LA-UR-19-26386
Approved for public release; distribution is unlimited.

Title: SYCL for Monte Carlo Transport

Author(s): Burke, Timothy Patrick

Intended for: Report

Issued: 2019-07-08

SYCL for Monte Carlo Transport

Timothy P. Burke

XCP-3 Monte Carlo Codes, Methods, and Applications
Los Alamos National Laboratory

SYCL

"SYCL is a cross-platform abstraction layer that builds on the underlying concepts, portability and efficiency of OpenCL that enables code for heterogeneous processors to be written in a "single-source" style using completely standard C++."

- AMD GPUs
- Nvidia GPUs
- ARMs
- Intel Graphics (on-chip graphics)
- Anything that uses OpenCL

"Completely standard"

SYCL is a standard specified by Khronos that is implemented by Codeplay (ComputeCPP) and intel

- This talk focuses on Codeplay's implementation of the standard
- Codeplay has implemented SYCL in an number of projects, including Tensorflow and Eigen (link)

SYCL Overview

SYCL provides an API for executing algorithms in parallel using parallel_for and buffers

- parallel_for iterates over indices in N dimensions

```
constexpr size_t N = 2000;
constexpr size_t M = 3000;
. . .
cgh.parallel_for<class foo>(range<2> {N, M}, [=](id<2> index){
  std::cout << "(i, j) (" << index[0] << "," << index[1] << ")\n");
});
```

- Buffers hold allocatable-data to be used in SYCL kernels

```
constexpr size_t N = 1000;
std::vector<double> data{N};
cl::sycl::buffer<double, 1> dataBuffer{data.data(), cl::sycl::range<1>{N}
```

  - SYCL manages the location of the data (host or device memory)
- Work is submitted to the device using queues and command groups

SYCL Basics - Adding two vectors

```
1  ...
2    size_t N = 100;
3    std::vector<T> VA(N, 1.0);
4    std::vector<T> VB(N, 2.0);
5    std::vector<T> VC(N);
6    cl::sycl::range<1> numOfItems{N};
7    { // scoping curly brackets
8      cl::sycl::buffer<T, 1> bufferA(VA.data(), numOfItems);
9      cl::sycl::buffer<T, 1> bufferB(VB.data(), numOfItems);
10     cl::sycl::buffer<T, 1> bufferC(VC.data(), numOfItems);
11     auto queueFunc = [&](cl::sycl::handler& cgh) {
12       auto accessorA = bufferA.template get_access<cl::sycl::access::mode::
       read>(cgh);
13       auto accessorB = bufferB.template get_access<cl::sycl::access::mode::
       read>(cgh);
14       auto accessorC = bufferC.template get_access<cl::sycl::access::mode::
       read_write>(cgh);
15     auto parallelFunc = [=] (cl::sycl::id<1> wiID){
16         accessorC[wiID] = accessorA[wiID] + accessorB[wiID];
17       };
18       cgh.parallel_for<class SimpleVadd>(numOfItems, parallelFunc);
19     };
20     cl::sycl::queue deviceQueue(cl::sycl::gpu_selector{});
21     deviceQueue.submit(queueFunc);
22   } // scoping curly brackets
23   for (auto& val : VC) std::cout << val << "\n";
24 ...
```

SYCL Basics

- Create buffers from pre-allocated host data

```
1  std::vector<T> VA(N, 1.0);
2  cl::sycl::buffer<T, 1> bufferA(VA.data(), numOfItems);
```

- Create a sycl queue targeting the device you want to run on

```
1  cl::sycl::queue deviceQueue(cl::sycl::gpu_selector{});
```

- Create and submit a functor to the queue

```
1  deviceQueue.submit(queueFunc);
```

SYCL Basics

- Queue functor
  - takes a SYCL control group handle as an argument

```
auto queueFunc = [&](cl::sycl::handler& cgh) {
```

  - Creates accessors to the sycl buffers

```
auto accessorA = bufferA.template get_access<cl::sycl::
    access::mode::read>(cgh);
```

  - Creates a functor to do the parallel work

```
auto parallelFunc = [=] (cl::sycl::id<1> wiID){
accessorC[wiID] = accessorA[wiID] + accessorB[wiID]; };
```

  - Calls parallel_for

```
  cgh.parallel_for<class SimpleVadd>(numOfItems,
    parallelFunc);
```

SYCL Gotchas

- Buffers synchronize with the data they were created from when they go out of scope
  - but not if they were constructed from iterators!

```
{ // scoping curly brackets
    cl::sycl::buffer<T, 1> bufferA(VA.data(), numOfItems);
    cl::sycl::buffer<T, 1> bufferB(VB.begin(), VB.end());
... // work on bufferA and bufferB
} // end scoping curly brackets
for (auto& val : VA) {
  std::cout << val << "\n"; // reflects updates to VA
}
for (auto& val : VB) {
  std::cout << val << "\n"; // VB unchanged!
}
```

SYCL Gotchas

- Accessors are created with the control group handle (cgh) as an argument - they must be created from within a queue submission.

```
1 auto queueFunc = [&](cl::sycl::handler& cgh) {
2   auto accessorA = bufferA.template get_access<cl::sycl::access
      ::mode::read>(cgh);
3 ... };
```

- The functor submitted to parallel-for can only capture by value and is copied to the device being executed on

```
1 auto parallelFunc = [=] (cl::sycl::id<1> wiID){
2 accessorC[wiID] = accessorA[wiID] + accessorB[wiID]; };
```

SYCL Gotchas

- Buffers and accessors only way to access heap data

```
1  struct Foo {
2    heap_container_t data;
3    ... // methods that access or operate on Foo data
4  };
5  ...
6  auto queueFunctor = [&] (cl::sycl::handler& cgh){
7    cgh.parallel_for(cl::sycl::range(N), [=] cl::sycl::id<1> id){
8      // how to use Foo in here?
9    });
10 };
```

SYCL Gotchas

- Buffers and accessors only way to access heap data

```
1  struct FooView {
2    Buffer* bufferPtr;
3    Accessor acc;
4    FooView(Buffer& buffer): bufferPtr(&buffer){}
5    void initialize(cl::sycl::handler& cgh){
6      acc = Accessor(*bufferPtr, cgh);
7    }
8    void bar(cl::sycl::id<1> id) {return acc[id];}
9  };
10 Foo foo;
11 Buffer bufferFooData(foo.data(), range(foo.size()));
12 FooView fooView(bufferFooData);
13 ...
14 auto queueFunctor = [&] (cl::sycl::handler& cgh){
15   fooView.initialize(cgh);
16   cgh.parallel_for(cl::sycl::range(N), [=] cl::sycl::id<1> id){
17     fooView.bar(id);
18   });
19 };
```

SYCL Gotchas

- Buffers and accessors only way to access heap data

```
1  struct FooView {
2    Buffer* bufferPtr;
3    Accessor acc;
4    FooView(Buffer& buffer): bufferPtr(&buffer){}
5    void initialize(cl::sycl::handler& cgh){
6      acc = Accessor(*bufferPtr, cgh);
7    }
8    void bar(cl::sycl::id<1> id) {return acc[id];}
9  };
10 Foo foo;
11 Buffer bufferFooData(foo.data(), range(foo.size()));
12 FooView fooView(bufferFooData);
13 ...
14 auto queueFunctor = [&] (cl::sycl::handler& cgh){
15   fooView.initialize(cgh);
16   cgh.parallel_for(cl::sycl::range(N), [=] cl::sycl::id<1> id){
17     fooView.bar(id);
18   });
19 };
```

What about a vector of Foo?

SYCL Gotchas

Objects in kernels must be standard layout type (cppref)

```
1  int main() {
2    class A { public: double a = 0.0; };
3    class B { public: int b = 0.0; };
4    class C : public A, public B { };
5    ...
6    q.submit([&](cl::sycl::handler &cgh) {
7      auto acc = buf.template get_access<access::mode::read_write
       >(cgh);
8      cgh.parallel_for<class ForEach>(Range(N), [=](cl::sycl::id
       <1> id) {
9        accC[id].a = id*0.6;
10       accC[id].b = id;
11     });
12   });
13 }
```

SyclParallelSTL

https://github.com/KhronosGroup/SyclParallelSTL

```cpp
#include <vector>
#include <iostream>
#include <sycl/execution_policy>
#include <experimental/algorithm>
using namespace std::experimental::parallel;
int main() {
  using T = double;
  int N = 100;
  std::vector<T> A(N, 1.0);
  std::vector<T> B(N, 2.0);
  std::vector<T> C(N);
  auto f = [](const double& a, const double& b){return a+b;};
  sycl::sycl_execution_policy<class transform1> ep;
  transform(ep, A.begin(), A.end(), B.begin(), C.begin(), f);
  for (auto& val : C) std::cout << val << "\n";
  return 0;
}
```

SYCL for MC Transport

```
1
2  auto transportFunctor = [=] DEVICE (Particle& p) {
3    auto& xs = *xsPtr;
4    while (p.alive()) {
5      auto dist = getDistanceAndMoveParticle(p, xs, eigenvalue);
6      eigenvalueTallyPtr->tallyTrack(dist, p, xs);
7      if (p.dead()) {
8        eigenvalueTallyPtr->tallyLeakage(p);
9      } else {
10       processCollision(p, xs, *fissionBankPtr, eigenvalue);
11     }
12   }
13 };
14
15 for_each(sourceBank->begin(), sourceBank->end(),
       transportFunctor, execPolicy);
```

Figure: Serial/OpenMP/CUDA transport kernel

SYCL for MC Transport

```
1  namespace sycl = cl::sycl
2  using ParticleBuffer = sycl::buffer<Particle, 1>;
3  using Range = sycl::range<1>;
4  using XSBuffer = sycl::buffer<MGXS, 1>;
5  using EigenvalueTallyBuffer = sycl::buffer<EigenvalueTally_t, 1>;
6  sycl::queue q(sycl::gpu_selector{});
7  { // Scoping brackets
8  ParticleBuffer sourceBankBuffer(sourceBank->data(), Range(3*nParticles));
9  ParticleBuffer fissionBankBuffer(fissionBank->data(), Range(3*nParticles));
10 XSBuffer xsBuffer(xsPtr, Range(1));
11 EigenvalueTallyBuffer eigenvalueTallyBuffer(eigenvalueTallyPtr, Range(1));
12 q.submit([&](sycl::handler &cgh){
13   auto sourceBankAcc = sourceBankBuffer.template get_access<sycl::access::mode::read_write>(cgh);
14   auto fissionBankAcc = fissionBankBuffer.template get_access<sycl::access::mode::read_write>(cgh);
15   auto xsAcc = xsBuffer.template get_access<sycl::access::mode::read_write>(cgh);
16   auto eigenvalueTallyAcc = eigenvalueTallyBuffer.template get_access<sycl::access::mode::read_write>(
       cgh);
17   auto transportFunctor = [=] (sycl::item<1> history){
18     auto historyID = history.get_linear_id();
19     Particle& p = sourceBankAcc[historyID];
20     auto& xs = xsAcc[0];
21     auto& eigenvalueTally = eigenvalueTallyAcc[0];
22     while (p.alive()) {
23       auto dist = getDistanceAndMoveParticle(p, xs, eigenvalue);
24       eigenvalueTally.tallyTrack(dist, p, xs);
25       if (p.dead()) {
26         eigenvalueTally.tallyLeakage(p);
27       } else {
28         processCollision(p, xs, fissionBankAcc, eigenvalue);
29       }
30     }
31   };
32   cgh.parallel_for<class TransportFunctor>(Range(sourceBank->size()), transportFunctor);
33 });
34 } // scoping brackets synchronize buffer data
```


Figure: SYCL transport kernel

SYCL for MC Transport

- How to access data (XS, geometry) within SYCL kernel?
  - Listing shown above assumes data within the XS object (total, fission, capture, etc) are containers with sizes known at compile time, i.e. they don't use an allocator. If they did use an allocator then every piece of data would need its own accessor: totalXSAcc, fissionXSAcc, etc.
- How to efficiently create secondary particles / add neutrons to fission bank?

Compiling

Need to set your compiler to Compute++, similar to setting your compiler to NVCC

- ComputeCPP's Ubuntu 16.04 version works on Darwin
- ComputeCpp built on top of Clang 6.0

Compiling

Need to set your compiler to Compute++, similar to setting your compiler to NVCC

- ComputeCPP's Ubuntu 16.04 version works on Darwin
- ComputeCpp built on top of Clang 6.0

Need to add SYCL code to your CMAKE project via add_sycl_to_target(TARGET SOURCE)

- provided by ComputeCpp SDK

Compiling

Need to set your compiler to Compute++, similar to setting your compiler to NVCC

- ComputeCPP's Ubuntu 16.04 version works on Darwin
- ComputeCpp built on top of Clang 6.0

Need to add SYCL code to your CMAKE project via add_sycl_to_target(TARGET SOURCE)

- provided by ComputeCpp SDK

Compilation does not work out-of-the-box on Darwin:

- Need to modify SDK's FindComputeCpp.cmake
- Need to specify gcc-toolchain

Compiling

```
1  list(APPEND COMPUTECPP_DEVICE_COMPILER_FLAGS --gcc-toolchain=${
       COMPUTECPP_TOOLCHAIN_DIR})
2  ...
3  add_library(ComputeCpp::ComputeCpp UNKNOWN IMPORTED GLOBAL)
4  add_library(OpenCL::OpenCL UNKNOWN IMPORTED GLOBAL)
5  ...
6  add_library(ComputeCpp INTERFACE)
7  target_compile_options(ComputeCpp INTERFACE "${
       COMPUTECPP_DEVICE_COMPILER_FLAGS}")
8  target_link_libraries(ComputeCpp INTERFACE "${
       COMPUTECPP_DEVICE_COMPILER_FLAGS}")
9  ...
10 target_link_libraries(${SDK_ADD_SYCL_TARGET} PRIVATE ComputeCpp)
```

Figure: Changes made to FindComputeCpp.cmake module.

Compiling

```
1 list(APPEND CMAKE_MODULE_PATH ${CMAKE_CURRENT_SOURCE_DIR}/
      dependencies/ComputeCpp/cmake/Modules)
2 find_package(ComputeCpp REQUIRED)
3 ...
4 add_sycl_to_target(TARGET ${SOURCE_NAME}.test SOURCES ${
      SOURCE_NAME}.test.cpp)
5 ...
```

Figure: Changes made to FindComputeCpp.cmake module.

Conclusions / Personal Reflections

SYCL is a powerful tool for heterogeneous computing.... if you're adding and multiplying floats.

- Difficult to use
  - Accessors and buffers are cumbersome
  - Objects w/ heap data are near impossible to use
- Full of surprises
  - Buffers two ways, one syncs, one doesn't
  - Only supports standard layout types in kernels

Conclusions / Personal Reflections

SYCL is a powerful tool for heterogeneous computing.... if you're adding and multiplying floats.

- Difficult to use
  - Accessors and buffers are cumbersome
  - Objects w/ heap data are near impossible to use
- Full of surprises
  - Buffers two ways, one syncs, one doesn't
  - Only supports standard layout types in kernels

SYCL is great when

- code is completely functional (no side-effects)
  - everything can be expressed as a std::algorithm
- only heap data being used is passed as arguments to parallel algorithms

Questions?

SYCL for Monte Carlo Transport

Timothy P. Burke

XCP-3 Monte Carlo Codes, Methods, and Applications
Los Alamos National Laboratory