GPU Programming I

Andrey Alekseenko

KTH Royal Institute of Technology & SciLifeLab

Portable models: SYCL

- Open standard for accelerator programming
  - Royalty-free, vendor-agnostic, high-level
- Single-source model based on C++17
  - Requires compiler support and C++ knowledge
  - Content warning: lambda functions, templates, exceptions
- Interoperability with native features (at the expense of portability)

Standard vs. implementation

SYCL itself is only a *standard*, with several open-source *implementations*:

- Intel oneAPI DPC++ (a.k.a. Intel LLVM)
  - Supports Intel GPUs natively, and NVIDIA and AMD GPUs with Codeplay oneAPI plugins
  - Also CPUs and FPGAs
- AdaptiveCpp (a.k.a. hipSYCL, Open SYCL): supports AMD, Intel, NVIDIA GPUs
  - Also CPUs
  - MooreThreads support contributed
- ComputeCPP, triSYCL, motorSYCL, SYCLops, Sylkan, …

Typical GPU application

1. Select/initialize the device
2. Allocate memory on the device
3. Copy input data from host to device
4. Launch kernel on the device
5. Copy output from device to host
6. Free allocated memory

Error checking: in SYCL, exceptions are thrown

sycl::queue

- A way to submit tasks (kernels, memory copies) to be executed on the device
- Can also be used as a handle for memory management
- Tasks are launched asynchronously!

```cpp
#include <sycl/sycl.hpp>

int main() {
  // Create an out-of-order queue on the default device
  sycl::queue q;
  // Now we can submit tasks to q!
}
```

Initialization

- Queue is associated with a device
- There can be multiple queues, on one or multiple devices
- Queues can have properties

```
// Iterate over all available devices (including CPUs)
for (const auto &device : sycl::device::get_devices()) {
  std::cout << "Creating a queue on " \
            << device.get_info<sycl::info::device::name>() << "\n";
  sycl::queue q(device, {sycl::property::queue::in_order()});
  // ...
}
```

Getting started on Dardel (lazy)

```
$ ssh abcd@dardel.pdc.kth.se
$ ml PDC/22.06 adaptivecpp/23.10.0-cpeGNU-22.06-rocm-5.3.3-llvm
$ export SLURM_ACCOUNT=edu23.aqti SLURM_TIMELIMIT=00:05:00
$ export SLURM_PARTITION=gpu SLURM_RESERVATION=XYZ
$ srun acpp-info -l
=================Backend information===================
Loaded backend 0: OpenMP
  Found device: hipSYCL OpenMP host device
Loaded backend 1: HIP
  Found device: AMD MI250X
  Found device: AMD MI250X
  Found device: AMD MI250X
  Found device: AMD MI250X
```

Getting started on Dardel (explicit)

```
$ ssh abcd@dardel.pdc.kth.se
$ ml PDC/22.06 adaptivecpp/23.10.0-cpeGNU-22.06-rocm-5.3.3-llvm
$ srun -Aedu23.aqti -t 1:00 -pgpu --reservation X acpp-info -l
===================Backend information====================
Loaded backend 0: OpenMP
  Found device: hipSYCL OpenMP host device
Loaded backend 1: HIP
  Found device: AMD MI250X
  Found device: AMD MI250X
  Found device: AMD MI250X
  Found device: AMD MI250X
```

Getting started on TCBLab (interactive mode)

```
$ ssh-copy-id wsXX@login.tcblab.org # Password: Aq2023$MudCow
$ ssh wsXX@login.tcblab.org
$ salloc
salloc: Nodes gpuYY are ready for job
$ ssh gpuYY
$ module load adaptivecpp/23.10.0-clang16-cuda12.1
$ acpp-info -l
=================Backend information===================
Loaded backend 0: CUDA
  Found device: NVIDIA RTX A5000
Loaded backend 1: OpenMP
  Found device: hipSYCL OpenMP host device
```

list_devices.cpp

```cpp
#include <iostream>
#include <sycl/sycl.hpp>

int main() {
  std::vector<sycl::device> all_gpus =
      sycl::device::get_devices(sycl::info::device_type::gpu);
  for (const auto &device : all_gpus) {
    std::cout << "Found device " << device.get_info<sycl::info::device::name>() << "\n";
    std::cout << "    It has "
              << device.get_info<sycl::info::device::global_mem_size>() / 1024 / 1024
              << " MiB of memory\n";
    // Now we can create a queue and submit tasks to the device
    // sycl::queue q(device, {sycl::property::queue::in_order()});
  }
  return 0;
}
```

Exercise: build and run `list_devices` (TCBLab)

```
$ cp /mnt/cephfs/home/aqtivate-ws/* ./

$ acpp -O3 list_devices.cpp -o list_devices

$ ./list_devices
Found device NVIDIA RTX A5000
    It has 24247 MiB of memory
```

Exercise: build and run `list_devices` (Dardel)

```
$ cp /cfs/klemming/home/a/andreyal/Public/* ./

$ acpp -O3 list_devices.cpp -o list_devices

$ srun ./list_devices
Found device
    It has 65520 MiB of memory
Found device
    It has 65520 MiB of memory
...
```

Typical GPU application

1. Select/initialize the device ✓
2. Allocate memory on the device
3. Copy input data from host to device
4. Launch kernel on the device
5. Copy output from device to host
6. Free allocated memory

Programming paradigms

USM

- Raw pointers.
- Manual data movement, allocation, synchronization.
- Works best with *in-order* queues.
- Ideal for translating CUDA/HIP code.
- Three kinds: device, host, shared.
- More control of the execution.

Buffer-accessor

- Define data-dependency graph through data access.
- Automatic data movement, resource allocation, synchronization.
- Works best with *out-of-order* queues.
- Allows more optimizations by the runtime.
    - Currently, runtimes are not stellar.

Programming paradigms

USM

- Raw pointers.
- Manual data movement, allocation, synchronization.
- Works best with *in-order* queues.
- Ideal for translating CUDA/HIP code.
- Three kinds: device, host, shared.
- More control of the execution.

Buffer-accessor

- Define data-dependency graph through data access.
- Automatic data movement, resource allocation, synchronization.
- Works best with *out-of-order* queues.
- Allows more optimizations by the runtime.
  - Currently, runtimes are not stellar.

Allocate and copy memory (USM)

```
sycl::queue q{{sycl::property::queue::in_order()}};
int n = //...
int* arr_host = //...

// Allocate `n` integers on a device associated with `q`
int* arr_device = sycl::malloc_device<int>(n, q);

// Copy data from `arr_host` to `arr_device`
q.copy<int>(arr_host, arr_device, n);
```

Allocate and copy memory (USM)

```
sycl::queue q{{sycl::property::queue::in_order()}};
int n = //...
int* arr_host = //...

// Allocate `n` integers on a device associated with `q`
int* arr_device = sycl::malloc_device<int>(n, q);

// Copy data from `arr_host` to `arr_device`
q.copy<int>(arr_host, arr_device, n);
// Copy is not done yet!

q.wait(); // Wait for all operations to finish

// Tidy up: free memory when it's no longer needed
sycl::free(arr_device, q);
```

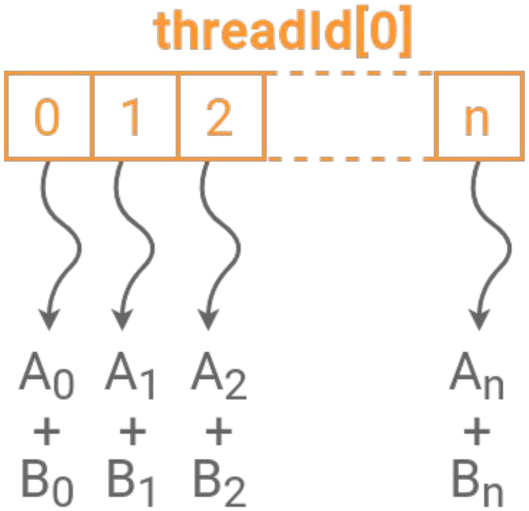
Typical GPU application

1. Select/initialize the device ✓
2. Allocate memory on the device ✓
3. Copy input data from host to device ✓
4. **Launch kernel on the device**
5. Copy output from device to host ✓
6. Free allocated memory ✓

Launching kernels

```
sycl::range<1> global_size(n);

q.submit([&](sycl::handler &h) {
  h.parallel_for<class VectorAdd>(global_size,
    [=](sycl::item<1> threadId) {
      int tid = threadId[0];
      Cd[tid] = Ad[tid] + Bd[tid];
    }
  );
});
```



Getting it all together: Vector addition

```
sycl::queue q{{sycl::property::queue::in_order()}};

float *Ad = sycl::malloc_device<float>(N, q); // Allocate the arrays on GPU
q.copy<float>(Ah.data(), Ad, N); // Copy the input data from host to the device

sycl::range<1> global_size(N); // Define grid dimensions
// Run our kernel
q.submit([&](sycl::handler &h) {
  h.parallel_for<class VectorAdd>(global_size, [=](sycl::item<1> threadId) {
    int tid = threadId[0]; // Get thread index
    Cd[tid] = Ad[tid] + Bd[tid]; // Do the math
  });
});

q.copy<float>(Cd, Ch.data(), N); // Copy results back to the host
// All the operations before were asynchronous!
q.wait(); // Wait for the copy to finish
sycl::free(Ad, q); // Free the GPU memory

// Work with Ch
```

Exercise: run vector addition (USM)

```
$ acpp -O3 vector_add.cpp -o vector_add
```

```
$ ./vector_add # Don’t forget srun on Dardel
Running on NVIDIA GeForce GTX 1080
A =           { 0   1.93538  ... 2.28384   1.463    }
B =           { 1.1 0.594333 ... 0.130176 -0.848779 }
A + B (GPU) = { 1.1 2.52972  ... 2.41401   0.614221 }
Total error: 0
```

Look at the source code. Try modifying it:

- Compute vector subtraction
- Compute rolling sum: `C[i] = A[i-1] + A[i] + A[i+1]`

Programming paradigms: USM (device/host)

```
sycl::queue q{{sycl::property::queue::in_order()}};
// Create a device allocation of n integers
int* v = sycl::malloc_device<int>(n, q);
// Submit a kernel into a queue; cgh is a helper object
q.submit([&](sycl::handler &cgh) {
  // Define a kernel: n threads execute the following lambda
  cgh.parallel_for<class Kernel>(sycl::range<1>{n}, [=](sycl::item<1> i) {
    // The data is directly written to v
    v[i] = /*...*/
  });
});
// If we want to access v, we should copy it to CPU
q.copy<int>(v, v_host, n).wait(); // and wait for it!
// After we're done, the memory must be deallocated
sycl::free(v, q);
```

Programming paradigms: USM (shared)

```
sycl::queue q{{sycl::property::queue::in_order()}};
// Create a shared (migratable) allocation of n integers
int* v = sycl::malloc_shared<int>(n, q);
// Submit a kernel into a queue; cgh is a helper object
q.submit([&](sycl::handler &cgh) {
  // Define a kernel: n threads execute the following lambda
  cgh.parallel_for<class Kernel>(sycl::range<1>{n}, [=](sycl::item<1> i) {
    // The data is directly written to v
    v[i] = /*...*/
  });
});
// If we want to access v, we have to ensure that the kernel has finished
q.wait();
// After we're done, the memory must be deallocated
sycl::free(v, q);
```

Programming paradigms: Buffer-accessor

```
sycl::queue q; // out-of-order by default
// Create a buffer of n integers
auto buf = sycl::buffer<int>(sycl::range<1>(n));
// Submit a kernel into a queue; cgh is a helper object
q.submit([&](sycl::handler &cgh) {
  // Create write-only accessor for buf
  auto acc = buf.get_access<sycl::access_mode::write>(cgh);
  // Define a kernel: n threads execute the following lambda
  cgh.parallel_for<class Kernel>(sycl::range<1>{n}, [=](sycl::item<1> i) {
    // The data is written to the buffer via acc
    acc[i] = /*...*/
  });
});
/* If we now submit another kernel with accessor to buf, it will not
 * start running until the kernel above is done */
```

Exercise: run vector addition (buffers)

```
$ acpp -O3 vector_add_buffers.cpp -o vector_add_buffers
```

```
$ ./vector_add_buffers # Don't forget srun on Dardel
Running on NVIDIA GeForce GTX 1080
A =           { 0   1.93538  ... 2.28384   1.463    }
B =           { 1.1 0.594333 ... 0.130176 -0.848779 }
A + B (GPU) = { 1.1 2.52972  ... 2.41401   0.614221 }
Total error: 0
```

Look at the source code. Try modifying it:

- Get rid of an extra buffer and store the result in A: `A[i] = A[i] + B[i]`
  - Use `sycl::access_mode::read`

Exercise: Try Intel oneAPI compiler (only on TCBLab)

```
$ module unload adaptivecpp
$ module load intel-oneapi/2023.2.1 cuda/12.1

$ clang++ -fsycl -fsycl-targets=nvidia_gpu_sm_86 \
         vector_add.cpp -o vector_add

$ ./vector_add

Running on NVIDIA RTX A5000
A = { 0 1.93538 ... 2.28384 1.463 }
B = { 1.1 0.594333 ... 0.130176 -0.848779 }
A + B (GPU) = { 1.1 2.52972 ... 2.41401 0.614221 }
Total error: 0
```

Exercise: Matrix transpose

- Build and run `transpose_matrix_v0.cpp`
- For now, it just copies the matrix
- Look at the source code
  - Many new constructs there! More on them tomorrow
- Modify the code to transpose the matrix