Introduction to SYCL

Based on introduction to heterogeneous programming with Data Parallel C++ from Intel

Sylabus

- What is DPC++ and SYCL
- SYCL Basics
  - ”Hello World” Example
  - Memory Model: Buffers, Unified Shared Memory
  - Device Selection
  - Synchronization
  - Error Handling
- Compilation and Execution Flow

What is DPC++ and SYCL

Data Parallel C++

Standards-based, Cross-architecture Language

DPC++ = ISO C++ and Khronos SYCL and community extensions

The final SYCL 2020 Specification published in 2021

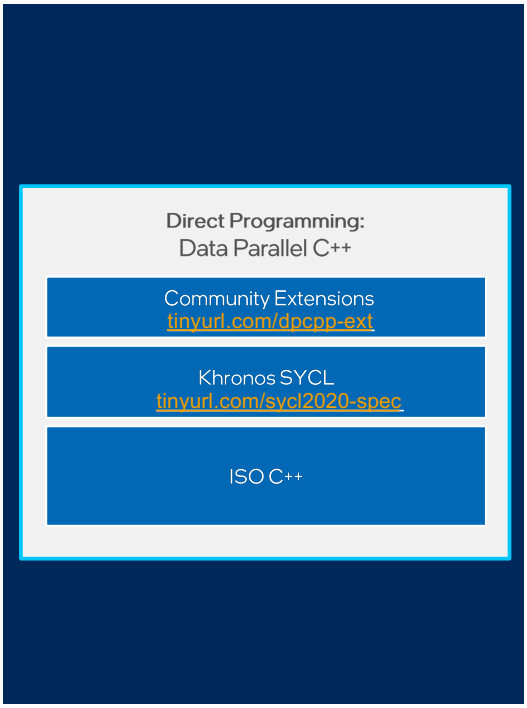
Today's DPC++ compiler is a mix of SYCL 1.2.1, SYCL 2020, and Language Extensions

Community Project Drives Language Enhancements

Many DPC++ extensions became features of SYCL 2020

- USM, sub-groups, group algorithms, reductions, etc.
- Interfaces enhanced based on feedback from SYCL working group
- Many APIs differ in SYCL 2020 to their DPC++ Extension versions

tinyurl.com/sycl2020-support-in-dpcpp



Intel oneAPI DPC++/C++ Compiler

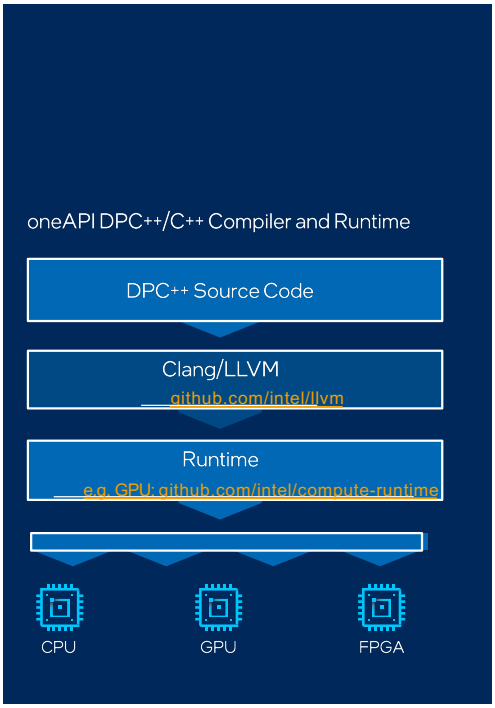
Parallel Programming Productivity & Performance

- Goal: deliver parallel programming productivity across CPU, GPU, FPGA, …
- Open, cross-industry alternative, Intel lead, to single architecture proprietary language
- The open source DPC++ compiler supports Intel CPUs, GPUs, and FPGAs + Nvidia GPUs
- SYCL backends supported: OpenCL, Level Zero, CUDA
- Performance portable ???

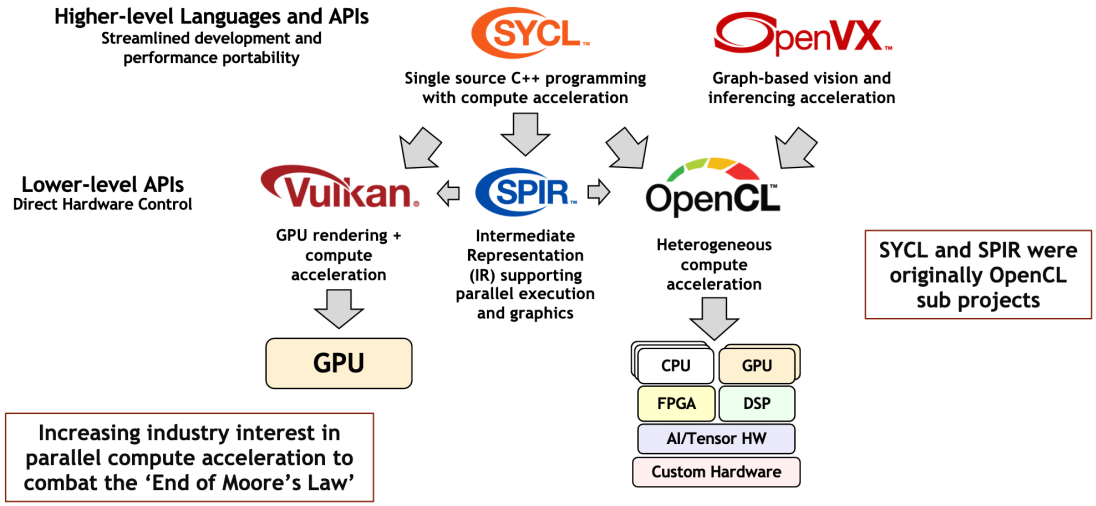
Code samples:

github.com/intel/llvm/tree/sycl/sycl/test

github.com/oneapi-src/oneAPI-samples



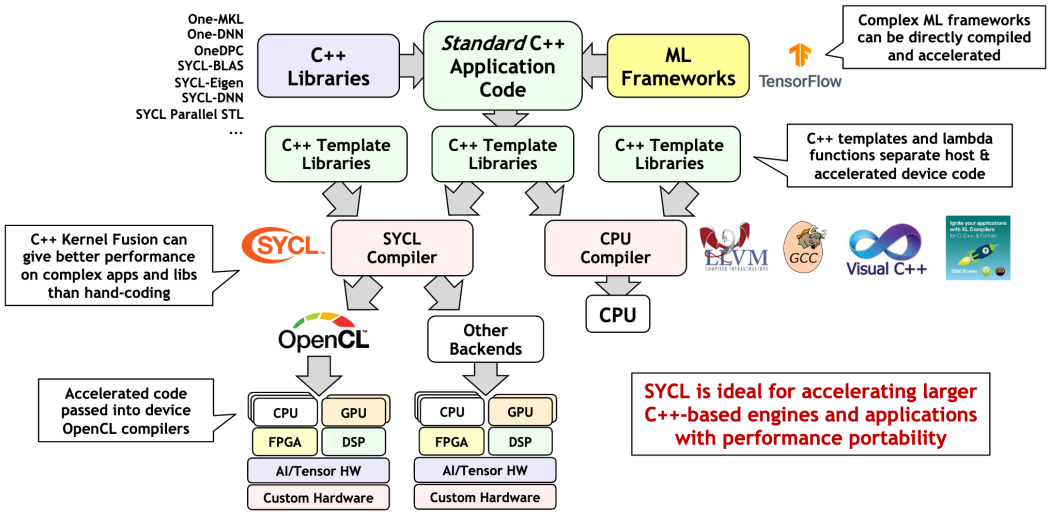
Khronos Compute Acceleration Standards



Source: https://www.khronos.org/assets/uploads/developers/presentations/SYCL-2020-Launch-Feb21.pdf

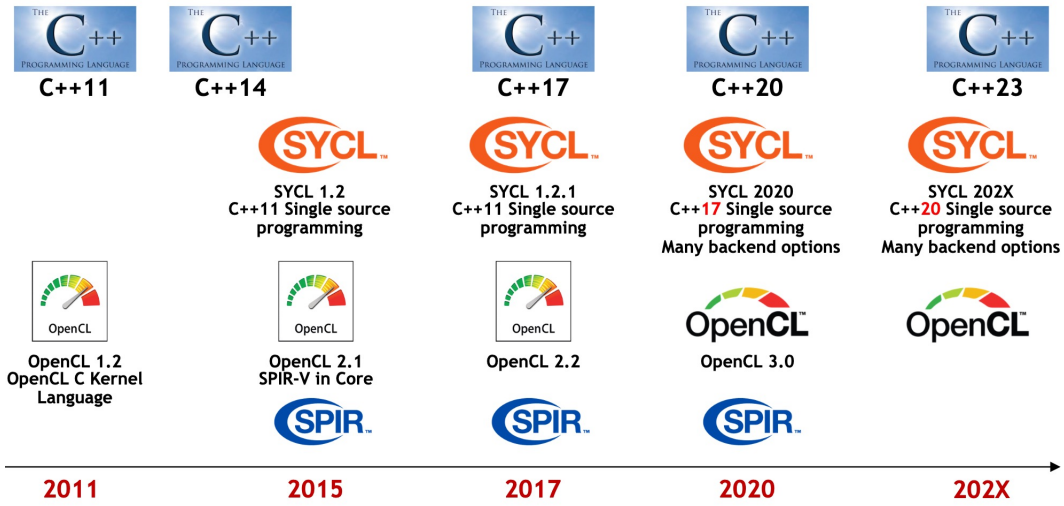
How familiar are you with C++?

SYCL ~~Single Source~~ C++ Parallel Programming

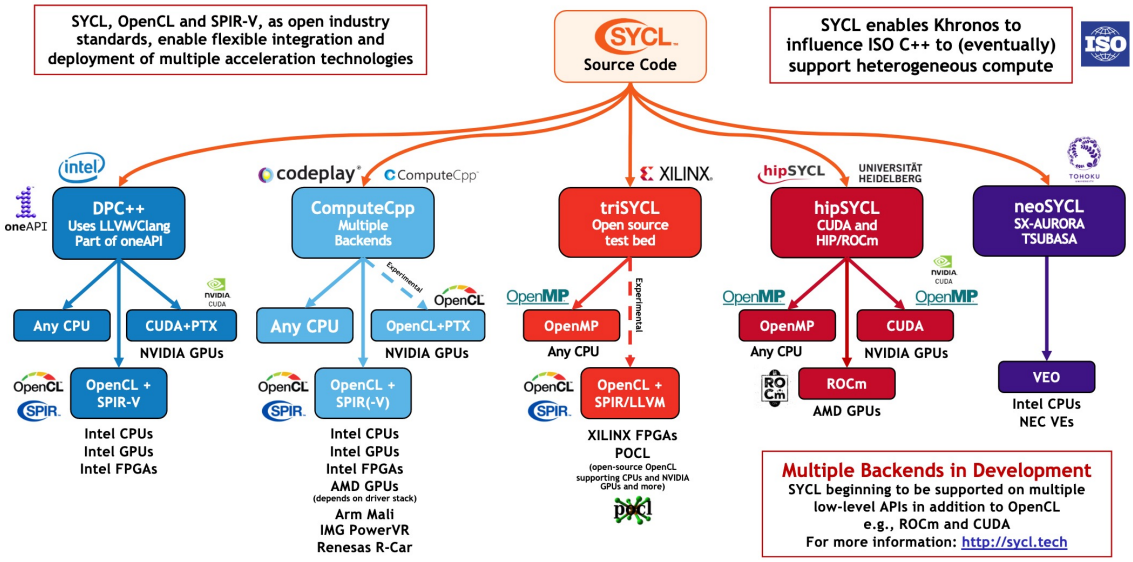


Source: https://www.khronos.org/assets/uploads/developers/presentations/SYCL-2020-Launch-Feb21.pdf

SYCL Chronology

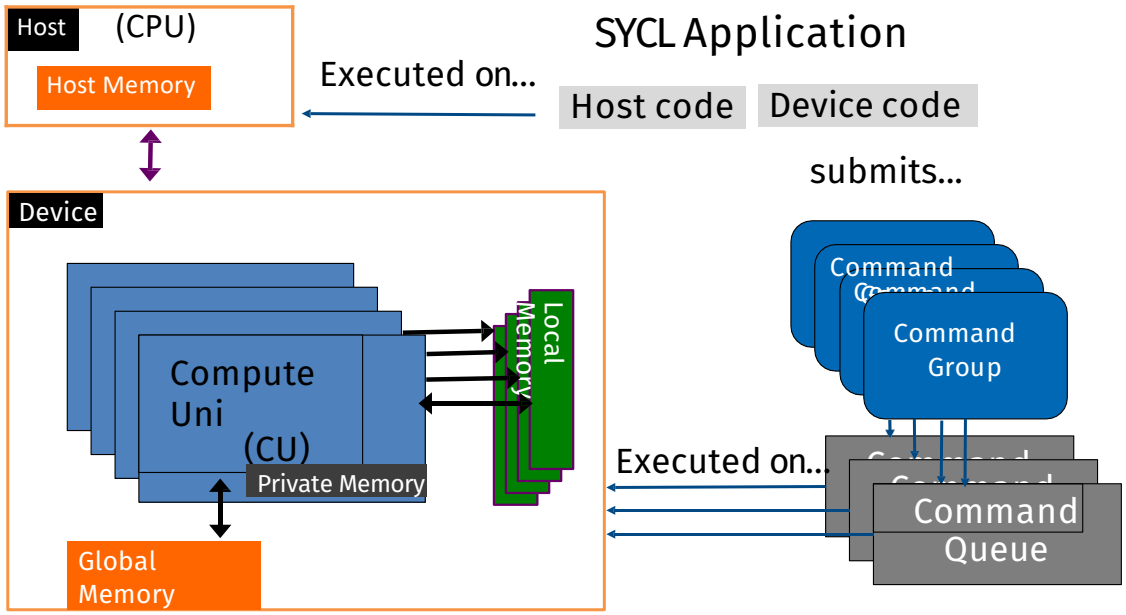


SYCL implementations



”Hello World” Example

SYCL Basics



Anatomy of a SYCL Application

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

int main() {
std::vector<float> A(1024), B(1024), C(1024);
// some data initialization
    {
      buffer bufA {A}, bufB {B}, bufC {C};
      queue q;
      q.submit([&](handler &h) {
          auto A = bufA.get_access(h, read_only);
          auto B = bufB.get_access(h, read_only);
          auto C = bufC.get_access(h, write_only);
          h.parallel_for(1024, [=](auto i){
              C[i] = A[i] + B[i];
          });
      });
    }
for (int i = 0; i < 1024; i++)
       std::cout << "C[" << i << "] = " << C[i] << std::endl;

}
```

Host code

Accelerator device code

Host code

Anatomy of a SYCL Application

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

int main() {
std::vector<float> A(1024), B(1024), C(1024);
// some data initialization
    {
      buffer bufA {A}, bufB {B}, bufC {C};
      queue q;
      q.submit([&](handler &h) {
          auto A = bufA.get_access(h, read_only);
          auto B = bufB.get_access(h, read_only);
          auto C = bufC.get_access(h, write_only);
          h.parallel_for(1024, [=](auto i){
              C[i] = A[i] + B[i];
          });
      });
    }
for (int i = 0; i < 1024; i++)
       std::cout << "C[" << i << "] = " << C[i] << std::endl;

}
```

Application scope

Command group scope

Device scope

Application scope

Memory Model

- Buffers: abstract view of memory that can be local to the host or a device, and is accessible only via accessors
- Images: a special type of buffer that has extra functionality specific to image processing
- Unified Shared Memory: pointer-based approach for memory model that is familiar for C++ programmers

SYCL Basics

```
std::vector<float> A(1024), B(1024), C(1024);
{
 buffer  bufA {A}, bufB {B}, bufC {C};


queue q;
q.submit([&](handler &h) {
    auto A = bufA.get_access(h, read_only);
    auto B = bufB.get_access(h, read_only);
    auto C = bufC.get_access(h, write_only);
    h.parallel_for(1024, [=](auto i){
        C[i] = A[i] + B[i];
    });
});  }
for (int i = 0; i < 1024; i++)
     std::cout << "C[" << i << "] = " << C[i] << std::endl;


}
```

Buffers creation via host vectors/pointers

Buffers encapsulate data in a SYCL application

- Across both devices and host!

SYCL Basics

```
std::vector<float> A(1024), B(1024), C(1024);
      {
     buffer bufA {A}, bufB {B}, bufC {C};
     queue q;
     q.submit([&](handler &h) {


     auto A = bufA.get_access(h, read_only);
     auto B = bufB.get_access(h, read_only);
     auto C = bufC.get_access(h, write_only);
     h.parallel_for(1024, [=](auto i){
          C[i] = A[i] + B[i];
     });
   }); // q.submit
}
for (int i = 0; i < 1024; i++)
     std::cout << "C[" << i << "] = " << C[i] << std::endl;

}
```

- A queue submits command groups to be executed by the SYCL runtime
- Queue is a mechanism where work is submitted to a device.

Queue Device Selection

- A programmer can select the device bound to a queue
- Multiple queues can be bound to the same device (concurrent operations)
- A queue **cannot be bound** to multiple devices

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

int main() {
  // Create queue to use the host device explicitly
  queue Q{ host_selector{} };

  std::cout << "Selected device: " <<
    Q.get_device().get_info<info::device::name>() << "\n";
  std::cout << " -> Device vendor: " <<
    Q.get_device().get_info<info::device::vendor>() << "\n";

  return 0;
}

Possible Output:
Device: SYCL host device
```

Source: Data Parallel C++

Where is my "Hello World" code executed?

Device Selector

Get a device (any device):	`queue q (); // default_selector{}`

Create queue targeting a pre-configured classes of devices:	`queue q(cpu_selector{});`
	`queue q(gpu_selector{});`
	`queue q(intel::fpga_selector{});`
	`queue q(accelerator_selector{});`
	`queue q(host_selector{});` SYCL 1.2.1

Create queue targeting specific device (custom criteria):	`class custom_selector : public device_selector {`
	`  int operator()(…… // Any logic you want!`
	
	`…`
	`queue q(custom_selector{});`

default_selector

- DPC++ runtime scores all devices and picks one with highest compute power
- Environment variable

export SYCL_DEVICE_TYPE=GPU | CPU | HOST

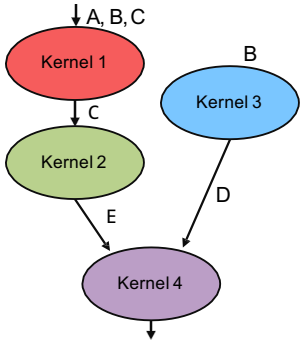
export SYCL_DEVICE_FILTER={backend:device_type:device_num}

tinyurl.com/dpcpp-env-vars for more details onenv variables

SYCL Basics

```
std::vector<float> A(1024), B(1024), C(1024);
    {
  buffer  bufA {A}, bufB {B}, bufC {C};
  queue q;  q.submit([&](handler
  &h) {
      auto A = bufA.get_access(h, read_only);
      auto B = bufB.get_access(h, read_only);
      auto C = bufC.get_access(h, write_only);
      h.parallel_for(1024, [=](auto i){
          C[i] = A[i] + B[i];
      });
  });
}
for (int i = 0; i < 1024; i++)

    std::cout << "C[" << i << "] = " << C[i] << std::endl;

}
```

- Accessors: Mechanism to access buffer data
- Create data dependencies in the SYCL graph that order kernel executions



Implicit Task/Kernel Dependencies with Accessors

- Data dependencies can create implicit task dependencies

Depedence Type	Description
Read-after-Write (RAW)	Occurs when task B needs to read data computed by task A
Write-after-Read (WAR)	Occurs when task B writes over data after task A has to read the same buffer
Write-after-Write (WAW)	Occurs when task B has to write data after task A writes the same data

¿Please identify the dependence in this code?

```
queue Q;

// We will learn how to simplify this example later
buffer A{a};
buffer B{b};
buffer C{c};

Q.submit([&](handler &h) {
    accessor accA(A, h, read_only);
    accessor accB(B, h, write_only);
    h.parallel_for( // computeB
      N,
      [=](id<1> i) { accB[i] = accA[i] + 1; });
  });

Q.submit([&](handler &h) {
    accessor accA(A, h, read_only);
    h.parallel_for( // readA
      N,
      [=](id<1> i) {
        // Useful only as an example
        int data = accA[i];
      });
  });
```

Source: Data Parallel C++

¿Please identify the dependence in these snippets?

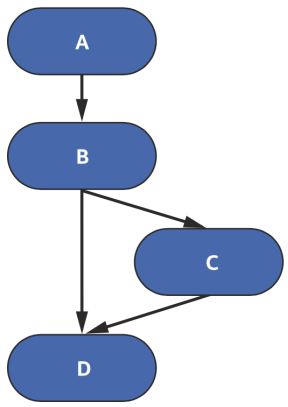
```
Q.submit([&](handler &h) {
    accessor accA(A, h, read_only);
    accessor accB(B, h, write_only);
    h.parallel_for( // computeB
      N,
      [=](id<1> i) { accB[i] = accA[i] + 1; });
  });

Q.submit([&](handler &h) {
    accessor accA(A, h, read_only);
    h.parallel_for( // readA
      N,
      [=](id<1> i) {
        // Useful only as an example
        int data = accA[i];
      });
  });

Q.submit([&](handler &h) {
    // RAW of buffer B
    accessor accB(B, h, read_only);
    accessor accC(C, h, write_only);
    h.parallel_for( // computeC
      N,
      [=](id<1> i) { accC[i] = accB[i] + 2; });
  });
```

```
Q.submit([&](handler &h) {
    accessor accA(A, h, read_only);
    accessor accB(B, h, write_only);
    h.parallel_for( // computeB
        N, [=](id<1> i) {
        accB[i] = accA[i] + 1;
        });
    });

Q.submit([&](handler &h) {
    // WAR of buffer A
    accessor accA(A, h, write_only);
    h.parallel_for( // rewriteA
        N, [=](id<1> i) {
        accA[i] = 21 + 21;
        });
    });

Q.submit([&](handler &h) {
    // WAW of buffer B
    accessor accB(B, h, write_only);
    h.parallel_for( // rewriteB
        N, [=](id<1> i) {
        accB[i] = 30 + 12;
        });
    });
```


Explicit dependencies with events

- Two approaches wait() and depends_on() events
- More later on …

```
#include <CL/sycl.hpp>
using namespace sycl;
constexpr int N = 4;

int main() {
  queue Q;

  // Task A
  auto eA = Q.submit([&](handler &h) {
    h.parallel_for(N, [=](id<1> i) { /*...*/ });
  });
  eA.wait();

  // Task B
  auto eB = Q.submit([&](handler &h) {
    h.parallel_for(N, [=](id<1> i) { /*...*/ });
  });

  // Task C
  auto eC = Q.submit([&](handler &h) {
    h.depends_on(eB);
    h.parallel_for(N, [=](id<1> i) { /*...*/ });
  });

  // Task D
  auto eD = Q.submit([&](handler &h) {
    h.depends_on({eB, eC});
    h.parallel_for(N, [=](id<1> i) { /*...*/ });
  });

  return 0;
}
```



SYCL Basics

```
std::vector<float> A(1024), B(1024), C(1024);
      {


buffer  bufA {A}, bufB {B}, bufC {C};
queue q;  q.submit([&](handler &h) {
    auto A = bufA.get_access(h, read_only);
    auto B = bufB.get_access(h, read_only);
    auto C = bufC.get_access(h, write_only);
        h.parallel_for(1024, [=](auto i){
            C[i] = A[i] + B[i];
        });
    });
  }
for (int i = 0; i < 1024; i++)

    std::cout << "C[" << i << "] = " << C[i] << std::endl;

}
```

range<1>{1024} id<1>

- Vector addition kernel enqueues a parallel_for task.
- Pass a function object/lambda to be executed by each work-item

Basic Parallel Kernels

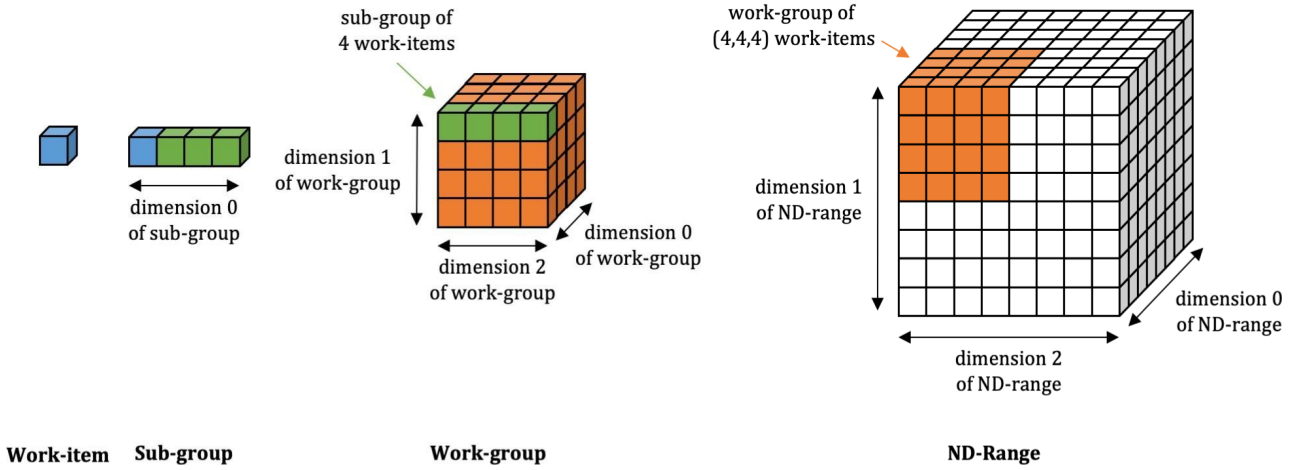
The functionality of basic parallel kernels is exposed via range, id and item classes

- range class is used to describe the iteration space of parallel execution
- id class is used to index an individual instance of a kernel in a parallel execution
- item class represents an individual instance of a kernel function, exposes additional functions to query properties of the execution range

```
h.parallel_for(range<1>(1024), [=](id<1> idx){
            // CODE THAT RUNS ON DEVICE
});
```

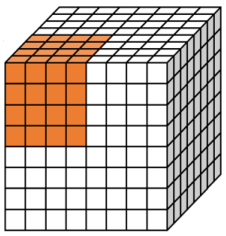
```
h.parallel_for(range<1>(1024), [=](item<1> item){
            auto idx = item.get_id();
            auto R = item.get_range();
            // CODE THAT RUNS ON DEVICE
});
```

SYCL Thread Hierarchy and Mapping



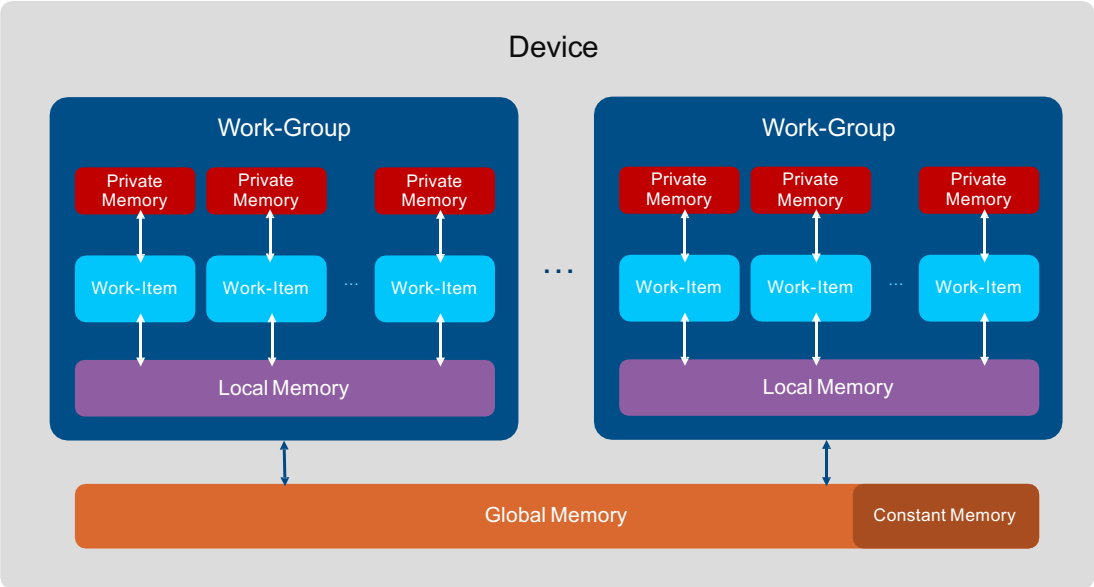
SYCL Thread Hierarchy and Mapping

All work-items in a **work-group** are scheduled on one Compute Unit, which has its own local memory



All work-items in a **sub-group** are mapped to vector hardware

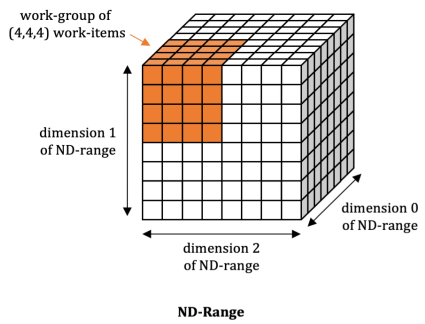
Logical Memory Hierarchy



ND-range Kernels

- Basic Parallel Kernels are easy way to parallelize a for-loop but does not allow performance optimization at hardware level
- ND-range kernel is another way to express parallelism which enable low level performance tuning by providing access to local memory and mapping executions to compute units on hardware

  - The entire iteration space is divided into smaller groups called work-groups, work-items within a work-group are scheduled on a single compute unit on hardware
  - The grouping of kernel executions into work-groups will allow control of resource usage and load balance work distribution



ND-Range

ND-range Kernels

The functionality of nd_range kernels is exposed via nd_range and nd_item classes

```
h.parallel_for(nd_range<1>(range<1>(1024),range<1>(64)), [=](nd_item<1> item){
    auto idx = item.get_global_id();
    auto local_id = item.get_local_id();
    // CODE THAT RUNS ON DEVICE
});
```

global size

work-group size

nd_range class represents a grouped execution range using global execution range and the local execution range of each work-group

nd_item class represents an individual instance of a kernel function and allows to query for work-group range and index

SYCL Basics

```
std::vector<float> A(1024), B(1024), C(1024);
     {


   buffer  bufA {A}, bufB {B}, bufC {C};
   queue q;  q.submit([&](handler &h) {
       auto A = bufA.get_access(h, read_only);
       auto B = bufB.get_access(h, read_only);  auto C = bufC.get_access(h,
       write_only);
       h.parallel_for(1024, [=](auto i){
            C[i] = A[i] + B[i];
       });
   });

     }
 for (int i = 0; i < 1024; i++)

        std::cout << "C[" << i << "] = " << C[i] << std::endl;

 }
```

Synchronization

Synchronization

- Synchronization within kernel function
  - Barriers for synchronizing work items within a workgroup
  - No synchronization primitives across workgroups
- Synchronization between host and device
  - Call to wait() member function of device queue
  - Buffer destruction will synchronize the data with host memory
  - Host accessor constructor is a blocked call and returns only after all enqueued kernels operating on this buffer finishes execution
  - DAG construction from command group function objects enqueued into the device queue

Host Accessors

- An accessor which uses host buffer access target
- Created outside of command group scope
- The data that this gives access to will be available on the host
- Used to synchronize the data back to the host by constructing the host accessor objects

Host Accessor

```
int main() {
  constexpr int N = 100;
  auto R = range<1>(N);
  std::vector<double> v(N, 10);
  queue q;

  buffer buf(v);
  q.submit([&](handler& h) {
  accessor a(buf, h)
  h.parallel_for(R, [=](auto i) {
    a[i] -= 2;
    });
  });

  host_accessor b(buf, read_only);
  for (int i = 0; i < N; i++)
    std::cout << b[i] << "\n";
  return 0;
}
```

- Buffer takes ownership of the data stored in vector
- Creating host accessor is a blocking call and will only return after all **enqueued DPC++ kernels** that modify the same buffer in any queue **completes** execution and the data is available to the host via this host accessor

Buffer Destruction

```
#include <CL/sycl.hpp>
constexpr int N=100;
using namespace cl::sycl;

void dpcpp_code(std::vector<double> &v, queue &q){
    auto R = range<1>(N);
    buffer buf(v);
    q.submit([&](handler& h) {
    accessor a(buf, h);
    h.parallel_for(R, [=](auto i) {
        a[i] -= 2;
        });
    });
}

int main() {
    std::vector<double> v(N, 10);
    queue q;
    dpcpp_code(v,q);
    for (int i = 0; i < N; i++)
        std::cout << v[i] << "\n";
    return 0;
}
```

- Buffer creation happens within a separate function scope
- When execution advances beyond this function scope, buffer destructor is invoked which relinquishes the ownership of data and copies back the data to the host memory

Error Handling

Synchronous exceptions

- Detected immediately
  - Failure to construct an object, e.g. can't create buffer
- Use try...catch block

```
try {
  device_queue.reset(new queue(device_selector));
}
catch (exception const& e) {
std::cout << "Caught a synchronous SYCL exception:" << e.what();
return;
}
```

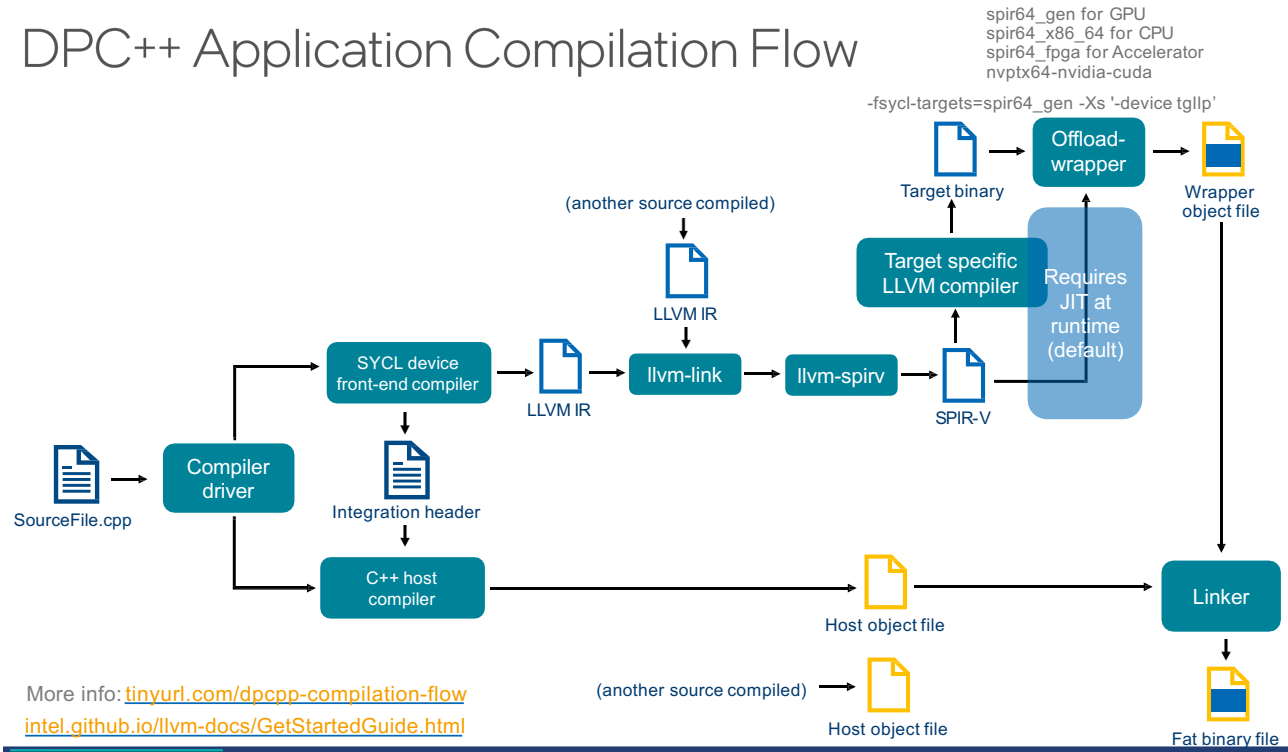
Asynchronous exceptions

- Caused by a future failure
  - E.g. error occurring during execution of a kernel on a device
  - Host program has already moved on to new things!
- Programmer provides processing function, and says when to process
- queue::wait _and_throw(), queue::throw_asynchronous(), event::wait_and_throw()

```
auto async_exception_handler = [](exception_list exceptions) {
  for (std::exception_ptr const& e : exceptions) {
    try {
      std::rethrow_exception(e);
    }
    catch (exception const& e) {
      std::cout << "Caught the Asynchronous SYCL exception"
                << e.what() << std::endl;
    }
  }
};
```

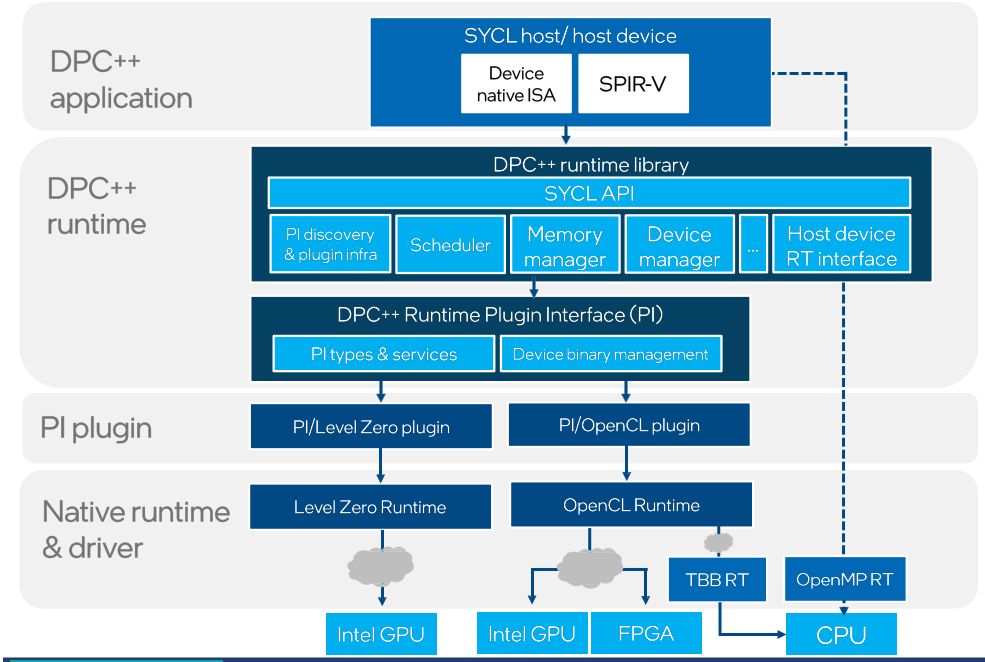
Compilation and Execution Flow

DPC++ Application Compilation Flow



More info: tinyurl.com/dpcpp-compilation-flow
intel.github.io/llvm-docs/GetStartedGuide.html

Runtime Architecture



Controlled via SYCL_BE
- PI_OPENCL
- PI_LEVEL_ZERO
- PI_CUDA
- PI_ROCM

or

SYCL_DEVICE_FILTER
- opencl
- level_zero
- cuda
- hip

Control Device Selection via SYCL_DEVICE_FILTER

- SYCL_BE is *deprecated* and replaced with SYCL_DEVICE_FILTER
- Syntax: backend:device_type:device_num, …
  - Backend: host, opencl, level_zero, cuda, hip, *
  - Device_type: host, cpu, gpu, acc, *
  - Device_num: unsigned integer
    - Enumeration index of devices from the sycl-ls utility

```
bso@scsel-cfl-10:/iusers/bso/sycl/llvm-priv3$ sycl-ls
0. ACC : Intel(R) FPGA Emulation Platform for OpenCL(TM) 1.2 [2020
1. GPU : Intel(R) OpenCL HD Graphics 3.0 [21.04.18912]
2. CPU : Intel(R) OpenCL 2.1 [2020.11.11.0.04_160000]
3. GPU : Intel(R) Level-Zero 1.0 [1.0.18912]
4. HOST: SYCL host platform 1.2 [1.2]
bso@scsel-cfl-10:/iusers/bso/sycl/llvm-priv3$
```

  - Each field is *optional*, so missing entry is regarded as '*'.
    - E.g., SYCL_DEVICE_FILTER=gpu ➔ SYCL_DEVICE_FILTER=*:gpu:*
  - Multiple triples can be specified separated by commas.
- Dual purposes
  - Users can specify their desired devices with the given triple(s).
  - SYCL only loads relevant plugins into runtime.

Check Your Configuration First

- sycl-ls --verbose

  0. CPU : Intel(R) OpenCL 2.1 [2021.12.6.0.19_160000]
  1. ACC : Intel(R) FPGA Emulation Platform for OpenCL(TM) 1.2 [2021.12.6.0.19_160000]
  2. GPU : Intel(R) OpenCL HD Graphics 3.0 [21.28.20343]
  3. GPU : Intel(R) Level-Zero 1.1 [1.1.20343]
  4. HOST: SYCL host platform 1.2 [1.2]

- https://github.com/intel/pti-gpu
  - https://github.com/intel/pti-gpu/tree/master/samples/gpu_info

```
Device Information:
  Device Name: Intel(R) HD Graphics 630
               (Kaby Lake GT2)
  EuCoresTotalCount: 24
  EuCoresPerSubsliceCount: 8
  EuSubslicesTotalCount: 3
  EuSubslicesPerSliceCount: 3
  EuSlicesTotalCount: 1
  EuThreadsCount: 7
  SubsliceMask: 7
  SliceMask: 1
  SamplersTotalCount: 3
  GpuMinFrequencyMHz: 350
  GpuMaxFrequencyMHz: 1150
  GpuCurrentFrequencyMHz: 350
  PciDeviceId: 22802
  SkuRevisionId: 4
  PlatformIndex: 12
  ApertureSize: 0
  NumberOfRenderOutputUnits: 4
  NumberOfShadingUnits: 28
  OABufferMinSize: 16777216
  OABufferMaxSize: 16777216
  GpuTimestampFrequency: 12000000
  MaxTimestamp: 357913941250
```


Unified Shared Memory

Motivation

The SYCL 1.2.1 standard provides a Buffer memory abstraction

- Powerful and elegantly expresses data dependences

However…

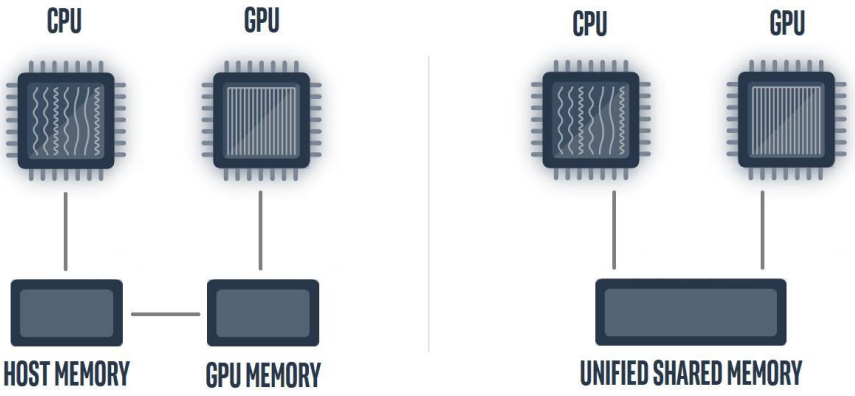
- Replacing all pointers and arrays with buffers in a C++ program can be a burden to programmers

USM provides a pointer-based alternative in SYCL

- Simplifies porting to an accelerator
- Gives programmers the desired level of control
- Complementary to buffers

Developer View Of USM

- Developers can reference same memory object in host and device code with Unified Shared Memory



Unified Shared Memory

Unified Shared Memory provides both explicit and implicit models for managing memory.

Allocation Type	Description	Accessible on HOST	Accessible on DEVICE
device	Allocations in device memory (**explicit**)	NO	YES
host	Allocations in host memory (**implicit**)	YES	YES
shared	Allocations can migrate between host and device memory (**implicit**)	YES	YES

Automatic data accessibility and explicit data movement supported

USM - Explicit Data Movement

```
queue q;
int hostArray[42];
int *deviceArray = (int*) malloc_device(42 * sizeof(int), q);

for (int i = 0; i < 42; i++) hostArray[i] = 42;
// copy hostArray to deviceArray
q.memcpy(deviceArray, &hostArray[0], 42 * sizeof(int));
q.wait();
q.submit([&](handler& h){
  h.parallel_for(42, [=](auto ID) {
    deviceArray[ID]++;
  });
});
q.wait();
// copy deviceArray back to hostArray
q.memcpy(&hostArray[0], deviceArray, 42 * sizeof(int));
q.wait();
free(deviceArray, q);
```

USM - Implicit Data Movement

```
queue q;
int *hostArray = (int*) malloc_host(42 * sizeof(int), q);
int *sharedArray = (int*) malloc_shared(42 * sizeof(int), q);

for (int i = 0; i < 42; i++) hostArray[i] = 1234;
q.submit([&](handler& h){
  h.parallel_for(42, [=](auto ID) {
    // access sharedArray and hostArray on device
    sharedArray[ID] = hostArray[ID] + 1;
  });
});
q.wait();
for (int i = 0; i < 42; i++) hostArray[i] = sharedArray[i];
free(sharedArray, q);
free(hostArray, q);
```


USM - Data Dependency in Queues

No accessors in USM

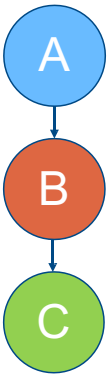
Dependences must be specified explicitly using events

- queue.wait()
- wait on event objects
- use the depends_on method inside a command group

USM - Data Dependency in Queues

Explicit wait() used to ensure data dependency in maintained

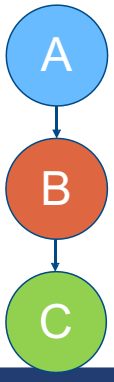
wait() will block execution on host




```
queue q;
int* data = malloc_shared<int>(N, q);
for(int i=0;i<N;i++) data[i] = 10;
q.submit([&] (handler &h){
    h.parallel_for<class taskA>(range<1>(N), [=](id<1> i){
        data[i] += 2;
    });
}).wait();
q.submit([&] (handler &h){
    h.parallel_for<class taskB>(range<1>(N), [=](id<1> i){
        data[i] += 3;
    });
}).wait();
q.submit([&] (handler &h){
    h.parallel_for<class taskC>(range<1>(N), [=](id<1> i){
        data[i] += 5;
    });
}).wait();
for(int i=0;i<N;i++) std::cout << data[i] << " ";
free(data, q);
```


USM - Data Dependency in Queues

Use depends_on() method to let command group handler know that specified event should be complete before specified task can execute.

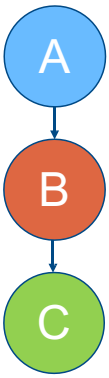



```
queue q;
int* data = malloc_shared<int>(N, q);
for(int i=0;i<N;i++) data[i] = 10;
auto e1 = q.submit([&] (handler &h){
    h.parallel_for<class taskA>(range<1>(N), [=](id<1> i){
        data[i] += 2;
    });
});
auto e2 = q.submit([&] (handler &h){
    h.depends_on(e1);
    h.parallel_for<class taskB>(range<1>(N), [=](id<1> i){
        data[i] += 3;
    });
});

// non-blocking; execution of host code is possible

q.submit([&] (handler &h){
    h.depends_on(e2);
    h.parallel_for<class taskC> (range<1>(N), [=](id<1> i){
        data[i] += 5;
    });
}).wait();
for(int i=0;i<N;i++) std::cout << data[i] << " ";
free(data, q);
```

USM - Data Dependency in Queues

Use in_queue property for the queue

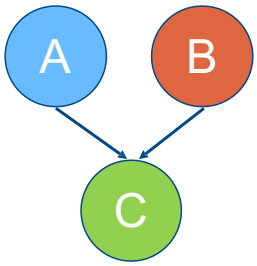
Execution will not overlap even if the queues have no data dependency




```
queue q{property::queue::in_order()};
int *data =  malloc_shared<int>(N, q);
for(int i=0;i<N;i++) data[i] = 10;
q.submit([&] (handler &h){
    h.parallel_for<class taskA>(range<1>(N), [=](id<1> i){
        data[i] += 2;
    });
});
// non-blocking; execution of host code is possible
q.submit([&] (handler &h){
    h.parallel_for<class taskB>(range<1>(N), [=](id<1> i){
        data[i] += 3;
    });
});
// non-blocking; execution of host code is possible
q.submit([&] (handler &h){
    h.parallel_for<class taskC>(range<1>(N), [=](id<1> i){
        data[i] += 5;
    });
}).wait();
for(int i=0;i<N;i++) std::cout << data[i] << " ";
free(data, q);
```


USM - Data Dependency in Queues

Use depends_on() method to let command group handler know that specified events should be complete before specified task can execute

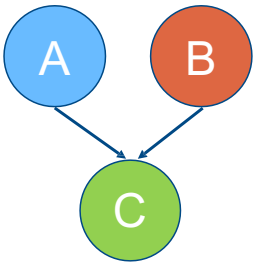



```
queue q;
int* data1 =  malloc_shared<int>(N, q);
int* data2 =  malloc_shared<int>(N, q);
for(int i=0;i<N;i++) {data1[i] = 10; data2[i] = 10;}
auto e1 = q.submit([&] (handler &h){
    h.parallel_for<class taskA>(range<1>(N), [=](id<1> i){
        data1[i] += 2;
    });
});
auto e2 = q.submit([&] (handler &h){
    h.parallel_for<class taskB>(range<1>(N), [=](id<1> i){
        data2[i] += 3;
    });
});
q.submit([&] (handler &h){
    h.depends_on({e1,e2});
    h.parallel_for<class taskC>(range<1>(N), [=](id<1> i){
        data1[i] += data2[i];
    });
}).wait();
for(int i=0;i<N;i++) std::cout << data[i] << " ";
free(data1, q); free(data2, q);
```

SYCL_PRINT_EXECUTION_GRAPH
tinyurl.com/dag-print

USM - Data Dependency in Queues

A more simplified way of specifying dependency as parameter of parallel_for




```
queue q;
int* data1 =  malloc_shared<int>(N, q);
int* data2 =  malloc_shared<int>(N, q);
for(int i=0;i<N;i++) {data1[i] = 10; data2[i] = 10;}
auto e1 = q.parallel_for <class taskA>(range<1>(N), [=](id<1> i){
  data1[i] += 2;
});
auto e2 = q.parallel_for <class taskB>(range<1>(N), [=](id<1> i){
  data2[i] += 3;
});
q.parallel_for <class taskC>(range<1>(N), {e1, e2}, [=](id<1> i){
  data1[i] += data2[i];
}).wait();

for(int i=0;i<N;i++) std::cout << data[i] << " ";
free(data1, q); free(data2, q);
```


Useful Links

Open source projects

oneAPI Data Parallel C++ compiler:	github.com/intel/llvm
Graphics Compute Runtime:	github.com/intel/compute-runtime
Graphics Compiler:	github.com/intel/intel-graphics-compiler
SYCL 2020:	tinyurl.com/sycl2020-spec
DPC++ Extensions:	tinyurl.com/dpcpp-ext
Environment Variables:	tinyurl.com/dpcpp-env-vars
DPC++ book:	tinyurl.com/dpcpp-book
oneAPI training:	colfax-intl.com/training/intel-oneapi-training
oneAPI Base Training Modules:	devcloud.intel.com/oneapi/get_started/baseTrainingModules/

Code samples:

github.com/intel/llvm/tree/sycl/sycl/test
github.com/oneapi-src/oneAPI-samples