Offloading to GPU with SYCL

Andrey Alekseenko

Science for Life Laboratory, KTH Royal Institute of Technology, Sweden

Introduction to GPU, Stockholm/Zoom, 22 September, 2023

Presentation based on the ENCCS lesson "Portable kernel-based models"

SYCL

SYCL is a royalty-free, open-standard C++ programming model for multi-device programming.

High-level, single-source, vendor-agnostic programming model for heterogeneous systems, including GPUs.

Often implemented on top of other backends (CUDA/HIP), with interoperability support: can use native functions and libraries from SYCL code, native tools.

Relatively high-level, but the developers are still required to write GPU kernels explicitly.

Warning: required C++ knowledge

- Lambda functions
- Templates
- Exceptions
- How variable scopes work

Standard vs. implementation

SYCL itself is only a *standard*, for which several open-source *implementations* exist.

- 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 NVIDIA and AMD GPUs, with pre-release Intel GPU support and possible MooreThreads support. Also CPUs.
- ComputeCPP, triSYCL, motorSYCL, SYCLops, Sylkan, ...

None are fully standard compliant, but things are getting better.

Programming model

SYCL is based on C++-17. Modern C++ features such as templates and lambdas are heavily used.

SYCL is primarily *kernel-based* model, but also includes some typical algorithms (reductions, *etc*.).

SYCL supports both automatic memory/dependency management (*buffer-accessor* model) and direct memory operations (*USM* model).

SYCL concepts

Underlying hardware is the same!

HIP/CUDA Term	SYCL Term	Approximate meaning
Thread	Work item	Single thread of work
Group	Work group	Group of threads with access to the SLM
Warp/Wavefront	Sub-group	Group of threads running in ~lockstep
Shared memory	Local memory	Fast memory shared by threads in a work group
Registers	Private memory	Per-thread fast memory

Initialization

`sycl::queue`: a way to submit tasks to be executed on the device.

Compared to `hipStream`:

- `sycl::queue` can be in-order or out-of-order; `hipStream` is always in-order
- `sycl::queue` must be explicitly created; in HIP, there is a "default" stream
- `sycl::queue` is not limited to GPUs

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

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

Initialization

```cpp
// 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()});
  // ...
}
```

- `sycl::device::get_devices(sycl::info::device_type::gpu)` to skip CPUs
- Or just create a queue with the default device!
- Respects `ROCR_VISIBLE_DEVICES` environment variable

Programming models

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 models: 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 KernelName>(sycl::range<1>{n}, [=](sycl::id<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 */
```

Programming models: 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 KernelName>(sycl::range<1>{n}, [=](sycl::id<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 models: 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 KernelName>(sycl::range<1>{n}, [=](sycl::id<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);
```

Choosing programming model

- USM device/host is similar to classical HIP/CUDA, close to hardware, best performance if done correctly
- USM shared is similar to HIP/CUDA with managed memory, relies on driver, performance usually good
- Buffers are more descriptive, less need for memory management and syncronizations, performance mixed

HIP vs SYCL

```
__global__ void vector_add(
           float *A, float *B, float *C, int n) {
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  if (tid < n) {
    C[tid] = A[tid] + B[tid];
  }
}
```

```
static auto vector_add_kernel(
            float* A, float* B, float* C, int n)
{
  return [=](sycl::id<1> tid) {
    if (tid < n) {
      C[tid] = A[tid] + B[tid];
    }
  };
}
```

```
// Allocate GPU memory, ...
hipMalloc((void**)&ad, n * sizeof(float));
// Copy the data from CPU to GPU, ...
hipMemcpy(ad, ah, sizeof(float) * n,
          hipMemcpyHostToDevice);
```

```
// Create queue
sycl::queue queue{{sycl::property::queue::in_order()}};
// Allocate GPU memory, ...
float* A = sycl::malloc_device<float>(n, queue);
// Copy the data from CPU to GPU, ...
queue.copy<float>(ad, ah, n);
```

```
dim3 blocks{256, 1, 1};
dim3 threads{(n + 255) / 256, 1, 1};
vector_add<<<blocks, threads>>>(ad, bd, cd, n);
```

```
q.submit([&](sycl::handler &h) {
   h.parallel_for<class VectorAdd>({n},
      vector_add_kernel(Ad, Bd, Cd, N));
});
```

Built-in functions: reduction

```
// Create a buffer for sum to get the reduction results
sycl::buffer<int> bufSum{&sum, 1};

// Submit a SYCL kernel into a queue
q.submit([&](sycl::handler &cgh) {
  // Create temporary object describing variables with reduction semantics
  auto accSum = bufSum.get_access<sycl::access_mode::read_write>(cgh);
  // We can use built-in reduction primitive
  auto reductionSum = sycl::reduction(accSum, sycl::plus<int>());

  // A reference to the reducer is passed to the lambda
  cgh.parallel_for(sycl::range<1>{n}, reductionSum,
                   [=](sycl::id<1> idx, auto &reducer) {
                     reducer.combine(idx[0]);
                   });
});
```

- Might not always be very efficient, but easy to use

Exercise 1: Dot product with SYCL

Build and test run a SYCL program that calculates the dot product of vectors.

- Load the necessary modules (only AdaptiveCpp/hipSYCL available):
  - `ml PDC/22.06 hipsycl/0.9.4-cpeGNU-22.06-rocm-5.3.3` (Dardel)
  - `module use /appl/local/csc/modulefiles && ml hipsycl/0.9.4` (LUMI)
- Download the source code
  - `wget https://raw.githubusercontent.com/PDC-support/introduction-to-gpu/main/examples/sycl/dot_sycl_usm.cpp`
- Compile the code on the login node
  - `syclcc -O2 dot_sycl_usm.cpp -o dot_sycl_usm`

Run the code as a batch job

- Edit `job_gpu_dot_sycl_usm.sh` to specify the compute project and reservation:

```
#!/bin/bash
#SBATCH -A edu23.introgpu # Set the allocation to be charged for this job
#SBATCH -J myjob # Name of the job
#SBATCH -p gpu # The partition
#SBATCH -t 00:05:00 # 5 minutes wall-clock time
#SBATCH --nodes=1 # Number of nodes
#SBATCH --ntasks-per-node=1 # Number of MPI processes per node
ml PDC/22.06 hipsycl/0.9.4-cpeGNU-22.06-rocm-5.3.3
srun ./dot_sycl_usm > output.txt # Run the executable
```

- Submit the script with `sbatch job_gpu_dot_sycl_usm.sh`
- Wait for the job to complete (`squeue -u $(whoami)`)
- Verify that `output.txt` contains `sum = 1.250000`

Optionally, test the code in interactive session.

- First queue to get one GPU node reserved for 10 minutes
  - `salloc -N 1 -t 0:10:00 -A <project name> -p gpu`
- Wait for a node, then run the program `srun -n 1 ./dot_sycl_usm`
- The output will be printed to the standard output`

Optionally, check out the version with accessors:

- `wget https://raw.githubusercontent.com/PDC-support/introduction-to-gpu/blob/main/examples/sycl/dot_sycl_accessors.cpp`
- `syclcc -O2 dot_sycl_accessors.cpp -o ./dot_sycl_accessors`
- Run interactively or via batch job

Exercise 2: Matrix transpose

- `wget https://raw.githubusercontent.com/PDC-support/introduction-to-gpu/main/examples/sycl/transpose_sycl.cpp`
- Add missing data management functions (see TODO)
- `syclcc -O2 ./transpose_sycl.cpp -o transpose_sycl`
- Run interactively or via batch job
- There are two versions of the kernel available: `transposeKernelSimple` and `transposeKernelLocal`. Edit the source to use `transposeKernelLocal`, recompile, check performance.

References

- Intel oneAPI DPC++
- AdaptiveCpp / hipSYCL / Open SYCL
- SYCL2020 standard
- ENCCS GPU course: Portable kernel-based models
- Intel oneAPI Student Ambassador program