Advanced Programming CPU/GPU using SYCL

Soner Steiner

Intel certified oneAPI Instructor
VSC

Overview

- Introduction
- Remainder of the lambda functions
- Compilation and run
- Queues and device selectors
- Manage the data transfer
  Buffers and Unified Shared Memory
- Basic parallel kernels
- ND-Range kernels
- Sub-groups

Data Parallel C++

- Standard-based, Cross-architecture Language

DPC++

=

ISO C++

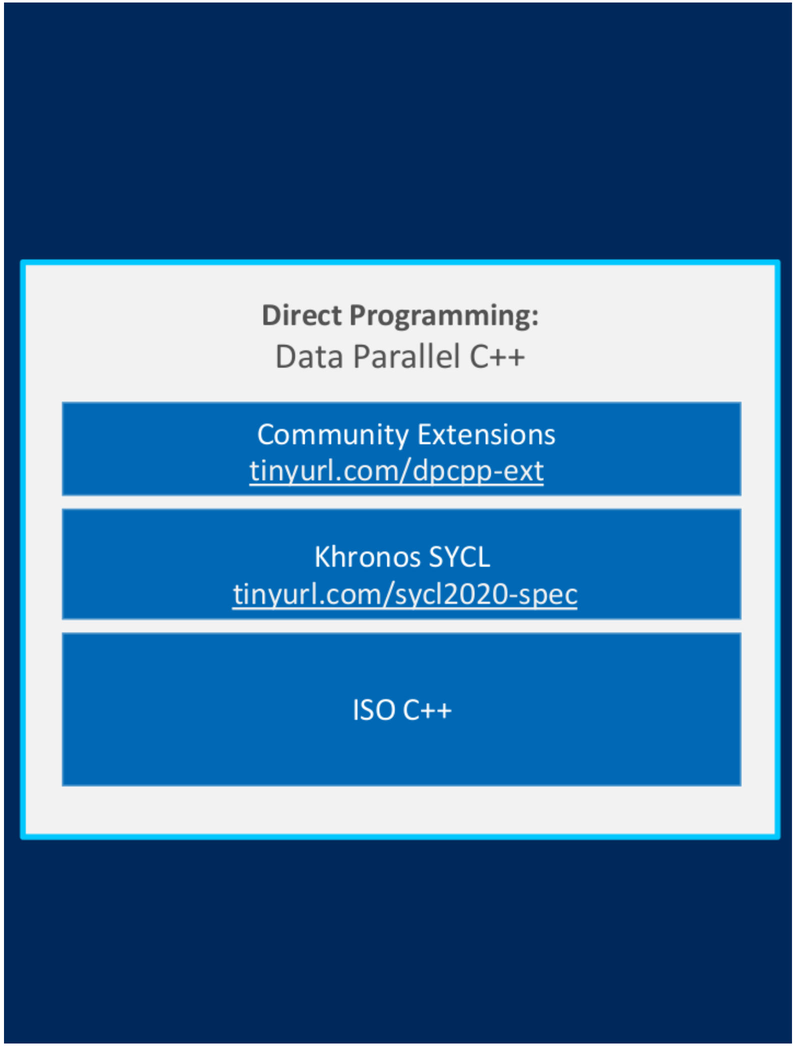
+

Khronos SYCL

+

Community Extensions

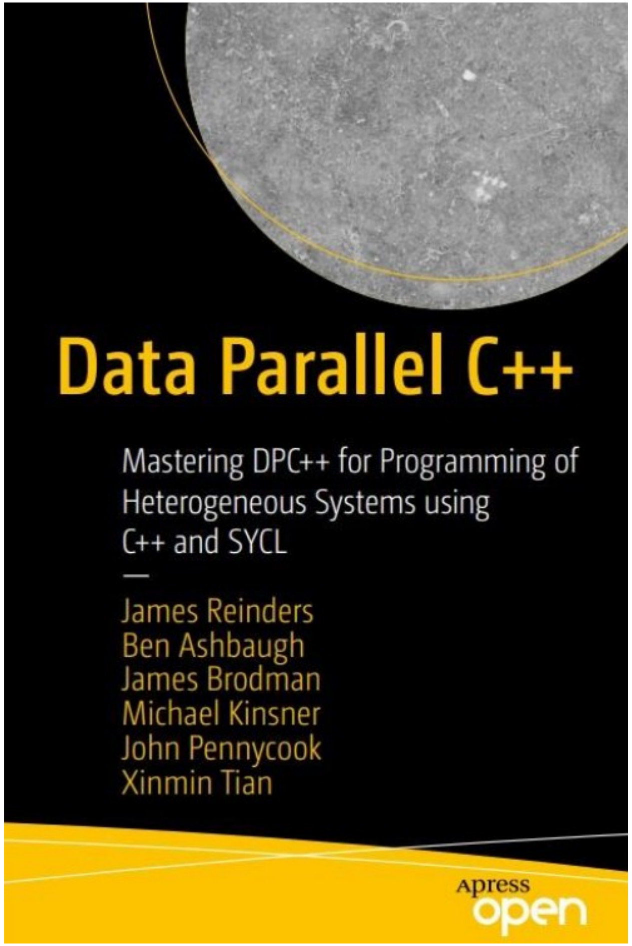
http://tinyurl.com/sycl2020-support-in-dpcpp



Many of the source examples are from Book:

- Source code accessible from
  $ oneapi-cl

```
Publication: Data Parallel C++
├──Chapter 01 - Introduction
├──Chapter 02 - Where Code Executes
├──Chapter 03 - Data Management
├──Chapter 04 - Expresssing Parallelism
├──Chapter 05 - Error Handling
├──Chapter 06 - Unified Shared Memory
├──Chapter 07 - Buffers
├──Chapter 08 - Scheduling Kernals and Data Movement
├──Chapter 09 - Communication and Synchronization
├──Chapter 10 - Defining Kernels
├──Chapter 11 - Vectors
├──Chapter 12 - Device Information
├──Chapter 13 - Practical Tips
├──Chapter 14 - Common Parallel Patterns
├──Chapter 15 - Programming for GPUs
├──Chapter 16 - Programming for CPUs
├──Chapter 17 - Programming for FPGA
├──Chapter 18 - Libraries
├──Chapter 19 - Memory Model and Atomics
└──Chapter 20 - Epilogue Future Direction
```



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

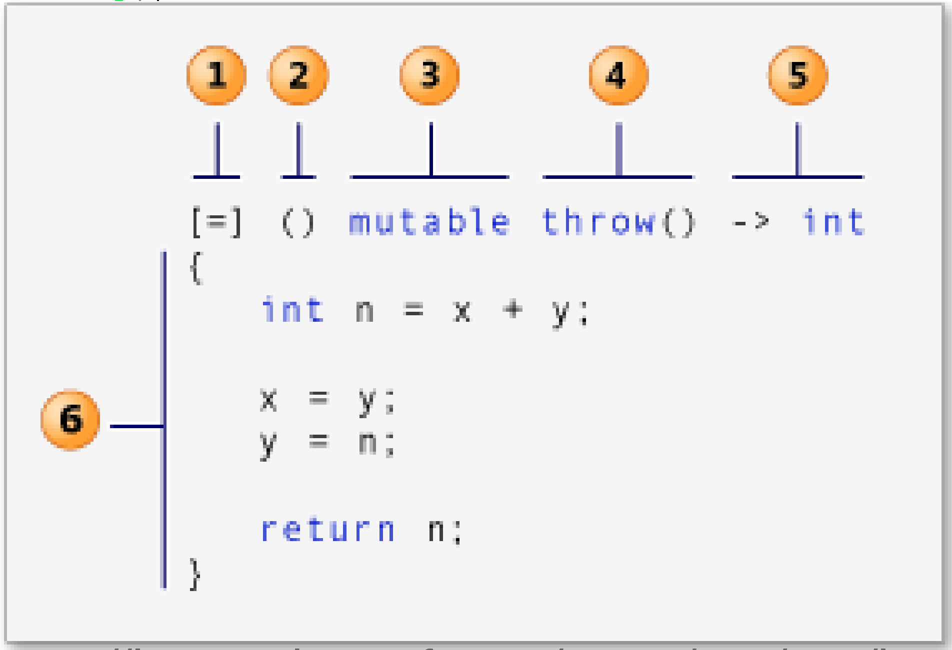
Command group scope

Device scope

Application scope

Lambda-functions ... Lambdas

```cpp
q.parallel_for(N, [=](auto i)
{
    a[i] -= 2;
});
```



1. capture clause
2. parameter list optional
3. mutable specification optional
4. exception-specification optional
5. trailing-return-type optional
6. lambda body

- [=] : capture by value
- [&] : capture by reference

https://learn.microsoft.com/en-us/cpp/cpp/lambda-expressions-in-cpp

!

Kernel Code

```
q.parallel_for(N, [=](auto i)
{
      a[i] -= 2;
});
```

- **Run Asynchronously**
- **Limitation on what kind of C++ code**

Kernel Code

Cannot use

these features

- Dynamic Polymorphism
- Dynamic memory allocations
- Static variables
- Function pointers
- Runtime Type Informatoion (RTTI)
- Exception Handling
- Recursion

Where Code Executes

- Queues
- Device Selectors

The queue class

Actions are submitted to a queue for execution on a single device

- Always bound to a single device
  - Q1 → GPU1
  - Q2 → CPU
  - Q3 → FPGA
  - Q4 → GPU

- Several queues can point to the same device
  - Q1 → GPU1
  - Q2 → GPU1
  - Q3 → GPU1
  - Q4 → CPU

Choosing Devices: Five use cases:

#	**Methods**	**Comments**
1	Anywhere (don’t care where)	Runtime chooses
2	Always on Host	Good for debugging
3	GPU or Accelerator	
4	Heterogeneous set of devices	
5	Specific Class of device	e.g. FPGA

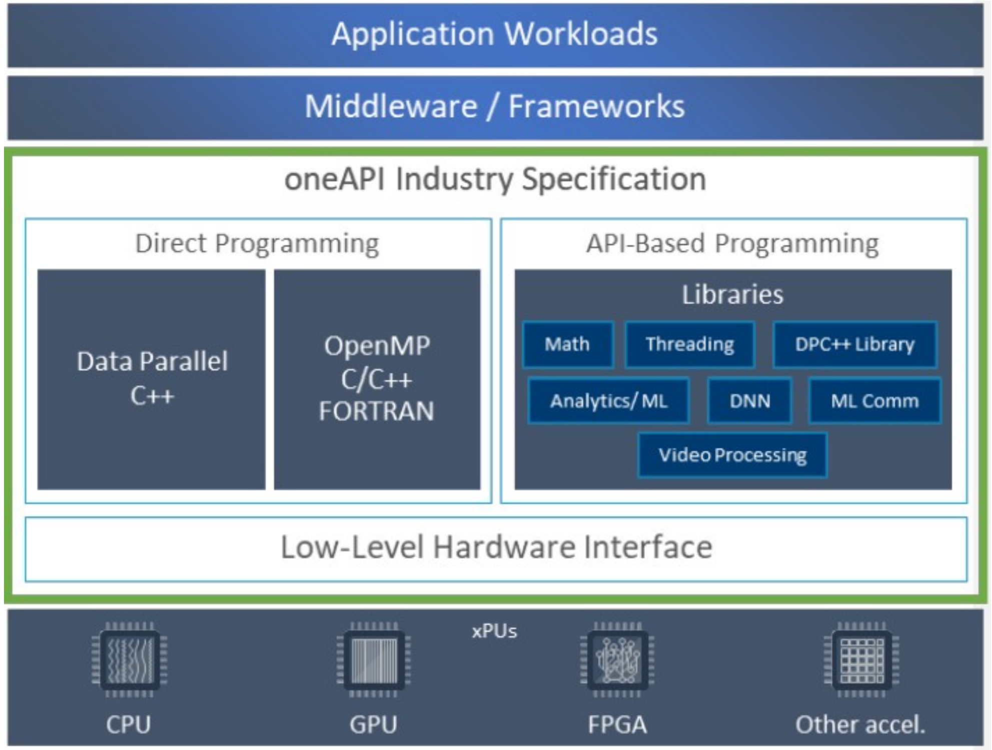
Actions

Work Type	Actions (handler class methods)	Summary
Device code execution	single_task	Execute a single instance of a device function.
	parallel_for	Multiple forms are available to launch device code with different combinations of work sizes.
	parallel_for_work_group	Launch a kernel using hierarchical parallelism, described in Chapter 4.
Explicit memory operation	copy	Copy data between locations specified by accessor, pointer, and/or shared_ptr. The copy occurs as part of the DAG, including dependence tracking.
	update_host	Trigger update of host data backing of a buffer object.
	fill	Initialize data in a buffer to a specified value.

J. Reinders et al., Data Parallel C++, download link

Programmers’ perspective: Three things to consider

1. Offload the code to device
2. Manage the transfer of Data
3. Implement Parallelism



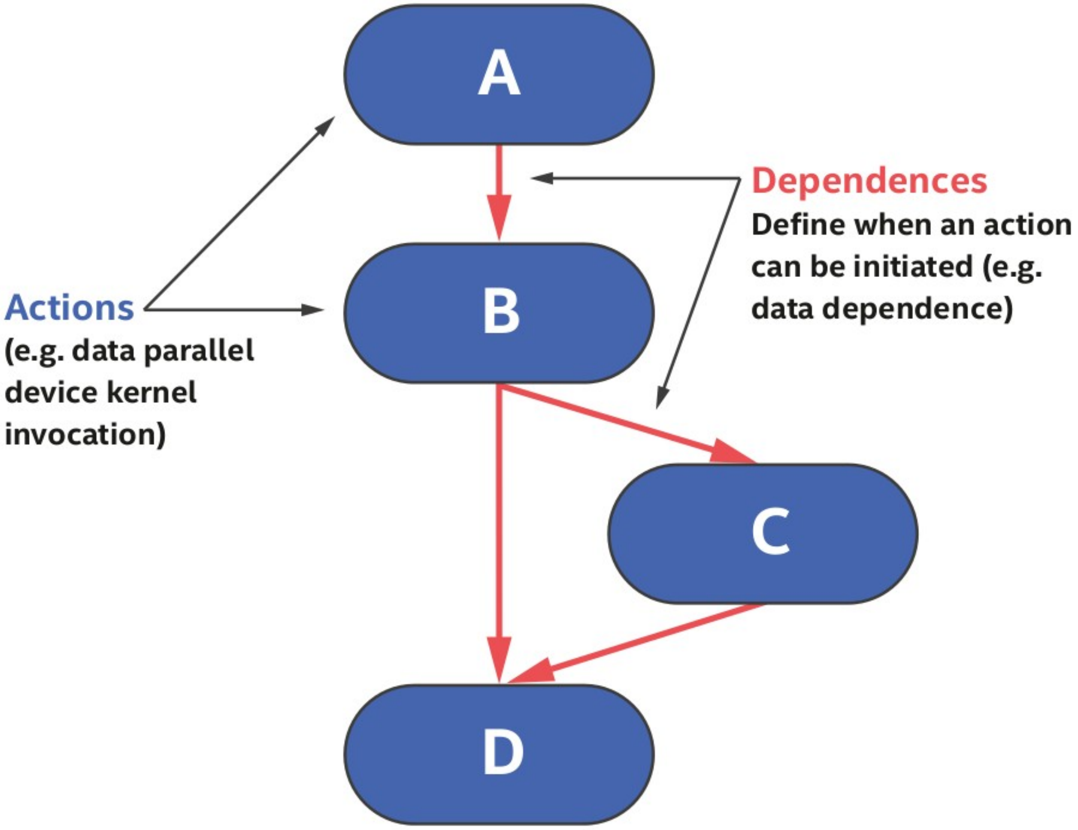
Memory Models

- Buffer Memory Model – abstract view of memory that can be local to the host or a device, and is accessible via accessors.

- Unified Shared Memory (USM)- pointer-based approach for memory model that os familiar for C++ programmers.

- Images: a special type of buffer that has an extra functionality specific to image processing

Task Graphs
(Directed Acyclic Graph)



- Dependency resolution and node execution are controlled by the runtime
- Dependencies determine the order that kernels are executed in
- Dependencies can be explicit or implicit

J. Reinders et al., Data Parallel C++, download link

Explicit Dependencies Using Events

```
constexpr int N = 101;
int main()
{
   queue q;
   int *data = malloc_shared<int>(N, q);

   auto e = q.parallel_for(N, [=] (id<1> i) { data[i] = i ;} );
   q.submit( [&] (handler &h)
     {
       h.depends_on(e);
       h.single_task([=] ()
           {
               for(int i = 1; i < N; ++i)
                   data[0] += data[i];
           } );
     } );
   q.wait();

   std::cout << "printing sum after computation \n" ;
   std::cout << data[0] << " ";
   std::cout << "\n" ;
}
```

- Create event to initialize the data in kernel1
- Kernel2 sums up the elements
- 5050

Buffer Memory Model

Buffers encapsulate data shared between host and device

Accessors provide access to data stored in buffers and create data dependencies in the graph.

Unified Shared Memory (USM) provides an alternative pointer-based mechanism for managing memory

```
queue q;
std::vector<int> v(N, 3);
{
    buffer buf(v);
    q.submit( [&] (handler& h)
     {
        accessor a(buf, h, write_only);
        h.parallel_for(N, [=] (auto i) { a[i] = i; } );
     } );

}

for (int i = 0; i < N; i++) std::cout << v[i] << " ";
```

Buffer Creation – two approaches

- Construct a new buffer using sycl::range to specify the size, data will not be initialized!

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

- Create buffer from existing data, data will be copied!

```
Buffer( T, hostData,
            const sycl::range<dimensions> &bufferRange,
            const sycl::property_list &proplist={} );
```

Examples of Buffer Creation

```
buffer b1{v};
buffer b2{v.begin(), v.end()};

// create a buffer of ints from std:array
std::array<int, 42> data;
buffer b3{data};

 // create a buffer of 5 doubles and initialize it from
 // a host pointer
double dd[5] = {1.1, 2.2, 3.14, 4.4, 5.5};
buffer b4{dd, range{5} };

std::cout << "printing v before computation \n" ;
for (int i = 0; i < N; i++) std::cout << v[i] << " ";
std::cout << "\n" ;
```

Buffer for vectors

Buffer for std::array

Buffer from a host pointer

Accessors

- Only means of accessing data in Buffers!
- They create the dependencies for the runtime.

Accessor Modes

Access Mode	Description
read_only	Read only Access
write_only	Write-only accessor Previous Contents not discarded
read_write	Read and Write access

Code Walkthrough

```
#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];
          });
      });
    }
```

Host Accessor
(up to now our accessors have been in the command group)

- The Host Accessor is an accessor which uses host buffer access target.
- Host accessors make data available for access on the host.
- They synchronize with the host by defining a new dependence between the currently accessing graph and the host.
- Creating host accessor is a blocking call.

Some Dependency Patterns

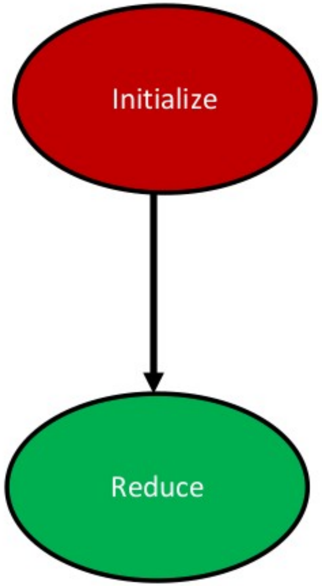
Linear Dependence Using In-order queue

Create In-order queue

Initialize the data in Kernel 1

Kernel 2 sums up the elements

```
constexpr int N=42;

int main()
{
    queue Q{property::queue::in_order()};

    int *data = malloc_shared<int>(N,Q);

    Q.parallel_for(N, [=](id<1> i) { data[i] = 1; });

    Q.single_task([=]()
      {
        for(int i=1; i < N; ++i)
            data[0] += data[i];
      });
    Q.wait();

    assert(data[0] == N);
    for(int i = 0; i < N; ++i)
        std::cout << data[i] << " ";
    std::cout  << "\n";
    return 0;
}
```

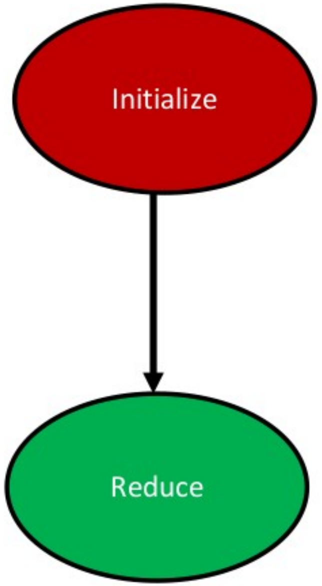


Linear Dependence Using Buffers and Accessors

Use Buffers and Accessors to Initialize the data in Kernel1

Kernel 2 sums up the elements

```
constexpr int N=101;
int main()
{
   queue q;
   buffer <int> data{ range{N} };

   q.submit( [&] (handler &h)
     {
       accessor a{data, h};
       h.parallel_for(N, [=] (id<1> i) { a[i] = i; } );
     } );

   q.submit( [&] (handler &h)
     {
      accessor a{data, h};
      h.single_task([=] ()
       {
           for(int i = 1; i < N; ++i)
               a[0] += a[i];
       } );
     } );
   host_accessor h_a{data};
   std::cout << h_a[0] << "\n";

   return 0;
}
```



5050

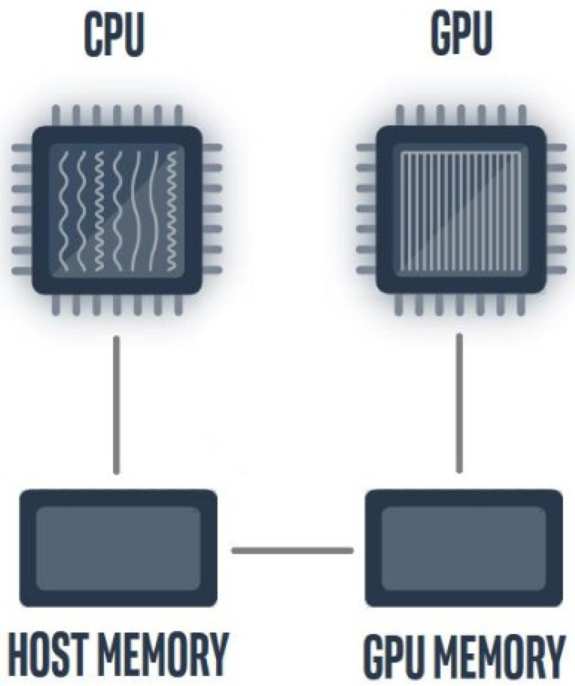
Unified shared memory (USM)

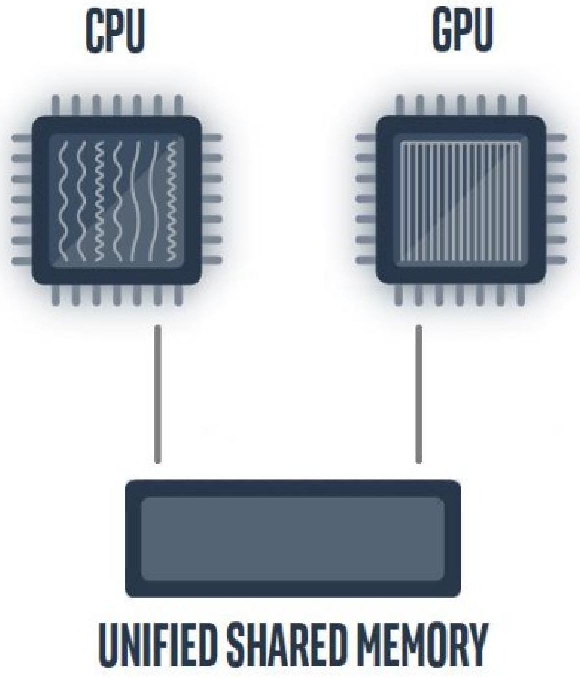
USM provides a pointer-based alternative in SYCL

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

Developer View of USM

Developers can reference the same memory object in host and device code with USM





Unified shared memory (USM)

USM 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[N];
int *deviceArray = (int*) malloc_device(N * sizeof(int), q);

for(int i = 0; i < N; ++i) hostArray[i] = i;

// copy hostArray tp deviceArray
q.memcpy(deviceArray, &hostArray[0], N*sizeof(int));
q.wait();

q.submit( [&] (handler &h)
  {
    h.parallel_for(N, [=] (auto ID)
        {
            deviceArray[ID] = ID*ID ;
        } );
  } );
  q.wait();

//copy deviceArray back to hostArray
q.memcpy(&hostArray[0], deviceArray, N*sizeof(int));
q.wait();
free(deviceArray, q);
```

malloc_device

mem_copy

mem_copy

USM – Implicit Data Movement

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

for(int i = 0; i < N; ++i) hostArray[i] = i;

q.submit( [&] (handler &h)
  {
    h.parallel_for(N, [=] (auto ID)
        {
            sharedArray[ID] = hostArray[ID] * hostArray[ID];
        } );
  } );
  q.wait();

for (int i = 0; i < N; i++) hostArray[i] = sharedArray[i] ;
free(hostArray, q);
free(sharedArray, q);
```

malloc_host
malloc_shared

USM – Data Dependency in Queues

No accessors in USM

Dependences must be specified explicitly using events

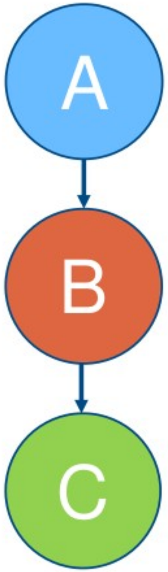
- queue.wait()
- wait on event objects
- use the depens_on method inside a command group

USM – Data Dependency in Queues

```
queue q;
int *data = (int*) malloc_shared(N * sizeof(int), q);
for(int i = 0; i < N; ++i) data[i] = i;

q.submit( [&] (handler &h)
  {
    h.parallel_for<class taskA>(range<1> (N), [=] (id<1> i)
      {
          data[i] += 1;
      } );
  } );
q.wait();
q.submit( [&] (handler &h)
  {
    h.parallel_for<class taskB>(range<1> (N), [=] (id<1> i)
      {
          data[i] += 2;
      } );
  } );
q.wait();
q.submit( [&] (handler &h)
  {
    h.parallel_for<class taskC>(range<1> (N), [=] (id<1> i)
      {
          data[i] += 3;
      } );
  } );
q.wait();

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

Explicit wait() used to ensure Data dependency is maintained

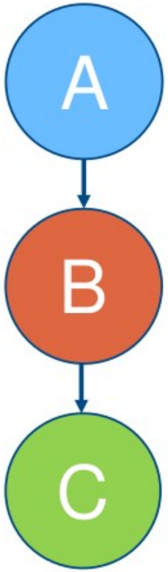
wait() will block execution on host



USM – Data Dependency in Queues

```
queue q;
int* data = malloc_shared<int>(N, q);

for(int i = 0; i < N; ++i) data[i] = i;

auto e1 = q.submit( [&] (handler &h)
  {
    h.parallel_for<class taskA>(range<1> (N), [=] (id<1> i)
        {
            data[i] += 1;
        } );
  } );
  auto e2 = q.submit( [&] (handler &h)
  {
    h.depends_on(e1);
    h.parallel_for<class taskB>(range<1> (N), [=] (id<1> i)
        {
            data[i] += 2;
        } );
  } );
// 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] += 3;
        } );
  } );
  q.wait();
std::cout << "printing data after computation \n" ;
for (int i = 0; i < N; i++) std::cout << data[i] << " ";
free(data, q);
```

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

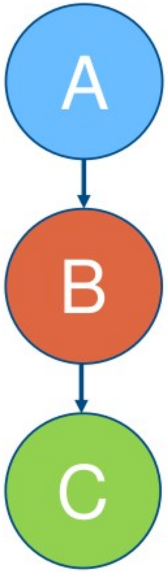


USM – Data Dependency in Queues

```
queue q{property::queue::in_order()};
int* data = malloc_shared<int>(N, q);

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

use in_queue property for the queue

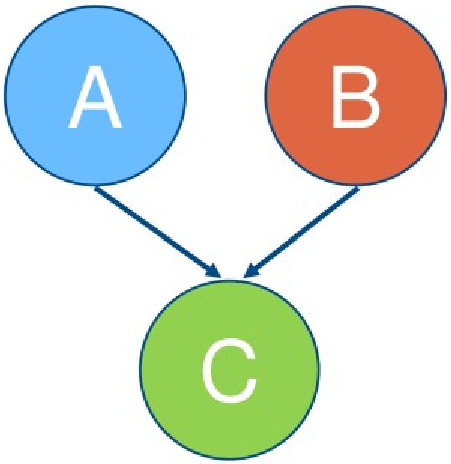
Execution will not overlap even
If the queues have no data dependency



USM – Data Dependency in Queues

```
queue q;
int *data1 = (int*) malloc_shared(N * sizeof(int), q);
int *data2 = (int*) malloc_shared(N * sizeof(int), q);

for(int i = 0; i < N; ++i){ data1[i] = 10; data2[i] = 20;}

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

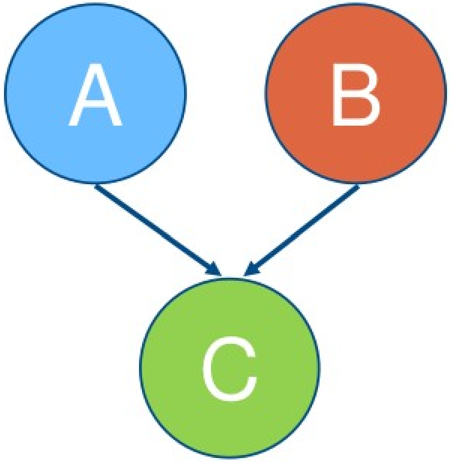
use depends_on() method to let command group handler know that specified events should be complete before specified tasks can execute.



USM – Data Dependency in Queues

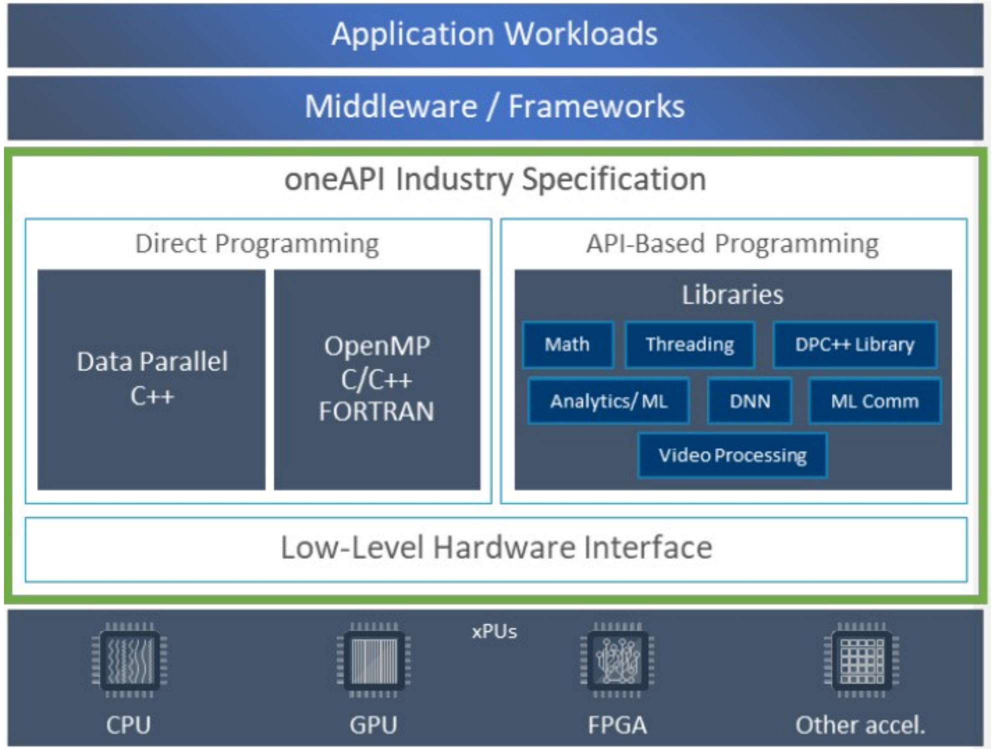
```
queue q;
int* data1 = malloc_shared<int>(N, q);
int* data2 = malloc_shared<int>(N, q);

for(int i = 0; i < N; ++i){ data1[i] = 10; data2[i] = 20;}

auto e1 = q.parallel_for<class taskA>(range<1> (N), [=] (id<1> i)
        {
            data1[i] += 1;
        } );
auto e2 = q.parallel_for<class taskB>(range<1> (N), [=] (id<1> i)
        {
            data2[i] += 2;
        } );
q.parallel_for<class taskC>(range<1> (N), {e1, e2}, [=] (id<1> i)
        {
            data1[i] += data2[i];
        } );
  q.wait();

free(data1, q); free(data2, q);
```

A more simplified way of specifying dependency as parameter of parallel_for



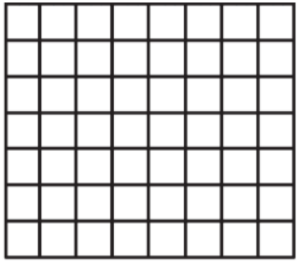
Programmers' perspective: Three things to consider

1. Offload the code to device
2. Manage the transfer of Data
3. Implement Parallelism

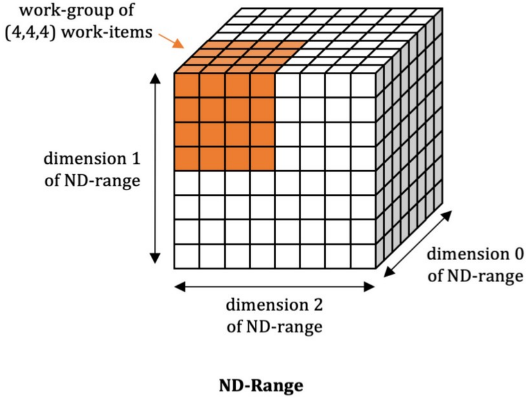


Three forms of Parallel Kernels

- **Basic** Parallel Kernels



- ND-range Parallel Kernels



- Hierarchical Parallel Kernels (‘Experimental alternative syntax’)

Basic Parallel Kernels

- Parallel kernel allows multiple instances of an operation to execute in parallel.
- Useful to offload parallel execution of a basic for-loop in which each iteration is completely independent and in any order.
- Parallel kernels are expressed using the parallel_for function.
- Up to the programmer to handle/confirm that there are no dependencies.

for-loop in CPU application

```
for(int i = 0; i < N; ++i)
{
    c[i] = a[i] + c[i] ;
}
```

Offload to a accelerator using parallel_for

```
h.parallel_for(range<1>(N), [=](id<1> i)
{
    C[i] = A[i] + B[i] ;
});
```

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

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

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>(N), [=](id<1> idx)
{
    //CODE THAT RUNS ON DEVICE
});
```

```
h.parallel_for(range<1>(N), [=](item<1> item)
{
    auto idx = item.get_id();
    auto R   = item.get_range();

    //CODE THAT RUNS ON DEVICE
});
```

Basic Parallel Kernels

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

- Dimensionality
  <1>, <2> or <3>
  is templated and must be declared at COMPILE time

- Size is dynamic – passed to constructor at runtime

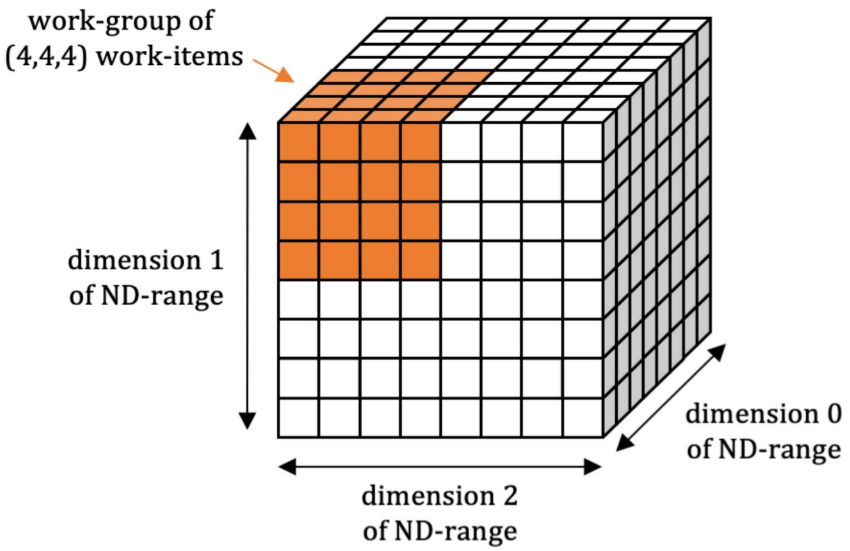
```
h.parallel_for(range<1>(N), [=](id<1> idx)
{
    //CODE THAT RUNS ON DEVICE
});
```

```
h.parallel_for(range<1>(N), [=](item<1> item)
{
    auto idx = item.get_id();
    auto R   = item.get_range();

    //CODE THAT RUNS ON DEVICE
});
```

ND-range Kernels

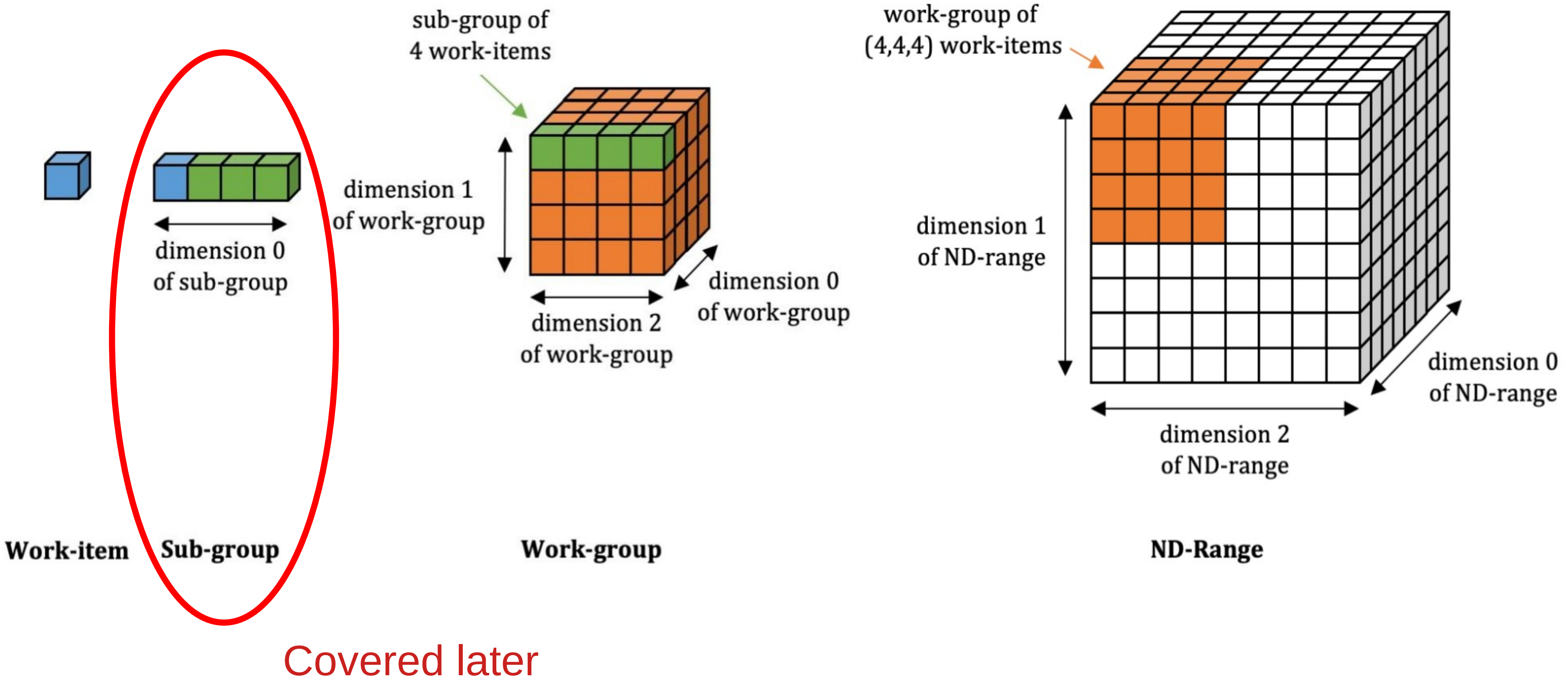
- ND-range kernels enable low level performance tuning by providing access to local memory and mapping executions to compute units on hardware.

- 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.



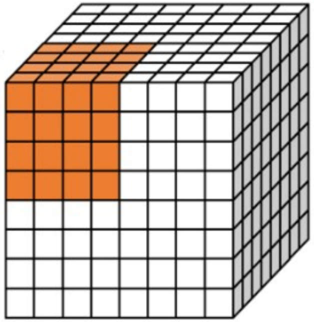
ND-Range

SYCL Thread Hierarchy and Mapping

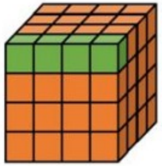


SYLC Thread Hierarchy and Mapping

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



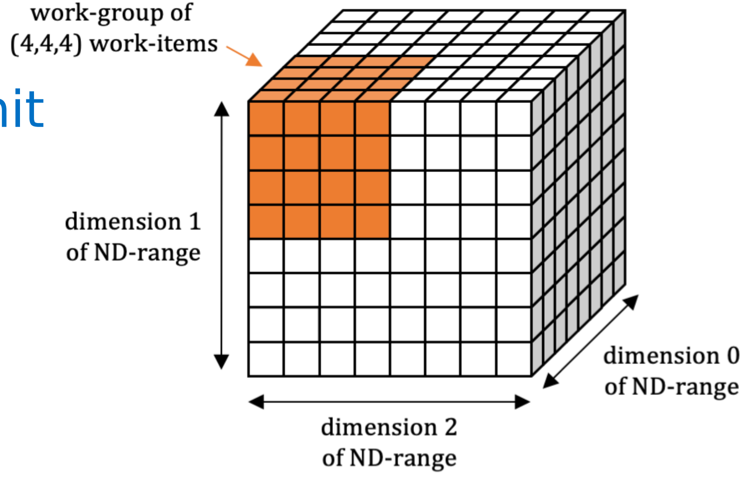
ND-range Kernels

- 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 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

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.

Sub-groups

Sub-groups

Understand how Sub-Groups map to GPU hardware

Understand how using Sub-Groups shuffle operations can achieve better performance and avoid repeated global memory access

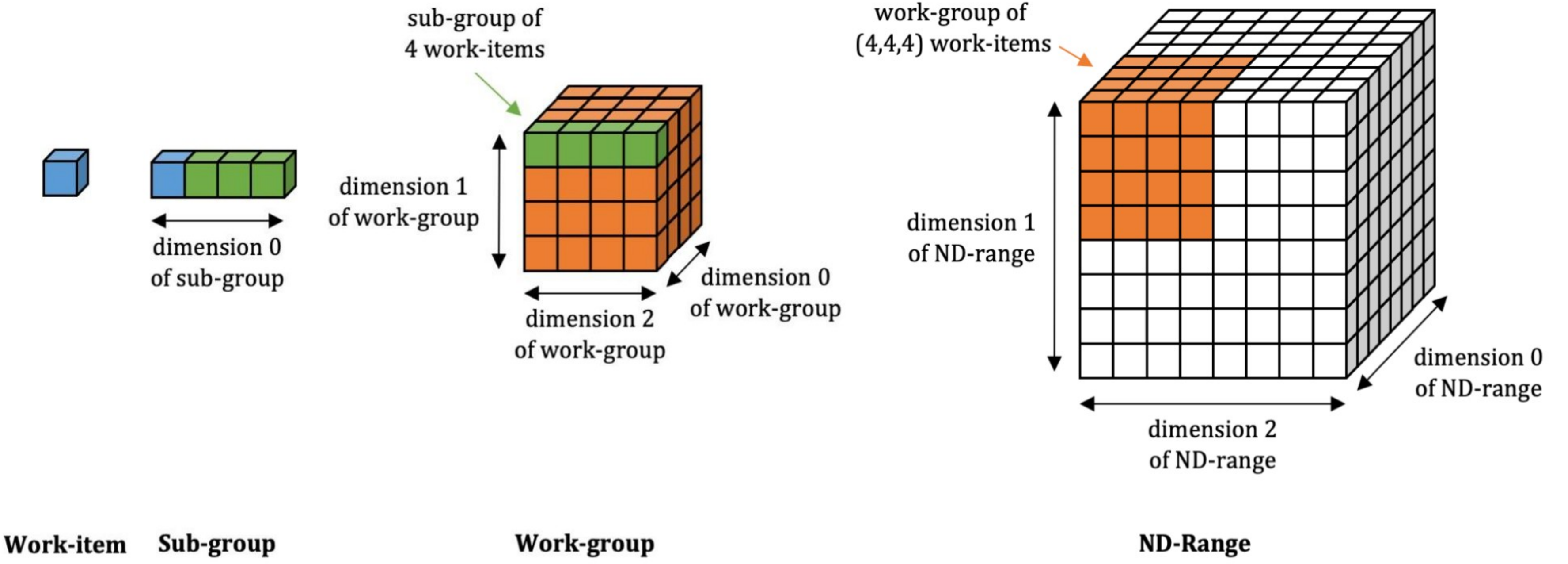
Write a SYCL program using Sub-Group and group algorithms to accomplish computation

Sub-groups

Sub-groups are a subset of the work-items that are executed Simultaneously or with additional scheduling guerantees.

Leveraging sub-groups will help to map execution to low level hardware and may help in achieving higher performance.

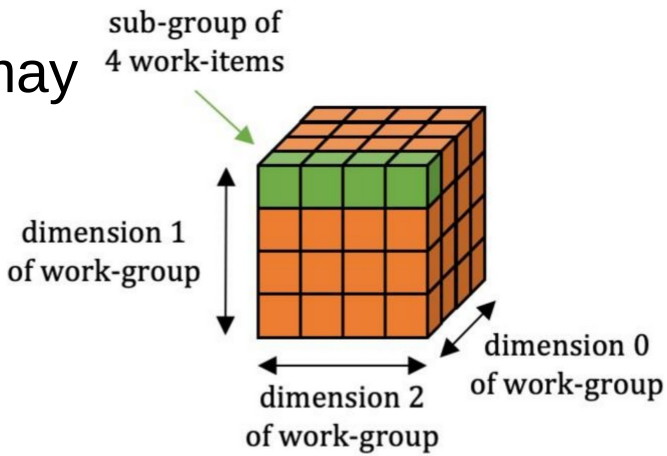
Sub-groups



sub-group of
4 work-items
work-group of
(4,4,4) work-items
dimension 0
of sub-group
dimension 1
of work-group
dimension 2
of work-group
dimension 0
of work-group
dimension 1
of ND-range
dimension 2
of ND-range
dimension 0
of ND-range
Work-item
Sub-group
Work-group
ND-Range

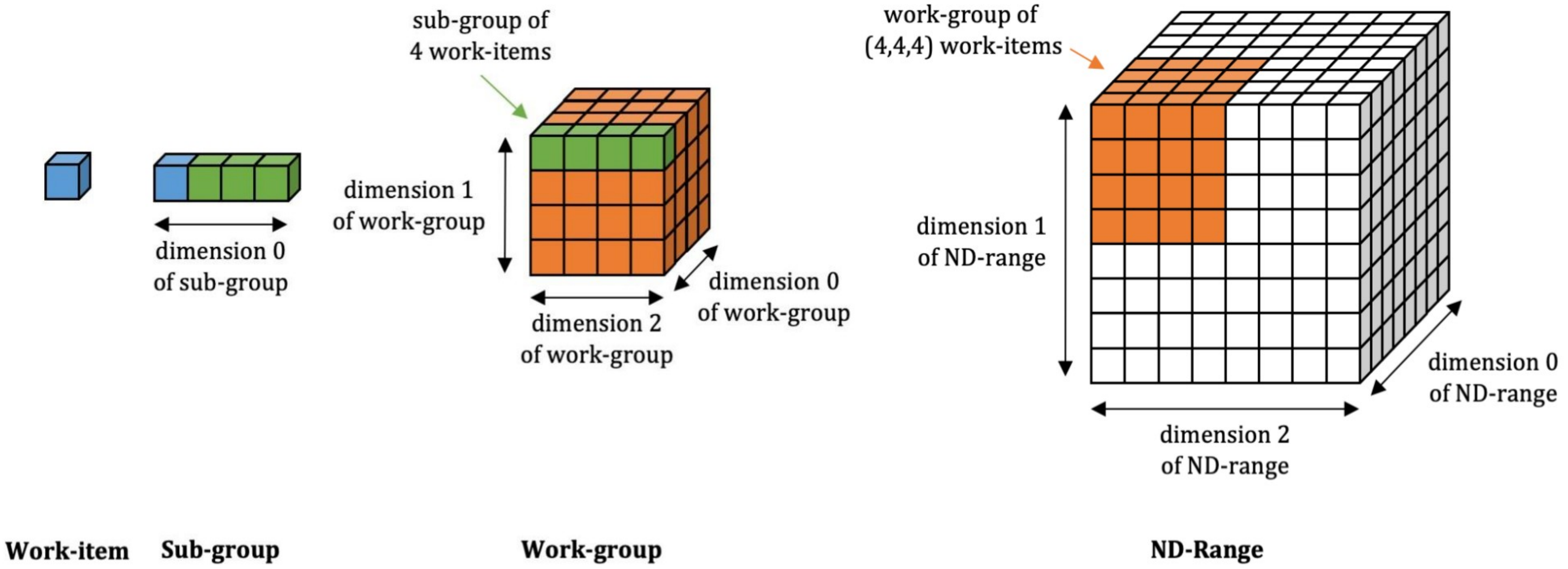
Sub-groups

- A subset of work-items withing 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
  - 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 = subset of work-items withing a work-group
- Parallel execution with ND-RANGE kernel helps to get access to work-group and sub-group



Sub-groups

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

sub_group class

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

- 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

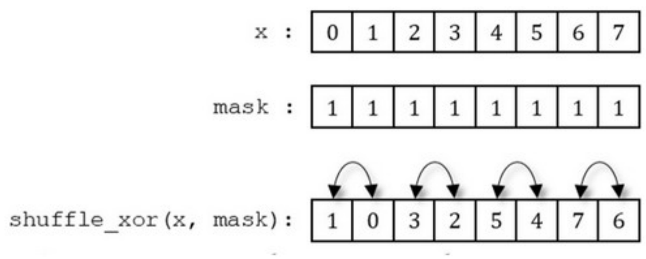
```
h.parallel_for(nd_range<1>(N,B), [=](nd_item<1> item){
        auto sg = item.get_sub_group();

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

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

Notices and Disclaimers

Back Up
Details about Intel® oneAPI Toolkits

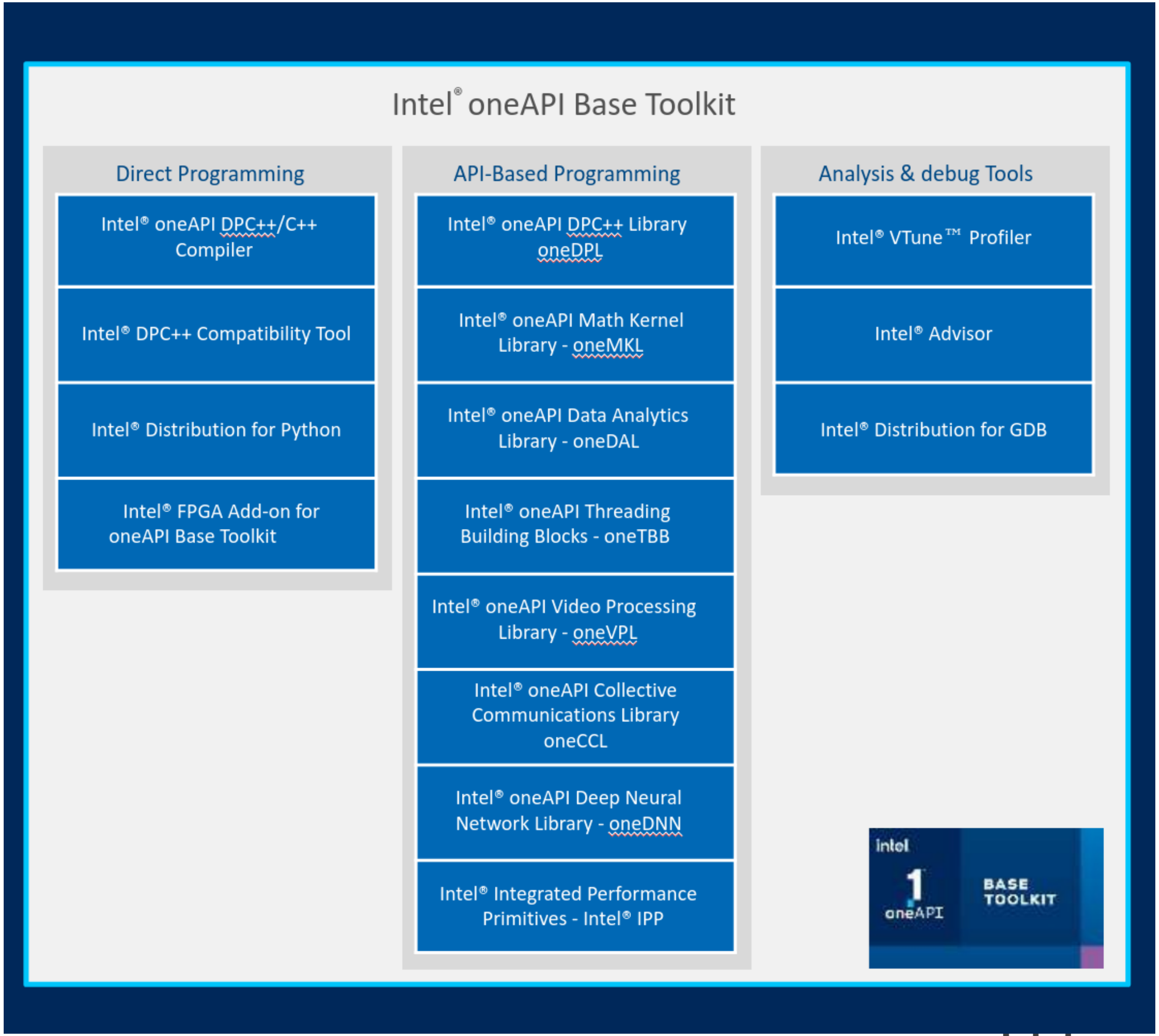
Intel® oneAPI Base Toolkit

Accelerate Data-centric Workloads

A core set of core tools and libraries for developing high-performance applications on Intel® CPUs, GPUs, and FPGAs.

Who Uses It?

- A broad range of developers across industries
- Add-on toolkit users since this is the base for all toolkits



Intel® oneAPI Base Toolkit

Accelerate Data-centric Workloads

Top Features/Benefits

- Data Parallel C++ compiler, library and analysis tools
- SYCLomatic / DPC++ Compatibility tool helps migrate CUDA code to C++ with SYCL
- Python distribution includes accelerated scikit-learn, NumPy, SciPy libraries
- Optimized performance libraries for threading, math, data analytics, deep learning, and video/image/signal processing

Learn More

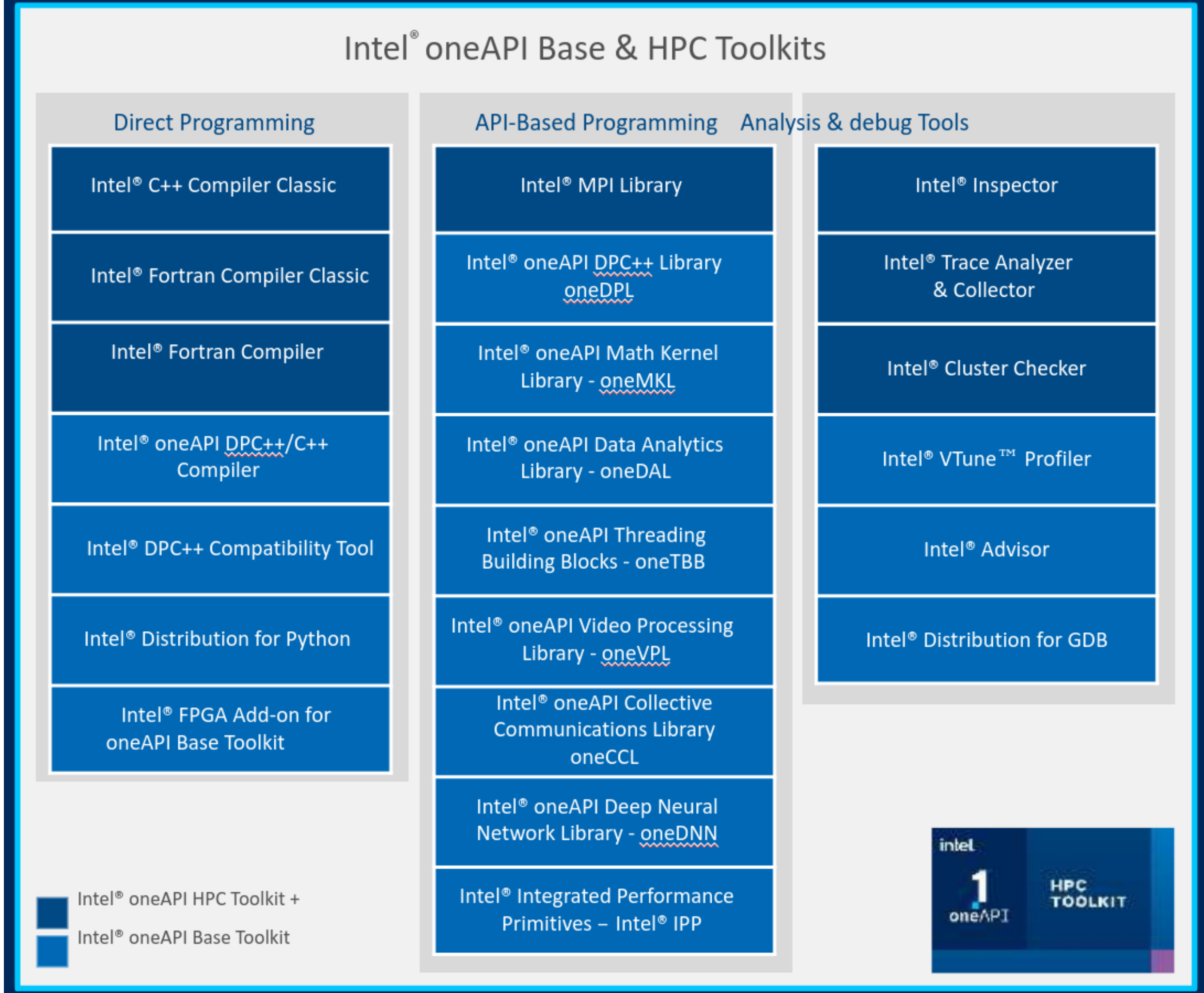
Intel® oneAPI HPC Toolkit

Accelerate Data-centric Workloads

A core set of core tools and libraries for developing high-performance applications on Intel® CPUs, GPUs, and FPGAs.

Who Uses It?

- A broad range of developers across industries
- Add-on toolkit users since this is the base for all toolkits



Intel® oneAPI HPC Toolkit

Accelerate Data-centric Workloads

Top Features/Benefits

- Data Parallel C++ compiler, library and analysis tools
- SYCLomatic / DPC++ Compatibility tool helps migrate CUDA code to C++ with SYCL
- Python distribution includes accelerated scikit-learn, NumPy, SciPy libraries
- Optimized performance libraries for threading, math, data analytics, deep learning, and video/image/signal processing

Summary

- oneAPI cross-architecture, one source programming model provides freedom of XPU choice.
  Apply your skills to the next innovation, not to rewriting software for the next hardware platform.

- Intel® oneAPI Toolkit products take full advantage of accelerated compute by maximizing performance across Intel CPUs, GPUs, and FPGAs.

- Develop confidently with a proven set of cross-architecture libraries and advanced tools that interoperate with existing performance programming models.

Intel® oneAPI HPC Toolkit

Accelerate Data-centric Workloads

Top Features/Benefits

- Data Parallel C++ compiler, library and analysis tools
- SYCLomatic / DPC++ Compatibility tool helps migrate CUDA code to C++ with SYCL
- Python distribution includes accelerated scikit-learn, NumPy, SciPy libraries
- Optimized performance libraries for threading, math, data analytics, deep learning, and video/image/signal processing