SYCL & DPC++: Improvements to the SYCL Programming Model

Nevin ":-)" Liber
nliber@anl.gov

U.S. DEPARTMENT OF ENERGY

Argonne NATIONAL LABORATORY

Nevin @ ANL

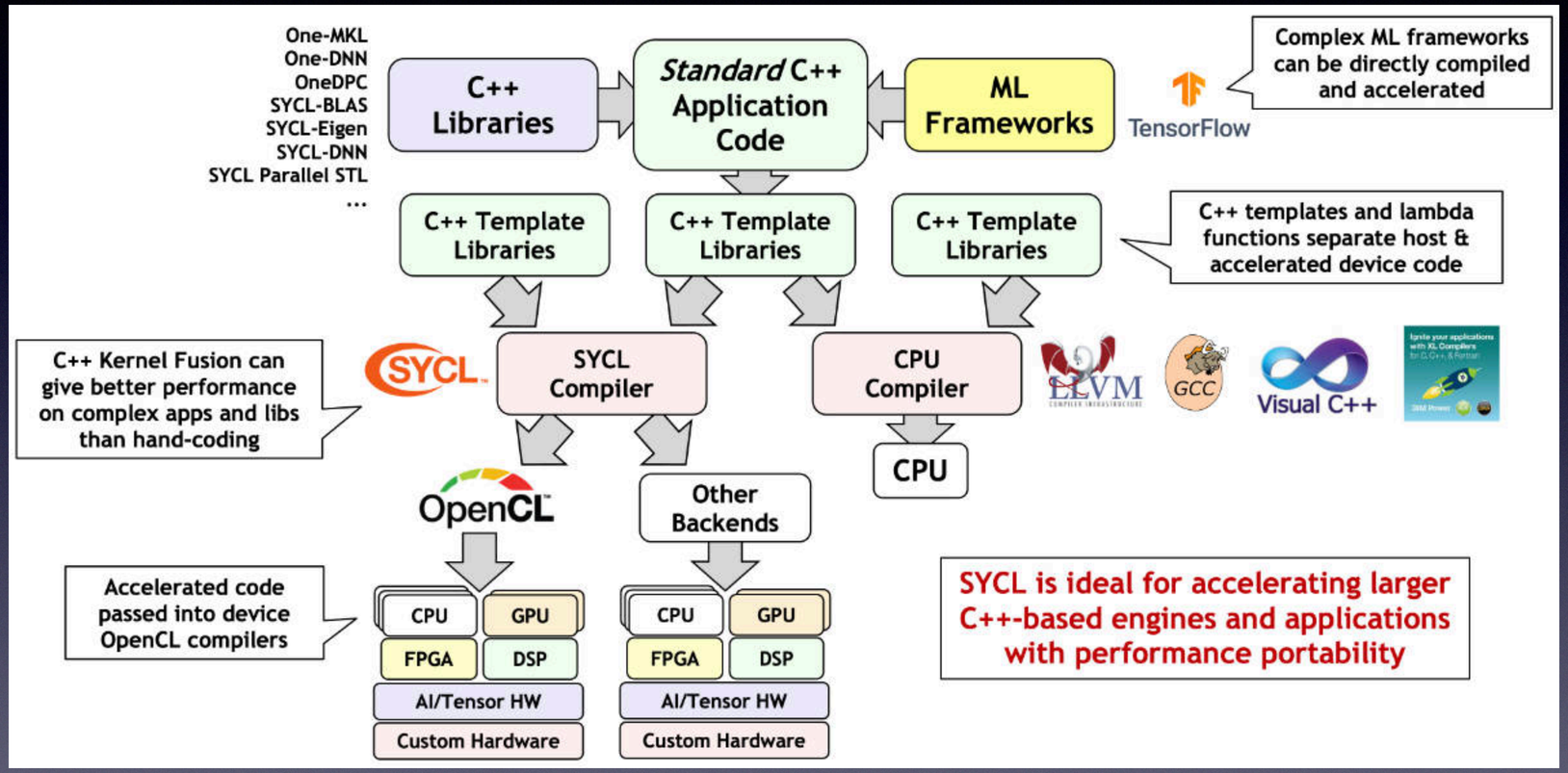
- SYCL / DPC++ backend for Kokkos
  - Initially targeting Aurora
- SYCL Committee Representative
- Vice Chair, Library Evolution Working Group Incubator (WG21 / C++ Committee)

Assumptions

- Some familiarity with SYCL 1.2.1
- Some familiarity with C++17

What is SYCL?

- Open standard
  - Khronos Group
- Parallel computing
- Heterogenous computing
  - Multiple devices involving CPUs, GPUs, DSPs, FPGAs, etc.


One-MKL
One-DNN
OneDPC
SYCL-BLAS
SYCL-Eigen
SYCL-DNN
SYCL Parallel STL
...
C++ Libraries
Standard C++ Application Code
ML Frameworks
TensorFlow
Complex ML frameworks can be directly compiled and accelerated
C++ Template Libraries
C++ Template Libraries
C++ Template Libraries
C++ templates and lambda functions separate host & accelerated device code
C++ Kernel Fusion can give better performance on complex apps and libs than hand-coding
SYCL
SYCL Compiler
CPU Compiler
LLVM
GCC
Visual C++
CPU
OpenCL
Other Backends
Accelerated code passed into device OpenCL compilers
CPU
GPU
FPGA
DSP
AI/Tensor HW
Custom Hardware
CPU
GPU
FPGA
DSP
AI/Tensor HW
Custom Hardware
SYCL is ideal for accelerating larger C++-based engines and applications with performance portability

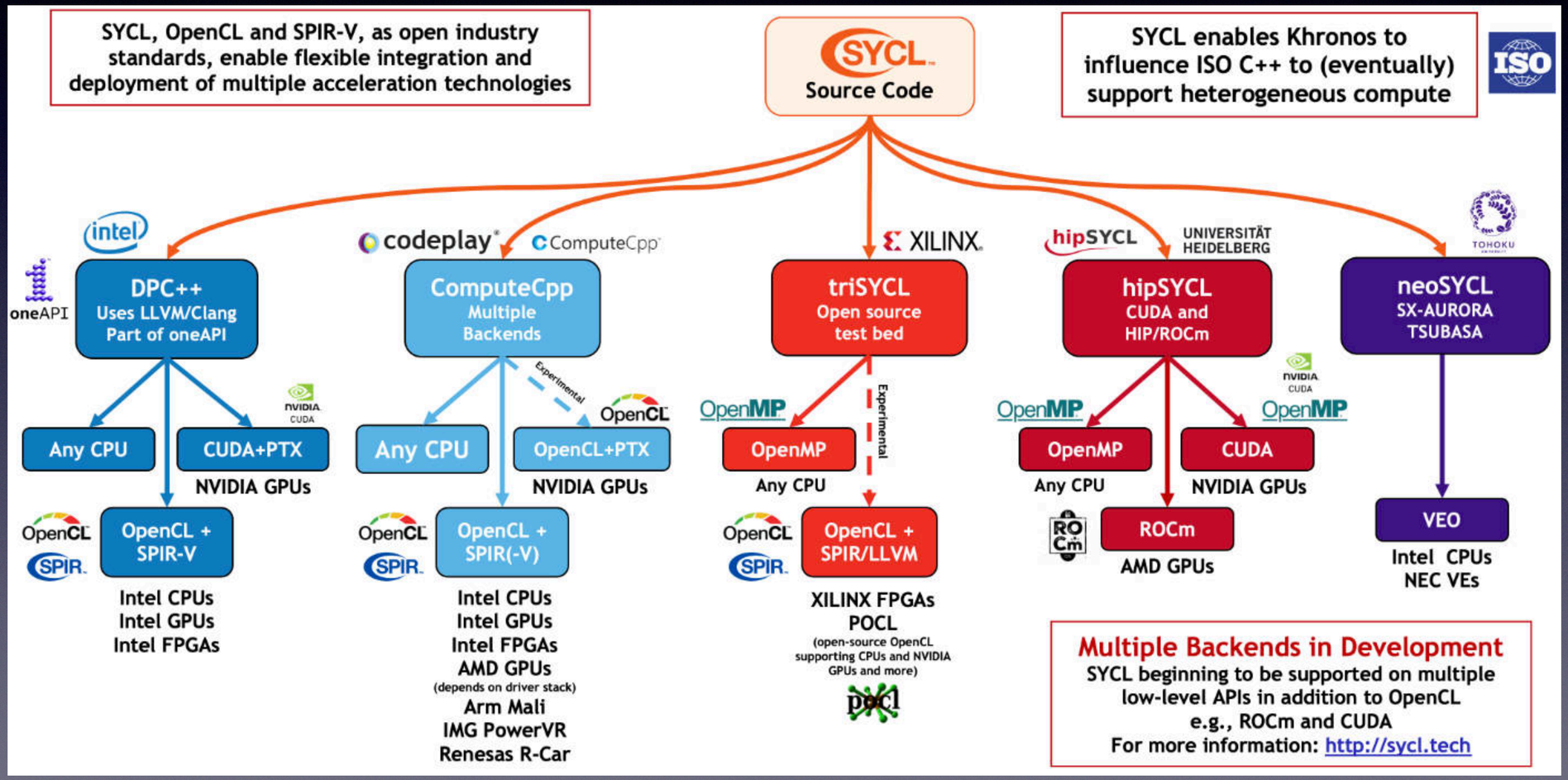
Why not just C++?

- C++ supports parallel computing
  - Execution policies, algorithms, atomics, etc.
- C++ has no notion of heterogeneous computing (multiple devices)
  - Heck, no notion of multiple processes on a single device
  - It may take decades to add these notions

What is the relation between SYCL & C++?

- SYCL is where we develop parallel and heterogenous abstractions
  - Long term goal: Standard C++ adopt many of these abstractions
- SYCL adopts Standard C++ features
  - Easier to develop code in SYCL
  - Including those added to C++ for other domains
    - E.g., Class Template Argument Deduction (deduction guides)

What hardware does SYCL support?



What is the relation between DPC++ & SYCL?

- DPC++ is the Intel implementation of SYCL with extensions
  - Based on Clang (Open Source)
  - Compiler for Aurora
  - A place for Intel to explore and prototype new features for SYCL
    - DPC++ -> SYCL -> C++

SYCL 2020

- Released February 9th, 2021
- C++17 baseline
- Moving away from being OpenCL™ - centric
  - #include <sycl/sycl.hpp>
  - **namespace** sycl instead of **namespace** cl::sycl
- Over forty new features…

Features

- Unified Shared Memory (USM)
- Parallel Reductions
- Atomics
- Device Copyable
- Unnamed Lambdas
- Class Template Argument Deduction (CTAD) / Deduction Guides

Unified Shared Memory (USM)

Unified Shared Memory (USM)

- Pointer based model
- Unified virtual address space
- An allocated pointer has the same value (object representation) on the host and on the device
  - Although there may be access restrictions when dereferencing

Unified Shared Memory (USM)

Allocation Type	Description	Accessible on host?	Accessible on device?	Located on
device	Allocations in device memory	X	√	device
host	Allocations in host memory	√	√	host
shared	Allocations shared between host and device	√	√	can migrate between host and device

Device Allocations

- Memory attached to device
- Not accessible on the host
  - If host needs access, must be explicitly copied via special memcpy calls

Host Allocations

- Resides on host
- Implicitly accessible on host and device
  - Device access to data over bus (e.g., PCI-E)
    - Slower than device allocations
- Rarely accessed data
- Large data sets

Shared Allocations

- Implicitly accessible on host and device
  - Data *can* migrate to where it is used on-demand
    - Could be implemented as device allocation
    - `prefetch`
      - Start the migration early
    - `mem_advise`

Context

- Allocations belong to a `context`
  - Device or set of devices
  - We can get the `context` from a `queue`
  - May not be useable across contexts
- Device allocations belong to a device

Data Movement (recap)

- Explicit
  - Special `memcpy` calls (`handler`, `queue`)
  - Fine-grained control
- Implicit
  - Host - data sent over bus (e.g., PCI-E)
  - Shared - migrated

Allocation Styles

- C
  - `malloc`, `aligned_alloc`, `malloc_host`, etc.
  - Specify size of the allocation
- C++
  - `template<typename T> T* malloc_host(…)`, etc.
  - Stateful C++17 allocators

USM vs. Buffers / Accessors

- USM Pointers
  - Very close to regular C++ programming
- Accessors
  - Implicitly builds data dependency DAG between kernels

Parallel Reductions

```
buffer<int> valuesBuf{1024};
{
    // Initialize buffer on the host with 0, 1, 2, 3, ..., 1023
    host_accessor a{valuesBuf};
    std::iota(a.begin(), a.end(), 0);
}

// Buffers with just 1 element to get the reduction results
int sumResult = 0;
buffer<int> sumBuf{&sumResult, 1};
int maxResult = 0;
buffer<int> maxBuf{&maxResult, 1};

myQueue.submit([&](handler& cgh) {
    // Input values to reductions are standard accessors
    auto inputValues = valuesBuf.get_access<access_mode::read>(cgh);

    // Create temporary objects describing variables with reduction semantics
    auto sumReduction = reduction(sumBuf, cgh, plus<>());
    auto maxReduction = reduction(maxBuf, cgh, maximum<>());

    // parallel_for performs two reduction operations
    // For each reduction variable, the implementation:
    // - Creates a corresponding reducer
    // - Passes a reference to the reducer to the lambda as a parameter
    cgh.parallel_for(range<1>{1024}, sumReduction, maxReduction,
                     [=](id<1> idx, auto& sum, auto& max) {
                         // plus<>() corresponds to += operator, so sum can be
                         // updated via += or combine()
                         sum += inputValues[idx];

                         // maximum<>() has no shorthand operator, so max can
                         // only be updated via combine()
                         max.combine(inputValues[idx]);
                     });
});

// sumBuf and maxBuf contain the reduction results once the kernel completes
assert(maxBuf.get_host_access()[0] == 1023 &&
       sumBuf.get_host_access()[0] == 523776);
```

- reduction is a function that returns an *unspecified* reduction object
  - Reduction variable (`sumBuf`, `maxBuf`)
  - Reduction operator / function object (`plus<>()`, `maximum<>()`)
  - Optional identity
  - This is why we use `auto`

```
buffer<int> valuesBuf{1024};
{
    // Initialize buffer on the host with 0, 1, 2, 3, ..., 1023
    host_accessor a{valuesBuf};
    std::iota(a.begin(), a.end(), 0);
}

// Buffers with just 1 element to get the reduction results
int sumResult = 0;
buffer<int> sumBuf{&sumResult, 1};
int maxResult = 0;
buffer<int> maxBuf{&maxResult, 1};

myQueue.submit([&](handler& cgh) {
    // Input values to reductions are standard accessors
    auto inputValues = valuesBuf.get_access<access_mode::read>(cgh);

    // Create temporary objects describing variables with reduction semantics
    auto sumReduction = reduction(sumBuf, cgh, plus<>());
    auto maxReduction = reduction(maxBuf, cgh, maximum<>());

    // parallel_for performs two reduction operations
    // For each reduction variable, the implementation:
    // - Creates a corresponding reducer
    // - Passes a reference to the reducer to the lambda as a parameter
    cgh.parallel_for(range<1>{1024}, sumReduction, maxReduction,
                     [=](id<1> idx, auto& sum, auto& max) {
                         // plus<>() corresponds to += operator, so sum can be
                         // updated via += or combine()
                         sum += inputValues[idx];

                         // maximum<>() has no shorthand operator, so max can
                         // only be updated via combine()
                         max.combine(inputValues[idx]);
                     });
});

// sumBuf and maxBuf contain the reduction results once the kernel completes
assert(maxBuf.get_host_access()[0] == 1023 &&
       sumBuf.get_host_access()[0] == 523776);
```

- When passed to `parallel_for`
  - Constructs an appropriate *unspecified* reducer object (`sum`, `max`)
  - Why we use `auto`
    - Generic / polymorphic lambda
    - templated `operator()`
      - Even though we only need one (monomorphic)
      - Don’t want to specify the type

```
buffer<int> valuesBuf{1024};
{
    // Initialize buffer on the host with 0, 1, 2, 3, ..., 1023
    host_accessor a{valuesBuf};
    std::iota(a.begin(), a.end(), 0);
}

// Buffers with just 1 element to get the reduction results
int sumResult = 0;
buffer<int> sumBuf{&sumResult, 1};
int maxResult = 0;
buffer<int> maxBuf{&maxResult, 1};

myQueue.submit([&](handler& cgh) {
    // Input values to reductions are standard accessors
    auto inputValues = valuesBuf.get_access<access_mode::read>(cgh);

    // Create temporary objects describing variables with reduction semantics
    auto sumReduction = reduction(sumBuf, cgh, plus<>());
    auto maxReduction = reduction(maxBuf, cgh, maximum<>());

    // parallel_for performs two reduction operations
    // For each reduction variable, the implementation:
    // - Creates a corresponding reducer
    // - Passes a reference to the reducer to the lambda as a parameter
    cgh.parallel_for(range<1>{1024}, sumReduction, maxReduction,
                     [=](id<1> idx, auto& sum, auto& max) {
                         // plus<>() corresponds to += operator, so sum can be
                         // updated via += or combine()
                         sum += inputValues[idx];

                         // maximum<>() has no shorthand operator, so max can
                         // only be updated via combine()
                         max.combine(inputValues[idx]);
                     });
});

// sumBuf and maxBuf contain the reduction results once the kernel completes
assert(maxBuf.get_host_access()[0] == 1023 &&
       sumBuf.get_host_access()[0] == 523776);
```

Reducer

```
// Exposition only
template <typename T, typename BinaryOperation, int Dimensions, /* unspecified */>
class reducer {

  reducer(const reducer&) = delete;
  reducer& operator(const reducer&) = delete;

  /* Only available if Dimensions == 0 */
  void combine(const T& partial);

  /* Only available if Dimensions > 0 */
  __unspecified__ &operator[](size_t index) const;

  /* Only available if identity value is known */
  T identity() const;

  };
```

Reducer

- *Unspecified* type
- Intermediate values cannot be inspected
- Can only be updated via `combine`
- `operator[]` returns a *different unspecified* reducer object with dimensionality of one less than `Dimensions`
- For known binary operations, synonyms for `combine()` are provided

```
// Exposition only
template <typename T, typename BinaryOperation, int Dimensions, /* unspecified */>
class reducer {

  reducer(const reducer&) = delete;
  reducer& operator(const reducer&) = delete;

  /* Only available if Dimensions == 0 */
  void combine(const T& partial);

  /* Only available if Dimensions > 0 */
  __unspecified__ &operator[](size_t index) const;

  /* Only available if identity value is known */
  T identity() const;

};
```

- Safe to call **`combine`** concurrently
- Reduction variable has the result when the kernel finishes executing
- The combination order of multiple reducers is unspecified

```
buffer<int> valuesBuf{1024};
{
    // Initialize buffer on the host with 0, 1, 2, 3, ..., 1023
    host_accessor a{valuesBuf};
    std::iota(a.begin(), a.end(), 0);
}

// Buffers with just 1 element to get the reduction results
int sumResult = 0;
buffer<int> sumBuf{&sumResult, 1};
int maxResult = 0;
buffer<int> maxBuf{&maxResult, 1};

myQueue.submit([&](handler& cgh) {
    // Input values to reductions are standard accessors
    auto inputValues = valuesBuf.get_access<access_mode::read>(cgh);

    // Create temporary objects describing variables with reduction semantics
    auto sumReduction = reduction(sumBuf, cgh, plus<>());
    auto maxReduction = reduction(maxBuf, cgh, maximum<>());

    // parallel_for performs two reduction operations
    // For each reduction variable, the implementation:
    // - Creates a corresponding reducer
    // - Passes a reference to the reducer to the lambda as a parameter
    cgh.parallel_for(range<1>{1024}, sumReduction, maxReduction,
                     [=](id<1> idx, auto& sum, auto& max) {
                         // plus<>() corresponds to += operator, so sum can be
                         // updated via += or combine()
                         sum += inputValues[idx];

                         // maximum<>() has no shorthand operator, so max can
                         // only be updated via combine()
                         max.combine(inputValues[idx]);
                     });
});

// sumBuf and maxBuf contain the reduction results once the kernel completes
assert(maxBuf.get_host_access()[0] == 1023 &&
       sumBuf.get_host_access()[0] == 523776);
```

- Because sumReduction uses `std::plus<>`, `operator+=` is defined to be a synonym for `combine`
  - If `T` is an integral type, also `operator++`
  - Similar synonyms for `multiplies<>`, `bit_and<>`, `bit_or<>`, `bit_xor<>`
    - *if there wasn't a bug in the SYCL 2020 standard…*

```
buffer<int> valuesBuf{1024};
{
    // Initialize buffer on the host with 0, 1, 2, 3, ..., 1023
    host_accessor a{valuesBuf};
    std::iota(a.begin(), a.end(), 0);
}

// Buffers with just 1 element to get the reduction results
int sumResult = 0;
buffer<int> sumBuf{&sumResult, 1};
int maxResult = 0;
buffer<int> maxBuf{&maxResult, 1};

myQueue.submit([&](handler& cgh) {
    // Input values to reductions are standard accessors
    auto inputValues = valuesBuf.get_access<access_mode::read>(cgh);

    // Create temporary objects describing variables with reduction semantics
    auto sumReduction = reduction(sumBuf, cgh, plus<>());
    auto maxReduction = reduction(maxBuf, cgh, maximum<>());

    // parallel_for performs two reduction operations
    // For each reduction variable, the implementation:
    // - Creates a corresponding reducer
    // - Passes a reference to the reducer to the lambda as a parameter
    cgh.parallel_for(range<1>{1024}, sumReduction, maxReduction,
                     [=](id<1> idx, auto& sum, auto& max) {
                         // plus<>() corresponds to += operator, so sum can be
                         // updated via += or combine()
                         sum += inputValues[idx];

                         // maximum<>() has no shorthand operator, so max can
                         // only be updated via combine()
                         max.combine(inputValues[idx]);
                     });
});

// sumBuf and maxBuf contain the reduction results once the kernel completes
assert(maxBuf.get_host_access()[0] == 1023 &&
       sumBuf.get_host_access()[0] == 523776);
```

Atomics

Atomic support in SYCL 1.2.1

- Modeled on C++11 atomic class
  - Only relaxed because of OpenCL
  - Owns the data
    - Over-constraint
      - Never non-atomic access

C++11

- No notion of non-owning interfaces
  - Raw pointers non-owning by convention
  - E.g., interfaces with strings

```
void A(std::string const&)  { /* ... */ } // owning; allocation?
void A(char const*, size_t) { /* ... */ } // non-owning; more efficient
```

C++17

- `string_view`
  - Non-owning reference to string data
    - Pointer and a size
  - Efficient
  - Great vocabulary type for interfaces
  - Separation of concerns - only concerned with operations

Non-owning Interfaces

- C++20
  - `atomic_ref`
  - `span`
    - Non-owning reference to contiguous data
  - Ranges
- C++23 (maybe)
  - `function_ref`
  - `mdspan`

atomic_ref

- Non-owning atomic access to data
- Modeled on C++20 `atomic_ref` class
  - SYCL only required to support `relaxed`
  - May support `acq_rel` and `seq_cst`

atomic_ref C++ vs. SYCL

```
namespace std { template<class T> struct atomic_ref {
    T load(memory_order = memory_order::seq_cst) const noexcept;
    //...
};}

namespace sycl { template <class T, memory_order DefaultOrder, memory_scope DefaultScope, access
::address_space Space = access::address_space::generic_space>
struct atomic_ref {
    T load(memory_order order = default_read_order, memory_scope scope = DefaultScope) const noexcept;
    //...
};}
```

atomic_ref C++ vs. SYCL

```
namespace std { template<class T> struct atomic_ref {
    T load(memory_order = memory_order::seq_cst) const noexcept;
    //...
};}

namespace sycl { template <class T, memory_order DefaultOrder, memory_scope DefaultScope, access
::address_space Space = access::address_space::generic_space>
struct atomic_ref {
    T load(memory_order order = default_read_order, memory_scope scope = DefaultScope) const noexcept;
    //...
};}
```

- C++
  - T is trivially copyable
- SYCL
  - `int`, `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`, `float`, `double`
  - No support for large types (e.g., `complex<double>`)
    - Might require globally accessible lock table

atomic_ref C++ vs. SYCL

```
namespace std { template<class T> struct atomic_ref {
    T load(memory_order = memory_order::seq_cst) const noexcept;
    //...
};}

namespace sycl { template <class T, memory_order DefaultOrder, memory_scope DefaultScope, access
::address_space Space = access::address_space::generic_space>
struct atomic_ref {
    T load(memory_order order = default_read_order, memory_scope scope = DefaultScope) const noexcept;
    //...
};}
```

- `memory_order`
  - More general and useful in heterogeneous computing than safe `seq_cst`
  - `default_read_order` or `default_write_order` may be different
    - `DefaultOrder==acq_rel` -> `default_read_order==acquire`

atomic_ref C++ vs. SYCL

```
namespace std { template<class T> struct atomic_ref {
    T load(memory_order = memory_order::seq_cst) const noexcept;
    //...
};}

namespace sycl { template <class T, memory_order DefaultOrder, memory_scope DefaultScope, access
::address_space Space = access::address_space::generic_space>
struct atomic_ref {
    T load(memory_order order = default_read_order, memory_scope scope = DefaultScope) const noexcept;
    //...
};}
```

- memory_scope
  - Memory ordering constraints
  - work_item, sub_group, work_group, device, system

atomic_ref C++ vs. SYCL

```
namespace std { template<class T> struct atomic_ref {
    T load(memory_order = memory_order::seq_cst) const noexcept;
    //...
};}

namespace sycl { template <class T, memory_order DefaultOrder, memory_scope DefaultScope, access
::address_space Space = access::address_space::generic_space>
struct atomic_ref {
    T load(memory_order order = default_read_order, memory_scope scope = DefaultScope) const noexcept;
    //...
};}
```

- address_space
  - generic_space, global_space, local_space
  - Override default of generic_space for performance tuning

atomic_ref C++ vs. SYCL

```
namespace std { template<class T> struct atomic_ref {
    T load(memory_order = memory_order::seq_cst) const noexcept;
    //...
};}

namespace sycl { template <class T, memory_order DefaultOrder, memory_scope DefaultScope, access
::address_space Space = access::address_space::generic_space>
struct atomic_ref {
    T load(memory_order order = default_read_order, memory_scope scope = DefaultScope) const noexcept;
    //...
};}
```

Device Copyable

Device Copyable

- How do we copy objects in C++?
  - Copy constructor / copy assignment operator
    - Running code
    - Code may access both source and destination

Device Copyable

- Can we do the same for inter-device copying?
  - Non-trivial copy constructor / copy assignment operator
    - Where would the code run?
    - May not be legal to access both source and destination
- About all we can do is copy the bytes (object representation) that make up the object

Device Copyable

- C++ trivially copyable types
  - Used as a proxy for types where we can copy the bytes

C++ Trivially Copyable

- All base classes and non-static members are trivially copyable
- Has at least one public non-deleted copy/move ctor/assign
- If it has a copy/move ctor/assign, it must be public and defaulted
- Has a public defaulted destructor

C++ Trivially Copyable

- Conflated into trivially copyable
  - Bitwise copyable
  - Layout
- Trivially copyable is too restrictive (not necessary)
  - Not sufficient either
    - Member functions can throw exceptions

C++ Trivially Copyable

- There are standard library types which are not necessarily trivially copyable for historical reasons
  - `pair`, `tuple` (even when the types it contains are trivially copyable)
  - And because layout is conflated, changing would be ABI break
- And some which are not yet guaranteed to be trivially copyable
  - `span`, `basic_string_view`
    - Although these might be by C++23 due to paper P2251

C++ Trivially Copyable

- If a lambda captures a non trivially copyable type
  - The lambda (which is just a struct) is not trivially copyable
  - The lambda cannot be implicitly copied to the kernel
- Lead to some interesting workarounds in Kokkos and RAJA
  - Manually copy the bytes to the device
    - Technically violates C++ object model (lifetime of objects)

Device Copyable

- Types where bitwise copy for inter-device copying has correct semantics
- Unspecified whether or not copy/move ctor/assign is called to do the inter-device copying
- Unspecified whether or not the destructor is called on the device
  - Since it must effectively have no effect on the device
- User specializable trait to indicate a type is device copyable
  - Specialize at your own risk

Device Copyable

- `sycl::is_device_copyable`
  - Defaults to `std::is_trivially_copyable`
  - Specialized for `array`, `pair`, `tuple`, `optional`, `variant`
    - When they contain all device copyable types
    - `array`, `optional`, `variant` already trivially copyable when they contain all trivially copyable types
      - Recursive definition: need to extend it to all device copyable types
  - Specialized for `span`, `basic_string_view`

Unnamed Lambdas

Unnamed Lambdas

```
cgh.parallel_for<class kernel_name>(range<1>{1024}, [=](id<1> idx) {
  writeResult[idx] = idx[0];
});
```

Unnamed Lambdas

```
cgh.parallel_for<class kernel_name>(range<1>{1024}, [=](id<1> idx) {
  writeResult[idx] = idx[0];
});
```

- Weird but valid C++ syntax
  - Forward declaration of a function local class
- SYCL 1.2.1
  - Name every kernel
  - Unique global name for toolchains with separate device compiler

Unnamed Lambdas

```
cgh.parallel_for<class kernel_name>(range<1>{1024}, [=](id<1> idx) {
  writeResult[idx] = idx[0];
});
```

- SYCL 2020
  - No need to specify it
  - Compiler will internally generate a unique name
  - May want to specify it to help with debugging

Unnamed Lambdas

```
cgh.parallel_for                                (range<1>{1024}, [=](id<1> idx) {
  writeResult[idx] = idx[0];
});
```

Unnamed Lambdas

```
cgh.parallel_for(range<1>{1024}, [=](id<1> idx) {
  writeResult[idx] = idx[0];
});
```

Class Template Argument Deduction (CTAD) / Deduction Guides

C++ Class Template Argument Deduction (CTAD)

```cpp
std::pair<int, double> p(2, 3.);
```

Class Template Argument Deduction (CTAD)

```
std::pair<int, double> p(2, 3.);
```

- C++17
- Template parameters can be deduced from the arguments
  - No need to specify them when declaring non-member variables
  - All parameters must be deduced
  - Implicit and user-defined ones

```
template<typename T1, typename T2>
pair(T1, T2) -> pair<T1, T2>;
```

C++ Class Template Argument Deduction (CTAD)

```
std::pair<int, double> p(2, 3.);
```

C++ Class Template Argument Deduction (CTAD)

```
std::pair<int, double> p(2, 3.);
```

C++ Class Template Argument Deduction (CTAD)

```
std::pair                    p(2, 3.);
```

C++ Class Template Argument Deduction (CTAD)

```
std::pair p(2, 3.);
```

C++ Class Template Argument Deduction (CTAD)

```
std::pair p(2, 3.);
```

- Downside
  - Still need to know the exact type (with template parameters) to declare member variables

CTAD

- We added user-defined deduction guides to
  - `id`, `vec`, `buffer`, `multi_ptr`, `range`

Deduction guides for id

```
template <int dimensions = 1>
class id {
    // ...
};

// Deduction guides
id(size_t) -> id<1>;
id(size_t, size_t) -> id<2>;
id(size_t, size_t, size_t) -> id<3>;
```

- Deduction guides not necessarily templates themselves

CTAD for accessors

- Added tagged constructors for accessors
  - Tag is just a (usually empty) type
    - May be a templated type
  - Variable of that type
    - So we can pass it by name

```
struct Tag_t {
    explicit Tag_t() = default; // disallow naked {} syntax
};
inline constexpr Tag_t tag{};
```

CTAD for accessors

```
buffer<int, 2, buffer_allocator> b1{range<2>{2, 5}};
accessor aA{b1, write_only, noinit};
```

CTAD for accessors

```
buffer<int, 2, buffer_allocator> b1{range<2>{2, 5}};
accessor aA{b1, write_only, noinit};
```

- `write_only`
  - Variable of type `mode_tag_t<access::mode::write>`
  - Call tagged constructor via deduction guide

```
template <typename dataT, int dimensions,
          access_mode accessMode =
              (std::is_const_v<dataT> ? access_mode::read
                                      : access_mode::read_write),
          target accessTarget = target::device,
          access::placeholder isPlaceholder = access::placeholder::false_t>
class accessor {
    //...
    template <typename AllocatorT, typename TagT>
    accessor(buffer<dataT, dimensions, AllocatorT>& bufferRef, TagT tag,
             const property_list& propList = {});
};
```

CTAD for accessors

```
buffer<int, 2, buffer_allocator> b1{range<2>{2, 5}};
accessor aA{b1, write_only, noinit};
```

- aA is really

```
accessor<int, 2, access_mode::write, target::device, access::placeholder::true_t>
```

CTAD for accessors

```
buffer<int, 2, buffer_allocator> b1{range<2>{2, 5}};
accessor aA{b1, write_only, noinit};
```

- Bonus CTAD
  - `range`

Resources and References

(Where I shamelessly got information and examples from)

Resources and References

- SYCL 2020 Specification

https://www.khronos.org/registry/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf

- Data Parallel C++ (Reinders, Ashbaugh, Brodman, Kinsner, Pennycook, Tian)

https://link.springer.com/book/10.1007/978-1-4842-5574-2

- Web

https://sycl.tech

https://www.khronos.org/blog/

Q&A

This presentation was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation's exascale computing imperative. Additionally, this presentation used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.