Learning SYCL by porting CUDA codes

Zheming Jin

Overview

- Motivation
- Background
- Porting experience
  - Not about automatic porting
  - Specific to SYCL buffers
  - Not comprehensive; get started
  - No performance comparison
    - Evaluating the Performance of the hipSYCL Toolchain for HPC Kernels on NVIDIA V100 GPUs
- Summary

Motivation

- Major SYCL features
  - Extension to OpenCL C
  - Single source (combine kernel/host programs)
  - Two coding styles (USM and/or Buffer)
  - Asynchronous programming (overlap kernel/host programs)
  - Portability (functional and performance)
  - Productivity

Background - SYCL

Step	OpenCL	SYCL
1	Platform query	Device selector class
2	Device query of a platform	
3	Create context for devices	
4	Create command queue for context	Queue class
5	Create memory objects	Buffer class
6	Create program object	Lambda expressions
7	Build a program	
8	Create kernel(s)	
9	Set kernel arguments	
10	Enqueue a kernel object for execution	Submit a SYCL kernel to a queue
11	Transfer data from device to host	Implicit via accessors
12	Event handling	Event class
13	Release resources	Implicit via destructor

My porting work

- Developing a repository of SYCL programs
  - For the development of SYCL compilers
  - Written with SYCL buffers
- Variety
  - Proxy/mini applications
  - Benchmarks
  - Machine learning
- Open source
  - https://github.com/zjin-lcf/oneAPI-DirectProgramming/
- Flat hierarchy
  - At most 3 levels for almost all programs

Porting experience: get started

- We don't need to be a C++ expert
  - Familiarity with CUDA and SYCL
- Need a computer
  - AMD, Intel, Nvidia graphics
  - Emulation of a CUDA program on a CPU ?
- Need a SYCL compiler
  - ComputeCpp (CPU/GPU)
  - DPC++ (CPU/GPU/FPGA)
  - HipSYCL (CPU/GPU)
  - triSYCL (FPGA)
  - neoSYCL (CPU/NEC)
  - CCE Compiler (Ascend AI)

SYCL buffer allocation

```
// CUDA codes

float *d_data;
cudaMalloc((void**)&d_data, size_byte);
…
…
cudaFree(d_data)
```

```
buffer<float, 1> d_data (size_elem);
```

SYCL buffer allocation and initialization

```
// CUDA codes

float *d_data;
cudaMalloc((void**)&d_data, size_byte);
cudaMemcpy(d_data, h_data, size_byte,
                        cudaMemcpyHostToDevice);
…
…
cudaFree(d_data)
```

```
buffer<float, 1> d_data (h_data, size_elem);
```

host-to-device copy

```
// CUDA codes

for (int i = 0; i < N ; i++)
   cudaMemcpyAsync(d_data, h_data, size_byte,
                                   cudaMemcpyHostToDevice);
```

```
for (int i = 0; i < N ; i++)
  q.submit([&] (handler& cgh) {
     auto acc = d_data.get_access<sycl_write>(cgh);
     cgh.copy(h_data, acc);
   });
```

device-to-host copy

```
// CUDA codes

cudaMemcpyAsync(h_data, d_data, size_byte,
                                cudaMemcopyDeviceToHost);
```

```
q.submit([&] (handler& cgh) {
   auto acc = d_data.get_access<sycl_read>(cgh);
   cgh.copy(acc, h_data);
 });
```

set data in a device memory

```
// CUDA codes

float *d_data;
…
cudaMemset(d_data, 0, size_byte);
```

```
q.submit([&] (handler& cgh) {
   auto acc = d_data.get_access<sycl_write>(cgh);
   cgh.fill(acc, 0.f);
 });
```

launch a kernel

```
// CUDA codes

dim3 grid(32);
dim3 block(256);
kernel<<<grid, block, bytes>(d_data);
```

```
range<1> gws (256*32);  // or range<3> gws (1, 1, 256*32)
range<1> lws (256);         // or range<3> lws (1, 1, 256)
q.submit([&] (handler& cgh) {
   auto acc = d_data.get_access<…>(cgh);
   accessor<float, 1, sycl_read_write, access::target::local> sm(elems, cgh);
   cgh.parallel_for(nd_range<1>(gws, lws), [=] (nd_item<1> item) {
      kernel(acc.get_pointer(), sm.get_pointer(), item);
   });
});
```

identifiers in a kernel

```
// CUDA codes

__global__ void kernel ( … )
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  …
}
```

```
void kernel (…, nd_item<1> &item)
{
  int i = item.get_global_id(0);
  …
  item.get_local_id(0);          // threadIdx.x
  item.get_local_range(0);       // blockDim.x
  item.get_group(0);             // blockIdx.x
  item.get_group_range(0);       // gridDim.x
}
```

shared local memory and barrier

```
// CUDA codes

__global__ void kernel ( … )
{
  extern __shared__ float smem[];
  …
  __syncthreads();
  …
}
```

```
void kernel (…, float* smem, nd_item<1> &item)
{
  …
  item.barrier(…);
  …
}
```

atomics & math functions

```
// CUDA codes

__global__ void kernel ( float *input, float *result )
{
  …
  atomicAdd(result, sqrtf(input[i]);
}
```

```
void kernel (float *input, float *result, …)
{
  …
  atomic_ref<float,
                memory_order::relaxed,
                memory_scope::device,
                 access::address_space::global_space> ao (result[0]);
   ao += sycl::sqrt(input[i]);
}
```

vector access

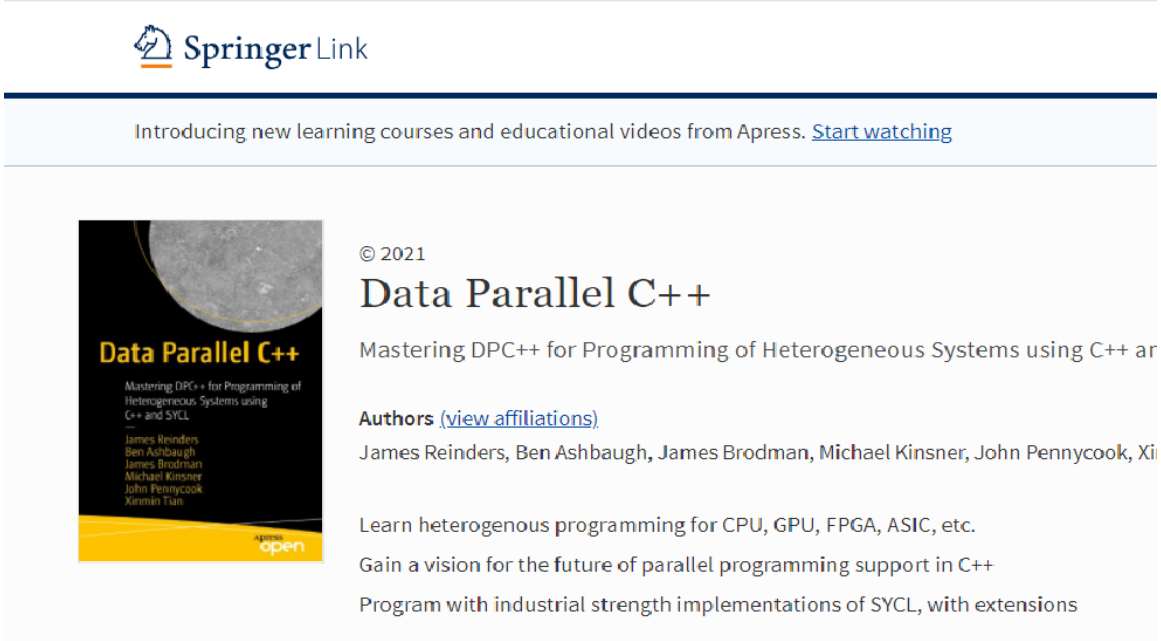
```
// CUDA codes

__global__ void kernel ( char *input, … )
{
   char4 p = reinterpret_cast<char4*>(input)[i];
    r = p.x;
   g = p.y;
   b = p.z;
    …
}
```

```
void kernel (char *input, …)
{
    vec<char, 4> v;
    v.load(i, input);
     r = v.x();
    g = v.y();
    b = v.z();
}
```

Summary

- Not all CUDA features have SYCL equivalents
- Intel® Data Parallel Compatibility Tool may be your assistant
- The SYCL specifications: https://www.khronos.org/sycl/
- The SYCL book (~390K downloads)



Acknowledgement

- SYCL developers
- This research used resources of the Experimental Computing Laboratory (ExCL) at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.