oneAPI

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

Introduction to heterogenous programming with Data Parallel C++

October 2022

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

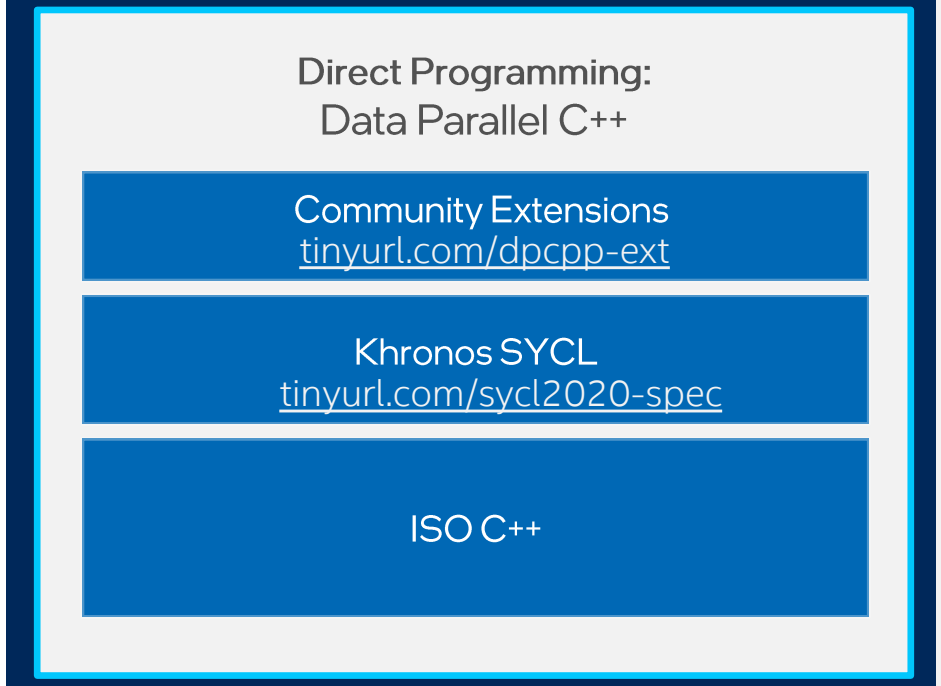
Today's DPC++ compiler is a mix of SYCL 1.2.1, SYCL 2020, and Language Extensions

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

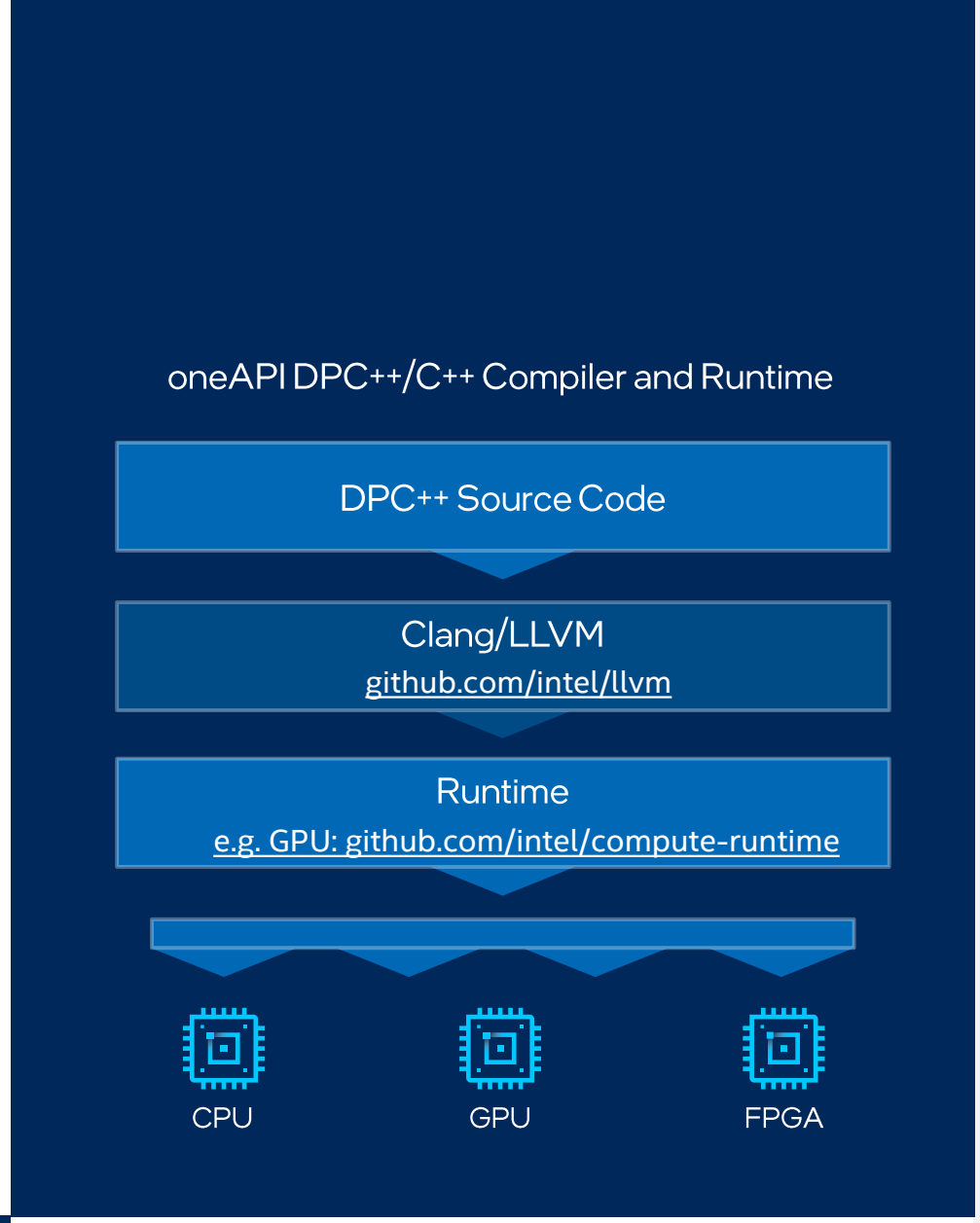


Intel® oneAPI DPC++/C++ Compiler

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

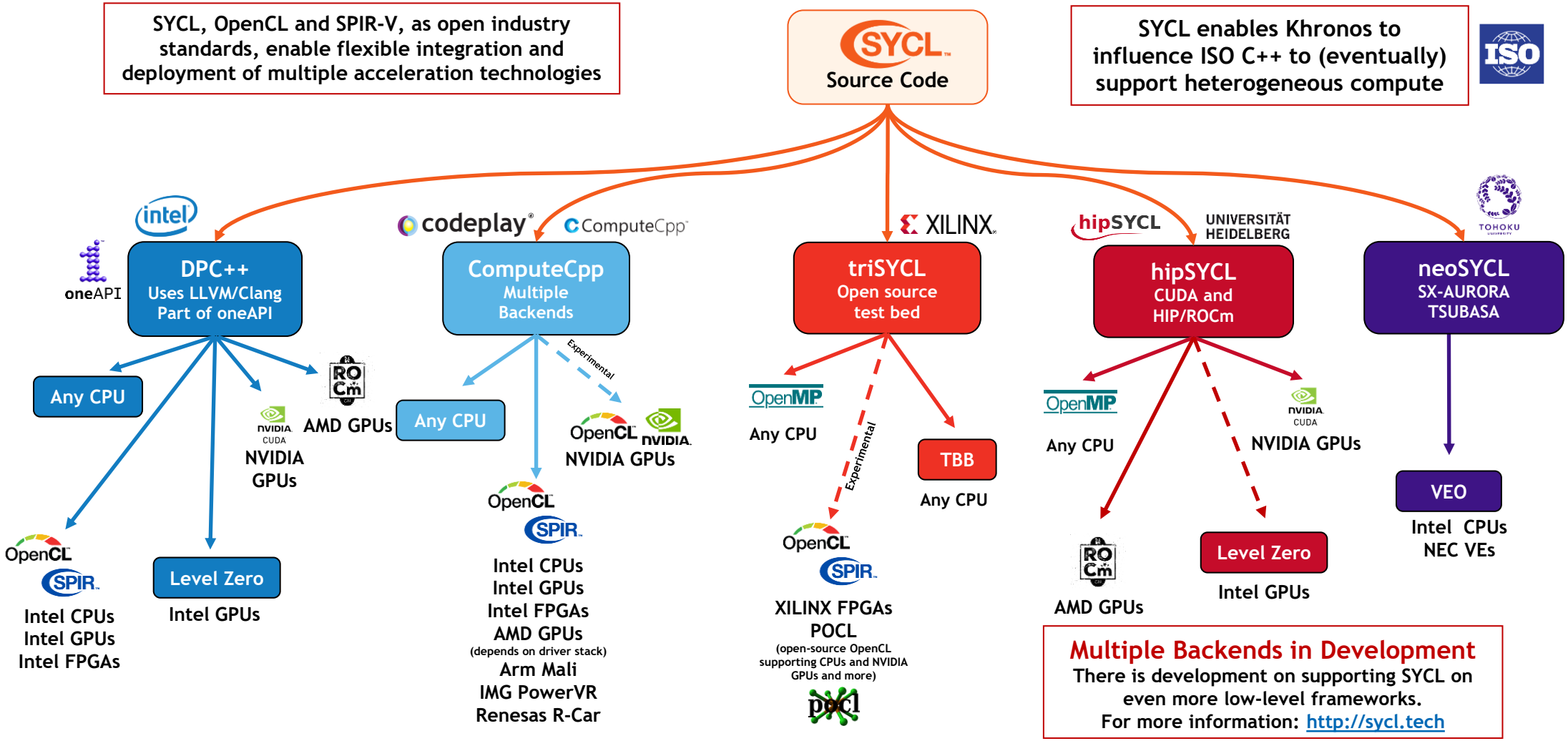


Code samples:

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

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

SYCL ecosystem is growing



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

+ Celerity: SYCL on MPI+SYCL

*This slide is prepared by The Khronos Group Inc

Codeplay Launched

Data Parallel C++ Compiler for Nvidia GPUs

- Developers can retarget and reuse code between NVIDIA and Intel compute accelerators from a single source base
- Codeplay is the first oneAPI industry contributor to implement a developer tool based on oneAPI specifications
- They leveraged the DPC++ LLVM-based open source project that Intel established
- Codeplay is a key driver of the Khronos SYCL standard, upon with DPC++ is based
- More details in the Codeplay blog post
- Build DPC++ toolchain with support for NVIDIA CUDA: tinyurl.com/dpcpp-cuda-be
  tinyurl.com/dpcpp-cuda-be-webinar

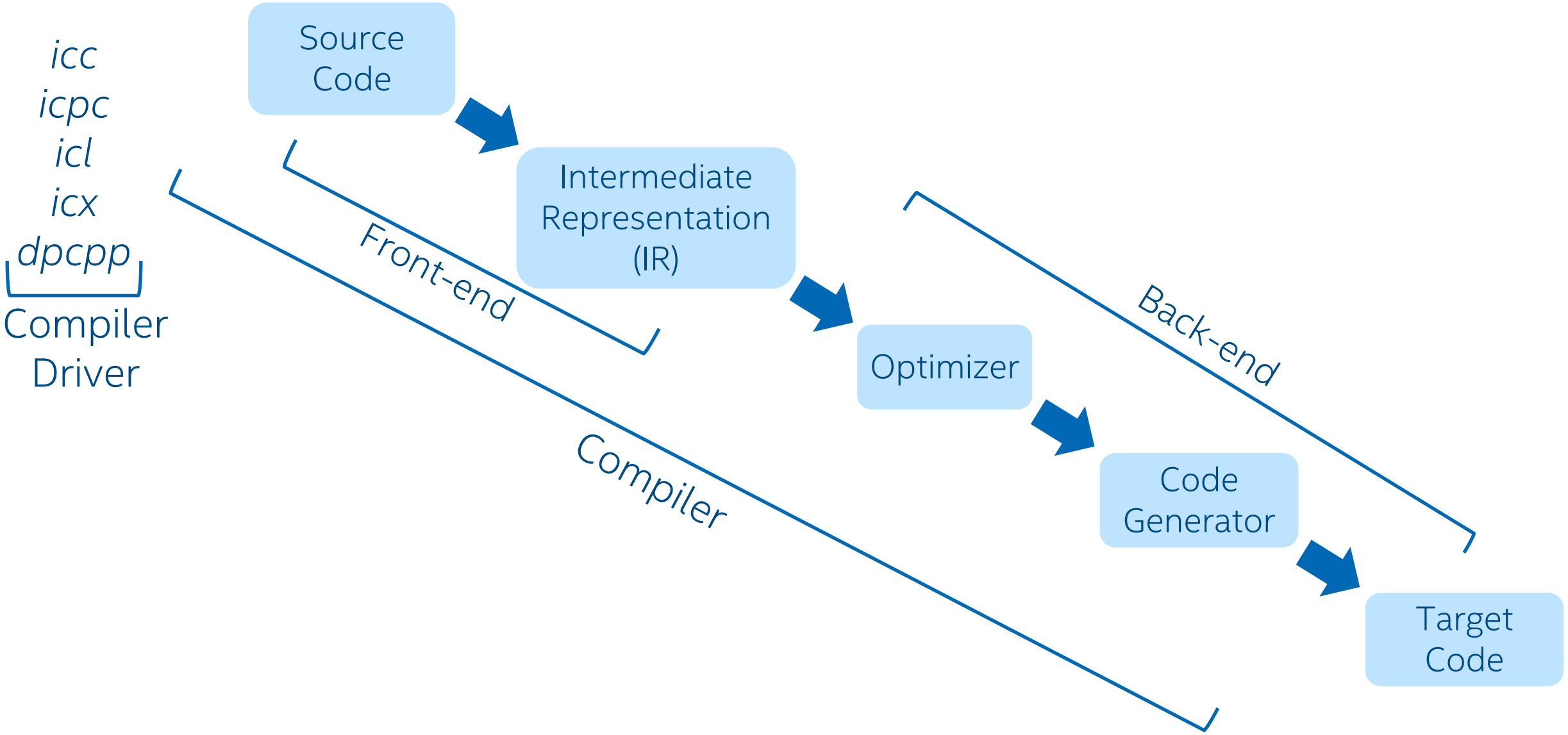
Other News

Codeplay Brings NVIDIA GPU Support to Industry-Standard Math Library

Intel Open Sources the oneAPI Math Kernel Library Interface

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, IL0 ***icc/icpc/icl - deprecated***	CPU	Yes	No	HPC
Intel® Fortran Compiler Classic, IL0 ***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

Packaging of C++ Compilers

- oneAPI Base Toolkit *PLUS* oneAPI HPC Toolkit

  Existing IL0 compilers ICC, ICPC in HPC Toolkit

  v2021.7 code base for IL0 compilers

  Compilers based on LLVM* framework

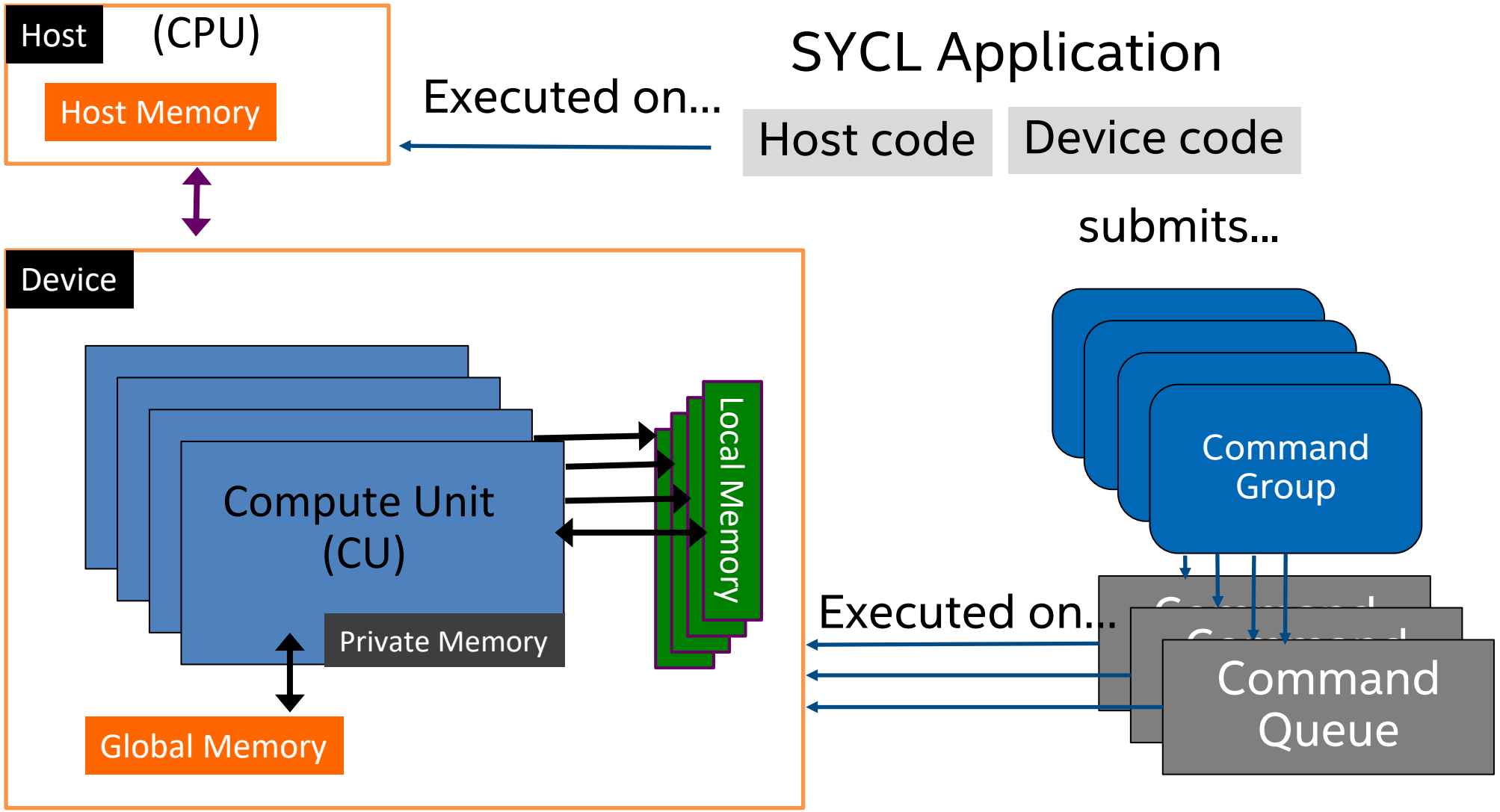
  Compiler Drivers: icx/icpx and dpcpp

  v2022.2 in oneAPI 2022.3

- Prerequisites: Set Up Your System for Intel GPU

“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
Executed on…
Command
Group
Command
Queue

Anatomy of a SYCL Application

```
#include <CL/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 <CL/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

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{}`

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

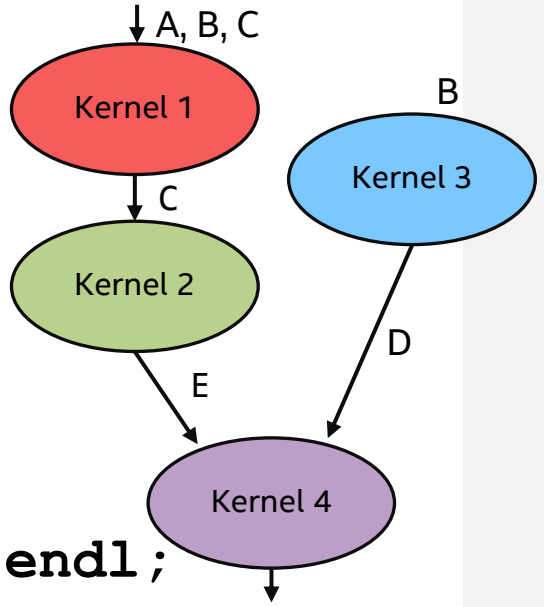
export SYCL_DEVICE_FILTER={backend:device_type:device_num}

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

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];
            });
        });
      }
 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

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

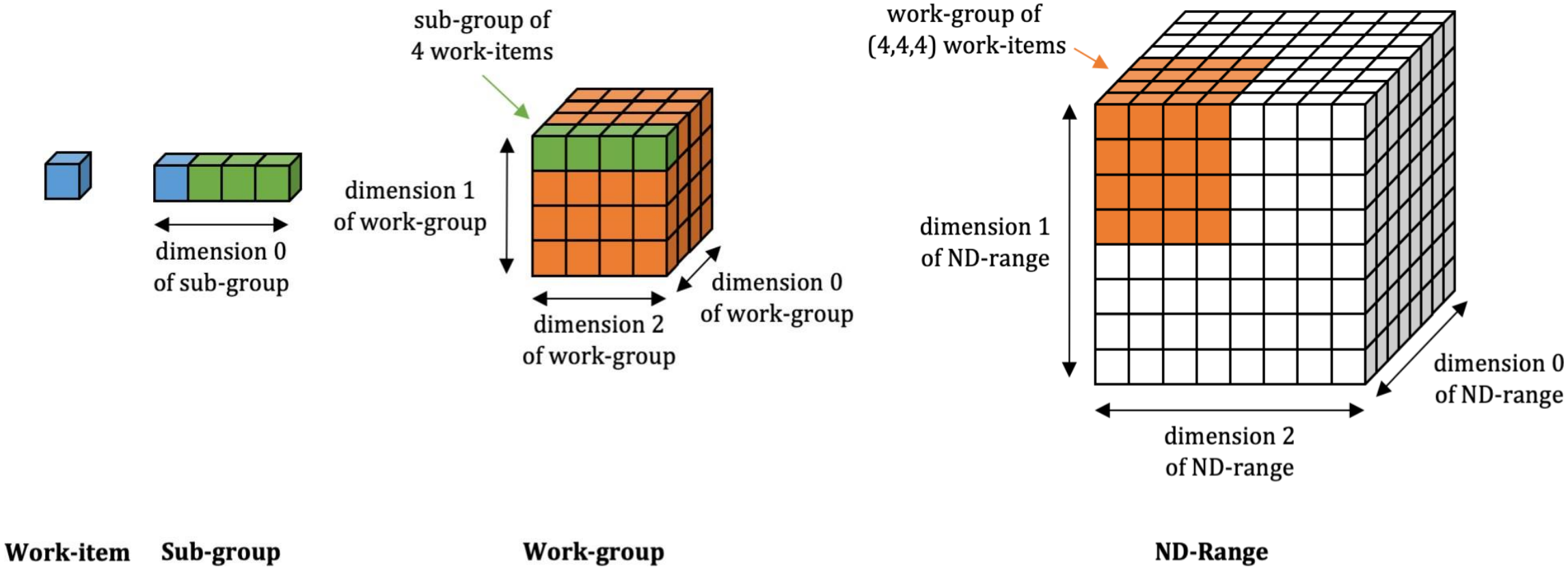
The functionality of basic parallel kernels is exposed via range, id and item classes

- 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
});
```

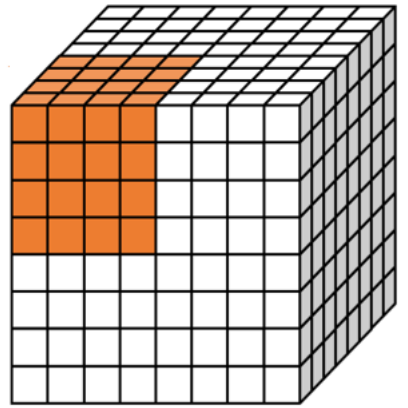
```
h.parallel_for(range<1>(1024), [=](item<1> item){
            auto idx = item.get_id();
            auto R = item.get_range();
            // CODE THAT RUNS ON DEVICE
});
```

SYCL Thread Hierarchy and Mapping

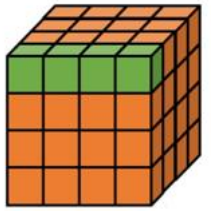


SYCL Thread Hierarchy and Mapping

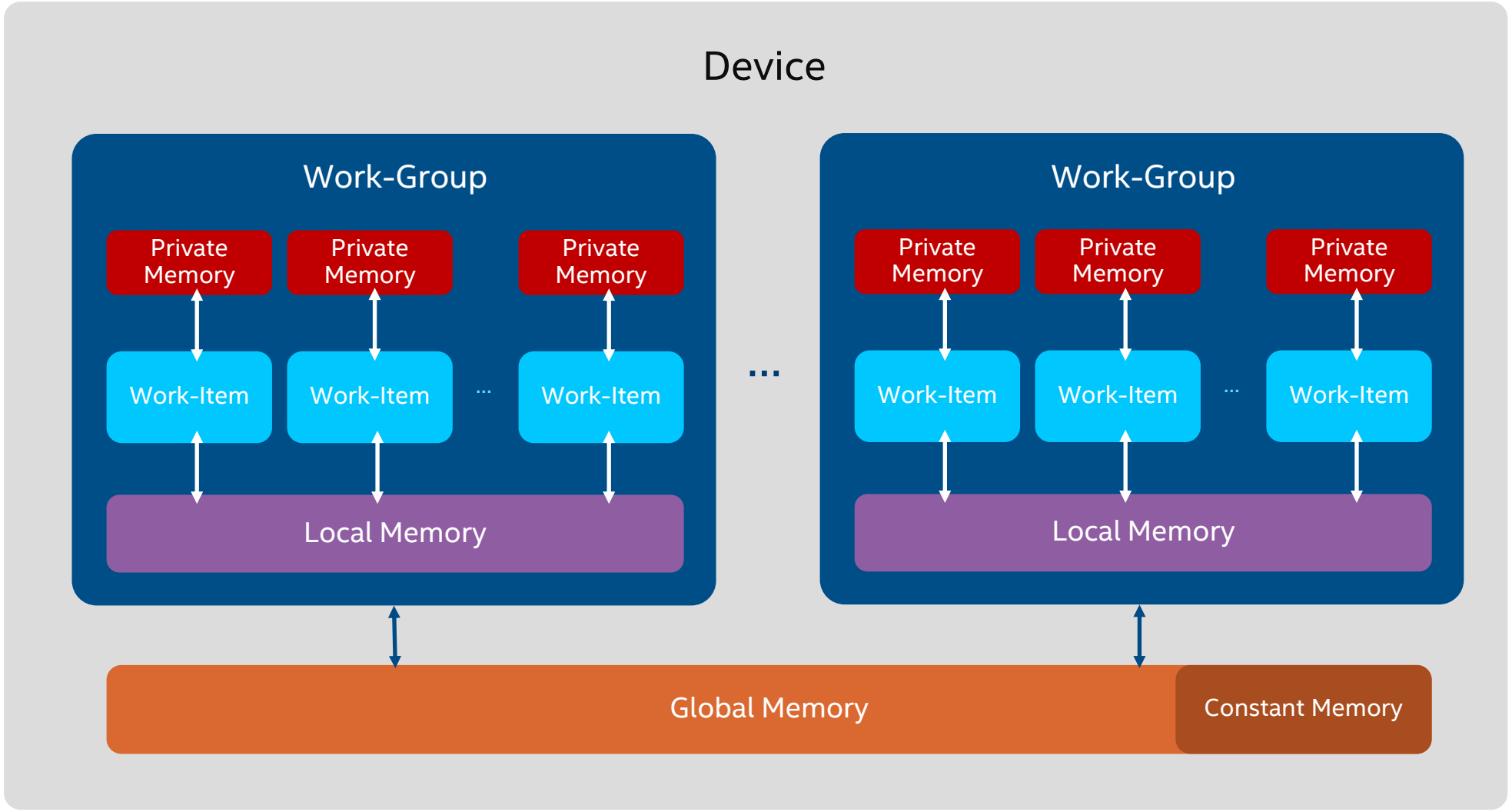
All work-items in a **work-group** are scheduled on one Compute Unit, which has its own local memory



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

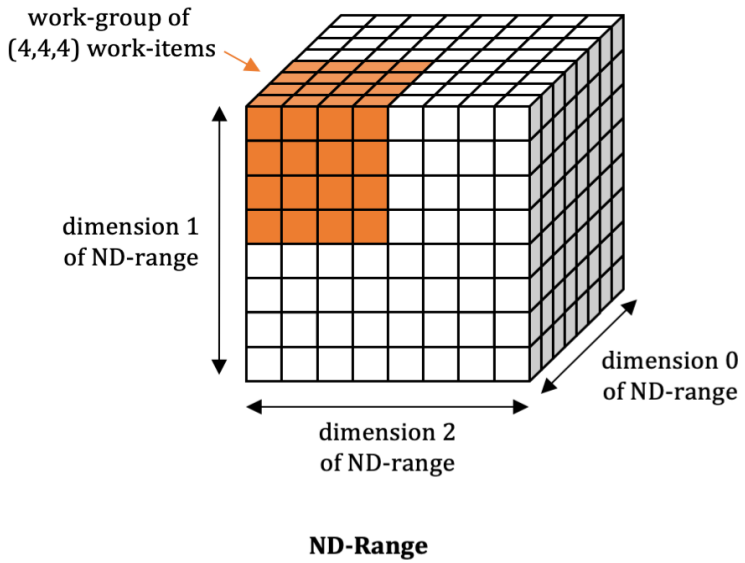


Logical Memory Hierarchy



ND-range Kernels

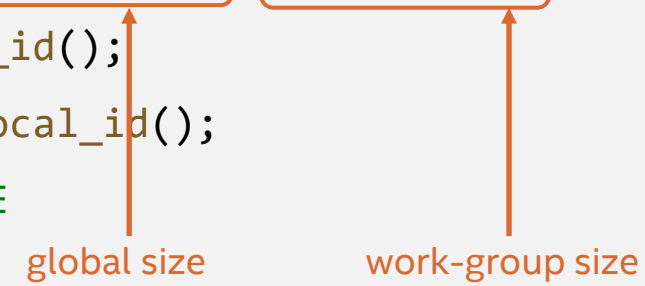
- Basic Parallel Kernels are easy way to parallelize a for-loop but does not allow performance optimization at hardware level.
- ND-range kernel is another way to express parallelism which enable low level performance tuning by providing access to local memory and mapping executions to compute units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

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

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;
    }
  }
};
```

Demo/Lab

Data Parallel C++ Essentials Modules

Module 1 - Introduction to oneAPI and DPC++
Module 2 - DPC++ Program Structure

Jupyter Notebook* Lab

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

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.

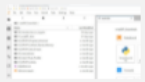
1)

Intel® oneAPI Base Toolkit

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

Get Started with your first Sample | View Training Modules

2) **Learn the Essentials of Data Parallel C++**



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

Sign in to try it in Jupyter

Module 1
Introduction to DPC++

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

Sign in to try it in Jupyter

Module 2
DPC++ 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 DPC++ application through hands-on lab exercises.

Sign in to try it in Jupyter

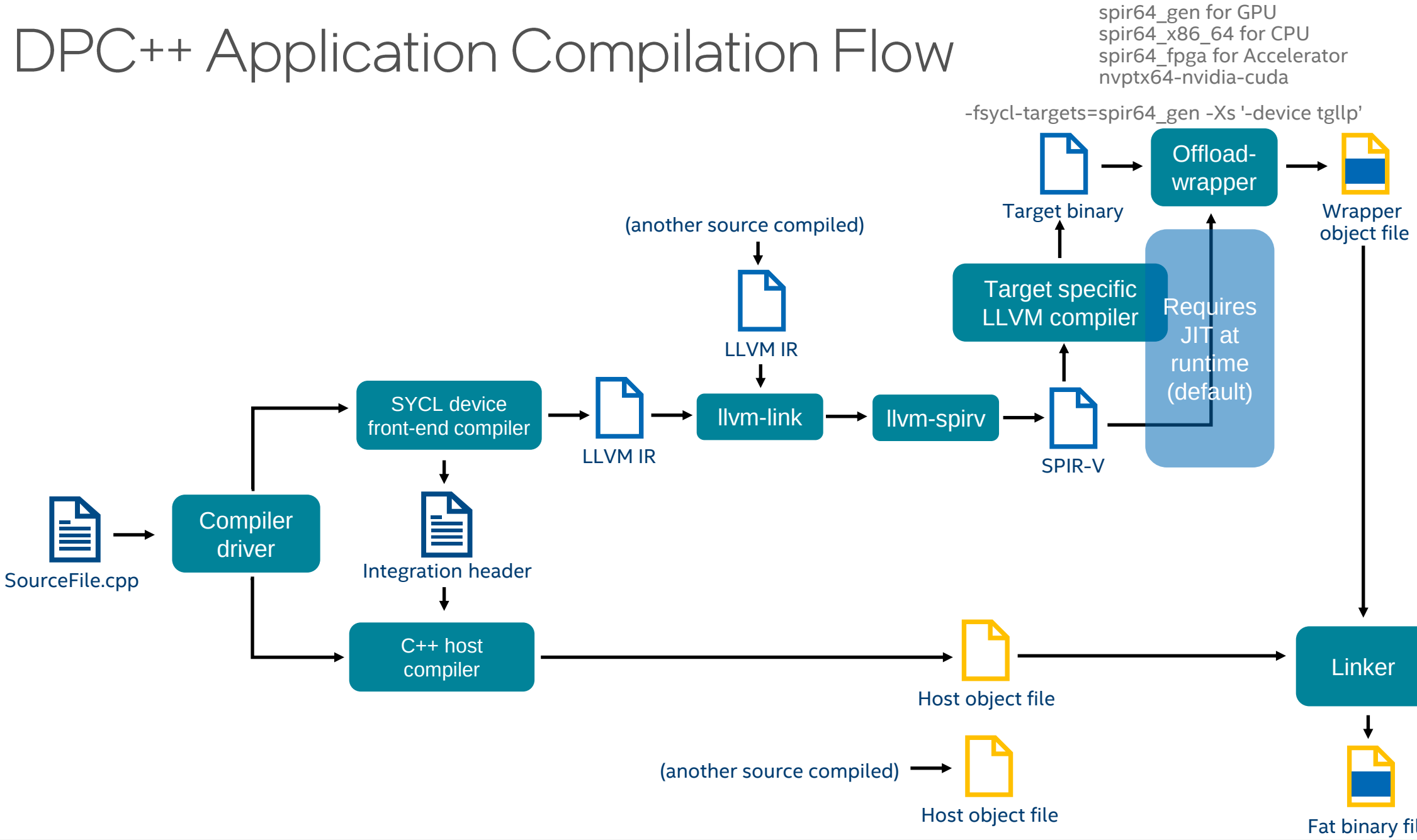
Module 3
DPC++ Unified Shared Memory

- Use new DPC++ 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.

Sign in to try it in Jupyter

Compilation and Execution Flow

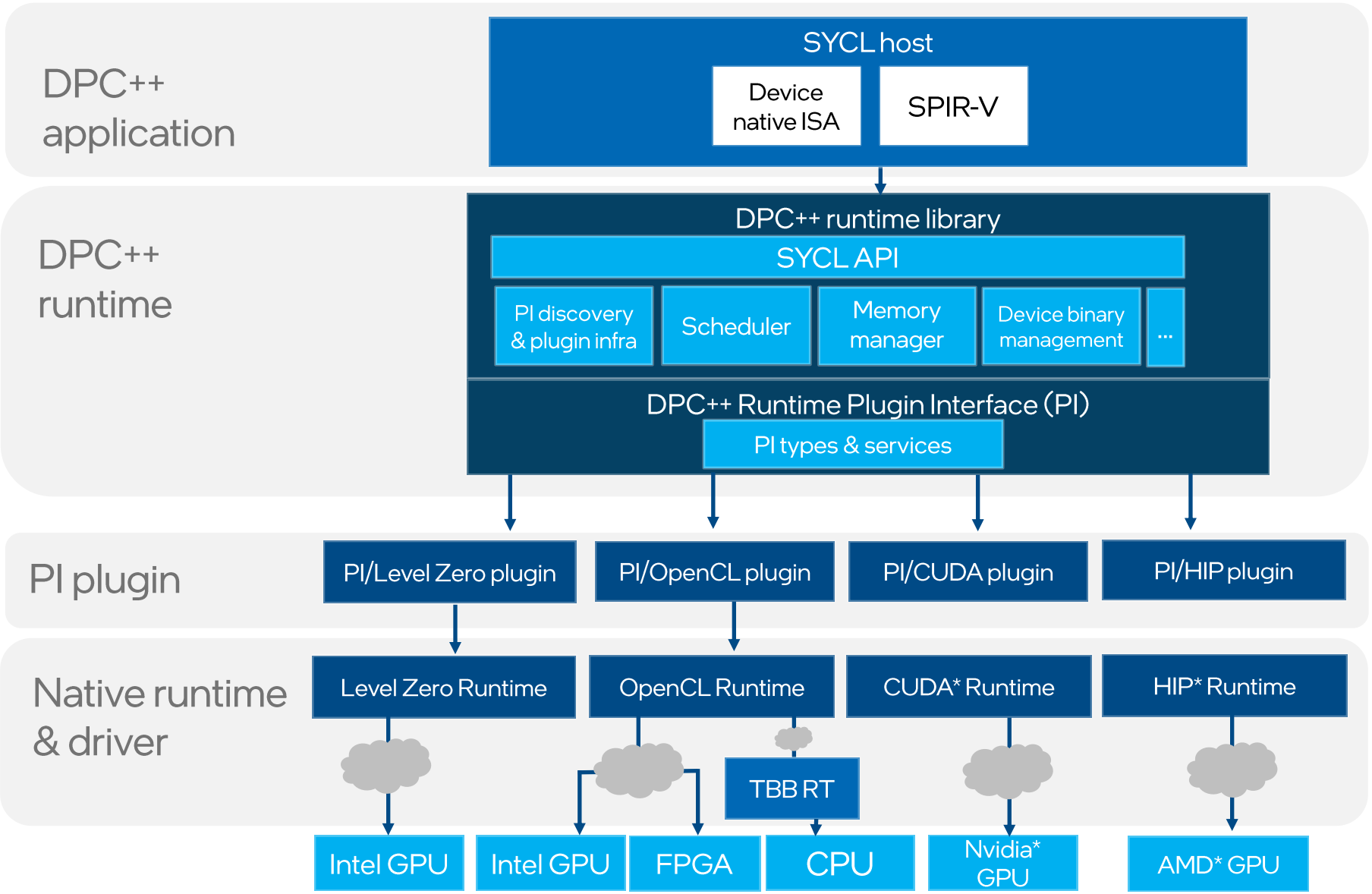
DPC++ Application Compilation Flow



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

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

Runtime Architecture



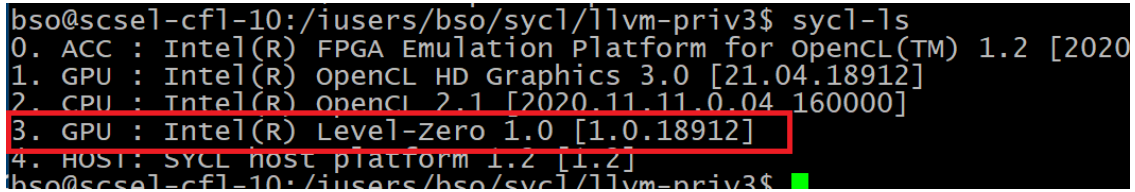
Controlled via SYCL_DEVICE_FILTER

- opencl
- level_zero
- cuda
- hip

More info: tinyurl.com/dpcpp-pi

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



  - 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
```

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…

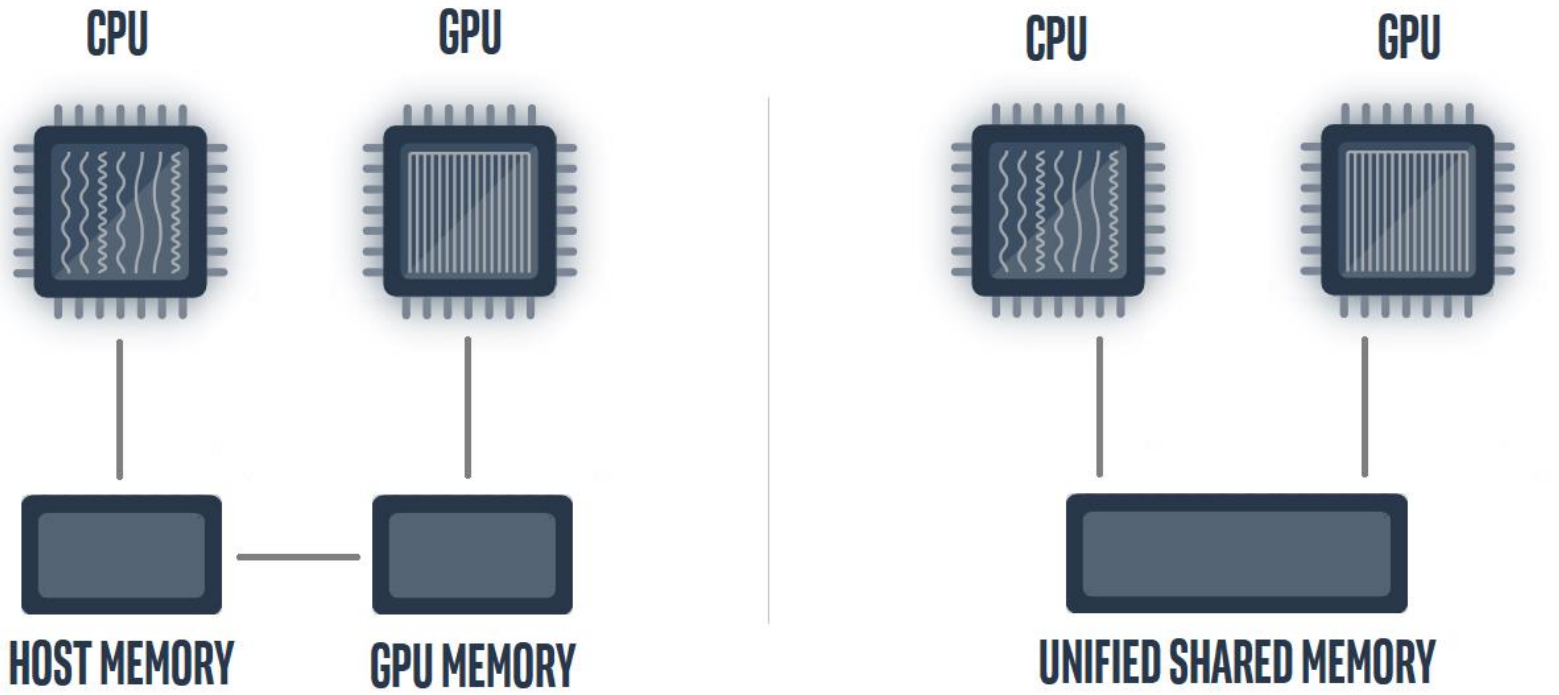
- Replacing all pointers and arrays with buffers in a C++ program can be a burden to programmers

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

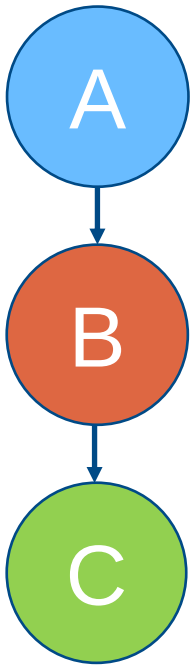
Dependences must be specified explicitly using events

- 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

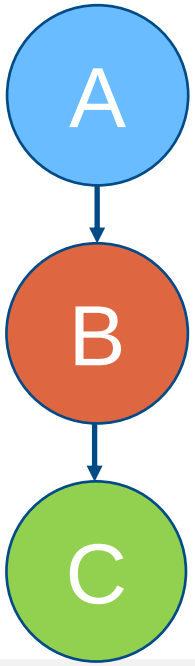
wait() will block execution on host




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

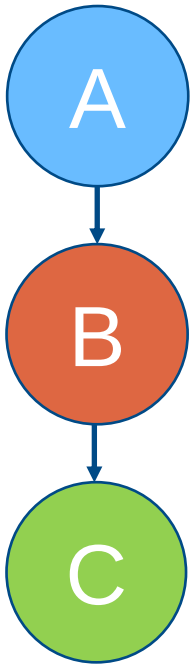



```
queue q;
int* data = malloc_shared<int>(N, q);
for(int i=0;i<N;i++) data[i] = 10;
auto e1 = q.submit([&] (handler &h){
    h.parallel_for<class taskA>(range<1>(N), [=](id<1> i){
        data[i] += 2;
    });
});
auto e2 = q.submit([&] (handler &h){
    h.depends_on(e1);
    h.parallel_for<class taskB>(range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

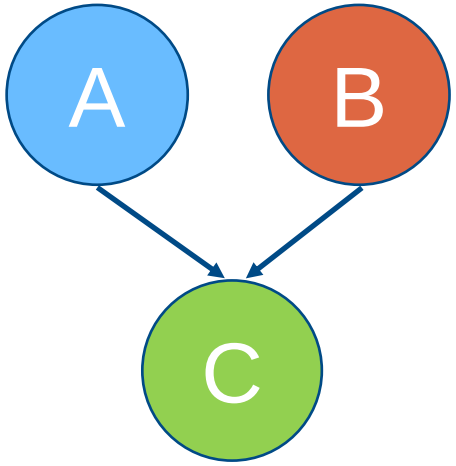
Execution will not overlap even if the queues have no data dependency




```
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

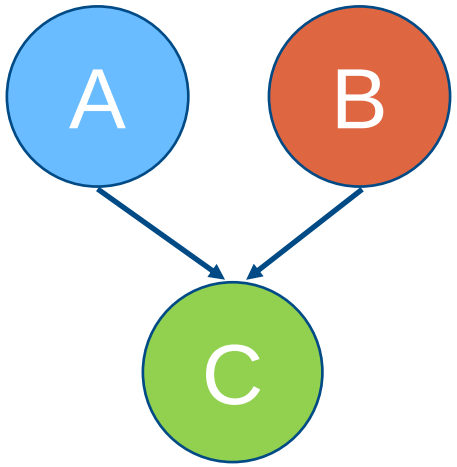



```
queue q;
int* data1 =  malloc_shared<int>(N, q);
int* data2 =  malloc_shared<int>(N, q);
for(int i=0;i<N;i++) {data1[i] = 10; data2[i] = 10;}
auto e1 = q.submit([&] (handler &h){
    h.parallel_for<class taskA>(range<1>(N), [=](id<1> i){
        data1[i] += 2;
    });
});
auto e2 = q.submit([&] (handler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



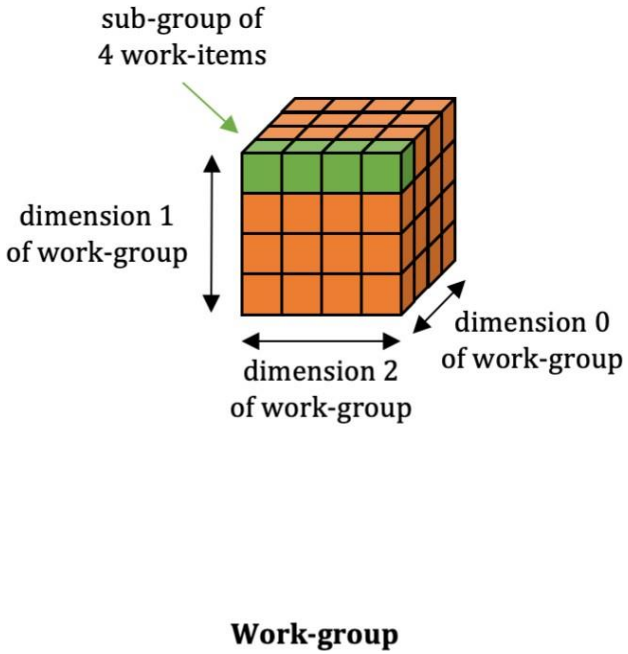

```
queue q;
int* data1 =  malloc_shared<int>(N, q);
int* data2 =  malloc_shared<int>(N, q);
for(int i=0;i<N;i++) {data1[i] = 10; data2[i] = 10;}
auto e1 = q.parallel_for <class taskA>(range<1>(N), [=](id<1> i){
  data1[i] += 2;
});
auto e2 = q.parallel_for <class taskB>(range<1>(N), [=](id<1> i){
 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

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-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

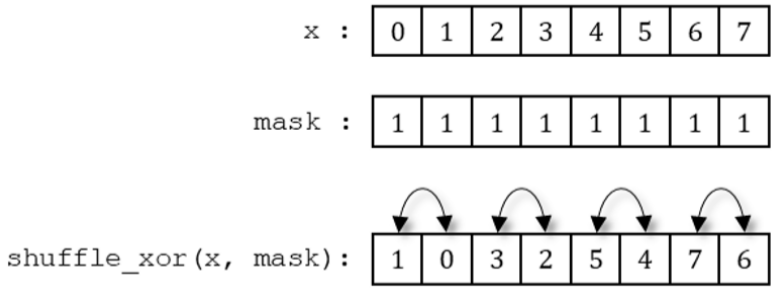
```
h.parallel_for(nd_range<1>(N,B), [=](nd_item<1> item){
          auto sg = item.get_sub_group();

         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
```

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<>());
});
```

Demo/Lab

Data Parallel C++ Essentials Modules

Module 3 - DPC++ Unified Shared Memory
Module 4 - DPC++ Subgroups

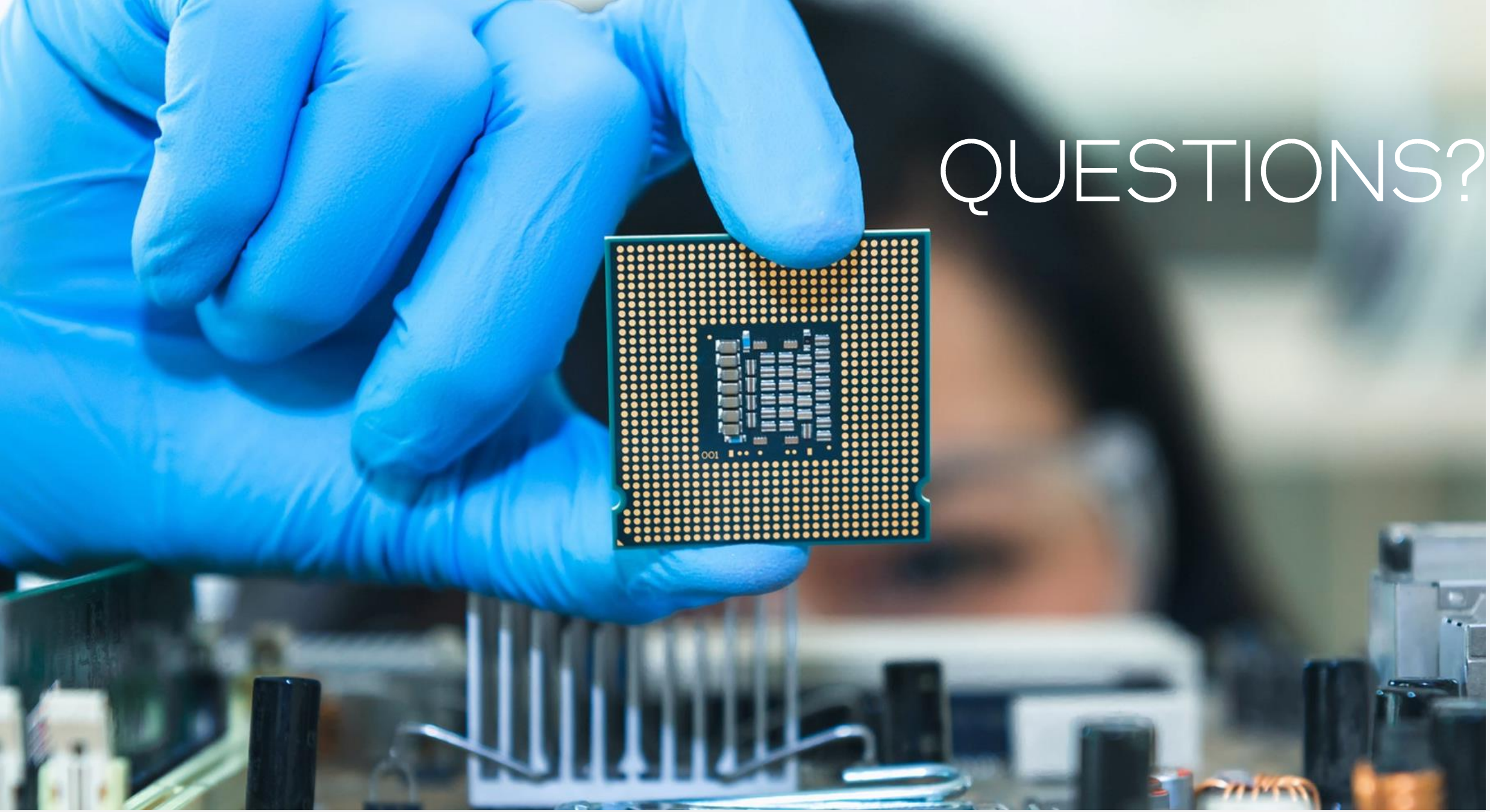
Useful Links

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


QUESTIONS?

Backup

Plugin Interface Implementation

Plugin Interface Implementation

Plugin Interface forwards function calls to supported RT:

OpenCL, Level 0, CUDA, ROCm

usm_impl.cpp

```
void *malloc_host(size_t Size, const context &Ctxt) {
  return detail::usm::alignedAllocHost(0, Size, Ctxt, alloc::host);
}
```

```
void *alignedAllocHost(size_t Alignment, size_t Size, const context &Ctxt, alloc Kind) {

...

    switch (Kind) {
    case alloc::host: {
      Error = Plugin.call_nocheck<PiApiKind::piextUSMHostAlloc>(
          &RetVal, C, nullptr, Size, Alignment);
      break;
    }
```

Plugin Interface implementation

piPluginInit

OpenCL:

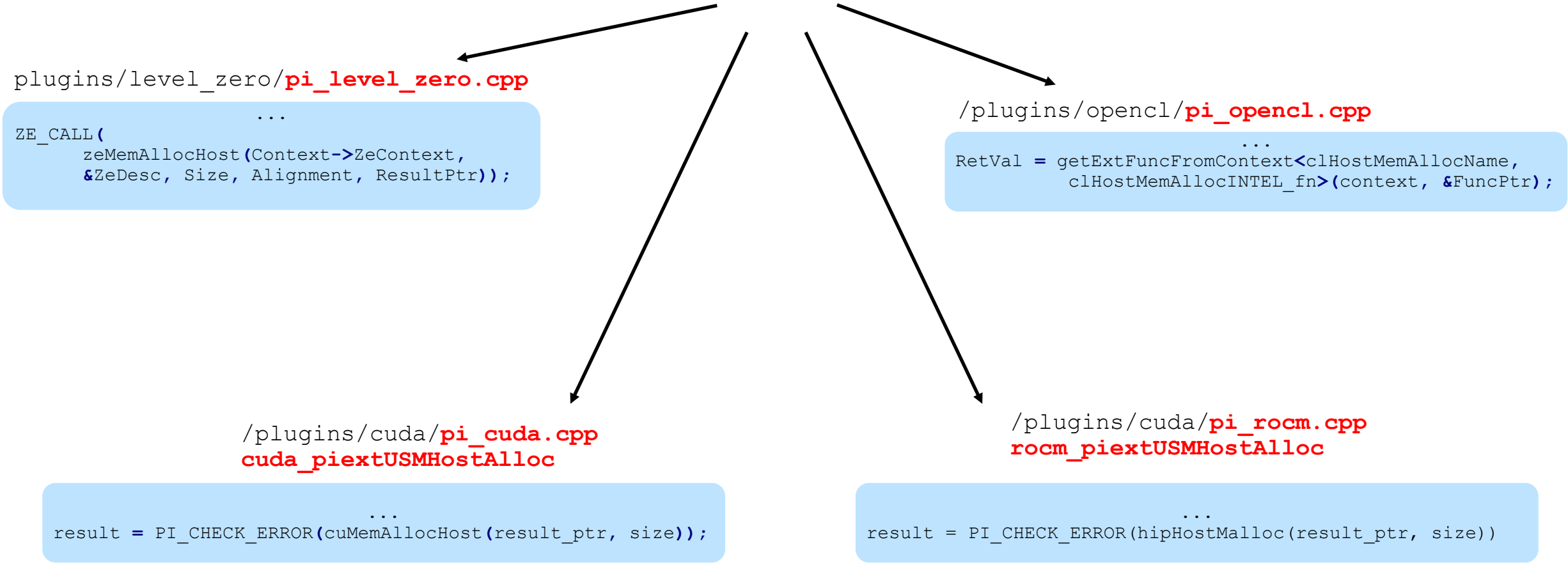
```
#define _PI_CL(pi_api, ocl_api)  \
 (PluginInit->PiFunctionTable).pi_api = (decltype(&::pi_api))(&ocl_api);
 ...
_PI_CL(piPlatformGetInfo, clGetPlatformInfo)
...
_PI_CL(piProgramBuild, clBuildProgram)
...
 _PI_CL(piextUSMHostAlloc, piextUSMHostAlloc)
```

CUDA:

```
// Platform
 _PI_CL(piPlatformsGet, cuda_piPlatformsGet)
 ...
 // Device
 _PI_CL(piDevicesGet, cuda_piDevicesGet)
 _PI_CL(piDeviceGetInfo, cuda_piDeviceGetInfo)
 // Context
 _PI_CL(piextContextSetExtendedDeleter,
cuda_piextContextSetExtendedDeleter)
 _PI_CL(piContextCreate, cuda_piContextCreate)
 // Queue
 _PI_CL(piQueueCreate, cuda_piQueueCreate)
 _PI_CL(piQueueGetInfo, cuda_piQueueGetInfo)
 // USM
 _PI_CL(piextUSMHostAlloc, cuda_piextUSMHostAlloc)
 _PI_CL(piextUSMDeviceAlloc, cuda_piextUSMDeviceAlloc)
 _PI_CL(piextUSMSharedAlloc, cuda_piextUSMSharedAlloc)
```

Plugin Interface implementation

```
pi_result piextUSMHostAlloc(void **result_ptr, pi_context context,
                            pi_usm_mem_properties *properties, size_t size,
                            pi_uint32 alignment) {
```



plugins/level_zero/**pi_level_zero.cpp**

```
...
ZE_CALL(
      zeMemAllocHost(Context->ZeContext,
      &ZeDesc, Size, Alignment, ResultPtr));
```

/plugins/opencl/**pi_opencl.cpp**

```
...
RetVal = getExtFuncFromContext<clHostMemAllocName,
          clHostMemAllocINTEL_fn>(context, &FuncPtr);
```

/plugins/cuda/**pi_cuda.cpp**
cuda_piextUSMHostAlloc

```
...
result = PI_CHECK_ERROR(cuMemAllocHost(result_ptr, size));
```

/plugins/cuda/**pi_rocm.cpp**
rocm_piextUSMHostAlloc

```
...
result = PI_CHECK_ERROR(hipHostMalloc(result_ptr, size))
```

Main optimization concepts

Remember Amdahl's Law

- The maximum speedup is bounded

Locality Matters

- Get your data closer to the execution
  - Keep your data on the accelerator for as long as possible
  - Access contiguous blocks of memory as your kernel executes
  - Restructure your code into blocks with higher data reuse

Rightsize Your Work

- Fully utilize a parallel processor
- 512 EU * 8 threads * 16 vector operations = 65536 parallel activities

Shared Local Memory

SLM has limited size, e.g. 64KB SLM for each work-group

```
std::cout << "Local Memory Size: " <<
q.get_device().get_info<sycl::info::device::local_mem_size>() << std::endl;
```

Use local_accessor class or target::local accessor specialization

```
q.submit([&](auto &h) {
  sycl::accessor<int, 1, sycl::access::mode::read_write,
                 sycl::access::target::local> slm(sycl::range(32 * 64), h);
  h.parallel_for(sycl::nd_range(sycl::range{N}, sycl::range{32}), [=](sycl::nd_item<1> it)
  {
    //...
```

tinyurl.com/SLM-in-SYCL

Fortran and DPC++

- iso_c_binding
  - Fortran 2003 and later standardized mechanism for allowing Fortran code to reliably communicate (or *interoperate*) with C/C++ code.
- DPC++ is a C++
- Need to create interfaces in Fortran and link

DPC++ Function and Kernel

kernel.cpp

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

extern "C" void launch_test_kernel(float *A, float* B, float *C, int size)
{
     {
       buffer bufA (A, range(size)), bufB (B, range(size)), bufC (C, range(size));
       queue q;
       std::cout << "Running on " << q.get_device().get_info<sycl::info::device::name>() << std::endl;

       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<class test_kernel>(range(size), [=](auto i){
               C[i] = A[i] + B[i];
           });
       });
     }
 for (int i = 0; i < size; i++)
        std::cout << "C[" << i << "] = " << C[i] << std::endl;

 }
```

Fortran Interface

module.f90

```
module cuda_c_kernel

implicit none

interface
    subroutine launch_test_kernel(A, B, C, sz) &
               bind(c)
      use, intrinsic :: iso_c_binding
      implicit none
      real ( c_float) :: A(*), B(*), C(*)
      integer ( c_int ), value :: sz
    end subroutine
  end interface

end module cuda_c_kernel
```

module.f90

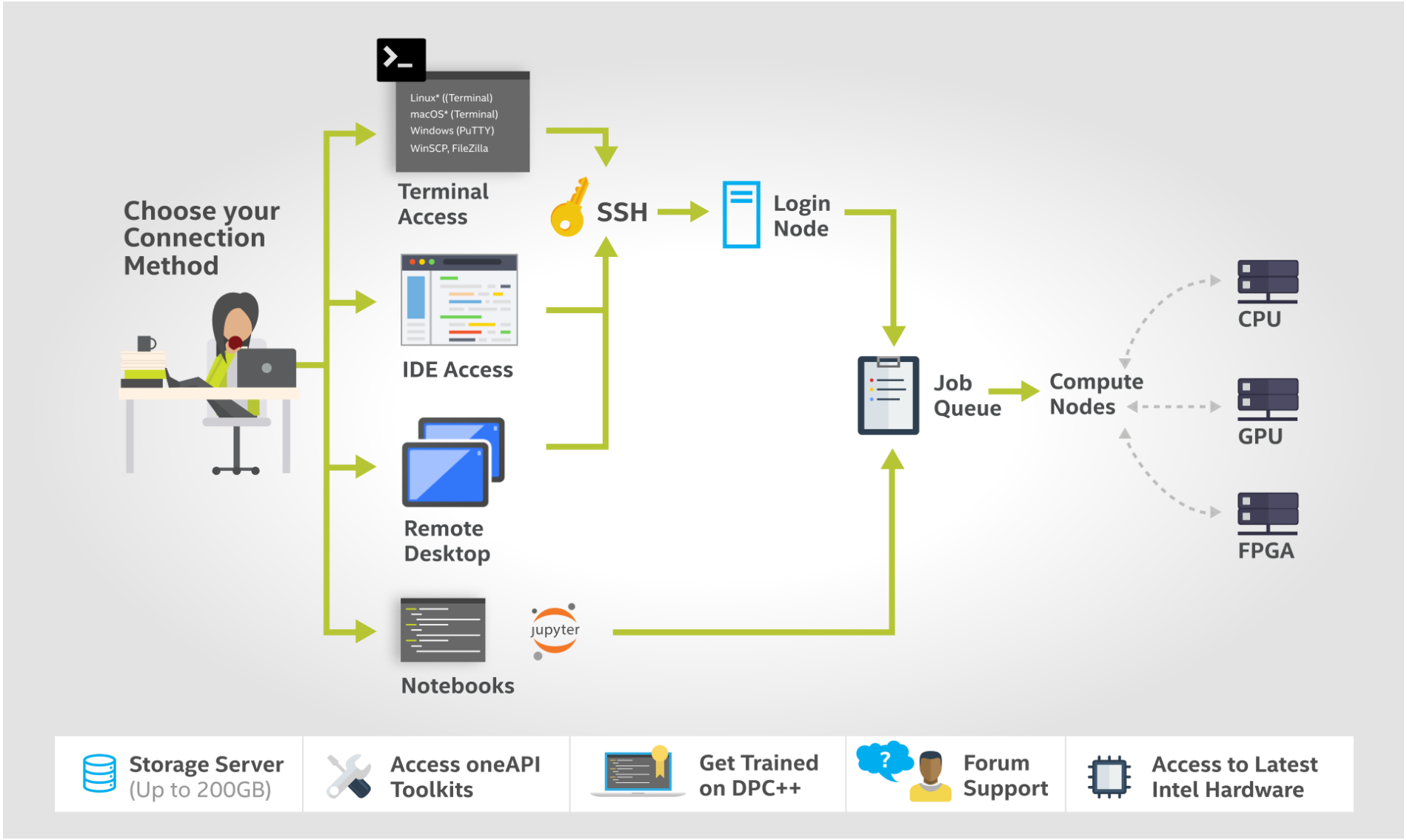
```
program test
        use cuda_c_kernel
        use, intrinsic :: iso_c_binding
        integer ( c_int ), parameter :: n = 1024
        real ( c_float ) :: A(n), B(n), C(n)

        A=1
        B=2

        call launch_test_kernel(A, B, C, n)

end program test
```

Compilation and Linking

```
source /opt/intel/oneapi/setvars.sh

dpcpp -c kernel.cpp

dpcpp -fsycl-link kernel.cpp

ifort module.f90 kernel.o kernel-spir64.o -lsycl -lstdc++
```

```
$ ./a.out
Running on Intel(R) Graphics [0x3e92]
C[0] = 3
C[1] = 3
C[2] = 3
...
C[1022] = 3
C[1023] = 3
```

Intel® DevCloud



Notices & Disclaimers

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details.
No product or component can be absolutely secure.

Your costs and results may vary.

Intel technologies may require enabled hardware, software or service activation.

Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.


intel