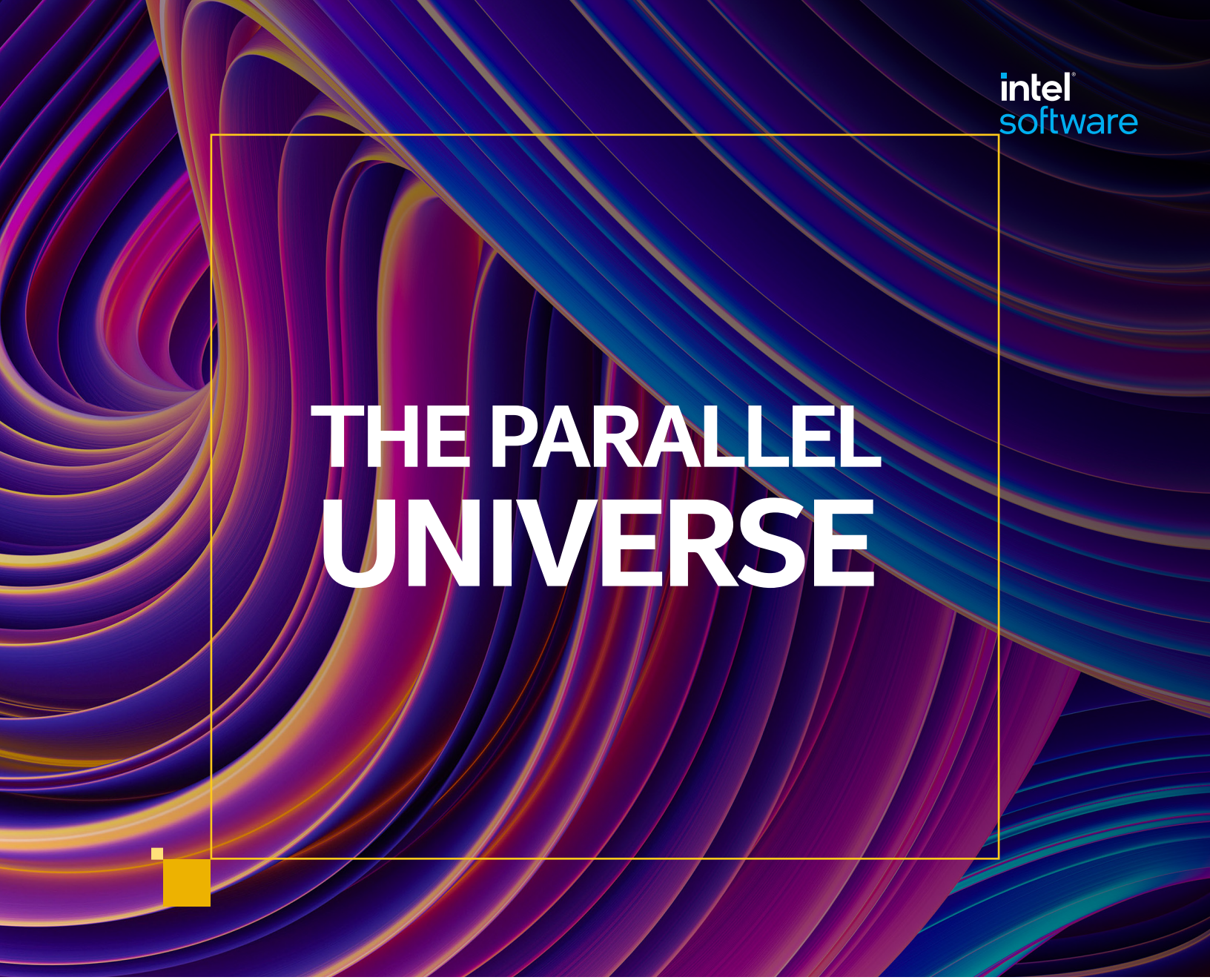


Vectorization in LLVM and GCC for Intel CPUs and GPUs

Efficient Heterogeneous Parallel Programming Using OpenMP

ArrayFire Interoperability with oneAPI, Libraries, and OpenCL

Issue
47
2022

Contents

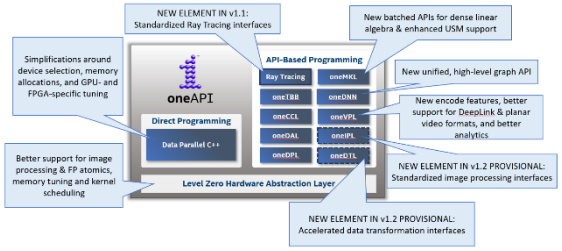
Sign up for future issues

Letter from the Editor

Henry A. Gabb, Senior Principal Engineer at Intel Corporation, is a longtime high-performance and parallel computing practitioner who has published numerous articles on parallel programming. He was editor/coauthor of "Developing Multithreaded Applications: A Platform Consistent Approach" and program manager of the Intel/Microsoft Universal Parallel Computing Research Centers.

oneAPI Continues Gaining Momentum

When I became editor of *The Parallel Universe*, I noted in my first letter to readers that I dreaded the heterogeneous parallel computing future. Five years later, I'm downright optimistic about it, mainly because of oneAPI™. Sanjiv Shah (Intel Vice President and General Manager of Developer Software) is similarly optimistic in his recent "Giving thanks for oneAPI progress" blog. The new oneAPI v1.1 specification is now live, and the provisional v1.2 specification adds a lot of new features. Eleven new oneAPI Centers of Excellence were launched around the world last year. And finally, oneAPI won yet another HPCwire Reader's Choice Award, this time for 2021 Best HPC Programming Tool or Technology. That's a lot of momentum going into 2022.



Our feature article in this issue, **Vectorization in LLVM and GCC for Intel CPUs and GPUs**, describes how single instruction, multiple data (SIMD) support is evolving in modern compilers. The authors show examples of automatic vectorization, programmer-guided vectorization, and a data parallel library approach.

Efficient Heterogeneous Parallel Programming Using OpenMP shows some best practices to keep both the CPU and GPU working at the same time. The authors provide advice and code examples to express true asynchronous, heterogeneous parallelism using standard OpenMP directives.

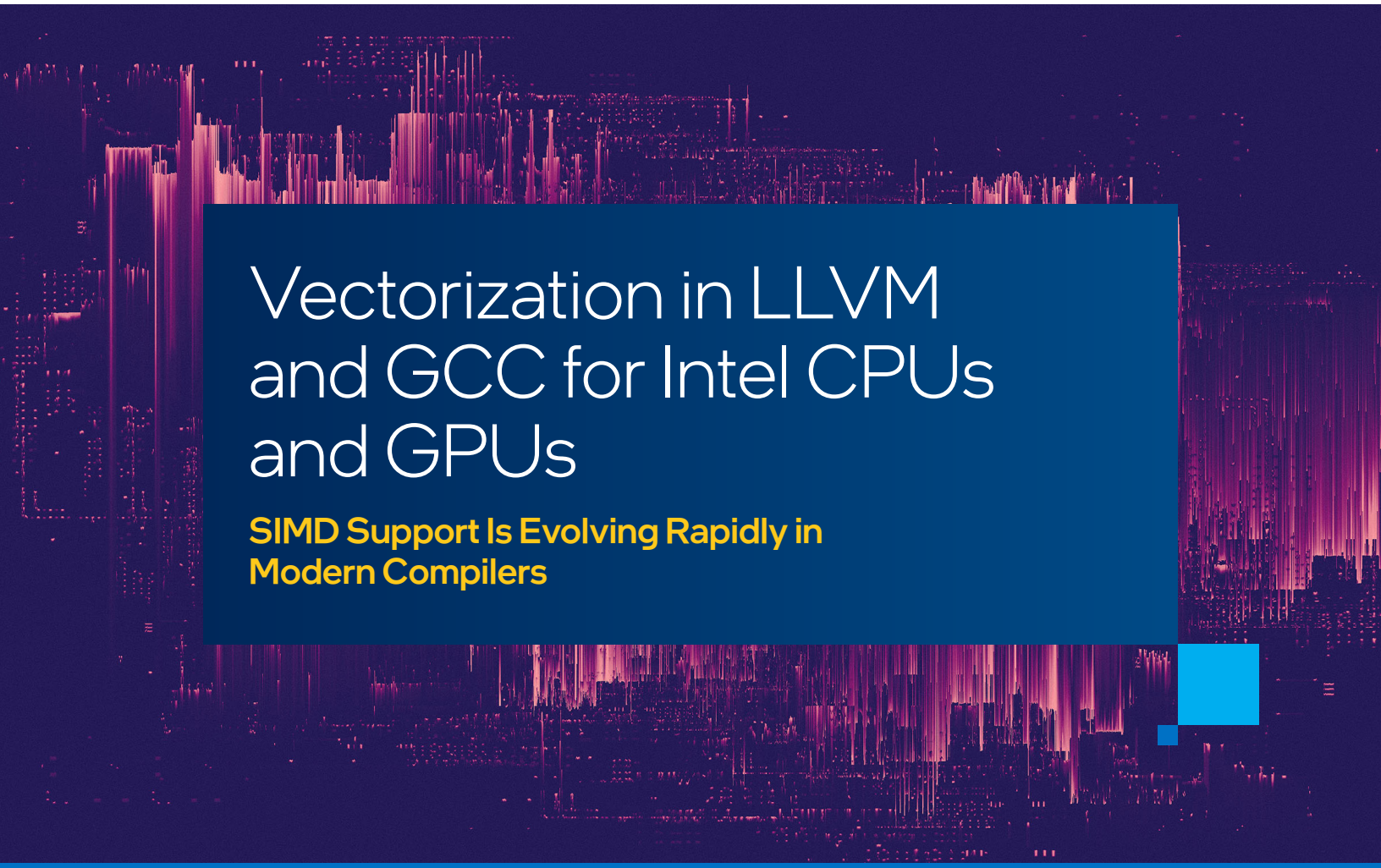
In the spectrum of separation of concerns, I'm more of a domain scientist than a performance tuning engineer, so I'm interested in programming abstractions that deliver performance while hiding hardware details. Lately, I've been experimenting with the ArrayFire heterogeneous parallel library. The results have been good in terms of productivity and performance. I plan to write an article about my experiments for a future issue, but in the meantime, Stefan Yurkevitch (ArrayFire, Software Engineer) discusses **ArrayFire Interoperability with oneAPI and OpenCL™** in this issue.

From high-level software abstractions for heterogeneous parallelism, we go lower in the stack to hardware abstractions with **Using the oneAPI Level Zero Interface**.

We close this issue with three data science articles. The first shows how to do efficient **Hyperparameter Optimization with SigOpt for MLPerf™ Training on Habana® Gaudi®** while also achieving better model accuracy. The next article describes how to **Scale Your Pandas Workflow with Modin** – no recoding necessary. The final article, **From Ray to Chronos**, shows how to build scalable, end-to-end AI workflows with BigDL on top of Ray.

As always, don't forget to check out Tech.Decoded for more information on Intel solutions for code modernization, visual computing, data center and cloud computing, data science, systems and IoT development, and heterogeneous parallel programming with oneAPI.

Henry A. Gabb
January 2021



Vectorization in LLVM and GCC for Intel CPUs and GPUs

SIMD Support Is Evolving Rapidly in Modern Compilers

Xinmin Tian, Senior Principal Engineer, Hideki Saito, Principal Engineer, Hongtao Liu, Compiler Engineer, James Reinders, oneAPI Evangelist and Editor Emeritus of The Parallel Universe, Intel Corporation

Modern CPU and GPU cores use single instruction, multiple data (SIMD) execution units to achieve higher performance and power efficiency. The underlying SIMD hardware is exposed via instructions such as SSE, AVX, AVX2, AVX-512, and those in the Intel® Xᵉ Architecture Gen12 ISA. While using these directly is an option, their low-level nature severely limits portability and proves unattractive for most projects.

To provide a more portable and easier to use interface for programmers, three avenues are explored in this article: auto-vectorization, programmer-guided SIMD vectorization through language constructs or programmer hints, and a SIMD data-parallel library approach. We provide an overview of these methods and show SIMD vectorization evolution in the LLVM and GCC compilers through code examples. We also examine a couple of vectorization techniques in the LLVM and GCC compilers to achieve optimal performance on Intel® Xeon® processors and Intel Xᵉ Architecture GPUs.

Enhancing LLVM and GCC

Our goal is to enhance vectorization of both the LLVM and GCC compilers, so contributing to open source has been a key design consideration. The VPlan vectorizer, and the related VectorABI, have been designed so they are applicable for integration into both LLVM and GCC [13] optimizers.

The framework for the VPlan vectorizer may be integrated into the LLVM trunk. Our VectorABI [12] is published and is being utilized by the LLVM and GCC communities for function vectorization. The VPlan vectorizer has started to surpass the results previously provided by the proprietary Intel compilers for Intel Xeon processors.

Utilizing SIMD

Modern CPUs support SIMD execution. SIMD is a hardware feature for a wavefront parallel execution of a single instruction over multiple data elements. It is useful for operating on multiple pieces of data at once given that their control flow is similar (minimal vector divergence) and the operation is not memory-bound. Unfortunately, writing a program that directly uses the SIMD ISA is not straightforward and has limited portability. We will discuss three approaches to improve this situation for programmers: auto-vectorization, programmer-guided SIMD vectorization through hints or language constructs, and using the C++ SIMD data-parallel library.

Auto-Vectorization

Automatically performing data- and control-dependency analysis and converting a scalar program to a corresponding vector form based on a built-in cost model is called auto-vectorization [4][5]. While the simplicity of this approach is attractive to programmers for its productivity and portability, auto-vectorization does not always produce optimal code because of compile-time unknowns like loop bounds and memory access patterns.

Programmer-Guided SIMD Vectorization

OpenMP (version 4.0 and later) includes SIMD constructs to support vector-level parallelism [7]. These constructs provide a standardized set of vector constructs so programmers no longer need to use non-portable, vendor-specific intrinsics or directives [6]. In addition, these constructs provide additional hints about the code structure to the compiler and allow for better vectorization that blends well with parallelization [5].

C++ SIMD Data-Parallel Library

There is an ISO C++ proposal for a data-parallel library [3]. Its intent is to support acceleration through data-parallel execution resources such as SIMD registers and instructions or execution units driven by a common instruction decoder. If such execution resources are unavailable, the interfaces support a

transparent fallback to sequential execution. A SIMD `memcpy` example using the C++ SIMD data-parallel library is shown in Figure 1. This example can be compiled to generate LLVM Vector IR and binary for core-avx512.

```
namespace stdsimd = std::experimental;

void simd_memcpy(
     stdsimd::native_simd<float> x,
     stdsimd::native_simd<float> y,
     void *p)
{
   auto cmp = x < y;
   memcpy(p, &cmp, cmp.size()*4);
}
```

```
define void @_Z11simd_memcpy_Pv(<16 x float> %x.coerce,
                                <16 x float> %y.coerce, i8*
                                nocapture %p)
{
entry:
  %0 = fcmp fast olt <16 x float> %x.coerce, %y.coerce
  %cmp.sroa.0.sroa.0.0.p.sroa_cast = bitcast i8* %p to <16 x i1>*
  store <16 x i1> %0, <16 x i1>* %cmp.sroa.0.sroa.0.0.p.sroa_cast
  ret void
}
```

Figure 1. An example of the C++ SIMD data-parallel library

The SIMD vectorization is critical to delivering optimal performance of compute-intensive workloads on modern CPUs and GPUs regardless of which vectorization method is used to produce SIMD code. In the next sections, we present recent LLVM SIMD vectorization advances for CPUs and GPUs with more code examples.

LLVM VPlan Vectorization

VPlan Vectorizer

Intel LLVM Compiler introduces a newly designed loop vectorizer aimed at matching or exceeding the capability and performance of the vectorizer in Intel Classic Compiler. The new vectorizer is often referred to as VPlan Vectorizer after the name of its major internal data structure, VPlan (vectorization plan), to distinguish it from the LLVM community Loop Vectorizer (a.k.a. LV). LORE and RAJAPerf experiments show that Intel LLVM Compiler can generate equivalent or better performing code than Intel Classic Compiler for a variety of computational kernels extracted from HPC applications [9]. At the time of writing, Intel LLVM Compiler enables VPlan Vectorizer for auto-vectorization at -O2 or higher optimization plus the -x (/Qx for Windows) target flag. Without the -x flag, the community Loop Vectorizer will be used. VPlan Vectorizer is enabled at -O0 or higher for OpenMP SIMD when Intel's OpenMP implementation is enabled with the -qopenmp (/Qopenmp for Windows) flag. At the time of writing, many of frequently used OpenMP 4.5 SIMD features are functional and performant. We continue our efforts to support the latest OpenMP 5.2 SIMD features.

Figure 2 shows how a simple outer loop (left column) is vectorized by Intel Classic Compiler (icc, center column) and Intel LLVM Compiler (icx, right column). Overall ASM code generated by Intel Classic Compiler looks more concise and easier to follow, but Intel LLVM Compiler generates noticeably better ASM code for the inner while-loop (basic block `.LBB0_7` for icx versus `..B1.7` for icc) due to its better handling of the inner loop execution condition in the %k1 mask register.

```
void foo(int N, float *a,
         float *b, float *c){
  #pragma omp simd
  for (int i=0;i<N;i++){
    float x = a[i];
    float y = b[i];
    while(x>y){
      x = x*x;
    }
    c[i] = x;
  }
}
```

```
icc -O2 -qopenmp-simd -xCORE-
AVX512 -c -S -unroll0

..B1.5:
 vmovups   (%rsi,%r8,4), %ymm1
 vmovups   (%rdx,%r8,4), %ymm0
 vcmpps    $14, %ymm0, %ymm1, %k1
 kortestw %k1, %k1
 je        ..B1.9
..B1.6:
 kmovw      %k1, %k0
..B1.7:
 kandw      %k0, %k1, %k2
 vmulps    %ymm1, %ymm1, %ymm1{%k2}
 vcmpps    $14, %ymm0, %ymm1, %k3
 kandw     %k3, %k2, %k4
 kandw     %k0, %k4, %k0
jne       ..B1.7
..B1.9:
 addl      $8, %r9d
 vmovups   %ymm1, (%rcx,%r8,4)
 addq      $8, %r8
 cmpl      %eax, %r9d
 jb        ..B1.5
```

```
icx -O2 -qopenmp-simd -xCORE-
AVX512 -c -S -unroll0

 jmp      .LBB0_4
.LBB0_5:
 vxorps   %xmm2, %xmm2, %xmm2
.LBB0_8:
 vcmpltps  %ymm0, %ymm1, %k1
 vmovaps %ymm2, %ymm0 {%k1}
 vmovups %ymm0, (%rcx,%rax,4)
 addq     $8, %rax
 cmpq     %rdi, %rax
 jae      .LBB0_9
.LBB0_4:
 vmovups (%rsi,%rax,4), %ymm0
 vmovups (%rdx,%rax,4), %ymm1
 vcmpltps  %ymm0, %ymm1, %k0
 kortestb  %k0, %k0
 je       .LBB0_5
# %bb.6:
 vmovaps %ymm0, %ymm3
 kmovq    %k0, %k1
.LBB0_7:
 vmulps   %ymm3, %ymm3, %ymm3
 vmovaps %ymm3, %ymm2 {%k1}
 vcmpltps  %ymm3, %ymm1, %k1 {%k1}
 ktestb   %k0, %k1
 jne      .LBB0_7
 jmp      .LBB0_8
```

Figure 2. Outer loop vectorization using VPlan Vectorizer

Kernel and Function Vectorization

Intel LLVM Compiler implements DPC++/OpenCL kernel vectorization and OpenMP function vectorization through VPlan vectorizer [5][10]. This is accomplished by converting a function vectorization problem into a loop vectorization problem. Customers can expect that most of the optimizations implemented for vectorizing loops are also available to vectorizing kernels/functions.

Figure 3 is an equivalent vectorization expressed in OpenMP declare SIMD directive form [7]. An 8-way non-mask vectorized AVX-512 vector variant function (`_ZGVcN8luuu_bar`) is shown. Even though the basic block layout is different, and the outer loop control flow is naturally absent because the compiler knows it is vectorizing for "8-instances" of the function `bar`, the rest of the ASM code is strikingly similar to icx-generated ASM code in the loop vectorization example (Figure 2) because it is vectorized by the same VPlan Vectorizer by letting the compiler inject an 8-iteration loop around the function body.

```
#pragma omp declare simd \
         linear(i) uniform(a,b,c)
void bar(int i, float *a, float *b, float *c){
  float x = a[i];
  float y = b[i];
  while(x>y){
    x = x*x;
  }
  c[i] = x;
}
```

```
icx -O2 -qopenmp-simd -xCORE-AVX512 -c -S -unroll0

_ZGVcN8luuu_bar:
 movslq   %edi, %rax
 vmovups  (%rsi,%rax,4), %ymm0
 vmovups  (%rdx,%rax,4), %ymm1
 vcmpltps %ymm0, %ymm1, %k1
 kortestb %k1, %k1
 je       .LBB3_1
# %bb.2:
 vcmpltps %ymm0, %ymm1, %k0
 vmovaps  %ymm0, %ymm3
.LBB3_3:
 vmulps   %ymm3, %ymm3, %ymm3
 vmovaps  %ymm3, %ymm2 {%k1}
 vcmpltps %ymm3, %ymm1, %k1 {%k1}
 ktestb   %k0, %k1
 jne      .LBB3_3
 jmp      .LBB3_4
.LBB3_1:
 vxorps   %xmm2, %xmm2, %xmm2
.LBB3_4:
 vcmpltps %ymm0, %ymm1, %k1
 vmovaps  %ymm2, %ymm0 {%k1}
 vmovups  %ymm0, (%rcx,%rax,4)
 vzeroupper
 retq
```

Figure 3. Function vectorization example using VPlan Vectorizer

New ISA Support

One of the benefits of implementing a vectorizer on the LLVM compiler framework is first-class support of vector data types. When AVX-512-FP16 [11] was introduced, the vectorizer was able to take advantages of it as soon as the ASM/OBJ code generation support was added, giving vectorizer developers a pleasant surprise. Figure 4 is a simple FP16 vectorization example.

```
void foo(int N, __fp16 *a, __fp16 *b, __fp16 *c)
{
  #pragma omp simd
  for (int i=0;i<N;i++)
  {
    c[i] = a[i]+b[i];
  }
}
```

```
icx -qopenmp-simd -O2 -xsapphirerapids -c -S -unroll0

.LBB0_3:
 vmovups (%rdx,%rax,2), %ymm0
 vaddph  (%rsi,%rax,2), %ymm0, %ymm0
 vmovups %ymm0, (%rcx,%rax,2)
 addq    $16, %rax
 cmpq    %rdi, %rax
 jb      .LBB0_3
```

Figure 4. FP16 vectorization example (I) using VPlan Vectorizer

Note that not all optimizers work well out-of-the-box for the newly introduced instruction set. Figure 5 is the same example from Figure 2 but using the FP16 data type. The innermost loop with the `vmulph` instruction is currently not as nicely optimized as in Figures 2 and 3. In the upcoming releases, we'll continue uncovering and improving these issues.

```
void foo(int N, __fp16 *a, __fp16 *b, __fp16 *c)
{
  #pragma omp simd
  for (int i=0;i<N;i++)
  {
    __fp16 x = a[i];
    __fp16 y = b[i];
    while(x>y)
    {
      x = x*x;
    }
    c[i] = x;
  }
}
```

```
icx -qopenmp-simd -O2 -xsapphirerapids -c -S -unroll0

 jmp     .LBB0_4
.LBB0_5:
 vpxor   %xmm2, %xmm2, %xmm2
.LBB0_12:
 vcmpltph        %ymm0, %ymm1, %k1
 vmovdqu16       %ymm2, %ymm0 {%k1}
 vmovdqu %ymm0, (%rcx,%rax,2)
 addq    $16, %rax
 cmpq    %rdi, %rax
 jae     .LBB0_13
.LBB0_4:
 vmovups (%rsi,%rax,2), %ymm0
 vmovups (%rdx,%rax,2), %ymm1
 vcmpltph        %ymm0, %ymm1, %k0
 kortestw        %k0, %k0
 je      .LBB0_5
# %bb.6:
 vmovaps %ymm0, %ymm3
 kmovq   %k0, %k1
 jmp     .LBB0_7
.LBB0_11:
 vmovdqu16       %ymm3, %ymm2 {%k1}
 kandw   %k1, %k2, %k1
 ktestw  %k0, %k1
 je      .LBB0_12
.LBB0_7:
 ktestw  %k1, %k0
 vmulph  %ymm3, %ymm3, %ymm4
 vxorps  %xmm3, %xmm3, %xmm3
 je      .LBB0_9
# %bb.8:
 vmovaps %ymm4, %ymm3
.LBB0_9:
 kxorw   %k0, %k0, %k2
 je      .LBB0_11
# %bb.10:
 vcmpltph %ymm4, %ymm1, %k2
 jmp     .LBB0_11
```

Figure 5. FP16 vectorization example (II) using VPlan Vectorizer

Enhancing Auto-Vectorization in GCC12

In this section, we describe several auto-vectorization enhancements developed recently for AVX-512/ AVX-512-VNNI support in GCC12 compiler based on GCC vectorization framework previously done for Intel Xeon Phi processors.

- GCC12 auto-vectorization is enabled by default at -O2 using a “cheap” cost model, which permits loop vectorization if the trip count of a scalar vectorizable loop is a multiple of the hardware vector length, and with no observable code size increasing. For example, Figure 6 shows an example of GCC -O2 auto-vectorization using SSE4.2. Meanwhile, the default cost model for loop vectorization at -O3 employs a "dynamic" model with more checkpoints to determine whether the vectorized code path will achieve performance gains.

```
void ArrayAdd(int* __restrict a, int* b)
{
   for (int i = 0; i != 32; i++)
      a[i] += b[i];
}
```

```
ArrayAdd:
  xorl    %eax, %eax
.L2:
  movdqu  (%rdi,%rax), %xmm0
  movdqu  (%rsi,%rax), %xmm1
  paddd   %xmm1, %xmm0
  movups  %xmm0, (%rdi,%rax)
  addq    $16, %rax
  cmpq    $128, %rax
  jne     .L2
  ret
```

Figure 6. GCC (-O2) auto-vectorization example

- GCC vectorization for the _Float16 type is enabled to generate corresponding AVX512FP16 instructions. In addition to those SIMD instructions that are similar to their float/double variants, the vectorizer also supports vectorization for the complex _Float16 type. Figure 7 shows an example that performs a conjugate complex multiply and accumulate operations on three arrays, and the vectorizable loop can be optimized to generate a `vfcmaddcph` instruction.

```
#include<complex.h>

void fmaconj (_Complex _Float16 a[restrict 16],
              _Complex _Float16 b[restrict 16],
              _Complex _Float16 c[restrict 16])
{
   for (int i = 0; i < 16; i++)
      c[i] += a[i] * ~b[i];
}
```

```
fmaconj:
  vmovdqu16   (%rdx), %zmm1
  vmovdqu16   (%rsi), %zmm0
  vfcmaddcph  (%rdi), %zmm1, %zmm0
  vmovdqu16   %zmm0, (%rdx)
  vzeroupper
  ret
```

Figure 7. GCC auto-vectorization of using the AVX512FP16 vfcmaddcph instruction

- GCC auto-vectorization is enhanced to perform idiom recognition such as the dot-plus idiom, which triggers the AVX/AVX512VNNI instruction generation. Figure 8 shows that the compiler generates the `vpdpbusd` instruction plus a summation reduction.

```
int usdot_prod_qi (unsigned char * restrict a,
                   char *restrict b, int c, int n)
{
  for (int i = 0; i < 32; i++)
  {
    c += ((int) a[i] * (int) b[i]);
  }
  return c;
}
```

```
usdot_prod_qi:
  vmovdqu  (%rdi), %ymm0
  vpxor    %xmm1, %xmm1, %xmm1
  vpdpbusd (%rsi), %ymm0, %ymm1
  vextracti128 $0x1, %ymm1, %xmm0
  vpaddd   %xmm1, %xmm0, %xmm0
  vpsrldq  $8, %xmm0, %xmm1
  vpaddd   %xmm1, %xmm0, %xmm0
  vpsrldq  $4, %xmm0, %xmm1
  vpaddd   %xmm1, %xmm0, %xmm0
  vmovd    %xmm0, %eax
  addl     %edx, %eax
  vzeroupper
  ret
```

Figure 8. AV512VNNI idiom recognition in GCC auto-vectorization

In addition to the three aforementioned enhancements in GCC auto-vectorization, we have improved GCC to utilize `vpopcnt[b,w,d,q]` instructions when the redundant zero extension and truncation is recognized by the vectorizer as well. These improvements significantly extend GCC auto-vectorization capability for Intel Xeon Scalable processors.

SIMD Vectorization for Intel GPUs

Design Rationale

Intel GPUs, using Intel Xe Architecture, are designed to support both OpenCL SIMT (Single Thread Multiple Data) and SIMD. In this section, we describe how to enable our LLVM VPlan vectorizer for converting OpenMP SIMD loops to SIMD code by leveraging underlying SIMD ISA in Xe GPUs. The rationale behind the design and implementation is two-fold:

- Provide a relatively smooth transition to migrate existing C++ and Fortran OpenMP CPU applications that uses SIMD constructs to Xe GPUs utilizing OpenMP offloading and SIMD.
- Exploit SIMD loop vectorization flexibility with different explicit SIMD schemes in the OpenMP offloading region to fully leverage Xe GPU SIMD ISA.

The oneAPI C++/Fortran OpenMP compiler SIMD vectorization for Intel GPUs is designed to exploit the underlying hardware features, allowing fine-grained register management, SIMD size control, and cross-lane data sharing.

High-Level SIMD Vectorization Framework

Figure 9 outlines the SIMD vectorization framework implemented in the device compilation path for Intel GPUs, which fully leverages the LLVM VPlan Vectorizer we built for CPUs [4][5] in oneAPI compilers. The VPlan Vectorizer (box IV) takes LLVM scalar IR from the language Front-End (box I) and middle end optimizations (boxes II and III) performing LLVM Vector IR generation in conjunction with a lowering transformation to GPU target intrinsics defined for Xe GPU operations (box V). Then, passing GPU-ready LLVM Vector IR to the GPU Vector Back-End compiler (boxes VI and VII) [8] using SPIR-V as an interface IR.

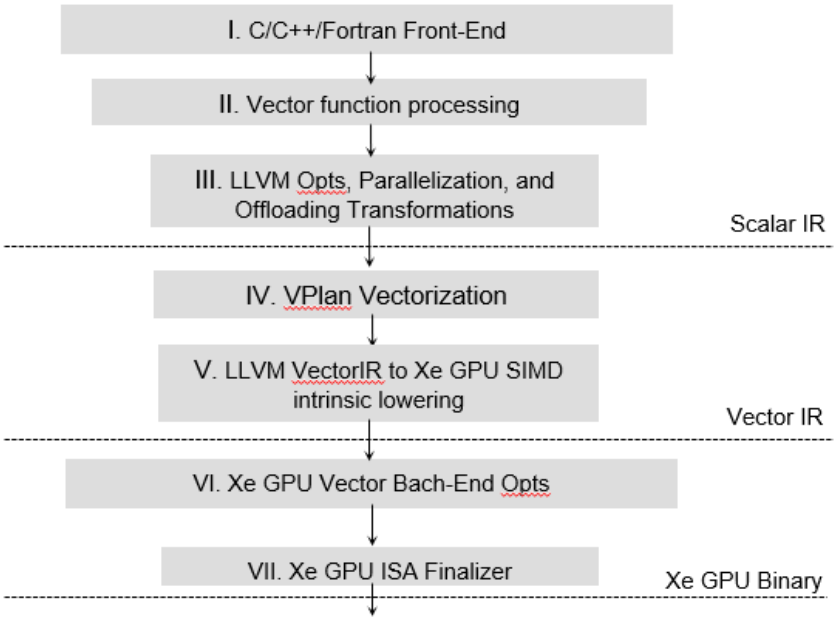


Figure 9. SIMD vectorization framework for device compilation

There is a sequence of explicit SIMD-specific optimizations and transformations (box VI) developed around those GPU-specific intrinsics. Note that programmers are provided with controls on loop vectorization and vector length selection through OpenMP programming APIs while the compiler Vector Back-End (boxes VI and VII) strives to achieve a tradeoff among various compiler optimizations based on programmer annotations. In addition, OpenMP explicit SIMD kernels generated by the compiler middle end are fully compatible with the Intel GPU OpenCL runtime [1] and oneAPI Level Zero [2] and can be launched directly as if they are written in OpenCL.

Intel Xᵉ Architecture GPU SIMD Code Generation Example

Figure 10 shows an OpenMP offload example. In the target region, there are two SIMD loops: one operates on single-precision multiply-and-add (FMA) with simdlen(8) and the other operates on double-precision multiply-and-add with the simdlen(8) clause. So, the compiler can perform 512-bit SIMD vectorization for both loops.

```
Float a[N][M]; double b[N][M];
... ...
#pragma omp target teams distribute parallel for map(tofrom:a[0:N][0:M]) map(tofrom:b[0:N][0:M])
    for (int k = 0; k < N; ++k) {
      float  x = k * 1.0f;
      double y = k * 1.0;
#pragma omp simd simdlen(16)
      for (int j = 0; j < M; ++j) {
        a[k][j] = a[k][j] + x*a[k][j];
      }

#pragma omp simd simdlen(8)
      for (int j = 0; j < M; ++j) {
        b[k][j] = b[k][j] + y*b[k][j];
      }
}
... ...
```

Figure 10. An example with different SIMD width in OpenMP target region

For SIMD loop vectorization, if a loop trip count is known at compile-time, the compiler can decide to unroll the loop. In this program example, the first SIMD loop is vectorized with SIMD16 and unrolled by two, the second SIMD loop is vectorized with SIMD8 and unrolled by four for the given trip count M=32 as shown in Figure 11. A common issue to compilers is that the loop trip count is unknown at compile-time. However, if application programmers can reason about and predict the trip count and provide a hint to the compilers using #pragma loop count, it will enable the compiler to perform the desired loop unrolling for compute-bound loops (i.e., computation takes more time than memory accesses).

```
        ... ...
        mad (16|M0)         r7.0<1>:f     r5.0<1;0>:f       r5.0<1;0>:f   r1.6<0>:f  {Compacted,$8.dst}
(W&f1.0.any16h) send.dc1 (16|M0)   null   r33     r7      0x80   0x020D43FF    {$3}
(W&f1.0.any16h) send.dc1 (16|M0)   r9     r34     null    0x0    0x022D0BFF    {$9}
        mad (16|M0)         r11.0<1>:f    r9.0<1;0>:f       r9.0<1;0>:f   r1.6<0>:f  {Compacted,$9.dst}
(W&f1.0.any16h) send.dc1 (16|M0)   null   r35     r11     0x80   0x020D43FF    {A@1,$6}
(W&f1.0.any16h) send.dc1 (16|M0)   r13    r36     null    0x0    0x022D0BFF    {$10}
        mad (8|M0)               r15.0<1>:df   r13.0<1;0>:df     r13.0<1;0>:df r4.2<0>:df {$10.dst}
(W&f1.0.any16h) send.dc1 (16|M0)   null   r37     r15     0x80   0x020D43FF
(W&f1.0.any16h) send.dc1 (16|M0)   r17    r38     null    0x0    0x022D0BFF
        mad (8|M0)               r19.0<1>:df   r17.0<1;0>:df     r17.0<1;0>:df r4.2<0>:df {$11.dst}
(W&f1.0.any16h) send.dc1 (16|M0)   null   r39     r19     0x80   0x020D43FF    {A@1,$4}
(W&f1.0.any16h) send.dc1 (16|M0)   r22    r40     null    0x0    0x022D0BFF    {$12}
        mad (8|M0)               r24.0<1>:df   r22.0<1;0>:df     r22.0<1;0>:df r4.2<0>:df {$12.dst}
(W&f1.0.any16h) send.dc1 (16|M0)   null   r41     r24     0x80   0x020D43FF    {A@1,$7}
(W&f1.0.any16h) send.dc1 (16|M0)   r26    r42     null    0x0    0x022D0BFF    {$13}
        mad (8|M0)               r28.0<1>:df   r26.0<1;0>:df     r26.0<1;0>:df r4.2<0>:df {$13.dst}
(W&f1.0.any16h) send.dc1 (16|M0)   null   r43     r28     0x80   0x020D43FF    {A@1,$5}
        ... ...
```

Figure 11. Intel GPU SIMD code generated with unrolling based on data types

Summary

We presented the recent evolution of SIMD vectorization technology in the LLVM and GCC compilers for underlying Intel CPU and Intel GPU ISAs. Several vectorization features are illustrated for how to expose the underlying hardware capabilities to exploit SIMD parallelism. On Intel GPUs, SIMD vectorization is a complementary to the existing popular SPMD model. As a continuous effort, more performance tuning and optimizations will be added into Intel oneAPI LLVM-based compilers and GCC compilers for Intel CPUs AVX-512 and AVX-512-FP16/VNNI ISA and Intel GPUs Gen12 ISA.

References

[1] Intel, Intel® Graphics Compute Runtime for oneAPI Level Zero and OpenCL Driver, https://github.com/intel/compute-runtime, 2020.

[2] Intel, oneAPI Level Zero Specification, 2020. https://spec.oneapi.com/level-zero/latest/index.html

[3] C++ Standards Committee, Data-parallel vector library, 2020. https://en.cppreference.com/w/cpp/experimental/simd

[4] H. Saito, S. Preis, N. Panchenko, and X. Tian. Reducing the Functionality Gap between Auto-Vectorization and Explicit Vectorization. In Proceedings of the International Workshop on OpenMP (IWOMP), LNCS9903, pp. 173-186, Springer, 2016.

[5] X. Tian, H. Saito, E. Su, J. Lin, et.al. *LLVM Compiler Implementation for Explicit Parallelization and SIMD Vectorization*. LLVM-HPC@SC 2017: 4:1-4:11

[6] X. Tian, R. Geva, B. Valentine. Unleash the Power of AVX-512 through Architecture, Compiler and Code Modernization, ACM Parallel Architecture and Compiler Technology, September 11-15, 2016, Haifa, Israel.

[7] X. Tian, Bronis R. de Supinski: Explicit Vector Programming with OpenMP* 4.0 SIMD Extensions, HPC Today America, Nov 19. 2014. http://www.hpctoday.com/hpc-labs/explicit-vector-programming-with-openmp-4-0-simd-extensions/

[8] Guei-Yuan Lueh, Kaiyu Chen, Gang Chen, Joel Fuentes, Wei-Yu Chen, Fangwen Fu, Hong Jiang, Hongzheng Li, and Daniel Rhee, C-for-Metal: High Performance SIMD Programming on Intel GPUs. CGO 2021, 289-300.

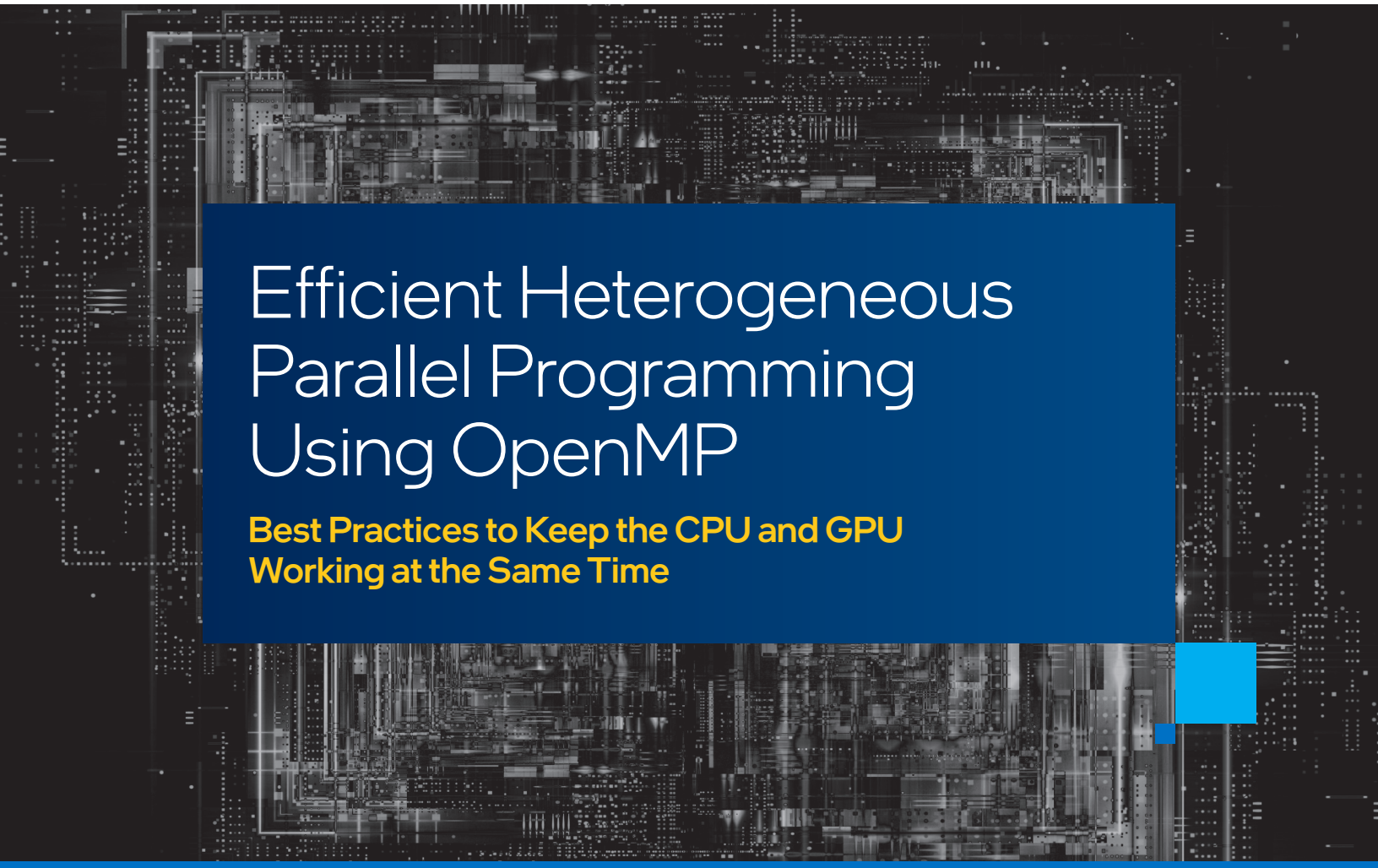
[9] "Intel C/C++ compilers complete adoption of LLVM" https://www.intel.com/content/www/us/en/developer/articles/technical/adoption-of-llvm-complete-icx.html

[10] Matt Masten, Evgeniy Tyurin, K. Mitropoulou, Eric N. Garcia, and H. Saito Function/Kernel Vectorization via Loop Vectorizer, 2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)

[11] Intel AVX-512-FP16 Architecture Specification https://software.intel.com/content/dam/develop/external/us/en/documents-tps/intel-avx512-fp16.pdf

[12] Intel Corporation, Vector Function Application Binary Interface https://docplayer.net/197118571-Vector-function-application-binary-interface.html

[13] GCC patches can be found under https:/gcc.gnu.org/git (look for AVX512/VNNI/FP16 support), see also https://www.phoronix.com/scan.php?page=news_item&px=AFX-512-FP16-GCC-Patches for more on FP16 patches.



Efficient Heterogeneous Parallel Programming Using OpenMP

Best Practices to Keep the CPU and GPU Working at the Same Time

Elmira Volkova, Undergraduate Intern, Alexander Bobyr, Software Enabling and Optimization Engineer, Igor Ermolaev, Principal Engineer

In some cases, offloading computations to an accelerator like a GPU means that the host CPU sits idle until the offloaded computations are finished. However, using the CPU and GPU resources simultaneously can improve the performance of an application. In OpenMP® programs that take advantage of heterogenous parallelism, the *master* clause can be used to exploit simultaneous CPU and GPU execution. In this article, we will show you how to do CPU+GPU asynchronous calculation using OpenMP.

The SPEC ACCEL 514.pomriq MRI reconstruction benchmark is written in C and parallelized using OpenMP. It can offload some calculations to accelerators for heterogenous parallel execution. In this article, we divide the computation between the host CPU and a discrete Intel® GPU such that both processors are kept busy. We'll also use Intel VTune™ Profiler to measure CPU and GPU utilization and analyze performance.

We'll look at five stages of heterogeneous parallel development and performance tuning:

1. Looking for appropriate code regions to parallelize
2. Parallelizing these regions so that both the CPU and GPU are kept busy
3. Finding the optimal work distribution coefficient
4. Launching the heterogeneous parallel application with this distribution coefficient
5. Measuring the performance improvement.

Initially, the parallel region only runs on the GPU while the CPU sits idle (Figure 1). As you can see, only the "OMP Primary Thread" is executing on the CPU while the GPU is fully occupied (GPU Execution Units→EU Array→Active) with the ComputeQ offloaded kernel.

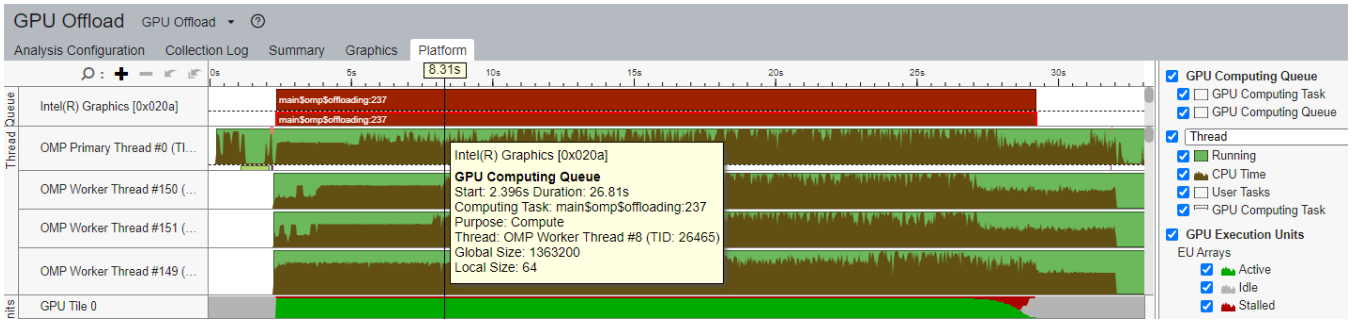


Figure 1. Profile of the initial code using Intel VTune Profiler

After examining the code, we decided to duplicate each array and each executed region so that the first copy is executed on the GPU and the second is executed on the CPU. The master thread uses the OpenMP *target* directive to offload work to the GPU. This is shown schematically in Figure 2. The *nowait* directives avoid unnecessary synchronization between the threads running on the CPU and GPU. They also improve load balance among the threads.

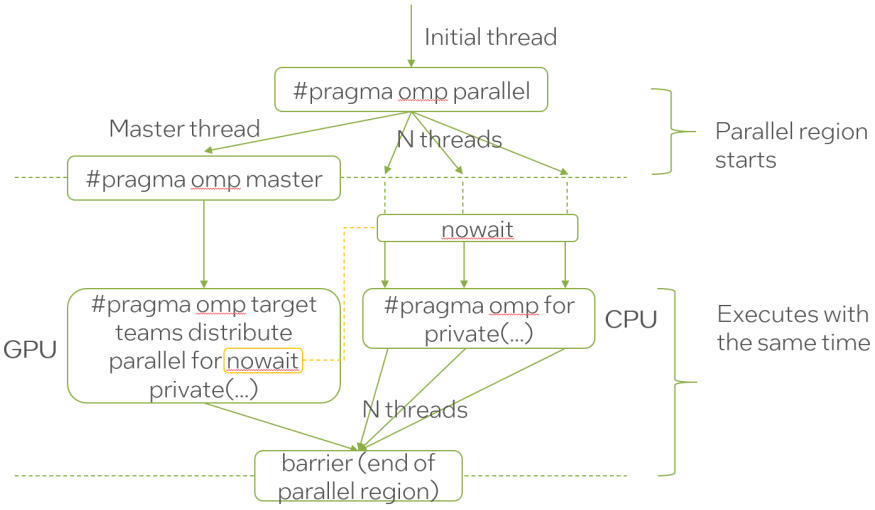


Figure 2. OpenMP parallelization scheme to keep the CPU and GPU busy

Balancing the work distribution between the CPU and GPU is regulated by the *part* variable that is read from STDIN (Figure 3). This variable is the percentage of the workload that will be offloaded to the GPU multiplied by *numX*. The remaining work will be done on the CPU. An example of the OpenMP heterogeneous parallel implementation is shown in Figure 4.

```
    float part;
    char *end;
    part   = strtof(argv[2], &end);

int TILE_SIZE = numX*part;

Qrc = (float*) memalign(alignment, numX * sizeof (float));
Qrg = (float*) memalign(alignment, TILE_SIZE * sizeof (float));
```

Figure 3. The coefficient of distribution work between the CPU and GPU

```
#pragma omp parallel
 {
        #pragma omp master
        #pragma omp target teams distribute parallel for nowait private(expArg, cosArg, sinArg)
        for (indexX = 0; indexX < TILE_SIZE; indexX++) {
            float QrSum = 0.0;
            float QiSum = 0.0;
            #pragma omp simd private(expArg, cosArg, sinArg) reduction(+:QrSum, QiSum)
            for (indexK = 0; indexK < numK; indexK++) {
                expArg = PIx2 * (GkVals[indexK].Kx * xg[indexX] + GkVals[indexK].Ky * yg[indexX] + GkVals[indexK].Kz * zg[indexX]);
                cosArg = cosf(expArg);
                sinArg = sinf(expArg);
                float phi = GkVals[indexK].PhiMag;
                QrSum += phi * cosArg;
                QiSum += phi * sinArg;
            }
            Qrg[indexX] += QrSum;
            Qig[indexX] += QiSum;
        }
        #pragma omp for private(expArg, cosArg, sinArg)
        for (indexX = TILE_SIZE; indexX < numK; indexX++) {
            float QrSum = 0.0;
            float QiSum = 0.0;
            #pragma omp simd private(expArg, cosArg, sinArg) reduction(+:QrSum, QiSum)
            for (indexK = 0; indexK < numK; indexK++) {
                expArg = PIx2 * (CkVals[indexK].Kx * xc[indexX] + CkVals[indexK].Ky * yc[indexX] + CkVals[indexK].Kz * zc[indexX]);
                cosArg = cosf(expArg);
                sinArg = sinf(expArg);
                float phi = CkVals[indexK].PhiMag;
                QrSum += phi * cosArg;
                QiSum += phi * sinArg;
            }
            Qrc[indexX] += QrSum;
            Qic[indexX] += QiSum;
        }
 }
```

Figure 4. Example code illustrating the OpenMP implementation that simultaneously utilizes the CPU and GPU

The Intel® oneAPI DPC++/C++ Compiler was used with following command-line options:

```
-O3 -Ofast -xCORE-AVX512 -mprefer-vector-width=512 -ffast-math
-qopt-multiple-gather-scatter-by-shuffles -fimf-precision=low
-fiopenmp -fopenmp-targets=spir64="-fp-model=precise"
```

Table 1 shows the performance for different CPU to GPU work ratios (i.e., the *part* variable described above). For our system and workload, an offload ratio of 0.65 gives the best load balance between the CPU and GPU, and hence the best utilization of processor resources. The profile from Intel VTune Profiler shows that work is more evenly distributed between the CPU and GPU, and that both processors are being effectively utilized (Figure 5). While "OMP Primary Thread" submits the offloaded kernel (main: 237) for execution on the GPU, other "OMP Worker Threads" are active on the CPU.

Offload part	Total time, s	GPU time, s
0.00	61.2	0.0
0.20	51.6	8.6
0.40	41.0	16.8
0.60	31.5	24.7
0.65	28.9	26.7
0.80	34.8	32.6
1.00	43.4	40.7

Table 1. Hotspot times corresponding to different amounts of offloaded work (i.e., the *part* variable)

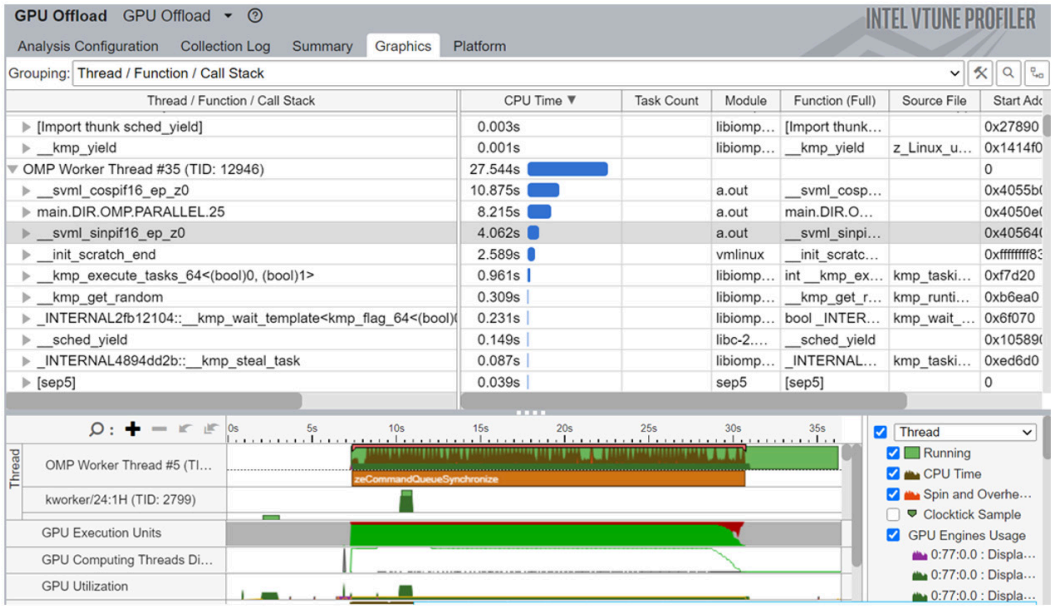


Figure 5. Profile of code with 65% GPU offload

Figure 6 shows the run times for different values of *part*. Keep in mind that a *part* of zero means that no work is offloaded to the GPU. A *part* of one means that all work is offloaded. It's clear that a balanced distribution of work across the CPU and GPU gives better performance than either extreme.

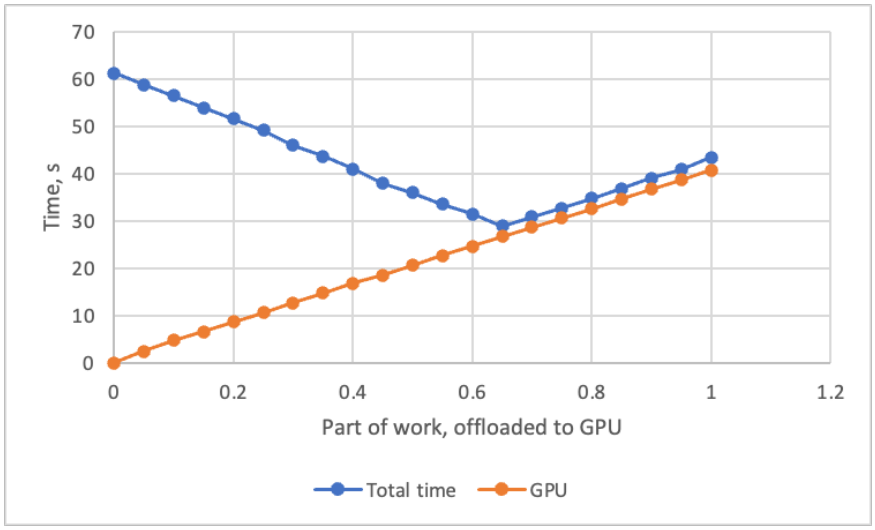
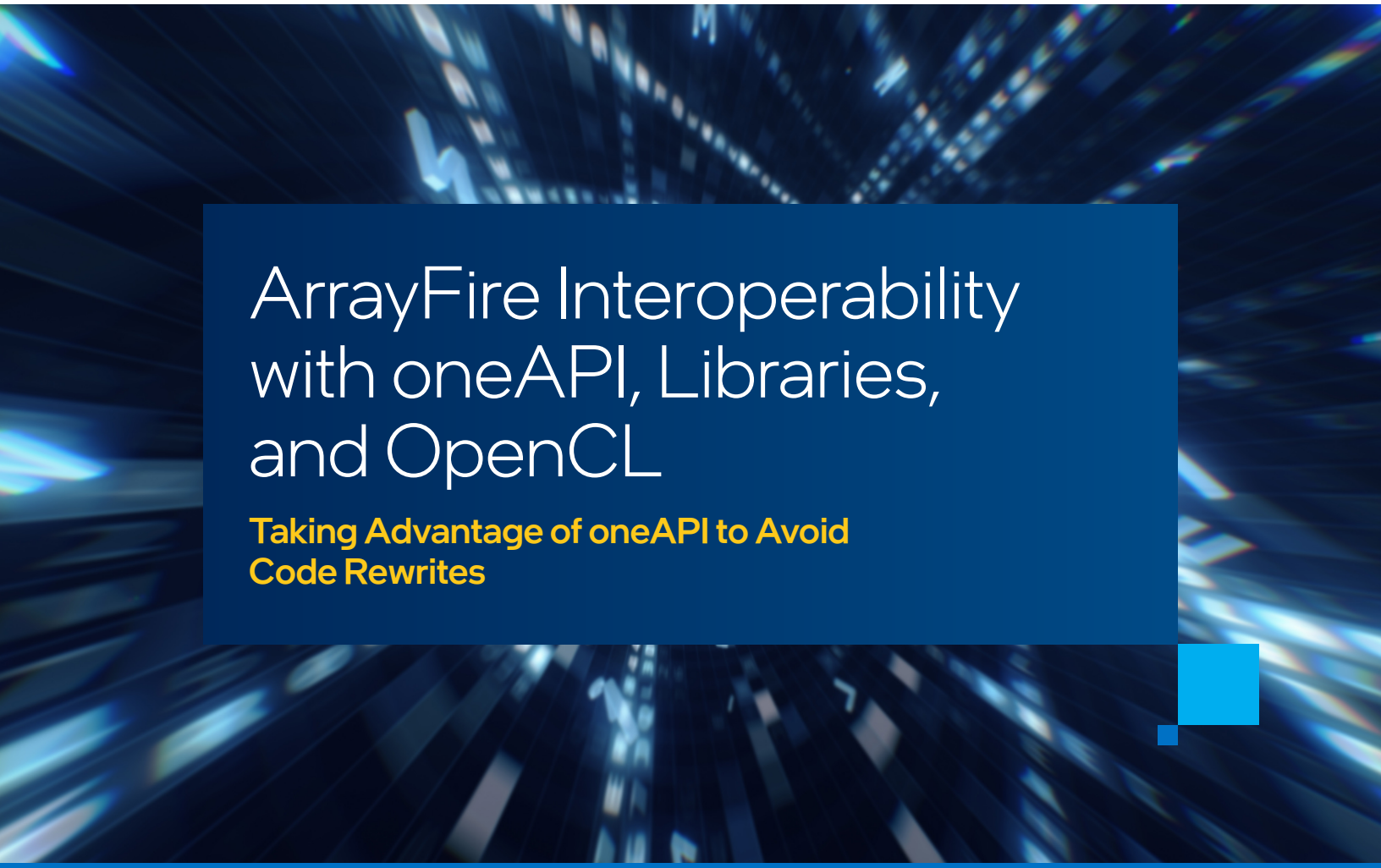


Table 1. Comparing training and prediction performance (all times in seconds)

OpenMP provides true asynchronous, heterogeneous execution on CPU+GPU systems. It's clear from our timing results and VTune profiles that keeping the CPU and GPU busy in the OpenMP parallel region gives the best performance. We encourage you to try this approach.



ArrayFire Interoperability with oneAPI, Libraries, and OpenCL

Taking Advantage of oneAPI to Avoid Code Rewrites

Stefan Yurkevitch, Software Engineer, ArrayFire

oneAPI greatly simplifies development on heterogeneous accelerators. With a code once, run anywhere approach, the APIs offer a powerful way to develop code. ArrayFire is a GPU library that already offers a vast collection of useful functions for many computational domains. It shares the philosophy oneAPI brings to the software development world. In this article, we'll be exploring how to integrate the oneAPI Deep Neural Network (oneDNN) library and the SYCL-based Data Parallel C++ (DPC++) programming language into existing codebases. Our goal is to allow the programmer to take advantage of oneAPI to avoid the code rewriting often required when migrating to a new programming model.

Interoperability with SYCL

oneAPI is the combination of DPC++ and libraries to simplify cross-architecture parallel programming. The libraries are tightly integrated with the DPC++ language. They both provide a variety of methods for interoperability with the underlying OpenCL implementation. The base language provides three main methods of interoperability with OpenCL that cover most use-cases (Figure 1). The flow of interoperability functions can be from existing code to SYCL or vice versa, specifically:

1. Using existing OpenCL kernels within DPC++ code by creating a kernel object from the kernel string
2. Extracting OpenCL objects from existing SYCL objects
3. Creating SYCL objects from existing OpenCL objects

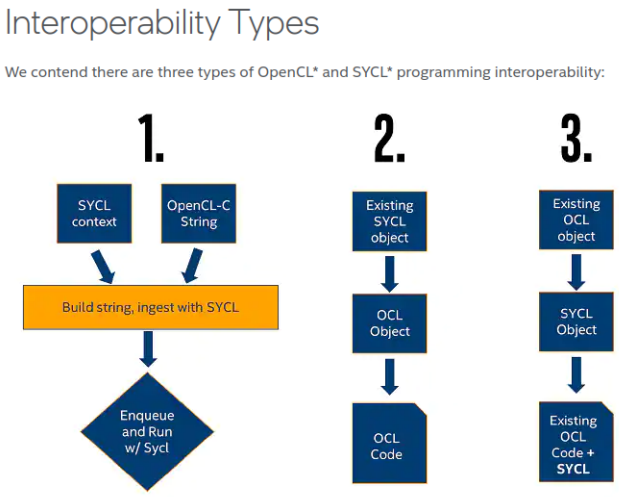


Figure 1. SYCL interoperability with OpenCL

Let's consider how the existing ArrayFire codebase could be integrated with these interoperability options. In the first case, we could directly reuse the raw ArrayFire kernels (Figure 1, left):

```
queue q{gpu_selector()};        // Create command queue targeting GPU
program p(q.get_context());     // Create program from the same context as q

// Compile OpenCL vecAdd kernel, which is expressed as a C++ Raw String as indicated by R"
p.build_with_source(R"( __kernel void existingArrayFireVecAdd(__global int *a,
                                                              __global int *b,
                                                              __global int *c)
                        {
                                int i = get_global_id(0);
                                c[i] = a[i] + b[i];
                        } )");
  // buffers here ...
  q.submit([&](handler& h) {
          // accessors here...
          // Set buffers as arguments to the kernel
          h.set_args(A, B, C);
          // Launch vecAdd kernel from the p program object across N elements.
          h.parallel_for(range<1> (N), p.get_kernel("vecAdd"));
  });
```

In reality, ArrayFire kernels rely on more complicated data structures than simple buffers so reusing kernels in this manner isn't as trivial as copy-pasting the CL string. We'll need to handle the data exchange using one of the other two methods.

The second method (Figure 1, middle) of extracting OpenCL components from SYCL objects is based on the simple convention of using the `.get()` method on existing SYCL objects. Each call on a SYCL object will return the corresponding underlying OpenCL object. For example, `cl::sycl::queue::get()` will return an OpenCL `cl_command_queue`.

The third method (Figure 1, right) takes existing OpenCL objects and uses them to create SYCL objects. This can be done using the SYCL object's constructors, such as `sycl::queue::queue(cl_command_queue,...)`. In these cases, the constructors will also retain the OpenCL instance to increase the reference count of the OpenCL resource during construction and will release the instance during the destruction of the SYCL object.

Interoperability with oneAPI Libraries

Similar interoperability conventions exist within the oneAPI libraries. Some of the libraries, like oneMKL, directly rely on the DPC++ interoperability. Their functions can accept unified shared memory (USM) pointers. Others, like oneDNN, which we will be using in our example, provide similar `.get()` and `constructor()` mechanisms.

oneDNN has similar, yet slightly different data structures from DPC++. The `sycl::device` and `sycl::context` are combined into a single `dnnl::engine` object, and the `dnnl::stream` replaces the `sycl::queue`. Despite these differences, the mechanism for OpenCL interoperability remains the same. OpenCL objects can be obtained with getter functions while new oneDNN objects can be created from existing OpenCL objects through their constructors. oneDNN also provides an explicit interoperability header with the same functionality.

oneDNN is flexible in terms of its supported runtime backend. It can use either the OpenCL runtime or the DPC++ runtime for CPU and GPU engines to interact with the hardware. Developers may need to use oneDNN with other code that uses either OpenCL or DPC++. For that purpose, the library provides API extensions to interoperate with the corresponding underlying objects. Depending on the target, the interoperability API is defined in either the `dnnl_ocl.hpp` or `dnnl_sycl.hpp` header. For our use-case, we're interested in supplementing the capabilities of oneDNN's inference engine with the existing preprocessing capabilities offered by the ArrayFire library. For now, this will be done through the OpenCL interoperability functions.

ArrayFire and oneDNN: The Details

The motivating example we'll be using to explore the details of OpenCL interop with oneDNN is based on the cnn_inference_f32.cpp sample. This example sets up an AlexNet network using oneDNN for inference. Our goal is to use ArrayFire's many OpenCL image processing functions to preprocess the user input images before feeding the data to the existing inference engine. The full workflow involves the following steps:

- Include the relevant interoperability headers
- Create a GPU engine while sharing the cl_context with ArrayFire
- Create a GPU command queue via the OpenCL interoperability interface
- Perform preprocessing and data preparation with ArrayFire
- Create a GPU memory descriptor/object
- Access GPU memory via OpenCL interoperability interface for input
- Create oneDNN primitives/descriptors/memory to build the network
- Execute the network as usual with oneDNN
- Release GPU memory

The first additions we need to make to the file include the interoperability headers for both ArrayFire and oneDNN. The OpenCL headers are included as well.

```
#include "oneapi/dnnl/dnnl.hpp"     // oneDNN header
#include "oneapi/dnnl/dnnl_ocl.hpp" // oneDNN OpenCL interop header

#include <CL/cl.h>       // OpenCL header

#include <arrayfire.h>   // ArrayFire header
#include <af/opencl.h>   // ArrayFire OpenCL interop header
```

Next, we'll grab the OpenCL context and queue from ArrayFire to share with oneDNN:

```
cl_device_id     af_device_id = afcl::getDeviceId();
cl_context       af_context   = afcl::getContext();
cl_command_queue af_queue     = afcl::getQueue();
```

The OpenCL objects will be used to create the corresponding oneDNN objects. This will use the interoperability functions defined in the interop header. These functions reside in the additional ocl_interop namespace. Remember that this will retain the objects throughout the lifetime of the oneDNN scope:

```
dnnl::engine eng = dnnl::ocl_interop::make_engine(af_device_id, af_context);
dnnl::stream s   = dnnl::ocl_interop::make_stream(eng, af_queue);
```

Then we can load and preprocess our images reusing the ArrayFire library's accelerated GPU functions:

```
// create empty array within same context as oneDNN
af::array images = af::constant(0.f, h, w, 3, batch);
images = read_images(directory);
images = af::resize(images, 227, 227) / 255.f; // resize to alexnet input size
                                               // and normalize [0-1]
     images = af::reorder(images, 3, 2, 0, 1); // hwcn -> nchw
...                                            // additional preprocessing
```

oneDNN finally requires the `dnnl::memory` object. This isn't raw memory, but rather some memory together with additional metadata such as a `dnnl::descriptor`. oneDNN supports both buffer and USM memory models. Buffering is the default. To construct a oneDNN memory object with interop support, we will use the following interop function:

```
ocl::interop make_memory(
   const memory::desc& memory_desc, // descriptor describing memory shape and layout
   const engine& aengine,           // our interop engine
   memory_kind kind,                // buffer or USM
   void* handle = DNNL_MEMORY_ALLOCATE // handle to underlying storage
   )
```

Here, the descriptors follow those of the sample where we expect the input to AlexNet to be a 227 x 227 NCHW image. The engine is just our execution engine that we have been sharing between ArrayFire and oneDNN. The memory kind should specify if we're using the USM or buffer interface. If we chose to pass in a handle pointer, it should then proceed to match the type of memory we pass in. If the handle is a USM pointer or an OpenCL buffer, the oneDNN library doesn't own the buffer and the user is responsible for managing the memory. With the special `DNNL_MEMORY_ALLOCATE` value, the library will allocate a new buffer on the user's behalf.

oneDNN supports both buffer and USM memory models, so replacing the engines and queues with objects shared with ArrayFire will result in incompatible memory creation modes. During the creation of the `dnnl::memory` object, the following error can occur:

```
oneDNN error caught:
        Status: invalid_arguments
        Message: could not create a memory object
```

Instead of the default method of `dnnl::memory` creation, the interoperability functions must be used instead, as follows:

```
cl_mem *src_mem = images.device<cl_mem>(); // get cl_mem from arrayfire
dnnl::memory user_src_memory = ocl_interop::make_memory( //  interop mem function
       {{conv1_src_tz}, dt::f32, tag::nchw},             // create descriptor
       eng,                                              // specify engine
       ocl_interop::memory_kind::buffer,                 // specify memory type
       *src_mem);                                        // pass cl_mem handle
```

This applies all instances of default `dnnl::memory` allocation. The interoperability functions to specify the `ocl_interop::memory_kind::buffer` must be used:

```
ocl_interop::make_memory(descriptor, engine, ocl_interop::memory_kind::buffer);
```

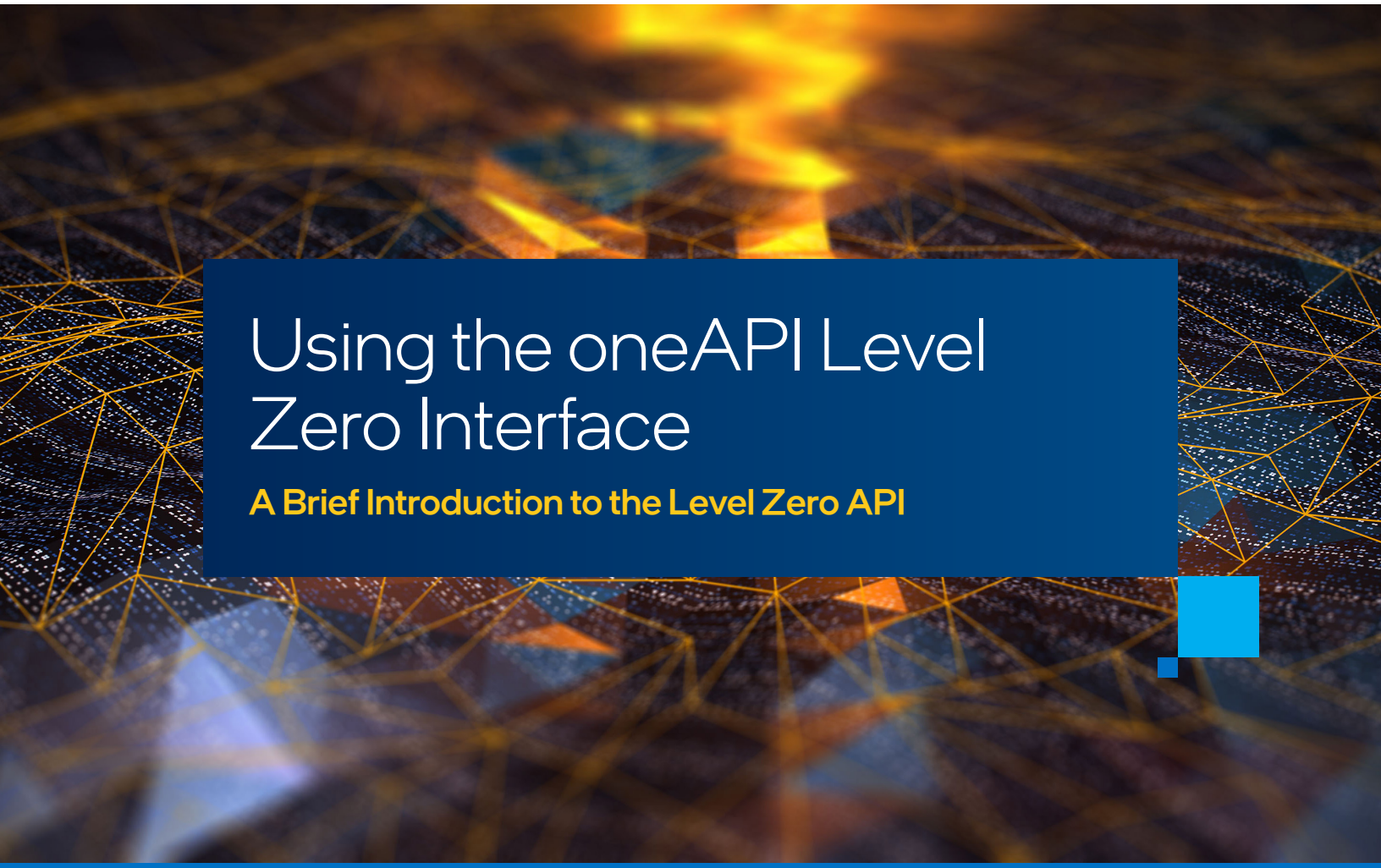
Finally, after all weights are loaded, the inference primitives can be created and called as usual. After the network has run, we should free the resources that we are responsible for:

```
// additional alexnet network setup
// loading of weights following cnn_inference_f32.cpp
...

// execute all primitive steps for full inference using our inputs
for (size_t i = 0; i < net.size(); ++i) {
  net.at(i).execute(s, net_args.at(i));
}

s.wait();   // wait until stream finishes writing to memory

images.unlock();  // return memory ownership to arrayfire to free resources
```

We want to make sure we're running oneDNN with the OpenCL runtime rather than the DPC++ runtime. This can be achieved by specifying the `SYCL_DEVICE_FILTER=opencl` environment variable. A modified, working `cnn_inference_f32.cpp` for reference can be found in this gist.

Conclusion

oneAPI provides all the tools required to integrate existing OpenCL codebases with the new heterogeneous programming approach. The underlying OpenCL objects can be shared in either direction with DPC++. oneAPI's libraries have their own methods to handle the interoperability tasks. With minor code changes, whole OpenCL libraries can be reused rather than rewritten. oneAPI saves future development time by avoiding redevelopment efforts of already useful code.



Using the oneAPI Level Zero Interface

A Brief Introduction to the Level Zero API

Rama Kishan Malladi, Solution Architect, Amazon Web Services
Nitya Hariharan, Application Engineer, Intel Corporation

The oneAPI specification simplifies software development by providing the same language, API, and programming model across accelerator architectures. It defines a set of APIs for common data parallel domains, across a variety of architectures. Both the API and the direct programming approaches are based on data parallelism (i.e., the same computation is performed on each data element). The oneAPI™ platform consists of a host and a collection of accelerator devices (Figure 1). The API programming model is implemented using oneMKL, oneDPL, oneDNN, oneCCL, and other libraries. Direct programming is done using DPC++.

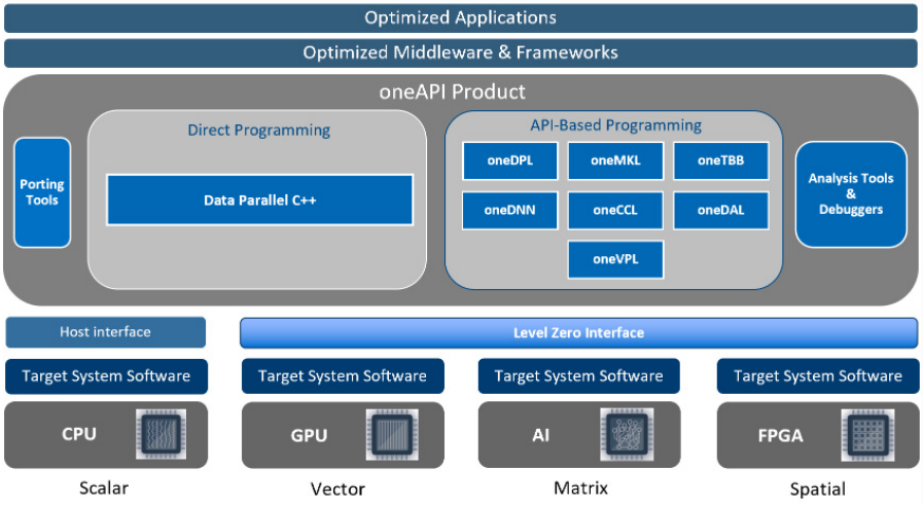


Figure 1. The oneAPI platform execution model. Note that the Intel® implementation of the oneAPI specification also contains programming tools.

Level Zero: Introduction

The oneAPI Level Zero provides a low-level, direct-to-metal interface for the devices in a oneAPI platform. Level Zero provides support for broad language features in addition to fine-grained explicit controls/APIs for device discovery, memory allocation, inter-process communication, kernel submission, synchronization, and metrics reporting. It has an API that exposes both the logical and physical abstractions of the underlying devices. While heavily influenced by other low-level APIs (i.e., OpenCL™), Level Zero is designed to evolve independently. It has support for GPUs and other compute devices, such as FPGAs. Most applications should not require the additional control provided by the Level Zero API. It is intended for the explicit controls needed by higher-level runtime APIs and libraries:

- Device discovery and partitioning
- Kernel execution and scheduling
- Peer-to-peer communication
- Metrics discovery and profiling
- Kernel profiling, instrumentation
- System management, query power, performance

The Level Zero C APIs are provided to applications by a shared import library. So, C/C++ applications must include "`ze_api.h`" and link with "`ze_api.lib`" (or a shared library).

Level Zero: APIs

Level Zero APIs are categorized into Core, Tools, and System Programming, but we will only discuss the Core Programming APIs (Figure 2) in this article. It has support for devices, drivers, contexts, memory, command queues/lists, synchronization, barriers, modules, and kernels. Tables 1 and 2 list the most commonly used APIs. Figure 3 shows the execution flow using Level Zero.

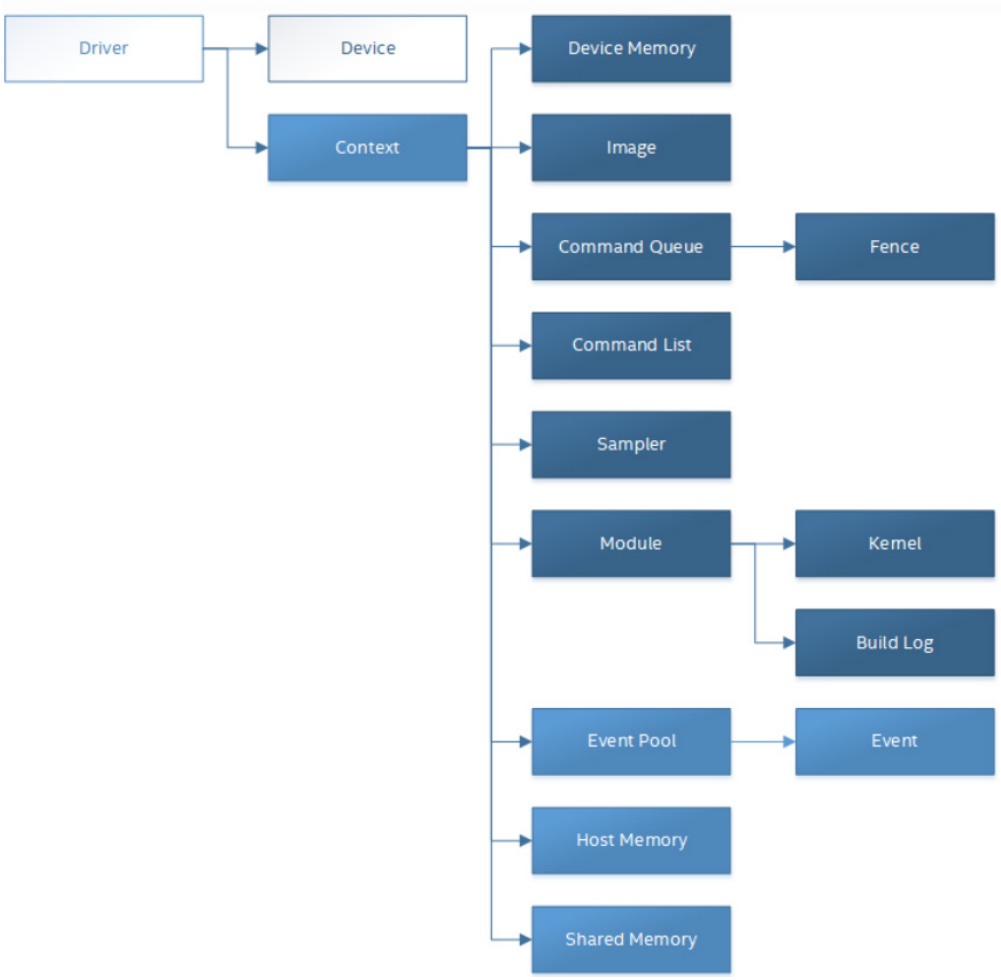


Figure 2. Components of the Core Programming APIs

APIs for Device, Context, Queue	Short description
`zeInit, zeDriverGet`	Initialize and discover all the drivers.
`zeDeviceGet, zeDeviceGetProperties`	Find a driver instance with a DEVICE_TYPE
`zeContextCreate`	Create a context
`zeMemAllocHost, zeMemAllocDevice, zeMemAllocShared`	Allocate memory on Host, Device or shared
`zeCommandQueueCreate`	Create a command queue
`zeCommandListCreate`	Create a command list
`zeCommandQueueExecuteCommandLists`	Execute command list in command queue
`zeCommandQueueSynchronize`	Synchronize host and device

Table 1. Level Zero APIs for Device, Context, and Queue

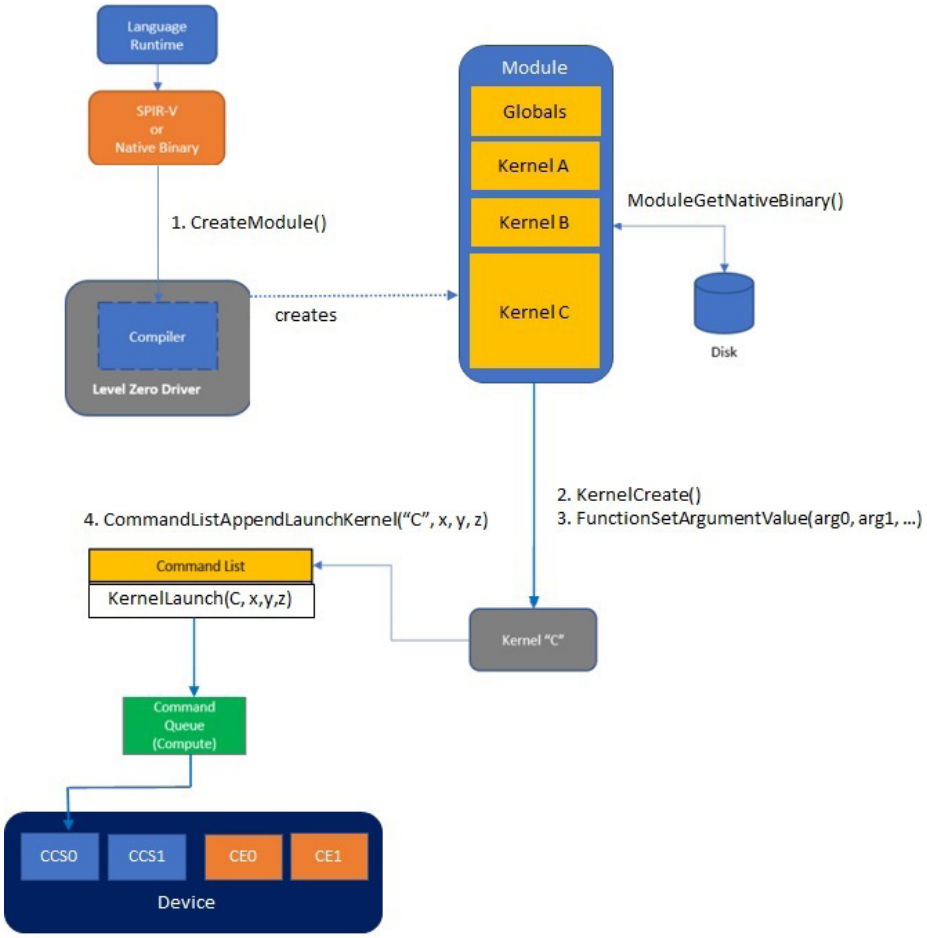


Figure 3. High-level flow: command lists, queues, module, and kernel execution on a device

APIs for Synchronization, Modules, and Kernels	Short description
zeEventPoolCreate	Create event pool
zeCommandListAppendLaunchKernel	Appends the kernel, its arguments, signals to a command list.
zeEventHostSynchronize	Wait on event to complete
zeModuleCreate	Creates a module by compiling IL code or load of a native binary
zeKernelCreate	Reference a kernel within a module
zeKernelSetArgumentValue	Setup arguments for kernel launch

Table 2. Level Zero APIs for synchronization and kernel-related functions

Listing 1 shows the `main` program that has driver and device discovery. Listing 2 shows kernel execution in function `RunTest`.

```
// Driver code (main)
int main(int argc, char *argv[])
{
   zeInit(ZE_INIT_FLAG_GPU_ONLY);

   ze_driver_handle_t driverHandle;
   zeDriverGet(&driverCount, &driverHandle);

   uint32_t deviceCount = 1;
   ze_device_handle_t device;
   zeDeviceGet(driverHandle, &deviceCount, &device);

   ze_device_properties_t deviceProperties = {};
   zeDeviceGetProperties(device, &deviceProperties);

   uint32_t subDeviceCount = 0;
   zeDeviceGetSubDevices(device, &subDeviceCount, nullptr);
   ze_device_handle_t subDevices[2] = {};
   zeDeviceGetSubDevices(device, &subDeviceCount, subDevices);

  for (uint32_t i = 0; i < subDeviceCount; i++) {
      ze_device_properties_t deviceProperties = {};
      zeDeviceGetProperties(subDevices[i], &deviceProperties);
   }

   RunTest(driverHandle, subDevices, device, subDeviceCount, outputValidBool);
}
```

Listing 1. Level Zero example for driver and device discovery

```
void RunTest(ze_driver_handle_t &driverHandle, ze_device_handle_t *subDevice,
             ze_device_handle_t rootDevice, uint32_t subDeviceCount,
             bool &validRet)
{
  // variables initialization, host memory allocation
  ...
  // create a context, command queue and list.
  zeContextCreate(driverHandle, &contextDesc, &context);
  ...
  for (uint32_t i=0; i<num_tiles; i++)
  {
    zeCommandQueueCreate(context, subDevice[i], &cmdQueueDesc, &cmdQueue[i]);
    zeCommandListCreate(context, subDevice[i], &cmdListDesc, &cmdList[i]);
  }
  ...
  // load the IL file (SPIRV format) which has the kernels to run on device
  const char *modulePath = "Gpu_Module_Kernel.spv";
  uint32_t spirvSize = 0;
  auto spirvModule = readBinaryFile(modulePath, spirvSize);
  ...
  zeModuleCreate(context, subDevice[i], &moduleDesc, &module[i], nullptr);
  zeKernelCreate(module[i], &kernelDesc, &kernel[i]);
  zeKernelSetGroupSize(kernel[i], groupSizeX, groupSizeY, groupSizeZ);
  ...
  // allocate device memory, append memory Copy instruction to the command list.
  for (uint32_t i=0; i<num_tiles; i++)
  {
    zeMemAllocDevice(context, &deviceDesc, bufferWidth*sizeof(float), 0,
                     subDevice[i], &d_input[i]);
    zeCommandListAppendMemoryCopy(cmdList[i], d_input[i], input[i],
                     bufferWidth*sizeof(float), nullptr, 0, nullptr);
    ...
  }
    // Copy data from host to device (execute the commands to allocate, copy).
  for (uint32_t i=0; i<num_tiles; i++) {
    zeCommandListClose(cmdList[i]);
    zeCommandQueueExecuteCommandLists(cmdQueue[i], 1, &cmdList[i], nullptr);
  }
  for (uint32_t i=0; i<num_tiles; i++) {
    zeCommandQueueSynchronize(cmdQueue[i], UINT32_MAX);
  }
  for (uint32_t i=0; i<num_tiles; i++) {
    zeCommandListReset(cmdList[i]);
  }
  ...
  // Set the kernel arguments
  for (uint32_t i=0; i<num_tiles; i++)
  {
    arg_indx = 0;
    start_idx = i * segment_size;

    zeKernelSetArgumentValue(kernel[i], arg_indx++, sizeof(d_input[i]),
                             &d_input[i]);
    zeKernelSetArgumentValue(kernel[i], arg_indx++, sizeof(d_input[i]),
                             &d_input[i + 1 == num_tiles ? 0 : i + 1]);
    ...
  }
  // Create an event pool, append it to the kernel launch command.
  ze_event_pool_handle_t eventPool;
  zeCommandListAppendLaunchKernel(cmdList[i], kernel[i], &group_count,
                                  kernelTsEvent[i], 0, nullptr);
  ... //continued
```

Listing 2. Level Zero example for kernel execution

```
  // Execute the command list, synchronize commands execution in the Queue.
  for (uint32_t i=0; i<num_tiles; i++) {
    zeCommandListClose(cmdList[i]);
    zeCommandQueueExecuteCommandLists(cmdQueue[i], 1, &cmdList[i], nullptr);
  }
  for (uint32_t i=0; i<num_tiles; i++) {
    zeCommandQueueSynchronize(cmdQueue[i], UINT32_MAX);
  }
  for (uint32_t i=0; i<num_tiles; i++) {
    zeCommandListReset(cmdList[i]);
  }
  // Get kernel event stats, compute execution duration.
  for (uint32_t i=0; i<num_tiles; i++)
  {
    zeEventQueryKernelTimestamp(kernelTsEvent[i], &kernelTsResults);
    uint64_t kernelDuration = kernelTsResults.context.kernelEnd -
                              kernelTsResults.context.kernelStart;
  }
  // Copy data from device to host.
  for (uint32_t i=0; i<num_tiles; i++)
  {
    zeCommandListAppendMemoryCopy(cmdList[i], output[i], d_output[i],
                    bufferWidth*sizeof(float), nullptr, 0, nullptr);
  }
  ...
  // Tear down, destroy the kernel, memory, context, event, and other objects.
  zeEventPoolDestroy(eventPool);
  for (size_t i=0; i<num_tiles; i++) {
    zeMemFree(context, d_input[i]);
    zeKernelDestroy(kernel[i]);
    zeCommandListDestroy(cmdList[i]);
    ...
  }
  zeContextDestroy(context);
  ...
}
```

Listing 2 (continued) Level Zero example for kernel execution

OpenMP Example

Level Zero APIs are also generated in the backend when compiling OpenMP offload code. These API calls are dumped when environment variables `LIBOMPTARGET_DEBUG` and `LIBOMPTARGET_INFO` are set to one or more. We show an example, in Listing 3, of an AoS (array-of-structures) being allocated on the device, and data initialized on the host copied to the memory allocated on the device, updated, and transferred back to the host. The logs were generated by setting the two environment variables mentioned above to 99.

Listing 4 shows some of the Level Zero API calls seen in the logs. The **`zeMemAllocDevice`** call allocates data on the device, followed by the **`zeCommandListAppendMemoryCopy`** call that copies data from the host to the device. Once the kernel computation is finished, the second call to **`zeCommandListAppendMemoryCopy`** copies the updated data from device to host. The AoS is then deleted with a call to **`zeMemFree`**, after which the device memory is returned to the memory pool. These

calls are similar to what we saw in Listing 2. We have only focused on the Level Zero calls that correlate directly to the OpenMP pragmas. The logs will show many more calls that are used to copy the pointers from host to the device, get memory block properties, map the host to the device pointer and clean up the device when done. In addition to this, the Level Zero API also provides other calls that give more control over how the memory is allocated, copied from host to device, or shared between the two.

```
struct force_data
{
  float Mass;
  int index;
};

#pragma omp declare target
struct force_data *myData;
#pragma omp end declare target

int main()
{
  int a[10], max=10;

  //allocate array of struct on host
  myData = (struct force_data*) malloc(max  * sizeof(struct force_data));

  for(int i = 0; i < max; i++)
  {
    myData[i].index = 1; a[i] = 2;
  }

 //1. Allocate data on device
  #pragma omp target enter data map(alloc:myData[0:max])
  {
    //2. Update data on device
    #pragma omp target teams distribute parallel for map(to:a)
    for(int i=0; i < max; i++)
      myData[i].index = myData[i].index + a[i];
  }

//3. Delete data on device
  #pragma omp target exit data map(delete:myData[0:max])
  for(int i=0; i < max; i++)
     printf("%d\n", myData[i].index);
}
```

Listing 3. OpenMP offload example

```
Libomptarget (pid:6780)  --> Entering OpenMP data region at unknown:31:31 with 1 arguments:
Libomptarget (pid:6780)  --> alloc(myData[0:10])[80]
…
Target LEVEL0 RTL (pid:10428)  --> ZE_CALLER: zeMemAllocDevice ( context, &deviceDesc,
Size, Align, Device, &mem )
…
Libomptarget (pid:10428)  --> Copying data from host to device, HstPtr=…., TgtPtr=…..,
Size=80, Name=myData[0:10]
…
Target LEVEL0 RTL (pid:10428)  --> Copy Engine is used for data transfer
Target LEVEL0 RTL (pid:10428)  --> ZE_CALLER: zeCommandListAppendMemoryCopy ( cmdList,
Dest, Src, Size, nullptr, 0, nullptr )
…
Libomptarget (pid:6780)  --> Entering OpenMP kernel at unknown:41:41 with 6 arguments:
Libomptarget (pid:6780)  --> tofrom(myData)[8] (implicit)
…

Libomptarget (pid:6780)  --> Updating OpenMP data at unknown:46:46 with 1 arguments:
Libomptarget (pid:6780)  --> from(myData[0:10])[80]

Libomptarget (pid:6780)  --> Copying data from device to host, TgtPtr=0xffffd556aa640000,
HstPtr=0x0000000000d362b0, Size=80, Name=myData[0:10]

Target LEVEL0 RTL (pid:6780)  --> Copy Engine is used for data transfer
Target LEVEL0 RTL (pid:6780)  --> ZE_CALLER: zeCommandListAppendMemoryCopy ( cmdList, Dest,
Src, Size, nullptr, 0, nullptr )

Libomptarget (pid:6780)  --> Deleting tgt data 0xffffd556aa640000 of size 80

Target LEVEL0 RTL (pid:6780)  --> Returned device memory 0xffffd556aa640000 to memory pool
Target LEVEL0 RTL (pid:6780)  --> ZE_CALLER: zeMemFree ( Context, (void *)block->Base )
Target LEVEL0 RTL (pid:6780)  --> ZE_CALLEE: zeMemFree (
Target LEVEL0 RTL (pid:6780)  -->     hContext = 0x0000000000d33500
Target LEVEL0 RTL (pid:6780)  -->     ptr = 0xffffd556aa640000
```

Listing 4. Level Zero API calls generated in the backend for OpenMP offload

Conclusions

Most application developers will not require the additional control provided by the Level Zero API. It is intended mainly for library and framework developers. The Level Zero API provides more fine-grained, explicit control over device discovery, memory management, kernel submission, inter-process communication and more. In this article, we have looked at a basic example to become familiar with Level Zero programming. The OpenMP offload example also provides some insights into the set of calls generated in the backend that provide a direct-to-metal interface to the offload accelerator device. The oneAPI Level Zero specification contains complete API details.

Hyperparameter Optimization with SigOpt for MLPerf Training on Habana Gaudi

Achieve Faster Convergence with Higher Accuracy in AI Training

Basem Barakat, Senior Engineer, Evelyn Ding, Senior Engineer, and Joshua Mora, Principal Engineer, Habana Labs an Intel® company

Optimizing the performance of large-scale, deep learning training workloads is expensive. It requires contributions from a multidisciplinary team, and it requires a large computational infrastructure designed for deep learning training. Optimization can be broken down into a wide range of methodologies: computation- and communication-related optimizations of the collective operations for the data parallel training paradigm, orchestration/scheduling of independent tasks, pipelining of preprocessing and augmentation of the dataset, lightweight algorithmic optimizations that increase the convergence rate, and numerical optimizations leveraging mixed precision of fp32 with bfloat16.

And then we have the optimization we will focus on in this article: a methodology based on hyperparameter optimization (HPO) to reduce the number of training epochs while preserving the desired target accuracy (the converged epoch). We will apply HPO to an MLPerf™ training workload on the Habana® Gaudi® training processor. This work is a collaboration between Habana Labs and SigOpt, both AI-focused Intel companies. The result of this work is improved model training time and reduced computational resources required to achieve optimal hyperparameters for the ResNet50 (RN50) model, resulting in reduced time-to-train the MLPerf model on top of grid search benefits, while using fewer Gaudi-hours with respect to the grid search approach.

MLPerf Training Workloads

Since 2018, the MLPerf benchmark suite has been used by the AI community to assess a wide range of neural network models running on different types of computing infrastructures. The neural networks are revisited at each submission to reflect the rapid evolution of AI.

We take advantage of the layer-wise adaptive rate scaling (LARS) algorithm (Figure 1) during RN50 training. The hyperparameters and their respective MLPerf constraints are listed in Table 1. For this workload, we must achieve a specific target accuracy (AC) of 75.9%.

Algorithm 1 LARS

Input: $x_1 \in \mathbb{R}^d$, learning rate $\{\eta_t\}_{t=1}^T$, parameter $0 < \beta_1 < 1$, scaling function ϕ, $\epsilon > 0$
Set $m_0 = 0$
for $t = 1$ **to** T **do**
  Draw b samples S_t from $\mathbb{P}$
  Compute $g_t = \frac{1}{|S_t|} \sum_{s_t \in S_t} \nabla \ell(x_t, s_t)$
  $m_t = \beta_1 m_{t-1} + (1 - \beta_1)(g_t + \lambda x_t)$
  $x_{t+1}^{(i)} = x_t^{(i)} - \eta_t \frac{\phi(\|x_t^{(i)}\|)}{\|m_t^{(i)}\|} m_t^{(i)}$ for all $i \in [h]$
end for

Figure 1. Pseudocode for LARS showing the hyperparameters used during HPO

Hyperparameters for RN50	Symbol	Type	Constraint
Number of Training Epochs	NTE	INT	Positive INT
Number of Warmup Epochs	NWE	INT	Positive INT
Base Learning Rate	BLR	FLOAT	Positive real
Weight Decay	WD	FLOAT	0.0001×2^N, N INT
Momentum	MM	FLOAT	Positive real, depends on global batch size

Table 1. Hyperparameters and MLPerf constraints

Integration of Large-Scale Training and HPO Workflow Processes

Figure 2 shows a conceptual representation of the training cluster that could be on-premise or cloud-based. It is being accessed by an AI user who submits batches of training jobs to the computing infrastructure. The user gets back the accuracy on the runs (AC) and the number of converged epochs (CE). That information is fed to SigOpt, which responds back to the user with new hyperparameter suggestions. The user can program the criteria to continue, refine or stop the hyperparameter search once the optimization objective of finding the lowest possible convergence epoch while reaching the target accuracy is met.

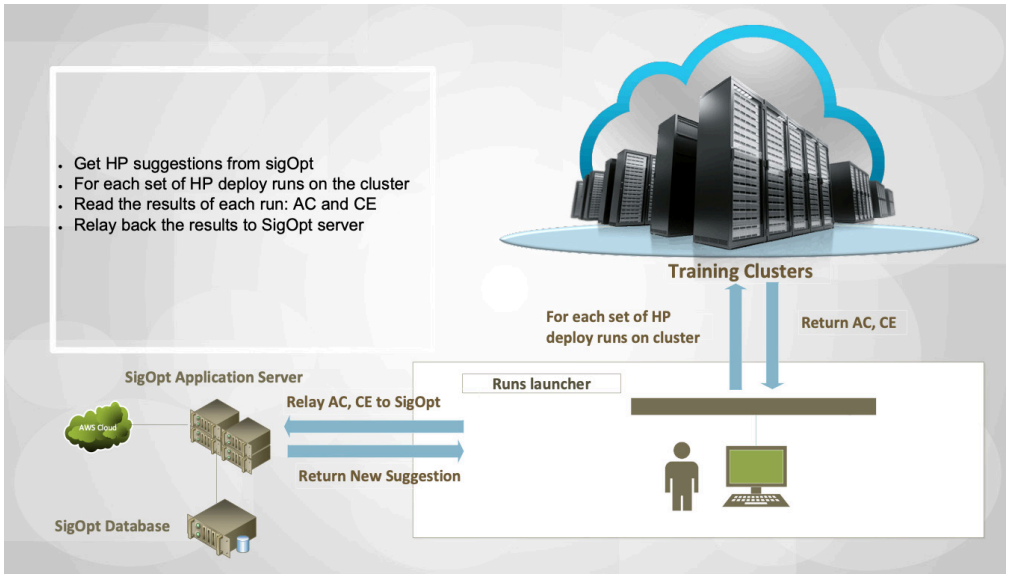


Figure 2. Running training jobs in the training clusters with SigOpt and the AI user controlling the HPO progress

HPO Workflow with SigOpt

Figure 3 illustrates our current HPO implementation of the workflow with SigOpt. SigOpt is started by defining the following variables:

- Initial boundaries of hyperparameters
- Threshold of evaluation metrics (e.g., AC)
- Experiment budgets (i.e., the number of runs we can afford within a time budget)
- Computational resources (i.e., the number of compute nodes to use)
- Parallel run budgets (i.e., the number of concurrent runs to speed-up the hyperparameter search within the allowed computational resources).

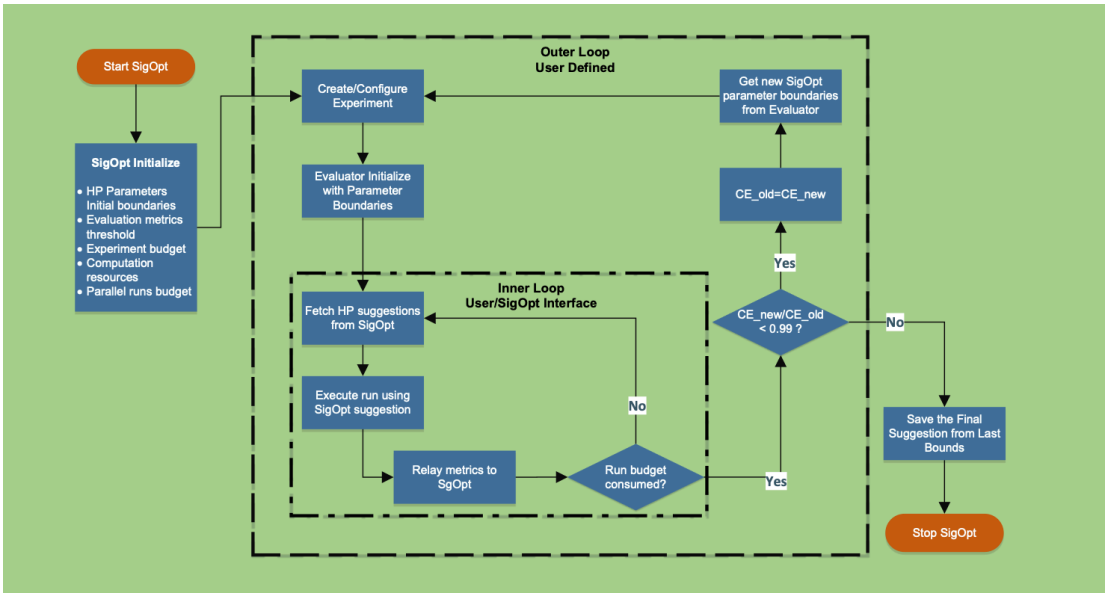


Figure 3. Workflow of HPO with SigOpt to find the HPs that meet target accuracy and minimize the number of epochs to converge.

The workflow consists of two nested loops with a set of building blocks with specific functionality: an inner loop (the dashed-dotted lines) and an outer loop (the dashed lines). After SigOpt is started, a new experiment is created and configured, and an evaluator algorithm is initialized with the hyperparameter boundaries. Then, the inner loop starts with suggested hyperparameter values provided by SigOpt, which uses the metrics from user training deep learning models on the training clusters to give recommendations. The inner loop completes its process once the predefined budget is reached.

Next, the outer loop starts by using the evaluator building block to validate the SigOpt hyperparameter suggestions and refines the boundaries based on user criteria. This allows you to bring your own additional optimizations on top of what SigOpt already provides. The evaluator we implemented leverages

a K-means unsupervised machine learning clustering method, which will be described in the next section.

Once the hyperparameter boundaries are refined, the outer loop can resume its process for another iteration or stop when there is no improvement on the convergence epoch. The final suggestions of hyperparameter values are saved and the evaluation is stopped.

Bringing Your Own Evaluator into the HPO Workflow

Figures 4-7 illustrate the implementation of our evaluator, which is based on K-means clustering. For all data received from SigOpt, only the data meeting the accuracy requirement are saved, and these data are further filtered by keeping only 75% of the datapoints with the best converge epochs (Figure 4). Then, the K-means classification is applied to the suggested part of the datapoints (namely: NTE, BLR, NWE and WD) to separate them into clusters with each having at least ten datapoints (Figure 5). The cluster with the best performance is selected based on the minimum mean runtime (the blue dashed box in Figures 5 and 6).

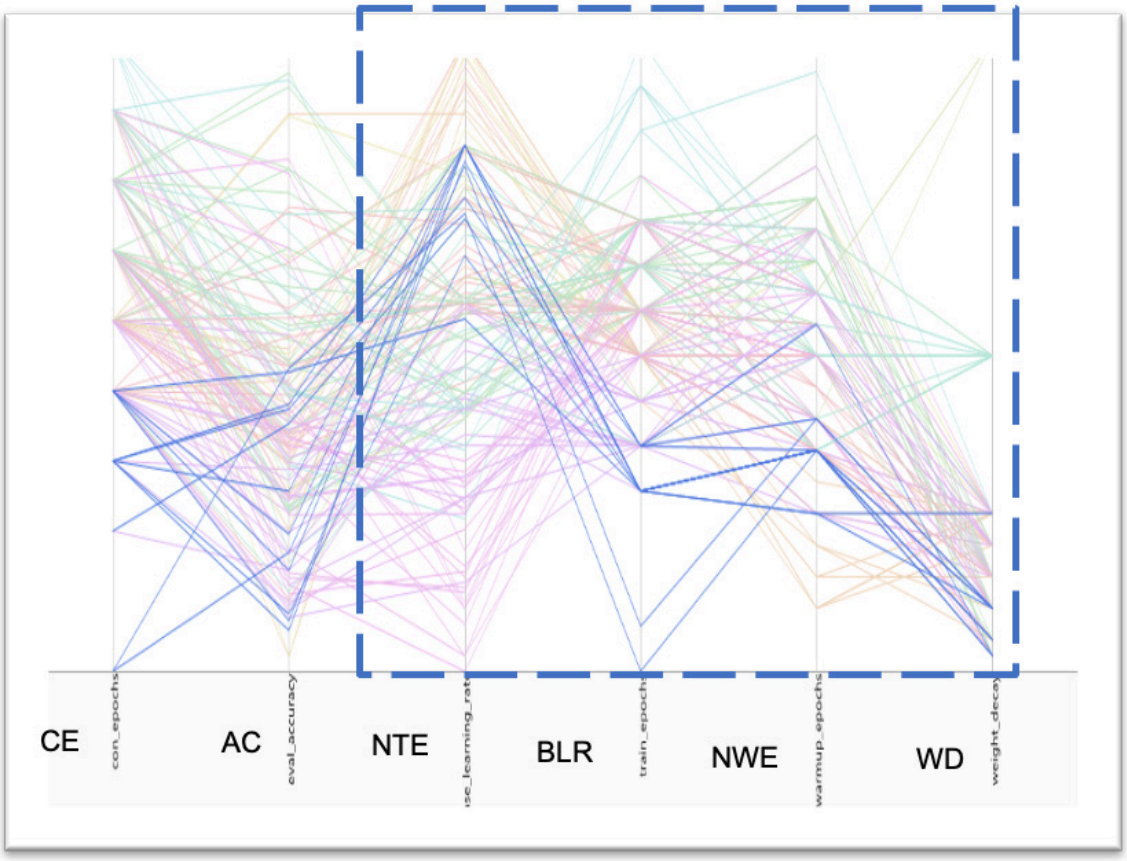


Figure 4. K-means clustering is applied to the parameter space of the measured points to group them into distinct clusters.

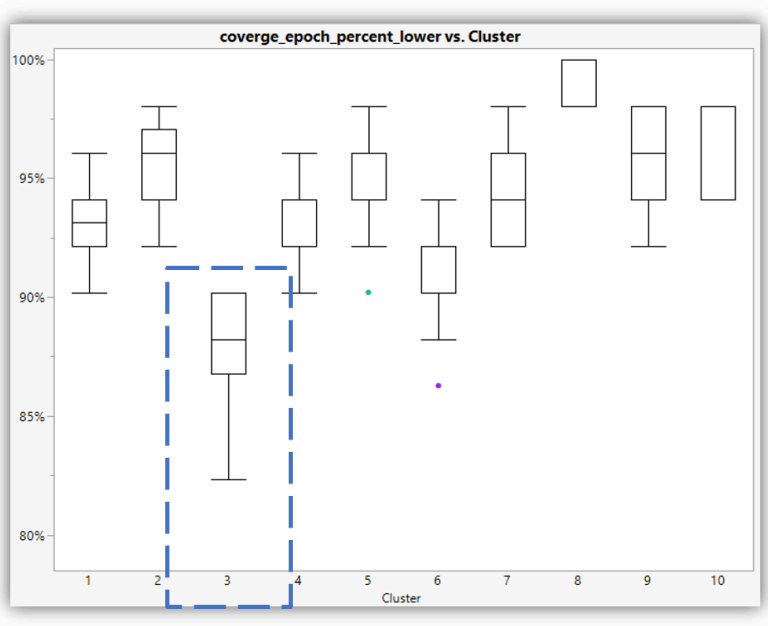


Figure 5. For each set of measurement in given cluster, the mean converge epoch is calculated. The blue square denotes the selected cluster with lowest mean converge epoch.

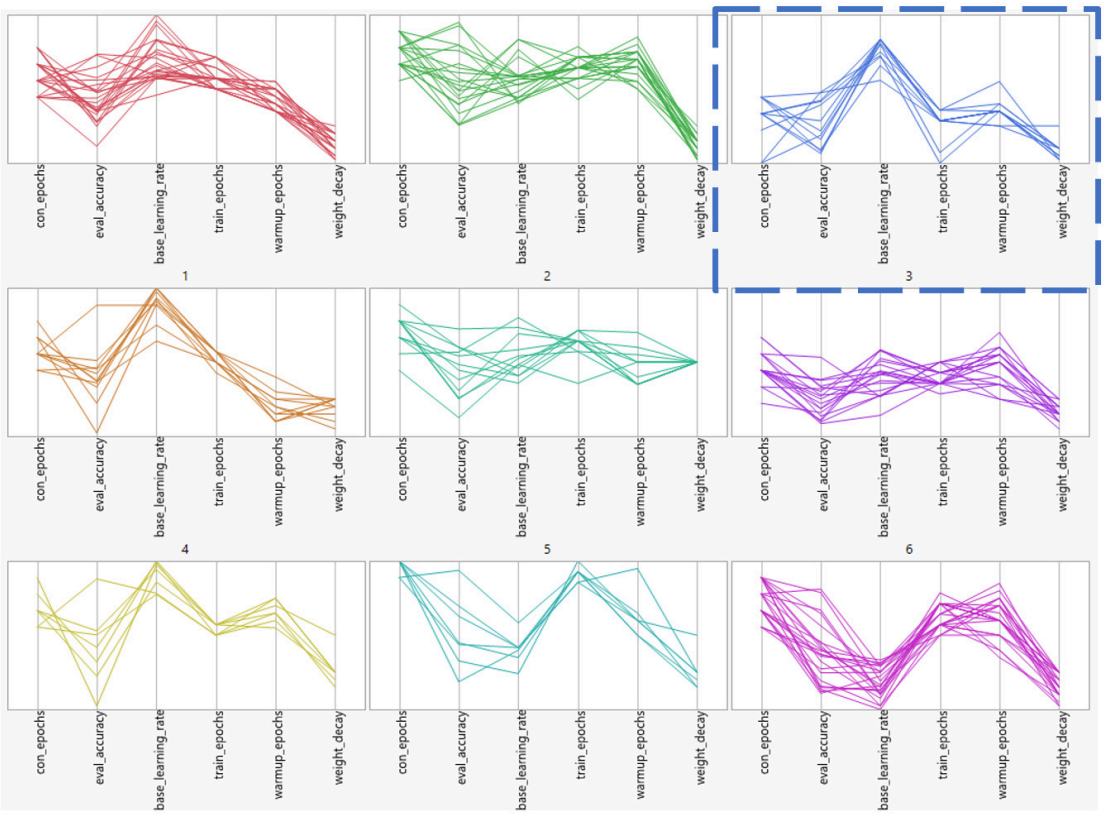


Figure 6. Measurement results clustered using the k-mean algorithm are plotted above. The selected cluster in Figure 5 is indicated by the blue square.

For a given hyperparameter, the range of values (boundaries) from the best cluster (with the lowest runtime/converged epochs) is compared with the SigOpt predefined boundaries (Figure 7). If any of the predefined boundaries (left and right in the range) are too close to any of the best cluster values found, the evaluator will adjust the corresponding SigOpt boundaries to allow the exploration of new hyperparameter values. On the other hand, if any of the SigOpt boundaries are far away from the best cluster values found, the evaluator will adjust the corresponding SigOpt boundaries to be closer to the best cluster hyperparameter value to allow for exploitation within the vicinity of that value. The 0.1 (10%) adjustment to the coefficient in Figure 7 is found empirically. We used 10% for RN50. The new adjusted SigOpt boundaries are then reconfigured for a new experiment. This adjustment speeds up the search of hyperparameter values. It is depicted graphically in Figure 8 for the learning rate hyperparameter.

Due to the initial condition randomness of the deep learning framework, we executed each run several

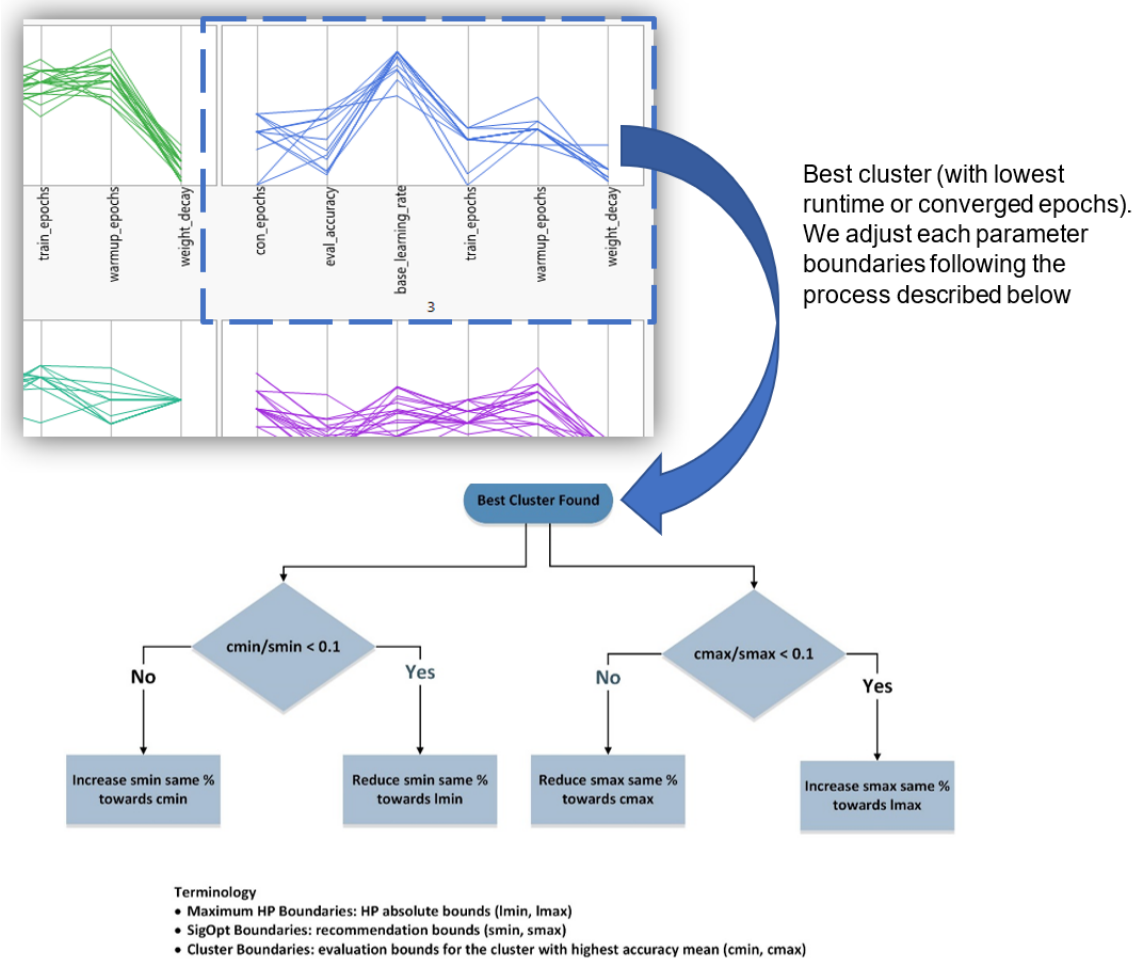


Figure 7. Illustration of the evaluator algorithm

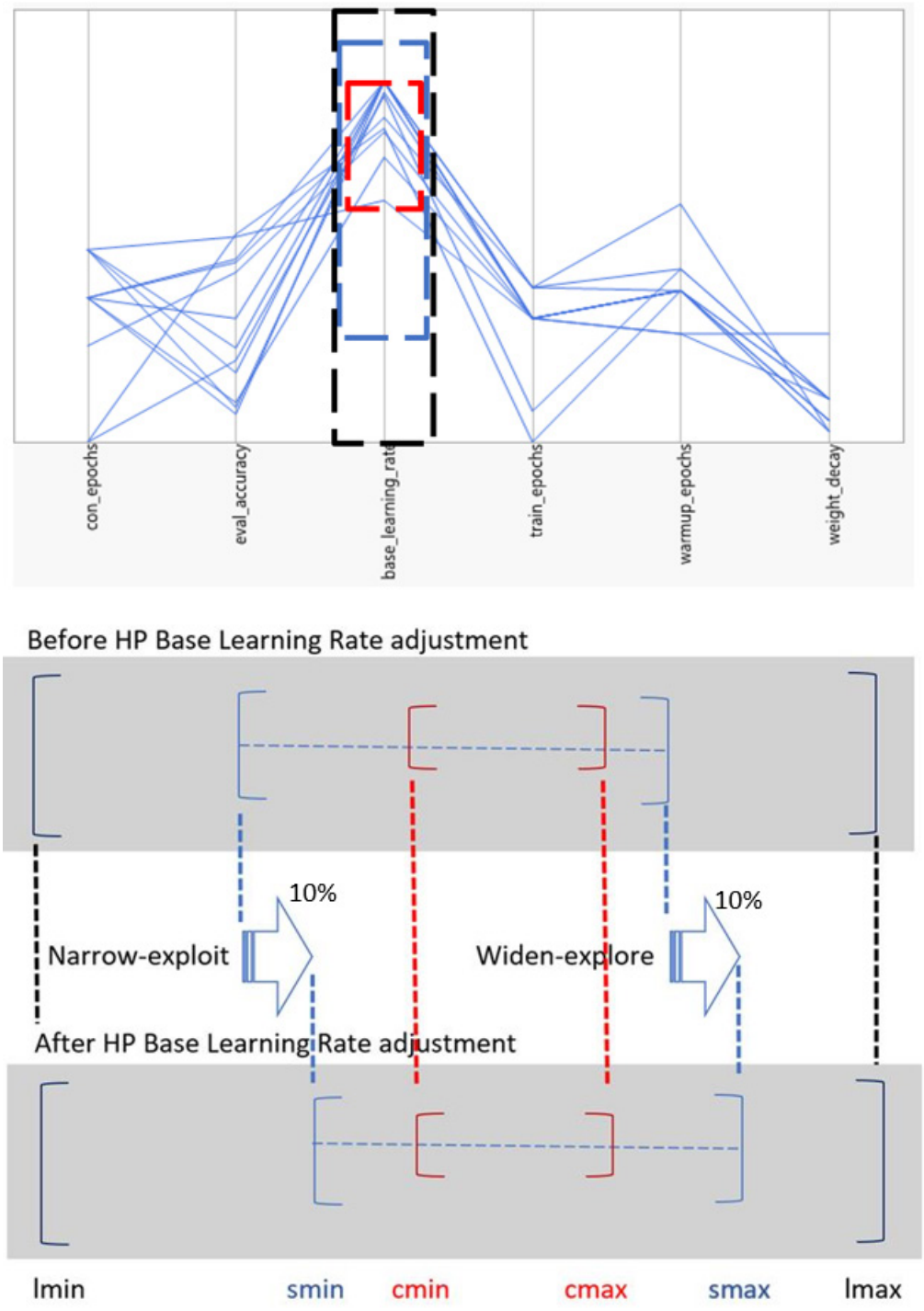


Figure 8. Hyperparameter value adjustment based on the best cluster hyperparameters

times with different random seeds for the parameter values. Then, we report the average and standard deviation for each metric to SigOpt. To speed up the search time and conserve compute resources we perform three runs for each data point during the search process. However, at end of the experiment we select the best set of points found by SigOpt and futher validate them with higher run counts (5-10 runs). This process will ensure that the results are stable and reproduceable. Running the optimization for a given global batch size also conserves compute resources because we are simultaneously covering a concurrent set of deployment senarios, each with different computing resources that share the same hyperparameters across different local batch sizes.

SigOpt Dashboard Charts

The charts embedded in the SigOpt dashboard such as parallel coordinates, contribution/sensitivity, and experiment history provide a set of complementary views that enrich the understanding of hyperparameters and their effects on convergence during an experiment. For RN50, we did 400 runs. The parallel coordinate plot (Figure 9) shows the relations between the hyperparameters suggested by SigOpt and the measured metrics (results) of these runs. The gray lines denote those runs that did not pass the target evaluation accuracy (SigOpt refers to it as the threshold), while the blue lines denote the runs that met the threshold. SigOpt highlights the best runs in orange (Figures 9 and 10). The parallel coordinate plots show correlations among the hyperparameters plus the accuracy and number of epochs to converge. For example, one can easily see that base_learning_rate and weight_decay are inversely correlated with the number of converged epochs.

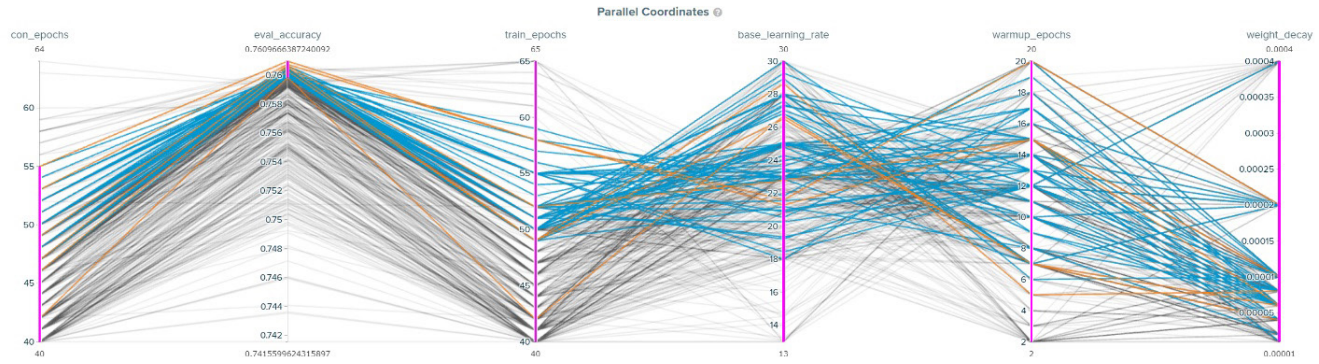


Figure 9. RN50 parallel coordinate plot showing the relation between HPO and measured metric values

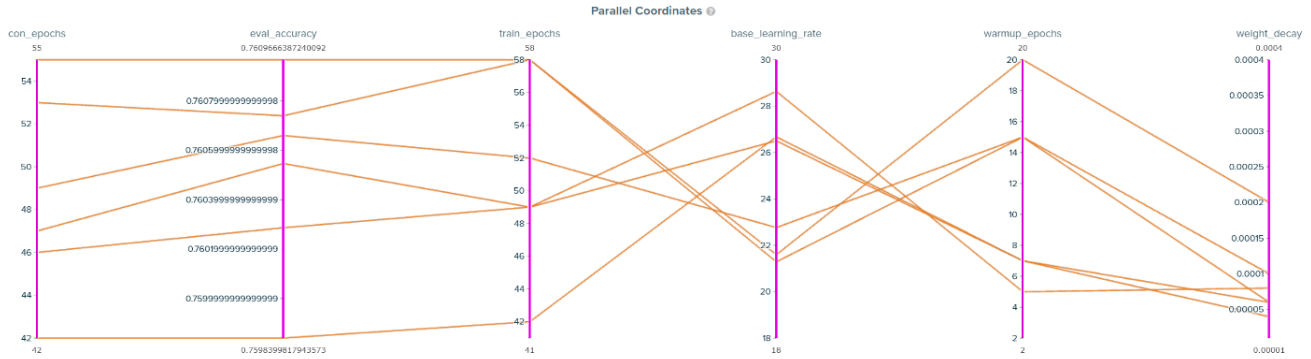


Figure 10. RN50 parallel coordinates for the best observations

Figure 11 shows the strong correlation between train_epochs and con_epochs. The con_epochs is simply the train_epochs number at which the target evaluation_accuracy is reached. As such, we expect a strong correlation. We can also see that con_epochs is approximately 1-3 epochs lower than train_epochs.

Figure 11. RN50 correlation between train_epochs and con_epochs

Figure 12 shows the contribution of each hyperparameter toward each of the metrics values. The orange and the blue bars show the contributions to eval_accuracy and con_epochs, respectively. The scale for the ranking is between 0 and 1 (100%). SigOpt uses the decision tree regressor model to measure these relations. In this figure, and as expected intuitively, we can see that train_epochs has the most impact on accuracy and con_epochs. Also, we can see that the weight decay parameter only impacts the con_epochs values, while base_learning_rate only impacts evaluation accuracy.

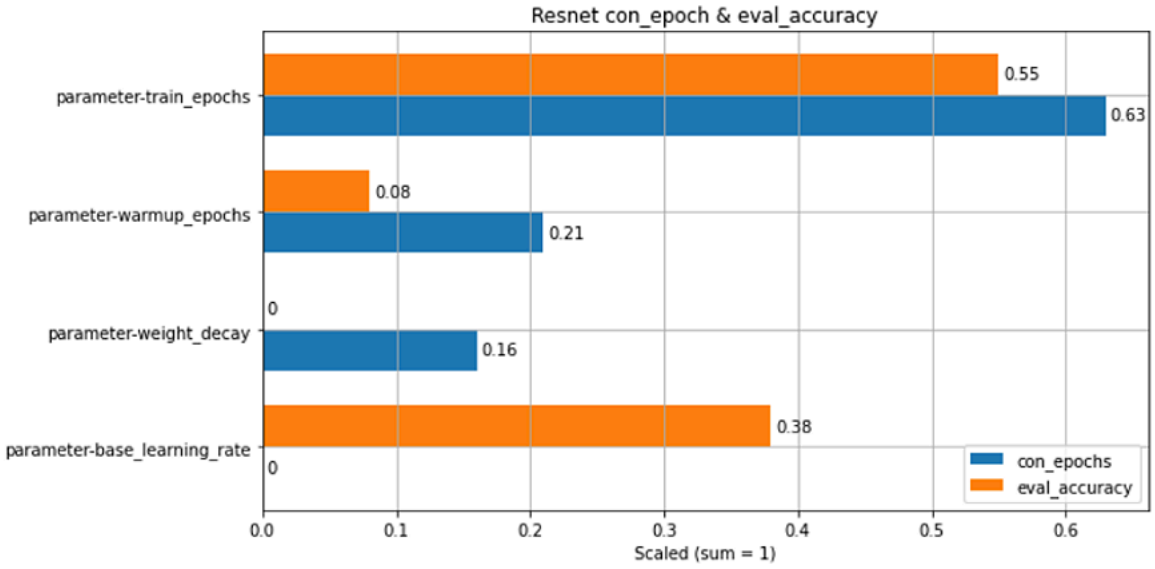


Figure 12. Relative contribution of the hyperparameter toward the metrics results for RN50. The scale is based on cumulative contribution of 100%.

Figure 13 is the main tracking plot in the SigOpt dashboard showing a scattered plot between all the metrics values. This plot updates during the experiment to show how the SigOpt optimizer is reducing con_epochs while improving evaluation accuracy. The best points are highlighted in orange. The best result is indicated by the red box.

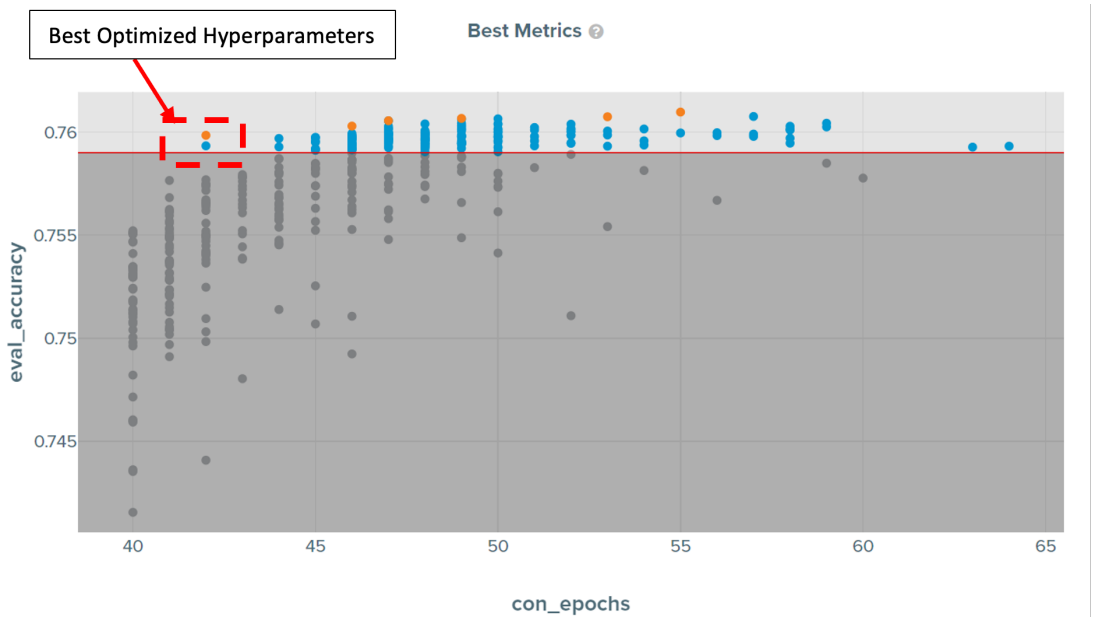


Figure 13. Correlation of best metrics

SigOpt HPO Results for MLPerf RN50

We start by providing the values used in HPO for both workloads. These are the hyperparameter values passed to MLPerf RN50 and the stopping criteria for the HPO workflow:

Hyperparameter	Symbol	Type	Constraint
Number of Training Epochs	NTE	INT	41 ~ 50
Number of Warmup Epochs	NWE	INT	2 ~ 15
Base Learning Rate	BLR	FLOAT	18 ~ 30
Weight Decay	WD	FLOAT	1.0e-5 ~ 4.0e-4

Metric	Symbol	Type	Objective
Evaluation Accuracy	AC	FLOAT	> 0.759
Converge Epoch	CE	INT	Minimize

We close this section by listing the advantages of SigOpt over grid search found during MLPerf RN50 HPO:

- Found better hyperparameters with lower converge epochs (CE)
  - Bayesian vs grid search
  - 6% epoch reduction relative to grid search, which translates to ~6% lower training time
- More efficient resource utilization

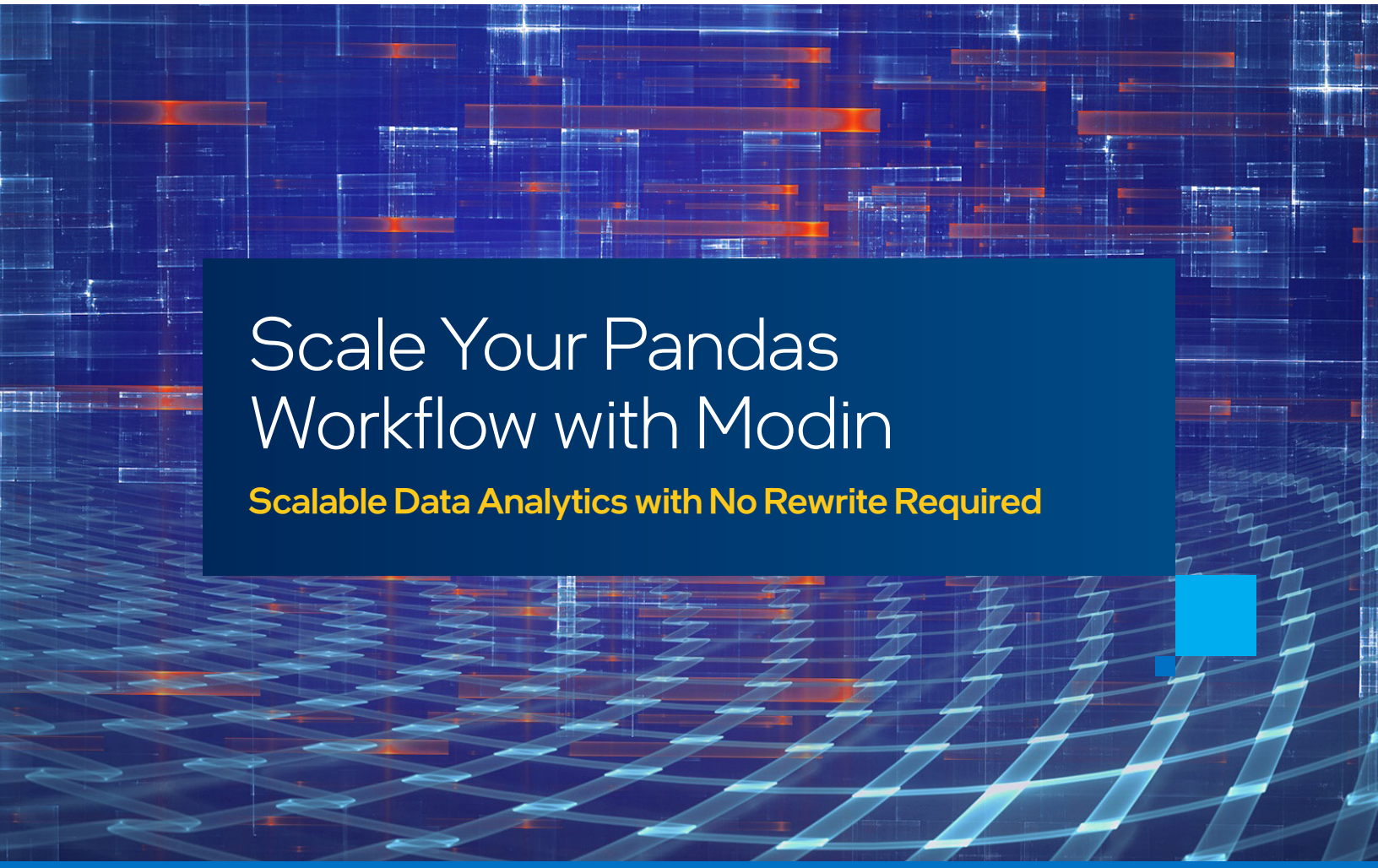
The following tables summarize the computational resources consumed during HPO and the runtime reduction achieved and expressed in terms of reduction of the convergence epochs.

Cluster Runs	Grid Search Compute Resource Utilization (Hours)	SigOpt Search Compute Resource Utilization (Hours)
K8s runs	53,948	21,333
Baremetal runs	31,464	0
Total runs	85,413	21,333

	Grid Search	SigOpt Search
Epoch reduction	28%	+6% (in addition to grid search)

Notice that the SigOpt search was performed after the grid search effectively starting from an already optimized search point. It was able to lower the converge epochs values by an additional 6% using less compute resources: 21,333 hours vs. 85,413 hours for SigOpt and Grid search, respectively.

Given these results, we hope AI developers will take advantage of the cost efficiency of Habana Gaudi and leverage SigOpt's HPO to accelerate model development.



Scale Your Pandas Workflow with Modin

Scalable Data Analytics with No Rewrite Required

Vasilij Litvinov, AI Frameworks Engineer, Intel Corporation

[This article originally appeared on Anaconda.com and is reprinted with permission.]

AI and data science are rapidly advancing, which brings an ever-increasing amount of data to the table and enables us to derive ideas and solutions that continue to grow more complex every day. But on the other hand, we see that these advances are shifting focus from value extraction to systems engineering. Also, hardware capabilities might be growing at a faster rate than people can learn how to properly utilize them.

This trend either requires adding a new position, a so-called "data engineer," or requires a data scientist to deal with infrastructure-related issues instead of focusing on the core part of data science — generating insights. One of the primary reasons for this is the absence of optimized data science and machine learning infrastructure for data scientists who are not necessarily software engineers by nature – these can be considered two separate, sometimes overlapping skill sets.

We know that data scientists are creatures of habit. They like the tools that they're familiar with in the Python data stack, such as pandas, Scikit-learn, NumPy, PyTorch, etc. However, these tools are often unsuited for parallel processing or terabytes of data.

Anaconda® and Intel® are collaborating to solve data scientists' most critical and central problem: how to make their familiar software stack and APIs scalable and faster? This article introduces Intel Distribution of Modin, part of the Intel oneAPI AI Analytics Toolkit (AI Kit), which is now available from Anaconda's "defaults" channel (and from conda-forge, too).

Why Pandas Is Not Enough

Though it is an industry standard, pandas is inherently single-threaded for a lot of cases, which makes it slow for huge datasets. It may not even work for datasets that don't fit in memory. There are other alternatives to solve this issue (e.g., Dask, pySpark, vaex.io, etc.), but none of them provide a fully pandas-compatible interface – users would have to modify their workloads accordingly.

What does Modin have to offer you as the end-user? It tries to adhere to the idea of "tools should work for the data scientist, not vice versa." It offers a simple, drop-in replacement for pandas – you just change your "`import pandas as pd`" statement to "`import modin.pandas as pd`" and gain better scalability for a lot of use-cases.

What Modin Offers

By removing the requirement to "rewrite pandas workflow to X framework," it's possible to speed up the development cycle for data insights (Figure 1).

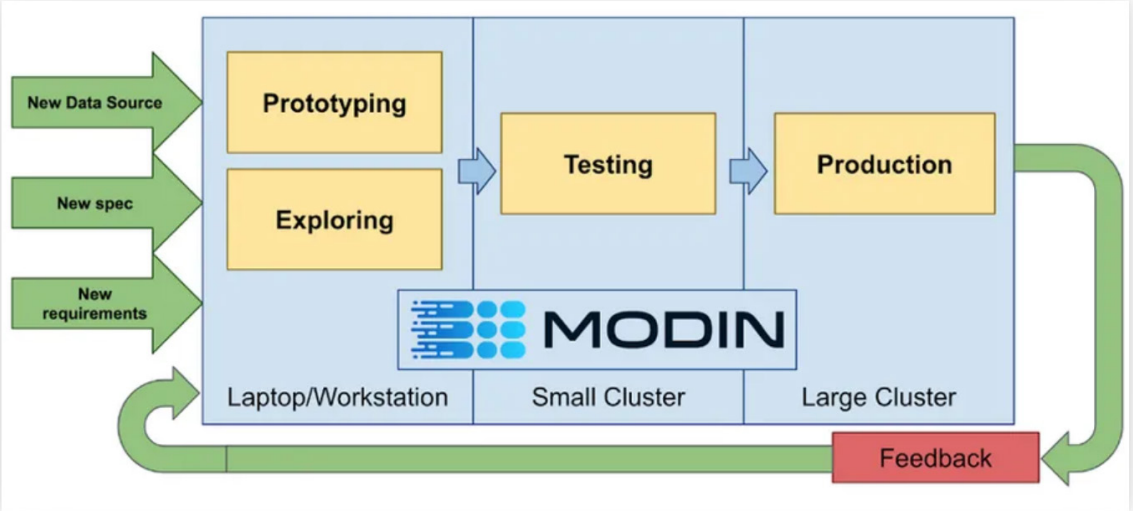


Figure 1. Using Modin in a continuous development cycle

Modin better utilizes the hardware by grid-splitting the dataframe, which allows certain operations to run in a parallel distributed way, be it cell-wise, column-wise, or row-wise (Figure 2). For certain operations, it's possible to utilize experimental integration with the OmniSci engine to leverage the power of multiple cores even better.

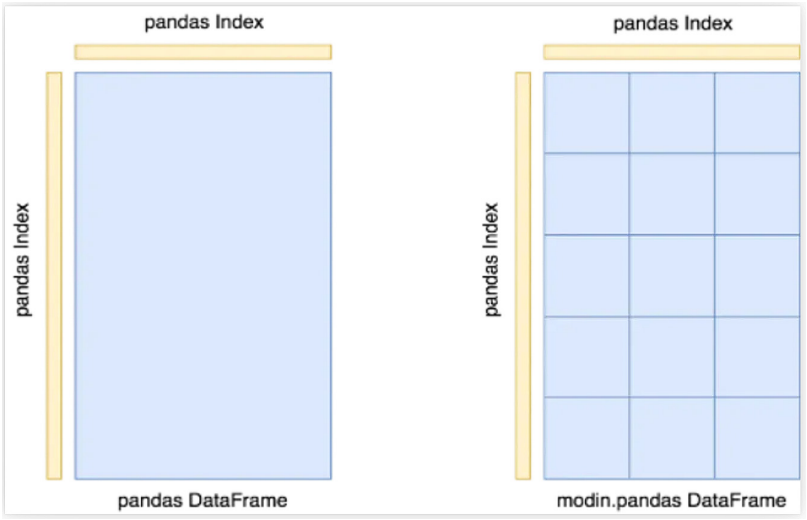


Figure 2. Comparing pandas and Modin DataFrames

By installing Modin through AI Kit or from the Anaconda defaults (or conda-forge) channel, an experimental, even faster OmniSci backend for Modin is also available. It just takes a few simple code changes to activate:

```
import modin.config as cfg
cfg.Engine.put('native')
cfg.Backend.put('omnisci')
import modin.experimental.pandas as pd
```

Show Me the Numbers!

Enough with the words, let's have a look at the benchmarks. For detailed comparison of different Modin engines, please refer to community-measured microbenchmarks: http://modin.org/modin-bench, which track performance of different data science operations over commits to the Modin repository.

To demonstrate the scalability of this approach, let's use a larger, more end-to-end workload running on an Intel Xeon® 8368 Platinum-based server (see full hardware info below) using OmniSci through Modin (Figures 3-5).

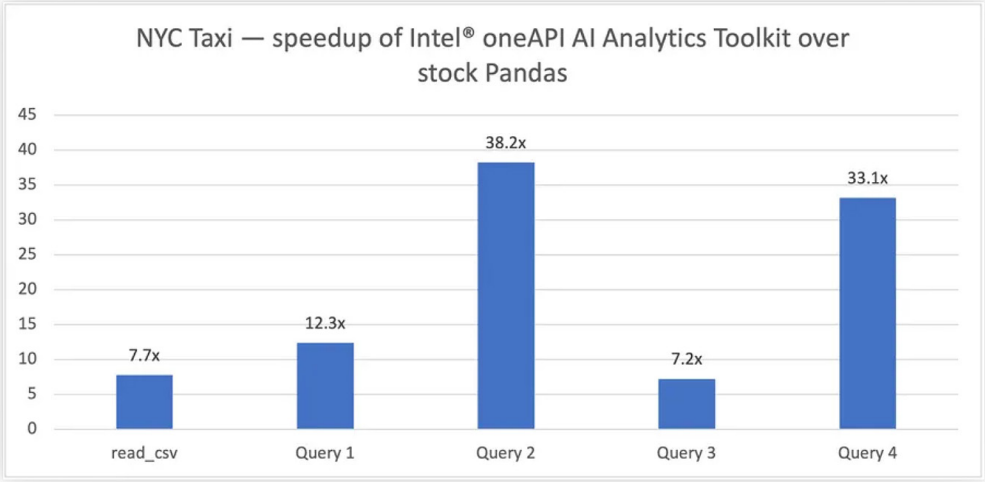


Figure 3. Running the NYC Taxi workload: 200M records, 79.2 GB input dataset

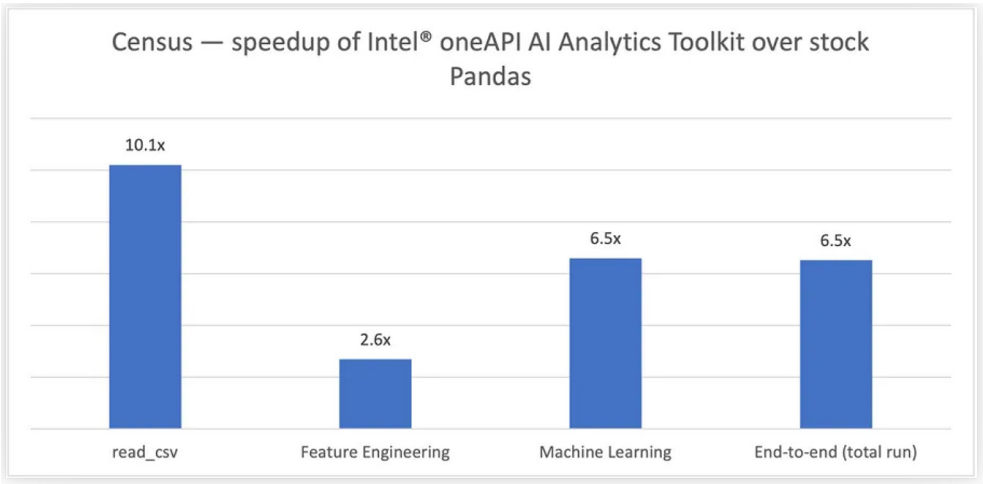


Figure 4 Running the Census workload: 21M records, 2.1 GB input dataset

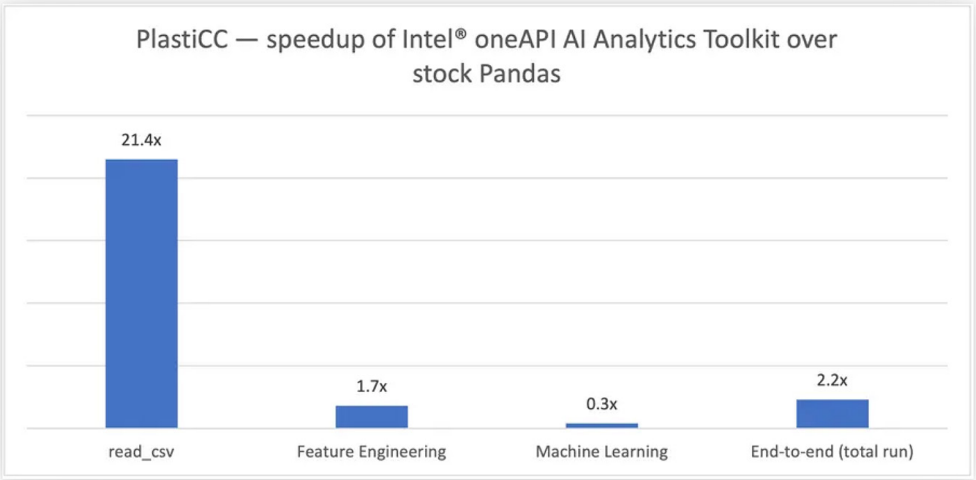


Figure 5. Running the PlastiCC workload: 460M records, 20 GB input dataset

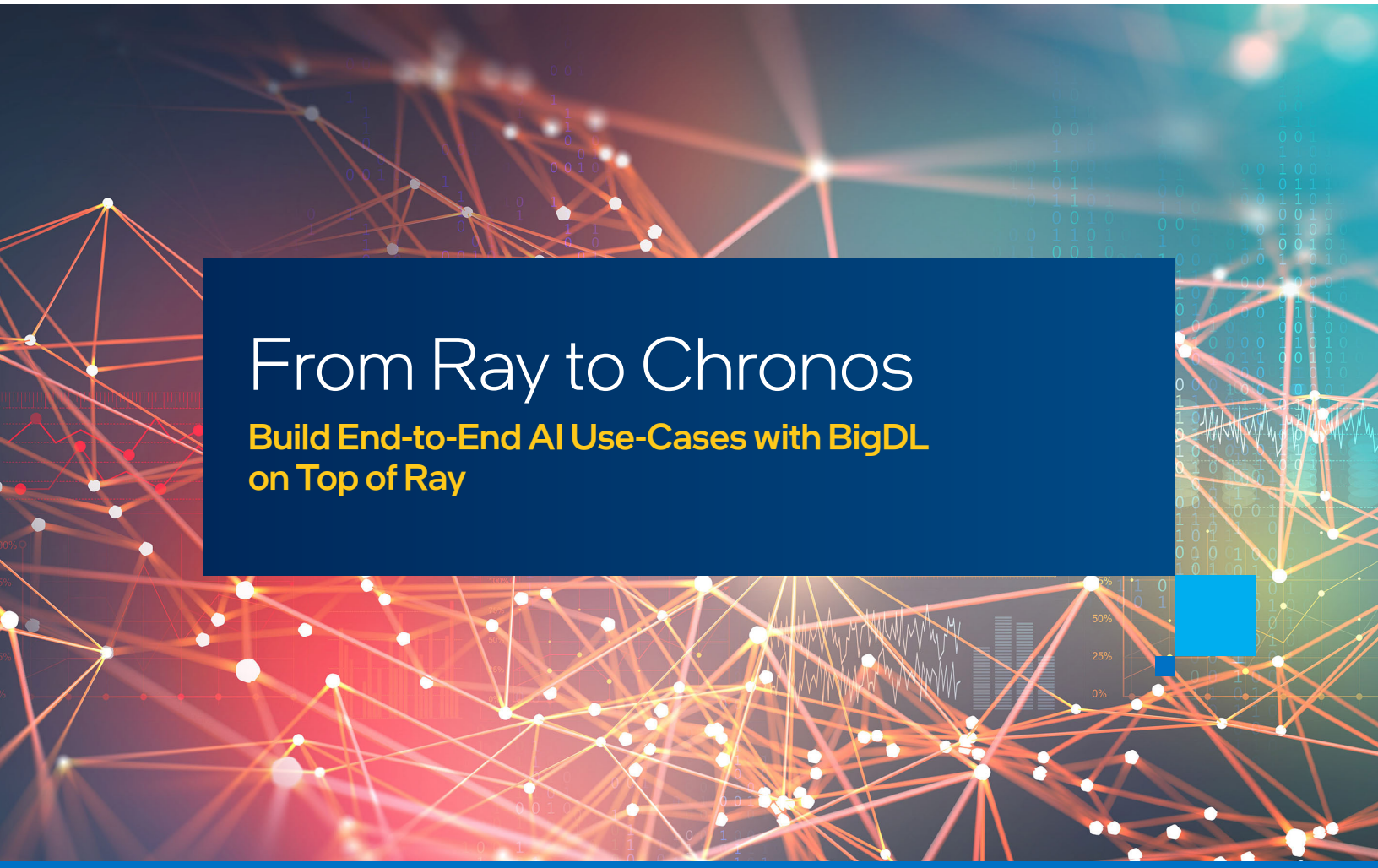
Hardware information: two 3rd Generation Intel Xeon 8368 Platinum on a C620 board with 512GB (16 slots/32GB/3200) total DDR4 memory, microcode 0xd0002a0, Hyper-Threading on, Turbo on, Centos 7.9.2009, 3.10.0-1160.36.2.el7.x86_64, one Intel 960 GB SSD OS Drive, three Intel 1.9 TB SSD data drives. Software information: Python 3.8.10, Pandas 1.3.2, Modin 0.10.2, OmnisciDB 5.7.0, Docker 20.10.8, tested by Intel on 10/05/2021.

Wait, There's More

If running on one node is not enough for your data, Modin supports running distributed on a Ray cluster or a Dask cluster. You can also use the experimental XGBoost integration, which will automatically utilize the Ray-based cluster for you without any special set up!

References

- Intel oneAPI AI Analytics Toolkit installation: `conda install intel-aikit -c intel`
- Modin Documentation
- Modin Source and Issue Tracker
- Switching Modin to the OmniSci Backend



From Ray to Chronos

Build End-to-End AI Use-Cases with BigDL on Top of Ray

Wesley Du, Junwei Deng, Kai Huang, Shan Yu, and Shane Huang, Solution Architects, Intel AI and Analytics

[This article originally appeared on anyscale.com and is reprinted with permission.]

Ray is a framework with simple and universal APIs for building innovative AI applications. BigDL is an open-source framework for building scalable end-to-end AI on distributed big data. It leverages Ray and its native libraries to support advanced AI use-cases such as AutoML and Automated Time Series Analysis. We will introduce some of the core components in BigDL and showcase how it takes advantage of Ray to build out the underlying infrastructure (i.e., RayOnSpark, AutoML, etc.), and how these will help users build AI applications using Project Chronos.

RayOnSpark: Seamlessly Run Ray Applications on Top of Apache Spark

Ray is an open-source distributed framework for easily running emerging AI applications such as deep reinforcement learning and automated machine learning (ML). BigDL seamlessly integrates Ray into big data preprocessing pipelines through RayOnSpark and it has already been used to build several advanced end-to-end AI applications for specific areas such as AutoML and Chronos. RayOnSpark runs Ray programs on top of Apache Spark™ on big data clusters (e.g., an Apache Hadoop™* or Kubernetes* cluster) and as a result, objects like in-memory DataFrames can be directly streamed into Ray applications for advanced AI applications. With RayOnSpark, users can directly try various emerging AI applications on their existing big data clusters in a production environment. It also allows Ray applications to seamlessly integrate into Big Data processing pipelines and directly run on in-memory DataFrames.

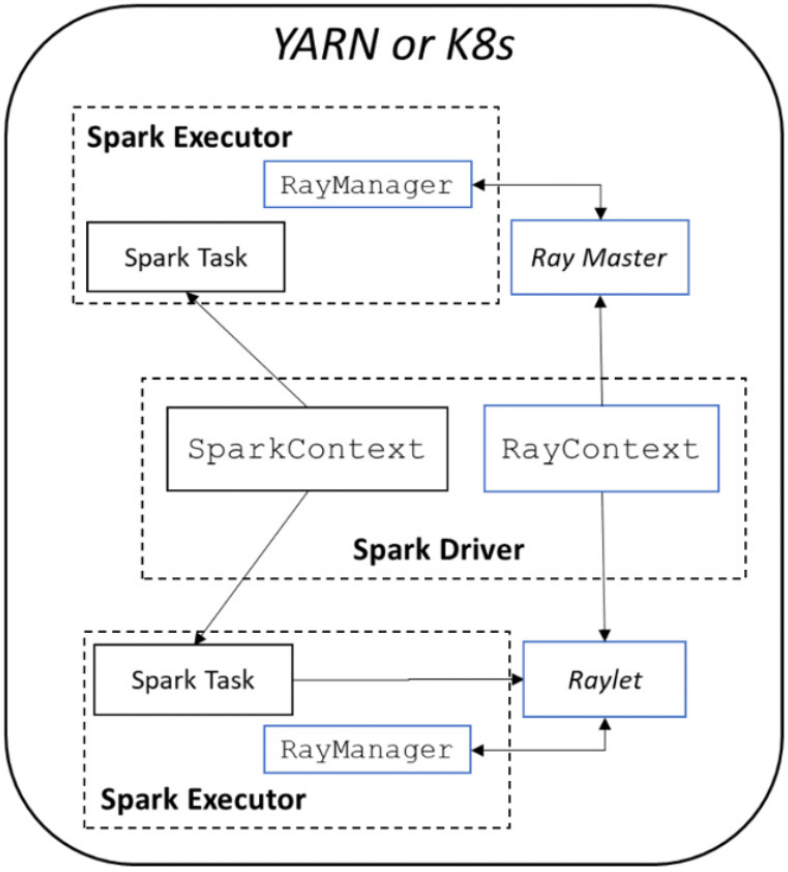


Figure 1. RayOnSpark architecture

Figure 1 illustrates the architecture of RayOnSpark. In the Spark implementation, a Spark program runs on the driver node and creates a SparkSession with a *SparkContext* object responsible for launching multiple Spark executors on a cluster to run Spark jobs. In RayOnSpark, the Spark driver program additionally creates a *RayContext* object, which will automatically launch Ray processes alongside each Spark executor across the same cluster. *RayContext* will also create a *RayManager* inside each Spark executor to manage Ray processes (e.g., automatically shutting down the processes when the program exits). Figure 2 demonstrates how users can directly write Ray code inside standard Spark applications after initializing RayOnSpark:

```
import ray
from bigdl.orca import init_orca_context
from bigdl.orca.ray import RayContext

# Initialize SparkContext on the underlying cluster (e.g. the Hadoop/Yarn cluster)
sc = init_orca_context(cluster_mode="yarn", cores=...,memory=...,num_nodes=...)
# Initialize RayContext and launch Ray under the same cluster.
ray_ctx = RayContext(sc, object_store_memory=...,...)
ray_ctx.init()

@ray.remote
class Counter(object)
   def __init__(self):
      self.n = 0
   def increment(self):
      self.n += 1
      return self.n

# The Ray actors are created across the big data cluster
counters = [Counter.remote() for i in range(5)]
ray.get([c.increment.remote() for c in counters])
ray_ctx.stop()
sc.stop()
```

Figure 2. Sample code of RayOnSpark

AutoML (orca.automl): Tune AI Applications Effortlessly Using Ray Tune

Hyperparameter optimization (HPO) is important for achieving accuracy, performance, etc. of a ML or deep learning (DL) model. However, manual HPO can be a time-consuming process that may not optimize thoroughly enough. On the other hand, HPO in a distributed environment can be difficult to implement. BigDL introduces an AutoML capability (via orca.automl) built on top of Ray Tune to make life easier for data scientists.

What Is orca.automl?

In many cases, data scientists would prefer to prototype, debug, and tune AI applications on their laptops, and if the same code can be moved intact to a cluster, it will greatly improve the end-to-end productivity. BigDL's Orca project helps users to seamlessly scale their code from a laptop to a cluster. Furthermore, BigDL's orca.automl leverages RayOnSpark and Ray Tune, and provides a distributed, hyperparameter tuning API called AutoEstimator. As Ray Tune is framework agnostic, AutoEstimator is suitable for both PyTorch and TensorFlow models. Users can tune models in a consistent manner on their laptops, local servers, Kubernetes clusters, Hadoop/YARN clusters, etc.

With these features, orca.automl in BigDL can be used to automatically explore the search space (including models, hyperparameters, etc.) for many AI applications. As an example, we have implemented AutoXGBoost (XGBoost with HPO) using BigDL's orca.automl to automatically fit and optimize XGBoost models. Compared to a similar solution on an Nvidia A100, training with AutoXGBoost is ~1.7x faster, and the final model is more accurate. See Scalable AutoXGBoost Using Analytics Zoo AutoML for more details. You may also refer to the orca.automl User Guide for design information and the AutoXGBoost Quick Start or Auto Tuning for arbitrary models for hands-on practice.

Chronos: Build Automated Time Series Analysis Using AutoTS on Ray

We have also developed a framework for Automatics Time Series Analysis, known as Project Chronos. orca.automl is leveraged to tune hyperparameters during the automatic analysis.

Why Do We Need Chronos?

Time series (TS) analysis is now widely used in many real-world applications (such as network quality analysis in telecommunications, log analysis for data center operations, predictive maintenance for high-value equipment, etc.) and getting more and more important. Accurate forecasting and detection have become the most sought-after tasks and prove to be huge challenges for traditional approaches. DL methods often perceive time series forecasting and detection as a sequence modeling problem and have recently been applied to these problems with much success.

On the other hand, building the ML applications for time series forecasting/detection can be a laborious and knowledge-intensive process. Hyperparameter setting, preprocessing, and feature engineering may all become bottlenecks for a dedicated DL model. To provide an efficient, powerful, and easy-to-use time series analysis toolkit, we launched Project Chronos, a framework for building large-scale time series analysis applications. This can be used to apply AutoML and distributed training because it is built on top of Ray Tune, Ray Train, and RayOnSpark.

Chronos Architecture

Chronos features several (10+) built-in DL and ML models for time series forecasting, detection, and simulation as well as many (70+) data processing and feature engineering utilities. Users can call standalone algorithms and models (forecasters, detectors, simulators) themselves to acquire the highest flexibility or use our highly integrated and scalable and automated workflow for time series (AutoTS). The inferencing process has also been optimized in a number of ways, including integrating ONNX runtimec.

Figure 3 illustrates Chronos's architecture on the top of BigDL and Ray. This section focuses on the AutoTS component. The AutoTS framework uses Ray Tune as a hyperparameter search engine (running on top of RayOnSpark). For automatic data processing, the search engine selects the best lookback value for a prediction task. For automatic feature engineering, the search engine selects the best subset from a set of features that are automatically generated by various feature generation tools (e.g., tsfresh). For automatic modeling, the search engine searches for hyperparameters such as hidden dim, learning rate, etc.

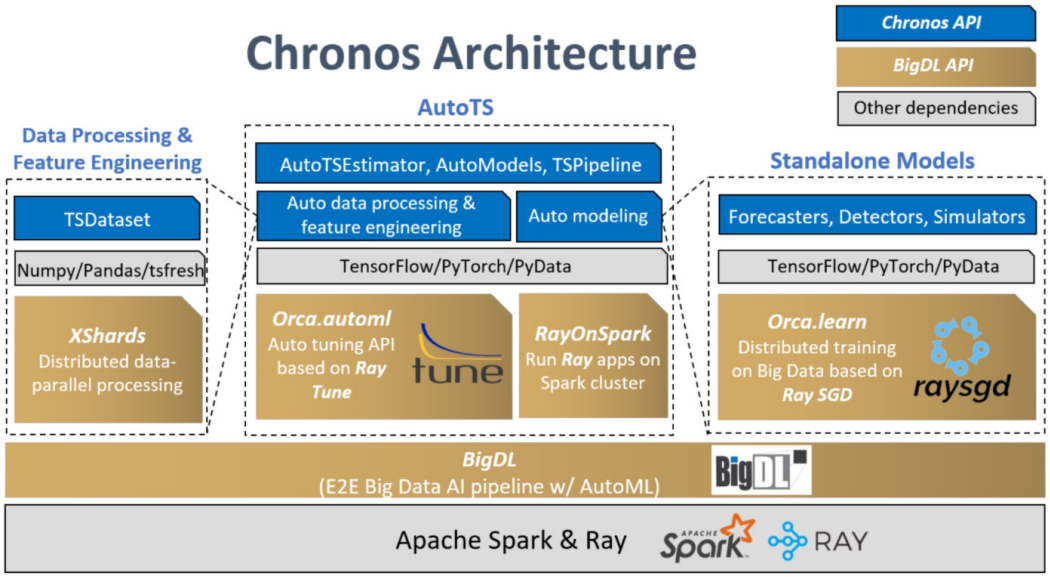


Figure 3. Project Chronos architecture

Chronos Hands-On Example for the AutoTS Workflow

The following code illustrates the training and inferencing process of a time series forecasting pipeline using Chronos's friendly and highly integrated AutoTS workflow:

```
import pandas as pd
from sklearn.preprocessing import StandardScaler
from bigdl.chronos.data import TSDataset

# data initialization and split
df = pd.read_csv("table.csv")
tsdata_train, tsdata_val, tsdata_test = TSDataset.from_pandas(df,
                      dt_col="StartTime",
                      target_col="AvgRate",
                      with_split=True,
                      val_ratio=0.1)

# data processing and feature engineering
standard_scaler = StandardScaler()
for tsdata in [tsdata_train, tsdata_val, tsdata_test]:
   tsdata.gen_dt_feature()\
         .impute(mode="last")\
         .scale(standard_scaler, fit=(tsdata is tsdata_train))
```

This particular workflow utilizes the simple and straightforward API on the `TSDataset` to do some typical time series processing (e.g., imputing, scaling, etc.) and feature generation.

Then, users can initialize `AutoTSEstimator` by stating the model (built-in model name/model create function for 3rd party model), lookback, and horizon. The `AutoTSEstimator` runs the search procedure on top of Ray Tune; each run generates several trials (each with a different combination of hyperparameters and subset of features) at a time and distributes the trials in the Ray cluster. After all trials complete, the best set of hyperparameters, optimized model, and data processing procedure are retrieved according to the target metrics, which are used to compose the resulting `TSPipeline`.

```
from bigdl.chronos.autots import AutoTSEstimator
import bigdl.orca.automl.hp as hp

# create a AutoTSEstimator
auto_estimator = AutoTSEstimator(model='tcn',
                                 past_seq_len=hp.randint(50,100),
                                 future_seq_len=1)

# fit on the AutoTSEstimator with HPO, auto feature, past_seq_len selector
ts_pipeline = auto_estimator.fit(data=tsdata_train,
                                 validation_data=tsdata_val)
```

The `TSPipeline` can be used for prediction, evaluation, and incremental fitting.

```
# predict/evaluate with TSPipeline
y_pred = ts_pipeline.predict(tsdata_test)
test_mse = ts_pipeline.evaluate(tsdata_test, metrics = ['mse'])
```

For detailed information, Chronos user guide is a great place to start.

5G Network Time Series Analysis Using Chronos AutoTS

Chronos has been adopted widely in many areas, such as telecommunication and AI operation. Capgemini Engineering leverages the Chronos AutoML workflow and inferencing optimization in their 5G Medium Access Controller (MAC) to realize cognitive capabilities as part of Intelligent to RAN Controller nodes. In their tasks, Chronos is used to forecast UE's mobility to assist the MAC scheduler in efficient link adaptation on two-key KPIs. With Chronos AutoTS, Capgemini engineers changed their model to our built-in TCN model and enlarged the lookback value, which successfully increased the AI accuracy by 55%. Please refer to the white paper for more details.

Conclusion

In this article, we showed how BigDL leverages Ray and its libraries to build scalable AI applications for big data (using RayOnSpark), improve end-to-end AI development productivity (using AutoML on top of RayTune), and build domain-specific AI use-cases such as Automatic Time Series Analysis with project Chronos. BigDL also adopts Ray in other aspects, for example Ray Train is being used in the BigDL Orca project to seamlessly scale-out single-node Python notebooks across large clusters. We are also exploring other use-cases such as recommendation systems, reinforcement learning, etc. which will leverage the AutoML capabilities built on top of Ray.

intel software

THE PARALLEL UNIVERSE

Printed in USA 707/IH Please Recycle.