An Introduction to SYCL and OneAPI

2 June 2023 | Jan H. Meinke

Motivation

19 of top 25 supercomputers are accelerated using GPUs

Used to be (almost) exclusively Nvidia GPUs but things have changed

x of the 17 systems now have AMD GPUs

Aurora (ANL) is finally coming online and the next top 10 list should have GPUs from at least three vendors

=> Portable GPU programming becomes vital

A Side Note

> **Note**
>
> Ironically, the ubiquity of GPU codes written in CUDA has driven AMD and Intel to provide tools that convert CUDA C++ code to HIP (hipify, AMD) or SYCL (SYCLomatic, Intel). This makes CUDA not a bad choice to start development of a GPU program due to its good documentation and excellent tool support assuming you have an Nvidia GPU available and you stay away from the newest features of CUDA.

Language Standards

pSTL provides parallel algorithms as part of the standard library since C++ 17

Fortran's `do concurrent` expresses a parallel loop. Its `matmul` operation maps directly to BLAS routines

Nvidia's HPC SDK supports parallel features in both languages on the GPU.

Intel's DPL implements the pSTL and can run on AMD, Intel, and Nvidia GPUs (and other accelerators as well).

OpenMP/OpenACC

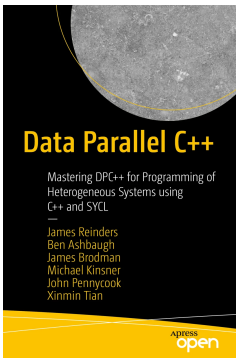
Annotate loops with pragmas to offload calculations to the GPU

Libraries

- Kokkos
- Raja
- **SYCL**
- ...

... includes HPX and many other C++ libraries that abstract the GPU programming model.

SYCL



Data Parallel C++,
Download

Open standard (Khronos)

Single source

SYCL

```
using namespace sycl;
using namespace std::numbers;

auto main() -> int {
   std::vector v(10'000'000, 0.0);
   queue q;
   {
      buffer v_buf{v};
      q.submit([&](handler& h){
            accessor v{v_buf, h, write_only, noinit};
            h.parallel_for(v.size(), [=](auto i){
               v[i] = 2 * pi * i / v.size();
            });
      });
   }
}
```

Not just for GPUs

General model for heterogeneous computing (CPU, GPU, FPGA, ...)

Can create different queues for different devices

Not necessarily performance portable

Queues

Every SYCL program needs at least one `queue`.

Often the default queue is sufficient

```
sycl::queue q;
```

Sometimes we want to be more specific

```
sycl::queue q_cpu{cpu_selector{}};
sycl::queue q_gpu{gpu_selector{}};
```

We can also create a custom device selector if we need something even more specific.

Exercise: Setup

Prepare:

- load CUDA

  ```
  ml CUDA
  ```

- load Intel OneAPI:

  ```
  source /p/usersoftware/paj1720/intel/oneapi/setvars.sh --include-intel-llvm
  ```

Check for device:

- run `sycl-ls` on a compute node with GPUs

Exercise: Queues

file: syclqueues.cpp

Request a CPU device and a GPU explicitly instead of using the default device. Compile the code on the login node with

```
clang++ -std=c++20 -fsycl -fsycl-targets=nvptx64-nvidia-cuda sysclqueues.cpp -o syclqueues
```

Run the code on a GPU node.

Compare the output.

Query *at least* one more value. You can find a list of possible values for `get_info` here.

Many of the values that can be queried are useful for fine tuning an algorithm for a specific device. They can also be used to write a custom device selector.

Buffers

Buffers are responsible for managing data.

They can have one, two, or three dimensions

Buffers are used to determine dependencies between kernel calls

Buffers are often initialized from host container (vector, array)

```
auto N{1'000'000};
std::vector v(N, 0.0};
sycl::buffer buf{v};
```

Multidimensional Buffer

```
sycl::buffer<double, 2> buf{v, sycl::range<2>{1000, 1000}};
sycl::buffer<double, 3> buf{sycl::range{100, 100, 10}};
```

Accessors

Buffers cannot be accessed directly

Create an accessor instead

```
sycl::accessor accA{bufA, h};  // read-write accessor
sycl::accessor accB{bufB, h, sycl::read_only}  // read-only accessor
```

(I will talk about `h` later).

Only one accessor can access a buffer at a time

Use `sycl::host_accessor` to access a buffer in the host program

```
for(auto i = 0; i < buf.size(); ++i){
    sycl::host_accessor acc{buf};
    std::cout << acc[i] << ' ';
}  // acc goes out of scope
```

Kernels

Kernels are executed for each item in the index space

Work is usually implemented using lambda functions

```
queue q;
buffer buf;
q.submit([&](handler& h){
    accessor acc(buf, h);
    h.parallel_for(buf.get_range(), [=](auto idx){
      acc[idx] = 3.1415;
    });
}
```

Handler

While a buffer is accessible from any device, accessors belong to a particular one. The handler provides as with a reference to the current submission.

Index Ranges

A kernel is executed for every element of an index range.

The range can be multidimensional

```
h.parallel_for(sycl::range{1000, 1000}, [=](sycl::id<2> idx){
    // Use 2D idx to access 2D buffer
    accA[idx] = 0;
    // extract index components for traditional 2D access
    auto i = idx[0];
    auto j = idx[1];
    for(auto k = i; k < j; ++k){
        accA[i][j]++
    }
});
```

id

The `id` represents the index of the current work item.

It knows nothing more than its coordinates

item

If our kernel accepts an `item`, it can get information about the range.

```
h.parallel_for(sycl::range{1000, 1000}, [=](sycl::item<2> it){
   // Use 2D idx to access 2D buffer
   accA[it.get_id()] = 0;
   // extract index components for traditional 2D access
   auto i = it[0];
   auto j = it[1];
   accA[i][j] = it.get_linear_id();
});
```

Exercise

file: fill_buffer.cpp (new)

Create a two dimensional buffer `buf` and write a kernel that fills it with the values $\sin(x)\cos(y)$ where $x \in [0, 2\pi]$ and $y \in [0, 2\pi]$.

Compile the program and run it on a GPU node.

Exercise: Simple matrix multiplication

file: simple_matmul.cpp (new)

Create 2 2D buffer.

Fill them with uniformly distributed random numbers between 0 and 1 on the host.

Write a simple matrix multiplication kernel and check the performance on the GPU.

ND-Range Kernels

So far we left the distribution of the range completely to the runtime.

CUDA programmers are used to specifying the block size:

```
sycl::queue q;
sycl::range global{N, N};
sycl::range local{16, 16};  // N must be a multiple of the block size
sycl::buffer<double, 2> bufA{global};
...
q.submit([&](handler& h){
    sycl::accessor accA{bufA, read_only};
    ...
    h.parallel_for(sycl::nd_range{global, local}, [=](sycl::nd_item<2> it){
        int i = it.get_global_id(0); int j = it.get_global_id(1);
        for (int k = 0; k < N; k++) accC[i][j] += accA[i][k] * accB[k][j];
    });
});
```

Exercise

Use an explicit ND-Range kernel for your matrix multiplication.

Local Accessors

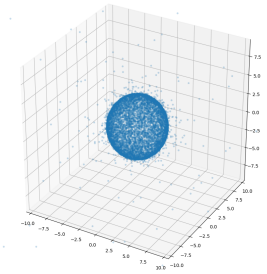
A work group can share fast local memory.

It's like a programmable cache

```
sycl::local_accessor<double, 2> tile{local, h};

h.parallel_for(sycl::nd_range<2>{global, local}, [=](sycl::nd_item<2> it){
    // Read an item from global memory into the tile
    ...
    // Synchronize work group before moving on
    it.barrier();
    ...
}
```

Particle Dynamics



$$\mathbf{F} = m\mathbf{a}$$

$$\mathbf{F}_{ij} = G\frac{m_i m_j}{r_{ij}^3}\mathbf{r}_{ij}$$

$$\mathbf{r}(t+\delta t) = \mathbf{r}(t) + \mathbf{v}(t)\delta t + \frac{1}{2}\mathbf{a}(t)\delta t^2$$

$$\mathbf{v}(t+\delta t) = \mathbf{v}(t) + \frac{\mathbf{a}(t)+\mathbf{a}(t+\delta t)}{2}\delta t$$