Heterogeneous Modern C++ with SYCL 2020

GORDON BROWN, MICHAEL WONG, NEVIN LIBER, TOM DEAKIN
Distinguished Engineer

- Chair of SYCL Heterogeneous Programming Language
- ISO C++ Directions Group past Chair
- Past CEO OpenMP
- ISO/IEC 14969 Director, VP
- http://isocpp.org/wiki/faq/wg21#michael-wong
- michael@codeplay.com
- fraggamuffin@gmail.com
- Head of Delegation for C++ Standard for Canada
- Chair of Programming Languages for Standards Council of Canada
- Chair of WG21 SG19 Machine Learning
- Chair of WG21 SG14 Games Dev/Low Latency/Financial Trading/Embedded
- Editor: C++ SG5 Transactional Memory Technical Specification
- Editor: C++ SG1 Concurrency Technical Specification
- MISRA C++ and AUTOSAR
- Chair of Standards Council Canada TC22/SC32 Electrical and electronic components (SOTIF)
- Chair of UL4600 Object Tracking
- RISC-V Datacenter/Cloud Computing Chair
- http://wongmichael.com/about
- C++11 book in Chinese: https://www.amazon.cn/dp/B00ETOV2OQ

We build GPU compilers for some of the most powerful supercomputers in the world
Nevin “:-)” Liber
nliber@anl.gov

- Advanced Leadership Computing Facility (ALCF)
  - Computer Scientist
  - Kokkos (SYCL/DPC++ backend)
  - Vice Chair WG21 Library Evolution Working Group Incubator (LEWGI / SG18)
  - SYCL Committee Representative
  - oneAPI, oneMKL Technical Advisory Board Representative

- C++ developer for over three decades
- C++ Committee member for over a decade
  - Former host WG21 (C++) & WG14 (C) meetings

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.
Gordon Brown

Principal Product Owner, oneAPI & Automotive

Currently leading team developing HIP & CUDA backends for DPC++
Background in C++ programming models for heterogeneous systems
  Worked on ComputeCpp (SYCL) since its inception
  Contributor to the Khronos SYCL standard for 8 years
  Contributor to C++ executors and heterogeneity or 5 years
Dr Tom Deakin
Senior Research Associate / Lecturer (fixed term)
High Performance Computing Research Group
Chair Khronos SYCL Advisory Panel
Khronos SYCL Outreach Officer

hpc.tomdeakin.com
Agenda

1. SYCL, oneAPI, and ecosystem
2. SYCL 2020 features
   • Moving with the Times and Any Backend
   • Memory Spaces and Dimensions
   • Reductions and Group Algorithms
3. SYCL futures
SYCL Single Source C++ Parallel Programming

C++ Kernel Fusion can give better performance on complex apps and libs than hand-coding.

Complex ML frameworks can be directly compiled and accelerated.

C++ templates and lambda functions separate host & accelerated device code.

SYCL is ideal for accelerating larger C++-based engines and applications with performance portability.

Accelerated code passed into device OpenCL compilers.
SYCL 2020 is here!
Open Standard for Single Source C++ Parallel Heterogeneous Programming

SYCL 2020 is released after 3 years of intense work
Significant adoption in Embedded, Desktop and HPC markets
Improved programmability, smaller code size, faster performance
Based on C++17, backwards compatible with SYCL 1.2.1
Simplify porting of standard C++ applications to SYCL
Closer alignment and integration with ISO C++
Multiple Backend acceleration and API independent

SYCL 2020 increases expressiveness and simplicity for modern C++ heterogeneous programming
SYCL 2020 Industry Momentum

Argonne Leadership Computing Facility

SYCL and DPC++ for Aurora

NSITEXE, Kyoto Microcomputer, Codeplay Software Are Bringing Open Standards Programming to RISC-V Vector Processor for HPC and AI Systems

SYCL (SYCL announced 2015) is a rapidly-growing cross-platform abstraction layer that enables the efficient and easy-to-use OpenCL-style programming model to be written in a "languages-aware" style using C++, making it easier for developers to write code that runs on a wide range of hardware, from desktops to supercomputers.

SYCL is supported by NVIDIA, AMD, and others, and is supported by open-source projects such as OpenCL and Vulkan. It is also supported by major vendors such as Intel, AMD, and ARM, who have developed SYCL-based tools and libraries to support SYCL.

SYCL support growing from Embedded Systems through Desktops to Supercomputers
SYCL 2020 Major Features

- Unified Shared Memory (USM)
  - Code with pointers can work naturally without buffers or accessors
  - Simplifies porting from most code (e.g. CUDA, C++)

- Parallel Reductions
  - Added built-in reduction operation to avoid boilerplate code and achieve maximum performance on hardware with built-in reduction operation acceleration.

- Work group and subgroup algorithms
  - Efficient parallel operations between work items

- Class template argument deduction (CTAD) and template deduction guides
  - Simplified class template instantiation

- Simplified use of Accessors with a built-in reduction operation
  - Reduces boilerplate code and streamlines the use of C++ software design patterns

- Expanded interoperability
  - Efficient acceleration by diverse backend acceleration APIs

- SYCL atomic operations are now more closely aligned to standard C++ atomics
  - Enhances parallel programming freedom
Parallel Industry Initiatives

- **OpenCL**: 2011
  - OpenCL 1.2
  - OpenCL C Kernel Language

- **SYCL**:
  - SYCL 1.2: 2015
    - C++11 Single source programming
  - SYCL 1.2.1: 2017
    - C++11 Single source programming
    - OpenCL 2.2
    - Many backend options
  - SYCL 2020: 2020
    - C++17 Single source programming
    - OpenCL 3.0
    - Many backend options
  - SYCL 202X: 202X
    - C++20 Single source programming
    - Many backend options

- **C++**:
  - C++11: 2011
  - C++14: 2014
  - C++17: 2017
  - C++20: 2020
  - C++23: 202X
SYCL Implementations in Development

SYCL, OpenCL and SPIR-V, as open industry standards, enable flexible integration and deployment of multiple acceleration technologies.

SYCL enables Khronos to influence ISO C++ to (eventually) support heterogeneous compute.

**Source Code**

- **DPC++**
  - Uses LLVM/Clang
  - Part of oneAPI
  - Intel CPUs
  - Intel GPUs
  - Intel FPGAs
  - NVIDIA GPUs
  - AMD GPUs

- **ComputeCpp**
  - Multiple Backends
  - Intel CPUs
  - Intel GPUs
  - Intel FPGAs
  - AMD GPUs
  - NVIDIA GPUs

- **trISYCL**
  - Open source test bed
  - Intel CPUs
  - Intel GPUs
  - Intel FPGAs
  - Arm Mali
  - IMG PowerVR
  - Renesas R-Car

- **hipSYCL**
  - CUDA and HIP/ROCm
  - NVIDIA GPUs
  - AMD GPUs

- **neoSYCL**
  - SX-AURORA TSUBASA
  - Level Zero
  - Intel CPUs
  - NEC VEs

**Multiple Backends in Development**

There is development on supporting SYCL on even more low-level frameworks.

For more information: [http://sycl.tech](http://sycl.tech)
News: Huawei Bisheng (毕昇) C++
## SYCL Ecosystem, Research and Benchmarks

**Implementations**
- neoSYCL
- SX-AURORA TSUBASA
- oneAPI
- DATA PARALLEL C++
- Celerity
- kokkos
- oneAPI
- triSYCL
- ComputeCpp
- hipSYCL
- GROMACS
- Fast, Flexible, Free.
- Argonne National Laboratory
- Qualcomm
- ARM
- SYCL
- University of Bristol
- Arm
- Codeplay
- Xilinx
- Intel
- TOHOKU
- University of Innsbruck

**Direct Programming Benchmark**
- SYCL-DNN
- Eigen
- oneDNN
- SYCL Parallel STL
- oneDPL

**Books**
- Taskflow: A General-purpose Parallel and Heterogeneous Task Programming System using Modern C++
- ATLAS EXPERIMENT
- RAJA
- SYCL-DNN

**Benchmarks**
- BLAS
- FFT
- Math
- RAND
- SYCLBLAS
- SYCL
- oneMKL
- oneMKL
- oneMKL
- oneMKL
- oneMKL
- oneMKL
- oneMKL
- SYCL-DNN
- Eigen
- oneDNN
- SYCL Parallel STL
- oneDPL

**Machine Learning Libraries and Parallel Acceleration Frameworks**
- BLAS
- FFT
- Math
- RAND
- SYCLBLAS
- oneMKL
- oneMKL
- oneMKL
- oneMKL
- oneMKL
- oneMKL
- oneMKL
- oneMKL
- SYCL-DNN
- Eigen
- oneDNN
- SYCL Parallel STL
- oneDPL

**Working Group Members**

---

*Image of the content with various logos and text.*

---

*Table showing various SYCL ecosystem components.*

---

*Diagram showing the SYCL ecosystem and its components.*

---

*Text listing direct programming benchmarks and books.*

---

*Table showing linear algebra libraries and machine learning frameworks.*

---

*Images of logos from various organizations.*
SYCL in Embedded Systems, Automotive, and AI

- Networks trained on high-end desktop and cloud systems
- Applications link to compiled inferencing code or call vision/inferencing API
- Diverse Embedded Hardware: Multi-core CPUs, GPUs, DSPs, FPGAs, Tensor Cores
  * Vulkan only runs on GPUs
- Open industry standards, enable flexible integration and deployment of multiple acceleration technologies

Diagram:
- Neural Network Training
  - NNEF
  - Training Data
- Trained Networks
- Compiled Code
- Vision / Inferencing Engine
  - OpenVX
- C++ Application Code
  - SYCL
- Hardware Acceleration APIs
  - OpenCL
  - Vulkan
- Sensor Data
- Compilation
- Ingestion

Applications link to compiled inferencing code or call vision/inferencing API.
Safety Critical API Evolution

OpenCL and SYCL SC work will minimize API surface area, reduce ambiguity, UB, increase determinism.

New Generation Safety Critical APIs for Graphics, Compute and Display

UNECE WP.29
Industry Need for GPU Acceleration APIs designed to ease system safety certification is increasing ISO 26262 / ASIL-D

ISO/PAS 21448
UL 4600

Rendering Compute Display
SYCL in HPC/Supercomputers

Simulation
HPC Languages
Solver Libraries, Parallel RT

Data
Productivity Languages
Big Data Stack, Stats Lib, Databases

Learning
Productivity Languages
Deep Learning, Linear Alg, ML

Choose Algorithm for target

Implement and Test Algorithm

Optimize Algorithm

Today’s Supercomputing Development Workflow needs knowledge of system architecture and tools that control data

Need Languages that allow control of these Data Issues
Set Data affinity, Data Layout, Data movement, Data Locality, highly Parameterized Code and dynamically compose the algorithms (C++ templates, parallel STL, inlining and fusion, abstractions)

Libraries augment compiler optimizations for Performance Portable programs

Use open standards to run Performance Portable code on new generation, or different vendor’s, hardware with compiler optimization, explicit parametrization and dynamically composed algorithm

Three Pillars of Science Problem

OpenMP for C and Fortran
CUDA/pthreads/OpenACC/OpenCL
C++ Application uses SYCL, Kokkos, Raja

Math, ML, Data Libraries; C++ Std, C, Python Libraries

OpenMP
OpenCL
SYCL

2021
2022
2022
2023

Perlmutter
Frontier
Archer
Custodian
Pantheo
Agenda

1. SYCL, oneAPI, and ecosystem

2. SYCL 2020 features
   • Moving with the Times and Any Backend
   • Memory Spaces and Dimensions
   • Reductions and Group Algorithms

3. SYCL futures
Moving with the Times
#include <SYCL/sycl.hpp>

int main(int argc, char *argv[]) {
    std::vector<float> dA{ ... }, dB{ ... }, dO{ ... };

    try {
        sycl::queue gpuQueue{sycl::gpu_selector_v, async_handler{});

        sycl::buffer bufA(dA.data(), sycl::range{dA.size()});
        sycl::buffer bufB(dB.data(), sycl::range{dB.size()});
        sycl::buffer bufO(dO.data(), sycl::range{dO.size()});

        gpuQueue.submit([&](sycl::handler &cgh){
            sycl::accessor inA(bufA, cgh, sycl::read_only);
            sycl::accessor inB(bufB, cgh, sycl::read_only);
            sycl::accessor out(bufO, cgh, sycl::write_only);

            cgh.parallel_for(sycl::range{dA.size()},
                             [=](id<1> i){ out[i] = inA[i] + inB[i]; });
        });

        gpuQueue.wait_and_throw();
    } catch (sycl::exception &e) {
        /* handle SYCL exception */
    }
}
First we include the `SYCL/sycl.hpp` header file

This includes the entire SYCL interface

SYCL 2020: Header file has changed from CL/sycl.hpp
In this example we want to add the values of two vectors in parallel on an accelerator

So we allocate two input vectors and an output vector of equal size
In SYCL all work is dispatch to an accelerator via a `queue`

Here we create a queue which will target a GPU device
# include <SYCL/sycl.hpp>

int main(int argc, char *argv[]) {
    std::vector<float> dA{ ... }, dB{ ... }, dO{ ... }

    try {
        sycl::queue gpuQueue{sycl::gpu_selector_v, async_handler{}};

        sycl::buffer bufA{dA.data(), sycl::range{dA.size()}};
        sycl::buffer bufB{dB.data(), sycl::range{dB.size()}};
        sycl::buffer bufO{dO.data(), sycl::range{dO.size()}};

        gpuQueue.submit([&](sycl::handler &cgh){
            sycl::accessor inA(bufA, cgh, sycl::read_only);
            sycl::accessor inB(bufB, cgh, sycl::read_only);
            sycl::accessor out(bufO, cgh, sycl::write_only);

            cgh.parallel_for(sycl::range{dA.size()},
                            [=](id<1> i){ out[i] = inA[i] + inB[i]; });
        });

        gpuQueue.wait_and_throw();
    } catch (sycl::exception &e) {
        /* handle SYCL exception */
    }
}

SYCL 2020 Hello World

The first parameter to construct the queue is a device selector which provides a heuristic for choosing a device

Here we use one of the standard device selectors `gpu_selector_v`

SYCL 2020: Device selectors are now simply a function object rather than requiring inheritance
The second parameter to construct the queue is an async handler which is a function object used to handle asynchronous exceptions thrown by the runtime.

SYCL 2020: If an async handler is not provided the default behaviour is now to terminate on encountering an error.
In SYCL there are two memory management models; buffers and USM - here we will focus on buffers.

The buffers memory management model separates the storage and access of data and automatically handles data dependencies.

USM provides a more explicit memory management.
In the buffers memory management model there are **buffers** and **accessors**

A buffer manages data across the host and one or more devices.

An accessor represents a request to access the data on a particular device with a particular set of properties.

```cpp
#include <SYCL/sycl.hpp>

int main(int argc, char *argv[]) {
    std::vector<float> dA{ ... }, dB{ ... }, dO{ ... };
    try {
        sycl::queue gpuQueue{sycl::gpu_selector_v, async_handler{}};

        sycl::buffer bufA{dA.data(), sycl::range{dA.size()});
        sycl::buffer bufB{dB.data(), sycl::range{dB.size()});
        sycl::buffer bufO{dO.data(), sycl::range{dO.size()});

        gpuQueue.submit([&](sycl::handler &cgh){
            sycl::accessor inA(bufA, cgh, sycl::read_only);
            sycl::accessor inB(bufB, cgh, sycl::read_only);
            sycl::accessor out(bufO, cgh, sycl::write_only);

            cgh.parallel_for(sycl::range{dA.size()}, [=](id<1> i){
                out[i] = inA[i] + inB[i];
            });

            gpuQueue.wait_and_throw();
        });
    } catch (sycl::exception &e) {
        /* handle SYCL exception */
    }
}
```
#include <SYCL/sycl.hpp>

int main(int argc, char *argv[]) {
    std::vector<float> dA{ ... }, dB{ ... }, dO{ ... };

    try {
        sycl::queue gpuQueue{sycl::gpu_selector_v, async_handler{});

        sycl::buffer bufA{dA.data(), sycl::range{dA.size()}};
        sycl::buffer bufB{dB.data(), sycl::range{dB.size()}};
        sycl::buffer bufO{dO.data(), sycl::range{dO.size()}};

        gpuQueue.submit([&](sycl::handler &cgh){
            sycl::accessor inA(bufA, cgh, sycl::read_only);
            sycl::accessor inB(bufB, cgh, sycl::read_only);
            sycl::accessor out(bufO, cgh, sycl::write_only);

            cgh.parallel_for(sycl::range{dA.size()}, [=](id<1> i){
                out[i] = inA[i] + inB[i];
            });

            gpuQueue.wait_and_throw();
        });
    } catch (sycl::exception &e) {
        /* handle SYCL exception */
    }
}
Buffers manage data during their lifetime. On destruction of the buffer, it will wait for any work the queue is doing which accesses the buffer’s data. It will then synchronize the data back to the original host pointer.
#include <SYCL/sycl.hpp>

int main(int argc, char *argv[]) {
  std::vector<float> dA{ ... }, dB{ ... }, dO{ ... };

  try {
    sycl::queue gpuQueue{sycl::gpu_selector_v, async_handler{}};

    sycl::buffer bufA(dA.data(), sycl::range{dA.size()});
    sycl::buffer bufB(dB.data(), sycl::range{dB.size()});
    sycl::buffer bufO(dO.data(), sycl::range{dO.size()});

    gpuQueue.submit([&](sycl::handler &cgh){
      sycl::accessor inA(bufA, cgh, sycl::read_only);
      sycl::accessor inB(bufB, cgh, sycl::read_only);
      sycl::accessor out(bufO, cgh, sycl::write_only);

      cgh.parallel_for(sycl::range{dA.size()},
                       [=](id<1> i){ out[i] = inA[i] + inB[i]; });
    });

    gpuQueue.wait_and_throw();
  } catch (sycl::exception &e) {
    /* handle SYCL exception */
  }
}
Inside the command group we create accessors for each buffer

Accessors are used to create dependencies to the data managed by their respective buffers

They are also used to access the data once on the device

```c++
#include <SYCL/sycl.hpp>

int main(int argc, char *argv[]) {
    std::vector<float> dA{ ... }, dB{ ... }, dO{ ... };

    try {
        sycl::queue gpuQueue{sycl::gpu_selector_v, async_handler{}};

        sycl::buffer bufA(dA.data(), sycl::range{dA.size()});
        sycl::buffer bufB(dB.data(), sycl::range{dB.size()});
        sycl::buffer bufO(dO.data(), sycl::range{dO.size()});

        gpuQueue.submit([&](sycl::handler &cgh){

            sycl::accessor inA(bufA, cgh, sycl::read_only);
            sycl::accessor inB(bufB, cgh, sycl::read_only);
            sycl::accessor out(bufO, cgh, sycl::write_only);

            cgh.parallel_for(sycl::range{dA.size()},
                [=](id<1> i){ out[i] = inA[i] + inB[i]; });

            gpuQueue.wait_and_throw();
        })
    } catch (sycl::exception &e) {
        /* handle SYCL exception */
    }
}
The accessors are constructed from their respective buffer, the handler and a tag which describes the access mode.

The access mode is used by the SYCL runtime to manage data dependencies.

SYCL 2020: Accessors have CTAD and tags so no longer require explicit template parameters.
Inside the command group we also define the kernel function by calling

```cpp
parallel_for()
```

This defines a device function which is the entry point for the code which runs on the accelerator

```
SYCL 2020: Kernel functions no longer require a template parameter to name them
```
The first parameter to `parallel_for` is a `range` which describes the iteration space the kernel will execute across.

The second parameter to `parallel_for` is a function object which defines the kernel function itself.

The kernel function takes as a parameter an `id` which represents the currently executing work-item.
Here the kernel function is a lambda

The body of the lambda is what is executed on the accelerator

If using a lambda accessors and other arguments must be captures by value
SYCL 2020 Hello World

#include <SYCL/sycl.hpp>

int main(int argc, char *argv[]) {
    std::vector<float> dA{ ... }, dB{ ... }, dO{ ... };

    try {
        sycl::queue gpuQueue{sycl::gpu_selector_v, async_handler{}};

        sycl::buffer bufA(dA.data(), sycl::range{dA.size()});
        sycl::buffer bufB(dB.data(), sycl::range{dB.size()});
        sycl::buffer bufO(dO.data(), sycl::range{dO.size()});

        gpuQueue.submit([&](sycl::handler &cgh){

            sycl::accessor inA(bufA, cgh, sycl::read_only);
            sycl::accessor inB(bufB, cgh, sycl::read_only);
            sycl::accessor out(bufO, cgh, sycl::write_only);

            cgh.parallel_for(sycl::range{dA.size()},
                [=](id<1> i){
                    out[i] = inA[i] + inB[i];
                });

            gpuQueue.wait_and_throw();
        });
    } catch (sycl::exception &e) {
        /* handle SYCL exception */
    }
}
Finally we call `wait_and_throw()` to wait for the command group to complete

The `wait_and_throw()` function will also throw any asynchronous errors encountered.
Any synchronous errors which are encountered are thrown immediately

Any asynchronous errors which are encountered are stored and then thrown when `wait_and_throw()` is called

It’s important to have a try-catch block to catch and handle any exceptions
#include <SYCL/sycl.hpp>

int main(int argc, char *argv[]) {
    std::vector<float> dA{ ... }, dB{ ... }, dO{ ... };

    try {
        sycl::queue gpuQueue{sycl::gpu_selector_v, async_handler{}};

        sycl::buffer bufA{dA.data(), sycl::range{dA.size()}};
        sycl::buffer bufB{dB.data(), sycl::range{dB.size()}};
        sycl::buffer bufO{dO.data(), sycl::range{dO.size()}};

        gpuQueue.submit([&](sycl::handler &cgh){
            sycl::accessor inA(bufA, cgh, sycl::read_only);
            sycl::accessor inB(bufB, cgh, sycl::read_only);
            sycl::accessor out(bufO, cgh, sycl::write_only);

            cgh.parallel_for(sycl::range{dA.size()},
                [=](id<1> i){ out[i] = inA[i] + inB[i]; });
        });

        gpuQueue.wait_and_throw();
    }
    catch (sycl::exception &e) {
        /* handle SYCL exception */
    }
}
Any Backend
The SYCL topology has been extended to include backends.

Device selectors can be used to find devices from any backend.
The host device is now available via the `aspect::emulated` aspect.
The same device can exist in more than one backend.
An application written to the core SYCL specification is portable to any SYCL implementation.
An application written using the backend-specific interoperability API is portable to any SYCL implementation which supports that backend.
An application written using a vendor-specific extension is not portable and can only be run by that vendor’s SYCL implementation.
Memory Spaces
SYCL Memory Model

- Initially based on OpenCL Memory Model
- Higher level of abstraction

- Buffers & Accessors
  - Buffers
    - Storage
  - Accessors
    - Access
  - Compiler automatically builds a DAG at runtime

- USM (Unified Shared Memory)
  - Explicit data allocation and movement
  - C++ pointers
Memory Spaces

Buffers & Accessors
SYCL Memory Model - Buffers & Accessors

- Buffers & Accessors
  - Private memory
  - Local memory
  - Global memory
  - Generic memory
  - Constant memory
SYCL Memory Model - Buffers & Accessors

- **Private memory**
  - Private to a *work item*
  - Not accessible to other *work items*
  - Kernel memory
  - Local stack variables

- **Work item**
  - An instance of a kernel
  - Has a global id
SYCL Memory Model - Buffers & Accessors

- **Local memory**
  - Accessible to all work items in a work group
  - For variables shared across work items

- **Work group**
  - Related work items
  - Single compute unit
SYCL Memory Model - Buffers & Accessors

- **Global memory**
  - Accessible to all work items in all work groups
  - Persistent across kernel invocations
  - Different devices (in the same context) can access the memory
  - No implied synchronization for simultaneous writes from two different kernels
SYCL Memory Model - Buffers & Accessors

- Generic memory
  - Virtual address space encompassing the global, local and private address spaces
SYCL Memory Model - Buffers & Accessors

- **Constant memory**
  - OpenCL platforms
  - Remains constant during execution of a kernel
  - Not part of *generic memory*
  - *multi_ptr* support for *constant memory* deprecated in SYCL 2020
SYCL Memory Model - Buffers & Accessors

```cpp
enum class address_space : int {
  global_space,
  local_space,
  constant_space, // Deprecated in SYCL 2020
  private_space,
  generic_space,
};

enum class decorated : int { /* ... */ };

template<typename ElementType, access::address_space Space, Access::decorated DecorateAddress>
class multi_ptr { /* ... */ };
```

- **multi_ptr**
  - Address space boundaries
  - Interoperability
- **Raw pointers**
  - Implementation defined
  - Sometimes requires explicit decoration for address space
    - Platform dependent keyword
enum class address_space : int {
    global_space,
    local_space,
    constant_space, // Deprecated in SYCL 2020
    private_space,
    generic_space,
};

enum class decorated : int { /* ... */ };

template <typename ElementType, 
    access::address_space Space, 
    Access::decorated DecorateAddress>
class multi_ptr { /* ... */ };
SYCL Memory Model - Buffers & Accessors

```cpp
enum class address_space : int {
    global_space,
    local_space,
    constant_space, // Deprecated in SYCL 2020
    private_space,
    generic_space,
};

enum class decorated : int { /* ... */ };

template <typename ElementType, access::address_space Space, Access::decorated DecorateAddress>
class multi_ptr { /* ... */ };
```

- **multi_ptr**
  - Address space boundaries
  - Interoperability
- **Raw pointers**
  - Implementation defined
  - Sometimes requires explicit decoration for address space
    - Platform dependent keyword
Memory Spaces

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)

<table>
<thead>
<tr>
<th>Allocation Type</th>
<th>Description</th>
<th>Accessible on host?</th>
<th>Accessible on device?</th>
<th>Located on</th>
</tr>
</thead>
<tbody>
<tr>
<td>device</td>
<td>Allocations in device memory</td>
<td>X</td>
<td>√</td>
<td>device</td>
</tr>
<tr>
<td>host</td>
<td>Allocations in host memory</td>
<td>√</td>
<td>√</td>
<td>host</td>
</tr>
<tr>
<td>shared</td>
<td>Allocations shared between host and device</td>
<td>√</td>
<td>√</td>
<td>can migrate between host and device</td>
</tr>
</tbody>
</table>
Unified Shared Memory (USM)

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

<table>
<thead>
<tr>
<th>Allocation Type</th>
<th>Description</th>
<th>Accessible on host?</th>
<th>Accessible on device?</th>
<th>Located on</th>
</tr>
</thead>
<tbody>
<tr>
<td>device</td>
<td>Allocations in device memory</td>
<td>X</td>
<td>✓</td>
<td>device</td>
</tr>
<tr>
<td>host</td>
<td>Allocations in host memory</td>
<td>✓</td>
<td>✓</td>
<td>host</td>
</tr>
<tr>
<td>shared</td>
<td>Allocations shared between host and device</td>
<td>✓</td>
<td>✓</td>
<td>can migrate between host and device</td>
</tr>
</tbody>
</table>
Unified Shared Memory (USM)

- **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

<table>
<thead>
<tr>
<th>Allocation Type</th>
<th>Description</th>
<th>Accessible on host?</th>
<th>Accessible on device?</th>
<th>Located on</th>
</tr>
</thead>
<tbody>
<tr>
<td>device</td>
<td>Allocations in device memory</td>
<td>X</td>
<td>✓</td>
<td>device</td>
</tr>
<tr>
<td>host</td>
<td>Allocations in host memory</td>
<td>✓</td>
<td>✓</td>
<td>host</td>
</tr>
<tr>
<td>shared</td>
<td>Allocations shared between host and device</td>
<td>✓</td>
<td>✓</td>
<td>can migrate between host and device</td>
</tr>
</tbody>
</table>


Unified Shared Memory (USM)

- **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 migration early
      - mem_advise

<table>
<thead>
<tr>
<th>Allocation Type</th>
<th>Description</th>
<th>Accessible on host?</th>
<th>Accessible on device?</th>
<th>Located on</th>
</tr>
</thead>
<tbody>
<tr>
<td>device</td>
<td>Allocations in device memory</td>
<td>X</td>
<td>✓</td>
<td>device</td>
</tr>
<tr>
<td>host</td>
<td>Allocations in host memory</td>
<td>✓</td>
<td>✓</td>
<td>host</td>
</tr>
<tr>
<td>shared</td>
<td>Allocations shared between host and device</td>
<td>✓</td>
<td>✓</td>
<td>can migrate between host and device</td>
</tr>
</tbody>
</table>
Unified Shared Memory (USM)

- Allocation styles
  - C
    - `malloc`, `aligned_alloc`, `malloc_host`, etc.
  - Specify size of 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
Device Copyable
Device Copyable

- How can we copy objects between a host or a device and another device?

- **C++**
  - Copy constructor `T::T(T const&)`
  - Running code

- **SYCL**
  - Where (what device) would the copy constructor run?
  - How would it gain access to both the source and destination bytes?
  - About all we can do in general is copy the bytes
Device Copyable

- **Trivially copyable**
  - Proxy
  - Bitwise copy (copy the bytes)
  - Some obvious C++ types aren’t trivially copyable for historical reasons
    - `pair<TriviallyCopyableTypes...>`
    - `tuple<TriviallyCopyableTypes...>`

- Trivially copyable has other (unnecessary) requirements
  - At least one copy/move constructor/assignment operator
  - Destructor and non-deleted copy/move constructor/assignment operators must be public and trivial
  - Layout & ABI conflated into trivially copyable
    - Why we can’t fix `pair`
Device Copyable

```cpp
struct A {
    #ifndef __SYCL_DEVICE_ONLY__
        ~A() {} // Not trivially copyable
    #endif
};

static_assert(std::is_trivially_copyable_v<A>);

template <bool B> void C() { /* ... */ }
C<std::is_trivially_copyable_v<A>>();
```

- **__SYCL_DEVICE_ONLY__**
  - Can violate the C++ One Define Rule (ODR)
  - `static_assert` only fires on the host
  - What is the value of that template parameter to `C`?
  - What does it mean to run the destructor on the host but not the device?
Device Copyable

```cpp
struct A {
    #ifndef __SYCL_DEVICE_ONLY__
        ~A() {} // Not trivially copyable
    #endif

    static_assert(std::is_trivially_copyable_v<A>);

    template <bool B> void C() { /* ... */ }
    C<std::is_trivially_copyable_v<A>>();
```

- Can violate the C++ One Define Rule (ODR)
- `static_assert` only fires on the host
- What is the value of that template parameter to `C`?
- What does it mean to run the destructor on the host but not the device?
Device Copyable

```cpp
struct A {
    #ifndef __SYCL_DEVICE_ONLY__
    ~A() {} // Not trivially copyable
    #endif
};

static_assert(std::is_trivially_copyable_v<A>);

template <bool B> void C() { /* ... */ }
C<std::is_trivially_copyable_v<A>>();
```

- **__SYCL_DEVICE_ONLY__**
  - Can violate the C++ One Define Rule (ODR)
  - static_assert only fires on the host
  - What is the value of that template parameter to C?
  - What does it mean to run the destructor on the host but not the device?
Device Copyable

```cpp
struct A {
    #ifndef __SYCL_DEVICE_ONLY__
    ~A() {} // Not trivially copyable
    #endif
};

static_assert(std::is_trivially_copyable_v<A>);

template <bool B> void C() { /* ... */ }
C<std::is_trivially_copyable_v<A>>();
```

- **__SYCLDEVICEONLY__**
  - Can violate the C++ One Define Rule (ODR)
  - `static_assert` only fires on the host

- What is the value of that template parameter to C?

- What does it mean to run the destructor on the host but not the device?
Device Copyable

```cpp
struct A {
#ifndef __SYCL_DEVICE_ONLY__
    ~A() {} // Not trivially copyable
#endif
};

static_assert(std::is_trivially_copyable_v<A>);

template <bool B> void C() { /* ... */ }
C<std::is_trivially_copyable_v<A>>();
```

- **__SYCL_DEVICE_ONLY__**
  - Can violate the C++ One Define Rule (ODR)
  - `static_assert` only fires on the host
  - What is the value of that template parameter to `C`?
  - What does it mean to run the destructor on the host but not the device?
Device Copyable

```cpp
struct A {
  #ifndef __SYCL_DEVICE_ONLY__
    ~A() {} // Not trivially copyable
  #endif
};

static_assert(std::is_trivially_copyable_v<A>);

template <bool B> void C() { /* ... */ }
C<std::is_trivially_copyable_v<A>>();

- __SYCL_DEVICE_ONLY__
  - Can violate the C++ One Define Rule (ODR)
  - static_assert only fires on the host
  - What is the value of that template parameter to C?
  - What does it mean to run the destructor on the host but not the device?
```
Device Copyable

- Manually copy the bytes to the device
  - Violates C++ Object Model (lifetime of objects)
  - Copying does not magically bring (non trivially copyable / non implicit lifetime) types into existence
  - Undefined behavior
    - May work today, but can easily break tomorrow
Device Copyable

- **is_device_copyable**
  - Defaults to `std::is_trivially_copyable`
  - Specialized for `array`, `optional`, `pair`, `tuple`, `variant` of *device copyable* types
    - Like trivially copyable, we need the recursive definition
    - C++ doesn’t yet have reflection
  - Users can specialize it if their type can be bitwise copied
    - At their own risk!
  - Unspecified if/where copy/move constructor/assignment and destructor are run
atomic_ref
atomic_ref

● C++11 std::atomic
  ○ Always defaults to seq_cst
    ■ Safety / easiest to reason about
  ○ Owns the data
  ○ Always requires atomic access
    ■ Over-constraint
  ○ Cannot be moved or copied
    ■ Cannot create objects on the host and copy them to the device

● SYCL 1.2.1 cl::sycl::atomic
  ○ Modeled on std::atomic
    ■ Does not own its data
    ■ Defaults to relaxed
      ● Compatible with OpenCL
  ○ Deprecated in SYCL 2020
**atomic_ref**

- **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**
  - std::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
      - void A(std::string_view) { /* ... */ }
  - Replace C++11 interfaces for A
atomic_ref

● C++20
  ○ std::span
    ■ Non-owning reference to contiguous data
  ○ Ranges
    ■ Non-owning views
  ○ std::atomic_ref
    ■ Non-owning atomic access to data
    ■ User promises only atomic access to data for lifetime of atomic_ref object

● SYCL 2020
  ○ sycl::atomic_ref
    ■ Non-owning atomic access to data
    ■ Modeled on C++20 std::atomic_ref
      ○ 3 more template arguments
    ■ SYCL only required to support relaxed
    ■ May support acq_rel and seq_cst
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;
    //...
};}
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++20**
  - T is trivially copyable
- **SYCL 2020**
  - int, unsigned int, long, unsigned long, long long, unsigned long long, float, double
  - Whether or not float & double supported natively
  - SYCL requires 32-bit support; 64-bit optional
  - No support for large types
    - complex<double>
    - Might require globally accessible lock table
atomic_ref

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
  - Why no default?
    - C++ seq_cst rules out lots of devices
    - Least common denominator relaxed leads to surprises when migrating existing code
  - default_read_order / default_write_order many be different than DefaultOrder
    - DefaultOrder == acq_rel -> default_read_order == acquire
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`
  - Why no default?
    - Safest `system` rules out lots of devices
    - Least common denominator `work_group` leads to surprises when migrating existing code
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
- Where the referenced object is allocated
  - generic_space, global_space, local_space
- Override default of generic_space for performance tuning
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;
    //...
};}
Multidimensional subscript & mdspan

C++23
Multidimensional subscript operator[]

- C++ 20 deprecated a comma expression within square brackets
  - `a[b, c];` // deprecated; uses c as index/key
  - `a[(b, c)]` // ok; uses c as index/key
- Comma expression now ill-formed in C++23
- Previous workarounds
  - `a(x, y, z)` // function call operator taking multiple indices
  - `A[x][y][z]` // chain of single argument array access operators
  - `a[{x, y, z}]` // array index operator taking a tuple-like index
- Definition & use match `operator()`
  - `A[b, c];` // calls `A::operator[](B, C);` uses b, c as two ordered indices
- No change to C arrays
mdspan

```cpp
template<
    class ElementType,
    class Extents,
    class LayoutPolicy = layout_right,
    class AccessorPolicy =
        default_accessor<ElementType>>
struct mdspan {
    //...
    template<class... SizeTypes>
    constexpr reference operator[](SizeTypes... indices) const
    { /* ... */ }
    //...
};
```

- **std::mdspan**
  - *Non-owning* multi-dimensional array view
  - First standard library type to use multidimensional subscript operator[]
mdspan

```
template<
class ElementType, class Extents,
  class LayoutPolicy = layout_right,
  class AccessorPolicy =
default_accessor<ElementType>>
struct mdspan {
  //...
  template<class... SizeTypes>
  constexpr reference operator[](SizeTypes... indices) const
  { /* ... */ }
  //...
};
```

- `std::mdspan`
  - **Non-owning**
    - multi-dimensional array view
  - First standard library type to use multidimensional subscript operator[

mdspan

```cpp
template<class ElementType, class Extents,
    class LayoutPolicy = layout_right,
    class AccessorPolicy =
default_accessor<ElementType>>
struct mdspan {
    //...
    template<class... SizeTypes>
    constexpr reference operator[](SizeTypes... indices) const
    { /* ... */ }
    //...
};
```

- `std::mdspan`
  - *Non-owning multi-dimensional array view*
  - First standard library type to use multidimensional subscript `operator[]`
mdspan

```cpp
template<class ElementType, class Extents, 
    class LayoutPolicy = layout_right, 
    class AccessorPolicy = 
default_accessor<ElementType>>
struct mdspan {
    //...
    template<class... SizeTypes>
    constexpr reference operator[](SizeTypes... indices) const
    { /* ... */ }
    //...
};
```

- **std::mdspan**
  - **Extents**
    - Describe each dimension
    - Both for sizes known at compile time and run time
  - **LayoutPolicy**
    - Layout_right
    - layout_left
    - FORTRAN
    - layout_stride
  - **AccessorPolicy**
```cpp
template<class ElementType, class Extents, 
    class LayoutPolicy = layout_right, 
    class AccessorPolicy = 
    default_accessor<ElementType>>
struct mdspan {
    //...
    template<class... SizeTypes>
    constexpr reference operator[](SizeTypes... indices) const
    { /* ... */ }
    //...
};
```

- **std::mdspan**
  - **Extents**
    - Describe each dimension
    - Both for sizes known at compile time and run time
  - **LayoutPolicy**
    - Layout_right
    - C
    - layout_left
    - FORTRAN
    - layout_stride
  - **AccessorPolicy**
mdspan

```cpp
template<class ElementType, class Extents,
    class LayoutPolicy = layout_right,
    class AccessorPolicy = 
default_accessor<ElementType>>
struct mdspan {
    //...
    template<class... SizeTypes>
    constexpr reference operator[](SizeTypes... indices) const
    { /* ... */ }
    //...
};
```

- **std::mdspan**
  - **Extents**
    - Describe each dimension
    - Both for sizes known at compile time and run time
  - **LayoutPolicy**
    - **C**
      - layout_left
    - **FORTRAN**
      - layout_stride
  - **AccessorPolicy**
**mdspan**

```cpp
template<class ElementType, class Extents, 
    class LayoutPolicy = layout_right, 
    class AccessorPolicy = 
default_accessor<ElementType>>
struct mdspan {
    //...
    template<class... SizeTypes>
    constexpr reference operator[](SizeTypes... indices) const
    { /* ... */ }
    //...
};
```

- **std::mdspan**
  - **Extents**
    - Describe each dimension
    - Both for sizes known at compile time and run time
  - **LayoutPolicy**
    - Layout_right
    - layout_left
    - FORTRAN
    - layout_stride
  - **AccessorPolicy**
mdspan

\[
\text{template<class ElementType, class Extents, class LayoutPolicy = layout_right, class AccessorPolicy = default_accessor<ElementType> > }
\text{struct mdspan \{} \\
\quad //...
\quad \text{template<class... SizeTypes>}
\quad \text{constexpr reference operator[]}(\text{SizeTypes... indices}) \text{ const} \\
\quad \quad \{ /* ... */ \} \\
\quad //...
\}\; \]

- \text{std::mdspan}
  - \text{Extents}
    - Describe each dimension
    - Both for sizes known at compile time and run time
  - \text{LayoutPolicy}
    - \text{Layout_right}
    - \text{FORTRAN}
    - \text{layout_left}
    - \text{layout_stride}
  - \text{AccessorPolicy}
mdspan

```cpp
template<class ElementType, class Extents,
         class LayoutPolicy = layout_right,
         class AccessorPolicy =
default_accessor<ElementType>>
struct mdspan {
    //...
    template<class... SizeTypes>
    constexpr reference operator[](SizeTypes... indices) const
    { /* ... */ }
    //...
};
```

- `std::mdspan`
  - SYCL design
    - Early Days!
    - More than three dimensions
      - How many?
    - Accessors
      - `accessor`-like indexing to USM pointer
    - Strides, offset and sub_buffers
Reductions and Group Algorithms
Reductions

- Commonly occurring parallel pattern.
- Combine input to produce a single value:
  - E.g. for (int i = 0; i < N; ++i) { r += a[i] * b[i]; }
  - std::transform_reduce(std::execution::par_unseq, a.begin(), a.end(), b.begin(), 0.0);

- Reduction variable is output of the reduction.
  - The variable that accumulates results from multiple iterations.
  - Implementations may make zero or more copies.
- Reduction operator used to accumulate/combine copies of reduction variable.
SYCL 1.2.1: reductions with atomic

1: cl::sycl::queue Q;
2: int N = 1024;
3: cl::sycl::buffer<int> bufA {N}; // input array
4: { // Initialize to integers 1..N inclusive.
5:     auto accA = bufA.get_access();
6:     std::iota(accA.begin(), accA.end(), 1);
7: }
8:
9: cl::sycl::buffer<int> bufSum {1}; // reduction result
10: Q.submit([&](sycl::handler &cgh) {
11:     auto accA = bufA.get_access<cl::sycl::access::mode::read>(cgh);
12:     auto sum = bufSum.get_access<cl::sycl::access::mode::atomic>(cgh);
13:     cgh.parallel_for(cl::sycl::range<1>(N),
14:             [=](cl::sycl::id<1> id) {
15:                 sum[0].fetch_add(accA[id]); // Atomic
16:             });
17:     assert(bufSum.get_access()[0] == (N * (N+1) / 2)); // Collect result on host
SYCL execution and memory model: recap

- Kernel invocations: work-items
- Work-items collected into work-groups, forming a 1-3 NDimensional grid of work-items
- Work-items can only synchronize within work-group
- Work-items collected into 1D sub-groups with extra execution guarantees:
  - Work-items in sub-group execute concurrently
  - Sub-groups make independent forward progress w.r.t other sub-groups in work-group
- All work-items have access to Global memory
- Work-items in a work-group can access Local memory
SYCL 1.2.1: reductions with local memory

```cpp
1:   cl::sycl::queue Q;
2:   int N = 1024; assert(N % 16 == 0); // Run with work-group size of 16
3:   cl::sycl::buffer<int> bufA {N}; { auto accA = bufA.get_access(); std::iota(accA.begin(), accA.end(), 1);}
4:   cl::sycl::buffer<int> bufSum {N/16}; // global buffer of one partial value per work-group
5:   Q.submit([&](sycl::handler &cgh) {
6:     auto accA = bufA.get_access<cl::sycl::access::mode::read>(cgh);
7:     auto sum = bufSum.get_access<cl::sycl::access::mode::write>(cgh);
8:     auto wg_sum = cl::sycl::accessor<int, 1, cl::sycl::access::mode::read_write, cl::sycl::access::target::local>(cl::sycl::range<1>(16), cgh); // local mem
9:       cgh.parallel_for(cl::sycl::nd_range<1>{N, 16}, [=](cl::sycl::nd_item<1> item) {
10:         // Copy to local memory, then reduce in work-group
11:         wg_sum[item.get_local_id(0)] = accA[item.get_global_id()];
12:         for (int off = item.get_local_range()[0] / 2; off > 0; off /= 2) {
13:             item.barrier(cl::sycl::access::fence_space::local_space);
14:             if (item.get_local_id(0) < off)
15:                 wg_sum[item.get_local_id(0)] += wg_sum[item.get_local_id(0) + off];
16:         }
17:         if (item.get_local_id(0) == 0) sum[item.get_group(0)] = wg_sum[0]; // Write partial to global buffer
18:     });
19:   });
20:   { auto sum = bufSum.get_access();
21:     assert(std::accumulate(sum.begin(), sum.end(), 0) == (N * (N+1) / 2)); } // Host accumulates partials
```


Group algorithms

- Updated interface for broadcast and barrier functions.
- C++17-based algorithm library for work-group collectives.
- Often two versions:
  - Work-items supply single value directly.
  - Work-items work collectively on input iterator.
- Algorithms:
  - any_of() / all_of() / none_of()
  - shift_group_left() / shift_group_right()
  - permute_group_by_xor()
  - select_from_group()
  - reduce() / inclusive_scan() / exclusive_scan()

```cpp
template <
    typename Group, typename Ptr,
    typename BinaryOperation>
std::iterator_traits<Ptr>::value_type
joint_reduce(
    Group g, Ptr first, Ptr last,
    BinaryOperation binary_op);

template <
    typename Group, typename Ptr,
    typename BinaryOperation>
T
reduce_over_group(
    Group g, T x,
    BinaryOperation binary_op);
```
SYCL 2020: group reductions

```cpp
1:   sycl::queue Q;
2:   int N = 1024; assert(N % 16 == 0);
3:   sycl::buffer<int> bufA {N}; { sycl::host_accessor accA{bufA}; std::iota(accA.begin(), accA.end(), 1);}
4: 
5:   sycl::buffer<int> bufSum {N/16}; // partial values
6:   Q.submit([&](sycl::handler &cgh) {
7:     sycl::accessor accA {bufA, cgh, sycl::read_only};
8:     sycl::accessor sum {bufSum, cgh, sycl::write_only};
9:     cgh.parallel_for(sycl::nd_range<1>{N, 16},
10:    [=](sycl::nd_item<1> item) {
11:      // Group algorithm
12:      int partial_sum =
13:          sycl::reduce_over_group(item.get_group(), accA[item.get_global_id()], std::plus<>());
14:      if (item.get_group().leader()) sum[item.get_group(0)] = partial_sum;
15:    });
16: 
17:   auto sum = bufSum.get_host_access();
18:   assert(std::accumulate(sum.begin(), sum.end(), 0) == (N * (N+1) / 2)); // Host accumulates partials
19: });
```
Reduction API in SYCL 2020

- `sycl::reduction` function to add reduction semantics to a variable.
  - Specify binary combination operation, and identity value if not known.
  - Variable contains final result when kernel finishes executing.
  - Pass return value to kernel invocation as a parameter pack.
    - i.e. tell the `parallel_for` it's a reduction.

- Implementation defined reducer class to wrap reduction variable inside kernel
  - Programmer adds `auto&` argument to lambda function.
  - Type restricts use to parallel-safe `combine()` only (equiv. `operator+=`, etc).
  - Implementation responsible for combining with the original variable.

- Implementation given flexibility to combine reducers.

- Limited to compile-time size variables:
  - single variables.
  - `std::span` 1D-array reductions using USM.
SYCL 2020: reductions

```cpp
1:   sycl::queue Q;
2:   int N = 1024;
3:   sycl::buffer<int> bufA {N};
4:   { // Initialize to integers 1..N inclusive.
5:     sycl::host_accessor accA {bufA};
6:     std::iota(accA.begin(), accA.end(), 1);
7:   }
8:
9:   sycl::buffer<int> bufSum {1};            // reduction variable
10:  Q.submit([&](sycl::handler &cgh) {
11:    sycl::accessor accA {bufA, cgh, sycl::read_only};
12:    cgh.parallel_for(1024,
13:      sycl::reduction(bufSum, cgh, std::plus<>(),
14:        [=](sycl::id<1> id, auto &sum) {  // kernel variable on lambda
15:          sum += accA[id];
16:        }); });
17:  assert(bufSum.get_host_access()[0] == (N * (N+1) / 2));  // Host reads final value
```
Agenda

1. SYCL, oneAPI, and ecosystem

2. SYCL 2020 features
   • Moving with the Times and Any Backend
   • Memory Spaces and Dimensions
   • Reductions and Group Algorithms

3. SYCL futures
oneAPI and SYCL

- SYCL sits at the heart of oneAPI
- Provides an open standard interface for developers
- Defined by the industry
Nvidia and AMD Support in oneAPI

• Extending DPC++ to target Nvidia and AMD GPUs

• Supporting Perlmutter, Polaris and Frontier supercomputers

• Open source and available to everyone

https://www.codeplay.com/oneapiforcuda
Resources for AMD coming soon
“this work supports the productivity of scientific application developers and users through performance portability of applications between Aurora and Perlmutter.”

Codeplay works in partnership with US National Laboratories to enable SYCL on exascale supercomputers

Enables a broad range of software frameworks and applications
SYCL Future Evolution

SYCL 2020 compared with SYCL 1.2.1
- Easier to integrate with C++17 (CTAD, Deduction Guides...)
- Less verbose, smaller code size, simplify patterns
- Backend independent
- Multiple object archives aka modules simplify interoperability
- Ease porting C++ applications to SYCL
- Enable capabilities to improve programmability
- Backwards compatible but minor API break based on user feedback

SYCL Future Roadmap (MAY CHANGE)

Improving Software Ecosystem
Books, Tutorials, Tool, libraries, GitHub

Expanding Implementations
- DPC++
- ComputeCpp
- triSYCL
- hipSYCL
- neoSYCL

Regular Maintenance Updates
- Spec clarifications, formatting and bug fixes
- https://www.khronos.org/registry/SYCL/

Conformance Tests
Working on Implementations
Future SYCL NEXT Proposals

Integration of successful Extensions plus new Core functionality

CONVERGE SYCL with ISO C++ and continue to support OpenCL to deploy on more devices
- CPU
- GPU
- FPGA
- AI processors
- Custom Processors

Repeat The Cycle every 1.5-3 years

Over 40 Selected Features for SYCL 2020
- Unified Shared Memory
- Parallel Reductions add a built-in reduction operation
- Work-group and sub-group algorithms
- Improvements to atomic operations
- Class template argument deduction (CTAD) and deduction guides
- Simplification of accessors
- Expanded interoperability with different backends
- Extension mechanism
- Address spaces
- Vector rework
- Specialization Constants
...
SYCL 2020 is here!
Open Standard for Single Source C++ Parallel Heterogeneous Programming

SYCL 2020 is released after 3 years of intense work
Significant adoption in Embedded, Desktop and HPC markets
Improved programmability, smaller code size, faster performance
Based on C++17, backwards compatible with SYCL 1.2.1
Simplify porting of standard C++ applications to SYCL
Closer alignment and integration with ISO C++
Multiple Backend acceleration and API independent

SYCL 2020 increases expressiveness and simplicity for modern C++ heterogeneous programming
Enabling Industry Engagement

- SYCL working group values industry feedback
  - https://community.khronos.org/c/sycl
  - https://sycl.tech

- SYCL FAQ
  - https://www.khronos.org/blog/sycl-2020-what-do-you-need-to-know

- What features would you like in future SYCL versions?
  - Advisory Panel
    - Chaired by Tom Deakin of U of Bristol
  - SYCL Advisory Panel meeting here at IWOCL/SYCLCon
  - Regular meetings to give feedback on roadmap and draft specifications

Public contributions to Specification, Conformance Tests and software
https://github.com/KhronosGroup/SYCL-CTS
https://github.com/KhronosGroup/SYCL-Docs
https://github.com/KhronosGroup/SYCL-Shared
https://github.com/KhronosGroup/SYCL-Registry
https://github.com/KhronosGroup/SyclParallelSTL

Kronos SYCL Forums, Slack Channels, Stackoverflow, reddit, and SYCL.tech
Open to all!
https://community.khronos.org/
https://www.khr.io/slack
https://app.slack.com/client/TDMDFS87M/CE9UX4CHG
https://community.khronos.org/c/sycl/
https://stackoverflow.com/questions/tagged/sycl
https://www.reddit.com/r/sycl
https://github.com/codeplaysoftware/syclacademy
https://sycl.tech/

Invited Experts
https://www.khronos.org/advisors/

Kronos members
https://www.khronos.org/members/
https://www.khronos.org/registry/SYCL/

SYCL Working Group
SYCL Advisory Panels