Data Parallel C++ (DPC++)
Programming Model

Abhishek Bagusetty
Performance Engineering Group
Argonne Leadership Computing Facility
abagusetty@anl.gov
SYCL – Specification

• SYCL is “not” a programming model but a “language specification”
  – Heuristics looks similar to OpenCL-C bindings
  – C++ single source (co-exists host and device source code)
  – Two distinct memory models (USM and/or Buffer)
  – Asynchronous programming (overlaps device-compute, copy, host operations)
  – Portability (functional and performance)
  – Productivity
Data Parallel C++ (DPC++)

Intel’s oneAPI Implementation of SYCL = C++ and SYCL*
standard and extensions

Based on modern C++
✓ C++ productivity features and familiar constructs

Standards-based, cross-architecture
✓ Incorporates the SYCL standard for data parallelism and heterogeneous programming
SYCL* extensions

Productivity

- Simple things should be simple to express
- Reduce verbosity and programmer burden enhance performance

• Give programmers control over program execution
• Enable hardware-specific features

Fast-moving open collaboration feeding into the SYCL* standard

✓ Open source implementation with goal of upstream LLVM
✓ Extensions aim to become core SYCL*, or Khronos* extensions
SYCL – A Portable Programming Model

A C++-based programming model for intra-node parallelism

- SYCL is a specification and “not” an implementation, currently compliant to C++17 ISO standards
- Cross-platform abstraction layer, heavily backed by industry
- Open-source, vendor agonistic
- Single-source model
SYCL – Compiler Players

SYCL, OpenCL and SPIR-V, as open industry standards, enable flexible integration and deployment of multiple acceleration technologies

SYCL enables Khronos to influence ISO C++ to (eventually) support heterogeneous compute
SYCL @ NERSC

- Collaboration between ALCF, NERSC and Codeplay to enable support for NVIDIA A100 GPUs in LLVM DPC++/SYCL2020

- Initial scope of work complete
  - support for tensor cores, USM, atomics, and more available

- Current focus on performance, upstreaming features to LLVM, tracking library support (e.g. FFT, oneMKL)

**PrgEnv-llvm for CPE**
NERSC has developed an additional PrgEnv which adds to the Cray Programming Environment (CPE) that HPE provides.
- LLVM compiler with support for OpenMP offload, SYCL

Credits: Brandon Cook (NERSC)
AN INTRODUCTION TO PROGRAMMING WITH SYCL ON PERLMUTTER AND BEYOND, MARCH 1, 2022

Introduction

SYCL is an open standard programming model that allows developers to use standard C++ code to program for a range of GPUs and other accelerator processors. This means that it is possible to develop using modern C++ code and target Nvidia, AMD and Intel GPUs from a single code base. To enable SYCL on the latest supercomputers, Codeplay has been working in partnership with different National Laboratories to bring SYCL support to Perlmutter, Polaris and Frontier.

Join engineers from Codeplay for a half day hands-on workshop that will walk through the fundamentals of SYCL programming using practical examples and exercises to help reinforce the learning. Attendees will also learn how to compile their SYCL code using the DPC++ compiler to target Nvidia GPUs including those on the Perlmutter supercomputer. Lastly, we’ll talk about some of the things you need to know to achieve good performance, including best practices for memory management, with free time for questions and discussions.

ALCF and OLCF users are welcome to this training. NERSC training accounts will be provided if needed.

Workshop Leader: Hugh Delaney, Software Engineer, Codeplay Software

Course Outline

- Introduction
Queues & Contexts

• “SYCL Queues” provide mechanism to submit work to a device
• “SYCL Contexts” is well known to be over-looked

\[
\text{sycl::queue Que; // implicitly creates a SYCL context}
\]

• **Context** (aka cuContext)
  • Contexts are used for resources isolation and sharing
  • A SYCL context may consist of one or multiple devices
  • Memory created can be shared only if their associated queue(s) are created using the same context

• **Queue** (aka CUDA Stream)
  - Executes “asynchronously” from host code
  - SYCL queue can execute tasks enqueued in either “in-order” or “out-of-order (default)"
  - SYCL queue (in-order) is similar to CUDA stream (FIFO)
Bring You Own Compiler – Perlmutter

Download the compiler:
git clone -b sycl https://github.com/intel/llvm

Build & Install: (takes a while)
module load cudatoolkit/11.5
export DPCPP_HOME=$HOME

cd llvm
export CUDA_LIB_PATH=/opt/nvidia/hpc_sdk/Linux_x86_64/21.11/cuda/lib64/stubs
CC=`which gcc` CXX=`which g++` python $DPCPP_HOME/llvm/buildbot/configure.py --cuda --cmake-gen="Unix Makefiles" --cmake-opt="-DCUDA_TOOLKIT_ROOT_DIR=/opt/nvidia/hpc_sdk/Linux_x86_64/21.11/cuda/11.5"

python $DPCPP_HOME/llvm/buildbot/compile.py

Where are my SYCL compilers installed?
train515@nid001608:~/llvm/build/bin>
Porting from CUDA to SYCL
Execution Model: CUDA vs SYCL

<table>
<thead>
<tr>
<th>CUDA</th>
<th>SYCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread</td>
<td>work-item</td>
</tr>
<tr>
<td>warp</td>
<td>sub-group</td>
</tr>
<tr>
<td>block</td>
<td>work-group</td>
</tr>
<tr>
<td>grid</td>
<td>nd-range</td>
</tr>
</tbody>
</table>

Sub-groups are subset of the work-items that are executed simultaneously or with additional scheduling guarantees.

Leveraging sub-groups will help to map execution to low-level hardware and may help in achieving higher performance.
Why use SYCL - sub groups?

Sub-Group = subset of work-items within a work-group.

A subset of work-items within a work-group that execute with additional guarantees and often map to SIMD hardware.

- Work-items in a sub-group can communicate directly using shuffle operations, without repeated access to local or global memory, and may provide better performance.
- Work-items in a sub-group have access to sub-group collectives, providing fast implementations of common parallel patterns.
## Memory Model: CUDA vs SYCL

<table>
<thead>
<tr>
<th></th>
<th>CUDA</th>
<th>SYCL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Memory Type</strong></td>
<td><strong>Scope</strong></td>
<td><strong>Memory Type</strong></td>
</tr>
<tr>
<td>Register memory</td>
<td>Thread</td>
<td>Private memory</td>
</tr>
<tr>
<td>Shared memory</td>
<td>Block</td>
<td>Local memory</td>
</tr>
<tr>
<td>Global memory</td>
<td>Grid (all threads)</td>
<td>Global memory</td>
</tr>
</tbody>
</table>

### Memory Allocation Types

<table>
<thead>
<tr>
<th>Allocation Type</th>
<th>Initial Location</th>
<th>Accessible By</th>
<th>Migratable To</th>
</tr>
</thead>
<tbody>
<tr>
<td>device</td>
<td>device</td>
<td>host</td>
<td>No</td>
</tr>
<tr>
<td>device</td>
<td>device</td>
<td>host</td>
<td>host</td>
</tr>
<tr>
<td>device</td>
<td>device</td>
<td>device</td>
<td>N/A</td>
</tr>
<tr>
<td>Another device</td>
<td>Optional (P2P)</td>
<td>Another device</td>
<td>No</td>
</tr>
<tr>
<td>host</td>
<td>host</td>
<td>host</td>
<td>N/A</td>
</tr>
<tr>
<td>Any device</td>
<td>device</td>
<td>device</td>
<td>No</td>
</tr>
<tr>
<td>shared</td>
<td>Unspecified</td>
<td>host</td>
<td>Yes</td>
</tr>
<tr>
<td>device</td>
<td>device</td>
<td>device</td>
<td>Yes</td>
</tr>
<tr>
<td>Another device</td>
<td>Optional</td>
<td>Another device</td>
<td>Optional</td>
</tr>
</tbody>
</table>

Memory Model: Global Memory

<table>
<thead>
<tr>
<th>CUDA</th>
<th>SYCL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Memory Type</strong></td>
<td><strong>Scope</strong></td>
</tr>
<tr>
<td>Register memory</td>
<td>Thread</td>
</tr>
<tr>
<td>Shared memory</td>
<td>Block</td>
</tr>
<tr>
<td><strong>Global memory</strong></td>
<td>Grid (all threads)</td>
</tr>
</tbody>
</table>

// allocating device memory

```c
float *A_dev;
cudaMalloc((void **)&A_dev, array_size * sizeof(float));
```

> SYCL’s Global/Device allocated memory is only valid on the device

> More importantly not accessible from host
# Vector Addition: SYCL Buffer memory model

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

void main() {
    using namespace sycl;
    float A[1024], B[1024], C[1024];
    {
        buffer<float, 1> bufA { A, range<1> {1024} };
        buffer<float, 1> bufB { B, range<1> {1024} };
        buffer<float, 1> bufC { C, range<1> {1024} };

        queue myQueue;
        myQueue.submit([&](handler & cgh) {
            auto accA = bufA.get_access<access::read>(cgh);
            auto accB = bufB.get_access<access::read>(cgh);
            auto accC = bufC.get_access<access::write>(cgh);

            cgh.parallel_for<class vector_add>(range<1> {1024}, [=](id<1> i) {
                accC[i] = accA[i] + accB[i];
            });
        }).wait();
    }

    for (int i = 0; i < 1024; i++)
        std::cout << "C[" << i << "] = " << C[i] << std::endl;
}
```

- **Host Code**: Create SYCL buffers using host pointers.
- **Device Code**: Create a queue to submit work to a GPU.
- **Device Code**: Read/write accessors create dependencies if other kernels or host access buffers.
- **Host Code**: Vector addition device kernel.
```cpp
#include <sycl/sycl.hpp>
#include <iostream>

void main() {
    float A[1024], B[1024], C[1024];
    // initialize A, B, C with values on host
    sycl::queue myQueue;
    float* devA = sycl::malloc_device<float>(1024, myQueue);
    float* devB = sycl::malloc_device<float>(1024, myQueue);
    float* devC = sycl::malloc_device<float>(1024, myQueue);
    myQueue.memcpy(devA, A, 1024 * sizeof(float));
    myQueue.memcpy(devB, B, 1024 * sizeof(float));
    myQueue.parallel_for<class vector_add>(range<1> {1024}, [=](id<1> i) {
        devC[i] = devA[i] + devB[i];
    });
    myQueue.memcpy(C, devC, 1024 * sizeof(float));
    for (int i = 0; i < 1024; i++)
        std::cout << "C[" << i << "] = " << C[i] << std::endl;
}
```
```
#include <sycl/sycl.hpp>
#include <iostream>

void main() {
float A[1024], B[1024], C[1024];
// initialize A, B, C with values on host
sycl::queue myQueue;

float* devA = sycl::malloc_device<float>(1024, myQueue);
float* devB = sycl::malloc_device<float>(1024, myQueue);
float* devC = sycl::malloc_device<float>(1024, myQueue);

myQueue.memcpy(devA, A, 1024 * sizeof(float));
myQueue.memcpy(devB, B, 1024 * sizeof(float));

myQueue.parallel_for<class vector_add>(range<1> {1024}, [=](id<1> i) {
    devC[i] = devA[i] + devB[i];
});

myQueue.memcpy(C, devC, 1024 * sizeof(float));

for (int i = 0; i < 1024; i++)
    std::cout << "C[" << i << "] = " << C[i] << std::endl;
}
```

**Vector Addition: SYCL USM memory model**

SYCL queue (by-default) is out-of-order. (i.e., the execution starts when possible. Duty of programmer to assure correct dependencies

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

void main()
{
    float A[1024], B[1024], C[1024];
    // initialize A, B, C with values on host

    sycl::queue myQueue(sycl::property_list{sycl::property::queue::in_order{}});

    float* devA = sycl::malloc_device<float>(1024, myQueue);
    float* devB = sycl::malloc_device<float>(1024, myQueue);
    float* devC = sycl::malloc_device<float>(1024, myQueue);

    myQueue.memcpy(devA, A, 1024 * sizeof(float));
    myQueue.memcpy(devB, B, 1024 * sizeof(float));

    myQueue.parallel_for<class vector_add>(range<1>{1024}, [=](id<1> i) {
        devC[i] = devA[i] + devB[i];
    });

    myQueue.memcpy(C, devC, 1024 * sizeof(float));

    for (int i = 0; i < 1024; i++)
        std::cout << "C[" << i << "] = " << C[i] << std::endl;
}

SYCL queue (in-order) i.e., FIFO like cudaStream_t

myQueue.wait(), wait for D2H to complete before printing "C"
Tools: How to port existing CUDA to SYCL?

Intel® DPC++ Compatibility Tool
Assist in migrating CUDA applications to SYCL/DPC++, extending user choices

Additional Resources:

SYCLomatic: A “open-source” New CUDA*-to-SYCL* Code Migration Tool
https://github.com/oneapi-src/SYCLomatic
Math Libraries: What are my options for cublas,cu*?

- open-source implementation of the oneMKL Data Parallel C++ (DPC++) interface
- works with multiple devices (backends) uses vendor device-specific libraries underneath

Note: Apart of device-backend, supports host-CPU interface: Intel MKL, NETLIB

<table>
<thead>
<tr>
<th></th>
<th>NVIDIA</th>
<th>AMD</th>
<th>Intel</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLAS</td>
<td>cuBLAS</td>
<td>rocBLAS</td>
<td>oneMKL</td>
</tr>
<tr>
<td>Linear Solvers</td>
<td>cuSOLVER</td>
<td>(rocSOLVER)</td>
<td>oneMKL</td>
</tr>
<tr>
<td>Random Numbers</td>
<td>cuRAND</td>
<td>rocRAND</td>
<td>oneMKL</td>
</tr>
<tr>
<td>FFT</td>
<td>(cuFFT)</td>
<td>(rocFFT)</td>
<td>(oneMKL)</td>
</tr>
</tbody>
</table>

(work-in-progress)
Questions