

# Paralelní a distribuované výpočty (B4B36PDV)

**Jakub Mareček**  
jakub.marecek@fel.cvut.cz

Artificial Intelligence Center  
Department of Computer Science  
Faculty of Electrical Engineering  
Czech Technical University in Prague

# What comes next?



# What comes next?

<https://github.com/jmarecek/parallel-cpp/>

The screenshot shows the GitHub interface for the repository 'jmarecek/parallel-cpp'. At the top, there is a search bar and navigation links for Pull requests, Issues, Codespaces, Marketplace, and Explore. Below the repository name, there are tabs for Code, Issues, Pull requests, Actions, Projects, Wiki, Security, Insights, and Settings. The 'Code' tab is selected. The repository is on the 'main' branch, with 1 branch and 0 tags. There are buttons for 'Go to file', 'Add file', and 'Code'. The commit history shows a commit by 'Jakub Marecek and Jakub Marecek' with the message 'minor fixes' and hash 'f7ded9e', made 35 minutes ago. Below the commit, a list of files and folders is shown, including '.devcontainer', '.vscode', 'build', 'chapter\_01', 'code\_examples', 'packages', 'static', '.gitignore', 'LICENSE.md', 'README.md', and 'book.tex'. The 'build' folder is highlighted.

Search or jump to... / Pull requests Issues Codespaces Marketplace Explore

jmarecek / parallel-cpp Public

<> Code Issues Pull requests Actions Projects Wiki Security Insights Settings

main 1 branch 0 tags Go to file Add file <> Code

Jakub Marecek and Jakub Marecek minor fixes f7ded9e 35 minutes ago 44 commits

.devcontainer	devcontainer support	2 days ago
.vscode	devcontainer support	2 days ago
<u>build</u>	initial commit	2 weeks ago
chapter_01	minor fixes	35 minutes ago
code_examples	minor fixes	35 minutes ago
packages	initial commit	2 weeks ago
static	initial commit	2 weeks ago
.gitignore	Update .gitignore	2 weeks ago
LICENSE.md	initial commit	2 weeks ago
README.md	Update README.md	2 days ago
book.tex	more edits	2 days ago

# What is SYCL?

A Specification



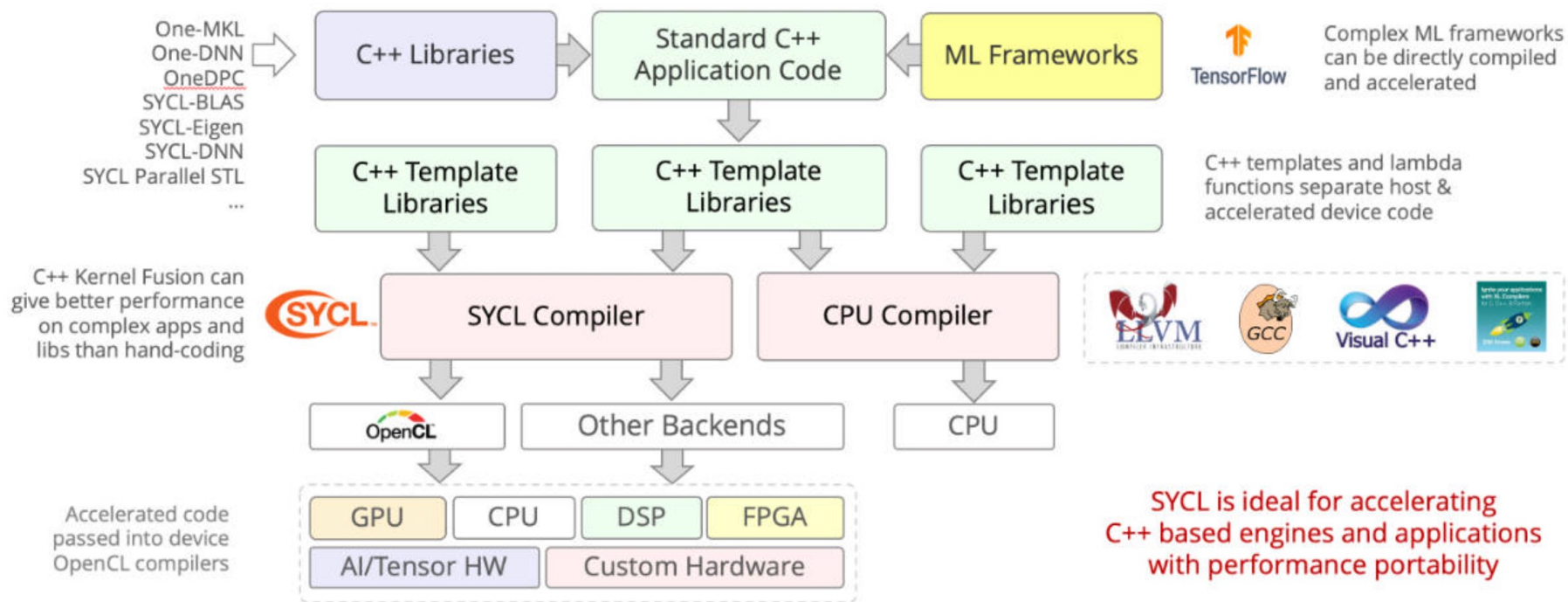
2015: SYCL 1.2 Specification

2022: SYCL 2020 Specification revision 6,

<https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html>

# What is SYCL?

## A Specification



<https://www.khronos.org/files/sycl/sycl-2020-reference-guide.pdf>

# What is SYCL?

## A Specification

- SYCL is a specification for parallel programming of diverse computing devices using standard C++20.
- Cheatsheet: <https://www.khronos.org/files/sycl/sycl-2020-reference-guide.pdf>
- SYCL Academy: <https://github.com/codeplaysoftware/syclacademy>
- Reference: <https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html>

# What is SYCL?

## A Specification

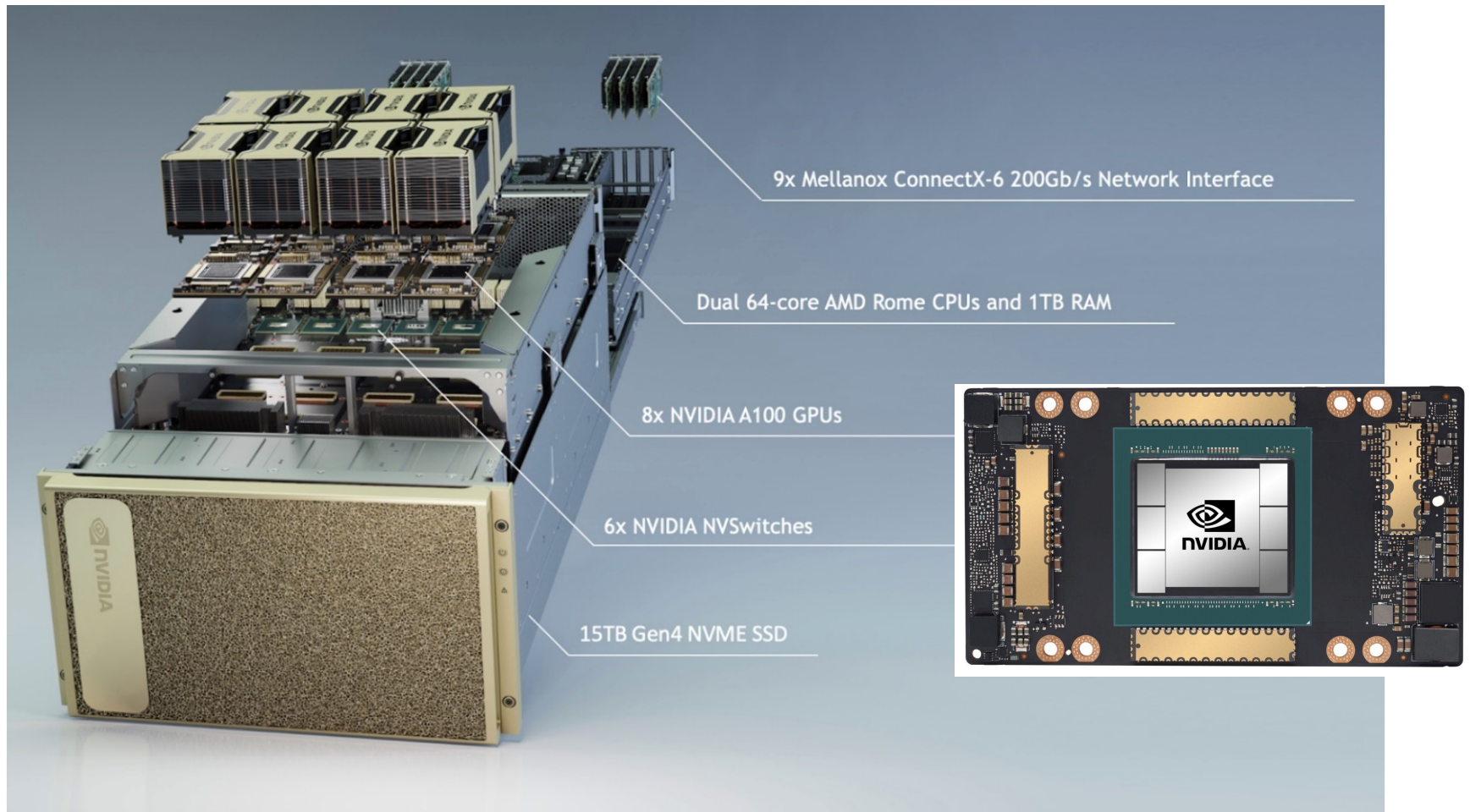
Prime implementations include

- Intel ComputeCpp™ with support for NVIDIA (via OpenCL) and AMD GPUs (via OpenCL) and Arm Mali GPUs (via OpenCL),
- Intel DPC++ and oneAPI with support for NVIDIA (CUDA) and Intel GPUs (oneAPI Level Zero) and Intel Altera FPGAs (OpenCL)
- AdaptiveCpp (formerly known as hipSYCL and OpenSYCL) with support for NVIDIA (CUDA) and AMD ROCm (clang HIP toolchain) and Intel GPUs (oneAPI Level Zero), as well as OpenMP and plain C++. Notably AdaptiveCpp adds GPGPU support for Parallel STL.
- Reference implementation triSYCL with support for AMD Xilinx FPGAs.

See <https://sycl.tech/#get-sycl>

# What is a GPGPU?

A Powerful Piece of Hardware





# What is a GPGPU?

## A Powerful Piece of Hardware

### AMD Threadripper 3990X + NVIDIA GeForce RTX 4090

- Singlethreaded:  
49 GFLOPS (0.05 %)
  - Multithreaded:  
3732 GFLOPS (4%)
  - Multithreaded with GPGPU:  
100 TFLOPS
- @ CZK 200K

### NVIDIA DGX:

- Singlethreaded:  
30+ GFLOPS (cca. 0.00001 %)
  - Multithreaded:  
4 TFLOPS across 128 cores  
(cca. 0.0016 %)
  - Multithreaded with GPGPU:  
2496 TFLOPS across 86016 cores
- @ \$300K

Table 1. NVIDIA A100 Tensor Core GPU Performance Specs

Peak FP64 <sup>1</sup>	9.7 TFLOPS
Peak FP64 Tensor Core <sup>1</sup>	19.5 TFLOPS
Peak FP32 <sup>1</sup>	19.5 TFLOPS
Peak FP16 <sup>1</sup>	78 TFLOPS
Peak BF16 <sup>1</sup>	39 TFLOPS
Peak TF32 Tensor Core <sup>1</sup>	156 TFLOPS   312 TFLOPS <sup>2</sup>
Peak FP16 Tensor Core <sup>1</sup>	312 TFLOPS   624 TFLOPS <sup>2</sup>
Peak BF16 Tensor Core <sup>1</sup>	312 TFLOPS   624 TFLOPS <sup>2</sup>
Peak INT8 Tensor Core <sup>1</sup>	624 TOPS   1,248 TOPS <sup>2</sup>
Peak INT4 Tensor Core <sup>1</sup>	1,248 TOPS   2,496 TOPS <sup>2</sup>

<sup>1</sup> - Peak rates are based on GPU Boost Clock.

<sup>2</sup> - Effective TFLOPS / TOPS using the new Sparsity feature

# What is a GPGPU?

A massively parallel system

Let us briefly consider a particular example of the NVIDIA Ampere architecture of GeForce RTX 3080 or NVIDIA A100:

- There are seven Graphics Processing Clusters (GPCs), sharing up to 40 MB of L2 cache and up to 40 GB of high-speed HBM2 memory
- Within each GPC, there are 12 Streaming Multiprocessors (SMs).



# What is a GPGPU?

A massively parallel system

Let us briefly consider a particular example of the NVIDIA Ampere architecture:

- Within each SM, there are 128 cores working with single-precision floating-point (FP32) precision and two double-precision (FP64) units. There is also 128 KB of L1/Shared Memory, shared across the 128 cores.
- Each SM is partitioned into four processing blocks (or partitions), each with a few kilobytes of L0 instruction cache and one warp scheduler.



# What is a GPGPU?

A massively parallel system *with plenty of restrictions*

Altogether, A100 has 10752 cores, but their use is rather constrained.

Threads should be running in warps of at least 32 for best performance. Each warp can have at most 32 threads and runs them in lock-step on one processing blocks.

Each streaming multiprocessors can run at most 64 warps of 32 threadblocks (i.e., 2048 threads per SM).

Further constraints are due to the register use: each thread can use at most 255 registers, but there are only 65536 32-bit registers for the SM (yielding a limit of 257 threads per SM, at full register utilization).

Further constraints are due to the use of memory hierarchy (esp. the 128 KB of shared memory, shared across the 128 cores).

# What is a GPGPU?

*A massively parallel system with plenty of restrictions*

Similar to the CPU, the GPU hence has a memory hierarchy:

- L1 cache with 33 cycle latency and shared memory with even lower latency, based on microbenchmarking
- L2 cache with up to 2080 GB/s read bandwidth (200 cycle latency), based on microbenchmarking
- on-board HBM2 memory with 1555 GB/sec bandwidth (290 cycle latency)
- intra-board NVLink with 50 Gb/sec per signal pair bandwidth
- access to RAM via PCI Express Gen 4 (PCIe Gen 4) at 31.5 GB/sec
- optionally, intra-node communication at 200 Gbit/sec using InfiniBand.

The interaction of the GPGPU memory hierarchy and CPU memory hierarchy is non-trivial, but summarized by the suggestion to reduce the number and volume of transfers between the host and the GPGPU, even at the price of increasing the volume of computation substantially. (Compare the numbers above to M.2 PCIe Gen4 SSDs with 7 GB/sec bandwidth.)

# Options for Programming GPGPUs

1991: OpenGL

1996: Direct3D

2006: OpenGL passes to Khronos

2007: NVIDIA CUDA (Compute Unified Device Architecture)

2009: OpenCL

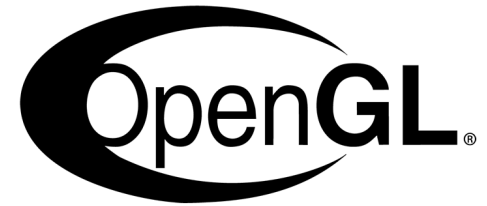
2016: Vulkan replaces OpenGL

2017: OpenCL replaces OpenGL ES

2020: OpenCL 3.0 replaces OpenCL C++ Kernel Language, with C++ for OpenCL.

2021: Nvidia Ampere and Maxwell supports OpenCL 3.0

2022: Arm Mali-G615 supports OpenCL 3.0



# Options for Programming NVIDIA GPGPUs

## NVIDIA CUDA C++

We have seen OpenMP offloading:

```
4 void saxpy(float a, float* x, float* y, int sz, int
  ↳ num_blocks) {
5 #pragma omp target teams distribute parallel for simd \
6   num_teams(num_blocks) map(to:x[0:sz]) map(tofrom:y[0:sz])
7   for (int i = 0; i < sz; i++) {
8     y[i] = a * x[i] + y[i];
9   }
10 }
```

$$\begin{array}{r} a \cdot \begin{array}{|c|} \hline x_1 \\ \hline \end{array} + \begin{array}{|c|} \hline y_1 \\ \hline \end{array} = \begin{array}{|c|} \hline a \cdot x_1 + y_1 \\ \hline \end{array} \\ a \cdot \begin{array}{|c|} \hline x_2 \\ \hline \end{array} + \begin{array}{|c|} \hline y_2 \\ \hline \end{array} = \begin{array}{|c|} \hline a \cdot x_2 + y_2 \\ \hline \end{array} \\ \vdots \\ a \cdot \begin{array}{|c|} \hline x_d \\ \hline \end{array} + \begin{array}{|c|} \hline y_d \\ \hline \end{array} = \begin{array}{|c|} \hline a \cdot x_d + y_d \\ \hline \end{array} \end{array}$$

# Options for Programming NVIDIA GPGPUs

## NVIDIA CUDA C++

```
1 #include <vector>
2 #include <iostream>
3 #include <cuda.h>
4 #define N 1048576
5
6 __global__ void saxpy_kernel(float a, float* x, float* y,
   ↪ float* z){
7     int i = blockIdx.x*blockDim.x + threadIdx.x;
8     if (i < n) y[i] = a*x[i] + y[i];
9 }
10
11 int main(){
12     std::vector<float> vx(N);
13     float* x = vx.data();
14     std::vector<float> vy(N);
15     float* y = vy.data();
16     std::vector<float> vz(N);
17     float* z = vz.data();
18     float *dx, *dy, *dz;
19     cudaMalloc(&dx, N*sizeof(float));
20     cudaMalloc(&dy, N*sizeof(float));
21     cudaMalloc(&dz, N*sizeof(float));
22     cudaMemcpy(dx, x, N*sizeof(float), cudaMemcpyHostToDevice);
23     cudaMemcpy(dy, y, N*sizeof(float), cudaMemcpyHostToDevice);
24     int nblocks = (n + 255) / 256;
25     saxpy_kernel<<<nblocks, 256>>>(3.1415, dx, dy, dz);
26     cudaMemcpy(z, dz, N*sizeof(float), cudaMemcpyDeviceToHost);
27     cudaFree(dx);
28     cudaFree(dy);
29     cudaFree(dz);
30     // we do not need free(x), free(y), free(z)
31 }
```



# Options for Programming NVIDIA GPGPUs

## NVIDIA Thrust: The C++ Parallel Algorithms Library

```
3 #include <thrust/device_vector.h>
4 #include <thrust/transform.h>
5 #include <thrust/copy.h>
6 #include <thrust/fill.h>
7 #include <thrust/functional.h>
8 #include <iostream>
9
10 struct saxpy_functor {
11     const float a;
12     saxpy_functor(float _a) : a(_a) {}
13     __host__ __device__
14     float operator()(const float& x, const float& y) const {
15         return a * x + y;
16     }
17 };
18
19 int main(){
20     thrust::device_vector<float> dx(1048576);
21     thrust::fill(dx.begin(), dx.end(), 1.0);
22     thrust::device_vector<float> dy(1048576);
23     thrust::fill(dx.begin(), dx.end(), 2.0);
24     // Y <- A * X + Y
25     // thrust::transform(dx.begin(), dx.end(), dy.begin(),
26     ↪ dy.begin(), 3.1415f * _1 + _2);
27     thrust::transform(dx.begin(), dx.end(), dy.begin(),
28     ↪ dy.begin(), saxpy_functor(3.1415));
29     thrust::copy(dy.begin(), dy.end(),
30     ↪ std::ostream_iterator<float>(std::cout, "\n"));
31 }
```

<https://github.com/NVIDIA/thrust>

# Options for Programming NVIDIA GPGPUs

## SYCL

The Thrust code is beautiful, but we it is not easily portable to another platform (AMD, Intel, ARM Mali). There may be several good reasons not to be tied specifically to NVIDIA:

- In consumer PC market, the biggest vendor of PC GPU is Intel with 71 percent of the market by unit count as of Q4 2022. NVIDIA has a market share of 17 percent and AMD 12 percent.
- In discrete PC GPU market, NVIDIA occupies 80-90% of the market, according to various estimates. Note that this market is rapidly declining.
- In mobile GPGPUs, ARM Mali is estimated to have close to 40% market share.
- In gaming consoles, both Xbox and PlayStation use accelerators by AMD.
- In high-performance computing, NVIDIA is popular, but the top 1 system (Frontier) currently uses AMD Instinct™ 250X accelerators, and number of other systems use ARM architectures.

# Options for Programming NVIDIA GPGPUs

## SYCL

```
3 #include <iostream>
4 #include "CL/sycl.hpp"
5
6 class saxpy3;
7
8 int main(int argc, char * argv[]) {
9     std::vector<float> vx(1048576, 1.0);
10    std::vector<float> vy(1048576, 2.01);
11    std::vector<float> vz(1048576, 0.0);
12    sycl::queue q(sycl::default_selector{});
13    try {
14        const float A(aval);
15        sycl::buffer<float,1> dx { vx.data(),
16        ↪ sycl::range<1>(vx.size()) };
17        sycl::buffer<float,1> dy { vy.data(),
18        ↪ sycl::range<1>(vy.size()) };
19        sycl::buffer<float,1> dz { vz.data(),
20        ↪ sycl::range<1>(vz.size()) };
21
22        q.submit([&](sycl::handler& h) {
23            sycl::accessor x(dx, h, sycl::read_only);
24            sycl::accessor y(dy, h, sycl::read_only);
25            sycl::accessor z(dz, h, sycl::read_write);
26            h.parallel_for<class saxpy3>( sycl::range<1>{length},
27            ↪ [=] (sycl::id<1> it) {
28                const size_t i = it[0];
29                z[I] += 3.1415 * x[i] + y[I];
30            });
31        });
32        q.wait();
33    }
34    catch (sycl::exception & e) {
35        std::cout << e.what() << std::endl;
36        return 1;
37    }
38    return 0;
39 }
```

# What comes next?

- Structuring computation
  - Device selector
  - Queue
  - Work items, Work groups, and Kernels
- Synchronization primitives
  - Unified shared memory
  - Buffers and accessor
  - Barriers

Asynchronous errors

# Structuring computation

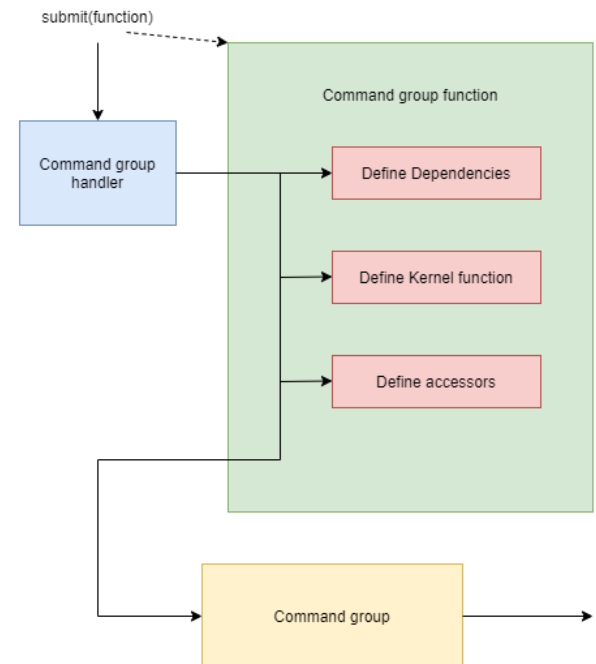
## Selectors

- Selectors are used to pick device to run on.
- In header `<device_selector.h>`, there is an abstract class **device\_selector** with numerous implementations such as **gpu\_selector**, **host\_selector**, **opengl\_selector**, and **default\_selector**.
- One can also implement its own subclasses that specify to the runtime how to perform device selection.
- For example, it may query the amount of memory on the GPGPU and if it is sufficient, use GPGPU. If it were not sufficient, it could use CPU as a fallback.
- In a **device\_selector**, one overrides  
`int operator()(const sycl::device& dev) const` override  
and returns an integer for the priority. The higher integer, the higher priority.

# Structuring computation

## Queues

- A queue is a SYCL construct through which we orchestrate work on the device.
- In a constructor of a queue, we pass the device, which cannot be changed later, but one can create further queues for the same device.
- A key method is **queue.submit**.
- In one memory model, a command group is a callable (a named type, a lambda function, or `std::function`), which receives a "command group handler" from the SYCL as an argument of **operator()** so as to access the API.
- A command group objects may also combine the callable and a set of requirements (edges of a task-graph) and accessors. Based on the accessors, the task-graph can be constructed automatically.



# Structuring computation

## Queues

- Asynchronous execution also means an undefined order of execution,
- unless we use **wait** or suggest the dependencies between the "actions" in the form of a task graph.
- We can also declare the queue to be in-order, similar to sorted in OpenMP:  
**queue q{property::queue::in\_order()};**

# Structuring computation

## Work Items, Work Groups, Kernels

- Within an action submitted to the queue, we execute kernels.
- Kernels are callables
  - receiving an index to the run of the kernel as **auto idx** or **id<1> idx** or similar.
  - returning nothing; with **void** return type
  - which cannot allocate memory dynamically
  - which cannot use certain other features (e.g., RTTI).
- Within the **single\_task function** method of the "command group handler" API, we pass a C++ function object as a parameter and have it executed once.
- Kernel can also be a class that overloads operator `void operator()(id<1> idx)`.
- Most often, we want the kernel executed many times, in a data-parallel fashion.
- In the so-called **nd-ranges** ("kernel grid" on NVIDIA), we partition the index-set of data hierarchically first into global ranges, and then into local ranges.
- The local range corresponds to a **work-group** and each element corresponds to a **work item** (= single run of a kernel).



# Structuring computation

## Work Items, Work Groups, Kernels

- The work-group local memory can often be accessed very efficiently, via **local\_accessor**, and can be used to coordinate multiple work items (= single runs) within a work group.
- The threads of one work group ('thread block' on NVIDIA) are sent to one Streaming Multiprocessors (SM), but one SM can execute threads by multiple work groups in its multiple processing blocks. At most one work group per processing block.
- Recalling the memory hierarchies of GPGPUs, each work item can access:
  - private memory
  - work-group local memory
  - global memory accessible to all work items within an nd-range, but whose access can be very expensive, as it involves copying data across PCIe bus
  - constant memory, which is a part of the global memory, but which can be very cheap to access.

# Memory Access

At the cost of some latency, one can use a unified shared memory across both the host and the device, wherein one uses the same pointer on both the host and the device. This requires:

- `template <typename T> T* malloc_shared(size_t count, const queue& q, const property_list &propList = {})`
- `void free(void* ptr, sycl::queue& syclQueue)`

Alternatively, one can use buffers and accessors for complete control.

```
1 auto shared = malloc_shared<double>(42, q);
2
3 q.submit([&](handler& cgh){
4     cgh.parallel_for(range{42}, [=](id<1> tid){
5         shared[i] = 0.0;
6     })
7 });
```

```
1 buffer<double> A{range{42}};
2
3 q.submit([&](handler& cgh){
4     accessor aA{A, cgh};
5     cgh.parallel_for(range{42}, [=](id<1> & idx){
6         aA[idx] = 0.0;
7     })
8 });
```

# Memory Access

- A **buffer** is a constrained view of a 1-, 2-, or 3-dimensional array.
- The constraints specify how it can be accessed on the host, the device or both.
- A buffer is constructed with a pre-allocated, trivially copyable C++ objects (e.g., STL container).
- Within the contract for the use of the buffer, one promises not to amend the memory used to initialise the buffer during the lifetime of the buffer.
- Buffer promises to update the memory in the host upon destruction, in RAII spirit.

```
1 auto shared = malloc_shared<double>(42, q);
2
3 q.submit([&](handler& cgh){
4     cgh.parallel_for(range{42}, [=](id<1> tid){
5         shared[i] = 0.0;
6     })
7 });
```

```
1 buffer<double> A{range{42}};
2
3 q.submit([&](handler& cgh){
4     accessor aA{A, cgh};
5     cgh.parallel_for(range{42}, [=](id<1> & idx){
6         aA[idx] = 0.0;
7     })
8 });
```

# Memory Access

In the case of one-dimensional arrays, one can call the constructor with an iterator:

```
template <typename InputIterator> buffer(InputIterator first,  
InputIterator last, const property_list &propList={});
```

Once in a kernel, an **accessor** specifies constraints on the use of a buffer therein. The key choices are:

- access mode: **read**, **write**, and **read\_write**, where write access mode also implicitly defines dependencies between tasks
- access target: **global\_memory** suggests that the data resides in the global memory space of the device.
- **no\_init** suggests that the initial data can be discarded (not moved to the device).

```
1  buffer<double> A{range{42}};  
2  
3  q.submit([&](handler& cgh){  
4      accessor aA{A, cgh};  
5      cgh.parallel_for(range{42}, [=](id<1> & idx){  
6          aA[idx] = 0.0;  
7      })  
8  });
```

# Memory Access

```
1 #include <iostream>
2 #include <vector>
3 #include <CL/sycl.hpp>
4
5 using namespace std;
6 using namespace cl::sycl;
7 class buffer3;
8
9 int main(int, char**) {
10
11     std::vector<float> a { 2.0, 3.0, 7.0, 4.0 };
12     std::vector<float> b { 4.0, 6.0, 1.0, 3.0 };
13     std::vector<float> c { 0.0, 0.0, 0.0, 0.0 };
14
15     default_selector device_selector;
16
17     queue q(device_selector);
18
19     std::cout << "Running on "
20               <<
21               ↪ queue.get_device().get_info<info::device::name>()
22               << "\n";
23     {
24         buffer bufA(a);
25         buffer bufB(b);
26         buffer bufC(c);
27
28         q.submit([&] (handler& cgh) {
29
30             auto accA = accessor(bufA, cgh, read_only);
31             auto accB = accessor(bufB, cgh, read_only);
32             auto accC = accessor(bufC, cgh, write_only);
33             cgh.single_task<buffer3>(bufC.get_range(), [=](id<1>
34             ↪ i) {
35                 accC[i] = accA[i] + accB[i];
36             });
37         });
38         q.wait_and_throw();
39     }
40     return 0;
41 }
```

# Error Handling

The SYCL implementation may throw "synchronous errors" (one at a time).

In contrast, asynchronous errors are produced by a command group or a kernel (with many kernels running at any point). By default, asynchronous errors are not propagated to the host.

One can, however, define an error handler and pass it to a queue **queue q(default\_selector{}, exception\_handler);**

The error handler receives an **exception\_list**, wherein one can iterate over **std::exception\_ptr**.

See <https://www.codingame.com/playgrounds/48226/introduction-to-sycl/error-handling> for a great tutorial with code that is editable, compilable, and runnable online. Let us simplify their main example in the next slide.

# Error Handling

```
4 #include <iostream>
5 #include <sycl/sycl.hpp>
6
7 using namespace sycl;
8
9 class exception1;
10
11 int main(int, char**) {
12     auto exception_handler = [] (exception_list exceptions) {
13         for (std::exception_ptr const& e : exceptions) {
14             try {
15                 std::rethrow_exception(e);
16             } catch(exception const& e) {
17                 std::cout << "Caught asynchronous SYCL exception:\n"
18                     << e.what() << std::endl;
19             }
20         }
21     };
22
23     queue q(default_selector{}, exception_handler);
24     // actual use of the q
25
26     try {
27         q.wait_and_throw();
28     } catch (exception const& e) {
29         std::cout << "Caught synchronous SYCL exception:\n"
30             << e.what() << std::endl;
31     }
32     return 0;
33 }
```

# Building with SYCL

Linking SYCL against OpenCL.

---

```
1 set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -O3 -fsycl
   ↪ -std=c++17")
2 set(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -lOpenCL
   ↪ -lsycl")
```

Linking SYCL against MKL.

---

```
1 set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -O3 -fsycl
   ↪ -std=c++17")
2 set(CMAKE_EXE_LINKER_FLAGS " -fsycl -lmkl_sycl
   ↪ -lmkl_intel_ilp64 -lmkl_sequential -lmkl_core")
```

Linking SYCL against AMD.

---

```
1 -fsycl -fsycl-targets=amdgc-n-amd-amdhsa -Xsycl-target-backend
   ↪ {offload-arch=gfx906
```



# What have we seen?

- Structuring computation
  - Device selector
  - Queue
  - Work items, Work groups, and Kernels
- Synchronization primitives
  - Unified shared memory
  - Buffers and accessor
  - Barriers

Asynchronous errors

