oneAPI Technical Advisory Board Meeting: sycl::accessor simplification

22 July 2020

Ilya Burylov

SYCL 1.2.1: Hello World

accessor An accessor is a class which allows a SYCL kernel function to access data managed by a buffer or image class. Accessors are used to express the dependencies among the different command groups.

```
// Create a queue to enqueue work to
queue myQueue;

// Wrap our data variable in a buffer
buffer<int, 1> resultBuf { data, range<1> { 1024 } };

// Create a command_group to issue commands to the queue
myQueue.submit([&](handler& cgh) {
  // request access to the buffer
  auto writeResult = resultBuf.get_access<access::mode::discard_write>(cgh);

  // Enqueue a parallel_for task
  cgh.parallel_for<class simple_test>(range<1> { 1024 }, [=](id<1> idx) {
    writeResult[idx] = idx[0];
  });
});
```

Buffer handles storage and data ownership

Accessor handles access to the data

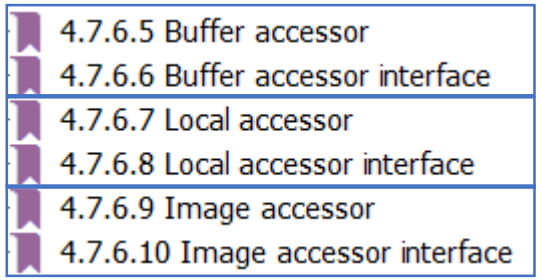
access

The problems – Verbosity and Consistency

1. **sycl::accessor** has 5 template arguments and only 2 of them have default values
   - One is forced to type at least 3 template arguments

```
// We define the subset of the accessor we require for the kernel
accessor<int, 1, access::mode::read_write, access::target::global_buffer>
    ptr(syclBuffer, cgh, singleRange, offset);
```

   - **sycl::buffer/sycl::image** have **get_access()** member functions with 1-2 template arguments, but they do not cover all accessor use cases (placeholder, local, dimension=0, etc.)



2. **sycl::accessor** behavior and API depends on the template arguments
   - There are 3 different sections of the standard, which describe **sycl::accessor** type, depending on **access::target** template argument
   - A set of available constructors depends on 3 out of 5 template arguments
   - Whether accessor constructor is a **blocking** operation or **non-blocking**, depends on **access::target** template argument

```
/* Available only when: ((isPlaceholder == access::placeholder::false_t &&
accessTarget == access::target::host_buffer) || (isPlaceholder ==
access::placeholder::true_t && (accessTarget == access::target::global_buffer
|| accessTarget == access::target::constant_buffer))) && dimensions > 0 */
template <typename AllocatorT>
accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef);
```

The challenges - Extensibility

New use cases appear

1. **Host tasks (SYCL 2020 Provisional)**
   - One may now build a DAG for running tasks on the Host, thus host accessors shall support non-blocking construction as well.
   - Existing access targets are insufficient to decide, whether accessor constructor should be a **blocking** or **non-blocking** operation.

2. **Atomicity (Intel extension)**
   - No reasonable default values for **memory_order** and **memory_scope** for all possible platforms
     - There is a need to set up defaults on the level of accessor:
       - 2 more template arguments
       - only if **access::mode** == **atomic**
   - Note: **access::mode::atomic** is deprecated in SYCL 2020 provisional

SYCL 2020 Provisional

- **sycl::host_accessor** is defined as a distinct type for access memory on host

- **sycl::accessor** and **sycl::host_accessor** now accept tag arguments in constructor.
  2 helper tag types are introduced with a set of predefined inline variables:

  - inline constexpr mode_tag_t<…> **read_only**{};
  - inline constexpr mode_tag_t<…> **read_write**{};
  - inline constexpr mode_tag_t<…> **write_only**{};
  - inline constexpr mode_target_tag_t<…> **read_constant**{};

- A set of deduction rules are requested on implementation level
  to deduce all accessor template arguments with optional use of tag types

- All accessor constructors accept **property_list**:

  - **discard_write** and **discard_read_write** modes were deprecated
  - replaced with **noinit**{} runtime property

New API comparison – device accessor

	SYCL 1.2.1	SYCL 2020 Provisional

via constructor	`sycl::accessor<int,1, sycl::access::mode::write > A1(buf,cgh);`
	`sycl::accessor<int,1, sycl::access::mode::read_write > A2(buf,cgh);`
	`sycl::accessor<int,1,`
	`sycl::access::mode::discard_read_write> A3(buf,cgh);`
	`sycl::accessor<int,1,`
	`sycl::access::mode::write,`
	`sycl::access::target::global_buffer,`
	`sycl::access::placeholder::true_t > A4(buf);`
	`sycl::accessor<int,1,`
	`sycl::access::mode::read,`
	`sycl::access::target::constant_buffer > A5(buf,cgh);`
		`sycl::accessor A1(buf, cgh, sycl::write_only);`
		`sycl::accessor A2(buf, cgh);`
		`sycl::accessor A3(buf, cgh, sycl::noinit);`
		`sycl::accessor A4(buf, sycl::write_only);`
		`sycl::accessor A5(buf, cgh, sycl::read_constant);`

via buffer	`auto A1 = buf.get_access<sycl::access::mode::write >(cgh);`
	`auto A2 = buf.get_access<sycl::access::mode::read_write >(cgh);`
	`auto A3 = buf.get_access<sycl::access::mode::discard_read_write>(cgh);`
	`//auto A4 = not supported`
	`auto A5 = buf.get_access<sycl::access::mode::read,`
	`sycl::access::target::constant_buffer> (cgh);`
		`auto A1 = buf.get_access(cgh, sycl::write_only);`
		`auto A2 = buf.get_access(cgh);`
		`auto A3 = buf.get_access(cgh, sycl::noinit);`
		`auto A4 = buf.get_access(sycl::write_only);`
		`auto A5 = buf.get_access(cgh, sycl::read_constant);`

New API comparison – host accessor

	SYCL 1.2.1	SYCL 2020 Provisional

via constructor	`sycl::accessor<int,1,`
	`  sycl::access::mode::write,`
	`  sycl::access::target::host_buffer> A1(buf);`
		`sycl::host_accessor A1(buf, sycl::write_only);`

	`sycl::accessor<int,1,`
	`  sycl::access::mode::read_write,`
	`  sycl::access::target::host_buffer> A2(buf);`
		`sycl::host_accessor A2(buf);`

	`//sycl::accessor<not supported> A3(buf);`	`sycl::host_accessor A3(buf, cgh, sycl::write_only);`
	`//sycl::accessor<not supported> A4(buf);`	`sycl::host_accessor A4(buf, cgh);`
via buffer	`auto A1 = buf.get_access<sycl::access::mode::write     >();`	`auto A1 = buf.get_host_access(sycl::write_only);`
	`auto A2 = buf.get_access<sycl::access::mode::read_write>();`	`auto A2 = buf.get_host_access();`
	`//auto A3 = not supported`	`auto A3 = buf.get_host_access(cgh, sycl::write_only);`
	`//auto A4 = not supported`	`auto A4 = buf.get_host_access(cgh);`

Hesitation

A ***placeholder*** accessor is not bound to a command group at construction time.

It is expected to be bound later.

Within SYCL 2020 provisional logic:

- accessor (buffer) -> **non-blocking**, **placeholder**
- accessor (buffer, handler) -> non-blocking, not-placeholder
- host_accessor (buffer) -> **blocking**, **not-placeholder**
- host_accessor (buffer, handler) -> non-blocking, not-placeholder

Absence of the handler is interpreted differently by deduction guides.

This difference is nonintuitive.

Placeholder host accessors are not supported.

Possible solution: split host_accessor in two types:

- host_accessor (buffer) -> **always blocking**
- host_task_accessor (buffer, handler) -> **always non-blocking**

Is such granularity good enough?

Further work

Create a dedicated atomic accessor (extension):

```
template <typename DataT, int Dimensions,
    memory_order DefaultOrder, memory_scope DefaultScope,
    access::target AccessTarget = access::target::global_buffer,
    access::placeholder IsPlaceholder = access::placeholder::false_t>
class atomic_accessor;
```

Create more dedicated types/aliases (the list is not finalized):

- local_accessor (alias)
- local_atomic_accessor (alias)
- host_task_accessor
- image_accessor
- host_image_accessor

Is such granularity good enough?

Resolve zero-dimensional use case:

- sycl::buffer<int, **1**> buf(range<1>(1));
- sycl::accessor<int, **0**, sycl::access::mode::write> acc(buf);
  - Zero-dimensional accessor are not deducible

Is it an important use case?

Other changes in accessor

- Removed size_t overload for operator[]
  - operator[](id<Dimensions> Index)
  - ~~operator[](size_t Index)~~
    - This allowed passing item<Dimensions> directly to operator[] (when Dimensions == 1)
    - This disallowed passing Types, with user defined conversion to size_t

Is a disallowed use case important one?

- Accessor allows const qualified type as a dataT
  - This is equivalent to **access::mode::read**

```
template <typename dataT,
          int dimensions = 1,
          access_mode accessmode =
            (std::is_const_v<dataT> ? access_mode::read
                                    : access_mode::read_write),
          target accessTarget = target::global_buffer,
          access::placeholder isPlaceholder = access::placeholder::false_t // Deprecated
>
```

- More defaults for template arguments
- isPlaceholder template argument is deprecated – replaced with runtime value evaluated from constructor arguments
- Accessors now meet the C++ requirements of ContiguousContainer and ReversibleContainer
- Read only accessor return const reference from operator[]
  - const DataT& operator[](id<Dimensions> Index)
  - ~~DataT operator[](id<Dimensions> Index)~~

Notices and Disclaimers

This specification is a continuation of Intel's decades-long history of working with standards groups and industry/academia initiatives such as The Khronos Group*, to create and define specifications in an open and fair process to achieve interoperability and interchangeability. oneAPI is intended to be an open specification and we encourage you to help us make it better. Your feedback is optional, but to enable Intel to incorporate any feedback you may provide to this specification, and to further upstream your feedback to other standards bodies, including The Khronos Group SYCL* specification, please submit your feedback under the terms and conditions below. Any contribution of your feedback to the oneAPI Specification does not prohibit you from also contributing your feedback directly to The Khronos Group or other standard bodies under their respective submission policies.

By opening an issue, providing feedback, or otherwise contributing to the specification, *you agree that Intel will be free to use, disclose, reproduce, modify, license, or otherwise distribute your feedback in its sole discretion without any obligations or restrictions of any kind, including without limitation, intellectual property rights or licensing obligations.* For complete contribution policies and guidelines, see Contribution Guidelines on www.spec.oneapi.com.