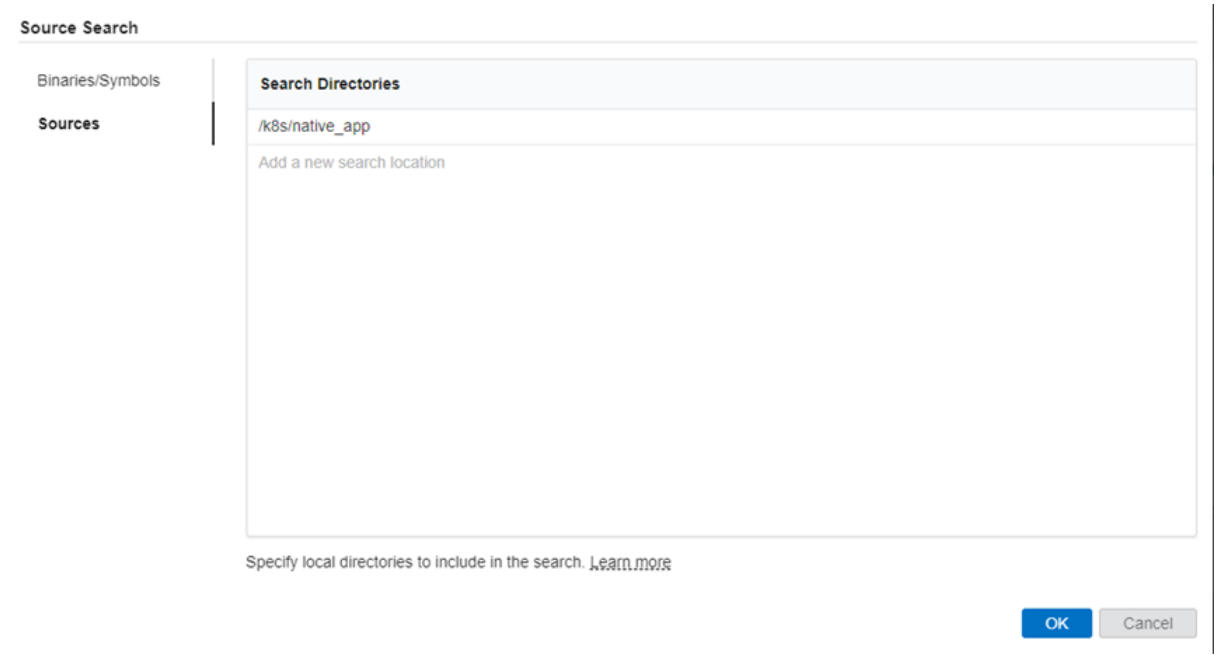
2. Run the native application in the pod.

```
pod> /tmp/test_application/native_app
pod> ./vtunedemo_fork -nonstop -nt 80
```

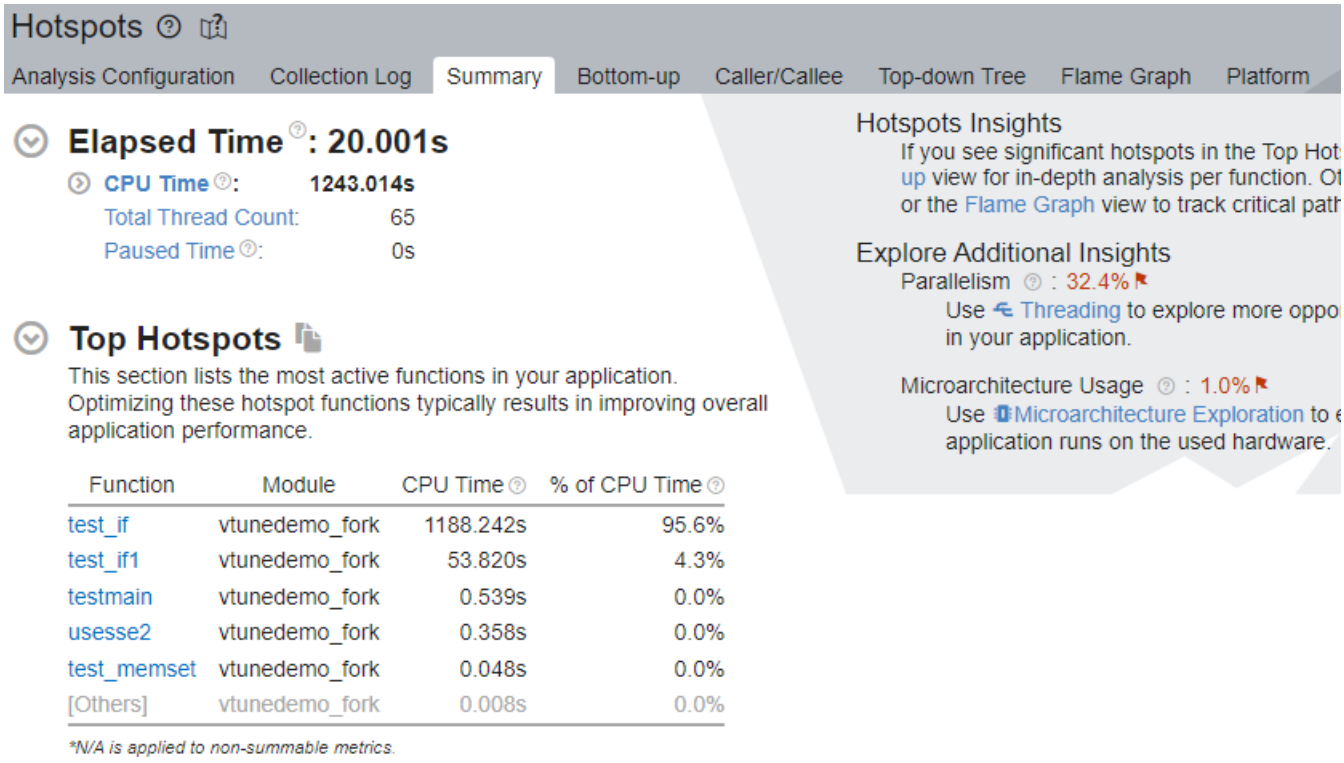
3. Run `vtune-backend` and open the URL you receive.
4. Create a project, for example `kubernetes_pod`.
5. To analyze the application running within the pod, run the Hotspots analysis in User-mode or Hardware Event-based Sampling mode. However, you must first configure the analysis to attach to the process. Specify the binaries and symbols of the application for Function level and Source level analysis of collected data:



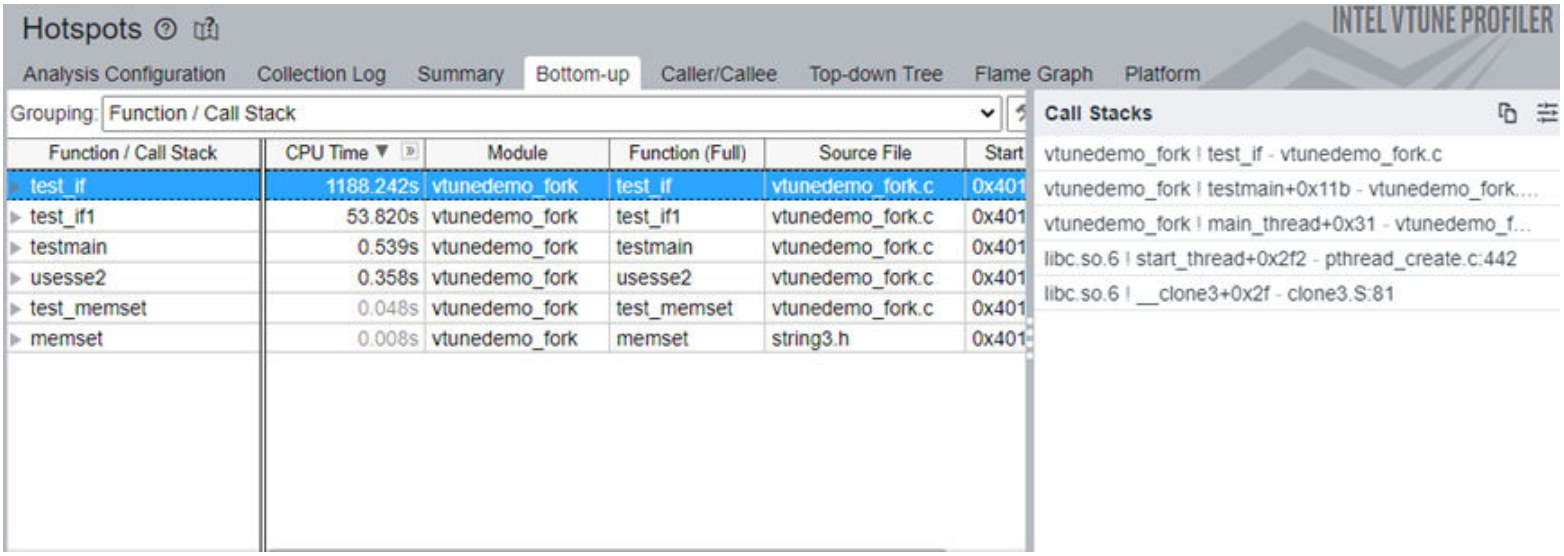
NOTE The file locations must be from the host.



Once the analysis completes, VTune Profiler displays results in the **Summary** window.



6. In the **Top Hotspots** section, we see that the `test_if` function of the target application consumed the most CPU time. Click on this function and switch to the **Bottom-up** window. See the stack flow for this hotspot.



Profiling Considerations

- You can only profile native C/C++ applications.
- You cannot profile applications that are instrumented with Intel® Instrumentation and Tracing Technology (ITT) APIs/JIT APIs and are running inside the container.

Run Profile System Analysis for Pods with Multiple Containers

Let us now run VTune Profiler on the host machine to profile a system with a Kubernetes pod that contains multiple containers.

Prerequisites:

- Install VTune Profilersampling drivers for Linux targets or enable driverless collection.
- To profile multiple containers inside a pod, each container must contain `privileged:true` in the `securityContext` section.

1. Start VTune Profiler Server on the host machine.

```
host> cd /opt/intel/oneapi/vtune/latest
host> source vtune-vars.sh
host> vtune-backend --allow-remote-access --web-port=50777 --enable-server-profiling &
```

   where:

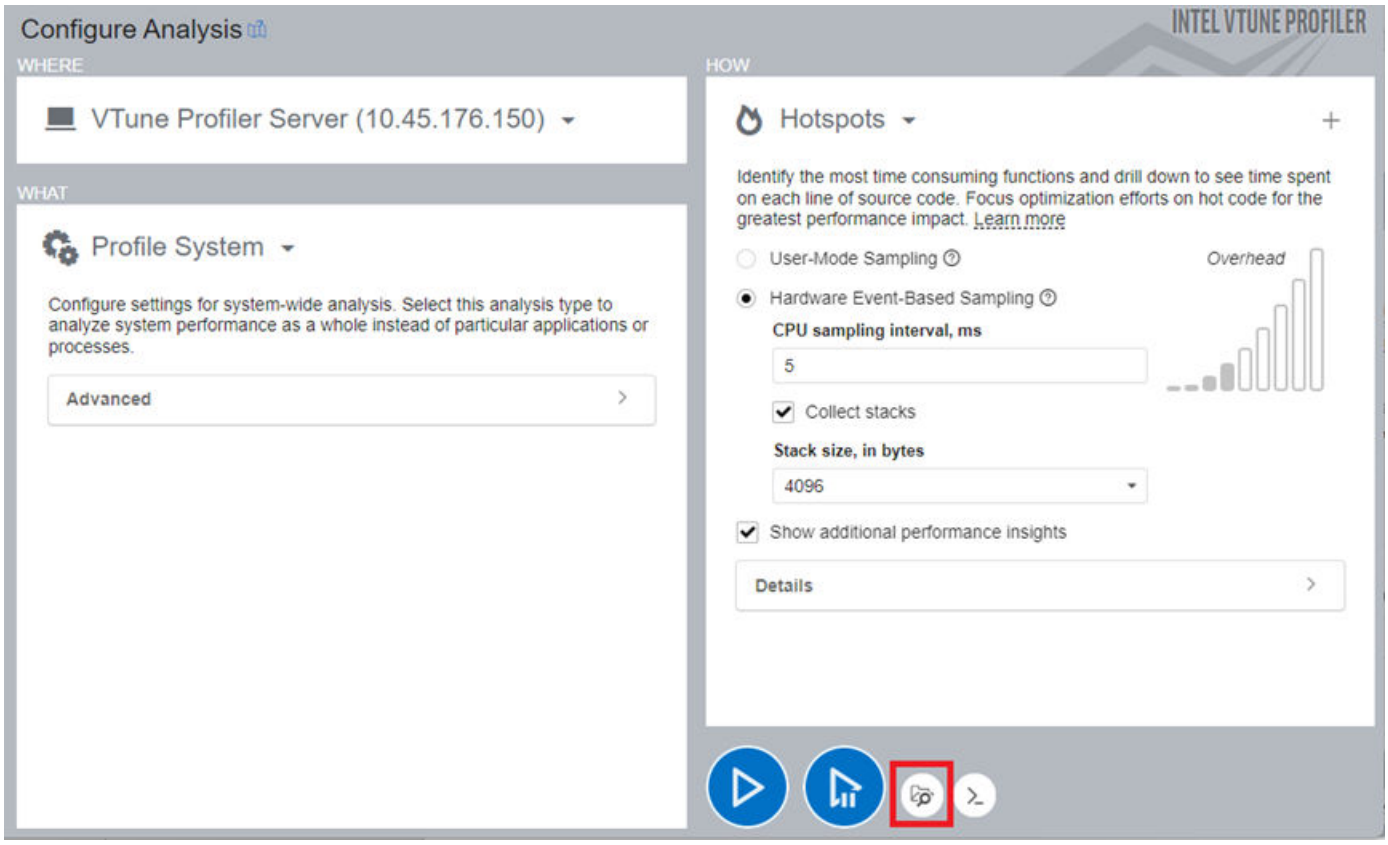
   - `--allow-remote-access` enables remote access
   - `--web-port=50777` sets the HTTP/HTTPS port for web UI and data APIs
   - `--enable-server-profiling` enables the user to select the hosting server as the profiling target
   - `&` runs the command in the background

2. Run `vtune-backend`. This command returns a URL which you can open outside the container. For example:

```
Serving GUI at https://10.45.176.150:50777/?one-time-token=0ee4ec13b6c33fe416b49fcb273d43ac
```

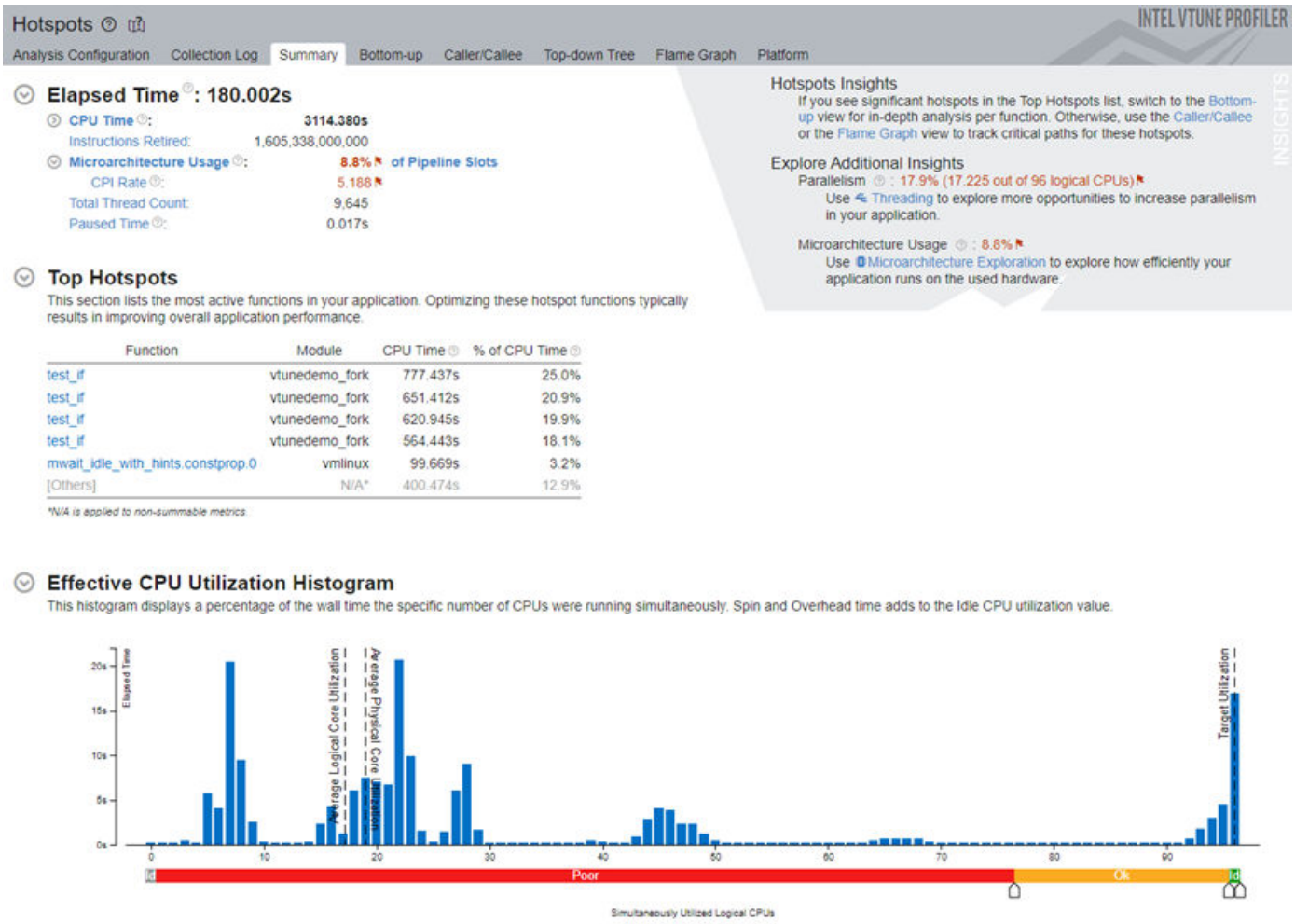
   Open the URL you receive.

3. On the host machine, start the **Profile System** analysis. Specify the binaries and symbols of the application for Function and Source level analysis of collected data.

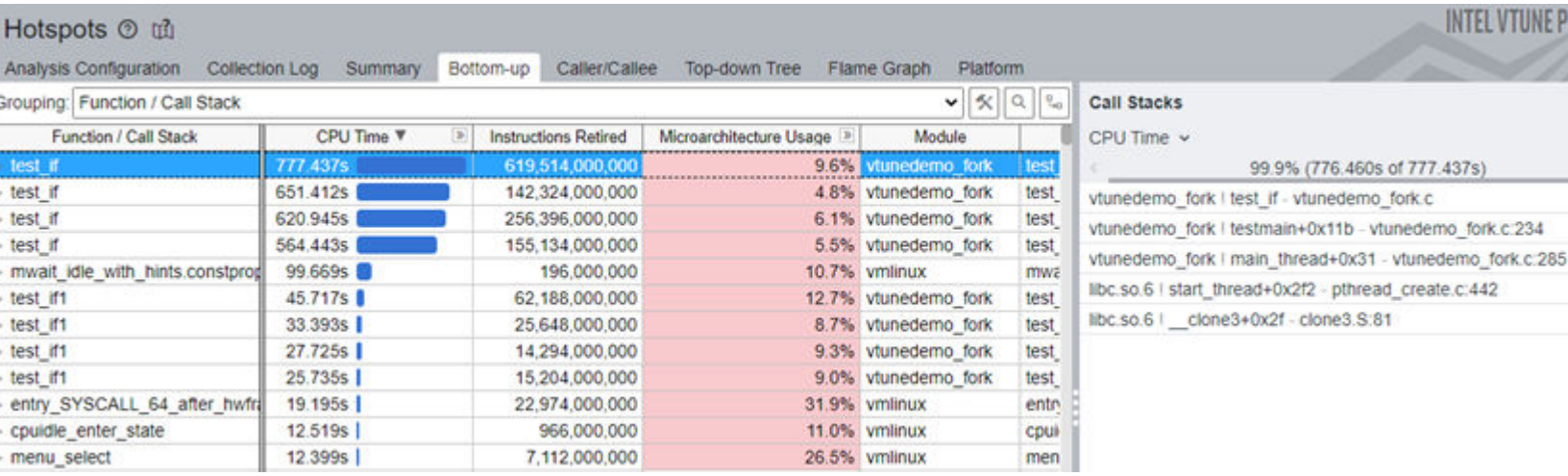


NOTE The file locations must be from the host.

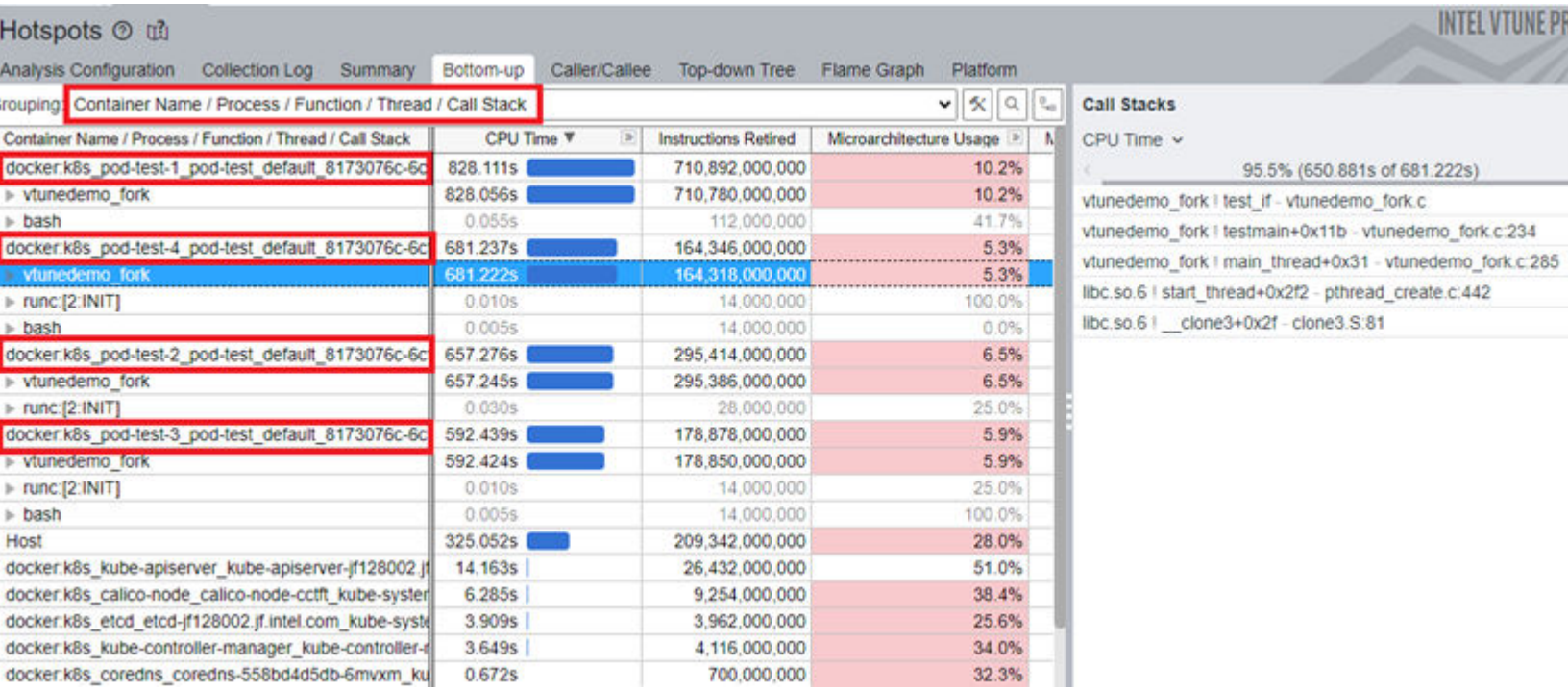
4. Inside the containers, run native applications.
5. Once the analysis completes, see results in the **Summary** tab.



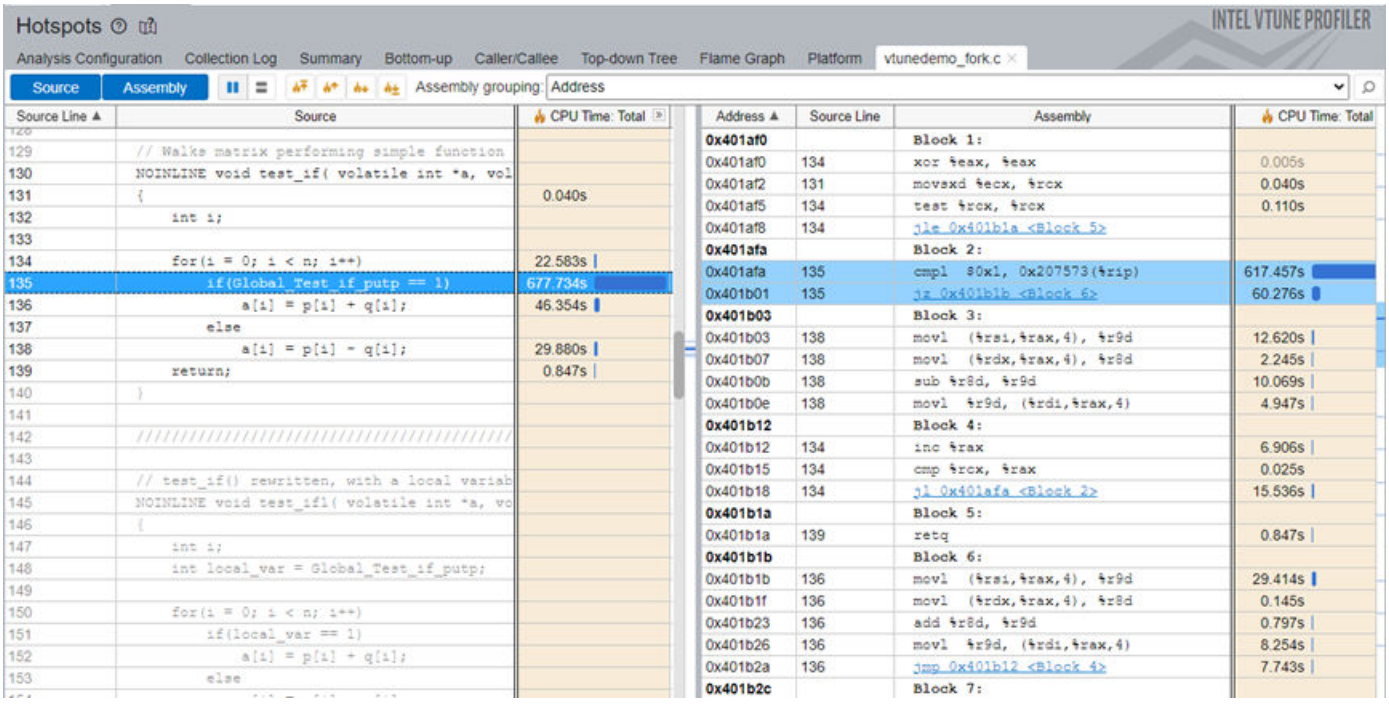
6. In the **Top Hotspots** section, click on the `test_if` function and switch to the **Bottom-up** window. See the stack flow for this hotspot.



7. To see performance data for the containers of the individual pod, select the **Container Name/ Process/Function/Thread/Call Stack** grouping from the pull down menu. Identify containers by the `docker:k8s` prefix.



8. Double click on the `test_if` function to do a source level analysis for this function.



Profiling Considerations:

- You can only profile native C/C++ applications.
- You cannot profile applications instrumented with ITT/JIT API.

See Also

Profiling Docker* Containers

Analyzing Hot Code Paths Using Flame Graphs (NEW)

Follow this recipe to understand how you can use Flame Graphs to detect hot spots and hot code paths in Java workloads.

Content Experts: Dmitry Kolosov, Elena Nuzhnova

A flame graph is a visual representation of the stacks and stack frames in your application. The graph plots all of the functions in your application on the X-axis and displays the stack depth on the Y-axis. Functions are stacked in order of ancestry, with parent functions directly below child functions. The width of a function displayed in the graph is an indication of the amount of time it engaged the CPU. Therefore, the hottest functions in your application occupy the widest portions on the flame graph.

You can use flame graphs when you run the hotspots analysis with stacks on any of these workloads:

- C++
- FORTRAN
- Java
- .NET
- Python

This recipe uses a Java application as an example. Typically, a poor selection of parameters (either sub-optimal or incorrect) for the Java Virtual Machine (JVM) can result in slow application performance. The slowdown is not always obvious to analyze or explain. When you visualize the application stacks in a flame graph, you may find it easier to identify hot paths for the application and its mixed stacks (Java and built-in).

> **NOTE** If you are interested in C++ optimization, you may want to see the Improving Hotspot Observability in a C++ Application Using Flame Graphs recipe. It demonstrates how the Flame Graph can help in a scenario where the bottleneck is obscured by an unclear hot path and long template function names.

- INGREDIENTS
- DIRECTIONS:

  1. Create a Baseline
  2. Run Hotspots Analysis
  3. Analyze Hotspots Information
  4. Identify Hot Code Paths in the Flame Graph
  5. Change JVM Options

Ingredients

Here are the hardware and software tools we use in this recipe:

- **Application**: SPECjbb2015® Benchmark. This benchmark is relevant to anyone who is interested in Java server performance including:
  - JVM vendors
  - Hardware developers
  - Java application developers
  - Researchers and members of the academic community
- OpenJDK11. This application is the open source reference implementation of the Java SE Platform (version 11) as specified by JSR384 in the Java Community Process.
- **Performance Analysis Tools**: Hotspots Analysis in Intel® VTune™ Profiler (version 2021.7 or newer)
- **Operating System**: Ubuntu* 18.04.1 LTS
- **CPU**: Intel® Xeon® Gold 6252 processor architecture codenamed Cascade Lake

Create a Baseline

1. For the purpose of this recipe, let us first shorten the runtime of SPECjbb2015. Change these properties in the `config/specjbb2015.props` file:

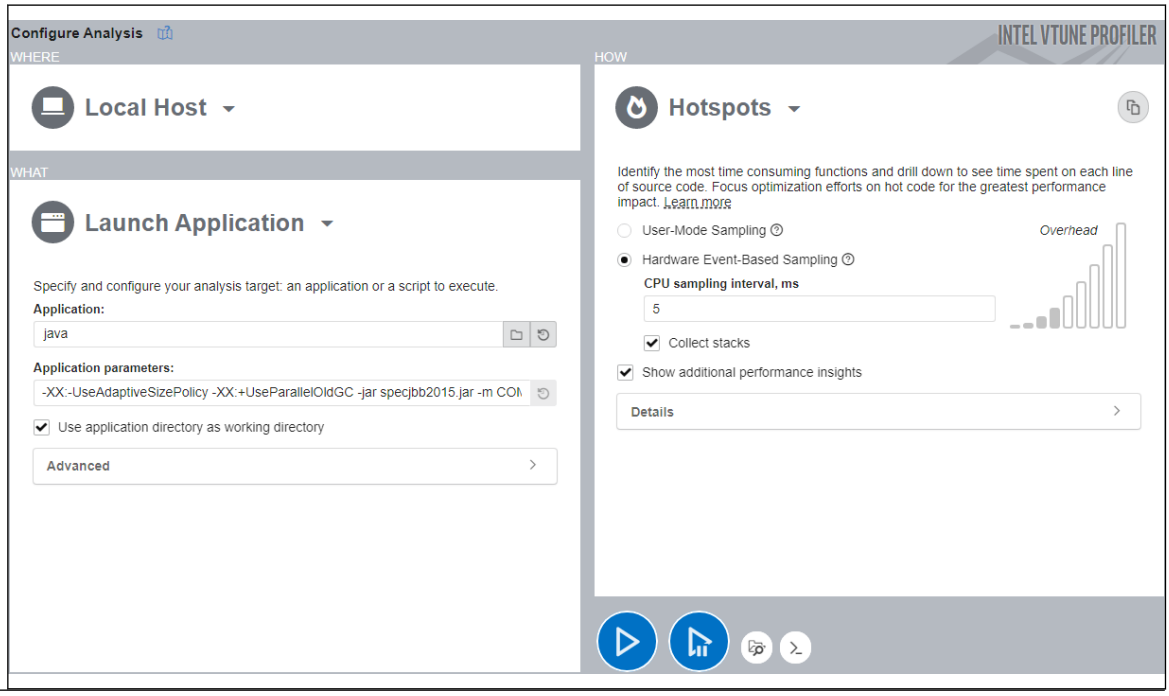
```
specjbb.input.number_customers=1
specjbb.input.number_products=1
```

2. In accordance with popular optimization practices and guidance, start optimizing the application with `-XX:+UseParallelOldGC` and `-XX:-UseAdaptiveSizePolicy` JVM options.
3. Make sure to tune these parameters for optimal performance of your Java application:
   - **Garbage Collection (GC) Algorithm** - When you enable the `UseParallelOldGC` option, you can collect old and young generation collections in parallel. Garbage collection can then work more efficiently because you have reduced the overall full GC pause. If throughput is your goal, specify `-XX:+UseParallelOldGC`.
   - **Heap Tuning** - By default, JVMs adapt their heap based on runtime heuristics. To achieve pause, throughput, and footprint goals, the GC can resize heap generations based on GC statistics. In some cases, to increase throughput, you may want to disable this option and set the heap size manually. Use the heap as a performance baseline for further optimizations.

```
java -XX:-UseAdaptiveSizePolicy -XX:+UseParallelOldGC -jar specjbb2015.jar -m COMPOSITE
```

Run Hotspots Analysis

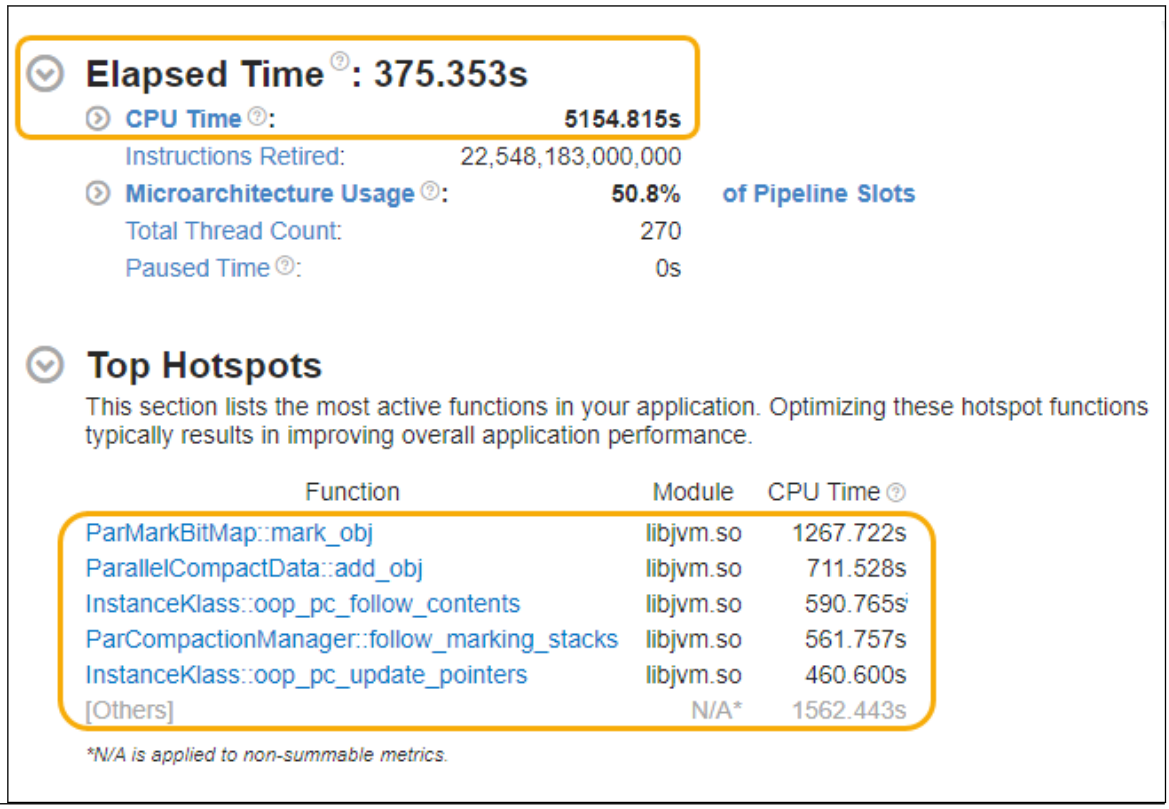
1. Run VTune Profiler (version 2021.7 or newer).
2. In the Welcome screen, click **Configure Analysis**.
3. In the **WHERE** pane, select Local Host.
4. In the **WHAT** pane, enter these values:
   - **Application:** `java`
   - **Application parameters:** `-XX:-UseAdaptiveSizePolicy -XX:+UseParallelOldGC -jar specjbb2015.jar -m COMPOSITE`
5. In the **HOW** pane, open the Analysis Tree and select **Hotspots** analysis in the **Algorithm** group.
6. Select **Hardware Event-Based Sampling** mode and check the **Collect stacks** option.
7. Click the **Start** button to run the analysis.



VTune Profiler profiles the Java application and collects data. Once this process completes, VTune Profiler finalizes the collected results and resolves symbol information.

Analyze Hot Spots Information

Start your analysis in the **Summary** window, where you can see high level statistics on the execution of your application. Focus on the **Elapsed Time** and **Top Hotspots** sections.



In this example, we see that the elapsed time for `SPECjbb2015` was around 375 seconds.

The top five hotspots in the summary are in JVM functions. No Java/Application functions appear in this list.

Look at the Bottom-up window next to continue searching for hotspots.

Grouping: Function / Call Stack

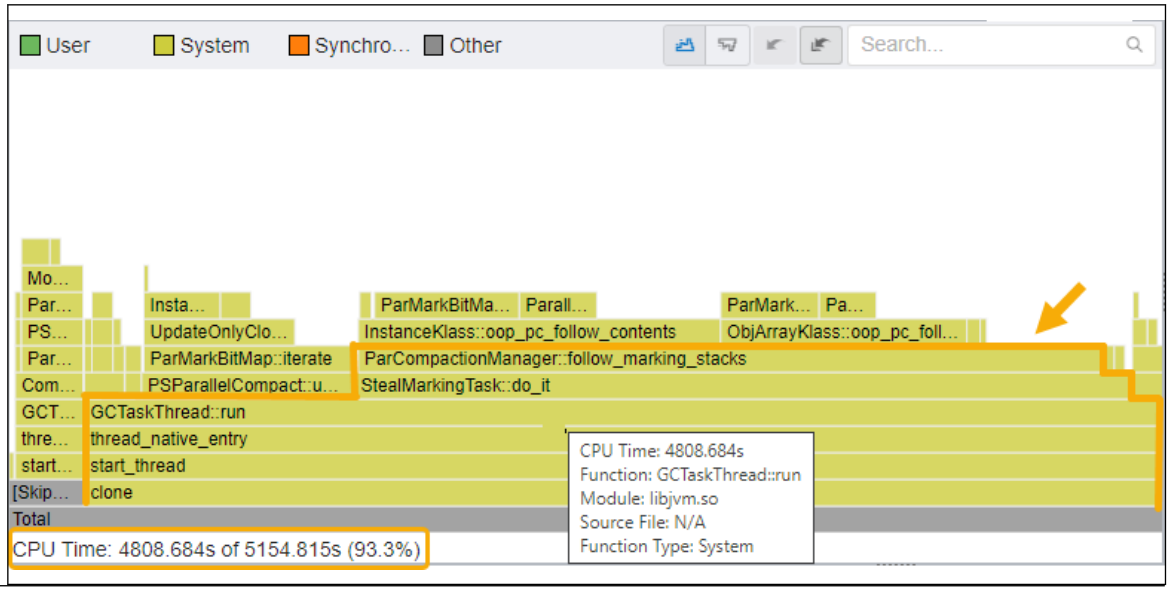
Function / Call Stack	CPU Time ▼	Instructions Retired	Microarchitecture Usage	Module
ParMarkBitMap::mark_obj	1267.722s	5,335,092,000,000	53.0%	libjvm.so
ParallelCompactData::add_obj	711.528s	1,119,510,000,000	24.8%	libjvm.so
InstanceKlass::oop_pc_follow_contents	590.765s	2,271,843,000,000	49.7%	libjvm.so
ParCompactionManager::follow_marking_stacks	561.757s	2,965,431,000,000	56.6%	libjvm.so
InstanceKlass::oop_pc_update_pointers	460.600s	3,285,429,000,000	77.5%	libjvm.so
ObjArrayKlass::oop_pc_follow_contents	423.723s	737,730,000,000	20.3%	libjvm.so
ParMarkBitMap::iterate	265.763s	1,899,429,000,000	76.9%	libjvm.so
ObjArrayKlass::oop_pc_update_pointers	202.826s	990,549,000,000	45.7%	libjvm.so
UpdateOnlyClosure::do_addr	194.656s	1,332,639,000,000	71.4%	libjvm.so
MoveAndUpdateClosure::do_addr	115.340s	864,906,000,000	78.6%	libjvm.so
ParMarkBitMap::iterate	109.206s	787,143,000,000	75.8%	libjvm.so
Klass::class_loader	55.841s	355,530,000,000	78.2%	libjvm.so
Klass::start_of_vtable	44.324s	339,969,000,000	81.3%	libjvm.so
SpinPause	35.974s	44,184,000,000	29.3%	libjvm.so
StealMarkingTask::do_it	29.128s	26,649,000,000	11.3%	libjvm.so
ParallelTaskTerminator::offer_termination	14.394s	13,146,000,000	14.9%	libjvm.so
func@0x18f120	8.129s	0	1.7%	libc-2.27.so
AccessInternal::PostRuntimeDispatch<CardTableBarr	7.087s	34,083,000,000	60.9%	libjvm.so
SymbolTable::unlink	4.190s	2,562,000,000	3.3%	libjvm.so
org::spec::jbb::sm::ReceiptBuilder$GenerateReceipts'	4.029s	1,176,000,000	3.8%	[Compiled Java code]
ParMarkBitMap::live_words_in_range_helper	3.629s	25,431,000,000	60.0%	libjvm.so
PSPromotionManager::copy_to_survivor_space<(boo	2.325s	2,352,000,000	9.7%	libjvm.so
hyperv_callback_vector	1.243s	777,000,000	29.9%	vmlinux
InstanceKlass::oop_pc_follow_contents.constprop.58	1.193s	3,465,000,000	28.5%	libjvm.so
nmethod::fix_oop_relocations	1.163s	11,193,000,000	81.1%	libjvm.so
InstanceMirrorKlass::oop_pc_follow_contents	1.032s	1,323,000,000	9.5%	libjvm.so

Although the Bottom-up window displays more hotspots in the JVM, we need a deeper analysis to explain the slowdown of the Java application. This would require an expansion of bunches of parent functions for every hotspot in the table above.

Let us now look at the flame graph for this data, where we can observe all application stacks at once and possibly identify hot code paths.

Identify Hot Code Paths in the Flame Graph

Switch to the **Flame Graph** window.



A Flame Graph is a visual representation of the stacks and stack frames in your application. Every box in the graph represents a stack frame with the complete function name. The horizontal axis shows the stack profile population, sorted alphabetically. The vertical axis shows the stack depth, starting from zero at the bottom.

The flame graph does not display data over time. The width of each box in the graph indicates the percentage of the function CPU time to total CPU time. The total function time includes processing times of the function and all of its children (*callees*).

The **Flame Graph** window contains a Call Stacks view, which displays the hottest stack when selected in the flame graph. You can also observe other stacks by selecting a function or drill down to its source code.

Types of Functions in a Flame Graph

The flame graph uses a color scheme to display these types of functions:

Function Type	Description
User	A function from the application module of the user.
System	A function from the System or Kernel module
Synchronization	A synchronization function from the Threading Library (like OpenMP Barrier)
Overhead	An overhead function from the Threading library (like OpenMP Fork or OpenMP Dispatcher)

Follow these techniques as you examine the information displayed in the flame graph.

- Start optimizing from the bottommost function and work your way up. Focus on hot functions that are wide on the flame graph.
- In this example, the flame graph displays stacks and frames that are only from the JVM. Therefore almost all of the CPU time was spent in the JVM.
- Consequently, the CPU time spent on the application was significantly low. Application stacks or frames are not even visible in the flame graph.
- The hottest code path is `clone --> start_thread --> thread_native_entry --> GCTaskThread::run --> StealMarkingTask::do_it -->` and so on.
- Pay attention to the `GCTaskThread::run` function/frame, which runs Java Garbage Collector tasks.
- When you hover over `GCTaskThread::run` function/frame, you can see in the details at the bottom that 93.3% of CPU Time was spent on the function and its callees.

Therefore, a lot of CPU Time was spent in the Java Garbage Collector.

Change JVM Options

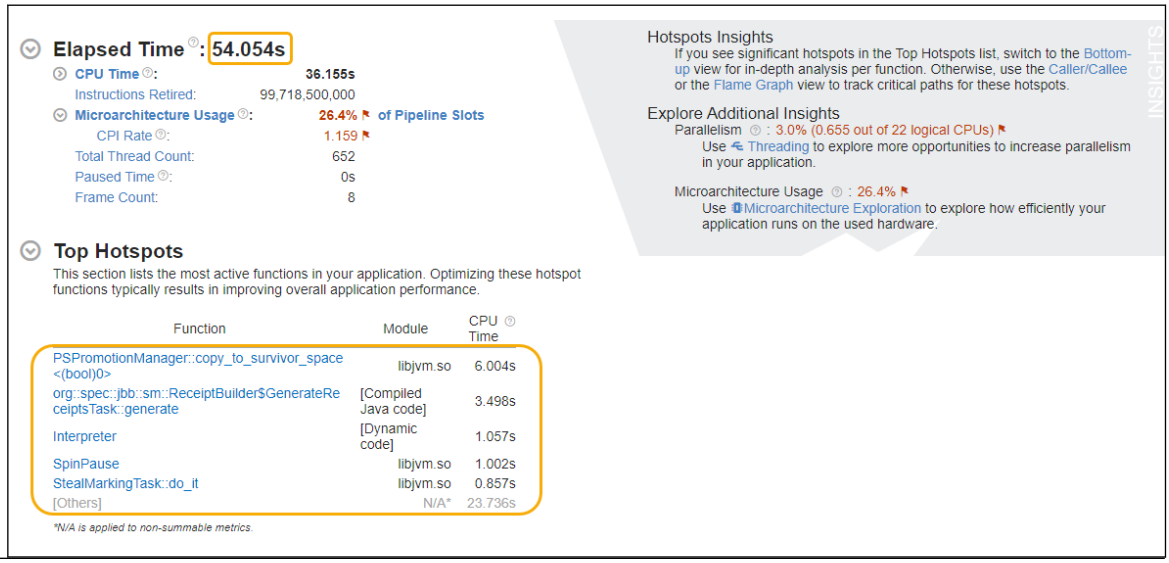
The `-XX:-UseAdaptiveSizePolicy` JVM option may not allow the application to adapt to the size of the JVM heap. The default values used for the run may also be insufficient. Let us now change the size of the JVM heap to decrease the executing time of the Garbage Collector (GC).

The `-Xms` and `-Xmx` options are used to set the operating range of the JVM where it can resize the heap. If the two values are the same, the heap size remains constant. It is good practice to refer to the JVM logs before you set values for these options.

Let us change the `-Xms` and `-Xmx` JVM options for the application to 2GB and 4GB respectively. We will then collect a new profile:

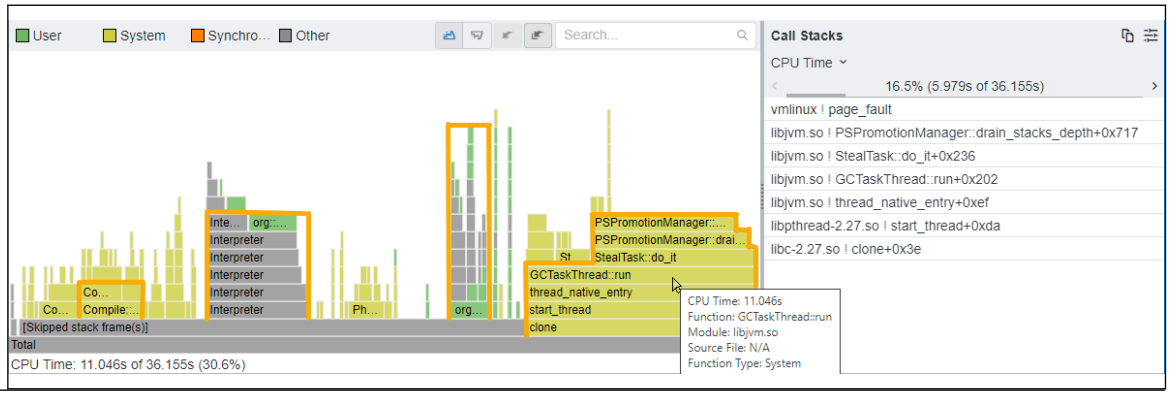
1. Click the **Configure Analysis** button in the Welcome screen of VTune Profiler.
2. In the **WHERE** pane, select **Local Host**.
3. In the **WHAT** pane, set **Application** to **java**.
4. Change application parameters. Use `-Xms2g -Xmx4g -XX:-UseAdaptiveSizePolicy -XX:+UseParallelOldGC -jar specjbb2015.jar -m COMPOSITE`.
5. Click **Start** to run the analysis.

Once the data collection completes, check the **Elapsed Time** and **Top Hotspots** in the **Summary** window.



- We can observe a 6x reduction in **Elapsed Time** from ~375 s to ~54s.
- The **Top Hotspots** section also displays a new list of functions (including the `GenerateReceipts` task) with shorter CPU times.

Switch to the **Flame Graph** window to identify new hot code paths.



The flame graph shows a hot code path that includes the JVM `GCTaskThread`.

However, this hot code path uses only 30.6% of **CPU Time** compared to 93.3% on the previous run.

Next Steps

- You may want to focus on new hot code paths that proceed in this direction:
  - `JVM Compile::Compile -> ...`
  - `JVM Interpreter -> org::spec::jbb::sm::ReceiptBuilder`
  - `org::spec::jbb::sm::ReceiptBuilder -> ...`
- Review your JVM options to identify more opportunities for optimization.
- If you want to optimize the JVM next, a good starting point is to focus on the **Microarchitecture Usage** metric and follow recommendations in the **Insights** section of the **Summary** window:
  - Apply **Threading** to increase parallelism n your application.
  - Run the **Microarchitecture Exploration** analysis to examine the efficiency of application runs on the hardware used.

See Also

Window: Flame Graph

Hotspots View

An explanation of Flame Graphs

Improving Hotspot Observability in a C++ Application Using Flame Graphs

See how the Flame Graph feature of Intel® VTune™ Profiler can help in a scenario where the true hotspot is obscured behind template functions and long function names.

Content experts: Dmitry Kolosov, Roman Khatko

The Flame Graph is a feature of the Hotspots analysis of Intel® VTune Profiler that visualizes application execution paths in an intuitive visual form.

In some cases, finding a hotspot using the **Top Hotspots** section of the **Summary** tab or the **Bottom-up** window is difficult. Some examples are:

- Applications with flat hotspots, or multiple hotspots with CPU Time spread evenly across many hotspots.
- Applications with nonactionable hotspots:
  - Library function calls (STL, Boost, MKL, …) are on the top of stacks, and the caller is not easily findable.
  - Template functions with long and complicated names.

In such cases, the Flame Graph helps identify true hotspots and hot code paths by providing greater observability for all application stacks. This increase in observability can save time and effort when analyzing applications with complex stacks.

This feature supports workloads written in all languages that are supported by the Hotspots analysis, including—but not limited to:

- C++
- FORTRAN
- Java
- .NET
- Python

This recipe demonstrates how the Flame Graph can help identify hot paths in your code more easily, especially in cases where the real hotspot is obscured, or there are multiple hotspots.

> **NOTE** If you are interested in Java optimization, you may want to see the Analyzing Hot Code Paths Using Flame Graphs recipe. It demonstrates how the Flame Graph can help detect a sub-optimal JVM configuration that results in loss of performance.

- Ingredients
- Directions
  - Configure Project and Build Sample
  - Establish Performance Baseline
  - Analyze Hotspots Result with Summary and Bottom-up Windows
  - Identify Hot Code Paths on the Flame Graph
  - Analyze Second Implementation
  - Analyze Third Implementation
  - Analyze Fourth Implementation
  - Summary

Ingredients

This section lists the hardware and software tools used for the performance analysis scenario:

- **Application:** C++ STL/Boost* based sample.

Boost is a set of libraries for C++ that supports tasks and structures such as linear algebra, pseudorandom number generation, multithreading, and more.

In this example, Boost 1.77.0 for Windows* is used (download sources or binaries).

- **Performance analysis tool:** Intel® VTune™ Profiler 2021.9—Hotspots analysis
- **Operating system:** Microsoft Windows Server* 2016
- **IDE:** Microsoft Visual Studio* 2017
- **CPU:** Intel® Xeon® E5-2695 v3 (formerly codenamed Haswell)

Configure Project and Build Sample

As a first step, we will build and analyze the sample application below to establish a performance baseline.

The sample code below splits random-generated long text (100 million words) by words in a service thread. It contains four implementations that handle the task differently:

Implementation	Uses Function	Based On
1	`splitByWordsBoost`	`boost::split`
2	`splitByWordsStdString`	`std::string::find` and `std::string::substr`
3	`splitByWordsStdStringView`	`std::string_view::find` and `std::string_view::substr` and `std::vector` of `string_view`
4	`splitByWordsStdStringView`	pre-allocated output vector

Each implementation is selectable by a command-line parameter. Over the course of this recipe, we will switch between different implementations to analyze how the performance changes.

```
#include <iostream>
#include <vector>
#include <chrono>
#include <thread>
#include <string_view>

#include <boost/algorithm/string.hpp>

using namespace std::chrono;

void generateRandomText(std::string& text, const size_t words, const size_t symbolsInWord)
{
    for (size_t i = 0; i < words; i++)
    {
        for (size_t j = 0; j < symbolsInWord; j++)
        {
            text += 'a' + i % 26;
        }
        text += ' ';
    }
}

void splitByWordsBoost(const std::string& text, std::vector<std::string>& splitWords)
{
    boost::split(splitWords, text, boost::is_any_of(" "));
}

void splitByWordsStdString(const std::string& text, std::vector<std::string>& splitWords)
{
    const char delimiter = ' ';
```

```
    size_t start, end = 0;
    while ((start = text.find_first_not_of(delimiter, end)) != std::string::npos) {
        end = text.find(delimiter, start);
        splitWords.push_back(text.substr(start, end - start));
    }
}

void splitByWordsStdStringView(const std::string& text, std::vector<std::string_view>&
splitWords)
{
    const char delimiter = ' ';
    size_t start, end = 0;
    std::string_view textView(text);
    while ((start = textView.find_first_not_of(delimiter, end)) != std::string::npos) {
        end = textView.find(delimiter, start);
        splitWords.emplace_back(textView.substr(start, end - start));
    }
}

int main(int argc, char** argv)
{
    int splitMode = 0;
    const char* msg = "splitByWordsBoost";
    if (argc > 1)
    {
        switch (*argv[1])
        {
        case '2': splitMode = 1; msg = "splitByWordsStdString"; break;
        case '3': splitMode = 2; msg = "splitByWordsStdStringView"; break;
        case '4': splitMode = 3; msg = "splitByWordsStdStringView(pre-allocated vector)"; break;
        }
    }

    const size_t numOfWords = 100000000, symbolsInWord = 10;
    std::string text;
    text.reserve(numOfWords * (symbolsInWord+1));

    std::cout << "Generating random text: ";
    auto start = high_resolution_clock::now();
    generateRandomText(text, numOfWords, symbolsInWord);
    auto stop = high_resolution_clock::now();
    std::cout << duration_cast<duration<float>>(stop - start).count() << " seconds" << std::endl;

    std::vector<std::string> splitWords;
    std::vector<std::string_view> splitWordsView;
    if (splitMode == 3) splitWordsView.reserve(numOfWords);

    std::cout << msg << " function: ";
    std::thread thread;
    start = high_resolution_clock::now();
    switch (splitMode)
    {
    case 0: thread = std::thread(splitByWordsBoost, std::ref(text), std::ref(splitWords)); break;
    case 1: thread = std::thread(splitByWordsStdString, std::ref(text), std::ref(splitWords));
break;
    case 2:
    case 3: thread = std::thread(splitByWordsStdStringView, std::ref(text),
std::ref(splitWordsView)); break;
```

```
    }
    thread.join();
    stop = high_resolution_clock::now();

    std::cout << duration_cast<duration<float>>(stop - start).count() << " seconds" << std::endl;
    auto splitWordsSize = splitMode >= 2 ? splitWordsView.size() : splitWords.size();
    std::cout << "Split words: " << splitWordsSize << std::endl;
}
```

The code can be copied over to a newly created C++ project, such as **Console App** or **Empty Project** in Visual Studio 2017, or any newer Visual Studio version with C++17 support.

Since the sample depends on the Boost library, additional configuration steps are required.

> **NOTE** Make sure that the **Configuration** is set to **Release** and the **Platform** is set to **x64** when editing project properties or compiling the application.

Follow these steps to set up the project in Visual Studio:

1. Right-click on the project node in the **Solution Explorer** and select **Properties**.
2. In the **Project Property Pages** window, make sure the **Configuration** drop-down is set to **Release** and **Platform** is set to **x64**.
3. In the **VC++ Directories** page, add the root Boost directory to **Include Directories**.
4. In the **VC++ Directories** page, add the Boost libraries directory to **Library Directories**.

   The default directory is:

   `<boost-root-directory>\libs`
5. In the **C/C++ > Language** page, make sure the **C++ Language Standard** option is set to **ISO C++17 Standard (/std:c++17)**.

   The configuration is now ready.

> **NOTE** The sample code can be built and tuned on Linux if the Boost library is installed and a compiler that supports C++17 is available. Use this command to build the sample on Linux:
>
> ```
> g++ -Wall -O2 -g -pthread -std=c++17 -l <path-to-boost-dir> ConsoleApplication1.cpp -o
> ConsoleApplication1
> ```

Build the project in the **Release x64** configuration. Once compiled, different implementations of the algorithm can be selected by using the <option> parameter:

```
>ConsoleApplication1.exe <option>
```

where `<option>` is: 1, 2, 3, or 4.

The application defaults to 1 if no arguments are given, so as a first step, run the application without parameters:

```
>ConsoleApplication1.exe
```

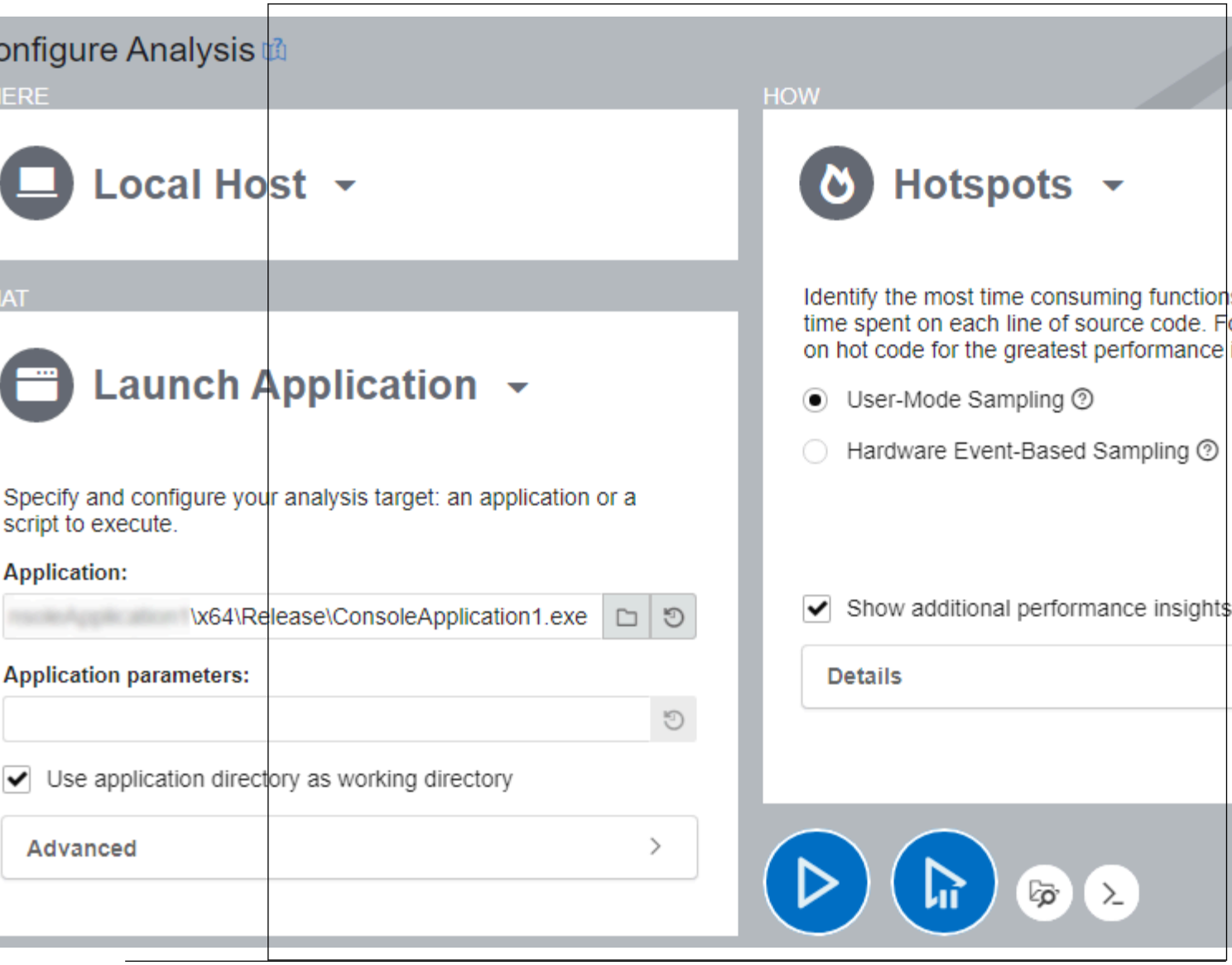
Establish Performance Baseline

For the next step, run the Hotspots analysis of VTune Profiler with the application using the first implementation—the one that is based on the `boost::split` function. The result of this analysis will become our performance baseline, against which we will compare future optimizations.

Run VTune Profiler and start your analysis:

1. Click the **New Project** button on the toolbar and specify a name for the new project, such as split_string.
2. Click **Create Project**.

   The **Configure Analysis** window opens.
3. On the **WHERE** pane, select the **Local Host** target system.
4. On the **WHAT** pane, select the **Launch Application** mode.
5. In the **Application** textbox, provide the path to the built application binary.
6. On the **HOW** pane, select the **Hotspots** analysis.
7. Click **Start** to run the analysis.



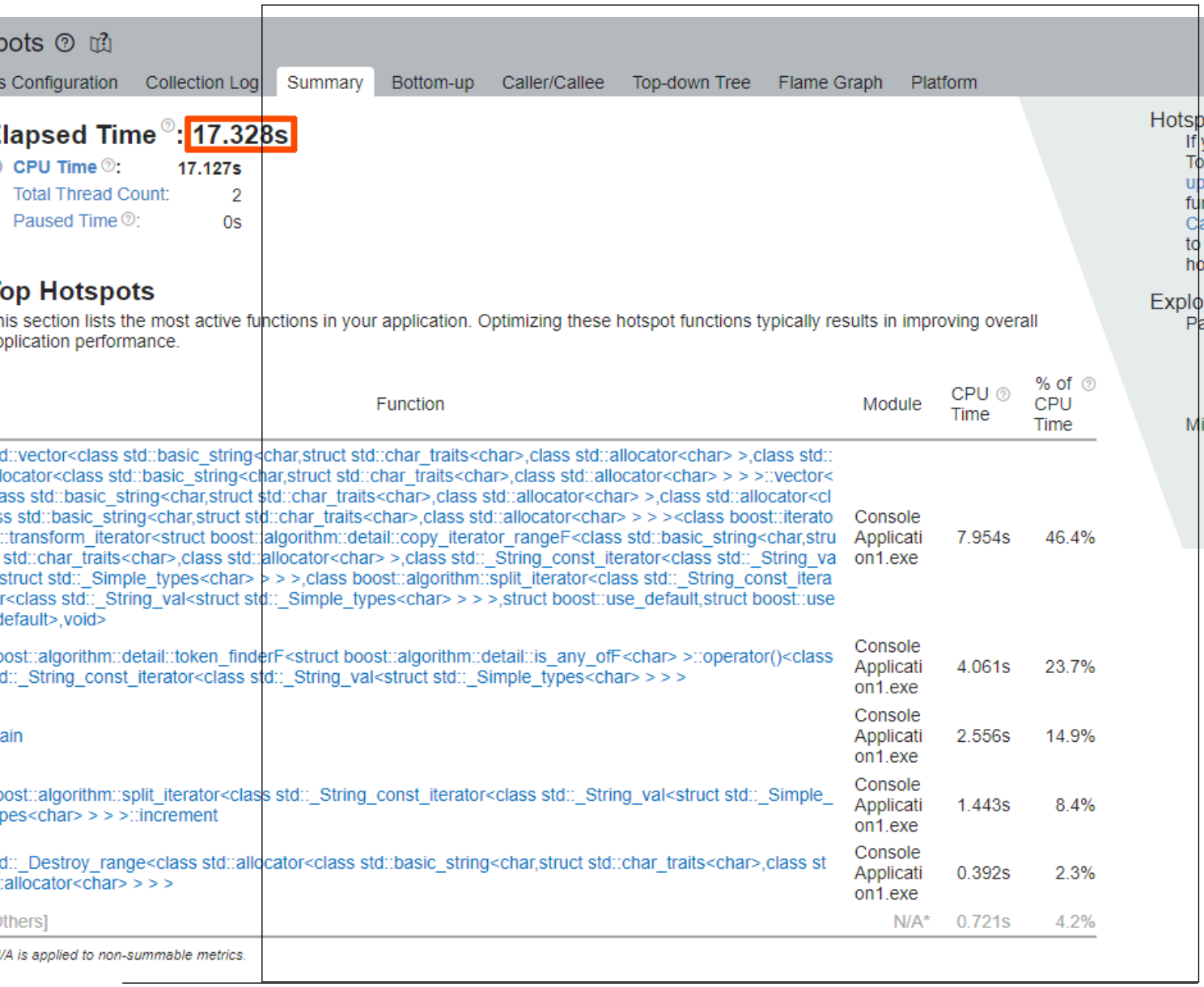
VTune Profiler launches the application and collects all necessary data before finalizing the result.

On the system specified in the Ingredients section, the total application working time is about 17 seconds.

Analyze Hotspots Result with Summary and Bottom-up Windows

Start your investigation with the **Summary** window that shows high-level information on application performance.

Focus on the **Elapsed Time** value and the **Top Hotspots** section that contains the list of hottest functions, in decreasing order.



pots

s Configuration | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Flame Graph | Platform

lapsed Time: 17.328s

CPU Time: 17.127s

Total Thread Count: 2

Paused Time: 0s

op Hotspots

his section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall pplication performance.

Function	Module	CPU Time	% of CPU Time
d::vector<class std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> >,class std::locator<class std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> > > >::vector< ass std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> >,class std::allocator<cl ss std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> > > ><class boost::iterato ::transform_iterator<struct boost::algorithm::detail::copy_iterator_rangeF<class std::basic_string<char,stru std::char_traits<char>,class std::allocator<char> >,class std::_String_const_iterator<class std::_String_va struct std::_Simple_types<char> > > >,class boost::algorithm::split_iterator<class std::_String_const_itera r<class std::_String_val<struct std::_Simple_types<char> > > >,struct boost::use_default,struct boost::use default>,void>	Console Application1.exe	7.954s	46.4%
oost::algorithm::detail::token_finderF<struct boost::algorithm::detail::is_any_ofF<char> >::operator()<class d::_String_const_iterator<class std::_String_val<struct std::_Simple_types<char> > > >	Console Application1.exe	4.061s	23.7%
ain	Console Application1.exe	2.556s	14.9%
oost::algorithm::split_iterator<class std::_String_const_iterator<class std::_String_val<struct std::_Simple_ pes<char> > > >::increment	Console Application1.exe	1.443s	8.4%
d::_Destroy_range<class std::allocator<class std::basic_string<char,struct std::char_traits<char>,class st :allocator<char> > > >	Console Application1.exe	0.392s	2.3%
thers]	N/A*	0.721s	4.2%

/A is applied to non-summable metrics.

Hotspots Insights

If you see signific Top Hotspots list, up view for in-de function. Otherwi Caller/Callee or t to track critical pa hotspots.

Explore Additional

Parallelism :

Use Threa opportunities parallelism i

Microarchitecture

Use Micro Exploration t efficiently yo the used har

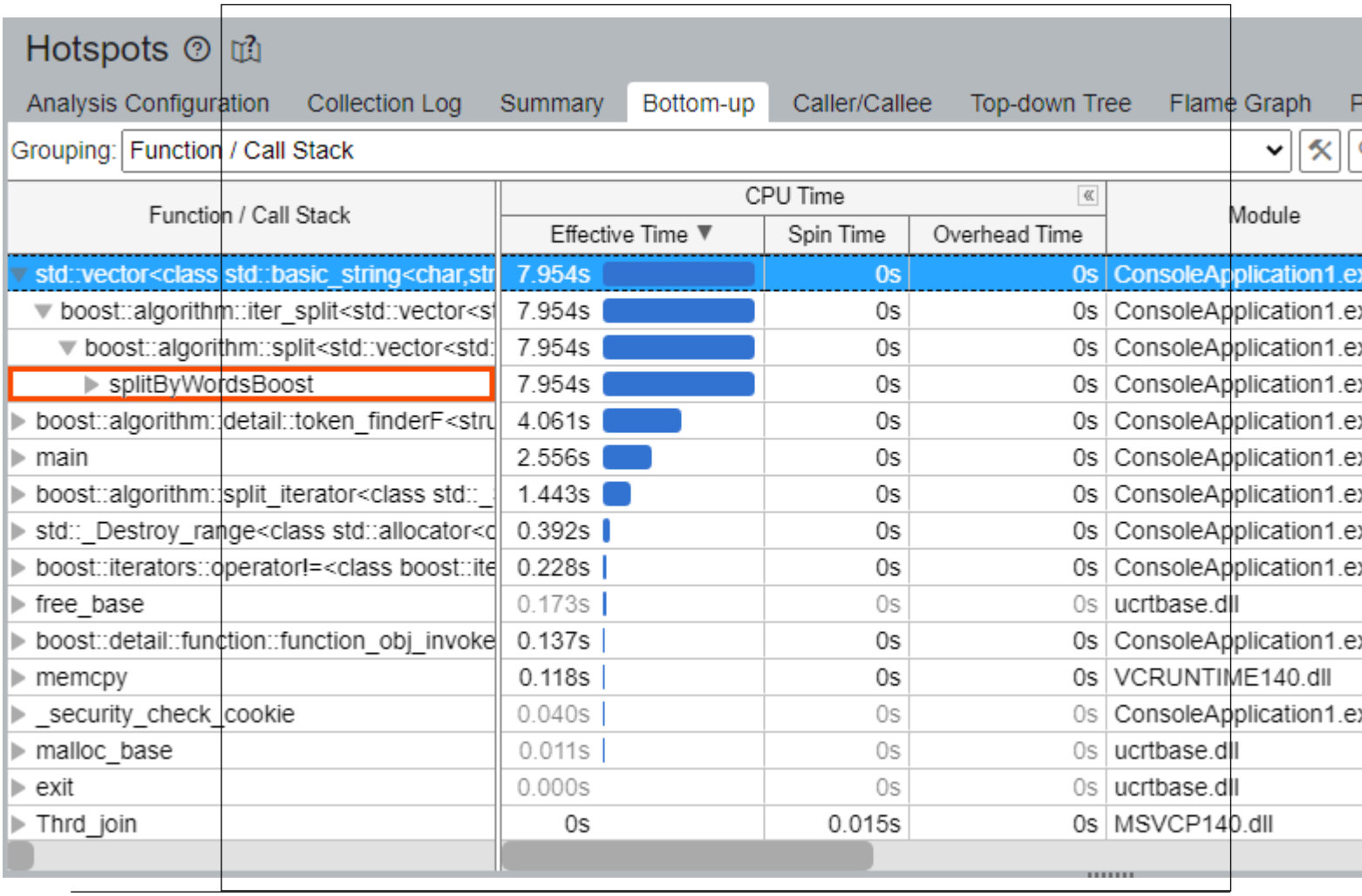
The **Elapsed Time** of the sample application is about 17 seconds.

The **Top Hotspots** are:

`std::vector` template	46.4% of CPU Time
`boost::algorithm::detail::token_finderF` template	23.7% of CPU Time
`main` function	14.9% of CPU Time
`boost::algorithm::split_iterator` template	8.4% of CPU Time

Note that the `splitByWordsBoost` or `boost::split` functions are not on the list. Since the **Top Hotspots** list only contains template functions and the `main` function, it does not make sense to start optimizations with the functions highlighted on the **Summary** window. The real hotspot is likely obscured by the template functions.

Try finding the hotspot in the **Bottom-up** window. Switch to the **Bottom-up** window to observe more hotspots related to `splitByWordsBoost` and `boost::split` functions that are the focus of our attention.



The **Bottom-up** window shows more functions and/or hotspots, but the `splitByWordsBoost` and `boost::split` functions are not at the top of the list. The target function and its relationship to the identified hotspots is not obvious.

If you expand the callers of the `std::vector` and `boost::algorithm::split`, the `splitByWordsBoost` function can only be found on the 3rd-4th levels and takes 7.954 seconds—46.4% of CPU Time—out of the total 17.328 seconds of **Elapsed Time**.

In such cases—when the hotspot is obscured by the template functions it calls—it may be more productive to use the **Flame Graph** to eliminate any unnecessary guesswork and effort related to figuring out the relations between hot functions.

Open the **Flame Graph** window to observe all application stacks, frames, and hot code paths.

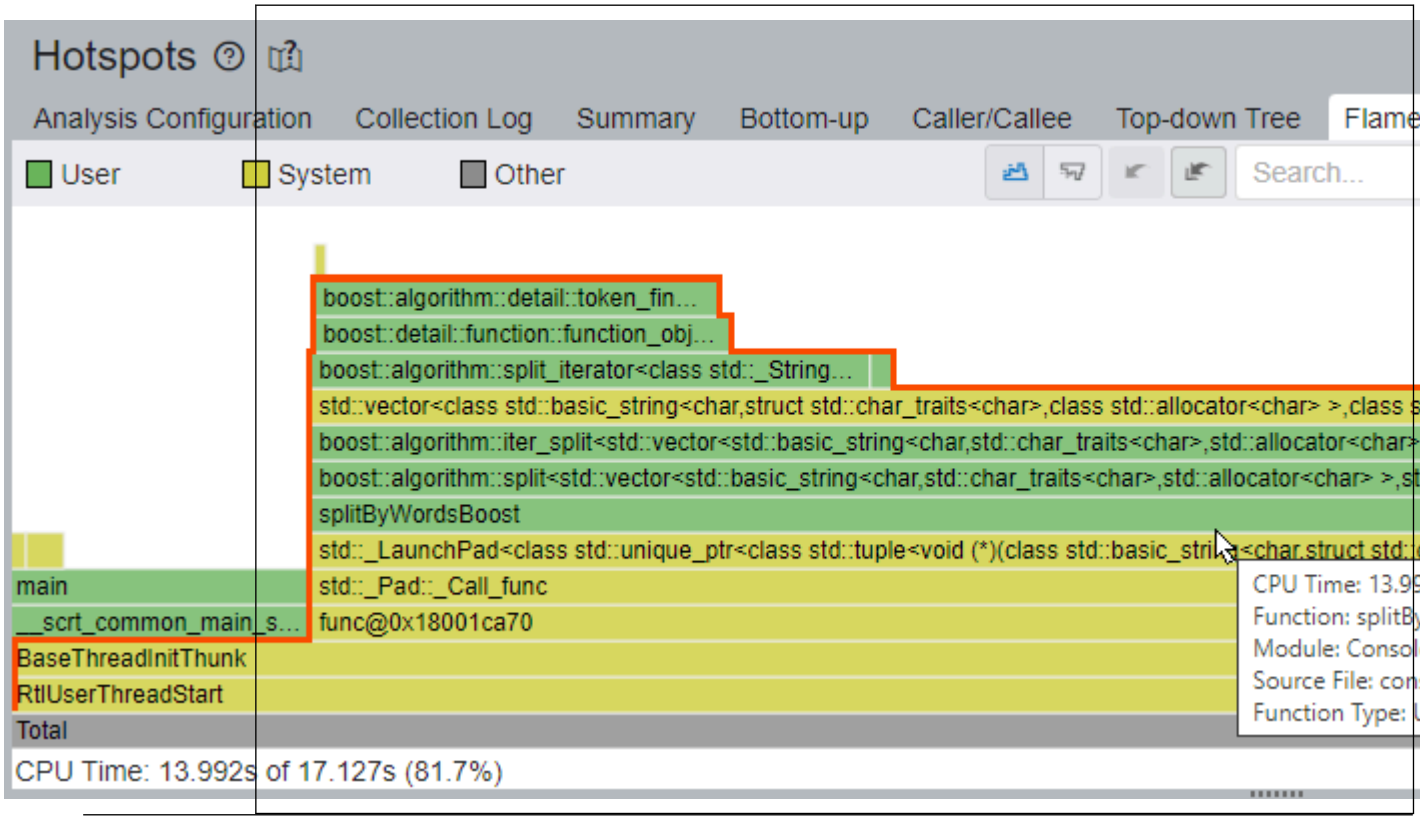
Identify Hot Code Paths on the Flame Graph

A Flame Graph is a visual representation of the stacks and stack frames in your application. Each box in the graph represents a stack frame with the complete function name. The horizontal axis shows the stack profile population, sorted alphabetically. The vertical axis is the stack depth, starting from zero at the bottom.

Note that the Flame Graph is not a timeline, so it does not display data over time. The width of each box in the graph is proportional to the amount of CPU Time taken by this function out of the total CPU Time. The total function time includes processing times of the function and all its children (callees).

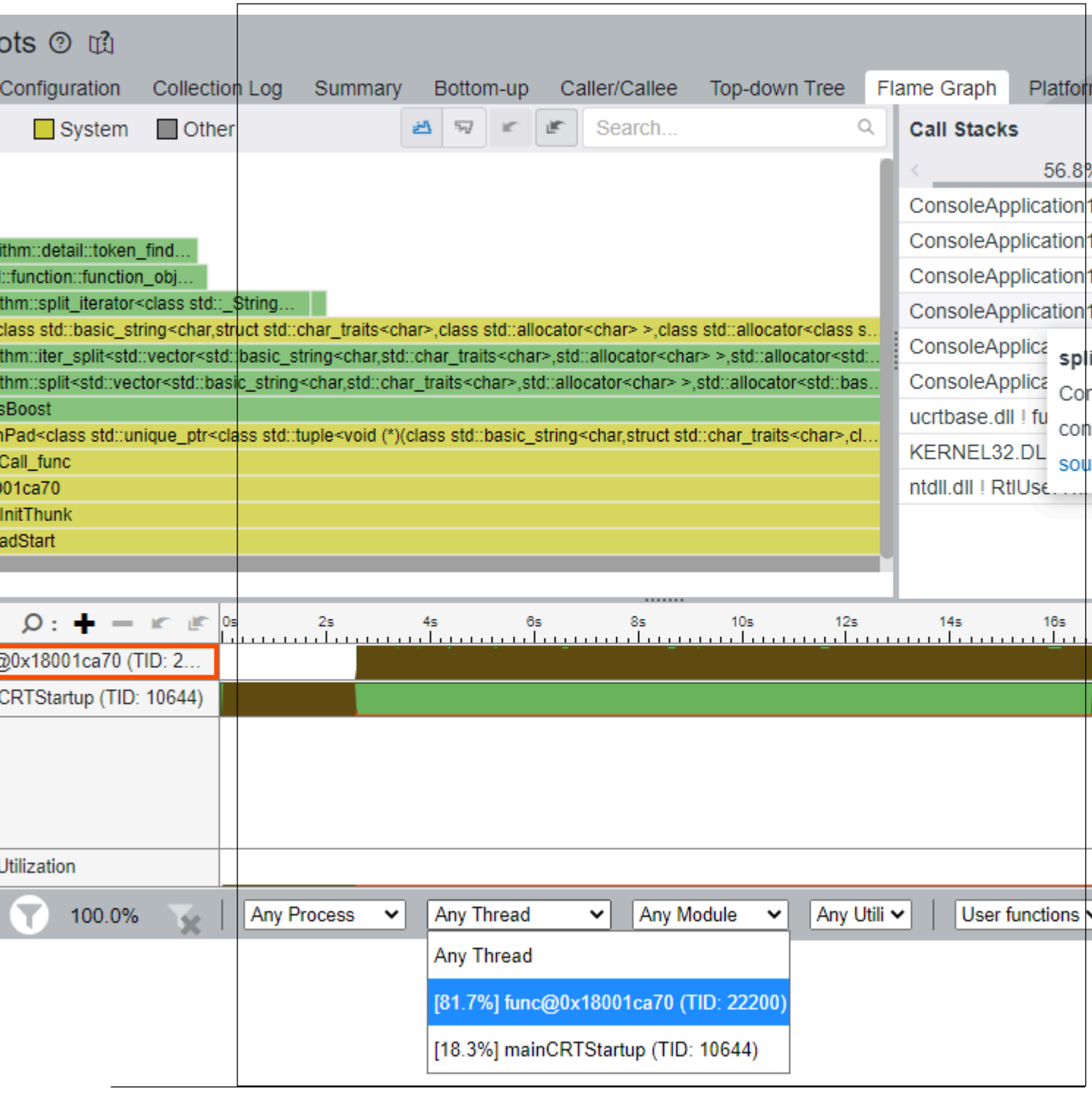
> **NOTE** For more information and background on Flame Graphs in general, see the Flame Graphs article by Flame Graph inventor Brendan Gregg.

Start optimizing from the bottom functions and move up. Pay attention to the hot (wide) functions first.

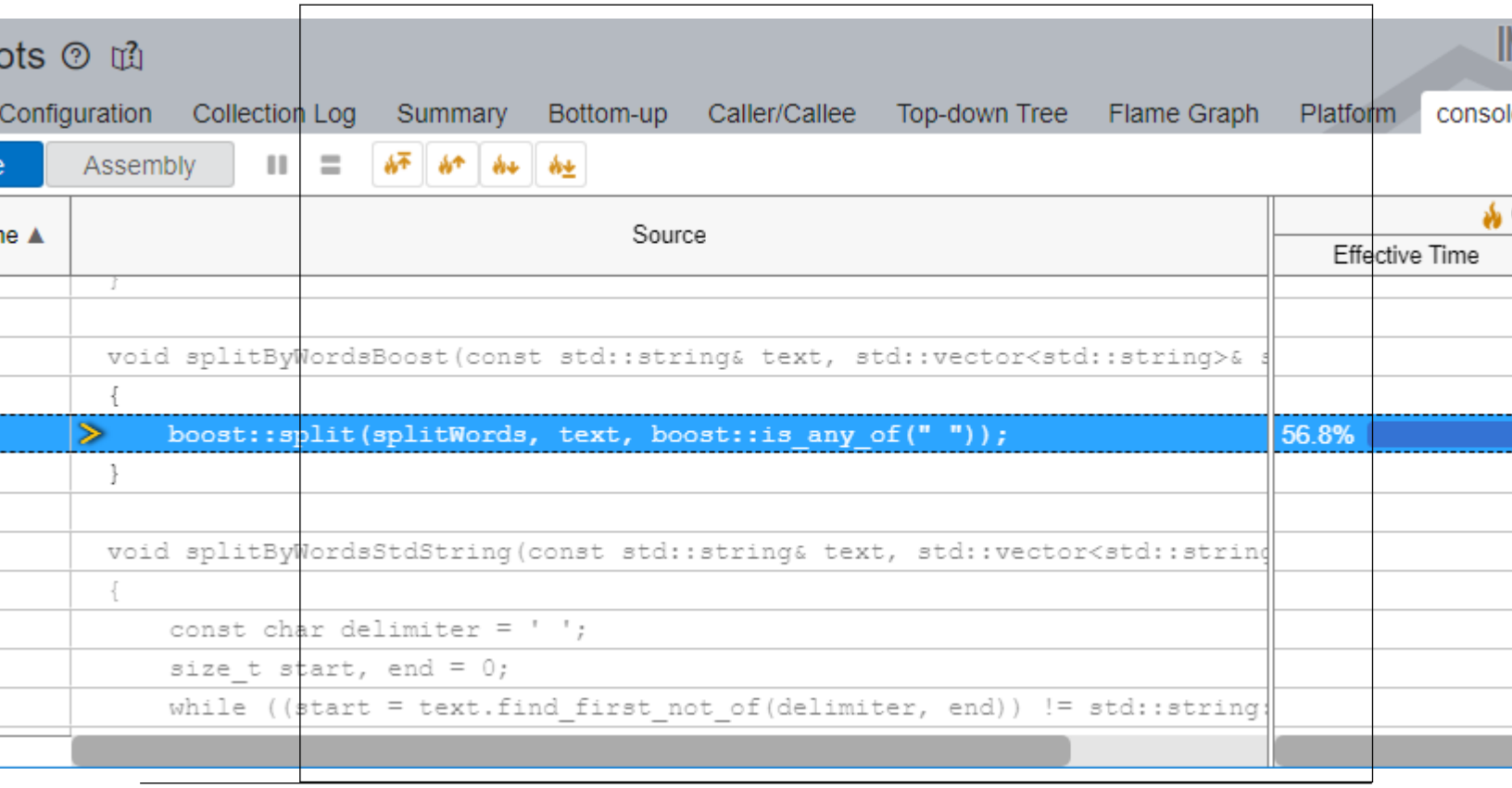


In case of this application, note that the Flame Graph provides perfect observability and presents a clear hot code path. Here, the path passes through the `splitByWordsBoost` and the `boost::algorigthm::split` functions, and arrives at `std::vector` and `boost` template frames at the top. The `splitByWordsBoost` function and its callees take 13.992 seconds (81.7%) of all CPU Time.

The Flame Graph works in conjunction with the **Timeline** pane and the **Filter** toolbar, which enables you to filter data by time region, process, thread, etc. The **Timeline** pane shows two application threads: main and service. The `splitByWordsBoost` function works in the service thread, so it makes sense to filter data by service thread via the **Filter By Thread** drop-down to make the Flame Graph view cleaner. In this case, the service thread is the thread `func@0x18001ca70 (TID:22200)`.



The **Call Stacks** pane on the right responds to selection/zoom on the Flame Graph and shows stacks with frames passing through a selected function. Hover over and click the `splitByWordsBoost` function in the **Call Stacks** pane to drill down to **Source/Assembly** view. The source view shows an obvious hotspot on the `boost::split` function.



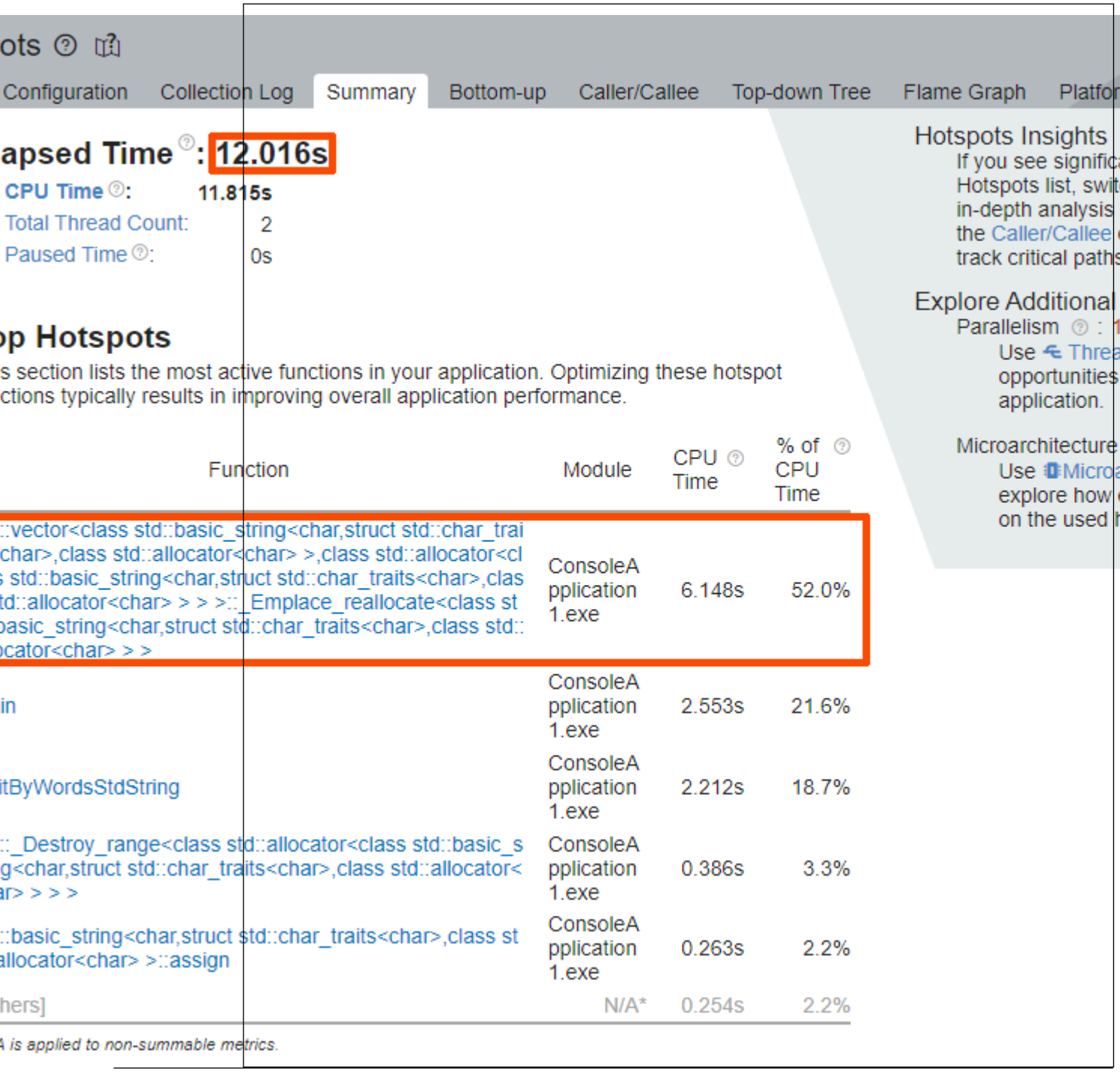
This implementation could use some optimization. However, we cannot optimize the library function `boost::algorithm::split`. Therefore, we will try our own implementation of the split function based on `std::string::find`.

Analyze Second Implementation

To analyze the code with the second implementation of the splitter function, open the **Configure Analysis** dialog and add **2** to the **Application Parameters** field. This enables the second implementation inside the application.

Click **Start** to run the analysis.

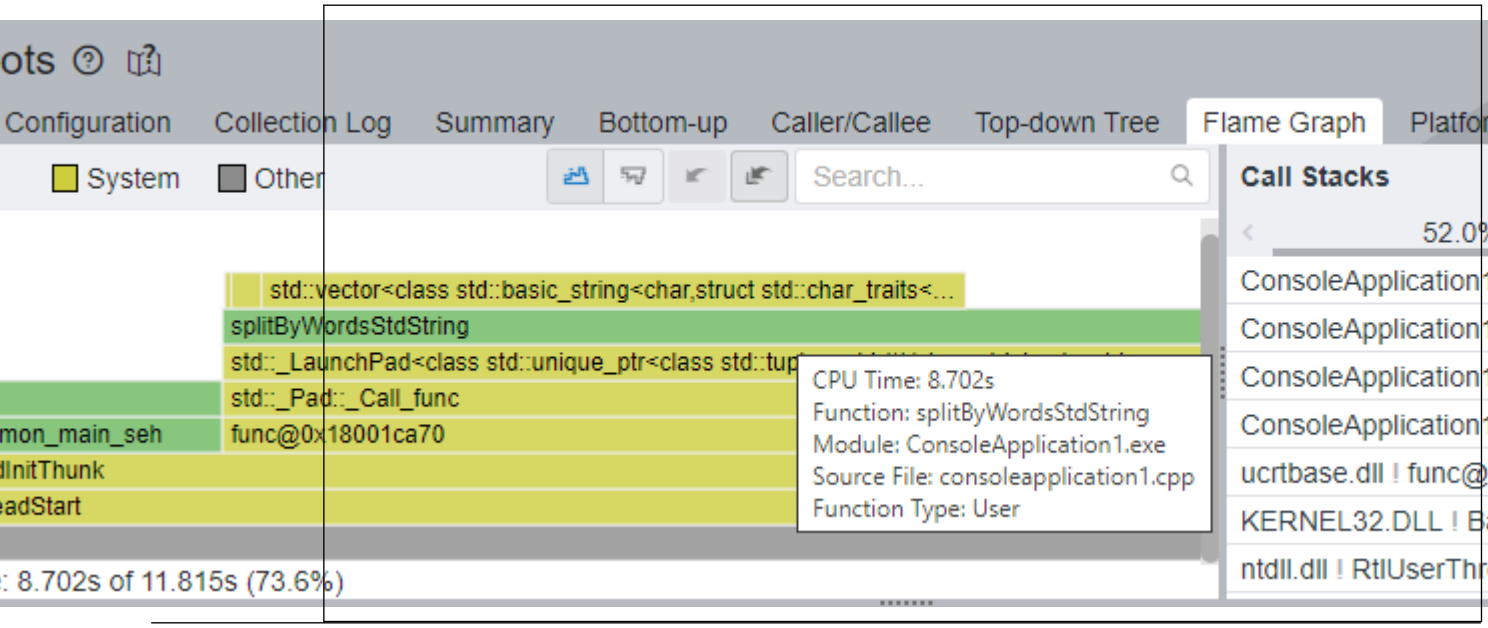
Once the analysis is done, pay attention to the **Elapsed Time** value and the **Top Hotspots** section of the **Summary** window.



The **Elapsed Time** was decreased from roughly 17 seconds to roughly 12 seconds. The **Top Hotspots** section shows a new list of functions, including the application function `splitByWordsStdString`.

The top hotspot is the `std::vector<>` template with vector re-allocation (`_Emplace_reallocate`) and the `std::string` object constructor. These template functions are showing up as hotspots since our `splitByWordsStdString` function performs other operations that are expensive in terms of overhead: it constructs new `std::string` objects and often re-allocates the buffer for our output vector.

Switch to the **Flame Graph** window to get the full picture of the application stacks.



A clear hot path is passing through the `splitByWordsStdString` function, with the `std::vector` template frame at the top. The `splitByWordsStdString` function and its callees take 8.702 seconds (73.6%) of all CPU Time vs 13.992 seconds (81.7%) of the previous implementation. A large portion of CPU Time is still spent on vector re-allocation and string creation.

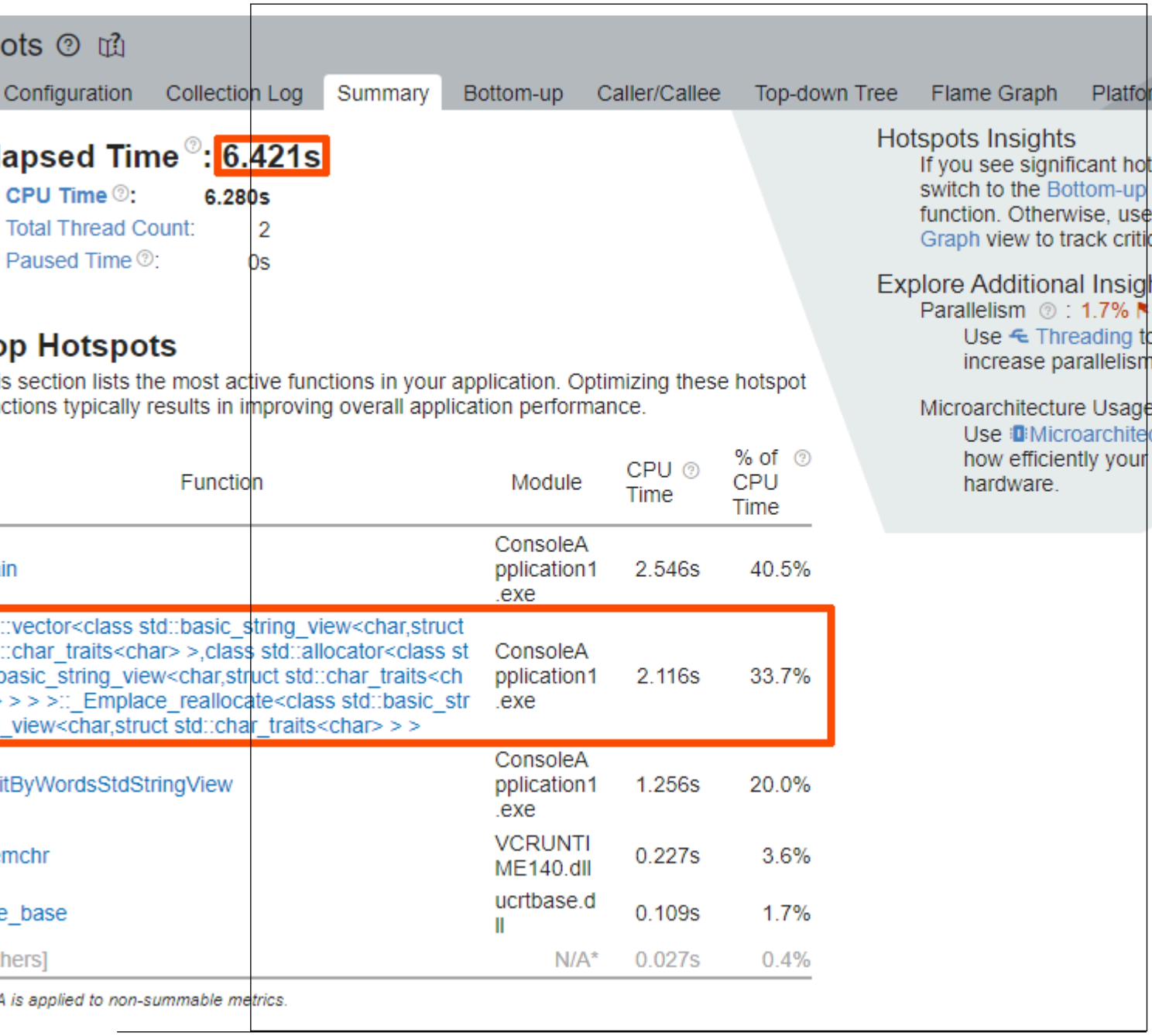
Therefore, let us try one more implementation based on `std::string_view` of the C++17 standard, which offers the benefits of the `std::string` interface without the cost of constructing an `std::string` object. The output vector will contain `std::string_view` objects.

Analyze Third Implementation

To analyze the code with the third implementation of the splitter function, open the **Configure Analysis** dialog and add **3** to the **Application Parameters** field. This enables the second implementation inside the application.

Click **Start** to run the analysis.

Once the analysis is done, pay attention to the **Elapsed Time** value and the **Top Hotspots** section of the **Summary** window.

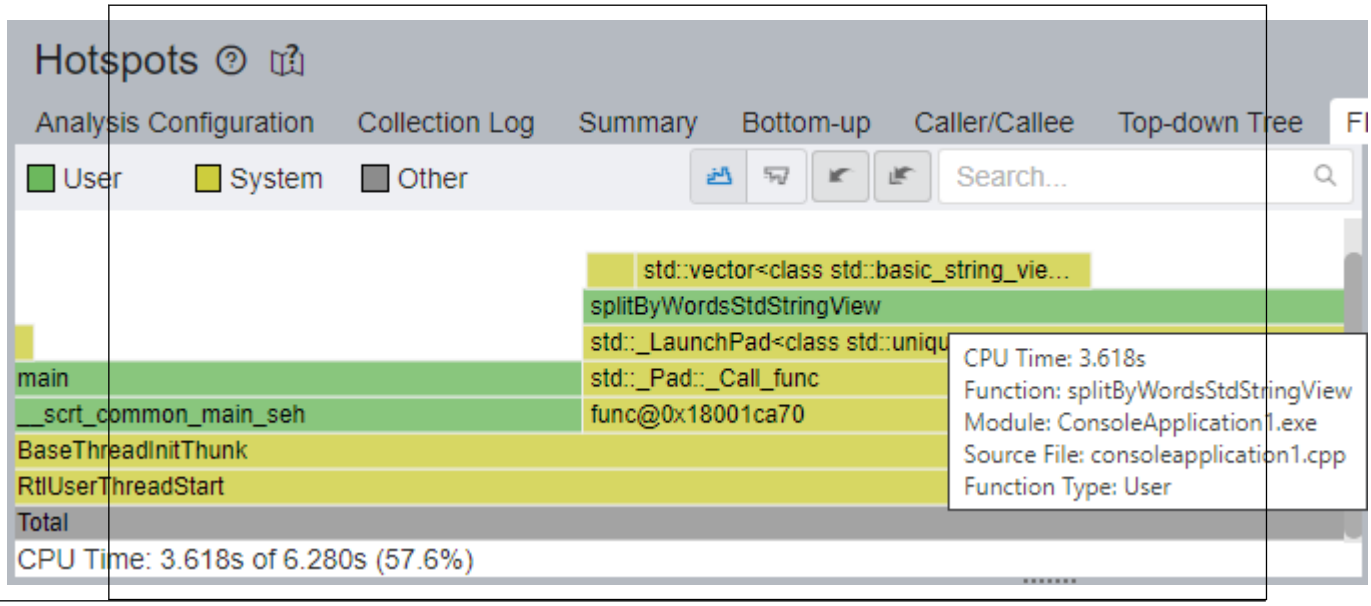


Once again, the **Elapsed Time** of the application was decreased from roughly 12 seconds to roughly 6.5 seconds.

The **Top Hotspots** section shows a new list of functions, including the application function `splitByWordsStdStringView`. The top hotspot is the `main` function, which poses no interest in terms of optimization. The second hotspot is the `std::vector<>` template with vector re-allocation (`_Emplace_reallocate`).

The `splitByWordsStdStingView` function adds split words to the vector of `string_view` objects, bypassing many `std::string` object creations. This has increased the split function performance significantly.

Switch to the **Flame Graph** window to get the full picture of the application stacks.



A clear hot path is passing through the `splitByWordsStdStringView` function, with the `std::vector` template frame at the top. The `splitByWordsStdStringView` function and its callees take 3.618 seconds (57.6%) of total CPU Time, compared to 8.702 seconds (73.6%) of the previous run.

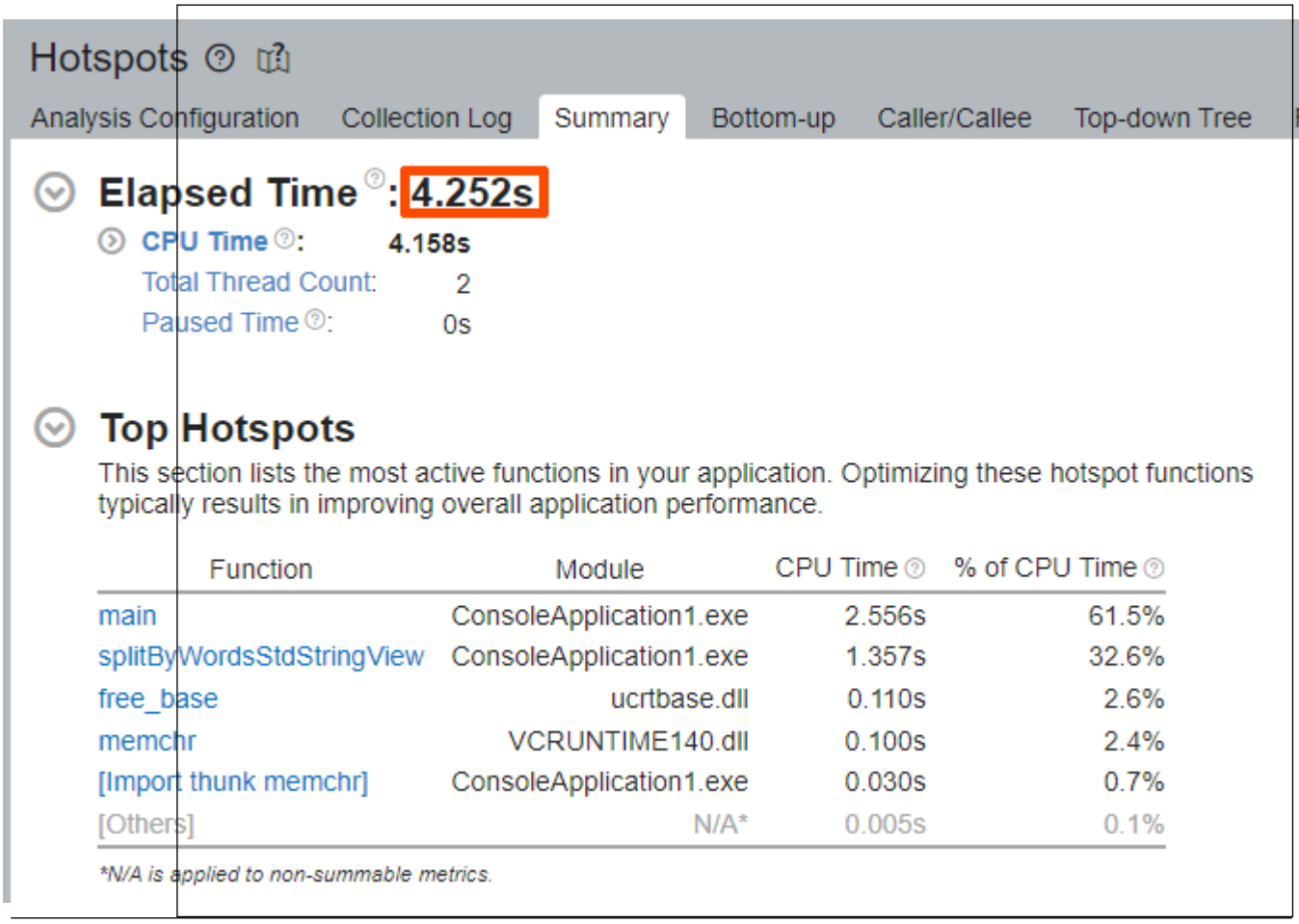
Let us try the final implementation of the splitter function based on `std::string_vew`, this time using pre-allocation of the output vector by using the `std::vector<>::reserve` method. If we can roughly estimate the amount of data that is going into our vector, we might as well pre-allocate some buffer space to avoid most of the re-allocations.

Analyze Fourth Implementation

To analyze the code with the fourth implementation of the splitter function, open the **Configure Analysis** dialog and add **4** to the **Application Parameters** field. This enables the second implementation inside the application.

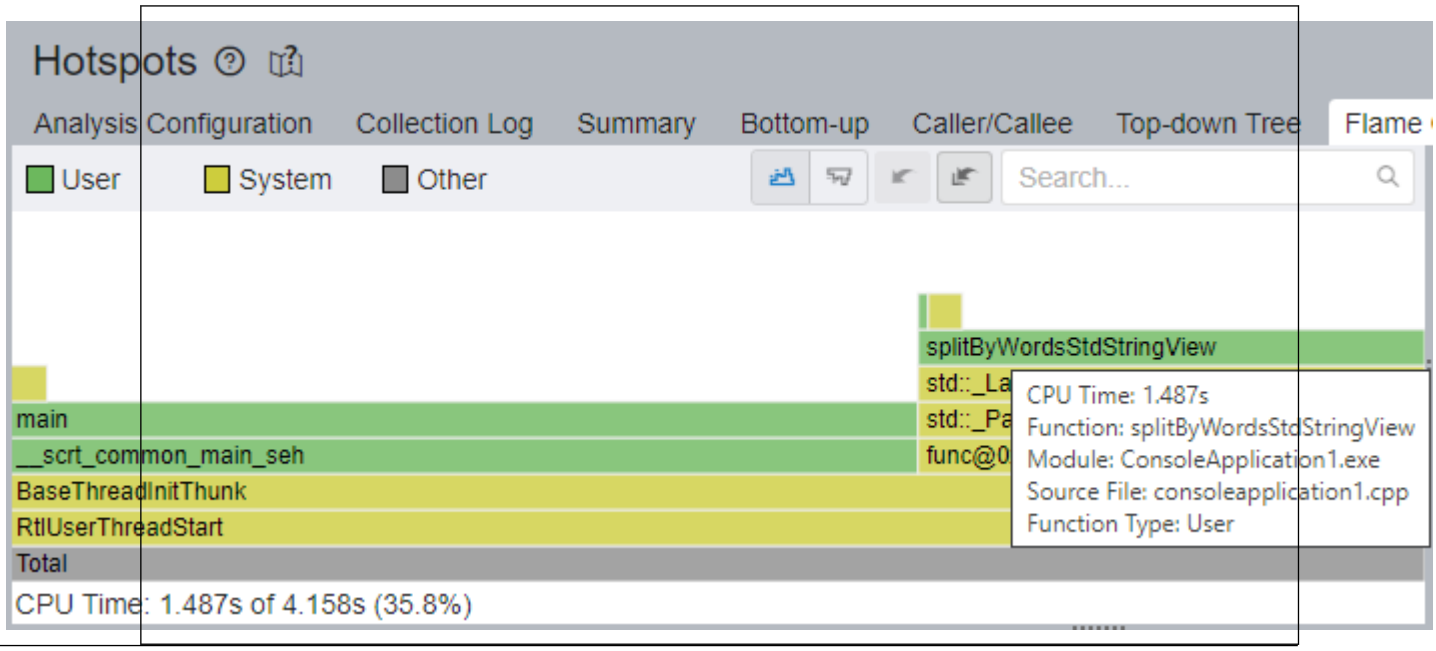
Click **Start** to run the analysis.

Once the analysis is done, pay attention to the **Elapsed Time** value and the **Top Hotspots** section of the **Summary** window.



Once again, the **Elapsed Time** was decreased from 6.421 to 4.252 seconds. There are no STL template functions among the **Top Hotspots**.

Switch to the **Flame Graph** window to get the full picture of the application stacks.



The hottest code path is passing through the `main` function now. The `splitByWordsStdStringView` function and its callees take 1.487 seconds (35.8%) of CPU Time, compared to 3.618 seconds (57.6%) from the previous run.

Summary

Considering the current gains in performance, optimization activities are stopped.

At each step of the iterative optimization process, the Flame Graph helped by being a visual guide into the stack profiles of our application. In real projects, finding hotspots may not be as easy as looking at the **Top Hotspots** section, and the Flame Graph can help uncover the hot paths that require optimization with less time and effort.

Here is how the optimizations progressed with each iteration:

Iteration	Elapsed Time, s	Split Function CPU Time, s
1	17.328	13.992
2	12.016	8.702
3	6.421	3.618
4	4.252	1.484
Total	- 13.076	- 12.508

We started with the first implementation that used the built-in `boost::split` function of the Boost library. Not satisfied with the performance, and being unable to change the function, we switched to our own implementation that uses `std::string`.

On second implementation, we discovered that the `std::string` implementation could also use some improvement. The third implementation used `std::string_view` to avoid constant `std::string` object creations and copying of string buffers, which improved performance. Finally, we opted to pre-allocate some space to our output vector to bypass frequent re-allocation operations.

All these optimizations enabled our application to perform the same amount of work in roughly 4 seconds, compared to roughly 17 seconds at the start.

Analyzing Hot Code Paths Using Flame Graphs (NEW) Follow this recipe to understand how you can use Flame Graphs to detect hot spots and hot code paths in Java workloads.
Window: Flame Graph
User Guide: Hotspots Analysis
Flame Graphs Article by Brendan Gregg

Measuring Performance Impact of NUMA in Multi-Processor Systems

Use this recipe to measure the performance impact of non-uniform memory access (NUMA) in multi-processor systems. This recipe uses the Intel® VTune™ Profiler-Platform Profiler application.

NOTE Starting with the 2024.0 release of Intel® VTune™ Profiler, the **Platform Profiler** application is available as a separate download. Access this application from the Intel Registration Center.

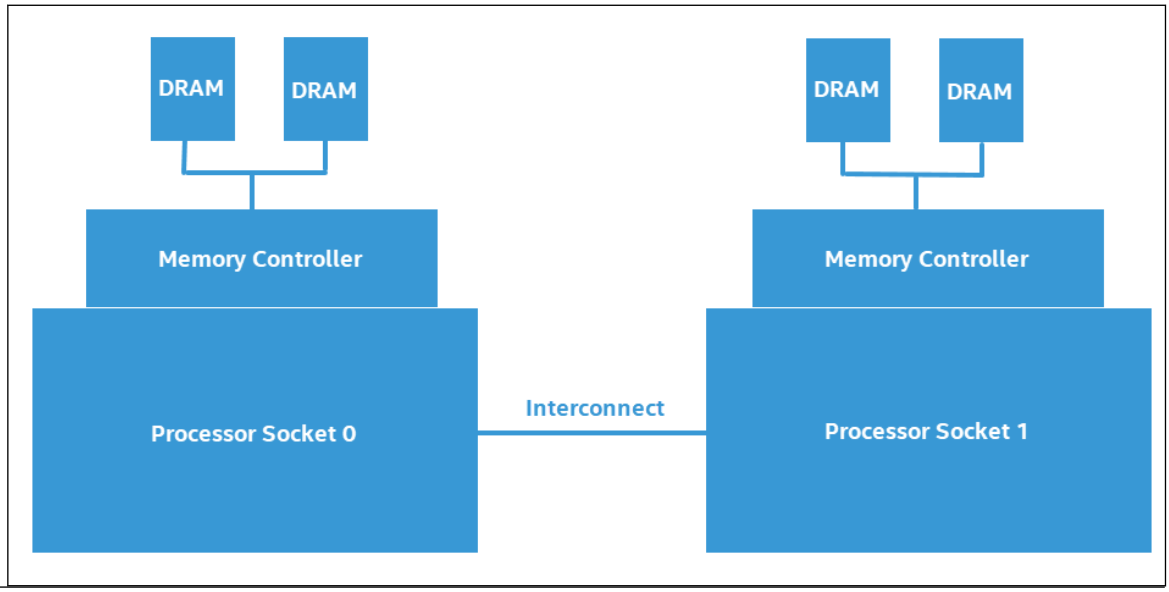
The **Platform Profiler** application will be discontinued in a future release. As a workaround, consider using the EMON data collector. To learn more, see this transition article.

- To collect platform behavior data using the 2023.2 or newer versions of VTune Profiler, use the standalone Platform Profiler collector. You can then visualize collected data with the Platform Profiler server. See Platform Analysis for more information.
- The Platform Profiler analysis type is not available in versions of VTune Profiler newer than 2023.2. To use the Platform Profiler analysis type (from the GUI or command line), switch to a version of VTune Profiler older than 2023.2. You can then follow procedures described in Platform Profiler Analysis.

Content expert: Jeffrey Reinemann

Non-uniform memory access (NUMA) is a computer memory design where the time for memory access depends on the location of memory relative to the processor. In NUMA, processor cores can access local memory (where the memory connected to the processor) faster than non-local memory (where the is memory connected to another processor or shared between processors).

This figure illustrates the design of a two-processor NUMA system.



Software that frequently accesses non-local (or remote) memory can suffer a measurable negative performance impact when compared to software that primarily accesses local memory. In this recipe, we look at measuring the negative performance impact of a NUMA system.

- INGREDIENTS
- DIRECTIONS:

  1. Run Platform analysis
  2. Identify NUMA issues
  3. Set thread CPU affinity to fix NUMA issues
  4. Verify NUMA optimizations

Ingredients

This section lists the hardware and software tools used in this scenario.

- **Application:** The sample application used in this recipe is not available for download.
- **Tool:** Intel® VTune™ Profiler-Platform Profiler

Run Platform Analysis

Intel® VTune™ Profiler-Platform Profiler consists of a data collector (run from the command line) and a server implementing a RESTful interface to a time series database. To collect and view platform profiler metrics, you must:

1. Collect data using the data collector on the command line.
2. Import and view results in the Platform Profiler server.

Collect Data

1. Set up the environment for Platform Profiler collector. Run `vpp-collect-vars`.

   In a Linux* environment, run this command:

```
source <INSTALL_DIR>/vpp/collect/vpp-collect-vars.sh
```

   If you have not loaded SEP drivers already, run:

```
vpp-collect config
```

   In a Windows* environment, run this command:

```
<INSTALL_DIR>\vpp\collector\vpp-collect-vars.cmd
```

2. Start data collection. Run this command:

```
vpp-collect start -c 'optional comment about data collection'
```

3. If you want to insert marks in the data timeline, run:

```
vpp-collect mark 'optional comment about workload transition'
```

4. Stop data collection. Run this command:

```
vpp-collect stop
```

After data collection completes, Platform Profiler compresses the results into a `.tgz` (Linux) or `.zip` (Windows) file whose name contains the name of the target system and a date/time stamp.

View Results

You view results in the Platform Profiler server. Follow these steps:

1. Set up the environment for Platform Profiler server. Run `vpp-server-vars`.

   In a Linux environment, run this command:

```
source <INSTALL_DIR>/vpp/server/vpp-server-vars.sh
```

   In a Windows environment, run this command:

```
<INSTALL_DIR>\vpp\server\vpp-server-vars.cmd
```

> **NOTE** When setting up the environment for the first time, make sure you have root/administrator access privileges.

2. Create the virtual Python* environment for Platform Profiler server. Run:

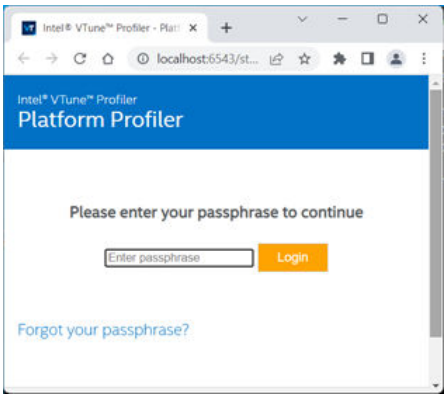
```
vpp-server config
```

3. Start the Platform Profiler server. Run this command:

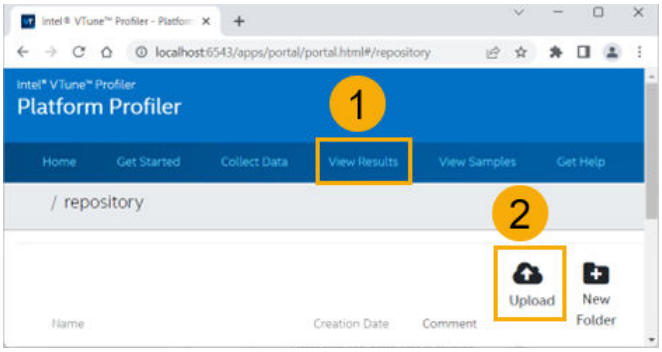
```
vpp-server start
```

4. If necessary, specify the location of the database (for results) and a passphrase for access.

5. Open a web browser to the address and port number of the server instance of Intel® VTune™ Profiler-Platform Profiler (e.g., `localhost:6543`)

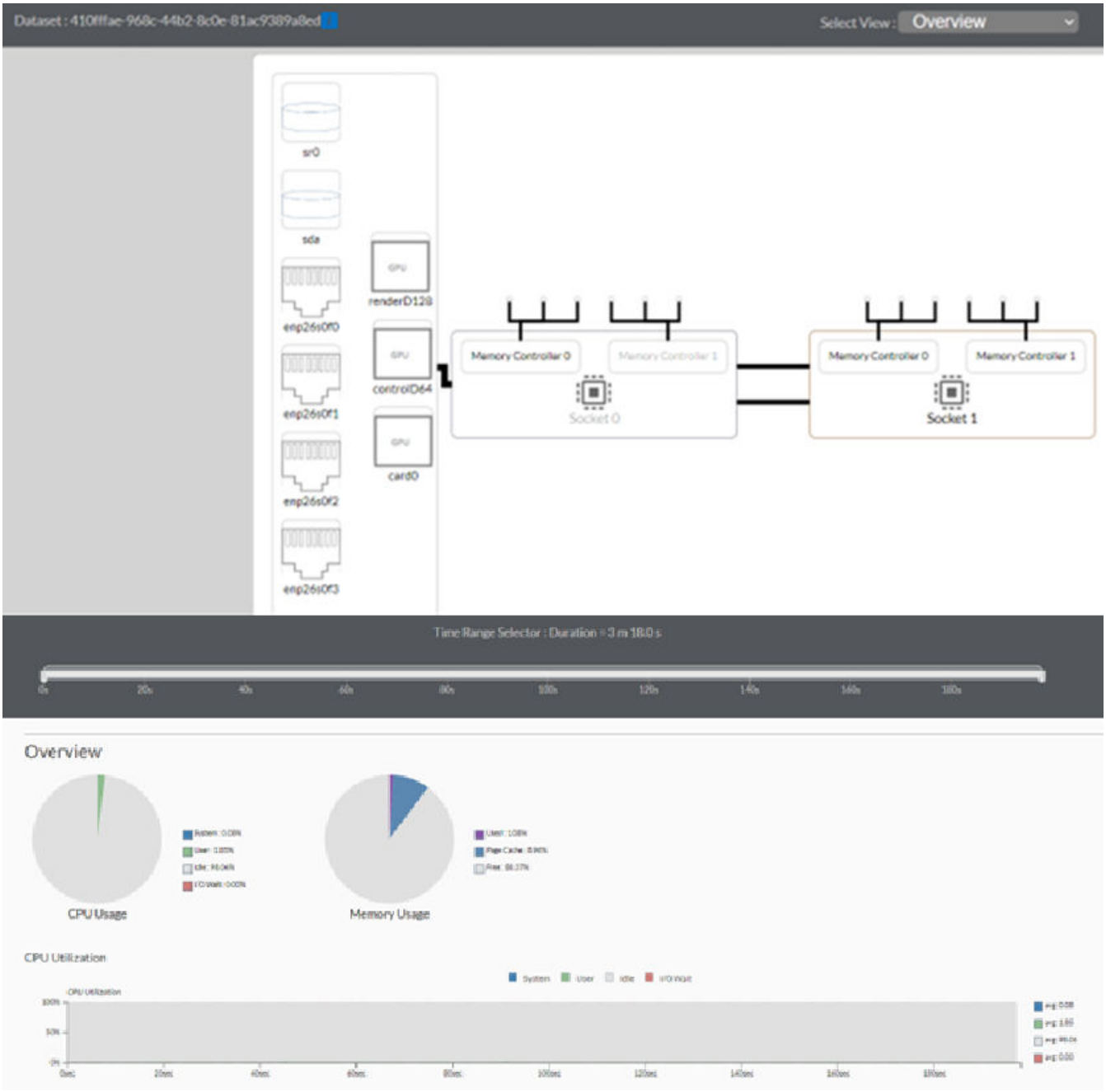


6. Enter the database access password.
7. Import the Platform Profiler collector results file. In the **View Results** tab, click the **Upload** button.



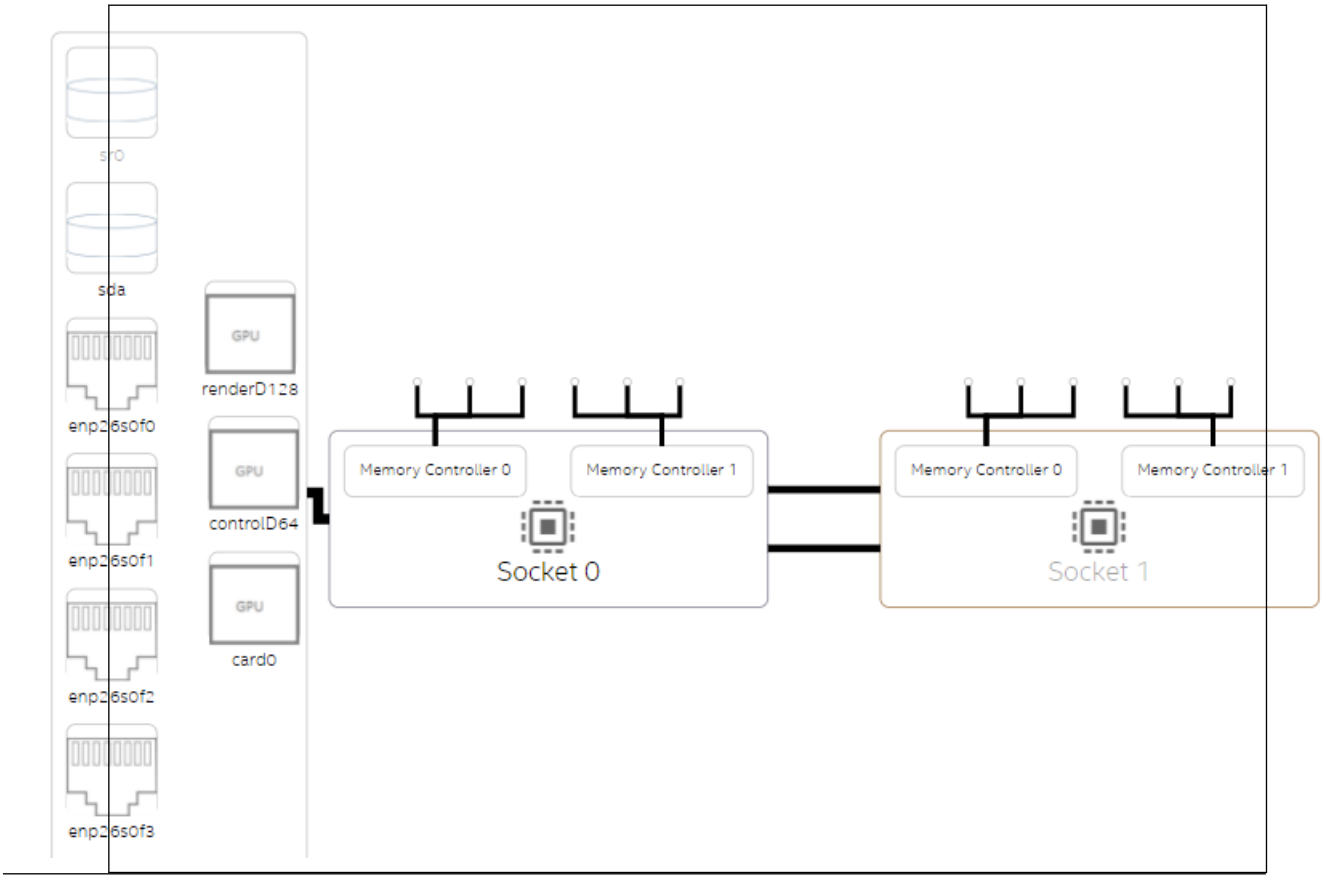
After the import completes, Platform Profiler displays information in three key areas:

- The Platform configuration diagram
- An interactive time line
- Detailed performance data charts



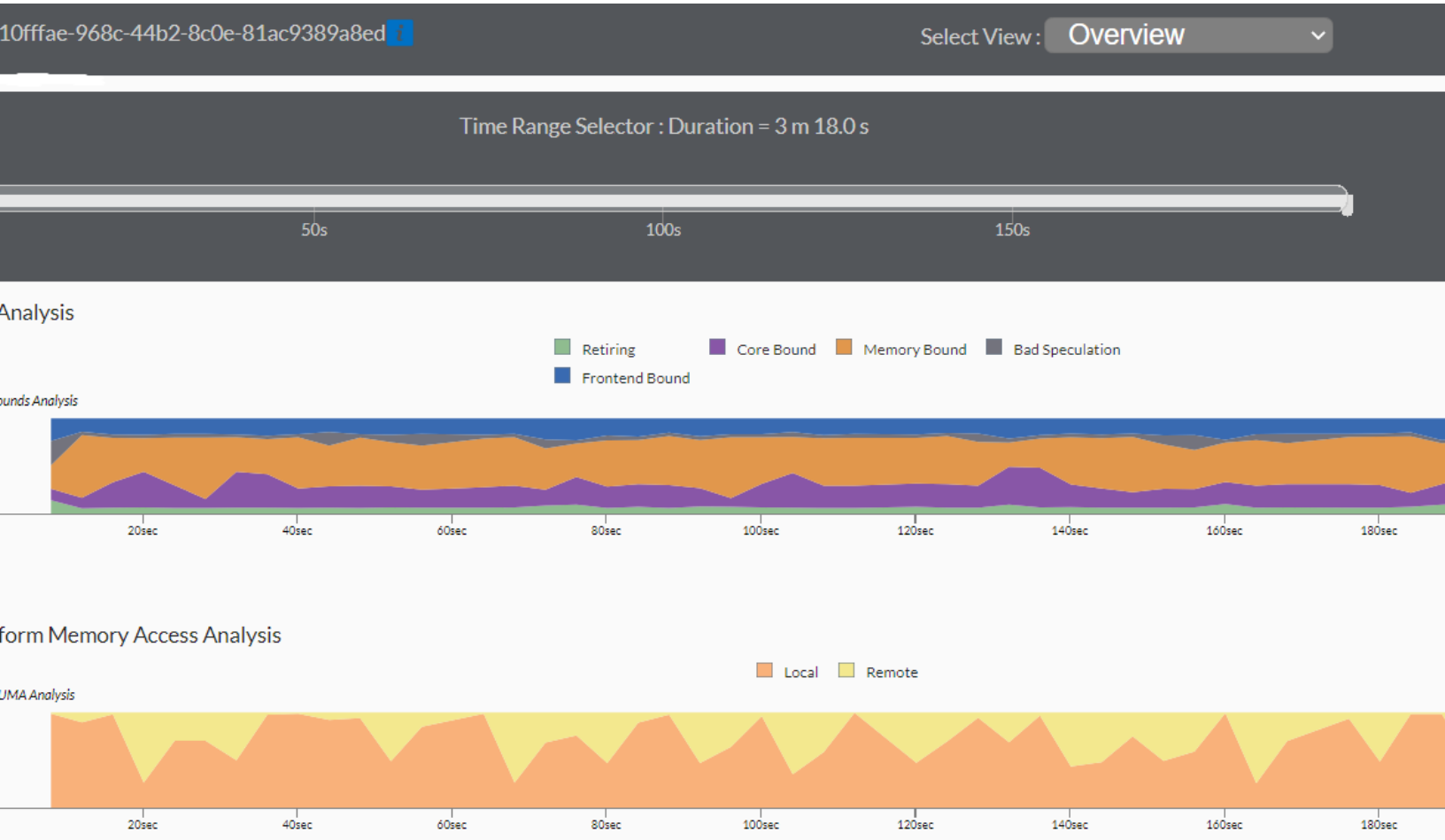
Identify NUMA Issues

1. Start your analysis of NUMA issues by selecting **Overview** in the **Select View** pulldown menu. In the system configuration overview, you can see that a NUMA system typically has multiple processor sockets, each of which has memory controllers. This diagram is an example of a platform diagram for a two-socket system.

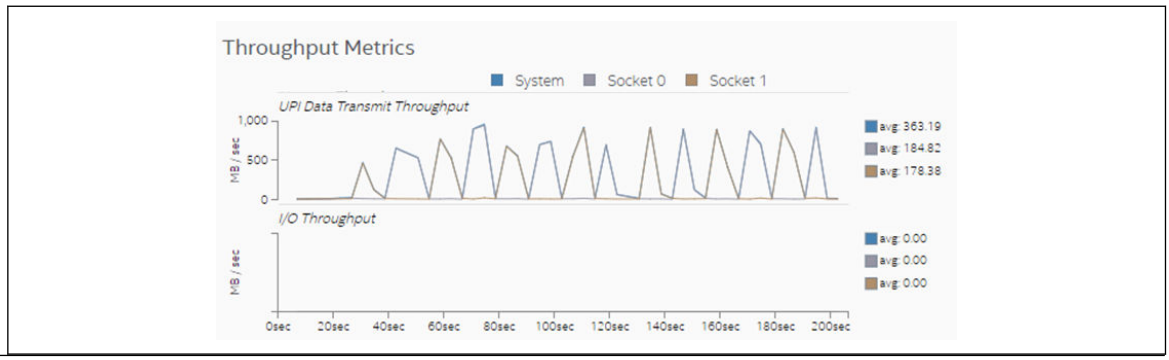


In the case of the sample application used here. the performance is mostly bound by memory access. There are a high number of memory accesses that are targeted to remote memory.

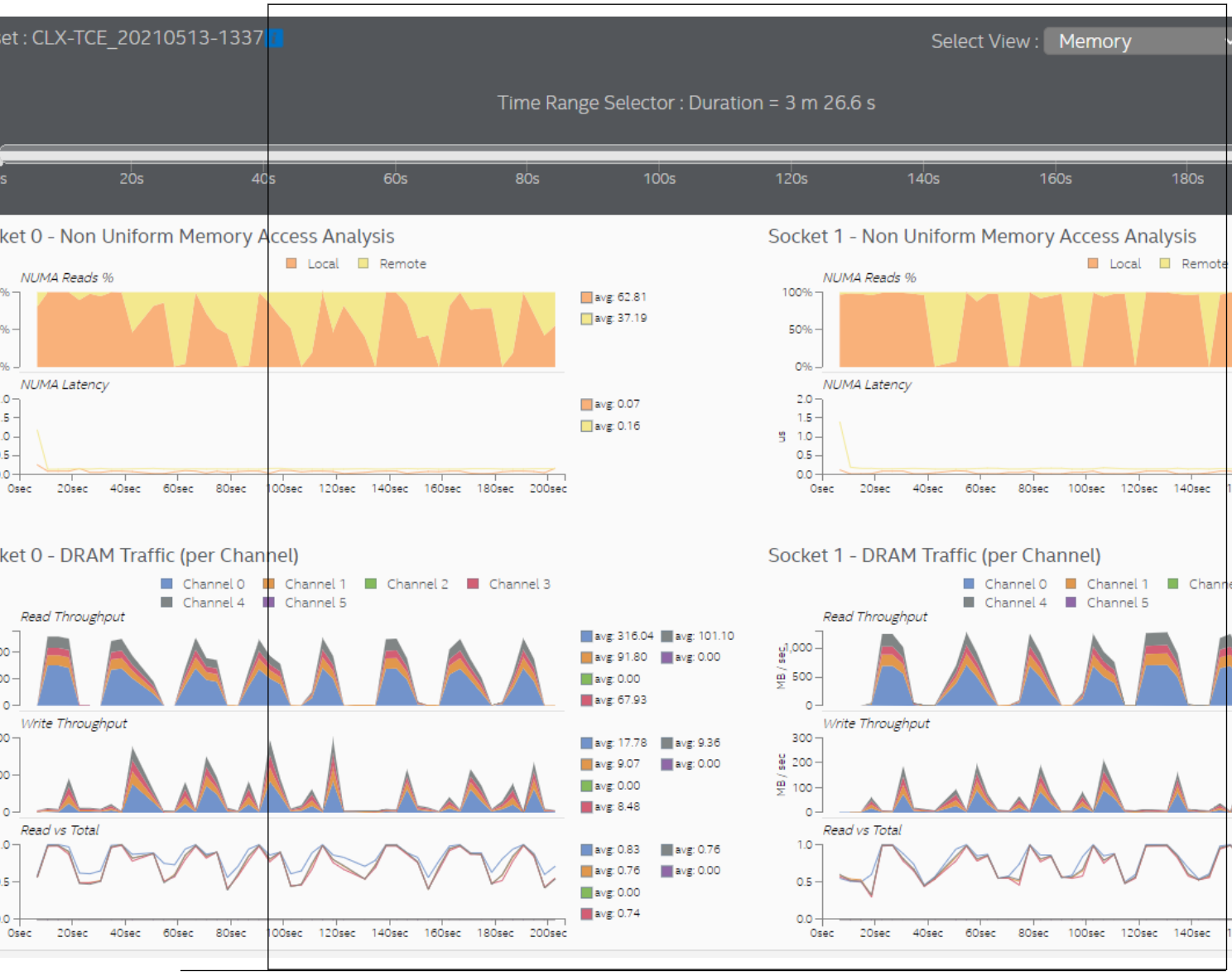
2. See the **Non-Uniform Memory Access Analysis** graph to compare local vs. remote memory accesses over time. A high percentage of remote accesses indicates a NUMA related performance issue.



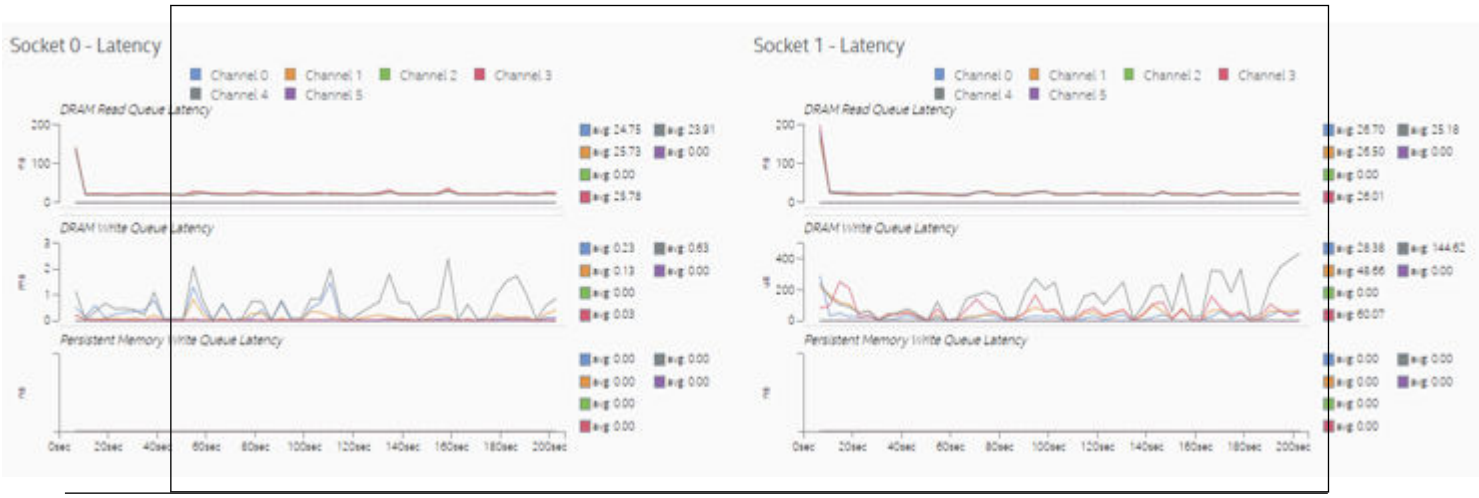
3. Observe the **Throughput Metrics** section. In this sample application, there are frequent spikes in cross-socket (UPI) traffic. These spikes correspond to remote memory accesses.



4. Switch to the **Memory** view to see additional information about memory accesses for each processor socket. In this sample application, both sockets initiate remote memory accesses.



Latencies in memory accesses spike when the remote memory is accessed. These spikes indicate an opportunity for performance improvement.



5. Switch to the **UPI** view to see the cross-socket traffic transmitted by each socket.



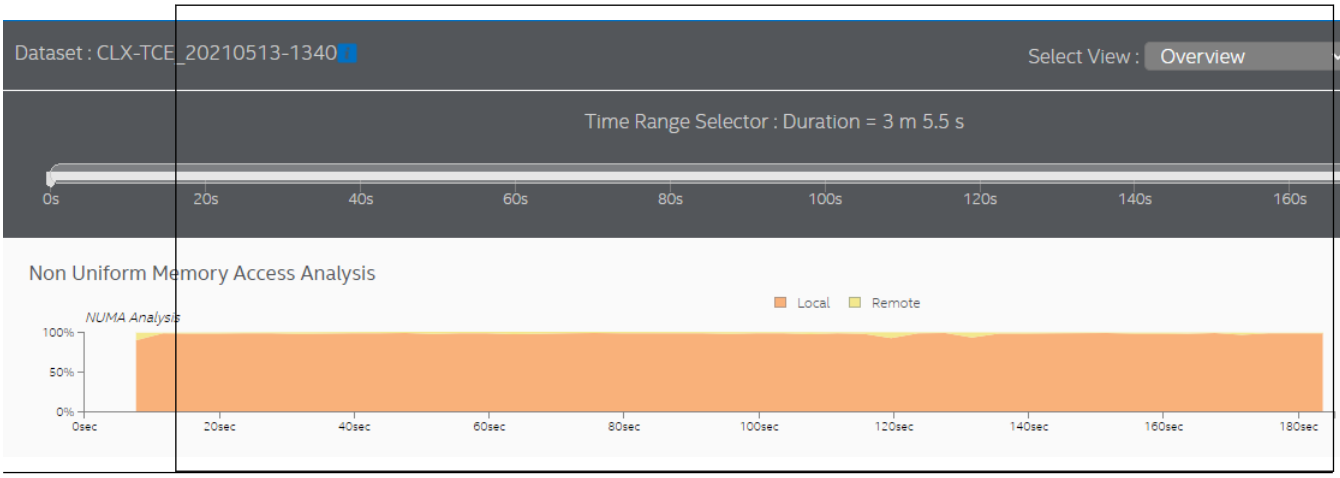
Set Thread CPU Affinity to Fix NUMA Issues

The sample application used in this recipe does not assign (or pin) software threads to specific sockets and cores. The absence of this assignment causes the operating system to periodically schedule threads on processor cores located in different sockets. In other words, the application is not '*NUMA-aware*'. This condition results in frequent accesses to remote memory, which in turn result in high cross-socket traffic and higher memory access latencies.

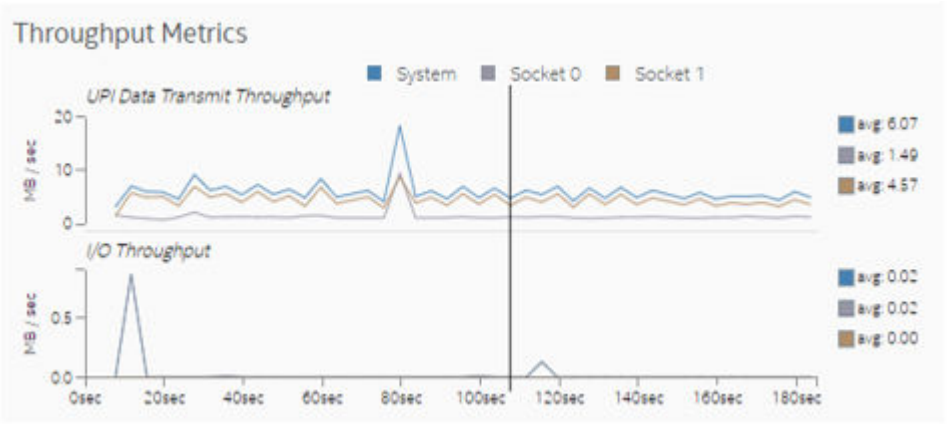
One way to reduce remote memory accesses and cross-socket traffic is to assign the affinity of the processes which execute on the same memory ranges, to processor cores in the same socket. This assignment helps to maintain memory access locality. The programming guide for your operating system may recommend other approaches to reduce cross-socket traffic on NUMA systems.

Verify NUMA Optimizations

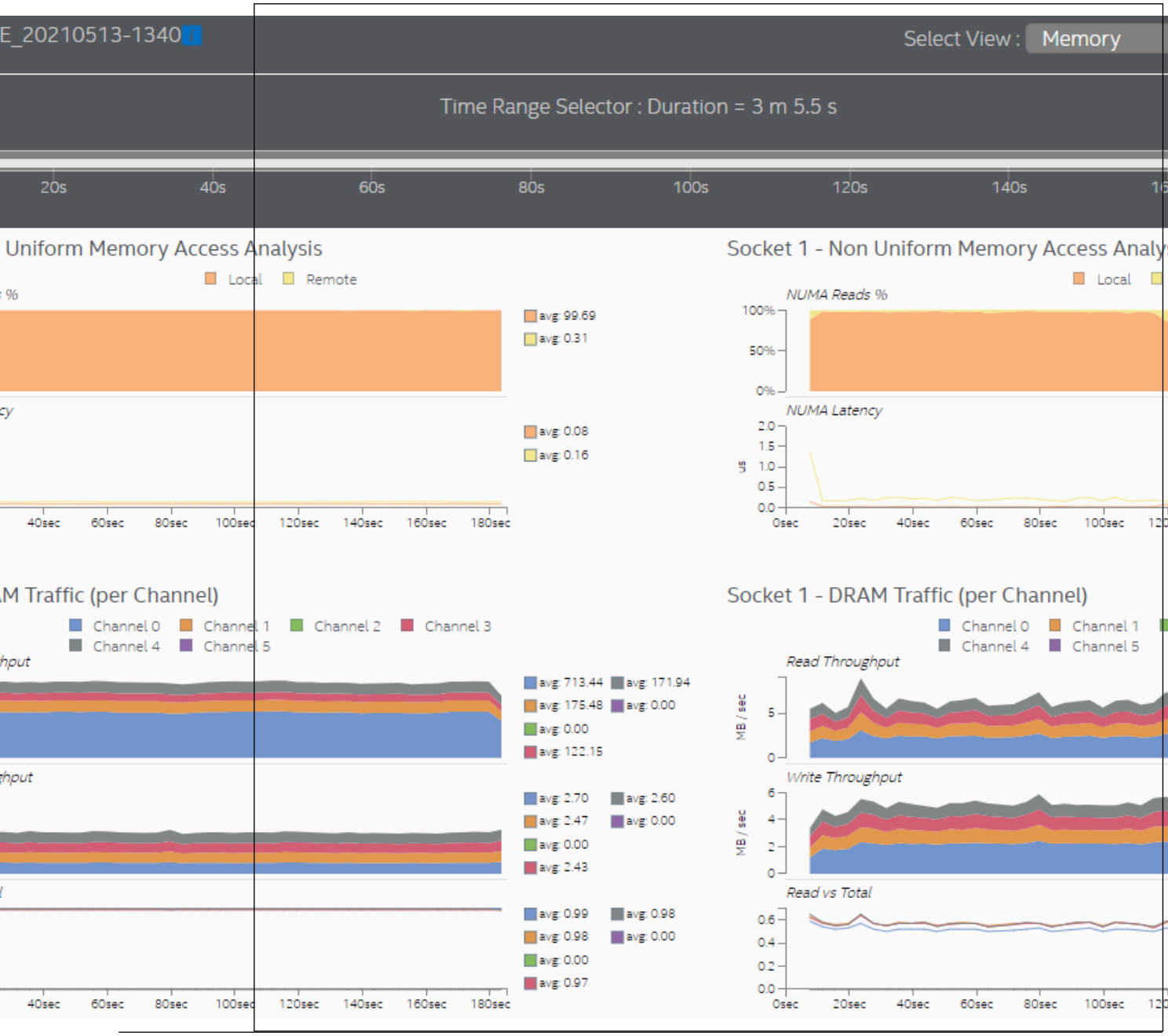
Once you have completed NUMA optimizations, run the Platform Profiler data collection again on the optimized code and import the results into the Platform Profiler server. The next graph shows memory accesses after assigning affinity to cores in socket 0.



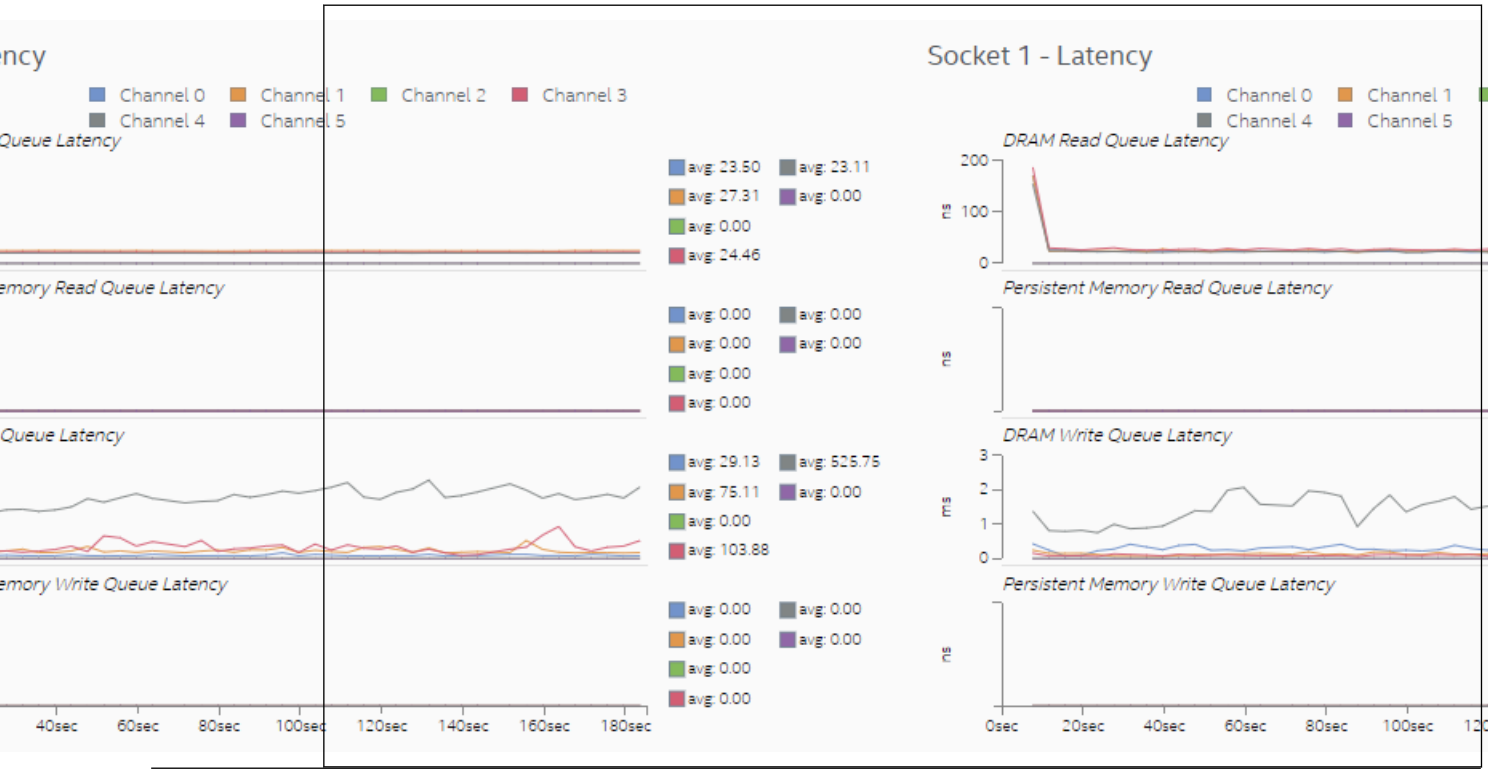
In the sample used here, the results show that almost all memory accesses are now local accesses. Most cross-socket traffic spikes were eliminated.



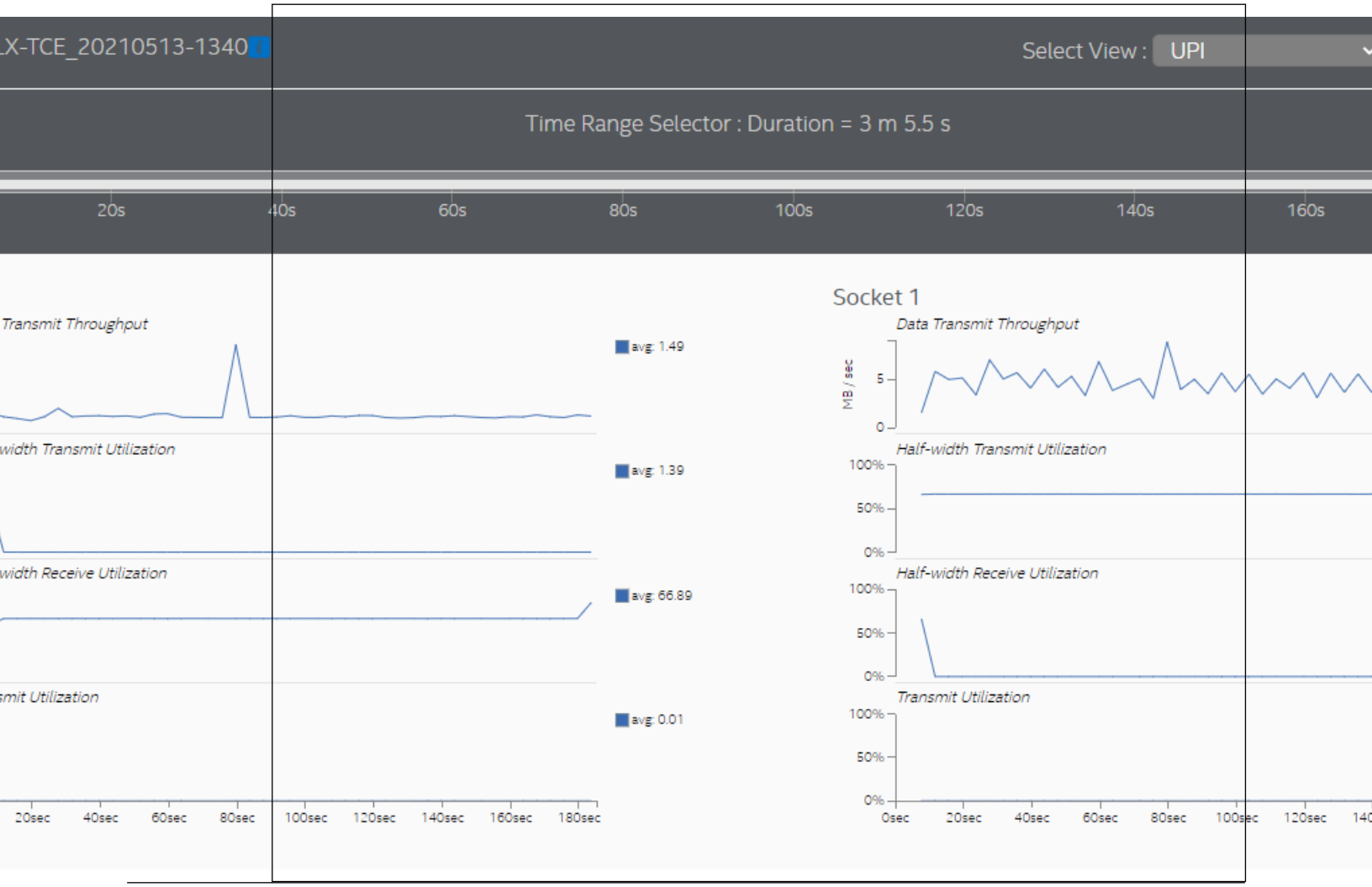
The **Memory** view shows that both sockets are mostly accessing local memory.



Setting thread CPU affinity and completing optimizations helped reduce memory access latencies significantly because more memory requests were directed to local memory. Observe in the **Memory** view that the scale of the **DRAM Write Queue Latency** graph changed from micro-seconds to nanoseconds.



The **UPI** view also shows much lower cross-socket traffic.



NOTE
You can discuss this recipe in the Analyzers community forum.

See Also

Platform Analysis

Profiling Games built with Unity* (NEW)

Use this recipe to profile a game built with the Unity game engine. See how you can run Intel® VTune™ Profiler within the Unity environment to profile your game.

Often, the most important factor that affects the performance of a game is the frame rate. This is the speed with which the GPU renders game graphics. However, the CPU can also impact game performance in several ways:

- Slow transfer of data to the GPU
- Slow or unnecessary operations
- Poor parallelism

With games that use the Unity engine, much of the optimization takes place in the Unity editor. Therefore, it is critical to understand the performance of Unity-defined tasks. The 2019.2 and newer versions of Unity have the Intel® VTune™ Profiler Instrumentation and Tracing Technology (ITT) API built into the Unity editor. This recipe demonstrates how you can run VTune Profiler to highlight Unity tasks in the editor.

- INGREDIENTS
- DIRECTIONS:

  1. Build the Game in the Unity Editor
  2. Configure Intel® VTune™ Profiler and Run Hotspots Analysis
  3. Review Results

Ingredients

Here are the hardware and software requirements for this performance recipe.

- **Application**: `Unity 2022.1.16`. The sample game in this version of Unity uses the free asset 3D Game Kit.
- **Tools**: Intel® VTune™ Profiler version 2022 - Hotspots Analysis (using User-Mode Sampling)

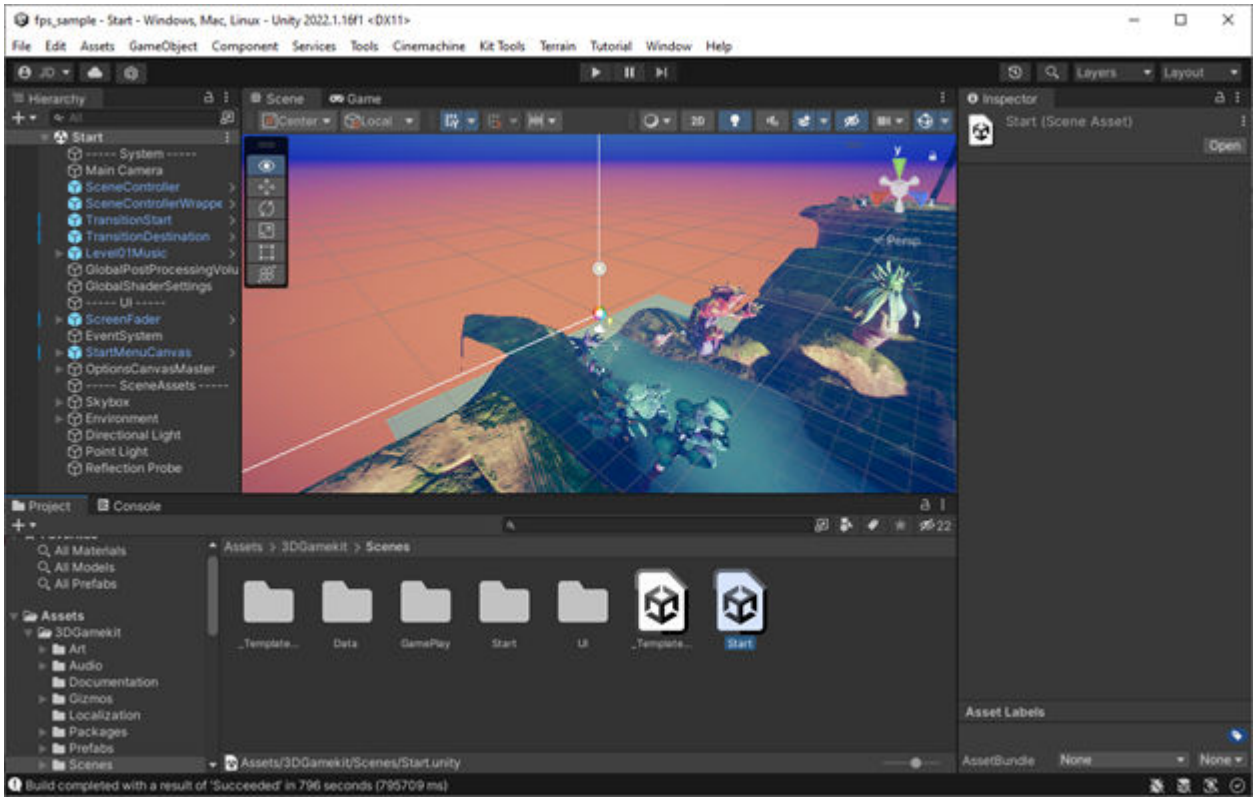
  NOTE

  - Starting with the 2020 release, Intel® VTune™ Amplifier has been renamed to Intel® VTune™ Profiler.
  - Most recipes in the Intel® VTune™ Profiler Performance Analysis Cookbook are flexible. You can apply them to different versions of Intel® VTune™ Profiler. In some cases, minor adjustments may be required.
  - Get the latest version of Intel® VTune™ Profiler:
    - From the Intel® VTune™ Profiler product page.
    - Download the latest standalone package from the Intel® oneAPI standalone components page.

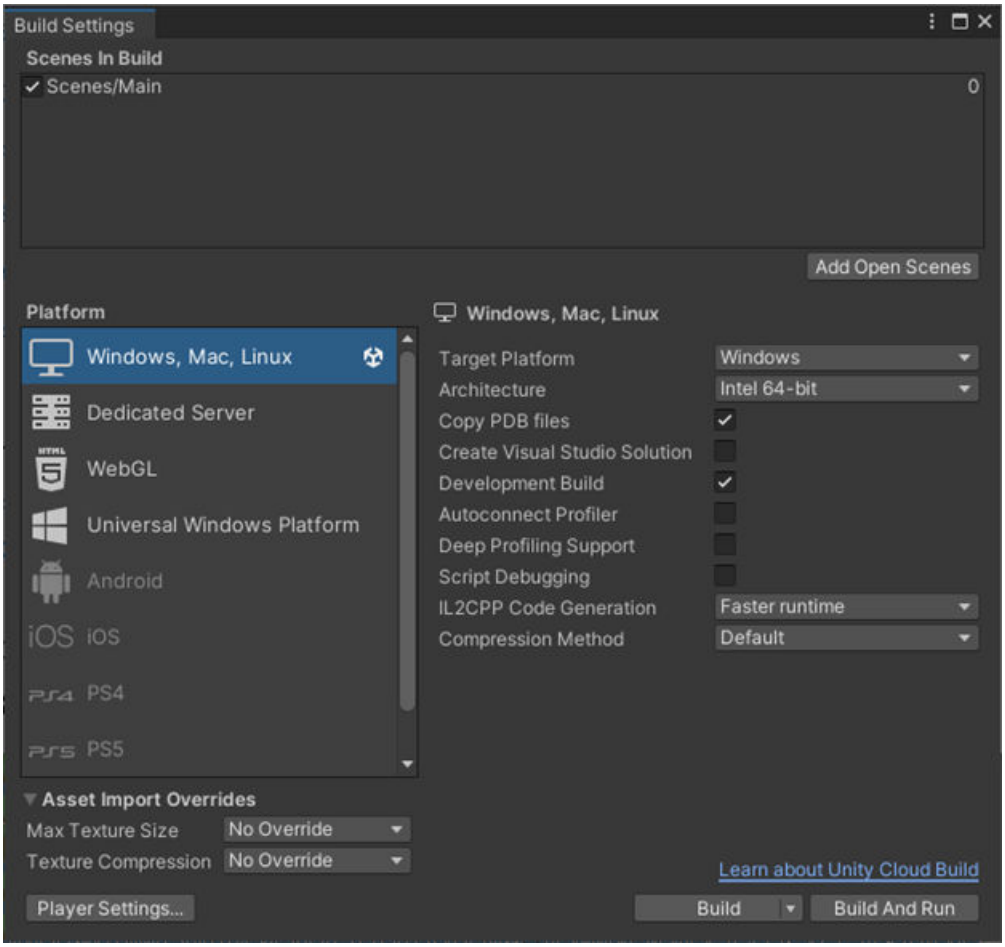
- **CPU/GPU**: Intel® Core™ i7-8665U CPU @ 1.90GHz with integrated GPU
- **Operating system**: Windows* 10 Enterprise

Build the Game in the Unity Editor

1. Open the game in the Unity editor.

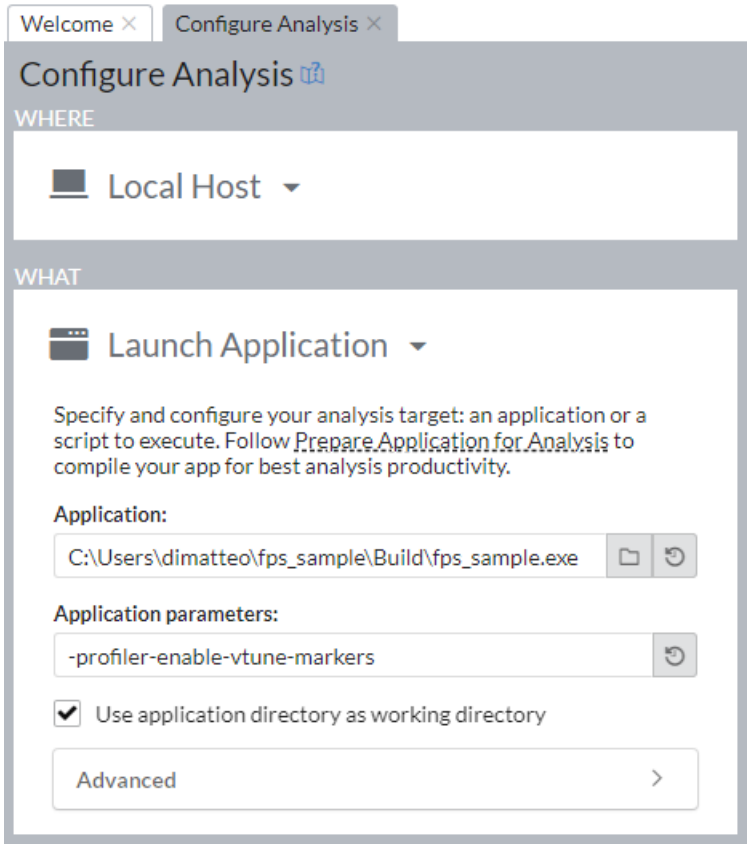


2. Build the game. Make sure to select the **Copy PDB Files** and **Development Build** options.

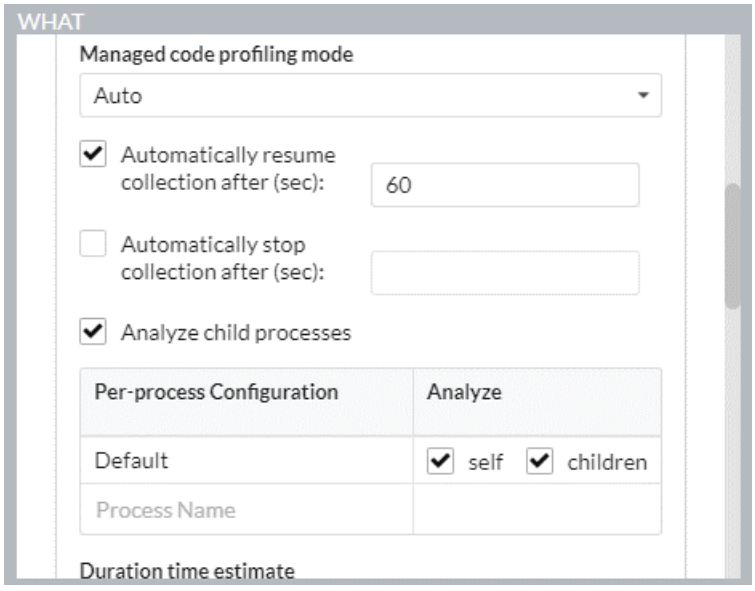


Configure Intel® VTune™ Profiler and Run Hotspots Analysis

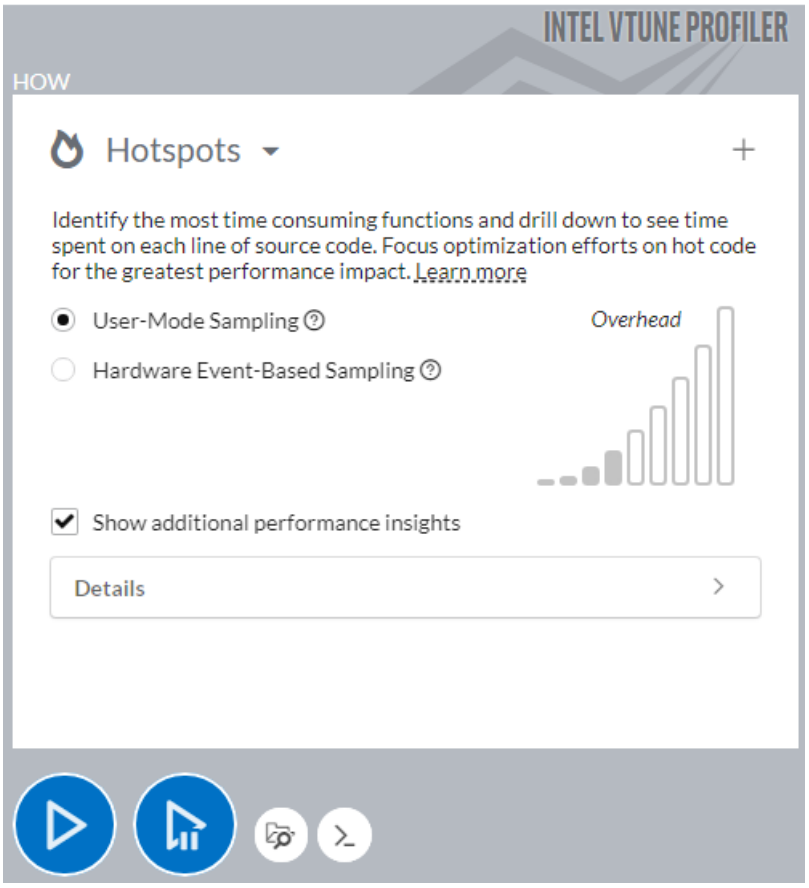
1. Open Intel® VTune™ Profiler and click **New Project** on the Welcome screen.
2. Specify a project name and a location for your project.
3. Click **Create Project**.
4. In the **Configure Analysis** window, set these options:
   - In the **WHERE** pane, select **Local Host**.
   - In the **Application** field of the **WHAT** pane, enter the path to the game executable.
   - In the **Application parameters** field, enter `-profiler-enable-vtune-markers`.



5. (Optional) If you want to skip profiling the start/loading phase of the game,
   - In the **WHAT** pane, open the **Advanced** section.
   - Set **Automatically resume collection after** to the number of seconds that Intel® VTune™ Profiler should wait for profiling to begin.



6. In the **HOW** pane, select the **Hotspots analysis type** and enable user-mode sampling.
7. Click **Start** to run the analysis.



NOTE If you set the **Automatically resume collection after** option in step 5, only the **Start Paused** button (



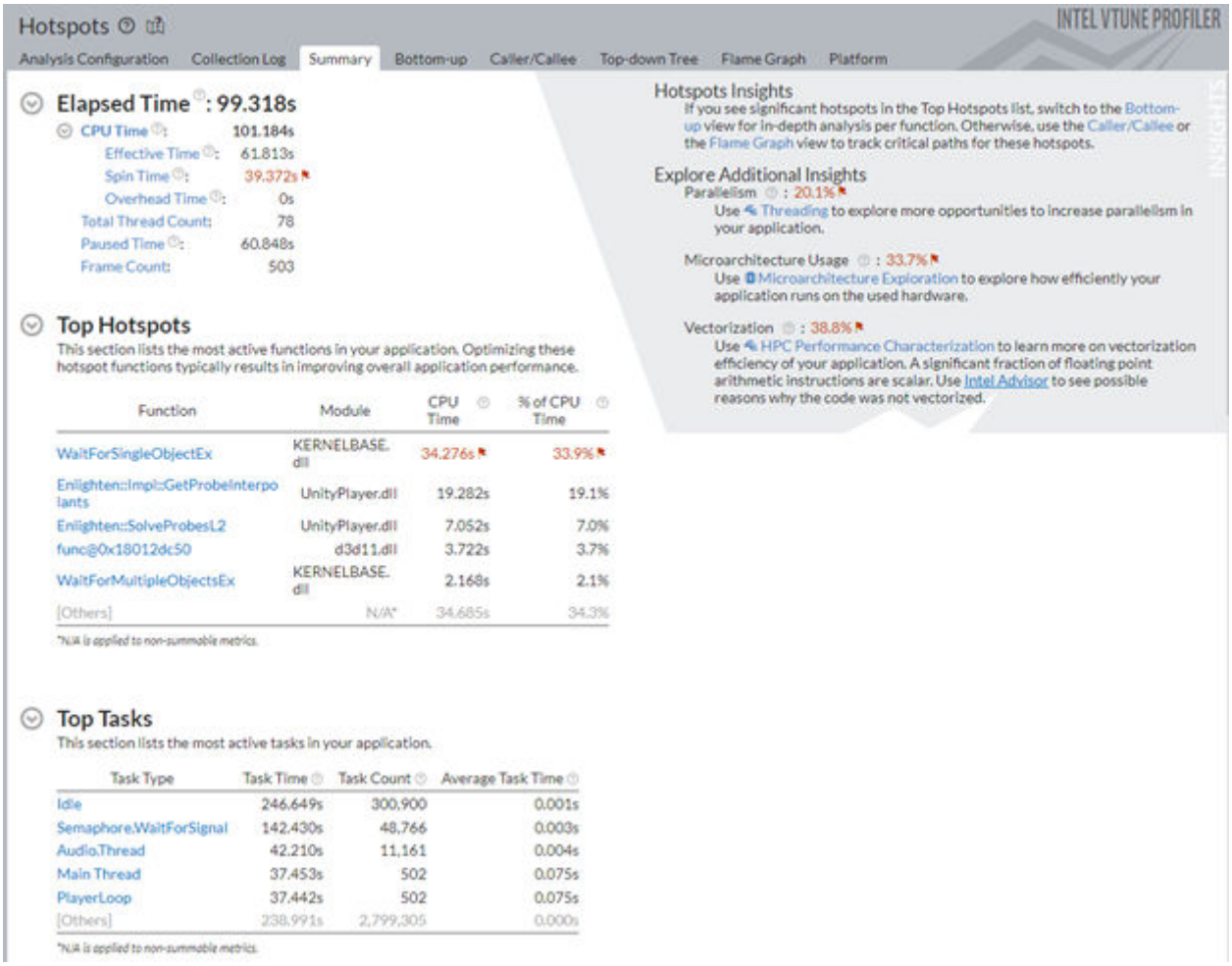
) is available.

Review Results

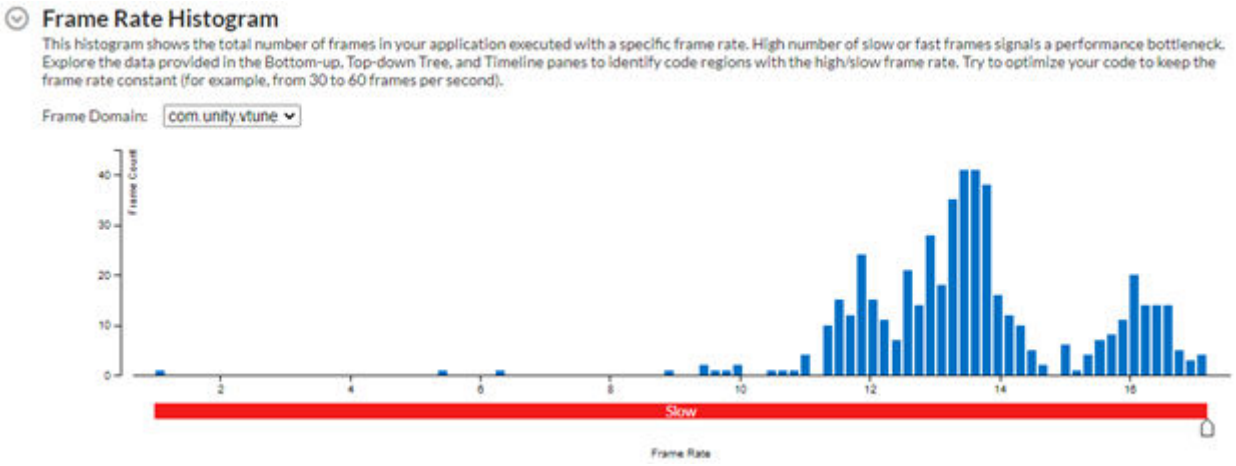
After the data collection runs for about 30 seconds, click **Stop** to exit the game and finalize the VTune results. This process may take a few minutes as Intel® VTune™ Profiler finds and resolves debug symbols.

Once results have been finalized, the **Summary** tab displays information about:

- Elapsed time
- Top hotspots
- Top Unity tasks
- Additional insights and guidance



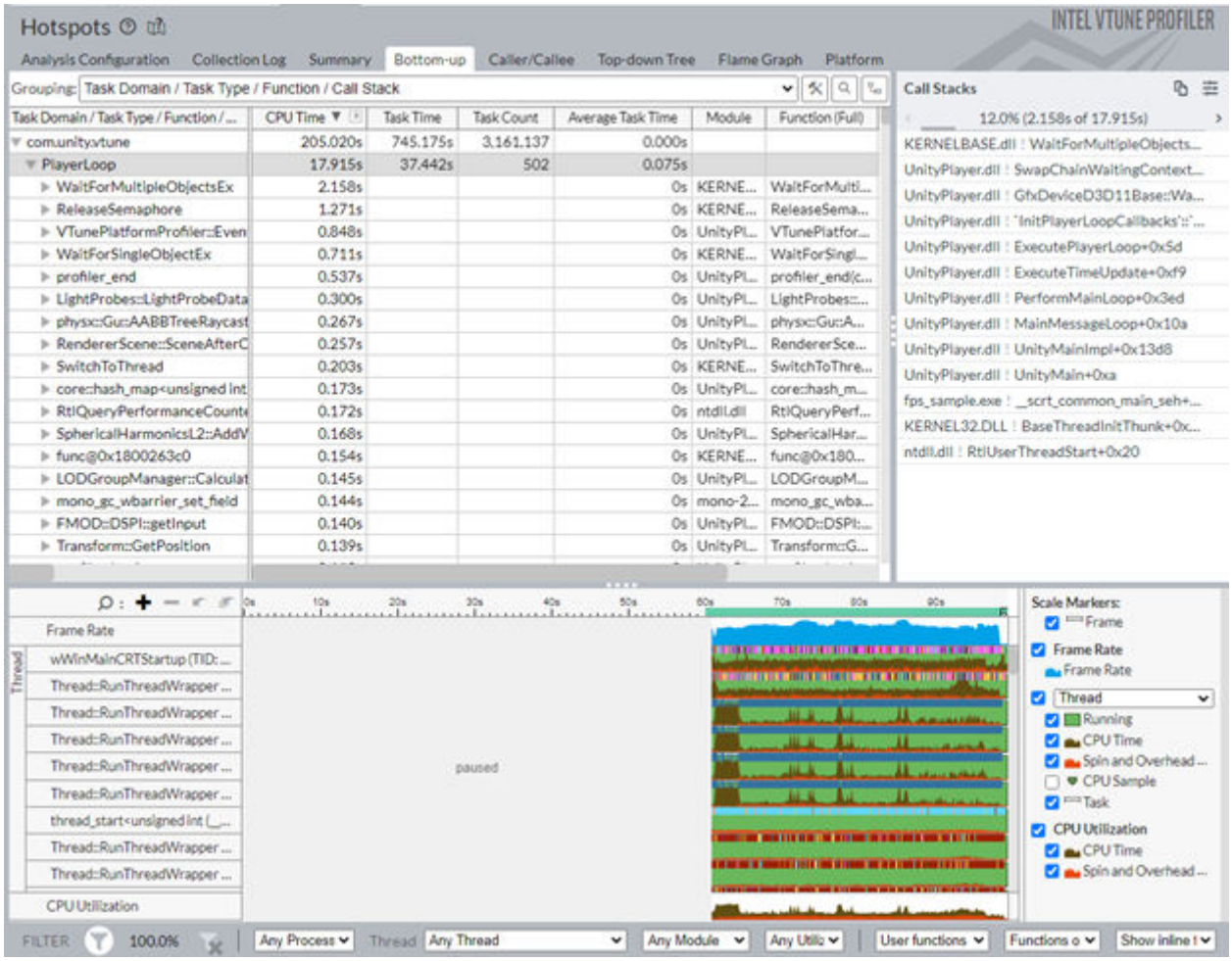
The **Frame Rate Histogram** shows the frequency at which frames executed during the collection.



In this example, most frames executed between 13-14 frames per second (FPS). This is much slower than the recommended minimum of 30 FPS.

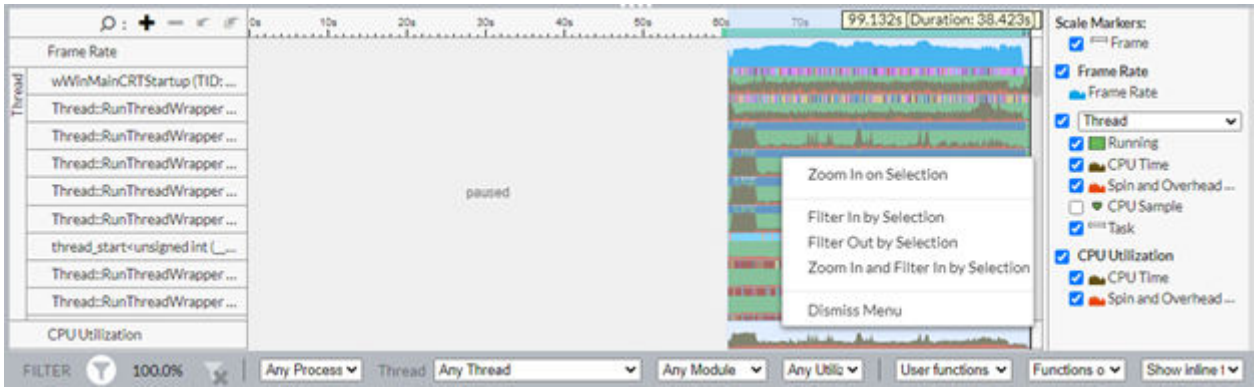
Switch to the **Bottom-up** window to see a list of functions. The default sorting is by descending order of CPU time.

Change the grouping from **Function / Call Stack** to **Task Domain / Task Type / Function / Call Stack** to focus on Unity tasks which were identified by the Intel® VTune™ Profiler Instrumentation and Tracing Technology API (ITT API).



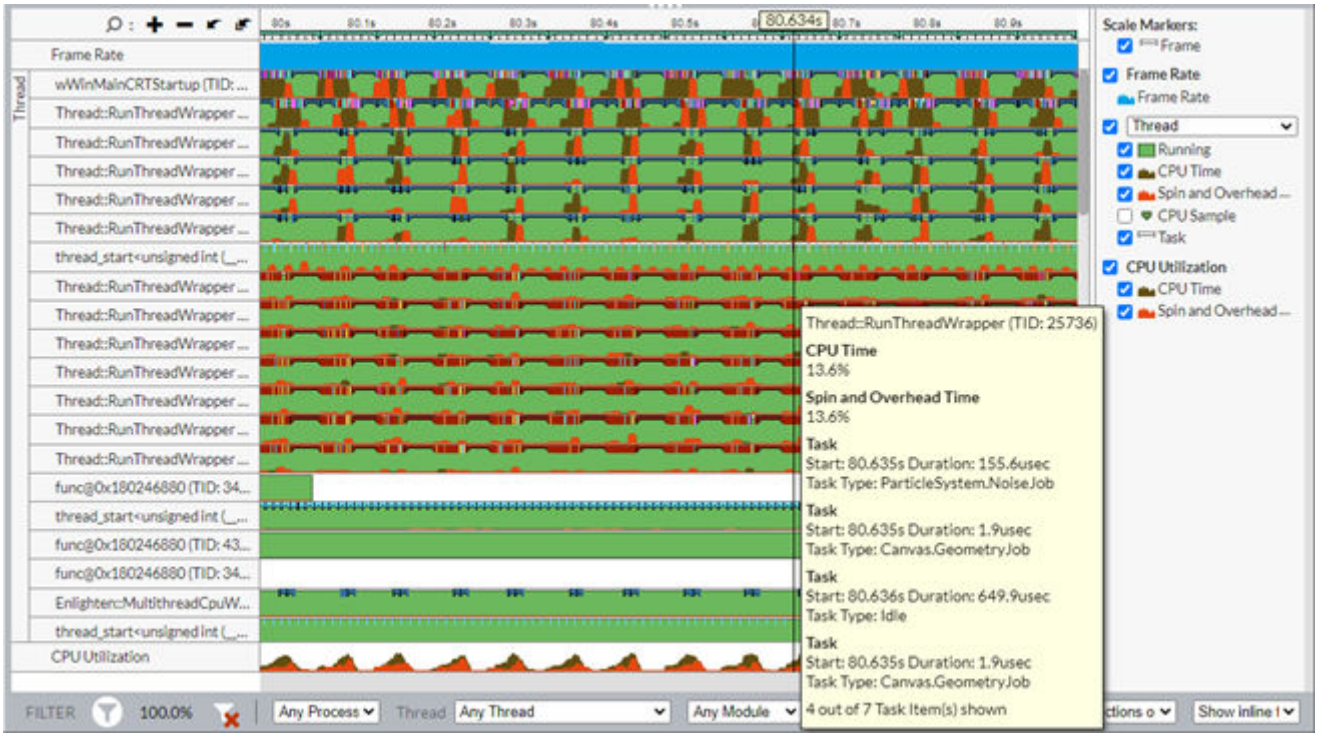
The collection was started in a paused state, and this is indicated in the timeline view as Intel® VTune™ Profiler was still actively running.

Select the time period when collection starts at 60 seconds and zoom in.

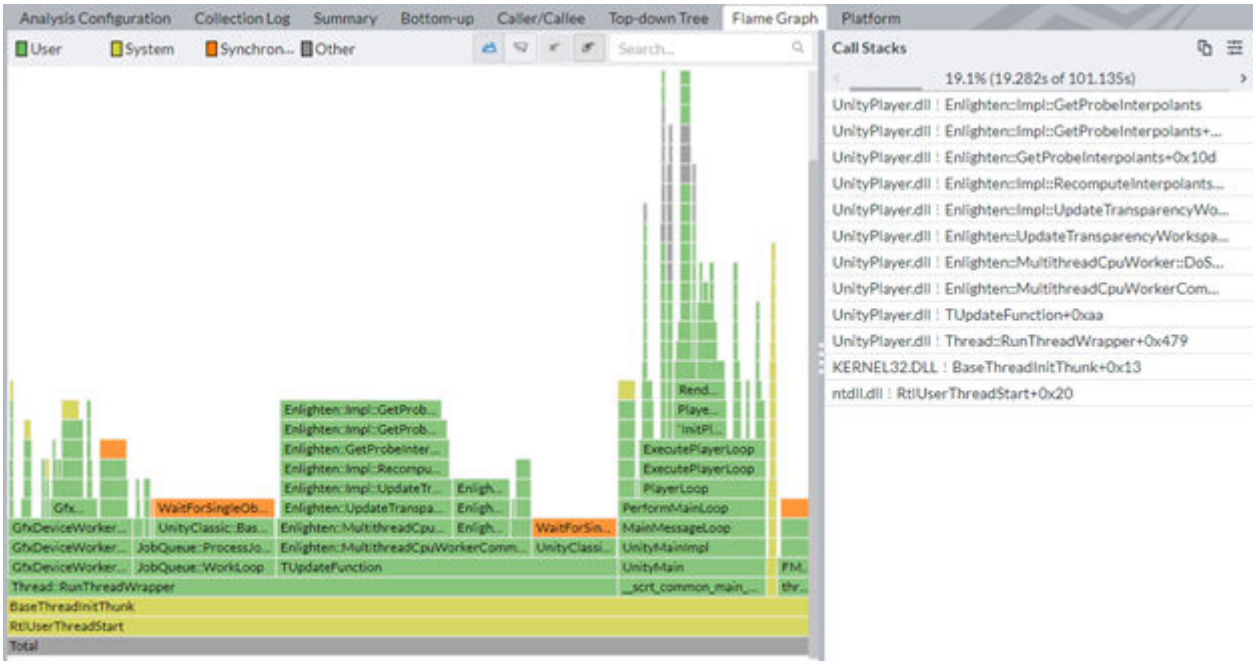


In this example, the frame rate is low and there is considerable spin time. In general, a game should not consume a high percentage of the CPU capacity of the system. If the worker threads have low parallelism, they may be adding too much overhead.

In one of the `RunThreadWrapper` threads, all of the CPU time is spin or overhead. Reducing the number of threads can improve performance here.



For additional insights into hotspots, open the **Flame Graph** view to see a graphical representation of call stacks from the top down.



See Also

Analyzing Hot Code Paths Using Flame Graphs Follow this recipe to understand how you can use Flame Graphs to detect hot spots and hot code paths in Java workloads.

Hotspots Analysis for CPU Usage Issues

Boost CPU Performance with Intel® VTune™ Profiler
Game Tuning with Intel

Profiling Games built with Unreal Engine* (NEW)

Use this recipe to profile a game built with Unreal Engine. See how you can run Intel® VTune™ Profiler within the Unreal Engine environment to profile your game.

Often, the most important factor that affects the performance of a game is the frame rate. This is the speed with which the GPU renders game graphics. However, the CPU can also impact game performance in several ways:

- Slow transfer of data to the GPU
- Slow or unnecessary operations
- Poor parallelism

With games that use the Unreal Engine (UE), much of the optimization takes place in the Unreal editor. Therefore, it is critical to understand the performance of Unreal Engine-defined tasks. The 4.19 and newer versions of Unreal Engine have the Intel® VTune™ Profiler Instrumentation and Tracing Technology (ITT) API built into the Unreal editor. This recipe demonstrates how you can run VTune Profiler to highlight UE tasks in the editor.

- INGREDIENTS
- DIRECTIONS:

  1. Build the Game in the Unreal Editor
  2. Configure Intel® VTune™ Profiler and Run Hotspots Analysis
  3. Review Results

Ingredients

Here are the hardware and software requirements for this performance recipe.

- **Application**: `Unreal Engine 4.25.4`. The sample game in this version of Unreal Engine is the Action RPG tutorial.
- **Tools**: Intel® VTune™ Profiler version 2022 - Hotspots Analysis (using User-Mode Sampling)

NOTE

- Starting with the 2020 release, Intel® VTune™ Amplifier has been renamed to Intel® VTune™ Profiler.
- Most recipes in the Intel® VTune™ Profiler Performance Analysis Cookbook are flexible. You can apply them to different versions of Intel® VTune™ Profiler. In some cases, minor adjustments may be required.
- Get the latest version of Intel® VTune™ Profiler:
  - From the Intel® VTune™ Profiler product page.
  - Download the latest standalone package from the Intel® oneAPI standalone components page.

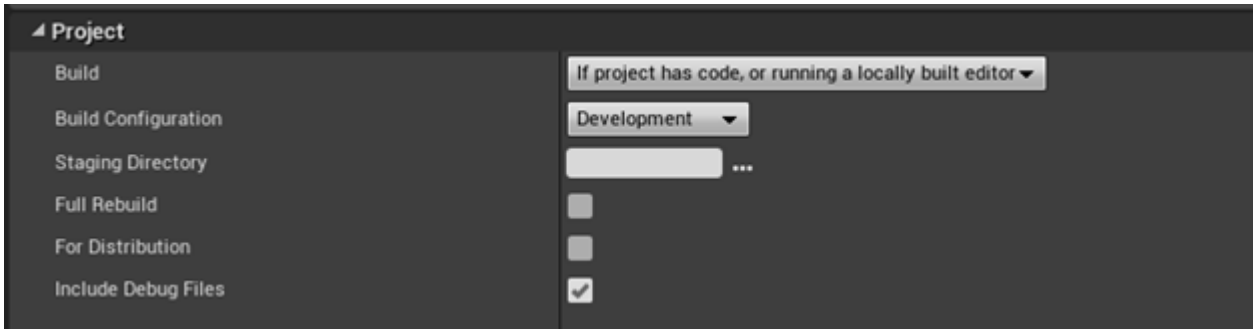
- **CPU/GPU**: 11th Generation Intel® Core™ i7-1165G7 CPU @ 2.80GHz with Intel® Iris® Xe MAX Graphics
- **Operating system**: Windows* 11 Enterprise

Build the Game in the Unreal Editor

1. Open the game in the Unreal editor.

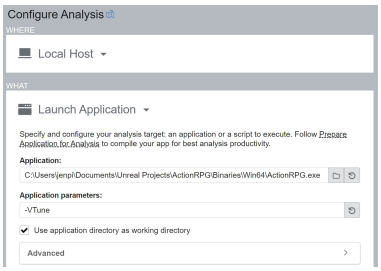


2. Build the game. Make sure to select the **Development** Build Configuration and **Include Debug Files** options.

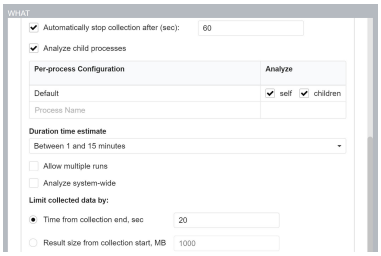


Configure Intel® VTune™ Profiler and Run Hotspots Analysis

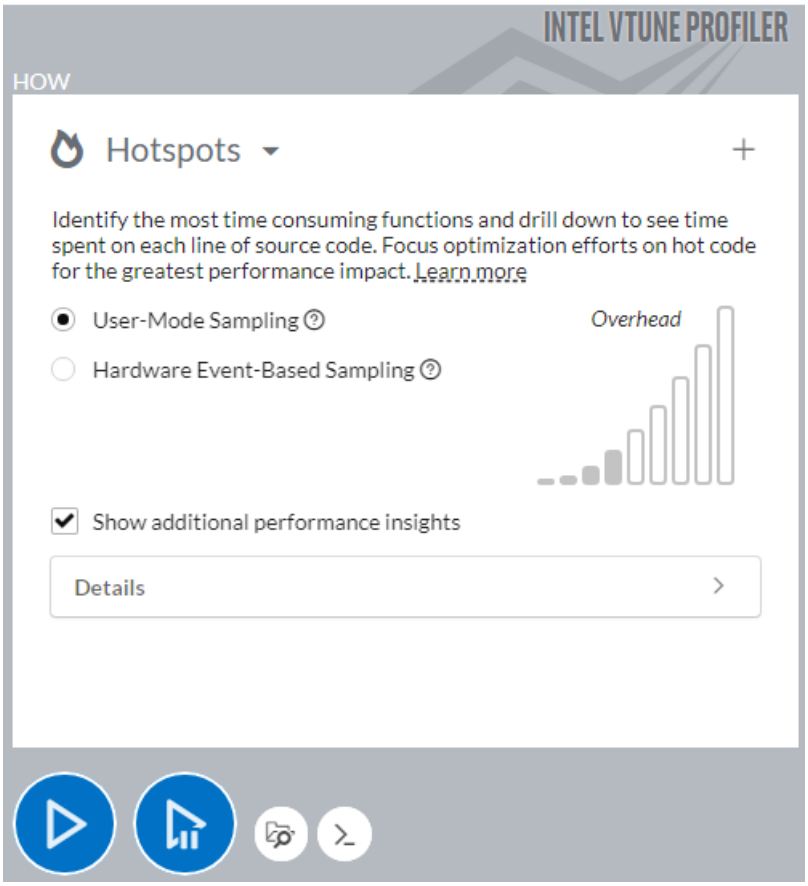
1. Open Intel® VTune™ Profiler and click **New Project** on the Welcome screen.
2. Specify a project name and a location for your project.
3. Click **Create Project**.
4. In the **Configure Analysis** window, set these options:
   - In the **WHERE** pane, select **Local Host**.
   - In the **Application** field of the **WHAT** pane, enter the path to the game executable.
   - In the **Application parameters** field, enter `-VTune`.



5. (Optional) If you want to skip profiling the start/loading phase of the game,
   - In the **WHAT** pane, open the **Advanced** section.
   - Set
   - Under **Limit collected data by**, set a value in seconds for **Time from collection end, sec**. This is the duration (in seconds) before the end of the collection for which Intel® VTune™ Profiler should retain the results. The data collected earlier than this time gets discarded.



6. In the **HOW** pane, select the **Hotspots analysis type** and enable user-mode sampling.
7. Click **Start** to run the analysis.



NOTE If you set the **Automatically resume collection after** option in step 5, only the **Start Paused** button (



) is available.

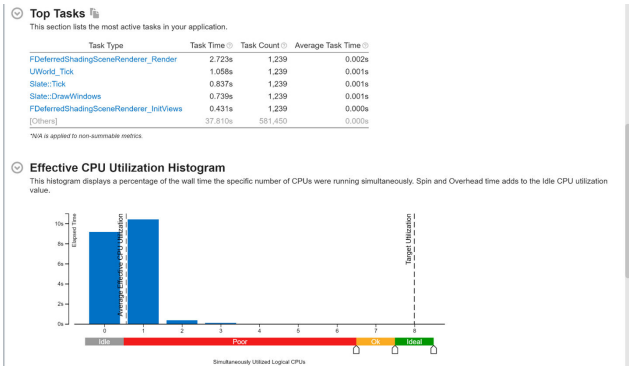
Review Results

When the data collection stops, Intel® VTune™ Profiler finalizes the results. This process may take a few minutes as Intel® VTune™ Profiler finds and resolves debug symbols.

Once results have been finalized, the **Summary** tab displays information about:

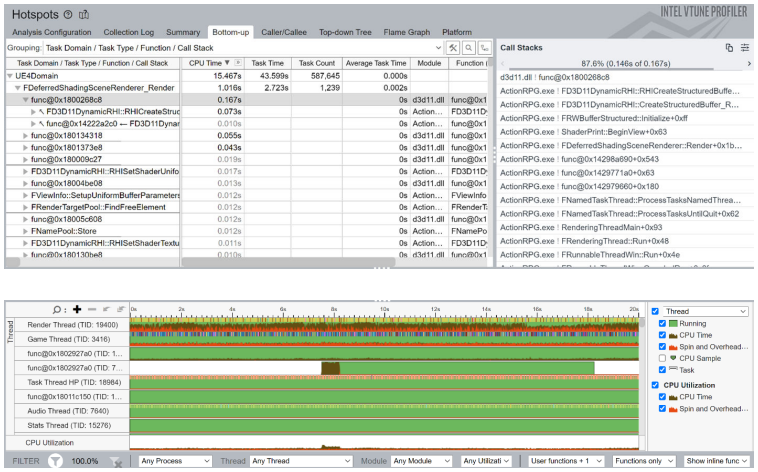
- Elapsed time

- Top hotspots
- Top Unreal Engine tasks
- Additional insights and guidance



Switch to the **Bottom-up** window to see a list of functions. The default sorting is by descending order of CPU time.

Change the grouping from **Function / Call Stack** to **Task Domain / Task Type / Function / Call Stack** to focus on Unreal Engine tasks which were identified by the Intel® VTune™ Profiler Instrumentation and Tracing Technology API (ITT API).

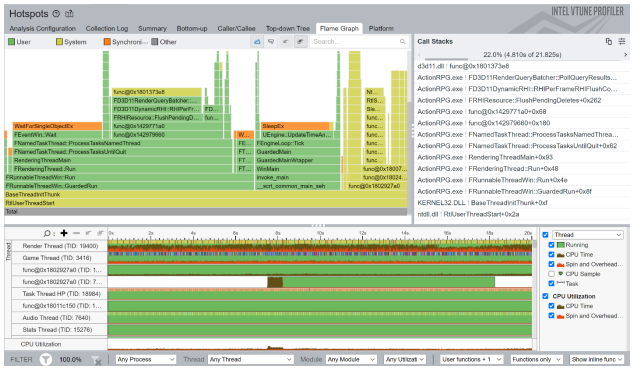


In this example, there is considerable spin time in the Render and Game threads. Also, nearly half of the overall CPU utilization was spent spinning, or waiting for a resource to return.

The CPU utilization histogram (in the **Summary** tab) tells us something important. Although this game was running over 100 threads, for the majority of the run, this game only utilized a single CPU or was mostly idle.

From here, you can try to identify whether the high amount of spin is affecting performance negatively. You can also examine how to make use of more CPU capability. Generally games are not expected to consume the full amount of hardware resources, but there is room on this system for more parallelism.

For additional insights into hotspots, open the **Flame Graph** view to see a graphical representation of call stacks from the top down.



See Also

Analyzing Hot Code Paths Using Flame Graphs

Hotspots Analysis for CPU Usage Issues

Boost CPU Performance with Intel® VTune™ Profiler

Game Tuning with Intel

Profiling Java Applications as a Remote User (NEW)

Use a wrapper script with Intel® VTune™ Profiler to profile Java applications as a remote user.

Normally, if you want to use VTune Profiler to profile a Java process that is in execution, you run a hardware-based sampling analysis and selecting **Attach to Process** as the target type. In Linux* environments, VTune Profiler uses the Linux perf tool to collect sampling data. For this purpose, you must run VTune Profiler as the same user who is running the Java process. This is because, if you run the Java process as a another user (even as root), the collector cannot attach to the Java process. However, in many cases, there is an arbitrary account that runs these applications, thus making it challenging to run them as a remote user.

In this recipe, we will see how you can use a wrapper script to run VTune Profiler and profile Java applications as a remote user.

- INGREDIENTS
- DIRECTIONS:

  1. Create the Java Application on the Remote Target
  2. Run the Java Application as a Remote User
  3. Configure VTune Profiler on the Local Machine for Remote Collection
  4. Create the Wrapper Script to Run VTune Collector
  5. Run Hotspots Analysis with Hardware Event-Based Sampling and Stack Collection
  6. Review Analysis Results

Ingredients

Here are the hardware and software tools you need for this recipe.

- **Application**: `Pi`. This Java application is used as a demo and is not available for download. The application uses a Monte Carlo algorithm to estimate the value of Pi with multiple threads.
- **Analysis Tool**: VTune Profiler version 2022 or newer - Hotspots Analysis using Hardware Event-Based Sampling (with **Collect Stacks** enabled)

NOTE

- Starting with the 2020 release, Intel® VTune™ Amplifier has been renamed to Intel® VTune™ Profiler.
- Most recipes in the Intel® VTune™ Profiler Performance Analysis Cookbook are flexible. You can apply them to different versions of Intel® VTune™ Profiler. In some cases, minor adjustments may be required.
- Get the latest version of Intel® VTune™ Profiler:
  - From the Intel® VTune™ Profiler product page.
  - Download the latest standalone package from the Intel® oneAPI standalone components page.

- **Java Development Kit (JDK) version**: OpenJDK 11.0.8 or newer
- **Local Operating System**: Windows* 10 Enterprise
- **Remote Operating System/Amazon Machine Image (AMI)**: Ubuntu Server 20.04 LTS (HVM)
- **Remote CPU/Instance Type**: AWS EC2 c5.9xlarge (Intel® microarchitecture code named Skylake with 36 logical CPUs)

Create the Java Application on the Remote Target

Prerequisite: Ensure that you have installed the JDK.

1. Create the Java file. In this example, we add an infinite loop around the body of `main (while(true))` to simulate a long-running process:

```
*Pi.java*
```

2. Compile the Java file with symbols:

```
$ javac -g Pi.java
```

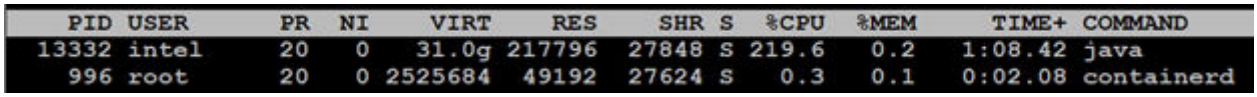
Run the Java Application as a Remote User

Start the Java application as a user other than the SSH user. In this example, the default user for the AWS instance is Ubuntu, so we create a new user (named `intel`) to run the application.

```
$ sudo adduser intel
$ su intel
$ java -Xcomp -Djava.library.path=native_lib/ia32 -cp ./ Pi
```

Verify that user `intel` is running the java process. Also note the ID of the process:

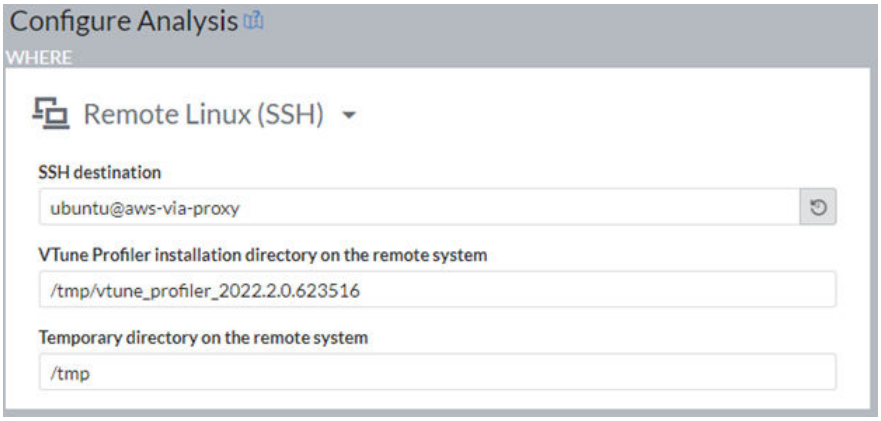
```
$ top
```



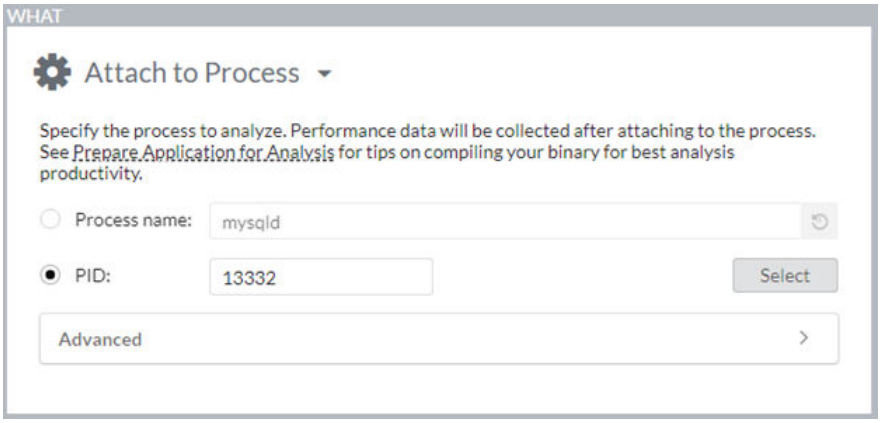
PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
13332	intel	20	0	31.0g	217796	27848	S	219.6	0.2	1:08.42	java
996	root	20	0	2525684	49192	27624	S	0.3	0.1	0:02.08	containerd

Configure VTune Profiler on the Local Machine for Remote Collection

1. Open the VTune Profiler GUI on your local machine.
2. In the **WHERE** pane, select **Remote Linux (SSH)**.
3. Configure the SSH destination for the target system with a user different from the one running the Java application. In this example, the SSH user is `Ubuntu`.
4. Deploy the VTune Profiler target package.



5. In the **WHAT** pane, select **Attach to Process**.
6. Enter the process ID for the Java application.

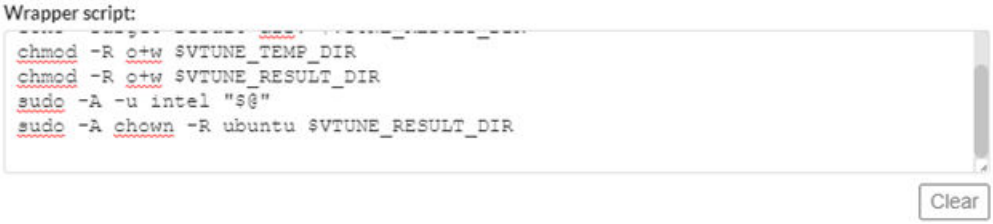


Create the Wrapper Script to Run VTune Collector

1. Use a text editor to create a file named `vtune_wrapper.sh`.
2. Populate the wrapper file with this text:

```
#!/bin/bash
echo "Target result dir: $VTUNE_RESULT_DIR"
chmod -R o+w $VTUNE_TEMP_DIR
chmod -R o+w $VTUNE_RESULT_DIR
sudo -A -u intel "$@"
sudo -A chown -R ubuntu $VTUNE_RESULT_DIR
```

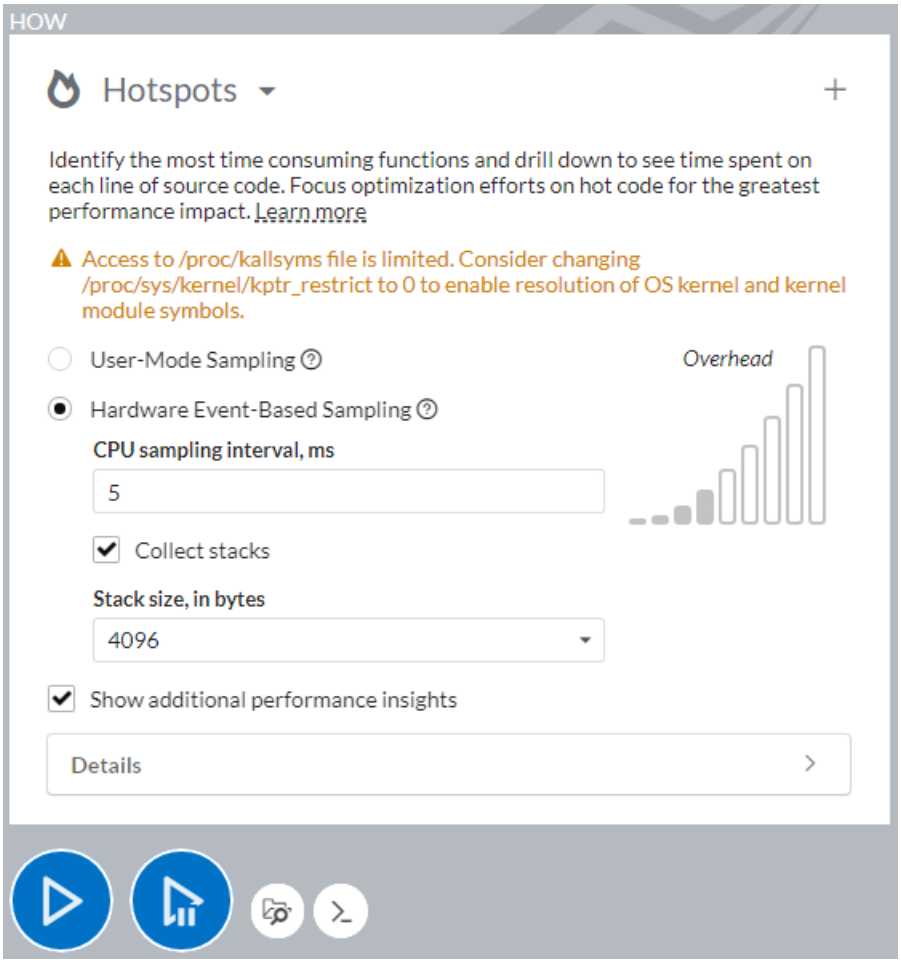
3. In the **WHAT** pane, under the **Advanced** section, scroll to the **Wrapper script** text box.
4. Select `vtune_wrapper.sh`.
5. In the text box, place the cursor at the end of the last line and hit Enter to add a line feed. This helps to ensure that the script is recognized by the collection.



6. In the **Advanced** section, scroll up and locate the option marked **Automatically stop collection after (sec)**. Enable this option and set a value of 30 to stop the collection after 30 seconds.

Run Hotspots Analysis with Hardware Event-Based Sampling and Stack Collection

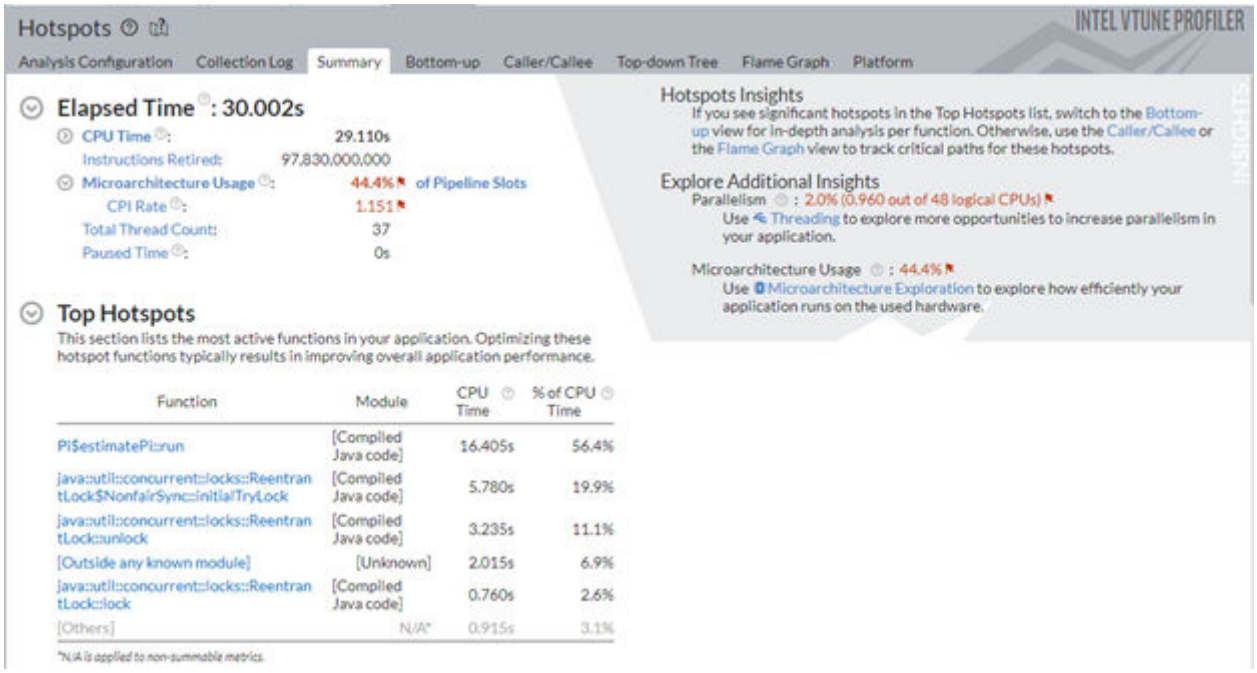
1. In the **HOW** pane, select **Hotspots** analysis type in the **Algorithm** group.
2. Enable **Hardware Event-Based Sampling**.
3. Enable the **Collect stacks** option and set the **Stack size** to 4096.



4. Click **Start** to run the analysis.

Review Analysis Results

When the collection completes, the **Summary** tab displays CPU performance information along with a list of hotspots found in the application. You can ignore warnings in the Collection Log about locating debugging information.



See Also

Profiling JavaScript* Code in Node.js* This recipe provides configuration steps to rebuild `Node.js`* and run a performance analysis of your JavaScript code using Intel® VTune™ Profiler. This analysis includes mixed-mode call stacks that contain JS frames and native frames.

Hotspots Analysis for CPU Usage Issues

Analyzing Hot Code Paths Using Flame Graphs Follow this recipe to understand how you can use Flame Graphs to detect hot spots and hot code paths in Java workloads.

Profiling JavaScript* Code in Node.js*

This recipe provides configuration steps to rebuild `Node.js` and run a performance analysis of your JavaScript code using Intel® VTune™ Profiler. This analysis includes mixed-mode call stacks that contain JS frames and native frames.*

Content Expert: Alexey Kireev

- INGREDIENTS
- DIRECTIONS:

1. Enable Support for VTune Profiler in Node.js
2. Profile JavaScript code running in Node.js

Ingredients

This section lists the hardware and software tools used for the performance analysis scenario.

- **Application**: `sample.js`. This application is used as a demo. It is not available for download.
- **JavaScript environment**: `Node.js` version 20.4.0 with Chrome* V8 version11.3.244.8
- **Performance analysis tools**:Intel® VTune™ Profiler version 2023.1 - Advanced Hotspots analysis with User-Mode Sampling

NOTE

- Starting with the 2020 release, Intel® VTune™ Amplifier has been renamed to Intel® VTune™ Profiler.
- Most recipes in the Intel® VTune™ Profiler Performance Analysis Cookbook are flexible. You can apply them to different versions of Intel® VTune™ Profiler. In some cases, minor adjustments may be required.
- Get the latest version of Intel® VTune™ Profiler:
  - From the Intel® VTune™ Profiler product page.
  - Download the latest standalone package from the Intel® oneAPI standalone components page.

- **Operating system**: Windows* 10

Enable Support for VTune Profiler in Node.js

1. Download the most recent `node.js` sources.
2. Run the `vcbuild.bat` script from the root `node-v20.4.0` folder:

```
vcbuild.bat enable-vtune
```

This script builds `Node.js` with support for VTune Profiler to profile JavaScript code.

Profile JavaScript Code Running in Node.js

This recipe uses a sample JavaScript application:

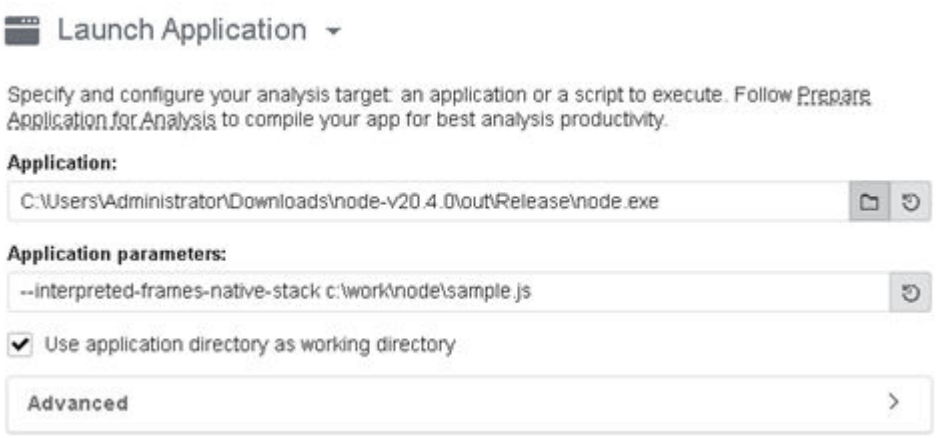
```
function say(word) {
console.log("Calculating ...");
var res = 1;
for (var i = 0; i < 40000; i++) {
  for (var j = 0; j < 40000; j++) {
    res = ( res + i * res - j )/ 2;
  }
}
console.log("Done.");
console.log(word);
}

function execute(someFunction, value) {
  someFunction(value);
}

execute(say, "Hello from Node.js!");
```

To profile this application with VTune Profiler:

1. Run VTune Profiler from the command line:

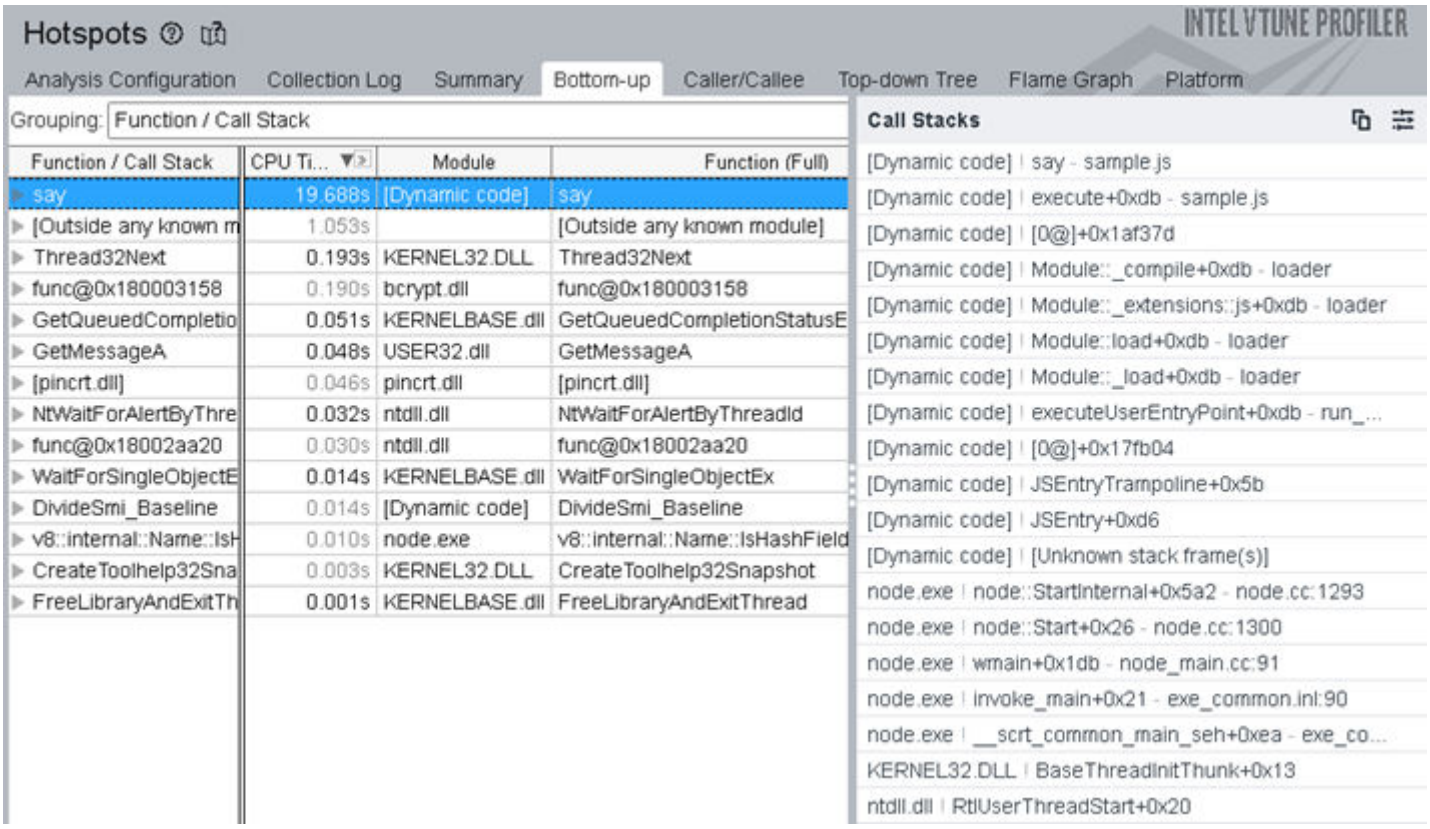
```
vtune-gui.exe
```

2. Click the **New Project** icon in the toolbar to create a new project.
3. In the **Launch Application** section,
   - Specify `node.exe` in the **Application** field
   - Set `sample.js` in the **Application parameters** field.
   - To translate frames interpreted by JavaScript, set `--interpreted-frames-native-stack` as an option in the **Application parameters** field.

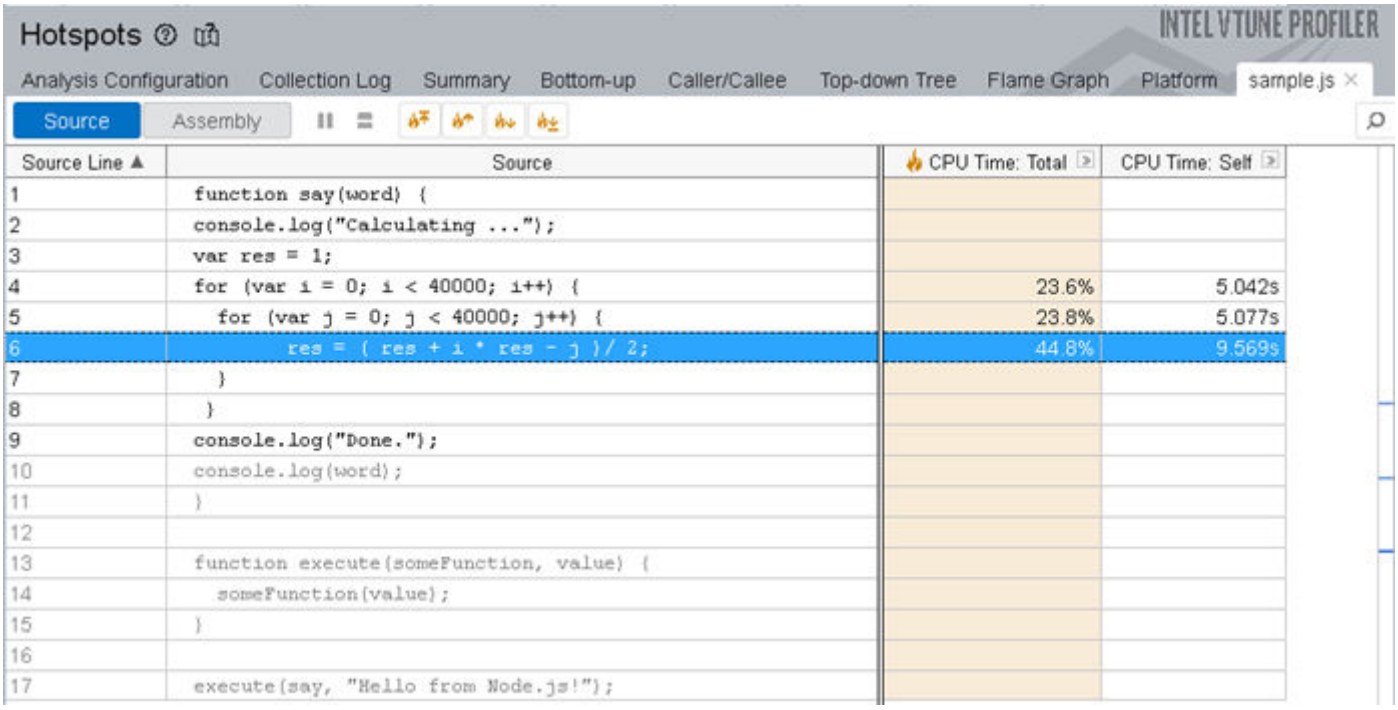


4. In the **Analysis Type** pane, select the **Hotspots** analysis type.
5. Click **Start** to run the analysis.

Once the analysis finishes, see results in the **Summary** window. We can see that the `say` function took the most CPU time to execute. Click on this function and switch to the **Bottom-up** view. See the stack flow for the hotspot:



Doubleclick on the `say` function to run a source level analysis.



Analyzing CPU and FPGA (Intel® Arria® 10 GX) Interaction

This recipe instructs you how to configure your platform to analyze an interaction of your CPU and FPGA, using Intel® Arria 10 GX FPGA as an example.

- INGREDIENTS
- DIRECTIONS:

1. Configure the Intel® Arria® 10 GX FPGA and Intel® FPGA SDK for OpenCL™
2. Build the Sample Application and Flash to the FPGA
3. Run CPU/FPGA Interaction Analysis
4. Interpret Results

Ingredients

This section lists the hardware and software tools used for the performance analysis scenario.

- **Application**: Matrix Multiplication OpenCL™ application. The Matrix Multiplication sample application is available for download from the Intel® FPGA SDK for OpenCL™ website
- **Tools**: Intel® FPGA SDK for OpenCL™, Intel® VTune™ Amplifier 2019 or higher

NOTE

- Starting with the 2020 release, Intel® VTune™ Amplifier has been renamed to Intel® VTune™ Profiler.
- Most recipes in the Intel® VTune™ Profiler Performance Analysis Cookbook are flexible. You can apply them to different versions of Intel® VTune™ Profiler. In some cases, minor adjustments may be required.
- Get the latest version of Intel® VTune™ Profiler:
  - From the Intel® VTune™ Profiler product page.
  - Download the latest standalone package from the Intel® oneAPI standalone components page.

- **Operating System**: CentOS* 7, Red Hat* Enterprise Linux 7 or higher
- **CPU**: Intel® server platform code named Skylake
- **FPGA**: Intel® Arria® 10 GX

Configure the Intel® Arria® 10 GX FPGA and Intel® FPGA SDK for OpenCL™

1. On your Intel Arria 10 GX FPGA, set up the DIP switches and connect the power and USB cables. See detailed instructions.
2. Download **Intel® FPGA SDK for OpenCL™ (includes CodeBuilder, Quartus Prime software and devices)** from FPGA Software Download Center.
3. Run the `setup_pro.sh` file to install the SDK.
4. Run `source init_opencl.sh` to set the appropriate environment variables.
5. Run `aocl version` to verify the installation. The output should look similar to the following:

```
aocl 17.1.0.240 (Intel(R) FPGA SDK for OpenCL(TM), Version 17.1.0 Build 240, Copyright (C) 2017
Intel Corporation)
```

6. Run `aocl install` to install the FPGA board.
7. Run `aocl diagnose` to verify the hardware installation. The output should look similar to the following:

```
Device Name:
acl0

Package Pat:
/home/tce/intelFPGA_pro/17.1/hld/board/a10_ref

Vendor: Intel(R) Corporation

Phys Dev Name  Status   Information

acla10_ref0   Passed   Arria 10 Reference Platform (acla10_ref0)
                       PCIe dev_id = 2494, bus:slot.func = 44:00.00, Gen3 x4
                       FPGA temperature = 44.3555 degrees C.

DIAGNOSTIC_PASSED
```

Build the Sample Application and Flash to the FPGA

1. Run `make` with the default `makefile` to build the host executable. The executable output filename is `host`.
2. Build the binary for the FPGA using the following command:

```
aoc -v -board=a10gx device/matrix_mult.cl -o bin/ matrix_mult.aocx
```

3. Set up the USB driver to flash.
   a. Run the following command:

```
sudo vim /etc/udev/rules.d/51-usbblaster.rules
```

   b. Add the following lines:

```
# usb blaster
SUBSYSTEM=="usb", ENV{DEVTYPE}=="usb_device", ATTRS{idVendor}=="09fb", ATTRS{idProduct}=="6001",
MODE="0666", NAME="bus/usb/$env{BUSNUM}/$env{DEVNUM}", RUN+="/bin/chmod 0666 %c"
SUBSYSTEM=="usb", ENV{DEVTYPE}=="usb_device", ATTRS{idVendor}=="09fb", ATTRS{idProduct}=="6002",
MODE="0666", NAME="bus/usb/$env{BUSNUM}/$env{DEVNUM}", RUN+="/bin/chmod 0666 %c"
SUBSYSTEM=="usb", ENV{DEVTYPE}=="usb_device", ATTRS{idVendor}=="09fb", ATTRS{idProduct}=="6003",
MODE="0666", NAME="bus/usb/$env{BUSNUM}/$env{DEVNUM}", RUN+="/bin/chmod 0666 %c"
SUBSYSTEM=="usb", ENV{DEVTYPE}=="usb_device", ATTRS{idVendor}=="09fb", ATTRS{idProduct}=="6010",
```

```
MODE="0666", NAME="bus/usb/$env{BUSNUM}/$env{DEVNUM}", RUN+="/bin/chmod 0666 %c"
SUBSYSTEM=="usb", ENV{DEVTYPE}=="usb_device", ATTRS{idVendor}=="09fb", ATTRS{idProduct}=="6810",
MODE="0666", NAME="bus/usb/$env{BUSNUM}/$env{DEVNUM}", RUN+="/bin/chmod 0666 %c"
```

4. Lower the JTAG clock speed to 6 MHz using the following command:

```
jtagconfig --setparam 1 JtagClock 6M
```

5. Flash the binary to the FPGA using the following command:

```
aocl flash acl0 ./bin/matrix_mult.aocx
```

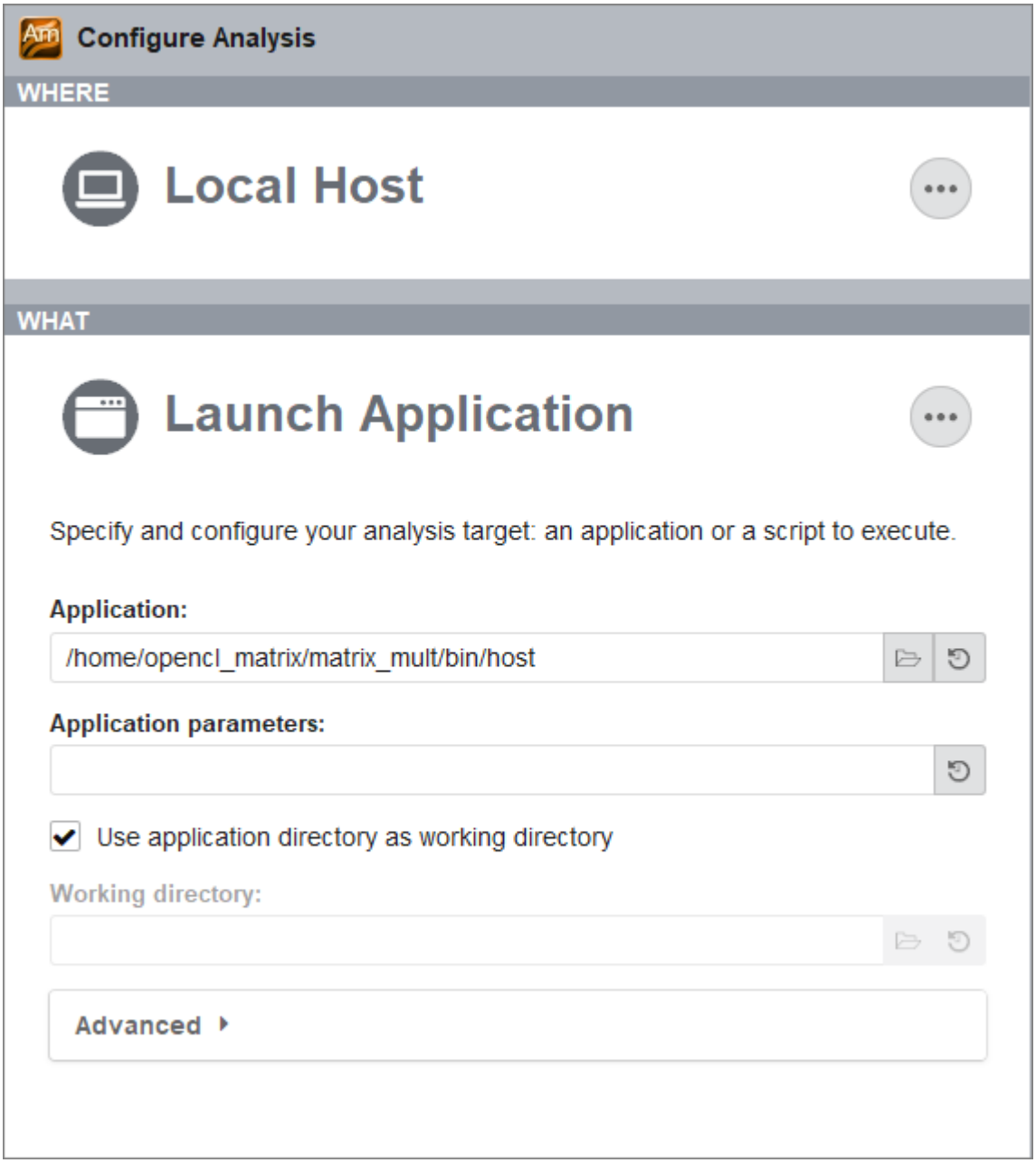
6. Reboot the host system with the FPGA.

Run CPU/FPGA Interaction Analysis

1. Launch the VTune Amplifier. For example:

```
/opt/intel/vtune_amplifier_2019/bin64/amplxe-gui
```

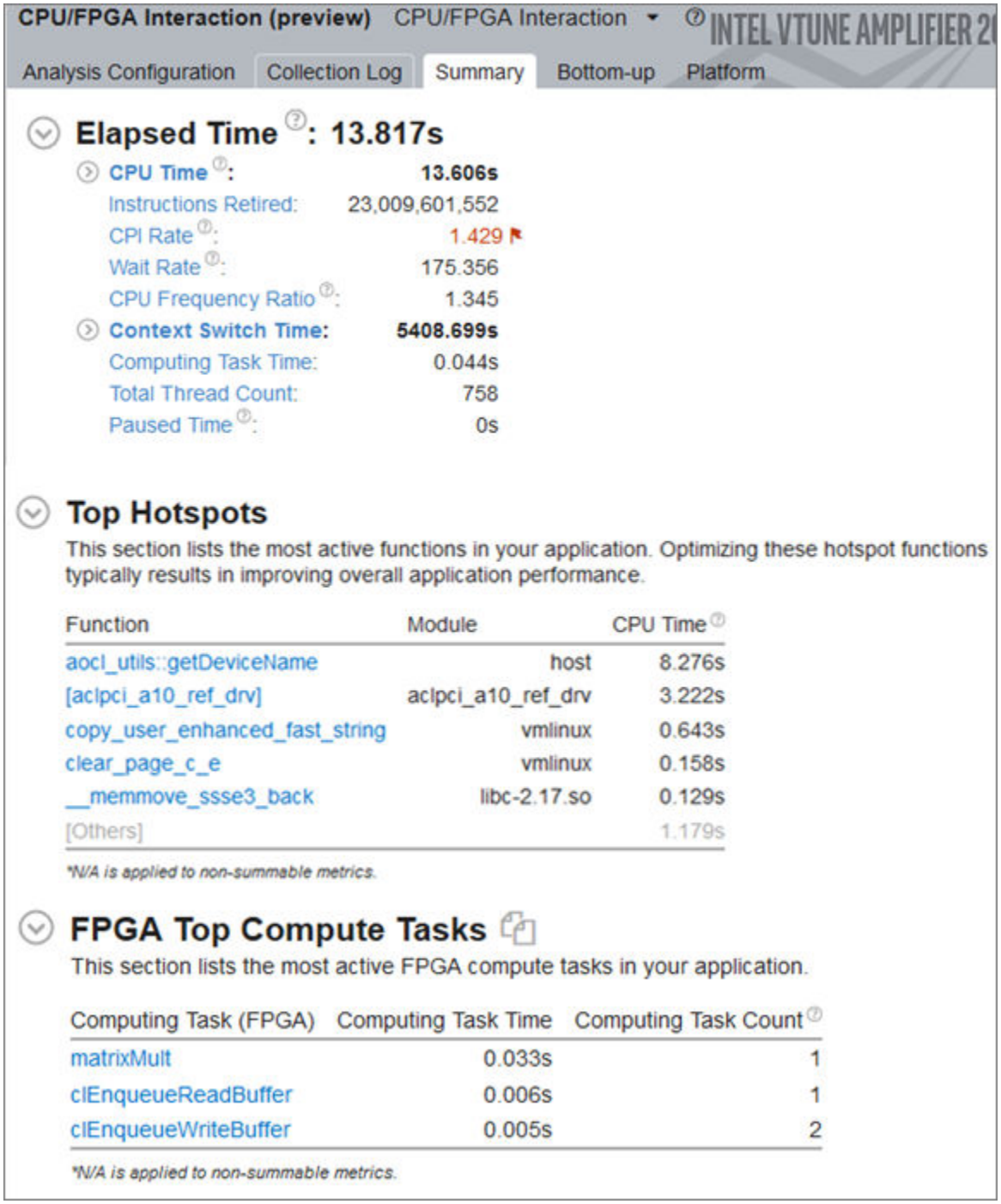
2. Create a project for your analysis, for example: `hello_world_opencl`.
3. Click **Configure Analysis** to start a new analysis.
4. Set up the **CPU/FPGA Interaction** analysis.



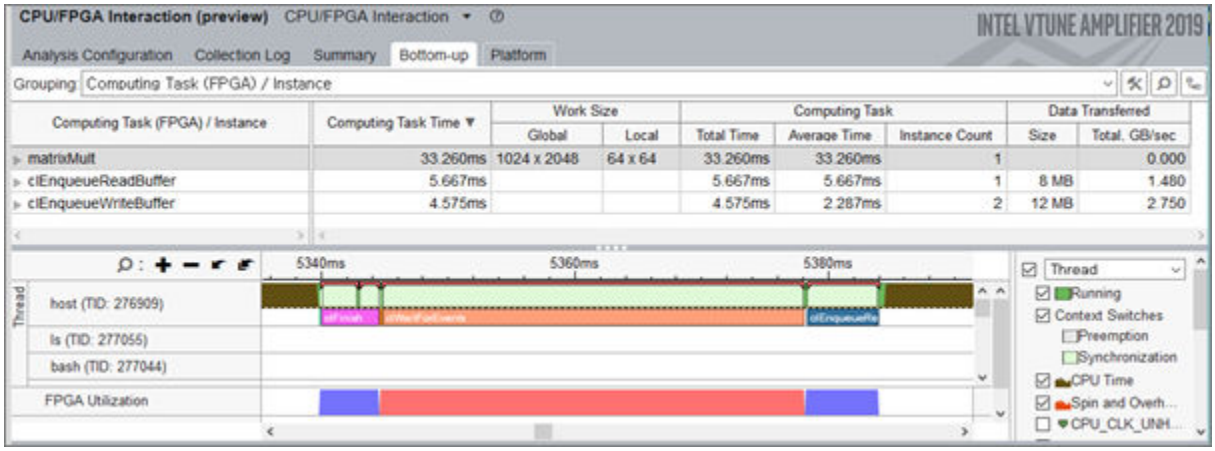
   a. In the **WHERE** pane, select **Local Host**.
   b. In the **WHAT** pane, select **Launch Application** and browse to the `hello world` application. Typically the application can be found under `<sample app>/bin/host`.
   c. In the **HOW** pane, select **CPU/FPGA Interaction** from the available analysis types.
5. Click **Start** to begin the analysis.

Interpret Results

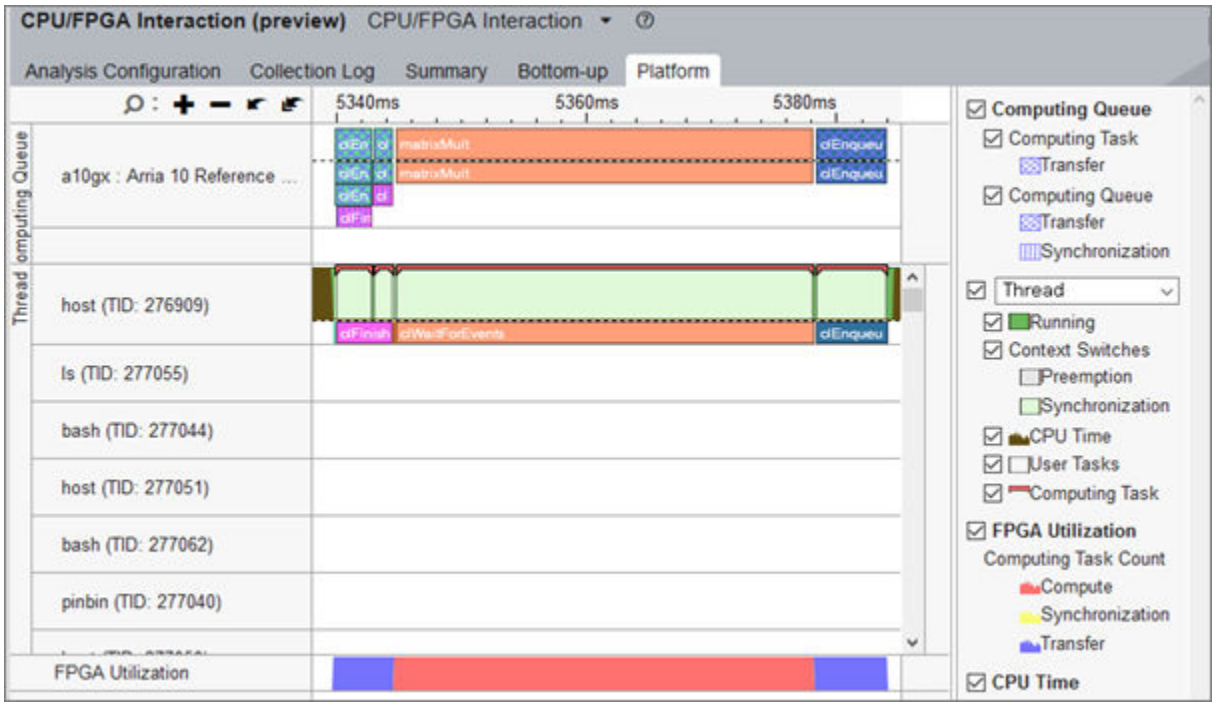
After data collection completes, the results are finalized and shown in the **CPU/FPGA Interaction** viewpoint. Start with the **Summary** tab to view the FPGA top compute tasks and well as the top tasks and hotspots for the CPU.



Switch to the **Bottom-up** tab to review the work size of a compute task and data transfer throughput. Use the timeline pane to review the FPGA utilization for compute and transfer.



Use the **Platform** tab to check the computing queue for the FPGA and host application. You can also find the start time and duration of each transfer and synchronization.



See Also

CPU/FPGA Interaction Analysis
Intel FPGA SDK for OpenCL Pro Edition: Best Practices Guide

Profiling a .NET* Core Application

Profile .NET Core dynamic code with Intel® VTune™ Profiler. Locate performance hot spots in your code and optimize as needed.

Content expert: Jennifer DiMatteo

- INGREDIENTS
- DIRECTIONS:

1. Prepare your application for analysis
2. Run Hotspots analysis
3. Identify hotspots in the managed code
4. Optimize the code with loop interchange
5. Verify the optimization

Ingredients

This section lists the hardware and software tools used for the performance analysis scenario.

- **Application:** A sample C# application that adds all the elements of an integer List. The application in this recipe is used for demonstration only and it is not available for download.
- **Tools:**
  - Intel VTune Profiler 2023 or newer
  - .NET Core 7.0 SDK
- **Operating system:** Microsoft* Windows* 11
- **CPU**: Intel microarchitecture code named Alder Lake

Prepare Your Application for Analysis

1. Open a new command window for the .NET environment variables to take effect. Make sure that you have installed .NET Core 7.0:

```
dotnet --version
```

2. Create a new `listadd` directory for the application:

```
mkdir C:\listadd
> cd C:\listadd
```

3. Enter `dotnet new console` to create a new skeleton project with the following structure:

```
C:\listadd>dir
 Volume in drive C has no label.
 Volume Serial Number is 0A4A-D42D

 Directory of C:\listadd

09/20/2017  04:09 PM    <DIR>          .
09/20/2017  04:09 PM    <DIR>          ..
09/20/2017  04:08 PM               178 listadd.csproj
09/20/2017  04:08 PM    <DIR>          obj
09/20/2017  04:08 PM               189 Program.cs
               2 File(s)            367 bytes
               3 Dir(s)  36,955,869,184 bytes free

C:\listadd>
```

4. Replace the contents of `Program.cs` in the `listadd` folder with C# code that adds the elements of an integer List:

```
using System;
using System.Linq;
using System.Collections.Generic;

namespace listadd
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Starting calculation...");
            List<int> numbers = Enumerable.Range(1,10000).ToList();
            for (int i =0; i < 100000; i ++)
            {
                ListAdd(numbers);
            }

            Console.WriteLine("Calculation complete");
        }

        static int ListAdd(List<int> candidateList)
        {
            int result = 0;
            foreach (int item in candidateList)
            {
                result += item;
            }

            return result;
        }
    }
}
```

5. Create `listadd.dll` in the `C:\listadd\bin\x64\Release\net7.0` folder:

```
dotnet build -c Release
```

6. Run the sample application:

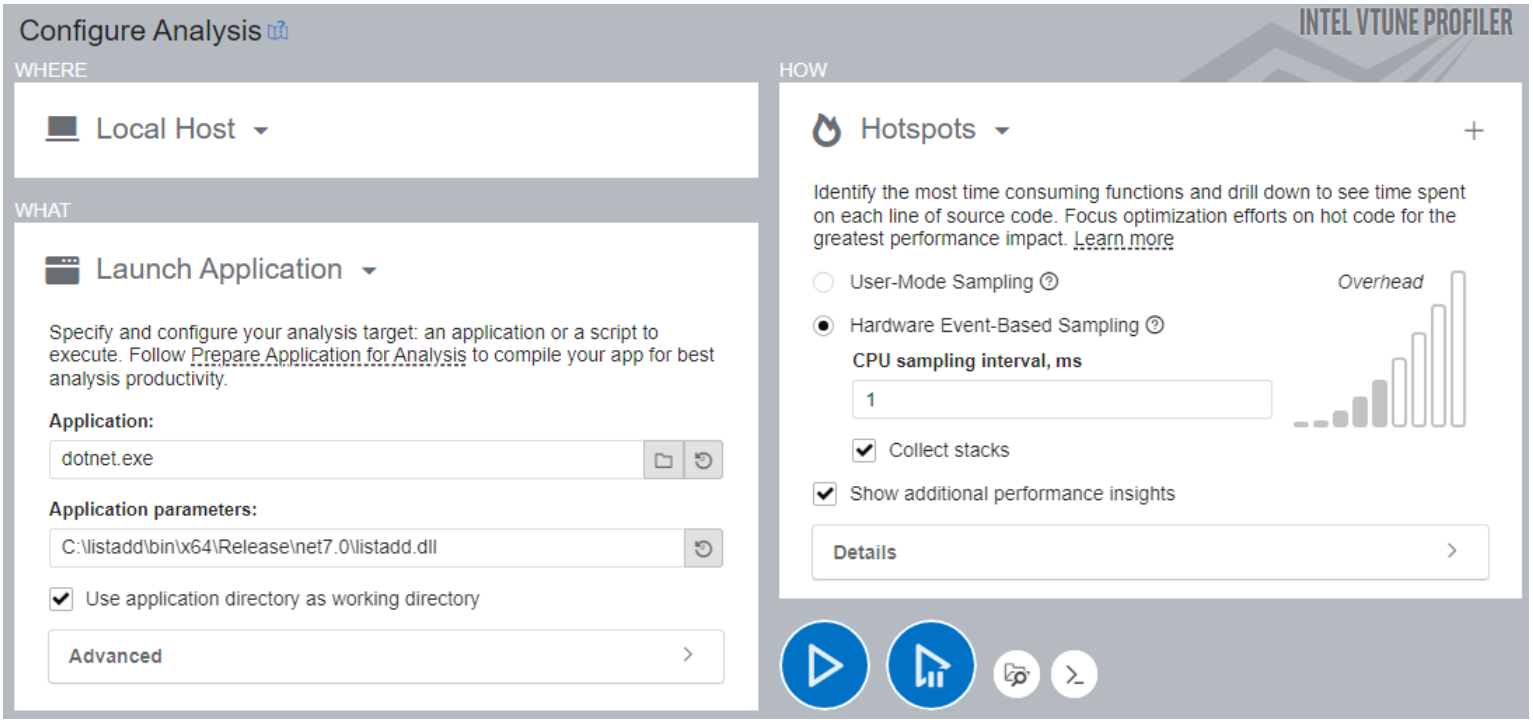
```
dotnet C:\listadd\bin\x64\Release\net7.0\listadd.dll
```

Run Hotspots Analysis

1. Run Intel VTune Profiler with administrator privileges.
2. Click the



New Project button on the toolbar. Specify a name for the new project, for example: `dotnet`.
3. In the **Analysis Target** window, select **local host** and **Launch Application** target type from the left pane.
4. In the **Launch Application** pane, specify the application to analyze:
   - **Application**: `C:\Program Files\dotnet\dotnet.exe`
   - **Application parameters**: `C:\listadd\bin\x64\Release\net7.0\listadd.dll`



NOTE
The location of `dotnet.exe` depends on your environment. Use the `where dotnet` command to find this location.

5. In the **How** pane, select **Hotspots** analysis. Make sure to enable **Hardware Event-Based Sampling** and **Collect Stacks** options.
6. Click **Start** to run the analysis.

Identify Hot Spots in the Managed Code

When the collected analysis result opens, switch to the **Bottom-up** tab. Set the data grouping level to **Process/Module/Function/Thread/Call Stack**:

Expand `dotnet.exe` > `listadd.dll` and notice that the `listadd::Program::ListAdd` function occupied most of the CPU Time:

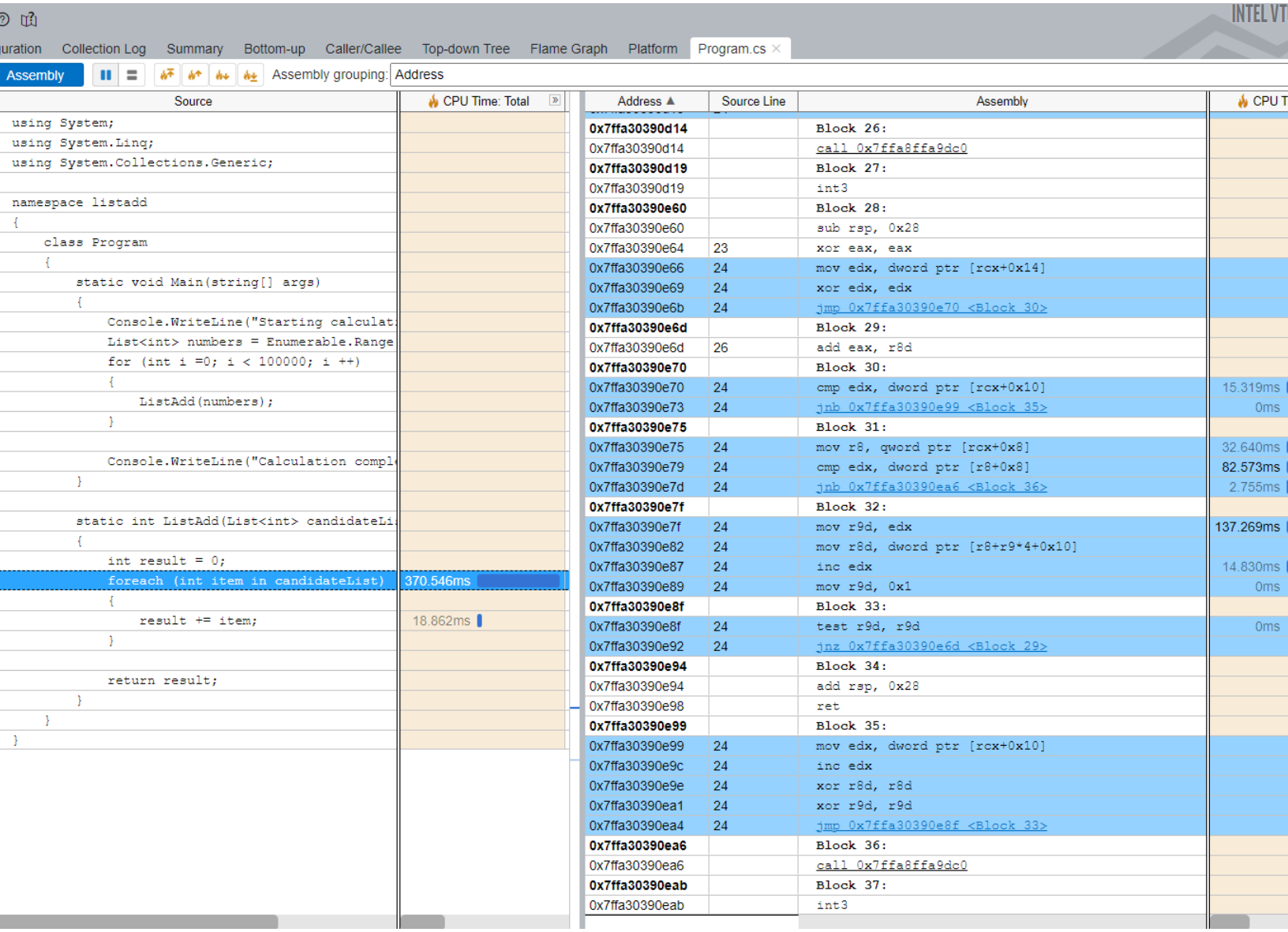
Hotspots

Analysis Configuration | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Flame Graph | Platform

Grouping: Process / Module / Function / Thread / Call Stack

Process / Module / Function / Thread / C...	CPU Time ▼	Instructions Retired	Microarchitecture Usage	Module	Function (Full)	Source File
dotnet.exe	511.004ms	14,674,102,493	69.4%			
listadd.dll	391.663ms	12,385,209,481	76.8%			
listadd::Program::ListAdd	391.663ms	12,385,209,481	76.8%	listadd.dll	listadd::Progr...	Program.cs
ntdll.dll	75.382ms	1,412,227,411	60.2%			
System.Private.CoreLib.dll	24.128ms	620,196,948	84.8%			
coreclr.dll	13.008ms	129,173,085	32.4%			
KernelBase.dll	4.567ms	124,569,774	84.2%			
amplxe_samplingmrte_clrprof_1.0.(	1.136ms	2,725,794	0.0%			
System.Linq.dll	1.119ms	0	11.0%			

Doubleclick this hot spot function to open the source view. To view the source and disassembly code side by side, click the **Assembly** toggle button on the toolbar:



Use the statistics per source line/assembly instruction to identify the most time-consuming code snippets (line 24 in the example above) and work on optimizations.

Optimize the Code with Loop Interchange

Intel VTune Profiler highlights the following code line as performance-critical:

```
foreach (int item in candidateList)
```

For optimization, consider using the `for` loop statement. Replace the contents of `Program.cs` with this C# code:

```
using System;
using System.Linq;
using System.Collections.Generic;

namespace listadd
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Starting calculation...");
            List<int> numbers = Enumerable.Range(1,10000).ToList();
            for (int i =0; i < 100000; i ++)
            {
                ListAdd(numbers);
            }

            Console.WriteLine("Calculation complete");
        }

        static int ListAdd(List<int> candidateList)
        {
            int result = 0;
            for (int i = 0; i < candidateList.Count; i++)
            {
                result += candidateList[i];
            }

            return result;
        }
    }
}
```

Verify the Optimization

To verify the optimization for the updated code, repeat the Hotspots analysis.

Prior to optimization, the sample application took 511.004 ms of CPU time:

Process / Module / Function / Thread / C...	CPU Time ▼	Instructions Retired
dotnet.exe	511.004ms	14,674,102,493
listadd.dll	391.663ms	12,385,209,481
listadd::Program::ListAdd	391.663ms	12,385,209,481

After optimization, the application ran for 430.477s, which is a 16% reduction in time over the original:

Process / Module / Function / Thread / Call Stack	CPU Time ▼	Instructions Retired
dotnet.exe	430.477ms	10,699,044,072
listadd.dll	353.307ms	9,794,911,335
listadd::Program::ListAdd	333.257ms	9,621,807,690
listadd::Program::Main	20.050ms	173,103,645

Discuss this recipe in the Analyzers community forum

NOTE

See Also

.NET Code Analysis

Profiling Applications in Amazon Web Services* (AWS) EC2 Instances

This recipe helps you set up a virtual machine (VM) instance in AWS to profile application performance with Intel® VTune™ Profiler.

Content Expert: Jennifer Dimatteo

- INGREDIENTS
- DIRECTIONS:

  1. Create and configure a virtual machine instance.
  2. Configure the instance for profiling.
  3. Run Hotspots analysis.

Ingredients

This section lists the hardware and software tools used for the performance analysis scenario.

- **Application:** Use any application of your choice.
- **Tools**: Intel® VTune™ Profiler version 2023 - Hotspots analysis

> **NOTE**
>
> - Starting with the 2020 release, Intel® VTune™ Amplifier has been renamed to Intel® VTune™ Profiler.
> - Most recipes in the Intel® VTune™ Profiler Performance Analysis Cookbook are flexible. You can apply them to different versions of Intel® VTune™ Profiler. In some cases, minor adjustments may be required.
> - Get the latest version of Intel® VTune™ Profiler:
>   - From the Intel® VTune™ Profiler product page.
>   - Download the latest standalone package from the Intel® oneAPI standalone components page.

Before You Begin

- Understand the system requirements to install VTune Profiler on your machine. Your system must have at least 4 GB of RAM and 10 GB of free disk space.
- Review the functionality of VTune Profiler available for different types of AWS instances.
- Ensure that the AWS instance can support a connection over SSH (port 22).
- The storage size for the root volume should be at least 10 GB for data collection and results.

Create and Configure a Virtual Machine Instance

Set up a virtual machine by following instructions on the AWS site.

Configure the Instance for Profiling

Prepare the target instance for profiling. by setting `/proc/sys/kernel/yama/ptrace_scope` to 0:

- For User-Mode sampling collections, set `/proc/sys/kernel/yama/ptrace_scope` to 0:

```
echo 0 | sudo tee /proc/sys/kernel/yama/ptrace_scope
```

- For Hardware Event-Based sampling collections, set `/proc/sys/kernel/perf_event_paranoid` to 0:

```
echo 0 | sudo tee /proc/sys/kernel/perf_event_paranoid
```

For more information on driverless profiling, see Profiling Hardware without Intel Sampling Drivers.

Run Hotspots Analysis

Run a Hotspots analysis using one of these methods:

- Use SSH to run a remote collection from VTune Profiler installed locally
- Run VTune Profiler directly from the AWS instance
- Run VTune Profiler as a web service on the AWS instance

Use SSH to run a remote collection from VTune Profiler installed locally:

1. (Optional) If you have a .pem key for your AWS instance, complete these steps:

   a. In your user home directory, create or open the config file:

```
<user home>\.ssh\config
```

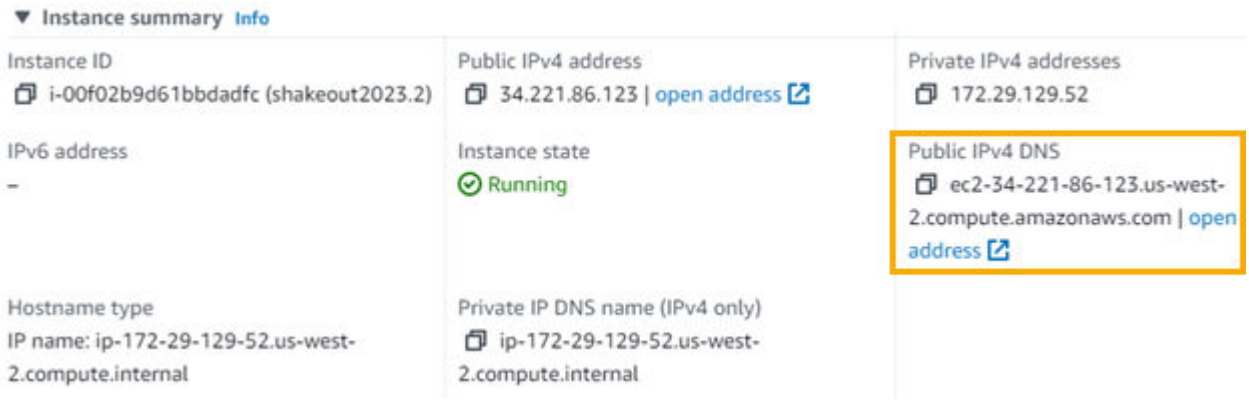
   b. Add these lines to the config file:

```
Host *.compute.amazonaws.com
User <instance user>
IdentityFile <path-to>\key.pem
```

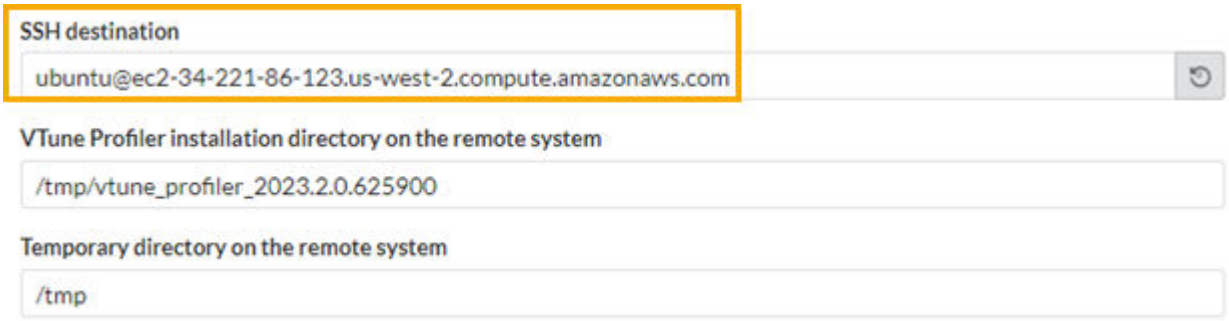
   c. If you are using a VPN where a proxy is required, include these lines:

```
LocalForward 4022 c009:22
ProxyCommand "<net connect utility>" -x proxy-server.com:1080 %h %p
```

2. Create a project in Intel® VTune™ Profiler.
3. In the **WHERE** pane of the **Configure Analysis** window, select **Remote Linux(SSH)**.
4. Locate the public IPv4 DNS address for your instance:



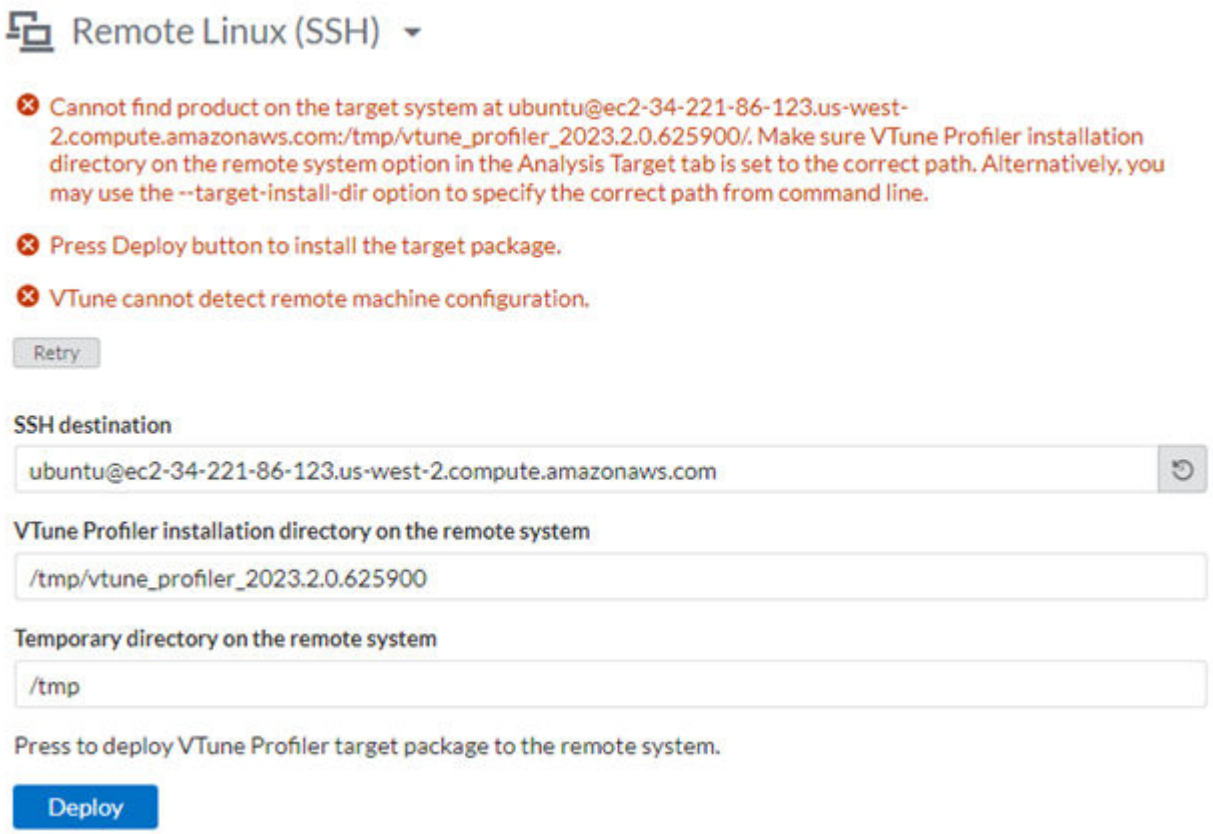
5. In the **SSH destination** field, enter `<instance user>@<public IPv4 DNS for instance>`.



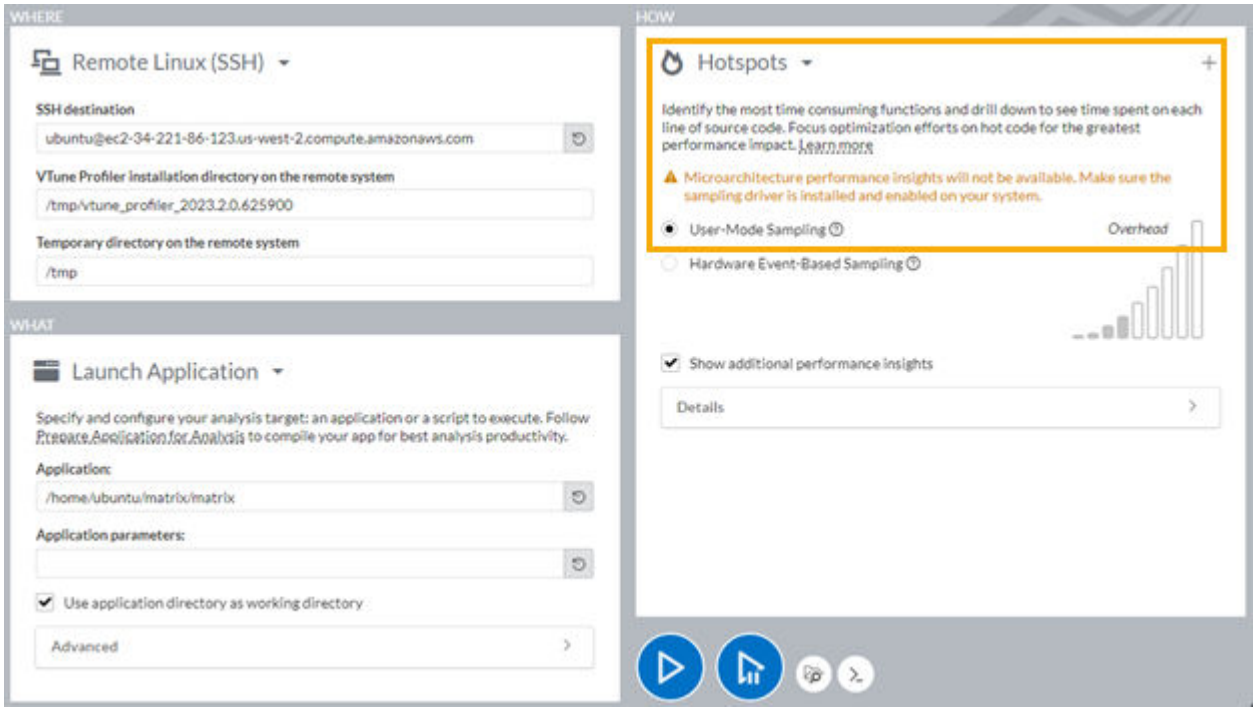
Intel® VTune™ Profiler attempts to connect to the remote system to determine if the binaries necessary for data collection have been installed.

If you updated the config file in step 1, Intel® VTune™ Profiler uses that configuration and key to connect.

Otherwise, you should be prompted to enter a password. Intel® VTune™ Profiler then creates its own SSH key and stores it in `<user home>/.ssh`. The SSH key is then copied to the `authorized_keys` file of the instance.



6. If you see a message that the product cannot be found on the target system, click **Deploy** to install.
7. In the **WHAT** pane, specify the location of your application and its working directory.
8. In the **HOW** pane, select **Hotspots** analysis with **User-Mode Sampling** collection.



9. Click

   to start the collection.

Once the analysis completes, Intel® VTune™ Profiler copies the results to your local system for analysis.

Run VTune Profiler directly from the AWS instance:

1. Install VTune Profiler by following instructions in the Installation Guide.
2. Run VTune Profiler. For example:

```
sudo <vtune_install_dir>/vtune_profiler/bin64/vtune-gui
```

3. Create a project.
4. In the **Configure Analysis** window,
   - In the **WHERE** pane, select **Local Host**.
   - In the **WHAT** pane, specify the location of your application and its working directory.
   - In the **HOW** pane, select a preferred collection mode for the Hotspots analysis. For example, select **Hardware Event-Based Sampling**.
5. Click

   to start the collection.

The analysis result opens in the **Hotspots by CPU Utilization** viewpoint.

Run VTune Profiler as a Web Service on the AWS Instance:

1. Install VTune Profiler by following instructions in the Installation Guide.
2. Run the VTune Profiler Web Service:

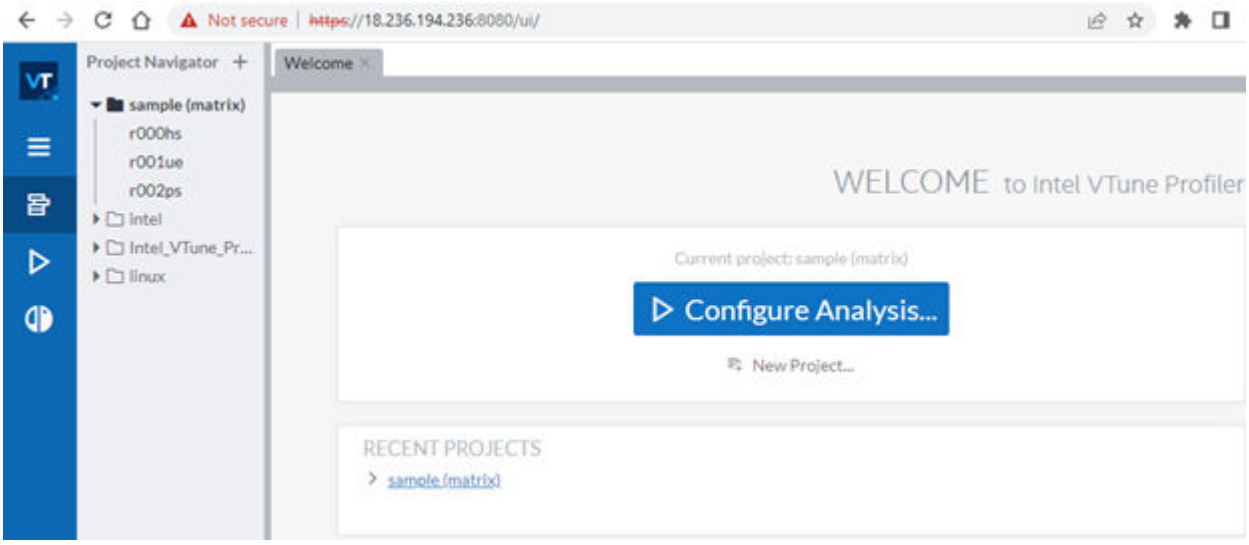
```
<vtune_install_dir>/latest/bin64/vtune-backend --allow-remote-access --enable-server-profiling --
web-port=8080
```

   The `vtune-backend` command returns a URL with the private IP for the instance as well as a one-time token.

3. Replace the private IP with the public IP or DNS for the instance. For example, `https://172.29.129.54:8080/?one-time-token=b8cafc89721e781161aba4ddcef5a718` becomes `https://18.236.194.236:8080/?one-time-token=b8cafc89721e781161aba4ddcef5a718`.
4. Copy the URL into your browser.

> **NOTE** The browser may display a warning indicating that the URL might be unsafe. This is because the web service is using a self-signed certificate instead of an officially signed certificate.

5. Create a passphrase at the prompt and proceed to the VTune GUI.



6. Select **Add new remote target**.
7. In the **Configure Analysis** window,
   - In the **WHERE** pane, select **Local Host**.
   - In the **WHAT** pane, specify the location of your application and its working directory.
   - In the **HOW** pane, select a preferred collection mode for the Hotspots analysis. For example, select **Hardware Event-Based Sampling**.
8. Click

   to start the collection.

The analysis result opens in the **Hotspots by CPU Utilization** viewpoint.

See Also

Targets in Virtualized Environments

Analyze Performance

Configure SSH Access for Remote Collection

Enabling Performance Profiling in GitLab* CI

This recipe helps you integrate Intel® VTune™ Profiler into your GitLab CI pipeline to profile your builds on-the-fly.*

This recipe demonstrates how you can enable unattended, automated profiling of your builds by including VTune Profiler into your GitLab Continuous Integration pipeline, and how to make access to the latest performance analysis data more convenient by using a static HTML feature of VTune Profiler.

NOTE
See more integration examples for different Intel software products and CI systems in the oneapi-ci GitHub* repository.

This approach offers the following advantages:

- **Automatic performance analysis:** If a performance regression was introduced into the build, setting up a test environment and collecting performance results manually can be monotonous and consumes valuable time of the engineers. Integration of VTune Profiler into the testing stage of your GitLab CI pipeline allows you to automatically collect performance data in a predetermined environment and to upload the results automatically as an artifact.

  This eliminates manual work and lets you focus on determining the cause of the regression by making the performance results data readily available once the build is complete.
- **Customized configuration:** The Command-Line Interface (CLI) capabilities of VTune Profiler allow for great flexibility in terms of CI integration. Using the CLI of VTune Profiler, you can set up a customized process that suits your team, for example, by selecting the necessary analysis types and parameters.

  Some examples of possible configurations are:

  - Automatically run a basic Hotspots analysis for each build
  - Profile a new build only when a performance regression was detected at the load testing stage of your build system and collect performance data for all the necessary analysis types
- **HTML analysis reports:** VTune Profiler offers a command line option that generates a static HTML page with the summary for the collected result. This allows you to view the HTML page using your browser and to determine whether additional analysis is needed. Optionally, you can host these HTML reports as GitLab Pages for convenience.

- INGREDIENTS
- DIRECTIONS:

  1. Install GitLab Runner.
  2. Configure automatic data collection.
  3. Configure an automatic upload of your results as artifacts.
  4. View the results data.
  5. (Optional) Resolve newly introduced issues.

INGREDIENTS

This section lists the software tools used for the performance analysis scenario.

- **Infrastructure:** GitLab repository with a pre-configured GitLab CI pipeline, which includes:
  - Makefile for your project
  - Pre-populated `.gitlab-ci.yml` file
  - GitLab Runner with VTune Profiler installed
- **Tools:** Intel® VTune™ Profiler 2020

NOTE

- Starting with the 2020 release, Intel® VTune™ Amplifier has been renamed to Intel® VTune™ Profiler.
- Most recipes in the Intel® VTune™ Profiler Performance Analysis Cookbook are flexible. You can apply them to different versions of Intel® VTune™ Profiler. In some cases, minor adjustments may be required.
- Get the latest version of Intel® VTune™ Profiler:
  - From the Intel® VTune™ Profiler product page.
  - Download the latest standalone package from the Intel® oneAPI standalone components page.

Install GitLab Runner

If you do not have a configured GitLab Runner, to install and configure the GitLab Runner software package, follow the instructions provided in the official GitLab documentation available at https://docs.gitlab.com/runner/#install-gitlab-runner.

After installing GitLab Runner, install VTune Profiler using your preferred method. For instructions and available installation methods, see the VTune Profiler Install Guide.

NOTE

- Make sure to set `/proc/sys/kernel/perf_event_paranoid` to 0 to allow hardware event-based sampling collection without root privileges. For more information on profiling without Intel sampling drivers or root privileges, see the Profiling Hardware Without Intel Sampling Drivers Cookbook recipe.
- The performance analysis result may vary depending on the Runner selection and the individual machine running the analysis.

Configure Automatic Data Collection

Add the `vtune` command calls and the artifact handling commands to your GitLab pipeline. For example, you can add the following command to your `.gitlab-ci.yml` file:

```
vtune -collect hotspots ./<your_application>
```

This command launches the application specified by the last command option and collects Hotspots analysis data. It also stores the results data in a separate directory.

You can upload any combination of VTune Profiler results and summary HTML pages. For example, you can upload the full analysis result and the static HTML summary page for a quick overview of the performance of your application.

To generate a static HTML page, include the following command:

```
vtune -report summary -format=html > hotspots_summary.html
```

To upload the full VTune Profiler result as an artifact, it is necessary to package the result directory with your tool of choice. For example, to package the result data with `tar`:

```
tar -c r00* > vtune_result.tar
```

NOTE

- Using the `vtune` command, you can specify any options and select any analysis types that are valid for your environment. For more information on how to run an analysis from the command line, see the Run Command Line Analysis page of the online User Guide.
- The graphical user interface of VTune Profiler offers a Command Line Configuration Generation feature, which allows you to conveniently pre-configure an analysis in the GUI and to instantly generate and copy a command that includes all the options and parameters that are necessary for your preferred analysis configuration. You can use this feature to quickly generate a command for later use.

For more information on manually creating a `vtune` command, see the Command Line Interface chapter of the online User Guide.

NOTE
You may encounter limitations regarding the type of analysis you can run in a certain environment. For example, the Microarchitecture Exploration analysis type is not available under certain virtual machine hypervisors.

Configure an Automatic Upload of your Results as Artifacts

Add the files that you wish to upload to the `artifacts/paths:` section of your `.gitlab-ci.yml` file. For example, to upload both the HTML summary and a `.tar` file with the result directory:

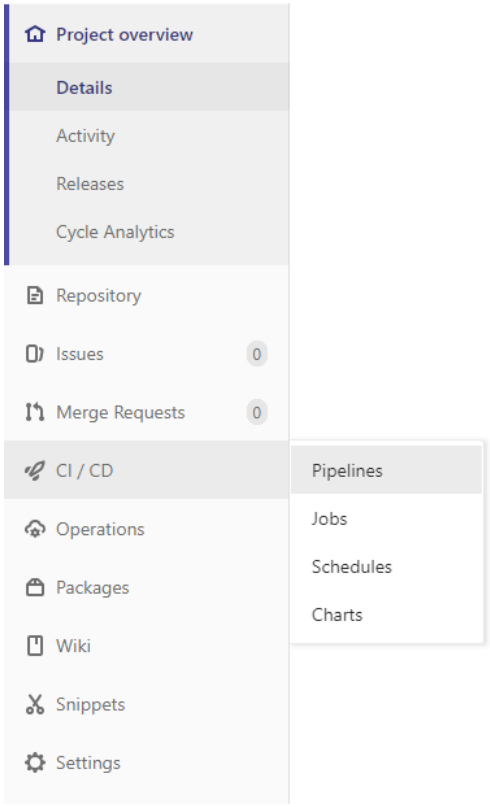
```
artifacts:
    paths:
        - <relative-path-to-project>/vtune_result.tar
        - <relative-path-to-project>/hotspots_summary.html
```

NOTE
Make sure to upload your result as a GitLab artifact. Otherwise, the results are still saved on the machine running VTune Profiler, but you will have to retrieve them manually.

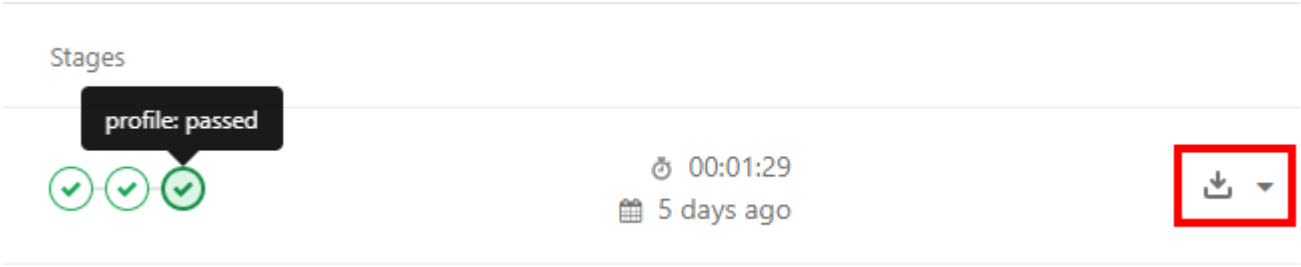
View the Results Data

Once a build is complete, you can use the GitLab web interface to quickly access the analysis summary page. To do this:

1. On GitLab, navigate to the Pipelines page.

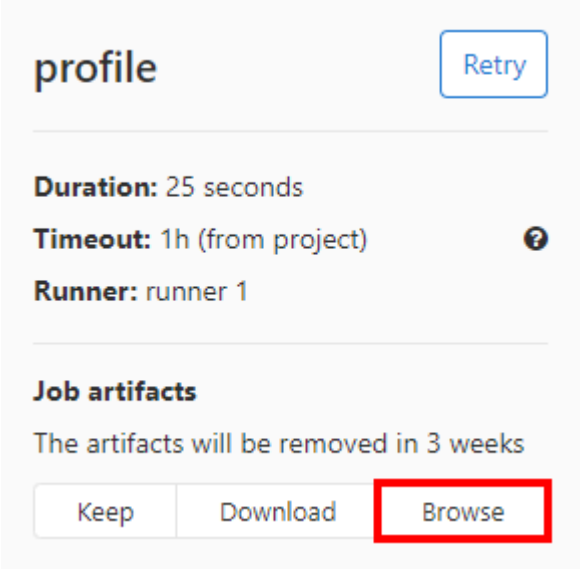


2. From this page, you can either download the entire artifact bundle, or browse to the HTML page separately and determine whether downloading the precollected result is necessary based on the summary.



To download the HTML page separately:

1. Navigate to the pipeline stage report page and click **Browse**.



2. Navigate to the location of the HTML page and download it.



3. Open the summary HTML in your browser of choice and determine whether additional analysis is needed.

Intel® VTune™ Profiler 2020

Elapsed Time:	1.652s
CPU Time:	1.070s
Effective Time:	1.070s
Idle:	0.010s
Poor:	1.060s
Ok:	0s
Ideal:	0s
Over:	0s
Spin Time:	0s
Overhead Time:	0s
Total Thread Count:	3
Paused Time:	0s

Top Hotspots:

Function	Module	CPU Time
memcmp	libc-dynamic.so	0.084s
OS_BARESYSCALL_DoCallAsmIntel64Linux	libc-dynamic.so	0.082s
[Outside any known module]	[Unknown]	0.073s
llvm::StringMapImpl::LookupBucketFor	libpin3dwarf.so	0.056s
std::priv::_Rb_tree<std::pair<std::string, unsigned long>, std::less<std::pair<std::string, unsigned long>>, std::pair<std::string, unsigned long>, std::priv::_Identity<std::pair<std::string, unsigned long>>, std::priv::_SetTraitsT<std::pair<std::string, unsigned long>>, std::allocator<std::pair<std::string, unsigned long>>>::insert_unique	libtpsstool.so	0.052s
[Others]	N/A	0.722s

Effective Physical Core Utilization: 21.5% (0.861 out of 4)

```
The metric value is low, which may signal a poor physical CPU cores
utilization caused by:
    - load imbalance
    - threading runtime overhead
    - contended synchronization
    - thread/process underutilization
    - incorrect affinity that utilizes logical cores instead of physical
cores
Explore sub-metrics to estimate the efficiency of MPI and OpenMP
parallelism or run the Locks and Waits analysis to identify parallel
bottlenecks for other parallel runtimes.
```

To view the results data using the graphical user interface of VTune Profiler:

1. Extract the results files from the `.tar` archives.
2. Launch the VTune Profiler GUI.
3. Click the



Open Result button and browse to the result file. The result data will open in a new tab.

(Optional) Resolve Newly Introduced Issues

If you detect a performance regression, you can use other analysis types offered by VTune Profiler to help you identify and resolve the issues.

NOTE
To discuss this recipe, visit the VTune Profiler developer forum.

See Also

VTune Profiler Command Line Interface

VTune Profiler CLI Reference

Command Line Configuration Generation Feature

Configuring a Hyper-V* Virtual Machine for Hardware-Based Hotspots Analysis

This recipe helps you set up a Virtual Machine instance in the Hyper-V environment for hardware performance profiling with Intel® VTune™ Profiler.

Content Expert: Alexey Kireev

- INGREDIENTS
- DIRECTIONS:

1. Configure a Hyper-V host.
2. Create and configure a Virtual Machine.
3. Configure the Virtual Machine for the hardware analysis.
4. Run Hotspots analysis (Hardware Event-Based Sampling mode).

Ingredients

This section lists the hardware and software tools used for the performance analysis scenario.

- **Tools:**
  - Intel® VTune™ Profiler (or any release of the Intel® VTune™ Amplifier 2019)

    NOTE
    - Starting with the 2020 release, Intel® VTune™ Amplifier has been renamed to Intel® VTune™ Profiler.
    - Most recipes in the Intel® VTune™ Profiler Performance Analysis Cookbook are flexible. You can apply them to different versions of Intel® VTune™ Profiler. In some cases, minor adjustments may be required.
    - Get the latest version of Intel® VTune™ Profiler:
      - From the Intel® VTune™ Profiler product page.
      - Download the latest standalone package from the Intel® oneAPI standalone components page.

  - A third-party virtual machine manager (VMM) may also be required.
- **CPU**: Intel® Xeon® processor that supports Intel® Virtualization Technology (Intel® VT)

  You can also use any Intel microarchitecture with performance monitoring hardware, such as:

Intel Microarchitecture code name	PMU Version	PEBS	LBR	IPT	PT2GPA
Kaby Lake	4	Yes	Yes	Yes	No
Ice Lake	4	Yes	Yes	Yes	Yes
Cascade Lake	4	Yes	Yes	Yes	No
Gemini Lake	4	Yes	Yes	Yes	No
Skylake	4	Yes	Yes	No	No
Haswell	3	Yes	Yes	No	No
Broadwell	3	Yes	Yes	No	No

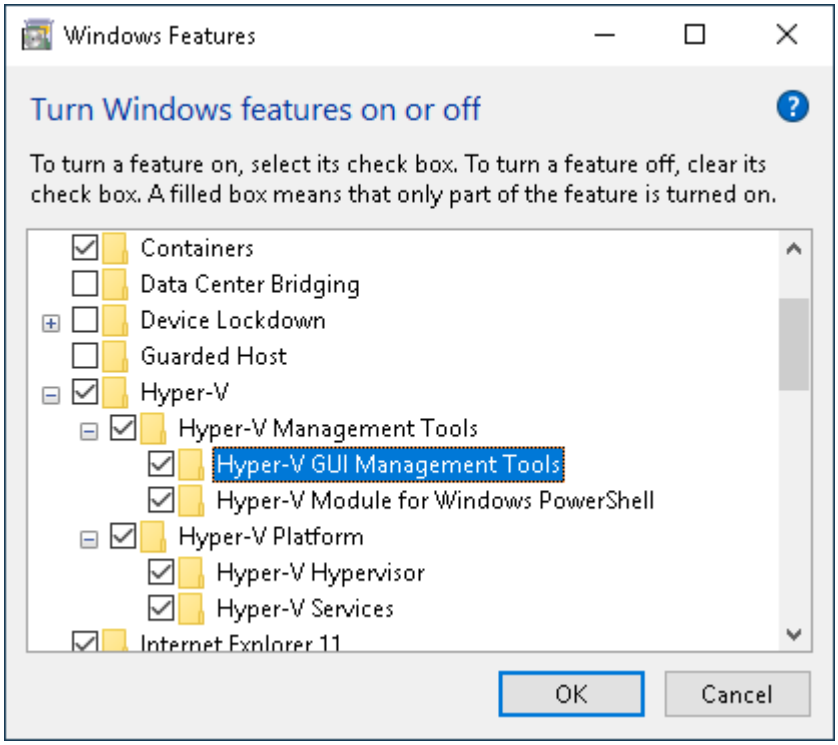
- **Operating system:** Microsoft* Windows Server 2019 or Windows 10 Version 1809 (October 2018 Update) or later
- **BIOS** with Intel VT support

> **NOTE** Hyper-V may impose additional requirements described in the Enable Intel Performance Monitoring Hardware in a Hyper-V virtual machine article.

Configure a Hyper-V Host

1. Enable the Intel VT in the BIOS setup on your server:
   a. Access the system BIOS by pressing **F2** during the system power-on self test (POST).
   b. Click **Advanced** > **Processor Configuration**.
   c. Select **Intel® Virtualization Technology** and change to **Enabled**.
   d. Do an AC power cycle for changes to take effect.
2. Go to **Control Panel** > **Programs** > **Programs and Features** and click **Turn Windows features on or off** on the left pane.

   The **Windows Features** dialog box opens.
3. Expand the **Hyper-V** feature and select all the check boxes:



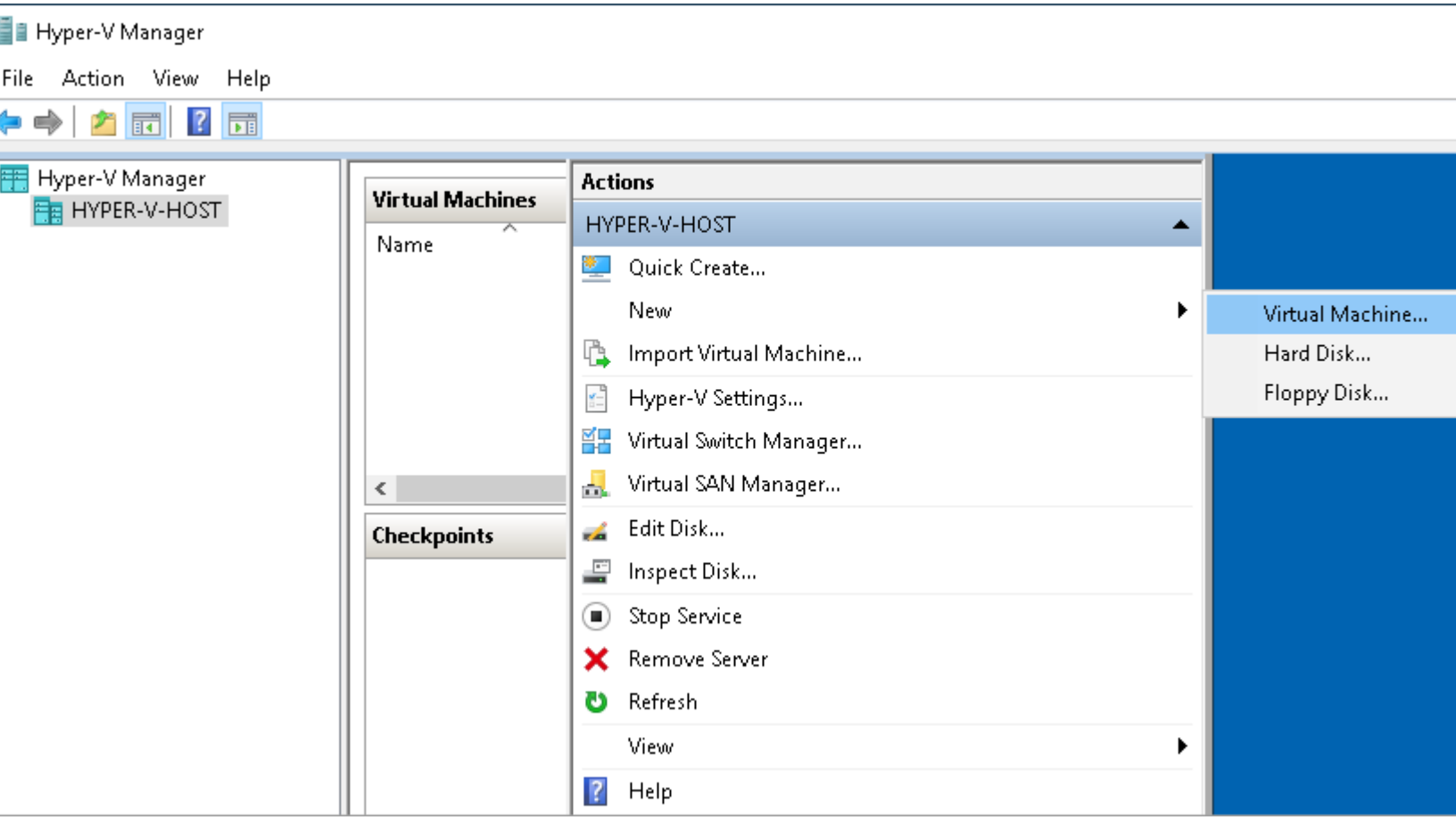
4. Click **OK** and proceed with the installation.

   When the installation is complete, click **Restart now**.

Create and Configure a Virtual Machine

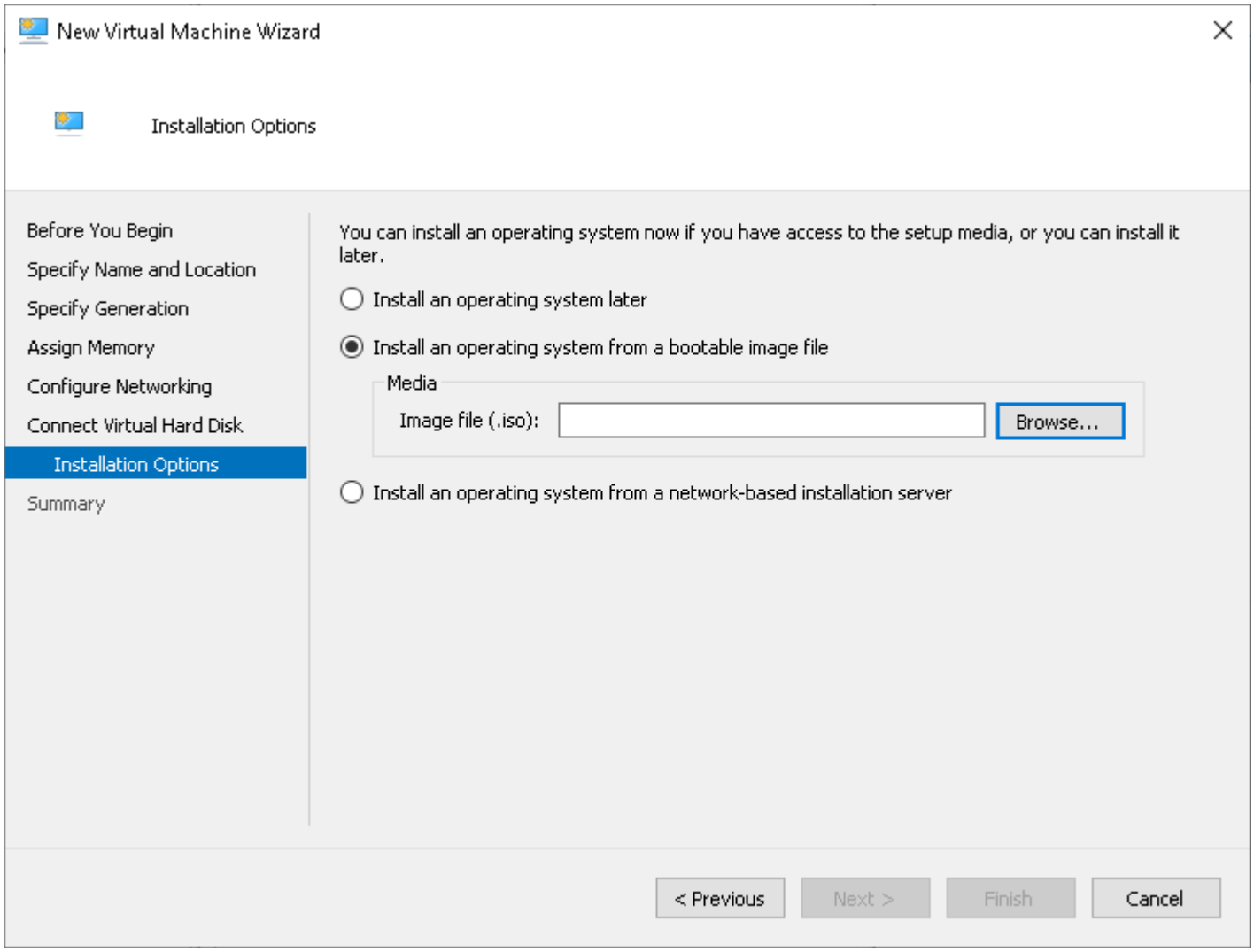
On your Hyper-V host (further, *host*), run Hyper-V Manager from the **Start** menu to create and configure a new VM (further, *VM*):

1. From the **Actions** panel, click **New** and select **Virtual Machine...** :



2. In the **New Virtual Machine Wizard**, specify all necessary parameters for the new machine (CPU configuration, Memory, Network, Hard Disk, and others).

   Recommended system requirements for the VTune Profiler are 4 GB RAM and at least 10 GB free disk space.
3. Configure your installation source. For example, you can use local installation `.iso` image:



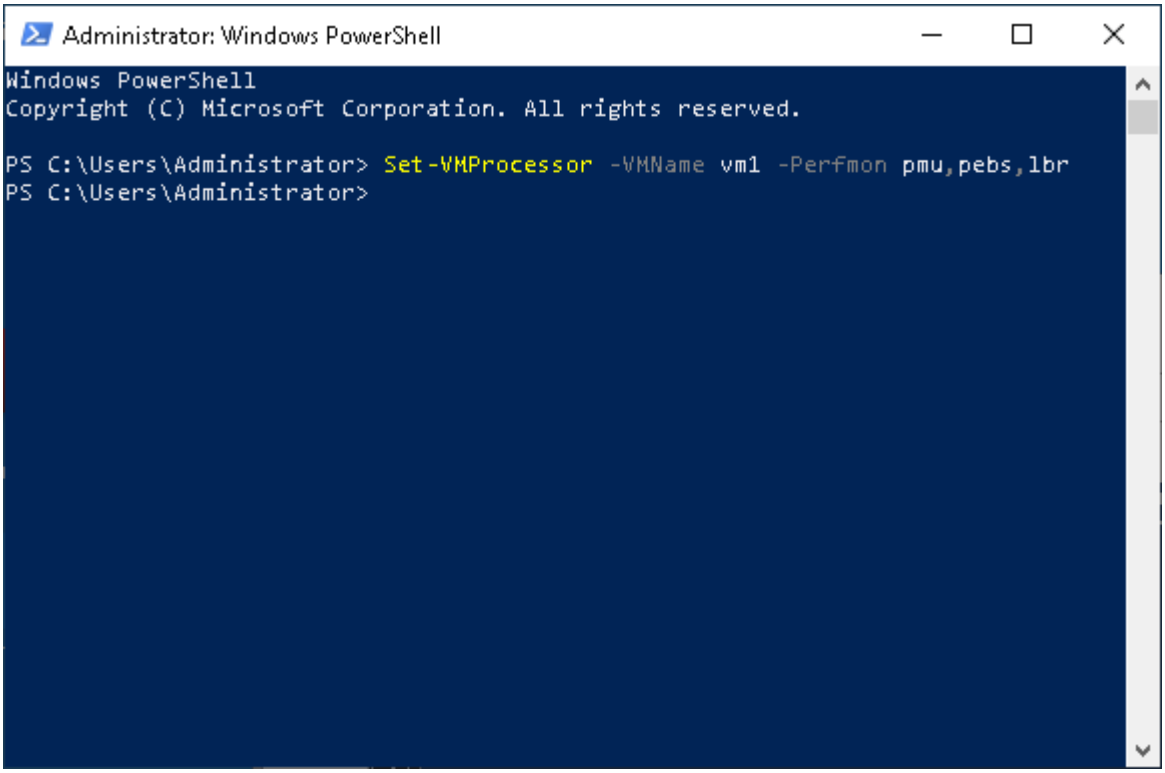
4. Start the newly created VM and proceed with the OS installation process.

Configure the Virtual Machine for the Hardware Analysis

Prepare a target system instance for profiling by exposing PMU/PEBS/LBR to the VM:

1. Turn off the newly created VM and launch your host Windows PowerShell as an Administrator from the **Start** menu.
2. In the PowerShell, configure your VM to expose PMU, PEBS, LBR to the VM as follows:

```
Set-VMProcessor -VMName your_vm_name -Perfmon pmu,pebs,lbr
```



3. Turn on your VM.

> **NOTE** For additional guidance, see the Enable Intel Performance Monitoring Hardware in a Hyper-V virtual machine article.

Run Hotspots Analysis

Run the VTune Profiler directly from the VM:

1. Install the VTune Profiler on the VM.
2. Run the VTune Profiler from **Start** menu.
3. Create a project.

   The **Configure Analysis** window opens.
4. On the **WHERE** pane, select **Local Host**.
5. On the **WHAT** pane, specify the location of your application and its working directory.
6. On the **HOW** pane, select the **Hardware Event-Based Sampling** collection mode for the Hotspots analysis.

   If you succeed with exposing PMU/PEBS/LBR, your analysis configuration pane will show no error messages and you will be able to perform the analysis.
7. Click

   Start to run the analysis.

   The analysis result opens in the default **Hotspots by CPU Utilization** viewpoint.

> **NOTE**
> You may perform the Hotspots analysis in multiple VMs running simultaneously.

See Also

Profile Targets in the Hyper-V* Environment

Enable Intel Performance Monitoring Hardware in a Hyper-V virtual machine

Hardware Event-based Sampling Collection

Profiling an Application for Performance Anomalies (NEW)

This recipe describes how you can use the Anomaly Detection analysis type in Intel® VTune™ Profiler to identify performance anomalies that could result from several factors. The recipe also includes some suggestions to help you fix these anomalies.

A performance anomaly is a sporadic issue that can cause irreparable loss when ignored. There can be several types of performance anomalies that can cause unwanted behavior including:

- Slow/skipped video frames
- Failure in tracking images
- Unexpectedly long financial transactions
- Long processing times for network packets
- Lost network packets

While these behaviors are not visible to traditional sampling-based methods, you can use the Anomaly Detection analysis type to locate them instead. Use this analysis to examine anomalies caused by:.

- Deviations in control flow
- Thread context switches
- Unexpected kernel activity (like interrupts or page faults)
- Drops in CPU frequency

Anomaly Detection is based on Intel® Processor Trace (Intel PT) technology. It provides granular information from the processor at the nanosecond level.

- INGREDIENTS
- DIRECTIONS:

  1. Prepare Application for Analysis
  2. Run Anomaly Detection
  3. Identify Anomalies
  4. Select Anomaly for Investigation
  5. Investigate Kernel Activity Anomaly
  6. Investigate Control Flow Deviation Anomaly

Ingredients

Here are the minimum hardware and software requirements for this performance analysis.

- **Application**: Use a sample application of your choice.
- **Microarchitecture**: Intel® Xeon® processor code named Skylake or newer.
- **Tools**: Anomaly Detection Analysis, available in Intel® VTune™ Profiler version 2021 or newer.

NOTE

- Starting with the 2020 release, Intel® VTune™ Amplifier has been renamed to Intel® VTune™ Profiler.
- Most recipes in the Intel® VTune™ Profiler Performance Analysis Cookbook are flexible. You can apply them to different versions of Intel® VTune™ Profiler. In some cases, minor adjustments may be required.
- Get the latest version of Intel® VTune™ Profiler:
  - From the Intel® VTune™ Profiler product page.
  - Download the latest standalone package from the Intel® oneAPI standalone components page.

- **Operating system**:
  - Linux* OS, Fedora 31(Workstation edition) - 64 bit version
  - Windows* 10 OS

Requirements for Intel® PT

- **Operating system:** Any version of Windows* OS or Linux* OS
- **Microarchitecture:** Intel processor code named Skylake or newer

Prepare Application for Analysis

Typically in software performance analysis, you collect massive sets of data. Since performance anomalies are rare and short-lived, they take up only a fraction of these data sets and thus can go easily unnoticed. A better approach is to focus the analysis on a specific code region. You can do this with the Intel® Instrumentation and Tracing Technology (ITT) API.

Prepare your application by selecting a code region:

1. Go to the directory that contains the sample application.
2. Register a name for the code region you want to profile.

```
__itt_pt_region region=__itt_pt_region_create("region of interest");
```

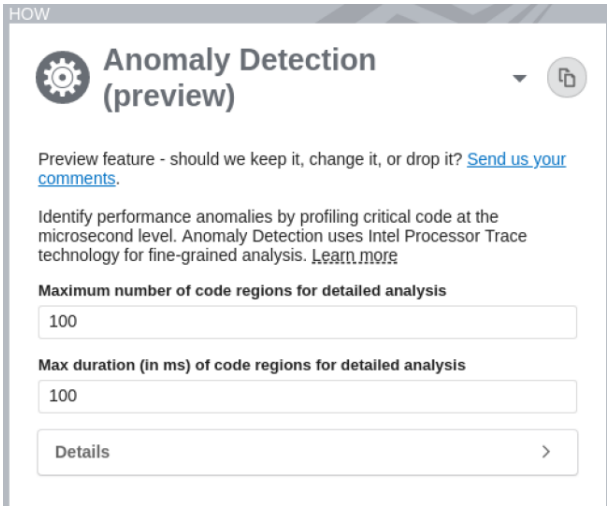
3. In the sample, find a loop that performs operations which are susceptible to anomalies. Use begin and end functions to mark iterations of that loop. For example:

```
double process(std::vector<double> &cache)
{
   double res=0;
   for (size_t i=0; i<ITERATIONS; i++)
   {
     __itt_mark_pt_region_begin(region);
     res+=calculate(i, cache);
     __itt_mark_pt_region_end(region);
   }
return res;
}
```

Run Anomaly Detection

1. On the Welcome screen, click **Configure Analysis**.
2. In the Analysis Tree, select the **Anomaly Detection** analysis type in the **Algorithm** group.
3. In the **WHAT** pane, specify your application and any relevant application parameters.
4. In the **HOW** pane, specify these parameters to define the volume of data collected for the analysis.

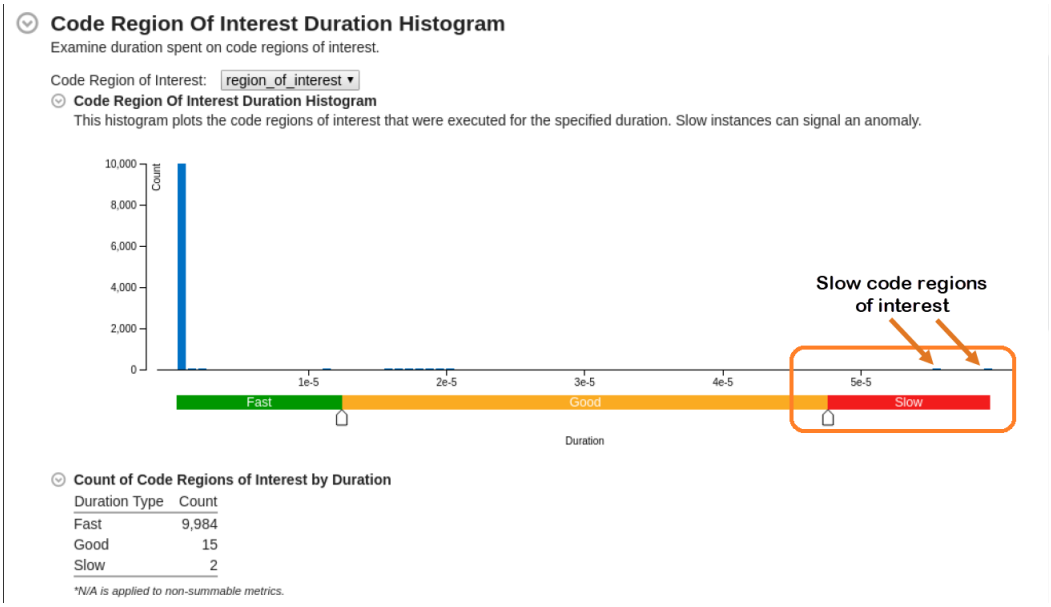
Parameter	Description	Range	Recommended Value
Maximum number of code regions for detailed analysis	Specify the maximum number of code region instances for your application that should be loaded with details simultaneously for result analysis.	10-5000	For faster loading of details, pick a value not more than 1000.
Maximum duration of code regions for detailed analysis	Specify the maximum duration of analysis time (ms) to be spent on each instance of a code region. Instances that require longer duration are either ignored or not loaded.	0.001-1000	Any value under 1000 ms. You may also want to consider some options to limit data collection as a large volume of data can impact processing efficiency adversely.



5. Click the **Start** button to run the analysis.

Identify Anomalies

1. Once the analysis completes, switch to the **Summary** window. Take a look at the **Code Region of Interest Duration Histogram**.



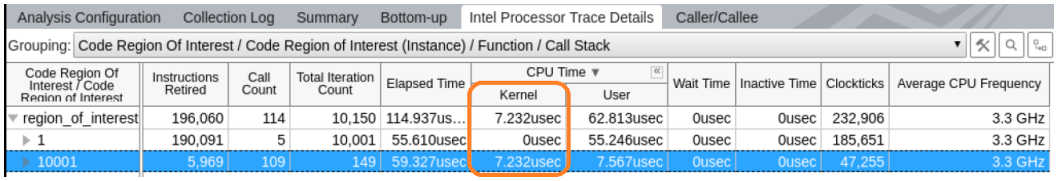
2. Where performance was slow, move the sliders in the histogram to expose performance outliers.
3. Switch to the **Bottom-up** window.
4. In the Grouping table, load details for slow code regions of interest.

   a. Expand the view to display **Fast** and **Slow** regions.
   b. Right click on the **Slow** region in the table.
   c. In the pop-up menu, select **Load Intel Processor Data by Selection**.

Select Anomaly for Investigation

Once you load data, switch to the Intel Processor Trace Details view. Examine the information collected for slow code regions.

In this example, the metrics for Inactive and Wait Times were zero, which indicates that there were no context switches.



Code Region Of Interest / Code Region of Interest	Instructions Retired	Call Count	Total Iteration Count	Elapsed Time	CPU Time ▼		Wait Time	Inactive Time	Clockticks	Average CPU Frequency
					Kernel	User				
region_of_interest	196,060	114	10,150	114.937us...	7.232usec	62.813usec	0usec	0usec	232,906	3.3 GHz
1	190,091	5	10,001	55.610usec	0usec	55.246usec	0usec	0usec	185,651	3.3 GHz
10001	5,969	109	149	59.327usec	7.232usec	7.567usec	0usec	0usec	47,255	3.3 GHz

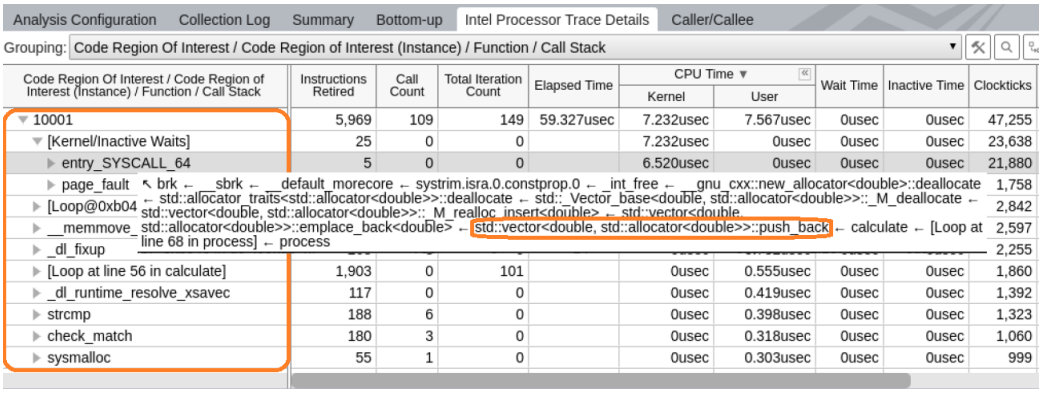
The non-zero kernel time give us a clue about unexpected kernel activity.

From the **Code Region of Interest Duration Histogram**, we identified two slow code regions of interest. Let us start our investigation with code region instance 10001 which has a significant value for Kernel CPU time.

Investigate Kernel Activity Anomaly

The first anomaly lies in region 10001.

Let us look at the execution details for every code region. In the table, expand the node for a region and check the list of functions that were executed in it.

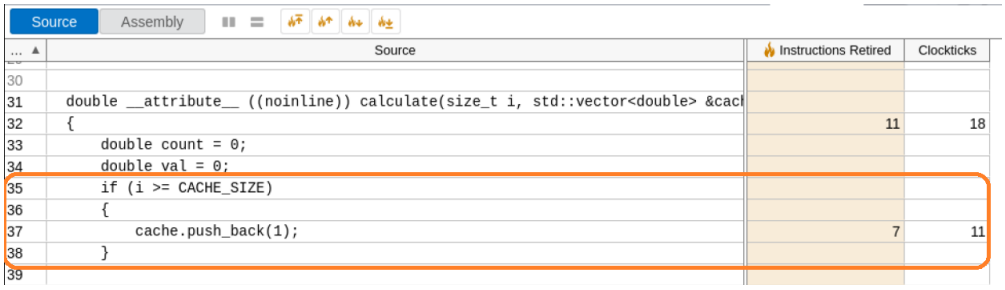


In this example, the *Kernel/Inactive Waits* element is at the top of the function list. Since the Linux kernel employs dynamic code modification, it is not possible to fully reconstruct the kernel control flow using static analysis of kernel binaries. This node aggregates all performance data for kernel activity that happened while executing this particular code region of interest.

Since kernel binaries are not processed, it is not possible to reconstruct control flow metrics like **Call Count**, **Iteration Count**, or **Instructions Retired**. While **Call Count** and **Iteration Count** are zero, **Instructions Retired** shows the number of entries to the kernel.

The stack for this node contains a full sequence of function calls, including kernel entry points. This explains why the application transfers control to the kernel.

The call stacks for the *Kernel/Inactive Waits* element grow from the call to the `push_back` method of `std::vector` from the `calculate` method. Open the function in the **Source** view by double clicking on it.



A close examination reveals the cause of the anomaly.

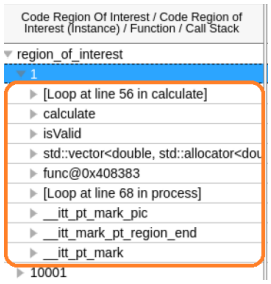
Problem The calculation ran out of the internal software cache size and added a new element into the cache.

Solution Increase the size of the software cache.

Investigate Control Flow Deviation Anomaly

Next, let us look at a different type of anomaly that we observe in the histogram. In this case, the **Instruction Retired** metric is unusually high.

This indicates a deviation in control flow during the execution of that code region. When we expand the node in the grid to see the functions executed, upon first glance, nothing looks abnormal.



Let us load the details for fast and slow iterations together so we can compare them.

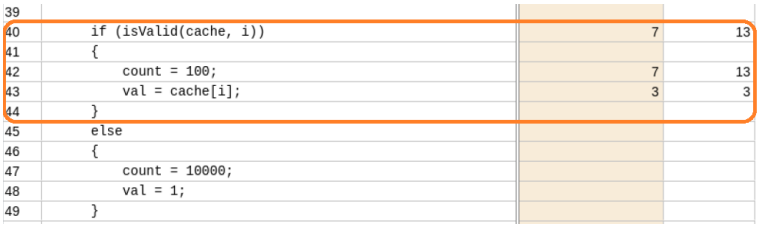
Code Region Of Interest / Code Region of Interest (Instance) / Function / Call Stack	Instructions Retired	Call Count	Total Iteration Count
region_of_interest	392,748	702	20,046
1	190,091	5	10,001
[Loop at line 56 in calculate]	190,003	0	10,001
[Loop at line 68 in process]	12	0	0
__itt_mark_pt_region_end	6	1	0
__itt_pt_mark	2	0	0
__itt_pt_mark_pic	3	1	0
calculate	29	1	0
func@0x408383	1	0	0
isValid	24	1	0
std::vector<double, std::allocator	11	1	0

Code Region Of Interest / Code Re...	Instructions R...	Call Count	Total Iteratio...
region_of_interest	392,748	702	20,046
1	190,091	5	10,001
2	2,007	6	101
[Loop at line 56 in calculate]	1,903	0	101
[Loop at line 68 in process]	12	0	0
__itt_mark_pt_region_end	6	1	0
__itt_pt_mark	2	0	0
__itt_pt_mark_pic	3	1	0
calculate	35	1	0
func@0x408383	1	0	0
isValid	23	1	0
std::vector<double, std::allocat	22	2	0

Although the list of executed functions is the same, the anomalous instance ran more loop iterations of the `calculate` function.

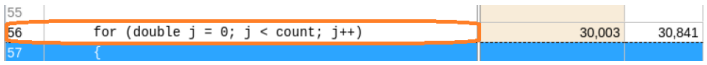
Let us open the `calculate` function in the **Source** view for both fast and slow instances .

In the fast instance, the `isValid` condition is satisfied and a data element is in the cache.



```
39
40      if (isValid(cache, i))                    7   13
41      {
42          count = 100;                          7   13
43          val = cache[i];                       3    3
44      }
45      else
46      {
47          count = 10000;
48          val = 1;
49      }
```

In the slow instance, the `isValid` condition is not satisfied and it fails to validate a data element in the cache. The `else` clause goes into effect and this results in additional calculations.



```
55
56      for (double j = 0; j < count; j++)        30,003   30,841
57      {
```

Problem There were additional calculations that happened in slower iterations in the absence of a valid data element in the cache.

Solution Update the cached data or modify caching algorithms before starting the calculations.

NOTE
Discuss this recipe in the VTune Profiler developer forum.

See Also

Anomaly Detection Analysis
Anomaly Detection View

Profiling an OpenMP* Offload Application running on a GPU

Use this recipe to build and compile an OpenMP application offloaded onto an Intel GPU. The recipe also describes how to use Intel® VTune™ Profiler to run analyses with GPU capabilities (HPC Performance Characterization, GPU Offload, and GPU Compute/Media Hotspots) on the OpenMP application and examine results.*

Content expert: Cory Levels

- INGREDIENTS
- DIRECTIONS:

  1. Build and Compile an OpenMP Offload Application
  2. Run HPC Performance Characterization Analysis on the OpenMP Offload Application
  3. Analyze HPC Performance Characterization Data
  4. Run GPU Offload Analysis on the OpenMP Offload Application
  5. Analyze GPU Offload Analysis Data
  6. Run GPU Compute/Media Hotspots Analysis
  7. Analyze Your Compute Task

Ingredients

Here are the minimum hardware and software requirements for this performance analysis.

- **Application**: `iso3dfd_omp_offload` OpenMP Offload sample. This sample application is available as part of the code sample package for Intel® oneAPI toolkits.
- **Compiler**: To profile a SYCL* application, you need the Intel® oneAPI DPC++/C++ Compiler that is available with the Intel® oneAPI Base Toolkit.
- **Tools**: Intel® VTune™ Profiler
  - HPC Performance Characterization analysis
  - GPU Offload analysis
  - GPU Compute/Media Hotspots analysis

  NOTE

  - Starting with the 2020 release, Intel® VTune™ Amplifier has been renamed to Intel® VTune™ Profiler.
  - Most recipes in the Intel® VTune™ Profiler Performance Analysis Cookbook are flexible. You can apply them to different versions of Intel® VTune™ Profiler. In some cases, minor adjustments may be required.
  - Get the latest version of Intel® VTune™ Profiler:
    - From the Intel® VTune™ Profiler product page.
    - Download the latest standalone package from the Intel® oneAPI standalone components page.

- **Microarchitecture**:
  - Intel Processor Graphics Gen 9 or newer
- **Operating system**:
  - Linux* OS, kernel version 4.14 or newer
  - Windows* 10 OS
- **System Configuration**:
  - Linux* OS: Follow instructions in Configure Your CPU or GPU System (Linux)

- Windows*: Follow instructions in Configure Your CPU or GPU System (Windows)

Build and Compile the OpenMP Offload Application

On Linux OS:

1. Set oneAPI environment variables. Run `setvars.h`. You can find this script here: `. /opt/intel/oneapi/setvars.sh`
2. Go to the sample directory.

```
cd <sample_dir>/DirectProgramming/C++/StructuredGrids/iso3dfd_omp_offload
```

3. Compile the OpenMP Offload application:

```
mkdir build;
cd build;
cmake -DVERIFY_RESULTS=0 -DCMAKE_CXX_FLAGS="-g -mllvm -parallel-source-info=2"  ..
make -j
```

   This generates an `src/iso3dfd` executable.

   To delete the program, type:

```
make clean
```

   This removes the executable and object files that were created by the `make` command.

On Windows OS:

1. Set oneAPI environment variables. Run `setvars.bat`. You can find this script here: `C:\Program Files (x86)\Intel\ oneAPI \setvars.bat`.
2. Open the sample directory:

```
cd <sample_dir>/ DirectProgramming/C++/StructuredGrids/iso3dfd_omp_offload
```

3. Compile the OpenMP Offload application:

```
mkdir build
cd build
icx /Zi -mllvm -parallel-source-info=2 /std:c++17 /EHsc /Qiopenmp /I../include\ /Qopenmp-
targets:spir64 /DUSE_BASELINE /DEBUG ..\src\iso3dfd.cpp ..\src\iso3dfd_verify.cpp ..\src
\utils.cpp
```

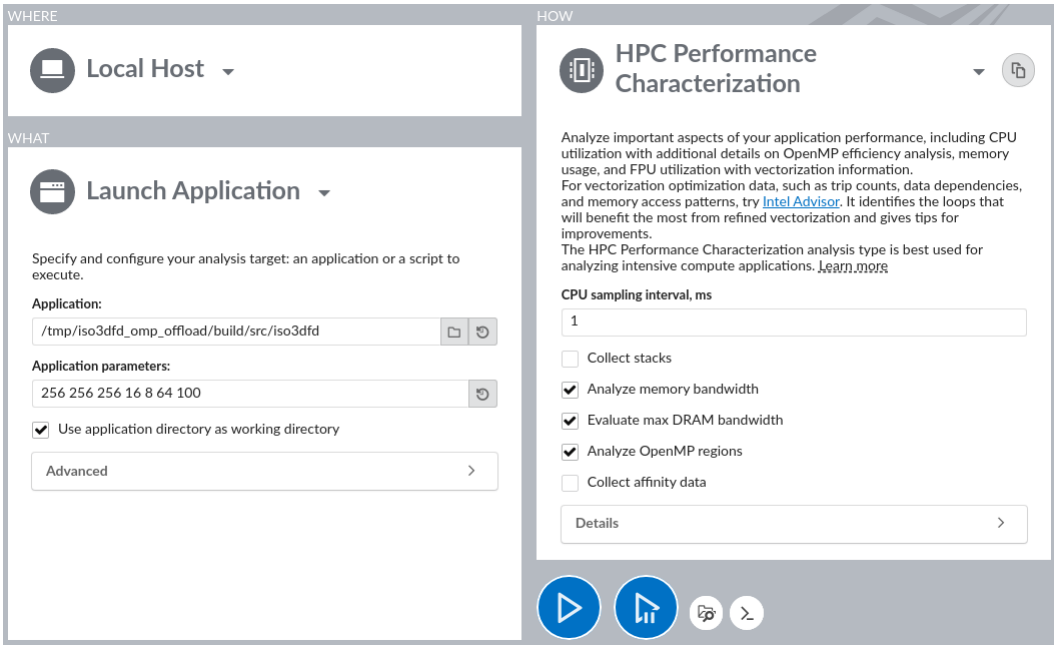
Run HPC Performance Characterization Analysis on the OpenMP Offload Application

To get a high-level summary of the performance of the OpenMP Offload application, run the HPC Performance Characterization analysis. This analysis type can help you understand how your application utilizes the CPU, GPU, and available memory. You can also see the extent to which your code is vectorized.

For OpenMP offload applications, the HPC Performance Characterization analysis shows you the hardware metrics associated with each of your OpenMP offload regions.

Prerequisites: Prepare the system to run a GPU analysis. See Set Up System for GPU Analysis.

1. Open VTune Profiler and click on **New Project** to create a project.
2. On the welcome page, click on **Configure Analysis** to set up your analysis.
3. Select these settings for your analysis.
   - In the **WHERE** pane, select **Local Host**.
   - In the **WHAT** pane, select **Launch Application** and specify the `iso3dfd_omp_offload` binary as the application to profile.
   - In the **HOW** pane, select the **HPC Performance Characterization** analysis type from the **Parallelism** group in the Analysis Tree.



4. Click the **Start** button to run the analysis.

Run Analysis from Command Line:

To run the HPC Performance Characterization analysis form the command line:

- On Linux OS:
  1. Set VTune Profiler environment variables by exporting the script:

     ```
     export <install_dir>/vtune-vars.sh
     ```
  2. Run the HPC Performance Characterization analysis:

     ```
     vtune -collect hpc-performance -- src/iso3dfd 256 256 256 16 8 64 100
     ```
- On Windows OS:
  1. Set VTune Profiler environment variables by running the batch file:

     ```
     <install_dir>\vtune-vars.bat
     ```
  2. Run the HPC Performance Characterization analysis:

     ```
     vtune -collect hpc-performance -- iso3dfd.exe 256 256 256 16 8 64 100
     ```

Analyze HPC Performance Characterization Data

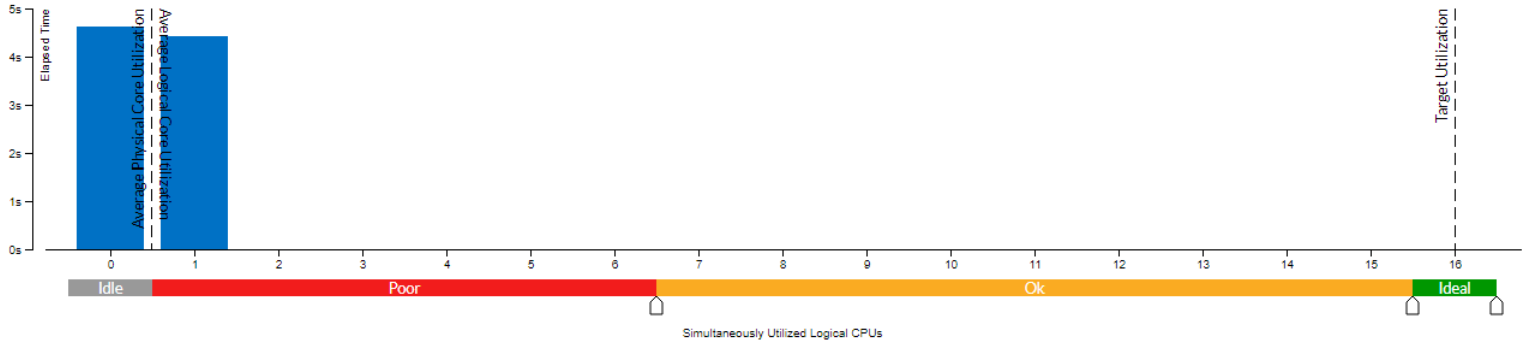
Start your analysis by examining the **Summary** pane. Look at the **Effective Physical Core Utilization** (or **Effective Logical Core Utilization**) and **GPU Stack Utilization** sections to see highlighted issues, if any.

Effective Physical Core Utilization : 6.1% (0.491 out of 8)

Effective Logical Core Utilization : 3.1% (0.495 out of 16)

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.



In the **GPU Stack Utilization** section, look at the top OpenMP offload regions sorted by offload time spent in those regions. You can see GPU utilization in each of these offload regions.

: 45.0%

eak value
) of elapsed time
070s) of offload time
45s) of offload time
32s) of offload time
gions

OpenMP Offload Region	Offload Time	Percentage of Elapsed Time	Data Trans
arget$region:dvc=0@/home/u203405/cookbook_updates/oneAPI-samples/DirectProgramming/C++/StructuredGrids/iso3dfd_omp_offload_cpu/src/iso3dfd.cpp:107	4.071s	44.9%	
gion:dvc=0@/home/u203405/cookbook_updates/oneAPI-samples/DirectProgramming/C++/StructuredGrids/iso3dfd_omp_offload_cpu/src/iso3dfd.cpp:389	0.076s	0.8%	0.04
fload Region]		0.0%	

metrics.

If you compiled your application with the full set of debug information, the names of the regions will contain their source locations. This includes:

- Name of the function
- Name of the source file
- Line number

Generate a Summary Report From the Command Line

To generate a summary report from the command line, type:

```
vtune -report summary -r <result>
```

In this example, the offload activity is classified almost entirely as **Compute** activity. Also, a single offload region consumed the majority of offload time. Click on its name to switch to the **Bottom-Up** view. Examine the grouping table with OpenMP offload region durations, region instance counts, and metrics for GPU and CPU.

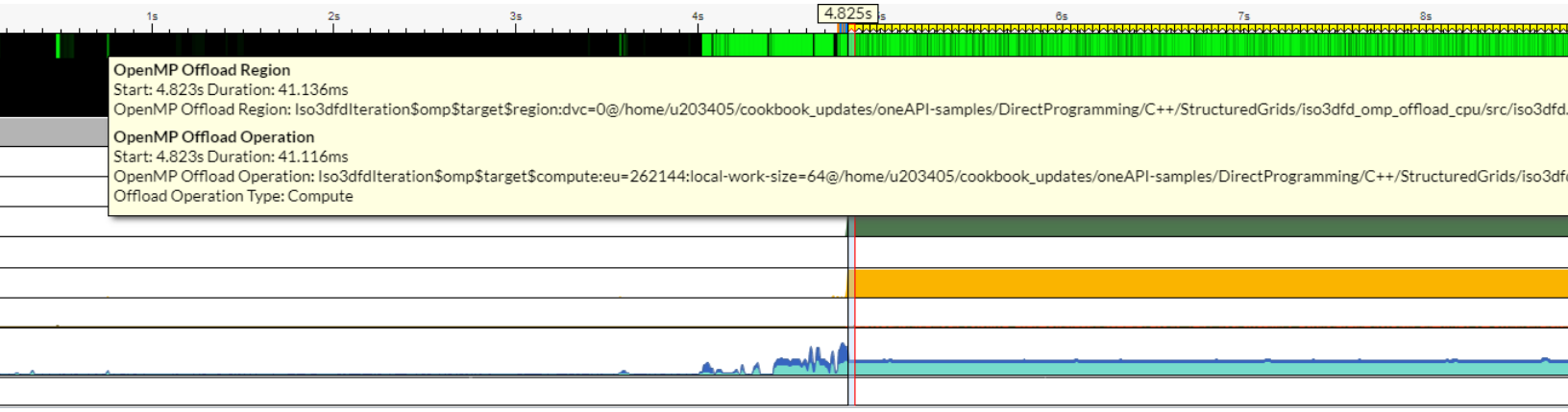
OpenMP Offload Region / Function / Call Stack	OpenMP Offload Time ▼			Instance Count	GPU				CPU Time		
	Compute	Data Transfer	Overhead		EU State			Occupancy	Effective Time	Spin Time	Overhead Time
					Active	Stalled	Idle				
▶ Iso3dfdIterationomptarget$region:dvc=0@	4.070s	0s	0.000s	100	44.7%	55.2%	0.1%	99.1%	3.589s	0.423s	0s
▶ Iso3dfdomptarget$region:dvc=0@/home/u	0s	0.045s	0.031s	2	4.4%	5.5%	90.2%	9.7%	0.072s	0s	0s
▶ [Outside any OpenMP Offload Region]					17.2%	21.4%	61.4%	37.8%	0.819s	0s	0s

Hover over the region markers at the top of the timeline view. You can see the name and duration of each offload region and offload operation within that region. The GPU metrics in the timeline help you understand how every instance of an offload region behaves over time.

Generate a Hotspots Report Grouped by Offload Region From the Command Line

To generate a Hotspots report (grouped by offload region) from the command line, type:

```
vtune -report hotspots -group-by=offload-region -r <result>
```

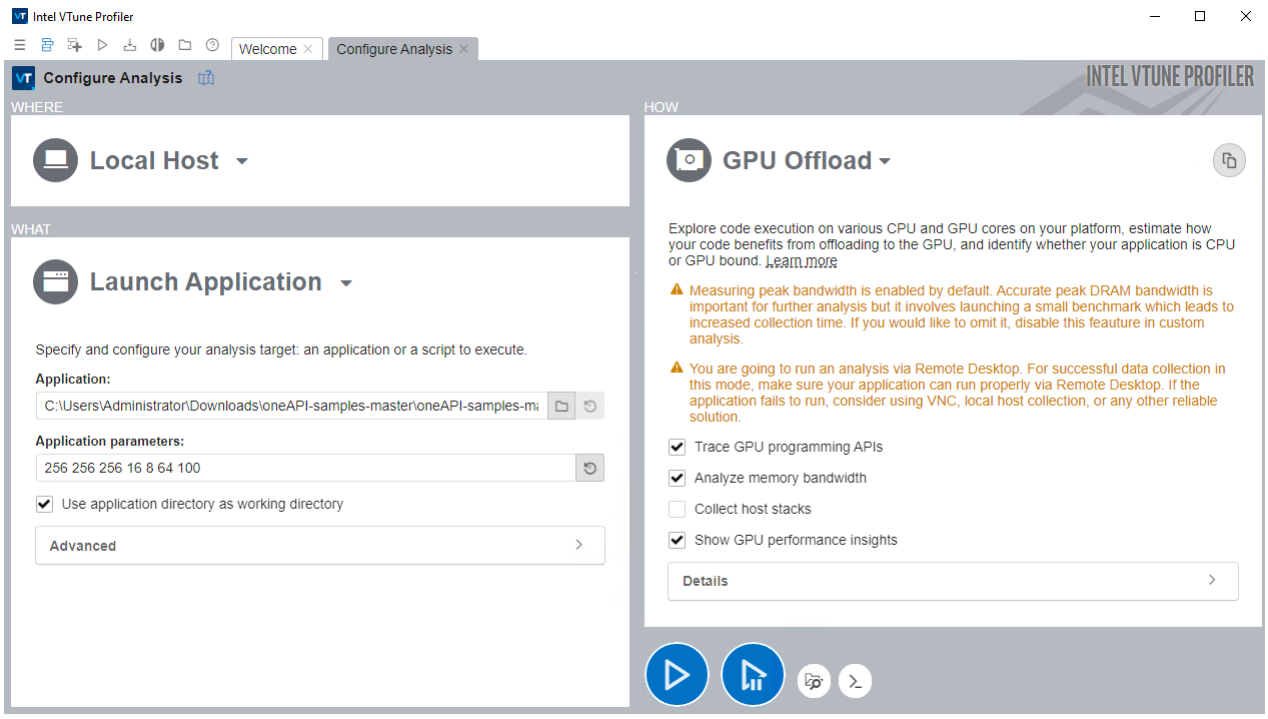


These details establish that GPU activity played an important role in the performance of this application. Next, let us run the GPU Offload Analysis to learn more.

Run GPU Offload Analysis on the OpenMP Offload Application

Prerequisites: If you have not already done so, prepare the system to run a GPU analysis. See Set Up System for GPU Analysis.

1. On the Analysis Tree, in the **Accelerators** group, select the **GPU Offload** analysis type .
2. Select these settings for your analysis:



3. Click the **Start** button to run the analysis.

Run Analysis from Command Line:

To run the GPU Offload analysis form the command line:

- On Linux OS, type:

```
vtune -collect gpu-offload – src/iso3dfd 256 256 256 16 8 64 100
```

- On Windows OS, type:

```
vtune -collect gpu-offload - iso3dfd.exe 256 256 256 16 8 64 100
```

Analyze GPU Offload Analysis Data

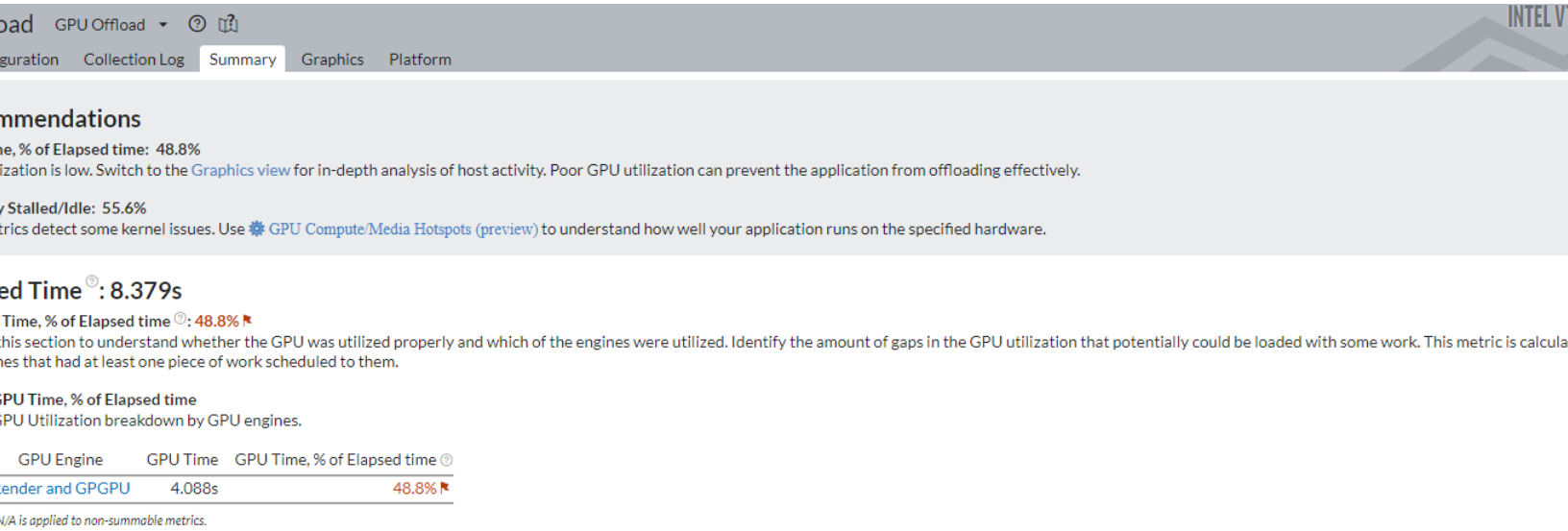
Start your analysis with the **GPU Offload** viewpoint.

In the **Summary** window, see statistics on CPU and GPU resource usage. Use this data to determine if your application is:

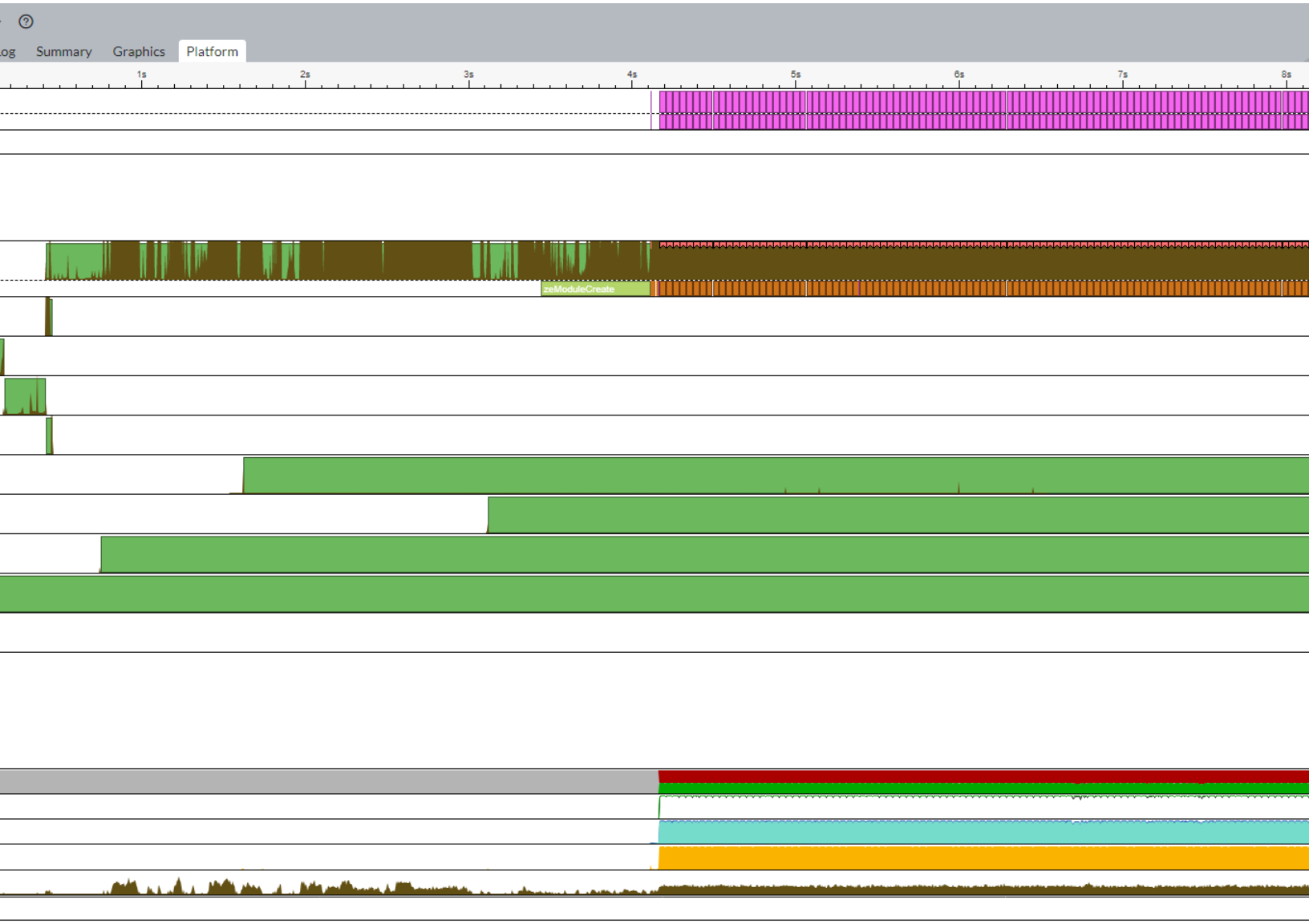
- GPU-bound
- CPU-bound
- Utilizing the compute resources of your system inefficiently

> **NOTE** Families of Intel® Xe graphics products starting with Intel® Arc™ Alchemist (formerly DG2) and newer generations feature GPU architecture terminology that shifts from legacy terms. For more information on the terminology changes and to understand their mapping with legacy content, see GPU Architecture Terminology for Intel® Xe Graphics.

In this example, the application should use the GPU for intensive computation. However, the result summary informs that GPU usage is actually low.



Switch to the **Platform** window. Here, you can see basic CPU and GPU metrics that help analyze GPU usage on a software queue. This data is correlated with CPU usage on the timeline.



The information in the **Platform** window can help you make some inferences.

GPU Bound Applications	CPU Bound Applications
The GPU is busy for a majority of the profiling time.	The CPU is busy for a majority of the profiling time.
There are small idle gaps between busy intervals.	There are large idle gaps between busy intervals.
The GPU software queue is rarely reduced to zero.	

NOTE
Most applications may not present obvious situations as described here. A detailed analysis is important to understand all dependencies. For example, GPU engines that are responsible for video processing and rendering are loaded in turns. In this case, they are used in a serial manner. When the application code runs on the CPU, this can cause an ineffective scheduling on the GPU. The behavior can mislead you to interpret the application to be GPU bound.

Identify the GPU execution phase based on the computing task reference and **GPU Utilization** metrics. Then, you can define the overhead for creating the task and placing it into a queue.

To investigate a computing task, switch to the **Graphics** window to examine the type of work (rendering or computation) running on the GPU per thread. Select the **Computing Task** grouping and use the table to study the performance characterization of your task.

tal Time by D... ▼	Instance Count	Transfer Size		Work Size		SIMD Width	SVM Usage Type	EU Array			Peak EU Threads Occupancy	EU Threa Occupanc
H.. | E | D..		Host-to-Device	Device-to-Host	Global	Local			Active	Stalled	Idle		
.084s	100	0 B	0 B	256 x 256 x 256	64 x 1 x 1	32		44.7%	55.2%	0.0%	100.0%	99.
.001s	0	5 KB	69 B					0.3%	0.4%	99.3%	0.0%	0.

Generate a Hotspots Report Grouped by Computing Task From the Command Line

To generate a Hotspots report (grouped by computing task) from the command line, type:

```
vtune -report hotspots -group-by=computing-task -r <result>
```

Use the README file in the sample to profile other implementations of `iso3dfd_omp_offload code`.

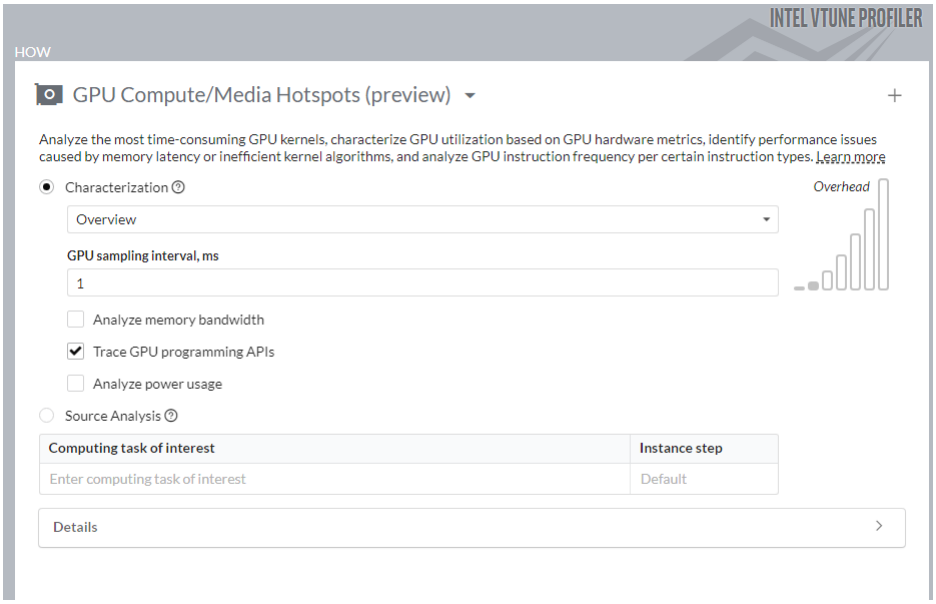
In the next section, continue your investigation with the GPU Compute/Media Hotspots analysis.

Run GPU Compute/Media Hotspots Analysis on the OpenMP Offload Application

Prerequisites: If you have not already done so, prepare the system to run a GPU analysis. See Set Up System for GPU Analysis.

To run the analysis:

1. In the **Accelerators** group, select the **GPU Compute/Media Hotspots** analysis type.
2. Configure analysis options as described in the previous section.
3. Click the **Start** button to run the analysis.



Run Analysis from Command Line

To run the analysis from the command line:

- On Linux OS:

```
vtune -collect gpu-hotspots - src/iso3dfd 256 256 256 16 8 64 100
```

- On Windows OS:

```
vtune -collect gpu-hotspots - iso3dfd.exe 256 256 256 16 8 64 100
```

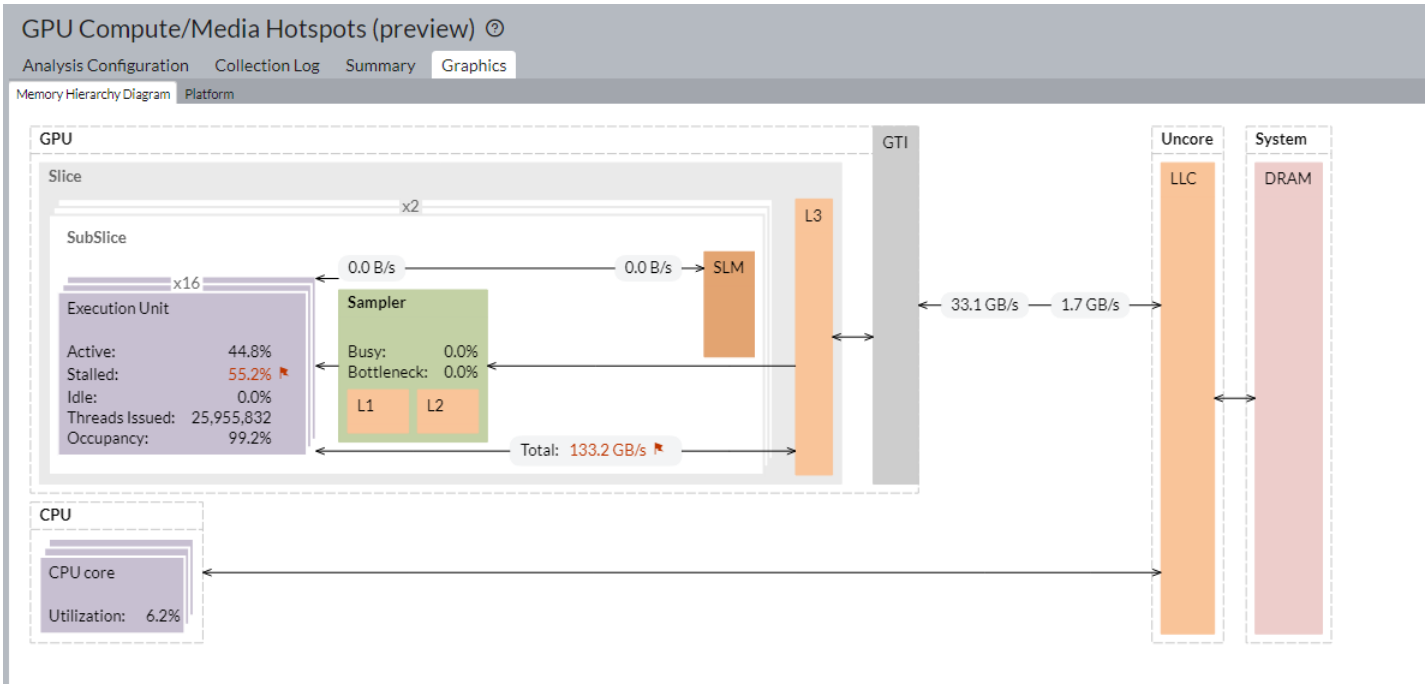
Analyze Your Compute Task

The default analysis configuration invokes the **Characterization** profile with the Overview metric set. In addition to individual compute task characterization that is available through the **GPU Offload** analysis, VTune Profiler provides memory bandwidth metrics that are categorized by different levels of GPU memory hierarchy.

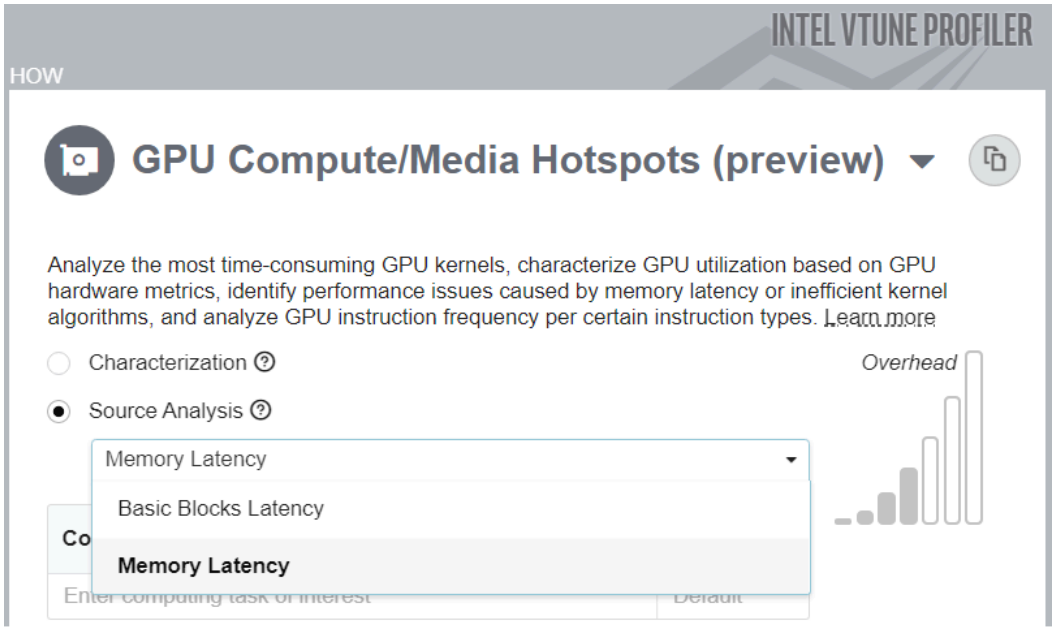
puting Task

Computing Task	Computing Task					Data Transferred		EU Array			Peak EU Threads Occupancy	EU Threads Occupancy	Computing Threads Started	L3 Bar
	Total Time	Average Time	Instance Count	SIMD Width	S...	Size	Total, GB/sec	Active	Stalled	Idle				
onompoffloading:107	4.063s	0.041s	100	32		0 B	0.000	44.8%	55.2%	0.0%	100.0%	99.2%	25,955,832	
istAppendMemoryCopy	0.000s	0.000s	11			5 KB	0.100	0.0%	0.0%	100.0%	0.0%	0.0%	0	
istAppendMemoryCopy	0.000s	0.000s	1			69 B	0.015	0.0%	0.0%	100.0%	0.0%	0.0%	0	
ask]	0s	0s	0			0 B	0.000	0.7%	0.8%	98.5%	0.0%	1.5%	258,587	

For a visual representation of the memory hierarchy, see the **Memory Hierarchy Diagram**. This diagram reflects the microarchitecture of the current GPU and shows memory bandwidth metrics. Use the diagram to understand the data traffic between memory units and execution units. You can also identify potential bottlenecks that cause EU stalls.



You can also analyze compute tasks at the source code level. For example, you can count GPU clock cycles spent on a particular task or due to memory latency. Use the **Source Analysis** option for this purpose.



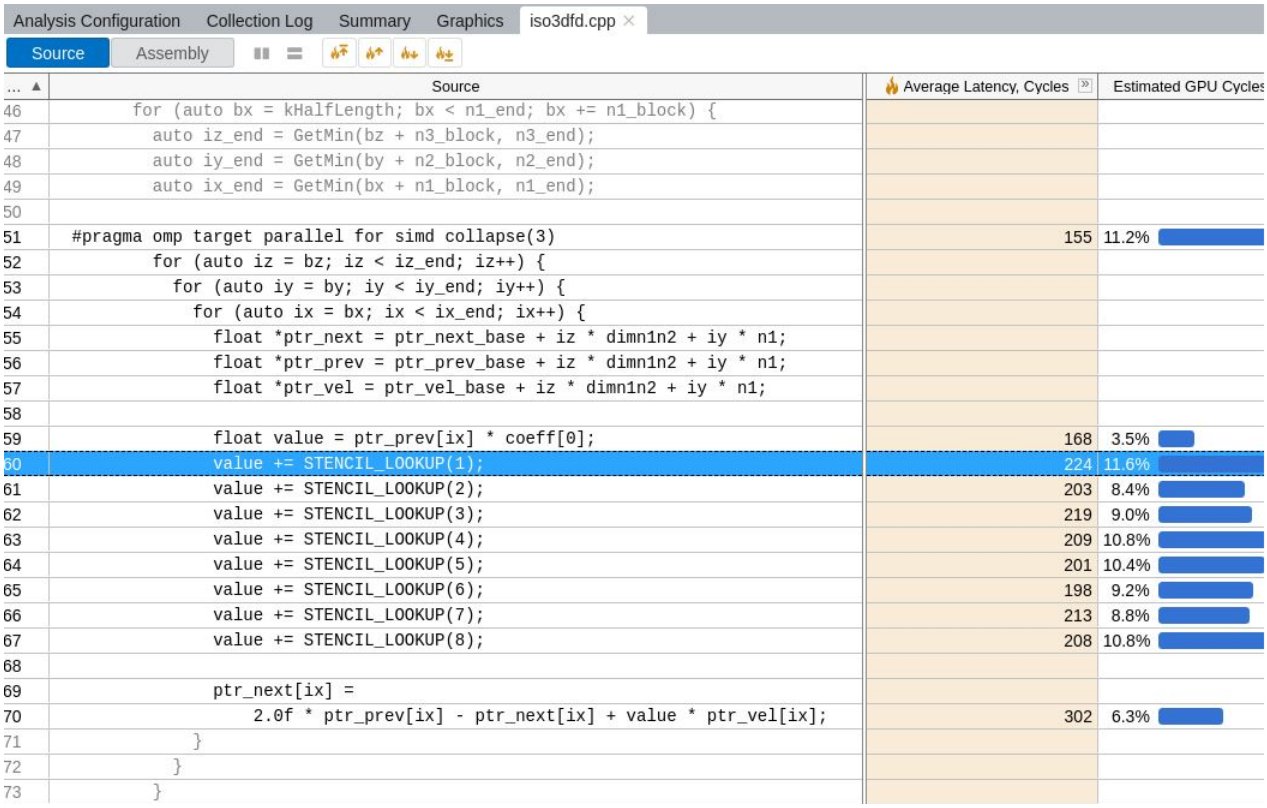
Run Memory Latency Source Analysis from Command Line

To run the analysis with the Memory Latency Source Analysis option from the command line:

- On Linux OS:

```
vtune -collect gpu-hotspots -knob profiling-mode=source-analysis -knob source-analysis=mem-latency -r iso_ghs_src-analysis_mem - ./src/iso3dfd 256 256 256 16 8 64 100
```

In the source view, examine the **Average Latency Cycles** for the offload kernel.



Generate a Hotspots Report With Source for a Computing Task From the Command Line

To generate a Hotspots report (with source for a computing task) from the command line, type:

```
vtune -report hotspots -source-object 'computing-task=Iso3dfdIteration$omp$offloading:50' -group-by=gpu-source-line -r <result>
```

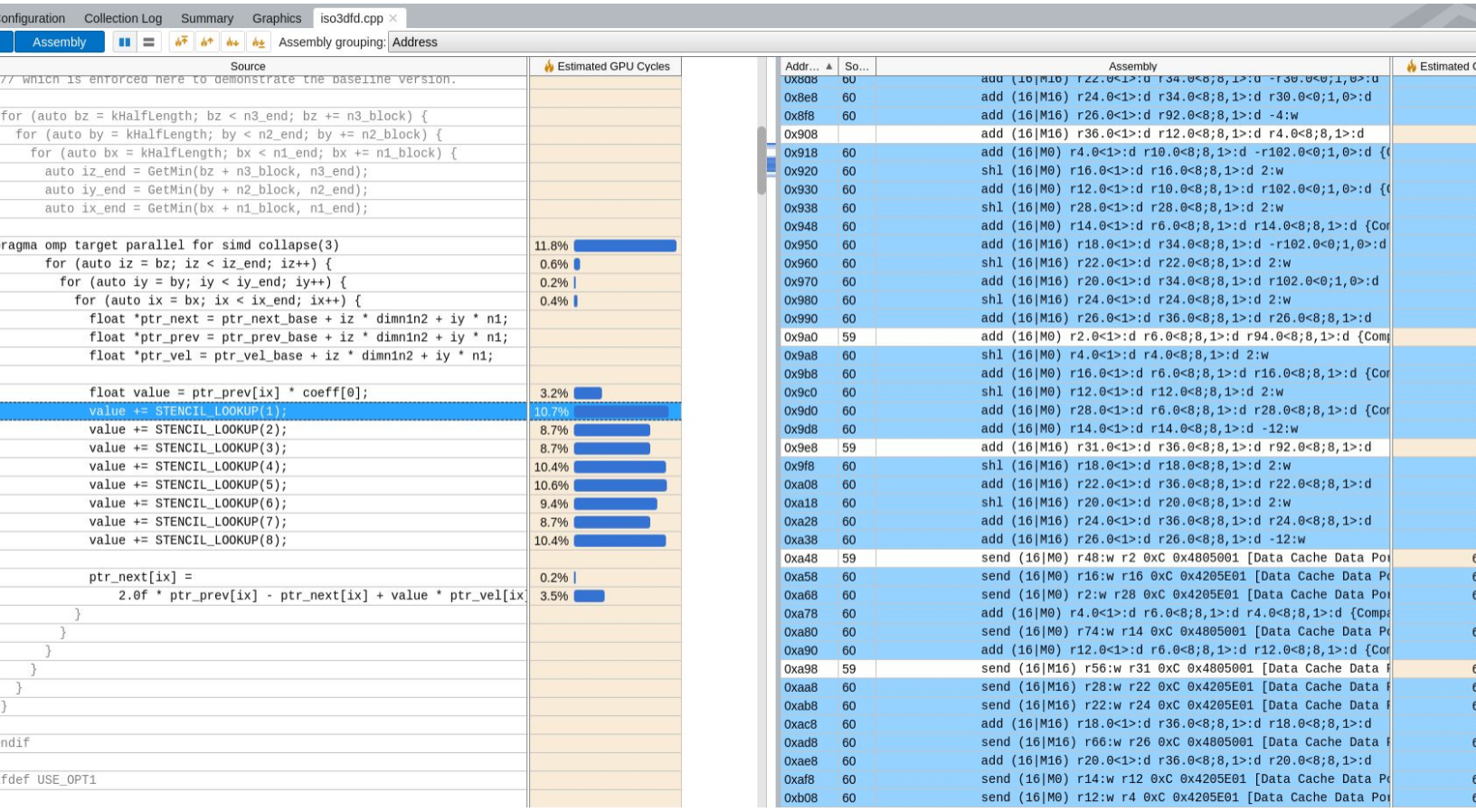
Run Basic Block Latency Source Analysis from Command Line

To run the analysis with the Basic Blocks Latency Source Analysis option from the command line:

- On Linux OS:

```
vtune -collect gpu-hotspots -knob profiling-mode=source-analysis -knob source-analysis=mem-latency -r iso_ghs_src-analysis – ./src/iso3dfd 256 256 256 16 8 64 100
```

In the source view, examine the **Average Latency Cycles** for the offload kernel.



Discuss this recipe in the VTune Profiler developer forum.

See Also

Optimize applications for Intel® GPUs with Intel® VTune Profiler

GPU Architecture Terminology for Intel® Xe Graphics

HPC Performance Characterization Analysis

GPU Offload Analysis

GPU Compute/Media Hotspots Analysis

Profiling a SYCL* Application running on a GPU

Learn how to use Intel® VTune™ Profiler to analyze a SYCL application that has been offloaded onto a GPU.

Content expert: Cory Levels

- INGREDIENTS
- DIRECTIONS:

  1. Build and Compile a SYCL Application
  2. Run GPU Offload Analysis on a SYCL Application
  3. Analyze Collected Data
  4. Run GPU Compute/Media Hotspots Analysis
  5. Analyze Your Compute Task

Ingredients

Here are the minimum hardware and software requirements for this performance analysis.

- **Application**: `matrix_multiply_vtune`. This sample application is available as part of the code sample package for Intel® oneAPI toolkits.
- **Compiler**: To profile a SYCL application, you need the Intel® oneAPI DPC++/C++ Compiler that is available with Intel oneAPI toolkits.
- **Tools**: Intel® VTune™ Profiler - GPU Offload and GPU Compute/Media Hotspots Analyses.
- **Microarchitecture**:
  - Intel Processor Graphics Gen 9 or newer
  - Intel microarchitectures code named Kaby Lake, Coffee Lake, or Ice Lake
- **Operating system**:
  - Linux* OS, kernel version 4.14 or newer.
  - Windows* 10 OS.
- **Graphical User Interface for Linux**:
  - GTK+ (2.10 or higher. ideally, use 2.18 or higher)
  - Pango (1.14 or higher)
  - X.Org (1.0 or higher, ideally use 1.7 or higher)

Build and Compile a SYCL Application

On Linux OS:

1. Go to the sample directory.

```
cd <sample_dir>/VtuneProfiler/matrix_multiply_vtune
```

2. The `multiply.cpp` file in the `src` directory contains several versions of matrix multiplication. Select a version by editing the corresponding `#define MULTIPLY` line in `multiply.hpp`.
3. Compile your sample application:

```
cmake .
make
```

This generates a `matrix.dpcpp -fsycl` executable.

To delete the program, type:

```
make clean
```

This removes the executable and object files that were created by the `make` command.

On Windows OS:

1. Open the sample directory:

```
<sample_dir>\VtuneProfiler\matrix_multiply_vtune
```

2. In this directory, open a Visual Studio* project file named `matrix_multiply.sln`
3. The `multiply.cpp` file contains several versions of matrix multiplication. Select a version by editing the corresponding `#define MULTIPLY` line in `multiply.hpp`
4. Build the entire project with a Release configuration.

   This generates an executable called `matrix_multiply.exe`.

Run GPU Offload Analysis on a SYCL Application

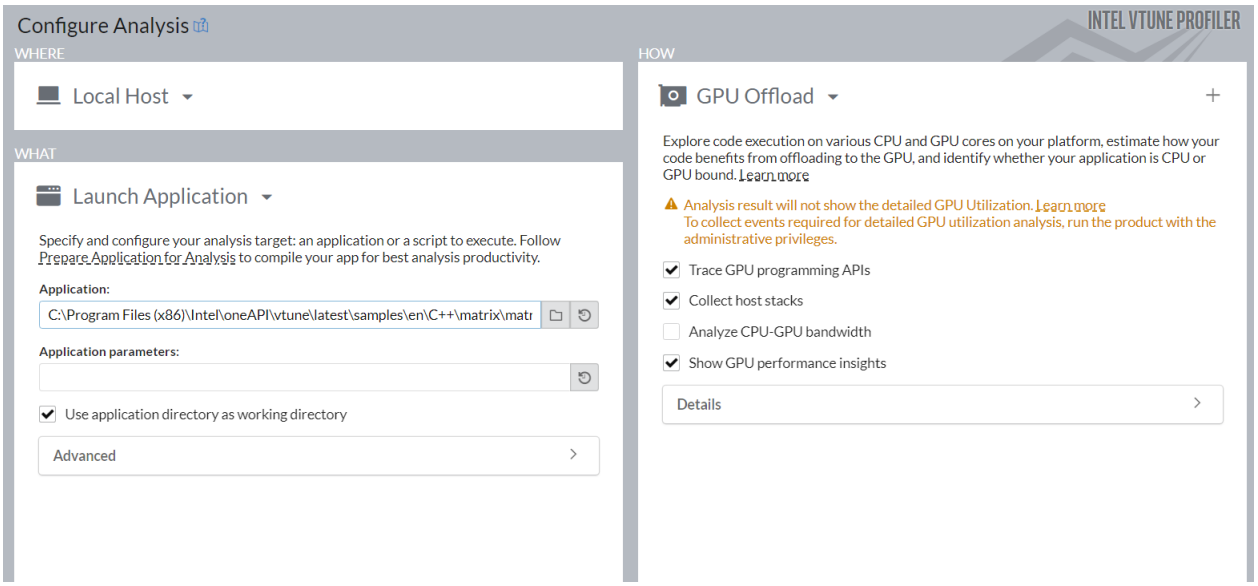
Prerequisites: Prepare the system to run a GPU analysis. See Set Up System for GPU Analysis.

1. Launch VTune Profiler and click **New Project** from the Welcome page.

   The **Create a Project** dialog box opens.
2. Specify a project name and a location for your project and click **Create Project**.

   The **Configure Analysis** window opens.
3. Make sure the **Local Host** is selected in the **WHERE** pane.
4. In the **WHAT** pane, make sure the **Launch Application** target is selected and specify the `matrix_multiply` binary as an **Application** to profile.
5. In the **HOW** pane, select **GPU Offload** analysis type from the **Accelerators** group.

   This is the least intrusive analysis for applications running on platforms with Intel Graphics as well as on other third-party GPUs supported by VTune Profiler.
6. Click the **Start** button to launch the analysis.



Run Analysis from Command Line:

To run the analysis from the command line:

- On Linux OS:

  1. Set VTune Profiler environment variables by exporting the script:

```
export <install_dir>/env/vars.sh
```

  2. Run the analysis:

```
vtune -collect gpu-offload - ./matrix.dpcpp -fsycl
```

- On Windows OS:

1. Set VTune Profiler environment variables by running the batch file:

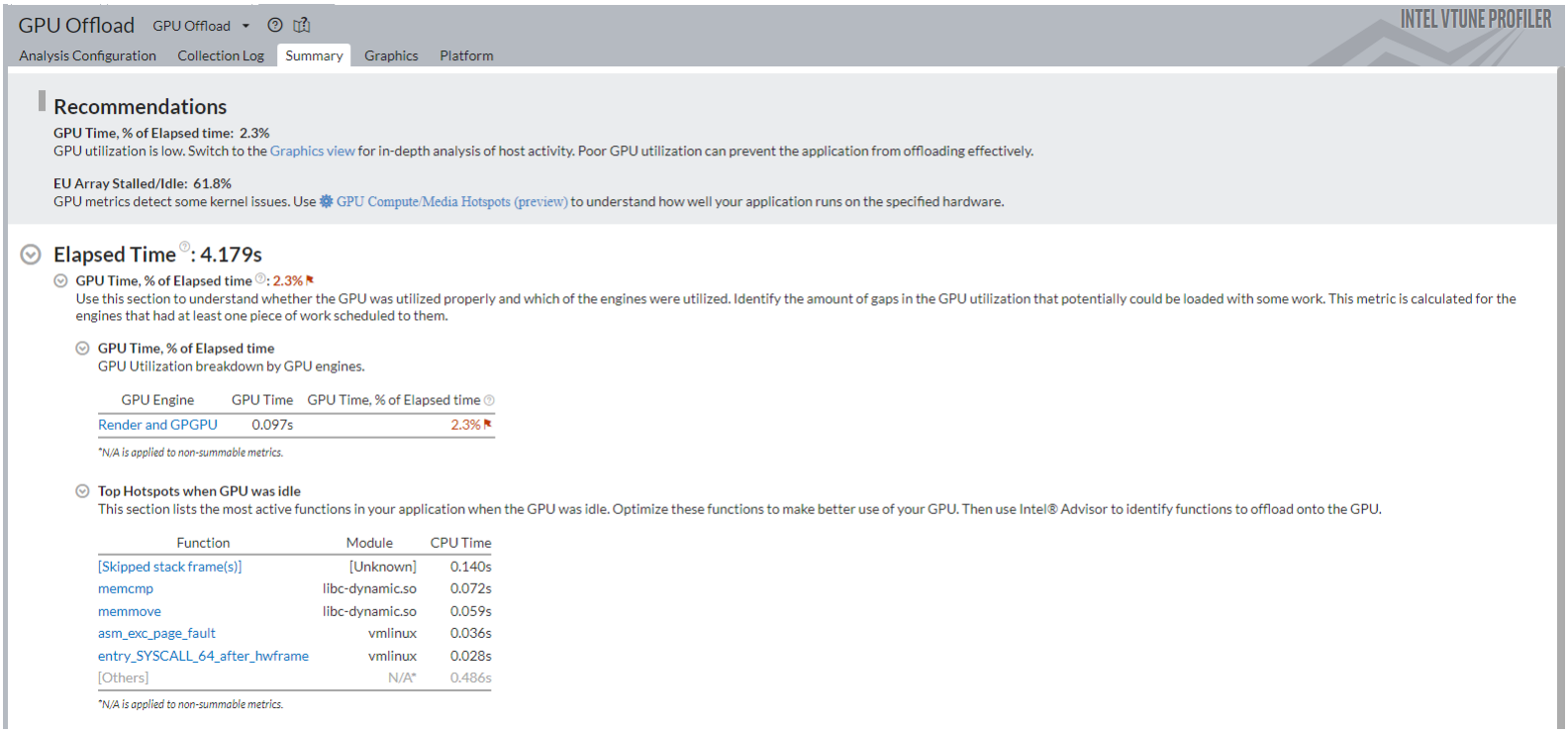
```
export <install_dir>\env\vars.bat
```

2. Run the analysis command:

```
vtune.exe -collect gpu-offload -- matrix_multiply.exe
```

Analyze Collected Data

Start your analysis with the **GPU Offload** viewpoint. In the **Summary** window, see statistics on CPU and GPU resource usage to determine if your application is GPU-bound, CPU-bound, or not effectively utilizing the compute capabilities of the system. In this example, the application should use the GPU for intensive computation. However, the result summary informs that GPU usage is actually low.



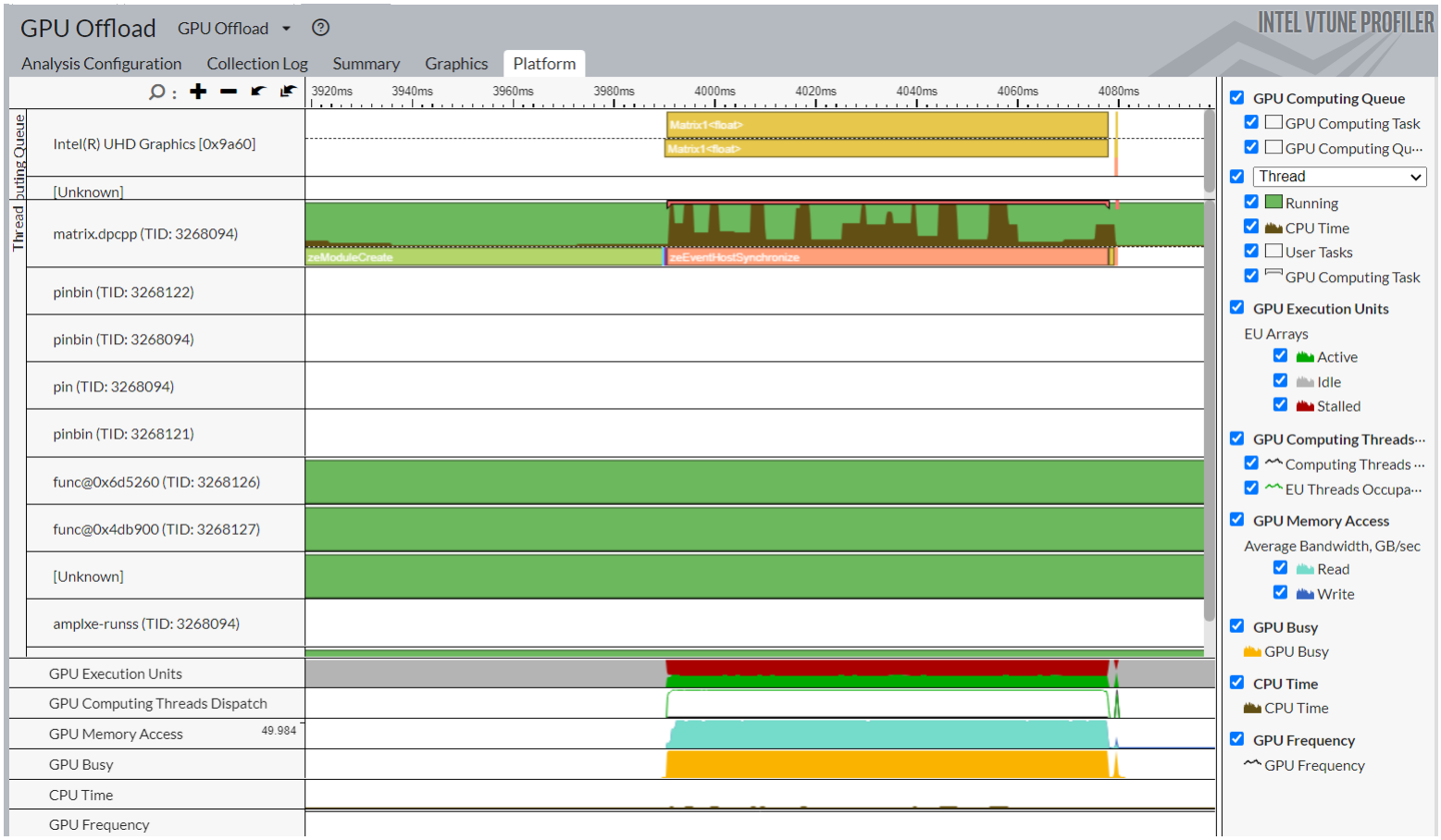
Switch to the **Platform** window. Here, you can see basic CPU and GPU metrics that help analyze GPU usage on a software queue. This data is correlated with CPU usage on the timeline. The information in the **Platform** window can help you make some inferences.

GPU Bound Applications	CPU Bound Applications
The GPU is busy for a majority of the profiling time.	The CPU is busy for a majority of the profiling time.
There are small idle gaps between busy intervals.	There are large idle gaps between busy intervals.
The GPU software queue is rarely reduced to zero.	

NOTE Families of Intel® Xe graphics products starting with Intel® Arc™ Alchemist (formerly DG2) and newer generations feature GPU architecture terminology that shifts from legacy terms. For more information on the terminology changes and to understand their mapping with legacy content, see GPU Architecture Terminology for Intel® Xe Graphics.

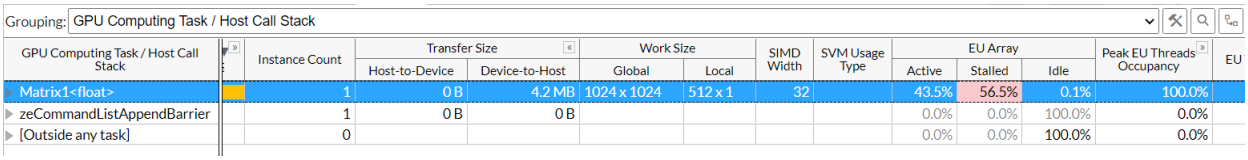
NOTE
Most applications may not present obvious situations as described above. A detailed analysis is important to understand all dependencies. For example, GPU engines that are responsible for video processing and rendering are loaded in turns. In this case, they are used in a serial manner. When the application code runs on the CPU, this can cause an ineffective scheduling on the GPU. The behavior can mislead you to interpret the application to be GPU bound.

Identify the GPU execution phase based on the computing task reference and **GPU Utilization** metrics. Then, you can define the overhead for creating the task and placing it into a queue.



To investigate a computing task, switch to the **Graphics** window to examine the type of work (rendering or computation) running on the GPU per thread. Select the **Computing Task** grouping and use the table to study the performance characterization of your task.

To further analyze your computing task, run the GPU Compute/Media Hotspots analysis type.



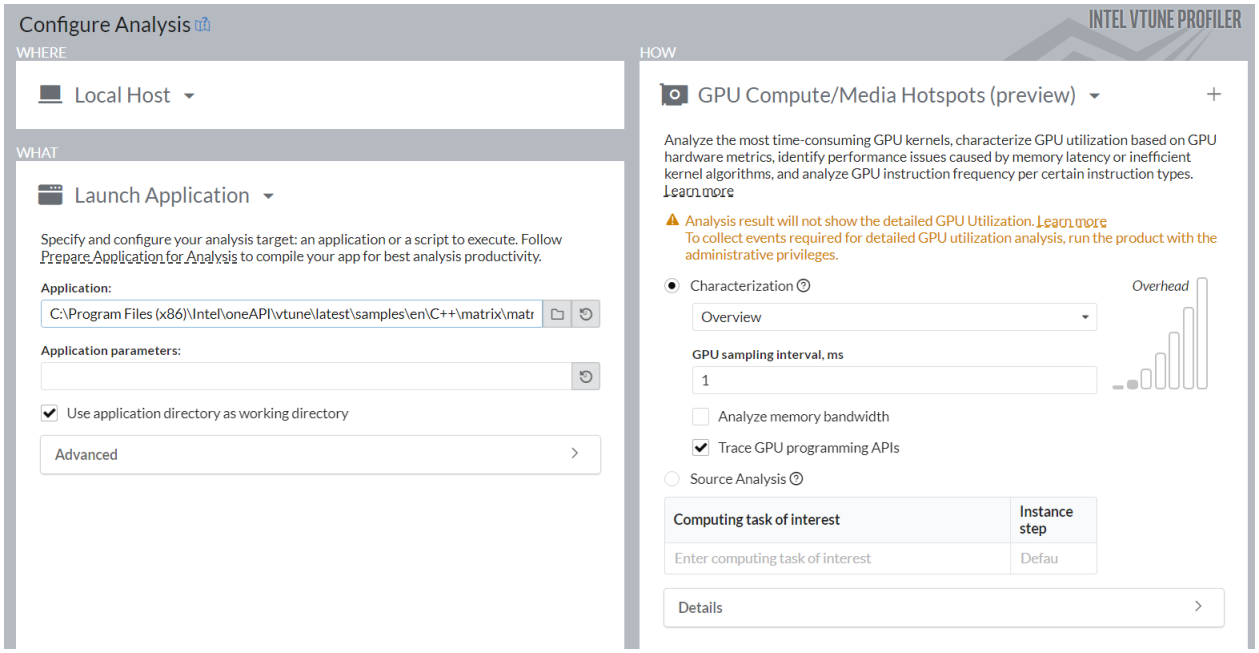
Use the README file in the sample to profile other implementations of `multiply.cpp`.

Run GPU Compute/Media Hotspots Analysis

Prerequisites: If you have not already done so, prepare the system to run a GPU analysis. See Set Up System for GPU Analysis.

To run the analysis:

1. In the **Accelerators** group, select the **GPU Compute/Media Hotspots** analysis type.
2. Configure analysis options as described in the previous section.
3. Click the **Start** button to run the analysis.



Run Analysis from Command Line

To run the analysis from the command line:

- On Linux OS:

```
vtune -collect gpu-hotspots - ./matrix.dpcpp -fsycl
```

- On Windows OS:

```
vtune.exe -collect gpu-hotspots -- matrix_multiply.exe
```

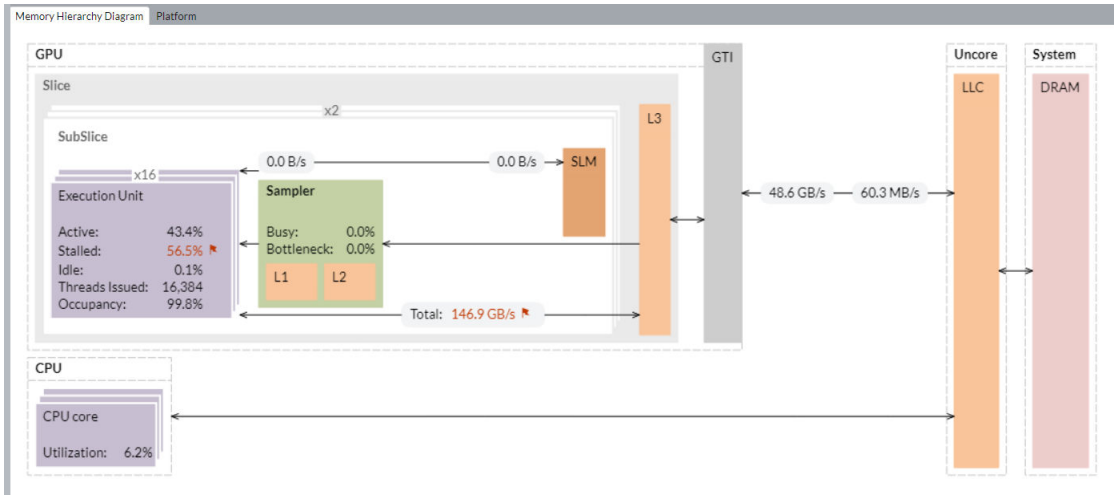
Analyze Your Compute Task

The default analysis configuration invokes the Characterization profile with the Overview metric set. In addition to individual compute task characterization that is available through the **GPU Offload** analysis, VTune Profiler provides memory bandwidth metrics that are categorized by different levels of GPU memory hierarchy.

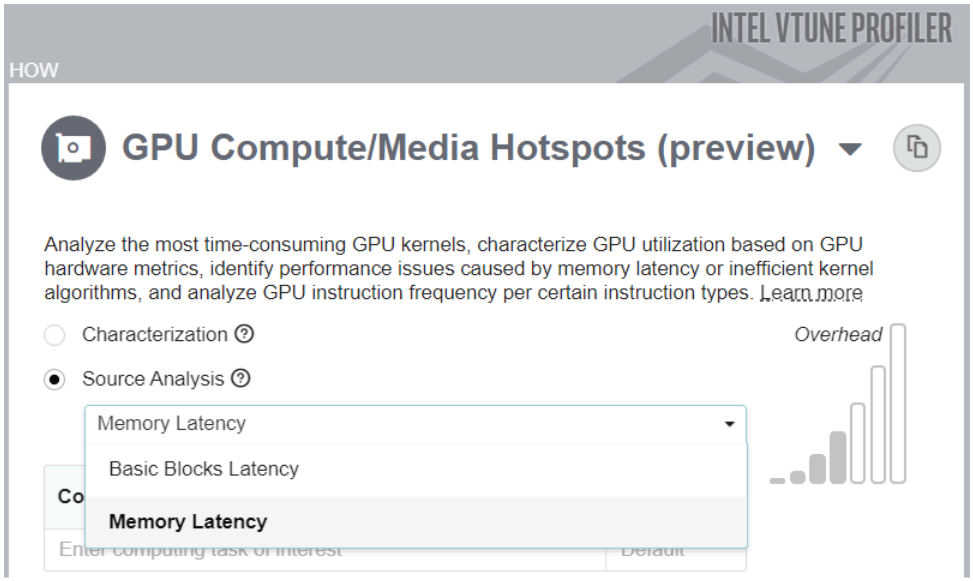
Grouping: Computing Task

Computing Task	Computing Threads Started	L3 Bandwidth, GB/sec	Shared Local Memory ...		GPU Memory Band...		Sampler		GPU Barriers	GPU Atomics
			Read	Write	Read	Write	Busy	Bottleneck		
Matrix1<float>	16,384	146.894	0.000	0.000	48.551	0.060	0.0%	0.0%	0	
zeCommandListAppendMemoryCopyRegion	48,703	59.029	0.000	0.000	7.457	5.325	0.0%	0.0%	0	
zeCommandListAppendBarrier	0	0.000	0.000	0.000	0.000	0.000	0.0%	0.0%	0	
[Outside any task]	16,833	0.039	0.000	0.000	0.011	0.002	0.0%	0.0%	0	

For a visual representation of the memory hierarchy, see the **Memory Hierarchy Diagram**. This diagram reflects the microarchitecture of the current GPU and shows memory bandwidth metrics. Use the diagram to understand the data traffic between memory units and execution units. You can also identify potential bottlenecks that cause EU stalls. For example, in the diagram below, you see that the L3 bandwidth and stalled EUs have both been flagged as potential issues to investigate.



You can also analyze compute tasks at the source code level. For example, to count GPU clock cycles spent on a particular task or due to memory latency, use the **Source Analysis** option.



Discuss this recipe in the VTune Profiler developer forum.

See Also

Optimize applications for Intel® GPUs with Intel® VTune Profiler
Optimize Your GPU Application with Intel oneAPI Base Toolkit
GPU Architecture Terminology for Intel® Xe Graphics
GPU Offload Analysis

GPU Compute/Media Hotspots Analysis

Profiling an FPGA-driven SYCL* Application

Use this recipe to profile an FPGA-driven SYCL application. The recipe features the AOCL Profiler integrated in the CPU/FPGA Interaction (preview) analysis type in Intel® VTune™ Profiler.

- INGREDIENTS
- DIRECTIONS:

  1. Install and Configure the Toolkit
  2. Build the Sample Application
  3. Run CPU/FPGA Interaction Analysis
  4. Analyze Results

Ingredients

Here are the minimum hardware and software requirements for this performance recipe.

- **Application**: `crr`. This sample FPGA design is available in the repository for Intel® oneAPI DPC++ Compiler samples.
- **Compiler**: To profile a SYCL application, you need the `dpcpp` compiler that is available with Intel® oneAPI toolkits.
- **Tools**:
  - Intel® oneAPI Base Toolkit for Linux*
  - Intel® FPGA Add-on for oneAPI Base Toolkit
  - Intel® VTune™ Profiler - CPU/FPGA Interaction (preview) Analysis

  NOTE

  - Starting with the 2020 release, Intel® VTune™ Amplifier has been renamed to Intel® VTune™ Profiler.
  - Most recipes in the Intel® VTune™ Profiler Performance Analysis Cookbook are flexible. You can apply them to different versions of Intel® VTune™ Profiler. In some cases, minor adjustments may be required.
  - Get the latest version of Intel® VTune™ Profiler:
    - From the Intel® VTune™ Profiler product page.
    - Download the latest standalone package from the Intel® oneAPI standalone components page.

- **Operating system**: Linux* OS (Ubuntu* 18.04)
- **CPU**: Intel server platform code-named Cascade Lake
- **FPGA**: Intel® Programmable Acceleration Card (Intel® PAC) with Intel® Arria® 10 GX FPGA or Intel® Stratix 10 GX FPGA PAC board for SYCL (with installable add-on)

Install and Configure the Toolkit

1. Plug the Intel PAC card into the PCIe slot on the machine.
2. Download and install Intel oneAPI Base Toolkit for Linux. Select all default options and either the online or offline installer.
3. Download Intel FPGA Add-on for oneAPI Base Toolkit.
4. Unzip the FPGA add-on package and run `setup.sh`. Select all default options.
5. Set up the oneAPI environment.

```
source <oneAPI-install-dir>/setvars.sh
```

6. Install the FPGA board.

```
aocl install
```

7. Run the diagnose command to ensure that all diagnostics pass.

```
aocl diagnose
```

Build the Sample Application

1. Download code samples from the repository for Intel oneAPI DPC++ Compiler samples.

```
git clone https://github.com/intel/BaseKit-code-samples.git
```

2. Open the `crr` sample folder.

```
cd BaseKit-code-samples/FPGAExampleDesigns/crr
```

3. Open the `src/CMakeLists.txt` file.
4. Locate the line of code that lists hardware flags. It should start with `set(HARDWARE_LINK_FLAGS`.
5. Add `-Xsprofile` to the set of flags.
6. Go back to the main directory for the sample. Create a new folder called `build` and open it.

```
mkdir build
cd build
```

7. Compile the sample.

```
cmake ..
make fpga
```

   This process can take several hours. Once it has finished, you should have an executable file called `crr.fpga`.

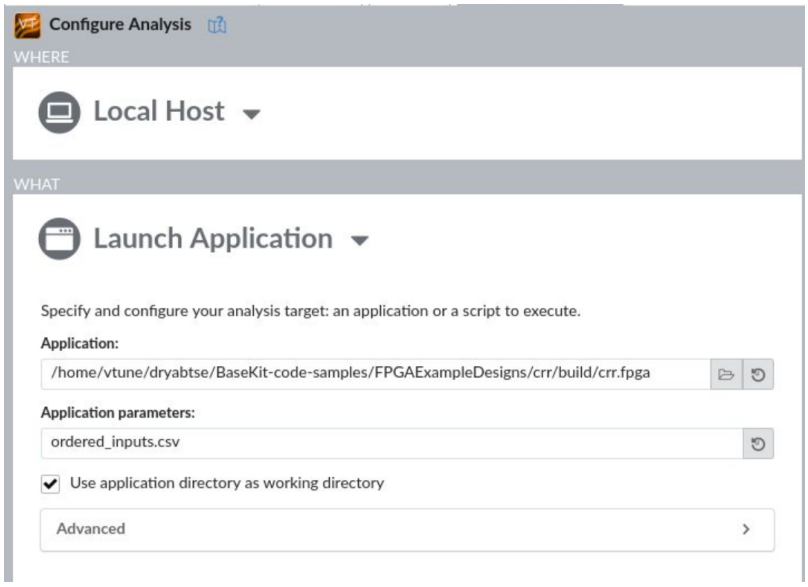
You can now run `crr.fpga` on FPGA hardware.

Run CPU/FPGA Interaction Analysis

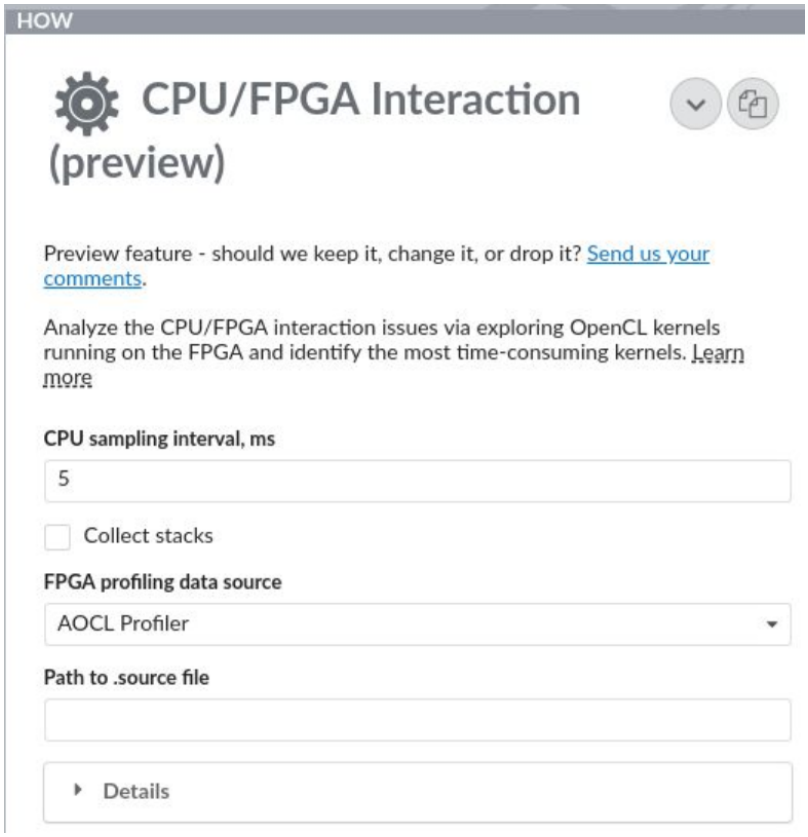
1. Open Intel® VTune™ Profiler and click **New Project** on the Welcome screen.

   The **Create a Project** dialog box opens.
2. Specify a project name, a location for your project, and click **Create Project**.

   The **Configure Analysis** window opens.
3. In the **WHERE** pane, select **Local Host**.
4. In the **WHAT** pane, select **Launch Application** as the target.
   - In the **Application** field, specify the path to the `crr.fpga` executable.
   - In the **Application parameters** field, enter `ordered_inputs.csv`.



5. In the **HOW** pane, select **CPU/FPGA Interaction (preview)** from the **Platform Analysis** group.
6. In the analysis settings, select `AOCL Profiler` for the **FPGA profiling data source**.

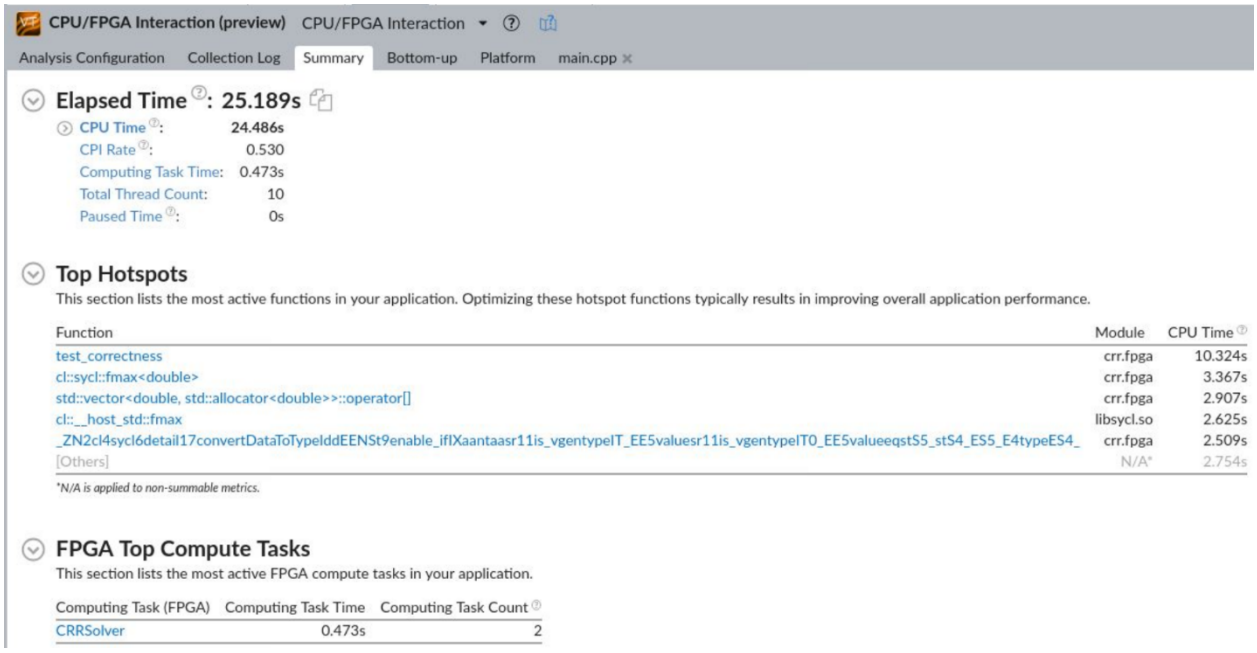


7. Click **Start** at the bottom to run the analysis.

Analyze Results

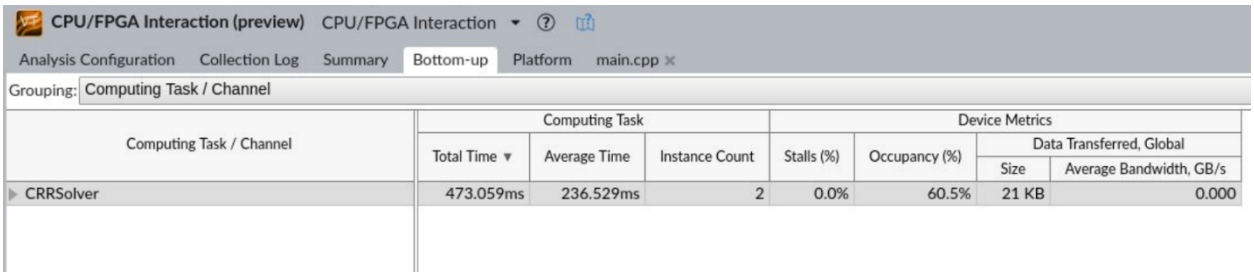
Once data collection completes, you can see the finalized results in the **CPU/FPGA Interaction** viewpoint. Start with the **Summary** window to view these details:

- FPGA top compute tasks
- Top tasks and hotspots for the CPU



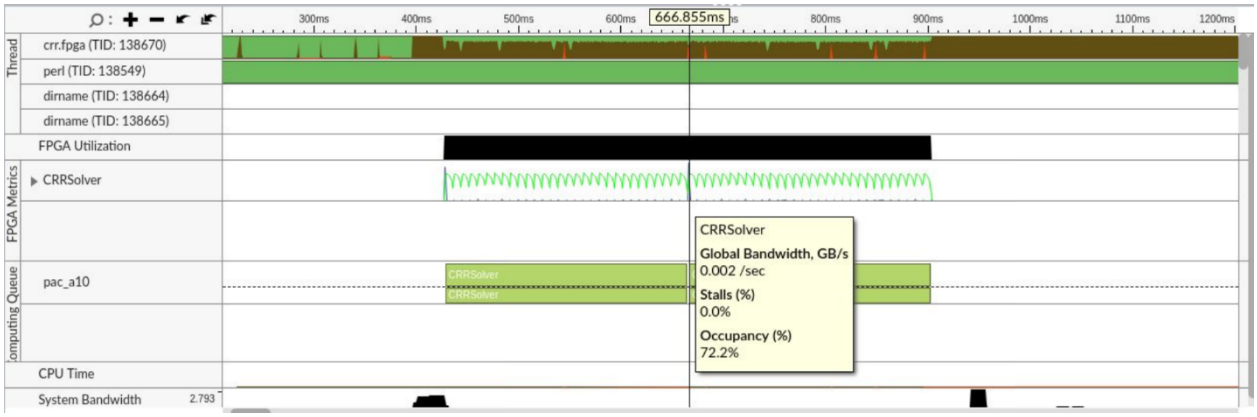
Switch to the **Bottom-up** window to see detailed information at the kernel level including:

- Stalls
- Occupancy
- Data transfer size
- Average bandwidth for transferred data



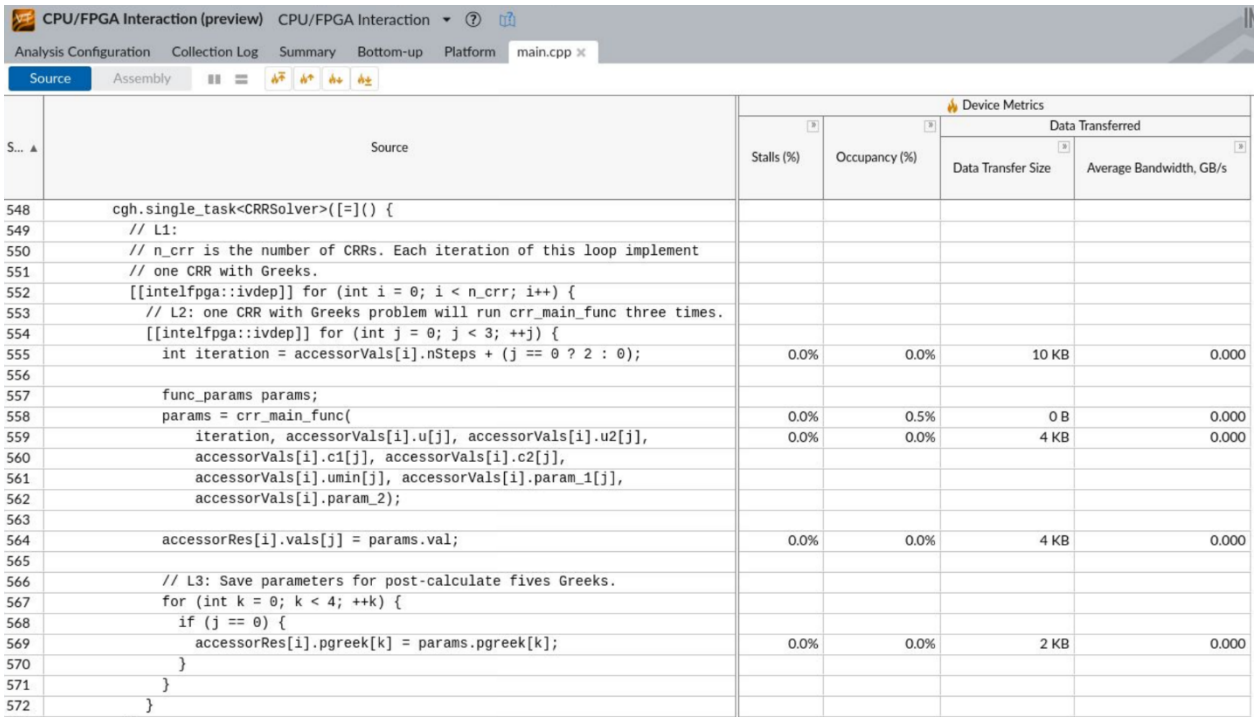
Use the timeline view to see these details about kernel instances:

- Start/end times
- Overtime stalls
- Occupancy
- Bandwidth metrics



In the **Bottom-up** window, right-click on a kernel and select **View Source** from context menu.

This opens the **Source View**, where you can see metrics for specific kernel source lines.



See Also

CPU/FPGA Interaction Analysis

Intel FPGA SDK for OpenCL Pro Edition: Best Practices Guide

Profiling Hardware Without Intel Sampling Drivers

Use this collection of recipes to set up Linux Perf*-based performance profiling with Intel® VTune™ Profiler in the driverless mode. Understand benefits and identify workarounds for possible limitations.*

Content expert: Jeffrey Reinemann , Alexey Bayduraev

Intel processors provide performance monitoring unit (PMU) events that you can employ to analyze how effectively your code utilizes hardware resources. VTune Profiler can collect and analyze PMU events for microarchitecture analysis types like:

- HPC Performance Characterization
- Memory Access
- Microarchitecture Exploration

If your analysis requires a smaller sampling interval, you can also configure the Hotspots and Threading analysis types to use the PMU event-based sampling instead of the default user-mode sampling.

For PMU event-based analysis, VTune Profiler uses Intel sampling drivers that require administrative privileges to install them on a target system. If you do not have administrative privileges or your environment does not allow third-party drivers to be injected to the systems, VTune Profiler cannot access PMU events via Intel sampling drivers and determine hardware performance bottlenecks. For those cases, VTune Profiler has adopted hardware performance monitoring capabilities through a built-in Linux Perf performance monitoring system.

VTune Profiler enables the hardware event-based sampling analysis in the Perf driverless mode when these conditions are met:

- Intel sampling drivers cannot be installed (for example, if installed without root privileges).
- Collection with stacks is selected with a non-zero (unlimited) stack size and the requirements for driverless collection are satisfied.
- The option to use driverless collection is enabled and the requirements for driverless collection are satisfied.

VTune Profiler extends driverless support for Linux Perf. This provides the profiling functionality, a collection overhead, and a trace size competitive with the solution based on using Intel sampling driver. However, VTune Profiler capabilities in the driverless mode depend on your Linux OS configuration. There may be some limitations. The examples in this recipe describe those limitations.

NOTE

- To enable the Perf driverless collection to match all the hardware profiling functionality provided with Intel drivers, you must have administrative privileges to configure the system options described below.
- To check the collector type used for the analysis (Perf or Intel sampling driver (SEP)), see the **Collection and Platform Info** section of the **Summary** window.

- **INGREDIENTS:**

  VTune Profiler - Run in driverless mode after meeting these prerequirements.

  - **Core and uncore events**. All hardware event-based collections in VTune Profiler use core PMU events. Some of them such as Memory Access and IO analysis types require access to uncore events that enable collecting metrics like DRAM bandwidth, QPI/UPI bandwidth, PCI bandwidth, and others.
  - **Perf for Linux kernel 2.6.32 and higher.** PMU events are exposed by Linux kernel through `/sys/bus/event_source/devices/cpu` and `/sys/bus/event_source/devices/uncore_*` directories. Empty directory content may indicate that the system configuration does not support PMU event collection. In this case, either update the OS or install the Intel sampling driver.

    NOTE In hybrid architectures with performance cores (P-Cores) and efficient cores (E-Cores), P-Core events are exposed through `/sys/bus/event_source/devices/cpu_core`. E-Core events are exposed through `/sys/bus/event_source/devices/cpu_atom`.

  - `/proc/sys/kernel/perf_event_paranoid` value is equal to or less than 1.

    NOTE When you run the Microarchitecture Exploration analysis on Intel® microarchitecture codenamed Skylake (or Skylake server), you must use Linux kernel version 4.3 or newer because front end event collection is necessary for this purpose. To verify if your system can support this analysis, see if the `frontend` file located in `/sys/devices/cpu/format/frontend`.

- **RECIPES FOR LIMITATIONS:**
  - Enable system wide or user process profiling
  - Enable core and uncore event collection
  - Enable multi-process profiling
  - Profile a large number of PMU events on multi-core systems
  - Enable stack sampling
  - Collect context switches
  - Resolve symbols for kernel functions
  - Avoid resource contention with the NMI Watchdog

- Reduce collection overhead
- Enable using driverless mode when required
- Enable profiling capabilities for the group

NOTE Run the `vtune-self-checker.sh` script provided with VTune Profiler to validate product capabilities on your analysis system. The script runs a representative set of analysis types on a stable benchmark and informs you on limitations the VTune Profiler encountered on the system. The recommendations in the diagnostics can help you configure your system properly for the driverless Perf collection or help you install the Intel sampling driver if the system configuration cannot help. To run the script, enter:

```
<vtune_install_dir>/bin64/vtune-self-checker.sh
```

Enable System Wide or User Process Profiling

Analysis types: All analyses

Concepts: *System-wide analysis* collects performance information about all processes running on the system, including system services and so on.

Driverless mode limitations: Additional configuration is required to enable system-wide or user process profiling.

To enable system-wide analysis in the driverless mode:

1. Configure a VTune Profiler project and from the **WHAT** pane select either the **Profile System** target or the **Launch Application** target with the **Analyze system-wide** option enabled.
2. Check the `/proc/sys/kernel/perf_event_paranoid` file value with the following command:

```
cat /proc/sys/kernel/perf_event_paranoid
```

   If the value is less than 1, VTune Profiler can proceed with the system-wide collection.
3. If the `perf_event_paranoid` value is equal to 1 (which limits the collection to user processes only) or more than 1 (which prevents VTune Profiler from using the Perf driverless mode), set the `perf_event_paranoid` value to 0 for the system-wide collection:

```
echo 0 > /proc/sys/kernel/perf_event_paranoid
```

NOTE
In some environments, `perf_event_paranoid` is regulated by the security policy. For more information about Linux Perf security requirements, see https://www.kernel.org/doc/html/latest/admin-guide/perf-security.html.

Intel sampling driver limitations: By default, the Intel sampling driver supports system-wide collection. But if it is built and loaded with the `--per-user` option, the collection will be limited to user processes only.

Enable Core and Uncore Event Collection

Analysis types: Memory Access, HPC Performance Characterization, and other analysis types based on uncore events

Core events can be collected both system wide and per user processes. To collect uncore events in the Perf driverless mode, enable system-wide analysis.

Driverless mode limitations:

- Memory Access analysis requires access to uncore events and will not run without ability to collect them. Other analysis types, like HPC Performance Characterization, will run but miss metrics based on uncore events such as DRAM Bandwidth, OPA Interconnect Bandwidth, and Packet Rate.

- Uncore collection in the driverless mode is not supported on Intel Atom® processors.

To collect uncore events in the driverless mode:

Set the `perf_event_paranoid` value to 0 to enable system-wide performance monitoring, which is a prerequisite for the uncore event collection.

Intel sampling driver limitations: None

Enable Multi-Process Profiling

Analysis types: all

By default, the Linux kernel limits the size of the memory available for capturing performance data by 518Kb. To know the current value, enter:

```
cat /proc/sys/kernel/perf_event_mlock_kb
```

Driverless mode limitations: For some parallel applications (for example, MPI applications with multiple ranks per node) in the user process collection mode on multi-core systems (>64 logical cores), the limit of 518Kb tends to be reached and the data collection will not be available.

To enable multi-process profiling on a multi-core system:

Set the `perf_event_paranoid` value to 0 to enable system-wide performance monitoring.

Intel sampling driver limitations: none. Any number of processes with default settings can be profiled.

Profile a Large Number of PMU Events on Multi-Core Systems

Analysis types: Microarchitecture Exploration

Driverless mode limitations: Linux Perf allocates file descriptors for every configured PMU event on each CPU. So, on a multi-core system with a long events list used by such analyses as Microarchitecture Exploration, this limit is easily reachable and can prevent the collection in the driverless mode.

To support profiling a large number of PMU events in the driverless mode:

1. Check the limit of opened files:

```
ulimit -n
```

2. If required, increase the limit in the `/etc/security/limits.conf` file. To do this, you must have administrator privilege. Increase the limit by adding or changing these lines (particular numbers are chosen as examples):

```
* soft nofile 65535
* hard nofile 65535
```

3. If you increased the limit in step 2, log out of the shell or close it and reopen a secure shell connection. Log back in.

> **NOTE** With administrator privilege, you can set the limit for a specific user. The change should be visible when the user logs in again.

For more information on using the `limits.conf` file, see http://man7.org/linux/man-pages/man5/limits.conf.5.html.

Intel sampling driver limitations: None.

Enable Stack Sampling

Analysis types: Hotspots (Hardware Event-Based Sampling mode), Threading (Hardware Event-Based Sampling and Stack Stitching mode), HPC Performance Characterization (**Collect stacks** option enabled), GPU Compute/Media Hotspots (**Collect stacks** option enabled).

Driverless mode limitations:

- Default 1024 byte stack size may not be enough for a full stack unwinding if a function intensively allocates data on the stack. This may lead to `[Skipped stack frame(s)]` displayed in the collected data.
- Linux kernel versions older than 3.7 support only frame-pointer (FP) based stack unwinding. This means that VTune Profiler can provide no stacks for binaries built without frame-pointer (`-fomit-frame-pointer` compiler option), as well as no Glibc stacks since Glibc is built without frame-pointers.

To avoid issues with stack unwinding in the driverless mode:

Increase the stack size. For example:

```
vtune -collect hotspots -knob sampling-mode=hw -knob enable-stack-collection=true -knob stack-size=2048 <application>
```

Otherwise, switch to the Intel sampling driver and set the **Stack size** option to 0 (unlimited value).

> **NOTE**
> Stack sampling collection with the Intel sampling driver depends on the kernel implementation. The collection requires an update for a new kernel version, which may bring additional product maintenance cost. To reduce this cost, VTune Profiler uses the driverless mode for all analysis types with stacks collection enabled, even when the Intel sampling driver is loaded. If you need to switch to the Intel sampling driver for stack sampling collection, do one of the following:
>
> - Create a custom analysis type and disable the **Enable driverless collection** option
> - Use the corresponding command line configuration
>
> ```
> vtune -collect-with runsa -knob enable-driverless-collection=false -knob event-config=<event-list> <application>
> ```

Intel sampling driver limitations: No limitation for the stack unwinding since the Intel sampling driver uses a different algorithm of call stack collection. The driver may require an update if your kernel version is newer than the latest kernel version supported by VTune Profiler.

Collect Context Switches

Analysis types: Threading

Concepts: *Context switch collection* helps expose metrics based on thread Inactive Wait time resulted from either synchronization or thread preemption.

Driverless mode limitations: Linux Perf collects context switches from kernel version 4.3 and higher. Identification of the context switch reason (synchronization or preemption) is available from kernel version 4.17. For older kernel versions, VTune Profiler switches the collection to the Intel sampling driver if it is available on the system.

Intel sampling driver limitations: none.

Resolve Symbols for Kernel Functions

Analysis types: All

Driverless mode limitations: Additional manual configuration of the `kptr_restrict` file is required.

To associate performance data with kernel function names:

Set the `kptr_restrict` configuration file value to 0 as a system administrator:

```
echo 0 > /proc/sys/kernel/kptr_restrict
```

Setting the value to 1 limits the file name resolution to user-level modules.

Intel sampling driver limitations: None. The Intel driver resolves kernel symbols if the `/boot/System.map-<kernel_version>` file is accessible for reading or `/proc/sys/kernel/kptr_restrict` is set to 0.

Avoid Resource Contention with the NMI Watchdog

Analysis types: all

Driverless mode limitations: NMI watchdog (a hard lockup detector) utilizes one CPU performance counter register that becomes unavailable for Linux Perf. This can increase the number of multiplexing groups and, as a result, impact the accuracy of statistical sampling data.

To improve the accuracy of analysis runs with long events lists in the driverless mode:

Disable the NMI watchdog, using administrative privileges:

```
echo 0 > /proc/sys/kernel/nmi_watchdog
```

When the driverless Perf collection is complete, you can re-enable the NMI watchdog (using the administrative privileges):

```
echo 1 > /proc/sys/kernel/nmi_watchdog
```

Intel sampling driver limitations: none. Intel driver automatically stops the NMI watchdog for the collection time to avoid this problem with data accuracy.

Reduce Collection Overhead

Analysis types: all

Driverless mode limitations: Linux Perf collection may incur an overhead on CPU intensive applications, since it fully loads all CPUs.

To reduce the collection overhead in the driverless mode:

- To reduce the trace size for stack sampling collections, the VTune Profiler uses a Linux Perf trace compression, which may introduce an additional overhead. To avoid this, disable the trace compression with the `-run-pass-thru` option:

  ```
  vtune -collect hotspots -knob sampling-mode=hw -knob enable-stack-collection=true -run-pass-thru=--perf-compression=0 <application>
  ```

  This can reduce collector overhead in rare cases, but the trace size increases dramatically.
- In some real-time and telecom applications, the default per-CPU trace collection mode can cause collection overhead. To overcome this, disable the per-CPU trace collection mode with the `-run-pass-thru` option:

  ```
  vtune -collect hotspots -knob sampling-mode=hw -knob enable-stack-collection=true -run-pass-thru=--perf-threads=none <application>
  ```
- Set the limit of CPU time consumption by Linux Perf collector. For example, for a 10% limit, use the following command (with administrative privileges):

  ```
  echo 10 > /proc/sys/kernel/perf_cpu_time_max_percent
  ```

  This can drop the sampling frequency and statistical accuracy to reach the limit.

Intel sampling driver limitations: none.

Enable Using Driverless Mode When Required

VTune Profiler uses the Intel sampling driver if it is loaded in all cases except for the stack sampling collection. To make the VTune Profiler use the driverless Perf mode for sampling without stacks, create a custom analysis type and select the **Enable driverless collection** option in the GUI, or set the command line `knob` value to `enable-driverless-collection=true` as follows:

```
vtune -collect-with runsa -knob enable-driverless-collection=true -knob event-config=<event-
list> <application>
```

The option is available starting with the VTune Amplifier 2019 Update 4.

Enable Profiling Capabilities for the Group

Driverless mode limitations: Setting the `perf_event_paranoid` option to a lower value could be inappropriate as this option applies to all users. Instead, you could set Linux capabilities to a specific user group and binary.

To enable capabilities for perf tool:

> **NOTE**
> The required capabilities cannot be assigned for a file system mounted with the `nosuid` option, or if the file system does not support extended file attributes.

- For Linux kernel versions older than 5.8, use `CAP_SYS_ADMIN`. To set up this configuration, run the `vtune-set-perf-caps.sh` script with this parameter:

  ```
  vtune-set-perf-caps.sh -v cap_sys_admin
  ```
- For Linux kernel versions 5.8 and newer, use `CAP_PERFMON`. To set up this configuration, run the `vtune-set-perf-caps.sh` script with this parameter:

  ```
  vtune-set-perf-caps.sh -v cap_perfmon
  ```

  Alternatively, you can set up this configuration manually:

  1. Create a `vtune` group for privileged `amplxe-perf` users.
  2. Assign the `vtune` group to the Perf tool executable.
  3. Restrict access to the executable to only those users who are in the `vtune` group.

```
# cp amplxe-perf amplxe-perf-priv
# groupadd vtune
# chgrp vtune amplxe-perf-priv
# chmod o-rwx amplxe-perf-priv
```

  4. Assign the required capabilities to the Perf tool executable.

```
# setcap -v "cap_perfmon,cap_sys_ptrace,cap_syslog=ep" amplxe-perf-priv
# getcap amplxe-perf-priv
amplxe-perf-priv = cap_sys_ptrace,cap_syslog,cap_perfmon+ep
```

  If the installed libcap does not support `cap_perfmon`, use `38` instead:

```
# setcap "38,cap_sys_ptrace,cap_syslog=ep" amplxe-perf-priv
# getcap amplxe-perf-priv
amplxe-perf-priv = cap_sys_ptrace,cap_syslog,38+ep
```

For more information on Linux capabilities, see https://man7.org/linux/man-pages/man7/capabilities.7.html.

For more information on `perf_event` access control, see https://www.kernel.org/doc/html/latest/admin-guide/perf-security.html

See Also

Sampling Drivers

Hardware Event-based Sampling Collection with Stacks

Enable Linux* Kernel Analysis

Profiling MPI Applications

Use Intel VTune Profiler to identify imbalances and communication issues in MPI-enabled applications.

Content experts: Rupak Roy, Xiao Zhu

- INGREDIENTS
- DIRECTIONS:

  1. Build the Application
  2. Establish Overall Performance Characteristics
  3. Configure and Run the HPC Performance Characterization Analysis
  4. Analyze Results using the VTune Profiler GUI
  5. [Optional] Generate a Command Line Run from the VTune Profiler GUI
  6. [Optional] Analyze Results with a Command Line Report
  7. [Optional] Selective Code Area Profiling

Ingredients

This section lists the hardware and software tools used for the performance analysis scenario.

- **Application:** `heart_demo` sample application
- **Tools:**
  - Intel® C++ Compiler
  - Intel® MPI Library 2021.11
  - Intel VTune Profiler 2024.0 or newer
  - Intel VTune Profiler - Application Performance Snapshot

  NOTE

  - Get a free download of the Intel MPI Library from https://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library.html.
  - Download the latest version of VTune Profiler from the product download page.

- **Operating system:** Linux*
- **CPU**: Intel® Xeon® Platinum 8480+ Processor (formerly code named Sapphire Rapids)

Build the Application

Build your application with debug symbols so Intel VTune Profiler can correlate performance data with your source code and assembly.

1. Clone the application GitHub repository to your local system:

```
git clone https://github.com/CardiacDemo/Cardiac_demo.git
```

2. Set up the Intel C++ Compiler and Intel MPI Library environment:

```
source <compiler_install_dir>/oneapi/setvars.sh
```

3. In the root level of the sample package, create a build directory and open it:

```
mkdir build
cd build
```

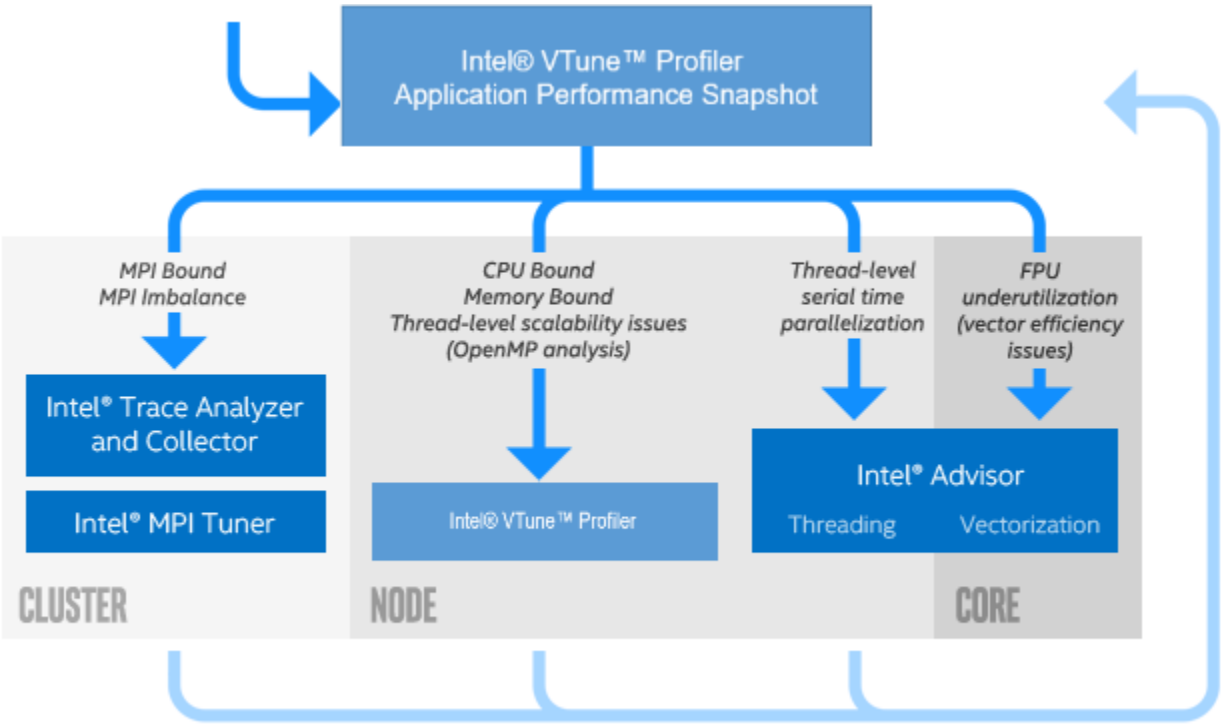
4. Build the application:

```
7.    mpiicpx ../heart_demo.cpp ../luo_rudy_1991.cpp ../rcm.cpp ../mesh.cpp -g -o heart_demo -O3
-std=c++17 -qopenmp -parallel-source-info=2
```

The executable `heart_demo` should be present in the current directory.

Establish Overall Performance Characteristics

Start tuning your MPI application by examining a snapshot of its performance, collected by Application Performance Snapshot in VTune Profiler. With this snapshot, you can understand the general properties of your application. Then focus on problematic areas using appropriate tools.



We begin by preparing a performance snapshot on a set of dual socket nodes using the Intel® Xeon® Scalable processor (code named Sapphire Rapids). This example uses Intel® Xeon® Platinum 8480+ Processor with 24 cores per socket. This processor configures the run to have 4 MPI ranks per node and 12 threads per rank. Modify the specific rank and thread counts in this example to match your own system specification.

To obtain a performance snapshot on four nodes, run this command in an interactive session or in a batch script :

```
export OMP_NUM_THREADS=12
mpirun -np 16 -ppn 4 aps ./heart_demo -m ../mesh_mid -s ../setup_mid.txt -t 100
```

When the analysis is complete, you can find profiling data in a directory named `aps_result_YYYYMMDD`, where the date of collection is included in YYYY/MM/DD format.

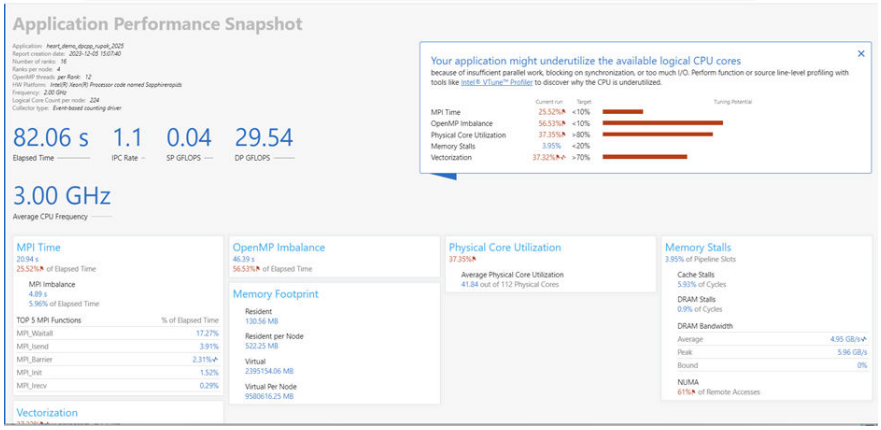
For example, to produce a single page HTML snapshot of the results collected on December 5 2023, type:

```
aps --report ./aps_result_20231205
```

The `aps_report_YYYYMMDD_<stamp>.html` file is created in your working directory, where the *<stamp>* number is used to prevent overwriting existing reports. The report contains information on overall performance, including:

- MPI and OpenMP* imbalance
- Memory footprint and physical core utilization
- floating point throughput

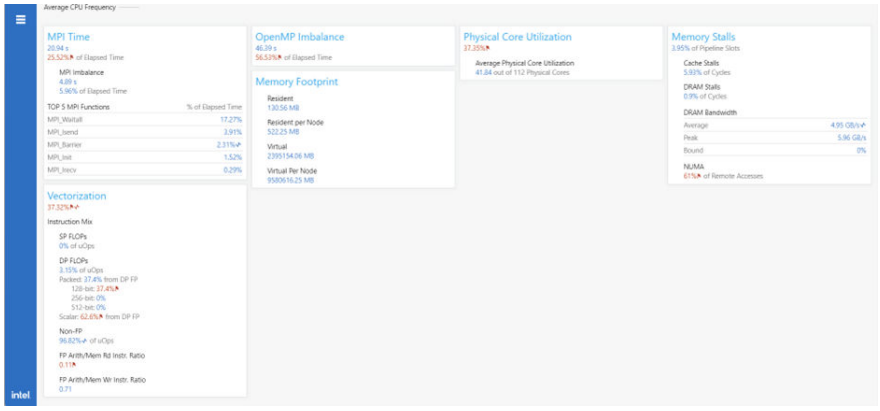
A note at the top of the report highlights the main areas of concern for the application.



The snapshot indicates that this application is bound overall by MPI communication. The application also suffers from:

- OpenMP* imbalance
- Physical core utilization
- Vectorization issues

The **MPI Time** section provides additional details, such as MPI imbalance and the top MPI function calls used. From this section, it appears that the code uses mainly point to point communication and that the imbalance is moderate.



This snapshot result points to complex issues in the code. To continue investigating the performance issues and isolate the problems, let us run the HPC Performance Characterization analysis in VTune Profiler next.

Configure and Run the HPC Performance Characterization Analysis

Most clusters are setup with login and compute nodes. Typically a user connects to a login node and uses a scheduler to submit a job to the compute nodes, where it executes. In a cluster environment, the most practical way to run VTune Profiler to profile an MPI application is by using the command line for data collection and the GUI for performance analysis, once the job has completed.

To report MPI-related metrics in a distributed environment, type:

`<mpi launcher> [options] vtune [options] -r <results dir> -- <application> [arguments]`

> **NOTE**
>
> - You can use the above command can be used in an interactive session or included in a batch submission script.
> - You must specify the results directory for MPI applications.
> - If you are not using the Intel MPI Library, add `-trace-mpi` to the above command .

Follow these steps to run the **HPC Performance Characterization** analysis in VTune Profiler from the command line:

1. Prepare your environment by sourcing the VTune Profiler files. For a default installation using the bash shell, use this command:

```
source /opt/intel/vtune_Profiler/vars.sh
```

2. Collect data for the `heart_demo` application using the `hpc-performance` analysis. The application uses both OpenMP and MPI. The application execution uses the configuration described earlier, with 16 MPI ranks over a total of 4 compute nodes using the Intel MPI Library. This example is run on Intel® Xeon® Platinum 8480 Processors and uses 12 OpenMP threads per MPI rank:

```
export OMP_NUM_THREADS=12
mpirun -np 16 -ppn 4 vtune -collect hpc-performance -r vtune_mpi -- ./heart_demo -m ../mesh_mid -
s ../setup_mid.txt -t 100
```

   The analysis begins and generates four output directories using the following naming convention: `vtune_mpi.<node host name>`.

> **NOTE**
> You can select specific MPI ranks to be profiled while running others simultaneously, but without collecting profiling data. For details, see Selective MPI Rank Profiling.

Analyze Results using the Intel VTune Profiler GUI

Open one of the collected results in the VTune Profiler user interface:

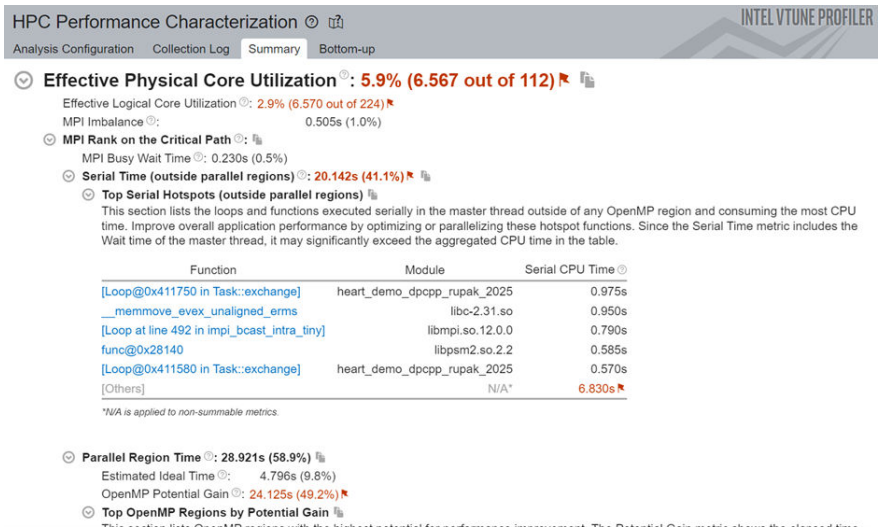
```
vtune-gui ./vtune_mpi.node_1
```

> **NOTE**
> To display the Intel VTune Profiler GUI, you need an X11 manager running on the local system or a VNC session connected to the system. Since each system is different, consult with your local administrator for a recommended method.

The result opens in Intel VTune Profiler and shows the **Summary** window. This window provides an overview of the application performance. Because `heart_demo` is an MPI parallel application, the **Summary** window shows MPI Imbalance information and details regarding the MPI rank in the execution critical path in addition to the usual metrics.

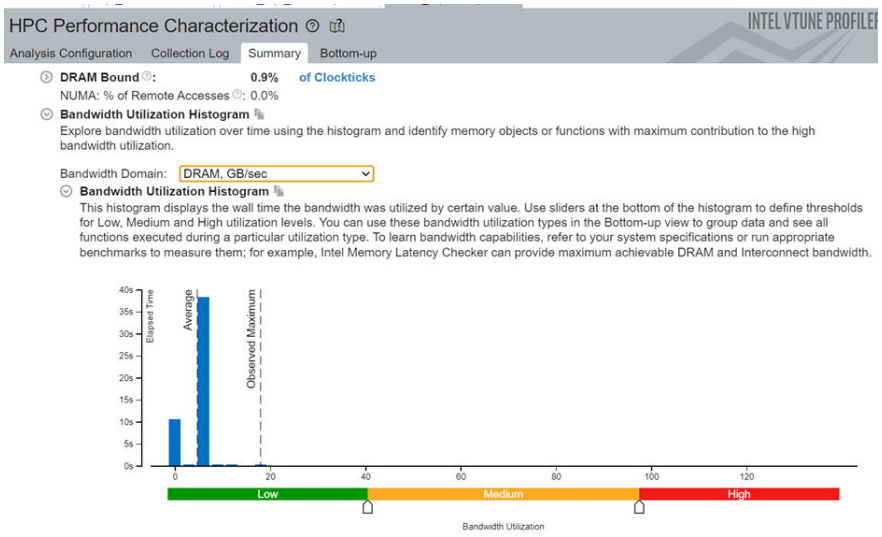
- **MPI Imbalance** is an average MPI busy wait time by all ranks on the node. The value indicates how much time could be saved if the balance was ideal.
- **MPI Rank on the Critical Path** is the rank with minimal busy wait time.
- **MPI Busy Wait Time** and **Top Serial Hotspots** are shown for the rank in the critical path. This is a good way to identify severe deficiencies in scalability since they typically correlate with high imbalance or busy wait metrics. Significant **MPI Busy Wait Time** for the rank on the critical path in a multi-node run could imply that the outlier rank is on a different node.

In our example, there is some imbalance and also a significant amount of time spent in serial regions of the code (not shown in the figure).

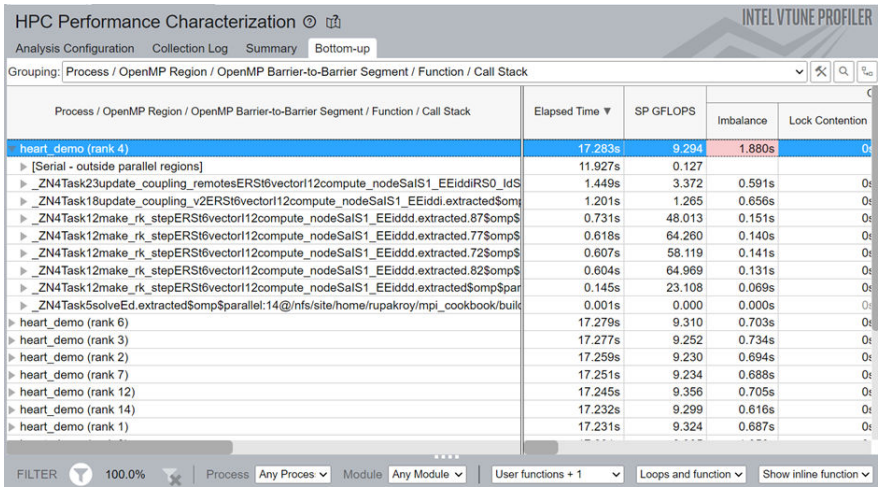


While you can collect profiles across nodes, the only way to view all MPI data is to load each node result independently. For detailed MPI traces, use Intel® Trace Analyzer and Collector.

In Intel VTune Profiler 2024.0 (and newer versions), the **Summary** window contains histograms of bandwidth utilization. The metrics show bandwidth and packet rate and indicate the percentage of the execution time for which the code was bound by high bandwidth or packet rate utilization. The histogram shows a maximum DRAM bandwidth utilization of 6 GB/s, which is low. This tells us that there is still room for improvement.

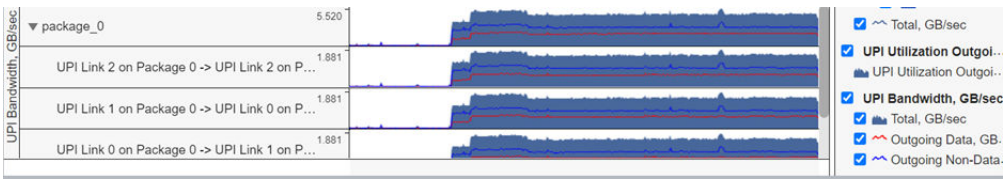


Switch to the **Bottom-up** tab to get more details. Set the **Grouping** to have **Process** at the top level. You should see this view:



Since this code uses both MPI and OpenMP, the Bottom-up window shows metrics related to both runtimes, in addition to the CPU and memory data. In our example, the OpenMP* Imbalance metric is highlighted in red. This hints that threading improvements could help performance.

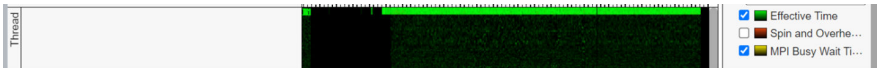
Review the execution timeline for several metrics at the bottom of the **Bottom-up** window, including DDR and MCDRAM bandwidth, as well as CPU time. The UPI bandwidth timeline for this code shows continuous utilization at a moderate bandwidth (the scale is in GB/s).



Of more interest is the detailed execution time per thread and the breakdown of these metrics:

- Effective Time
- Spin and Overhead Time
- MPI Busy Wait Times

The default view uses the Super Tiny settings to show all processes and threads together in a visual map of performance.



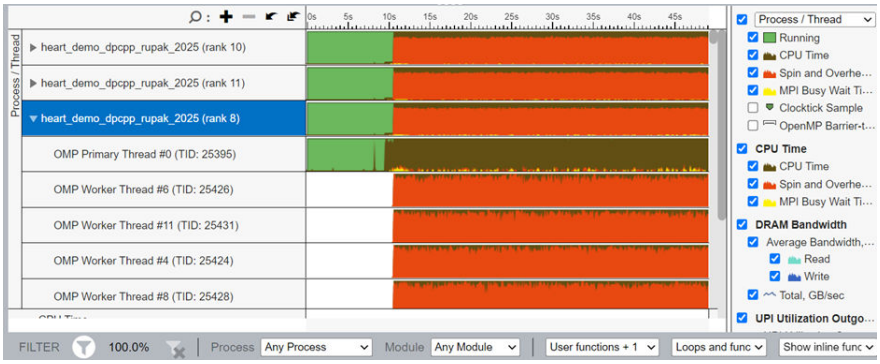
In this case you should see that there is little effective time in most of the threads (green) and that the amount of MPI overhead is also small (yellow). This points to potential issues in the threading implementation.

To investigate this further,

1. Right-click on the grey area to the left of the graph.
2. Select the **Rich** view for the band height.
3. To the right of the graph, group results by **Process/Thread**.

By selecting this grouping, you get better clarity with the roles of each MPI Rank and each thread. The top bar for each process shows the average result for all children threads. Below that average, each thread is listed with its own thread number and process ID.

In our example, the primary thread takes care of all MPI communication for each MPI rank. This behavior is common in hybrid applications. A significant amount of time is spent in MPI communication (yellow) in the first ten seconds of the execution, likely to set up the problem and distribute data. After that period, there is regular MPI communication, which matches the results observed in the Bandwidth Utilization timeline and the **Summary** report.



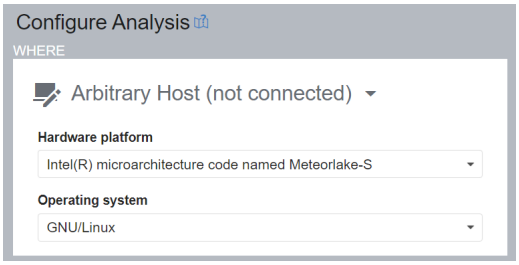
The high amount of spin and overhead (shown in red by default) is noticeable. This indicates issues with the way threading was implemented in the application.

1. At the top of the **Bottom-up** window, group the data by **OpenMP Region / Thread / Function / Call Stack**.
2. Apply the filter at the bottom of the window to show **Functions only**.
3. Expand the tree to see that the function *init_send_bufs* is only called by thread 0 and is responsible for the low performance observed.
4. Double click on a line to open the source code viewer.

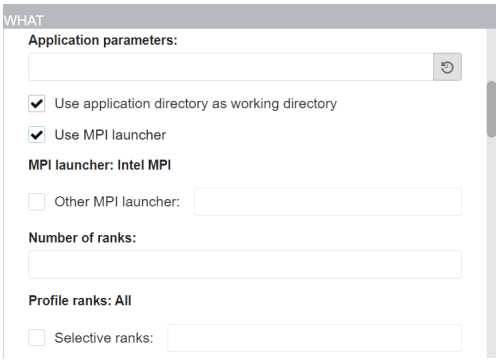
Generate a Command Line Run from the Intel VTune Profiler GUI (optional)

You can configure an analysis in Intel VTune Profiler using the GUI and then save the equivalent command to run the analysis directly from the command line. Use this feature for heavily customized profiles or for quickly building a complex command.

1. Open Intel VTune Profiler.
2. Click **New Project** or open an existing project.
3. Click **Configure Analysis**.
4. In the **Where** pane, select **Arbitrary Host (not connected)** and specify the hardware platform.

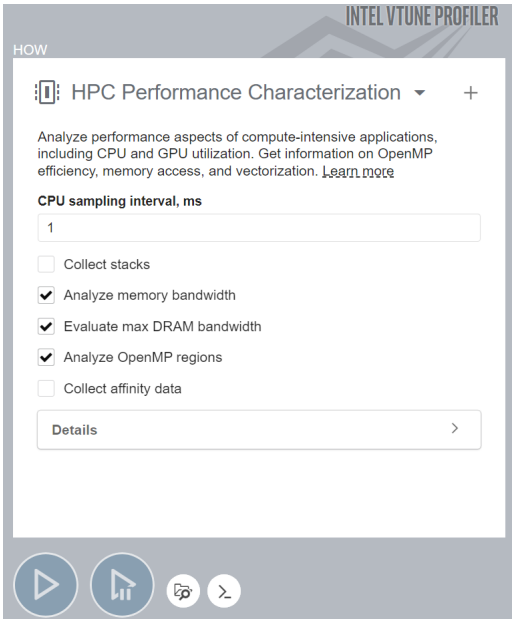


5. In the **What** pane:



   a. Specify the application.
   b. Set the parameters and working directory.
   c. Select the **Use MPI launcher** option and provide information related to the MPI run.
   d. [Optional] Choose particular ranks to profile.

6. In the **How** pane, change the default **Hotspots** analysis to **HPC Performance Characterization**. Customize the available options.



7. Click the



Command Line button at the bottom of the window. A pop-up window displays the equivalent command you should run to perform the customize analysis you just configured on the GUI. You can add additional MPI options to complete the command.

NOTE
For Intel MPI, the command line is generated in terms of the `-gtool` option. Use this option to simplify selective rank profiling syntax.

Analyze Results with a Command Line Report (optional)

Intel VTune Profiler provides informative command line text reports. For example, to obtain a summary report, run:

```
vtune -report summary -r ./results_dir
```

A summary of the results prints to the screen. Options to save the output directly to file and in other formats (csv, xml, html) are also available. For details on the full command line options, type **vtunel -help** in the command line or see Intel® VTune™ Profiler Command Line Interface.

Selective Code Area Profiling (optional)

By default, Intel VTune Profiler collects performance statistics for the whole application. The 2019.3 and newer versions of Intel VTune Profiler contain the ability to control data collection for MPI applications. There are several advantages to this capability:

- You can generate smaller result files.
- Result files process quickly.
- You can completely focus on a region of interest.

The region selection process is done using the standard `MPI_Pcontrol` function. Call `MPI_Pcontrol(0)` to pause data collection and call `MPI_Pcontrol(1)` to resume it again.

You can use the API together with the command line option `-start-paused` to exclude the application initialization phase. In this case, a `MPI_Pcontrol(1)` call should follow right after initialization to resume data collection. This method of controlling collection requires no changes in the application building process, unlike using ITT API calls, which require linking of a static ITT API library.

Additional Resources

- Intel Advisor Cookbook: Optimize Vectorization Aspects of a Real-Time 3D Cardiac Electrophysiology Simulation
- VTune Profiler Installation Guide for Linux
- Using Intel® Advisor and VTune Profiler with MPI
- Tutorial: Analyzing an OpenMP and MPI Application

Product and Performance Information

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Notice revision #20201201

Profiling Docker* Containers

Intel® VTune™ Profiler allows you to profile applications running in Docker containers, including profiling multiple containers simultaneously. This recipe guides you through the configuration of a Docker container and describes ways to use VTune Profiler to analyze one or multiple concurrently running containers. This recipe also utilizes the Java* Code Analysis capabilities of VTune Profiler.*

- INGREDIENTS
- DIRECTIONS:

1. Install and configure a Docker* Container
2. Run Hotspots Analysis with Hardware Event-Based Sampling for Target in Container
3. Analyze Data Collected for Target in Container
4. Run Hardware Event-Based Hotspots Analysis With VTune Profiler and Target Running in Same Container
5. Run Profile System Analysis for Host Target From Container

Ingredients

This section lists the hardware and software tools used for the performance analysis scenario.

- **Application**: `MatrixMultiplication`. This Java application is used as a demo and is not available for download.
- **Tools**: Intel VTune Profiler 2021.2.0 - Hotspots analysis with Hardware Event-Based Sampling.

> **NOTE**
>
> - Starting with the 2020 release, Intel® VTune™ Amplifier has been renamed to Intel® VTune™ Profiler.
> - Most recipes in the Intel® VTune™ Profiler Performance Analysis Cookbook are flexible. You can apply them to different versions of Intel® VTune™ Profiler. In some cases, minor adjustments may be required.
> - Get the latest version of Intel® VTune™ Profiler:
>   - From the Intel® VTune™ Profiler product page.
>   - Download the latest standalone package from the Intel® oneAPI standalone components page.

- **Linux container runtime**: docker.io.
- **Operating system**: Ubuntu* 20.04 based on Linux* kernel version 5.4 or newer.
- **CPU**: Intel® microarchitecture code named Skylake or newer

Install and Configure a Docker* Container

Prerequisites:

- Install a Docker container. See official Docker documentation available at https://docs.docker.com/install/linux/docker-ce/ubuntu/.
- Check if your user is in the docker group. Otherwise, use `sudo` for the docker.

1. Pull a docker image that you want to use.

```
host> docker pull ubuntu:latest
```

2. Run the container. Keep it running using the `-t` and `-d` options.

```
host> docker run -td ubuntu:latest
```

3. To analyze Docker containers with VTune Profiler using User-Mode Sampling or Memory Consumption analyses, make sure to enable `ptrace` support.

```
host> docker run --cap-add CAP_SYS_PTRACE --name=test_container -td ubuntu:latest
```

4. If you want to analyze Docker containers with VTune Profiler using Hardware Event-Based Sampling analysis, enable the `CAP_SYS_ADMIN` capability.

```
host> docker run --cap-add CAP_SYS_ADMIN --name=test_container_0 -td ubuntu:latest
```

   You can also launch the container in the privileged mode.

```
host> docker run --privileged --name=test_container_0 -td ubuntu:latest
```

5. Copy your Java application with the Java Virtual Machine (JVM) to the docker instance that is running.

```
host> docker cp openjdk-16_linux-x64_bin.tar.gz test_container_0:/var/local
host> docker cp MatrixMultiplication.java test_container_0:/var/local
```

6. Use the container name to get bash into this container in the background mode.

```
host> docker exec -it test_container_0 /bin/bash
```

7. Extract the `jdk` archive.

Run Hotspots Analysis with Hardware Event-Based Sampling for Target in Container

In this procedure, we run VTune Profiler on the host machine to profile a target in a docker container.

1. Run the Java application in the container.

```
container> cd /var/local
container> /var/local/jdk-16/bin/java -cp . MatrixMultiplication 2000 2000 2000 2000
```

2. On the host, run a system-wide analysis by starting the **Profile System** analysis:

```
host> cd /home/user/intel/oneapi/vtune/latest
host> source vtune-vars.sh
host> vtune -collect hotspots -knob sampling-mode=hw -knob enable-stack-collection=true -knob
stack-size=4096 --duration 60
```

NOTE You can also profile your application when it is running in a Docker container, using the **Attach to Process** target type.

```
host> vtune -collect hotspots -knob sampling-mode=hw -knob enable-stack-collection=true -
knob stack-size=4096 -target-process java
```

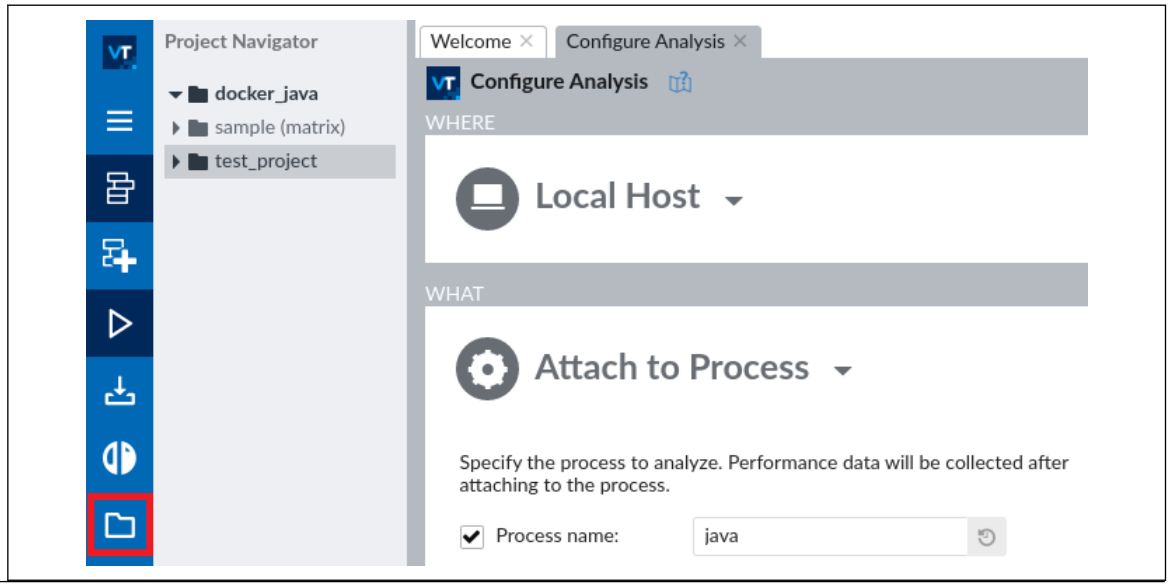
However, you cannot profile applications running in the container that are instrumented with ITT/JIT API.

Analyze Data Collected for Target in Container

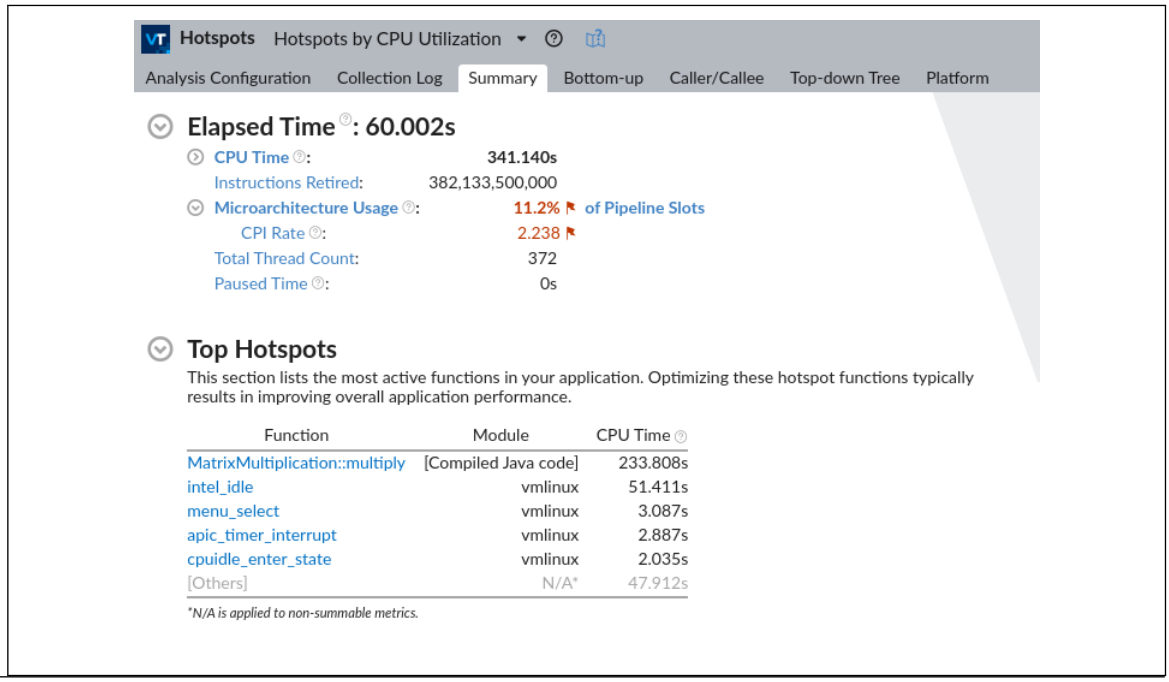
1. When the data collection is complete, start the VTune Profiler GUI.

```
host> vtune-gui
```

2. Create a project for the collected results, say `docker_java`.
3. Open the collected results.

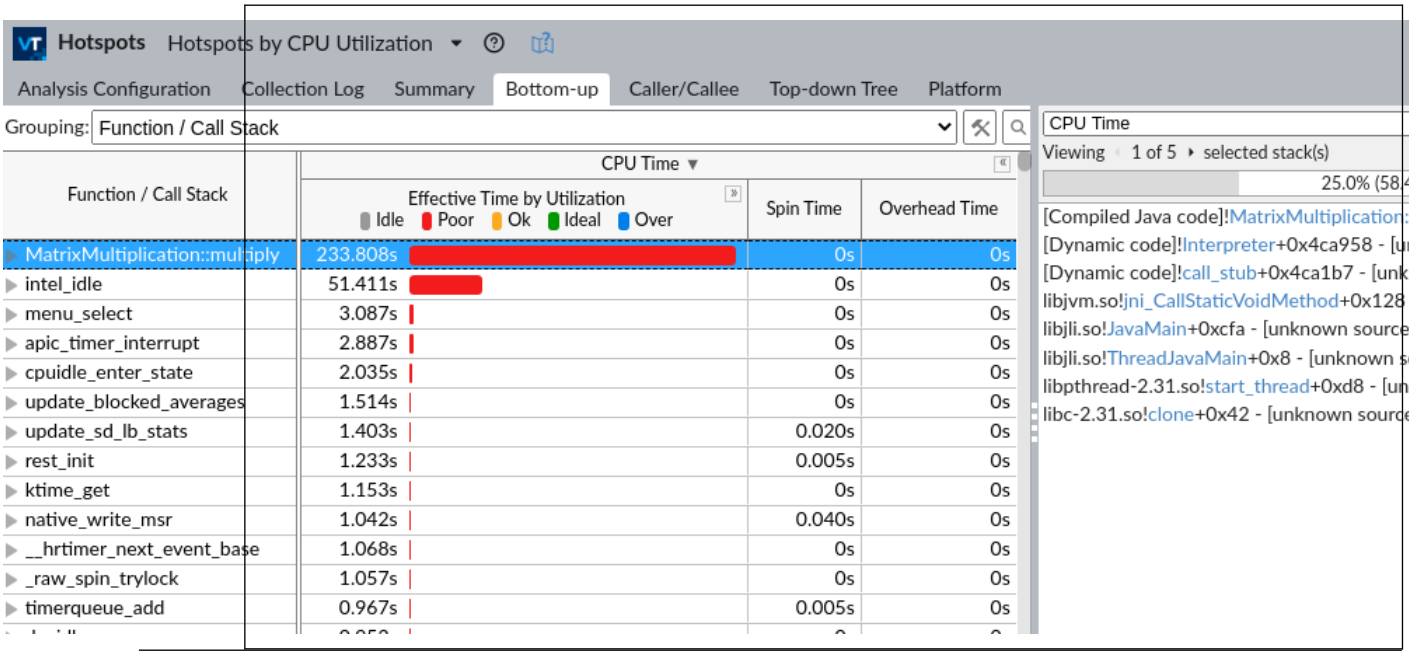


4. Review the results in the **Summary** tab of the Hotspots analysis.

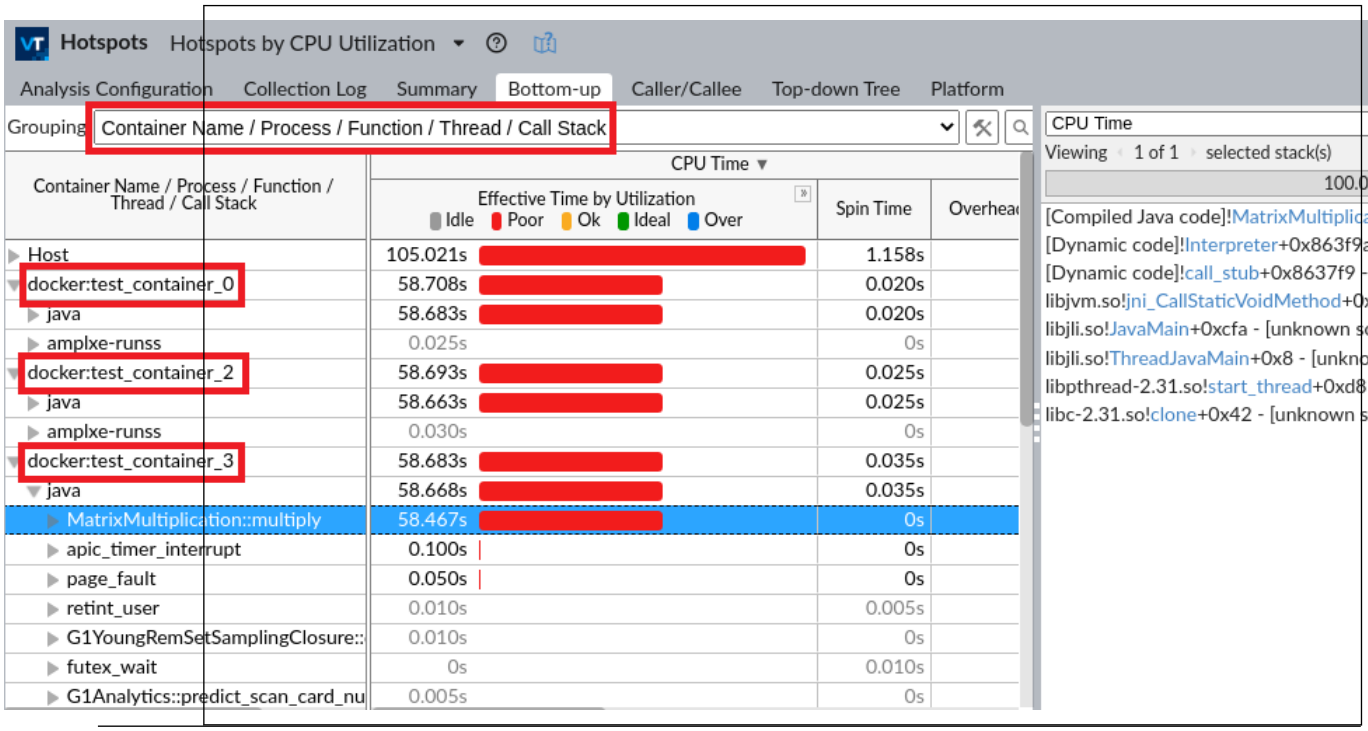


We infer from the **Top Hotspots** section that the `multiply` function of the target application consumed the most CPU time.

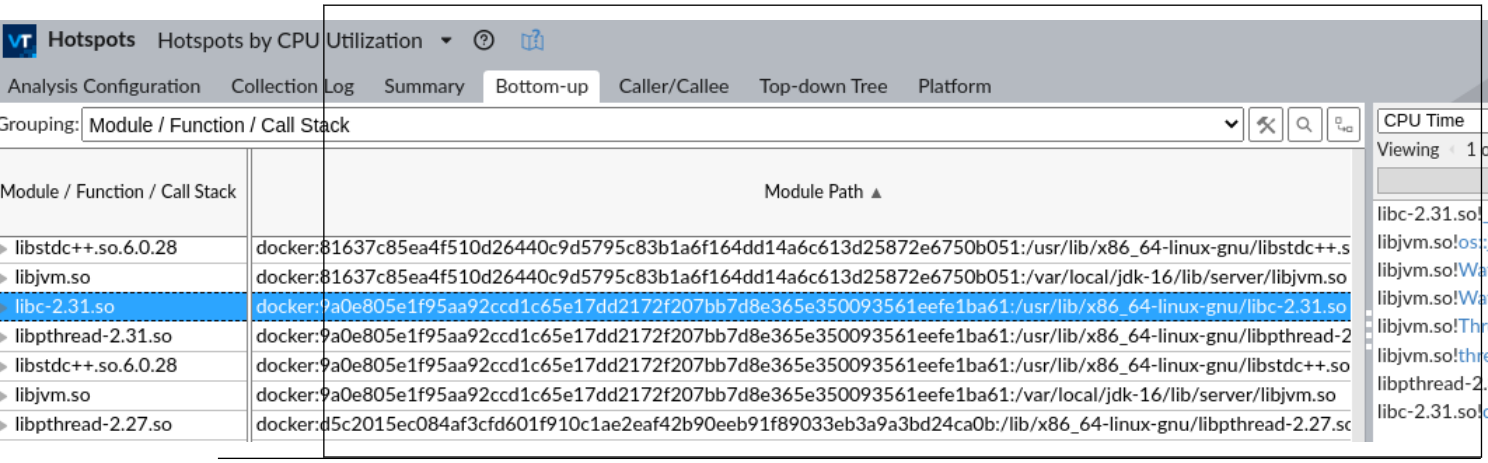
5. Click on the `multiply` function in the list to switch to the **Bottom-up** tab. View the stack flow for this hotspot function.



6. Double click on the hottest function in the table to identify the hotspot source code line for that function. You can then analyze the metric data collected for this line.
7. To view performance data for individual docker containers, select the **Container Name/Process/Function/Thread/Call Stack** grouping from the pull down menu. Identify containers by the docker prefix.



8. To view the performance data for system binaries that are running inside a container, select the **Module/Function/Call Stack** grouping. Locate the entries with the docker prefix in the **Module Path** column.



In this grouping mode, you can also view performance data for host system binaries and containerized system binaries simultaneously.

Run Hardware Event-Based Hotspots Analysis With VTune Profiler and Target Running in Same Container

1. Pull the docker image of `oneapi-basekit`.

```
host> docker pull intel/oneapi-basekit
```

2. Run the docker container with `CAP_SYS_ADMIN` capability to enable profiling from the container.

```
host> docker run -dt --name=my_oneapi_container --cap-add CAP_SYS_ADMIN intel/oneapi-basekit
```

3. Once the collection is complete, do one of the following: you can either or .
   - Copy and view the collected data outside the container. Exit this procedure.
   - Use VTune Profiler Server opened in the same container. Go to step 4.
4. Use VTune Profiler Server to view collected data.
   a. Publish a port outside the container by using `--publish`.

```
host> docker run -dt --name=my_oneapi_container --cap-add CAP_SYS_ADMIN --publish     7788:7788
intel/oneapi-basekit
```

   where

   `--publish 7788:7788` maps TCP port 7788 in the container to port 7788 on the host.

   b. Start VTune Profiler Server inside the container.

```
my_oneapi_container> vtune-backend --allow-remote-ui --web-port=7788 --enable-        server-
profiling &
```

   where

   `--allow-remote-ui` allows remote UI clients.

   `--web-port=7788` is the HTTP/HTTPS port for web UI and data APIs.

   `--enable-server-profiling` allows users to select the hosting server as the profiling target.

   `&` runs the command in the background.

   The `vtune-backend` command returns a URL that you can open outside the container. For example,

```
Serving GUI at https://b06036cef42c:7788?one-time-token=4db58f1ad7225e4dccca60573e4c1fd2
Serving GUI at https://172.17.0.8:7788?one-time-token=4db58f1ad7225e4dccca60573e4c1fd2
```

   c. On the host machine, open the URL reported by `vtune-backend` in a browser.

d. Change the port on the container (used by `vtune-backend`) to the port you specified when creating the container.

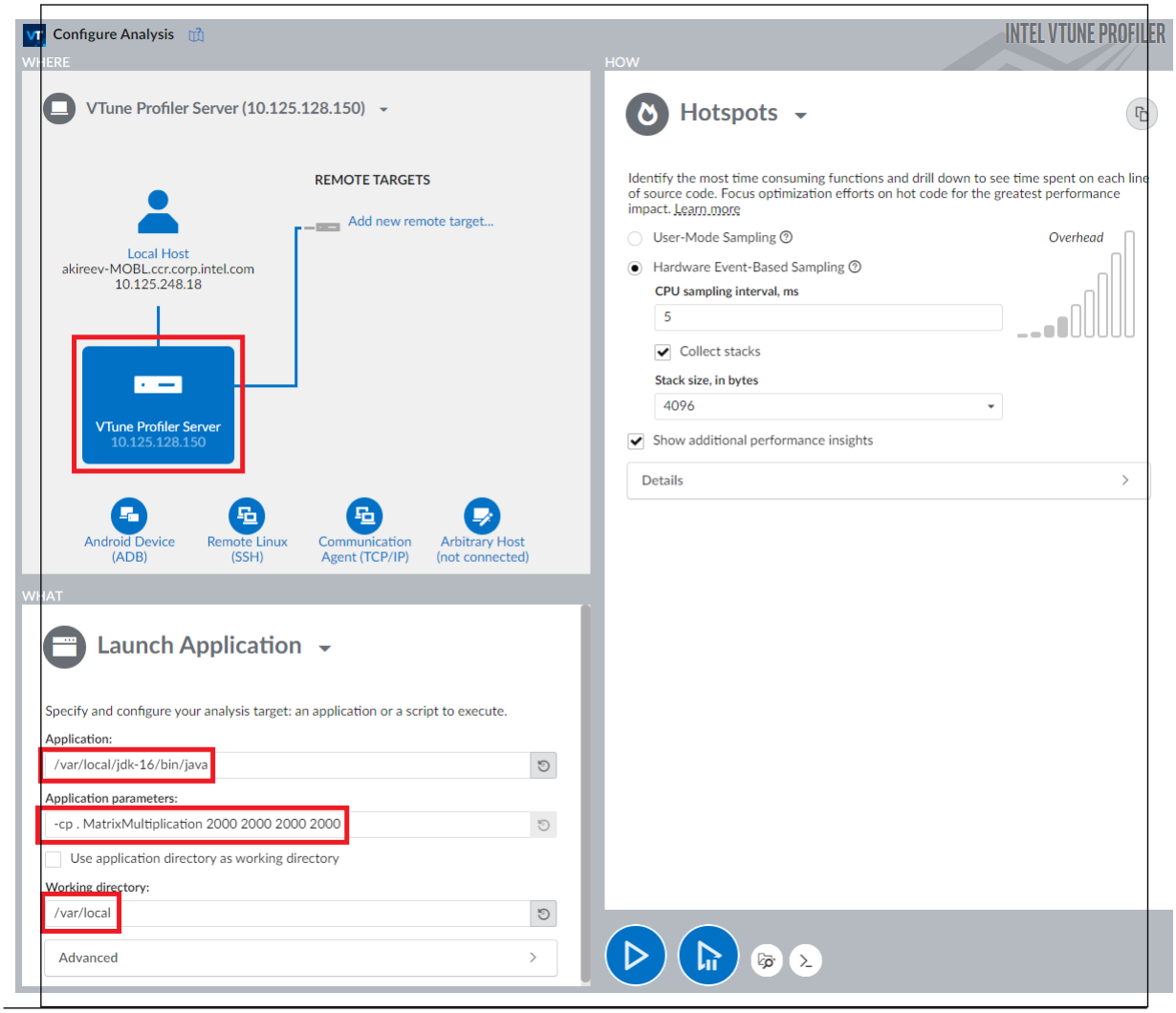
NOTE The IP address in this output is the IP address of the container. You can access this address only from the host where the container is running. To access VTune Profiler Server from outside the host, use the IP address or hostname of the external host.

e. Create a project, say `vtune_in_docker`.

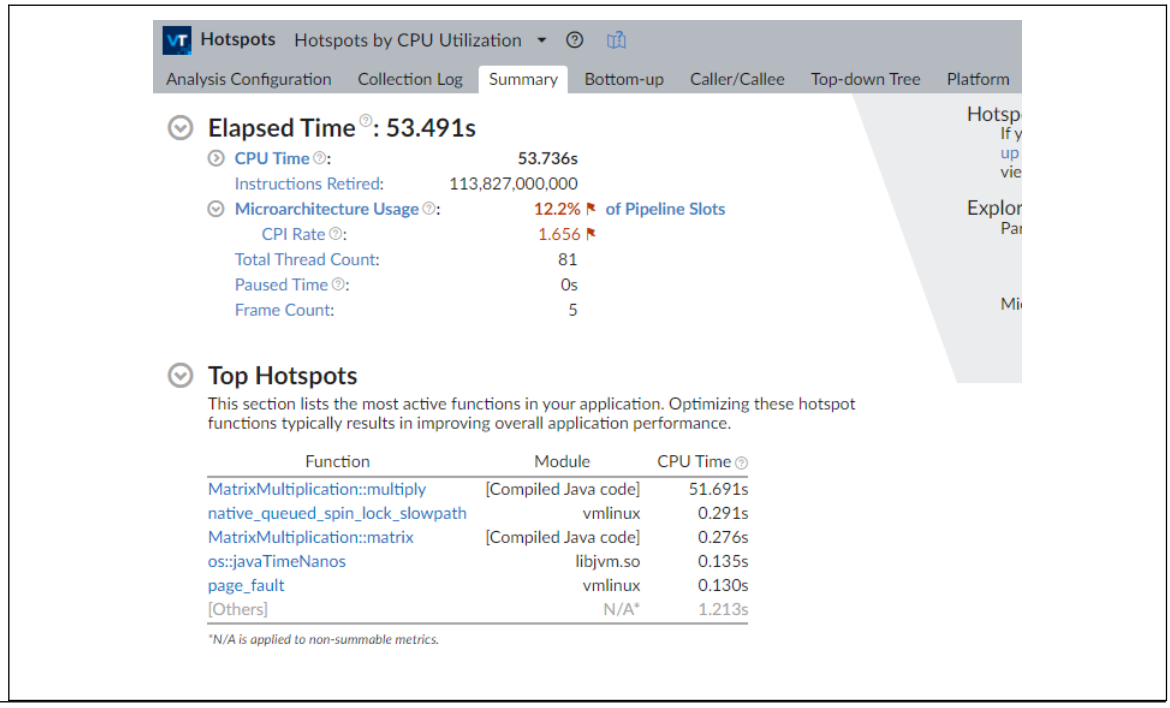
f. Copy your Java application to the host folder in the container or mount application.

```
host> docker cp openjdk-16_linux-x64_bin.tar.gz my_oneapi_container:/var/local/
host> docker cp MatrixMultiplication.java my_oneapi_container:/var/local/
```

g. You can run User-mode and Hardware Event-based Hotspots analysis in both **Launch** and **Attach to Process** modes within the container. For example, start the Hardware Event-based Hotspots analysis in **Launch** mode.



h. Once the analysis finishes, view results in the **Summary** tab.



Run Profile System Analysis for Host Target From Container

1. Start a docker container with `--pid=host` and `--cap-add CAP_SYS_ADMIN` options to collect data and with mounted host folders. Specify the binaries and symbols of the application for Function and Source level analysis of collected data.

```
host> docker run -dt --name=my_oneapi_container -v /host_path:/container_path --pid=host --cap-
add CAP_SYS_ADMIN --publish 7788:7788 intel/oneapi-basekit
```

   where:

   `-v /host_path:/container_path` mounts the host path "`/host_path`" inside the container path "`/container_path`".

   `--pid=host` sets the PID namespace of the host inside the container.

   `--publish 7788:7788` maps the TCP port 7788 in the container to port 7788 on the host.

2. To analyze the collected data, do one of the following:
   - Copy and view the collected data outside the container and on a different system. Exit this procedure.
   - Use VTune Profiler Server opened in the same container. Go to step 3.

3. To view results in VTune Profiler Server, start the server inside the container.

```
my_oneapi_container> vtune-backend --allow-remote-ui --web-port=7788 --enable-server-profiling &
```

   where

   `--allow-remote-ui` allows remote UI clients

   `--web-port=7788` sets the HTTP/HTTPS port for web UI and data APIs

   `--enable-server-profiling` allows users to select the hosting server as the profiling target

   `&` runs the command in the background

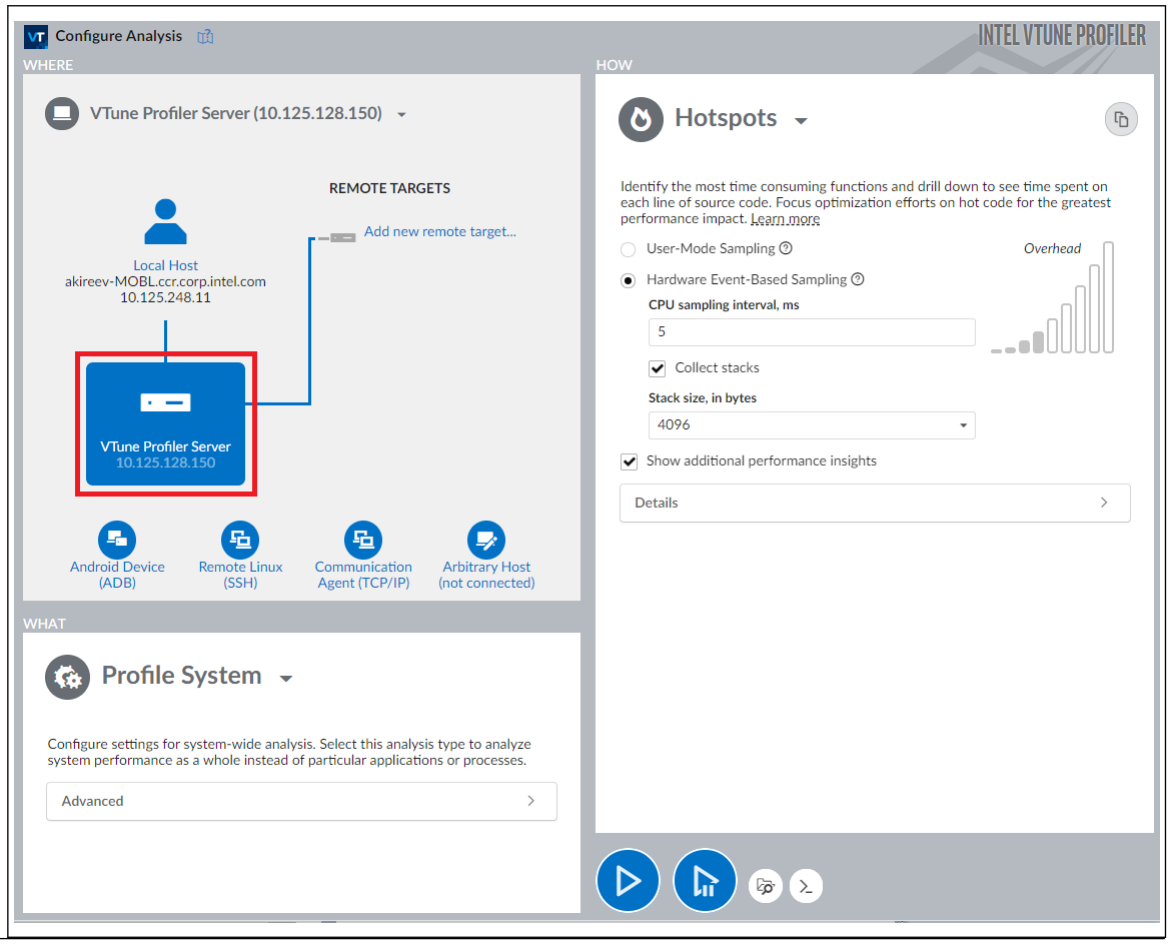
The `vtune-backend` command returns a URL that you can open outside the container. For example,

```
Serving GUI at https://b06036cef42c:7788?one-time-token=4db58f1ad7225e4dccca60573e4c1fd2
Serving GUI at https://172.17.0.8:7788?one-time-token=4db58f1ad7225e4dccca60573e4c1fd2
```

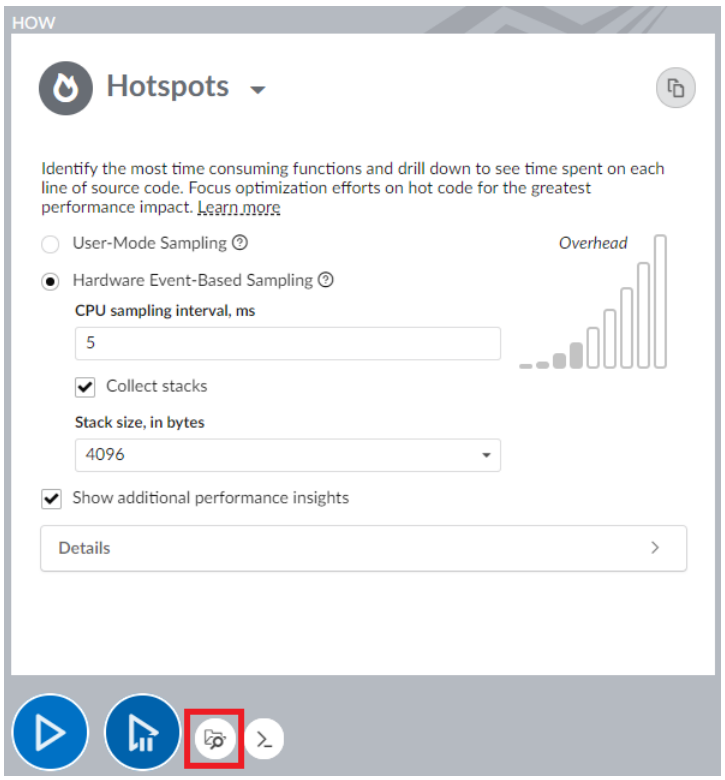
4. On the host machine, use a browser to open the URL reported by `vtune-backend`. Change the port of the container used by `vtune-backend` to the port you specified when creating the container.

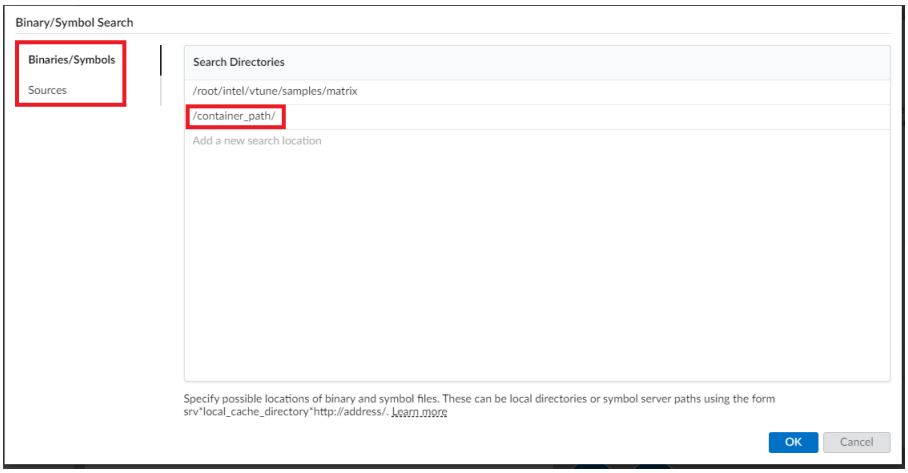
> **NOTE** The IP address in this output is the IP address of the container. You can access it only from the host where the container is running. To access `vtune-server` from outside of the host, use IP address or hostname of the external host.

5. Create a project, say `vtune_in_docker`.
6. In the container, run Hardware Event-based Hotspots in the Profile System mode.



7. Specify the locations of source and binaries to enable source-level and function-level analysis.





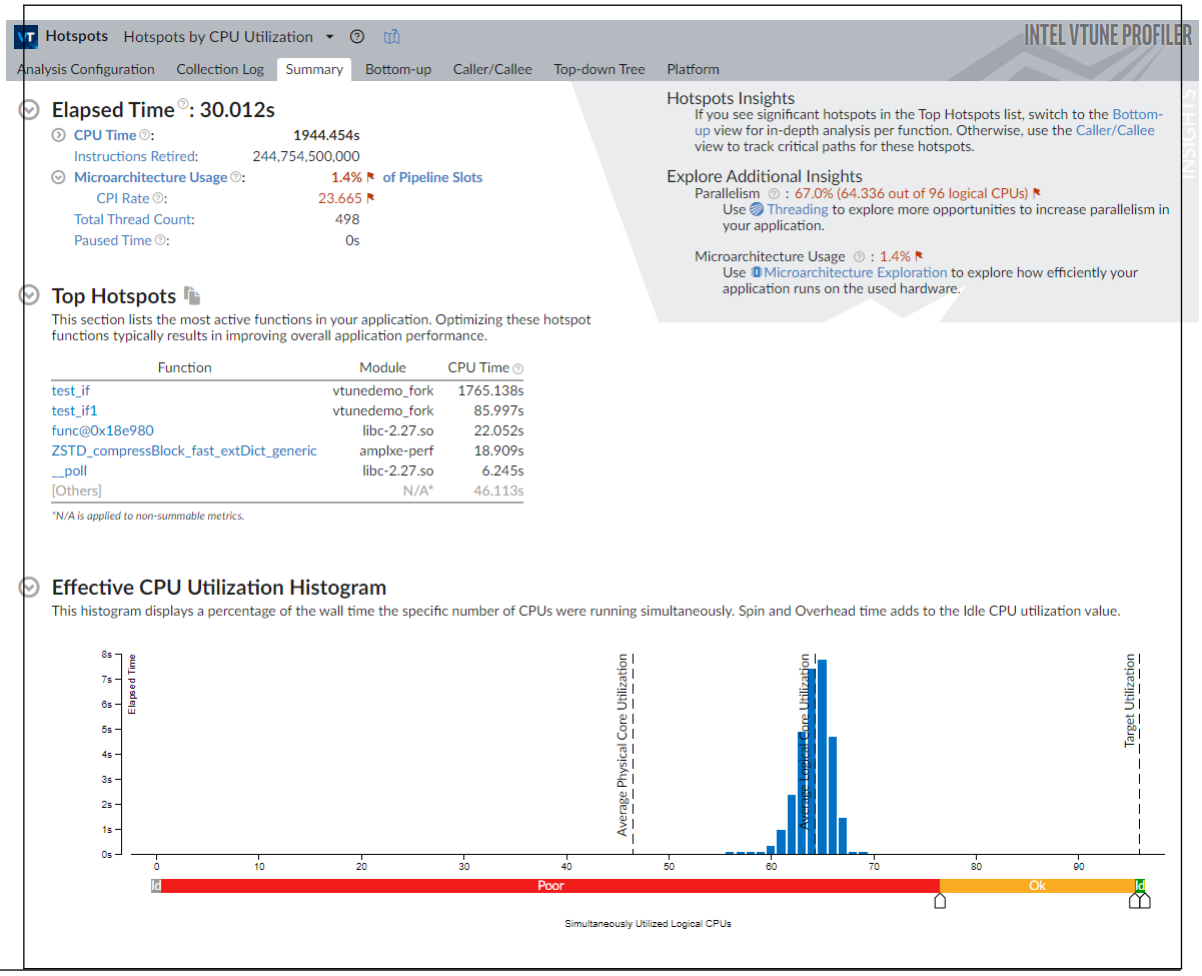
8. Start a command line collection from the container by specifying binary and search directories with the `-search-dir` and `-source-search-dir` options . To access the results from the GUI, point `-result-dir` to the current location of the VTune project.

```
my_oneapi_container> vtune -collect hotspots -knob sampling-mode=hw -knob stack-size=4096 --
duration 30  -result-dir=/root/intel/vtune/projects/vtune_in_docker/r@@@{at} -search-dir /
container_path -source-search-dir /container_path
```

NOTE Use the **Attach to Process** mode to profile your application running in a docker container.

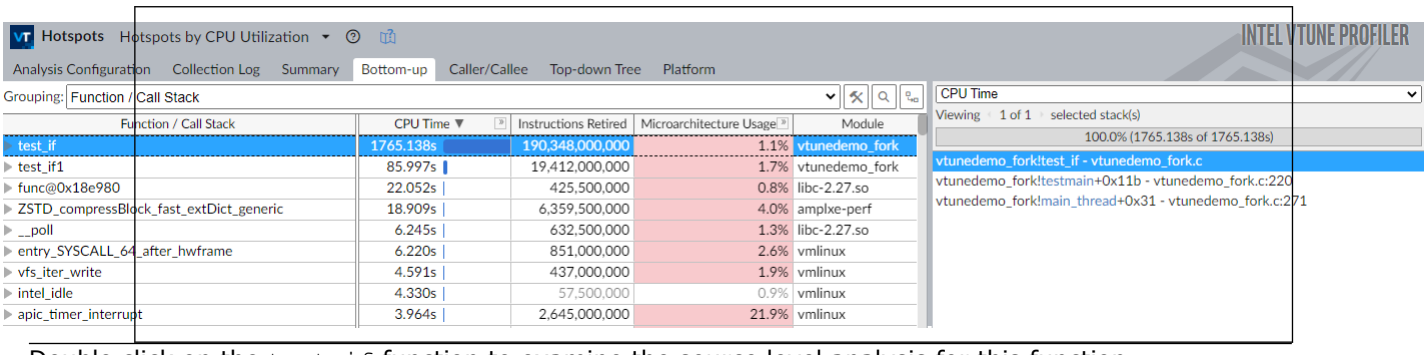
```
my_oneapi_container> docker exec my_oneapi_container vtune -collect hotspots -knob
sampling-mode=hw  -result-dir=/root/intel/vtune/projects/vtune_in_docker/r@@@{at}  -search-
dir /container_path -source-search-dir /container_path -target-process matrix
```

9. Once the analysis completes, see results in the **Summary** tab.

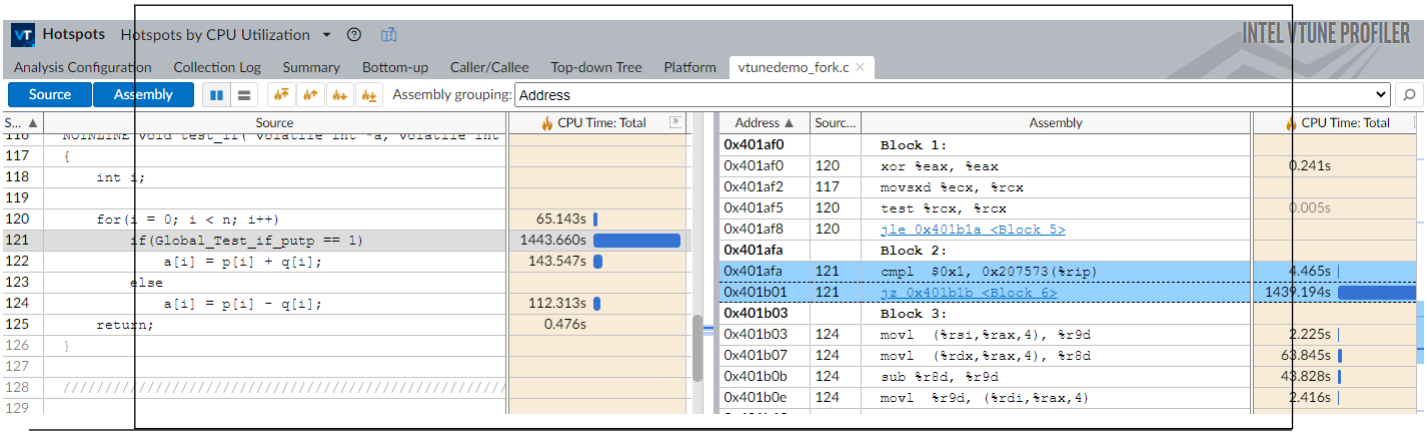


The **Top Hotspots** section of the **Summary** view shows that the `multiply` function of the target application consumed the most CPU time.

10. Click the `multiply` function in the list and switch to the **Bottom-up** tab.



11. Double click on the `test_if` function to examine the source level analysis for this function.



NOTE You can have source-level analyses for native applications that run simultaneously in multiple containers, if all of these containers have the same mounted host folder with the binaries.

Profiling Considerations:

- You can only profile C/C++ applications.
- You cannot profile applications instrumented with ITT/JIT API.

NOTE
Discuss this recipe in the Analyzers developer forum.

See Also

Java* Code Analysis

Profiling Container Targets

Profiling a Remote Target Through a Proxy Server (NEW)

This recipe describes how to run Intel® VTune™ Profiler through a proxy server to profile remote targets.

When you need to profile remote target systems, follow this recipe to run Intel® VTune™ Profiler through a proxy server. The recipe describes host configurations for Windows or Linux* systems.

- INGREDIENTS
- DIRECTIONS:
  - Windows Host Configuration
  - Linux Host Configuration

Ingredients

This section lists the software used for the performance analysis scenario.

- **Operating system**: Windows, Linux, or macOS systems
- **Tools:**Intel® VTune™ Profiler

Windows Host Configuration

Prerequisites:

If you do not already have RSA private/public keys for passwordless SSH access, follow this procedure to generate them using the internal Intel® VTune™ Profiler generator.

1. Create a python script called `generator.py`. In this script, replace `USER` with your username and `HOSTNAME` with the name of your target machine.

```
import sys
import pythonhelpers1.genhelpers as genhelpers
if len(sys.argv) < 2:
    print("Usage: amplxe-python generator.py USER@HOSTNAME")
    sys.exit(1)
private_key = "id_rsa_vtune_" + str(sys.argv[1])
genhelpers.ssh_keygen(private_key, private_key + '.pub')
```

2. In a command window, run this command:

```
VTUNE_INSTALL_DIR\bin64\amplxe-python generator.py USER@HOSTNAME
```

The private and public keys should get created in the current directory.

Run Intel® VTune™ Profiler on Windows Host through Proxy Server

1. Copy these keys to the `%USERPROFILE%\.ssh` directory on the host system:
   - `id_rsa_vtune_`**`USER@HOSTNAME`**
   - `id_rsa_vtune_`**`USER@HOSTNAME`**`.pub`
2. Add the content of the public key to `~/.ssh/authorized_keys` on the target system and `~ – USER` home directory.
3. Install the `Ncat` third-party utility on the Windows host for remote connection through a proxy server. You can download this from https://nmap.org/ncat.
4. Create `%USERPROFILE%\.ssh\config` with these lines:

```
Host HOSTNAME
    ProxyCommand 'PATH\TO\NCAT\ncat.exe' --proxy-type <TYPE> --proxy <PROXYADDR[:PORT]> %h %p
```

   Here,
   - `TYPE` refers to the type of proxy server.
   - `PROXYADDR` refers to the address of the proxy server.
   - `PORT` refers to the port number.
5. Fix permissions for the config and private key.

```
icacls %USERPROFILE%\.ssh\id_rsa_vtune_USER@HOSTNAME /inheritance:r
icacls %USERPROFILE%\.ssh\id_rsa_vtune_USER@HOSTNAME /grant:r "%USERNAME%":"(R)"
icacls %USERPROFILE%\.ssh\config /inheritance:r
icacls %USERPROFILE%\.ssh\config /grant:r "%USERNAME%":"(R)"
```

6. Check the connection by calling the `uname` command.

```
VTUNE_INSTALL_DIR\bin64\ssh.exe -i "%USERPROFILE%\.ssh\id_rsa_vtune_USER@HOSTNAME" USER@HOSTNAME
uname
```

You are now ready to run Intel® VTune™ Profiler on the Windows host to profile the remote target.

Linux/macOS Host Configuration

Prerequisites: If you do not already have RSA private/public keys for passwordless SSH access, generate them using an empty passphrase :

```
host> ssh-keygen -t rsa
```

Run Intel® VTune™ Profiler on Linux Host through Proxy Server

1. Add the content of the public key to `~/.ssh/authorized_keys` on the target system, `~ – USER` home directory. Replace USER with your username.
2. If you do not already have it on the host machine, download and install Ncat or Netcat third-party utilities on the host for remote connection through proxy. You can download them from:
   - https://nc110.sourceforge.io/
   - https://nmap.org/ncat
3. Create `~/.ssh/config` with these lines. The `nc` option depends on your version.

```
Host HOSTNAME
    ProxyCommand nc -X <TYPE> -x <PROXYADDR[:PORT]> %h %p
```

   or

```
Host HOSTNAME
    ProxyCommand nc --proxy-type <TYPE> --proxy <PROXYADDR[:PORT]> %h %p
```

   where:

   - `HOSTNAME` refers to the name of the target machine.
   - `TYPE` refers to the type of proxy server.
   - `PROXYADDR` refers to the address of the proxy server.
   - `PORT` refers to the port number.
4. Check the connection by calling the `uname` command.

```
ssh USER@HOSTNAME uname
```

You are now ready to run Intel® VTune™ Profiler on the Linux host to profile the remote target.

See Also

Set up Linux System for Remote Analysis

Configure SSH Access for Remote Collection

Profiling in an Apptainer* Container

Learn how to configure an Apptainer container to run analyses with Intel® VTune™ Profiler. Identify hot spots in an application that is running in the isolated container environment.

Content expert: Alexey Kireev

- INGREDIENTS
- DIRECTIONS:

  1. Install and configure an Apptainer* container
  2. Run performance analysis inside the container

Ingredients

This section lists the hardware and software tools used for the performance analysis scenario.

- **Application**: `MatrixMultiplication`

  This Java* application is used as a demo and not available for download.
- **Tools**: Intel VTune Profiler
- **Linux container runtime**: `Apptainer`
- **Operating system**: Ubuntu* 20.04
- **CPU**: Intel® microarchitecture code named Skylake with 8 logical CPUs

Install and Configure an Apptainer* Container

1. Install Apptainer. For installation instructions, see the Apptainer installation guide.
2. Create an Apptainer* container, for example, using the Docker Hub:

```
host> apptainer build ubuntu.img docker://ubuntu:latest
INFO:    Starting build...
Getting image source signatures
Copying blob 5e8117c0bd28 done
Copying config b6548eacb0 done
Writing manifest to image destination
Storing signatures
2023/12/08 15:23:58  info unpack layer:
sha256:5e8117c0bd28aecad06f7e76d4d3b64734d59c1a0a44541d18060cd8fba30c50
INFO:    Creating SIF file...
INFO:    Build complete: ubuntu.img
```

NOTE
Make sure the `ubuntu.img` file is created in the current directory.

3. Run the container.

   Apptainer allows you to map directories on your host system to directories within your container. This way, you can read and write data on the host system. For example, if you have a host folder `/tmp/vtune` with VTune Profiler and a Java application, you must run the container and map `/tmp/vtune` to `/local/vtune` within the container.

```
host> apptainer shell --bind
/opt/intel/oneapi/vtune/2024.0:/local/vtune --bind
/test_application:/local/test_application ./ubuntu.img
```

Run Analysis inside the Container

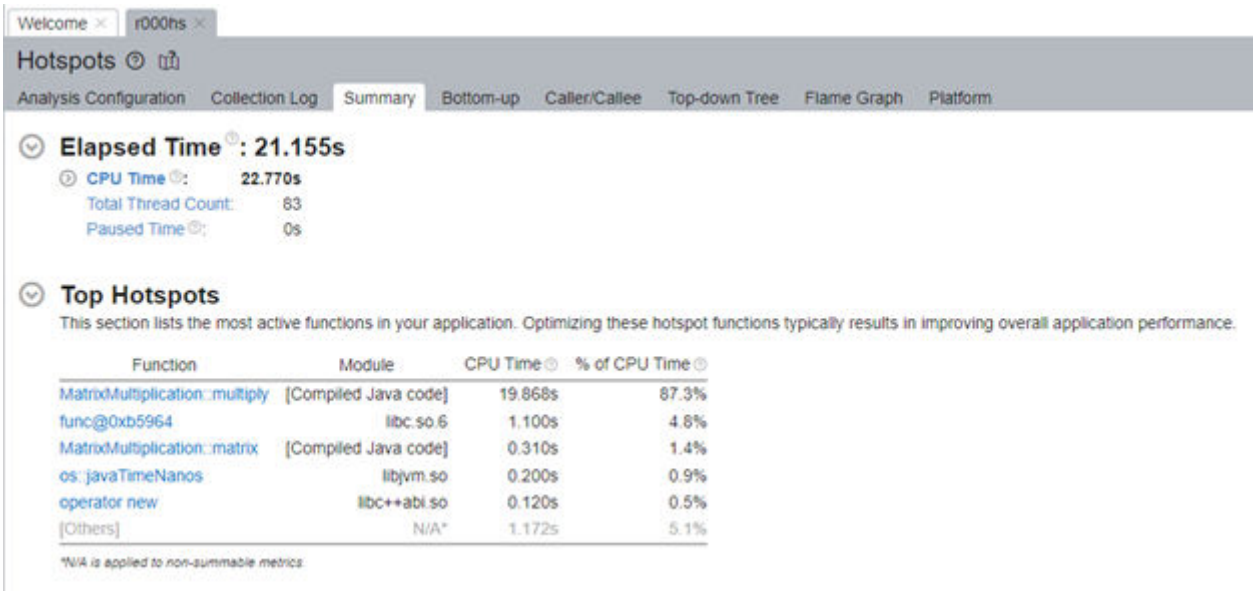
From the Apptainer container, open the command line interface of VTune Profiler(`vtune-cl`). Run an analysis for your Java application.

For example, to run the Hotspots analysis on the `MatrixMultiplication` application, type:

```
Apptainer> /local/vtune/bin64/vtune -collect hotspots -- /local/test_application/jdk-21.0.1/bin/
java -cp /local/test_application MatrixMultiplication 2000 2000 2000 2000
```

NOTE
To profile a target application running in the Apptainer container, you must open VTune Profiler from the same container.

When the analysis result is ready, use the VTune Profiler GUI installed on the host system to open the result. Start in the **Summary** window to see a performance overview for the application:



NOTE
If you need to re-finalize an analysis result outside the Apptainer container (for example, in the GUI version of VTune Profiler installed on the host system), ensure that all binary and source files required for module resolution are accessible outside the container.

See Also
Java* Code Analysis

Profiling Linux*, Android*, and QNX* System Boot Time

This recipe illustrates how you integrate performance analysis with Intel® VTune™ Profiler into the boot flow of Linux, Android, and QNX operating systems. Use this analysis to improve boot order inspection by identifying activities that execute very slowly on CPU cores during the OS boot.

Content Expert: Jeffrey Reinemann

When profiling boot time, you inject the performance data collection command of Intel® VTune™ Profiler into the early stage of the OS boot (either configured via an init script or using a particular service). For optimum results, follow these guidelines:

- Place the data collector binary files of Intel® VTune™ Profiler in the earliest available file directory.
- For Linux and Android OS data collector writes to the file system, the output file name must use the earliest available writable directory.
- For Linux and Android OS, the data collection command of Intel® VTune™ Profiler depends on file system availability. For QNX OS, the command depends on network availability.

NOTE While this approach is suitable to address several problems during OS boot time, it cannot cover the entire boot process. For example, the kernel decompression stage and file system mount stages are not covered.

- INGREDIENTS

- DIRECTIONS:

  1. Profile target system boot time:
     - Linux system via systemd
     - QNX system
     - Android system
  2. Import the Result to a VTune Profiler Project.
  3. Analyze process execution.

Ingredients

This section lists the software you need for the performance analysis scenario.

- **Operating system**:
  - Linux with the `systemd` system initialization type. Root access is enabled.
  - QNX
- **Tools:**
  - QNX* Momentics* Tool Suite
  - QNX 7.0 SDK
  - Intel® VTune™ Profiler version 2023 (or newer)

NOTE

- Starting with the 2020 release, Intel® VTune™ Amplifier has been renamed to Intel® VTune™ Profiler.
- Most recipes in the Intel® VTune™ Profiler Performance Analysis Cookbook are flexible. You can apply them to different versions of Intel® VTune™ Profiler. In some cases, minor adjustments may be required.
- Get the latest version of Intel® VTune™ Profiler:
  - From the Intel® VTune™ Profiler product page.
  - Download the latest standalone package from the Intel® oneAPI standalone components page.

Profile Linux System Boot Time via systemd

Prerequisites:

- Install Intel® VTune™ Profiler on your target Linux system.
- Check the type of system initialization. To confirm the system is using `systemd`, enter:

```
systemctl | grep "\-\.mount"
```

If `systemd` is used, you can expect the following output:

```
-.mount  loaded  active  mounted  /
```

To profile Linux system boot time:

1. Create a `/boot_profile` file and configure it to run a hardware analysis (Hotspots, I/O Analysis, and others) with Intel® VTune™ Profiler.

   For example, to execute the Hotspots analysis system-wide for 30 seconds using a low sampling interval for higher data precision, use:

```
#!/bin/bash
/opt/intel/oneapi/vtune/latest/bin64/vtune -c hotspots -knob sampling-mode=hw -knob sampling-
interval=0.1 -d 30 -finalization-mode=none -r /tmp/boot_profile &
```

NOTE

- Make sure the path in the file correctly specifies the Intel® VTune™ Profiler installation directory. By default, the installation directory on Linux is `/opt/intel/oneapi/vtune/<version>`.
- The path to the `boot_profile` script can be any local path available at early boot stages, for example: `/tmp`.

2. Change the permission for the data collection startup script:

```
chmod 755 /boot_profile
```

3. Create a `/etc/systemd/system/vtune_boot.service` file with the following content:

```
[Unit]
Description=VTune Profiler boot profile service

[Service]
Type=forking
ExecStart=/boot_profile

[Install]
WantedBy=multi-user.target
```

4. Enable the service:

```
systemctl enable vtune_boot
```

5. Reboot your system to start a Intel® VTune™ Profiler data collection during the OS boot process.

When the data collection is completed, you can find the result directory in `/tmp/boot_profile`. This directory is created under the `root` user. If you need to open a result under a regular user, change the folder permissions:

```
sudo chmod -R a+w /tmp/boot_profile
```

Additional commands:

- To disable the service:

```
systemctl disable vtune_boot
```

- To analyze Intel® VTune™ Profiler collector output during the OS boot process in case of any failures:

```
sudo journalctl -u vtune_boot
```

Profile Android System Boot Time

Prerequisites:

1. Install Intel® VTune™ Profiler on your host system.
2. Run the `lsmod` command on the target Android system to make sure Intel sampling drivers are available.

   If the drivers (`pax.ko`, `sep5.ko`, `socperf3.ko`) are not present, you can either continue with driverless approach and skip the subsequent steps or you can build and sign the drivers as follows:

   a. Run the following command on your host system:

      `<vtune-install-dir>/target/<android-version-arch>/sepdk/build-driver`

   b. When prompted, specify the path to the GCC* compiler and Android kernel source directory used to build the target system.

      For example, the kernel source directory is `<android-source-dir>/out/target/product/<name>/obj/kernel`, and the compiler directory is `<android-source-dir>/prebuilts/gcc/linux-x86/x86/x86_64-linux-android-<version>/bin/x86_64-linux-android-gcc`.

Successfully built drivers are located in the following directories:

- <*vtune-install-dir*>/target/<*android-version-arch*>/sepdk/pax/pax.ko
- <*vtune-install-dir*>/target/<*android-version-arch*>/sepdk/sep5.ko
- <*vtune-install-dir*>/target/<*android-version-arch*>/sepdk/src/socperf/src/socperf.ko

c. Sign the drivers as follows:

```
$KERNEL_DIR/scripts/sign-file $(CONFIG_MODULE_SIG_HASH)
$KERNEL_DIR/$(CONFIG_MODULE_SIG_KEY)
$KERNEL_DIR/certs/signing_key.x509 <driver_file_name.ko>
```

where <*driver_file_name.ko*> is the name of the driver you sign. You have to sign each driver separately.

Use the kernel `config` file from `KERNEL_DIR` to get values for `CONFIG_MODULE_SIG_HASH` and `CONFIG_MODULE_SIG_KEY` parameters.

To profile Android system boot time:

1. Install the Intel® VTune™ Profiler target collector.

a. Boot the target system in a normal manner.
b. Run the Intel® VTune™ Profiler GUI and create a new project.
c. Configure a new analysis. In the WHERE field, select **Android Device (ADB)** as the connection type. Select the target device in the ADB destination field. From this point onwards, Intel® VTune™ Profiler should automatically upload the target collector to the target system.

2. Copy the target collector to the earliest available file system location (for example, to `/vendor`):

```
adb shell cp -rf /data/data/com.intel.vtune/perfrun /vendor/vtune
```

3. If Intel sampling drivers are available, copy `pax.ko`, `sep5.ko`, and `socperf3.ko` drivers to `/vendor/vtune`.

4. Choose the earliest available writable location as the destination directory for the collected traces. For example, choose `/data/vtune` and create an executable script (`/vendor/vtune/vtune.sh`) with this content, using either Intel sampling drivers or driver-less mode:

Driver mode	Driver-less mode

`#!/bin/sh`
`rm -rf /data/vtune`
`mkdir /data/vtune 0777`
`/system/bin/insmod /vendor/vtune/pax.ko`
`/system/bin/insmod /vendor/vtune/socperf3.ko`
`/system/bin/insmod /vendor/vtune/sep5.ko`
`LD_LIBRARY_PATH=/vendor/vtune/perfrun/lib64`
`SEP_BASE_DIR=/vendor/vtune/perfrun/lib64 /vendor/vtune/perfrun/bin64/sep -start -d 10 -out /data/vtune/android_boot.tb7`
	`#!/bin/sh rm -rf /data/vtune`
	`mkdir /data/vtune 0777`
	`echo 0 > /proc/sys/kernel/perf_event_paranoid`
	`echo 0 > /proc/sys/kernel/kptr_restrict`
	`/vendor/vtune/perfrun/bin64/amplxe-perf record -a -o /data/vtune/android_boot.data -- sleep 10`

This should start the Hotspots collection for 10 seconds.

5. Add this section to `init.rc` on the target. Consider using post-fs or any other trigger depending on the actual boot flow:

```
on fs
          start vtune
service vtune /vendor/vtune/vtune.sh
          user root
```

```
        group root
        seclabel u:r:init:s0
        oneshot
        disabled
```

NOTE If you have a read-only file system, consider changing these files on the host and building your Android system from the source code.

6. Optionally, depending on the OS configuration, add these lines to the `/system/sepolicy/private/file_contexts` file:

```
/system/bin/toolbox      u:object_r:toolbox_exec:s0
+ /system/bin/insmod     u:object_r:toolbox_exec:s0
+ /system/bin/sep        u:object_r:toolbox_exec:s0
+ /system/bin/sh         u:object_r:toolbox_exec:s0
```

NOTE Make sure your Android device is booted in the permissive mode.

7. Reboot the target Android system and wait until data is collected.
8. Copy the `/data/vtune/android_boot.tb7` file to the host system for further analysis.

Profile QNX System Boot Time

Prerequisites:

- Install QNX* Momentics* Tool Suite on your host.
- Install QNX 7.0 SDK.
- Import a BSP to your QNX Momentics workspace via **File** > **Import** > **QNX** > **QNX Source Package and BSP**.
- Install VTune Profiler on your host system.

To profile QNX system boot time:

1. Copy the target profiling agent (`sep` binary) from `<vtune-install-dir>/target/qnx_x86_64` to `<qnx-sdk-path>\qnx700\target\qnx7\x86_64\usr\bin`.
2. Modify a `*.build` file of your QNX image.
   a. Find a string `/usr/bin/gzip=gzip` and add `/usr/bin/sep=sep` after it.
   b. Find a startup script section and add `sep -p1 &`.

```
[+script] startup-script = {
...
    # NOTE: Temporary enable for UART devices on OCP bridge
    # will be able to removed once ABL is fixed
    ocp_init -d 0:24:0 0x200=0xffff04b5 0x204=7
    ocp_init -d 0:24:1 0x200=0xffff04b5 0x204=7
    ocp_init -d 0:24:2 0x200=0xffff04b5 0x204=7 # console
    ocp_init -d 0:24:3 0x200=0xffff04b5 0x204=7
    # the sep run before this could move system to unstable
    # state and crash it
    sep -p1 -d 10 &
```

The `sep` target profiling agent options are:

- `-p<mode>` configures the collection mode:

  `0` sets regular default mode. The profiling agent waits for the connection from the host over TCP/IP.

`1` enables the agent to start a preconfigured collection without stacks. Collected samples are stored in the target memory. To transfer the data to the host, TCP/IP connection is required.

`2` enables the agent to start a preconfigured collection with sample call stacks. Collected samples are stored in the target memory. To transfer the data to the host, TCP/IP connection is required.

- `-d <sec>` sets the maximum duration of a collection (in seconds). The collection stops after the specified time or when the memory buffer is full.
- `-s <sec>` defers the start of a collection by the specified time.
- `-b <size_ratio>` sets the collection buffer size (1 ^ size_ratio bytes); for a single CPU core, for example, specify `-b 23` for 8Mb buffer size. The agent uses a double-buffer schema for switching buffers so that the amount of target memory consumption per a CPU core would be 16Mb. In case of 4 CPU cores, the overall memory consumption by the agent would be 64Mb. The default value is 19 (0.5Mb).

3. Rebuild and flash QNX image.
4. Reboot your system to start a data collection with Intel® VTune™ Profiler during OS boot process.
5. Propagate the collection results to the host.

   In the preconfigured collection mode (`-p1` or `-p2`), the target agent profiles the workload for the specified duration inside the memory buffer and switches to the listening mode sending a message like this: *'sep5_0: Waiting for control connection from host on port XXXX...' to console*. After this, you can launch the `sep` utility on the host to copy the collected data from the target over the network. Make sure the host command line options you use correspond to the target agent options. For example, for the `-p1` mode, the host command looks like this:

```
<vtune-install-dir>/bin64/sep -start -target-ip <target-system-ip-address> -target-port 9321 -
out /tmp/qnx_boot.tb7
```

   For the `-p2` mode:

```
<vtune-install-dir>/bin64/sep -start -target-ip <target-system-ip-address> -target-port 9321 -
lbr call_stack -out /tmp/qnx_boot.tb7
```

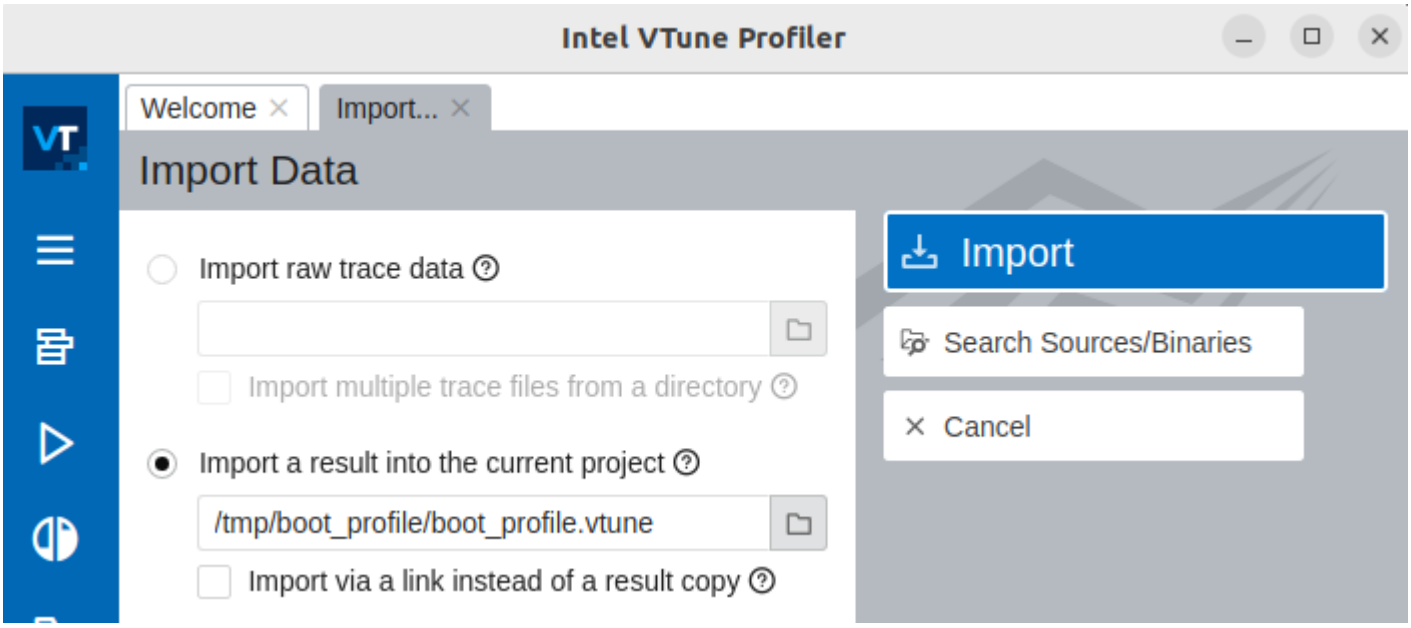
Import the Result to an Intel® VTune™ Profiler Project

1. Launch the Intel® VTune™ Profiler standalone GUI on the host system to pick up proper binary files during result finalization.

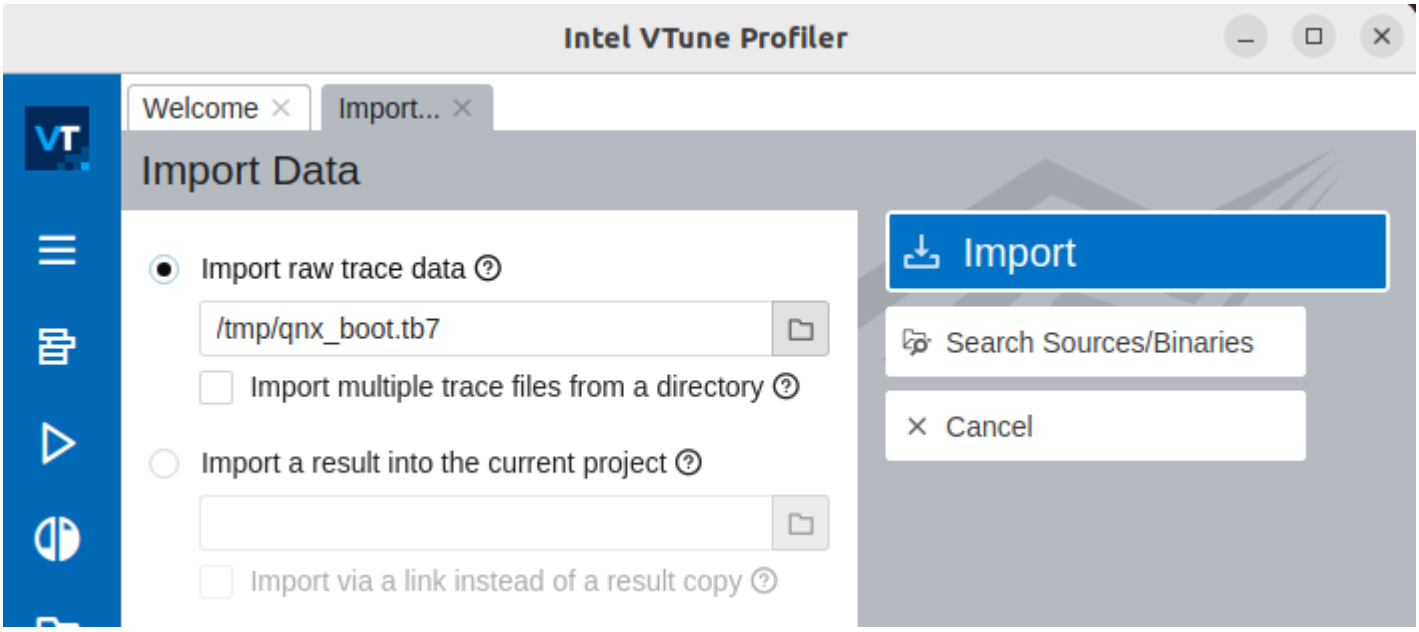
   For example, to launch Intel® VTune™ Profiler on Windows OS, enter:

   `<vtune-install-dir>\bin64\amplxe-gui.exe`
2. Create a new VTune Profiler project and configure binary/symbol search directories to include paths for the debug files of the kernel and/or drivers.

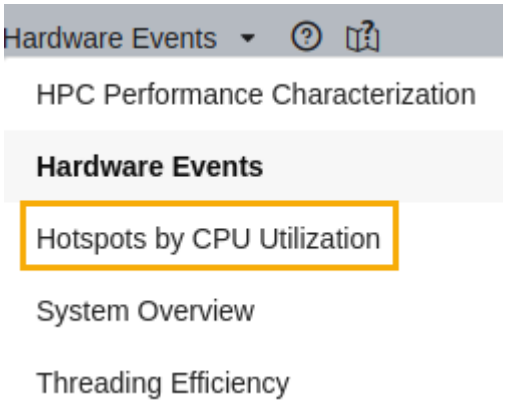
   On a Linux host, you can change the kptr_restrict value to 0 to enable resolving kernel function names.
3. Import your result to the project:
   - To import a Linux result, use the **Import a result into the current project** option.



- To import a result from Android or QNX, use the **Import raw trace data** option. Click the browse button to select the required `*.tb7` file:



When the `*.tb7` file is imported and the result is finalized, switch to the **Hotspots by CPU Utilization** viewpoint:



Analyze Process Execution

Open the result and switch to the **Bottom-up** tab to identify the processes which occupied the most CPU resources:

boot_profile

Hotspots

INTEL VTUNE PROFILER

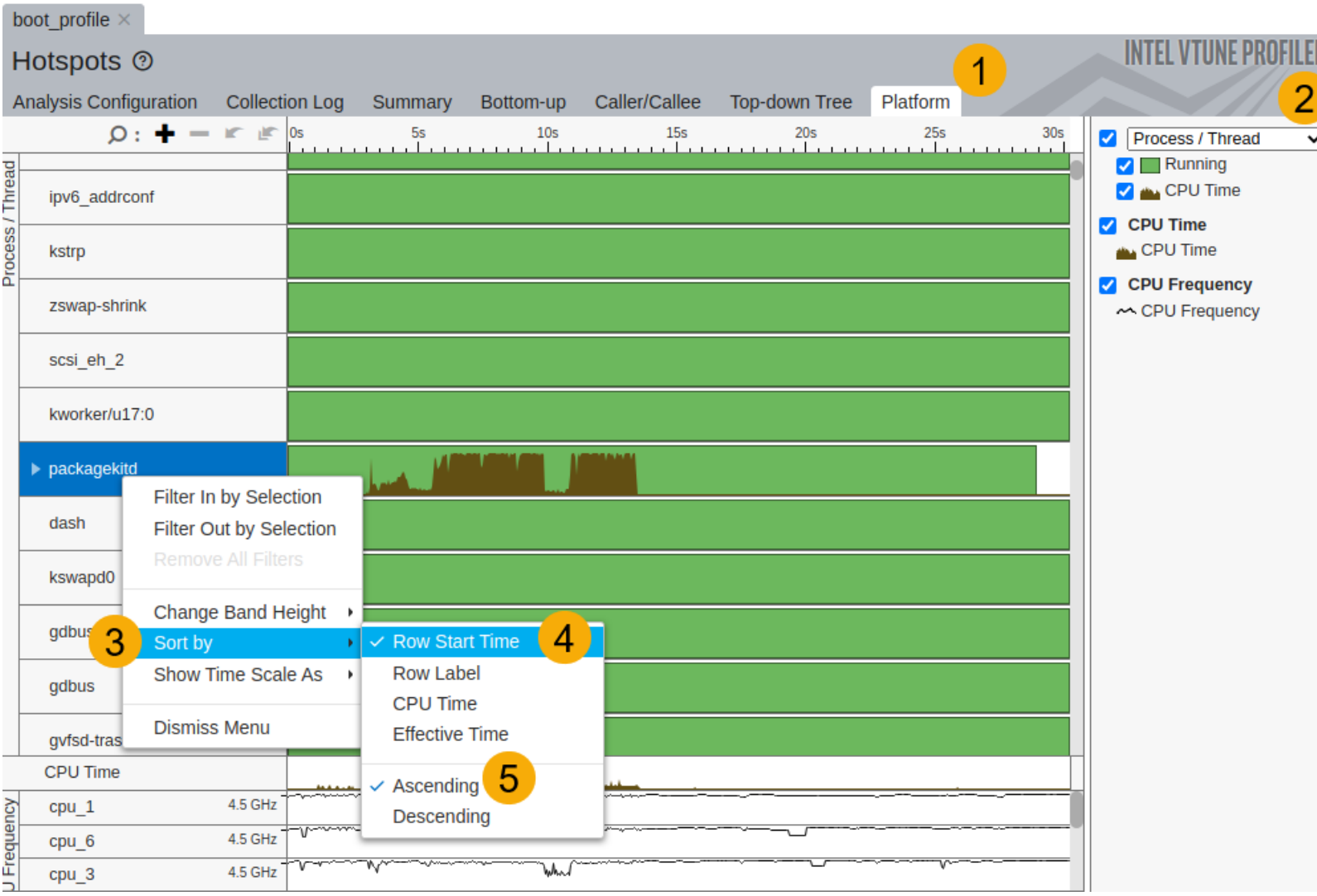
Analysis Configuration | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Platform

Grouping: Process / Thread / Module / Function / Call Stack

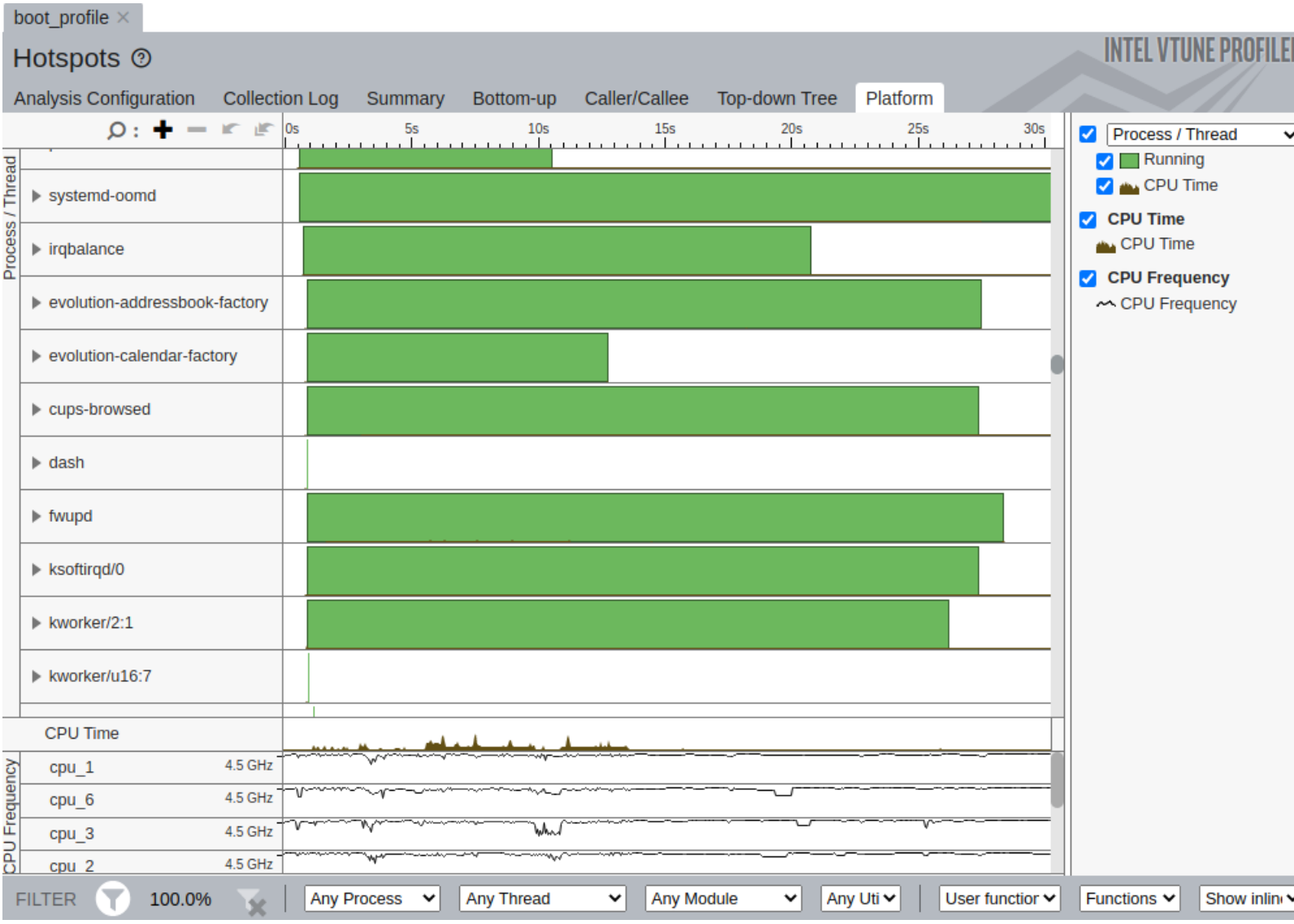
Process / Thread / Module / ...	CPU Time ▼	Instructions Retired	Microarchitecture Usage	Module
packagekitd	7.139s	41,013,000,000	33.1%	
snap-store	1.071s	6,747,300,000	46.3%	
vmlinux	0.801s	483,300,000	8.1%	
amplxe-runss	0.511s	2,200,500,000	31.8%	
snapd	0.311s	1,480,950,000	34.7%	
gnome-shell	0.213s	814,050,000	23.5%	
python3.10	0.149s	604,800,000	35.5%	
python3.10	0.136s	596,700,000	31.2%	
python3.10	0.113s	557,550,000	30.7%	
python3.10	0.105s	469,800,000	28.5%	
dbus-daemon	0.105s	360,450,000	27.2%	
python3.10	0.101s	471,150,000	29.7%	
python3.10	0.095s	469,800,000	29.2%	
python3.10	0.094s	469,800,000	28.2%	
plymouthd	0.078s	118,800,000	23.5%	
ld-2.23.so	0.061s	340,200,000	32.7%	
systemd	0.042s	155,250,000	33.5%	
gjs-console	0.035s	140,400,000	31.0%	
python3.10	0.032s	167,400,000	26.6%	
kworker/u16:10	0.026s	140,400,000	35.0%	

Next, let us analyze the sequence of execution of processes/services.

1. Switch to the **Platform** tab.
2. Change the Timeline grouping to **Process/Thread**.
3. Right-click to open the context menu.
4. Sort the rows by **Row Start Time**.
5. Select **Ascending** order for the display.



6. Analyze a process execution order:



See Also

QNX Targets

Import Results and Traces into the Intel® VTune™ Profiler GUI

Using Intel® VTune™ Profiler Server with Visual Studio Code and Intel® DevCloud for oneAPI (NEW)

This recipe demonstrates how you use Intel® VTune™ Profiler as a web server when you develop and tune performance on a remote development machine. For a remote machine, the recipe uses a compute node at Intel® DevCloud for oneAPI.

Content expert: Jennifer DiMatteo

- INGREDIENTS
- DIRECTIONS:

- Setup Overview
- Option 1: Use Intel® VTune™ Profiler Server for Remote Development with Visual Studio Code
- Option 2: Use Intel® VTune™ Profiler Server on a Remote System via SSH Terminal
- Finish Setup
- Usage Considerations

Ingredients

- Access to Intel® DevCloud for oneAPI
- Visual Studio (VS) Code
- Intel® VTune™ Profiler (available on Intel® DevCloud for oneAPI

NOTE

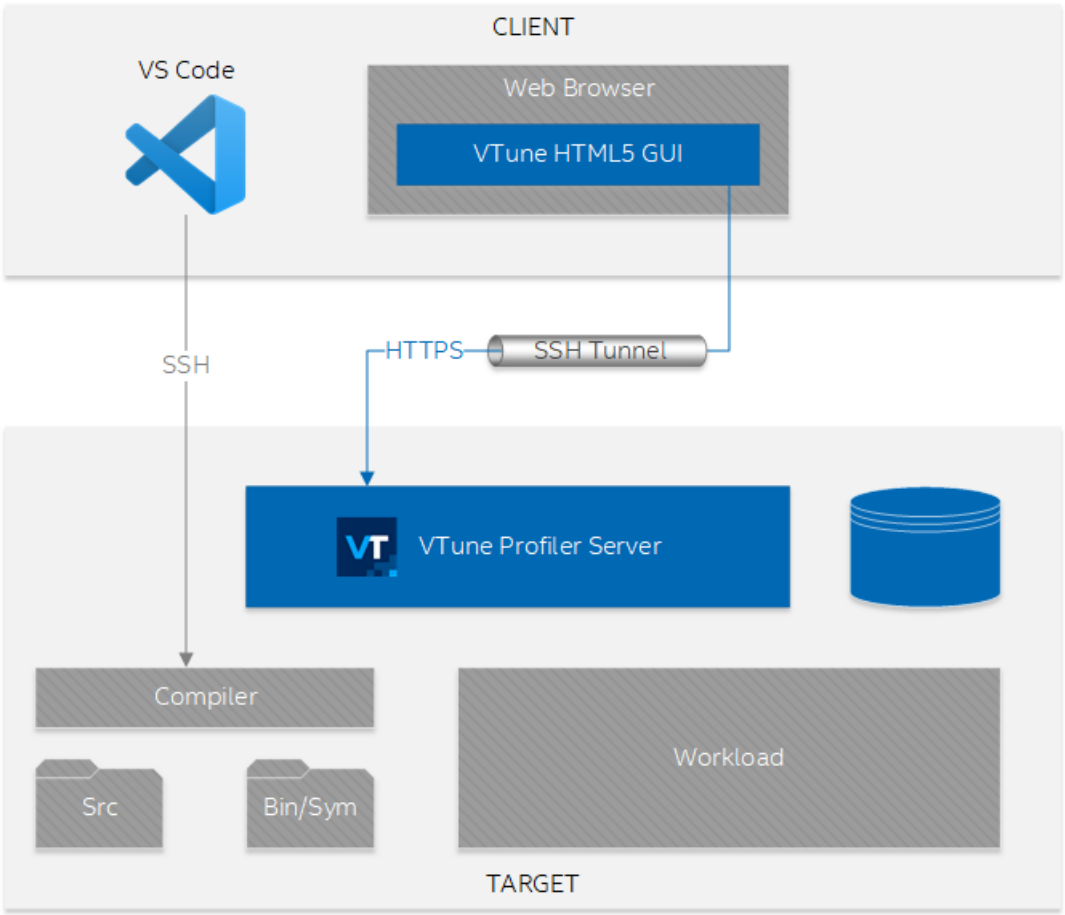
- Starting with the 2020 release, Intel® VTune™ Amplifier has been renamed to Intel® VTune™ Profiler.
- Most recipes in the Intel® VTune™ Profiler Performance Analysis Cookbook are flexible. You can apply them to different versions of Intel® VTune™ Profiler. In some cases, minor adjustments may be required.
- Get the latest version of Intel® VTune™ Profiler:
  - From the Intel® VTune™ Profiler product page.
  - Download the latest standalone package from the Intel® oneAPI standalone components page.

Setup Overview

With v2021.1.1 and newer versions, you can run Intel® VTune™ Profiler as a server and access it remotely using a web browser. This setup is useful when you develop applications on a remote system.

- You can run Intel® VTune™ Profiler on the system where you do development, so it has direct access to the binaries, debug info, and source files.
- Intel® VTune™ Profiler also stores the collected traces and processed data on the same system so you do not have to transfer this heavy data to your client system for analysis.
- You do not need to install anything on your client system. You only need a web browser to access the Intel® VTune™ Profiler GUI.

The following figure illustrates this setup:



Option 1: Use Intel® VTune™ Profiler Server for Remote Development with Visual Studio Code

1. Log into Intel® DevCloud for oneAPI.
2. Set up the VS Code connection. When you complete this procedure, you should get local VS Code connected to a compute node in the DevCloud.
3. Run Intel® VTune™ Profiler server on the compute node from the VS Code terminal:

```
vtune-backend --enable-server-profiling
```

4. Open the URL displayed by Intel® VTune™ Profiler server in the VS Code terminal. This starts the Intel® VTune™ Profiler GUI in your web browser.
5. Finish the setup.

Option 2. Use Intel® VTune™ Profiler Server on a Remote System via SSH Terminal

In this case, you must manually set up SSH tunneling. To simplify this procedure, run Intel® VTune™ Profiler on a specific port (`55001` in this example). You can select a different port if `55001` is busy.

1. Log into Intel® DevCloud for oneAPI.
2. Follow the instructions for Windows or Linux / MacOS systems and set up an SSH connection into the DevCloud.
3. Log into the DevCloud login node:

```
ssh devcloud
```

4. Reserve a DevCloud compute node:

```
qsub -I
```

NOTE Do not close the terminal after this step, as the action will release your compute node.

5. Open a new terminal.
6. Log into the DevCloud node again, this time with SSH Port forwarding enabled:

```
ssh -L 127.0.0.1:55001:127.0.0.1:55001 devcloud
```

7. Establish an SSH connection from the login node to the compute node with one more SSH tunnel:

```
ssh -L 127.0.0.1:55001:127.0.0.1:55001 s000-n000
```

   Replace `s000-n000` with your compute node name.

8. Start Intel® VTune™ Profiler server on the compute node:

```
vtune-backend --web-port=55001 --enable-server-profiling
```

9. Open the Intel® VTune™ Profiler GUI. Use your web browser to open the URL displayed by the Intel® VTune™ Profiler server.
10. Finish the setup.

Finish Setup

To complete the setup:

1. Accept the Intel® VTune™ Profiler server certificate.

   NOTE
   When you open the Intel® VTune™ Profiler GUI, your web browser may prompt you about the Intel® VTune™ Profiler server self-signed certificate. You can proceed safely without installing the certificate because the SSH tunnel provides protection from Man-in-the-Middle (MitM) attacks. For more information on transport security, see Set Up Transport Security.

2. Set the passphrase.

   When you run Intel® VTune™ Profiler server for the first time, the URL that it displays should contain a one-time-token. When you open this URL on a browser, Intel® VTune™ Profiler server prompts you to set a passphrase. Other users cannot access your Intel® VTune™ Profiler server without the passphrase. The hash of the passphrase is persisted on the server. Also, your browser stores a secure HTTP cookie so that you do not need to enter the passphrase each time you open the VTune GUI. Once you set the passphrase, the Intel® VTune™ Profiler welcome screen opens.
3. Create a project.
4. Configure an analysis. Your remote machine (running the Intel® VTune™ Profiler server) is selected as the target system by default since you ran the server with `--enable-server-profiling` option.
5. Set the target application path and any command-line arguments. For more information, see Set Up Analysis Target.
6. Run the analysis.

Usage Considerations

- The setup described in Option 1 relies on the functionality of the VS Code Remote - SSH extension to watch port numbers used by processes that are started through the VS Code terminal. The Remote - SSH extension automatically forwards these ports through the SSH tunnel. This action is controlled by the **remote.autoForwardPorts** setting, which is enabled by default.
- You can use the `--enable-remote-profiling` command-line option to enable the system that hosts VTune server as the performance profiling target. This option is disabled by default for security because running a VTune analysis involves launching a target application, which is an arbitrary command line. If multiple users have access to a single instance of VTune server, they would get access to execute arbitrary

code on behalf of the user account that runs the Intel® VTune™ Profiler server. Enable `--enable-remote-profiling` only when VTune server is intended for a single user and you do not share the passphrase used to access the server.
- Use the `--web-port=PORT` command-line option to run Intel® VTune™ Profiler server on a specific port. Otherwise, Intel® VTune™ Profiler may run on any arbitrary port available on the system. If the specific port is already in use, increase the number until a free port is available.
- Intel® VTune™ Profiler server displays this warning in the output:

```
warn: Server access is limited to localhost only. To enable remote access restart with
    --allow-remote-ui.
```

Because this procedure uses SSH port forwarding, you do not need to enable `--allow-remote-ui`. Incoming connections to the Intel® VTune™ Profiler server come from the SSH server and they are localhost connections. If you enable `--allow-remote-ui`, Intel® VTune™ Profiler server builds a URL with the real network card IP address or FQDN name, which may not be accessible from your client machine.
- By default, Intel® VTune™ Profiler server stores profiling results in your home directory. Use the `--data-directory` command-line argument to specify a different data directory. You can also use this argument to open pre-collected Intel® VTune™ Profiler results in Intel® VTune™ Profiler server. Intel® VTune™ Profiler can locate its results in any child folders.

See Also

Intel® VTune™ Profiler Web Server Interface

Intel® VTune™ Profiler Server Usage Models

Intel® DevCloud

Run a Profiling Analysis with Intel® VTune™ Profiler

Using Intel® VTune™ Profiler Server in HPC Clusters

This recipe demonstrates the usage of Intel® VTune™ Profiler server in High Performance Computing (HPC) clusters for interactive performance profiling or accessing performance data for scheduled jobs.

Typically, nodes in HPC clusters do not have any GUI context. This can cause inconveniences when using VTune Profiler for performance analysis. Users in HPC clusters either have to rely on command-line reports or have to the transfer the result files out of the cluster to view them on other machines using the desktop VTune Profiler GUI.

This recipe aims to eliminate these inconveniences by offering a better workflow. Starting with VTune Profiler version 2021.1.1, you can launch VTune Profiler as a server inside the HPC cluster and view the results remotely with the full-featured GUI, requiring nothing more than a machine that can run a modern web browser.

Intel® DevCloud for oneAPI is used as an example in this recipe, but this workflow is valid for any other HPC cluster or similar environment.

Content experts: Stas Neverov

- Ingredients
- Directions
  - Setup Overview
  - Interactive Performance Profiling with VTune Profiler Server
  - Serving Profiling Results for Scheduled Jobs
  - Usage Considerations

Ingredients

Here are the environment and software tools that you need:

- Access to Intel DevCloud for oneAPI
- Intel VTune Profiler—pre-installed on most HPC clusters, including Intel DevCloud for oneAPI.

Setup Overview

After you complete this setup, you will be able to:

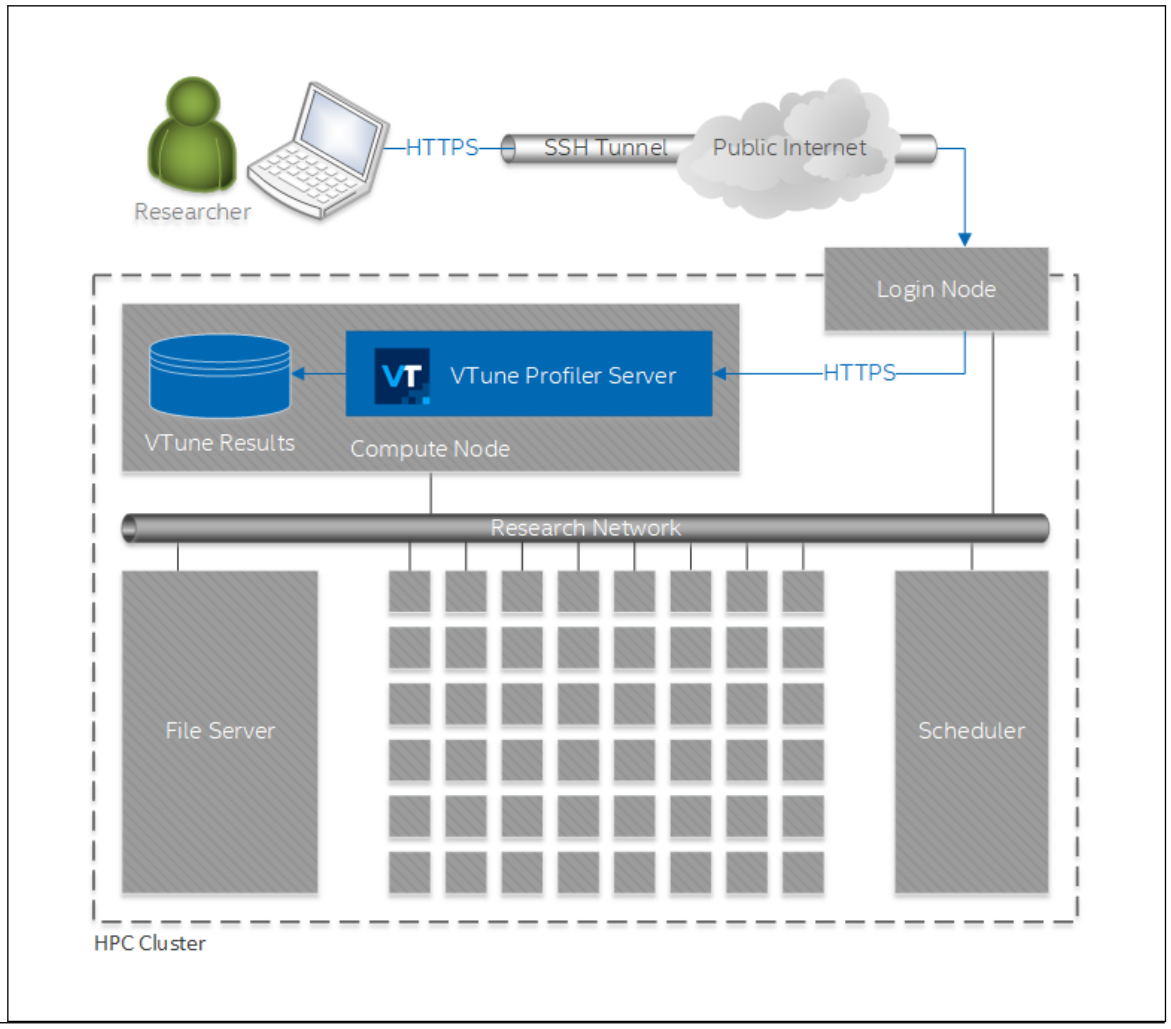
- Book a compute node in an HPC cluster for interactive use, launch VTune Profiler server, and use the VTune Profiler web GUI on your laptop to configure analysis and view the collected data.
- Schedule your job in an HPC cluster, wrap it with VTune Profiler command-line data collection, return when the job is complete and run the VTune Profiler server to view collected results in a web browser on your laptop.

For both cases, you do not need to install anything on your client system. You only need a web browser to access the VTune Profiler GUI.

Interactive Performance Profiling with VTune Profiler Server

In this scenario, you run VTune Profiler server on a compute node inside an HPC cluster and access the VTune Profiler GUI via a web browser on your laptop. This usage model is somewhat similar to using VTune Profiler desktop GUI via VNC, but it is easier to set up and provides a better user experience.

This figure illustrates the setup for this scenario:



Follow these steps to enable this workflow:

1. Log into Intel DevCloud for oneAPI.
2. Set up an SSH connection into the Intel DevCloud by following the instructions:
   - for Windows
   - for Linux and macOS
3. Log into the Intel DevCloud login node:

```
ssh devcloud
```

4. Reserve an Intel DevCloud compute node in interactive mode:

```
qsub -I
```

5. Run VTune Profiler server:

```
vtune-backend --enable-server-profiling --data-directory=~/intel/vtune/projects
```

   VTune Profiler server outputs a string like this:

```
Serving GUI at https://127.0.0.1:42277?one-time-token=456e20b6dcaad209ea2157744c1dc6c5
```

   Take note of the port number, compute node name, and the URL. You will need this information for the next steps.

> **NOTE** The port number—`42277` in the sample output—is a random port out of those available on the compute node. Port number `42277` is used here as an example. It will be different when you start VTune Profiler server.

6. Open a new terminal window. Do not close the first terminal, as this will stop the VTune Profiler server and will release the compute node.
7. Log into the DevCloud login node again, this time with SSH port forwarding enabled:

```
ssh -L 127.0.0.1:42277:127.0.0.1:42277 devcloud
```

> **NOTE** Replace port `42277` with the actual port printed out in step 5.

8. Establish an SSH connection from the login node to the compute node with one more SSH tunnel:

```
ssh -L 127.0.0.1:42277:127.0.0.1:42277 s000-n000
```

> **NOTE** Replace `s000-n000` with the compute node name on which VTune Profiler server was started on step 5; replace port number `42277` with the actual port number from step 5.

9. Open the VTune Profiler web GUI on your laptop. To do this, paste the URL printed out by VTune Profiler server in step 5 into a web browser on your laptop.
10. Accept the VTune Profiler server certificate.

    When you open the VTune Profiler GUI, your web browser will prompt you about the VTune Profiler server self-signed certificate. You can proceed safely without installing the certificate, because the SSH tunnel provides protection from Man-in-the-Middle (MitM) attacks. For more information on transport security, see the Set Up Transport Security topic.
11. Set the passphrase.

    When you run VTune Profiler server for the first time, the URL that it prints should contain a `one-time-token`. When you open this URL in a browser, VTune Profiler server prompts you to set a passphrase. Other users cannot access your VTune Profiler server without the passphrase. The hash of the passphrase is persisted on the server in your user home directory. Also, your browser stores a secure HTTP cookie, so that you do not need to enter the passphrase each time you open the VTune Profiler web GUI.

Once you set the passphrase, the VTune Profiler **Welcome** screen opens.

12. Create a new project and configure an analysis using VTune Profiler web GUI:
    a. Click the **New Project...** button on the **Welcome** screen.
    b. Enter the project name in the prompt dialog.

       VTune Profiler automatically opens the **Configure Analysis** dialog.

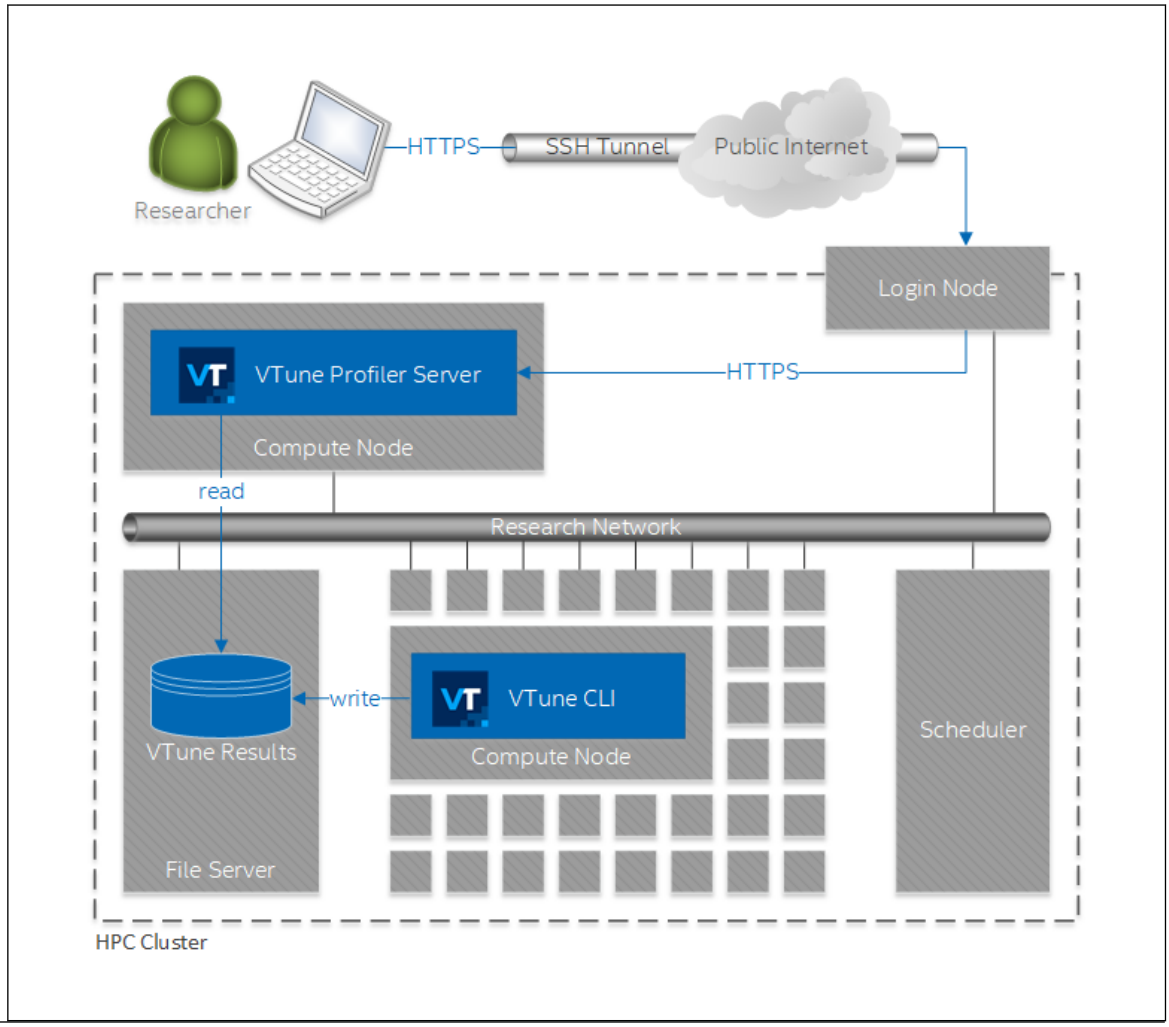
       Your compute node that is running the VTune Profiler server is selected as the target system by default.
    c. Set the target application path and any command-line arguments. See Set Up Analysis Target for details.
13. Click the **Start** button to run the analysis.

Once the analysis completes, VTune Profiler opens the **Summary** tab of the analysis result.

Serving Profiling Results for Scheduled Jobs

In this scenario, you collect performance data for scheduled jobs using the VTune Profiler command-line interface, and then serve these results by running VTune Profiler server inside the cluster. You can then access the results on your laptop via a web browser.

This figure illustrates the setup for this scenario:



Follow these steps to enable this workflow:

1. Log into Intel DevCloud for oneAPI.
2. Set up an SSH connection into the DevCloud by following the instructions:
   - for Windows
   - for Linux and macOS
3. Log into the DevCloud login node:

```
ssh devcloud
```

4. Create a script to wrap a job with VTune Profiler data collection:

```
echo vtune -collect hotspots \
        -r ~/intel/vtune/projects/demo/matrix/hotspots01 \
        -- /opt/intel/oneapi/vtune/latest/samples/en/C++/matrix/matrix \
    > ~/run_with_vtune.sh
```

   The `-r ~/intel/vtune/projects/demo/matrix/hotspots01` option defines where the collected VTune Profiler result will be stored. You can change this location, but, when you start VTune Profiler server at a later step, it should point to some parent folder of this location, e.g. `~/intel/vtune/projects`.

   NOTE

   - We assume that the user home directory is network mounted, and thus shared between all compute nodes.
   - Some HPC clusters could have an additional network mounted space that you could use to store VTune Profiler results. This might be a good idea since VTune Profiler results are typically large in size and your user home directory space is likely limited.
   - You can replace the sample `/opt/intel/oneapi/vtune/latest/samples/en/C++/matrix/matrix` with your own application.

5. Schedule a job using the script created in step 4:

```
qsub ./run_with_vtune.sh
```

6. Wait for the job to complete.

   The sample matrix application will take about a minute to complete. Your real-life HPC jobs could take hours to complete. You do not need to wait for the job to complete and can resume this flow the next day—your VTune Profiler results will be waiting for you in the specified location.
7. Reserve a DevCloud compute node in interactive mode:

```
qsub -I
```

8. Run VTune Profiler server:

```
vtune-backend --data-directory=~/intel/vtune/projects
```

   The `--data-directory=~/intel/vtune/projects` should refer to some parent folder of the result folder that you specified in step 4.

   VTune Profiler server outputs a string like this:

```
Serving GUI at https://127.0.0.1:42277?one-time-token=456e20b6dcaad209ea2157744c1dc6c5
```

   Take note of the port number, compute node name, and the URL. You will need this information for the next steps.

   NOTE The port number—`42277` in the sample output—is a random port out of those available on the compute node. Port number `42277` is used here as an example. It will be different when you start VTune Profiler server.

9. Open a new terminal window. Do not close the first terminal, as this will stop the VTune Profiler server and will release the compute node.
10. Log into the DevCloud login node again, this time with SSH port forwarding enabled:

```
ssh -L 127.0.0.1:42277:127.0.0.1:42277 devcloud
```

> **NOTE** Replace port number `42277` with the actual port number from step 8.

11. Establish an SSH connection from the login node to the compute node with one more SSH tunnel:

```
ssh -L 127.0.0.1:42277:127.0.0.1:42277 s000-n000
```

> **NOTE** Replace `s000-n000` with the compute node name on which VTune Profiler server was started on step 8; replace port number `42277` with the actual port number from step 8.

12. Open the VTune Profiler GUI on your laptop. To do this, paste the URL printed out by VTune Profiler server in step 8 into a web browser on your laptop.
13. Accept the VTune Profiler server certificate.

    When you open the VTune Profiler GUI, your web browser will prompt you about the VTune Profiler server self-signed certificate. You can proceed safely without installing the certificate, because the SSH tunnel provides protection from Man-in-the-Middle (MitM) attacks. For more information on transport security, see the Set Up Transport Security topic.
14. Set the passphrase.

    When you run VTune Profiler server for the first time, the URL that it prints should contain a `one-time-token`. When you open this URL in a browser, VTune Profiler server prompts you to set a passphrase. Other users cannot access your VTune Profiler server without the passphrase. The hash of the passphrase is persisted on the server in your user home directory. Also, your browser stores a secure HTTP cookie, so that you do not need to enter the passphrase each time you open the VTune Profiler GUI.

    Once you set the passphrase, the VTune Profiler **Welcome** screen opens.
15. Open the analysis result for the scheduled job.

    You should see a `demo/matrix/hotspots01` result in the **Project Navigator** panel. Double-click this result to open it.

Usage Considerations

- You can use the `--enable-server-profiling` command-line option to enable the system that hosts the VTune Profiler server as the performance profiling target. This option is disabled by default for security reasons, since running an analysis with VTune Profiler involves launching a target application with an arbitrary command line. If multiple users have access to a single instance of VTune Profiler server, they would be able to execute arbitrary code on behalf of the user account that runs the VTune Profiler server. Enable the `--enable-server-profiling` option only when VTune Profiler server is intended for a single user and you do not share the passphrase to access the server.
- Use the `--web-port=PORT` command-line option to run VTune Profiler server on a specific port. Otherwise, VTune Profiler server will run on an arbitrary port available on the system.
- VTune Profiler server displays this warning in the output:

  ```
  warn: Server access is limited to localhost only. To enable remote access restart with --allow-remote-access.
  ```

  Because the usage models described above use SSH port forwarding, you do not need to enable `--allow-remote-access`. Incoming connections to the VTune Profiler server come from the SSH server, and thus are essentially localhost connections.

If you enable the `--allow-remote-access` option, VTune Profiler server builds a URL with the real network card IP address or FQDN name, which may not be accessible from your client machine.

- By default, VTune Profiler server stores profiling results in your home directory. Use the `--data-directory` command-line argument to specify a different data directory. You can also use this argument to open pre-collected VTune Profiler results in VTune Profiler server. VTune Profiler can locate its results in any child folders.

See Also

Web Server Interface
VTune Profiler Server Usage Models
Intel® DevCloud for oneAPI
Run Analysis with VTune Profiler
VTune Profiler Command Line Interface

Using the Command-Line Interface to Analyze the Performance of a SYCL* Application running on a GPU (NEW)

This recipe illustrates how you use the command-line interface (CLI) in Intel® VTune™ Profiler to analyze the performance of a SYCL application offloaded on an Intel GPU. The recipe also describes how you can customize your report with collected data.

Intel® VTune™ Profiler provides a command line interface (the `vtune` tool) for remote analysis, scripted commands, and performance regression checks to monitor software performance over time. The `vtune` command line interface (CLI) provides an extensive set of options with which you can perform almost every task that is possible through the GUI. You can initiate analysis via the command line (running it as a background task or on a remote system) and then view the result or generate a report.

This recipe explores how you can use the CLI efficiently to generate reports on hotspots for these purposes:

- Explore hotspots on the CPU/GPU side by running `gpu-offload` and `gpu-hotspots` analyses.
- View the hottest GPU computing tasks annotated with:
  - Execution time
  - Data transfers
  - Working group sizes
  - SIMD width
  - Average GPU hardware metrics
- Generate Source/Assembly code views to analyze instructions that possibly contributed to performance issues.

Here are the ingredients and instructions you need to explore efficient CLI use for GPU performance analysis.

- INGREDIENTS
- DIRECTIONS:

  1. Build and Compile a SYCL Application
  2. Ensure Prerequisites for GPU Analyses
  3. Run GPU Offload Analysis
  4. Run GPU Compute/Media Hotspots Analysis

Ingredients

Here are the minimum hardware and software requirements for this performance analysis.

- **Application**: `matrix_multiply_vtune`. This sample application is available as part of the code sample package for Intel® oneAPI toolkits.

- **Compiler**: To compile a SYCL application, you need the Intel® oneAPI DPC++/C++ Compiler (`icpx -fsycl`) that is available with the Intel® oneAPI Base Toolkit.
- **Tools**: Intel® VTune™ Profiler 2021 - GPU Offload and GPU Compute/Media Hotspots Analyses.

  NOTE

  - Starting with the 2020 release, Intel® VTune™ Amplifier has been renamed to Intel® VTune™ Profiler.
  - Most recipes in the Intel® VTune™ Profiler Performance Analysis Cookbook are flexible. You can apply them to different versions of Intel® VTune™ Profiler. In some cases, minor adjustments may be required.
  - Get the latest version of Intel® VTune™ Profiler:
    - From the Intel® VTune™ Profiler product page.
    - Download the latest standalone package from the Intel® oneAPI standalone components page.

- **Microarchitecture**:
  - Intel® Iris® Pro Graphics 580
  - Intel microarchitecture codenamed Skylake S
- **Operating system**:
  - Ubuntu 20.04 LTS

Build and Compile the SYCL Application

1. Go to the sample directory.

```
cd <sample_dir>/VtuneProfiler/matrix_multiply_vtune
```

2. The `multiply.cpp` file in the `src` directory contains several versions of matrix multiplication. Select a version by editing the corresponding `#define MULTIPLY` line in `multiply.hpp`.
3. Compile your sample application:

```
cmake . && make
```

   This command generates a `matrix.icpx -fsycl` executable.

   To delete the program, type:

```
make clean
```

   This command removes the executable and object files that were created by the `make` command.

Ensure Prerequisites for GPU Analyses

Complete these steps before you run the **GPU Offload Analysis** or the **GPU Compute/Media Hotspots Analysis**.

1. Prepare the system to run a GPU analysis. See Set Up System for GPU Analysis.
2. Set up environment variables for Intel software tools:

```
source $ONEAPI_ROOT/setvars.sh
```

Run GPU Offload Analysis on the SYCL Application

Use the **GPU Offload Analysis** as a starting point to identify if an application is CPU or GPU bound. Explore GPU offload efficiency through data transfer analysis and find performance-critical kernels for further analysis and optimization.

Run GPU Offload Analysis

In the CLI, type:

```
vtune -collect gpu-offload -r ./result_gpu-offload -- ./matrix.icpx -fsycl
```

By default, VTune Profiler generates a summary report after collecting data. This report includes information on the following fields:

- Elapsed time
- GPU utilization information
- Information about the hottest computing tasks
- Recommendations

To see the summary report, type:

```
vtune -report summary -r ./result_gpu-offload
```

If you do not need to see the summary report immediately after data collection, change this setting with the `-no-summary` option:

```
vtune -collect gpu-offload -no-summary -r ./result_gpu-offload -- ./matrix.icpx -fsycl
```

```
root@nntpat99-39:/home/vtune/matrix_multiply_vtune# vtune -report summary -r ./result_gpu-offload
vtune: Using result path `/home/vtune/matrix_multiply_vtune/result_gpu-offload'
vtune: Executing actions 75 % Generating a report                              Elapsed Time: 11.235s
    GPU Utilization: 10.1%
     | GPU utilization is low. Consider offloading more work to the GPU to
     | increase overall application performance.
     |

        GPU Utilization
        GPU Engine        Packet Type  GPU Time     GPU Utilization(%)
        ----------------  -----------  --------  --------------------
        Render and GPGPU  Unknown        1.134s                  10.1%

Hottest GPU Computing Tasks
Computing Task             Total Time  Execution  % of Total Time(%)  Instance Count
-------------------------  ----------  ---------  ------------------  --------------
Matrix1_1<float>               1.146s     1.130s               98.6%               1
zeCommandListAppendBarrier     0.000s         0s                0.0%               0
Collection and Platform Info
    Application Command Line: ./matrix.dpcpp
    Operating System: 5.4.30 DISTRIB_ID=Ubuntu DISTRIB_RELEASE=20.04 DISTRIB_CODENAME=focal DISTRIB_DESCRIPTION="Ubuntu 20
    Computer Name: nntpat99-39
    Result Size: 98,0 MB
    Collection start time: 12:44:24 20/02/2021 UTC
    Collection stop time: 12:44:36 20/02/2021 UTC
    Collector Type: Event-based sampling driver,Driverless Perf system-wide sampling,User-mode sampling and tracing
    CPU
        Name: Intel(R) Processor code named Skylake
        Frequency: 2.592 GHz
        Logical CPU Count: 8
        Max DRAM Single-Package Bandwidth: 19.000 GB/s
    GPU
        Name: Iris Pro Graphics 580
        Vendor: Intel Corporation
        EU Count: 72
        Max EU Thread Count: 7
        Max Core Frequency: 950.000 MHz
        GPU OpenCL Info
            Version
            Max Compute Units: 72
            Max Work Group Size: 256
            Local Memory: 65,5 KB
            SVM Capabilities

Recommendations:
    GPU Utilization: 10.1%
     | GPU utilization is low. Switch to the for in-depth analysis of host
     | activity. Poor GPU utilization can prevent the application from
     | offloading effectively.
    EU Array Stalled/Idle: 69.8% of Elapsed time with GPU busy
     | GPU metrics detect some kernel issues. Use GPU Compute/Media Hotspots
     | (preview) to understand how well your application runs on the specified
     | hardware.
```

NOTE Families of Intel® Xe graphics products starting with Intel® Arc™ Alchemist (formerly DG2) and newer generations feature GPU architecture terminology that shifts from legacy terms. For more information on the terminology changes and to understand their mapping with legacy content, see GPU Architecture Terminology for Intel® Xe Graphics.

Generate Additional Reports to View Collected Data

- **CPU Hotspots Report**

This report displays a list of executed functions with CPU Time metrics, module names, source file paths and other parameters. The report also lists the hottest program units, starting with the most performance-critical unit. Use the `-column`, `-filter`, and `-limit` options to sort data into a tabular view:

```
vtune -report hotspots -r ./result_gpu-offload
```

```
                                CPU Time      CPU Time:Execution      CPU:Time:Queue      CPU Time:Idle
--------------------------      --------      ------------------      --------------      -------------
foEntryMinimal::extractFast     1.153s        0s                      0s                  1.153s
r::getCStr                      0.582s        0s                      0s                  0.582s
                                0.493s        0.417s                  0s                  0.075s
r::getULEB128                   0.317s        0s                      0s                  0.317s
                                0.289s        0s                      0s                  0.289s
_fast_string                    0.277s        0s                      0s                  0.277s
```

- **CPU Hotspots Report Filtered by Module and Grouped by Function**

  Use the `-filter` option to focus on a specific part of report like a particular module. You can then use `-group-by` option to group results in a specific sequence.

```
vtune -report hotspots -r ./result_gpu-offload -group-by=function -filter module=matrix.icpx -
fsycl -q
```

```
/home/vtune/matrix_multiply_vtune# vtune -report hotspots -r ./result_gpu-offload -group-by=function -filter modu
       CPU Time  CPU Time:Execution  CPU Time:Queue  CPU Time:Idle  Module        Function (Full)          Source
-----  --------  ------------------  --------------  -------------  ------------  -----------------------  -------
emcpy    0.003s                  0s              0s         0.003s  matrix.dpcpp  __intel_avx_rep_memcpy   [Unknown
         0.002s                  0s              0s         0.002s  matrix.dpcpp  main                     matrix.
```

  You can group the generated data in several ways like function name, module, source file path, or computing task.

  To see available groupings for a specific result, type:

```
vtune -report hotspots -r ./result_gpu-offload -group-by=?
```

- **CPU Hotspots Report Sorted by Order**

  Use the `sort-desc` and `sort-asc` options to sort specific information about hotspots in descending or ascending order. You can specify an order for up to three columns.

```
vtune -report hotspots -r result_gpu-offload -group-by module -sort-desc="CPU Time:Execution" -q
```

```
:/home/vtune/matrix_multiply_vtune# vtune -report hotspots -r result_gpu-offload -group-by module -sort-desc="CPU
         CPU Time  CPU Time:Execution  CPU Time:Queue  CPU Time:Idle  Module Path
--------  --------  ------------------  --------------  -------------  ----------------------------------------
           3.126s              0.936s              0s         2.190s  vmlinux
so.1.0.0   0.111s              0.110s              0s         0.001s  /usr/lib/x86_64-linux-gnu/libze_intel_gpu.s
           0.069s              0.024s              0s         0.045s  /usr/lib/x86_64-linux-gnu/libc-2.31.so
           3.766s                  0s              0s         3.766s  /tmp/1/Intel_oneAPI_VTune_Profiler_2021.2.0
           2.204s                  0s              0s         2.204s  /tmp/1/Intel_oneAPI_VTune_Profiler_2021.2.0
           0.577s                  0s              0s         0.577s  /tmp/1/Intel_oneAPI_VTune_Profiler_2021.2.0
           0.424s                  0s              0s         0.424s  /tmp/1/Intel_oneAPI_VTune_Profiler_2021.2.0
31         0.262s                  0s              0s         0.262s  /usr/lib/x86_64-linux-gnu/libigc.so.2.0.708
           0.016s                  0s              0s         0.016s  /usr/lib/x86_64-linux-gnu/ld-2.31.so
ic.so      0.012s                  0s              0s         0.012s  /tmp/1/Intel_oneAPI_VTune_Profiler_2021.2.0
```

  Here is another example:

```
vtune -report hotspots -r result_gpu-offload -group-by module -sort-asc="CPU Time:Idle" -q
```

```
-39:/home/vtune/matrix_multiply_vtune# vtune -report hotspots -r result_gpu-offload -group-by module -sort-asc="CP
            CPU Time  CPU Time:Execution  CPU Time:Queue  CPU Time:Idle  Module Path
----------  --------  ------------------  --------------  -------------  -----------------------------------
pu.so.1.0.0    0.111s              0.110s              0s         0.001s  /usr/lib/x86_64-linux-gnu/libze_intel_gp
module]        0.001s                  0s              0s         0.001s  [Unknown]
6.0.28         0.001s                  0s              0s         0.001s  /usr/lib/x86_64-linux-gnu/libstdc++.so.6
               0.002s                  0s              0s         0.002s  /lib/modules/5.4.30/kernel/drivers/gpu/d
               0.003s                  0s              0s         0.003s  /tmp/1/Intel_oneAPI_VTune_Profiler_2021.
o              0.005s                  0s              0s         0.005s  /opt/intel/oneapi/compiler/2021.1.1/linu
               0.005s                  0s              0s         0.005s  /tmp/1/Intel_oneAPI_VTune_Profiler_2021.
               0.005s                  0s              0s         0.005s  /home/vtune/matrix_multiply_vtune/matrix
               0.007s                  0s              0s         0.007s  /usr/lib/x86_64-linux-gnu/libgcc_s.so.1
namic.so       0.012s                  0s              0s         0.012s  /tmp/1/Intel_oneAPI_VTune_Profiler_2021.
```

To see available columns for a specific result, type:

```
vtune -report hotspots -r ./result_gpu-offload -column=?
```

The report data can contain such columns as **CPU Time:Self**, **Module**, and **Source File**.

- **Report of Top 'n' Time-Intensive Program Modules**

  Use the limit option to see information about the top 'n' hotspots. For example, to understand details about the top five time-intensive program modules in your application, type:

```
vtune -report hotspots -r result_gpu-offload -group-by module -sort-desc="CPU Time" -limit=5 -q
```

```
ome/vtune/matrix_multiply_vtune# vtune -report hotspots -r result_gpu-offload -group-by module -sort-desc="CPU Tim
U Time  CPU Time:Execution  CPU Time:Queue  CPU Time:Idle  Module Path
------  ------------------  --------------  -------------  ----------------------------------------------------
3.766s                  0s              0s         3.766s  /tmp/1/Intel_oneAPI_VTune_Profiler_2021.2.0/lib64/pinru
3.126s              0.936s              0s         2.190s  vmlinux
2.204s                  0s              0s         2.204s  /tmp/1/Intel_oneAPI_VTune_Profiler_2021.2.0/lib64/pinru
0.577s                  0s              0s         0.577s  /tmp/1/Intel_oneAPI_VTune_Profiler_2021.2.0/bin64/pinbi
0.424s                  0s              0s         0.424s  /tmp/1/Intel_oneAPI_VTune_Profiler_2021.2.0/lib64/libtp
```

- **Hotspots Report Grouped by Computing Task (offloaded on GPU) with Transfer Columns**

  This command displays hotspots information grouped by GPU computing task and also lists details about transfer sizes and transfer times between CPU and GPU:

```
vtune -report hotspots -r ./result_gpu-offload -group-by=computing-task -column=Transfer -q
```

The report contains data transfers that are attributed to the respective computing task.

```
atrix_multiply_vtune# vtune -report hotspots -r ./result_gpu-offload/ -group-by=computing-task-offload -column=Tr
tal Time:Host-to-Device Transfer  Total Time:Device-to-Host Transfer  Transfer Size  Transfer Size:Host-to-Device
--------------------------------  ----------------------------------  -------------  ----------------------------
                          0.010s                              0.006s       16,8 MB                       0,0 B
                              0s                                  0s         0,0 B                       0,0 B
```

- **Hotspots Report Grouped by GPU Offload Computing Task and Time Columns**

  This command displays hotspots information grouped by offload computing tasks and also lists details about transfer times between CPU and GPU:

```
vtune -report hotspots -r ./result_gpu-offload -group-by=computing-task-offload -column='Time' -q
```

```
/vtune/matrix_multiply_vtune# vtune -report hotspots -r ./result_gpu-offload -group-by=computing-task-offload -co
      Total Time  Total Time:Execution  Total Time:Host-to-Device Transfer  Total Time:Device-to-Host Transfer  To
----  ----------  --------------------  ----------------------------------  ----------------------------------  --
          1.146s                1.130s                              0.010s                              0.006s
rier      0.000s                   0s                                  0s                                  0s
```

Run GPU Compute/Media Hotspots Analysis

Our next step is to run the **GPU Compute/Media Hotspots** analysis. This analysis can help us to further explore performance improvements for the GPU-bound application or its stages.

Run GPU Compute/Media Hotspots Analysis

In the CLI, type this command to run the analysis:

```
vtune -collect gpu-hotspots -r ./result_gpu-hotspots -- ./matrix.icpx -fsycl
```

To see the summary report, type:

```
vtune -report summary -r ./result_gpu-hotspots
```

```
pat99-39:/home/vtune/matrix_multiply_vtune# vtune -report summary -r ./result_gpu-hotspots
sing result path `/home/vtune/matrix_multiply_vtune/result_gpu-hotspots'
xecuting actions 75 % Generating a report                              Elapsed Time: 11.174s
Time: 1.151s
 Stalled/Idle: 69.6%
ercentage of time when the EUs were stalled or idle is high, which has a
ive impact on compute-bound applications.

L3 Bandwidth Bound: 11.6%

Hottest GPU Computing Tasks Bound by GPU L3 Bandwidth
Computing Task  Total Time
--------------  ----------
ler Busy: 0.0%

Hottest GPU Computing Tasks with High Sampler Usage
Computing Task  Total Time
--------------  ----------
ization: 10.8%

est GPU Computing Tasks with High FPU Utilization
uting Task  Total Time
----------  ----------
on and Platform Info
ication Command Line: ./matrix.dpcpp
ating System: 5.4.30 DISTRIB_ID=Ubuntu DISTRIB_RELEASE=20.04 DISTRIB_CODENAME=focal DISTRIB_DESCRIPTION="Ubuntu 20
uter Name: nntpat99-39
lt Size: 117,2 MB
ection start time: 12:43:19 20/02/2021 UTC
ection stop time: 12:43:30 20/02/2021 UTC
ector Type: Event-based sampling driver,Driverless Perf system-wide sampling,User-mode sampling and tracing

Name: Intel(R) Processor code named Skylake
Frequency: 2.592 GHz
Logical CPU Count: 8

Name: Iris Pro Graphics 580
Vendor: Intel Corporation
EU Count: 72
Max EU Thread Count: 7
Max Core Frequency: 950.000 MHz
GPU OpenCL Info
    Version
    Max Compute Units: 72
    Max Work Group Size: 256
    Local Memory: 65,5 KB
    SVM Capabilities
```

Generate Report to View Computing Tasks with L3 Metrics

Use this command to generate a report that lists only L3 metrics for computing tasks:

```
vtune -report hotspots -r result_gpu-hotspots -group-by=computing-task -column='L3' -q
```

```
ome/vtune/matrix_multiply_vtune# vtune -report hotspots -r result_gpu-hotspots -group-by=computing-task -column='
.
                 L3 Bandwidth, GB/sec  L3 Sampler Bandwidth, GB/sec  L3 <-> GTI Total Bandwidth, GB/sec  GPU L3
---------------  --------------------  ---------------------------  ----------------------------------  -------
                               59.833                        0.000                              27.542
Barrier                         0.000                        0.000                               0.000
MemoryCopyRegion               62.615                        0.000                              23.892
                                0.003                        0.000                               0.002
```

Run GPU Compute/Media Hotspots Analysis with Dynamic Instruction Count and SIMD Utilization

Run the GPU Compute/Media Hotspots Analysis in the Characterization mode to collect data on dynamic instruction count and SIMD utilization:

```
vtune -collect gpu-hotspots -knob characterization-mode=instruction-count -r ./result_gpu-
hotspots_inst-count -- ./matrix.icpx -fsycl
```

Generate Reports to View Source and Assembly Metrics

- **Source Code for Specific Computing Tasks**

  Use this command to get the source code for a specific computing task:

```
vtune -report hotspots -r result_gpu-hotspots_inst-count -source-object computing-
task="Matrix1_1<float>" -group-by=gpu-source-line -column="Source","GPU Instructions
Executed:Int32 & SP Float" -q
```

```
99-39:/home/vtune/matrix_multiply_vtune# vtune -report hotspots -r result_gpu-hotspots_inst-count -source-object c
loat>" -group-by=gpu-source-line -column="Source","GPU Instructions Executed:Int32 & SP Float" -q
r is ON.
 Source                                                                                GPU Instructions Executed:Int
 ------------------------------------------------------------------------------------  ----------------------------
     accessor accessorB(bufferB, h, read_only);
     accessor accessorC(bufferC, h);

     // Execute matrix multiply in parallel over our matrix_range
     // ind is an index into this range
     h.parallel_for<class Matrix1_1<TYPE>>(matrix_range,[=](cl::sycl::id<2> ind) {
       int k;
       TYPE acc = 0.0;
       for (k = 0; k < NUM; k++) {
         // Perform computation ind[0] is row, ind[1] is col
         acc += accessorA[ind[0]][k] * accessorB[k][ind[1]];
```

- **Assembly Code for Specific Computing Tasks**

  Use this command to get the assembly code for a specific computing task:

```
vtune -report hotspots -r result_gpu-hotspots_inst-count -source-object computing-
task="Matrix1_1<float>" -group-by=address -limit=5 -q
```

```
root@nntpat99-39:/home/vtune/matrix_multiply_vtune# vtune -report hotspots -r result_gpu-hotspots_inst-count -source-object com
ing-task="Matrix1_1<float>" -group-by=address -limit=5 -q
Address  Assembly                                                  Source Line  GPU Instructions Executed  SIMD Utilization(
-------  --------------------------------------------------------  -----------  -------------------------  ----------------
0        Block 1:
0        (W) mov (8|M0) r101.0<1>:ud r0.0<1;1,0>:ud                89                           131,072             100.
0x10     (W) or (1|M0) cr0.0<1>:ud cr0.0<0;1,0>:ud 0x4C0:uw {Switch}  89                           131,072             100.
0x20     (W) mul (1|M0) r10.3<1>:d r14.2<0;1,0>:d r101.6<0;1,0>:d                               131,072             100.
0x30     (W) mul (1|M0) r10.4<1>:d r14.1<0;1,0>:d r101.1<0;1,0>:d                               131,072             100.
```

- **Save Report as CSV File**

  Use the `-report-output` option to save the generated report as a file. To specify the generation of a `.csv` report, use `-format` and `-csv-delimiter` options:

```
vtune -report hotspots -r result_gpu-hotspots_inst-count -source-object computing-task="Matrix1_1<float>" -group-by=address -limit=5 -report-output=result.csv -format=csv -csv-delimiter=comma -q
```

	A	B	C	D	E
1	Address	Assembly	Source Line	GPU Instructions Executed	SIMD Utilization(%
2	0x0	Block 1:			
3	0x0	(W) mov (8|M0) r101.0<1>:ud r0.0<1;1,0>:ud	89	131072	1
4	0x10	(W) or (1|M0) cr0.0<1>:ud cr0.0<0;1,0>:ud 0x4C0:uw {Switch}	89	131072	1
5	0x20	(W) mul (1|M0) r10.3<1>:d r14.2<0;1,0>:d r101.6<0;1,0>:d		131072	1
6	0x30	(W) mul (1|M0) r10.4<1>:d r14.1<0;1,0>:d r101.1<0;1,0>:d		131072	1

Run Custom Analysis with GPU Programming API Statistics

To get a focused analysis of timing and statistics related to GPU compute kernels, follow the GPU Compute/Media Hotspots analysis with a custom analysis that collects **GPU Programming API** statistics.

The kernel data available through this collection is similar to the data you collect when running the `CLIntercept` tool (with DevicePerformanceTiming option enabled) and with the `nvprof` tool in Summary mode.

Collect GPU Programming API Statistics

In the command line, type:

```
vtune -collect-with runss -knob collect-programming-api=true -no-summary -r ./result_gpu-programming-api -- ./matrix.icpx -fsycl
```

Generate Report to View Timing and Statistics for GPU Compute Kernels

This command generates a report that lists timings and instance count for computing tasks. The data is sorted by **Total Time** in descending order.

```
vtune -report hotspots -group-by=source-computing-task -column="Total Time,Average Time,Instance Count" -sort-desc="Total Time" -r ./result_gpu-programming-api/ -q
```

```
va@dtc-nuc-031:/localdisk/mpetrova/matrix_multiply_vtune$ vtune -report hotspots -group-by=source-comput
 Time,Average Time,Instance Count" -sort-desc="Total Time" -r ./result_gpu-programming-api/ -q
 filter is ON.
 Computing Task (GPU)          Computing Task:Total Time  Computing Task:Average Time  Computing Task:In
 ----------------------------  -------------------------  ---------------------------  -----------------
 l<float>                                          0.117s                       0.117s
 andListAppendMemoryCopyRegion                     0.001s                       0.001s
 andListAppendBarrier                              0.000s                       0.000s
```

NOTE
Discuss this recipe in the VTune Profiler developer forum.

See Also

Optimize applications for Intel® GPUs with Intel® VTune Profiler
Optimize Your GPU Application with Intel oneAPI Base Toolkit

GPU Architecture Terminology for Intel® Xe Graphics
GPU Offload Analysis

GPU Offload Analysis from the Command Line
GPU Compute/Media Hotspots Analysis

GPU Compute/Media Hotspots Analysis from the Command Line

Tuning Recipes

These recipes explore typical application performance problems that you can detect with Intel® VTune™ Profiler or its predecessor, Intel® VTune™ Amplifier. Use the guidance in these recipes to optimize performance.

Cache-Related Latency Issues in Segmented Cache Environment

This recipe demonstrates how to use Cache Allocation Technology (CAT) to handle cache-related latency issues (cache misses) when you split a cache between cores.

- INGREDIENTS
- DIRECTIONS:

  1. Run Memory Access analysis
  2. Locate Cache Misses
  3. Use CAT to Reorganize Cache Segments Between Cores

Ingredients

These are the hardware and software tools you need for this performance scenario:

- **Application:**

  A real-time application (RTA) with an allocated buffer that fits into Last Level Cache (LLC). The RTA reads from this buffer continuously.

  RTAs are programs that function within a time frame specified by the user as immediate or current. Real-time programs must guarantee a response within specified time constraints, also known as "deadlines". There can be several reasons for missing a deadline:

  - Preemptions
  - Interrupts
  - An unexpected latency in the critical code execution

  A Real-Time Operating System (RTOS) provides effective solutions to isolate an application to avoid its preemptions and interrupts. However, latency can still result from CPU Microarchitecture issues, like a CPU cache miss penalty. Here is an example of an RTA:

```
struct timespec sleep_timeout =
     (struct timespec) { .tv_sec = 0, .tv_nsec = 10000000 };
...
buffer=malloc(128*1024);
...
run_workload(buffer,128*1024);
void run_workload(void *start_addr,size_t size) {
```

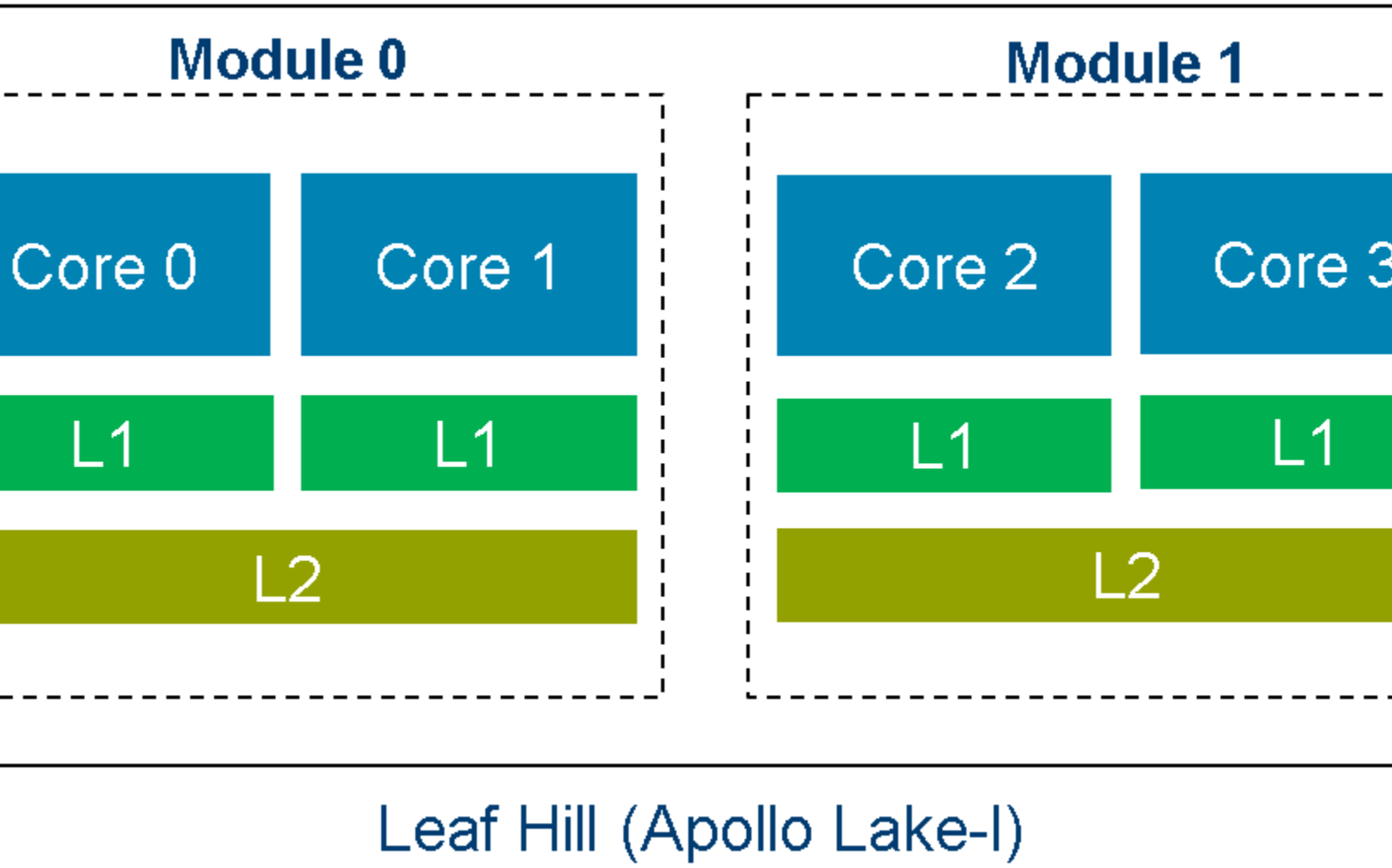
```
unsigned long long i,j;
for (i=0;i<1000;i++)
{
     nanosleep(&sleep_timeout,NULL);
     for (j=0;j<size;j+=32)
     {
        asm volatile("mov(%0,%1,1),%%eax"
            :
        :"r" (start_addr),"r"(i)
            :"%eax","memory");
     }
}
}
```

- **'Noisy Neighbors' Application:** A stress-ng tool to load and stress a cache.
- **Tools:** Intel® VTune™ Profiler - Memory Access Analysis. Set `AMPLXE_EXPERIMENTAL=cat` to enable the **Cache Availability (preview)** feature.

  NOTE

  - Starting with the 2020 release, Intel® VTune™ Amplifier has been renamed to Intel® VTune™ Profiler.
  - Most recipes in the Intel® VTune™ Profiler Performance Analysis Cookbook are flexible. You can apply them to different versions of Intel® VTune™ Profiler. In some cases, minor adjustments may be required.
  - Get the latest version of Intel® VTune™ Profiler:
    - From the Intel® VTune™ Profiler product page.
    - Download the latest standalone package from the Intel® oneAPI standalone components page.

- **Operating system:** Linux* OS
- **Hardware:** Intel Atom® Processor E3900 Series (code named Apollo Lake) Leaf Hill with L2 CAT capabilities enabled.



Run Memory Access Analysis

When you run an RTA in a 'noisy neighbors' environment and notice performance degradation, run Memory Access analysis to investigate microarchitecture issues like CPU cache miss penalties.

In the following examples, the RTA is pinned on Core 3 while noisy neighbors are pinned on Core 2.

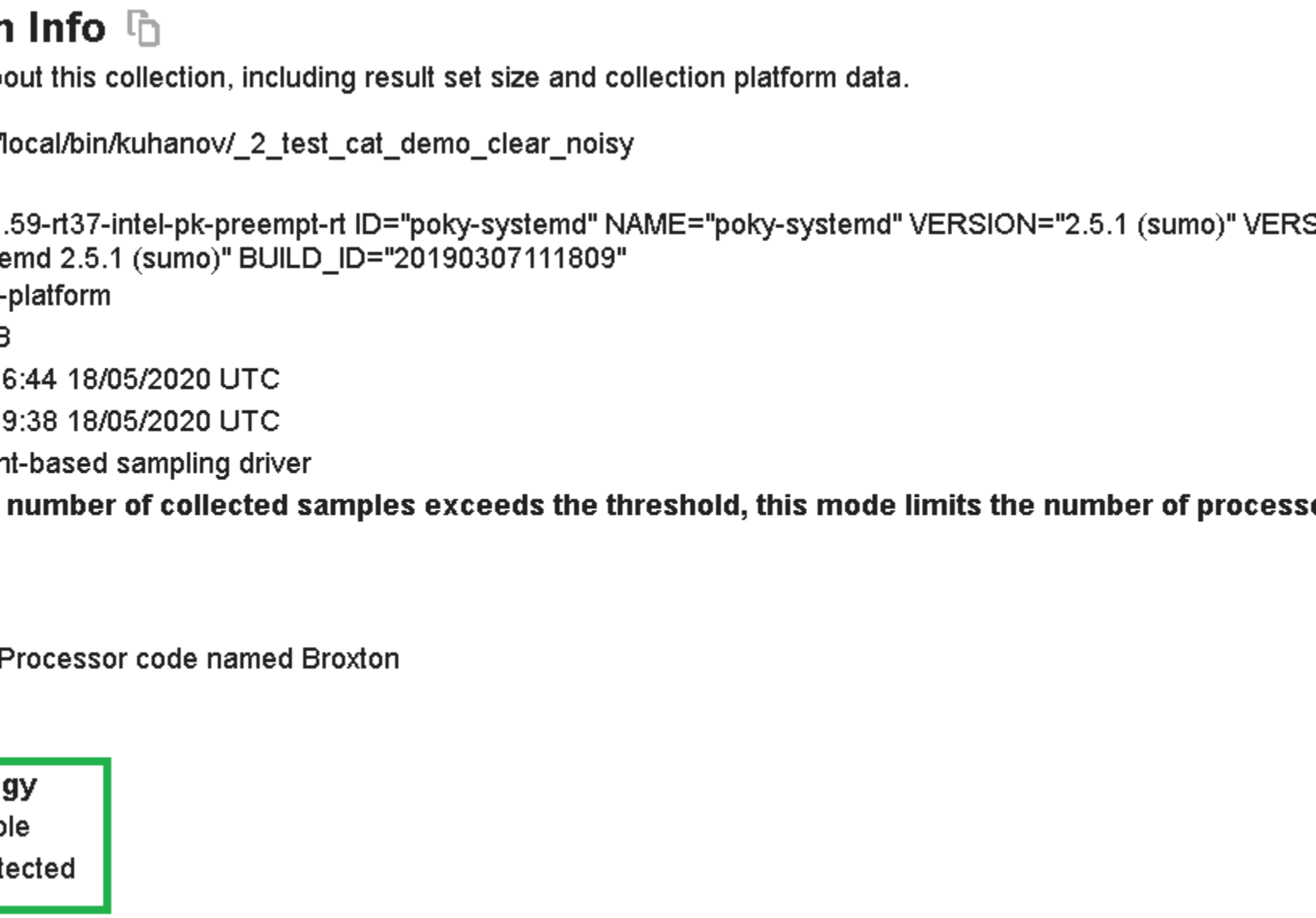
```
stress-ng -C 10 --cache-level 2 --taskset 2 --aggressive -v --metrics-brief
```

Both cores belong to Module 1 and share the L2 LLC.

1. Set `AMPLXE_EXPERIMENTAL=cat` to enable the **Cache Availability (preview)** feature.
2. Open the Intel® VTune™ Profiler GUI.
3. Create a new project. The **Create a Project** dialog box opens.
4. Specify a project name, a location for your project, and click **Create Project**. The **Configure Analysis** window opens.
5. In the **WHERE** pane, select **Remote Linux(SSH)** as the target system for the analysis.
6. Set up a passwordless connection to the Linux target.
7. In the **WHAT** pane, select **Launch Application** and specify your target application for analysis.
8. In the **HOW** pane, click the analysis header and select **Memory Access analysis** from the Analysis Tree.
9. Set **Analyze cache allocation** to catch cache segment usage by cores.
10. Click the **Start** button to launch the analysis.

Locate Cache Misses

Once Intel® VTune™ Profiler completes the analysis, see the **Collection and Platform Info** section in the **Summary** pane. Here, you can see information about CPU support for L2 and L3 Cache Allocation Technology (CAT). In this example, the hardware allows for a split manipulation on LLC (L2 cache).



Next, switch to the **Bottom-Up** pane. Select **Module / Function / Call Stack** grouping and check the LLC Miss Count value for the `cache_sample` module.

g: Module / Function / Call Stack

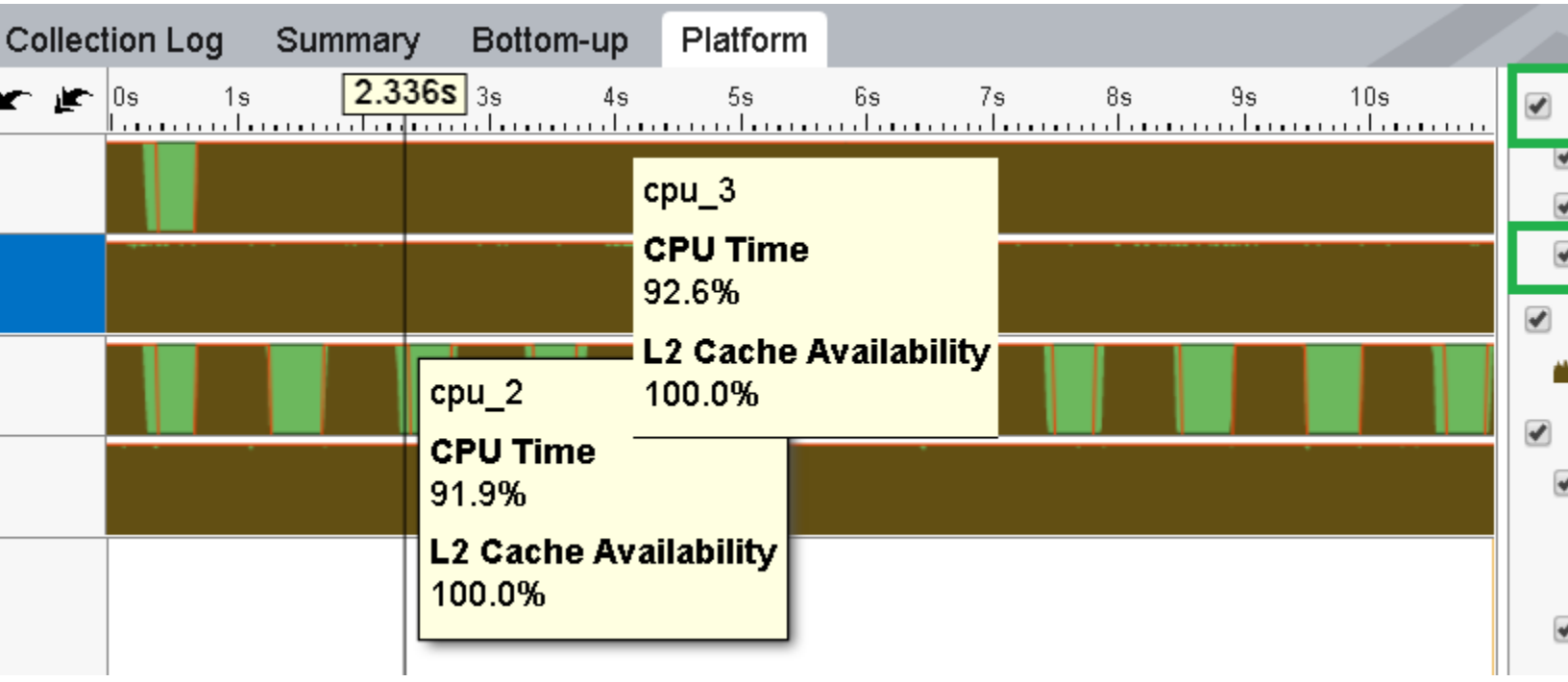
Module / Function / Call Stack	warding	4K Aliasing	LLC Miss Count	Average L2 Cache Ava
	0.0%	100.0%	40,000	
_sample	0.0%	0.0%	460,000	
lxe_boost_filesystem_1.70.so	0.0%	0.0%	20,000	
-ng	0.0%	0.0%	15,160,000	

This is a high number of cache misses. However, there are zero cache misses in the absence of noisy neighbors.

: Module / Function / Call Stack

Module / Function / Call Stack	CPU Time	LLC Miss Count	Average L2 Cache Av
_sample	0.026s	0	

Now, open the **Platform** pane. Group results by **Logical Core/Thread** and select the **L2 Cache Availability** checkbox.



The timeline informs us that the L2 Cache Availability is 100%. This means that all segments of L2 cache were available for the entire lifetime of the target application on **cpu_2**. However, noisy neighbors from **cpu_3** also shared the L2 cache for the entire lifetime (100%). This is an important factor responsible for the enormous amount of cache misses in the `cache_sample` module.

You can use CAT to split cache segments between cores so that each core has a segment for exclusive use.

Use CAT to Reorganize Cache Segments Between Cores

Split the cache to give a segment to the RTA and the rest to noisy neighbors. With CAT, you can program software to control the amount of cache used by a thread, application, virtual machine, or container. You can isolate cache segments and thereby address concerns of sharing resources. You can do this in two ways:

- Configure the MSRs directly. See Section 17.19 of the Intel® 64 and IA-32 Architectures Software Developer's Manual (Volume 3B).
- Use Resource Control (resctrl) - A kernel interface for CPU Resource Allocation.

In this example, we use Resource Control to assign:

- 1 cache segment to **cpu_3**.
- 7 cache segments to **cpu_2**.

```
#set '00000001' Capacity Bit Mask for CORE 3
mkdir /sys/fs/resctrl/clos0
echo 8 > /sys/fs/resctrl/clos0/cpus
echo 'L2:1=1' > /sys/fs/resctrl/clos0/schemata
#set '11111110' CBM for rest CORE
echo 'L2:1=fe' > /sys/fs/resctrl/schemata
```

Run Memory Access analysis again. Once the collection completes, see the **Bottom-Up** pane.

ng: Module / Function / Call Stack

Module / Function / Call Stack	CPU Time	Average L2 Cache Availability	LLC Miss Cou
e_sample	0.409s	12.5%	380
	0.014s	86.5%	20
27.so	0.016s	100.0%	
erf3	0.004s	100.0%	
o]	0.004s	100.0%	
stemd-shared-237.so	0.011s	100.0%	
	0.002s	100.0%	

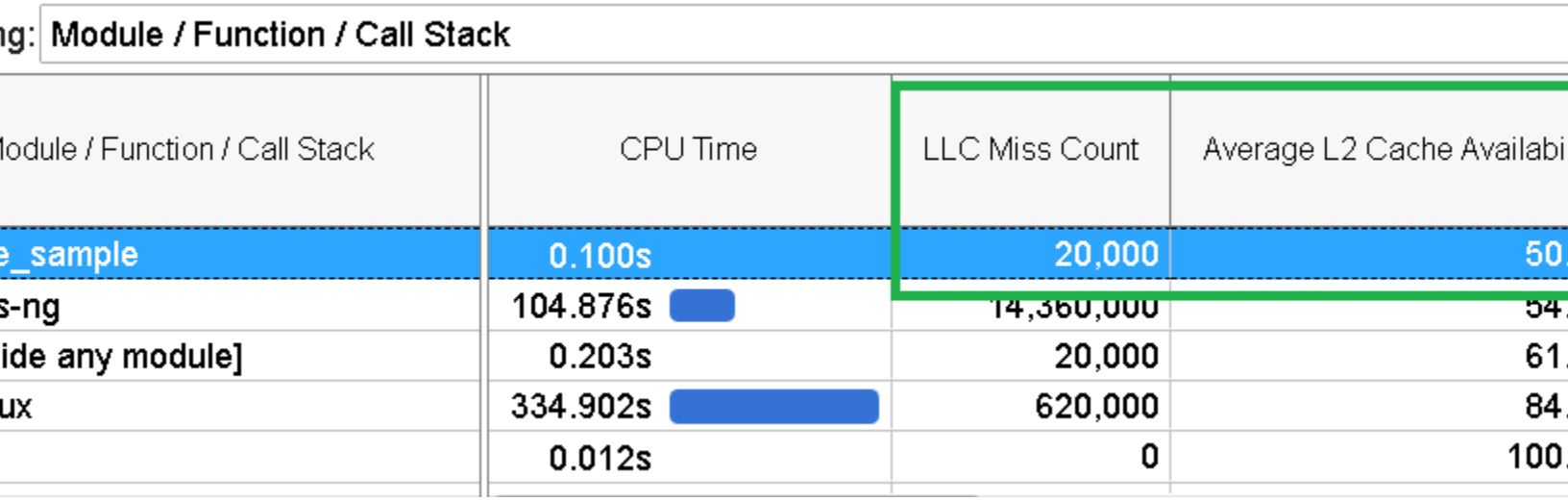
A small fraction of the cache (12.5%) is available exclusively. However, the count of cache misses has not improved significantly.

Let us devote more cache to the application and try the analysis again. Increase the cache allocation to 50% or 4 segments.

```
#set '00001111' Capacity Bit Mask for CORE 3
mkdir /sys/fs/resctrl/clos0
echo 8 > /sys/fs/resctrl/clos0/cpus
```

```
echo 'L2:1=f' > /sys/fs/resctrl/clos0/schemata
#set '11110000' CBM for rest CORE
echo 'L2:1=f0' > /sys/fs/resctrl/schemata
```

With increased cache allocation, the **Bottom-Up** pane shows that the count of cache misses has reduced significantly. But this comes at a price since we have now dedicated one half of the entire available cache for the application exclusively.



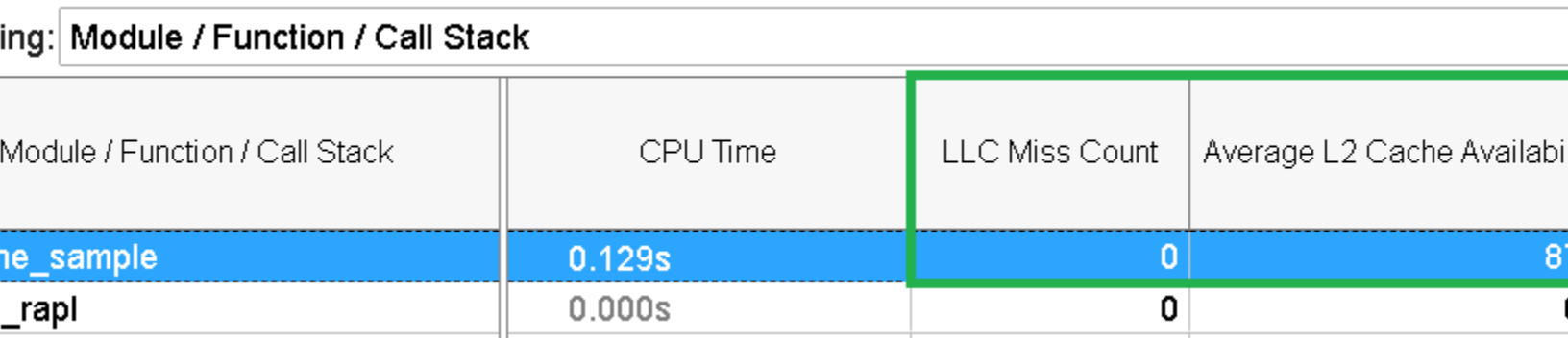
It is possible that the exit is in the use of *Pseudo-Locking* technique. Pseudo-locking helps to protect data eviction from the cache by other processes that attempt to use the same cache. The RTA can allocate critical data to a special segment of the cache and protect it from use by another thread, process, or core. While the segment is hidden, we still have access to data in the segment.

```
#Create the pseudo-locked region with 1 cache segment
mkdir /sys/fs/resctrl/demolock
echo pseudo-locksetup > /sys/fs/resctrl/demolock/mode
echo 'L2:1=0X1' > /sys/fs/resctrl/demolock/schemata
cat /sys/fs/resctrl/demolock/mode
pseudo-locked
```

```
struct timespec sleep_timeout =
     (struct timespec) { .tv_sec = 0, .tv_nsec = 10000000 };
...
/* buffer=malloc(128*1024); */
open("/dev/pseudo_lock/demolock",0_RDWR);
buffer=mmap(0,128*1024,PROT_READ|PROT_WRITE,MAP_SHARED,dev_fd,0);
...
run_workload(buffer,128*1024);
void run_workload(void *start_addr,size_t size) {
unsigned long long i,j;
for (i=0;i<1000;i++)
{
     nanosleep(&sleep_timeout,NULL);
     for (j=0;j<size;j+=32)
     {
        asm volatile("mov(%0,%1,1),%%eax"
            :
        :"r" (start_addr),"r"(i)
            :"%eax","memory");
```

```
        }
    }
}
```

Run the analysis again. Open the **Bottom-Up** pane to see results.



Locking just a single segment on the system helped to resolve *all* of the cache misses.

Cache Allocation Technology is very useful in real-time environments or workloads where a small latency (caused by memory accesses) is critical, irrespective of its size. As described in this recipe, allocate cache segments in iterative runs until you observe zero cache misses.

NOTE
Discuss this recipe in the Analyzers developer forum.

See Also

Microarchitecture Exploration Analysis

Memory Access Analysis

Introduction to Cache Allocation Technology (CAT)

False Sharing

This recipe explores profiling a memory-bound `linear_regression` *application using the General Exploration and Memory Access analyses of the Intel© VTune™ Amplifier.*

- INGREDIENTS
- DIRECTIONS:

1. Run General Exploration analysis
2. Identify a bottleneck
3. Find a contended data structure
4. Fix the false sharing issue

NOTE
General Exploration analysis is renamed to Microarchitecture Exploration analysis starting with Intel VTune Amplifier 2019.

Ingredients

This section lists the hardware and software tools used for the performance analysis scenario.

- **Application:** `linear_regression`. The `linear_regression.tgz` sample package is available with the product in the `<install-dir>/samples/en/C++` directory and at https://github.com/kozyraki/phoenix/tree/master/sample_apps/linear_regression.
- **Performance analysis tools:**
  - Intel VTune Amplifier 2018: General Exploration, Memory Access analysis

    NOTE

    - Starting with the 2020 release, Intel® VTune™ Amplifier has been renamed to Intel® VTune™ Profiler.
    - Most recipes in the Intel® VTune™ Profiler Performance Analysis Cookbook are flexible. You can apply them to different versions of Intel® VTune™ Profiler. In some cases, minor adjustments may be required.
    - Get the latest version of Intel® VTune™ Profiler:
      - From the Intel® VTune™ Profiler product page.
      - Download the latest standalone package from the Intel® oneAPI standalone components page.

- **Operating system:** Linux*, Ubuntu* 16.04 64-bit
- **CPU:** Intel® Core™ i7-6700K processor

Run General Exploration Analysis

To have a high-level understanding of potential performance bottlenecks for the sample, start with the General Exploration analysis provided by the VTune Amplifier:

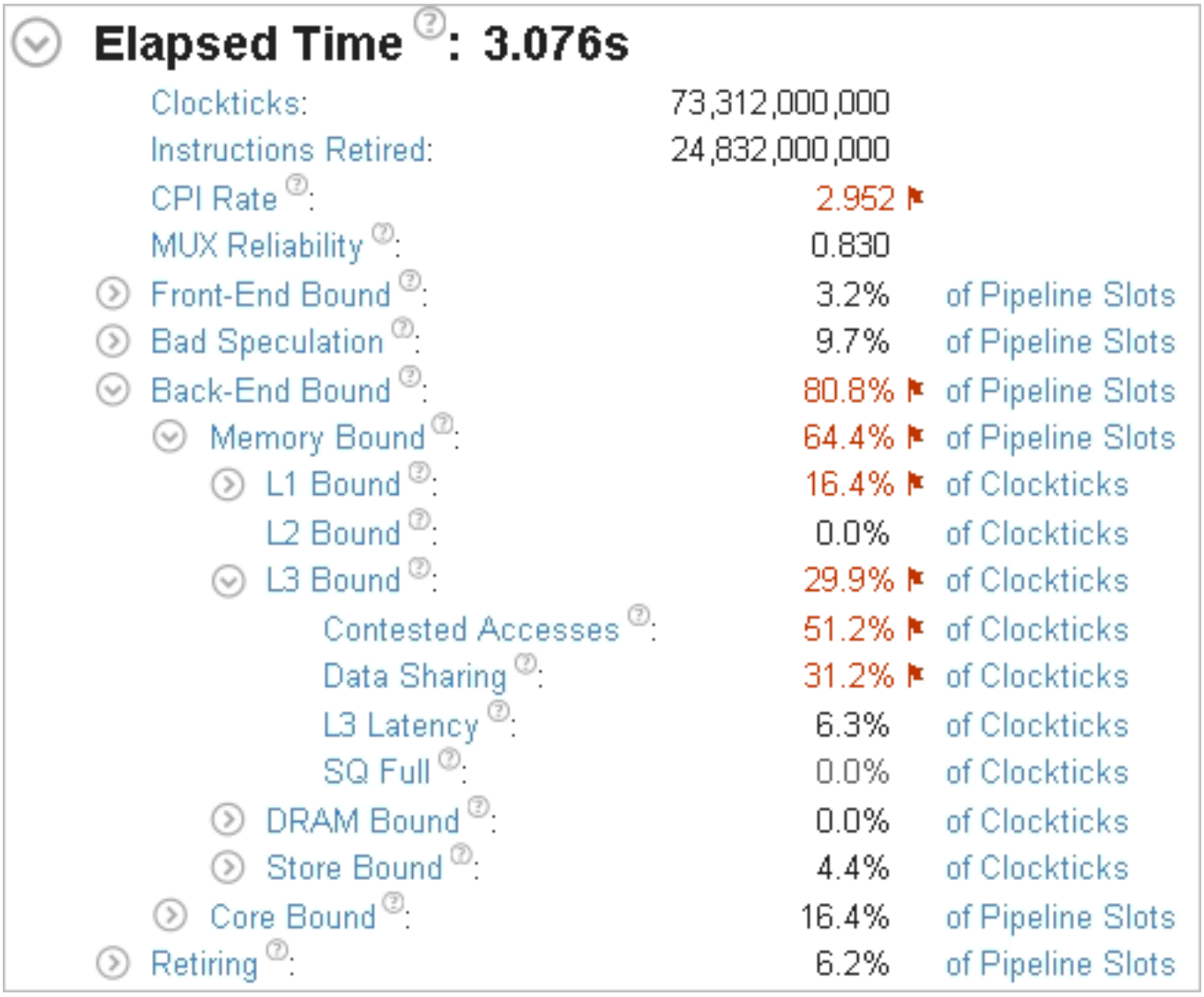
1. Click the **New Project** button on the toolbar and specify a name for the new project, for example: `linear_regression`.
2. In the **Analysis Target** window, select the **local host** target system type for the host-based analysis.
3. Select the **Launch Application** target type and specify an application for analysis on the right.
4. Click the **Choose Analysis** button on the right, select **Microarchitecture Analysis** > **General Exploration** and click **Start**.

   VTune Amplifier launches the application, collects data, finalizes the data collection result resolving symbol information, which is required for successful source analysis.

Identify a Bottleneck

Start with the **Summary** view that provides application-level statistics per hardware metrics.

Typically, for performance analysis you are recommended to create a *baseline* to measure your future optimizations. In this case, consider the application Elapsed Time as your baseline:



A brief analysis of the summary metrics shows that the application is mostly bound by contested memory accesses.

Find a Contended Data Structure

High value for the **Contested Accesses** metric prompts you to dig deeper and run the Memory Access analysis with the **Analyze dynamic memory objects** option enabled. This analysis helps you find out an access to what data structure caused contention issues:

Top Memory Objects by Latency

This section lists memory objects that introduced the highest latency to the overall application execution.

Memory Object	Total Latency	Loads	Stores	LLC Miss Count
stddefines.h:52 (512 B)	73.6%	9,515,385,453	1,643,624,654	0
[Stack]	20.4%	3,320,899,624	5,600,084	0
lreg-pthread!main (517 MB)	5.7%	1,220,836,624	0	0
[Stack]	0.2%	32,200,966	21,000,315	0
[Unknown]	0.0%	700,021	0	0
[Others]	0.0%	2,800,084	0	0

From the Summary view, you see that a memory allocation data object in file `stddefines.h` at line 52 introduced the highest latency to the application execution. The size of the allocation is quite small - only 512 bytes, so it should fit fully into the L1 cache. For more details, click this object in the table to switch to the Bottom-up view:

Grouping: Memory Object / Function / Allocation Stack

Memory Object / Function / Allocation ...	CP...	Loads ▼	Stores	LLC Miss Count	Average Latency (cycles)
stddefines.h:52 (512 B)		9,515,385,453	1,643,624,654	0	59
[Stack]		3,320,899,624	5,600,084	0	8
lreg-pthread!main (517 MB)		1,220,836,624	0	0	8
[Stack]		32,200,966	21,000,315	0	9
[lreg-pthread]		2,800,084	0	0	0
[Unknown]		700,021	0	0	2

The average access latency to this object is 59 cycles, which is a very high value for the memory size that is expected to reside in the L1 cache. This can be the source for the contested accesses performance problem.

Expand the **stddefines.h:52 (512B)** memory object in the grid to view the allocation stack. Double-click the allocation stack to go deeper to the Source view that highlights the line where the object is allocated:

```
124     num_threads = num_procs;
125
126     printf("Linear Regression P-Threads: Running...\n");
127     printf("lreg_args size: %d\n", sizeof(lreg_args));
128
129     POINT_T *points = (POINT_T*)fdata;
130     long long n = (long long) finfo.st_size / sizeof(POINT_T);
131
132     req_units = n / num_threads;
133     tid_args = (lreg_args *)CALLOC(sizeof(lreg_args), num_procs);
134     //tid_args = (lreg_args *)_mm_malloc(sizeof(lreg_args)*num_procs, 64);
135     //memset(tid_args, 0, sizeof(lreg_args) * num_procs);
```

Where `lreg_args` is:

```
typedef struct
{
    pthread_t tid;
    POINT_T *points;
    int num_elems;
    long long SX;
    long long SY;
    long long SXX;
    long long SYY;
    long long SXY;
} lreg_args;
```

Threads code accessing the `lreg_args` array looks like this:

```
// ADD Up RESULTS
for (i = 0; i < args->num_elems; i++)
{
    //Compute SX, SY, SYY, SXX, SXY
    args->SX  += args->points[i].x;
    args->SXX += args->points[i].x*args->points[i].x;
    args->SY  += args->points[i].y;
    args->SYY += args->points[i].y*args->points[i].y;
    args->SXY += args->points[i].x*args->points[i].y;
}
```

Each thread is independently accessing its element in the array, which looks like false sharing.

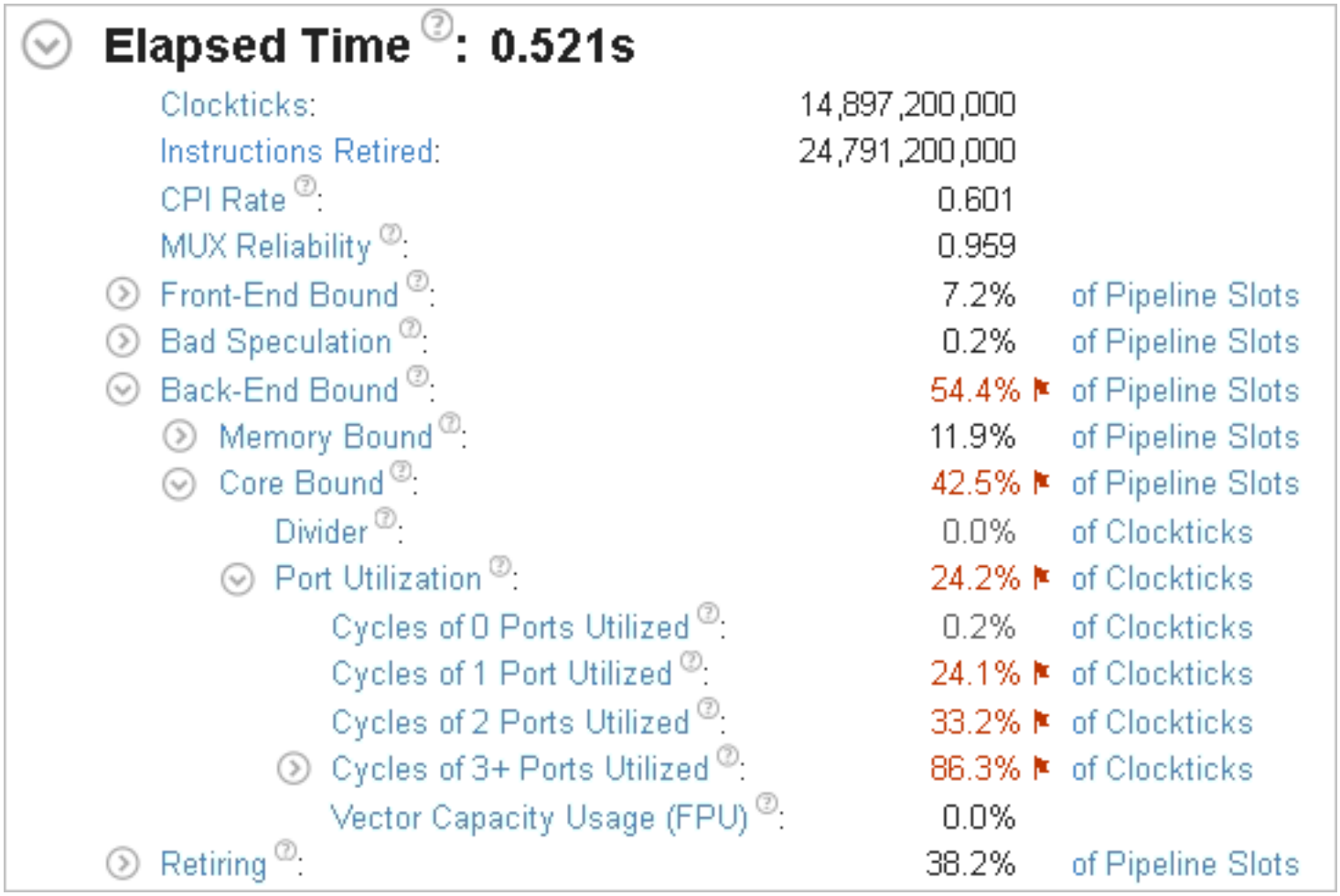
The size of the `lreg_args` structure in the sample is 64 bytes, which matches the cacheline size. But when you allocate an array of these structures, there is no guarantee that this array will be aligned with 64 bytes. As a result, array elements may cross cacheline boundaries, which triggers an unintended contention issue - false sharing.

Fix False Sharing Issue

To fix this false sharing problem, switch to an `_mm_malloc` function, which is used to allocate memory with 64 bytes alignment:

```
50  inline void * CALLOC(size_t num, size_t size)
51  {
52      void * temp =  mm_malloc(num*size, 64);
53      assert(temp);
54      memset(temp, 0, num*size);
55      return temp;
56  }
```

Re-compiling and re-running the application analysis with VTune Profiler provides the following result:



The Elapsed time is now 0.5 seconds, which is a significant improvement from original 3 seconds. The Memory Bound bottleneck went away. The false sharing performance issue is successfully fixed.

NOTE
To discuss this recipe, visit the developer forum.

See Also

Microarchitecture Exploration Analysis

Memory Access Analysis

Top-down Microarchitecture Analysis Method

Frequent DRAM Accesses

This recipe explores profiling a memory-bound `matrix` application using the Microarchitecture Exploration and Memory Access analyses of the Intel® VTune™ Profiler to understand the cause of the frequent DRAM accesses.

- INGREDIENTS
- DIRECTIONS:

  1. Create a baseline
  2. Run Microarchitecture Exploration analysis
  3. Identify hardware hotspots
  4. Run Memory Access analysis
  5. Identify hot memory accesses
  6. Apply loop interchange for optimization

Ingredients

This section lists the hardware and software tools used for the performance analysis scenario.

- **Application:** `matrix multiplication` sample that multiplies 2 matrices of 2048x2048 size, matrix elements have the double type. The `matrix` sample package is available with the product in the `<install-dir>/samples/en/C++` directory and from the Intel Developer Zone at GitHub repository.
- **Performance analysis tools:**
  - Intel® VTune™ Profiler version 2019 or newer: Microarchitecture Exploration (formerly, General Exploration), Memory Access analysis

  NOTE

  - Starting with the 2020 release, Intel® VTune™ Amplifier has been renamed to Intel® VTune™ Profiler.
  - Most recipes in the Intel® VTune™ Profiler Performance Analysis Cookbook are flexible. You can apply them to different versions of Intel® VTune™ Profiler. In some cases, minor adjustments may be required.
  - Get the latest version of Intel® VTune™ Profiler:
    - From the Intel® VTune™ Profiler product page.
    - Download the latest standalone package from the Intel® oneAPI standalone components page.

- **Operating system:** Linux*, Ubuntu* 16.04 64-bit
- **CPU:** Intel® Core™ i7-6700K processor

Create a Baseline

The initial version of the sample code provides a naïve multiplication algorithm with the following code for the main kernel:

```
void multiply1(int msize, int tidx, int numt, TYPE a[][NUM], TYPE v[][NUM], TYPE c[][NUM], TYPE
t[][NUM])
{
    int i,j,k;

    // Naive implementation
```

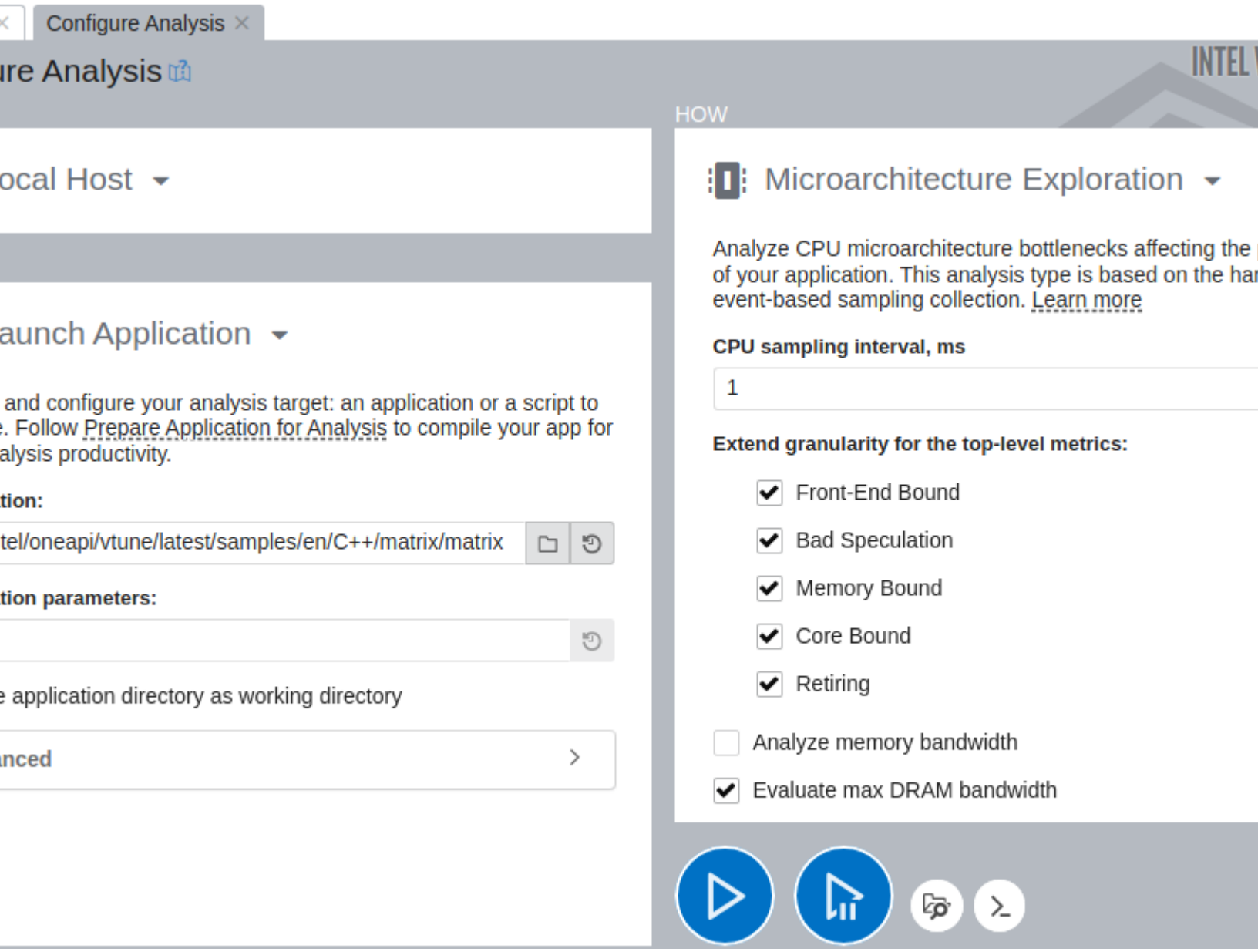
```
    for(i=tidx; i<msize; i=i+numt) {
        for(j=0; j<msize; j++) {
            for(k=0; k<msize; k++) {
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
            }
        }
    }
}
```

Running the compiled application takes about 22 seconds. This is a performance *baseline* that could be used for further optimizations.

Run Microarchitecture Exploration Analysis

To have a high-level understanding of potential performance bottlenecks for the sample, start with the Microarchitecture Exploration analysis provided by Intel® VTune™ Profiler:

1. Click the  **New Project** button on the toolbar and specify a name for the new project, for example: `matrix`.
2. In the **Configure Analysis** window, select the **Local Host** target system type on the **WHERE** pane.
3. On the **WHAT** pane, select the **Launch Application** target type and specify an application for analysis.
4. On the **HOW** pane, click the browse button and select **Microarchitecture Exploration** analysis from the **Microarchitecture** group.

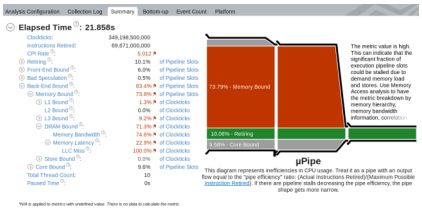


5. Click the  **Start** button.

   VTune Profiler launches the application, collects data, finalizes the data collection result resolving symbol information, which is required for successful source analysis.

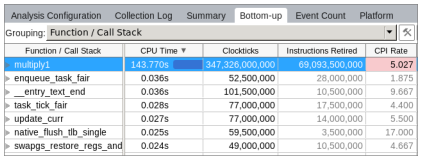
Identify Hardware Hotspots

Microarchitecture Exploration helps you see dominant performance bottlenecks in your code. Start your analysis with the **µPipe** representation in the **Summary** view that displays CPU microarchitecture efficiency and CPU pipeline stalls for the analyzed application. According to the **µPipe** below, the output pipe flow is very narrow, which means that the **Retiring** metric value needs to be increased to improve application performance. The primary obstacle in the pipe is the **Memory Bound** metric value:



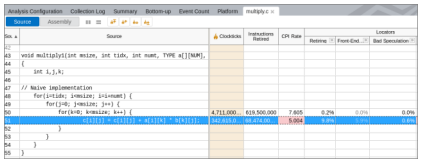
From the metric tree on the left, you see that performance is mostly bound by access to the DRAM.

When you switch to the Bottom-up view, you see that the application has one big hotspot function `multiply1`:



Function / Call Stack	CPU Time ▼	Clockticks	Instructions Retired	CPI Rate
multiply1	143.770s	347,326,000,000	69,093,500,000	5.027
enqueue_task_fair	0.036s	52,500,000	28,000,000	1.875
__entry_text_end	0.036s	101,500,000	10,500,000	9.667
task_tick_fair	0.028s	77,000,000	17,500,000	4.400
update_curr	0.027s	77,000,000	14,000,000	5.500
native_flush_tlb_single	0.025s	59,500,000	3,500,000	17.000
swapgs_restore_regs_and	0.024s	49,000,000	10,500,000	4.667

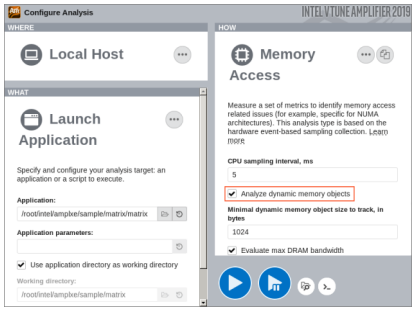
Double-clicking this function opens the Source view that highlights the most performance-critical code line:



Almost all the time was spent in source line #51 that operates over three arrays - a, b, and c.

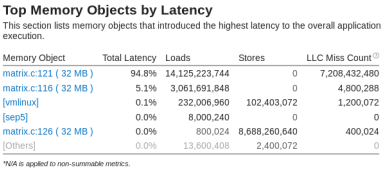
Run Memory Access Analysis

To find out an access to what array was the most expensive, run the Memory Access analysis with the **Analyze dynamic memory objects** option enabled:



Identify Hot Memory Accesses

The **Summary** window for the Memory Access analysis result shows the top memory objects as follows:



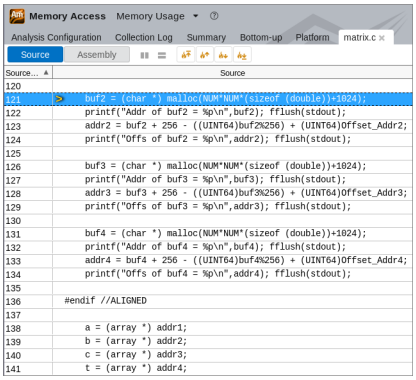
Top Memory Objects by Latency

This section lists memory objects that introduced the highest latency to the overall application execution.

Memory Object	Total Latency	Loads	Stores	LLC Miss Count
matrix.c:121 (32 MB)	94.8%	14,125,223,744	0	7,208,432,480
matrix.c:116 (32 MB)	5.1%	3,061,691,848	0	4,800,288
[vmlinux]	0.1%	232,006,960	102,403,072	1,200,072
[sep5]	0.0%	8,000,240	0	0
matrix.c:126 (32 MB)	0.0%	800,024	8,688,260,640	400,024
[Others]	0.0%	13,600,408	2,400,072	0

*N/A is applied to non-summable metrics.

Click the first hotspot object `matrix.c:121` in the list to switch to the Bottom-up view and then double-click this object highlighted in the grid to open the Source view and see the line allocating this memory object:



You see the allocation for the `buf2` variable that is assigned to `addr2`, which is in its turn assigned to array `b`. So, you may conclude that the problematic array is `b`. Click the



Open Source File Editor button on the toolbar and have a look at the code again:

```
void multiply1(int msize, int tidx, int numt, TYPE a[][NUM], TYPE v[][NUM], TYPE c[][NUM], TYPE
t[][NUM])
{
    int i,j,k;

    // Naive implementation
    for(i=tidx; i<msize; i=i+numt) {
        for(j=0; j<msize; j++) {
            for(k=0; k<msize; k++) {
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
            }
        }
    }
}
```

You see now the root cause of the problem: the innermost cycle iterates over array `b` in an inefficient way. On each iteration it jumps over big chunks of memory.

Apply Loop Interchange for Optimization

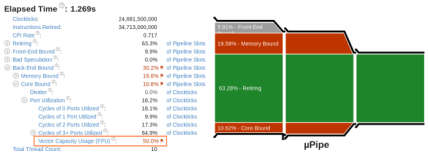
Apply the loop interchange algorithm to `j` and `k` as follows:

```
for(i=tidx; i<msize; i=i+numt) {
                for(k=0; k<msize; k++) {
                                for(j=0; j<msize; j++) {
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
        }
    }
}
```

Compiling and running the new code will result in 1.3-second runtime, which is a significant 20x improvement over original 26 seconds.

What's Next

Re-run the Microarchitecture Exploration analysis on the optimized `matrix` code. The **µPipe** diagram shows a significant increase of the **Retiring** metric value, from 10.06% to 63.28%:



You can focus on other flagged metrics to identify further areas for improvement, for example: poor port utilization.

See Also

Microarchitecture Exploration Analysis for Hardware Issues

Memory Access Analysis

Top-down Microarchitecture Analysis Method Use this recipe to know how an application is utilizing available hardware resources and how to make it take advantage of CPU microarchitectures. One way to obtain this knowledge is by using on-chip Performance Monitoring Units (PMUs).

Poor Port Utilization

Profile a core-bound `matrix` application using the Microarchitecture Exploration analysis in Intel® VTune™ Profiler. Understand the cause for poor port utilization and use Intel® Advisor to benefit from compiler vectorization.

Content expert: Jeffrey Reinemann

- INGREDIENTS
- DIRECTIONS:

  1. Create baseline
  2. Run Microarchitecture Exploration analysis
  3. Identify a cause for poor port utilization
  4. Explore options for vectorization
  5. Compile with the latest instruction set

Ingredients

This section lists the hardware and software tools used for the performance analysis scenario.

- **Application:** The `matrix` multiplication sample that multiplies 2 matrices of 2048x2048 size, matrix elements have the double type. Find the `matrix` sample package in the VTune Profiler package in the `<install-dir>/samples/en/C++` directory or download it from the GitHub samples repository.
- **Performance analysis tools:**
  - Intel VTune Profiler-Microarchitecture Exploration analysis
  - Intel Advisor-Vectorization analysis
- **Operating system:** Linux*, Ubuntu 22.04.2 LTS
- **CPU:** Intel® Core™ i7-6700K processor code named SkyLake (6th generation)

Create Baseline

Optimize the initial version of the `matrix` code with a naïve multiplication algorithm. See the Frequent DRAM Accesses recipe), the execution time has reduced from 22 seconds to 1.3 seconds. This is a new performance baseline for further optimizations.

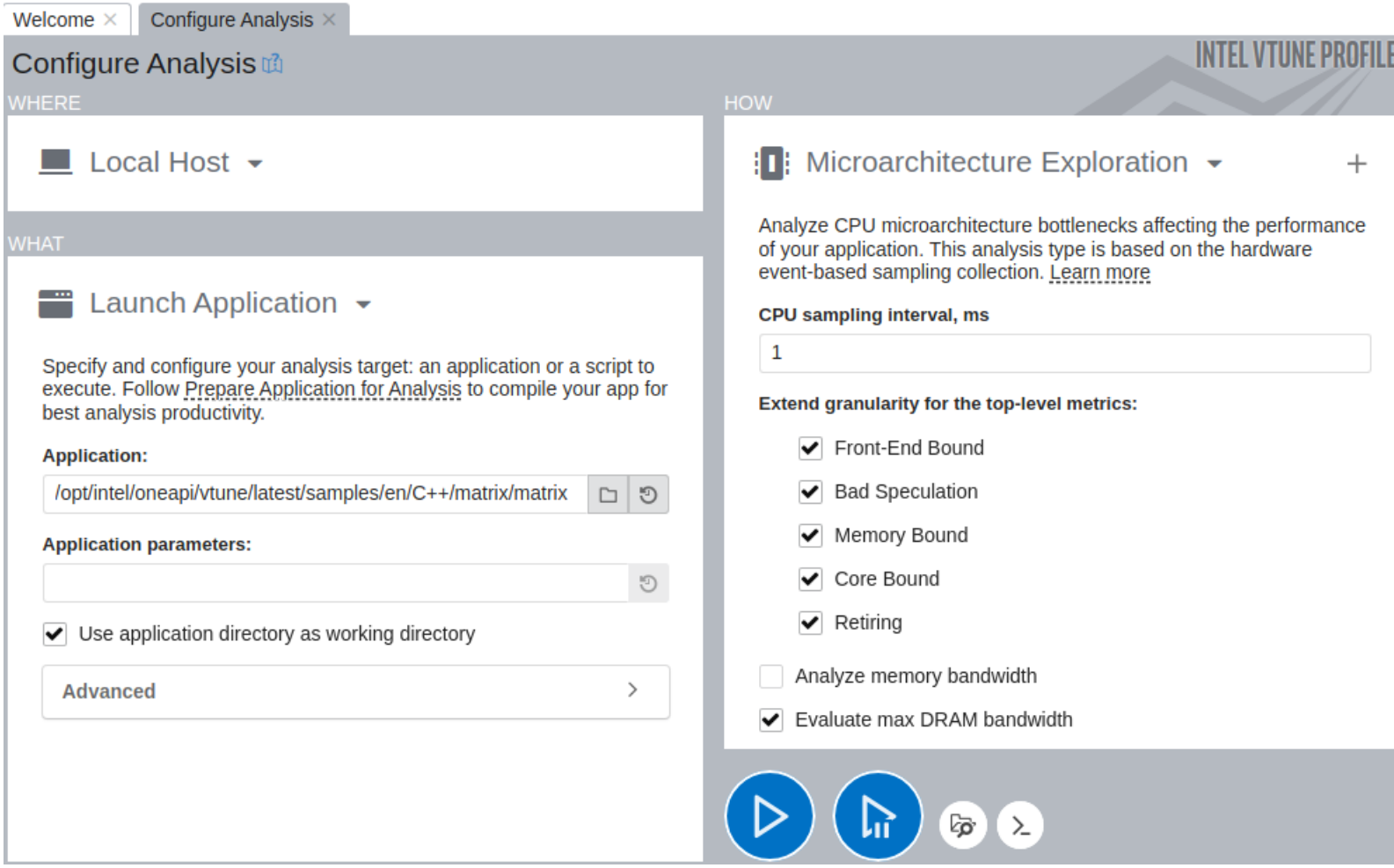
Run Microarchitecture Exploration Analysis

Next, run the **Microarchitecture Exploration** analysis to get a high-level understanding of potential performance bottlenecks in the sample application:

1. Click the 

   New Project button on the toolbar and specify a name for the new project, for example: `matrix`.
2. In the **Configure Analysis** window, make these selections:
   - In the **WHERE** pane, select the **Local Host** target system type.
   - In the **WHAT** pane, select the **Launch Application** target type and specify an application for analysis.
   - In the **HOW** pane, select the **Microarchitecture Exploration** analysis type.

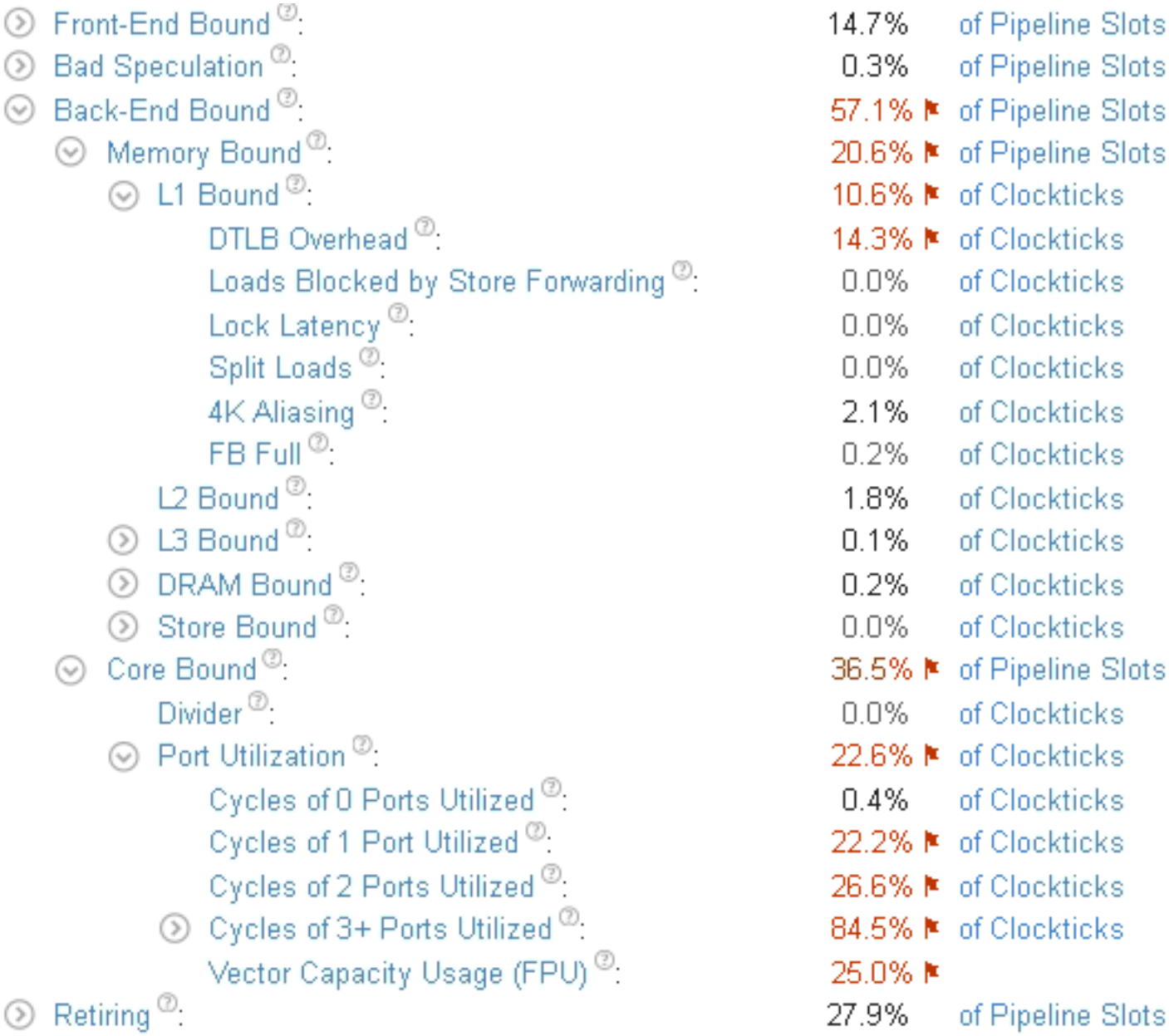
   NOTE For short duration workloads(~under 5 seconds) like the optimized version of the `matrix` application, you may get more accurate values by reducing the sampling interval to 0.5 seconds.
3. Click **Start** to run the analysis.



Once VTune Profiler collects data and finalizes results (after resolving symbol information for successful source analysis), you are ready to examine the causes for poor port utilization.

Understand Poor Port Utilization

Start with the **Summary** view that shows high-level statistics for the application performance per hardware metrics:



You see that the dominant bottleneck has moved to **Core Bound** > **Port Utilization**. More than 3 execution ports were utilized simultaneously for the majority of the time. The **Vector Capacity Usage** metric value is also flagged as critical, which means that the code was either not vectorized or vectorized poorly. To confirm this, switch to the Assembly view of the kernel as follows:

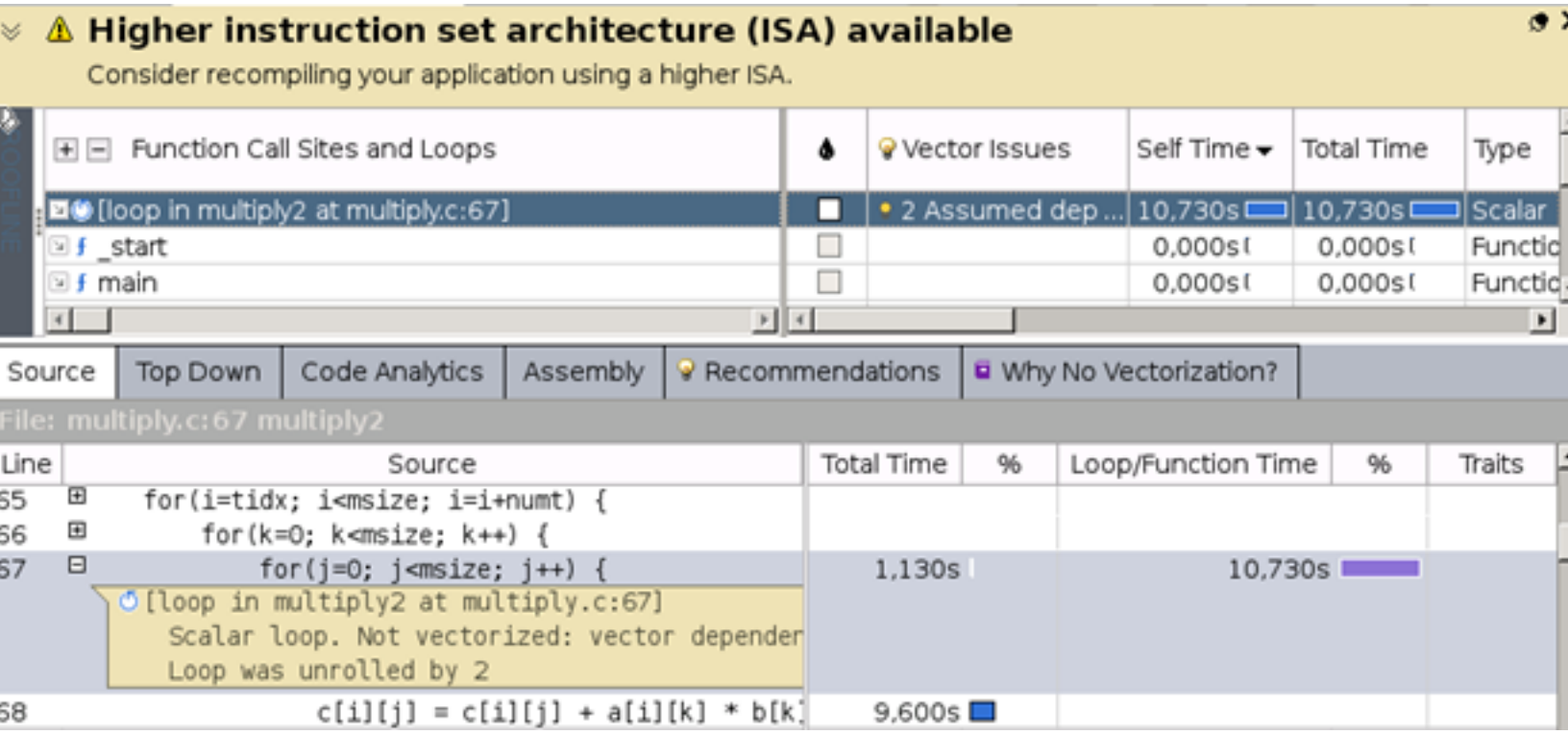
1. Click the **Vector Capacity Usage (FPU)** metric to switch to the Bottom-up view sorted by this metric.
2. Double-click the hot `multiply1` function to open its Source view.
3. Click the **Assembly** button on the toolbar to view the disassembly code:

0x4016ca	68	movsd xmm0, qword ptr [r12]	1,200,000	0
0x4016d0	67	inc rbx	94,800,000	91,200,000
0x4016d3	68	mulsd xmm0, qword ptr [r11+rcx*1]	4,330,800,000	7,659,200,000
0x4016d9	68	addsd xmm0, qword ptr [r11+r15*1]	4,378,400,000	2,163,200,000
0x4016df	68	movsd qword ptr [r11+r15*1], xmm0	11,392,400, ...	4,576,800,000
0x4016e5	68	movsd xmm1, qword ptr [r12]	3,642,800,000	9,865,200,000
0x4016eb	68	mulsd xmm1, qword ptr [r11+rcx*1+0x8]	988,800,000	836,000,000
0x4016f2	68	addsd xmm1, qword ptr [r11+r15*1+0x8]	2,749,600,000	1,459,600,000
0x4016f9	68	movsd qword ptr [r11+r15*1+0x8], xmm1	9,280,000,000	6,056,800,000
0x401700	67	add r11, 0x10	4,434,000,000	18,955,600, ...
0x401704	67	cmp rbx, rdx	38,400,000	1,200,000
0x401707	67	jb 0x4016ca <Block 10>		
0x401709		**Block 11:**		
0x401709	68	lea ecx, ptr [rbx+rbx*1+0x1]		
0x40170d		**Block 12:**		
0x40170d	67	lea ebx, ptr [rcx-0x1]	1,200,000	2,000,000
0x401710	67	cmp ebx, edi	2,800,000	7,200,000
0x401712	67	jnb 0x401736 <Block 14>		
0x401714		**Block 13:**		
0x401714	68	movsxd rcx, ecx		
0x401717	68	movsd xmm0, qword ptr [r12]		
0x40171d	68	lea rbx, ptr [r8+rcx*8]		
0x401721	68	mulsd xmm0, qword ptr [rbx+r14*1-0x8]		
0x401728	68	addsd xmm0, qword ptr [r15+rcx*8-0x8]		
0x40172f	68	movsd qword ptr [r15+rcx*8-0x8], xmm0		

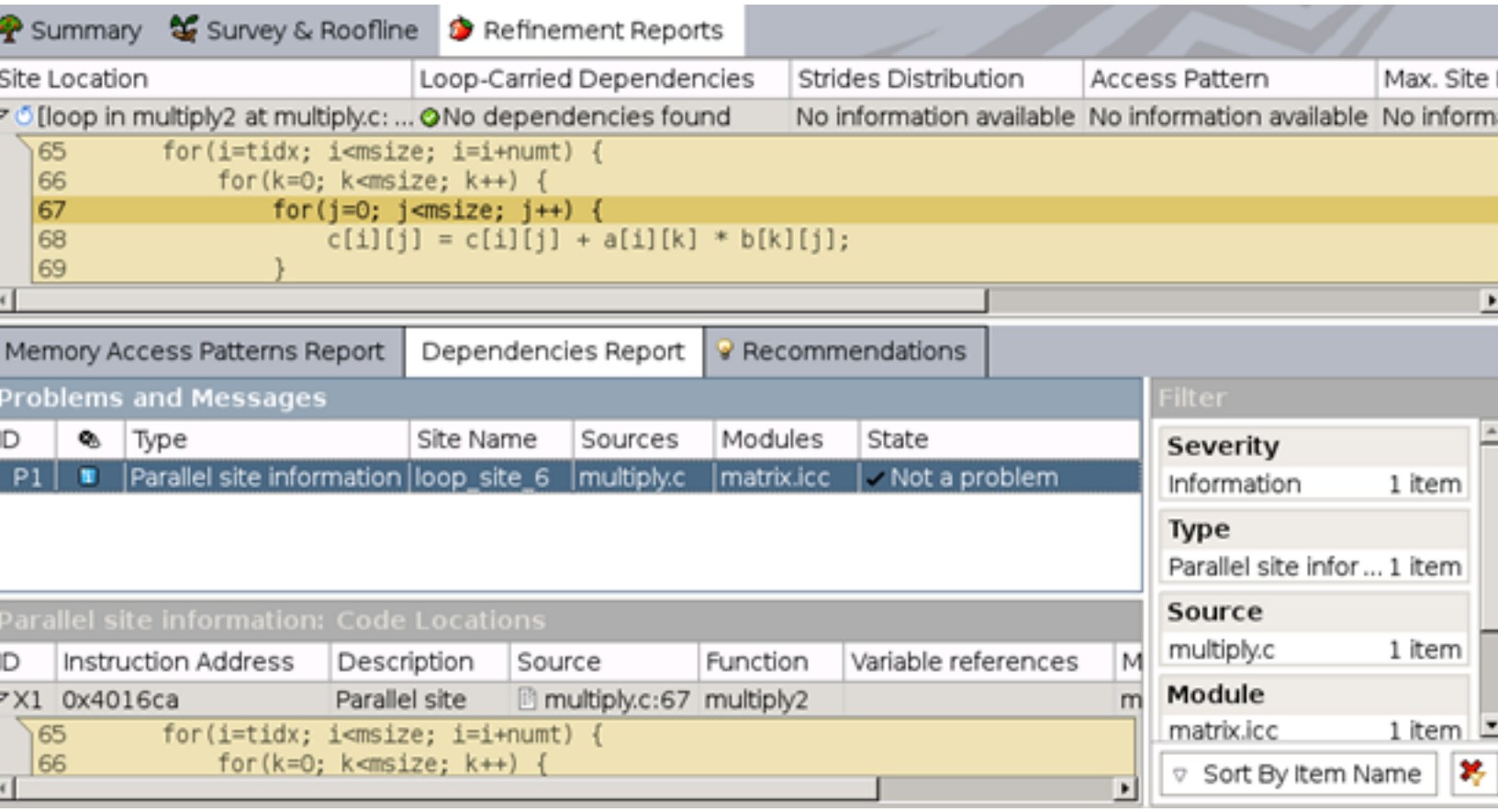
You see that scalar instructions are used. The code is not vectorized.

Explore Options for Vectorization

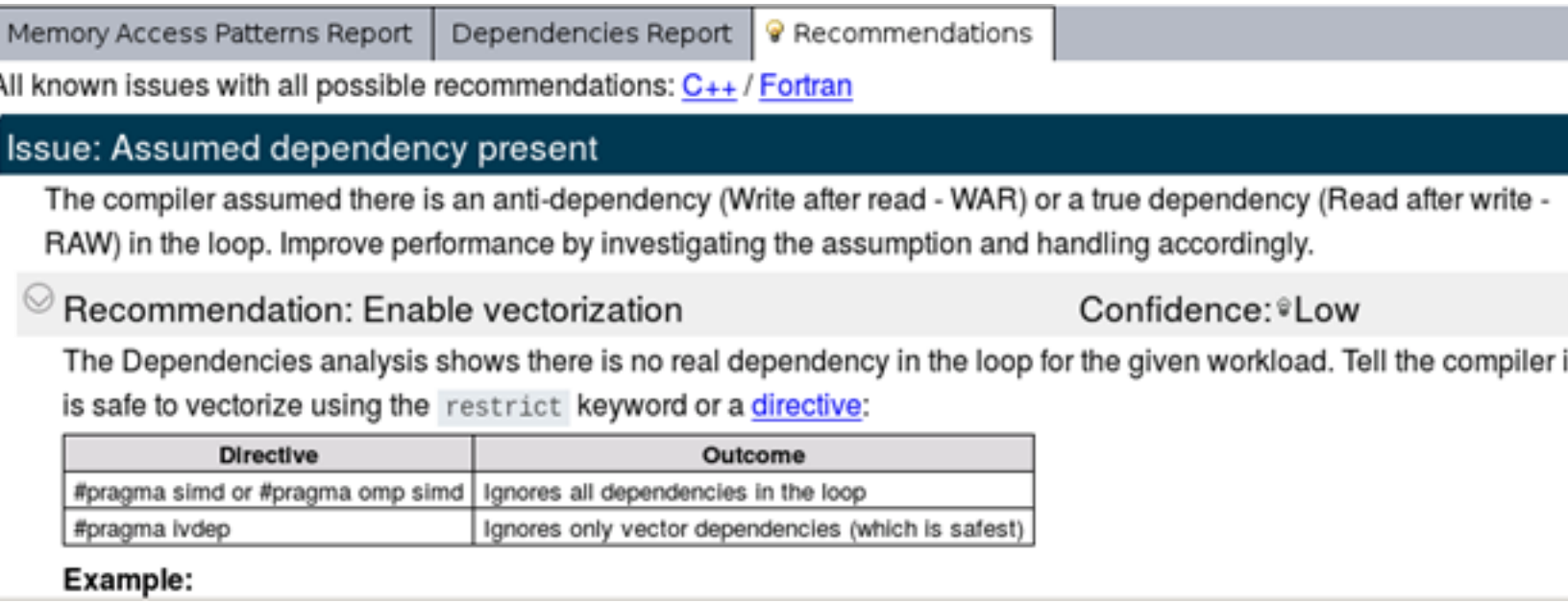
To understand what prevents the code from being vectorized, use the **Vectorization Advisor** tool in Intel® Advisor.



In the graphic, we see that the loop was not vectorized due to assumed dependencies. For further details, mark the loop and run the **Dependencies** analysis in Intel Advisor:



The **Dependencies** report informs us that there are no actual dependencies found. There is a recommendation to use `#pragma` to make the compiler ignore the assumed dependencies:



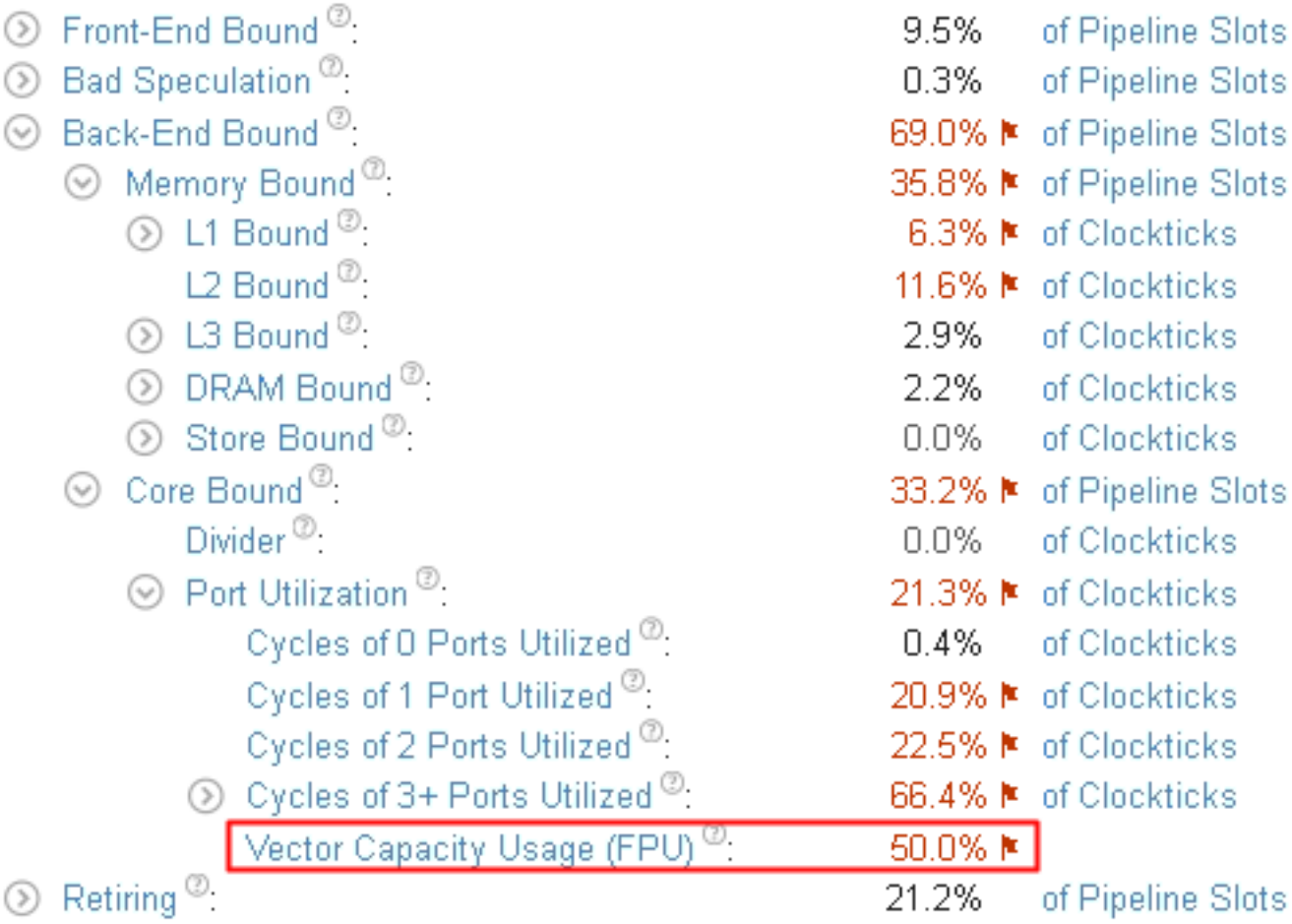
With the `#pragma` added, the `matrix` code looks as follows:

```
void multiply2_vec(inte msize, int tidx, int numt, TYPE a[][NUM],
    TYPE b[][NUM], TYPE c[][NUM], TYPE t[][NUM]
{
    int i,j,k;

    for(i=tidx; i<msize; i=i+numt) {
        for(k=0; k<msize; k++) {
#pragma ivdep
            for(j=0; j<msize; j++) {
                c[i][j] = c[i][j] + a[i][j] * b[i][j];
            }
        }
    }
}
```

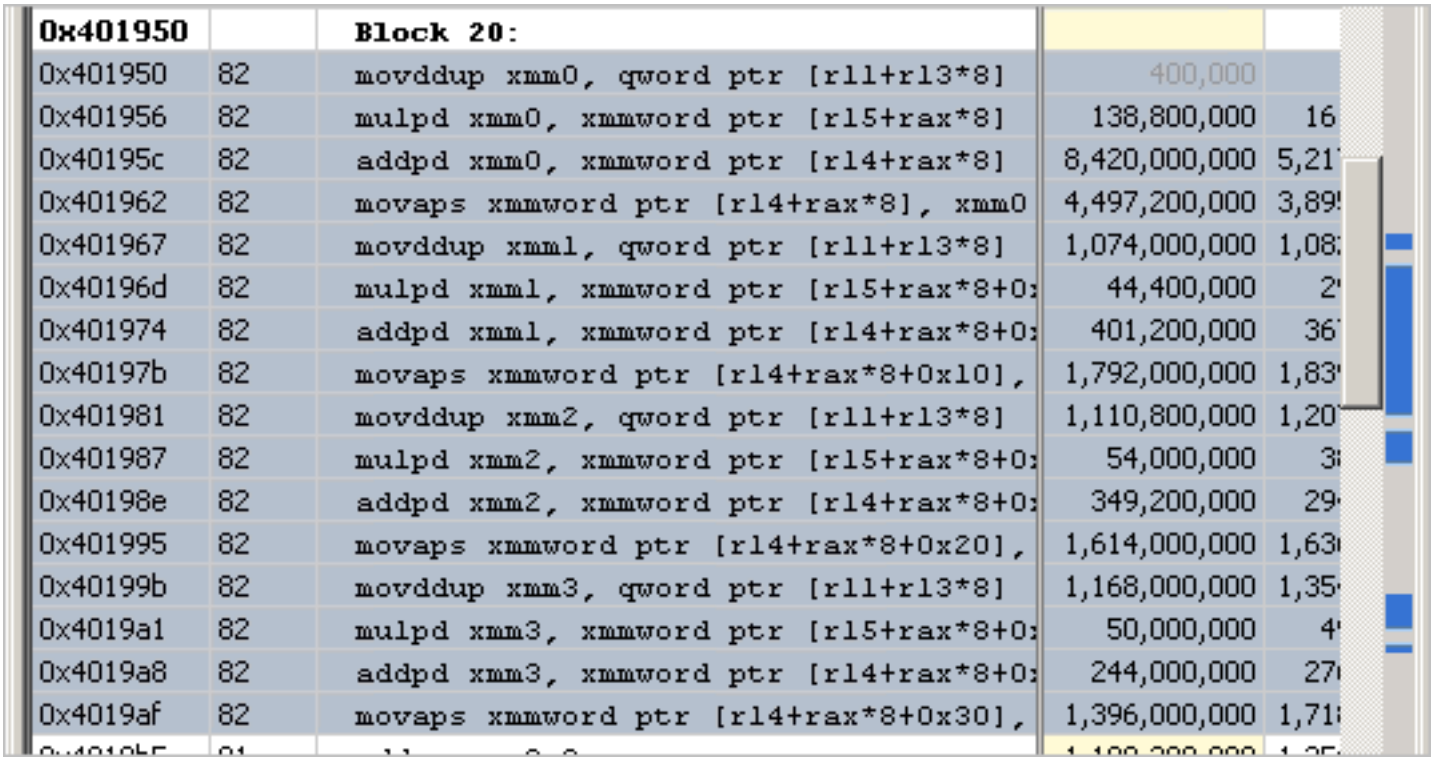
Compiling and running the updated code results in 0.7 second speed-up in the execution time.

Compile with the Latest Instruction Set

Repeat the Microarchitecture Exploration analysis in VTune Profiler on the updated code to see the following result:



The **Vector Capacity Usage** has improved but is still only 50% and has been flagged. Look into the Assembly view once again:



0x401950		Block 20:		
0x401950	82	movddup xmm0, qword ptr [r11+r13*8]	400,000	
0x401956	82	mulpd xmm0, xmmword ptr [r15+rax*8]	138,800,000	16
0x40195c	82	addpd xmm0, xmmword ptr [r14+rax*8]	8,420,000,000	5,21
0x401962	82	movaps xmmword ptr [r14+rax*8], xmm0	4,497,200,000	3,89
0x401967	82	movddup xmm1, qword ptr [r11+r13*8]	1,074,000,000	1,08
0x40196d	82	mulpd xmm1, xmmword ptr [r15+rax*8+0	44,400,000	2
0x401974	82	addpd xmm1, xmmword ptr [r14+rax*8+0	401,200,000	36
0x40197b	82	movaps xmmword ptr [r14+rax*8+0x10],	1,792,000,000	1,83
0x401981	82	movddup xmm2, qword ptr [r11+r13*8]	1,110,800,000	1,20
0x401987	82	mulpd xmm2, xmmword ptr [r15+rax*8+0	54,000,000	3
0x40198e	82	addpd xmm2, xmmword ptr [r14+rax*8+0	349,200,000	29
0x401995	82	movaps xmmword ptr [r14+rax*8+0x20],	1,614,000,000	1,63
0x40199b	82	movddup xmm3, qword ptr [r11+r13*8]	1,168,000,000	1,35
0x4019a1	82	mulpd xmm3, xmmword ptr [r15+rax*8+0	50,000,000	4
0x4019a8	82	addpd xmm3, xmmword ptr [r14+rax*8+0	244,000,000	27
0x4019af	82	movaps xmmword ptr [r14+rax*8+0x30],	1,396,000,000	1,71

Here, we see that the code uses SSE instructions while the CPU in this use case supports the AVX2 instruction set.

To apply it, re-compile the code with the `-xCORE-AVX2` option and run the Microarchitecture Exploration analysis once more.

For the recompiled code, the execution time has dropped to 0.6 seconds. Repeat the Microarchitecture Exploration analysis to verify the optimization. The **Vector Capacity Usage** metric value is now 100%:

Front-End Bound:	8.5%	of Pipeline Slots
Bad Speculation:	0.4%	of Pipeline Slots
Back-End Bound:	73.8%	of Pipeline Slots
Memory Bound:	48.4%	of Pipeline Slots
L1 Bound:	7.1%	of Clockticks
DTLB Overhead:	4.9%	of Clockticks
Loads Blocked by Store Forwarding:	0.0%	of Clockticks
Lock Latency:	0.0%	of Clockticks
Split Loads:	0.0%	of Clockticks
4K Aliasing:	4.7%	of Clockticks
FB Full:	49.9%	of Clockticks
L2 Bound:	10.0%	of Clockticks
L3 Bound:	19.0%	of Clockticks
Contested Accesses:	0.0%	of Clockticks
Data Sharing:	17.1%	of Clockticks
L3 Latency:	45.7%	of Clockticks
SQ Full:	13.6%	of Clockticks
DRAM Bound:	11.0%	of Clockticks
Memory Bandwidth:	50.8%	of Clockticks
Memory Latency:	19.1%	of Clockticks
LLC Miss:	22.4%	of Clockticks
Store Bound:	0.0%	of Clockticks
Core Bound:	25.4%	of Pipeline Slots
Divider:	0.0%	of Clockticks
Port Utilization:	24.7%	of Clockticks
Cycles of 0 Ports Utilized:	0.4%	of Clockticks
Cycles of 1 Port Utilized:	24.3%	of Clockticks
Cycles of 2 Ports Utilized:	17.0%	of Clockticks
Cycles of 3+ Ports Utilized:	27.4%	of Clockticks
Vector Capacity Usage (FPU):	100.0%	
Retiring:	17.3%	of Pipeline Slots

See Also

Microarchitecture Exploration Analysis for Hardware Issues

Top-down Microarchitecture Analysis Method

Page Faults

Identify and measure the impact of page faults on application performance. Use Microarchitecture Exploration, System Overview, and Memory Consumption analyses in Intel® VTune™ Profiler.

Content Expert: Jeffrey Reinemann

A page fault occurs when a running program accesses a memory page that is not currently mapped to the virtual address space of a process. The Memory-Management Unit (MMU) handles mapping. The MMU uses a Translation Lookaside Buffer (TLB) as a cache to reduce the time taken to access a memory location. When a TLB miss occurs, the page may be accessible to the process but not just actually mapped. Alternatively, the page content may need to be loaded from the storage device issuing a page fault exception. While page faults are a common mechanism for handling virtual memory, their impact on the performance of your application can be significant due to a variety of ways to increase the page size.

- INGREDIENTS
- DIRECTIONS:

  1. Identify TLB issues with Microarchitecture Exploration analysis.
  2. Trace kernel activity with System Overview analysis.
  3. Calculate the amount of allocated memory with Memory Consumption analysis.
  4. Reduce page faults with huge pages.

Ingredients

This section lists the hardware and software tools used for the performance analysis scenario.

- **Application**: `matrix` application available in the product directory (`<install-dir>/samples/en/C++`). For this recipe,

  1. Change the size of matrices. In `src/multiply.h`, modify the NUM value from 2048 to 8192.
  2. Rebuild the `matrix` application. Run `make` from the `/linux` directory.
- **Performance analysis tools**: Intel® VTune™ Profiler - Microarchitecture Exploration, System Overview, and Memory Consumption analysis types
- **Operating System**: Ubuntu* 22.04.1 LTS 64-bit

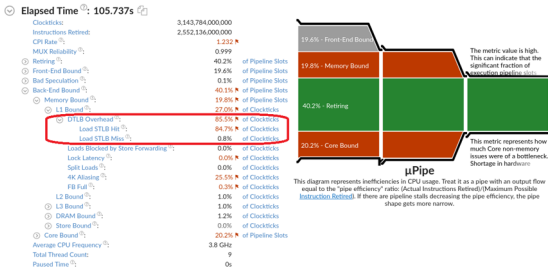
Identify TLB Issues with Microarchitecture Exploration Analysis

Assess the usage of hardware resources by your application. Run the Microarchitecture Exploration analysis:

1. Open Intel® VTune™ Profiler. By default, the `sample (matrix)` project opens as the current project. Make sure this project is configured to launch the `matrix` application with NUM=8192 in `src/multiply.h`. Otherwise, create a new project for the updated application.
2. On the Welcome page, click **Configure Analysis**.
3. In the **HOW** pane, select **Microarchitecture Exploration** from the **Microarchitecture** analysis group.
4. Click the  **Start** button to run the analysis.

When the analysis completes, Intel® VTune™ Profiler finalizes results and opens the **Summary** window with application-level statistics.

Explore the Back-End Bound issues caused by TLB misses:

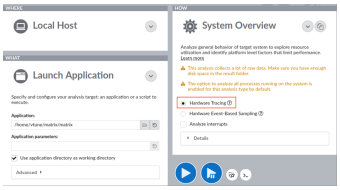


The **DTLB Overhead** metric estimates the performance penalty paid for missing TLB. Most of the overhead is attributed to the **Load STLB Hit** metric, counting first-level (DTLB) misses that hit the second-level TLB (STLB).

There is a small value of the **Load STLB Miss** metric representing a fraction of cycles performing a hardware page walk. Know that these metrics do not account for the overall time spent within page fault exceptions. While the Microarchitecture Exploration analysis helps you diagnose TLB-related issues, you still need to estimate an impact of page fault exceptions on the application elapsed time.

Trace Kernel Activity with System Overview Analysis

A page fault triggers an interrupt caught by the Linux kernel. To measure the exact CPU time spent within the Linux kernel, you need an analysis that is more granular. The **System Overview analysis** in the **Hardware Tracing** mode uses Intel® Processor Trace technology to capture all the retired branch instructions on CPU cores. In particular, this analysis enables accurate tracing of all the kernel activities including interrupts:



Even with the **Launch Application** target configuration, this analysis performs a system-wide data collection.

Due to a significant amount of branch instructions, this analysis collects a lot of raw data. You can run the analysis from the command line and limit the scope of data collection scope to the first 3 seconds. Before you run the analysis from the command-line, make sure to set up environment variables by running this script from the product installation directory: `source env/vars.sh`.

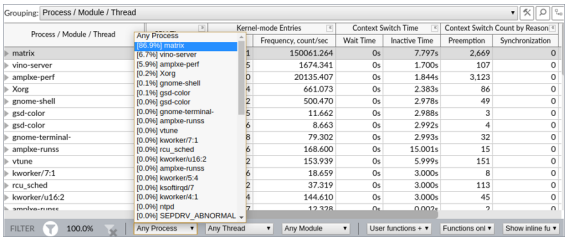
Next, run the analysis:

```
vtune -collect system-overview -knob collecting-mode=hw-tracing -d 3 -r matrix-so ./matrix
```

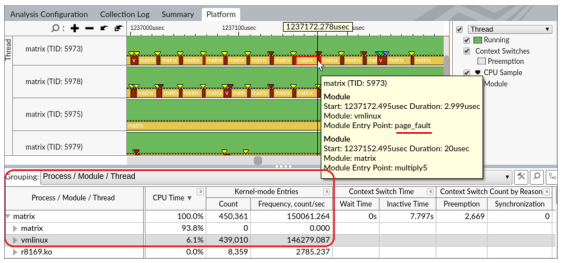
Open the result in the VTune Profiler GUI:

```
vtune-gui ./matrix-so
```

When the result opens, switch to the **Platform** tab and filter the collected data by the `matrix` process using the filter bar drop-down menu:



In the Timeline pane, you can see that most of the CPU time is spent within the `matrix` module executing the `multiply` function. This function is not executed continuously. In a few milliseconds, the `multiply` function is interrupted, and the heaviest interrupts are caused by page faults:



The grid view helps you discover that overall time spent by the sample application within the Linux kernel is 6.1%, where 439K kernel entries occurred just within the first 3 seconds of the application execution. To resolve this, consider using huge pages.

Calculate the Amount of Allocated Memory with Memory Consumption Analysis

To switch to huge pages, define the number of pages you need.

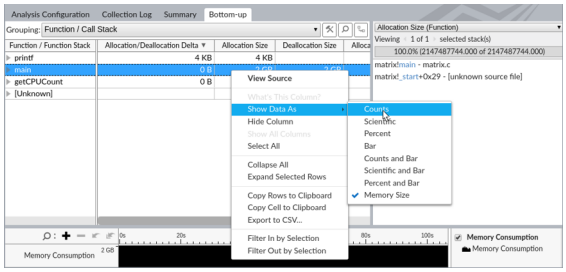
To find this number, calculate the amount of memory allocated by the application. For simple applications like `matrix`, you can inspect the source code. For more complex applications, run the **Memory Consumption** analysis to find the exact allocated memory size or identify objects that should use huge pages.

1. Click **Configure Analysis** to open your `matrix` project configuration.
2. In the **HOW** pane, select **Memory Consumption** from the **Hotspots** analysis group.
3. Change the **Minimal dynamic memory object size to track** option value to 1.
4. Click the

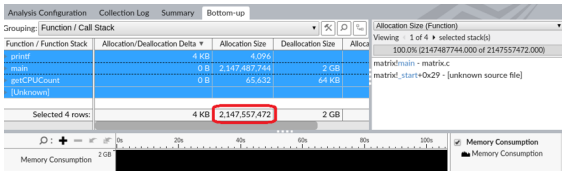
   Start button to run the analysis.

   Once Intel® VTune™ Profiler completes data collection, the results are finalized and displayed in the **Summary** window with application-level statistics.
5. Click the **Bottom-up** tab. In the **Allocation Size** column, right-click and select **Show Data As** > **Counts** for a bytes representation:



6. Right-click the grid again and choose **Select All** (alternatively, press **Ctrl-A**) to see the total allocation size.

   The application allocates 2147557472 bytes:



Reduce Page Faults with Huge Pages

By default, a page size is 4Kb. With huge pages, the default page size is 2Mb and it can be increased up to 1Gb. To switch to huge pages, use `libhugetlbfs`.

First, calculate the number of 2Mb pages you need. The sample `matrix` allocates 2147557472 bytes. This means that you need 2147557472 / 2097152 = 1025 pages of 2Mb (using top rounding).

To switch to huge pages:

1. Configure the number of pages:

```
sudo hugeadm --pool-pages-min 2Mb:1025
```

2. Create a `matrix.sh` script with this content:

```
#!/bin/bash
LD_PRELOAD=libhugetlbfs.so HUGETLB_MORECORE=yes ./matrix
```

3. Set the executable mode for the script:

```
chmod u+x ./matrix.sh
```

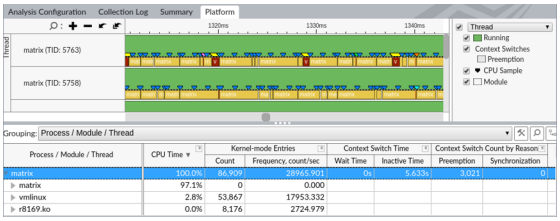
4. Repeat the System Overview analysis.

```
vtune -collect system-overview -knob collecting-mode=hw-tracing -d 3 -r matrix-so-hp ./matrix.sh
```

5. Open the result in the Intel® VTune™ Profiler GUI:

```
vtune-gui ./matrix-so-hp
```

The **Platform** view shows a 3.3% reduction of kernel CPU time and 8.1x reduction on kernel-mode entries:



The elapsed time of the `matrix` application with huge pages is reduced from 106.4s to 100.5s, which is around 5% of an overall elapsed time improvement without requiring any code change.

See Also

Microarchitecture Exploration Analysis for Hardware Issues
System Overview Analysis
Memory Consumption Analysis

Instruction Cache Misses

Profile an application bound on the front-end and reduce ICache misses using the Microarchitecture Exploration analysis with the PGO option.

Content Expert: Jeffrey Reinemann

- INGREDIENTS
- DIRECTIONS:

1. Run Microarchitecture Exploration analysis
2. Identify Hardware Hotspots
3. Compile Your Code Again with Profile Guided Optimization
4. Verify Optimization

Ingredients

This section lists the hardware and software tools used for the performance analysis scenario.

- **Application**: A test sample based on the `sqlite` database. The application is used as a demo and not available for download.
- **Tools**:
  - Intel® VTune™ Profiler version 2018 or newer - Microarchitecture Exploration analysis
  - Intel® DPC++/C++ Compiler
- **CPU**: Intel® microarchitecture code named Skylake

Run Microarchitecture Exploration Analysis

Get an overall assessment of potential performance bottlenecks in the application. Run the **Microarchitecture Exploration analysis**:

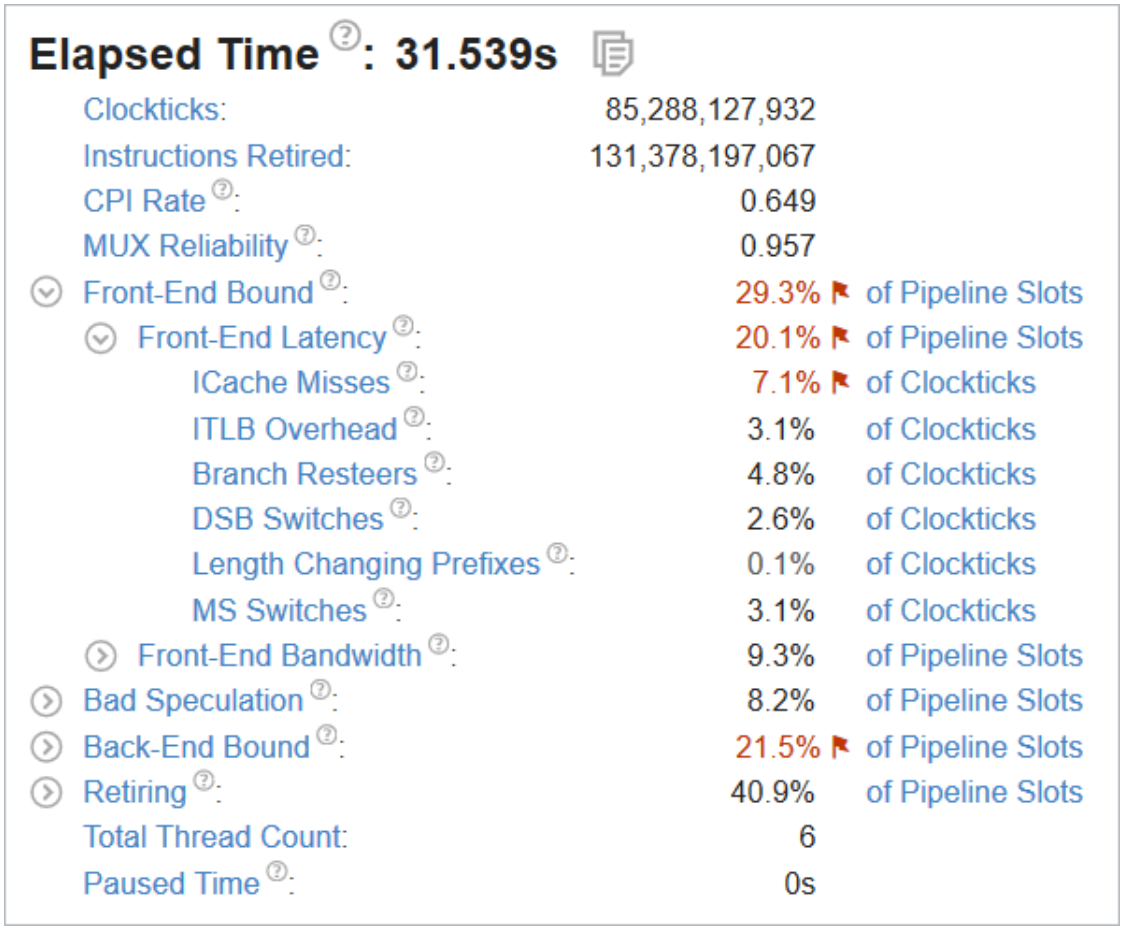
1. In the Intel® VTune™ Profiler UI, click the 

   New Project button on the toolbar. Specify a name for the new project, for example: `sqlite`.
2. In the **Analysis Target** window, select the **local host** target system type for the host-based analysis.
3. Select the **Launch Application** target type and specify an application for analysis.
4. In the Analysis Tree, select **Microarchitecture > Microarchitecture Exploration**.
5. Click **Start**.

Intel® VTune™ Profiler launches the application and collects data. When the collection completes, Intel® VTune™ Profiler finalizes the result and resolves symbol information. This is necessary for proper source analysis.

Identify Hardware Hotspots

The Microarchitecture Exploration analysis helps you identify dominant performance bottlenecks in your code. Start your analysis with the Summary view. Here, you see application-level statistics for each hardware metric. Focus on the performance issues that have been flagged:

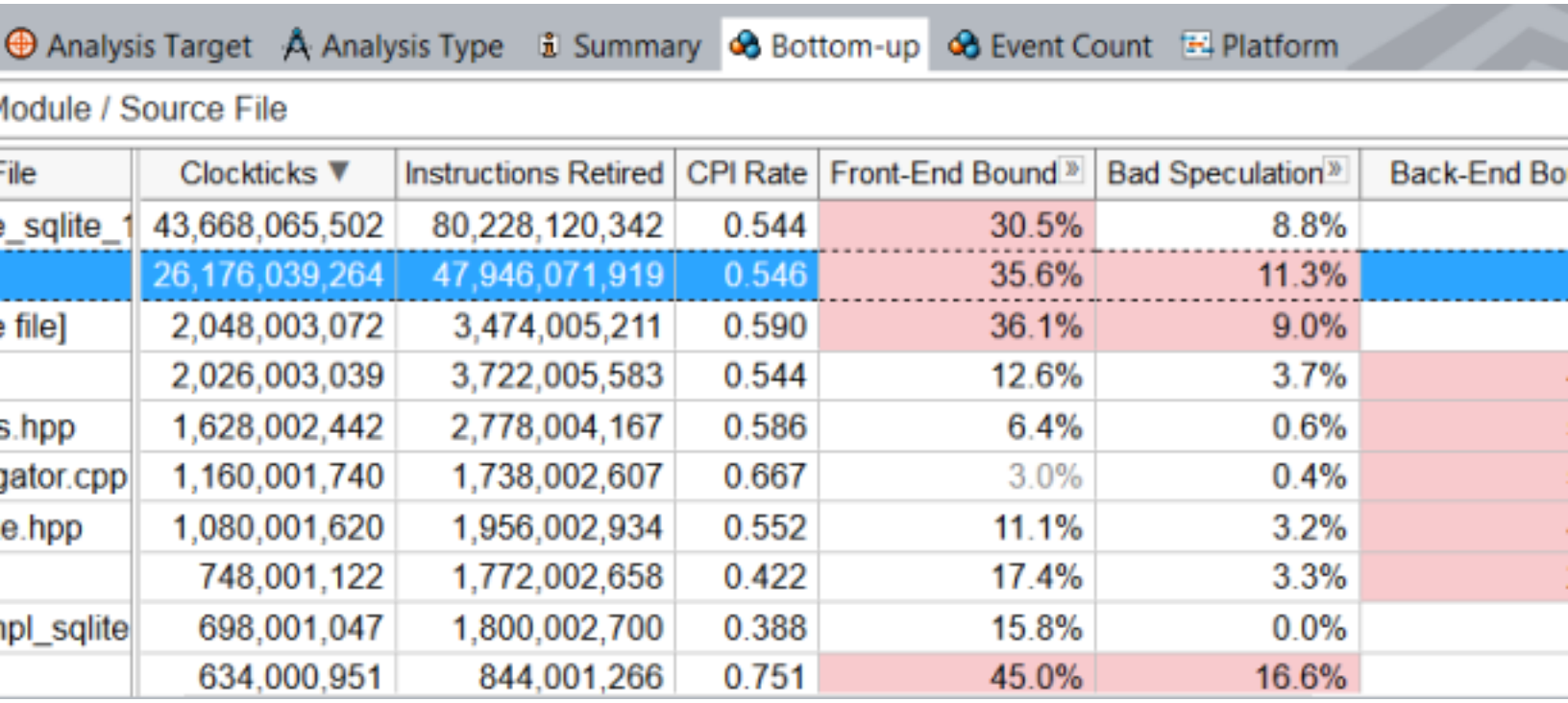


In this example, the sample application is front-end bound (29.3% of Pipeline Slots) with the instruction cache misses as a dominant bottleneck (7.1% of Clockticks).

Next, locate the issue in the code by switching to the **Bottom-up** window. Click the



Customize Grouping button, next to the **Grouping** toolbar. Create a new custom grouping called **Module/Source File**:

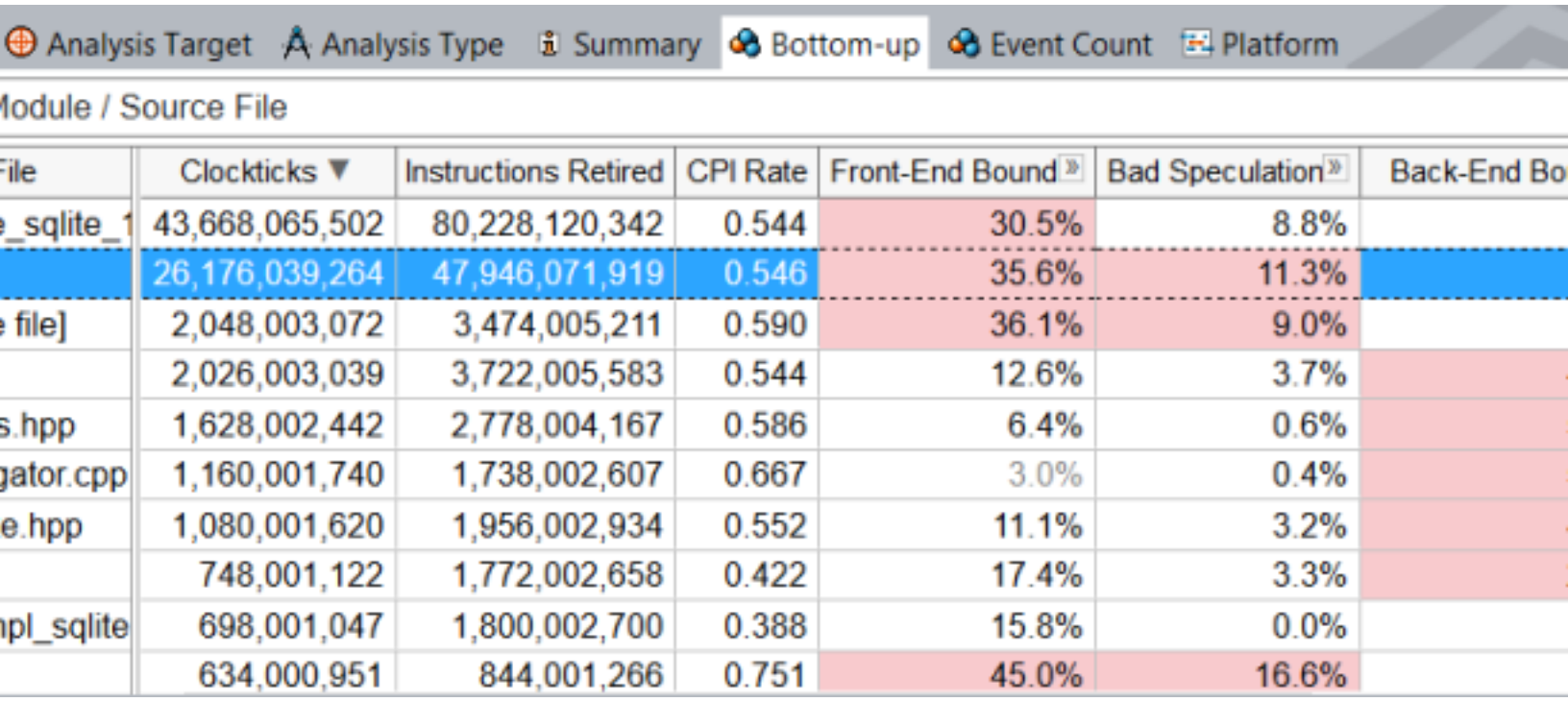


Analysis Target | Analysis Type | Summary | Bottom-up | Event Count | Platform

Module / Source File

File	Clockticks ▼	Instructions Retired	CPI Rate	Front-End Bound	Bad Speculation	Back-End Bo
e_sqlite_1	43,668,065,502	80,228,120,342	0.544	30.5%	8.8%	
	26,176,039,264	47,946,071,919	0.546	35.6%	11.3%	
e file]	2,048,003,072	3,474,005,211	0.590	36.1%	9.0%	
	2,026,003,039	3,722,005,583	0.544	12.6%	3.7%	
s.hpp	1,628,002,442	2,778,004,167	0.586	6.4%	0.6%	
gator.cpp	1,160,001,740	1,738,002,607	0.667	3.0%	0.4%	
e.hpp	1,080,001,620	1,956,002,934	0.552	11.1%	3.2%	
	748,001,122	1,772,002,658	0.422	17.4%	3.3%	
npl_sqlite	698,001,047	1,800,002,700	0.388	15.8%	0.0%	
	634,000,951	844,001,266	0.751	45.0%	16.6%	

When we apply the new grouping to the collected results, we see that `sqlite3.c` file is the main hotspot which takes the most CPU cycles to execute:



Analysis Target | Analysis Type | Summary | Bottom-up | Event Count | Platform

Module / Source File

File	Clockticks ▼	Instructions Retired	CPI Rate	Front-End Bound	Bad Speculation	Back-End Bo
e_sqlite_1	43,668,065,502	80,228,120,342	0.544	30.5%	8.8%	
	26,176,039,264	47,946,071,919	0.546	35.6%	11.3%	
e file]	2,048,003,072	3,474,005,211	0.590	36.1%	9.0%	
	2,026,003,039	3,722,005,583	0.544	12.6%	3.7%	
s.hpp	1,628,002,442	2,778,004,167	0.586	6.4%	0.6%	
gator.cpp	1,160,001,740	1,738,002,607	0.667	3.0%	0.4%	
e.hpp	1,080,001,620	1,956,002,934	0.552	11.1%	3.2%	
	748,001,122	1,772,002,658	0.422	17.4%	3.3%	
npl_sqlite	698,001,047	1,800,002,700	0.388	15.8%	0.0%	
	634,000,951	844,001,266	0.751	45.0%	16.6%	

The **ICache Misses** metric displays the highest value for the `sqlite3.c` file:

File	Front-End Bound					
	Front-End Latency					
	ICache Misses	ITLB Overhead	Branch Resteers	DSB Switches	Length Changing...	MS Switches
ce_sqlite	7.3%	2.8%	4.2%	3.3%	0.0%	2.0%
	9.3%	3.2%	4.0%	3.4%	0.0%	2.3%
ce file]	9.8%	2.1%	5.9%	2.9%	0.0%	0.0%
	5.9%	0.8%	3.0%	1.0%	0.0%	2.0%
es.hpp	2.5%	1.8%	1.2%	2.5%	0.0%	2.5%
egator.cp	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
ite.hpp	0.0%	1.9%	1.9%	1.9%	0.0%	0.0%

Compile Your Code Again with Profile Guided Optimization

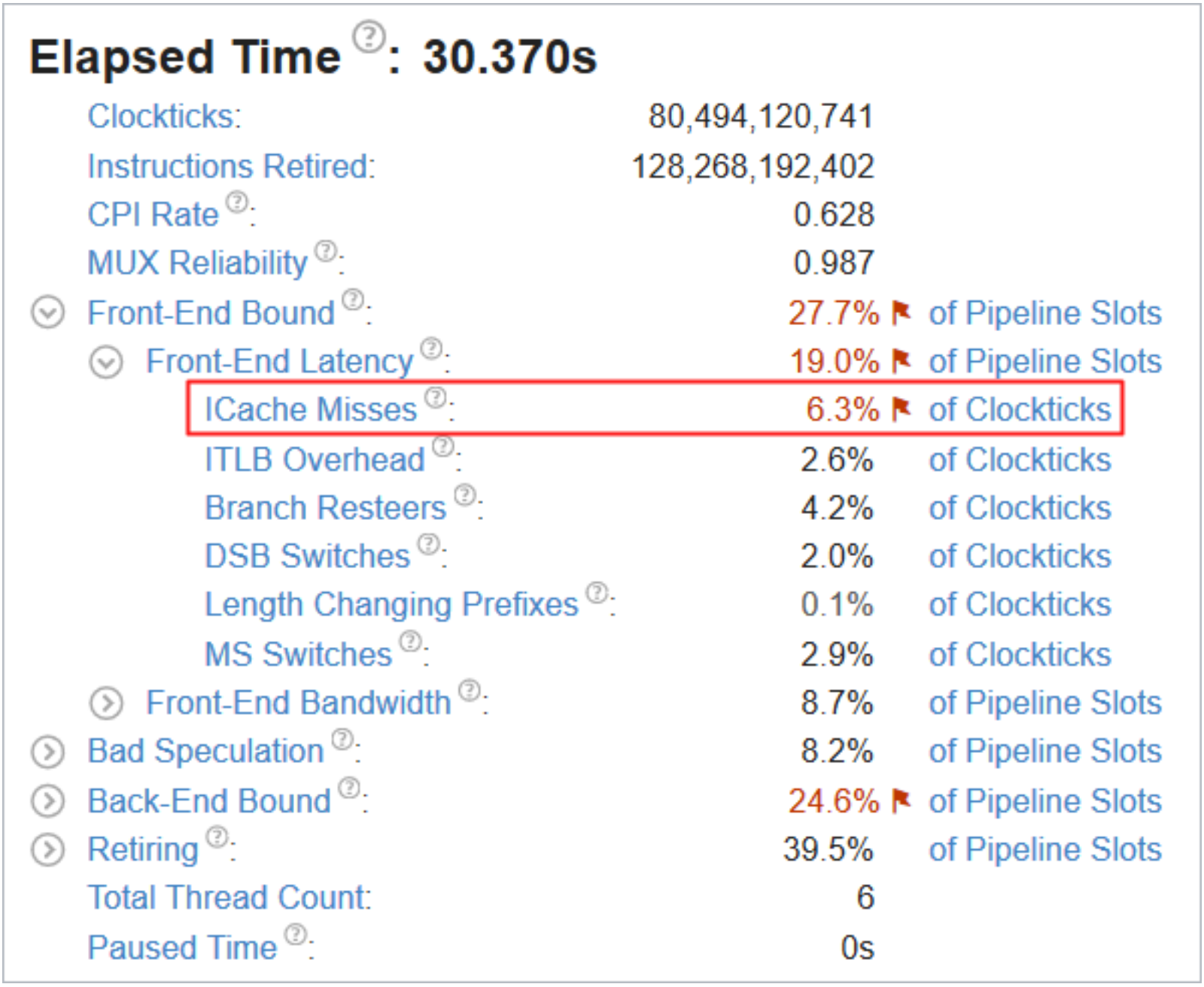
Use the Intel® DPC++/C++ Compiler to apply Profile Guided Optimization (PGO) to the `sqlite` library:

1. Compile your code once again with the `/Qprof-gen` option.
2. Run the benchmark.
3. Again, compile your code with the `/Qprof-use` option.

For more information on PGO, see the Profile-Guided Optimizations overview.

Verify Optimization

Repeat the Microarchitecture Exploration analysis on the optimized code. The new result shows 30.3 seconds of Elapsed time, which is almost 4% better than the original 31.5 seconds:



The number of clockticks stalled due to ICache Misses for the `sqlite` library has also reduced to 6.4% from 9.3%:

File	Front-End Bound					
	Front-End Latency					
	ICache Misses	ITLB Overhead	Branch Resteers	DSB Switches	Length Changing...	MS Switches
e_sqlite	5.5%	1.9%	4.0%	2.6%	0.0%	0.9%
	6.4%	2.1%	4.9%	2.7%	0.0%	0.9%
	3.0%	1.5%	4.0%	4.0%	0.0%	0.0%
e file]	5.7%	2.3%	2.3%	1.1%	0.0%	0.0%
es.hpp	1.2%	0.5%	1.2%	1.2%	0.0%	2.5%
gator.cp	0.0%	0.2%	0.0%	0.0%	0.0%	0.0%
ite.hpp	5.2%	1.0%	1.7%	0.0%	0.0%	0.0%

See Also

Microarchitecture Exploration Analysis for Hardware Issues

Top-down Microarchitecture Analysis Method

OS Thread Migration

Identify OS thread migration on the NUMA architecture with the Hotspots analysis in Intel® VTune™ Profiler.

Complex operating systems use a scheduler to assign application threads to processor cores. These threads are called software threads. The scheduler may choose the placement of the application threads on the physical cores depending on a number of different factors such as system state or system policies.

A software thread can execute on a core for some period of time before it gets swapped out to wait. Several reasons can cause a software thread to wait. Getting blocked for I/O is one factor. If available, another software thread may be given a chance to execute on this core. When the original software thread is available to execute once again, the scheduler may migrate the thread over to another core to ensure timely execution.

This poses a problem to newer computing architectures as this software thread migration disassociates the thread from data that has already been fetched into the caches, resulting in longer data access latencies. This problem is further amplified in Non-Uniform Memory Access (NUMA) architectures, where each processor has its own local memory module that it can access directly with a distinct performance advantage. In a NUMA architecture, when a software thread is migrated to another core, the data stored in the local memory of the first core becomes remote and memory access times increase significantly. Hence, thread migration can hurt performance.

Follow this recipe to see if thread migration occurs in your application.

Content Expert: Jeffrey Reinemann

1. INGREDIENTS
2. DIRECTIONS:
   a. Run Hotspots Analysis with Hardware Event-Based Sampling.
   b. Identify thread migration.
   c. Correct thread migration.

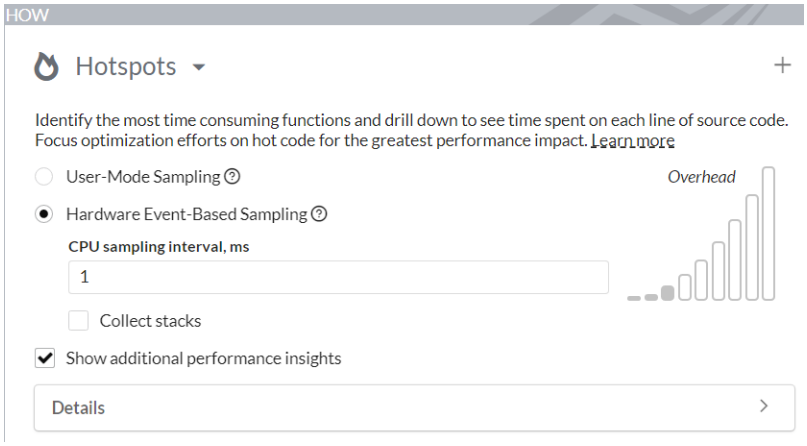
Ingredients

This section lists the hardware and software tools used for the performance analysis scenario.

- **Application:** Sample OpenMP* application. The application is used as a demo and not available for download.
- **Performance analysis tools:** Intel® VTune™ Profiler version 2018 or newer - Hotspots analysis
- **Operating system:** Linux*, Ubuntu* 22.04 64-bit
- **CPU:** Intel® Core™ i7-6700K processor

Run Hotspots Analysis with Hardware Event-Based Sampling

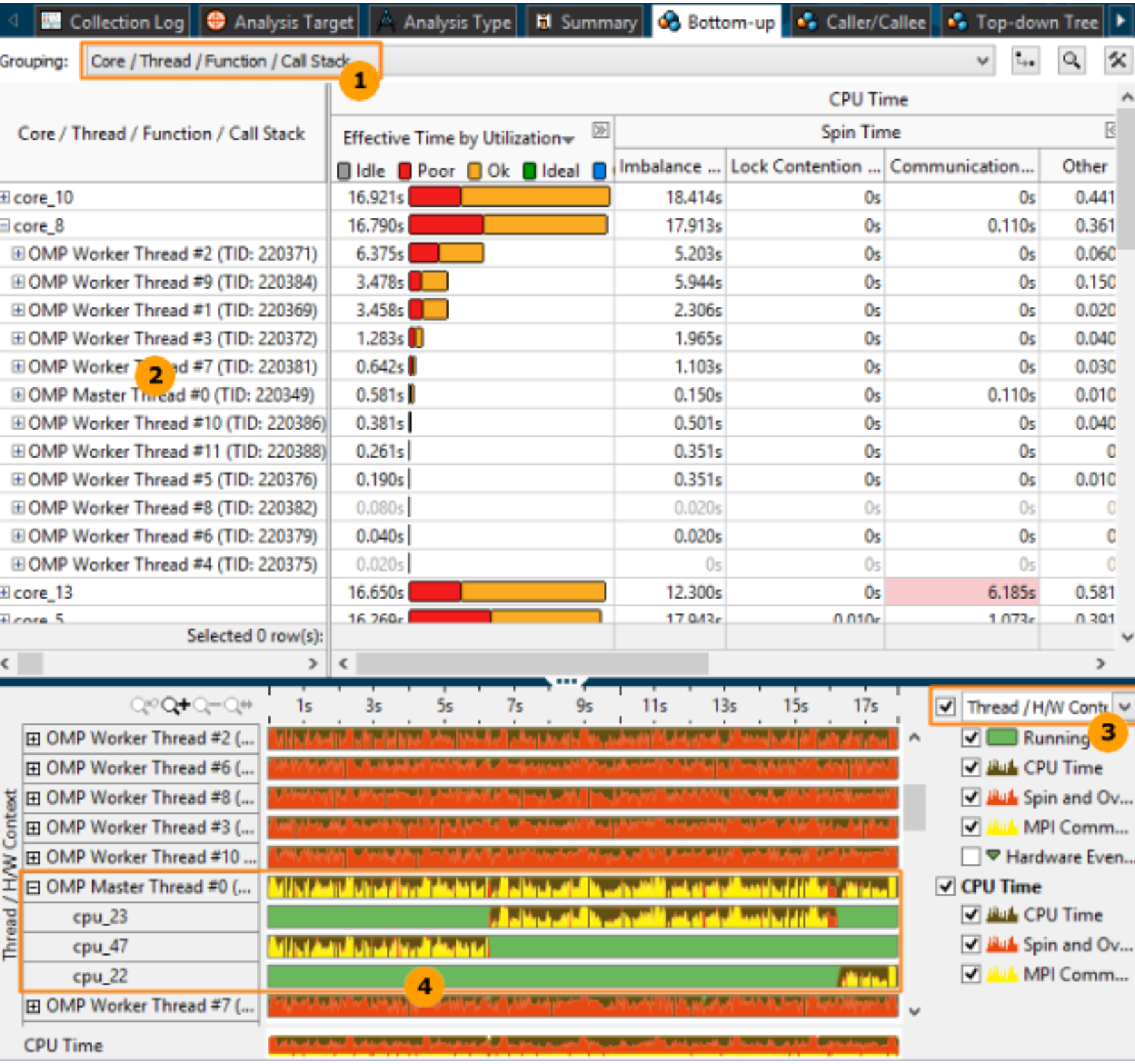
1. In the Intel® VTune™ Profiler UI, select **Hotspots Analysis** from the Analysis Tree.
2. Configure the analysis. Select a sample application.
3. Select **Hardware Event-Based Sampling** mode with a **CPU sampling interval** of 1 ms.
4. Run the analysis.



Identify Thread Migration

Once the analysis completes, the **Summary** window opens with a list of top hotspots in your application.

Examine this list and then switch to the **Bottom-up** window.



Follow these steps:

1. Select the **Core/Thread/Function/Call Stack** grouping.

2. Expand core nodes to see the number of software threads. The number of software threads should be less than or equal to the total number of hardware threads which are supported by the CPU. Also, the software threads should be equally distributed across the cores. If you see a higher count of software threads under any core in your result, there is a thread migration occurring in your application. In this example, there are 12 OpenMP* worker threads in place of 2 threads. This example uses an Intel® Xeon® processor which supports Intel® Hyper-Threading Technology. In core_8, we see that thread migration is happening.

3. Next, analyze thread migration in the Timeline pane. Select the **Thread/Logical Core** grouping.

4. Expand the thread nodes to see the number of CPUs where this thread was executed. Analyze thread execution over time. In this example, OpenMP thread #0 was executing on cpu_23 and then migrated to cpu_47.

To run this analysis from the command line, type:

```
vtune -group-by thread,cpuid -report hotspots -r  /temp/test/omp -s "Logical Core" -q | less
```

```
Thread                   Logical Core  CPU Time:Self
------------------------------  -----------  -------------
OMP Worker Thread #5 (0x3d86)    cpu_0                0.004
matmul-intel64 (0x3d52)          cpu_1                0.013
OMP Worker Thread #15 (0x3d90)   cpu_10               2.418
matmul-intel64 (0x3d52)          cpu_10               2.023
OMP Worker Thread #8 (0x3d89)    cpu_10               0.687
OMP Worker Thread #13 (0x3d8e)   cpu_10               0.097
OMP Worker Thread #6 (0x3d87)    cpu_10               0.065
OMP Worker Thread #4 (0x3d85)    cpu_10               0.059
OMP Worker Thread #1 (0x3d82)    cpu_10               0.048
OMP Worker Thread #9 (0x3d8a)    cpu_10               0.034
OMP Worker Thread #11 (0x3d8c)   cpu_10               0.009
```

Similarly, you can notice the large number of OpenMP worker threads running on cpu_10.

Correct Thread Migration

You can correct the effects of thread migration by setting the thread affinity. Thread affinity refers to the action of restricting the execution of certain threads to a subset of the physical processing units in a multiprocessor computer.

To set thread affinity for your OpenMP application, use the Intel® runtime library which can bind OpenMP threads to physical processing units. You can also use one of these environment variables:

- `OMP_PROC_BIND`
- `OMP_PLACES`
- Intel runtime specific `KMP_AFFINITY`

See Also

OpenMP* Code Analysis

OpenMP* Imbalance and Scheduling Overhead

Follow this recipe to detect and fix frequent parallel bottlenecks of OpenMP programs, such as imbalance on barriers and scheduling overhead.

A barrier is a synchronization point when execution is allowed after all threads in the thread team have arrived. If the execution work is irregular and the chunks of work are equally and statically distributed by worker threads, then the threads that already arrived at the barrier have to now wait for the remaining threads. This is wasted time. When the aggregated wait time on a barrier is normalized by the number of threads in the team, you get the elapsed time that the application can reduce, if the imbalance is eliminated.

One way to eliminate the imbalance on a barrier is to use dynamic scheduling to have chunks of work distributed dynamically between threads. However, following this method with fine-grain chunks can worsen the situation due to scheduling overhead. This recipe describes how you can address OpenMP load imbalance and scheduling overhead problems.

Content Expert: Rupak Roy

- INGREDIENTS
- DIRECTIONS:

  1. Create a baseline
  2. Run HPC Performance Characterization analysis
  3. Identify OpenMP imbalance
  4. Apply dynamic scheduling
  5. Identify OpenMP scheduling overhead
  6. Apply dynamic scheduling with a chunk parameter

Ingredients

This section lists the hardware and software tools you may need for this recipe.

- **Application:** The sample application used in this recipe calculates prime numbers in a particular range. The main loop is parallelized with the OpenMP parallel `for` construct.
- **Compiler**: Intel® oneAPI DPC++/C++ Compiler version 2023.1 or newer. An appropriate version of the Intel Compiler is necessary to have instrumentation inside the Intel OpenMP runtime library, which Intel® VTune™ Profiler uses for analysis. The `parallel-source-info=2` compiler option provides source file information in the OpenMP region names, which helps to identify them.
- **Performance analysis tools**:
  - VTune Profiler version 2023.1 or newer: HPC Performance Characterization analysis
- **Operating system:** Linux*, CentOS Stream release 8
- **CPU:** Intel® Xeon® Gold 6148 CPU @ 2.40GHz

Create a Baseline

The initial version of the sample code uses the OpenMP parallel `for` pragma for the loop by numbers, with the default scheduling that implies static (line 21):

```
#include <stdio.h>
#include <omp.h>

#define NUM 100000000

int isprime( int x )
{
    for( int y = 2; y * y <= x; y++ )
    {
        if( x % y == 0 )
            return 0;
    }

    return 1;
}
```

```
int main( )
{
    int sum = 0;

#pragma omp parallel for reduction (+:sum)
    for( int i = 2; i <= NUM ; i++ )
    {
        sum += isprime ( i );
    }

    printf( "Number of primes numbers: %d", sum );

    return 0;
}
```

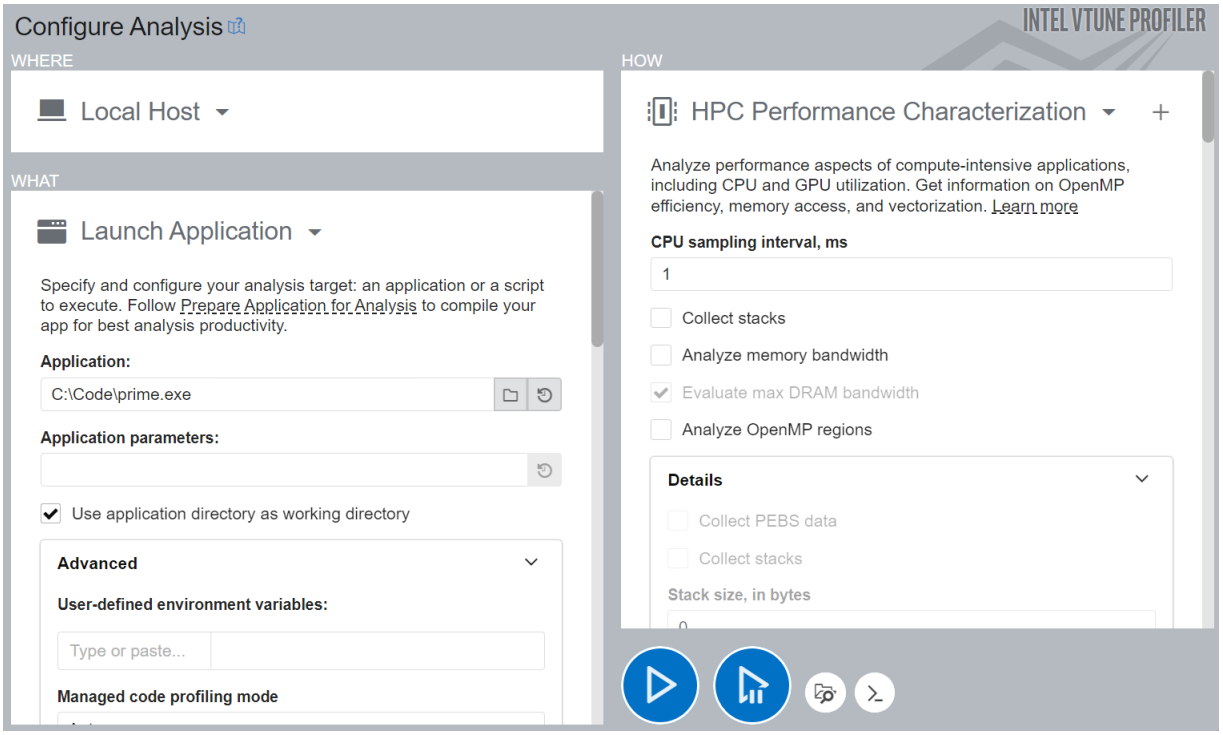
Running the compiled application in this case takes about 3.9 seconds. This is a performance baseline that we will use for further optimizations.

Run HPC Performance Characterization Analysis

To get a high-level understanding of potential performance bottlenecks for the sample, start with the **HPC Performance Characterization analysis**:

1. Click the  **New Project** button on the toolbar and specify a name for the new project, for example: `primes`.
2. Click **Create Project**.

   The **Configure Analysis** window opens.
3. On the **WHERE** pane, select the **Local Host** target system type.
4. On the **WHAT** pane, select the **Launch Application** target type and specify an application for analysis. For example:



5. In the **HOW** pane, in the Analysis Tree, select **HPC Performance Characterization** in the **Parallelism** group.
6. Click the

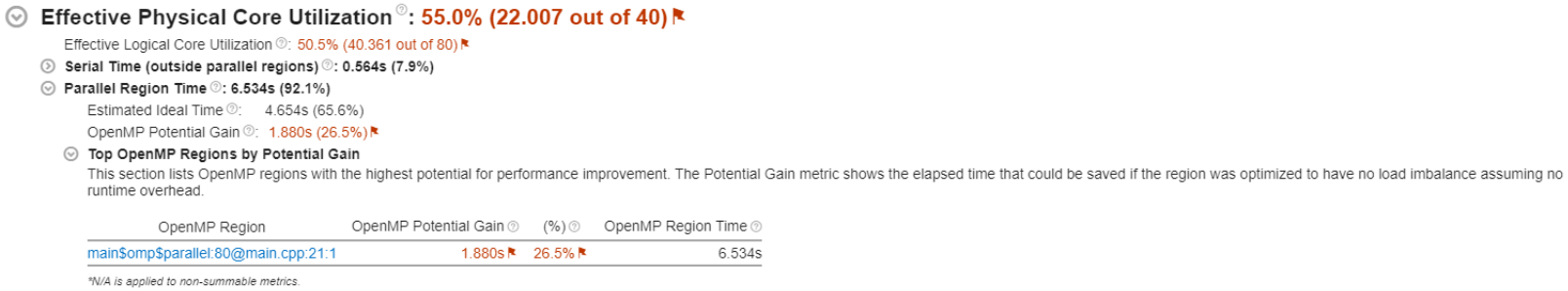
   Start button.

VTune Profiler runs the application, collects data, and finalizes the data collection result (resolving symbol information which is required for successful source analysis).

Identify OpenMP Imbalance

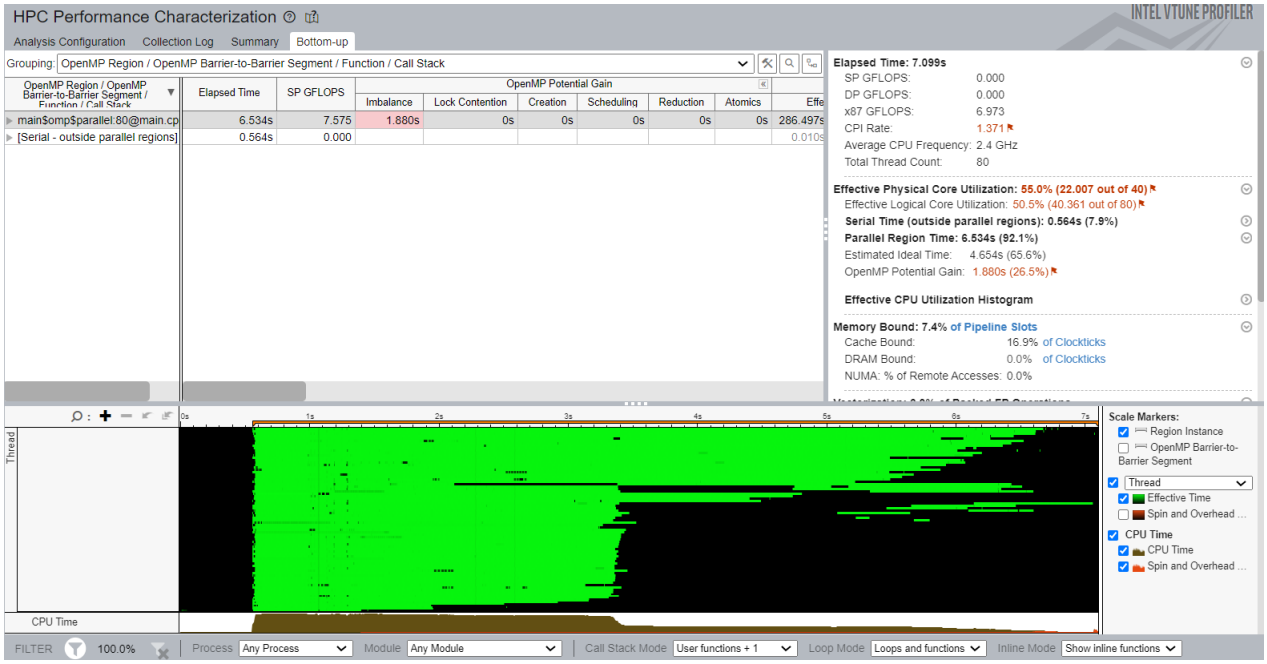
When the **HPC Performance Characterization analysis** completes, the **Summary** window shows important HPC metrics that help understand performance bottlenecks like CPU utilization (parallelism), memory access efficiency, and vectorization. For applications that run with the Intel OpenMP runtime, you can benefit from special OpenMP efficiency metrics that help identify issues with threading parallelism.

Start your analysis by reviewing application-level statistics in the VTune Profiler. If the **Effective Physical Core Utilization** metric (on some systems just **CPU Utilization**) is flagged, a performance problem exists that you should investigate.

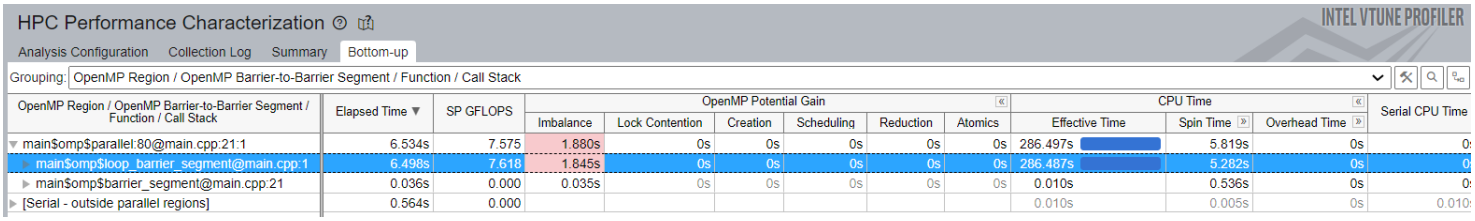


In the **Parallel Region Time** section, see the **OpenMP Potential Gain** metric. Use this to estimate the maximum potential gain that you can get by eliminating parallel inefficiencies. In the sample application, the potential gain is 1.880s (equal to 26.5% of the application runtime). This metric is flagged in the example, so it may be worthwhile to explore the breakdown by parallel constructs.

In the sample application, there is one parallel construct provided in the **Top OpenMP Regions** section. Click the region name in the table to explore more details in the Bottom-up view. To see the breakdown by inefficiencies, expand the **OpenMP Potential Gain** column in the Bottom-up grid. This data helps you understand why the processor time was spent on the OpenMP runtime rather than the sample application. You can also understand how it impacts the elapsed time:



The hot region in the grid row has a value highlighted for the **Imbalance** metric. Hover your mouse over this value to see a recommendatiion to try dynamic scheduling to eliminate the imbalance. If you have more than one barrier in a region, you must expand the region node by barrier-to-barrier segments and identify a performance-critical barrier:



In this sample, there is a loop barrier with critical imbalance and a parallel region join barrier that is not classified by VTune Profiler as a bottleneck.

NOTE
To better visualize the imbalance during the application run, see the Timeline view. The green sections indicate useful work while the black sections show code segments where time was wasted.

Apply Dynamic Scheduling

The imbalance could be caused by static work distribution and assigning large numbers to particular threads while some of the threads processed their chunks with small numbers quickly and wasted the time on the barrier. To eliminate this imbalance, apply dynamic scheduling with the default parameters:

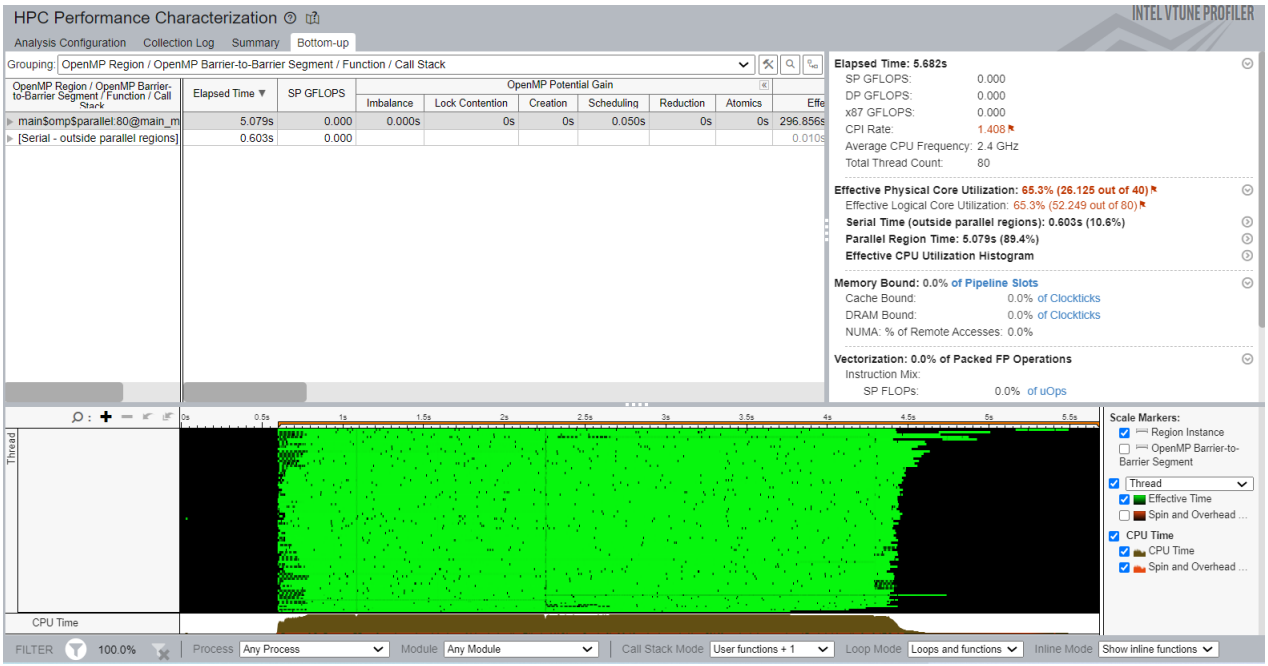
```
    int sum = 0;

#pragma omp parallel for schedule(dynamic) reduction (+:sum)
    for( int i = 2; i <= NUM ; i++ )
    {
        sum += isprime ( i );
    }
```

Recompile the application and compare the execution time versus the original performance baseline to verify your optimization.

Identify OpenMP Scheduling Overhead

Running the modified sample application does not result in any speedup but we can see that the execution time has increased to 5.3 seconds. This is a possible side effect of applying dynamic scheduling with fine-grain chunks of work. To get more insights on possible bottlenecks, repeat the HPC Performance Characterization analysis on the modified code to see the reasons for performance degradation.

After repeating the analysis, open the **Bottom-up** view:



We can observe that the **Imbalance Overhead** has come down to zero. However, the **Scheduling** overhead has jumped to 0.05s. This is caused by the default behavior of the scheduler that assigns one loop iteration per worker thread. Dynamic Scheduling has additional scheduling overhead because threads turn back to the scheduler very frequently during runtime, thus creating a bottleneck. If the scheduling time is significant, try using chunks of coarse-grain work.

Apply Dynamic Scheduling with a Chunk Parameter

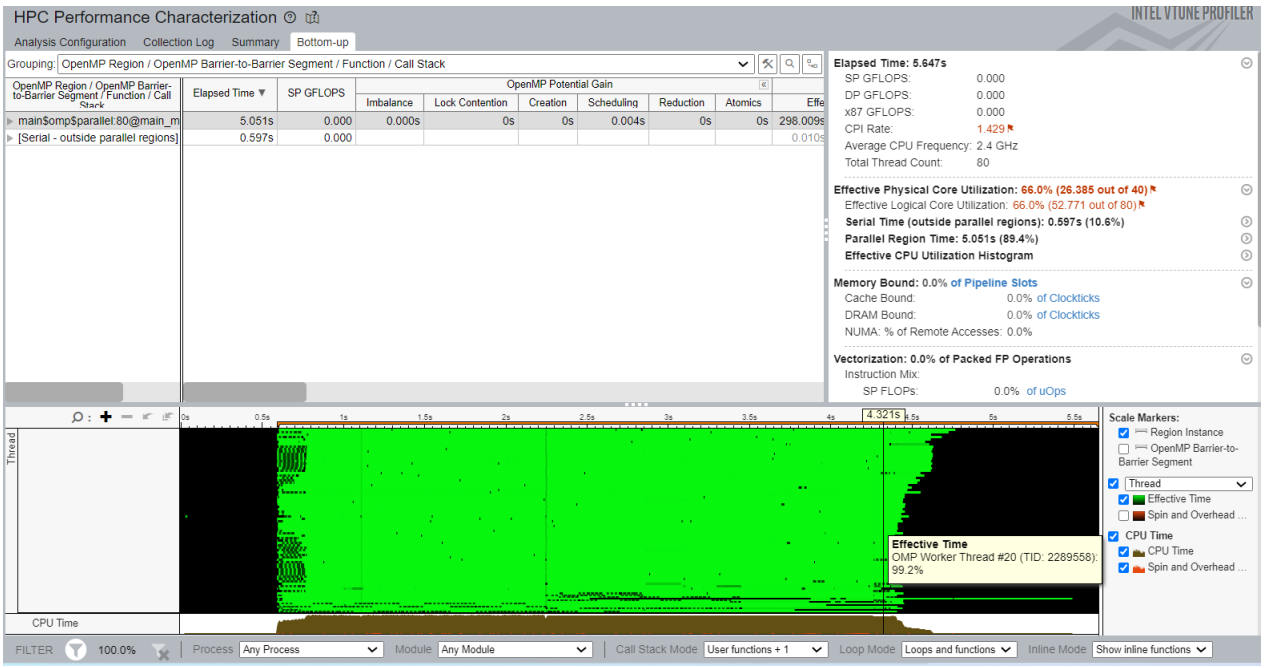
Use the chunk parameter 20 for the `schedule` clause as follows:

```
#pragma omp parallel for schedule(dynamic,20) reduction (+:sum)
    for( int i = 2; i <= NUM ; i++ )
    {
        sum += isprime ( i );
    }
```

After compiling the application again, the elapsed runtime was 5.647 seconds. The physical core utilization improved from 55% to 66%. The Summary view shows the **Parallel Region Time** as 89.4%:



The Bottom-up view shows a good density of useful work (highlighted in green) on the timeline:



Dynamic scheduling may cause code execution that is not cache-friendly. This is because frequent assignments of new chunks of work to a thread can prevent the thread from reusing the cache effectively. So, a well-balanced optimized application with effective CPU utilization can run slower than an imbalanced one with static scheduling. If you observe this behavior, see the **Memory Bound** section of the HPC Performance Characterization view for more information.

NOTE
You can discuss this recipe in the Analyzers developer forum.

See Also

OpenMP* Code Analysis Method

HPC Performance Characterization Analysis
Potential Gain
Tutorial: Analyzing an OpenMP* and MPI Application
Processor Cores Underutilization: OpenMP* Serial Time

Processor Cores Underutilization: OpenMP* Serial Time

Use this recipe to identify a fraction of serial execution in an application that was parallelized with OpenMP. Discover additional opportunities for parallelization, and improve the scalability of the application.

Content Expert: Rupak Roy

The presence of a fraction of serial time in a parallel application can limit the scalability of the application. Scalability is the capacity of the application to fully utilize available hardware resources like cores for code execution.

According to Amdahl's law, the maximum speed-up for a parallel application is given by:

$$\frac{1}{((1-P)+\left(\frac{P}{N}\right))}$$

where:

- P is a parallel portion of the application execution
- N is a number of processor elements

If the serial part (1-P) of the application execution is greater, this decreases the possibility of a linear speed-up. The serial portion limits the performance scalability.

When your application is parallelized with OpenMP, the sequential code execution may be a result of one of these code executions:

- The code executed out of the OpenMP regions.
- The code executed inside the `#pragma omp master` or `#pragma omp single` constructs.

This recipe focuses on the code executed out of OpenMP regions. In this recipe, use Intel® VTune™ Profiler to:

- Detect the serial time of the code executed outside of OpenMP regions
- Analyze the distribution of serial hotspot functions/loops
- Understand opportunities for code parallelization

- INGREDIENTS
- DIRECTIONS:

1. Create a baseline.
2. Run HPC Performance Characterization analysis.
3. Identify OpenMP Serial Time.
4. Parallelize the code.
5. Inspect threading errors.

Ingredients

This section lists hardware and software tools used for the performance analysis scenario.

- **Application**: A `miniFE` Finite Element Mini-Application that is available for download from https://github.com/Mantevo/miniFE (OpenMP version)
- **Compiler**: Intel® oneAPI DPC++/C++ Compiler 2024.0.0 or newer. This recipe relies on this compiler version to have necessary instrumentation inside Intel OpenMP runtime library used by VTune Profiler for analysis.
- **Performance analysis tools**:
  - VTune Profiler version 2024.0 or newer - HPC Performance Characterization analysis
  - Intel® Inspector 2022.1: Threading Error analysis

  NOTE
  Get the latest version of Intel® Inspector from this download page.

- **Operating system**: Linux*, Ubuntu* 20.04.6
- **CPU**: 11th Gen Intel® Core™ i9-11900KB @ 3.30GHz

Create a Baseline

1. Use the `openmp/src/Makefile.intel.openmp` make file to build the application.
   - To enable debug information, add the `-g` option.
   - For simpler identification, you can see source file information in the names of the OpenMP regions by including the `-parallel-source-info=2` compiler option.
2. Run the compiled application with these parameters:
   - nx=200
   - ny=200
   - nz=200

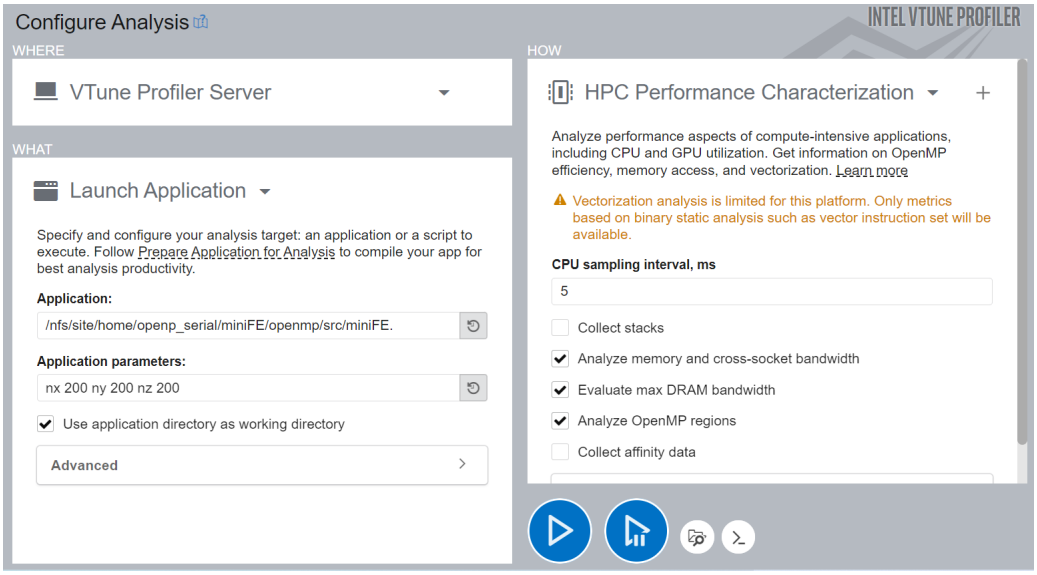
The number of OpenMP threads corresponds to the number of physical cores. With one thread running per core (OMP_NUM_THREADS=16, OMP_PLACES=cores), the application takes about 50 seconds.

This is a performance baseline. You use this baseline for further optimizations.

Run HPC Performance Characterization Analysis

To understand potential performance bottlenecks in the sample, run the HPC Performance Characterization analysis in VTune Profiler.

1. Click the **New Project** button on the toolbar and specify a name for the new project, for example: `miniFE`.
2. In the **Configure Analysis** window, set these options:
   - In the **WHERE** pane, select the **Local Host** target system type.
   - In the **WHAT** pane, select the **Launch Application** target type.
   - Specify an application for analysis and use these parameters: `nx 200 ny 200 nz 200`.
   - In the **HOW** pane, from the **Parallelism** group, select **HPC Performance Characterization** .
3. Click the  **Start** button to run the analysis.



To run the analysis from the command line, use this command:

```
vtune -collect hpc-performance -data-limit=0 ./miniFE.x nx 200 ny 200 nz 200
```

Once data collection is complete, VTune Profiler processes the result and resolves symbol information. This is necessary for source analysis.

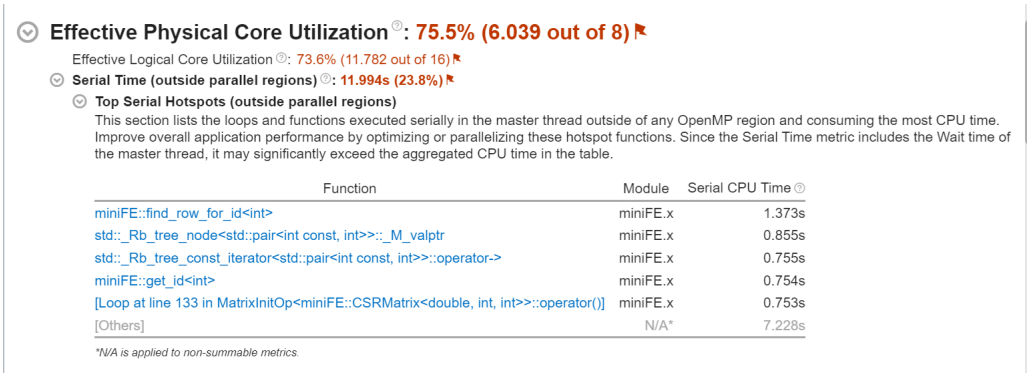
Identify OpenMP Serial Time

Using the HPC Performance Characterization analysis, you can see HPC metrics that help you to understand these performance bottlenecks:

- CPU utilization (parallelism)
- Memory access efficiency
- Vectorization

For applications which use the Intel OpenMP runtime, you can benefit from special OpenMP efficiency metrics that help identify issues with threading parallelism.

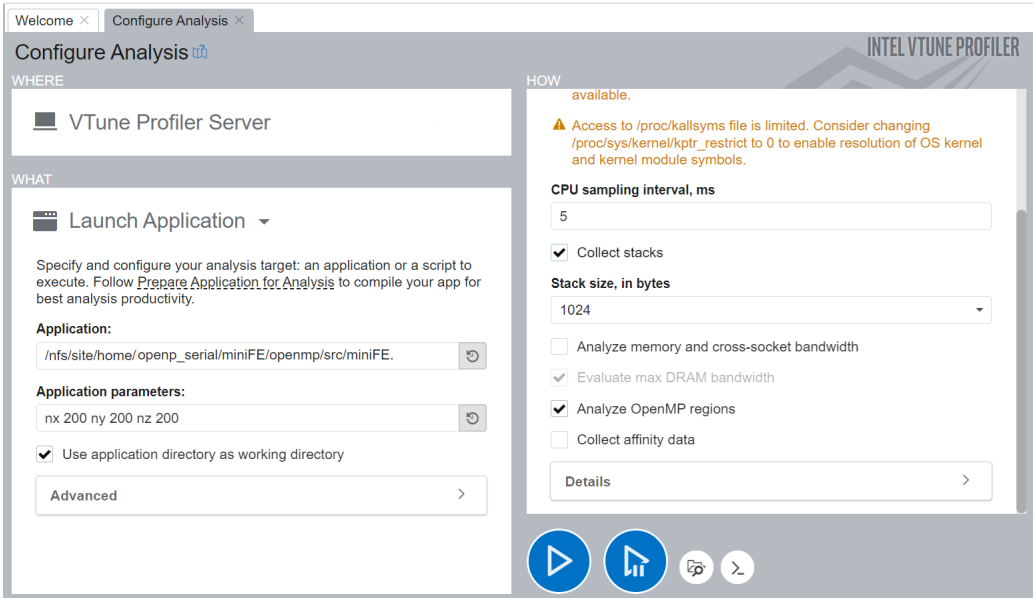
Start your analysis with the **Summary** view where you see application-level statistics. The **Effective Physical Core Utilization** metric has been flagged. This signals a performance problem that requires investigation:



When you dive deeper into the metric hierarchy, you see that the **Serial Time (outside parallel regions)** of the application occupies ~24% of its elapsed time.

Let us look at the serial hotspot in the matrix initialization code, which is the loop at line 133.

Consider running the HPC Performance Characterization analysis with call stacks to explore available optimization opportunities. Using call stacks can help you find a candidate for parallelism at a proper level of granularity. Since the call stack collection is not compatible with memory bandwidth analysis, make sure to disable the **Analyze memory bandwidth** configuration option first:

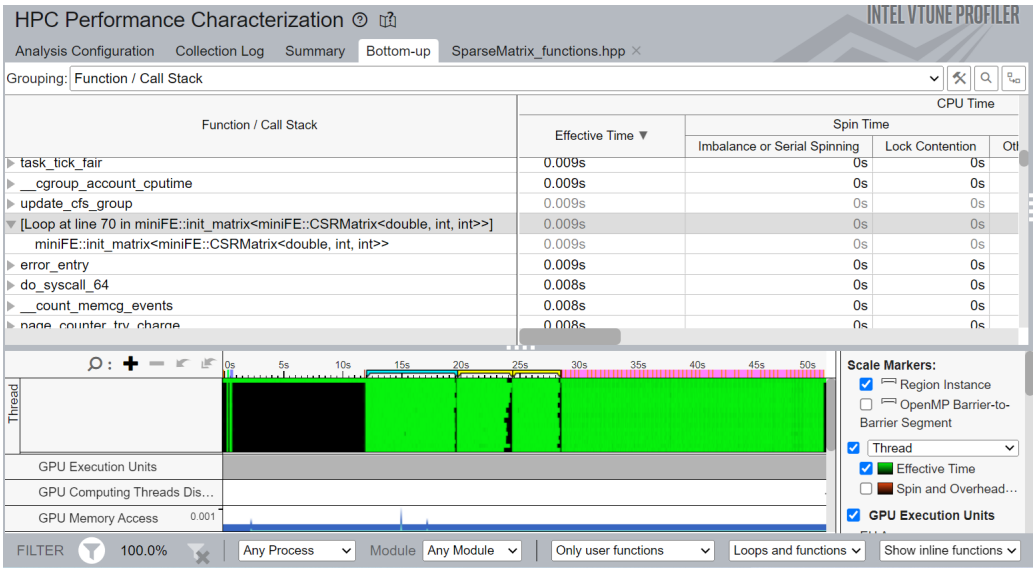


To run this analysis from the command line, use this command:

```
vtune -collect hpc-performance -data-limit=0 -knob enable-stack-collection=true -knob collect-memory-bandwidth=false ./miniFE.x nx 200 ny 200 nz 200
```

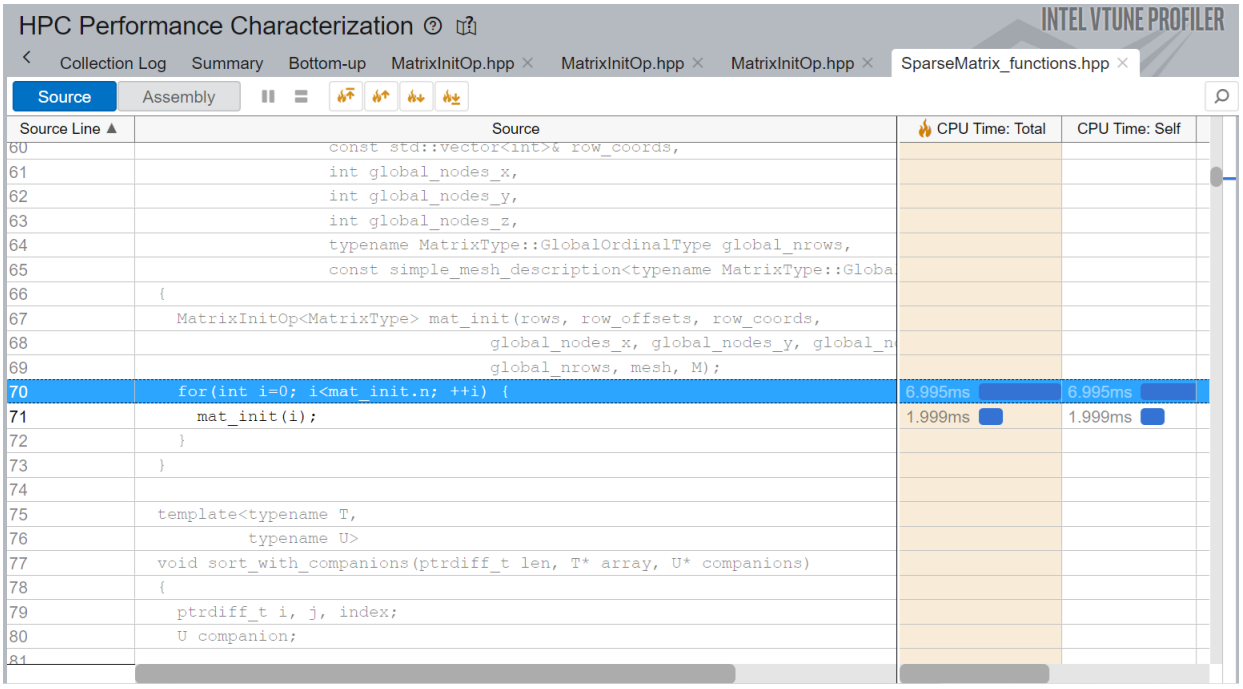
NOTE
While you can also use the Threading analysis to analyze OpenMP Serial Time with stacks, the HPC Performance Characterization analysis is a better starting point for a high level understanding of performance bottlenecks.

Once the data collection finishes, VTune Profiler displays results starting with the **Summary** view. To identify top hot spots and see their call stacks, switch to the **Bottom-up** view.



`SparseMatrix_functions.hpp` has a loop with iterations by matrix elements. This is a good location to insert parallelism.

Double click this row to open the source file at that location:



Parallelize the Code

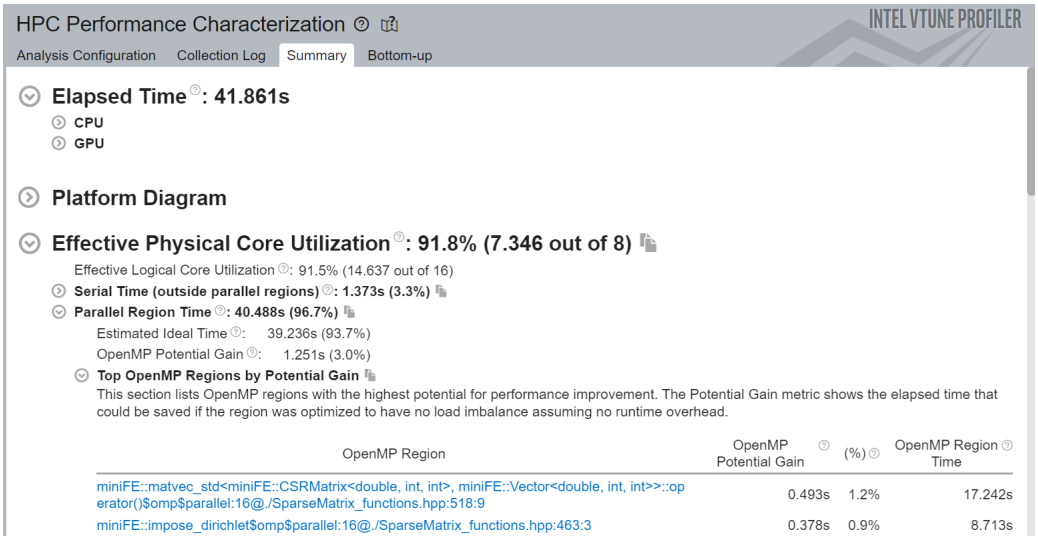
To make the matrix initialization parallelized by OpenMP, insert the `omp parallel for` pragma :

```
#pragma omp parallel for
for(int i=0; i<mat_init.n; ++i) {
    mat_init(i);
```

Re-compile the application and compare the execution time versus the original performance baseline to verify your optimization.

In this recipe, the **Elapsed time** of the application after optimization is approximately 42 seconds, which is ~21% speed-up of the application execution.

Re-run the HPC Performance Characterization analysis for the optimized version of the application:



Overall,

- The **Effective Physical Core Utilization** has improved by 16%.
- The fraction of **OpenMP Serial Time** has reduced to 3.3%.
- VTune Profiler does not flag the **OpenMP Serial Time** metric since the threshold is 15%.

To further improve parallel efficiency, you can analyze the most imbalanced barriers. For more information, see OpenMP Imbalance and Scheduling Overhead.

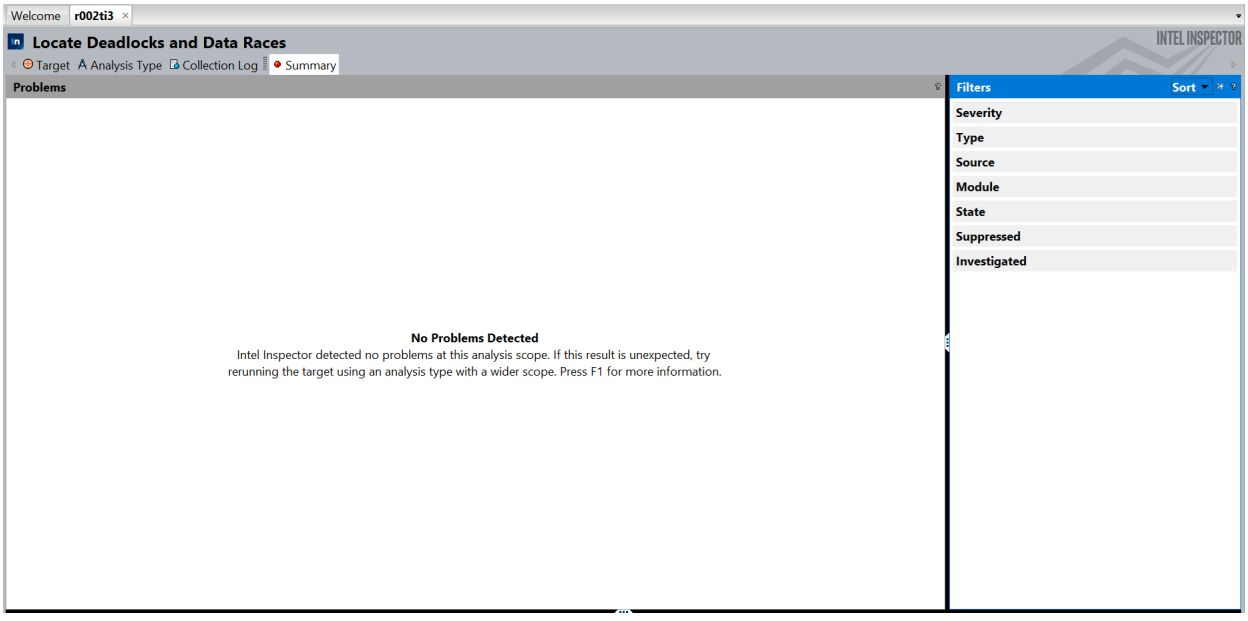
Inspect Threading Errors

To complete your analysis for parallelism, check your code for threading errors like data races or deadlocks. Use Intel® Inspector to find potential data races and deadlocks that may not happen in particular hardware but can hurt in a different environment or even when the same environment uses different settings.

To speed up the check, use the command line interface and reduce the workload size:

```
inspxe-cl -collect ti3 ./miniFE.x nx 40 ny 40 nz 40
```

You see that the Intel Inspector does not report any issues for the parallelized code.



Discuss this recipe in the Analyzer forum.

See Also

OpenMP* Code Analysis Method

HPC Performance Characterization Analysis

Threading Analysis

Tutorial: Analyzing an OpenMP* and MPI Application

Scheduling Overhead in an Intel® oneAPI Threading Building Blocks Application

Detect and fix scheduling overhead in an Intel® oneAPI Threading Building Blocks (oneTBB) application.

Content expert: Jennifer Dimatteo

> **NOTE** Intel® Threading Building Blocks library (previously a part of Intel® Parallel Studio XE and Intel® System Studio packages) has been replaced by Intel® oneAPI Threading Building Blocks (oneTBB). Download oneTBB from the Intel® oneAPI Base Toolkit.

During dynamic distribution of fine-grain chunks of work between threads, you can encounter scheduling overhead. When this happens, parallelism can be inefficient due to two reasons:

- The scheduler takes a significant amount of time to assign work for working threads.
- Working threads spend a lot of time waiting to receive new chunks of work.

In extreme cases, the threaded version of a program can actually be slower than the sequential version. The majority of oneTBB constructs use a default auto-partitioner. To avoid overhead, the auto-partitioner tailors the number of chunks larger than the default grain size.

If you use a simple partitioner either intentionally or with constructs like `parallel_deterministic_reduce`, you should take care of a grain size. A simple partitioner divides the work into chunk sizes up to the default grain size of one iteration.

In this recipe, learn how to use Intel® VTune™ Profiler to detect scheduling overhead in a oneTBB application. The profiling results also give advice on increasing the grain size to avoid an associated slowdown.

- INGREDIENTS
- DIRECTIONS:

  1. Create a baseline
  2. Run Threading analysis
  3. Identify scheduling overhead
  4. Increase grain size of parallel work

Ingredients

This section lists the hardware and software tools used for the performance analysis scenario.

- **Application:** A sample application that calculates the sum of vector elements using the oneTBB `parallel_deterministic_reduce` template function.
- **Compiler** : Intel® Compiler or GNU* compiler with compiler/linker options, for example:

```
icpx -I <tbb_install_dir>/include -g -O2 -std=c++17 -o vector-reduce vector-
reduce.cpp -L <tbb_install_dir>/lib/intel64/gcc4.8 -ltbb
```

- **Performance analysis tools:**VTune Profiler - Threading analysis
- **Operating system:** Ubuntu* 16.04 LTS
- **CPU:** Intel Xeon® CPU E5-2699 v4 @ 2.20GHz

Create a Baseline

The initial version of the sample code uses `parallel_deterministic_reduce` with the default grain size (line 17-23):

```
#include <stdlib.h>
#include "tbb/tbb.h"

static const size_t SIZE = 50*1000*1000;
double v[SIZE];

using namespace tbb;

void VectorInit( double *v, size_t n )
{
    parallel_for( size_t( 0 ), n, size_t( 1 ), [=](size_t i){ v[i] = i * 2; } );
}

double VectorReduction( double *v, size_t n )
{

    return parallel_deterministic_reduce(
        blocked_range<double*>( v, v + n ),
        0.f,
        [](const blocked_range<double*>& r, double value)->double {
            return std::accumulate(r.begin(), r.end(), value);
        },
        std::plus<double>()
    );
}

int main(int argc, char *argv[])
{
    task_scheduler_init( task_scheduler_init::automatic );

    VectorInit( v, SIZE );

    double sum;

    for (int i=0; i<100; i++)
          sum = VectorReduction( v, SIZE );

    return 0;
}
```

To make compute work more significant and measurable for statistical analysis, the vector sum calculation is repeated in the loop in line 35 .

Running the compiled application takes about 9 seconds. This is a performance baseline that you use for further optimizations.

Run Threading Analysis

To estimate threading parallelism in your application and the time spent on scheduling overhead, run the Threading analysis in VTune Profiler:

1. Click the

   New Project button on the toolbar and specify a name for the new project, for example: `vector-reduce`.
2. Click **Create Project**.

   The **Configure Analysis** window opens.
3. On the **WHERE** pane, select the **Local Host** target system type.
4. On the **WHAT** pane, select the **Launch Application** target type and specify an application for analysis.
5. On the **HOW** pane, click the browse button and select **Parallelism > Threading** analysis.
6. Click the

   Start button.

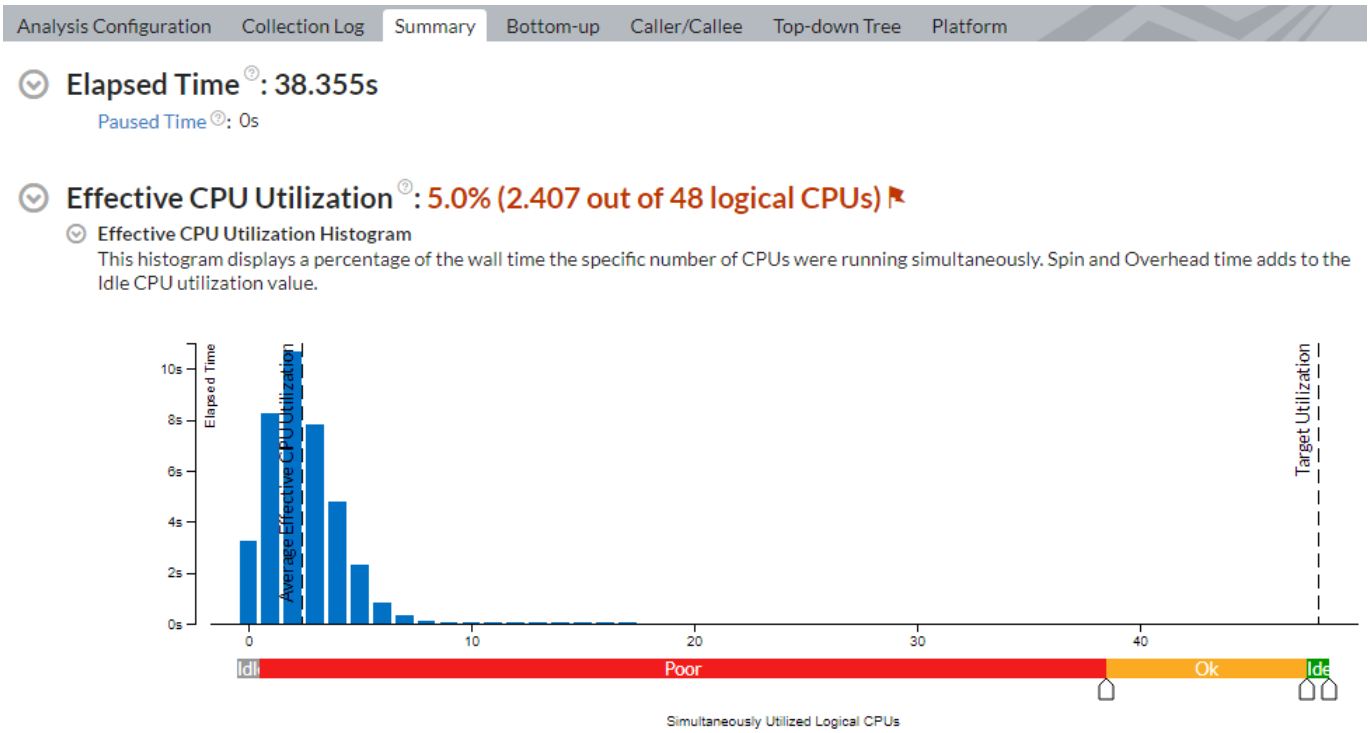
VTune Profiler runs the application and collects data. Once data collection is finalized, the symbol information is resolved. This is necessary for successful source analysis.

> **NOTE**
> Since the analysis is based on instrumentation and uses a stack stitching technology, the elapsed time of the instrumented application can be slower than the original application run because of the collection overhead.

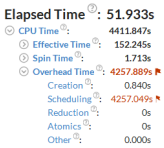
Identify Scheduling Overhead

Start your analysis with the **Summary** view that displays application-level statistics.

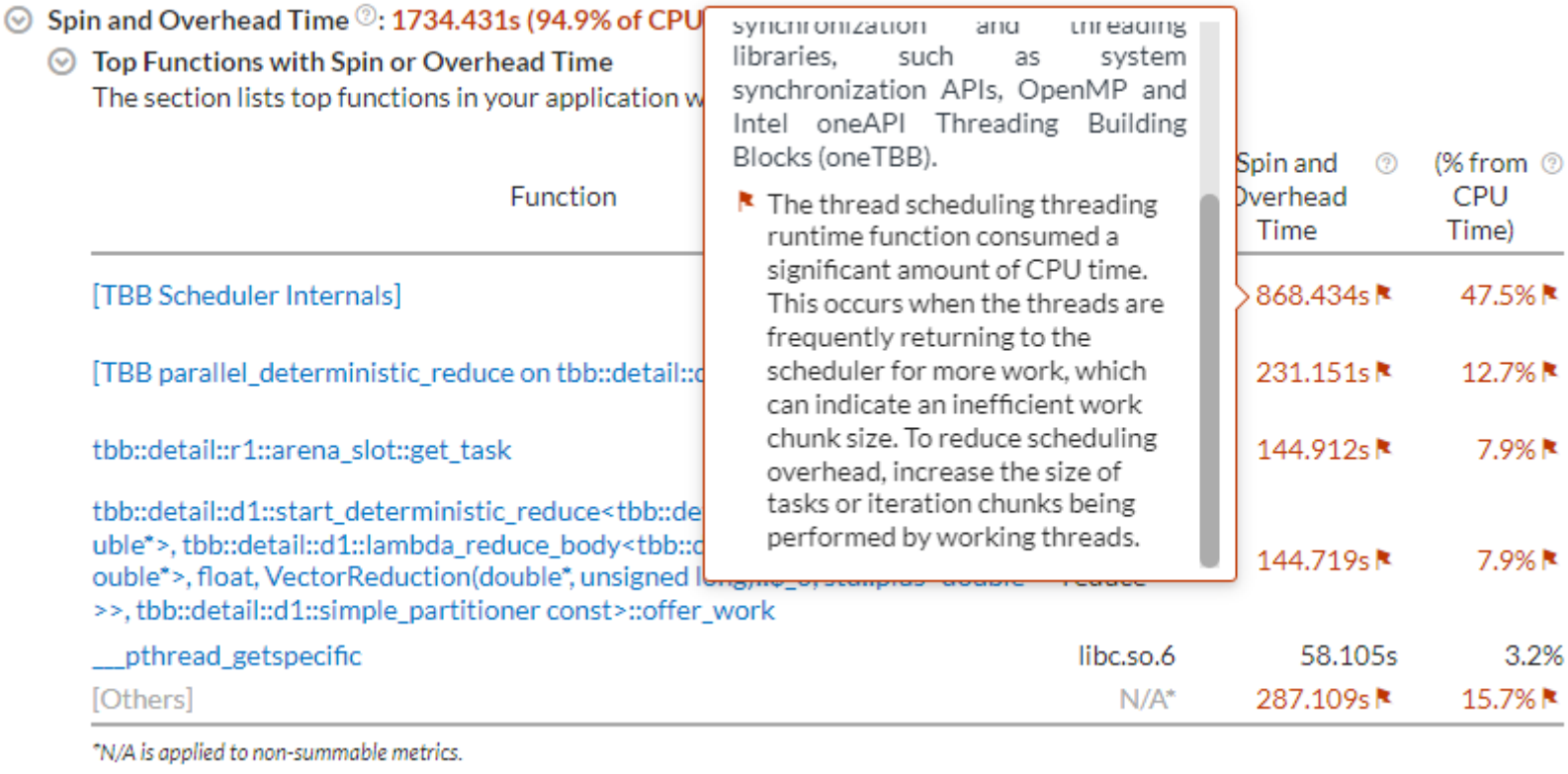
The **Effective CPU Utilization Histogram** shows that, on average, the application utilized only ~3 physical cores out of 48 available cores:



Flagged **Overhead Time** metric and the **Scheduling** sub-metric signal a threading inefficiency problem that should be explored:



The **Spin and Overhead Time** section shows `[TBB Scheduler Internals]` as the main time-consuming function. The hint on the flag associated with the function informs about scheduling overhead that should be addressed by increasing a grain size of parallel work:

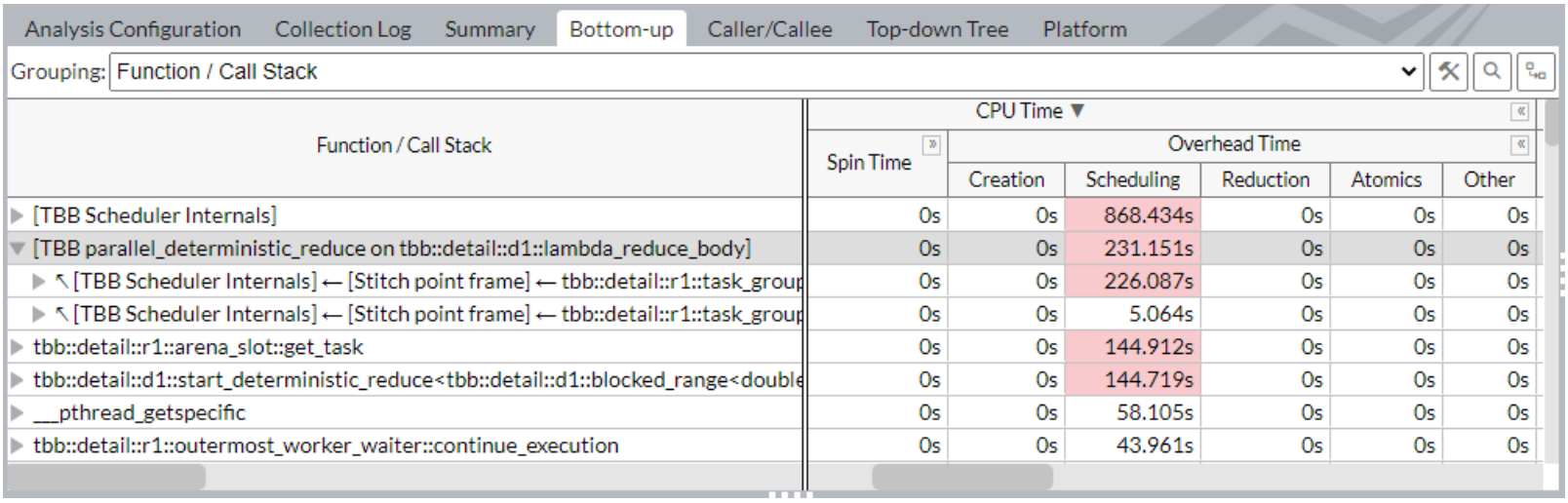


The sample application contains two oneTBB constructs:

- `parallel_for` initialization
- `parallel_deterministic_reduce` to calculate the sum of vector elements

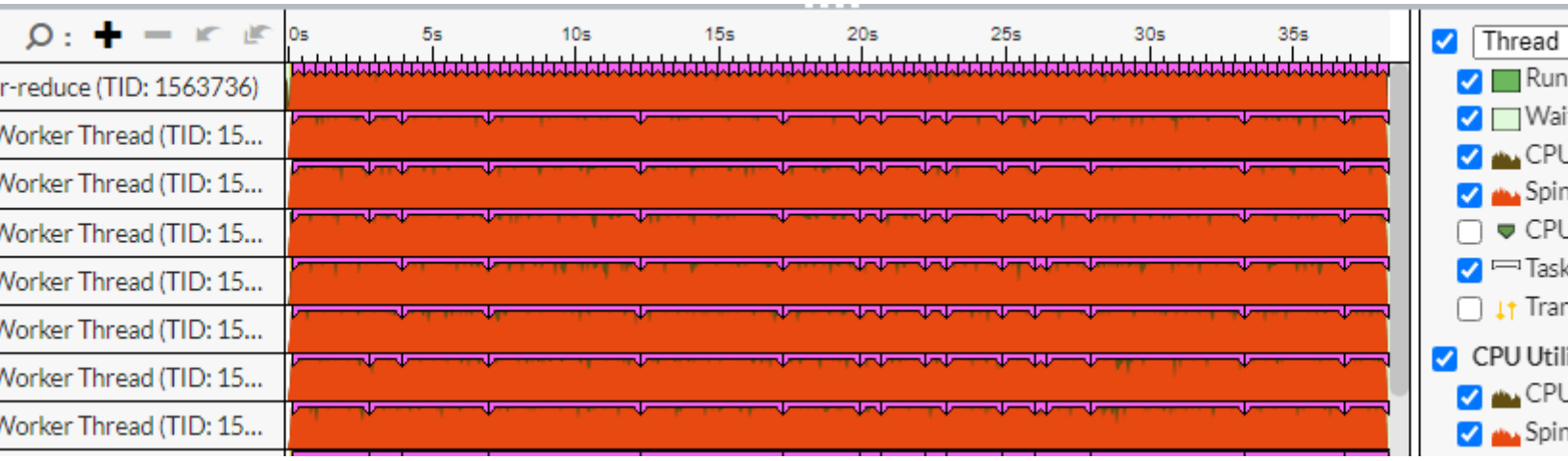
To identify the oneTBB construct that introduces overhead, switch to the **Bottom-up** view.

Expand the **CPU Time > Overhead Time** columns to sort the grid by the **Scheduling** column. Look at the first row with the oneTBB parallel construct in the **Function** list:



This row points to the `parallel_deterministic_reduce` construct in the `VectorReduction` function that has the highest scheduling overhead. Try to make work chunks more coarse-grain to eliminate the overhead in parallelization with this construct.

NOTE
To better visualize the imbalance during the application run, use the **Timeline** view. The brown sections show useful work and the red sections show where time was wasted.



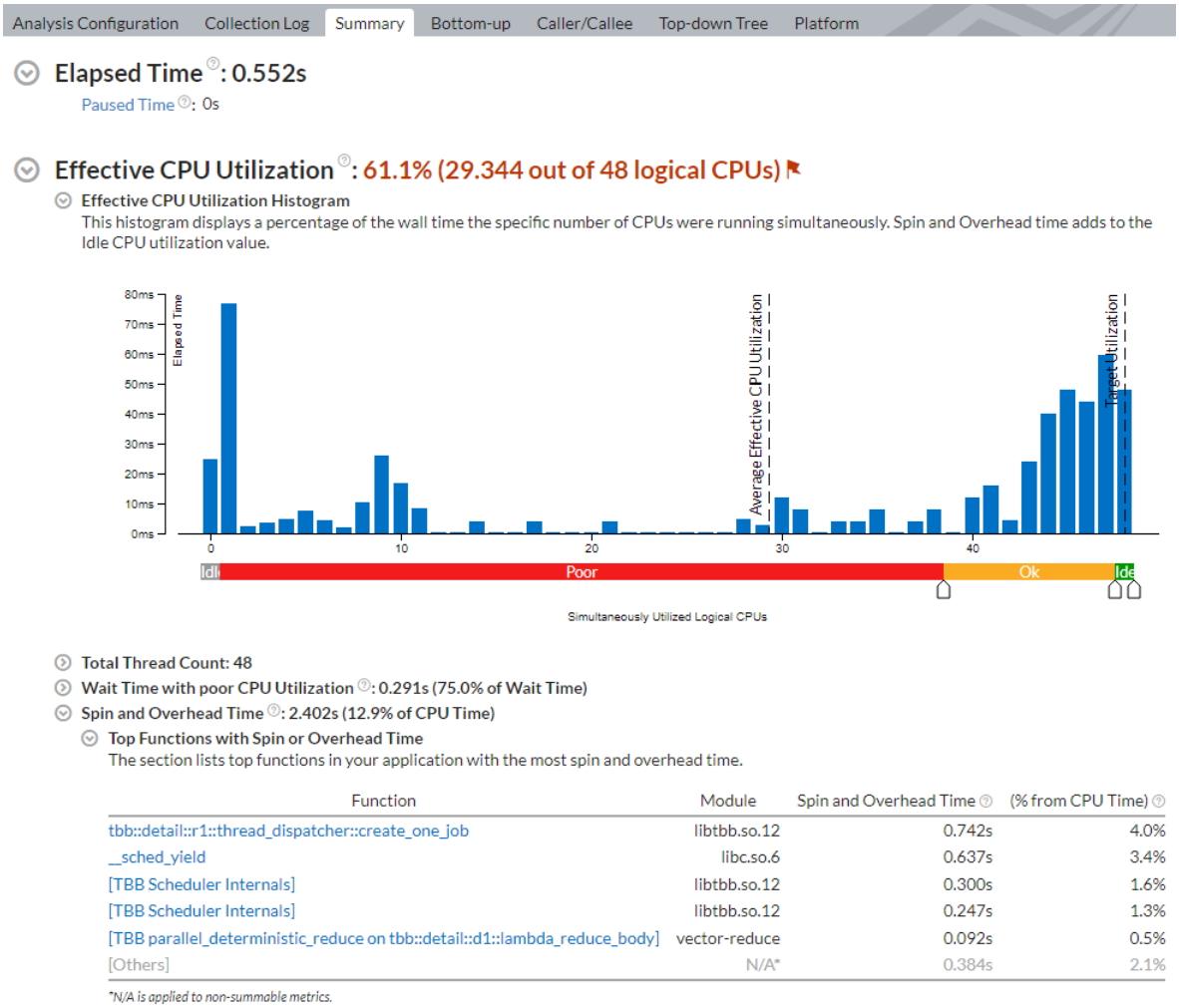
Increase Grain Size of Parallel Work

The fine-grain chunks of work which were assigned to worker threads cannot compensate the time spent by the scheduler on the work assignment. The default chunk size for `parallel_deterministic_reduce` that uses a simple partitioner is 1. This means that worker threads will take just one loop iteration to execute before turning back to the scheduler for a new portion of work. Consider increasing the minimal chunk size to 10000 (line 5 in the snippet below):

```
double VectorReduction( double *v, size_t n )
{

    return parallel_deterministic_reduce(
        blocked_range<double*>( v, v + n, 10000 ),
        0.f,
```

```
        [](const blocked_range<double*>& r, double value)->double {
            return std::accumulate(r.begin(), r.end(), value);
        },
        std::plus<double>()
    );
}
```

Repeat the Threading analysis:



You see that the elapsed time of the application has significantly reduced.

- The average effective CPU utilization is ~29 logical cores. This is because the metric counts a warmup phase. In the compute phase, CPU utilization is closer to 80 cores.
- The CPU time spent on oneTBB scheduling or other parallel work arrangement is negligible.
- The modified code runs more than 10x faster than the original version of the application without collection time.

See Also

Threading Analysis

PMDK Application Overhead

Find and fix an overhead on memory accesses for a PMDK-based application.

Content Expert: Eugeny Parshutin, Alexander Antonov

Persistent Memory Development Kit (PMDK) provides support for transactional and atomic operations to keep the data consistent and durable. This kit is a collection of open source libraries and utilities that are available for 64-bit Linux* OS. You can learn more about PMDK from the Persistent Memory Programming web site (pmem.io).

In addition to memory and storage tiers, persistent memory from Intel contains an additional persistent memory tier. This third tier offers greater capacity than DRAM and significantly faster performance than storage. Applications can access data structures resident on persistent memory in-place, in the same way they access traditional memory. This eliminates the need to page blocks of data back and forth between memory and storage.

However, this benefit of using PMDK libraries can influence application performance. This recipe describes how you use Intel® VTune™ Profiler to detect these issues.

- INGREDIENTS
- DIRECTIONS:

  1. Run Memory Access analysis on the PMDK Application
  2. Identify Hot Spots in the PMDK Application
  3. Remove redundant PMDK function calls

Ingredients

This section lists the hardware and software tools used for the performance analysis scenario:

- **Application:** a sample application that calculates the sum of two vector elementwise using PMDK memory allocators.
- **Compiler**: GNU* compiler with the following compiler/linker options:

  ```
  gcc -c -o array.o -O2 -g -fopenmp -I <pmdk-install-dir>/src/include -I <pmdk-install-dir>/src/examples array.c
  ```

  ```
  gcc -o arrayBefore array.o -fopenmp -L <pmdk-install-dir>/src/nondebug -lpmemobj -lpmem -pthread
  ```
- **Performance analysis tools:**Intel® VTune™ Profiler 2024.0: Memory Access / Hotspots analyses
- **Operating system:** Ubuntu* 16.04 LTS
- **CPU:** Intel® Core™ i7-6700K CPU @ 4.00GHz

Run Memory Access Analysis on the PMDK Application

This recipe starts with a sample application that utilizes the persistent memory. This application uses a triad kernel from a stream benchmark and should fully utilize the DRAM bandwidth.

In this sample, the vector sum calculation is repeated in the loop to make compute work more significant and measurable for statistical analysis:

```
#include <ex_common.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <libpmemobj.h>
#include <omp.h>

#define REPEATS 32
```

```
POBJ_LAYOUT_BEGIN(array);
POBJ_LAYOUT_TOID(array, int);
POBJ_LAYOUT_END(array);

int
main()
{
    size_t size = 82955000;
    size_t pool_size = 16200000000;
    int i,j;
    int multiplier = 3;

    PMEMobjpool *pop;
    char* path = "test_file1";
    if (file_exists(path) != 0)
    {
        if ((pop = pmemobj_create(path, POBJ_LAYOUT_NAME(array),
            pool_size, CREATE_MODE_RW)) == NULL)
        {
            printf("failed to create pool\n");
            return 1;
        }
    }
    else
    {
        if ((pop = pmemobj_open(path, POBJ_LAYOUT_NAME(array))) == NULL)
        {
            printf("failed to open pool\n");
            return 1;
        }
    }

    TOID(int) a;
    TOID(int) b;
    TOID(int) c;

    POBJ_ALLOC(pop, &a, int, sizeof(int) * size, NULL, NULL);
    POBJ_ALLOC(pop, &b, int, sizeof(int) * size, NULL, NULL);
    POBJ_ALLOC(pop, &c, int, sizeof(int) * size, NULL, NULL);

    for (i = 0; i < size; i++)
    {
        D_RW(a)[i] = (int)i;
        D_RW(b)[i] = (int)i+100;
        D_RW(c)[i] = (int)i+3;
    }

    pmemobj_persist(pop, D_RW(a), size * sizeof(*D_RW(a)));
    pmemobj_persist(pop, D_RW(b), size * sizeof(*D_RW(b)));
    pmemobj_persist(pop, D_RW(c), size * sizeof(*D_RW(c)));

    for (j = 0; j < REPEATS; j++)
    {
        #pragma omp parallel for
        for (i = 0; i < size; i++)
        {
```

```
            D_RW(c)[i] = multiplier * D_RO(a)[i] + D_RO(b)[i];
        }
    }

    POBJ_FREE(&a);
    POBJ_FREE(&b);
    POBJ_FREE(&c);

    pmemobj_close(pop);
    return 0;
}
```

To identify performance issues in the sample code and estimate the time spent on memory accesses, run the Memory Access analysis in Intel VTune Profiler:

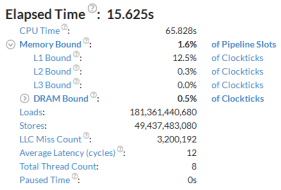
1. Open Intel® VTune™ Profiler.
2. Click the **New Project** button on the welcome screen and specify a name for the new project, for example: `arraysum`.
3. In the **Analysis Target** window, select the **Local host** target system for the host-based analysis.
4. Select the **Launch Application** target type and specify an application for analysis on the right pane.
5. Click the **Choose Analysis** button on the right, select **Microarchitecture Analysis > Memory Access** on the left pane and click **Start** to run the analysis.

   Intel VTune Profiler collects data and finalizes the data collection result resolving symbol information. This is necessary for successful source analysis.

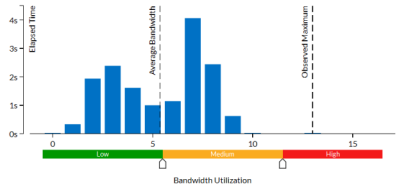
Identify Hot Spots in the PMDK Application

Start your analysis in the **Summary** view. Here you see application-level statistics per hardware metrics. Typically, the basic performance baseline is the application Elapsed time, which is equal to ~16 seconds for this sample code.

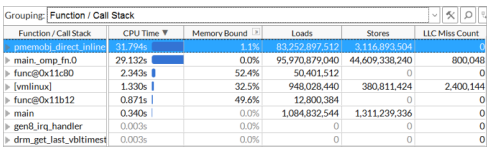
In spite of the expected high DRAM utilization for the PMDK code, the summary metrics do not define this sample app as DRAM bandwidth bound:



The **Bandwidth Utilization Histogram** also shows that the application underutilized the DRAM bandwidth with the **Observed Maximum** about 13 GB/sec, which is much less than expected:



The PMDK has introduced some overhead into the code. For details, switch to **Bottom-Up** view and choose the **Function / Call Stack** grouping level:



Grouping: Function / Call Stack

Function / Call Stack	CPU Time ▼	Memory Bound	Loads	Stores	LLC Miss Count
pmemobj_direct_inline	31.794s	1.1%	83,252,897,512	3,116,893,504	0
main._omp_fn.0	29.132s	0.0%	95,970,879,040	44,609,338,240	800,048
func@0x11c80	2.343s	52.4%	50,401,512	0	0
[vmlinux]	1.330s	32.5%	948,028,440	380,811,424	2,400,144
func@0x11b12	0.871s	49.6%	12,800,384	0	0
main	0.340s	0.0%	1,084,832,544	1,311,239,336	0
gen8_irq_handler	0.003s	0.0%	0	0	0
drm_get_last_vbltimest	0.003s	0.0%	0	0	0

The largest hotspot is `pmemobj_direct_inline`. This is a function called inside `D_RO` and `D_RW` macros. Double-click the function to view the source code in *<pmdk-install-dir>*`/src/include/libpmemobj/types.h`:

```
#define DIRECT_RW(o) \
    (reinterpret_cast < __typeof__((o)._type) > (pmemobj_direct((o).oid)))
#define DIRECT_RO(o) \
    (reinterpret_cast < const __typeof__((o)._type) > \
    (pmemobj_direct((o).oid)))

#endif /* (defined(_MSC_VER) || defined(__cplusplus)) */

#define D_RW    DIRECT_RW
#define D_RO    DIRECT_RO
```

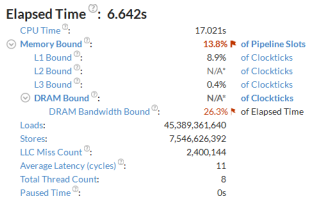
NOTE
To better visualize the DRAM bandwidth utilization during the application run, open the **Platform** view. The DRAM Bandwidth shows up in green and blue.

Remove Redundant PMDK Function Calls

Since the memory for each array is allocated as one chunk, you only need to call `D_RO` and `D_RW` once before the calculation to get the start addresses of the array:

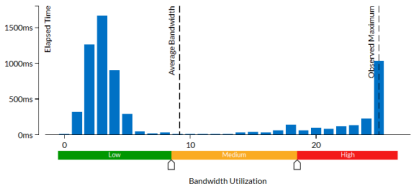
```
int* _c = D_RW(c);
const int* _a = D_RO(a);
const int* _b = D_RO(b);

for (j = 0; j < REPEATS; j++)
{
    #pragma omp parallel for
    for (i = 0; i < size; i++)
    {
      _c[i] = multiplier * _a[i] + _b[i];
    }
}
```

Re-compile the application and re-run the Memory Access analysis to see how this change affected the performance:



You see that the Elapsed time of the application has significantly reduced. PMDK overhead does not influence the performance.

The **Bandwidth Utilization Histogram** shows that the application fully utilizes DRAM bandwidth with the **Observed Maximum** about 25 GB/sec:



See Also

Introduction to Programming with Persistent Memory from Intel

Memory Usage View

Notices and Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

*Other names and brands may be claimed as the property of others.