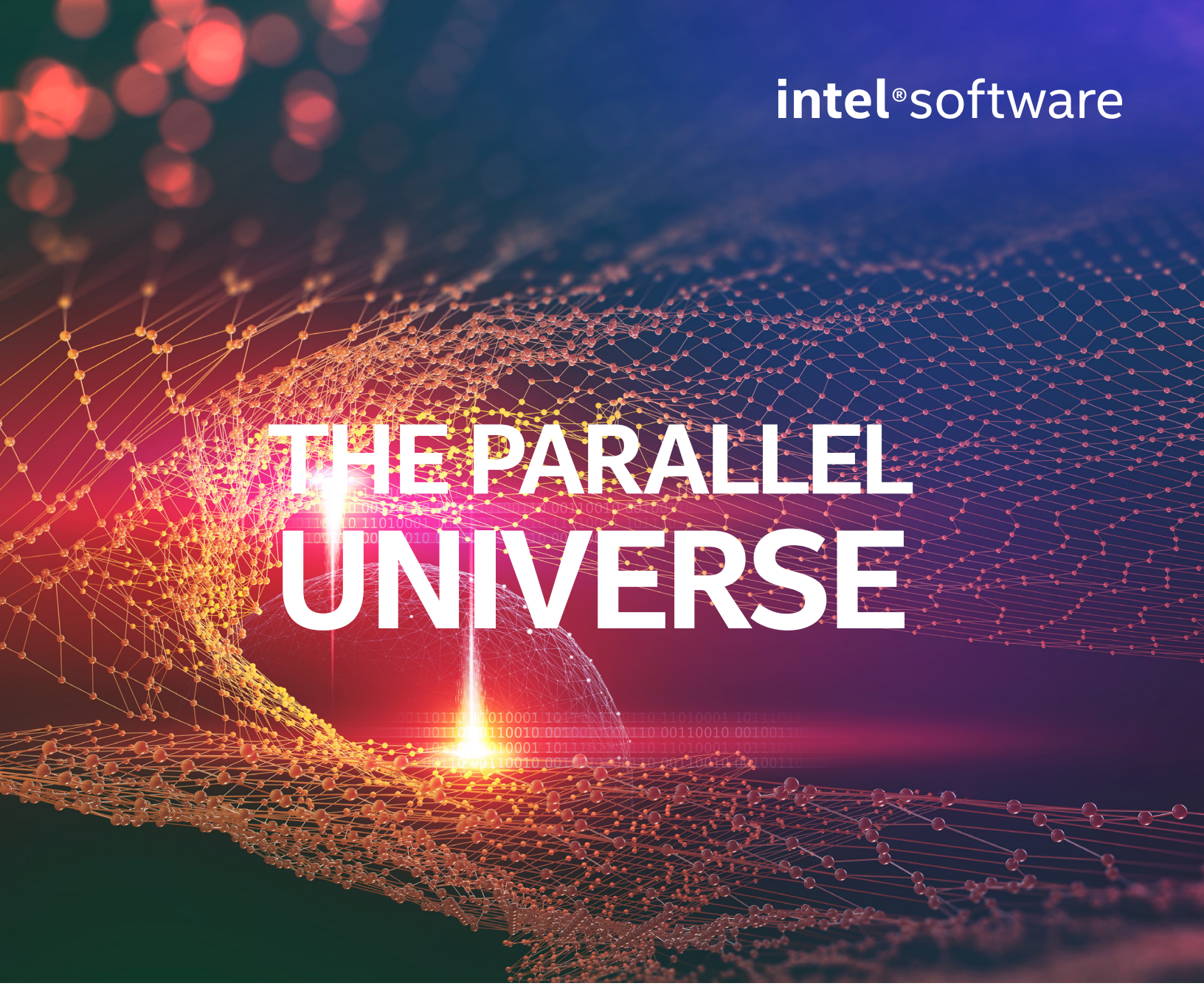


Programming Data Parallel C++

Building an Open, Standards-Based Ecosystem for Heterogeneous Parallelism

A Vendor-Neutral Path to Math Acceleration

How to Speed Up Performance by Exploring GPU Configurations

Issue 42 2020

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.

One Year Into the oneAPI Era

It's a New World of Open, Standards-Based Heterogeneous Parallelism

It was about this time last year that Raja Koduri, Intel's senior vice president, chief architect, and general manager of Architecture, Graphics, and Software, announced the **oneAPI industry initiative** at the **Intel® HPC Developer Conference**. **A lot has happened** in the oneAPI ecosystem since then.

Now **SC20** is just around the corner, and I'm sure there will be plenty of announcements. It will be a virtual conference this year, but I'm looking forward to hearing what's new with oneAPI. You can find out more at **The oneAPI Software Abstraction for Heterogeneous Computing panel discussion** with luminaries from academia and industry. There'll also be a tutorial on **C++ for Heterogeneous Programming: oneAPI (DPC++ and oneTBB)**.

In this issue of *The Parallel Universe*, we have four articles to help you make the most of oneAPI. Our feature, **Programming Data Parallel C++**, takes you through the basics of developing and offloading computational kernels to accelerators. Next is a guest editorial from Andrew Richards, CEO and founder of Codeplay, on **Building an Open, Standards-Based Ecosystem for Heterogeneous Parallelism**. Then there's **A Vendor-Neutral Path to Math Acceleration** from Mehdi Goli, principal software engineer at Codeplay. Finally, we'll see **How to Speed Up Performance by Exploring GPU Configurations** using the Intel® Advisor Offload Advisor tool, which helps you decide when it's beneficial to offload computational kernels to an accelerator.

We also have the next installment in our series on using Intel® compiler optimization reports to more effectively tune performance. **Vectorization and SIMD Optimizations** digs deeper into the compiler's auto-vectorization capability, interpreting the compiler report, and ensuring that code is being vectorized.

We close this issue with two articles on HPC clusters. The first, **Boosting Performance of HPC Cluster Workloads Using Intel® MPI Library Tuning**, builds on **Making HPC Clusters More Efficient Using Intel® MPI Library** from **our last issue**. This time, the authors demonstrate the possible performance gains using real HPC applications. The second, **Simplifying Cluster Use with OpenHPC 2.0**, describes how the OpenHPC community effort makes deploying and managing HPC clusters easier.

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

Henry A. Gabb

October 2020

Programming Data Parallel C++

A Step-by-Step Tutorial for Developing Data Parallel C++ Kernels

Jeff Hammond, Principal Engineer, Intel Data Center Group

[Editor's note: This tutorial was adapted from **Jeff's GitHub repository***.]*

This is an introduction to the **Data Parallel C++** programming model, or DPC++ for short. DPC++ is based on Khronos SYCL, which means it's a modern C++ parallel programming model. SYCL 1.2.1 is the latest Khronos standard, although the SYCL 2020 provisional specific is available for review. Intel and other members of the SYCL working group are finalizing the next version of the language specification. DPC++ contains extensions that make SYCL easier to use, although many of these are expected to be part of SYCL 2020. Implementing extensions in the DPC++ compiler helps the community evaluate their effectiveness before standardization.

Who Is This Tutorial For?

This tutorial is for programmers who already have a decent understanding of C++ and parallelism. Teaching C++ and parallelism is hard—and there's a lot of material out there already. There's far less information on SYCL itself, and even less about DPC++, so that's our focus.

One of the important C++ concepts used in SYCL is a lambda. If you don't know what a lambda is, please see **this introduction**.

SYCL is derived from OpenCL, and the execution models are quite similar. If you need help understanding the SYCL/OpenCL execution model, please check out **this overview**.

Who Is This Tutorial *Not* For?

When I tell people about SYCL, I often say, "If you like modern C++, you'll like SYCL because it's definitely modern C++." A corollary to this is that if you hate C++, you'll hate SYCL and DPC++. So, if you don't want to write modern C++, this tutorial is not for you.

OpenMP 5.0 offers many of the same features as SYCL/DPC++ but supports the ISO language triumvirate of C++, C, and Fortran. If you want to program CPUs and GPUs using Fortran, C, or pre-modern C++ (i.e., before C++11) using an open industry standard, try OpenMP.

Another alternative to SYCL/DPC++ without the C++ is OpenCL. OpenCL is a lot more verbose than SYCL, but if you're a C programmer, you likely prefer explicit control to syntactic efficiency.

The Tutorial

We'll start with vector addition, which is the "Hello, world!" of HPC and numerical computation. Printing "Hello, world!" doesn't make a lot of sense in a programming model used for doing lots of things in parallel.

Vector Addition in SYCL

The operation we're trying to implement is SAXPY, which stands for Single-precision A times X plus Y which can be implemented in C or C++ as follows:

```
for (size_t i = 0; i < length; ++i) {
    Z[i] += A * X[i] + Y[i];
}
```

There are lots of ways to write this in C++. For example, we could use ranges, which would make the code look a bit more like the upcoming SYCL version. But teaching you every possible way to write a loop in C++ isn't the point of this tutorial, and everybody understands the version that looks like C.

Here's the same loop in SYCL. There's a lot to unpack here, so we'll break down in pieces:

```
h.parallel_for<class saxpy> (sycl::range<1>{length}, [=](sycl::id<1> it) {
    const int i = it[0];
    Z[i] += A * X[i] + Y[i];
});
```

As you might have guessed, `parallel_for` is a parallel for-loop. The loop body is expressed as a lambda. The lambda is the code that looks like [..]{..}.

The loop iterator is expressed in terms of a `sycl::range` and a `sycl::id`. In our simple example, both are one-dimension, as indicated by the `<1>`. SYCL ranges and ids can be one-, two-, or three-dimensional. (OpenCL and CUDA have the same limitation.)

It may be a bit unfamiliar to write loops like this, but it's consistent with how lambdas work. However, if you've ever used **parallel STL**, **TBB**, **Kokkos**, or **RAJA**, you'll recognize the pattern.

You might be wondering about the `<class saxpy>` template argument to `parallel_for`. This is just a way to name the kernel, which is necessary because you might want to use SYCL with a different host C++ compiler than the SYCL device compiler. In this case, the two compilers need a way to agree on the kernel name. In many SYCL compilers, such as Intel DPC++, this isn't necessary. And we can tell the compiler to not worry about looking for names by using the option `-fsycl-unnamed-lambda`.

We won't try to explain what the `h` in `h.parallel_for` is right now. We'll cover that later.

SYCL Queues

One challenge of heterogeneous programming is the multiple types of processing elements and, often, different memory types. These things make compilers and runtimes more complicated. The SYCL programming model embraces heterogeneous execution, although at a much higher level than OpenCL. Not everything is explicit, either. Unlike other popular GPU programming models, SYCL kernels can be inlined into the host program flow, which improves readability.

Whenever we want to compute on a device, we need to create a work queue:

```
sycl::queue q(sycl::default_selector{});
```

The default selector favors a GPU, if present, and a CPU otherwise. We can create queues associated with specific device types using this:

```
sycl::queue q(sycl::host_selector{});   // run on the CPU without a runtime (i.e., no OpenCL)
sycl::queue q(sycl::cpu_selector{});    // run on the CPU with a runtime (e.g., OpenCL)
sycl::queue q(sycl::gpu_selector{});           // run on the GPU
sycl::queue q(sycl::accelerator_selector{});   // run on an FPGA or other accelerator
```

The host and CPU selectors may lead to significantly different results, even though they target the same hardware, because the host selector might use a sequential implementation optimized for debugging, while the CPU selector uses the OpenCL runtime and runs across all the cores. Also, the OpenCL just-in-time (JIT) compiler might generate different code because it's using a different compiler altogether. Don't assume that just because the host is a CPU, that host and CPU mean the same thing in SYCL.

Managing Data in SYCL Using Buffers

The canonical way to manage data in SYCL is with buffers. A SYCL buffer is an opaque container. This is an elegant design, but some applications would like pointers, which are provided by the USM extension, discussed later.

```
// T is a data type, e.g., float
std::vector<T> h_X(length,xval);
sycl::buffer<T,1> d_X { h_X.data(), sycl::range<1>(h_X.size()) };
```

In the previous example, the user allocates a C++ container on the host and then hands it over to SYCL. Until the destructor of the SYCL buffer is invoked, the user can't access the data through a non-SYCL mechanism. SYCL accessors are the important aspect of SYCL data management with buffers, which we'll explain below.

Controlling Device Execution

Because device code may require a different compiler or code generation mechanism from the host, it's necessary to clearly identify sections of device code. Below we see how this looks in SYCL 1.2.1. We use the `submit` method to enqueue work to the device queue, `q`. This method returns an opaque handler against which we execute kernels, in this case via `parallel_for`.

```
q.submit([&](sycl::handler& h) {
    ...
    h.parallel_for<class nstream> (sycl::range<1>{length}, [=] (sycl::id<1> i) {
        ...
    });
});
q.wait();
```

We can synchronize device execution using the `wait()` method. There are finer-grain methods for synchronizing device execution, but we start with simplest one, which is a heavy hammer.

Some users may find the above code a bit verbose, particularly compared to models like Kokkos. The Intel DPC++ compiler supports a terse syntax, which we'll cover below.

Compute Kernels and Buffers

SYCL accessors are the final piece in our first SYCL program. Accessors may be unfamiliar to GPU programmers, but they have a number of nice properties compared to other methods. While SYCL allows the programmer to move data explicitly using, for example, the `copy()` method, the accessor methods don't require this because they generate a dataflow graph that the compiler and runtime can use to move data at the right time. This is particularly effective when multiple kernels are invoked in sequence. In this case, the SYCL implementation will deduce that data is reused and not copy it back to the host unnecessarily. Also, we can schedule data movement asynchronously (i.e., overlapped with device execution). While expert GPU programmers can do this manually, we often find that SYCL accessors lead to better performance than OpenCL programs where programmers must move data explicitly.

```
q.submit([&](sycl::handler& h) {

    auto X = d_X.template get_access<sycl::access::mode::read>(h);
    auto Y = d_Y.template get_access<sycl::access::mode::read>(h);
    auto Z = d_Z.template get_access<sycl::access::mode::read_write>(h);

    h.parallel_for<class nstream> (sycl::range<1>{length}, [=] (sycl::id<1> i) {
        ...
    });
});
```

Because programming models that assume pointers are handles to memory have a hard time with SYCL accessors, the USM extension makes accessors unnecessary. USM places a greater burden on the programmer in terms of data movement and synchronization but helps with compatibility in legacy code that wants to use pointers.

Review of Our First SYCL Program

Here are all the components of our SYCL SAXPY program we just described:

```
std::vector<float> h_X(length,xval);
   std::vector<float> h_Y(length,yval);
   std::vector<float> h_Z(length,zval);

   try {

       sycl::queue q(sycl::default_selector{});

       const float A(aval);

       sycl::buffer<float,1> d_X { h_X.data(), sycl::range<1>(h_X.size()) };
       sycl::buffer<float,1> d_Y { h_Y.data(), sycl::range<1>(h_Y.size()) };
       sycl::buffer<float,1> d_Z { h_Z.data(), sycl::range<1>(h_Z.size()) };

       q.submit([&](sycl::handler& h) {

           auto X = d_X.template get_access<sycl::access::mode::read>(h);
           auto Y = d_Y.template get_access<sycl::access::mode::read>(h);
           auto Z = d_Z.template get_access<sycl::access::mode::read_write>(h);

           h.parallel_for<class nstream>( sycl::range<1>{length}, [=] (sycl::id<1> it) {
               const int i = it[0];
               Z[i] += A * X[i] + Y[i];
           });
         });
         q.wait();
   }
   catch (sycl::exception & e) {
       std::cout << e.what() << std::endl;
       return 1;
   }
```

The full source code for this example is available in the GitHub repository at **https://github.com/jeffhammond/dpcpp-tutorial**.

SYCL 2020 Unified Shared Memory (USM)

While the program above is perfectly functional and can be implemented across a wide range of platforms, some users will find it rather verbose. Furthermore, it's not compatible with libraries and frameworks that need to manage memory using pointers. To address this issue with SYCL 1.2.1, Intel developed an extension in DPC++ called Unified Shared Memory (USM) that supports pointer-based memory management.

USM supports two important usage models, both of which will be illustrated below. The first one supports automatic data movement between the host and device. The second one is for explicit data movement to and from device allocations.

The details are in the SYCL 2020 provisional specification, but to get started, all you need to know is below. The q argument is the queue associated with the device where the allocated data will live (either permanently or temporarily):

```
//shared allocation (can migrate between host and device)
auto d_X = sycl::malloc_shared<float>(length, q);

//device allocation  (does not migrate)
auto d_X = sycl::malloc_device<float>(length, q);

// deallocation (works with any allocation type)
sycl::free(d_X, q);
```

If we're using device allocation, data must be moved explicitly (e.g., using the SYCL `memcpy` method), which behaves the same way `std::memcpy` does (e.g., the destination is on the left):

```
const size_t bytes = length * sizeof(float);

// d_Z <- h_Z
q.memcpy(d_Z, h_Z.data(), bytes);
q.wait();

// h_Z <- d_Z
q.memcpy(h_Z.data(), d_Z, bytes);
q. wait();
```

If we use USM, accessors are no longer required, which means we can simplify the kernel code above to:

```
q.submit([&](sycl::handler& h) {
      h.parallel_for<class saxpy>(sycl::range<1>{length}, ::id<1> i)
            d_Z[i] += A * d_X[i] + d_Y[i];
      });
});
```

You can find the complete working examples of both versions of USM in this repo, named `saxpy-usm.cc` and `saxpy-usm2.cc`, respectively.

SYCL 2020 Terse Syntax

Finally, in case you've been wondering why the opaque handler `h` was required in each of these programs, it turns out that it isn't required after all. The following is an equivalent implementation, which was added in the SYCL 2020 provisional specification. Furthermore, we can take advantage of lambda names being optional in the SYCL 2020 provisional specification. Together, these two small changes make SYCL kernels the same length as the original C++ loop listed at the beginning of this tutorial:

```
q.parallel_for(sycl::range<1>{length}, [=](sycl::id<1> i) {
      d_Z[i] += A * d_X[i] + d_Y[i];
});
```

We started with three lines of code that run sequentially on a CPU and end with three lines of code that run in parallel on CPUs, GPUs, FPGAs, and other devices. Obviously, not everything will be as simple as SAXPY, but at least now you know that SYCL isn't going to make easy things hard, and it builds on a number of modern C++ features and universal concepts like "parallel for" rather than introducing new things to learn.

More Resources

- **Khronos SYCL 1.2.1 specification**
- **oneAPI DPC++ documentation**
- **SYCL/DPC++ compiler on GitHub**
- **DPC++ language extensions**

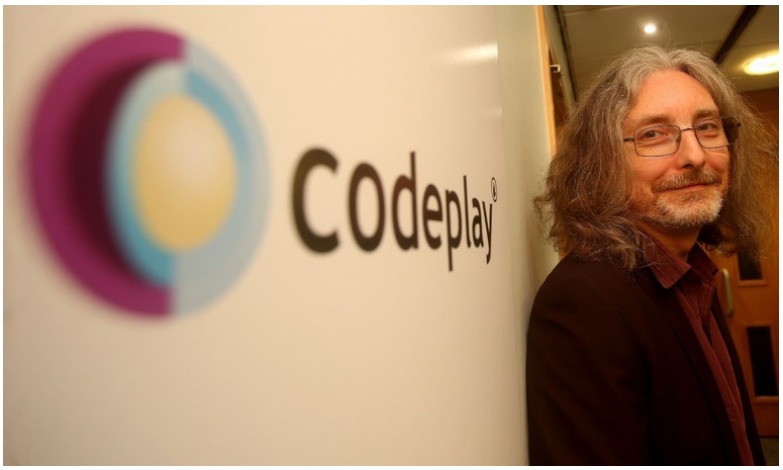
Building an Open, Standards-Based Ecosystem for Heterogeneous Parallelism

Guest Editorial

Andrew Richards, CEO and Founder, Codeplay

Codeplay, based in Edinburgh, Scotland, started developing tools for graphics processors more than a decade ago. We always thought developers would do more than just graphics processing with GPUs because they're so incredibly powerful. But we're now seeing demand in a wide range of fields—from self-driving cars to the latest exascale supercomputers.

What's become clear is that the only way GPU programming will work is if we standardize it. That's the only way to get software developers and hardware companies working together. And the only way to create a vision where developers can write C++ and actually make it run very fast on lots of different platforms is by having standards.



Andrew Richards, CEO and founder of Codeplay

Codeplay led the definition of the **SYCL** programming model, an open standard designed to bring performance and portability to software developers writing parallel applications. We've always believed that open standards are the way forward to enable a broad community of developers. In fact, we've spent many years successfully working with processor vendors to enable their processors for software developers using open standards.

SYCL is designed as a C++ programming model for accelerators (e.g., GPUs, FPGAs, specialist processors like DSPs). If you want to productively develop software that gives great performance on one of these processors, you need a well-defined programming model. You also need highly-optimized libraries that do specific things. So, if you want to do matrix multiplication, for example, SYCL allows you to write your own matrix multiply operation.

What it doesn't do is work out exactly the right algorithms to run on each type of processor.

This is where **oneAPI** helps. oneAPI is an industry initiative that encourages collaboration on open specifications and compatible implementations across the ecosystem. oneAPI actually delivers matrix multiplication, convolutions, and all sorts of standard, high-performance software libraries built on top of these open, standard programming models. The key thing is that it all fits together, and I think that's what will make it successful.

It's been great to see Intel's adoption of the SYCL standard. The **DPC++ compiler**, which is part of oneAPI, implements SYCL, and Intel is contributing greatly to SYCL's evolution.

Developers are increasingly looking for solutions that offer openness rather than proprietary programming interfaces. And what's really great about oneAPI is that it integrates with other open standards and open source frameworks.

The learnings from the oneAPI initiative are already feeding back into other standards bodies, like the SYCL and C++ language standards. oneAPI also uses the **SPIR-V standard**, which is another open standard from Khronos that defines the intermediate format for compiler tools enabling such exciting technologies as AI graph compilers, for example.

I know the vision of oneAPI says it does everything for everyone, making it possible to run every bit of software everywhere. But it's also very pragmatic. oneAPI understands that it's part of a wider ecosystem. I think that's really important for those of us actually building real technology.

You can find out more about SYCL at **https://sycl.tech** and see the oneAPI specifications at **https://www.oneapi.com**.

NEWS HIGHLIGHTS

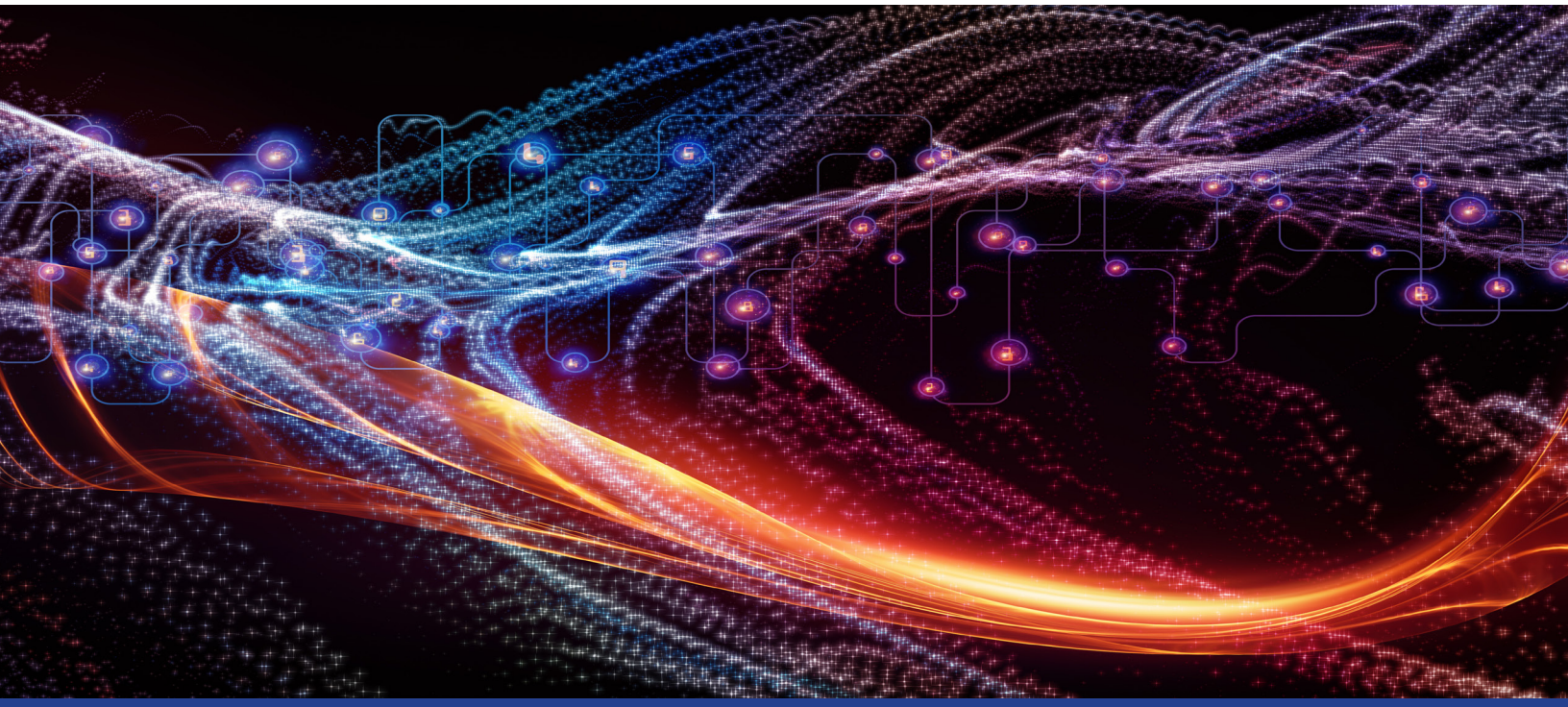
oneAPI Academic Center of Excellence established at the Heidelberg University Computing Center (URZ)

A oneAPI Academic Center of Excellence (CoE) is now established at the Heidelberg University Computing Center (URZ). The new CoE will conduct research supporting the oneAPI industry initiative to create a uniform, open programming model for heterogeneous computer architectures.

Read on >

Deliver uncompromised performance for diverse workloads across multiple architectures with oneAPI.

Learn How >



A Vendor-Neutral Path to Math Acceleration

Bringing Standardized APIs Across Multiple Math Domains Using oneAPI Math Kernel Library

Mehdi Goli, Principal Software Engineer, Codeplay Software, and Maria Kraynyuk, Software Engineer, Intel Corporation

The Basic Linear Algebra Subprograms (BLAS) provide key functionality across CPUs, GPUs, and other accelerators for high-performance computing and artificial intelligence. Historically, developers needed to write code for each hardware platform. And there was no easy way to port source code from one accelerator to another. For the first time, the **oneAPI Math Kernel Library (oneMKL)** open source interface project bridges the gap to support x86 CPU, Intel® GPU, and NVIDIA GPU linear algebra

functionality by creating a single open source interface that can use highly optimized third-party libraries for different accelerators underneath. This project also opens BLAS functions to other industry accelerators. Learn how the BLAS interface implementation is just the first step to supporting math functionality of other oneMKL domains beyond BLAS.

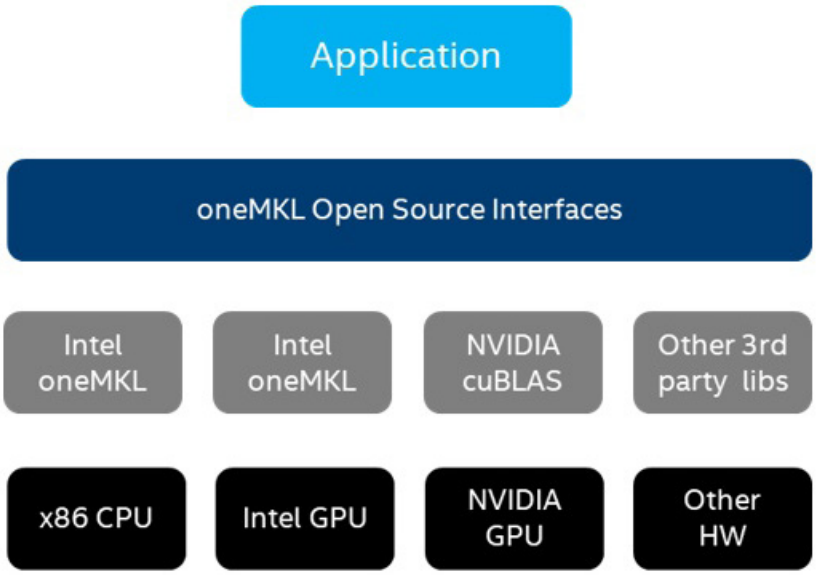
oneMKL Open Source Interface

Libraries play a crucial role in software development:

- **First, libraries provide functionality that can be reused across applications** to reduce both development efforts and the possibility of errors.
- **Second, some libraries are highly optimized for target hardware platforms**, allowing developers to achieve the best application performance with less effort.

As the industry has developed, the selection of hardware has become more diverse, with the rise of accelerators including GPUs, FPGAs, and specialized hardware. Most vendor libraries only function on specific hardware platforms. These libraries are sometimes implemented in proprietary languages that aren't portable to other architectures. For large applications, this means developers are locked into a specific hardware architecture—and can't benefit from different hardware solutions without a major new development effort.

The oneMKL open source interface project aims to solve this cross-architecture issue by taking advantage of the **Data Parallel C++ (DPC++) language**. It addresses standardization by enabling implementation of the vendor-agnostic DPC++ API defined in the **oneMKL specification** and addresses portability by using existing, well-known, hardware-specific libraries underneath (**Figure 1**).



1 **High-level overview of oneMKL open source interfaces**

Usage Model and Third-Party Library Selection

The oneMKL open source interfaces project provides a DPC++ API common for all hardware devices. For example, it runs the GEMM (general matrix multiplication) function on both Intel and NVIDIA GPUs identically, except for the SYCL queue creation step. In the code snapshot in **Figure 2**, we create two different queues targeted for CPU and GPU devices and call GEMM for each queue to execute the function on the targeted devices. Compilation is simple (**Figure 3**).

```
#include "oneapi/mkl.hpp"
...
sycl::queue cpu_queue(sycl::cpu_selector{});
sycl::queue gpu_queue(sycl::gpu_selector{});
oneapi::mkl::blas::gemm(cpu_queue, transA, transB, m, ...);
oneapi::mkl::blas::gemm(gpu_queue, transA, transB, m, ...);
```

2 **Code snapshot of application `app.cpp`**

```
$> dpcpp app.cpp -lonemkl
```

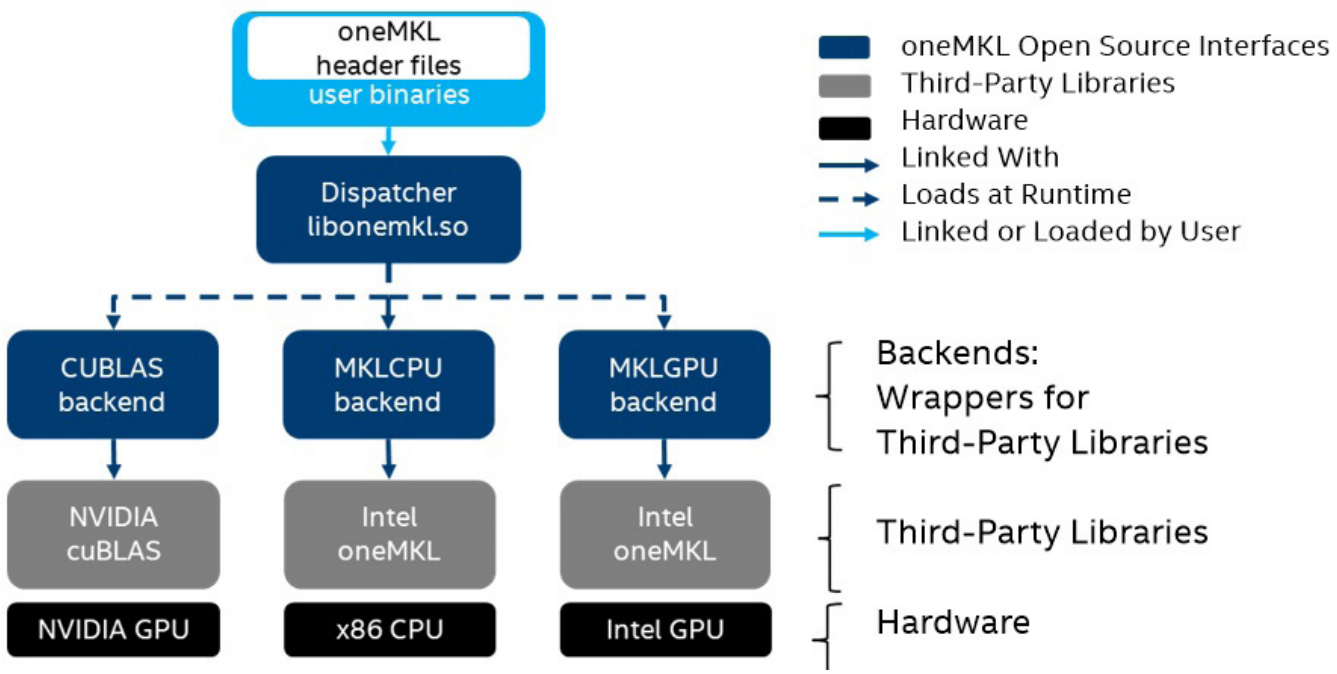
3 **Compile/link line for application `app.cpp`**

At the application execution time, oneMKL:

- **Selects the third-party library** based on the device information provided by the SYCL queue
- **Converts input parameters** to the proper third-party library input data
- **Calls functions from the third-party library** to execute on the targeted device

Figure 4 shows the execution flow, where oneMKL is divided into two main parts:

- **Dispatcher library:** Identifies the targeted device and loads the appropriate backend library for that device.
- **Backend library:** Wrappers that convert DPC++ run-time objects from the application to third-party library-specific objects, then calls the third-party function. There is a backend library for each supported third-party library.



4 **Execution flow**

For the given example, if the targeted device is an Intel CPU, oneMKL will use the Intel oneMKL library underneath. If the targeted device is an NVIDIA GPU, oneMKL will use NVIDIA cuBLAS. This approach allows a oneMKL project to achieve the level of performance third-party libraries provide with minimal overhead for object conversion.

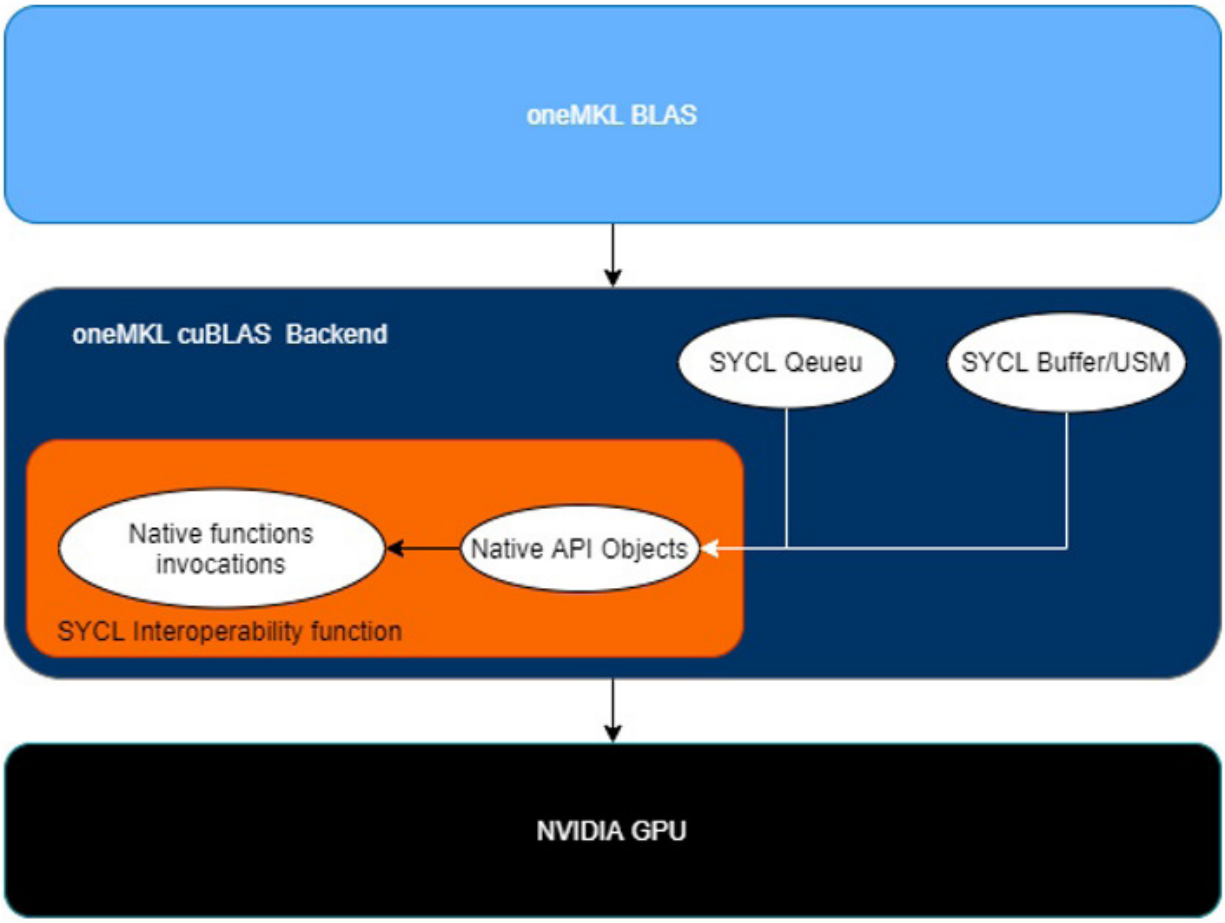
Third-Party Library Enabling: cuBLAS

Integrating CUDA on the oneMKL SYCL backend relies on the existing SYCL CUDA support integrated with the DPC++ CUDA backend, using the SYCL 1.2.1 interface with some additional extensions to expose the CUDA interoperability objects. The SYCL interoperability with existing native objects is the `interop_task` interface inside the command group scope. `interop_task` is an extension to the SYCL programming model contributed by Codeplay for interfacing with the native CUDA libraries (in this case, cuBLAS). A SYCL application using interoperability with CUDA functions uses the same SYCL programming model. It's composed of three scopes to control the construction and lifetime of different objects used in the applications:

- **The kernel scope** represents a single kernel function interfacing with native objects executing on a device.
- **The command group scope** determines a unit of work comprised of a kernel function and accessors.
- **The application scope** includes all other code outside of a command group scope.

Once invoked, `interop_task` is injected into the SYCL runtime DAG to preserve data dependencies among kernels on the selected device.

The cuBLAS library contains BLAS routines integrated with an explicit scheduling control object, called a handle. The handle enables users to assign different routines to different devices or streams with different or the same contexts.



5 Architectural view of cuBLAS integration in oneMKL

Figure 5 shows the architectural view of mapping the cuBLAS backend to oneMKL. Mapping cuBLAS to oneMKL has three parts:

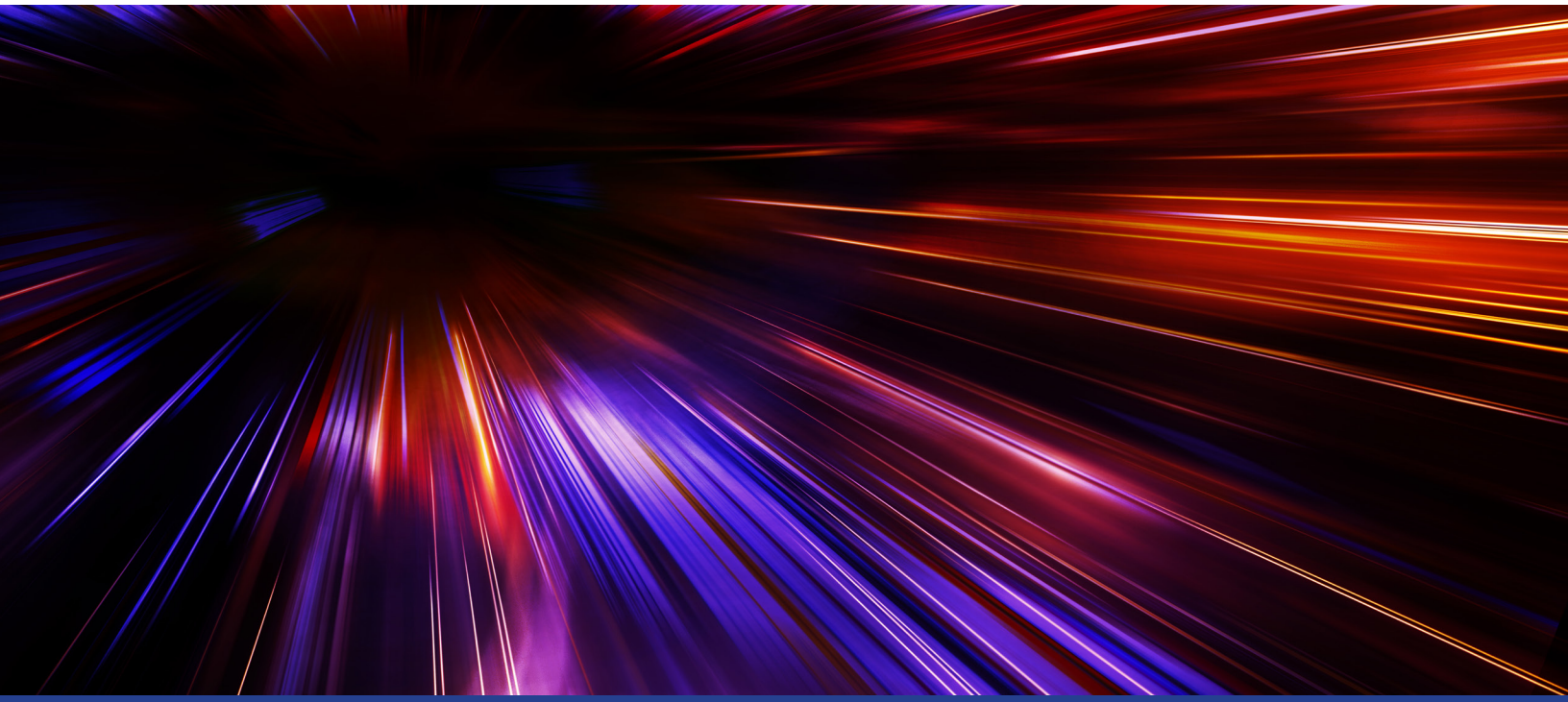
- **Constructing the cuBLAS handle.** Creating a cuBLAS handle is costly. You should create one handle per thread per CUDA context to avoid performance overhead and to maintain correctness. Since there's one-to-one mapping between the SYCL context the and underlying CUDA context, one cuBLAS handle is automatically created by a user-created SYCL context per thread to minimize the performance overhead and to guarantee correctness. Because you can create multiple SYCL queues for a single SYCL context, you can submit different BLAS routines to different streams with the same context using the same cuBLAS handle.

- **Extracting CUDA runtime API objects from the SYCL runtime API.** The SYCL queue embeds the SYCL context used to create the queue. Using the SYCL interoperability features, the underlying native CUDA context and stream are extracted from the user-created SYCL queue to create and retrieve the cuBLAS handle required to invoke cuBLAS routines. Also, the allocated CUDA memories for a BLAS routine are obtained from accessors, created from SYCL buffers, or the SYCL Unified Shared Memory (USM) model. Interoperability with CUDA runtime API objects is only accessible from `host_task/interop_task` inside the command group scope.
- **Interfacing with cuBLAS routines inside SYCL interoperability functions.** For a BLAS routine requested by a user from the oneMKL BLAS interface, an equivalent cuBLAS routine is invoked from the `host_task/interop_task` function using the retrieved cuBLAS handle and allocated memories. During the execution of a cuBLAS routine, the CUDA context used for creating a cuBLAS handle must be the active CUDA runtime context. Therefore, at the beginning of each routine invocation, the CUDA context associated with the selected cuBLAS handle is activated.

Results and Next Steps

The oneMKL open source interface project is **available on GitHub** and open for contributions from anyone. Currently, it includes all levels of BLAS and BLAS-like extensions. Intel developers built initial for x86 CPUs and Intel GPUs using Intel® oneMKL. From the open source community, Codeplay contributed NVIDIA GPU support via NVIDIA cuBLAS.

While API standardization exists for math domains like linear algebra, and de facto standards like **FFTW** exist for discrete Fourier transforms, standard APIs are harder to come by in other domains like sparse solvers and vector statistics. The oneMKL project's goal is to bring standardized APIs to multiple math domains. For example, the Intel team plans to introduce an API for **Random Number Generator (RNG)**. The oneMKL project is a place for community-driven standardization of math APIs.



How to Speed Up Performance by Exploring GPU Configurations

Performance Analysis with Intel® Advisor Offload Advisor

Kevin O'Leary, Lead Technical Consulting Engineer, and Md Khaledur Rahman, Graduate Technical Intern, Intel Corporation

Intel® oneAPI Toolkits provide a unified, standards-based programming model for delivering uncompromised performance for diverse workloads across multiple architectures. One recent addition is the **Intel® Advisor** Offload Advisor feature, which you can use for interactive performance modeling. In this study, we'll explore Offload Advisor and see how it helps you see future performance headroom, or what hardware parameters are the most sensitive for a given application. This type of what-if study

lets you explore how your application will perform if some computations are offloaded to different GPUs. To conduct the analysis, we'll use the popular, high-performance **Rodinia** computational fluid dynamics (CFD) application.

Background

Heterogeneous parallelism was originally implemented in CUDA for CFD. Basically, it computes a three-dimensional Euler's equation for fluid dynamics. This application has intensive computation that makes it a compute-bound problem.

For our experiment, we migrated the CUDA application to **Data Parallel C++ (DPC++)** using the oneAPI compatibility tool. (To learn more, see **Heterogeneous Programming Using oneAPI** from **issue 39** of *The Parallel Universe*.)

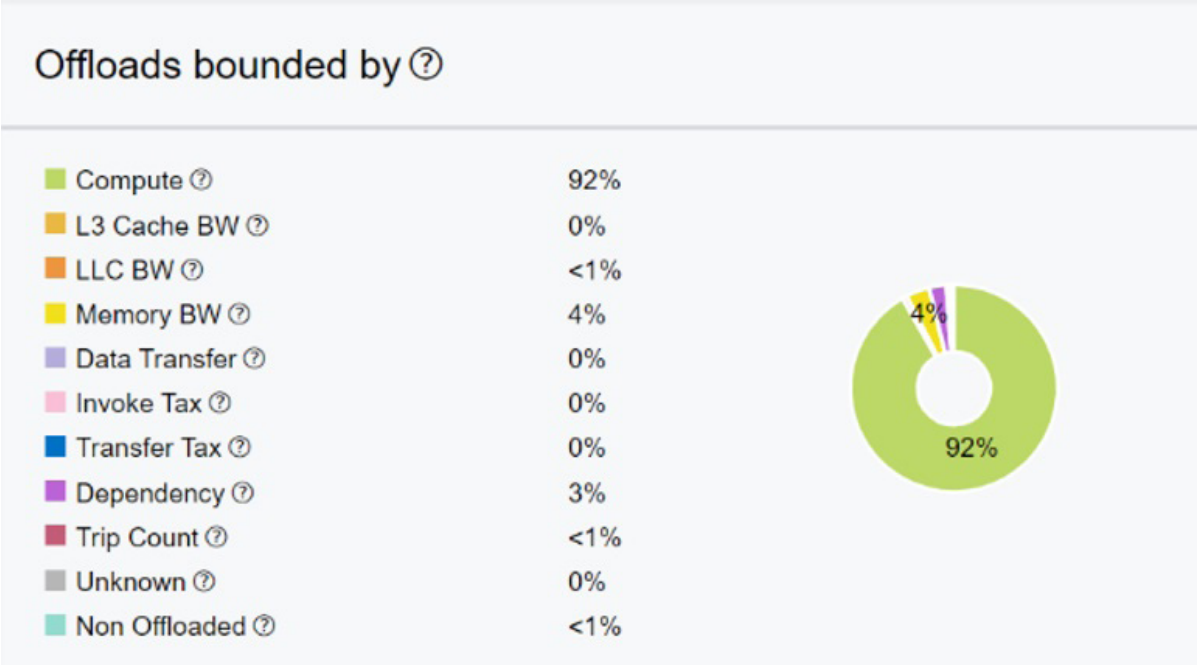
Methodology

In general, we used the following methodology:

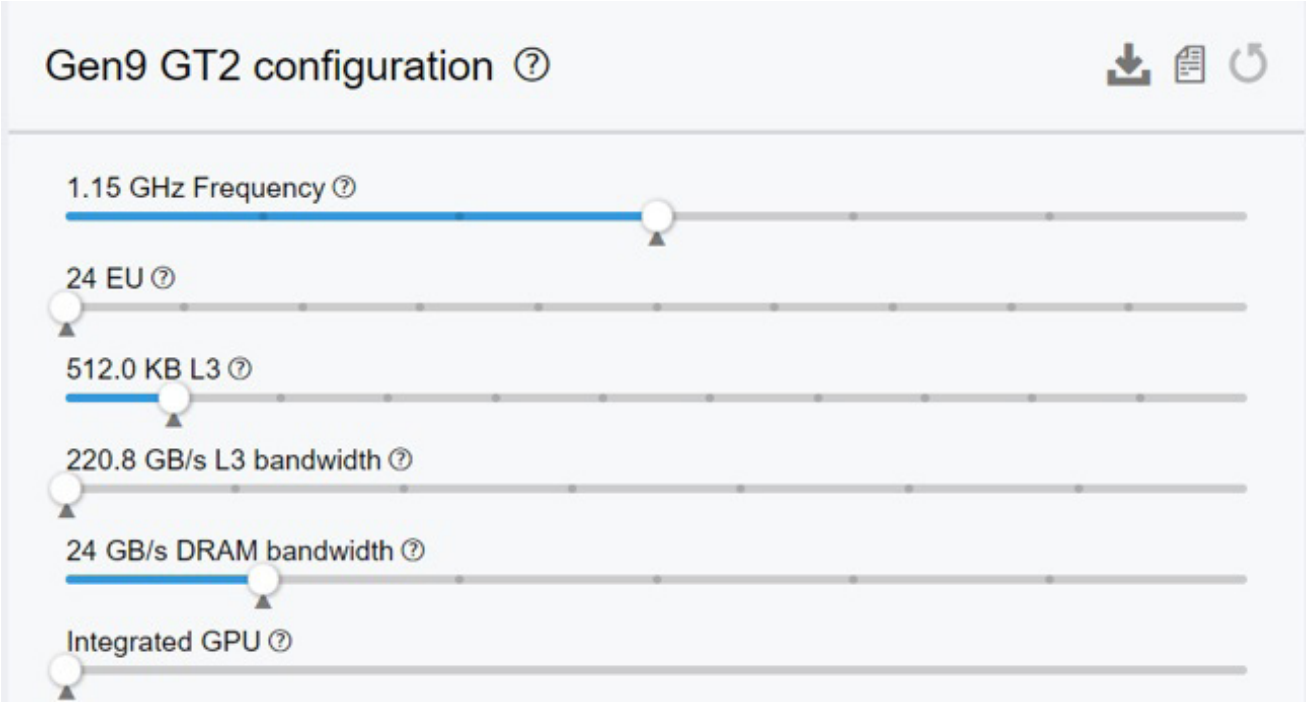
1. **Run an analysis** using Offload Advisor.
2. **From the report,** see the main bottlenecks for the application. For example, is the workload bound by compute, memory, or some other factor?
3. **Based on our bottleneck**, do a what-if analysis with various GPU configurations to see if this removes the bottleneck. For example, if the workload is compute-bound, try increasing the number of execution units.

The Rodinia application has high computational intensity and the offloaded regions are mostly compute-bound. When we examine the Offload Advisor report for the baseline version, we see that 92% of the offloads are compute-bound (**Figure 1**). This means increasing computational units may help the application run faster. We can also see that the invocation tax is 0%, which gives in an optimistic estimation for the offload regions. Other parameters are negligible for the current settings.

From the offload bounding parameters, we get an idea of which configuration we need to change to boost performance on the target. For example, if offloads are bounded by compute, we may want to increase execution units (EUs). On the other hand, if the offloads are bounded by memory bandwidth, we may want to increase DRAM bandwidth for the target to estimate speedup for GPU acceleration. Offload Advisor offers an interactive interface to tune the estimated performance on the target device (**Figure 2**). Notice there are several options to consider. For example, we can increase or decrease the EUs by moving the slider left or right. We can do the same for other parameters and then download the configuration by clicking on the icon, which we can use for future analysis.



1 **Breakdown of offload bounding parameters**



2 **Estimate performance by changing the configuration of the target device**

Results

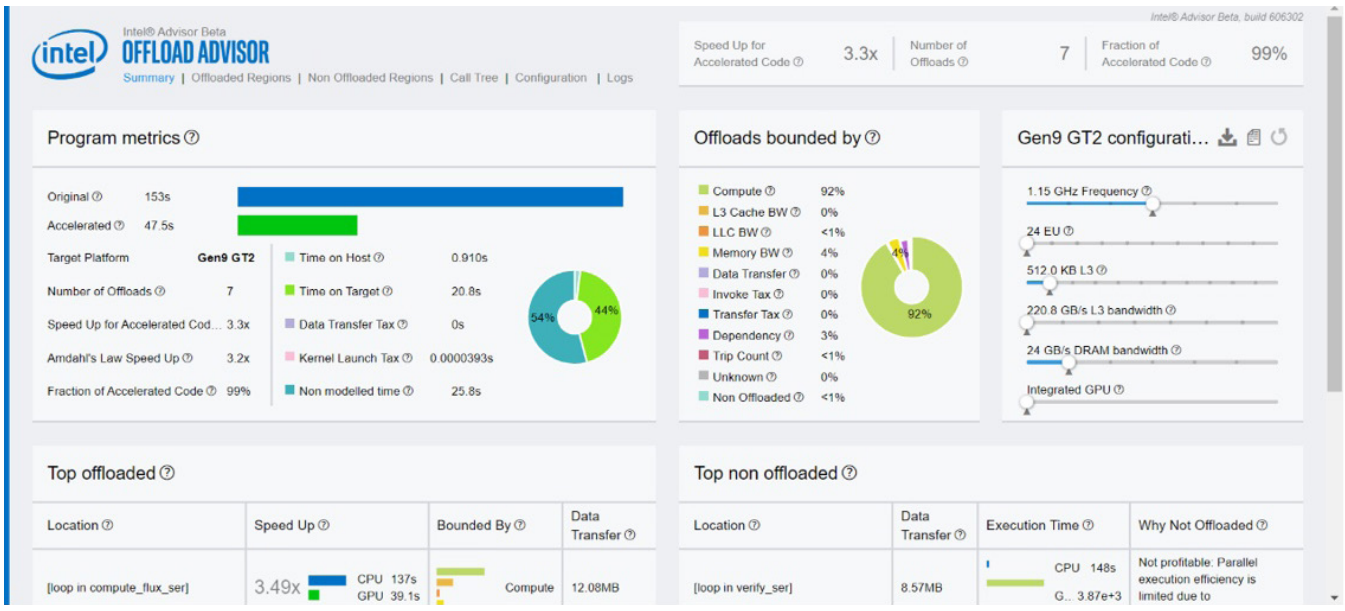
For all of our experiments, we use the **fvcorr.domn.097K** dataset. We disable the invocation tax in our experiments using the option `–jit`. We run the baseline version, collect results, and analyze using Offload Advisor. We collected results as follows:

```
# advixe-python $APM/collect.py -jit -config gen9 ./proj_dir -- ./a.out ../data/ fvcorr.domn.097K
```

To analyze the results, we use this command:

```
# advixe-python $APM/analyze.py -jit -config gen9 ./proj_dir
```

The report file is written to the `perf_models/m0000` folder. Just click on the `report.html` file and see the results in your Web browser. Every time we rerun the analysis, it creates a folder with prefix `m****` inside the `perf_models` folder. There are several windows in the report file, but we'll focus on the results summary (**Figure 3**).



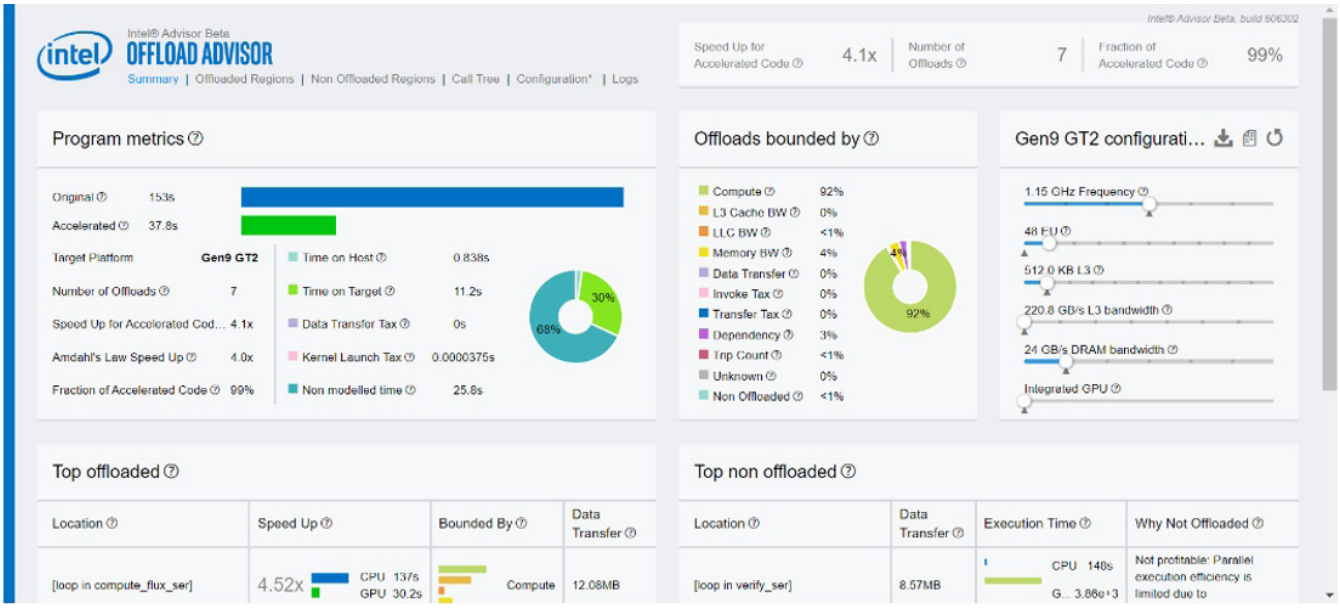
3 **Summary of Intel Advisor Offload Advisor**

We observe in **Figure 3** that the baseline version spends 44% of the time on the target device and achieves around a 3.3x speedup for the accelerated code. We also see that the offloads are mostly compute-bound.

Increasing EUs may also improve speedup. We can use sliders on the configuration window to adjust the value based on our expectation. To perform this experiment, we create and download a configuration file, named `scalers.toml`, multiplying the EU's counter by two. We generate this from the `report.html` file by changing options on the right side under the `Gen9 GT2` configuration and clicking to download. Make sure that there is a line `EU_count_multiplier=2` in the `toml` file. All other options remain the same. Then, we rerun the analysis:

```
#  advixe-python $APM/analyze.py -jit -config gen9 -config scalers.toml ./proj_dir
```

We observe that the EUs are now 48 and the speedup also increases for the accelerated code. The offload regions remain compute-bound (**Figure 4**).

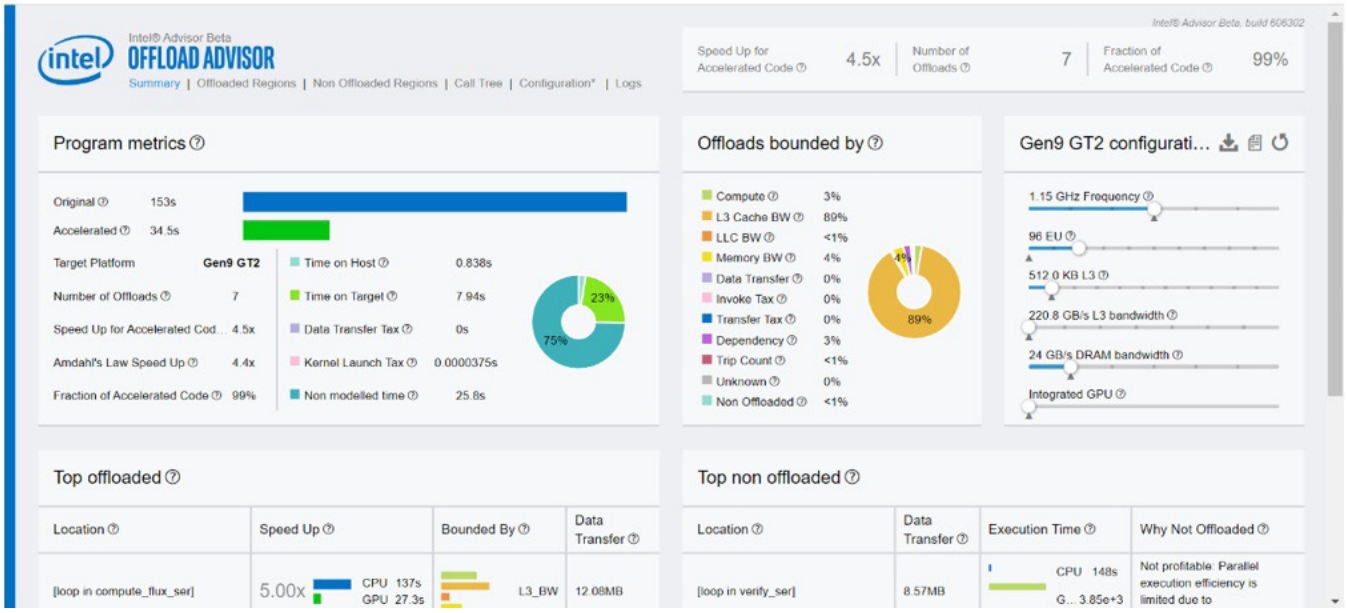


4 **Summary of Offload Advisor for 2x execution units with respect to baseline**

Since the offloaded regions are compute-bound, we further increase the EUs by setting `EU_count_multiplier=4` in the `toml` file. It uses 96 EUs. We rerun the analysis as follows:

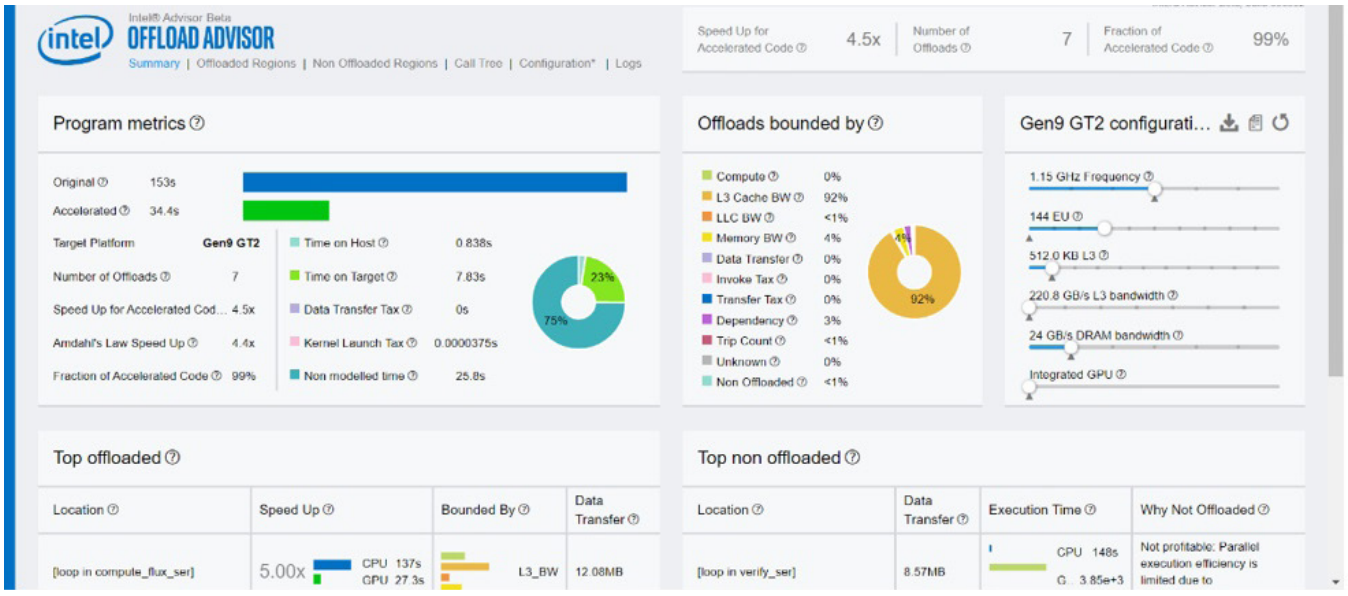
```
# advixe-python $APM/analyze.py -jit -config gen9 -config scalers.toml ./proj_dir
```

We observe that the speedup increases. However, the offloaded regions are now L3 bandwidth-bound (**Figure 5**).



5 **Summary of Offload Advisor for 4x more execution units than baseline**

We want to increase the speedup by setting `EU_count_multiplier=6`. Because the offloaded regions are now L3 bandwidth-bound, increasing the number of EUs doesn't help (**Figure 6**). The speedup of accelerated code remains the same. We rerun the analysis, which creates a folder named `m0003` in the `perf_models` folder.

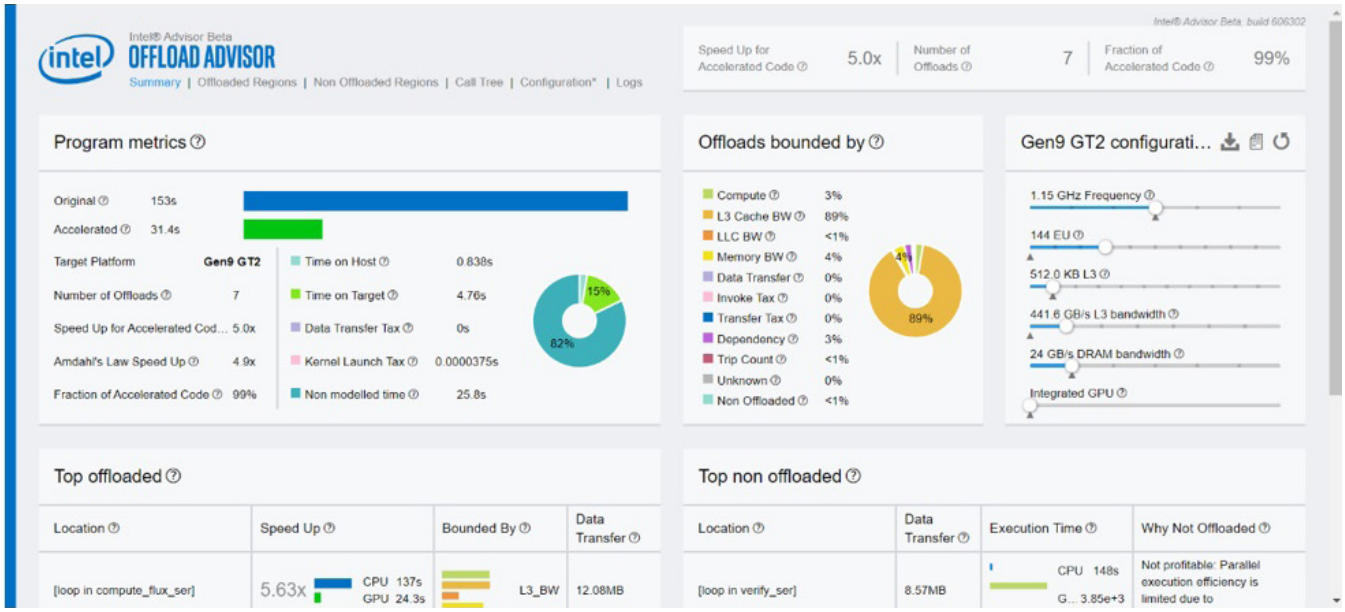


6 **Summary of Offload Advisor for 6x more execution units than baseline**

Next, we focus on increasing L3 bandwidth in the `scalers.toml` file. We can do this by setting `L3_bandwidth_multiplier=2`. We keep the other parameters the same and rerun the analysis:

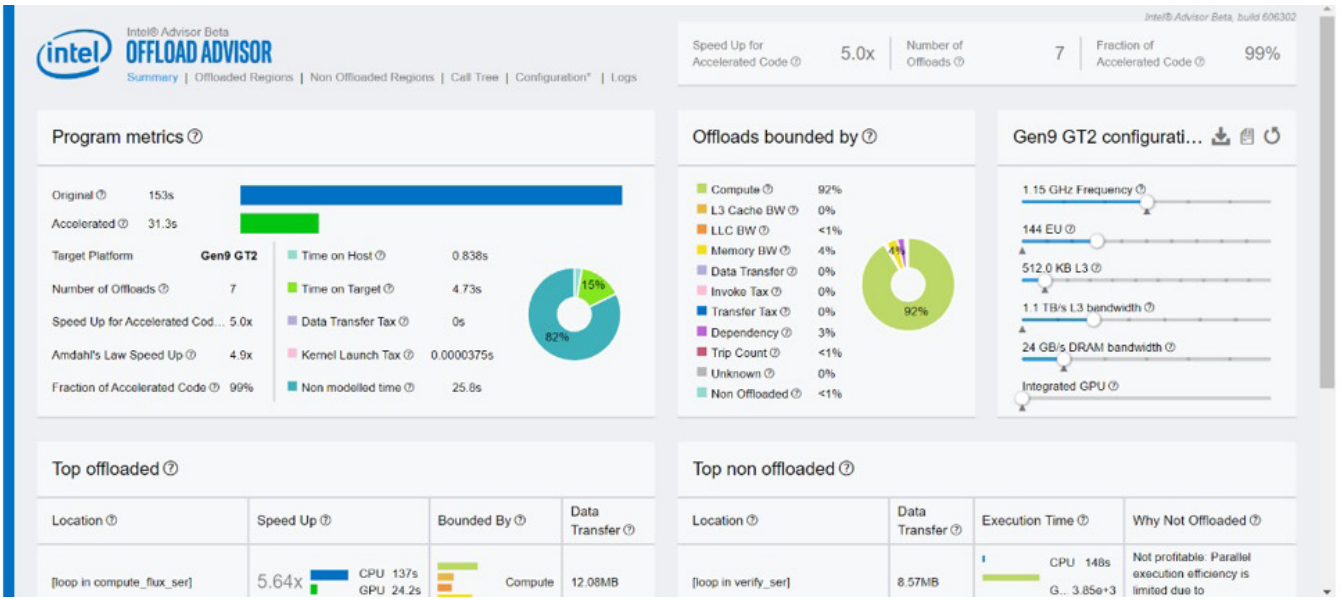
```
# advixe-python $APM/analyze.py -jit -config gen9 -config scalers.toml ./proj_dir
```

In the `reports.html` file, we can see that L3 bandwidth jumps from 220.8 GB/s to 441.6 GB/s, and speedup also increases. However, the offloaded regions remain L3 bandwidth-bound (**Figure 7**).



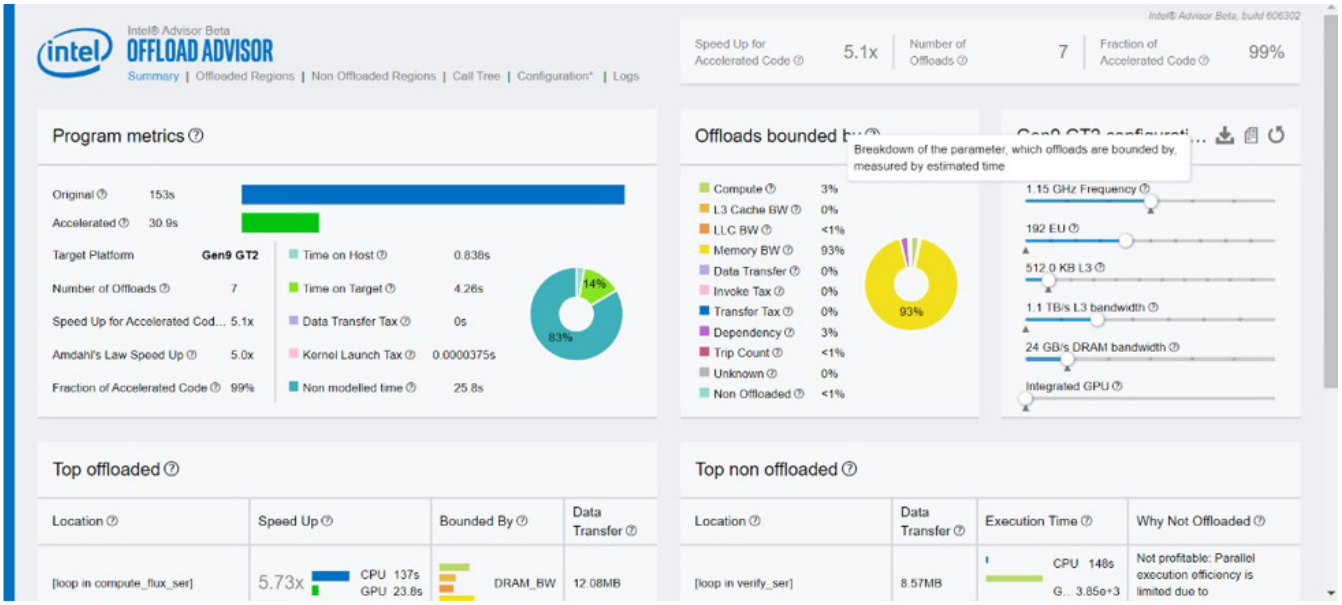
7 **Summary of Offload Advisor for 6x more execution units and 2x higher L3 bandwidth**

Since the offloaded regions are still L3 bandwidth-bound, we increase L3 bandwidth to see the effect and rerun the analysis phase as before. This didn't change the speedup. Interestingly, it makes the offloaded regions compute-bound again. This is shown in **Figure 8**, where we see that the L3 bandwidth is now 1.1 TB/s but there is no change in the speedup.



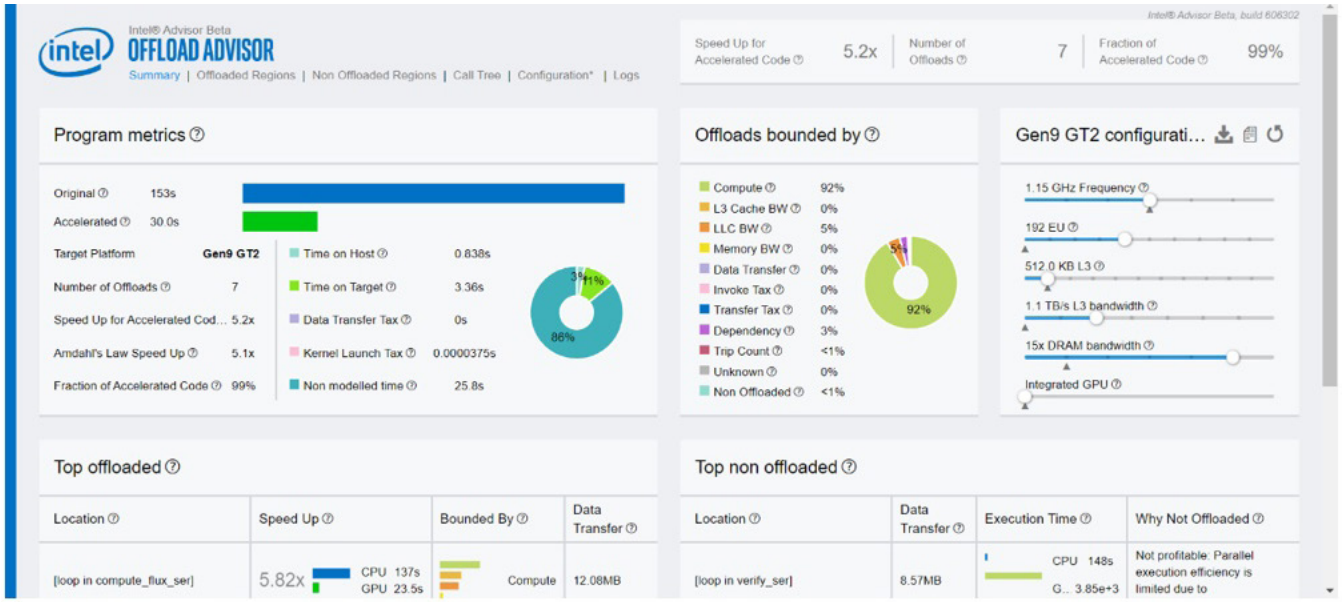
8 **Summary of Offload Advisor for 6x more execution units and 5x higher L3 bandwidth**

Again, we increase EUs by setting `EU_count_multiplier=8` and rerun the experiment. This shows a speedup, but the offloaded regions are now mostly memory-bound (**Figure 9**).



9 **Summary of Offload Advisor for 8x execution units and 5x L3 bandwidth**

Finally, we increase the memory bandwidth by setting `Memory_bandwidth=360000000000` in the `toml` file and rerun the experiment keeping other parameters the same. In **Figure 10**, we can see this setting increases the speedup of accelerated code a little. Notably, the offloaded regions are now compute-bound again.



10 **Summary of Offload Advisor for 8x execution units, 5x L3 bandwidth, and 15x memory bandwidth**

Boosting Performance

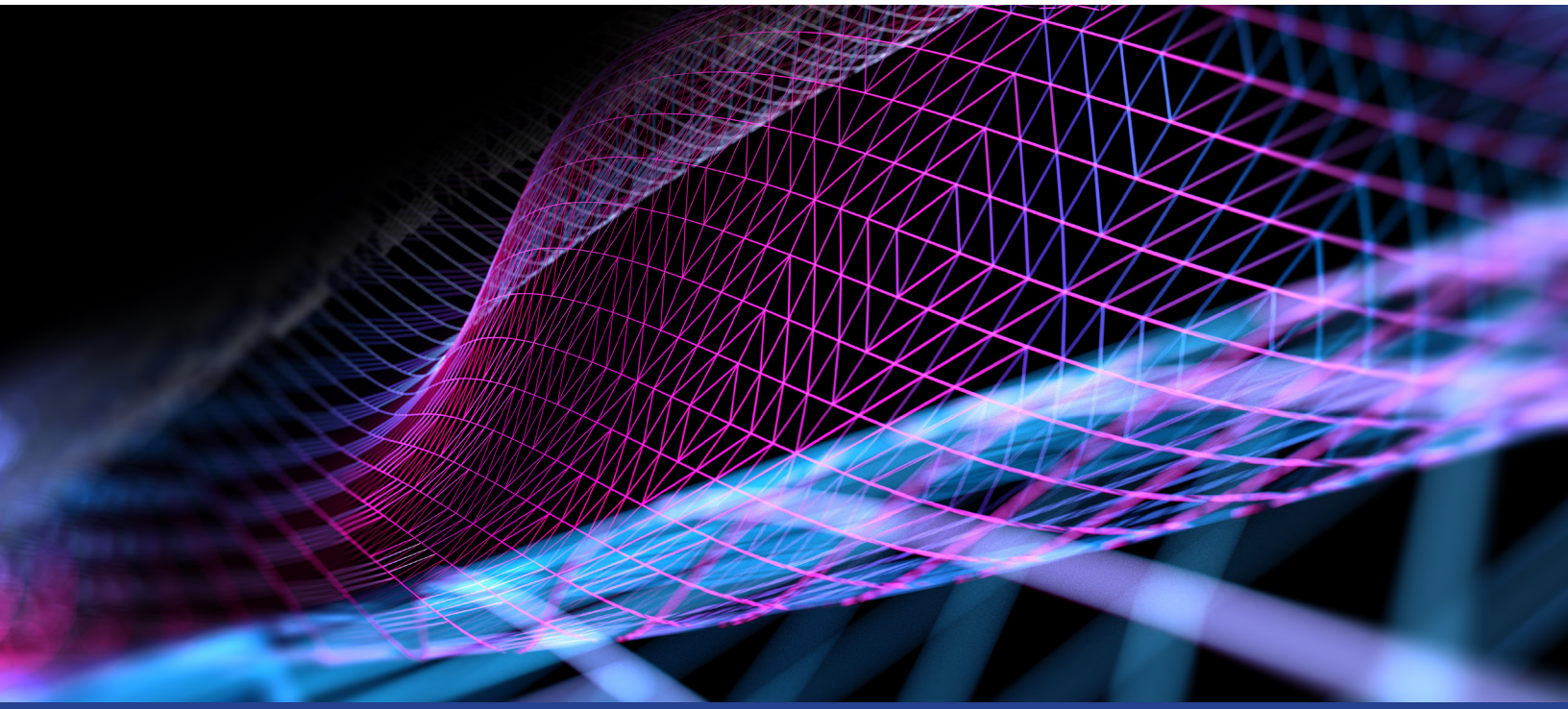
Intel Advisor Offload Advisor can help you speed up your application's performance on target devices, both present and future, with lots of options for exploring different GPU configurations.

VIDEO HIGHLIGHTS

Tuning Applications for Multiple Architectures

Henry Gabb and Senior Principal Engineer Paul Petersen discuss how oneAPI delivers specific benefits for developing and tuning applications to run across multiple architecture types with optimal performance.

Watch >



Vectorization and SIMD Optimizations

Making the Most of the Intel® Compiler Optimization Report

Mayank Tiwari, Technical Consulting Engineer, and Rama Malladi, Graphics Performance Modeling Engineer, Intel Corporation

Modern CPUs implement parallelism at many levels. The highest level of parallelism in a single CPU node is multithreading. To implement parallelism at the instruction level, CPUs use pipelining. [Editor's note: see **Understanding the Instruction Pipeline** in issue 32 of *The Parallel Universe* for a detailed description of pipelining.] The innermost level of parallelism in a CPU is data-level parallelism, which is implemented through vectorization. It's the capability of a CPU to apply the same operation to multiple elements.

In simple terms, vectorization is the process of rewriting a loop using the single instruction, multiple data (SIMD) instruction set to reduce the number of loop iterations. For example, **Intel® Xeon® processors** support the AVX-512 instruction set. This means an N iteration loop operating on float 32-bit type would in an ideal scenario be executed in N/16 iterations, giving a 16x boost in performance.

In this article, we'll show how to make use of the compiler reports from **Intel® C/C++, and Fortran compilers** to apply and improve vectorization in your applications.

Compiler Optimizations and Reports

When it's feasible, Intel C/C++ and Fortran compilers generate SIMD instructions. This is known as the auto-vectorization feature of the compiler. Vector code is generated by default when using the `-O2` or `-O3` compiler options. However, we don't always get vector code generated successfully, and the compiler gives the reason(s) in optimization reports. For the Intel C/C++ Compiler in Linux and macOS, we can use the option `qopt-report[=n]` to generate an optimization report. In Windows, we can use the `/Qopt-report[:n]` compiler option. The `n` is optional and indicates the level of detail in the report. Valid values are `0` (no report) through `5` (most detailed). For `n=1` through `n=5`, each level includes all the information of the previous level plus some additional information.

Optimization Report: Vector [vec]

Listing 1 shows the code for a simple vector add. The compiler optimization report is shown in **Figure 1**.

```
17:    clock_t t1, t2;
18:
19:    t1 = clock();
20:    for(int k = 0; k < N_ITER; k++){
21:           for(int i = 0; i < size; i++)
22:                  crr[i] = arr[i] + brr[i];
23:    }
24:    t2 = clock();
25:
26:    cout << (double)(t2 - t1)/CLOCKS_PER_SEC << " seconds" << endl;
```

Listing 1. Simple vector add code

The command-line for Windows is:

```
icl vec01.cpp -o vec01 /Qopt-report:5 /O3
```

The command-line for Linux is:

```
icpc vec01.cpp -o vec01 -qopt-report=5 -O3
```

```
LOOP BEGIN at C:\Users\Vectorization\vec01\vec01.cpp(20,2)
   remark #15542: loop was not vectorized: inner loop was already vectorized

   LOOP BEGIN at C:\Users\Vectorization\vec01\vec01.cpp(21,3)
      remark #15305: vectorization support: vector length 4
      remark #15399: vectorization support: unroll factor set to 4
      remark #15300: LOOP WAS VECTORIZED
      remark #15475: --- begin vector cost summary ---
      remark #15476: scalar cost: 6
      remark #15477: vector cost: 1.250
      remark #15478: estimated potential speedup: 4.800
      remark #15488: --- end vector cost summary ---
      remark #25015: Estimate of max trip count of loop=64
   LOOP END
LOOP END
```

1 **Compiler optimization report for a simple vector add code**

The first remark in the report is for the outer loop at `vec01.cpp: line 20`. The compiler report tells us clearly that the outer loop has not been vectorized, while the inner loop has been vectorized. Later, we will see that the inner loop has been vectorized and the vector length is four, which means the CPU is able to process four integer elements in one instruction. The compiler also gives an estimate of our potential speedup for vector versus scalar execution.

To get the actual speedup, we can use **Intel® Advisor**, which does dynamic/runtime analysis. The compiler also estimates the maximum number of trips for vectorized loop execution. We can calculate the approximate trip counts as:

```
array_size/(vector_length * #loop_unrolling)
```

In this example code, it is `1024/(4*4)=64`. With peel and remainder loops, this formula may need some tweaking. Finally, from the vector length of four, we can guesstimate that the SIMD instructions used were operating on 128 bits of data (= `4*32-bit INTs`), and these are likely SSE instructions.

Now, let's recompile the same code with new options.

For Windows:

```
icl vec01.cpp -o vec01 /Qopt-report:5 /O3 /Qvec-
```

For Linux:

```
icpc vec01.cpp -o vec01 -opt-report=5 -O3 -no-vec-
```

Figure 2 shows that with the vectorization disabled option, the compiler disables auto vectorization, so the new compiler report looks different.

```
LOOP BEGIN at C:\Users\Vectorization\vec01\vec01.cpp(20,2)
   remark #15344: loop was not vectorized: vector dependence prevents vectorization
     LOOP BEGIN at C:\Users\Vectorization\vec01\vec01.cpp(21,3)
      remark #15540: loop was not vectorized: auto-vectorization is disabled with /Qvec- flag
      remark #25438: unrolled without remainder by 2
   LOOP END
LOOP END
```

2 **Optimization report when auto vectorization is disabled during compilation**

Now let's try compiling the vector add code with the `/QxHost (Windows*)` or `-xHost (Linux*)` option (**Figure 3**). This instructs the compiler to choose the highest available SIMD instruction set on the host machine. For modern client platforms, that would be AVX2. For servers, it could be AVX-512.

```
LOOP BEGIN at C:\Users\Vectorization\vec01\vec01.cpp(20,2)
   remark #15542: loop was not vectorized: inner loop was already vectorized

   LOOP BEGIN at C:\Users\Vectorization\vec01\vec01.cpp(21,3)
      remark #15305: vectorization support: vector length 8
      remark #15399: vectorization support: unroll factor set to 4
      remark #15300: LOOP WAS VECTORIZED
      remark #15475: --- begin vector cost summary ---
      remark #15476: scalar cost: 6
      remark #15477: vector cost: 0.620
      remark #15478: estimated potential speedup: 9.600
      remark #15488: --- end vector cost summary ---
      remark #25015: Estimate of max trip count of loop=32
  LOOP END
LOOP END
```

3 **Optimization report for vector add code using the `/QxHost` or `-xHost` compilation**

The speedup with the `*xHost` option is twice that of the auto-vector SSE code. The vector length is also twice as long. The vector length impacts the maximum trip counts, down to 32 from 64. A vector length of eight would imply that vector registers of size 256 bits are being used, thus implying likely AVX2 code generation.

We encourage you to try these compiler options and measure the time of the loops for all possible configurations (scalar, vectorized with SSE, and vectorized with AVX2 instructions).

Optimization Report: Vector Dependencies

The Intel C/C++ and Fortran compilers can apply vectorization, but there can be a few tricky cases where the compiler is unable to determine if vectorization is safe. One such case is vector/data dependency, in which the compiler would generate scalar code for such a loop, thus producing correct results. There can be dependencies like RAW (read-after-write) or WAW (write-after-write). If not resolved, this may lead to an incorrect output if vectorization is forced. However, some of these dependencies can be vectorized safely. **Listing 2** shows such an example with a WAR (write-after-read) dependency. (Scalar and vector execution produce the same output.) The optimization report shows that this loop at line 13 was vectorized successfully.

```
11: t1 = clock();
12:    for(int k = 0; k < N_ITER; k++){
13:          for(int i = 0; i < size-1; i++)
14:                brr[i] = brr[i+1] + arr[i];
15:    }
16: t2 = clock();
```

Listing 2. WAR dependencies can be vectorized easily by the compiler

Optimization Report: Vector Inlining of Functions

Loops with function calls can get vectorized if the function calls are inlined or if it is a SIMD function. **Listing 3** shows an example. If the pragma at line 14 is uncommented, the loop at line 13 doesn't get vectorized.

```
7: int fun(int x, int y){ return x+y; }
:             :
:             :
11:
12:    for(int k = 0; k < N_ITER; k++){
13:           for(int i = 0; i < size; i++){
14:                  //#pragma noinline
15:                  brr[i] = fun(arr[i], z);
16:           }
17:    }
```

Listing 3. Inlined functions get auto vectorized

With inline replacement of `fun()`, the compiler auto-vectorizes the loop. If the function body is large, then the function may not get inline replaced, meaning the loop might not get vectorized. The compiler optimization report shown in **Figure 4** has some interesting details. It says that vectorization is "feasible but seems inefficient" and that adding `declare simd` to the function might help vectorization. (We recommend looking up the compiler help documents for details on these optimizations.)

```
LOOP BEGIN at C:\Users\Vectorization\vec02\vec02.cpp(13,2)inlined into
      remark #15335: loop was not vectorized: vectorization possible but seems inefficient.
Use vector always directive or /Qvec-threshold0 to override
      remark #15475: --- begin vector cost summary ---
      remark #15476: scalar cost: 106
      remark #15477: vector cost: 119.000
      remark #15478: estimated potential speedup: 0.890
      remark #15485: serialized function calls: 1
      remark #15488: --- end vector cost summary ---
      remark #15489: --- begin vector function matching report ---
      remark #15490: Function call: fun..0(int, int) with simdlen=2, actual parameter types:
(vector,uniform)    [ LOOP BEGIN at C:\Users\Vectorization\vec02\vec02.cpp (21,13) ]
      remark #15545: SIMD annotation was not seen, consider adding 'declare simd' directives
at function declaration
      remark #15493: --- end vector function matching report ---
   LOOP END
```

4 **Optimization report for a code snippet without inline function replacement**

Optimization Report: Vector Conditional Execution

Loops with `if-else` conditions may not get vectorized efficiently. The code snippet in **Listing 4** doesn't get vectorized by the compiler. Here, we observe a strange behavior: the compiler predicts speedup with a vectorized loop. However, when we vectorize the loop by adding `#pragma openmp simd`, we observe a degradation in performance.

```
12:    t1 = clock();
13:          for(int k = 0; k < N_ITER; k++){
14:                //#pragma omp simd
15:                for(int i = 0; i < size; i++){
16:                      if(i % 2 == 0)
17:                            arr[i] = brr[i] * z;
18:                      else
19:                            arr[i] = brr[i] + z;
20:                }
21:          }
22:    t2 = clock();
```

Listing 4. Loops with `if-else` conditions may not get vectorized efficiently

For loops with `if-else` conditions, the compiler usually creates two versions for the loop, with `if` and `else` conditions separated in the two versions. This can be seen clearly from the optimization report (**Figure 5**), in which the compiler also shows an associated vectorization overhead. A similar optimization report is available for the `else` condition as `<Predicate Optimized v2>`.

```
<Predicate Optimized v1>
      remark #25423: Condition at line 15 hoisted out of this loop
      remark #15305: vectorization support: vector length 4
      remark #15309: vectorization support: normalized vectorization overhead 0.417
      remark #15475: --- begin vector cost summary ---
      remark #15476: scalar cost: 3
      remark #15477: vector cost: 0.750
      remark #15478: estimated potential speedup: 3.990
```

5 **Optimization report for vectorization of a loop code with `if-else` conditions**

Optimization Report: Vector Stride Access

Vectorization of loops is most efficient when the arrays have unit stride access. For **Listing 5**, the inner loop at line 17 doesn't get vectorized unless we add `#pragma omp simd`. However, adding the pragma and forcing vectorization degraded performance because of the non-unit array access. Forcing vectorization may not always give better performance.

```
13: int z = 2;
14: t1 = clock();
15:    for(int k = 0; k < N_ITER; k++){
16:           //#pragma omp simd
17:           for(int i = 0; i < size/2 ; i++){
18:                  arr[i*z] = brr[i];
19:           }
20:    }
21: t2 = clock();
```

Listing 5. Non-unit strides can degrade vectorization performance

Optimization Report: Vector Definitions in Headers

One more scenario where vectorization can become a challenge is if you're using user-defined header files. **Listing 6** shows such an example. **Figure 6** shows the optimization report.

```
/*********************  util.h  *********************/
1:      int sum (int x, int y);

/********************* sum.cpp  *********************/
1:      int sum (int x, int y){
2:             return x+y;
3:      }

/******************** vec05.cpp *********************/
4:      #include "util.h"
5:
:
12:     clock_t t1, t2;
13:     int z = 2;
14:
15:     t1 = clock();
16:     for(int k = 0; k < N_ITER; k++){
17:            //#pragma omp simd
18:            for(int i = 0; i < size; i++){
19:                   arr[i] = sum(z, z);
20:            }
21:     }
22:     t2 = clock();
```

Listing 6. Auto-vectorization may fail or perform poorly with external references for functions

```
LOOP BEGIN at C:\Users\Vectorization\vec05\vec05.cpp(18,3) inlined into C:\Users
\Vectorization\vec05\vec05.cpp(38,2)
      remark #15382: vectorization support: call to function sum(int, int) cannot be
vectorized   [ C:\Users\Vectorization\vec05\vec05.cpp(19,4) ]
      remark #15344: loop was not vectorized: vector dependence prevents vectorization
   LOOP END
```

6 Optimization report for vectorization of loop with external references for functions

When `#pragma omp simd` was added before the inner loop, the loop does get vectorized, but the speedup estimated is less than one. However, the actual runtime of the code did show 33% improvement with the pragma. These discrepancies could occur because the compiler performs static analysis (and not dynamic analysis) of the generated code for estimating speedup. **Figure 7** shows such an optimization report.

```
remark #15301: SIMD LOOP WAS VECTORIZED
remark #15449: unmasked aligned unit stride stores: 1
remark #15475: --- begin vector cost summary ---
remark #15476: scalar cost: 104
remark #15477: vector cost: 112.750
remark #15478: estimated potential speedup: 0.920
remark #15485: serialized function calls: 1
remark #15488: --- end vector cost summary ---
```

7 Optimization report for vectorization of loop with external references using `pragma omp simd`

If we bring the function definition of `sum()` within the header file `util.h`, we see about a 20x improvement in the performance. The compiler could vectorize the loop even without `#pragma omp simd`. Inspecting the generated code, or using **Intel® VTune™ Profiler** or Intel Advisor, can give you valuable insights into these speedups.

Boosting the Performance of HPC Cluster Workloads Using Intel® MPI Library Tuning

Enhancing Operational Efficiency with a Simple-to-Use, Runtime-Based Autotuner

Amarpal S Kapoor, Technical Consulting Engineer, and Marat Shamshetdinov, Software Development Engineer, Intel Corporation

This article is a sequel to **our article** from **issue 41** of *The Parallel Universe* about MPI tuning using the utilities in **Intel® MPI Library**. Our previous article provided an overview of tuning utilities in Intel MPI Library, typical user workflows for Autotuner, and the resulting performance gains with Autotuner for **Intel MPI Benchmarks**. In this article, we'll demonstrate performance gains on six real-world HPC applications.

The data is a summary of 168 tests run on Intel's **Intel® Xeon® processor**-based HPC cluster named Endeavour. We used two types of 2nd generation **Intel® Xeon® Scalable processors**:

1. Intel® Xeon® Platinum 9242 processor
2. Intel® Xeon® Platinum 8260L processor

They were connected over two HPC interconnects:

1. Intel® Omni-Path Architecture (Intel® OPA)
2. Mellanox* Quantum™ HDR

Methodology

We started with a list of six popular HPC applications from domains such as quantum molecular dynamics, molecular dynamics, and computational fluid dynamics (**Table 1**). We also picked solvers like algebraic multigrid and conjugate gradient, which represent Krylov subspace solvers that are frequently used in HPC applications relying on approximate solutions of partial differential equations.

Table 1. HPC applications and workloads used in our tuning experiments

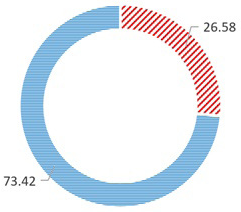
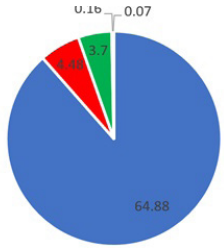
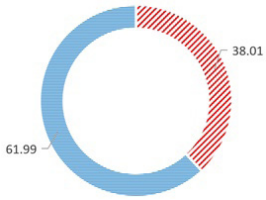
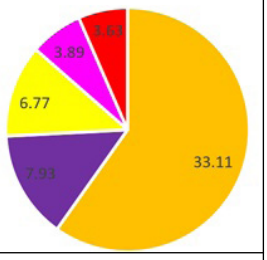
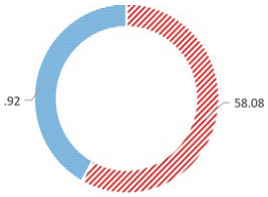
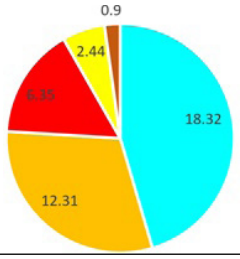
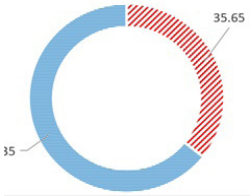
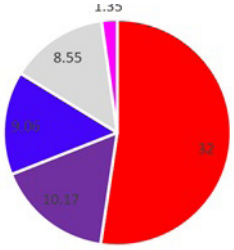
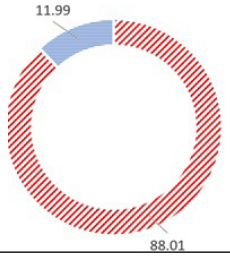
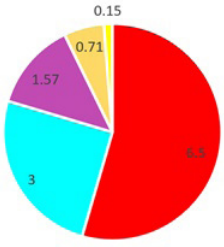
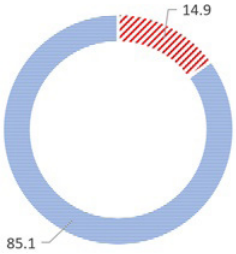
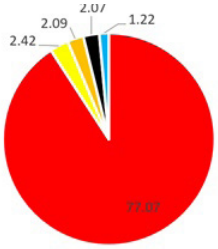
Application	Version	Workload
CPMD	3.13	il64_md_tg8
AMG	2.1	solver4
LAMMPS	2018-12-12	Copper
OpenFOAM	7.0	motorbike-2m
HPCG	3.0	hpcg_128
Amber	18	native

The applications were built using:

- **Intel® C++ Compiler** 2020 U1
- **Intel® Fortran Compiler** 2020 U1
- **Intel® Math Kernel Library** 2019 U5
- **Intel® MPI Library** 2019 U8

In line with the application selection methodology for Autotuner, described in our previous article, we analyzed the nature of the above applications using **Intel® VTune™ Profiler** Application Performance Snapshot 2020 U2 [4], APS for short.

Table 2. Summary of performance traits from APS

Application/ Topology	Wall Time (S)	Time Wheel (% of Wall Time)	MPI Function Wheel (% of Wall Time)
CPMD/ [CLX9242] 16:1536X1/96*	1,352.91		
AMG/ [CLX9242] 11:1024X1/96	5.3		
LAMMPS/ [CLX9242] 16:1536X1/96	14.28		
OpenFOAM/ [CLX9242] 16:1536X1/96	25.84		
HPCG/ [CLX9242] 16:1536X1/97	238.29		
Amber/ [CLX8268] 4:192X1/49	80.78		

Legend

Topology format – [CLX9242]16:1536X1/96 represents execution on 16 Intel® Xeon® Platinum 9242 Processor nodes (Code name - Cascade Lake) with 1536 MPI Ranks, 1 OMP thread and 96 MPI Ranks per node.

Table 2 summarizes the collected APS statistics. The time wheel shows the percentage of time split between MPI and serial execution. The MPI function wheel shows the percentage of elapsed time spent in the top five MPI hotspots. Considering just the top five MPI functions was sufficient because, on average, they represented 96.6% of the total MPI time.

All applications except HPCG and LAMMPS were found to spend much of their time in MPI regions. Also, all applications had at least one MPI collective communication hotspot (from the top five MPI functions). `Allreduce, Alltoall`, `Bcast`, `Reduce`, `Reduce_scatter`, and `Allgatherv` were the top collective communication routines used in our test applications (**Table 1**).

Table 3 presents another important metric, MPI imbalance, as a percentage of wall time. For most applications, a significant portion of their MPI time was spent waiting for other ranks. There could be several reasons on the application side that might result in such load imbalances, including:

- Sub-optimal domain
- Matrix decomposition
- Rank-specific conditional execution

These are inefficiencies attributed to the MPI side. Also, there might be a possibility to tune this further using Autotuner.

Table 3. MPI imbalance

Application	MPI Imbalance (% of Wall Time)
CPMD	41.02
AMG	7.35
LAMMPS	13.29
OpenFOAM	34.42
HPCG	9.94
Amber	70.37

Considering the applications' traits presented above, we tuned all of them through Autotuner using a two-step procedure. Step 1 generates an application-specific tuned configuration in addition to tuning on-the-fly:

```
$ I_MPI_TUNING_MODE=auto I_MPI_TUNING_BIN_DUMP=tuning.dat
mpiexec.hydra -n X -ppn Y -f hostfile ./binary
```

Step 2 uses the tuned configuration from step 1, which was stored in tuning data:

```
$ I_MPI_TUNING_BIN=tuning.dat mpiexec.hydra -n X -ppn Y -f hostfile ./binary
```

Although step 1 alone might give performance gains, our current recommendation would be to use the two-step approach.

We should note that Autotuner is an application-specific tuning tool and must be run for each application, workload, and configuration of interest. The generated tuning settings are valid only for the scenario at hand. However, an application's users can maintain a central tuning file that can be enriched over time with tuned settings for a large range of workloads.

We tried two variants of step 1 for our tests. The first variant (step 1A) used the default configuration. In the second variant (step 1B), Autotuner tuning logic was fed with timing data from 20 iterations (to better account for noise). This was achieved by setting `I_MPI_TUNING_AUTO_ITER_NUM=20`. The default value of this variable is 1. It's important to be careful in selecting an appropriate value for `I_MPI_TUNING_AUTO_ITER_NUM` to ensure that Autotuner tuning can iterate through all possible algorithms. The total number of invocations of a collective operation for a message size should at least be equal to the value of `I_MPI_TUNING_AUTO_ITER_NUM` multiplied by the number of algorithms. You can find the number of algorithms for a collective from the Intel MPI Library terminal-based documentation utility for environment variables, **impi_info**.

For example:

```
$ impi_info -v I_MPI_ADJUST_ALLREDUCE | grep -i
  range: 0-24
```

Here, the range shows 25 possible algorithms for `Allreduce` in the current version of Intel MPI Library. Therefore, if `I_MPI_TUNING_AUTO_ITER_NUM` is set to 20, we can assume the application calls `Allreduce` on a specific message size at least `20*25=500` times. The same condition must be valid for every message size that `Allreduce` acts upon. Also, the same must be valid (with suitable correction based on `<range>`) for all collective operations in the application. We can easily obtain the number of MPI function invocations for every message size from APS, using the `–mDPF` postprocessing flag in the following command (after the APS profile has been collected with `APS_STAT_LEVEL=3`):

```
$ aps-report aps_result_<postfix> -mDPF
```

Step 1B generates and uses tuning data (with data from 20 iterations fed to tuning logic):

```
$ I_MPI_TUNING_MODE=auto I_MPI_TUNING_BIN_DUMP=tuning.dat
I_MPI_TUNING_AUTO_ITER_NUM=20 mpiexec.hydra -n X -ppn Y -f hostfile
./binary
```

Results

We tested all the applications with two Autotuner configurations:

1. **C1:** Run step 1A followed by step 2,
2. **C2:** Run step 1B followed by step 2,

on two interconnects, Intel OPA and Mellanox Quantum HDR. We averaged the baseline and step 2 runs over three executions, while steps 1A and 1B were run just once per application per configuration to save time. The APS data summarized in **Table 2** came from a dedicated set of runs (one per application) on respective hardware and topology. In total, around 168 executions were made to generate data for calculation of figure of merit (FOM) based, averaged performance gains.

For Amber and HPCG, we observed no significant performance gains. This implies that Autotuner couldn't find a tuning better than Intel MPI Library default out-of-box (OOB) tuning.

By design, the worst performance Autotuner delivers is the same as OOB tuning, which is already highly performant in the tested scenarios. The best performance we can expect from Autotuner should be proportional (within reasonable limits) to the MPI time of the application. We found significant performance gains for CPMD, AMG, LAMMPS, and OpenFOAM on both Intel OPA and Mellanox Quantum HDR.

Tables 4 and **5** show the FOM-based improvements for each application on both interconnects. We report the gain numbers for the best-performing configuration along with the configuration name. Note that the same processor SKU wasn't available on the two interconnects we tested. However, the total number of MPI ranks in both cases was the same. As mentioned before, selecting the tuning algorithms in Intel MPI Library depends heavily on the interconnect (OFI provider), processor, and topology, making it difficult to compare gains resulting from different hardware. The FOM gain percentage was calculated as:

$$\text{FOM Gain \%} = \max\left\{\text{abs}\left(1 - \frac{\text{FOM\_Vavg\_C1}}{\text{FOM\_Bavg}}\right)*100 \;,\; \text{abs}\left(1 - \frac{\text{FOM\_Vavg\_C2}}{\text{FOM\_Bavg}}\right)*100\right\}$$

where:

- `FOM_Bavg` is the averaged FOM without Autotuner (baseline)
- `FOM_Vavg_C1` is the averaged FOM from step 2 for the C1 configuration
- `FOM_Vavg_C2` is the averaged FOM from step 2 for the C2 configurationTable

Table 4. Application-specific performance gains derived from Autotuner on an Intel® Xeon® Platinum 9242 processor-based cluster with Mellanox Quantum HDR interconnect

Application	FOM	FOM Gain %	Topology**	Machine
CPMD	Seconds	[C1]# 31.16	16:1536X1/96	Intel® Xeon® Platinum 9242 processor with Mellanox Quantum HDR interconnect
AMG	(size*iter)/solve_time	[C2]# 27.96	11:1024X1/96	
LAMMPS	Timesteps/second	[C1] 2.98	16:1536X1/96	
OpenFOAM	Seconds	[C1=C2] 5.00	16:1536X1/96	

**Topology – Nodes:Ranks x Threads/Ranks-per-node

Autotuner configuration used

Table 5: Application-specific performance gains derived from Autotuner on an Intel® Xeon® Platinum 8260L processor-based cluster with Intel OPA interconnect

Application	FOM	FOM Gain %	Topology**	Machine
CPMD	Seconds	[C2]# 9.24	32:1536X1/48	Intel® Xeon® Platinum 8260L processor with Intel OPA interconnect
AMG	(size*iter)/solve_time	[C1]# 6.68	22:1024X1/48	
LAMMPS	Timesteps/second	~0	32:1536X1/48	
OpenFOAM	Seconds	[C2] 40.92	32:1536X1/48	

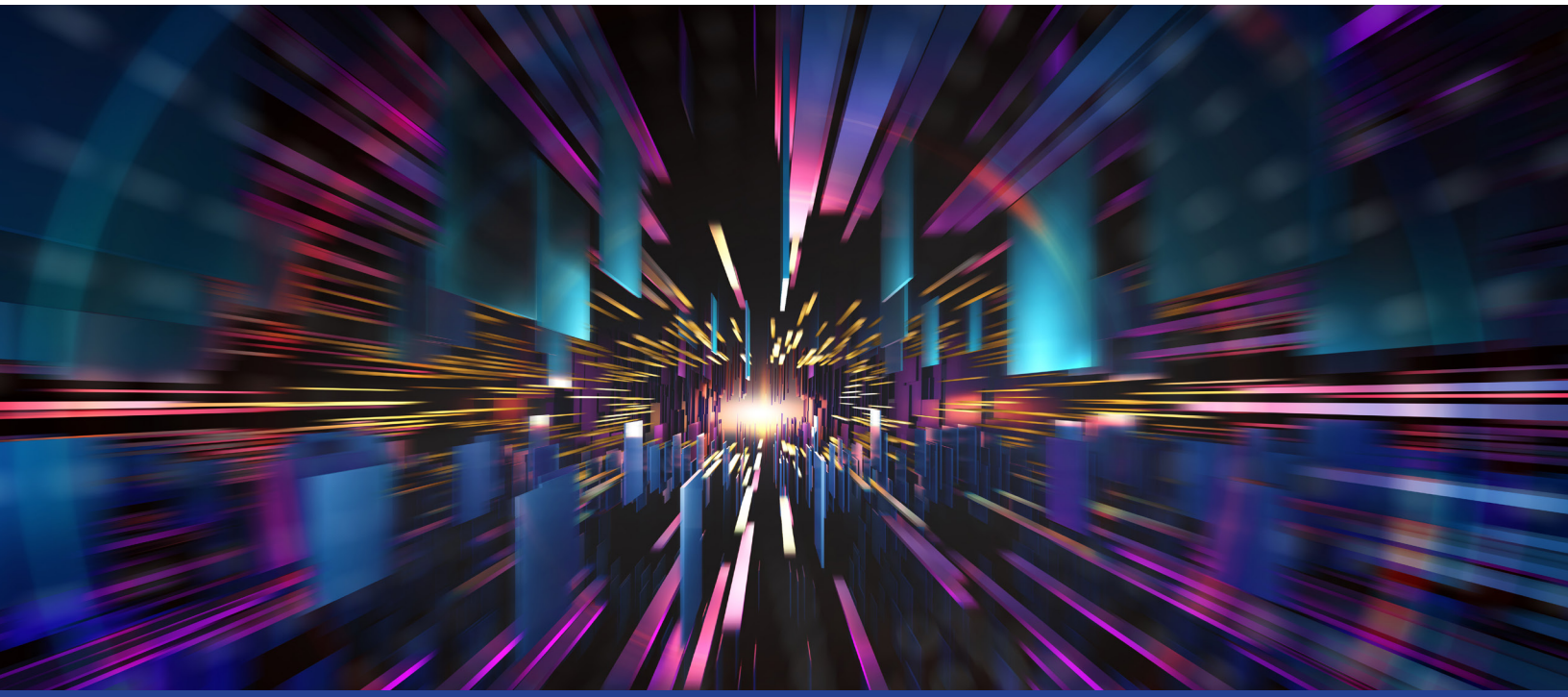
**Topology – Nodes:Ranks x Threads/Ranks-per-node

Autotuner configuration used

Tuning for Performance Gains

This article presented high-level performance characteristics of six HPC applications: CPMD, AMG, LAMMPS, OpenFOAM, HPCG, and Amber (**Table 1**) using Intel VTune Profiler APS (**Table 2**). We also demonstrated the use of Autotuner for HPC applications, along with performance gains, which were as high as 40.92%.

OOB tuning data that is shipped with the Intel MPI Library is highly performant for a large range of applications, workloads, and topologies. However, generating tuning data is an expensive task that demands scanning a large search space. Limited time between release schedules of Intel MPI Library demands carefully selecting a generic search space. Consequently, there are possibilities, such as the ones we demonstrated here, where HPC users can benefit further by running Autotuner, which is an easy-to-use, runtime-based tuning utility that requires no code changes or recompilation using special flags. With minimal overhead, on-the-fly tuning capability for production jobs, and the ability to build a tuning database over time for quick and simple reuse, Autotuner is a powerful tool that can significantly enhance the operational efficiency of modern HPC clusters using Intel MPI Library.



Simplifying Cluster Use

Introducing OpenHPC 2.0

Jeremy Siadal, Senior Software Engineer, Intel Corporation

With the release of **OpenHPC 2.0**, the **OpenHPC Open Source Software (OSS) project** updates its software stack to support the latest **CentOS 8** and **OpenSUSE LEAP** releases. If you aren't already familiar with OpenHPC, it's a **Linux Foundation** project whose mission is to create a reference collection of open-source HPC software components. For the HPC user (and administrator), this simplifies the deployment and use of HPC software by aggregating prebuilt packages into a single repository.

OpenHPC is a community project with academic, government, and commercial members. With version 1.0 released in 2016, the project has seen increasing annual adoption and now includes almost 100 software components and over 300 packages. A 15-member Technical Steering Committee oversees software development, maintaining the **GitHub repository** and Open Build System infrastructure, as well as software decision-making and planning. Like the project itself, the Technical Steering Committee is comprised of HPC community members. Committee members are elected to one-year terms each June and anyone can be nominated (or self-nominate).

A Cluster Software Distribution

You can think of OpenHPC as a cluster software distribution for the sysadmin. The RPM (Red Hat Package Manager) design doesn't easily support installation by regular users, so OpenHPC is well-suited to enhance the Linux distribution and other vendor software repositories used for installing HPC systems.

Its key benefit is that with software packages already built from source, tested, and in one location, adding and upgrading system-wide components is fast and simple.

Contrast this to package managers such as Spack and EasyBuild, where packages are downloaded and then built locally, but can be installed by regular users into their own accounts. Conveniently, OpenHPC includes both Spack and EasyBuild as available components.

HPC technology and software evolves rapidly. And as Linux distributions grow, they add popular HPC software packages built using recent source code. Other HPC software becomes unmaintained or sees minimal use. OpenHPC components are regularly reviewed and may also be removed from the distribution. For example, with the OpenHPC 2.0 release, the munge package is no longer included because both CentOS and OpenSUSE LEAP include a current version. This doesn't mean packages are removed when Linux distributions include them, since performance libraries and binaries in OpenHPC are built with a variety of toolchains not found in major Linux distributions.

Conversely, new packages are added regularly. New to OpenHPC 2.0 are the Libfabric and UCX communication libraries. MagPie, buildtest, OSU Benchmarks, and Annobin support have been accepted for a future release.

All completed packages undergo automated testing as part of the OpenHPC Build Service.

Installation

As a software distribution supporting CentOS and OpenSUSE LEAP, all components are available as RPM packages in a centralized repository. These packages are easily installed, along with all dependencies, using DNF, Zypper, or a similar package manager.

To install the repository on an **Intel® Xeon® processor**-based server running CentOS 8.2, simply execute:

```
dnf install http://repos.openhpc.community/OpenHPC/2/ \
CentOS_8/x86_64/ohpc-release-2-1.el8.x86_64.rpm
```

Once the repository is active, OpenHPC components can be installed either directly—components include the suffix –`ohpc` in their package name—or collectively through meta-RPMs. Meta-RPMs define collections of similar components that will be installed along with the meta-RPM. Meta-RPMs include the prefix `ohpc-` in their package names.

As with most Linux distributions, users can install as few or as many software packages as they need. The benefit of using RPM package managers is that all prerequisite software for any package is installed automatically. With a few exceptions for system-level components, OpenHPC software is installed into `/opt/ohpc`, and the distribution can be made available cluster-wide by sharing this directory.

As an example, users can install the Boost libraries prebuilt using gcc and OpenMPI (which will also be installed automatically) by running:

```
dnf install boost-gnu9-openmpi4-ohpc
```

Software is available to support the installation and management of a new cluster. OpenHPC provides Warewulf cluster provisioning and management software. It also supports xCAT distributed computing management software. The OpenHPC repository includes the SLURM and OpenPBS resource managers, as well as other administrative utilities. Installation instructions, including automated scripts, are available for download from **http://openhpc.community**.

The benefit of the Warewulf Cluster Management Software in OpenHPC is that the software on each server is applied from the same central image, either through a full copy or a hybridized copy/remote mount. When running in its default stateless configuration, a reboot will restore any changes to local software. This avoids minor inconsistencies that can lead to stability or performance issues.

Verifying the Cluster

A consistent software image reduces inconsistencies. But physical setup, software configuration, and hardware and firmware consistency present additional sources for error, which are important to address. For a regular user, a simple precheck of the assigned nodes can help identify potential problems.

While OpenHPC includes tools for system checking, what's needed is an analysis of all cluster hardware and software. Luckily, if you already have **Intel® Parallel Studio XE Cluster Edition**, the necessary analysis tool is already installed. As the name implies, **Intel® Cluster Checker** can verify an entire cluster for hardware and software consistency, functionality of components, and expected performance.

Environment Switching Made Easy with Toolchains

A key advantage of OpenHPC is the incorporation of environment modules using Lmod, a Lua-based environment module system developed by the Texas Advanced Computing Center (TACC). An OpenHPC component package also installs its corresponding environment module.

OpenHPC aims to be vendor-neutral regarding both Linux distribution and processor architecture. None of its included components are specific to any vendor's hardware. There's an exception for software development tools. Although OpenHPC doesn't include Intel® developer tools, it does integrate them. Once the Intel® compilers or **Intel® MPI Library** is installed and licensed, an OpenHPC package can be installed to generate environment modules, including individual modules for each recent version that it finds. On CentOS, use the command:

```
dnf install intel-compilers-devel-ohpc intel-mpi-devel-ohpc
```

It's important to note that the OpenHPC module generation currently supports full Intel Parallel Studio, but it won't generate modules for Intel Cluster Runtimes.

The modules are structured hierarchically, hiding dependent modules until the parent modules are loaded. This creates many toolchain permutations. The GNU compiler suite 9.2 is included, and support is provided for both the Intel and ARM compilers. Once a compiler is loaded, the toolchain can be extended to the MPICH, MVAPICH, OpenMPI, or Intel MPI libraries. Additional serial or parallel numerical libraries built with the underlying toolchain can then be loaded. Any combination of these is supported, and Lmod prevents conflicting environments from being enabled concurrently.

The LLVM 10 compiler is also available in OpenHPC. However, it's not currently enabled for any development toolchains.

How does this work in practice? Let's say I'm interested in running the Laghos (LAGrangian High-Order Solver) mini-app from the CEED software suite. Laghos requires several parallel libraries to run, including the MFEM, METIS, and HYPRE libraries.

Let's further assume that I want Laghos to use the GNU gcc compiler and the MVAPICH MPI library.

If I'm running on a cluster with the full OpenHPC distribution installed, this is easy. To build the application from my downloaded source, I need to initialize the environment:

```
module load gnu9
module load mvapich2
module load mfem
```

The included mfem module automatically pulls in the metis and hypre modules, among others. That's it. With the environment configured, I can build Laghos from source.

```
cd laghos
make install
```

If I want a second build with the Intel compilers and Intel MPI Library, I just need to unload the modules in use and load the new environment.

```
module reset
module load intel
module load impi
module load mfem
```

With the module hierarchy, a different mfem environment module is loaded here—this one pointing to the MFEM library compiled with Intel Parallel Studio XE. To run laghos later, I either reload the modules, add them to my login profile, or even create my own LMOD module.

What's Next?

OpenHPC 2.0 is a point-in-time repository of packages and versions. The build system also includes additional repositories for validated updates to the base software distribution, as well as a factory for continuous development. OpenHPC 1.3.9 will continue to be available for users on CentOS 7 or SUSE Linux Enterprise (SLE) 12, at least until the end of 2020. Further package development will be directed to OpenHPC 2.1.

As a cluster administrator or software developer on a dedicated system, the OpenHPC software distribution simplifies installation of popular HPC community software, avoiding the pitfalls of building software packages locally. For developers provisioning their own clusters, OpenHPC provides recipes and software. And for the user, the included Lmod package simplifies setting up and switching between development environments.

To learn more, visit **OpenHPC**.

ARTICLE HIGHLIGHTS

Brightskies Deploys Open Source RTM Application for Easy Optimization across Multiple Architectures

To improve computing performance for seismic imaging on heterogeneous hardware without the problems of vendor lock-in, Brightskies chose to implement reverse time migration (RTM) using DPC++.

Read On >

intel® software

THE PARALLEL UNIVERSE

Printed in USA 0720/SS Please Recycle