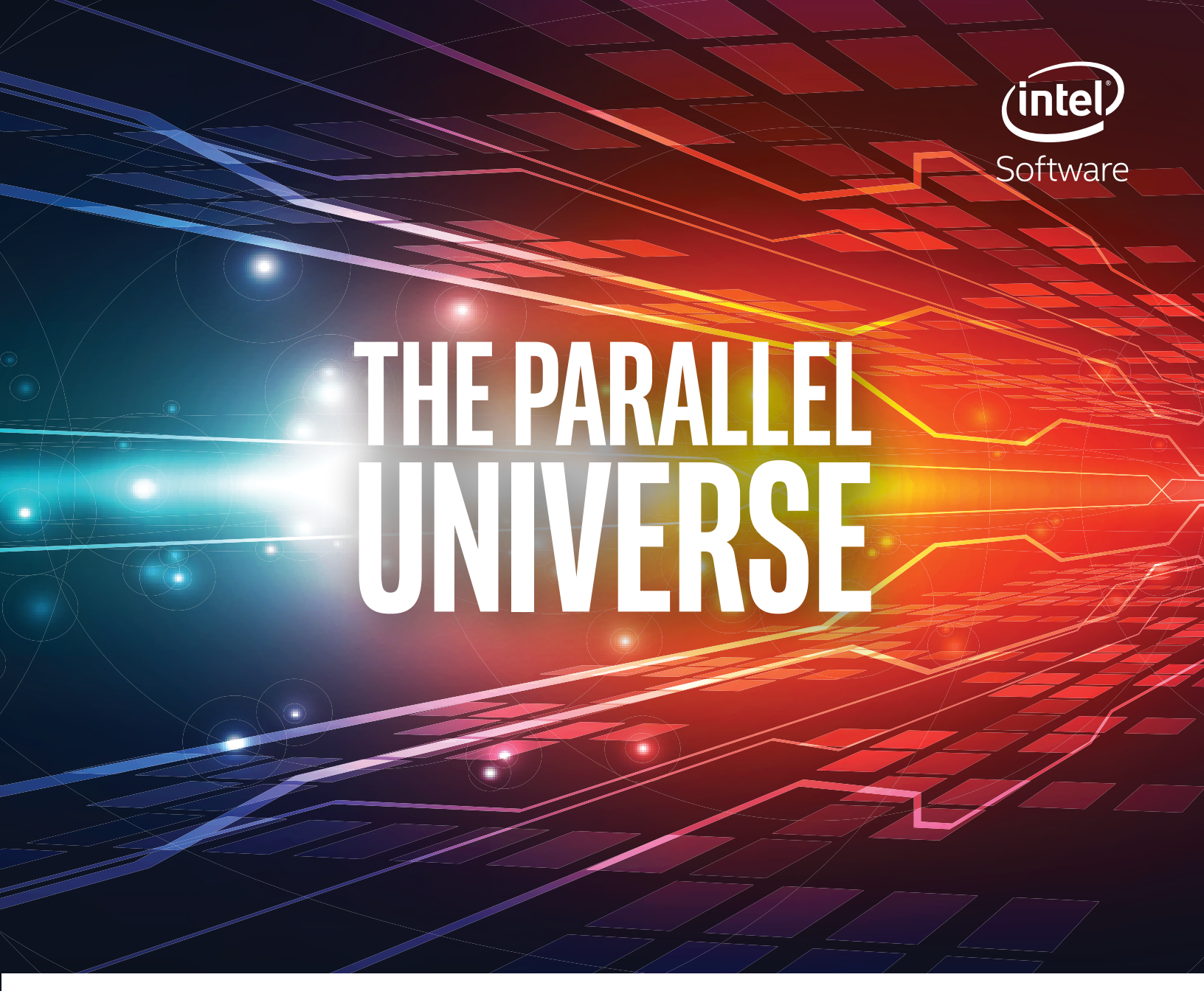


Leadership Performance with 2nd-Generation Intel® Xeon® Scalable Processors

Using the Latest Performance Analysis Tools to Prepare for Intel® Optane™ DC Persistent Memory

Measuring the Impact of NUMA Migrations on Performance

Issue
37
2019

CONTENTS

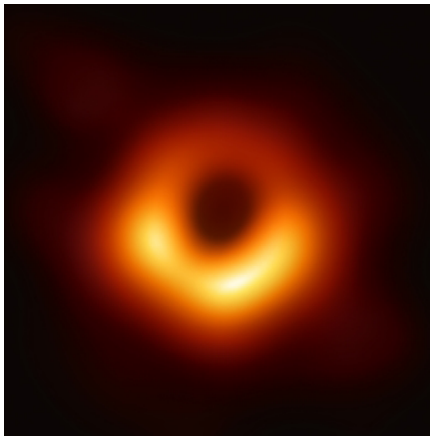
Sign up for future issues

LETTER FROM THE EDITOR

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

Black Holes and High-Performance Computing

It probably seems like a long time ago, but it's just been three months since the Event Horizon Telescope published its black hole image.



Credit: Event Horizon Telescope Collaboration

This was obviously an amazing scientific feat. But a single image doesn't convey the vast amount of expertise, data, and computation that went into its creation. The Event Horizon General Relativistic Magnetohydrodynamic Code Comparison Project provides details about some of the codes involved, including ECHO*. Advancing the Performance of Astrophysics Simulations with ECHO-3DHPC* (published last year in issue 34 of *The Parallel Universe*) describes the optimization of this code by researchers from the Leibniz Supercomputing Centre in collaboration with Intel.

Our feature article in this issue, **Leadership Performance with 2nd-Generation Intel® Xeon® Scalable Processors**, describes the newest addition to the Intel Xeon processor family. This new processor includes Intel® Deep Learning Boost, support for Intel® Optane™ DC persistent memory, and up to 56 cores and 12 DDR4 memory channels per socket. After reading this article, you'll know why this new processor is setting new performance records. **Using the Latest Performance Analysis Tools to Prepare for Intel® Optane™ DC Persistent Memory** shows you how to determine if your application will benefit from this new memory technology, and how to analyze applications that use this technology.

Non-uniform memory access (NUMA) architectures have been around for a long time. Most of us know that threads should stay close to their data for faster memory access, but few of us pay attention to where our threads are actually running or whether the operating system is moving our threads around. **Measuring the Impact of NUMA Migrations on Performance** helps you to understand how your threads are behaving on NUMA systems.

Our series on optimizing and parallelizing Python* codes continues in this issue. **Parallelism in Python*: Directing Vectorization with NumExpr*** shows how simple code modifications can drastically improve the performance of complex mathematical expressions.

Turbo-Charged Open Shading Language* on Intel® Xeon® Processors with Intel® Advanced Vector Extensions 512 describes Intel's efforts to vectorize the Oscar*-winning Open Shader Language*, the *de facto* open source standard for digital content creation that has over 100 movie credits.

Finally, we close this issue with two guest editorials: one from Mike Croucher from Numerical Algorithms Group and another from James Reinders, our editor emeritus. Mike describes **The Performance Optimisation and Productivity (POP) Project** that the European Union funds to improve software performance. In **Seven Ways HPC Software Developers Can Benefit from Intel® Software Investments**, James describes how you can maximize performance while minimizing effort by taking advantage of work that Intel has already done. These editorials show that sometimes the path to performance is just a matter of knowing what's available.

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

Henry A. Gabb
July 2019

LEADERSHIP PERFORMANCE WITH 2ND-GENERATION INTEL® XEON® SCALABLE PROCESSORS

New Features and Tools to Maximize Your HPC, AI, and Analytics Applications

Amarpal S. Kapoor, Technical Consulting Engineer; Rama Kishan V. Malladi, Performance Modeling Engineer; and Avinash Karani and Nitya Hariharan, Application Engineers; Intel Corporation

April 2019 saw the launch of the 2nd-generation **Intel® Xeon® Scalable processor** (formerly codenamed Cascade Lake), a server-class processor. This new processor family has already set 95 performance world records, earning performance leadership[1]. New features include Intel® Deep Learning Boost (Intel® DL Boost) for AI deep learning inference acceleration and support for **Intel® Optane™ DC** (data center) persistent memory. These processors will continue to deliver leadership performance with up to 56 cores per CPU socket and 12 DDR4 memory channels per socket—making them ideal for a wide variety of HPC, AI, and analytics applications with high-density infrastructures.

The Intel Xeon Scalable processor is designed to address a range of compute needs and demands, including more than 50 workload-optimized solutions and a variety of custom processors. The 8200 series offers up to 28 cores (56 threads), while the 9200 series has up to 56 cores (112 threads). Each processor core has a 1MB dedicated L2 cache and a non-inclusive shared L3 cache of up to 38.5 MB. On each socket, there are up to three Intel® Ultra Path Interconnect (Intel® UPI) links operating at 10.4 GT/s for cross-die (multi-socket) communication. The processor memory interface now supports up to six channels (on the 8200 series) and 12 channels (on the 9200 series) of DDR4 memory, operating at 2,933 MT/s. Also, the processor supports up to 4.5 TB of memory per socket using the Intel Optane DC persistent memory modules. To help improve the performance of DL applications, Intel Xeon Scalable processors have 512-bit VNNI (vector neural network instructions), which help in processing up to 16 DP/32 SP/128 INT8 MAC (multiply accumulate) instructions per cycle per core. To address some side-channel security issues, Intel Xeon Scalable processors implement hardware mitigations, which have smaller overhead compared to software-based methods[2]. These processor features apply in multiple computational domains, some of which we'll discuss below.

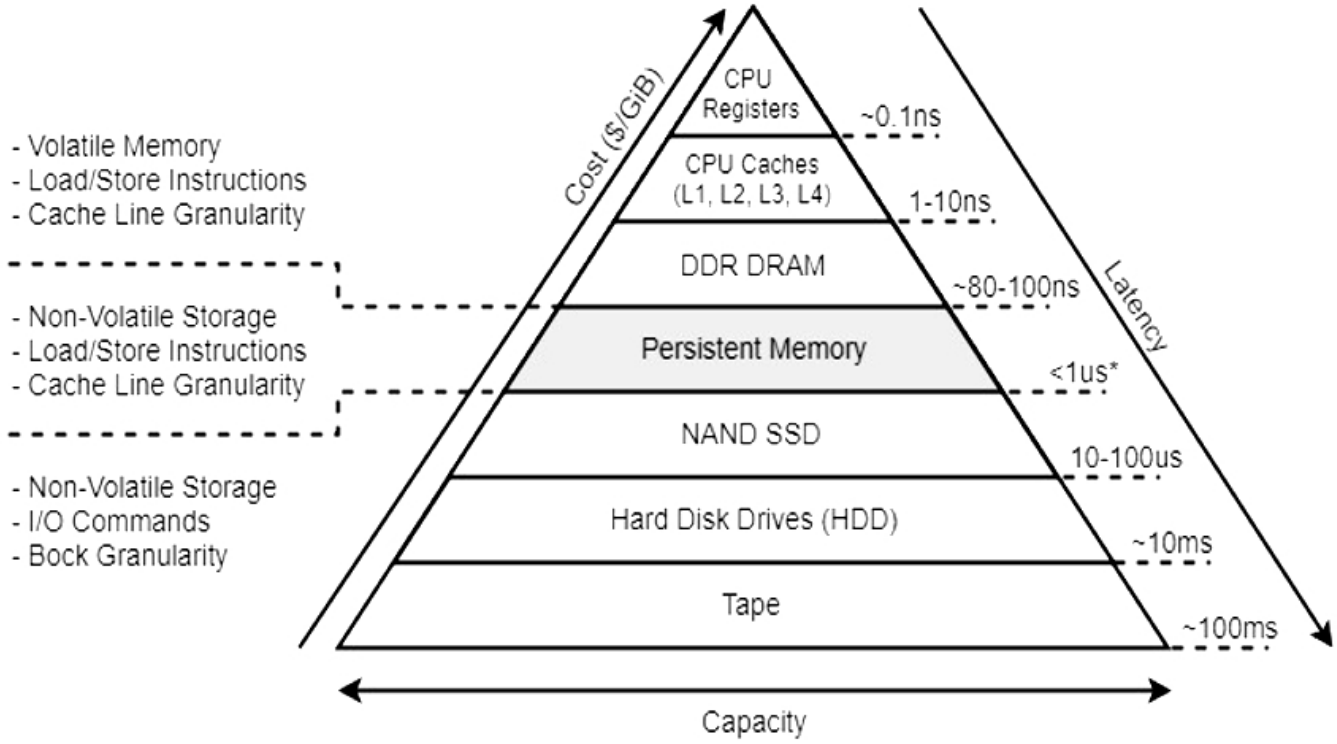
We'll also discuss working with Intel Optane DC persistent memory, Intel® AVX-512 Vector Neural Network Instructions (VNNI) for faster DL inference, and relative performance gains achieved in HPC applications on the Intel Xeon Scalable processor.

[Editor's note: We discuss ways to determine how applications can best utilize this new memory in **Using the Latest Performance Analysis Tools to Prepare for Intel® Optane™ DC Persistent Memory** in this issue.]

Intel® Optane™ DC Persistent Memory

Intel Optane DC persistent memory is a new type of non-volatile, high-capacity memory with near-DRAM latency, offering affordable, high-capacity data persistence. **Figure 1** shows latency estimates for different classes of memory and storage devices. Note the new tier that Intel Optane DC persistent memory creates between SSD and conventional DRAM.

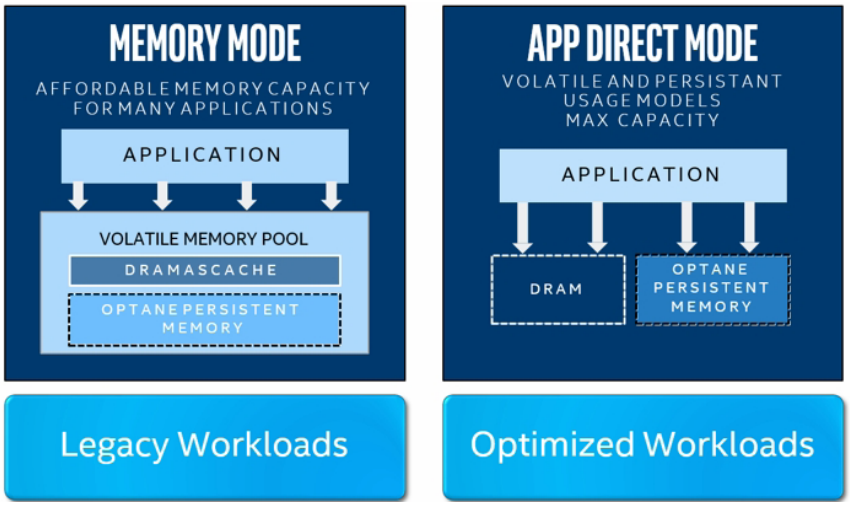
Intel Optane DC persistent memory is available in the same form factor as DRAM and is both physically and electrically compatible with DDR4 interfaces and slots. Intel Xeon Scalable processor-based machines must be populated with a combination of DRAM and Intel Optane DC persistent memory. (It's not possible to just have Intel Optane DC persistent memory on an Intel Xeon Scalable processor-based machine, since DRAM is necessary to serve system activities.)



1 **Latency estimates for different storage and memory devices**

Intel Optane DC persistent memory can be used in two different modes (**Figure 2**):

- Memory mode
- App Direct mode



2 **Modes of operation for Intel® Optane™ DC persistent memory**

Memory Mode

This is the simplest mode for using Intel Optane DC persistent memory, since existing applications can benefit without any source changes. In this mode, a new pool of volatile memory becomes visible to the operating system and user applications. The DRAM acts as a cache for hot (frequently accessed) data, while Intel Optane DC persistent memory provides a large volatile memory capacity. Memory management is handled by the Intel Xeon Scalable processor memory controller. When data is requested from memory, the memory controller first checks the DRAM cache. If data is found, the response latency is identical to DRAM latency. If data isn't found in the DRAM cache, it's read from Intel Optane DC persistent memory, which has higher latency. Memory controller prediction mechanisms aid in delivering better cache hit rates by fetching the required data in advance. However, workloads with random access patterns over a wide address range may not benefit from the prediction mechanisms and would experience slightly higher latencies compared to DRAM latency[3].

App Direct Mode

For data to persist in memory, Intel Optane DC persistent memory should be configured for use in App Direct mode. In this mode, the operating system and user applications become aware of both DRAM and Intel Optane DC persistent memory as discrete memory pools. The programmer can allocate objects in either of the memory pools. Data that needs to be fetched with the least latency must be allocated in DRAM (this data will be inherently volatile). Large data, which might not fit in DRAM, or data that needs to be persistent, must be allocated in Intel Optane DC persistent memory. These new memory allocation possibilities are the reason behind the need for source code changes in App Direct mode. Interestingly, in App Direct mode, it's also possible to use Intel Optane DC persistent memory as a faster storage alternative to conventional HDD/NVMe storage devices.

Configuration

Switching between Memory and App Direct modes requires changes in BIOS settings. `ipmctl` is an open source utility available for configuring and managing Optane persistent memory modules (PMM)[4]. Here are some useful management commands:

```
$ ipmctl show -topology
$ ipmctl show -memoryresources
```

Provisioning PMMs is a two-step process. First, you specify a goal and store it on the PMMs. Then the BIOS reads it at the next reboot.

Memory mode
```
$ ipmctl create -goal MemoryMode=100
```

App Direct mode
```
$ ipmctl create -goal PersistentMemoryType=AppDirect
```

Mixed mode
```
$ ipmctl create -goal MemoryMode=60
```

In the Mixed mode, a specified percentage of Intel Optane DC persistent memory can be used in memory mode and the remaining memory can be used in App Direct mode. The command above will assign 60% of available persistent memory in memory mode and 40% in App Direct mode. (For details, see references 4 and 5.)

Persistent Memory Development Kit (PMDK)

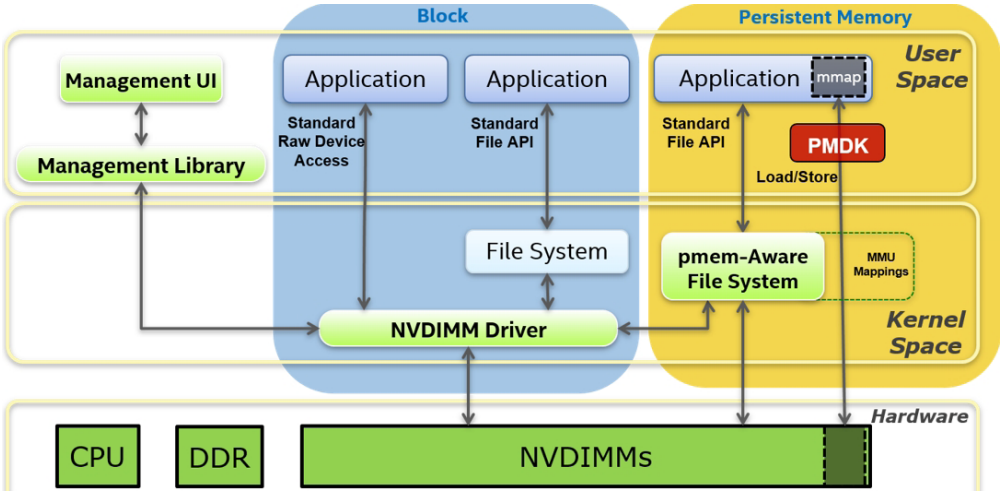
Applications can access persistent, memory-resident data structures in place, as they do with traditional memory, eliminating the need to page blocks of data back and forth between memory and storage. Getting this low-latency direct access requires a new software architecture that allows applications to access ranges of persistent memory[6]. The Storage Network Industry Association (SNIA) Programming Model comes to our rescue here, as shown in **Figure 3**.

PMDK is a collection of libraries and tools that system administrators and application developers can use to simplify managing and accessing persistent memory devices. These libraries let applications access persistent memory as memory-mapped files. **Figure 3** shows the SNIA model, which describes how applications can access persistent memory devices using traditional POSIX standard APIs such as `read`, `write`, `pread`, and `pwrite`, or load/store operations such as `memcpy` when the data is memory-mapped to the application. The Persistent Memory area represents the fastest possible access because the application I/O bypasses existing filesystem page caches and goes directly to or from the persistent memory media[6].

PMDK contains the following libraries and utilities to address common programming requirements with persistent memory systems:

PMDK libraries
`Libpmem`
`Libpmemobj`
`Libpmemblk`
`Libpmemlog`
`Libvmem`
`Libvmmalloc`
`Libpmempool`
`Librmem`
`Libvmemcache`

PMDK libraries
`pmempool`
`pmemcheck`



3 **SNIA Programming Model[6]**

PMDK Example

This section demonstrates the use of PMDK through the `libpmemobj` library, which provides a transactional object store, providing memory allocation, transactions, and general facilities for persistent memory programming. This example demonstrates two applications:

- **`writer.c`**, which writes a string to persistent memory
- **`reader.c`**, which reads that string from persistent memory

Code snippets with comments are shown in **Table 1**. The complete source for this example is available in reference 7.

Table 1. Working with strings in persistent memory

writer.c	reader.c

`int main(int argc, char *argv[1]`
`{`
`PMEMobjpoo] *pop = pmemobj_create(argv[1],`
`LAYOUT_NAME,`
`PMEMOB]_MIN_POOL, 0666),`
`if (pop == NULL) {`
`Perror("pmemobj_create");`
`return 1;`
`}`
`...`
`...`
`pmemobj_close(pop);`
`return 0;`
`}`
	`int main(int argc, char *argv[1]`
	`{`
	`PMEMobjpoo] *pop = pmemobj_open(argv[1],`
	`LAYOUT_NAME,`
	`if (pop == NULL)`
	`{`
	`perror("pmemobj_open");`
	`return 1;`
	`}`
	`...`
	`...`
	`pmemobj_close(pop);`
	`return 0;`
	`}`

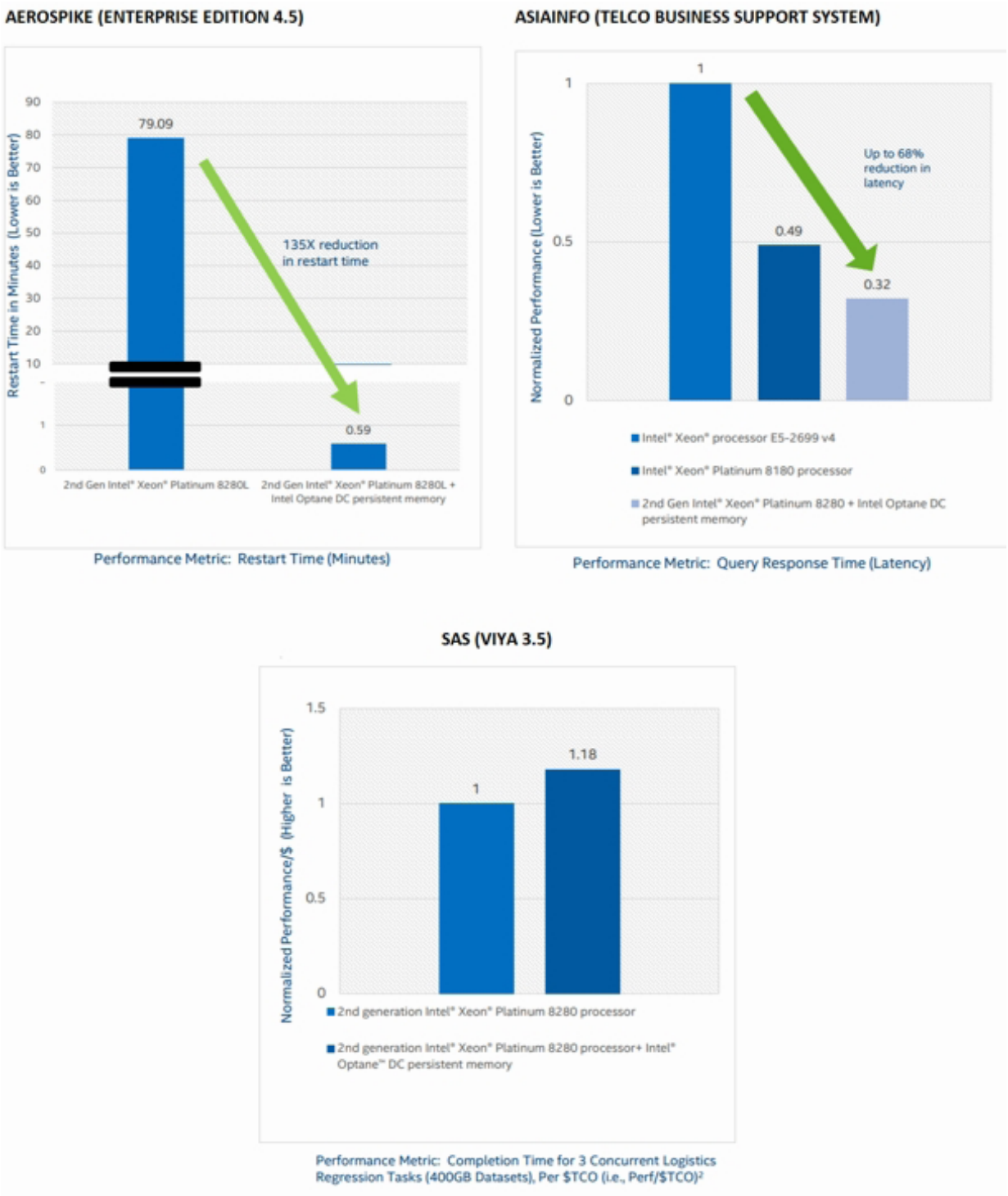
The `pmemobj_create` API function takes the usual parameters you would expect for a function creating a file plus a layout, which is a string of your choosing that identifies the pool.	In the reader, instead of creating a new pool, we open the pool we have created in the writer code using the same layout.
`PMEMoid root = pmemobj_root(pop, sizeof (struct my_root));` `struct my)root *rootp = pmemobj_ direct(root);`	`PMEMoid root = pmeobj_root(pop, sizeof (structr my_root));` `struct my_root *rootp = pmeobj_direct(root);`
It is required to keep a known location for the application in the memory pool, called the root object. It is the anchor to which all the memory structures can be attached. In the above code, we are creating a root object using `pmemobj_root` in the `pop` memory pool. We are also translating the `root` object to a usable, direct pointer using `pmemobj_direct`.	Since we already created the root object in the pool, **`pmeobj_root`** returns the root object without initializing it with zeros. It will contain whatever string the writer was tasked with storing.
`char buf[MAX_BUT_LEN];` `scanf("X9s", buf):`	`if (rootp->len == strlen(rootp->buf))` `printrf("%s\n", rootp->buf);`
A maximum of 9 bytes are then read to the temporary buffer.	The above section reads the string from persistent memory.

`root->len = strlen(bvuf);`
`pmemmobj_persistr(pop, &rootrs->len, sizeof`
`(rootrp->len));`
`pmemobvj_memcpy_persistr(pop, rootp->buf,`
`my_buf, rootp->len);`	

In the above section, we force any changes in the range (&rootp->len &rootp->len+sizeof(rootp->len))to be stored durably in persistent memory using pmeobj_persist and we copy the string from local buffer to persistent memory using pmeobj_ memcopy_persist.	

Performance Gains from Intel Optane DC Persistent Memory

This section presents the performance gains achieved in enterprise applications like Aerospike*, Asiainfo's benchmark, and SAS VIYA* using Intel Optane DC persistent memory (**Figure 4**). Aerospike is a NoSQL* key-value database application which saw a 135x reduction in restart times with the use of Intel Optane

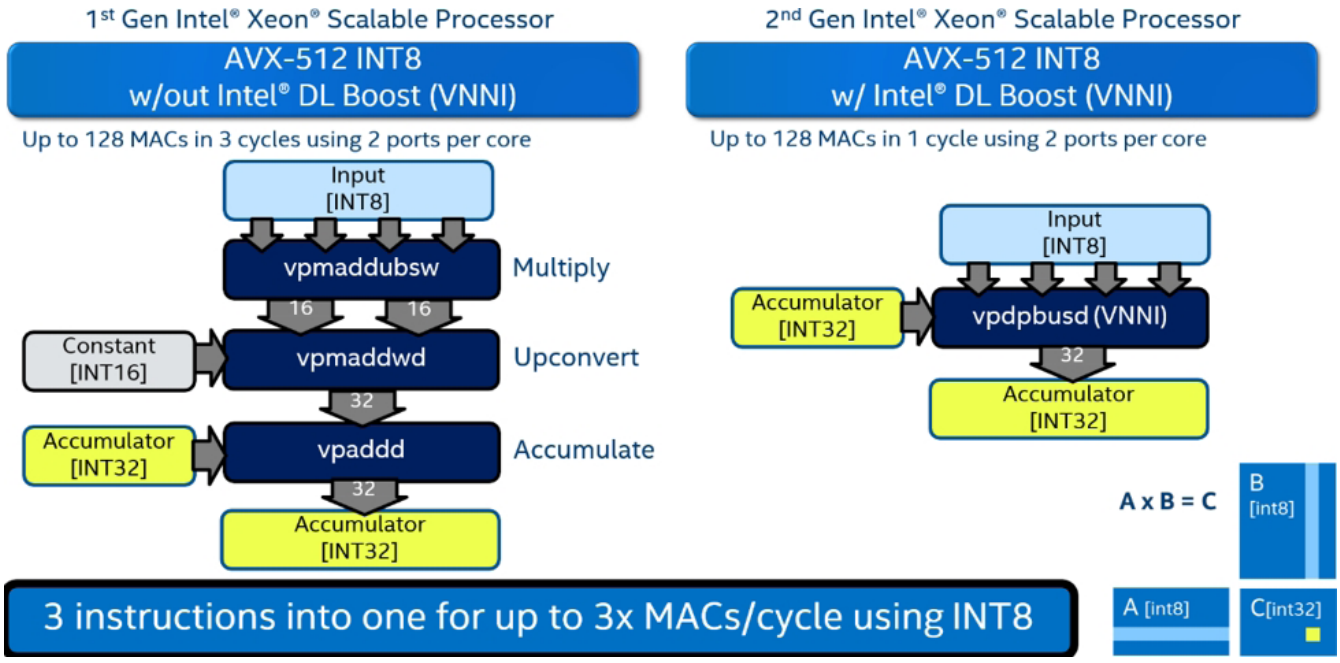


4 **Better performance in enterprise applications using Intel Optane DC persistent memory**

DC persistent memory in App Direct mode. This helps Aerospike restart in seconds instead of hours, allowing for more frequent software and security updates, while significantly reducing disruption. One of Asiainfo's benchmarks saw a 68% reduction in latency due to the combined effects of Intel Xeon Scalable processors and Intel Optane DC persistent memory in App Direct mode. The performance gains were attributed to the ability to store more data in memory with reduced spillover to slower SSDs. SAS VIYA* is a unified, open analytics platform with AI capabilities deployed on the cloud. Using Intel Optane DC persistent memory mode, larger datasets needed for gradient boosting models could be placed in memory, with little or no performance degradation, at reduced costs. The performance gain was up to 18%.

Faster AI Inference with Intel® AVX-512 VNNI

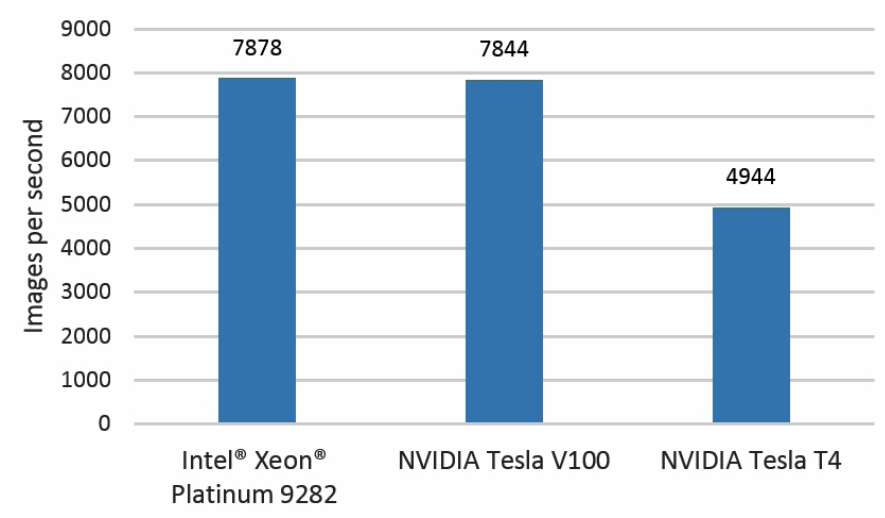
Neural networks require several matrix manipulations, which may be realized using MAC instructions. In the previous generation of Intel Xeon Scalable processors, multiplying two 8-bit (INT8) values and accumulating the result to 32 bits required three instructions. In the latest generation of Intel Xeon Scalable processors, this is done in one instruction[8]. This instruction count reduction represents a performance gain, which is accomplished by having simultaneous execution of the MAC instructions on both port 0 and 5 of the execution pipeline (**Figure 5**).



5 **Intel® DL Boost technology**

Currently, Intel® compilers support VNNI instructions through intrinsics and inline assembly only. For users intending to leverage VNNI capabilities without using intrinsics or assembly, **Intel® Math Kernel Library for Deep Neural Networks (Intel® MKL-DNN)**[9] and **BigDL**[10] are the recommended alternatives. Intel MKL-DNN is a collection of highly optimized DL primitives for traditional HPC environments, while BigDL (powered by Intel MKL-DNN) provides similar optimized DL capabilities for big data users in Apache Spark*.

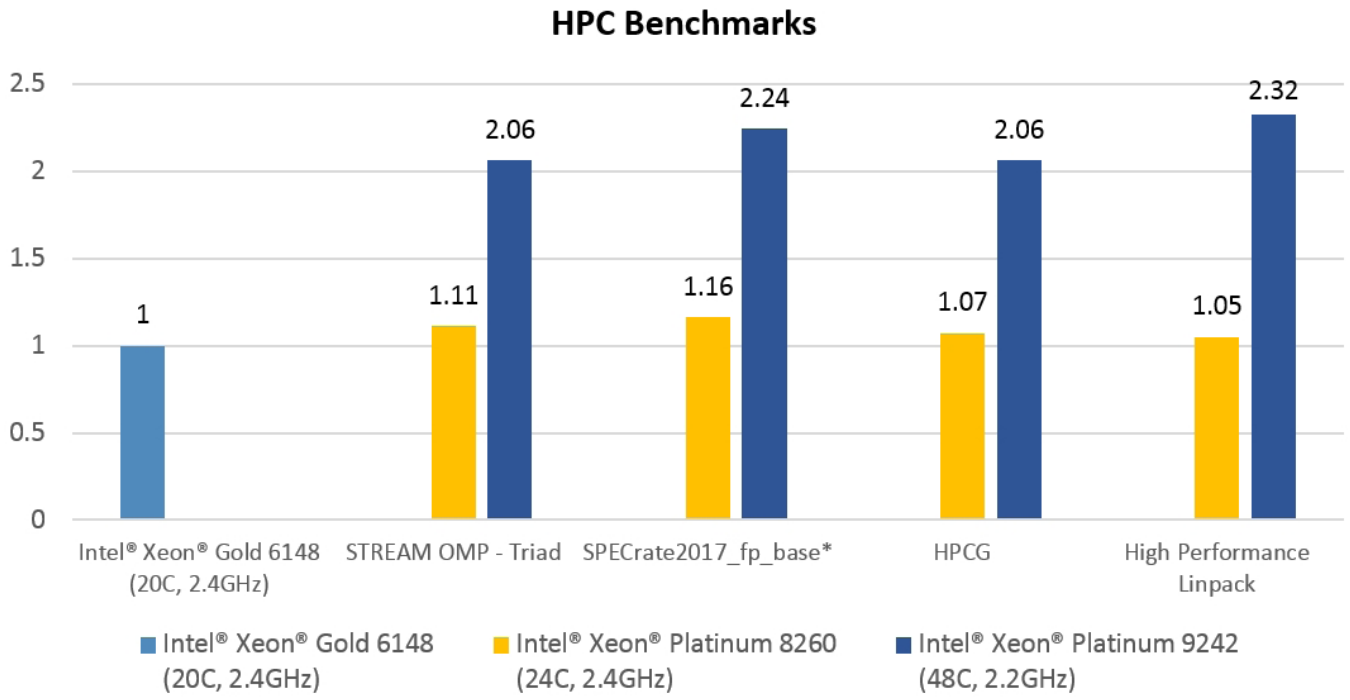
Caffe* 1.1.3 optimized with Intel MKL-DNN gives 14x better inference throughput on a dual-socket Intel® Xeon® Platinum 8280 processor and 30x better inference throughput on a dual-socket Intel Xeon Platinum 9282 processor, in comparison to the previous-generation Intel Xeon Scalable processors[11]. A similar study with Intel Xeon Platinum 9282 processor for Caffe ResNet-50* demonstrated better inference throughput than NVIDIA* GPUs (**Figure 6**)[12]. Other popular frameworks like Chainer*, DeepBench*, PaddlePaddle*, and PyTorch* also use Intel MKL-DNN for better performance.



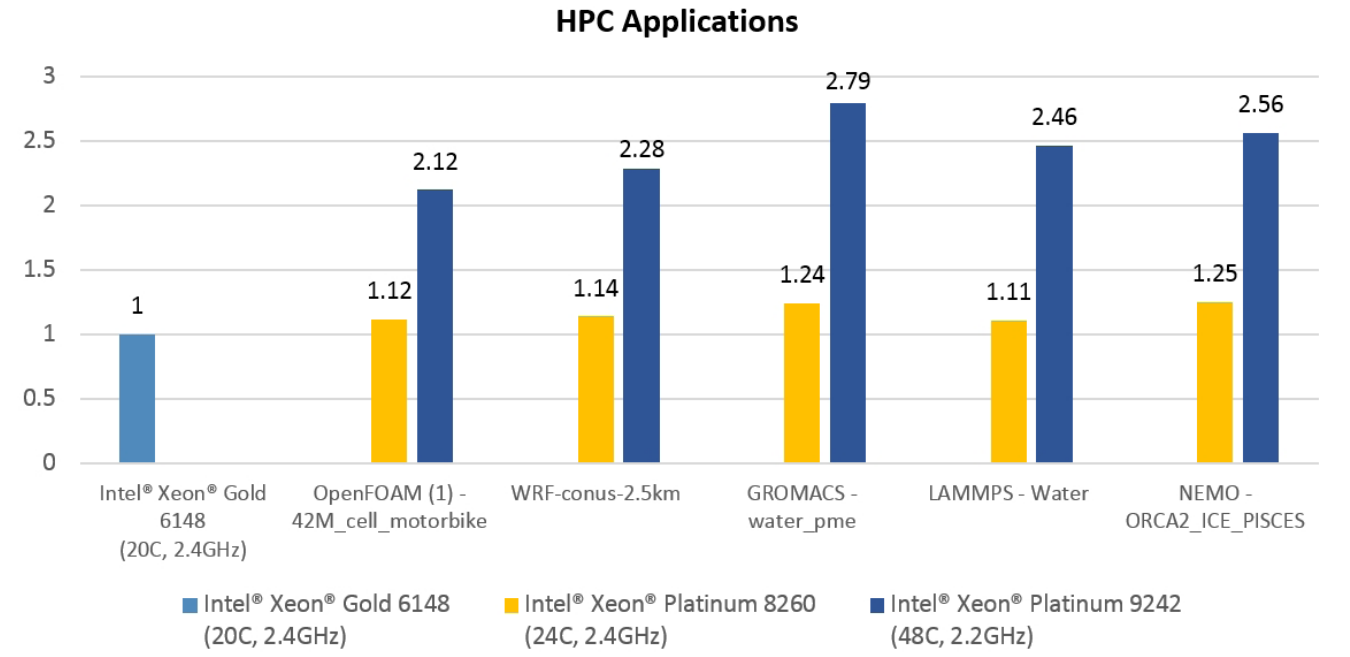
6 **CPU-GPU inference throughput comparison for Caffe ResNet-50* (higher is better)**

HPC Application Performance on Intel Xeon Scalable Processors

The increased core count and higher bandwidth available on Intel Xeon Scalable 8200 and 9200 processors provide substantial gain for HPC applications. Performance gains are shown for industry standard benchmarks in **Figure 7**, and for real-world applications in **Figure 8**. LAMMPS* and GROMACS* benefit from AVX-512, higher core count, and hyperthreading. The higher available bandwidth on Intel Xeon Scalable processors shows a positive gain for memory bandwidth-bound codes like OpenFOAM*, WRF*, and NEMO*.



7 **Relative performance of HPC industry standard benchmarks**



8 **Relative performance of HPC applications**

OpenFOAM Disclaimer: This offering is not approved or endorsed by OpenCFD Limited, producer and distributor of the OpenFOAM software via **www.openfoam.com**, and owner of the OPENFOAM® and OpenCFD® trademark.

Improving HPC, AI, and Analytics Application Performance

New hardware features in the latest Intel Xeon Scalable processors enable developers to improve performance for a wide variety of HPC, AI, and analytics applications. Intel continues to innovate in processor technologies. The upcoming Cooper Lake architecture will introduce bfloat[16] for enhanced AI training support. Also, Intel recently released 10th-generation Intel® Core™ processors on 10nm, delivering better performance and density improvements.

References

1. **Data-Centric Business Update Media Briefing**
2. **Addressing Hardware Vulnerabilities**
3. **Intel® Optane™ DC Persistent Memory Operating Modes Explained**
4. **ipmctl for Intel Optane DC Persistent Memory**
5. **Configure Intel® Optane™ DC Persistent Memory Modules on Linux***
6. **Persistent Memory Development Kit**
7. **PMDK on GitHub**
8. **Intel® Architecture Instruction Set**
9. **Intel® Math Kernel Library for Deep Neural Networks (Intel® MKL-DNN)**
10. **BigDL: Distributed Deep Learning Library for Apache Spark***
11. **2nd-Generation Intel® Xeon® Scalable Processors, Solutions**
12. **Intel® CPU Outperforms NVIDIA* GPU on ResNet-50 Deep Learning Inference**

NEWS HIGHLIGHTS

Intel's 'oneAPI' Project Delivers Unified Programming Model Across Diverse Architectures

INTEL CORPORATION

At Intel's Software Technology Day in London, Intel engineering leaders provided an update on Intel's software project–"oneAPI"–to deliver a unified programming model to simplify application development across diverse computing architectures.

"oneAPI is a project to deliver a set of developer tools that provide a unified programming model that simplifies development for workloads across diverse architectures. As our breadth of compute has grown to include specialized accelerators, Intel will deliver software solutions that allow developers to get the full performance out of the hardware," said Bill Savage, Intel vice president and general manager of Compute Performance Developer Products.

Read more >

Appendix

Configuration: Single-Node Intel® Xeon® Generational HPC Performance

Intel® Xeon® 6148 processor: Intel Reference Platform with 2S 6148 Intel processors (2.4GHz, 20C), 12x16GB DDR4-2666, 1 SSD, Cluster File System: Panasas (124 TB storage) Firmware v6.3.3.a & OPA based IEEL Lustre, BIOS: SE5C620.86B.00.01.0015.110720180833, Microcode: 0x200004d, Oracle Linux Server release 7.6 (compatible with RHEL 7.6) on a 7.5 kernel using ksplice for security fixes, Kernel: 3.10.0-862.14.4.el7.crt1.x86_64, OFED stack: OFED OPA 10.8 on RH7.5 with Lustre v2.10.4.

Intel® Xeon® Platinum 8260 processor: Intel Reference Platform with 2S 8260 Intel processors (2.4GHz, 24C), 12x16GB DDR4-2933, 1 SSD, Cluster File System: Panasas (124 TB storage) Firmware v6.3.3.a & OPA based IEEL Lustre, BIOS: SE5C620.86B.0D.01.0286.011120190816, Microcode: 0x4000013, Oracle Linux Server release 7.6 (compatible with RHEL 7.6) on a 7.5 kernel using ksplice for security fixes, Kernel: 3.10.0-957.5.1.el7.crt1.x86_64, OFED stack: OFED OPA 10.9 on Oracle Linux 7.6 (Compatible w/RHEL 7.6) w/Lustre v2.10.6.

Intel® Xeon® Platinum 9242 processor: Intel Reference Platform with 2S Intel Xeon 9242 processors (2.2GHz, 48C), 24x16GB DDR4-2933, 1 SSD, Cluster File System: 2.12.0-1 (server) 2.11.0-14.1 (client), BIOS: PLYXCRB1.86B.0572.D02.1901180818, Microcode: 0x4000017, CentOS 7.6, Kernel: 3.10.0-957.5.1.el7.x86_64

TOOL HIGHLIGHTS

Intel® Advisor Performance Analysis Cookbook
INTEL CORPORATION

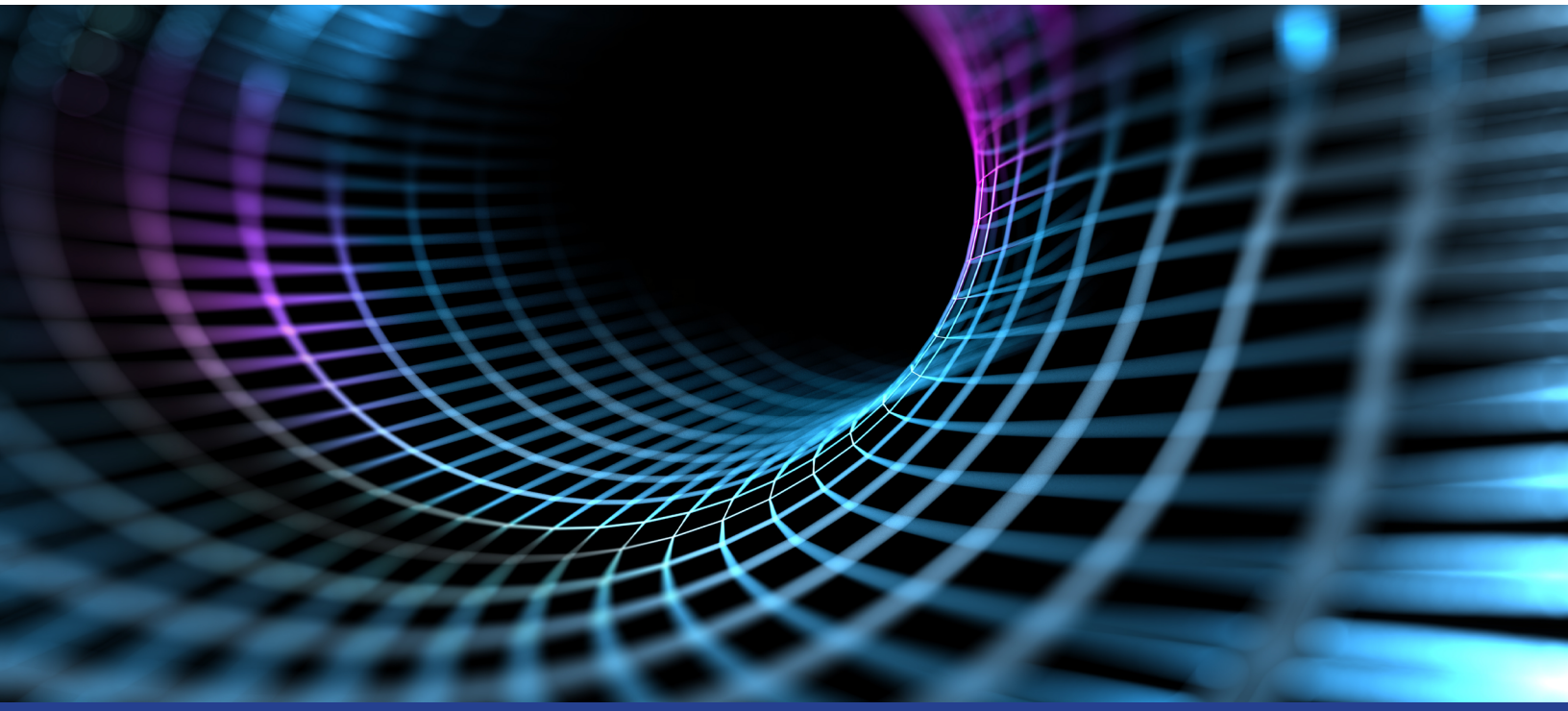
Learn how to optimize memory access patterns using loop interchange and cache blocking with Intel® Advisor, which helps you identify memory bottlenecks and improve performance.

This step-by-step cookbook shows you how to:
1. Establish a baseline
2. Perform a loop interchange
3. Examine memory trafc at each level of the memory hierarchy
4. Implement a cache-blocking strategy

Read more >

Intel® Xeon® 6148/8260/9242 Processors

Application	Workload	Intel® Compiler	Intel® Software CoSoftware	BIOS Settings
STREAM OMP 5.10	Triad	2019u2		HT=ON, Turbo=OFF, 1 thread per core
HPCG 2018u3	Binary included MKL	2019u1	MPI 2019u1, MKL 2019u1	HT=ON, Turbo=OFF, 1 thread per core
SPECrate2017_fp_base	Best published result as of June 20, 2019: • **6148** • **8260** • **9242**			
HPL 2.1	Binary included MKL	2019u1	MKL 2019, MPI 2019u1	HT=ON, Turbo=OFF, 2 threads per core
WRF 3.9.1.1	conus-2.5km	2018u3	MPI 2018u3	HT=ON, HT=ON, 1 threads per core
GROMACS 2018.2	All workloads	2019u2	MKL 2019u2, MPI 2019u2	HT=ON, Turbo=OFF, 2 threads per core
LAMMPS 12 Dec 2018	All workloads	2019u2	MPI 2019u2	HT=ON, Turbo=ON, 2 threads per core
OpenFOAM 6.0	42M_cell_motorbike	2019u1	MPI 2018u3	HT=ON, Turbo=OFF, 1 thread per core
NEMO v4.0	ORCA2_ICE_PISCES	2018u3	MPI 2018u3	HT=off, TURBO=ON, pure MPI run



USING THE LATEST PERFORMANCE ANALYSIS TOOLS TO PREPARE FOR INTEL® OPTANE™ DC PERSISTENT MEMORY

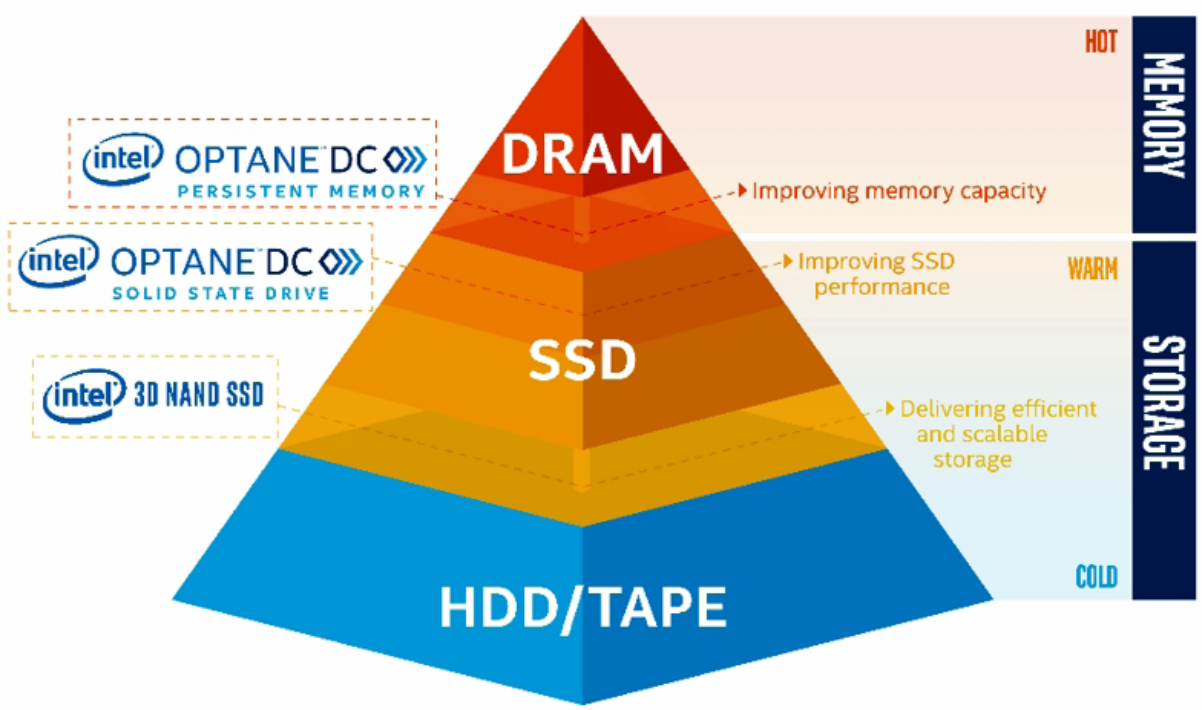
Getting Past Bottlenecks and Storage Issues

Jackson Marusarz, Technical Consulting Engineer, and Kevin O'Leary, Senior Technical Consulting Engineer, Intel Corporation

We have some good news and some bad news. First, the bad news: With the exponential growth in data year after year, and advances in fields like data analytics and artificial intelligence, many applications are becoming bottlenecked by the available system memory or fast storage on a platform. The good news: **Intel® Optane™ DC persistent memory** has arrived.

This new technology introduces a nonvolatile memory/storage tier that's faster than SSDs or hard drives, with latencies near DRAM and much larger capacity. It has implications for any workloads that are currently bound by memory capacity or the slow speeds of storage devices.

Figure 1 shows how Intel Optane DC persistent memory slots into the memory hierarchy of current platforms. This article will help you understand how you can use Intel® tools to profile your existing workloads and evaluate how they can benefit from this new hardware.



1 **The new memory hierarchy**

Intel Optane DC persistent memory can be configured in two different modes:

1. **Memory Mode**
2. **App Direct Mode**

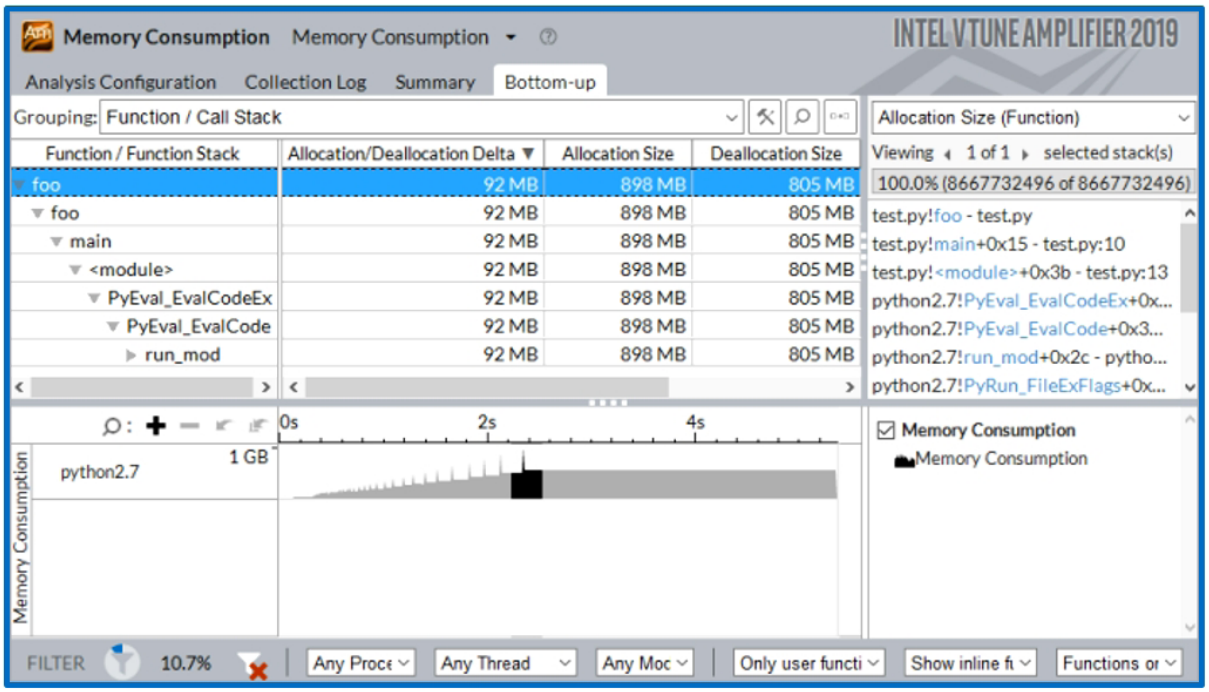
In Memory Mode, Intel Optane DC persistent memory extends the system memory available to the operating system. DRAM is used as a cache for Intel Optane DC persistent memory, and all the memory management is transparent to the user. No code modifications are required.

In App Direct Mode, users manually allocate objects on Intel Optane DC persistent memory via APIs and can also use the memory as traditional storage. This mode enables the non-volatile (persistent) capabilities of the technology.

To determine how your workloads can benefit from Intel Optane DC persistent memory, and which mode to choose, it's important to characterize the behavior and understand specific performance metrics. Intel has tools to help with this process.

Measure the Memory Footprint of the Application

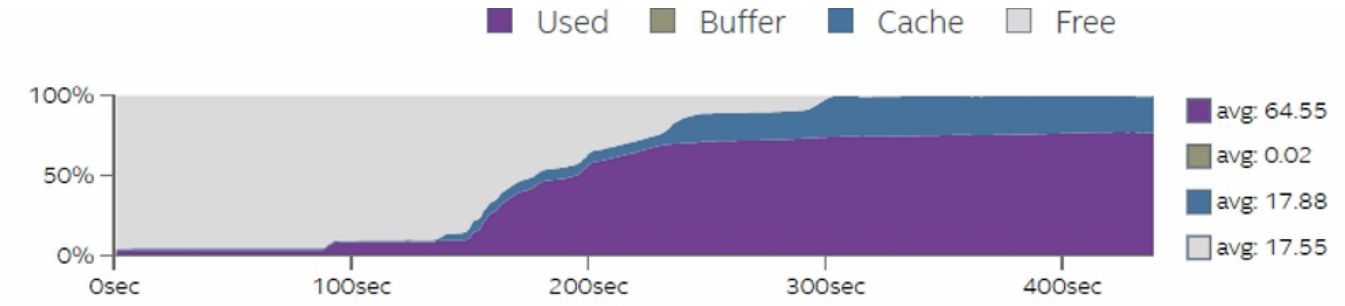
If you're planning to use Intel Optane DC persistent memory as additional system memory—in either mode—the first metric to understand is the memory footprint of your workload. There are many tools that can measure memory consumption, including **Intel® VTune™ Amplifier**. The **Memory Consumption analysis** in Intel VTune Amplifier will monitor the allocations and deallocations of an application and track the memory consumption over time (**Figure 2**).



2 **Memory Consumption report**

The timeline in the Bottom-Up view of the Memory Consumption report can be used to identify the high-water mark of memory usage for the workload. Also, the **Platform Profiler** feature in Intel VTune Amplifier can track memory consumption using OS statistics and provide a timeline as a percentage of available memory (**Figure 3**).

To improve performance with Intel Optane DC persistent memory, the application should benefit from more physical memory. This means the memory consumption should be close to the total amount of DRAM available on the system. Since physical memory is a finite resource, you need to consider that the operating system and other processes also consume memory. If the memory footprint, plus the expected usage of these other memory consumers, is near the available DRAM size, it ensures the application can use the Intel Optane DC persistent memory because it can't fit all of its data in DRAM. If available memory isn't the limiting factor for your workload, then adding more memory probably isn't going to improve performance.



3 **Platform Profiler Memory Utilization analysis**

Identify the Hot Working Set Size

If you determine that your workload is consuming most of the available memory, then you may have a good candidate for Intel Optane DC persistent memory.

The next step is to determine how your application might behave in each mode, Memory or App Direct. The key metric for this step is the hot working set size. The hot working set is made up of the set objects frequently accessed by your application. And the hot working set size is the sum of the sizes of these objects. This metric isn't as straightforward to calculate as the footprint, since the line of what is frequently and infrequently accessed isn't always clearly defined. However, the Memory Access Analysis in Intel VTune Amplifier, with the knob to analyze dynamic memory objects enabled, can help.

After running a Memory Access analysis, the Bottom-Up view in the GUI will display a grid that lists each memory object that was allocated by the application, its size in parentheses, and the number of loads and stores that accessed it (**Figure 4**). Identify the objects with the most loads and stores. Sum up the sizes (the values in parentheses) of these objects to get the hot working set size.

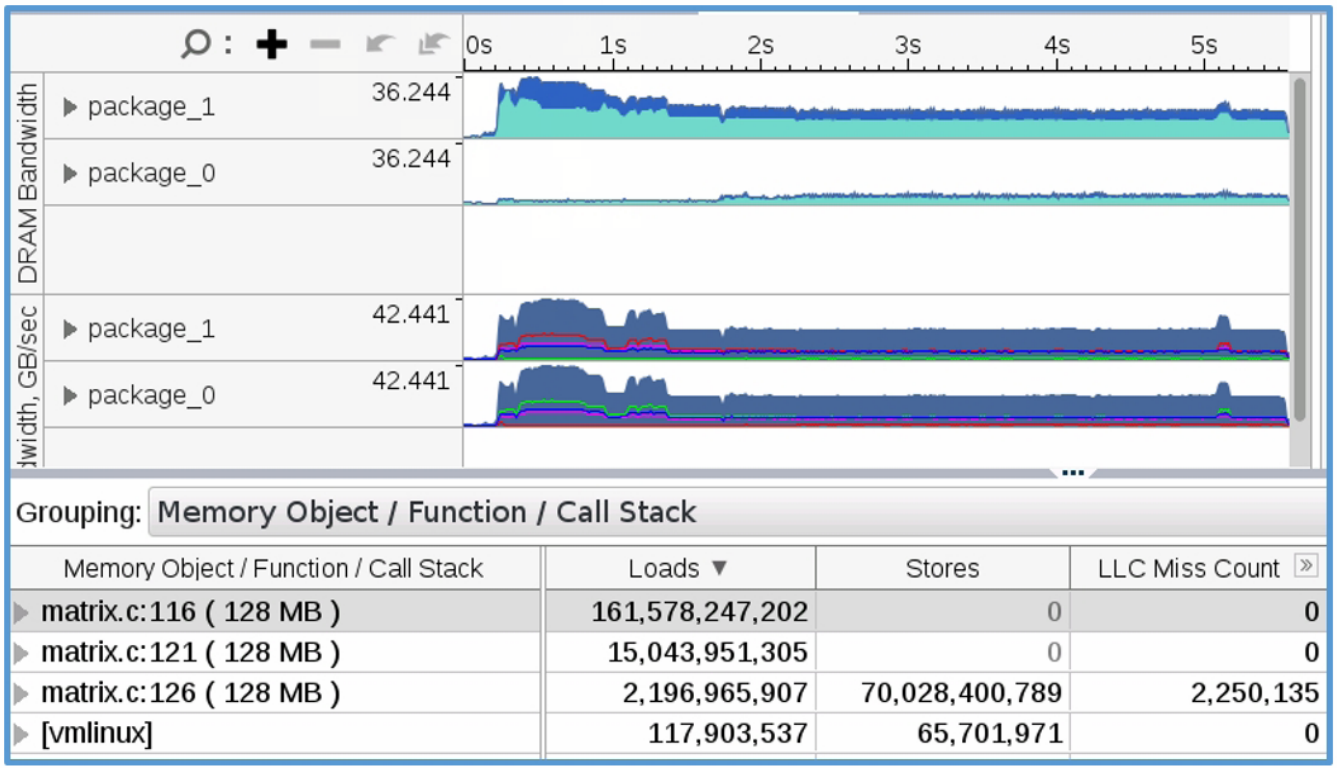
The size of your hot working set is important for determining how your application will behave in each of the memory modes.

Considerations for Choosing a Memory Configuration and Mode

The important concept to remember when you're thinking about persistent memory performance is that you still want the majority of memory accesses to come from DRAM. The persistent memory acts as additional memory that can be used when DRAM isn't available.

Based on that concept, Memory Mode could be a good solution for applications whose hot working set fits into DRAM (i.e., the hot working set size calculated in the last step should be smaller than the available DRAM

on the system). This will ensure that the working set will routinely be cached in DRAM and, as long as the memory footprint is smaller than the available persistent memory, the remaining data will sit in Intel Optane DC persistent memory instead of out on disk.



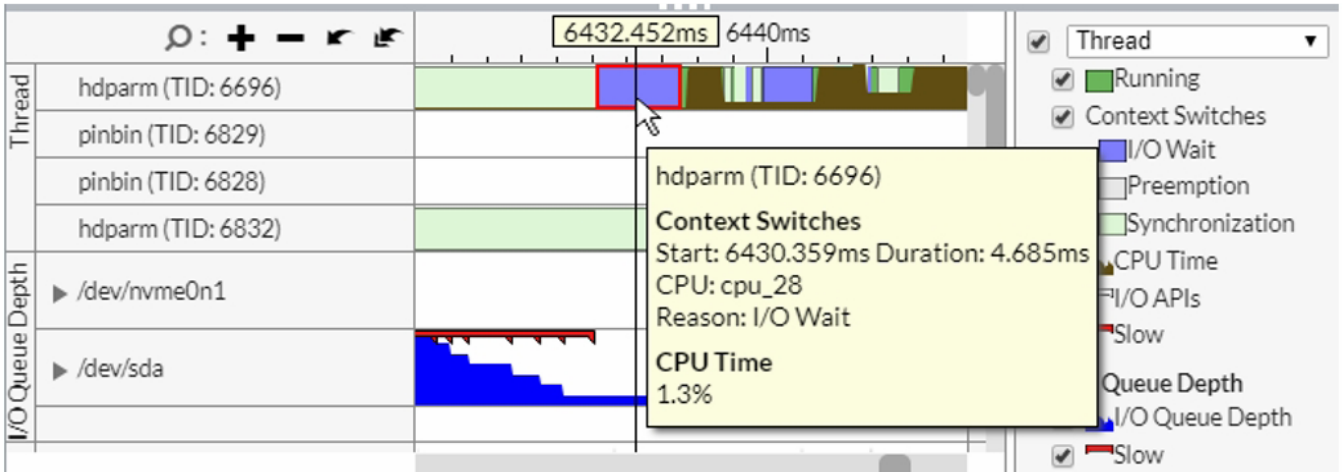
4 **Memory Access Analysis report with Dynamic Memory Object Analysis**

If the hot working set size is much larger than the available DRAM, it's a good indication that persistent memory in App Direct mode could be a better solution than Memory mode. App Direct mode requires the user to explicitly define which objects should be allocated in DRAM and which should be allocated in Intel Optane DC persistent memory. It's important to make educated choices, since allocating incorrectly could hurt application performance. A good starting heuristic for choosing where to allocate objects is identifying the objects with the most last-level core cache (LLC) misses and allocating as many as possible into the available DRAM. The Memory Access analysis in Intel VTune Amplifier (**Figure 4**) has this information. This ensures they will have lower access latency compared to the latency of Intel Optane DC persistent memory. As for the remaining objects that have fewer LLC misses or are too large to put in DRAM, allocate them in Intel Optane DC persistent memory.

One additional consideration for allocation is the load/store ratio for object accesses. Intel Optane DC persistent memory loads are generally much faster than stores. Identify objects with high load/store ratios (load-heavy objects) and allocate them in persistent memory. Allocate the store-heavy objects in DRAM. The load and store counts can also be found with the Memory Access analysis.

Using Intel Optane DC Persistent Memory for Non-Volatile Storage

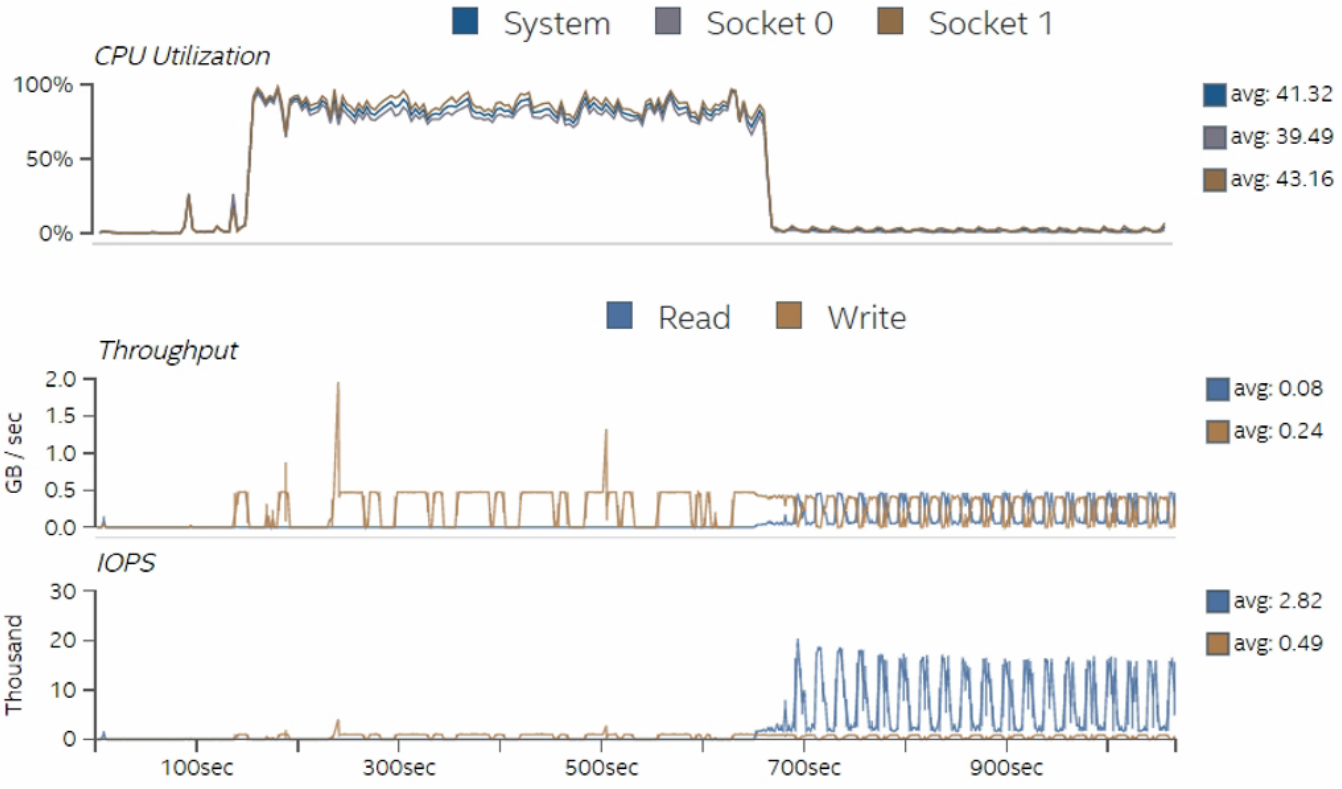
The uses for Intel Optane DC persistent memory as non-volatile storage are fairly straightforward. If your application has any performance issues related to reading and writing to disk, this new technology could give you a boost. Many developers are already aware of disks being their bottleneck. If this is you, then you're one step ahead. If you aren't sure whether storage is causing performance issues, there are features in Intel® tools to help. For instance, the **Input and Output Analysis** in Intel VTune Amplifier helps diagnose CPU stalls correlated with disk accesses (**Figure 5**).



5 **Intel VTune Amplifier Input and Output analysis**

Also, the Platform Profiler analysis in Intel VTune Amplifier displays disk statistics that can be correlated with CPU performance (**Figure 6**).

Use these metrics to identify performance bottlenecks from storage accesses. If this is causing a significant performance issue, using Intel Optane DC persistent memory as fast and persistent storage could increase performance. The persistent memory can be configured as part of the filesystem, and you can put your most accessed files directly on the memory modules.

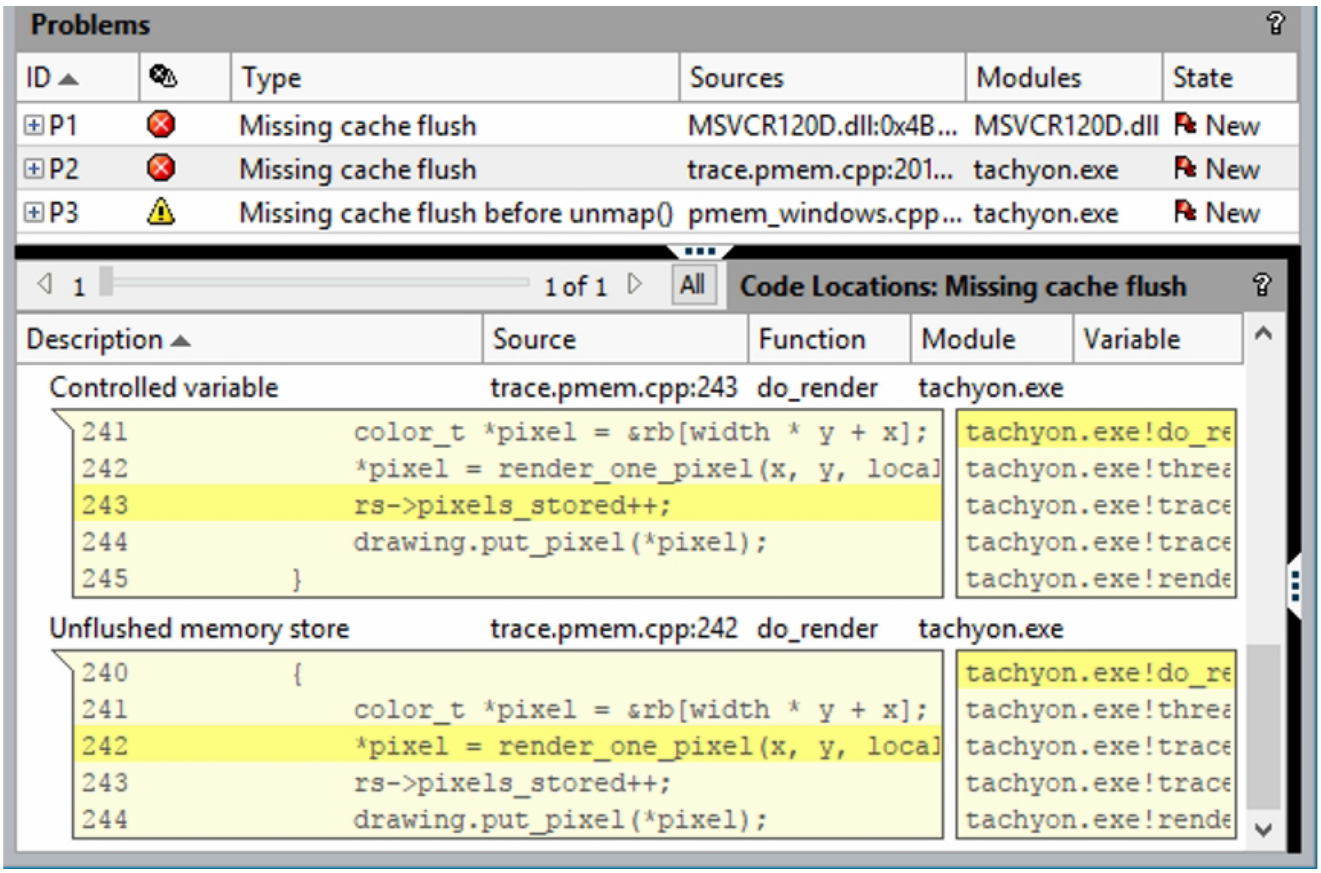


6 **Platform Profiler metrics for CPU utilization and disk usage**

Verifying the Correctness of Persistent Memory Applications

Besides identifying performance issues, there are also some software challenges to programming persistent memory applications. One challenge is that a store to persistent memory is not actually persistent until after it's out of the cache hierarchy and visible to the memory subsystem. **Intel® Inspector** Persistence Inspector is a new runtime tool developers can use to detect potential errors (**Figure 7**). In addition to cache flush misses, this tool detects:

- **Redundant** cache flushes and memory fences
- **Out-of-order** persistent memory stores
- **Incorrect** undo logging for the Persistent Memory Development Kit (PMDK)



7 **Intel® Inspector Persistence Inspector**

Getting Past Bottlenecks and Storage Issues

We've just scratched the surface of the possibilities this new technology enables. If you've been struggling with the rise of big data and have performance issues related to limited system memory or fast storage, Intel Optane DC persistent memory is here to help. Intel also has tools like Intel VTune Amplifier and Intel Inspector to help you understand how your workloads may be limited by these issues and how you can take advantage of persistent memory.

To learn more, check out the Intel Optane DC persistent memory **webpage** and the software tools **landing page**.

MEASURING THE IMPACT OF NUMA MIGRATIONS ON PERFORMANCE

Weighing the Tradeoffs to Maximize Performance

Gurbinder Gill, Graduate Research Assistant, University of Texas at Austin, and Ramesh V. Peri, Senior Principal Engineer, Intel Corporation

These days, memory systems use non-uniform memory access (NUMA) architectures, where cores and the total DRAM are divided among sockets. Each core can access the whole memory as a single address space. However, accessing the memory local to its local socket is faster than the remote socket—hence the non-uniform memory access. Because of the different access latency, access to the local socket memory should always be preferred.

To achieve this, the Linux* kernel does NUMA migrations, which try to move memory pages to the sockets where the data is being accessed. Linux maintains bookkeeping information—such as the number of memory accesses to the pages from a given socket and latency of accesses—to make

decisions regarding page migration. NUMA migrations in Linux are enabled by default unless an OS-level NUMA allocation policy is specified using utilities such as `numactl`.

NUMA page migrations can be very useful in scenarios where multiple applications are running on a single machine, each with its own memory allocation. In such a multi-application scenario, where the system is being shared, it makes sense to move memory pages belonging to a particular application closer to the cores assigned to that application.

In this article, we'll argue that if a single application is using the entire machine—which is the most common scenario for high-performance applications—NUMA migrations can actually hurt performance. Also, using application-level NUMA allocation policies is often preferred over OS-level utilities such as `numactl` because they give finer control over the allocation of different data structures and design allocation policies.

We'll look at two application-level NUMA allocation polices (**Figure 1**):

- **NUMA interleave**, in which memory pages are equally distributed among NUMA sockets in round-robin fashion (similar to the `numactl -interleave all` command).
- **NUMA blocked**, in which equal chunks of the allocated memory are divided among NUMA sockets.

1 **NUMA allocation policies (color-coded for two processors)**

Evaluation on Intel® Xeon® Gold Processors

We'll evaluate the efficacy of NUMA migrations using a simple microbenchmark that allocates m amount of memory (using both NUMA interleaved and blocked policies) and writes to each location once using t threads such that each thread gets a contiguous block to write sequentially.

The pseudocode code memory allocation policies and simple computation are shown in **Figure 2** and **Figure 3**, respectively. The experiments are conducted on a four-socket system with Intel® Xeon® Gold 5120 processors (56 cores with a clock rate of 2.2 GHz and 187GB of DDR4 DRAM). Hyperthreading was disabled during our evaluation.

```
1. /*** Memory allocation Policies ***/
2. /*
3.  * Let m be the amount of memory to allocate.
4.  * Default page size (PS) is 4KB.
5.  * Use MAP_HUGETLB for huge page size (2MB).
6.  * Touching pages bring them into the memory.
7.  */
8.  _PROT = PROT_READ | PROT_WRITE;
9.  _MAP_FLAGS   = MAP_PRIVATE | MAP_ANONYMOUS;
10. mmap_ptr = mmap(0, m, _PROT, _MAP_FLAGS, -1, 0);
11.
12./** NUMA Interleave **/
13./*
14. * Round robin page distribution among threads (e.g. thread 0 gets
15. * a page, then thread 1, then thread n, then back to thread 0 and
16. * so on until the end of the region).
17. */
18.
19./* Parallel loop */
20.for_each_thread{
21.    for(x = PS*ThreadID; x < m; x += PS*numThreads)
22.    {
23.      touch(mmap_ptr[x]); /* bring page in-memory */
24.    }
25.}
26.
27./** NUMA Blocked **/
28./*
29. * Distribute equal chunks of memory among threads (e.g. thread 0 gets
30. * first chunk, thread 1 gets next chunk, ... last thread gets last
31. * chunk).
32. */
33.CS = m/numThreads; /* CS is Chunk Size */
34.
35./* Parallel loop */
36.for_each_thread{
37.    for(x = CS*ThreadID; x < m && x < (ThreadID + 1) * CS; x += PS)
38.    {
39.      touch(mmap_ptr[x]); /* bring page in-memory */
40.    }
41. }
```

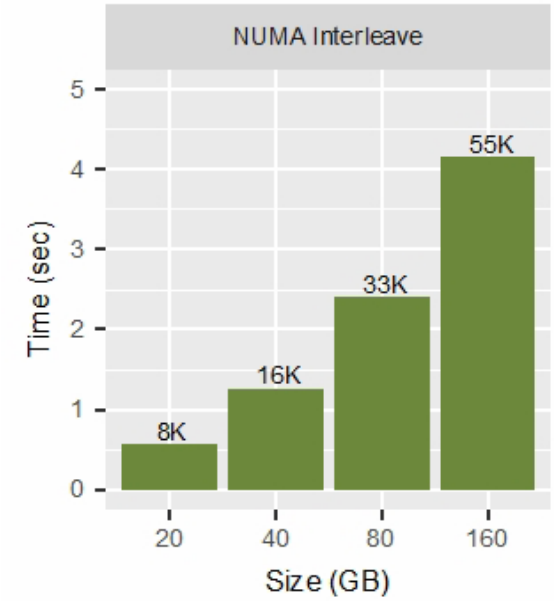
2 **Pseudocode showing the memory allocation policies: NUMA Interleaved and NUMA Blocked**

```
1. /*** Microbenchmark Computation ***/
2. /*
3.  * Write to every 32nd location in m (memory allocated).
4.  * 32 is randomly chosen.
5.  */
6. NE = m/32; /* Total number of entries. */
7. ECS = NE/numThreads; /* Entries Chunk Size */
8.
9. /* Parallel loop */
10.for_each_thread{
11.   for(x = ECS*ThreadID; x < NE && x < (ThreadID + 1) * ECS; x += 1)
12.     mmap_ptr[x*32] = 42;
13.}
```

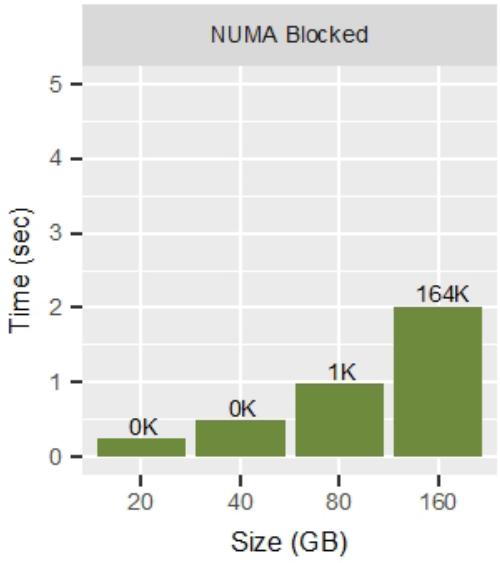
3 **Pseudocode showing the simple microbenchmark computation used in this study**

Effect of NUMA Migration for Different NUMA Allocation Policies

Figure 4 shows the time of the microbenchmark using t = 56 threads and interleaved allocation as memory allocation size (m) increases (**Figure 1**). Doubling the workload doubles the execution time, which is expected. However, the number of pages migrated during execution also increases significantly. We observe a similar pattern for NUMA blocked allocation (**Figure 5**). However, blocked allocation gives better performance because no page migration is required up to a workload size of 40GB. The memory pages are allocated and accessed locally during the computation.



4 **Microbenchmark using 56 threads and NUMA interleaved allocation with increasing workload size. The number on the bars shows how many memory pages migrated (in thousands).**

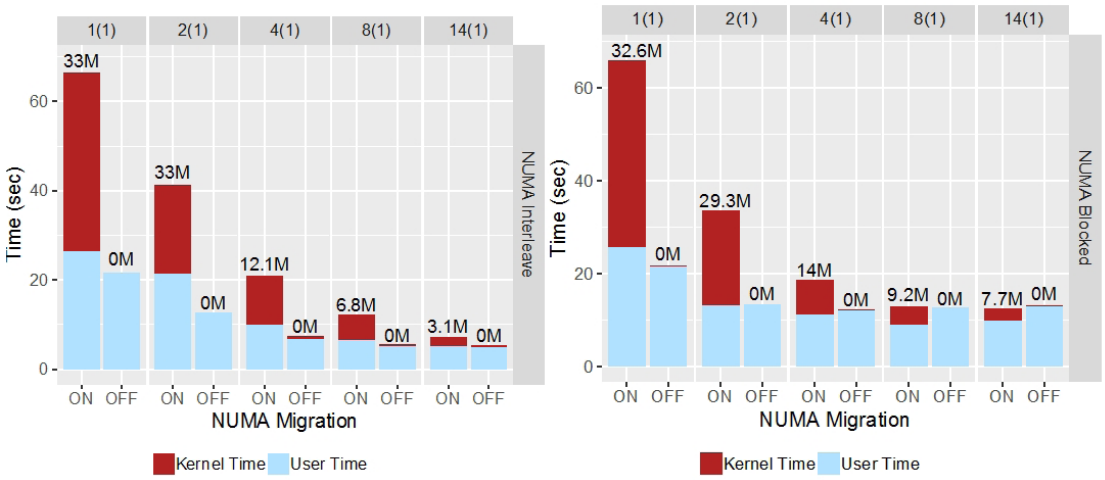


5 **Microbenchmark using 56 threads and NUMA blocked allocation with increasing workload size. The number on the bars shows how many memory pages migrated (in thousands).**

Effect of NUMA Migration on a Single Socket

Figure 6 shows the total time taken with a 160GB workload using different numbers of threads on a single socket, as well as the time spent in user code and kernel code. Since the total memory is equally divided among sockets, each socket will have approximately 47GB of memory (187GB divided among four sockets). We allocated 160GB across all four sockets. The microbenchmark scales with the number of threads for both allocation policies. Increasing the number of threads decreased execution time, which in turn reduces the number of pages migrated because the longer an application runs, the more pages will be migrated by the OS kernel.

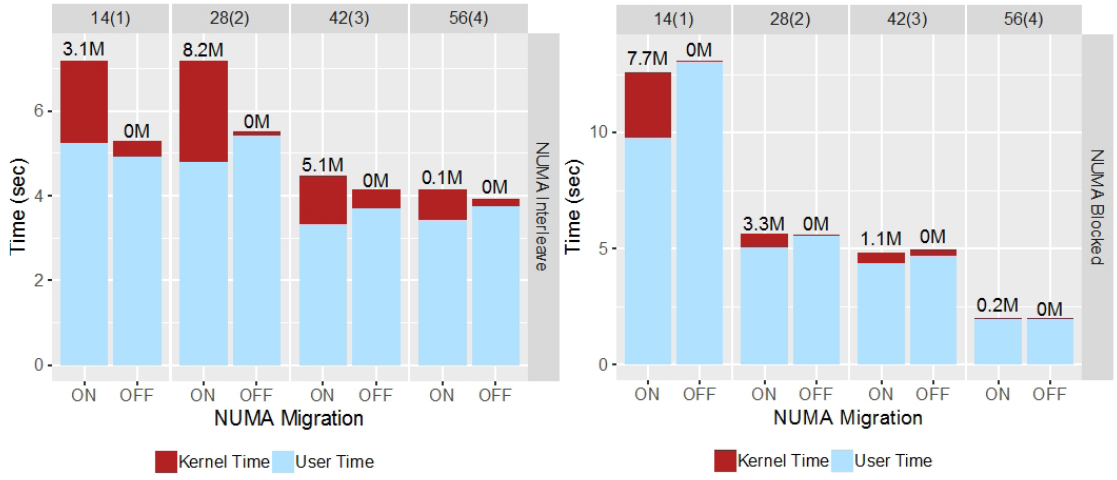
The red part of the stacked plots shows the time spent in the kernel code to migrate pages. This is reduced to almost zero when NUMA migration is disabled. The geomean speedup gained by turning off NUMA migrations is 2.4x for interleave and 1.6x for blocked, which shows that NUMA migration has a significant impact on performance.



6 **Microbenchmark with fixed workload size (160GB) using a different number of threads on a single socket. The number of threads is shown at the top, with the number of sockets in parentheses. The number on the bars shows how many memory pages migrated (in millions).**

The Effect of NUMA Migration across Multiple Sockets

A pattern similar to a single socket (**Figure 6**) is also observed as we go beyond one socket (**Figure 7**). Each chart shows the performance with and without NUMA migration as the number of sockets increases. All the cores on the sockets are used. Note that the time spent in the kernel is always reduced when NUMA migration is disabled. Another interesting thing to note is that the time spent in the user code increases slightly when NUMA migration is disabled, indicating that NUMA migrations reduce memory access latency. However, the overhead of NUMA migrations can outweigh the benefits and end up hurting overall performance.



7 **Microbenchmark with fixed workload size (160GB) using a different number of threads on different sockets. The number of threads is shown at the top, with the number of sockets in parentheses. The number on the bars shows how many memory pages migrated (in millions).**

Maximizing Performance

From our results, we can conclude that OS-level features such as NUMA migrations must be used with caution because they can have significant performance overhead, especially for single applications running on the entire machine, the most common scenario for high-performance computations.

The effect of NUMA migrations on the runtime of an application depends on various factors such as:

- **Type** of NUMA allocation policies used
- **Number** of sockets used on the processor

To avoid the performance noise introduced by NUMA page migrations, ensure that such OS-level features are turned off (NUMA migrations are on by default) as shown in **Figure 8**:

```
# echo 0 > /proc/sys/kernel/numa_balancing
```

8 **Turning off OS-level features**

References

1. **Linux numactl utility**
2. David Ott, "**Optimizing Applications for NUMA**," Intel Corporation, 2011.

PARALLELISM IN PYTHON*: DIRECTING VECTORIZATION WITH NUMEXPR*

Boosting Performance for Computing with Arrays and Numerical Expressions

Fabio Baruffa, PhD, Technical Consulting Engineer, Intel Corporation

Python* has several pathways to vectorization (i.e., instruction-level parallelism), ranging from just-in-time (JIT) compilation with Numba*[1] to C-like code with Cython*. One interesting way of achieving Python parallelism is through NumExpr*, in which a symbolic evaluator transforms numerical Python expressions into high-performance, vectorized code. NumExpr achieves this by vectorizing in chunks of elements instead of compiling everything at once—thus creating accelerated object kernels that are usable from Python code. In this article, we'll explore how to refactor Python code to take advantage of NumExpr's capabilities.

Parallelization of Numerical Expressions

The flexibility of Python, with its easy syntax, allows developers to rapidly prototype numerical computations with the help of libraries like NumPy* and SciPy*. But the Python language wasn't developed with parallelism in mind—although it's a key requirement to get performance out of modern vector and multicore processors. So how is it possible to vectorize numerical expressions using Python?

A numerical expression is a mathematical statement that involves numbers and mathematical symbols to perform a calculation (e.g., `11*a-42*b`). In Python, this expression can also operate on arrays `a` and `b` defined from the NumExpr package. In this case, similar expressions working on arrays are accelerated, making use of intrinsic parallelism and vectorization, compared to the same calculation in standard Python.

To boost performance, NumExpr can use the optimized Intel® Vector Mathematical Function Library (Intel® VML), included in **Intel® Math Kernel Library (Intel® MKL)**. This makes it possible to accelerate the evaluation of mathematical functions (e.g., sine, exponential, or square root) that operate on vectors stored contiguously in memory.

Refactoring Common NumPy Calls for NumExpr

To make use of the NumExpr package, you only need to pass the computational string to the `evaluate` function. Then it's compiled into an object, leaving the entire computation at low-level code before completion. After that, the result is returned to the Python layer, avoiding too many calls to the Python interpreter.

Let's look at an example where we compute a simple expression for NumPy arrays:

```
import numpy as np
import numexpr as ne
a = np.arange(1e6)
b = np.arange(1e6)
%timeit 11*a-42*b
14.2 ms ± 826 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
%timeit ne.evaluate("11*a-42*b")
3.51 ms ± 248 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In this case, we have a 4x speedup due to the intrinsic vectorization enabled by Intel VML. The library can also perform in-place operations, where the copying overhead is negligible.

Now let's evaluate the speedup from NumExpr when we use a mathematical function, where the benefit of Intel VML becomes more evident:

```
import numpy as np
import numexpr as ne
a = np.arange(1e6)
b = np.arange(1e6)
%timeit np.exp(a)**2 + np.sqrt(b)**3
498 ms ± 11.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
%timeit ne.evaluate("exp(a)**2 + sqrt(b)**3")
26.4 ms ± 2.23 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

In this case, we achieve higher performance due to the optimize `sqrt` function in Intel MKL. The speedup is close to 19x. This indicates that the NumPy library doesn't provide the acceleration we expect for some expressions. Also, the NumExpr implementation circumvents memory allocations for intermediate results, which gives better cache utilization and reduces memory overhead. We can really see the benefit of these optimizations in computations with large arrays.

Controlling the NumExpr Evaluator

Since NumExpr uses the Intel VML library internally, it computes the mathematical functions only for the types the library allows. It also operates on real and complex vector arguments with unit increment, integer, and Boolean. In cases where the types of arrays don't match in the evaluate expression, they're cast according to the usual inference rules.

The performance depends on a number of factors, including vectorization and memory overhead. For this reason, you can use some of Intel VML's functions to tune performance and control numerical accuracy (and eventually the number of threads).

To get information about the Intel VML library version, you can call the function `get_vml_version()`, which might be useful for checking the installation. All the vector functions support the following accuracy modes through the function `set_vml_accuracy_mode(mode)`. The mode can be set to:

- **High**, equivalent to High Accuracy (HA), the default mode.
- **Low**, equivalent to Low Accuracy (LA), which improves performance by reducing accuracy of the two least significant bits.

- **Fast**, equivalent to Enhanced Performance (EP), which provides better performance at the cost of significantly reduced accuracy. Approximately half of the bits in the mantissa are correct.

For more information, see the Intel MKL Developer Reference[2] and the official documentation of NumExpr.[3]

NumExpr can also be used to control the number of threads. The function `set_num_threads(nthreads)` sets the maximum number of threads to be used by Intel VML operations. The return value is the previous setting of the number of threads in the current environment. Let's modify the previous example to use threads to improve performance even further:

```
import numpy as np
import numexpr ad ne
a = np.arange(1e6)
b = np.arange(1e6)
ne.set_num_threads(4)
%timeit ne.evaluate("exp(a)**2 + sqrt(b)**3")
7.2 ms ± 137 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

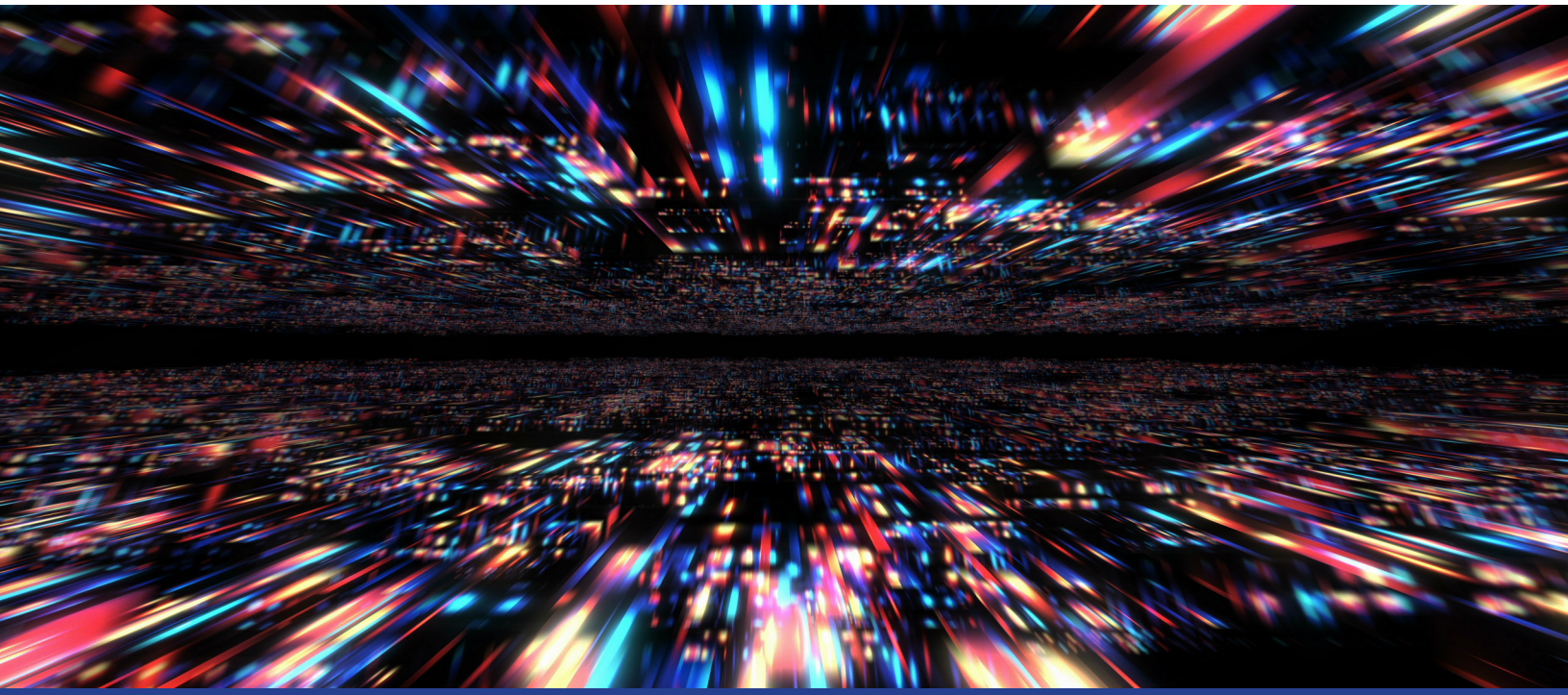
The speedup is 3.7x, with 93% parallel efficiency. In this example, more threads equal better performance.

```
ne.set_num_threads(8)
%timeit ne.evaluate("exp(a)**2 + sqrt(b)**3")
3.94 ms ± 191 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Using NumExpr as alternative to NumPy can give significant performance benefits for computing with arrays and numerical expressions, thanks to the Intel VML performance library. The syntax is very similar to NumPy and, with a couple of easy function calls, you can transition your code to NumExpr.

References

1. ***The Parallel Universe*, issue 36**
2. **Developer Reference for Intel® Math Kernel Library**
3. **NumExpr 2.0 User Guide**



TURBO-CHARGED OPEN SHADING LANGUAGE* ON INTEL® XEON® PROCESSORS WITH INTEL® ADVANCED VECTOR EXTENSIONS 512

Up to 2x Faster Full Renders Speed Digital Content Creation

Steena Monteiro, Software Engineer, and Alex M. Wells, Principal Engineer, Intel Corporation

Oscar*-winning Open Shading Language* (OSL*)[1] is the *de facto* open-source standard for digital content creation. OSL has been adopted industry-wide, used in renderers such as Pixar's RenderMan* and Sony ImageWorks' Arnold*, and in more than 100 movies.[2]

Intel has been leading the rearchitecture of OSL to add single instruction multiple data (SIMD) to leverage **Intel® Advanced Vector Extensions 512 (Intel® AVX-512)** in modern Intel® processors. SIMD OSL uses single program multiple data (SPMD) with existing OSL shaders and OpenMP* explicit vectorization of OSL library functions. This effort can be broadly summarized in two steps:

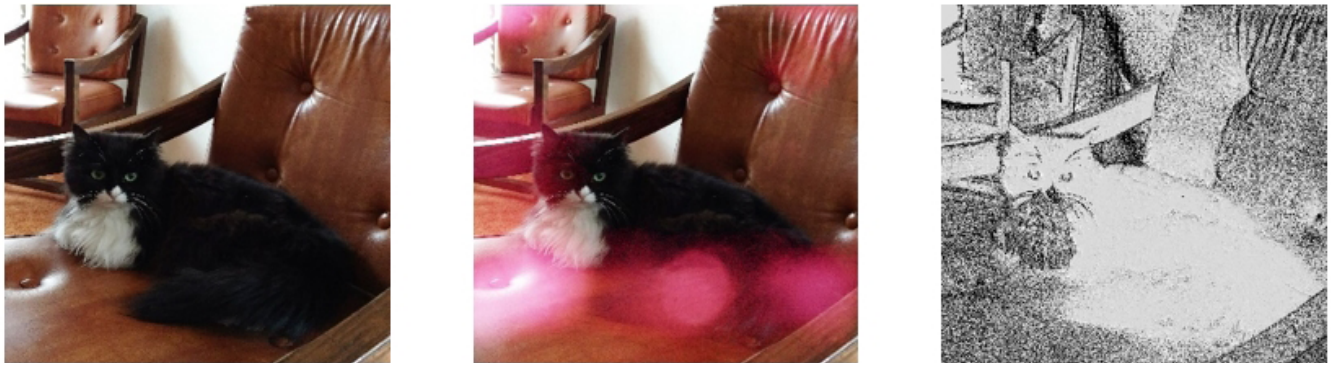
1. **Introducing vector LLVM IR generation** for just-in-time (JIT) compilation during render time optimization
2. **Adding a batched interface** to the OSL runtime

Since its start at SIGGRAPH 2016, SIMD OSL has been improved to natively support AVX*, AVX2*, and AVX-512* and include enhanced library features, debugging support, and an extensive test framework. SIMD OSL enables parallel execution of an entire shading network on Intel SIMD hardware.[4] It dynamically schedules concurrent operations over 8 or 16 data points in a single CPU instruction based on the CPU capabilities. According to Pixar, the company's RenderMan 22.5 now contains "SIMD hardware-accelerated OSL–up to 2x faster full renders, and 15% average speedups using **Intel® Xeon® Scalable processors** with Intel Advanced Vector Extensions."[3,5]

This article explores Intel's efforts in leading the rearchitecture of OSL to leverage the power of Intel AVX-512 on Intel SIMD hardware. We specifically discuss software engineering techniques used in SIMD OSL including strategic memory layout for OSL datatypes, masking for divergent control flows, and addition of an LLVM backend for vector code generation.

Shading and Its Role in Rendering Software

Shading in physically-based renderers implies providing surface description for objects in a 3D scene. Surface descriptions include color values, lighting values (specular, diffused, spot), and textures such as metal, ceramic, and marble (**Figure 1**). Shading in large scenes is done using several individual shader nodes, where each node represents a specific shading behavior. Individual shaders can be connected through directed acyclic graphs to procedurally create complex shading effects. Production shaders can grow to several hundreds and thousands of shaders, representing a multitude of shading behaviors. Renderers in production can spend up to 80% of their time shading via OSL. Due to the complexity of shading networks, shading in a render consumes millions of CPU-hours on render farms, both on-site and in the cloud.



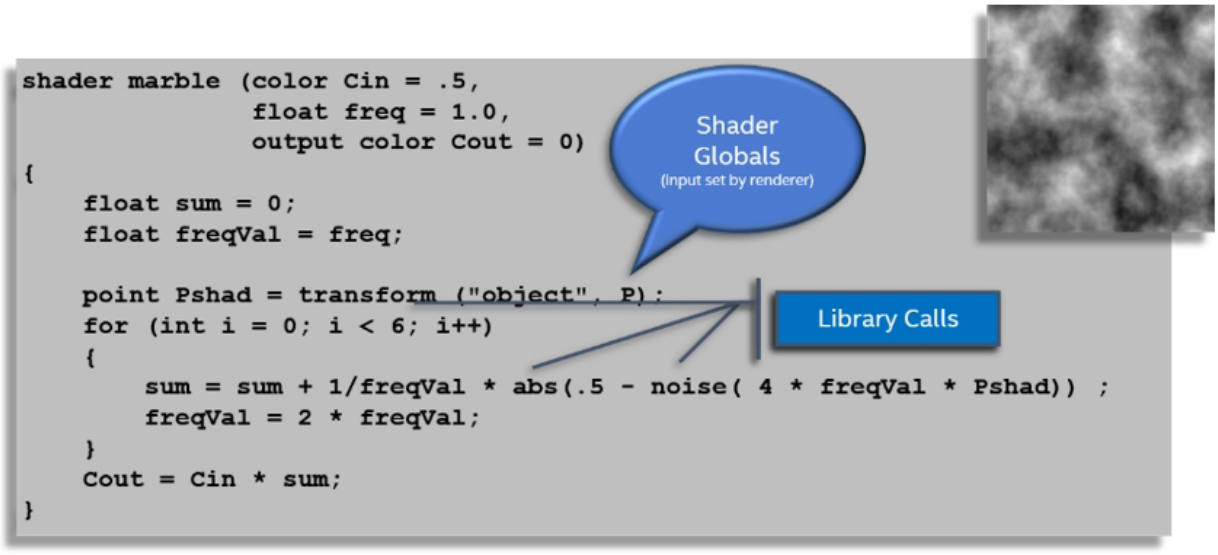
1 **Example of lighting and texture**

Shaders in shading networks can be written in C++. However, using C++ presents a host of challenges. These C++ shaders lack relevant shading information such as values of input parameters and geometry of the scene being shaded at compile time. The shaders don't know the mode of shading required and remain oblivious about the state of the surrounding shading network. Writing, maintaining, and optimizing performance of shaders written in C++ is challenging—primarily, because these shaders lack portability and necessitate tests with complex and nested control flows (branchy testing). Artists who design shading networks need extreme expertise in optimizing C++ to achieve high performance.

OSL, designed by Sony Pictures Imageworks[6], makes writing performance-compliant shading networks a little easier. Structured in C style, OSL is a domain-specific language designed for writing shaders. Designed primarily for physically-based rendering, OSL is restricted to shading and doesn't include raytracing, sampling, integrations, and tight loops (which reside in the renderer). Under the hood, OSL maximizes performance by JIT-ing machine code with extensive runtime optimization and yields shading networks with lazy evaluations.

Open Shading Language

Shaders in OSL are programs with inputs and outputs that perform a specific task when rendering a scene[6]. **Figure 2** shows a simple OSL shader representing a marble texture.[7] The important elements in this shader are the shader global P and the OSL library functions `abs()` and `noise()`. Shader globals are variables such as position, surface normals, and ray directions, provided by a renderer, that are consumed by the shader.



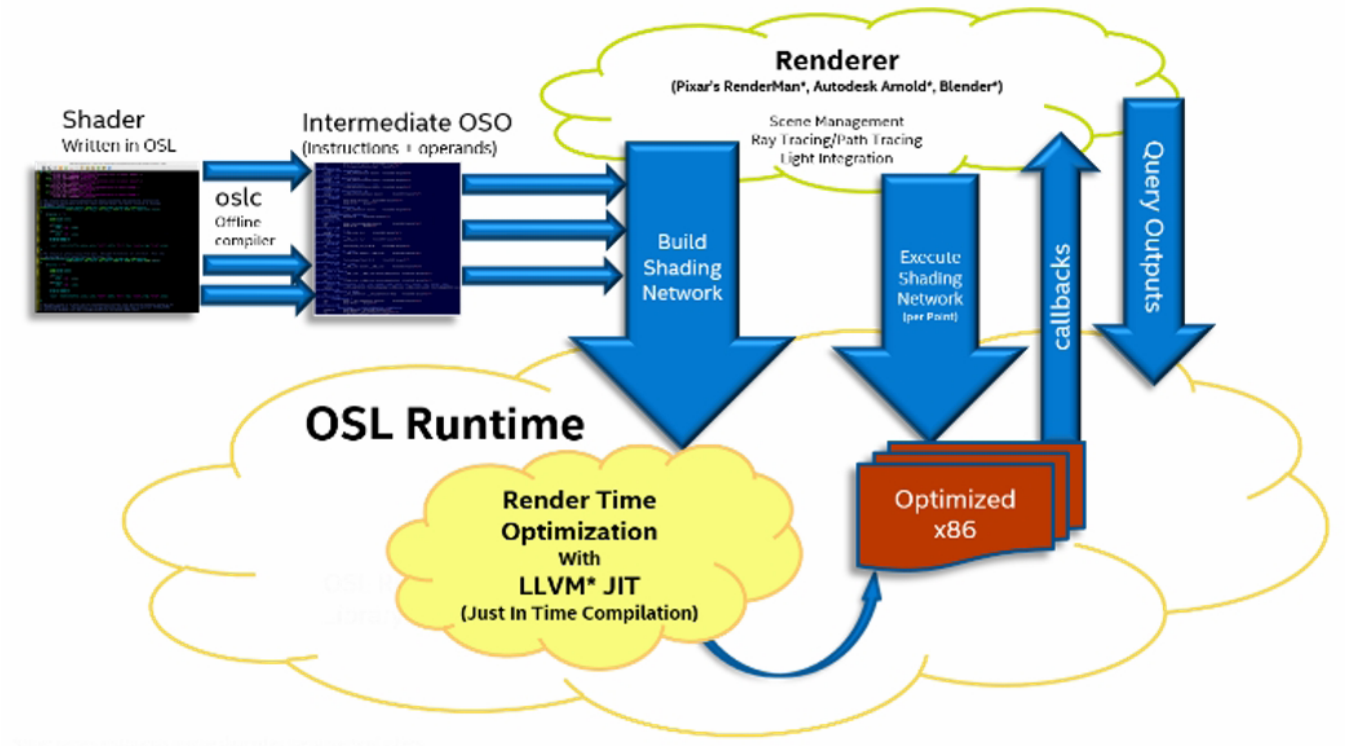
2 **A shader written in OSL (`marble.osl`[7])**

Benefit of OSL Shaders Over C++ Shaders

Shaders written in OSL are compiled by the OSLC (OSL compiler) into intermediate oso files that contain a mix of operands and instructions representing shader operations. Multiple shaders are compiled to build a large shading network. The OSL runtime employs LLVM to generate an intermediate representation (IR) of the shader, optimize it, and finally produce optimized x86 code, as demonstrated in **Figure 3**. Because of render time optimization, production scenes have benefitted from an orders-of-magnitude reduction in the number of operations, symbols, and empty shader instances. Scenes have demonstrated a:

- **99% reduction in operations** (from 280 million to 2.68 million operations)
- **98.8% reduction in symbols** (from 161 million to 1.9 million symbols)
- **63% optimization** by eliminating empty shader instances[8]

Because of its ability to leverage LLVM and JIT for render-time optimization, OSL can outperform precompiled C++ shaders.



3 **OSL framework from OSL shaders to optimized x86 code**

Introducing SIMD OSL

Even with all its advantages over C++ shaders, OSL in its original form lacks opportunities to leverage newer Intel SIMD instructions. Its block vectorization from using Intel® Streaming SIMD Extensions (Intel® SSE) uses only four lanes and offers limited support for complex noise, math, string, and texturing functions, among others. SIMD OSL uses the SPMD model to create a batched interface (process multiple points in the shading network) to the renderer. Features of SIMD OSL include:

- **Retaining OSL language specifications:** SIMD OSL does not change the way users interface with the OSL library (i.e., the original OSL shaders remain unchanged).
- **A new batched interface** enables the renderer to process batches of points from the shading network. Even so, it retains the original single-point interface, where a single point from a shading network is processed by the renderer.
- **Generating SIMD code via a wide backend:** SIMD OSL uses LLVM* vector data types `<16 * float>` for datatypes in vectorized intermediate representation (IR).
- **A new wide library:** Through its rearchitecture, SIMD OSL provides a wide interface to OSL functions (function families such as math, noise, etc.) using OpenMP-explicit vectorization.
- **Creating a comprehensive test framework** to test OSL library functions over combinations of uniform, varying, and constant operands.

Architecture of SIMD OSL

The rearchitecture of SIMD OSL introduces three major structural changes in existing single-point OSL

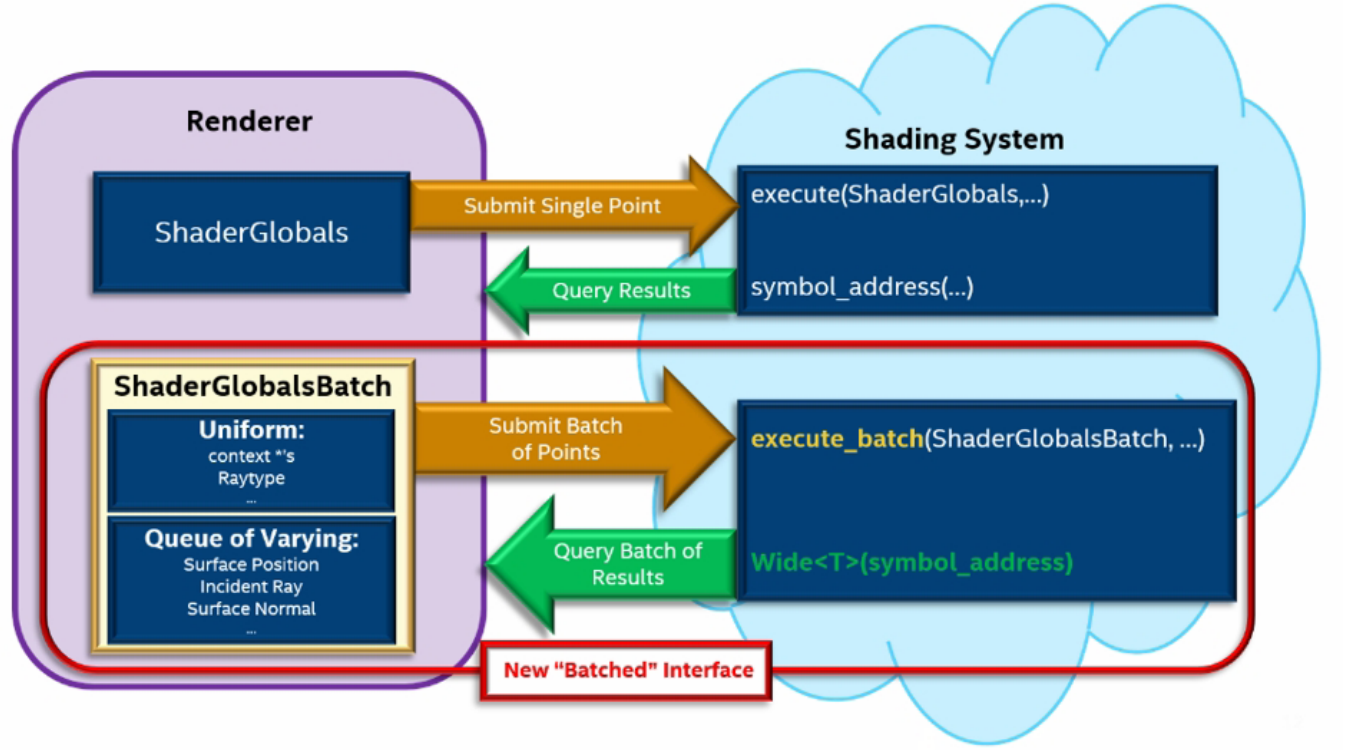
- **Providing** a batched interface (**Figure 4**)
- **Adding** wide accessors to represent and access varying data types
- **Handling** divergence among values across different SIMD lanes

Batched Interface in SIMD OSL

One of the changes in SIMD OSL is storing shader globals differently in the batched subsystem, depending on whether they are uniform or varying. The ability to use batches of points means the renderer can submit sets of points to the shading system, and the shading system can, in turn, query a set of results from the queue of varying globals in the renderer. For a renderer to be able to use SIMD OSL, it's important that it support the new wide interfaces while also accommodating for wide callbacks.

Datatypes in SIMD OSL

All variables are considered uniform unless they can be proven to be varying. A varying variable is one whose dependence can be traced to shader globals, which are known to be varying. For instance, shade globals like surface position, incident ray, and surface normal are always varying.



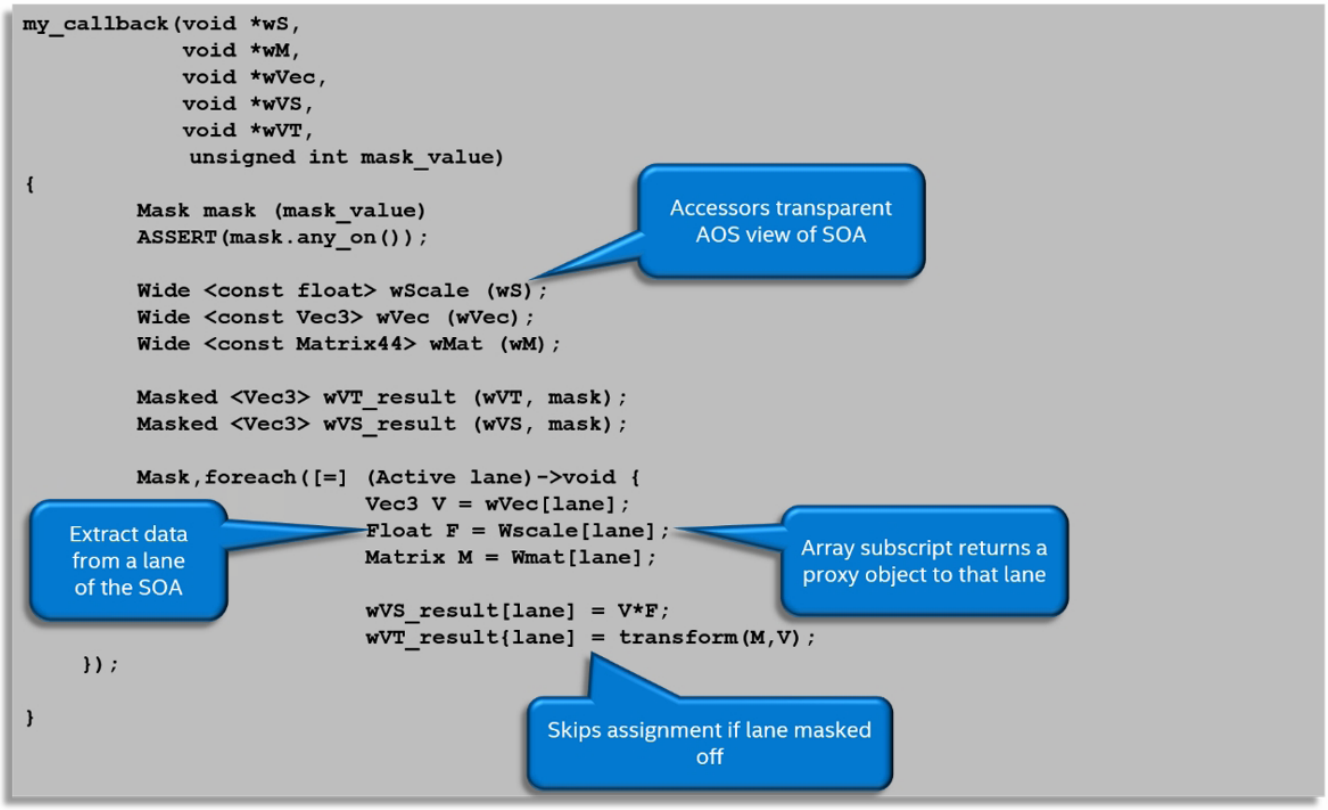
4 **Batched interface in SIMD OSL**

OSL contains a few aggregate datatypes—such as vector, color, point, and matrix—that are tuples. When used in arrays, data is stored in an array of structures (AOS) format. The overhead of looping through AOS values for storing into vector registers hinders vectorization. The Intel SIMD data layout template (SDLT)[9] uses containers with SIMD-friendly internal memory layout. Traditionally, datatypes of this form are represented as arrays of values in memory. SDLT containers provider accessor objects to import and export primitive datatypes between underlying memory layout and the original representation of the object.[9] Inspired by the SDLT library, the SIMD OSL Library provides wide accessors to support varying datatypes and callbacks through the renderer. Wide accessors resemble arrays of the datatype across each SIMD lane (**Figure 5**) and abstract the underlying structure of arrays (SOA) layout. Under the hood, masked accessors will skip inactive data lanes via a mask.

Masking Algorithm in SIMD OSL to Track Nested Control Flows

SIMD OSL uses a masking strategy to keep track of lanes that diverge at an if-condition. A mask tracks points that will execute on either path. Both code paths are then executed with the mask activated for each appropriate lane in each branch. However, this technique becomes complicated with nested control

flows, because performance would be bottlenecked by tracking points and their masks across various lanes along different code paths. To overcome this and still track the right points executing on the right control flow path, SIMD OSL uses a stack to keep track of masks at each conditional statement (**Figure 6**).

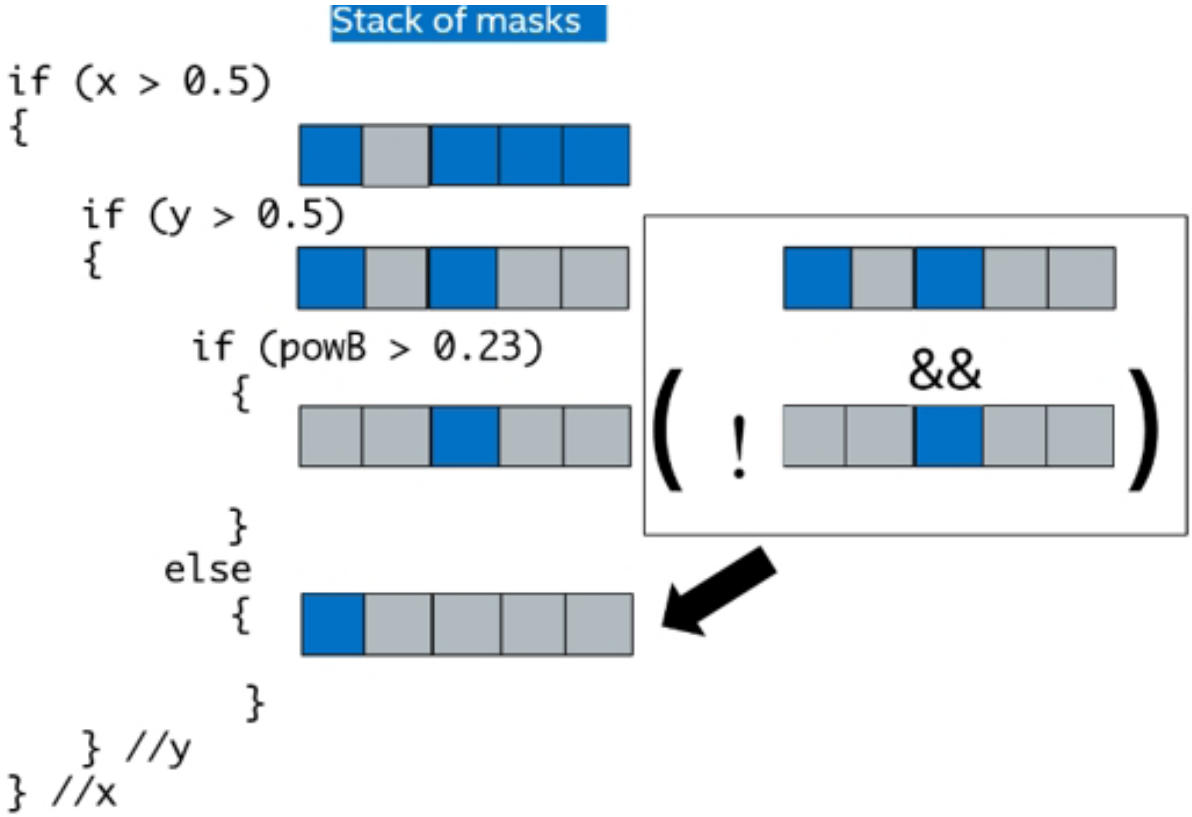


5 **Accessing varying data in SIMD OSL**

SIMD OSL's LLVM Backend

SIMD OSL uses LLVM to JIT target-specific code. For precompiled library functions, SIMD OSL generates different shared libraries for each supported platform—AVX, AVX2, and AVX-512. The appropriate library is loaded at run time to link addresses of each OSL precompiled library function with the JIT code.

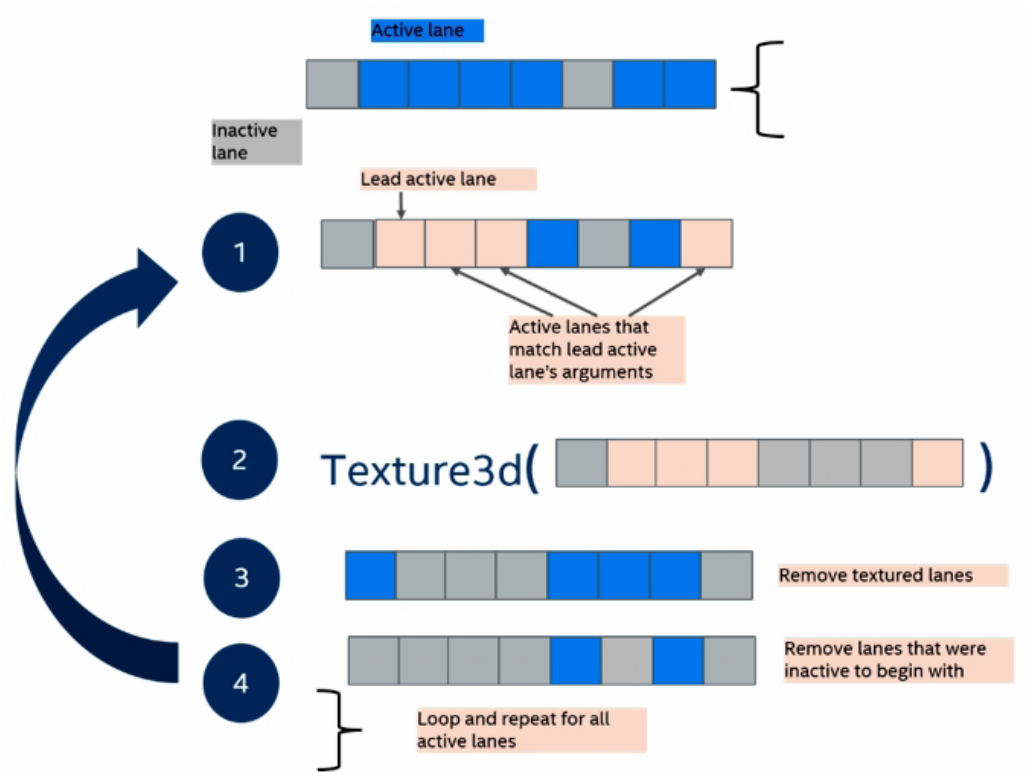
OSL, in its original form, contains an LLVM backend to support all families of functions. Intel rearchitected this backend to support our wide datatypes and masking controls when dealing with varying operands and control flows. In OSL's `andor` function, we use a four-step process to add SIMD support:



6 **Stack of masks to track divergence in control flow in SIMD OSL**

1. **Check** if the operand and result are uniform
2. **Load** operand values while accommodating for their type (uniform or varying)
3. **Emit** IR to either perform the operation or to call the appropriate precompiled library function
4. **Widen** result prior to storage if the result is varying

Because the `andor` function is simple, with only one operand that's required to be uniform, its support in the SIMD LLVM backend is uncomplicated. However, functions such as `texture3d()`, which contain multiple operands, require more complex LLVM backend support. The `texture3d()` function performs a 3D lookup of a volume texture, indexed by 3D coordinate `p`.[6] When we call `texture3d()`, the function expects a set of options, some of which are varying (`blur`, `width`, and the texture coordinate `p`), while some are expected to only be uniform (e.g., `wrap`). We first scan for lanes with the same argument settings so that they can execute together. The remaining lanes that don't match are turned off. The control flow is described in **Figure 7**.



7 **LLVM lane masking and filtering in `texture3d()` in SIMD OSL**

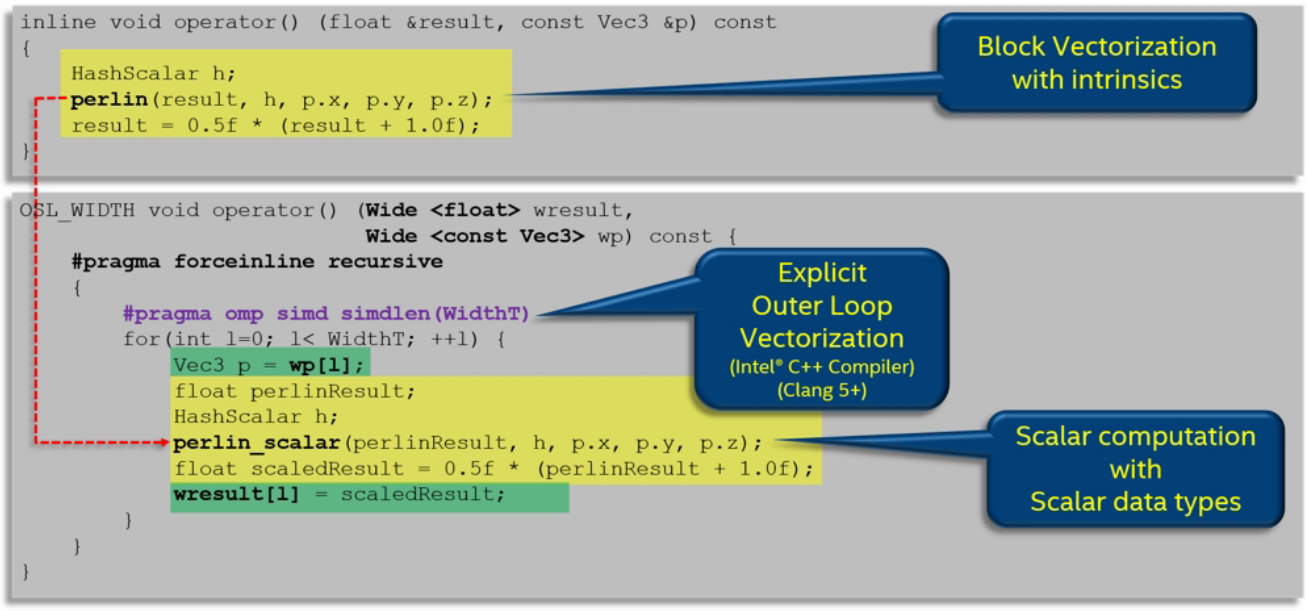
LLVM in SIMD OSL Loops and Control Flow

In OSL library functions, we can detect active lanes and implement a body of function calls in different ways, depending on lane utilization. For instance, in the Perlin* noise function (described below in greater detail), we use the default block vectorized Perlin noise implementation when the number of active lanes is less than four. We can also process each lane individually and vectorize it. In summary, leveraging LLVM for the SIMD OSL backend gives us the ability to change directions and vary the scope of vectorization, both inside and across lanes.

Perlin* Noise in SIMD OSL

The original version of Perlin noise in non-SIMD OSL is optimized to perform block vectorization within the algorithm using Intel SSE intrinsics (**Figure 8**). To enable outer loop vectorization while retaining performance of the original Perlin noise, we eliminate SSE intrinsics and revert to the original C++ version of the algorithm by creating a `perlin_scalar` helper. We then leverage the wide accessors to import and export the data type out of the underlying SOA data layout. The outer loop vectorization is implemented by OpenMP #pragma and specifying the SIMD width. Inside the loop, we export the data for the current lane, perform scalar computation via `perlin_scalar`, and then import results for the lane. Note that the

actual `perlin_scalar` computation is oblivious to our data layout and our outer SIMD loop. Once this is all inlined, the compiler can produce ideal code for multiple target ISAs (SSE2, AVX, AVX2, AVX-512, etc.). To ensure proper inlining on the **Intel® C++ Compiler**, we judiciously use `#pragma forceinline recursive`.



8 **Enabling SIMD in Perlin noise in OSL**

Performance of SIMD OSL on Intel® Xeon® Processors

We evaluate performance of OSL benchmarks and individual OSL shaders on two Intel® Xeon® processors:

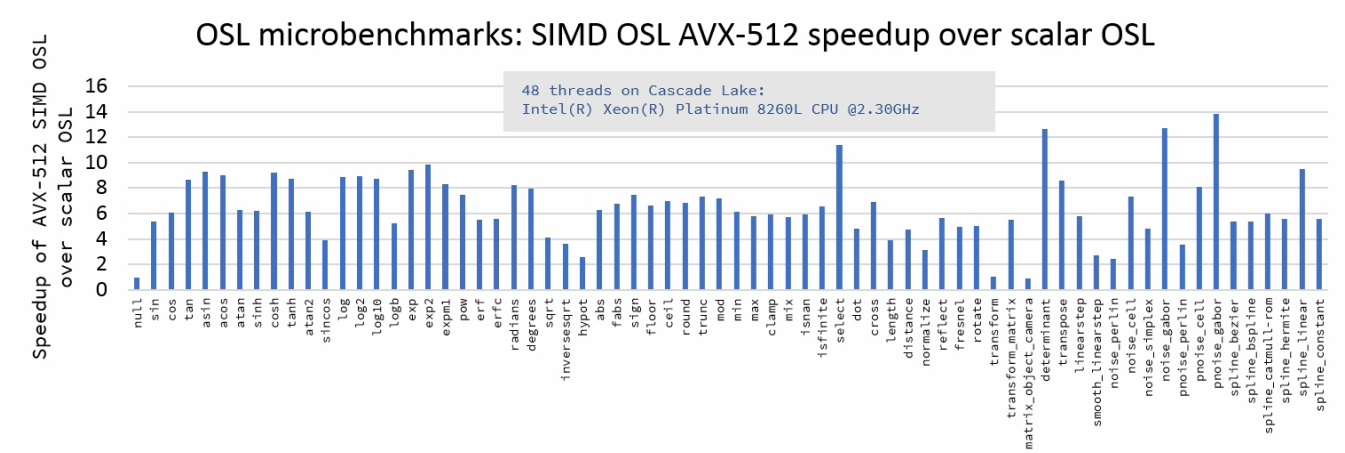
- A two-socket, 40-core Intel Xeon Gold 6248 processor @2.50GHz
- A two-socket, 48-core Intel Xeon Platinum 8260L processor @2.30GHz with hyperthreading turned off

OSL is run via testshade, a test harness that exercises shaders and shader groups. Testshade can also be viewed as a substitute for a shading module in a renderer. Testshade can be executed as single- or multi-threaded. We showcase the superior performance of SIMD OSL via:

- **A suite of microbenchmarks** comprising important OSL functions
- **A set of individual shaders** that represent different textures and patterns

Performance of SIMD OSL Microbenchmarks on Intel® Xeon® Platinum 8260L Processor

The OSL microbenchmark suite includes individual OSL functions from OSL function families—string, noise, math, trigonometry, logical operations, binary operations, spline, and others. **Figure 9** shows the speedup of AVX-512 SIMD OSL over scalar OSL from trials using 48 threads with batch size of 16.

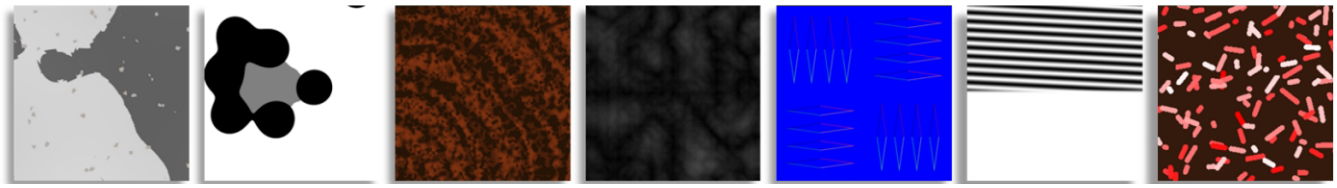


9 **SIMD OSL microbenchmarks on an Intel Xeon Platinum 8260L processor @2.30GHz**

The speedup across the 67 functions in the microbenchmark averages 7x, with a maximum speedup of 13.8x (Gabor noise).

Speedup of SIMD OSL over Single-Point Scalar OSL on Individual Shaders on Intel® Xeon® Gold 6248 Processor

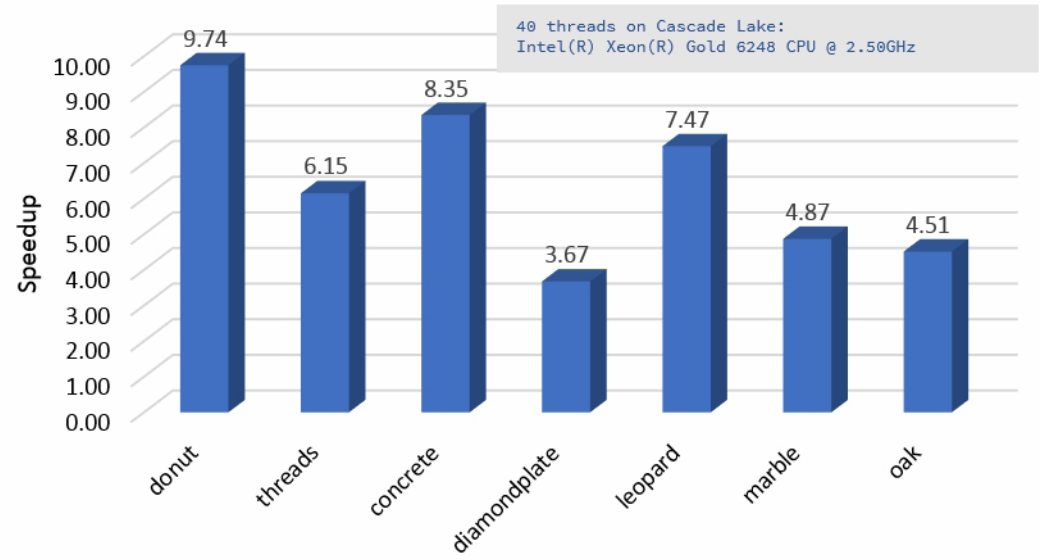
We evaluated a set of open source shaders using AVX-512 SIMD OSL, AVX2 SIMD OSL, and scalar OSL on an Intel Xeon Gold 6248 processor @2.5GHz. Each shader—marble[7], concrete[10], diamond plate[11], donut[12], leopard[13], oak[14], threads[15]—represents a distinct texture, as shown in **Figure 10**. The shaders differ in their complexity and the types and quantity of OSL functions they employ. For instance, the thread, marble, and oak shaders have a relatively simple control flow with only one or no branches, a single input, and a single output. On the other hand, shaders like concrete and leopard have a more complicated and divergent control flow.



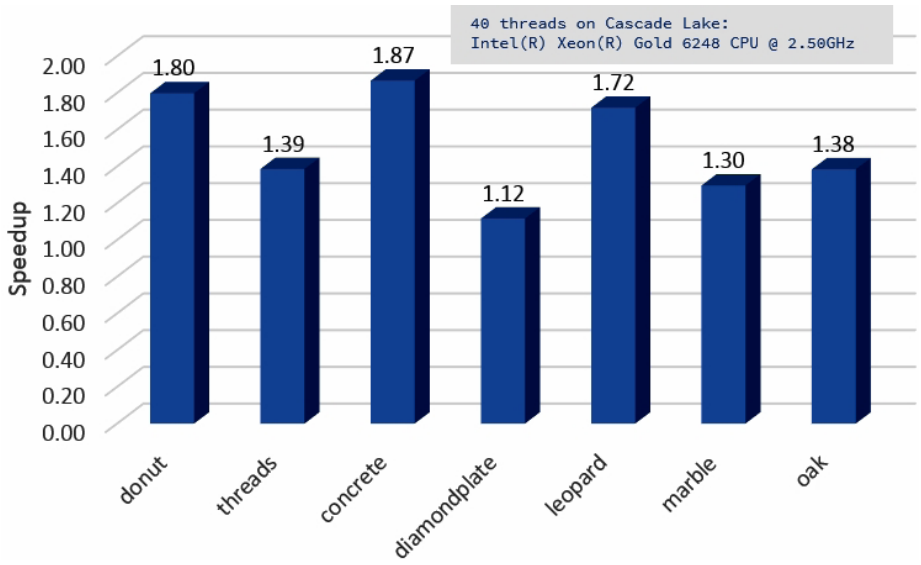
10 **OSL shaders. From left to right: concrete, leopard, oak, marble, diamondplate, threads, and donut**

First, we evaluate the speedup of the OSL shaders using SIMD OSL and compare performance against scalar single-point OSL (**Figure 11**). Note the concrete shader with its noise function calls enjoys a performance benefit of 9.7x. All shaders show a speedup between 3.7x to 8.4x.

We next evaluate the speedup of AVX-512 SIMD OSL over AVX2 SIMD OSL (**Figure 12**). All the shaders showed a benefit using the wider batch size that AVX-512 provides.



11 **Speedup of SIMD OSL over scalar OSL on an individual OSL shaders on Intel Xeon Gold 6248 processor @2.50GHz**



12 **Speedup of AVX-512 SIMD OSL over AVX2 SIMD OSL on individual OSL shaders on an Intel Xeon Gold 6248 processor @2.50GHz**

Turbo-Charged Open Shading Language

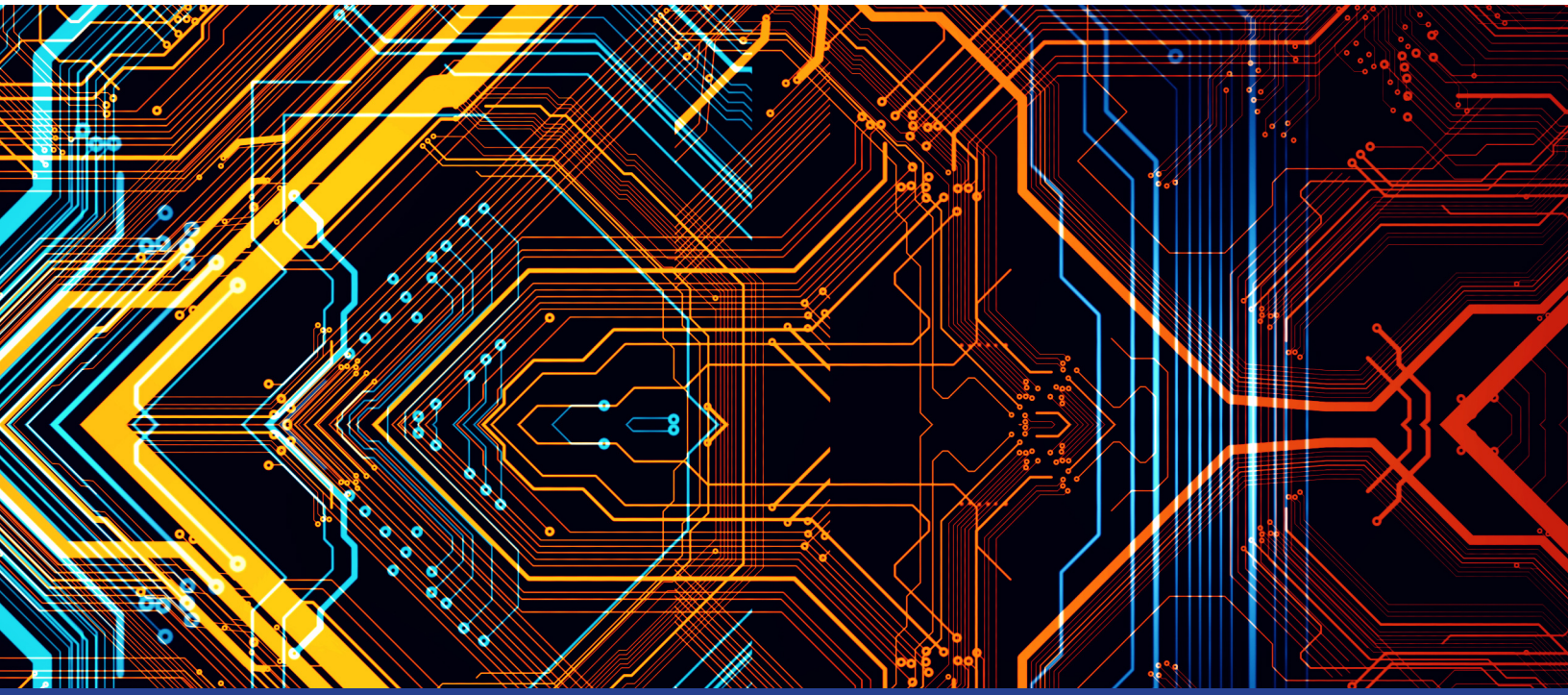
Intel has been leading the rearchitecture of SIMD OSL since 2016. This rearchitecture can be broadly summarized in two steps:

- **Introducing vector LLVM IR generation** (for JIT) during render-time optimization
- **Adding a batched interface** to the default single-point interface in OSL

SIMD OSL produced considerable benefit in physically-based renderers such as Pixar's RenderMan. The recently released RenderMan 22.5 with SIMD OSL has seen up to 2x faster full renders and a 15% average speedup using Intel Xeon Scalable processors with Intel AVX-512.[3]

References

1. **Open Shading Language Sci Tech Award in 2017**
2. **Open Shading Language Repository**
3. **Pixar Animation Studios Releases RenderMan 22.5**
4. **RenderMan: What's New**
5. **FMX 2019**
6. **Open Shading Language Specification**
7. **Marble Shader**
8. **OSL Talk at SIGGRAPH 2018**
9. **Intel® SIMD Data Layout Templates (SDLT)**
10. **Concrete.osl**
11. **DiamondPlate.osl**
12. **TheDonutShader.osl**
13. **Leopard.osl**
14. **Oak.osl**
15. **Threads.osl**



THE PERFORMANCE OPTIMISATION AND PRODUCTIVITY (POP) PROJECT

Pursuing the Never-Ending Quest for Performance

Mike Croucher, Developer Advocate, Numerical Algorithms Group (NAG)

For a long time now, the route to increased performance has been via parallelization. Vectorization, threads, MPI*, OpenMP*, GPUs, FPGAs, and dozens more hardware and software technologies promise to give you the performance you and your users crave. So you choose a set of technologies, embark on your code optimisation journey, and realize some fantastic speedups that your users eagerly consume. The success stories roll in and you sit back, content that the community is now using your product to solve bigger and more advanced problems than anyone ever considered feasible. All is going well.

But the quest for improved performance is never over, and soon your users want you to perform the speedup trick once again. The models they're building are bigger and more complex than ever. And the hardware they're running them on has new vectorization tricks—and much higher core counts—than you ever considered before. Your code base is huge, your budget limited, and all the low-hanging fruit has been picked and devoured.

Where do you start applying your development efforts?

The POP Project

The **Performance Optimisation and Productivity (POP) project** is a European Union-funded, international group of partners working to improve parallel software via several complementary routes including:

- **Developing a general methodology** that can be used to understand parallel performance
- **Developing open source tools** that can be used to apply the POP methodology
- **Creating a set of detailed case studies** where POP experts demonstrate these developments by auditing and refactoring the code of academic and industrial clients (available for free for clients within the EU).

The POP methodology can be applied to a range of parallelization schemes and programming languages. OpenMP and MPI in Fortran*, C, and C++ are the most popular, but POP has also worked on applications written in MATLAB*, Python*, and Perl*, among others.

The POP Methodology

Traditionally, there are several things we can try to gather intelligence about our application, such as scaling experiments, profiling, and tracing using products like **Intel® VTune™ Amplifier** or the open-source tools developed by some POP partners. These can result in a huge amount of data to sift through, containing everything from instruction counters to cache misses. It can be difficult to move from this sea of information to the kind of insights that would really help a code developer determine the most appropriate direction to follow to improve the code.

The POP methodology distills this sea of data into a **small hierarchy of metrics** that measure the relative impact of the different factors inherent in parallelization. Each metric is a measure of efficiency between 0 and 1, where higher numbers are better. As a rule of thumb, POP considers anything below 0.8 as worthy of further attention.

A case study can help us understand these metrics.

The POP Metrics and zCFD*

One of the POP partners, **The Numerical Algorithms Group (NAG)**, recently worked on the commercial computational fluid dynamics solver **zCFD*, developed by Zenotech**. By generating the POP metrics from Intel VTune Amplifier data and collaborating with the original developers, NAG helped improve the runtime of one particular simulation by 3x.

The first step in the audit was to limit the collection of Intel VTune Amplifier data to only the region of interest (RoI). zCFD uses a Python package (**zCFD-driver***) that calls computational kernels written in C++. As such, the team used the **NERSC Python VTune Instrumentation and Tracing Technology (ITT) API bindings** to disable tracing outside the RoI.

Once the Intel VTune Amplifier data was collected for simulation runs on varying numbers of cores, the first set of POP metrics could be computed (**Table 1**). (How to compute the POP metrics from Intel VTune Amplifier data is outside the scope of this article. For details, see the **POP webinar on this case study**. An alternative method is described in the article **Automatic Calculation of POP Metrics Using Scalasca.**)

Table 1. POP metrics

Threads	1	2	4	6	8	10	12
Global Efficiency	0.97	0.71	0.66	0.52	0.55	0.49	0.33
Parallel Efficiency	0.97	0.80	0.77	0.64	0.67	0.60	0.50
Computational Efficiency	1.00	0.89	0.85	0.82	0.82	0.82	0.66

The headline figure is global efficiency, which is the product of the parallel and computational efficiencies. Parallel efficiency measures the effect that parallelizing the code has on the runtime. This includes the impact of factors such as:

- **How well-balanced** the computational load is between threads
- **How much time** is lost to parallel overheads

It's calculated as the ratio between the average amount of time that threads spend in useful computation and the total runtime of the application.

Computational efficiency describes how well the computational load of the application scales with the number of threads. It's the ratio between the total time across all threads that the code spends in useful computation and the time the serial code spends in useful computation.

We observe that there's a general decline in global efficiency as the number of threads increases. This is largely driven by a corresponding decline in parallel efficiency. The computational efficiency doesn't decline as much, except on 12 threads.

Taken together, these efficiencies suggest that the prime opportunity for improvement lies in the way work is divided among threads rather than the computations each thread performs. For example, on 10 threads, the computational efficiency of 0.82 denotes that there's the potential to improve runtime by 18% if issues associated with computation are addressed—compared with a potential 44% improvement from addressing parallelization issues that the parallel efficiency of 0.56 suggests. With that said, there's something very strange going on with computational efficiency at 12 cores.

Parallel Efficiency

Now that we understand that focusing on parallel efficiency should give us the most gains, we can dive deeper to try to understand why it's so poor. A straightforward metric we can obtain from Intel VTune Amplifier is the percentage of runtime spent in serial sections of code (**Table 2**).

Table 2. Runtime in serial code

Threads	1	2	4	6	8	10	12
Percentage of Runtime in Serial	—	88.6	84.6	75.0	74.2	70.1	66.6

By the time we reach 12 cores, 33% of our runtime is spent in serial code sections. Further investigation determines there was a region the developers had attempted to parallelize, but that was actually still running sequentially. Some refactoring corrected this.

Load balance efficiency (**Table 3**) shows that work is spread unevenly across threads.

Table 3. Load balance efficiency

Threads	1	2	4	6	8	10	12
Load Balance Efficiency	1.00	0.88	0.89	0.85	0.89	0.86	0.85

Further investigation shows that the main load imbalance occurred in a region of code that called the `pow()` function. This was hitting a **slow code path**. Because both the base and the exponent were close to 1, `pow()` was computing the result to high accuracy, which took a lot of time. But this level of accuracy was not needed by the computation. This was resolved by scaling the base, raising it to the power, and then undoing the scaling[1]:

```
double pow(double a, double b) {
    a *= 5.0;
    tmp = pow(a,b);
    return tmp/pow(5.0,b);
}
```

The two calls to `pow()` can be computed at the same time using vectorization, so this change only incurred the cost of a single extra divide.

Computational Efficiency

Although the metrics showed us that computational efficiency isn't as important as parallel efficiency for this particular problem, there's something very strange going on when we move from 10 to 12 cores that warrants a closer look. We might hope it's something straightforward that we can easily fix. Happily, this is the case.

There are three submetrics that make up computational efficiency (**Table 4**):

1. Instructions per cycle (IPC) efficiency
2. Instructions efficiency
3. CPU frequency efficiency

Instruction efficiency is the ratio of the total number of useful instructions for a reference case (e.g., one processor) compared to values when increasing the numbers of processes. A decrease in instruction efficiency corresponds to an increase in the total number of instructions required to solve a computational problem.

IPC efficiency compares IPC to the reference, where lower values indicate that the rate of computation has slowed. Typical causes for this include decreasing cache hit rate and exhaustion of memory bandwidth, which can leave processes stalled and waiting for data.

CPU frequency efficiency looks at how clock speed changes as the number of threads increases.

Table 4. Submetrics that make up computational efficiency

Threads	1	2	4	6	8	10	12
IPC Efficiency	1.00	0.94	0.93	0.92	0.91	0.90	0.91
Instructions Efficiency	1.00	1.00	1.00	1.00	1.00	1.00	1.00
CPU Frequency Efficiency	1.00	0.94	0.91	0.89	0.90	0.91	0.72

There's nothing much of interest going on with the IPC and instructions efficiencies, but the CPU frequency drops sharply going from 10 to 12 cores.

Zenotech determined that the CPU frequency governor was set to on-demand by default on the machine used for the audit, and that this was responsible for the drop in operating frequency. Adding `--cpu-freq=performance` to the Slurm* commands resolved the issue by instructing the CPU to run at its base frequency even when fully populated with threads.

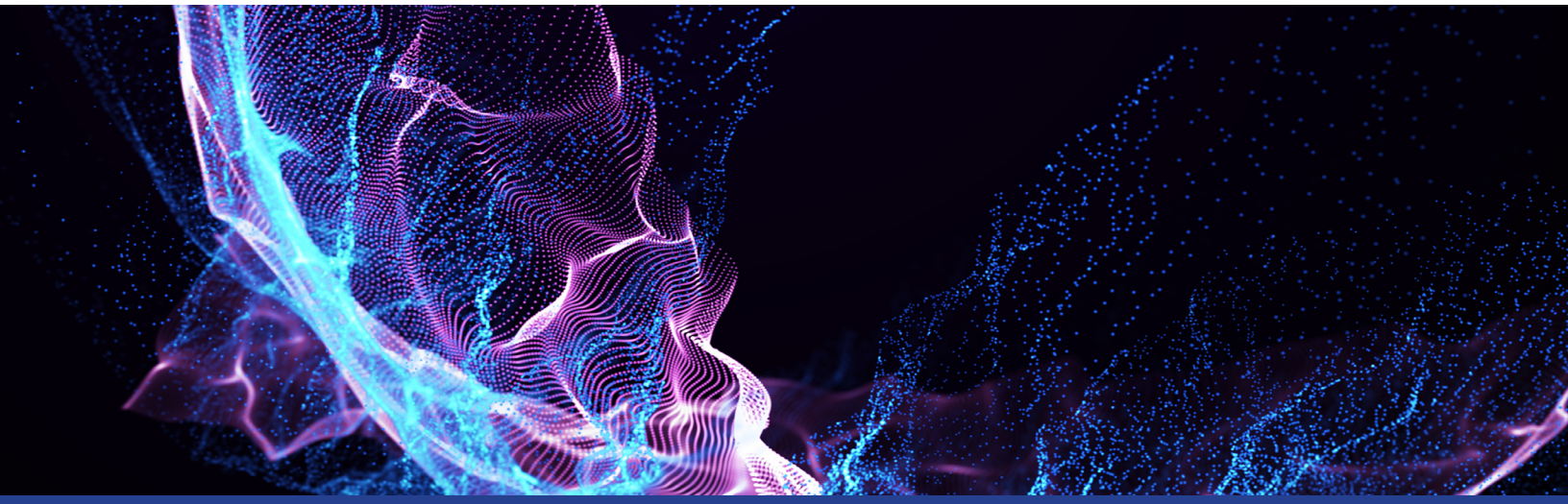
Results

Guided by these metrics, the developers of zCFD made the changes to the code and compute environment described above (along with a few more that we don't have the space to describe here). Recalculating the metrics on the new code resulted in the efficiencies shown in **Table 5.**

Table 5. Efficiencies

Threads	1	2	6	12
Global Efficiency	1.00	0.89	0.73	0.56
Parallel Efficiency	1.00	0.98	0.89	0.76
Computational Efficiency	1.00	0.91	0.82	0.74

We see across-the-board improvements comparing **Table 5** to **Table 1**. And when Zenotech ran the new code on a much larger problem, they observed speedups of up to 3x compared to the original code. Even with this success, the metrics suggest there might be yet more room for improvement. The quest continues.



SEVEN WAYS HPC SOFTWARE DEVELOPERS CAN BENEFIT FROM INTEL® SOFTWARE INVESTMENTS

Taking Another Look at Intel and HPC Software

James Reinders, Editor Emeritus, The Parallel Universe

Intel has been a powerhouse in supporting HPC software, but much has changed over the years. Here's a quick look at changes from Intel's software teams that you may not have noticed as things evolve.

HPC is at the dawn of a new golden age of hardware variety that recalls the early days of vector supercomputers, systolic arrays, and hypercubes. But today, a huge difference is the enormous installed base of mission-critical scientific and engineering software. And now, success in HPC is about making hardware subservient to software needs.

Here's a list of seven key ways Intel's contributions matter for HPC software developers.

1 Community Support: Lifting Popular Community Codes to Higher Performance—and Letting Us Learn from It, Too

Intel established the **Intel® Parallel Computing Centers (IPCC)** and the **Intel® Modern Code** initiative to help HPC software harness new hardware. IPCCs have contributed changes to a wide variety of open source applications used in HPC. And the Modern Code initiative has many online resources for learning essential techniques (visit **the site** for interesting interviews, training, white papers, and more).

The IPCCs documented their learnings in a highly accessible two-book series known as ***High-Performance Parallelism Pearls*** (which I helped edit). These books detail techniques for modernizing code using parallelization, vectorization, and algorithm selection—all of which would help any developer targeting the second-generation **Intel® Xeon® Scalable processors** (previously code-named Cascade Lake) covered in this issue's **feature article**.

2 Deep Program Analysis Tools: Helping Experts Tune their Applications and Systems

Nothing is more valuable than knowing what's really going on when you run an application. Intel's tools can also help with forecasting maximum speedup, locating bottlenecks, and pointing out parallel programming errors (pinpointing potential data races and deadlock). The name VTune has become legendary among profiling tools. New innovations offer roofline analysis, application performance snapshots, storage performance snapshots, MPI communications analysis (**Intel® Trace Analyzer and Collector**), and profiling of compiled code mixed with Python* and Java*. An application performance snapshot is a great place to start, but you'll find yourself eager to learn more in your quest to improve performance. Start with **Intel® VTune™ Amplifier**, but don't miss **Intel® Inspector** and **Intel® Advisor**.

3 Highly-Tuned Libraries: Drop Them In and Run Faster

Nothing's easier than using a vendor-optimized library to make a program run faster. **Intel® Math Kernel Library (Intel® MKL)** has done this for BLAS, FFTs, and solvers for years. Today, the complete suite of **Intel® Performance Libraries** should be in every HPC developer's toolkit:

- **Intel® Math Kernel Library:** Accelerate math processing routines, increase application performance, and reduce development time.
- **Intel® MPI Library:** Deliver flexible, efficient, and scalable cluster messaging on Intel® architecture.
- **Intel® Threading Building Blocks:** Get advanced threading for fast, scalable parallel applications.
- **Intel® Integrated Performance Primitives:** Speed performance for imaging, vision, signal, security, and storage applications.
- **Intel® Data Analytics Acceleration Library:** Boost machine learning and data analytics performance with this easy-to-use library.

4 Optimizing Compilers: Compile with Them and Run Faster

Augment your development process with compilers from Intel to create applications that run faster and more efficiently. These tools produce optimized code that takes advantage of the ever-increasing core count and vector register width in Intel® processors. The compilers plug into popular development environments and are compatible with third-party compilers such as the Microsoft* Visual C++ compiler (for Windows*) and GNU* compiler (for Linux* and macOS*). Learn more **here**.

5 Software-Defined Visualization

I've been repeatedly reminded that most visualization work in HPC is done on CPUs. Intel has invested heavily to support high-performance scientific visualization. The *in situ* nature of such work (plus the sophisticated rendering tasks) lend it to CPU rendering because it minimizes data movement. Also, a CPU's rendering pipeline is not hardwired.

Learn more about Intel's software **here** and more about software-defined visualization in general at **http://sdvis.org/**.

6 High-Performance AI with Intel-Optimized Deep Learning Frameworks and Accelerated Python*

Intel has invested in optimizing **popular deep learning frameworks** (e.g., TensorFlow* and PyTorch*) to do high-performance training and inference. Learn more at the **Intel AI Developer Program**. Intel has also optimized Python (focusing on NumPy*, SciPy*, and scikit-learn*) to make scientific codes run faster without any code changes required. Learn about the many ways to install it **here**.

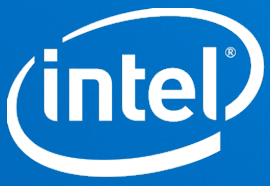
7 Cross-Architecture Tools: Intel® Distribution of OpenVINO™ Toolkit and a Vision to Help You Code Once and Run Faster Across a Variety of Hardware

The concept of code once and run everywhere isn't new, but all the solutions seem to incur severe performance penalties on at least some platforms. Intel started an open-source project known as **Intel® Distribution of OpenVINO™ toolkit** (which stands for open visual inferencing and neural network optimization). It spans CPUs, GPUs, FPGAs, and VPUs. For anyone looking to portably run neural networks across many architectures, this is well worth a look.

For now, that's a pretty specific crowd. However, at **Intel Architecture Day 2018**, the company laid out its vision for a **"oneAPI" project** with a goal of "no transistor left behind." It's a vision that looks to be a logical extension of their powerhouse of software support for HPC, which is already in the market and useful for us today.

Intel and HPC Software Development

In many ways, Intel and HPC have grown up together—and both have become very diverse and complex. Intel's investments in software continue to expand the ways it helps HPC software developers. I've listed seven concrete things to download, learn, and use to be more productive thanks to Intel. And—as the seventh of these tells us—Intel is nowhere near done contributing to the HPC community.



Software

THE PARALLEL UNIVERSE

Printed in USA 0719/SS Please Recycle