intel software

THE PARALLEL UNIVERSE

Trying the Future of AI Development on an AI PC

oneMKL Random Number Generators Device API in Financial Services Risk Calculation

On the Migration of OpenACC* API to OpenMP* API

Issue 56
2024

Contents

Letter from the Editor 3

FEATURE

Trying the Future of AI Development on an AI PC 4

oneMKL Random Number Generators Device API in Financial Services Risk Calculation 11

On the Migration of OpenACC* API to OpenMP* API 18

Efficient Natural Language Embedding Models with Intel® Extension for Transformers 25
Making Retrieval-Augmented Generation More Efficient

Learn SYCL* in an Hour (Maybe Less) 31

Our Intel® C++ Compiler Is the First to Earn Khronos SYCL* 2020 Conformance 37

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

Letter from the Editor

James R. Reinders is an engineer at Intel with longtime parallel computing experience. James was the original editor of Parallel Universe Magazine and has coauthored twelve technical books related to parallel programming. His most recent book teaches the use of C++ with SYCL*. James has had the great fortune to help make key contributions to a couple of the world's fastest computers (#1 on the Top 500 list), as well as many other supercomputers and software developer tools.

I am covering this issue while Henry is out for a well-deserved sabbatical.

My journey in parallel computing has been focused on C++ with SYCL* for the past few years. It is exciting that the Intel® compiler team reached conformance with their C++ compiler for their SYCL support. Greg, a key member of the Intel compiler team, briefly shares this news with us.

In this issue, we have several how-to-do-it articles that invite us to get our feet wet by trying out things the authors show us how they did step-by-step. In the feature piece, **Trying the Future of AI Development on an AI PC**, we can get OpenAI* Triton running on an AI PC. We have an article for you that details methods and experiences for anyone wanting to migrate OpenACC* code to OpenMP*. To help gain a quick understanding of SYCL, I have contributed a piece based on some quick intro sessions I have taught in the past year.

We have an article on random numbers for financial risk simulation. The topics of high-quality random numbers and good financial-risk simulations are serious indeed; the article shares important techniques.

Highlighting AI techniques, we have an article that shares an innovative approach to enhance the performance of embedding models for natural language processing.

For our feature piece, Tony Mongkolsmai shares an example of some AI fun he has tried on his personal AI PC. We hope you gain a few insights with his peek at running OpenAI Triton on an Intel® Core™ Ultra processor — the heart of the first AI PCs.

The AI PC is a very real and important next step in the continuous evolution of personal computing. We have come to expect that our personal computing devices support graphics and Wi-Fi*. In both cases, these were additions to PCs that were first accompanied by some marketing hype to encourage adoption. I am confident that the incorporation of some serious levels of AI support into PCs will become just as required as graphics and Wi-Fi connectivity. Why? Because it will make the user experience far better just like graphics and Wi-Fi have already done.

Future articles on AI and AI PC coverage will be a key part of our continuing commitment to rich coverage of all things high-performance and parallel. We are definitely in the early days of this exciting and important AI PC era.

I hope you enjoy this issue. We always appreciate your feedback and contributions.

James Reinders
April 2024

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

Trying the Future of AI Development on an AI PC

Tony Mongkolsmai, Intel Corporation

Learn How to Build and Run the Open Source OpenAI* Triton Backend for Intel® GPUs

As deep learning (DL) AI solutions become larger and larger, one of the biggest challenges facing AI developers is how to create performant models efficiently. Traditionally, AI model developers had to write their kernels in C++ and then expose it to Python* using pybind. This required AI developers to not only understand DL kernels and models, but also learn C++ and the associated C++ tensor abstractions for kernel development. Recognizing this challenge, OpenAI* released Triton, an open-source, domain-specific, Python-like programming language and compiler that enables developers to write efficient GPU code.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

OpenAI Triton Overview

Triton provides an intermediate layer between C++ and Python in terms of functionality and performance. The goal is to allow DL developers to build optimized kernels without having to implement them across multiple layers of the software stack. While Triton is not native Python, it does allow us to develop within the same source file as our Python code, which makes code management much simpler.

As DL inherently deals with large-scale, parallel computation, Triton is designed to work with blocks of data instead of individual elements. This inherently simplifies the syntax of the language where a variable, once defined, represents a set of data. For example, writing:

```
result = x + y
```

performs an elementwise addition of the `x` and `y` vectors and writes that output vector to the `result` variable.

Beyond a simplification of syntax, a key benefit of Triton is performance. Triton is a language *and* a compiler, which means there is the ability to map syntax to a specific hardware design. Neural networks are large and require significant memory operations to execute. Triton is defined intentionally to allow developers and the compiler backends to map the operations in the language to more optimized usage of cache hierarchies and memory than would be available in native Python.

This is just a small part of the functionality enabled by Triton, which includes a variety of operations essential for DL developers. For a deeper dive into Triton, check out OpenAI's Triton resources.

Triton on Intel® Core™ Ultra Processors

OpenAI is simplifying the software development challenges. As a developer, I want to develop my code *where* I want and *when* I want. One could buy a workstation-level laptop, but those tend to be heavy and clunky. Enter my new Intel® Core™ Ultra processor-based laptop, with its more powerful integrated GPU and neural processing unit. I decided to take it for a spin and try Triton on my brand-new AI PC.

Compiling Triton for Intel Core Ultra Processors

Triton is open source, but support for Intel® GPUs is in development and has yet to be upstreamed into the Triton main repository. Intel is developing their **Triton backend in open source here**. Similar to other hardware vendors, Intel is developing this backend off a fork of Triton and will work to upstream this work. While the GitHub* site mentions that this code is being developed and tested on Intel® Data Center GPU Max Series cards, software developed using oneAPI for any Intel GPU should work on other Intel GPUs too. So, I decided to try to build the code from the source and see how it works on my MSI* Prestige 16 AI Evo laptop. (As an aside, I also did it on my consumer-level Intel® Arc™ A770 GPU and it worked as well.)

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

System Setup

The system I am using has the following configuration:

- Ubuntu* 23.10, Kernel 6.5.0-17
- Intel® oneAPI Basekit 2024.0.1 (installation instructions)
- Anaconda* (installation instructions)

Note that once after installing Linux*, I did have to update `/etc/default/grub` to replace:

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash"
```

with

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash i915.force_probe=7d55"
```

where 7d55 is the PCI ID of the integrated GPU. You must do this as the upstream Ubuntu integrated driver does not yet recognize the PCI ID of the latest iGPU. Next, run

```
sudo update-grub
```

and then reboot.

Getting and Building Triton

Following the build instructions on GitHub can be challenging, so here is a breakdown of the commands I ran to build and run Triton on my AI PC. First, let's set up the environment:

```
# checkout Intel XPU Backend for Triton and change to that directory
git clone https://github.com/intel/intel-xpu-backend-for-triton.git -b llvm-target
cd intel-xpu-backend-for-triton

# Create a conda environment using Python 3.10 and activate the environment
conda create --name triton python=3.10
conda activate triton

# setup the oneAPI build and runtime environment
source /opt/intel/oneapi/setvars.sh

# install components as required by Triton build infrastructure and set their recommend
environmentvariable to build with Clang
pip install ninja cmake wheel
export TRITON_BUILD_WITH_CLANG_LLD=true
```

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation
or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

You will notice that I use Python 3.10 instead of 3.11 or later because Triton does not currently work with Python 3.11 or 3.12. Also, `llvm-target` is the main branch of the code, so you could get away without specifying the branch.

Following the instructions in the repo, we can run the build:

```
./scripts/compile-triton.sh
```

Unfortunately, this build throws an error that CUDA* is not found. A quick attempt to install the CUDA toolkit on 23.10 fails as there are some libraries it depends on that are not available out-of-the-box on 23.10. To work around this, we can add the 23.04 repository to our Ubuntu 23.10 APT sources by editing `/etc/apt/sources.list` and adding the following line:

```
deb http://archive.ubuntu.com/ubuntu/ lunar universe
```

and then do a quick

```
sudo apt update
```

which allows us to install CUDA.

Rerunning the build then hits another error, which is that the `clang` and `clang++` compilers are not found. Since I have the Intel oneAPI DPC++ compiler installed, which is based on clang, it is simple to point to the clang included with the Intel DPC++ compiler by setting the `PATH` variable:

```
export PATH=$PATH:/opt/intel/oneapi/compiler/latest/bin/compiler
```

One last run of the compile script and I am up and running.

```
          ********************************************************************************
          Please be careful with folders in your working directory with the same
          name as your package as they may take precedence during imports.
          ********************************************************************************

  !!
    with strategy, WheelFile(wheel_path, "w") as wheel_obj:
  Building editable for triton (pyproject.toml) ... done
  Created wheel for triton: filename=triton-3.0.0-0.editable-cp310-cp310-linux_x86_64.whl size=3072 sha256=bf4fd604e7ba68134d
2d935f96a9e96d0b62865cbf74081b39653a47a1708867
  Stored in directory: /tmp/pip-ephem-wheel-cache-5z1lh_f8/wheels/98/85/b6/6ad462c1a1735d0b1af8dba5e03cd79040eac6c513748a5631
Successfully built triton
Installing collected packages: triton
  Attempting uninstall: triton
    Found existing installation: triton 3.0.0
    Uninstalling triton-3.0.0:
      Created temporary directory: /tmp/pip-uninstall-yqqxblnw
      Removing file or directory /home/tonym/anaconda3/envs/triton/lib/python3.10/site-packages/__editable__.triton-3.0.0.pth
      Removing file or directory /home/tonym/anaconda3/envs/triton/lib/python3.10/site-packages/__editable___triton_3_0_0_fin
der.py
      Removing file or directory /home/tonym/anaconda3/envs/triton/lib/python3.10/site-packages/__pycache__/__editable___trit
on_3_0_0_finder.cpython-310.pyc
      Created temporary directory: /home/tonym/anaconda3/envs/triton/lib/python3.10/site-packages/~riton-3.0.0.dist-info
      Removing file or directory /home/tonym/anaconda3/envs/triton/lib/python3.10/site-packages/triton-3.0.0.dist-info/
      Successfully uninstalled triton-3.0.0

Successfully installed triton-3.0.0
Remote version of pip: 24.0
Local version of pip:  23.3.1
Was pip installed by pip? False
Removed build tracker: '/tmp/pip-build-tracker-ts1gk_qf'
(triton) tonym@prestige:/scratch/intel-xpu-backend-for-triton$
```

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

Testing Triton on Intel Core Ultra

A quick first test is to check out the `python/tutorials` directory within the `intel-xpu-backend-for-triton` code base. To evaluate some basic functionality, I am running the first couple of examples from the directory. Looking at the test codes, you will see that there are some basic changes from the OpenAI Triton repository to select the Intel GPU to be used. Changing code like this:

```
x = torch.rand(size, device='cuda')
y = torch.rand(size, device='cuda')
```

to this:

```
x = torch.rand(size, device='xpu')
y = torch.rand(size, device='xpu')
```

Setting Up the Runtime Environment

Trying out my new Triton build requires I add PyTorch* to my environment, so my first step is to install the Intel® Extension for PyTorch:

```
python -m pip install torch==2.1.0a0 torchvision==0.16.0a0 torchaudio==2.1.0a0 intel-extension-for-pytorch==2.1.10+xpu --extra-index-url https://pytorch-extension.intel.com/release-whl/stable/xpu/us/
```

The tutorial examples also require some additional Python libraries, which are easily installed using pip.

```
pip install matplotlib pandas
```

Running Examples

```
(triton) tonym@prestige:/scratch/intel-xpu-backend-for-triton/python/tutorials$ python 01-vector-add.py
/home/tonym/anaconda3/envs/triton/lib/python3.10/site-packages/torchvision/io/image.py:13: UserWarning: Failed to loa
d image Python extension: ''If you don't plan on using image functionality from `torchvision.io`, you can ignore this
 warning. Otherwise, there might be something wrong with your environment. Did you have `libjpeg` or `libpng` install
ed before building `torchvision` from source?
  warn(
No CUDA runtime is found, using CUDA_HOME='/usr/local/cuda'
tensor([1.3713, 1.3076, 0.4940,  ..., 0.8654, 1.1553, 0.4071], device='xpu:0')
tensor([1.3713, 1.3076, 0.4940,  ..., 0.8654, 1.1553, 0.4071], device='xpu:0')
The maximum difference between torch and triton is 0.0
vector-add-performance:
           size     Triton      Torch
0        4096.0   0.003057   0.003096
1        8192.0   0.006198   0.006193
2       16384.0   0.012383   0.012375
3       32768.0   0.024736   0.024673
4       65536.0   0.049506   0.049383
5      131072.0   0.098313   0.098653
6      262144.0   0.196897   0.196460
7      524288.0   0.388927   0.389345
8     1048576.0   0.769500   0.768111
9     2097152.0   1.496443   1.492719
10    4194304.0   2.858620   2.823484
11    8388608.0   5.277260   5.095062
12   16777216.0   9.116155   8.511318
13   33554432.0  14.523118  12.744209
14   67108864.0  15.574265  12.825385
15  134217728.0  14.566962  11.500764
```

The first tutorial is a simple vector add between two blocks of data. The output looks like this:

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. *Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

The output measures the correctness of the data and the relative performance of the operations in the vector addition. Here we can see that the numbers that I saw here are in line, but as the size of vectors increases, the Triton approach begins to provide improved performance on my Intel Core Ultra integrated GPU.

The second tutorial is a fused-softmax implementation. The softmax function maps a vector to a probability distribution of outcomes and is often used in DL algorithms as the last activation function of a neural network. As softmax is doing a signification number of operations across the input vector, vectorization and can result in significant performance benefits. The fused-softmax implementation aims to block data accesses to improve a naïve implementation. Note that there is already a native fused-softmax implementation in PyTorch, which means there is a hardware-optimized C++ version of this operation against which we can compare. Running the tutorial gives the following output:

```
(triton) tonym@prestige:/scratch/intel-xpu-backend-for-triton/python/tutorials$ python 02-fused-softmax.py
/home/tonym/anaconda3/envs/triton/lib/python3.10/site-packages/torchvision/io/image.py:13: UserWarning: Failed to loa
d image Python extension: ''If you don't plan on using image functionality from `torchvision.io`, you can ignore this
 warning. Otherwise, there might be something wrong with your environment. Did you have `libjpeg` or `libpng` install
ed before building `torchvision` from source?
  warn(
No CUDA runtime is found, using CUDA_HOME='/usr/local/cuda'
/scratch/intel-xpu-backend-for-triton/python/tutorials/02-fused-softmax.py:184: UserWarning: The grad mode is detecte
d as torch.no_grad() is NOT enabled. In this mode on XPU, please expect NO graph and fusion optimization will be appl
ied.
 (Triggered internally at /build/intel-pytorch-extension/csrc/gpu/jit/fusion_pass.cpp:829.)
  ms, min_ms, max_ms = triton.testing.do_bench(lambda: naive_softmax(x), quantiles=quantiles)
softmax-performance:
          N    Triton  Torch (native)  Torch (jit)
0     256.0  0.508614        0.520325     0.485650
1     384.0  0.738521        0.779249     0.711792
2     512.0  0.976950        1.029912     0.916427
3     640.0  1.210549        1.273200     1.120450
4     768.0  1.441097        1.514099     1.302048
..      ...       ...             ...          ...
93  12160.0  8.014298       13.657079     3.345754
94  12288.0  8.042449       13.338677     3.417873
95  12416.0  8.148614       13.899843     3.363594
96  12544.0  8.153968       13.621084     3.355575
97  12672.0  8.267947       15.834839     4.628440

[98 rows x 4 columns]
```

Here we can see the performance of three different implementations of softmax. The first column is the Triton version, the second is the C++-optimized version, and the third is the naïve version. For the Intel Core Ultra, the Triton version is faster than the naïve version, but still slower than the C++-native version of the code. This makes sense because the Torch C++ implementation has already been heavily optimized. However, the ability to write code that is much faster than the naïve implementation using only Python is exciting as a developer.

Conclusion

The Triton framework has significant support among DL kernel developers. The simple parallel syntax and ability to optimize data access on the underlying hardware, all from higher-level Python code, provide a promising path to simplifying DL kernel development. Combined with an Intel Core Ultra processor-based AI PC, this framework provides a promising path to do DL kernel development where I want, when I want. Development of the `intel-xpu-backend-for-triton` is a work in progress, but I am looking forward to reaping the benefits of this exciting hardware and software combination.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

intel® Developer Cloud

Accelerate Your Code's ROI

Sitting in the Comfort of Your Own Cloud

Electrify your code's capabilities with a single, open programming model that supports multiple languages to deliver accelerated heterogeneous computing. Firmly formed around open standards, the Intel® Developer Cloud offers the ideal enterprise environment for energizing your AI development—for exceptional performance.

A new era of accelerating computing across Enterprise, Cloud, HPC, & AI.

Start Now For Free

cloud.intel.com

oneMKL Random Number Generators Device API in Financial Services Risk Calculation

Andrey Fedorov, Math Algorithm Engineer; Gennady Fedorov, Software Technical Consulting Engineer; Vladimir Polin, AI Software Solutions Engineer; and Robert Mueller-Albrecht, Product Marketing Engineer, Intel Corporation

The Need for More Flexible Parallel Compute Power

The Basel Committee on Banking Supervision* (BCBS) forecasts a four- to twentyfold increase in compute demand due to the increasing need for financial risk mitigation measures. These include advanced market prediction algorithms and risk simulations to better understand asset volatility associated with climate change, supply chain disruption, and geopolitical changes.

Software developers in the banking and financial services industry (FSI) rely on high-performance computing (HPC) environments with GPU-based acceleration to implement these risk simulation models. The challenge they face is that the wide deployment of these workloads makes it paramount that they

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

can be run on a large variety of hardware platform configurations. Before the Unified Acceleration (UXL) Foundation's efforts and the ever-increasing popularity of SYCL*, they faced substantial obstacles and inefficiencies caused by the need to port and refactor code every time a workload had to be deployed in a new environment.

The Monte Carlo method is a well-known method in finance, relying on highly performant random number generation as seeds for its simulation scenarios. It is used for option pricing, as the expected return on investment (ROI) is often too complex to compute directly, especially with exotic options. American and European option models are two widely used Monte Carlo simulation methods to predict the probability of various option investment outcomes.

Since the introduction of the C++ with SYCL-based oneAPI Math Kernel Library (oneMKL) Interfaces in 2020, this component of the oneAPI specification and open source extension of the Intel® oneAPI Math Kernel Library provides Data Parallel C++ interfaces for random number generator (RNG) routines, implementing commonly used pseudorandom, quasi-random, and non-deterministic generators with continuous and discrete distributions.

Just like some other oneMKL function domains, as a part of its implementation oneMKL RNG includes:

1. Manual offload functionality (Random Number Generators Host API)
2. Device functionality as a set of functions callable directly from SYCL kernels (Random Number Generators Device Routines).

We will show how the oneMKL Random Number Generators (RNG) Device API helps you significantly increase the computation performance on Intel® Data Center GPUs by a factor of two, compared to the Host API implementation on the same hardware.

We do this by looking at Host API and Device API RNG function call performance for the European and American option pricing models.

In addition, we will use the American option pricing model, using the Monte Carlo algorithm, as an example to demonstrate in a bit more detail how to migrate financial workloads from proprietary CUDA* code to the open, multi-platform SYCL programming model using SYCLomatic (or the Intel® DPC++ Compatibility Tool).

Results were achieved on Intel Data Center GPU Max 1550 with the oneAPI 2024.0 release.

RNG Device API Basics

The main purpose of device interfaces is to make them callable from SYCL kernels. It brings an essential performance boost because submitting time may greatly affect the overall execution time of the application. The idea is to get random numbers on the fly and process them in the same kernel without paying for global memory transfer. For example:

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

Host API

```
auto engine = oneapi::mkl::rng::mrg32k3a(*stream, 1ull);
oneapi::mkl::rng::generate(oneapi::mkl::rng::gaussian<double>(0.0, 1.0),
                           engine, n, d_samples);

sycl_queue.parallel_for(
    sycl::nd_range<3>(sycl::range<1>(n_groups) * sycl::range<1>(n_items),
                      sycl::range<1>(n_items)),
    [=](sycl::nd_item<1> item_1) {
        post_processing_kernel(d_samples);
    }).wait();
```

Device API

```
sycl_queue.parallel_for(
    sycl::nd_range<3>(sycl::range<1>(n_groups) * sycl::range<1>(n_items),
                      sycl::range<1>(n_items)),
    [=](sycl::nd_item<1> item_1) {
        oneapi::mkl::rng::device::mrg32k3a<1> engine_device(1ull, n);
        oneapi::mkl::rng::device::gaussian<double> distr(0.0, 1.0);
        double rng_val = oneapi::mkl::rng::device::generate(distr, engine_
device);

        post_processing_kernel(rng_val);
    }).wait();
```

The Host API code sequence above shows two instances of calls being initiated from CPU to GPU:

1. The `generate()` function call that contains at least one SYCL kernel
2. The offloaded `parallel_for` loop kernel, including `nd_range` and `postprocessing`

The Device API code sequence has all this embedded within a single offloaded `parallel_for` SYCL kernel, minimizing the data exchange between CPU and GPU.

So, it is easy to see that the number of kernels required for the implementation when using the Device API is smaller than the Host API. The RNG Device API is also available as a part of oneMKL Interfaces, an open-source project that can be found on GitHub*.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

Let us consider American and European Monte Carlo workloads separately.

American Monte Carlo Option Pricing Model

To run this benchmark on an Intel® GPU, we took the original code from the NVIDIA* Developer Code Samples GitHub repository. We then migrated the native CUDA GPU code to SYCL using the SYCLomatic open-source project (SYCLomatic). This tool is included in the Intel® oneAPI Base Toolkit or available from GitHub under the Apache* 2.0 license. SYCLomatic allows us to port the original CUDA code to SYCL and automatically migrates about 95% of a general CUDA code to SYCL code.

Please find step-by-step instructions for migrating the American Monte Carlo option pricing model example source code to SYCL and inspect the SYCL-enabled example source code project in the GitHub repository.

To finish the process, we made some manual code changes and tuned it to the desired level of performance for our target architecture.

Additionally, after completing the SYCLomatic migration, we added a host interface call. To reduce the number of SYCL kernels used, we added device calls to the next kernel (`generate_paths_kernel`). This way, we reduce the number of kernels needed and remove the need for extra memory to store random numbers, as we use these numbers as soon as they are generated.

Applying the Device API interface capability, we obtain up to 2.13 times performance speed-up compared with traditional host interface calls. We observed a ~15% performance improvement for the overall application speed-up.

Figure 1 shows the performance scalability of American Monte Carlo benchmark results depending on the number of calculated paths. The **RNG & next kernel speedup** curve shows the performance increase of combining RNG device API and `generate_paths_kernel` usage compared to the Host API version using RNG Host interface calls and `generate_paths_kernel` separately. **whole bench speedup** shows overall the benchmark benefits of using RNG Device API compared to RNG Host API usage.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

American Monte Carlo Scalability Results

oneMKL RNG Device API vs oneMKL RNG Host API on Intel® Data Center GPU Max 1550

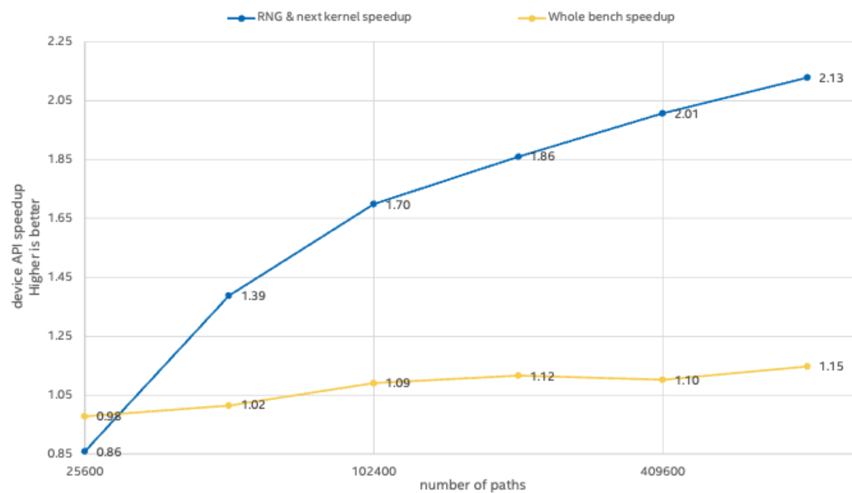

Figure 1. American Monte Carlo scalability results

Configuration details

Testing date: Performance results are **based on testing by Intel as of November 24, 2023**, and may not reflect all publicly available updates.

Configuration details and workload setup: 1-node, 2x Intel® Xeon® Platinum 8480+ CPU 56 cores, 512 GB; Ubuntu* 22.04.2 LTS, kernel: 5.15.47+prerelease70; Intel® oneAPI Math Kernel Library 2024.0 (oneMKL); Intel® DPC++ Compiler 2024.0.0 (2024.0.0.20231017); Intel® Level-Zero, Intel® Data Center GPU Max 1550 1.3 [1.3.26690]. GPU: Intel Data Center GPU Max 1550, GT frequency: 1.6 GHz, 1024 units, 2 tiles.

Performance varies by use, configuration, and other factors.

European Monte Carlo Option Pricing Model

Now that we looked at Device API and Host API performance for the American Monte Carlo option model, let us do the same for the European Monte Carlo option model and consider the performance of the Monte Carlo European Options Sample available on the oneAPI samples GitHub.

Let us see whether, for this use case, too, the use of the Device API is beneficial.

The difference from the American Monte Carlo model is that the code in the repository already uses a Device API. So, to be able to make the performance comparison, this time we need to add a Host API implementation. To add the Host API and model option pricing for a duration of one year, the `option_years` variable is being set to 1. After this consideration, we can easily replace device calls with a host call as a separate SYCL kernel. However, we need extra memory to store generated numbers.

Applying those changes, we found that operating memory on the GPU is insufficient to store ~100 000 000 000 double precision numbers as the original sample was configured to do. One drawback of Device API usage is the limited availability of localized memory on a GPU compared to a CPU, which can sometimes require more advanced

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

shared memory management for large datasets. To keep the reference example fundamentally unchanged, other than the use of device API vs host API, we decided to reduce the number of calculated asset pricing evolution paths to 16,000. This allowed the code to run successfully using device API implementation without any large code changes.

Comparing Device API and Host API performance for the European options pricing prediction model, we see that here, too, execution speed benefits from the use of GPU Device APIs.

Figure 2 shows how the RNG Device API usage allows users to obtain a 3.69 times performance improvement for the whole European Monte Carlo benchmark compared to RNG Host API.

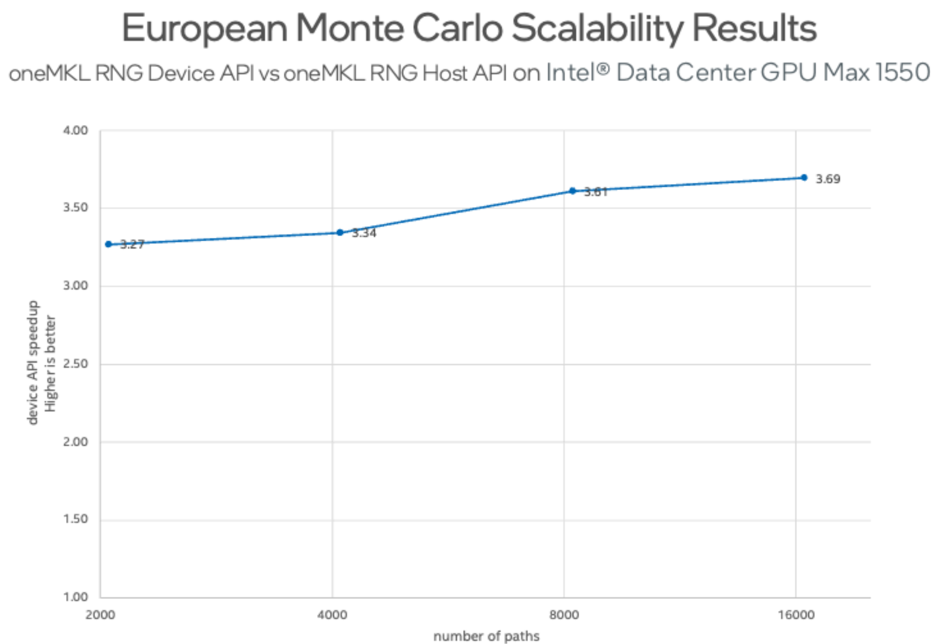

Figure 2. European Monte Carlo scalability results

Configuration details

Testing date: Performance results are **based on testing by Intel as of November 24, 2023**, and may not reflect all publicly available updates.

Configuration details and workload setup: 1-node, 2x Intel® Xeon® Platinum 8480+ CPU 56 cores, 512 GB; Ubuntu* 22.04.2 LTS, kernel: 5.15.47+prerelease70; Intel® oneAPI Math Kernel Library 2024.0 (oneMKL); Intel® DPC++ Compiler 2024.0.0 (2024.0.0.20231017); Intel Level-Zero, Intel® Data Center GPU Max 1550 1.3 [1.3.26690]. GPU: Intel Data Center GPU Max 1550, GT frequency: 1.6 GHz, 1024 units, 2 tiles.

Performance varies by use, configuration, and other factors.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

Use oneMKL RNG Device API Routines to Your Advantage

Device API allows users to achieve an essential performance boost compared to host interfaces but requires much more coding and knowledge about specific domain implementations. It is also a more flexible API since users can control the parameters of SYCL parallel routines.

Results in this article can be applied to other applications (not only financial) that have calculations of random numbers on the critical performance path.

Addressing challenges is a regular process for Intel® Compiler and oneMKL software development teams. To make Intel® software better for our customers, we are constantly looking for both more efficient algorithms and better optimization techniques.

To get started now, look at the Intel oneAPI Math Kernel Library, the oneAPI Math Kernel Library (oneMKL) Interfaces project, and SYCLomatic.

Additional Resources

- Intel oneAPI Math Kernel Library
- oneAPI Math Kernel Library (oneMKL) Interfaces
- SYCLomatic
- Intel DPC++ Compatibility Tool
- Unified Acceleration (UXL) Foundation

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

Sign up for future issues

On the Migration of OpenACC* API to OpenMP* API

Harald Servat, Shiquan Su, and Tobias Kloeffel, Intel Corporation; Ron Caplan, Predictive Science, Inc.; and Junyi Cheng, University of Colorado Boulder

Introduction

OpenACC* and OpenMP* both support offloading to accelerators. The earlier availability of OpenACC made it appealing to users willing to use GPU devices for their computational workloads. However, OpenACC supports a limited number of device vendors, and there has been an increasing demand to adopt OpenMP for offloading to accelerators, especially now with an increasing number of accelerator vendors supporting OpenMP offload.

The Intel® Application Migration Tool for OpenACC* to OpenMP* API is an open-source project that aims to help migrate OpenACC-based applications to OpenMP (reference to the IWOMP paper). This tool parses C/C++ and Fortran application sources, identifies OpenACC constructs, and proposes OpenMP (5.0 or higher) constructs that are semantically equivalent, when possible. Not all OpenACC constructs

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

can be translated to OpenMP due to a variety of reasons, including the descriptive/prescriptive differences between them. This is especially true when dealing with performance tuning options. Hence, the tool only focuses on providing a semantically correct migrated code but does not focus on providing a performance-optimized version of the migrated code, thus leaving performance optimizations to a later stage. The tool generates a message in the migration report if it cannot convert any of the OpenACC constructs. There is further information on the mapping between the two languages in the IWOMP paper. Once the application sources have been migrated, the developer only needs to pick an OpenMP 5.0 compliant compiler and compile the application for the architecture of their interest.

In the subsequent sections we cover two successful migrations stories: POT3D and GEM. Later, we conclude this paper with some remarks.

Application Migration Examples

We have selected two applications to demonstrate the value of the migration tool. The applications are POT3D and GEM, and the reader will find some details in **Table 1**. These applications have been migrated and later compiled with the compilers and libraries found in Intel® HPC Toolkit 2024.0.1. Both applications have been executed on a single node containing 2S Intel® Xeon® Platinum 8480+ and four Intel® Data Center GPU Max 1550s. Each GPU consists of two compute tiles, for a total of eight compute tiles in the node. We validated the execution results against the reference output provided in the corresponding repository.

			OpenACC* constructs / clauses				
Application	**Language**	**# lines**	*# compute*	*# data*	*# atomic*	*# async*	*# API calls*
POT3D	Fortran	~11k	33	33	2	23	0
GEM	Fortran	~18k	34	69	118	0	1

Table 1. Application characteristics

Note**:** Compute constructs covered: kernels, loop, parallel and loop. Data constructs covered: enter/exit data, host_data, and update. Atomic constructs covered: atomic. Async constructs covered: async and wait.

POT3D

POT3D (commit-id: 5e8ee69f92860a372d5746462199ddfa702bbac2) is a Fortran code that computes potential field solutions to approximate the solar coronal magnetic field using observed photospheric magnetic fields as a boundary condition. The code is parallelized using MPI but has two alternatives when it comes down to GPU acceleration: OpenACC and the Fortran standard `do concurrent` construct. We focus on the OpenACC version in this article.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

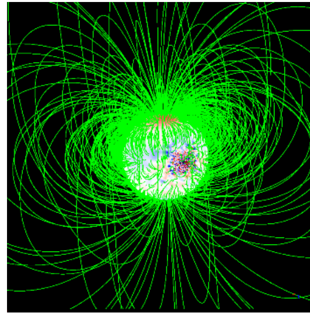

Figure 1. POT3D tracing magnetic field lines through the potential field solution, with the surface radial magnetic field shown as the red-blue colormap.

The code was translated using the following command:

```
${PATH_TO_TRANSLATOR}/intel-application-migration-tool-for-openacc-to-openmp src -async=none
```

Where `src` refers to the source directory of the application, and `-async=none` translates the code by ignoring asynchronous OpenACC statements (including `wait` and `async`) that are present in the source code. OpenMP does not provide an exact mapping for these statements. Fortunately, according to the developers, these asynchronisms provide little benefit to this application.

Some Notes on the Migration

Among the comments and warnings emitted in the migration report, we want to highlight that the application uses `!$acc set device_num(iprocsh)` to specify which OpenACC device to use. OpenMP offers an API call (`omp_set_default_device`) and an environment variable (`OMP_DEFAULT_DEVICE`) that mimic this behavior. Intel® MPI also offers an environment variable to pin MPI ranks with GPU devices (or tiles) (`I_MPI_OFFLOAD_PIN`). Consequently, we have ignored the translation of the construct and relied on the Intel MPI infrastructure when it comes to MPI rank – GPU tile binding.

When it comes to the translation, we observe that many data allocations are annotated with `!$acc data create`, which allocates the data counterpart in the accelerator address space, and that is translated using `!$omp target enter data`. We also notice that the most time-consuming region (i.e., `cgsolve`) contains `!$acc parallel loop` clauses that have been converted into `!$omp target teams loop`. Interestingly, the original constructs explicitly use the `present(X)` or `default(present)` clauses to let the offload runtime to check for the presence of the given variables in the accelerator address space. Those clauses are translated to `map` with present map-type-modifier or `defaultmap` clauses, respectively. There are also clauses such as `!$acc update` in the boundary exchanges or before/after doing I/O, among others, in order to synchronize contents of the host and accelerator address spaces.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

The corresponding Makefile was manually tuned to use the MPI compiler wrapper for IFX (i.e., `mpiifx`) and added the necessary flags to process the OpenMP offload constructs (i.e., `-fiopenmp -fopenmp-targets=spir64`) and the HDF5 dependencies.

With respect to the application execution, we used the small test that comes with the application repository, and we invoked the binary (through `mpirun`) while setting the following environment variables `I_MPI_ADJUST_BARRIER=1 I_MPI_OFFLOAD_PIN=1 I_MPI_OFFLOAD_PRINT_TOPOLOGY=1 I_MPI_DEBUG=3`. The timings achieved are shown in **Figure 2**. The results show that the application exposes a close-to-ideal scalability up to four MPI ranks (using one compute tile on each GPU) but efficiency drops when moving to eight MPI ranks (using two compute tiles on each GPU). We have exchanged some ideas with the application developers to study the rationale of this performance drop.

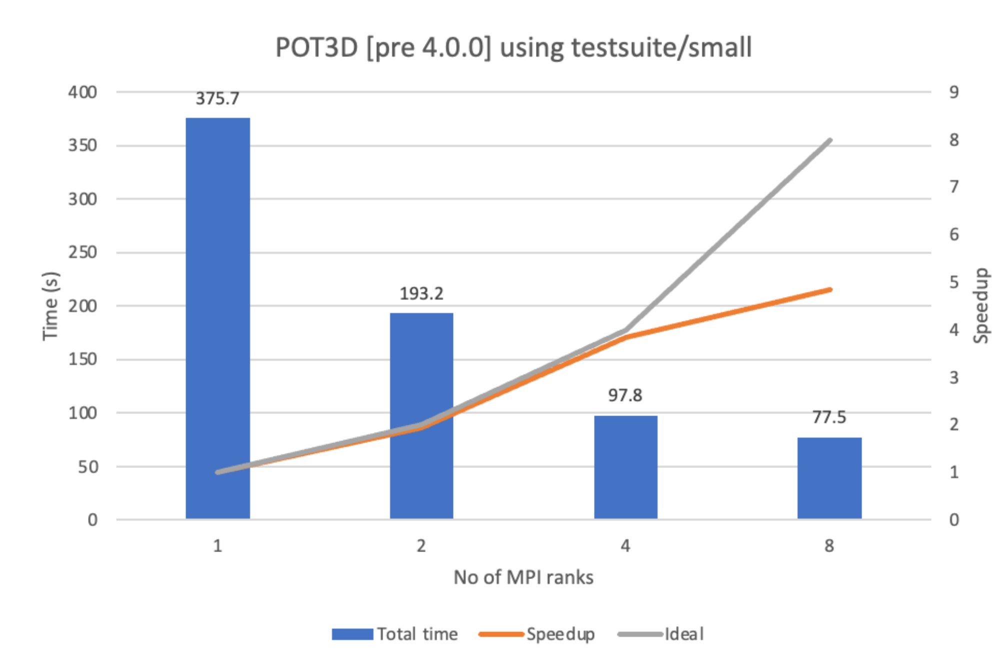

Figure 2. Performance achieved on POT3D when executed on the target system

GEM

GEM is an open-source particle-in-cell (PIC) simulation code considering physical field effect written by Fortran since 1990s. It aims to study transport in magnetically confined fusion devices, such as ITER.

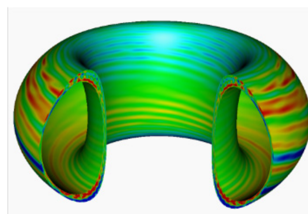

Figure 3. GEM simulates the plasma particles moving in a torus shape in a magnetic equilibrium.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

There are two time-consuming tasks in GEM:

1. Nested loops with/without reduction
2. Matrix solvers of the physical field

Both are subject to be accelerated by means of GPUs, but the code uses OpenACC to accelerate only the nested loops because developers prefer to use the standard LAPACK to implement the solvers for code maintainability. Consequently, we rely on the migration tool to convert the OpenACC constructs into OpenMP, and then we manually modify the translated code to use the oneAPI Math Kernel Library (oneMKL), which provides a parallel and accelerated LAPACK implementation, through the `!$omp dispatch` construct. The latter translation is not within the context of this article.

The file structure of the application is simple: All source files are contained in the same folder. The command to translate the code is as follows:

```
lTRANSLATOR}/intel-application-migration-tool-for-openacc-to-openmp *.f90
```

After the translation, we have replaced the OpenACC compiler flags by `-DOPENACC2OPENMP_ORIGINAL_OPENMP -fiopenmp -fopenmp-targets=spir64` to enable OpenMP offload capability on the IFX Fortran compiler. We added the `-DOPENACC2OPENMP_ORIGINAL_OPENMP` compiler flag because the original code also applies OpenMP constructs launching multiple threads when handling some loops too small for OpenACC to send to GPU, and these are kept by the migration tool but protected with this preprocessor variable.

Some Notes on the Migration

We would like to highlight idiosyncrasies we found during the migration:

1. On the data management
2. When expressing parallelism on loop constructs

Data Management

With respect to moving data around GPU and CPU, GEM uses an unstructured data approach, which provides the flexibility to establish a data mapping between the host and the device immediately after the code allocates (or deallocates) the data on host memory (i.e., using the `!$acc enter data create(X)` construct at data allocation time). The construct is translated by the migration tool as `!$omp target enter data(X).`

Then, when other application modules need to access the device data, they rely on the `present(X)` clause from OpenACC. This clause is converted into a map-type modifier (i.e., `map(present,alloc:X)`). Even though they do not look the same, the two express the same semantics; but in OpenMP, the presence check needs to go with a map-type (`alloc` in this case), even though the allocation is not performed.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

In addition, the application code relies on the `acc_is_present(X)` API to identify potential cases where data is not present in the GPU address space. This API routine is not translated by the migration tool, and the tool generates a warning about this mistranslation.

Parallelism Through Loops Constructs

We identify two major parallel regions accelerated by the means of `!$acc loop`. The first implements some reductions using the `!$acc atomic` constructs. These have been translated to their OpenMP counterparts. The second region is represented by the code below. Note that there is a `control-flow (if)` structure inside. This loop contains a nested loop construct but, given that this is not a perfectly nested loop, one cannot use the collapse clause.

```
!$acc parallel present(xp,s_buf,ipsend,s_displ,s_counts)
!$omp target teams map(present,alloc:xp,s_buf,ipsend,s_displ,s_counts)
!$acc loop independent  private(i,isrt,iend,isbuf)
!$omp loop order(concurrent) private(i,isrt,iend,isbuf)
    DO i=0,nvp-1
       IF( s_counts(i) .GT. 0 ) THEN
          isrt = s_displ(i)+1
          iend = s_displ(i)+s_counts(i)
          !$acc loop independent
          !$omp loop order(concurrent)
          DO isbuf=isrt,iend
             s_buf(isbuf) = xp(ipsend(isbuf))
          END DO
          !$acc end loop
          !$omp end loop
       END IF
    END DO
!$omp end loop
!$acc end parallel
!$omp end target teams
```

The initial OpenMP implementation resulting from the migration can compile and execute with several minor manual modifications. The performance observed when run with 4 MPI ranks using 1 GPU tile per rank is shown in **Figure 4**, and it is aligned with developer expectations.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

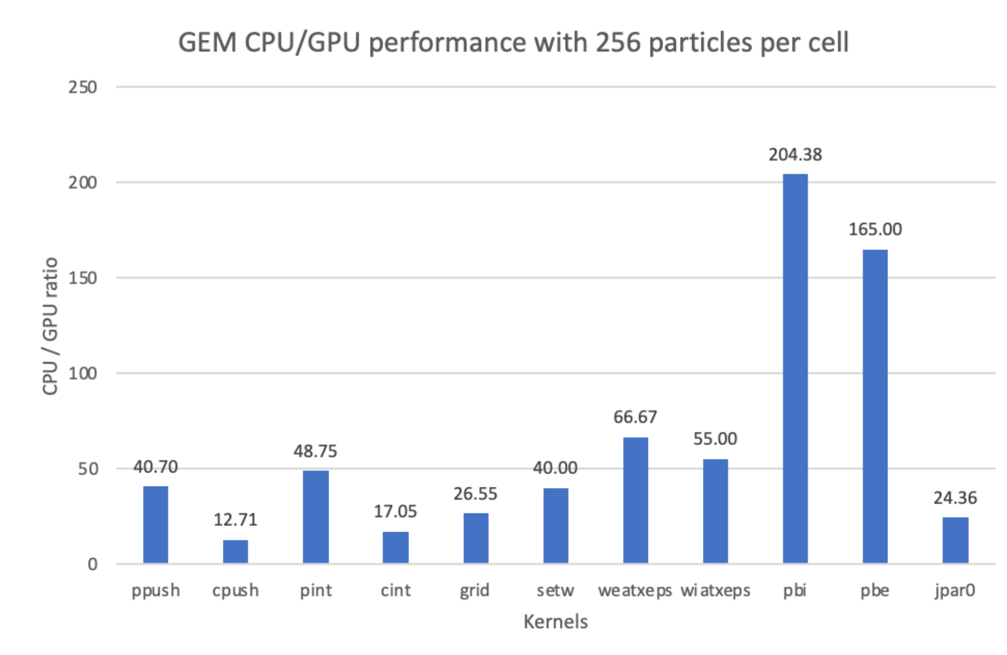

Figure 4. CPU-to-GPU ratio observed in the GEM application for the meaningful offloaded kernels

Concluding Remarks

We have shown that the Intel Application Migration Tool for OpenACC to OpenMP API has helped in porting OpenACC codes to OpenMP, allowing a user to run these codes on a variety of hardware/software if they support OpenMP 5.0. Even though the tool cannot provide a perfect translation for every OpenACC application, due to the implicit differences between the two programming models, it provides a particularly good starting point for this task. As of today, GEM developers have created a repository fork for the OpenACC translated code. We are still in discussions with the POT3D developers about adding the changes to their repository at the time of publishing this article.

In terms of potential future functionality, we are adding more translations of the OpenACC API calls to their OpenMP counterpart (if they exist) based on user feedback. We are also working towards offering a mapping alternative to the dissimilar async mechanisms found in OpenACC and OpenMP. Finally, there have been some efforts to identify CPU OpenMP directives in the original codes, and we could consider using OpenMP metadirectives for a translation supporting both CPUs and GPUs. Performance-wise, the tool cannot ensure the most performant translation due to the variety of hardware and software available, so we strongly recommend using performance tools to tune the source code.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

Sign up for future issues

Efficient Natural Language Embedding Models with Intel® Extension for Transformers

Making Retrieval-Augmented Generation More Efficient

Yuwen Zhou, Zhenzhong Xu, Xin He, Bo Dong, Wenxin Zhang, and Haihao Shen, Intel Corporation

Introduction to Natural Language Embeddings

Natural language embeddings are essential for natural language processing (NLP). They represent text as vectors that capture semantic information. These embeddings are crucial for various downstream NLP tasks, as they provide numerical input necessary for computational operations.

Among different embedding models, BGE (BAAI General Embedding) stands out for its efficiency. With its small and base versions, it strikes a balance between speed and effectiveness, making it an ideal choice for text embedding. Beyond text embedding, BGE integrates seamlessly with vector databases, further expanding its potential.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

We present an innovative approach to enhance the performance of embedding models that leverages Intel® Extension for Transformers, an open-source tool to significantly improve the speed and accuracy of BGE small models.

INT8 Static Post-Training Quantization

The static post-training quantization (PTQ) is an effective approach to quantize additional training steps. It requires calibration using a representative dataset to determine the quantization parameters (e.g., scale, zero point) of the model. We apply PTQ with automatic accuracy-aware tuning on the bge-small-en-v1.5 to produce an optimal quantized model. The code snippets below show how to leverage post-training quantization to optimize the BGE-small model. The CQADupStack dataset is used for calibration, and the MTEB STS task is used as an evaluation benchmark. The complete code is available here. (See readme for documentation.)

```
from intel_extension_for_transformers.transformers import metrics, objectives, QuantizationConfig
from intel_extension_for_transformers.transformers.trainer import NLPTrainer
# Replace transformers.Trainer with NLPTrainer
# trainer = transformers.Trainer(......)
trainer = NLPTrainer(......)
metric = metrics.Metric(
    name="eval_accuracy", is_relative=True, criterion=0.01
)
objective = objectives.performance
q_config = QuantizationConfig(
    approach="PostTrainingStatic",
    metrics=[metric],
    objectives=[objective]
)
model = trainer.quantize(quant_config=q_config, eval_func=mteb_sts_eval)
```

Unlock Faster NLP Inference with BGE

We now introduce a high-performance NLP backend designed to accelerate the inference of these BGE models without compromising accuracy. We leverage Apple Neural Engine* (ANE), a lightweight, bare-metal inference backend, to unlock the performance of compressed NLP models. It leverages both hardware and software optimizations to maximize performance.

To streamline the process for developers, Intel Extension for Transformers extends Hugging Face's familiar transformer APIs with easy-to-use model compression tools. This seamless integration allows users to leverage ANE's capabilities and optimize NLP models for faster inference, boosting productivity. The following example shows how to get started:

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

```
from transformers import AutoTokenizer
from intel_extension_for_transformers.transformers import AutoModel

sentences_batch = ['sentence-1', 'sentence-2', 'sentence-3', 'sentence-4']
tokenizer = AutoTokenizer.from_pretrained('BAAI/bge-small-en-v1.5')
encoded_input = tokenizer(sentences_batch,
                          padding=True,
                          truncation=True,
                          max_length=512,
                          return_tensors="np")

engine_input = [encoded_input['input_ids'], encoded_input['token_type_ids'], encoded_input['attention_mask']]

model = AutoModel.from_pretrained('./model_and_tokenizer/int8-model.onnx', use_embedding_runtime=True)
sentence_embeddings = model.generate(engine_input)['last_hidden_state:0']
print("Sentence embeddings:", sentence_embeddings)
```

Measuring Performance

We measured the optimal quantized BGE models on MTEB STS. The accuracy relative loss for all models is within 1% (**Table 1**):

We also measured embedding latency, which is the average milliseconds to encode one sentence using one socket, 24 cores/instance, one instance with sequence length = 512 and batch size = 1 **(Figure 1)**:

Model	MTEB STS average (10 datasets)	
	PyTorch FP32	ITREX INT8
BAAI/bge-small-en-v1.5	81.59	81.43
BAAI/bge-base-en-v1.5	82.39	82.19
BAAI/bge-large-en-v1.5	83.11	82.82

Table 1. FP32 and INT8 accuracy of embedding models

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

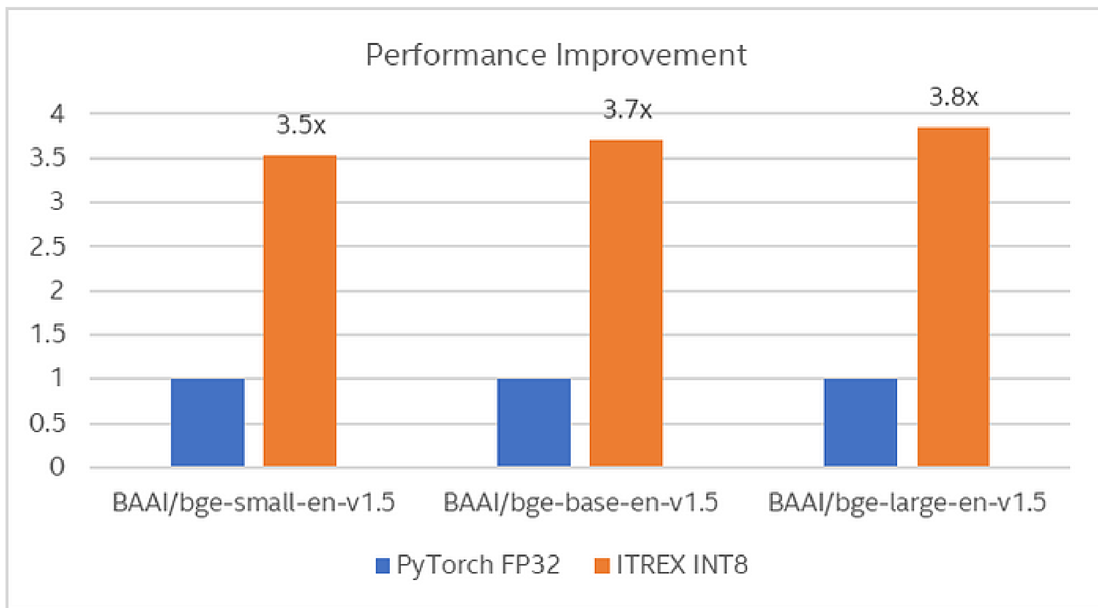

Figure 1. Performance improvement of INT8 embedding models

Hardware configuration: Intel® Xeon® Platinum 8480+ processor, two sockets with 56 cores per socket, 2048 GB RAM (16 slots/128 GB/4800 MHz), HT:on. OS: Ubuntu* 22.04.2LTS; **Software configuration:** Python* 3.9, NumPy 1.26.3, ONNX Runtime 1.13.1, ONNX 1.13.1, Torch 2.1.0+cpu, Transformers 4.36.2. **Testing date:** 01/26/2024.

Building a Chatbot Using the Optimized BGE Model

We have extended the LangChain embedding API in Intel Extension for Transformers, enabling users to load quantized BGE models as shown below:

```
from intel_extension_for_transformers.langchain.embeddings import HuggingFaceBgeEmbeddings
embed_model = HuggingFaceBgeEmbeddings(model_name="/path/to/quantized/bge/model"
```

Furthermore, we have introduced a customizable chatbot framework called NeuralChat, which is part of Intel Extension for Transformers. This framework allows us to quickly build a chatbot on multiple architectures (e.g., Intel® Xeon® Scalable processors and Intel® Gaudi® AI accelerators). Here is an example showing how to use the quantized BGE small model to develop a chatbot application for knowledge retrieval:

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

```
from intel_extension_for_transformers.neural_chat import plugins
from intel_extension_for_transformers.neural_chat import build_chatbot
from intel_extension_for_transformers.neural_chat import PipelineConfig

plugins.retrieval.enable = True
plugins.retrieval.args["input_path"] = "/path/to/docs"
plugins.retrieval.args["embedding_model"] = "/path/to/quantized/bge/model"
pipeline_config = PipelineConfig(model_name_or_path="facebook/opt-125m", plugins=plugins)
chatbot = build_chatbot(pipeline_config)
response = chatbot.predict(query="What is Intel extension for transformers?")
```

Concluding Remarks

We have demonstrated the effectiveness of quantization and optimization of embedding models using Intel Extension for Transformers to achieve better performance. We plan to explore other model compression techniques (e.g., pruning) to further improve inference efficiency without sacrificing quality. We encourage you to give it a try and explore other Intel® AI tools. Star the Intel Extension for Transformers repository to receive notifications about our latest optimizations. You are also welcome to create pull requests or submit issues to the repository. Feel free to contact us if you have any questions.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

Sign up for future issues

intel. software

That Look When ...

Your Code is Crushing it

Power your code's performance with a single, open programming model that supports multiple languages to deliver heterogeneous computing performance. Securely established in open standards, oneAPI offers cross-architecture libraries, compilers, and tools that open your code to more hardware choices—for exceptional performance.

1™ oneAPI

A new era of accelerating computing across Enterprise, Cloud, HPC, & AI, powered by oneAPI.

Explore & Download for Free

Learn SYCL* in an Hour (Maybe Less)

James Reinders, Intel Engineer

In this piece, I will introduce all the key things to know to program in C++ with SYCL*.

I obviously will cover only the bare essentials; therefore, I'm not teaching everything that you would find in a 500-page book on SYCL. If you tell me that surprises you, I will assume you are joking.

We will learn the basics (1-2-3) and have a small working program that we can use to explore more as we wish.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. *SYCL and the SYCL logo are trademarks of the Khronos® Group. *Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

What Is SYCL?

C++ with SYCL offers us the ability to use accelerators from a C++ program, regardless of vendor (e.g., NVIDIA, AMD, Intel, etc.) or architecture (e.g., GPU, CPU, FPGA, DSP, etc.). To do so, we just need a C++ compiler that supports SYCL and a runtime that supports our accelerator (most often a driver, such as an OpenCL* runtime from the vendor). SYCL 2020 (the current standard) allows for support beyond just OpenCL. This freedom has allowed SYCL implementations to find the best path to hardware in numerous ways, including PTX for NVIDIA, ROCm/HIP for AMD, OpenCL for many vendors, SPIR-V* for Intel, OpenMP* for multiple vendors, etc. The Intel® compiler uses some of these paths. A SYCL compiler project (AdaptiveCpp), led by Heidelberg University, is well known for pioneering a number of innovative paths as well. There are many options for getting SYCL support for various accelerators.

SYCL Is C++

SYCL is designed for C++ and will feel very comfortable to C++ programmers. Nevertheless, SYCL can be learned and used with minimal C++ knowledge.

Follow Along with Instructions at the QR code

In order to follow along and get the most out of the experience, we have instructions under "Learn SYCL in an Hour (Maybe Less)" at tinyurl.com/learnSYCLnow. (The QR code goes here, too.) The instructions tell you how to access Intel® Developer Cloud freely to use a system already configured with multiple GPUs and the installed software we need. It has the information on where to fetch the code examples from GitHub*.

There Are Three Keys to SYCL

SYCL solves three problems for us by providing these capabilities:

1. Find what accelerator(s) are available at runtime.
2. Share data with accelerator(s).
3. Offload computational work to accelerator(s).

I will also mention that SYCL has many additional features that we will want and appreciate, including support for C++ error handling even for offload computations, and built-in support for reduction operations. However, these are things we can learn later, as needed after we solidly understand the three keys.

Finding/Choosing Accelerator(s)

Our goal in finding/choosing an accelerator is to get a connection to an accelerator so we can move on to sharing data and offloading code. In SYCL terms, this means getting a queue.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

A queue connects us to a specific accelerator. We can create as many queues as we like, and different queues may point to the same accelerator if we like. In the example program, I show off and try to fill an array of queues with handles to every accelerator on the machine. That might mean it only gets one, or it might be many. Following along with my instructions on the Intel Developer Cloud, you will probably get four (at least it will as I write this).

SYCL offers us a lot of control to find and select from the accelerator(s) that are available at runtime. We should start off simple and simply say:

```
sycl::queue q;
```

That will give us the handle `q`, which we happily use for all our data sharing and offloading needs. In this simple case, the SYCL runtime will simply pick an accelerator for us (hopefully the best one available at runtime).

Of course, I would normally get rid of the `sycl:: by using a namespace sycl` early on. However, for examples, I leave them explicit to help highlight where we are using SYCL while we are learning.

It is important to note that SYCL will always have a device available. This is incredibly useful for writing a simple program that will always work. In a system with no accelerators, the host (a CPU in all the implementations that I have seen) will be used.

If we want to know what device we connected to, we can simply print the name out:

```
std::cout << "Running on "
          << q.get_device().get_info<sycl::info::device::name>();
```

Sharing Data with Accelerator(s)

Sharing data is easy with SYCL. We can use USM (Unified Shared Memory) with memory allocation that look much like mallocs, and memory allocated in that way is magically shared between the host and accelerators. This is usually only supported with accelerators that support USM in hardware. That is not much of an issue, as all modern GPUs, CPUs, and FPGAs can support USM. SYCL also supports explicit buffers that are also magically shared between the host and accelerators, but without allowing regular pointers to work across hosts and accelerators. For now, I will just recommend using USM unless you know you want to use buffers.

In the example, the program uses buffers for the job that computes digits of pi.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

That code looks like this:

```
std::array<int, 200> d4;
sycl::buffer outD4(d4);  // this is a sycl buffer
```

If you want to convert to use USM, do this:
Comment out the two lines shown before for `d4`, and use this:

```
// this is a sycl USM memory allocation
auto d4 = (int *)sycl::malloc_shared( sizeof(int)*200, myQueue2 );
```

And get rid of the accessors (`outAccessor` and `myD4`) since USM can just use pointers. You can replace the declarations with these macros instead (being lazy and not wanting to edit the actual usages):

```
#define outAccessor d4
#define myD4 d4
```

Offloading Computational Works to Accelerator(s)

We can simply write some code and say, "Run this code on the accelerator." This is a simple Hello World! that runs on an accelerator.

```
q.submit([&](sycl::handler& cg) {|
           auto os = sycl::stream{128, 128, cg};
           cg.single_task(
                            [=]() { os << "Hello World!\n"; });
         });
```

The *submit* says we have work to offload. The `single_task` is used to specify a single thing to run; in this case, a print of `Hello World!` The `single_task` can specify a function to offload, but often we just specify it inline using a C++ lambda function (as I have done in this case).

Since accelerators are used to gain performance through parallelization, we need something a little more involved if we want to do anything for higher performance. This is when we need to know that SYCL emphasizes a style of programming that invokes a kernel in parallel. This is the same programming style you find in CUDA and OpenCL. The idea is simple: We write a kernel that can be simple serial code to operate on one piece of data, and then we invoke it in parallel such that a kernel is invoked on each data element independently.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

In the sample code, we actually do a blur in parallel. Understanding that code is not particularly hard, and once we understand it SYCL really starts to make sense. Here is a simpler look at going parallel by simply making `Hello World!` run in parallel:

```
// this is the entire program

#include <sycl/sycl.hpp>
int main(int argc, char* argv[]) {
  sycl::queue q;
  std::cout << "Running on "
            << q.get_device().get_info<sycl::info::device::name>()
            << "\n";
  q.submit([&](sycl::handler& cg) {
             auto os = sycl::stream{1024, 1024, cg};
             cg.parallel_for(10, [=](sycl::id<1> myid)
                 {
                   os << "Hello World! My ID is " << myid << "\n";
                 });

         });
}
```

Running on my laptop, under WSL, it printed this:

```
Running on Intel(R) Core(TM) i7-8665U CPU @ 1.90GHz
Hello World! My ID is {5}
Hello World! My ID is {0}
Hello World! My ID is {7}
Hello World! My ID is {2}
Hello World! My ID is {1}
Hello World! My ID is {8}
Hello World! My ID is {6}
Hello World! My ID is {9}
Hello World! My ID is {3}
Hello World! My ID is {4}
```

The Example Program

Now you know plenty to go off and explore. Keep in mind that a queue connects to an accelerator, and uses of the queue will then be either to set up data sharing or to offload computations. That should be enough to puzzle out the sample program and start changing it to experiment and learn more.

The sample program that I provided online is my own mess. It is intended only to show off the basics with absolutely no regard for writing an effective parallel program. I think all the other learning resources for

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

SYCL in the world try to show well-considered parallel programming examples. I wanted to be different and inspire fun exploring to get started.

The sample program does three different jobs. Each job will run on one of three different accelerators when available, or they will happily run all on the host if that is all we have, or whatever mix of accelerators we find at runtime.

I pulled together this code so that I could show a variety of things that are easy to change and understand. I believe that has high value when starting out. You will even discover that part of the code has `if-defs` to change between using buffers and USM for sharing for one of the examples if you wander over to the Exercise_02_... subdirectory.

Two notable extras in the code that I tossed in to encourage more playing while learning valuable techniques:

- I create an array of queues and load them up with all the accelerators that I can find. Then I take the first, second, or third accelerators for the three different jobs (modulo the actual number of accelerators we found at runtime). Do not be thrown by the seeming complexity of this silly code: It is not doing much of anything different than simply creating a queue with `sycl::queue q;`. It does hint at more fun you can have in selecting your favorite accelerators at run time, and possible matching them to custom algorithms if you like. So much fun available — but not necessary to learn SYCL.
- I set up the queues with profiling selected. (I assume it is available, which it usually is, but this limits where we can run the program unless we spend a little more effort and add logic to only use profiling on devices that support it.) The profiling option lets us gather information about the actual run times on the accelerator. This is more valuable than wall-clock time when tuning our kernels because it helps reduce noise caused by other code if all we are doing is tuning the kernel itself. Together, kernel timing and wall-clock timing give us great data while we tune an application.

Once you are comfortable with SYCL, we can learn much more from the resources listed at sycl.tech, including a book that I co-authored, some examples and tutorials, and much more.

Summary

We learned two key things: (1) SYCL fundamentally addresses three things for using accelerators from C++, and (2) we have a silly little program we can get lost in for hours changing and experimenting with as our first SYCL program.

I believe it Is highly valuable to know as many programming models/languages as we can so we can use the appropriate tool for the job. If our job is writing C++ that uses accelerators, and the kernel style of expressing parallelism makes sense for our algorithm, and we want our application to be highly portable across vendors and/or architectures, then knowing SYCL could well be valuable.

Learning SYCL is not hard at all. Mastering effective parallel programming… Well, that is a whole different matter. ☺️

Happy coding!

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

Our Intel® C++ Compiler Is the First to Earn Khronos SYCL* 2020 Conformance

Greg Lueck, Intel Principal Engineer with the Intel® C++ Compiler Team

Our Intel® C++ Compiler Is the First to Earn Khronos SYCL* 2020 Conformance

It is no secret that GPU accelerators have become an important solution for parallel workloads ranging from AI to HPC to image processing. Until recently, the programming model for these platforms has been dominated by proprietary languages that lock application developers to a single vendor's hardware. The Khronos Group has been solving this problem, though, by defining SYCL*, a new multi-vendor open standard for programming GPUs and other offload accelerators. SYCL is based on modern C++ programming features, and it provides an API that can target any vendor's GPU devices or even other accelerator devices, like field-programmable gate arrays (FPGAs).

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

The Khronos Group ratified the SYCL 2020 specification early in 2021, and vendors have been working hard to achieve full conformance to the specification since then. Intel Corporation recently became the first vendor to do this with the 2024.1 release of the Intel® oneAPI DPC++/C++ Compiler.

The Khronos Group does not just take a vendor's word when they claim conformance. In addition to publishing a specification for SYCL, the Khronos Group also provides a comprehensive SYCL conformance test suite. To claim conformance, a vendor must first demonstrate that its compiler passes this test suite. This level of rigor is good for the industry because it helps ensure that application code is portable from one vendor's compiler to another.

Applications written in C++ with SYCL are also portable across different vendors' hardware, but the responsibility is shared between the compiler and the application developer. A fundamental challenge is that GPU features differ from one vendor to another, and even more so from one type of accelerator (e.g., GPU) to another (e.g., FPGA). History has shown that hiding these differences often leads to code that performs poorly or code that runs with vastly different performance on different vendors' hardware.

SYCL takes a different approach and exposes these differences to the programmer. Application developers can then decide whether portability across vendor hardware is important. If a developer only cares to run an application on a single vendor's hardware, they can simply use the features that are supported on that hardware. However, if a developer wants to write a portable application, they can use the rich query mechanisms in the SYCL language to conditionally make use of features that each device provides, eking out the last bit of performance on each hardware device. The power of SYCL is that it provides a common programming language that can be used across diverse types of accelerators while still exposing the features that are unique to each of those accelerators.

With Intel's announcement of the first conformant SYCL compiler, this portability is now available to developers. Application developers can make use of the full set of features in the SYCL 2020 specification, with the assurance that their code will be portable to any other compiler that is conformant to that specification.

In this world of accelerators, we do need to ask what SYCL compiler means in this context. First, it is a C++ compiler with SYCL support. C++ compilers might also support other parallelization or offloading with another standard such as OpenMP*. It is important to know we are programming in C++ with some added support to give us control over offloading data and computations to accelerators. Secondly, conformance is proven with a combination of a compiler and runtime(s). Therefore, the Khronos website lists the conformant products (link is given at the end of this article) in a precise enough fashion to see what has passed the rigors of submission, review, and acceptance.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

I will conclude with a list of some of the most valuable resources to learn more about SYCL and this first conformant SYCL compiler.

The specification for SYCL is available at khronos.org/sycl.

The conformant product list is available at khronos.org/conformance/adopters/conformant-products/sycl.

The Intel C++ compiler is available at intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html.

Great resources for learning SYCL are the book (eBook is free) from tinyurl.com/book-SYCL and the resources of sycl.tech.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

intel software

THE PARALLEL UNIVERSE

Intel technologies may require enabled hardware, software or service activation. Learn more at intel.com or from the OEM or retailer.

Your costs and results may vary.

Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors.

Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. See backup for configuration details. For more complete information about performance and benchmark results, visit www.intel.com/benchmarks.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details. No product or component can be absolutely secure.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

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.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. *Other names and brands may be claimed as the property of others.

Printed in USA 707/IH Please Recycle.