Alessandro Casalino <a.casalino@cineca.it>
Alessandro Masini <a.masini@cineca.it>

July 12, 2023

CINECA

Outline

`C++` Pre-Requisites

- References
- Lambdas

`SYCL`

- Introduction
- `queue` (*where to run*)
- Actions (*what to run*)
- Memory management (*how to move data*)
- Task Graphs (*how to avoid using barriers*)
- Reduction operations (*how to avoid data race*)
- Error Handling

C++ Pre-Requisites: References

Pass by value

```
#include <iostream>

void foo (int i) {
    i++;
    std::cout << i << std::endl;
}

int main() {

    int i {0};

    foo(i);

    std::cout << i << std::endl;
}
```

`C++` (and `C`) functions do not modify the external value of variables passed by value.

Output
1
0

Pass by pointer

```
#include <iostream>

void foo (int* i) {
    (*i)++;
    std::cout << *i << std::endl;
}

int main() {

    int i {0};

    foo(&i);

    std::cout << i << std::endl;
}
```

Passing by pointer, `C++` (and `C`) functions modify the variable value.

Output
1
1

Pass by reference

```cpp
#include <iostream>

// Reference: using &
void foo (int& i) {
    i++;
    std::cout << i << std::endl;
}

int main() {

    int i {0};

    foo(i);

    std::cout << i << std::endl;
}
```

Passing by reference, `C++` functions modify the variable value.

Output

```
1
1
```

Best practice

When the value should be modified by a function, always pass by reference, unless `nullptr` is required.

Note

Since references are not object (they do not occupy memory), they should be preferred to pointers.

C++ Pre-Requisites: Lambdas

Lambda Functions: Example

```cpp
#include <iostream>
#include <algorithm>
#include <array>

bool isEven (int i) {
    return ((i % 2) == 0);
}

int main() {

    std::array<int, 4> a {0, 3, 6, 2};

    auto allEven =
    std::all_of(a.begin(), a.end(), isEven);

    std::cout
    << (allEven ? "All even" : "Not all even")
    << std::endl;
}
```

Output

```
Not all Even
```

- We use `isEven` only once…
- …but we add an additional global function

Can we simplify this?

godbolt.org/z/78xYWGc71

Lambda Functions: Example

```cpp
#include <iostream>
#include <algorithm>
#include <array>

int main() {

    std::array<int, 4> a {0, 3, 6, 2};

    auto isEven = [](int i) {
        return ((i % 2) == 0);
    };

    auto allEven =
    std::all_of(a.begin(), a.end(), isEven);

    std::cout
    << (allEven ? "All even" : "Not all even")
    << std::endl;
}
```

Output

```
Not all Even
```

The **anonymous function** (**lambda**) is in a limited context.

godbolt.org/z/hT6nea4v3

Lambda Functions

```
[ captureClause ] ( parameters ) -> returnType
{
    statements;
}
```

Anatomy of a lambda function:

- `[ captureClause ]`: capture clause
- `( parameters )`: args of the lambda function (as in standard functions)
- `{ statements }`: body of the lambda function (as in standard functions)
- `-> returnType`: explicit return type of the lambda function

Best practice
Prefer lambdas over standard functions when a small routine used in a limited context is needed.

Lambda Functions: Examples

```
#include <iostream>

int main() {

    // Literal void lambda
    [] () {};

    // Void Lambda
    auto a = [] () {};
    a();

    // One input lambda
    auto b = [] (int i) { return i; };
    std::cout << b(5) << std::endl;

    return 0;
}
```

Lambdas can also be used similarly to functions.

godbolt.org/z/rnWxeMd93

Lambda Functions: Example

```
#include <iostream>
#include <algorithm>
#include <array>

int main() {

    std::array<int, 4> a {0, 3, 6, 2};
    int mul {3};

    auto isMul = [](int i) {
        return ((i % mul) == 0);
    }; // This does not compile!

    auto allMul =
    std::all_of(a.begin(), a.end(),
                isMul);

}
```

What if we want a more general lambda?

Warning

This does not compile!

Note

We can **not** use

```
auto isMul = [](int i, int mul) {
    return ((i % mul) == 0);
};
```

In fact, `std::all_of` requires a function of one argument only.
Example: godbolt.org/z/b1nsq6Gse

godbolt.org/z/58s9bbd74

Lambda Functions: Example

```
#include <iostream>
#include <algorithm>
#include <array>

int main() {

    std::array<int, 4> a {0, 3, 6, 2};
    int mul {3};

    auto isMul = [mul](int i) {
        return ((i % mul) == 0);
    }; // This compiles!

    auto allMul =
    std::all_of(a.begin(), a.end(),
                isMul);

}
```

With `[ captureClause ]`, the lambda get access to variables in the surrounding scope.

Note

The variables are captured as `const` by default.

godbolt.org/z/hjTx3cGKa

Lambda Functions: Example

```cpp
#include <iostream>
#include <algorithm>
#include <array>

int main() {

    std::array<int, 4> a {0, 3, 6, 2};
    int mul {3};

    // Capture all by value with =
    auto isMul = [=](int i) {
        return ((i % mul) == 0);
    };

    auto allMul =
    std::all_of(a.begin(), a.end(),
                isMul);

}
```

Or use **default captures** to capture all external context variables

- `=` : by **value**;
- `&` : by **reference**.

Note

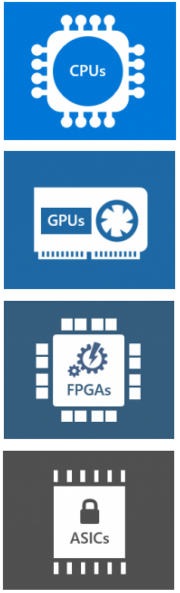
The variables are captured preserving constantness by default.

godbolt.org/z/sehnxK673

Introduction to SYCL

What is `SYCL`?

`SYCL` is an open **standard** for heterogeneous computing, defining an **abstraction layer for programming on accelerators** (CPUs, GPUs, FPGAs).



Features

- `C++` syntax
- **Portability** with accelerator-agnostic APIs and abstractions for
  - compatible **devices finding**,
  - **data movement**,
  - **parallel execution**.
- Ecosystem to **write and tune code on specific devices** or architectures

What is SYCL?

Device Selection And Queries	Parallel Work	Reducing Operations	Buffers	Unified Shared Memory
Choose device and access properties	Distributes work among resources	Combine results from workers	C++-style memory management	C-style memory management
`queue` `info::device`	`single_task` `parallel_for`	`reduction` `combine`	`buffer` `accessor`	`malloc` `memcpy`

Portability

Main `SYCL` feature: **portability**.

- **Functional portability**: correct compilation and running on different device classes/vendors.
- **Performance portability**: performance and scaling similar on different device classes/vendors.

Warning

While functional portability is mostly taken care of by `SYCL`, **it is up to the programmer to ensure performance portability**. This is achieved with specific tuning for different device classes/vendors.

Versions

- SYCL 1.2 released in 2015

  Based on OpenCL 1.2 and C++11

- Provisional SYCL 2.2 introduced in 2016

  Added support for OpenCL 2.2, **never finalized**

- **SYCL 1.2.1** released in 2017 (revision 7 April 2020)

  Introduces support for C++17 and parallel STL algorithms

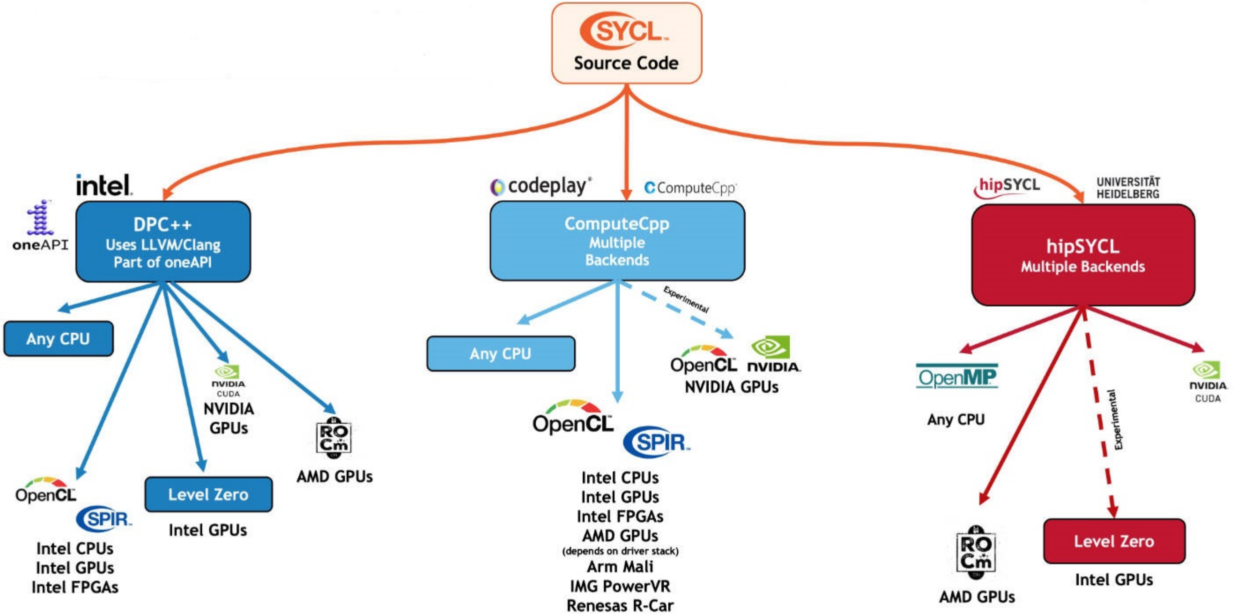
- **SYCL 2020** released in 2021 (revision 7 April 2023)

  Based on C++17, **generalized backend**, much more...

Note
Support for SYCL 2020 is still partially incomplete in major implementations.

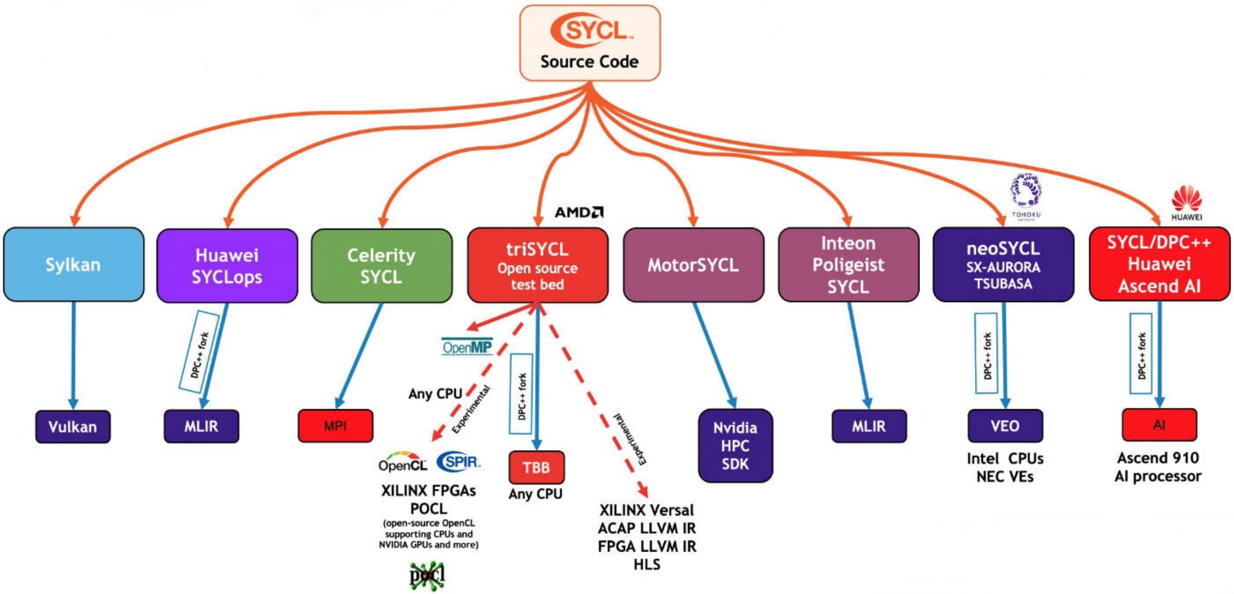
Implementations

Three **more actively developed** implementations.



Implementations

But **many** others are available...



Up to now…

OpenMP (CPU)

```
void daxpy(int n, double alpha, double restrict *x, double restrict *y) {
  #pragma omp parallel for default(none) shared(x, y)
  for (int i = 0; i < n; i++) {
    y[i] += alpha * x[i];
  }
}
```

CUDA (Nvidia GPU)

```
// Kernel
__global__ void daxpy(int n, double alpha, double *x, double *y) {
  for (int idx = blockIdx.x * blockDim.x + threadIdx.x;
       idx < n;
       idx += blockDim.x * gridDim.x) {
    y[idx] += alpha * x[idx];
  }
}
// In the main
daxpy<<<32,256>>>(N, 1.0, device_x, device_y);
```

SYCL "Hello, World!"

```
#include <iostream>
#include <sycl/sycl.hpp>

using namespace sycl;

// ...

int main() {
    queue q;

    char* result =
            malloc_shared<char>(sz, q);
    std::memcpy(result,secret.data(),sz);

    q.parallel_for(sz, [=](auto& i) {
        result[i] -= 1; // Kernel action
    }).wait();

    std::cout << result << std::endl;
    free(result, q);
}
```

Output

```
Hello, world! I'm sorry, Dave.
I'm afraid I can't do that.  -
HAL
```

- Define `queue` (where to run the code)
- Manage memory
- Perform an `action` (express parallelism)

godbolt.org/z/Y7oG3fYqb

Compile with `OpenSYCL`

```
syclcc -opensycl-targets=<target> -O2 my_awesome_code.cpp
```

Target	Version	`opensycl-targets`
CPU	Any	`omp`
Nvidia GPU	Volta V100	`cuda:sm_70`

Note

The Nvidia GPUs compute capability number (e.g., `sm_70`) depends on the GPU version.

Warning

`HipSYCL` has recently been renamed to `OpenSYCL`. Only recent versions support `opensycl-targets`. If the compiler complains, try with the older `hipsycl-targets`.

Compile with `Intel OneApi`

```
icpx -fsycl -fsycl-targets=<target> -O2 my_awesome_code.cpp
```

Target	Version	`opensycl-targets`
CPU	Any	`x86_64`
Nvidia GPU	Volta V100	`nvidia_gpu_sm_70`

Note

The Nvidia GPUs compute capability number (e.g., `sm_70`) depends on the GPU version.

Exercise 0

```
syclcc -opensycl-targets=<target> -O2 hello_world.cpp
```

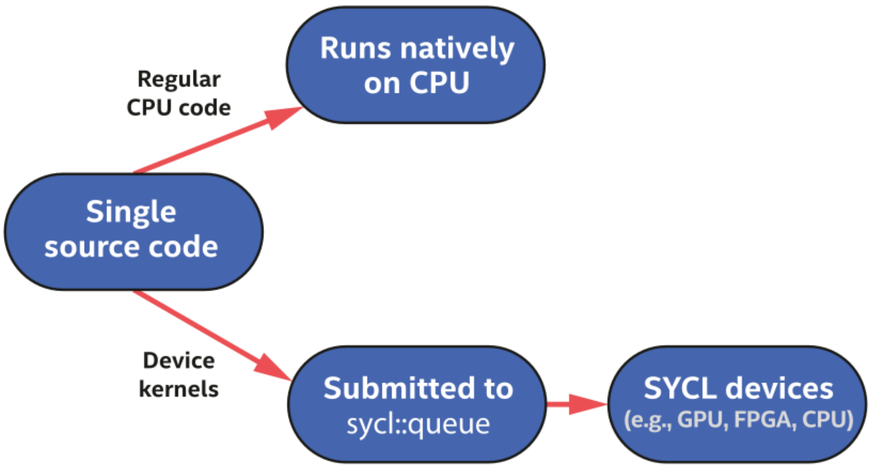
- Compile with the appropriate flags
- Run
- Try to change the target

`queue`

`queue`

```
queue q (const device&);
```

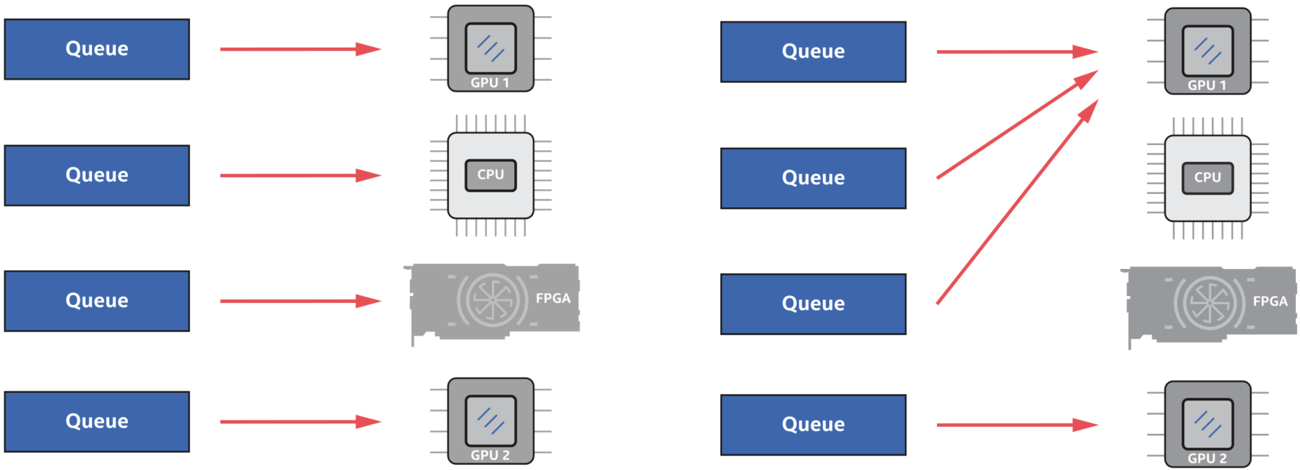
`queue` is an abstraction to which actions are submitted for execution on a **single** device.



queue

```
queue q (const device&);
```

`queue` is an abstraction to which actions are submitted for execution on a **single** device.



queue - Default

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

using namespace sycl;

int main() {
  // Create queue on implementation
  // dependent default device
  queue q;
  // Equivalent to
  // queue q (default_selector_v);

  std::cout << "Selected device: "
  << q.get_device()
        .get_info<info::device::name>()
  << std::endl;

  return 0;
}
```

Possible Output

```
Selected device:  Intel(R)
Xeon(R) Platinum 8259CL CPU @
2.50GHz
```

Best practice

This option might be the best option in the following cases.

- **Developement** on systems without accelerators.
- **Debugging** using non-accelerator tools.
- **Backup** if no other devices are available.

godbolt.org/z/vvhjPvvjf

queue - Accelerator

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

using namespace sycl;

int main() {
  // Create queue on GPU
  queue q (gpu_selector_v);

  std::cout << "Selected device: "
  << q.get_device()
        .get_info<info::device::name>()
  << std::endl;

  return 0;
}
```

Possible Output

```
Selected device:  Tesla
V100S-PCIE-32GB
```

Warning

If no requested device is available, a run-time exception is thrown.

`queue` device

```
queue q (const device&);
```

`device`	Target
None	Implementation-defined (usually host)
`default_selector`	Equivalent to None
`cpu_selector`	CPU
`gpu_selector`	GPU
`accelerator_selector`	Generic accelerators (FPGA, . . .)

Note

`queue` is overloaded on (void) `structs`. Pass to `queue` either `default_selector_v` (preferred) or `default_selector {}`.

queue.get_device()

```
// Create queue on GPU
queue q (gpu_selector_v);
// Get device information
auto device {q.get_device()};

// Device name (std::string)
auto name =
device.get_info<info::device::name>();
// Device vendor (std::string)
auto vendor =
device.get_info<info::device::vendor>();
// Is_type (bool)
// type = cpu, gpu, accelerator
auto is_cpu = device.is_cpu();
// Number of compute units, e.g.,
// streaming multiprocessors for GPUs
auto num_cu = device.get_info
    <info::device::max_compute_units>();
```

From `queue.get_device()` we can extract **device information**.

Possible Output

```
Tesla V100S-PCIE-32GB
NVIDIA
0
80
```

Note

The complete list can be found on the official reference guide.

Device Selection Hierarchy

Compiler flag: `opensycl-targets=omp`

```
#include <iostream>
#include <sycl/sycl.hpp>

using namespace sycl;

int main() {
  queue q (gpu_selector_v);

  std::cout << "Selected device: "
  << q.get_device()
        .get_info<info::device::name>()
  << std::endl;

  return 0;
}
```

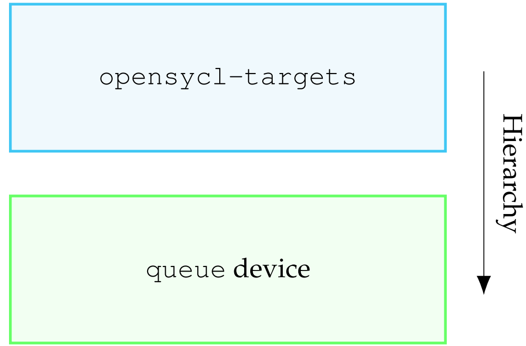
Question

Which device is selected?

Device Selection Hierarchy

Compiler flag: `opensycl-targets=omp`

```
#include <iostream>
#include <sycl/sycl.hpp>

using namespace sycl;

int main() {
  queue q (gpu_selector_v);

  std::cout << "Selected device: "
  << q.get_device()
        .get_info<info::device::name>()
  << std::endl;

  return 0;
}
```



Best practice

For a generic code, leave the `queue` device unspecified. Otherwise check the presence of the requested device to avoid run-time errors.

Exercise 1

Write a program to print the following information.

- **Name** and **vendor** of the device
- Number of **compute units**
- Maximum amount of **memory** available

Question

What happens if you change the `queue` device? And the `opensycl-targets` flag? Try to experiment with the device hierarchy.

Actions

queue.submit()

```
// ...
queue q;

std::array<int, N> data;
buffer B{data};

q.submit(

  // Command group
  [&](handler& h) {
      accessor A{B, h}; // On host

      h.parallel_for(N, [=](id<1>& i){
          A[idx] = idx; // On device
      });
  } // end of command group

);
// ...
```

The `queue.submit()` function allows us to submit work to a device for execution.

The `.submit()` function only takes a C++ lambda as parameter.

Note

This lambda is known as the **command group**, and its parameter is the **command group handler**.

Warning

The command group lambda must capture by reference `[&]`, not by value `[=]`.

godbolt.org/z/KTorsf791

Command Group

```cpp
// ...
[&](handler& h) {

    // Host code for setup
    accessor A{B, h};

    // Action (offloaded to device)
    h.parallel_for(N, [=](id<1>& i){

        // Device code logic
        A[idx] = idx;

    });

  } // end of command group
// ...
```

A command group contains:

- an **action**, i.e. code that executes on device
- host code to set up dependences for device execution

The `handler` allows us to access command group functionality by:

- providing methods to execute actions
- encoding all memory requirements

godbolt.org/z/KTorsf791

Where the Code Runs

```
constexpr int size = 16;
std::array<int, size> data;

// Create queue on default device
queue q;

// Create buffer using host array data
buffer B{data};

q.submit([&](handler& h) {
    accessor A{B, h};
    h.parallel_for(size, [=](auto& idx){
        A[idx] = idx; // On device
    });
});

// Obtain access to buffer on the host
// Wait for device kernel to create data
host_accessor A{B};
```

SYCL programs can be **single-source** (both standard and device code in the same program).

- Standard code on **host** (CPU running the code).
- Submitted actions to **device**.

godbolt.org/z/oc8aPW17o

Where the Code Runs

Host Code

- **Assign work** and **manage data.**
- Assigned by the OS, i.e., **CPU** where program is started.

Device Code

- **Defines actions** on device.
- **Asynchronous from host code**.
- `C++` features are **restricted**: no dynamic allocation, dynamic polymorphism, function pointers or recursion.
- Some **dedicated functions and queries**.

Asynchronicity of Device Code

```
// ...

int main() {
    queue q;

    char* result =
          malloc_shared<char>(sz, q);

    // Introducing data race with q
    // instead of std::memcpy
    q.memcpy(result,secret.data(),sz);

    q.parallel_for(sz, [=](auto& i) {
        result[i] -= 1;
    }); // No wait here = data race!

    std::cout << result << std::endl;
    free(result, q);
}
```

Output

```
Ifmmp-!xpsme"
J(n!tpssz-!Ebwf/!J(n!bgsbje!J!
dbo(u!ep!uibu/!.!IBM staA
```

Warning

This example introduces two sources of possible data race.

- `q.memcpy` is asynchronous, while `std::memcpy` is performed by the host.
- Not waiting means that the host can proceed with `std::cout` without waiting for device to finish.

godbolt.org/z/Esf7rsxaM

Allowed Actions

There are only six possible actions that can be submitted to a queue, and they are separated in two different types

Device code execution

- `single_task`
- `parallel_for`
- `parallel_for_work_group`

Memory Operations

- `copy`
- `fill`
- `update_host`

Warning
Only one action is allowed per command group. Trying to insert more than one action in the same `q.submit()` call is an error, and the program will not compile.

Structure of an Action

```
// ...

h.parallel_for( // pattern

  // work-items distribution
  range{N, N},

  // device kernel
  [=](id<2> idx) { // work-item
      int i = idx[1];
      int j = idx[0];
      c[i*N + j] = a[i*N + j]
                 + b[i*N + j];
  } // end of kernel
);

// ...
```

An action is composed by three elements:

- An execution pattern
- The number of work-items to use and their distribution (in each dimension)
- A lambda with code to be executed by each worker (kernel)

Note

The lambda parameter represents a single work-item. Type depends on work-item distribution.

Warning

The action lambda must capture by value `[=]`, not by reference `[&]`.

godbolt.org/z/hWxW5xzfd

Work-Items Distribution

Work-item distribution is specified using a `range` object

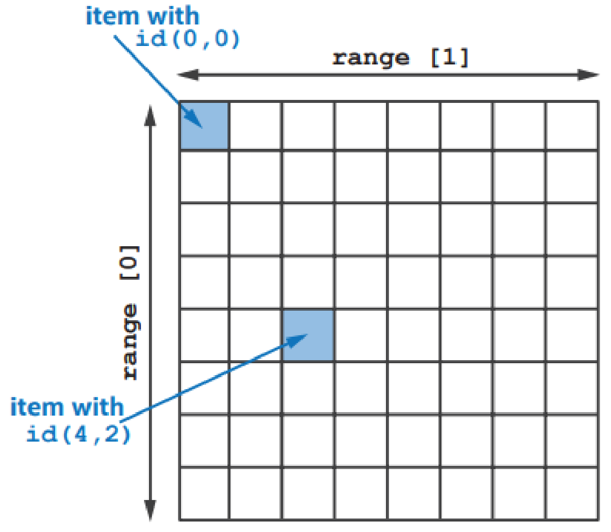
```
template<int dimensions=1>
class range { //... };
```

The single work-items are represented using an `id` object

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

Warning

The dimensions of the `id` in an action's lambda and the corresponding `range` object must match.



`single_task`

```
// ...

q.submit([&](handler& h){
  accessor acc_data {buf_data, h};

  // using only one work-item
  h.single_task([=]() {

    // loop executed by a single WI
    for (int i = 1; i < N; i++) {
        acc_data[0] += acc_data[i];
    }

  }); // end of single_task
}); // end of submit

// ...
```

- **Single instance** of device function
- **Executed by a single work-item**
- No need to specify work distribution

Best practice

Useful for CPU and FPGA programming

Note

When running on CPUs, `single_task` may be able to vectorize code.

godbolt.org/z/8hqbr5GTM

parallel_for

```
// ...

// Using bi-dimensional range
h.parallel_for(range{N, N},
  [=](id<2> idx) {
    // One matrix element per work-item
    int i = idx[1];
    int j = idx[0];
    c[i*N + j] = a[i*N + j]
               + b[i*N + j];
  }
); // end of parallel_for

// ...
```

- Code runs **on a specified work-item distribution**
- **No execution order**
- **No synchronization between work-items**, unless explicitly requested

Best practice

Ideal for embarassingly-parallel problems, and most common action used in GPU programming.

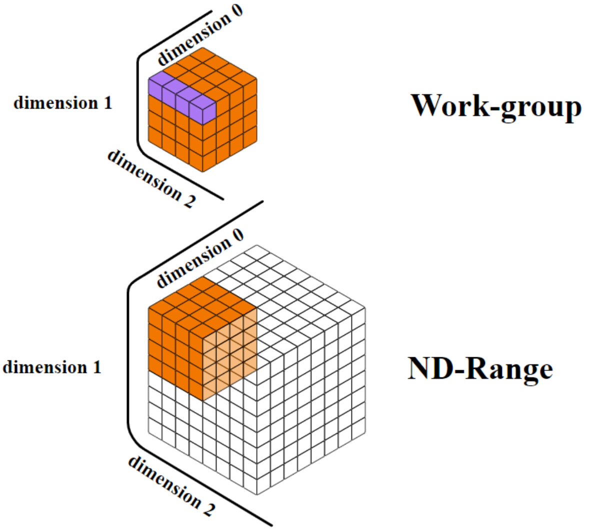
godbolt.org/z/hWxW5xzfd

Work-Items Distribution

Work-Items can be more finely distributed using `nd_ranges`.

An `nd_range` has two ranges:

- the global number of work-items;
- the number of items in a **work-group**.



Note

A **work-group** is a subset of the global work-items distribution which provides extra functionality (such as synchronization and communication routines).

Warning

Using `nd_ranges` the `id` in the action's lambda must be substituted with an `nd_item` with the appropriate number of dimensions.

parallel_for_work_group

```
// ...

h.parallel_for_work_group(
  range {N/B, N/B}, // number of WGs
  range {B, B}, // WIs inside WG

  // execute on each WG
  [=](group<2> grp) {
    int jb = grp.get_group_id(0);
    int ib = grp.get_group_id(1);

    // execute on WIs inside WG
    grp.parallel_for_work_item(
      [&](h_item<2> it) {
        // ...
    });

}); // end of parallel_for_work_group

// ...
```

- Alternative to `parallel_for` with `nd_ranges`
- Requires specification of number of WGs and number of WIs in WG
- Uses `parallel_for_work_item` to distribute work to WIs

Warning

The first lambda has `group` as first argument and captures by `[=]`, the other takes an `h_item` as input and captures by `[&]`.

godbolt.org/z/Y5veEr9vo

Exercises 2 and 3

Modify the matrix multiplication code using `SYCL`.

- Firstly try with `single_task` (godbolt.org/z/qvWcYvb1b)
- Then use `parallel_for` (godbolt.org/z/v3r3GsorM)

Hint

When using `parallel_for`, consider using a multi-dimensional `id`.

Question

Which device is the best for the two actions?

Memory Management

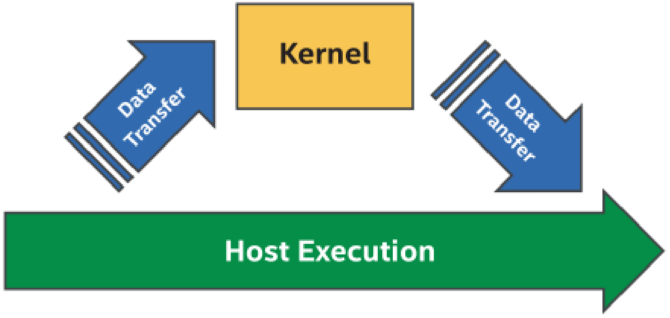
Device and Host Memory

Device usually needs access to data stored on host (or other devices), which is **physically separated** from the device's own memory.

For better performance, we want to **copy data on the device before kernels start**, and **copy it back when the computation has finished**.

`SYCL` Data Movements can be accomplished in two ways:

- **implicitly** by the `SYCL` runtime
- **explicitly** by the programmer



Data Movement

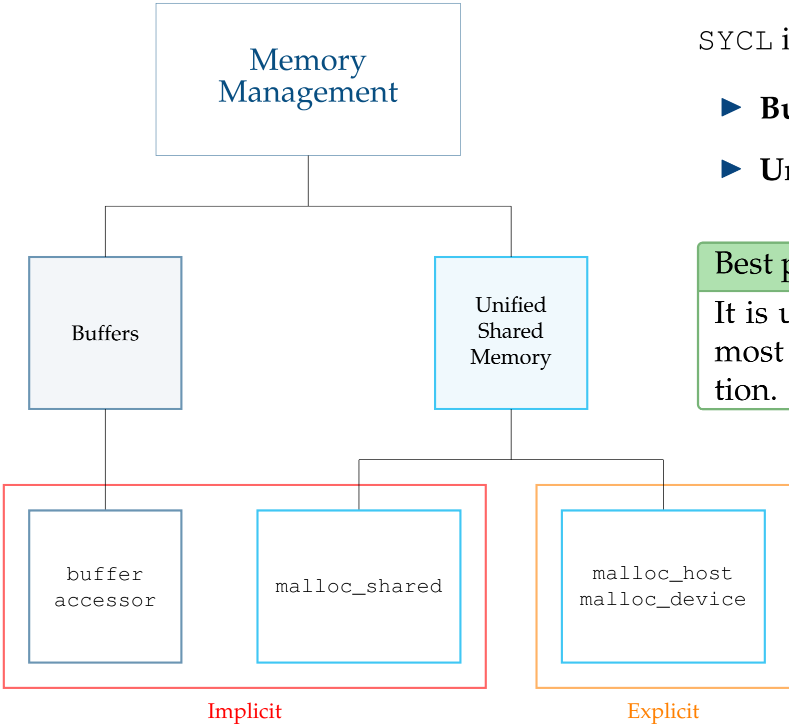
Explicit

- Provides **full control** on where and when data is transferred
- Allows **overlapping movement and computations** for performance
- Error prone
- Time consuming

Implicit

- No control over the behavior of the run-time
- Possible sub-optimal performance
- **Easier to code and debug**
- **Less developer effort**

Memory Management Implementations



`SYCL` implementations

- **Buffers** (and accessors)
- **Unified Shared Memory** (USM)

Best practice

It is up to the programmer to choose the most suitable approach for their application.

Memory Management
Unified Shared Memory

Unified Shared Memory

```
// Allocation
// Array data type within <>
int* host_array =
    malloc_host<int>(N, q);
int* device_array =
    malloc_device<int>(N, q);
// Data migrated on demand
int* shared_array =
    malloc_shared<int>(N, q);

// Copy (both directions)
// Array data type within <>
q.copy<int>(&host_array[0],
    device_array, N);
q.copy<int>(device_array,
    &host_array[0], N);

// Free of memory
free(host_array, q);
free(device_array, q);
free(shared_array, q);
```

- Pointer based approach (like `CUDA`)
- Easy integration with `C/C++` code using pointers
- Can be explicit or implicit

Best practice

Unified Shared Memory best use-cases.

- Porting from already existent `C/C++` codes
- Code with defined computation order (one action after another)

Explicit Data Movement

```
// ...
queue q;
std::array<int, N> host_array;
int* device_array =
    malloc_device<int>(N, q);

for (int i = 0; i < N; i++)
    host_array[i] = N;

q.copy<int>(&host_array[0],
    device_array, N).wait();

q.parallel_for(N, [=](id<1> i) {
    device_array[i]++;
}).wait();

q.copy<int>(device_array,
    &host_array[0], N).wait();

free(device_array, q);
// ...
```

Steps:

- **Allocate device memory** manually
- **Submit memory transfers** to queue
- **Free device memory** at the end

Note

In addition to `C++` API (with templated pointers), a `C` version (`void` pointers) is available.

```
void* malloc_device
        (size_t numBytes, ...);
event memcpy
        (..., size_t numBytes);
```

godbolt.org/z/68Y4d485E

Implicit Data Movement

```
// ...
queue q;
int* shared_array =
    malloc_shared<int>(N, q);

for (int i = 0; i < N; i++)
    shared_array[i] = N;

q.parallel_for(N, [=](id<1> i) {
    shared_array[i] += 1;
}).wait();

free(shared_array, q);
// ...
```

Steps:

- **Perform shared memory allocation**
- **Free shared memory** at the end

Data transfers is handled automatically.

> **Note**
> Memory is not physically shared between host and device, it is just an abstraction

godbolt.org/z/MPhTGPrre

malloc

```
malloc_type<DataType>(size_t, const queue&);
```

type	Access on host	Access on device	Location
device	No	Yes	Device
host	Yes	Yes	Host
shared	Yes	Yes	Migrates on demand

Warning

Not all devices support all USM features.
Check device information from `queue.get_info():` `usm_shared_allocations`, `usm_host_allocations` and `usm_device_allocations`.

Exercise 4

Using the results of the previous Exercise 3, **modify the matrix multiplication code to handle memory with USM**.

- Allocate memory with an explicit data movement
- Then try with an implicit data movement

Hint
Always remember to free the memory.

godbolt.org/z/fx9bo58W9

Memory Management
Buffers

Buffers and Accessors

```
// Allocation from C++ STL objects
std::array<int, N> host_array;
buffer my_buffer {host_array};

// Inside actions
// To access buffer memory
// from device
accessor my_accessor {my_buffer, h,
        read_write};

// Outside actions
// To access buffer memory from host
host_accessor host_accessor
        {my_buffer, read_only};

// No need to free memory
```

- Uses `C++` abstractions
  - Buffers represent abstract data
  - Accessors allow reading/writing
- Only **implicit** data movement

Best practice

Buffers best use-case: we prefer to think about data dependences between kernels.

Warning

Never access a `buffer` directly! Always use accessors.

Buffers and Accessors

```
// ...
queue q;
std::array<int, N> host_array;

for (int i = 0; i < N; ++i)
    host_array[i] = N;

{ // Entering new scope
buffer my_buffer {host_array};

q.submit([&](handler& h) {
    accessor my_accessor {my_buffer, h,
        read_write};

    h.parallel_for(N, [=](id<1> i) {
        my_accessor[i] += 1;
    });
}).wait();
// ...
} // Buffer destroyed, array released
// ...
```

Steps:

- Create a `buffer` **for any data we need on device**
- Create a device `accessor` for the required data

`accessor` access patterns:

- `read_only`
- `write_only`
- `read_write`

Best practice

Avoid touching underlying data (array) when buffer exists.

Host Accessors

```
// ...
// Continued from previous slide
{
    buffer my_buffer {host_array};

    q.submit(...) // Device code

    host_accessor host_accessor
        {my_buffer, read_only};

    std::cout
        << host_accessor[0]
        << " "
        << host_accessor[N-1];
    // ...
} // Buffer destroyed, array released
// ...
```

Access to buffer data on host with `host_accessor`

Same access specifiers as device `accessor`

- `read_only`
- `write_only`
- `read_write`

Warning

While `host_accessor` is guaranteed to have updated memory, the original `host_array` might not be updated until the buffer is destroyed, e.g., out-of-scope.

godbolt.org/z/n5oqhsMfz

Exercise 5

Using the results of the previous Exercise 3, **modify the matrix multiplication code to handle memory with buffers**.

- Create the `buffer` from the data available
- Use appropriate accessors depending on the context

Hint

Remember to use `host_accessor` to show the results.

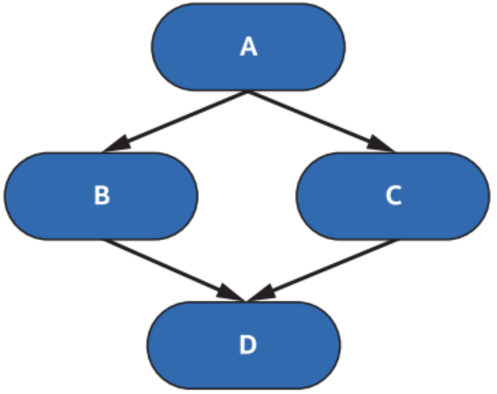
godbolt.org/z/4dMrxq78Y

Task Graphs

Task Graph

Usually **kernels needs to be executed in a specific order** for correct results. Furthermore, **any data accessed by a kernel must be available on the device** before the kernel starts executing.

The dependences can be abstractly represented via a **task graph**.



Tasks (blobs) are either

- kernel executions
- memory transfers (implicit or explicit)

Dependences (arrows) determine the order in which tasks are executed.

Queue Orders

In-order queues

- **Deterministic execution** (tasks execute in order of submission)
- **Forced serialization** even with no dependencies
- Available explicitly **only for USM**

Out-of-order queues

- **Execution order based on task graph**
- **Task graph implicitly generated with buffers/accessors**
- **Task dependencies can be explicitly specified in USM** using events in command groups
- **Allows complex flows** that might result in better performance
- **Default queue type**

Best practice

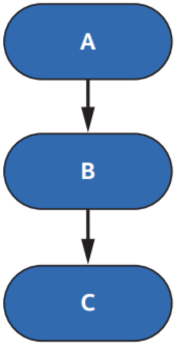
Usually in-order queues are more intuitive and easy to use. In fact, **in-order queues are usually better for prototyping applications**, while the **out-of-order queues allow for more complex codes**.

In-Order Queue

```cpp
// ...
queue q {property::queue::in_order()};

std::array<int, N> host_array;
int* device_array =
    malloc_device<int>(N, q);

// Task A
q.fill<int>(device_array, 1, N);

// Task B
q.parallel_for(N, [=](id<1> i) {
    device_array[i] *= 5;
});

// Task C
q.copy<int>(device_array,
    &host_array[0], N);

q.wait();
// ...
```

Note

No need to use `.wait()` after each task. Execution is serialized, we only need to wait for completion at the end.



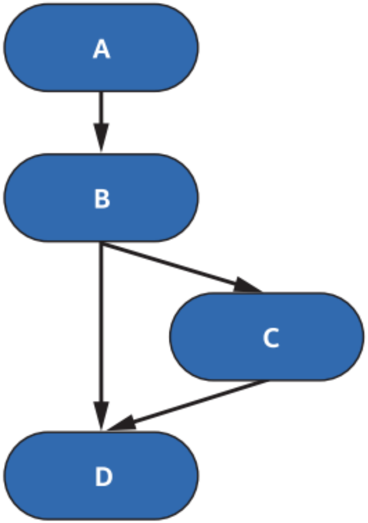
godbolt.org/z/c5Pbjdqxx

Out-of-Order Queue

Out-of-order `queue` is the default `queue`.

Ways to enforce execution order:

- forcing synchronization with `.wait();`
- creating **implicit dependences with accessors** (buffers);
- setting **explicit dependences with events** (USM).



Best practice

Needless to say, avoid `wait` as much as possible!

Warning

Kernels are not guaranteed to be executed in submission order, i.e. the order in the code.

Explicit Dependencies with Events

```
// ...
event m1, m2;
m1 = q.copy<int>(&host_A[0],
      device_A, N);
m2 = q.copy<int>(&host_B[0],
      device_B, N);

event c1 = q.submit([&](handler& h){

  // explicit dependences
  h.depends_on(m1);
  h.depends_on(m2);

  h.parallel_for(N, [=](id<1> i){
    device_D[i] = device_A[i]
                + device_B[i];
  }); // end of parallel_for
}); // end of submit
// ...
```

Steps:

- Create one `event` per each memory transfer or computation
- Declare dependence on an `event` by calling `.depends_on()` on command group handler

Best practice

If possible, avoid calling `.wait()` on events, since it constrains execution and reduces performance.

godbolt.org/z/8rW7cjsve

Implicit Dependencies with Accessors

```
// ...
q.submit([&](handler& h){

  // implicit dependences
  accessor device_A {buffer_A, h,
            read_only};
  accessor device_B {buffer_B, h,
            read_only};
  accessor device_D {buffer_D, h,
            write_only};

  h.parallel_for(N, [=](id<1> i){
    device_D[i] = device_A[i]
                + device_B[i];
  });
});
// ...
```

Steps:

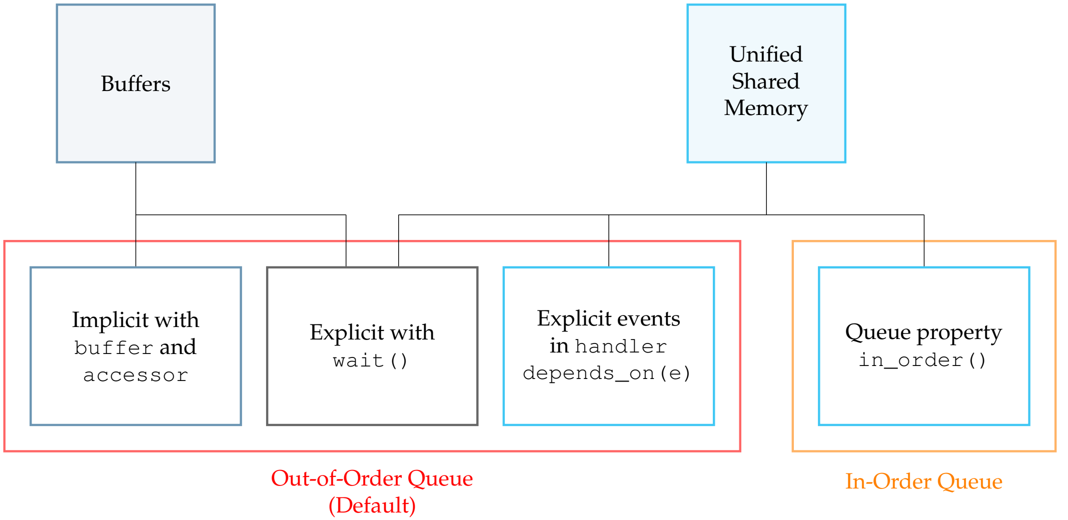
- Create accessors with proper access tags to specify required data and type of dependence
- Runtime generates the task graph based on accessors and program order

Best practice

Implicit strategy is sensible to program order. If possible try to keep program flow as linear as possible.

godbolt.org/z/3YYnK99zv

Queue Order Recap



Best practice

When choosing USM or Buffers, the dependencies handling should be taken into consideration.

Error Handling

Why error handling?

C++

- **Avoid crashes** that might be handled at run-time
- Show more useful error messages

SYCL

- **Host throws exceptions at run-time as standard** C++
- But devices are **asynchronous**: **when/how should I catch the error?**

C++ Error Handling

```
double mySqrt(double x) {
    if (x < 0.0)
      throw "No sqrt of <0 number";
    return std::sqrt(x);
}

int main() {
    double x {-1.};

    try {
        double d = mySqrt(x);
        std::cout << "Result: "
                  << d << std::endl;
    }
    catch (const char* exception) {
        std::cerr << "Error: "
                  << exception
                  << std::endl;
    }
}
```

- `try` block evaluated until the end or until an exception is thrown with `throw`
- If an exception is thrown, `catch` block is evaluated. Catch arg is `throw` arg (can be any type)

Output with `x = 4`

```
Result: 2
```

Output with `x = -1`

```
Error: No sqrt of <0 number
```

godbolt.org/z/E7G1P5hjM

Synchronous Errors

Synchronous errors are thrown by the **host**.

```
int main() {
  buffer<int> b{range{16}};

  // ERROR: Create sub-buffer larger
  // than size of parent buffer.
  // An exception is thrown from
  // within the buffer constructor
  buffer<int> b2(b, id{8}, range{16});

  return 0;
}
```

Output

```
terminate called after
throwing an instance of
'cl::sycl::invalid_object_error'

what(): Requested sub-buffer
size exceeds the size of
the parent buffer -30
(CL_INVALID_VALUE)

Program terminated with
signal: SIGSEGV
```

godbolt.org/z/jjKKxqonj

Synchronous Errors

```
try{
  buffer<int> B{ range{16} };
  buffer<int> B2(B, id{8}, range{16});
}
catch (sycl::exception &e) {
std::cout
  << "Caught sync SYCL exception: "
  << e.what() << std::endl;
  return 1;
}
catch (std::exception &e) {
std::cout
  << "Caught std exception: "
  << e.what() << std::endl;
  return 2;
}
catch (...) { // Any other data-type
  std::cout
    << "Caught unknown exception\n";
 return 3;
}
```

Synchronous errors can be handles with standard `C++ catch`.

- Exception type is `sycl::exception`
- `sycl::exception` has a member `what()` to extract a string with info on the exception thrown.

Output

```
Caught sync SYCL exception:
Requested sub-buffer size
exceeds the size of the parent
buffer -30 (CL_INVALID_VALUE)
```

godbolt.org/z/rxPazcE3P

Asynchronous Errors

```
int main() {
  queue q{gpu_selector_v};

  q.submit(
    [&](handler& h) {
      // Empty command group
      // is illegal and generates an
      // error
    });

  return 0;
}
```

Asynchronous errors are thrown by the **device**.

These can not be catched at run-time: they happen asynchronously to host (which deals with `throw/catch`).

Asynchronous Errors

We specify an **handler function**.
This is called at certain events (e.g., end of `submit`).

```
// Asynchronous handler function
auto handle_async_error =
[](exception_list elist) {
  for (auto& e : elist) {
    try {
      std::rethrow_exception(e);
    } catch (sycl::exception& e) {
      std::cout
            << "ASYNC EXCEPTION!!\n";
      std::cout << e.what() << "\n";
    }
  }
};
```

```
int main() {
  queue q{gpu_selector_v,
          handle_async_error};
  say_device(q1);

  try {
    q.submit(
      [&](handler& h) {
        // Empty command group
        // is illegal and
        // generates an error
      }).wait_and_throw();
  } catch (...) {
  }  // Discard regular C++ exceptions
  return 0;
}
```

Reducing Operations

Data Race

Compiler flag:
`opensycl-targets=cuda:sm_70`

```
queue q;
int sum {0};

buffer<int> sumBuf {&sum, 1};

q.submit([&](handler& h) {
  accessor sumAcc
           {sumBuf, h, read_write};

  h.parallel_for(range(N), [=](id<1> i)
  {
    sumAcc[0]+=i;
  });
}).wait();

host_accessor hAcc {sumBuf, read_only};
assert(hAcc[0] == 523776);
```

Output

```
data_race:  data_race.cpp:31:
int main():  Assertion
'host_accessor[0] == 523776'
failed.
```

Question

What happens if you run this on CPU (`cpu_selector` or `opensycl-targets=omp`)?

`reduction`

`reduction` **gathers data from workers** preventing data races.

```
__unspecified__ reduction (buffer, handler&, BinaryOperation);
```

Binary operations

- `plus`
- `multiplies`
- `bit_and`
- `bit_or`
- `bit_xor`
- `maximum`
- `minimum`
- `logical_and`
- `logical_or`

Note

The binary operations should be accessed as *templated* structures, e.g., the `BinaryOperation` for addition becomes `plus<int>()`.

reduction

Compiler flag:
`opensycl-targets=cuda:sm_70`

```
queue q;
int sum {0};

buffer<int> sumBuf {&sum, 1};

q.submit([&](handler& h) {
  auto sumRed
      {reduction(sumBuf, h, plus<>())};

  h.parallel_for(range(N), sumRed,
    [=](id<1> i, auto& sum) {
      sum+=i;
    });
}).wait();

host_accessor hAcc {sumBuf, read_only};
assert(hAcc[0] == 523776);
```

`reduction` **gathers data from workers** preventing data races.

Warning

`OpenSYCL` does not support this standard way of performing reductions yet (on version 0.9.4). We might need to substitute

```
auto sumRed
 {reduction(sumBuf, h, plus<>()};
```

with

```
accessor sumVal
 {sumBuf, h, read_write};
auto sumRed
 {reduction(sumVal, plus<int>())};
```

Exercise 6

Write a program to perform a `float` maximum operation on a vector.

- Initialize elements of an array with the `float` version of their index.
- Allocate a `float` buffer.
- First try without reduction.
- Perform the reduction.
- Test the code without and with reduction on both CPU and GPU.

Question

Do you get the same behaviour on CPU and GPU?

Take-Home Messages

Take-Home Messages

- `SYCL` is a standard for heterogeneous device parallel programming in pure `C++`
- The targets can be CPUS, GPUS and other accelerators (FPGAs, vectorization accelerators, etc..)
- Parallelization actions can be used to parallize the code, similarly to `CUDA` (SPMD)
- The memory can be implicitly handled with buffers
- Or with a more porting-friendly approach called USM, both for implicit and explicit memory handling

Take-Home Messages

Best practice

Prefer `SYCL` over vendor-specific implementations (`CUDA`, `AMD`, etc...) if the application should run on different vendors devices.

Warning

Be aware of the overheads of `SYCL`. Always time (or better profile) your code.

Resources

- **Essential**: Official `SYCL` Reference Guides
- **Essential**: `SYCL` Text-Book (a bit outdated in some parts)
- `SYCL` Academy Lectures
- Last `SYCL 2020` Specifications

Thank you!