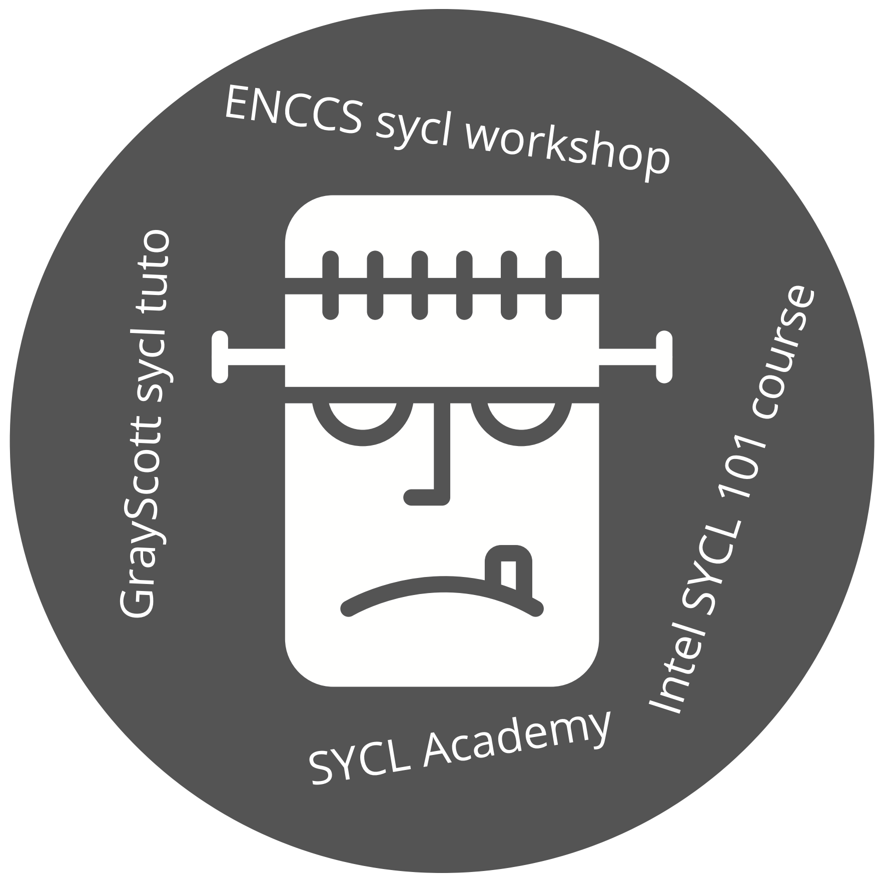
HPC TRAINING EVENTS

tutorial – A brief intro to oneAPI & SYCL

DR OSCAR FABIAN MOJICA LADINO
oscar.ladino@fieb.org.br

This course material can be likened to a Frankenstein creation, assembled from various sources, but I have strictly adhered to the terms specified in the respective licenses to guarantee complete compliance. The material is derived from Intel's SYCL 101 course, where the code is licensed under the MIT License, Codeplay's SYCL Academy repository, which provides an excellent introductory tutorial on SYCL's basic features, with code licensed under the CC-BY-4.0 License, and the SYCL Workshop by EuroCC National Competence Center Sweden (ENCCS), featuring code licensed under the MIT License and additional material under the CC-BY-4.0 License. Additionally, the codes used for checking the SYCL environment setup were sourced from the Gray Scott sycl tutorial and are licensed under the GPL-3.0 License.



See also:

- free ebook on SYCL programming
- DPC++ examples from intel
- sycl.tech playground
- sycl academy

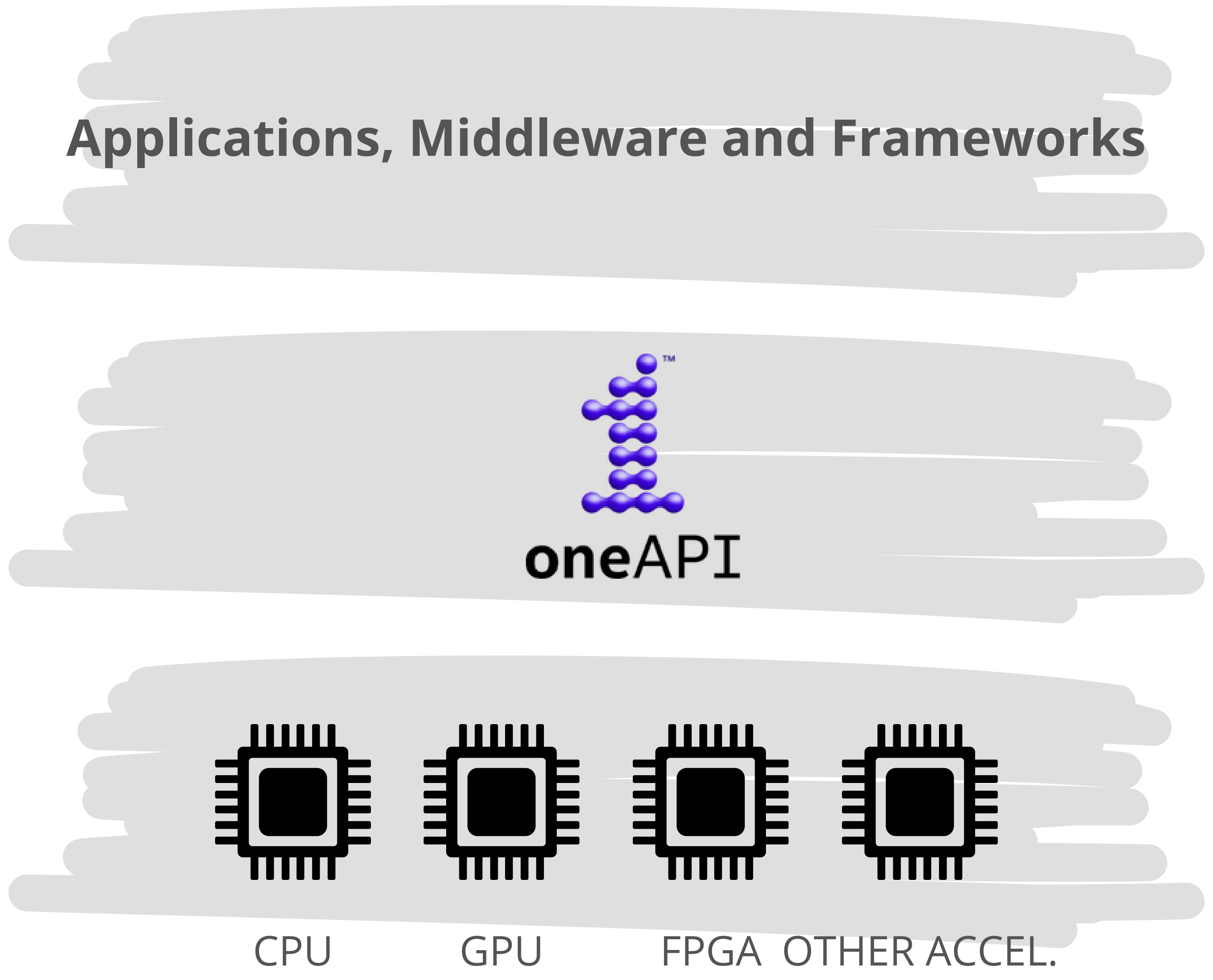
What is oneAPI?

oneAPI is an open source project with an open and standards-based set of well-defined interfaces, supporting multiple architecture types including but not limited to GPU, CPU, and FPGA

oneAPI, the Open Specification

Any organization can create their own implementation of the oneAPI open specification. Intel® leads the development of the largely open source oneAPI Base Toolkit

A freely available implementation of the standard is available through Intel® oneAPI Toolkits. The Intel® Base Toolkit features an industry-leading C++ compiler that implements SYCL, an evolution of C++ for heterogeneous computing



What is SYCL?

C++ Programming for Heterogeneous Parallel Computing

SYCL is an open, royalty-free, cross-platform abstraction layer that enables code for heterogeneous and offload processors to be written using modern ISO C++, and provides APIs and abstractions to find devices (CPUs, GPUs, FPGAs ...) on which code can be executed, and to manage data resources and code execution on those devices.

It is built as a header-only library for ISO C++17. SYCL code can be compiled with a standards-compliant compiler and the necessary headers: it does not require special compiler extension

It is a single-source-style framework. Host and device code are in the same translation unit.

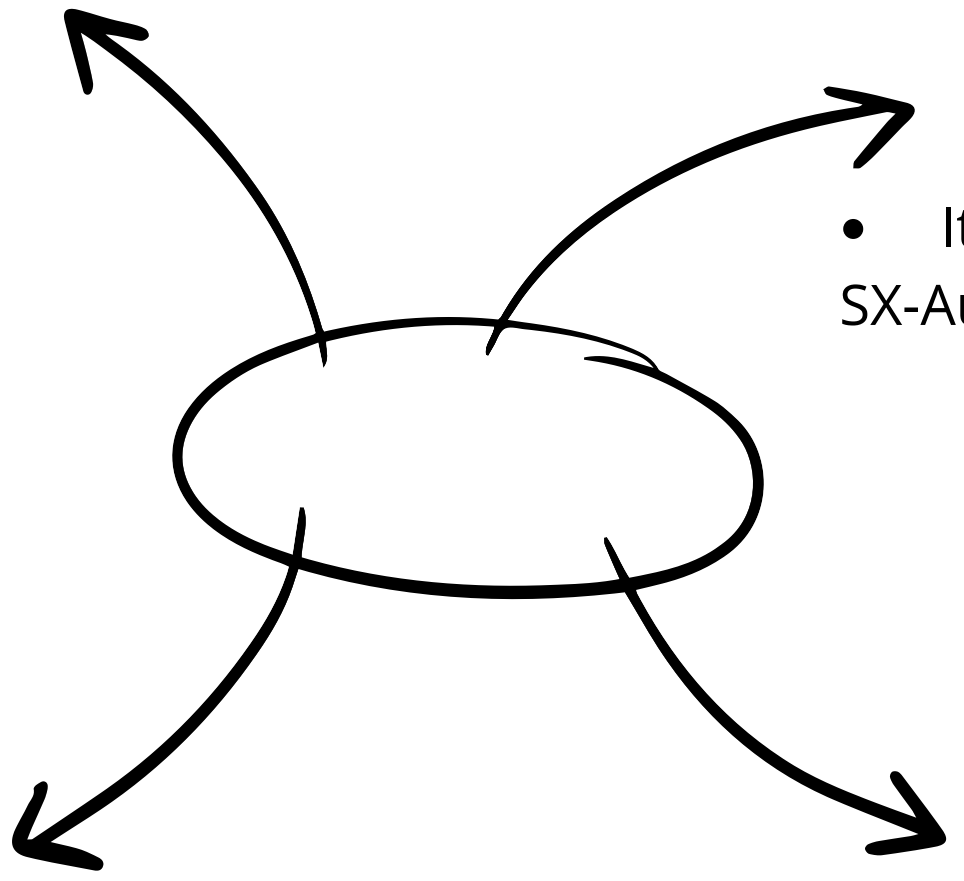
Nice interoperability with other programming model (OpenMP, CUDA, Hip, OpenCL)

The SYCL specifications that have been published as of today are:

- SYCL 1.2: Specification released in May of 2015: It is considered obsolete.
- SYCL 2.2: It was a provisional specification published in 2016 that is considered deprecated.
- SYCL 1.2.1: Its latest revision was released on April 27, 2020.
- SYCL 2020: Specification revision 7 released on April 18, 2023.

SYCL implementation

SYCL implementations are available from an increasing number of vendors, including adding support for diverse acceleration API back-ends in addition to OpenCL.



triSYCL

- It is no longer maintained

neoSYCL

- It focuses exclusively on the SX-Aurora TSUBASA architecture

Intel® oneAPI DPC++ Library (oneDPL)

- implements the Khronos SYCL 2020 standard
- It is a companion to the oneAPI DPC++/C++ Compiler
- The oneAPI DPC++ compiler is based on the LLVM compiler, which speeds up compilation times

AdaptiveCpp

- (formerly known as hipSYCL /OpenSYCL)
- Roots at the University of Heidelberg.
- It supports CPUs (including x86, arm and other LLVM-supported architectures) as well as GPUs from Intel, NVIDIA, and AMD.

Advantages

1. SYCL provides an easier programming model for heterogeneous applications than traditional models such as OpenCL or CUDA.

2. SYCL supports modern C++ language features and can help simplify writing portable and maintainable code.

3. SYCL allows developers to take advantage of heterogeneous hardware architectures and utilize multiple processors or accelerators simultaneously.

4. SYCL provides an abstraction layer that makes it easier to port code to different hardware architectures.

5. SYCL enables the development of high-performance and data-parallel applications.

6. SYCL allows third-party vendors to provide tools that help optimize code for different hardware architectures.

Basic SYCL Code #1: single_task

```
#include <sycl/sycl.hpp>
using namespace sycl;

int main() {
   queue Q;   // The queue, Q, is the object that submits the task to a device.
   int const size = 10;
   buffer<int> A{ size };          // The buffer, A, is the memory used to
                                    // transfer data between host and device.

   Q.submit([&](handler& h) {      // The handler, h, is the object that contains
                                    // the single_task function to be used.

      accessor A_acc(A, h);        // The accessor, A_acc, is the object that
                                    // efficiently accesses the buffer elements.

      h.single_task([=]() {
         A_acc[5] = 77;
         });

      });

   host_accessor result(A);        // host_accessor is the object that allows
                                    // the host to access the buffer memory.

   for (int i = 0; i < size; i++)  // Print output
      std::cout << result[i] << " "; std::cout << "\n";
   return 0;
}
```

Basic SYCL Code #1: single_task

The construct **single_task** is used to define a unit of work that should be executed on a single processing element, typically a single CPU core or GPU thread. The goal in this code is to modify specifically the 6th element of a 10-size vector using the SYCL single_task command

The **queue** serves as the central construct used to submit work items, control execution flow, and facilitate data transfers in a parallel and heterogeneous computing environment, such as CPUs or GPUs.

The **buffer** serves as a data container that defines the region of memory accessible by both the host and the device, enabling efficient data sharing and transfer between them.

The **command group** represents a unit of work that can be submitted to a SYCL queue for execution; its main function is to define the operations or computations that are to be performed on the target device

- The **accessor**, A_acc, is the object used to define the access rights (read-only, write-only, or read-write) of specific kernels to the buffer elements.
- Inside of the command group resides the specific SYCL kernel function, which is **single_task** in this case. The handler is the object that represents a context in which command groups are defined. It specifies the operations and dependencies within a command group and controls the execution behavior of those operations. One consideration to keep in mind is that only one SYCL kernel function, even if it is the same, can be executed in the command group

```
#include <sycl/sycl.hpp>
using namespace sycl;

int main() {
   queue Q;   // The queue, Q, is the object that submits the task to a device.
   int const size = 10;
   buffer<int> A{ size };          // The buffer, A, is the memory used to
                                   // transfer data between host and device.

   Q.submit([&](handler& h) {      // The handler, h, is the object that contains
                                   // the single_task function to be used.

      accessor A_acc(A, h);        // The accessor, A_acc, is the object that
                                   // efficiently accesses the buffer elements.

      h.single_task([=]() {
         A_acc[5] = 77;
         });

      });

   host_accessor result(A);        // host_accessor is the object that allows
                                   // the host to access the buffer memory.

   for (int i = 0; i < size; i++)  // Print output
      std::cout << result[i] << " "; std::cout << "\n";
   return 0;
}
```

SYCL kernel function:
Note that it is provided by the handler, h

parallel_for

Essentially **parallel_for** distributes work across multiple processing elements for parallel execution and allows developers to express parallelism easily . The equivalent of **parallel_for** in standard C++ would be std::for_each. With **parallel_for**, programmers only have to define a function or a lambda expression that represents the work to be done in parallel while maintaining code portability.

- h.parallel_for: This line initiates a parallel computation using the parallel_for construct. It means that the enclosed code block will be executed in parallel across multiple processing elements (e.g., CPU cores or GPU threads). The h is the handler, which manages the execution of SYCL tasks on a specific device.

- range<1>(size): This part specifies the range of work items for the parallel computation. In this case, it's a 1D range defined by range<1> with a size of size=10. The range defines how many times the enclosed code block will execute in parallel meaning that in this case the code block will be executed 10 times concurrently, each time with a different value of indx ranging from 0 to 9.

- [=](id<1> indx) { A_acc[indx] = 77; }: This is a lambda function that represents the work to be performed in parallel. The lambda function takes an id<1> argument named indx, which represents the unique identifier of the current work item. Inside the lambda function, the code sets the value at the indx-th position of the array A_acc to 77. In essence, each parallel instance will update a different element of the A_acc array to the value 77.

Basic SYCL Code #2: parallel_for

```
#include <CL/sycl.hpp>
using namespace sycl;

int main() {

    queue Q;                        // The queue, Q, is the object that
                                    // submits the task to a device.
    int const size = 10;
    buffer<int> A{ size };          // The buffer, A, is the memory used to
                                    // transfer data between host and device.

    Q.submit([&](handler& h) {      // The handler, h, is the object that contains
                                    // the parallel_for function to be used.

        accessor A_acc(A, h);       // The accessor, A_acc, is the object that
                                    // efficiently accesses the buffer elements.

        h.parallel_for(range<1>(size), [=](id<1> indx) {
            A_acc[indx] = 77;
            });

        });

    host_accessor result(A);        // host_accessor is the object that allows
                                    // the host to access the buffer memory.

    for (int i = 0; i < size; i++)  // Print output
        std::cout << result[i] << " "; std::cout << "\n";
    return 0;
}
```

command group

SYCL kernel function

lambda functions

- Lambda functions are introduced in C++11
- Used to create anonymous function objects, can also be used with named variables
- Lambda expression has the following syntax:

```
[ capture-list ] ( params ) -> ret { body }
```

params:

- List of function parameters similar to named function
- In SYCL, it can be unique 1D id, or 2D/3D id

ret:

- Defines the return type of the lambda expression
- If -> ret is not specified, it is inferred from the return statement in the body
- Return with no value implies void return type
- SYCL kernels must always have a void return type thus it is not specified in SYCL kernel

body:

- Contains function body
- SYCL kernel body does not have any return statement

capture-list:

- Lambda expression starts with a square bracket
- Denotes how to capture variables that are used within the lambda but not passed as parameters
- Comma separated list of captures
- Variables from the surrounding scope mentioned in this list are available in the lambda body
- Controls the visibility and lifetime of variables captured by lambda
- Global variables are not captured in lambda expression
- Non-global static variables can be used in a kernel but only if they are const
- Capture a variable by value by listing the variable name in the capture-list. Even the value is modified in the body, it does not affect the original value outside the lambda.
- Capture a variable by reference by listing the variable name prefixing with ampersand (&variable). If this variable is modified in the body, the original variable is also modified.
- [=] shorthand to capture all variables used in the body by value and current object by reference
- [&] shorthand to capture all variables used in the body by reference and current object by reference
- [] captures nothing
- SYCL uses [=] as kernel does not support capturing variable by reference

queue

A SYCL queue manages the execution of command groups on a specific device. It acts as a command queue, allowing you to submit command groups for execution and control the order of execution

At this point, it is important to stress that a queue can be mapped to one device only. The mapping happens at queue construction and cannot be changed afterwards.

queue constructor

```
template <typename DeviceSelector>
explicit queue(const DeviceSelector &deviceSelector,
                    const property_list &propList = {});
```

The selector passed as first parameter lets us specify how the runtime should go about mapping the queue to a device.

- *Somewhere*

```
queue Q;
```

- *On the host device*

A standards-compliant SYCL implementation will always define a host device and we can bind a queue to it by passing the host_selector object to its constructor:

```
queue Q{host_selector{}};
```

The host device makes the host CPU "look like" an independent device, such that device code will run regardless of whether the hardware is available.

- On a specific class of devices

Such as GPUs or FPGAs. The SYCL standard defines a few selectors for this use case.

```
queue Q_cpu{default_selector{}};
queue Q_cpu{cpu_selector{}};
queue Q_device{gpu_selector{}};
queue Q_accelerator{accelerator_selector{}};
```

- On a specific device in a specific class

For example on a GPU with well-defined compute capabilities. SYCL defines the device_selector base class, which we can inherit from and customize to our needs.

```
class special_device_selector : public device_selector {
   /* we will look at what goes here soon! */
};
queue Q{special_device_selector{}};
```

Data management

- ***unified shared memory (USM).*** This is a pointer-based approach, familiar to C/C++ programmers and similar to CUDA/HIP low-level languages for accelerators. USM pointers on the host are valid pointers also on the device. This is at variance with "classic" pointers in CUDA/HIP. USM needs device support for a unified virtual address space.

- ***the buffer and accessor API***. A buffer is a handle to a 1-, 2-, or 3-dimensional memory location. It specifies where the memory location and where it can be accessed: host, device or both. As such, the buffer does not own the memory: it's only a constrained view into it. We don't work on buffer directly, but rather use accessors into them. This is analogous to a RAII-like approach, similar to what the STL does in C++.

- ***images.*** They offer a similar API to buffer types, with extra functionality tailored for image processing.

Buffers are constructed by specifying their size and what memory they should provide a view for. The buffer class is templated over the type of the underlying memory and its dimensionality (1, 2, or 3). We give the size as an object of range type

Some **buffer** constructors

```
buffer(const range<dimensions> &bufferRange,
           const property_list &propList={});

buffer(T *hostData,
       const range<dimensions> &bufferRange,
       const property_list &propList={});
```

Each constructor takes as the last parameter an optional sycl::property_list to provide properties to the sycl::buffer. See other ways to create buffers in https://github.khronos.org/SYCL_Reference/iface/buffer.html

Creation of buffers is just one side of the coin. The buffer is only a view into memory and no migration of data occurs when we construct one and we cannot manipulate the underlying data of a buffer directly: both goals are achieved with accessors.

Data management

Accessors: Working with Buffers

Accessors provide a way to access and manipulate the data within Buffers. They come with different access modes that dictate how the data can be accessed and modified.

Accessor objects are templated over five parameters:

- the type and the dimension, which will be the same as for the underlying buffer.
- the access mode: how do we intend to access the data in the buffer? The possible values are read, write, and read_write for read-only (default for const data types), for write-only, and for read-write (default for non-const data types) access, respectively.
- the access target: what memory and where do we intend to access? The default is global_memory stating that the data resides in the device global memory space.
- the placeholder status: is this accessor a placeholder or not?

Device accessors can be created within a command group, for example:

```
buffer<double> A{range{42}};

Q.submit([&](handler &cgh){
  accessor aA{A, cgh};
});
```

Tag value	Access mode	Access target
read_write	read_write	default
read_only	read	default
write_only	write	default

objects of type host_accessor are used to read data on the host from a buffer previously accessed on a device:

```
buffer<double> A{range{42}};

Q.submit([&](handler &cgh){
  accessor aA{A, cgh};

  // fill buffer
  cgh.parallel_for(range{42}, [=](id<1> & idx){
    aA[idx] = 42.0;
  })
});

host_accessor result{A};
for (int i = 0; i < N; i++) {
  assert(result[i] == N);
}
```

These objects are similar to device accessors, but you will note that they are constructed with just a buffer as argument. Further, we inspect the contents of the buffer directly, even though we didn't put buffer and queue submission in a separate scope, nor did we wait on the queue. The constructor for the host_accessor implicitly waits for the data to be available.

```
#include <CL/sycl.hpp>
#include <stdio.h>

constexpr int N = 100;
int main() {

  int AData[N];
  int BData[N];
  int CData[N];

  sycl::queue Q;

  // Kernel1
  {
    // Create 3 buffers, each holding N integers
    sycl::buffer<int> ABuf(&AData[0], N);
    sycl::buffer<int> BBuf(&BData[0], N);
    sycl::buffer<int> CBuf(&CData[0], N);

    Q.submit([&](auto &h) {
      // Create device accessors.
      // The property no_init lets the runtime know that the
      // previous contents of the buffer can be discarded.
      sycl::accessor aA(ABuf, h, sycl::write_only, sycl::no_init);
      sycl::accessor aB(BBuf, h, sycl::write_only, sycl::no_init);
      sycl::accessor aC(CBuf, h, sycl::write_only, sycl::no_init);

      h.parallel_for(N, [=](auto i) {
        aA[i] = 11;
        aB[i] = 22;
        aC[i] = 0;
      });
    });
  } // end Kernel1

  // Kernel2
  {
    // Create 3 buffers, each holding N integers
    sycl::buffer<int> ABuf(&AData[0], N);
    sycl::buffer<int> BBuf(&BData[0], N);
    sycl::buffer<int> CBuf(&CData[0], N);

    Q.submit([&](auto &h) {
      // Create device accessors
      sycl::accessor aA(ABuf, h, sycl::read_only);
      sycl::accessor aB(BBuf, h, sycl::read_only);
      sycl::accessor aC(CBuf, h);
      h.parallel_for(N, [=](auto i) { aC[i] += aA[i] + aB[i]; });
    });
  } // end Kernel2

  // Buffers are destroyed and so CData is updated and can be accessed
  for (int i = 0; i < N; i++) {
    printf("%d\n", CData[i]);
  }

  return 0;
}
```

initializes the A, B, and C buffers, so we specify that the access modes for these buffers is **write_only.**

The second kernel reads the A and B buffers, and reads and writes the C buffer, so we specify that the access mode for the A and B buffers is **read_only**, and the access mode for the C buffer is **read_write.**

The accessor's access modes are used by the runtime to create an execution order for the kernels and perform data movement. This will ensure that kernels are executed in an order intended by the programmer. Depending on the capabilities of the underlying hardware, the runtime can execute kernels concurrently if the dependencies do not give rise to dependency violations or race conditions.

If this second kernel were to use read_write for A and B instead of read_only, then the memory associated with A and B is copied from the device to the host at the end of kernel execution, even though the data has not been modified by the device

!

A better way to write the code that avoids these unnecessary memory transfers is shown below.

```
#include <CL/sycl.hpp>
#include <stdio.h>

constexpr int N = 100;

int main() {

  int AData[N];
  int BData[N];
  int CData[N];

  sycl::queue Q;

  // Create 3 buffers, each holding N integers
  sycl::buffer<int> ABuf(&AData[0], N);
  sycl::buffer<int> BBuf(&BData[0], N);
  sycl::buffer<int> CBuf(&CData[0], N);

  // Kernel1
  Q.submit([&](auto &h) {
    // Create device accessors.
    // The property no_init lets the runtime know that the
    // previous contents of the buffer can be discarded.
    sycl::accessor aA(ABuf, h, sycl::write_only, sycl::no_init);
    sycl::accessor aB(BBuf, h, sycl::write_only, sycl::no_init);
    sycl::accessor aC(CBuf, h, sycl::write_only, sycl::no_init);

    h.parallel_for(N, [=](auto i) {
      aA[i] = 11;
      aB[i] = 22;
      aC[i] = 0;
    });
  });

  // Kernel2
  Q.submit([&](auto &h) {
    // Create device sycl::accessors
    sycl::accessor aA(ABuf, h, sycl::read_only);
    sycl::accessor aB(BBuf, h, sycl::read_only);
    sycl::accessor aC(CBuf, h);
    h.parallel_for(N, [=](auto i) { aC[i] += aA[i] + aB[i]; });
  });

  // The host accessor creation will ensure that a wait for kernel to finish
  // is triggered and data from device to host is copied
  sycl::host_accessor h_acc(CBuf);
  for (int i = 0; i < N; i++) {
    printf("%d\n", h_acc[i]);
  }

  return 0;
}
```

another way to run the same code with different scope blocking. In this case, there will not be a copy of buffers from host to device at the end of kernel1 and from host to device at the beginning of kernel2. The copy of all three buffers happens at the end of kernel2 when these buffers go out of scope.

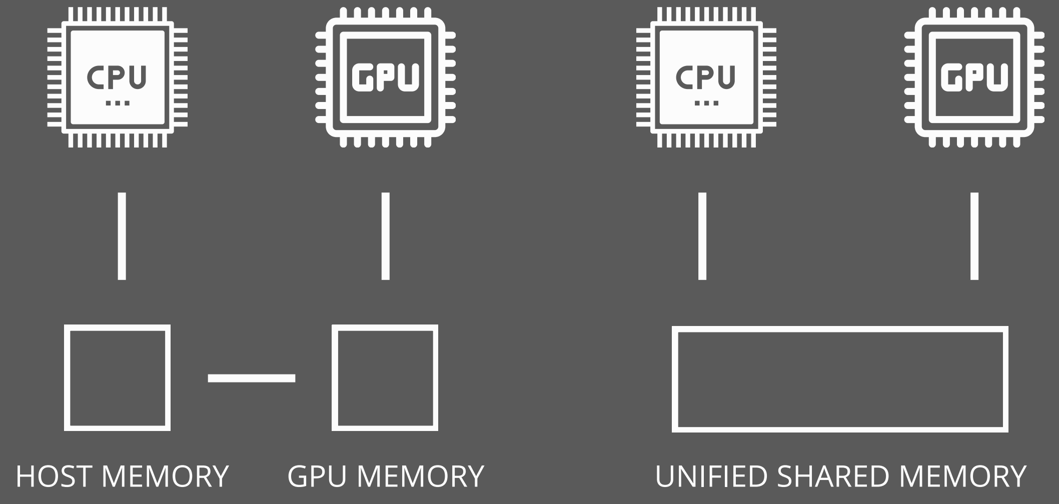
```
#include <CL/sycl.hpp>
#include <stdio.h>

constexpr int N = 100;

int main() {

  int AData[N];
  int BData[N];
  int CData[N];

  sycl::queue Q;

  {
    // Create 3 buffers, each holding N integers
    sycl::buffer<int> ABuf(&AData[0], N);
    sycl::buffer<int> BBuf(&BData[0], N);
    sycl::buffer<int> CBuf(&CData[0], N);

    // Kernel1
    Q.submit([&](auto &h) {
      // Create device accessors.
      // The property no_init lets the runtime know that the
      // previous contents of the buffer can be discarded.
      sycl::accessor aA(ABuf, h, sycl::write_only, sycl::no_init);
      sycl::accessor aB(BBuf, h, sycl::write_only, sycl::no_init);
      sycl::accessor aC(CBuf, h, sycl::write_only, sycl::no_init);

      h.parallel_for(N, [=](auto i) {
        aA[i] = 11;
        aB[i] = 22;
        aC[i] = 0;
      });
    });

    // Kernel2
    Q.submit([&](auto &h) {
      // Create device accessors
      sycl::accessor aA(ABuf, h, sycl::read_only);
      sycl::accessor aB(BBuf, h, sycl::read_only);
      sycl::accessor aC(CBuf, h);
      h.parallel_for(N, [=](auto i) { aC[i] += aA[i] + aB[i]; });
    });
  }
  // Since the buffers are going out of scope, they will have to be
  // copied back from device to host and this will require a wait for
  // all the kernels to finish and so no explicit wait is needed
  for (int i = 0; i < N; i++) {
    printf("%d\n", CData[i]);
  }

  return 0;
}
```

unified shared memory

There are three types of USM allocations available in the SYCL standard

- Device allocations will return a pointer to memory physically located on the device.
- Host allocations will return a pointer to memory physically located on the host. These are accessible both on the host and the device. In the latter case, however, memory will not migrate to the device automatically, but rather be accessed remotely. This is a crucial aspect to keep in mind for performance!
- Shared allocations will return a pointer to the unified virtual address space. Such allocations can be accessed from both host and device, and the memory can migrate freely, without programmer intervention, between host and device. This comes at the cost of increasing latency.

Kind	Host-accessible	Device-accessible	Memory space	Automatic migration
device	No	Yes	Device	No
host	Yes	Yes	Host	No
shared	Yes	Yes	Shared	Yes



USM is probably the biggest new feature adopted in the SYCL 2020 standard. Why? The value of any pointer returned by a USM allocation on the host is guaranteed to be a valid pointer value also on the device. We have seen that the buffer-accessor API is powerful and also quite intuitive in a modern C++ setting. However, most programmers are quite familiar with pointer-based memory management, especially if they have been working with low-level CUDA/HIP languages. Furthermore, it is difficult to adopt SYCL in an existing codebase when it requires radical changes in fundamental infrastructure. USM offers a path forward.

unified shared memory

The standard provides three APIs for allocating USM:

- C-like (untyped)
- C++-like (typed)
- C++ allocator object usm_allocator.

C-like malloc

```
void* malloc_device(size_t numBytes,
               const queue& syclQueue,
               const property_list& prop_list = {});

void* malloc_host(size_t numBytes,
              const queue& syclQueue,
              const property_list& prop_list = {});

void* malloc_shared(size_t numBytes,
               const queue& syclQueue,
               const property_list& prop_list = {});
```

C++-like malloc

```
template <typename T>
T* malloc_device(size_t count,
             const queue& syclQueue,
             const property_list& prop_list = {});

template <typename T>
T* malloc_host(size_t count,
            const queue& syclQueue,
            const property_list& prop_list = {});

template <typename T>
T* malloc_shared(size_t count,
             const queue& syclQueue,
             const property_list& prop_list = {});
```

free

```
void free(void* ptr,
        queue& syclQueue);
```

As usual, you need to free any memory you claim dynamically from the runtime. The free function also needs information about the location of the memory, which can be conveniently conveyed by a queue object

unified shared memory

USM supports both explicit and implicit data movement strategies.

Explicit

We have to call memcpy (untyped C-like API) and copy (typed C++-like API) explicitly whenever data needs to migrate between different backends. These methods are available both on the queue and handler classes. Remember methods of the queue and handler class are asynchronous! Copies are not an exception! Explicit data movement is only strictly necessary for host-to-device and device-to-host data migrations. Indeed, device allocations cannot be directly accessed from the host.

SYCL

```
constexpr auto N = 256;
queue Q;

std::vector<double> x_h(N);
std::iota(x_h.begin(), x_h.end(), 0.0);

auto x_d = malloc_device<double>(N, Q);

// in a handler
Q.submit([&](handler& cgh){
   // untyped API
   cgh.memcpy(x_d, x_h.data(), N*sizeof(double));
   // or typed API
   //cgh.copy(x_d, x_h.data(), N);
});

// or on the queue directly
// with typed API
//Q.copy(x_d, x_h.data(), N);
//or untype API
//Q.memcpy(x_d, x_h.data(), N*sizeof(double));

// copies are ASYNCHRONOUS!!
Q.wait();
```

CUDA

```
constexpr auto N = 256;

std::vector<double> x_h(N);
std::iota(x_h.begin(), x_h.end(), 0.0);

double* x_d;
cudaMalloc((void**)&x_d, N*sizeof(double));

cudaMemcpy(x_d, x_h.data(), N*sizeof(double), cudaMemcpyHostToDevice);
```

unified shared memory

USM supports both explicit and implicit data movement strategies.

Implicit

This movement strategy requires no programmer intervention and is relevant for host and shared allocations. When the former are accessed on a device, the runtime will transfer the memory through the appropriate hardware interface. Host memory allocations do not migrate to the device, so they incur latency and repeated accesses are discouraged. Shared memory is essentially defined by its ability to migrate between host and device. This happens simply by accessing the same memory location from different locations.

```
constexpr auto N = 256;
queue Q;

auto x_h = malloc_host<double>(N, Q);
for (auto i = 0; i < N; ++i) {
  x_h[i] = static_cast<double>(i);
}

auto x_s = malloc_shared<double>(N, Q);

// in a handler
Q.submit([&](handler& cgh){
   cgh.parallel_for(range{N}, [=](id<1> tid){
   // get index out of id object
   auto i = tid[0];
   x_s[i] = x_h[i] + 1.0;
  }
});

// or on the queue directly
//Q.parallel_for(range{N}, [=](id<1> tid){
//  // get index out of id object
//  auto i = tid[0];
//  x_s[i] = x_h[i] + 1.0;
//}

Q.wait();
```

data & task parallelism

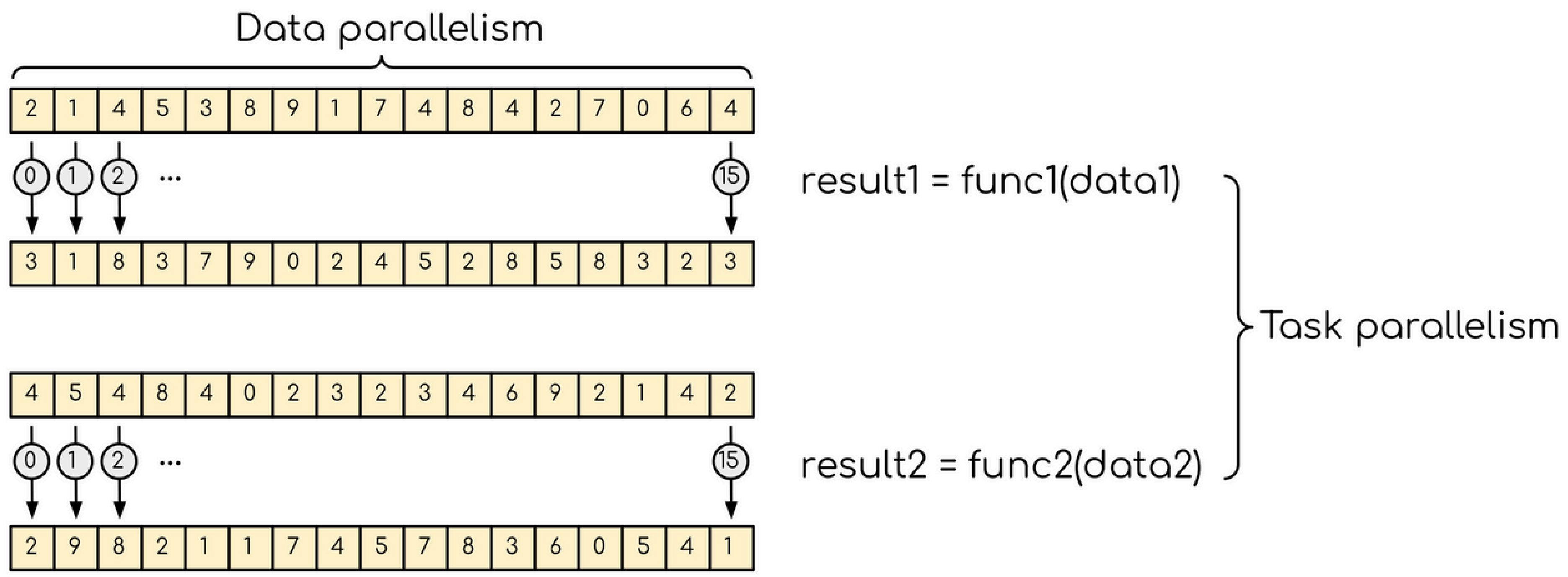
Data parallelism

The data can be distributed across computational units that can run in parallel. Each unit processes the data applying the same or very similar operations to different data elements. A common example is applying a blur filter to an image: the same function is applied to all the pixels the image is made of. This parallelism is natural for the GPU, where the same instruction set is executed in multiple threads.

Task parallelism

When an application consists of many tasks that perform different operations with (the same or) different data. An example of task parallelism is cooking: slicing vegetables and grilling are very different tasks and can be done at the same time, if more cooks are available. Note that the tasks can consume totally different resources, for example a CPU and a GPU, adding yet another dimension for performance optimization.

same operation applies to multiple data



From SYCL workshop by ENCSS. (Under CC-BY-4.0 license)

there are more than one independent task that, in principle, can be executed in parallel.

data parallelism

The parallel_for function is the basic data-parallel construct in SYCL and it accepts two arguments: an execution range and a kernel function. Based on the way we specify the execution range, we can distinguish three flavors of data-parallel kernels in SYCL:

- **basic:** when execution is parameterized over a 1-, 2- or 3-dimensional range object. As the name suggests, simple kernels do not provide control over low-level features, for example, control over the locality of memory accesses.
- **ND-range:** when execution is parameterized over a 1-, 2-, or 3-dimensional nd_range object. While superficially similar to simple kernels, the use of nd_range will allow you to group instances of the kernel together. You will gain more flexible control over locality and mapping to hardware resources.
- **hierarchical:** these form allows to nest kernel invocations, affording some simplifications with respect to ND-range kernels. We will not discuss hierarchical parallel kernels. This is a fast-changing area in the standard: support in various implementation varies and may not work properly.

simple parallel_for

```
 template <typename KernelName, int Dims, typename... Rest>
event parallel_for(range<Dims> numWorkItems, Rest&&... rest);
```

ND-range parallel_for

```
template <typename KernelName, int Dims, typename... Rest>
event parallel_for(nd_range<Dims> executionRange, Rest&&... rest);
```

data parallelism

For parallel processing, it’s necessary to specify processing units and, for each processing unit, which data to process. In SYCL, this is specified with **sycl::range** and **sycl::id**. While “processing unit” was mentioned, in SYCL, this is called a **work-item**. Therefore, **sycl::range** specifies the number of work-items, **sycl::id** specifies the ID of each work-item, and **parallel_for** describes the processing content to be actually executed by each work-item.

The first argument to **parallel_for** specifies **sycl::range**. **sycl::range** indicates the number of work-items.
In the case of a 1D array, there’s no particular reason to use **sycl::range**, but for 2D or 3D arrays, **sycl::range** can be used to explicitly indicate the dimensionality of the array. The second argument to **parallel_for** specifies the SYCL kernel function, and **sycl::id** is passed as an argument to this function object. **sycl::id** carries information about each work-item, allowing its ID to be checked.

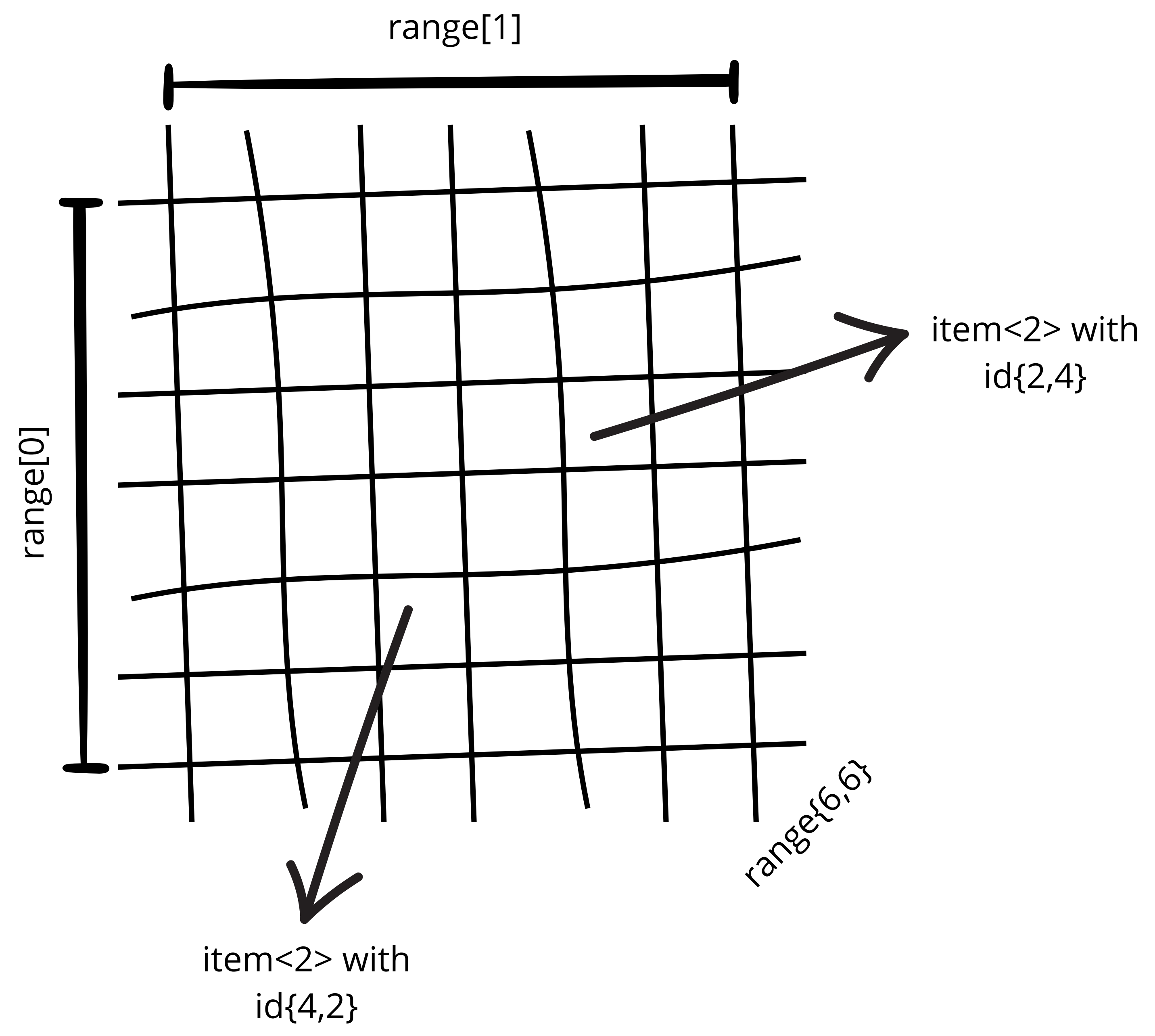


Figure adapted from SYCL workshop by ENCSS. (Under CC-BY-4.0 license)

A range<2> object, representing a 2-dimensional execution range. Each element in the range is of type item<2> and is indexed by an object of type id<2>. Items are instances of the kernel. An N-dimensional range is in row-major order: dimension N-1 is contiguous

ND-range data-parallel

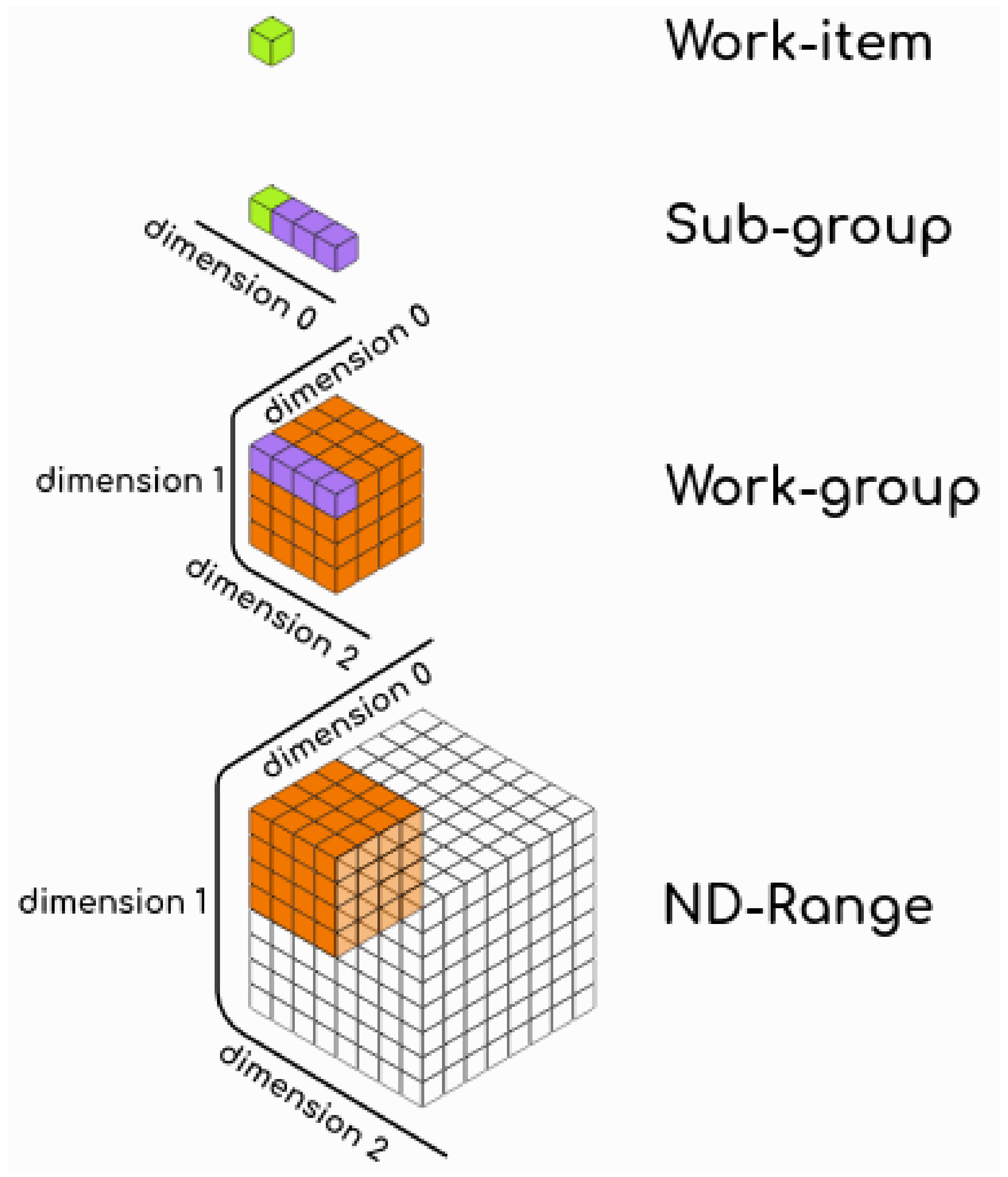
ND-ranges are represented with objects of type nd_range, templated over the number of dimensions:

nd_range constructor

```
nd_range(range<dimensions> globalSize, range<dimensions> localSize);
```

these are constructed using two range objects, representing the global and local execution ranges:

- The *global range* gives the total size of the nd_range: a 1-, 2-, or 3-dimensional collection of **work-items**. This is exactly like range objects: at their coarsest level the two objects look exactly the same.
- The *local range* gives the size of each **work-group** comprising the nd_range.
- The implementation can further subdivide each work-group into 1-dimensional sub-groups. Since this is an implementation-dependent feature, its size cannot be set in the nd_range constructor.

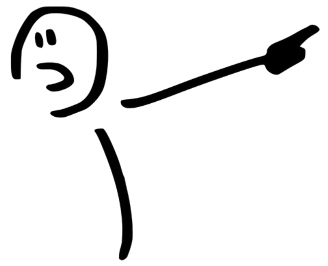


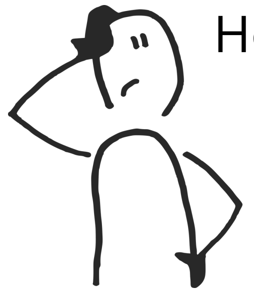
From SYCL workshop by ENCSS. (Under CC-BY-4.0 license)

The global execution range is 8x8x8 , thus containing 512 work-items. The global range is further subdivided into 8 work-groups each comprised of 4x4x4 work-items. At an even finer level, each work-group has sub-groups of 4 work-items. ***The availability of sub-groups is implementation-dependent.*** *Note that the contiguous dimension (dimension 2 in this example) of ND-range and work-group coincide. Furthermore, sub-groups are laid out along the contiguous dimension of their work-groups.*

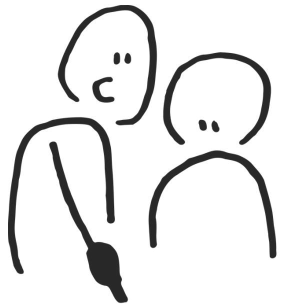
The task graph

The essential role of the runtime in SYCL applications is to schedule work submitted to the queues in our program as actions and execute all enqueued work in an asynchronous fashion.





How is this achieved?



A **task graph** is composed of nodes and edges, it has a start-to-finish direction, and no self-loops: **it is a directed acyclic graph (DAG).** Each node is an action to be performed on a device, such as a parallel_for. Each edge connecting two nodes is a dependency between the two, such as the data for task B produced by task A.

It is not usually necessary to directly define the task graph for your application.

- If you use buffers and accessors, the runtime can correctly generate the task graph based on their use in kernel code.
- Similarly when you use USM with implicit data movement.
- For USM with explicit data movements, you are taking up scheduling power.

However, interacting with the task graph might be necessary to get performance

Examples of a linear chain (left) and Y-pattern (right) of dependencies. The Y-pattern might occur, for example, in the AXPY routine: tasks A and B fill the operand vectors, thus are able to run independently, and task C performs the summation. Task D might be a subsequent reduction. We can always “linearize” a Y-pattern, for example, by using a in-order queue

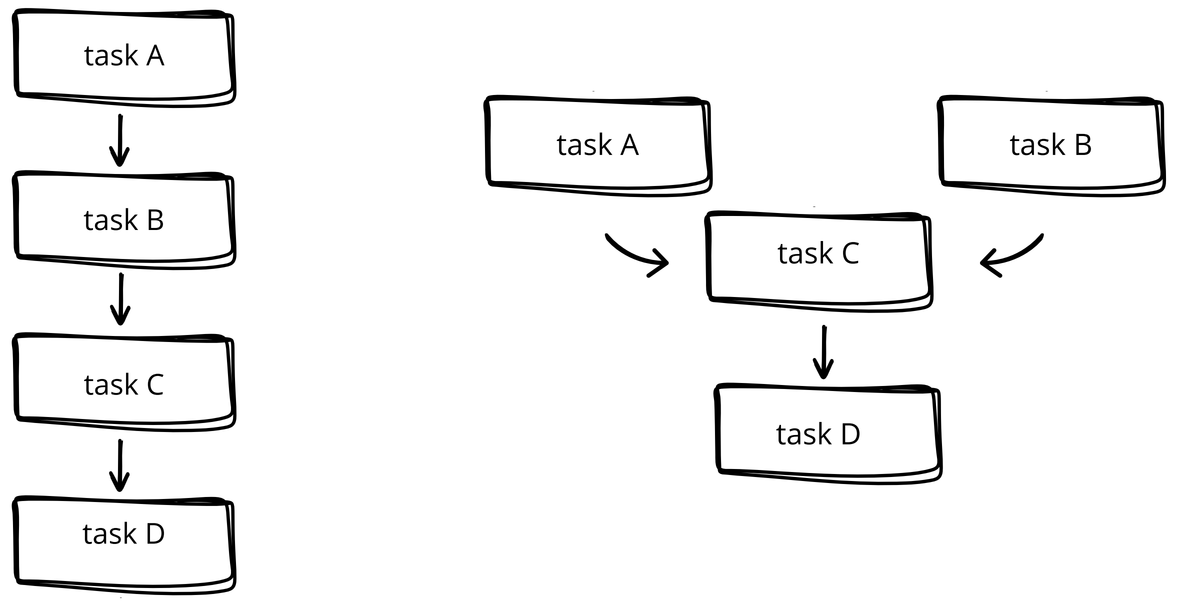


Figure adapted from SYCL workshop by ENCSS. (Under CC-BY-4.0 license)

The task graph

Out-of-order queues

This is the default for queue objects and leaves all decisions on task ordering to the runtime, unless we intervene. The runtime will decide ordering based on the data dependencies declared in our code.

In-order queues

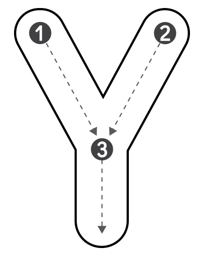
Each action is assumed to be dependent on the action immediately preceding it in the queue. In-order queues are quite simple to reason about, but they will constrain the runtime too much. Tasks will be serialized in a linear chain, even though they do not read/write to the same data. For this reason, in-order semantics is not the default, but must be set explicitly at construction:

```
queue ioQ{property::queue::in_order()};
```

HOW TO DEFINE DEPENDENCIES

events

```
auto a = malloc_shared<double>(N, Q);
auto b = malloc_shared<double>(N, Q);

// task A
auto e1 = Q.parallel_for(range { N }, [=](id<1> id) {
  a[id[0]] = 1;
});

// task B
auto e2 = Q.parallel_for(range { N }, [=](id<1> id) {
  b[id[0]] = 2;
});

// task C
auto e3 = Q.parallel_for(range { N }, { e1, e2 }, [=](id<1> id) {
  a[id[0]] += b[id[0]];
});

// task D
Q.single_task(e3, [=]() {
  for (int i = 1; i < N; i++)
    data1[0] += data1[i];

  data1[0] /= 3;
});
```



Expressing the Y-pattern

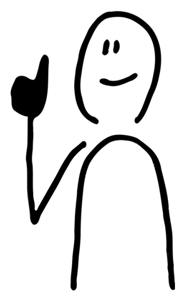
accessors

```
// task A
Q.submit([&](handler &h) {
  accessor aA { A, h };
  h.parallel_for(range { N }, [=](id<1> id) {
    aA[id] = 1;
  });
});

// task B
Q.submit([&](handler &h) {
  accessor aB { B, h };
  h.parallel_for(range { N }, [=](id<1> id) {
    aB[id] = 2;
  });
});

// task C
Q.submit([&](handler &h) {
  accessor aA { A, h };
  accessor aB { B, h, read_only };
  h.parallel_for(range { N }, [=](id<1> id) {
    aA[id] += aB[id];
  });
});

// task D
Q.submit([&](handler &h) {
  accessor aA { A, h };
  h.single_task([=]() {
    for (int i = 1; i < N; i++)
      aA[0] += aA[i];

    aA[0] /= 3;
  });
});
```

in-order queue

```
queue Q { property::queue::in_order() };

auto A = malloc_shared<double>(N, Q);
auto B = malloc_shared<double>(N, Q);

// task A
Q.parallel_for(range { N }, [=](id<1> id) {
  A[id[0]] = 1;
});

// task B
Q.parallel_for(range { N }, [=](id<1> id) {
  B[id[0]] = 2;
});

// task C
Q.parallel_for(range { N }, [=](id<1> id) {
  A[id[0]] += B[id[0]];
});

// task D
Q.single_task([=]() {
  for (int i = 1; i < N; i++)
    A[0] += A[i];

  A[0] /= 3;
});
```

three available ways



Synchronization

Once our device computations are done, we'd obviously like to get the results back on the host. In CUDA/HIP this usually takes the form of device-to-host copies. These represent implicit synchronization points between host and device: we wait until all kernels have completed and then perform the copy. In SYCL, we have few options:

1 We can use the **wait** method on the q**ueue** object. Even though this has been used extensively in our example, it is also the coarsest synchronization level and might not be a good idea in larger-scale applications. We might submit many actions to a queue and **using wait will block execution** until each and every one of them has completed, which is clearly not always ideal.

2 For finer control, you can synchronize on events: either a single one or a list.

```
// waiting on a single event

auto e1 = Q.parallel_for(...);
e1.wait();

// waiting on multiple events
auto e2 = Q.parallel_for(...);
auto e3 = Q.single_task(...);
event::wait({e2, e3});
```

3 Use of objects of host_accessor type sits at an even finer level. They define a new dependency between a task in the graph and the host, such that execution cannot proceed past their construction until the data they access is available on the host. More concisely, construction of an host_accessor is blocking.

```
// declare buffer
buffer<double> A{range{256}};

// fill with ones
Q.submit([&](handler &cgh){
  accessor aA{A, cgh};
  cgh.parallel_for(range{N}, [=](id<1> id){
    aA[id] = 1.0;
  });
});

// enqueue more work

// host accessor for buffer A
// the constructor will *block* until data is available on host
host_accessor h_a{A};
```

Buffer destructors are also blocking: when a buffer goes out of scope, it will implicitly wait for all actions that use it to complete. If the buffer was initialized with a host pointer, then the runtime will schedule a copy back to the host

```
std::vector a(256, 0.0);

{ // open scope
  // buffer to a
  buffer<double> buf_a(a.data(), range{256});

  // use buffer in work submitted to the queue
  Q.submit([&](handler &cgh){
    auto acc_a = accessor(buf_a, cgh);

    cgh.parallel_for(...);
  });
} // close scope: buffer destructor will wait and host data will be updated
```