Intel Software

THE PARALLEL UNIVERSE

10th Anniversary Issue

Optimizing the Performance of oneAPI Applications

Speeding Up Monte Carlo Simulation with oneAPI

Issue 40 2020

CONTENTS

Sign up for future issues

LETTER FROM THE EDITOR

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

The Parallel Universe Turns 10

Reflections on Where We're Going...and Where We've Been

This issue marks the 10th anniversary of *The Parallel Universe*—and my third year as editor. As the name implies, the magazine was originally conceived as a venue for articles related to parallel computing. Our core topic is still largely the same because parallelism is even more prevalent today than it was 10 years ago. However, the mix of topics has changed with the times. New vector instruction sets provide better instruction-level parallelism. With AVX-512, Intel CPUs are practically accelerators. We've also witnessed the rise of data analytics, so our editorial policy was expanded to include parallel frameworks (e.g., Apache Spark*) and achieving high performance with productivity languages (e.g., Python*, Julia*). The universe of parallel computing has also expanded to include heterogeneous parallelism.

As we discussed in our last issue, oneAPI is the next big thing in heterogeneous parallel computing. That's why the first three articles in this issue cover topics related to oneAPI. Our feature article, **GPU-Quicksort** walks you through a step-by-step translation of OpenCL™ to Data Parallel C++. This is followed by **Optimizing Performance of oneAPI Applications** and **Speeding Up Monte Carlo Simulation with oneAPI**. The former describes Intel's programming tools and provides an optimization case study for oneAPI applications. The latter shows how to offload Intel® Math Kernel Library functions to accelerators using Intel® oneMKL.

Next, Venkat Krishnamurthy and Kathryn Vandiver from OmniSci discuss the unification of data science and traditional analytics in **Bringing Accelerated Analytics at Scale to Intel® Architecture**. This is followed by **A New Approach to Parallel Computing Using Automatic Differentiation**, in which Dmitri Goloubentsev (Matlogica) and Evgeny Lakshtanov (University of Aviero) describe a tool to convert object-oriented, single-threaded scalar code into vectorized, multithreaded lambda functions.

In the first issue of *The Parallel Universe*, our founding editor, James Reinders, published **8 Rules for Parallel Programming for Multicore** that are still relevant and correct today so we're republishing them in honor of our 10th anniversary.

Finally, we close this issue with something we've never done before—a book review. Ruud van der Pas from Oracle Corporation was kind enough to review **The OpenMP Common Core: Making OpenMP Simple Again** by Timothy G. Mattson, Yun (Helen) He, and Alice E. Koniges. The original OpenMP specification (published in 1997) was a marvel of technical writing. It was concise (only 63 pages) and full of code examples to help you get started. The OpenMP specification has since grown to 666 pages because important new capabilities have been added to give programmers fine control of vectorization and thread placement, plus accelerator offload. However, the "common core" of OpenMP referred to in the book title is largely the same as it was 23 years ago.

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

Henry A. Gabb
April 2020

oneAPI DevCloud is Open for Coding 24/7

Developing code at home requires easy access to compute cycles and the latest software development tools that support unified programming across hardware architectures. Intel is here to help improve developers' work-from-home experience with the oneAPI DevCloud, which is open whenever you're ready to code.

Six Ways oneAPI DevCloud Now Makes #WFH Better

1. **Extended access.** All DevCloud users now automatically get free access through July 1, 2020.
2. **Increased capacity and expanded hardware availability.** Additional processor nodes and accelerators will provide access to the latest Intel CPUs, GPUs, and FPGAs.
3. **Updated software development tools.** The newest releases of Intel® oneAPI beta tools and libraries are pre-configured, including improvements to unified shared memory and more GPU library functions.
4. **New simplified access.** Improvements include immediate sign-up access; a new, modern IDE-like experience via JupyterLab*; and a simpler Secure Shell (SSH) client experience.
5. **Increased storage allocation.** Each user now receives 220 GB of file storage and 192 GB of RAM.
6. **Expanded Dev-to-Dev #WFH forum support.** Technical experts are available to provide tips and best practices for working from home.

SIGN UP NOW for free access. Already have an account? SIGN IN.

GPU-QUICKSORT*

How to Move from OpenCL™ to Data Parallel C++

Robert Ioffe, Senior Exascale Performance Software Engineer, Intel Corporation

Data Parallel C++ (DPC++) is a heterogeneous, portable programming language based on the Khronos SYCL* standard. This single-source programming language can target an array of platforms: CPUs, integrated and discrete GPUs, FPGAs, and other accelerators. To give you an idea of what DPC++ can do, we'll port a non-trivial OpenCL™ application, GPU-Quicksort*, to DPC++ and document the experience. Our goal will be to exceed the capabilities of the initial application. OpenCL™ C makes it very hard to write generic algorithms, and it becomes clear that it's a serious shortcoming when you try to implement algorithms—like sorting—that need to work for different data types. The original GPU-Quicksort for OpenCL™ was written to sort unsigned integers. We'll demonstrate how to use templates with DPC++ and implement GPU-Quicksort for multiple data types. Finally, we'll port GPU-Quicksort to Windows* and Ubuntu* 18.04 to show DPC++ portability.

What's GPU-Quicksort?

GPU-Quicksort is a high-performance sorting algorithm designed specifically for highly parallel, multicore graphics processors. It was invented in 2009 by Daniel Cederman and Phillippas Tsigas, a student and professor from the Chalmers University of Technology in Sweden. Originally implemented in CUDA*, it was reimplemented in 2014 in OpenCL™ 1.2 and OpenCL™ 2.0 by me to demonstrate high performance on Intel® Integrated Processor Graphics and showcase nested parallelism and work-group scan functions in OpenCL™ 2.0 and fully implemented in Intel OpenCL™ drivers. We'll port an OpenCL™ 1.2 implementation of the GPU-Quicksort to DPC++ and make the implementation generic so that it can sort not just unsigned integers, but also floats and doubles.

What's OpenCL™?

We'll start with the OpenCL™ 1.2 implementation. Intel fully supports OpenCL™, a Khronos standard for programming heterogeneous parallel systems, on a variety of operating systems and platforms. OpenCL™ consists of:

- The runtime
- The host API
- The device C-based programming language OpenCL™ C

Here lie both its power and its limitations. The power is the ability to write high-performance, portable, heterogeneous parallel applications. Its main limitation is the necessity to write and debug two separate codes—the host side and the device side—as well as the lack of templates and other C++ features modern programmers are accustomed to, which makes writing generic libraries in OpenCL™ hard.

What's Data Parallel C++?

DPC++ is an Intel implementation of Khronos SYCL* with extensions. The SYCL standard designed to address the OpenCL™ limitations outlined above. DPC++ provides:

- **A single-source programming model**, which consists of a single code base for both host and device programming
- **The full use of C++ templates and template metaprogramming** on the device with minimal impact on performance without compromising portability

DPC++ lets a programmer target CPUs, GPUs, and FPGAs while permitting accelerator-specific tuning—a definite improvement over OpenCL™. It's also supported by Intel® software tools like **Intel® VTune™ Profiler** and **Intel® Advisor**, as well as by GDB*. We'll make full use of DPC++, especially its template features.

The Starting Point: Windows* Apps from 2014

We'll start with GPU-Quicksort for OpenCL™ 1.2 (as described in the article **GPU-Quicksort in OpenCL™ 2.0: Nested Parallelism and Work-Group Scan Functions**). The original application was written for Windows, so we port it to Ubuntu 18.04 by adding the cross-platform code to measure time and use `aligned_alloc/free` for aligned memory allocation/deallocation, as opposed to `_alligned_malloc/_aligned_free` on Windows.

Let's get a brief overview of GPU-Quicksort architecture. It consists of two kernels:

1. `gqsort_kernel`
2. `lqsort_kernel`

Written in OpenCL™ 1.2, these are glued together by a dispatcher code, which iteratively calls `gqsort_kernel` until the input is split into small enough chunks, which can be fully sorted by `lqsort_kernel`. The application allows the user to select:

- **The number of times** to run sort for measurement purposes
- **The vendor and device** on which to run the kernels
- **The size** of the input
- **Whether to show** device details

The application follows a typical OpenCL™ architecture of supporting utilities for initializing OpenCL™ platforms and devices and building code for them. A separate file, with the OpenCL™ kernels and their supporting functions and the main application that accepts user arguments, initializes the platform and device, builds the kernels, properly allocates memory, and creates buffers from it, and then binds them to the kernel arguments and launches the dispatcher function.

Data Parallel C++/OpenCL™ Interoperability: Platform Initialization

First, install the **Intel® oneAPI Base Toolkit**, which includes the **Intel® oneAPI DPC++ Compiler**. We start our port to DPC++ by including the `CL/sycl.hpp` header and, to spare us the verbosity of DPC++, using the namespace `cl::sycl` clause:

```
#include <CL/sycl.hpp>
...
using namespace cl::sycl;
```

Now, instead of initializing a platform, a device, a context, and a queue the OpenCL™ way, we do it the concise DPC++ way:

```
device d(default_selector);
  if (d.is_host()) {
    // This platform has no OpenCL devices
    return;
  }

  queue queue(default_selector, [](cl::sycl::exception_list l) {
    for (auto ep : l) {
      try {
        std::rethrow_exception(ep);
      } catch (cl::sycl::exception e) {
        std::cout << e.what() << std::endl;
      }
    }
  });
```

We also need to retrieve the underlying OpenCL™ context, device, and queue, since the rest of the application is OpenCL™ based:

```
pOCL->contextHdl = queue.get_context().get();
pOCL->deviceID = queue.get_device().get();
pOCL->cmdQHdl = queue.get();
```

That's our first iteration: configure and compile it with the Intel DPC++ Compiler and run it.

Data Parallel C++: How to Select an Intel GPU

The shortcoming of the first iteration is that it always selects the default device, which may or may not be an Intel GPU. To specify an Intel GPU, we need to write a custom device selector:

```
class intel_gpu_selector : public device_selector {
 public:
  intel_gpu_selector() : device_selector() {}

  int operator()(const device& device) const override {
    if (device.get_info<info::device::name>().find("Intel") != std::string::npos) {
      if (device.get_info<info::device::device_type>() ==
          info::device_type::gpu) {
        return 50;
      }
    }
    /* Never choose device with a negative score */
    return -1;
  }
};
```

We use `intel_gpu_selector` to select an Intel GPU when the user asks for it:

```
auto get_queue = [&pDeviceStr, &pVendorStr]() {
   device_selector* pds;
   if (pVendorStr == std::string("intel")) {
     if (pDeviceStr == std::string("gpu")) {
         static intel_gpu_selector selector;
     pds = &selector;
   } else if (pDeviceStr == std::string("cpu")) {
     static cpu_selector selector;
     pds = &selector;
   }
 } else {
   static default_selector selector;
   pds = &selector;
 }
   device d(*pds);

   queue queue(*pds, [](cl::sycl::exception_list l) {
     for (auto ep : l) {
       try {
         std::rethrow_exception(ep);
       } catch (cl::sycl::exception e) {
         std::cout << e.what() << std::endl;
       }
     }
   });
   return queue;
 };

 auto queue = get_queue();
```

Data Parallel C++: How to Set Kernel Arguments and Launch Kernels

The third iteration of our code uses DPC++ to set kernel arguments and launch kernels. The program is still built, and the kernels are obtained, in the OpenCL™ way. We use `cl::sycl::kernel` objects to wrap original OpenCL™ kernels. For example:

```
cl::sycl::kernel sycl_gqsort_kernel(gqsort_kernel, pOCL->queue.get_context());
```

We replace a number of `clSetKernelArg` methods with `set_arg` DPC++ methods and `clEnqueueNDRange` calls with `parallel_for` calls. This example below shows `gqsort_kernel`, but a `lqsort_kernel` upgrade is very similar:

```
  pOCL->queue.submit([&](handler& cgh) {
        cgh.set_arg(0, db);
        cgh.set_arg(1, dnb);
        cgh.set_arg(2, blocksb);
        cgh.set_arg(3, parentsb);
        cgh.set_arg(4, newsb);

        cgh.parallel_for(nd_range<1>(range<1>(GQSORT_LOCAL_WORKGROUP_SIZE *
 (blocks.size())),
                                    range<1>(GQSORT_LOCAL_WORKGROUP_SIZE)),
        sycl_gqsort_kernel);
     });
     pOCL->queue.wait_and_throw();
```

Here's a less verbose style to set all the arguments of the kernel with one `set_args` call:

```
cgh.set_args(db, dnb, blocksb, parentsb, newsb);
```

We can also use a less verbose version of the `parallel_for`:

```
cgh.parallel_for(nd_range<>(GQSORT_LOCAL_WORKGROUP_SIZE * blocks.size(),
                            GQSORT_LOCAL_WORKGROUP_SIZE),
```

Data Parallel C++: How to Create Buffers and Set the Access Mode

We convert OpenCL™ buffers to DPC++ buffers. (The first two are wrapping the memory that was align-allocated and passed into the function by reference. The other three are created from an STL vector.) We use the template keyword in front of the `get_access` member function for buffers that we pass by reference. Note the different access modes for various buffers, depending on whether we need read- or write-access, or both. We don't directly pass buffers as kernel arguments; we pass the accessors to them:

```
    buffer<T>  d_buffer(d, range<>(size), {property::buffer::use_host_ptr()});
    buffer<T>  dn_buffer(dn, range<>(size), {property::buffer::use_host_ptr()});
…
    buffer<block_record>  blocks_buffer(blocks.data(), blocks.size(),
{property::buffer::use_host_ptr()});
    buffer<parent_record>  parents_buffer(parents.data(), parents.size(),
{property::buffer::use_host_ptr()});
    buffer<work_record>  news_buffer(news.data(), news.size(),
{property::buffer::use_host_ptr()});

    pOCL->queue.submit([&](handler& cgh) {
      auto db = d_buffer.template get_access<access::mode::discard_read_write>(cgh);
      auto dnb = dn_buffer.template get_access<access::mode::discard_read_write>(cgh);
      auto blocksb = blocks_buffer.get_access<access::mode::read>(cgh);
      auto parentsb = parents_buffer.get_access<access::mode::read>(cgh);
      auto newsb = news_buffer.get_access<access::mode::write>(cgh);

      cgh.set_args(db, dnb, blocksb, parentsb, newsb);

      cgh.parallel_for(nd_range<>(GQSORT_LOCAL_WORKGROUP_SIZE * blocks.size(),
                                  GQSORT_LOCAL_WORKGROUP_SIZE),
        sycl_gqsort_kernel);
    });
    pOCL->queue.wait_and_throw();
```

Data Parallel C++: How to Query Platform and Device Properties

In OpenCL™, we used the methods `clGetPlatformInfo` and `clGetDeviceInfo` to query various platform and device properties. Now we can use `get_info<>` methods to query the same information. For example:

```
auto max_work_item_dimensions = q.get_device().get_info<info::device::max_work_item_dimensions>();
std::cout << "CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS: " << max_work_item_dimensions << std::endl;
```

or query properties with a more complex structure:

```
auto max_work_item_sizes = q.get_device().get_info<info::device::max_work_item_sizes>();
printf ("CL_DEVICE_MAX_WORK_ITEM_SIZES    :     (%5zu, %5zu, %5zu)\n",
                    max_work_item_sizes[0],max_work_item_sizes[1], max_work_item_sizes[2]);
```

Porting OpenCL™ Kernels to Data Parallel C++, Part 1: `gqsort_kernel`

So far, we've initialized the platform and the device, created the buffers and their accessors and bound them to the kernels, and launched those kernels on the device in a DPC++ way. But we still need to create the kernels in an OpenCL™ way. We use OpenCL™ C and `clBuildProgram/clCreateKernel` APIs to build the program and create kernels. The OpenCL™ C kernels are stored in a separate file that's loaded into the program at runtime before being built. We'll change that, starting with the `gqsort_kernel`, the simpler of the two kernels.

The DPC++ way of creating kernels is via lambdas or functors. The use of lambdas for kernel creation is typically reserved for smaller kernels. When you have a more complex kernel that uses supporting functions, it's a good idea to create a functor class. We're going to create a `gqsort_kernel_class` functor and make it templated right from the start so that we can sort more than one datatype in the future.

```
template <class T>
class gqsort_kernel_class {
…
};
```

A typical functor class will have a void `operator()` that will take as a parameter an iteration id (in our case, a one-dimensional `nd_item<1> id`). The body of the kernel will reside in the `void operator()`. The functor will also have a constructor that will take global and local accessors, the equivalent of global and local memory pointers for an OpenCL™ kernel. The typical DPC++ functor will have a preamble, with using clauses defining various global and local accessor types. In the case of `gqsort_kernel`, it will look like this:

```
using blocks_read_accessor = accessor<block_record<T>, 1, access::mode::read,
access::target::global_buffer>;
using parents_read_write_accessor = accessor<parent_record, 1, access::mode::read_write,
access::target::global_buffer>;
using news_write_accessor = accessor<work_record<T>, 1, access::mode::write,
access::target::global_buffer>;
using discard_read_write_accessor =
      accessor<T, 1, access::mode::discard_read_write, access::target::global_buffer>;
using local_read_write_accessor = accessor<uint, 1, access::mode::read_write,
access::target::local>;
```

The private section of the functor will contain all the global and local accessors used within the body of the `void operator()`. In our case, it will look like this, with the first five accessors to global buffers and the rest to the local buffers:

```
  private:
      discard_read_write_accessor d, dn;
      blocks_read_accessor blocks;
      parents_read_write_accessor parents;
      news_write_accessor news;
      local_read_write_accessor lt, gt, ltsum, gtsum, lbeg, gbeg;
```

`gqsort_kernel` is a complex kernel that uses supporting structs and two supporting functions: `plus_prescan` and `median`, which, in turn, use specialized OpenCL™ functions and extensively use local memory arrays and variables, local and global barriers, and atomics. All these elements must be translated into DPC++.

Let's start with the functions. We omit structs, since they're trivially templatized. The `plus_prescan` function that's used to calculate scan sums is relatively simple, so the only change we'll make to bring it to DPC++ is to make it a template function in preparation of making our sort generic:

```
    template <class T>
void plus_prescan(T *a, T *b) {
    T av = *a;
    T bv = *b;
    *a = bv;
    *b = bv + av;
}
```

The `median` function is next. We not only need to make it a template function, we also need to replace the OpenCL™ C `select` function with the DPC++ `cl::sycl::select` function and rename it `median_select` to differentiate it from a similar host function:

```
template <class T>
T median_select(T x1, T x2, T x3) {
    if (x1 < x2) {
        if (x2 < x3) {
            return x2;
        } else {
            return cl::sycl::select(x1, x3, (uint)(x1 < x3));
        }
    } else {
        if (x1 < x3) {
            return x1;
        } else {
            return cl::sycl::select(x2, x3,(uint)(x2 < x3));
        }
    }
}
```

In OpenCL™ C, it's possible to both create local memory variables and arrays inside the body of the kernel and pass them as kernel parameters. But in DPC++, when using functors, we pass local buffer accessors when constructing the functor. In our case, all local memory variables and arrays will hold unsigned integers, so we'll create a special `local_read_write_accessor` type:

```
using local_read_write_accessor = accessor<uint, 1,
access::mode::read_write, access::target::local>;
```

We declare all the local memory variables:

```
local_read_write_accessor
    lt(range<>(GQSORT_LOCAL_WORKGROUP_SIZE+1), cgh),
gt(range<>(GQSORT_LOCAL_WORKGROUP_SIZE+1), cgh),
    ltsum(range<>(1), cgh), gtsum(range<>(1), cgh), lbeg(range<>(1), cgh), gbeg(range<>(1),
cgh);
```

We then pass them as parameters, along with global buffer accessors, to our functor constructor. Then the resulting object is passed to the `parallel_for`:

```
auto gqsort = gqsort_kernel_class<T>(db, dnb, blocksb, parentsb, newsb, lt, gt, ltsum,
gtsum, lbeg, gbeg);

      cgh.parallel_for(
        nd_range<>(GQSORT_LOCAL_WORKGROUP_SIZE * blocks.size(),
                   GQSORT_LOCAL_WORKGROUP_SIZE),
        gqsort);
```

Here, DPC++ lacks simplicity compared to OpenCL™ C. Next, `get_group_id` and `get_local_id` functions become:

```
const size_t blockid = id.get_group(0);
const size_t localid = id.get_local_id(0);
```

Local barriers go from:

```
barrier(CLK_LOCAL_MEM_FENCE);
```

to:

```
id.barrier(access::fence_space::local_space);
```

Global and local barriers go from:

```
barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
```

to:

```
id.barrier(access::fence_space::global_and_local);
```

For atomic operations, DPC++ is not as elegant as OpenCL™ C. So, what was concise:

```
psstart = &pparent->sstart;
psend = &pparent->send;
lbeg = atomic_add(psstart, ltsum);
gbeg = atomic_sub(psend, gtsum) - gtsum;
```

becomes unwieldy:

```
          cl::sycl::atomic<uint> psstart_a(multi_ptr<uint,
access::address_space::global_space>(&pparent.sstart));
          cl::sycl::atomic<uint> psend_a(multi_ptr<uint,
access::address_space::global_space>(&pparent.send));
          lbeg[0] = cl::sycl::atomic_fetch_add(psstart_a, ltsum[0]);
          gbeg[0] = cl::sycl::atomic_fetch_sub(psend_a, gtsum[0]) - gtsum[0];
```

Note the creation of `cl::sycl::atomic<>` variables prior to the use of DPC++ atomic operations, which cannot operate on the global or local memory pointers directly.

So far, we've translated and templatized supporting structs and functions, converting specialized OpenCL™ C functions to DPC++. We've also created a template functor class with local accessors and translated barriers and atomics.

Porting OpenCL™ Kernels to Data Parallel C++, Part 2: `lqsort_kernel`

Translation of `lqsort_kernel` follows the familiar patterns outlined by the translation of `gqsort_kernel`: create a `lqsort_kernel_class` functor and then translate local memory arrays and variables and barriers (no atomics here). `lqsort_kernel` also uses supporting functions and structs. In addition to `plus_prescan` and `median_select` used by `gqsort_kernel`, we have `bitonic_sort` and `sort_threshold` that are considerably more complex and specific to `lqsort_kernel`. After translation, they become the member functions of the `lqsort_kernel_class`. Their signatures change due to barrier use which, in the case of DPC++, requires the iteration objects. They work on local and global memory pointers, which require special handling so the OpenCL™ C signature:

```
void bitonic_sort(local uint* sh_data, const uint localid)
```

becomes:

```
void bitonic_sort(local_ptr<T> sh_data, const uint localid, nd_item<1> id)
```

and:

```
void sort_threshold(local uint* data_in, global uint* data_out,
                     uint start,  uint end,
                    local uint* temp, uint localid)
```

becomes:

```
void sort_threshold(local_ptr<T> data_in, global_ptr<T> data_out,
              uint start, uint end,
              local_ptr<T> temp_, uint localid,
              nd_item<1> id)
```

These functions are translated similarly to `gqsort_kernel`, with the `UINT_MAX` macro being replaced with `std::numeric_limits<T>::max()` to handle various data types in the future.

When translating the `lqsort_kernel`, pointers to local memory (e.g., `local uint* sn;`) are replaced with `local_ptr<>` objects (e.g., `local_ptr<T> sn;`). To retrieve the local pointer from the local accessor, we call the `get_pointer` member function of the accessor:

```
sn = mys.get_pointer();
```

`local_ptr<>` and `global_ptr<>` objects work with pointer arithmetic, so what previously was `d + d_offset`, where `d` was a global pointer, becomes:

```
d.get_pointer() + d_offset
```

We translate local memory variables as accessors of size 1, meaning array accesses at index 0 (e.g., `gtsum[0]`). When we complete the `lqsort_kernel` translation, we fully transition to DPC++, but still sort unsigned integers. We did all the prework of templatizing supporting structs and functions and the functor classes of the two main kernels—and will enjoy the benefits.

The Power of Data Parallel C++: Templates...And Their Caveats

The real power of DPC++ is the ability to use C++ templates, which enable writing generic code. We want our GPU-Quicksort to be generic and to be able to sort not only unsigned integers, but also other basic data types (e.g., floats and doubles). In addition to the `UINT_MAX` to `std::numeric_limits<T>::max()` change mentioned above, we need additional modification of the `median_select` function. `cl::sycl::select` takes a different type of the third argument, depending on the size of the type of the first two arguments, so we introduce the `select_type_selector` type traits class:

```
template <class T> struct select_type_selector;

template <> struct select_type_selector<uint>
{
  typedef uint data_t;
};

template <> struct select_type_selector<float>
{
  typedef uint data_t;
};

template <> struct select_type_selector<double>
{
  typedef ulong data_t;
};
```

It allows us to convert a Boolean comparison to an appropriate type required by `cl::sycl::select`; `median_select` becomes:

```
template <class T>
T median_select(T x1, T x2, T x3) {
  if (x1 < x2) {
    if (x2 < x3) {
      return x2;
    } else {
      return cl::sycl::select(x1, x3, typename select_type_selector<T>::data_t(x1 < x3));
    }
  } else {
    if (x1 < x3) {
      return x1;
    } else {
      return cl::sycl::select(x2, x3, typename select_type_selector<T>::data_t(x2 < x3));
    }
  }
}
```

To handle additional types, we need more specializations of `select_type_selector`. Now `GPUQSort` can sort floats and doubles on the GPU.

Back to Windows...And RHEL*

To demonstrate DPC++ portability, we port the code to Windows and RHEL*. The RHEL port is minimal: we add the Intel imf math library at link time. Windows porting is slightly more complex. Add the following definitions when compiling:

```
-D_CRT_SECURE_NO_WARNINGS -D_MT=1
```

Accounting for the fact that `cl::sycl::select` for doubles requires `unsigned long long` type as the third parameter (as opposed to `unsigned long` on Linux), `select_type_selector` for doubles becomes:

```
template <> struct select_type_selector<double>
{
#ifdef _MSC_VER
  typedef ulonglong data_t;
#else
  typedef ulong data_t;
#endif
};
```

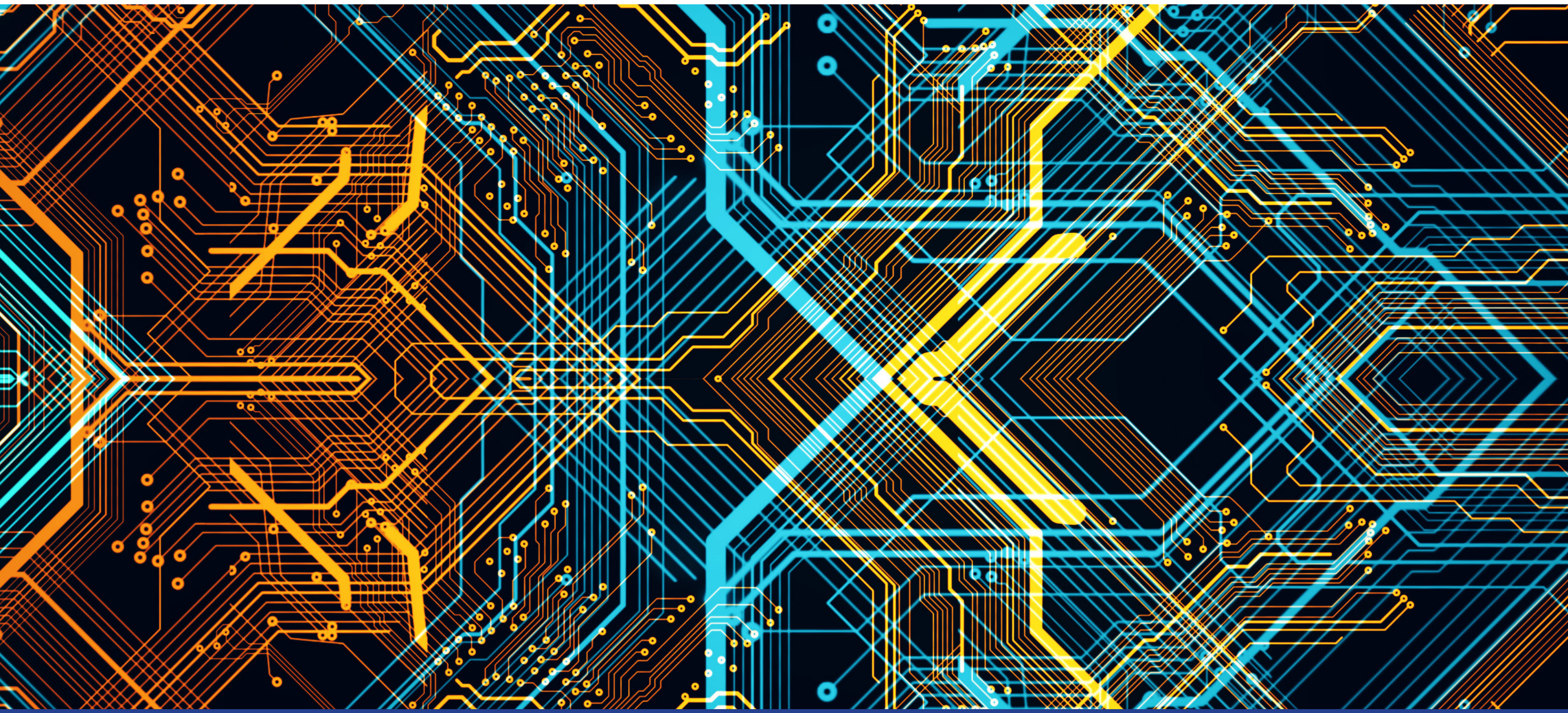
On Windows, we undefine `max` and `min` to prevent the macro definitions from colliding with `std::min` and `std::max`. That's all there is to it. We can sort unsigned integers, floats, and doubles using Intel GPUs on Windows and two Linux flavors.

Get Started Now

We gradually translated GPU-Quicksort from its original OpenCL™ 1.2 into DPC++. At every step along the way, we had a working application. So, when you're considering bringing DPC++ to your workflow, start small and either add on or fully transition to DPC++ as time allows. Easily mix OpenCL™ and DPC++ in your code base and enjoy the benefits of both. Use legacy OpenCL™ kernels in their original form and enjoy the full power of C++ templates, classes, and lambdas when you're developing new code in DPC++. Easily port code between Windows and various Linux flavors and choose which platform to develop on. You also have the full power of Intel tools to help you debug, profile, and analyze your DPC++ program.

Resources

- **Khronos OpenCL™**, the open standard for parallel programming of heterogeneous systems
- **Khronos SYCL**, C++ single-source heterogeneous programming for OpenCL™
- **GPU-Quicksort: A practical Quicksort Algorithm for Graphics Processors** by Daniel Cederman and Philippas Tsigas
- **GPU-Quicksort in OpenCL™ 2.0: Nested Parallelism and Work-Group Scan Functions** by Robert Ioffe
- **Intel® oneAPI Toolkits**
- **Accompanying code for this article**



OPTIMIZING THE PERFORMANCE OF ONEAPI APPLICATIONS

Getting the Most from this Unified, Standards-Based Programming Model

Kevin O'Leary, Software Technical Consulting Engineer, Intel Corporation

Modern workloads are incredibly diverse—and so are processor architectures. No single architecture is best for every workload. Maximizing performance takes a mix of scalar, vector, matrix, and spatial (SVMS) architectures deployed in CPU, GPU, FPGA, and future accelerators. Intel® oneAPI products will deliver what you need to deploy your applications across SVMS. This set of complementary toolkits—a base kit and specialty add-ons—simplifies programming and helps you improve efficiency and innovation.

The **Intel® oneAPI Base Toolkit (beta)** includes advanced analysis and debug tools for profiling, design advice, and debugging:

- **Intel® VTune™ Profiler (beta)** finds performance bottlenecks in CPU, GPU, and FPGA systems.
- **Intel® Advisor (beta)** provides vectorization, threading, and accelerator offload advice.
- **Intel-enhanced GDB* (beta)** helps efficiently debug code.

Performance Analysis Tools

This article will focus on Intel Advisor (beta) and Intel VTune Profiler (beta) and the new features they provide as part of the Intel oneAPI Base Toolkit (beta).

Intel® Advisor (Beta)

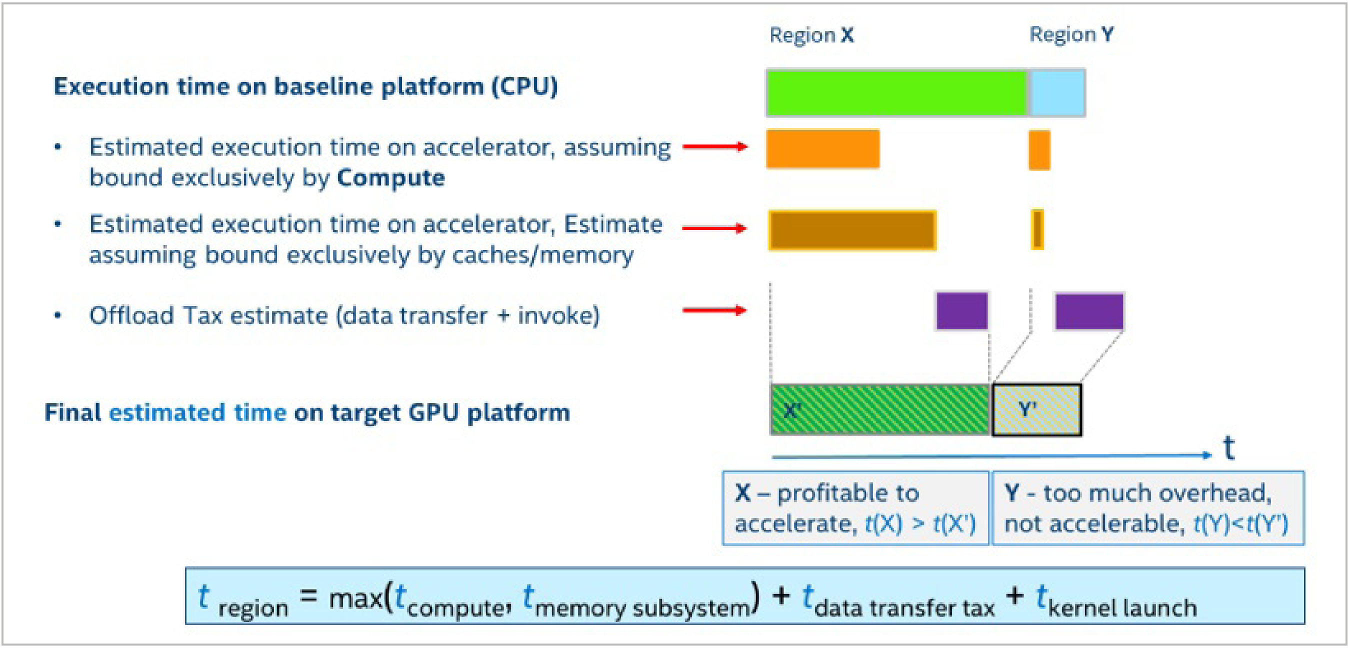
Intel Advisor (beta) is an extended version of Intel Advisor, a tool for code modernization, programming guidance, and performance estimation that supports the **DPC++ language** on CPUs and GPUs. It provides codesign, performance modeling, analysis, and characterization features for C, C++, Fortran*, and mixed Python* applications.

Intel Advisor (beta) includes:

- **Offload Advisor** to help you identify high-impact opportunities to offload to the GPU as well as areas that aren't useful to offload. You can also project performance speedup on accelerators, estimate offload overhead, and pinpoint accelerator performance bottlenecks.
- **Vectorization Advisor** to help you identify high-impact, under-optimized loops and see what's blocking vectorization and where it's safe to force vectorization.
- **Threading Advisor** to help you analyze, design, tune, and check threading design options without disrupting your normal development.
- **Roofline Analysis** to help you visualize performance on both your CPU and GPU and see how close you are to the maximum possible performance.
- **Intel® FPGA Add-On for oneAPI Base Toolkit (beta)** (optional) to help you program these reconfigurable hardware accelerators to speed specialized, data-centric workloads. (Requires installation of the Intel oneAPI Base Toolkit.)

Use the Offload Advisor command-line feature to design code for efficient offloading to accelerators—even before you have hardware. Estimate code performance and compare it with data transfer costs. No recompilation is required.

The Intel Advisor (beta) GPU performance evaluation (**Figure 1**) produces upper-bound speedup estimates using a bounds and bottlenecks performance model. It takes measured x86 CPU metrics and application characterization as input and applies an analytical model to estimate execution time and characteristics on a target GPU.


Region X
Region Y
Execution time on baseline platform (CPU)
Estimated execution time on accelerator, assuming bound exclusively by Compute
Estimated execution time on accelerator, Estimate assuming bound exclusively by caches/memory
Offload Tax estimate (data transfer + invoke)
Final estimated time on target GPU platform
X'
Y'
t
X – profitable to accelerate, t(X) > t(X')
Y - too much overhead, not accelerable, t(Y)<t(Y')

$$t_{\text{region}} = \max(t_{\text{compute}}, t_{\text{memory subsystem}}) + t_{\text{data transfer tax}} + t_{\text{kernel launch}}$$

1 **Intel Advisor (beta) GPU performance evaluation**

The Roofline Analysis feature helps you optimize your CPU or GPU code for compute and memory. Locate bottlenecks and determine performance headroom for each loop or kernel to prioritize which optimizations will deliver the highest performance payoff. (Note that GPU Roofline Analysis is in technical preview.)

Intel® VTune™ Profiler (Beta)

Intel VTune Profiler (beta) is a performance analysis tool for serial and multithreaded applications. It helps you analyze algorithm choices and identify where and how your application can benefit from available hardware resources. Use it to locate or determine:

- **The most time-consuming (hot) functions** in your application and/or on the whole system
- **Sections of code that don't effectively utilize** available processor resources
- **The best sections of code to optimize** for both sequential and threaded performance
- **Synchronization objects** that affect the application performance
- **Whether, where, and why your application spends time** on input/output operations
- **Whether your application is CPU- or GPU-bound** and how effectively it offloads code to the GPU
- **The performance impact** of different synchronization methods, different numbers of threads, or different algorithms
- **Thread activity** and transitions
- **Hardware-related issues** in your code such as data sharing, cache misses, branch misprediction, and others

The tool also has new features to support GPU analysis:

- **VTune GPU Offload Analysis** (technical preview)
- **GPU Compute/Media Hotspots Analysis** (technical preview)

GPU Offload Analysis (Preview)

Use this tool to analyze code execution on the CPU and GPU cores of your platform, correlate CPU and GPU activity, and identify whether your application is GPU- or CPU-bound. The tool infrastructure automatically aligns clocks across all cores in the system so you can analyze some CPU-based workloads together with GPU-based workloads within a unified time domain. This analysis lets you:

- **Identify how effectively** your application uses DPC++ or OpenCL™ kernels.
- **Analyze execution** of Intel® Media SDK tasks over time (for Linux targets only).
- **Explore GPU usage** and analyze a software queue for GPU engines at each moment of time.

GPU Compute/Media Hotspots Analysis (Preview)

Use this tool to analyze the most time-consuming GPU kernels, characterize GPU usage based on GPU hardware metrics, identify performance issues caused by memory latency or inefficient kernel algorithms, and analyze GPU instruction frequency for certain instruction types. The GPU Compute/Media Hotspots analysis allows you to:

- **Explore GPU kernels with high GPU utilization**, estimate the efficiency of this utilization, and identify possible reasons for stalls or low occupancy.
- **Explore the performance of your application** per selected GPU metrics over time.
- **Analyze the hottest DPC++ or OpenCL™ kernels** for inefficient kernel code algorithms or incorrect work item configuration.

Case Study: Using Software Tools to Optimize oneAPI Applications

Now that we know some of the tools and features available in the Intel oneAPI Base Toolkit, let's try working through an example. In this case study, we look at the usage of Intel VTune Profiler (beta) and Intel Advisor (beta) to optimize an application. We'll look at several Intel Advisor (beta) features including Roofline Analysis and Offload Advisor to determine the bottlenecks in an application and the regions to offload to an accelerator.

Matrix multiplication is a common operation in many applications. Here's a sample matrix multiplication kernel:

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

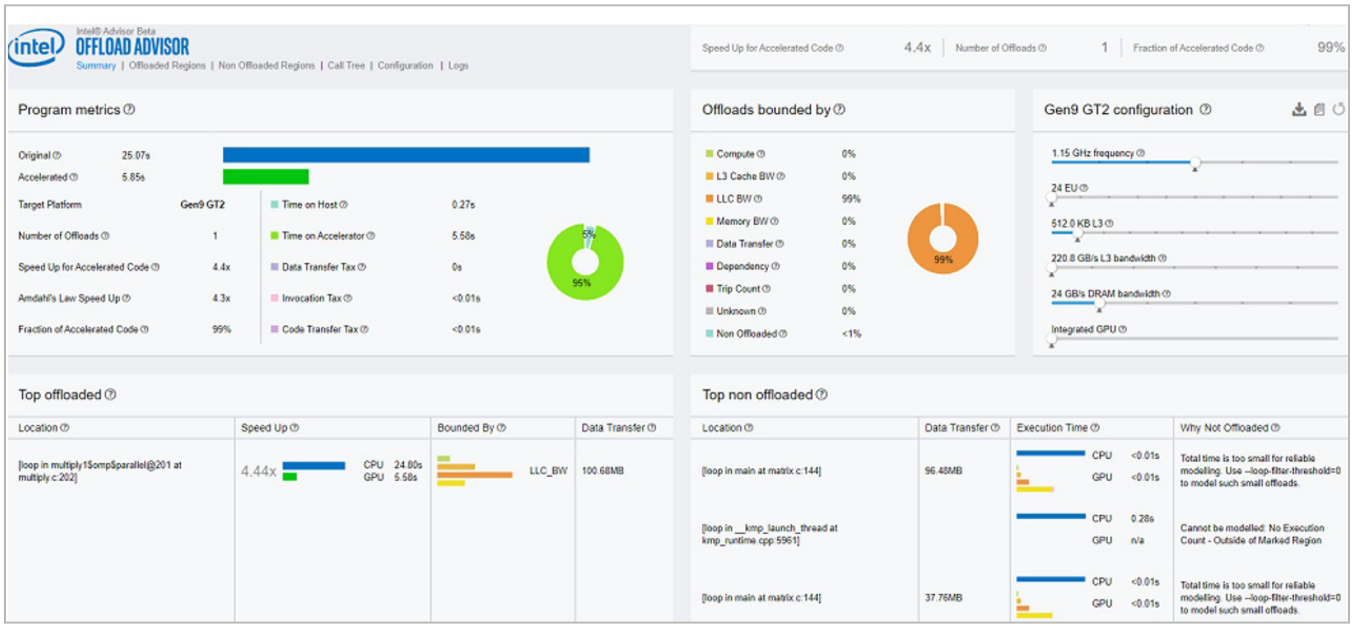
The algorithm is a triply nested loop with a multiplication and addition in each iteration. Code like this is computationally intensive with many memory accesses. Intel Advisor is ideal for helping you analyze it.

Using Intel Advisor (Beta) to Help Port to GPU

Intel Advisor (beta) has a feature that lets you see the portions of the code that can profitably be offloaded to a GPU. It also can predict the code performance when run on the GPU and lets you experiment based on several criteria. Analyzing DPC++ code with Intel Advisor (beta) requires a two-stage analysis:

```
$ source $ADVISOR_INSTALL/env/vars.sh

$ advixe-python $APM/collect.py --config gen9 advisor_project
/home/matrix

$ advixe-python $APM/analyze.py --config gen9 advisor_project --out-
dir /home/test/analyze
```

The screenshot in **Figure 2** shows the CPU run time and the predicted time when run on the accelerator (in this case, a GPU). It shows how many regions were offloaded and the net speedup. You can also see what the offloads are bounded by. In our case, we are 99% bounded by the last-level cache bandwidth (LLC BW).


Intel® Advisor Beta
OFFLOAD ADVISOR
Summary | Offloaded Regions | Non Offloaded Regions | Call Tree | Configuration | Logs

Speed Up for Accelerated Code	Number of Offloads	Fraction of Accelerated Code
4.4x	1	99%

Program metrics

Original	25.07s
Accelerated	5.85s
Target Platform	Gen9 GT2
Number of Offloads	1
Speed Up for Accelerated Code	4.4x
Amdahl's Law Speed Up	4.3x
Fraction of Accelerated Code	99%

Time on Host	0.27s
Time on Accelerator	5.58s
Data Transfer Tax	0s
Invocation Tax	<0.01s
Code Transfer Tax	<0.01s

Offloads bounded by

Compute	0%
L3 Cache BW	0%
LLC BW	99%
Memory BW	0%
Data Transfer	0%
Dependency	0%
Trip Count	0%
Unknown	0%
Non Offloaded	<1%

Gen9 GT2 configuration

1.15 GHz frequency
24 EU
512.0 KB L3
220.8 GB/s L3 bandwidth
24 GB/s DRAM bandwidth
Integrated GPU

Top offloaded

Location	Speed Up	Bounded By	Data Transfer
[loop in multiply1$omp$parallel@201 at multiply.c:202]	4.44x CPU 24.80s GPU 5.58s	LLC_BW	100.68MB

Top non offloaded

Location	Data Transfer	Execution Time	Why Not Offloaded
[loop in main at matrix.c:144]	96.48MB	CPU <0.01s GPU <0.01s	Total time is too small for reliable modelling. Use --loop-filter-threshold=0 to model such small offloads.
[loop in __kmp_launch_thread at kmp_runtime.cpp:5961]		CPU 0.28s GPU n/a	Cannot be modelled: No Execution Count - Outside of Marked Region
[loop in main at matrix.c:144]	37.76MB	CPU <0.01s GPU <0.01s	Total time is too small for reliable modelling. Use --loop-filter-threshold=0 to model such small offloads.

2 **CPU run time and predicted GPU run time**

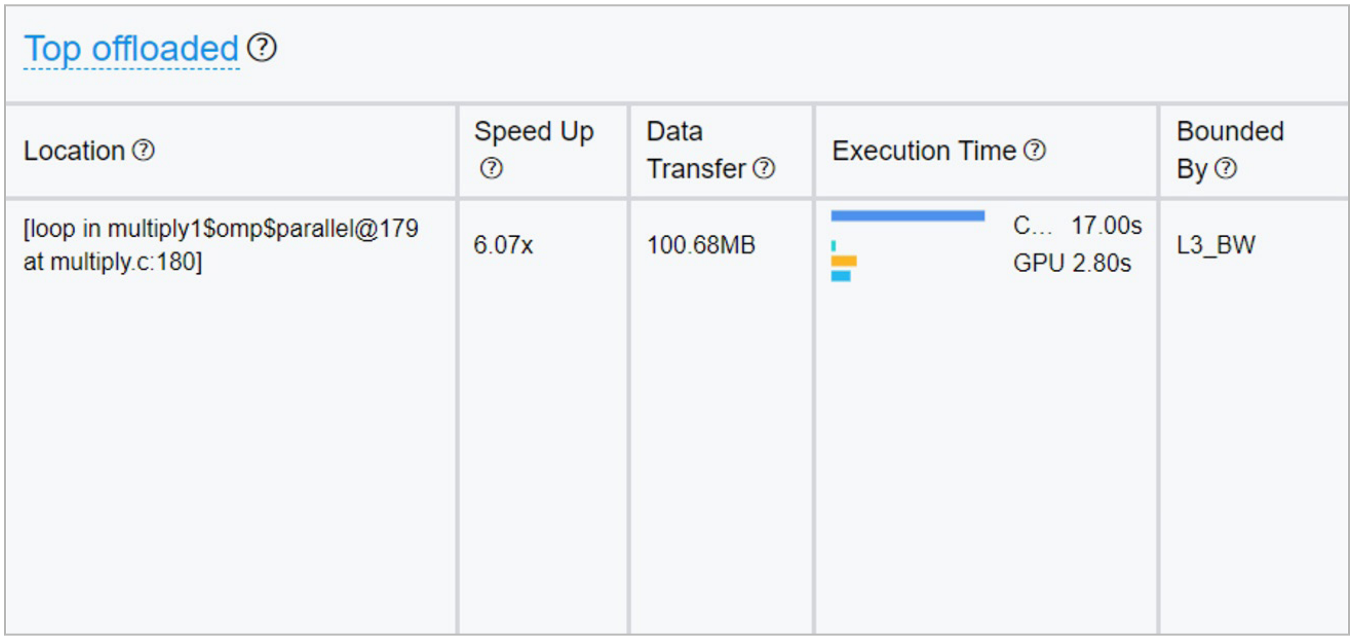
In the Summary section of the report, you can see:

- **The original CPU execution time**, the predicted execution time on the GPU accelerator, the number of offloaded regions, and the speedup in the "Program metrics" pane.
- **What the offloads are bounded by**. In our case, the offloads are 99% bounded by the last-level cache (LLC) bandwidth.
- **Exact source lines of the top offloaded code regions** that will benefit from offloading to the GPU. In our case, there's only one code region recommended for offload.
- **Exact source lines of the "Top non-offloaded" code regions** that aren't recommended for offload for various reasons. In our case, the time spent in the loops is too small to be modeled accurately and one of the loops is outside the code region marked for offloading.

Use this information to rewrite the matrix multiplication kernel in DPC++.

Rewrite the Matrix Multiplication Kernel in DPC++

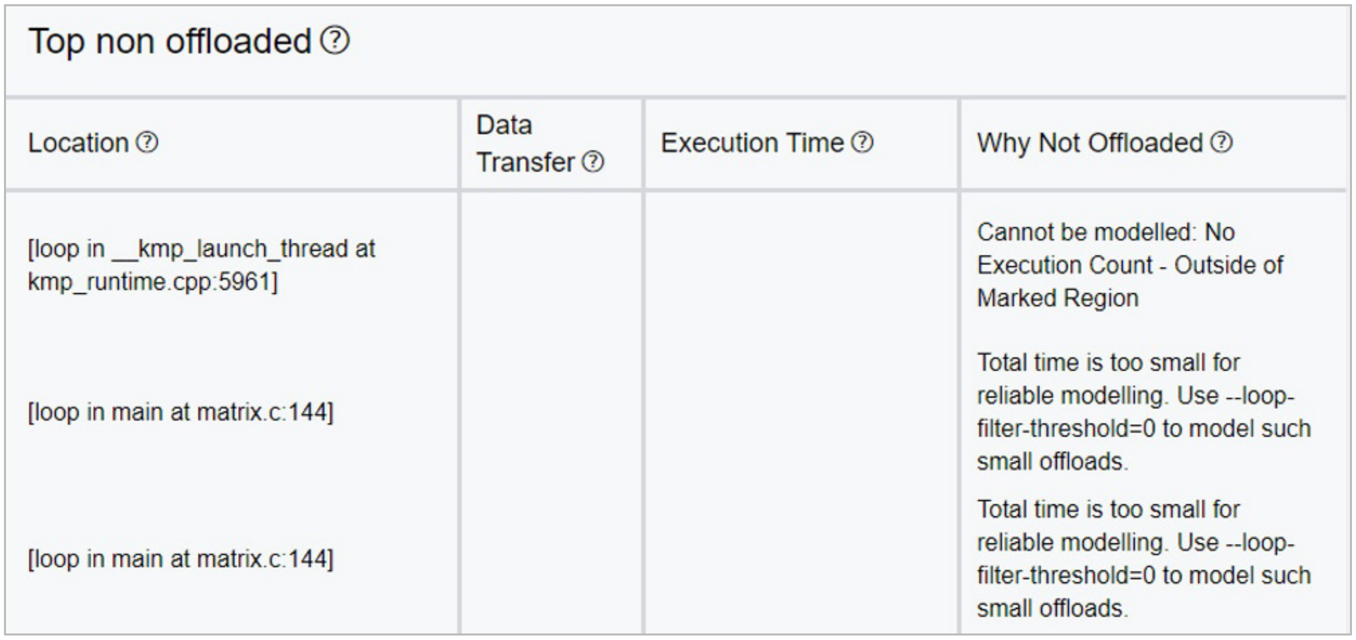
Intel Advisor (beta) provides the exact source line of the offloaded region, as shown in **Figure 3**. The tool also recommends loops that don't need to be offloaded because their compute time is too small to be modeled accurately or they're outside of a marked region (**Figure 4**).



Top offloaded

Location	Speed Up	Data Transfer	Execution Time	Bounded By
[loop in multiply1$omp$parallel@179 at multiply.c:180]	6.07x	100.68MB	C... 17.00s GPU 2.80s	L3_BW

3 Top offloaded region



Top non offloaded

Location	Data Transfer	Execution Time	Why Not Offloaded
[loop in __kmp_launch_thread at kmp_runtime.cpp:5961]			Cannot be modelled: No Execution Count - Outside of Marked Region
[loop in main at matrix.c:144]			Total time is too small for reliable modelling. Use --loop-filter-threshold=0 to model such small offloads.
[loop in main at matrix.c:144]			Total time is too small for reliable modelling. Use --loop-filter-threshold=0 to model such small offloads.

4 Top non-offloaded region

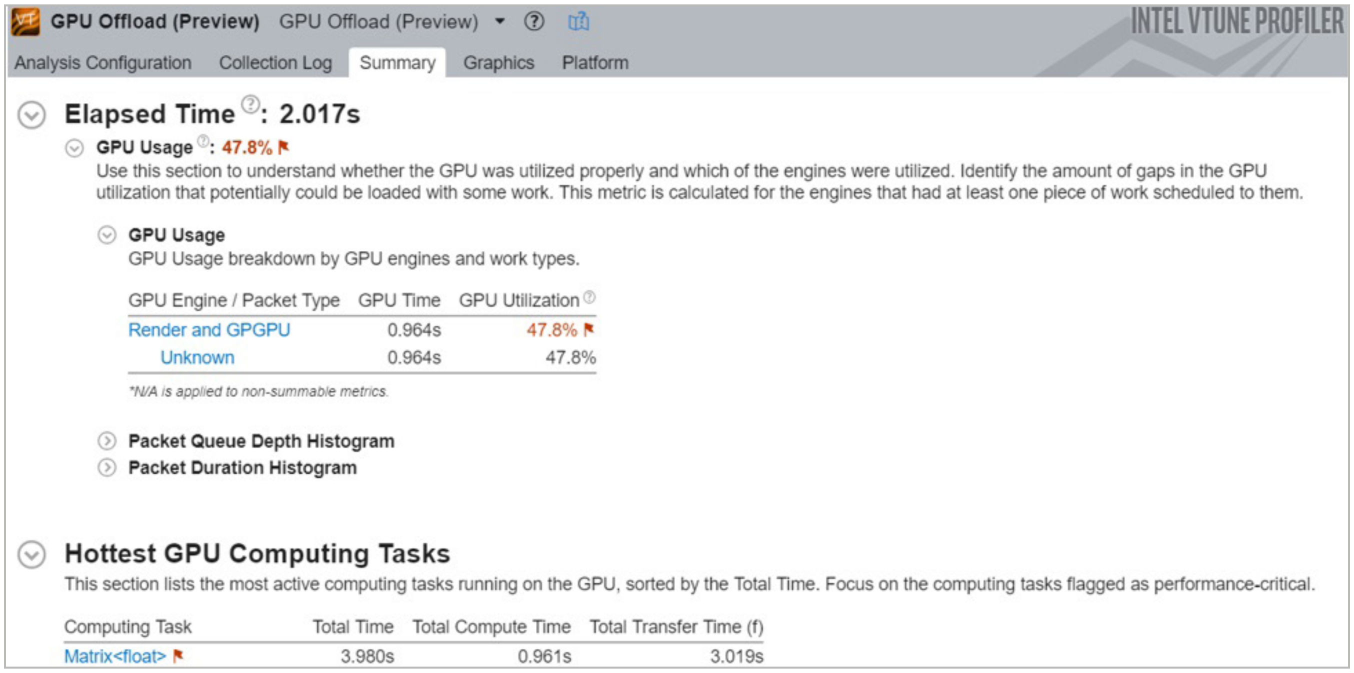
Follow these steps to rewrite the matrix multiplication kernel in DPC++ (as shown in the code sample below):

1. **Select** an offload device.
2. **Declare** a device queue.
3. **Declare** some buffers to hold the matrix.
4. **Submit** work to the device queue.
5. **Execute** the matrix multiplication in parallel.

```
void multiply1(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM], TYPE c[][NUM], TYPE t[][NUM])
{
           int i,j,k;

  // Select device
  cl::sycl::gpu_selector device;
  // Declare a deviceQueue
  cl::sycl::queue deviceQueue(device);
  // Declare a 2 dimensional range
  cl::sycl::range<2> matrix_range{NUM, NUM};

  // Declare 3 buffers and Initialize them
  cl::sycl::buffer<TYPE, 2> bufferA((TYPE*)a, matrix_range);
  cl::sycl::buffer<TYPE, 2> bufferB((TYPE*)b, matrix_range);
  cl::sycl::buffer<TYPE, 2> bufferC((TYPE*)c, matrix_range);

  // Submit our job to the queue
  deviceQueue.submit([&](cl::sycl::handler& cgh) {
    // Declare 3 accessprs to our buffers. The first 2 read and the last read_write
    auto accessorA = bufferA.template get_access<sycl_read>(cgh);
    auto accessorB = bufferB.template get_access<sycl_read>(cgh);
    auto accessorC = bufferC.template get_access<sycl_read_write>(cgh);

    // Execute Matrix multiply in parallel over our matrix_range
    // Ind is an index into this range
    cgh.parallel_for<class Matrix<TYPE>>(matrix_range,
                        [=](cl::sycl::id<2> ind) {
            int k;
               for(k=0; k<NUM; k++) {
                  // Perform computation ind[0] is row, ind[1] is col
               accessorC[ind[0]][ind[1]]  += accessorA[ind[0]][k] * accessorB[k][ind[1]];
            }
     });
  });
}
```

Optimize GPU Usage with Intel VTune Profiler (beta)

Offload Advisor helped us port our CPU kernel to a GPU, yet our initial implementation is far from optimal. We'll use the GPU offload features of Intel VTune Profiler (beta) to see how effectively we're using our GPU (**Figure 5**). GPU offload is showing that our application has an elapsed time of 2.017 seconds and our GPU utilization is 100%. We can also see that matrix multiplication is our hotspot.



Elapsed Time: 2.017s

GPU Usage: 47.8%

Use this section to understand whether the GPU was utilized properly and which of the engines were utilized. Identify the amount of gaps in the GPU utilization that potentially could be loaded with some work. This metric is calculated for the engines that had at least one piece of work scheduled to them.

GPU Usage

GPU Usage breakdown by GPU engines and work types.

GPU Engine / Packet Type	GPU Time	GPU Utilization
Render and GPGPU	0.964s	47.8%
Unknown	0.964s	47.8%

*N/A is applied to non-summable metrics.

Packet Queue Depth Histogram

Packet Duration Histogram

Hottest GPU Computing Tasks

This section lists the most active computing tasks running on the GPU, sorted by the Total Time. Focus on the computing tasks flagged as performance-critical.

Computing Task	Total Time	Total Compute Time	Total Transfer Time (f)
Matrix<float>	3.980s	0.961s	3.019s

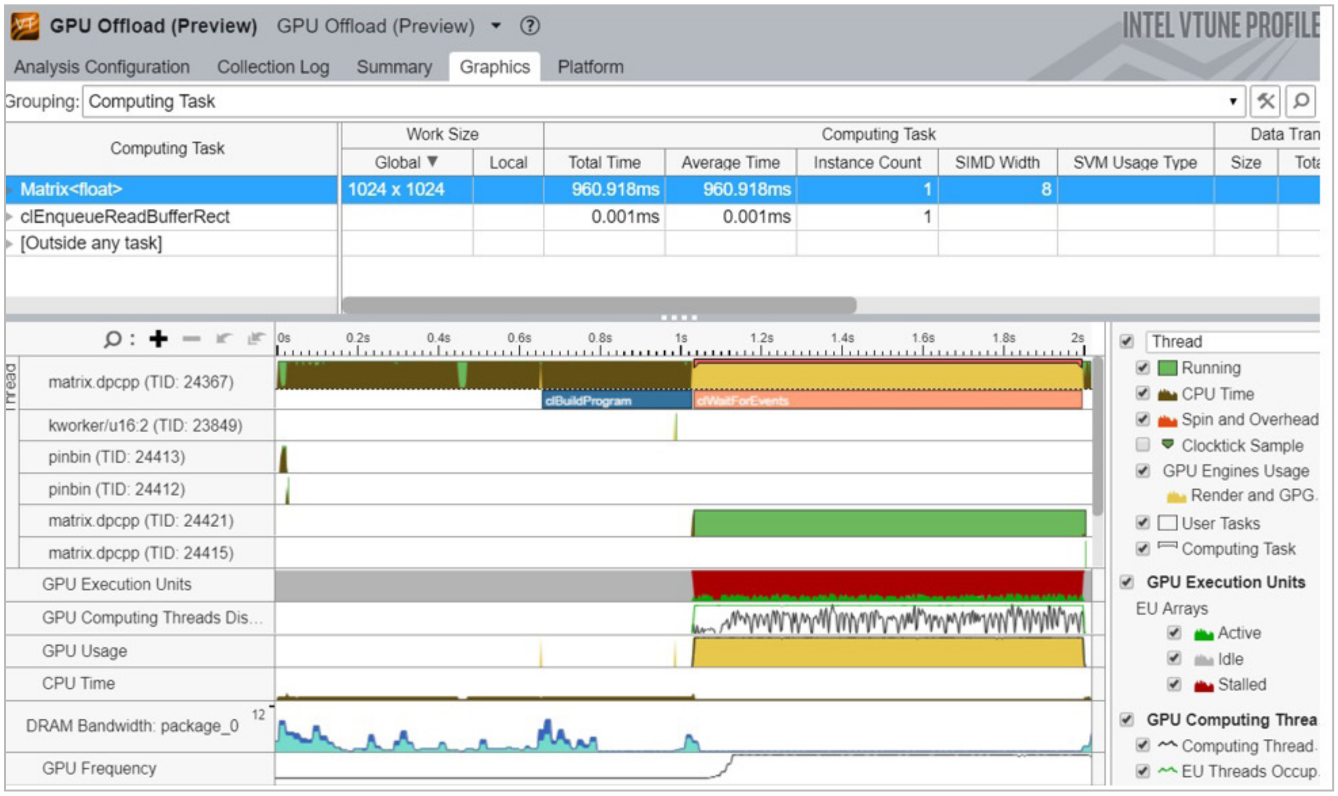
5 GPU offload report

By switching to the Graphics and Platform tabs, we can see more details. Intel VTune Profiler (beta) shows a synchronized timeline between the CPU and GPU. GPU offload does indicate that our GPU execution units are stalling, as indicated by the dark red bar in the timeline (**Figure 6**).

Next, we'll run the Intel VTune Profiler (beta) GPU Hotspots report to try to identify the source of our low GPU utilization and stalls. Click on the Graphics tab in GPU Hotspots and you can see a high-level diagram of your architecture (**Figure 7**). Notice that we're not using the shared local memory (SLM) cache. Also notice that we're moving around 159.02 GB/s in total.

We'll try two optimization techniques:

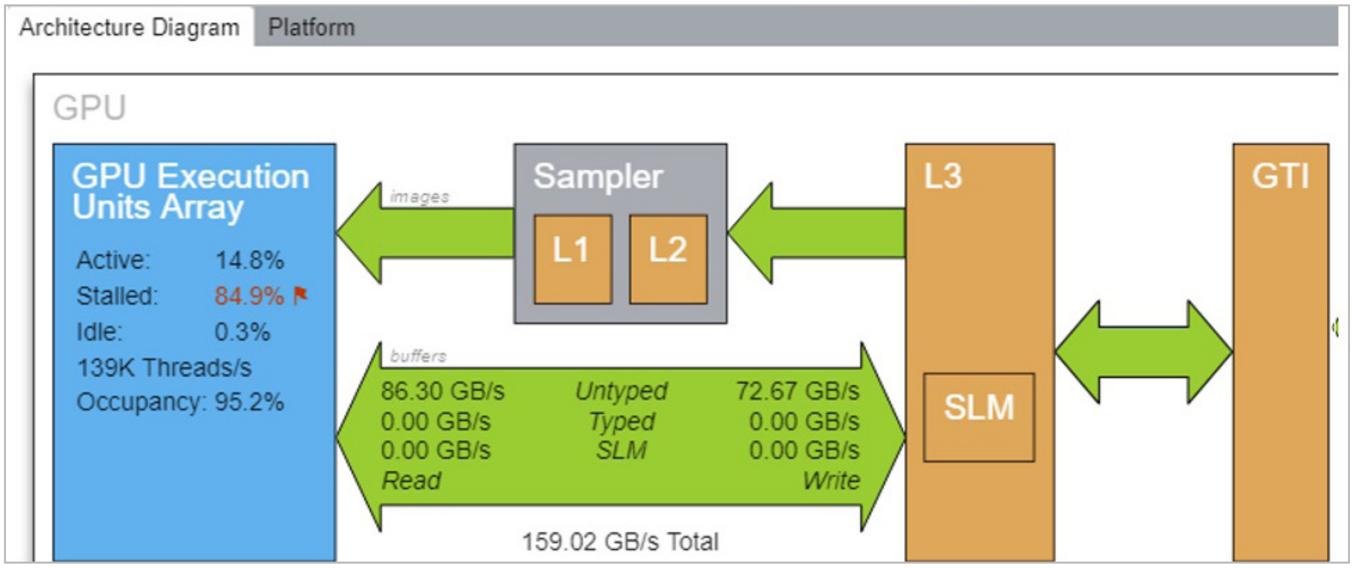
1. Cache blocking the matrix
2. Using local memory



Grouping: Computing Task

Computing Task	Work Size		Computing Task					Data Tran	
	Global ▼	Local	Total Time	Average Time	Instance Count	SIMD Width	SVM Usage Type	Size	Tota
Matrix<float>	1024 x 1024		960.918ms	960.918ms	1	8			
clEnqueueReadBufferRect			0.001ms	0.001ms	1				
[Outside any task]									

6 GPU execution units are stalling

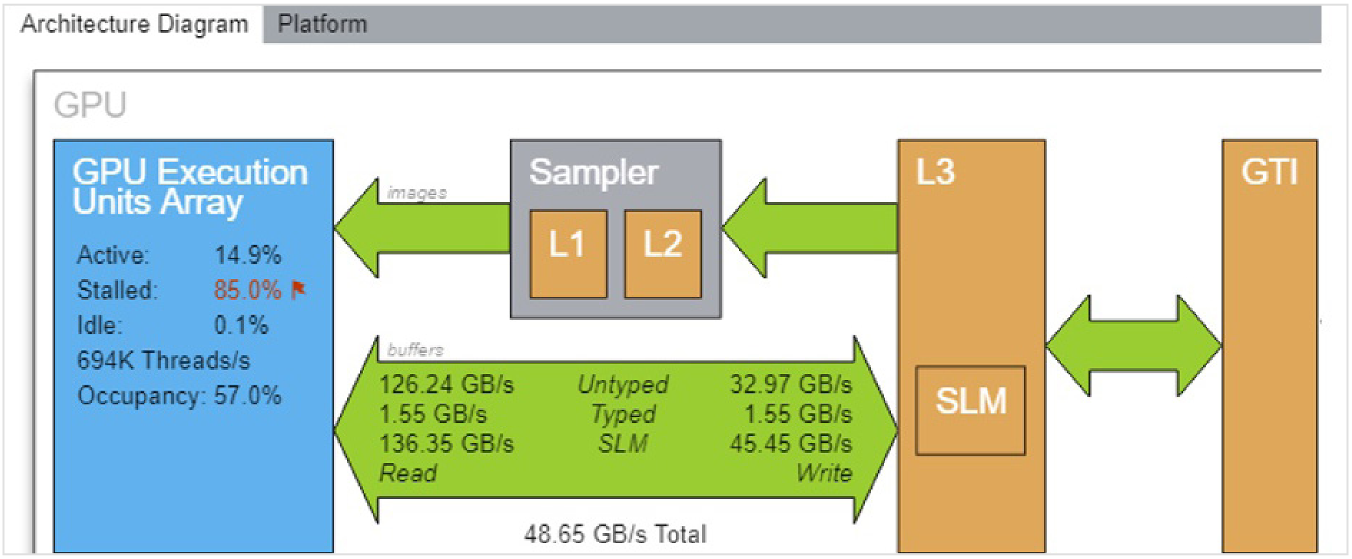


7 Architecture diagram prior to memory optimization

To implement these techniques, we need to break our matrix into tiles and work on them separately in the SLM cache:

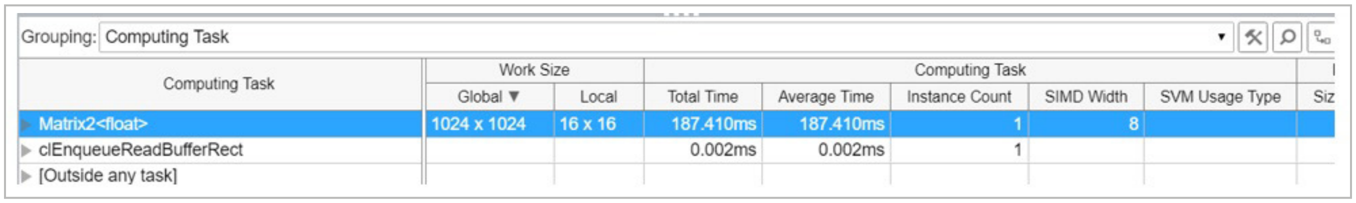
```
deviceQueue.submit([&](cl::sycl::handler& cgh) {
    // Declare 3 accessors to our buffers. The first 2 read and the last read_write
    auto accessorA = bufferA.template get_access<sycl_read>(cgh);
    auto accessorB = bufferB.template get_access<sycl_read>(cgh);
    auto accessorC = bufferC.template get_access<sycl_read_write>(cgh);

           //Create matrix tiles
    cl::sycl::accessor<TYPE, 2, cl::sycl::access::mode::read_write, cl::sycl::access::target::local>
aTile(cl::sycl::range<2>(MATRIX_TILE_SIZE, MATRIX_TILE_SIZE), cgh);
    cl::sycl::accessor<TYPE, 2, cl::sycl::access::mode::read_write, cl::sycl::access::target::local>
bTile(cl::sycl::range<2>(MATRIX_TILE_SIZE, MATRIX_TILE_SIZE), cgh);
    // Execute matrix multiply in parallel over our matrix_range
    // ind is an index into this range
    cgh.parallel_for<class Matrix2<TYPE>>(cl::sycl::nd_range<2>(matrix_range,tile_range),
                  [=](cl::sycl::nd_item<2> it) {
            int k;
            const int numTiles = NUM/MATRIX_TILE_SIZE;
            const int row = it.get_local_id(0);
            const int col = it.get_local_id(1);
            const int globalRow = MATRIX_TILE_SIZE*it.get_group(0) + row;
            const int globalCol = MATRIX_TILE_SIZE*it.get_group(1) + col;
            TYPE acc = 0.0;
            for (int t=0; t<numTiles; t++) {
                 const int tiledRow = MATRIX_TILE_SIZE*t + row;
                 const int tiledCol = MATRIX_TILE_SIZE*t + col;
                 aTile[row][col] = accessorA[globalRow][tiledCol];
                 bTile[row][col] = accessorB[tiledRow][globalCol];
                 it.barrier(cl::sycl::access::fence_space::local_space);
                 for(k=0; k<MATRIX_TILE_SIZE; k++) {
                     // Perform computation ind[0] is row, ind[1] is col
                     acc += aTile[row][k] * bTile[k][col];
                 }
                 it.barrier(cl::sycl::access::fence_space::local_space);
            }
            accessorC[globalRow][globalCol] = acc;
     });
  });
```

The new architecture diagram shows that this is much more efficient (**Figure 8**). We're making use of the SLM: 136.35 GB/s for read and 45.45 GB/s for write.



8 **Architecture diagram after memory optimization**

Click on the Platform tab (**Figure 9**) to see some additional metrics. You can see that our matrix is stored in 1024x1024 global memory, but we make use of local memory in a 16x16 tile. The elapsed time for our new matrix is 1.22s, a 1.64x improvement.


Grouping: Computing Task

Computing Task	Work Size		Computing Task					
	Global ▼	Local	Total Time	Average Time	Instance Count	SIMD Width	SVM Usage Type	Siz
Matrix2<float>	1024 x 1024	16 x 16	187.410ms	187.410ms	1	8		
clEnqueueReadBufferRect			0.002ms	0.002ms	1			
[Outside any task]								

9 **Platform tab**

Intel Advisor GPU Roofline

To see if the GPU version of our matrix multiplication kernel is getting the maximum performance from our hardware, we can use the new GPU Roofline feature. Intel Advisor (beta) can generate a Roofline Model for kernels running on Intel GPUs. The Roofline Model offers a very efficient way to characterize your kernels and visualize how far you are from ideal performance.

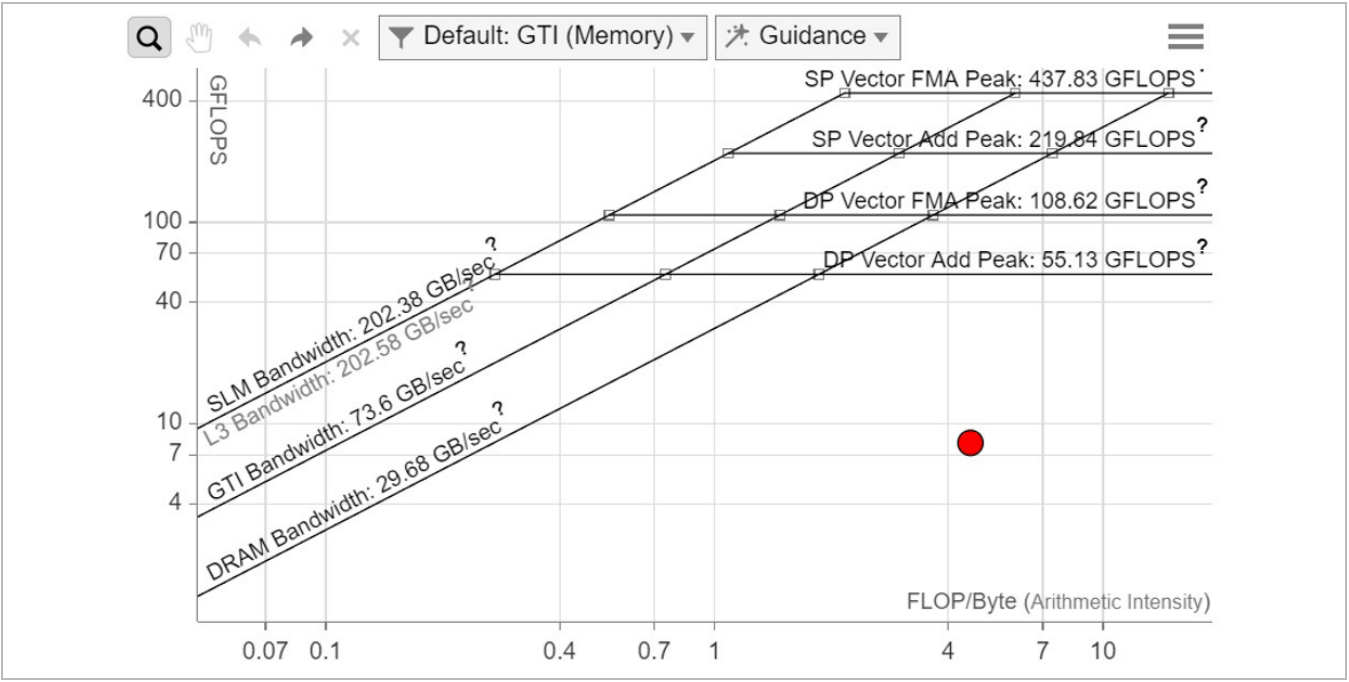
The Roofline Model on GPU is a tech preview feature that's not available by default. Here's a five-step process to enable it:

- First, ensure that you have a DPC++ code that correctly runs on the GPU. You can easily check which hardware you are running on by doing something like this:

```
Cl::sycl::default_selector selector;
Cl::sycl::queue queue(delector);
auto d = queue.get_device();
std::cout<<"Running on
:"<<d.get_info<cl::sycl::info::device::name>()<<std::endl;
```

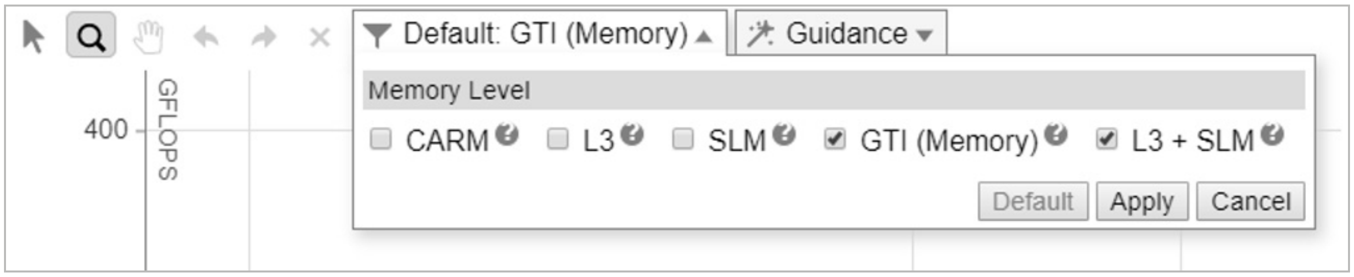
- Since this is a technical preview, you need to enable GPU profiling by setting the following environment variable: `export ADVIXE_EXPERIMENTAL=gpu-profiling`.
- Next, run the survey with the `--enable-gpu-profiling` option: `advixe-cl -collect survey --enable-gpu-profiling --project-dir <my_project_directory> --search-dir src:r=<my_source_directory> -- ./myapp param1 param2`
- Run the tripcount analysis with the `--enable-gpu-profiling` option: `advixe-cl -collect tripcounts --stacks --flop --enable-gpu-profiling --project-dir <my_project_directory> --search-dir src:r=<my_source_directory> -- ./myapp param1 param2`
- Generate the Roofline Model: `advixe-cl --report=roofline --gpu --project-dir <my_project_directory> --report-output=roofline.html`

Once the last step is executed, the file `roofline.html` will be generated and can be opened in any Web browser (**Figure 10**).

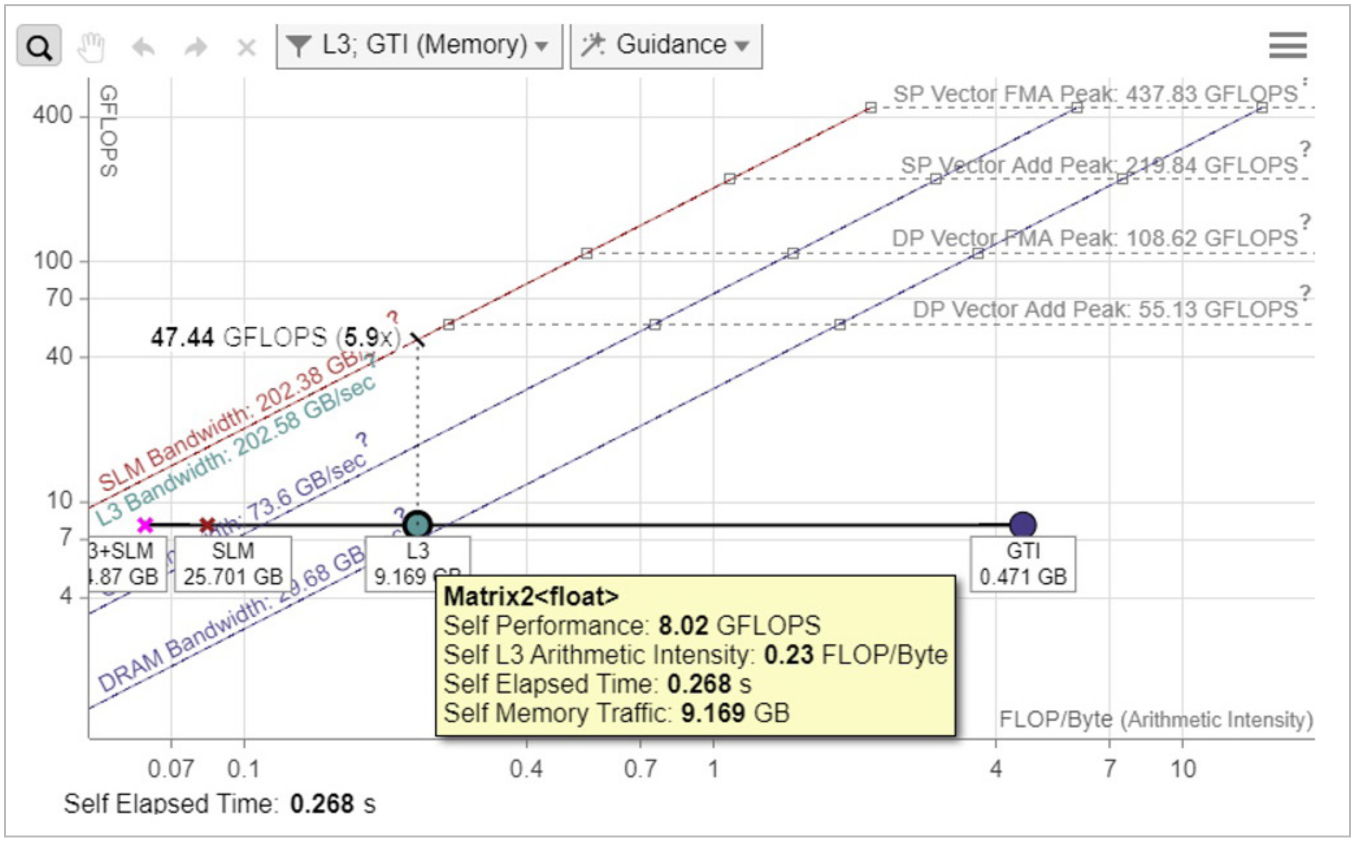


10 Roofline report

It's also possible to display different dots based on which memory subsystem is used for the arithmetic intensity computation (**Figure 11**).



11 Memory level



Matrix2<float>
Self Performance: **8.02** GFLOPS
Self L3 Arithmetic Intensity: **0.23** FLOP/Byte
Self Elapsed Time: **0.268** s
Self Memory Traffic: **9.169** GB

Self Elapsed Time: **0.268** s

12 Roofline chart

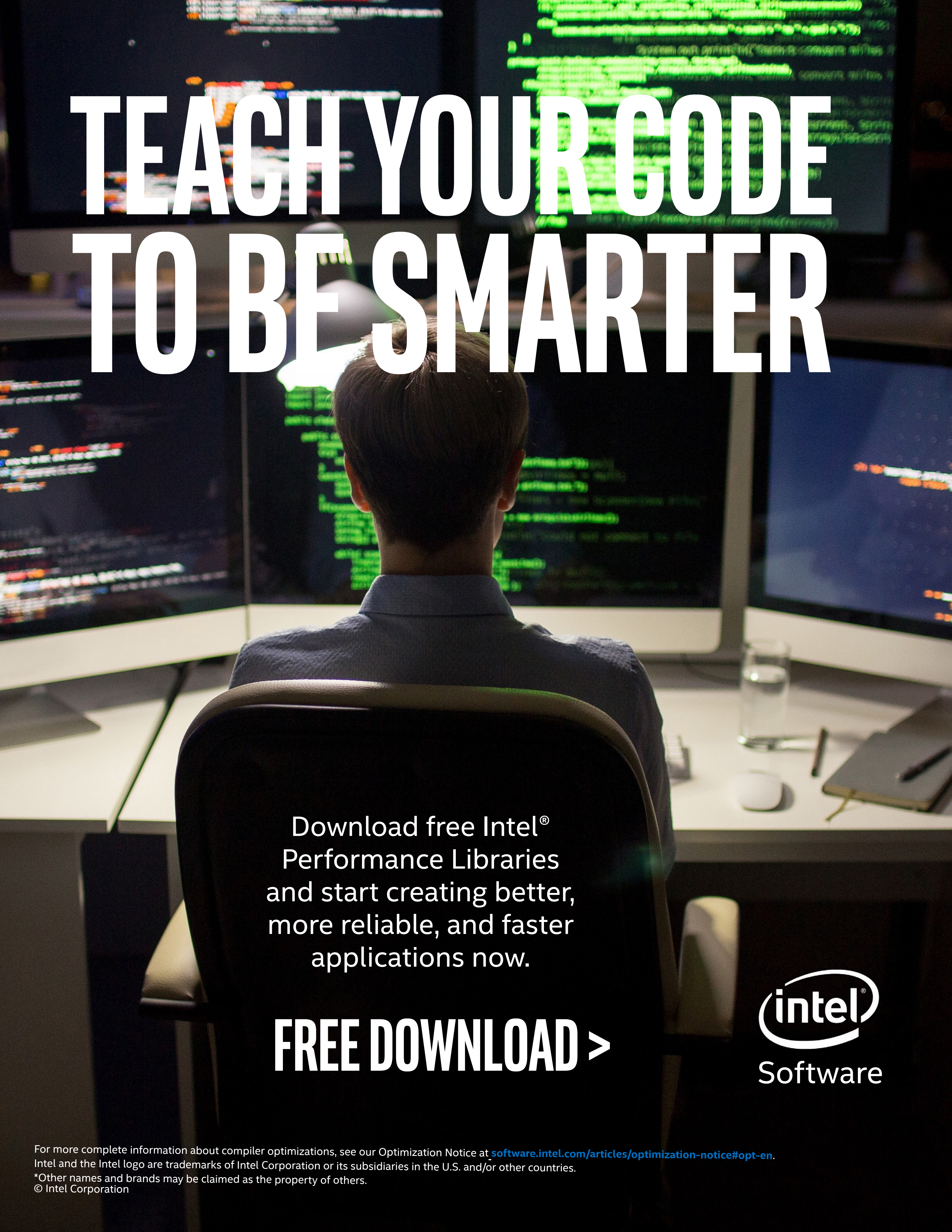
As you can can see from the roofline chart in **Figure 12**, our L3 dot is very close to the L3 maximium bandwidth. To get more FLOPS, we need to optimize our cache utilization further. A cache-blocking optimization strategy can make better use of memory and should increase our performance. The GTI (traffic between our GPU, GPU uncore [LLC], and main memory) is far from the GTI roofline, so transfer costs between CPU and GPU do not seem to be an issue.

Freedom to Focus

Intel oneAPI products will provide a standard, simplified programming model that can run seamlessly on the scalar, vector, matrix, and spatial architectures deployed in CPUs and accelerators. It will give users the freedom to focus on their code instead of the underlying mechanism that generates the best possible machine instructions.

Learn More

- **oneAPI Initiative**
- **Intel® oneAPI Toolkits (beta)**
- **Intel® Advisor (beta)**
- **Intel® VTune™ Profiler (beta)**



TEACH YOUR CODE TO BE SMARTER

Download free Intel® Performance Libraries and start creating better, more reliable, and faster applications now.

FREE DOWNLOAD >

intel® Software

SPEEDING UP MONTE CARLO SIMULATION WITH ONEAPI

Intel® oneAPI Math Kernel Library (Beta) Data Parallel C++ Usage Models

Alina Elizarova and Pavel Dyakov, Math Algorithm Engineers, and Gennady Fedorov, Software Technical Consulting Engineer, Intel Corporation

Intel® oneAPI Math Kernel Library (beta) (oneMKL beta) gives developers and data scientists enhanced math routines for creating science, engineering, or financial applications. You can use it to optimize code for current and future generations of Intel® CPUs and GPUs. In this article, we'll apply some oneMKL beta **Data Parallel C++ (DPC++)** usage models to a Monte Carlo (MC) simulation example using the oneMKL beta random number generators (RNG).

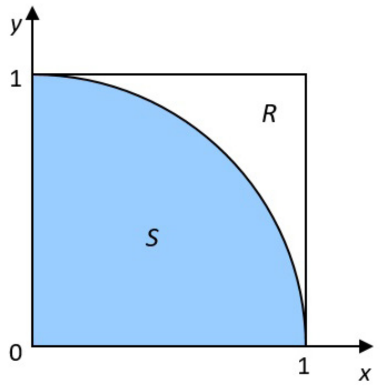
We'll look at five usage models:

1. Reference C++
2. oneMKL beta DPC++
3. Extended oneMKL beta DPC++
4. Heterogeneous parallel implementations
5. Implementations based on different memory allocation techniques

The bottom line? Minimizing data transfer between the host and device significantly improves performance.

Computing π by Numerical Integration

MC simulations are a broad class of computational algorithms that use repeated, random sampling to obtain numerical results[1]. **Figure 1** shows how to compute π using the MC method.



1 Computing π using the Monte Carlo method

We'll consider a quadrant inscribed in a unit square. The area of sector S is equal to $Area(S)=1/4\ \pi r^2=\pi/4$, and the area of the square R is equal to $Area(R)=1$. If we randomly choose the point $c=(x,y)$ from the unit square R, the probability that c is within sector S is:

$$Pr(c \in S) = \frac{Area(S)}{Area(R)} = \pi/4$$

We can consider n points, where n is sufficiently large, and count the number of points falling into S – k. Then, we can approximate the probability $Pr(c \in S)$ by the ratio k/n, or:

$$\pi/4 \simeq k/n$$

We can approximate π as:

$$\pi \cong \frac{4k}{n}$$

According to the laws of large numbers, the larger the `n`, the more accurate our π approximation. More accurately, from the Bernoulli theorem, for any $\varepsilon > 0$

$$Pr\left(\left|\frac{k}{n} - 4\pi\right| \geq \varepsilon\right) \leq \frac{1}{4n\varepsilon^2}$$

If `x` and `y` coordinates of the tested point c are $0 \leq x \leq 1$ and $0 \leq y \leq 1$ (abscissa and ordinate, respectively), then `c` falls into sector `S` when:

$$x^2 + y^2 \leq 1$$

To summarize:

1. Generate `n` 2D points where each point is represented by two uniformly distributed random numbers over the `[0,1)` interval.
2. Count the number of points that fall into sector `S`.
3. Calculate the approximate value of π using the previous formula.

Reference C++ Example of π Estimation

Let's consider a function `estimate_pi` that takes a number of 2D points `n_points` and performs the computation described above:

```
float estimate_pi( size_t n_points ) {
    float estimated_pi;           // Estimated value of Pi
    size_t n_under_curve = 0;     // Number of points fallen under the curve

    // Allocate storage for random numbers
    std::vector<float> x(n_points);
    std::vector<float> y(n_points);

    // Step 1. Generate n_points random numbers
    // 1.1. Generator initialization
    std::default_random_engine engine(SEED);
    std::uniform_real_distribution<float> distr(0.0f, 1.0f);

    // 1.2. Random number generation
    for(int i = 0; i < n_points; i++) {
        x[i] = distr(engine);
        y[i] = distr(engine);
    }

    // Step 2. Count the number of points fallen under the curve
    for ( int i = 0; i < n_points; i++ ) {
        if (x[i] * x[i] + y[i] * y[i] <= 1.0f)
            n_under_curve++;
    }

    // Step 3. Calculate approximated value of Pi
    estimated_pi = n_under_curve / ((float)n_points) * 4.0;
    return estimated_pi;
}
```

This example uses RNG functionality from the C++ 11 standard. In Step 1, we initialize the RNG by creating two instances: `engine` and `distribution`. The engine holds a state of a generator and provides independent and identically distributed random variables. `distr` describes transforming generator output with statistics and parameters. In this example, `uniform_real_distribution` produces random floating-point values, uniformly distributed in the interval `[a, b)`.

In Step 1.2, random numbers are obtained by passing the engine to an operator() distribution. A single floating-point variable is obtained. The loop fills vectors `x` and `y` with random numbers. In Step 2, each `(x, y)` position is checked to determine how many points fall into the `S` sector. The result is stored in the `n_under_curve` variable. Finally, the estimated `π` value is calculated and returned to the main program (Step 3).

oneMKL Beta DPC++ Example of π Estimation

Now let's modify the previous `estimate_pi` function and add `cl::sycl::queue`. DPC++ lets you choose a device to run on using a **selector interface**:

```
float estimate_pi(size_t n_points) {
    float estimated_pi;          // Estimated value of Pi
    size_t n_under_curve = 0;    // Number of points fallen under the curve

    // Allocate storage for random numbers
    cl::sycl::buffer<float, 1> x_buf(cl::sycl::range<1>{n_points});
    cl::sycl::buffer<float, 1> y_buf(cl::sycl::range<1>{n_points});

    // Choose device to run on and create queue
    cl::sycl::gpu_selector selector;
    cl::sycl::queue queue(selector);

  std::cout << "Running on: " <<
        queue.get_device().get_info<cl::sycl::info::device::name>() <<
        std::endl;

    // Step 1. Generate n_points random numbers
    // 1.1. Generator initialization
    mkl::rng::philox4x32x10 engine(queue, SEED);
    mkl::rng::uniform<float, mkl::rng::standard> distr(0.0f, 1.0f);

    // 1.2. Random number generation
    mkl::rng::generate(distr, engine, n_points, x_buf);
    mkl::rng::generate(distr, engine, n_points, y_buf);

    // Step 2. Count the number of points fallen under the curve
    auto x_acc = x_buf.template get_access<cl::sycl::access::mode::read>();
    auto y_acc = y_buf.template get_access<cl::sycl::access::mode::read>();
    for ( int i = 0; i < n_points; i++ ) {
        if (x_acc[i] * x_acc[i] + y_acc[i] * y_acc[i] <= 1.0f)
            n_under_curve++;
    }
    // Step 3. Calculate approximated value of Pi
    estimated_pi = n_under_curve / ((float)n_points) * 4.0;
    return estimated_pi;
}
```

`cl::sycl::queue` is an input argument to oneMKL beta functions. The library's kernels are submitted in this queue, with no code changes required to switch between devices (i.e., host or accelerator). CPU and GPU devices are available in the oneMKL beta release.

Instead of `std::vector`, `cl::sycl::buffer` is used to store the random numbers:

```
// Reference:
    std::vector<float> x(n_points);
    std::vector<float> y(n_points);

    // Data Parallel C++:
    cl::sycl::buffer<float, 1> x_buf(cl::sycl::range<1>{n_points});
    cl::sycl::buffer<float, 1> y_buf(cl::sycl::range<1>{n_points});
```

Buffers manage data transfer between the host application and device kernels. The buffer and accessor classes are responsible for tracking memory transfers and guaranteeing data consistency across the different kernels.

In Step 1, oneMKL beta RNG APIs also initialize two entities (the RNG engine and random number distribution) as shown in Step 1.1. engine takes `cl::sycl::queue` and an initial `SEED` value as input to the constructor. The distribution `mkl::rng::uniform` has template parameters for the type of output values and method used to transform the engine's output (see the **Intel® oneAPI Math Kernel Library Data Parallel C++ Developer Reference** for details) and distribution parameters.

The `mkl::rng::generate` function is called at Step 1.2 to obtain random numbers. This function takes the distribution and engine created in the previous step, the number of elements to be generated, and storage for the result in buffers. The oneMKL beta RNG API `mkl::rng::generate()` is vectorized. Vector library subroutines often perform better than scalar routines. (See **Intel® oneMKL Vector Statistics Notes** for details.)

In Step 2, the random numbers are postprocessed on the host. Host accessors for buffers are created to access the data:

```
auto x_acc = x_buf.template get_access<cl::sycl::access::mode::read>();
auto y_acc = y_buf.template get_access<cl::sycl::access::mode::read>();
```

Other steps are the same as in the C++ reference example.

Extended oneMKL Beta DPC++ Example of π Estimation

We can optimize Step 2 in the previous example using the DPC++ Parallel STL function. This approach reduces data transfer between the host and device, which improves performance. The modification to Step 2 is as follows:

```
auto policy = dpstd::execution::make_sycl_policy<class count>(queue);

  auto x_buf_begin = dpstd::begin(x_buf);
  auto y_buf_begin = dpstd::begin(y_buf);
  auto zip_begin = dpstd::make_zip_iterator(x_buf_begin, y_buf_begin);

  n_under_curve = std::count_if(policy, zip_begin, zip_begin + n_points,
    [](auto p) {
      using std::get;
      float x, y;
      x = get<0>(p);
      y = get<1>(p);
      return x*x + y*y <= 1.0f;
    });
```

The zip iterator provides a pair of random numbers as input to the `count_if` function. Other steps are the same as in the previous oneMKL beta DPC++ example.

Unified Shared Memory (USM)-Based Example of oneMKL Beta DPC++ π Estimation

There are two approaches to DPC++ pointer-based memory management: `cl::sycl::malloc` and `cl::sycl::usm_allocator` (find more details **here**).

`cl::sycl::malloc` allows you to allocate memory directly on the host or device, or to access memory from both the host and device:

```
float* x = (float*) cl::sycl::malloc_shared(n_points * sizeof(float),
queue.get_device(), queue.get_context());
```

When you use this memory allocation, you get all the advantages of pointer arithmetic as well. This approach requires you to free any allocated memory.

The `cl::sycl::usm_allocator` approach allows you to work with standard or user containers without worrying about direct memory control:

```
  // Create usm allocator
    cl::sycl::usm_allocator<float, cl::sycl::usm::alloc::shared>
    allocator(queue.get_context(), queue.get_device());
    // Allocate storage for random numbers
    std::vector <float, cl::sycl::usm_allocator<float,
    cl::sycl::usm::alloc::shared>> x(n_points, allocator);
```

Generation is performed in the same way, but instead of `cl::sycl::buffer<float, 1>`, `float*` is used:

```
auto event = mkl::rng::generate(distr, engine, n_points, x.data());
```

Each function returns a `cl::sycl::event` that can be used for synchronization. You can call `wait()` or `wait_and_throw()` functions for these events or for the entire queue. This allows manual control of data dependencies between DPC++ kernels by calling `event.wait()` or `queue.wait()` functions.

To obtain random numbers and to count points in Step 2, you can use vectors/pointers nativelywithout creating host accessors:

```
   for ( int i = 0; i < n_points; i++ ) {
      if (x[i] * x[i] + y[i] * y[i] <= 1.0f)
        n_under_curve++;
    }
```

Heterogeneous Execution of oneMKL Beta DPC++ π Estimation

We can modify the previous examples to offload part of the computation to an accelerator instead of doing the entire computation on the host. The APIs stay the same. You just need to choose the device for `cl::sycl::queue`. Two queues are needed for parallel execution on different devices:

```
  // Create queues for Host and GPU
    cl::sycl::queue queue_gpu(cl::sycl::gpu_selector(), exception_handler);
    cl::sycl::queue queue_host(cl::sycl::host_selector(),
    exception_handler);
```

You can also allocate memory separately. CPU allocation can be done directly on the host:

```
// Create usm allocators for shared and Host memory allocation
cl::sycl::usm_allocator<float, cl::sycl::usm::alloc::shared>
allocator_gpu(queue_gpu.get_context(), queue_gpu.get_device());
// Allocate storage for random numbers
std::vector <float, decltype(allocator_gpu)> x(n_points, allocator_gpu);
std::vector <float> y(n_points);
```

oneMKL beta RNG engines should be constructed for the exact type of device, so two objects are required. The second engine may be initialized with another seed, or should continue the sequence offset by `n_points`. (See RNGs in parallel computations in **Intel® MKL Vector Statistics Notes** for details.)

```
mkl::rng::philox4x32x10 engine_gpu(queue_gpu, SEED);
mkl::rng::philox4x32x10 engine_host(queue_host, SEED);
mkl::rng::skip_ahead(engine_host, n_points);
```

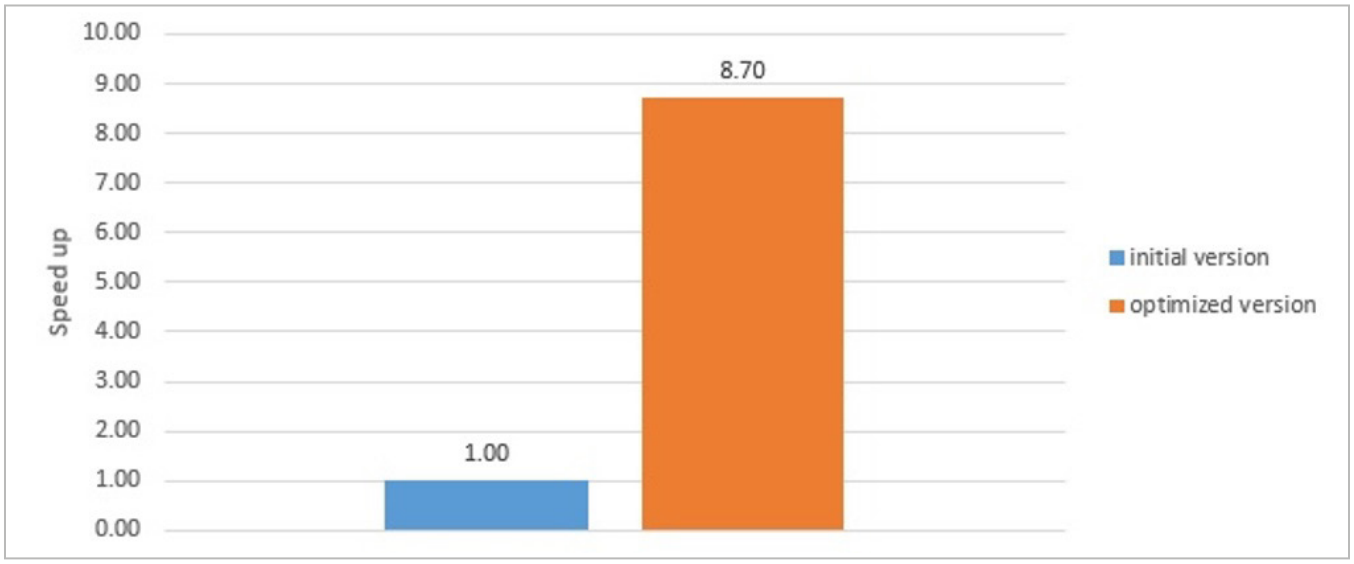
Generation and postprocessing may be called as in the previous examples. In this example, `std::count_if` is used on a host with a parallel execution policy:

```
auto event1 = mkl::rng::generate(distr, engine_gpu, n_points, x.data());
auto event2 = mkl::rng::generate(distr, engine_host, n_points,
y.data());
// Wait to finish generation
event1.wait_and_throw();
event2.wait_and_throw();
n_under_curve = std::count_if(std::execution::par,
dpstd::make_zip_iterator(x.begin(),y.begin()),
dpstd::make_zip_iterator(x.end(),y.end()), [](auto p) {
using std::get;
float dx, dy;
dx = get<0>(p);
dy = get<1>(p);
return dx*dx + dy*dy <= 1.0f;
```

This approach allows balancing work between devices and/or completes different tasks in parallel on any supported devices within a single API.

Performance Comparison

Figure 2 compares the oneMKL beta DPC++ and Extended oneMKL beta DPC++ examples.



2 **Performance comparison**

Hardware and software parameters:

- **Hardware:** Intel® Core™ i9-9900K processor @ 3.60GHz, Intel® Gen12LP HD Graphics, NEO Graphics NEO
- **Operating system:** Ubuntu* 18.04.2 LTS
- **Software:** oneMKL beta

Simulation parameters:

- **Number of generated 2D points:** 10^8
- **Random number engine:** `mkl::rng::philox4x32x10`
- **Random number distribution:** Uniform single-precision
- **Measurement region:** Computational part of `estimate_pi()` functions (without memory allocation overhead)

Speeding Up Performance

We've demonstrated different oneMKL beta DPC++ usage models applied to π estimation by MC simulations. Slightly modifying the reference C++ example will let you use DPC++ features and oneMKL beta functions to execute code on the different supported devices, including heterogeneous execution. Reducing data transfer between the host and device can significantly improve performance, as shown in **Figure 2**.

References

1. Knuth, Donald E. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms, 2nd edition*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1981.

BLOG HIGHLIGHTS

More Adventures in Graph Analytics Benchmarking

BY HENRY GABB

If you've read my last two blogs, **Measuring Graph Analytics Performance** and **Adventures in Graph Analytics Benchmarking**, you know that I've been harping on graph analytics benchmarking a lot lately. You also know that I use the **GAP Benchmark Suite** from the University of California, Berkeley, because it's easy to run, tests multiple graph algorithms and topologies, provides good coverage of the graph analytics landscape—and, most important—gives comprehensive, objective, and reproducible results.

Read on >

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

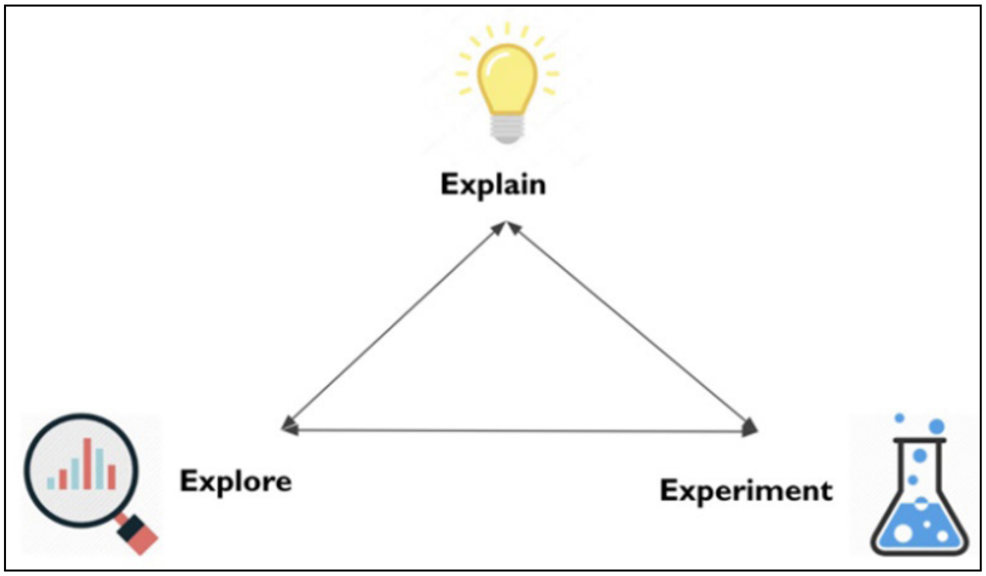
BRINGING ACCELERATED ANALYTICS AT SCALE TO INTEL® ARCHITECTURE

Unifying Data Science with Traditional Analytics on Modern Hardware

Venkat Krishnamurthy, Product Vice President, and Kathryn Vandiver, Senior Director, Platform and Core Engineering, OmniSci

OmniSci has a vision of pioneering modern hardware and software to allow for data insights at the speed of curiosity. One catalyst for this effort is today's open data science stack, a platform for experimentation. Through our shared vision and collaboration with Intel, we're working to advance and unify the worlds of data science with traditional analytics on modern hardware. In this article, we'll look at the relationship between data science applications and data discovery and explain how OmniSci and Intel are working together to advance innovation in this ecosystem.

It's clear that we now live in a world where artificial intelligence (AI) is rapidly extending human perception and intuition. Our colleagues at Intel also see this shift, and the natural progression of AI, as part of the continuum of ways for people to understand the world through data. It's useful to think of a natural "loop" in how this understanding develops (**Figure 1**), going from data exploration with visual analytics tools (since there's no match for human visual perception to quickly understand trends in data), to experimentation, where a data scientist builds models, and finally to explanation, where both visual analytics tools and machine learning methods combine to reveal key insights in a seamless workflow.



1 **The loop of understanding**

Today's open data science stack is a platform for experimentation, founded on interlocking open source innovation across multiple ecosystems:

- The Python* and PyData* stack (Numpy*/SciPy*)
- Pandas*
- MatPlotLib*
- Dask*
- Numba*
- The R language and its thriving ecosystem
- The Julia* language
- The Jupyter* and JupyterLab* project

These components lower the cost of curiosity and help drive interactive computing in general, and data-driven storytelling in particular. As an example, **Figure 2** shows the first-ever photograph of a black hole, coming to life inside a Jupyter notebook, powered by these tools.

Katie Bouman
1 hr

Watching in disbelief as the first image I ever made of a black hole was in the process of being reconstructed.

97 · 8 Comments 5 Shares

View 2 more comments

Kayhan Batmanghelich This is amazing Katie! Congratulations
1h

Wardah Inam That is so cool!
1h

Shaun Pursglove You are so cute!
1h

Vikas Ramachandra very cool Katie!
47m

Adrian Dalca Congrats 😃
36m

Miki Rubinstein Congratulations!!
17m

2 Photographing a black hole (learn more here)

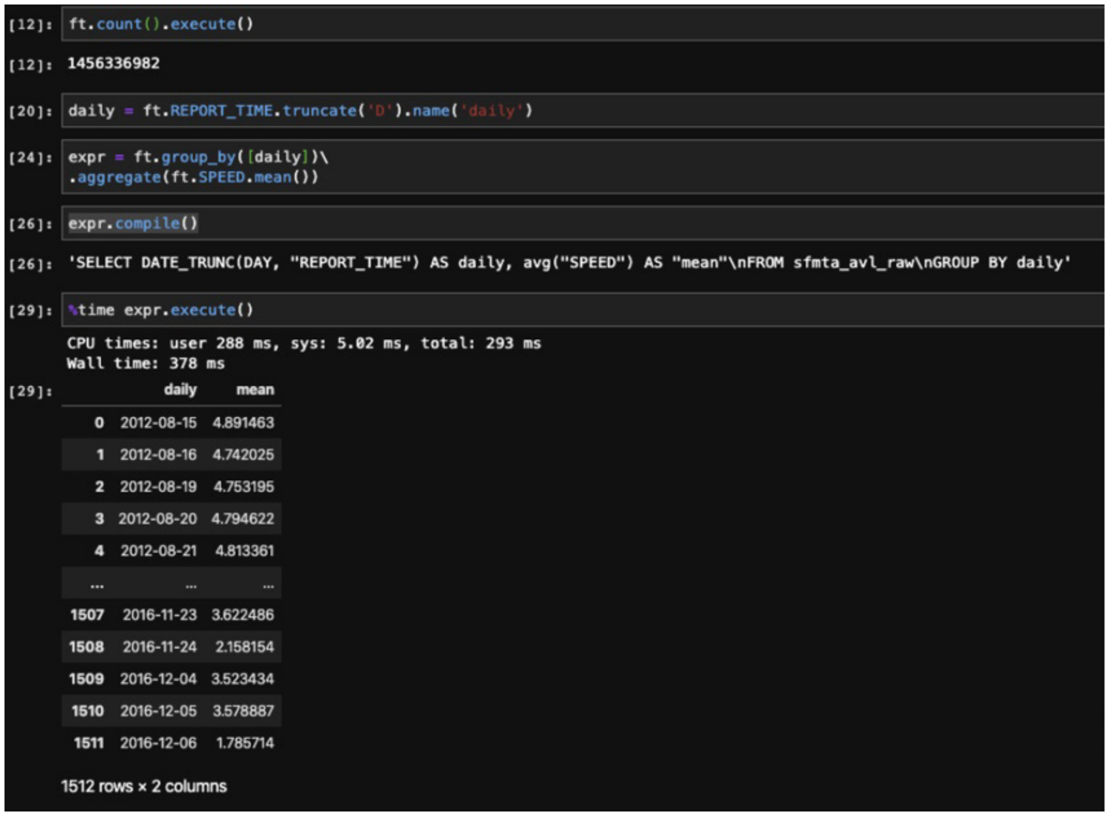
Back in 2017, when OmniSci was still known as MapD, we recognized the need to be part of this world and contribute to its growth. In-memory databases and dataframes are becoming essential technology, serving as an entry point for any type of data in the preparation, preprocessing, extraction, transformation, and loading phases of end-to-end analytics.

For a Python-powered end-to-end analytics pipeline, there are libraries, like Pandas, that have functional power but lack optimizations for Intel® architecture. This means there's a need for an open-source, performant framework that can harnesses the computing power of existing and emerging Intel® hardware. Our collaboration with Intel allows OmniSciDB* to be the basis for such a framework.

Pandas is a critical component of the PyData ecosystem, and we decided it would be counterproductive to replicate its entire API. Instead, we believe a great deal of Pandas' value lies in its powerful, expressive way of evaluating analytic expressions on an in-memory dataframe. This is how OmniSciDB query execution works already, so the problem became one of finding (or building) an API that was familiar and "pythonic" while leveraging OmniSciDB underneath. As a next step in this process, working with Intel, we're putting in place a common dataframe for OmniSci that allows for a scalable and performant workflow execution within the open data science ecosystem.

As it turns out, Wes McKinney, the creator of Pandas, had already started down this road with **Ibis***, whose stated goal is to take the productivity that Pandas provides and adapt it to a higher level of scalability using languages such as SQL*. Ibis, as we're discovering, is an absolute delight to use—especially paired with OmniSci's speed and scale. The deferred expression model makes it really easy to perform complex analytics on **any backend** that's accessible via SQL (and, in a nice twist, Pandas itself). By aligning on a consistent dataframe and Python bindings, the creation of a scalable workflow with OmniSci and Pandas can now become a high-performance backend with Ibis.

Figure 3 shows Ibis at work on a telematics dataset with 1.45 billion rows where we're setting up an aggregate expression and evaluating it, producing a Pandas dataframe as a result. The expression is lazily evaluated. Ibis compiles it into a SQL query and then executes it only when needed. Also, notice that the response is close to instantaneous, even running against an OmniSci server in our data center over a VPN.



```
[12]: ft.count().execute()
[12]: 1456336982
[20]: daily = ft.REPORT_TIME.truncate('D').name('daily')
[24]: expr = ft.group_by([daily])\
      .aggregate(ft.SPEED.mean())
[26]: expr.compile()
[26]: 'SELECT DATE_TRUNC(DAY, "REPORT_TIME") AS daily, avg("SPEED") AS "mean"\nFROM sfmta_avl_raw\nGROUP BY daily'
[29]: %time expr.execute()
      CPU times: user 288 ms, sys: 5.02 ms, total: 293 ms
      Wall time: 378 ms
[29]:
             daily       mean
         0   2012-08-15  4.891463
         1   2012-08-16  4.742025
         2   2012-08-19  4.753195
         3   2012-08-20  4.794622
         4   2012-08-21  4.813361
       ...   ...         ...
      1507   2016-11-23  3.622486
      1508   2016-11-24  2.158154
      1509   2016-12-04  3.523434
      1510   2016-12-05  3.578887
      1511   2016-12-06  1.785714

      1512 rows × 2 columns
```

3 **Ibis on a dataset with 1.45 billion rows**

Thanks to this new Python API, we can already direct Ibis output to Apache Arrow*-based data frames memory using Pandas. Work is underway, in conjunction with Intel data scientists and machine learning experts, to continue to reduce the overhead of the interface with further acceleration on **Intel® Xeon® Scalable processors** with our internal result set format.

(Here's a pretty cool aside: This Ibis- and Pandas-supported backend means you can do analysis across data sources inside a single JupyterLab notebook with one API. Simply create a connection over any of these backends and you can run Ibis expressions against them. Also, because the returned results for remote backends default to Pandas, you can wrap an Ibis connection around that **Pandas dataframe**. The possibilities are endless.)

There and Back Again

Finally, we've worked on **pymapd**, too, keeping up with the breathless pace of change on underlying projects (particularly Apache Arrow). A data scientist can now use all these tools and produce a dataframe that, as always, can be loaded into OmniSci via the `load_table` APIs in pymapd (and Ibis). The icing on the cake is OmniSci Immerse's Visual Data Fusion (VDF) feature announced in OmniSci 4.7, which lets a user set up charts (combo and multi-layer geo charts for now) across multiple tables/sources.

Putting it All Together

A key point bears repeating: We've developed each one of the capabilities we've discussed within the respective open source project communities rather than simply forcing them to develop their own. Our work on Ibis and Pandas is available to everyone involved with those communities. OmniSciDB itself is open source, as it has been for more than two years. We've also invested deeply in the packaging/installation aspects so that users can easily add OmniSci to their data science workflows in multiple ways.

We took care to package everything with Docker*. Our JupyterLab image includes all the tools you need to get started. As a matter of fact, you can download and try the whole setup, including OmniSciDB, on your Mac* or Linux* laptop. We use Anaconda* for Python package management, so you can download and install OmniSciDB itself from Conda forge with `conda install -c conda-forge omniscidb-cpu`, and then the PyData tools for OmniSci with `conda install -c conda-forge omnisci-pytools`. Alternatively, you can get a prebuilt version of OmniSci from our **downloads page**.

Looking Back, Looking Forward

Let’s give credit where it’s due. None of these new capabilities would’ve seen the light of day without the work and guidance of our open source collaborators, including those at Intel. Ultimately, we believe cutting-edge methods in data science—or, for that matter, the plumbing—are in the service of the user, not the other way around. As far as a consumer of insights is concerned, we think it’s better to have the entire assembly of tools become invisible, but ensure the insights and their explanations become obvious. Through our shared vision, we’re working to advance and unify the world of data science with traditional analytics on modern hardware.

A NEW APPROACH TO PARALLEL COMPUTING USING AUTOMATIC DIFFERENTIATION

Getting Top Performance on Modern Multicore Systems

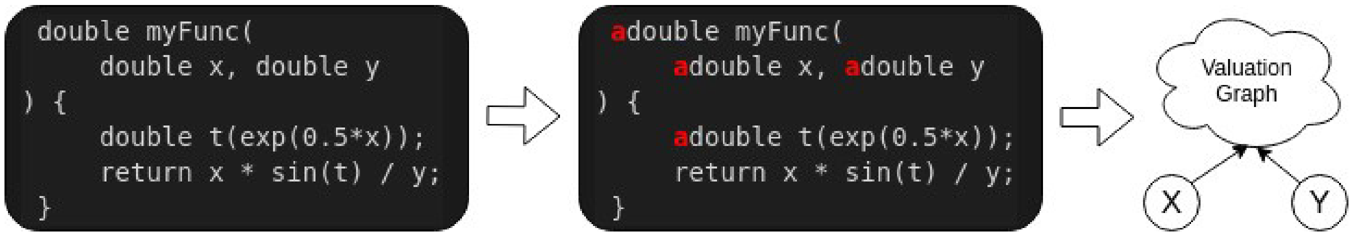
Dmitri Goloubentsev, Head of Automatic Adjoint Differentiation, Matlogica, and Evgeny Lakshtanov, Principal Researcher, Department of Mathematics, University of Aveiro, Portugal and Matlogica LTD

If you're interested in high-performance computing, high-level, object-oriented languages aren't the first things that come to mind. Object abstractions come with a runtime penalty and are often difficult for compilers to vectorize. Adapting your code for multithreading execution is a huge challenge, and the resulting code is often a headache to maintain.

You're in luck if performance-critical parts of your code are localized and can be flattened and safely parallelized. However, many performance-critical problems can benefit from object-oriented programming abstractions. We're proposing a different programming model that lets you achieve top performance on single instruction multiple data (SIMD), non-uniform memory access (NUMA) multicore systems.

Operator Overloading for Valuation Graph Extraction

We'll focus on problems where the same function, `F(1)`, needs to be executed on a data set `X[i]`. For example, let's look at Monte Carlo simulations in the finance world where `X[i]` is a random sample and `F(.)` is a pricing function (**Figure 1**). We use an **operator overloading pattern** to extract all primitive operations performed by `F(.)`.



```
double myFunc(
    double x, double y
) {
    double t(exp(0.5*x));
    return x * sin(t) / y;
}
```

```
adouble myFunc(
    adouble x, adouble y
) {
    adouble t(exp(0.5*x));
    return x * sin(t) / y;
}
```

1 **Example operator overloading pattern**

This pattern is very common in **automatic adjoint differentiation (AAD)** libraries. Unlike traditional AAD libraries, we don't build a data structure to represent the valuation graph. Instead, we compile binary machine code instructions to replicate valuations as defined in the graph, which can be seen as a just-in-time (JIT) compilation. However, we don't work with the source code directly. Instead, we compile a valuation graph produced by the user's algorithm. Since we want to apply `F(.)` to a large set of data points, we can compile this code to expand all scalar operations to full SIMD vector operations and process four (AVX2) or eight (AVX-512) data samples in parallel.

Learn by Example

Let's look at a simple option pricing framework where we use various abstract business objects. In this example, we simulate asset values as a random process:

```
template<class vtype>
vtype simulateAssetOneStep(
   const vtype current_value
   , Time current_t
   , Time next_t
   , const BankRate<vtype>& rate
   , const AssetVolatility<vtype>& vol_obj
   , const vtype& random_sample
) {
   double dt = (next_t-current_t);
   vtype vol=vol_obj(current_value, current_t);
   vtype next_value = current_value * (
       1 + (-vol*vol / 2+ rate(current_t))*dt
       +  vol * std::sqrt(dt) * random_sample
   );
   return next_value;
}
```

The classes `BankRate` and `AssetVolatility` can define different ways of computing model parameters, and implementation can be done deep in the derived classes. This function can be used with the native `double` type. When applied along timepoints, `t[i]` can be used to simulate asset value at the option expiry:

```
template<class vtype>
vtype onePathPricing (
   vtype asset
   , double strike
   , const std::vector<Time>& t
   , const BankRate<vtype>& rate_obj
   , const AssetVolatility<vtype>& vol_obj
   , const std::vector<vtype>& random_samples
) {
   for (int t_i = 0; t_i < t.size()-1; ++t_i) {
       asset = simulateAssetOneStep(asset, t[t_i], t[t_i+1], rate_obj,
vol_obj, random_samples[t_i]);
   }
   return std::max(asset - strike, 0.);
}
```

However, this leads to bad performance because the compiler can't effectively vectorize the code and business objects may contain virtual function calls. Using the AAD runtime compiler, we can execute the function, record one random path of asset evolution, and compute option intrinsic value at the expiry:

```
typedef __m256d mmType;  // __mm256d and __mm512d are supported

  int AVXsize = sizeof(mmType) / sizeof(double);

  aadc::AADCFunctions<mmType> aad_funcs;

  std::vector<idouble> random_samples(num_time_steps, 0.);
  idouble rate(0.03), vol(0.15), asset(100.0);
  aadc::VectorArg random_arg;

  aad_funcs.startRecording();
      // Mark vector of random variables as input only
      markVectorAsInput(random_arg, random_samples, false);

      // Mark rate, initial asset value and volatility as inputs
      aadc::AADCArgument rate_arg(rate.markAsInput());
      aadc::AADCArgument asset_arg(asset.markAsInput());
      aadc::AADCArgument vol_arg(vol.markAsInput());

      BankRate<idouble> rate_obj(rate);
      AssetVolatility<idouble> vol_obj(vol);

      idouble payoff= onePathPricing(asset, strike, t, rate_obj, vol_obj, random_samples);
      aadc::AADCResult payoff_arg(payoff.markAsOutput());
  aad_funcs.stopRecording();
```

At this stage, the `func` object contains compiled, vectorized machine code that replicates valuations to produce the final payoff output value given the arbitrary `random_samples` vector as input. The function object remains constant after recording and requires memory context for execution:

```
 // allocate memory needed for vectorized execution
shared_ptr<WorkSpace<mmType> > ws = func.createWorkSpace();

mmType mm_total_price(mmSetConst<mmType>(0.0));

// initialize function inputs
ws->val(rate_arg) = mmSetConst<mmType>(init_rate);
ws->val(asset_arg) = mmSetConst<mmType>(init_asset);
ws->val(vol_arg) = mmSetConst<mmType>(init_vol);

// run Monte Carlo loop
for (int mc_i = 0; mc_i < (num_mc_paths / AVXsize); ++mc_i) {
    vector<mmType> avx_random_samples(generateRandomAvxVector());
    // Set function arguments
    ws->setVector(random_samples_arg, avx_random_samples);

    // call the recorded function
    func.forward(ws);
    // get result
   mm_total_price=mmAdd(ws->val(payoff_arg), mm_total_price);
}
// calc avx vector element wise sum
cout << mmSum(mm_total_price) << endl;
```

Free Multithreading

Making efficient and safe multithreaded code can be difficult. Notice that the recording happens only for one input sample and can be executed in the controlled, stable, single-threaded environment. The resulting recorded function, however, is threadsafe and only needs separate workspace memory allocated for each thread. This is a very attractive property, since it lets us turn non-multi-thread-safe code into something that can be safely executed on multicore systems. Even optimal NUMA memory allocation becomes a trivial task. (You can view the full code listing for the multithreaded example **here**.)

Automatic Differentiation

This technique not only accelerates your function, it can also create an adjoint function to compute derivatives of all inputs with respect to all outputs. This is similar to the back-propagation algorithm used for deep neural network (DNN) training. Unlike DNN training libraries, this approach works for almost any arbitrary C++ code. To record an adjoint function, simply mark which input variables are required for differentiation:

```
// compute derivative w.r.t. initial value
AADC::Argument asset_arg(asset.markAsDiff());
```

Finally, to execute the adjoint function, initialize the gradient values of outputs and call the `reverse()` method on the function object:

```
ws->diff(payoff_arg) = mmSetConst<mmType>(1.0);

func.reverse(ws);

cout << "Derivative of dPrice/dSpot = " << ws->diff(asset_arg)[0] << endl;
```

Getting Top Performance

Hardware is evolving toward increasing parallelism with a lot more cores, wider vector registers, and accelerators. For object-oriented programmers, it's hard to adapt single-threaded code to existing parallel methods like OpenMP and CUDA. Using the AADC tool from Matlogica, programmers can turn their object-oriented, single-threaded, scalar code into AVX2/AVX512 vectorized, multithreaded, and threadsafe lambda functions. Crucially, the AADC tool can also generate a lambda function for the Adjoint method of computing, with all required derivatives using the same interface. Visit **Matlogica** for more details and a demo version of AAD-C.

Acknowledgements

Evgeny Lakshtanov is partially supported by Portuguese funds through the Center for Research and Development in Mathematics and Applications (CIDMA) and the Portuguese Foundation for Science and Technology (FCT, Fundação para a Ciência e a Tecnologia), within project UIDP/04106/2020.

8 RULES FOR PARALLEL PROGRAMMING FOR MULTICORE

There are Some Consistent Rules that can Help you Solve the Parallelism Challenge and Tap Into the Potential of Multicore

***James Reinders, Founding Editor and Editor Emeritus of* The Parallel Universe**

[Editor's Note: This article was originally published in The Parallel Universe *issue #1, back in April 2009. To celebrate our anniversary, we're reprinting it here to show that James' advice is still timely, relevant, and prescient 11 years later.]*

Programming for multicore processors poses new challenges. Here are eight rules for multicore programming to help you be successful.

Rule 1: Think Parallel

Approach all problems looking for the parallelism. Understand where parallelism is and organize your thinking to express it. Decide on the best parallel approach before other design or implementation decisions. Learn to "think parallel."

Rule 2: Program Using Abstraction

Focus on writing code to express parallelism, but avoid writing code to manage threads or processor cores. Libraries, OpenMP, and Threading Building Blocks are all examples of using abstractions. Do not use native threads (Pthreads*, Windows* threads, Boost* threads, and the like). Native thread libraries are the assembly languages for parallelism. They offer maximum flexibility but require too much time to write, debug, and maintain. Your programming should be at a high enough level that your code is about your problem, not about thread or core management.

Rule 3: Program in Tasks (Chores), Not Threads (Cores)

Leave the mapping of tasks to threads or processor cores as a distinctly separate operation in your program, preferably an abstraction you are using that handles thread/core management for you. Create an abundance of tasks in your program, or a task that can be spread across processor cores automatically (such as an OpenMP loop). By creating tasks, you are free to create as many as you can without worrying about oversubscription.

Rule 4: Design with the Option to Turn Concurrency Off

To make debugging simpler, create programs that can run without concurrency. This way, when debugging, you can run programs first with—then without—concurrency to see if both runs fail or not. Debugging common issues is simpler when the program is not running concurrently because it is more familiar and better supported by today's tools. Knowing that something fails only when run concurrently hints at the type of bug you are tracking down. If you ignore this rule and can't force your program to run in only one thread, you'll spend too much time debugging. Because you want to have the capability to run in a single thread specifically for debugging, it doesn't need to be efficient. You just need to avoid creating parallel programs that require concurrency to work correctly, such as many producer-consumer models.

Rule 5: Avoid Using Locks

Simply say "no" to locks. Locks slow programs, reduce scalability, and are the source of many bugs in parallel programs. Make implicit synchronization the solution for your program. If you still need explicit synchronization, use atomic operations. Use locks only as a last resort. Work hard to design programs that don't need locks.

Rule 6: Use Tools and Libraries Designed to Help with Concurrency

Don't "tough it out" with old tools. Be critical of tool support with regard to how it presents and interacts with parallelism. Most tools are not yet ready for parallelism. Look for thread-safe libraries—ideally, ones that are designed to utilize parallelism themselves.

Rule 7: Use Scalable Memory Allocators

Threaded programs need to use scalable memory allocators. Period. There are a number of solutions, and I'd guess that all of them are better than `malloc()`. Using scalable memory allocators speeds up applications by eliminating global bottlenecks, reusing memory within threads to better utilize caches, and partitioning properly to avoid cache line sharing.

Rule 8: Design to Scale Through Increased Workloads

The amount of work your program needs to handle increases over time. Plan for that. Designed with scaling in mind, your program will handle more work as the number of processor cores increases. Every year, we ask our computers to do more and more. Your designs should favor using increases in parallelism to give you advantages in handling bigger workloads in the future.

Getting the Most Out of Multicore Processors

I wrote these rules with implicit mention of threading everywhere. Only Rule 7 is specifically related to threading. Threading is not the only way to get value out of multicore. Running multiple programs or multiple processes is often used, especially in server applications.

These rules will help you get the most out of multicore processors. *Some will grow in importance over the next 10 years as the number of processor cores increases and we see an increase in the diversity of the cores themselves. The coming of heterogeneous processors and NUMA, for instance, will make Rule 3 more and more important. [Editor's note: The emphasis is mine. James was already thinking about heterogeneous parallelism in the first issue of* The Parallel Universe.*]*

You should understand all eight rules and take them to heart.

LIFT YOUR CODING TO THE NEXT LEVEL

It’s easier to build great things with our free code samples. To get started, just tell us your interest, tool, or hardware.

GET STARTED >

intel® Software

BOOK REVIEW: THE OPENMP* COMMON CORE

By Timothy G. Mattson, Yun (Helen) He, and Alice Konigs

Making OpenMP Simple Again

Review by Ruud van der Pas, Senior Principal Software Engineer, Oracle Corporation

OpenMP* provides an application programming interface (API) for shared-memory parallel computing. It consists of compiler directives, library routines, and environment variables. These are all available to the developer to define and control parallel execution. The OpenMP specification has evolved since 1997 to keep up with the trends in hardware architectures and programming languages. This is a positive development, but the current functionality and features can be overwhelming for the developer interested in getting started with OpenMP. The book *The OpenMP Common Core* by Timothy G. Mattson, Yun (Helen) He, and Alice Konigs helps address this issue by defining the idea of a "common core."

The common core, as the authors define it, is a compact subset of the OpenMP API, ideally suited to get new users started. The core functionality presented in this book is sufficient to parallelize many applications using OpenMP, but for those who require additional functionality, it provides pointers to additional reading material. While it's assumed the reader has basic programming skills in C, C++, or Fortran*, no background in parallel computing is required.

The book is organized in three parts:

1. Setting the Stage
2. The OpenMP Common Core
3. Beyond the Common Core

Setting the Stage

The first part covers the prerequisites, introducing and explaining the relevant concepts in parallel computing that readers need to understand what follows. An overview of various contemporary parallel architectures is included as well. The history of OpenMP is also presented, which is helpful to understand what OpenMP is today—and why a common core is needed. The inclusion of this first part is helpful for readers new to parallel computing in general and OpenMP in particular. It eliminates the need to consult other references and makes this a standalone book.

The OpenMP Common Core

With over 120 pages, the second part constitutes the bulk of the book. The authors discuss and define the OpenMP common core. They do a good job of clearly explaining the concepts and constructs that define this subset of the OpenMP API. Their approach is very practical, using examples to illustrate the features and show how to apply them. Some of these examples, like the numerical integration algorithm to approximate the value of π, are shown in full. These are small but fully functional programs that are important to demonstrate how to transform a sequential program into a correct OpenMP program. The devil is often in the details, so the authors cover potential pitfalls throughout the book.

The OpenMP data model is one of the harder things for newcomers to master. This is mostly because in a sequential application, you don't have to think about it. However, it's vital to shared-memory parallel programming. The coverage of this topic is nothing short of excellent. The authors help take away the fear and confusion of going from a single-threaded to multithreaded program. Even more experienced OpenMP developers may find this section useful.

There's another element that differentiates this book from others. Many OpenMP books don't go beyond parallel loops. For several classes of parallel algorithms, we need a different form of parallelism—one that's more asynchronous and dynamic. In OpenMP, support for this is provided through the concept of tasks. The book thoroughly covers OpenMP tasks, including several code examples. To my pleasant surprise, the data environment with tasks is also explained very clearly. This isn't an easy topic to understand, but the authors do a very good job of explaining it.

Every OpenMP developer must have a good understanding of the memory model. This is specific to OpenMP and a rather complex part of the specification. The chapter covering this topic is a true gem. It also clearly introduces memory consistency, another difficult topic for users new to shared-memory parallel programming. I haven't read a better description of these topics.

The second part ends with a recap of the OpenMP common core. Initially, I wondered whether this was redundant, but it isn't. Assuming readers have digested the previous chapters, this part provides a good reference. It makes it easier to look things up and to find details covered in earlier chapters. This is where it all comes together, and the dots are connected. Also, the authors introduce some nuances on the use of certain features here.

Beyond the Common Core

The third and last part of the book covers what developers may need beyond the constructs that constitute the OpenMP common core (e.g., additional clauses, more runtime functions, atomic operations, and locking). By design, the coverage is brief, but the authors include pointers to more information. You may wonder why these features aren't part of the common core. The reason is that not every developer needs these features. They're more specific to the algorithm being parallelized.

A relatively large section in this third part is dedicated to the important topic of non-uniform memory access (NUMA). Although NUMA is a performance feature and not related to writing a correct OpenMP program, it's good to see it covered in this book. NUMA is mentioned earlier in the book, but the surface is barely scratched. Here, the authors discuss it in detail, including what a contemporary memory system looks like and how this can affect the performance of an application. OpenMP has provided support for NUMA since version 4.0 of the specification was released in 2013. The features available to the user to control data placement and thread affinity are explained and illustrated with several examples.

The last part of this section provides useful pointers and more information. For instance, it explains how to read and understand the OpenMP specifications. The specs are available free of charge, but with over 600 pages in the current spec, it's quite a volume. It also targets the implementer rather than the end user. This can make the text hard to understand if you're not a compiler guru. Sometimes you need to check the specifications, though, and this section helps readers to find their way.

Not only novice OpenMP developers will find this book extremely useful. More experienced developers will find enough nuggets scattered throughout the chapters to warrant having a copy.

Software

THE PARALLEL UNIVERSE