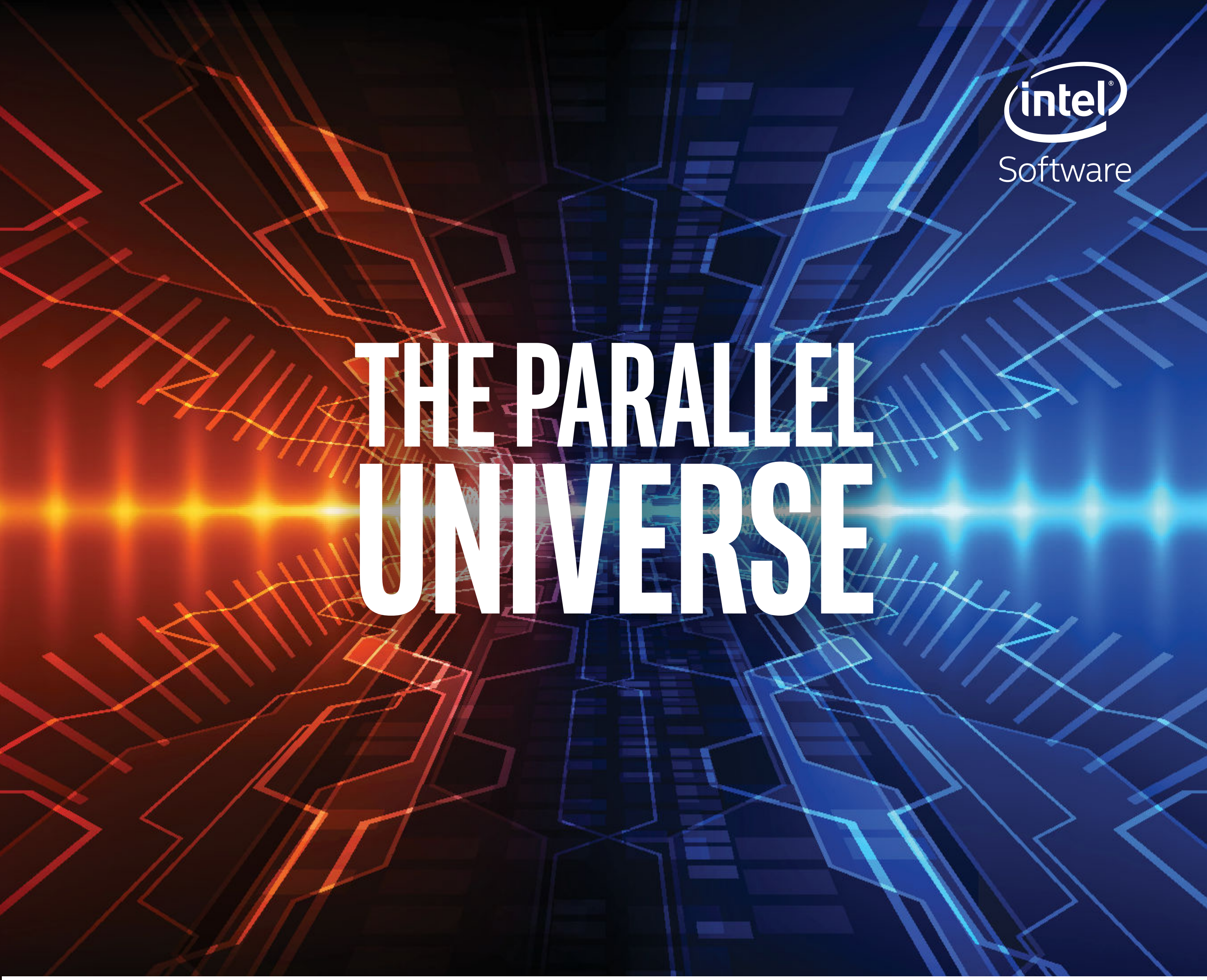


# THE PARALLEL UNIVERSE

## Intel® Rendering Framework Using Software-Defined Visualization

Unifying AI, Analytics, and HPC on a Single Cluster

Advancing OpenCL™ for FPGAs

Issue 35 2019

# CONTENTS

# 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.

## Happy New Year...and May 2019 Bring You High Performance

Welcome to our first issue of 2019. I didn't make any bold predictions at the start of 2018—just that the parallel computing future is heterogeneous. However, this trend was already well underway, and will continue to gain momentum this year. It wasn't exactly a bold prediction, and I won't make any bold predictions this year either. I'll just call out a few trends I'm watching.

The open source community initiative on software-defined visualization (**SDVis.org**) continues to demonstrate that the CPU is better for large-scale rendering than GPU-based solutions, which suffer from memory limitations and high cost. This is the topic of our feature article, **Intel® Rendering Framework Using Software-Defined Visualization**. The advantage of SDVis isn't news to the film industry, which has been doing CPU-based rendering for many years, but SDVis is spreading to other computational domains where visualization of ever-larger datasets is needed.

This brings us to another trend I'm watching closely: **"The Convergence of HPC, BDA, and AI in Future Workflows"** (a talk I gave recently at the **2018 New York Scientific Data Summit at Brookhaven National Laboratory**). Trish Damkroger, Intel's vice president and general manager of Extreme Computing, published a similar viewpoint recently on Top500.org: **The Intersection of AI, HPC, and HPDA: How Next-Generation Workflows Will Drive Tomorrow's Breakthroughs**. The line between traditional high-performance computing, artificial intelligence, and big data analytics is blurring, so I asked the Intel Data Center Group to provide a guest commentary: **Unifying AI, Analytics, and HPC on a Single Cluster**.

As I've said before, heterogeneous parallelism is the future, and FPGAs are getting attention as an offload device for software acceleration. James Reinders, our editor emeritus, published several articles last year on programming FPGAs. In this issue, Professor Martin Herbordt from Boston University shares some of his best practices for OpenCL programming on FPGAs. In **Advancing OpenCL™ for FPGAs**, he walks us through the optimization of some common numerical algorithms.

We round out this issue with three articles on code optimization: **Parallelism in Python***, **Remove Memory Bottlenecks Using Intel® Advisor**, and **MPI-3 Non-Blocking I/O Collectives in Intel® MPI Library**.

Future issues of *The Parallel Universe* will feature articles on using just-in-time compilation to optimize Python code, new features in Intel® Software Development Tools, performance case studies, and much more. Be sure to **subscribe** so you won't miss a thing.

Also, 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, and systems and IoT development.

**Henry A. Gabb**
January 2019

# INTEL® RENDERING FRAMEWORK USING SOFTWARE-DEFINED VISUALIZATION

## Why Intel® Xeon® Processors Excel at Visualization

*Rob Farber, Global Technology Consultant, TechEnablement*

Software-defined visualization (SDVis) is akin to software-defined networking, software-defined infrastructure, and other initiatives Intel is taking to maximize the benefits—and inherent performance—of modern **Intel® Xeon® processors** with software that takes advantage of high thread count and data parallelism. The performance is there, and the advantages over dedicated devices with limited available memory are manifold—including the ability to use ever-improving advanced algorithms that exploit the:

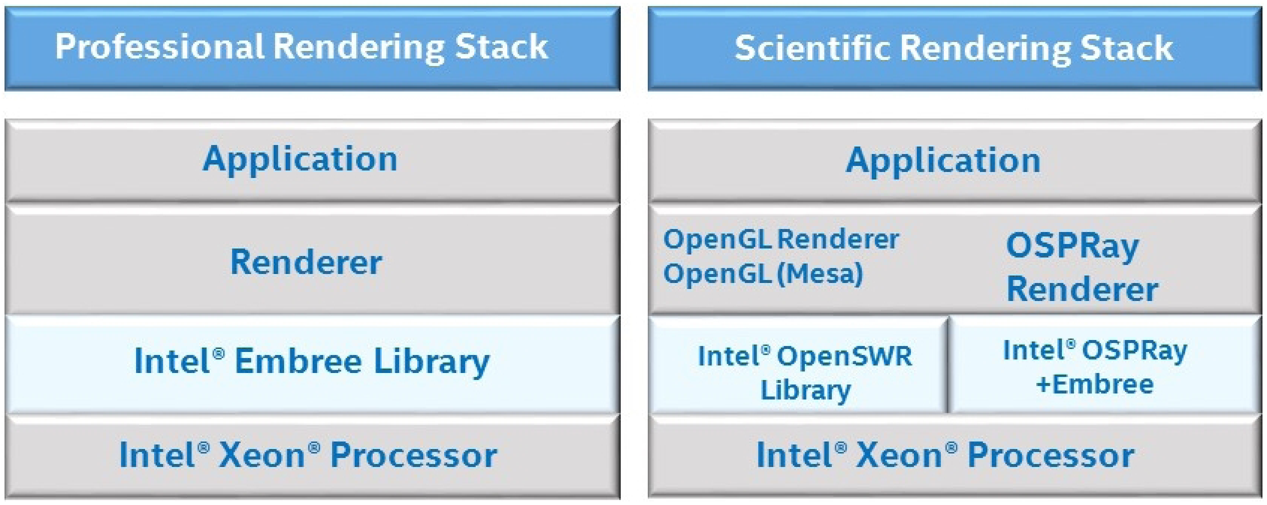
- **Larger memory capacity** of the processor
- **Flexibility and easy upgradability** of software versus hardware replacement
- **Overall cost savings** during procurement and improved total cost of ownership over the lifespan of the hardware

## Performance and Scalability that have Redefined Visualization

Jim Jeffers, senior director and senior principal engineer for Intel's visualization solutions, notes, "With the **Intel® Rendering Framework**, all the work is being done on the CPU, while users are getting the same—or better—experience than with today's dedicated graphics hardware." The Intel Rendering Framework provides both scalable and interactive ray tracing and OpenGL* visualization via the **Intel® Embree**, **Intel® OSPRay**, and **Intel® OpenSWR** libraries. Plus, the Intel Rendering Framework now includes the new **Intel® Open Image Denoise library**.

Not surprisingly, modern high-throughput processor cores packaged in multi- and many-core processors can execute many tasks interactively, and with performance unequaled by earlier generations of processors. Jeffers points out that "Benchmarks show a 100x increase in rendering performance compared to what was available in 2016 when rendering OpenGL triangle-based images with Mesa."*

This level of performance has redefined scientific visualization and is making significant inroads into the cinematic and professional visualization market segments (**Figure 1**). Jeffers points out that with its ability to exploit the available CPU memory (commonly 192 GB or more for a processor versus 16 GB for a high-end GPU), the Intel Rendering Framework can deliver the same or better performance with fidelity that a GPU can't match. That, coupled with the ability to run and visualize anywhere, regardless of the scale of the visualization task and without requiring specialized hardware for interactive response, is the reason high-performance computing (HPC) centers no longer need to procure GPUs for visualization clusters.



**1 The Intel® Rendering Framework with SDVis technology supports rendering on platforms of all sizes including cloud and HPC clusters.**

## The Primary HPC Visual Analysis Approach for Many

Five years ago, you never would have heard an HPC user say, “I prefer rendering my images on CPUs.” However, that mindset changed as CPU-based interactive and photorealistic rendering supplanted GPUs in many HPC centers. Paul Navrátil, director of visualization at the Texas Advanced Computing Center (TACC), highlights TACC’s commitment by pointing out that “CPU-based SDVis will be our primary visual analysis mode on Frontera*, leveraging the Intel Rendering Framework stack.” Frontera is expected to be the fastest academic supercomputer in the U.S. when it becomes operational in 2019.

In a word, the scalability is “outstanding” as demonstrated by a 1.1 trillion triangle OpenGL hero benchmark by Kitware[1] on the Trinity* supercomputer at Los Alamos National Laboratory. However, it doesn’t take a supercomputer to run SDVis. The integration of Intel Rendering Framework components such as OSPRay into Paraview makes exploring the benefits of ray tracing easy on most hardware platforms. David DeMarle, principal engineer at Kitware, notes that with the Intel Rendering Framework, “A one-line change is all that is required for VTK* and ParaView* users to switch between OSPRay ray tracing and OpenGL rendering.”

## Traditional Batch and New In Situ and In-Transit Visualization Workflows

The software-defined nature of the Intel Rendering Framework means that scientists can now perform *in situ* rendering, where visualization occurs using the same nodes as the computation. *In situ* visualization has been identified as a key technology to enable science at the exascale.[2] Jeffers points out, “As we move to exascale, we have to manage exabytes of data. While the data can be computed, the I/O systems aren’t getting there to move the data. Hence, *in situ*. Otherwise, it can take days, weeks, or months to visualize.” He likes to summarize this by stating, “A picture is worth an exabyte.”

## A Path to Exascale Visualization

As part of a U.S. Department of Energy (DOE) multi-institutional effort, and in collaboration with private companies and other national labs, Argonne National Laboratory is working to leverage the **SENSEI*** framework to help people prepare for the arrival of Aurora*, a new Intel-Cray system. Aurora will be capable of delivering more than an exaflop of floating-point performance. SENSEI is one example of a portable framework that enables *in situ*, in-transit, and traditional batch visualization workflows for analysis and scalable interactive rendering of the huge data volumes generated when using an exascale supercomputer.

Depending on the application, researchers sometimes may prefer to dedicate more supercomputer nodes to a computationally expensive simulation, while using a smaller number of nodes for rendering. This asymmetric load balancing is called in-transit visualization. Unlike *in situ* visualization that renders data in place on the node, in-transit visualization does incur some overhead as data must be moved across the communications fabric between nodes. The payoff is the additional compute power that can be dedicated to the simulation. Both in-transit and *in situ* workflows keep the data in memory and avoid writing to storage. Joseph Insley, Visualization and Analysis Team lead at the Argonne Leadership Computing Facility, points out, “With SENSEI, users can utilize *in situ* and in-transit techniques to address the widening gap between flop/s and I/O capacity, which is making full-resolution, I/O-intensive post hoc analysis prohibitively expensive, if not impossible.”

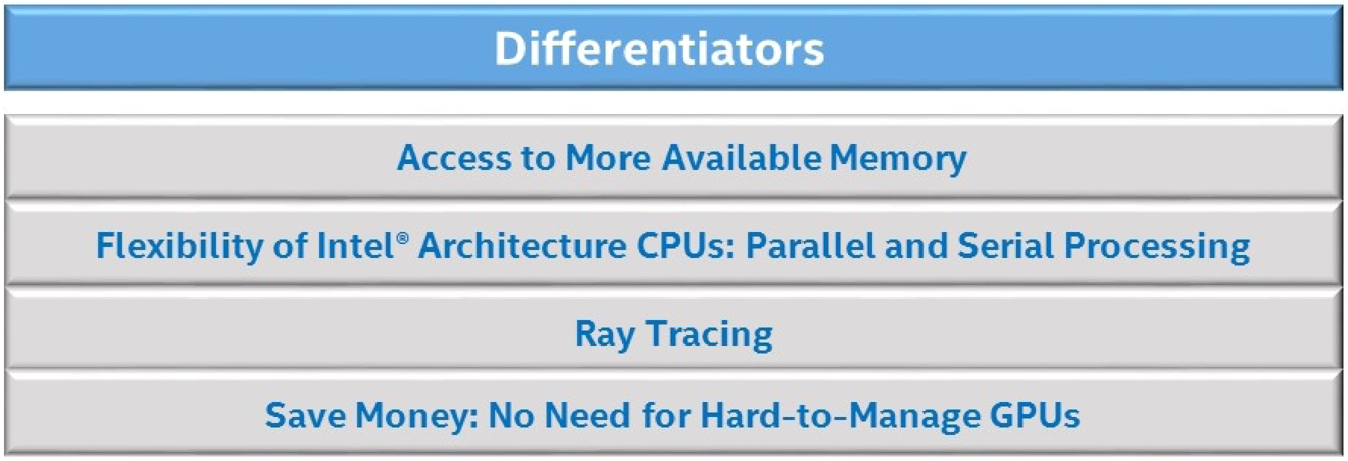
## Visualization for All, No Special Hardware Required

A big advantage of CPU-based rendering is that no special hardware is required, which means it can be used by nearly everyone on most computational hardware, from laptops and workstations to organizational clusters and leadership-class supercomputers, and even in the cloud.

Interactive photorealistic ray tracing can occur on as few as eight **Intel® Xeon® Scalable 8180 processors** or scale to big data, high-quality rendering using *in situ* nodes.[3,4,5,6] Jeffers notes that the interactive performance delivered by the Intel Rendering Framework, and photorealistic rendering with the freely available OSPRay library and viewer, “address the need and create the want.” Eliminating the requirement for specialized display hardware means even exabyte simulation data can be “visualized anywhere.” Users appreciate how they can view results on their laptops and switch to display walls or a fully immersive cave.

The ability to run and visualize anywhere using CPUs—regardless of the scale of the visualization task and without requiring specialized hardware for interactive response—is the reason HPC users are now using CPUs for visualization tasks. The integration of the Intel Rendering Framework SDVis capabilities into the popular VisIt*[7] and ParaView* viewers, along with frameworks like SENSEI*, gives everyone the ability to perform analysis and use either OpenGL rendering or create up to photoreal images.

**Figure 2** summarizes the advantages of software- versus hardware-defined visualization.


Differentiators
Access to More Available Memory
Flexibility of Intel® Architecture CPUs: Parallel and Serial Processing
Ray Tracing
Save Money: No Need for Hard-to-Manage GPUs

**2** **Advantages of software- versus hardware-defined visualization**

## From HPC to Professional Rendering Applications

Jeffers observes that one of the key factors driving SDVis adoption is the visual fidelity of the ray tracing. Basically, users get up to photorealism because the software is able to model the physics of light using both serial and parallel processing on the CPU, along with scalable, interactive performance.

The cross-market appeal of the Intel Rendering Framework with SDVis is clear. As Jeffers observes, “There is a real pull from submarkets like CAD and automotive. Photorealism is extremely important in improving ‘virtual’ vehicle design and manufacturing from commercial airplanes to military vehicles. Essentially, decisions can be made about what vehicle to build without ever having to build it. Meanwhile, there is increasing pull from adjunct markets that include offline and interactive rendering for animation and photoreal visual effects.”

## It’s All About Separation of Costs

From a software perspective, the Intel Rendering Framework provides the tuned and optimized low-level operations. This is why Jeffers claims it delivers great performance to the applications developer by simply calling the rendering APIs. The scalability to run in distributed environments is also there, which has enabled the big advance in professional rendering to “interactive”[8] rendering and ray tracing with full visual effects on huge, complicated data sets. This is why movie studios run on render farms containing thousands of Intel® CPUs.

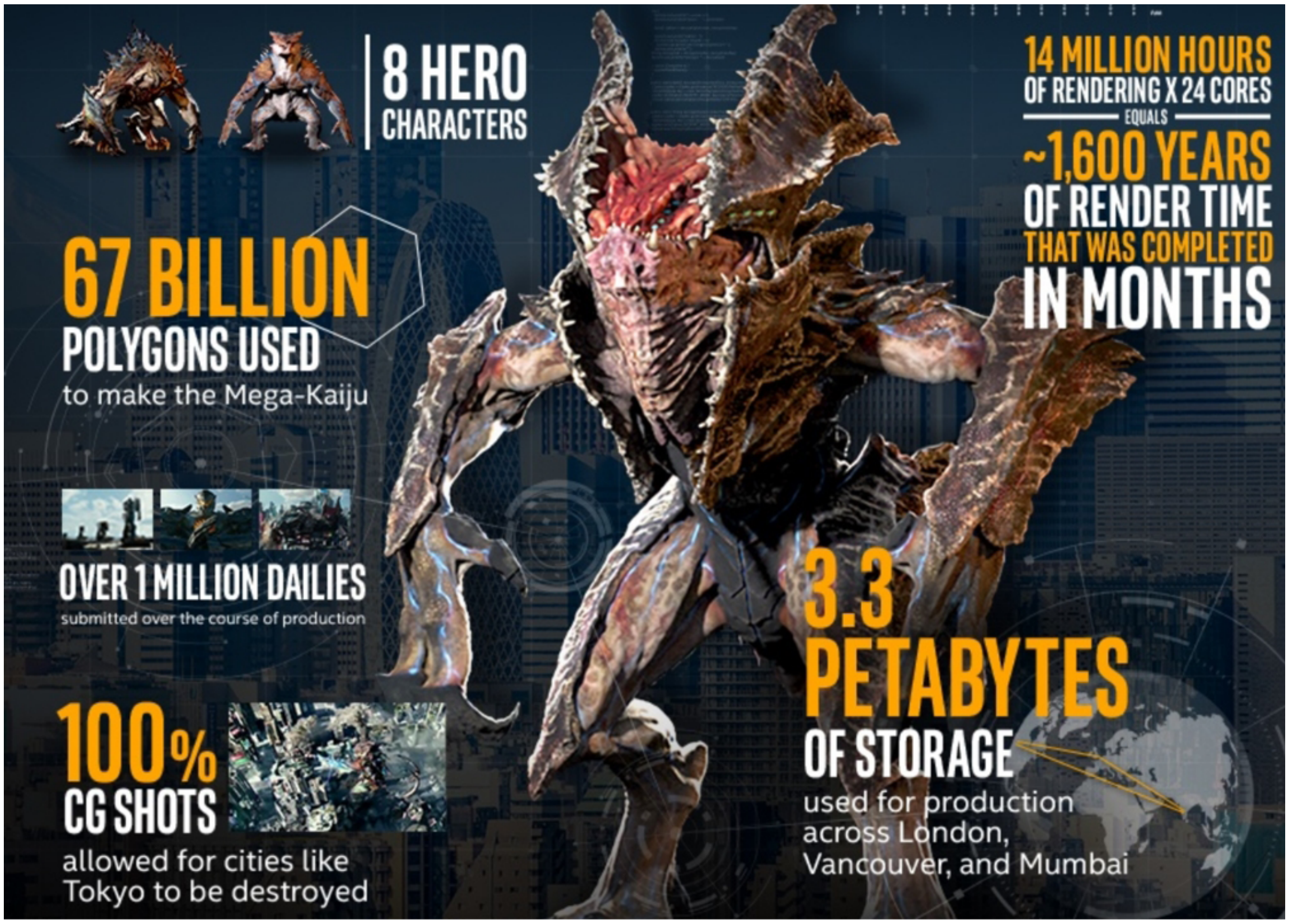
Jeffers likes to point out the differences between the animation used in the three-time Academy Award* nominated 1989 film *The Little Mermaid* and the recent *Moana* image shown below to highlight the improvements enabled by ray tracing using the Intel Rendering Framework. Previously, an overnight rendering workflow would yield a few seconds of video. The 160-billion-object ***Moana* island scene**, shown in **Figure 3** (recently made publicly available courtesy of Walt Disney Animation Studios to enable research and best industry practices), was rendered live using Intel OSPRay and Intel Embree ray tracing libraries along with the new **Intel Open Image Denoise library**. System memory capacity was important, since the rendering process consumed more than 100 GB.

**3 This image containing 160 billion objects was ray traced live using the Intel Rendering Framework and the Intel Open Image Denoise Library (Image courtesy Walt Disney Animation Studios).**

## Looking to the Future

Jeffers is also excited about the convergence of artificial intelligence (AI) and the ray tracing capabilities of Intel OSPRay and Intel Embree. For example, AI was used to define the believable movement of the robots that were rendered using these libraries in the movie *Pacific Rim* (**Figure 4**). Intel Xeon Scalable processors give the Ziva* AI software the performance needed to generate the real-time characters that can progressively learn body movements, while also easily applying features and behaviors from one character to another.[9]

When asked if photorealist animation will replace actors, Jeffers replied that he thinks humans are necessary to provide the emotional impact a movie demands. However, the technology may advance to the point where voice-overs and actor overlays will become more important as the visual fidelity of state-of-the-art rendering technology continues to improve.


8 HERO CHARACTERS
14 MILLION HOURS OF RENDERING X 24 CORES
EQUALS
~1,600 YEARS OF RENDER TIME THAT WAS COMPLETED IN MONTHS
67 BILLION POLYGONS USED to make the Mega-Kaiju
OVER 1 MILLION DAILIES submitted over the course of production
3.3 PETABYTES OF STORAGE used for production across London, Vancouver, and Mumbai
100% CG SHOTS allowed for cities like Tokyo to be destroyed

**4 AI joins with ray tracing to deliver more lifelike characters in the movie Pacific Rim (Image courtesy Intel)**

## CPU-Based Visualization

As mentioned, Intel has initiatives aside from the Intel Rendering Framework to exploit the serial and parallel performance of modern many- and multi-core Intel Xeon processors to replace dedicated hardware devices. However, the spectacular images created by the Intel Rendering Framework clearly demonstrate the appeal of CPU-based visualization. The software libraries are open-source and available for download.

Users who simply wish to experience SDVis without doing any development can download the ParaView* or VisIt* applications or the recently announced OSPRay Studio viewer. Meanwhile, HPC developers can use a framework like SENSEI* to exploit *in situ* and in-transit visualization to run at scale.

Organizations looking to experience the benefit of SDVis can look to **Intel Select Solutions for Professional Visualization** for verified hardware and software solutions that combine the latest Intel Xeon Scalable processors with other technologies such as **Intel® Omni-Path Architecture**, **Intel® SSDs**, and the **OpenHPC** cluster software stack.

Here's where application developers can get more information:

- **The Embree Ray Tracing Kernel Library**
- **The OSPRay Distributed Ray Tracing Infrastructure**
- **The OpenSWR OpenGL Software Rasterizer**
- The Intel Open Image Denoise Library will soon become available at **https://openimagedenoise.github.io/**.

While not the point of this article, interested readers can look to other Intel initiatives such as **Intel Software Defined Networking** and **Intel Software Defined Infrastructure** to see how Intel Xeon Scalable processors are being used to replace other dedicated pieces of hardware.

*Rob Farber is a global technology consultant and author with an extensive background in HPC and in developing machine learning technology that he applies at national labs and commercial organizations. Rob can be reached at* **info@techenablement.com**.

## References

[1]The benchmark only used 1/19th of the machine to render 1.1 trillion triangles. Kitware believes they could have rendered 10-20 trillion triangles per second on the full machine. (**http://www.techenablement.com/third-party-use-cases-illustrate-the-success-of-cpu-based-visualization/**)

[2]**https://science.energy.gov/~/media/ascr/pdf/program-documents/docs/Exascale-ASCR-Analysis.pdf**

[3]**http://sdvis.org/**

[4]**http://www.cgw.com/Press-Center/In-Focus/2018/Scalable-CPU-Based-SDVis-Enables-Interactive-Pho.aspx**

[5]**https://www.ixpug.org/documents/1496440983IXPUG_insitu_S1_Jeffers.pdf**

[6]**http://www.techenablement.com/third-party-use-cases-illustrate-the-success-of-cpu-based-visualization/**

[7]**https://tacc.github.io/visitOSPRay/**

[8]Interactive does not imply fluid real-time frame rates.

[9]**https://ai.intel.com/ziva-pacific-rim/**

# UNIFYING AI, ANALYTICS, AND HPC ON A SINGLE CLUSTER

**Maximizing Efficiency and Lowering Costs for Tomorrow's Enterprise**

***Allene Bhasker and Keith Mannthey, Solution Architects, Data Center Group, Intel Corporation***

The next few years will be remembered as the time when artificial intelligence (AI)—including machine and deep learning—became mainstream all over the enterprise. One study shows that more than 60% of enterprises are currently putting AI solutions in place, with predictive analytics as the most common application.

## Hosting Strategy for AI, Analytics, and HPC Workloads

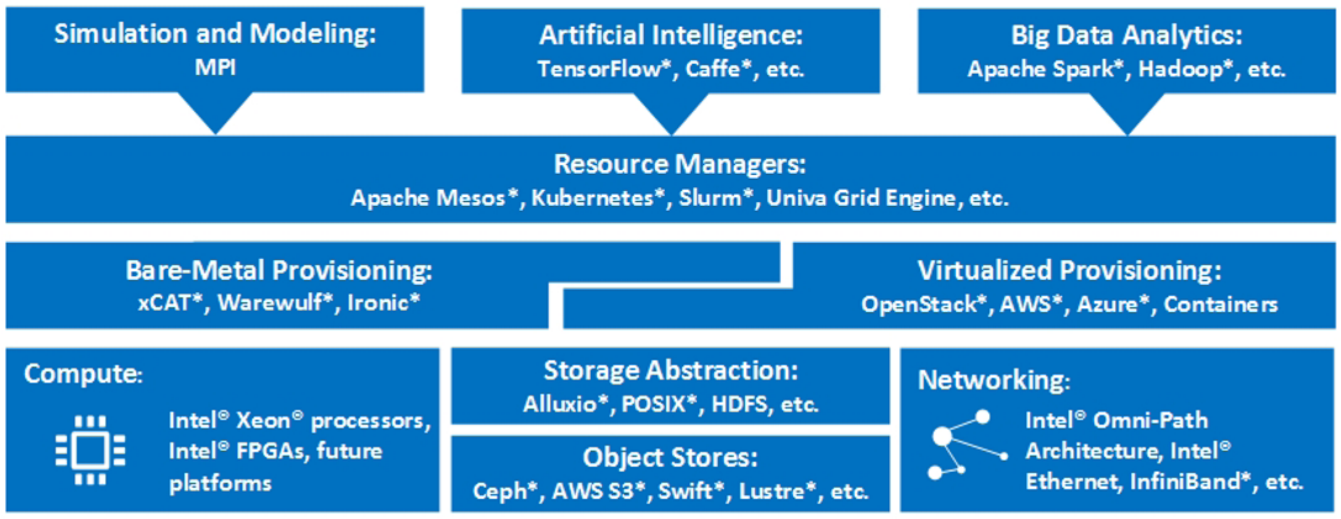
As IT organizations decide where to host AI workloads and AI-driven analytics, many consider purpose-built servers equipped with specialized accelerators and GPUs. Those who are forward-looking may think in terms of clusters of these servers to handle the expected growth of AI's role in their day-to-day operations. A broader perspective still notes that AI, analytics, and HPC workloads all run well on similar cluster hardware, based on robust individual cores and high-speed interconnects.

At this point, a common question arises: “What would it take to run AI, analytics, and HPC workloads together on the same cluster?” This approach is particularly attractive if you consider a business process that uses all three types of processing. For example, running simulation and modeling, data cleaning, and AI-based inference steps all on the same cluster is far more efficient than maintaining separate clusters.

Convergence onto a single cluster also has obvious cost benefits. Server utilization is higher with a single cluster, so you can buy fewer servers. A simpler environment is less expensive to configure and maintain—and lets you avoid expensive requirements to move and stage data among multiple clusters. Integrating these workloads onto a single environment also helps reduce latency, something that gets more important every year as real-time requirements emerge.

## Build AI and Analytics Capabilities on the Existing HPC Platform

The approach to convergence focuses on adding AI and analytics capabilities on top of an HPC cluster. The Intel HPC Platform Specification defines requirements for a base cluster solution that includes common industry standards and practices for Intel-based solutions (**Figure 1**). This provides a common and consistent interface for HPC applications, and many commercial HPC software vendors have validated application support of solutions compliant with this platform specification.


Simulation and Modeling:
MPI
Artificial Intelligence:
TensorFlow*, Caffe*, etc.
Big Data Analytics:
Apache Spark*, Hadoop*, etc.
Resource Managers:
Apache Mesos*, Kubernetes*, Slurm*, Univa Grid Engine, etc.
Bare-Metal Provisioning:
xCAT*, Warewulf*, Ironic*
Virtualized Provisioning:
OpenStack*, AWS*, Azure*, Containers
Compute:
Intel® Xeon® processors, Intel® FPGAs, future platforms
Storage Abstraction:
Alluxio*, POSIX*, HDFS, etc.
Object Stores:
Ceph*, AWS S3*, Swift*, Lustre*, etc.
Networking:
Intel® Omni-Path Architecture, Intel® Ethernet, InfiniBand*, etc.

1 **Generalized Intel solution stack for converged clusters**

Beyond the base definition, additional requirements for specific capabilities and functionalities are described in distinct sections. Compliant solutions are composed by meeting the requirements of the base solution plus the desired capability layers. This streamlines introduction or expansion of capabilities while still maintaining the interoperability with applications targeting the platform. The path to a converged platform involves adding new sections to the Intel HPC Platform Specification that describe the requirements for AI and analytics capabilities.

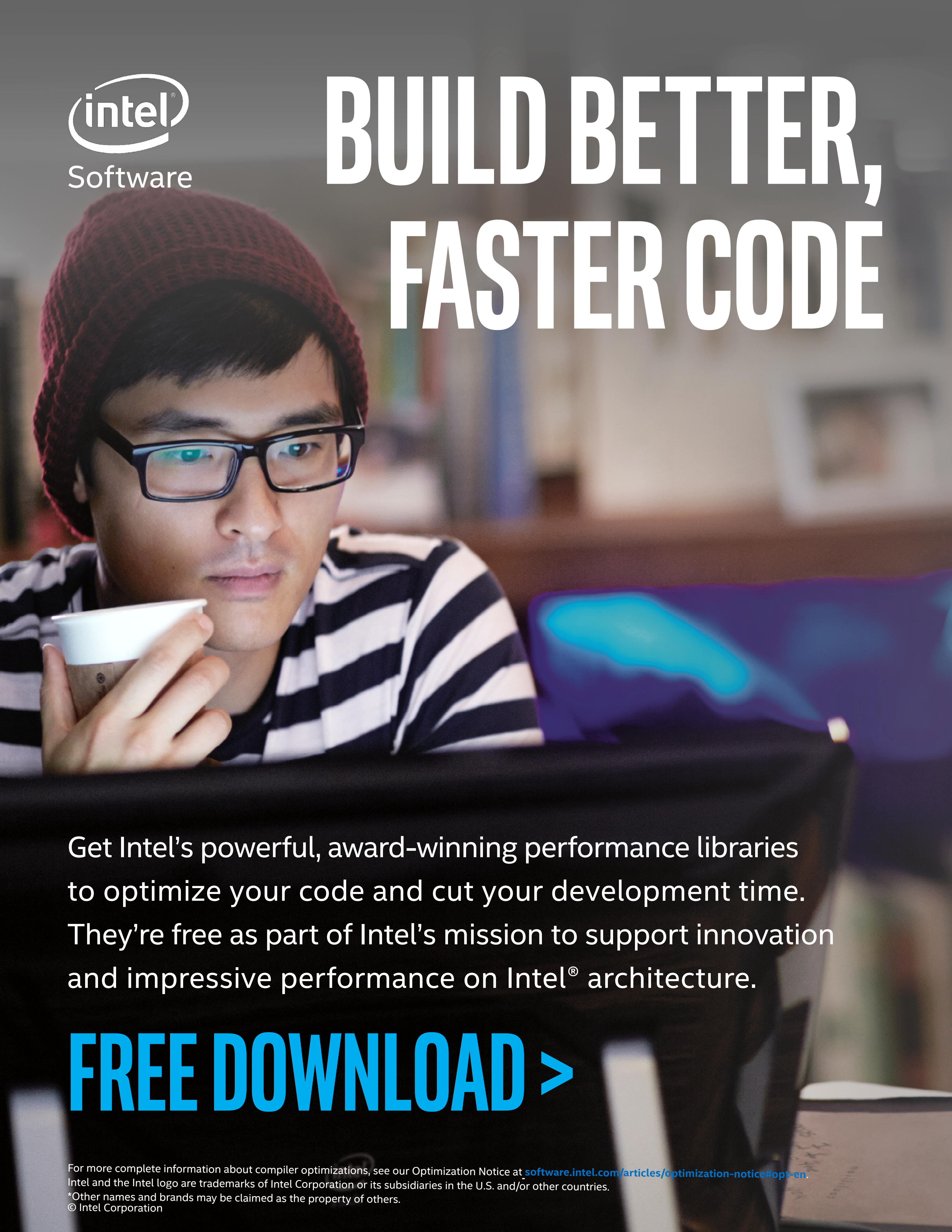
## Combining Solutions Built for Different Customer Environments

Intel is developing a series of solution architectures to help define requirements that converge AI, analytics, and HPC workloads into a single, unified cluster. Making multiple resource managers work together smoothly is a daunting challenge. And the solutions implement various approaches to integrating capabilities such as maintaining job queues and scheduling jobs in a centralized way for all types of workloads.

- **Solution 1: Extend HPC Batch Schedulers.** This approach extends batch schedulers using wrapper scripts that submit jobs on behalf of AI and analytics workloads. This simple approach has almost no systems overhead.
- **Solution 2: Univa* Grid Engine and Resource Broker.** For shops that are already using Univa Grid Engine*, this solution uses Univa Resource Broker* to integrate Apache Mesos*-compatible AI and analytics software.
- **Solution 3: Apache Mesos and Batch Schedulers.** This forthcoming solution architecture integrates Apache Mesos and batch schedulers to work together seamlessly across HPC, AI, and analytics workloads.

The solution architectures are flexible in terms of supporting different ways of provisioning, whether on bare metal or with virtual machines and containers on hybrid clouds. They also include storage abstraction to unify data across object stores, providing a single source of data to be used in place, without large-scale data movement. Intel is involved with enablement activities across the software ecosystem, including open-source contributions and co-engineering with technology providers. This optimization work is key to making sure that all three types of workloads benefit from the full range of Intel® platform features for performance and security.

To make it easier to deploy converged cluster solutions, Intel makes pre-optimized, integrated infrastructure available through participating OEMs as **Intel® Select Solutions**. Because these architectures are validated in advance, mainstream enterprises now have a clear path to the efficiency and cost benefits of converged clusters for tomorrow's AI, analytics, and HPC workloads.

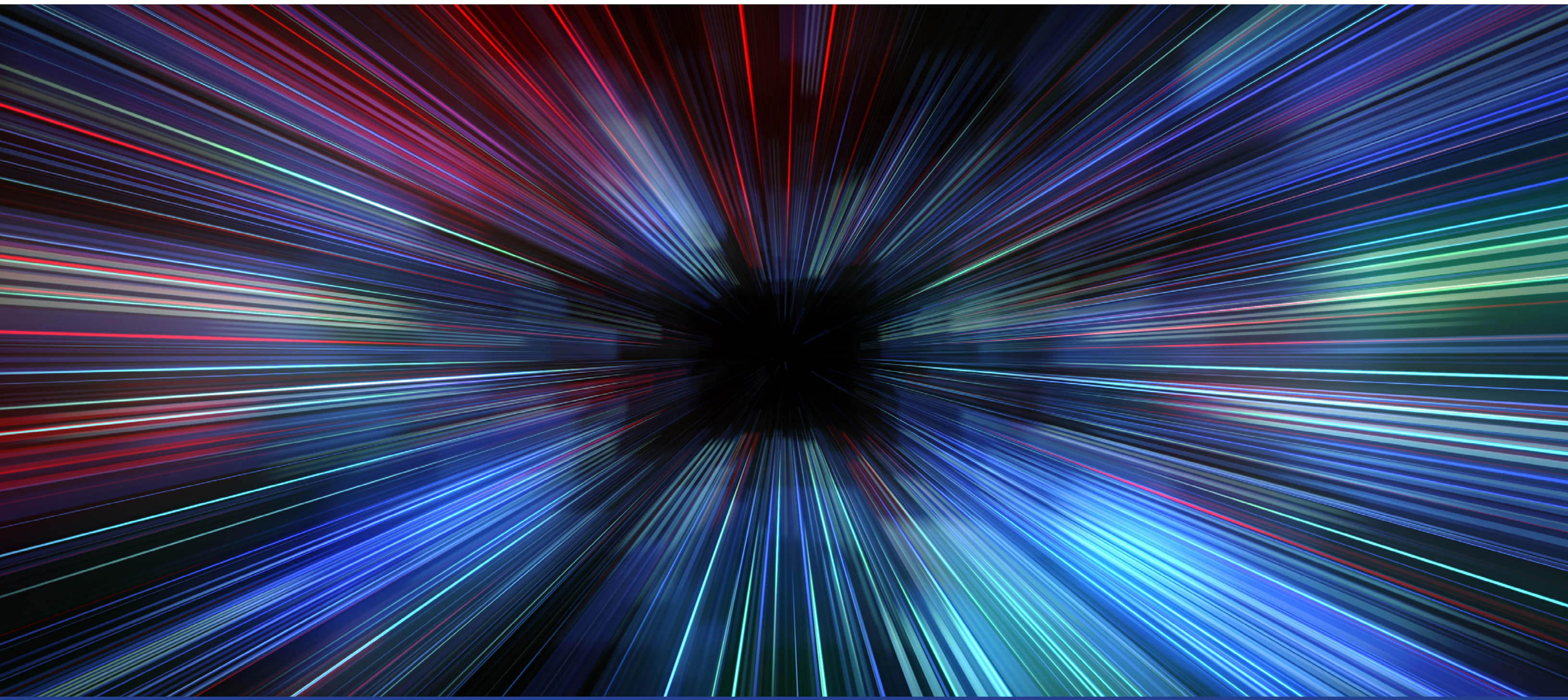


Software

# BUILD BETTER, FASTER CODE

Get Intel's powerful, award-winning performance libraries to optimize your code and cut your development time. They're free as part of Intel's mission to support innovation and impressive performance on Intel® architecture.

**FREE DOWNLOAD >**



# ADVANCING OPENCL™ FOR FPGAS

## Boosting Performance with Intel® FPGA SDK for OpenCL™ Technology

*Martin C. Herbordt, Professor, Department of Electrical and Computer Engineering, Boston University*

Field programmable gate arrays (FPGAs) are capable of very high performance, especially power-performance. This is perhaps not surprising. After all, FPGA hardware itself is malleable—configurable to match the application rather than the other way around. Also not surprising is that this additional degree of freedom—that the application developer can change the hardware as well as the software—should lead to increased complexity everywhere in the application development workflow.

And indeed, this has been the case. Until recently, most developers of FPGA applications relied on tools and methods that have more in common with those used by hardware designers than by software programmers. The languages used have been hardware description languages (HDLs) such as Verilog* and VHDL*. These describe the nature of the logic rather than a flow of instructions. The compile times (called synthesis) have been very long. And an uncomfortable amount of system knowledge has been required, especially for debug and test.

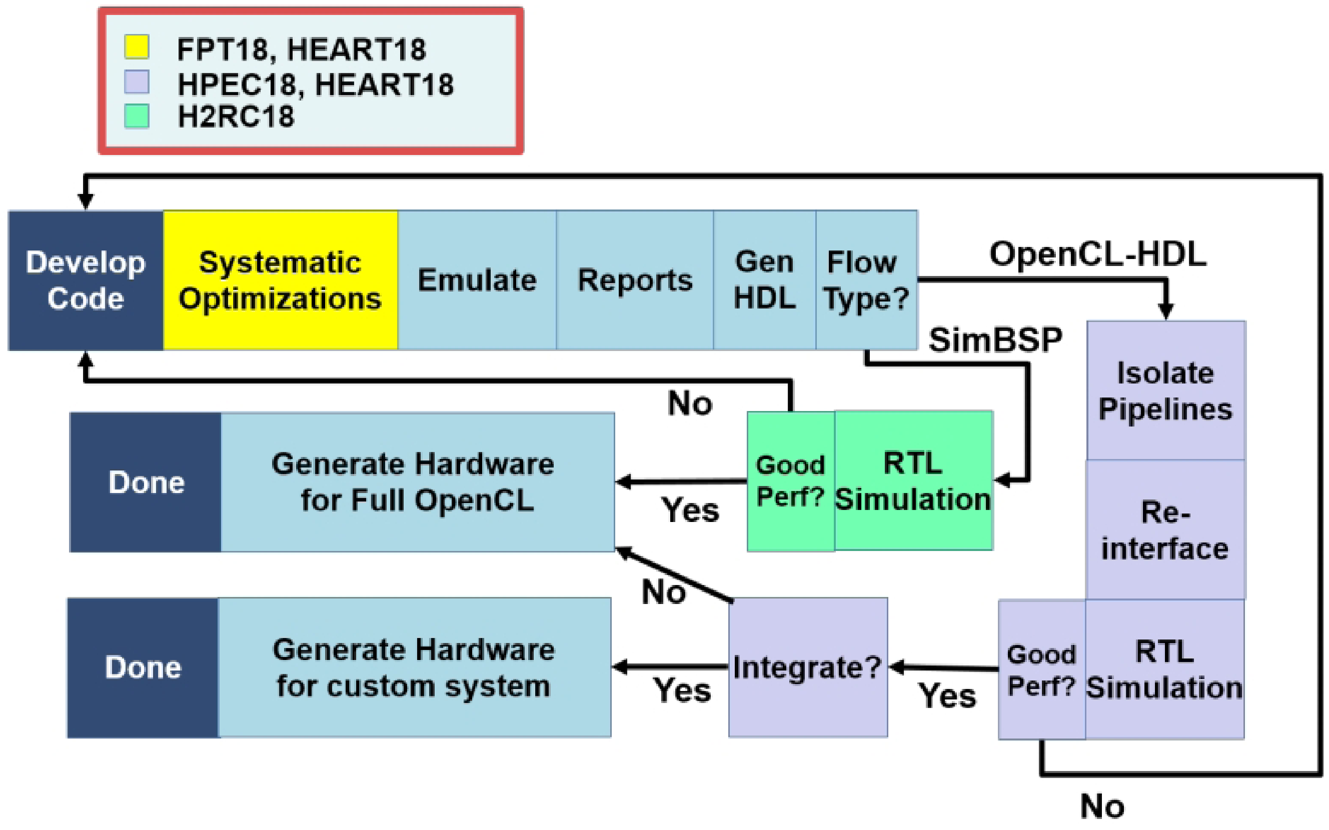
All of this is now changing. Intel has created **Intel® FPGA SDK for OpenCL™ technology**[1], which provides an alternative to HDL programming. The technology and related tools belong to a class of techniques called high-level synthesis (HLS) that enable designs to be expressed with higher levels of abstraction. Intel FPGA SDK for OpenCL technology is now in widespread use. Amazingly for long-time FPGA application developers, the performance achieved is often close to—or even better than—HDL code. But it also seems apparent that achieving this performance is often limited to developers who already know how the C-to-hardware translation works, and who have an in-house toolkit of optimization methods.

At Boston University, we've worked on enumerating, characterizing, and systematizing one such optimization toolkit. There are already a number of best practices for FPGA OpenCL documents. This work augments them, largely by applying additional methods well known to the high-performance computing (HPC) community[2]. In this methodology, we believe we're on the right track. It's taken decades for HPC performance programming to reach its current level of sophistication. We shouldn't be surprised that users of Intel FPGA SDK for OpenCL technology need to follow a similar path and learning curve.

Please note that you can see more details on the work described here in references 3 and 4 at the end of the article. The first uses the FFT as a detailed case study. The second describes the empirically guided optimization framework. Also of potential interest, in related work, references 5 and 6 show how we augmented the toolflow, which can be used to test/verify design functionality and performance without generating hardware for the entire system. As a result, we can identify design bottlenecks and the impact of optimizations with greater accuracy, and thus achieve rapid turnaround. **Figure 1** shows how these pieces fit together with the existing toolflow.

## Empirically Guided Code Optimizations

We've proposed a series of systematic and empirically guided code optimizations for OpenCL that augment current best practices and substantially improve performance. Our work characterizes and measures the impact of all these optimizations. This not only enables programmers to follow a script when optimizing their own kernels, but also opens the way for the development of autotuners to perform optimizations automatically.



**1** **The standard Intel® FPGA SDK for OpenCL™ technology toolflow is shown in light blue. Our augmentations to the standard toolflow are shown in yellow, green, and purple. This article describes the systematic optimizations.**

We broadly categorize code optimizations in this domain into three sets:

1. Intel's best practices (IBPs)
2. Universal code optimizations (UCOs)
3. FPGA-specific optimizations (FSOs)

IBPs are design strategies given in the Intel Best Practices Guide[7], which show how to express hardware using OpenCL semantics. We separate these from UCOs and FSOs because IBPs are well-known to the FPGA OpenCL community and there have been several studies characterizing their behavior.

UCOs consist of general approaches to optimizing programs that, to a large degree, are independent of the compute platform, e.g.:

- Use 1D arrays
- Records of arrays
- Predication
- Loop merging
- Scalar replacement
- Precomputing constants

Though described (for example, in reference 2), they are largely missing from IBP documentation. FSOs consist of a number of FPGA-specific optimizations that typically augment IBPs. They're based on:

- **Obtaining** a particular FPGA-specific mapping not found as an IBP
- **Facts** stated in IBPs, but which we have leveraged and converted into optimizations
- **Typically used practices** which (we have found) should actually be avoided

There are seven code versions, discussed in detail in references 4 and 6, which are incrementally developed. Each version contains one or more applied optimizations. **Table 1** summarizes the optimizations and their type (IBP, FSO, and/or UCO).

**Table 1. Summary of code versions and optimizations**

| Version | Optimizations | Type |
|---|---|---|
| 0 | (**GPU code** for porting to FPGA OpenCL) | — |
| 1 | **Single thread code with cache optimization** | IBP, FSO |
| 2 | Implement **task parallel computations in separate kernels** and connect them using channels | IBP |
|  | **Fully unroll all loops** with `#pragma unroll` | IBP, UCO |
|  | **Minimize variable declaration outside compute loops** (use temps where possible) | IBP, UCO |
|  | **Use constants** for problem sizes and data values (instead of relying on off-chip memory access) | IBP, FSO, UCO |
|  | **Coalesce** memory operations | IBP, UCO |
| 3 | Implement the entire computation within **a single kernel** and avoid using channels | FSO |
| 4 | Reduce array sizes to infer pipeline registers as registers instead of BRAMs | FSO |
| 5 | Perform **computations in detail,** using temporary variables to store intermediate results | FSO, UCO |
| 6 | **Use predication** instead of conditional branch statements when defining forks in the data path | FSO, UCO |

## Version 0: Sub-Optimal Baseline Code

A popular starting point (for example, in reference 8) is kernels based on multiple work items (MWI) such as is appropriate for GPUs. Advantages of starting here include ease of exploiting data parallelism through SIMD, and compute unit replication (CUR), which is exclusive to MWI structures.

**Algorithm 1** shows a V0-type kernel (based on reference 9). The core operation is to populate a matrix using known values of the first row and the first column. Each unknown entry is computed based on the values of its left, up, and up-left locations. This is achieved using loops which iterate in order over all matrix entries. The max function is implemented using “if-else” statements. In **Algorithm 1**, SIZE represents the dimension of blocks of matrix entries being processed.

**Algorithm 1. Needleman Wunsch-V0**

```
int tx = get_local_id(0)
_local int* temp
_local int* ref
Initialize temp from global memory
barrier(CLK_LOCAL_MEM_FENCE);
Initialize ref from global memory
barrier(CLK_LOCAL_MEM_FENCE);
for i = 1 : SIZE do
    if tx≤i then
        compute t_idx_x and t_idx_y based on tx and i
        temp[t_idx_y][t_idx_x] =
            max (temp[t_idx_y-1][t_idx_x-1] +
            ref[t_idx_y-1][t_idx_x-1],
            temp[t_idx_y][t_idx_x-1] - penalty,
            temp[t_idx_y-1][t_idx_x] - penalty);
        barrier(CLK_LOCAL_MEM_FENCE);
barrier(CLK_LOCAL_MEM_FENCE);
for i = SIZE - 2 : 0 do
    Perform computations similar to above
barrier(CLK_LOCAL_MEM_FENCE);
Store temp to global memory
```

## Version 1: Preferred Baseline Code (Used for Reference)

A less intuitive, but preferred, alternative is to use (as a baseline) single-threaded CPU code. In particular, initial designs should be implemented as single work item (SWI) kernels as recommended by IBPs. SWI kernels can infer and exploit all forms of parallelism effectively, and do so in a more efficient way than MWI kernels. The CPU-like baseline code should also be optimized for cache performance. This:

- **Helps** the compiler infer connectivity between parallel pipelines (i.e., whether data can potentially be directly transferred between pipelines instead of being stored in memory)
- **Improves** bandwidth for on-chip data access
- **Efficiently uses** the internal cache of load store units which are responsible for off-chip memory transactions

**Algorithm 2** shows the preferred baseline kernel. The first row and column of the matrix are Vector A and Vector B, respectively.

### Algorithm 2. Needleman Wunsch-V1

```
for $i$ = 1 : $Vector\_B\_Size$ do
   for $j$ = 1 : $Vector\_A\_Size$ do
      Out[i,j] = max( Out[$i$-1,$j$ ] - penalty,
         Out[$i$-1,$j$ -1] + ref[$i$,$j$ ] , Out[$i$,$j$ -1] - penalty)
```

## Version 2: IBPs

Given the preferred baseline code, we then apply the following commonly used IBPs:

- Multiple task parallel kernels
- Fully unroll all loops
- Minimizing state register usage
- Constant arrays
- Coalescing

**Algorithm 3** shows the Needleman Wunsch kernel structure after we apply IBPs. Parallelism is exploited using a systolic array, with each processing element (PE) implemented in a separate kernel. Channels are used to connect PEs in a specified sequence. For each inner loop iteration, PEs compute consecutive columns within the same row. This ensures spatial locality for memory transactions. The drawback is data dependencies between kernels, which can't be reliably broken down by the compiler since it optimizes each kernel as an individual entity. Thus, the overhead of synchronizing data paths can result in performance degradation.

### Algorithm 3. Needleman Wunsch-V2

```
N ← Size of systolic array
$PE_k$ → Kernel Begin
int up, left, up_left, cached_up, cached_up_left
   Initialize cached_up & cached_up_left
   for $j$ = 1 : 1 : $Vector\_B\_Size$ do
      left ← read_channel ($PE_{k-1}$)
      up = cached_up
      up_left = cached_up_left
      cached_up = max(up - penalty,
         left - penalty , up_left + ref [$j$,$i$])
      cached_up_left = left
      Out[j,i] = cached_up
      write_channel ($PE_{k-1}$) ← cached_up
```

## Version 3: Single-Kernel Design

In Version 3, we merge the IBP-optimized task parallel kernels and declare all compute loops within the same kernel. This is because the compiler is still able to automatically infer task parallel pipelines. Having a single kernel carries a number of advantages over the multi-kernel approach, e.g.:

- **Inherent** global synchronization
- **Reduced** resource usage and delays through pipeline merging/reordering
- **Simplified** control logic

**Algorithm 4** shows the kernel structure for implementing the systolic array as a single kernel. The compiler can now optimize the entire computation, as opposed to individual PEs. Synchronization overhead is also reduced, since almost all computation is tied to a single loop variable (j ). Nested loops are used because, in this particular case, the cost of initiation intervals is outweighed by the reduction in resource usage. This is because the compiler was unable to infer data access patterns when loops were coalesced.

### Algorithm 4. Needleman Wunsch-V3

```
N ← Size of systolic array
int value[N+1], left[Vector_B_Size]
left ← Vector_B
for i = 1 : 1 : Vector_A_Size=N do
   base = f(i)
   value ← Vector_A[base:base+N+1]
   for j = 1 : 1 : Vector_B_Size do
      int up_left[N+1]
      for k = 2 : 1 : N + 1 do
         up_left[k] = value[k-1]
      value[1] = left[j]
      #pragma unroll
      for k = 2 : 1 : N + 1 do
         value[k] = max(value[k-1] - penalty,
           up_left[k] + ref[j, base+k] , value[k] - penalty)
      left[j] = value[N+1]
      Out value[2:N+1]
```

## Version 4: Reduced Array Sizes

Having large variable arrays results in pipeline registers being inferred as BRAMs instead of registers, which can have significant drawbacks on the design. Since BRAMs can't source and sink data with the same throughput as registers, barrel shifters and memory replication are required. This drastically increases resource usage. Moreover, the compiler is also unable to launch stall-free iterations of compute loops due to memory dependencies. The solution is to break large arrays corresponding to intermediate variables into smaller ones.

**Algorithm 5** shows the kernel structure for inferring pipeline registers as registers. All arrays are expressed as individual variables, generated using scripts, with the exception of local storage of Vector B in "left," which has low throughput requirements.

**Algorithm 5. Needleman Wunsch-V4**

```
N ← Size of systolic array
int value_1, value_2 ... Value_N_plus_1
int left [Vector_B_Size]
left Vector_B
for i = 1 : 1 : Vector_A_Size=N do
   base = f(i)
   value_1 Vector_A[base]
         ↓
   value_N plus 1 Vector_A[base+N+1]
   for j = 1 : 1 : Vector_B_Size do
      int up_left_2 ... Up_left_N_plus_1
      up_left_2 = value_1
         ↓
   up_left_N_plus_1 = value_N
   value_1 = left [j]
   value_2 = max(value_1 - penalty,
      up_left_2 + ref[j,base+2], value_2 - penalty)
         ↓
   value_N_plus_1 = max(value_N - penalty,
      up_left_N_plus_1 + ref[j,base+N+1],
      value_N_plus_1 - penalty)
   left[j] = value_N_plus_1
   Out ← value 2 ... value N plus 1
```

## Version 5: Detailed Computations

The OpenCL compiler doesn't reliably break down large computations being assigned to a single variable into intermediate stages. This reduces the number of possible pipeline stages and can result in larger critical paths and data dependency stalls. Our solution is to do computations in as much detail as possible by using intermediate variables to help the compiler infer pipelines. If the logic is already optimal, these variables will be synthesized away and won't waste resources.

**Algorithm 6** shows the kernel structure after performing computations in detail with a number of intermediate variables added. The "max" function is also explicitly implemented.

**Algorithm 6. Needleman Wunsch-V5**

```
N ← Size of systolic array
int value_1, value_2 ... Value_N_plus_1
int left [Vector B Size]
left ← Vector_B
for i = 1 : 1 : Vector_A_Size=N do
   base = f(i)
   value_1 Vector_A[base]
      ↓
   value_N_plus_1 Vector_A[base+N+1]
   for j = 1 : 1 : Vector_B_Size do
     int a_2 = value_1 + ref[j,base+2];
     value_1 = left[j]

   int b_2 = value_1 - penalty
   int a_3 = value_2 + ref[j,base+3];
   int c_2 = value_2 - penalty

   if ((a_2 ≥ b_2) && (a_2 ≥ c_2))
     value_2 = a_2
   else if ((b_2 > a_2) && (b_2 ≥ c_2))
     value_2 = b_2
   else
     value_2 = c_2

   int b_3 = value 2 - penalty
   int a_4 = value 3 + ref[j,base+4];
   int c_3 = value 3 - penalty
     .
     .
     .
left[j] = value_N_plus_1
Out ← value 2 ... Value_N_plus_1
```

## Version 6: Predication

We optimize conditional operations by explicitly specifying architecture states which ensure the validity of the computation. Since hardware is persistent and will always exist once synthesized, we avoid using conditional branch statements. Instead, variable values are conditionally assigned such that the output of invalid operations is not committed and hence does not impact the overall result. **Algorithm 7** shows the "if-else" operations replaced with conditional assignments.

**Algorithm 7. Needleman Wunsch-V6**

```
.
.
.
int a_2 = value_1 + ref[j,base+2];
value_1 = left[j]

int b_2 = value_1 - penalty
int a_3 = value_2 + ref[j,base+3];
int c_2 = value_2 - penalty

int d_2 = (a_2 > b_2) ? a_2 : b_2
value_2 = (c_2 > d_2) ? C_2 : d_2
.
.
.
```

## Hardware Specifications

The designs are implemented using an **Intel® Arria® 10AX115H3F34I2SG FPGA** and **Intel® FPGA SDK for OpenCL™ technology 16.0**. This FPGA has 427,200 ALMs, 1,506K logic elements, 1,518 DSP blocks, and 53 MB of on-chip storage. For GPU implementations, we use the NVIDIA Tesla* P100 PCIe 12GB GPU with CUDA* 8.0. It has 3,584 CUDA cores and peak bandwidth of 549 GB/s. CPU codes are implemented on a 14-core, 2.4 GHz **Intel® Xeon® E5-2680v4 processor** with **Intel® C++ Compiler v16.0.1**.

## Optimization Characterization

The optimizations are tested for the full OpenCL compilation flow using these benchmarks:

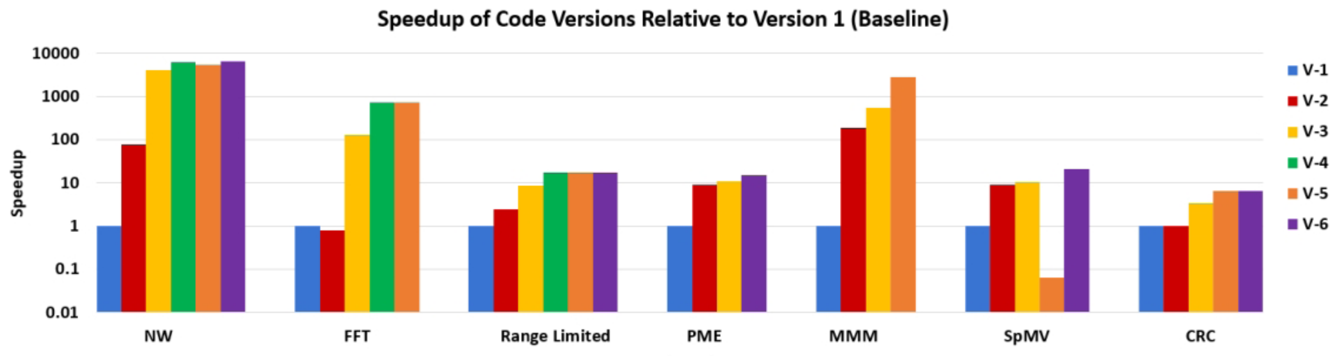
- Needleman Wunsch (NW)
- Fast Fourier Transform (FFT)
- Range Limited Molecular Dynamics (Range Limited)
- Particle Mesh Ewald (PME)
- Dense Matrix-Matrix Multiplication (MMM)
- Sparse Matrix Dense Vector Multiplication (SpMV) and Cyclic Redundancy Check (CRC)

**Table 2** provides a summary of these benchmarks, their associated dwarfs[8], tested problem sizes, and applicable code versions that are developed.

**Table 2. Benchmark summary**

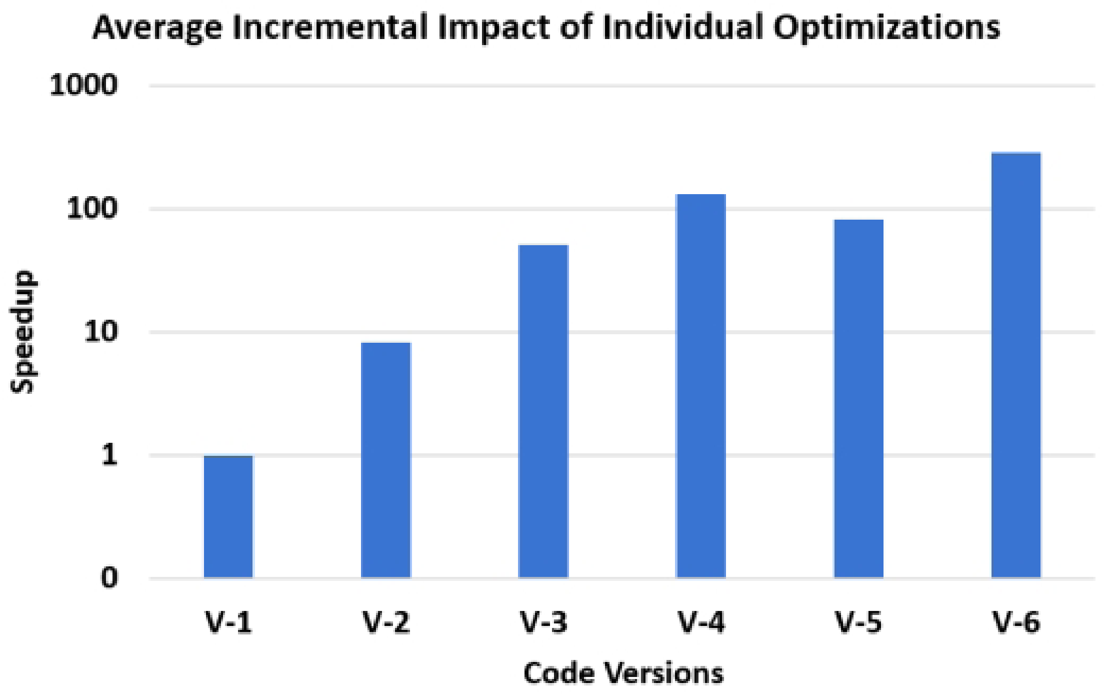
| Benchmark | Dwarf | Problem Size | V1 | V2 | V3 | V4 | V5 | V6 |
|---|---|---|---|---|---|---|---|---|
| NW | Dynamic Programming | 16K x 16K Integer Table | ● | ● | ● | ● | ● | ● |
| FFT | Spectral Methods | 64 point Radix-2 1D FFT, 8,192 Vectors | ● | ● | ● | ● | ● | ● |
| Range Limited | N-Body | 180 Particles per Cell, 15% Pass Rate | ● | ● | ● | ● | ● | ● |
| PME | Structured Grids | 1,000,000 Particles, 323 Grid, 3D Tri-Cubic | ● | ● | ● |  |  | ● |
| MMM | Dense Linear Algebra | 1K x 1K Matrix, Single Precision | ● | ● |  |  | ● | ● |
| SpMV | Sparse Linear Algebra | 1K x 1K Matrix, Single Precision, 5%-Sparsity, NZ=51,122 | ● | ● | ● |  | ● | ● |
| CRC | Combinational Logic | 100 MB CRC32 | ● | ● | ● |  | ● | ● |

**Figure 2** shows the results of individual optimizations. In almost all cases, we can see the same trend where traditional optimizations (V2) only result in a fraction of the speedup possible. By applying the additional optimizations on top of V2, performance is improved by orders of magnitude.



**2** **Impact of systematic application of the proposed optimizations to a cache-optimized CPU baseline code. In almost all cases, every subsequent code version shows increasing performance, with up to orders of magnitude better performance possible for fully-optimized**

The average impact of individual optimizations is shown in **Figure 3**. Generally, each successive set of optimizations applied results in increasing performance. The exception is V5. This is due to higher execution times of V5 for NW and SpMV. In both cases, performing computations in as much detail as possible results in the use of conditional statements that outweigh the benefits of the optimization. Once these statements are removed in V6, the speedup increases.



**3** **Performance for different code versions, obtained by averaging the speedup of all applicable benchmarks.**

To demonstrate the overall effectiveness of the approach, we compare the performance of the optimized kernels against existing CPU, GPU, Verilog, and FPGA-OpenCL implementations. **Table 3** lists the references for these designs. They're either obtained from the literature or implemented using available source code/libraries (denoted by an asterisk). Verilog FFT measurement from reference 3 has been extended to include off-chip access overhead.

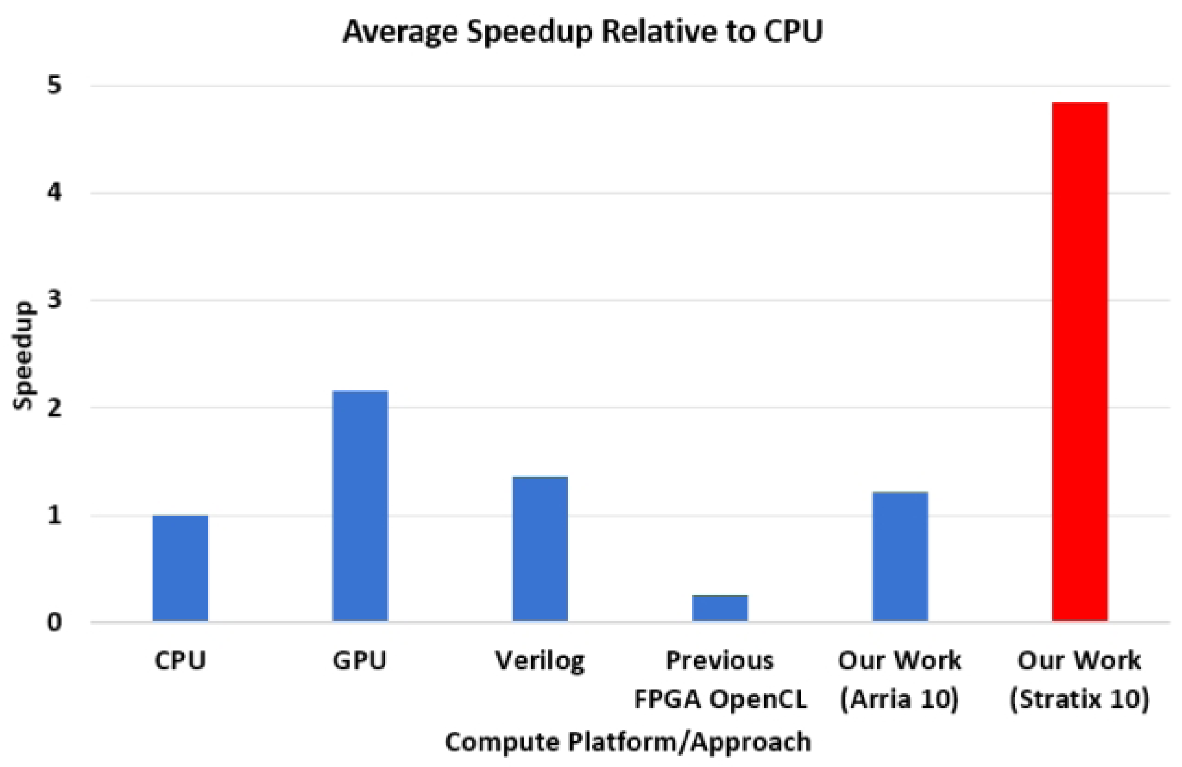
**Table 3. References for existing implementations**

| Benchmark | CPU | GPU | Verilog* | OpenCL™ |
|---|---|---|---|---|
| NW | Rodinia*[9] | Rodinia*[9] | Benkrid*[10] | Zohouri*[11] |
| FFT | MKL*[12] | cuFFT**[13] | Sanaullah*[3] | Intel[14] |
| Range Limited | — | — | Yang*[15] | Yang[15] |
| PME | Ferit[16] | Ferit*[16] | Sanaullah[17] | — |
| MMM | MKL*[12] | cuBLAS*[18] | Shen*[19] | Spector*[20] |
| SpMV | MKL*[12] | cuSPARSE*[18] | Zhou*[22] | OpenDwarfs*[8] |
| CRC | Brumme*[23] | — | Anand*[24] | OpenDwarfs[8] |

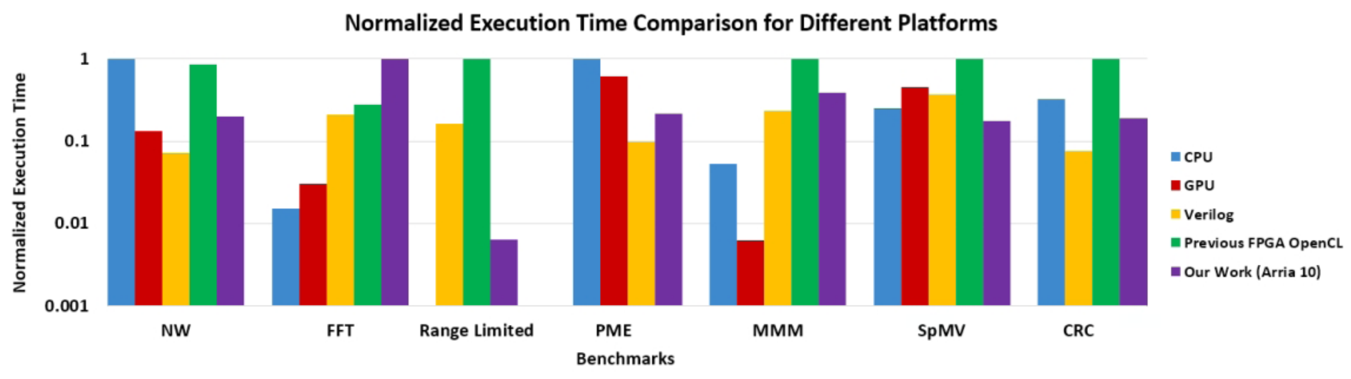
**Figure 4** shows the average speedup achieved over the CPU code, while **Figure 5** shows the normalized execution times for all implementations. From the results, we observe that our work outperforms multicore CPU implementations by approximately 1.2x due to the performance of codes written using **Intel® Math Kernel Library (Intel® MKL)**. We've also achieved an average of approximately 5x lower execution time than existing FPGA OpenCL work. The GPU speedup of 2.4x relative to our work is due to the use of a high-end GPU (Tesla* P100) compared to a midrange FPGA (**Intel® Arria® 10 FPGAs**). We therefore also provide an estimate of high-end FPGA performance (Stratix R 10*) using a conservative factor of 4x to account for an increase in resource only. Results show that the optimized kernels on Stratix 10 are expected to outperform GPU designs by 65%, on average.

## Conclusions

Comparison with existing Verilog* implementations show that the OpenCL kernels are, on average, within 12% of hand-tuned HDL. This demonstrates that the optimizations are able to bridge the performance-programmability gap for FPGAs and deliver HDL-like performance using OpenCL.



**4** Average speedup with respect to CPU across all applicable benchmarks



**5** Performance of our work compared to existing CPU, GPU, Verilog*, and FPGA OpenCL implementations. Our work outperforms CPU and OpenCL for most of the benchmarks. Moreover, we also achieve speedups over GPU (SpMV, PME) and Verilog (SpMV, Range Limited).

## References

[1]**Intel® FPGA SDK for OpenCL™ technology**

[2]S. Chellappa, F. Franchetti, and M. Pueschel, “How To Write Fast Numerical Code: A Small Introduction," in *Generative and Transformational Techniques in Software Engineering II, Lecture Notes in Computer Science v5235*, 2008, pp. 196 - 259.

[3]A. Sanaullah and M. Herbordt, “FPGA HPC using OpenCL: Case Study in 3D FFT," in *Proceedings of the International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, 2018.

[4]A. Sanaullah, R. Patel, and M. Herbordt, “An Empirically Guided Optimization Framework for FPGA OpenCL," in *Proceedings of the IEEE Conference on Field Programmable Technology*, 2018.

[5]A. Sanaullah, C. Yang, D. Crawley, and M. Herbordt, “SimBSP: Enabling RTL Simulation for Intel FPGA OpenCL Kernels," in *Proceedings on Heterogeneous High-Performance Reconfigurable Computing*, 2018.

[6]A. Sanaullah and M. Herbordt, “Unlocking Performance-Programmability by Penetrating the Intel FPGA OpenCL Toolow," in *IEEE High Performance Extreme Computing Conference*, 2018.

[7]"**Intel FPGA SDK for OpenCL Pro Edition Best Practices Guide**"

[8]K. Krommydas, A. E. Helal, A. Verma, and W.C. Feng, “Bridging the Performance Programmability Gap for FPGAs via OpenCL: A Case Study with Opendwarfs," Department of Computer Science, Virginia Polytechnic Institute and State University, Tech. Rep., 2016.

[9]S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A Benchmark Suite for Heterogeneous Computing," in I*EEE International Symposium on Workload Characterization*, 2009, pp. 44{54.

[10]K. Benkrid, Y. Liu, and A. Benkrid, “A Highly Parameterized and Efficient FPGA-Based Skeleton for Pairwise Biological Sequence Alignment," I*EEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 17, No. 4, pp. 561 - 570, 2009.

[11]H. R. Zohouri, N. Maruyamay, A. Smith, M. Matsuda, and S. Matsuoka, “Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs" in *International Conference for High Performance Computing, Networking, Storage and Analysis*, SC16, 2016, pp. 409{420.

[12]**Intel® Math Kernel Library**

[13]C. Nvidia, “CuFFT Library," 2010.

[14]**FFT (1D) Design Example**

## References (Continued)

[15]C. Yang, J. Sheng, R. Patel, A. Sanaullah, V. Sachdeva, and M. Herbordt, "OpenCL for HPC with FPGAs: Case Study in Molecular Electrostatics," in IEEE High Performance Extreme Computing Conference, 2017.

[16]F. Buyukkececi, O. Awile, and I. F. Sbalzarini, "A Portable OpenCL Implementation of Generic Particle—Mesh and Mesh—Particle Interpolation in 2D and 3D," *Parallel Computing*, Vol. 39, No. 2, pp. 94 - 111, 2013V

[17]A. Sanaullah, A. Khoshparvar, and M. C. Herbordt, "FPGA-Accelerated Particle-Grid Mapping," in *Field-Programmable Custom Computing Machines (FCCM)*, 2016 IEEE 24th Annual International Symposium on. IEEE, 2016, pp. 192 - 195.

[18]Nvidia, "CUBLAS Library," NVIDIA Corporation, Santa Clara, CA, 2008.

[19]J. Shen, Y. Qiao, Y. Huang, M. Wen, and C. Zhang, "Towards a Multi-Array Architecture for Accelerating Large-Scale Matrix Multiplication on FPGAs," in *Circuits and Systems (ISCAS)*, 2018 IEEE International Symposium on. IEEE, 2018, pp. 1 - 5.

[20]Q. Gautier, A. Althoff, P. Meng, and R. Kastner, "Spector: An OpenCL FPGA Benchmark Suite," in *International Conference on Field-Programmable Technology*, 2016.

[21]Nvidia, "CuSparse Library," NVIDIA Corporation, Santa Clara, CA, 2014.

[22]L. Zhuo and V. K. Prasanna, "Sparse Matrix-Vector Multiplication on FPGAs," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 2005.

[23]S. Brumme, "Fast CRC32," http://create.stephan-brumme.com/crc32/, 2018.

[24]P. A. Anand et al., "Design of High Speed CRC Algorithm for Ethernet on FPGA using Reduced Lookup Table Algorithm," in India Conference (INDICON), 2016 IEEE Annual. IEEE, 2016, pp. 1- 6.

# PARALLELISM IN PYTHON*

## Dispelling the Myths with Tools to Achieve Parallelism

***David Liu, Software Technical Consulting Engineer, and Anton Malakhov, Software Development Engineer, Intel Corporation***

Python* as a programming language has enjoyed nearly a decade of usage in both industry and academia. This high-productivity language has been one of the most popular abstractions to scientific computing and machine learning, yet the base Python language remains single-threaded. Just how is productivity in these fields being maintained with a single-threaded language?

The Python language's design, by Guido van Rossum, was meant to trade off type flexibility and predictable, thread-safe behavior against the complexity of having to manage static types and threading primitives. This, in turn, meant having to enforce a global interpreter lock (GIL) to limit execution to a single thread at a time to preserve this design mentality. Over the last decade, many concurrency implementations have been made for Python—but few in the region of parallelism. Does this mean the language isn't performant? Let's explore further.

The base language's fundamental constructs for loops and other asynchronous or concurrent calls all abide by the single-threaded GIL, so even list comprehensions such as `[x*x for x in range(0,10)]` will always be single-threaded. The threading library's existence in the base language is also a bit misleading, since it provides the behavior of a threading implementation but still operates under the GIL. Many of the features of Python's concurrent futures to almost-parallel tasks also operate under the GIL. Why does such an expressive productivity language restrict the language to these rules?

The reason is the level of abstraction the language design adopted. It ships with many tools to wrap C code, from ctypes to cffi. It prefers multiprocessing over multithreading in the base language, as evidenced by the multiprocessing package in the native Python library. These two design ideas are evident in some of the popular packages, like NumPy* and SciPy*, which use C code under the Python API to dispatch to a mathematical runtime library such as **Intel® Math Kernel Library (Intel® MKL)** or OpenBLAS*. The community has adopted the paradigm to dispatch to higher-speed C-based libraries, and has become the preferred method to implement parallelism in Python.

In the combination of these accepted methods and language limitations are options to escape them and apply parallelism in Python through unique parallelism frameworks:

- **Numba*** allows for JIT-based compilation of Python code which can also run LLVM*-based Python-compatible code.
- **Cython*** gives Python-like syntax with compiled modules that can target hardware vectorization as it compiles to a C module.
- **numexpr*** allows for symbolic evaluation to utilize compilers and advanced vectorization.

These methods escape Python's GIL in different ways while preserving the original intent of the language, and all three implement different models of parallelism.

Let's take the general example of one of the most common language constructs on which we'd want to apply parallelism—the for loop. Looking at the loop below, we can see that it provides a basic service, returning all the numbers less than 50 in a list:

```
def test_func(list_of_items):
        final_list = []
        for item in list_of_items:
                if item < 50:
                final_list.append(item)
        return final_list
```

Running this code gives the following result:

```
import random
random_list = [random.randint(0,1000000) for x in range(0,1000000)]
%timeit test_func(random_list)
27.4 ms ± 331 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Python handles the list of items in a single-threaded way under the GIL, since it's written in pure Python. Thus, it handles everything sequentially and doesn't apply any parallelism to the code. Because of the way this code is written, it's a good candidate for the Numba framework. Numba uses a decorator (with the @ symbol) to flag functions for just-in-time (JIT) compilation, which we'll try to apply on this function:

```
@jit(nopython=True)
def test_func(list_of_items):
    final_list = []
    for item in list_of_items:
        if item < 50:
            final_list.append(item)
    return final_list
```

Running this code now gives the following result:

```
%timeit test_func(random_list)
15.7 ms ± 173 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Including this simple decorator nearly doubled performance. This works because the original Python code is written in primitives and datatypes that can be easily compiled and vectorized to a CPU. Python lists are the first place to look. Normally, this data structure is quite heavy with its loose typing and built-in allocator. However, if we look at the datatypes that `random_list` contains, they're all integers. Because of this consistency, the JIT compiler of Numba can vectorize the loop.

If the list contains mixed items (e.g., a list of chars and ints), the compiled code will throw a TypeError because it can't handle the heterogeneous list. Also, if the function contains mixed datatype operations, Numba will fail to produce a high-performance JIT-compiled code and will fall back to Python object code.

The lesson here is that achieving parallelism in Python depends on how the original code is written. Cleanliness of datatypes and the use of vectorizable data structures allow Numba to parallelize code with the insertion of a simple decorator. Being careful about the use of Python dictionaries pays dividends, because historically they don't vectorize well. Generators and comprehensions suffer from the same problem. Refactoring such code to lists, sets, or arrays can facilitate vectorization.

Parallelism is much easier to achieve in numerical and symbolic mathematics. NumPy and SciPy do a great job dispatching the computation outside of Python's GIL to lower-level C code and the Intel MKL runtime. Take, for example, the simple NumPy symbolic expression, `((2*a + 3*b)/b)`, expressed below:

```
import numpy as np
a = np.random.rand(int(1e6))
b = np.random.rand(int(1e6))

%timeit (2*a + 3*b)/b
8.61 ms ± 108 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

This expression makes multiple trips through the single-threaded Python interpreter because of the structure and design of NumPy. Each return from NumPy is dispatched to C and returned back to the Python level. Then, the Python object is sent to each subsequent call to be dispatched to C again. This back-and-forth jumping becomes a bottleneck in the computation, so when you need to compute custom kernels that can't be described in NumPy or SciPy, numexpr is a better option:

```
import numexpr as ne
%timeit ne.evaluate('(2*a + 3*b)/b')
2.22 ms ± 52.7 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

How does numexpr achieve nearly a 4x speedup? The previous code takes the symbolic representation of the computation into numexpr's engine to generate code that works with the vectorization commands from the vector math library in Intel MKL. Thus, the entire computation stays in low-level code before completing and returning the result back to the Python layer. This method also avoids multiple trips through the Python interpreter, cutting down on single-threaded sections while also providing a concise syntax.

By looking at the Python ecosystem and evaluating the different parallelism frameworks, it's evident that there are good options. To master Python parallelism, it's important to understand the tools and their limitations. Python chose the GIL as a design consideration to simplify framework development and give predictable language behavior. But, at the end of the day, the GIL and its single-threaded restrictions are easy to sidestep with the right tools.

## Learn More

- **Intel® Distribution for Python**
- **Intel® Math Kernel Library**

## BLOG HIGHLIGHTS

### 10 Huge Benefits of Edge AI & the Software Tools to Deliver Them

**CHARLOTTE DRYDEN, INTEL CORPORATION**

Artificial Intelligence (AI) continues to show up in our everyday lives, but its presence is gentle and welcome, largely due to the advancements in Edge AI. Many AI use cases are best suited for the edge where processing happens at or close to the data source, lowering costs, reducing application or service latency, improving reliability and increasing data privacy.

Whether we realize it or not, Edge AI technologies—seen and unseen—provide huge benefits in a world that's digitally connected, 24x7. This rapid advancement of Edge AI is not because of one or two "killer apps"—new solutions and usages continue to emerge all the time.

**Read more >**

# DECODE YOUR TECH FUTURE

intel Software

## Welcome to Tech.Decoded, the Knowledge Hub for Developers

You'll find an always-growing library of information curated to help you get the most out of modern hardware. Boost your competitive edge. And get to market faster.

### Get Expert Insights

Watch tech forecasters and visionaries explore today's tech landscape: code modernization, systems and IoT, data science, and more.

### Dig Deeper

Learn how to get every last ounce of performance from your code with on-demand webinars covering today's most important strategies, practices, and tools.

### Put it All to Work in your Code

Use short videos and articles to understand the how-to's of key programming tasks using specific development tools.

**EXPLORE TECH.DECODED NOW >**

# REMOVE MEMORY BOTTLENECKS USING INTEL® ADVISOR

## Understanding How Your Program is Accessing Memory Helps You Get More from Your Hardware

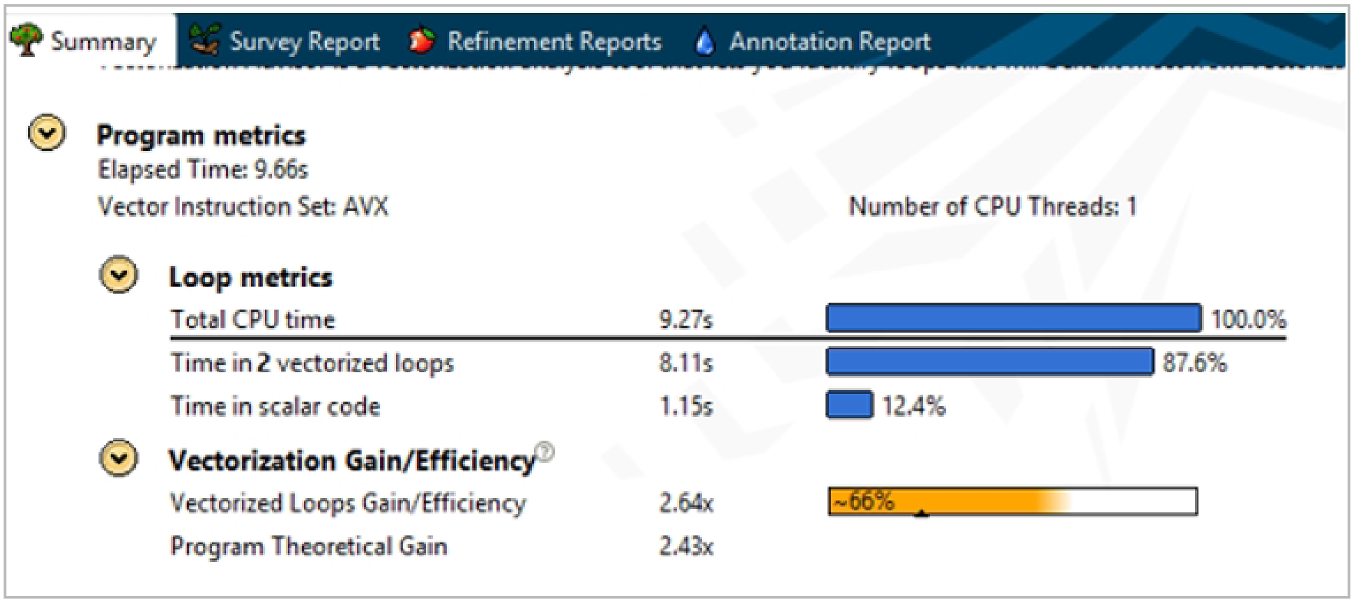
*Kevin O'Leary and Alex Shinsel, Technical Consulting Engineers, Intel Corporation*

How your application accesses memory dramatically impacts performance. It's not enough to parallelize your application by adding threads and vectorization. Effective use of memory bandwidth is just as important. But often, software developers just don't understand it. Tools that help minimize memory latency and increase bandwidth can help pinpoint performance bottlenecks and diagnose their causes. One such tool is **Intel® Advisor**, which has features to help optimize memory access and eliminate memory bottlenecks:

- **Roofline analysis** with the new Integrated Roofline feature
- **Memory Access Pattern Analysis (MAP)**
- **Memory Footprint analysis**

## Getting Great Performance

To get top performance out of your application, you need to know how well you're using all system resources. You can see some useful metrics on your overall program in the Intel Advisor Summary view (**Figure 1**), which gives you an indication of how well the application is vectorized.



Program metrics
Elapsed Time: 9.66s
Vector Instruction Set: AVX
Number of CPU Threads: 1

Loop metrics

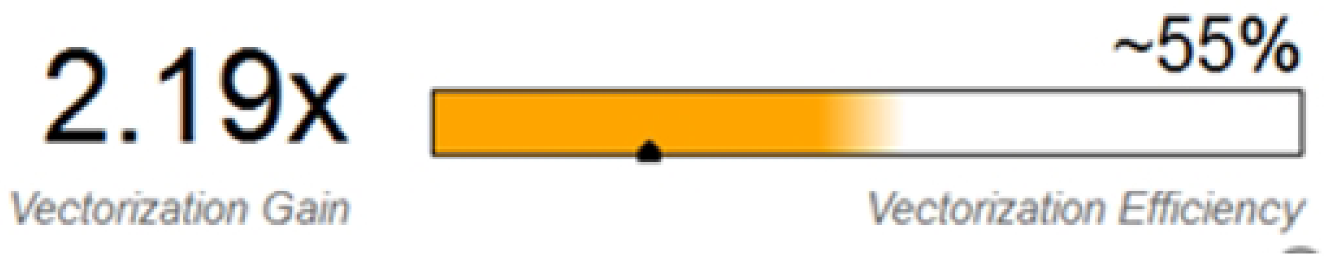
| | | |
|---|---|---|
| Total CPU time | 9.27s | 100.0% |
| Time in 2 vectorized loops | 8.11s | 87.6% |
| Time in scalar code | 1.15s | 12.4% |

Vectorization Gain/Efficiency

| | | |
|---|---|---|
| Vectorized Loops Gain/Efficiency | 2.64x | ~66% |
| Program Theoretical Gain | 2.43x | |

1 **Intel® Advisor Summary view**

You'll also need to systematically investigate the loops in your program that are taking the most time. A key metric for this is Vectorization Efficiency (**Figure 2**). In this example, Intel Advisor is showing a vectorization gain of 2.19x. But this only gives us a vectorization efficiency score of 55%. Where did we lose 45% of our efficiency? There are many factors that can cause inefficient vectorization.



2 **Intel Advisor Vectorization Efficiency view**

## Performance Problems

### Bad Access Pattern

Indirect memory access is a common cause of slowdowns. Notice in the following code snippet that we can't decode `A` without first decoding `B[i]`:

```
for (i = 0; i < N; i++)
      A[B[i]] = C[i] * D[i];
```

This gives us an irregular access pattern. The compiler can often vectorize this by using a technique called gather/scatter—which is great because it allows that loop to vectorize, but bad because these gather/scatter instructions aren't as fast as sequential access. For fast code, it's important to try to have your data structures arranged so that data is accessed in unit stride. (You'll see how Intel Advisor can show you this information later.)

### Memory Subsystem Latency/Throughput

Getting your code to fit into the various memory caches, and making optimal use of data reuse, are crucial to getting the best performance out of your system. In the following example we're indexing by `i` over a very large piece of data. This data is too big to fit in cache, which is bad—and made doubly so by `A` being a multidimensional array:

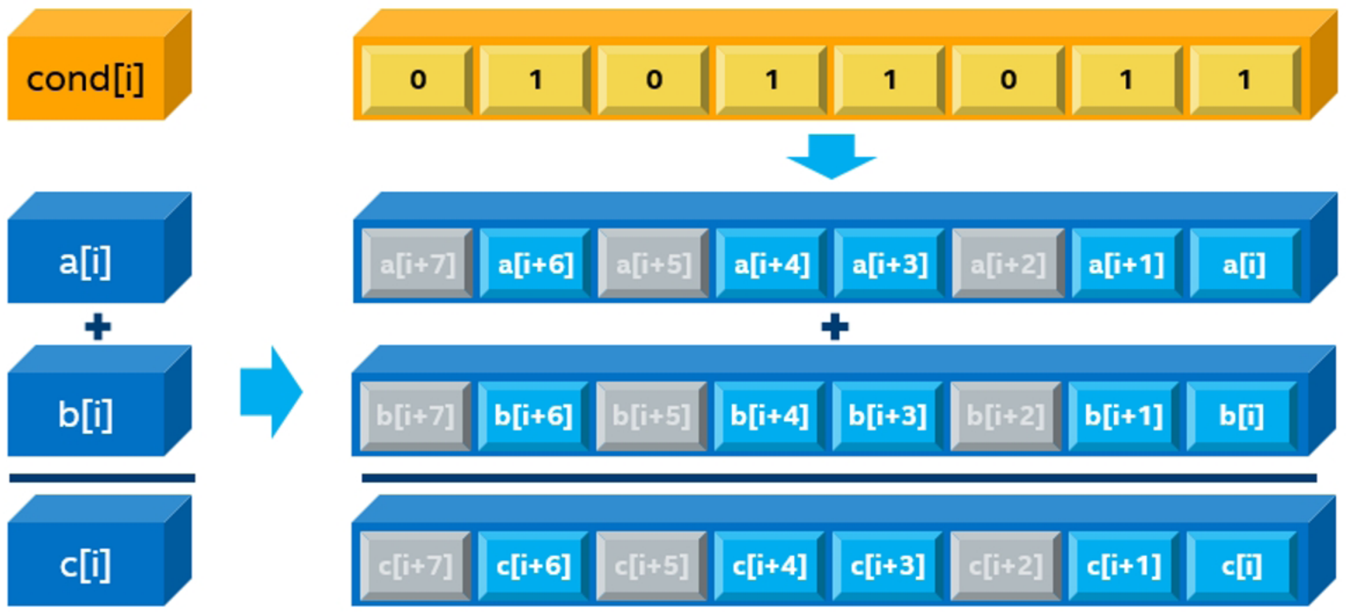
```
for (i = 0; i < VERY_BIG; i++)
      C[i] = z * A[i][j];
```

References to `A[i][j]` and `A[i+1][j]` are not located next to each other in memory. So, to get each new reference, we need to bring in a new cache line—and potentially evict a cache line. This "cache thrashing" will have a negative impact on performance. Techniques such as cache blocking, where we add a new inner loop that indexes over a much smaller range that is designed to fit in cache, can help optimize these types of applications.

### Branchy Code

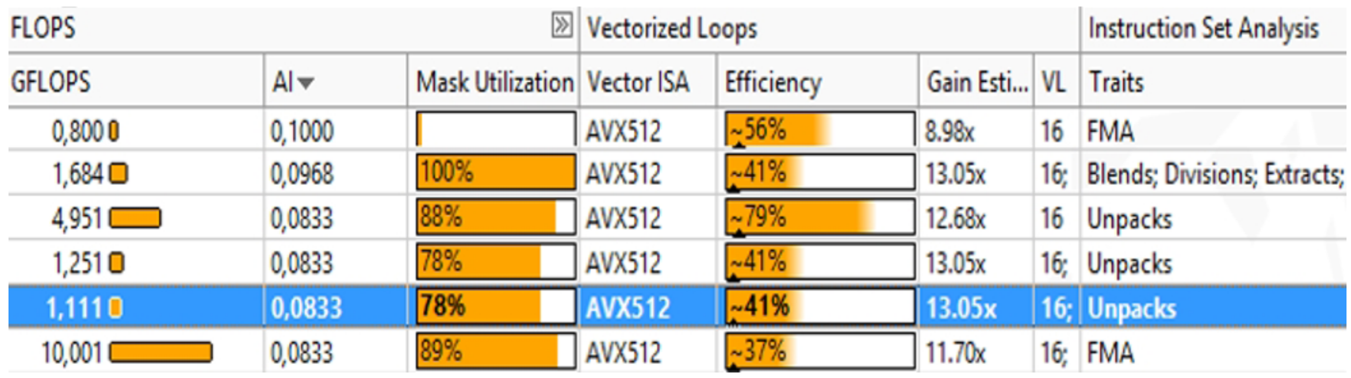
Applications with a lot of branches (e.g., the `for` loop below with the `if(cond(i))` can be vectorized using mask registers to block the SIMD lanes where the condition is not true. In these iterations, a SIMD lane does not do useful work. Intel Advisor uses the Mask utilization metric (**Figure 3**). Three elements are being suppressed, giving us a Mask utilization of 5/8 = 62.5%.

```
for (i=0; i< MAX; i++)
      if (cond(i))
            C[i] = A[i] + B[i];
```



**3** **Mask utilization metric**

You could potentially access your data in a unit stride fashion and have excellent vector efficiency, but still not get the performance you need because of low mask utilization (**Figure 4**). **Table 1** shows Memory access types.



| FLOPS | | | Vectorized Loops | | | | Instruction Set Analysis |
|---|---|---|---|---|---|---|---|
| GFLOPS | AI | Mask Utilization | Vector ISA | Efficiency | Gain Esti... | VL | Traits |
| 0,800 | 0,1000 | | AVX512 | ~56% | 8.98x | 16 | FMA |
| 1,684 | 0,0968 | 100% | AVX512 | ~41% | 13.05x | 16; | Blends; Divisions; Extracts; |
| 4,951 | 0,0833 | 88% | AVX512 | ~79% | 12.68x | 16 | Unpacks |
| 1,251 | 0,0833 | 78% | AVX512 | ~41% | 13.05x | 16; | Unpacks |
| 1,111 | 0,0833 | 78% | AVX512 | ~41% | 13.05x | 16; | Unpacks |
| 10,001 | 0,0833 | 89% | AVX512 | ~37% | 11.70x | 16; | FMA |

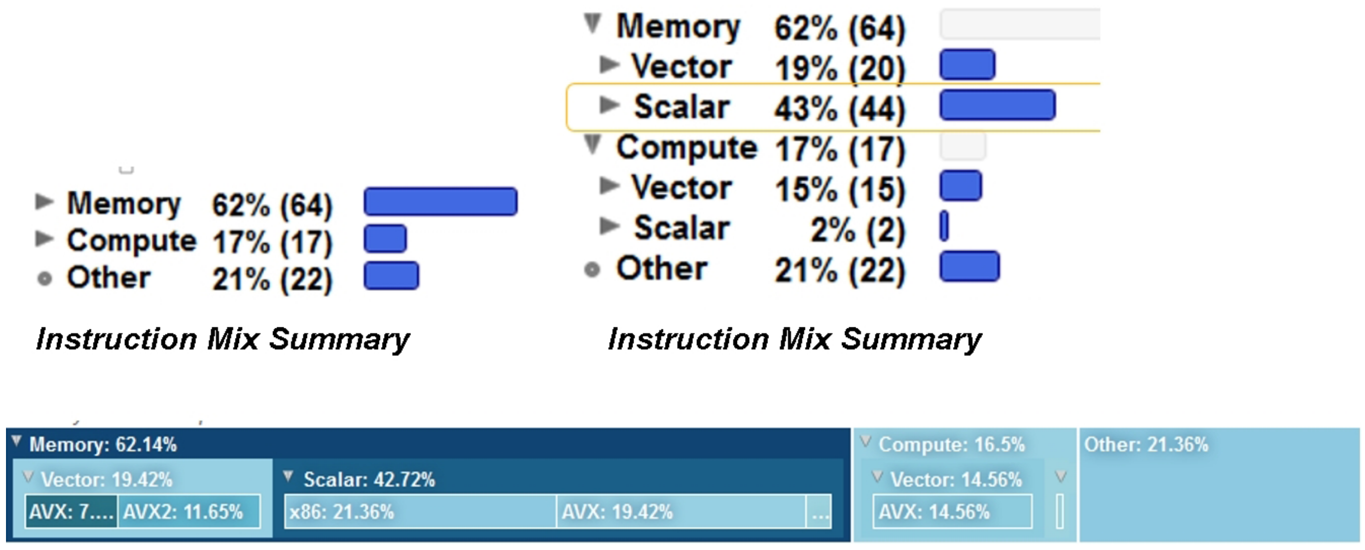
**4** **Mask utilization versus efficiency**

**Table 1. Memory access types**

| Access Pattern | Small Memory Footprint | Large Memory Footprint |
|---|---|---|
| **Unit Stride** | • **Effective SIMD**<br>• **No latency or bandwidth bottlenecks** | • **Effective SIMD**<br>• Bandwidth bottleneck |
| **Constant Stride** | • Medium SIMD<br>• Latency bottlenecks possible | • Medium SIMD<br>• Latency and bottlenecks possible |
| **Irregular Access, Gather/Scatter** | • Bad SIMD<br>• Latency bottlenecks possible | • Bad SIMD<br>• Latency Bottlenecks |

## Are You Bound by CPU/VPU or Memory?

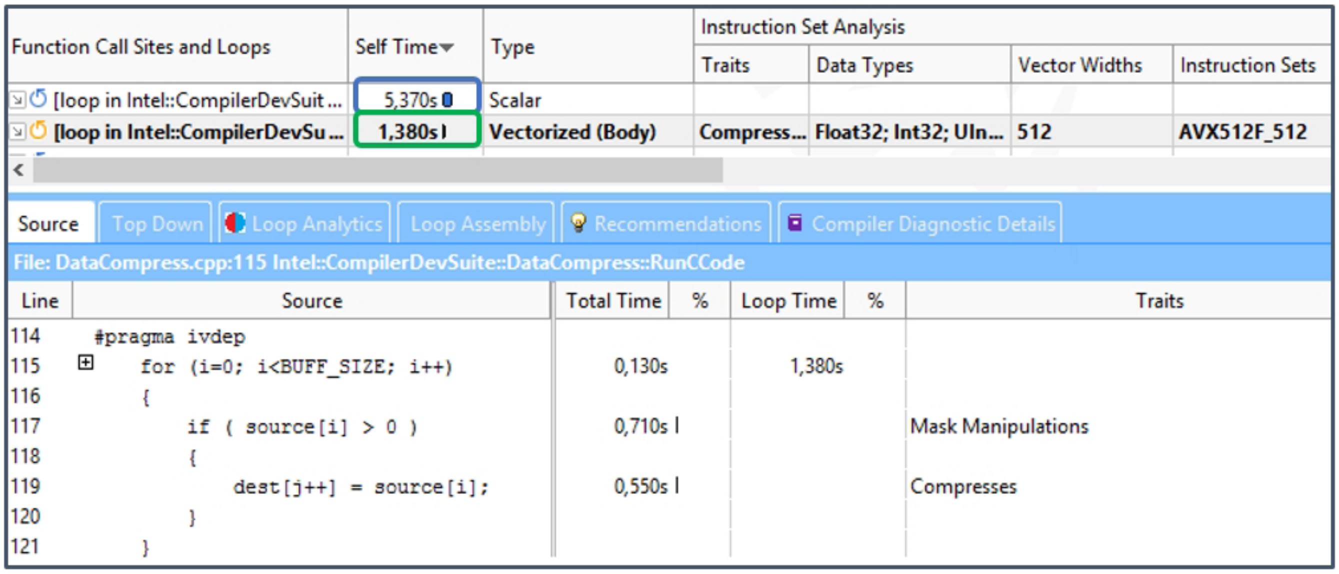
If your application is memory bound, there are several features in Intel Advisor that can help you optimize. But first, you need to determine if you're memory bound or CPU/VPU bound. A quick way to determine this is by looking at your instructions. The Intel Advisor Code Analytics windows (**Figure 5**) can give you a very basic way to see the mix of instructions that you're code is executing.



**5** **Code Analytics windows**

A good rule of thumb is that applications that are executing a lot of memory instructions tend to be memory bound, whereas those that are executing a lot of compute instructions tend to be compute bound. Notice the breakdown in **Figure 5**. The ratio of scalar to vector instructions is particularly important. You should try to have as many vector instructions as possible.

Another, slightly more complicated, technique is to use the Traits column in the Intel Advisor Survey view (**Figure 6**).



| Function Call Sites and Loops | Self Time | Type | Traits | Data Types | Vector Widths | Instruction Sets |
|---|---|---|---|---|---|---|
| [loop in Intel::CompilerDevSuit... | 5,370s | Scalar | | | | |
| **[loop in Intel::CompilerDevSu...** | **1,380s** | **Vectorized (Body)** | **Compress...** | **Float32; Int32; UIn...** | **512** | **AVX512F_512** |

File: DataCompress.cpp:115 Intel::CompilerDevSuite::DataCompress::RunCCode

| Line | Source | Total Time | % | Loop Time | % | Traits |
|---|---|---|---|---|---|---|
| 114 | `#pragma ivdep` | | | | | |
| 115 | `for (i=0; i<BUFF_SIZE; i++)` | 0,130s | | 1,380s | | |
| 116 | `{` | | | | | |
| 117 | `if ( source[i] > 0 )` | 0,710s | | | | Mask Manipulations |
| 118 | `{` | | | | | |
| 119 | `dest[j++] = source[i];` | 0,550s | | | | Compresses |
| 120 | `}` | | | | | |
| 121 | `}` | | | | | |

**6** **Traits column in the Intel Advisor Survey view**

Think of Traits as what the compiler needed to do to vectorize your loop. In the latest vector instructions sets, such as **Intel® AVX-512**, there are many new instructions and idioms the compiler can use to vectorize your code. Techniques like register masking and compress instructions, shown in **Figure 6**, do allow applications to vectorize when this was not previously possible—but sometimes at a cost. Anything the compiler needed to do to get your data structures to fit in a vector (such as memory manipulation) will often appear in the Traits column. These Traits often indicate a problem that you can explore with Memory Access Pattern analysis.

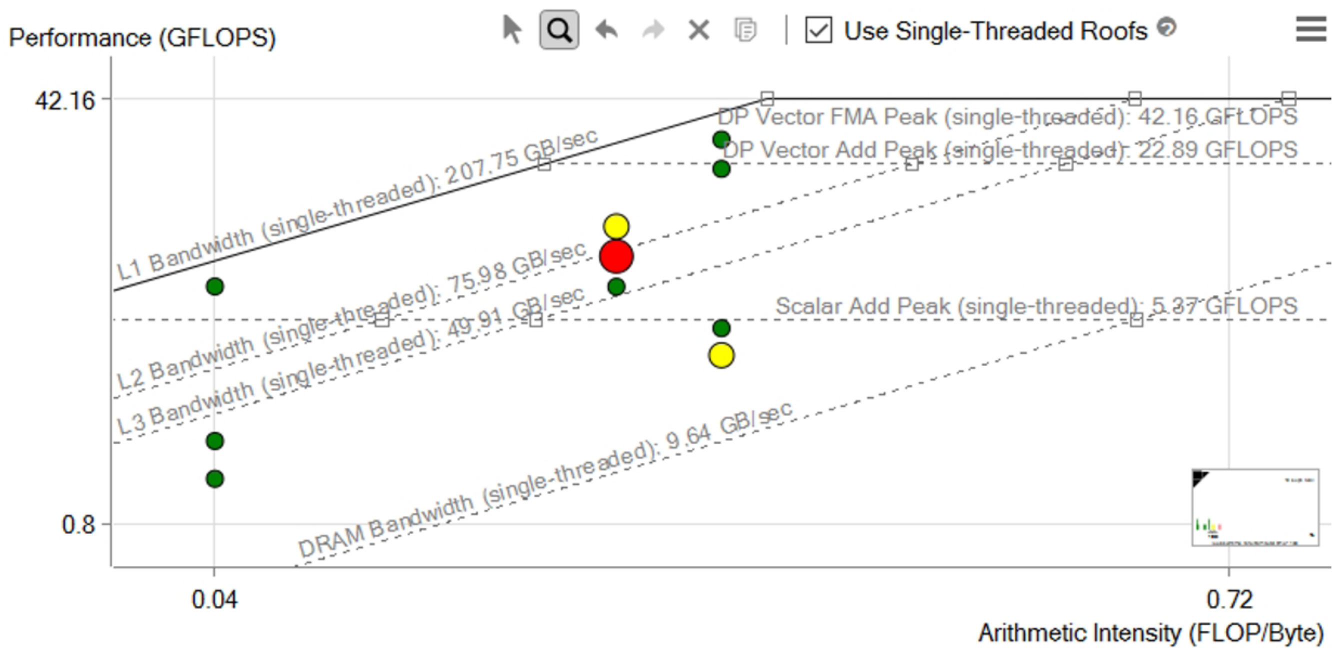
## Helpful Optimization Features

### Roofline Analysis

A Roofline chart is a visual representation of application performance in relation to hardware limitations, including memory bandwidth and computational peaks. It was first proposed by researchers at the University of California at Berkeley in the 2008 paper "**Roofline: An Insightful Visual Performance Model for Multicore Architectures**." In 2014, this model was extended by researchers at the Technical University of Lisbon in a paper called "**Cache-Aware Roofline Model: Upgrading the Loft**." Traditionally, Roofline charts have been calculated and plotted manually. But Intel Advisor now automatically builds Roofline plots.

The Roofline provides insight into:

- **Where** your performance bottlenecks are
- **How much** performance is left on the table because of them
- **Which** bottlenecks are possible to address, and which ones are worth addressing
- **Why** these bottlenecks are most likely occurring
- **What** your next steps should be

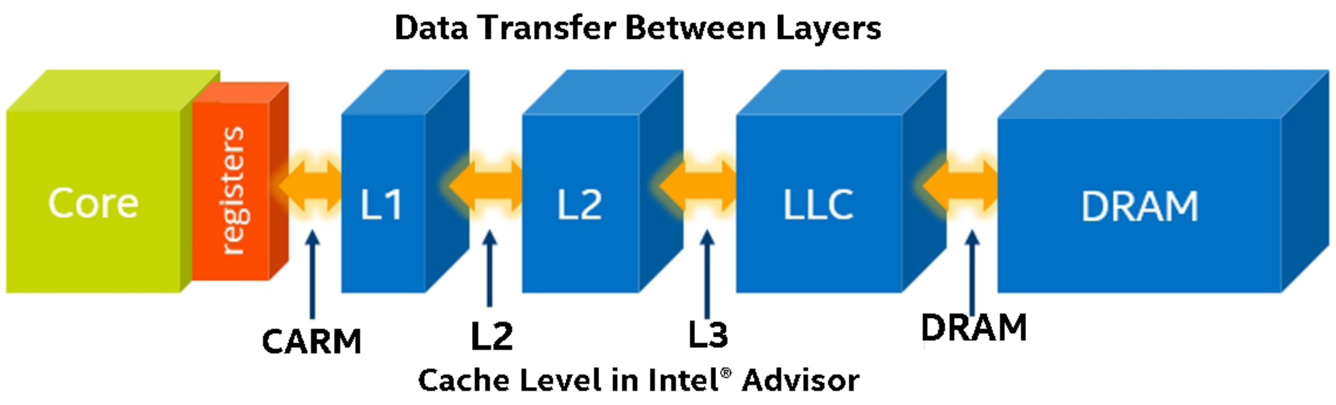


7 **Roofline analysis**

The horizontal lines in **Figure 7** represent the number of floating-point or integer computations of a given type your hardware can perform in a given span of time. The diagonal lines represent how many bytes of data a given memory subsystem can deliver per second. Each dot is a loop or function in your program, with its position indicating its performance, which is affected by its optimization and its arithmetic intensity (AI).

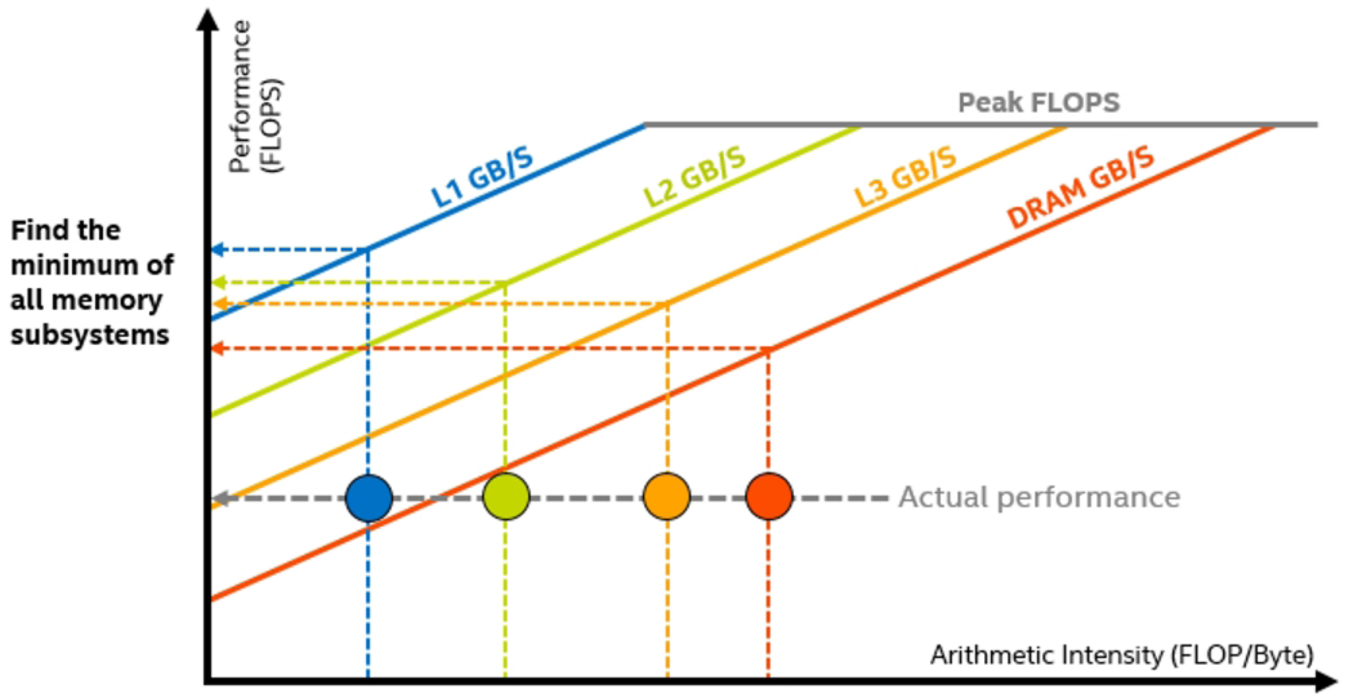
## Intel Advisor Integrated Roofline

The Integrated Roofline model offers a more detailed analysis, showing directly where the bottleneck comes from. Intel Advisor collects data for all memory types using cache simulation (**Figure 8**).



**8** **Cache simulation in Intel Advisor**

With this data, Intel Advisor counts the number of data transfers for a given cache level and computes the specific AI for each loop and each memory level. By observing the changes in this traffic from one level to another, and then comparing it to respective roofs representing the best possible bandwidths for these levels, it's possible to pinpoint the memory hierarchy bottleneck for the kernel and determine optimization steps (**Figure 9**).

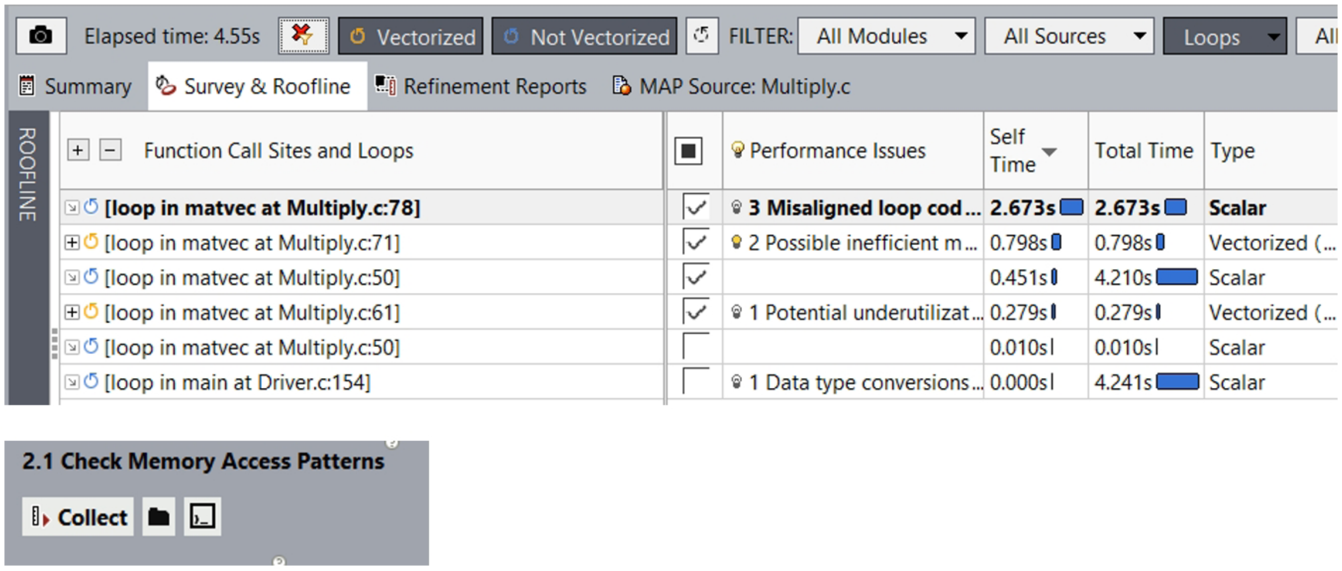


**9** **Pinpointing the memory hierarchy bottleneck**

## Memory Access Pattern (MAP) Analysis

Intel Advisor MAP analysis gives you the deepest insight into how you're accessing memory. Your memory access pattern affects both the efficiency of vectorization as well as how much memory bandwidth you can ultimately achieve. The MAP collection observes data accesses during execution and spots the instructions that contain the memory accesses. The data collected and analyzed appears in the Memory Access Patterns Report tab of the Refinement Reports window.

To run a MAP analysis from the GUI (**Figure 10**), you need to select loops using the checkboxes in the Survey report and run a MAP collection.



| Function Call Sites and Loops | | Performance Issues | Self Time | Total Time | Type |
|---|---|---|---|---|---|
| **[loop in matvec at Multiply.c:78]** | ✓ | **3 Misaligned loop cod...** | **2.673s** | **2.673s** | **Scalar** |
| [loop in matvec at Multiply.c:71] | ✓ | 2 Possible inefficient m... | 0.798s | 0.798s | Vectorized (... |
| [loop in matvec at Multiply.c:50] | ✓ | | 0.451s | 4.210s | Scalar |
| [loop in matvec at Multiply.c:61] | ✓ | 1 Potential underutilizat... | 0.279s | 0.279s | Vectorized (... |
| [loop in matvec at Multiply.c:50] | | | 0.010s | 0.010s | Scalar |
| [loop in main at Driver.c:154] | | 1 Data type conversions... | 0.000s | 4.241s | Scalar |

**10** **Memory Access Patterns report**

You can also run a MAP collection from the command-line. Use the `-mark-up-list` option to select loops to be analyzed.

```
advixe-cl -collect map -mark-up-
list=Multiply.c:78,Multiply.c:71,Multiply.c:50,Multiply.c:6
1 -project-dir C:/my_advisor_project -- my_application.exe
```

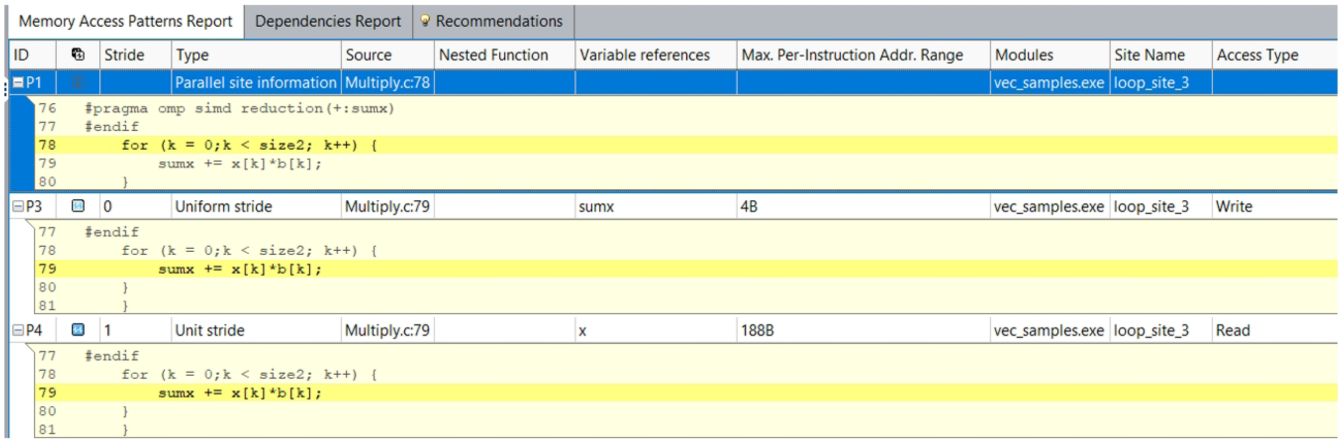
The Memory Access Patterns report provides information about the types of strides observed in the memory access operations during loop execution. The tool reports both unit/uniform and constant strides (**Figure 11**).

## Unit/Uniform Stride Types

- **Unit stride (stride 1)** instruction accesses memory that consistently changes by one element from iteration to iteration.
- **Uniform stride 0** instruction accesses the same memory from iteration to iteration.
- **Constant stride (stride N)** instruction accesses memory that consistently changes by N elements (N>1) from iteration to iteration.

## Variable Stride Types

- **Irregular stride** instruction accesses memory addresses that change by an unpredictable number of elements from iteration to iteration.
- **Gather (irregular) stride** is detected for `v(p)gather*` instructions on AVX2 Instruction Set Architecture.


Memory Access Patterns Report | Dependencies Report | Recommendations

| ID | | Stride | Type | Source | Nested Function | Variable references | Max. Per-Instruction Addr. Range | Modules | Site Name | Access Type |
|---|---|---|---|---|---|---|---|---|---|---|
| P1 | | | Parallel site information | Multiply.c:78 | | | | vec_samples.exe | loop_site_3 | |
| P3 | | 0 | Uniform stride | Multiply.c:79 | | sumx | 4B | vec_samples.exe | loop_site_3 | Write |
| P4 | | 1 | Unit stride | Multiply.c:79 | | x | 188B | vec_samples.exe | loop_site_3 | Read |

```
#pragma omp simd reduction(+:sumx)
#endif
        for (k = 0;k < size2; k++) {
            sumx += x[k]*b[k];
        }
```

```
#endif
        for (k = 0;k < size2; k++) {
            sumx += x[k]*b[k];
        }
    }
```

```
#endif
        for (k = 0;k < size2; k++) {
            sumx += x[k]*b[k];
        }
    }
```

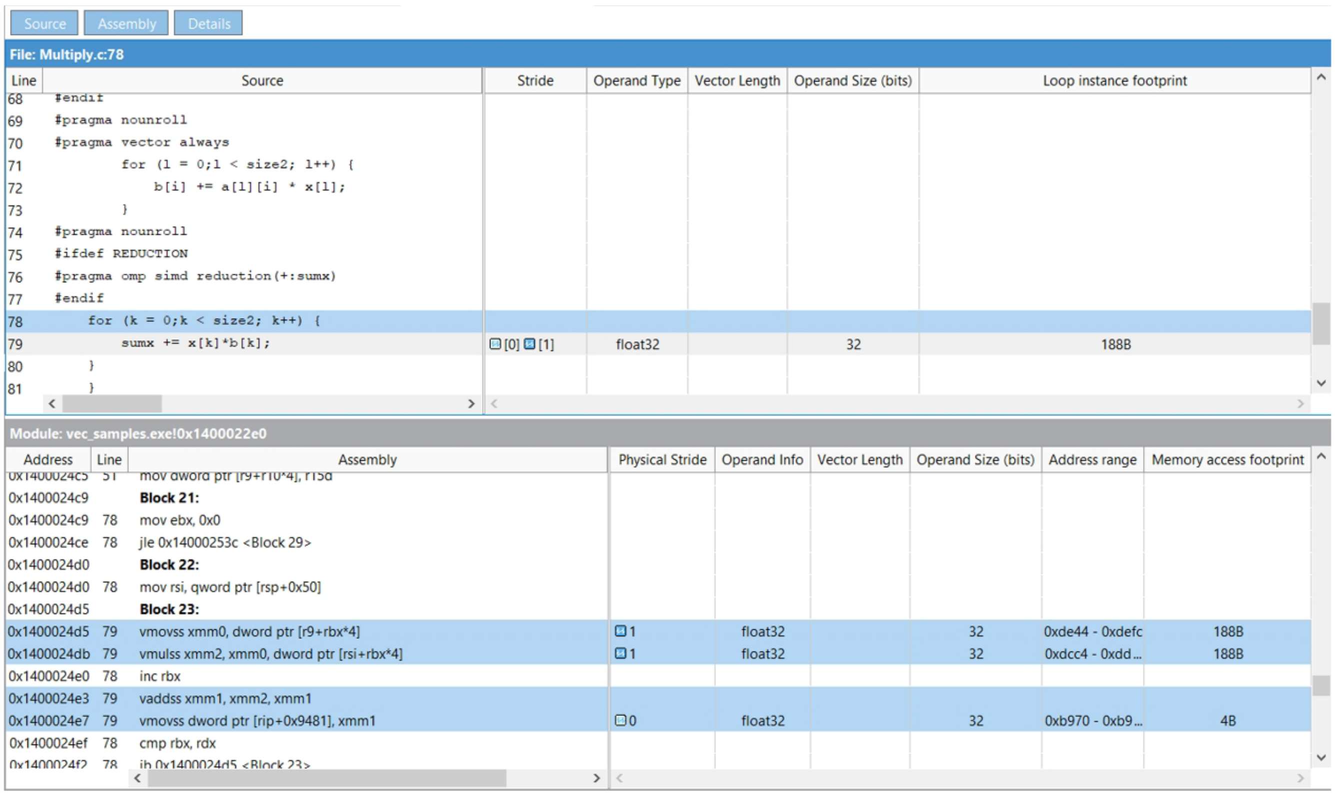
**11** **Stride types**

Double-click any line in the Memory Access Patterns report tab to see the selected operation's source code (**Figure 12**).

The Source and Details views (**Figure 13**) both give you insights into another key Intel Advisor memory feature, Memory Footprint.

## Memory Footprint Analysis

Memory Footprint is basically the range of memory a given loop accesses. This footprint can be a key indicator of your memory bandwidth. If the range is very large, then you might not be able to fit in cache. Optimization strategies such as cache blocking can make a big difference in these cases. Intel Advisor has three different memory footprint metrics (**Figure 14**).



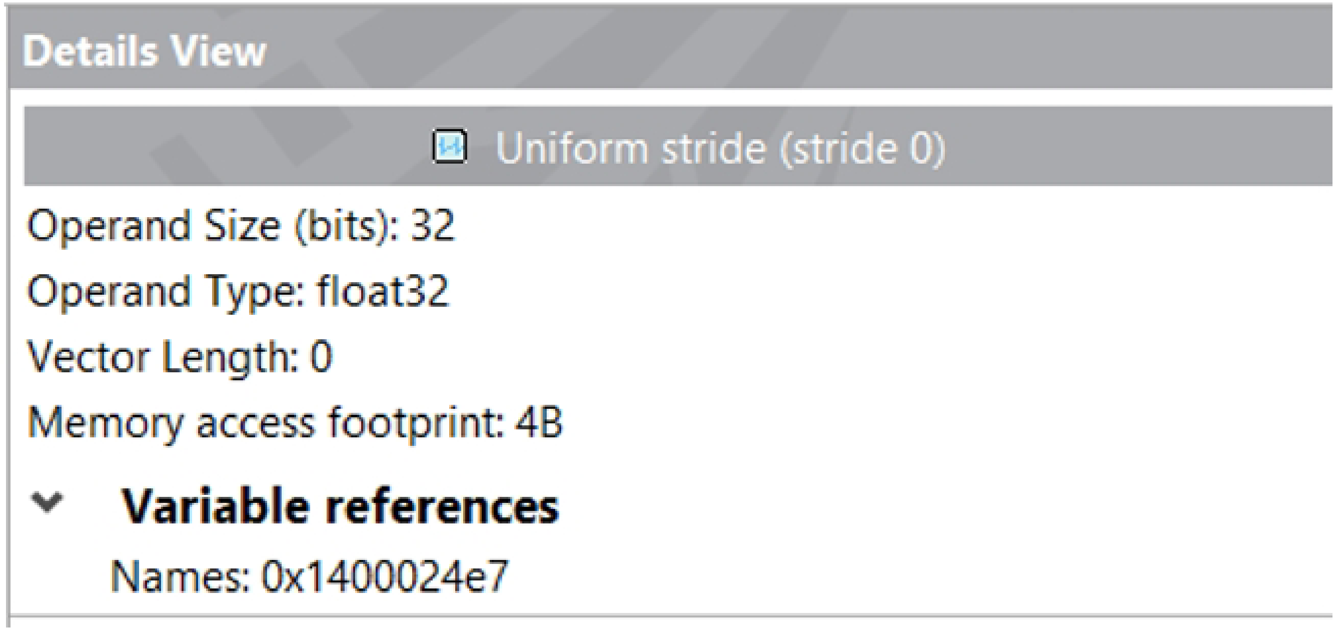
File: Multiply.c:78

| Line | Source | Stride | Operand Type | Vector Length | Operand Size (bits) | Loop instance footprint |
|---|---|---|---|---|---|---|
| 68 | `#endif` | | | | | |
| 69 | `#pragma nounroll` | | | | | |
| 70 | `#pragma vector always` | | | | | |
| 71 | `for (l = 0;l < size2; l++) {` | | | | | |
| 72 | `b[i] += a[l][i] * x[l];` | | | | | |
| 73 | `}` | | | | | |
| 74 | `#pragma nounroll` | | | | | |
| 75 | `#ifdef REDUCTION` | | | | | |
| 76 | `#pragma omp simd reduction(+:sumx)` | | | | | |
| 77 | `#endif` | | | | | |
| 78 | `for (k = 0;k < size2; k++) {` | | | | | |
| 79 | `sumx += x[k]*b[k];` | [0] [1] | float32 | | 32 | 188B |
| 80 | `}` | | | | | |
| 81 | `}` | | | | | |

Module: vec_samples.exe!0x1400022e0

| Address | Line | Assembly | Physical Stride | Operand Info | Vector Length | Operand Size (bits) | Address range | Memory access footprint |
|---|---|---|---|---|---|---|---|---|
| 0x1400024c5 | 51 | mov dword ptr [r9+r10*4], r15d | | | | | | |
| 0x1400024c9 | | **Block 21:** | | | | | | |
| 0x1400024c9 | 78 | mov ebx, 0x0 | | | | | | |
| 0x1400024ce | 78 | jle 0x14000253c <Block 29> | | | | | | |
| 0x1400024d0 | | **Block 22:** | | | | | | |
| 0x1400024d0 | 78 | mov rsi, qword ptr [rsp+0x50] | | | | | | |
| 0x1400024d5 | | **Block 23:** | | | | | | |
| 0x1400024d5 | 79 | vmovss xmm0, dword ptr [r9+rbx*4] | 1 | float32 | | 32 | 0xde44 - 0xdefc | 188B |
| 0x1400024db | 79 | vmulss xmm2, xmm0, dword ptr [rsi+rbx*4] | 1 | float32 | | 32 | 0xdcc4 - 0xdd... | 188B |
| 0x1400024e0 | 78 | inc rbx | | | | | | |
| 0x1400024e3 | 79 | vaddss xmm1, xmm2, xmm1 | | | | | | |
| 0x1400024e7 | 79 | vmovss dword ptr [rip+0x9481], xmm1 | 0 | float32 | | 32 | 0xb970 - 0xb9... | 4B |
| 0x1400024ef | 78 | cmp rbx, rdx | | | | | | |
| 0x1400024f2 | 78 | jb 0x1400024d5 <Block 23> | | | | | | |

**12** **See the selected operation's source code**


Details View

Uniform stride (stride 0)

Operand Size (bits): 32

Operand Type: float32

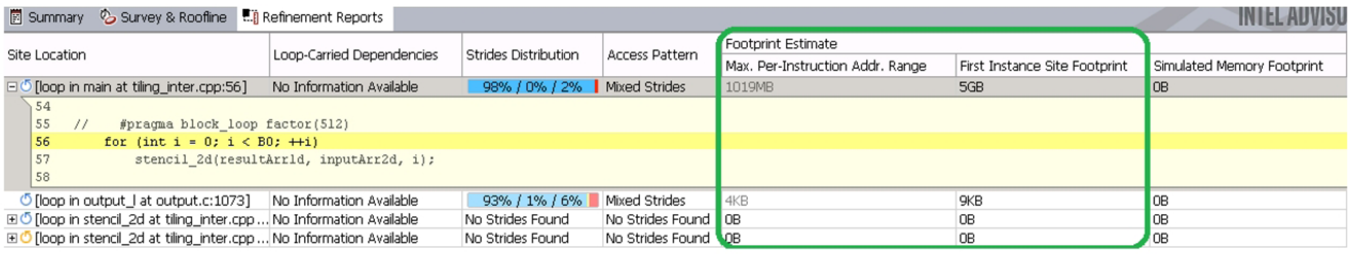
Vector Length: 0

Memory access footprint: 4B

**Variable references**

Names: 0x1400024e7

**13** **Details view**



Summary | Survey & Roofline | Refinement Reports

| Site Location | Loop-Carried Dependencies | Strides Distribution | Access Pattern | Footprint Estimate | | |
|---|---|---|---|---|---|---|
| | | | | Max. Per-Instruction Addr. Range | First Instance Site Footprint | Simulated Memory Footprint |
| [loop in main at tiling_inter.cpp:56] | No Information Available | 98% / 0% / 2% | Mixed Strides | 1019MB | 5GB | 0B |
| [loop in output_l at output.c:1073] | No Information Available | 93% / 1% / 6% | Mixed Strides | 4KB | 9KB | 0B |
| [loop in stencil_2d at tiling_inter.cpp ... | No Information Available | No Strides Found | No Strides Found | 0B | 0B | 0B |
| [loop in stencil_2d at tiling_inter.cpp ... | No Information Available | No Strides Found | No Strides Found | 0B | 0B | 0B |

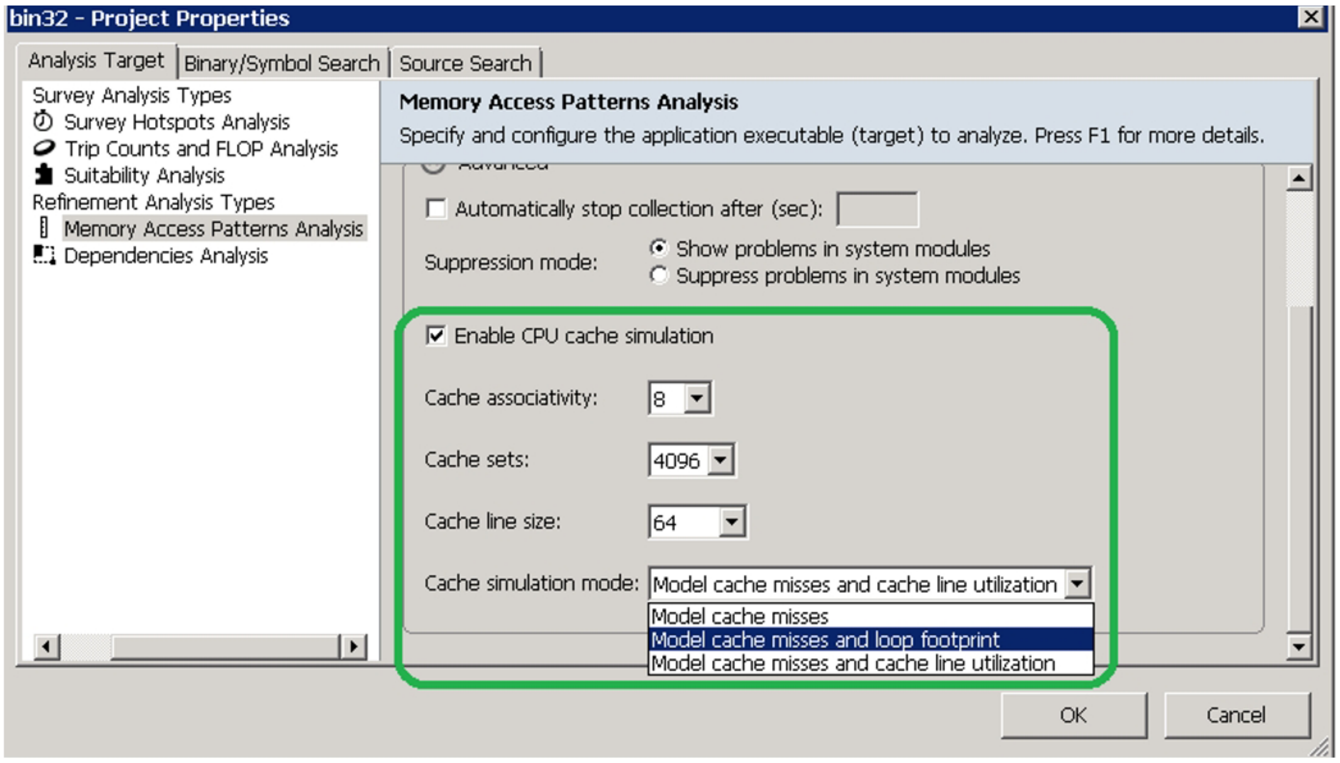
```

//    #pragma block_loop factor(512)
    for (int i = 0; i < B0; ++i)
        stencil_2d(resultArr1d, inputArr2d, i);

```

**14** **Memory footprint metrics**

Two basic footprint metrics represent just some aspects of your memory footprint. These metrics are collected by default in Memory Access Pattern (MAP) analysis:

- **Max Per-Instruction Address Range** represents the maximum distance between minimum and maximum memory address values accessed by instructions in this loop. For each memory access instruction, the minimum and maximum address of its access is recorded and the maximum range of this address for all loop instructions is reported. It covers more than one loop instance, with some filtering, which is why sometimes Intel Advisor is less confident in this metric and reports it in gray.
- **First Instance Site Footprint** is a more accurate memory footprint, since it's aware of overlaps in address ranges in the loop iterations and gaps between address ranges accessed by the loop, but is calculated only for the first instance (call) of this loop.

There's a more advanced footprint calculated based on cache simulation, called the Simulated Memory Footprint. This metric shows the summarized and overlap-aware picture across all loop instances, but limited to one thread. It is calculated as the number of unique cache lines accessed during cache simulation multiplied by cache line size. To enable it in the GUI, select the Enable CPU cache simulation checkbox in the Memory Access Patterns tab of the Project Properties, and select Model Cache Misses and Loop Footprint Simulation Mode in the dropdown list (**Figure 15**). Then select the loops of interest with the checkboxes in the Survey view and run a MAP analysis.



Memory Access Patterns Analysis
Specify and configure the application executable (target) to analyze. Press F1 for more details.

☐ Automatically stop collection after (sec):

Suppression mode: [✓ Show problems in system modules] Suppress problems in system modules

☑ Enable CPU cache simulation

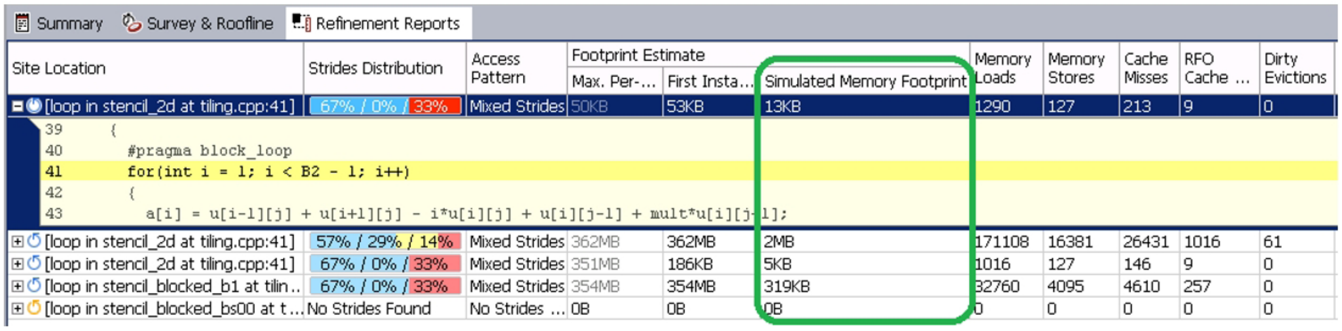
Cache associativity: 8
Cache sets: 4096
Cache line size: 64
Cache simulation mode: Model cache misses and cache line utilization
- Model cache misses
- [✓ Model cache misses and loop footprint]
- Model cache misses and cache line utilization

**15** **Simulated Memory Footprint**

To enable in the command-line, you need to use the MAP command, as previously specified, with these options: `-enable-cache-simulation` and `-cachesim-mode=footprint`.

```
advixe-cl -collect map -mark-up-
list=tiling_inter.cpp:56,output.c:1073 -enable-cache-
simulation -cachesim-mode=footprint -project-dir
C:\my_advisor_project -- my_application.exe
```

You can see the results of the analysis in the Intel Advisor GUI Refinement Report view (**Figure 16**). The more detailed cache-related metrics—like the total number of memory loads, stores, cache misses, and cache-simulated memory footprint—allow a more detailed study of loop behavior with respect to memory. **Table 2** shows Intel Advisor footprint metrics applicability, limitations, and relevance for analyzing different types of codes.



| Site Location | Strides Distribution | Access Pattern | Footprint Estimate |  |  | Memory Loads | Memory Stores | Cache Misses | RFO Cache … | Dirty Evictions |
|---|---|---|---|---|---|---|---|---|---|---|
|  |  |  | Max. Per-… | First Insta… | Simulated Memory Footprint |  |  |  |  |  |
| [loop in stencil_2d at tiling.cpp:41] | 67% / 0% / 33% | Mixed Strides | 50KB | 53KB | 13KB | 1290 | 127 | 213 | 9 | 0 |
| [loop in stencil_2d at tiling.cpp:41] | 57% / 29% / 14% | Mixed Strides | 362MB | 362MB | 2MB | 171108 | 16381 | 26431 | 1016 | 61 |
| [loop in stencil_2d at tiling.cpp:41] | 67% / 0% / 33% | Mixed Strides | 351MB | 186KB | 5KB | 1016 | 127 | 146 | 9 | 0 |
| [loop in stencil_blocked_b1 at tilin.. | 67% / 0% / 33% | Mixed Strides | 354MB | 354MB | 319KB | 32760 | 4095 | 4610 | 257 | 0 |
| [loop in stencil_blocked_bs00 at t… | No Strides Found | No Strides … | 0B | 0B | 0B | 0 | 0 | 0 | 0 | 0 |

```
    {
      #pragma block_loop
      for(int i = 1; i < B2 - 1; i++)
      {
        a[i] = u[i-1][j] + u[i+1][j] - i*u[i][j] + u[i][j-1] + mult*u[i][j-1];
```

**16** **Intel Advisor GUI Refinement Report view**

**Table 2. Intel Advisor footprint metrics**

|  | Max Per-Instruction Address Range | First Instance Site Footprint | Simulated Memory Footprint |
|---|---|---|---|
| Threads analyzed for the loop/site | 1 | 1 | 1 |
| Loop instances analyzed | All instances, but with some shortcuts | 1, only first instance | Depends on loop-call-count limit option |
| Aware of address range overlap? | No | Yes | Yes |
| Suitable for codes with random memory access | No | No | Yes |

## A Real-World Example

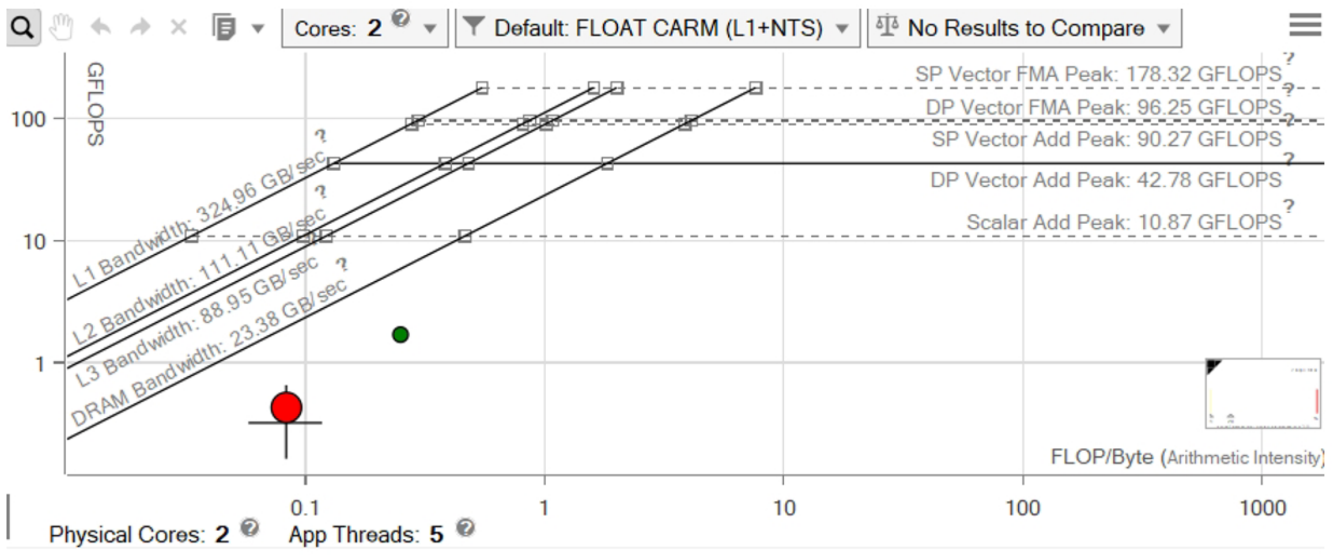
Some of the most common problems in computational science require matrix multiplication. The list of domains that use matrices is almost endless, but artificial intelligence, simulation, and modeling are just a few examples. The sample algorithm below is a triply nested loop where we do a multiply and an add for each iteration. Besides being very computationally intensive, it also accesses a lot of memory. Let's use Intel Advisor to see how much.

```
for(i=0; i<msize; i++) {
    for(j=0; j<msize; j++) {
        for(k=0; k<msize; k++) {
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
        }
    }
}
```

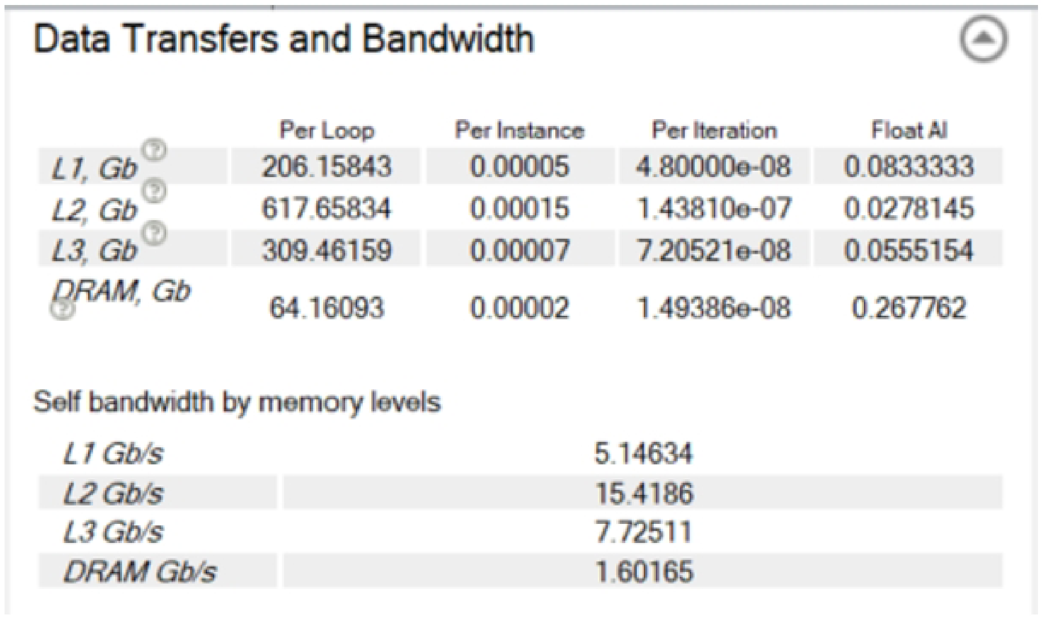
### Create a Baseline

The elapsed time was 53.94 seconds for our initial run. **Figure 17** is a Cache-aware Roofline chart. The red dot is our main computational loop. It's far below even DRAM bandwidth, and even farther below L1 bandwidth, which is the maximum bandwidth we're trying to achieve. You can see the precise bandwidth we're achieving at each level of the memory hierarchy using the Code Analytics tab for the loop (**Figure 18**).

Why is our performance so poor? How can we do better? These are questions Intel Advisor was designed to answer. First, we need to examine the Survey view (**Figure 19**) to see what's going on and whether Intel Advisor has any recommendations. Intel Advisor has noted that we have an Inefficient memory access pattern, and also that the loop has not been vectorized because of an assumed dependency. To examine the memory access pattern, we can run a Memory Access Pattern (MAP) analysis (**Figure 20**).



**17** **Cache-aware Roofline chart**



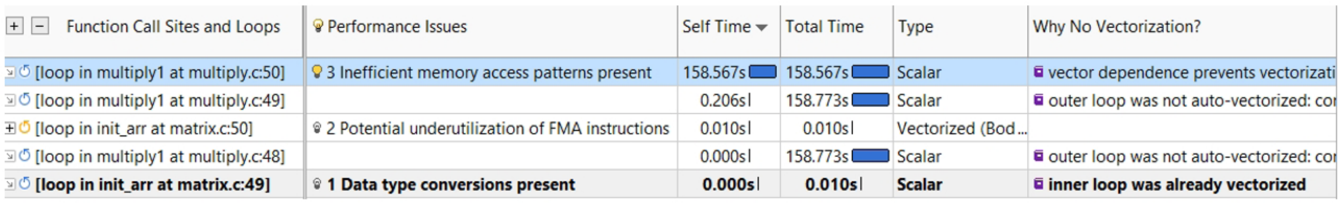
Data Transfers and Bandwidth

| | Per Loop | Per Instance | Per Iteration | Float AI |
|---|---|---|---|---|
| L1, Gb | 206.15843 | 0.00005 | 4.80000e-08 | 0.0833333 |
| L2, Gb | 617.65834 | 0.00015 | 1.43810e-07 | 0.0278145 |
| L3, Gb | 309.46159 | 0.00007 | 7.20521e-08 | 0.0555154 |
| DRAM, Gb | 64.16093 | 0.00002 | 1.49386e-08 | 0.267762 |

Self bandwidth by memory levels

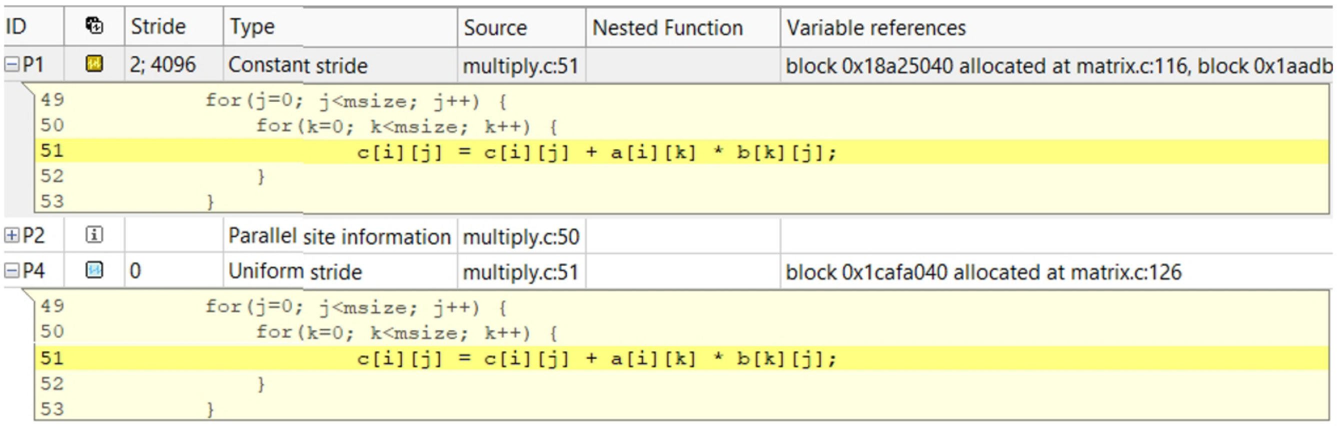
| | |
|---|---|
| L1 Gb/s | 5.14634 |
| L2 Gb/s | 15.4186 |
| L3 Gb/s | 7.72511 |
| DRAM Gb/s | 1.60165 |

**18** **Data transfers and bandwidth**



| Function Call Sites and Loops | Performance Issues | Self Time | Total Time | Type | Why No Vectorization? |
|---|---|---|---|---|---|
| [loop in multiply1 at multiply.c:50] | 3 Inefficient memory access patterns present | 158.567s | 158.567s | Scalar | vector dependence prevents vectorizati |
| [loop in multiply1 at multiply.c:49] | | 0.206s | 158.773s | Scalar | outer loop was not auto-vectorized: co |
| [loop in init_arr at matrix.c:50] | 2 Potential underutilization of FMA instructions | 0.010s | 0.010s | Vectorized (Bod… | |
| [loop in multiply1 at multiply.c:48] | | 0.000s | 158.773s | Scalar | outer loop was not auto-vectorized: co |
| **[loop in init_arr at matrix.c:49]** | **1 Data type conversions present** | **0.000s** | **0.010s** | **Scalar** | **inner loop was already vectorized** |

**19** **Survey view**



| ID | | Stride | Type | Source | Nested Function | Variable references |
|---|---|---|---|---|---|---|
| P1 | | 2; 4096 | Constant stride | multiply.c:51 | | block 0x18a25040 allocated at matrix.c:116, block 0x1aadb |

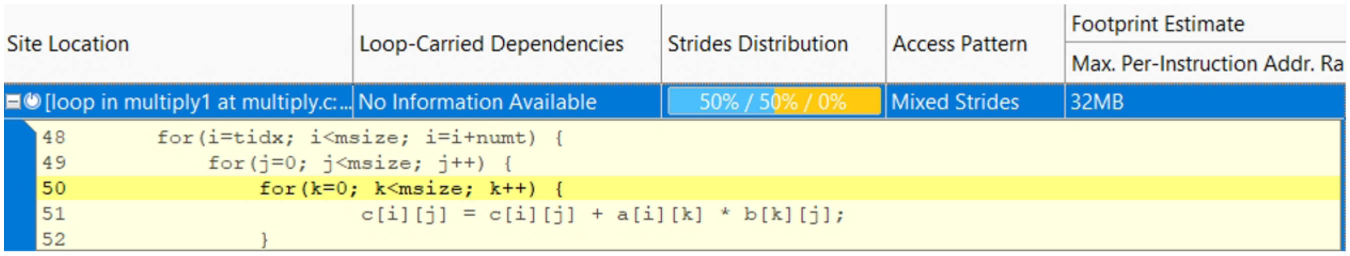
```
49          for(j=0; j<msize; j++) {
50              for(k=0; k<msize; k++) {
51                      c[i][j] = c[i][j] + a[i][k] * b[k][j];
52              }
53          }
```

| ID | | Stride | Type | Source | Nested Function | Variable references |
|---|---|---|---|---|---|---|
| P2 | | | Parallel site information | multiply.c:50 | | |
| P4 | | 0 | Uniform stride | multiply.c:51 | | block 0x1cafa040 allocated at matrix.c:126 |

```
49          for(j=0; j<msize; j++) {
50              for(k=0; k<msize; k++) {
51                      c[i][j] = c[i][j] + a[i][k] * b[k][j];
52              }
53          }
```

**20** **Memory Access Pattern (MAP) analysis**

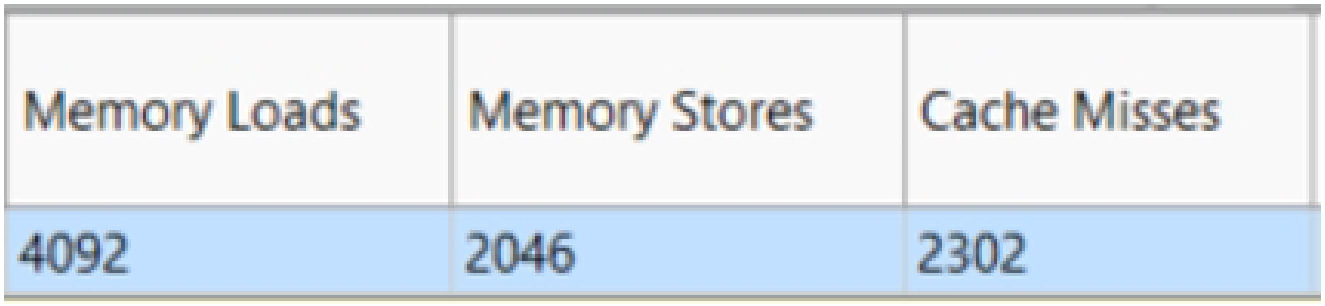
Intel Advisor has detected a constant stride on our read access and a uniform stride of 0 for the write. The range of memory we're accessing is 32MB, far bigger than any of our cache sizes (**Figure 21**). We can also see how well the caches are performing using the MAP report (**Figure 22**). We're missing over 2,300 cache lines, so it's no wonder performance is bad. But there are several ways we can fix this.



| Site Location | Loop-Carried Dependencies | Strides Distribution | Access Pattern | Footprint Estimate |
|---|---|---|---|---|
| | | | | Max. Per-Instruction Addr. Ra |
| [loop in multiply1 at multiply.c:... | No Information Available | 50% / 50% / 0% | Mixed Strides | 32MB |

```
        for(i=tidx; i<msize; i=i+numt) {
            for(j=0; j<msize; j++) {
                for(k=0; k<msize; k++) {
                        c[i][j] = c[i][j] + a[i][k] * b[k][j];
                }
```

**21** **Strides distribution**



| Memory Loads | Memory Stores | Cache Misses |
|---|---|---|
| 4092 | 2046 | 2302 |

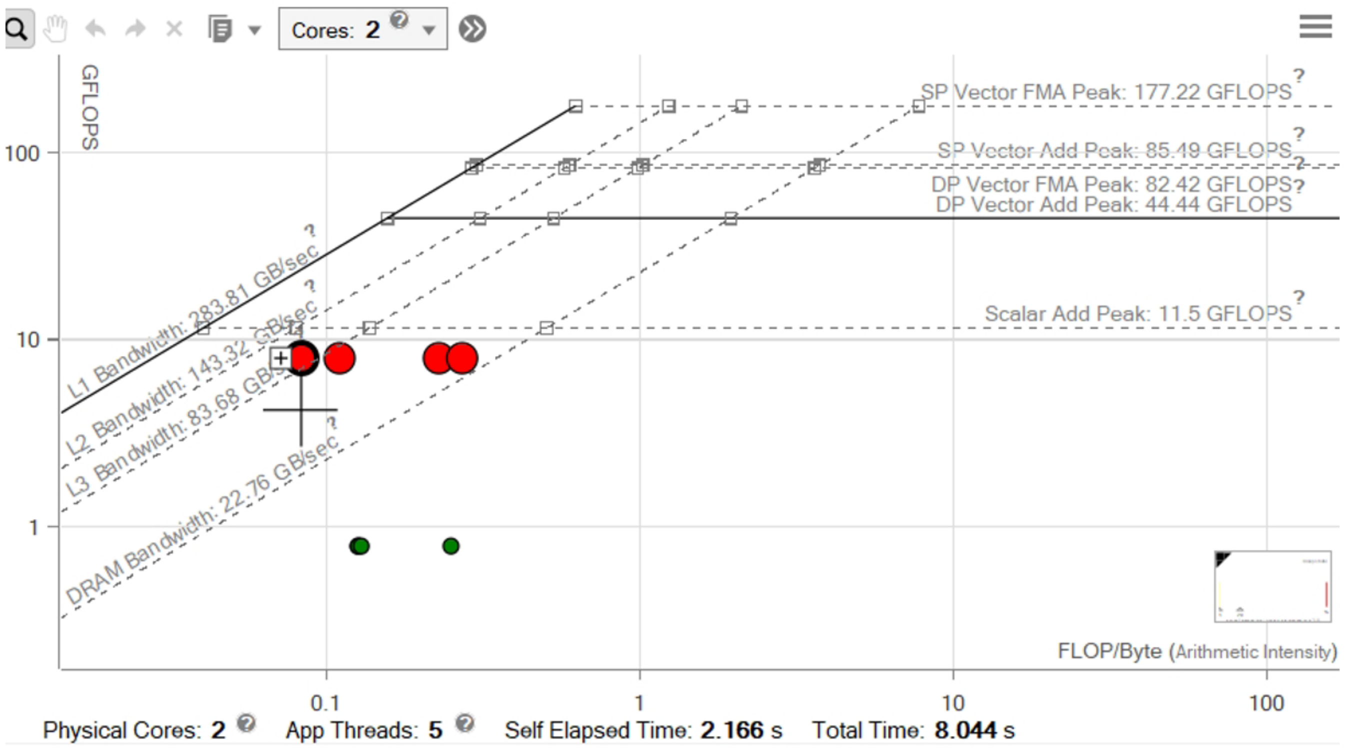
**22** **MAP report**

## Step 1

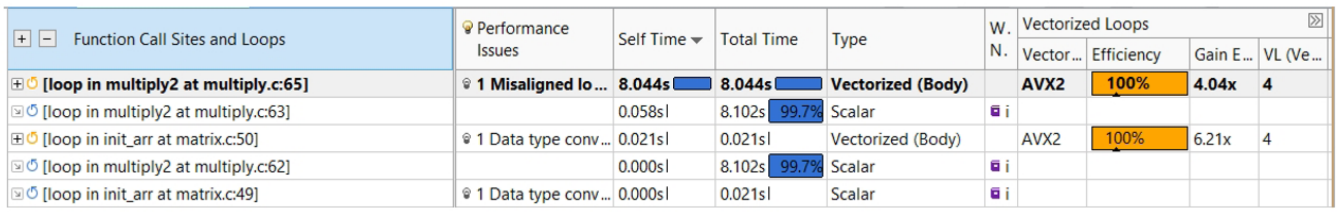
Do a loop interchange so that we don't need a constant stride and also don't need to access memory over such a wide range. We can also vectorize the loop by including a pragma `ivdep` that informs the compiler that we don't have a dependency that prevents vectorization.

```
for(i=tidx; i<msize; i=i+numt) {
    for(k=0; k<msize; k++) {
#pragma ivdep
        for(j=0; j<msize; j++) {
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
        }
    }
}
```

The elapsed time for our new run is 4.12 seconds, an improvement of more than12x. Why is our new performance so much better? First, let's take a look at our Integrated Roofline chart (**Figure 23**). Each of the red circles represents the bandwidth of the corresponding memory hierarchy: L1, L2, L3, and DRAM. We can see that our computational loop's L1 memory bandwidth, represented by the leftmost red circle, is now 95 GB/second. We can also use the Survey view (**Figure 24**) to see that we're also now vectorized at 100% efficiency using AVX2 instructions.



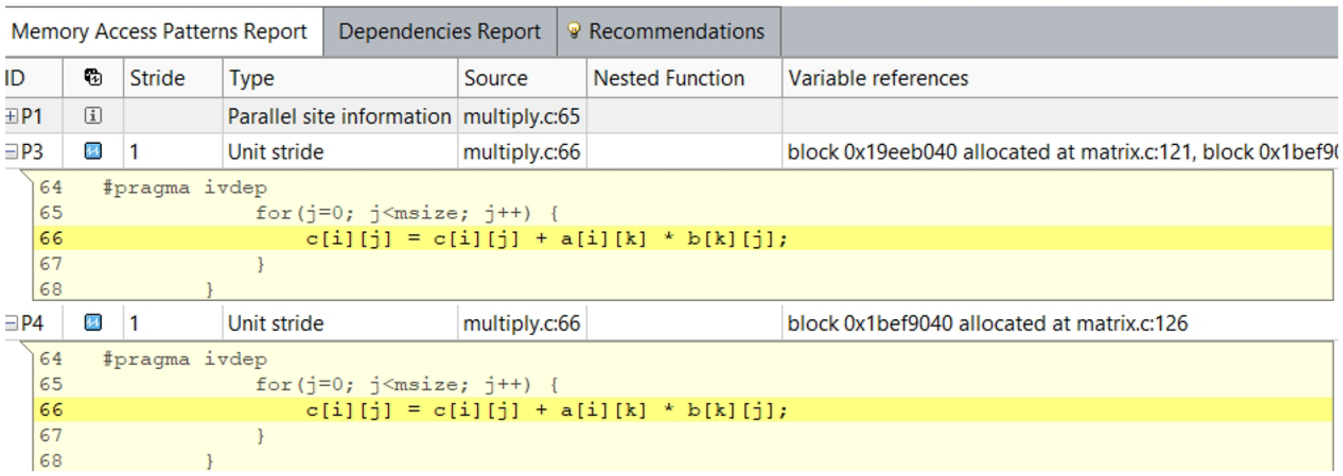
**23** **Integrated Roofline chart**



| Function Call Sites and Loops | Performance Issues | Self Time | Total Time | Type | W. N. | Vectorized Loops | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | Vector... | Efficiency | Gain E... | VL (Ve... |
| **[loop in multiply2 at multiply.c:65]** | **1 Misaligned lo...** | **8.044s** | **8.044s** | **Vectorized (Body)** | | **AVX2** | **100%** | **4.04x** | **4** |
| [loop in multiply2 at multiply.c:63] | | 0.058s | 8.102s 99.7% | Scalar | i | | | | |
| [loop in init_arr at matrix.c:50] | 1 Data type conv... | 0.021s | 0.021s | Vectorized (Body) | | AVX2 | 100% | 6.21x | 4 |
| [loop in multiply2 at multiply.c:62] | | 0.000s | 8.102s 99.7% | Scalar | i | | | | |
| [loop in init_arr at matrix.c:49] | 1 Data type conv... | 0.000s | 0.021s | Scalar | i | | | | |

**24** **Survey view**

Our MAP report (**Figure 25**) informs us that all our accesses are now unit stride and the maximum address range is 16KB, well within the range of our cache size. Our cache is also performing much better (**Figure 26**). We've dropped to 512 cache misses, down from 2,302. So we're getting better performance, but we're still not near the peak.

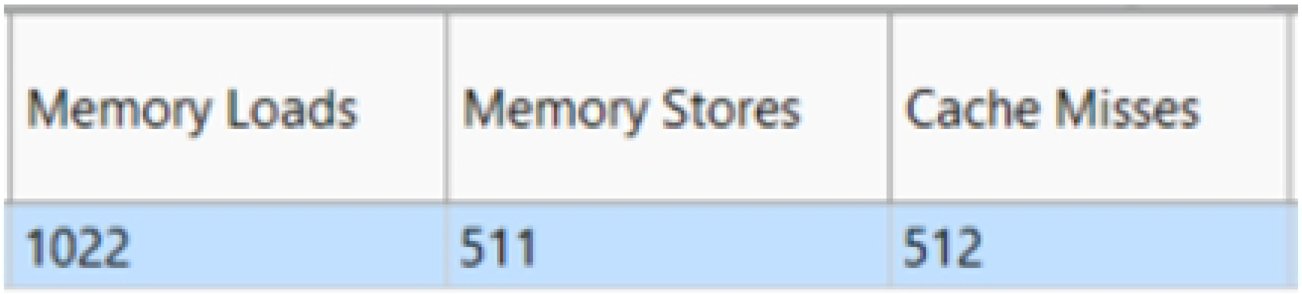


| ID | | Stride | Type | Source | Nested Function | Variable references |
|---|---|---|---|---|---|---|
| P1 | | | Parallel site information | multiply.c:65 | | |
| P3 | | 1 | Unit stride | multiply.c:66 | | block 0x19eeb040 allocated at matrix.c:121, block 0x1bef9 |
| P4 | | 1 | Unit stride | multiply.c:66 | | block 0x1bef9040 allocated at matrix.c:126 |

```
#pragma ivdep
            for(j=0; j<msize; j++) {
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
            }
        }
```

```
#pragma ivdep
            for(j=0; j<msize; j++) {
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
            }
        }
```

**25** **Map report**



| Memory Loads | Memory Stores | Cache Misses |
|---|---|---|
| 1022 | 511 | 512 |

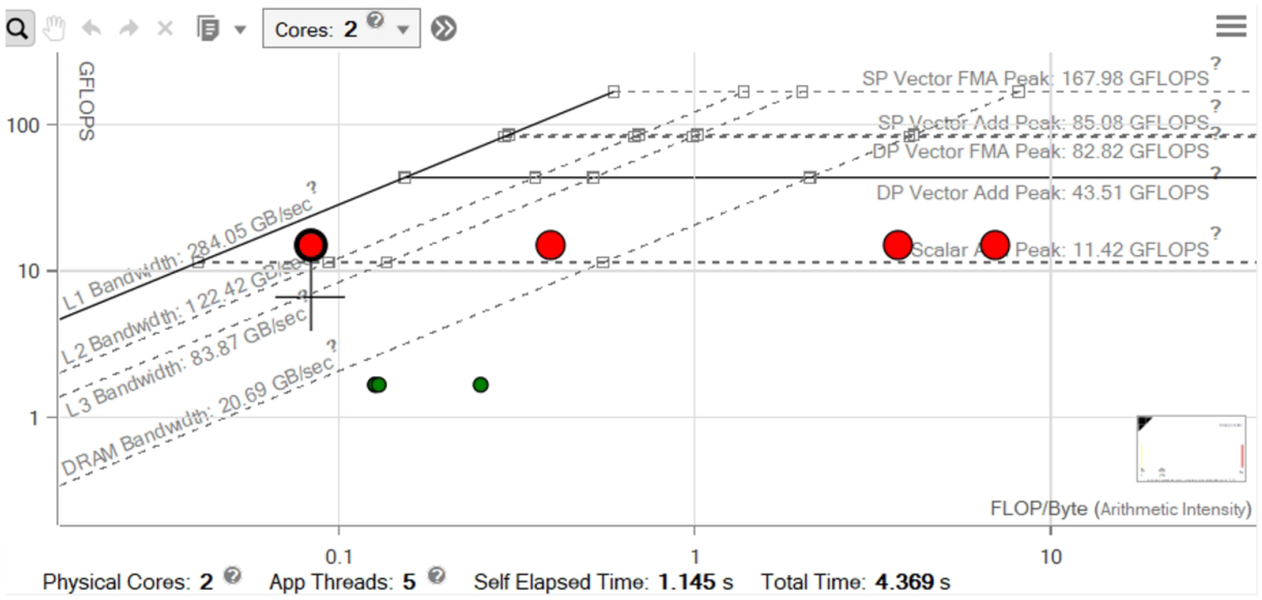
**26** **Cache performance**

## Step 2

Implement cache-blocking so that our computations are over a smaller range of memory:

```
for (i0 = ibeg; i0 < ibound; i0 +=mblock) {
    for (k0 = 0; k0 < msize; k0 += mblock) {
        for (j0 =0; j0 < msize; j0 += mblock) {
            for (i = i0; i < i0 + mblock; i++) {
                for (k = k0; k < k0 + mblock; k++) {
#pragma ivdep
#ifdef ALIGNED
     #pragma vector aligned
#endif //ALIGNED
               #pragma nounroll
                    for (j = j0; j < j0 + mblock; j++) {
                        c[i][j]  = c[i][j] + a[i][k] * b[k][j];
                    }
                }
            }
        }
    }
}
```

In the above code, we're adding three additional nested loops so that we do our computations in sections (or blocks). After we're done with one block, we move to the next. The elapsed time of our cache-blocked case is 2.60 seconds, a 1.58x improvement from the previous run (**Figure 27**). Our loop's L1 memory bandwidth is now 182 GB/second, much closer to the L1 roof. Our vectorization and striding have not changed, but we now have only 15 cache misses for our inner loop, and our address range has been reduced to 480 bytes (**Table 3**).



Physical Cores: 2 App Threads: 5 Self Elapsed Time: 1.145 s Total Time: 4.369 s

**27** **Performance improvements**

**Table 3. Summary of results**

| Run | Time Elapsed, Seconds | Total GFlops | Memory Address Range | Cache Misses | Improvement (Time) |
|---|---|---|---|---|---|
| Baseline | 53.94 | 0.32 | 32 MB | 2,302 | N/A (baseline) |
| Loop Interchange | 4.19 | 4.17 | 16 KB | 511 | 12.87x |
| Blocked | 2.6 | 6.61 | 480 B | 15 | 20.74x |

## Optimizing Memory Accesses

It's crucial to optimize the memory accesses of your program. Understanding how your program is accessing memory, using a tool like Intel Advisor, can help you get the most out of your hardware. By using the Roofline and new Integrated Roofline features of Intel Advisor, you can visualize your memory bottlenecks. You can get even greater insight when you combine Roofline features with Memory Access Pattern analysis.

## Related Resources

- **Intel Advisor Roofline**
- **Intel Advisor Integer Roofline**
- **Intel Advisor Integrated Roofline**

HOW'D THEY DO THAT?
Developers worldwide have upped the ante for application performance, scalability, and portability with Intel® Software Development Tools. And they're sharing their stories to help you do the same.
EXPLORE >
intel
Software

# MPI-3* NON-BLOCKING I/O COLLECTIVES IN INTEL® MPI LIBRARY

## Speeding Up I/O for HPC Applications

***Nitya Hariharan, Amarpal Singh Kapoor, and Rama Kishan Malladi, Technical Marketing Engineers, Core and Visual Computing Group, Intel Corporation; Md Vasimuddin, Research Scientist, Parallel Computing Lab, Intel Labs***

For an application to be truly scalable, every section must scale linearly—or at least tend toward linear scalability. Amdahl's Law tells us that a small serial fraction, or a poorly scaling parallel code section, can have a significant impact on the overall scalability of the application. This article focuses on one such section of HPC applications that tends to be serial: file I/O. We demonstrate the use and performance benefits of non-blocking MPI* I/O calls in the context of some real-world HPC applications.

MPI provides non-blocking calls to allow developers to benefit from overlapping communication and computation, communication with other communication, and to aid I/O-intensive codes. [In **Issue 33 of *The Parallel Universe***[1], the article "Hiding Communication Latency Using MPI-3 Non-Blocking Collectives" demonstrates the use and benefits of non-blocking collective (NBC) communication. This article focuses on non-blocking I/O (e.g., `MPI_File_iwrite_at`).]

## NBC I/O

NBC I/O operations are attractive because they can take advantage of both non-blocking and collective operations[2]. Non-blocking calls have the potential of hiding I/O cost by allowing execution of other independent computations in parallel. Apart from maximizing the hardware utilization, the non-blocking nature of these I/O calls also helps hide the synchronization costs of delayed processes. Another motivation to consider optimizations of this nature is the lower I/O performance relative to computation.

We'll look at two application codes, LAMMPS* and BWA-MEM*, to study the performance gains from using non-blocking I/O calls. LAMMPS is a molecular dynamics application with a focus on material modeling[3]. BWA-MEM* is one of the most popular tools for mapping short DNA fragments, also called reads, to reference sequences such as the human genome[4]. Both use MPI I/O, making them good candidates to test NBC I/O.

## Test Case 1: LAMMPS

LAMMPS executes using an input file with a command in each line. Each command causes LAMMPS to take an action (e.g., set an internal variable or run a simulation for a given number of time steps). The Dump command looks like this:

```
dump ID group-ID style N file args
```

For our I/O testing, we used the 3D Lennard-Jones (LJ) workload and added a Dump command to the input file. The corresponding inputs provided for the test are:

```
dump myDump all atom/mpiio 20 dump.atom.mpiio
```

(Details of each input to the Dump command are given in reference 4 at the end of this article.)

The code maintains a list of dumps that need to be done and checks the list to determine the dumps that need to be done at every N time step. The atom data to be written is copied out to a string buffer. And the offset into the output file is calculated by each MPI rank. Then, the collective, blocking MPI I/O call `MPI_File_write_at_all` is used to write the data to the file.

Here's the baseline code using a blocking `MPI_File_write_at_all`:

```
//Iterate over n time steps, check for dump at N time steps
//Copy atom data to string and calculate offset into file
call MPI_File_write_at_all(mpifh, offset, sbuf, nsme, MPI_CHAR, status);
```

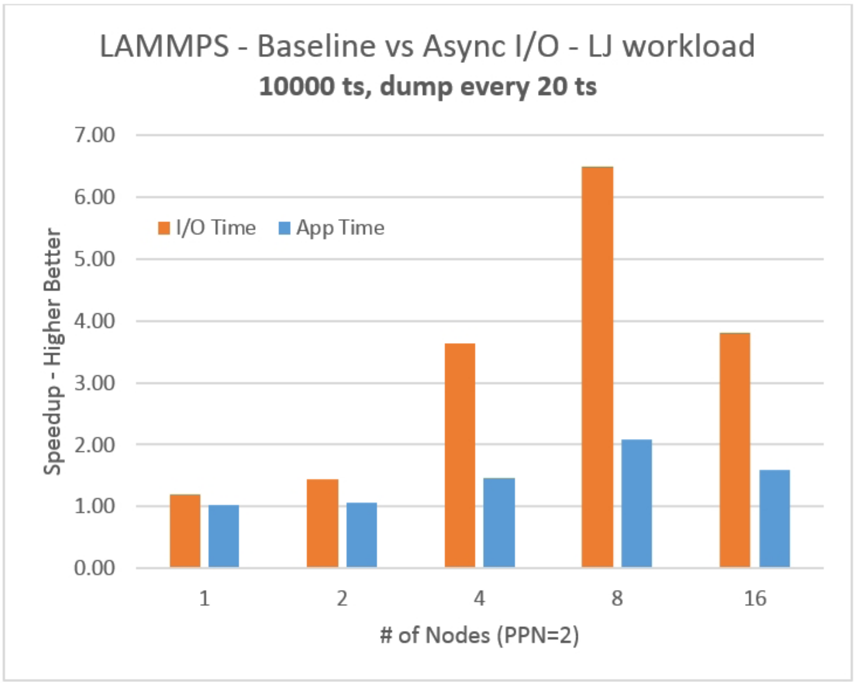
Here's the optimized code using a non-blocking `MPI_File_iwrite_at`:

```
       //Iterate over n time steps, check for dump at N time steps
if(countWrites%2==0){
       //Copy atom data to string buffer sbuf, and calculate offset into file
       MPI_File_iwrite_at(mpifh, offset, sbuf, count, MPI_CHAR, request); }
else {
       //Copy atom data to string buffer sbuf1, and calculate offset into file
       MPI_File_iwrite_at(mpifh, offset, sbuf1, count, MPI_CHAR, request1);}
//Compute for next iteration
if(countWrites%2==0) MPI_Wait(&request1, &status1);
else MPI_Wait(&request, &status);
```

Given that the data being written in each iteration is independent, we can make use of an NBC I/O call here to overlap computation and file I/O. Since the offset into the file is already being calculated for each rank to prepare for MPI I/O, we only need to change the code to use the NBC I/O call and add the corresponding wait.

Also, each MPI rank writes independently to the file, so we use the non-blocking `MPI_File_iwrite_at` call instead of the collective, non-blocking `MPI_File_iwrite_at_all` call. To ensure good overlap of computation and I/O, we use a "prologue" and "epilogue" phase (i.e., a double-buffering algorithm), which uses different buffers to issue the asynchronous write for different iterations. We keep track of the number of writes being issued to decide which buffer to use and issue the `MPI_Wait` call for the previous iteration in the current iteration. This ensures correct buffer reuse.

**Figure 1** shows the speedup for MPI I/O and the total runtime for the LJ workload (10,000 time steps with a data dump every 20 time steps) at increasing node counts (two MPI processes per node and 20 OpenMP threads per MPI rank). The runs were done on an **Intel® Xeon® Gold 6148 processor**-based cluster with two sockets per node connected by **Intel® Omni-Path Architecture**. The I/O and total runtime were measured using the default timers in LAMMPS. Note that the speedup in I/O time doesn't translate into a corresponding speedup in the total runtime. However, the parallel, non-blocking I/O does improve overall performance, even doubling the application performance at eight nodes.



**1** **Speedup from MPI I/O time and total run time for LAMMPS**

## Test Case 2: BWA-MEM

In next-generation sequencing (NGS) applications, sequence mapping is compute-intensive. As a primary step in the GATK (Genome Analysis ToolKit) workflow[5]—a popular workflow for genome assembly and finding genetic variants—sequence mapping accounts for 30% of the overall runtime. BWA-MEM[6] is one of the most popular tools for mapping short DNA reads to reference genome sequences. Given a set of input sequences, BWA-MEM tries to find their most probable positions in the reference sequence.

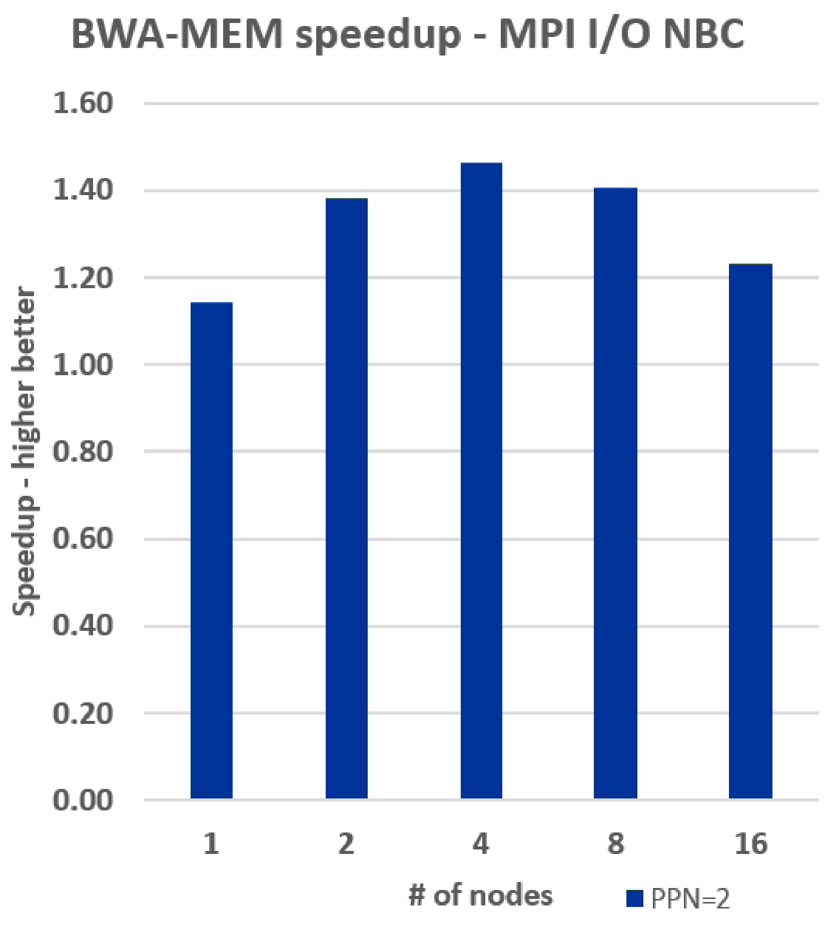
BWA-MEM supports and scales well on a distributed-memory system and makes use of MPI I/O for writing its output. The code samples below show implementations using an MPI I/O blocking call and an optimization using non-blocking MPI I/O. Here's the baseline code using a blocking `MPI_File_write_at`:

```
    MPI_Status status;
  for (int l=0; l<index; l++) {
          std::pair<int64_t, ktp_data_t*> pr = samCache[l];
          assert(pr.first < aux->ntasks);
          int64_t offset = totSamSize[pr.first];
          ret = pr.second;
          std::string str;

          for (i = 0; i < ret->n_seqs; ++i) {
                  if (ret->seqs[i].sam)
                          str += ret->seqs[i].sam;
                  free(ret->seqs[i].sam);
           }
           free(ret->seqs);


           /* Write to output SAM file */
           MPI_File_write_at(aux->mfp, offset, str.c_str(), \
                           samSize[pr.first], MPI_INT8_T, &status);
```

Here's the optimized code using a non-blocking `MPI_File_iwrite_at`:

```
    MPI_Request request[index];
            int64_t offset[index];
            int64_t first[index];
            int test_flag;
            std::vector<std::string> str(index);
            for (int l=0; l<index; l++)
            {
                    std::pair<int64_t, ktp_data_t*> pr = samCache[l];
                    assert(pr.first < aux->ntasks);
                    first[l] = pr.first;
                    offset[l] = totSamSize[first[l]];
                    ret = pr.second;

                    for (i = 0; i < ret->n_seqs; ++i) {
                            if (ret->seqs[i].sam)
                                    str[l] += ret->seqs[i].sam;
                            free(ret->seqs[i].sam);
                    }
                    free(ret->seqs);

                    /* Write to output SAM file */
                    MPI_File_iwrite_at(aux->mfp, offset[l], str[l].c_str(), \
                                       samSize[first[l]], MPI_INT8_T, &request[l]);
            }
            MPI_Waitall(index,request,MPI_STATUSES_IGNORE);
```

**Figure 2** shows the BWA-MEM performance improvement using NBC MPI I/O (two MPI processes per node and 20 threads per MPI rank). The runs were done on an Intel Xeon Gold 6148 processor-based cluster with two sockets per node connected by Intel Omni-Path Architecture. Parallel, non-blocking I/O improves the performance of each test.



**2** **BWA-MEM performance improvement using NBC MPI I/O**

## Speeding Up I/O for HPC Applications

Modern HPC applications are quite complex and often deal with iterative solution procedures. As a consequence, several solution steps need to be written to a file in addition to safe-keeping mechanisms like writing additional restart/temporary files, should the current run fail. Moving to MPI I/O is the first step for efficiently handling I/O in an HPC environment. This article demonstrated speedups with non-blocking MPI I/O calls for two real applications: LAMMPS and BWA-MEM. The code changes also highlight the ease of using NBC MPI I/O in these applications.

## References

[1] ***The Parallel Universe*, Issue 33**

[2] Seo S., Latham R., Zhang J., and Balaji P. "Implementation and Evaluation of MPI Nonblocking Collective I/O."

[3] S. Plimpton, "Fast Parallel Algorithms for Short-Range Molecular Dynamics," J Comp Phys, 117, 1-19 (1995).

[4] LAMMPS dump command: **https://lammps.sandia.gov/doc/dump.html**

[5] M. Vasimuddin, S. Misra, and S. Aluru, "Identification of Significant Computational Building Blocks through Comprehensive Investigation of NGS Secondary Analysis Methods," [Preprint] bioRXiv, April 2018

[6] Li H. (2013) Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. arXiv:1303.3997v1 [q-bio.GN].

YOUR PYTHON* SHOULDN'T BITE.

Supercharge your applications with Intel® Distribution for Python.*

FREE DOWNLOAD >

intel®
Software

Software

# THE PARALLEL UNIVERSE