SYCL Tutorial

July 20, 2023

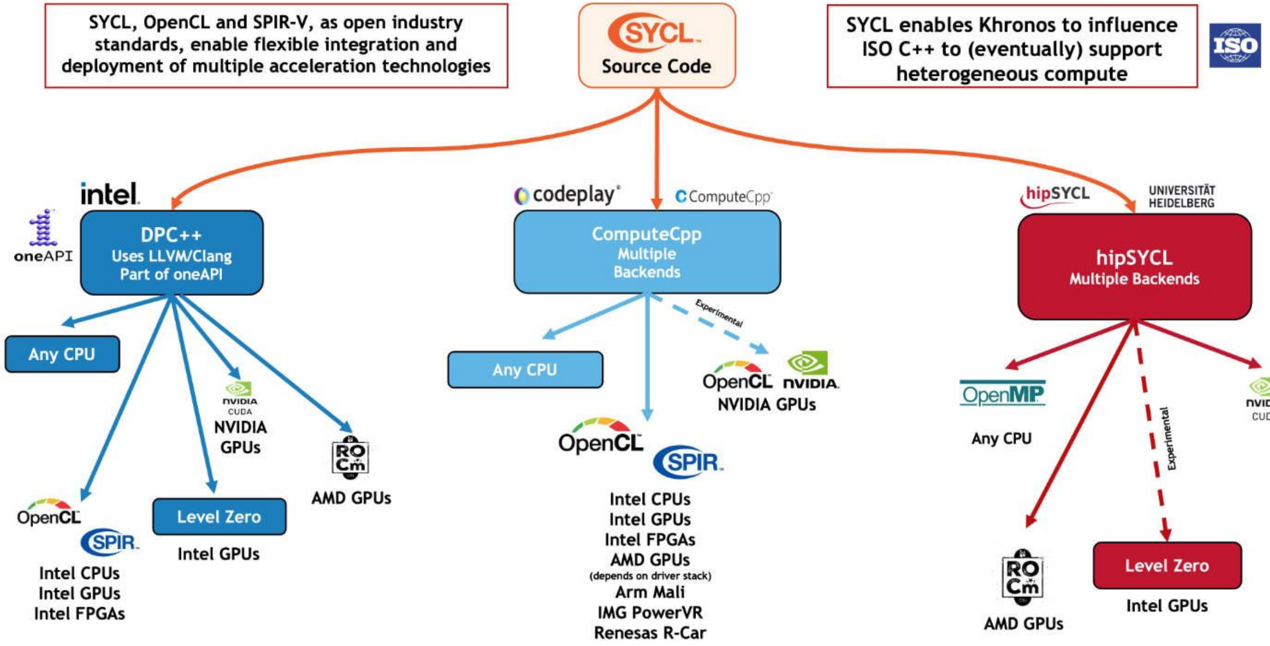
What is SYCL

- ❖ Open-standard parallel programming framework that can target a range of hardware accelerators (CPUs, GPUs, FPGA’s).

- ❖ SYCL is a C++ library that functions as an OpenCL abstraction layer
  - ➢ There is also ongoing development to support CUDA, HIP, and ROCm in the backend

SYCL Implementations

- SYCL Implementations include
  - ComputeCpp, DPC++, hipSYCL, neoSYCL, and triSYCL



Compiling SYCL Code

- Compiling SYCL for Different GPUs (Intel support builds)
  - https://www.intel.com/content/www/us/en/developer/articles/technical/compiling-sycl-with-different-gpus.html

- Compile SYCL with Jupyter Notebook on Intel Dev Cloud (Account Required).
  - https://devcloud.intel.com/oneapi/get_started/baseTrainingModules/

- Compile SYCL with interactive online tool provided by ComputeCPP (deprecated version of SYCL)
  - https://tech.io/playgrounds/48226/introduction-to-sycl/introduction-to-sycl-2

Tutorial Module 1: Queues

In SYCL, we refer to the queue as the selection of a device (GPU) that allows the scheduling and execution of kernels (submitted tasks).

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

// output device information
template<typename Queue_type>
void output_device_information(Queue_type& Q){
  std::cout << "DEVICE NAME: "
            << Q.get_device().template get_info<sycl::info::device::name>()
            << "\nDEVICE VENDOR: "
            << Q.get_device().template get_info<sycl::info::device::vendor>()
            << "\n" << std::endl;
}

int main(){
  // selecting a default device for queue
  sycl::queue Q{sycl::default_selector_v};
  output_device_information(Q);
}
```

```
DEVICE NAME: Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz
DEVICE VENDOR: Intel(R) Corporation
```

➔ **We can select the default devices available on our system.**

Tutorial Module 1: Queues

SYCL also has the capability to inquire about available platforms and devices for queue. This allows explicit device selection.

```cpp
// selecting a device based on platform and device number
sycl::queue get_queue(int platform_index = 0, int device_index = 0){
  // get the available platforms
  auto platforms = sycl::platform::get_platforms();

  // select the platform based on the platform index
  auto selected_platform = platforms[platform_index];

  // get the devices on the selected platform
  auto devices = selected_platform.get_devices();
  auto selected_device = devices[device_index];

  // create the queue based on the selected device
  sycl::queue q(selected_device);
  return q;
}
```

➔ **Select platform**

➔ **Select device**

Tutorial Module 2: Vector Addition (Hello World)

NOTE:
Host = CPU
Device = GPU

❖ Vector addition is essentially “Hello World” in the realm of GPGPU programming!

```
// copying data from host to device
Q.memcpy(A_device, &A_host[0], N*sizeof(double));
Q.memcpy(B_device, &B_host[0], N*sizeof(double));
Q.wait();

// executing the kernel
auto event = Q.submit([&](sycl::handler& h){
  h.parallel_for(sycl::range<1>(N), [=](sycl::id<1> idx){
    C_device[idx] = A_device[idx] + B_device[idx];
  });
});

event.wait();

// copying data from device to host
Q.memcpy(&C_host[0], C_device, N*sizeof(double)).wait();
```

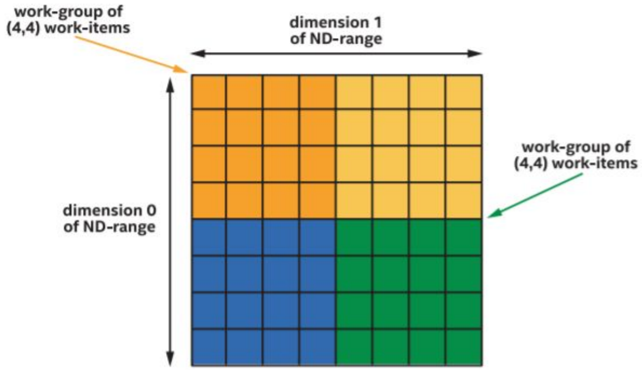
→ Copy host to device

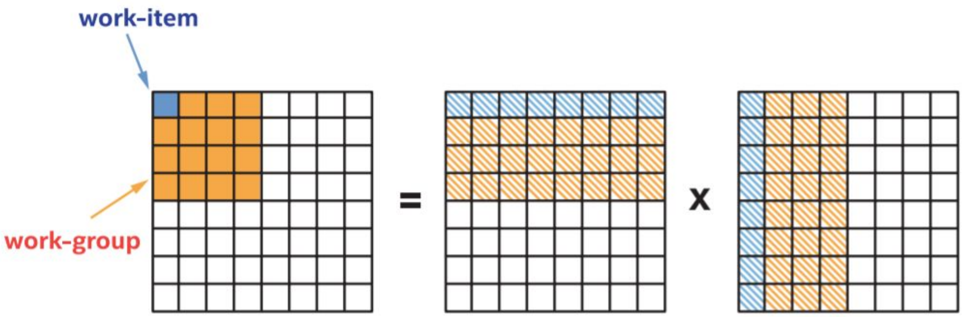
→ Parallel Vector Add

→ Copy device to host

Tutorial Module 3: Matrix Multiplication

- Matrix multiplication is another great example when introducing the advantages of GPGPU programming.
  - Embarrassingly parallel and expensive task for a CPU
  - Easy to add optimizations at the hardware level (ex: ND-Range)





Tutorial Module 3: Matrix Multiplication

❖ Comparing a basic matrix multiplication kernel and an ndrange kernel

```
// executing the kernel
auto event = Q.submit([&](sycl::handler& h){
  h.parallel_for(sycl::range<2>{M, P}, [=](sycl::id<2> idx){
    const int i = idx[0];
    const int j = idx[1];

    double c_ij = 0.0;

    for(int k = 0; k < N; ++k){
      c_ij += A_device[i*N + k] * B_device[k*P + j];
    }

    C_device[i*P + j] = c_ij;
  });
});
```

➔ **Basic matmul kernel**

```
auto event = Q.submit([&](sycl::handler& h){
  // global range and local work group size
  auto global = sycl::range<2>(M, P);
  auto local = sycl::range<2>(b, b);
  h.parallel_for(sycl::nd_range<2>(global, local), [=](sycl::nd_item<2> it){
    const int i = it.get_global_id(0);
    const int j = it.get_global_id(1);

    double c_ij = 0.0;

    for(int k = 0; k < N; ++k){
      c_ij += A[i*N + k] * B[k*P + j];
    }

    C[i*P + j] = c_ij;
  });
});
```

➔ **ND-Range matmul kernel**