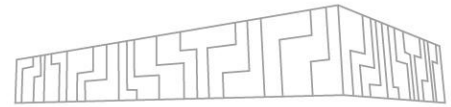
VSB TECHNICAL UNIVERSITY OF OSTRAVA | IT4INNOVATIONS NATIONAL SUPERCOMPUTING CENTER

Leveraging accelerated hardware

GPUs

Jakub Homola

Outline



- What is a GPU? Why should I use it?
- Architecture of GPUs and computational nodes with GPUs
- How to use GPUs
- Introduction to CUDA
- Allocating GPU nodes on Karolina, CUDA hands-on
- Other GPU programming models

What is a GPU?

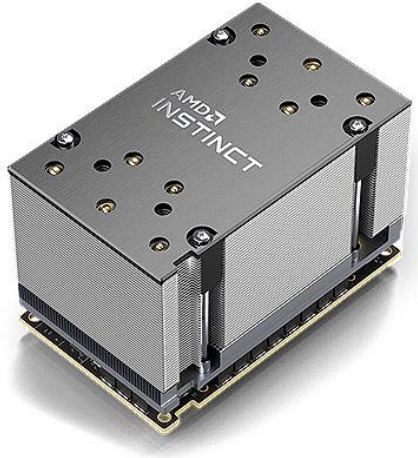
Why should I use it?

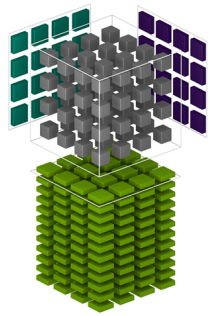
What is a GPU?

- GPU = Graphics Processing Unit
- Not much common with graphics *in today's HPC*
  - No display output
- => Accelerators, GPGPU (General Purpose computing on GPU)

NVIDIA

AMD

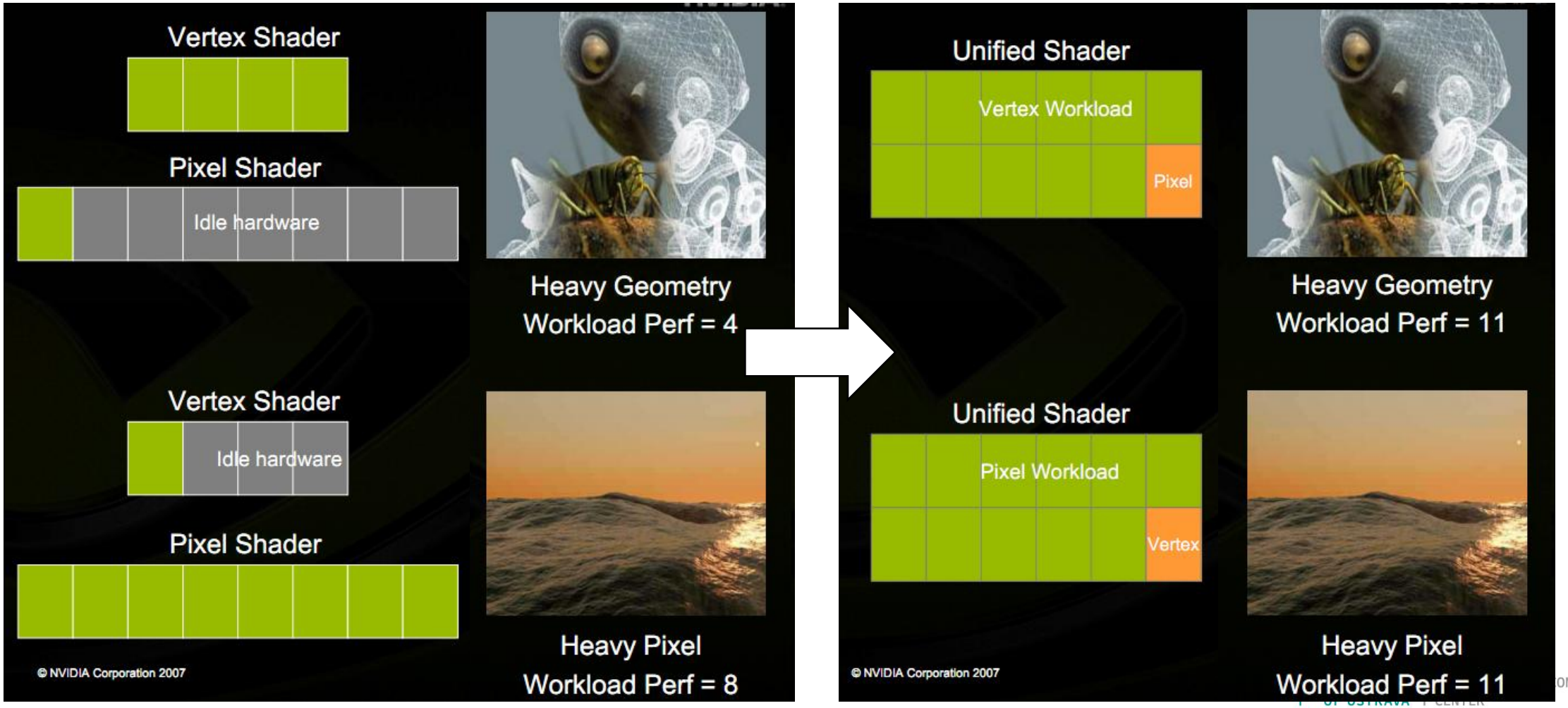




intel

How GPGPUs came to be

- Generalizing/unifying specialized hardware

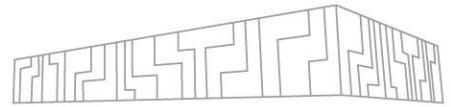


GPUs in today's supercomputers

- TOP500, GREEN500 lists (June 2024)
  - Ranking supercomputers
  - FLOP/s performance, FLOP/s/W energy efficiency
- 38 % of the TOP500 supercomputers contain GPUs
  - 66 out of top 100, 9 out of top 10 in TOP500
  - 78 out of top 100, 10 out of top 10 in GREEN500
- GPUs are the way to go for massive performance and energy efficiency
- All vendors are represented
  - NVIDIA
  - AMD
  - Intel

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904	1,206.00	1,714.81	22,786
2	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	9,264,128	1,012.00	1,980.01	38,698
3	**Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Azure Microsoft Azure United States	2,073,600	561.20	846.84	
4	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
5	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,752,704	379.70	531.51	7,107
6	**Alps** - HPE Cray EX254n, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, HPE Swiss National Supercomputing Centre (CSCS) Switzerland	1,305,600	270.00	353.75	5,194
7	**Leonardo** - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, EVIDEN EuroHPC/CINECA Italy	1,824,768	241.20	306.31	7,494
8	**MareNostrum 5 ACC** - BullSequana XH3000, Xeon Platinum 8460Y+ 32C 2.3GHz, NVIDIA H100 64GB, Infiniband NDR, EVIDEN EuroHPC/BSC Spain	663,040	175.30	249.44	4,159
9	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	200.79	10,096
10	**Eos NVIDIA DGX SuperPOD** - NVIDIA DGX H100, Xeon Platinum 8480C 56C 3.8GHz, NVIDIA H100, Infiniband NDR400, Nvidia NVIDIA Corporation United States	485,888	121.40	188.65	

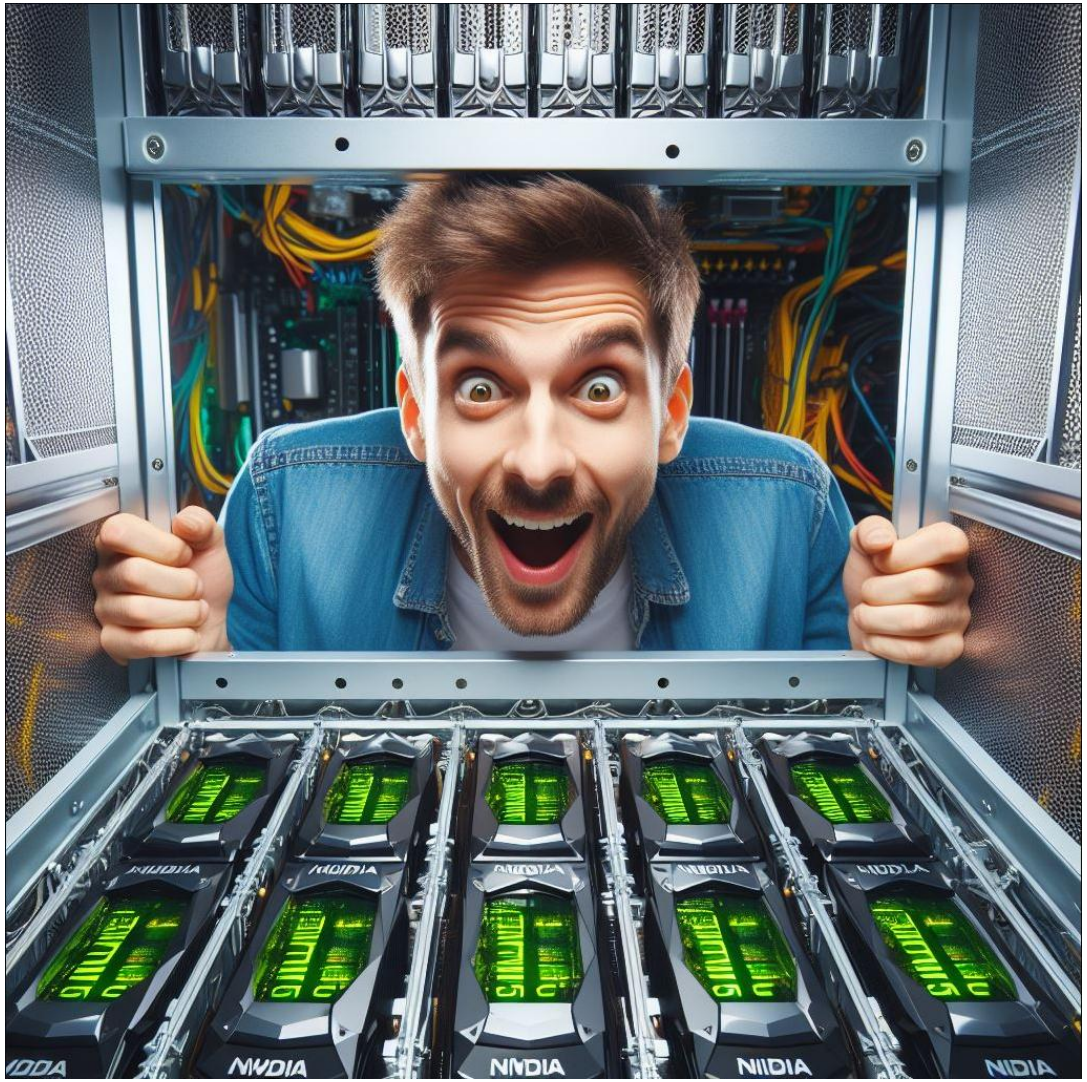
CPU vs GPU



	Device name	Performance [TFLOP/s]	Memory bandwidth [GB/s]	TDP [W]	Energy efficiency [GFLOP/s/W]
GPU	NVIDIA H100	66.9	3350	700	96
GPU	AMD MI250X	95.7	3200	500	191
GPU	Intel Data Center GPU Max	52.0	3280	600	87
CPU	NVIDIA Grace CPU	3.5	500	250	14
CPU	AMD EPYC 9654	3.7	460	360	10
CPU	Intel Xeon Max 9480 DDR5	3.4	300	350	10
CPU	Intel Xeon Max 9480 HBM	3.4	1640	350	10
CPU	Fujitsu A64FX	3.4	1024	180	19

Data might be slightly inaccurate, but the main point stands

So, I hope you are convinced.



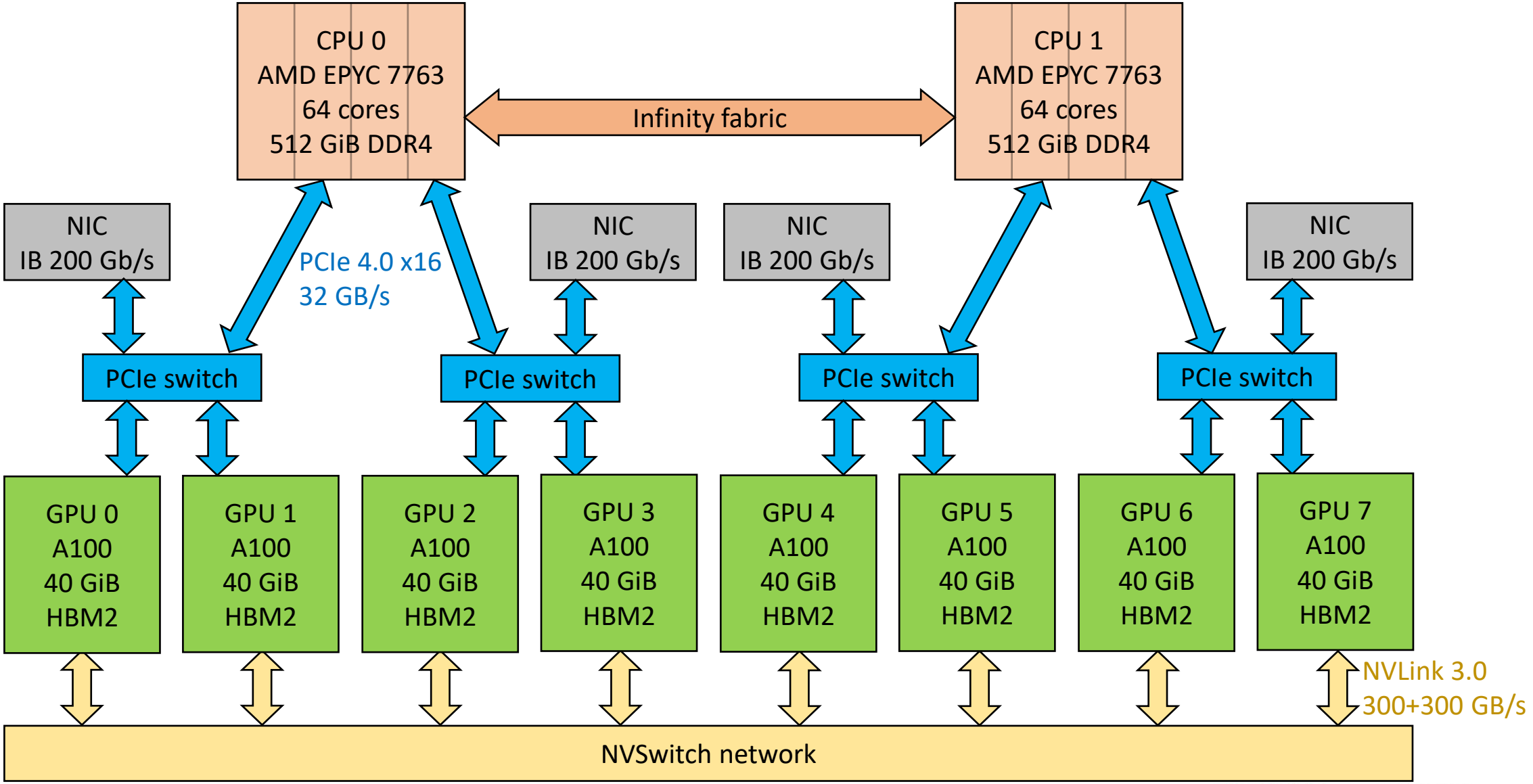
DALL·E 3 via copilot: man with an excited face expression looking though a supercomputer cabinet containing nvidia gpus

The problem with GPUs

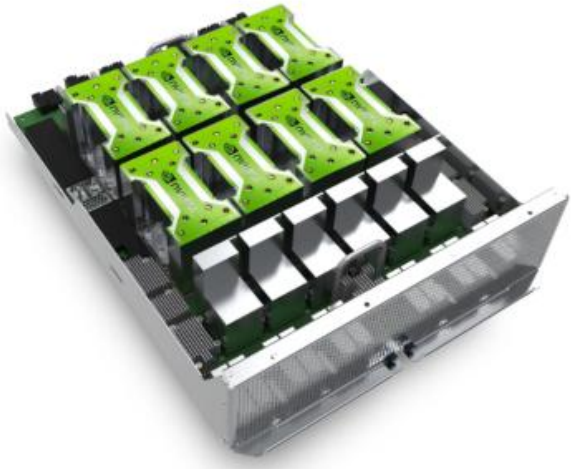
- Of course, there's a catch
- GPU is not a do-it-all device
- Located separately from the CPU, as a co-processor
- GPUs are focused on throughput and FLOP/s performance
  - Calculating many things at once, but slowly
- Not all algorithms are suitable for GPUs
  - High parallelism is required
- GPUs are inferior in:
  - Latency-sensitive operations
  - Highly branching code
  - Random memory access
- Specialized hardware for specific types of problems

Architecture of GPUs and GPU nodes

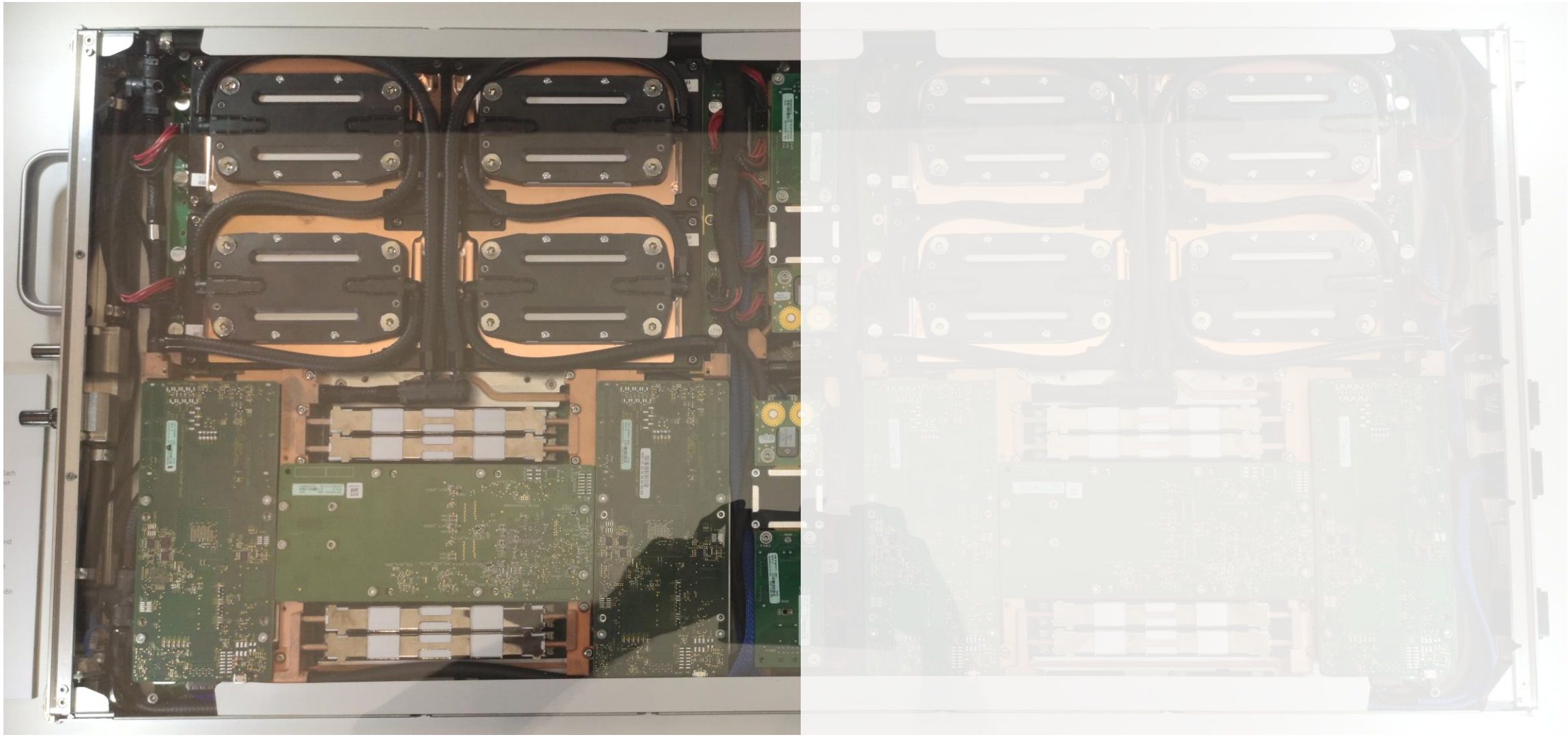
Karolina GPU node architecture



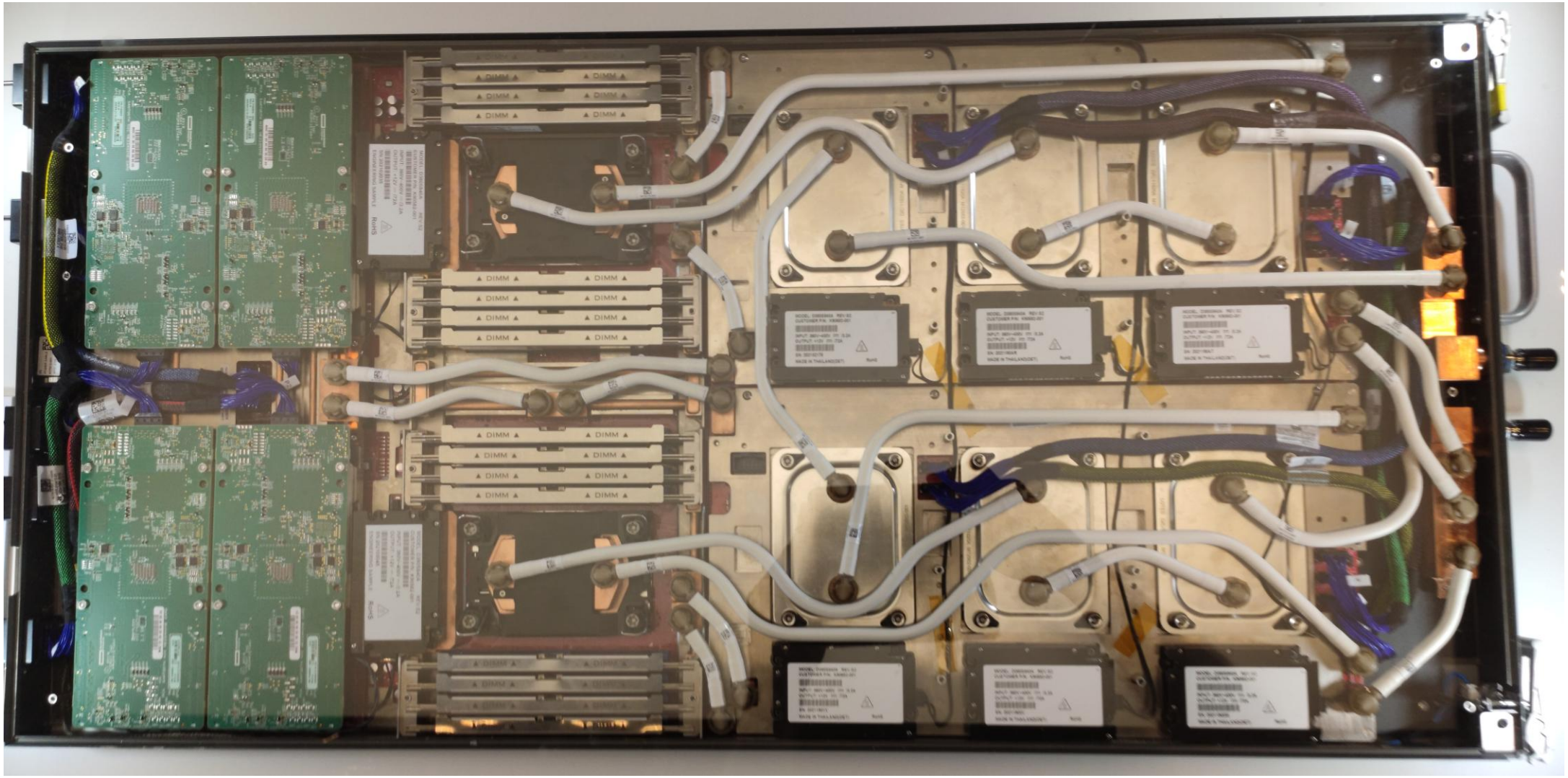
Karolina GPU node (8x NVIDIA A100)



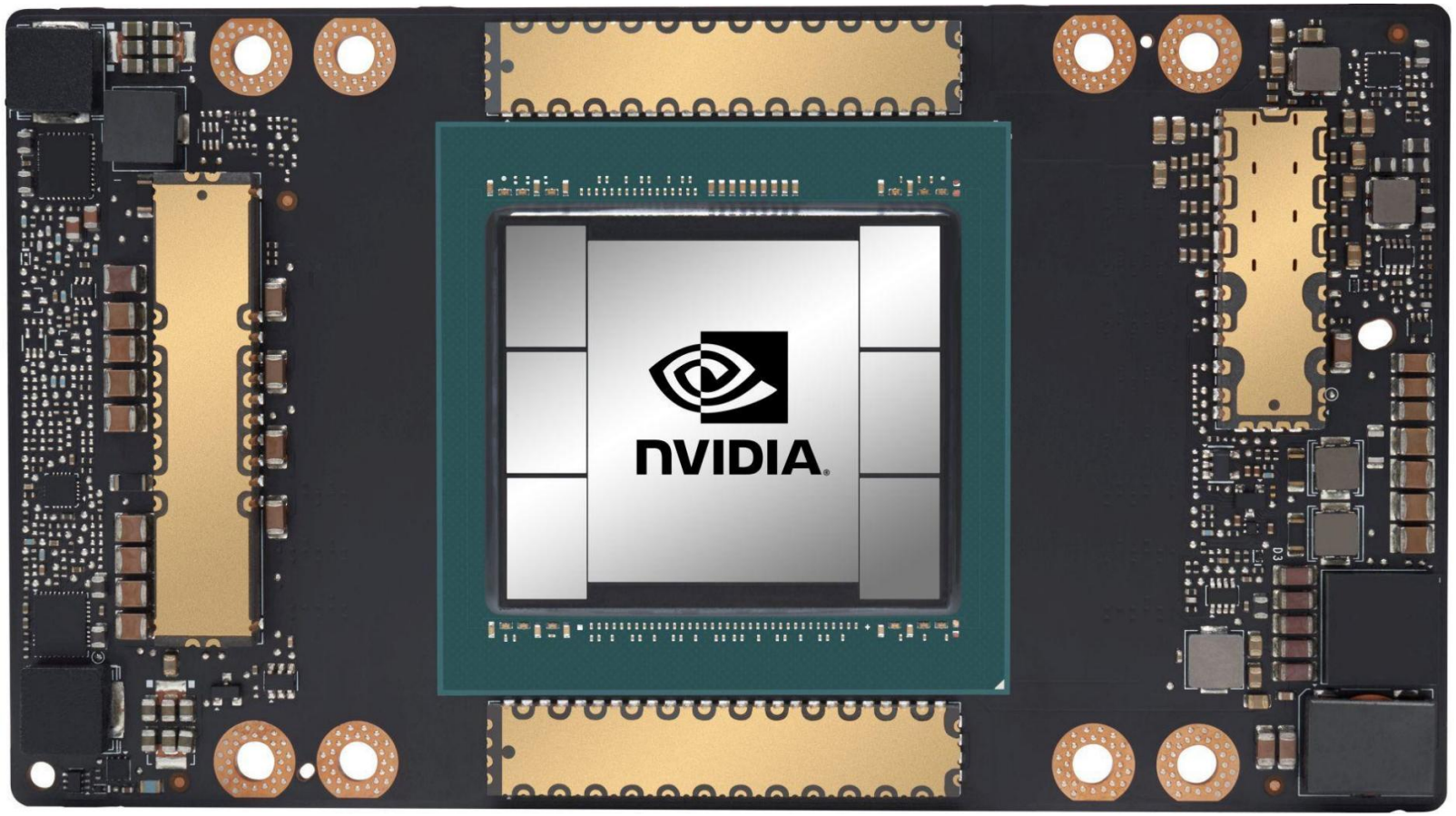
Frontier node (4x AMD MI250X)



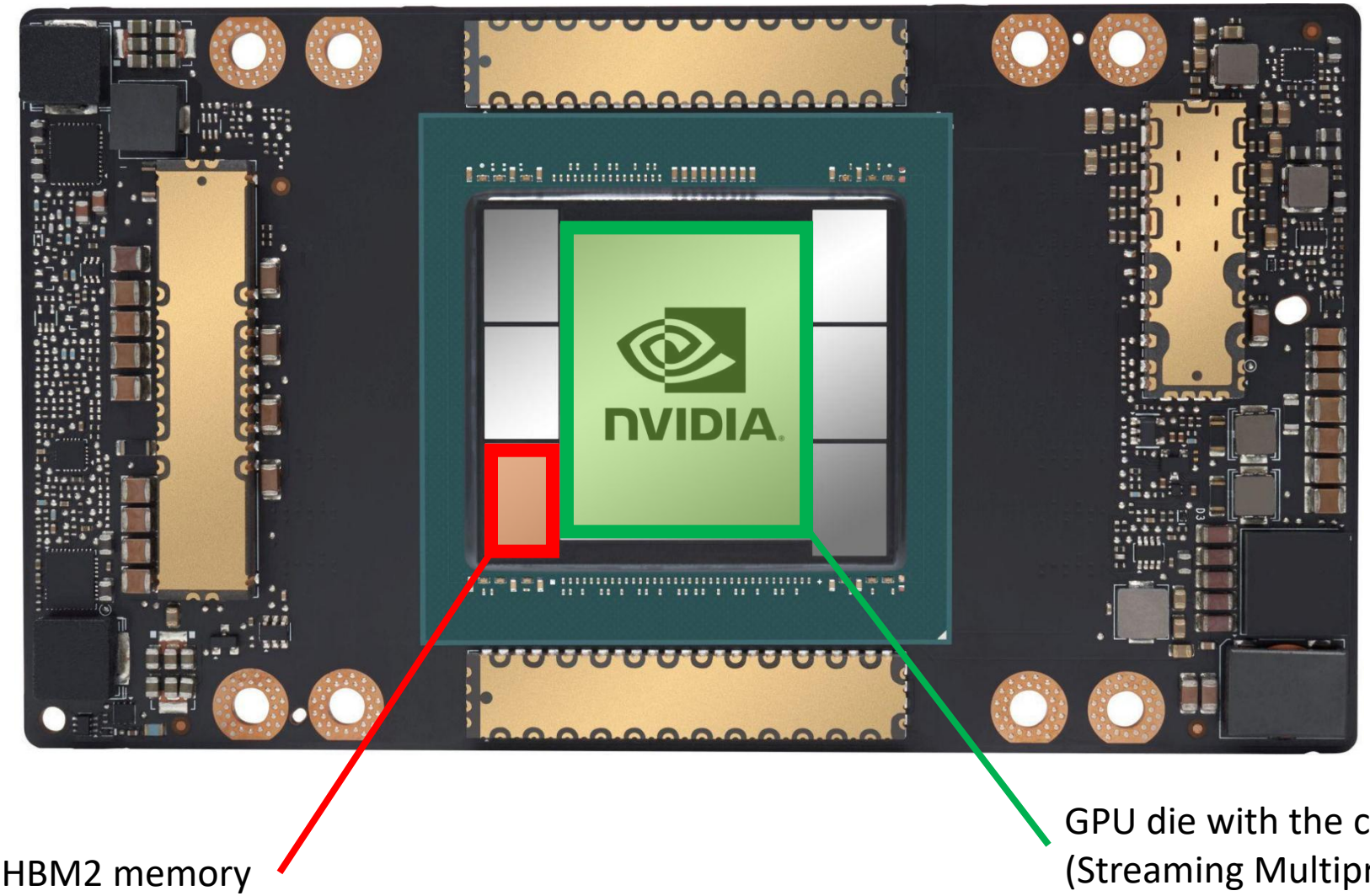
Aurora node (6x Intel Data Center GPU Max)



NVIDIA A100 SXM4 module

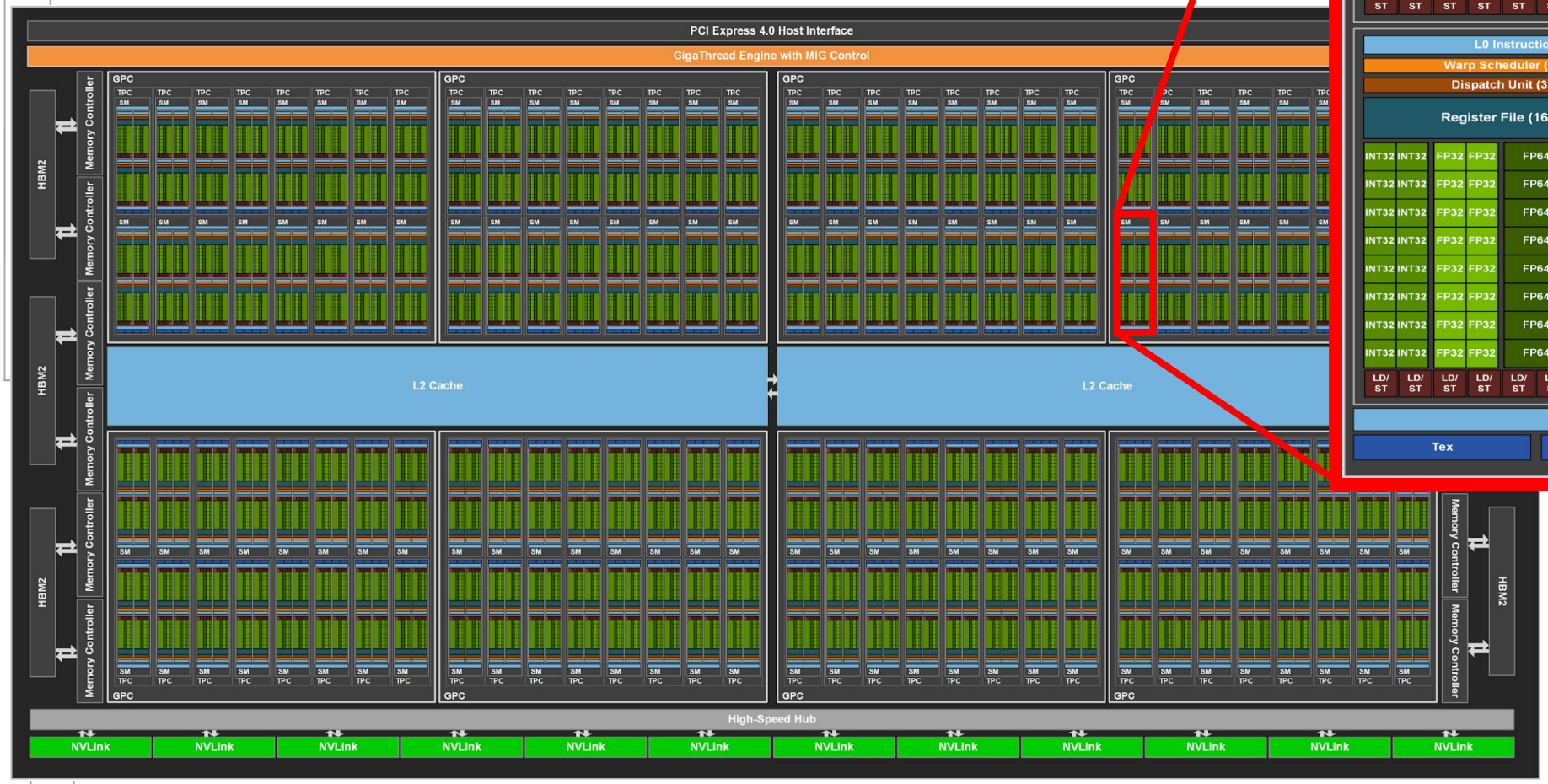


NVIDIA A100 SXM4 module



NVIDIA A100 die

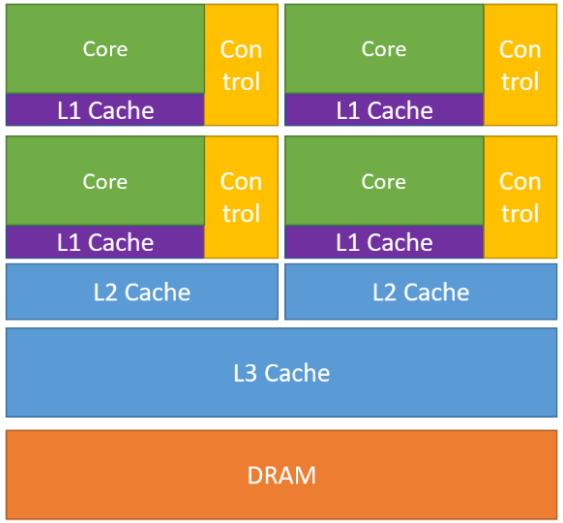
- Nvidia GA100 full GPU architecture
- 128 streaming multiprocessors (SM)
  - Each SM has 32 FP64 units, 64 FP32 units, …



CPU vs GPU

GPUs

- tailored for compute-intensive, highly data parallel computation
- many parallel execution units
- have significantly faster and more advanced memory interfaces
- more transistors is devoted to data processing
- less transistors for data caching and flow control

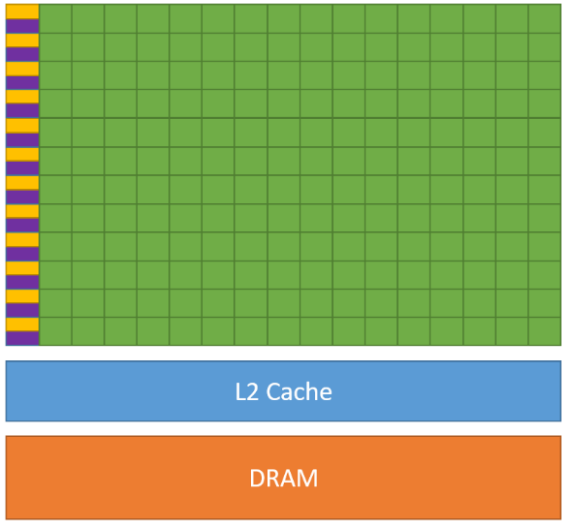


CPUs

- Powerful ALU
- Large caches
- Branch prediction

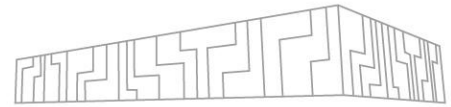
GPUs

- Small caches
- Simple control
- Many energy efficient ALUs
- Require massive number of threads



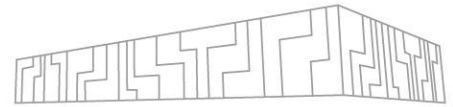
How do I use GPUs?

How do I use GPUs?



- Applications
  - Use applications that can use the GPU
- Libraries, packages
  - Use libraries/packages which use the GPU internally
- Compiler directives
  - Annotate your current code to make it run on GPU
- Programming languages
  - Write GPU kernels manually

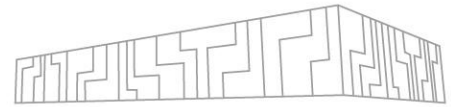
GPU-enabled applications



- Just enable GPU support in the app/program you are using
- E.g., VASP, GROMACS, OpenFOAM, …
- Some apps use the GPU by default
- Some apps might need the correct config
  - ./app --use-gpu
- Some apps might need recompilation
  - cmake -DAPP_ENABLE_GPU=true
- Some apps don’t support GPUs at all
  - Try searching for an alternative
- Consult the application documentation

- If no such application exists => need to write your own

Libraries, packages

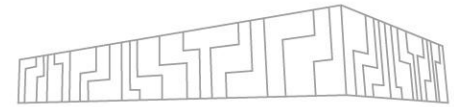


- If you write your own app/script, use libraries/packages that use the GPU internally

- PyTorch
- TensorFlow
- PETSc
- Trilinos
- TNL
- cuBLAS, rocBLAS, oneapi::mkl::blas, …
- *sparse, *fft, …

- Again, consult the documentation

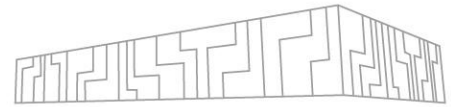
Compiler directives



- Start with your C/C++/Fortran CPU code
- Annotate it to offload certain parts to the GPU
- Generic, no GPU or accelerator type assumed
- Examples
  - OpenACC
  - OpenMP offloading

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

Programming languages



- Language (extensions) for writing GPU kernels
  - Special code that runs on the GPU

- CUDA
  - Programming model for NVIDIA GPUs
  - The state of the art
- HIP
  - Mainly for AMD, but also for NVIDIA GPUs
  - Created by AMD to mimic CUDA, to ease the transition to AMD GPUs
- SYCL
  - Mainly for Intel GPUs, but developed as a generic programming model for all GPUs
  - Modern C++17, only headers and libraries, no language extensions
- OpenCL
  - Older, for all GPUs

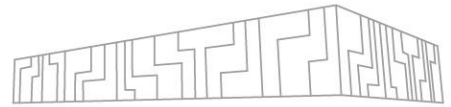
Introduction to CUDA

CUDA overview

- Previously Compute Unified Device Architecture, now only CUDA
- Programming model for NVIDIA GPUs
- C/C++/Fortran language extension
  - We will work with C++

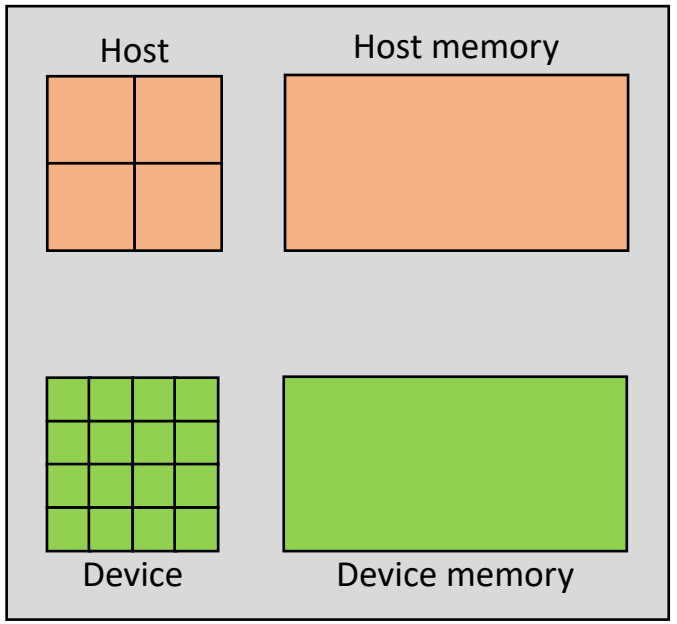
- CUDA toolkit
  - NVCC compiler
  - Drivers, runtime libraries
  - High performance libraries – cuBLAS, cuSPARSE, cuFFT, CUB, …
  - Profiler, debugger
  - …

The elephant in the room - memory

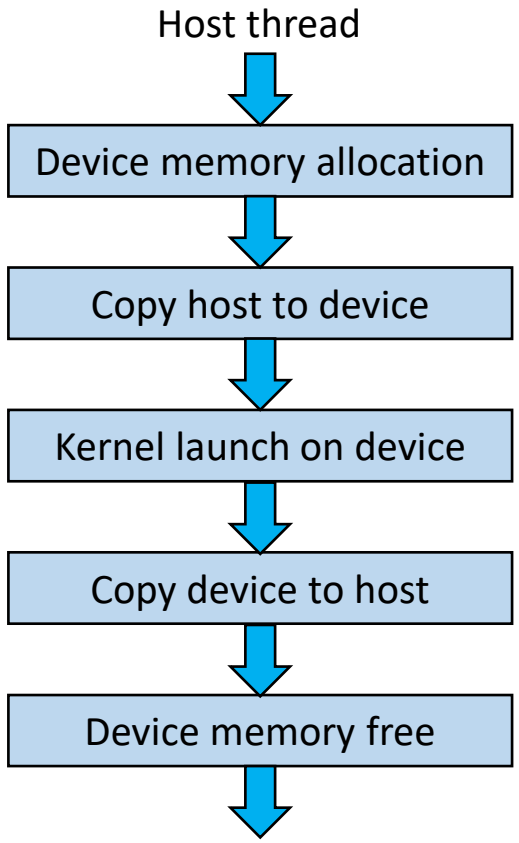


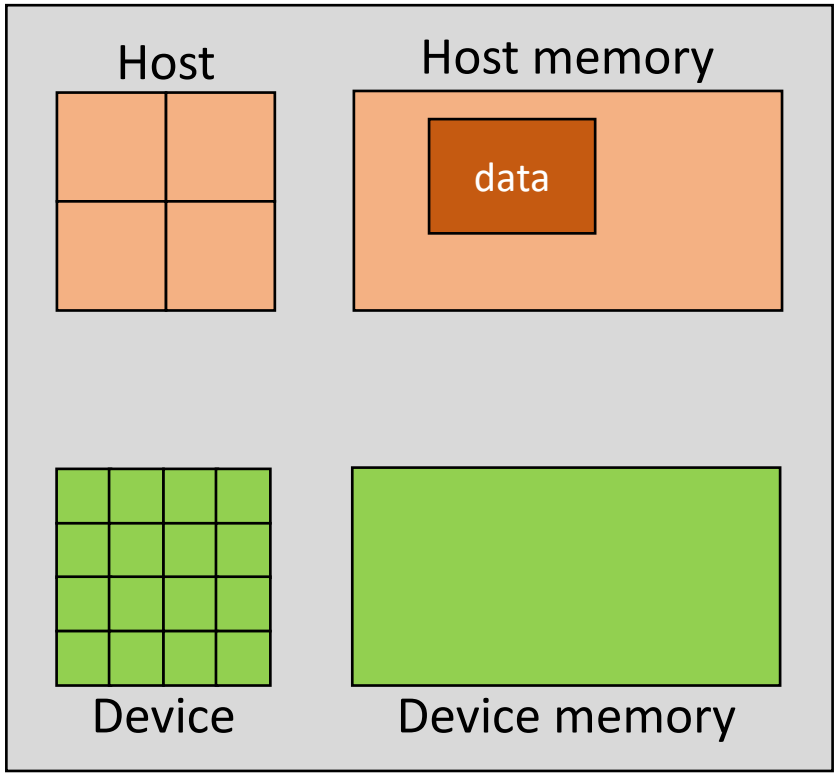
- CPU and GPU have separate memories
- Memory needs to be manually allocated on the GPU
- Data needs to be copied between CPU and GPU

- Host = CPU
- Device = GPU
- Kernel = function running on the device

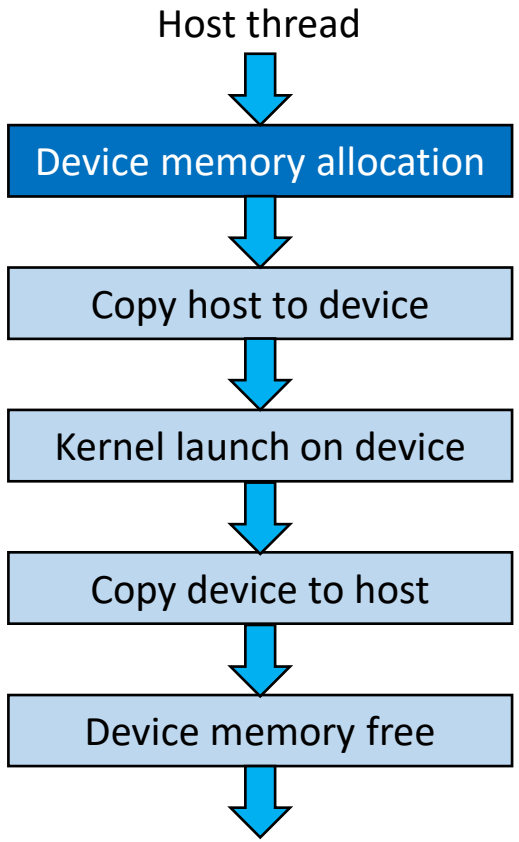


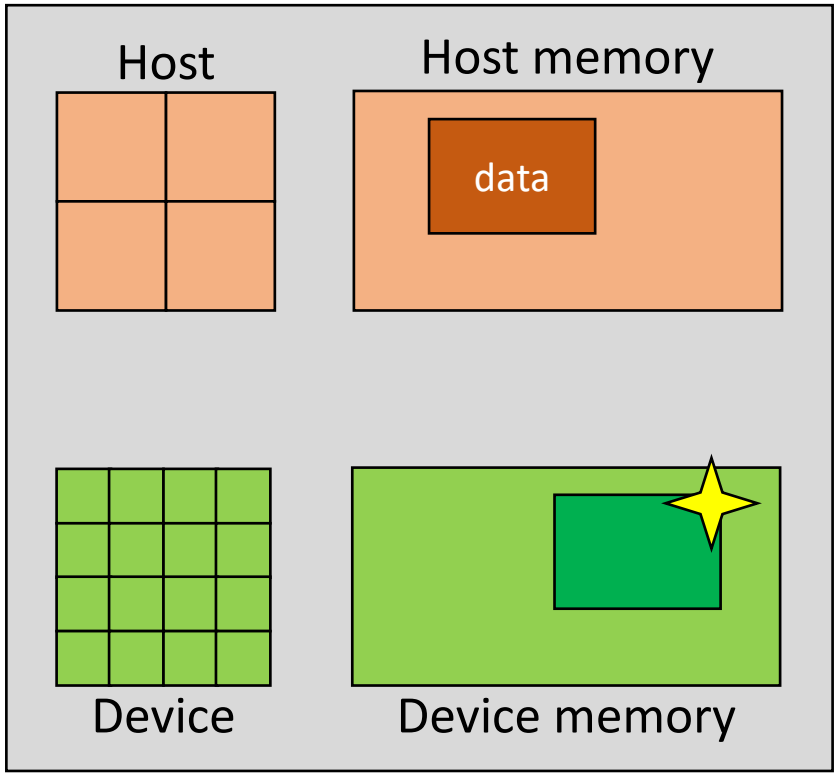
Structure of a basic CUDA program



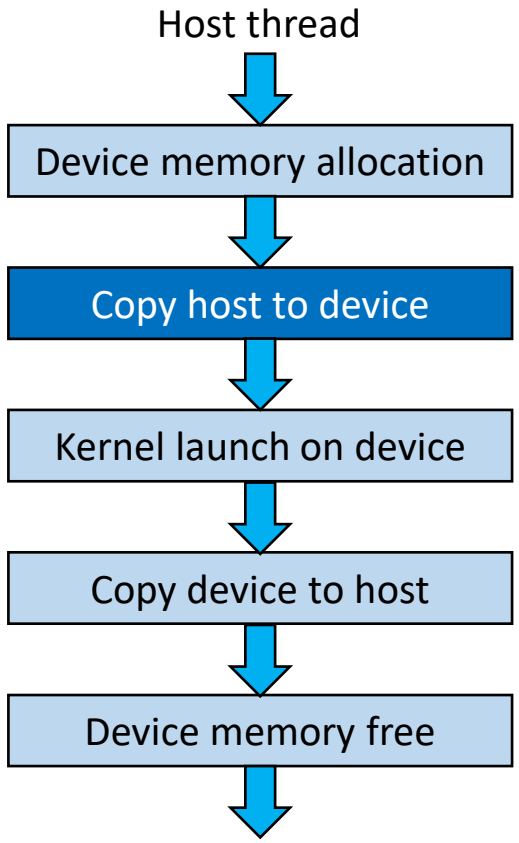


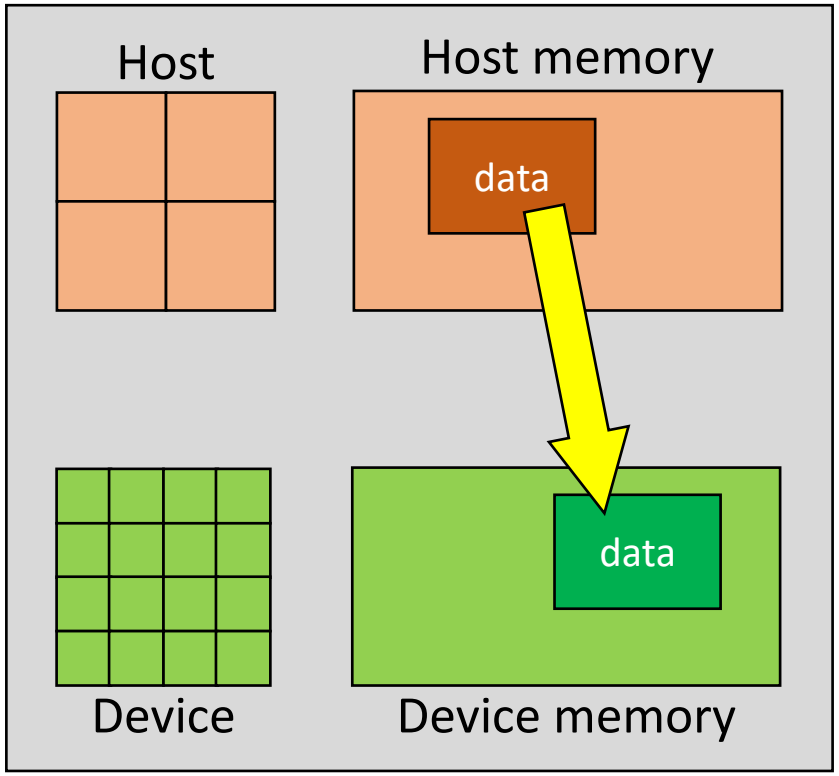
Structure of a basic CUDA program



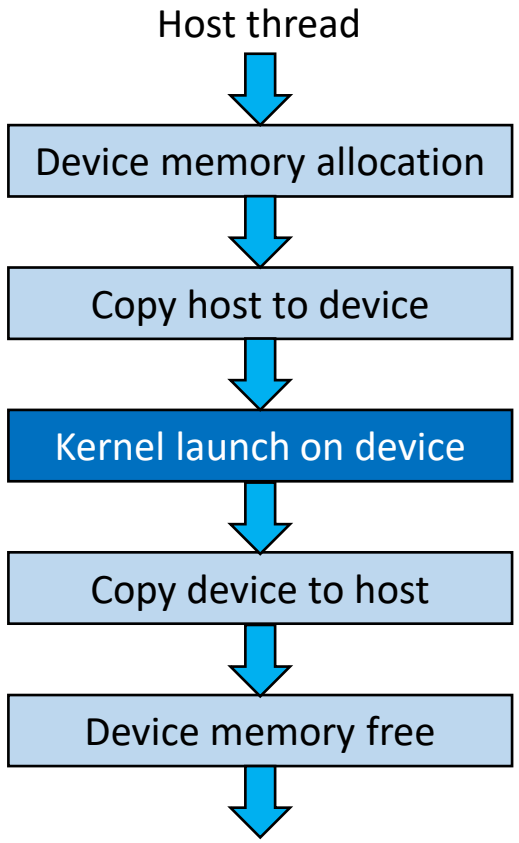


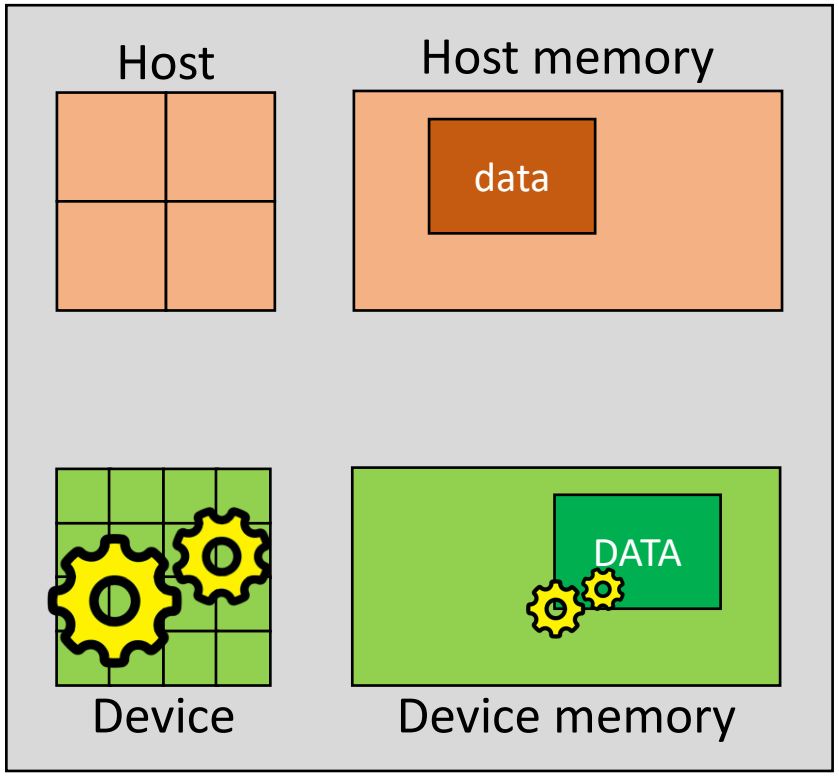
Structure of a basic CUDA program



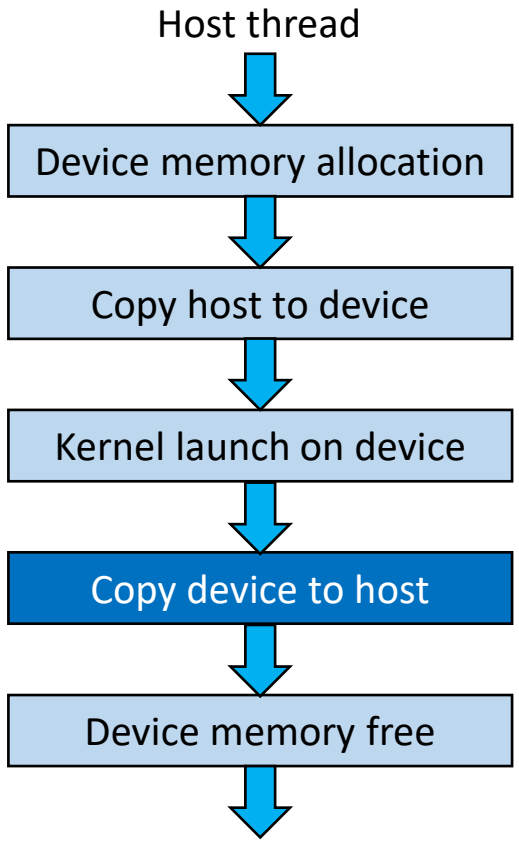


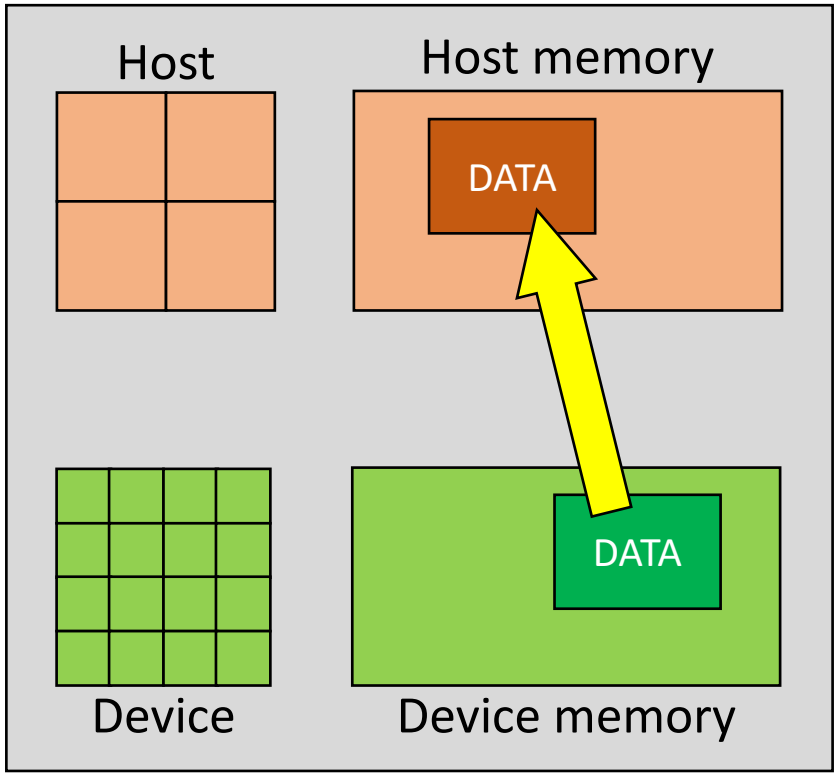
Structure of a basic CUDA program



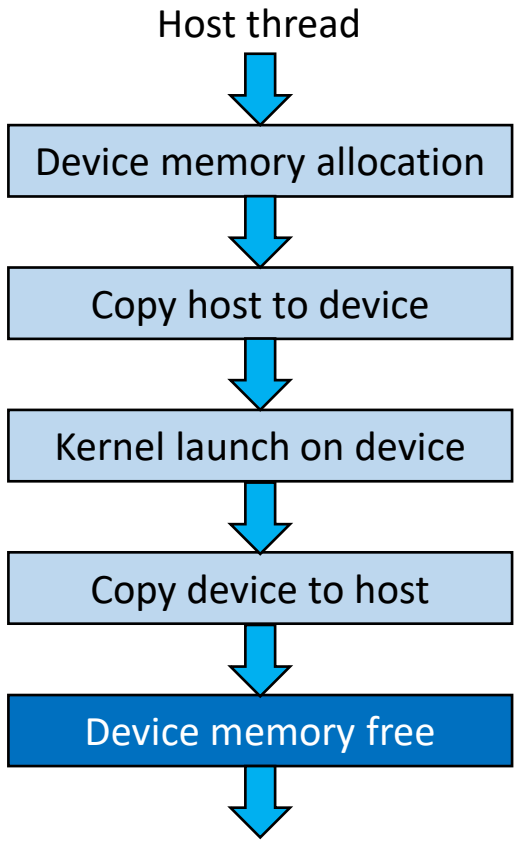


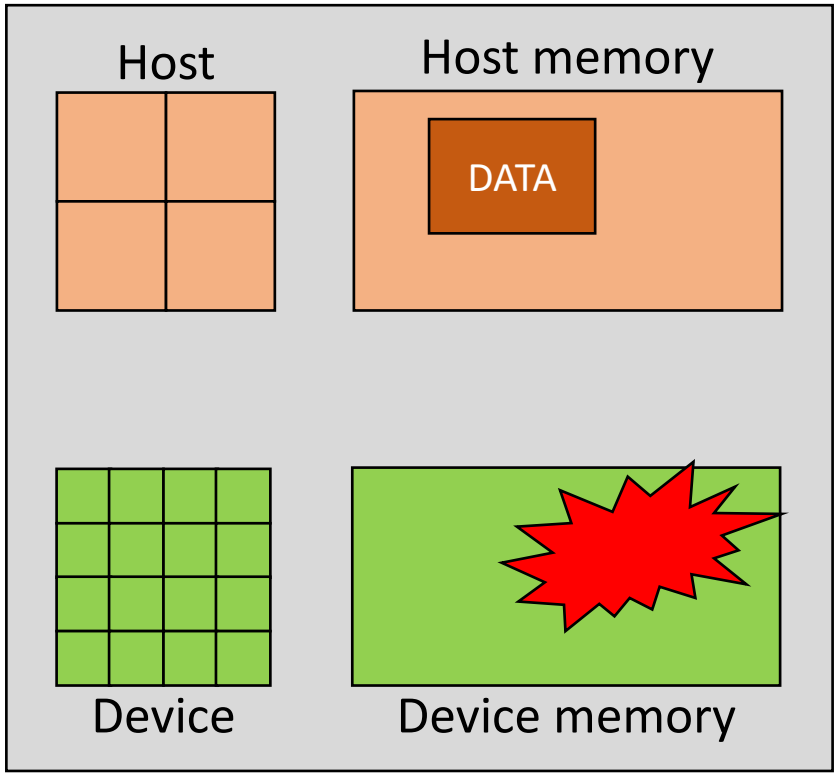
Structure of a basic CUDA program



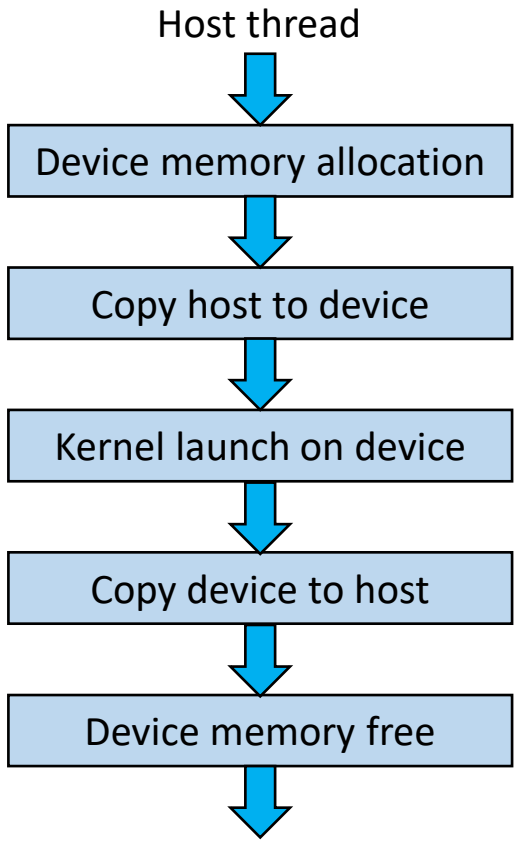


Structure of a basic CUDA program



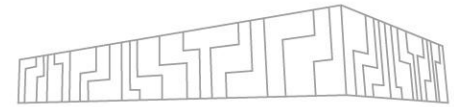


Structure of a basic CUDA program




```
cudaMalloc()

cudaMemcpy()

my_kernel<<< … >>>()

cudaMemcpy()

cudaFree()
```

Quick sidenote



- Multi-GPU system
- Which GPU is being used?

- Some default one.

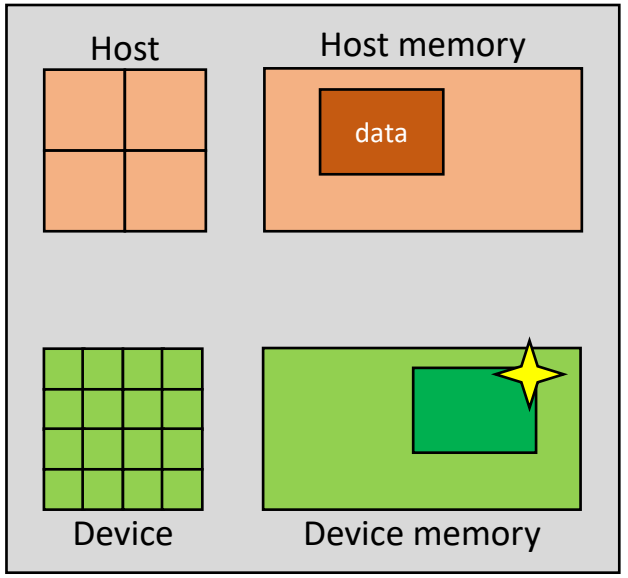
- It is possible to set which GPU we want to work with
  - cudaSetDevice()
  - CUDA_VISIBLE_DEVICES environment variable
- Multi-GPU programming
  - Iterate over all GPUs in a loop, submit work to all of them
  - Or OpenMP thread per GPU, MPI rank per GPU, …

cudaMalloc, cudaFree

- Allocates/deallocates memory in the memory space of the GPU device

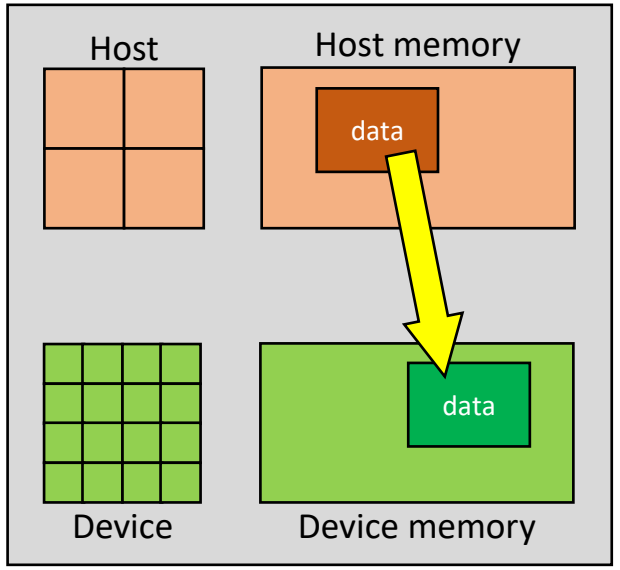
```
cudaError_t cudaMalloc(void ** ptr, size_t num_bytes);
cudaError_t cudaFree(void * ptr);
```

```
...
int count = 2024;
double * d_array;
cudaMalloc(&d_array, count * sizeof(double));
...
cudaFree(d_array);
...
```



cudaMemcpy

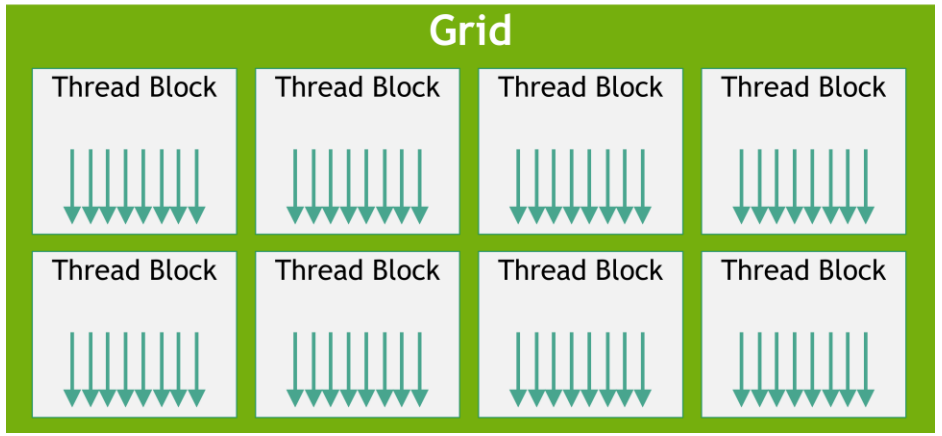
- Copy data between host and device
  - Or host-host, or device-device



- `cudaError_t cudaMemcpy(void * dst, const void * src, size_t num_bytes, cudaMemcpyKind kind);`
- kind ∈ {cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDefault, …}

```
...
int count = 2024;
double * h_array;
double * d_array;
...
cudaMemcpy(d_array, h_array, count * sizeof(double), cudaMemcpyHostToDevice);
...
cudaMemcpy(h_array, d_array, count * sizeof(double), cudaMemcpyDeviceToHost);
...
```

Kernel

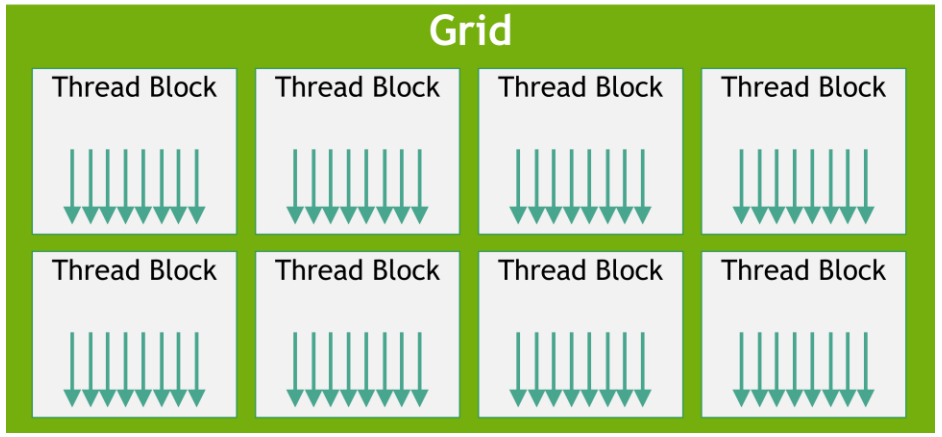


- Special function that runs on the device
- Hierarchy of blocks and threads
- Each thread executes the kernel exactly once
- Threads are grouped into threadblocks (blocks)
- Each block and thread receives a unique set of IDs
  - Built-in variables available in the kernel
  - blockIdx.x – index of block in the grid
  - threadIdx.x – index if thread within the block
  - blockDim.x – dimension (size) of block, number of threads in block
  - gridDim.x – number of blocks in the grid
- Similar to OpenMP, but CUDA has two layers of parallelism

```
__global__ void my_kernel()
{
    ...
}
```

Q: Why .x ?
A: This is 1-dimensional kernel. 2D and 3D kernels also exist, where .y and .z are also used

Kernel

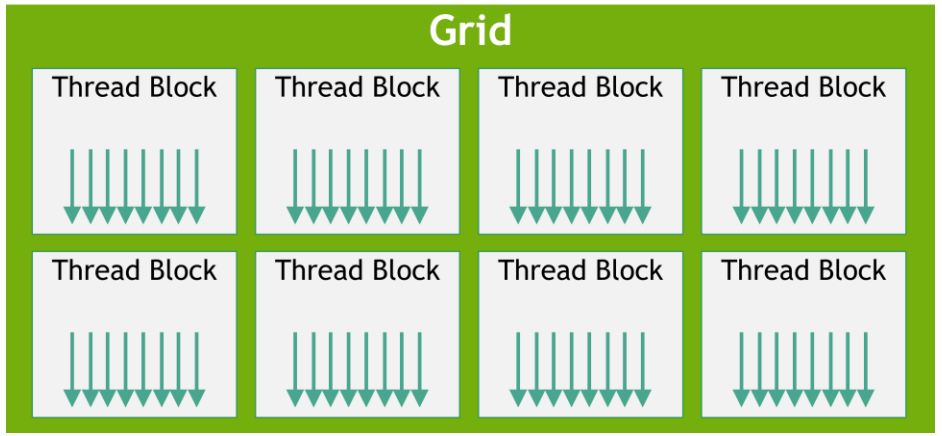


- Function in global scope returning void
- Annotated with __global__ keyword
  - Two underscores on each side
- Launched using <<< >>> syntax
  - Submitting the GPU kernel from CPU code
  - Number of blocks in grid and threads per block needs to be provided
    - `threads_per_block` is usually set arbitrarily (e.g., 256, max 1024)
    - `blocks_in_grid` is calculated to fit the data (max 2^31)
  - `my_kernel<<< blocks_in_grid, threads_per_block >>>(params…)`
- Kernel is launched asynchronously
  - Submitted to the GPU for later execution
  - Need to make sure the kernel (and all operations on the device) finished
    - `cudaDeviceSynchronize()`
    - `cudaMemcpy()` has implicit synchronization inside

```
__global__ void my_kernel()
{
    ...
}
```

```
my_kernel<<< 2, 4 >>>();
```

Kernel



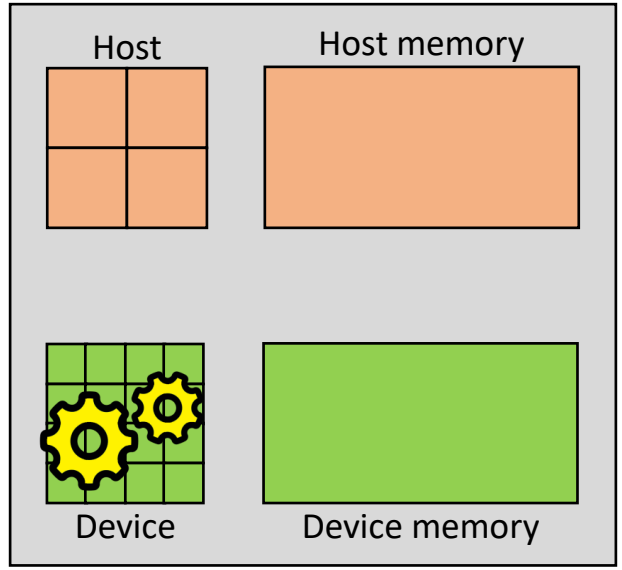
say_hello kernel

```
__global__ void say_hello()
{
    printf("Hello from thread %d/%d, block %d/%d\n",
        threadIdx.x, blockDim.x,
        blockIdx.x, gridDim.x);
}
```

Launched with 2 blocks, each block has 4 threads

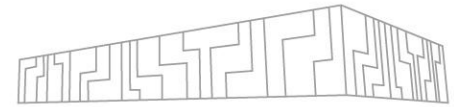
```
say_hello<<< 2, 4 >>>();
cudaDeviceSynchronize();
```

```
Hello from thread 0/4, block 1/2
Hello from thread 1/4, block 1/2
Hello from thread 2/4, block 1/2
Hello from thread 3/4, block 1/2
Hello from thread 0/4, block 0/2
Hello from thread 1/4, block 0/2
Hello from thread 2/4, block 0/2
Hello from thread 3/4, block 0/2
```



Note:
std::cout does not work inside kernel

Compilation



- CUDA C/C++ code belongs in `*.cu` files

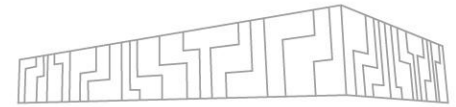
- CUDA is not standard C/C++, need special compiler
- Compiler for CUDA applications – nvcc
  - `nvcc source.cu -o program`

- Run just like a normal program
  - `./hello`

- CMake includes CUDA as a language
  - As a package it is also available

Hands-on on Karolina

Access Karolina GPU nodes



- 8 GPUs and 128 CPU cores per node, 72 nodes
- Possible to allocate only 1 GPU and 16 cores = 1/8 of the node

-A, --account
-p, --partition
-N, --nodes
-t, --time
-G, --gpus

```
salloc -A DD-23-116 -p qgpu --gpus 1  --nodes 1 --time 2:00:00
```
- Request **1** GPU on **1** node for 2 hours

```
salloc -A DD-23-116 -p qgpu
```
- Default: **1** GPU, **1** node, 24h time limit

```
salloc -A DD-23-116 -p qgpu --gpus 4           --time 2:00:00
```
- Request **4** GPUs for 2 hours. You might get the GPUs scattered across **1-4** nodes

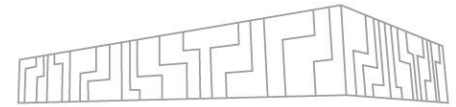
```
salloc -A DD-23-116 -p qgpu --gpus 4  --nodes 1 --time 2:00:00
```
- Request **4** GPUs on **1** node for 2 hours

```
salloc -A DD-23-116 -p qgpu --gpus 16 --nodes 2 --time 2:00:00
```
- Request **16** GPUs on **2** nodes for 2 hours. You will get 2 full nodes.

- No way to enforce to get 4 “neighboring” GPUs on the node
- qgpu_exp – higher priority, but max 8 GPUs for 1 hour
- salloc -> sbatch … ./job.sh to submit batch jobs

https://docs.it4i.cz/general/karolina-slurm/#using-gpu-queues

Access Karolina GPU node



- Connect to Karolina (ssh, VS Code, …)

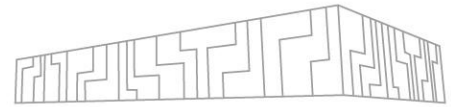
- We have a reservation prepared

```
salloc --account=DD-23-116 --reservation=dd-23-116_2024-06-05T09:00:00_2024-06-05T12:30:00_5_qgpu --gpus 1
```

- Load the CUDA module

```
module load CUDA
```

Hands on – Hello world

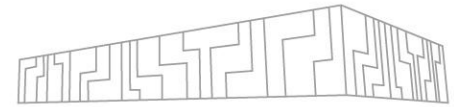


- Write a Hello world program similar to the one already shown, compile and run it
- Start with `hello_world.task.cu`
- Additionally, compute and print in each thread:
  - Global index of the thread in the whole grid
  - Total number of threads

`blockIdx.x`	0	0	0	0	1	1	1	1
`threadIdx.x`	0	1	2	3	0	1	2	3
`global index`	0	1	2	3	4	5	6	7

```
Hello from thread 0/4, block 1/2, my global index is 4/8
Hello from thread 1/4, block 1/2, my global index is 5/8
Hello from thread 2/4, block 1/2, my global index is 6/8
Hello from thread 3/4, block 1/2, my global index is 7/8
Hello from thread 0/4, block 0/2, my global index is 0/8
Hello from thread 1/4, block 0/2, my global index is 1/8
Hello from thread 2/4, block 0/2, my global index is 2/8
Hello from thread 3/4, block 0/2, my global index is 3/8
```

Vector scale example

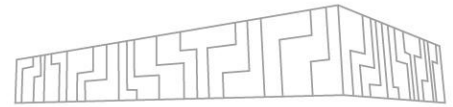


- Scale a vector (array) of floats on the GPU using CUDA
  - X := scalar*X

- The array starts and ends on host
- Perform the scaling on device

- Aside from running the kernel, we need to copy the data

Hands on – Vector add



- Write a CUDA program that adds two vectors into a third vector
  - C = A + B
- Start from the `vector_add.task.cu` file
- Complete the TODOs
- Feel free to get inspiration from the Vector scale

Other notable GPU programming models

HIP

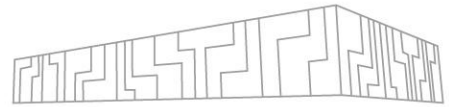
- Created by AMD to mimic CUDA
  - To ease users' transition from NVIDIA to AMD GPUs
- Works on both AMD and NVIDIA GPUs
- cuda* functions and types replaced by hip*
- hip* libraries (BLAS etc.)
  - Wrappers around cuda* or roc* functions
- Hipify – convert CUDA source code to HIP code

- ROCm software ecosystem/platform
- roc* libraries (blas, sparse, fft, …)

- Frontier (#1) and LUMI (#5) use AMD GPUs

AMD ROCm

HIP



source.cu

```cuda
__global__ void vector_scale(float * x, float alpha, int count)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if(idx < count) x[idx] = alpha * x[idx];
}

int main()
{
    int count = 20 * 256;

    float * h_data = new float[count];
    for(int i = 0; i < count; i++) h_data[i] = i;

    float * d_data;
    cudaMalloc(&d_data, count * sizeof(float));

    cudaMemcpy(d_data, h_data, count * sizeof(float), cudaMemcpyHostToDevice);
    vector_scale<<< 20, 256 >>>(d_data, 10, count);
    cudaMemcpy(h_data, d_data, count * sizeof(float), cudaMemcpyDeviceToHost);

    cudaFree(d_data);
    delete[] h_data;
    return 0;
}
```

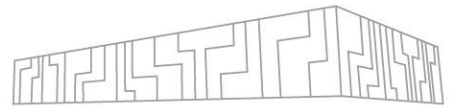
```
$ nvcc source.cu -o program_cuda.x
```

source.hip.cpp

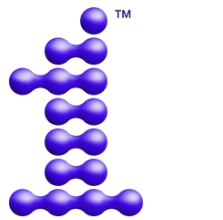
```hip
#include <hip/hip_runtime.h>

__global__ void vector_scale(float * x, float alpha, int count)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if(idx < count) x[idx] = alpha * x[idx];
}

int main()
{
    int count = 20 * 256;

    float * h_data = new float[count];
    for(int i = 0; i < count; i++) h_data[i] = i;

    float * d_data;
    hipMalloc(&d_data, count * sizeof(float));

    hipMemcpy(d_data, h_data, count * sizeof(float), hipMemcpyHostToDevice);
    vector_scale<<< 20, 256 >>>(d_data, 10, count);
    hipMemcpy(h_data, d_data, count * sizeof(float), hipMemcpyDeviceToHost);

    hipFree(d_data);
    delete[] h_data;
    return 0;
}
```

```
$ hipcc source.hip.cpp -o program_hip.x
```

SYCL



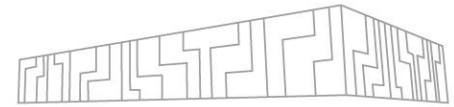
- Open standard, modern C++17 interface
- A way to do parallel programming not only for GPUs
  - CPUs, FPGAs
- Primary way to utilize Intel GPUs
  - Aurora supercomputer (#2)
- Source code portability. Not necessarily performance portability.
- Implementations for all of Intel, AMD and NVIDIA GPUs exist
  - DPC++ (Intel), AdaptiveCPP
- oneAPI – SYCL interface for high performance libraries (BLAS, SPARSE, FFT, …)
  - Also a standard
  - Has implementations for all of Intel, AMD and NVIDIA GPUs
    - Intel's oneAPI, Codeplay



oneAPI

intel

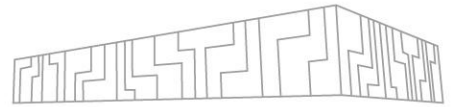
SYCL



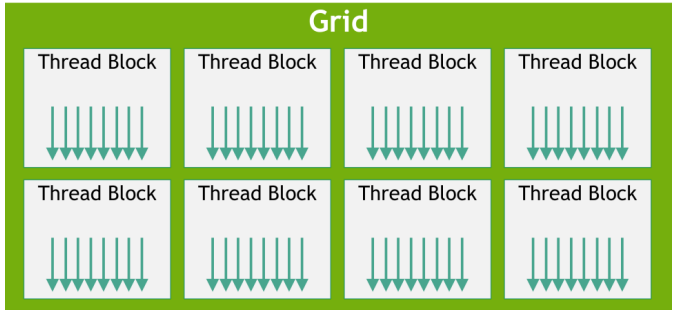
```
sycl::queue q(sycl::gpu_selector_v, {sycl::property::queue::in_order()});

float * d_vector = sycl::malloc_device<float>(count, q);

q.copy<float>(h_vector, d_vector, count);

q.parallel_for(
    sycl::nd_range<1>(sycl::range<1>(count), sycl::range<1>(256)),
    [d_vector,scalar](sycl::nd_item<1> item){
        d_vector[item.get_global_id()] *= scalar;
    }
);

q.copy<float>(d_vector, h_vector, count);

q.wait();
sycl::free(d_vector, q);
```

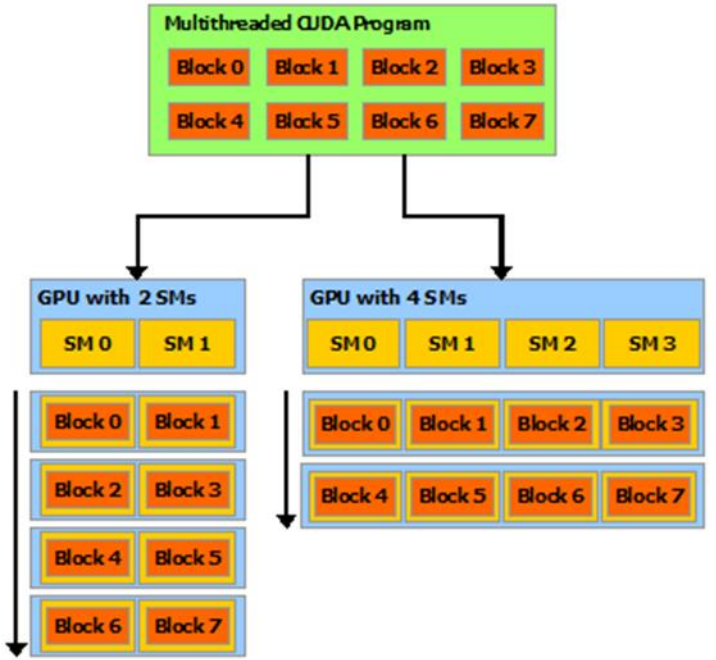
Other useful info

CUDA kernel execution



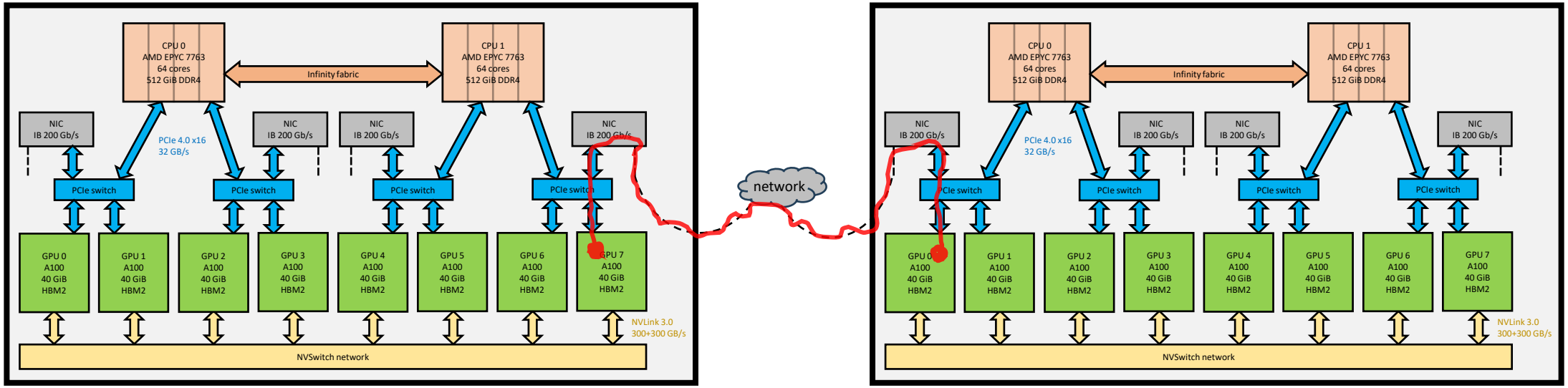
- Each block is executed on a single Streaming Multiprocessor (SM)
  - Independently of other blocks
  - Multiple blocks can be on a single SM if resources allow
- Threads in a block are executed in a SIMD fashion – SIMT
  - Single Instruction Multiple Threads
- Threads are grouped into warps (32 threads)
- Warp is a unit of scheduling in the SM
  - All warps of the block are running on a single SM
- If some warp cannot continue, other warp is scheduled
  - Fast context switching
- Some warps are computing, while other wait for data
  - Latency hiding
- If only some threads in warp branch => control divergence
  - Both paths are taken, threads are masked out



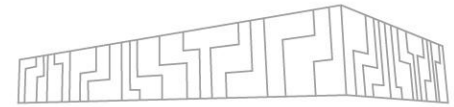


GPU-aware MPI

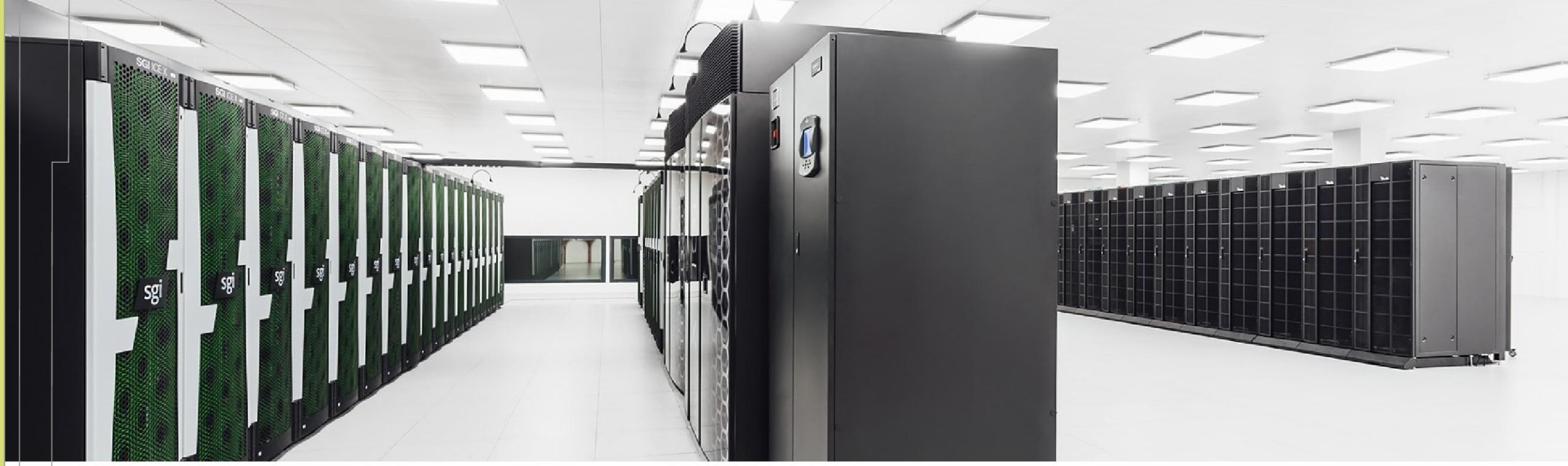
- Copy between nodes directly GPU to GPU
- No need to route through CPU memory
- Just use pointers to GPU memory in MPI functions
- Make sure the MPI you use is correctly compiled
  - Use e.g. the OpenMPI/4.1.6-NVHPC-24.1-CUDA-12.4.0 module



More information, resources



- More CUDA exercises
  - https://code.it4i.cz/training/cuda_examples
- CUDA programming guide
  - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
- CUDA toolkit documentation
  - https://docs.nvidia.com/cuda/
- HIP
  - https://github.com/ROCm/HIP
- SYCL
  - https://www.khronos.org/api/index_2017/sycl
- IT4I documentation
  - https://docs.it4i.cz



Jakub Homola
jakub.homola@vsb.cz

IT4Innovations National Supercomputing Center
VSB – Technical University of Ostrava
Studentská 6231/1B
708 00 Ostrava-Poruba, Czech Republic
www.it4i.cz

VSB TECHNICAL UNIVERSITY OF OSTRAVA | IT4INNOVATIONS NATIONAL SUPERCOMPUTING CENTER