SYCL as an Asynchronous Dataflow

Ruyman Reyes

ruyman@codeplay.com

Codeplay Software Ltd.

DHPCC++ – 16th May, 2017

Main goal of this proposal:

Bring data-flow programming as a first-level citizen

Current OpenCL specification

OpenCL 2.2 is low-level language

- Kernel synchronization via events and queues
- No interaction with host scheduling or threads
- Does not directly map the current trends of C++
- Only some SVM levels support atomics and synchronization

Current OpenCL specification

OpenCL 2.2 is low-level language

- Kernel synchronization via events and queues
- No interaction with host scheduling or threads
- Does not directly map the current trends of C++
- Only some SVM levels support atomics and synchronization

OpenCL behaviour is well defined

- Memory model defines data available to kernel
- Different levels have different visibility
- Clear when data is on host or not

OpenCL is too low level, but well defined

Current SYCL specification

SYCL behaves like a DAG

- Higher abstraction than OpenCL
- Command group and accessors define dependencies
- Access mode defines dependencies

SYCL Dag is vaguely defined

- Only expected behaviour is described
- Not clear how synchronization across context is possible
- No direct control over the generated DAG
- Cannot integrate easily with other schedulers

SYCL is high-level, but behaviour not well defined!

Objective: Fully define SYCL as Data Flow

Rules for memory synchronization

- Define the concepts behind accessor:
  - **Requisite**
  - **Action**
- Elaborate definitions for command group dependency
  - Enable users to reason an order of execution

Extending interface

- Update the current interface definitions
- Support C++ futures
- Support for updates to/from buffers
- Calling host functions from the SYCL dag.

What is an accessor?

```
auto cg = [&](handler& h) {
  auto accA = buf.get_access<access::mode::read>(h);
  auto accB = buf.get_access<access::mode::write>(h);
  h.parallel_for<class myKernel>(myRange, [=](item it) {
    accA[it] = accB[it];
  });
};
someQueue.submit(cg);
```

Accessors define requirements

- accA: Requires being able to read data on a context
- accB: Requires being able to write data on a context

What is an accessor?

```
auto cg = [&](handler& h) {
  auto accA = buf.get_access<access::mode::read>(h);
  auto accB = buf.get_access<access::mode::write>(h);
  h.parallel_for<class myKernel>(myRange, [=](item it) {
    accA[it] = accB[it];
  });
};
someQueue.submit(cg);
```

Accessors define requirements

- accA: Requires being able to read data on a context
- accB: Requires being able to write data on a context

Satisfy a requirement implies an action

- accA: Copy data into the context
- accB: Data must be available for writing

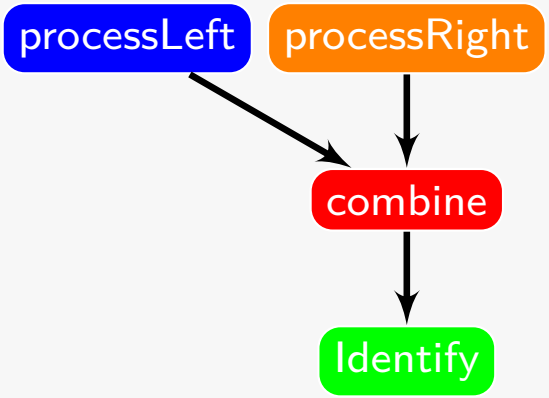
Actions are implementation-specific

```
buffer<int, 1> leftCameraInput{...};
buffer<int, 1> rightCameraInput{...};
buffer<int, 1> output{...};

queue q1(context1, vp1);
queue q2(context1, vp2);
queue q3(context2, gpu);

q1.submit(processLeft(lCam));
q2.submit(processRight(rCam));
q3.submit(combine(lCam, rCam, output));

{
  using r_mode = access::mode::write;
  using h_target = access::mode::host_buffer;
  auto hostC =
      output.get_access<r_mode, h_target>();
  identify(hostC);
}
```



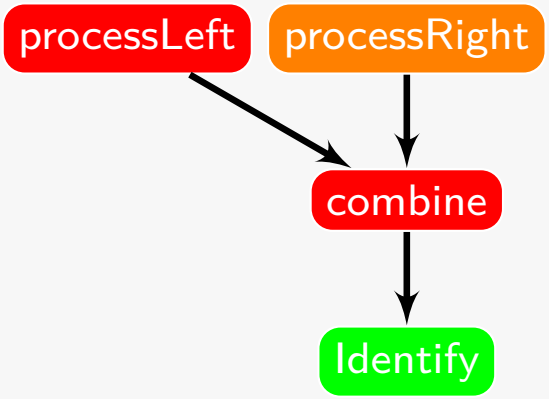
Actions are implementation-specific

```
buffer<int, 1> lCam{...};
buffer<int, 1> rCam{...};
buffer<int, 1> output{...};

queue q1(context1, vp1);
queue q2(context1, vp2);
queue q3(context2, gpu);

q1.submit(processLeft(lCam));
q1.submit(processRight(rCam));
q3.submit(combine(lCam, rCam, output));

{
  using r_mode = access::mode::write;
  using h_target = access::mode::host_buffer;
  auto hostC =
      output.get_access<r_mode, h_target>();
  identify(hostC);
}
```



Same requirements, different actions

Formalization of concepts

Requisite r_i

Must be fullfiled for one or more kernel-functions K_i to be executed on a particular device.

Actions R_i

An action a_i is a collection of implementation-defined operations that must be performed in order to satisfy a requisite.

Command Group CG

A CG named *foo* is expressed as: CG_{foo}. Contains a set of requisites (R) and a set of kernel functors K. Each $r_i \in R$ represents the requirements for the kernels in K.

Requierements affect all kernels in the CG

Formalization of concepts

Satisfaction of a requirement

- A requirement is satisfied when no actions are required.
- Evaluation of a requisite only observes (CG state not changed)

$$Eval(r_i) = \begin{cases} true & \text{if } n\ r_i \text{ is satisfied} \\ false & \text{if } n\ r_i \text{ is not satisfied} \end{cases}$$

CG_foo can only be executed iff $Eval(r_i) == true \forall r_i \in CG_foo$

Accessors as requirements

CG access to memory object

Accessors are expressed as $mode_{memory\ object}$, e.g: RW_{bufA} means Read Write access to buffer A.

Rules accessing the same memory object

- Multiple *CG* can request *RO* access simultaneously
- Only one *CG* can request *RW* access at certain time
- Multiple *CG* can request *DRW* or *DW* simultaneously
  $\rightarrow$ Only if accessing it whole
  $\rightarrow$ Partial discard access possible?

Clear definition of dependency rules across context

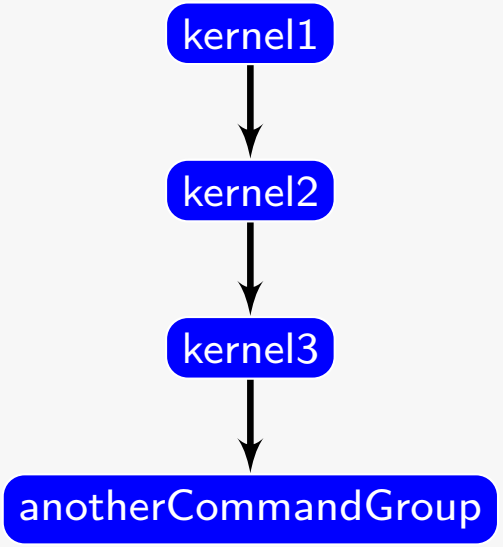
Interface Changes

Combining kernel API calls

```
q.submit([&](handler& h) {
  auto accA = bufA.get_access<
                    access::mode::read>(h);
  auto accB = bufB.get_access<
                    access::mode::read>(h);
  auto accC = bufC.get_access<
                    access::mode::read_write>(h);

  h.parallel_for(myRange1, kernel1(accA, accC));
  h.parallel_for(myRange2, kernel2(accB, accC));

  auto accD = bufD.get_access<access::mode::read
       >(h);
  h.parallel_for(myRange3, kernel3(accD, accC));
});
q.submit(anotherCommandGroup);
```



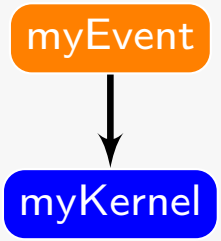
Kernels in the *CG* execute one after the other

Accessor resolution rules apply

Events as requisites

```
q.submit([&](handler& h) {
   h.wait_for(myEvent);
  auto accD = bufD.get_access<access::mode::
       read_write>(h);

  h.parallel_for(myRange1, myKernel(accD));
});
```



Command Group requires event *CL_FINISHED* to execute

Futures as requisites

```
q.submit([&](handler& h) {
  auto val = h.wait_for(std::move(myFuture));
  auto accD = bufD.get_access<access::mode::
       read_write>(h);

  h.parallel_for(myRange1, myKernel(accD, val));
});
```

- *CG* cannot start until future is retrieved
- Value retrieved from future can be used in kernel

Promise interface too?

Tasks executing on the host

```
qA.submit(cg1);
auto cgH = [=] (host_handler& h) {
  auto accA = bufA.get_access<access::mode::
       read>(h);
  auto accB = bufB.get_access<access::mode::
       read_write>(h);

  h.single_task([=]() {
    accB[0] = accA[0] * std::rand();
  }
};
qA.submit(cgH);
qA.submit(cg2);
```

Update host or device

```
auto cgU = [=] (handler& h) {
  auto accA = bufA.get_access<access::
       mode::read>(h);
  h.parallel_for<class kernel>(range,
       SomeKernel(accA));
  h.update_from_device(hostPtr, accA);
};
qA.submit(cgU);
```

```
auto cgH = [=] (handler& h) {
  auto accA = bufA.get_access<access::
       mode::read_write>(h);
  h.update_to_device(accA, hostPtr);
  h.parallel_for<class kernel>(range,
       SomeKernel(accA));
};
qA.submit(cgH);
```

To summarize

Extensions proposals

- Well explained behaviour for *CG* interaction
- Extensions to add new scheduling features
- Enables interaction with existing schedulers (e.g, TF)

Current status

- Some features available via codeplay handler
- Update to/from required for TensorFlow
- Multiple kernels per command group implementable (but not tested)

Do we want this features on 2.2?

Do we want/need to backport some features?

We're Hiring!
codeplay.com/careers
/

codeplay
THE HETEROGENEOUS SYSTEMS EXPERTS


@codeplaysoft


info@codeplay.com

codeplay.com