GPU-Quicksort:
from OpenCL to Data Parallel C++

By Robert Ioffe, Senior Exascale Performance Software Engineer

March 9, 2020

Contents

Introduction

Intel introduced Data Parallel C++ (DPC++): a heterogeneous portable programming language based on the Khronos SYCL standard. DPC++ is a single source programming language that can target an array of platforms: CPUs, integrated and discrete GPUs, FPGAs and other accelerators. We will port a non-trivial OpenCL application, GPU-Quicksort, to DPC++ and document the experience. We also set the goal of exceeding the capabilities of the initial application: OpenCL C makes it very hard to write generic algorithms, and it becomes clear that it is a serious shortcoming when one tries to implement algorithms such as sorting, which need to work for different data types. The original GPU-Quicksort for OpenCL was written to sort unsigned integers. We demonstrate how to use templates with DPC++ and implement GPU-Quicksort for multiple data types. Finally, we port GPU-Quicksort to Windows and SLES to show DPC++ portability.

What is GPU-Quicksort?

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

What is OpenCL?

We start with the OpenCL 1.2 implementation. OpenCL is a Khronos standard for programming heterogeneous systems and is fully supported by Intel on a variety of the operating systems and platforms. OpenCL consists of the runtime, the host API and the device C-based programming language OpenCL C. Here lies both its power and its limitations: the power is the ability to write high performance portable heterogeneous applications, and the main limitation is the necessity to write and debug two separate codes: the host side and the device side, and the lack of templates and other C++ features that modern programmers got accustomed to, which make writing generic libraries in OpenCL hard.

What is Data Parallel C++?

Data Parallel C++ (DPC++) is an Intel implementation of Khronos SYCL, the standard designed to address the limitations of OpenCL outlined above. DPC++ provides both a single source programming model, which enables to have a single code base for host and device programming and the full use of C++ templates and template metaprogramming on the device without the loss of portability and a very minimal impact to performance. DPC++ allows a programmer to target CPUs, GPUs and FPGAs while permitting accelerator-specific tuning, so it is a definite improvement on OpenCL. It is also supported by the Intel tool, such as Intel® VTune™ Profiler and Intel® Advisor as well as GDB. We will take full advantage of DPC++, especially the its template features.

The Starting Point: Windows Apps from 2014

We 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” by the author. 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.

A brief overview of GPU-Quicksort architecture: it consists of two kernels: gqsort_kernel and lqsort_kernel, written in OpenCL 1.2, that 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 to run the kernels on, the size of the input and 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 platform and device, builds the kernels, properly allocates memory and creates buffers from it, binds them to the kernel arguments and then launches the dispatcher function.

Data Parallel C++/OpenCL Interoperability: Platform Initialization

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

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

Now, instead of initializing a platform, a device and a context and a queue the OpenCL way, we need to do it the 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 dpcpp 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 select 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 is to use DPC++ to set kernel arguments and launch kernels. The program is still built, and the kernels are obtained the OpenCL way. We use `cl::sycl::kernel` objects to wrap original OpenCL kernels, e.g.

```
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 (the example below shows gqsort_kernel, but 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 is 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 where the other three are created from an STL vector). We use template keyword in front of the `get_access` member function for buffers that we pass by reference. Note 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, e.g.

```
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 initialized the platform and the device, created the buffers, their accessors and bound them to the kernels and launched those kernel on the device in a DPC++ way, but we still 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 is loaded into the program at runtime before being built. We will change that starting with the gqsort_kernel, the simpler of the two kernels.

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 is a good idea to create a functor class. We are going to create a `gqsort_kernel_class` functor and make it templated right from the start to be able to handle sorting more that 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 1-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 uses specialized OpenCL functions, extensively uses 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 are trivially templatized). `plus_prescan` function that is used to calculate scan sums is relatively simple, so the only change that we will make to bring it to DPC++ is to make it a template function in preparation of making our sort a generic algorithm:

```
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: not only do we make it a template function, but we also need to replace OpenCL C `select` functions with DPC++ `cl::sycl::select` functions 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));
        }
    }
}
```

While in OpenCL C it is possible both to create local memory variables and arrays inside the body of the kernel and pass them as kernel parameters, 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 will 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 to our functor constructor along with global buffer accessors, and 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 in simplicity vs 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);
```

Atomics: here again DPC++ is not as elegant as OpenCL C, so 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 translated and templatized supporting structs and functions, converting specialized OpenCL C functions to DPC++, 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 `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 barriers use, which in DPC++ case require the iteration objects. They work on local and global memory pointers, which require special handling, so the following 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 other basic data types, e.g. floats and doubles. In addition to 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 `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 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 imf library at the link time. Windows port is slightly more complex: add the following definitions when compiling:

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

and account 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 it! We can sort unsigned integers, floats and doubles using Intel GPUs on Windows and two Linux flavors!

Conclusion

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 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 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 with which to debug, profile and analyze your DPC++ program. We hope that you enjoyed this article. Start using DPC++ in your projects!

References

1. Khronos OpenCL The open standard for parallel programming of heterogeneous systems
2. Khronos SYCL C++ Single-source Heterogeneous Programming for OpenCL
3. GPU-Quicksort: A practical Quicksort algorithm for graphics processors by Daniel Cederman and Philippas Tsigas
4. GPU-Quicksort in OpenCL 2.0: Nested Parallelism and Work-Group Scan Functions by Robert Ioffe
5. Intel® oneAPI Toolkits
6. Accompanying code: https://software.intel.com/en-us/article/code-for-parallel-universe-gpu-quicksort-for-dpcpp by Robert Ioffe