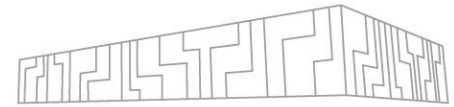
VSB TECHNICAL UNIVERSITY OF OSTRAVA | IT4INNOVATIONS NATIONAL SUPERCOMPUTING CENTER

Leveraging accelerated hardware

GPUs

Jakub Homola, Radim Vavřík

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

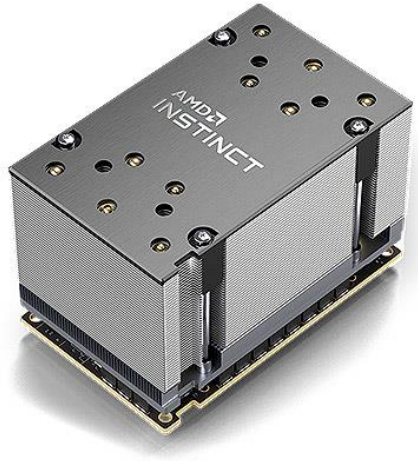
What is a GPU?

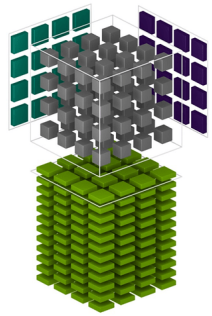
Why should I use it?

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)

AMD

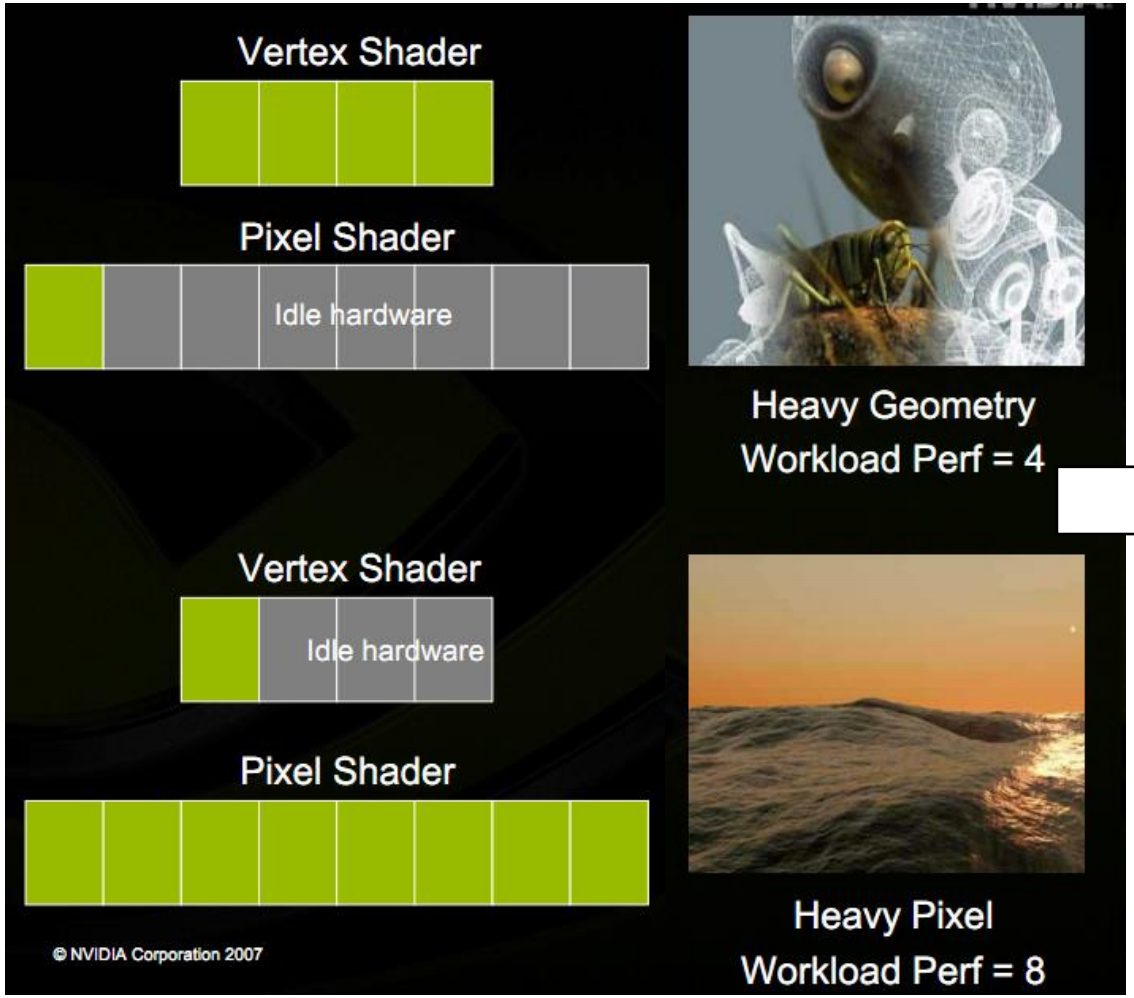


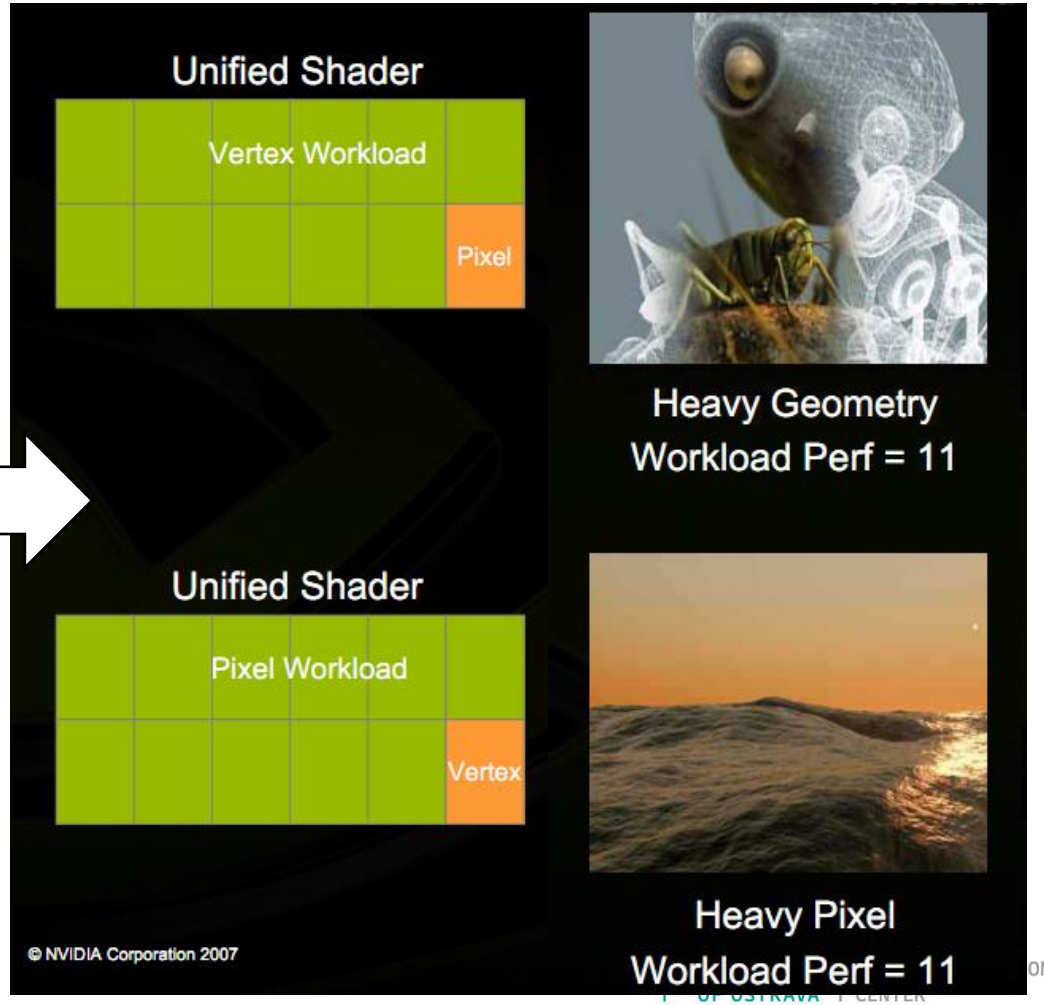


intel

How GPGPUs came to be

- Generalizing/unifying specialized hardware



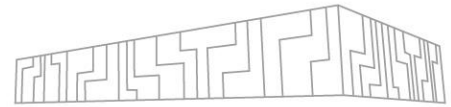


GPUs in today's supercomputers

- TOP500, GREEN500 lists (November 2025)
  - Ranking supercomputers
  - FLOP/s performance, FLOP/s/W energy efficiency
- 51 % of the TOP500 supercomputers contain GPUs
  - TOP500: 88 out of top 100, 9 out of top 10
  - GREEN500: 100 out of top 100
- GPUs are the way to go for massive performance and energy efficiency
- All vendors are represented
  - NVIDIA
  - AMD
  - Intel

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	DOE/NNSA/LLNL United States	**El Capitan** - HPE Cray EX255a, AMD 4th Gen EPYC 24C 1.8GHz, AMD Instinct MI300A, Slingshot-11, TOSS HPE	11,039,616	1,742.00	2,746.38	29,581
2	DOE/SC/Oak Ridge National Laboratory United States	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE Cray OS HPE	9,066,176	1,353.00	2,055.72	24,607
3	DOE/SC/Argonne National Laboratory United States	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11 Intel	9,264,128	1,012.00	1,980.01	38,698
4	EuroHPC/FZJ Germany	**JUPITER Booster** - BullSequana XH3000, GH Superchip 72C 3GHz, NVIDIA GH200 Superchip, Quad-Rail NVIDIA InfiniBand NDR200, RedHat Enterprise Linux EVIDEN	4,801,344	793.40	930.00	13,088
5	Microsoft Azure United States	**Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR Microsoft Azure	2,073,600	561.20	846.84	
6	Eni S.p.A. Italy	**HPC6** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, RHEL 8.9 HPE	3,143,520	477.90	606.97	8,461
7	RIKEN Center for Computational Science Japan	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D Fujitsu	7,630,848	442.01	537.21	29,899
8	Swiss National Supercomputing Centre (CSCS) Switzerland	**Alps** - HPE Cray EX254n, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, HPE Cray OS HPE	2,121,600	434.90	574.84	7,124
9	EuroHPC/CSC Finland	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11 HPE	2,752,704	379.70	531.51	7,107
10	EuroHPC/CINECA Italy	**Leonardo** - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband EVIDEN	1,824,768	241.20	306.31	7,494

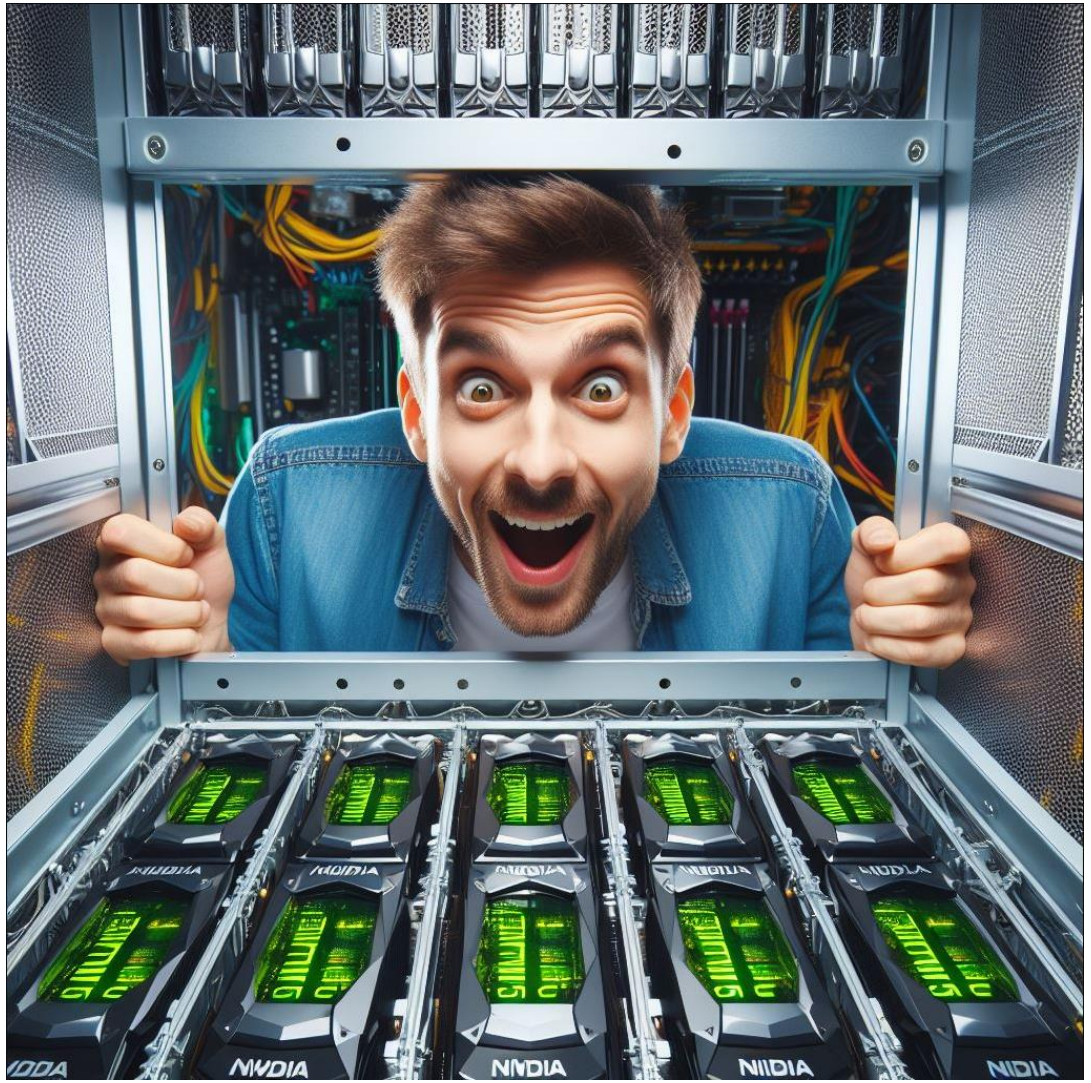
CPU vs GPU



	Device name	Performance FP64 [TFLOP/s]	Memory bandwidth [GB/s]	TDP [W]	Energy efficiency [GFLOP/s/W]
GPU	NVIDIA H200	66.9	4800	700	96
GPU	NVIDIA B200	40.0	8000	1000	40
GPU	AMD MI250X	95.7	3200	560	170
GPU	AMD MI300X	163.4	5300	750	218
GPU	Intel Data Center GPU Max	52.0	3280	600	87
CPU	NVIDIA Grace CPU (ARM)	3.5	512	250	14
CPU	AMD EPYC 9654 (Zen 4)	7.4	460	360	20
CPU	AMD EPYC 9965 (Zen 5c)	13.8	614	500	28
CPU	Intel Xeon Max 9480 (Sapphire Rapids) DDR5	3.4	300	350	10
CPU	Intel Xeon Max 9480 (Sapphire Rapids) HBM	3.4	1640	350	10
CPU	Intel Xeon 6980P (Granite Rapids)	13	845	500	26
CPU	Fujitsu A64FX (ARM)	3.1	1024	140	22

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

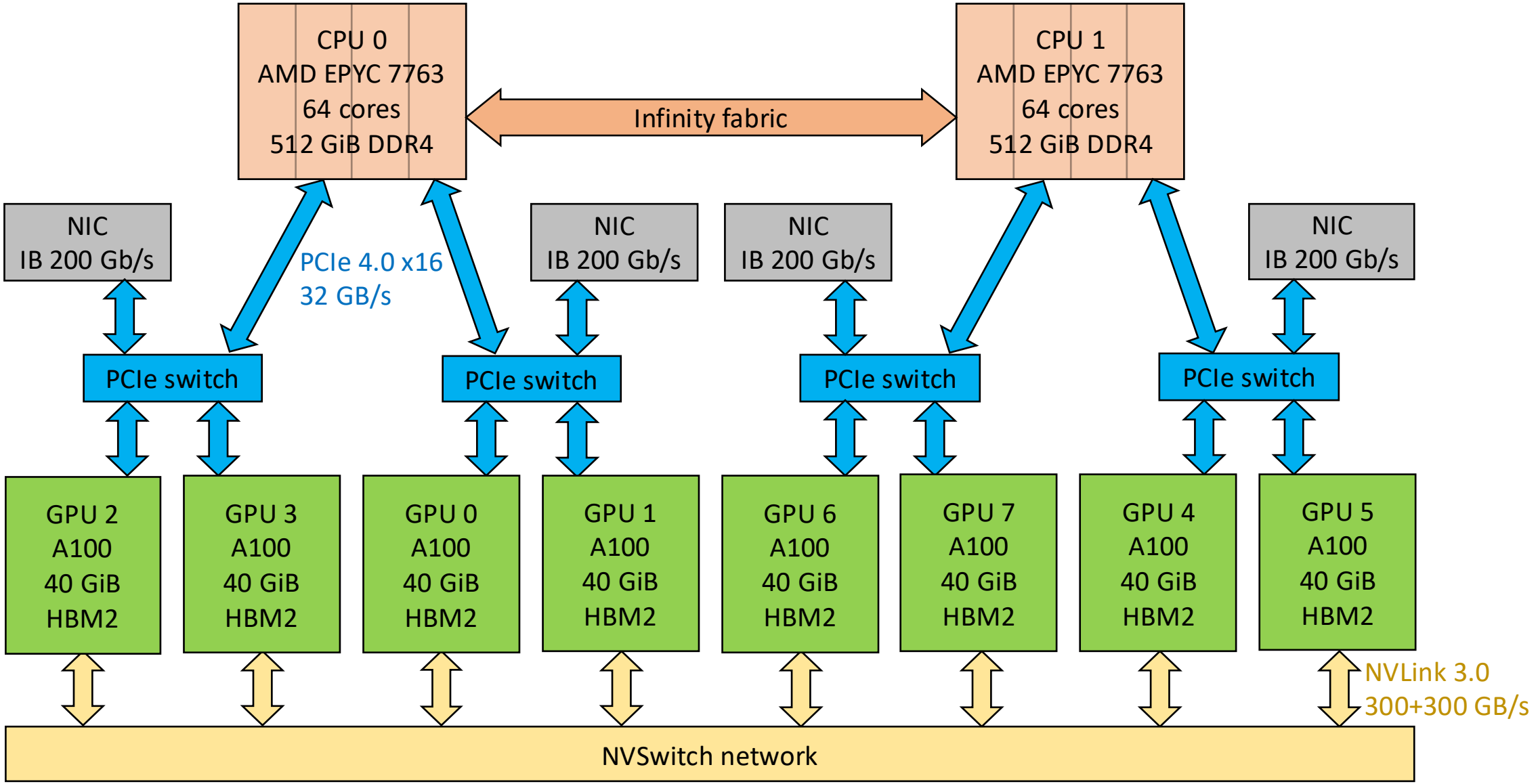
The problem with GPUs

- Of course, there's a catch
- GPU is not a do-it-all device
- Specialized hardware for specific types of problems

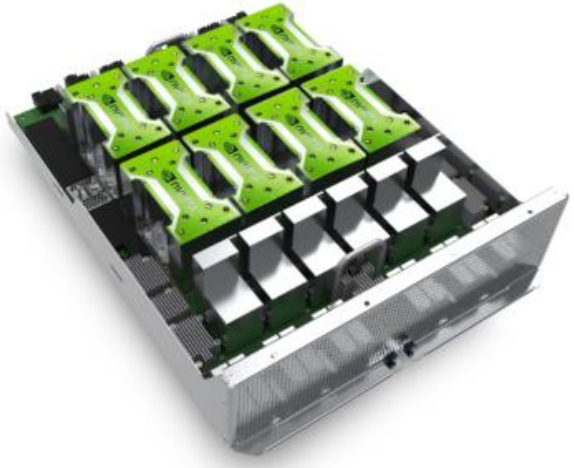
- Located separately from the CPU, as a co-processor
- GPUs are focused on throughput and FLOP/s performance
  - Calculating many things at once, but slower
- Not all algorithms are suitable for GPUs
  - High parallelism is required
- GPUs are inferior in:
  - Latency-sensitive operations
  - Highly branching code
  - Random memory access

Architecture of GPUs and GPU nodes

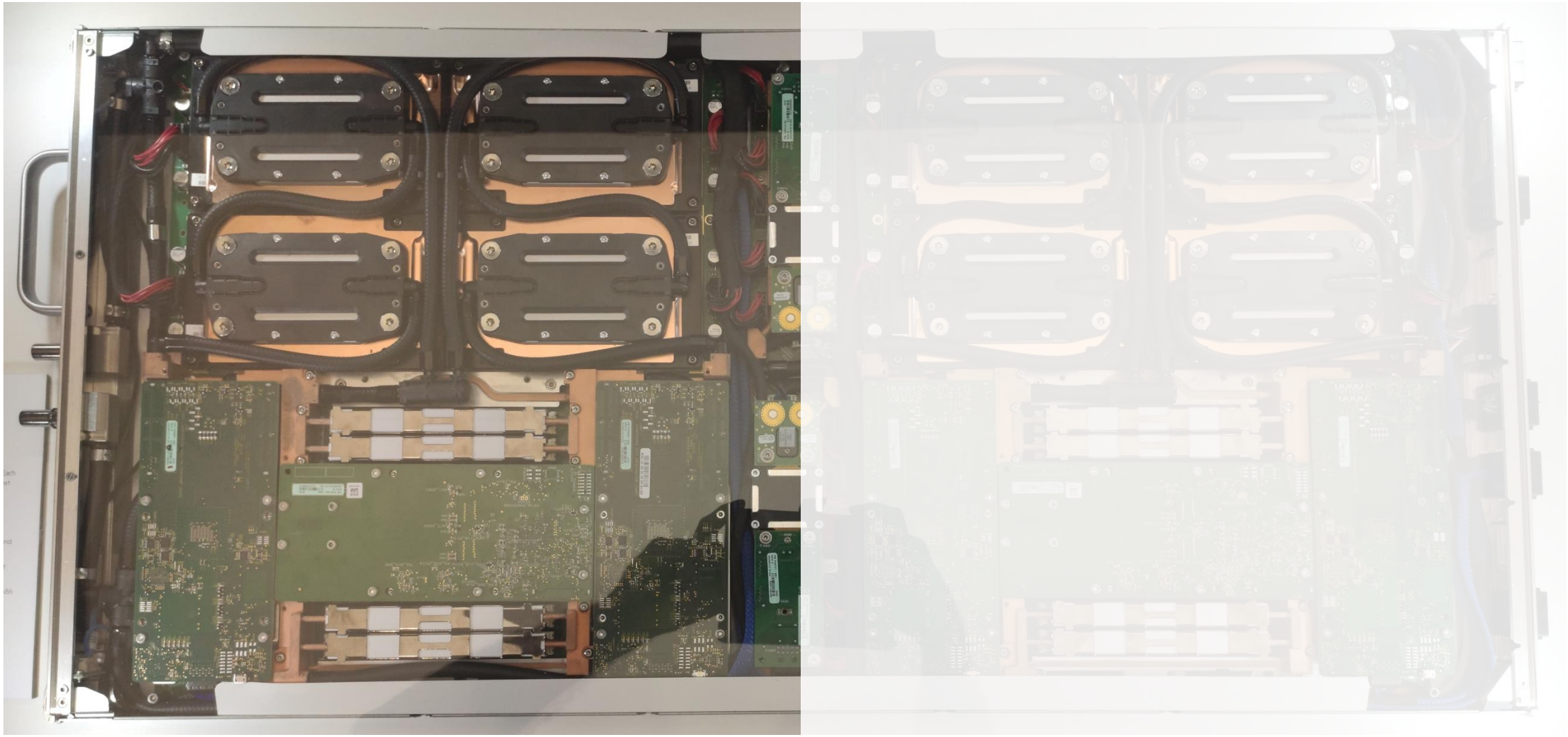
Karolina GPU node architecture



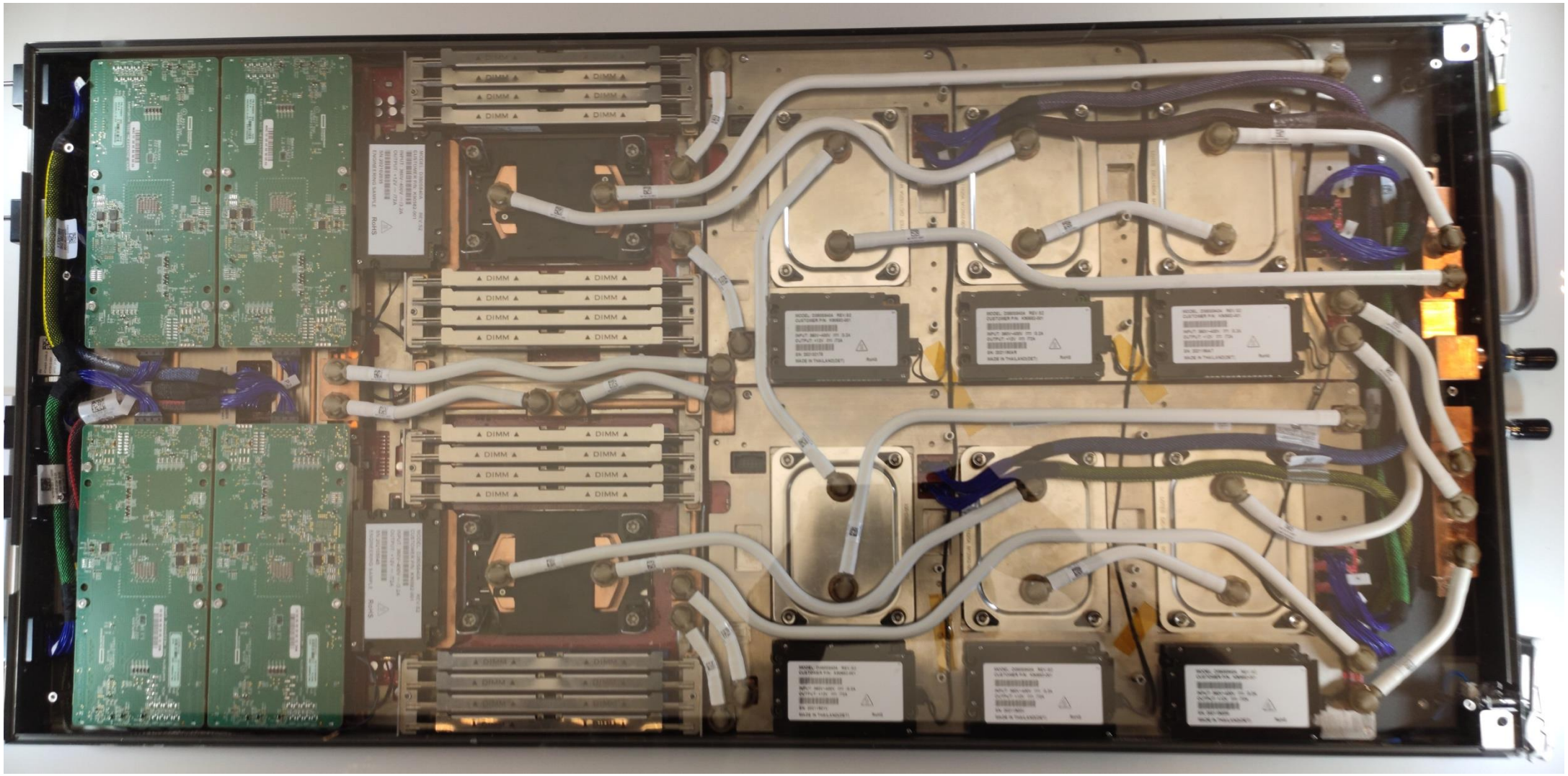
Karolina GPU node (8x NVIDIA A100)



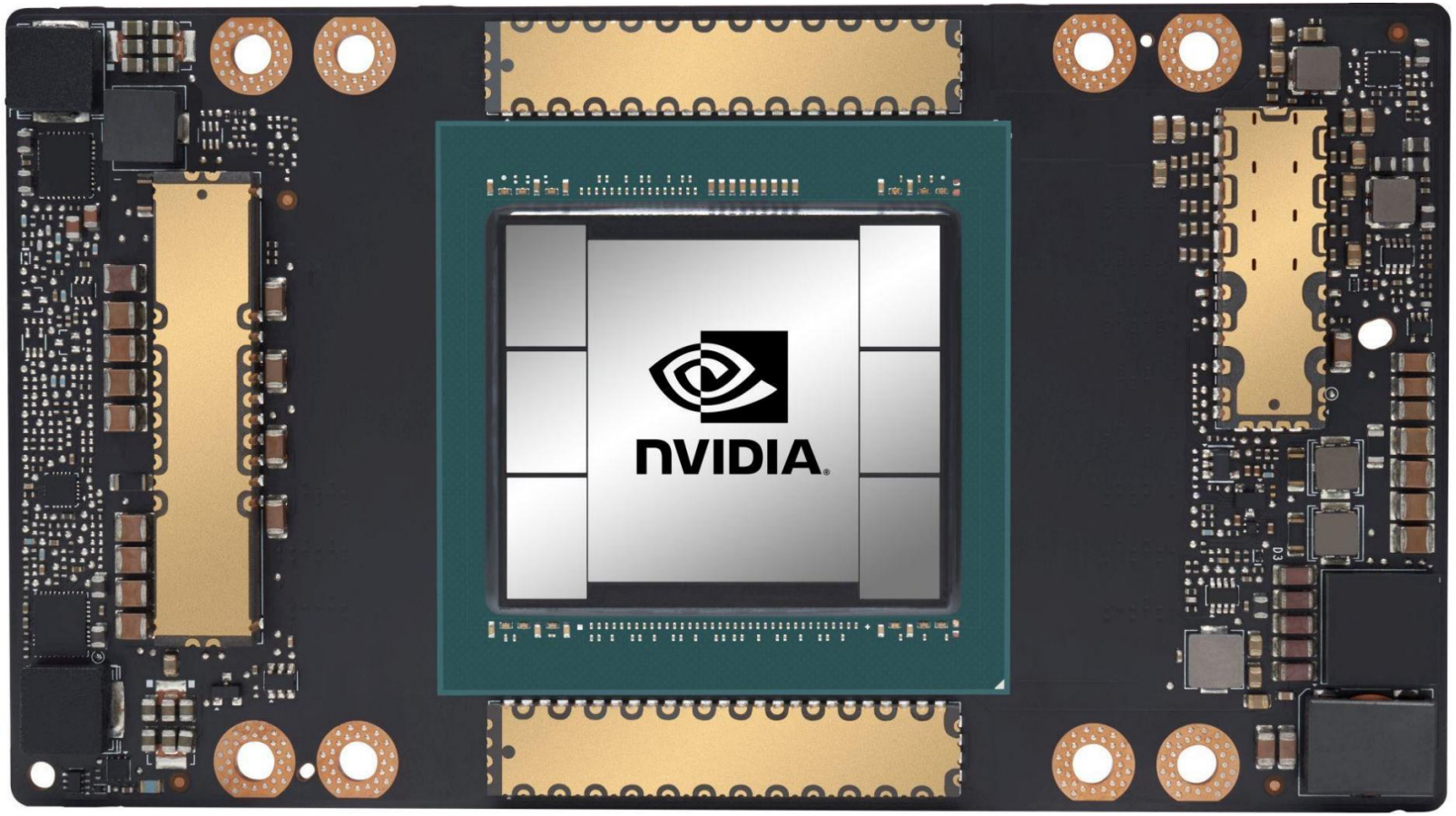
Frontier node (4x AMD MI250X)



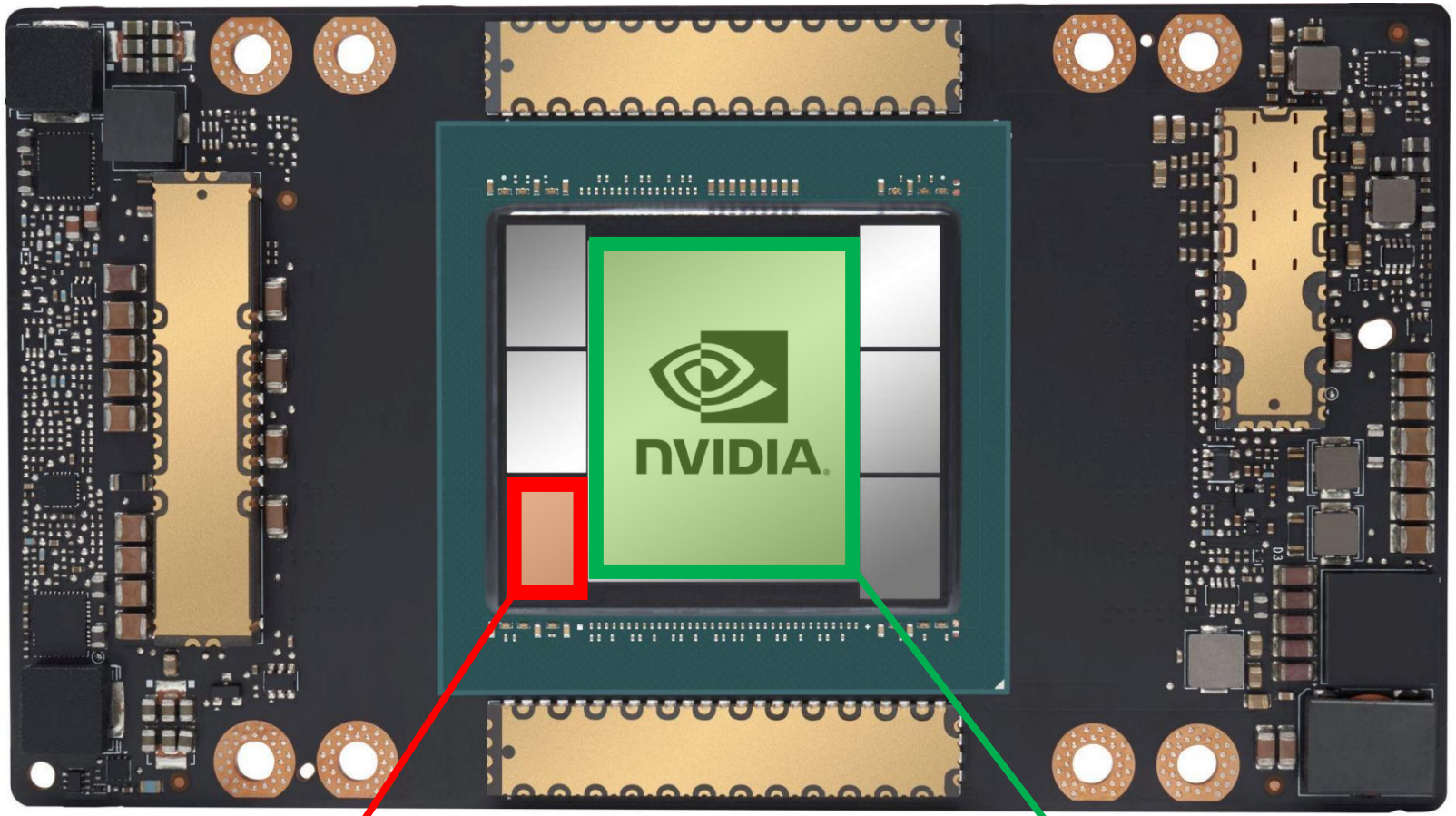
Aurora node (6x Intel Data Center GPU Max)



NVIDIA A100 SXM4 module



NVIDIA A100 SXM4 module

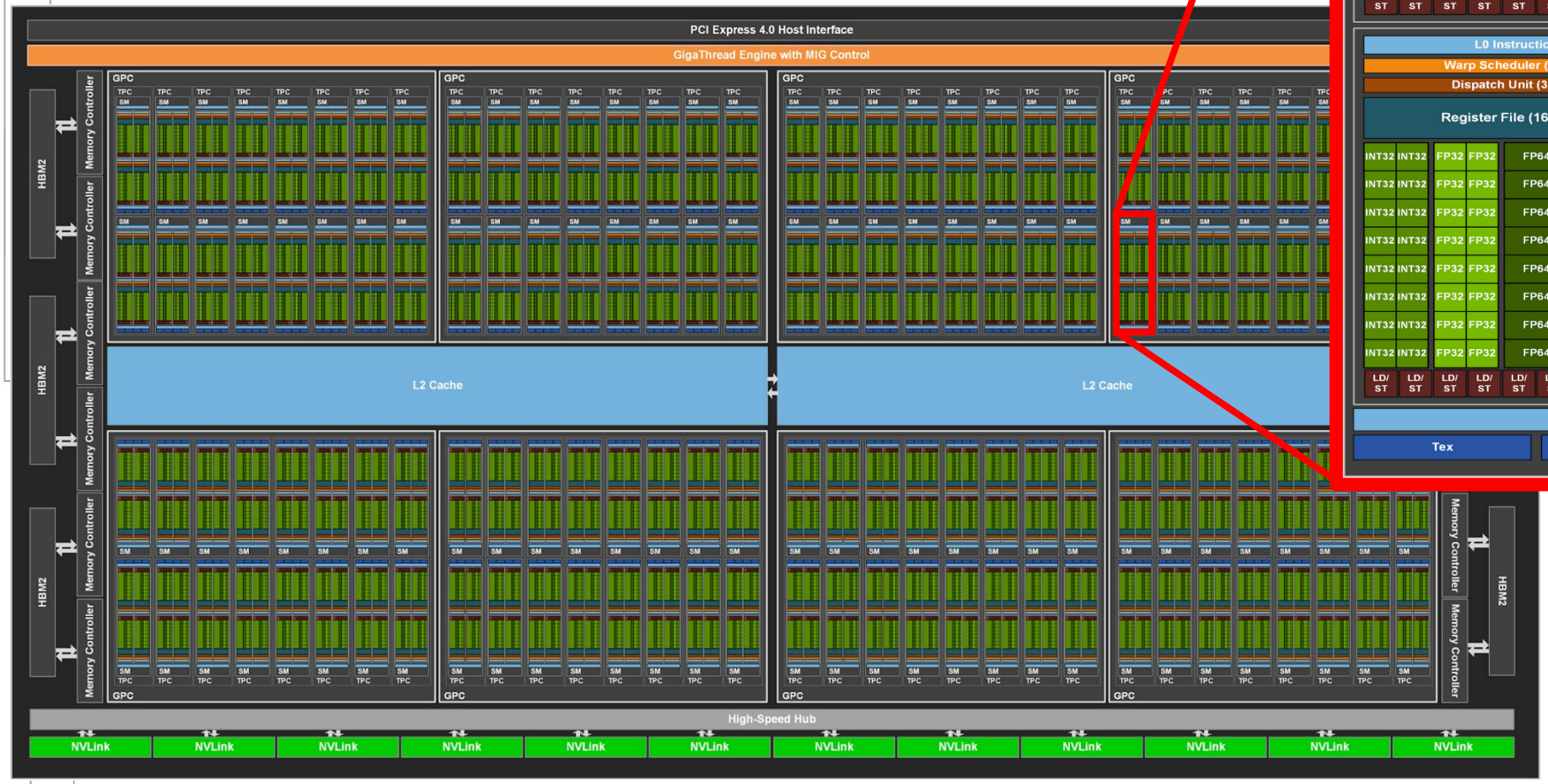


HBM2 memory

GPU die with streaming multiprocessors, caches, memory controller, …

NVIDIA A100 die

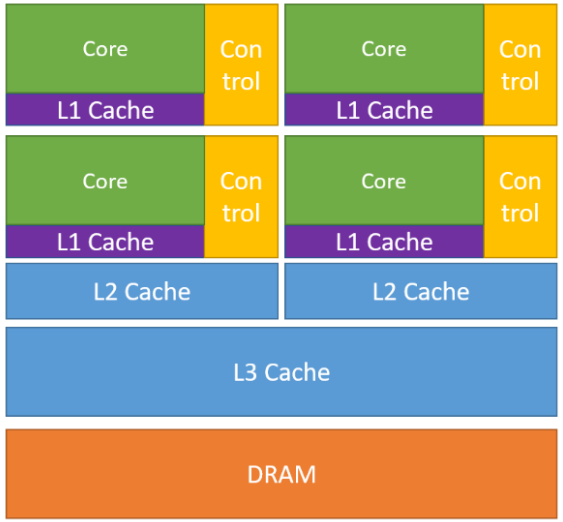
- Nvidia GA100 full GPU architecture
- 128 streaming multiprocessors (SM)
  - Each SM has 32 FP64 units, 64 FP32 units, …



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

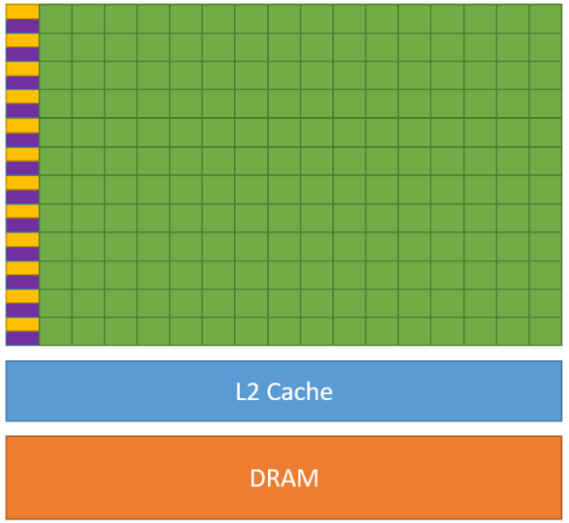


CPUs

- Powerful ALU
- Large caches
- Branch prediction

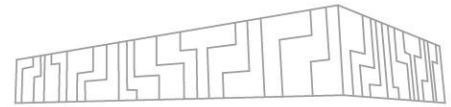
GPUs

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



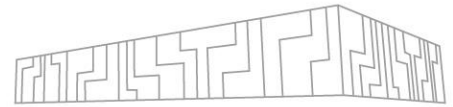
How do I use GPUs?

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

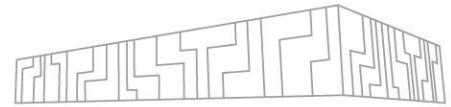
GPU-enabled applications



- 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 => you can try writing your own

Libraries, packages

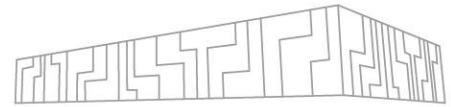


- If you write your own app/script, use libraries/packages that use the GPU internally

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

- Again, consult the documentation

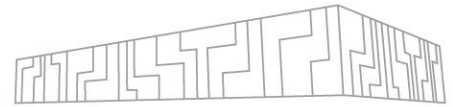
Compiler directives



- Start with your current C/C++/Fortran code for CPU
- Annotate it to offload certain parts to the GPU
- Generic, portable, 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 parallel programming model
  - Modern C++17, only headers and libraries, no language extensions
- OpenCL
  - Older, for all GPUs

VSB TECHNICAL UNIVERSITY OF OSTRAVA | IT4INNOVATIONS NATIONAL SUPERCOMPUTING CENTER

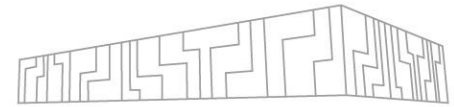
Introduction to CUDA

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++

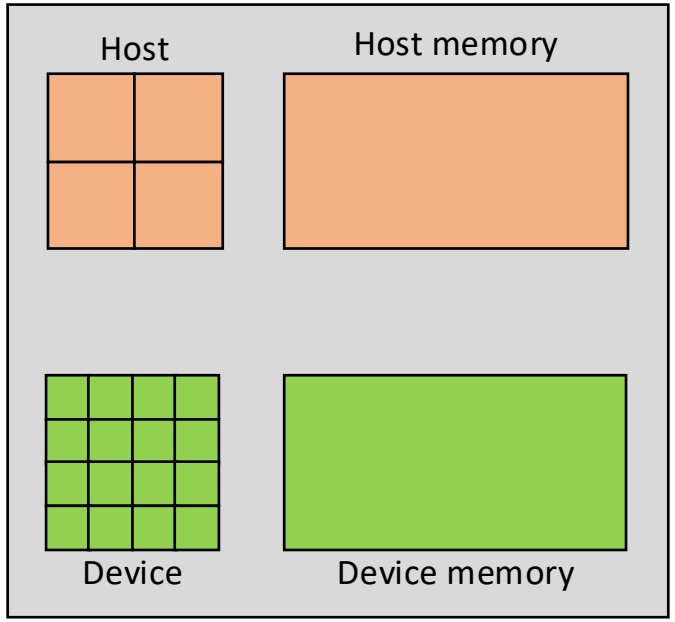
- CUDA toolkit
  - NVCC compiler
  - Drivers
  - Libraries – cuBLAS, cuSPARSE, cuFFT, CUB, NCCL, …
  - Profiler, debugger
  - …

Memory model

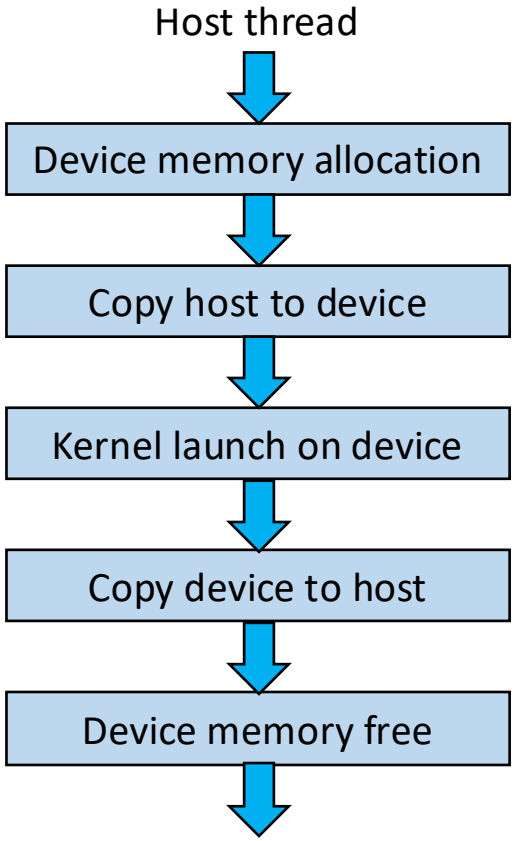


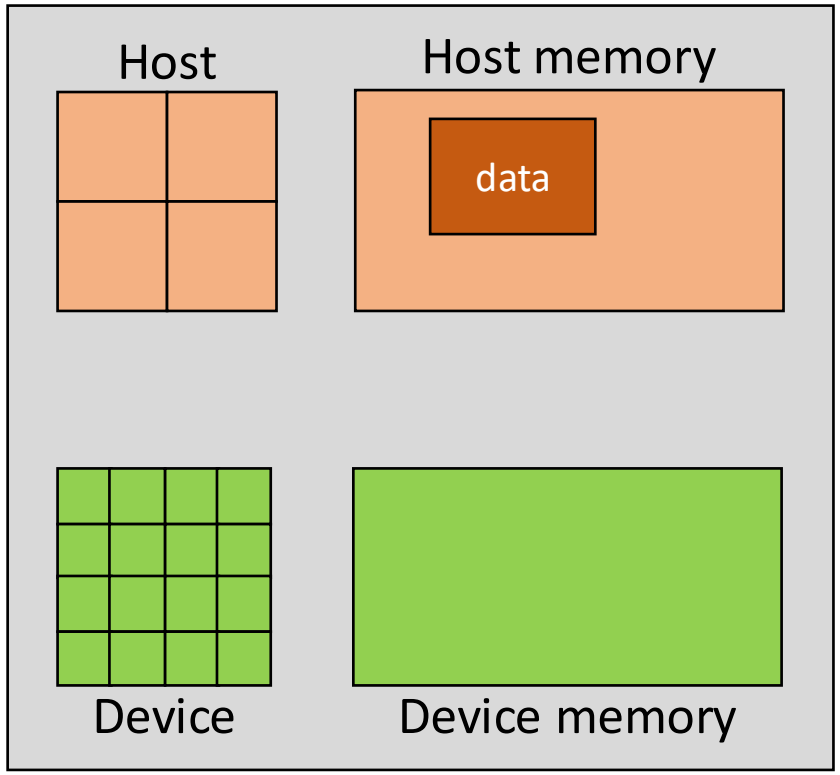
- CPU and GPU have separate memories
- Memory needs to be manually allocated on the GPU
- Data needs to be copied between CPU and GPU

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

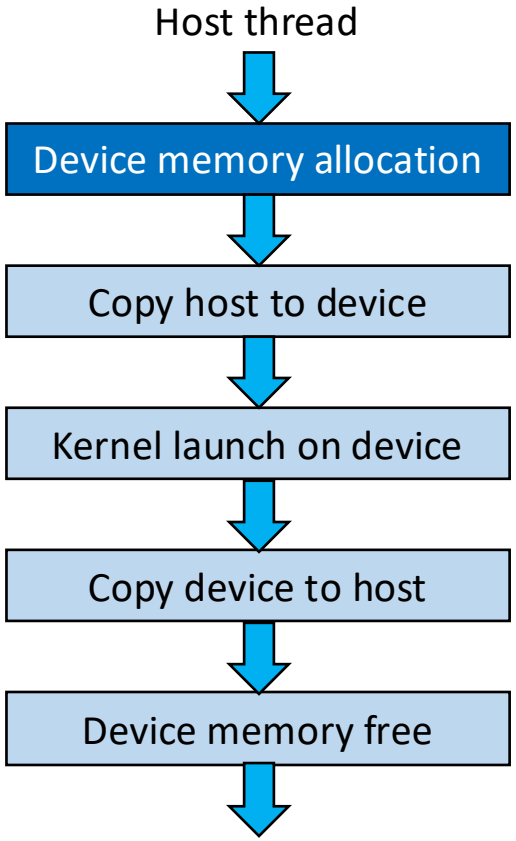


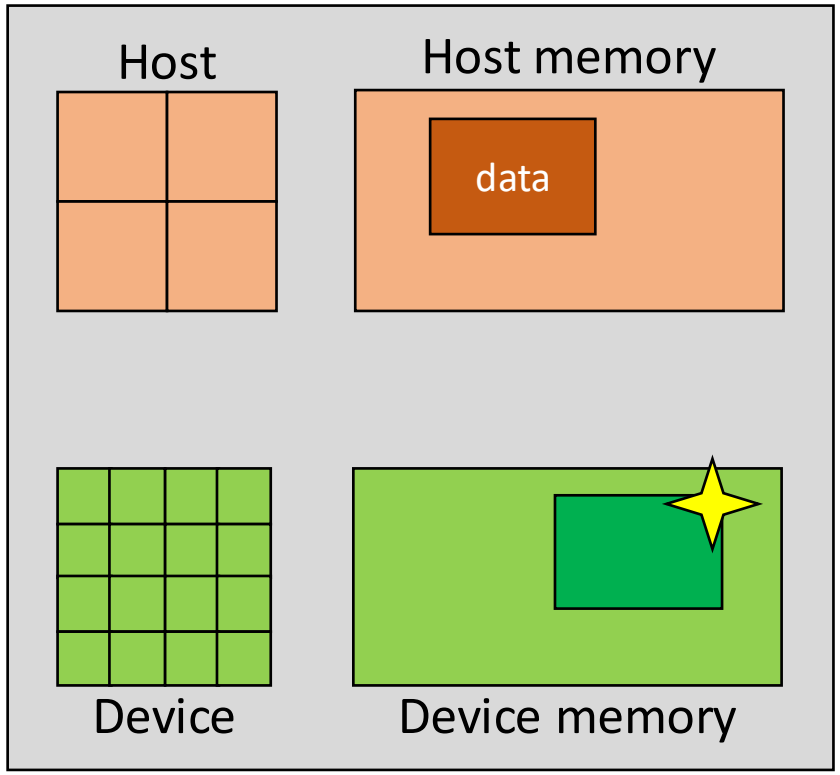
Structure of a basic CUDA program



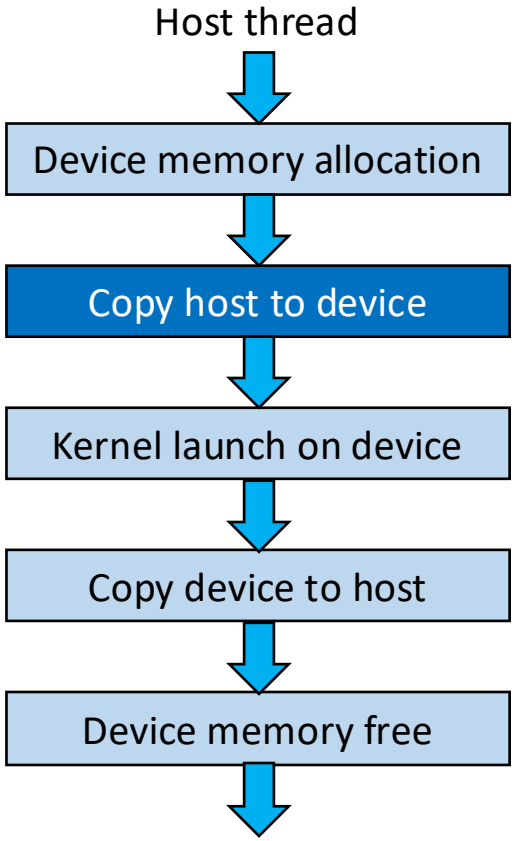


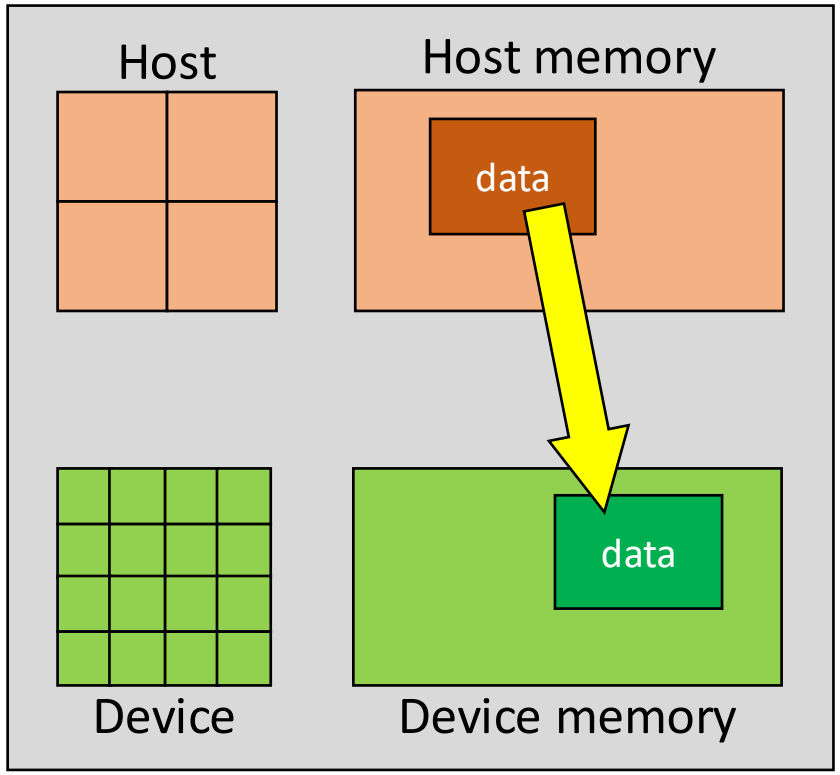
Structure of a basic CUDA program



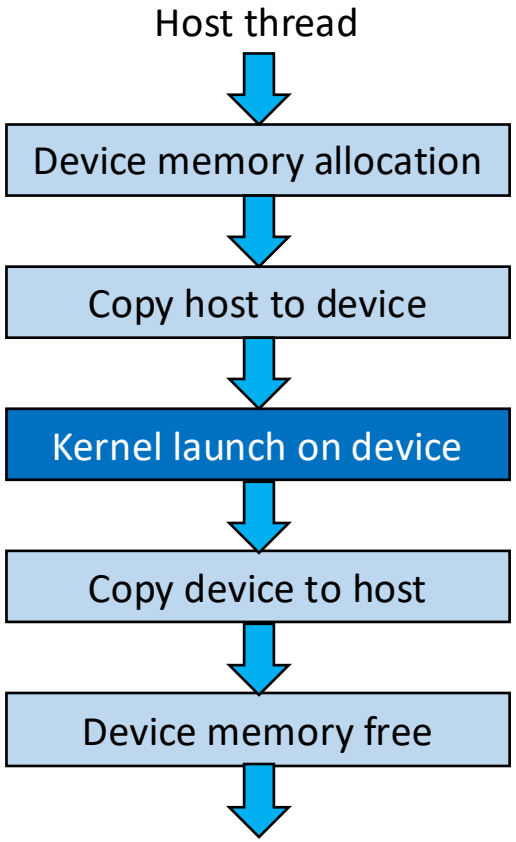


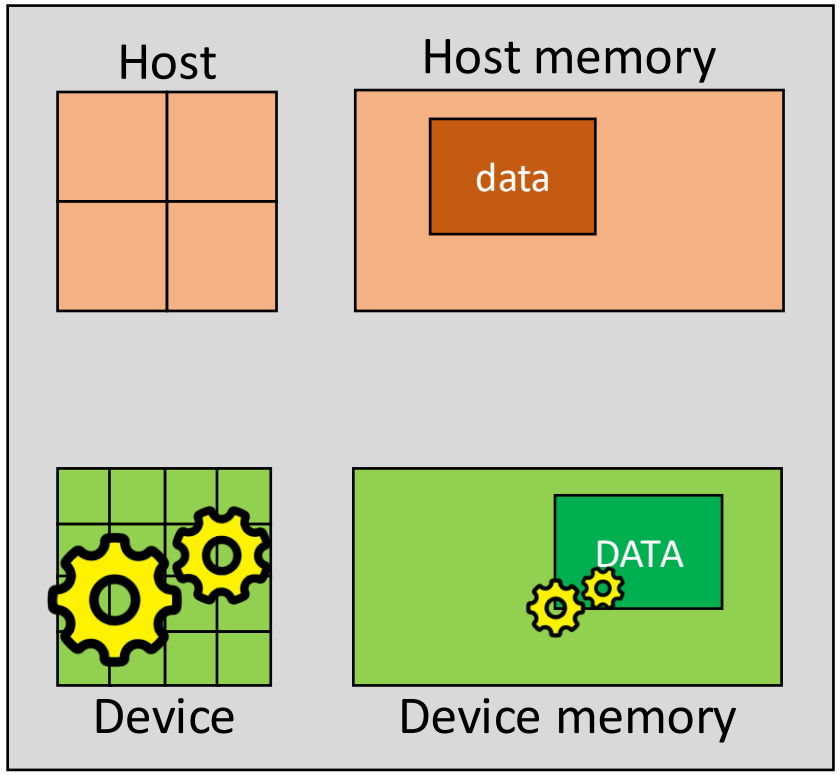
Structure of a basic CUDA program



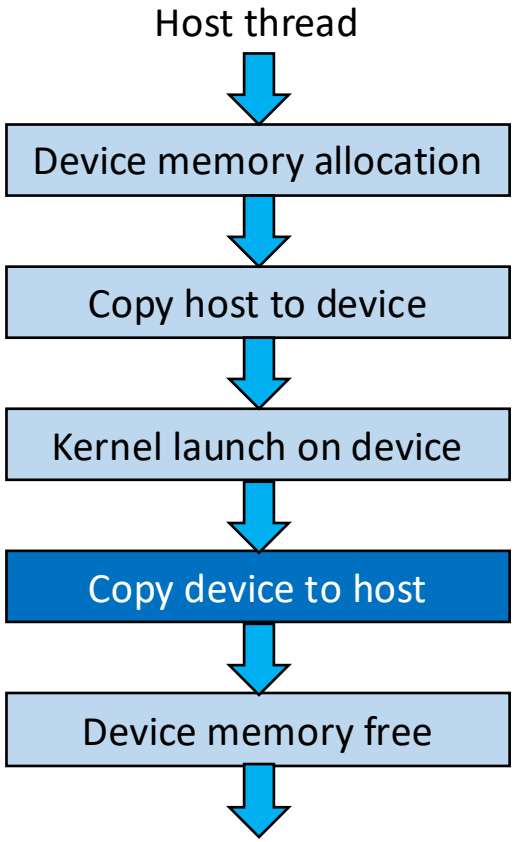


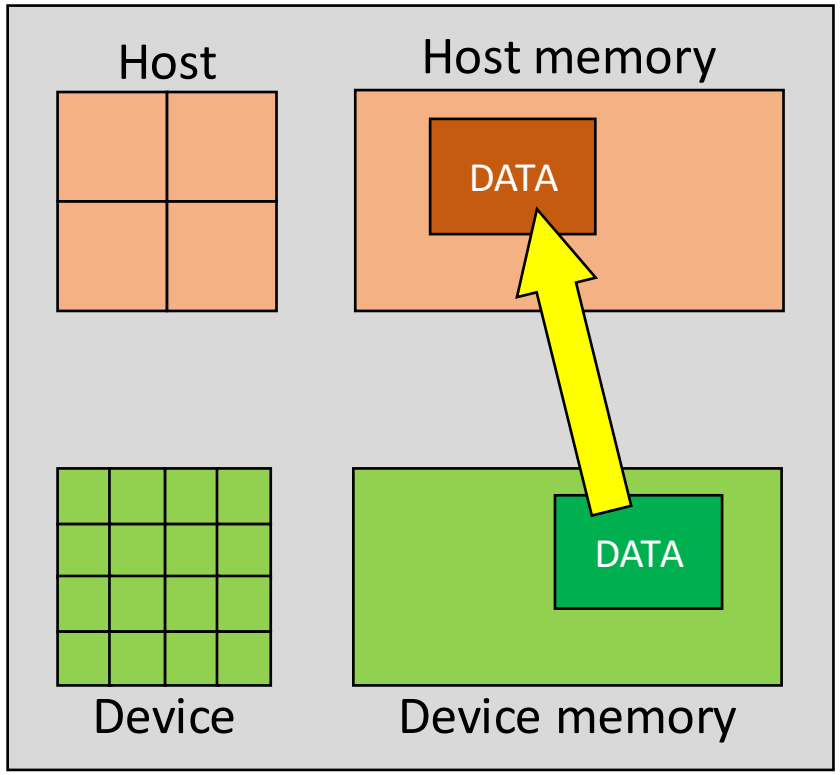
Structure of a basic CUDA program



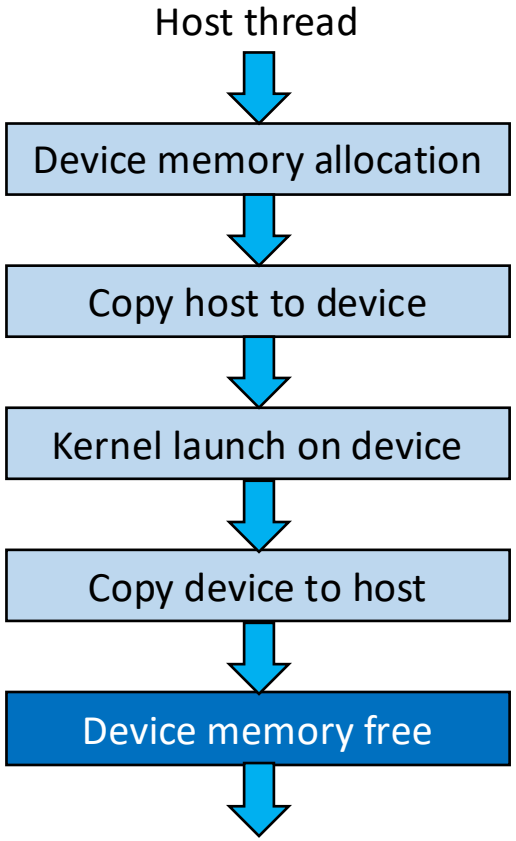


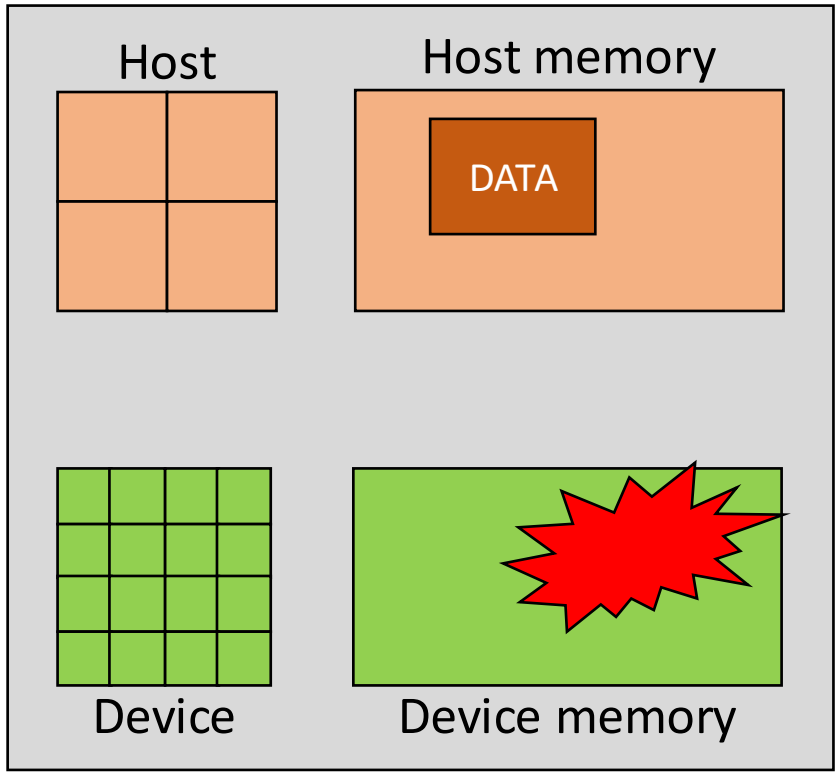
Structure of a basic CUDA program



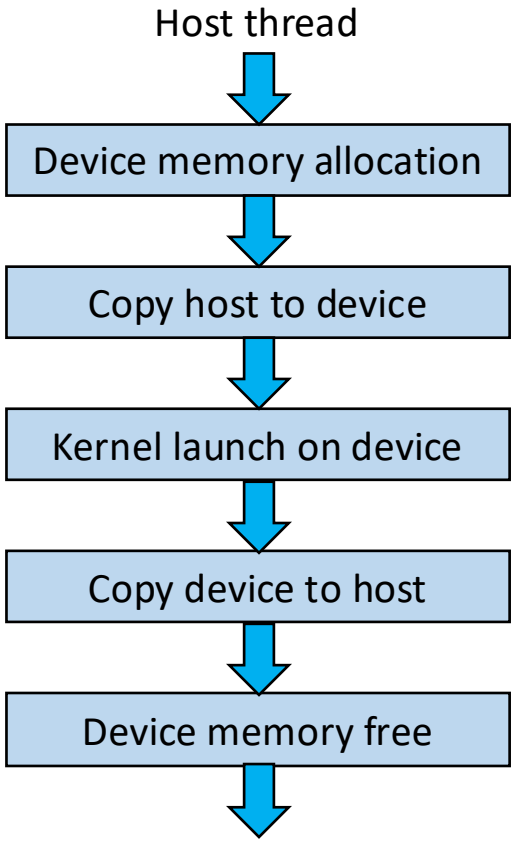


Structure of a basic CUDA program





Structure of a basic CUDA program



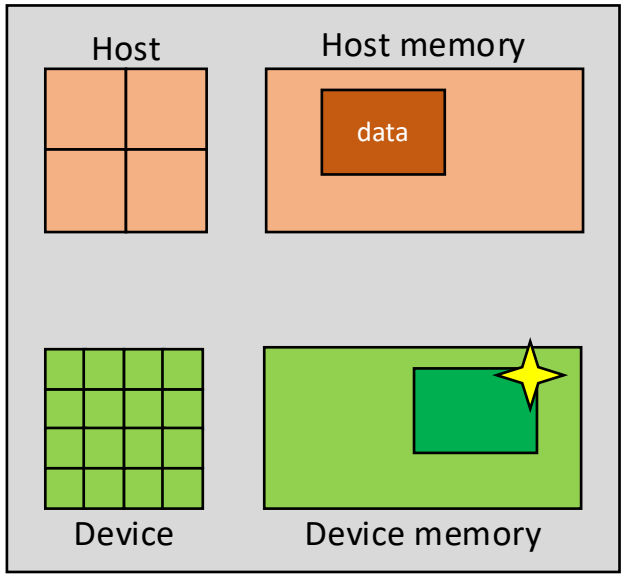

```
cudaMalloc()

cudaMemcpy()

my_kernel<<< … >>>()

cudaMemcpy()

cudaFree()
```

cudaMalloc, cudaFree

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

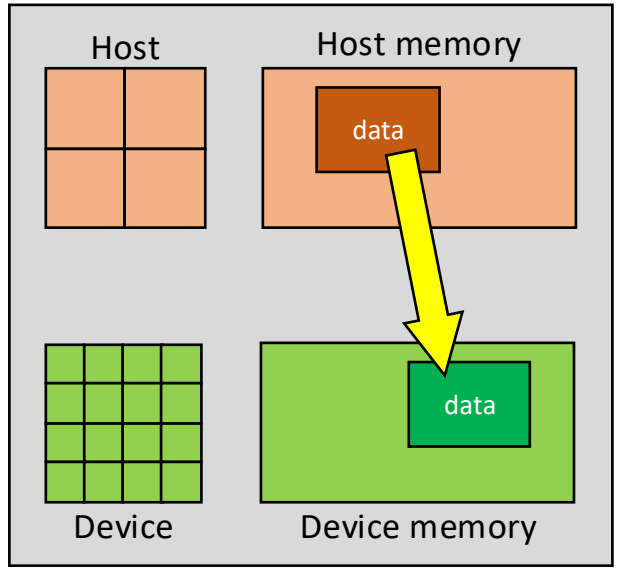
```
cudaError_t cudaMalloc(void ** ptr, size_t num_bytes);
cudaError_t cudaFree(void * ptr);
```

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



cudaMemcpy

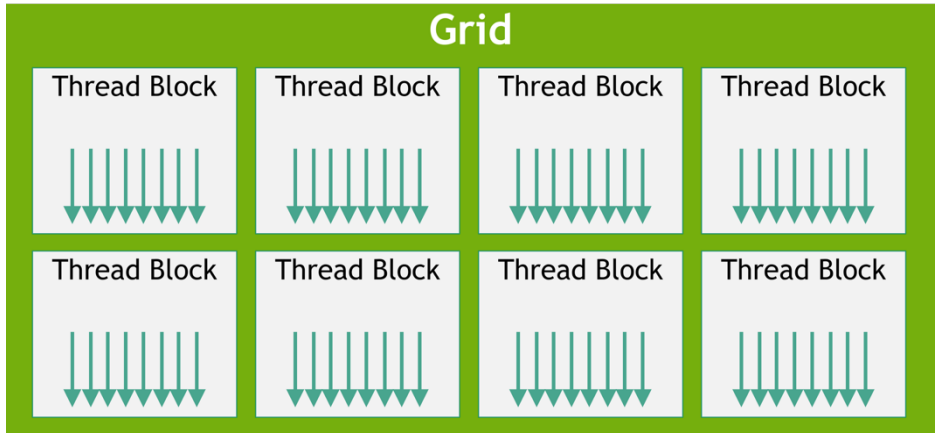
- Copy data between host and device
  - Or host-host, or device-device



- `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

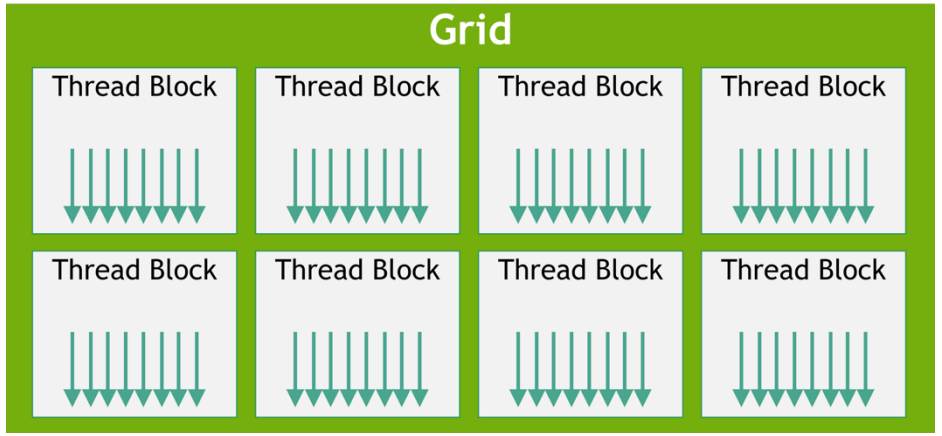


- Special function that runs on the device
- **Threads** are grouped into thread **blocks**
- Blocks are grouped into a **grid**
- Each thread executes the kernel exactly once
- Each block and thread receives a unique set of IDs
  - Built-in variables available inside of the kernel
  - **`blockIdx.x`** – index of block within the grid
  - **`threadIdx.x`** – **local** index of thread within its 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

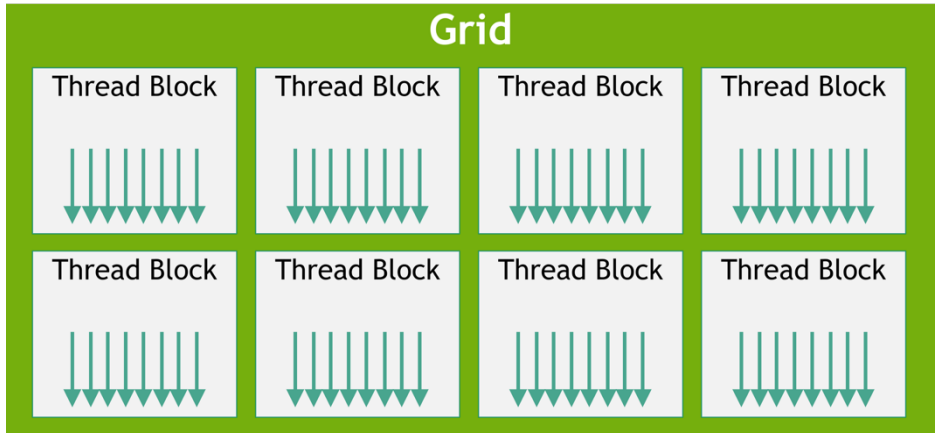


- Free function returning void
- Annotated with `__global__` keyword
  - Two underscores on each side
- Launched using <<< >>> syntax
  - Submitting the GPU kernel from CPU code
  - Number of blocks in grid and threads per block needs to be provided
    - `threads_per_block` is usually set arbitrarily (e.g., 256, max 1024)
    - `blocks_in_grid` is calculated to fit the data ($\left\lceil \frac{data\_size}{threads\_per\_block} \right\rceil$), (max 2^31)
  - `my_kernel<<< blocks_in_grid, threads_per_block >>>(parameters)`
- 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()` to host has implicit synchronization inside

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

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

Kernel



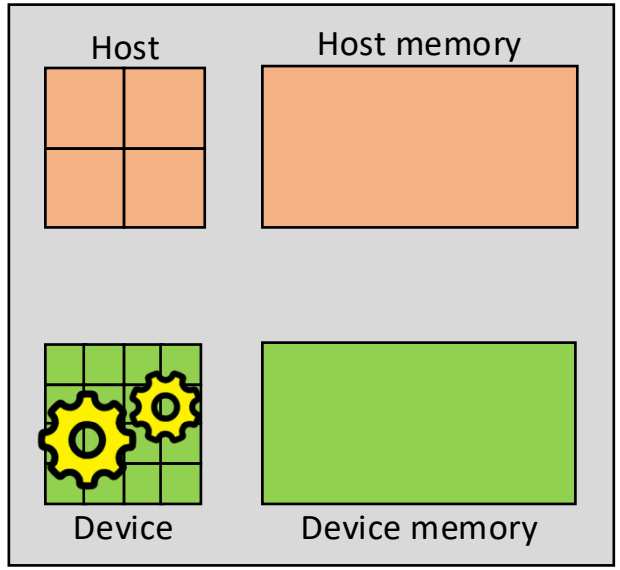
say_hello kernel

```
__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

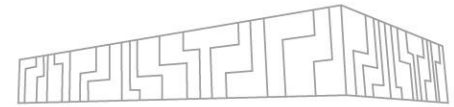
```
say_hello<<< 2, 4 >>>();

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

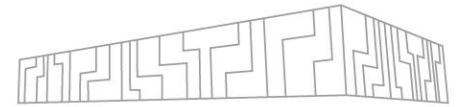
- CUDA is not standard C/C++, need special compiler
- Compiler for CUDA applications – nvcc
  - `nvcc source.cu -o program`

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

- 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 1/8 of the node = 1 GPU and 16 cores

-A, --account
-p, --partition
-G, --gpus
-N, --nodes
-t, --time
-c, --cpus-per-task

```
salloc -A ATR-25-5 -p qgpu -G 1  -N 1 -t 2:00:00
```
- Request **1** GPU on **1** node for 2 hours

```
salloc -A ATR-25-5 -p qgpu
```
- Default: **1** GPU, **1** node, **24h** time limit

```
salloc -A ATR-25-5 -p qgpu -G 4       -t 2:00:00
```
- Request **4** GPUs for 2 hours. You might get the GPUs scattered across **1-4** nodes

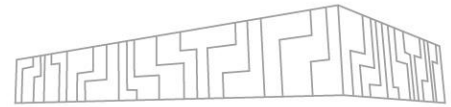
```
salloc -A ATR-25-5 -p qgpu -G 4  -N 1 -t 2:00:00
```
- Request **4** GPUs on **1** node for 2 hours

```
salloc -A ATR-25-5 -p qgpu -G 16 -N 2 -t 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

Hands-on preparation



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

- Clone the git repository, if you haven’t yet
  - `git clone https://code.it4i.cz/training/intro2hpc.git`
- Or copy from project folder
  - `cp -r /mnt/proj1/atr-25-5/intro2hpc ~/`
- Navigate to 5_gpu_accelerators/handson

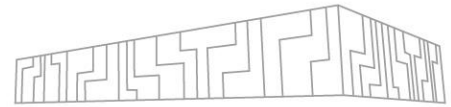
- Run an interactive job in today’s reservation

```
salloc -A ATR-25-5 --reservation=atr-25-5_2026-06-05T09:00_2026-06-05T12:30__qgpu -p qgpu -G 1 -c 16 -t 1:00:00
```

- Load the CUDA module

```
module load CUDA/13.0.0
```

Hands on – Hello world



- Write a Hello world program similar to the one already shown, compile and run it
  - Feel free to use the presentation :)
- Start with `hello_world.task.cu`

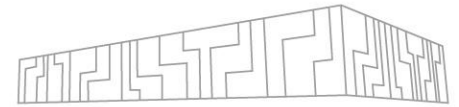
- Additionally, compute and print in each thread:
  - Global index of the thread in the whole grid
  - Total number of threads

`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

Example output:

```
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 (live coding)

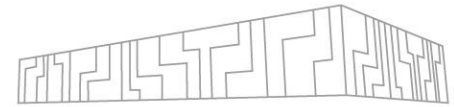


- Scale a vector (array) of floats on the GPU using CUDA
  - X := scalar*X

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

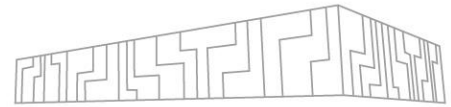
- Aside from running the kernel, we need to copy the data

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 solution

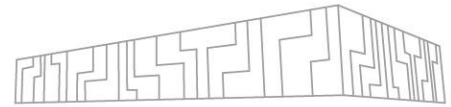
More notes



- Vector add makes no sense to do on GPU
  - Bottleneck: PCIe bandwidth 32 GB/s
- Managed/unified memory
  - Accessible from both CPU and GPU
  - cudaMallocManaged()
  - Migrates on page fault
    - Or cudaMemPrefetchAsync()
- Nvidia A100 has 108 streaming multiprocessors, each can execute 2048 threads
  - ≈220000 threads can be in execution at once
- Asynchronous execution, copy-compute overlap, CPU-GPU overlap

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

AMD ROCm

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

- E.g. El Capitan (#1) and LUMI (#9) use AMD GPUs

HIP

source.cu

```
__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;
}
```

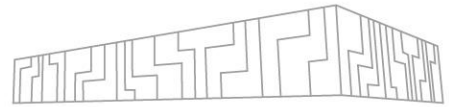
```
$ nvcc source.cu -o program_cuda.x
```

source.hip.cpp

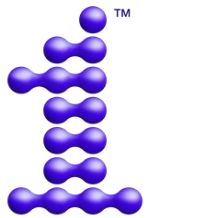
```
#include <hip/hip_runtime.h>

__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 (#3)
- 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

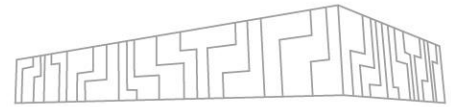


oneAPI

AdaptiveCpp

intel

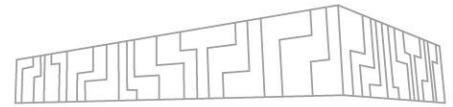
SYCL



```
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

Multi-GPU program

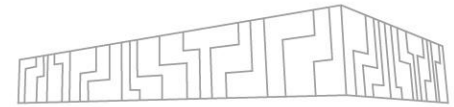


- Multi-GPU system
- Which GPU is being used?

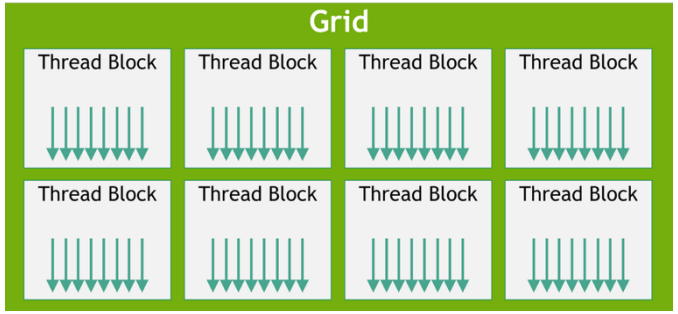
- Some default one.

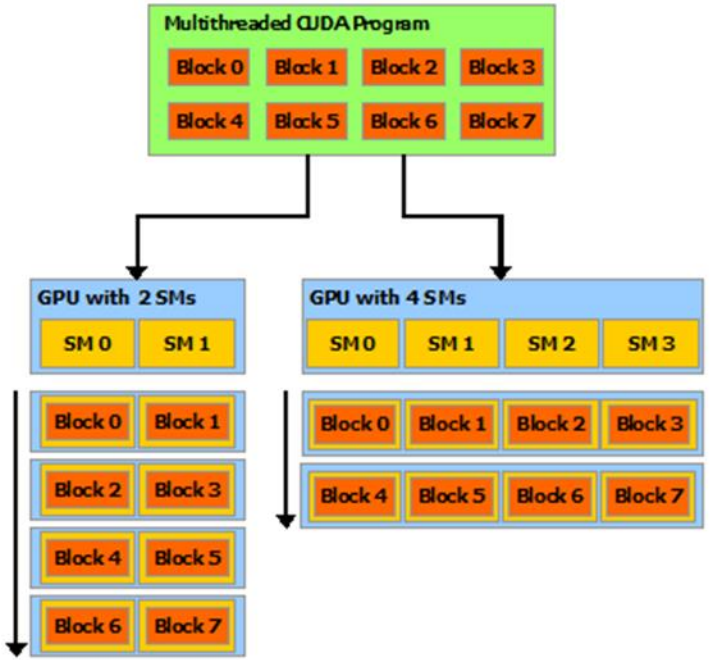
- It is possible to set which GPU we want to work with
  - cudaSetDevice()
  - CUDA_VISIBLE_DEVICES environment variable
- Multi-GPU programming
  - Iterate over all GPUs in a loop, submit work to all of them
  - Or OpenMP thread per GPU, MPI rank per GPU, …
  - cudaMemcpy between GPUs (peer to peer),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

CUDA kernel execution

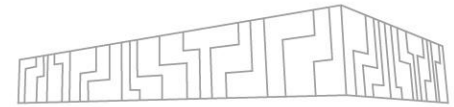


- Each block is executed on a single Streaming Multiprocessor (SM)
  - Independently of other blocks
  - Multiple blocks can run on a single SM if resources allow
- Threads in a block are executed in a SIMD fashion – SIMT
  - Single Instruction Multiple Threads
- Threads are grouped into warps (32 threads)
- 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

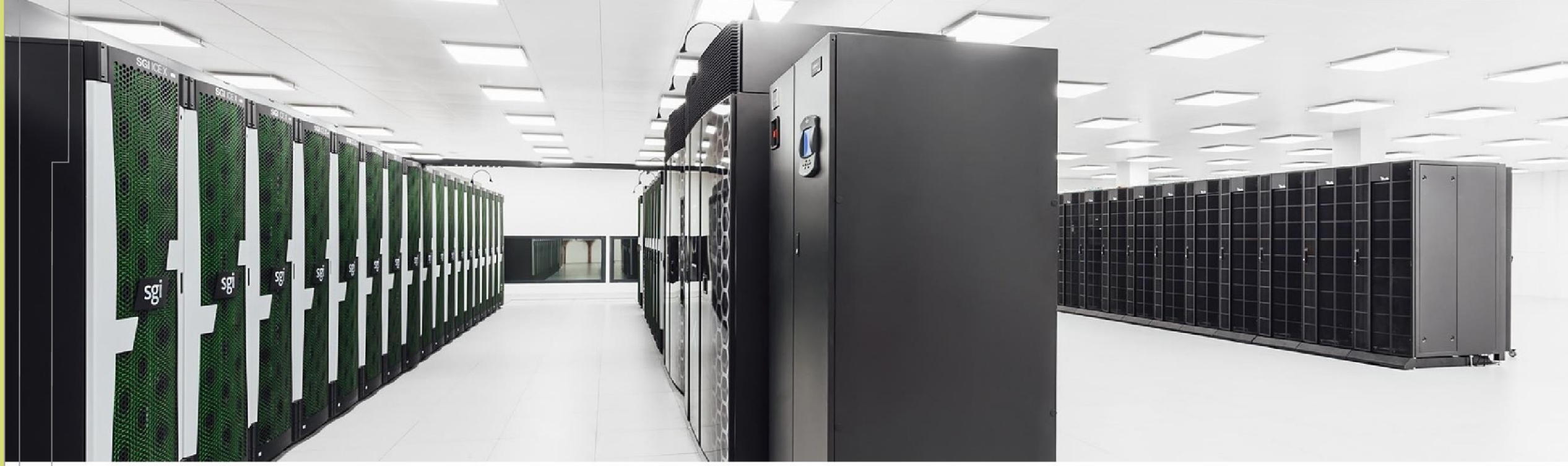




More information, resources



- 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, Radim Vavřík
jakub.homola@vsb.cz, radim.vavrik@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