CINECA

Webinar: Heterogeneous computing with SYCL & oneAPI on CINECA's Leonardo

Webinar • October 10th, 2024

Biagio Cosenza
Department of Computer Science
University of Salerno, Italy

Khronos SYCL Working Group
Intel oneAPI Certified Instructor
www.cosenza.eu bcosenza@unisa.it

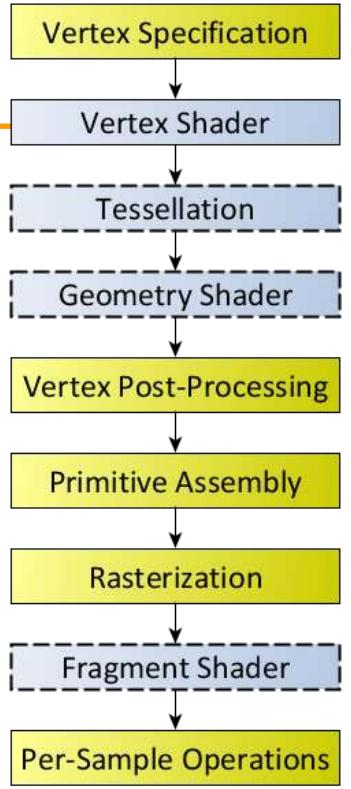
Teaching materials can be found at https://cosenza.eu/sycl/cineca24.htm

Outline

- Introduction to heterogenous computing
- Introduction to SYCL
  - oneAPI setup at CINECA
  - basics: queue and parallelism, device, range, task scheduling, memory access modes
  - advanced: kernel reduction, subgroups and group algorithms, atomics
  - SYCL & the oneAPI ecosystem
- CUDA to SYCL
  - migration tool
  - examples of successful migration
- Next steps:
  - online learning on the CodeReckons platform
  - SYCL oneAPI Hackaton at CINECA

GPU Architecture: an Historical Perspective

- GPU: Graphics Processing Unit
  - accelerates graphics
  - graphics processing as pipeline: rasterization pipeline
  - programmer can program specific stage of the pipeline



- Example: fragment shader in the OpenGL Shading Language

```
uniform sampler2D texture1;
uniform sampler2D texture2;

void main()
{
 FragColor = mix(texture(texture1, TexCoord), texture(texture2, TexCoord), 0.2);
}
```

GPU Architecture: an Historical Perspective

- GPGPU (General Purpose GPU) programming
  - early approach: using GPUs for general purpose computing other than graphics
- Map the problem into the rasterization pipeline
  - arrays as textures
  - kernels as shaders
  - computing as drawing (rendering pass)

A saxpy kernel in OpenGL SL

```
uniform sampler2D textureY;
uniform sampler2D textureX;
uniform float alpha;

void main(void) {
  vec4 y = texture2D(textureY, gl_TexCoord[0].st);
  vec4 x = texture2D(texture, gl_TexCoord[0].st);
  gl_FragColor = y + alpha*x;
}
```

- Modern GPGPU programming
  - Open: by Khronos (OpenCL, SYCL)
  - Proprietary: by NVIDIA (CUDA), AMD (ROCm/HIP), Intel (Level Zero)

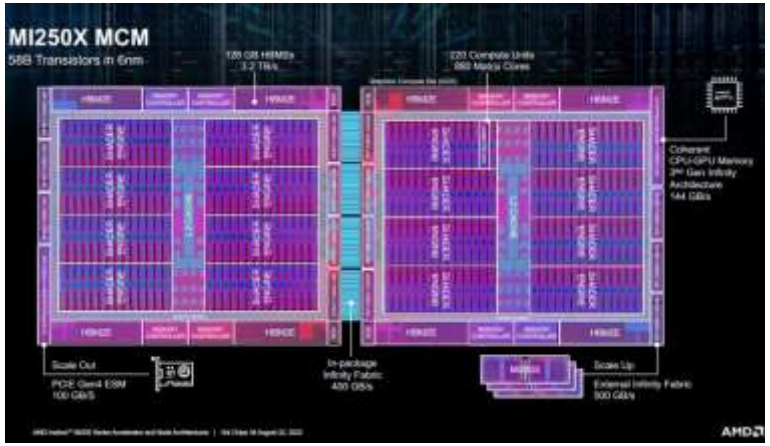
Leonardo at CINECA

Leonardo
BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband,
EVIDEN
EuroHPC/CINECA
Italy

Towards Modern GPU Architectures

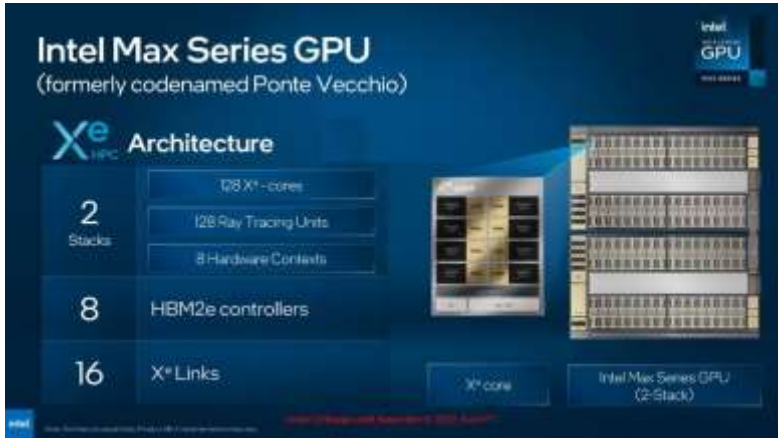
AMD Instinct MI250X

- #1 Frontier
- #5 LUMI



Intel Max 1100

- #2 Aurora

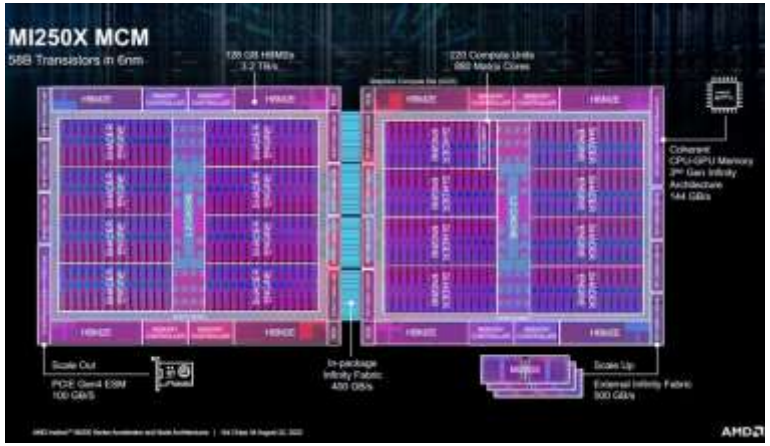


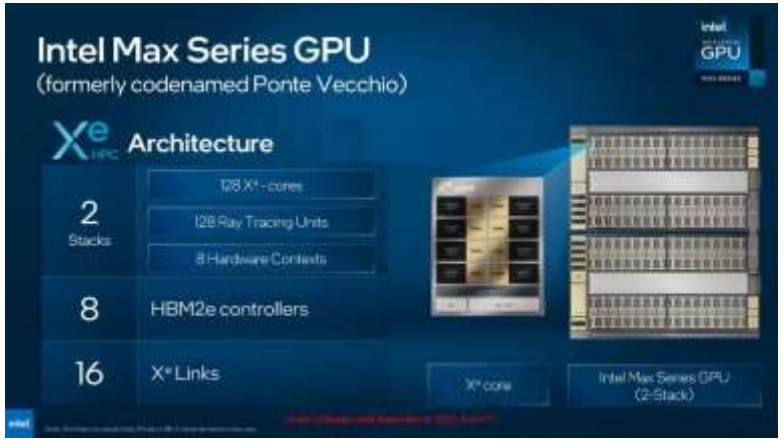
NVIDIA A100

- #7 Leonardo

Features of modern GPUs – A quick overview

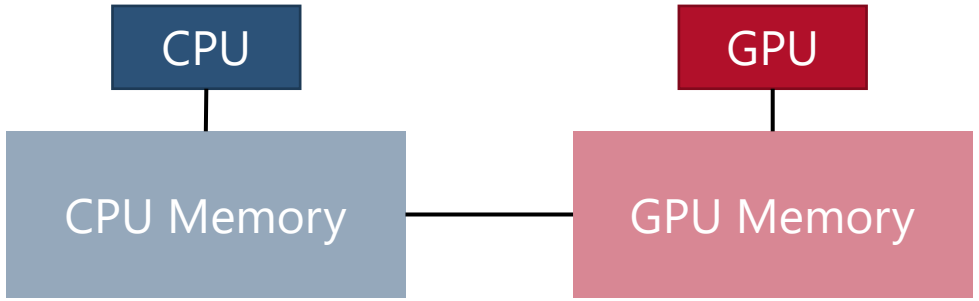
1. Platform model
2. Memory model
3. Execution model

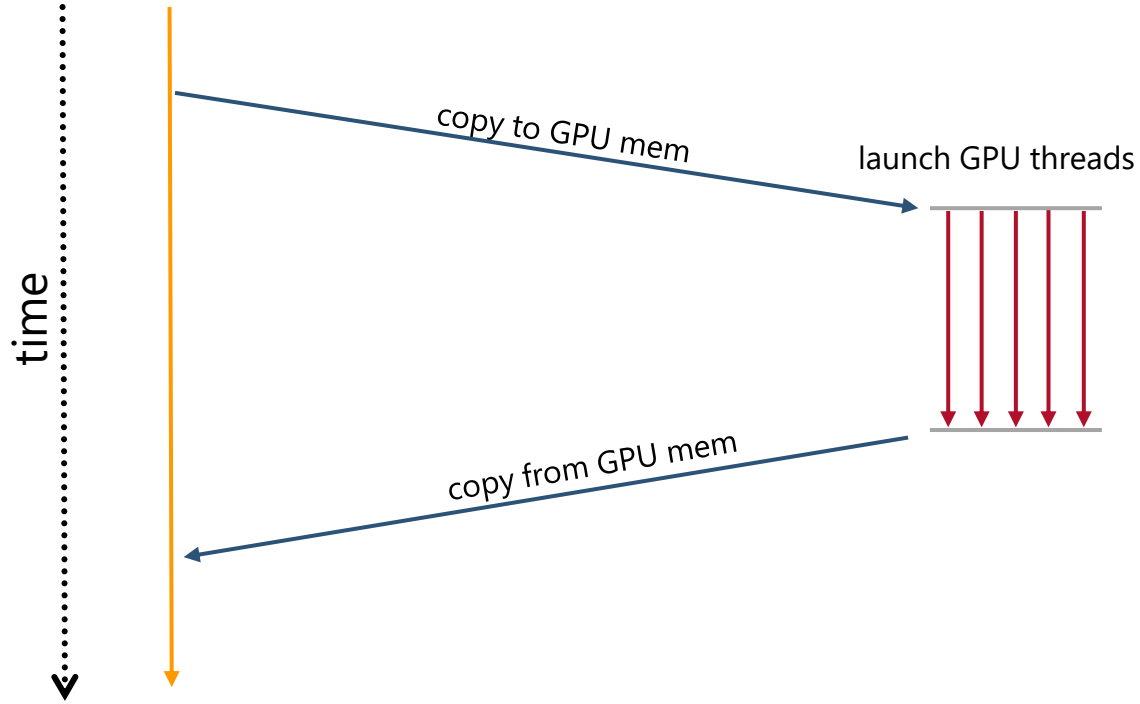




GPU Platform Model (1)

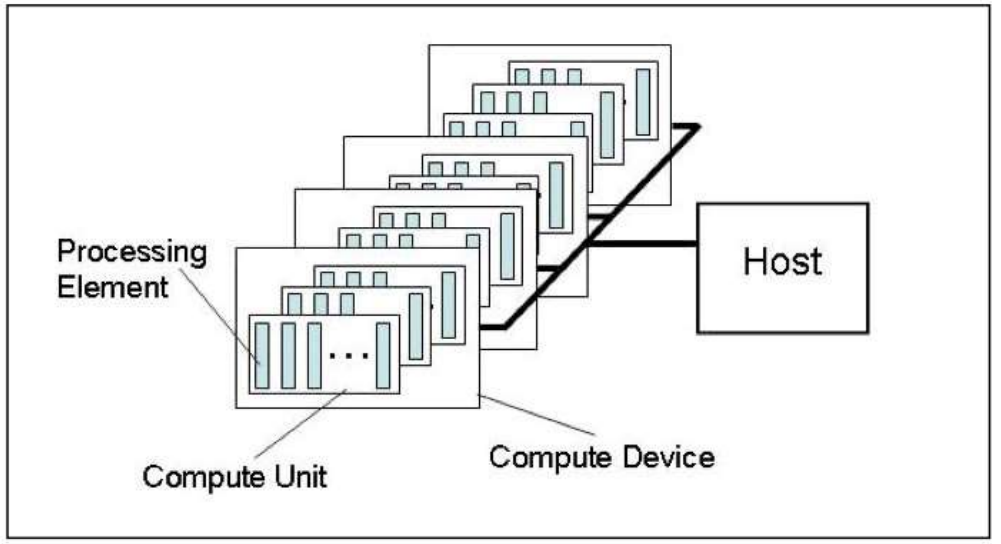
- GPU cards connected to the CPU via bus (e.g., PCI bus, NVlink)
  - accelerator
- Typical steps
  1. data copied to the GPU memory
  2. computation on the GPU
  3. copy data back from GPU to CPU

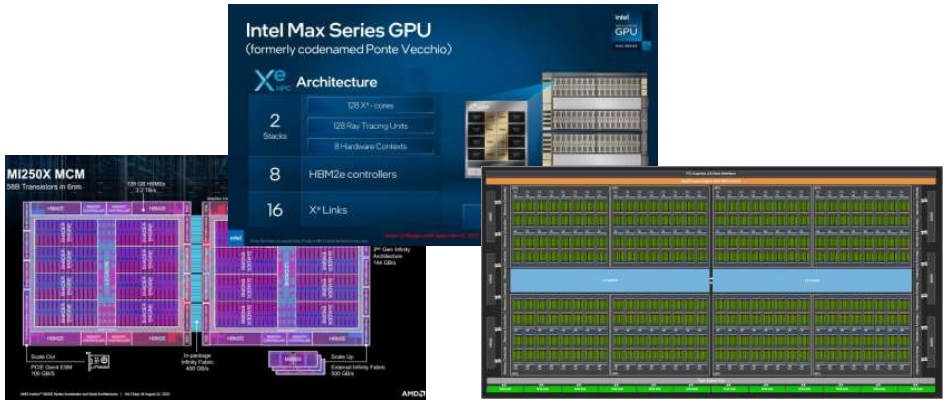




GPU Platform Model (2)

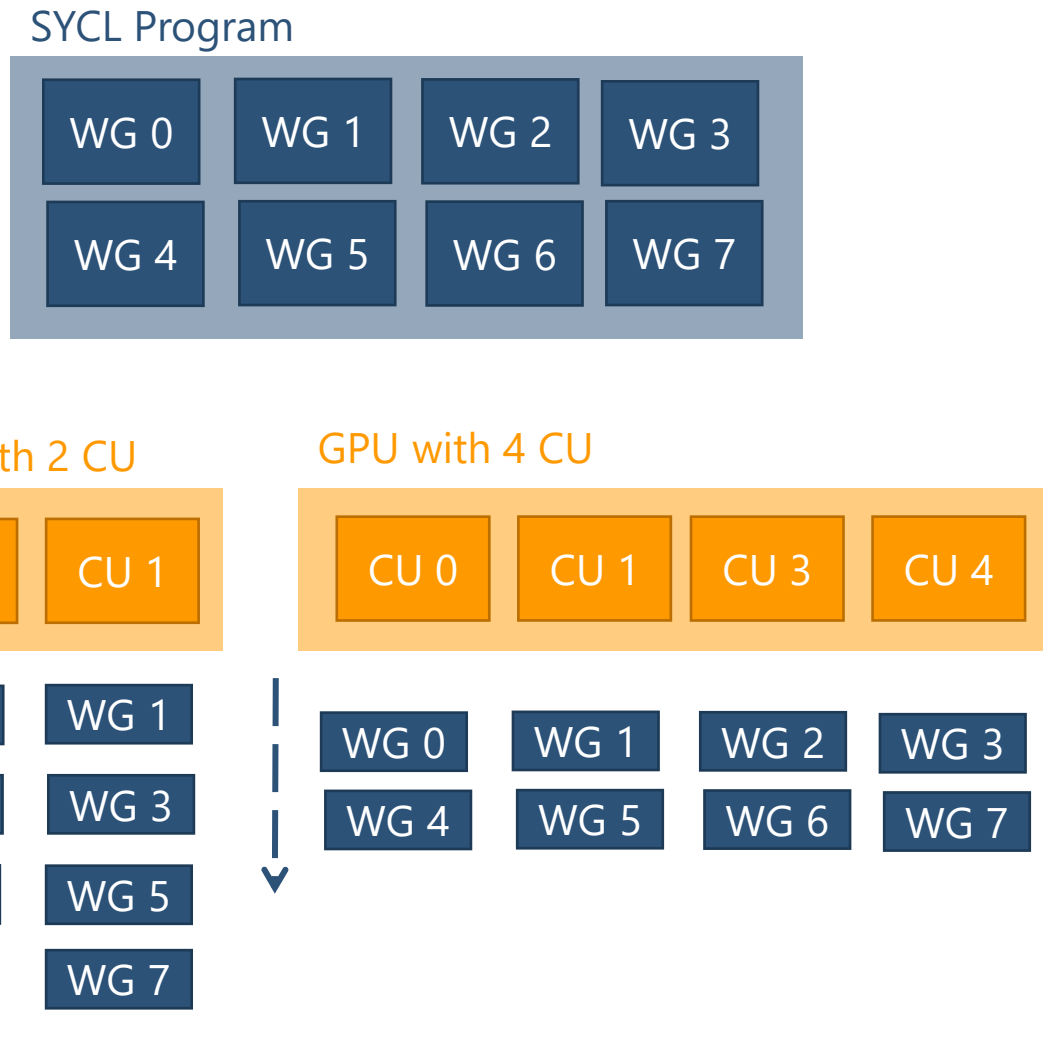
- Platform model
  - one host and one or more OpenCL devices
  - a device is divided into one or more compute units
  - compute units (CU) are divided into one or more processing elements
  - each compute unit is divided into one or more processing elements (PE)
- Memory divided into host memory and device memory





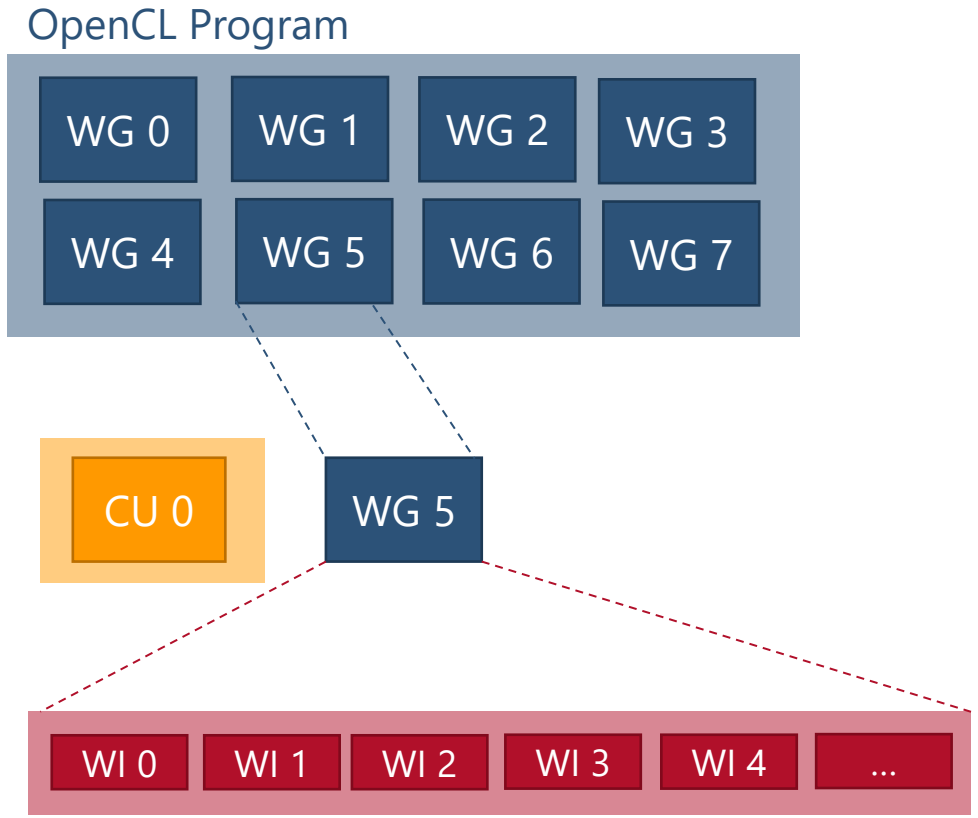
GPU Execution Model (1)

- Work-group: a collection of work-items that execute on a single compute unit
  - CUDA block
- The work-items in the same group
  - execute the same kernel instance
  - share local memory and work-group functions
- We can specify the number of work-items in a work-group
  - this is called the local (work-group) size
  - alternatively, the run-time can choose the work-group size for you (may be not optimal)



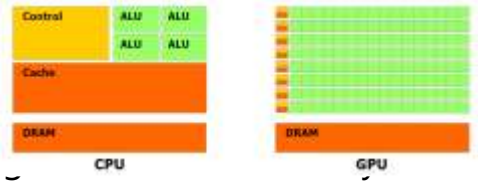
GPU Execution Model (2)

- Work-item: a collection of parallel executions of a kernel invoked on a device by a command
  - CUDA CUDA thread
- A work-item
  - is executed by one or more processing elements as part of a work-group executing on a compute unit
  - is distinguished from other work-items by its global ID or the combination of its work-group ID and its local ID within a work-group

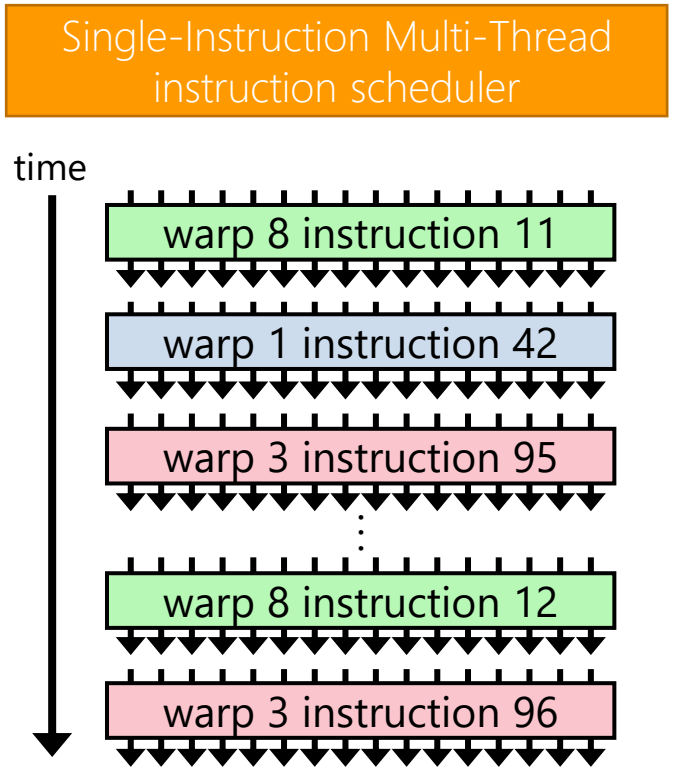


GPU Execution Model (3)

- SIMT: Single-Instruction Multi-Thread
  - executes one instruction across many independent threads
  - NVIDIA Warp: 32 parallel threads execute a SIMT instruction
  - AMD Wavefront: 64 parallel threads execute a SIMT instruction
  - Intel Xe Vector Engine (XVE): variable size
  - SYCL: subgroup
  - SIMT provides easy single-thread scalar programmin
  - hardware implements zero-overhead thread scheduling

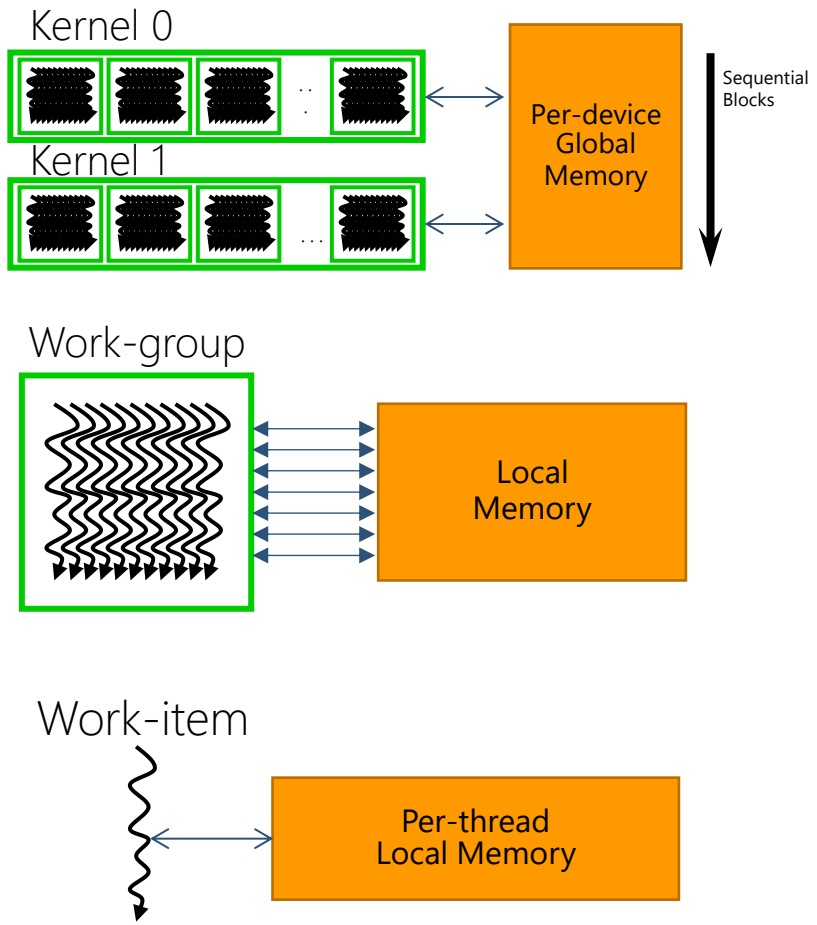


- SIMT threads can execute independently
  - SIMT warp/wavefront diverges and converges when threads branch independently
  - best efficiency and performance when threads of a warp execute together



GPU Memory Model

- Relaxed memory model
- Global memory
  - traditionally, the GPU frame buffer
  - can be accessed by any work-item
- Local memory
  - small memory close to the processor, low latency
  - allocated per work-group
  - NVIDIA shared memory
- Private memory
  - each thread has its own local memory
  - stacks, other private data



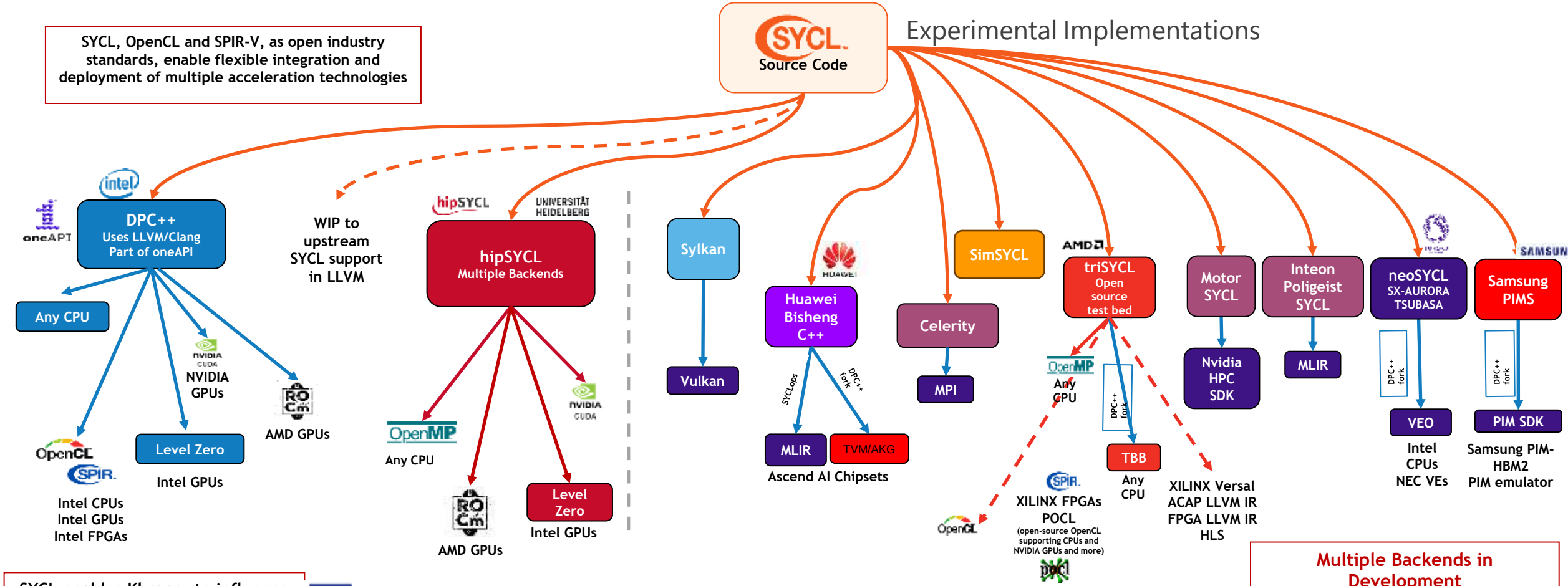
Introduction to SYCL

- SYCL is a single source, high-level, standard C++ programming model, that can target a range of heterogeneous platforms
- Open standard by Khronos Group
  - SYCL and the SYCL logo are trademarks of the Khronos Group Inc.

- Enables programming for heterogenous hardware from different vendors
  - CPU, GPU, FPGA, accelerators
  - freedom from vendor lock-in
- Comparable performance to native CUDA
  - Migration tool: SYCLomatic

SYCL Implementations

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



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

Multiple Backends in Development

SYCL on even more low-level frameworks.
For more information: http://sycl.tech

Edited image, original image courtesy of Michael Wong, CodePlay Ltd

Setup: login to Leonardo @ CINECA

1. Setup a new certificate

```
$ eval $(ssh-agent)
Agent pid 10523
$ step ssh login 'bcosenza@unisa.it' --provisioner cineca-hpc
✓ Provisioner: cineca-hpc (OIDC) [client: step-ca]
Your default web browser has been opened to visit:
```


```
https://sso.hpc.cineca.it/realms/CINECA-HPC/protocol/openid-connect/...

✓ CA: https://sshproxy.hpc.cineca.it
✓ SSH Agent: yes
```

2. SSH login

```
$ ssh bcosenza@login.leonardo.cineca.it
Welcome to:      _                                          _
                | |                                        | |
                | |      ___   ___   _ __    __ _ _ __  __| | ___
                | |     / _ \/ _ \| '_ \  / _` | '__/ _` |/ _ \
                | |____|  __/ (_) | | | | | (_| | | | (_| | (_) |
                |______\___|\___/|_| |_|\__,_|_|  \__,_|\___/
```

Setup: oneAPI at CINECA

/leonardo/pub/userexternal/bcosenza/oneapi

https://github.com/unisa-hpc/hpc-course/tree/main/03_gpu

3. Install oneAPI
4. Install Codeplay NVIDIA plugin
5. Run setvars.sh script
6. Download the code examples

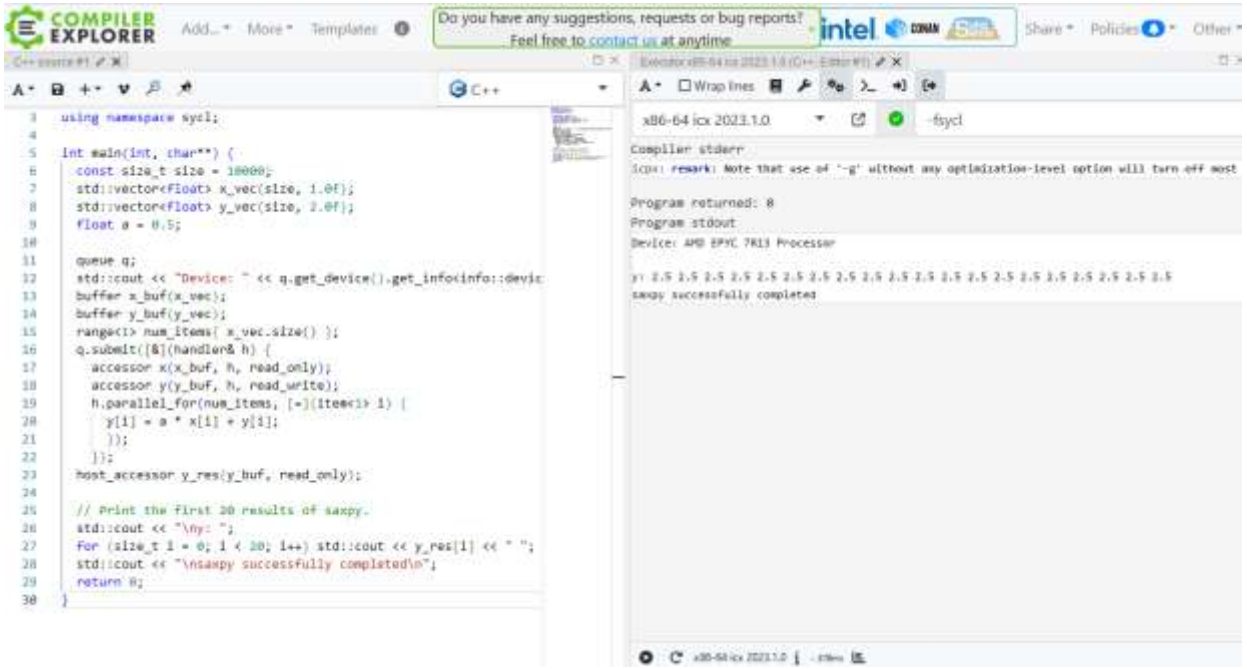
Recommended!

3. Use my setvar.sh script to configure a prebuild installation of oneAPI + NVIDIA plugin
4. Download the code examples

```
$ source /leonardo/pub/userexternal/bcosenza/oneapi/setvars.sh
:: initializing oneAPI environment ...
   -bash: BASH_VERSION = 4.4.20(1)-release
   args: Using "$@" for setvars.sh arguments:
:: advisor -- latest
…
:: vtune -- latest
:: oneAPI environment initialized ::
$ srun -n 1 -p boost_usr_prod --ntasks-per-node=1 --gres=gpu:4 --pty sycl-ls
…
[opencl:acc:0] Intel(R) FPGA Emulation Platform for OpenCL(TM), Intel(R) FPGA Emulation Device OpenCL 1.2 …
[opencl:cpu:1] Intel(R) OpenCL, Intel(R) Xeon(R) Platinum 8358 CPU @ 2.60GHz OpenCL 3.0 (Build 0) …
[ext_oneapi_cuda:gpu:0] NVIDIA CUDA BACKEND, NVIDIA A100-SXM-64GB 8.0 [CUDA 12.1]
[ext_oneapi_cuda:gpu:1] NVIDIA CUDA BACKEND, NVIDIA A100-SXM-64GB 8.0 [CUDA 12.1]
[ext_oneapi_cuda:gpu:2] NVIDIA CUDA BACKEND, NVIDIA A100-SXM-64GB 8.0 [CUDA 12.1]
[ext_oneapi_cuda:gpu:3] NVIDIA CUDA BACKEND, NVIDIA A100-SXM-64GB 8.0 [CUDA 12.1]
$ module load git; module load cuda/12.3; git clone https://github.com/unisa-hpc/hpc-course.git; cd hpc-course/03_gpu
```

Alternative setup: Compiler Explorer

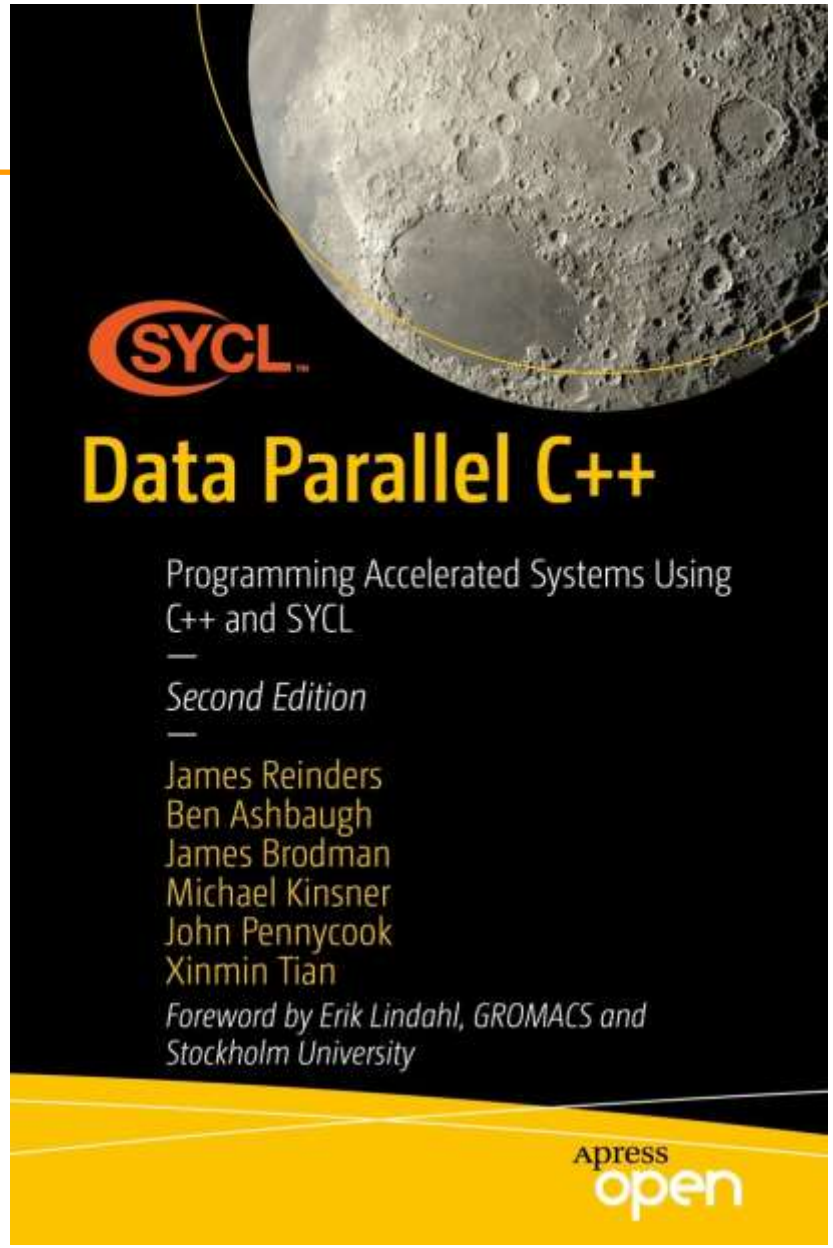
1. All exercise are available via browser on Compiler Explorer
   - Full list available here: http://cosenza.eu/sycl/cineca25.htm
   - Code are be executed on AWS instances (either Intel or AMD multicore)



Note: it will run on CPU instead of GPU, with smaller input size

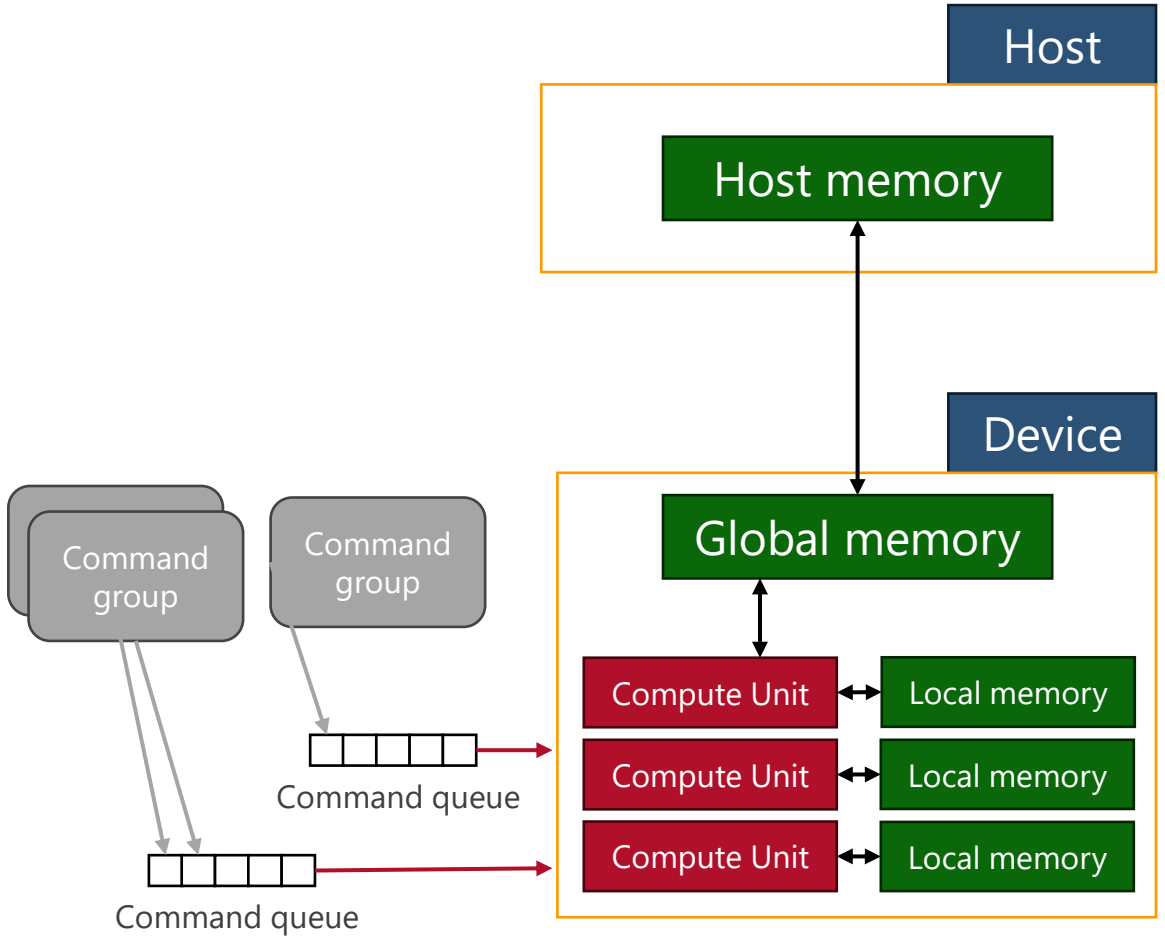
Recommended Book

- Data Parallel C++: Programming Accelerated Systems Using C++ and SYCL
- Authors: James Reinders, Ben Ashbaugh, James Brodman, Michael Kinsner, John Pennycook, Xinmin Tian
- Freely available here: https://link.springer.com/book/10.1007/978-1-4842-9691-2
- Updated 2nd edition!



SYCL Platform Model

- A host + one or more device
- A SYCL program is single source
  - both host and device code are in the same file
  - host part of the code is executed on the CPU
  - device part of the code is executed on a device, e.g., GPU or FPGA
    - submit command group to a queue
    - a queue execute commands on a device



First SYCL Program: SAXPY

- SAXPY
  - single-precision A times X plus Y
  - very simple C sequential implementation
- Apply SYCL data parallelism: parallel_for
  - replace loop with a function, called kernel, executing at each point in a problem domain

Traditional loops

```
void saxpy(int n, float a, float* x, float* y) {
  for (int i = 0; i < n; ++i)
    y[i] = a * x[i] + y[i];
}
```

SYCL parallel_for

```
h.parallel_for(num_items, [=](item<1> i) {
    y[i] = a * x[i] + y[i];
  });
```

SAXPY in SYCL (1)

1) Single source
   - host + device code
2) Queue
   - using a default device
   - can submit command groups
3) Buffers
   - encapsulate data across both devices and host
   - use accessor to access buffer data
4) Parallelism through ŕăśăĭĭêĭ ğộs
   - Index space: sắŋĝê ,
   - Kernel: C++ lambda function

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

int main(int, char**) {
  const int size = 10000;
  std::vector<float> x_vec(size, 1.0f);
  std::vector<float> y_vec(size, 2.0f);
  float a = 0.5;

  queue q;
  buffer x_buf(x_vec);
  buffer y_buf(y_vec);
  range<1> num_items{ x_vec.size() };
  q.submit([&](handler& h) {
    accessor x(x_buf, h, read_only);
    accessor y(y_buf, h, read_write);
    h.parallel_for(num_items, [=](item<1> i) {
      y[i] = a * x[i] + y[i];
    });
  });
  host_accessor y_res(y_buf, read_only);
  // … print results and returns
}
```

Compile with icpx and two flags `icpx -fsycl -fsycl-targets=nvptx64-nvidia-cuda saxpy.cpp`

Run with SLURM `srun -n 1 -p boost_usr_prod --ntasks-per-node=1 --gres=gpu:1 --pty ./a.out`

SAXPY in SYCL (2)

5) Accessor

- create data dependencies in the SYCL graph
- Can be:
  - `sêắđ ộŋľỳ`, `xsîʯê ộŋľỳ`, `sêắđ xsîʯê`

6) Host accessor

- Provides access to data in a buffer from host code that is outside of a command
- blocking call
- return after all enqueued SYCL kernels that modify the same buffer in any queue completes execution
- the data is available to the host via this host accessor

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

int main(int, char**) {
  const int size = 10000;
  std::vector<float> x_vec(size, 1.0f);
  std::vector<float> y_vec(size, 2.0f);
  float a = 0.5;

  queue q;
  buffer x_buf(x_vec);
  buffer y_buf(y_vec);
  range<1> num_items{ x_vec.size() };
  q.submit([&](handler& h) {
    accessor x(x_buf, h, read_only);
    accessor y(y_buf, h, read_write);
    h.parallel_for(num_items, [=](item<1> i) {
      y[i] = a * x[i] + y[i];
    });
  });
  host_accessor y_res(y_buf, read_only);
  // … print results and returns
}
```

Buffer and Accessors

- Next
  - How to select a device
    - heterogenous systems typically have many devices of different types
  - Queue synchronization
  - How to handle errors

SAXPY with device selector and exceptions (1)

1) Queue creation

- With device selector

```
{gpu|cpu|accelerator|default}_selector_v
```

- By directly passing the device

```
std::vector<device> dv =
  device::get_devices(sycl::info::device_type::gpu);
queue q(dv[0]); // pick the 1st GPU
```

- With device aspect selector

```
device d = device{aspect_selector(
  std::vector{aspect::fp16, aspect::gpu}, // allowed
  std::vector{sycl::aspect::custom}    // disallowed
) };
queue q(d)
```

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

int main(int, char**) {
  constexpr size_t size = 10000;
  // … initialization of x_vec, y_vec and a
  try {  // exception handling
    queue q(gpu_selector_v);
    {
      buffer x_buf(x_vec);
      buffer y_buf(y_vec);
      range<1> num_items{ x_vec.size() };
      q.submit([&](handler& h) {
        accessor x(x_buf, h, read_only);
        accessor y(y_buf, h, read_write);
        h.parallel_for(num_items, [=](item<1> i) {
          y[i] = a * x[i] + y[i];
        });
      });
    } // Queue syncronize at buffer destruction
   // … print results
  }
  catch (exception const& e) {
    std::cout << "An exception is caught for saxpy.\n";
  }
  return 0;
}
```

SAXPY with device selector and exceptions (2)

2) Buffer destruction

- by default, synchronize the data back to the host
  - unless `use_host_ptr`

3) Explicit queue synchronization

- blocking wait for the completion of all enqueued tasks in the queue
  - `wait()` or `wait_and_throw()`

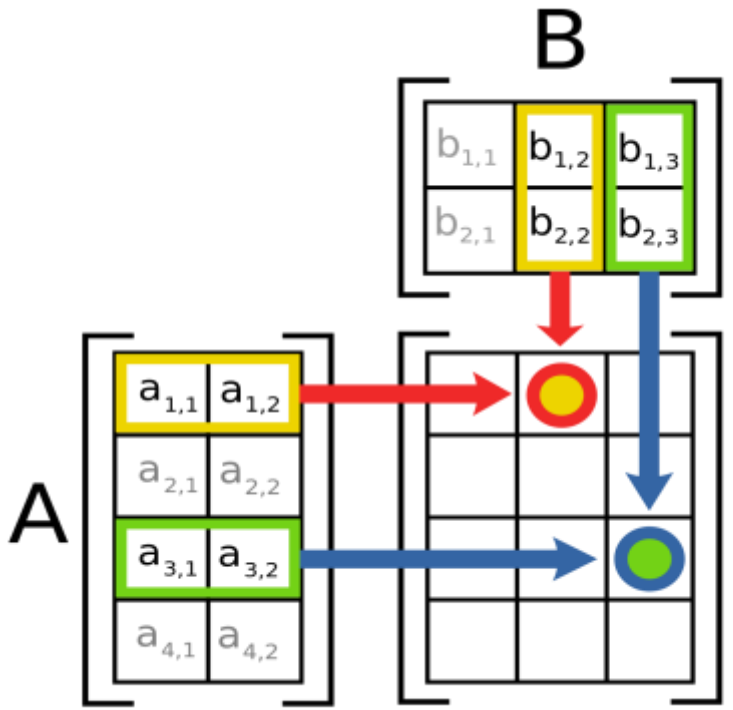
4) Exception handling

- Synchronous exceptions thrown in main thread by API functions
- Asynchronous exceptions need a call to `throw_asynchronous()` or `wait_and_throw()`
  - possible to attach asynch error handler to the queue

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

int main(int, char**) {
  constexpr size_t size = 10000;
  // … initialization of x_vec, y_vec and a
  try {  // exception handling
    queue q(gpu_selector_v);
    {
      buffer x_buf(x_vec);
      buffer y_buf(y_vec);
      range<1> num_items{ x_vec.size() };
      q.submit([&](handler& h) {
        accessor x(x_buf, h, read_only);
        accessor y(y_buf, h, read_write);
        h.parallel_for(num_items, [=](item<1> i) {
          y[i] = a * x[i] + y[i];
        });
      });
    } // Queue syncronize at buffer destruction
   // … print results
  }
  catch (exception const& e) {
    std::cout << "An exception is caught for saxpy.\n";
  }
  return 0;
}
```

Multidimensional Kernels

- Next
  - Towards multidimensional kernels
  - Example: matrix multiplication



Source: https://en.wikipedia.org/wiki/Matrix_multiplication

Matmul with 2d Kernels

- A `parallel_for` takes a `range`
  - specifies a 1-, 2- or 3-dimensional grid of work items
  - each work item executes the kernel function

1) Multi-dimensional buffer
2) Multi-dimensional accessors
   - both accessor and host_accessor
3) Host accessor will match the buffer type

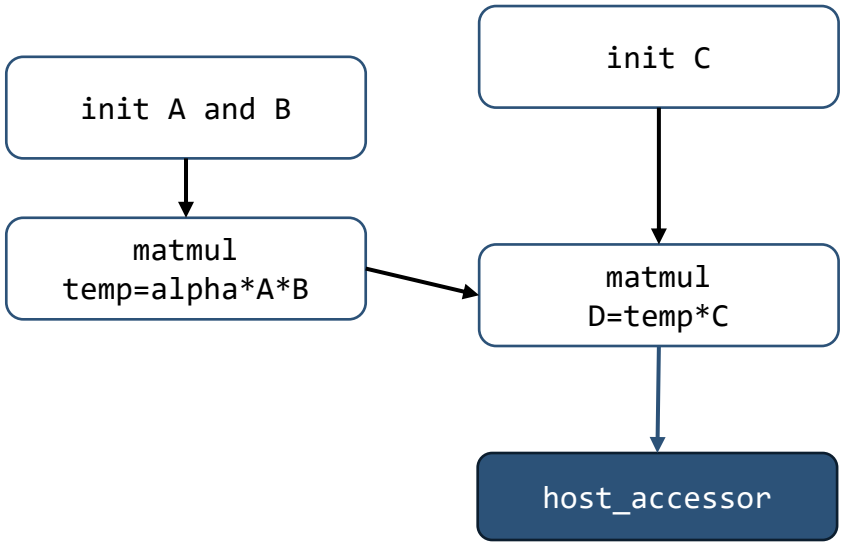
```
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl;

int main(int, char**) {
  const size_t size = 256;
  std::vector<float> A_mat(size * size, 1.0f);
  std::vector<float> B_mat(size * size, 2.0f);
  std::vector<float> C_mat(size * size, 0.0f);

  queue q;
  range<2> mat_size(size, size);
  buffer<float, 2> A_buf(A_mat.data(), mat_size);
  buffer<float, 2> B_buf(B_mat.data(), mat_size);
  buffer<float, 2> C_buf(C_mat.data(), mat_size);

  q.submit([&](handler& h) {
    accessor A(A_buf, h, read_only);
    accessor B(B_buf, h, read_only);
    accessor C(C_buf, h, write_only);
    h.parallel_for(mat_size, [=](item<2> id) {
      C[id] = 0;
      for (size_t k = 0; k < size; k++)
        C[id] += A[{id[0], k}] + B[{k, id[1]}];
    });
  });
  host_accessor C_res(C_buf, read_only);

  // … print results
  return 0;
}
```

Asynchronous Execution and Task Graphs

- Accessor provides information about data dependency between kernel
- Execution graph scheduling
  - Read-after-Write (RAW): when one task needs to read data produced by a different task
  - Write-after-Read (WAR): when one task needs to update data after another task has read it
  - Write-after-Write (WAW): when two tasks try to write the same data

Task graph with two matmul

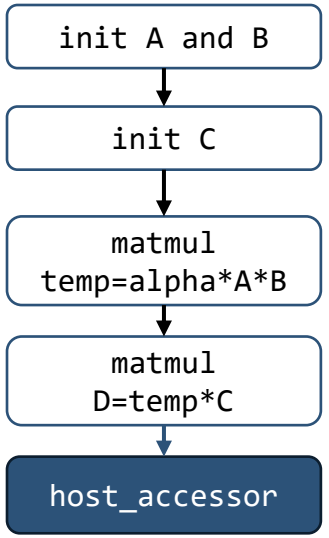
- Four `parallel_for`
  - Two initialize the data
  - Two matmul
- Dependency




```
// init A and B
q.submit([&](handler& h) {
  accessor A(A_buf, h, write_only);
  accessor B(B_buf, h, write_only);
  h.parallel_for(mat_size, [=](item<2> id) { A[id] = 1.0f; B[id] = 2.0f; });
});
// init C
q.submit([&](handler& h) {
  accessor C(C_buf, h, write_only);
  h.parallel_for(mat_size, [=](item<2> id) { C[id] = 0.3f; });
});
// matmul temp = alpha*A*B
q.submit([&](handler& h) {
  accessor A(A_buf, h, read_only);
  accessor B(B_buf, h, read_only);
  accessor temp(temp_buf, h, read_write);
  h.parallel_for(mat_size, [=](item<2> id) {
    temp[id] = 0;
    for (size_t k = 0; k < size; k++)
      temp[id] += alpha * A[{id[0], k}] * B[{k, id[1]}];
  });
});
// matmul D = temp * C
q.submit([&](handler& h) {
  accessor temp(temp_buf, h, read_only);
  accessor C(C_buf, h, read_only);
  accessor D(D_buf, h, read_write);
  h.parallel_for(mat_size, [=](item<2> id) {
    D[id] = 0;
    for (size_t k = 0; k < size; k++)
      D[id] += temp[{id[0], k}] + C[{k, id[1]}];
  });
});
host_accessor D_res(D_buf, read_only);
// … print results
```

In-order queue

- It is possible define an in-order semantics for the queue
  - commands submitted to the queue are executed in the order in which they are submitted
  - commands have an implicit dependence on the previous command submitted to that queue
  - no guarantees about the ordering of commands submitted to different queues with respect to each other




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

int main(int, char**) {
  const size_t size = 128;

  queue q
{default_selector_v,{property::queue::in_order{}}};
  range<2> mat_size(size, size);
  buffer<float, 2> A_buf(mat_size);
  buffer<float, 2> B_buf(mat_size);
  buffer<float, 2> C_buf(mat_size);
  buffer<float, 2> D_buf( mat_size);
  buffer<float, 2> temp_buf(mat_size);
  float alpha = 0.5f;

  // init A and B
  q.submit([&](handler& h) {
    accessor A(A_buf, h, write_only);
    accessor B(B_buf, h, write_only);
    h.parallel_for(mat_size, [=](item<2> id) {
      A[id] = 1.0f;
      B[id] = 2.0f;
    });
// ...
```

SYCL Memory Access Models: Buffer/Accessors vs USM

- Buffer and accessors
  - data access separated from data storage
  - relying on the C++-style resource acquisition is initialization (RAII) idiom to capture data dependencies
    - the runtime can track data movement and provide correct behavior without manually managing event dependencies
    - no explicitly move data
  - enables the data-parallel task-graphs as a part of the execution model, to be built up easily and safely
- Unified Shared Memory
  - pointer-based alternative to the buffer programming model
  - easier integration into existing code bases by representing allocations as pointers rather than buffers, with full support for pointer arithmetic into allocations
  - fine-grain control over ownership and accessibility of allocations, to optimally choose between performance and programmer convenience

Unified Shared Memory (USM)

- Pointer-based memory management in SYCL
  - use malloc / free to allocate and deallocate data
  - simplifies development for the programmer when porting existing C/C++ code to SYCL
  - use shared allocations when porting code to get functional quickly
  - use explicit USM allocations when controlled data movement is needed
- Model types: explicit or implicit
- Types:

USM allocation type	Description
host	Allocations in host memory that are accessible by a device
device	Allocations in device memory that are **not** accessible by the host
shared	Allocations in shared memory that are accessible by both host and device

SAXPY with Implicit USM

1. Implicit: `malloc_shared`
   - specify the queue
   - the programmer must do the allocation
     - Potential issued with std::vector
2. Requires a matching `free`
3. Synchronization using `wait()`

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

int main(int, char**) {
    const int size = 10000;
    queue q;
    // USM allocation, implicit data movement
    float* x = malloc_shared<float>(size, q);
    float* y = malloc_shared<float>(size, q);
    // … initialization of x_vec, y_vec and a

    range<1> num_items{ size };
    q.submit([&](handler& h) {
        h.parallel_for(num_items, [=](item<1> i) {
            y[i] = a * x[i] + y[i];
        });
    });
    q.wait();
    // … print results
    free(x, q);
    free(y, q);
    return 0;
}
```

SAXPY with Explicit USM

1. Implicit: `malloc_device`
2. Requires a matching `free`
3. Synchronizations: `wait()`
4. Explicit data movement between host and device
   - Use `memcpy`

- Note: data dependency are not automatic (as with accessors)
  - Solutions: (1) use in_order queue property; (2) use `wait()`; (3) use `depends_on`

```
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl;
int main(int, char**) {
    const int size = 10000;
    queue q;
    // data allocation
    float* x_data = static_cast<float*>(malloc(size * sizeof(float)));
    float* y_data = static_cast<float*>(malloc(size * sizeof(float)));
    // … initialization of x_vec, y_vec and a
    // USM allocation
    float* x_device = malloc_device<float>(size, q);
    float* y_device = malloc_device<float>(size, q);
    // copy from host to device
    q.memcpy(x_device, x_data, size * sizeof(float));
    q.memcpy(y_device, y_data, size * sizeof(float)).wait();
    q.submit([&](handler& h) {
        h.parallel_for(range<1>{size}, [=](item<1> i) {
            y_device[i] = a * x_device[i] + y_device[i];
        });
    }).wait();
    // copy from device to host
    q.memcpy(x_data, x_device, size * sizeof(float));
    q.memcpy(y_data, y_device, size * sizeof(float)).wait();
    // … print results
    free(x_device, q);
    free(y_device, q);
    free(x_data);
    free(y_data);
    return 0;
}
```

SYCL Advanced Features

- Work-group
- Sub-groups and group algorithms
- Kernel reductions
- Local memory
- Atomic operations
- Specialization constants

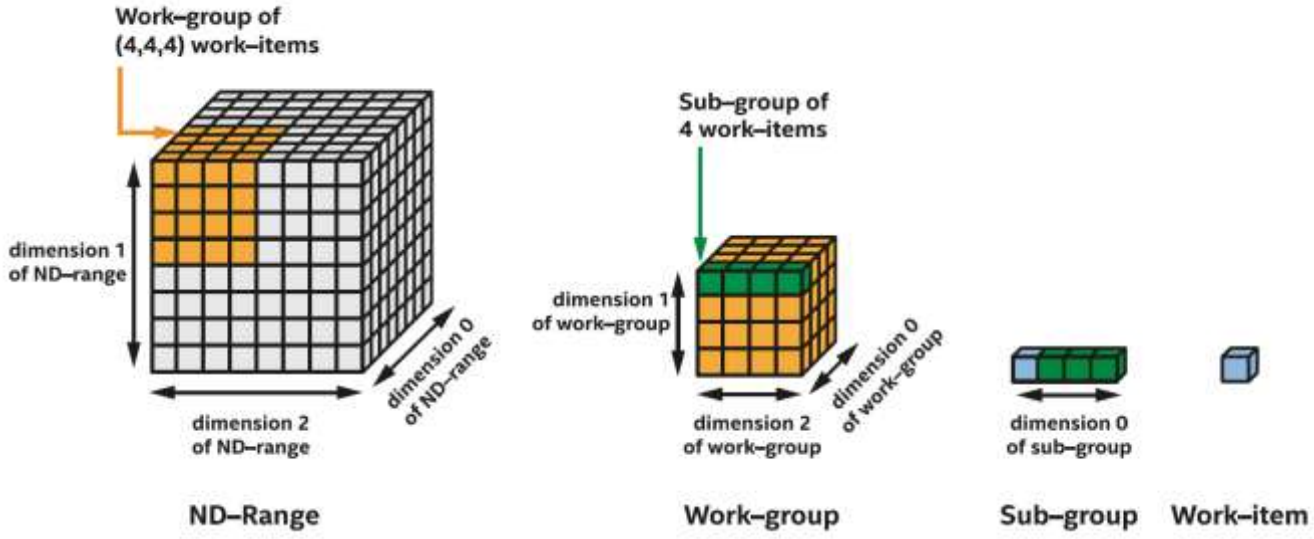


Image from *Data Parallel C++ Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*

ND ranges

- The work-items in an ND-range are organized into work-groups
  - Work-items in a work-group
    - have access to work-group local memory
    - can synchronize using work-group barriers and guarantee memory consistency using work-group memory fences
    - have access to group functions, providing implementations of comr communication routines
  - Work-group in multi-dimensional

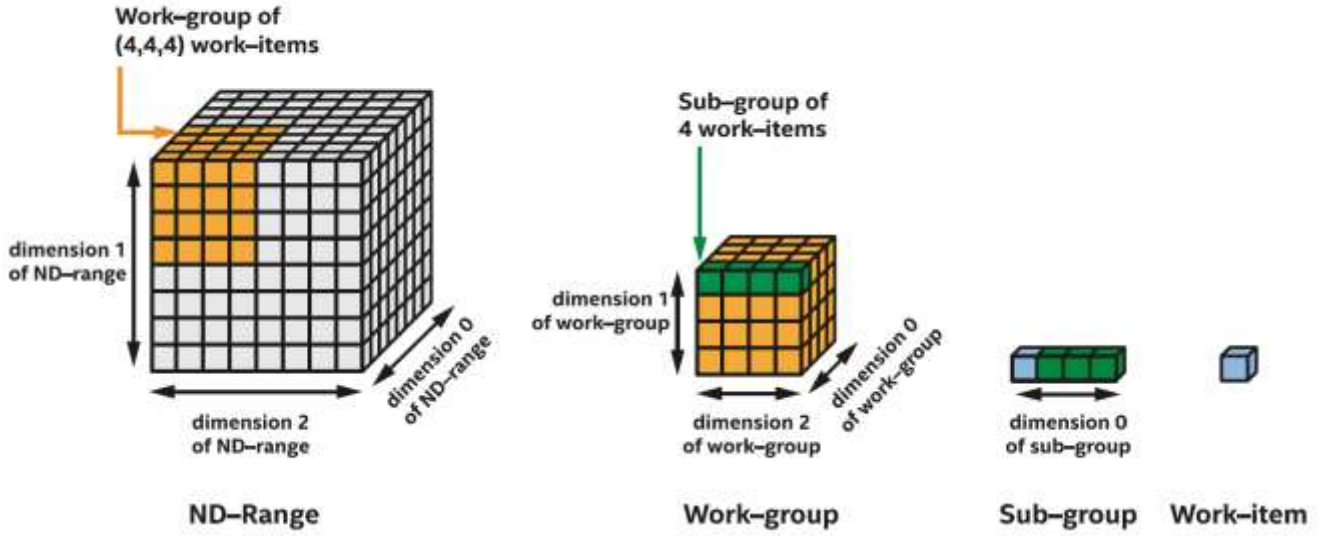


Image from *Data Parallel C++ Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*

Subgroups

- Subgroups: Subsets of the work-items in a work-group executed with additional scheduling guarantees
- Work-items in a sub-group could be executed simultaneously
  - because of vectorization (CPU), warp (NVIDIA) or wavefront (AMD) synchronization or Xe Vector Engine XVE (Intel)
- Subgroup is one-dimensional

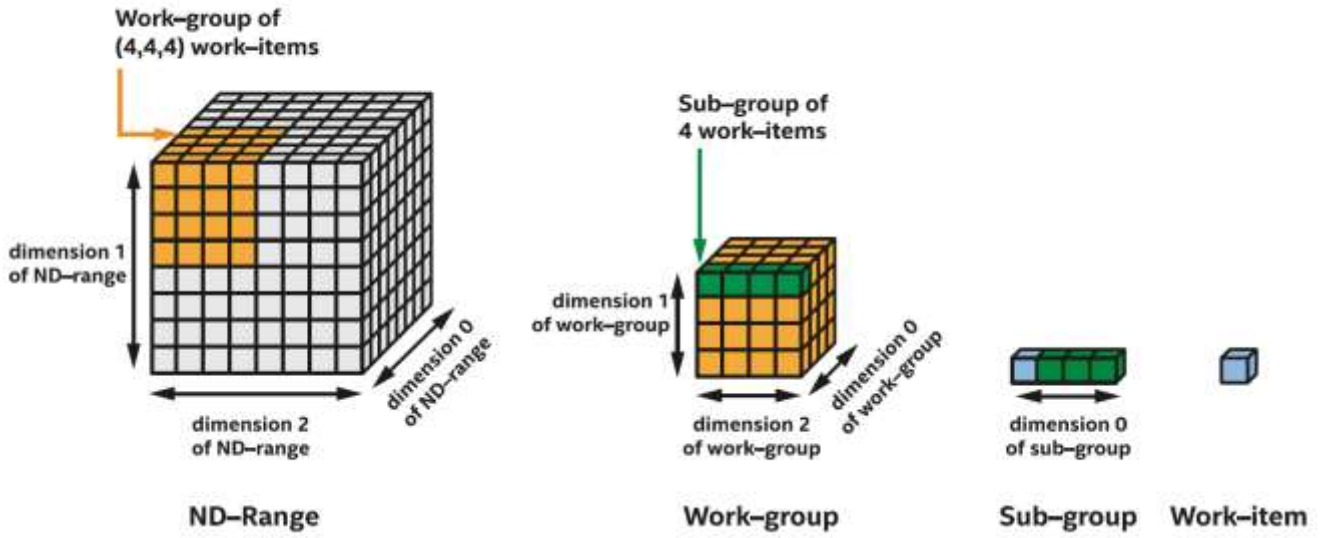


Image from *Data Parallel C++ Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*

Subgroup Functions and Algorithms

- Synchronization
  - by calling a `group_barrier` function
- Collective functions
  - Broadcast: takes a value from one work-item in the group and communicates it to all other work-items in the group (`group_broadcast`)
  - Votes: `any_of` and `all_of`, `none_of` functions enable work-items to compare the result of a boolean condition across their group
  - Shift and permutation: `shift_group_left`, `shift_group_right`, `permute_group_by_xor`, `select_from_group`
  - Other collectives: `joint_reduce`, `reduce_over_group`, `joint_exclusive_scan`, `exclusive_scan_over_group`, `joint_exclusive_scan`, `exclusive_scan_over_group`

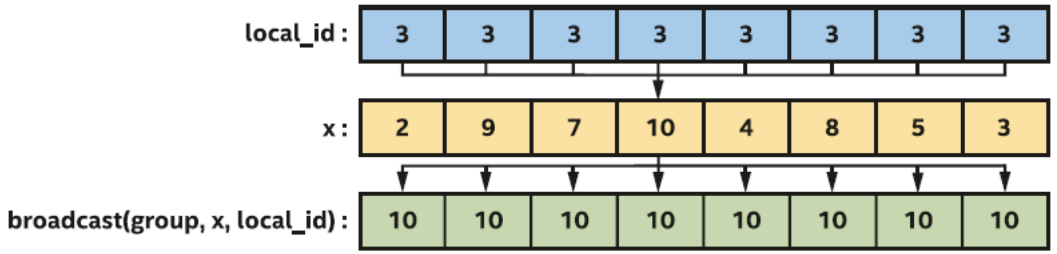


Image from *Data Parallel C++ Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*

Matmul with group broadcast

1. Matrix multiplication with one-dimensional tiling
   - tile size must be <= than subgroup size
2. Perform computation
   - by broadcasting from the matrix tile
   - loading from matrix B in global memory
3. T is the tile number
   - k describes which work-item in the sub-group to broadcast data from

```
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl;
int main(int, char**) {
  //...
  constexpr int tile_size = 4;
  q.submit([&](handler& h) {
    accessor A(A_buf, h, read_only);
    accessor B(B_buf, h, read_only);
    accessor C(C_buf, h, write_only);
    h.parallel_for(nd_range<2>{{size, size}, { 1, tile_size }},
      [=](nd_item<2> id) {
      auto sg = id.get_sub_group();
      size_t  i = id.get_global_id()[0];
      size_t  j = id.get_global_id()[1];
      size_t  l = id.get_local_id()[1];
      float sum = 0.f;
      for (size_t t = 0; t < size; t += tile_size) {
        float tileA = A[{i, t + l}]; // load a 1D tile
        for (size_t k = 0; k < tile_size; k++)
          sum += group_broadcast(sg, tileA, k) * B[{t + k, j}];
      }
      C[{i,j}] = sum;
    });
  });
  host_accessor C_res(C_buf, read_only);
  // …
  return 0;
}
```

Kernel Reduction: Dot Product

1) Reduction temporary object
   - describe reduction semantics
   - specify `plus<>()` operator
2) `parallel_for` extra parameters
   - reduction object
   - possible to list more reductions
3) Lambda `[]` extra parameters
   - reference to the reducer object
4) Value update via `combine()`

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

int main(int, char**) {
  constexpr size_t size = 10000;
  std::vector<float> x_vec(size, 0.5f);
  std::vector<float> y_vec(size, 2.0f);
  float dot = 0.0f;

  queue q;
  buffer x_buf(x_vec);
  buffer y_buf(y_vec);
  buffer dot_buf(&dot, range(1));
  range<1> num_items{ x_vec.size() };
  q.submit([&](handler& h) {
    accessor x(x_buf, h, read_only);
    accessor y(y_buf, h, read_write);
    auto sum_reduction = reduction(dot_buf, h, plus<>());
    h.parallel_for(num_items, sum_reduction,
      [=](item<1> i, auto &dot) {
        float product = x[i] * y[i];
        dot.combine(product);
      });
    });
  host_accessor dot_res(dot_buf, read_only);

  // Print the dot product
  std::cout << "\ndot product value: " << dot_res[0];
  return 0;
}
```

Note on Reduction Kernel

Optimizing Parallel Reduction in CUDA. Mark Harris. Technical report

```
template <unsigned int blockSize>
__device__ void warpReduce(volatile int* sdata, unsigned int tid) {
  if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
  if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
  if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
  if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
  if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
  if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
}
template <unsigned int blockSize>
__global__ void reduce6(int* g_idata, int* g_odata, unsigned int n) {
  extern __shared__ int sdata[];
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x * (blockSize * 2) + tid;
  unsigned int gridSize = blockSize * 2 * gridDim.x;
  sdata[tid] = 0;
  while (i < n) {
    sdata[tid] += g_idata[i] + g_idata[i + blockSize];
    i += gridSize;
  }
  __syncthreads();
  if (blockSize >= 512) {
    if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads();
  }
  if (blockSize >= 256) {
    if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads();
  }
  if (blockSize >= 128) {
    if (tid < 64) { sdata[tid] += sdata[tid + 64]; } __syncthreads();
  }
  if (tid < 32) warpReduce(sdata, tid);
  if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

from CUDA to SYCL

Reduction kernels are optimized and portable among different target devices

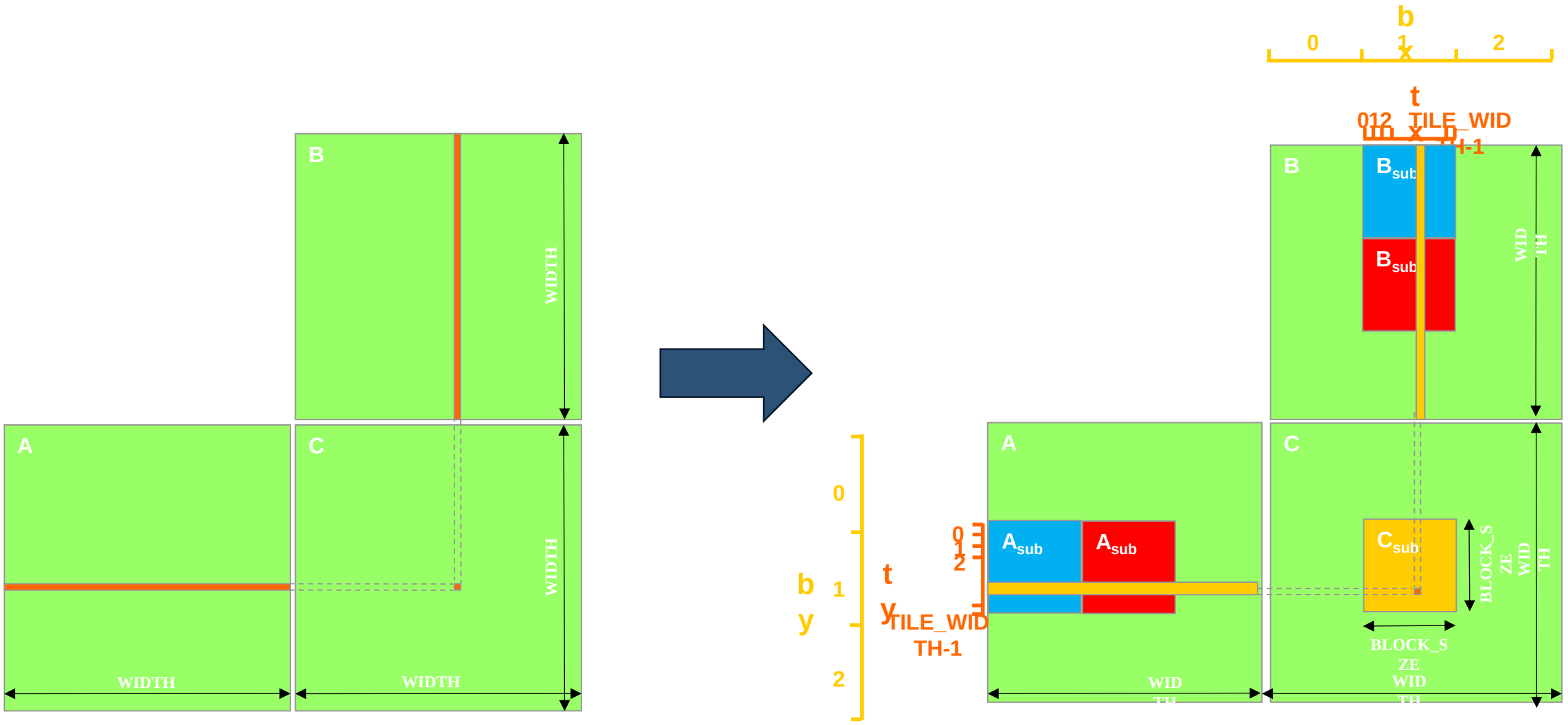
```
queue.submit([&](handler& cgh) {
  accessor  idata = accessor(inputBuf, cgh);
  auto sumReduction = reduction(sumBuf, cgh, plus<>());
  cgh.parallel_for(range<1>{n},sumReduction,
    [=](id<1> idx, auto& sum) {
      sum.combine(idata[idx]);
  });
});
```

Local Memory: Matrix Multiplication with Tiling

- The matrix multiplication we saw is memory bounded
  - Global vs local memory bandwidth
- The key is to use local memory
  - and registers when possible
- Tiling
  - Split result matrix into smaller blocks
    - block as a two-dimensional tiling
  - Copy portion of matrix to local memory
    - maximize data reuse

```
accessor inA_acc{ inA_buf, cgh, sycl::read_only };
accessor inB_acc{ inB_buf, cgh, sycl::read_only };
accessor out_acc{ out_buf, cgh, sycl::write_only, sycl::no_init };
…
cgh.parallel_for(sycl::range<2>{hA, wB}, [=](sycl::id<2> idx) {
  float sum = 0;
  for (int i = 0; i < commSide; i++) {
    float elemA = inA_acc[idx[0]][i];
    float elemB = inB_acc[i][idx[1]];
    sum += elemA * elemB;
  }
  out_acc[idx] = sum;
});
```

Matrix Multiplication with Tiling



Matrix Multiplication with Tiling: Allocating Local Memory

```
q.submit([&](sycl::handler& cgh) {
   sycl::accessor inA_acc{ inA_buf, cgh, sycl::read_only };
   sycl::accessor inB_acc{ inB_buf, cgh, sycl::read_only };
   sycl::accessor out_acc{ out_buf, cgh, sycl::write_only, sycl::no_init };

   sycl::local_accessor<int, 2> tileA{ sycl::range<2>{LOCAL_SIZE, LOCAL_SIZE}, cgh };
   sycl::local_accessor<int, 2> tileB{ sycl::range<2>{LOCAL_SIZE, LOCAL_SIZE}, cgh };

   size_t hA = SIZE;
   size_t wB = SIZE;
   size_t commSide = SIZE;
   size_t localSize = LOCAL_SIZE;

   cgh.parallel_for(sycl::nd_range<2>{{hA, wB}, { localSize, localSize }},
   [=](sycl::nd_item<2> item) {
       // parallel for in the next slide
   });
 q.wait_and_throw();
```

Matrix Multiplication with Tiling: Parallel For

```
cgh.parallel_for(sycl::nd_range<2>{{hA, wB}, { localSize, localSize }},
    [=](sycl::nd_item<2> item) {
     uint gidX = item.get_global_id(0);
     uint gidY = item.get_global_id(1);
     uint lidX = item.get_local_id(0);
     uint lidY = item.get_local_id(1);

     uint num_tiles = (hA * wB) / (localSize * localSize);
     int sum = 0;
     for (int i = 0; i < num_tiles; i++) {
       tileA[lidX][lidY] = inA_acc[gidX][i * localSize + lidY];
       tileB[lidX][lidY] = inB_acc[i * localSize + lidX][gidY];
       sycl::group_barrier(item.get_group());

       for (int j = 0; j < localSize; j++) {
         sum += tileA[lidX][j] * tileB[j][lidY];
       }
       sycl::group_barrier(item.get_group());
     }
     out_acc[gidX][gidY] = sum;
     });
   });
```

Atomics

- Atomic operations are needed to avoid race condition when different work items write to the same data location
- SYCL memory consistency model is based upon the memory consistency model of the C++ core language
  - Memory ordering: `relaxed, acquire, release, acq_rel, seq_cst`
- In addition
  - Memory scope: `work_item, sub_group, work_group, device, system`
  - Address space: `global_space, local_space, private_space`
- Using atomics, values can be combined across work items without data races
  - `fetch_add, fetch_sub, fetch_max, fetch_min fetch_and, fetch_or`

Counter with Atomics

1. We use an `atomic_ref` to update the value
   - must specify type, memory order, scope and address space
2. Call `fetch_add` on the `atomic_ref`

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

int main(int, char**) {
    constexpr int size = 1000;
    std::vector<int> x_vec(size, 1);
    std::fill_n(x_vec.begin(), 500, 0); // we only place 500 zeros
    int counter = 0;

    queue q;
    buffer x_buf(x_vec);
    buffer c_buf(&counter, range(1));
    range<1> num_items{ x_vec.size() };
    q.submit([&](handler& h) {
      accessor x(x_buf, h, read_only);
      accessor c(c_buf, h, read_write);
      h.parallel_for(num_items, [=](item<1> i) {
        if (x[i] == 0)
          atomic_ref<int,
            memory_order_acq_rel,
            memory_scope_device,
            access::address_space::global_space>(c[0]).fetch_add(1);
      });
    });
    host_accessor c_res(c_buf, read_only);

    // Print the number of zeros
    std::cout << "\nnumber of zeros: " << c_res[0];
    std::cout << "\natomics successfully completed\n";
    return 0;
}
```

SYCL and the oneAPI Open Ecosystem

- SYCL is a royalty-free standard with many open implementations
- oneAPI as part of the UxL foundation under the Linux Foundation
- Seven code elements
  1. oneDPL: Data Parallel C++ Library (C++ standard library, Parallel STL, and extensions)
  2. oneDNN: Deep Neural Network Library
  3. oneCCL: Collective Communications Library
  4. LevelZero: System interface for oneAPI languages and libraries
  5. oneDAL: Data Analytics Library
  6. oneTBB: Threading Building Blocks
  7. oneMKL: Math Kernel Library

oneDNN: Deep Neural Network Library

- oneDNN is an open-source performance library for deep learning applications
  - Helps developers create high performance deep learning frameworks
  - Abstracts out instruction set and other complexities of performance optimizations
  - Same API for both Intel CPU's and GPU's, use the best technology for the job
  - Supports Linux, Windows, and macOS
  - Open source for community contributions
- Distribution
  - Binary distribution: https://software.intel.com/en-us/oneapi/onednn
  - Github repository: https://github.com/oneapi-src/oneDNN

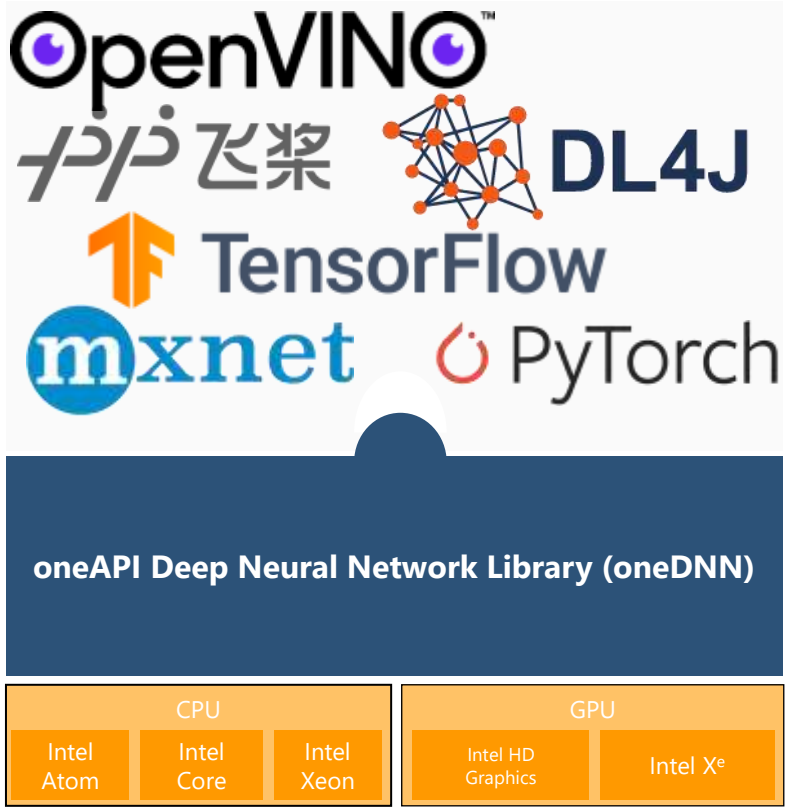
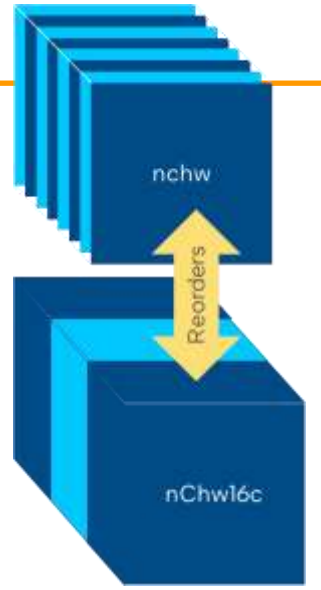


Image from courtesy of Intel@

oneDNN: Deep Neural Network Library

- Memory Layouts
  - beyond plain tensor layout (nchw, nhwc, oihw, hw)
  - oneDNN convolutions and matmuls support blocked layouts
  - examples: nChw16c, OIhw16i16o
- Lower precision inference
  - f16/bf16 compression
  - Int8 quantization
- Interoperability layer with OpenCL and SYCL



Joint Matrix Extension

- SYCL extension for matrix hardware programming
- Unique interface for
  - Intel® Advanced Matrix Extensions (Intel® AMX) for CPUs
  - Intel® Xe Matrix Extensions (Intel® XMX) for GPUs
  - NVIDIA* Tensor Cores
- Used by neural network applications with reduced precision data types (e.g., bloat16)
  - Tensor-Flow, oneDNN, …
- A set of operations that work like (sub)group algorithms
  - joint matrix API has to be accessed by all the workitems in a subgroup
  - functions will be called once by the subgroup no code divergence between the workitems

SYCL Joint Matrix: Datatype & operations

New datatype

- Joint matrix

Key operations

- Load matrix
- Store matrix
- Fill matrix
- MAD operation

```
buffer<bfloat16, 2> bufA(A.get_data(), sycl::range<2>(M, K));
buffer<bfloat16, 2> bufB(B.get_data(), sycl::range<2>(K, N));
buffer<float, 2> bufC((float*)C.get_data(), sycl::range<2>(M, N));
…
sycl::queue q;
q.submit([&](sycl::handler& cgh) {
 accessor accC(bufC, cgh, sycl::read_write);
 accessor accA(bufA, cgh, sycl::read_only);
 accessor accB(bufB, cgh, sycl::read_only);
 cgh.parallel_for(nd_range<2>({ NDRangeM, NDRangeN * SG_SZ }, { 1, 1 * SG_SZ }),
  [=](sycl::nd_item<2> spmd_item) [[intel::reqd_sub_group_size(SG_SZ)]]{
   const auto global_idx = spmd_item.get_global_id(0);
   const auto global_idy = spmd_item.get_global_id(1);
   const auto sg_startx = global_idx - spmd_item.get_local_id(0);
   const auto sg_starty = global_idy - spmd_item.get_local_id(1);
   sub_group sg = spmd_item.get_sub_group();
   ext::oneapi::experimental::matrix::joint_matrix<sub_group,bfloat16,…> sub_a;
   ext::oneapi::experimental::matrix::joint_matrix<sub_group,bfloat16,…> sub_b;
   ext::oneapi::experimental::matrix::joint_matrix<sub_group, float, …> sub_c;
   joint_matrix_fill(sg, sub_c, 1.0);
   for (int k = 0; k < K / TK; k += 1) {
    joint_matrix_load(sg, sub_a,accA.template get_multi_ptr<access::decorated::no>()+(…);
    joint_matrix_load(sg,sub_b,accB.template get_multi_ptr<access::decorated::no>()+(…);
    joint_matrix_mad(sg, sub_c, sub_a, sub_b, sub_c);
   }
   joint_matrix_apply(sg, sub_c, [=](float& x) { x *= ALPHA; });
   joint_matrix_store(sg, sub_c,accC.template get_multi_ptr<sycl::access::decorated::no>()+(…);
 }); // parallel for
}).wait();
```

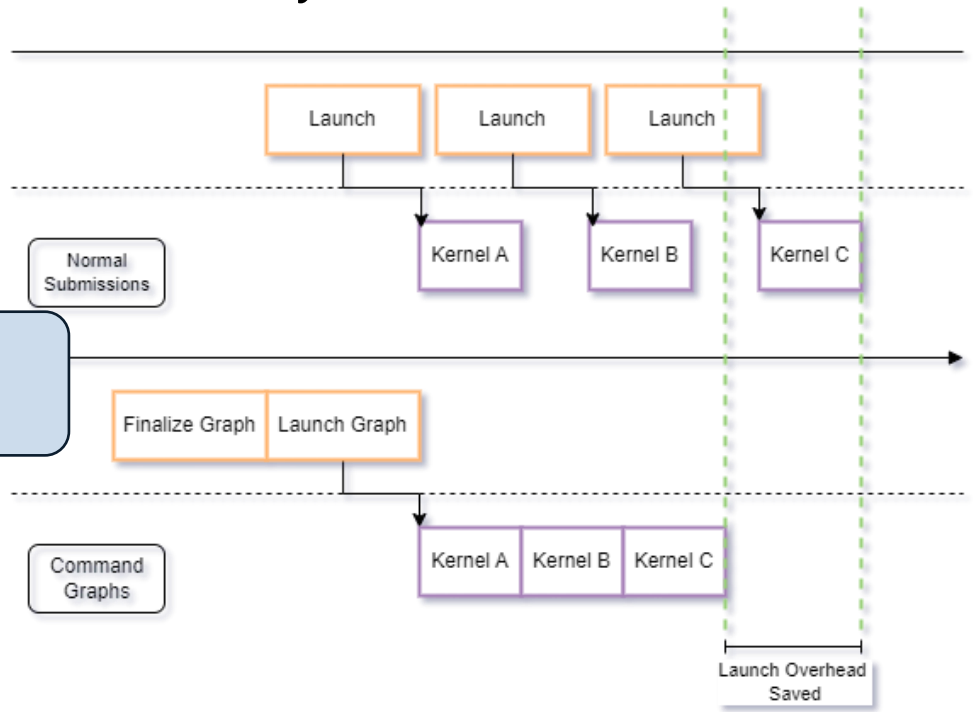
SYCL Graph

- API for deferred execution of SYCL command graph
  - This extension decouples command submission from command execution, allowing the user to define ahead of time the full DAG of commands that they wish to execute

```
Graph.begin_recording(Queue);
// Submit commands here via Queue.submit()...
Graph.end_recording(Queue);
```

1. Record and replay API
   - records commands submitted to SYCL queues as nodes in the graph and automatically infers dependencies
2. Explicit Graph Creation
   - the user explicitly define the nodes and edges of a graph, giving more control over its structure

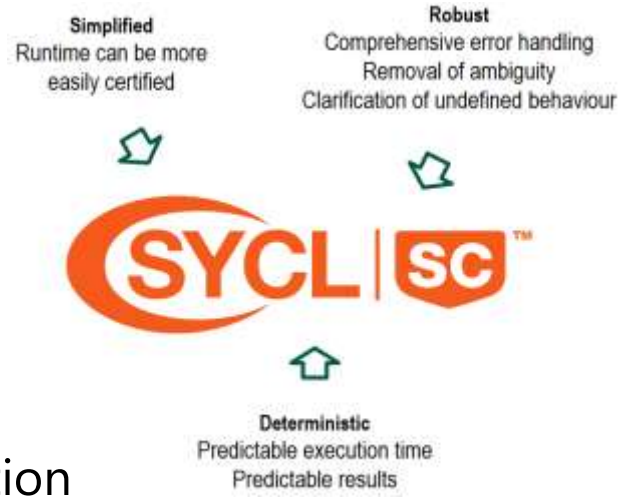
```
Graph.make_edge(NodeInit, NodeA);
…
auto ExecGraph = Graph.finalize();
Queue.ext_oneapi_graph(ExecGraph);
```



Source: https://codeplay.com/portal/blogs/2024/01/22/sycl-graphs

SYCL Safety Critical

- Safety-critical industries (avionics, automotive, nuclear, and rail) require
  - to be compliant to safety standards: ISO 26262, ISO 21448/SOTIF, DO-178C, and UL4600
  - to adhere to guidelines defined by AUTOSAR and MISRA
- SYCL SC Working Group
  - Mission: to develop C++-based heterogeneous parallel compute programming framework for safety-critical systems
- (Recent) Focus points
  - Online vs offline compilation: offline is better for safety certification
  - Avoiding dynamic memory on host
  - Deterministic error management



Victor Perez & Hugh Delaney, IWOCL 24

CUDA to SYCL

Migration to SYCL from CUDA

- Migration tool
  - DPC++ compatibility tool & SYCLomatic
- Migration examples
  - By tool: vector add, Rodinia's NW, BLAS
  - Large project migration: LIGATE
- Porting hints

CUDA to SYCL Migration Workflow

- Intel® DPC++ Compatibility Tool
  - assists developers migrating code written in CUDA to DPC++ once, generating human readable code wherever possible
  - ~90-95% of code typically migrates automatically
    - based on measurements on a set of 70 HPC benchmarks and samples (Rodinia, SHOC, PENNANT, ect)
  - inline comments are provided to help developers finish porting the application
- The tool is open source on GitHub:
  - http://github.com/oneapi-src/SYCLomatic

https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compatibility-tool.html

Migration Workflow in three steps

1. Prepare the project using the `intercept-build`
   - Create a compilation database file
   - `intercept-build make`
2. Migrate using `dpct`
   - migrate your source to DPC++
   - `dpct -p compile_commands.json -in-root=$PROJ_DIR -out-root=dpcpp_out *.cu`
3. Review by verify and optionally manually edit the generated code
   - verify the source for correctness and fix not migrated parts

Migration Workflow Overview: vector add

- Step-by-step on Leonardo
  - download the vector addition code

```
$ git clone https://github.com/oneapi-src/oneAPI-samples.git
...
$ cd oneAPI-samples/Tools/Migration/vector-add-dpct/src
```

  - run the `dpct` tool with the right CUDA include

```
$ dpct vector_add.cu
NOTE: Could not auto-detect compilation database for file 'vector_add.cu' in '/leonardo/…/vector-add-dpct/src'
or any parent directory.
The directory "dpct_output" is used as "out-root"
Parsing: /leonardo/home/userexternal/bcosenza/oneAPI-samples/Tools/Migration/vector-add-dpct/src/vector_add.cu
Analyzing: /leonardo/home/userexternal/bcosenza/oneAPI-samples/Tools/Migration/vector-add-dpct/src/vector_add.cu
Migrating: /leonardo/home/userexternal/bcosenza/oneAPI-samples/Tools/Migration/vector-add-dpct/src/vector_add.cu
Processed 1 file(s) in -in-root folder "/leonardo/home/userexternal/…/Migration/vector-add-dpct/src"

See Diagnostics Reference to resolve warnings and complete the migration:
https://www.intel.com/.../dpcpp-compatibility-tool/developer-guide-reference/current/diagnostics-reference.html
```

  - The file `vector_add.dp.cpp` is generated in `dpct_output` directory
- Warning: to support CUDA 12.3 you need `dpct` from at least the 2024.1.0 toolkit

Migration Workflow Overview: vector add migration output (1)

vector_add.cu

```
#include <cuda.h>
#include <stdio.h>
#include <stdlib.h>
#define VECTOR_SIZE 256

__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];
}
```

vector_add.dp.cpp

```
#include <sycl/sycl.hpp>
#include <dpct/dpct.hpp>
#include <stdio.h>
#include <stdlib.h>
#define VECTOR_SIZE 256
void VectorAddKernel(float* A, float* B, float* C,
                     const 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)];
}
```

- Hint: use `--assume-nd-range-dim=1` to specify the dimension of the nd_range to one

Migration Workflow Overview: vector add migration output (2)

vector_add.cu

```
int main()
{
    float *d_A, *d_B, *d_C;
    cudaError_t status;


    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] = { };

    status = cudaMemcpy(Result, d_C,
VECTOR_SIZE*sizeof(float), cudaMemcpyDeviceToHost);
    if (status != cudaSuccess) {
        printf("Could not copy result to host\n");
        exit(EXIT_FAILURE);
    }

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // … printf omitted …

    return 0;
}
```

vector_add.dp.cpp

```
int main() try {
    dpct::device_ext &dev_ct1 = dpct::get_current_device();
    sycl::queue &q_ct1 = dev_ct1.in_order_queue();
    float *d_A, *d_B, *d_C;
    dpct::err0 status;

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

    float Result[VECTOR_SIZE] = { };

    status = DPCT_CHECK_ERROR(q_ct1.memcpy(Result, d_C, VECTOR_SIZE *
sizeof(float)).wait());

    dpct::dpct_free(d_A, q_ct1);
    dpct::dpct_free(d_B, q_ct1);
    dpct::dpct_free(d_C, q_ct1);

    // … printf omitted …

    return 0;
}
catch (sycl::exception const &exc) {…}
```

Migration Workflow Overview: in-order vs out-of-order queue

- Be careful to SYCL queue when migrating

```
int main() try {
    dpct::device_ext &dev_ct1 = dpct::get_current_device();
    sycl::queue &q_ct1 = dev_ct1.in_order_queue();
    float *d_A, *d_B, *d_C;
    dpct::err0 status;
```

```
sycl::queue &default_queue() {
#ifdef DPCT_USM_LEVEL_NONE
    return out_of_order_queue();
#else
    return in_order_queue();
#endif // DPCT_USM_LEVEL_NONE
  }
```

- CUDA is in-order by default
  - but in SYCL, queue are out-of-order by default
  - the migration tool generates **in-order queues** when translating CUDA code
    - if not specified differently on the command line

Migration Workflow Overview: Rodinia Needleman Wunsch

- Step-by-step on Leonardo
  - this time we need the make interceptor
  - dpct returns a very verbose output
  - generated code is also annotated

```
$ cd oneAPI-samples/Tools/Migration/rodinia-nw-dpct/
$ make clean
$ intercept-build make
nvcc src/needle.cu -o needleman_wunsch_cu
```

```
$ dpct -p compile_commands.json$ make clean
The directory "dpct_output" is used as "out-root"
Parsing: /leonardo/home/userexternal/bcosenza/oneAPI-samples/Tools/Migration/rodinia-nw-dpct/src/needle.cu
Analyzing: /leonardo/home/userexternal/bcosenza/oneAPI-samples/Tools/Migration/rodinia-nw-dpct/src/needle.cu
In file included from /leonardo/home/userexternal/…/rodinia-nw-dpct/src/needle.cu:10:
/leonardo/home/userexternal/…/rodinia-nw-dpct/src/needle_kernel.cu:81:7: warning: DPCT1118:0: SYCL group
functions and algorithms must be encountered in converged control flow. You may need to adjust the code.
   81 |       __syncthreads();
…
/leonardo/home/userexternal/…/rodinia-nw-dpct/src/needle_kernel.cu:46:19: warning: DPCT1101:20: 'BLOCK_SIZE’
expression was replaced with a value. Modify the code to use the original expression, provided in comments,
if it is correct.
   46 |   __shared__  int ref[BLOCK_SIZE][BLOCK_SIZE];
```

Migration Workflow Overview: CUBLAS to MKL

- Step-by-step on Leonardo

```
$ git clone https://github.com/NVIDIA/cuda-samples.git
...
$ cd cuda-samples/Samples/4_CUDA_Libraries/simpleCUBLAS
$ intercept-build make
/usr/local/cuda/bin/nvcc -ccbin g++ -I../../../Common  -m64    --threads 0 --std=c++11 -gencode
arch=compute_50,code=compute_50 -o simpleCUBLAS.o -c simpleCUBLAS.cpp
/usr/local/cuda/bin/nvcc -ccbin g++   -m64      -gencode arch=compute_50,code=compute_50 -o simpleCUBLAS
simpleCUBLAS.o  -lcublas
mkdir -p ../../../bin/x86_64/linux/release
cp simpleCUBLAS ../../../bin/x86_64/linux/release
$ dpct -p compile_commands.json
```

- The tool translates CUBLAS calls into MKL calls

simpleCUBLAS.cpp

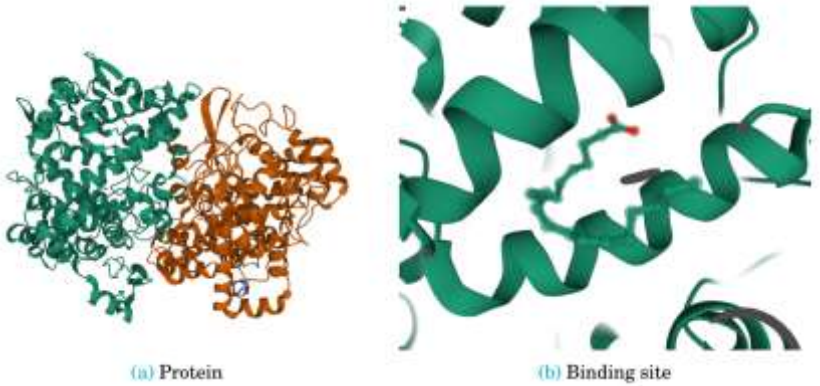
```
/* Performs operation using cublas */
status = cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, N,
N, N, &alpha, d_A,
  N, d_B, N, &beta, d_C, N);
```

simpleCUBLAS.cpp.dp.cpp

```
/* Performs operation using cublas */
status = DPCT_CHECK_ERROR(oneapi::mkl::blas::column_major::gemm(
  *handle, oneapi::mkl::transpose::nontrans,
  oneapi::mkl::transpose::nontrans, N, N, N, alpha, d_A, N, d_B,
N, beta, d_C, N));
```

Migration Hints from Large-Scale Projects: LIGATE Project

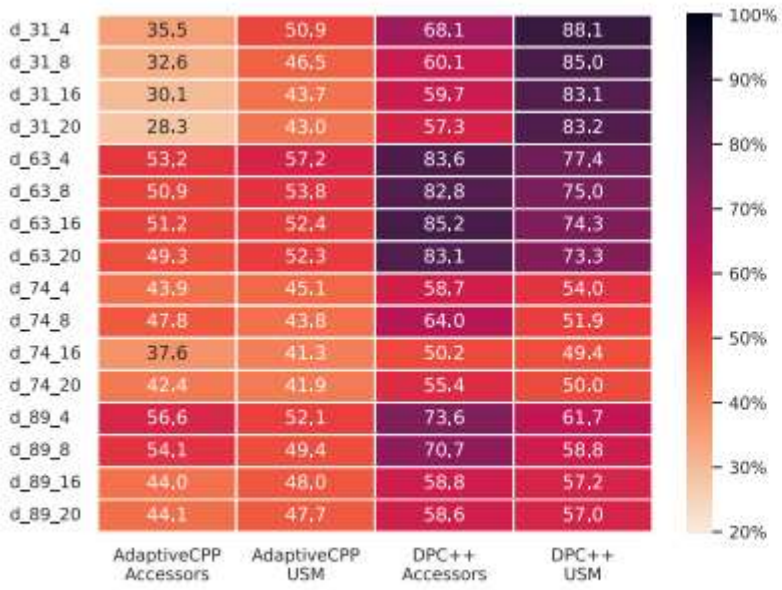
- Ligate project with Dompe spa, CINECA, POLIMI, KTH and others
  - porting from manually optimized CUDA to SYCL
- Molecular docking as “relaxed” lock and key model
  - protein considered a rigid body
  - ligand considered a flexible body
  - a ligand is docked on target protein’s binding sites
  - multiple pose for each ligand
- Application tuning
  - block size and number of iteration of the heaviest kernels depending on workload type
  - kernel tuning: workgroup size, blocking, local memory usage
  - we developed two version of LiGen with different register pressure



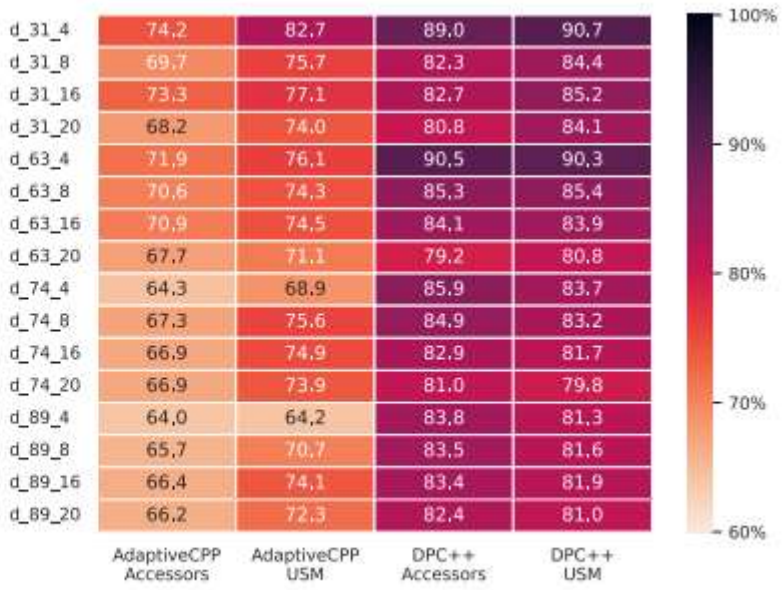
This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 956137. The JU receives support from the European Union’s Horizon 2020 research and innovation programme and Italy, Sweden, Austria, Czech Republic, Switzerland.

Migration Hints from Large-Scale Projects: LIGATE Project

- SYCL achieves up to 90% of the native CUDA manually-tuned application
  - register optimization required because of high register pressure
  - the choose of the memory model matters



(a) LiGen Latency



(b) LiGen Batch

Lesson learnt from the LIGATE project: main takeaways

1. Memory access models influence performance
   - buffer/accessor vs USM: accessor slightly increase the register pressure
2. Group algorithms and kernel reductions reduce code complexity
   - group algorithms efficient mapping into warp/wavefront/Xe Vector Engine based instructions
   - kernel reductions rely on compiler-based reduction algorithms
   - by using SYCL 2020 reduction and group algorithms we removed more than 430 lines of code
3. Get rid of cumbersome CUDA features
   - dynamic parallelism: removed without performance difference

Crisci, Carpentieri, Cosenza, Accordi, Gadioli, Vitali, Palermo, Beccari: Enabling performance portability on the LiGen drug discovery pipeline. **Future Gener. Comput. Syst.** 158: 44-59 (2024)

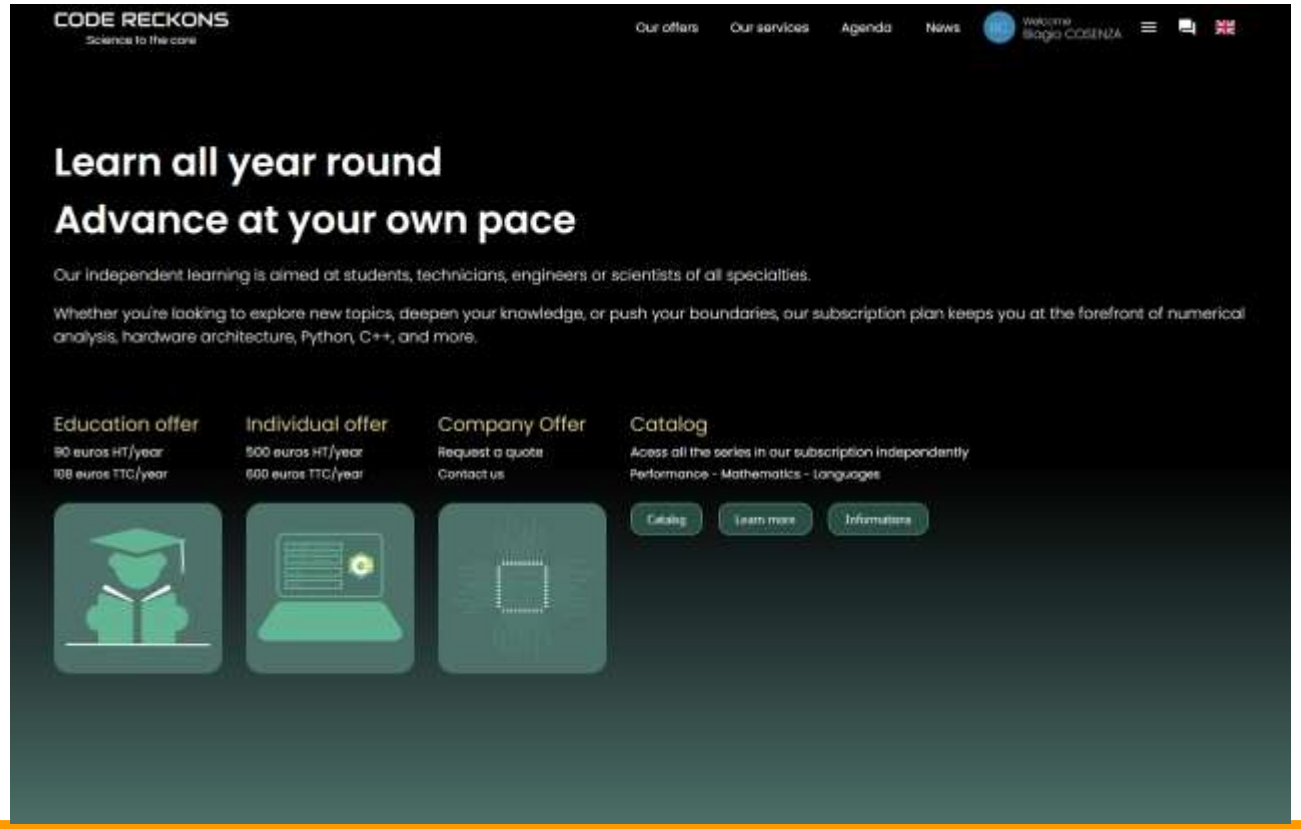
Benchmarking SYCL 2020

- Latest specification SYCL 2020 allow for third-party backends
  - NVIDIA CUDA, AMD ROCm, Intel LevelZero, OpenMP, TBB, etc.
- New SYCL 2020 features
  - Unified Shared Memory (USM)
  - Built-in parallel reduction support
  - Support for native API interoperability
  - Work group and subgroup common algorithm libraries
- Third-party backends + multiple compilers complicate validation
- SYCL-Bench 2020: 9 new benchmark patterns and 44 configurations

Luigi Crisci, Lorenzo Carpentieri, Peter Thoman, Aksel Alpay, Vincent Heuveline, Biagio Cosenza: SYCL-Bench 2020: Benchmarking SYCL 2020 on AMD, Intel, and NVIDIA GPUs. IWOCL 2024: 1:1-1:12

Online teaching phase on CodeReckons

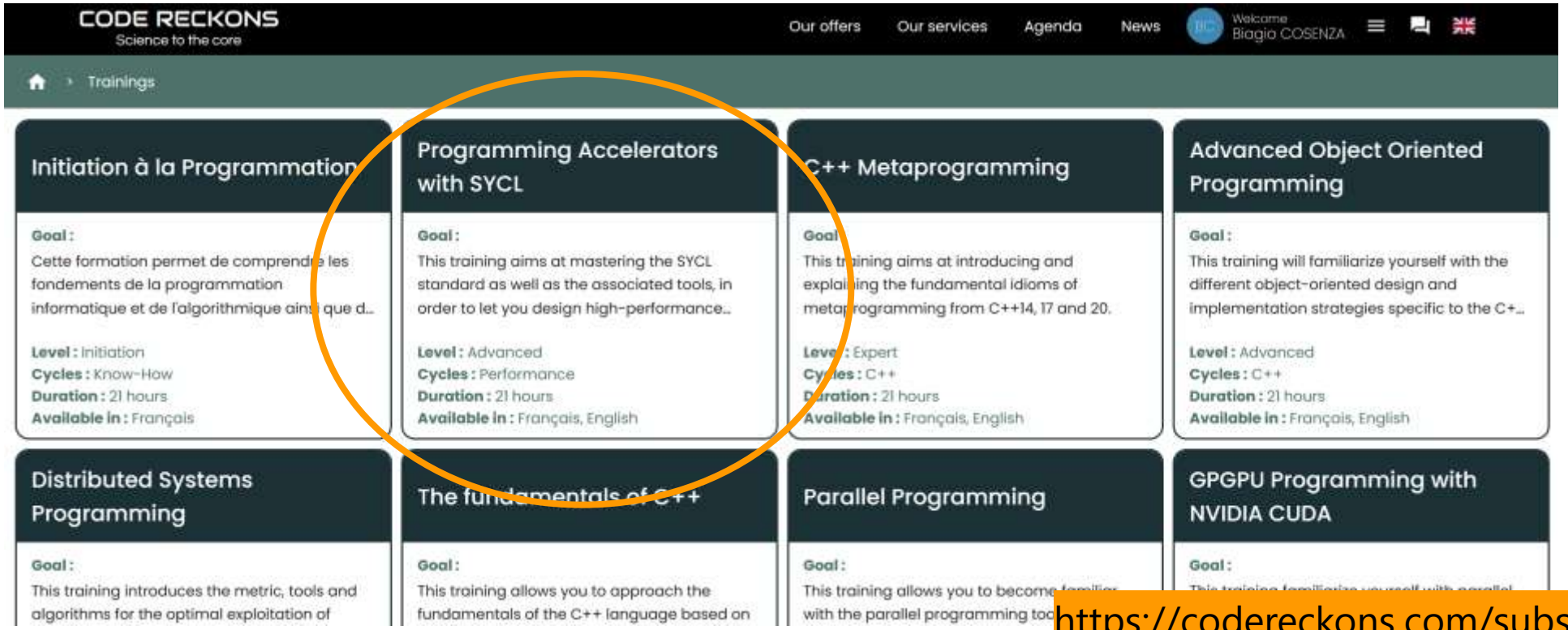
- Next phase: online learning on codereckons.com
  - Webinar participants will receive a free subscription



https://codereckons.com/

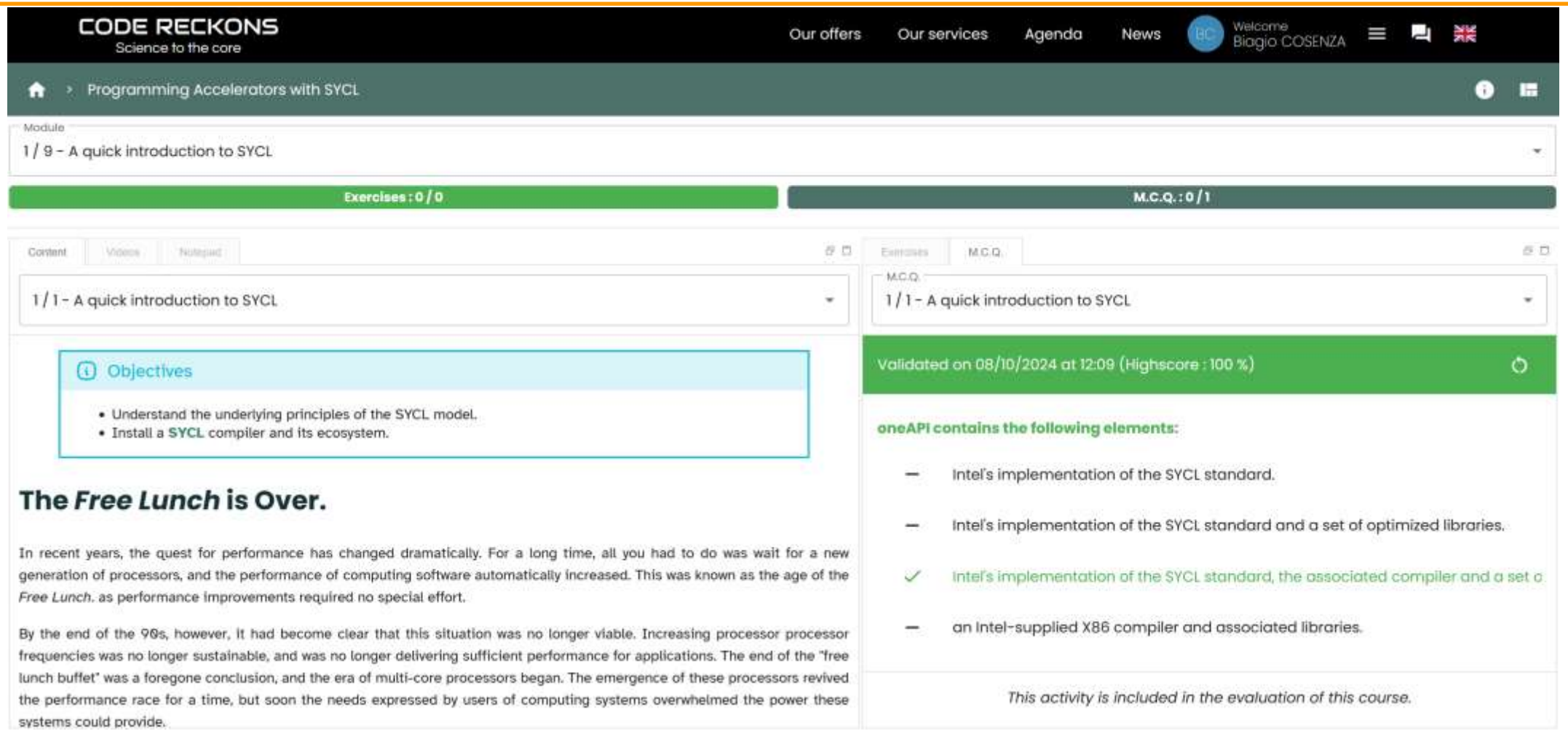
Online teaching phase on CodeReckons

- Programming Accelerators with SYCL
  - 9 chapters, once 6 are completed can access to other courses

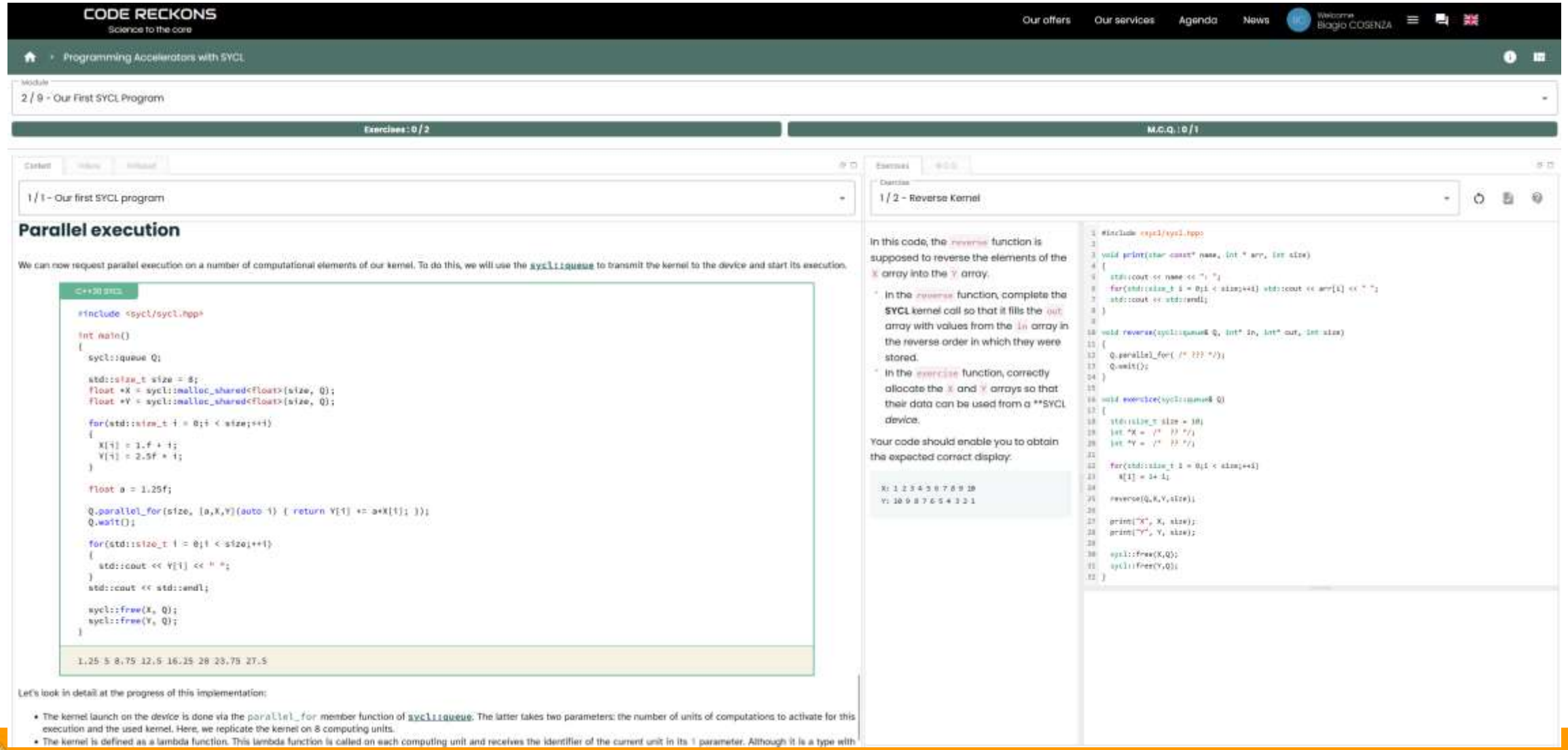


https://codereckons.com/subscriptions

Online teaching phase on CodeReckons: Content & Test



Online teaching phase on CodeReckons: Exercises



Conclusions

- Heterogenous programming with SYCL
  - Basics: buffer/accessor and USM memory model, unorderd queue, …
  - Advanced: group algorithms, atomics, kernel reductions, …
  - SYCL & oneAPI ecosystem
- From CUDA to SYCL
  - Migration tool and hints
- Next:
  - online learning on https://codereckons.com/
  - Hackaton at CINECA