oneAPI

Cross-Architecture Programming for Accelerated Compute, Freedom of Choice for Hardware

Direct Programming with Intel® oneAPI DPC++/C++ Compiler

June 2023

intel®

Agenda

- What is DPC++ and SYCL?
- Intel Compilers
- SYCL Basics
  - “Hello World” Example
    - Basic Concepts: buffer, accessor, queue, kernel, etc.
  - Device Selection
  - Synchronization
  - Error Handling
- Demo – part I

- Compilation and Execution Flow
- Unified Shared Memory
- Sub-groups
- Demo – part II

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

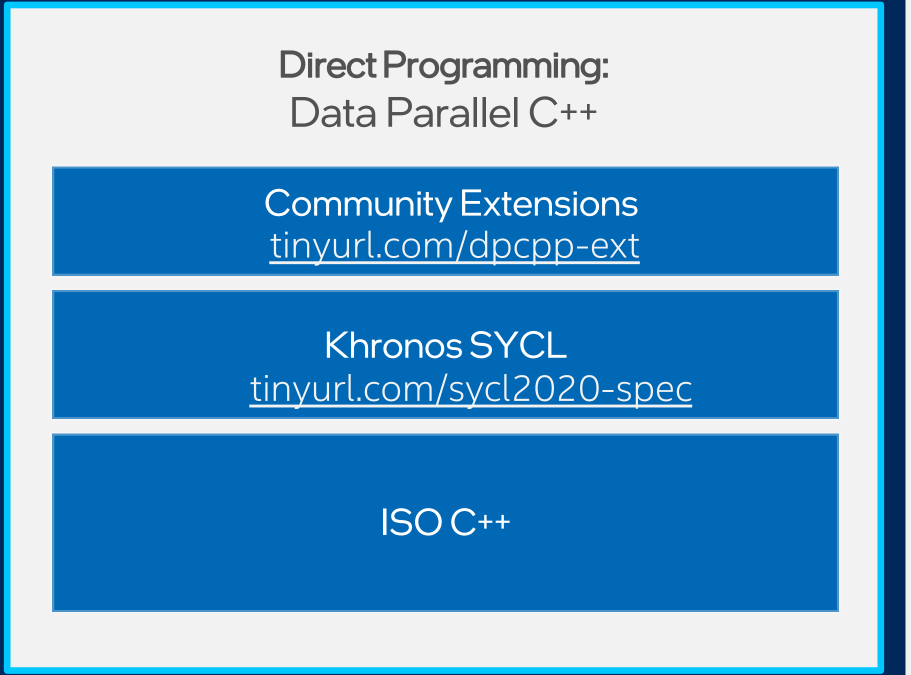
Today’s DPC++ compiler is a mix of SYCL 1.2.1, SYCL 2020, and Language Extensions

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

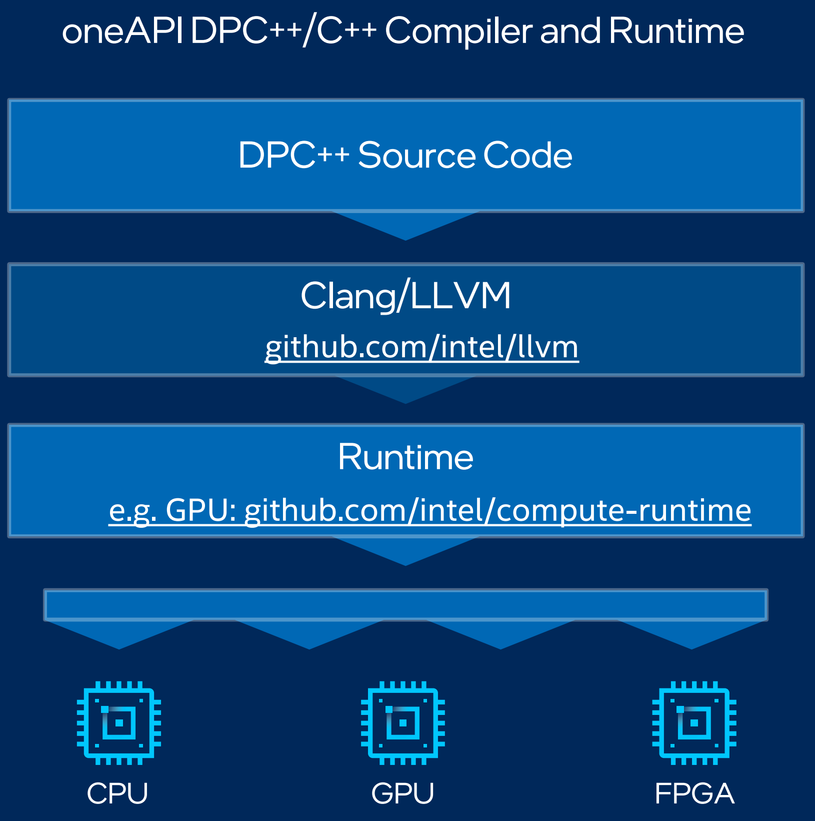
Parallel Programming Productivity & Performance

Compiler to deliver uncompromised parallel programming productivity and performance across CPUs and accelerators

- Open, cross-industry alternative to single architecture proprietary language
- The open source DPC++ compiler supports Intel CPUs, GPUs, and FPGAs + Nvidia and AMD GPUs
  - SYCL backends supported: OpenCL, Level Zero, CUDA*, HIP*

Code samples:

github.com/intel/llvm/tree/sycl/sycl/test
github.com/intel/llvm/tree/sycl/sycl/test-e2e
github.com/oneapi-src/oneAPI-samples



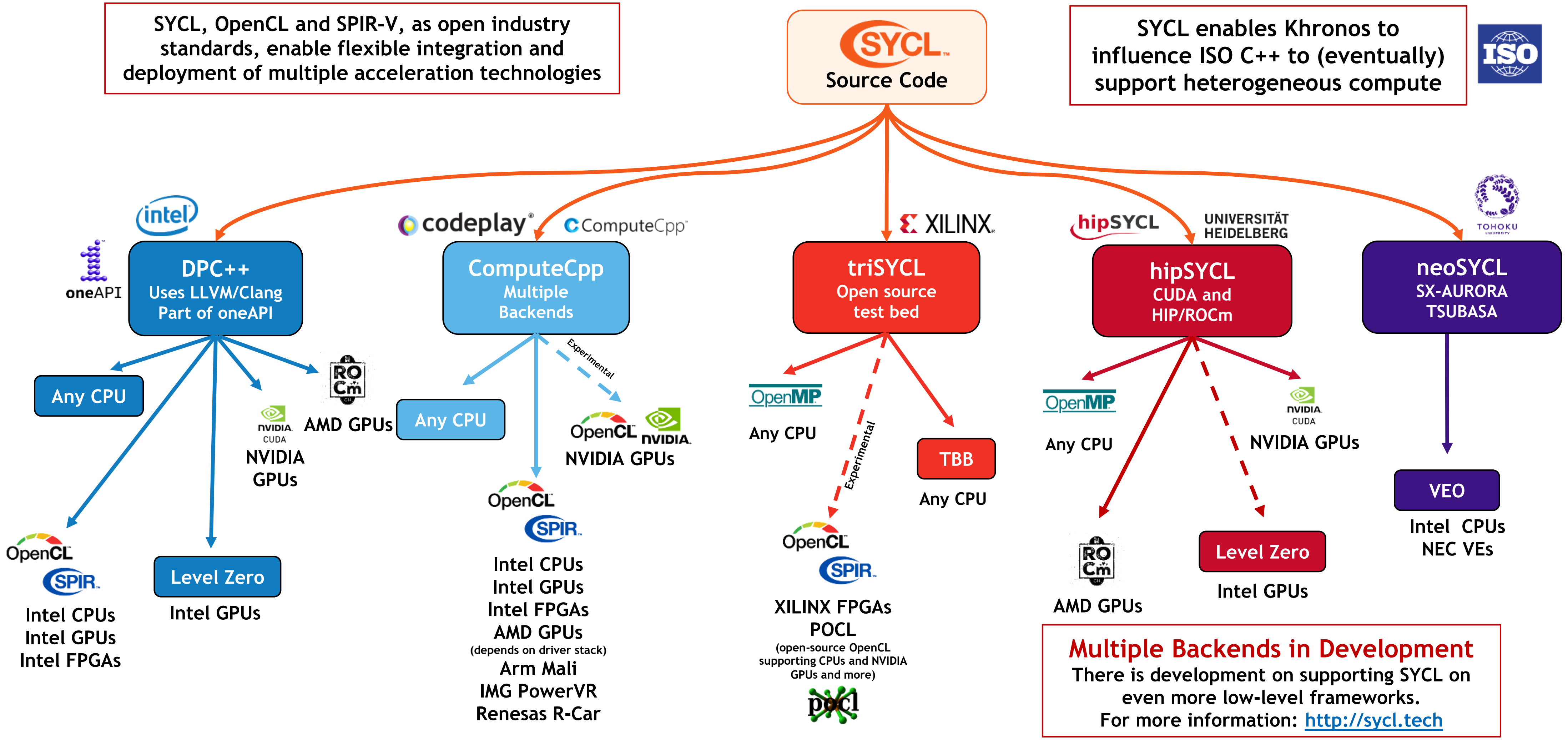
oneAPI DPC++/C++ Compiler and Runtime
DPC++ Source Code
Clang/LLVM
github.com/intel/llvm
Runtime
e.g. GPU: github.com/intel/compute-runtime
CPU
GPU
FPGA

There will still be a need to tune for each architecture.

SYCL ecosystem is growing

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

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



Multiple Backends in Development
There is development on supporting SYCL on even more low-level frameworks.
For more information: http://sycl.tech

+ Celerity: SYCL on MPI+SYCL

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

*This slide is prepared by The Khronos Group Inc

Codeplay oneAPI Plug-ins for Nvidia* & AMD*

Support for Nvidia & AMD GPUs to Intel® oneAPI Base Toolkit

oneAPI for NVIDIA & AMD GPUs

- Free download of binary plugins to Intel® oneAPI DPC++/C++ Compiler:
- Nvidia GPU
- AMD beta GPU
- No need to build from source!
- Plug-ins updated quarterly in-sync with SYCL 2020 conformance & performance

Priority Support

- Available through Intel, Codeplay & our channel
- Requires Intel Priority Support for Intel® oneAPI DPC++/C++ Compiler
- Intel takes first call, Codeplay delivers backend support
- Codeplay provides access to older plug-in versions

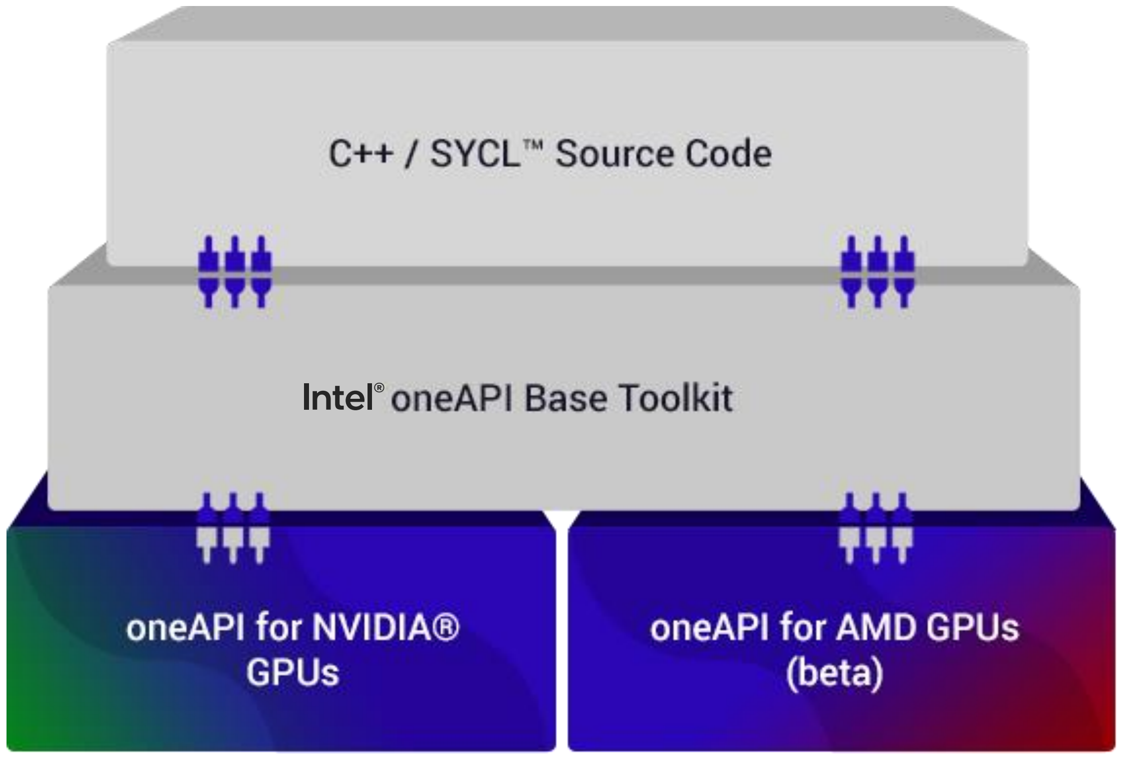


Image courtesy of Codeplay Software Ltd.

Nvidia GPU plug-in

AMD GPU plug-in

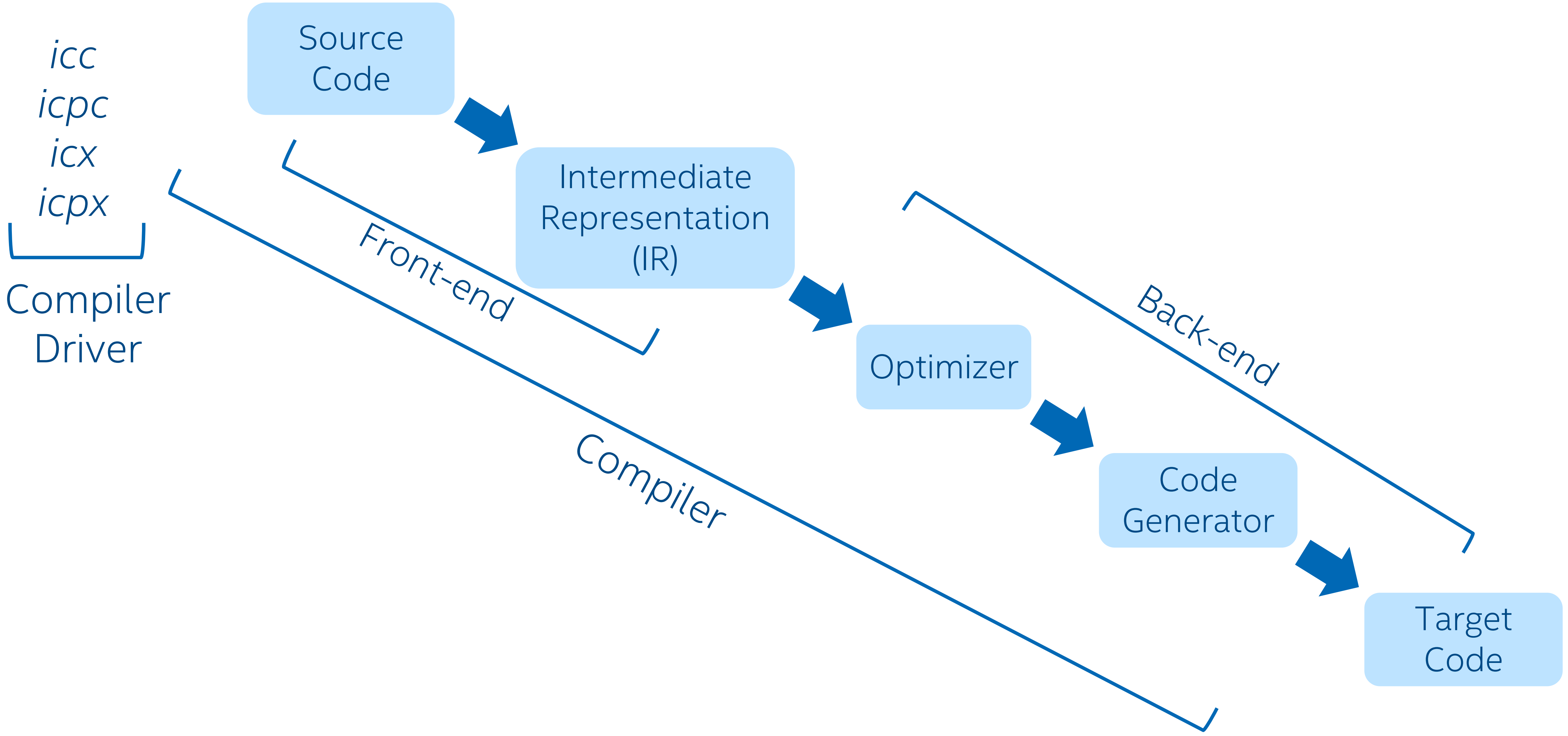
Codeplay blog

Codeplay press release

*Other names and brands may be claimed as the property of others.

Intel® Compilers

Compiler Architecture – Simplified View



Intel® C++ Compilers

Intel Compiler	Target	OpenMP Support	OpenMP Offload Support	Included in oneAPI Toolkit
Intel® C++ Compiler Classic, ILO ***icc/icpc/icl - deprecated***	CPU	Yes	No	HPC
Intel® Fortran Compiler Classic, ILO ***ifort***	CPU	Yes	No	HPC
Intel® oneAPI DPC++/C++ Compiler, LLVM ***icx/icpx/dpcpp****	CPU, GPU, FPGA	Yes	Yes	Base
Intel® Fortran Compiler, LLVM ***ifx***	CPU, GPU	Yes	Yes	HPC

Cross Compiler Binary Compatible and Linkable!

tinyurl.com/oneapi-standalone-components

* 'dpcpp' is deprecated and will be removed in a future release. Use 'icpx -fsycl'

Packaging of C++ Compilers

- oneAPI Base Toolkit *PLUS* oneAPI HPC Toolkit

  Classic compilers (icc/icpc) in HPC Toolkit

  v2021.9 code base

  Compilers based on LLVM* framework

  Compiler Drivers: icx/icpx and dpcpp*

  v2023.1 in oneAPI 2023.1

- Prerequisites: Set Up Your System for Intel GPU

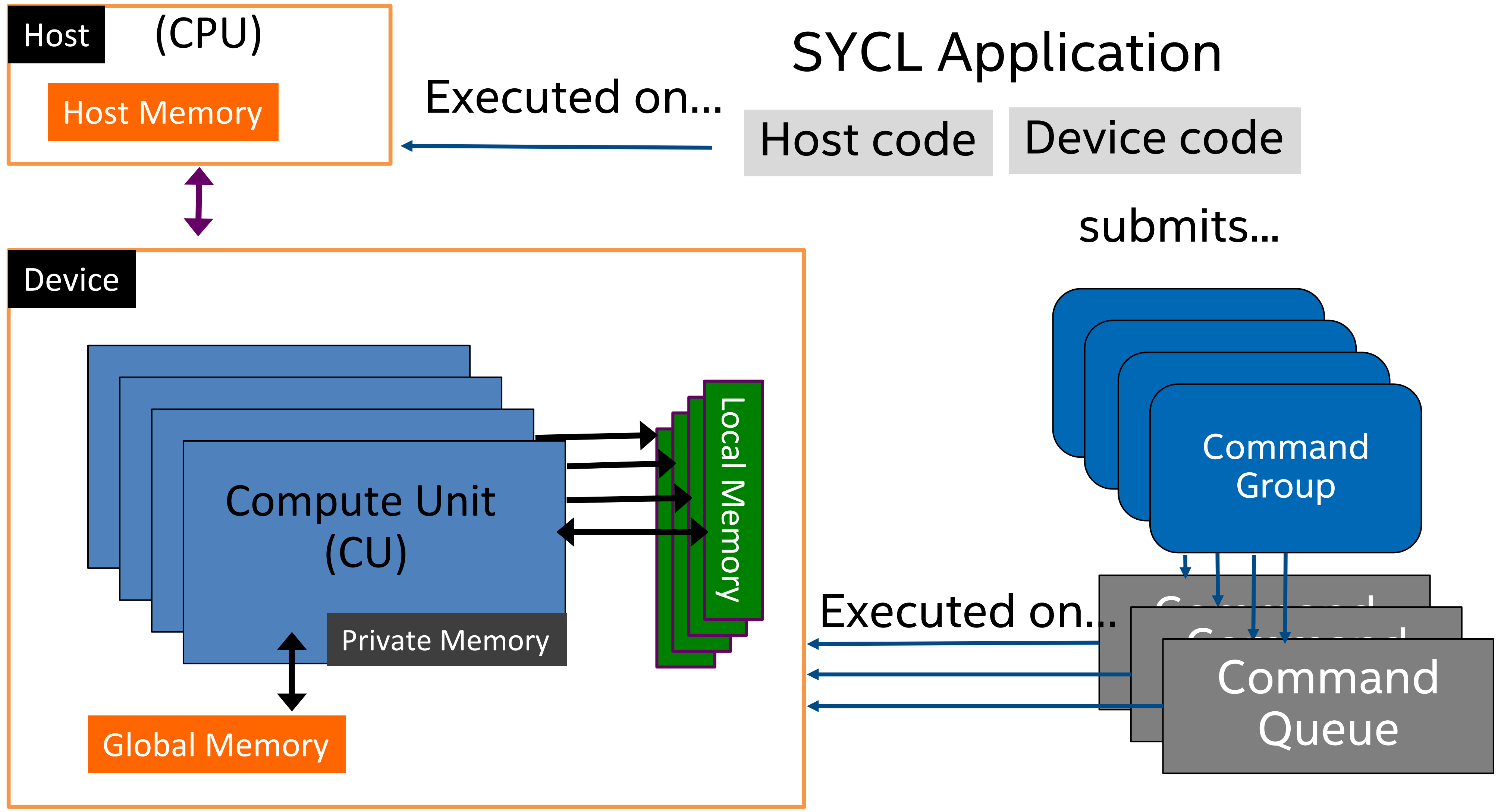
  Install Intel GPU Drivers, Disable Hangcheck etc.

tinyurl.com/oneapi-linux-install-guide

* 'dpcpp' is deprecated and will be removed in a future release. Use 'icpx -fsycl'

“Hello World” Example

SYCL Basics


Host
(CPU)
Host Memory
Executed on...
SYCL Application
Host code
Device code
submits...
Device
Compute Unit
(CU)
Private Memory
Local Memory
Global Memory
Command
Group
Executed on...
Command
Queue

Anatomy of a SYCL Application

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

int main() {
std::vector<float> A(1024, 1.0f), B(1024, 2.0f), 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;

}
```

Host code

Accelerator device code

Host code

Anatomy of a SYCL Application

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

int main() {
std::vector<float> A(1024, 1.0f), B(1024, 2.0f), 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;

}
```

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

intel.

SYCL Basics

```
std::vector<float> A(1024, 1.0f), B(1024, 2.0f), 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, 1.0f), B(1024, 2.0f), 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;

 }
```

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

Where is my “Hello World” code executed?

Device Selector

Get a device (any device):	`queue q (); // default_selector_v`

Create a queue with predefined device selectors	`queue q(cpu_selector_v);`
	`queue q(gpu_selector_v);`
	`queue q(accelerator_selector_v);`

Create a queue via custom selector	`int usm_selector(const sycl::device& dev) {`
	`  if (dev.has(sycl::aspect::usm_device_allocations)) {`
	`    if (dev.has(sycl::aspect::gpu)) return 2;`
	`    return 1;`
	`  }`
	`  return -1;`
	`}`
	`…`
	`queue q(usm_selector);`

default_selector_v

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

export ONEAPI_DEVICE_SELECTOR={backend:device_type:device_num}

ONEAPI_DEVICE_SELECTOR

Examples

ONEAPI_DEVICE_SELECTOR=

*opencl:**	Only the OpenCL devices are available
level_zero:gpu	Only GPU devices on the Level Zero platform are available.
“opencl:gpu;level_zero:gpu"	GPU devices from both Level Zero and OpenCL are available. Note that escaping (like quotation marks) will likely be needed when using semi-colon separated entries.
opencl:gpu,cpu	Only CPU and GPU devices on the OpenCL platform are available.
opencl:0	Only the device with index 0 on the OpenCL backend is available.
hip:0,2	Only devices with indices of 0 and 2 from the HIP backend are available.

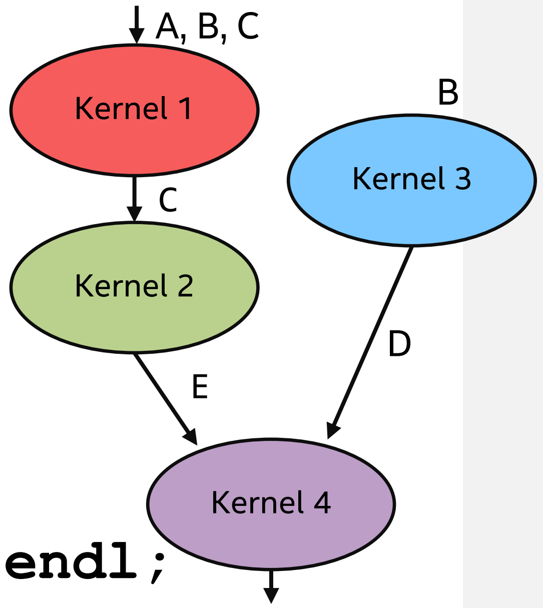
SYCL Basics

```
std::vector<float> A(1024, 1.0f), B(1024, 2.0f), 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;

 }
```

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



SYCL Basics

```
std::vector<float> A(1024, 1.0f), B(1024, 2.0f), 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];
           });                      range<1>{1024}  id<1>
       });
     }
 for (int i = 0; i < 1024; i++)

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

 }
```

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

SYCL 1.2.1 vs SYCL 2020

```
std::vector<float> A(1024, 1.0f), B(1024, 2.0f), C(1024);
{
  buffer<float> bufA {A.data(), A.size()};
  buffer<float> bufB{B.data(), B.size()};
  buffer<float> bufC {C.data(), C.size()};

  queue q;
  q.submit([&](handler &h) {
    auto A = bufA.get_access<access::mode::read>(h);
    auto B = bufB.get_access<access::mode::read>(h);
    auto C = bufC.get_access<access::mode::write>(h);
    h.parallel_for <class vector_add>(range<1>{1024}, [=](id<1> i){
               C[i] = A[i] + B[i];
           });
       });
 }
 for (int i = 0; i < 1024; i++)
         std::cout << "C[" << i << "] = " << C[i] << std::endl;
```

Basic Parallel Kernels

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

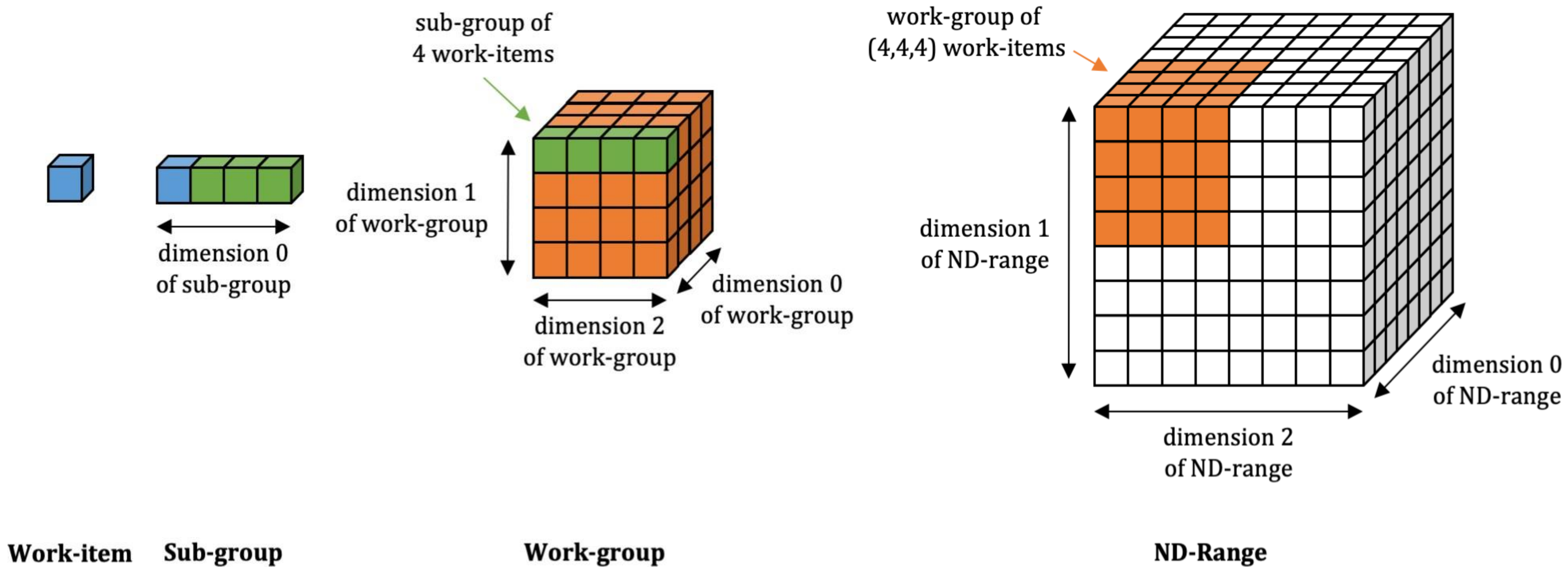
- range class is used to describe the iteration space of parallel execution
- id class is used to index an individual instance of a kernel in a parallel execution
- item class represents an individual instance of a kernel function, exposes additional functions to query properties of the execution range

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

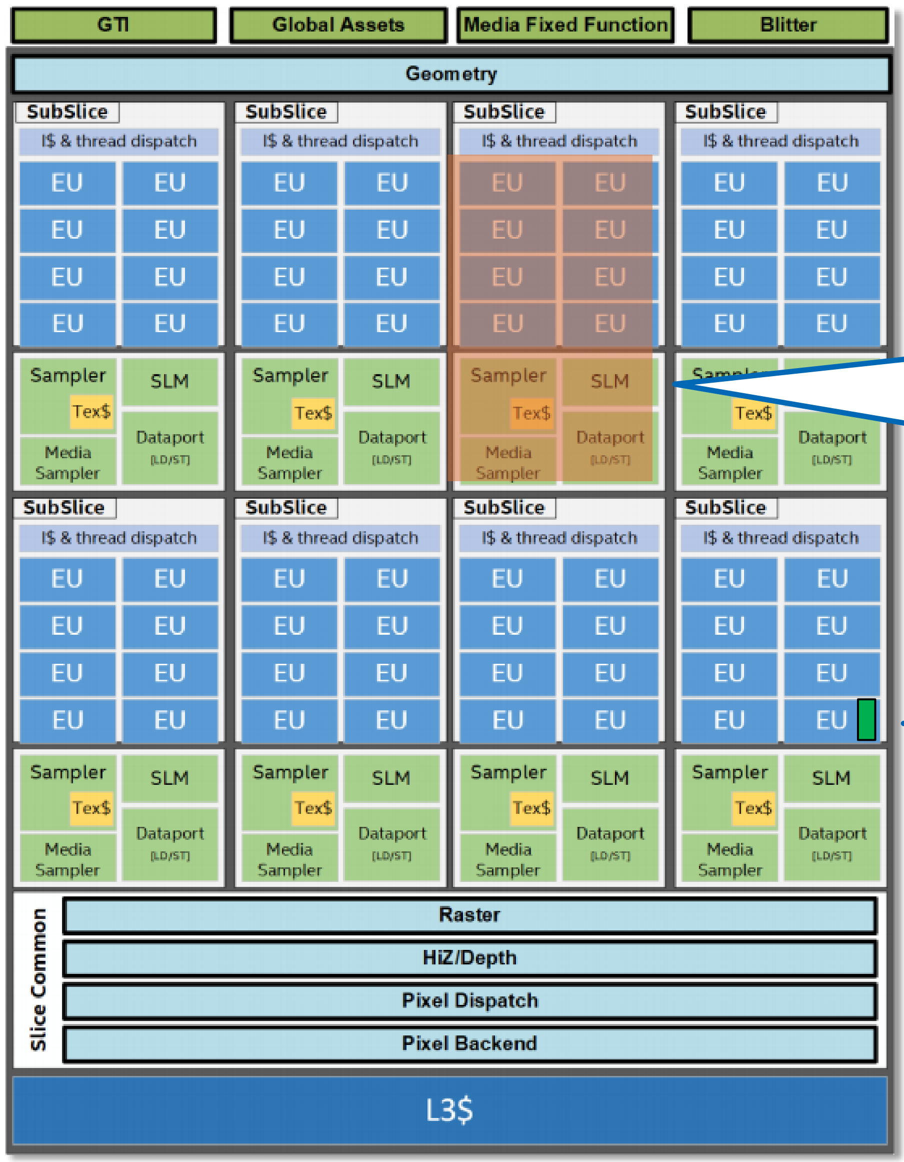
```
h.parallel_for(range<1>(1024), [=](item<1> item){
           auto idx = item.get_id();
           auto R = item.get_range();
           // CODE THAT RUNS ON DEVICE
});
```

intel

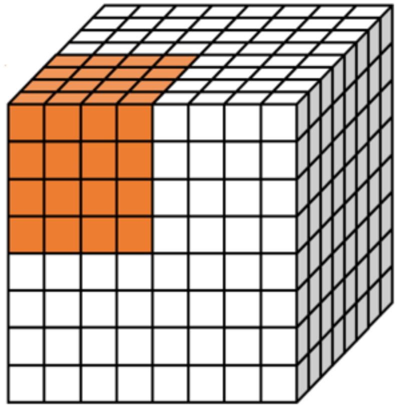
SYCL Thread Hierarchy and Mapping



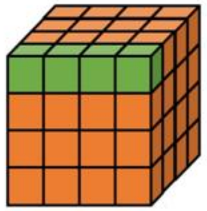
SYCL Thread Hierarchy and Mapping



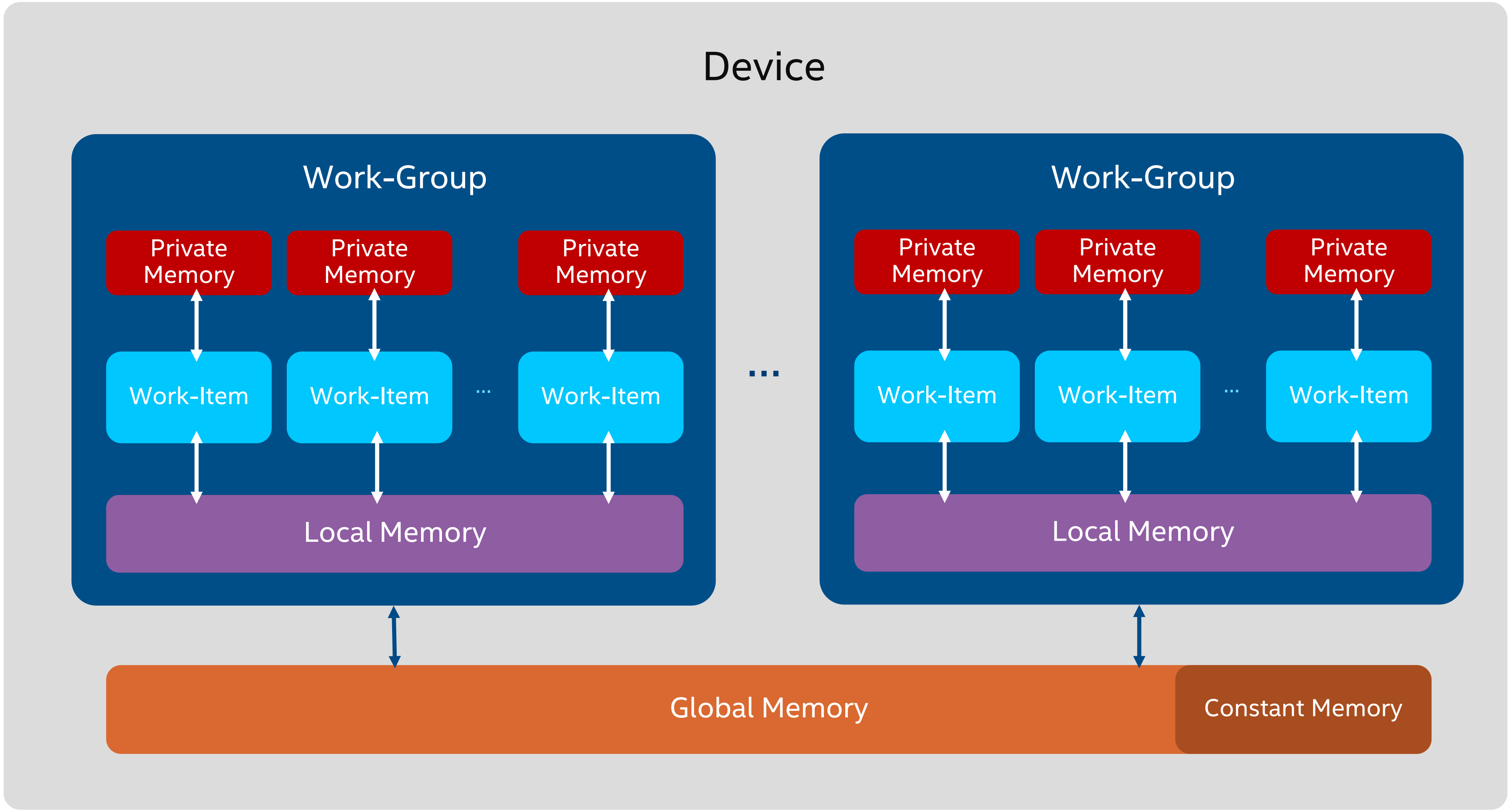
All work-items in a **work-group** are scheduled on one Compute Unit, which has its own local memory



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

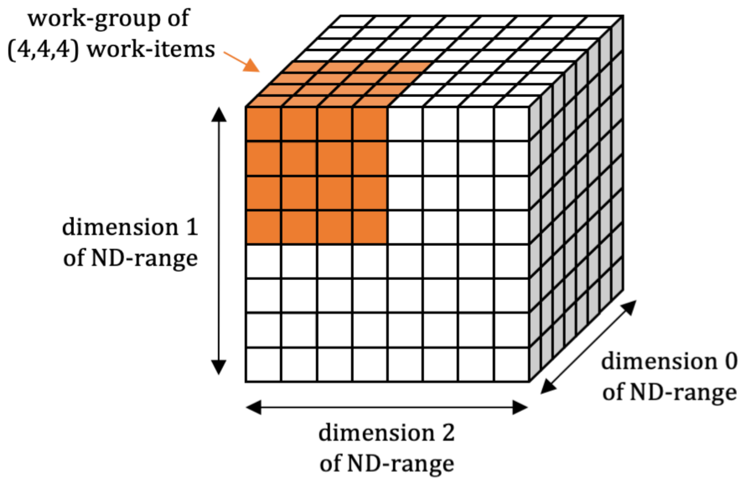


Logical Memory Hierarchy



ND-range Kernels

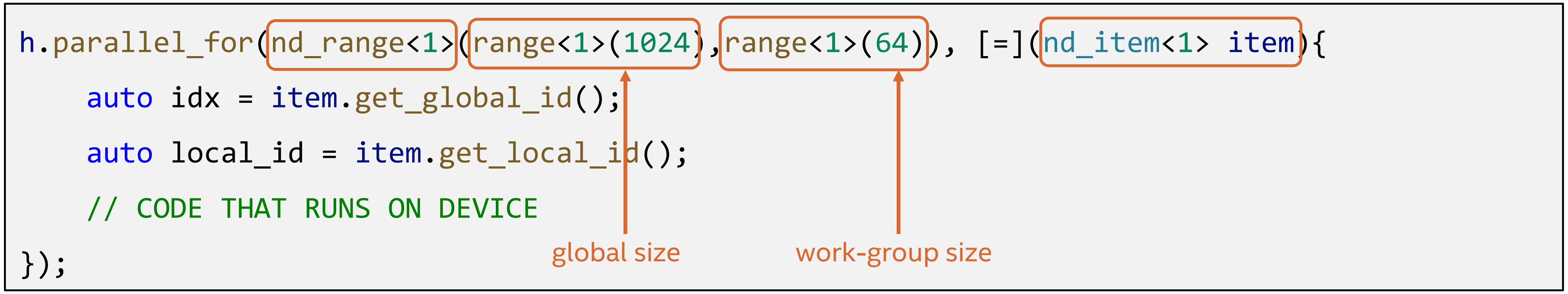
- Basic Parallel Kernels are easy way to parallelize a for-loop but does not allow performance optimization at hardware level.
- ND-range kernel is another way to express parallelism which enable low level performance tuning by providing access to local memory and mapping executions to compute units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

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, 1.0f), B(1024, 2.0f), 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.
- *Note: set SYCL_THROW_ON_BLOCK to throw an exception on attempt to wait for a blocked command.*

intel.

Buffer Destruction

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

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()

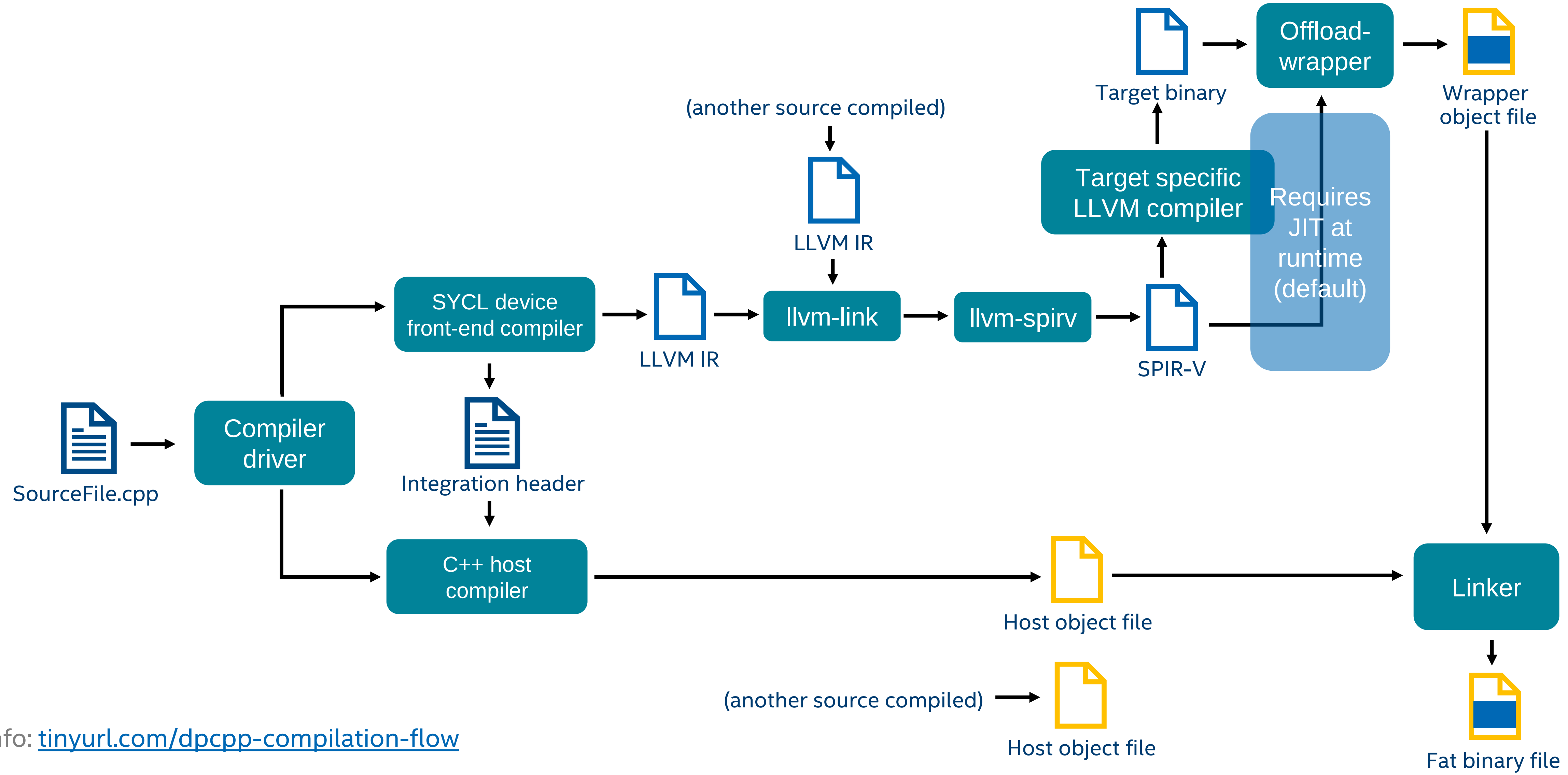
```
auto async_exception_handler = [](exception_list exceptions) {
  for (std::exception_ptr const& e : exceptions) {
    try {
      std::rethrow_exception(e);
    }
    catch (exception const& e) {
      std::cout << "Caught the Asynchronous SYCL exception"
                << e.what() << std::endl;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

Compilation and Execution Flow

SYCL Application Compilation Flow

-fsycl-targets=spir64_gen -Xs '-device pvc' / -fsycl-targets= intel_gpu_pvc
spir64_x86_64 for CPU
amdgcn-amd-amdhsa
nvptx64-nvidia-cuda
...

nvidia_gpu_sm_90
amd_gpu_gfx90a
...

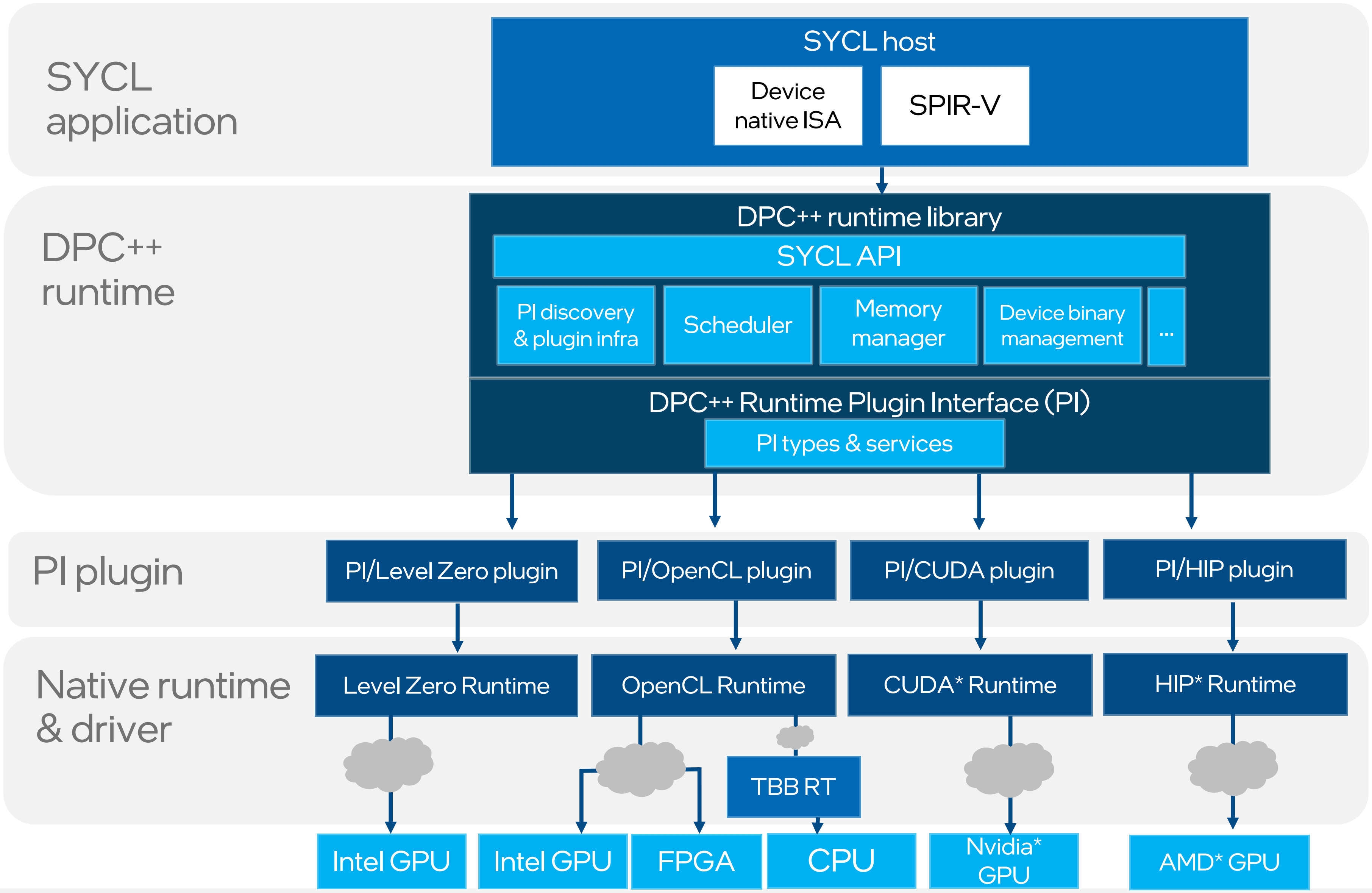


More info: tinyurl.com/dpcpp-compilation-flow

intel.github.io/llvm-docs/GetStartedGuide.html

intel.

Runtime Architecture



Controlled via [ONEAPI_DEVICE_SELECTOR]

opencl
level_zero
cuda
hip

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
```

Getting Started on DevCloud

- qsub -I -l nodes=1:gpu:ppn=2 -d .
- sycl-ls (control devices via SYCL_DEVICE_FILTER)
- Compile and run simple vecAdd code
- export SYCL_PI_TRACE=1
- export SYCL_DEVICE_FILTER=level_zero

Unified Shared Memory

Motivation

The SYCL 1.2.1 standard provides a Buffer memory abstraction

- Powerful and elegantly expresses data dependences

However…

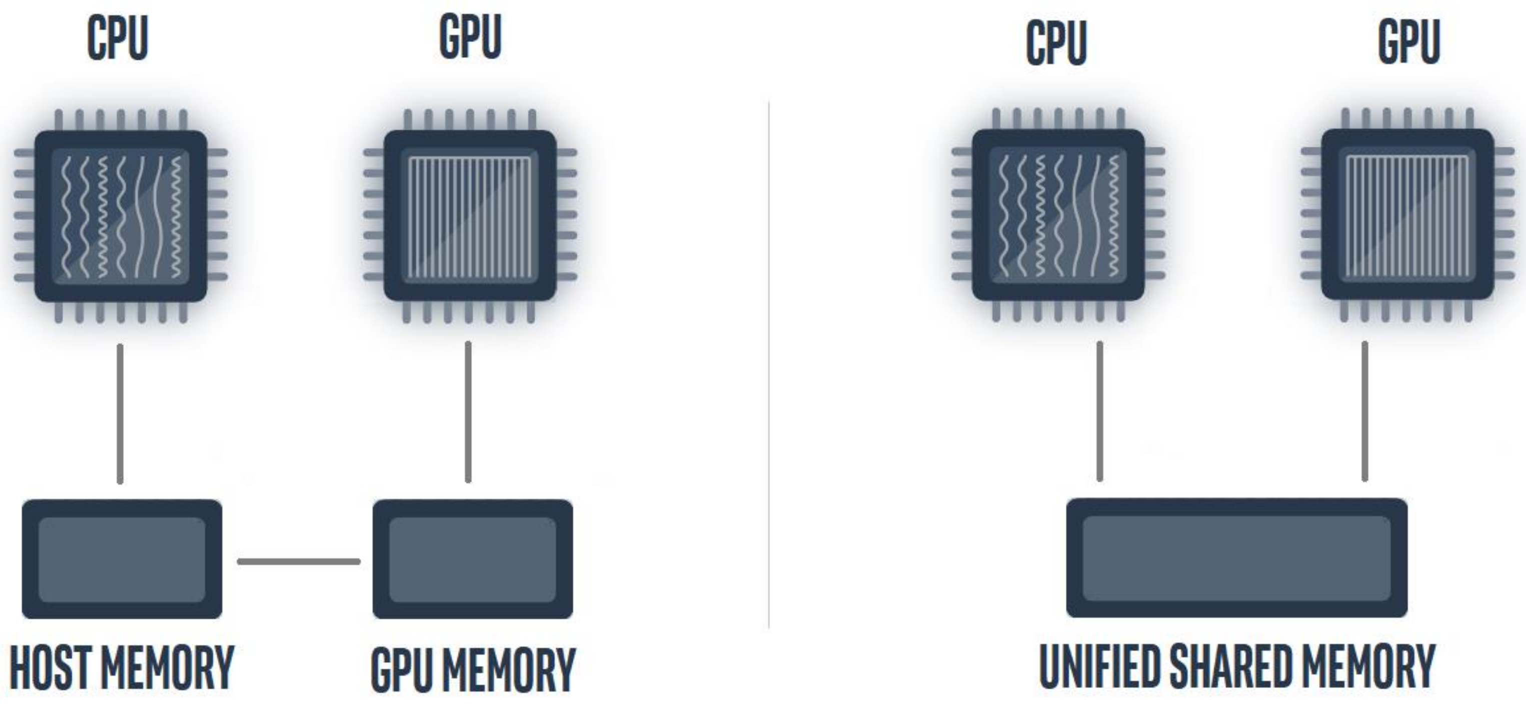
- Replacing all pointers and arrays with buffers in a C++ program can be a burden to programmers

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

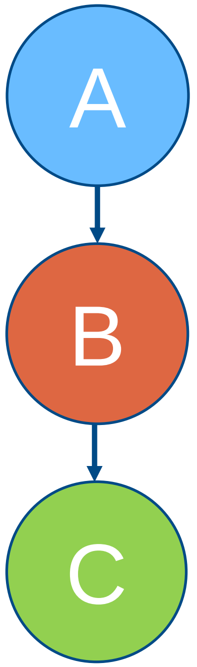
Dependences must be specified explicitly using events

- 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

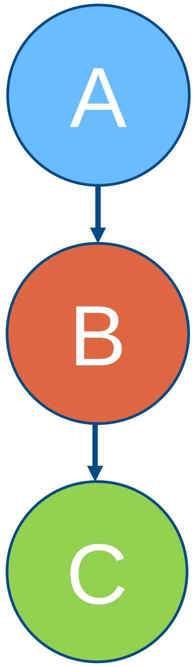


```
queue q;
int* data =  malloc_shared<int>(N, q);
for(int i=0;i<N;i++) data[i] = 10;
q.submit([&] (handler &h){
    h.parallel_for<class taskA>(range<1>(N), [=](id<1> i){
        data[i] += 2;
    });
}).wait();
q.submit([&] (handler &h){
    h.parallel_for<class taskB>(range<1>(N), [=](id<1> i){
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

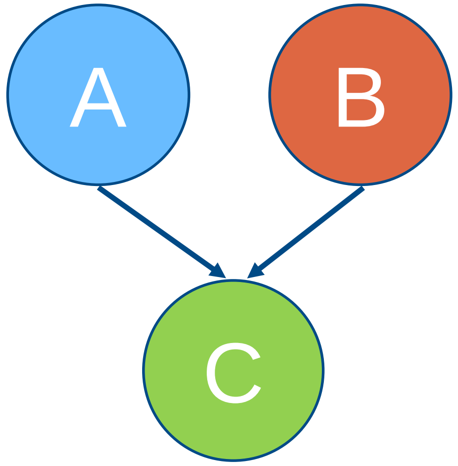
Execution will not overlap even if the queues have no data dependency



```
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

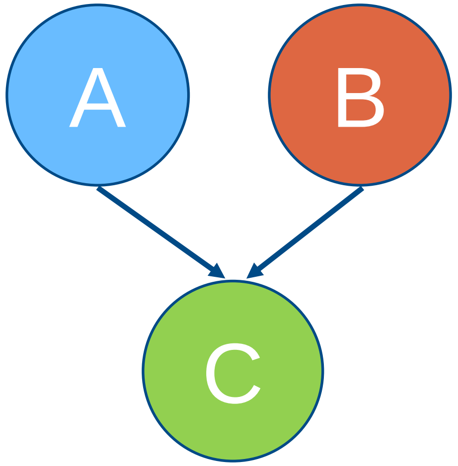


```
queue q;
int* data1 =  malloc_shared<int>(N, q);
int* data2 =  malloc_shared<int>(N, q);
for(int i=0;i<N;i++) {data1[i] = 10; data2[i] = 10;}
auto e1 = q.submit([&] (handler &h){
    h.parallel_for<class taskA>(range<1>(N), [=](id<1> i){
        data1[i] += 2;
    });
});
auto e2 = q.submit([&] (handler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



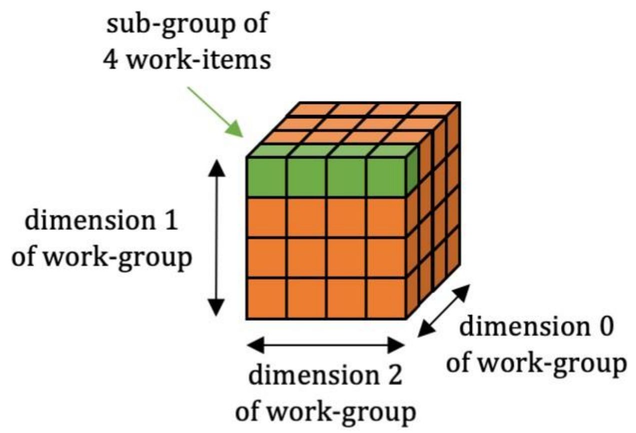
```
queue q;
int* data1 =  malloc_shared<int>(N, q);
int* data2 =  malloc_shared<int>(N, q);
for(int i=0;i<N;i++) {data1[i] = 10; data2[i] = 10;}
auto e1 = q.parallel_for <class taskA>(range<1>(N), [=](id<1> i){
  data1[i] += 2;
});
auto e2 = q.parallel_for <class taskB>(range<1>(N), [=](id<1> i){
 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

Sub-groups

Sub-groups

- A subset of work-items within a work-group that may map to vector hardware.
- Why use Sub-groups?
  - Work-items in a sub-group can communicate directly using shuffle operations, without explicit memory operations.
  - Work-items in a sub-group can synchronize using sub-group barriers and guarantee memory consistency using sub-group memory fences.
  - Work-items in a sub-group have access to sub-group collectives, providing fast implementations of common parallel patterns.


sub-group of 4 work-items
dimension 1 of work-group
dimension 0 of work-group
dimension 2 of work-group

Work-group

Sub-groups

sub_group class

- The sub-group handle can be obtained from the nd_item using the get_sub_group()

```
h.parallel_for(nd_range<1>(N,B), [=](nd_item<1> item)
{
          auto sg = item.get_sub_group();

          // KERNEL CODE
});
```

- Once you have the sub-group handle, you can query for more information about the sub-group, do shuffle operations or use collective functions.

- Explicit kernel attribute [[intel::reqd_sub_group_size(N)]] to control the sub-group size

Sub-groups

The sub-group handle can be quired to get other information:

- get_local_id() returns the index of the work-item within its sub-group
- get_local_range() returns the size of sub_group
- get_group_id() returns the index of the sub-group
- get_group_range() returns the number of sub-groups within the parent work-group

```
h.parallel_for(nd_range<1>(N,B), [=](nd_item<1> item){
         auto sg = item.get_sub_group();

        if(sg.get_local_id() == 0){
            out << "sub_group id: " << sg.get_group_id()[0]
                << " of " << sg.get_group_range()
                << ", size=" << sg.get_local_range()[0]
                                      << endl;
        }
});
```

```
sub_group id: 1 of 4, size=16
sub_group id: 3 of 4, size=16
sub_group id: 2 of 4, size=16
sub_group id: 0 of 4, size=16
```

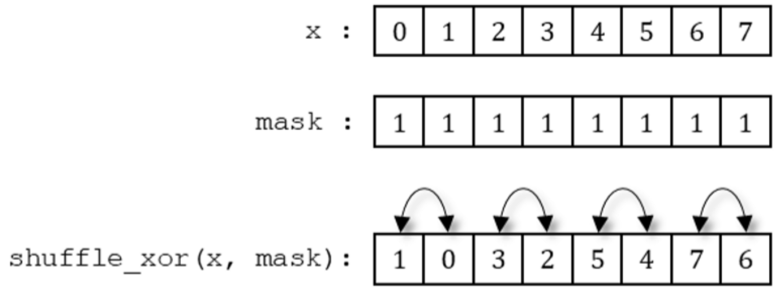
intel.

Sub-Group Shuffles

- One of the most useful features of sub-groups is the ability to communicate directly between individual work-items without explicit memory operations.
- Shuffle operations enable us to remove work-group local memory usage from our kernels and/or to avoid unnecessary repeated accesses to global memory.

```
h.parallel_for(nd_range<1>(N,B), [=](nd_item<1> item){
        auto  sg = item.get_sub_group();
        size_t i = item.get_global_id(0);

        /* Shuffles */
        //data[i] = sg.shuffle(data[i], 2);
        //data[i] = sg.shuffle_up(0, data[i], 1);
        //data[i] = sg.shuffle_down(data[i], 0, 1);
        data[i] = sg.shuffle_xor(data[i], 1);
});
```



Sub-Group Collectives

- The collective functions provide implementations of closely-related common parallel patterns.
- Providing these implementations as library functions increases developer productivity and gives implementations the ability to generate highly optimized code for individual target devices.

```
h.parallel_for(nd_range<1>(N,B), [=](nd_item<1> item){
        auto sg = item.get_sub_group();
        size_t i = item.get_global_id(0);

        /* Collectives */
        data[i] = reduce(sg, data[i],  plus<>());
        //data[i] = reduce(sg, data[i], std::maximum<>());
        //data[i] = reduce(sg, data[i], std::minimum<>());
});
```

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
SYCL Academy	github.com/codeplaysoftware/syclacademy/tree/main
Code samples:	github.com/intel/llvm/tree/sycl/sycl/test github.com/intel/llvm/tree/sycl/sycl/test-e2e github.com/oneapi-src/oneAPI-samples

Hands-on Exercises

Essentials of oneAPI and SYCL

Introduction

Module 1 - Introduction to oneAPI and SYCL

Module 2 - SYCL Program Structure

Module 3 - SYCL Unified Shared Memory

Advanced

Module 4 - SYCL Sub-Groups

Module 9 - SYCL Buffers and Accessors in depth

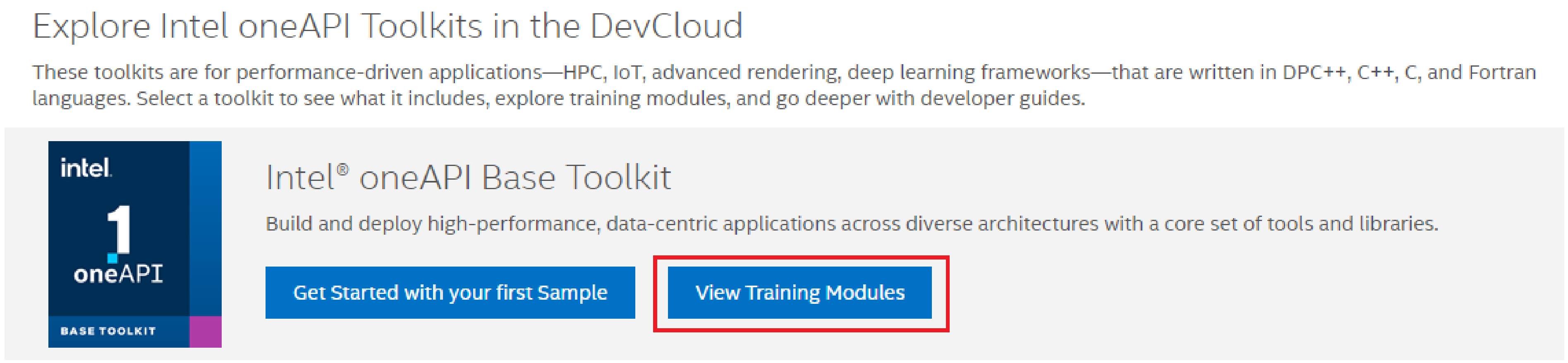
Module 10 - SYCL Task Scheduling and Data Dependences

Jupyter Notebook* Lab

Getting Started with Intel DevCloud

https://devcloud.intel.com/oneapi/get_started/

1)



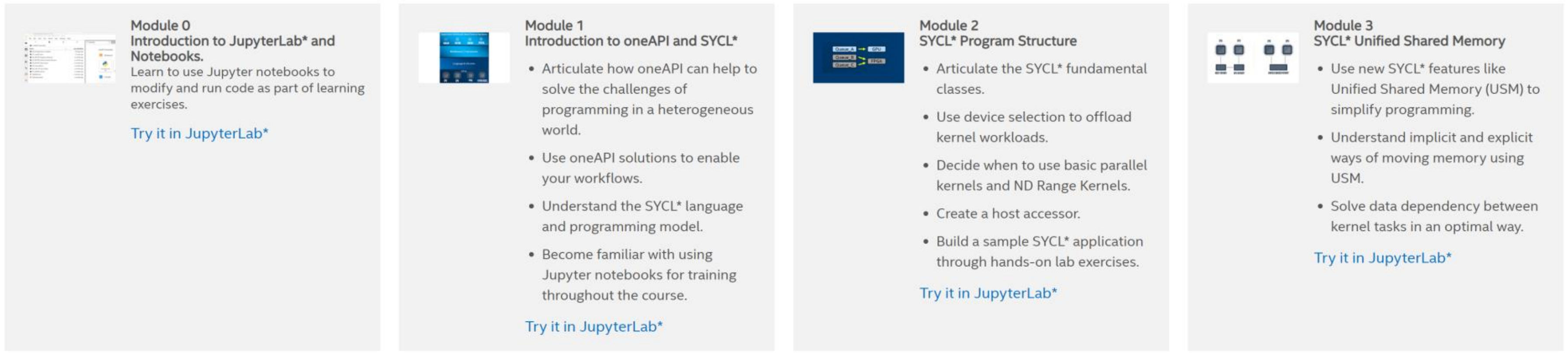
Explore Intel oneAPI Toolkits in the DevCloud

These toolkits are for performance-driven applications—HPC, IoT, advanced rendering, deep learning frameworks—that are written in DPC++, C++, C, and Fortran languages. Select a toolkit to see what it includes, explore training modules, and go deeper with developer guides.

Intel® oneAPI Base Toolkit

Build and deploy high-performance, data-centric applications across diverse architectures with a core set of tools and libraries.

2)



Module 0
Introduction to JupyterLab* and Notebooks.
Learn to use Jupyter notebooks to modify and run code as part of learning exercises.

Try it in JupyterLab*

Module 1
Introduction to oneAPI and SYCL*

- Articulate how oneAPI can help to solve the challenges of programming in a heterogeneous world.
- Use oneAPI solutions to enable your workflows.
- Understand the SYCL* language and programming model.
- Become familiar with using Jupyter notebooks for training throughout the course.

Try it in JupyterLab*

Module 2
SYCL* Program Structure

- Articulate the SYCL* fundamental classes.
- Use device selection to offload kernel workloads.
- Decide when to use basic parallel kernels and ND Range Kernels.
- Create a host accessor.
- Build a sample SYCL* application through hands-on lab exercises.

Try it in JupyterLab*

Module 3
SYCL* Unified Shared Memory

- Use new SYCL* features like Unified Shared Memory (USM) to simplify programming.
- Understand implicit and explicit ways of moving memory using USM.
- Solve data dependency between kernel tasks in an optimal way.

Try it in JupyterLab*

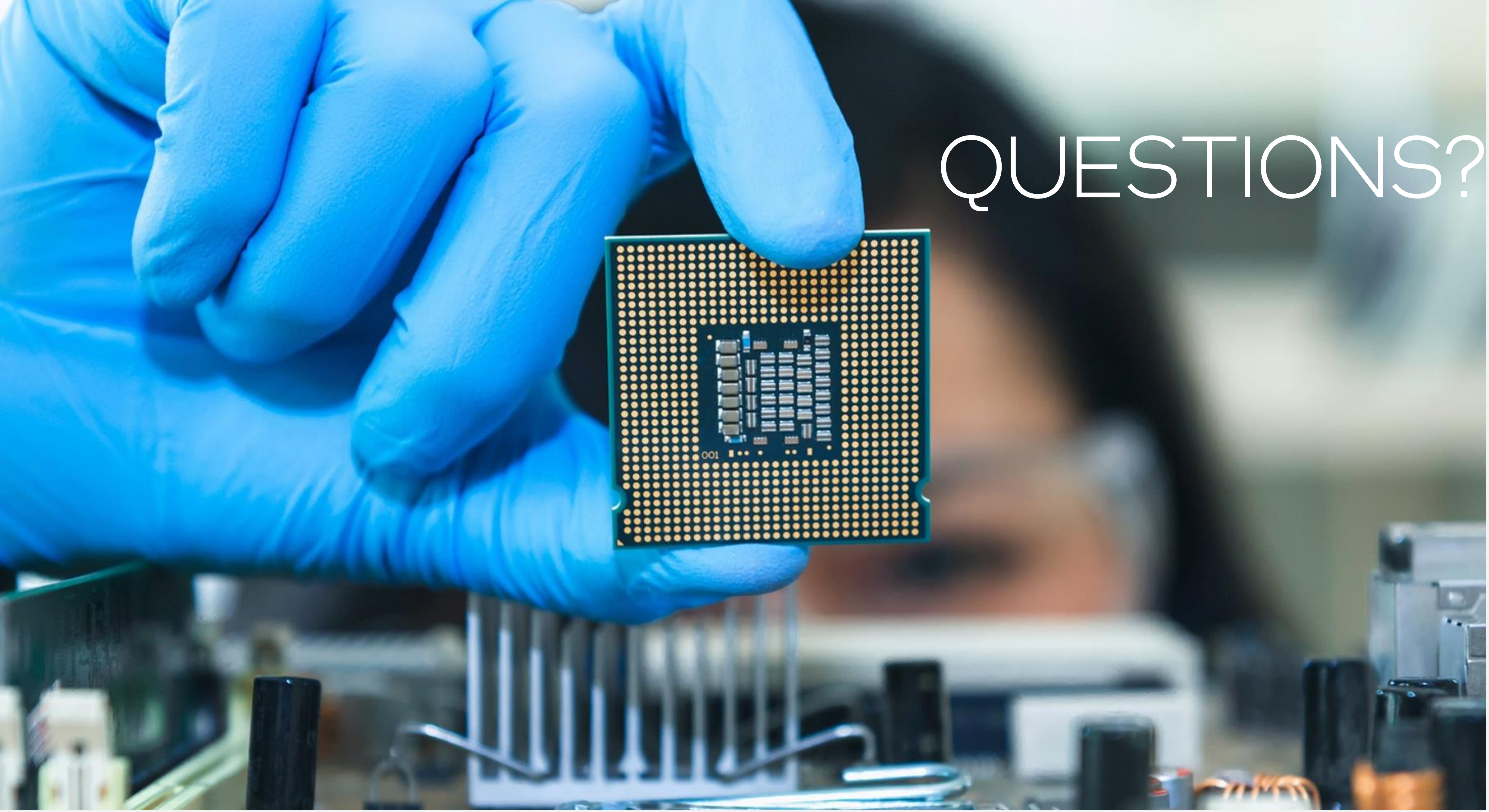
Jupyter Notebook* Lab

Allocate the compute node in interactive mode :

qsub -I -l nodes=1:gpu:ppn=2 -d .

SYCL Academy: github.com/codeplaysoftware/syclacademy/tree/main

- Many branches available there: main, iwocl23, isc23, etc.



QUESTIONS?

Notices & Disclaimers