Direct Programming for CPU/GPU using SYCL

Europar24-Madrid

CGS

July 30, 2024

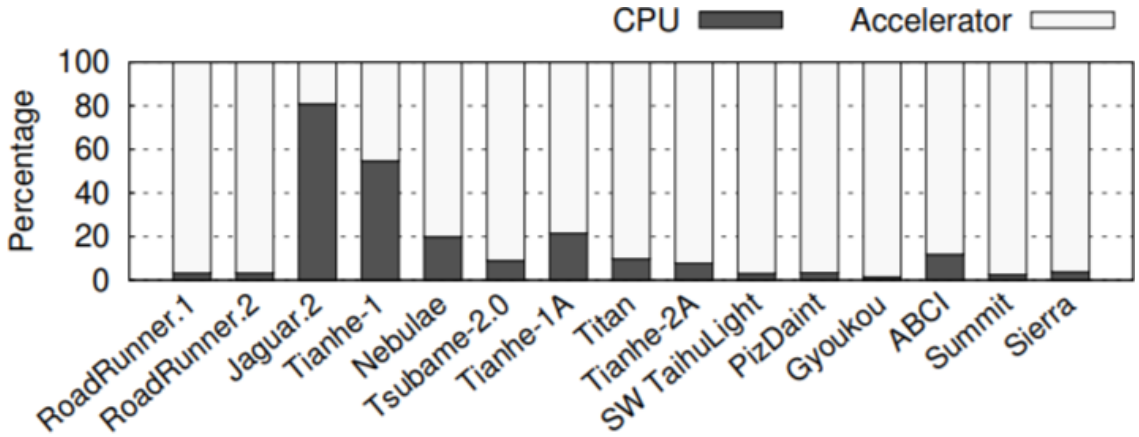
- "SYCL 2020 Specs", https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html
- "Data Parallel C++", https://link.springer.com/book/10.1007/978-1-4842-9691-2

Outline

Introduction

Motivation

- Heterogeneous-Computing (CPU+Accel) is one of the keys to improving performance in the **top500**[1]
  - The proportion of accelerators in the overall system performance for the 16 most powerful heterogeneous supercomputers.
  - Accelerators contribute 84% of the R_{peak} of the systems.



(a) Flops ratio.

[1] Extracted from *"An Analysis of System Balance and Architectural Trends Based on Top500 Supercomputers," HPC Asia 2021"*

Motivation

- Challenges: numerous programming models
  - More abstraction vs More performance

+++ Abstraction					+++ Performance
python	C/C++ Fortran	OpenMP (Cores)	OpenMP target OpenACC	OpenCL CUDA	Vector Intrs. GPU Intrinsics

Challenges

1. (Variety): Many languages with their toolchains, versions to maintain, and integrate
2. (Performance): Developing applications with high performance often requires programming expertise
3. (Porting): Some abstractions provide solutions for high performance on different architectures

Some Comparisons

- Some programming models for accelerators
  - CUDA: Proprietary model
  - Directives like OpenMP allow applying incremental programming techniques for heterogeneous systems (e.g., accelerator offloading, tasks...)
    - List of compilers supporting OpenMP-offload: OpenMP Compilers and Tools
  - SYCL: Multi-vendor, standard defined by the Khronos Group

	CUDA	OpenACC	OpenMP (5.0)	SYCL
Language	C/C++	C/C++ Fortran	C/C++ Fortran	C/C++
Prog. Style		pragmas	pragmas	C++11 lambdas
Parallelism	SIMT	SIMD, Fork/join CUDA	SPMD, SIMD Tasks, Fork/join, CUDA	OpenCL
Licensing	Proprietary	Few comp.	Open-source	Open-source
Abstraction	Low	High	High	Medium

Example: vector adding (C)

vectorAdd.c

```
// Compute vector sum C = A+B
void vecAdd(float* A, float* B, float* C,
   int n)
{
   for (i = 0, i < n, i++)
      C[i] = A[i] + B[i];
}

int main()
{
   // Memory allocation for A_h, B_h, C_h
   // I/O to read A_h and B_h, N elements
   ...
   vecAdd(A_h, B_h, C_h, N);
}
```

... porting for accelerators

- Rewriting for each kernel

Example: vector adding (CUDA)

vectorAdd.cu

```
// Compute vector sum C = A+B
__global__
void vecAddkernel(float* A_d, float* B_d, float* C_d, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}


int main()
{
   float* A_d, B_d, C_d;
   int size = n* sizeof(float);

   // A, B and C malloc and init
   ...

   // Get device memory for A, B, C
   // copy A and B to device memory
   cudaMalloc((void **) &A_d, size);
   cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
   cudaMalloc((void **) &B_d, size);
   cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

   // Kernel execution in device
   // (vector add in device)
   dim3 DimBlock(256, 1, 1);
   dim3 DimGrid(ceil(n/256.0), 1, 1);
   vecAddkernel<<<DimGrid,DimBlock>>>(A_d, B_d, C_d, n);

   // copy C from device memory
   // free A, B, C
   cudaFree(A_d); cudaFree(B_d); cudaFree (C_d);
}
```

Ejemplo: Example: vector adding (OpenCL)

add_vector_kernel.cl

```
// many instances of the kernel,
// called work-items, execute in parallel
__kernel void add_vector_kernel(__global const float *a,
    __global const float *b, __global float *c)
{
  int id = get_global_id(0);
  c[id] = a[id] + b[id];
}
```

Example: vector adding (OpenCL)

```
// create the OpenCL context on a GPU device
cl_context context = clCreateContextFromType(0,
        CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);

// get the list of GPU devices associated with context
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &cb);

cl_device_id[] devices = malloc(cb);
clGetContextInfo(context,CL_CONTEXT_DEVICES,cb,devices,NULL);
```

DEFINIR PLATAFORMA

```
// create a command-queue
cmd_queue = clCreateCommandQueue(context,devices[0],0,NULL);

// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcA, NULL);
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcb, NULL);

memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                sizeof(cl_float)*n, NULL, NULL);
```

OBJETOS MEMORIA

```
// create the program
program = clCreateProgramWithSource(context, 1,
                &program_source, NULL, NULL);
```

CREAR PROGRAMA

```
// build the program
err = clBuildProgram(program, 0, NULL,NULL,NULL,NULL);
```

CREAR PROGRAMA

```
// create the kernel
kernel = clCreateKernel(program, “vec_add”, NULL);

// set the args values
err  = clSetKernelArg(kernel, 0, (void *) &memobjs[0],
                sizeof(cl_mem));
err |= clSetKernelArg(kernel, 1, (void *) &memobjs[1],
                sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *) &memobjs[2],
                sizeof(cl_mem));
```

CREAR KERNEL Y ENVÍO DATOS

```
// set work-item dimensions
global_work_size[0] = n;

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1, NULL,
            global_work_size, NULL,0,NULL,NULL);
```

EJECUCION KERNEL

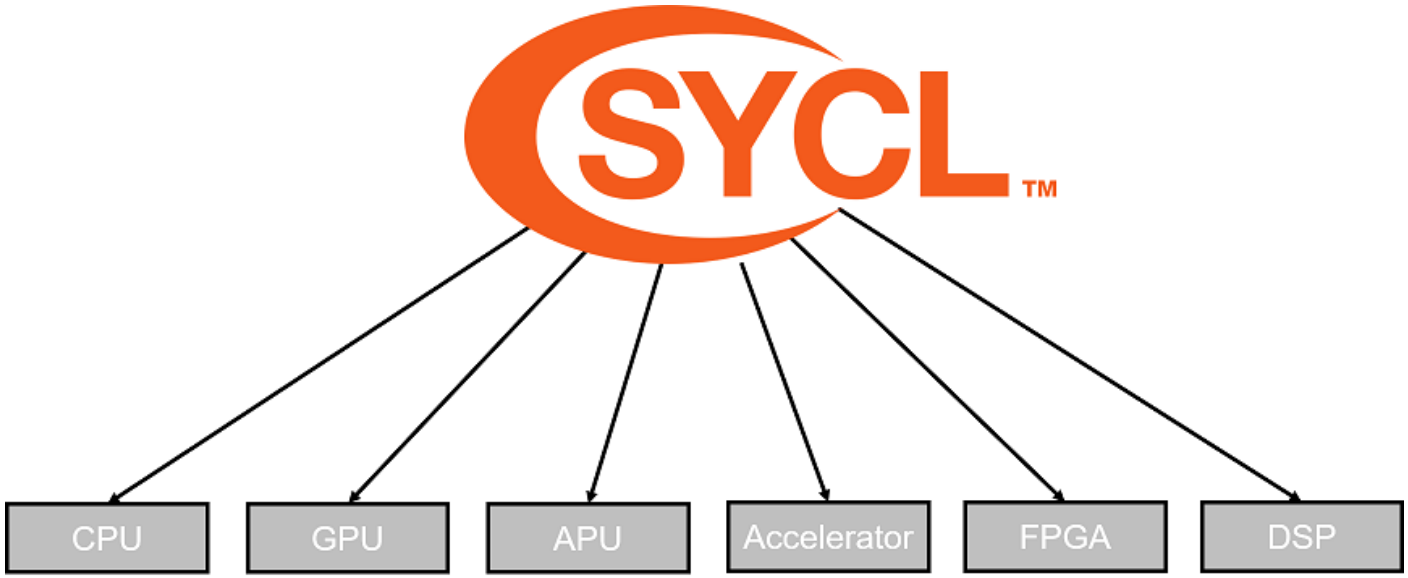
```
// read output array
err = clEnqueueReadBuffer(cmd_queue, memobjs[2],
                CL_TRUE, 0,
                n*sizeof(cl_float), dst,
                0, NULL, NULL);
```

ENVIO RESULTADOS HOST

SYCL

What's SYCL?

- SYCL is a high-level standard C++ programming model
- Single source code that can target a **variety of heterogeneous platforms**



What's SYCL?

- Based on OpenCL: reuses the same architecture concepts
  - Platform, device, queue, NDRange, work-group, work-item...

Overview

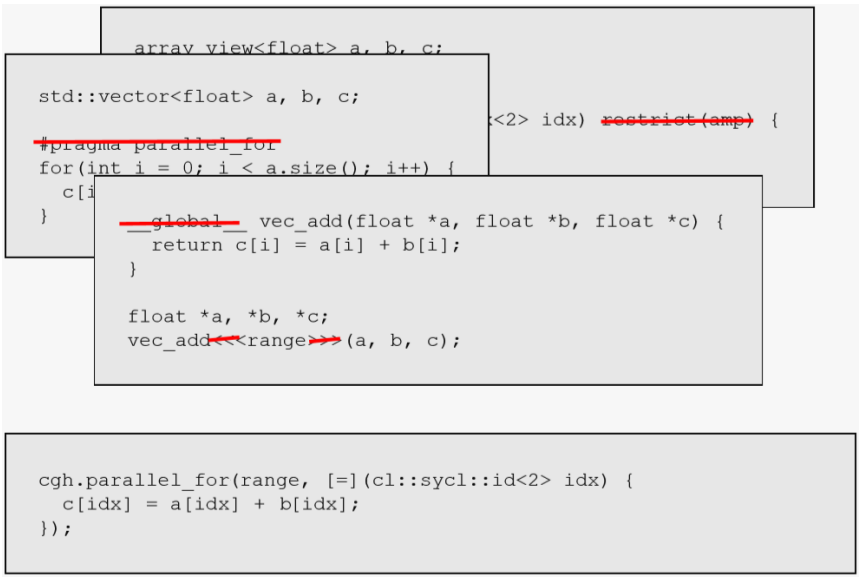
"... royalty-free, cross-platform abstraction layer that builds on the underlying concepts, portability and efficiency of OpenCL that enables code for heterogeneous processors to be written in a single-source style using completely standard C++"
– SYCL Specification –

What's SYCL?

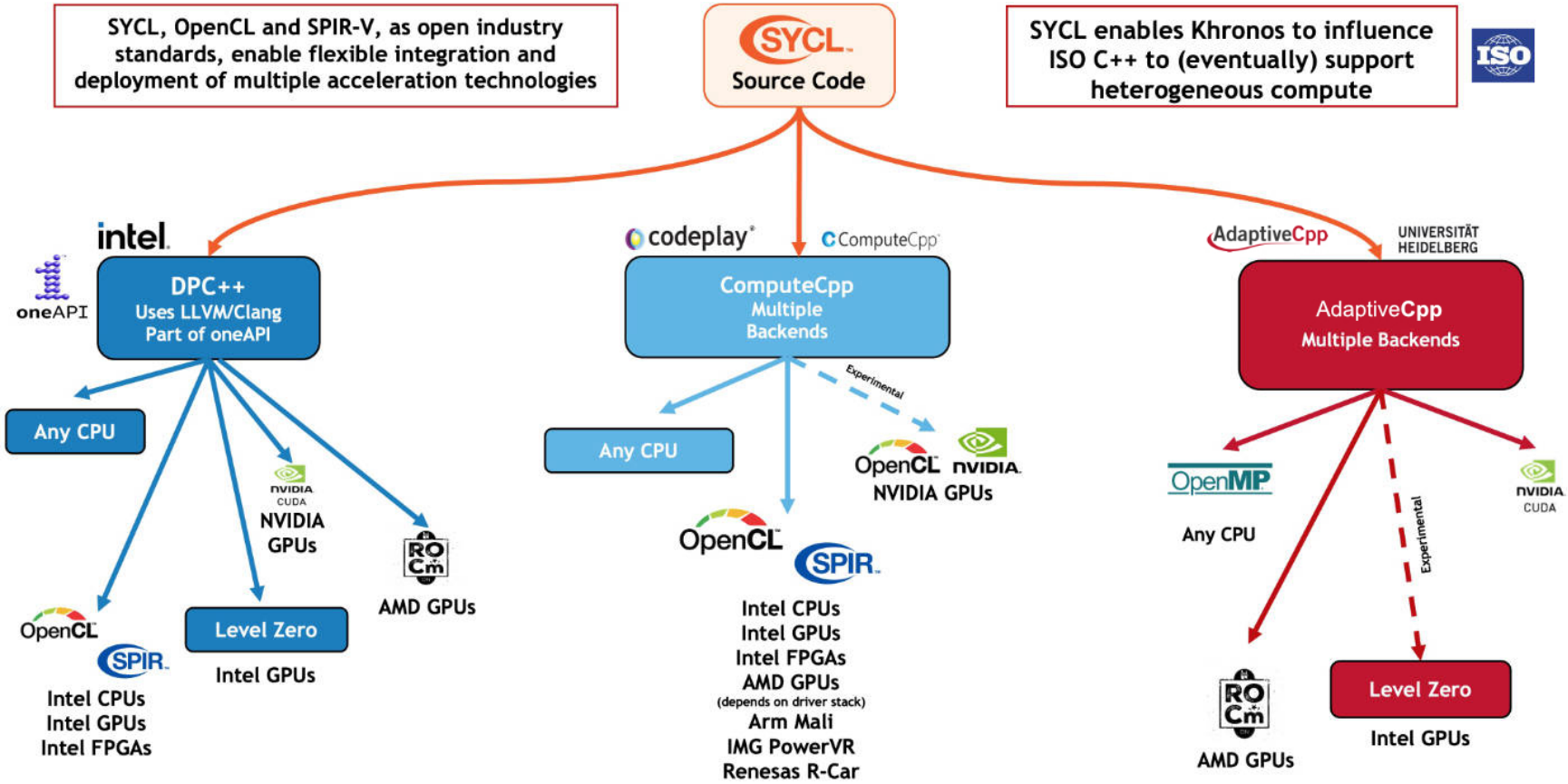
- SYCL provides high-level abstractions over common repetitive code
  - Platform/device selection
  - Buffer creation and data movement
  - Kernel function compilation
  - Dependency management and programming

SYCL

- SYCL allows you to write standard C++
  - SYCL 2020 is based on C++17
- Unlike the other implementations shown on the left, there are:
  - No language extensions
  - No pragmas
  - No attributes



SYCL Implementations

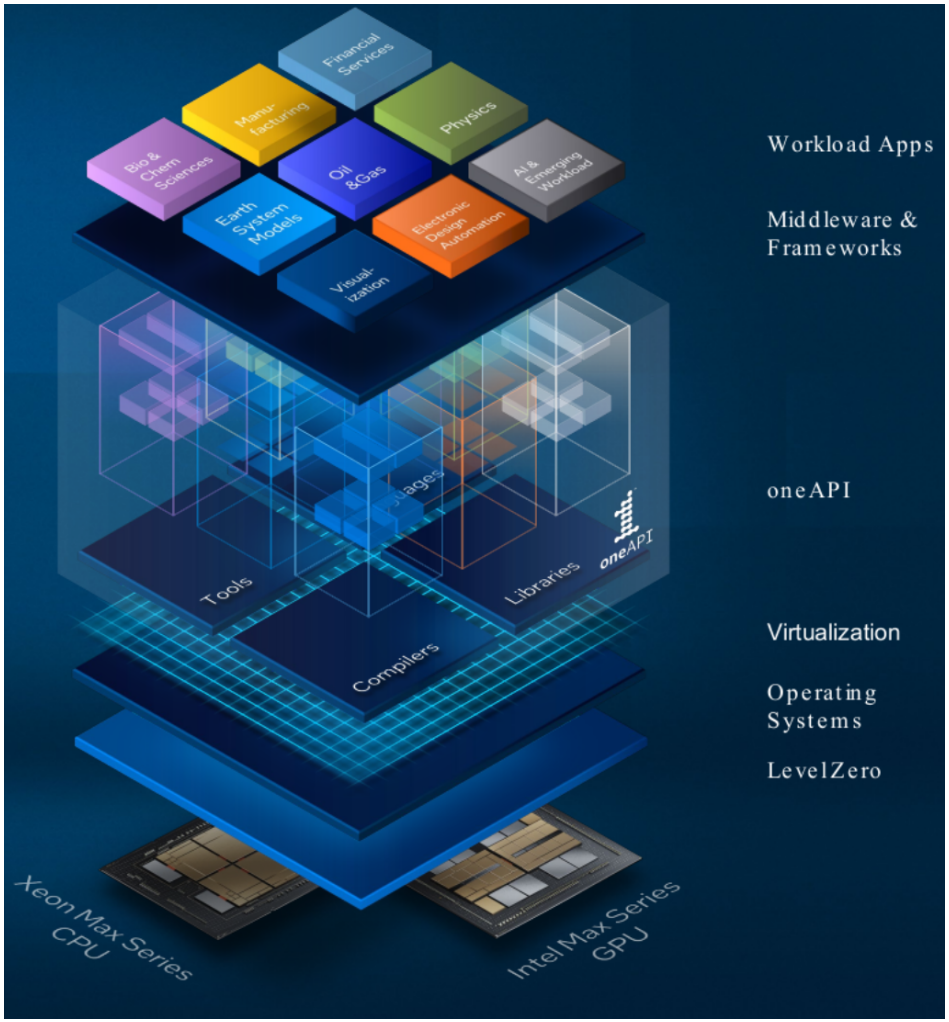


What's Data Parallel C++

- Data Parallel C++ ⇔ DPC++
  - C++ and SYCL standard with some extensions
- Based on C++
  - Productivity benefits by supporting C++ constructs
- Incorporates the SYCL standard
  - With support for data parallelism and heterogeneous programming

Intel oneAPI

- Unified programming model: diverse architectures
- Optimized language and libraries
- Performance equivalent to high-level native language
- Based on industry standards and open specifications
- Compatible with existing HPC programming models

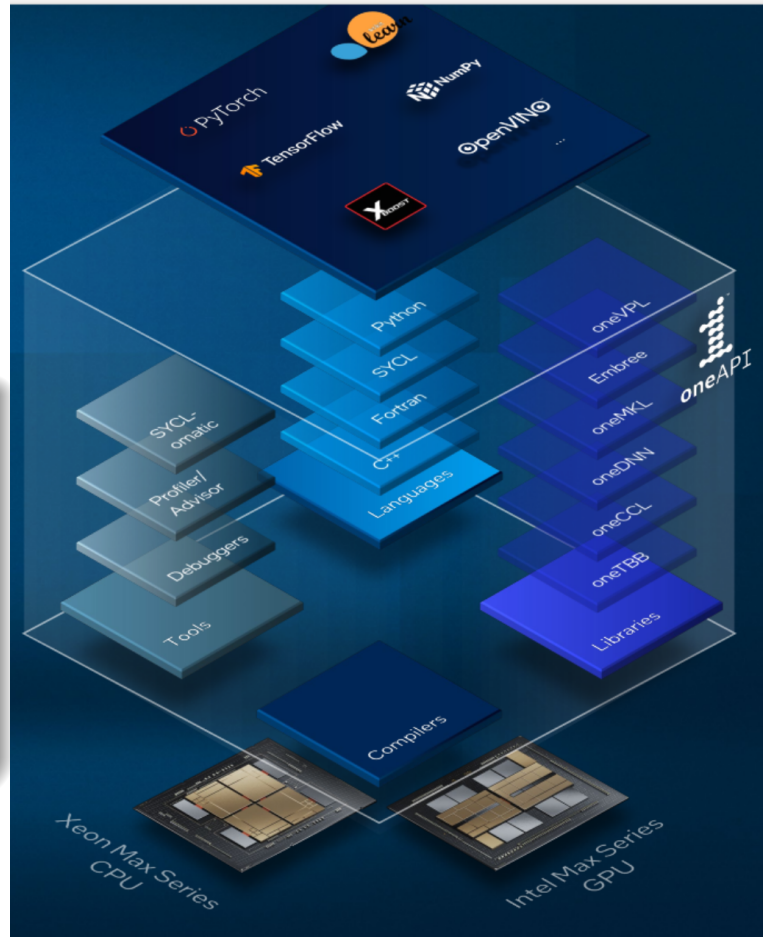


Intel oneAPI

- Standard-based language: C++ and SYCL
- Powerful APIs for accelerating domain-specific functions

Solution to unique provider

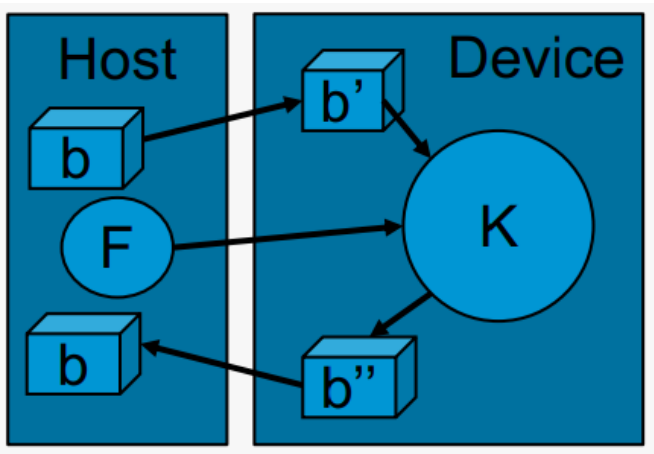
- Open standard to promote community and industry support
- Allows code reuse across different architectures and providers



SYCL Pseudo-code

pseudo_code.cpp

```
selector = default_selector()
q = queue(selector)
b = buffer (double, 1000)
q.submit (
    F ( ) {
        a = accessor(b, READ_AND_WRITE)
        K (i) {
            a[i] = a[i] * 2
        }
    }
)
q.wait_and_throw()
a2 = accessor(b, READ_ONLY)
printf("%f", a2[0])
```



Queues

Queue Types

- In SYCL, all work is submitted through commands to a queue
- **Queue** has an **associated device** to which all queued commands will be directed
  - A series of commands represented in the *command group* is executed in the **queue**

type	Device
default_selector_v	Selects any device or host device if no device can be found.
gpu_selector_v	Select a GPU
accelerator_selector_v	Select an accelerator
cpu_selector_v	Select a CPU device
my_device_selector	*Custom selector*

Queues example

queue_device.cpp

```
int main() {
  sycl::device d;

  d = sycl::device(sycl::gpu_selector_v);
  std::cout << "Using " << d.get_info<sycl::info::device::name>();

  sycl::queue Q(d);

  Q.submit([&](sycl::handler &cgh) {
    // Create a output stream
    sycl::stream sout(1024, 256, cgh);
    // Submit a unique task, using a lambda
    cgh.single_task([=]() {
      sout << "Hello, World!" << sycl::endl;
    }); // End of the kernel function
  });  // End of the queue commands. The kernel is now submited

  // wait for all queue submissions to complete
  Q.wait();
}
```

C++ Reminder

- *Lambda* function is common in many languages, including C++
  - Dynamically creates a function at runtime that can capture variables from its scope when invoked
- Syntax: [capture](parameters){body}
  - [] means empty capture
  - [&] captures by reference (entire scope)
  - [=] captures local variables by value
  - [a, &b] captures variables explicitly

```
void func (std::function<void(int)> f) { int x = 6; f(x); }
{
    int a = 6;
    func([&](int q){ if (q == a) std::cout << ""success; });
}
```

C++ Reminder

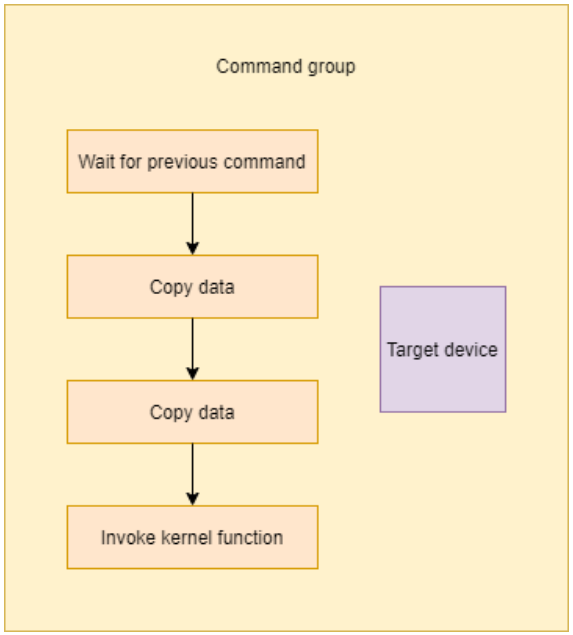
- In the following example, the function f has access to the entire scope of the main function (including variables b and c by reference)
  - The function f can modify the contents of b and c
  - It would print 121, 11, 101 to the screen

lambda.cpp

```
int main() {
    int b = 10;
    int c = 100;
    auto f = [&](int& a) -> int {
        b += 1;
        c +=1;
        return a + b + c;
    };
    int a = 10;
    std::cout << f(a) << std::endl;
    std::cout << b << std::endl;
    std::cout << c << std::endl;
    return 0;
}
```

Command Group

- A *command group* represents a series of **commands** to be executed on the device
  - Invokes the kernel function
  - Copies data to/from the device
  - Waits for other commands to complete



Queues

- The **submit** function of the *command group* takes a C++ object referenced by a handler
  - The object can be a **lambda expression** or a class with an operator function
  - The **body** of the object function represents the function in the *command group*

```
class my_kernel;

gpuQueue.submit([&](handler &cgh){

  cgh.single_task<my_kernel>([=]() {
    /* kernel code */
  });
});
```

Hands-on

1. Connect to the Intel Developer Cloud
2. In the **Training and Workshops** section
3. ... but we will work on **Essentials of SYCL**
   - https://console.cloud.intel.com/training/detail/9b8933a6-b466-4c69-8217-9a09d084a55a
4. Inside the IDC, open the notebook in the path **Training/HPC/oneapi-essentials-training/01_oneAPI_Intro/oneAPI_Intro.ipynb**

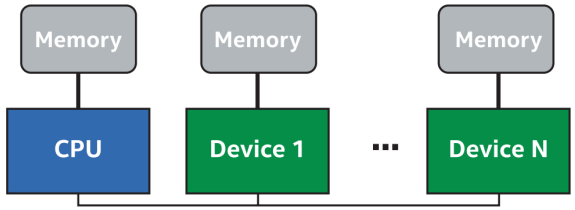
Important class in SYCL

Class	Functionality
sycl::device	Represents the device: CPU, GPU, FPGA... where the SYCL kernel is executed
sycl::queue	Represents a queue where the kernel is sent (enqueued) Multiple queues can map to the same `sycl::device`
sycl:buffer	Encapsulates a memory allocation that can be transferred between host and the device at runtime
sycl::handler	Handler. Used to define a scope of the command group. Connects buffer and kernel concepts
sycl::accessor	Used to define access requirements for kernels (e.g., read, write, read-write)
sycl::range, sycl::nd_range sycl::id, sycl::item, sycl::nd_item	Represents execution ranges and parallelism in execution

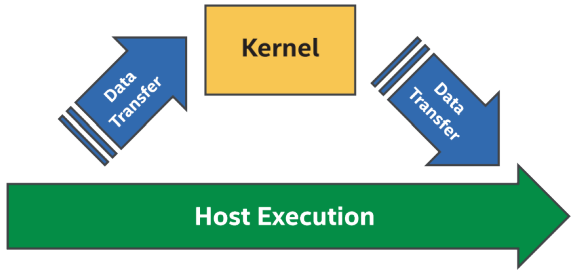
Memory Model

Memory

- The memory model is based on several memories: devices and host but there are separatedd



- Data is often transferred from host memories to devices ones so that it can be accessed from kernels



Data Handling

- In SYCL, there are two memory models:

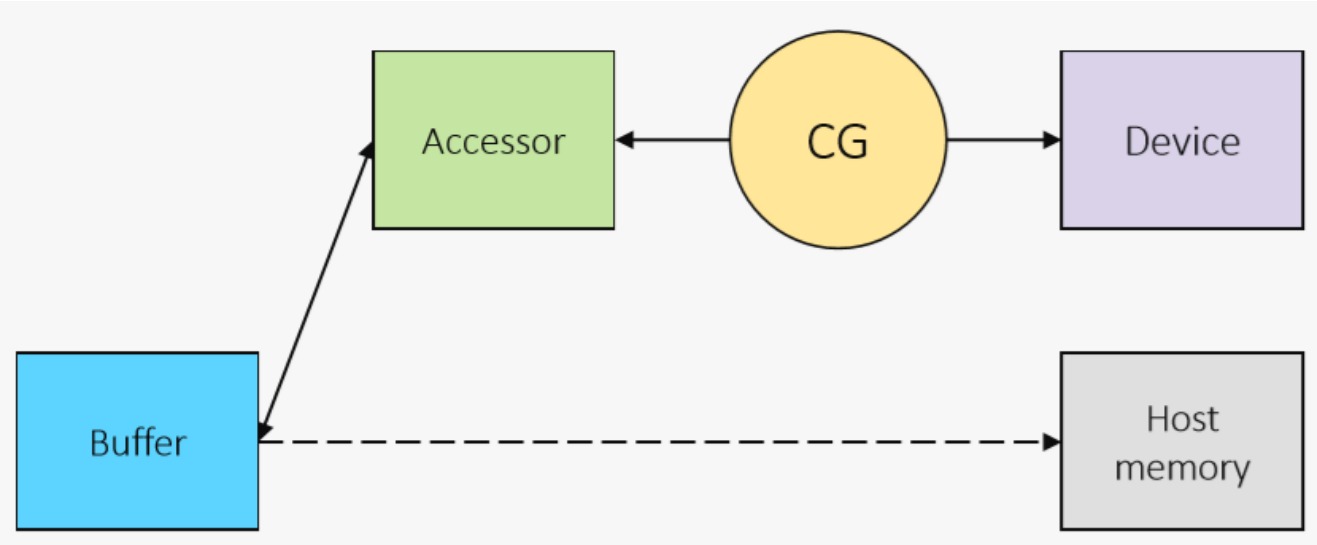
1. Buffer & Accessors
2. USM

Data Handling (Buffer & Accessors)

- The **buffer** is constructed as a pointer to host memory
  - The buffer's lifetime is determined by the SYCL runtime
- The buffer's information is transferred to the device when a kernel needs it

Data Handling (Buffer & Accessors)

- The **accessor** is the mechanism for accessing a *buffer*
- When an accessor is created, it is associated with a command group with a *handler*
  - Connects the accessible buffer and the device where the *command group* is launched



Data Handling (Buffer & Accessors)

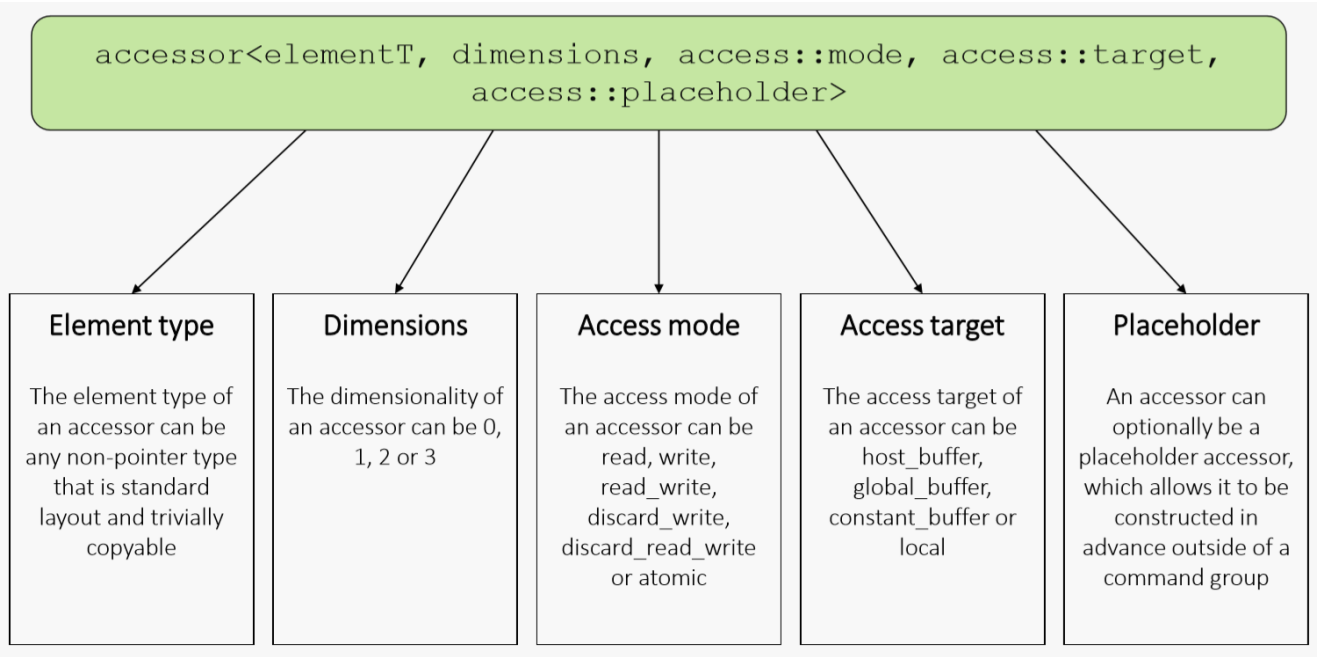
- Buffer construction:
  - As a pointer to the data
  - The `range` describes the number of elements

```
int var = 42;
auto buf = sycl::buffer{&var, sycl::range{1}};
```

Data Handling (Buffer & Accessors)

- Accesor
  - Usage: access on the device, access from the host, or reserve local memory (*shared* notation in CUDA)

```
auto readAcc = sycl::accessor{buf, cgh, sycl::read_only};
```



Data Handling (Buffer & Accessors)

- Let's see an example

buffer_example.cpp

```
{
   // Create a buffer of ints from an input iterator
   std::vector<int> myVec;
   buffer b1{myVec};                    //   Buffer for Vectors
   buffer b2{myVec.begin(), myVec.end()}; // Buffer for Vectors

   // Create a buffer of ints from std::array
   std::array<int,42> my_data;
   buffer b3{my_data};                  //    Buffer for std::array

   // Create a buffer of 4 doubles and initialize it from a host pointer
   double myDoubles[4] = {1.1, 2.2, 3.3, 4.4};
   buffer b4{myDoubles, range{4}};  //     Buffer from a host pointer
}
```

Data Handling (Buffer & Accessors)

buffer_example.cpp

```
void main(){
    sycl::queue Q();

    std::vector<float> a(N);

    for(int i=0; i<N; i++) a[i] = i; // Init a

    //Create a submit a kernel
    {
        buffer buffer_a{a}; //Create a buffer with values of array a

        // Create a command_group to issue command to the group
        Q.submit([&](handler &h) {
            accessor acc_a{buffer_a, h, read_write}; // Accessor to buffer_a

            // Submit the kernel
            h.parallel_for(N, [=](id<1> i) {
                acc_a[i]*=2.0f;
            }); // End of the kernel function
        }).wait();     // End of the queue commands we want on the event reported.
    };

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

USM

- It is a mechanism for managing memory managed by pointers like C/C++
- No accessors are required for memory access
- Considering the virtual memory access mechanism: this means that each pointer returned in a USM alloc can be used from the *host* or on the *device*
  - USM allows describing three types of memory allocated in: *host*, *device*, or *shared*

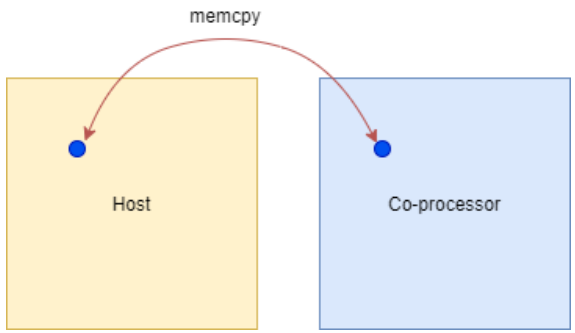
USM

- USM allows describing three types of memory allocated in: *host*, *device*, or *shared*

Allocation Type	Description	Accessible from the host?	Accessible from the device?	Location
device	Allocated in device mem	NO	YES	device
host	Allocated in host mem.	YES	YES	host
shared	Allocated between host and device mem.	YES	YES	can migrate

Explicit Handling

- Explicit Memory Handling
  - Data is transferred between host and device explicitly
  - SYCL runtime does not perform any dependency analysis, and the programmer must do it explicitly



Explicit Handling (USM)

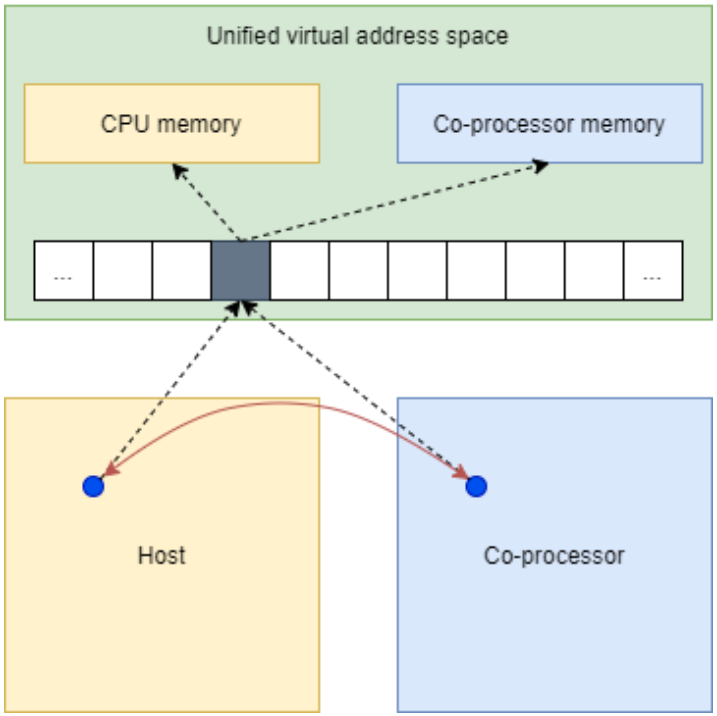
- Explicit Memory Handling with **memcpy**

explicit_memory.cpp

```
int main() {
  queue q;
  int *data = static_cast<int *>(malloc(N * sizeof(int)));
  for (int i = 0; i < N; i++) data[i] = i;

  // Explicit USM allocation using malloc_device
  int *data_device = malloc_device<int>(N, q));

  // copy mem from host to device
  auto e1 = q.memcpy(data_device, data, sizeof(int) * N);

  // update device memory
  auto e2 = q.submit([&](handler &h) {
    h.depends_on(e1);
    h.parallel_for(range<1>(N), [=](id<1> i) { data_device[i] *= 2; });
  });

  // copy mem from device to host
  q.submit([&](handler &h) {
    h.depends_on(e2);
    h.memcpy(data, data_device, sizeof(int) * N);
  }).wait();

  //print output
  for (int i = 0; i < N; i++) std::cout << data[i] << std::endl;
  free(data_device, q);
  free(data);
  return 0;
}
```

USM

- USM allows implicit data movement between the *host* and *device* with **shared** allocation
  - **Simplifies portability** in coding for accelerators
- Provides the programmer with the desired level of control
- Complementary to the **buffers** model



USM

usm_memory.cpp

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

static const int N = 16;

int main() {
  queue q;

  // USM allocation using malloc_shared
  int *data = static_cast<int *>(malloc_shared(N * sizeof(int), q));

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

  // Modify data array on device
  q.parallel_for({N}, [=](id<1> i) { data[i] *= 2; }).wait();

  // print output
  for (int i = 0; i < N; i++) std::cout << data[i] << std::endl;
  free(data, q);
  return 0;
}
```

Buffer/accessor vs USM

- The buffer/accessor provides guaranteed coherence and automatically manages dependencies
  - Recommended when iteration is needed
  - Recommended for maximum performance portability
  - Provides a lot of information to the SYCL runtime, which can perform many optimizations automatically.
- The USM model provides a pointer-based solution
  - Recommended when migrating existing C++ applications or using pointer-based structures
  - The user is responsible for certain performance-relevant aspects

Hands-on

1. Connect to the Intel Developer Cloud
2. In the **Training and Workshops** section
3. ... but we will work on **Essentials of SYCL**
   - https://console.cloud.intel.com/training/detail/9b8933a6-b466-4c69-8217-9a09d084a55a
4. Inside the IDC, open the notebook in the path **Training/HPC/oneapi-essentials-training/03_SYCL_Unified_Shared_Memory/Unified_Shared_Memory.ipynb**
5. Inside the IDC, open the notebook in the path **Training/HPC/oneapi-essentials-training/09_SYCL_Buffers_And_Accessors_In-depth/SYCL_Buffers_accessors.ipynb**

Parallelism levels

Kernels

- There are three types of kernels in SYCL:
  - **single_task**: Executes a single instance of the kernel
  - **parallel_for**: Executes as many instances as work-items (grouped in work-groups)
  - **parallel_for_work_group**: Similar to the NDRange of OpenCL (or grid in CUDA)
    - Code of a work-group can be executed, distributed among work-items
    - Use of local memory is allowed (equivalent to CUDA's *shared* or *private* memories)

Parallel Kernels

- Expressing parallelism through *kernels* allows multiple instances of an operation to run in parallel
- Useful for offloading parallel execution of an independent **for-loop**
- Parallel *kernels* are expressed using the **parallel_for** function

`for-loop in CPU application` ⟶ `Offload to Accelerator using parallel_for`

```
for (int i=0; i<1024; i++) {
   a[i] = b[i] + c[i];
}
```

```
h.parallel_for(range(1024), [=] (id<1> i){
   a[i] = b[i] + c[i];
});
```

Basic Parallel Kernels

- The functionality of kernels is expressed through classes of *range*, *id*, and *item*
  - *range* is used to describe the iteration space of parallel execution
  - *id* is used to index an individual instance of a kernel in a parallel execution
  - *item* represents an individual instance of the kernel function

matrix_mult.cpp

```
h.parallel_for(range<2>(N, N), [=](id<2> item) {
    auto i = item[0];
    auto j = item[1];

    // CODE THAT RUNS ON DEVICE
    c[i*N+j] = 0.0f;
    for(int k=0; k<N; k++)
        c[i*N+j] += a[i*N+k]*b[k*N+j];
}); // End of the kernel function
```

Parallel Kernels

- `parallel_for_work_group` and `parallel_for_work_item`: hierarchical parallelism can be expressed

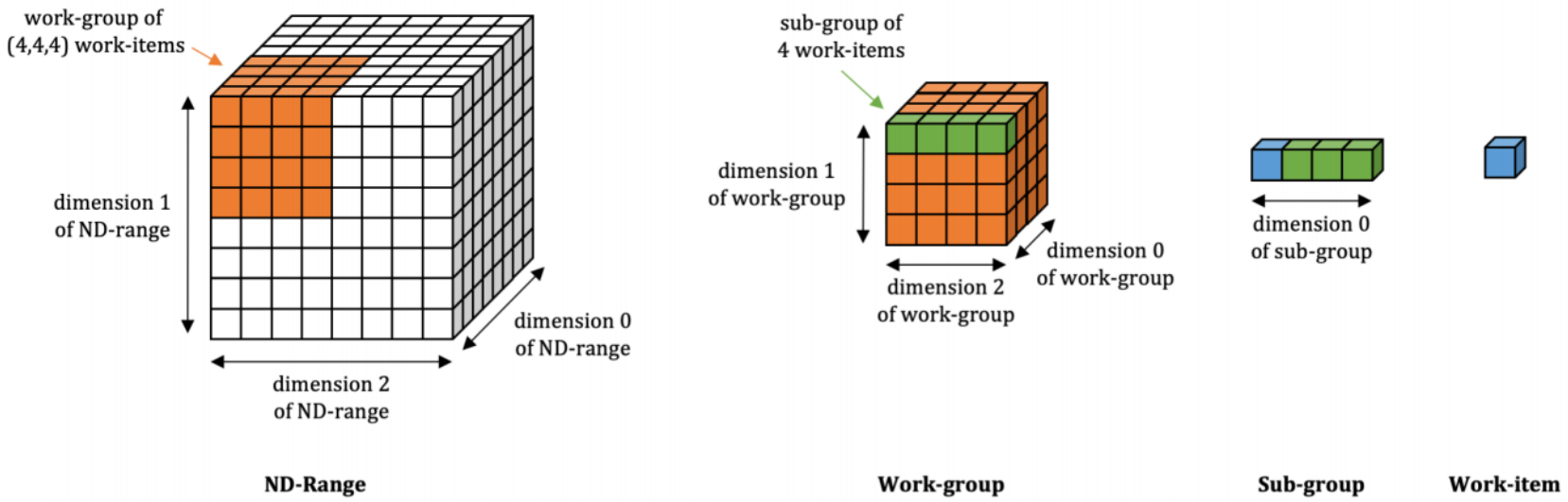
matrix_mult.cpp

```
const int B=16;

// Create a command_group to issue command to the group
Q.submit([&](handler &h) {

    range num_groups = range<2>(N/B, N/B); // N is a multiple of B
    range group_size = range<2>(B, B);

    h.parallel_for_work_group(num_groups, group_size, [=](group<2> grp) {
        // kernel function is executed once per work-group
        int ib = grp.get_group_id(0);
        int jb = grp.get_group_id(1);
        grp.parallel_for_work_item([&](h_item<2> it) {
            // kernel function is executed once per work-item
            int i = ib * B + it.get_local_id(0);
            int j = jb * B + it.get_local_id(1);

            // CODE THAT RUNS ON DEVICE
            c[i*N+j] = 0.0f;
            for(int k=0; k<N; k++)
                c[i*N+j] += a[i*N+k]*b[k*N+j];
        });
    });     // End of the kernel function
}).wait(); // End of the queue commands we want on the event reported.
```

ND-Range Kernels

- Parallel **kernels** are an easy way to parallelize a *for-loop*, but do not allow performance **optimization at a low level** (hardware exploitation)
- ND-Range is another way to express parallelism that allows **low-level performance tuning** by providing access to local memory and mapping of hardware processing units
  - The entire iteration space is divided into smaller groups called **work-groups**, where **work-items** are mapped to a single processing unit in hardware
  - Grouping work-items into work-groups allows controlling **resource usage** and **balancing the load of work** distributions

ND-Range Kernels



- Notation similar to CUDA or OpenCL

ND-Range Kernels

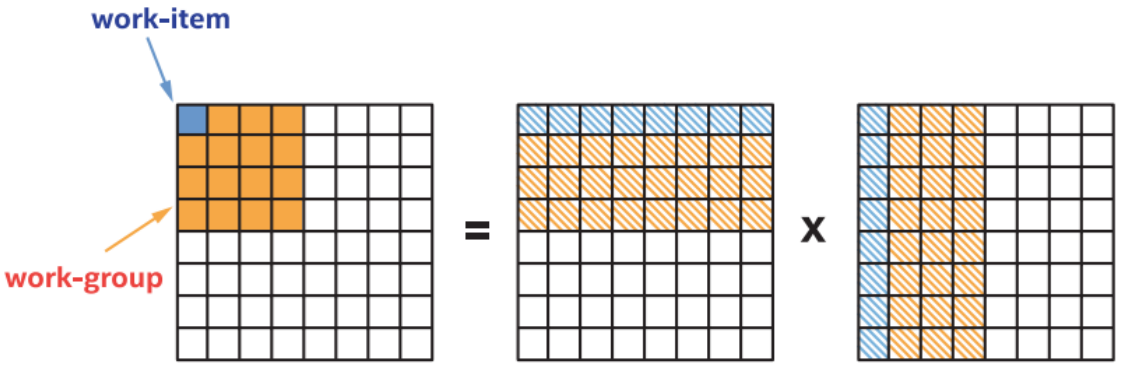
- The functionality of the kernel's **nd_range** is expressed with the **nd_range** and **nd_item** classes
  - **nd_range** represents grouped execution using the global execution range and the local execution range of each work-group
  - **nd_item** represents the execution of an individual instance of the kernel and allows querying the work-group range and its index

matrix_mult.cpp

```
// Create a command_group to issue command to the group
Q.submit([&](handler &h) {
    const int B = 16;
    range global = range<2>(N, N);
    range local = range<2>(B, B);

    h.parallel_for(nd_range{global, local}, [=](nd_item<2> item){
        auto i = item.get_global_id()[0];
        auto j = item.get_global_id()[1];

        // CODE THAT RUNS ON DEVICE
        c[i*N+j] = 0.0f;
        for(int k=0; k<N; k++)
            c[i*N+j] += a[i*N+k]*b[k*N+j];
    }); // End of the kernel function
}).wait(); // End of the queue commands we want on the event reported.
```

ND-Range Kernels

- The functionality of the kernel's **nd_range** is expressed with the **nd_range** and **nd_item** classes
  - **nd_range** represents grouped execution using the global execution range and the local execution range of each work-group
  - **nd_item** represents the execution of an individual instance of the kernel and allows querying the work-group range and its index



Hands-on

1. Connect to the Intel Developer Cloud
2. In the **Training and Workshops** section
3. ... but we will work on **Essentials of SYCL**
   - https://console.cloud.intel.com/training/detail/9b8933a6-b466-4c69-8217-9a09d084a55a
4. Inside the IDC, open the notebook in the path **Training/HPC/oneapi-essentials-training/02_SYCL_Program_Structure/SYCL_Program_Structure.ipynb**
5. Inside the IDC, open the notebook in the path **Training/HPC/oneapi-essentials-training/04_SYCL_Sub_Groups/Sub_Groups.ipynb**

Coding example

Code anatomy

- SYCL-based programs require the inclusion of the header `CL/sycl.hpp`
- In the case of using FPGAs, include the following:

```
#include <CL/sycl/intel/fpga_extensions.hpp>
```

- It is useful to add the namespace declaration to enable references to `sycl`

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

Code Anatomy

- Each execution on the device is associated with a queue
- A queue is connected to a single device (e.g., GPU, FPGA, CPU, host)

code.cpp

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

int main(int argc, char *argv[]) {
...
  queue myQueue{...};
...
```

Code Anatomy

- Devices types associated to a queue

type	Device
default_selector_v	Selects any device or host device if no device can be found.
gpu_selector_v	Select a GPU
accelerator_selector_v	Select an accelerator
cpu_selector_v	Select a CPU device
my_device_selector	*Custom selector*

custom_device.cpp

```
class my_device_selector: public cl::sycl::device_selector {
public:
   int operator()(const cl::sycl::device &Device) const override {
      int rating;
      if (dev.is_gpu()&(dev.get_info<info::device::name>().find("Intel") != std:
           string::npos))
        rating=3;
      else if (dev.is_gpu()) rating=2;
      else if (dev.is_cpu()) rating=1;
      return rating;
    }
};

int main() {
   my_device_selector selector;
   queue q(selector);
}
```

Code Anatomy

- The queue accepts work requests, and submissions finish asynchronously
- Only one kernel runs simultaneously on each queue

code.cpp

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

int main(int argc, char *argv[]) {
  ...
  queue q{...};
  ...
  q.submit([&](handler& h) {
    // accessors (for connecting to memory via buffers)
    // kernel defined here (with lambda -
    // by value captures only)
  });
}
```

Code Anatomy

- Host and device code in the same source file
- Functionality: *queue* (target code), *buffer* (data handling) y *parallel_for* (parallelism)

vector_add.cpp

```
using namespace cl::sycl;

int main() {
    float A[1024], B[1024], C[1024];

    for(int i=0; i<1024; i++) {A[i] = i; B[i] = 1;}

    {
        auto bufA = buffer<float>(A, 1024);
        auto bufB = buffer<float>(B, 1024);
        auto bufC = buffer<float>(C, 1024);

        queue q;
        q.submit([&](handler& h) {
            accessor A(bufA, h, read_only);
            accessor B(bufB, h, read_only);
            accessor C(bufC, h, write_only, no_init);


            h.parallel_for(range<1> {1024}, [=](auto i) {
                C[i] = A[i] + B[i];
            });
        });
    }
    for (int i = 0; i < 1024; i++)
        std::cout << "C[" << i << "] = " << C[i] << std::endl;
}
```

Matrix Multiplication

- Basic matrix multiplication $C_{N\times M} = A_{N\times K} * B_{K\times M}$

matrix_mult.cpp

```
h.parallel_for(range<2>(N, N), [=](id<2> item) {
    auto i = item[0];
    auto j = item[1];

    // CODE THAT RUNS ON DEVICE
    c[i*N+j] = 0.0f;
    for(int k=0; k<N; k++)
        c[i*N+j] += a[i*N+k]*b[k*N+j];
}); // End of the kernel function
```

Matrix Multiplication

- Matrix multiplication with local memory: $C_{N\times M} = A_{N\times K} * B_{K\times M}$
  - Local memory is expressed in the kernel invocation by indicating *tile_size*

matrix_mult.cpp

```
// Local accessor, for one matrix tile:
constexpr int tile_size = 16;

// Create a command_group to issue command to the group
Q.submit([&](handler &h) {
   const int B = 16;
   range global = range<2>(N, N);
   range local = range<2>(B, B);

   local_accessor<float, 2> tileA(range<2>(tile_size,tile_size), h);
   local_accessor<float, 2> tileB(range<2>(tile_size,tile_size), h);

   h.parallel_for(nd_range{global, local}, [=](nd_item<2> item){
      auto n = item.get_global_id()[0];
      auto m = item.get_global_id()[1];

      // Index in the local index space:
      auto j = item.get_local_id()[0];
      auto i = item.get_local_id()[1];
      float sum = 0;
      for (int kk = 0; kk < N; kk += tile_size) {
         // Load the matrix A, matrix B and synchronize
         // to ensure all work-items have a consistent view
         // of the matrix tile in local memory.
         tileA[i][j] = a[m*N+(j+kk)];
         tileB[i][j] = b[(i+kk)*N+n];
         item.barrier();
         // Perform computation using the local memory tile
         for (int k = 0; k < tile_size; k++)
            sum += tileA[i][k] * tileB[k][j];
         // After computation, synchronize again, to ensure all
         // reads from the local memory tile are complete.
         item.barrier();
      }
      // Write the final result to global memory.
      c[m*N+n] = sum;
   });     // End of the kernel function
}).wait(); // End of the queue commands we want on the event reported.
```

Execution Model

Execution Model

- Source code in a single file

Host Code

- Executed on the CPU(s)

Execution Model

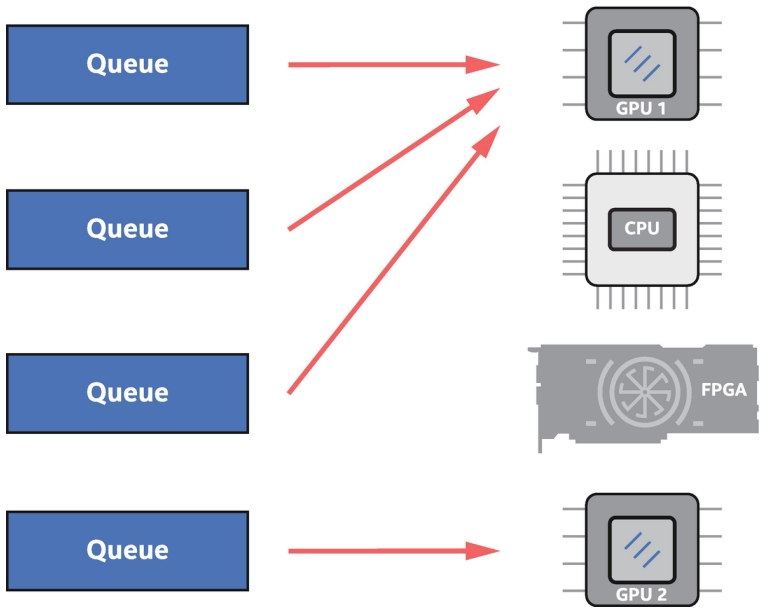
- Source code in a single file

Device Code

- Kernel code defines the work to be offloaded
- Mechanism to dispatch to a device: **queues**
- Asynchronous from the host's point of view
- Choose devices:
  - Run the code "somewhere": device doesn't matter
  - Explicitly (host, GPU, FPGA...)

Execution Model

- Where the kernels is executed?
  - A source code can create many **queues**
  - A **queue can not** be linked with more than a device
  - Several queues can be joined to a the same device



In-Order Queues

- The simplest option for task ordering is to emit them in order (in-order queue).
  - Advantage: Simplicity
  - Disadvantage: Task execution is sequential (even if there are no dependencies between tasks)

in-order_queue.cpp

```
#include <CL/sycl.hpp>
using namespace sycl;
constexpr int N = 4;
int main() {
  queue Q{property::queue::in_order()};

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

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

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

  return 0;
}
```

Out-of-Order Queues

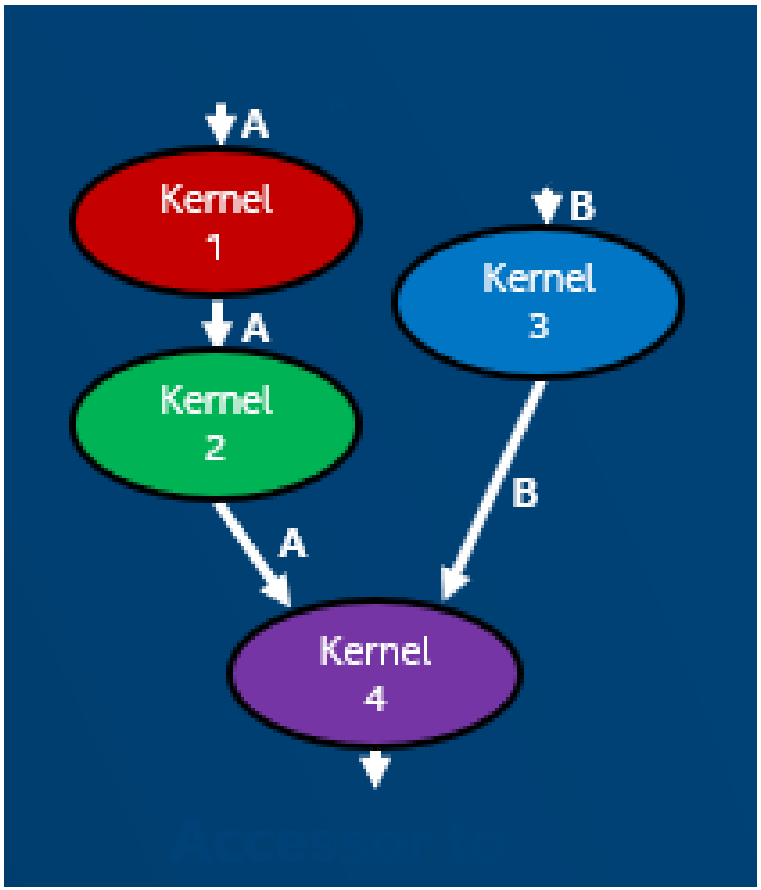
- SYCL runtime generates a dependency graph.
  - Synchronization between kernels according to **accessors**
  - Kernel 4 depends on K3 and K2; Kernel 2 depends on K1

dependencies.cpp

```
int main() {
  auto R = range<1>{ num };
  buffer<int> A{ R }, B{ R }; //Create Buffers A and B
  queue Q; //Create a device queue

  Q.submit([&](handler& h) {//Submit Kernel 1
    auto out = A.get_access<access::mode::write>(h);//Accessor buffer A
    h.parallel_for(R, [=](id<1> idx) {
      out[idx] = idx[0]; }); });

  Q.submit([&](handler& h) { //Wait till the first queue is complete
    auto out = A.get_access<access::mode::write>(h);//Accessor buffer A
    h.parallel_for(R, [=](id<1> idx) {
      out[idx] = idx[0]; }); });

  Q.submit([&](handler& h)
    auto out = B.get_access<access::mode::write>(h);//Accessor buffer B
    h.parallel_for(R, [=](id<1> idx) {
      out[idx] = idx[0]; }); });

  Q.submit([&](handler& h) { //Wait till kernel 2 and 3 are complete
    auto in = A.get_access<access::mode::read>(h);
    auto inout = B.get_access<access::mode::read_write>(h);
    h.parallel_for(R, [=](id<1> idx) {
      inout[idx] *= in[idx]; }); });

  // And the following is back to device code
  auto result = B.get_access<access::mode::read>();
  for (int i=0; i<num; ++i) std::cout << result[i] << "\n";

  return 0;
}
```

Out-of-Order Queues

- Kernels sycro: DaG
  - Kernel 4: depends to K3 and K2; Kernel 2 depends of K1

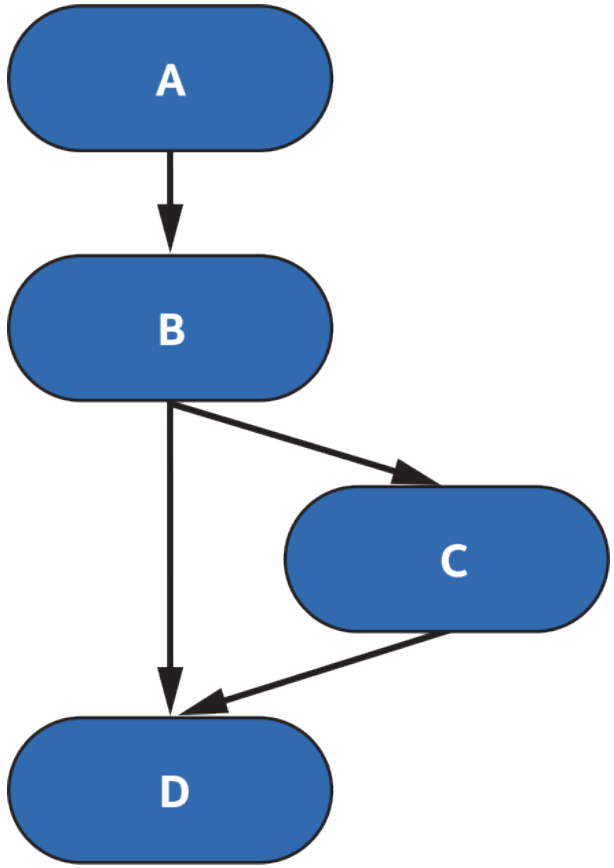


Colas en fuera de orden

- Sincronización de kernels con eventos (*depends_on*)

queue_events.cpp

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



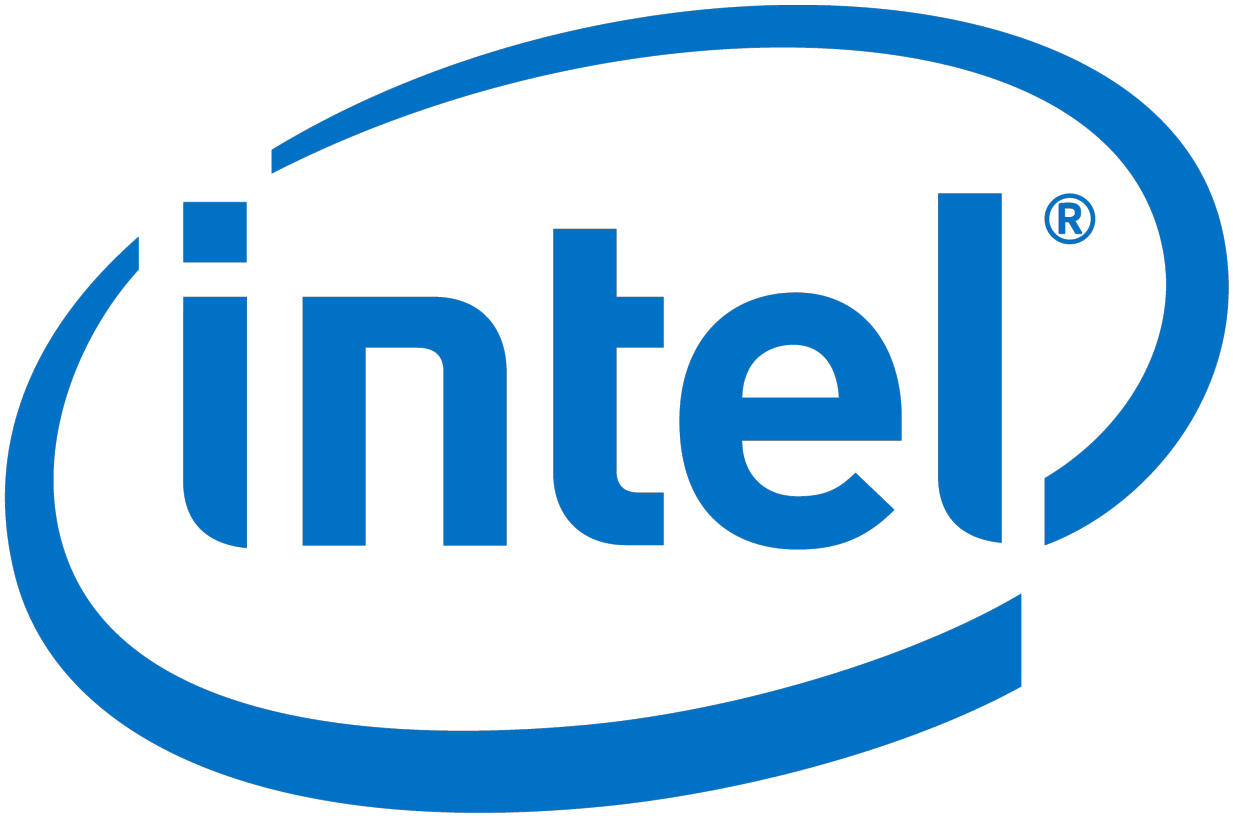
Hands-on

1. Connect to the Intel Developer Cloud
2. In the **Training and Workshops** section
3. ... but we will work on **Essentials of SYCL**
   - https://console.cloud.intel.com/training/detail/9b8933a6-b466-4c69-8217-9a09d084a55a
4. Inside the IDC, open the notebook in the path **Training/HPC/oneapi-essentials-training/10_SYCL_Graphs_Scheduling_Data_management/SYCL_Task_Scheduling_Data_dependency.ipynb**

Extra

Additional Resources

- oneAPI Initiative https://www.oneapi.io/
- Intel oneAPI Base & HPC Toolkit https://www.intel.com/content/www/us/en/developer/tools/oneapi/commercial-base-hpc.html
- **New** Book 2023 *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL* available at the link



Software

Contacto

Thanks for your attention!!!

Danysoft
Haciendo visible lo invisible

intel
software

oneAPI

Address
Avda. de la industria 4, edif. 1
28108 Alcobendas | Madrid | España

Phone
[+34] 91 663 8683

Email
info@danysoft.com

Website
www.danysoft.com/intel

intel
software

Danysoft
Haciendo visible lo invisible

intel
software elite reseller