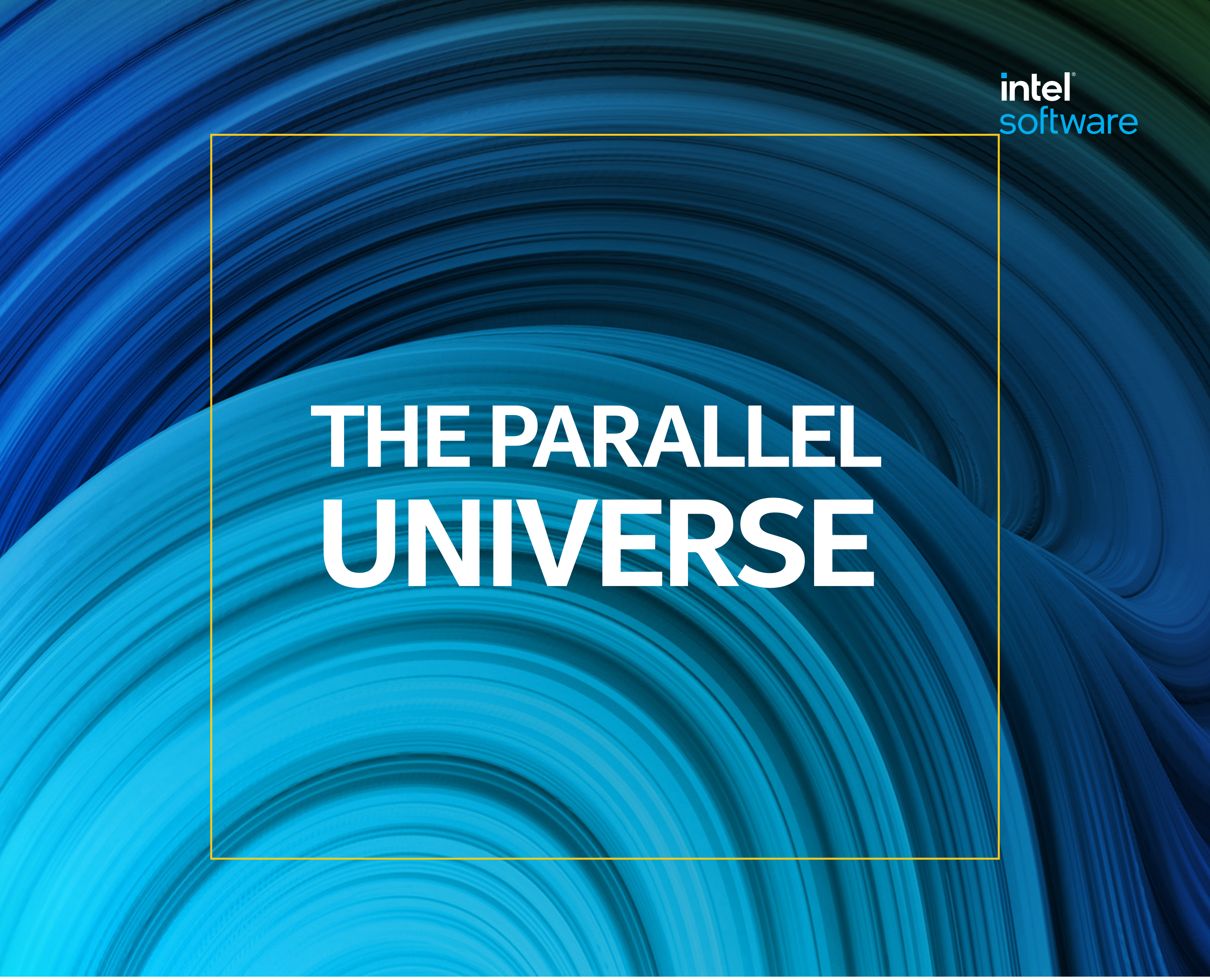
intel software

THE PARALLEL UNIVERSE



Analyzing the Performance of Reduction Operations in Data Parallel C++

Heterogeneous Processing Requires Data Parallelization: SYCL* and DPC++ Are a Good Start

OpenMP* Accelerator Offload

Issue 45 2021

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.

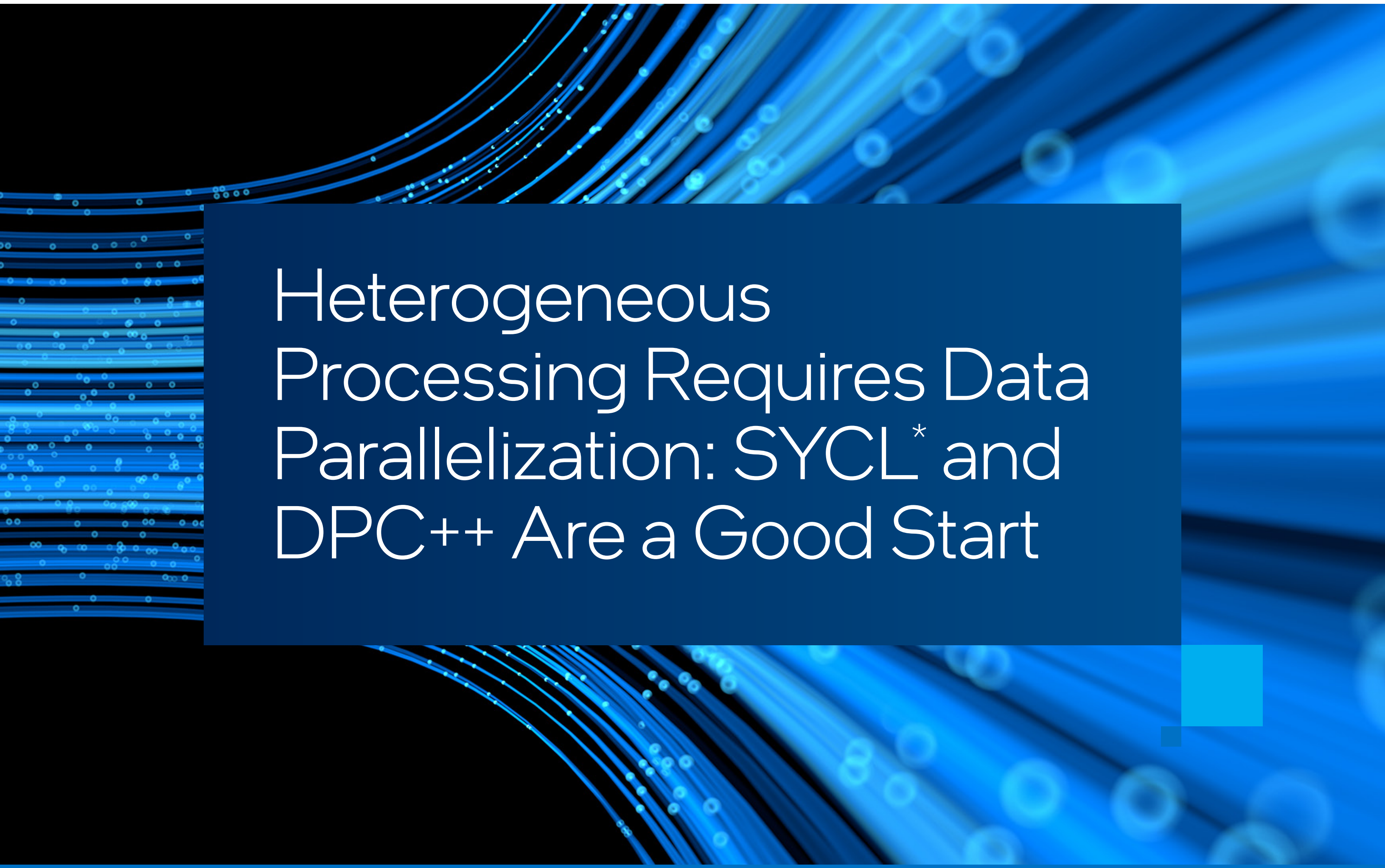
Big Announcements at ISC 2021!

Trish Damkroger's (Intel Vice President and General Manager, High Performance Computing) keynote at the 2021 International Supercomputing Conference (ISC) covered several new products and product enhancements. I can't cover them all here (see New Intel XPU Innovations Target HPC and AI for more details), but I'm particularly enthusiastic about two of them. First, Intel's Xe-HPC-based discrete GPU (codenamed Ponte Vecchio, PVC) has powered-on. I'm enjoying experimenting with oneAPI on integrated GPUs, but I'm anxious to run production-scale workloads on PVC. Second, integrated high-bandwidth memory (HBM) and Intel® Advanced Matrix Extensions (AMX) are coming in the next generation of Xeon Scalable processors (codenamed Sapphire Rapids, SPR). Vectorization with the Intel® Advanced Vector Extensions (AVX) already make Xeon an accelerator in its own right, but HBM and AMX will make Xeon-based acceleration even more effective.

We have a mix of topics in this issue – everything from low-level performance tuning in Data Parallel C++ to simple but effective tricks to boost AI performance. On the oneAPI and heterogeneous parallelism side of the spectrum, our feature article, **Analyzing the Performance of Reduction Operations in Data Parallel C++**, is a continuation of the in-depth analysis from the previous issue of The Parallel Universe (see Reduction Operations in Data Parallel C++). We also have a guest editorial from our editor emeritus, James Reinders: **Heterogeneous Processing Requires Data Parallelization**. OpenMP Accelerator Offload shows you how to convert OpenACC to the more portable OpenMP standard and provides tips to improve OpenMP offload performance.

On the data analytics side of the topic spectrum, we have another guest editorial, this one from Professor Sparsh Mittal (Indian Institute of Technology – Roorkee) on **The Role and Potential of CPUs in Deep Learning**. From there, we have two articles on tuning neural network training performance: **Optimizing Distributed AI Training Using Intel® oneAPI Toolkits** and **MiniNAS Neural Architecture Search Using SigOpt and Ray Tune**. We close with two articles that show how to drastically improve classical machine learning performance using the Intel® oneAPI AI Analytics Toolkit and off-the-shelf packages that contain Intel optimizations: **Performance Optimizations for End-to-End AI Pipelines** and **Optimizing CatBoost Performance by Up to 4x**.

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.



Heterogeneous Processing Requires Data Parallelization: SYCL* and DPC++ Are a Good Start

James Reinders, Editor Emeritus, The Parallel Universe

[Note that this article was originally published at The New Stack.]

I like to say that "It's all about XPUs."

We are in a wonderful time when hardware innovation is leading to an explosion in CPUs, GPUs, FPGAs, DSPs, ASICs, and more—which I simply abbreviate as XPUs. XPUs is shorthand for any type of "processing unit"—any hardware that can help my application compute.

As developers, the onslaught of XPUs means that we are increasingly challenged to code for a larger collection of diverse processing units. We are tasked with factoring in extra time, and money, to rewrite and test code to boost application performance for new architectures. More than ever, to preserve our sanity and the maintainability of our code, it is paramount that the code we write is applicable to as many XPUs as possible. Moving to cross-architecture models for application development has shown that this can save organizations significant time and money, and this becomes an even more pressing concern with the rise in popularity of heterogeneous computing.

Underway today is a rethinking because our world is rapidly becoming a world of XPUs that will eventually transform all of computing.

XPUs: Reinventing Software for Accelerated Compute

CUDA*, a widely used proprietary software programming system, was designed and is effective for NVIDIA* GPUs. OpenCL™ took an open approach and achieved a certain level of multivendor support. OpenCL had its own shortcomings—most notably being C-centric and failing to address C++ needs well.

CUDA* and OpenCL have served their purposes well. Going forward, developers need a truly open and multivendor approach to help deliver on the promises of XPUs.

Why SYCL* and Data Parallel C++ (DPC++) Offer the Best Path Forward

The learnings from both CUDA* and OpenCL set the stage for the emergence of a truly popular and open solution for data parallelism based on C++ for heterogeneous systems. That solution is SYCL*, which is a higher-level programming model to improve programming productivity on multiple hardware accelerators. It has quickly gained broad multivendor support, widespread interest, and the support of multiple serious compiler projects.

SYCL* is important because effective programming in our increasingly heterogeneous world requires that we offer performant access for all XPUs. Only a truly open approach can provide that.

SYCL* is an open standard for single-source C++ data-parallel programming of heterogeneous hardware, or XPUs. SYCL* allows single-source compilation in C++ to target multiple devices on a system, rather than using C++ for the host and domain-specific kernel language(s) for the device(s).

SYCL* brings to C++ both kernel-style programming and a mechanism to locate, query, and use accelerators in a system. Kernel-based programming is an important programming style for harnessing data parallelism, which was also supported in OpenCL and CUDA*. An ability to enumerate and access accelerators in a standard way was previously introduced by OpenCL.

Also take a look at DPC++, which provides an open implementation to the LLVM community, with ambitions to upstream everything into LLVM C++ compilers. DPC++ aims to implement SYCL* with some extensions. DPC++ pioneered many features that are now in SYCL* 2020, and therefore had a head start in implementing much of SYCL* 2020, even before the ink was dry on the standard. Work remains to complete alignment with the entire SYCL* 2020 specification; all the work is easy to observe in the very active open source repository. DPC++ is used by Intel to target Intel® CPUs, GPUs, and FPGAs. DPC++ is also used by Codeplay* to target NVIDIA* GPUs. Another SYCL* compiler, hipSYCL, supports AMD* CPUs and GPUs by connecting with AMD's* HIP/ROCm*. Having multiple open source compilers for SYCL* is fantastic for the community, and it demonstrates that SYCL* has broad, diverse, and open support.

Over the course of 2019 and 2020, I worked with a dedicated small team to create the first book about SYCL* and DPC++. You can download a free copy from the Apress website. Shortly after its publication, the Khronos Group* announced the finalized specification for SYCL* 2020.

The recent ratification of the SYCL* 2020 specification is a significant milestone. It is truly an open specification with a bright future ahead, and it is the product of years of specification development by many dedicated individuals from around the industry. Based on C++17, SYCL* 2020 enables easier acceleration of standard C++ applications and drives a closer alignment with the ISO C++ roadmap. The Khronos* Group highlighted, in their SYCL* 2020 announcement, a number of SYCL* 2020 features, including support for Unified Shared Memory (USM), built-in reductions, extensive use of class template argument deduction (CTAD), and atomic operations that align with standard C++ atomics.

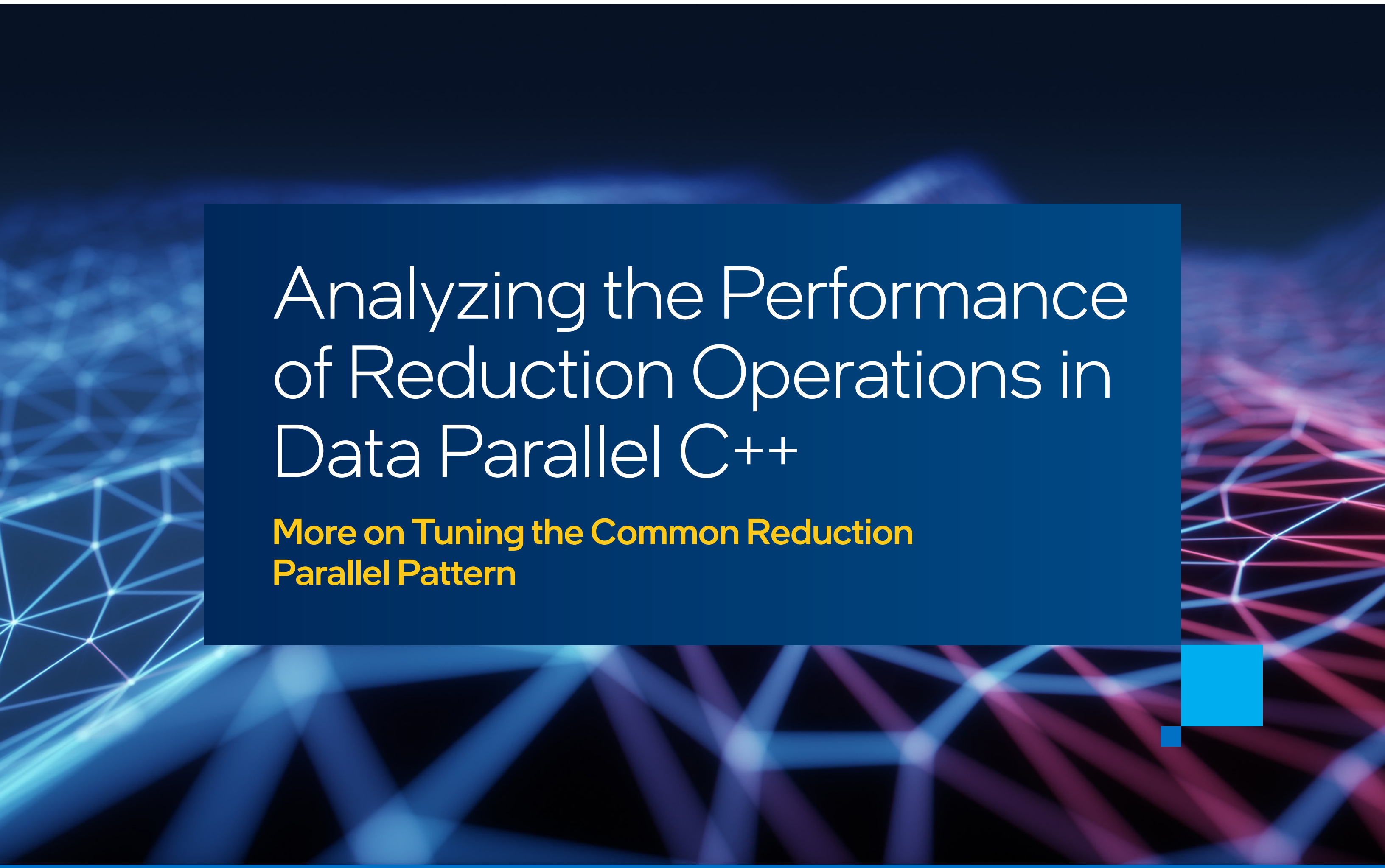
XPUs Are the Future: Let's Keep It Open for the Benefits of XPU Diversity and Programming Sanity

SYCL* and DPC++ will help us make effective use of XPUs. They are part of a broader push for support of XPUs that extends into libraries and all software development tools, building on the ambitions of SYCL* and its compilers. That is the origin of the oneAPI industry initiative, which I'm really passionate about and was excited to be a part of as I rejoined Intel. The support for this whole topic—of easing the challenges of using all XPUs openly—is driving interest in SYCL* and oneAPI. A solid example is the use of the Intel® oneAPI Deep Neural Network Library (oneDNN), initially highly optimized for Intel processors, which accelerates the world's fastest computer (with ARM* processors). As a result, oneDNN has strong ARM* support now, too. The openness of SYCL* and oneAPI libraries and tools are helping usher in a new era for openness and performance to give us useful programming access to all XPUs.

Together, the software developer community has an opportunity to create standards, including SYCL*, that serve the whole industry, and strongly support the adoption of heterogeneous programming (XPUs) and modern C++ as it embraces parallelism.

SYCL* offers an open standard with broad support, lots of ability to participate, multiple open source implementations, and seemingly infinite possibilities. DPC++ provides an open LLVM-based compiler to reduce the effort to support SYCL* and encourage strong compatibility across XPUs. oneAPI offers a forum to discuss and drive open and performant access for XPUs into all aspects of software development.

I hope you'll take the opportunity to get educated about SYCL*, DPC++, and oneAPI because XPUs are the future of compute. We should shape support for XPUs together, in the open, and enjoy the benefits of the enormous diversity in XPUs available for us to program effectively.



Analyzing the Performance of Reduction Operations in Data Parallel C++

More on Tuning the Common Reduction Parallel Pattern

Ramesh Peri, Senior Principal Engineer, Intel Corporation

In the previous article, Reduction Operations in Data Parallel C++, we explored a number of kernels to reduce an array of 10 million elements into a single value using the summation operator. In this article, we will introduce one more reduction technique, called multi-block interleaved reduction. We compare all of these reduction operations using Intel® VTune™ Profiler on both 9th generation and 12th generation Intel® GPUs and explain the reasons for performance differences among these kernels.

Multi-Block Interleaved Reduction

Data Parallel C++ (DPC++) defines short vectors as basic data types with operations like load/store and arithmetic operators defined. These short vector data types can be used to add another level of blocking to get the compiler to generate very long vector operations for architectures that can support them. We use the vec<int, 8> data type, which is a vector of eight integers, to implement the reduction operation shown pictorially in **Figure 1**. The access pattern shown in the illustration has a vector size of two and a sub-group size of four, with each work-item processing four elements of the input vector.

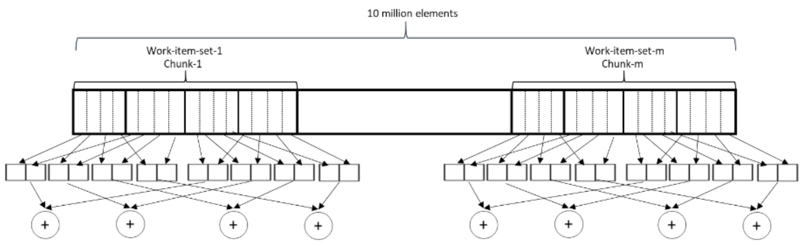


Figure 1. Load a vector of elements, do vector reduction operations on them, and then reduce the final resulting vector.

The following code implements the reduction operation with the memory access pattern described above, using a vector size of eight and a sub-group size of 16, with each work-item processing 256 elements of the input vector:

```
void multiBlockInterleavedReduction(sycl::queue &q,
                    sycl::buffer<int> inbuf,
                    int &res) {
  const size_t data_size = inbuf.get_size()/sizeof(int);
  int work_group_size =
      q.get_device().get_info<sycl::info::device::max_work_group_size>();
  int elements_per_work_item = 256;
  int num_work_items = data_size / elements_per_work_item;
  int num_work_groups = num_work_items / work_group_size;
  sycl::buffer<int> sum_buf(&res, 1);

  q.submit([&](auto &h) {
      const sycl::accessor buf_acc(inbuf, h);
      sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::noinit);
      sycl::accessor<sycl::vec<int, 8>, 1, sycl::access::mode::read_write,
                     sycl::access::target::local>
          scratch(work_group_size, h);
      h.parallel_for(sycl::nd_range<1>{num_work_items, work_group_size},
                [=](sycl::nd_item<1> item)
                [[intel::reqd_sub_group_size(16)]] {
         size_t glob_id = item.get_global_id(0);
         size_t group_id = item.get_group(0);
         size_t loc_id = item.get_local_id(0);
         sycl::ONEAPI::sub_group sg = item.get_sub_group();
         size_t sg_size = sg.get_local_range()[0];
         size_t sg_id = sg.get_group_id()[0];
         sycl::vec<int, 8> sum{0, 0, 0, 0, 0, 0, 0, 0};
         using global_ptr =
            sycl::multi_ptr<int,sycl::access::address_space::global_space>;
         int base = (group_id * work_group_size + sg_id * sg_size)
                            * elements_per_work_item;
         for (size_t i = 0; i < elements_per_work_item / 8; i++)
           sum += sg.load<8>(global_ptr(&buf_acc[base + i * 8 * sg_size]));
         scratch[loc_id] = sum;
         for (int i = work_group_size / 2; i > 0; i >>= 1) {
           item.barrier(sycl::access::fence_space::local_space);
           if (loc_id < i)
              scratch[loc_id] += scratch[loc_id + i];
         }
         if (loc_id == 0) {
            int sum=0;
            for (int i = 0; i < 8; i++)
               sum += scratch[0][i];
            auto v = sycl::ONEAPI::atomic_ref<int,
                        sycl::ONEAPI::memory_order::relaxed,
                        sycl::ONEAPI::memory_scope::device,
                        sycl::access::address_space::global_space>(
                        sum_acc[0]);
            v.fetch_add(sum);
         }
      });
  });
}
```

This kernel can be encoded in a different manner by utilizing the vector load operations instead of explicitly computing the addresses. There is also a small change in dealing with the vector loaded by each work-item to reduce it first locally. (The access pattern for this implementation is shown in **Figure 2**.)

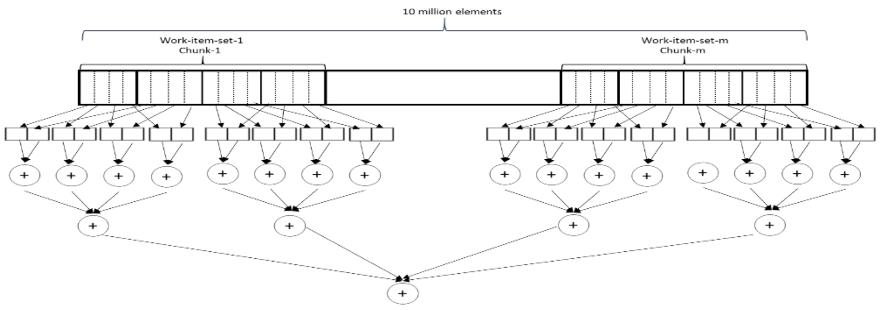


Figure 2. Load a vector of elements, reduce the vector to a single result, and then do the reduction.

```
void multiBlockInterleavedReductionVector(sycl::queue &q,
                    sycl::buffer<int> inbuf,
                    int &res) {
  const size_t data_size = inbuf.get_size()/sizeof(int);
  int work_group_size =
      q.get_device().get_info<sycl::info::device::max_work_group_size>();
  int elements_per_work_item = 256;
  int num_work_items = data_size / 4;
  int num_work_groups = num_work_items / work_group_size;
  sycl::buffer<int> sum_buf(&res, 1);

  q.submit([&](auto &h) {
      const sycl::accessor buf_acc(inbuf, h);
      sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::noinit);
      sycl::accessor<int, 1, sycl::access::mode::read_write,
                     sycl::access::target::local>
          scratch(1, h);
      h.parallel_for(sycl::nd_range<1>{num_work_items, work_group_size},
                [=](sycl::nd_item<1> item)
                [[intel::reqd_sub_group_size(16)]] {
         size_t glob_id = item.get_global_id(0);
         size_t group_id = item.get_group(0);
         size_t loc_id = item.get_local_id(0);
         if (loc_id==0)
            scratch[0]=0;
         sycl::vec<int, 4> val;
         val.load(glob_id,buf_acc);
         int sum=val[0]+val[1]+val[2]+val[3];
         item.barrier(sycl::access::fence_space::local_space);
         auto vl = sycl::ONEAPI::atomic_ref<int,
                         sycl::ONEAPI::memory_order::relaxed,
                         sycl::ONEAPI::memory_scope::work_group,
                         sycl::access::address_space::local_space>(
                         scratch[0]);
         vl.fetch_add(sum);
         item.barrier(sycl::access::fence_space::local_space);
         if (loc_id==0) {
               auto v = sycl::ONEAPI::atomic_ref<int,
                         sycl::ONEAPI::memory_order::relaxed,
                         sycl::ONEAPI::memory_scope::device,
                         sycl::access::address_space::global_space>(
                         sum_acc[0]);
               v.fetch_add(scratch[0]);
        }
      });
  });
}
```

Performance Analysis of the Reduction Kernels

To evaluate the performance of these kernels, we ran them on two different Intel GPUs:

1. Intel® HD Graphics 630 (9th generation integrated graphics). This GPU has 24 execution units (EUs) with seven threads each.
2. Intel® Iris® Xe graphics (12th generation integrated graphics). This GPU has 96 EUs with seven threads each.

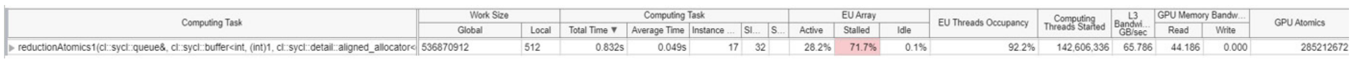
We used VTune Profiler to analyze the performance of the kernels. Also, larger reductions (i.e., 512 million elements instead of 10 million) were performed so that the kernels run long enough to collect good profiling data. The performance of each kernel is shown in **Table 1**. These kernels were each run 16 times, and the average performance was recorded. The Intel® oneAPI Base Toolkit (v2021.2.0) was used to collect the data in this article.

Kernel	Intel HD Graphics 630	Intel Iris Xe Graphics
reductionAtomics1	146	49
reductionAtomics2	258	141
reductionAtomics3	111	38
Tree reduction	288	115
Built-in reduction operator	429	162
multiblockinterleavedreduction	83	37
multiblockinterleavedreductionVector	67	34

Table 1. Performance of different reduction implementations (time in milliseconds).

reductionAtomics1

This kernel is limited by the number of atomic updates that can be performed by the hardware (**Figure 3**).



Computing Task	Work Size		Computing Task					EU Array			EU Threads Occupancy	Computing Threads Started	L3 Bandwi... GB/sec	GPU Memory Bandw...		GPU Atomics
	Global	Local	Total Time ▼	Average Time	Instance ...	SI...	S...	Active	Stalled	Idle				Read	Write	
▸ reductionAtomics1(cl::sycl::queue&, cl::sycl::buffer<int, (int)1, cl::sycl::detail::aligned_allocator<	536870912	512	0.832s	0.049s	17	32		28.2%	71.7%	0.1%	92.2%	142,606,336	65.786	44.186	0.000	285212672

Figure 3. Statistics for reductionAtomics1 on Intel Iris Xe graphics.

Here, the Global Work Size column gives the total work items in this kernel, which is the size of the problem (i.e., 512 million elements). The Instance column is the number of times the kernel is called, 17 in this case. The SIMD Width column is 32, the vector size that the compiler chose for this kernel. The Computing Threads Started column shows the actual number of independent threads that this kernel executed. It is equal to the Global Work Size divided by the SIMD Width and then multiplied by the Instance count. Lastly, the GPU Atomics column gives the total number of atomic operations executed by this kernel. For the reductionAtomics1 kernel, it is twice the number of threads because each thread issues two atomic operations (a SIMD32 in Gen12 is encoded as two SIMD16 instructions).

VTune Profiler's annotated architecture diagram shows that the 9th generation Intel GPU has significant headroom in terms of memory bandwidth (reductionAtomics1 only achieves 15.3GB/s of the 32GB/s peak) (**Figure 4**). The same kernel on the 12th generation GPU achieves much higher memory bandwidth because Intel Iris Xe graphics can handle more atomic memory updates than the 9th generation Intel HD Graphics (**Figure 5**).

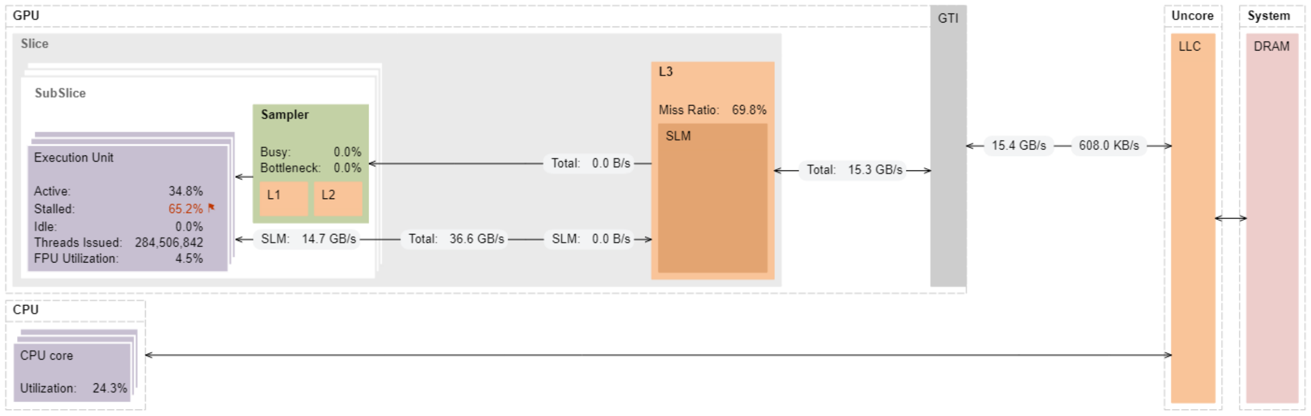


Figure 4. Architecture diagram of reductionAtomics1 on Intel HD Graphics 630.

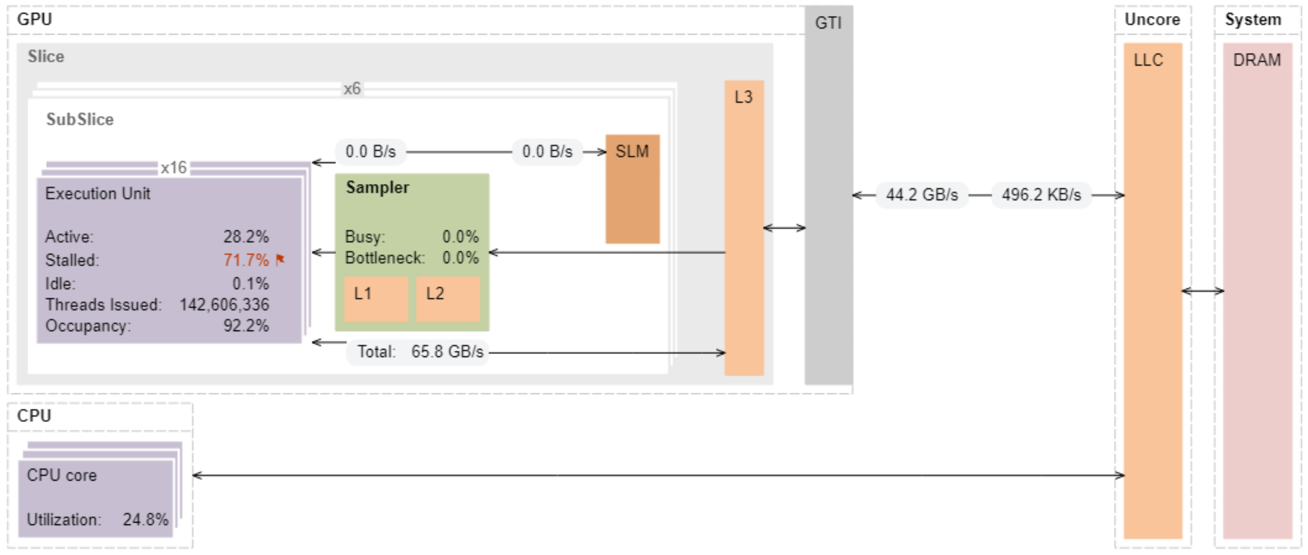


Figure 5. Architecture diagram of reductionAtomics1 on Intel Iris Xe graphics.

reductionAtomics2

This kernel performs quite poorly on both the 9th generation and 12th generation GPUs. The memory access pattern in this kernel results in the compiler generating a vector load instruction that only accesses one element of 16 different cache lines at a time. This results in the first access incurring 16 cache misses with long latency, while all other 15 references will hit in the cache. This is a good cache hit rate and bandwidth from L3, but overall performance is limited by the latency of the first memory reference, which incurs 16 cache misses. It can be seen that the cache miss rate is very low and the L3 memory bandwidth is high when compared to reductionAtomics1, but its overall performance is quite poor on both the platforms (**Figure 6**).

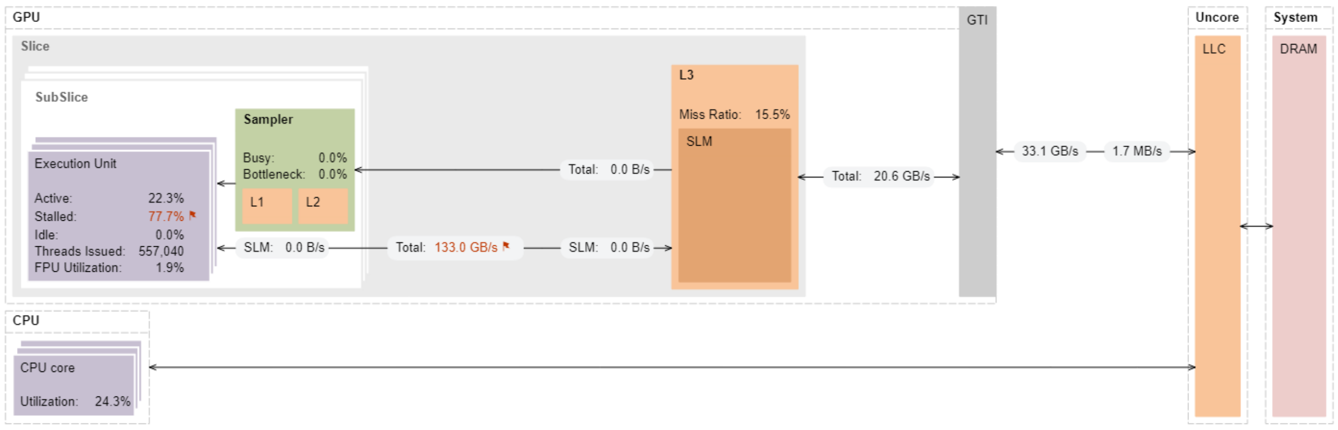


Figure 6. Architecture diagram of reductionAtomics2 on Intel HD Graphics 630.

reductionAtomics3

The memory access pattern in this kernel is such that the vector load instruction loads all the elements of one cache line at a time. This results in 100% cache misses. Even though the cache miss rate is 100%, the latency is better tolerated because multiple threads can be in flight at the same time. This can be seen from the fact that this kernel performs significantly better than reductionAtomics1 and reductionAtomics2 (**Table 1**), even though they have significantly lower cache miss rates (15.5% on reductionAtomics2 and 69.8% on reductionAtomics1) and lower L3 memory bandwidth (**Figure 7**).

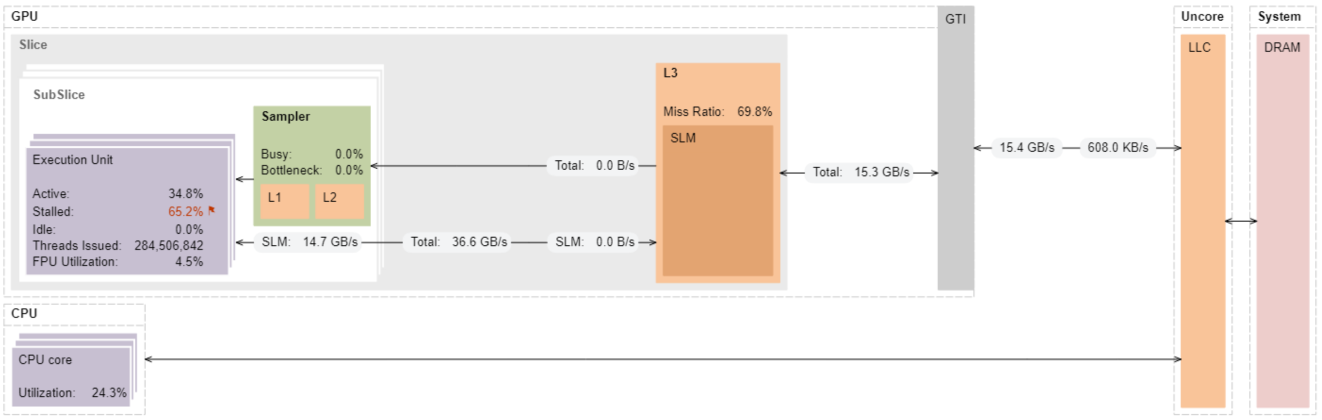


Figure 7. Architecture diagram of reductionAtomic3 on Intel HD Graphics 630.

The annotated architecture diagram for reductionAtomics3 on the 12th generation GPU shows that the memory bandwidth from main memory is almost the same as the L3 bandwidth (56.5GB/s L3 BW to 58.5GB/s memory bandwidth) (**Figure 8**).

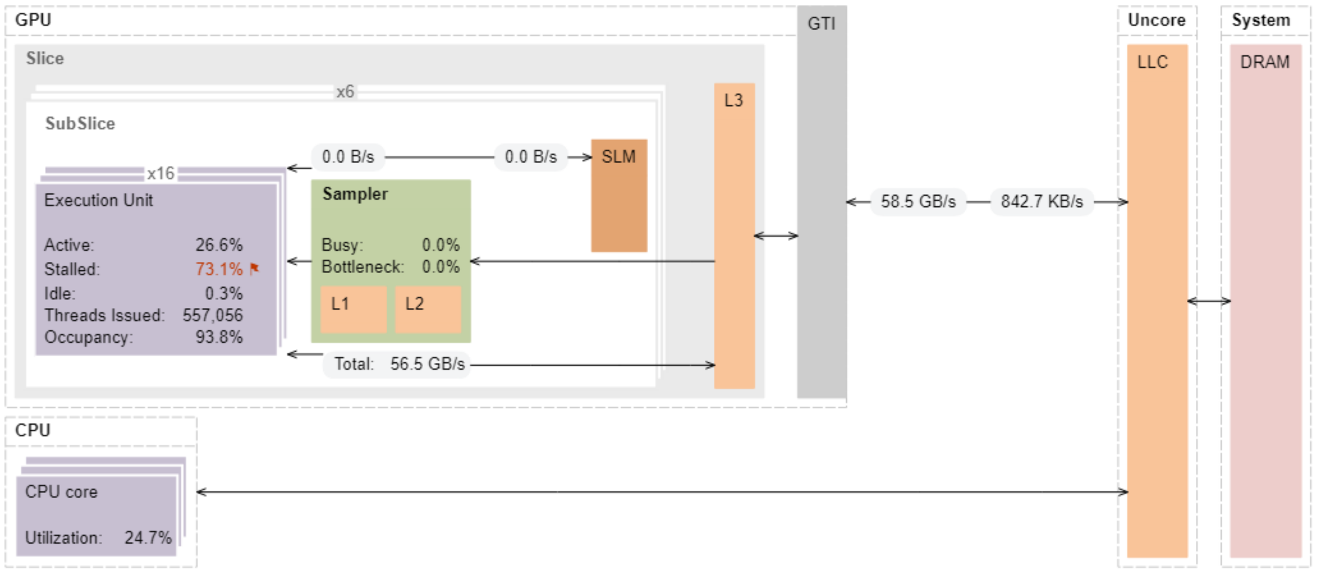


Figure 8. Architecture diagram of reductionAtomic3 on Intel Iris Xe graphics.

Tree Reduction

Tree reduction is a popular technique, but it does not perform very well on either the 9th generation or 12th generation GPU. The inherent imbalance in the algorithm—where half of the EUs are idle in each level of the reduction tree—hurts efficiency (**Figure 9**).

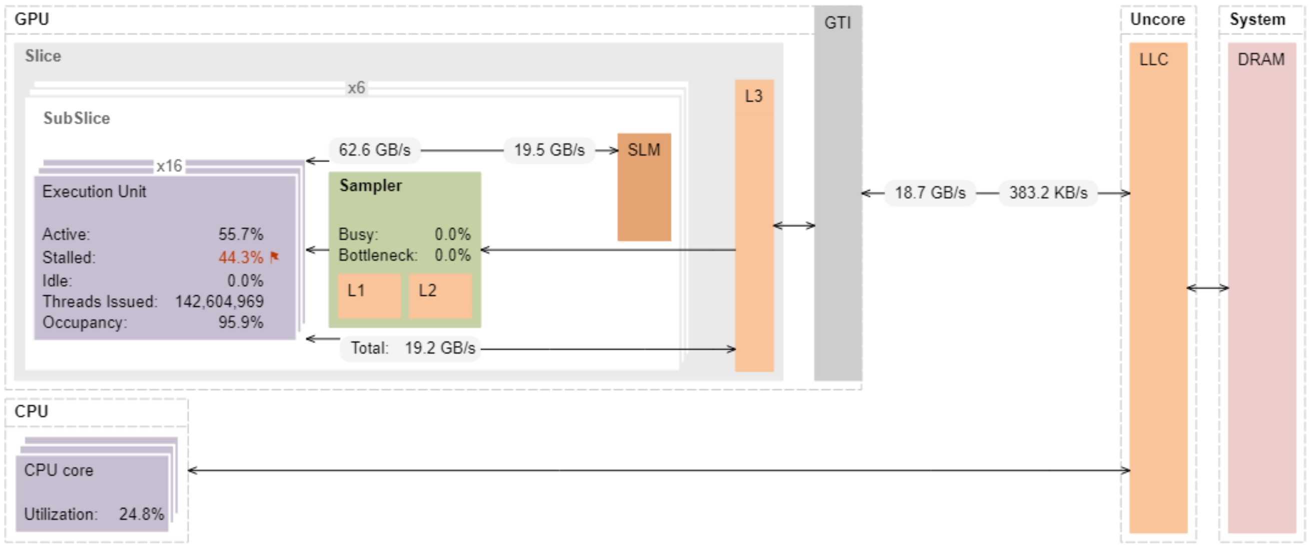


Figure 9. Architecture diagram of Tree Reduction on Intel Iris Xe graphics.

Compiler Built-In Reduction

The compiler built-in reduction operator is still under development and needs additional tuning to reach the performance of other techniques presented here. Looking at the metrics reported by VTune Profiler about the number of atomics and the number of computing threads started, and comparing them to the Tree Reduction, we can conclude that a form of tree reduction on shared local memory (SLM) is used to implement the built-in operator. It also seems that this implementation first copies data from main memory into the shared local memory before applying the reduction operator. This can be seen from the activity in the GPU Shared Local Memory lane in the VTune Profiler platform view (**Figure 10**).

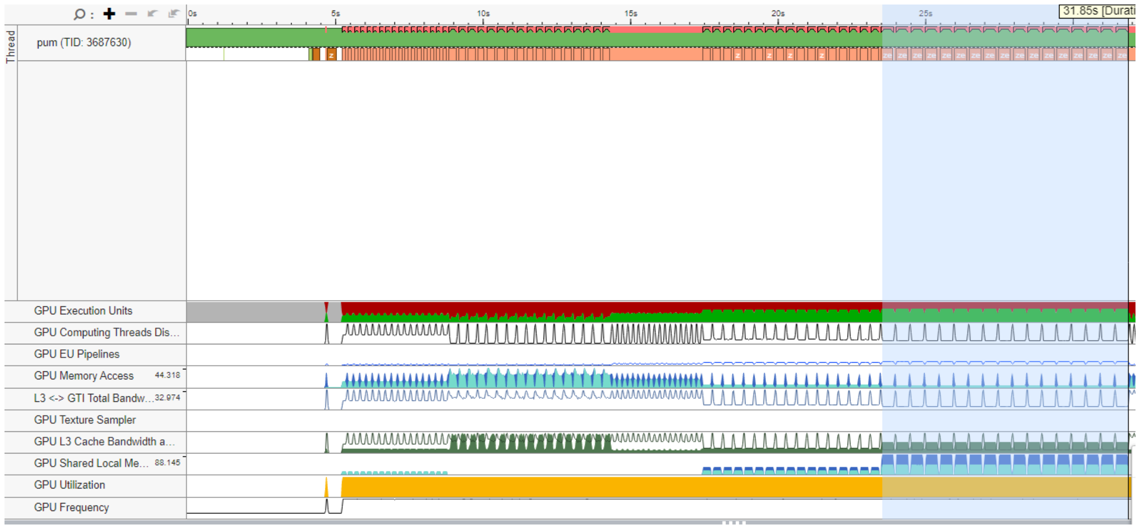


Figure 10. Platform view of the built-in reduction operator on Intel HD Graphics 630.

Comparing the architecture diagrams of Tree Reduction and the compiler built-in reduction operator, it can be seen that the usage of SLM for the latter is much higher (116GB/s vs. 62.6GB/s for read and 59.7GB/s vs. 19.5GB/s) (**Figures 9 and 11**). This is due to the copying of data by each thread before it is used in the reduction operation. In the Tree Reduction implementation, there is no copying of data to SLM; SLM is only used for the intermediate values that need to be produced by each work-group. Hence, our implementation of Tree Reduction performs better than the built-in operator even though they are using the same algorithm.

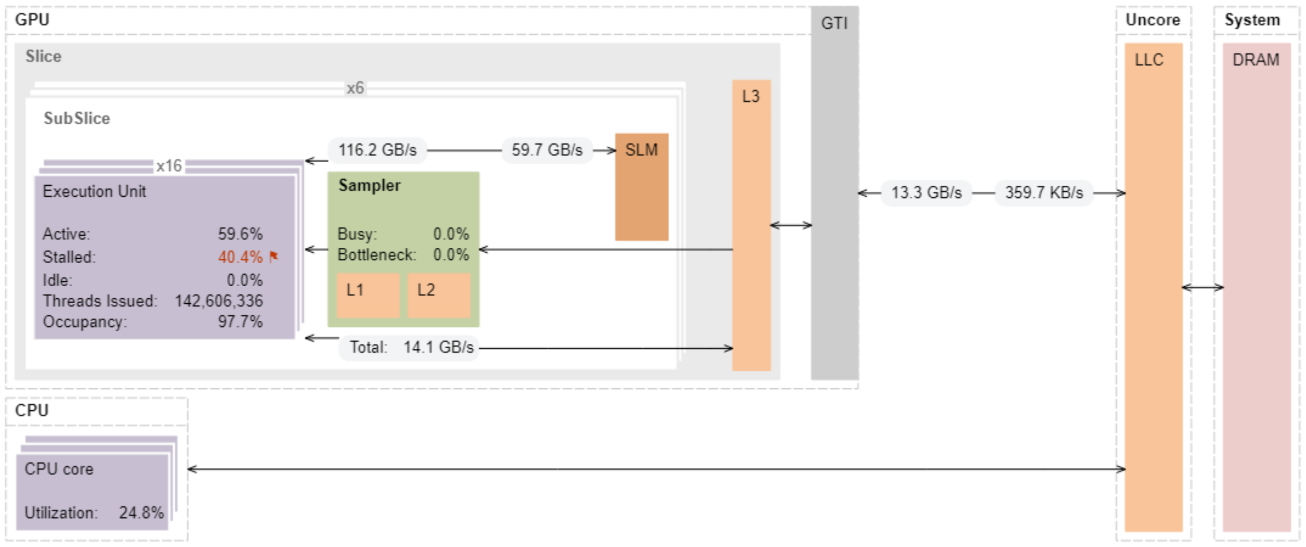


Figure 11. Architecture diagram of the compiler built-in reduction operator on Intel Iris Xe graphics.

It must be remembered that the performance of these reduction algorithms can vary quite a bit among architectures. The performance of the built-in reduction operator will be improved in future oneAPI compilers.

MultiBlockInterleaved

The memory access pattern in this kernel is carefully crafted so that the compiler can generate a block load operation to load 128 elements per thread, which can achieve much higher bandwidth than the other kernels (**Figure 12**).

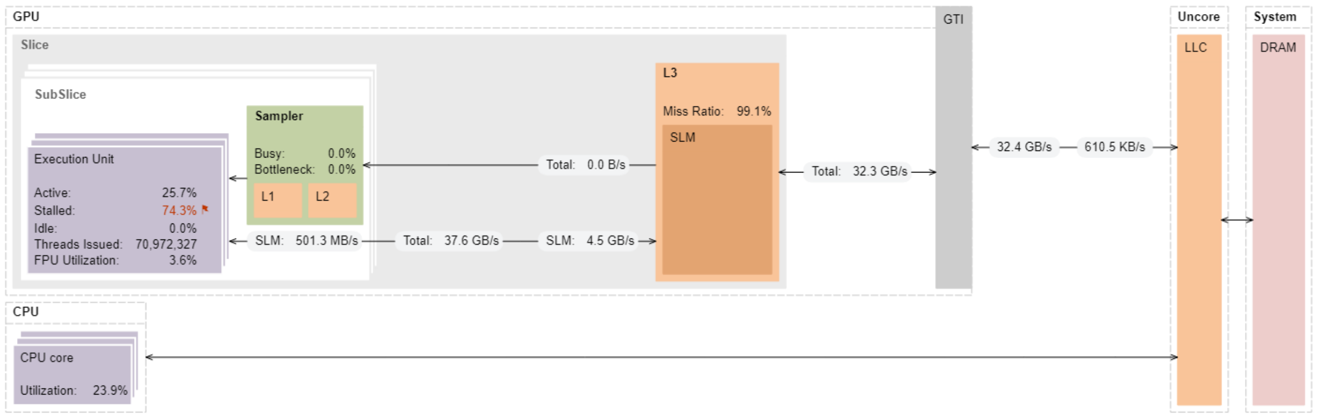


Figure 12. Architecture diagram of multiblockinterleavedvector on Intel Iris Xe graphics.

The subgroup load operations in this kernel are converted by the compiler into the following four SIMD16 load instructions, where each of them populates four registers (Figure 13).

```
(W) send.dc0 (16|M0) r44 r60 null 0x0 0x02484400 {@5, $0} [, msg-length:1, resp-length:4, header:yes, func-control:4400]
(W) send.dc0 (16|M0) r48 r61 null 0x0 0x02484400 {@4, $1} [, msg-length:1, resp-length:4, header:yes, func-control:4400]
(W) send.dc0 (16|M0) r56 r63 null 0x0 0x02484400 {@2, $2} [, msg-length:1, resp-length:4, header:yes, func-control:4400]
(W) send.dc0 (16|M0) r52 r62 null 0x0 0x02484400 {@1, $3} [, msg-length:1, resp-length:4, header:yes, func-control:4400]
```

Figure 13. Assembly code generated by the compiler for the sg.load operation in the kernel.

MultiBlockInterleaved

In this final kernel, we use the DPC++ built-in vectors, which result in SIMD32 instructions, as well as block reads that give even better performance than the previous MultiBlockInterleaved kernel. The MultiBlockInterleavedVector kernel achieves peak memory bandwidth for both platforms: 32.5GB/s on the 9th generation GPU (**Figure 14**) and 62GB/s on the 12th generation GPU (**Figure 15**).

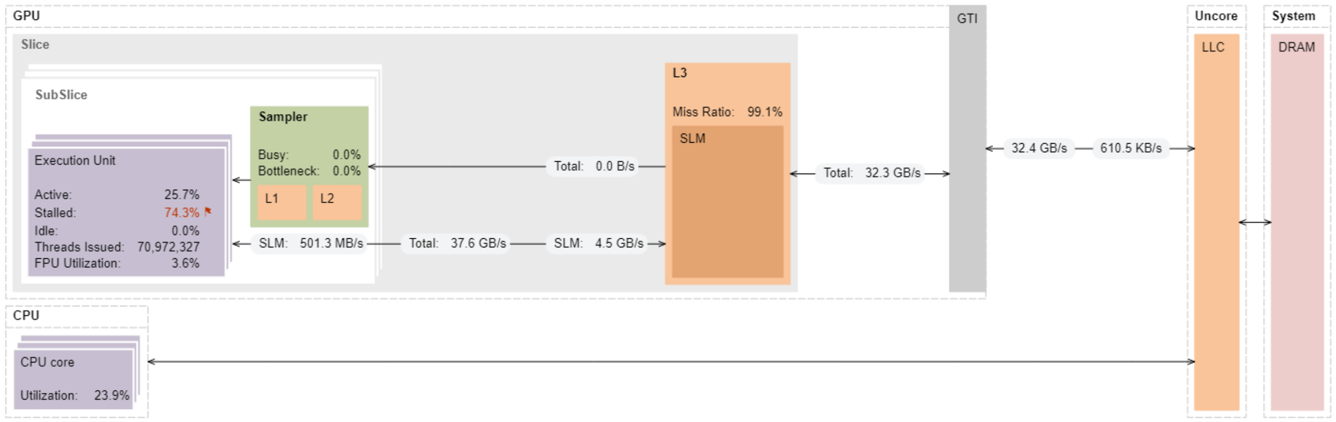


Figure 14. Architecture diagram of the MultiBlockInterleavedVector on Intel HD Graphics 630.

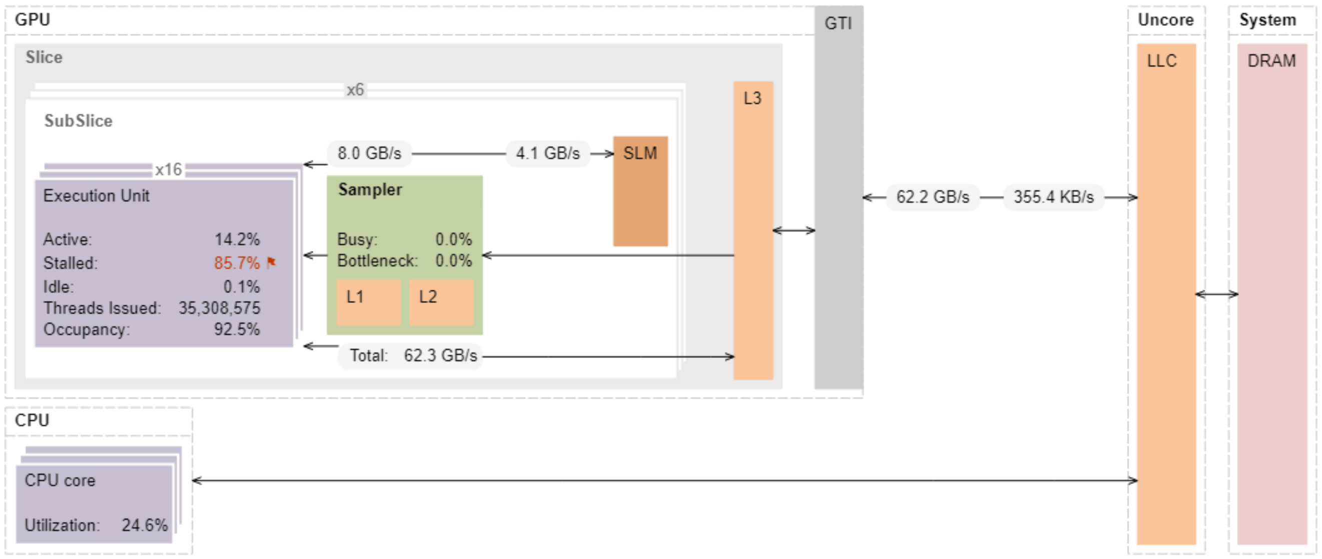


Figure 15. Architecture diagram of the MultiBlockInterleavedVector on Intel Iris Xe graphics.

The assembly code generated by the compiler will load 128 elements per thread. As we see below, a SIMD32 load instruction loads 128 elements into 16 registers (Figure 16). In this case, we achieve peak bandwidth to fetch data from memory into the GPU for this platform.

```
send.dc1 (16|M0) r25 r21 null 0x0 0x04805000 {@2, $1} [, msg-length:2, resp-length:8, header:no, func-control:5000]
send.dc1 (16|M16) r33 r23 null 0x0 0x04805000 {@1, $2} [, msg-length:2, resp-length:8, header:no, func-control:5000]
```

Figure 16. Assembly code generated by the compiler, SIMD32 vector load instruction to load 128 elements on Intel Iris Xe graphics.

Concluding Remarks

Reduction is an important operation in parallel programming and is used in many applications. In this two-part article, we showed various ways in which reduction operations can be coded in DPC++ and evaluated their performance on two integrated Intel GPUs. All the source code for the kernels used in this article are available at https://github.com/rvperi/DPCPP-Reduction.

OpenMP* Accelerator Offload

Portability Across Heterogenous Architectures

Nitya Hariharan, Application Engineer, and Rama Kishan Malladi, Performance Modeling Engineer, Intel Corporation

The OpenMP* standard has supported accelerator offload since version 4.0. These directives enable users to offload data and computation to devices like GPUs. This makes it easier to write portable, heterogeneous parallel code. In this article, we discuss some of the OpenMP* offload directives and show their usage with code samples. We also show some OpenACC* to OpenMP* porting examples.

Porting OpenACC* to OpenMP*

OpenACC* is the directive-based programming method for NVIDIA* GPUs, but lack of support from other vendors limits it to one platform. OpenMP* offload, on the other hand, has broader industry support: the oneAPI framework, the NVIDIA* HPC SDK, the AMD ROCm* stack, and the IBM* XL compiler suite. There is nearly a 1:1 mapping of OpenACC* directives to OpenMP* (**Table 1**), so porting legacy OpenACC* code to OpenMP* is usually easy and straightforward. **Table 1** shows some commonly used OpenACC* pragmas and their OpenMP* equivalents.

OpenACC* Pragma	OpenMP* Pragma	Intent
`#pragma acc parallel`	`#pragma omp target teams`	Create a team of threads on the GPU, with the master thread in each team executing the region
`#pragma acc parallel loop gang worker vector collapse(2)`	`#pragma omp target teams distribute parallel for simd collapse(2)`	Parallel computation, distribute work across GPU hardware threads
`#pragma acc kernels loop reduction(+:norm)`	`#pragma omp parallel for reduction(+:norm)`	Parallel reduction computation
`#pragma acc data copy(A[0:Sz])`	`#pragma omp target data map(tofrom: A[0:Sz])`	Copy data to/from the target device
`#pragma acc update host(A[0:Sz])`	`#pragma omp target update from(A[0:Sz])`	Update data on the host from the device
`#pragma acc data copyin(A[0:Sz])`	`#pragma omp target data map(alloc:A[0:Sz])`	Allocate memory on the device
`#pragma acc update device(X)`	`#pragma omp target update to(X[0:Sz])`	Update data on the device from the host
`#pragma acc loop vector`	`#pragma omp simd`	Vectorization
`#pragma acc atomic update`	`#pragma omp atomic update`	Atomically update a memory location

Table 1. Common OpenACC* pragmas and their OpenMP* equivalents

Figures 1a and **1b** show a code snippet ported from OpenACC* to OpenMP*. This is a kernel from a radio astronomy package `tConvolveACC`. The OpenACC* directive, `#pragma acc parallel loop`, is replaced with the OpenMP* offload directive, `#pragma omp target parallel for`, plus explicit data transfer directives to and from the target device. The OpenACC* implementation possibly did an implicit copy or used unified shared memory allocation to manage the data transfer.

```
degridKernelACC(...)
{
  ...
  #pragma acc parallel loop
  for (dind = 0; dind < d_size; ++dind) {
    ...
  }
}
...
gridKernelACC(...)
{
  ...
  #pragma acc parallel loop
  ...
        #pragma acc atomic update
        gptr_re[0] = gptr_re[0] + cval.real();
  ...
}
```

Figure 1a. The sample kernel from `tConvolveACC` implemented in OpenACC*.

```
degridKernelOmpOffload(...)
{
  ...
  #pragma omp target parallel for                 \
          map(tofrom:d_data[0:d_size])            \
          map(to:d_grid[:grid.size()])            \
          map(to:d_C[:C.size()])                  ...
  for (dind = 0; dind < d_size; ++dind) {
    ...
  }
}
...
gridKernelOmpOffload(...)
{
  ...
  #pragma omp target teams distribute parallel for     \
          map(tofrom:d_grid[:grid.size()])             ...
  ...
        #pragma omp atomic update
        gptr_re[0] = gptr_re[0] + cval.real();
  ...
}
```

Figure 1b. The sample kernel from `tConvolveACC` implemented in OpenMP*.

OpenMP* Offload on Intel® Platforms

We now look at the steps required to build and execute the offload code. We tested our OpenMP* offload code with the 2021.2.0 version of the Intel® oneAPI Base Toolkit, using the following compiler flags:

```
-fiopenmp -fopenmp-targets=spir64="-mllvm \
-vpo-paropt-enable-64bit-opencl-atomics=true \
-fp-model=precise"
```

The `fiopenmp  fopenmp targets=spir64` flags are two new options that tell the compiler to generate a fat binary for the GPU. The `-vpo-paropt-enable-64bit-opencl-atomics=true` compiler option enables atomic and reduction operations. See the online documentation for more details.

The user needs to set the `OMP_TARGET_OFFLOAD` environment variable to run OpenMP* offload code on the GPU. (A runtime error will result if the GPU is unavailable.) The user can also choose between the Level Zero or OpenCL™ backends:

```
export OMP_TARGET_OFFLOAD = MANDATORY
export LIBOMPTARGET_PLUGIN = {LEVEL0|OPENCL}
```

The `LIBOMPTARGET_DEBUG` environment variable can be set to one or higher to obtain GPU offload debugging information. In **Figure 2a**, we highlight the debug information from the `tConvolveACC` OpenMP* offload kernel when run with the Level Zero plugin. The two offload regions are in functions `gridKernelACC` and `degridKernelACC`, which belong to a class named `Benchmark`. **Figure 2b** shows the variable being transferred to the target device by the map clause. **Figure 2c** shows the data being transferred from the host to the target device. Once all the data required for the computation is present on the device, the kernel is executed, as shown at the bottom of **Figure 2a**.

```
Target LEVEL0 RTL -->   0:
__omp_offloading_35_198bc1a__ZN9Benchmark13gridKernelACCERKSt6vector
Target LEVEL0 RTL -->   1:
__omp_offloading_35_198bc1a__ZN9Benchmark15degridKernelACCERKSt6vector

...

Libomptarget --> Launching target execution
__omp_offloading_35_198bc1a__ZN9Benchmark13gridKernelACCERKSt6vector with pointer 0x0000000000b72070
(index=0)
```

Figure 2a. Class and function information for the `tConvolveACC` **OpenMP* offload kernel, highlighted in red.**

```
Libomptarget --> Entry  0: Base=0x00007f468ce16010, Begin=0x00007f468ce16010, Size=15310080, Type=0x21,
Name=d_iw[:this->wPlane.size()]

Libomptarget --> Creating new map entry with HstPtrBegin=0x00007f468ce16010,
TgtPtrBegin=0xffffd556aaa00000, Size=15310080, Name=d_iw[:this->wPlane.size()]
```

Figure 2b. Variable information for the `tConvolveACC` **OpenMP* offload kernel, highlighted in blue.**

```
Libomptarget --> Moving 15310080 bytes (hst:0x00007f468ce16010) -> (tgt:0xffffd556aaa00000)
Target LEVEL0 RTL --> Copied 15310080 bytes (hst:0x00007f468ce16010) -> (tgt:0xffffd556aaa00000)
```

Figure 2c. Data transfer information for the tConvolveACC OpenMP* offload kernel, highlighted in green.

Mapping OpenMP* Threads to the Target Device

At run time, the OpenMP* thread hierarchy is mapped to the target device. The `#pragma omp teams` construct creates a league of teams, and the initial thread in each team executes the region. The `#pragma omp distribute` clause distributes the work across the initial threads in the teams, with each team scheduled on a subslice (on Intel® GPUs). Further parallelization of work within each team is done with the `parallel for` clause, with the threads in a team mapped onto the execution unit (EU) threads. Finally, the `#pragma omp simd` clause uses the EU vector lanes to run vectorized code. Threads within a team synchronize at the end of a work sharing construct. This is illustrated for Intel® processor graphics (9th generation), which has one slice, three subslices, eight EUs/subslice, seven threads/EU, and SIMD vector processing units in each EU (**Figure 3**). Mapping of OpenMP* offload pragmas to these respective units on 9th generation Intel processor graphics is also shown.

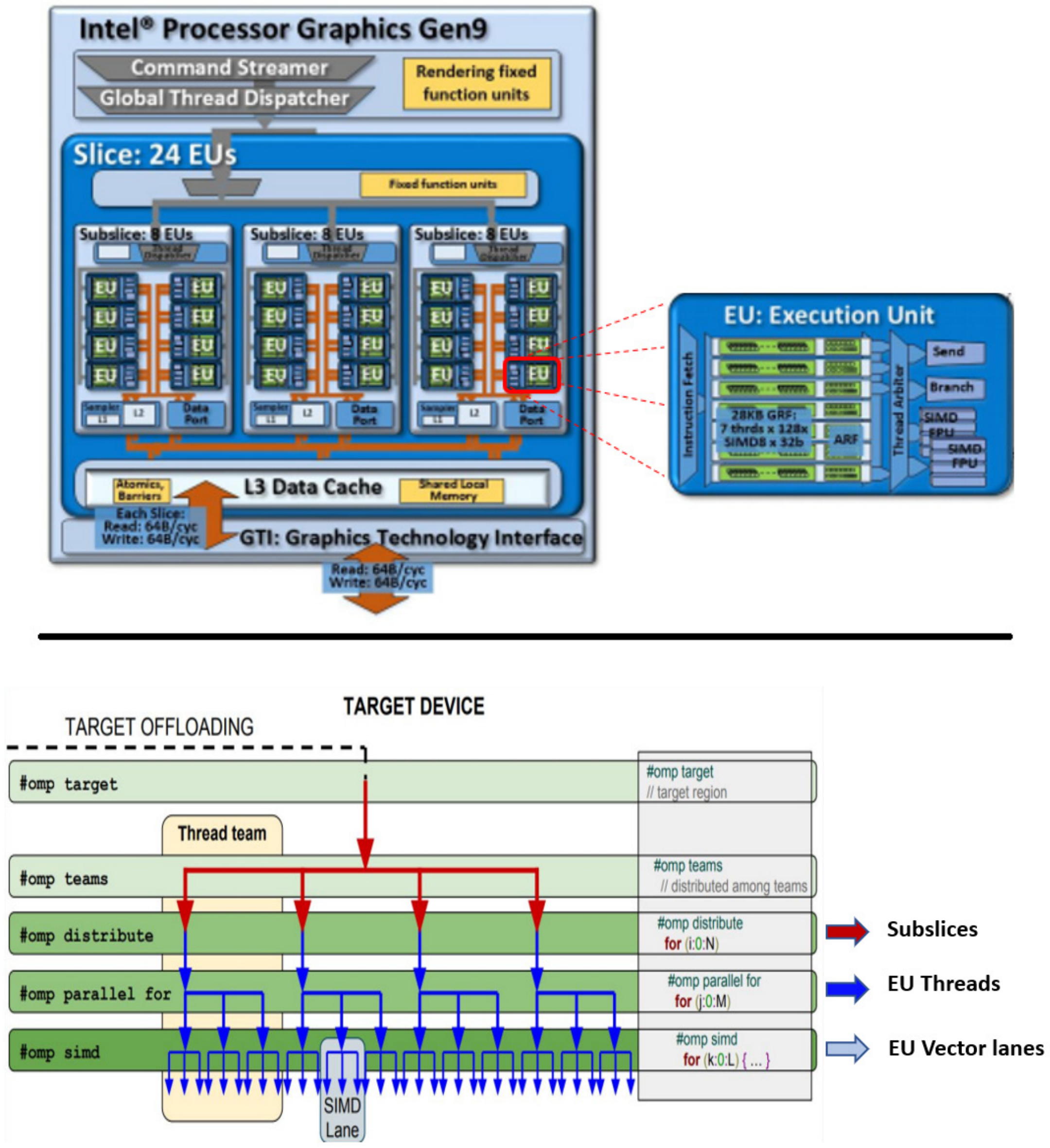


Figure 3. Mapping OpenMP* offload to hardware features on 9th generation Intel processor graphics (adapted from OpenMP* Offloading Verification and Validation: Workflow and Road to 5.0).

OpenMP* Directives for Better Data Transfer to/from the Target Device

Having built an application and successfully offloaded some of the kernels to the target, the next step would be to explore optimization opportunities, such as data transfer. OpenMP* has directives to implement efficient data transfer between host and target. Shown below is an example, `tHogbomCleanACC`, which has two offload targets in the `HogbomClean` function. A naïve OpenMP* offload would result in data transfer during both target invocations. The problem gets worse if this is repeated in a loop for `g_niters`, as shown in the code snippet (**Figure 4a**).

```
HogbomClean(...)
{
...for (unsigned int i = 0; i < g_niters; ++i)
    {
      findPeakOffload(resdata, absPeakVal, absPeakPos, ressize);...
      subtractPSFOffload(psfdata, psfWidth, psfsize, resdata, ...);...
    }...
}

subtractPSFOffload(...)
{
...#pragma omp target teams distribute parallel for map(to: resdata...
}

findPeakOffload(...)
{
...#pragma omp target teams distribute parallel for map(to: resdata...
}
```

Figure 4a. Naïve implementation of two OpenMP* offload kernels resulting in unnecessary data transfers.

Shown in **Figure 4b** is an optimized implementation of `HogbomClean` function that does more efficient data transfer. The `#pragma omp target data map` statement defines the scope for the data to be persistent on the target. Any kernel offload within this scope can reuse the data (with the handle). Subsequent `map` calls to the offloaded kernel will not require data transfers (except for the ones that are explicitly marked for transfer).

```
HogbomClean(...)
{
  ...
  #pragma omp target data map(tofrom: resdata...
  {
    for (unsigned int i = 0; i < g_niters; ++i)
    {
      findPeakOffload(resdata, absPeakVal, absPeakPos, ressize);...
      subtractPSFOffload(psfdata, psfWidth, psfsize, resdata, ...);...
    }...
  }...
}

subtractPSFOffload(...)
{
...#pragma omp target teams distribute parallel for map(to: resdata...
}

findPeakOffload(...)
{
...#pragma omp target teams distribute parallel for map(to: resdata...
}
```

Figure 4b. OpenMP* orphaning example with more efficient copy once and reuse data transfer.

Enhanced Support for Variant Function Dispatch

The OpenMP* offload specification supports function variants that can be conditionally invoked instead of the base function. The implementation of this Intel-specific OpenMP* offload function variant API is supported using `#pragma omp target variant dispatch`. This directive tells the compiler to emit a conditional dispatch code around the function call. If the target device is available, the function variant is invoked instead of the base function. **Figures 5a**, **5b**, and **5c** show an example of the `target variant dispatch` API. Note that the function variant must have the same arguments as the base function, plus an additional last argument of type `void *`.

```
findPeakOffload(..., void *p)
{
  ...
  #pragma omp target teams distribute parallel for
          reduction(max:threadAbsMaxVal) map(to:data[0:size])
  for (size_t i = 0; i < size; ++i) {
    if ( abs(data[i]) > threadAbsMaxVal)
      threadAbsMaxVal = abs(data[i]);
  }

  #pragma omp target teams distribute parallel for
          map(to:data[0:size]) map(from:tmpPos)
  for (size_t i = 0; i < size; ++i) {
    if (abs(data[i]) == threadAbsMaxVal)
      tmpPos = i;
  }
  maxVal = data[tmpPos];
  maxPos = tmpPos;
}
```

Figure 5a. The function variant, findPeakOffload, executes on the target device.

Enhanced Support for Variant Function Dispatch

The OpenMP* offload specification supports function variants that can be conditionally invoked instead of the base function. The implementation of this Intel-specific OpenMP* offload function variant API is supported using `#pragma omp target variant dispatch`. This directive tells the compiler to emit a conditional dispatch code around the function call. If the target device is available, the function variant is invoked instead of the base function. **Figures 5a**, **5b**, and **5c** show an example of the `target variant dispatch` API. Note that the function variant must have the same arguments as the base function, plus an additional last argument of type `void`

```
HogbomClean(...)
{
  ...
  #pragma omp target data map(to: psfdata[0:psfsize])
                          map(tofrom: resdata[0:ressize])
  {
    for (unsigned int i = 0; i < g_niters; ++i)
    {
      #pragma omp target
      findPeakBase(resdata, absPeakVal, absPeakPos, ressize);
      ...
    }
  }
}
```

Figure 5c. Invocation would just need to be of the host version. The offload target function would be executed if target device is present; otherwise, the host version is executed.

Closing Remarks

The platform- and vendor-agnostic device offload support provided by the OpenMP* standard makes it easier for users to target multiple heterogeneous architectures using the same code base. Therefore, we expect increasing adoption of OpenMP* heterogeneous parallelism among users and hardware and software vendors.

Optimizing Distributed AI Training Using Intel® oneAPI Toolkits

Incremental Tuning Can Yield Significant Performance Improvements

Abhay Rawat and Dr. Amarpal S Kapoor, Technical Consulting Engineers, Intel Corporation

Deep learning (DL) workloads have been growing at a rapid pace. DL-based algorithms process massive amounts of data to find patterns for image classification, object detection, time-series prediction, and much more. With the increase in data availability, the complexity of DL models also increased. Models like ResNet and VGG have millions of parameters and perform on the order of billions of floating-point operations. Recent models, like GPT-3 and BERT, have multi-billion to a trillion parameters. Therefore, training DL models is a computationally expensive and time-consuming process. To reduce the time to solution, apart from optimizing single- and multi-core performance, one might also consider scaling out to multiple nodes (i.e., distributed training). This can be achieved by splitting the model (model parallelism), splitting the data (data parallelism), or a combination of both schemes (hybrid parallelism).

This article focuses on tuning and scaling a DL-based algorithm on a cluster of compute nodes. We'll illustrate using a semi-supervised generative adversarial network (S-GAN) to classify images. We demonstrate various tuning options in OpenMP*, TensorFlow*, and the Intel® MPI Library. On a single node, our optimizations achieve a 2x speedup. Scaling across an eight-node cluster achieves an overall speedup of 16x. Image throughput (images/second) for the Intel MPI Library was consistently better than the Open MPI* library by up to 27% on a single node and up to 18% on an eight-node cluster. Many of these performance gains were achieved without code modifications, indicating that these optimizations can be effectively applied to other applications.

Semi-Supervised Generative Adversarial Networks (S-GANs)

Supervised learning requires large amounts of labeled data. Labeling and annotation must be done manually by human experts, so it is laborious and expensive. Semi-supervised learning is a technique where both labeled and unlabeled data are used to train the model. Usually, the number of labeled data points is significantly less than the unlabeled data points. Semi-supervised learning exploits patterns and trends in data for classification.

S-GANs tackle the requirement for vast amounts of training data by generating data points using generative models. The generative adversarial network (GAN) is an architecture that uses large, unlabeled datasets to train an image generator model via an image discriminator model. GANs comprise two models: generative and discriminative. They are trained together in a zero-sum game. The generator's job is to generate data similar to those present in the dataset. The discriminator's job is to identify the actual data among the generated data. S-GANs extend the GAN architecture by adding a supervised discriminator (classifier) to the classification task (**Figure 1**). This results in a classifier that generalizes well across unseen data.

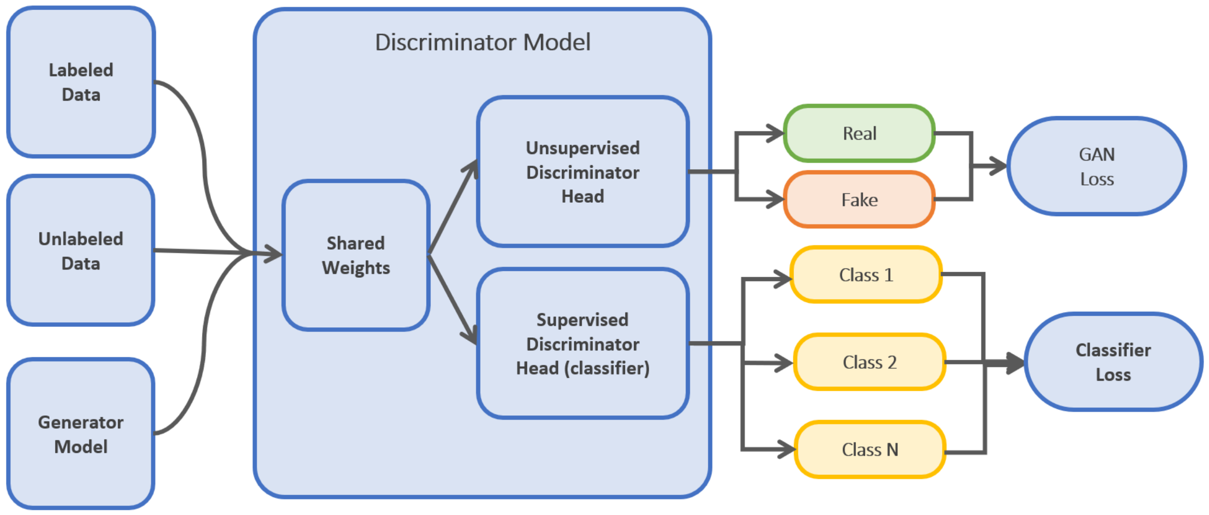


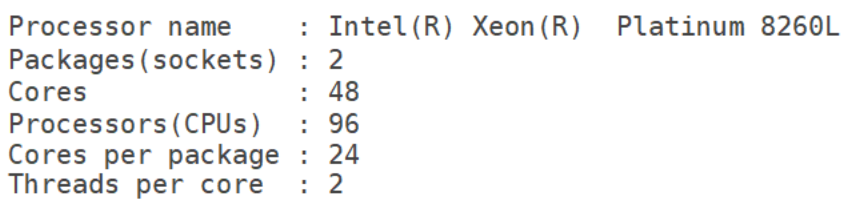
Figure 1. Architecture of S-GAN.

Software and Hardware

Three Intel oneAPI toolkits (v2021.1) were used for these experiments:

1. Intel® oneAPI Base Toolkit
2. Intel® oneAPI HPC Toolkit
3. Intel® oneAPI AI Analytics Toolkit

Intel® Distribution for Python* (IDP, v3.6), which leverages Intel® oneAPI Math Kernel Library and Intel® oneAPI Data Analytics Library, was used to accelerate core Python* numerical packages. The S-GAN model was implemented using Intel® Optimization for TensorFlow* (v1.15). Horovod (v0.20.2) was used for distributed training. Horovod relies on MPI for internode communication, so the performance of two MPI libraries was compared: Intel MPI Library and Open MPI* (v4.0.5). Intel® VTune™ Profiler was used to analyze performance. All tests were run on the Intel Endeavor cluster using Intel® Xeon® Platinum 8260L processors, connected through the Intel® Omni-Path Fabric running at 100 Gbps (**Figure 2**).



```
Processor name    : Intel(R) Xeon(R)  Platinum 8260L
Packages(sockets) : 2
Cores             : 48
Processors(CPUs)  : 96
Cores per package : 24
Threads per core  : 2
```

Figure 2. Node composition ($ cpuinfo -g).

Tuning Methodology

It is good practice to measure baseline performance before optimizing an application. Profiling tools help identify areas for potential optimization, such as threading, vectorization, I/O, multi-node communications, etc. We used the Application Performance Snapshot (APS) in Intel VTune Profiler. The APS profile showed that far too many threads were being spawned by the application. Some threads came from OpenMP*, while others came from the Eigen library that TensorFlow* invokes. The number of OpenMP* and Eigen threads exceeded the number of logical cores per node, resulting in resource oversubscription, which usually hurts performance.

Selecting the Optimal Number of Threads

The first step, therefore, was to find the optimal number of threads. First, we tested different numbers of OpenMP* threads by setting the `OMP_NUM_THREADS` environment variable on a single compute node, using low-resolution images (256 x 256 pixels) and limiting the number of epochs (two) to save time. We found that 50 threads gave the best performance, with 19.67 images/second (**Figure 3**).

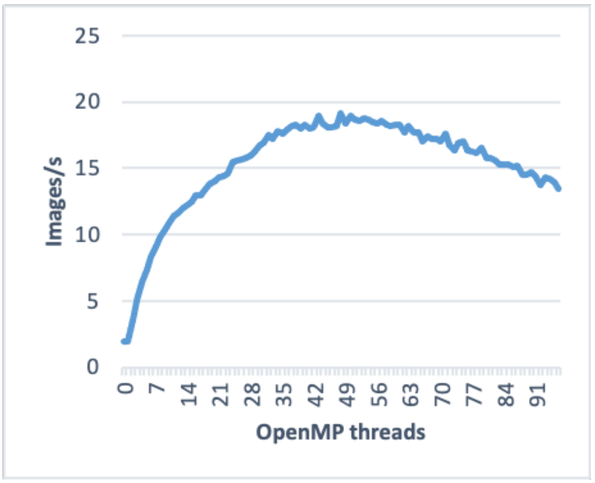


Figure 3. Finding the optimal number of OpenMP* threads.

Next, we used TensorFlow's* threading API to control the number of threads. The value of `inter_op_parallelism_threads` specifies the number of threads used by independent nonblocking operations, while the value of `intra_op_parallelism_threads` specifies the number of threads used for individual operations like matrix multiplication and reductions. **Figure 4** shows the performance at different values of these two parameters. Dark green blocks indicate good performance, while dark red blocks indicate poor performance. White blocks indicate failed runs.

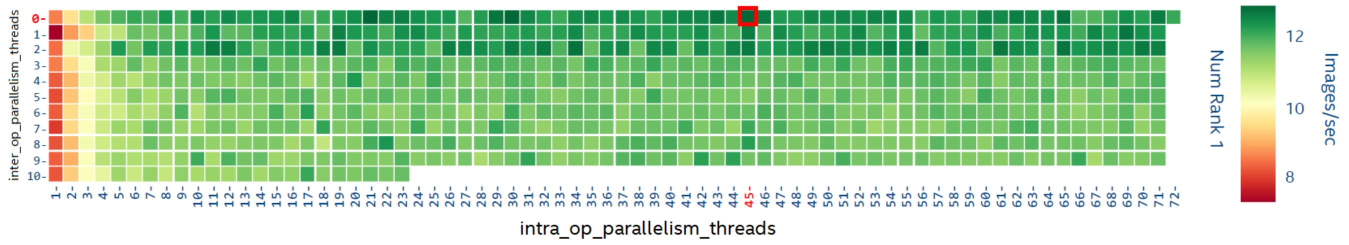


Figure 4. Finding the optimal values of inter/intra_op_parallelism_threads

For a single MPI rank per node, the optimal values of `inter_op_parallelism_threads` and `intra_op_parallelism_threads` were found to be 0 and 45, respectively, which corresponds to 12.34 images/second. This was lower than the value of 19.67 images/second achieved using OMP_NUM_THREADS (**Figure 3**), so we used this environment variable to control the number of threads instead of using TensorFlow's* threading API.

Selecting the Optimal Number of MPI Ranks per Node

The application uses MPI/OpenMP* hybrid parallelism, so it is important to find the best combination of OpenMP* threads and MPI ranks per node. We repeated the test from **Figure**

3 (i.e., 1 ≤ `OMP_NUM_THREADS` ≤ 96), but varied the number of MPI ranks per node (21 ≤ ppn ≤ 24, where ppn is the number of ranks per node) (**Figure 5**). The best performance (26.3 images/second) was achieved with eight MPI ranks and nine OpenMP* threads on each node.

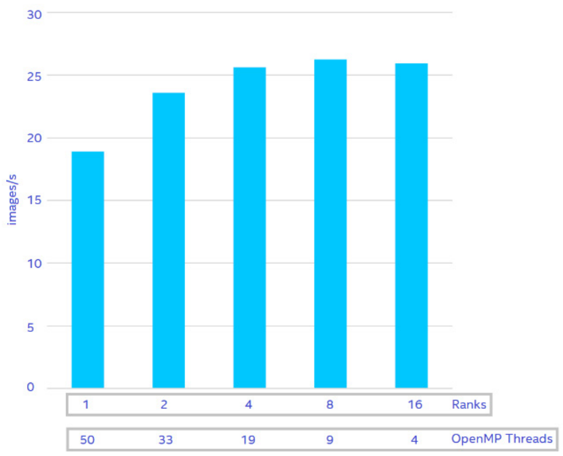
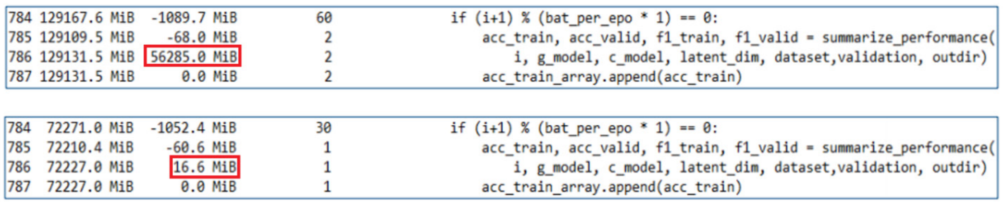


Figure 5. Finding the best combination of OpenMP* threads and MPI ranks in a single node.

Overcoming Memory Leakage

Low-resolution images were used for the experiments described so far. For higher-resolution images (1360 x 1360 pixels), the memory footprint of each worker process increased significantly, so running eight MPI ranks per node causes out-of-memory errors. It turned out that the memory footprint of a single rank was over 129 GB. With only about 200 GB of DRAM available on each node, it was only possible to launch a single rank per node, which would have been suboptimal for a dual-socket machine (because of NUMA issues). Running our application through the Memory Profiler for Python* utility revealed a memory leak. Around 56 GB of memory was not released during the forward pass of the model (**Figure 6**, top). This turned out to be a known bug (TensorFlow* issue #33009 and Keras issue #13118) in the Keras `Model.predict` method in TensorFlow*. Based on recommendations from the TensorFlow* community, `Model.predict` was replaced with `Model.predict_on_batch`, which lowered the overall per-process memory consumption (**Figure 6**, bottom). With the memory leak fixed, we were still limited to only two ranks per node for high-resolution images. We didn't optimize the memory consumption further, although this might be possible.


```
784 129167.6 MiB  -1089.7 MiB      60           if (i+1) % (bat_per_epo * 1) == 0:
785 129109.5 MiB    -68.0 MiB       2               acc_train, acc_valid, f1_train, f1_valid = summarize_performance(
786 129131.5 MiB  56285.0 MiB       2                   i, g_model, c_model, latent_dim, dataset,validation, outdir)
787 129131.5 MiB      0.0 MiB       2               acc_train_array.append(acc_train)
```

```
784  72271.0 MiB  -1052.4 MiB      30           if (i+1) % (bat_per_epo * 1) == 0:
785  72210.4 MiB    -60.6 MiB       1               acc_train, acc_valid, f1_train, f1_valid = summarize_performance(
786  72227.0 MiB     16.6 MiB       1                   i, g_model, c_model, latent_dim, dataset,validation, outdir)
787  72227.0 MiB      0.0 MiB       1               acc_train_array.append(acc_train)
```

Figure 6. Memory leak detected by Memory Profiler

Setting Other OpenMP* and Intel MPI Library Environment Variables

The optimal settings of three other environment variables (`KMP_BLOCKTIME`, `KMP_AFFINITY`, and `I_MPI_PIN_DOMAIN`) were also explored. Based on previous performance recommendations, we ran the six experiments shown in **Table 1**. Note that the environmental variable settings in set #5 were also used in the other five sets. Set #3 gave the best training performance (**Figure 7**).

Set #	Environment Variable Settings

1	KMP_BLOCKTIME=1
	KMP_AFFINITY=granularity=fine,verbose,compact,1,0
	I_MPI_PIN_DOMAIN=auto:compact

2	KMP_BLOCKTIME=0
	KMP_AFFINITY=granularity=fine,verbose,compact,1,0
	I_MPI_PIN_DOMAIN=auto:compact

3	KMP_BLOCKTIME=1
	I_MPI_PIN_DOMAIN=auto:compact

4	KMP_BLOCKTIME=0
	I_MPI_PIN_DOMAIN=auto:compact

5 (common)	I_MPI_DEBUG=100
	HOROVOD_FUSION_THRESHOLD=33554432

6	KMP_BLOCKTIME=200
	I_MPI_PIN_DOMAIN=auto:compact

Table 1. Experimental settings for other OpenMP* and Intel MPI Library environment variables.

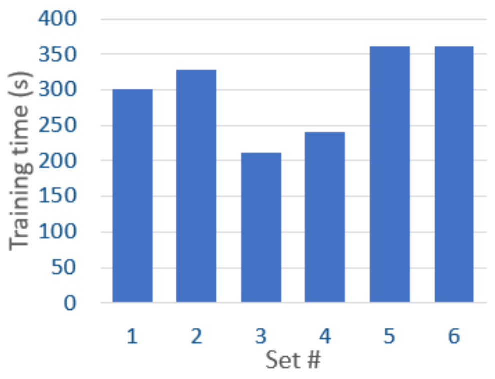


Figure 7. Performance comparison of the six experiments from Table 1

Choosing Between MPI Libraries

Next, we gathered the optimal settings from the previous tuning experiments and compared the performance of Intel MPI Library against Open MPI* for the high-resolution images (1360 x 1360 pixels), with two MPI ranks per node and 33 OpenMP* threads for a total of 100 epochs. The following Intel MPI Library command-line was used:

```
$ OMP_NUM_THREADS=33 KMP_BLOCKTIME=1 I_MPI_PIN_DOMAIN=auto:compact \
   mpirun -n 2/4/8/16 -ppn 2 -f $HOSTFILE \
   python SGAN_pob.py --image_size 1360 --num_epochs 100 --save_checkpoints False
```

Roughly equivalent runtime settings were used for Open MPI*:

```
$ OMP_NUM_THREADS=33 KMP_BLOCKTIME=1 \   mpirun -n 2/4/8/16 -hostfile $HOSTFILE
-bind-to socket -map-by ppr:1:socket \
   --mca btl ^openib --mca pml ob1 --report-bindings -x LD_LIBRARY_PATH \
   python SGAN_pob.py --image_size 1360 --num_epochs 100 --save_checkpoints False
```

Intel MPI Library outperformed Open MPI* both in single- and multi-node scenarios (**Figure 8**).

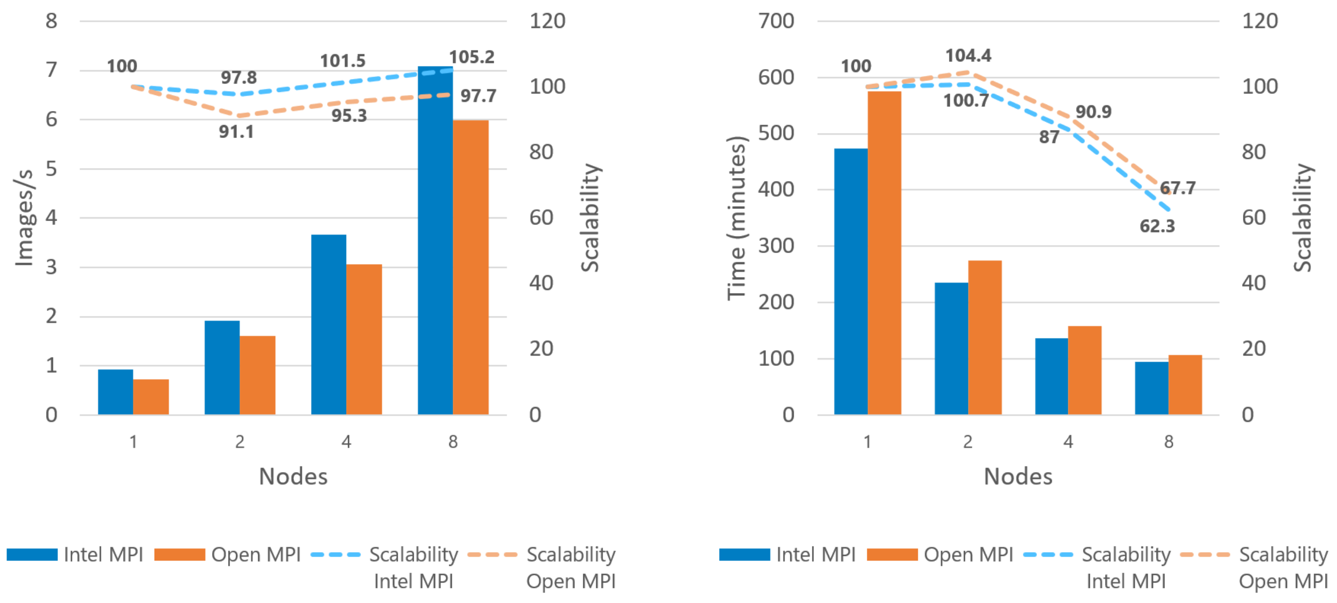


Figure 8. Performance comparison of Intel MPI Library and Open MPI*.

Although parallel scalability based on images/second appears linear, scaling based on time is sublinear, indicating application inefficiency. Further analysis revealed that the frequency of evaluations did not correctly scale according to the total number of nodes. As the number of nodes increased, so did the frequency of evaluations. Also, these evaluations were done by the MPI master rank, so all other ranks stalled until the master rank finished the evaluation. We fixed this issue by performing the evaluation step every 15 training iterations, regardless of the number of nodes (**Figures 9** and **10**). This modification did not affect model accuracy.

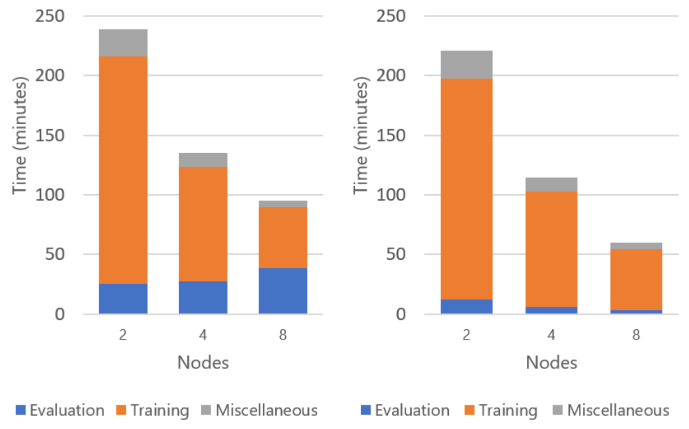


Figure 9. Fraction of total time spent on training and evaluation before optimization (left) and after optimization (right)

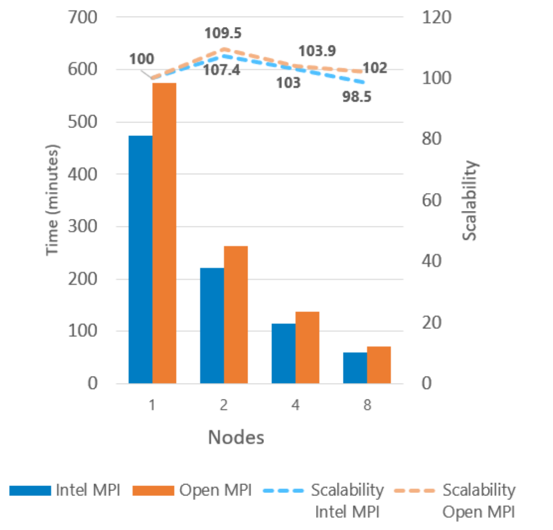


Figure 10. Multi-node performance with linear time scaling

Conclusions

In this article, we presented an incremental optimization approach to achieve better S-GAN training performance. Both runtime and source code-based optimizations were performed to resolve memory issues, scaling issues, and single-node performance inefficiencies. Horovod was used to implement distributed training of the S-GAN model on a multi-node cluster. Two MPI libraries (Intel MPI Library and Open MPI*) were compared.

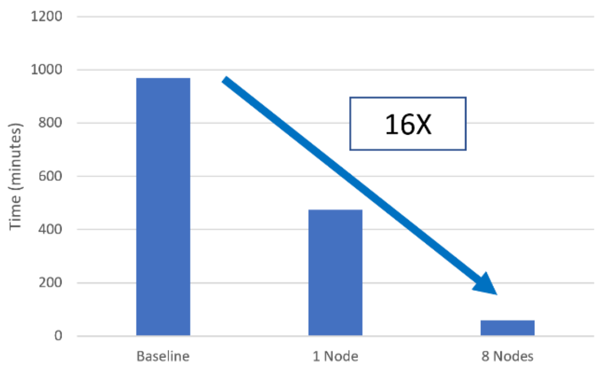
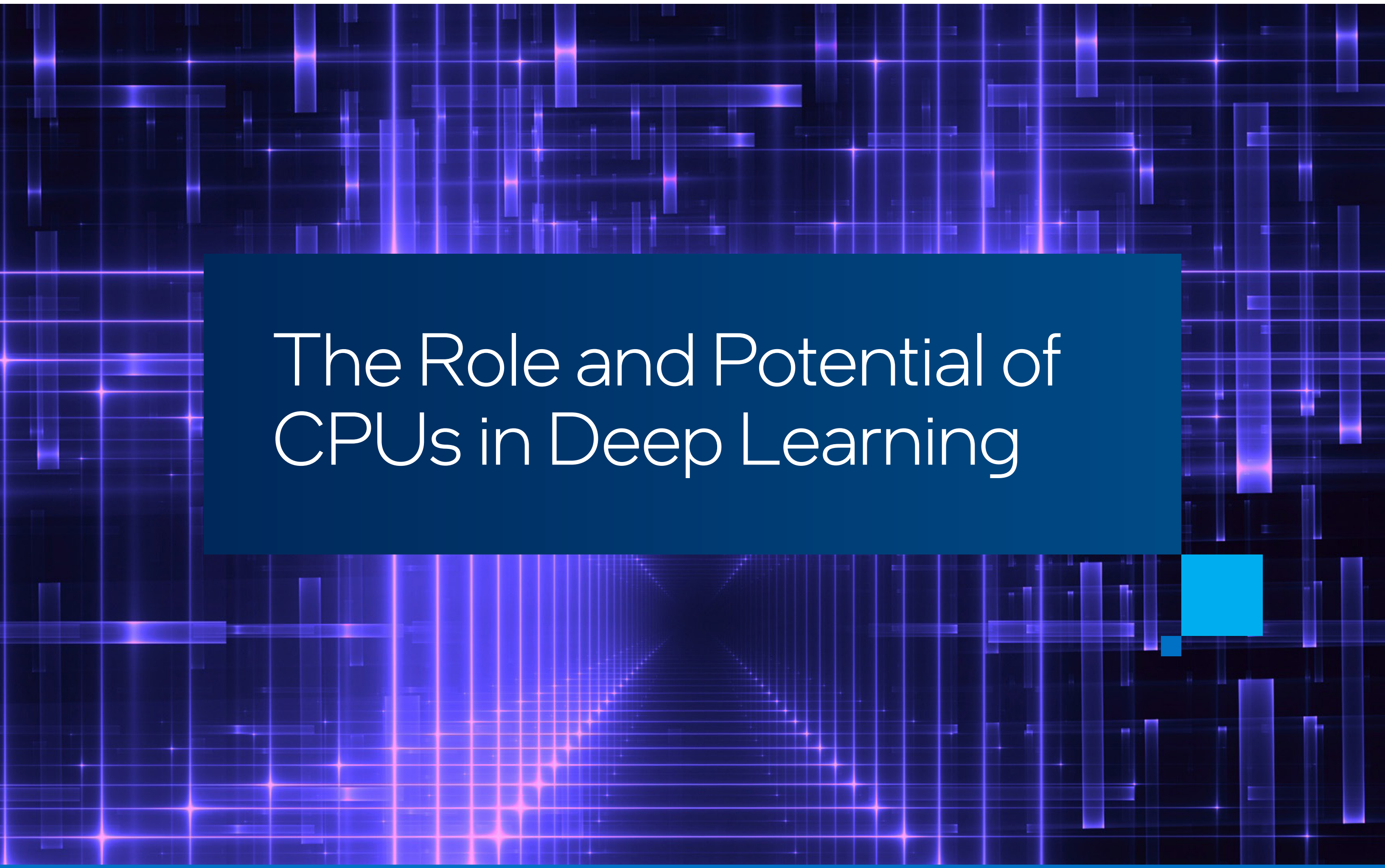


Figure 11. Final speedup obtained with Intel MPI Library

Figure 11 summarizes the performance gains from our optimization effort. The single-node optimizations achieved a 2x speedup, while multi-node optimizations resulted in a further 8x speedup, bringing overall time-to-solution down by a factor of 16, with no loss in model accuracy.



The Role and Potential of CPUs in Deep Learning

Sparsh Mittal, Assistant Professor, Indian Institute of Technology, Roorkee, India

[This article originally appeared in *HPCwire* and is reprinted with permission.]

In this invited guest piece, Sparsh Mittal provides perspective on the role of the central processing unit (CPU) for deep learning workloads in an increasingly diverse processor space, reviewing use cases where the performance of the CPU excels, and noting some of the architectural changes and directions spurred by deep learning applications. The article serves as an introduction to a new survey research paper (written by Mittal et al.) published this April in IEEE Transactions on Neural Networks and Learning Systems.

Deep learning (DL) applications have unique architectural characteristics and efficiency requirements. Hence, the choice of computing system has a profound impact on how large a piece of the DL pie a user can finally enjoy. Even though accelerators may provide higher throughput than general-purpose computing systems (CPUs), there are several other metrics and usage scenarios on which CPUs are preferred or are superior. A recent survey paper I've coauthored with Poonam Rajput and Sreenivas Subramoney (A Survey of Deep Learning on CPUs: Opportunities and Co-optimizations) highlights the strengths of CPUs in DL, and identifies opportunities for further optimization.

A CPU Has Its Forte, and an Accelerator Is Not a Panacea

Sparse deep neural networks (DNNs) are inefficient on massively parallel processors because of their irregular memory accesses and inability to leverage optimizations such as cache tiling and vectorization. Further, recurrent neural networks (RNNs) are difficult to parallelize due to the dependencies between the steps. Similarly, DNNs such as InceptionNet variants have filter shapes of 1×1, 3×3, 1×3, 3×1, etc., which lead to irregular memory accesses and a variable amount of parallelism across the layers. CPUs are more suitable for such applications with limited parallelism because of their advanced memory management techniques. For example, researchers from Rice University have shown that for fully connected networks over sparse datasets such as Amazon-670K* and Delicious-200K*, the DL training problem can be modeled as a search problem. This allows replacing matrix multiplication operations with hash tables. Their technique on CPUs provides higher performance than a TensorFlow*-based implementation on GPUs.

3D convolutional neural networks (CNNs), and even 2D CNNs, with large batch-sizes require a massive amount of memory. Since CPU-managed hosts in cloud and data center scenarios have much larger memory capacities than accelerators, running memory-hungry operations on CPUs is not only merely attractive but often imperative. Accelerators such as TPUs provide high throughput for large batch sizes; however, for applications requiring real-time inference, the use of large batch sizes is not preferred. At small batch sizes, CPUs generally provide competitive latency. There are a host of techniques that can be applied to further tune the DL applications on CPUs; for example, hardware-aware pruning, vectorization, cache tiling, and approximate computing. Our survey paper summarizes many such techniques.

Across the Board: From Tiny Wearables to Large Data Centers

IoT devices and wearables have tight power and area budgets, which precludes over-specialization. For example, a smartwatch chip cannot host separate accelerators for speech/audio/image/video processing. In smartphones running Android*, the programming support for mobile GPU or DSP is not fully mature. In fact, on a typical mobile SoC, the theoretical peak performance of mobile CPUs equals that of mobile GPUs. Further, data centers supporting web services such as social networks see a significant fluctuation in computing demand over time. CPUs can meet this variability in demand due to their high availability and efficiency for both DL and non-DL tasks. Finally, in extreme environments, such as defense and medical, which require security certifications, CPUs are sometimes the only platform of choice.

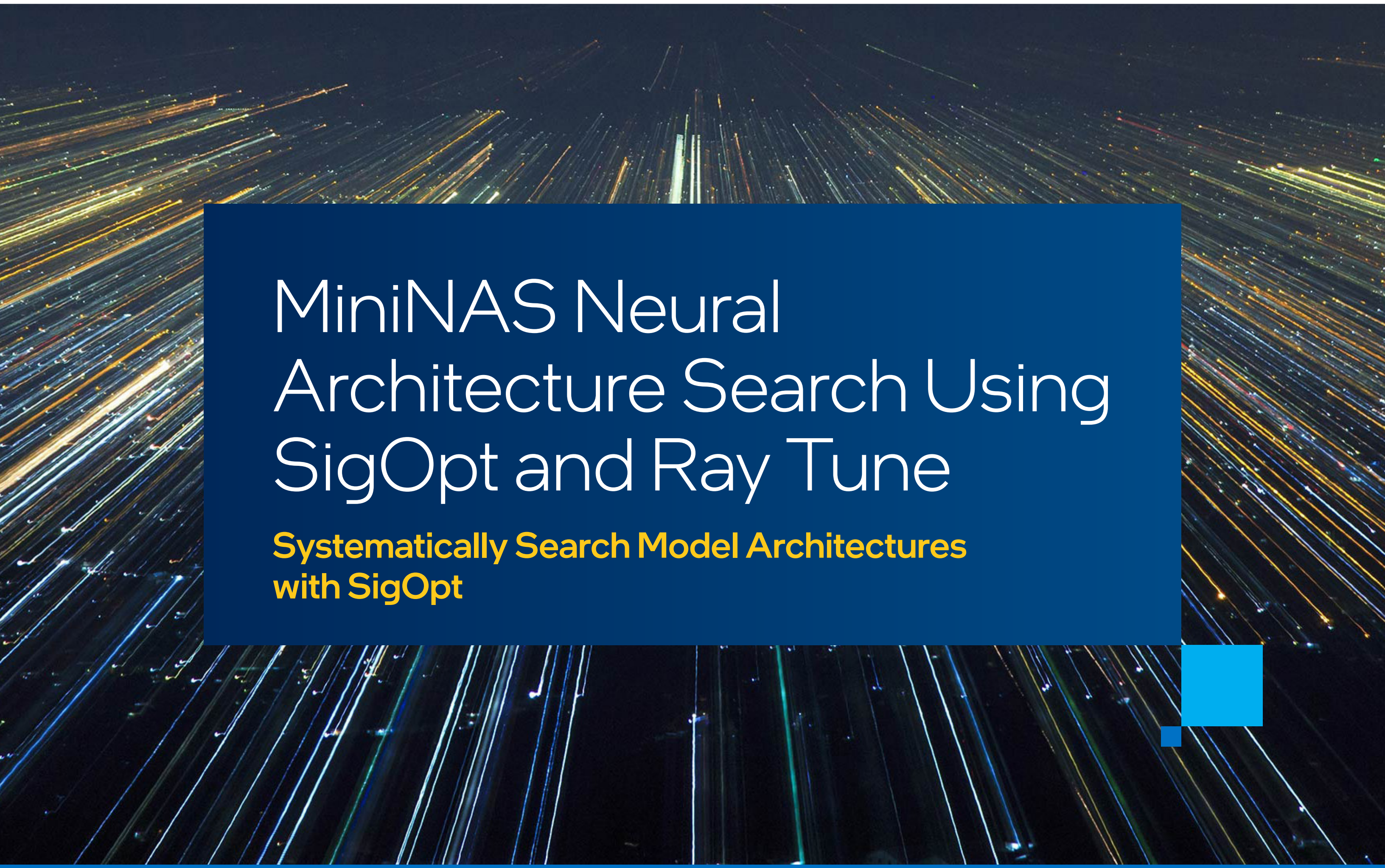
Not Missing the Obvious: Economy and Ease of Use

Accelerators require long design cycles and massive investment. Integrating them into existing ecosystems requires high costs and engineering work. By contrast, the hardware/software stack of CPUs is already well established and understood. They can provide reasonable speedups on a broad range of applications. While large-scale companies have the resources to build and maintain their custom accelerators, CPUs (or GPUs) remain the most feasible platform for other companies.

Future Outlook: Brighter Than You Think

Going forward, merely increasing peak performance will not be sufficient; more revolutionary improvements are required to boost the performance of a broad range of DL applications, such as reinforcement learning and generative adversarial networks. Recent CPUs have begun to provide hardware support for low-precision computing. Once in-memory computing reaches maturity, the large caches of CPUs would turn into massive compute units. Development of open source ISA, such as RISC-V, would further break the portability and proprietary barriers of accelerators.

The metrics of interest are numerous and varied, and so are the state-of-the-art DL models. We believe that instead of a "general-purpose processor versus accelerator" debate, the future will see a CPU-accelerator heterogeneous computing approach that brings together the best of both worlds.



MiniNAS Neural Architecture Search Using SigOpt and Ray Tune

Systematically Search Model Architectures with SigOpt

Ellick Chan, Head of Intel® AI Academy, University Relations and Research and Barrett Williams, SigOpt Product Marketing Lead

Neural architecture search (NAS) is a modern technique to find optimized neural networks (NNs) for a given task, such as image classification, through structural iteration and permutation. Network parameters—such as the depth of the network, the number of convolutional filters, pooling, epochs, and learning rate—can substantially impact a network's accuracy, inference throughput, and latency for a given dataset.

The search space for these parameters is large, so NAS can take many compute-hours to train. In this article, we show how you can use smarter search algorithms provided by SigOpt paired with raw cluster computing power provided by Ray Tune to accelerate this process. We use a simple example so that practitioners can apply this technique to their own workflows.

To illustrate the core concept of NAS, consider the original network in **Figure 1a**. This reference network consists of a single input layer followed by one or more copies of Block 1. Block 1 is based on a convolution-pooling motif consisting of 3×3 convolutions with 32 filters, optionally followed by a pooling operation. This pattern continues with one or more copies of Block 2, similarly composed of 32 3×3 convolutional filters. These convolutional blocks are then flattened to a vector, processed through a fully connected layer, and topped off with a softmax function for final classification.

NAS helps the data scientist test a variety of permutations of a reference architecture. **Figure 1b** shows one option, called "depth scaling," in which Block 2 is repeated to increase the effective depth of the network. For good measure, we also optionally add another fully connected layer of 1024 neurons. In this tutorial, the two fully connected layers are the same size, but they can be different sizes in your application.

Figure 1c shows "width scaling," in which the depth of the network remains constant, but parameters on the operators are varied. In this case, we reduce the number of convolutions in L1 (layer 1) from 32 to 16, and increase the number of convolutions in L2 from 32 to 64. We also make the fully connected layer wider, going from 1024 to 2048. Note that NAS doesn't have to search for all the parameters at once. It's possible to optimize one parameter at a time and fix the others; or, if your optimizer is intelligent like SigOpt, it's both possible and more efficient to strategically update multiple parameters at once to find the best network architecture more quickly.

Figure 1d explores one more dimension by challenging our assumption of using 3×3 filters. Instead, we substitute the filters in Block 1 with 5×5 filters and Block 2 with 7×7 filters. This can help the performance of certain models and datasets, depending on data characteristics and input image resolution.

By now, it's fairly clear that even with a simple example, there are a combinatorially large number of NN parameters to customize and explore. In the rest of this article, we will show you how to use SigOpt and Ray Tune to fine tune the space of simple NN used to classify images in the classic CIFAR-10 dataset.

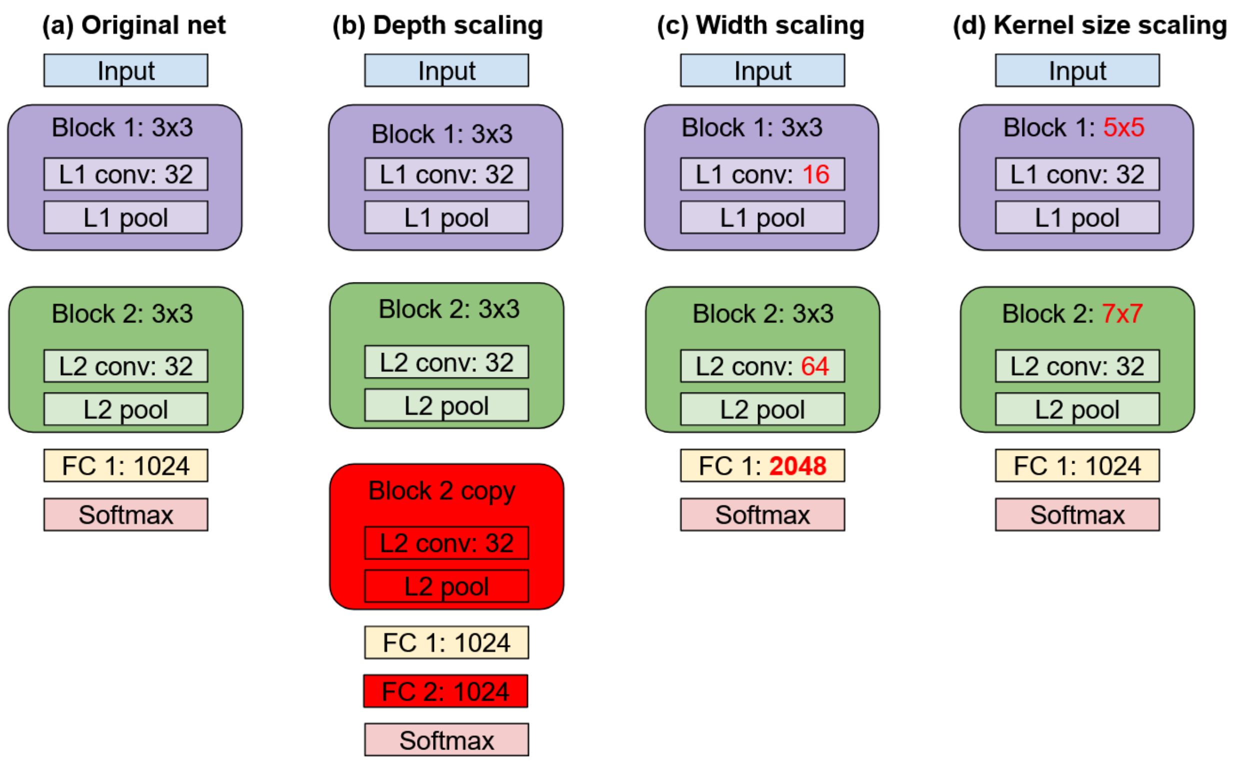


Figure 1. Network variations used in this tutorial

Overall Workflow

1. Define an NN training task: Choose a dataset and a model template (e.g., CIFAR-10 or a convolutional neural network [CNN]) and define the parameters to tune (e.g., number of layers and/or filters).
2. Apply Ray Tune to search for a preliminary set of model parameters.
3. Adapt the search algorithm to SigOpt to get better parameters more efficiently.

Parameterizing the Model

For the purposes of this article, we define an NN training task as a convolutional network with one or more convolutional blocks. We'll use the CIFAR-10 dataset and the Keras* API from TensorFlow*.

```
# Install prerequisites
!pip install -qqq ray[tune] pandas

# Download dataset
from tensorflow.keras.datasets import cifar10
d = cifar10.load_data()

# Ray Tune calls this function many times in parallel with different hyperparameters
def train(config):
    import os, numpy as np
    import psutil
    os.environ['OMP_NUM_THREADS'] = str(int(psutil.cpu_count()/2))
    from tensorflow import keras
    from tensorflow.keras.models import Sequential
    from tensorflow.keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D

    # Load the dataset
    (X_train, Y_train), (X_test, Y_test) = keras.datasets.cifar10.load_data()

    # Create the model
    model = Sequential()

    # Build first convolutional block motif
    model.add(Conv2D(config['nconv0'], kernel_size=(3, 3),
                     activation='relu', input_shape=(32, 32, 3)))
    for i in range(config['nblocks1']):     # Repeat this block nblocks1 times
        if len(model.layers) > config['layers']: break     # Limit depth to "layers"
        # Add nconv1 3x3 conv kernels with relu activation
        model.add(Conv2D(config['nconv1'], kernel_size=(3, 3), activation='relu'))
        # Add pooling
        if config["pooling"] == "True": model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Dropout(0.25))     # Add dropout

    for i in range(config['nblocks2']):
        if len(model.layers) > config['layers']: break
        model.add(Conv2D(config['nconv2'], kernel_size=(3, 3), activation='relu'))
        if config["pooling"] == "True": model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Dropout(0.25))

    model.add(Flatten())     # Flatten output tensor to a vector
    for i in range(config['nfcll']):     # number of fully connected last layers
        if len(model.layers) > config['layers']: break
        model.add(Dense(1024, activation='relu'))

    model.add(Dropout(0.5))
    model.add(Dense(10, activation='softmax'))     # Apply softmax classification
    print("Layers: %d max layers: %d" % (len(model.layers), config['layers']))

    # Compile and setup training
    from tensorflow.keras.utils import to_categorical
    model.compile(loss='categorical_crossentropy',
      # Adam uses an adaptive learning rate that we do not explicitly tune this
      optimizer=keras.optimizers.Adam(lr=0.0001, decay=1e-6), metrics=['accuracy'])

    # Train the model
    model.fit(X_train / 255.0, to_categorical(Y_train),
      batch_size=128, shuffle=True, verbose=0,
      epochs=config["epochs"])

    # Evaluate the model
    scores = model.evaluate(X_test / 255.0, to_categorical(Y_test))
    tune.report(Accuracy=scores[1], Loss=scores[0])
```

To parameterize the model, we define the following:

- `Epochs`: Number of epochs to train a model
- `Layers`: Maximum number of layers of the desired model (subsequent layers are pruned)
- `Nconv0`: Number of 3×3 convolution filters for the input layer
- `Nfcll`: Number of fully connected last layers, with 1,024 neurons each
- `Pooling`: Global setting to enable/disable pooling in convolution blocks 1 and 2
- `Nblocks1`: Number of copies of convolution block 1
- `Nconv1`: Number of 3×3 convolution filters for convolution blocks 1 and 2
- `Nblocks2`: Number of copies of convolution block 2
- `Nconv2`: Number of 3×3 convolution filters for block 2

To be consistent for deploying clusters in Part 2 (to be published later), we'll start Ray from the command line. If you're running on a single node, the following commands aren't necessary:

```
!ray stop
!sleep 3
!nohup ray start --head --num-cpus 1
!sleep 3
import ray
ray.init('localhost:6379')
```

If you're running on a cluster (such as Intel® DevCloud) that uses a job scheduler (e.g., a Portable Batch System), the following commands start worker processes on multiple nodes:

```
!which qsub && echo ray start --address `hostname`:6379 --block --num-cpus 1 | qsub
!which qsub && echo ray start --address `hostname`:6379 --block --num-cpus 1 | qsub
!which qsub && echo ray start --address `hostname`:6379 --block --num-cpus 1 | qsub
!which qsub && sleep 20
```

Finally, set the parameters:

```
from ray import tune
config = {
        "epochs":   tune.randint(20, 30),
        "layers":   tune.randint(1, 20),      # maximum number of layers
        "nconv0":   tune.randint(16, 64),     # input layer
        "nfcll":    tune.randint(0, 2),       # fully connected last layer
        "nblocks1": tune.randint(1, 3),       # conv block 1
        "nconv1":   tune.randint(16, 64),
        "nblocks2": tune.randint(1, 3),       # conv block 2
        "nconv2":   tune.randint(16, 64),
        "pooling":  tune.choice(['True', 'False'])
    }
```

Apply Ray Tune

Ray Tune is a Python* library that facilitates scaled experimentation, as well as hyperparameter optimization via SigOpt, allowing multiple worker nodes to explore the search space in parallel. A naïve grid search of our defined parameter space would explore nearly 1.2 billion possible configurations[1]. In this article, we show how to use a random search to speed up this process, and then follow up with a smarter guided search using SigOpt, effectively comparing the performance and output of the two approaches.

For Ray Tune, the most important inputs are the function to optimize (train) and the search space for the parameters (config). We defined both of these earlier and provide the corresponding code below. Other options include a choice of search algorithm and scheduler for more guided searches.

```
# Common options for Ray Tune
tune_opts = {
    # Number of sample points to try, increase for better results
    'num_samples': 5,
    # Some net configs are invalid (pooling too many times creates negative dim)
    'raise_on_failed_trial': False
}

import subprocess, psutil

# Enable accelerator, if present
try:
  if subprocess.run('nvidia-smi').returncode == 0:
    tune_opts['resources_per_trial'] = {'cpu': 1, 'gpu': 1}
except FileNotFoundError: pass

analysis = tune.run(
    train_wrapper,
    config=config,
    verbose=1,
    **tune_opts)

# To see the full optimization results, inspect the results dataframe
analysis.results_df

# Visualize Ray Tune results
d = analysis.results_df
d.plot.scatter('timestamp', 'Accuracy')
```

Integrating with SigOpt

To sign up for free access to SigOpt, please use this sign-up form. You'll then be able to create an account, which will give you access to an API key that you can use in your Google Colab* notebook or Intel DevCloud Jupyter Notebook*.

[1] This estimate is derived from 10*20*(64-16)*3*3*(64-16)*3*(64-16)*2.

```
# Fill in your SigOpt key here
SIGOPT_TOKEN   = "YOUR_SIGOPT_API_KEY_HERE"
SIGOPT_PROJECT = "raytune-simplenas"
!pip install -qqq sigopt

# Convert Ray Tune parameter space to SigOpt format
def convert_space_to_sigopt(config):
    c = []
    for k, v in config.items():
        print(k,v)
        if isinstance(v, ray.tune.sample.Float):
            c.append({'name': k, 'type': 'double',
                      'bounds': {'min': v.lower, 'max': v.upper}})
        elif isinstance(v, ray.tune.sample.Integer):
            c.append({'name': k, 'type': 'int',
                      'bounds': {'min': int(v.lower), 'max': int(v.upper)}})
        elif isinstance(v, ray.tune.sample.Categorical):
            vals = [{'enum_index': i+1, 'name': str(z),
                     'object': 'categorical_value'}
                    for i, z in enumerate(v.categories)]
            c.append({'name': k, 'type': 'categorical', 'categorical_values': vals})
        else:
            print('Unknown type:', k, type(v))
            raise ValueError
    print('config:', c)
    return c

import ray, os
from ray.tune.suggest.sigopt import SigOptSearch
from ray.tune.schedulers import FIFOScheduler
sigopt_connection = ray.tune.suggest.sigopt.Connection(client_token=SIGOPT_TOKEN)
algo = SigOptSearch(
    convert_space_to_sigopt(config),
    name=SIGOPT_PROJECT,
    connection=sigopt_connection,
    project=SIGOPT_PROJECT,
    max_concurrent=3,
    metric="Accuracy",
    mode="max")

analysis_sigopt = tune.run(
    train_wrapper,
    search_alg=algo,
    verbose=1,
    **tune_opts)

analysis_sigopt.results_df

# Visualize Ray Tune results
a = analysis.results_df[['timestamp', 'Accuracy']]
s = analysis_sigopt.results_df[['timestamp', 'Accuracy']]
a['timestamp'] -= a['timestamp'].min()
s['timestamp'] -= s['timestamp'].min()
ax = a.plot.scatter('timestamp', 'Accuracy', c='b',
                    label='Random Search- Acc %0.3f%%' % a['Accuracy'].max())
s.plot.scatter('timestamp', 'Accuracy', c='r', ax=ax,
               label='SigOpt- Acc %0.3f%%' % s['Accuracy'].max(),
               title='Accuracy vs time')
ax.legend(loc='lower right')
```

Interpreting SigOpt Results

In the example above, we used a limited number of sample points to allow the experiment to complete quickly. If more data points are sampled, you might see a figure like the one shown in **Figure 2**, which illustrates that SigOpt's directed search finds better solutions more efficiently than a random search.

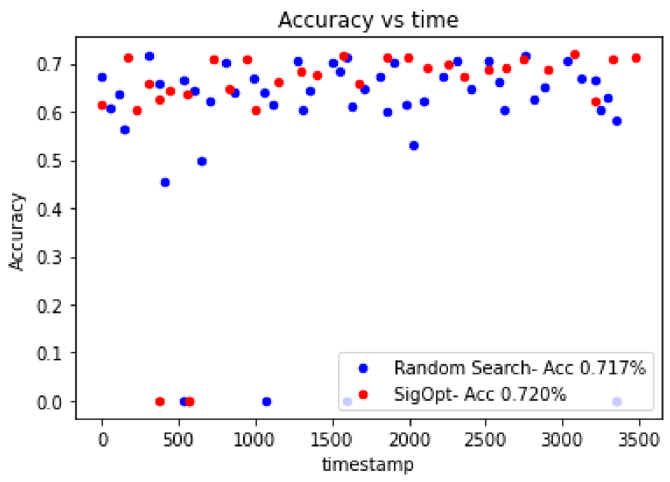
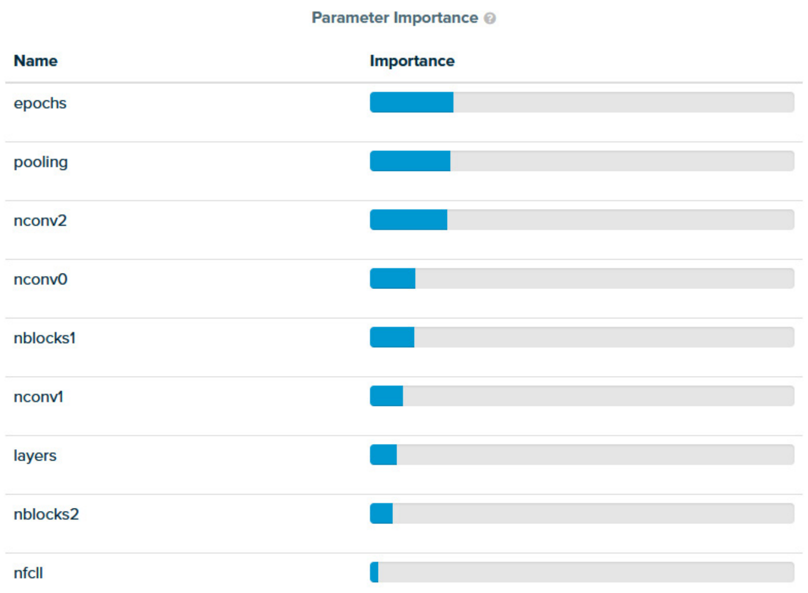


Figure 2. SigOpt directed search vs. random search

SigOpt helps data scientists understand which parameters matter most for their NAS. We see that the number of epochs has the biggest influence on the accuracy of the network, followed by pooling layers, and then the number of convolutional filters (**Figure 3**). SigOpt searches this space intelligently to find the best values more efficiently.

To help data scientists understand the influence of various parameters, SigOpt visualizes the relative parameter importance with respect to the points sampled. Note that this is a bit of a biased sample, as the points are chosen intelligently by the optimizer (instead of at random).



epochs
pooling
nconv2
nconv0
nblocks1
nconv1
layers
nblocks2
nfcll

Figure 3. Determining which parameters matter the most

Given the relative importance of the parameters, we examine the relationship between convolutional filter parameters **nconv0** and **nconv1** and find that this particular problem prefers around 50 filters for **nconv0** and a small number of filters for **nconv1** (**Figure 4**). Any pair of variables can be visualized in this plot.

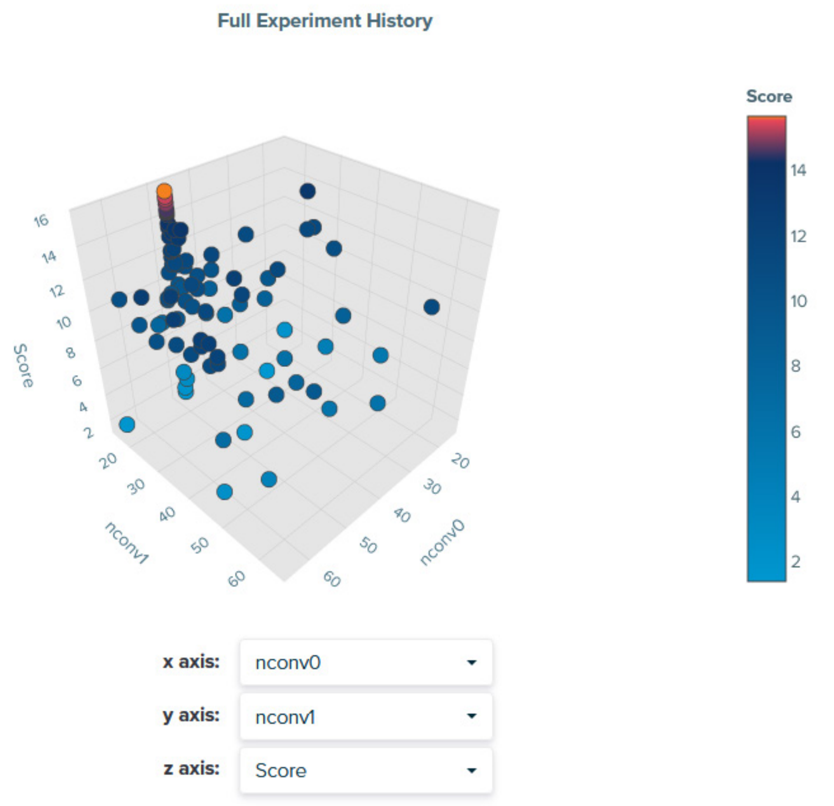


Figure 4. Visualizing the relationship between parameters

A parallel coordinate plot shows the trajectory of the parameter search (**Figure 5**). In this case, the highest scores are obtained with a larger number of epochs, pooling, and different combinations of layer parameters. This plot shows what this particular problem prefers. If the dataset or objective is changed, the preferred parameters may differ.

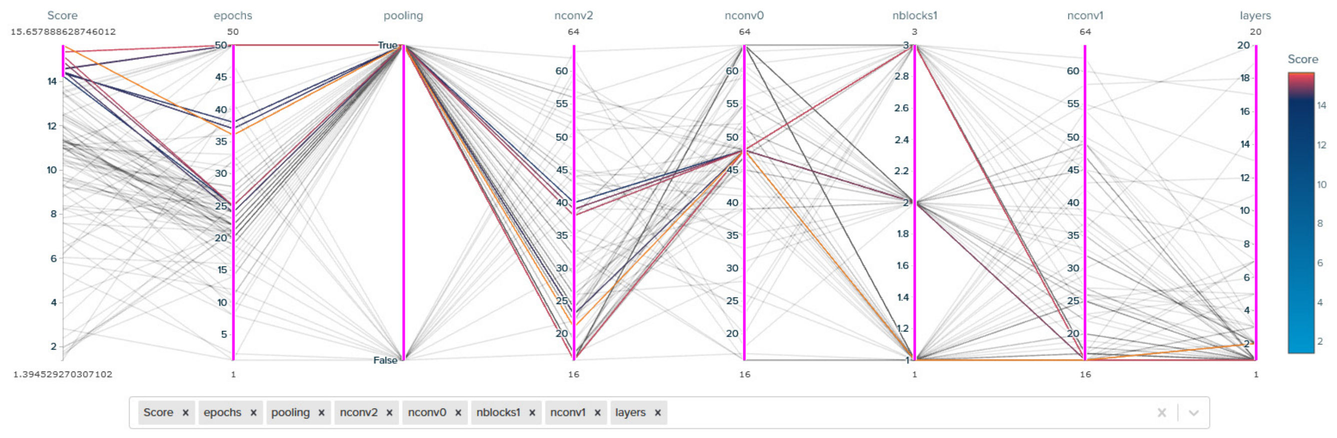
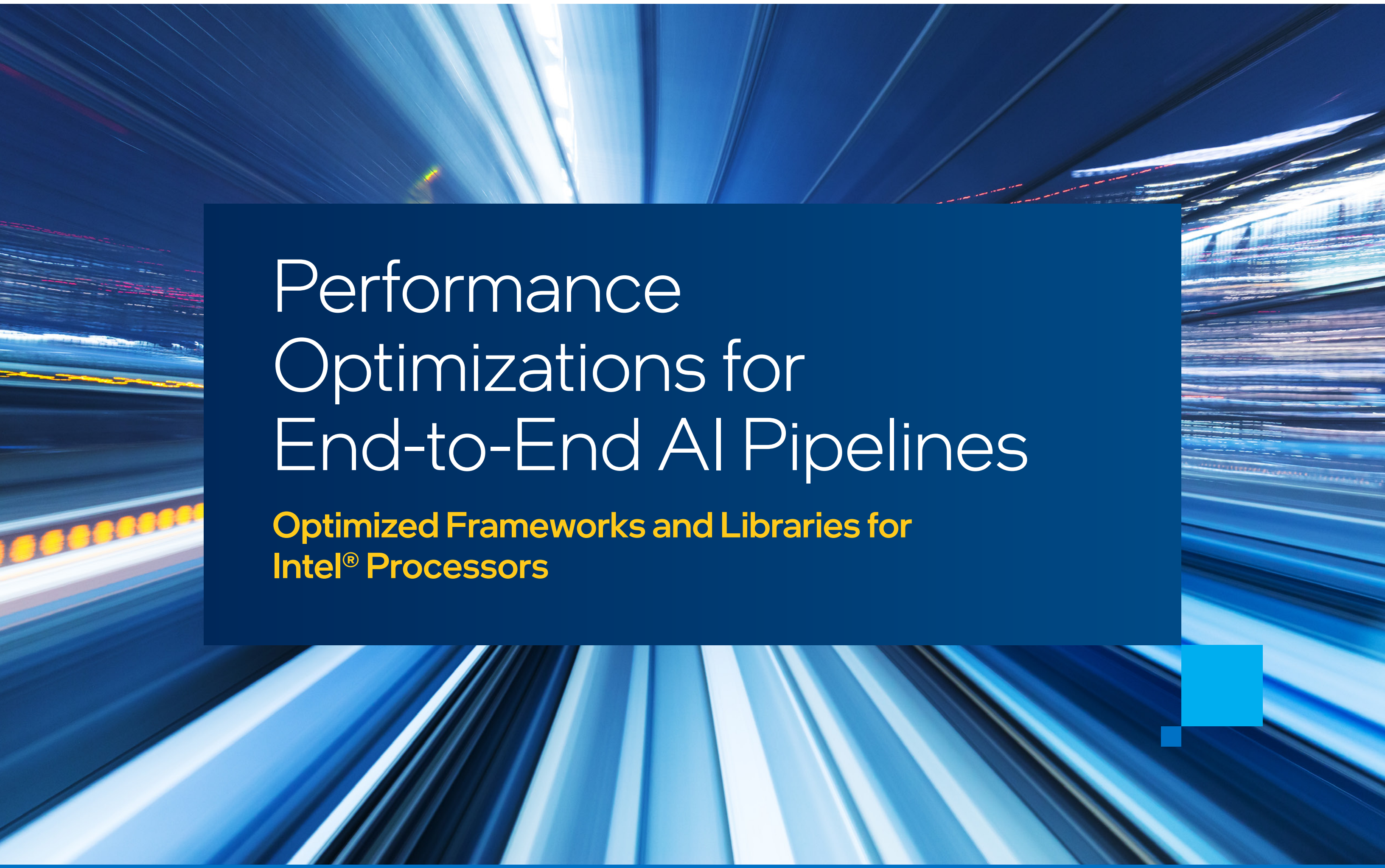


Figure 5. Trajectory of the parameter search

Understanding the relationships between the parameters helps data scientists better optimize parameter values for the problem and better manage tradeoffs. Be sure to sign up for free access to SigOpt, and start optimizing, tracking, and systematizing today.



Performance Optimizations for End-to-End AI Pipelines

Optimized Frameworks and Libraries for Intel® Processors

Meena Arunachalam, Principal Engineer, Intel Corporation

Modern artificial intelligence (AI) and machine learning (ML) applications perform a range of tasks that convert raw data into valuable insights. Data scientists create multiphase, end-to-end pipelines for AI and ML applications (**Figure 1**). The phases include data ingestion, data cleaning, data exploration, and feature engineering, followed by prototyping, model building, and, finally, model deployment. The phases are often repeated many times, and it may be necessary to scale the entire pipeline across a cluster and/or to deploy it to the cloud.

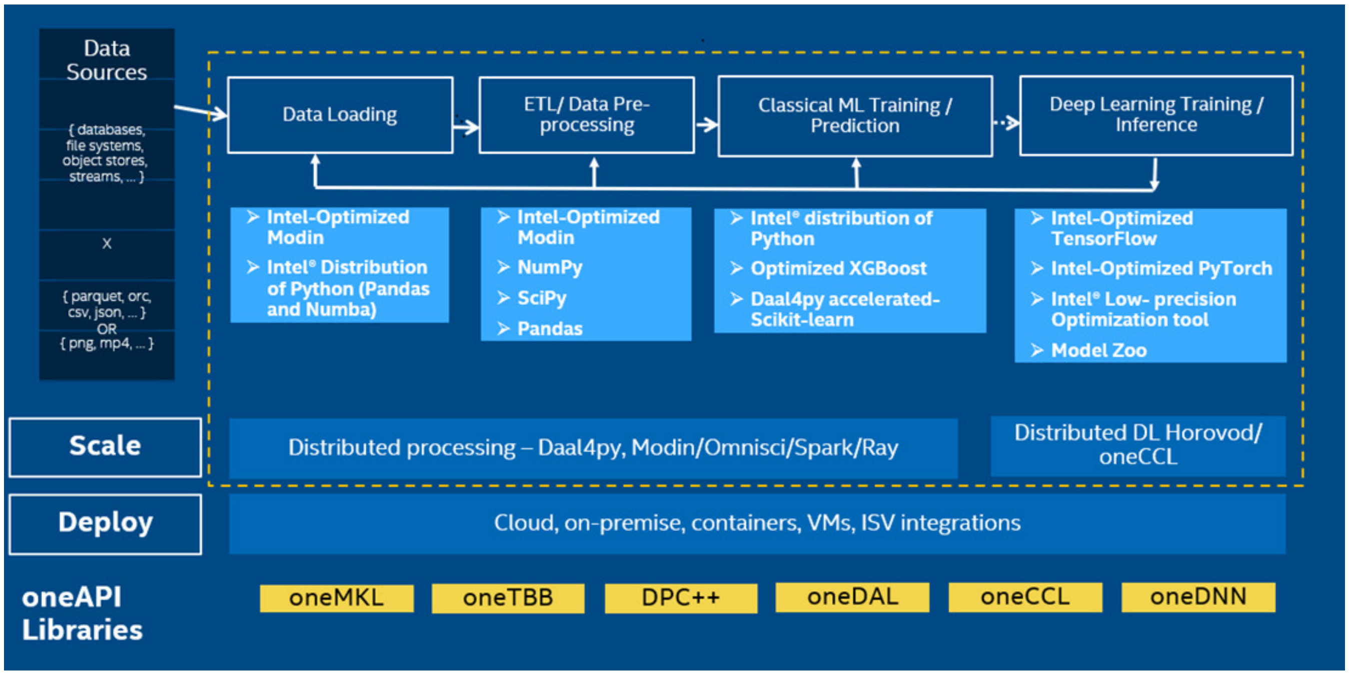

Data Sources
{ databases, file systems, object stores, streams, ... }
X
{ parquet, orc, csv, json, ... } OR { png, mp4, ... }
Data Loading
ETL/ Data Pre-processing
Classical ML Training / Prediction
Deep Learning Training / Inference
Intel-Optimized Modin
Intel® Distribution of Python (Pandas and Numba)
Intel-Optimized Modin
NumPy
SciPy
Pandas
Intel® distribution of Python
Optimized XGBoost
Daal4py accelerated-Scikit-learn
Intel-Optimized TensorFlow
Intel-Optimized PyTorch
Intel® Low- precision Optimization tool
Model Zoo
Scale
Distributed processing – Daal4py, Modin/Omnisci/Spark/Ray
Distributed DL Horovod/ oneCCL
Deploy
Cloud, on-premise, containers, VMs, ISV integrations
oneAPI Libraries
oneMKL
oneTBB
DPC++
oneDAL
oneCCL
oneDNN

Figure 1. End-to-end analytics pipeline

The Intel® oneAPI AI Analytics Toolkit provides high-performance APIs and Python* packages to accelerate the phases of these pipelines (**Figure 2**).

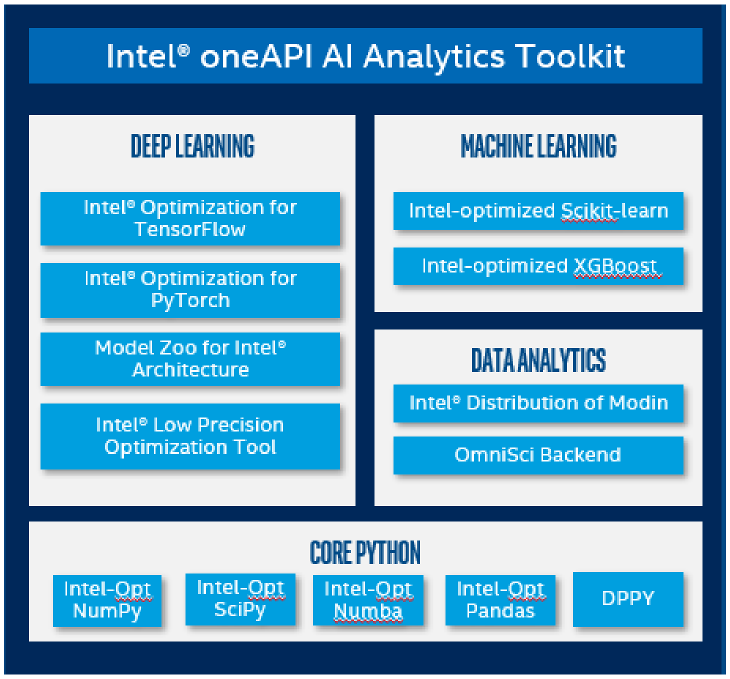

Intel® oneAPI AI Analytics Toolkit
DEEP LEARNING
Intel® Optimization for TensorFlow
Intel® Optimization for PyTorch
Model Zoo for Intel® Architecture
Intel® Low Precision Optimization Tool
MACHINE LEARNING
Intel-optimized Scikit-learn
Intel-optimized XGBoost
DATA ANALYTICS
Intel® Distribution of Modin
OmniSci Backend
CORE PYTHON
Intel-Opt NumPy
Intel-Opt SciPy
Intel-Opt Numba
Intel-Opt Pandas
DPPY

Figure 2. Intel AI Analytics Toolkit

Intel® Distribution of Modin*, with the OmniSci* DB engine, provides a scalable pandas API by simply changing a single line of code. Modin* significantly improves the performance and scalability of pandas dataframe processing.

For classical ML training and inference, the Intel oneAPI AI Analytics Toolkit contains Intel Extension for Scikit-learn extension to accelerate common estimators (e.g., logistic regression, singular value decomposition, principal component analysis, etc.), transformers, and clustering algorithms (e.g., k-means, DBSCAN).

For gradient boosting, Intel also optimized the XGBoost* and CatBoost libraries, which provide efficient parallel tree boosting used to solve many data science problems in a fast and accurate manner.

Let's look at two examples where the Intel oneAPI AI Analytics Toolkit helps data scientists accelerate their AI pipelines:

1. **Census:** This workload trains a ridge regression model to predict education level using U.S. census data (1970 to 2010, published by IPUMS).
2. **PLAsTiCC Astronomical Classification:** This workload is an open data challenge on Kaggle* with the aim to classify objects in the night sky. It uses simulated astronomical time series data to classify objects.

Both workloads have three broad phases:

1. **Ingestion** loads the numerical data into dataframes.
2. **Preprocessing and Transformation** runs a variety of ETL operations to clean and prepare the data for modeling, such as dropping columns, dealing with missing values, type conversions, arithmetic operations, and aggregation.
3. **Data Modeling** creates separate training and test sets, model building and training, model validation, and inference.

Figure 3 shows the breakdown by phase for both workloads, which illustrates the importance of optimizing each phase to speed up the entire end-to-end pipeline.

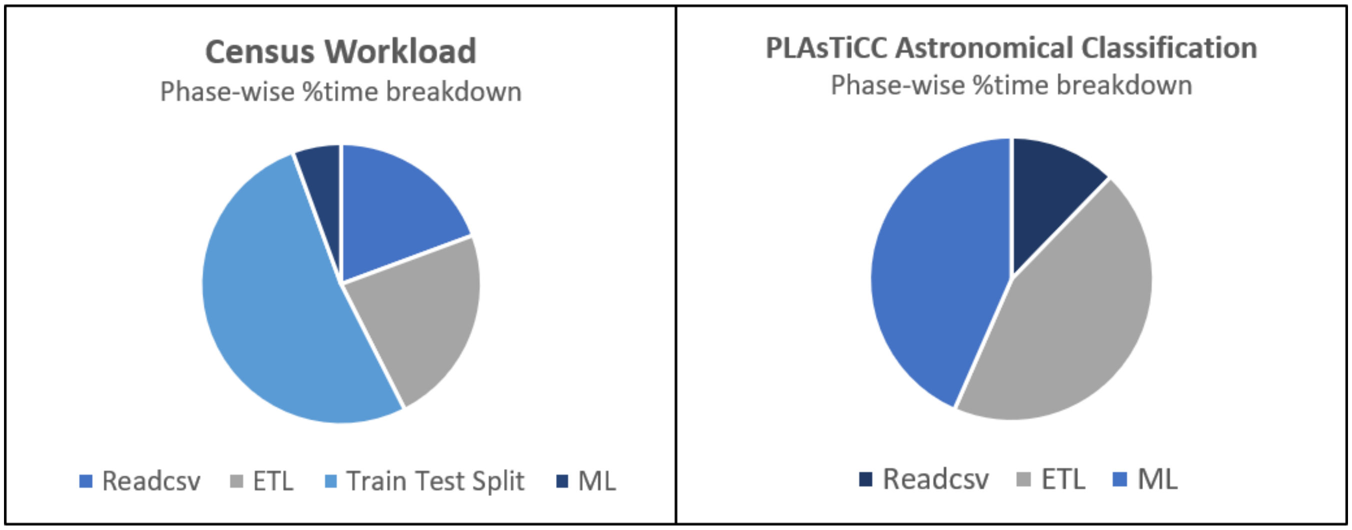


Figure 3. Breakdown by phase for each workload

Figures 4 and **5** show the relative performance and the subsequent speedups for each phase using the Intel-optimized software stack (shown in blue) compared to the stock software (shown in orange). On 2nd Generation Intel® Xeon® Scalable processors, the optimized software stack gives a 10x speedup on Census and an 18x speedup for PLAsTiCC compared to the stock software stack.

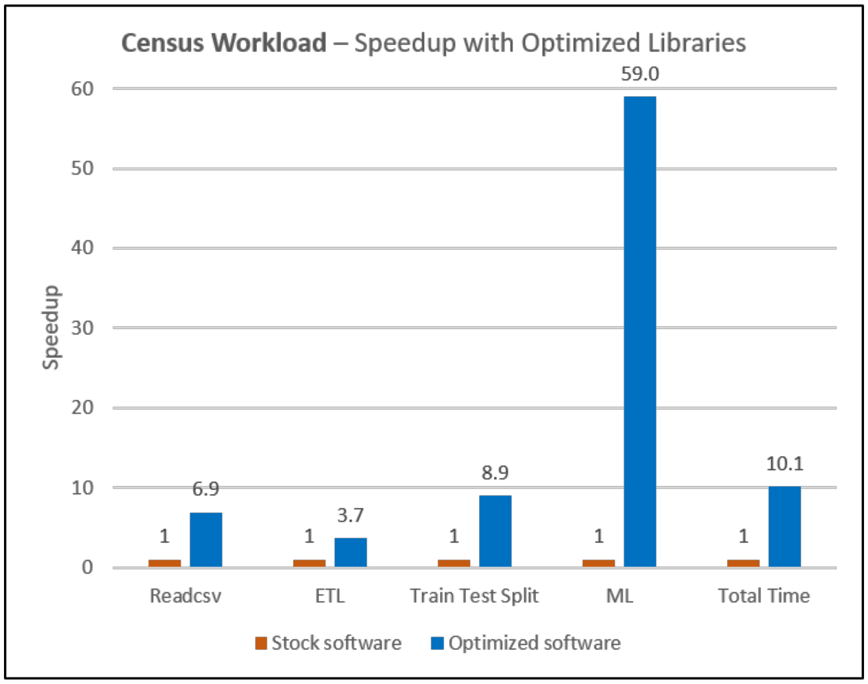


Figure 4. End-to-end performance of the Census workload showing the speedup for each phase

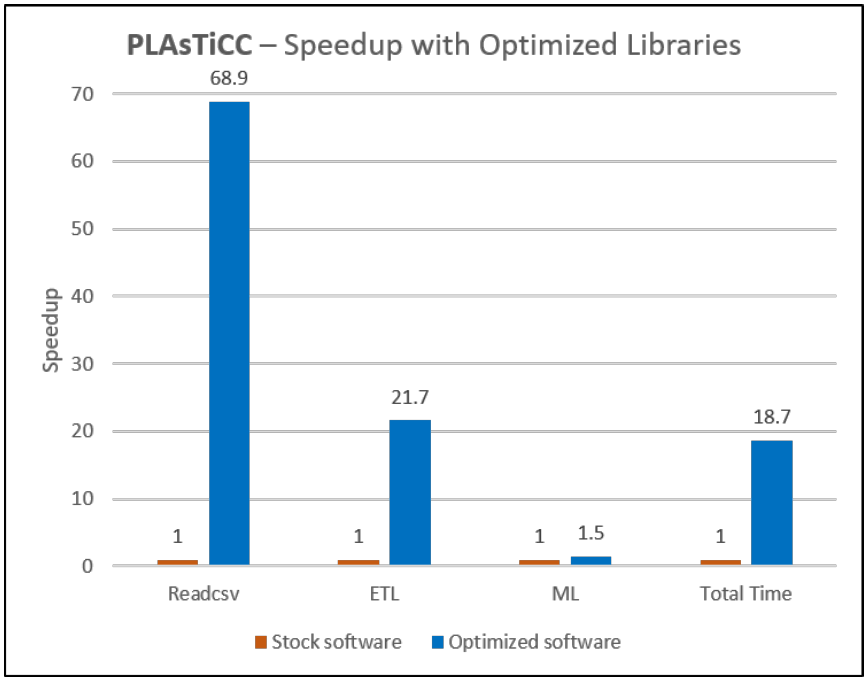


Figure 5. End-to-end performance of the PLAsTiCC workload showing the speedup for each phase

On Census, using the Intel Distribution of Modin* instead of pandas gives a 7x speedup for readcsv and a 4x speedup for ETL operations. For training and prediction, using the Intel-optimized scikit-learn instead of the stock package gives a 9x speedup for the train_test_split function and a 59x speedup for training and inference. On PLAsTiCC, using the Intel Distribution of Modin* gives a 69x speedup for readcsv and a 21x speedup for ETL operations. These speedups are achieved through a variety of optimizations in the Intel oneAPI AI Analytics Toolkit, including parallelization, vectorization, core scaling, improved memory layouts, cache reuse, cache-friendly blocking, efficient memory bandwidth usage, and more effective use of the processor instruction sets.

Figures 6 and **7** show the end-to-end performance of the two workloads on 2nd and 3rd Generation Intel Xeon Scalable processors compared to 2nd and 3rd Generation AMD EPYC* processors and NVIDIA Tesla* V100 and A100 processors. The same optimized software stack is used on the Intel and AMD* CPUs, while the RAPIDS stack is used on the NVIDIA* GPUs. The complete hardware and software configurations are included below.

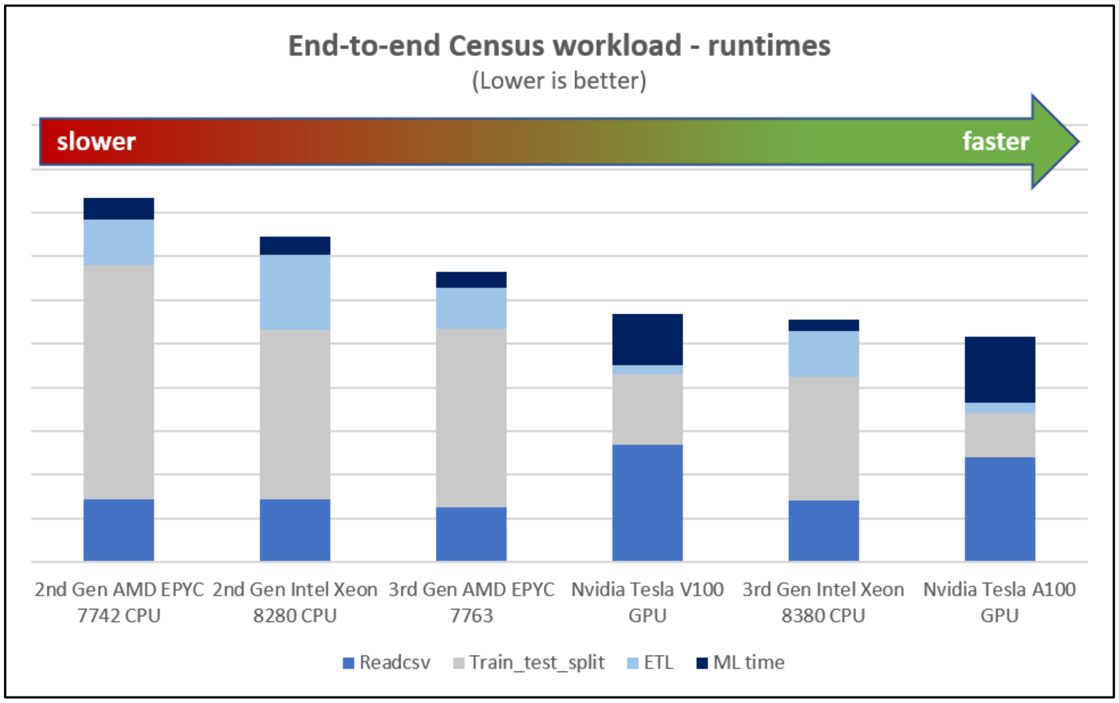


Figure 6. Competitive performance for all phases of the Census pipeline

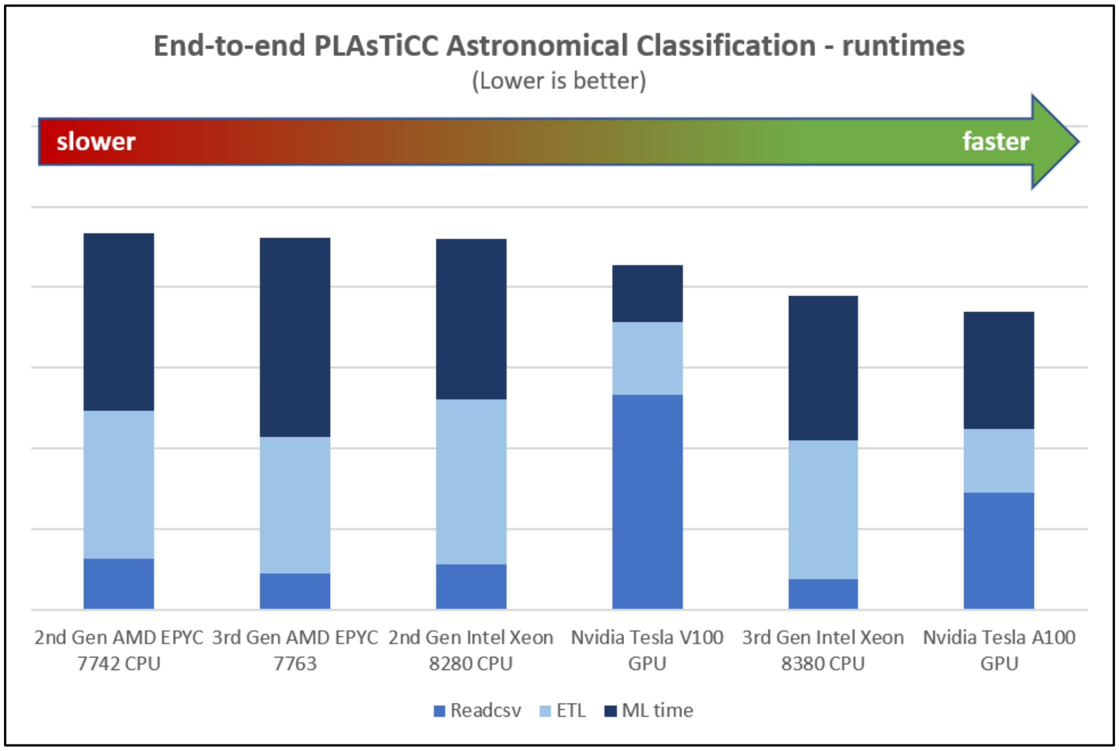


Figure 7. Competitive performance for all phases of the PLAsTiCC pipeline

In this article, we demonstrate a significant performance boost (~10x–18x speedup) on Intel Xeon processors using optimized software packages with simple drop-in replacement over stock data analytics software. The results also show that CPUs and GPUs excel in different phases of the pipelines, but the 3rd Generation Intel Xeon Platinum 8380 processor outperforms the NVIDIA* V100 and is competitive with the NVIDIA* A100. The 3rd Generation Intel Xeon processor is also cheaper and more power-efficient. These observations reinforce the notion that generality is critical in data analytics.

You can get Modin*, XGboost*, the scikit-learn extension, and other optimized software for Intel® architectures through many common channels such as Intel's website, YUM, APT, Anaconda*, etc. Select and download the distribution package that you prefer and follow the Get Started Guide for post-installation instructions.

Hardware and Software Configurations

3rd Generation Intel Xeon Platinum 8380: dual-socket server, 40 cores per socket, 2.30 GHz base frequency, Turbo mode enabled, hyperthreading enabled. OS: Ubuntu* 20.04.1 LTS, 512GB RAM (16x 32GB 3200MHz), kernel: 5.4.0–64-generic, microcode: 0x8d055260, BIOS: SE5C620.86B.OR.64.2021.10.3.02.0417, CPU governor: performance.

2nd Generation Intel Xeon Platinum 8280L: dual-socket server, 28 cores per socket, 2.70 GHz base frequency, Turbo mode enabled, hyperthreading enabled. OS: Ubuntu* 20.04.1 LTS, 384GB RAM (12x 32GB 2933MHz), kernel: 5.4.0–65-generic, microcode: 0x4003003, BIOS: SE5C620.86B.OR.64.2020.51.2.04.0651, CPU governor: performance.

3rd Generation AMD EPYC* 7763: dual-socket server, 64 cores per socket, 1.50 GHz base frequency, simultaneous multithreading enabled. OS: Red Hat Enterprise* Linux 8.3 (Ootpa), 1024 GB RAM (16x 64GB 3200MHz), kernel: 4.18.0–240.el8.x86_64, microcode: 0xa001119, BIOS: Gigabyte version M03, CPU governor: performance.

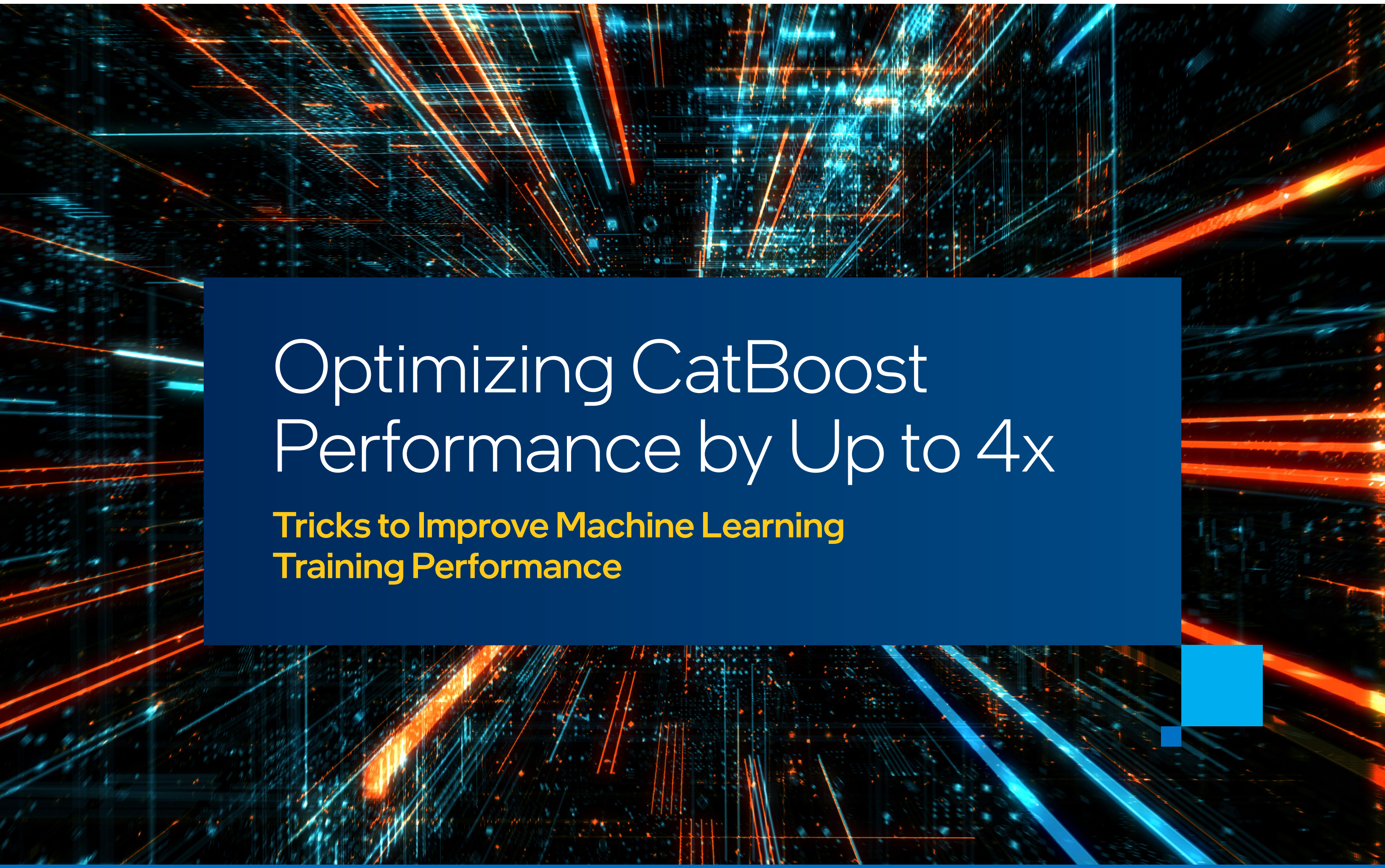
2nd Generation AMD EPYC* 7742: dual-socket server, 64 cores per socket, 1.50 GHz base frequency, simultaneous multithreading enabled. OS: Ubuntu* 20.04.1 LTS, 512 GB RAM (16x 32GB 3200MHz), kernel: 5.4.0–62-generic, microcode: 0x8301038, BIOS: American Megatrends Inc* 2.1c, CPU governor: performance.

NVIDIA Tesla* A100 GPU: Part of DGX-A100, dual-socket 2nd generation AMD EPYC* 7742 host CPU. OS: Ubuntu* 18.04.5 LTS, 512 GB RAM (16x 32GB 3200MHz), kernel: 5.4.0–42-generic, microcode: 0x8301034, BIOS revision 0.23, CPU governor: performance.

NVIDIA Tesla* V100 GPU: 32GB GPU, dual-socket 2nd generation Intel Xeon Platinum 8268 host CPU. OS: CentOS* Linux release 7.8.2003, 384 GB RAM (12x 32GB 2933MHz), kernel: 5.4.69, microcode: 0x5003003, BIOS SE5C620.86B.OR.64.2020.51.2.04.0651, CPU governor: performance.

CPU SW: Scikit-learn 0.24.1 accelerated by daal4py 2021.2, modin* 0.8.3, omniscidbe v5.4.1, Pandas 1.2.2, XGBoost 1.3.3, Python 3.9.7

GPU SW: NVIDIA* RAPIDS 0.17, CUDA* Toolkit 11.0.221, Python* 3.7.9



Optimizing CatBoost Performance by Up to 4x

Tricks to Improve Machine Learning Training Performance

Kirill Shvets, Machine Learning Engineer, Intel Corporation

Gradient Boosting for Data Science

There are plenty of well-known gradient boosting frameworks that deliver accuracy and efficiency in real-world applications. They are regarded as a multipurpose tool to deal with many types of machine learning problems.

According to the latest Kaggle* 2020 survey, 61.4% of data scientists use gradient boosting (such as XGBoost*, CatBoost, or LightGBM) on a regular basis, and these frameworks are more commonly used than the various types of neural networks. Therefore, reducing the computational cost of gradient boosting is critical.

This article covers the CatBoost gradient boosting library. Compared to other libraries, CatBoost effectively handles categorical features and provides a larger variety of growing policies. For example, the SymmetricTree policy reduces the variance of a trained model and significantly improves training time. CatBoost v0.25 introduces optimizations that accelerate training up to 4x compared to the previous release (**Figure 1**).

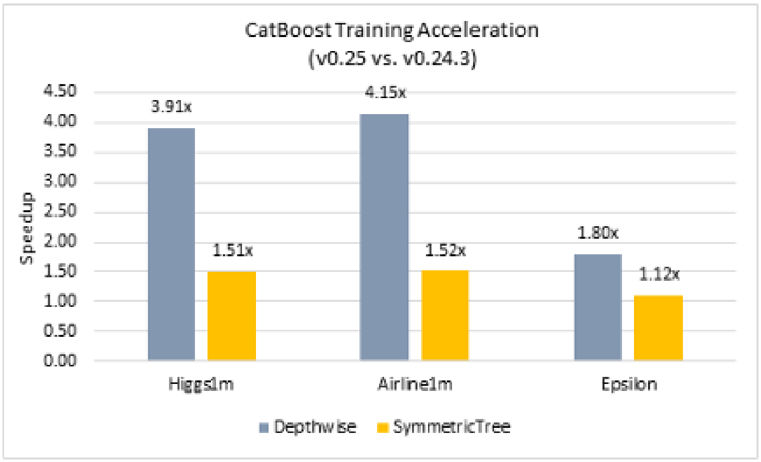


Figure 1. Relative speedup of CatBoost v0.25 over v0.24.3 for the Higgs1m and Airline1m datasets [1] and the Epsilon dataset [2]

Let's have a closer look at these latest CatBoost optimizations.

Optimization Details

The training stage of gradient boosting is quite complex. There are many computational kernels requiring specific optimizations to mitigate irregular memory access patterns, parallelize loops with dependencies, and eliminate branch misprediction. Some optimizations for memory-bound workloads (e.g., using the smallest integer type, int8) were already implemented in earlier versions of CatBoost; however, there were still opportunities for enhancement (i.e., using subtraction for histogram calculation, reducing threading overheads, and introducing a more effective default threading layer backend). Let's go through these latest optimizations.

The Subtraction Trick

With the Depthwise growing policy, a tree is built level by level in each training iteration. This common approach can be parallelized efficiently over the feature columns and over node construction for nodes on the same tree level. Before training, CatBoost performs a quantization of feature columns to bins. The number of bins is controlled by the `max_bin` parameter. During training, CatBoost computes a histogram for each decision tree node. Histogram calculations consist of finding the sum of gradients and Hessians for each bin.

Histogram calculations are the most compute-intensive part of the training stage; however, a histogram's additivity offers a solution to optimize the histogram calculation process. Because the histogram of a parent node is equal to the sum of the histograms of its child nodes, there is no need to compute the exact histogram for each node on the same level. Instead, it is possible to calculate the histogram for the smallest child of the parent node, and then compute the largest child's histogram by subtracting the smallest child's histogram from the previously saved parent's histogram.

This algorithmic optimization was added to the main branch of the CatBoost GitHub repository. It significantly reduces the training time when the Depthwise growing policy is used (**Table 1**).

CatBoost Performance (Depthwise) Speedup	Higgs1m	Airline1m	Epsilon
	2.37	3.14	1.53

Table 1. The speedup in CatBoost training with the Depthwise policy going from v0.24.3 to v0.25.

Threading Layer Improvements

The performance improvements described in the previous section would not have been possible without threading fixes made by the CatBoost maintainers. Significant threading overhead was observed with Intel® VTune™ Profiler. Much of this overhead was eliminated by increasing the size of each task, but it did not solve the root cause of the issue.

We improved the threading layer by providing an alternative to the custom CatBoost threading layer backend: the Intel® oneAPI Threading Building Blocks (oneTBB) library. Integrating the mature, well-supported, and highly optimized oneTBB library into CatBoost resolved the threading overhead problem by providing more effective task scheduling and better nested threading (**Table 2**).

CatBoost Performance (SymmetricTree)	Higgs1m, s	Airline1m, s	Epsilon, s
Before oneTBB integration	76.8	64.8	27.7
After oneTBB integration	50.9	42.8	24.8
Speedup	1.51	1.51	1.12

Table 2. The performance improvements in CatBoost training with the SymmetricTree policy going from v0.24.3 to v0.25. Times are in seconds.

When using the SymmetricTree policy, oneTBB integration improved performance up to 1.5x over the previous threading layer. The CatBoost maintainers report up to 2x speedups. When combined with the subtraction trick, the oneTBB threading layer improved the training performance for the Depthwise policy (1.65x speedup for Higgs1m, 1.32x speedup for Airline1m, and 1.17x speedup for Epsilon) and yielded total improvements of 3.9x for Higgs1m, 4.1x for Airline1m, and 1.8x for Epsilon.

As of CatBoost v0.25, oneTBB is the default threading layer.

Conclusions

If you're using CatBoost to train machine learning models, be sure to use the latest version. Up to 4x speedup can be obtained from the optimizations in v0.25, and there's still more that can be done to improve CatBoost performance. Further core scalability improvements, better memory bandwidth utilization, and vector instructions usage are just a few examples that we expect to be added in future releases.

Hardware and Software Configurations

Intel® Xeon® Platinum 8280L (2nd generation Intel Xeon processors): 2 sockets; 28 cores per socket; HT: on; Turbo: on; total memory of 384 GB (12 slots/16 GB/2933 MHz). CatBoost 0.24.3 (before optimizations), 0.25 (after optimizations); NumPy 1.15.1; Scikit-learn 0.24.1, oneTBB 2021.2.

Training Parameters

Higgs1m [1] and Airline1m [1]:

```
{iterations=1000, learning_rate=0.1, grow_policy='Depthwise/SymmetricTree',
score_function='L2', border_count=256, scale_pos_weight=2, l2_leaf_reg=1,logging_
level='Silent', depth=8}
```

Epsilon [2]:

```
{max_depth: 8, learning_rate: 0.1, reg_lambda: 0, iterations=500, grow_
policy='Depthwise/SymmetricTree', logging_level='Silent'}
```

1. The Airline1m and Higgs1m datasets are available in the dmlc/xgboost benchmark repository.
2. The Epsilom dataset is available in the Nvidia benchmark repository.

THE PARALLEL UNIVERSE

Printed in USA 707/IH Please Recycle.