Paralelní a distribuované výpočty (B4B36PDV)

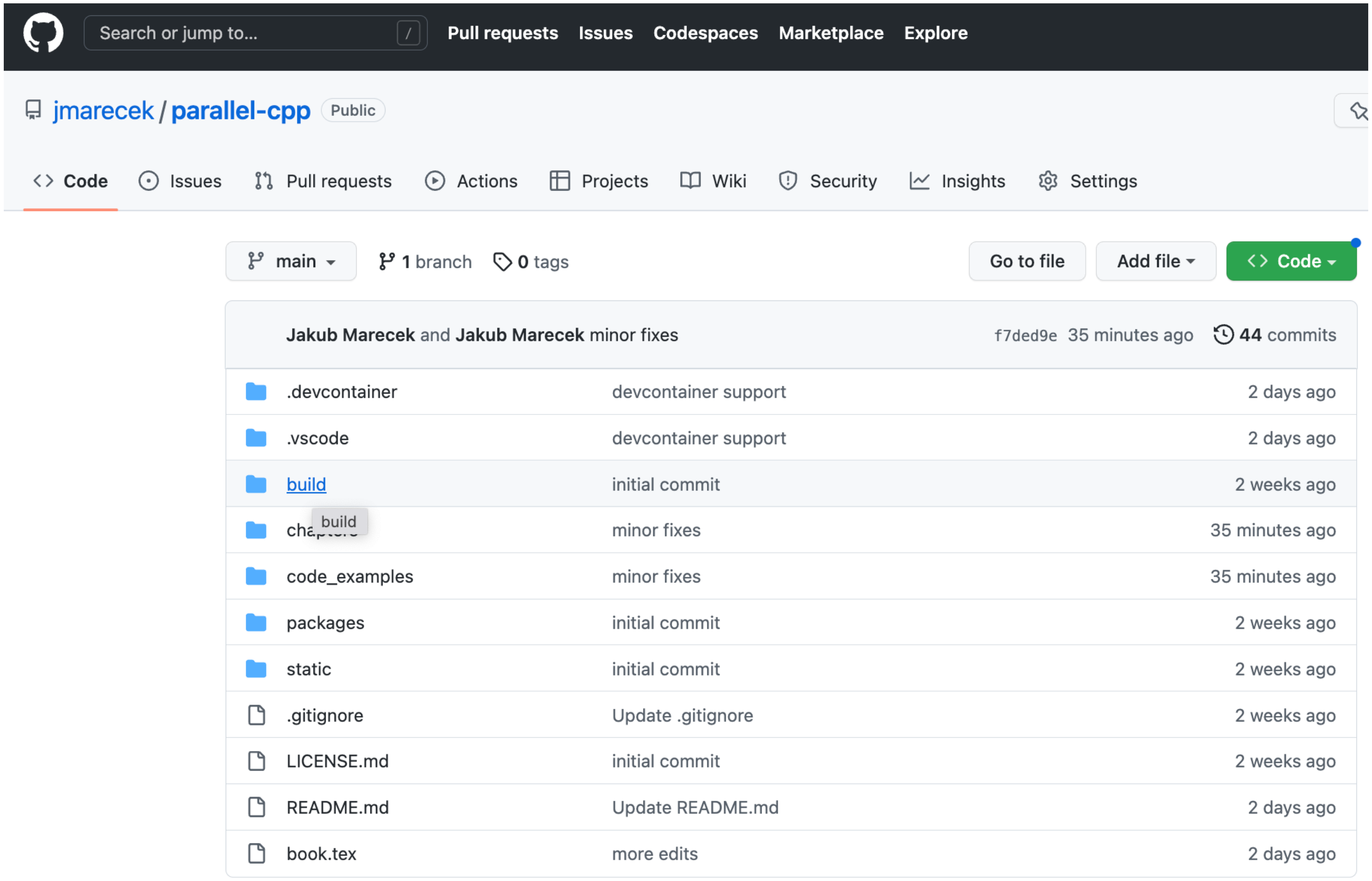
Jakub Mareček
jakub.marecek@fel.cvut.cz

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/blob/main/static/2023March15.pdf



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
build	initial commit	2 weeks ago
chapters	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?

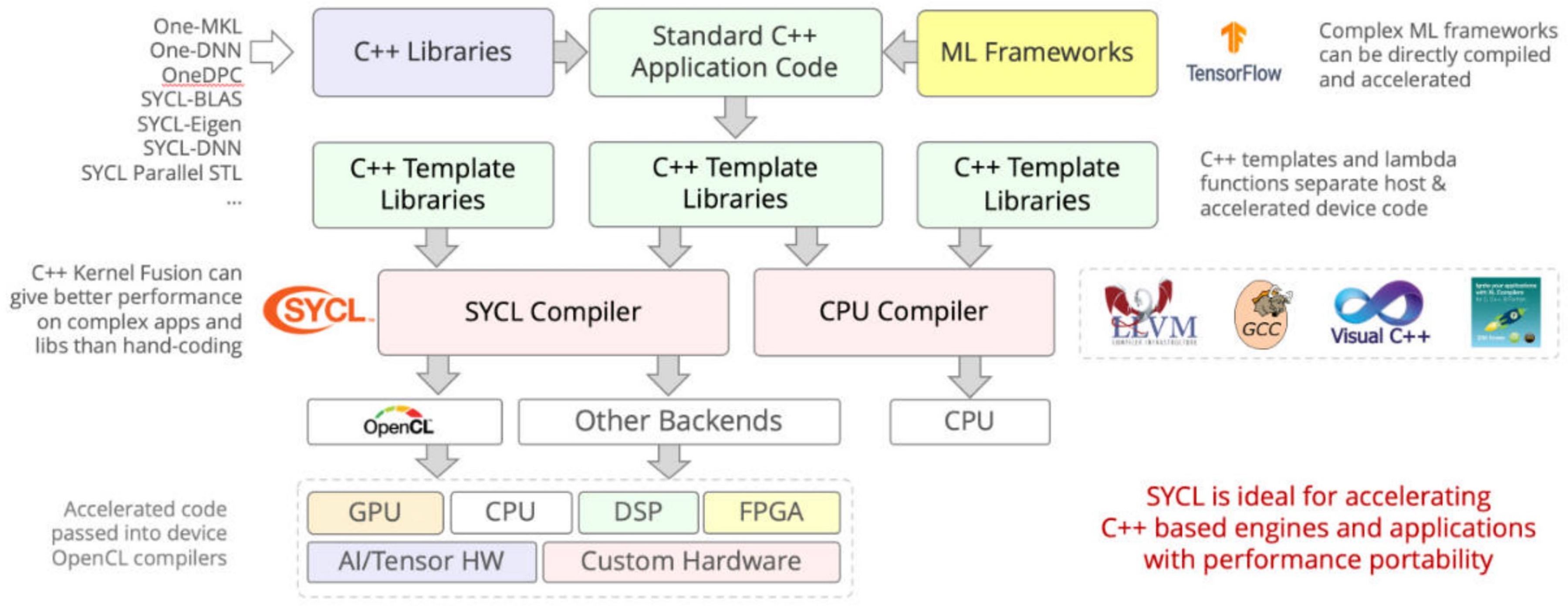
A Specification

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


One-MKL
One-DNN
OneDPC
SYCL-BLAS
SYCL-Eigen
SYCL-DNN
SYCL Parallel STL
...
C++ Libraries
Standard C++ Application Code
ML Frameworks
TensorFlow
Complex ML frameworks can be directly compiled and accelerated
C++ Template Libraries
C++ Template Libraries
C++ Template Libraries
C++ templates and lambda functions separate host & accelerated device code
C++ Kernel Fusion can give better performance on complex apps and libs than hand-coding
SYCL
SYCL Compiler
CPU Compiler
LLVM
GCC
Visual C++
OpenCL
Other Backends
CPU
Accelerated code passed into device OpenCL compilers
GPU
CPU
DSP
FPGA
AI/Tensor HW
Custom Hardware
SYCL is ideal for accelerating C++ based engines and applications with performance portability

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)
- Open SYCL (formerly known as hipSYCL) with support for NVIDIA (CUDA) and AMD ROCm (clang HIP toolchain) and Intel GPUs (oneAPI Level Zero), as well as OpenMP and plain C++.
- Reference implementation triSYCL with support for AMD Xilinx FPGAs.

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

What is a GPGPU?

A Powerful Piece of Hardware

AMD Threadripper 3990X + NVIDIA GeForce RTX 4090 Ti

- 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

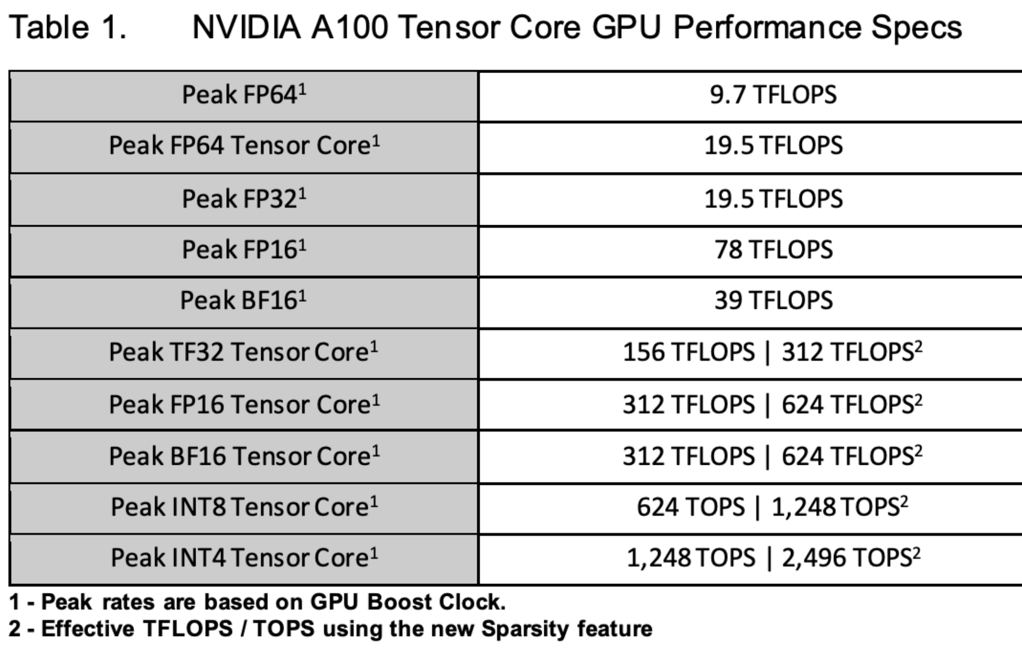


Table 1. NVIDIA A100 Tensor Core GPU Performance Specs

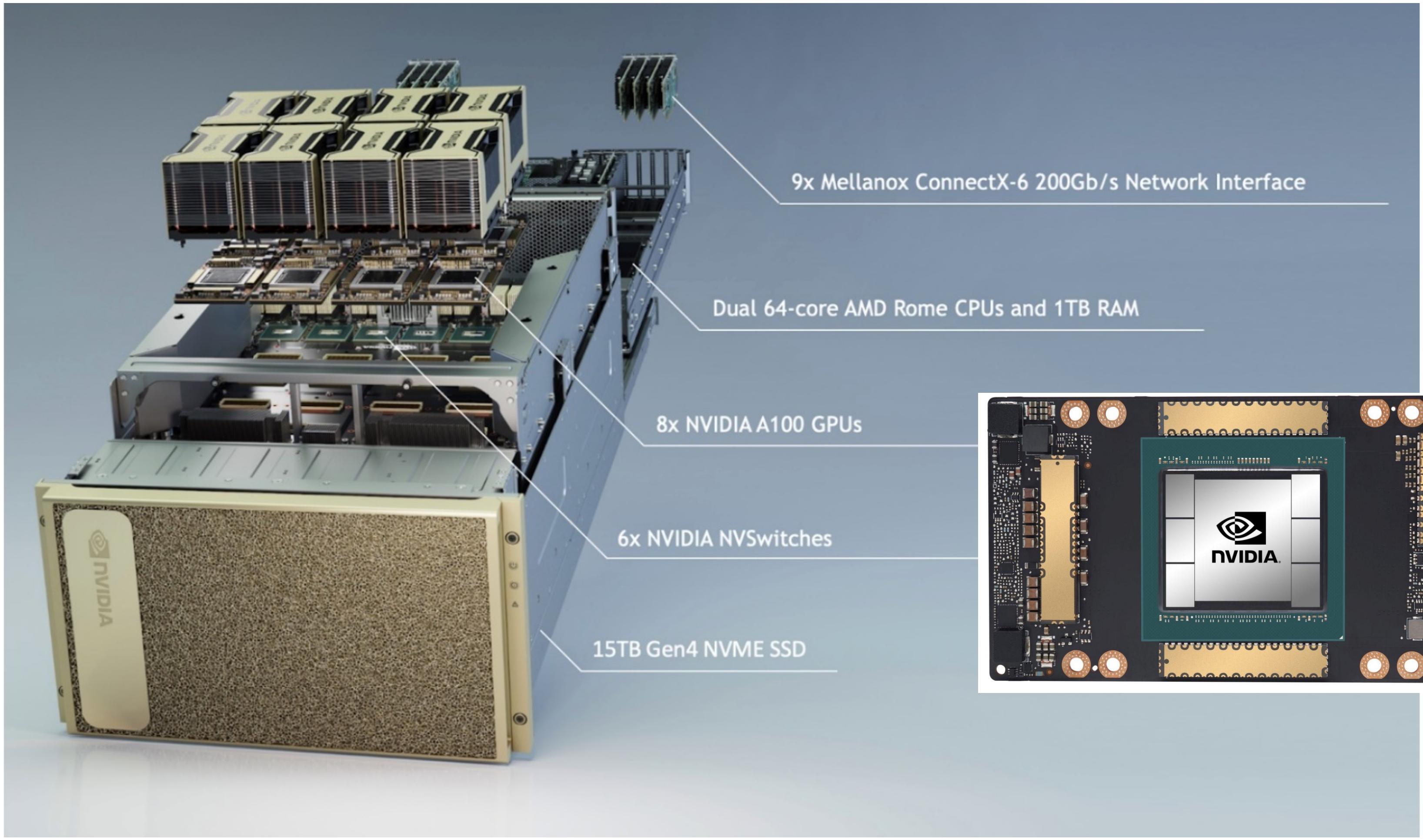
Peak FP64[1]	9.7 TFLOPS
Peak FP64 Tensor Core[1]	19.5 TFLOPS
Peak FP32[1]	19.5 TFLOPS
Peak FP16[1]	78 TFLOPS
Peak BF16[1]	39 TFLOPS
Peak TF32 Tensor Core[1]	156 TFLOPS | 312 TFLOPS[2]
Peak FP16 Tensor Core[1]	312 TFLOPS | 624 TFLOPS[2]
Peak BF16 Tensor Core[1]	312 TFLOPS | 624 TFLOPS[2]
Peak INT8 Tensor Core[1]	624 TOPS | 1,248 TOPS[2]
Peak INT4 Tensor Core[1]	1,248 TOPS | 2,496 TOPS[2]

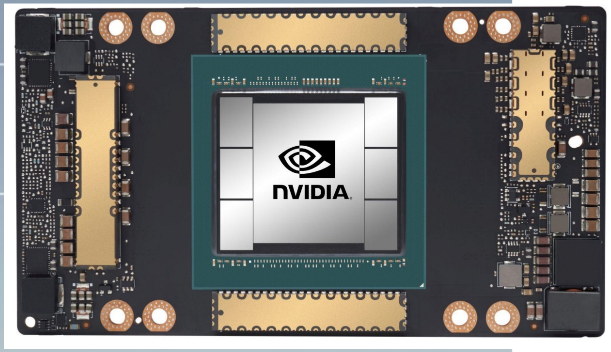
1 - Peak rates are based on GPU Boost Clock.
2 - Effective TFLOPS / TOPS using the new Sparsity feature

https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

What is a GPGPU?

A Powerful Piece of Hardware


9x Mellanox ConnectX-6 200Gb/s Network Interface
Dual 64-core AMD Rome CPUs and 1TB RAM
8x NVIDIA A100 GPUs
6x NVIDIA NVSwitches
15TB Gen4 NVME SSD



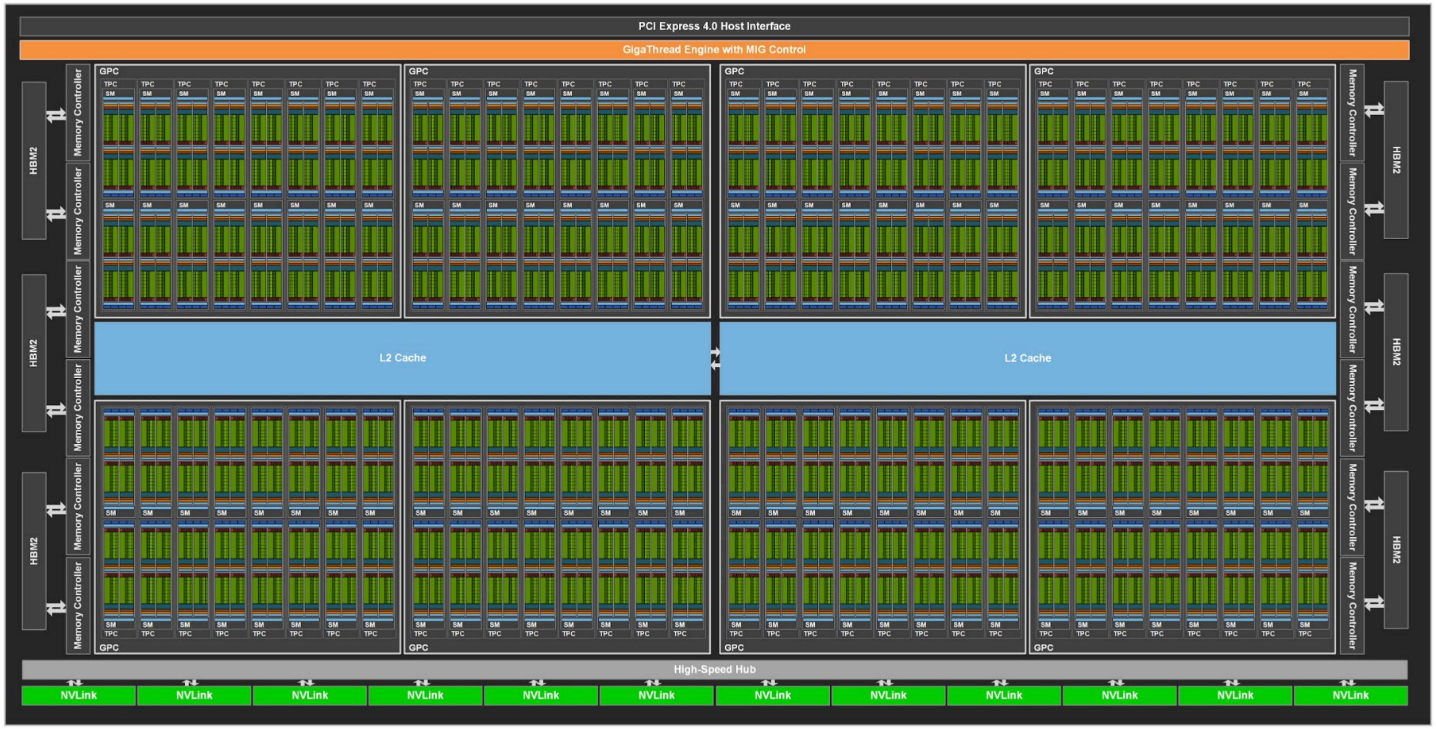
https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

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).


PCI Express 4.0 Host Interface
GigaThread Engine with MIG Control
GPC
TPC
SM
Memory Controller
HBM2
L2 Cache
High-Speed Hub
NVLink

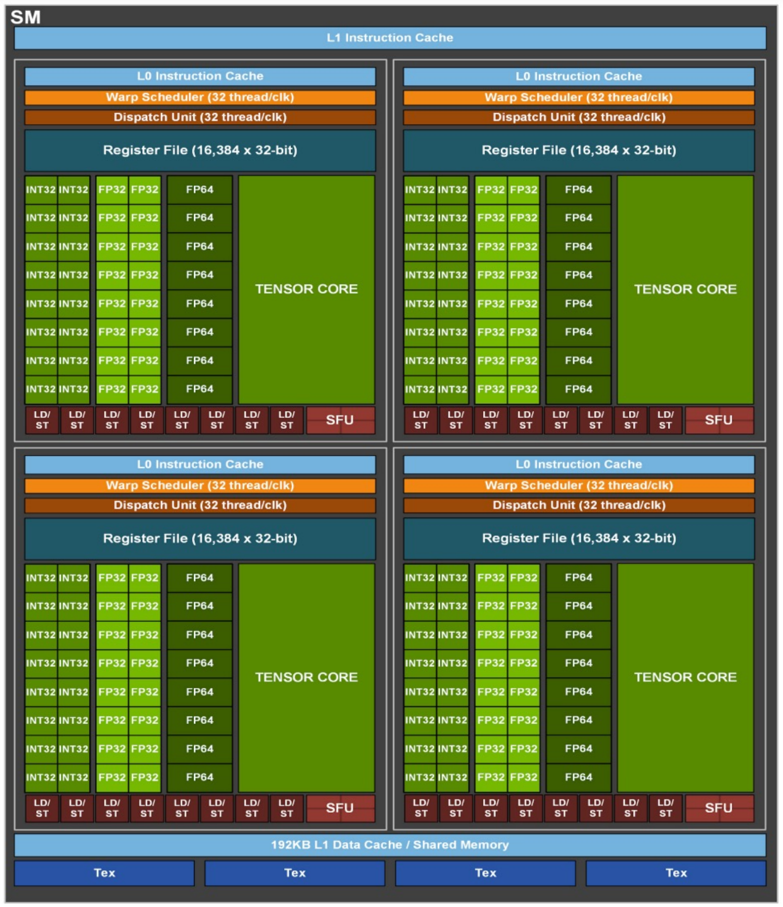
https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

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.



SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE
LD/ST
SFU
192KB L1 Data Cache / Shared Memory
Tex

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

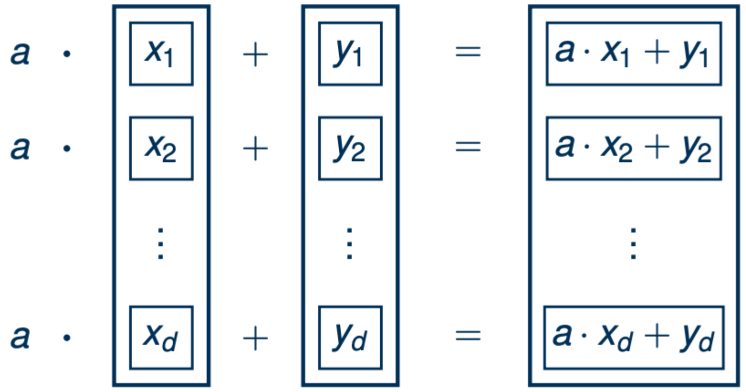
1991: OpenGL
1996: Direct3D
2006: OpenGL passes to Khronos
2007: NVIDIA CUDA (Compute Unified Device Architecture)
2009: OpenCL
2016: Vulkan replaces OpenGL
2017: OpenCL replaces OpenGL ES
2020: OpenCL 3.0 replaces OpenCL C++ Kernel Language, with C++ for OpenCL.
2021: Nvidia Ampere and Maxwell supports OpenCL 3.0
2022: Arm Mali-G615 supports OpenCL 3.0

Options for Programming NVIDIA GPGPUs

NVIDIA CUDA C++

We have seen OpenMP offloading:

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



$a \cdot x_1 + y_1 = a \cdot x_1 + y_1$
$a \cdot x_2 + y_2 = a \cdot x_2 + y_2$
$\vdots$
$a \cdot x_d + y_d = a \cdot x_d + y_d$

https://hpc-wiki.info/mediawiki/hpc_images/a/ab/GPU_tutorial_saxpy_cuda_c.pdf

Options for Programming NVIDIA GPGPUs

NVIDIA CUDA C++

```cpp
#include <vector>
#include <iostream>
#include <cuda.h>
#define N 1048576

__global__ void saxpy_kernel(float a, float* x, float* y,
↪  float* z){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

int main(){
  std::vector<float> vx(N);
  float* x = vx.data();
  std::vector<float> vy(N);
  float* y = vy.data();
  std::vector<float> vz(N);
  float* z = vz.data();
  float *dx, *dy, *dz;
  cudaMalloc(&dx, N*sizeof(float));
  cudaMalloc(&dy, N*sizeof(float));
  cudaMalloc(&dz, N*sizeof(float));
  cudaMemcpy(dx, x, N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(dy, y, N*sizeof(float), cudaMemcpyHostToDevice);
  int nblocks = (n + 255) / 256;
  saxpy_kernel<<<nblocks, 256>>>(3.1415, dx, dy, dz);
  cudaMemcpy(z, dz, N*sizeof(float), cudaMemcpyDeviceToHost);
  cudaFree(dx);
  cudaFree(dy);
  cudaFree(dz);
  // we do not need free(x), free(y), free(z)
}
```

Options for Programming NVIDIA GPGPUs

NVIDIA Thrust: The C++ Parallel Algorithms Library

https://github.com/NVIDIA/thrust

```cpp
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/copy.h>
#include <thrust/fill.h>
#include <thrust/functional.h>
#include <iostream>

struct saxpy_functor {
  const float a;
  saxpy_functor(float _a) : a(_a) {}
  __host__ __device__
  float operator()(const float& x, const float& y) const {
    return a * x + y;
  }
};

int main(){
  thrust::device_vector<float> dx(1048576);
  thrust::fill(dx.begin(), dx.end(), 1.0);
  thrust::device_vector<float> dy(1048576);
  thrust::fill(dx.begin(), dx.end(), 2.0);
  // Y <- A * X + Y
  // thrust::transform(dx.begin(), dx.end(), dy.begin(),
  ↪  dy.begin(), 3.1415f * _1 + _2);
  thrust::transform(dx.begin(), dx.end(), dy.begin(),
  ↪  dy.begin(), saxpy_functor(3.1415));
  thrust::copy(dy.begin(), dy.end(),
  ↪  std::ostream_iterator<float>(std::cout, "\n"));
}
```

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 dominant, 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

```
#include <iostream>
#include "CL/sycl.hpp"

class saxpy3;

int main(int argc, char * argv[]) {
  std::vector<float> vx(1048576, 1.0);
  std::vector<float> vy(1048576, 2.0l);
  std::vector<float> vz(1048576, 0.0);
  sycl::queue q(sycl::default_selector{});
  try {
    const float A(aval);
    sycl::buffer<float,1> dx { vx.data(),
    ↪  sycl::range<1>(vx.size()) };
    sycl::buffer<float,1> dy { vy.data(),
    ↪  sycl::range<1>(vy.size()) };
    sycl::buffer<float,1> dz { vz.data(),
    ↪  sycl::range<1>(vz.size()) };

    q.submit([&](sycl::handler& h) {
      sycl::accessor x(dx, h, sycl::read_only);
      sycl::accessor y(dy, h, sycl::read_only);
      sycl::accessor z(dz, h, sycl::read_write);
      h.parallel_for<class saxpy3>( sycl::range<1>{length},
      ↪  [=] (sycl::id<1> it) {
        const size_t i = it[0];
        z[I] += 3.1415 * x[i] + y[I];
      });
    });
    q.wait();
  }
  catch (sycl::exception & e) {
    std::cout << e.what() << std::endl;
    return 1;
  }
  return 0;
}
```

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**, **opencl_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.

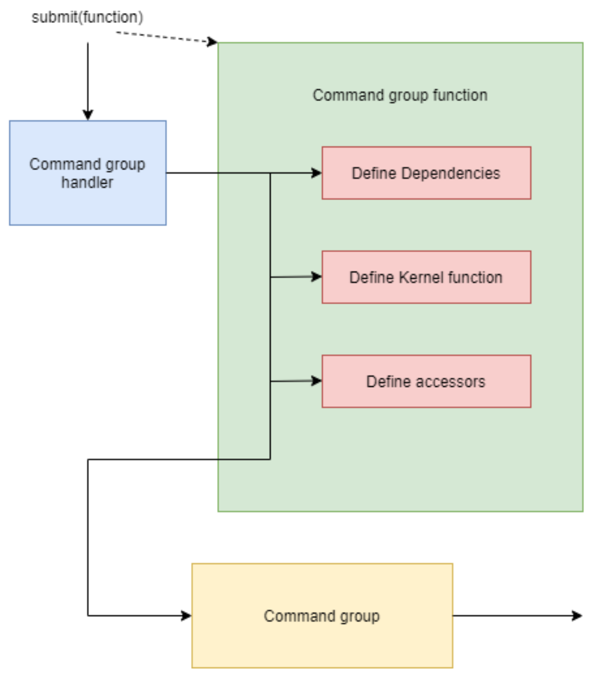
- In a **device_selector**, one overrides
  int operator()(const sycl::device& dev) const override
  and returns an integer for the priority. The higher integer, the higher priority.

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.



https://github.com/codeplaysoftware/syclacademy/blob/main/Lesson_Materials/Lecture_02_Enqueuing_a_Kernel/composing_a_command_group.png

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 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 group 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 buffers and accessors for complete control.

```
auto shared = malloc_shared<double>(42, q);

q.submit([&](handler& cgh){
  cgh.parallel_for(range{42}, [=](id<1> tid){
    shared[i] = 0.0;
  }
});
```

```
buffer<double> A{range{42}};

q.submit([&](handler& cgh){
   accessor aA{A, cgh};
   cgh.parallel_for(range{42}, [=](id<1> & idx){
     aA[idx] = 0.0;
   })
});
```

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.

```
auto shared = malloc_shared<double>(42, q);

q.submit([&](handler& cgh){
  cgh.parallel_for(range{42}, [=](id<1> tid){
    shared[i] = 0.0;
  }
});
```

```
buffer<double> A{range{42}};

q.submit([&](handler& cgh){
   accessor aA{A, cgh};
   cgh.parallel_for(range{42}, [=](id<1> & idx){
     aA[idx] = 0.0;
   })
});
```

Memory Access

In the case of one-dimensional arrays, one can call the constructor with an interator:
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).

```
buffer<double> A{range{42}};

q.submit([&](handler& cgh){
   accessor aA{A, cgh};
   cgh.parallel_for(range{42}, [=](id<1> & idx){
     aA[idx] = 0.0;
   })
});
```

Memory Access

```
#include <iostream>
#include <vector>
#include <CL/sycl.hpp>

using namespace std;
using namespace cl::sycl;
class buffer3;

int main(int, char**) {

   std::vector<float> a { 2.0, 3.0, 7.0, 4.0 };
   std::vector<float> b { 4.0, 6.0, 1.0, 3.0 };
   std::vector<float> c { 0.0, 0.0, 0.0, 0.0 };

   default_selector device_selector;

   queue q(device_selector);

   std::cout << "Running on "
             <<
             ↪  queue.get_device().get_info<info::device::name>()
             << "\n";
   {
      buffer bufA(a);
      buffer bufB(b);
      buffer bufC(c);

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

         auto accA = accessor(bufA, cgh, read_only);
         auto accB = accessor(bufB, cgh, read_only);
         auto accC = accessor(buff, cgh, write_only);
         cgh.single_task<buffer3>(bufC.get_range(), [=](id<1>
         ↪  i) {
           accC[i] = accA[i] + accB[i];
         });
      });
      q.wait_and_throw();
   }

   return 0;
}
```

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, defined and 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

```cpp
#include <iostream>
#include <sycl/sycl.hpp>

using namespace sycl;

class exception1;

int main(int, char**) {
   auto exception_handler = [] (exception_list exceptions) {
      for (std::exception_ptr const& e : exceptions) {
         try {
            std::rethrow_exception(e);
         } catch(exception const& e) {
         std::cout << "Caught asynchronous SYCL exception:\n"
                   << e.what() << std::endl;
         }
      }
   };

   queue q(default_selector{}, exception_handler);
   // actual use of the q

   try {
      q.wait_and_throw();
   } catch (exception const& e) {
      std::cout << "Caught synchronous SYCL exception:\n"
      << e.what() << std::endl;
   }
   return 0;
}
```

Building with SYCL

Linking SYCL against OpenCL.

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

Linking SYCL against MKL.

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

Linking SYCL against AMD.

```
-fsycl -fsycl-targets=amdgcn-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