OpenVINO™ Toolkit and FPGAs
Floating-Point Reproducibility in Intel® Software Tools
Comparing C++ Memory Allocation Libraries
CONTENTS

Letter from the Editor 3

Edge-to-Cloud Heterogeneous Parallelism with OpenVINO™ Toolkit  
by Henry A. Gabb, Senior Principal Engineer, Intel Corporation

OpenVINO™ Toolkit and FPGAs 5  
A Look at the FPGA Targeting of this Versatile Visual Computing Toolkit

Floating-Point Reproducibility in Intel® Software Tools 15  
Getting Beyond the Undertainty

Comparing C++ Memory Allocation Libraries 25  
Boosting Performance with Better Dynamic Memory Allocation

LIBXSMM*: An Open-Source-Based Inspiration for Hardware and Software Development at Intel 37  
Meet the Library that Targets Intel® Architecture for Specialized Dense and Sparse Matrix Operations and Deep Learning Primitives

Advancing the Performance of Astrophysics Simulations with ECHO-3DHP 49  
Using the Latest Intel® Software Development Tools to Make More Efficient Use of Hardware

Your Guide to Understanding System Performance 57  
Meet Intel® VTune™ Amplifier’s Platform Profiler

For more complete information about compiler optimizations, see our Optimization Notice.
Edge-to-Cloud Heterogeneous Parallelism with OpenVINO™ Toolkit

In a previous editorial, I mentioned that I used to dread the heterogeneous parallel computing future. Ordinary parallelism was hard enough. Spreading concurrent operations across different processor architectures would add a level of complexity beyond my programming ability. Fortunately, as the University of California at Berkeley Parallel Computing Laboratory predicted in 2006, this increasing complexity would force a greater separation of concerns between domain experts and tuning experts. (See The Landscape of Parallel Computing Research: A View from Berkeley for details.) For example, I know how to apply the Fast Fourier Transform in my scientific domain, but I would never dream of writing an FFT myself because experts have already done it for me. I can just use their libraries to get all the benefit of their expertise.

With that in mind, James Reinders, our editor emeritus, joins us again to continue his series on FPGA programming—this time, to show us how to exploit heterogeneous parallelism using Intel’s new OpenVINO™ toolkit (which stands for open visual inference and neural network optimization). OpenVINO™ Toolkit and FPGAs describes this toolkit to incorporate computer vision in applications that span processor architectures, including FPGAs, from edge devices all the way to cloud and data center. OpenVINO toolkit encapsulates the expertise of computer vision and hardware experts and makes it accessible to application developers. (I also interviewed James recently about the future of Intel® Threading Building Blocks (Intel® TBB). Among other things, we discuss how the Intel TBB parallel abstraction embodies the separation the concerns between application developers and parallel runtime developers, and how the IntelTBB Flow Graph API could provide a path to heterogeneous parallelism. You can find this interview on the Tech.Decoded knowledge hub.)

The remaining articles in this issue start close to the metal and gradually move up the hardware/software stack. Floating-Point Reproducibility in Intel® Software Tools discusses the inexactness of binary floating-point representations and how to deal with it using the Intel®
compilers and performance libraries. **Comparing C++ Memory Allocation Libraries** does just what the title says. It compares the performance of various C++ memory allocation libraries using two off-the-shelf benchmarks, then digs into the profiles using **Intel® VTune™ Amplifier** to explain the often significant performance differences. Moving a little higher up the stack, **LIBXSMM: An Open-Source-Based Inspiration for Hardware and Software Development at Intel** describes a library that’s part research tool and part just-in-time code generator for high-performance small matrix multiplication—an important computational kernel in convolution neural networks and many other algorithms.

At the application level of the hardware/software stack, **Advancing the Performance of Astrophysics Simulations with ECHO-3D HPC**, from our collaborators at CEA Saclay and LRZ, describes how they optimized one of their critical applications using various tools in **Intel® Parallel Studio XE**. Finally, at the top of the stack, we have **Your Guide to Understanding System Performance**. This article gives an overview of the Platform Profiler tech preview feature in Intel® VTune™ Amplifier. As the name implies, Platform Profiler monitors the entire platform to help diagnose system configuration issues that affect performance.

Future issues of *The Parallel Universe* will bring you articles on parallel computing using Python*, new approaches to large-scale distributed data analytics, new features in Intel® software tools, and much more. In the meantime, check out **Tech.Decoded** for more information on Intel solutions for code modernization, visual computing, data center and cloud computing, data science, and systems and IoT development.

**Henry A. Gabb**  
October 2018
In this article, we'll take a firsthand look at how to use Intel® Arria® 10 FPGAs with the OpenVINO™ toolkit (which stands for open visual inference and neural network optimization). The OpenVINO toolkit has much to offer, so I'll start with a high-level overview showing how it helps develop applications and solutions that emulate human vision using a common API. Intel supports targeting of CPUs, GPUs, Intel® Movidius™ hardware including their Neural Compute Sticks, and FPGAs with the common API. I especially want to highlight another way to use FPGAs that doesn't require knowledge of OpenCL* or VHDL* to get great performance. However, like any effort to get maximum performance, it doesn't hurt to have some
understanding about what’s happening under the hood. I’ll shed some light on that to satisfy your curiosity—and to help you survive the buzzwords if you have to debug your setup to get things working.

We’ll start with a brief introduction to the OpenVINO toolkit and its ability to support vision-oriented applications across a variety of platforms using a common API. Then we’ll take a look at the software stack needed to put the OpenVINO toolkit to work on an FPGA. This will define key vocabulary terms we encounter in documentation and help us debug the machine setup should the need arise. Next, we’ll take the OpenVINO toolkit for a spin with a CPU and a CPU+FPGA. I’ll discuss why “heterogeneous” is a key concept here (not everything runs on the FPGA). Specifically, we’ll use a high-performance Intel® Programmable Acceleration Card with an Intel Arria® 10 GX FPGA. Finally, we’ll peek under the hood. I’m not the type to just drive a car and never see what’s making it run. Likewise, my curiosity about what’s inside the OpenVINO toolkit when targeting an FPGA is partially addressed by a brief discussion of some of the magic inside.

The Intel Arria 10 GX FPGAs I used are not the sort of FPGAs that show up in $150 FPGA development kits. (I have more than a few of those.) Instead, they’re PCIe cards costing several thousand dollars each. To help me write this article, Intel graciously gave me access for a few weeks to a Dell EMC PowerEdge* R740 system, featuring an Intel Programmable Acceleration Card with an Arria 10 GX FPGA. This gave me time to check out the installation and usage of the OpenVINO toolkit on FPGAs instead of just CPUs.

The OpenVINO Toolkit

To set the stage, let’s discuss OpenVINO toolkit and its ability to support vision-oriented applications across a variety of platforms using a common API. Intel recently renamed the Intel® Computer Vision SDK as the OpenVINO toolkit. Looking at all that’s been added, it’s not surprising Intel wanted a new name to go with all the new functionality. The toolkit includes three new APIs: the Deep Learning Deployment toolkit, a common deep learning inference toolkit, and optimized functions for OpenCV* and OpenVX*, with support for the ONNX*, TensorFlow*, MXNet*, and Caffe* frameworks.

The OpenVINO toolkit offers software developers a single toolkit for applications that need human-like vision capabilities. It does this by supporting deep learning, computer vision, and hardware acceleration with heterogeneous support—all in a single toolkit. The OpenVINO toolkit is aimed at data scientists and software developers working on computer vision, neural network inference, and deep learning deployments who want to accelerate their solutions across multiple hardware platforms. This should help developers bring vision intelligence into their applications from edge to cloud. Figure 1 shows potential performance improvements using the toolkit.
Accuracy changes can occur with Fp16. The benchmark results reported in this deck may need to be revised as additional testing is conducted. The results spend on the specific platform configurations and workloads utilized in the testing, and may not be applicable to any particular user's components, computer system, or workloads. The results are not necessarily representative of other benchmarks and other benchmark results may show greater or lesser impact from mitigations. For more complete information about the performance and benchmark results, visit www.intel.com/benchmarks. Configuration: Intel® Core™ i7 processor 6700 at 2.90 GHz fixed. GPU GT2 at 1.00 GHz fixed. Internal ONLY testing performed 6/13/2018, test v3 13.21. Ubuntu* 16.04 OpenVINO™ toolkit 2018 RC4, Intel® Arria 10 FPGA 1150GX. Tests were based on various parameters such as model used (these are public), batch size, and other factors. Different models can be accelerated with different Intel® hardware solutions, yet use the same Intel® Software Tools. Benchmark source: Intel Corporation.

1 Performance improvement using the OpenVINO toolkit

While it's clear that Intel has included optimized support for Intel® hardware, top-to-bottom support for OpenVX APIs provides a strong non-Intel connection, too. The toolkit supports both OpenCV and OpenVX. Wikipedia sums up as follows: “OpenVX is complementary to the open source vision library OpenCV. OpenVX, in some applications, offers a better optimized graph management than OpenCV.” The toolkit includes a library of functions, pre-optimized kernels, and optimized calls for both OpenCV and OpenVX.

The OpenVINO toolkit offers specific capabilities for CNN-based deep learning inference on the edge. It also offers a common API that supports heterogeneous execution across CPUs and computer vision accelerators including GPUs, Intel Movidius hardware, and FPGAs.

Vision systems hold incredible promise to change the world and help us solve problems. The OpenVINO toolkit can help in the development of high-performance computer vision and deep learning inference solutions—and, best of all, it's a free download.
FPGA Software Stack, from the FPGA up to the OpenVINO Toolkit

Before we jump into using the OpenVINO toolkit with an FPGA, let’s walk through what software had to be installed and configured to make this work. I’ll lay a foundational vocabulary and try not to dwell too much on the underpinnings. In the final section of this article, we’ll revisit to ponder some of the under-the-hood aspects of the stack. For now, it’s all about knowing what has to be installed and working.

Fortunately, most of what we need for the OpenVINO toolkit to connect to FPGAs is collected in a single install called the Intel Acceleration Stack, which can be downloaded from the Intel FPGA Acceleration Hub. All we need is the Runtime version (619 MB in size). There's also a larger development version (16.9 GB), which we could also use because it includes the Runtime. This is much like the choice of installing a runtime for Java* or a complete Java Development Kit. The choice is ours. The Acceleration Stack for Runtime includes:

- **The FPGA programmer** (called Intel® Quartus® Prime Pro Edition Programmer Only)
- **The OpenCL runtime** (Intel® FPGA Runtime Environment for OpenCL)
- **The Intel FPGA Acceleration Stack**, which includes the Open Programmable Acceleration Engine (OPAE). OPAE is an open-source project that has created a software framework for managing and accessing programmable accelerators.

I know from personal experience that there are a couple of housekeeping details that are easy to forget when setting up an FPGA environment: the firmware for the FPGA and the OpenCL Board Support Package (BSP). Environmental setup for an FPGA was a new world for me, and reading through FPGA user forums confirmed that I’m not alone. Hopefully, the summary I’m about to walk through, “up-to-date acceleration stack, up-to-date firmware, up-to-date OpenCL with BSP,” can be a checklist to help you know what to research and assure on your own system.

**FPGA Board Firmware: Be Up to Date**

My general advice about firmware is to find the most up-to-date version and install it. I say the same thing about BIOS updates, and firmware for any PCIe card. Firmware will come from the board maker (for an FPGA board like I was using, the Intel® Programmable Acceleration Card [PAC] with an Arria® 10 GX FPGA). Intel actually has a nice chart showing which firmware is compatible with which release of the Acceleration Stack. Updating to the most recent Acceleration Stack requires the most recent firmware. That’s what I did. You can check the latest firmware version with the command `sudo fpgainfo fme`. 
OpenCL BSP: Be Up to Date

You can hardly use OpenCL and not worry about having the right BSP. BSPs originally served in the embedded world to connect boards and real-time operating systems—which certainly predates OpenCL. However, today, for FPGAs, a BSP is generally a topic of concern because it connects an FPGA in a system to OpenCL. Because support for OpenCL can evolve with a platform, it's essential to have the latest version of a BSP for our particular FPGA card. Intel integrates the BSPs with their Acceleration Stack distributions, which is fortunate because this will keep the BSP and OpenCL in sync if we just keep the latest software installed. I took advantage of this method, following the instructions to select the BSP for my board. This process included installing OpenCL itself with the BSP using the `aocl install` command (the name of which is an abbreviation of Altera OpenCL*).

Is the FPGA Ready?

When we can type `aocl list-devices` and get a good response, we're ready. If not, then we need to pause and figure out how to get our FPGA recognized and working. The three things to check:

1. Install the latest Acceleration Stack software
2. Verify firmware is up-to-date
3. Verify the OpenCL is installed with the right BSP

I goofed on the last two, and lost some time until I corrected my error—so I was happy when I finally saw:

<table>
<thead>
<tr>
<th>Device Name:</th>
<th>acl0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package Path:</td>
<td>/home/james/tools/intelrtestack/a10_gx_pac_ias_1_1_pv/opencl/opencl_bsp</td>
</tr>
<tr>
<td>Vendor:</td>
<td>Intel Corp</td>
</tr>
<tr>
<td>Physical Dev Name</td>
<td>Status</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------</td>
</tr>
<tr>
<td>pac_a10_eb000000</td>
<td>Passed</td>
</tr>
<tr>
<td>(pac_a10_eb000000)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For more complete information about compiler optimizations, see our Optimization Notice.
The OpenVINO Toolkit Targeting CPU+FPGA

After making sure that we’ve installed the FPGA Acceleration Stack, updated our board firmware, and activated OpenCL with the proper BSP, we’re ready to install the OpenVINO toolkit. I visited the OpenVINO toolkit website to obtain a prebuilt toolkit by registering and downloading "OpenVINO toolkit for Linux* with FPGA Support v2018R3.” The complete offline download package was 2.3 GB. Installation was simple. I tried both the command-line installer and the GUI installer (setup_GUI.sh). The GUI installer uses X11 to popup windows and was a nicer experience.

We'll start by taking OpenVINO toolkit for a spin on a CPU, and then add the performance of an Intel Programmable Acceleration Card with an Arria 10 GX FPGA.

SqueezeNet

Intel has packaged a few demos to showcase OpenVINO toolkit usage, including SqueezeNet. SqueezeNet is a small CNN architecture that achieves AlexNet*-level accuracy on ImageNet* with 50x fewer parameters. The creators said it well in their paper: “It’s no secret that much of deep learning is tied up in the hell that is parameter tuning. [We make] a case for increased study into the area of convolutional neural network design in order to drastically reduce the number of parameters you have to deal with.” Intel’s demo uses a Caffe SqueezeNet model—helping show how the OpenVINO toolkit connects with popular platforms.
I was able to run SqueezeNet on the CPU by typing:

```bash
$ cd /opt/intel/computer_vision_sdk_fpga <version>/deployment_tools/demo
  ./demo_squeezenet_download_convert_run.sh
```

I was able to run SqueezeNet on the FPGA by typing:

```bash
$ cd /opt/intel/computer_vision_sdk_fpga <version>/deployment_tools/demo
  ./demo_squeezenet_download_convert_run.sh -d HETERO:FPGA,CPU
```

I said “FPGA,” but you’ll note that I actually typed `HETERO:FPGA,CPU`. That’s because, technically, the FPGA is asked to run the core of the neural network (inferencing), but not our entire program. The inferencing engine has a very nice error message to help us understand what we’ve specified that still runs on the CPU:

```bash
./demo_squeezenet_download_convert_run.sh -d FPGA
```

I’ll be told:

```
Graph is not supported on FPGA plugin due to existence of layer (Name:prob, Type: SoftMax) in topology. Most likely you need to use heterogeneous plugin instead of FPGA plugin directly.
```

This simple demo example will run slower on an FPGA because the demo is so brief that the overhead of FPGA setup dominates the runtime. To overcome this, I did the following:

```bash
$ export myDIR=/opt/intel/computer_vision_sdk_fpga_2018.3.343
$ cd $myDIR/deployment_tools/demo/
$ aocl program acl10 $myDIR/a10_dcp_bitstreams/2-0-1_RC_FP11_SqueezeNet.aocx
$ alias csa='~/inference_engine_samples/intel64/Release/classification_sample_async'
$ export myPIC=$IE_INSTALL/demo/car.png
$ csa -m squeezenetl1.1.xml -i $myPIC -d HETERO:FPGA,CPU -ni 100 -nireq 3
$ csa -m squeezenetl1.1.xml -i $myPIC -ni 100 -nireq 3
```
These commands let me avoid the redundant commands in the script, since I know I'll run twice. I manually increased the iteration counts (the \texttt{--ni} parameter) to simulate a more realistic workload that overcomes the FPGA setup costs of a single run. This simulates what I'd expect in a long-running or continuous inferencing situation that would be appropriate with an FPGA-equipped system in a data center.

On my system, the CPU did an impressive 368 frames per second (FPS), but the version that used the FPGA was even more impressive at 850 FPS. I'm told that the FPGA can outstrip the CPU by even more than that for more substantial inferencing workloads, but I'm impressed with this showing. By the way, the CPU that I used was a dual-socket \textit{Intel® Xeon® Silver processor} with eight cores per socket and hyperthreading. Beating such CPU horsepower is fun.

\section*{What Runs on the FPGA? A Bitstream}

What I would call a “program” is usually called a “bitstream” when talking about an FPGA. Therefore, FPGA people will ask, “What bitstream are you running?” The \texttt{demo_squeezenet_download_convert_run.sh} script hid the magic of creating and loading a bitstream. Compiling a bitstream isn’t fast, and loading is pretty fast, but neither needs to happen every time because, once loaded on the FPGA, it remains available for future runs. The \texttt{aocl program acl0...} command that I issued loads the bitstream, which was supplied by Intel for supported neural networks. I didn't technically need to reload it, but I choose to expose that step to ensure the command will work even if I ran other programs on the FPGA in between.

\section*{Wait...Is that All?}

The thing I liked about using the OpenVINO toolkit with an FPGA was that I could easily say, “Hey, when are you going to tell me more?” Let's review what we've covered:

\begin{itemize}
  \item \textbf{If we have a computer vision application}, and we can train it using any popular platform (like Caffe), then we can deploy the trained network with the OpenVINO toolkit on a wide variety of systems.
  \item \textbf{Getting an FPGA working} means installing the right Acceleration Stack, updating board firmware, getting OpenCL installed with the right BSP, and following the OpenVINO toolkit Inference Engine steps to generate use the appropriate FPGA bitstream for our neural net.
  \item \textbf{And then it just works}.
\end{itemize}

Sorry, there's no need to discuss OpenCL or VHDL programming. (You can always read my article on OpenCL programing in \textbf{Issue 31 of The Parallel Universe}.)

For computer vision, the OpenVINO toolkit, with its Inference Engine, lets us leave the coding to FPGA experts—so we can focus on our models.
Inside FPGA Support for the OpenVINO Toolkit

There are two very different under-the-hood things that made the OpenVINO toolkit targeting an FPGA very successful:

- **An abstraction** that spans devices but includes FPGA support
- **Very cool** FPGA support

The abstraction I speak of is Intel’s Model Optimizer and its usage by the Intel Inference Engine. The Model Optimizer is a cross-platform, command-line tool that:

- **Facilitates** the transition between the training and deployment environment
- **Performs** static model analysis
- **Adjusts** deep learning models for optimal execution on end-point target devices

**Figure 3** shows the process of using the Model Optimizer, which starts with a network model trained using a supported framework, and the typical workflow for deploying a trained deep learning model.

The inference engine in our SqueezeNet example simply sends the work to the CPU or the FPGA based on our command. The intermediate representation (IR) that came out of the Model Optimizer can be used by the inferencing engine to process on a variety of devices including CPUs, GPUs, Intel Movidius hardware, and FPGAs. Intel had also done the coding work to create an optimized bitstream for the FPGA that uses the IR to configure itself to handle our network, which brings us to my second under-the-hood item.
The very cool FPGA support is a collection of carefully tuned codes written by FPGA experts. They're collectively called the Deep Learning Accelerator (DLA) for FPGAs, and they form the heart of the FPGA acceleration for the OpenVINO toolkit. Using the DLA gives us software programmability that's close to the efficiency of custom hardware designs, thanks to those expert FPGA programmers who worked hard to handcraft it. (If you want to learn more about the DLA, I recommend the team's paper, "DLA: Compiler and FPGA Overlay for Neural Network Inference Acceleration." They describe their work as “a methodology to achieve software ease-of-use with hardware efficiency by implementing a domain-specific, customizable overlay architecture.”)

Wrapping Up and Where to Learn More

I want to thank the folks at Intel for granting me access to systems with Arria 10 FPGAs cards. This enabled me to evaluate firsthand the ease with which I was able to exploit heterogeneous parallelism and FPGA-based acceleration. I'm a need-for-speed type of programmer—and the FPGA access satisfied my craving for speed without making me use any knowledge of FPGA programming.

I hope you found this walkthrough interesting and useful. And I hope sharing the journey as FPGA capabilities get more and more software support is exciting to you, too.

Here are a few links to help you continue learning and exploring these possibilities:

- OpenVINO toolkit main website (source/github site is here)
- OpenVINO toolkit Inference Engine Developer Guide
- Intel Acceleration Stack can be downloaded from the Intel FPGA Acceleration Hub
- ONNX, an open format to represent deep learning models
- Deep Learning Deployment Toolkit Beta from Intel
- "FPGA Programming with the OpenCL™ Platform," by James Reinders and Tom Hill, The Parallel Universe, Issue 31
- Official OpenCL standards information
- Intel FPGA product information: Intel® Cyclone® 10 LP, Intel® Arria® 10, and Intel® Stratix® 10
Binary floating-point (FP) representations of most real numbers are inexact—and there's an inherent uncertainty in the result of most calculations involving FP numbers. Consequently, computations repeated under different conditions may give different results, although the results remain consistent within the expected uncertainty. This usually isn't concerning, but some contexts demand reproducibility beyond this uncertainty (e.g., for quality assurance, legal issues, or functional safety requirements). However, improved or exact reproducibility typically comes at a cost in performance.
What’s Reproducibility?

Reproducibility means different things to different people. At its most basic, it means rerunning the same executable on the same data using the same processor should always yield the exact same result. This is sometimes called repeatability or run-to-run reproducibility. Users are sometimes surprised—or even shocked—to learn that this isn’t automatic, and that results aren’t necessarily deterministic.

Reproducibility can also mean getting identical results when targeting and/or running on different processor types, building at different optimization levels, or running with different types and degrees of parallelism. This is sometimes called conditional numerical reproducibility. The conditions required for exactly reproducible results depend on the context—and may result in some loss of performance.

Many software tools don’t provide exactly reproducible results by default.

Sources of Variability

The primary source of variations in FP results is optimization. Optimizations can include:

- Targeting specific processors and instruction sets at either build- or run-time
- Various forms of parallelism

On modern processors, the performance benefits are so great that users can rarely afford not to optimize a large application. Differences in accuracy can result from:

- Different approximations to math functions or operations such as division
- The accuracy with which intermediate results are calculated and stored
- Denormalized (very small) results being treated as zero
- The use of special instructions such as fused multiply-add (FMA) instructions

Special instructions are typically more accurate than the separate multiply and add instructions they replace, but the consequence is still that the final result may change.

FMA generation is an optimization that may occur at O1 and above for instruction set targets of Intel® Advanced Vector Extensions 2 (Intel® AVX2) and higher. It’s not covered by language standards, so the compiler may optimize differently in different contexts (e.g., for different processor targets, even when both targets support FMA instructions).
Probably the most important source of variability, especially for parallel applications, is variations in the order of operations. Although different orderings may be mathematically equivalent, in finite precision arithmetic, the rounding errors change and accumulate differently. A different result doesn’t necessarily mean a less accurate one, though users sometimes consider the unoptimized result as correct.

Examples are transformations such as Figure 1, which the compiler may make to improve performance.

\[
(x[i] + y) + z \rightarrow x[i] + (y + z);
\]
\[
a*b + a*c \rightarrow a*(b+c)
\]

The optimizations we’ve considered so far impact sequential and parallel applications similarly. For compiled code, they can be controlled or suppressed by compiler options.

**Reductions**

Reductions are a particularly important example showing how results depend on the order of FP operations. We take summation as an example, but the discussion also applies to other reductions such as product, maximum, and minimum. Parallel implementations of summations break these down into partial sums, one per thread (e.g., for OpenMP*), per process (e.g., for MPI), or per SIMD lane (for vectorization). All of these partial sums can then be safely incremented in parallel. Figure 2 shows an example.

```
// original sequential version

float Sum(const float A[], int n) {
    float sum=0.; // total sum
    float part_sum[nt]; // 1 partial sum per thread
    for (int it=0; it<nt; it++) part_sum[it]=0.;
    for (int it=0; it<nt; it++) { // loop over threads or ranks
        for (int j=it*n/nt; j<(it+1)*n/nt; j++)
            part_sum[it] += A[j]; // partial sum within a thread
    }
    for (int it=0; it<nt; it++) sum += part_sum[it];
    return sum; // parallel version
}
```

For more complete information about compiler optimizations, see our Optimization Notice.
Note that the order in which the elements of A are added—and hence the rounding of intermediate results to machine precision—is very different in the two cases. If there are big cancellations between positive and negative terms, the impact on the final result can be surprisingly large. While users tend to consider the first, serial version to be the “correct” result, the parallel version with multiple partial sums tends to reduce the accumulation of rounding errors and give a result closer to what we’d see with infinite precision, especially for large numbers of elements. The parallel version also runs much faster.

Can Reductions be Reproducible?
For reductions to be reproducible, the composition of the partial sums must not change. For vectorization, that means the vector length must not change. For OpenMP, it means that the number of threads must be constant. For Intel® MPI Library, it means the number of ranks must not change. Also, the partial sums must be added together in the same, fixed order. This happens automatically for vectorization.

For OpenMP threading, the standard allows partial sums to be combined in any order. In Intel’s implementation, the default is first come, first served for low numbers of threads (less than four for Intel® Xeon processors, less than eight on Intel® Xeon Phi™ processors). To ensure that the partial sums are added in a fixed order, you should set the environment variable KMP_DETERMINISTIC_REDUCTION=true and use static scheduling (the default scheduling protocol).

Intel® Threading Building Blocks (TBB) uses dynamic scheduling, so the parallel_reduce() method does not produce run-to-run reproducible results. However, an alternative method, parallel_deterministic_reduce(), is supported. This creates fixed tasks to compute partial sums and then a fixed, ordered tree for combining them. The dynamic scheduler can then schedule the tasks as it sees fit, provided it respects the dependencies between them. This not only yields reproducible results from run to run in the same environment, it ensures the results remain reproducible, even when the number of worker threads is varied. (The OpenMP standard doesn't provide for an analogous reduction operation based on a fixed tree, but one can be written by making use of OpenMP tasks and dependencies.)

For Intel MPI Library, we can optimize the order in which partial results are combined according to how MPI ranks are distributed among processor nodes. The only other way to get reproducible results is to choose from a restricted set of reduction algorithms that are topology unaware (see below).

In all the examples we've looked at, the result of a parallel or vectorized reduction will normally be different from the sequential result. If that's not acceptable, the reduction must be performed by a single thread, process, or SIMD lane. For the latter, that means compiling with /fp:precise (Windows*) or -fp-model precise (Linux* or macOS*) to ensure that reduction loops are not automatically vectorized.
The Intel® Compiler

The high-level option /fp:consistent (Windows) or -fp-model-consistent (Linux and macOS) is recommended for best reproducibility between runs, between different optimization levels, and between different processor types of the same architecture. It's equivalent to the set of options /Qfma-(-no-fma) to disable FMA generation, /Qimf-arch-consistency:true(-fimf-arch-consistency=true), to limit math functions to implementations that give the same result on all processor types, and /fp:precise(-fp-model-precise) to disable other compiler optimizations that might cause variations in results.

This reproducibility comes at some cost in performance. How much is application-dependent, but performance loss of about 10% is common. The impact is typically greatest for compute-intensive applications with many vectorizable loops containing floating-point reductions or calls to transcendental math functions. It can sometimes be mitigated by adding the option /Qimf-use-svml(-fimf-use-svml), which causes the short vector math library to be used for scalar calls to math functions as well as for vector calls, ensuring consistency and re-enabling automatic vectorization of loops containing math functions.

The default option /fp:fast(-fp-model fast) allows the compiler to optimize without regard for reproducibility. If the only requirement is repeatability—that repeated runs of the same executable on the same processor with the same data yield the same result—it may be sufficient to recompile with /Qopt-dynamic-align-(-qno-opt-dynamic-align). This disables only the generation of peel loops that test for data alignment at run-time and has far less impact on performance than the /fp(-fp-model) options discussed above.

Reproducibility between Different Compilers and Operating Systems

Reproducibility is constrained between different compilers and operating systems by the lack of generally accepted requirements for the results of most math functions. Adhering to an eventual standard for math functions (e.g., one that required exact rounding) would improve consistency, but at a significant cost in performance.

There's currently no systematic testing of the reproducibility of results for code targeting different operating systems such as Windows and Linux. The options /Qimf-use-svml and -fimf-use-svml address certain known sources of differences related to vectorization of loops containing math functions and are recommended for improving consistency between floating-point results on both Windows and Linux.
There’s no way to ensure consistency between application builds that use different major versions of the Intel® Compiler. Improved implementations of math library functions may lead to results that are more accurate but different from previous implementations, though the /Qimf-precision:high (-fimf-precision=high) option may reduce any such differences. Likewise, there’s no way to ensure reproducibility between builds using the Intel Compiler and builds using compilers from other vendors. Using options such as /fp:consistent (-fp-model consistent) and the equivalent for other compilers can help to reduce differences resulting from compiled code. So may using the same math runtime library with both compilers, where possible.

**Intel® Math Kernel Library**

Intel® Math Kernel Library (Intel®MKL) contains highly optimized functions for linear algebra, fast Fourier transforms, sparse solvers, statistical analyses, and other domains that may be vectorized and threaded internally using OpenMP or TBB. By default, repeated runs on the same processor might not give identical results due to variations in the order of operations within an optimized function. Intel MKL functions detect the processor on which they are running and execute a code path that’s optimized for that processor—so repeated runs on different processors may yield different results. To overcome this, Intel MKL has implemented conditional numerical reproducibility. The conditions are:

- Use the version of Intel MKL layered on OpenMP, not on TBB
- Keep the number of threads constant
- Use static scheduling (OMP_SCHEDULE=static, the default)
- Disable dynamic adjustment of the number of active threads (OMP_DYNAMIC=false and MKL_DYNAMIC=false, the default)
- Use the same operating system and architecture (e.g., Intel 64 Linux)
- Use the same microarchitecture or specify a minimum microarchitecture

The minimum microarchitecture may be specified by a function or subroutine call (e.g., mkl_cbwr_set (MKL_CBWR_AVX) or by setting a run-time environment variable (e.g., MKL_CBWR_BRANCH=MKL_CBWR_AVX). This leads to consistent results on any Intel® processor that supports Intel® AVX or later instruction sets such as Intel® AVX2 or Intel® AVX-512, though at a potential cost in performance on processors that support the more advanced instruction sets. The argument MKL_CBWR_COMPATIBLE would lead to consistent results on any Intel or compatible non-Intel processor of the same architecture. The argument MKL_CBWR_AUTO causes the code path corresponding to the processor detected at runtime to be taken. It ensures that repeated runs on that processor yield the same result, though results on other processor types may differ. If the runtime processor doesn’t support the specified minimum microarchitecture, the executable still runs but takes the code path corresponding to the actual run-time microarchitecture, as if MKL_CBWR_AUTO had been specified. Results may differ from those obtained on other processors, without warning.
The impact on performance from limiting the instruction set can sometimes be substantial for compute-intensive Intel MKL functions. Table 1 shows the relative slowdown of a DGEMM matrix-matrix multiply on an Intel® Xeon® Scalable processor for different choices of the minimum microarchitecture.

Table 1. Effect of instruction set architecture (ISA) on DGEMM running on an Intel Xeon Scalable processor

<table>
<thead>
<tr>
<th>Targeted ISA</th>
<th>Estimated Relative Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>MKL_CBWR_AUTO</td>
<td>1.0</td>
</tr>
<tr>
<td>MKL_CBWR_AVX512</td>
<td>1.0</td>
</tr>
<tr>
<td>MKL_CBWR_AVX2</td>
<td>0.50</td>
</tr>
<tr>
<td>MKL_CBWR_AVX</td>
<td>0.27</td>
</tr>
<tr>
<td>MKL_CBWR_COMPATIBLE</td>
<td>0.12</td>
</tr>
</tbody>
</table>

Performance results are based on testing by Intel as of Sept. 6, 2018 and may not reflect all publicly available security updates. See configuration disclosure for details. No product can be absolutely secure. Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. Configuration: Instruction set architecture (ISA) on DGEMM running on an Intel Xeon Scalable processor. For more complete information visit www.intel.com/benchmarks.

Intel® MPI Library

Results using Intel MPI Library are reproducible provided that:

- **Compiled code and library calls** respect the reproducibility conditions for the compiler and libraries
- **Nothing in the MPI and cluster environment changes**, including the number of ranks and the processor type

As usual, collective operations like summations and other reductions are the most sensitive to small changes in the environment. Many implementations of collective operations are optimized according to how the MPI ranks are distributed over cluster nodes, which can lead to changed orders of operations and variations in results. Intel MPI Library supports conditional numerical reproducibility in the sense that an application will get reproducible results for the same binary, even when the distribution of ranks over nodes varies. This requires selecting an algorithm that's topology unaware—that is, one that doesn't optimize according to the distribution of ranks over nodes, using the `I_MPI_ADJUST_` family of environment variables:

- `I_MPI_ADJUST_ALLREDUCE`
- `I_MPI_ADJUST_REDUCE`,
- `I_MPI_ADJUST_REDUCE_SCATTER`
- `I_MPI_ADJUST_SCAN`
- `I_MPI_ADJUST_EXSCAN`
- And others

For example, Intel MPI Library Developer Reference documents 11 different implementations of `MPI_REDUCE()`, of which the first seven are listed in Table 2.
Table 2. Comparison of results from `MPI_REDUCE()` for different rank distributions

<table>
<thead>
<tr>
<th>Rank Distribution Algorithm</th>
<th>0246 node1 1357 node2</th>
<th>All node1 2367 node2</th>
<th>0145 node1 2367 node2</th>
<th>0123 node1 4567 node2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Shumilin’s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 Binomial</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 Topology-aware Shumilin’s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 Topology-aware binomial</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 Rabenseifner’s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 Topology-aware Rabenseifner’s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 Knomial</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Two nodes of Intel® Core™ i5-4670T processors at 2.30 GHz, 4 cores and 8 GB memory each, one running Red Hat* EL 6.5, the other running Ubuntu* 16.04. The sample code from Intel® MPI Library Conditional Reproducibility in The Parallel Universe, Issue 21 was used (see references below).

Table 2 compares results from a sample program for a selection of implementations of `MPI_REDUCE()` and for four different distributions of eight MPI ranks over two cluster nodes. The five colors correspond to five different results that were observed. The differences in results are very small—close to the limit of precision—but small differences can sometimes get amplified by cancellations in a larger computation. The topology-independent implementations gave the same result, no matter what the distribution of ranks over nodes, whereas the topology-aware implementations did not. The default implementation of `MPI_REDUCE` (not shown) is a blend of algorithms that depend on workload as well as topology. It also gives results that vary with the distribution of ranks over nodes.
Bottom Line

Intel® Software Development Tools provide methods for obtaining reproducible FP results under clearly-defined conditions.

References

1. "Consistency of Floating-Point Results using the Intel® Compiler"
3. "Intel® MPI Library Conditional Reproducibility," The Parallel Universe, issue 21

BLOG HIGHLIGHTS

Bridging the Gap between Domain Experts and Tuning Experts
HENRY A. GABB, SENIOR PRINCIPAL ENGINEER, INTEL CORPORATION

In 2006, the University of California at Berkeley Parallel Computing Laboratory suggested that widespread adoption of parallel processing required greater separation of concerns between domain experts and tuning experts. Separation of concerns is a spectrum (Figure 1). On the one hand, you have users who are just trying to solve a problem. They can be from any field and their formal computing training varies. They really just want to do the least amount of coding required to get an answer so that they can move on to the larger task that they’re trying to complete, whether it’s a business decision, research article, engineering design, etc. Code tuning is only considered when the performance bottleneck prevents them from reaching this goal. At the other extreme are tuning experts (often referred to internally as ninja programmers) intent on squeezing every ounce of performance from a piece of code whose role in the larger application is unimportant.

Read more >
SPEED SYSTEM AND IOT APPS

See What’s New in Intel® System Studio 2019

Shorten the development cycle and go to market faster with new power and performance tools.

Get advanced cloud connectors and access to 400+ sensors.

Quickly identify performance bottlenecks, reduce power use, and more.

FREE DOWNLOAD
https://intel.ly/2OfRMFg

*Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit intel.com/performance.


For more complete information about compiler optimizations, see our Optimization Notice at software.intel.com/articles/optimization-notice#opt-en.

Intel and the Intel logo are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

© Intel Corporation
Development of C++ is community driven, which allows developers to address issues with the framework in a very detailed way. However, since its implementation is open-source, third parties can develop alternative implementations of C++ libraries, and possibly make their implementations proprietary (e.g., Intel® C++ Compiler). This leads to multiple implementations of the same components—and confusion about which are best for your needs.
To help bring clarity, we compared a few memory allocation libraries, including the allocators in *Threading Building Blocks (TBB)*. Memory allocation is an integral part of programming—and different allocators can affect application performance.

For this study, we chose two benchmarks: **OMNeT++* and **Xalan-C++*. Our tests show significant differences in execution time for these benchmarks when using different memory allocators.

**OMNeT++***

OMNeT++ is an object-oriented, modular, discrete event network simulation framework. It has a generic architecture, so it can be used in various problem domains such as:

- **Modeling** queuing networks
- **Validating** hardware architectures
- **General modeling and simulation** of any system where the discrete event approach is suitable

The benchmark we evaluated performs a discrete event simulation of a large 10 gigabit Ethernet* network.

**Xalan-C++***

Xalan is an XSLT processor for transforming XML documents into HTML, text, or other XML document types. Xalan-C++ version 1.10 is a robust implementation of the W3C Recommendations for XSL Transformations* (XSLT*) and the XML Path Language* (XPath*). It works with a compatible release of the Xerces-C++ XML* parser, Xerces-C++* version 3.0.1. The benchmark program we evaluated is a modified version of Xalan-C++, an XSLT processor written in a portable subset of C++.

**Performance Data**

**Figure 1** shows that the performance of the two benchmarks can vary significantly when using different memory allocation libraries. To understand this better, we studied the implementation differences (**Table 1**).
1 Results using dual-socket Intel® Xeon® Gold 6148 processor with 192GB DDR4 memory, Red Hat® Enterprise Linux® 7.3 OS

Table 1. Brief comparison of memory allocation libraries

<table>
<thead>
<tr>
<th></th>
<th>Malloc</th>
<th>JEMalloc</th>
<th>TBBMalloc</th>
<th>TCMalloc</th>
<th>SmartHeap</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Memory Tracking</strong></td>
<td>Allocates memory from free store</td>
<td>Uses arenas</td>
<td>Uses linked lists</td>
<td>Uses linked lists</td>
<td>Uses linked lists</td>
</tr>
<tr>
<td><strong>Speedup Features</strong></td>
<td>Binning based on allocation size</td>
<td>Thread-specific caching</td>
<td>Binning based on allocation size</td>
<td>Thread-local and central cache</td>
<td>Additional headers in metadata</td>
</tr>
</tbody>
</table>
Memory Allocation Libraries

We'll only discuss the implementation of `operator new` and `operator new []` (Figure 2) because we found the impact of other methods to be negligible.

```c++
void* operator new ( std::size_t count );
void* operator new[]( std::size_t count );
```

### Syntax of the operators

The `new` operator is used to allocate storage required for a single object. The standard library implementation allocates count bytes from free store. In case of failure, the standard library implementation calls the function pointer returned by `std::get_new_handler` and repeats allocation attempts until the `new_handler` function does not return or becomes a null pointer, at which time it throws `std::bad_alloc`. This function is required to return a pointer suitably aligned to hold an object of any fundamental alignment.

The `new[]` operator is the array form of `new` to allocate storage required for an array of objects. The standard library implementation of `new[]` calls `new`.

**Default malloc**

Implemented in the standard C library, the default `malloc` has the call flow shown in Figure 3.

```c
operator new (std::size_t sz) __GLIBCXX_THROW (std::bad_alloc)
```

```c
void* __libc_malloc(size_t bytes)
```

### Default malloc call flow

For more complete information about compiler optimizations, see our Optimization Notice.
**TBBMalloc**

In TBB, the default `malloc` implementation is overridden. This implementation focuses on improving scalability with respect to parallelism. It maintains a linked list of free chunks in memory, with different free lists for different size classes (similar to the bins in traditional `malloc`). To improve spatial locality, it uses a thread-local cache as much as possible, performing allocations on thread-local free lists. Figure 4 shows the default TBBMalloc call flow.

```
void * operator new(size_t sz) throw (std::bad_alloc)

void * operator new[](size_t sz) throw (std::bad_alloc)

extern "C" void * scalable_malloc(size_t size)

static void * internalPoolMalloc(MemoryPool* memPool, size_t size)

inline FreeObject* Block::allocate()

FreeObject *Block::allocateFromFreeList()
```

**JEMalloc**

Traditionally, allocators have used `sbrk(2)` to obtain memory. This is suboptimal for several reasons, including race conditions, increased fragmentation, and artificial limitations on maximum usable memory. If `sbrk(2)` is supported by the operating system, then the JEMalloc allocator uses both `mmap(2)` and `sbrk(2)`, in that order of preference. Otherwise, only `mmap(2)` is used. The allocator uses multiple arenas to reduce lock contention for threaded programs on multiprocessor systems. This works well
with regard to threading scalability, but incurs some costs. There is a small, fixed, per-arena overhead. Additionally, arenas manage memory completely independently of each other, which means a small, fixed increase in overall memory fragmentation. These overheads are not generally an issue, given the number of arenas normally used. In addition to multiple arenas, this allocator supports thread-specific caching to make it possible to completely avoid synchronization for most allocation requests. Such caching allows very fast allocation in most cases, but it increases memory usage and fragmentation, since a bounded number of objects can remain allocated in each thread cache.

Memory is conceptually broken into extents. Extents are always aligned to multiples of the page size. This alignment makes it possible to quickly find metadata for user objects. User objects are broken into two categories according to size: small and large. Contiguous small objects comprise a slab, which resides within a single extent. Each large object has its own extents backing it.

The definitions for the traditional memory allocation functions have been overridden. Figure 5 shows the function calls for the replacement for `operator new`.

```
newImpl(std::size_t size) noexcept(IsNoExcept)

je_malloc(size_t size)

JEMALLOC_ALWAYS_INLINE int imalloc(static_opts_t *sopts, dynamic_opts_t *dopts)

JEMALLOC_ALWAYS_INLINE int imalloc_body(static_opts_t *sopts, dynamic_opts_t *dopts, tsd_t *tsd)

JEMALLOC_ALWAYS_INLINE void *imalloc_no_sample(static_opts_t *sopts, dynamic_opts_t *dopts, tsd_t *tsd, size_t size, size_t usize, szind_t ind)

JEMALLOC_ALWAYS_INLINE void *imalloc_sample(static_opts_t *sopts, dynamic_opts_t *dopts, tsd_t *tsd, size_t usize, szind_t ind)

ipallocct[tsd_tsdn(tsd), usize, dopts->alignment, dopts->zero, tcache, arena]

ialloczmc(tsd_tsdn(tsd), size, ind, dopts->zero, tcache, false, arena, dopts->slow)

arena_prof_promise(tsd_tsdn(tsd), ret, usize)
```

JEmalloc function calls
Smartheap
Smartheap uses a fixed-size allocator for small objects. For other objects, it only stores the largest free block of a page in the free list. It allocates consecutive objects from the same page to reduce randomness in memory traversal in the free list. It uses bits in block headers to check and merge adjacent free blocks as opposed to traversing the free list after a `free()` call.

TCMalloc
TCMalloc assigns each thread a thread-local cache. Small allocations are satisfied from the thread-local cache. Objects are moved from central data structures into a thread-local cache as needed. Periodic garbage collections are used to migrate memory back from a thread-local cache into the central data structures.

Detailed Performance Analysis Using Intel® VTune™ Amplifier
Figures 6 through 21 show the detailed results of our performance analyses using Intel® VTune™ Amplifier, a tool that provides advanced profiling capabilities with a single, user-friendly interface.

OMNeT++ Performance Results
Figures 6 through 9 show results for `operator new` and `operator new[]`.

6 libc malloc performance analysis

7 TBBMalloc performance analysis

For more complete information about compiler optimizations, see our Optimization Notice.
Our attempt to get an application profile with the TCGmalloc library was unsuccessful. We observed that the group of mov and cmp instructions in the assembly code show the performance difference between the libraries (Figures 10 through 13).
The Parallel Universe

11 TBBMalloc performance

12 JEMalloc performance

13 Smartheap performance
Xalan-C++ Performance Results

Figures 14 through 17 show results for `operator new` and `operator new[]`.

14  Performance impact of libc malloc on Xalan-C++

15  Performance impact of TBBmalloc on Xalan-C++

16  Performance impact of JEMalloc on Xalan-C++
Performance impact of SmartHeap on Xalan-C++

Note that we've only shown the first few contributors to CPU time. There are many more methods in the call stack, but we have ignored their contributions to execution time since they are comparatively very small.

The assembly instruction sequences in Figures 18 through 21 show the performance gains with different memory allocators.

Assembly instructions and performance for libc malloc

Assembly instructions and performance for JEMalloc
Speeding Execution Time with the Right Tools

The performance of the memory allocation libraries we tested shows significant improvement over the default memory allocator. Looking at the hotspots in the assembly code, we can speculate that the improvement is due to faster memory access resulting from the way each allocator handles the memory. The allocators claim to reduce fragmentation, and the results support some of these claims. Detailed analysis of memory access patterns could strengthen these hypotheses. It’s easy to evaluate these memory allocators by just linking the application to the memory allocator library of interest.

References
- OMNeT++ Discrete Event Simulator
- Xalan – C++ Version 1.10
- cppreference.com, operator new, operator new[]
- JEMalloc
- MicroQuill
- TCMalloc
- Intel® C++ Compiler
- Threading Building Blocks
- Intel® VTune™ Amplifier

For more complete information about compiler optimizations, see our Optimization Notice.
LIBXSMM*: AN OPEN-SOURCE-BASED INSPIRATION FOR HARDWARE AND SOFTWARE DEVELOPMENT AT INTEL

Meet the Library that Targets Intel® Architecture for Specialized Dense and Sparse Matrix Operations and Deep Learning Primitives

Hans Pabst, Application Engineer; Greg Henry, Pathfinding Engineer; and Alexander Heinecke, Research Scientist; Intel Corporation

What's LIBXSMM?

LIBXSMM* is an open-source library with two main objectives:

1. **Researching** future hardware and software directions (e.g., codesign)
2. **Accelerating** science and inspiring open-source software development

The library targets Intel® architecture with a focus on Intel® Xeon® processors. Server-class processors are the general optimization target, but due to the microarchitectural commonalities, LIBXSMM also applies to...
desktop CPUs and compatible processors. LIBXSMM embraces open-source development and is compatible with relevant tool chains (e.g., GNU* GCC*, and Clang*). A key innovation is the just-in-time (JIT) code generation. Code specialization (not just at runtime) generally unlocks performance that’s hardly accessible otherwise. LIBXSMM accelerates:

- Small and packed matrix multiplications
- Matrix transpose and copy
- Sparse functionality
- Small convolutions

LIBXSMM is a research code that has informed (since 2015) development of Intel® Math Kernel Library (Intel® MKL) and MKL-DNN*. The library is available as source code and prebuilt packages for Linux* (RPM*- and Debian*-based distributions).

Matrix Multiplication

In this article, we'll focus on small matrix multiplications (SMMs), historically the initial function domain covered. LIBXSMM is binary compatible with the industry-standard interface for General Matrix to Matrix Multiplication (GEMM), and even allows interception of existing GEMM calls. Intercepting calls works for applications that are linked statically or dynamically against a BLAS library (e.g., Intel MKL). No relink is needed for the dynamic case (LD_PRELOAD). In either case, source code can remain unchanged. LIBXSMM’s own API, however, can unlock even higher performance.

Given $C_{m \times n} = \alpha \cdot A_{m \times k} \cdot B_{k \times n} + \beta \cdot C_{m \times n}$, the library falls back to LAPACK/BLAS if $\alpha \neq 1$, $\beta \neq \{1, 0\}$, $\text{TransA} \neq \{\text{'N'}, \text{'n'}\}$, or $\text{TransB} \neq \{\text{'N'}, \text{'n'}\}$. However, SMMs (microkernels) can be leveraged together with transpose and copy kernels so that only the alpha-beta limitation remains. For instance, libxsmm_gemm falls back in this case or if $(M \cdot N \cdot K)^{1/3}$ exceeds an adjustable threshold (Figure 1).
The Parallel Universe

```c
char transa = 'n', transb = 'n';
int m = 25, n = 7, k = 75;
int lda = m, ldb = k, ldc = m;

double a[lda*k], b[ldb*n], c[ldc*n];
double alpha = 1, beta = 1;

libxsmm_dgemm(&transa, &transb, &m, &n, &k, 
    &alpha, a, &lda, b, &ldb, 
    &beta, c, &ldc);
```

1. **With libxsmm_dgemm, SMMs are handled according to a configurable threshold (initialization of the matrices is omitted for brevity)**

The BLAS dependency is resolved when linking the application, not at link-time of LIBXSMM (i.e., Intel MKL, OpenBLAS, etc.). BLAS can be substituted with dummy functions if no fallback is needed (libxsmmnoblas).

**Code Dispatch**

Calling `libxsmm_dgemm` (or `dgemm`) automatically dispatches JIT-generated code. LIBXSMM’s powerful API can customize beyond just a threshold, and generates or queries code, which results in a normal function pointer (C/C++) or PROCEDURE POINTER (Fortran*) that’s valid for the lifetime of the application (static or SAVE) *(Figure 2)*.

```c
int flags = LIBXSMM_GEMM_FLAG_NONE;
libxsmm_dmmfunction fun = libxsmm_dmmdispatch(
    m, n, k, &lda, &ldb, &ldc, &alpha, &beta, 
    &flags, NULL/*prefetch*/);

fun(a, b, c);
```

2. **Manually dispatched SMMs that perform the same multiplication as Figure 1**
NULL, in case of a pointer argument of `libxsmm_dmmdispatch`, denotes a default value. For LDx, it means a tight leading dimension. For alpha, beta, or flags, a default is derived from the compile-time configuration.

**C/C++ or Fortran?**

We've only looked at C code so far, but there are C++-specific elements for a more natural user syntax (i.e., overloaded function names to avoid type prefixes, and function templates to deduce an overloaded function suitable for a type). Further, if the C interface accepts a NULL argument (to designate an optional argument), the C++ and Fortran interfaces allow omitting such an argument (Figure 3).

```c
libxsmm_mmfunction<double> fun(m, n, k,
    &lda, &ldb, &ldc, &alpha, &beta);

fun(a, b, c);
```

The type template `libxsmm_mmfunction` designates a functor that represents an SMM kernel.

LIBXMM has an explicit Fortran interface (`IMPLICIT NONE`) that can be (pre-)compiled into a compiler-specific MODule or simply included into an application (`libxsmm.f`). This interface requires the Fortran 2003 standard. Fortran 77 can access a subset of this functionality implicitly and without the need for `ISO_C_BINDING` (Figure 4).

```fortran
TYPE (LIBXSMM_DMMFUNCTION) :: fun
CALL libxsmm_dmmdispatch(fun, m, n, k, &
    lda, ldb, ldc, alpha, beta)

CALL libxsmm_dmmcall(fun, a, b, c)
```

TD dispatches code like the previously presented C/C++ examples. Additional generic procedure overloads can omit the type prefix (`libxsmm_mmdispatch` and `libxsmm_mncall`).
LIBXSMM supports header-only usage in C++ and C (the latter is quite unique). The file libxsmm_source.h allows us to get around building the library but gives up a clearly defined application binary interface (ABI). The header file is intentionally named libxsmm_source.h, since it includes the implementation (the src directory). It is common for C++ code to give up on an ABI, and more so since the advent of generic programming (templates). An ABI allows hot-fixing a dynamically linked application after deployment. The header-only form can generally decrease the turnaround during development of an application (longer compilation time).

**Type-Generic API**

All the language interfaces we've presented so far (Fortran 77 is not supported per an interface) are type-safe, with type deduction at compile time. An additional lower-level descriptor or handle-based interface is available for C/C++ and Fortran. This can ease integration with other libraries or enable generic programming (Figure 5).

```c
#include "libxsmm.h"

libxsmm_descriptor_blob blob;
libxsmm_xmmfunction fun;
libxsmm_gemm_descriptor* desc;

desc = libxsmm_gemm_descriptor_init(&blob,
   LIBXSMM_GEMM_PRECISION_F64, m, n, k, lda, ldb, ldc,
   &alpha, &beta, flags, NULL/*prefetch*/);

fun = libxsmm_xmmdispatch(desc);

fun.xmm(a, b, c);
```

5 Low-level code dispatch using a handle-based approach. Note that the descriptor is only forward-declared but can be efficiently created without dynamic memory allocation (blob must be alive for desc to be valid).

Often, it's possible to avoid dispatching a kernel every time it's called. This is trivial if the same SMM is called consecutively (making it an ideal application of the dispatch API). The cost of the dispatch is on the order of tens of nanoseconds. JIT-generating code takes place only the first time a code version is requested, and it takes tens of microseconds. As with all functions in LIBXSMM, code generation and dispatch are thread-safe. It's also convenient to query code, since all GEMM arguments act as a key. Moreover, the JIT-generated code (function pointer) is valid until program termination (like ordinary function pointers referring to static code).
Prefetches

Batched SMMs can use the same kernel for more than one multiplication, and it can be beneficial to supply next locations so that upcoming operands are prefetched ahead of time. For LIBXSMM, such a location would be the address of the next matrix to be multiplied. Of course, next operands aren’t required to be consecutive in memory.

A prefetch strategy (besides `LIBXSMM_PREFETCH_NONE`) changes the kernel signature to accept six arguments (a, b, c and pa, pb, pc) instead of three (a, b, and c). Further, `LIBXSMM_PREFETCH_AUTO` assumes which operands are streamed and a strategy is chosen based on `CPUID` (Figure 6).

```c
int prefetch = LIBXSMM_PREFETCH_AUTO;
libxsmm_dmmfunction fun = libxsmm_dmmdispatch(
    m, n, k, &lda, &ldb, &ldc, &alpha, &beta, 
    &flags, &prefetch);

fun(a, b, c, pa, pb, pc);
```

Kernel that accepts prefetch locations (pa, pb, and pc)

Using a `NULL` pointer for the prefetch strategy refers to no prefetch and is equivalent to `LIBXSMM_PREFETCH_NONE`. By design, the strategy can be adjusted without changing the call site (six valid arguments should be supplied) (Figure 7).
The same kernel (\texttt{dmm}) is used for a series of multiplications that prefetch the next operands. The last multiplication is peeled from the loop to avoid an out-of-bounds prefetch.

Prefetching from an invalid address doesn't trap an exception, but causes a page fault (which we can avoid by peeling the last multiplication of a series out of the loop).

**Domain-Specific Libraries**

There's an extensive history of domain-specific languages and libraries, not only for matrix multiplications, but even for small problem sizes in general or SMMs in particular. Even complete LAPACK/BLAS implementations such as Intel MKL include optimizations for small problem sizes (\texttt{MKL\_DIRECT\_CALL}) and continue to do so (MKL 2019 can JIT-generate SMM kernels). Let's compare some popular open-source C++ libraries (according to Google Trends):

- Armadillo*
- Blaze*
- Eigen*
- uBLAS* (Boost*)

For fairness, the candidates need not only continuous development, but a self-statement about performance (e.g., publication) or an obvious performance orientation (i.e., uBlas is out of scope). The ability to (re-)use an existing memory buffer, along with support for leading dimensions (non-intrusive design) are
important for C++ libraries with data structures (matrix type), even when templates outpaced classic object orientation. LAPACK/BLAS is surprisingly more STL-like, since there are only algorithms (behavior and no state). Essentially, BLAS-like parametrization was made a prerequisite. Finally, our candidate libraries need to provide their own implementations (beyond fallback code), because even perfect mappings to LAPACK/BLAS would not bring any new performance results (i.e., Armadillo is out of scope).

After considering popularity based on C++ as a language and related state-of-the-art techniques (expression templates), design for a domain-specific language, continuous development, code quality, and performance, we finally settled on Blaze and Eigen for evaluation, since:

- **Blaze** combines the elegance and ease of use of a DSL with HPC-grade performance.
- **Eigen** is a versatile, fast, reliable, and elegant library with good compiler support. [Editor's note: This library was featured previously in “Accelerating the Eigen Math Library for Automated Driving Workloads” in *The Parallel Universe, Issue 31.*]

For this article, we’ll use code developed under LIBXSMM’s sample collection (**Figure 8**). A series of matrices are multiplied such that A, B, and C are streamed (i.e., A and B matrices are accumulated into C matrices \[C += A \times B \text{ with } \alpha=1 \text{ and } \beta=1\]). This is in addition to only running a kernel without loading any matrix operands from memory (except for the first iteration).

```cpp
for (i = 0; i < n; ++i) {
    const MatrixType ai = Map(a + i * an);
    const MatrixType bi = Map(b + i * bn);
    MatrixType ci = Map(c + i * cn);

    ci += ai * bi;
}
```

**Pseudocode of the benchmark implemented in Blaze and Eigen.** Existing data is mapped into a variable with the library’s matrix datatype (ideally without any copy). The operands ai, bi, and ci are loaded/stored in every iteration.
Lazy expression evaluation (expression templates) was not a subject of our comparison, so we avoided the canonical (GEMM) expression \( c_i = \alpha \cdot a_i \cdot b_i + \beta \cdot c_i \). Blaze's CustomMatrix adopted user data (including leading dimensions) and even avoided intermediate copies for the canonical GEMM expression. Eigen introduced a copy for taking the result of \( \text{Matrix::Map} \) as \( \text{Matrix(auto} \ \text{was used instead), and had lower performance with leading dimensions due to the dynamic stride and related copies. Both issues have been worked around and are not reflected in the performance results (Figures 9 and 10).}

![Graph showing performance comparison between Eigen, Blaze, and xsmm](image)

**Figure 9** Double-precision SMMs \((M, N, K)\) for the fully streamed case \( C^\top + A^\top \cdot B^\top \) (GNU GCC 8.2, LIBXSMM 1.10, Eigen 3.3.5, Blaze 3.4). Single-threaded using Intel Xeon 8168 processor (2666 MHz DIMMs, HT/Turbo on).
Unrolling the kernel beyond what is needed for SIMD vectors only helps for the cache-hot regime. SMMs are memory- rather than compute-bound when streaming operands: $\text{FLOPS} = 2 \times m \times n \times k$, $\text{BYTES} = 8 \times (m \times k + k \times n + m \times n \times 2)$ in case of streaming $A$, $B$, and $C$ in double-precision plus read for ownership of $C$. Square shapes and ignoring cache-line granularity yields: $\text{AI}(n) = 2 \times n^3 / (32 \times n^2) = 0.0625 \times n$ (with arithmetic intensity in FLOPS per byte). For example, a multiplication of 16x16 matrices using the above-mentioned regime to stream operands only yields 1 FLOP/byte (Figure 11). But a typical system used for scientific computing, such as a dual-socket Intel Xeon processor-based server, can yield 15 FLOPS/byte in double-precision. Therefore, an instruction mix to optimally hide memory operations—thereby exploiting the memory bandwidth—is more beneficial for streamed SMMs.
Real Speedups for Scientific Applications

LIBXSMM has been public since the beginning of 2014 and came up with in-memory JIT code generation for SMMs by the end of 2015. The SMM domain is rather stable (mostly additions to the API) and deployed by major applications. The SMM domain offers fast, specialized code and fast generation (tens of microseconds) and querying of code (tens of nanoseconds). The library is useful for research and supports new instructions as soon as they’re published in the Intel Instruction Set manual. LIBXSMM delivers real speedups for scientific applications in areas where matrix multiplication was only recognized as FLOPS-bound in the past.

Double-precision SMMs \((M, N, K)\) for the cache-hot case \(C += A \times B\) (GNU GCC 8.2, LIBXSMM 1.10, Eigen 3.3.5, Blaze 3.4). Single-threaded using Intel Xeon processor 8168 (2666 MHz DIMMs, HT/Turbo ON).
Learn More

Documentation

- Online: Main and sample documentation with full text search (ReadtheDocs).
- PDFs: Main and separate sample documentation.
- You can find a list of LIBXSMM applications on the GitHub LIBXSMM homepage.

Articles

- "LIBXSMM Brings Deep Learning Lessons Learned to Many HPC Applications" by Rob Farber, 2018.
- "Largest Supercomputer Simulation of Sumatra-Andaman Earthquake" by Linda Barney, 2018.
- SC’18: “Anatomy of High-Performance Deep Learning Convolutions on SIMD Architectures”
- SC’16: "LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation"

BLOG HIGHLIGHTS

Open Source and the Future of Visual Cloud
IMAD SOUSOU

Netflix reported 130 million subscribers this July 1, or more than the populations of Los Angeles, New York, Toronto, Mexico City, London, Shanghai, and Tokyo combined.

As Intel Vice President Lynn Comp noted, the vast majority of network traffic will soon be video. Netflix alone has tens of petabytes of storage on Amazon Web Services (AWS). People are consuming and generating video content at an explosive rate. Business workloads from AR/VR training to graphics rendering increase the need for cloud and edge network transformation. Then there's the vast, global future of cloud gaming. To meet this demand, Intel is announcing new open source software projects servicing the four core building blocks of visual cloud: Render, encode, decode, and inference.
Accurate and fast numerical modeling is essential to investigating astrophysical objects like neutron stars and black holes, which populate every galaxy in our universe and play a key role in our understanding of astrophysical sources of high-energy radiation. It’s particularly important to study how hot plasma accretes onto black holes, since this physical phenomenon can release huge amounts of energy and power some of the most energetic objects in the cosmos (e.g., gamma-ray bursts, active galactic nuclei, and x-ray binaries). The numerical studies that aim to investigate the accretion scenario are computationally expensive—especially when understanding the fundamental physical mechanisms requires exploring multiple parameters. This is why modeling of relativistic plasmas greatly benefits from maximizing computational efficiency.
ECHO-3DHPC* is a Fortran* application that solves the magneto-hydrodynamic equations in general relativity using the 3D finite-difference and high-order reconstruction algorithms (Figure 1). Originally based on the code ECHO¹, its most recent version uses a multidimensional MPI domain decomposition scheme that scales to a large number of cores.

Recent code improvements added thread-level parallelism using OpenMP*, which allows the application to scale beyond the current limit set by MPI communications overhead. With the help of Intel® Software Development Tools like Intel® Fortran Compiler with profile-guided optimization (PGO), Intel® MPI Library, Intel® VTune™ Amplifier, and Intel® Inspector, we investigated the performance issues and improved scalability and time-to-solution.

Using Intel® Fortran Compiler for Performance Optimization

A key (and mostly overlooked) ingredient for performance optimization is better use of compiler features like PGO, which facilitates the developer’s work in reordering code layout to reduce instruction-cache problems, shrink code size, and reduce branch mispredictions. PGO has three steps:

1. **Compile** your application with the `-prof-gen` option, which instruments the binary.
2. **Run** the executable produced in step 1 to generate a dynamic information file.
3. **Recompile** your code with the option `-prof-use` to merge the collected information and generate an optimized executable.

For more complete information about compiler optimizations, see our Optimization Notice.
Performance measurements show an improvement up to 15% on the time-to-solution for the largest run with 16,384 cores (Table 1).

Table 1. Hardware configuration: Dual-socket Intel Xeon E5-2680 v1 processor @ 2.7GHz, 16 cores per node; Software: SLES11, Intel MPI library 2018

<table>
<thead>
<tr>
<th>Grid Size</th>
<th>No. of Cores</th>
<th>Intel® Fortran Compiler 18 (Seconds/Iteration)</th>
<th>Intel Fortran Compiler 18 with PGO (Seconds/Iteration)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$512^3$</td>
<td>8,192</td>
<td>1.27</td>
<td>1.13</td>
</tr>
<tr>
<td>$512^3$</td>
<td>16,384</td>
<td>0.66</td>
<td>0.57</td>
</tr>
<tr>
<td>$1,024^3$</td>
<td>16,384</td>
<td>4.91</td>
<td>4.22</td>
</tr>
</tbody>
</table>

OpenMP Optimization using Intel® VTune Amplifier and Intel® Inspector

ECHO-3D HPC has recently added thread-level parallelism using OpenMP. This makes it possible to reduce the number of MPI tasks by using OpenMP threads within a CPU node, thus eliminating some MPI communication overhead.

Performance optimization is based on single-node tests on a dual-socket Intel® Xeon® processor E5-2697 v3 with a total of 28 cores @ 2.6 GHz. The command for generating the HPC performance characterization analysis with Intel VTune Amplifier is:

```
amplxe-cl -collect hpc-performance -- <executable and arguments>
```

Analyzing the performance of the baseline code shows the following bottlenecks:

- A large fraction of the CPU time is classified as imbalance or serial spinning (Figure 2). This is caused by insufficient concurrency of working threads due to a significant fraction of sequential execution.
- As a consequence of the imbalance, the node-level scalability suffers at higher numbers of threads (Figure 3, red line).
- The Intel VTune Amplifier memory analysis shows that the code is memory-bound, with about 80% of execution pipeline slots stalled because of demand from memory loads and stores (Figure 4).
Screen shot from the Intel VTune Amplifier HPC Performance Optimization, Hotspots viewpoint analysis for the baseline version of ECHO-3DHPC.

Parallel speedup within a node as a function of the number of threads (the last point refers to hyperthreading). The red and orange lines refer to the baseline and optimized code versions. Ideal scaling is indicated by the dashed black line.
Besides the performance considerations above, the correctness of the OpenMP implementation (only recently introduced into the code) is worth investigating with Intel Inspector, an easy-to-use memory and threading debugger. It doesn't require special compilers or build configurations—a normal debug or production build will suffice. However, this kind of analysis can impact application runtime due to instrumentation overhead. The command-line to investigate and correct threading errors is:

```
inspxe-cl -collect=ti3 -- <executable and arguments>
```

The analysis shows a number of data races (Figure 5). These are due to multiple threads simultaneously accessing the same memory location, which can cause non-deterministic behavior.
Based on our findings, we’ve increased the number of OpenMP parallel regions, changed the scheduling protocol to guided, and applied PRIVATE clauses to selected variables in OpenMP parallel loops, thus removing the data races previously detected by Intel Inspector. These optimizations had the following effects:

- **With 28 threads, the optimized code has a performance improvement of 2.3x (Figure 3) over the baseline version, and a parallel speedup of 5.5x.** This is in agreement with the prediction of Amdahl’s Law for a parallel application with 15% sequential execution, as currently measured in the code using Intel Advisor.
- **The spin time decreases by 1.7x.**
- **The code is still memory-bound**, but the fraction of stalled pipeline slots has decreased slightly, to 68%.

### Overcoming the MPI Communication Bottleneck

We can also see the effect of the OpenMP optimization by analyzing large-scale MPI/OpenMP hybrid runs. In **Figure 6**, we compare the scalability of the pure MPI and MPI/OpenMP hybrid codes in terms of time per iteration (lower is better). The figure shows the results for the best hybrid configuration, with four MPI tasks per compute node and seven OpenMP threads per task. With fewer compute nodes, the MPI-only version of the code (blue line) has better performance. The picture changes at larger numbers of nodes, where scalability starts degrading because of MPI communication overhead. The problem size per MPI task is not large enough to compensate for the communication time (communication is dominating the computation). Best performance (the lowest point in the scaling curve) moves to the right in **Figure 6**—
Scalability of the pure MPI (blue line) and the hybrid MPI/OpenMP (green line) versions of ECHO-3DHPC. The dashed lines represent the ideal strong scaling. Each node has 28 cores.

thus enabling more effective use of larger HPC systems. Comparing the runs with best performance, the time spent in MPI communication has decreased by 2x in the MPI/OpenMP hybrid configuration. Additional optimization of the code will allow the use of even more OpenMP threads per node, further relieving the communication overhead.

More Efficient Use of Hardware for Faster Time-to-Solution

Recent developments in the parallelization scheme of ECHO-3DHPC, an astrophysical application used to model relativistic plasmas, can lead to more efficient use of hardware—which translates to faster time-to-solution. The code's new version uses an optimized hybrid MPI/OpenMP parallel algorithm, exhibiting good scaling to more than 65,000 cores.
References


Have you ever wondered how well your system is being utilized throughout a long stretch of application runs? Or whether your system was misconfigured, leading to a performance degradation? Or, most importantly, how to reconfigure it to get the best performance out of your code? State-of-the-art performance analysis tools, which allow users to collect performance data for longer runs, don't always give detailed performance metrics. On the other hand, performance analysis tools suitable for shorter application runs can overwhelm you with a huge amount of data.
This article introduces you to Intel® VTune™ Amplifier's Platform Profiler, which provides data to learn whether there are problems with your system configuration that can lead to low performance, or if there's pressure on specific system components that can cause performance bottlenecks. It analyzes performance from either the system or hardware point of view, and helps you identify under- or over-utilized resources. Platform Profiler uses a progressive disclosure method, so you’re not overwhelmed with information. That means it can run for multiple hours, giving you the freedom to monitor and analyze long-running or always-running workloads in either development or production environments.

You can use Platform Profiler to:

- **Identify** common system configuration problems
- **Analyze** the performance of the underlying platform and find performance bottlenecks

First, the platform configuration charts Platform Profiler provides can help you easily see how the system is configured and identify potential problems with the configuration. Second, you get system performance metrics including:

- CPU and memory utilization
- Memory and socket interconnect bandwidth
- Cycles per instruction
- Cache miss rates
- Type of instructions executed
- Storage device access metrics

These metrics provide system-wide data to help you identify if the system—or a specific platform component such as CPU, memory, storage, or network—is under- or over-utilized, and whether you need to upgrade or reconfigure any of these components to improve overall performance.

**Platform Profiler in Action**

To see it in action, let's look at some analysis results collected during a run of the open-source HPC Challenge (HPCC) benchmark suite and see how it uses our test system. HPCC consists of seven tests to measure performance of:

- Floating-point (FP) execution
- Memory access
- Network communication operations.
Figure 1 shows system configuration view of the machine where we ran our tests. The two-socket machine contained Intel® Xeon® Platinum 8168 processors, with two memory controllers and six memory channels per socket, and two storage devices connected to Socket 0.

Figure 2 shows CPU utilization metrics and the cycles per Instruction (CPI) metric, which measures how much work the CPUs are performing. Figure 3 shows memory, socket interconnect, and I/O bandwidth metrics. Figure 4 shows the ratio of load, store, branch, and FP instructions being used per core. Figures 5 and 6 show memory bandwidth and latency chart for each memory channel. Figure 7 shows a rate of branch and FP instructions over all instructions. Figure 8 shows L1 and L2 cache miss rate per instruction. Figure 9 shows memory consumption chart. On average, only 51% of memory was consumed throughout the run. A larger test case can be run to increase memory consumption.

In Figures 5 and 6, we see that only two channels instead of six are being used. This clearly shows that there's a problem with the memory DIMM configuration on our test system that's preventing us from making full usage of memory channel capacity—leading to a performance degradation of HPCC.

The CPI (Figure 2), DDR memory bandwidth utilization, and instruction mix metrics in the figures show which specific type of test—either compute or FP operation- or memory-based—is being executed at a specific time during the HPCC run. For example, we can see that during 80-130 and 200-260 seconds of the run, both the memory bandwidth utilization and CPI rate increase—confirming that a memory-based test inside HPCC was executed during that period of time. Moreover, the Instruction Mix chart in Figure 7 shows that between 280-410 seconds, threads execute FP instructions in addition to some memory access operations during 275-360 seconds (Figure 3). This observation leads us to the idea that a test with a mixture of both compute and memory operations is executed during this period. Another observation is that we may be able to improve the performance of the compute part in this test by optimizing the execution of FP operations using code vectorization.
System Configuration View

CPU Metrics

CPU Utilization

- System
- Socket 0
- Socket 1

CPU Utilization in Kernel Mode

- Average: 47.41%
- Average: 47.51%
- Average: 47.30%

CPU Frequency

- Average: 3.16 GHz
- Average: 3.17 GHz
- Average: 3.15 GHz

CPI

- Average: 15.54
- Average: 13.50
- Average: 17.51

CPU utilization metrics
Throughput Metrics

3 Throughput metrics for memory, UPI and I/O

Operations Metrics

4 Types of instructions used in throughout program execution
Memory Throughput

Memory bandwidth chart at a memory channel level

Memory Latency

Memory latency chart at a memory channel level
Instruction Mix

Rate of branch and floating point instructions over all instructions

L1 and L2 Miss per Instruction

L1 and L2 miss rate per instruction
Memory Utilization

Memory consumption

HPCC doesn’t perform any tests that include I/O, so we’ll show Platform Profiler results specifically on disk access from a second test case, LS-Dyna, a proprietary multiphysics simulation software developed by LSTC. Figure 10 shows disk I/O throughput for LS-Dyna. Figure 11 shows I/O per second (IOPS) and latency metrics for LS-Dyna application. The LS-Dyna implicit model periodically flushes the data to the disk, so we see periodic spikes in the I/O throughput chart (see read/write throughput in Figure 10). Since the amount of data to be written isn’t large, the I/O latency remains consistent during the whole run (see read/write latency in Figure 11).

Read/Write Throughput

Disk I/O throughput for LS-Dyna
Understanding System Performance

In this article, we presented Platform Profiler, a tool that analyzes performance from the system or hardware point of view. It provides insights into where the system is bottlenecked and identifies whether there are any over- or under-utilized subsystems and platform-level imbalances. We also showed its usage and the results collected from the HPCC benchmark suite and the LS-Dyna application. Using the tool, we found that poor memory DIMM placement was limiting memory bandwidth. Also, we found a part of the test had a high FP execution, which we could optimize for better performance using code vectorization. Overall, we found that this specific test case for HPCC and LS-Dyna doesn't put any pressure on our test system, and there's more room for system resources—meaning we can run an even larger test case next time.