DATA PARALLEL C++ TAB MEETING : ATOMICS

oneAPI Technical Advisory Board - July 1, 2020

Agenda

1. Motivation
2. Proposed Features
3. Questions & Next Steps

Motivation: Code Before Alignment with C++

```
template <typename T>
void foo(global_ptr<T> a, local_ptr<T> b) {
  atomic(a).fetch_add(1); // Defaults to memory_order_relaxed
  atomic(b).fetch_add(1); // Defaults to memory_order_relaxed
}
```

Three important differences relative to standard C++:

- `std::atomic` cannot be constructed from a pointer
- `std::atomic` and `sycl::atomic` default to different memory orderings
- SYCL says nothing about how local memory is ordered with respect to global memory

Motivation: Common Patterns Are Broken

```
// write partial output to global reduction array
reduction[it.get_group_id()[0]] = LocalSum[0];

// signal completion of write to global array
int position = atomic(ncompleted).fetch_add(1);

// if work-group is last to arrive
if (position == it.get_group_range()[0] - 1) {
  // finish reduction
}
```

Code like this is not guaranteed to work according to SYCL 1.2.1

Completion of atomic operation does not imply visibility of preceding writes

Motivation: Code After Alignment with C++

```
namespace sycl {
namespace intel {
  template <typename T, memory_order DefaultOrder,
            memory_scope DefaultScope, access::address_space Space>
  struct atomic_ref;
}
}

template <typename T>
void foo(global_ptr<T>* a, local_ptr<T>* b) {
  atomic_ref<T, memory_order::seq_cst, memory_scope::device,
             access::address_space::global_space>(*a).fetch_add(1);
  atomic_ref<T, memory_order::seq_cst, memory_scope::device,
             access::address_space::local_space>(*b).fetch_add(1);
}
```

Three changes:

- `std::atomic_ref` and `sycl::intel::atomic_ref` can be constructed from reference
- `std::atomic_ref` and `sycl::intel::atomic_ref` support all standard memory orders
- a must always be updated before b ⇒ a single *happens-before* for all address spaces

`sycl::atomic => intel::atomic_ref`

Deprecate `cl::sycl::atomic` and replace it with `intel::atomic_ref`.

`intel::atomic_ref` is mostly aligned with C++20 `std::atomic_ref`:

- Stores a reference instead of owning data
- Can be constructed from a reference in the local or global address space
- Default argument for `memory_order` determined by templates
- Convenience operators like += and ++
- Support for arithmetic operators on float/double
- No `wait()`, `notify_one()` or `notify_all()` – synchronization is error-prone today

Memory Orderings and Scopes

We want to support:

- All standard C++17 memory orderings (without consume)
- Devices that cannot implement sequential consistency (e.g. OpenCL 1.2 devices)
- Contexts that cannot implement inter-device sequential consistency
- Optimizations based on intended visibility of memory updates

Queries return lists of supported values:

- OpenCL 1.2 device:
  ```
  orders = { relaxed }
  scopes = { work_item, sub_group, work_group }
  ```
- OpenCL 2.0 device:
  ```
  orders = { relaxed, acquire, release, acq_rel, seq_cst }
  scopes = { work_item, sub_group, work_group, device, system }
  ```

A Single *Happens-Before* Relation

OpenCL 2.0 splits the happens-before relation for local and global memory

More recent work suggests this split is unnecessary:

- Gaster *et. al* suggest treating local memory as memory supporting limited scope [1]
- CUDA does not appear to split the happens-before relation [2]

Writing a formal memory model extending C++ with scopes is a WIP

[1] https://uwe-repository.worktribe.com/preview/836458/opencl_memory_model.pdf
[2] https://github.com/NVlabs/ptxmemorymodel/blob/master/PTXMemoryModelASPLOS2019.pdf

Changes to Fences and Barriers

Fences:

- Deprecate `nd_item::fence(address_space = global_and_local)`
- Introduce `intel::atomic_fence(memory_order, memory_scope)`

Barriers:

- Need to be described in terms of how they interact with the memory model
- e.g. A barrier should execute a release fence, synchronize, then an acquire fence

Changes to Memory Consistency Model

Based upon C++ memory model; default behavior of SYCL is very close* to C++.

- All SYCL compilers required to support C++17 compilation (and C++17 memory model)

Different consistency guarantees at different scopes:

- Work-item code follows *sequenced-before* rules
- Work-groups can always enforce consistency via barriers, conditionally via atomics
- Work-items in different work-groups can conditionally enforce consistency via atomics

Memory consistency guarantees are independent of forward progress guarantees. Implementing work-group synchronization is still not portable.

* There is still a difference due to the visibility of private and local memory

Questions

- Should we support `std::atomic_ref` in device code?
  - If yes, should it forward to `intel::atomic_ref<seq_cst, system>`?

- Do we need a `std::atomic`-like interface as well as `atomic_ref`?
  - If yes, what are the expectations regarding definitions and capture?
  - e.g. Passing a `std::atomic<T>*` to a kernel?
  - e.g. Defining `std::atomic<T>` objects in global or local memory?

Summary & Next Steps

New atomic features:

- Adopt familiar C++ syntax and behavior
- Expose new functionality to more capable devices

Next steps:

- Shorthand `atomic_accessor` to replace `access::atomic`; need order and scope
- Define formal memory model extending C++ with scopes

intel®

experience
what's inside™

NOTICES & DISCLAIMERS