oneAPI

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

oneAPI, Vendor Neutral Programming Model for Heterogeneous Programming

Anoop Madhusoodhanan Prabha

Intel Architecture, Graphics & Software

May 2021

intel

Agenda

What is oneAPI?

What is DPC++ ?

Intel Compilers

“Hello World” Example

- Basic Concepts: buffer, accessor, queue, kernel, etc.

Synchronization

Error Handling

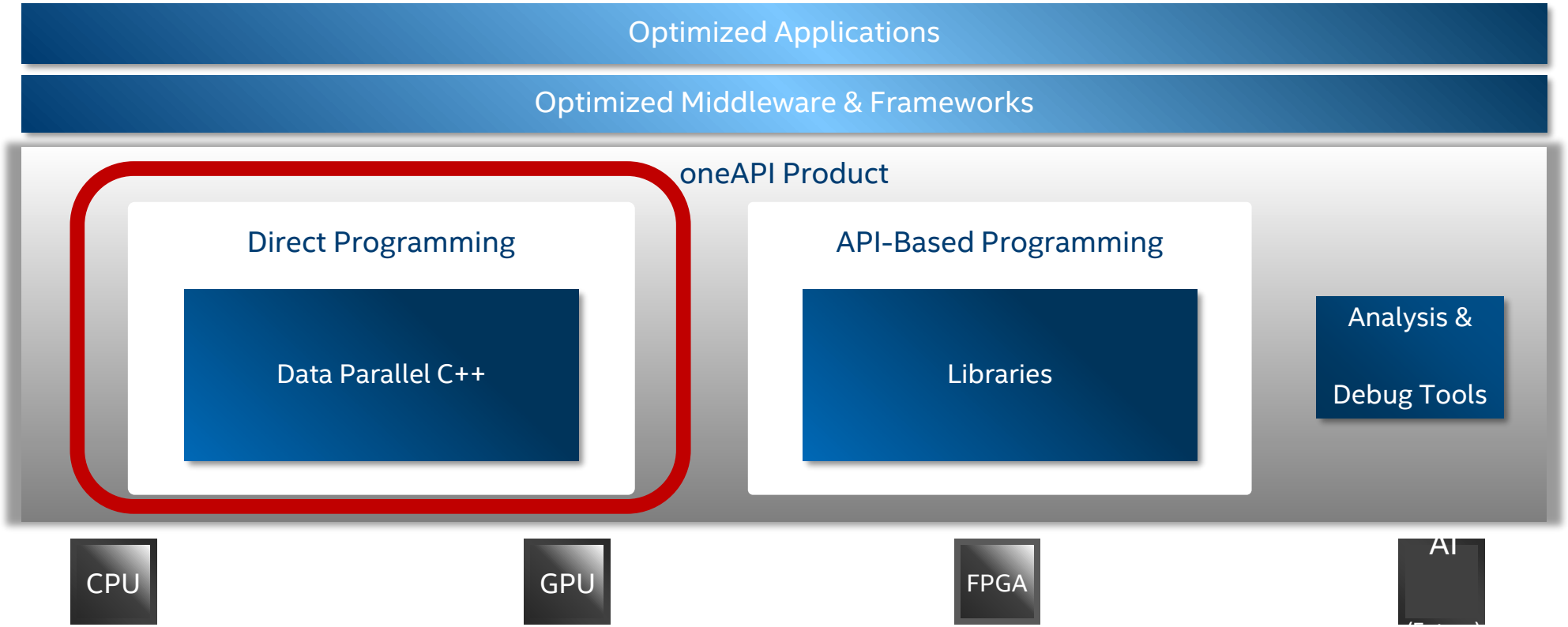
Device Selection

Unified Shared Memory

DPC++ Compatibility Tool

What is oneAPI?

oneAPI for Cross-Architecture Performance



- Get functional quickly. Then analyze and tune.

Some capabilities may differ per architecture.

What is DPC++?

Data Parallel C++

Standards-based, Cross-architecture Language

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

Freedom of Choice: Future-Ready Programming Model

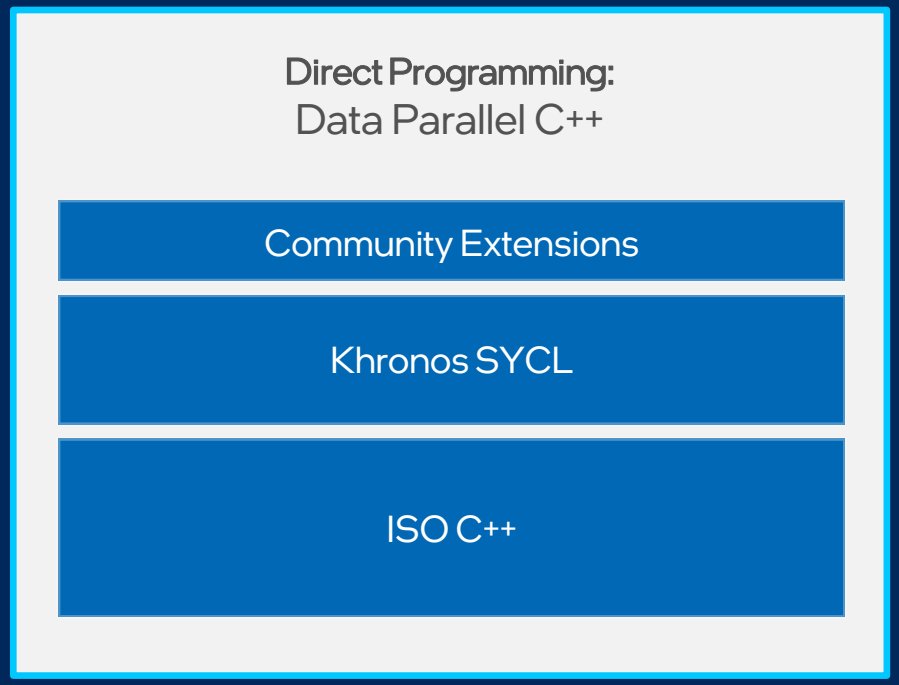
- Allows code reuse across hardware targets
- Permits custom tuning for a specific accelerator
- Open, cross-industry alternative to proprietary language

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

- Delivers C++ productivity benefits, using common, familiar C and C++ constructs
- Adds SYCL from the Khronos Group for data parallelism and heterogeneous programming

Community Project Drives Language Enhancements

- Provides extensions to simplify data parallel programming
- Continues evolution through open and cooperative development



The open source and Intel DPC++/C++ compiler supports Intel CPUs, GPUs, and FPGAs.
Codeplay announced a DPC++ compiler that targets Nvidia GPUs.

Intel® oneAPI DPC++/C++ Compiler

Parallel Programming Productivity & Performance

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

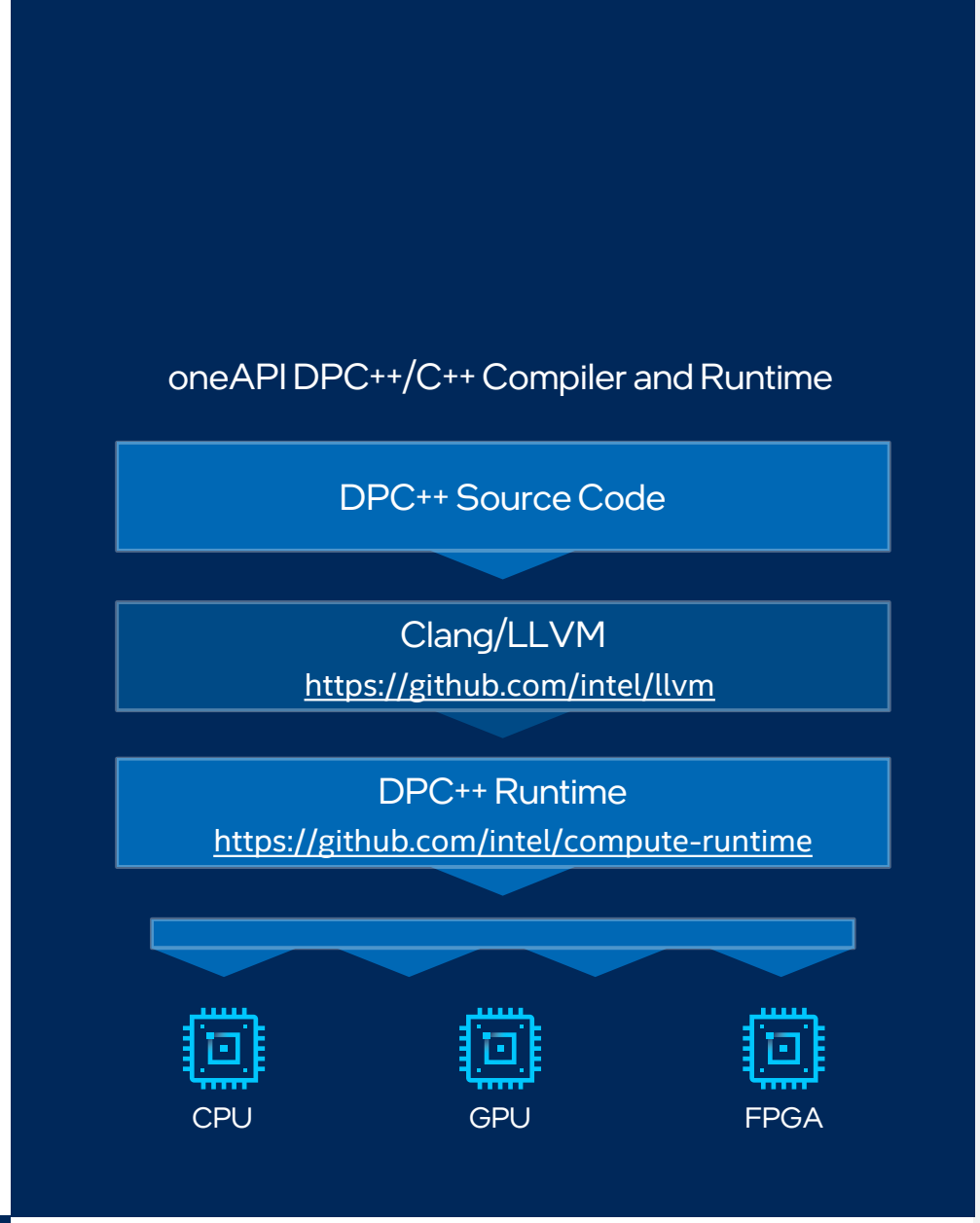
- Allows code reuse across hardware targets, while permitting custom tuning for a specific accelerator
- Open, cross-industry alternative to single architecture proprietary language

Builds upon Intel's decades of experience in architecture and high-performance compilers

Code samples:

tinyurl.com/dpcpp-tests

tinyurl.com/oneapi-samples



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

SYCL™ 2020 final specification

- Released Feb 9, 2021
  - khronos.org/news/press/khronos-releases-sycl-2020-final-specification
- 40+ new features
  - Unified Shared Memory (USM)
  - Parallel reductions
  - Work group and subgroup algorithms
  - Class template argument deduction (CTAD) and template deduction
  - Simplified use of Accessors with a built-in reduction operation
  - Expanded interoperability
- Data Parallel C++ already incorporates many SYCL 2020 features

Intel® Compilers

tinyurl.com/openmp-and-dpcpp

Intel Compiler	Target	OpenMP Support	OpenMP Offload Support	Included in oneAPI Toolkit
Intel® C++ Compiler, IL0 ***icc***	CPU	Yes	No	HPC
Intel® oneAPI DPC++/C++ Compiler ***dpcpp***	CPU, GPU, FPGA*	Yes	Yes	Base
Intel® oneAPI DPC++/C++ Compiler ***icx***	CPU GPU*	Yes	Yes	Base
Intel® Fortran Compiler, IL0 ***ifort***	CPU	Yes	No	HPC
Intel® Fortran Compiler ***ifx***	CPU, GPU*	Yes	Yes	HPC

Cross Compiler Binary Compatible and Linkable!

tinyurl.com/oneAPI-Base-download

tinyurl.com/oneAPI-HPC-download

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 which 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

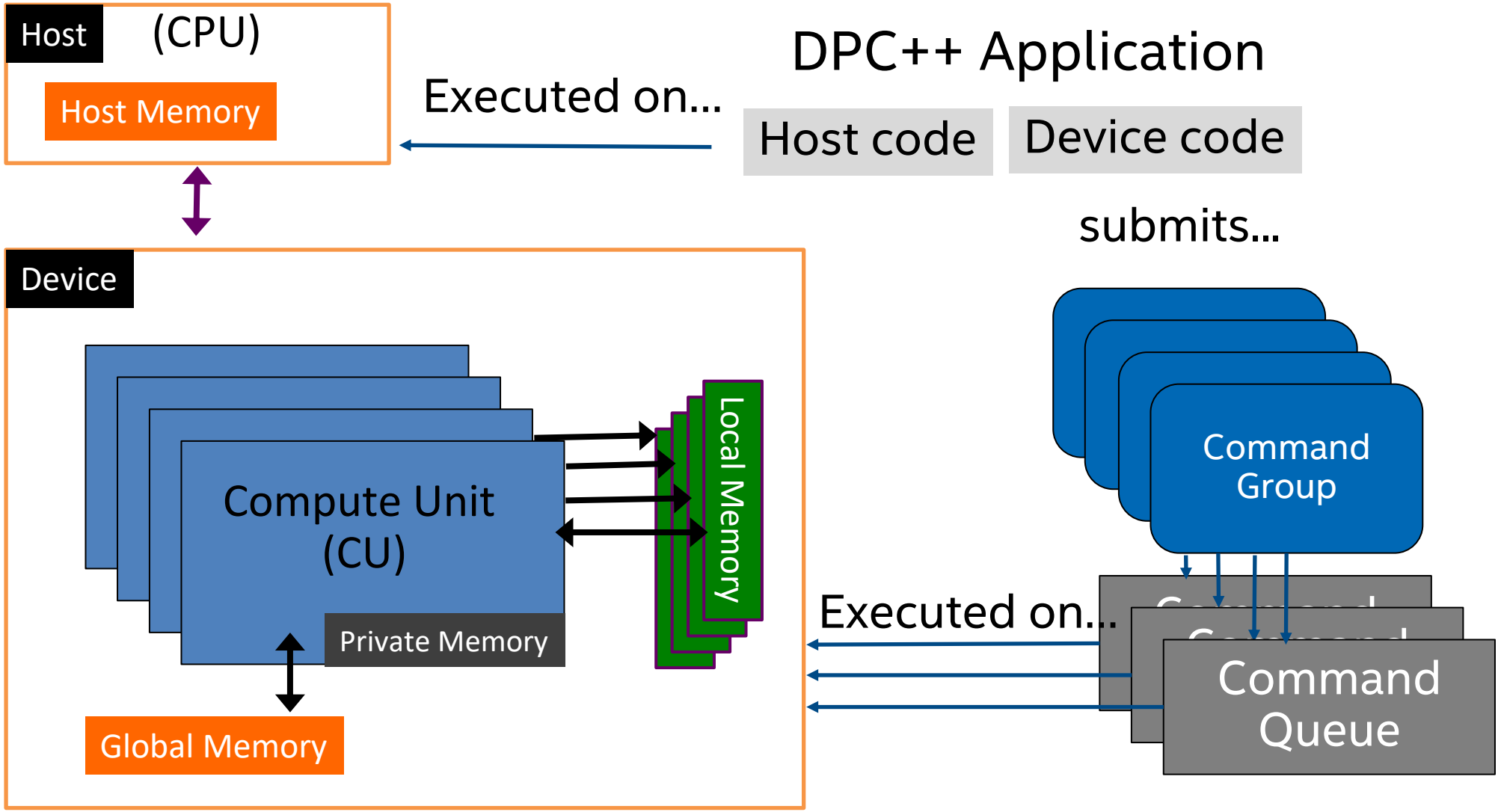
Other News

Codeplay Brings NVIDIA GPU Support to Industry-Standard Math Library

Intel Open Sources the oneAPI Math Kernel Library Interface

“Hello World” Example

DPC++ Basics


Host
(CPU)
Host Memory
Executed on…
DPC++ 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 DPC++ 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 DPC++ 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

DPC++ 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!

DPC++ 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;

 }
```

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

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

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

The Buffer Model

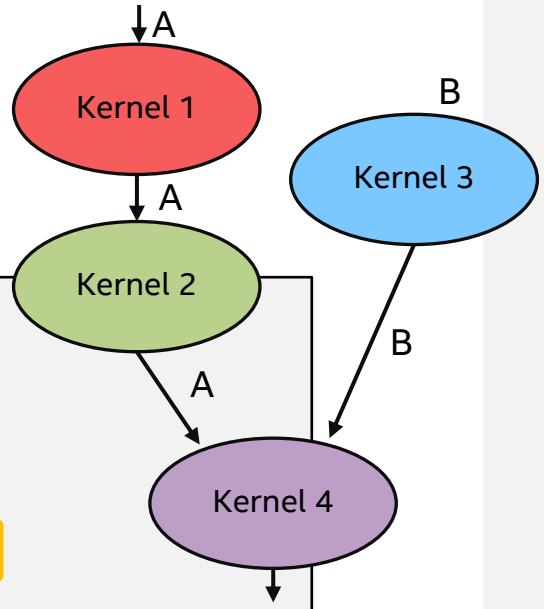
Buffers: Encapsulate data in a SYCL application

- Across both devices and host!

Accessors: Mechanism to access buffer data

- Create data dependencies in the SYCL graph that order kernel executions

```
int main() {
  auto R = range<1>{ num };
  buffer<int> A{ R }, B{ R };
  queue Q;

  Q.submit([&](handler& h) {
    accessor out(A, h, write_only);

    h.parallel_for(R, [=](auto idx) {
      out[idx] = idx[0]; }); });

  Q.submit([&](handler& h) {
              accessor out(A, h, write_only);
    h.parallel_for(R, [=](auto idx) {
      out[idx] = idx[0]; }); });
…
```



DPC++ 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

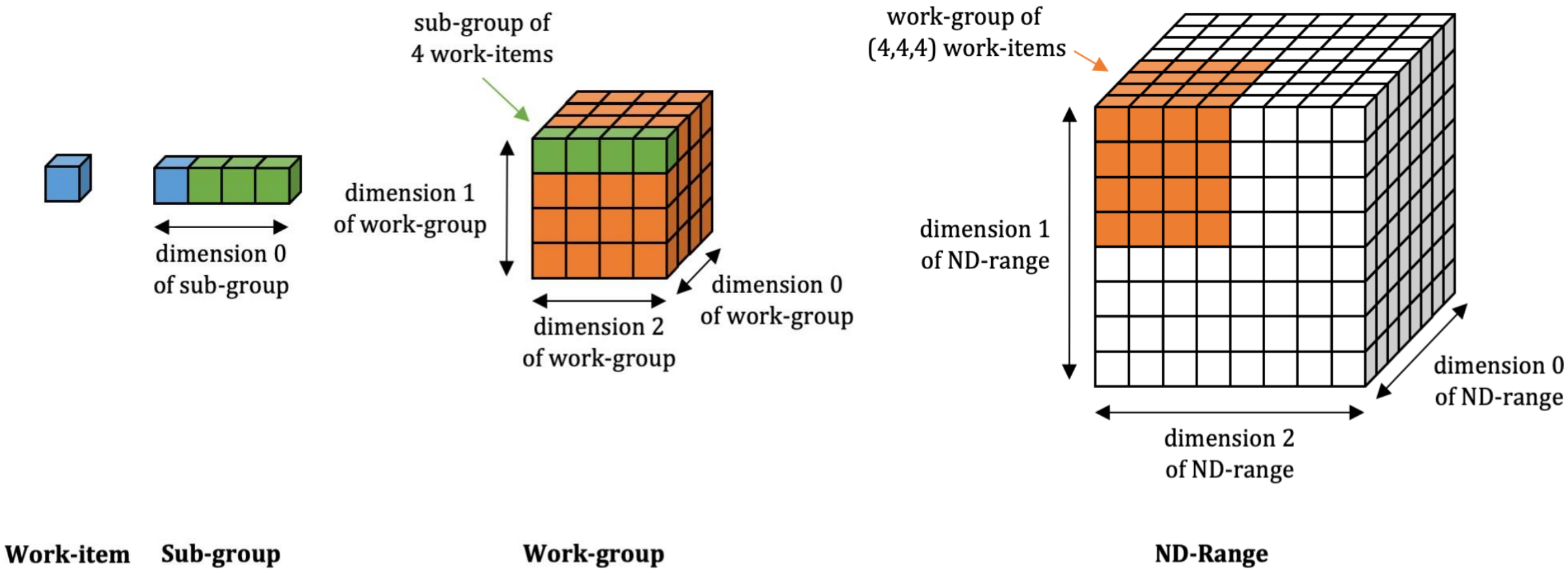
The functionality of basic parallel kernels is exposed via range, id and item classes

- 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
});
```

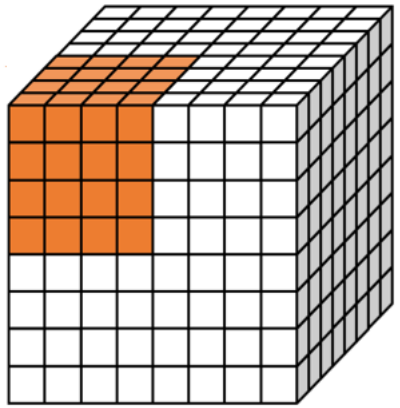
```
h.parallel_for(range<1>(1024), [=](item<1> item){
            auto idx = item.get_id();
            auto R = item.get_range();
            // CODE THAT RUNS ON DEVICE
});
```

DPC++ Thread Hierarchy and Mapping

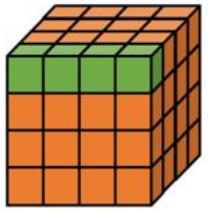


DPC++ Thread Hierarchy and Mapping

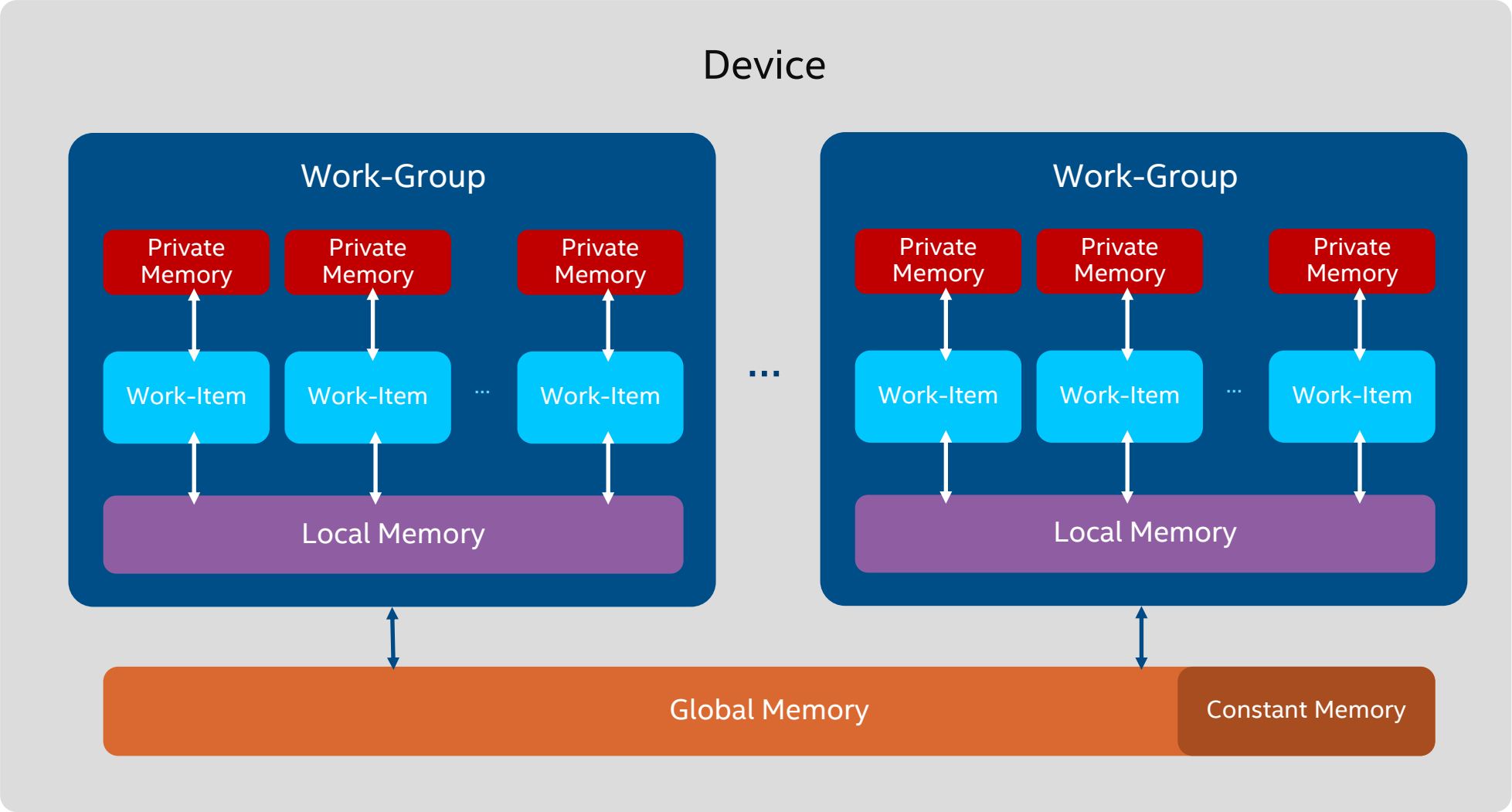
All work-items in a **work-group** are scheduled on one Compute Unit, which has its own local memory



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

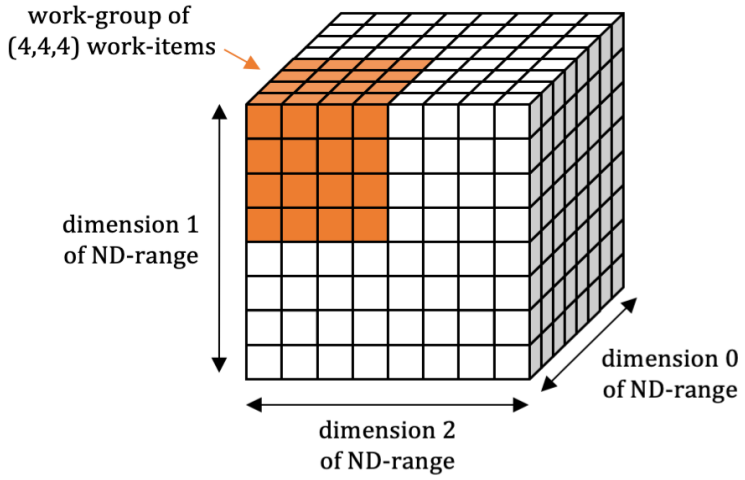


Logical Memory Hierarchy



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.

DPC++ Functions for Invoking Kernels

```
h.single_task(
  [=]() {
    // kernel function is executed EXACTLY once on a SINGLE work-item
});

h.parallel_for(
  range<3>(1024,1024,1024), // using 3D in this example
  [=](id<3> myID) {
    // kernel function is executed on an n-dimensional range (NDrange)
});
h.parallel_for(
  nd_range<3>({1024,1024,1024},{16,16,16}), // using 3D in this example
  [=](nd_item<3> myID) {
    // kernel function is executed on an n-dimensional range (NDrange)

});
h.parallel_for_work_group(
  range<2>(1024,1024),      // using 2D in this example
  [=](group<2> grp) {
    // kernel function is executed once per work-group
});
grp.parallel_for_work_item(
  range<1>(1024),           // using 1D in this example
    // kernel function is executed once per work-item

  [=](h_item<1> myItem) {

});
```

Basic data parallel

Explicit ND-Range

Hierarchical parallelism

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

Error Handling

DPC++ is based on C++

- Errors in C++ are handled through exceptions
- SYCL uses exceptions, not return codes!

Synchronous exceptions

- Detected immediately
- Use try...catch block

Asynchronous exceptions

- Detected later after an API call has returned
- E.g. faults inside a command group or a kernel
- Use error handler function

Synchronous exceptions

- Thrown immediately when an API call fails
  - failure to construct an object, e.g. can't create buffer
- Normal C++ exceptions

```
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

```
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;
    }
  }
};
```

- queue::wait _and_throw(), queue::throw_asynchronous(), event::wait_and_throw()

Device Selection

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

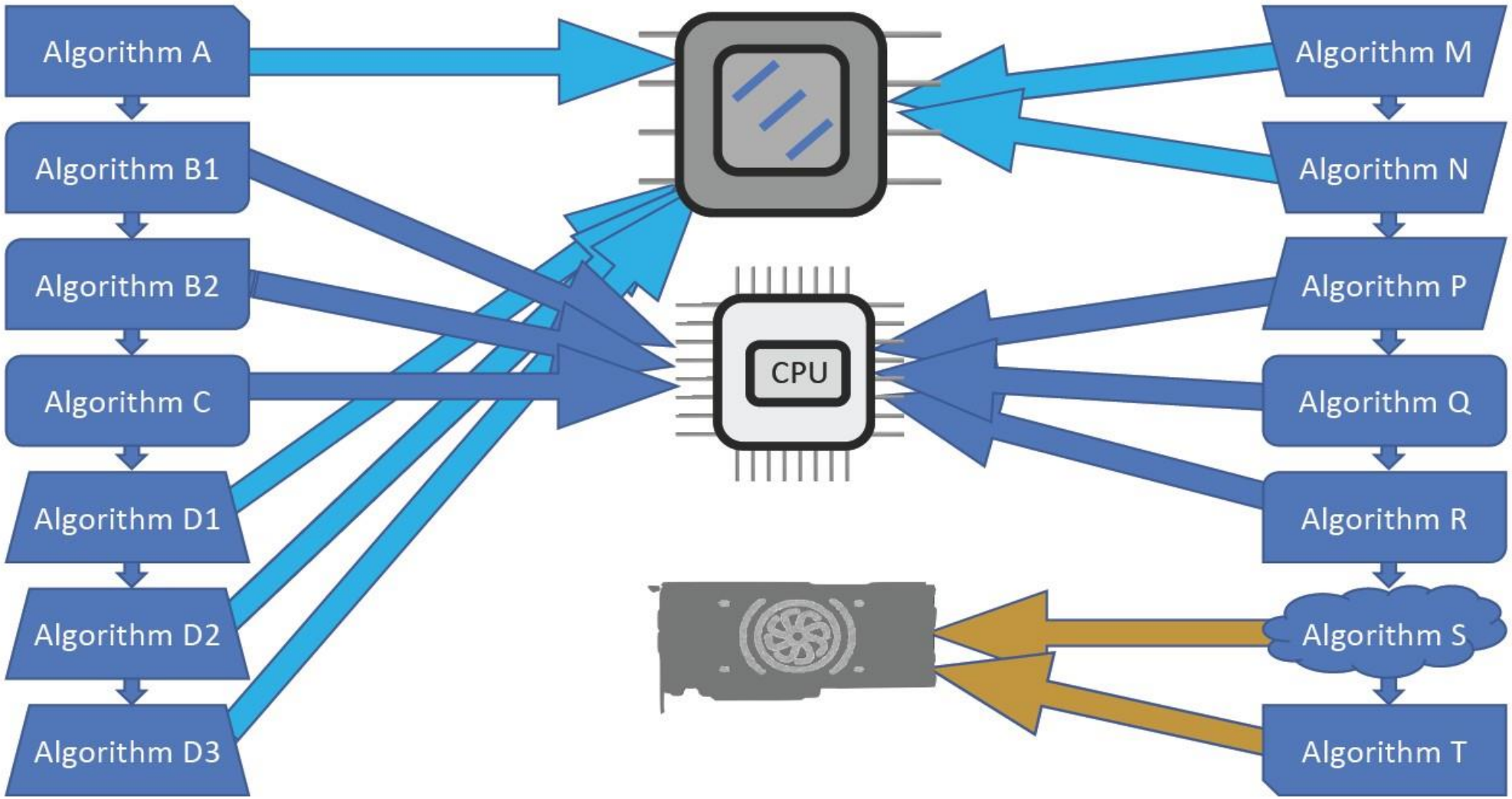
new

export SYCL_DEVICE_FILTER={backend:device_type:device_num}

*will be available in oneAPI Update 1

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

DPC++ Device Selection



Unified Shared Memory

Motivation

The SYCL 1.2.1 standard provides a Buffer memory abstraction

- Powerful and elegantly expresses data dependences

However…

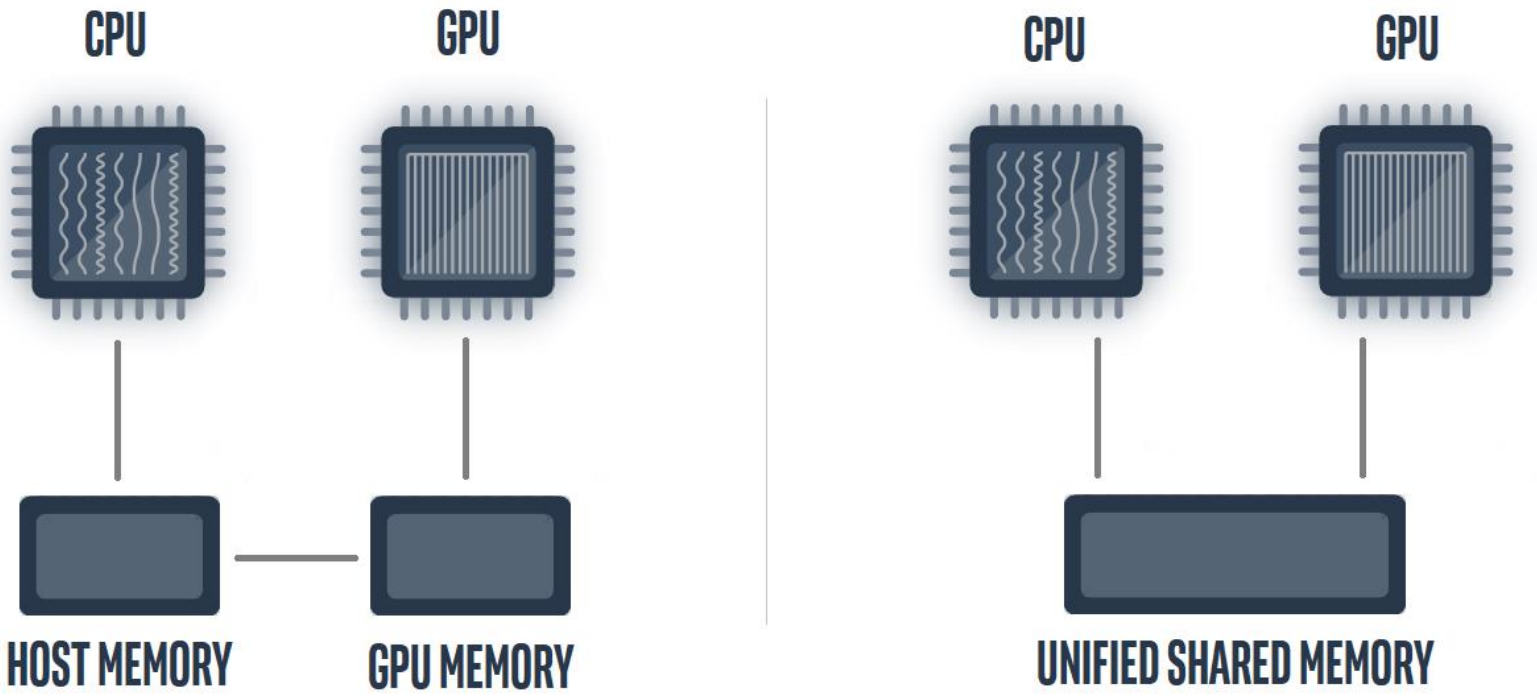
- Replacing all pointers and arrays with buffers in a C++ program can be a burden to programmers

USM provides a pointer-based alternative in DPC++

- 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



DPC++ 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

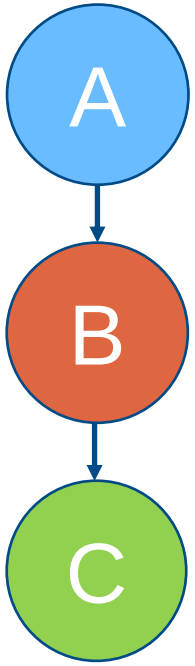
Dependences must be specified explicitly using events

- 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

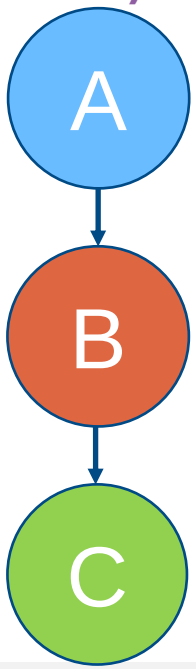
wait() will block execution on host




```
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. (SYCL2020)

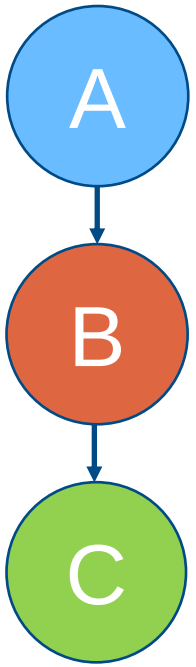



```
queue q;
int* data = malloc_shared<int>(N, q);
for(int i=0;i<N;i++) data[i] = 10;
auto e1 = q.submit([&] (handler &h){
    h.parallel_for<class taskA>(range<1>(N), [=](id<1> i){
        data[i] += 2;
    });
});
auto e2 = q.submit([&] (handler &h){
    h.depends_on(e1);
    h.parallel_for<class taskB>(range<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 (SYCL2020)

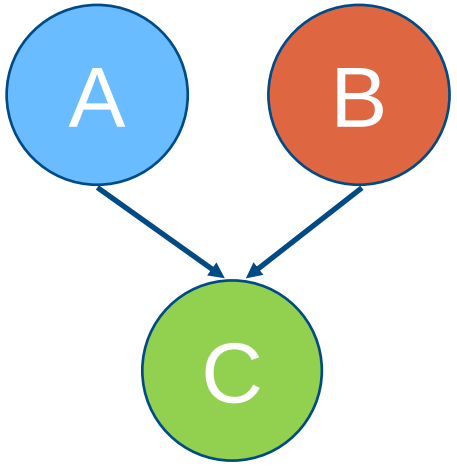
Execution will not overlap even if the queues have no data dependency




```
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

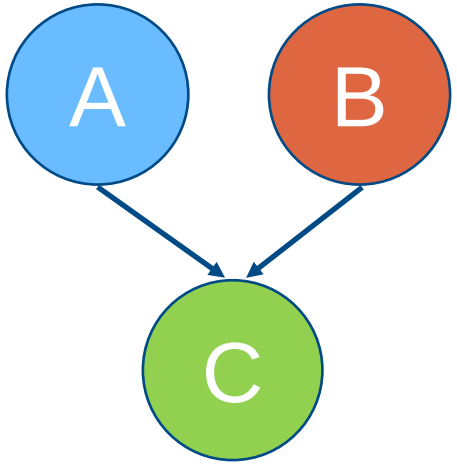



```
queue q;
int* data1 =  malloc_shared<int>(N, q);
int* data2 =  malloc_shared<int>(N, q);
for(int i=0;i<N;i++) {data1[i] = 10; data2[i] = 10;}
auto e1 = q.submit([&] (handler &h){
    h.parallel_for<class taskA>(range<1>(N), [=](id<1> i){
        data1[i] += 2;
    });
});
auto e2 = q.submit([&] (handler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

Intel® DPC++ Compatibility Tool

Intel® DPC++ Compatibility Tool

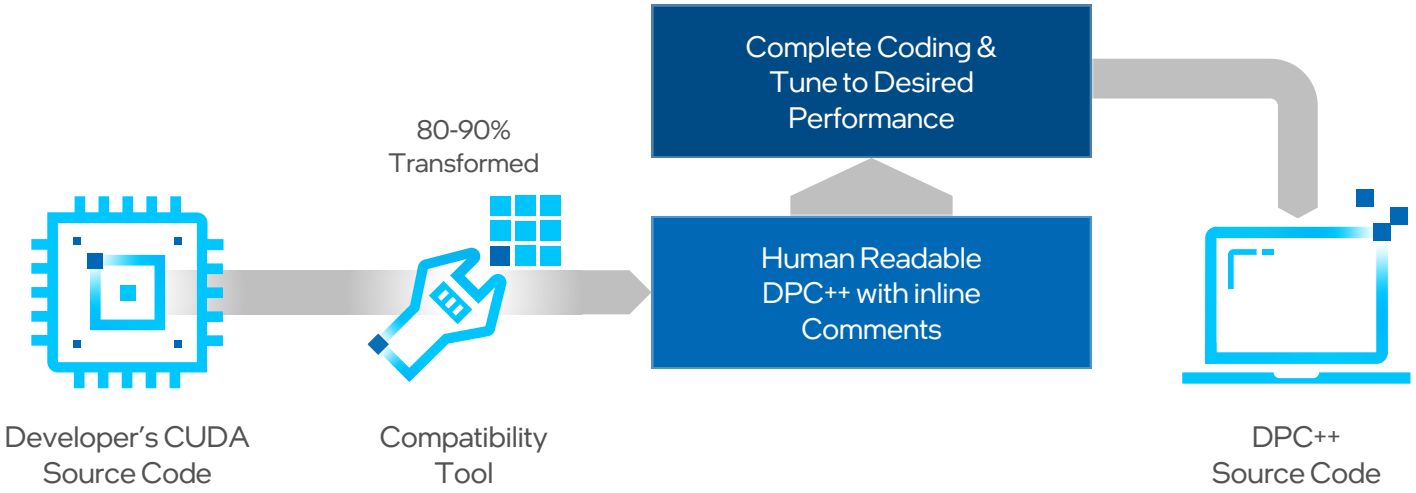
Minimizes Code Migration Time

Assists developers migrating code written in CUDA to DPC++ once, generating **human readable** code wherever possible

~80-90% of code typically migrates automatically

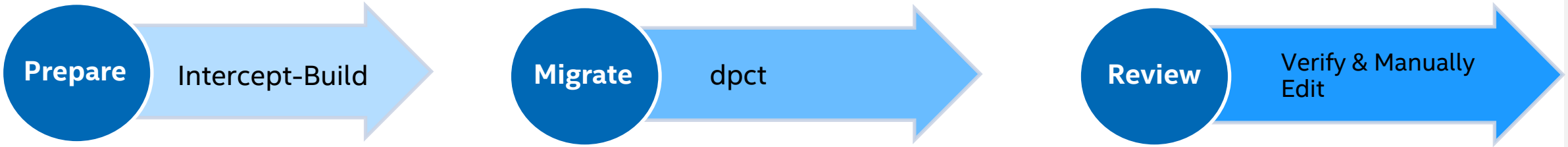
Inline comments are provided to help developers finish porting the application

Intel DPC ++ Compatibility Tool Usage Flow



Intel® DPC++ Compatibility Tool

Migration of Large Code Bases



1. Create a compilation database file

   intercept-build make

2. Migrate your source to DPC++
   dpct -p compile_commands.json
   -in-root=$PROJ_DIR
   -out-root=dpcpp_out *.cu

3. Verify the source for correctness and fix not migrated parts

https://tinyurl.com/intel-dpcpp-compatibility-tool

Migration example

Header files

```cuda
#include <cuda.h>

#define VECTOR_SIZE 4
__global__ void VectorAddKernel (float *A, float *B, float *C)
{
  A[threadIdx.x] = threadIdx.x + 1.0f;
  B[threadIdx.x] = threadIdx.x + 1.0f;
  C[threadIdx.x] = A[threadIdx.x] + B[threadIdx.x];
}


int main()
{


  float *d_A, *d_B, *d_C;
  cudaMalloc(&d_A, VECTOR_SIZE*sizeof(float));
  cudaMalloc(&d_B, VECTOR_SIZE*sizeof(float));
  cudaMalloc(&d_C, VECTOR_SIZE*sizeof(float));



  VectorAddKernel<<<1, VECTOR_SIZE>>>(d_A, d_B, d_C);



  float Result[VECTOR_SIZE] = { };
  cudaMemcpy(Result, d_C,VECTOR_SIZE*sizeof(float),cudaMemcpyDeviceToHost);

  cudaFree(d_A);
  cudaFree(d_B);
  cudaFree(d_C);
}
```

Kernel

Mem alloc

Kernel call

Mem copy

Mem free

```sycl
#include <CL/sycl.hpp>
#include <dpct/dpct.hpp>
#define VECTOR_SIZE 4
void VectorAddKernel (float *A, float *B, float *C, sycl::nd_item<3> item_ct1)
{
  A[item_ct1.get_local_id(2)] = item_ct1.get_local_id(2) + 1.0f;
  B[item_ct1.get_local_id(2)] = item_ct1.get_local_id(2) + 1.0f;
  C[item_ct1.get_local_id(2)] = A[item_ct1.get_local_id(2)] + B[item_ct1.get_local_id(2)];
}


int main()
{
  dpct::device_ext &dev_ct1 = dpct::get_current_device();
  sycl::queue &q_ct1 = dev_ct1.default_queue();

  float *d_A, *d_B, *d_C;
  d_A = sycl::malloc_device<float>(VECTOR_SIZE, q_ct1);
  d_B = sycl::malloc_device<float>(VECTOR_SIZE, q_ct1);
  d_C = sycl::malloc_device<float>(VECTOR_SIZE, q_ct1);

  q_ct1.submit([&](sycl::handler &cgh){
    cgh.parallel_for(sycl::nd_range(sycl::range(1, 1, VECTOR_SIZE),
      sycl::range(1, 1, VECTOR_SIZE)), [=](sycl::nd_item<3> item_ct1) {
      VectorAddKernel(d_A, d_B, d_C, item_ct1);
    });
  });


  float Result[VECTOR_SIZE] = { };
  q_ct1.memcpy(Result, d_C, VECTOR_SIZE * sizeof(float)).wait();

  sycl::free(d_A, q_ct1);
  sycl::free(d_B, q_ct1);
  sycl::free(d_C, q_ct1);
}
```

General Best Known Methods

- Migrate Incrementally
  - If you see *dpct* generate multiple errors when migrating a long list of CUDA source files in one run, do it one-by-one
- Start with a clean project - “make clean” before running “intercept-build make”

Demo

Demo

Pre-requisites on your own system:

- Get the latest oneAPI Base Toolkit:

  https://software.intel.com/content/www/us/en/develop/tools/oneapi/base-toolkit/download.html

- Set the environment, e.g. source /opt/intel/oneapi/setvars.sh

or use Intel DevCloud devcloud.intel.com

- git clone https://github.com/oneapi-src/oneAPI-samples.git
- cd oneAPI-samples/Tools/Migration/
- dpct --help

NOTE: You need to have CUDA* headers available to migrate the code

vecAdd

- *cd vector-add-dpct/src*
- *dpct --cuda-include-path=/home/u64609/include vector_add.cu*

```
NOTE: Could not auto-detect compilation database for file 'vector_add.cu' in '/home/u64609/oneAPI-samples/Tools/Migration/vector-add-dpct/src' or any parent directory.
The directory "dpct_output" is used as "out-root"
Processing: /home/u64609/oneAPI-samples/Tools/Migration/vector-add-dpct/src/vector_add.cu
/home/u64609/oneAPI-samples/Tools/Migration/vector-add-dpct/src/vector_add.cu:32:14: warning: DPCT1003:0: Migrated API does not return error code. (*, 0) is inserted. You may need to rewrite this code.
   status = cudaMemcpy(Result, d_C, VECTOR_SIZE*sizeof(float), cudaMemcpyDeviceToHost);
          ^
Processed 1 file(s) in -in-root folder "/home/u64609/oneAPI-samples/Tools/Migration/vector-add-dpct/src"

See Diagnostics Reference to resolve warnings and complete the migration:
https://software.intel.com/content/www/us/en/develop/documentation/intel-dpcpp-compatibility-tool-user-guide/top/diagnostics-reference.html
```

- File vector_add.dp.cpp is generated in dpct_output directory

vecAdd

- *dpct --cuda-include-path=/home/u64609/include --enable-ctad --out-root=test1 vector_add.cu*

  diff dpct_output/vector_add.dp.cpp test1/vector_add.dp.cpp

```
32,33c32,33
<       cgh.parallel_for(sycl::nd_range<3>(sycl::range<3>(1, 1, VECTOR_SIZE),
<                                          sycl::range<3>(1, 1, VECTOR_SIZE)),
---
>       cgh.parallel_for(sycl::nd_range(sycl::range(1, 1, VECTOR_SIZE),
>                                       sycl::range(1, 1, VECTOR_SIZE)),
```

- *dpct --cuda-include-path=/home/u64609/include --enable-ctad --out-root=test2 --keep-original-code vector_add.cu*

```
/* DPCT_ORIG __global__ void VectorAddKernel(float* A, float* B, float* C)*/
void VectorAddKernel(float *A, float *B, float *C, sycl::nd_item<3> item_ct1)
{
/* DPCT_ORIG     A[threadIdx.x] = threadIdx.x + 1.0f;*/
    A[item_ct1.get_local_id(2)] = item_ct1.get_local_id(2) + 1.0f;
/* DPCT_ORIG     B[threadIdx.x] = threadIdx.x + 1.0f;*/
    B[item_ct1.get_local_id(2)] = item_ct1.get_local_id(2) + 1.0f;
/* DPCT_ORIG     C[threadIdx.x] = A[threadIdx.x] + B[threadIdx.x];*/
    C[item_ct1.get_local_id(2)] =
        A[item_ct1.get_local_id(2)] + B[item_ct1.get_local_id(2)];
} ...
  /* DPCT_ORIG     cudaMalloc(&d_A, VECTOR_SIZE*sizeof(float));*/
      d_A = sycl::malloc_device<float>(VECTOR_SIZE, q_ct1);
  /* DPCT_ORIG     cudaMalloc(&d_B, VECTOR_SIZE*sizeof(float));*/
      d_B = sycl::malloc_device<float>(VECTOR_SIZE, q_ct1);
  /* DPCT_ORIG     cudaMalloc(&d_C, VECTOR_SIZE*sizeof(float));*/
      d_C = sycl::malloc_device<float>(VECTOR_SIZE, q_ct1);

  /* DPCT_ORIG     VectorAddKernel<<<1, VECTOR_SIZE>>>(d_A, d_B, d_C);*/
      q_ct1.submit([&](sycl::handler &cgh) {
```

Rodinia NW

- *cd ~/dpct_demo/oneAPI-samples/Tools/Migration/rodinia-nw-dpct*
- *make clean*

1. *intercept-build make*

cat compile_commands.json

clang.llvm.org/docs/JSONCompilationDatabase.html

```
[
  {
    "command": "nvcc -c  -o needleman_wunsch_cu -D__CUDACC__=1 src/needle.cu",
    "directory": "/home/u64609/oneAPI-samples/Tools/Migration/rodinia-nw-dpct",
    "file": "/home/u64609/oneAPI-samples/Tools/Migration/rodinia-nw-dpct/src/needle.cu"
  }
]
```

2. *dpct --cuda-include-path=/home/u64609/include -p compile_commands.json --in-root=. --out-root=migration*

warning: DPCT1003:0: Migrated API does not return error code. (*, 0) is inserted. You may need to rewrite this code.

warning: DPCT1043:1: The version-related API is different in SYCL. An initial code was generated, but you need to adjust it.

warning: DPCT1009:2: SYCL uses exceptions to report errors and does not use the error codes. The original code was commented out and a warning string was inserted. You need to rewrite this code.

...

warning: DPCT1049:5: The workgroup size passed to the SYCL kernel may exceed the limit. To get the device limit, query info::device::max_work_group_size. Adjust the workgroup size if needed.

Rodinia NW

- *cp Makefile migration/*
- Replace the CUDA configurations in that new `Makefile` with the following for use with DPC++:

```
CXX = dpcpp
TARGET = needleman_wunsch_dpcpp
SRCS = src/needle.dp.cpp
DEPS = src/needle_kernel.dp.cpp src/needle.h
```

- Compilation out-of-box fails with an error similar to the following:

```
error: assigning to 'int' from incompatible type 'typename info::param_traits<info::device,
(device)4143U>::return_type' (aka 'basic_string<char>’)
```

- Need to address warnings first

Addressing Warnings in Migrated Code

warning: DPCT1003:0: Migrated API does not return error code. (*, 0) is inserted. You may need to rewrite this code.

warning: DPCT1043:1: The version-related API is different in SYCL. An initial code was generated, but you need to adjust it.

warning: DPCT1009:2: SYCL uses exceptions to report errors and does not use the error codes. The original code was commented out and a warning string was inserted. You need to rewrite this code.

- remove unnecessary code processing error codes
- need to update the code with correct SYCL device API

needle.dp.cpp

```
int version = 0;
int err_code = 999;
/* ...dpct generated comments... */
err_code = (version = dpct::get_current_device().get_info<sycl::info::device::version>(), 0);
if (err_code != 0)
/* ...dpct generated comments... */
    printf("Error \\"%s\\" checking driver version: %s.\\n",
    "cudaGetErrorName not supported" /*cudaGetErrorName(err_code)*/,
    "cudaGetErrorString not supported" /*cudaGetErrorString(err_code)*/);
else
    printf("CUDA driver version: %d.%d\n", version/1000, version%1000/10);
```

```
std::string version = dpct::get_current_device().get_info<sycl::info::device::version>();
printf("SYCL device version: %s\n", version.c_str());
```

Addressing Warnings in Migrated Code

Check warning DPCT1049:

```
 /*
DPCT1049:5: The workgroup size passed to the SYCL kernel may exceed the limit.
To get the device limit, query info::device::max_work_group_size. Adjust the
workgroup size if needed.
*/
        q_ct1.submit([&](sycl::handler &cgh) {
            sycl::range<2> temp_range_ct1(17 /*BLOCK_SIZE+1*/,
                                          17 /*BLOCK_SIZE+1*/);
            sycl::range<2> ref_range_ct1(16 /*BLOCK_SIZE*/, 16 /*BLOCK_SIZE*/);
```

Once migration is completed, compile DPC++ code and run via make commands:

- *make*
- *make run*
- *./needleman_wunsch_dpcpp 4096 16*

  WG size of kernel = 128

  Start Needleman-Wunsch

  Processing top-left matrix

  Processing bottom-right matrix

Useful Links

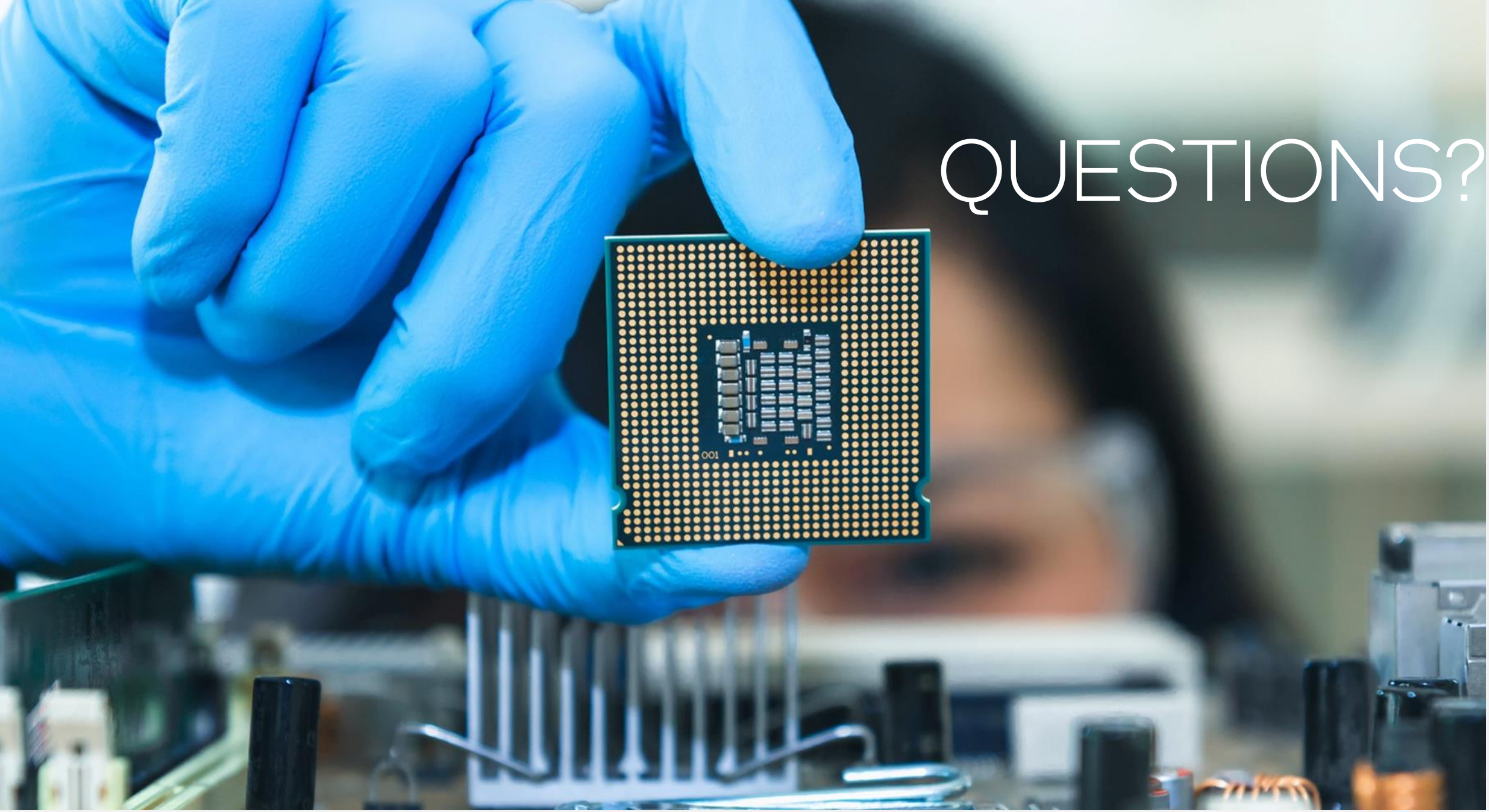
Open source projects

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:

tinyurl.com/dpcpp-tests

tinyurl.com/oneapi-samples


QUESTIONS?

Backup

DPC++ Function and Kernel

kernel.cpp

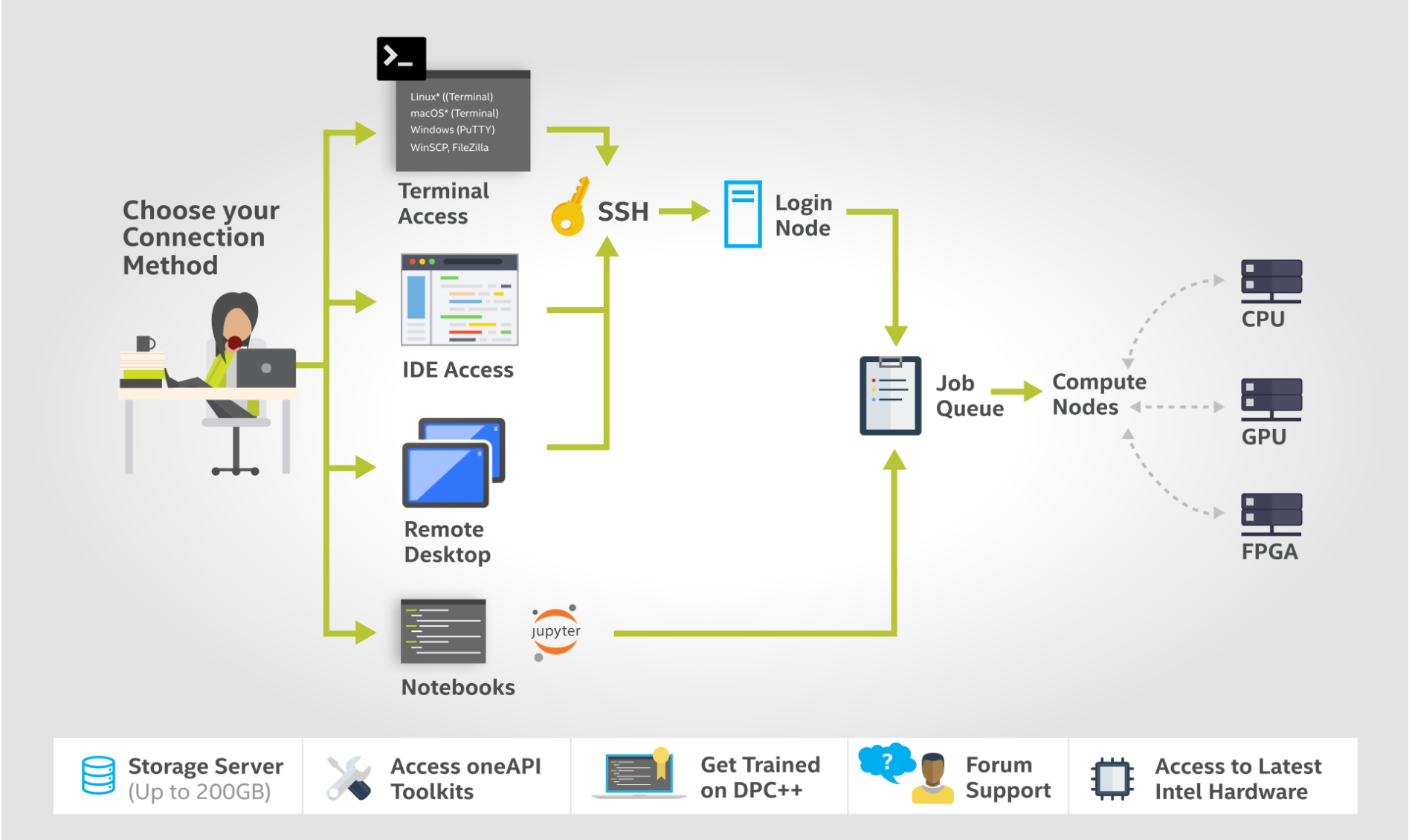
```
#include <CL/sycl.hpp>
using namespace sycl;

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;

 }
```

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

Intel technologies may require enabled hardware, software or service activation. Learn more at intel.com or from the OEM or retailer.

Your costs and results may vary.

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

Optimization Notice: Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice Revision #20110804. https://software.intel.com/en-us/articles/optimization-notice

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors.

Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. See backup for configuration details. For more complete information about performance and benchmark results, visit www.intel.com/benchmarks.

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

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

intel®