

THE PARALLEL UNIVERSE

Reduction Operations in Data Parallel C++

Winners Announced for the oneAPI Great
Cross-Architecture Challenge

Implementing the Fourier Correlation Algorithm
Using oneAPI

00001101
00001010
00001101
00001010
01001100
01101111

Issue
44
2021

01110001
01110011
01110101

Contents

Letter from the Editor	3
And the Oscar Goes to: The Intel® Embree Ray Tracing Library! by Henry A. Gabb, Senior Principal Engineer, Intel Corporation	
Winners Announced for the oneAPI Great Cross-Architecture Challenge	5
Reduction Operations in Data Parallel C++	9
Implementing and Tuning the Common Reduction Parallel Pattern	
Implementing the Fourier Correlation Algorithm Using oneAPI	17
Performing Complex Mathematical Operations with Just a Few Lines of DPC++ and oneMKL	
Using oneAPI to Speed Up the Finite-Difference Method	25
Migrating a CUDA-Based Stencil Computation to DPC++	
Advanced Ray Tracing APIs Proposed for the oneAPI Specification	33
“Write Once” High-Fidelity Ray Tracing across Multiple Vendors’ Accelerators	
oneTBB Flow Graph and the OpenVINO™ Inference Engine	35
Expressing Dependencies across Deep Learning Models in C++	
Optimization of Scan Operations Using Explicit Vectorization	46
Exploiting AVX-512 SIMD Instructions to Accelerate Prefix Sum Computations	

FEATURE

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.



And the Oscar Goes to: The Intel® Embree Ray Tracing Library!

I’m serious. The Intel® Embree Ray Tracing library, a key component of the [Intel® oneAPI Rendering Toolkit](#), received a [Scientific and Technical Achievement Award](#) from the Academy of Motion Picture Arts and Sciences (“The Oscars”):

“For the past decade, the Intel Embree Ray Tracing Library has provided a high-performance, industry-leading, CPU-based ray-geometry intersection framework through well-engineered open source code, supported by a comprehensive set of research publications. It has become an indispensable resource for motion picture production rendering.”

In this issue, Jim Jeffers (Intel Senior Director of Advanced Rendering and Visualization) gives a brief overview of the [Advanced Ray Tracing APIs Proposed for the oneAPI Specification](#). We also announce the winners of [The Great Cross-Architecture Challenge](#), a oneAPI coding contest sponsored by Intel in collaboration with the European Organization for Nuclear Research (CERN) and Argonne National Laboratory.

With the [oneAPI](#) industry initiative building on its momentum from the past year, this issue contains several articles on developing and tuning oneAPI code. Our feature discusses [Reduction Operations in Data Parallel C++](#) (DPC++). It’s the first in a two-part series on optimizing this common parallel pattern. I’ve been experimenting with DPC++ and oneMKL lately, so I decided to write an article about an algorithm that I use in my research: [Implementing the Fourier Correlation Algorithm Using oneAPI](#). Our friends at the

SENAI CIMATEC Supercomputing Center in Brazil were kind enough to contribute an article describing their experience using oneAPI tools to migrate a CUDA-based stencil code to DPC++: **Using oneAPI to Speed Up the Finite-Difference Method. oneTBB Flow Graph and the OpenVINO Inference Engine** shows you how to coordinate multiple machine learning models in the same pipeline using a lightweight C++ API.

Finally, we close this issue with some code modernization from the old school. **Optimization of Scan Operations Using Explicit Vectorization** illustrates the use of OpenMP directives and vector intrinsics to tune the computation of prefix sums.

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

Henry A. Gabb

April 2021



Winners Announced for the oneAPI Great Cross-Architecture Challenge

Intel announced the winners of the [oneAPI Great Cross-Architecture Challenge](#) in collaboration with the European Organization for Nuclear Research (CERN) and Argonne National Laboratory. The challenge attracted participants from 52 countries across five continents, showing the growing momentum of oneAPI's cross-architecture, multi-vendor, and open approach. The entrants used oneAPI and Data Parallel C++ (DPC++) to create a variety of applications in domains such as bioinformatics, cryptography, data analytics, education, financial services, genomics, healthcare, image processing, mathematics, molecular dynamics, particle physics, and ray tracing.

The oneAPI Great Cross-Architecture Challenge asked professional and student software developers to use oneAPI to create fast, efficient, and future-ready heterogeneous applications that take full advantage of various XPUs including CPUs, GPUs, FPGAs, and other accelerators. Using the free access to the [Intel® oneAPI Toolkits](#) and the [Intel® DevCloud](#), which provides the ability to test code and workloads across Intel® XPUs, participants had the option of porting an existing C/C++ or CUDA application using the Intel® DPC++ Compatibility Tool or creating an entirely new oneAPI application.

"This challenge really showcases the ease of use and freedom of choice that oneAPI's open, cross-architecture programming model delivers. The participants were able to either quickly port or develop from scratch applications with real-world impact across a range of disciplines. We are highly impressed with the innovative and creative submissions received from around the world, and the positive feedback and growing adoption for oneAPI."

Jeff McVeigh, vice president, Datacenter XPU Products and Solutions at Intel

"The participants in the Great Cross-Architecture Challenge demonstrated the potential of oneAPI. Through its use, they were able to write code for heterogeneous hardware architectures with a diverse range of applications. Opening the Intel® development environment leveled the playing field for this competition. People from across the world were able to access cutting-edge technology through this developer challenge. We look forward to welcoming the winners of the competition to CERN."

Maria Girone, Chief Technology Officer, CERN openlab

The winning student submissions receiving the internship award include:

- [Rafael Campos](#) of Portugal demonstrated oneAPI's fast and efficient development by adapting OpenCL* applications using modern constructs and minimal programming effort. The result is the boosting of performance and power efficiency of bioinformatic applications, specifically for epistasis detection.
- [Andrew Pastrello](#) of Australia showed the ease of porting CUDA code to DPC++ by optimizing a music production tool to synthesize audio from gravitational waveforms produced by binary black hole inspiral-merger-ringdown simulations.

Professional developers receiving the opportunity to take a special tour of CERN include:

- [Ricardo Nobre](#) of Portugal used the Intel DPC++ Compatibility Tool to seamlessly port a CUDA-based application, with more than 95% of their hand-tuned code automatically migrated. The application features collaborative utilization of CPU and GPU devices to find new associations between genotypes and phenotypes.
- [Zhen Ju](#) of China showcased the migration of a CUDA-based application and the benefits of an open programming model for all architectures. The ported application offered a more efficient and accurate solution to filter out redundant sequences in genetic data.
- [Eugenio Marinelli](#) of France leveraged oneAPI's complete set of cross-architecture libraries and tools to efficiently develop a new application for implementing scalable, heterogeneous parallel processing algorithms that can be used to quickly and accurately decode digital data stored in synthetic DNA.

Participants had access to additional free resources such as code samples, developer guides, webinars, and the [Intel® DevMesh](#) collaboration portal.

The contest offered more than \$40,000 in prizes as well as once-in-a-lifetime opportunities, like trips to CERN, a CERN openlab internship, and a chance to work on a project with Argonne National Laboratory. Five grand prize winners were selected by a [panel of six esteemed judges](#):

- Maria Girone, Chief Technology Officer, CERN
- Erik Lindahl, Professor of Physics, Stockholm University
- Simon McIntosh-Smith, Professor of High-Performance Computing, Bristol University
- Heidi Poxon, Distinguished Technologist, Hewlett Packard Enterprise
- Katherine Riley, Director of Science, Argonne National Laboratory
- Michael Wong, Distinguished Engineer, Codeplay Software

Entries were evaluated based on innovation, impact on humanity, use of cross-architecture computing, level of coding expertise, and quality of project explanation. The top five winners were awarded one of the following grand prizes:

- One of three trips to CERN for a special tour¹, or \$5,000 cash
- A summer CERN openlab internship (in person or virtual), or \$8,000 in cash
- Participate in a oneAPI-related project at Argonne National Laboratory (in person or virtual), or \$8,000 in cash

In addition to the grand prizes, 20 contestants received \$500 cash prizes for their quality submissions.

Since 2019, oneAPI ecosystem support has steadily grown. More than 60 leading research organizations, companies, and universities support the oneAPI initiative. Their success using Intel oneAPI Toolkits is noted in [oneAPI ecosystem support](#) and [reviews site](#). A new [oneAPI applications catalog](#) details more than 230 applications powered by oneAPI. The following resources are available to help developers build high-performance, cross-architecture applications using oneAPI and the Intel oneAPI Toolkits:

- Key Links: [Documentation](#), [oneAPI programming guide](#), and [code samples](#), and [free training](#).
- Support: For technical support using the Intel oneAPI Toolkits, developers can access the free [community forums](#). [Priority Support](#) with direct, private interactions with Intel engineers is included in [toolkit commercial packages](#).
- Training: [Free training](#) is available via webinars, deep-dive workshops, full learning paths, and more. For customers needing assistance accelerating HPC and AI solutions using oneAPI, a group of specially trained companies offer consulting through a new [Intel® oneAPI Technology Partner program](#).

¹ Contingent upon COVID-19 travel restrictions

Intel[®] DevCloud

A Development Sandbox for
Data Center to Edge Workloads

Develop, Test, and Run Your Workloads on a Cluster of the Latest Intel[®] Hardware and Software.

With integrated Intel[®] optimized frameworks, tools, and libraries, you'll have everything you need for your projects.

Try Out a Diverse Collection of Intel[®] Hardware.

Expand your skills and experiment with this state-of-the-art cluster that offers capabilities like natural language processing and time-series analysis—plus edge acceleration hardware.

Develop with Intel[®] Software Development Tools.

Jump-start your projects without having to download, configure, or install the latest compilers, performance libraries, and tools from Intel.

Use Popular AI Frameworks.

Accelerate your algorithms and applications with Intel[®] Optimized AI Frameworks that are ready for training and inference.

GET STARTED NOW >

Reduction Operations in Data Parallel C++

Implementing and Tuning the Common Reduction Parallel Pattern

Ramesh Peri, Senior Principal Engineer, Intel Corporation

An Introduction to the Reduction Operation

Reduction is a common operation in parallel programming that reduces the elements of an array into a single result. Let's say we want to add the elements of a large array into a single sum (**Figure 1**). Doing this operation in parallel requires the computation of partial sums that are sequentially combined, or reduced, to produce the final result. Reduction operators (e.g., summation, minimum, maximum, minimum location, and maximum location) are associative and are often commutative. There are many ways to implement a reduction, and its

performance depends on the underlying processor architecture. This article explores several ways to express reductions in Data Parallel C++ (DPC++) and discusses the performance implications of each.

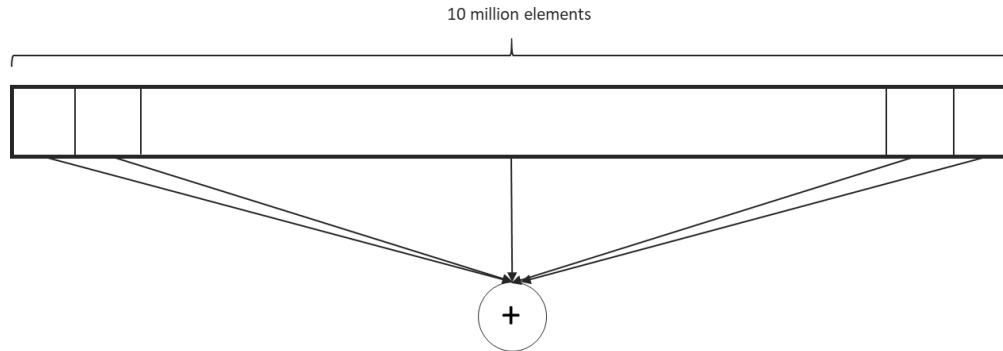


Figure 1 Pictorial representation of a summation reduction

Reduction in DPC++ Using Global Atomics

In the following implementation, each work-item in the DPC++ kernel is responsible for an element of the input array and atomically updates a global variable:

```
void reductionAtomics1(sycl::queue &q,
                     sycl::buffer<int> inbuf,
                     int &res,
                     int size) {
    const size_t data_size = inbuf.get_size() / sizeof(int);
    int num_work_items = data_size;
    sycl::buffer<int> sum_buf(&res, 1);
    q.submit([&](auto &h) {
        sycl::accessor buf_acc(inbuf, h, sycl::read_only);
        sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::noinit);

        h.parallel_for(num_work_items, [=](auto index) {
            size_t glob_id = index[0];
            auto v = sycl::ONEAPI::atomic_ref<int,
                sycl::ONEAPI::memory_order::relaxed,
                sycl::ONEAPI::memory_scope::device,
                sycl::access::address_space::global_space>(sum_acc[0]);
            v.fetch_add(buf_acc[glob_id]);
        });
    });
};
```

Depending on the number of threads created by the compiler (which in turn depends on the default work-group and sub-group sizes chosen by the compiler for the target device), the contention for accessing the single global variable, *sum_buf*, will be quite high. In general, the performance of such a solution will not be very good.

Reducing Contention on Global Atomics

One way to reduce the contention on the global variable update is to decrease the number of threads accessing this variable. This can be achieved by making each work-item process multiple elements of the array, perform a local reduction on a chunk of elements, and then perform the global atomic update (**Figure 2**).

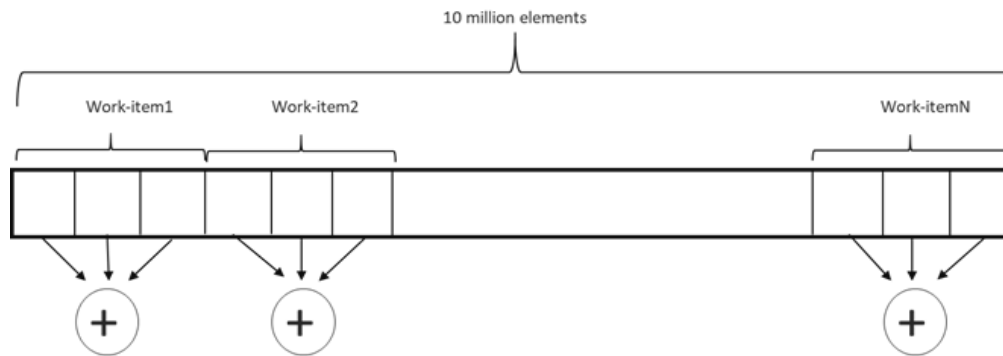


Figure 2 Each work-item computes a partial sum

This implementation is shown in the following code:

```
void reductionAtomics2(sycl::queue &q,
                      sycl::buffer<int> inbuf,
                      int &res) {
    const size_t data_size = inbuf.get_size() / sizeof(int);
    sycl::buffer<int> sum_buf(&res, 1);
    int num_work_items =
        q.get_device().get_info<sycl::info::device::max_compute_units>();
    int BATCH = (data_size + num_work_items - 1) / num_work_items;
    q.submit([&](auto &h) {
        sycl::accessor buf_acc(inbuf, h, sycl::read_only);
        sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::noinit);

        h.parallel_for(num_processing_elements, [=](auto index) {
            size_t glob_id = index[0];
            size_t start = glob_id * BATCH;
            size_t end = (glob_id + 1) * BATCH;
            if (end > N)
                end = N;
            int sum = 0;
            for (size_t i = start; i < end; i++)
                sum += buf_acc[i];
            auto v = sycl::ONEAPI::atomic_ref<int,
                sycl::ONEAPI::memory_order::relaxed,
                sycl::ONEAPI::memory_scope::device,
                sycl::access::address_space::global_space>(sum_acc[0]);
            v.fetch_add(sum);
        });
    });
}
```

This implementation is not very efficient because each work-item is accessing contiguous locations in memory, which causes the compiler to generate inefficient code. The DPC++ compiler treats each work-item like a vector lane, so work-items accessing contiguous locations of memory will be inefficient.

Efficient Access of Memory by Work-Items

The DPC++ compiler maps each work-item to a vector lane of the underlying processor, which allows the compiler to generate efficient code when the work-items access memory locations with a stride greater than the vector length of the processor. This access pattern is shown in **Figure 3**, where each of the labels $w_{i-1} \dots w_{i-n}$ represent each of the n work-items.

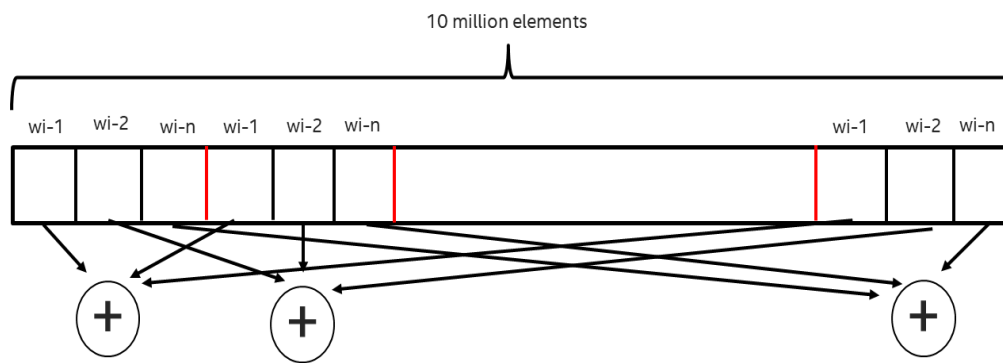


Figure 3 Each work-item computes a partial sum from non-contiguous memory locations

The kernel where each work-item operates on multiple, interleaved elements of the input vector is shown in the following code:

```
void reductionAtomics3(sycl::queue &q,
                      sycl::buffer<int> inbuf,
                      int &res) {
    const size_t data_size = inbuf.get_size() / sizeof(int);
    sycl::buffer<int> sum_buf(&res, 1);
    int num_work_items =
        q.get_device().get_info<sycl::info::device::max_compute_units>() *
        q.get_device().get_info<sycl::info::device::native_vector_width_int>();
    q.submit([&](auto &h) {
        sycl::accessor buf_acc(inbuf, h, sycl::read_only);
        sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::noinit);

        h.parallel_for(num_work_items, [=](auto index) {
            size_t glob_id = index[0];
            int sum = 0;
            for (size_t i = glob_id; i < data_size; i += num_work_items)
                sum += buf_acc[i];
            auto v = sycl::ONEAPI::atomic_ref<int,
                sycl::ONEAPI::memory_order::relaxed,
                sycl::ONEAPI::memory_scope::device,
                sycl::access::address_space::global_space>(sum_acc[0]);
            v.fetch_add(sum);
        });
    });
}
```

The choice of number of work-items is important. It is selected to be the product of the number of processing elements and the preferred vector width of the processing element. This number may be sufficient on some platforms, but perhaps not for others where the number of threads supported is much larger than the number of processing elements. Consequently, this number must be chosen carefully based on the target platform.

Tree Reduction

A tree reduction is a popular technique in which each of the work-items in a kernel apply the reduction operator to adjacent elements, producing intermediate results with multiple levels. This can be applied within a work-group, as shown in **Figure 4**, because thread scheduling and synchronization are highly efficient, with hardware support within a work-group.

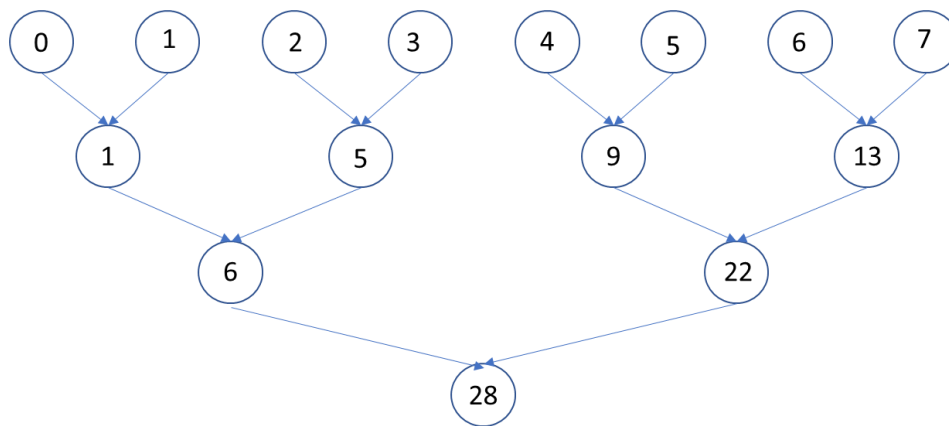


Figure 4 Tree reduction

The size of a work-group is fairly small (about 256 or 512) and is dependent on the hardware device, while the number of elements to be reduced is an order of magnitude larger than the work-group size. This means that the reduction operation performed by each work-group produces an intermediate result that needs to be further reduced. This can be handled in two ways:

1. By each work-group calling a global atomic add with its intermediate result, as shown in the following code:


```

void buf2finalReduction(sycl::queue &q,
                      sycl::buffer<int> inbuf,
                      int &res) {
    const size_t num_work_items = inbuf.get_size() / sizeof(int);
    int work_group_size =
        q.get_device().get_info<sycl::info::device::max_work_group_size>();
    sycl::buffer<int> sum_buf(&res, sizeof(int));
    q.submit([&](auto &h) {
        sycl::accessor buf_acc(inbuf, h, sycl::read_only);
        sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::noinit);
        sycl::accessor<int, 1, sycl::access::mode::read_write,
            sycl::access::target::local> scratch(work_group_size, h);

        h.parallel_for(sycl::nd_range<1>(num_work_items, work_group_size),
            [=](sycl::nd_item<1> item) {
                size_t global_id = item.get_global_id(0);
                int local_id = item.get_local_id(0);
                int group_id = item.get_group(0);
                int sum = 0;

                if (global_id < data_size)
                    scratch[local_id] = buf_acc[global_id];
                else
                    scratch[local_id] = 0;
                for (int i = work_group_size / 2; i > 0; i >>= 1)
                    item.barrier(sycl::access::fence_space::local_space);
                if (local_id < i)
                    scratch[local_id] += scratch[local_id + i];
            }

            if (local_id == 0) {
                auto v = sycl::ONEAPI::atomic_ref<int,
                    sycl::ONEAPI::memory_order::relaxed, sycl::ONEAPI::memory_scope::device,
                    sycl::access::address_space::global_space>(sum_acc[0]);
                v.fetch_add(scratch[0]);
            }
        });
    });
}

```

2. By calling the same reduction kernel once again on the list of intermediate values produced by the work-groups to produce another set of intermediate values. At some point, once the number of intermediate results is small enough, one can simply use the global atomic updates instead of calling the kernel to get the final result.

The kernel implementing this technique is shown below. Here, the output of the kernel is another set of values that are stored by each work-group in the output buffer.

```
void buf2bufReduction(sycl::queue &q,
                    sycl::buffer<int> inbuf,
                    sycl::buffer<int> outbuf) {
    const size_t data_size = inbuf.get_size() / sizeof(int);
    int num_work_items = data_size;
    int work_group_size =
        q.get_device().get_info<sycl::info::device::max_work_group_size>();
    q.submit([&](auto &h) {
        sycl::accessor inbuf_acc(inbuf, h, sycl::read_only);
        sycl::accessor outbuf_acc(outbuf, h, sycl::write_only, sycl::noinit);
        sycl::accessor<int, 1, sycl::access::mode::read_write,
            sycl::access::target::local> scratch(work_group_size, h);

        h.parallel_for(sycl::nd_range<1>(num_work_items, work_group_size),
            [=](sycl::nd_item<1> item) {
                size_t global_id = item.get_global_id(0);
                int local_id = item.get_local_id(0);
                int group_id = item.get_group(0);
                int sum = 0;

                if (global_id < data_size)
                    scratch[local_id] = buf_acc[global_id];
                else
                    scratch[local_id] = 0;
                for (int i = work_group_size / 2; i > 0; i >>= 1)
                    item.barrier(sycl::access::fence_space::local_space);
                    if (local_id < i)
                        scratch[local_id] += scratch[local_id + i];
                }
                if (local_id == 0)
                    outbuf_acc[group_id] = scratch[0];
            });
    });
}
```

The previous two kernels can be called as shown below to get the final result. Here, X is the size of the intermediate result where the final atomics-based reduction is called, and this value is chosen based on the efficiency of the global atomics-based reduction.

```
int work_group_size =
    q.get_device().get_info<sycl::info::device::max_work_group_size>();
int ibufsize = (size + work_group_size - 1) / work_group_size;
sycl::buffer<int> *prev = &buf;
while (ibufsize > X) {
    int *bufptr=(int *)malloc(sizeof(int) * ibufsize);
    sycl::buffer<int> *cur = new sycl::buffer<int>(bufptr, ibufsize);
    ibufsize = (ibufsize + work_group_size-1) / work_group_size;
    buf2bufReduction(q, *prev, *cur);
    prev = cur;
}
buf2finalReduction(q, *prev, res);
```

DPC++ Built-In Reduction Operator

DPC++ provides a summation reduction operator, which is used in the kernel below. In this case, the compiler chooses an implementation that is the most efficient for the underlying platform. It is usually recommended to use the built-in reduction operator.

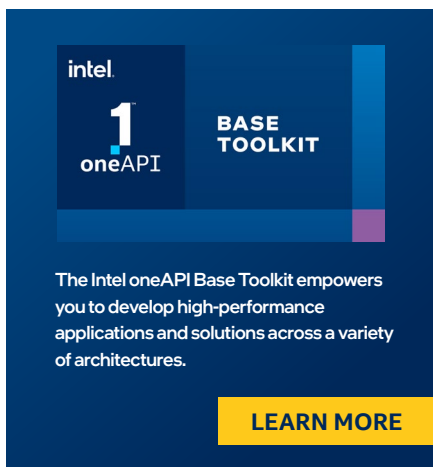
```
void builtinReduction(sycl::queue &q,  
                    sycl::buffer<int> inbuf,  
                    int &res) {  
    sycl::buffer<int> sum_buf(&res,1);  
    q.submit([&](auto &h) {  
        sycl::accessor buf_acc(inbuf, h, sycl::read_only);  
        sycl::accessor sum_acc(sum_buf, h, sycl::read_write);  
        auto sumr = sycl::ONEAPI::reduction(sum_acc, sycl::ONEAPI::plus<>());  
        h.parallel_for(sycl::nd_range<1>{data_size, 256}, sumr,  
                      [=](sycl::nd_item<1> item, auto &sumr_arg) {  
                        int glob_id = item.get_global_id(0);  
                        sumr_arg += buf_acc[glob_id];  
                    });  
    });  
}
```

Final Thoughts

Reduction is a common parallel programming pattern used in many applications. In this article, we explored some ways of implementing this operation in DPC++. The performance of these implementations can be quite different depending on the compiler and the target platform. My next article will explore the performance of these kernels on CPUs and GPUs.

Further Reading and Resources

- [oneAPI](#)
- [Intel® oneAPI Toolkits](#)
- [Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL](#)



intel
1
oneAPI

**BASE
TOOLKIT**


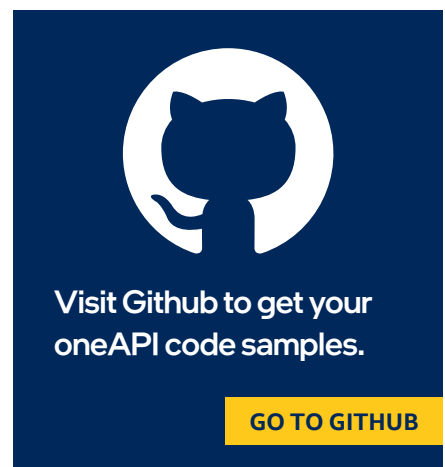
The Intel oneAPI Base Toolkit empowers you to develop high-performance applications and solutions across a variety of architectures.

LEARN MORE



Get the oneAPI GPU Optimization Guide for the best performance with everything from Parallelization to Kernels to memory.

LEARN MORE



Visit Github to get your oneAPI code samples.

GO TO GITHUB

Implementing the Fourier Correlation Algorithm Using oneAPI

Performing Complex Mathematical Operations with Just a Few Lines of DPC++ and oneMKL

Henry A Gabb, Senior Principal Engineer and Editor-in-Chief of The Parallel Universe, Intel Corporation

[Editor's note: This article was adapted from an example that I created for the [oneAPI GPU Optimization Guide](#). The complete source code and build and run instructions are available in the [oneAPI-samples repository](#). The guide and the repo are both excellent resources for oneAPI developers.]

Introduction to Cross-Correlation

Offloading individual oneMKL kernel functions to accelerators is straightforward, so let's look at a more complex mathematical operation requiring multiple kernel functions: cross-correlation. Cross-correlation

has many applications (e.g.: measuring the similarity of two 1D signals, finding the best translation to overlay similar images, volumetric medical image segmentation, etc.). Consider the following simple signals, represented as vectors of ones and zeros:

```
Signal 1: 0 0 0 0 0 1 1 0
Signal 2: 0 0 1 1 0 0 0 0
```

The signals are treated as circularly shifted versions of each other, so shifting the second signal three elements relative to the first signal will give the maximum correlation score of two:

```
Signal 1: 0 0 0 0 0 1 1 0
Signal 2:      0 0 1 1 0 0 0 0

Correlation: (1 * 1) + (1 * 1) = 2
```

Shifts of two or four elements give a correlation score of one. Any other shift gives a correlation score of zero. This is computed as follows:

$$corr_i = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} sig1_i \times sig2_{i+j}$$

where N is the number of elements in the signal vectors and i is the shift of $sig2$ relative to $sig1$.

Real signals contain more data (and noise), but the principle is the same whether you are aligning 1D signals, overlaying 2D images, or performing 3D volumetric image registration. The goal is to find the translation that maximizes correlation. However, the brute force summation shown above requires N multiplications and additions for every N shifts. In 1D, 2D, and 3D, the problem is $O(N^2)$, $O(N^3)$, and $O(N^4)$, respectively.

The Fourier correlation algorithm is a much more efficient way to perform this computation because it takes advantage of the $O(N \log N)$ complexity of the Fourier transform:

$$corr = \text{IDFT}(\text{DFT}(sig1) * \text{CONJG}(\text{DFT}(sig2)))$$

where DFT is the discrete Fourier transform, IDFT is the inverse DFT, and CONJG is the complex conjugate. The Fourier correlation algorithm can be composed using oneMKL, which contains optimized forward and backward transforms and complex conjugate multiplication functions. Therefore, the entire computation can be performed on the accelerator device.

Generating Some Test Data Using oneMKL

In many applications, only the final correlation result matters, so this is all that has to be transferred from the device back to the host. In this example, two artificial signals will be created on the device, transformed in place, and then correlated. The host will retrieve the final result and report the optimal translation and correlation score.

Conventional wisdom suggests that buffering would give the best performance because it provides explicit control over data movement between the host and the device. To test this hypothesis, let's generate two input signals:

```
// Create buffers for signal data. This will only be used on the device.
sycl::buffer<float> sig1_buf{N + 2};
sycl::buffer<float> sig2_buf{N + 2};
// Declare container to hold the correlation result (computed on the device,
// used on the host)
std::vector<float> corr(N + 2);
```

Random noise is often added to signals to prevent overfitting during neural network training, to add visual effects to images, or to improve the detectability of signals obtained from suboptimal detectors, etc. The buffers are initialized with random noise, using a basic random number generator in oneMKL:

```
// Initialize SYCL queue
sycl::queue Q(sycl::default_selector{});

// Open new scope to trigger update of correlation result
{
    sycl::buffer<float> corr_buf(corr);

    // Initialize the input signals with artificial data
    std::uint32_t seed = (unsigned)time(NULL); // Get RNG seed value
    oneapi::mkl::rng::mcg31m1 engine(Q, seed); // Initialize RNG engine
                                           // Set RNG distribution
    oneapi::mkl::rng::uniform<float, oneapi::mkl::rng::uniform_method::standard>
        rng_distribution(-0.00005, 0.00005);

    oneapi::mkl::rng::generate(rng_distribution, engine, N, sig1_buf); // Noise
    oneapi::mkl::rng::generate(rng_distribution, engine, N, sig2_buf);
```

Notice that a new scope is opened and a buffer, *corr_buf*, is declared for the correlation result. When this buffer goes out of scope, *corr* will be updated on the host.

An artificial signal is placed at opposite ends of each buffer, similar to the trivial example above:

```
Q.submit([&](sycl::handler &h) {
    sycl::accessor sig1_acc{sig1_buf, h, sycl::write_only};
    sycl::accessor sig2_acc{sig2_buf, h, sycl::write_only};
    h.single_task<>([=]() {
        sig1_acc[N - N / 4 - 1] = 1.0;
        sig1_acc[N - N / 4] = 1.0;
        sig1_acc[N - N / 4 + 1] = 1.0; // Signal
        sig2_acc[N / 4 - 1] = 1.0;
        sig2_acc[N / 4] = 1.0;
        sig2_acc[N / 4 + 1] = 1.0;
    });
}); // End signal initialization
```

Implementing the 1D Fourier Correlation Using Explicit Buffering

Now that the signals are ready, let's transform them using the DFT functions in oneMKL:

```
// Initialize FFT descriptor
oneapi::mkl::dft::descriptor<oneapi::mkl::dft::precision::SINGLE,
                             oneapi::mkl::dft::domain::REAL>
    transform_plan(N);
transform_plan.commit(Q);
// Perform forward transforms on real arrays
oneapi::mkl::dft::compute_forward(transform_plan, sig1_buf);
oneapi::mkl::dft::compute_forward(transform_plan, sig2_buf);
```

A single-precision, real-to-complex forward transform is committed to the SYCL queue, and then an in-place DFT is performed on the data in both buffers. The result of `DFT(sig1)` must now be multiplied by `CONJG(DFT(sig2))`. This could be done with a hand-coded Data Parallel C++ (DPC++) kernel:

```
Q.submit([&](sycl::handler &h)
{
    sycl::accessor sig1_acc{sig1_buf, h, sycl::read_only};
    sycl::accessor sig2_acc{sig2_buf, h, sycl::read_only};
    sycl::accessor corr_acc{corr_buf, h, sycl::write_only};

    h.parallel_for<>(sycl::range<1>{N/2}, [=](auto idx)
    {
        corr_acc[idx*2+0] = sig1_acc[idx*2+0] * sig2_acc[idx*2+0] +
                            sig1_acc[idx*2+1] * sig2_acc[idx*2+1];
        corr_acc[idx*2+1] = sig1_acc[idx*2+1] * sig2_acc[idx*2+0] -
                            sig1_acc[idx*2+0] * sig2_acc[idx*2+1];
    });
});
```

However, this basic implementation is unlikely to give optimal cross-architecture performance. Fortunately, oneMKL provides a convenience function, `oneapi::mkl::vm::mulbyconj`, that can be used for this step. The `mulbyconj` function expects `std::complex<float>` input, but the buffers were initialized as the float data type. Even though they contain complex data after the forward transform, the buffers will have to be recast:

```
// Compute: DFT(sig1) * CONJG(DFT(sig2))
auto sig1_buf_cplx =
    sig1_buf.template reinterpret<std::complex<float>, 1>(N + 2) / 2);
auto sig2_buf_cplx =
    sig2_buf.template reinterpret<std::complex<float>, 1>(N + 2) / 2);
auto corr_buf_cplx =
    corr_buf.template reinterpret<std::complex<float>, 1>(N + 2) / 2);
oneapi::mkl::vm::mulbyconj(Q, N / 2, sig1_buf_cplx, sig2_buf_cplx,
    corr_buf_cplx);
```

The IDFT step completes the computation:

```
// Perform backward transform on complex correlation array
oneapi::mkl::dft::compute_backward(transform_plan, corr_buf);
} // Buffer holding correlation result is now out of scope, forcing update of
// host container
```

When the scope that was opened at the start of this example is closed, the buffer holding the correlation result goes out of scope, forcing an update of the host container. This is the only data transfer between the host and the device.

The complete source code ([fcorr_1d_buffers.cpp](#)) is available in the oneAPI-samples repository.

Implementing the 1D Fourier Correlation Using USM

The Fourier correlation algorithm will now be reimplemented using Unified Shared Memory (USM) to compare to explicit buffering. Only the differences in the two implementations will be highlighted. First, the signal and correlation arrays are allocated in USM, and then initialized with artificial data:

```

// Initialize signal and correlation arrays
auto sig1 = sycl::malloc_shared<float>(N + 2, sycl_device, sycl_context);
auto sig2 = sycl::malloc_shared<float>(N + 2, sycl_device, sycl_context);
auto corr = sycl::malloc_shared<float>(N + 2, sycl_device, sycl_context);

// Initialize input signals with artificial data
std::uint32_t seed = (unsigned)time(NULL); // Get RNG seed value
oneapi::mkl::rng::mcg31m1 engine(Q, seed); // Initialize RNG engine
// Set RNG distribution
oneapi::mkl::rng::uniform<float, oneapi::mkl::rng::uniform_method::standard>
    rng_distribution(-0.00005, 0.00005);

// Warning: These statements run on the device.
auto evt1 =
    oneapi::mkl::rng::generate(rng_distribution, engine, N, sig1); // Noise
auto evt2 = oneapi::mkl::rng::generate(rng_distribution, engine, N, sig2);
evt1.wait();
evt2.wait();

// Warning: These statements run on the host, so sig1 and sig2 will have to be
// updated on the device.
sig1[N - N / 4 - 1] = 1.0;
sig1[N - N / 4] = 1.0;
sig1[N - N / 4 + 1] = 1.0; // Signal
sig2[N / 4 - 1] = 1.0;
sig2[N / 4] = 1.0;
sig2[N / 4 + 1] = 1.0;

```

The rest of the implementation is largely the same, except pointers to USM are passed to the oneMKL functions instead of SYCL buffers:

```

// Perform forward transforms on real arrays
evt1 = oneapi::mkl::dft::compute_forward(transform_plan, sig1);
evt2 = oneapi::mkl::dft::compute_forward(transform_plan, sig2);

// Compute: DFT(sig1) * CONJG(DFT(sig2))
oneapi::mkl::vm::mulbyconj(
    Q, N / 2, reinterpret_cast<std::complex<float>*>(sig1),
    reinterpret_cast<std::complex<float>*>(sig2),
    reinterpret_cast<std::complex<float>*>(corr), {evt1, evt2})
    .wait();

// Perform backward transform on complex correlation array
oneapi::mkl::dft::compute_backward(transform_plan, corr).wait();

```

It is also necessary to free the allocated memory:

```

sycl::free(sig1, sycl_context);
sycl::free(sig2, sycl_context);
sycl::free(corr, sycl_context);

```

The USM implementation has a more familiar syntax. It is also conceptually simpler because it relies on implicit data transfer handled by the DPC++ runtime. However, a programmer error hurts performance.

Notice the warning messages in the previous code snippets. The oneMKL random number generation engine is initialized on the device, so *sig1* and *sig2* are initialized with random noise on the device. Unfortunately, the code adding the artificial signal runs on the host, so the DPC++ runtime has to make the host and device data consistent. The signals used in Fourier correlation are usually large, especially in 3D imaging applications, so unnecessary data transfer degrades performance.

Updating the signal data directly on the device keeps the data consistent, thereby avoiding the unnecessary data transfer:

```

Q.single_task<>([=] () {
    sig1[N - N / 4 - 1] = 1.0;
    sig1[N - N / 4] = 1.0;
    sig1[N - N / 4 + 1] = 1.0; // Signal
    sig2[N / 4 - 1] = 1.0;
    sig2[N / 4] = 1.0;
    sig2[N / 4 + 1] = 1.0;
}).wait();

```

The explicit buffering and USM implementations have equivalent performance, indicating that the DPC++ runtime is good at avoiding unnecessary data transfers (provided the programmer pays attention to data consistency).

The complete source code ([fcorr_1d_usm.cpp](#)) is available in the oneAPI-samples repository.

Final Thoughts

Note that the final step of finding the location of the maximum correlation value is performed on the host:

```
// Find the shift that gives maximum correlation value
float max_corr = 0.0;
int shift = 0;
for (unsigned int idx = 0; idx < N; idx++) {
    if (corr[idx] > max_corr) {
        max_corr = corr[idx];
        shift = idx;
    }
}
shift =
    (shift > N / 2) ? shift - N : shift; // Treat the signals as circularly
                                        // shifted versions of each other.
std::cout << "Shift the second signal " << shift
    << " elements relative to the first signal to get a maximum, "
    << "normalized correlation score of "
    << max_corr / N << "." << std::endl;
```

It would be better to do this computation on the device, especially when the input data is large. Fortunately, the MAXLOC reduction is a common parallel pattern that can be implemented using DPC++. This is left as an exercise for the reader, but Figure 14-11 of [Data Parallel C++](#) provides a suitable example to help you get started. If you're not in the mood to exercise, the USM example in the oneAPI-samples repository has the MAXLOC reduction already implemented so that the entire computation is done on the device.

Using oneAPI to Speed Up the Finite-Difference Method

Migrating a CUDA-Based Stencil Computation to DPC++

Clícia Pinto, Technical Leader and Performance Engineer; and Lucas Batista, Pedro de Santana, and Georgina González, HPC Developers; Supercomputing Center SENAI CIMATEC

Reverse Time Migration (RTM) takes advantage of the finite-difference (FD) method to compute numerical approximations for the acoustic wave equation. It is a computational bottleneck for RTM applications, and therefore needs to be optimized to guarantee timely results and efficiency when allocating resources for hydrocarbon exploration. This article describes our experience migrating a CUDA-based RTM code to Data Parallel C++ (DPC++) using the Intel® DPC++ Compatibility Tool.

RTM Overview and Input Data

In several seismic imaging methods, a stencil is applied in the FD scheme as a numerical solution for the wave equation. This is the case for RTM, widely used in the oil and gas industry to generate images of subsurface structures. Despite the advantages inherent in the method, two major computational bottlenecks characterize it: the high number of floating-point operations during the propagation step and the difficulty in storing the wavefields in main memory. To mitigate the effect of these bottlenecks, engineers seek to explore both the intrinsic parallelism of tasks and the optimization of computational resources, designing solutions capable of running on different accelerators. The optimization of this method represents a great economic advantage for exploration geophysics because it reduces the chances of errors in well drilling. As proposed by Claerbout¹, the RTM algorithm usually has a forward time propagation, a backward propagation, and a cross-correlation of image condition. The flowchart for the RTM algorithm highlights these steps along with host and device communication (Figure 1).

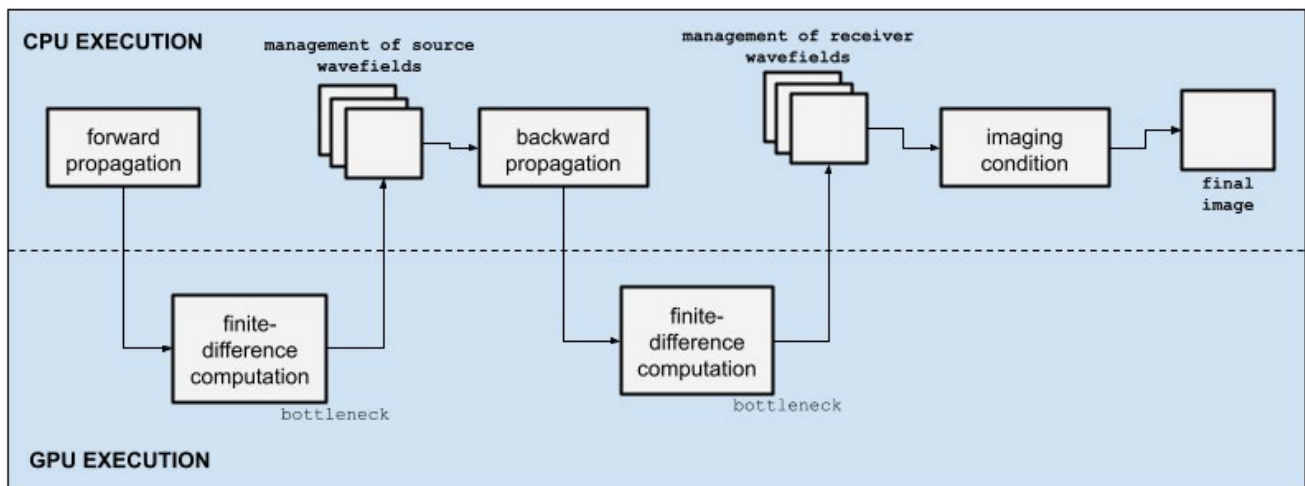


Figure 1. Simplified 2D-RTM flowchart

Figure 2 shows the stencil computation in a 2D CUDA-based implementation where order is the order of the FD scheme, n_x and n_z are the size of the input matrix that represents a 2D acoustic velocity model, p is the source/receiver wavefield, c_x and c_y are the x- and y-axes of the FD coefficients, respectively, and l is the extrapolated wavefield. In the entire RTM algorithm, the vector P stores the state of pressure points in different time steps. Because of its compute- and data-intensive characteristics, RTM is a suitable candidate for acceleration by specialized processing units.

```

Input: order, nx, nz, *p, *l, *cx, *cy
Output: *l
Assignments: h_order ← order/2
i ← h_order + blockDim.x * blockIdx.x + threadIdx.x
j ← h_order + blockDim.y * blockDim.y + threadIdx.y
mult ← i * nz
if < nx - h_order then
    if < nz - h_order then
        for k = 0 to h_order do
            aux = k - h_order
            aux += p[mult + j+aux] * cz[k]
            aux += p[(i+aux)*nz + j] * cx[k]
        end for
        l[mult + j] = accz + accx
        accz = 0
        accx = 0
    end if
end if

```

Figure 2. RTM Stencil Computation Algorithm

Migrating a Reference RTM to oneAPI

The Intel DPC++ Compatibility Tool helps port CUDA language kernels and library API calls to DPC++. Typically, 80–90% of CUDA source code is automatically migrated, so we structured this process in three steps: preparation, migration, and review. The preparation step seeks to adapt the source code to the migration tool. At this stage, it is necessary to make sure that all CUDA header files are accessible in the default location or in a custom location by using the `--cuda-include-path=<path/to/cuda/include>` option. In the migration step, the Intel DPC++ Compatibility Tool takes the original application as input and generates annotated DPC++ code. During the review step, we inspect the automatic code conversions, review the annotations to help manually convert unmigrated code, and look for possibilities for code improvement.

During our first migration experience, we observed that the Intel DPC++ Compatibility Tool migrates CUDA memory-copy API calls to `sycl::queue.memcpy()` as shown in **Figure 3**. Despite having obtained a functional and error-free migrated source code, explicit memory management may not provide the best performance. To investigate memory management improvements, we manually changed the migrated source code to use SYCL buffers and accessors for each data object.

```

dpct::device_ext &dev_ctl = dpct::get_current_device();
sycl::queue &q_ctl = dev_ctl.default_queue();
q_ctl.memcpy(d_p, p[0], mtxBufferLength).wait();
    q_ctl.memcpy(d_pp, pp[0], mtxBufferLength).wait();
    q_ctl.memcpy(d_v2, v2[0], mtxBufferLength).wait();
    q_ctl.memcpy(d_coefs_x, coefs_x, coefsBufferLength).wait();
    q_ctl.memcpy(d_coefs_z, coefs_z, coefsBufferLength).wait();
    q_ctl.memcpy(d_taperx, taperx, brdBufferLength).wait();
    q_ctl.memcpy(d_taperz, taperz, brdBufferLength).wait();

```

Figure 3. Migrated memory management

```

void fd_forward(int order, float **p, float **pp, float **v2, int nz, int nx, int nt, int is,
int sz, int *sx, float *srce, int propag)
{
dpct::device_ext &dev_ctl = dpct::get_current_device();
sycl::queue &q_ctl = dev_ctl.default_queue();
sycl::range<3> dimGrid(1, gridz, gridx);
sycl::range<3> dimGridTaper(1, gridBorder_z, gridx);
sycl::range<3> dimGridSingle(1, 1, 1);
sycl::range<3> dimGridUpb(1, 1, gridx);
sycl::range<3> dimBlock(1, sizeblock, sizeblock);

{
sycl::buffer<float, 1> *b_p = new sycl::buffer<float, 1>(p[0], sycl::range<1>(nx*nze));
sycl::buffer<float, 1> *b_pp = new sycl::buffer<float, 1>(pp[0], sycl::range<1>(nx*nze));
sycl::buffer<float, 1> b_v2(v2[0], sycl::range<1>(nx*nze));
sycl::buffer<float, 1> b_coefs_x(coefs_x, sycl::range<1>(order+1));
sycl::buffer<float, 1> b_coefs_z(coefs_z, sycl::range<1>(order+1));
sycl::buffer<float, 1> b_taperx(taper_x, sycl::range<1>(nxb));
sycl::buffer<float, 1> b_taperz(taper_z, sycl::range<1>(nxb));
sycl::buffer<float, 1> *b_swap;

for (int it = 0; it < nt; it++){
    b_swap = b_pp;
    b_pp = b_p;
    b_p = b_swap;

    kernel_tapper()
    kernel_lap()
    kernel_time()
    kernel_scr()
}
}
}

```

Figure 4. DPC++ Forward propagation (simplified)

Figure 4 shows DPC++ migrated function that performs the forward time propagation, where nt is the number of time steps needed to model the wave equation. This algorithm shows data objects defined as buffers that are used to control and modify the device's memory. The backward propagation follows the same structure. Both forward and backward functions perform kernel invocations to handle GPU execution.

```
q_ct1.submit([&](sycl::handler &cgh) {
    auto acc_pr = b_pr->get_access<sycl::access::mode::read_write>(cgh);
    auto d_laplace_ct4 = d_laplace;
    auto acc_coefs_x = b_coefs_x.get_access<sycl::access::mode::read>(cgh);
    auto acc_coefs_z = b_coefs_z.get_access<sycl::access::mode::read>(cgh);

    cgh.parallel_for(
        sycl::nd_range<3>(dimGrid * dimBlock, dimBlock),
        [=](sycl::nd_item<3> item_ct1) {
            kernel_lap(order, nx, nz, acc_pr, d_laplace_ct4,
                acc_coefs_x, acc_coefs_z, item_ct1);
        });
});
```

Figure 5. DPC++ kernel_lap function invocation

Figure 5 shows DPC++ kernel invocation using the *kernel_lap* call because that is the main procedure related to stencil computation. Each kernel from our migrated application is submitted to queues targeting a specific device, where data access requirements must be completed before a parallel kernel is launched.

The final RTM image was used to compare the original CUDA implementation to the migrated DPC++ code. To generate the seismic image, we used the input parameters shown in **Table 1** and the velocity model shown in **Figure 6**. The respective output images from the CUDA and DPC++ implementations are shown in **Figure 7**. The final image from the DPC++ implementation achieved satisfactory accuracy compared to the reference.

Parameters	Values
points in z axis	195
points in x axis	375
time steps	1700
sample intervals in z	10
sample intervals in x	10
time step sample intervals	0.001
frequency peek	20.0
number of shots	6
order	8

Table 1. Parameter description for the specific modeling presented

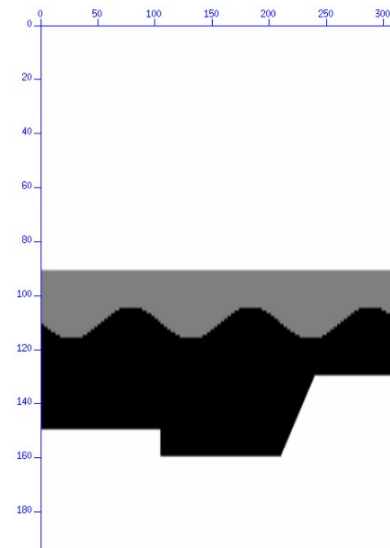


Figure 6. Koslov velocity model

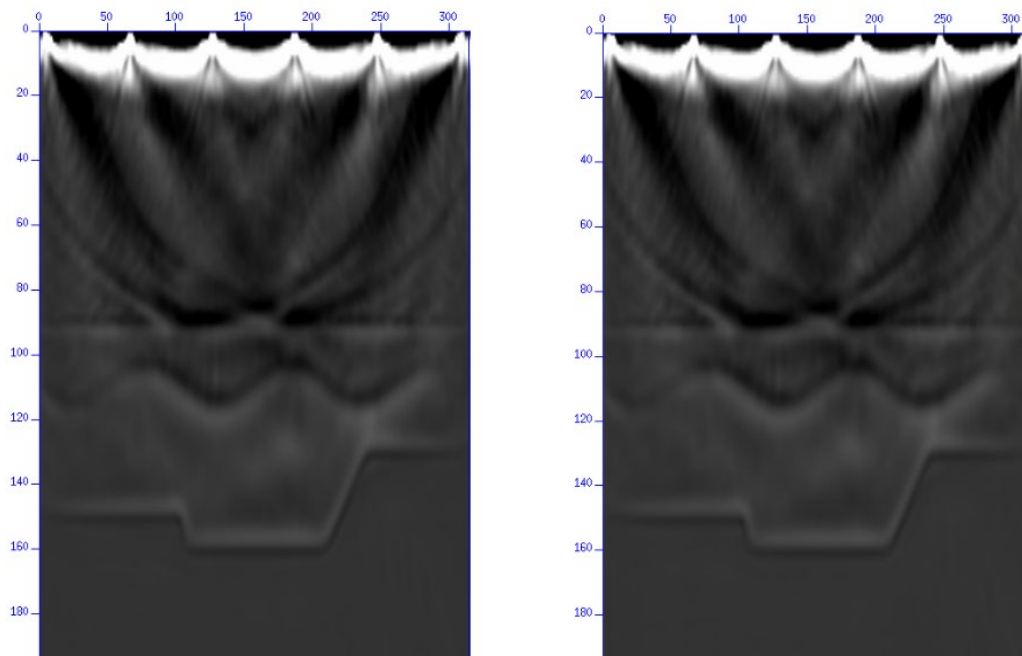


Figure 7. Seismic image generated by the original CUDA-based RTM source code (left) and the migrated DPC++ code (right)

Memory Management Improvements

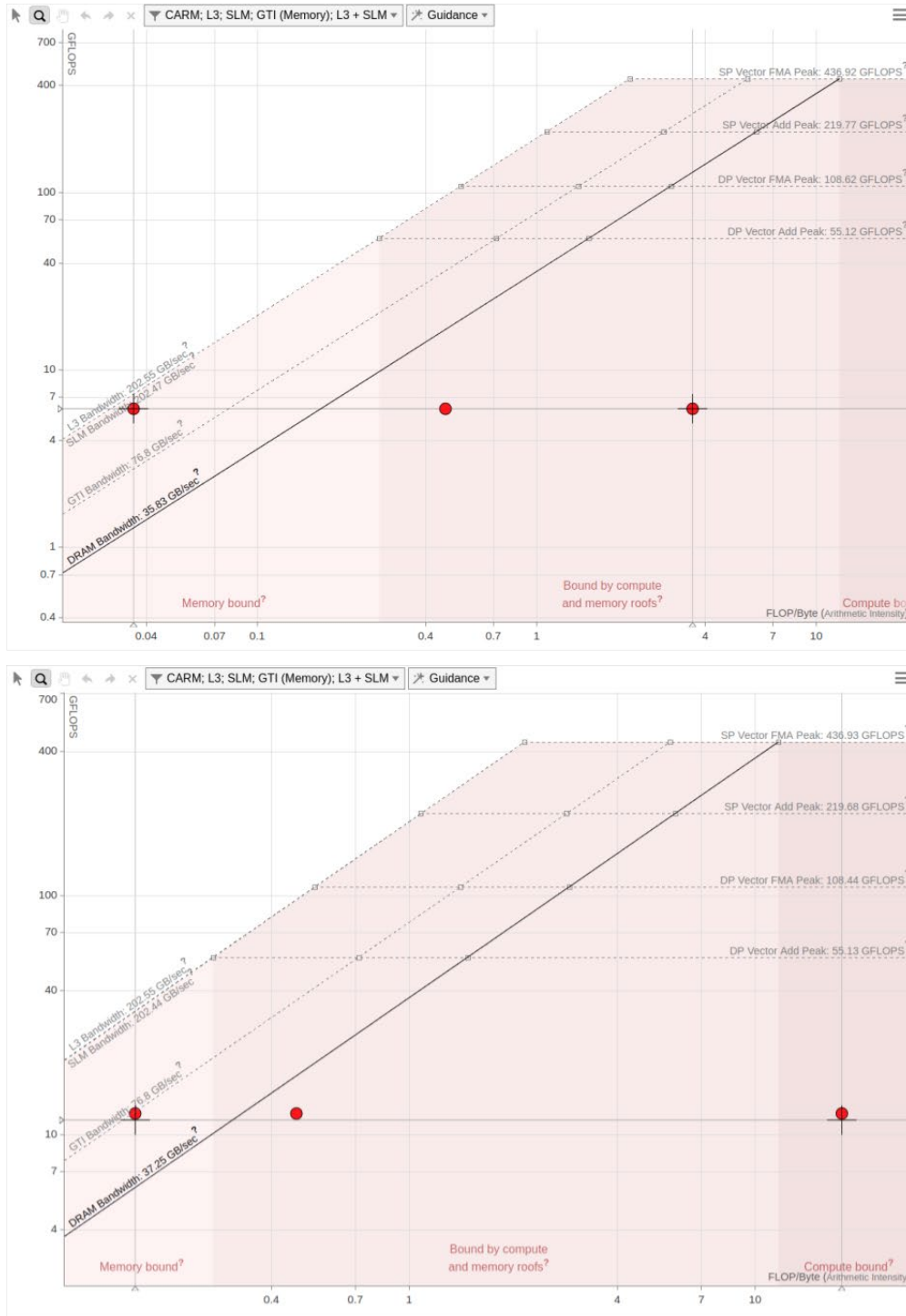


Figure 8. Roofline views of the DPC++ version running on an Intel® Gen9 HD Graphics NEO using explicit memory allocation (top) and buffers/accessors memory management (bottom)

In order to explore DPC++ memory management options, we developed a buffer/accessor-based version from our migrated application. This strategy eliminates the need to explicitly allocate and free memory on the device. It also eliminates the need to manage data transfer to/from different processing units. To achieve this, we created buffers/accessors for each structure that were explicitly copied to the device. The buffers are destroyed after the computation is completed, and data are copied back to the host (memory synchronization). Braces create a scope around the buffer definition where data objects can be shared. When execution leaves this scope, there is a synchronization between execution flows and the buffers are destroyed. For comparison between original and modified migrated source code, we performed a roofline analysis using Intel® Advisor to estimate the performance by evaluating the hardware limitations and data transactions between the different memory layers on the system.

Figure 8 shows the roofline graph for a simplified RTM execution with only a single time step. Since we have a reduced number of floating-point operations, we can expect low-performance metrics. The top graph shows the roofline for the migrated source code that uses explicit data management. It achieved a performance of 6.052 GFLOPS, with an arithmetic intensity of 3.617 FLOP/byte. Arithmetic intensity can be understood as the ratio of total floating-point operations to the amount of data being moved (memory traffic). The bottom graph shows the roofline for the buffer/accessor version, which achieved twice the performance: 12.246 GFLOPS with an arithmetic intensity of 17.896 FLOP/byte.

Conclusion

This paper describes a successful oneAPI proof-of-concept to migrate an RTM code from CUDA to DPC++ using the Intel DPC++ Compatibility Tool, and then tune it using Intel Advisor. The migrated source code is more readable and easier to maintain because it unifies the algorithm execution flow for our application in a unique structure. Besides migration experience, we could easily explore memory management by applying buffers/accessors to achieve better performance and arithmetic intensity. The source code described in this article is available in a public repository along with instructions².

References

1. Claerbout, JF. Toward a unified theory of reflector mapping. *Geophysics*, v. 36, n. 3, p. 467-481, 1971.
2. <https://github.com/cs2isenaicimatec/finite-difference-computation/>



Intel® DPC++ Compatibility Tool
Transform Your CUDA Applications into Standards-Based Data Parallel C++ Code

LEARN MORE



Advanced Ray Tracing APIs Proposed for the oneAPI Specification

“Write Once” High-Fidelity Ray Tracing across Multiple Vendors’ Accelerators

Jim Jeffers, Senior Principal Engineer, Senior Director of Advanced Rendering and Visualization, Intel Corporation

Image courtesy Bentley Motors Limited: Advanced Ray Tracing via OSPRay Studio, Embree and Open Image Denoise

I'm pleased to announce that a set of [Advanced Ray Tracing APIs](#) are being made available for comment and inclusion in the [oneAPI specification](#). The rapid growth of ray tracing compute across film, scientific visualization, design, and gaming suggests that adding these APIs to the oneAPI specification for XPU architectures will help foster robust and efficient development in this area.

By introducing ray tracing capabilities to the oneAPI specification, software developers across the industry will have the ability to “write once” for high-fidelity ray-traced computations across multiple vendors’ systems and accelerators. Standardizing these interfaces will provide well-designed, tried and true APIs and options for broad compute and rendering infrastructure development. The functionality is subdivided into several domains:

- Geometric ray tracing computations
- Volumetric computation and rendering
- Image denoising
- Scalable rendering and visualization infrastructure

Open source implementations of the ray tracing APIs are in active use via the [Intel® oneAPI Rendering Toolkit](#). The libraries are used in a wide variety of software applications’ 3D graphics computations for film and television photorealistic visual effects and animation, scientific visualization, high-performance computing computations, computer-aided design, architectural engineering, gaming, and more.

The ray tracing libraries recommended for the oneAPI specification include:

- [Academy Award*-winning Intel® Embree](#) — geometric ray tracing
- [Intel® Open Volume Kernel Library](#) (Open VKL) — volumetric processing
- [Intel® Open Image Denoise](#) — AI-based denoiser
- [Intel® OSPRay](#) — scalable middleware rendering API

The specification review process and robust community feedback on these ray tracing APIs’ features and functionality, recent adoption by Apple and others on ARM-based processors, and Intel’s own development preparation for future Intel® Iris® Xe graphics confirm these ray tracing APIs are cross-architecture and industry-standard ready. The ease of integration and benefits the APIs provide for developers are well known by open source community partners, and make this a logical next step in delivering the benefits of an open, community-driven, cross-platform, ray tracing ecosystem.

I invite developer feedback and collaboration to further extend these APIs for application development.

Contribute to the oneAPI specification through this [GitHub repository](#).



PODCAST
On a Mission of Disaster Management & Scientific Discoveries

LISTEN NOW

If you’re an animator, digital content creator, architectural engineer, or skilled gamer, push the boundaries of visualization with the Intel oneAPI + Rendering Toolkit.
[Learn More >](#)



oneTBB Flow Graph and the OpenVINO™ Inference Engine

Expressing Dependencies across Deep Learning Models in C++

Elvis G Fefey, Software Development Engineer, Michael J Voss, Principal Software Engineer, and Maxim Y Shevtsov, Deep Learning Software Engineer, Intel Corporation

OpenVINO™ and the Need for a Coordination Framework

With the increasing availability of data in today's world, traditional approaches to solving problems are being replaced by machine learning (ML) methods that learn from the data. The two main stages in an end-to-end ML pipeline are training and inferencing. Training is where data is used to create a model. Inferencing is where the model generates output from new data.

The Intel® Distribution of OpenVINO™ toolkit is a developer tool suite for high-performance deep learning inference on Intel® architectures. While we are starting to see the emergence of dedicated inference accelerators, the ubiquity of multicore CPUs means that better inference performance can deliver significant gains to a larger number of users than ever before. OpenVINO offers a multithreading model that is portable and free of low-level details. It's not necessary for users to explicitly start and stop any threads, or even know how many processors or cores are being used. This results in optimized performance that is easy to deploy.

As ML becomes more complex, inferencing applications require multiple models, with some models depending on the output of other models. Such applications require coordination that enforces dependencies among the models during execution, while allowing models that can make progress independently to execute concurrently.

The OpenVINO Inference Engine itself does not provide a way to piece different models together. Its primary role is to provide mechanisms to tune and deploy high-performing models onto Intel architectures. The Intel Distribution of OpenVINO toolkit, however, does include DL Streamer, an extension of the widely used, open source GStreamer framework. GStreamer is a framework for creating complex media analytics pipelines that enforces dependencies between models. DL Streamer extends GStreamer to provide pipeline interoperability and optimized inferencing across Intel architectures. Both DL Streamer and GStreamer are excellent choices for building complex media pipelines, but not all developers want or need these larger frameworks.

In this article, we describe how the Intel® oneAPI Threading Building Blocks (oneTBB) library included in the Intel Distribution of the OpenVINO toolkit can coordinate OpenVINO inferencing models using a lightweight C++ alternative to GStreamer or DL Streamer.

oneTBB Flow Graph

oneTBB is a generic C++ library for parallel programming on CPUs. It has a long history, being an evolution of the Threading Building Blocks (TBB) library that has been available since 2006. It provides generic parallel algorithms, a flow graph interface, concurrent containers, a task-based work scheduler, a scalable memory manager, and auxiliary features that make parallel programming easier. The flow graph feature provides functions and classes (in the `tbb::flow` namespace) for applications that can be expressed as graphs of computations. A flow graph is used when you want to express the execution dependencies in your code, or if you have a streaming application that requires more than just a simple linear pipeline.

The flow graph interface consists are three main types of components:

1. A *graph object* that represent a whole graph of computations
2. *Nodes* that execute user-supplied lambda expressions, join streams of data together, or split and broadcast data
3. *Edges* that express the dependencies or communication channels between nodes

Here is a summary of some of the nodes that are useful in building a graph of inference engine models:

- **source_node (or input_node):** A *source_node* provides functionality to generate data that is fed into the rest of the graph. This node can be used for reading and making available frames from a video stream. Note that in oneTBB, *source_node* has been replaced with *input_node*, which has a different API. However, the latest version of OpenVINO at the time of writing is shipped with *source_node*.
- **function_node:** A *function_node* body executes user-provided code on each data item that flows into the node to generate the data that flows out of the node. This node type can be used to run the inference computations. Function nodes that do not directly or indirectly depend on each other (as expressed by the graph edges) are allowed to execute concurrently. If the edges in the graph express a dependence between nodes, the oneTBB runtime library will enforce the dependence and ensure that the nodes execute in the correct order. A *function_node* can be configured to allow or disallow concurrent execution of its user-provided body on different data items as they flow through the node, making it suitable for operating on different data items in parallel, as well as for executing operations that must be serialized, such as displaying processed video frames in the correct order.
- **sequencer_node:** In cases where the output is required in a certain order, the sequencer node can be used to maintain that order. For example, it can be used to sequence video frames that have been computed out of order.
- **join_node:** The join node brings multiple streams of data together. If independently computed outputs need to be brought together as a unit, a *join_node* can be used.

Sample Application

To demonstrate how oneTBB is used to coordinate OpenVINO inferencing, we adapted a security barrier demo that is included in the Intel Distribution of OpenVINO toolkit. It implements inferencing on three trained models: a vehicle detector that detects vehicles in a video frame, a vehicle classifier that classifies the color and type of the detected vehicle, and a license plate recognition model that extracts the license plate text. The vehicle classifier and license plate recognition models depend on the vehicle detector model.

To set up the graph, we use a *source_node* to read the video frames, followed by a *function_node* to perform the vehicle detection. Two other function nodes are used to run the vehicle classifier and license plate recognition. A *join_node* is used to aggregate the results from each frame, followed by a *sequencer_node* to reorder the inferred video frames. The final node is another *function_node* that shows or saves the results as desired. The graph is shown in **Figure 1**.

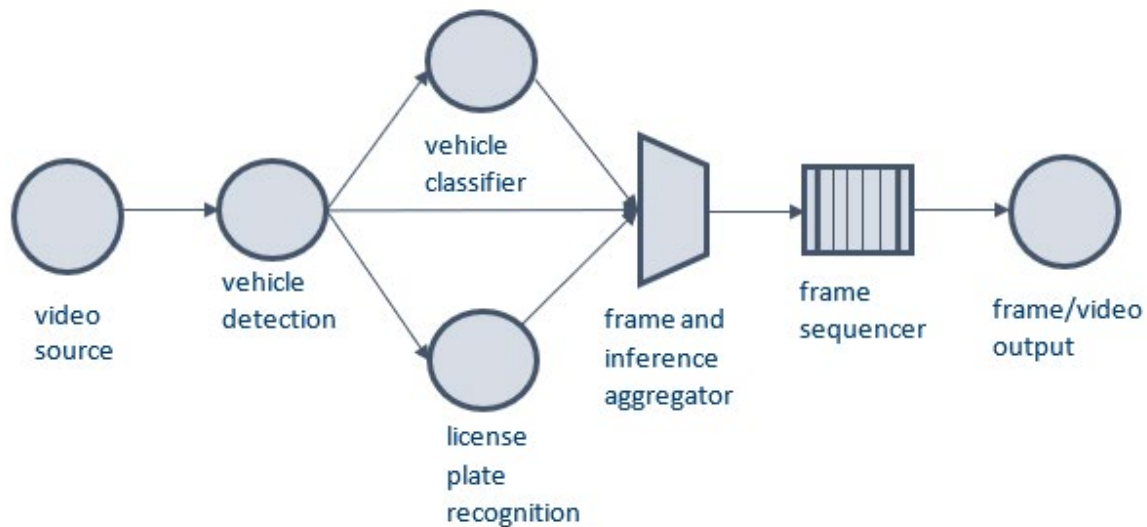


Figure1. Flow graph of the security barrier demo

Reading the Input Video Frames

We use a *source_node* to generate images from the video for inferencing. It will repeatedly execute its body and generate frames until a terminating condition is encountered. In this case, the terminating condition is an empty frame at the end of the video file. The template parameter of the *source_node* is the data type of the output. The node returns a *std::tuple* comprised of an image and a corresponding image number. The frame number is included because it is required downstream for aggregation and to reestablish the frame order. If this were not required, the frame number could be omitted.

We begin by including the header file with the flow graph nodes.

```
#include "tbb/flow_graph.h"
```

We declare several data types to hold the messages that flow through the graph. We use *f* to denote that the message includes the frame itself, *n* to mean that the messages includes the frame number, *v* to mean it includes the bounding rectangles for the vehicles, *p* to mean it includes the bounding rectangles for the license plates, and *s* to mean it includes a *vector<std::string>*.

```
using fn_t = std::tuple<cv::Mat, size_t>;
using fnvp_t = std::tuple<cv::Mat, size_t,
                        std::vector<cv::Rect> /* vehicles */,
                        std::vector<cv::Rect> /* plates */ >;
using sn_t = std::tuple<std::vector<std::string>, size_t>;
```

We also initialize a frame counter and a capture object to read the frames.

```
size_t frame_nu = 0;
VideoCapture my_video("video_file_name");
```

We then create a flow graph object and the *source_node* that reads the frames.

```
tbb::flow::graph g;
tbb::flow::source_node<fn_t> src_node(g,
  [&](fn_t &fn)->bool {
    cv::Mat f;
    my_video.read(f);
    if (f.empty()) { return false; }
    std::get<0>(fn) = f;
    std::get<1>(fn) = frame_nu++;
    return true;
  }, false);
```

Inference Computations

The security barrier demo comes with three classes for detection, classification, and license plate recognition. The classes provide functionality to create inference requests, to submit inference computations, and to collect results. We instantiate the following classes: *detector* for detection, *vclassifier* for classification, and *preader* for license plate recognition.

Vehicle Detection

A *function_node* is used for vehicle detection inference computations. There are two template parameters for the function node: the input data type that the function node receives and the data type that the function node sends out. The input of the vehicle detector is the pair of image and corresponding number received from the *source_node*. In the body of the node, we create an inference request, prepare the blob for inferencing, start the inference computation, and then wait for the results of the computation to be ready. The result from the detector is a confidence level, a label showing whether it is a vehicle or a license plate, and the coordinates of the detected object (x, y, width, and height). We populate a vector with a list of the detected objects on each frame and send it downstream for further processing. The output type, which we named *fnvp_t*, contains the frame, the frame number, and a list of locations of the detected vehicles.

```
tbb::flow::function_node<fn_t, fnvp_t> detector_node(g,
    tbb::flow::unlimited, [&](const fn_t &fn)->fnvp_t
{
    std::vector<cv::Rect> vehicles, plates;
    auto req = detector.createInferRequest();
    auto& f = std::get<0>(fn);
    auto& n = std::get<1>(fn);
    detector.setImage(req, f);
    req.StartAsync();
    req.Wait(IInferRequest::WaitMode::RESULT_READY);
    auto results = detector.getResults(req, f.size());
    for (auto& r : results) {
        if (r.label == 1) {
            if (r.confidence > 0.5) {
                vehicles.push_back(r.location &
                    cv::Rect{cv::Point(0, 0), f.size()});
            }
        }
        if (r.label == 2) {
            if (r.confidence > 0.5) {
                plates.push_back(r.location &
                    cv::Rect{cv::Point(0, 0), f.size()});
            }
        }
    }
    return std::make_tuple(f, n, vehicles, plates);
});
```

Detected Vehicle Classification

The vehicle classifier receives the frame, the frame number, and a list of detected vehicle locations. For each vehicle, we create an inference request, prepare the blob, start the inferencing to classify the vehicle, and then collect the results when they are ready. The classification results are the color and type of the vehicle. This is passed downstream for further processing.

The datatype for the output of the classifier is a list of strings.

```
tbb::flow::function_node<fnvp_t, sn_t> vclassifier_node(g,
    tbb::flow::unlimited, [&](const fnvp_t &fnvp)->sn_t
{
    auto& f = std::get<0>(fnvp);
    auto& n = std::get<1>(fnvp);
    auto& v = std::get<2>(fnvp);
    std::vector<std::string> rlist;

    if (!FLAGS_m_va.empty() && !v.empty()) {
        for( auto& x : v ) {
            auto req = vclassifier.createInferRequest();
            vclassifier.setImage(req, f, x);
            req.StartAsync();
            req.Wait(IIInferRequest::WaitMode::RESULT_READY);
            auto r = vclassifier.getResults(req);
            rlist.push_back((r.first + " " + r.second));
        }
    }
    return std::make_tuple(rlist, n);
});
```

Detected Vehicle License Plate Recognition

Like the classifier, the license plate recognition receives the frame, its number, and a list of detected vehicles locations. Likewise, an inference request is created and submitted. The result is a string of the license plate number.

```
tbb::flow::function_node<fnvp_t, sn_t> preader_node(g,
    tbb::flow::unlimited, [&](fnvp_t fnvp)->sn_t
{
    auto& f = std::get<0>(fnvp);
    auto& n = std::get<1>(fnvp);
    auto& p = std::get<3>(fnvp);
    std::vector<std::string> rlist;

    if (!FLAGS_m_lpr.empty() && !p.empty()) {
        for( auto& x : p ) {
            auto req = preader.createInferRequest();
            preader.setImage(req, f, x);
            req.StartAsync();
            req.Wait(IIInferRequest::WaitMode::RESULT_READY);
            auto r = preader.getResults(req);
            rlist.push_back(r);
        }
    }
    return std::make_tuple( rlist, n);
});
```


If only the raw inferencing results are required, the nodes described to this point should be enough. However, if postprocessing is required, the nodes that follow can be added.

Aggregation

Here we aggregate the computations for each frame. Using the frame number as the key, a tag-matching *join_node* is used to put together all the results for each frame. Each frame will therefore have its number, the classification result of any detected vehicles, and a string of any detected license plates.

```
tbb::flow::join_node<std::tuple<fnvp_t, sn_t, sn_t,
                                tbb::flow::tag_matching >
agg_node(g, [] (const fnvp_t &x) { return std::get<1>(x); },
          [] (const sn_t &x)    { return std::get<1>(x); },
          [] (const sn_t &x)    { return std::get<1>(x); });
```

Sequencing

The computation is done in parallel, so it is possible that the inferencing results for the frames can complete out of order. With a *sequencer_node*, we can reorder the frames if video playback is required.

```
tbb::flow::sequencer_node<std::tuple<fnvp_t, sn_t, sn_t >>
seq_node(g, [] (const std::tuple<fnvp_t, sn_t, sn_t > &x) -> size_t {
    return std::get<1>(std::get<0>(x));
});
```

Superimposing the Results

Lastly, we implement a *function_node* that superimposes the locations of the detected vehicles and license plate numbers onto the frame. The results can be played back or saved to an output video. In cases where the computations complete much faster than the playback rate, a throttling mechanism can be used to prevent a build-up of processed frames.

```
tbb::flow::function_node<std::tuple<fnvp_t, sn_t, sn_t>, cv::Mat>
    output_node(g, tbb::flow::serial,
    [&](std::tuple<fnvp_t, sn_t, sn_t> agg_results)
{
    auto& f = std::get<0>(std::get<0>(agg_results));
    auto& v = std::get<2>(std::get<0>(agg_results));
    auto& p = std::get<3>(std::get<0>(agg_results));
    auto& vs = std::get<0>(std::get<1>(agg_results));
    auto& ps = std::get<0>(std::get<2>(agg_results));
    auto frame_nu = std::get<1>(std::get<0>(agg_results));

    for (auto &r : v) {
        cv::rectangle(f, r, {0, 255, 0}, 4);
    }
    for (auto &r : p) {
        cv::rectangle(f, r, {0, 255, 0}, 4);
    }
    if(!v.empty()) {
        for (size_t i=0; i< v.size(); ++i) {
            cv::putText( f, vs[i], cv::Point{v[i].x, v[i].y + 35},
            cv::FONT_HERSHEY_COMPLEX, 1.3, cv::Scalar(0, 255, 0), 4);
        }
    }

    if (!p.empty()) {
        for (size_t i=0; i< p.size(); ++i) {
            slog::info << "Frame Number: " << frame_nu << slog::endl;
            slog::info << "License Plate: " << ps[i] << slog::endl;
            slog::info << "====" << slog::endl;

            cv::putText( f, ps[i], cv::Point{p[i].x, p[i].y + 35},
            cv::FONT_HERSHEY_COMPLEX, 1.3, cv::Scalar(0, 255, 0), 4);
        }
    }
    return f;
});
```

Building the Graph and Expressing Dependencies

The graph is built and dependencies are expressed with the oneTBB function `tbb::flow::make_edge`. It takes a predecessor node and a successor node. This can be used to build any desired graph topology. The only requirement is to make sure that the data types on the edges match. That is, the data type of the output node matches the data type of the input node.

We build the graph by making an edge from the frame generation to the vehicle detection.

```
make_edge(src_node, detector_node);
```

The vehicle detector is connected in parallel to the classifier and license plate recognition.

```
make_edge(detector_node, vclassifier_node);
make_edge(detector_node, preader_node);
```

The detector, classifier, and license plate recognition are connected to the aggregator. The detector provides the frame, while the classifier and the license plate recognition provide the inference results for that frame for aggregation.

```
make_edge(detector_node, tbb::flow::input_port<0>(agg_node));
make_edge(vclassifier_node, tbb::flow::input_port<1>(agg_node));
make_edge(preader_node, tbb::flow::input_port<2>(agg_node));
```

The aggregator is connected to the sequencer for reordering of the frames.

```
make_edge(agg_node, seq_node);
```

The sequencer is connected to the output node to output the results.

```
make_edge(seq_node, output_node);
```

Activating the Graph

The final step is to activate the graph so that it can run. This is done with the `activate()` member of the source node. A call to `g.wait_for_all()` ensures that all computations in the graph are completed before the graph exits.

```
src_node.activate();
g.wait_for_all();
```

Results

To evaluate our implementation, we used pretrained models from the [Open Model Zoo](#) repository: vehicle-license-plate-detection-barrier-0106_fp16, vehicle-attributes-recognition-barrier-0039_fp16, and license-plate-recognition-barrier-0001. We ran our application on an Intel® Core™ i7-6770HQ processor with integrated Iris Pro Graphics 580. The elapsed time for running all inferences on 32,030 frames from a video was measured. The performance gain of the oneTBB flow graph over a corresponding manually threaded implementation is shown in **Figure 2**. Performance gains of up to 11% were obtained. We evaluated five different configurations that assigned models differently to the CPU and integrated GPU.

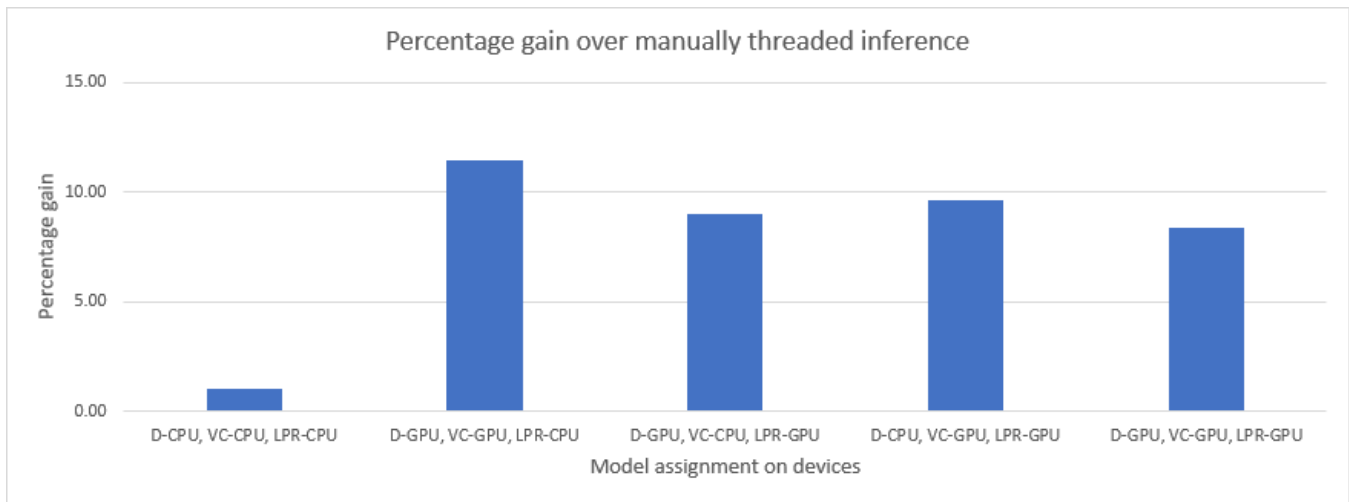


Figure 2. Performance results of the flow graph security barrier demo. D-CPU, VC-CPU, LPR-CPU: All three models run on the CPU. D-GPU, VC-GPU, LPR-GPU: All three models run on the GPU. D-GPU, VC-CPU, LPR-CPU: Detection and classification run on the GPU, while recognition runs on the CPU. D-GPU, VC-CPU, LPR-GPU: Detection and recognition run on the GPU, while classification runs on the CPU. D-CPU, VC-GPU, LPR-GPU: Classification and recognition run on the GPU, while detection runs on the CPU.

Where to Get oneTBB Flow Graph

The Intel Distribution of OpenVINO toolkit already ships with oneTBB included, so using the flow graph is not an additional dependency as it is already there in OpenVINO. One simply needs to update the build infrastructure (CMakeLists.txt) to enable its use:

```
include_directories (/path/to/opencvino/inference_engine/external/tbb/include)
link_libraries (-L/path/to/opencvino/inference_engine/external/tbb/lib -ltbb)
```

Final Thoughts

We have shown how to use a oneTBB flow graph to express dependencies across models used for inferencing. The flow graph makes it easy to express such dependencies, as well as express parallelism across the models directly in C++. For developers who don't want to use larger frameworks like Gstreamer or DL Streamer, we believe that using a oneTBB flow graph is a good option to consider. Our implementation of the OpenVINO security barrier demo showed performance gains of up to 11% over a manually threaded implementation when using a oneTBB flow graph. The fact that the flow graph is a C++ framework, is shipped with Intel Distribution of OpenVINO toolkit, and has support for threading makes it a natural choice to build graphs of ML models for inferencing in OpenVINO.

Optimization of Scan Operations Using Explicit Vectorization

Exploiting Intel® AVX-512 SIMD Instructions to Accelerate Prefix Sum Computations

Vamsi Sripathi and Ruchira Sasanka, Senior HPC Application Engineers, Intel Corporation

Introduction

Modern Intel® processors offer instruction-, data-, and thread-level parallelism. The ability to simultaneously execute a single instruction on multiple data (SIMD) operands maximizes utilization of processor arithmetic execution units. In this article, we will focus on SIMD optimizations applied to vector scan operations.

Scan (also known as inclusive/exclusive scan, prefix sum, or cumulative sum) is a common operation in many application domains¹. As such, it is defined as a standard library function in C++, the OpenMP runtime, and

the Python* NumPy package^{2,3}. A scan of a vector is another vector where the result at index i is obtained by summing all the values up to index i (for inclusive scan) and $i-1$ (for exclusive scan) from the source vector. For instance, the inclusive scan of vector x with n elements is obtained by:

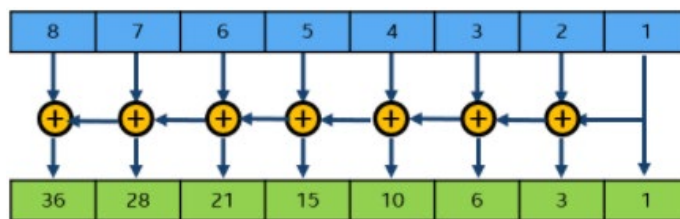
$$y_0 = x_0$$

$$y_1 = x_0 + x_1$$

$$y_2 = x_0 + x_1 + x_2$$

And generally defined as:

$$y_j = \sum_{i=0}^j x_i$$



We will be demonstrating inclusive scan, henceforth referred to simply as scan. We'll begin with a brief overview of SIMD, also known as vectorization, followed by a baseline implementation of the scan operation in C and with OpenMP directives. Next, we'll implement an optimized scan algorithm in SIMD operating on 512-bit vector registers. We'll conclude with performance comparisons of the baseline and optimized implementations.

SIMD Overview

SIMD instructions operate on wider vector registers that can hold multiple data operands. The width and the number of vector registers are architecture-dependent, with the latest generation of Intel processors supporting 32, 512-bit-wide vector registers. Intel provides x86 ISA extensions in the form of Advanced Vector Extensions (AVX), AVX2, and AVX-512 that allow various arithmetic and logical operations to be performed on the 256-bit- and 512-bit-wide registers, respectively. Depending on the width of vector register, they are named XMM, YMM, or ZMM (**Table 1**).

Register Width	Register Name	Num. 64b operands/register	x86 ISA	Num. 64b ops per SIMD Add
128-bit	XMM	2	SSE	2
256-bit	YMM	4	AVX/AVX2	4
512-bit	ZMM	8	AVX512	8

Table 1. SIMD register naming

SIMD execution is mainly facilitated through two methods:

1. Implicit vectorization: This relies on the compiler to automatically transform the scalar computational loops into vectorized/SIMD blocks. The compiler does the heavy lifting of identifying which computations to vectorize, how aggressively to unroll loops, etc. based on a cost model. It also handles the data alignment requirements of SIMD instructions by producing the necessary loop prologue and epilogue sections in the generated object code. All the programmer has to do is use the appropriate compiler optimization flags for the target CPU architecture. Intel® compilers also provide pragmas⁴ to give explicit hints to control the generation of SIMD instructions.
2. Explicit vectorization:
 - a. Vector intrinsics: These are C APIs provided by the compiler that map to the underlying hardware SIMD instructions⁵. They allow the programming to control the type of SIMD instructions (AVX, AVX2, and AVX-512) that are generated in the code and how aggressively they are used. This can be viewed as a hybrid mode wherein the programmer still relies on the compiler for critical aspects of tuning, such as register renaming and instruction scheduling, but at the same time has the ability to control the code generation of non-trivial, data-dependent operations, such as scan.
 - b. Assembly language: This completely bypasses compiler optimization in favor of assembly language coding, which requires expert knowledge of underlying microarchitecture. This approach is not recommended.

The tradeoffs of these approaches are summarized in **Table 2**.

Mode	Portability	Effort
Implicit Vectorization	High	Low
Vector Intrinsics	Medium	Medium
Assembly Instructions	Low	High

Table 2. Comparison of SIMD programming approaches

Baseline Code

The baseline code for computing the scan of a vector is shown below, followed by the Intel compiler optimization report (generated using the **-qopt-report=5** compiler option). The scan operation has inherent loop-carried dependencies, so the compiler is unable to fully vectorize the computation.

```

8 void ref_scan (int *p_n, double *restrict src, double *restrict dst, double *p_init_val)
9 {
10  int n = *p_n;
11  double tmp = *p_init_val;
12
13  for (int i=0; i<n; i++) {
14    tmp += src[i];
15    dst[i] = tmp;
16  }
17 }

```

```

LOOP BEGIN at kernels.c(13,2)
  remark #15344: loop was not vectorized: vector dependence prevents vectorization
  remark #15346: vector dependence: assumed ANTI dependence between tmp (14:3) and tmp (14:3)
  remark #15346: vector dependence: assumed FLOW dependence between tmp (14:3) and tmp (14:3)
  remark #25439: unrolled with remainder by 2
LOOP END

```


The same operation with the addition of OpenMP directives is shown below, along with the compiler optimization report. OpenMP is a portable, directive-based parallel programming model that includes SIMD support. The SIMD construct (combined with the reduction clause and scan directive) can be used to perform a scan operation on the vector. (We refer the reader to the OpenMP specification, which provides good documentation^{6,7}.) Because we are interested in understanding the impact of SIMD on workload performance, we are not parallelizing the loop, but instead limiting the OpenMP directives to vectorization only. The compiler optimization report shows that vectorization was performed using a vector length of eight (i.e., AVX-512) and that the loop was unrolled by a factor of 16 elements.

```

20 void omp_scan (int *p_n, double *restrict src, double *restrict dst, double *p_init_val)
21 {
22     int n = *p_n;
23     double tmp = *p_init_val;
24
25     #pragma omp simd reduction(incscan, +:tmp)
26     for(int i=0; i<n; i++){
27         tmp += src[i];
28     #pragma omp scan inclusive(tmp)
29         dst[i] = tmp;
30     }
31 }

```

```

LOOP BEGIN at kernels.c(26,5)
remark #15389: vectorization support: reference src[i] has unaligned access [ kernels.c(27,14) ]
remark #15388: vectorization support: reference dst[i] has aligned access [ kernels.c(29,7) ]
remark #15381: vectorization support: unaligned access used inside loop body
remark #15305: vectorization support: vector length 8
remark #15399: vectorization support: unroll factor set to 2
remark #15309: vectorization support: normalized vectorization overhead 0.342
remark #15301: SIMD LOOP WAS VECTORIZED
remark #15449: unmasked aligned unit stride stores: 1
remark #15450: unmasked unaligned unit stride loads: 1
remark #15475: --- begin vector cost summary ---
remark #15476: scalar cost: 9
remark #15477: vector cost: 2.370
remark #15478: estimated potential speedup: 3.490
remark #15488: --- end vector cost summary ---
LOOP END

```

We will compare the baseline and auto-vectorized variants to the explicit AVX-512 SIMD implementation described in the next section.

Explicit Vectorization

In this section, we demonstrate the SIMD techniques to optimize the vector scan operations on Intel processors that support AVX-512 execution units. We will be using the double precision floating point (FP64) datatype for input and output vectors. The AVX-512 SIMD instructions used in our implementation are shown in **Table 3**.

The main idea behind our implementation is to simultaneously perform a series of add operations in the lower and upper 256-bit lanes of the 512-bit register after applying the necessary shuffle sequence. For every eight input elements, we perform a total of five adds and five permutes. All the add operations form a dependency chain because we are accumulating the results in a single register. However, some of the

permutes are independent of the add, so they can be executed simultaneously. **Figure 1** shows a visual representation of the implementation, followed by the code showing the main loop block where we are processing 16 elements per iteration.

Name	AVX-512 Vector Intrinsics API (x86 Instruction)	Description
SIMD Load	<code>__m512d _mm512_loadu_pd (void *mem_address);</code>	Load a set of eight FP64 elements from memory to AVX-512 vector register.
SIMD Add	<code>__m512d _mm512_add_pd (__m512d src1, __m512d src2);</code>	Add FP64 elements in <i>src1</i> and <i>src2</i> and return the results.
SIMD Permute with mask (128b lanes)	<code>__m512d _mm512_maskz_permute_pd (__mmask8 k, __m512d src, const int imm8);</code>	Shuffle FP64 elements in <i>src</i> within 128-bit lanes using the 8-bit control in <i>imm8</i> , and store the results using zeromask <i>k</i> . (Elements are zeroed out when corresponding mask bit is not set.)
SIMD Permute with mask (256b lanes)	<code>__m512d _mm512_maskz_permutex_pd (__mmask8 k, __m512d src, const int imm8);</code>	Shuffle FP64 elements in <i>src</i> within 256-bit lanes using the 8-bit control in <i>imm8</i> , and store the results using zeromask <i>k</i> . (Elements are zeroed out when corresponding mask bit is not set.)
SIMD Store	<code>void _mm512_storeu_pd (void *mem_address, __m512d src);</code>	Store a set of eight FP64 elements to memory from AVX-512 vector register.

Table 3. AVX-512 SIMD instructions

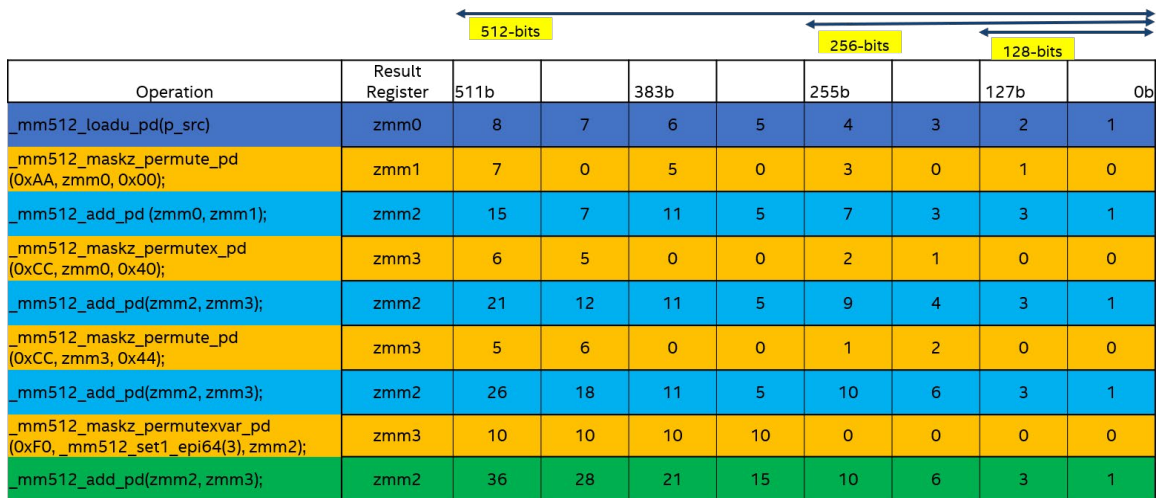


Figure 1. Explicit AVX-512 SIMD scan operation sequence

```

#include "immintrin.h"
#define N_UNROLL      (16)
#define NUM_ELES_IN_ZMM  (8)

void awe_scan (int *p_n, double *restrict src, double *restrict dst, double *p_init_val)
{
    __m512d zmm0, zmm1, zmm2, zmm3, zmm4, zmm_acc, zmm_tmp1 ,zmm_tmp2;
    __m512d zmm5, zmm6, zmm7, zmm8, zmm9, zmm10, zmm11;
    __m512i idx, acc_idx;

    int n = *p_n;

    zmm_acc = _mm512_set1_pd(*p_init_val);
    acc_idx = _mm512_set1_epi64(7);
    idx     = _mm512_set1_epi64(3);

    int n_block = (n/N_UNROLL)*N_UNROLL;
    int n_tail  = n - n_block;

    for (int j=0; j<n_block; j+=N_UNROLL) {
        zmm0 = _mm512_loadu_pd(src);
        zmm5 = _mm512_loadu_pd(src+NUM_ELES_IN_ZMM);

        zmm2 = _mm512_maskz_permute_pd(0xAA, zmm0, 0x00);
        zmm7 = _mm512_maskz_permute_pd(0xAA, zmm5, 0x00);
        zmm10 = _mm512_add_pd(zmm0, zmm2);
        zmm11 = _mm512_add_pd(zmm5, zmm7);

        zmm1 = _mm512_maskz_permutex_pd(0xCC, zmm0, 0x40);
        zmm6 = _mm512_maskz_permutex_pd(0xCC, zmm5, 0x40);
        zmm10 = _mm512_add_pd(zmm10, zmm1);
        zmm11 = _mm512_add_pd(zmm11, zmm6);

        zmm1 = _mm512_maskz_permute_pd(0xCC, zmm1, 0x44);
        zmm6 = _mm512_maskz_permute_pd(0xCC, zmm6, 0x44);
        zmm10 = _mm512_add_pd(zmm10, zmm1);
        zmm11 = _mm512_add_pd(zmm11, zmm6);

        zmm_tmp1 = _mm512_maskz_permutexvar_pd(0xF0, idx, zmm10);
        zmm_tmp2 = _mm512_maskz_permutexvar_pd(0xF0, idx, zmm11);
        zmm10 = _mm512_add_pd(zmm10, zmm_tmp1);
        zmm11 = _mm512_add_pd(zmm11, zmm_tmp2);

        zmm_tmp1 = _mm512_add_pd(zmm10, zmm_acc);
        zmm_acc = _mm512_add_pd(zmm11, zmm_tmp1);
        zmm_acc = _mm512_permutexvar_pd(acc_idx, zmm_acc);

        zmm_tmp2 = _mm512_permutexvar_pd(acc_idx, zmm_tmp1);
        _mm512_storeu_pd(dst, zmm_tmp1);

        zmm11 = _mm512_add_pd(zmm11, zmm_tmp2);
        _mm512_storeu_pd(dst+NUM_ELES_IN_ZMM, zmm11);

        src += N_UNROLL;
        dst += N_UNROLL;
    }
}

```

Performance Evaluation

We evaluate the performance between the three versions of scan implementations on a single core of an Intel® Xeon® Platinum 8260L processor. The GCC (v9.3.0), Clang (v10.0.1), and ICC (Intel® C++ Compiler, v19.1.3.304) compilers were compared for the baseline, OpenMP SIMD, and explicit AVX-512 SIMD implementations. **Table 4** shows the assembly code generated by ICC for the main block of each implementation.

Baseline/SSE	OMP SIMD scan	Explicit AVX512 SIMD
<pre> ..B1.4: incq %r8 vaddsd (%rdi,%rsi), %xmm0,%xmm0 vmovsd %xmm0, (%rdi,%rdx) vaddsd 8(%rdi,%rsi), %xmm0,%xmm0 vmovsd %xmm0, 8(%rdi,%rdx) addq \$16,%rdi cmpq %rcx,%r8 jb ..B1.4 </pre>	<pre> ..B2.15: vmovups (%rsi,%r9,8),%zmm5 vmovups 64(%rsi,%r9,8),%zmm13 vbroadcastsd %xmm0,%zmm0 vpermpd %zmm5,%zmm4,%zmm6{%k3}{z} vpermpd %zmm13,%zmm4, %zmm14{%k3}{z} vaddpd %zmm6,%zmm5,%zmm7 vaddpd %zmm14,%zmm13,%zmm15 vpermpd %zmm7,%zmm3,%zmm8{%k2}{z} vpermpd %zmm15,%zmm3, %zmm16{%k2}{z} vaddpd %zmm8,%zmm7,%zmm9 vaddpd %zmm16,%zmm15,%zmm17 vpermpd %zmm9,%zmm2,%zmm10{%k1}{z} vpermpd %zmm17,%zmm2,%zmm18{%k1}{z} vaddpd %zmm10,%zmm9,%zmm11 vaddpd %zmm18,%zmm17,%zmm21 vaddpd %zmm11,%zmm0,%zmm12 vpermpd %zmm12,%zmm1,%zmm19 vmovupd %zmm12,(%rcx,%r9,8) vbroadcastsd %xmm19,%zmm20 vaddpd %zmm21,%zmm20,%zmm22 vpermpd %zmm22,%zmm1,%zmm0 vmovupd %zmm22,64(%rcx,%r9,8) addq \$16,%r9 cmpq %rdi,%r9 jb ..B2.15 </pre>	<pre> ..B2.3: vmovups (%rsi),%zmm5 addl \$16,%eax vmovups 64(%rsi),%zmm6 vpermpd \$64,%zmm5,%zmm9{%k2}{z} vpermpd \$64,%zmm6,%zmm10{%k2}{z} vpermilpd \$0,%zmm5,%zmm3{%k3}{z} addq \$128,%rsi vaddpd %zmm5,%zmm3,%zmm7 vaddpd %zmm7,%zmm9,%zmm12 vpermilpd \$0,%zmm6,%zmm4{%k3}{z} vpermilpd \$68,%zmm9,%zmm11{%k2}{z} vaddpd %zmm6,%zmm4,%zmm8 vaddpd %zmm12,%zmm11,%zmm18 vaddpd %zmm8,%zmm10,%zmm14 vpermpd %zmm18,%zmm1, %zmm17{%k1}{z} vpermilpd \$68,%zmm10,%zmm13{%k2}{z} vaddpd %zmm14,%zmm13,%zmm16 vaddpd %zmm18,%zmm17,%zmm19 vpermpd %zmm16,%zmm1, %zmm15{%k1}{z} vaddpd %zmm2,%zmm19,%zmm20 vaddpd %zmm16,%zmm15,%zmm22 vpermpd %zmm20,%zmm0,%zmm21 vmovups %zmm20,(%rdx) vaddpd %zmm22,%zmm21,%zmm23 vaddpd %zmm20,%zmm22,%zmm2 vmovups %zmm23,64(%rdx) vpermpd %zmm2,%zmm0,%zmm2 addq \$128,%rdx cml %edi,%eax jl ..B2.3 </pre>

Table 4. Assembly code generated by ICC for the three scan implementations

We observe that for the baseline code, the loop is unrolled by only two elements, and both scalar add operations form a dependency chain. This is inefficient because the generated code is not exploiting the wider AVX-512 execution units.

The OpenMP SIMD implementation is more interesting because the generated code uses AVX-512 add and permute operations and is unrolled by 16 elements. There are total of eight adds, eight permute, and two broadcast instructions to process 16 input elements. For a set of eight elements, the add and permutes form a dependency chain (highlighted in red). In addition, all the permutes generated in this version of the code shuffle elements within 256b lanes. Permutes that shuffle elements within a 128b lanes have lower latency (one cycle) than permutes that shuffle within 256b lanes (three cycles).

Our explicit AVX-512 SIMD implementation is also unrolled by 16 elements, and uses 11 add and 10 permute instructions to process those 16 elements. Even though we use greater number of add and permute instructions compared to the OpenMP SIMD scan, it has two advantages. First, four out of ten permutes are within a 128b lane (one-cycle latency), and hence are more efficient. These four permute instructions are highlighted in green. Second, some of the permutes are independent of the add instructions. This helps by providing more instruction-level parallelism and more effective utilization of AVX-512 execution units. We do an extra add to compute the accumulation result as soon as possible. This puts more concurrent instructions in flight and mitigates stalls in the execution pipeline for future iterations.

Figure 2 shows the performance for vector sizes ranging from 64 to 1,024 elements in steps of 32 and with all vectors residing in L1 cache. We can make the following observations from the performance data:

1. The explicit AVX-512 SIMD implementation outperforms both the baseline and OpenMP SIMD implementations.
2. GCC and Clang are unable to vectorize the scan computations. Their performance remains unchanged, even with the OpenMP SIMD directives.
3. ICC does a great job of auto-vectorization when OpenMP SIMD directives are used.
4. The average speed-up of the explicit SIMD scan implementation over the baseline and OpenMP SIMD scans is 4.6x (GCC and Clang) and 1.6x (ICC), respectively.

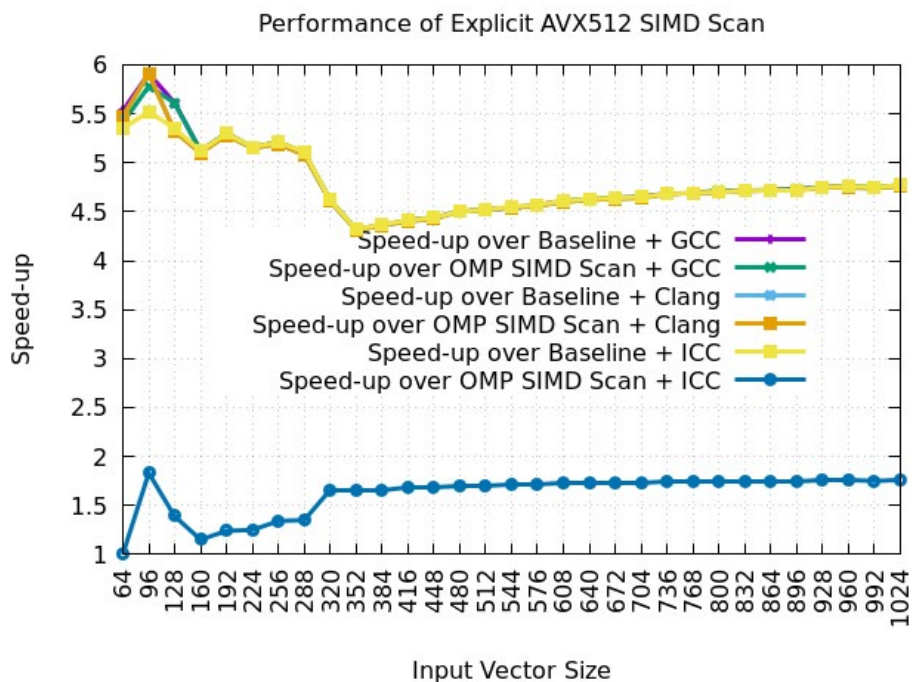


Figure 2. Performance comparison of explicit AVX-512 scan

Final Thoughts

We have briefly introduced explicit SIMD programming and applied it to vector scan computations. While optimizing compilers can often provide good performance, there may be cases, as demonstrated in this article, where there is room for improvement. Therefore, it is useful for developers to understand explicit SIMD programming to achieve maximum performance.

References

1. Guy E Blelloch (2018) [Prefix sums and their applications](#)
2. https://en.cppreference.com/w/cpp/algorithm/inclusive_scan
3. <https://numpy.org/doc/stable/reference/generated/numpy.cumsum.html>
4. <https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/pragmas/intel-specific-pragma-reference.html#intel-specific-pragma-reference>
5. <https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/details-about-intrinsics.html>
6. <https://www.openmp.org/spec-html/5.0/openmpsu42.html#x65-1400002.9.3.1>
7. <https://www.openmp.org/spec-html/5.0/openmpsu45.html#x68-1940002.9.6>

THE PARALLEL UNIVERSE

Intel technologies may require enabled hardware, software or service activation. Learn more at intel.com or from the OEM or retailer. Your costs and results may vary.

Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.

Optimization Notice: Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice Revision #20110804. <https://software.intel.com/en-us/articles/optimization-notice>

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors.

Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. See backup for configuration details. For more complete information about performance and benchmark results, visit www.intel.com/benchmarks.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details. No product or component can be absolutely secure.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.