GPU Programming III

Andrey Alekseenko

KTH Royal Institute of Technology & SciLifeLab

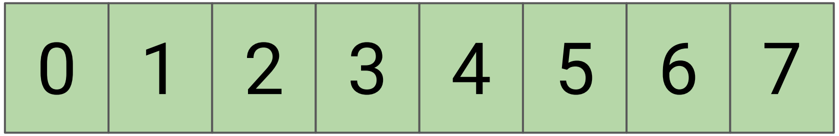
Thread divergence

```
q.submit([&](sycl::handler &cgh) {
  cgh.parallel_for(sycl::range<1>{8},
    [=](sycl::item<1> threadId) {
      int id = threadId[0];
      if (id % 2 == 0) {
        arrA[id] *= 2;
      } else {
        arrA[id] = 0;
      }
      // ...
    }
  );
});
```

0	1	2	3	4	5	6	7

Thread divergence

```
q.submit([&](sycl::handler &cgh) {
  cgh.parallel_for(sycl::range<1>{8},
    [=](sycl::item<1> threadId) {
      int id = threadId[0];
      if (id % 2 == 0) {
        arrA[id] *= 2;
      } else {
        arrA[id] = 0;
      }
      // ...
    }
  );
});
```



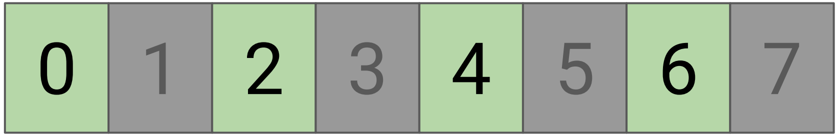
Thread divergence

```
q.submit([&](sycl::handler &cgh) {
  cgh.parallel_for(sycl::range<1>{8},
    [=](sycl::item<1> threadId) {
      int id = threadId[0];
      if (id % 2 == 0) {
        arrA[id] *= 2;
      } else {
        arrA[id] = 0;
      }
      // ...
    }
  );
});
```

0	1	2	3	4	5	6	7

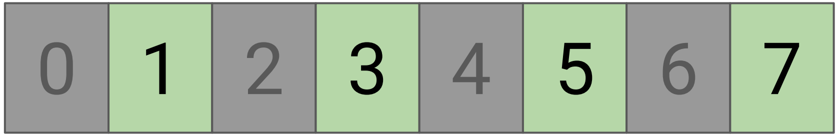
Thread divergence

```
q.submit([&](sycl::handler &cgh) {
  cgh.parallel_for(sycl::range<1>{8},
    [=](sycl::item<1> threadId) {
      int id = threadId[0];
      if (id % 2 == 0) {
        arrA[id] *= 2;
      } else {
        arrA[id] = 0;
      }
      // ...
    }
  );
});
```



Thread divergence

```
q.submit([&](sycl::handler &cgh) {
  cgh.parallel_for(sycl::range<1>{8},
    [=](sycl::item<1> threadId) {
      int id = threadId[0];
      if (id % 2 == 0) {
        arrA[id] *= 2;
      } else {
        arrA[id] = 0;
      }
      // ...
    }
  );
});
```



Thread divergence

```
q.submit([&](sycl::handler &cgh) {
  cgh.parallel_for(sycl::range<1>{8},
    [=](sycl::item<1> threadId) {
      int id = threadId[0];
      if (id % 2 == 0) {
        arrA[id] *= 2;
      } else {
        arrA[id] = 0;
      }
      // ...
    }
  );
});
```

0	1	2	3	4	5	6	7

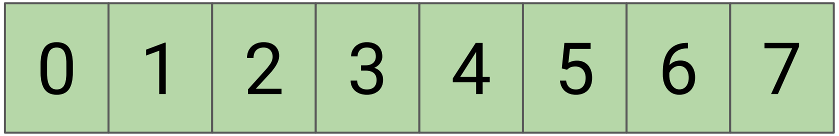
Thread divergence

```
q.submit([&](sycl::handler &cgh) {
  cgh.parallel_for(sycl::range<1>{8},
    [=](sycl::item<1> threadId) {
      int id = threadId[0];
      if (id < 256) {
        arrA[id] *= 2;
      } else {
        arrA[id] = 0;
      }
      // ...
    }
  );
});
```

0	1	2	3	4	5	6	7

Thread divergence

```
q.submit([&](sycl::handler &cgh) {
  cgh.parallel_for(sycl::range<1>{8},
    [=](sycl::item<1> threadId) {
      int id = threadId[0];
      if (id < 256) {
        arrA[id] *= 2;
      } else {
        arrA[id] = 0;
      }
      // ...
    }
  );
});
```



Thread divergence

```
q.submit([&](sycl::handler &cgh) {
  cgh.parallel_for(sycl::range<1>{8},
    [=](sycl::item<1> threadId) {
      int id = threadId[0];
      if (id < 256) {
        arrA[id] *= 2;
      } else {
        arrA[id] = 0;
      }
      // ...
    }
  );
});
```

0	1	2	3	4	5	6	7

Thread divergence

```
q.submit([&](sycl::handler &cgh) {
  cgh.parallel_for(sycl::range<1>{8},
    [=](sycl::item<1> threadId) {
      int id = threadId[0];
      if (id < 256) {
        arrA[id] *= 2;
      } else {
        arrA[id] = 0;
      }
      // ...
    }
  );
});
```

0	1	2	3	4	5	6	7

Thread divergence

- Details depend on hardware, but:
    - Divergence within sub-group: bad
    - Divergence between sub-groups: ok
        - Optimization opportunity: kernel fusion

Thread divergence

- Details depend on hardware, but:
  - Divergence within sub-group: bad
  - Divergence between sub-groups: ok
    - Optimization opportunity: kernel fusion

```
cgh.parallel_for(sycl::nd_range<1>{n, k},
  [=](sycl::nd_item<1> item) {
    int blockId = item.get_group(0);
    switch(blockId) {
      case 0:
        // ...
      case 1:
        // ...
      // ...
     }
```

Thread divergence

- Details depend on hardware, but:
  - Divergence within sub-group: bad
  - Divergence between sub-groups: ok
    - Optimization opportunity: kernel fusion

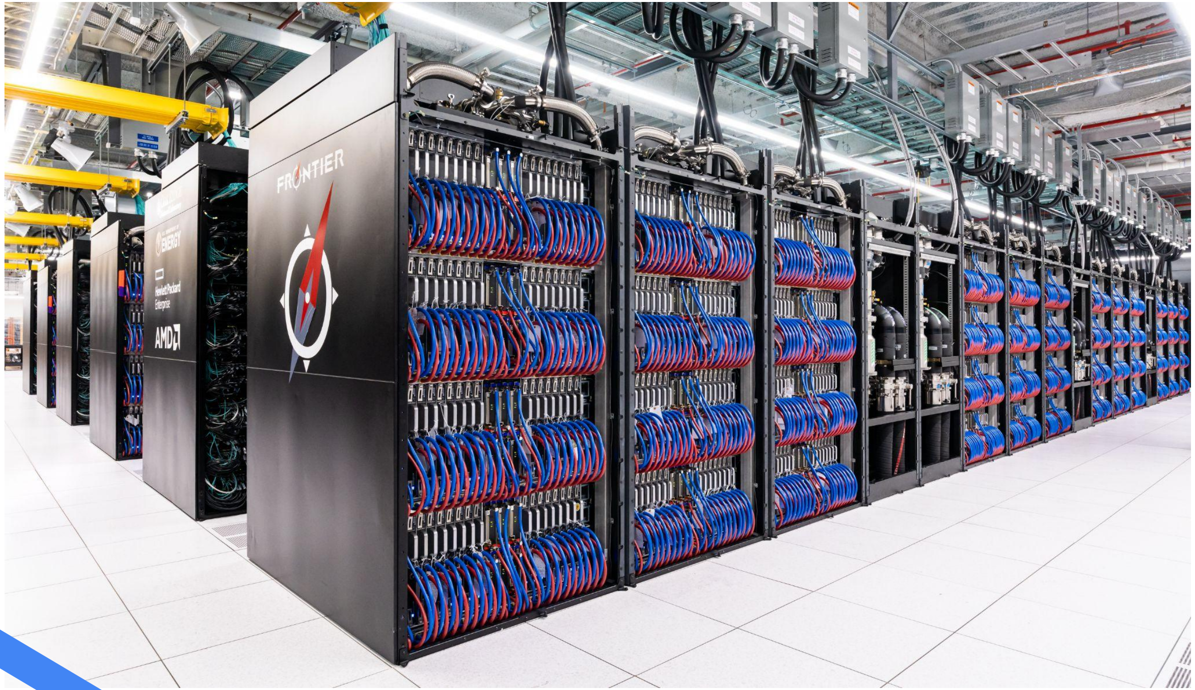
- Don’t assume lock-step to avoid barriers on local memory
  - It’s not true in hardware
  - Compiler might optimize things away
    - At least use memory fences

Multi-GPU programming

https://www.flickr.com/photos/olcf/52117623798/

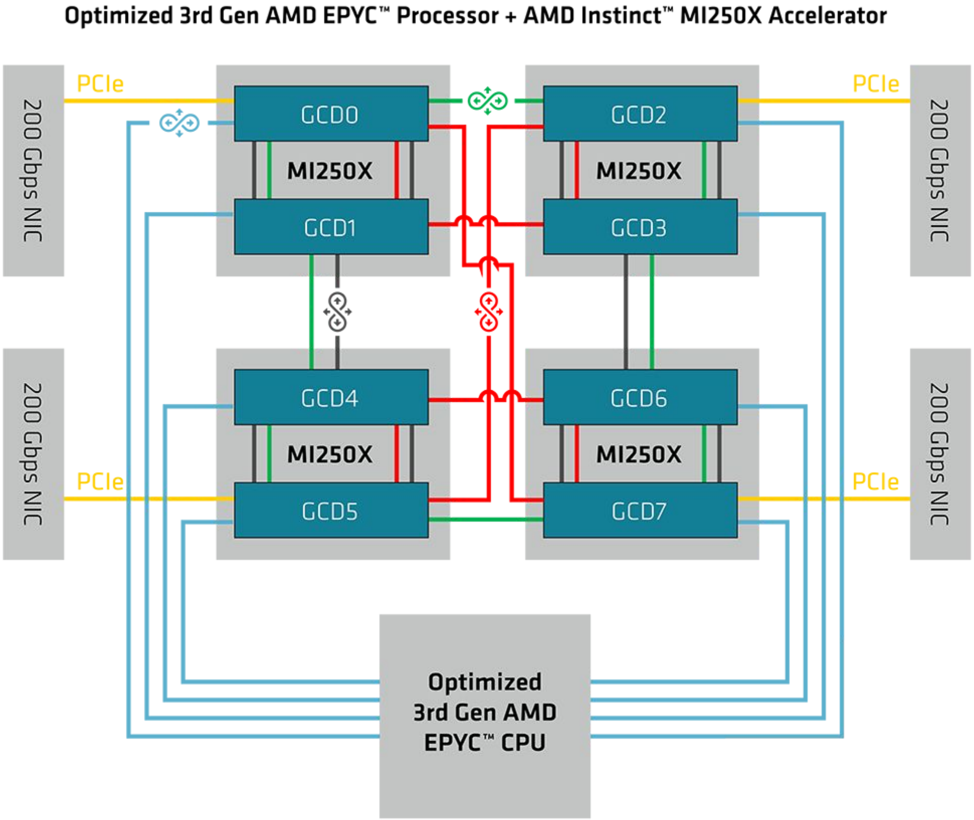
Multi-GPU programming

- Your problem does not fit into a memory of a single GPU?
- You want your program to run even faster?
-



Multi-GPU programming

- When you have existing MPI program, do the simplest thing first
- It might not be great, but it can be good enough.



https://www.pdc.kth.se/hpc-services/computing-systems/about-the-dardel-hpc-system-1.1053338

Multi-GPU programming without MPI

- Different queues for different devices
- Accessors or events to set up dependencies
- `queue::copy` can copy between devices
  - Devices from the same vendor
  - Memory is allocated via a multi-device context

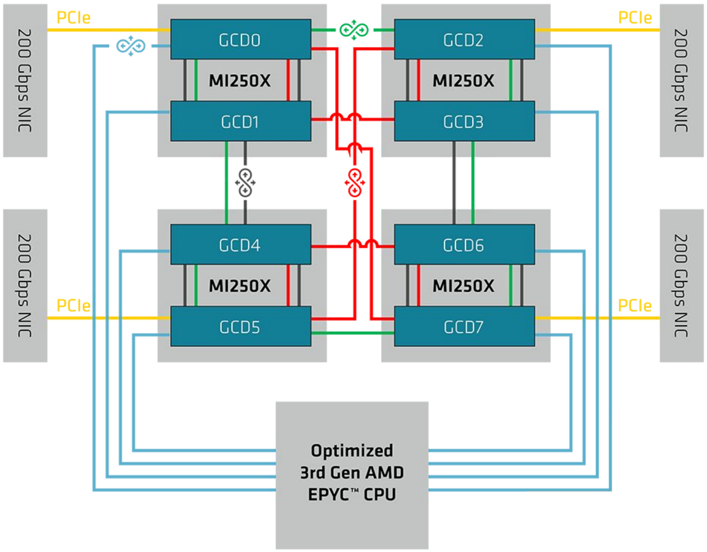
```
std::vector<sycl::device> devices{dev1, dev2};
sycl::context ctx{devices};
sycl::malloc_device<int>(n, dev1, ctx);
```

- Possibly, your kernel can read from another device's memory

Multi-GPU programming with MPI

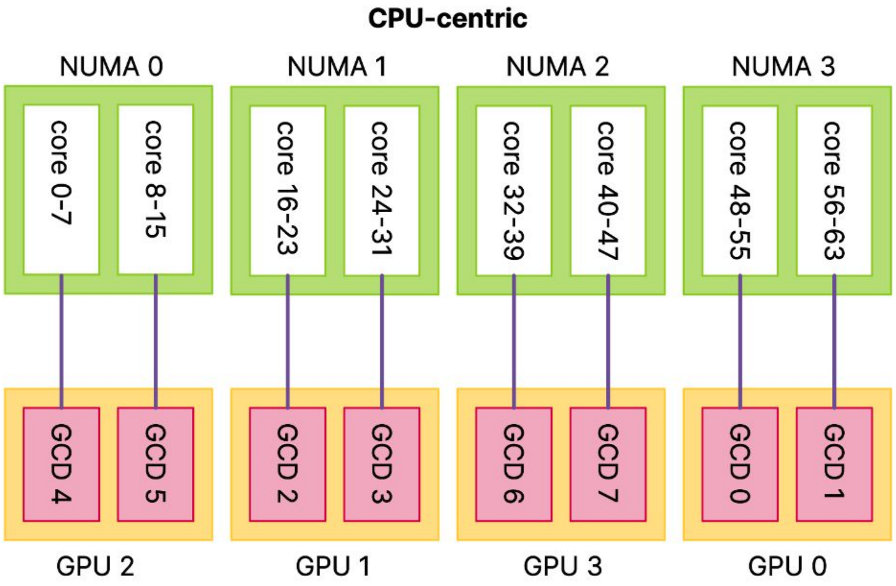
- Usually, 1 device = 1 rank
- Make sure MPI library is *GPU-aware* (read cluster docs!)
- Beware of hardware topology

Optimized 3rd Gen AMD EPYC™ Processor + AMD Instinct™ MI250X Accelerator



Multi-GPU programming with MPI

- Usually, 1 device = 1 rank
- Make sure MPI library is *GPU-aware* (read cluster docs!)
- Beware of hardware topology



https://docs.lumi-supercomputer.eu/hardware/lumig/

Multi-GPU programming with MPI

- Usually, 1 device = 1 rank
- Make sure MPI library is *GPU-aware* (read cluster docs!)
- Beware of hardware topology

- Just give your GPU pointers to `MPI_Send`/`MPI_Recv` & co.
  - In SYCL, will work only with USM
  - Can be used with CUDA, HIP, Kokkos, OpenMP, OpenACC
- Data copies can overlap with compute
- Main problem: synchronization

Multi-GPU programming with MPI

Different kind of MPI libraries:

- GPU-aware: MPI functions accept GPU pointers
  - Widely supported
- Stream-aware: MPI functions can synchronize with GPU queues
  - Only prototypes
- Device-initiated: MPI functions can be called from GPU
  - Early support
  - Usually, one-sided communications only (MPI_Put/MPI_Get)
  - OpenSHMEM/NVSHMEM is here too

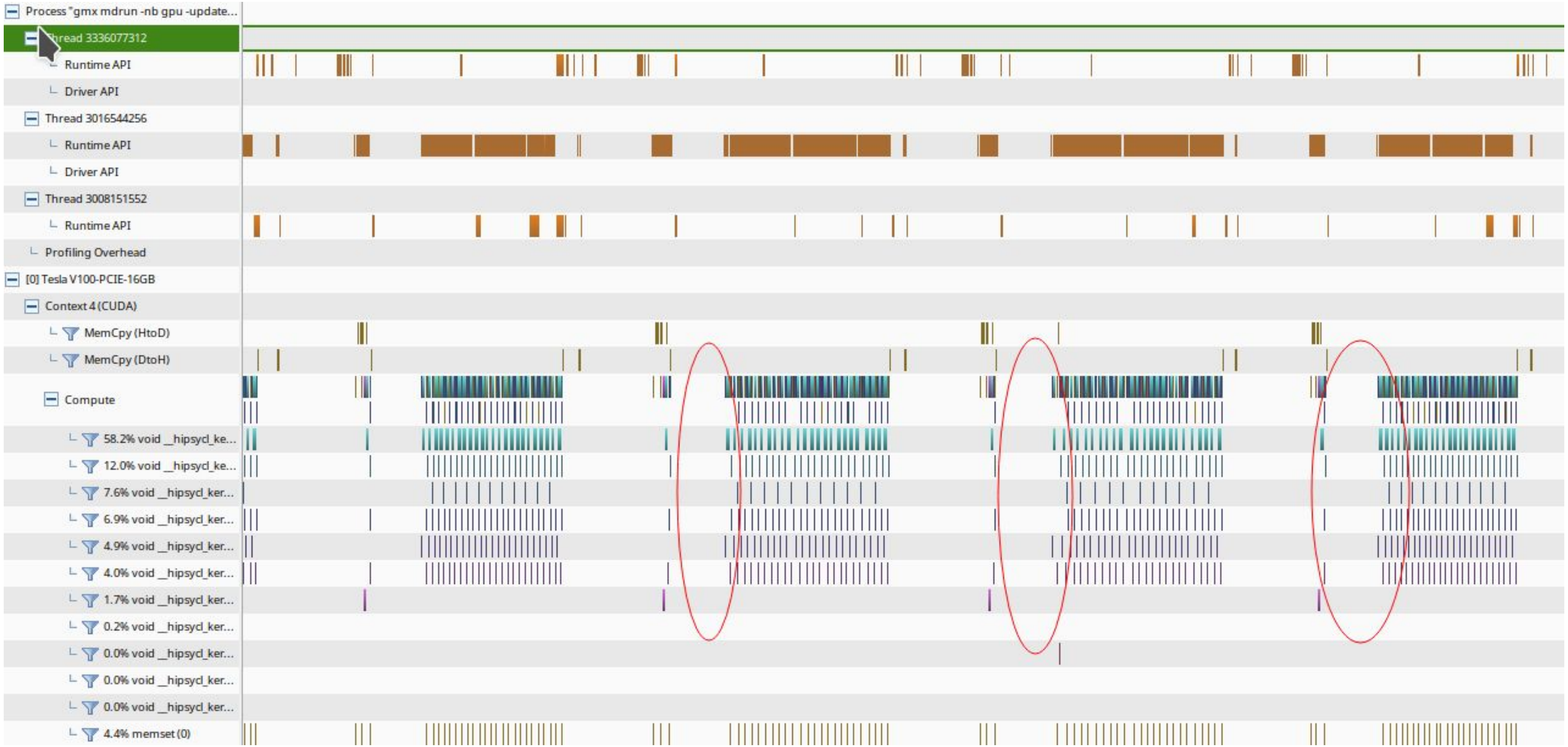
Note: this is not an established terminology

Performance measurements best practices

- Measure before you optimize!
  - Both kernel performance and application performance.

- Don’t compare to single-threaded application
- Discard first few iterations
- Check that application performance is stable
  - For integrated GPUs: total power limit
- Use vendor profiling and tracing tools
  - Use instrumentation

Performance measurements best practices



Debugging

A lot of problems disappear in debug mode.
From least to most “intrusive”:

- Temporary output array
- Use sanitizer
  - Only NVIDIA has it working well nowadays
    - Benefits of portable programming models!
- Use proper debugger
  - Can require messing with system configuration
- `sycl::stream` / CUDA/HIP `printf` from kernel
- Running on the CPU

Debugging

```
$ ZET_ENABLE_PROGRAM_DEBUGGING=1 gdb-oneapi ./sycl_simple_kernel
(gdb) break 16
Breakpoint 1 at 0x406a4e: file sycl_simple_kernel.cpp, line 16.
(gdb) run
...
Thread 2.97 hit Breakpoint 1.2, with SIMD lanes [0-7], can_run_kernel(sycl::_V1::device
const&)::{lambda(sycl::_V1::handler&)#1}[....] (this=..., threadId=sycl::_V1::id<1> = {...}) at
sycl_simple_kernel.cpp:16
16                    buffer_dev[threadId] = val;
(gdb) thread 2.97:0
[Switching to thread 2.97:0 (Thread 1.97 lane 0)]
16                    buffer_dev[threadId] = val;
(gdb) print val
$2 = 0
(gdb) thread 2.97:1
[Switching to thread 2.97:1 (Thread 1.97 lane 1)]
16                    buffer_dev[threadId] = val;
(gdb) print val
$3 = 42
```

Runtime sanitizers

```
$ compute-sanitizer --help
  --tool arg (=memcheck)                Set the tool to use.
                                        memcheck  : Memory access checking
                                        racecheck : Shared memory hazard checking
                                        synccheck : Synchronization checking
                                        initcheck : Global memory initialization checking
$ compute-sanitizer --tool=initcheck ./transpose_matrix
========= COMPUTE-SANITIZER
========= Uninitialized __global__ memory read of size 4 bytes
=========  at 0x90 in transpose_matrix_v2.cpp:24:void __hipsycl_kernel<....>) (instance 1)]>()
=========  by thread (0,2,0) in block (0,0,0)
=========  Address 0x7fe1f8008000
=========  Saved host backtrace up to driver entry point at kernel launch time
=========  Host Frame: [0x3304e0]
=========             in /lib/x86_64-linux-gnu/libcuda.so.1
=========  Host Frame: [0x1498c]
=========             in /opt/tcbsys/cuda/12.1/lib64/libcudart.so.12
...
```

Preparing code for GPU porting

- Identify targeted parts
  - 10% of the code accounts for 90% of the execution time (if you are lucky)
- Equivalent GPU libraries
- Refactor loops
  - Minimize dependencies, even at the cost of extra compute
- Memory access optimizations

Porting between GPU frameworks

- CUDA → HIP
  - hipify-perl
  - hipify-clang
- CUDA → SYCL (oneAPI)
  - SYCLomatic
- OpenACC → OpenMP
  - Clacc
- Any → Any
  - ChatGPT

Vector addition: CUDA/HIP

```
__global__ void vector_add(float *A, float *B, float *C, int n) {
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  if (tid < n) {
    C[tid] = A[tid] + B[tid];
  }
}

// Allocate GPU memory
(cuda|hip)Malloc((void**)&Ad, n * sizeof(float)); // ...
// Copy the data from the CPU to the GPU
(cuda|hip)Memcpy(Ad, Ah, sizeof(float) * n, (cuda|hip)MemcpyHostToDevice);
// ...
// Define grid dimensions: how many threads to launch
dim3 blockDim{256, 1, 1};
dim3 gridDim{(n/256)*256 + 1, 1, 1};
vector_add<<<gridDim, blockDim>>>(Ad, Bd, Cd, n);
```

Porting between GPU frameworks

- CUDA → HIP
  - hipify-perl
  - hipify-clang
- CUDA → SYCL (oneAPI)
  - SYCLomatic
- OpenACC → OpenMP
  - Clacc
- Any → Any
  - ChatGPT

Choosing the framework

- Portability?
  - Portability ≠ performance-portability
- Programming effort?
  - Familiarity with language
  - Support (commercial and community), documentation
  - Future maintenance
- Performance requirements?
  - How fast the application must go to get things done

Intel® Student Ambassador for oneAPI

- This program is focused on undergraduate and graduate students who are passionate about technology and working with developer communities to promote learning, sharing, and collaboration. It provides opportunities for students to enhance their oneAPI skills, expand their network, and learn about the cutting-edge Intel® hardware and software products.

- https://www.intel.com/content/www/us/en/developer/tools/oneapi/training/academic-program/student-ambassador.html

Questions?

- andreyal@kth.se

- More detailed version of the course: https://enccs.github.io/gpu-programming/

- Courses, documentation and existing code for the chosen framework