GPU Programming with CUDA

Lectures: Lubomír Říha

Hands-on: Jakub Homola, Milan Jaroš, Radim Vavřík, Filip Vaverka and Joao Barbosa

IT4Innovations, VSB-TU Ostrava

Co-funded by the European Union

EuroHPC Joint Undertaking

This project has received funding from the European High Performance Computing Joint Undertaking under grant agreementNo.101139786. Views and opinions expressed are, however, those of the author(s) only and do not necessarily reflect those of the European Union orEuroHPC Joint Undertaking. Neither the European Union nor the granting authority can be held responsible for them.

Schedule: Day 1

Day 1	**Length [minutes]**	**Start**	**End**
Heterogeneous Parallel Computing	10	10:00	12:00
GPU Architecture	25		
Hands-on: Accessing GPU accelerated nodes	25		
Hands-on: Benchmark HW properties	15		
CUDA Programming	30		
Hands-on: Hello World in CUDA	15		
Lunch break	**60**	**12:00**	**13:00**
CUDA Programming cont.	20	13:00	14:35
Hands-on: Vector Addition (single GPU, two versions)	40		
Multi-GPU programming	15		
Hands-on: Vector Addition (multi-GPU)	20		
Break	**20**	**14:35**	**14:55**
Efficient Host-Device Data Transfer and CUDA Streams	15	14:55	16:40
Hands-on: Streams	40		
Multi-Dimensional Grids	15		
Hands-on: Image Blur	20		
Thread Execution	15		

Schedule: Day 2

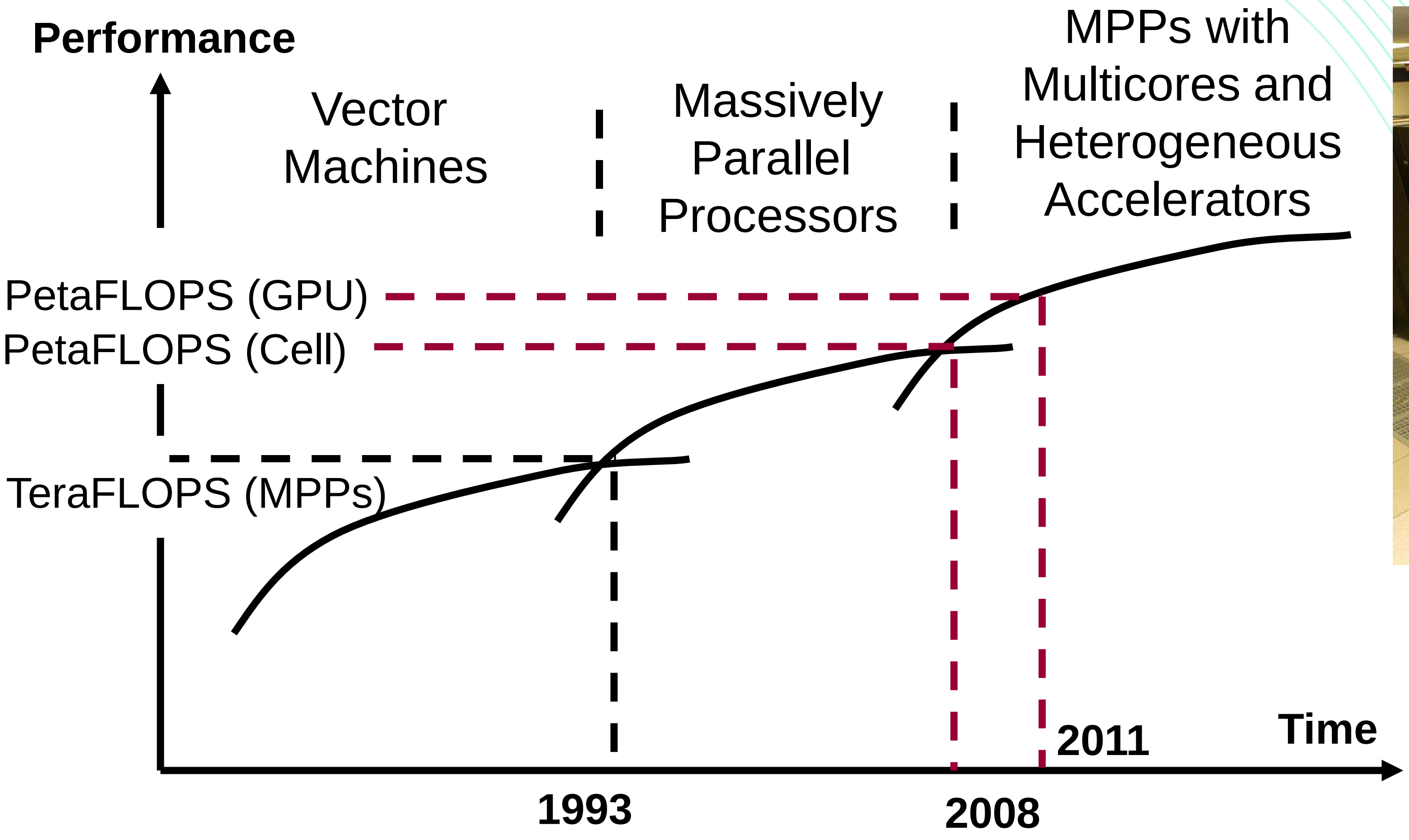
Day 2	**Length [minutes]**	**Start**	**End**
CUDA Memories	10	9:00	10:30
Global Memory	15		
Hands-on: Matrix Sum	20		
Shared Memory – Basics	10		
Shared Memory – Bank conflicts	15		
Hands-on: Matrix Transpose – Shared memory bank conflicts	20		
Break	**20**	**10:30**	**10:50**
Memory and Data Locality: Tiling Technique	15	10:50	13:00
Parallel Computation Patterns: Stencil	20		
Hands-on: Stencil – 1D Convolution	20		
Parallel Computation Patterns: Reduction	15		
Hands-on: Parallel Reduction	20		
Parallel Computation Patterns: Histogram	20		
Hands-on: Histogram – Data Race, Atomics, Privatization	20		

EPICURE

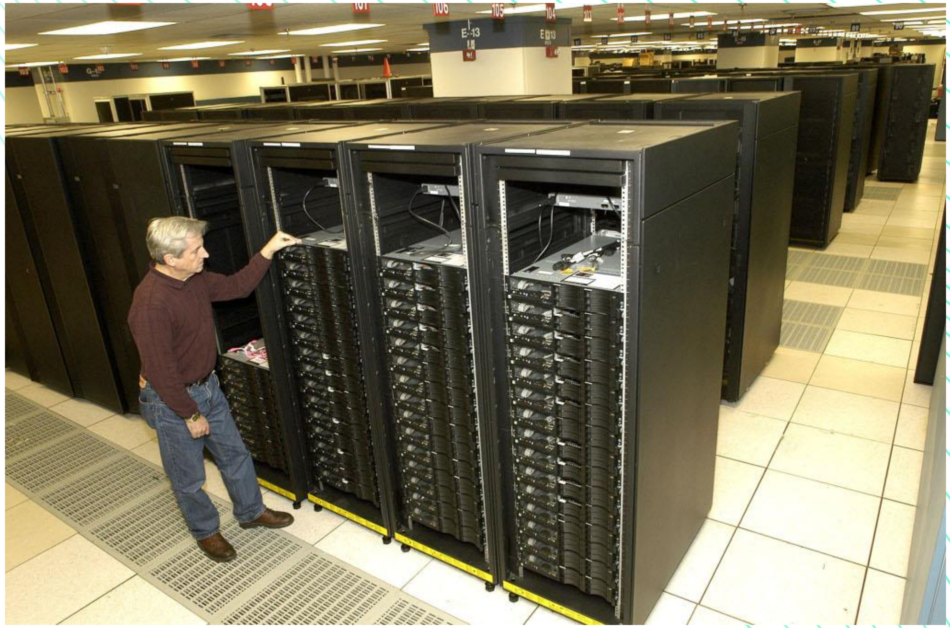
Unlocking European-level HPC Support

Heterogeneous Parallel Computing

Accelerators in HPC
Historical Analysis



End of Moore’s Law in Clocking!

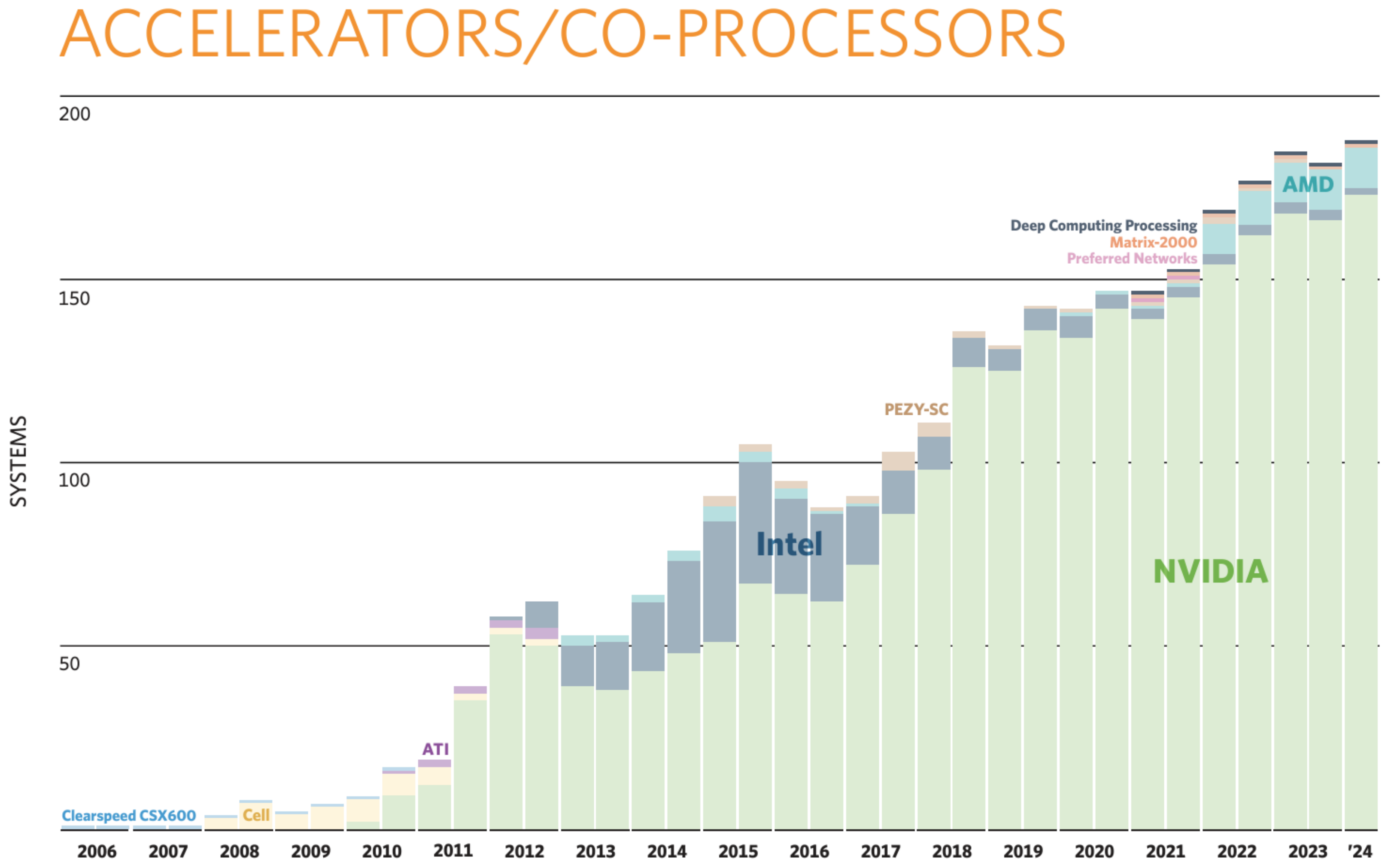


IBM Roadrunner (2008)

- the first heterogeneous supercomputer
- installed in Los Alamos National Lab
- 6,480 AMD Opteron processors
  - with 52 TB RAM
- 12,960 PowerXCell 8i processors
- 296 racks - 2.35 MW power consumption

Accelerators in HPC Historical Analysis

Computer	# CPU cores	Year
Frontier, USA	8 730 112	2022
Fugaku, Japan	7 630 848	2020
Summit, USA	2 414 592	2018
Sunway TAIHULIGHT	10 649 600	2016
TIANHE-2, CHINA	3 120 000	2015
Titan, USA	560 640	2012
Sequoia, USA (BlueGene/Q)	1 572 864	2012
K-Computer, Japan	548 352	2011
Tianhe-1A, China	186 368	2010
Jaguar, Cray	224 162	2009
Roadrunner, USA	122 400	2008
BlueGene/L	212 992	2007



ACCELERATORS/CO-PROCESSORS

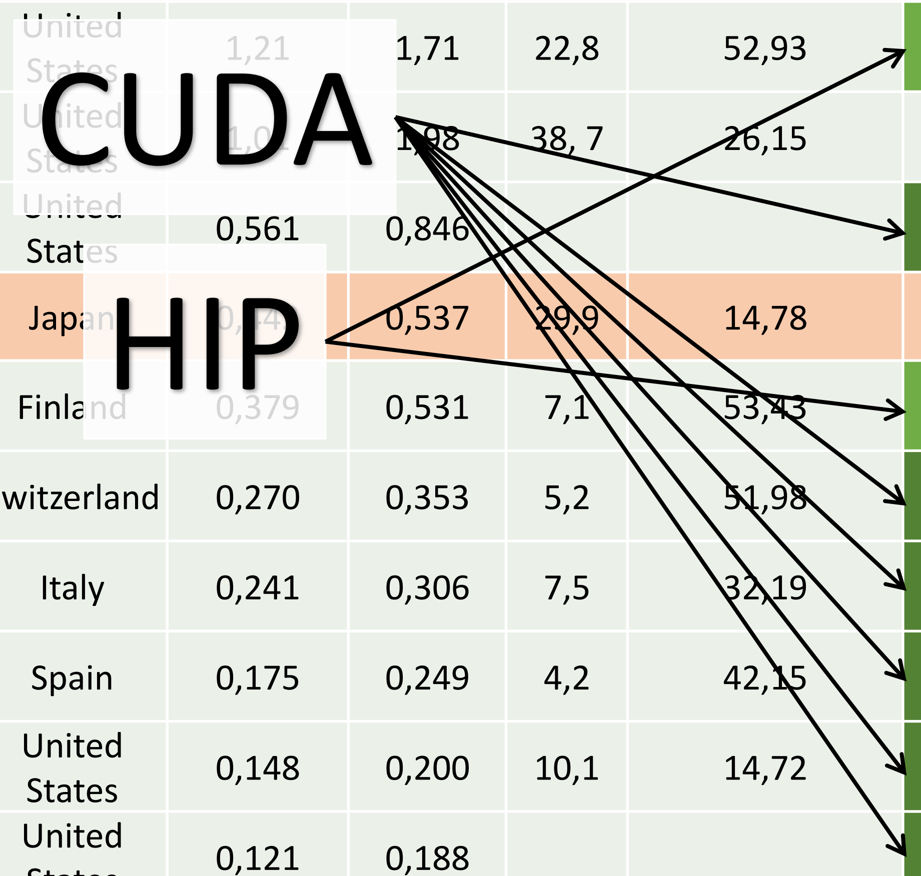
Source: Top500 list, https://www.top500.org/lists/top500/2024/06/

Accelerators in HPC as of June 2024

Rank	Name	Computer	Site	Country	Rmax [EFlop/s]	Rpeak [EFlop/s]	Power (MW)	Energy Efficiency [GFlops/Watts]	Accelerator
1	Frontier	HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11	DOE/SC/ORNL	United States	1,21	1,71	22,8	52,93	AMD MI250X
2	Aurora	HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Max GPU, Slingshot-11	DOE/SC/AANL	United States	1,01	1,98	38, 7	26,15	Intel Max
3	Eagle	Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, Infiniband NDR	Microsoft Azure	United States	0,561	0,846			NVIDIA H100
4	Fugaku	Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D	RIKEN R-CCS	Japan	0,442	0,537	29,9	14,78	None
5	LUMI	HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11	EuroHPC CSC	Finland	0,379	0,531	7,1	53,43	AMD MI250X
6	Alps	HPE Cray EX254n, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11	CSCS	Switzerland	0,270	0,353	5,2	51,98	NVIDIA GH200
7	Leonardo	BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, HDR100 Infiniband	EuroHPC CINECA	Italy	0,241	0,306	7,5	32,19	NVIDIA A100
8	MareNostrum 5 ACC	BullSequana XH3000, Xeon Platinum 8460Y+ 32C 2.3GHz, NVIDIA H100 64GB, Infiniband NDR	EuroHPC BSC	Spain	0,175	0,249	4,2	42,15	NVIDIA H100
9	Summit	IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, EDR Infiniband	DOE/SC/ORNL	United States	0,148	0,200	10,1	14,72	NVIDIA GV100
10	Eos NVIDIA DGX SuperPOD	NVIDIA DGX H100, Xeon Platinum 8480C 56C 3.8GHz, NVIDIA H100, Infiniband NDR400	NVIDIA Corporation	United States	0,121	0,188			NVIDIA H100

Accelerators in HPC as of June 2024

Rank	Name	Computer	Site	Country	Rmax [EFlop/s]	Rpeak [EFlop/s]	Power (MW)	Energy Efficiency [GFlops/Watts]	Accelerator
1	Frontier	HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11	DOE/SC/ORNL	United States	1,21	1,71	22,8	52,93	AMD MI250X
2	Aurora	HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Max GPU, Slingshot-11	DOE/SC/AANL	United States	1,01	1,98	38, 7	26,15	Intel Max
3	Eagle	Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, Infiniband NDR	Microsoft Azure	United States	0,561	0,846			NVIDIA H100
4	Fugaku	Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D	RIKEN R-CCS	Japan	0,442	0,537	29,9	14,78	None
5	LUMI	HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11	EuroHPC CSC	Finland	0,379	0,531	7,1	53,43	AMD MI250X
6	Alps	HPE Cray EX254n, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11	CSCS	Switzerland	0,270	0,353	5,2	51,98	NVIDIA GH200
7	Leonardo	BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, HDR100 Infiniband	EuroHPC CINECA	Italy	0,241	0,306	7,5	32,19	NVIDIA A100
8	MareNostrum 5 ACC	BullSequana XH3000, Xeon Platinum 8460Y+ 32C 2.3GHz, NVIDIA H100 64GB, Infiniband NDR	EuroHPC BSC	Spain	0,175	0,249	4,2	42,15	NVIDIA H100
9	Summit	IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, EDR Infiniband	DOE/SC/ORNL	United States	0,148	0,200	10,1	14,72	NVIDIA GV100
10	Eos NVIDIA DGX SuperPOD	NVIDIA DGX H100, Xeon Platinum 8480C 56C 3.8GHz, NVIDIA H100, Infiniband NDR400	NVIDIA Corporation	United States	0,121	0,188			NVIDIA H100



Accelerators in HPC Now

					Rmax	Rpeak	Power	Energy Efficiency	
10	DGX SuperPOD	H100, Infiniband NDR400	Corporation	States	0,121	0,188			H100

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;
}
```

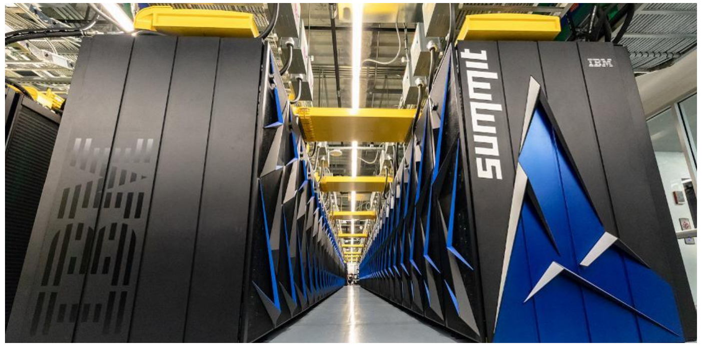
```
$ nvcc source.cu -o program_cuda.x
```

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
```

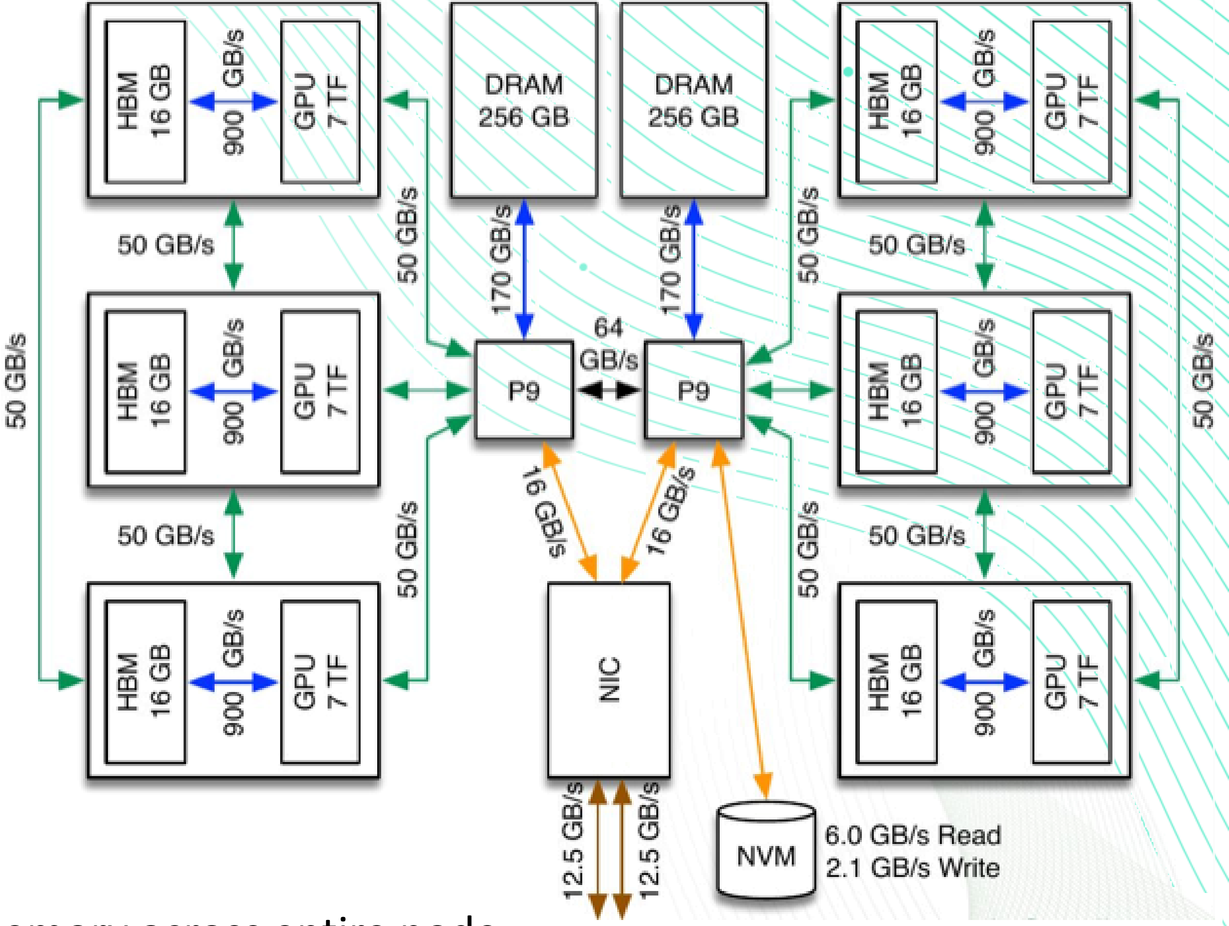
Accelerators in HPC
ORNL Summit Supercomputer



Summit: DOE/SC/Oak Ridge National Laboratory

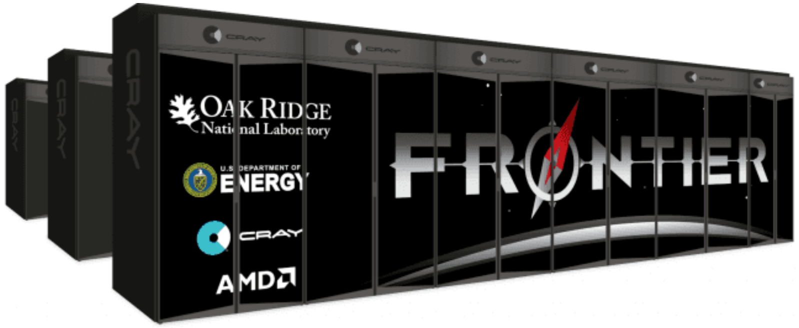
No.1 from Jun 2018 until Nov 2019

Number of Nodes	**4,608 = 27 648 GPUs**
Performance	**200 PF Peak, 148 Linpack (FP64)** **3.3 ExaOps (FP16)**
Node performance	**42 TF**
Memory per Node	**512 GB DDR4 + 96 GB HBM2**
NV memory per Node	1600 GB
Total System Memory	>10 PB DDR4 + HBM2 + Non-volatile
System Interconnect	Dual Rail Infiniband EDR (25 GB/s)
Interconnect Topology	Non-blocking Fat Tree
Processors	**2x IBM POWER9** **6x NVIDIA Volta**
File System	250 PB, 2.5 TB/s, GPFSTM
Power Consumption	13 MW



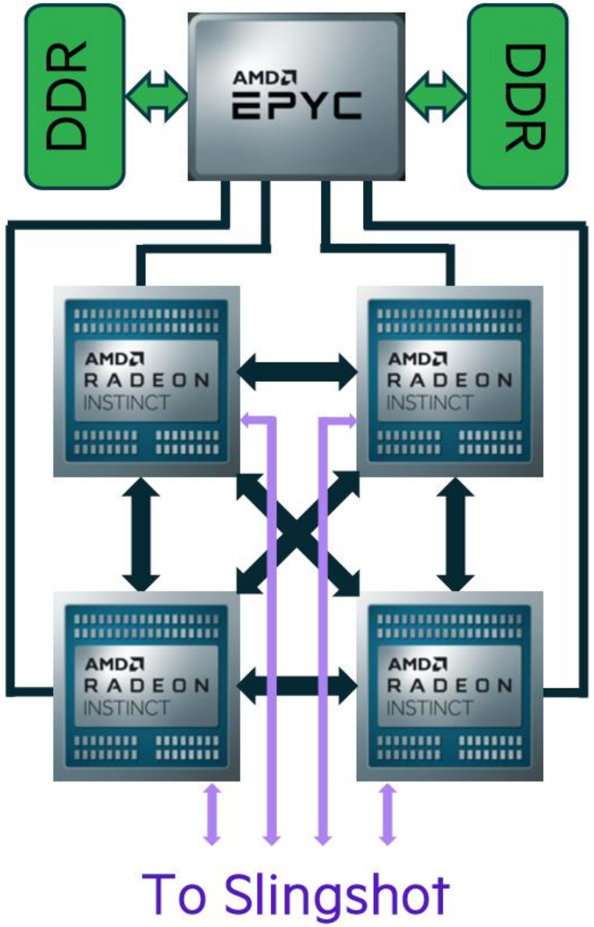
- Coherent memory across entire node
- NVLink v2 fully interconnects three GPUs and one CPU on each side node
- PCIe Gen4 connects NVMe and NIC
- Single shared NIC with dual EDR ports

Accelerators in HPC
ORNL Frontier Supercomputer



System

- 1.7 EF Peak DP FLOPS
- 74 compute racks
- 29 MW Power Consumption
- 9,408 nodes **= 37 632 GPUs**
- 9.2 PB memory
  (4.6 PB HBM, 4.6 PB DDR4)
- Cray Slingshot network with dragonfly topology
- 37 PB Node Local Storage
- 716 PB Center-wide storage
- 360 m^2 foot print



AMD node

- 1 AMD “Trento” CPU
- 4 AMD MI250X GPUs
- 512 GiB DDR4 memory on CPU
- 512 GiB HBM2e total per node
  (128 GiB HBM per GPU)
- Coherent memory across the node
- 4 TB NVM
- GPUs & CPU fully connected with AMD Infinity Fabric
- 4 Cassini NICs, 100 GB/s network BW

Compute blade

- 2 AMD nodes

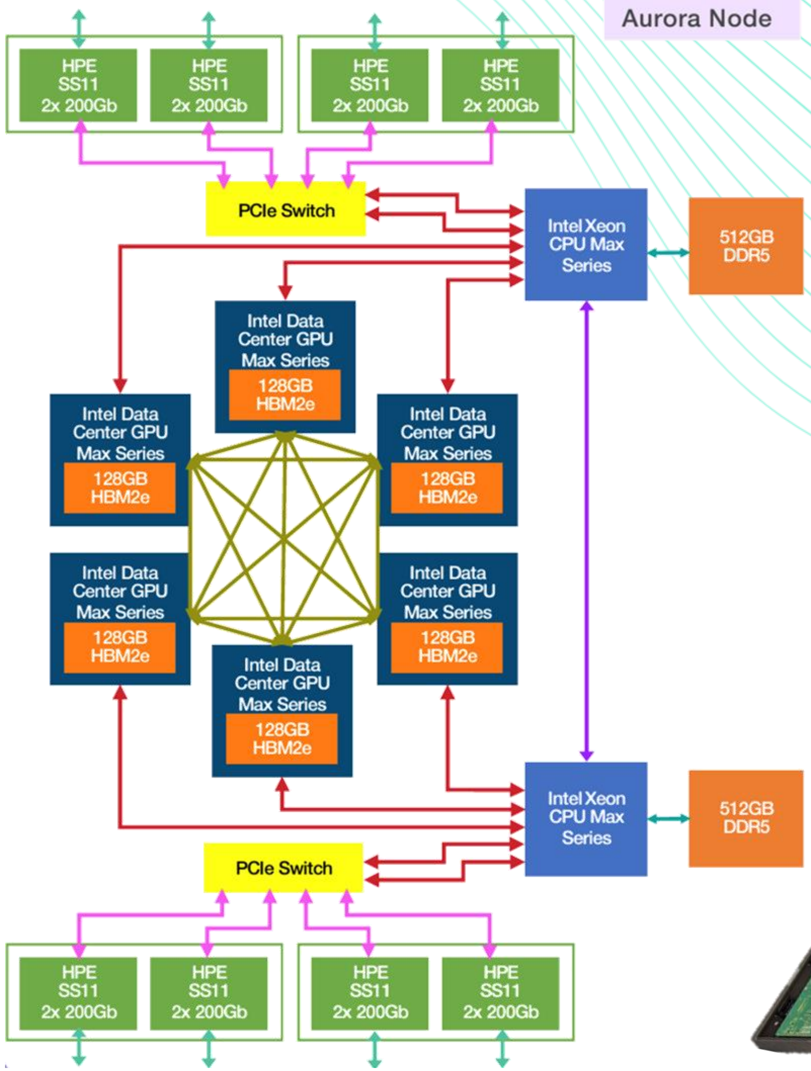
OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Source: Frontier’s Exascale Architecture by Scott Atchley, ExaComm2022

Accelerators in HPC
ALCF Aurora Supercomputer

Argonne NATIONAL LABORATORY
U.S. DEPARTMENT OF ENERGY
intel
Hewlett Packard Enterprise
Aurora

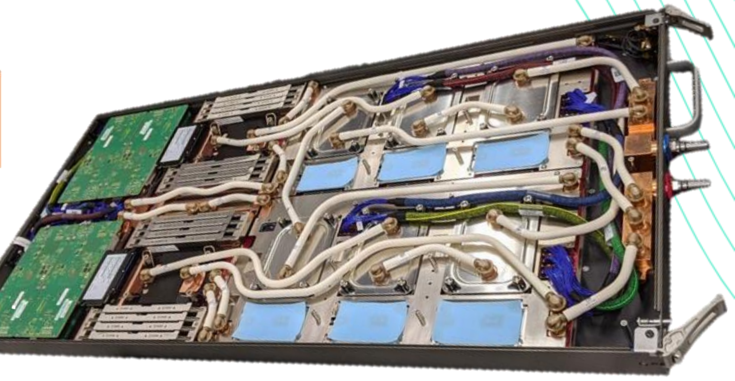
- Nodes: 10,624 (Racks: 166)
  - CPUs: 21,248
  - GPUs: **63,744**
- Peak FP Performance ≧ 2 Exaflops DP
- Memory
  - 10.9PB of DDR @ 5.95 PB/s
  - 1.36PB of CPU HBM @ 30.5 PB/s
  - 8.16PB of GPU HBM @ 208.9 PB/s
- Network: HPE Slingshot 11
  - Dragonfly topology
- Storage:
  - 230PB DAOS Capacity
  - 31 TB/s DAOS Bandwidth



Aurora Node

HPE SS11 2x 200Gb
HPE SS11 2x 200Gb
HPE SS11 2x 200Gb
HPE SS11 2x 200Gb
PCIe Switch
Intel Xeon CPU Max Series
512GB DDR5
Intel Data Center GPU Max Series 128GB HBM2e
Intel Data Center GPU Max Series 128GB HBM2e
Intel Data Center GPU Max Series 128GB HBM2e
Intel Data Center GPU Max Series 128GB HBM2e
Intel Data Center GPU Max Series 128GB HBM2e
Intel Data Center GPU Max Series 128GB HBM2e
Intel Xeon CPU Max Series
512GB DDR5
PCIe Switch
HPE SS11 2x 200Gb
HPE SS11 2x 200Gb
HPE SS11 2x 200Gb
HPE SS11 2x 200Gb

Node characteristics:
- 6x GPUs - Intel Max GPU
- 2x CPUs - Intel Xeon Max CPU
- 768 GB GPU HBM Memory
  - 19.66 TB/s Peak GPU HBM BW
- 128 GB CPU HBM Memory
  - 2.87 TB/s Peak CPU HBM BW
- 1024 GB CPU DDR5 Memory
  - 0.56 TB/s Peak CPU DDR5 BW
- ≧ 130 TF Peak Node DP FLOPS
- 200 GB/s Max Fabric Injection
- 8 NICs

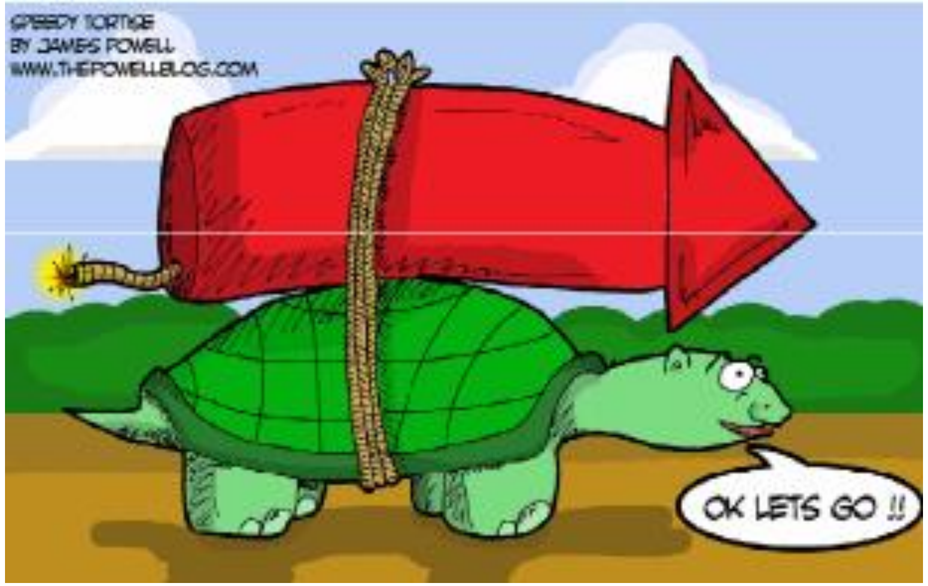


Source: IXPUG Webinar: Preparing for Exascale on Aurora, https://www.ixpug.org/resources/preparing-for-exascale-on-aurora

Accelerators in HPC

Device	Fabrica-tion process [nm]	Clock freq. [GHz]	No. of cores	Peak floating point performance SP/DP [TFLOPs]	Peak power consumption [W]	Perf. Per Watt SP/DP [GFLOPs/W]	Theoretical Memory Bandwidth [GB/s]	Memory type
Intel Xeon® 6 - 6980P (Granite Rapids)	5	2.0	128	16,3/8,2	500	32,7/16,3	614 844	DDR5 MRDIMM
AMD EPYC™ 9654	5	2.4	96	14,7/7,4	360	40,9/20,5	461	DDR5
AMD EPYC™ 7763	7	2.45	64	5,1/2,5	280	26/13	190	DDR4
Nvidia H100 Nvidia H200	4N	1.83	16896 (132 SMs)	67/34	700	95,7/48,5	3350 4800	HBM3
NVidia A100	7	1.41	6912 (108 SMs)	19,5/9,7	400	49/24	2039	HBM2e
AMD MI300A	5	2.1 GPU 3.7 CPU	14592 (228 CUs) 24 CPU cores	122/61,3	760	161/80	5300	HBM3
AMD MI300X	5	2.1	19,456 (304 CUs)	163/81,7	750	217/109	5300	HBM3
AMD MI250X	6	1.7	14080 (220 CUs)	47,9/47,9	560	86/86	3277	HBM2e
Intel (PVC) Max GPU	7	1.6	16384 (1024 Eus)	52,4/52,4	600	87/87	3210	HBM2e
Intel Xeon Phi KNL	14	1.3	64	5,3/2,7	215	25/12	400 102	MCDRAM DDR4

Accelerators in HPC
Heterogeneous Computing

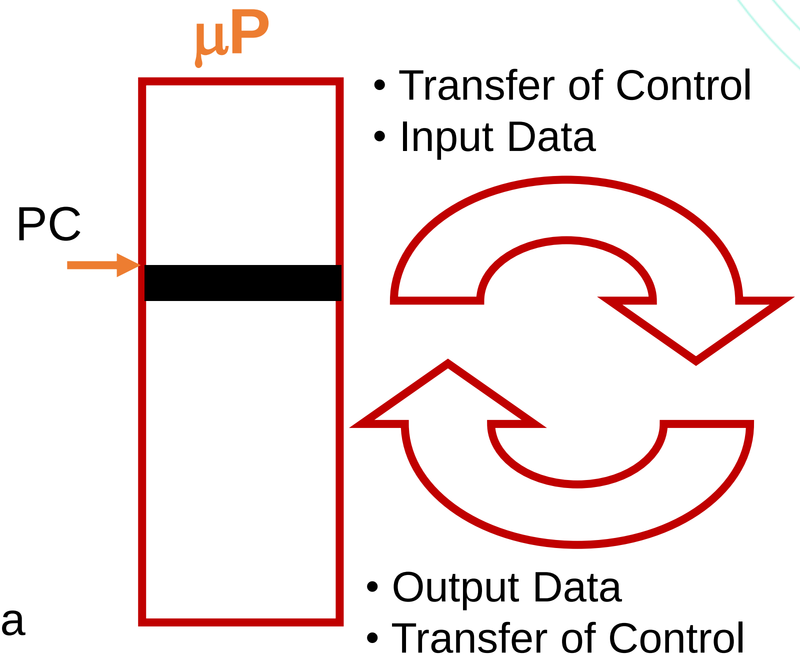


Main Features

- Coprocessor to the CPU
- PCIe based interconnection
- Separate GPU memory
- Provide high bandwidth access to local data
- Slow access to the CPU memory

Hardware Accelerators - Speeding up the Slow Part of the Code

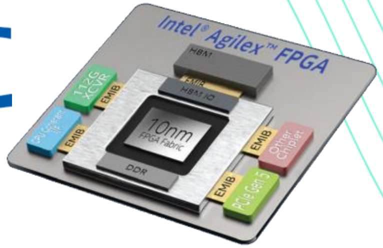
- Enable higher performance through fine-grained parallelism
- Offer higher computational density than CPUs
- Accelerators present heterogeneity!



- Transfer of Control
- Input Data

- Output Data
- Transfer of Control

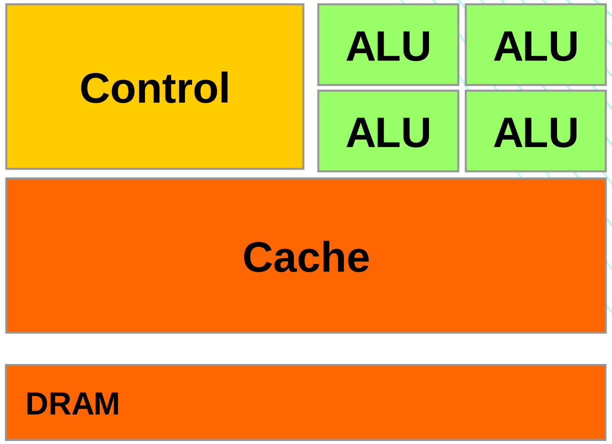
Vector Engine Processors



Accelerators in HPC

Accelerators

- 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

- reduced operation latency

Large caches

- convert long latency memory accesses to short latency cache accesses

Sophisticated control with branch prediction for reduced branch latency

Very Efficient For

- Fast Parallel Floating Point Processing
- High Computation per Memory Access

Not As Efficient For

- Branching-Intensive Operations
- Random Access,
- Memory-Intensive Operations

GPUs

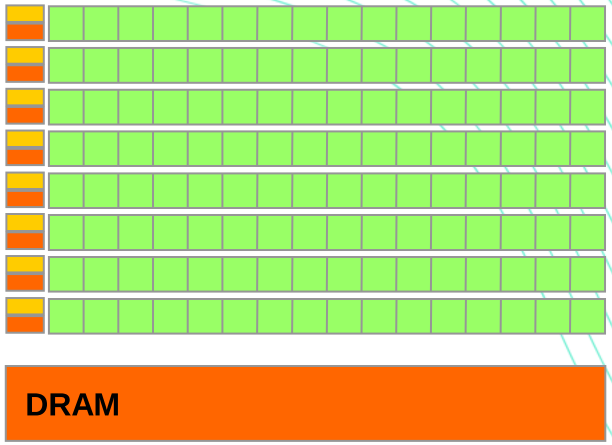
Small caches to boost memory throughput

Simple control with no branch prediction

Energy efficient ALUs

- many, long latency but heavily pipelined for high throughput

Require massive number of threads to tolerate latencies



Accelerators in HPC

Accelerators

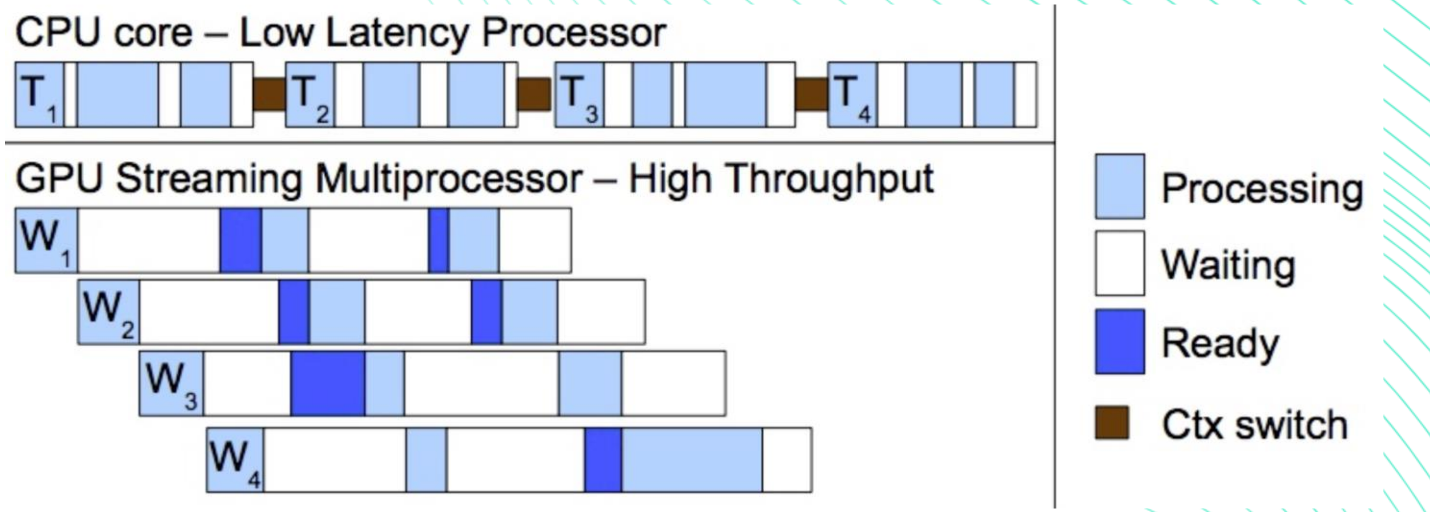
- tailored for compute-intensive, highly data parallel computation
- many parallel execution units
- have significantly faster and more advanced memory interfaces
- more transistors is devoted to data processing
- less transistors for data caching and flow control

Very Efficient For

- Fast Parallel Floating Point Processing
- High Computation per Memory Access

Not As Efficient For

- Branching-Intensive Operations
- Random Access,
- Memory-Intensive Operations

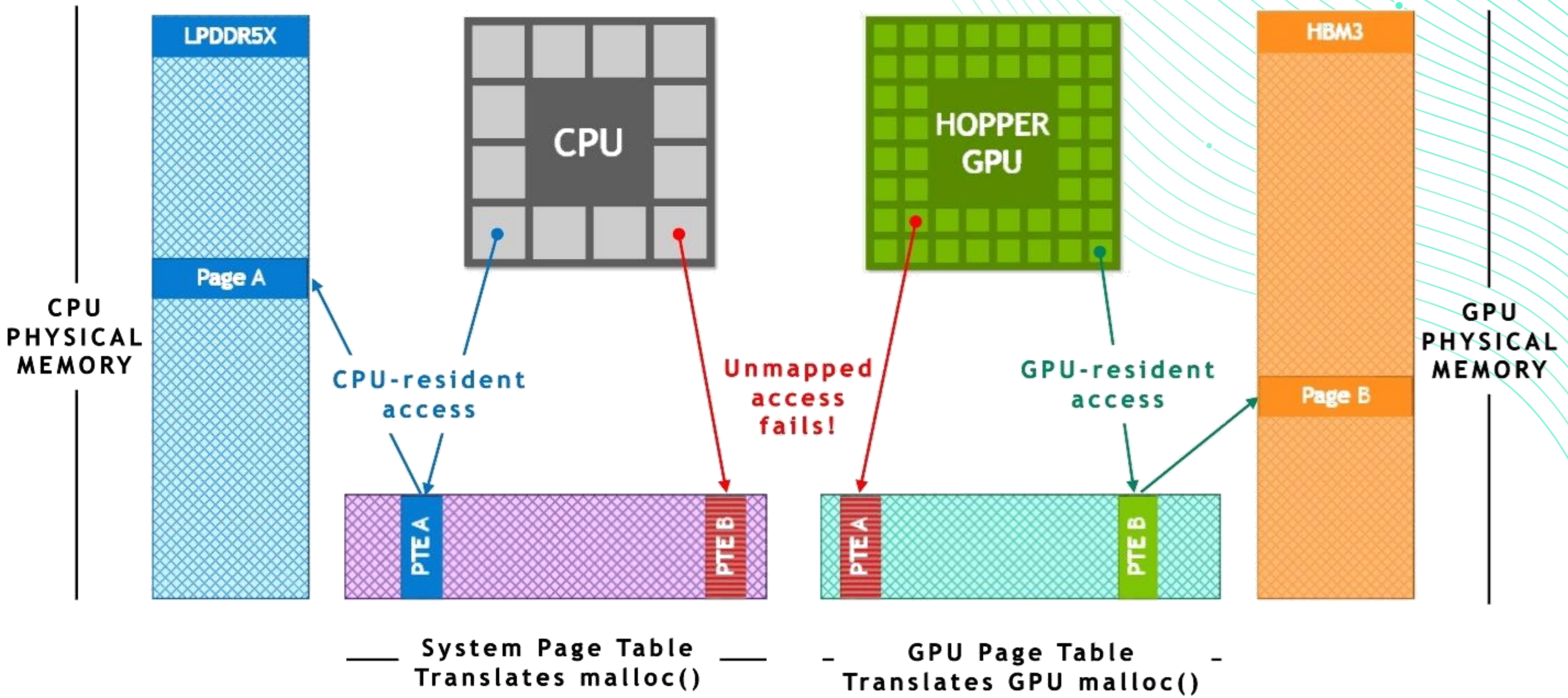

CPU core – Low Latency Processor
T1
T2
T3
T4
GPU Streaming Multiprocessor – High Throughput
W1
W2
W3
W4
Processing
Waiting
Ready
Ctx switch

NVIDIA Corporation 2010

GPU are throughput devices

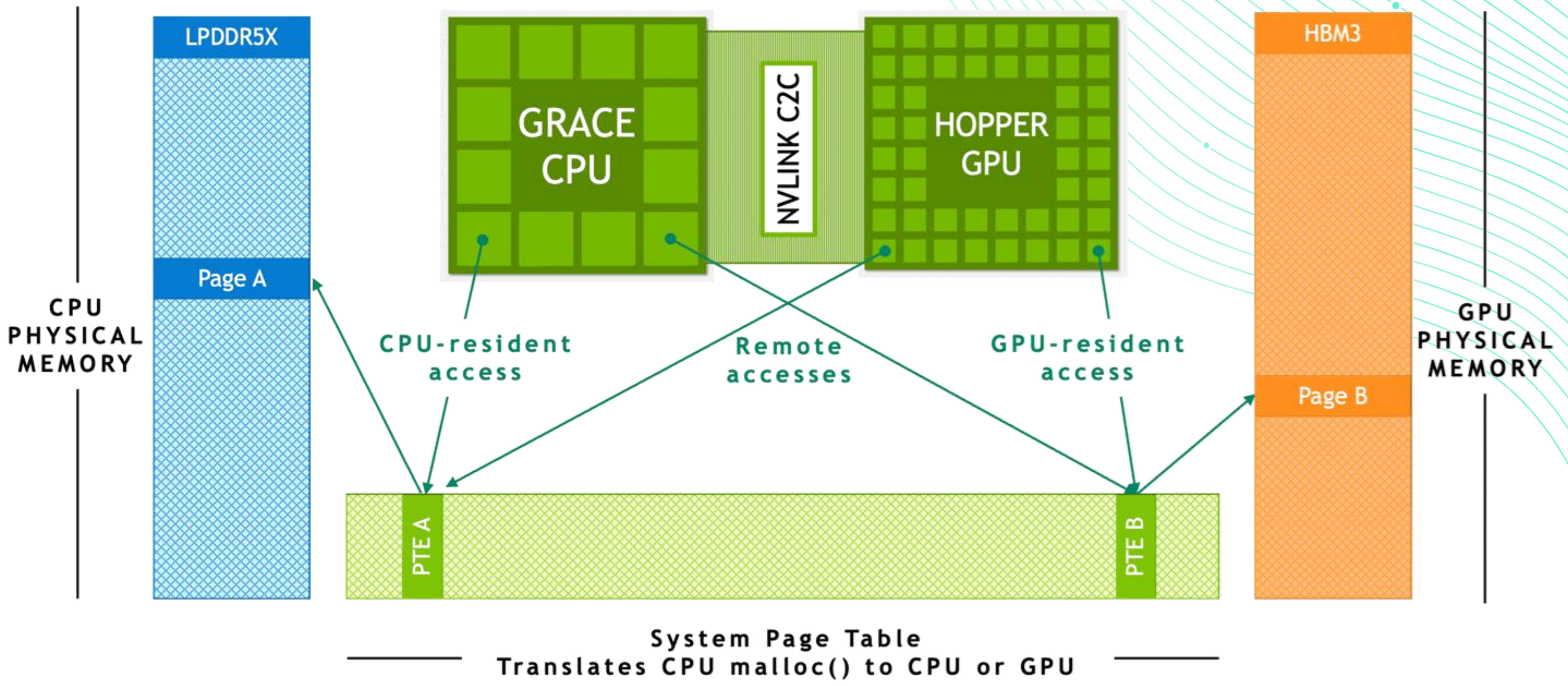
- CPU cores are optimized to minimize latency between operations.
- GPUs aim to minimize latency between operations by scheduling multiple warps (thread bundles).

Accelerators in HPC: Current trends
Unified memory address space



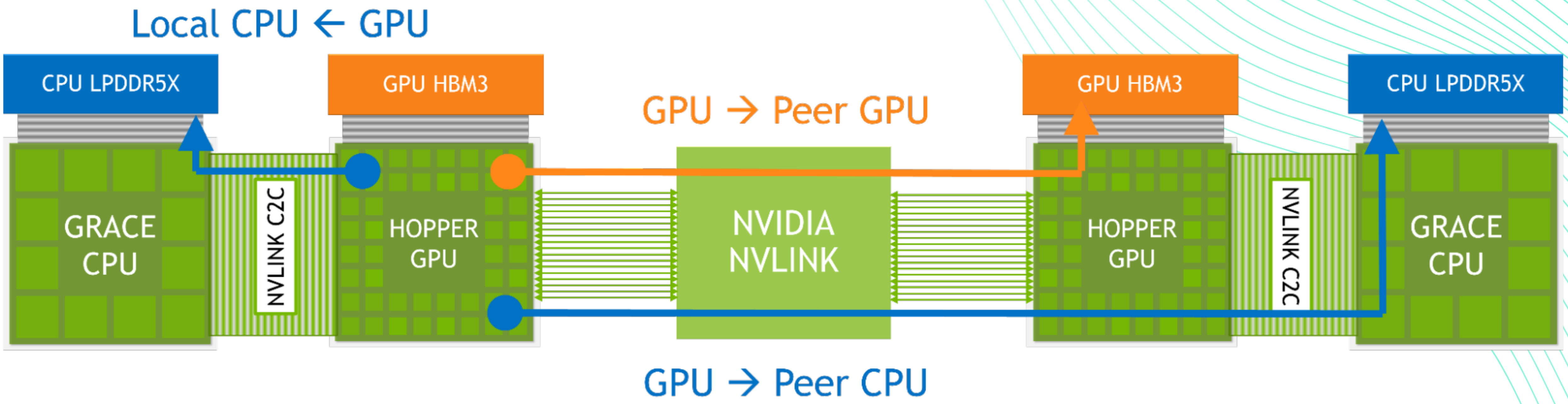
Source: NVIDIA Grace Hopper Superchip, https://www.boston.co.uk/blog/2023/07/10/nvidia-grace-hopper-superchip.aspx

Accelerators in HPC: Current trends
Unified memory address space – hardware coherency



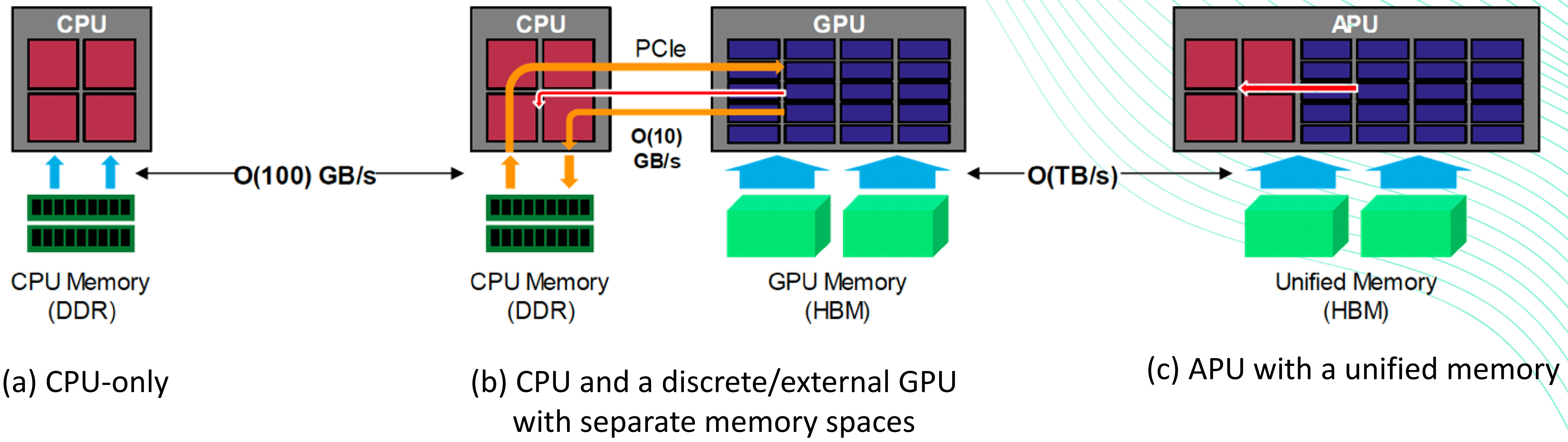
Source: NVIDIA Grace Hopper Superchip, https://www.boston.co.uk/blog/2023/07/10/nvidia-grace-hopper-superchip.aspx

Accelerators in HPC: Current trends
Unified memory address space



Source: NVIDIA Grace Hopper Superchip, https://www.boston.co.uk/blog/2023/07/10/nvidia-grace-hopper-superchip.aspx

Accelerators in HPC: Current trends
AMD APU



(a) CPU-only

(b) CPU and a discrete/external GPU with separate memory spaces

(c) APU with a unified memory

Example code and data movement/synchronization for (a) CPU-only, (b) CPU and a discrete/external GPU with separate memory spaces, and (c) APU with a unified memory.

Source: A. Smith et al., "Realizing the AMD Exascale Heterogeneous Processor Vision, ISCA, 2024, https://ieeexplore.ieee.org/document/10609636

EPICURE

Unlocking European-level HPC Support

GPU Architecture

Accelerators in HPC
Evolution of Graphics Processors

Till 90s

- VGA controllers used to accelerate some display functions

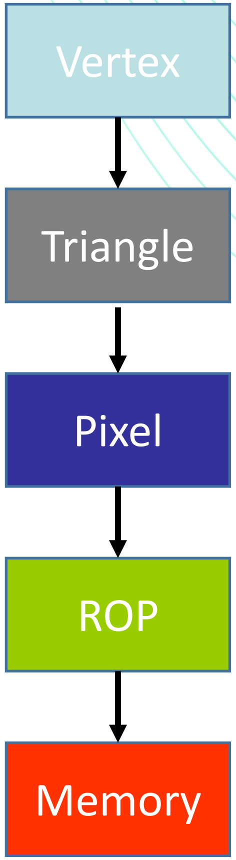
Mid 90s to mid 00s

- Fixed-function graphic accelerators for the OpenGL and DirectX APIs
  - Some GP-GPU capabilities on top of the interface
- 3D graphic: triangle setup & rasterization, texture mapping & shading

Modern GPUs

- Programmable multiprocessors (optimized for data-parallel ops)
  - OpenGL/DirectX and general purpose language
- Some fixed function hardware (texture, raster, ops, ….)

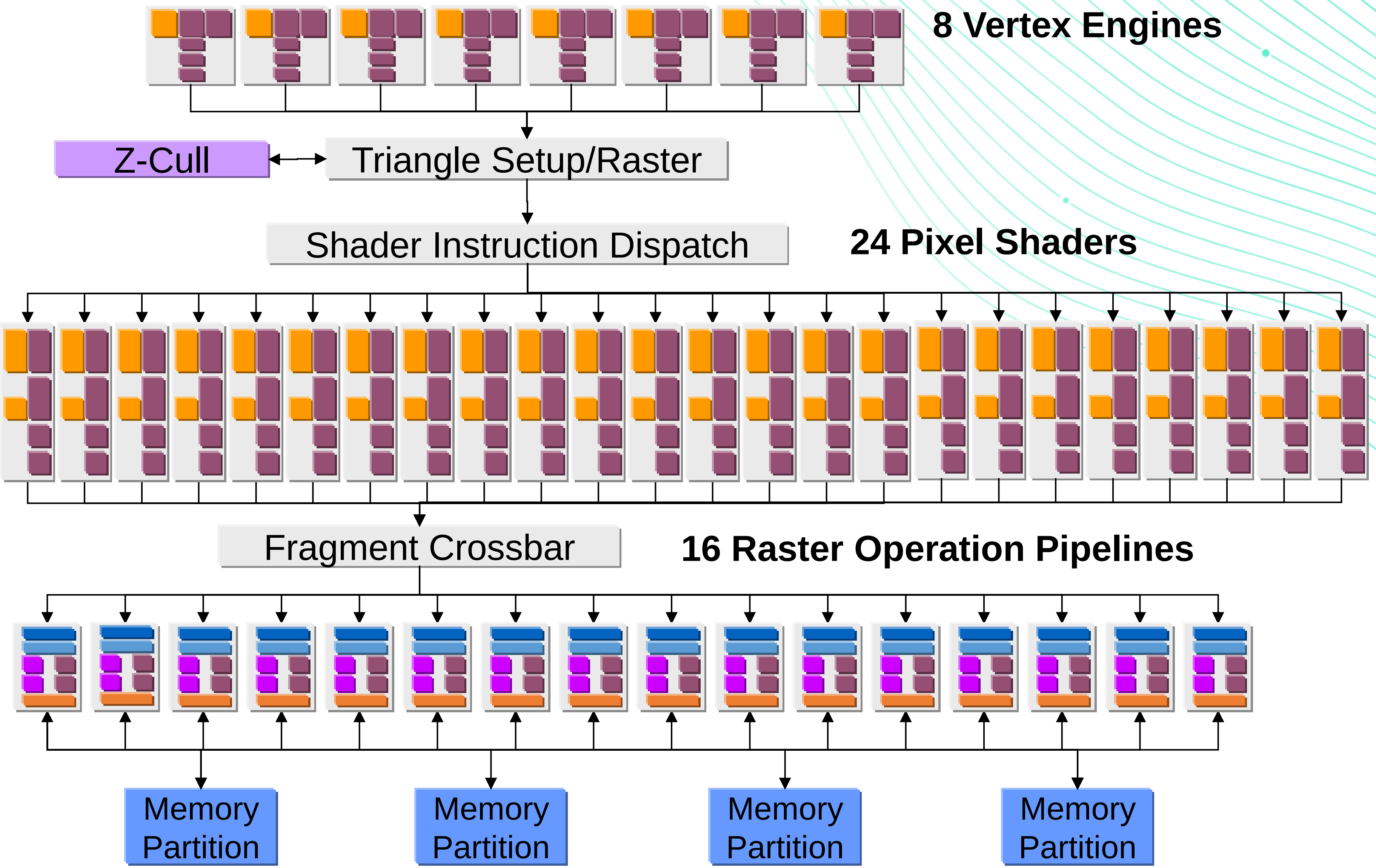
Graphic Pipeline (for last 20 years)



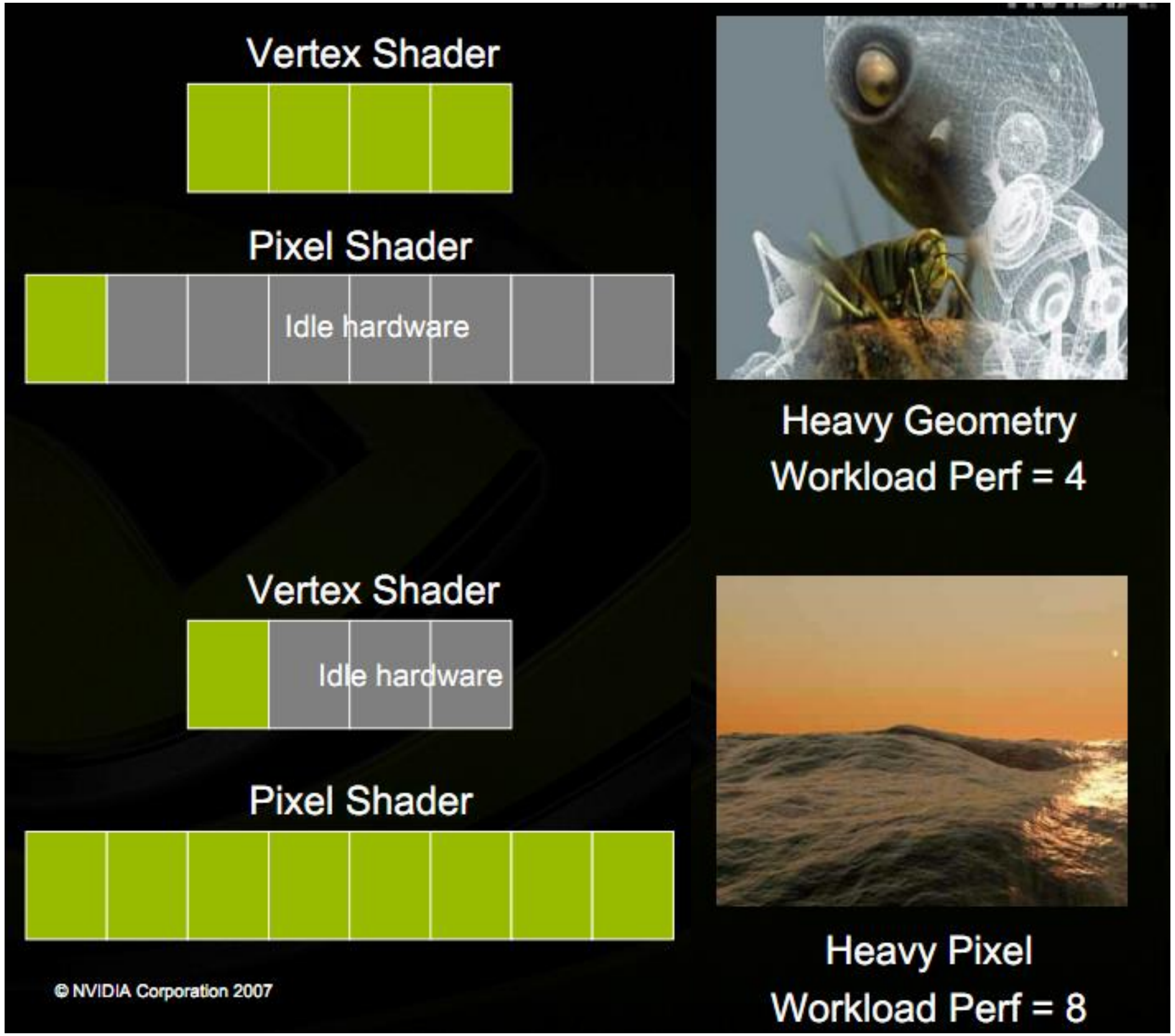
Vertex	T&L evolved to vertex shading
Triangle	Triangle, point, line - setup
Pixel	Flat shading, texturing, eventually, Pixel shading
ROP	Blending, Z-buffering, antialliasing
Memory	Wider and faster over years

Accelerators in HPC

Non-unified GPU Architecture GeForce 7800 GTX

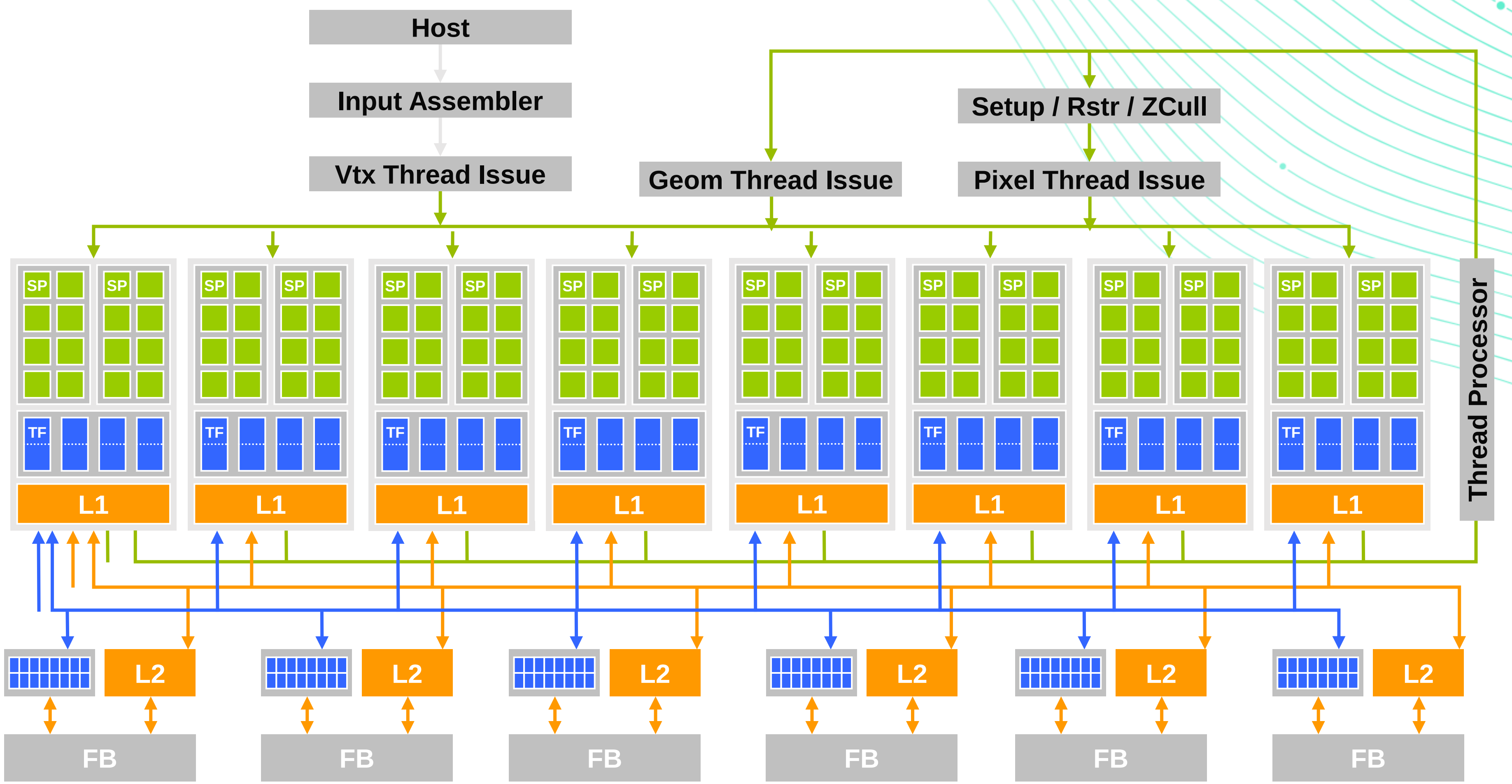


Accelerators in HPC
Why Unify Shader Processors?



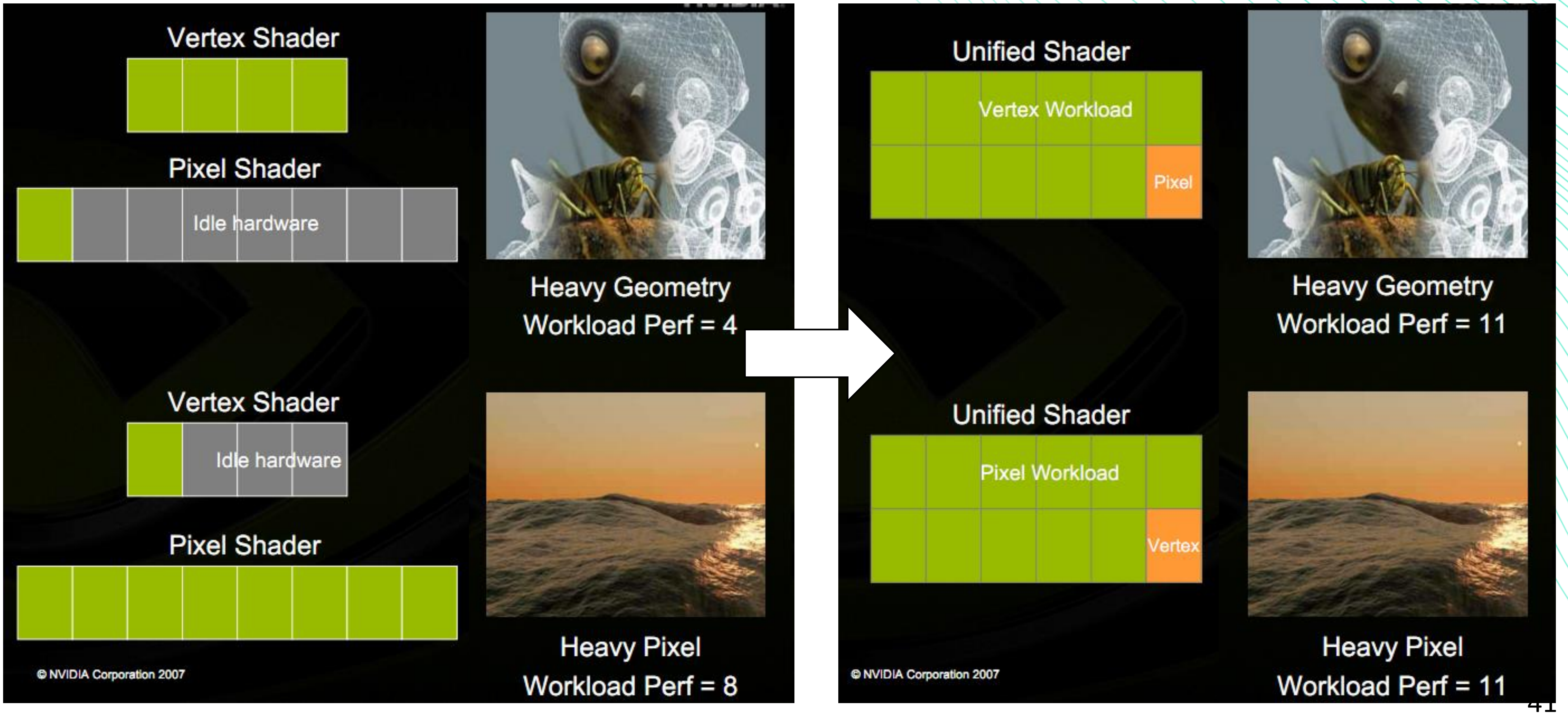
Accelerators in HPC

Unified Architecture G80 - Graphics Mode



The future of GPUs is programmable processing architecture built around the processor.

Accelerators in HPC
Why Unify Shader Processors?



Vertex Shader

Pixel Shader

Idle hardware

Heavy Geometry

Workload Perf = 4

Vertex Shader

Idle hardware

Pixel Shader

Heavy Pixel

Workload Perf = 8

© NVIDIA Corporation 2007

Unified Shader

Vertex Workload

Pixel

Heavy Geometry

Workload Perf = 11

Unified Shader

Pixel Workload

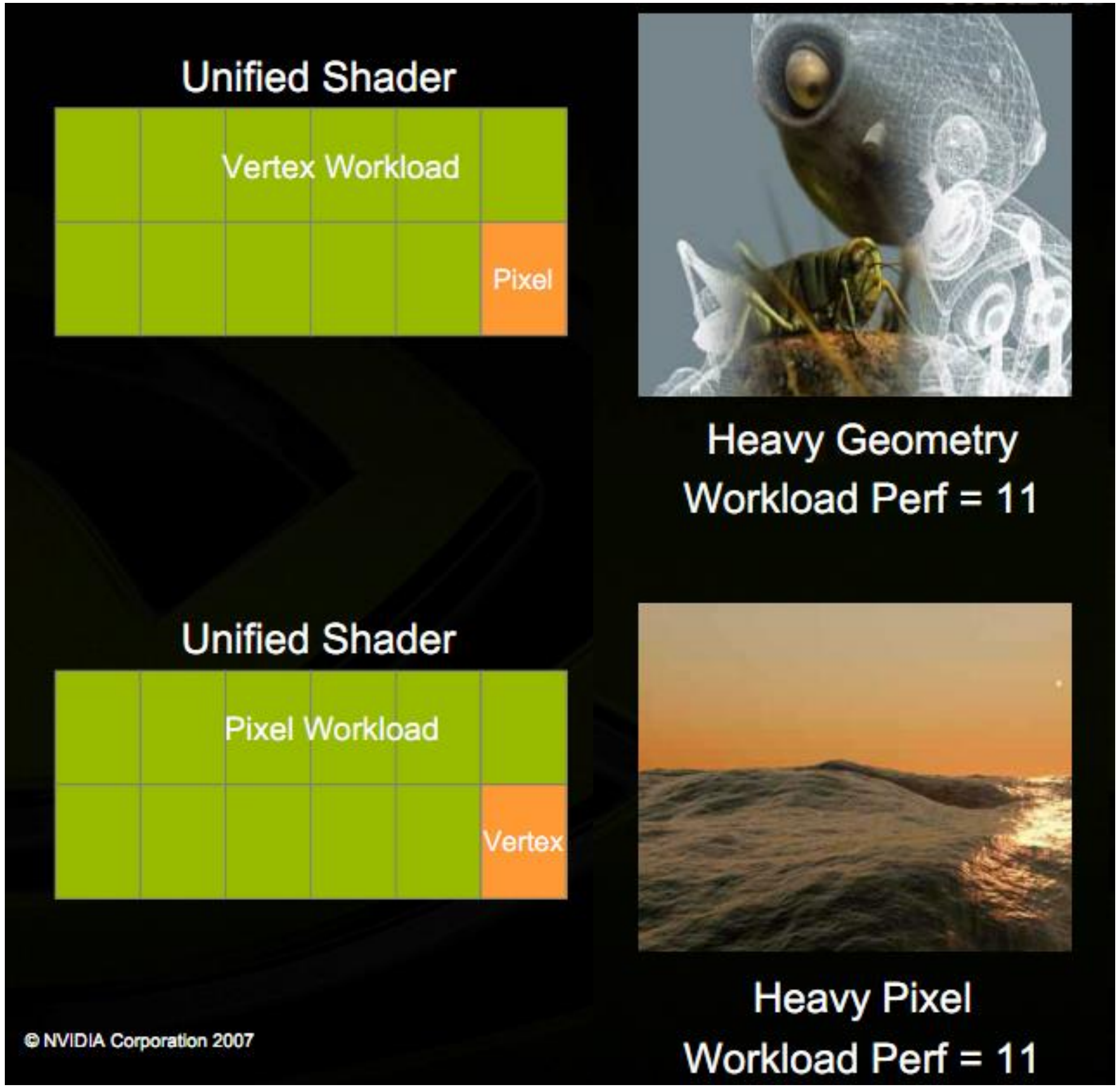
Vertex

Heavy Pixel

Workload Perf = 11

© NVIDIA Corporation 2007

Accelerators in HPC
Why Unify Shader Processors?



Unified Shader

Vertex Workload

Pixel

Heavy Geometry
Workload Perf = 11

Unified Shader

Pixel Workload

Vertex

Heavy Pixel
Workload Perf = 11

© NVIDIA Corporation 2007

Dynamic resource realocation

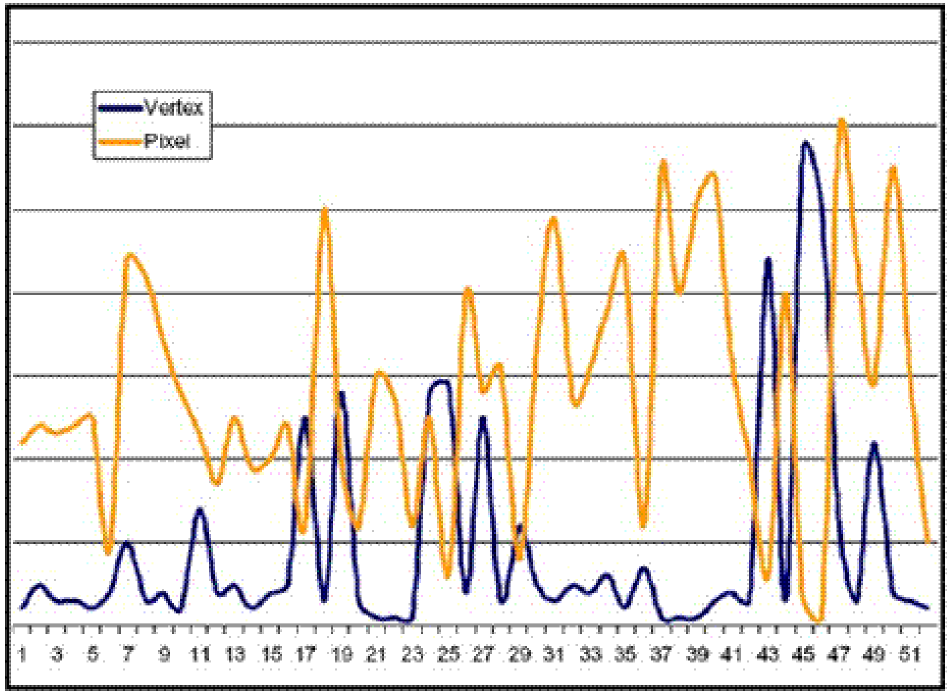
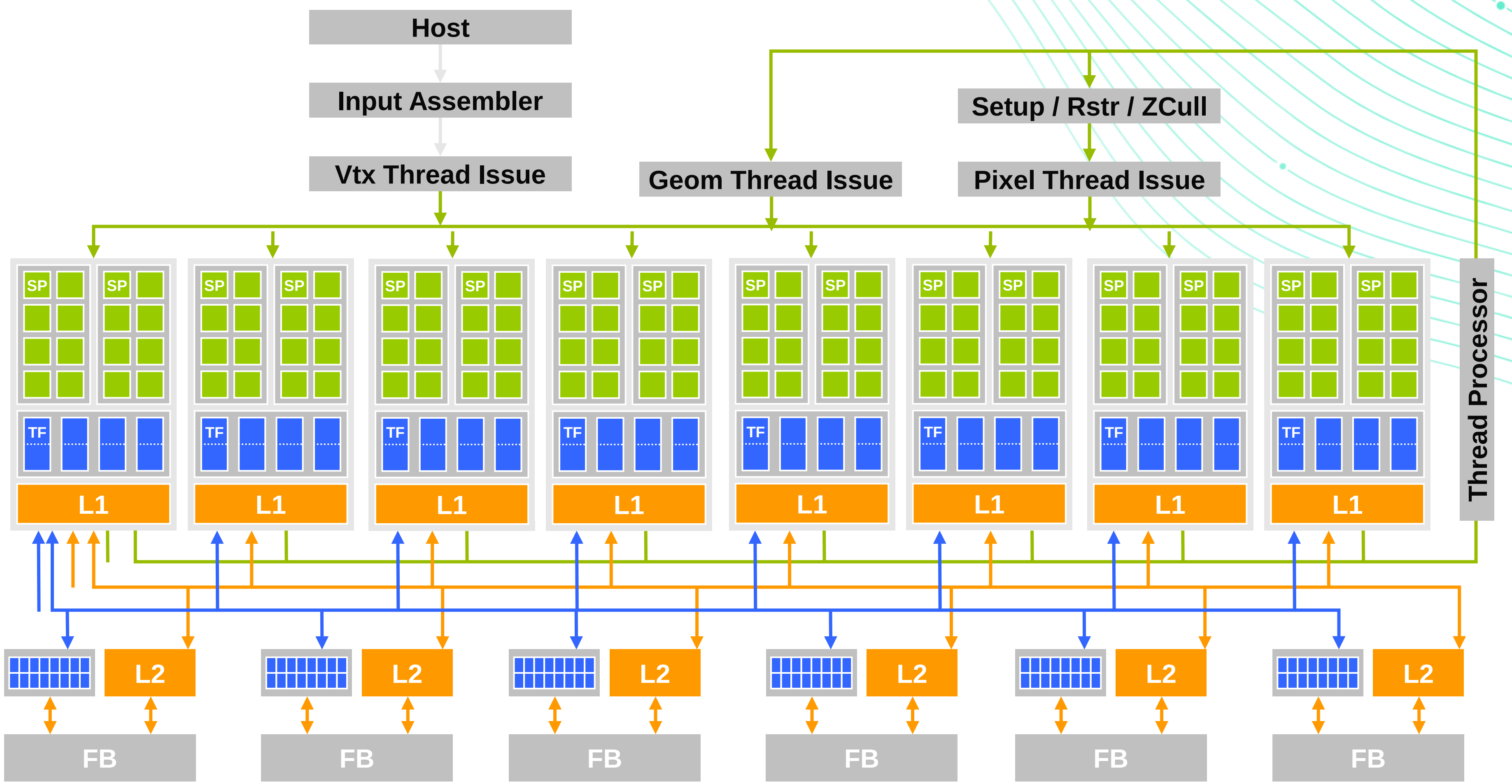


Figure 14. Characteristic pixel and vertex shader workload variation over time

Accelerators in HPC

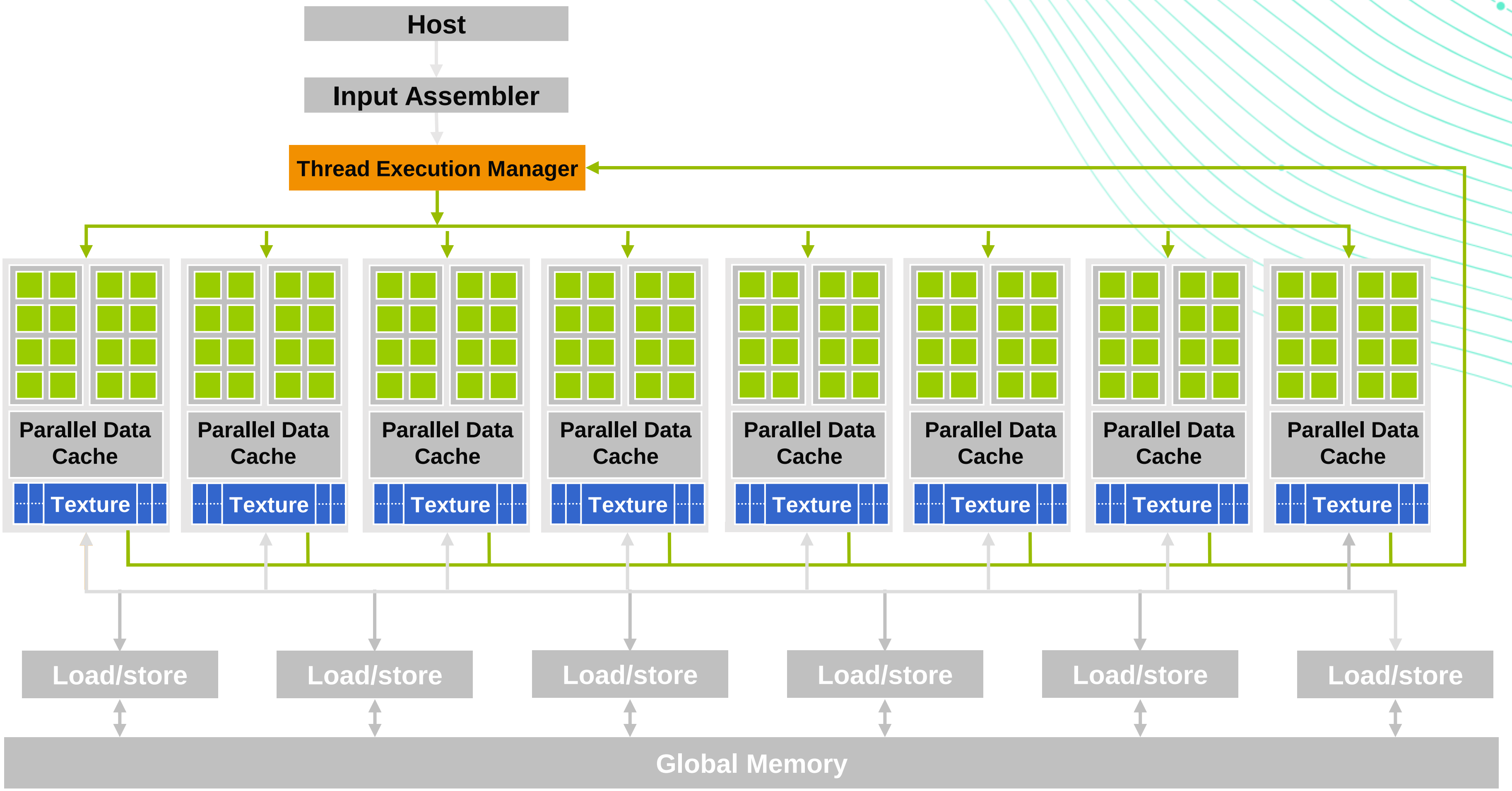
Unified Architecture G80 - Graphics Mode



The future of GPUs is programmable processing architecture built around the processor.

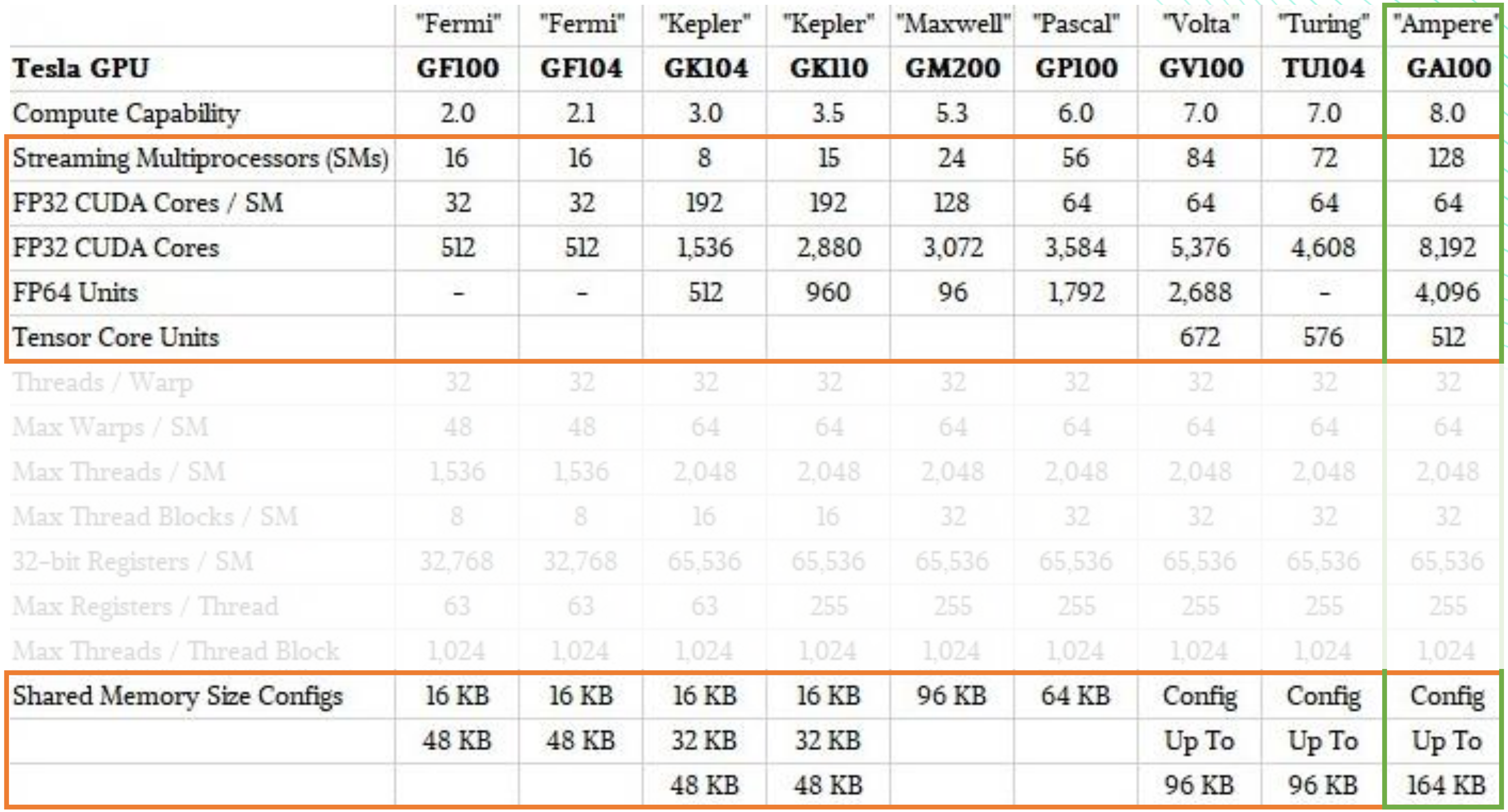
Accelerators in HPC

Unified Architecture G80 - Compute Mode



- processors execute computing threads
- new operating mode - HW interface for computing or accelerator

Evolution of NVIDIA GPU Accelerators in HPC



	"Fermi"	"Fermi"	"Kepler"	"Kepler"	"Maxwell"	"Pascal"	"Volta"	"Turing"	"Ampere"
Tesla GPU	**GF100**	**GF104**	**GK104**	**GK110**	**GM200**	**GP100**	**GV100**	**TU104**	**GA100**
Compute Capability	2.0	2.1	3.0	3.5	5.3	6.0	7.0	7.0	8.0
Streaming Multiprocessors (SMs)	16	16	8	15	24	56	84	72	128
FP32 CUDA Cores / SM	32	32	192	192	128	64	64	64	64
FP32 CUDA Cores	512	512	1,536	2,880	3,072	3,584	5,376	4,608	8,192
FP64 Units	-	-	512	960	96	1,792	2,688	-	4,096
Tensor Core Units							672	576	512
Threads / Warp	32	32	32	32	32	32	32	32	32
Max Warps / SM	48	48	64	64	64	64	64	64	64
Max Threads / SM	1,536	1,536	2,048	2,048	2,048	2,048	2,048	2,048	2,048
Max Thread Blocks / SM	8	8	16	16	32	32	32	32	32
32-bit Registers / SM	32,768	32,768	65,536	65,536	65,536	65,536	65,536	65,536	65,536
Max Registers / Thread	63	63	63	255	255	255	255	255	255
Max Threads / Thread Block	1,024	1,024	1,024	1,024	1,024	1,024	1,024	1,024	1,024
Shared Memory Size Configs	16 KB	16 KB	16 KB	16 KB	96 KB	64 KB	Config	Config	Config
	48 KB	48 KB	32 KB	32 KB			Up To	Up To	Up To
			48 KB	48 KB			96 KB	96 KB	164 KB

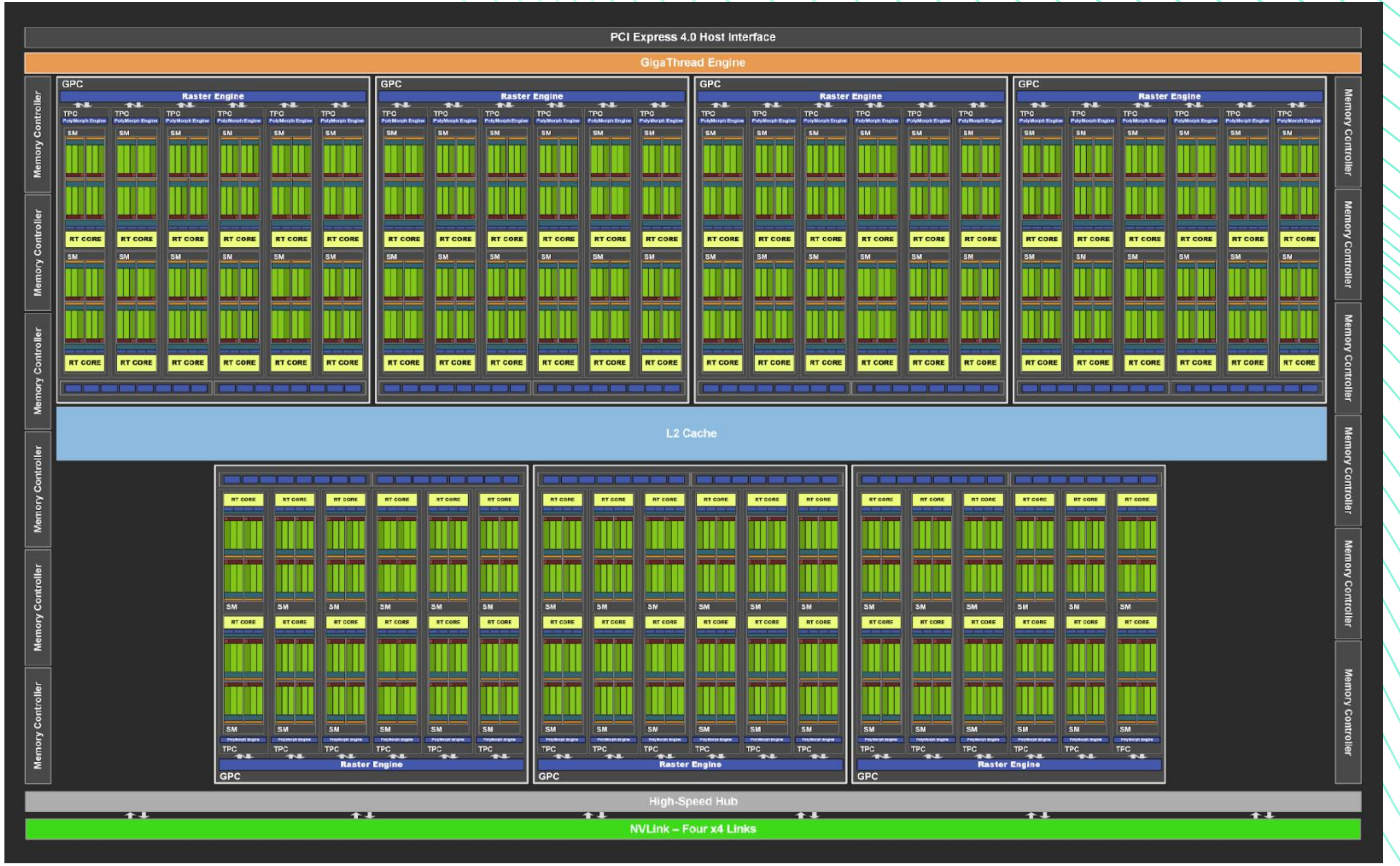
Performance

Fast local memory

https://www.nextplatform.com/2020/05/28/diving-deep-into-the-nvidia-ampere-gpu-architecture/

Accelerators in HPC
NVIDIA A40 Architecture

- Based on Ampere architecture GA102 chip designed for 3D graphics rather than scientific computing
  - GA102 GPU also features 168 FP64 units (two per SM),
  - **FP64 TFLOP rate is 1/64th the TFLOP rate of FP32 operations.**
  - the small number of FP64 hardware units are included to ensure any programs with FP64 code operate correctly



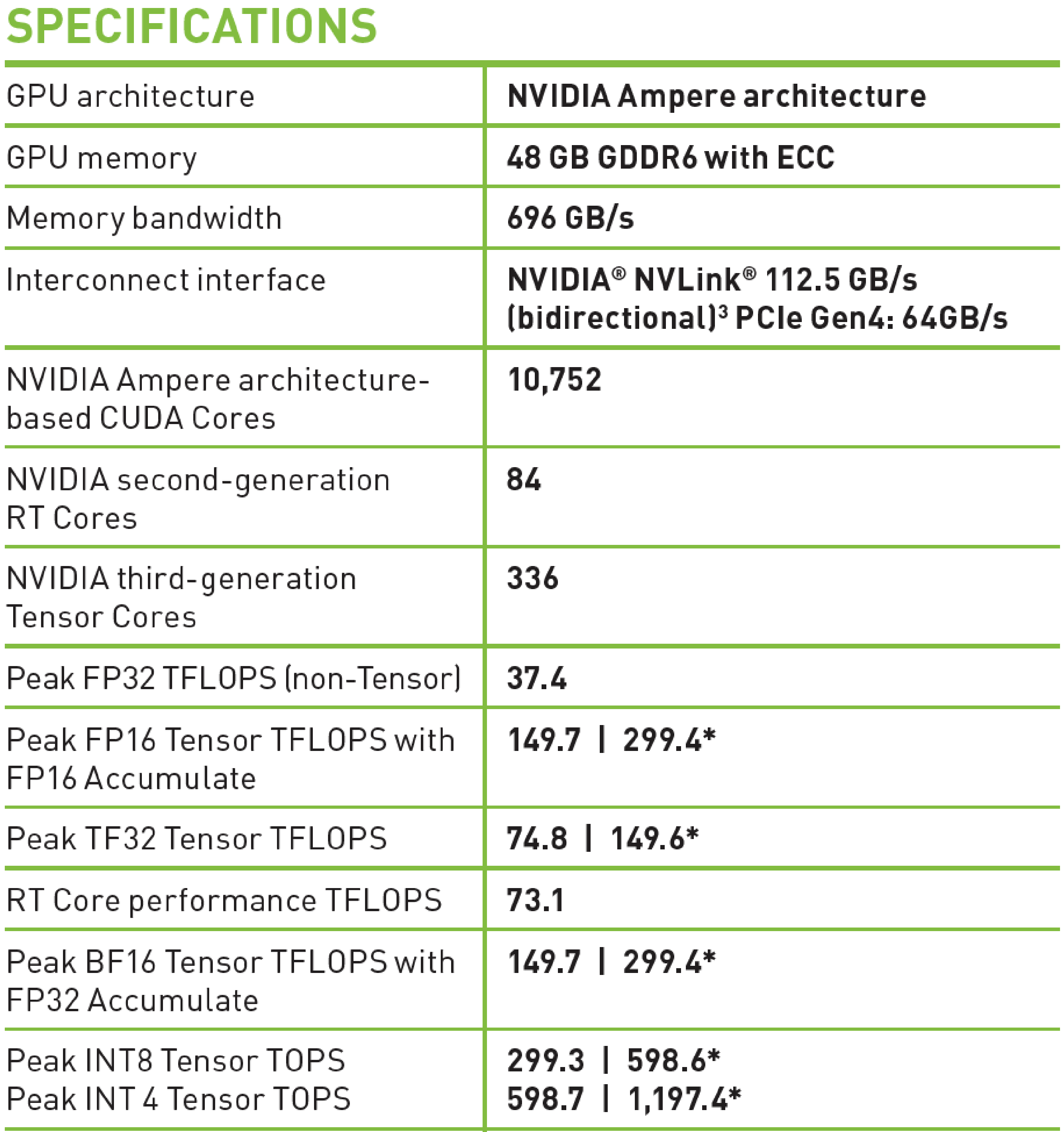
GA102 Full GPU with 84 SMs

https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf

Accelerators in HPC
NVIDIA A40 Architecture

- Based on Ampere architecture GA102 chip designed for 3D graphics rather than scientific computing
  - GA102 GPU also features 168 FP64 units (two per SM),
  - **FP64 TFLOP rate is 1/64th the TFLOP rate of FP32 operations.**
  - the small number of FP64 hardware units are included to ensure any programs with FP64 code operate correctly

https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf



SPECIFICATIONS

GPU architecture	**NVIDIA Ampere architecture**
GPU memory	**48 GB GDDR6 with ECC**
Memory bandwidth	**696 GB/s**
Interconnect interface	**NVIDIA® NVLink® 112.5 GB/s (bidirectional)[3] PCIe Gen4: 64GB/s**
NVIDIA Ampere architecture-based CUDA Cores	**10,752**
NVIDIA second-generation RT Cores	**84**
NVIDIA third-generation Tensor Cores	**336**
Peak FP32 TFLOPS (non-Tensor)	**37.4**
Peak FP16 Tensor TFLOPS with FP16 Accumulate	**149.7 | 299.4***
Peak TF32 Tensor TFLOPS	**74.8 | 149.6***
RT Core performance TFLOPS	**73.1**
Peak BF16 Tensor TFLOPS with FP32 Accumulate	**149.7 | 299.4***
Peak INT8 Tensor TOPS Peak INT 4 Tensor TOPS	**299.3 | 598.6*** **598.7 | 1,197.4***

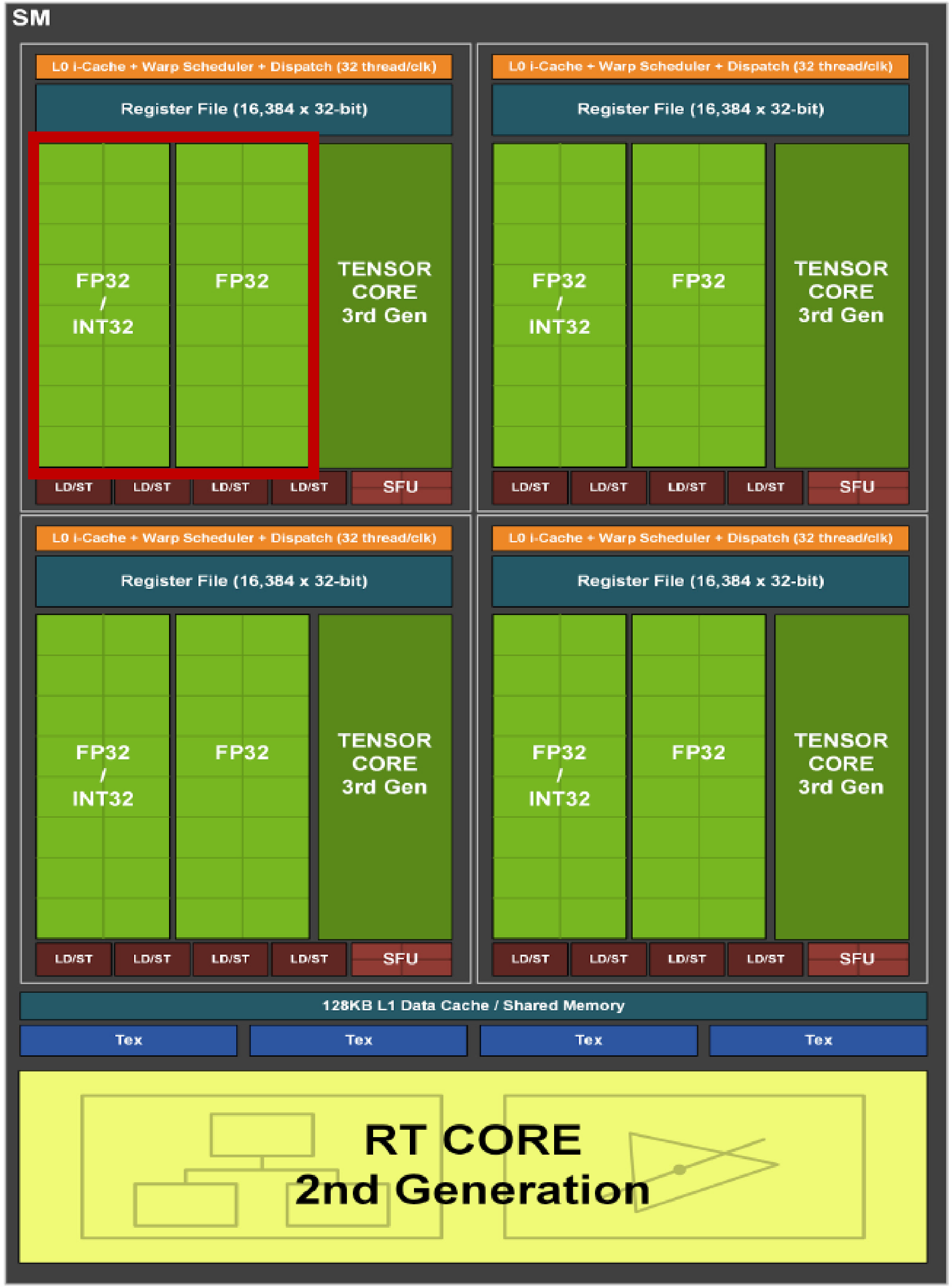
Accelerators in HPC
NVIDIA A40 Architecture

GA102 Streaming Multiprocessor (SM)

- includes four SM processing blocks (also called partitions)
  - 32 FP32 operations per clock, or
  - 16 FP32 and 16 INT32 operations per clock

- In compute mode, the GA102 SM will support the following configurations:
  - **128 KB L1 + 0 KB Shared Memory**
  - 120 KB L1 + 8 KB Shared Memory
  - 112 KB L1 + 16 KB Shared Memory
  - 96 KB L1 + 32 KB Shared Memory
  - 64 KB L1 + 64 KB Shared Memory
  - **28 KB L1 + 100 KB Shared Memory**

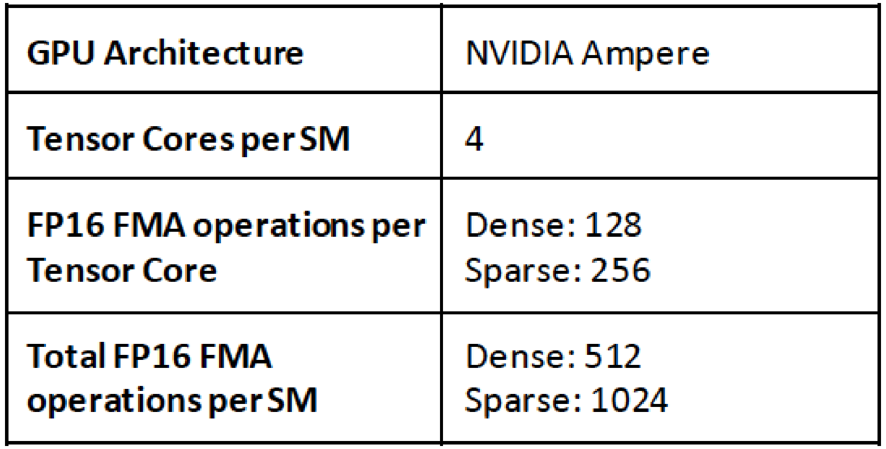
https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf


SM
L0 i-Cache + Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
FP32 / INT32
FP32
TENSOR CORE 3rd Gen
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex
RT CORE 2nd Generation

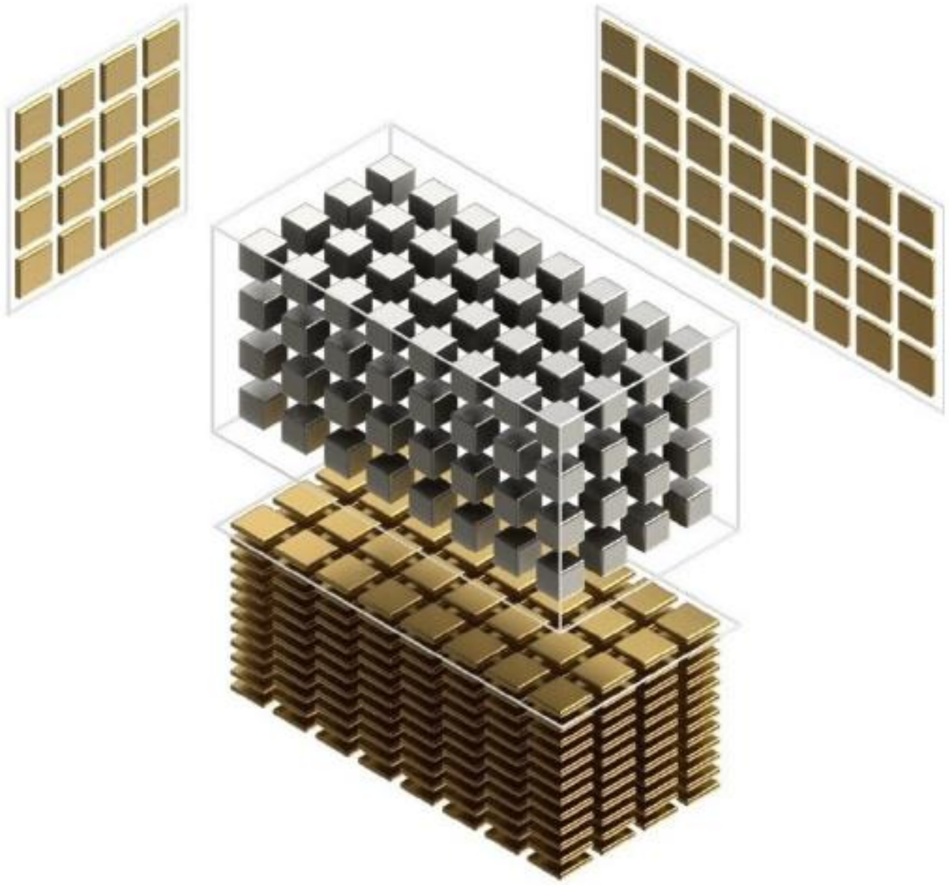
Accelerators in HPC
NVIDIA A40 Architecture

Tensor Cores

- specialized execution units designed specifically for performing the tensor / matrix operations that are the core compute function used in Deep Learning
- accelerate the matrix-matrix multiplication



GPU Architecture	NVIDIA Ampere
Tensor Cores per SM	4
FP16 FMA operations per Tensor Core	Dense: 128 Sparse: 256
Total FP16 FMA operations per SM	Dense: 512 Sparse: 1024

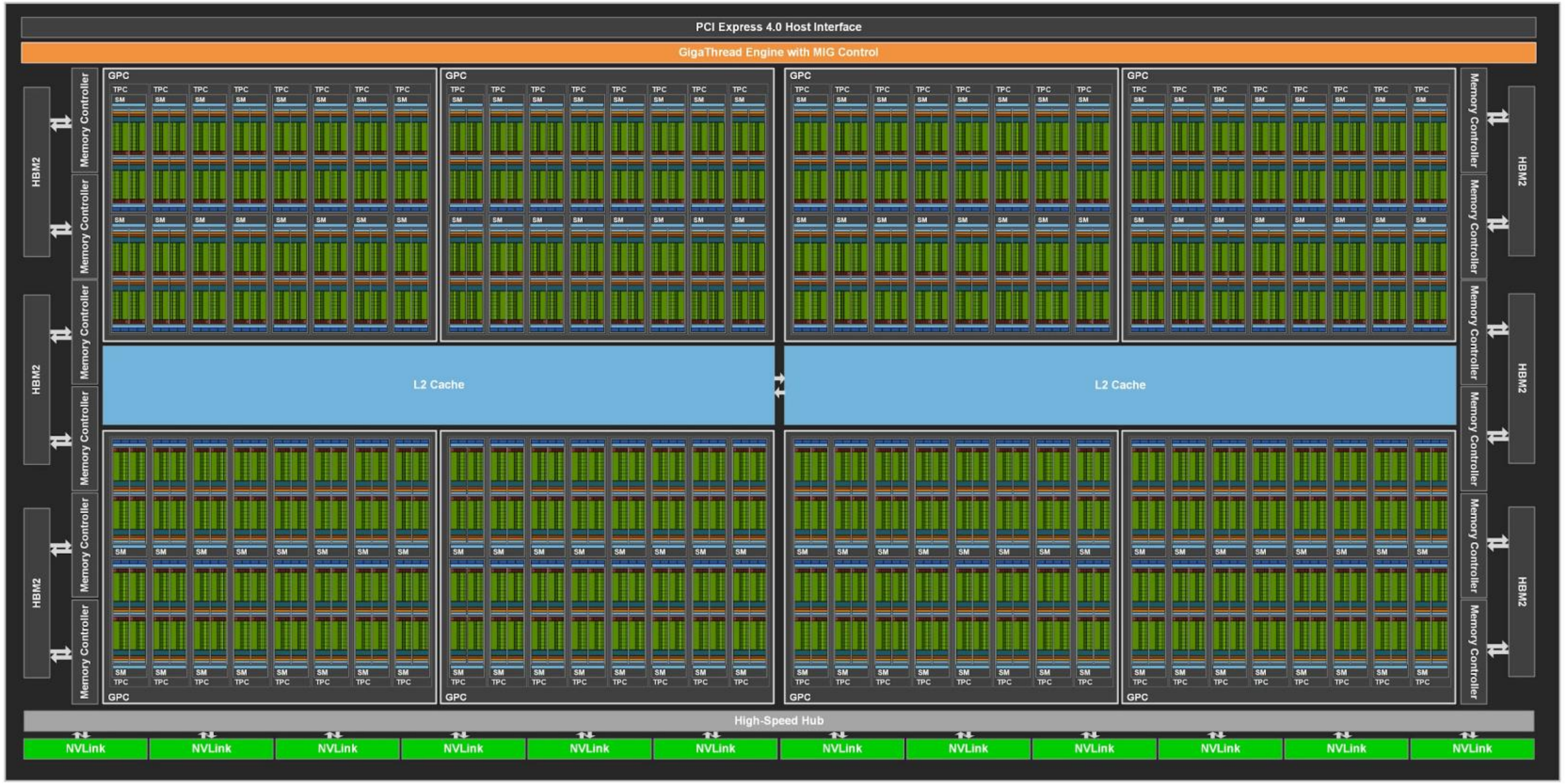


Ampere architecture tensor core

https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf

Accelerators in HPC
NVIDIA A100 Architecture

- Based on Ampere architecture GA100 chip designed for scientific computing
- The NVIDIA A100 GPU implementation of the GA100 GPU includes the following units:
  - 108 Streaming Multiprocessors (SMs)
  - 6912 FP32 CUDA Cores per GPU
    - 64 FP32 CUDA Cores per SM
  - 432 Third-generation Tensor Cores per GPU
    - 4 Third-generation Tensor Cores per SM
  - 5 HBM2 stacks,
    - 10x 512-bit Memory Controllers


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

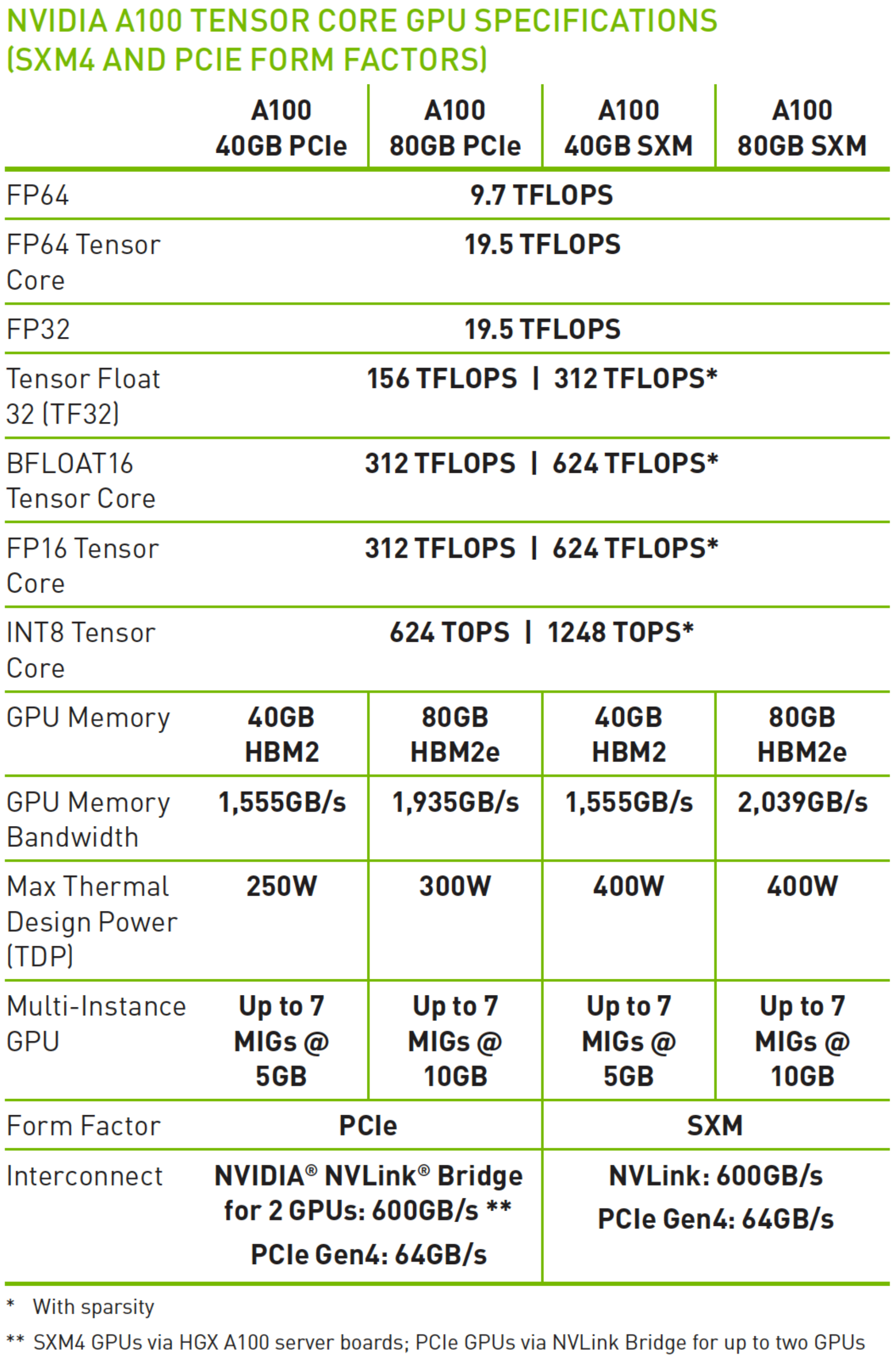
GA100 Full GPU with 128 SMs

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

Accelerators in HPC
NVIDIA A100 Architecture

- Based on Ampere architecture GA100 chip designed for scientific computing
- The NVIDIA A100 GPU implementation of the GA100 GPU includes the following units:
  - 108 Streaming Multiprocessors (SMs)
  - 6912 FP32 CUDA Cores per GPU
    - 64 FP32 CUDA Cores per SM
  - 432 Third-generation Tensor Cores per GPU
    - 4 Third-generation Tensor Cores per SM
  - 5 HBM2 stacks,
    - 10x 512-bit Memory Controllers

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



NVIDIA A100 TENSOR CORE GPU SPECIFICATIONS (SXM4 AND PCIE FORM FACTORS)

	A100 40GB PCIe	A100 80GB PCIe	A100 40GB SXM	A100 80GB SXM
FP64	9.7 TFLOPS			
FP64 Tensor Core	19.5 TFLOPS			
FP32	19.5 TFLOPS			
Tensor Float 32 (TF32)	156 TFLOPS | 312 TFLOPS*			
BFLOAT16 Tensor Core	312 TFLOPS | 624 TFLOPS*			
FP16 Tensor Core	312 TFLOPS | 624 TFLOPS*			
INT8 Tensor Core	624 TOPS | 1248 TOPS*			
GPU Memory	40GB HBM2	80GB HBM2e	40GB HBM2	80GB HBM2e
GPU Memory Bandwidth	1,555GB/s	1,935GB/s	1,555GB/s	2,039GB/s
Max Thermal Design Power (TDP)	250W	300W	400W	400W
Multi-Instance GPU	Up to 7 MIGs @ 5GB	Up to 7 MIGs @ 10GB	Up to 7 MIGs @ 5GB	Up to 7 MIGs @ 10GB
Form Factor	PCIe		SXM	
Interconnect	NVIDIA® NVLink® Bridge for 2 GPUs: 600GB/s ** PCIe Gen4: 64GB/s		NVLink: 600GB/s PCIe Gen4: 64GB/s	

* With sparsity

** SXM4 GPUs via HGX A100 server boards; PCIe GPUs via NVLink Bridge for up to two GPUs

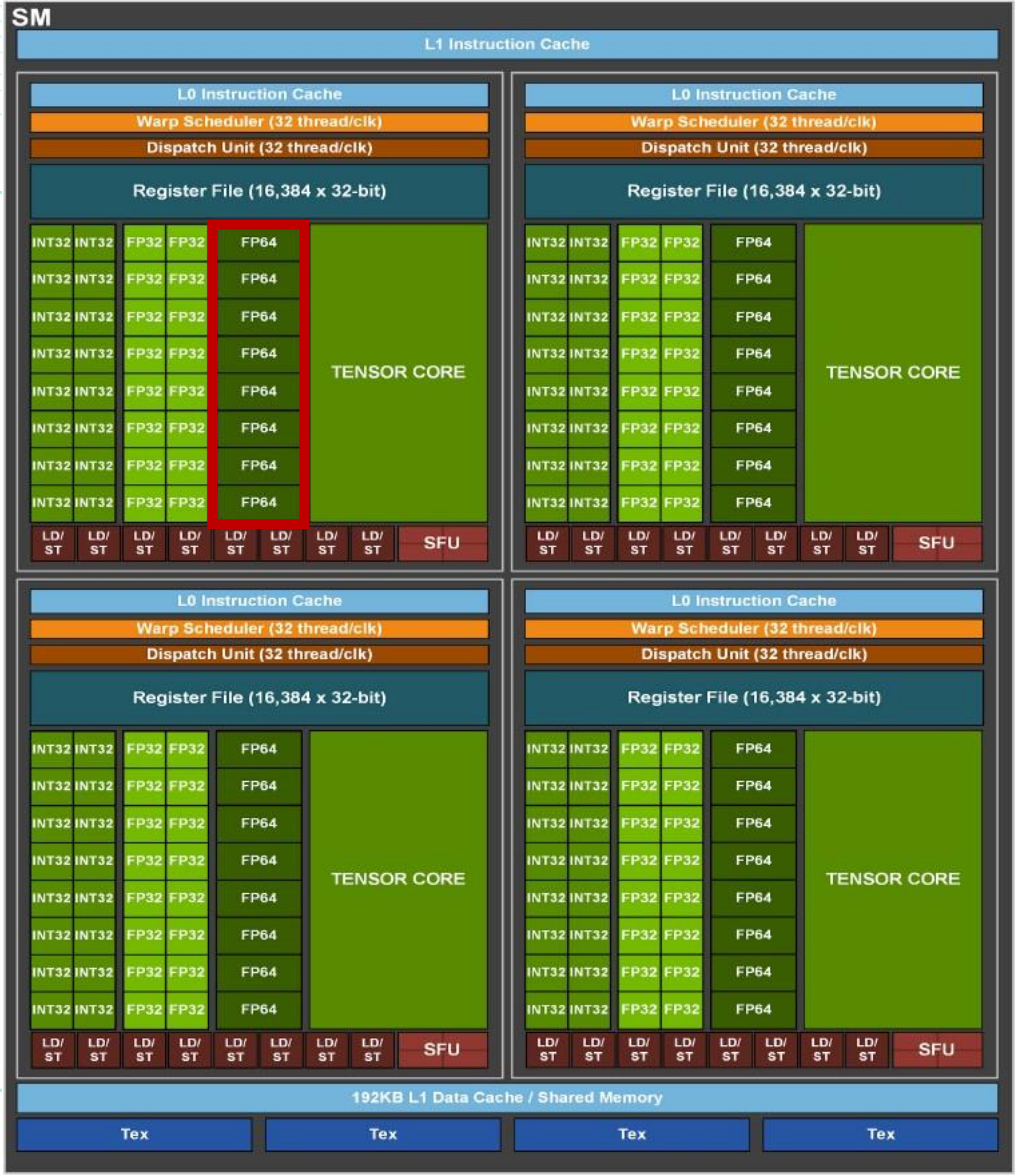
Accelerators in HPC
NVIDIA A100 Architecture

GA100 Streaming Multiprocessor (SM)

- includes four SM processing blocks (also called partitions)
  - 16 FP32 operations per clock,
  - 16 INT32 operations per clock, and
  - 8 FP64 operations per clock,

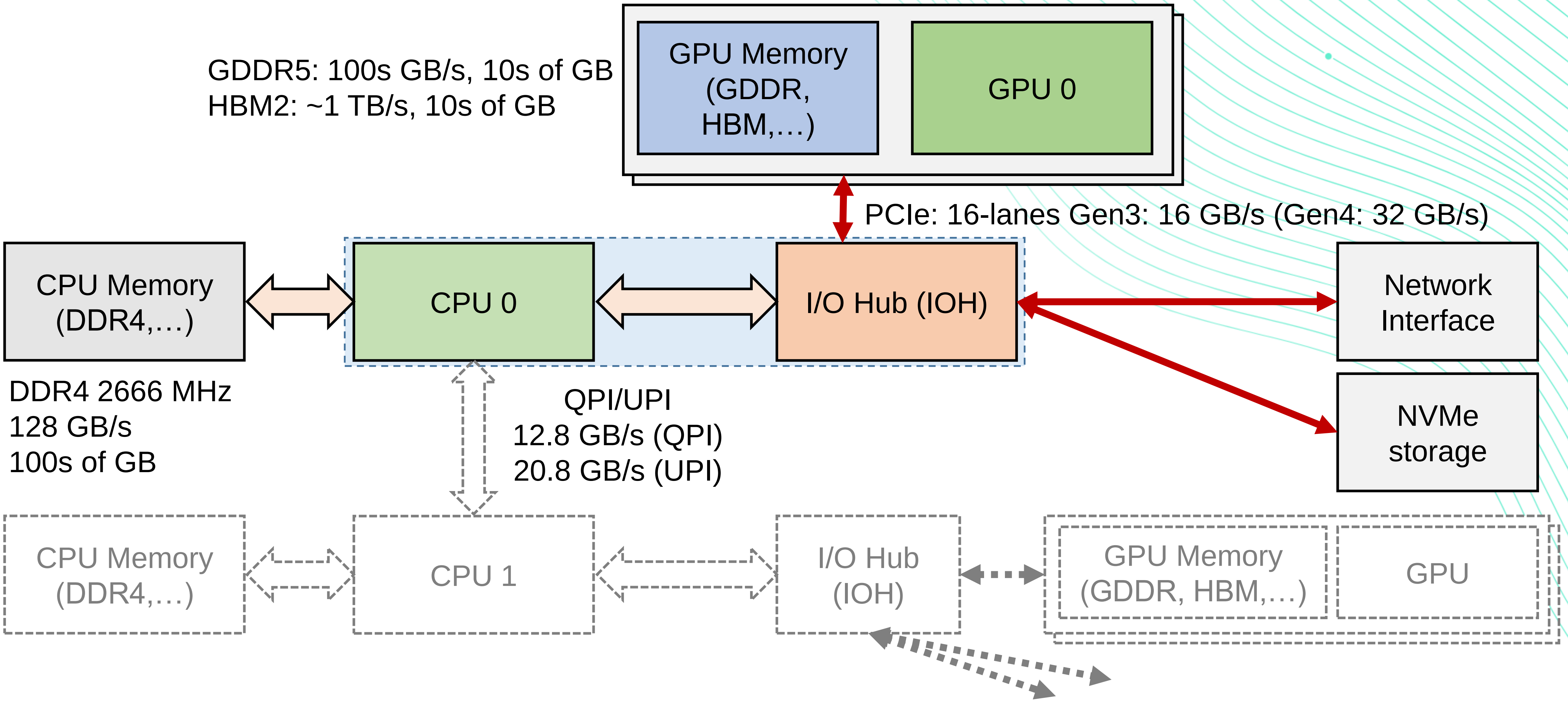
- FP64 Tensor Core operations running 2x faster DFMA operations
- 192 KB of combined shared memory and L1 data cache

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



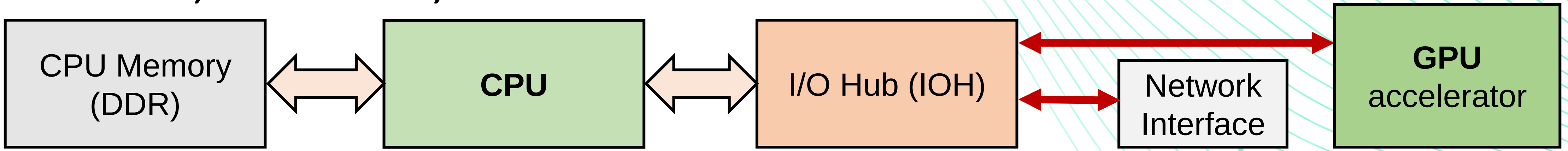
SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE
LD/ST
SFU
192KB L1 Data Cache / Shared Memory
Tex

General Architecture of GPU Accelerated Compute Node

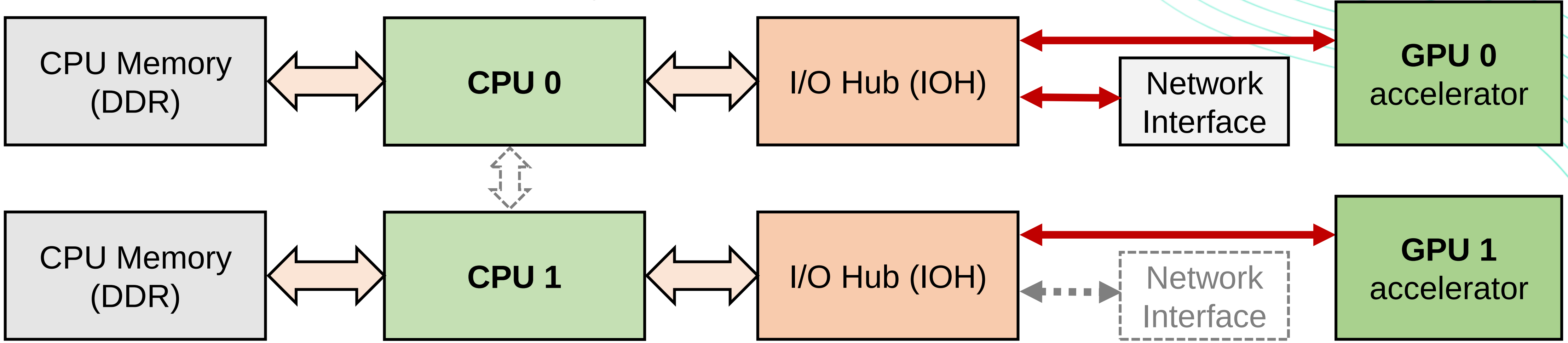


Evolution of GPU Accelerated nodes

ORNL Titan, CSCS Piz Daint, ….

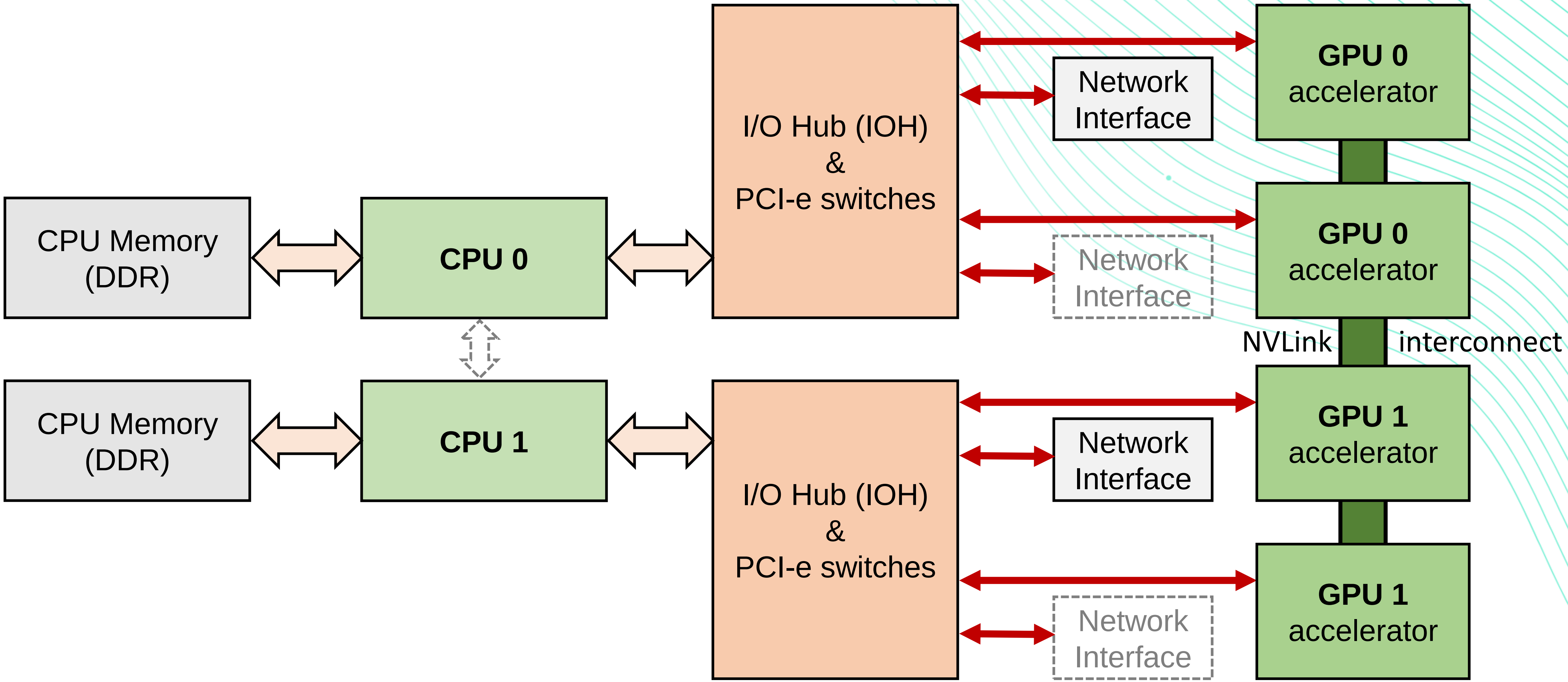


Dual socket machines with one GPU per CPU



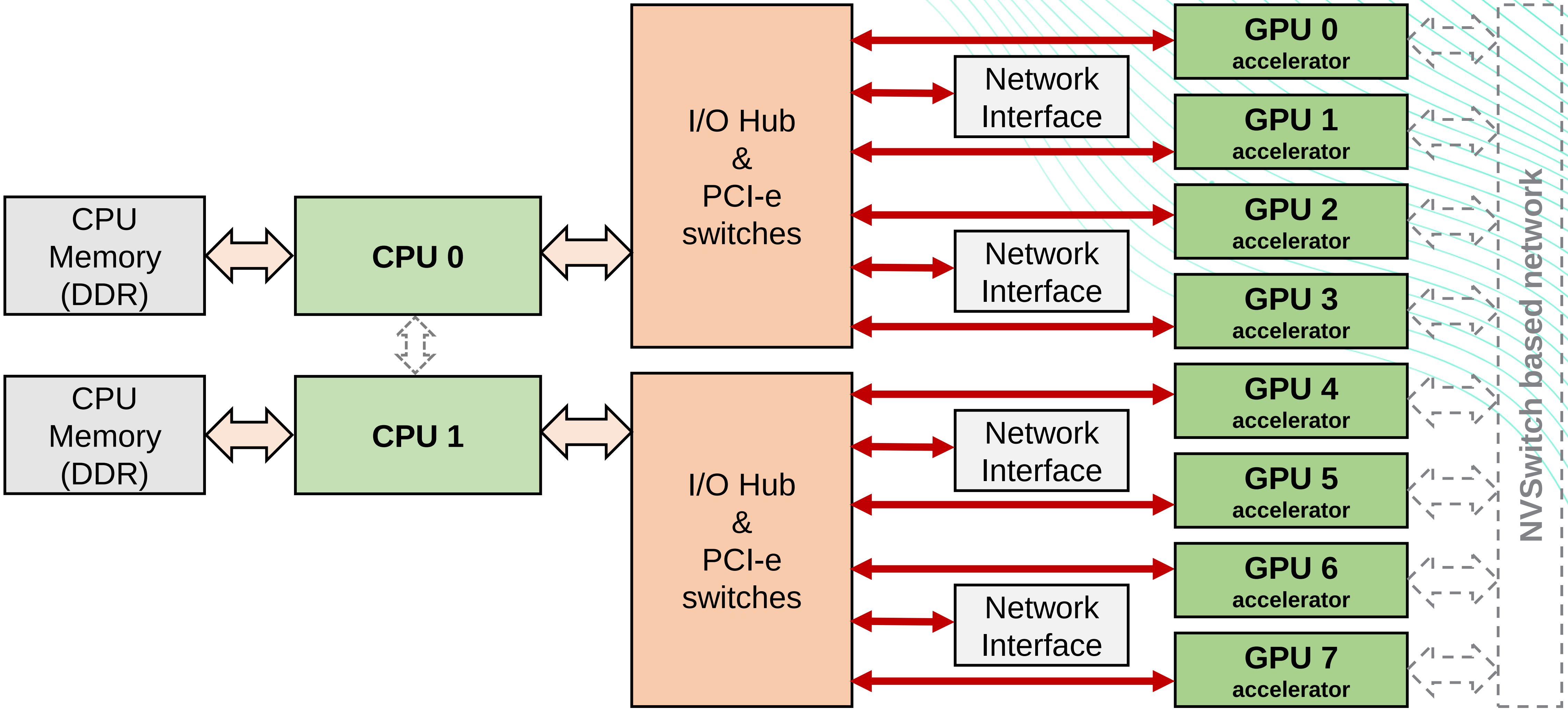
- 1:1 ratio between processors and accelerators
- GPUs used to have relativelly small amount of memory

Evolution of GPU Accelerated nodes



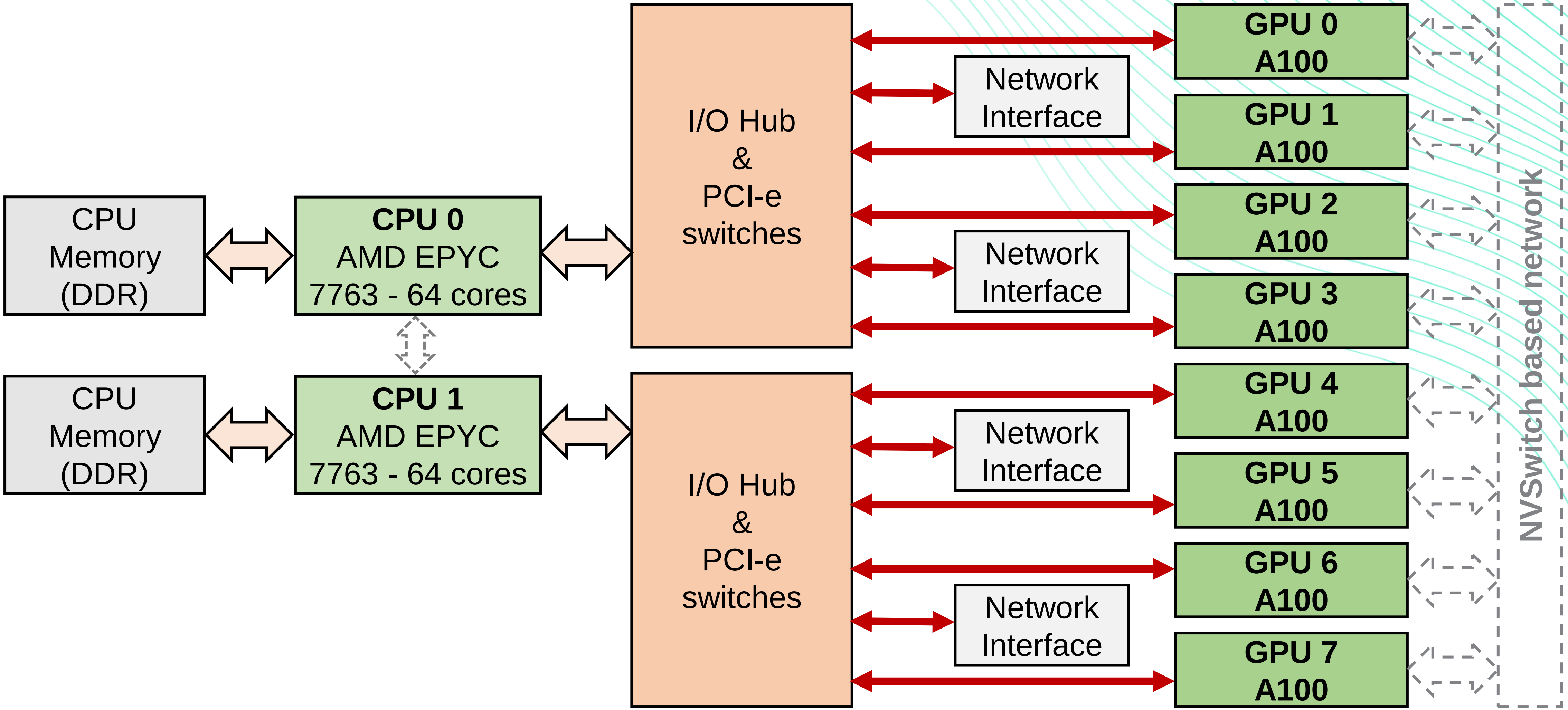
- Dominant architecture in todays systems: 1:2 ratio between processors and accelerators.
- A100 GPUs now have 40 or 80GB of memory – 160 - 320 GB of GPU memory in total

Evolution of GPU Accelerated nodes



- Fat GPU nodes contain 8 GPUs: **1:4 ratio between processors and accelerators**.
- 40 or 80 GB per GPU --> 320 or 640 GB of GPU memory in total which can be shared among GPUs

Karolina GPU Accelerated nodes



- Karolina GPU nodes contain 8 GPUs: **1:4 ratio between processors and accelerators**.
- 40 GB per GPU --> 320 GB of GPU memory in total which can be shared among GPUs

Compute node evaluation of the Karolina GPU Accelerated nodes

```
$ nvidia-smi topo -m

        GPU0  GPU1  GPU2  GPU3  GPU4  GPU5  GPU6  GPU7  mlx5_0  mlx5_1  mlx5_2  mlx5_3  CPU Affinity  NUMA Affinity
GPU0     X    NV12  NV12  NV12  NV12  NV12  NV12  NV12  SYS     PXB     SYS     SYS     48-63         3
GPU1    NV12   X    NV12  NV12  NV12  NV12  NV12  NV12  SYS     PXB     SYS     SYS     48-63         3
GPU2    NV12  NV12   X    NV12  NV12  NV12  NV12  NV12  PXB     SYS     SYS     SYS     16-31         1
GPU3    NV12  NV12  NV12   X    NV12  NV12  NV12  NV12  PXB     SYS     SYS     SYS     16-31         1
GPU4    NV12  NV12  NV12  NV12   X    NV12  NV12  NV12  SYS     SYS     SYS     PXB     112-127       7
GPU5    NV12  NV12  NV12  NV12  NV12   X    NV12  NV12  SYS     SYS     SYS     PXB     112-127       7
GPU6    NV12  NV12  NV12  NV12  NV12  NV12   X    NV12  SYS     SYS     PXB     SYS     80-95         5
GPU7    NV12  NV12  NV12  NV12  NV12  NV12  NV12   X    SYS     SYS     PXB     SYS     80-95         5
mlx5_0  SYS   SYS   PXB   PXB   SYS   SYS   SYS   SYS    X      SYS     SYS     SYS
mlx5_1  PXB   PXB   SYS   SYS   SYS   SYS   SYS   SYS   SYS      X      SYS     SYS
mlx5_2  SYS   SYS   SYS   SYS   SYS   SYS   PXB   PXB   SYS     SYS      X      SYS
mlx5_3  SYS   SYS   SYS   SYS   PXB   PXB   SYS   SYS   SYS     SYS     SYS      X

Legend:

  X    = Self
  SYS  = Connection traversing PCIe as well as the SMP interconnect between NUMA nodes (e.g., QPI/UPI)
  NODE = Connection traversing PCIe as well as the interconnect between PCIe Host Bridges within a NUMA node
  PHB  = Connection traversing PCIe as well as a PCIe Host Bridge (typically the CPU)
  PXB  = Connection traversing multiple PCIe bridges (without traversing the PCIe Host Bridge)
  PIX  = Connection traversing at most a single PCIe bridge
  NV#  = Connection traversing a bonded set of # NVLinks
```

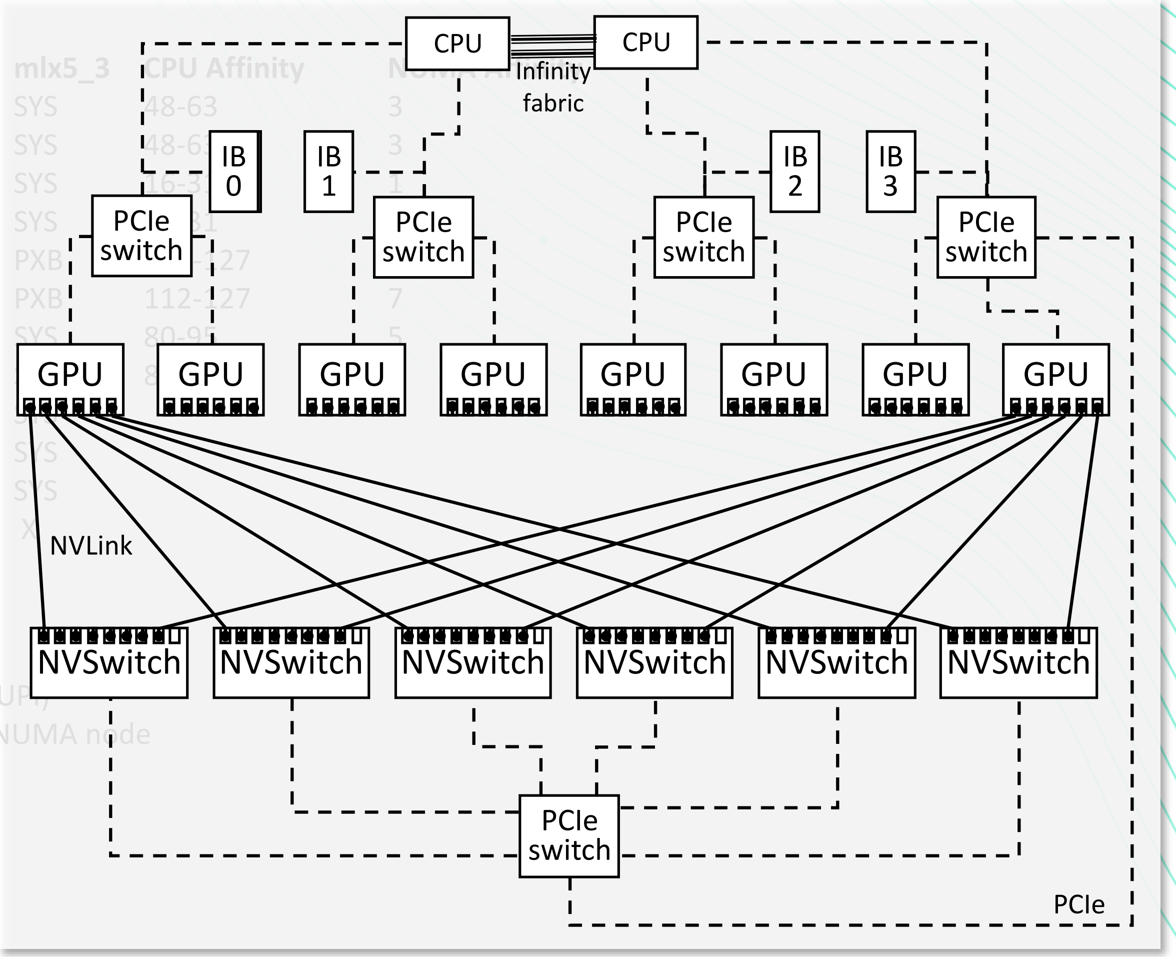
Note:

CPU: 2 x AMD Zen 3 EPYC™ 7763, 2.45 GHz and GPU: 8x NVIDIA A100 SXM4 GPUs

Compute node evaluation of the Karolina GPU Accelerated nodes

$ nvidia-smi topo -m

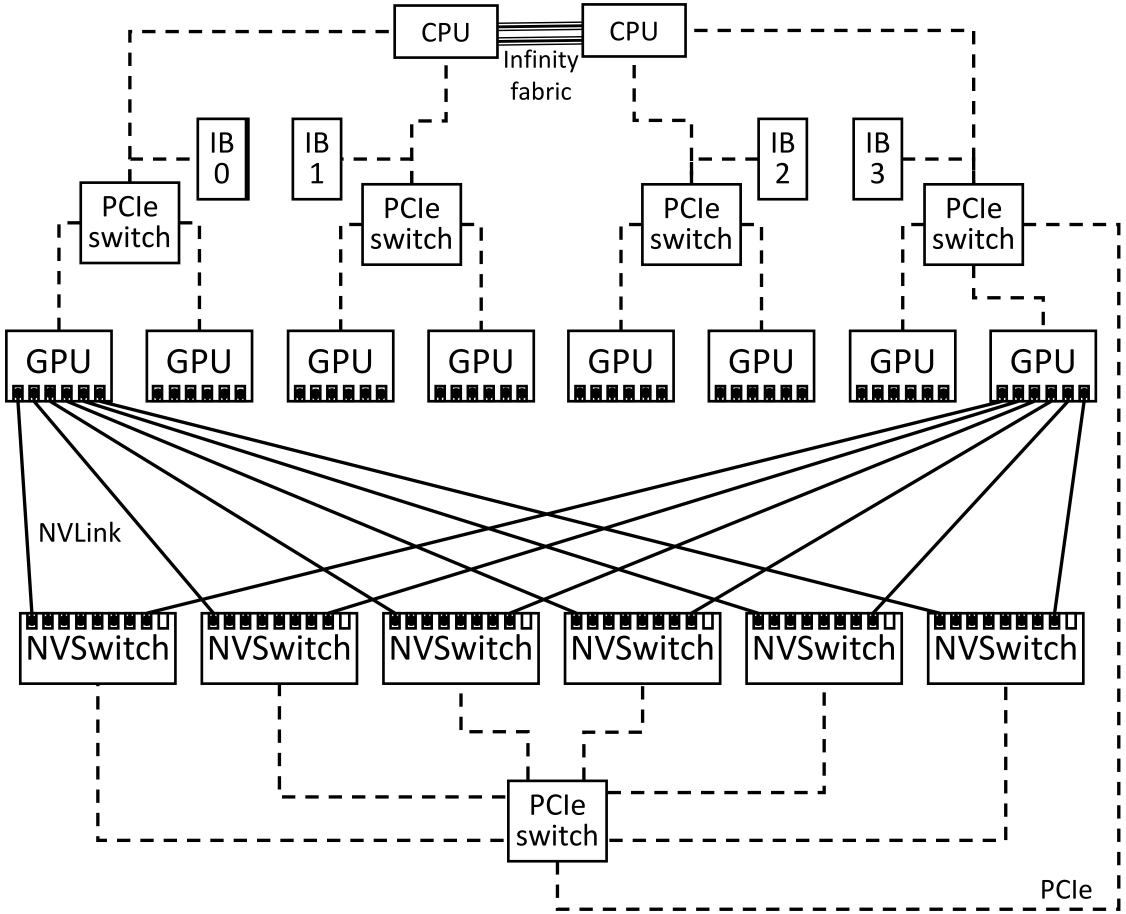
```
        GPU0  GPU1  GPU2  GPU3  GPU4  GPU5  GPU6  GPU7   mlx5_0  mlx5_1  mlx5_2
GPU0     X    NV12  NV12  NV12  NV12  NV12  NV12  NV12   SYS     PXB     SYS
GPU1    NV12   X    NV12  NV12  NV12  NV12  NV12  NV12   SYS     PXB     SYS
GPU2    NV12  NV12   X    NV12  NV12  NV12  NV12  NV12   PXB     SYS     SYS
GPU3    NV12  NV12  NV12   X    NV12  NV12  NV12  NV12   PXB     SYS     SYS
GPU4    NV12  NV12  NV12  NV12   X    NV12  NV12  NV12   SYS     SYS     SYS
GPU5    NV12  NV12  NV12  NV12  NV12   X    NV12  NV12   SYS     SYS     SYS
GPU6    NV12  NV12  NV12  NV12  NV12  NV12   X    NV12   SYS     SYS     PXB
GPU7    NV12  NV12  NV12  NV12  NV12  NV12  NV12   X     SYS     SYS     PXB
mlx5_0  SYS   SYS   PXB   PXB   SYS   SYS   SYS   SYS     X      SYS     SYS
mlx5_1  PXB   PXB   SYS   SYS   SYS   SYS   SYS   SYS    SYS      X      SYS
mlx5_2  SYS   SYS   SYS   SYS   SYS   SYS   PXB   PXB    SYS     SYS      X
mlx5_3  SYS   SYS   SYS   SYS   PXB   PXB   SYS   SYS    SYS     SYS     SYS

Legend:

  X    = Self
  SYS  = Connection traversing PCIe as well as the SMP interconnect between NUMA nodes (e.g., QPI/UPI)
  NODE = Connection traversing PCIe as well as the interconnect between PCIe Host Bridges within a NUMA node
  PHB  = Connection traversing PCIe as well as a PCIe Host Bridge (typically the CPU)
  PXB  = Connection traversing multiple PCIe bridges (without traversing the PCIe Host Bridge)
  PIX  = Connection traversing at most a single PCIe bridge
  NV#  = Connection traversing a bonded set of # NVLinks
```



Note:

CPU: 2 x AMD Zen 3 EPYC™ 7763, 2.45 GHz and GPU: 8x NVIDIA A100 SXM4 GPUs

Karolina GPU Accelerated nodes
NVLink GPU to GPU interconnect



Bandwidth for accessing remote memory over NVLink 3.0 for all combinations of GPUs

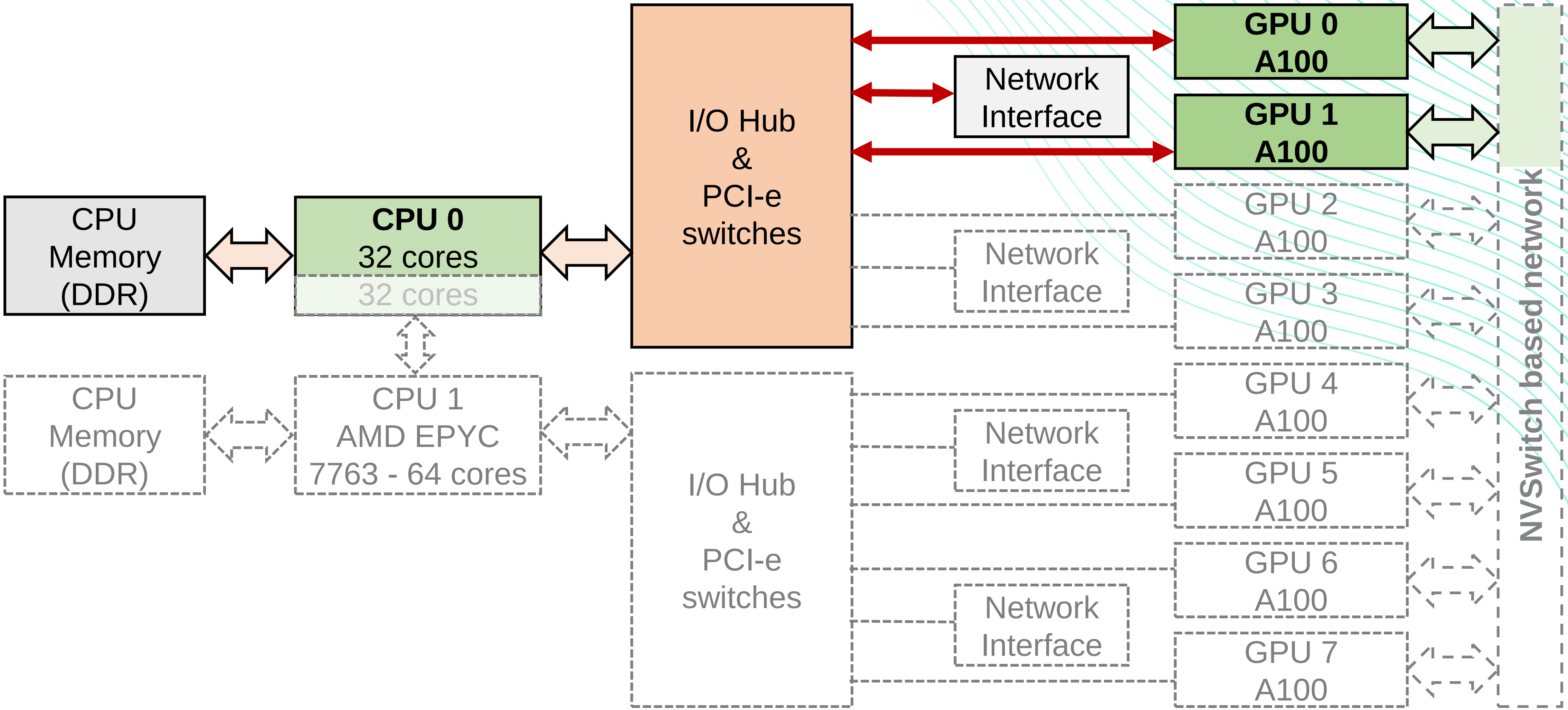
Unidirectional Bandwidth [GB/s]

GPU	0	1	2	3	4	5	6	7
0	1180	244	255	251	255	255	249	255
1	251	1202	256	245	256	256	252	257
2	248	256	1195	255	252	255	255	248
3	252	257	257	1198	253	255	255	249
4	244	255	256	249	1173	254	249	253
5	251	256	255	251	256	1198	255	252
6	256	251	255	255	253	254	1195	248
7	257	256	248	255	257	251	255	1206

1) 8x NVIDIA A100 (320GB)
2) 6x NVIDIA NVSwitches
3) 4x Mellanox ConnectX-6 (200 Gb/s)
4) Dual 64-Core AMD CPUs and 1 TB System Memory

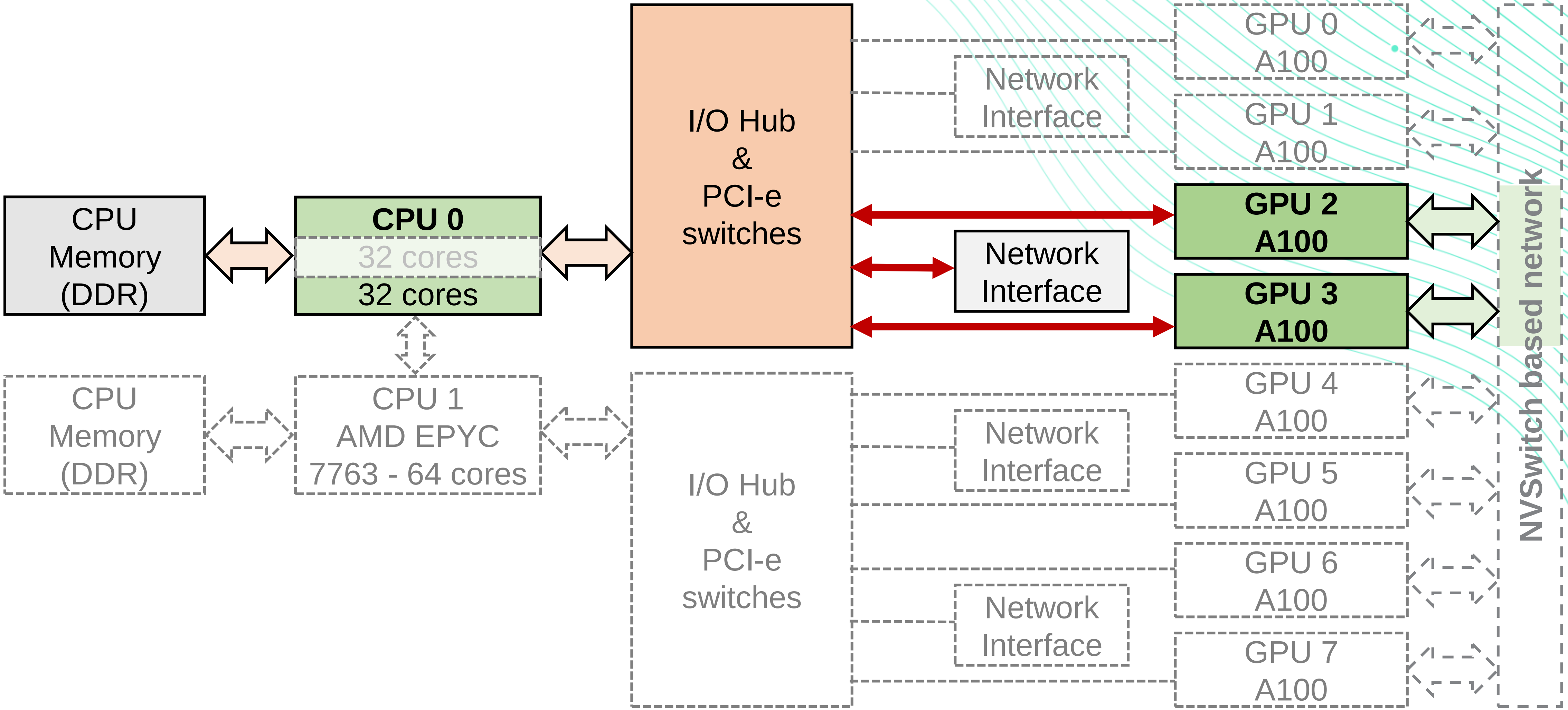
Note: CPU: 2 x AMD Zen 3 EPYC™ 7763, 2.45 GHz and GPU: 8x NVIDIA A100 SXM4 GPUs

Karolina GPU Accelerated nodes - Partial node allocation



- Karolina GPU nodes contain 8 GPUs: **1:4 ratio between processors and accelerators**.
- 40 GB per GPU --> 320 GB of GPU memory in total which can be shared among GPUs

Karolina GPU Accelerated nodes - Partial node allocation



- Karolina GPU nodes contain 8 GPUs: **1:4 ratio between processors and accelerators**.
- 40 GB per GPU --> 320 GB of GPU memory in total which can be shared among GPUs

EPICURE
Unlocking European-level HPC Support

Hands on:
Connecting to Karolina cluster
and
installation of VS Code

Hands on
Accessing GPU accelerated nodes

- IT4Innovations Documentation: https://docs.it4i.cz/

- **What OS do you use?** (Linux, Windows, MacOS)
- **Accessing the Clusters**
  - https://docs.it4i.cz/general/shell-and-data-access/
  - Generate SSH key pairs (id_rsa, id_rsa.pub):
    - **ssh-keygen** (preferred): https://docs.it4i.cz/general/accessing-the-clusters/shell-access-and-data-transfer/ssh-keys/
- **IT4I Account**
  - Training Login Credentials
  - https://extranet.it4i.cz/ssp/?action=changesshkey
  - https://youtu.be/zM1EPE3qw-8

Hands on
Accessing GPU accelerated nodes

- SSH configuration
  - Windows: *c:\Users\jarXXX\.ssh*
  - Linux: */home/jarXXX/.ssh/*
  - MacOS: /Users/*jarXXX*/.ssh/

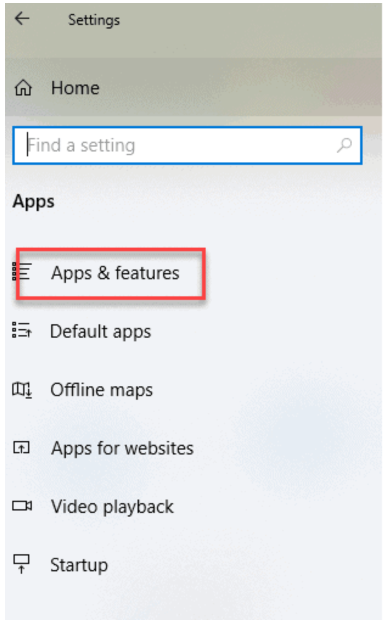
- .ssh/*
  - authorized_keys
  - config
  - id_rsa
  - id_rsa.pub
  - known_hosts

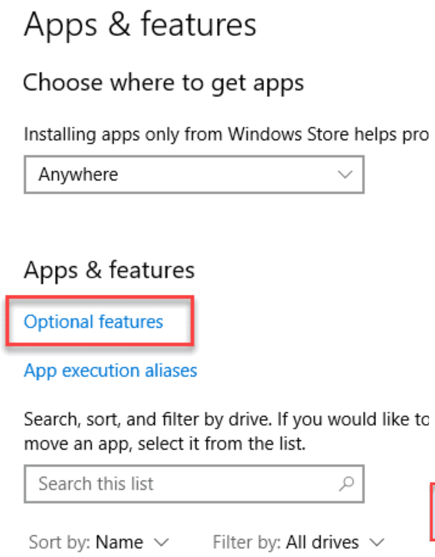
```
host karolina
        HostName karolina.it4i.cz
        IdentityFile ~/.ssh/id_rsa
        User dd-XX-XX-XX
```

Hands on
Accessing GPU accelerated nodes

Linux: *sudo apt install openssh-client*
cd ~/.ssh
ssh-keygen

Windows: *cd %USERPROFILE%/.ssh*
ssh-keygen





Apps & features

Choose where to get apps

Installing apps only from Windows Store helps pro

Anywhere

Apps & features

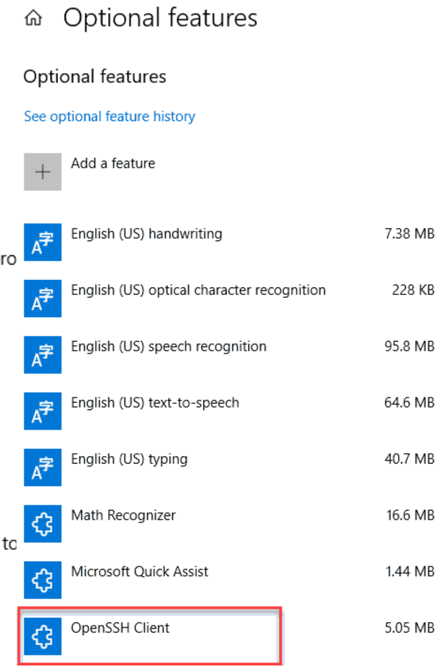
Optional features

App execution aliases

Search, sort, and filter by drive. If you would like to move an app, select it from the list.

Search this list

Sort by: Name   Filter by: All drives



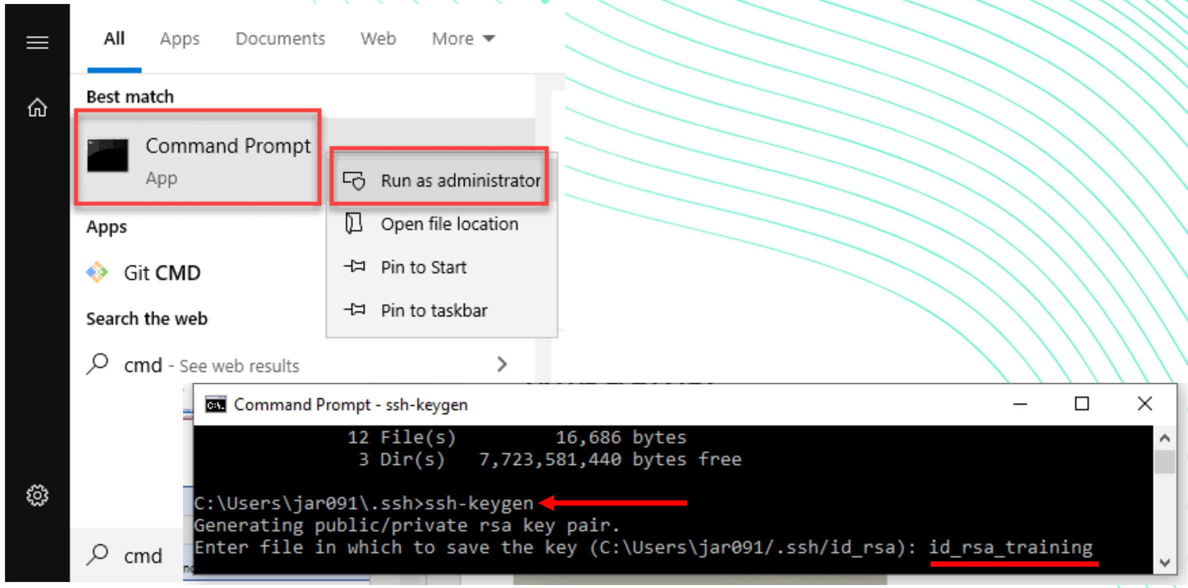
Optional features

Optional features

See optional feature history

Add a feature

English (US) handwriting	7.38 MB
English (US) optical character recognition	228 KB
English (US) speech recognition	95.8 MB
English (US) text-to-speech	64.6 MB
English (US) typing	40.7 MB
Math Recognizer	16.6 MB
Microsoft Quick Assist	1.44 MB
OpenSSH Client	5.05 MB

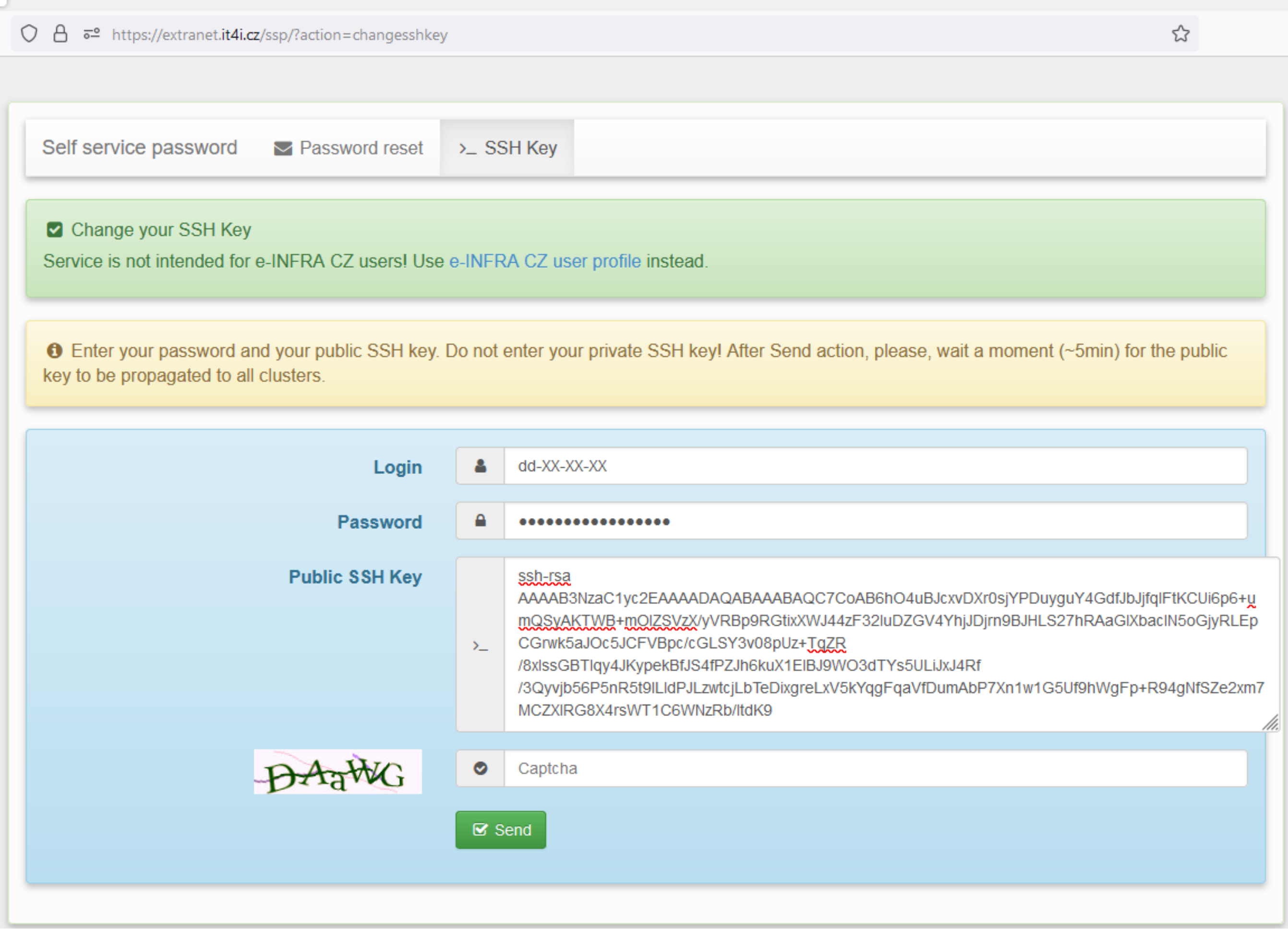


```
              12 File(s)         16,686 bytes
               3 Dir(s)   7,723,581,440 bytes free

C:\Users\jar091\.ssh>ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (C:\Users\jar091/.ssh/id_rsa): id_rsa_training
```

Hands on
Accessing GPU accelerated nodes

- **Print SSH public key**
  - Windows: *cd c:\Users\jarXXX\.ssh*
    notepad id_rsa_training.pub
  - Linux/MacOS: *cd /home/jarXXX/.ssh/*
    cat id_rsa_training.pub

- **IT4I Account**
  - Training Login Credentials
  - https://extranet.it4i.cz/ssp/?action=changesshkey
  - https://youtu.be/zM1EPE3qw-8



☑ Change your SSH Key

Service is not intended for e-INFRA CZ users! Use e-INFRA CZ user profile instead.

Enter your password and your public SSH key. Do not enter your private SSH key! After Send action, please, wait a moment (~5min) for the public key to be propagated to all clusters.

Login: dd-XX-XX-XX

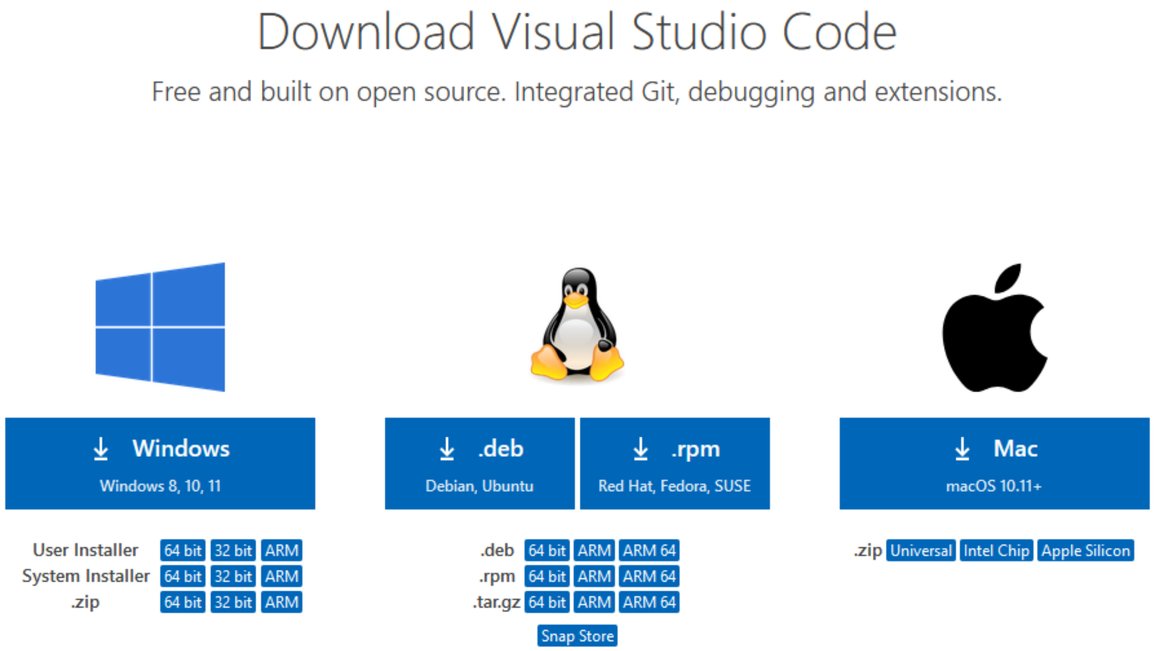
Password: •••••••••••••••••

Public SSH Key:

```
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQC7CoAB6hO4uBJcxvDXr0sjYPDuyguY4GdfJbJjfqlFtKCUi6p6+u
mQSyAKTWB+mOlZSVzX/yVRBp9RGtixXWJ44zF32luDZGV4YhjJDjrn9BJHLS27hRAaGlXbacIN5oGjyRLEp
CGrwk5aJOc5JCFVBpc/cGLSY3v08pUz+TqZR
/8xlssGBTlqy4JKypekBfJS4fPZJh6kuX1ElBJ9WO3dTYs5ULiJxJ4Rf
/3Qyvjb56P5nR5t9lLldPJLzwtcjLbTeDixgreLxV5kYqgFqaVfDumAbP7Xn1w1G5Uf9hWgFp+R94gNfSZe2xm7
MCZXlRG8X4rsWT1C6WNzRb/ltdK9
```

Hands on
Visual Studio Code

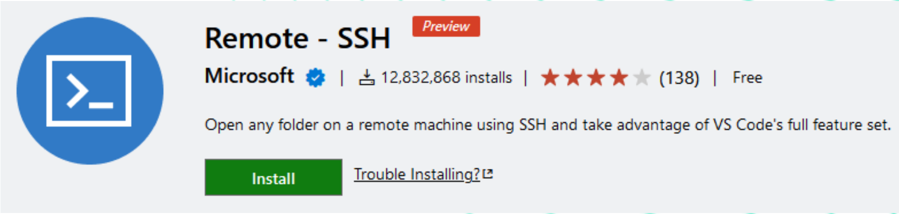
- VSCode
  - https://code.visualstudio.com/download



Download Visual Studio Code

Free and built on open source. Integrated Git, debugging and extensions.

- Extensions
  - https://marketplace.visualstudio.com/items?itemName=ms-vscode-remote.remote-ssh

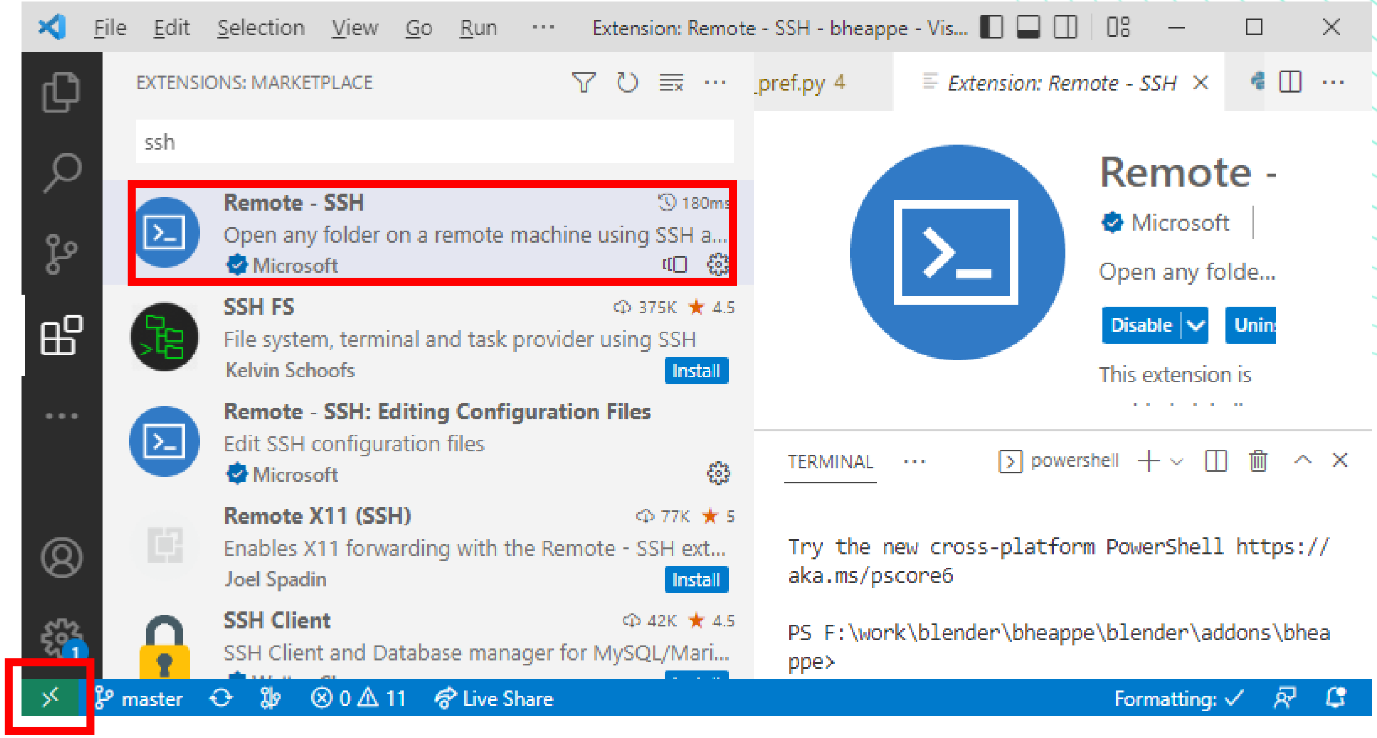


Remote - SSH

Microsoft | 12,832,868 installs | ★★★★☆ (138) | Free

Open any folder on a remote machine using SSH and take advantage of VS Code's full feature set.

Hands on
Visual Studio Code



Remote - SSH 180ms
Open any folder on a remote machine using SSH a...
Microsoft

SSH FS 375K ★ 4.5
File system, terminal and task provider using SSH
Kelvin Schoofs — Install

Remote - SSH: Editing Configuration Files
Edit SSH configuration files
Microsoft

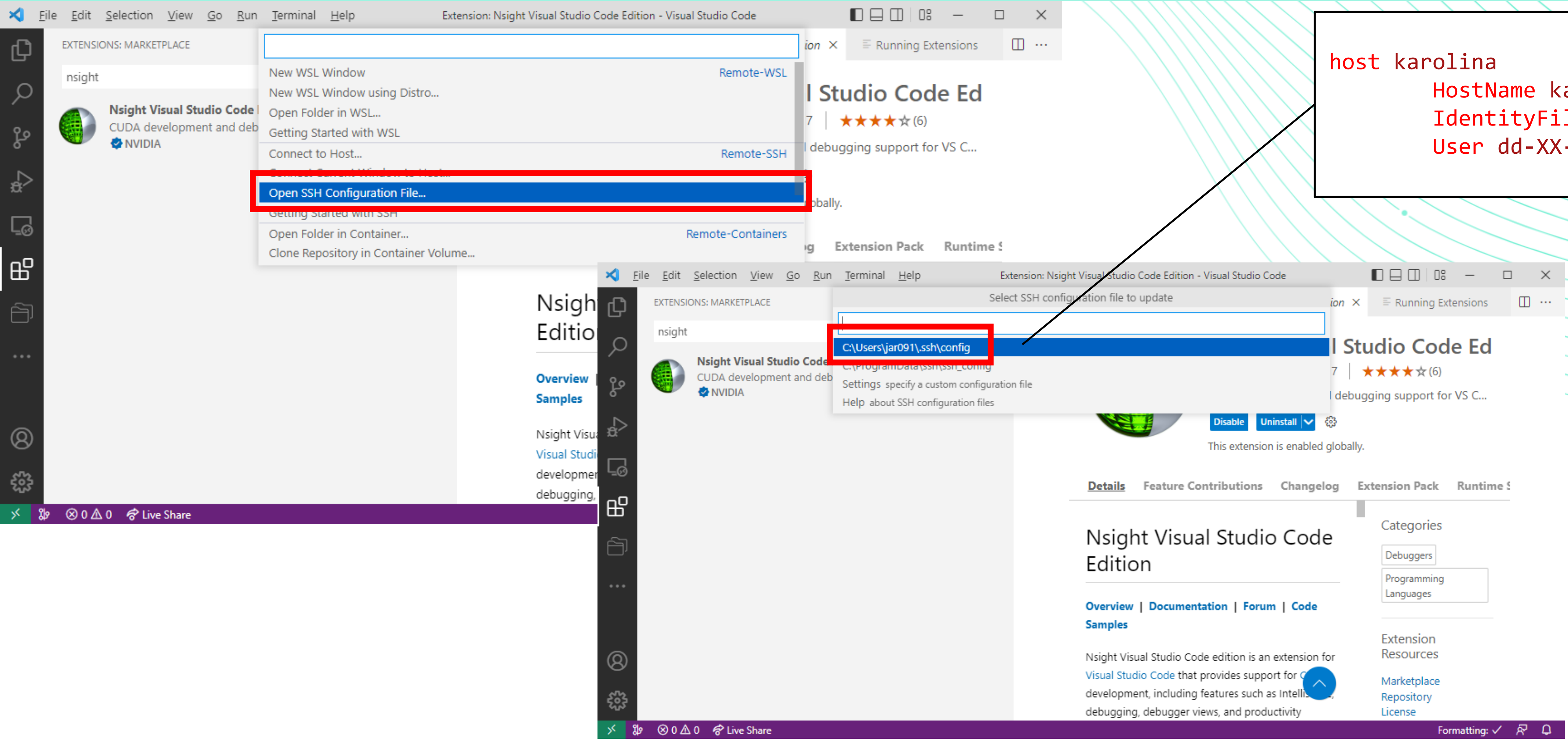
Remote X11 (SSH) 77K ★ 5
Enables X11 forwarding with the Remote - SSH ext...
Joel Spadin — Install

SSH Client 42K ★ 4.5
SSH Client and Database manager for MySQL/Mari...

Remote -
Microsoft
Open any folde...
Disable | Unin
This extension is

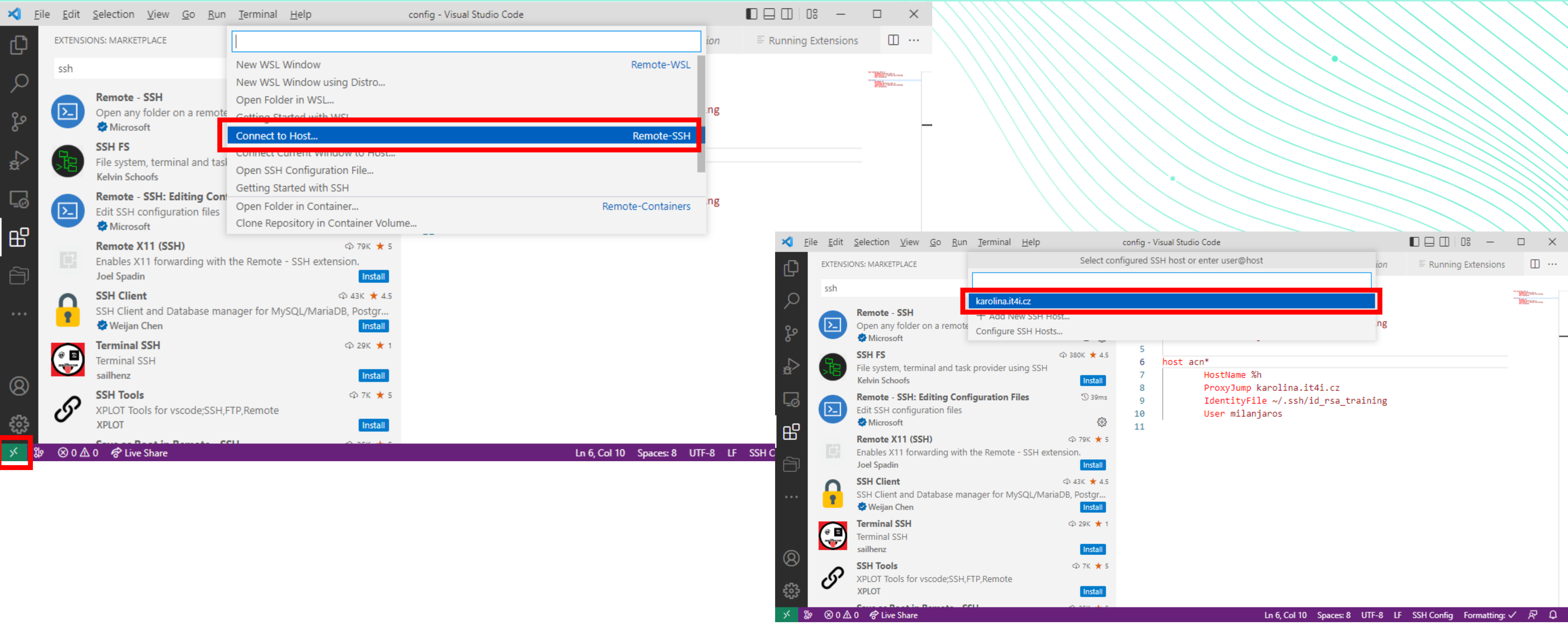
```
Try the new cross-platform PowerShell https://
aka.ms/pscore6

PS F:\work\blender\bheappe\blender\addons\bhea
ppe>
```

Hands on
Visual Studio Code


File Edit Selection View Go Run Terminal Help
Extension: Nsight Visual Studio Code Edition - Visual Studio Code
EXTENSIONS: MARKETPLACE
nsight
Nsight Visual Studio Code
CUDA development and deb
NVIDIA
New WSL Window
Remote-WSL
New WSL Window using Distro...
Open Folder in WSL...
Getting Started with WSL
Connect to Host...
Remote-SSH
Open SSH Configuration File...
Getting Started with SSH
Open Folder in Container...
Remote-Containers
Clone Repository in Container Volume...
Running Extensions
Extension Pack
Live Share
Select SSH configuration file to update
C:\Users\jar091\.ssh\config
C:\ProgramData\ssh\ssh_config
Settings specify a custom configuration file
Help about SSH configuration files
Disable
Uninstall
This extension is enabled globally.
Details
Feature Contributions
Changelog
Runtime S
Nsight Visual Studio Code Edition
Overview | Documentation | Forum | Code
Samples
Nsight Visual Studio Code edition is an extension for Visual Studio Code that provides support for C development, including features such as Intelli debugging, debugger views, and productivity
Categories
Debuggers
Programming Languages
Extension Resources
Marketplace
Repository
License
Formatting: ✓

```
host karolina
        HostName karolina.it4i.cz
        IdentityFile ~/.ssh/id_rsa
        User dd-XX-XX-XX
```

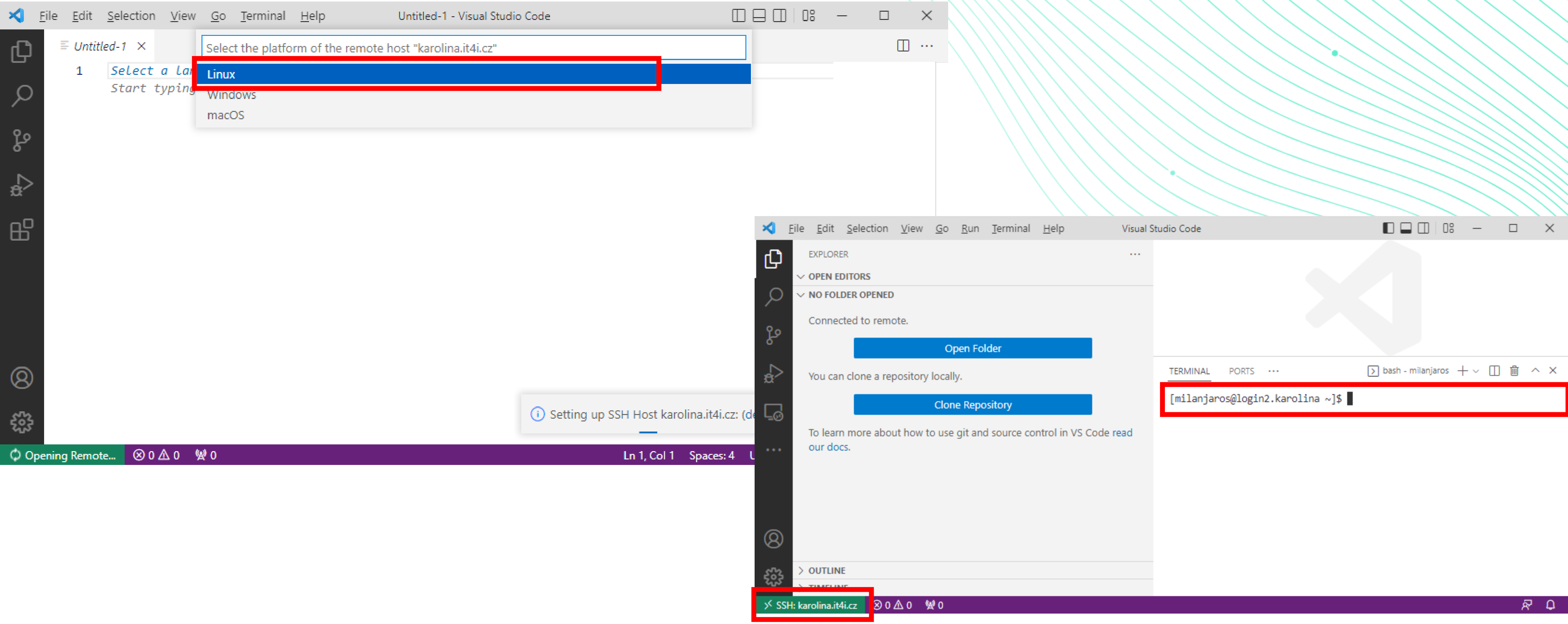
Hands on
Visual Studio Code



config - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXTENSIONS: MARKETPLACE
ssh
Remote - SSH
Open any folder on a remote
Microsoft
SSH FS
File system, terminal and task provider using SSH
Kelvin Schoofs
Remote - SSH: Editing Configuration Files
Edit SSH configuration files
Microsoft
Remote X11 (SSH)
Enables X11 forwarding with the Remote - SSH extension.
Joel Spadin
SSH Client
SSH Client and Database manager for MySQL/MariaDB, Postgr...
Weijan Chen
Terminal SSH
Terminal SSH
sailhenz
SSH Tools
XPLOT Tools for vscode;SSH,FTP,Remote
XPLOT
Install
Running Extensions
New WSL Window Remote-WSL
New WSL Window using Distro...
Open Folder in WSL...
Getting Started with WSL
Connect to Host... Remote-SSH
Connect Current Window to Host...
Open SSH Configuration File...
Getting Started with SSH
Open Folder in Container... Remote-Containers
Clone Repository in Container Volume...
⊗ 0 ⚠ 0 Live Share
Ln 6, Col 10 Spaces: 8 UTF-8 LF SSH Config Formatting: ✓
Select configured SSH host or enter user@host
karolina.it4i.cz
+ Add New SSH Host...
Configure SSH Hosts...

```
host acn*
        HostName %h
        ProxyJump karolina.it4i.cz
        IdentityFile ~/.ssh/id_rsa_training
        User milanjaros
```

Hands on
Visual Studio Code



Select the platform of the remote host "karolina.it4i.cz"

```
1  Select a lan
   Start typing
```

Setting up SSH Host karolina.it4i.cz: (de

Connected to remote.

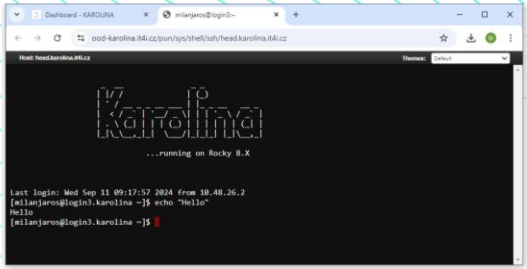
You can clone a repository locally.

To learn more about how to use git and source control in VS Code read our docs.

```
[milanjaros@login2.karolina ~]$
```

Hands on
Visual Studio Code (via Open OnDemand)

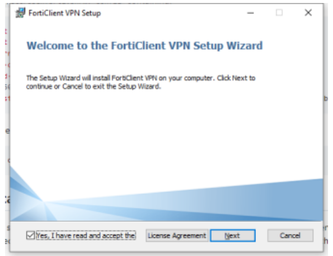
- OOD: https://docs.it4i.cz/general/accessing-the-clusters/graphical-user-interface/ood/
- VPN – IT4I
  - https://docs.it4i.cz/general/accessing-the-clusters/vpn-access/
  - Windows: Download the FortiClient app from the official page or the Windows Store.
  - Mac: Download the FortiClient VPN app from the Apple Store.
  - Linux: Download the FortiClient or OpenFortiVPN app.



ood-karolina.it4i.cz/pun/sys/shell/ssh/head.karolina.it4i.cz

```
                 Karolina

            ...running on Rocky 8.X

Last login: Wed Sep 11 09:17:57 2024 from 10.48.26.2
[milanjaros@login3.karolina ~]$ echo "Hello"
Hello
[milanjaros@login3.karolina ~]$
```

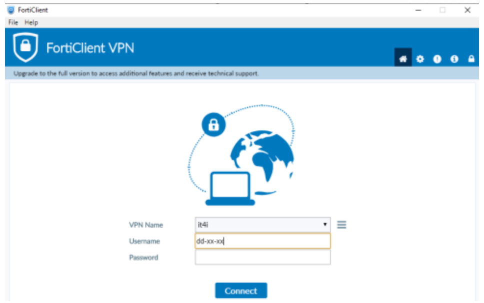
Windows:



Welcome to the FortiClient VPN Setup Wizard

The Setup Wizard will install FortiClient VPN on your computer. Click Next to continue or Cancel to exit the Setup Wizard.

☑ Yes, I have read and accept the



Upgrade to the full version to access additional features and receive technical support.

VPN Name: it4i
Username: dd-xx-xx
Password:

Linux:

Fedora:

```
root@fedora:~# dnf install openfortivpn
```

Ubuntu:

```
root@ubuntu:~# apt install openfortivpn
```

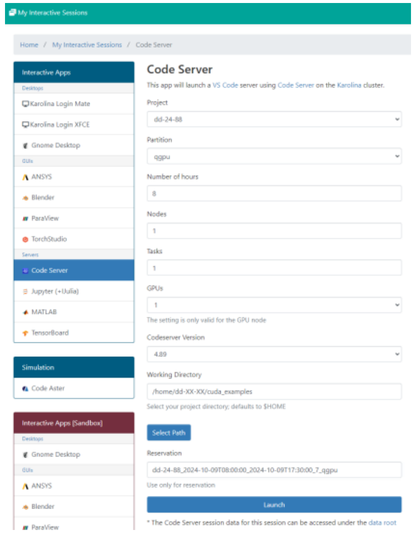
Debian:

```
root@debian:~# apt install openfortivpn
```

```
$cat ~/it4i-vpn-config
    host = reconnect.it4i.cz
    port = 443
    username = USER
    set-dns = 1
    pppd-use-peerdns = 0

$sudo openfortivpn -c ~/it4i-vpn-config
```

VS Code (via OOD)

https://ood-karolina.it4i.cz/



Code Server

This app will launch a VS Code server using Code Server on the Karolina cluster.

Project: dd-24-88

Partition: qgpu

Number of hours: 8

Nodes: 1

Tasks: 1

GPUs: 1

The setting is only valid for the GPU node

Codeserver Version: 4.89

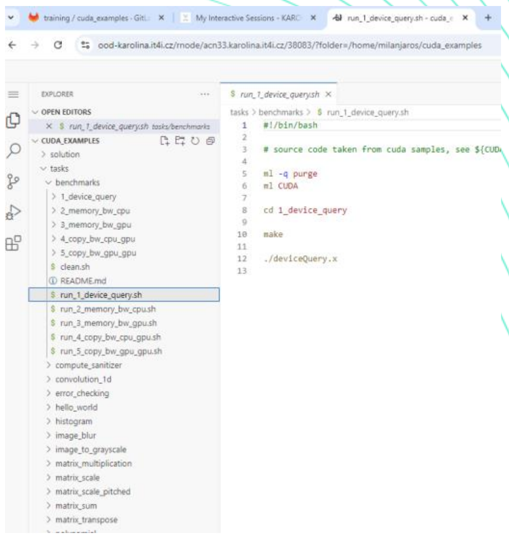
Working Directory: /home/dd-XX-XX/cuda_examples

Select your project directory; defaults to $HOME

Reservation: dd-24-88_2024-10-09T08:00:00_2024-10-09T17:30:00_7_qgpu

Use only for reservation

* The Code Server session data for this session can be accessed under the data root



ood-karolina.it4i.cz/rnode/acn33.karolina.it4i.cz/38083/?folder=/home/milanjaros/cuda_examples

```
#!/bin/bash

# source code taken from cuda samples, see ${CUD

ml -q purge
ml CUDA

cd 1_device_query

make

./deviceQuery.x

```

Hands-on exercises git repository

- https://code.it4i.cz/training/cuda_examples
- Clone the repository on Karolina
  - `git clone https://code.it4i.cz/training/cuda_examples.git`
- Open the repository folder in VS Code (Open Folder)

- Tasks – assignments and starting codes for the exercises
- Solution – finished solutions

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-24-88 -p qgpu --gpus 1  --nodes 1 --time 2:00:00`
  - Request 1 GPU on 1 node for 2 hours
- `salloc -A DD-24-88 -p qgpu`
  - Default: 1 GPU, 1 node, 24h time limit
- `salloc -A DD-24-88 -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-24-88 -p qgpu --gpus 4  --nodes 1 --time 2:00:00`
  - Request 4 GPUs on 1 node for 2 hours
- `salloc -A DD-24-88 -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

- No need to hope for a free node
- We have a reservation prepared
- Wednesday, 2024-10-09
- `salloc --account=DD-24-88 --reservation=dd-24-88_2024-10-09T08:00:00_2024-10-09T17:30:00_7_qgpu --nodes 1 --gpus 2 --cpus-per-gpu 16`
- Thursday, 2024-10-10
- `salloc --account=DD-24-88 --reservation=dd-24-88_2024-10-10T09:00:00_2024-10-10T14:30:00_7_qgpu --nodes 1 --gpus 2 --cpus-per-gpu 16`

- Load the CUDA module to setup the environment
  - `module load CUDA`

EPICURE

Unlocking European-level HPC Support

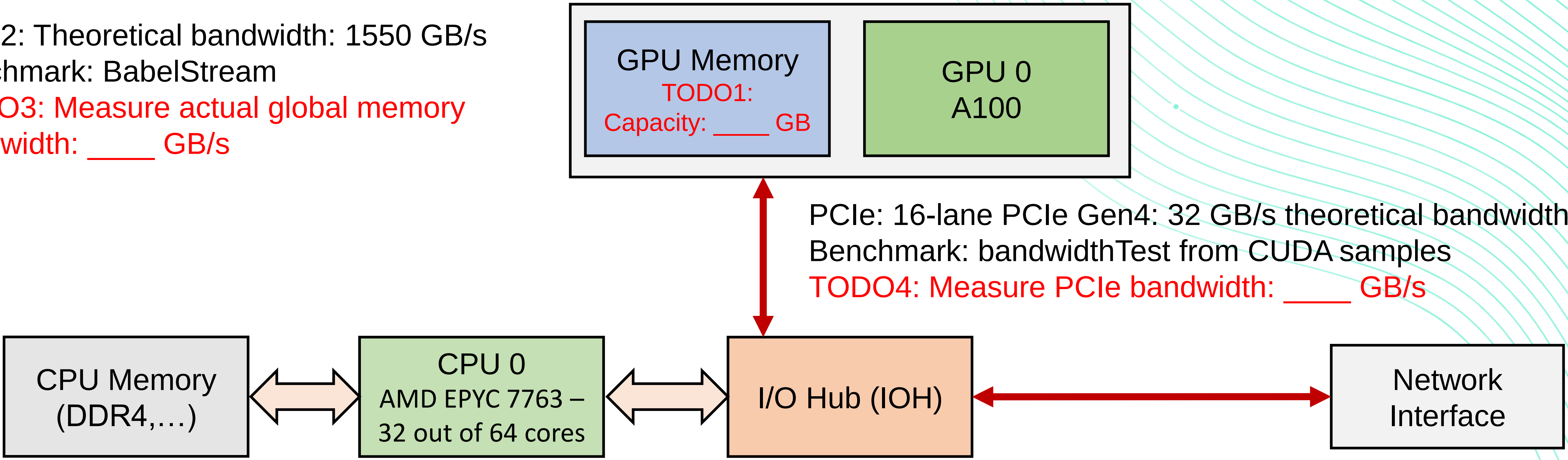
Hands on: Benchmark Hardware Properties

Hands on
Benchmark Hardware Properties

- `cd /home/dd-XX-XX-XX/cuda_examples/tasks/benchmarks`
- Run the following benchmarks and complete the TODO values on the following 2 slides
- Retrieve information about the available GPUs, find global memory capacity
    - `./run_1_device_query.sh`
- Measure CPU memory (RAM) bandwidth
    - `./run_2_memory_bw_cpu.sh`
- Measure GPU memory bandwidth, compare it with CPU memory bandwidth
    - `./run_3_memory_bw_gpu.sh`
- Measure CPU-GPU data transfer bandwidth
    - `./run_4_copy_bw_cpu_gpu.sh`
- Measure GPU-GPU data transfer bandwidth, compare with CPU-GPU data transfer bandwidth
    - `./run_5_copy_bw_gpu_gpu.sh`

Hands on
Benchmark Hardware Properties

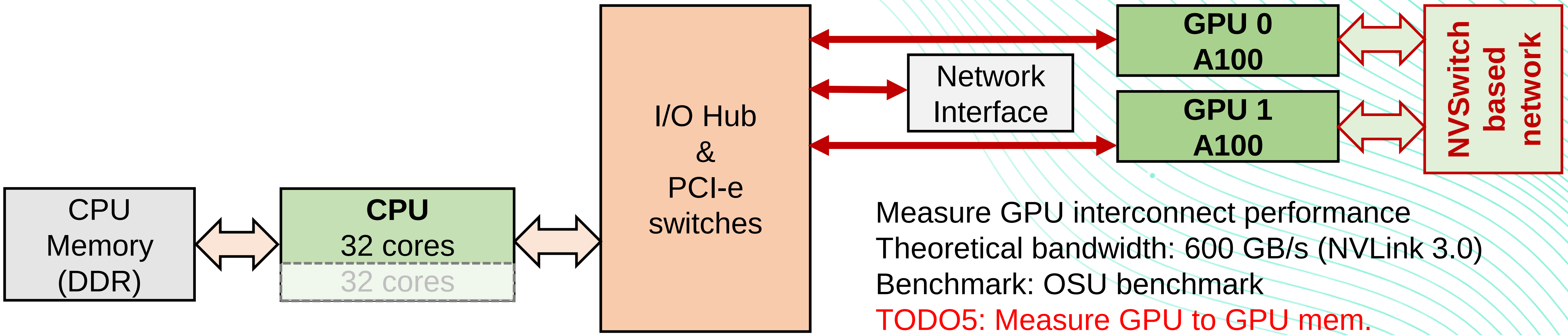
HBM2: Theoretical bandwidth: 1550 GB/s
Benchmark: BabelStream
TODO3: Measure actual global memory bandwidth: ____ GB/s



PCIe: 16-lane PCIe Gen4: 32 GB/s theoretical bandwidth
Benchmark: bandwidthTest from CUDA samples
TODO4: Measure PCIe bandwidth: ____ GB/s

Theoretical 205 GB/s per chip or ±100 GB/s per 32 cores
Benchmark: STREAM benchmark
TODO2: Measure CPU memory bandwidth: ____ GB/s

Hands on
Benchmark Hardware Properties



Measure GPU interconnect performance
Theoretical bandwidth: 600 GB/s (NVLink 3.0)
Benchmark: OSU benchmark
TODO5: Measure GPU to GPU mem.
bandwidth: ____ GB

Hands on – solution, output
Benchmark Hardware Properties

```
$ ./run_1_device_query.sh
 ...
Detected 2 CUDA Capable device(s)

Device 0: "NVIDIA A100-SXM4-40GB"
  CUDA Driver Version / Runtime Version          12.4 / 12.4
  CUDA Capability Major/Minor version number:    8.0
  Total amount of global memory:                 40326 MBytes (42285268992 bytes)
  (108) Multiprocessors, (064) CUDA Cores/MP:    6912 CUDA Cores
  GPU Max Clock rate:                            1290 MHz (1.29 GHz)
  Memory Clock rate:                             1215 Mhz
  Memory Bus Width:                              5120-bit
  L2 Cache Size:                                 41943040 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536),
3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        167936 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 3 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 76 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device
simultaneously) >

Device 1: "NVIDIA A100-SXM4-40GB"
...
> Peer access from NVIDIA A100-SXM4-40GB (GPU0) -> NVIDIA A100-SXM4-40GB (GPU1) : Yes
> Peer access from NVIDIA A100-SXM4-40GB (GPU1) -> NVIDIA A100-SXM4-40GB (GPU0) : Yes
```

```
$ ./run_2_memory_bw_cpu.sh
 ...
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:           81048.9     0.197680     0.197412     0.197956
Scale:          54754.0     0.292969     0.292216     0.293471
Add:            61358.3     0.391617     0.391145     0.392164
Triad:          61553.2     0.390135     0.389907     0.390916
-------------------------------------------------------------
Solution Validates: avg error less than 1.000000e-13 on all three arrays
-------------------------------------------------------------
```

```
$ ./run_3_memory_bw_gpu.sh
 ...
Function      MBytes/sec  Min (sec)   Max         Average
Copy          1398531.090 0.00077     0.00077     0.00077
Mul           1374268.159 0.00078     0.00079     0.00078
Add           1384314.665 0.00116     0.00120     0.00118
Triad         1388344.358 0.00116     0.00120     0.00117
Dot           1288563.379 0.00083     0.00088     0.00085
```

```
$ ./run_4_copy_bw_cpu_gpu.sh
 ...
Host to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(GB/s)
   32000000                     24.5

 Device to Host Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(GB/s)
   32000000                     25.9
```

```
$ ./run_5_copy_bw_gpu_gpu.sh
 ...
# OSU MPI-CUDA Bandwidth Test
# Send Buffer on DEVICE (D) and Receive Buffer on DEVICE
(D)
# Size          Bandwidth (MB/s)
1024                    221.03
2048                    440.67
4096                    863.83
8192                   1714.29
16384                  3539.78
32768                  7118.31
65536                 14161.52
131072                28030.01
262144                50905.00
524288                84376.06
1048576              131603.12
2097152              178780.51
4194304              218991.18
8388608              243920.48
16777216             258973.37
33554432             269537.56
67108864             274921.10
134217728            278010.86
268435456            279650.07
```

EPICURE

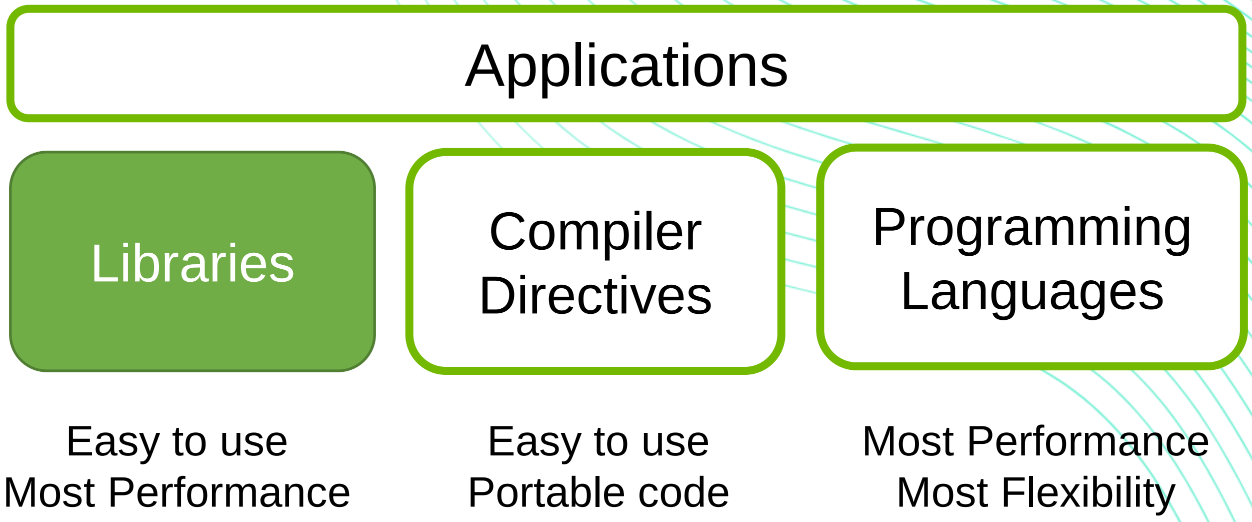
Unlocking European-level HPC Support

CUDA Programming

Ways to Accelerate Applications

Libraries

- ease of use:
  - enables GPU acceleration without any GPU programming
- drop-in:
  - follow standard APIs
  - minimal code changes
- quality:
  - high-quality implementations

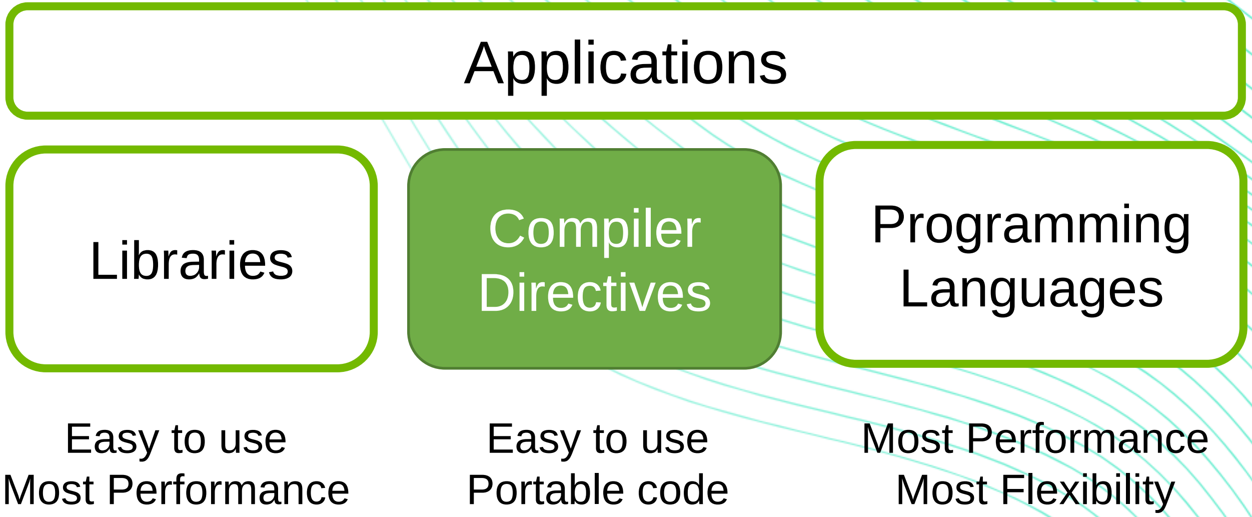


Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

Ways to Accelerate Applications

Compiler Directives

- ease of use
  - compiler takes care of details of parallelism management and data movement
- portable
  - code is generic, not specific to any type of hardware



- Example: OpenACC
  - Compiler directives for C, C++, and FORTRAN

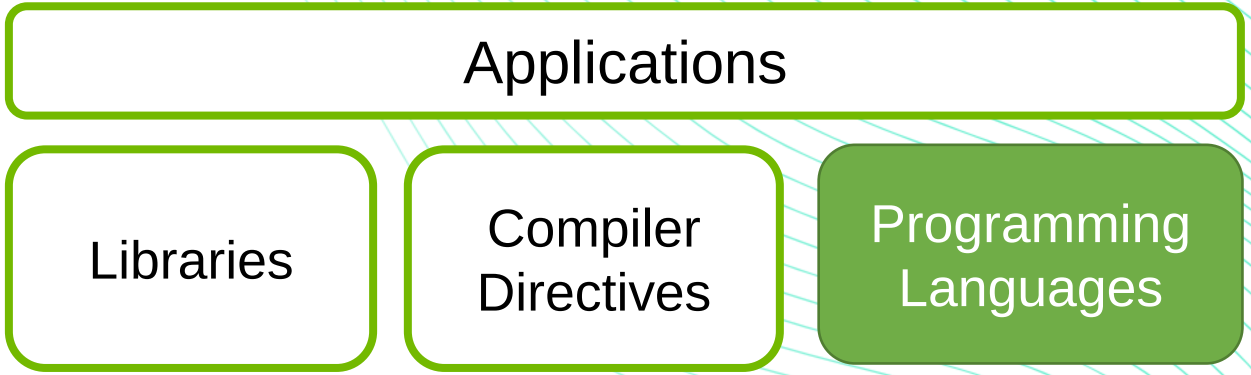
```
#pragma acc parallel loop
copyin(input1[0:inputLength],input2[0:inputLength]),
       copyout(output[0:inputLength])
  for(i = 0; i < inputLength; ++i) {
    output[i] = input1[i] + input2[i];
  }
```

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

Ways to Accelerate Applications

Programming Languages

- Performance: best control of parallelism and data movement
- Flexible: the computation does not need to fit into a limited set of library patterns or directives
- Complex: programmer often needs to express more details



Easy to use
Most Performance

Easy to use
Portable code

Most Performance
Most Flexibility

GPU Programming Languages

Numerical analytics	MATLAB, Mathematica	C	CUDA C, OpenACC
Python	PyCUDA, Numba	C++	CUDA C++, Thrust
Fortran	CUDA Fortran, OpenACC	C#	Hybridizer

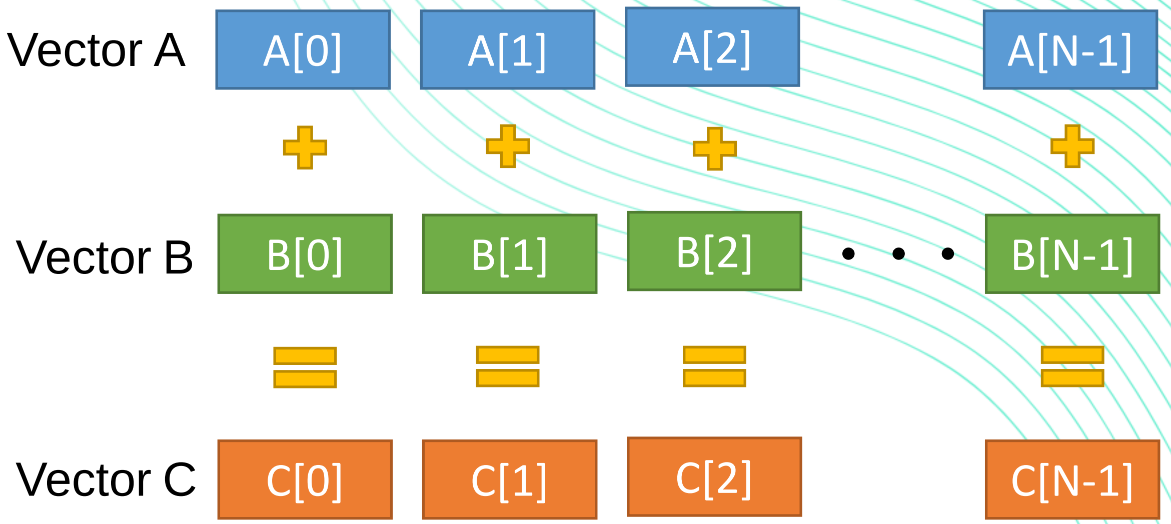
Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CUDA programming
Data Parallelism

Vector addition example

```
// Compute vector sum C = A + B
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
  int i;
  for (i = 0; i < n; i++)
       h_C[i] = h_A[i] + h_B[i];
}

int main()
{
 // Memory allocation for h_A, h_B, and h_C
 // read h_A and h_B from file for N elements
 …
 vecAdd(h_A, h_B, h_C, N);
}
```

$$A + B = C$$



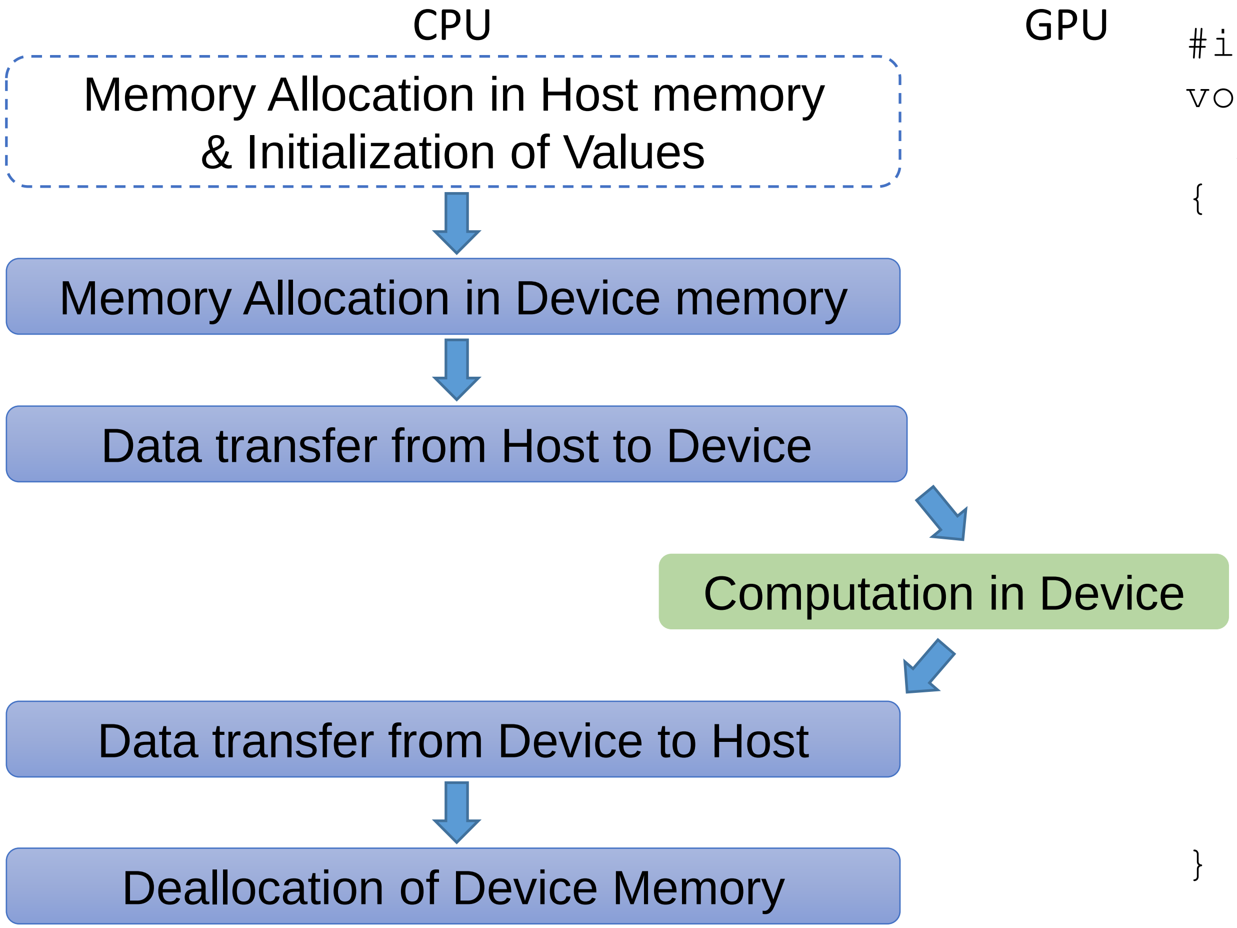
Vector A: A[0], A[1], A[2], A[N-1]

Vector B: B[0], B[1], B[2] • • • B[N-1]

Vector C: C[0], C[1], C[2], C[N-1]

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

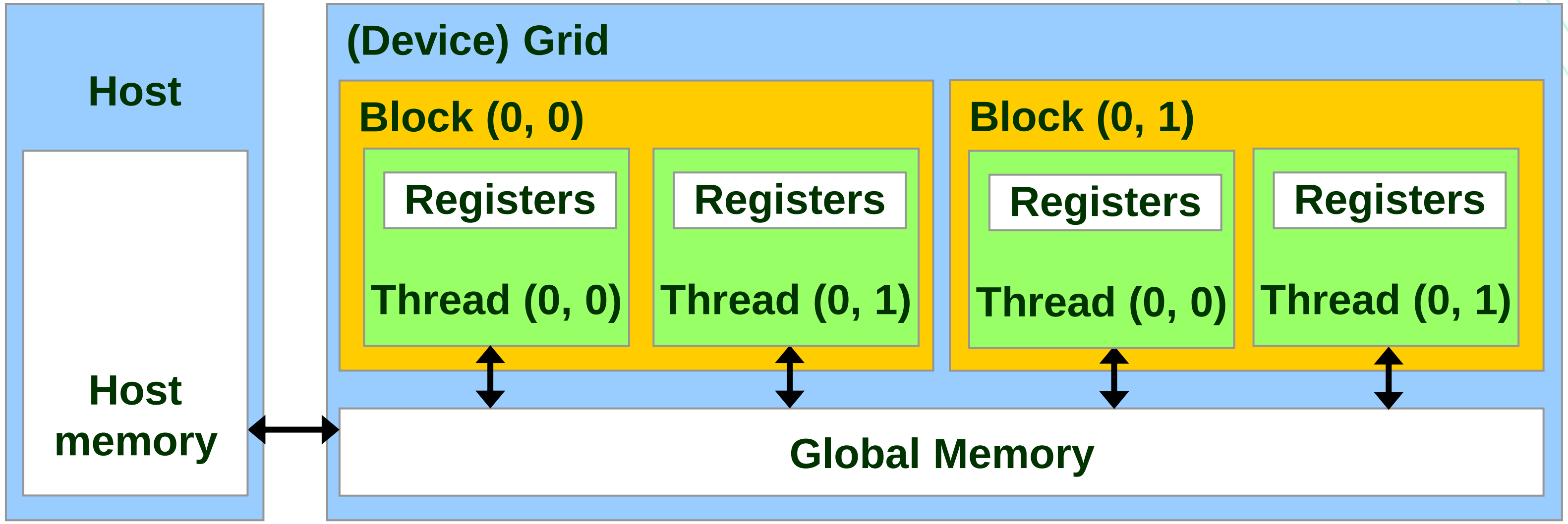
CUDA programming
Heterogenous Program


CPU
GPU
Memory Allocation in Host memory & Initialization of Values
Memory Allocation in Device memory
Data transfer from Host to Device
Computation in Device
Data transfer from Device to Host
Deallocation of Device Memory

```
#include <cuda.h>
void vecAdd(float *h_A, float *h_B, float *h_C,
  int n)
{
  int size = n* sizeof(float);
  float *d_A, *d_B, *d_C;

  // allocate device memory for A, B, and C
  // copy A and B to device memory

  // kernel launch code
  // – GPU performs the actual vector addition

  // copy C from the device memory

  // Free device vectors
}
```

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CUDA programming
Partial Overview of CUDA Memories



Device code (kernel) can:

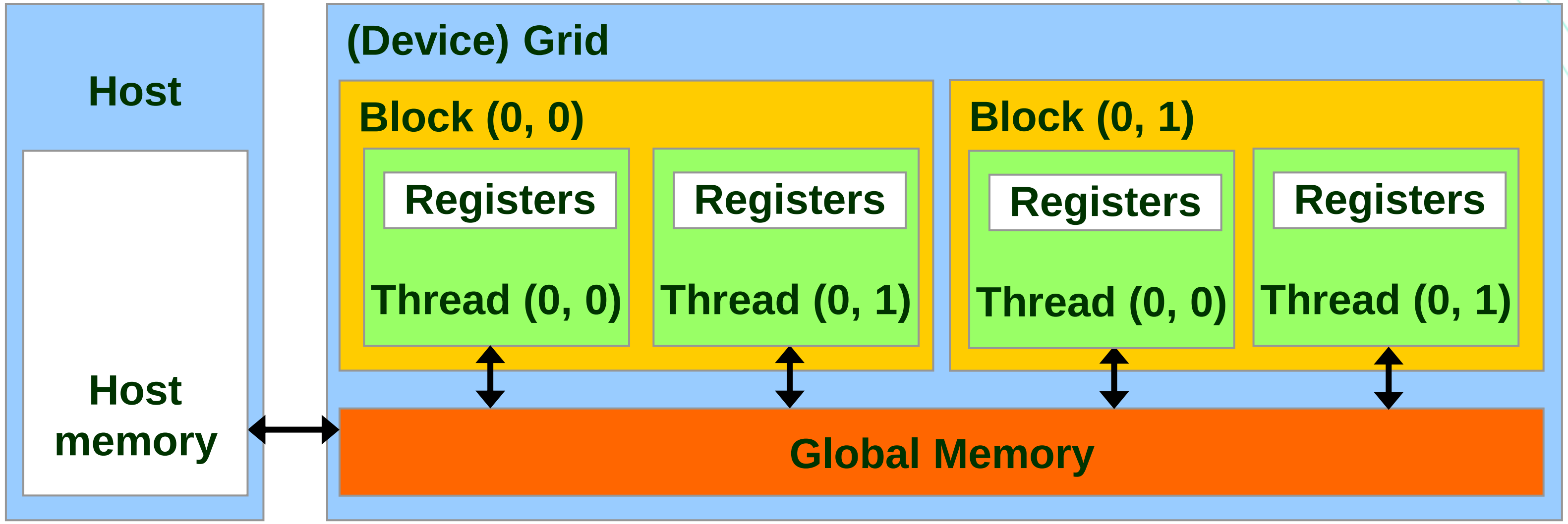
- R/W per-thread registers
- R/W all-shared global memory

Host code can

- Transfer data to/from per grid global memory

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CUDA programming
Partial Overview of CUDA Memories



cudaMalloc()

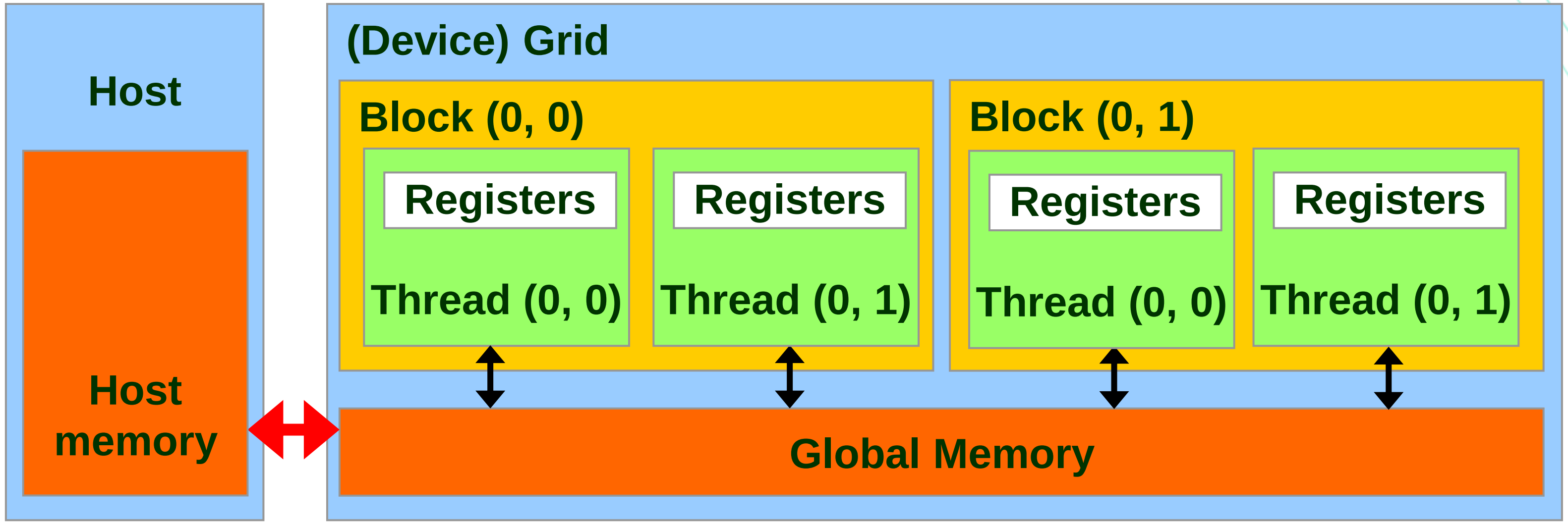
- Allocates an object in the device global memory
- Two parameters
  - Address of a pointer to the allocated object
  - Size of allocated object in terms of bytes

cudaFree()

- Frees object from device global memory
- One parameter
  - Pointer to freed object

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CUDA programming
Partial Overview of CUDA Memories



cudaMemcpy()

- memory data transfer
- Requires four parameters
  - Pointer to destination
  - Pointer to source
  - Number of bytes copied
  - Type/Direction of transfer
- Transfer to device is synchronous with respect to the host

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CUDA programming
Explicit Memory Management

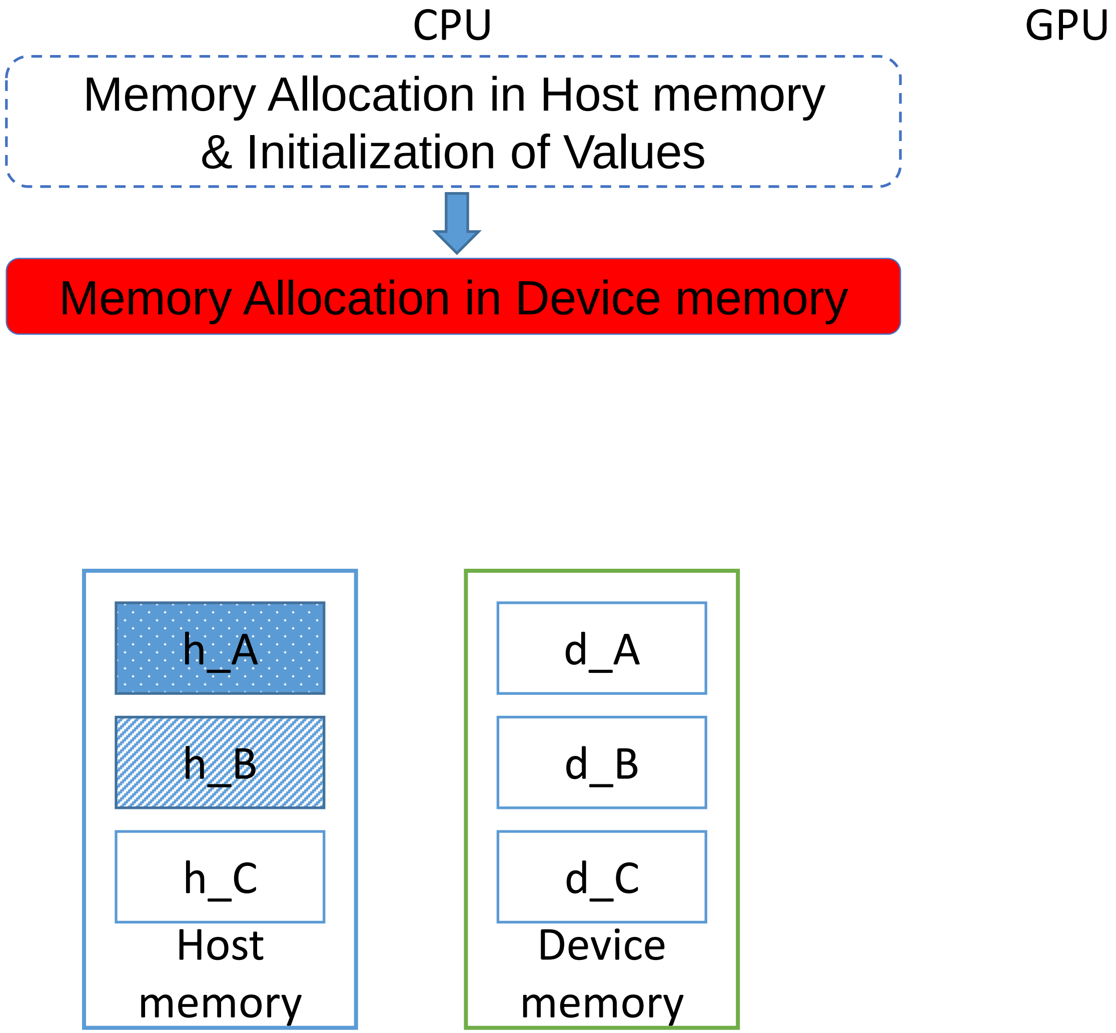
CPU

Memory Allocation in Host memory
& Initialization of Values

```
int main(){

 float *h_A, *h_B, *h_C;

 int n = 10000000 // size of an array
 int size = n * sizeof(float);

 h_A = (float*)malloc(size);
 h_B = (float*)malloc(size);
 h_C = (float*)malloc(size);

 // Initialize array
 for(int i = 0; i < array_size; i++){
 h_A[i] = 1.0f;
 h_B[i] = 2.0f;}

 vecAdd(h_A, h_B, h_C, n);

 // Deallocate host memory
 free(h_A); free(h_A); free(h_C);
}
```

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

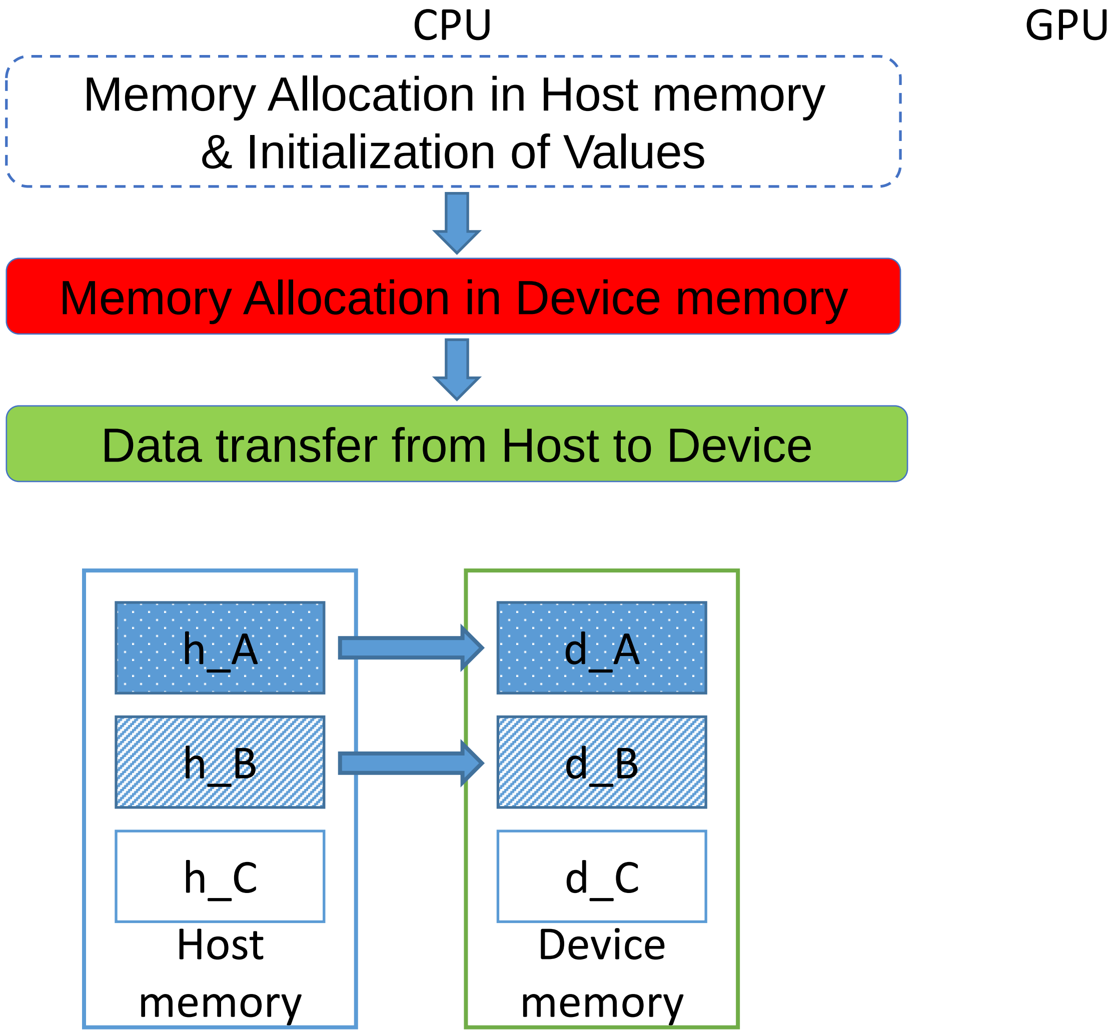
CUDA programming
Explicit Memory Management


CPU
GPU
Memory Allocation in Host memory
& Initialization of Values
Memory Allocation in Device memory
h_A
h_B
h_C
Host memory
d_A
d_B
d_C
Device memory

```
void vecAdd(float *h_A, float *h_B, float *h_C,
int n)
{
   int size = n * sizeof(float);
   float *d_A, *d_B, *d_C;
   cudaMalloc((void **) &d_A, size);
   cudaMalloc((void **) &d_B, size);
   cudaMalloc((void **) &d_C, size);
```

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

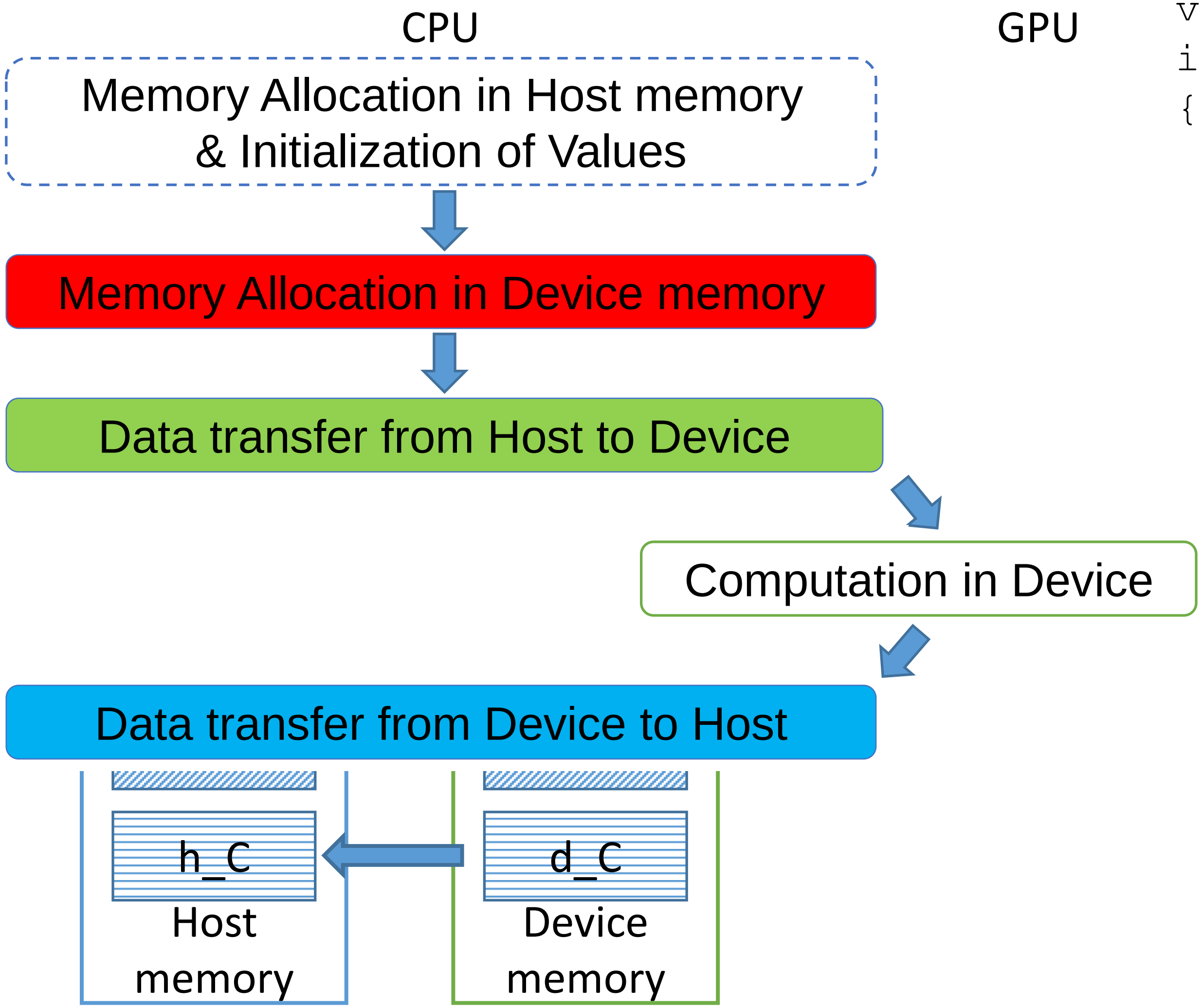
CUDA programming
Explicit Memory Management


CPU
GPU
Memory Allocation in Host memory & Initialization of Values
Memory Allocation in Device memory
Data transfer from Host to Device
h_A
h_B
h_C
Host memory
d_A
d_B
d_C
Device memory

```
void vecAdd(float *h_A, float *h_B, float *h_C,
int n)
{
   int size = n * sizeof(float);
   float *d_A, *d_B, *d_C;
   cudaMalloc((void **) &d_A, size);
   cudaMalloc((void **) &d_B, size);
   cudaMalloc((void **) &d_C, size);

   cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

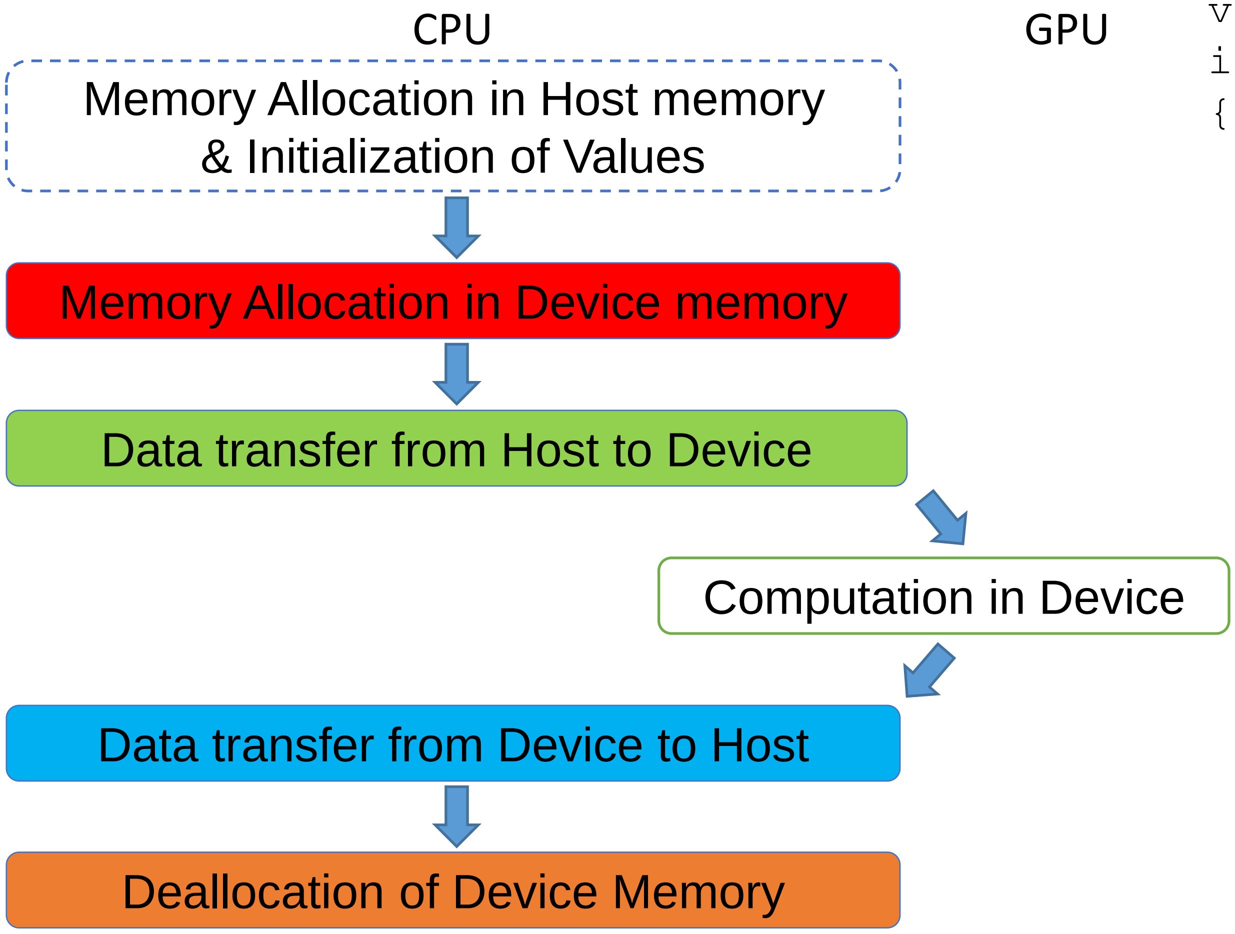
CUDA programming
Explicit Memory Management


CPU
GPU
Memory Allocation in Host memory & Initialization of Values
Memory Allocation in Device memory
Data transfer from Host to Device
Computation in Device
Data transfer from Device to Host
h_C
Host memory
d_C
Device memory

```
void vecAdd(float *h_A, float *h_B, float *h_C,
int n)
{
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;
    cudaMalloc((void **) &d_A, size);
    cudaMalloc((void **) &d_B, size);
    cudaMalloc((void **) &d_C, size);

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
     cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Kernel run

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
```

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CUDA programming
Explicit Memory Management

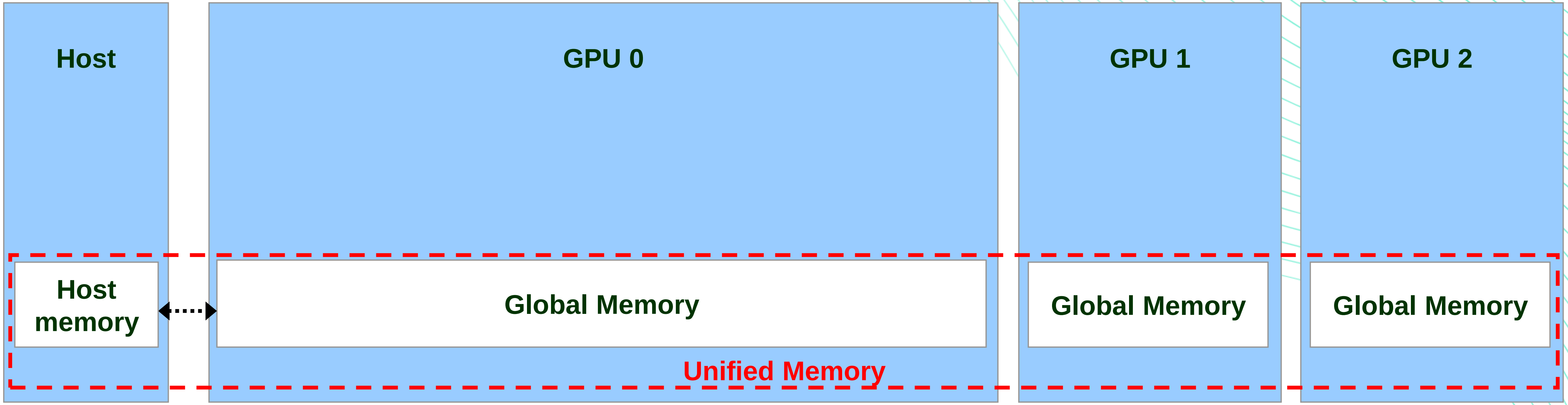


CPU
GPU
Memory Allocation in Host memory & Initialization of Values
Memory Allocation in Device memory
Data transfer from Host to Device
Computation in Device
Data transfer from Device to Host
Deallocation of Device Memory

```
void vecAdd(float *h_A, float *h_B, float *h_C,
int n)
{
  int size = n * sizeof(float);
  float *d_A, *d_B, *d_C;
  cudaMalloc((void **) &d_A, size);
  cudaMalloc((void **) &d_B, size);
  cudaMalloc((void **) &d_C, size);

  cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
   cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

  // Kernel run

  cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

  cudaFree(d_A);
  cudaFree(d_B);
  cudaFree(d_C);
}
```

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

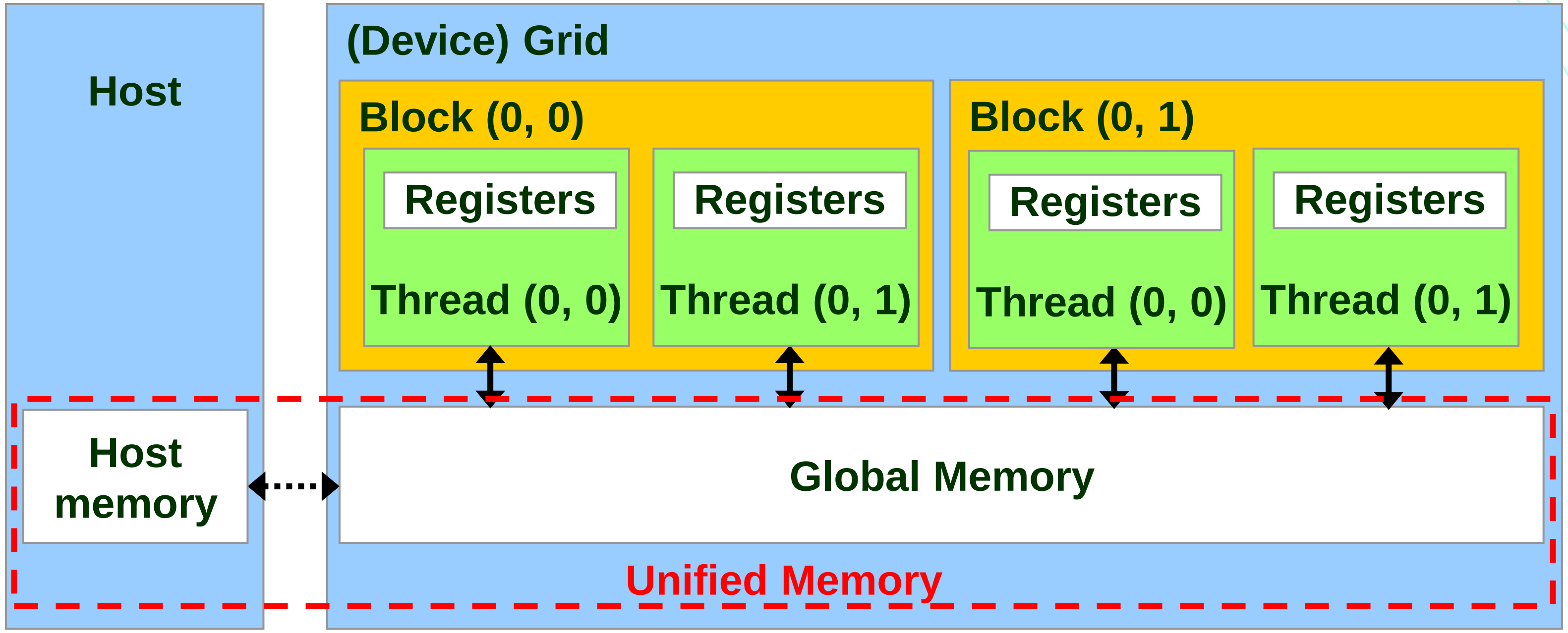
CUDA programming
Unified Memory



- Single memory address space accessible from all CPUs/GPUs in a single server
  - maintain single copy of data
- On-demand page migration - hardware/software handles automatically the data migration between the host and the device maintaining consistency between them

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CUDA programming
Unified Memory



Device code (kernel) can:

- R/W per-thread registers
- R/W all-shared global memory
- **R/W managed memory (Unified Memory)**

Host code can

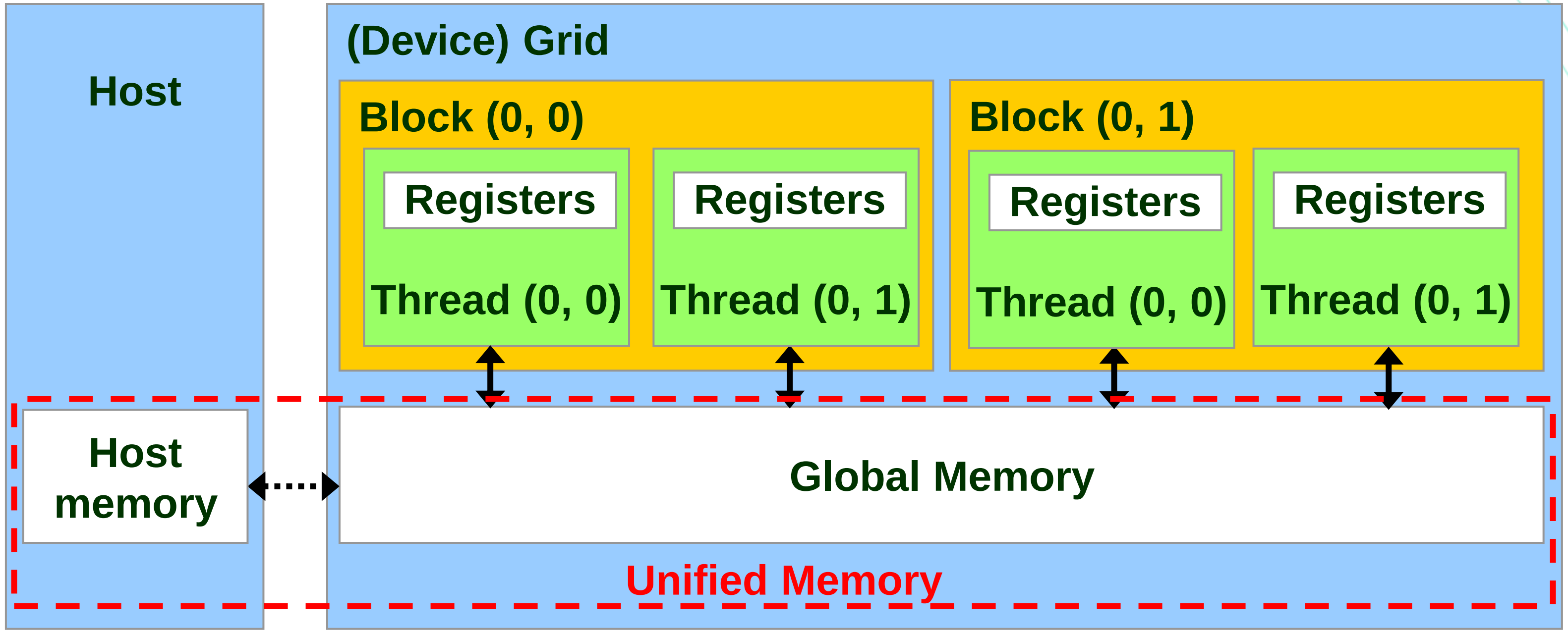
- Transfer data to/from per grid global memory
- **R/W managed memory Unified Memory)**

In modern GPUs:

- there are specialized hardware units managing page faulting
- data is migrated on demand, meaning that data gets copied only on page fault
- possibility to oversubscribe memory, enabling larger arrays than the device memory size

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CUDA programming
Unified Memory



cudaMallocManaged(void ptr, size_t size)**

- Allocates an object in the Unified Memory address space.
- Two parameters, with an optional third parameter.
  - Address of a pointer to the allocated object
  - Size of the allocated object in terms of bytes
  - [Optional] Flag indicating if memory can be accessed from any device or stream

cudaFree()

- Frees object from unified memory.
- One parameter
  - Pointer to freed object

Can be optimized

- cudaMemAdvise(),
- cudaMemPrefetchAsync(),
- cudaMemcpyAsync()

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CUDA programming
Unified Memory

CPU

Memory Allocation in Host memory
& Initialization of Values

```
int main(){

 float *h_A, *h_B, *h_C;

 int n = 10000000 // size of an array
 int size = n * sizeof(float);

 h_A = (float*)malloc(size);
 h_B = (float*)malloc(size);
 h_C = (float*)malloc(size);

 // Initialize array
 for(int i = 0; i < array_size; i++){
 h_A[i] = 1.0f;
 h_B[i] = 2.0f;}

 vecAdd(h_A, h_B, h_C, n);

 // Deallocate host memory
 free(h_A); free(h_B); free(h_C);
}
```

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CUDA programming
Unified Memory

```
int main(){

 float *h_A, *h_B, *h_C;

 int n = 10000000 // size of an array
 int size = n * sizeof(float);

 h_A = (float*)malloc(size);
 h_B = (float*)malloc(size);
 h_C = (float*)malloc(size);

 // Initialize array
 for(int i = 0; i < array_size; i++){
 h_A[i] = 1.0f;
 h_B[i] = 2.0f;}

 vecAdd(h_A, h_B, h_C, n);

 // Deallocate host memory
 free(h_A); free(h_A); free(h_C);
}
```

```
int main(){

 float *A, *B, *C;

 int n = 10000000 // size of an array
 int size = n * sizeof(float);

 cudaMallocManaged(&A, size);
 cudaMallocManaged(&B, size);
 cudaMallocManaged(&C, size);

 // Initialize array
 for(int i = 0; i < array_size; i++){
 A[i] = 1.0f;
 B[i] = 2.0f;}

 vecAdd(A, B, C, n);

 // Deallocate host memory
 cudaFree(A); cudaFree(B); cudaFree(C);
}
```

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CUDA programming
Unified Memory

```
void vecAdd(float *h_A, float *h_B, float *h_C,
int n)
{
  int size = n * sizeof(float);
  float *d_A, *d_B, *d_C;
  cudaMalloc((void **) &d_A, size);
  cudaMalloc((void **) &d_B, size);
  cudaMalloc((void **) &d_C, size);

  cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

  // Kernel run

  cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

  cudaFree(d_A);
  cudaFree(d_B);
  cudaFree(d_C);
}
```

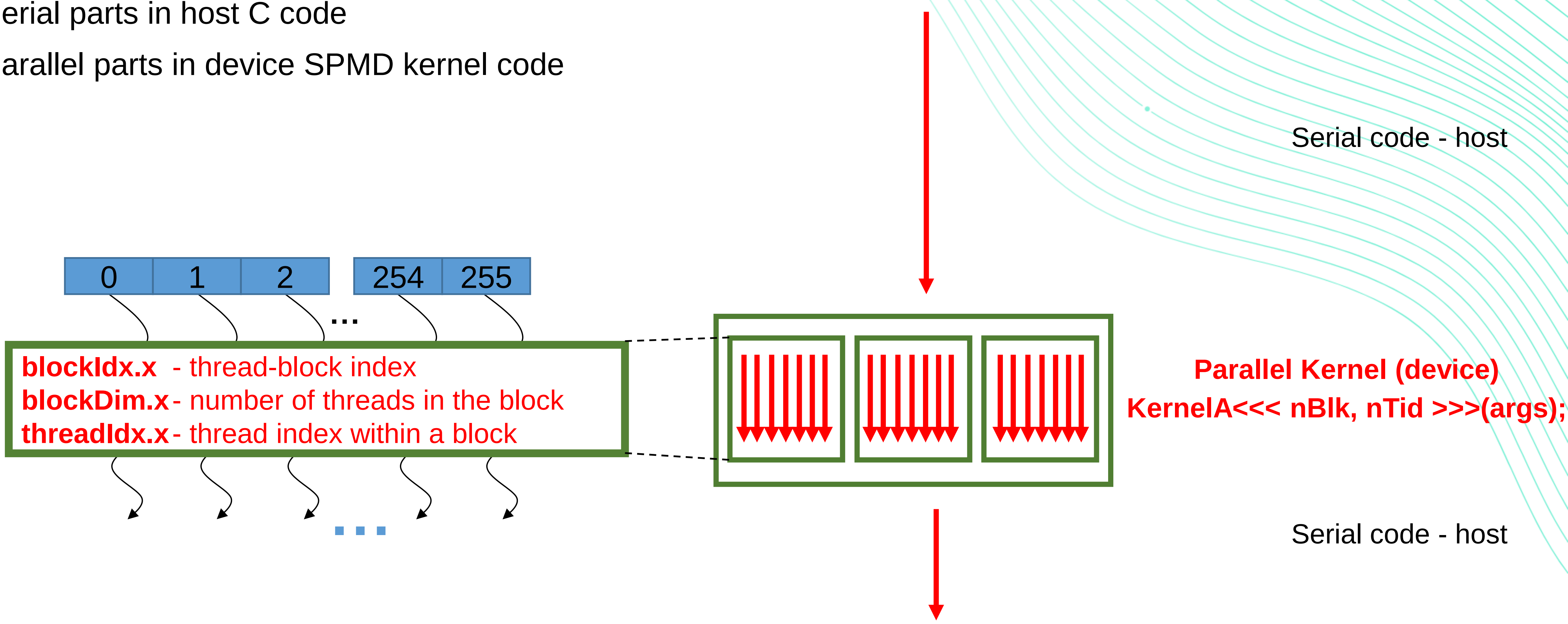
```
void vecAdd(float *A, float *B, float *C, int n)
{
  // Kernel run
}
```

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CUDA programming
CUDA Execution Model

Heterogeneous host (CPU) + device (GPU) application C program

- Serial parts in host C code
- Parallel parts in device SPMD kernel code


0
1
2
…
254
255
blockIdx.x - thread-block index
blockDim.x - number of threads in the block
threadIdx.x - thread index within a block
…
Serial code - host
Parallel Kernel (device)
KernelA<<< nBlk, nTid >>>(args);
Serial code - host

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CUDA programming
Device Code / Kernel

Device code or kernel

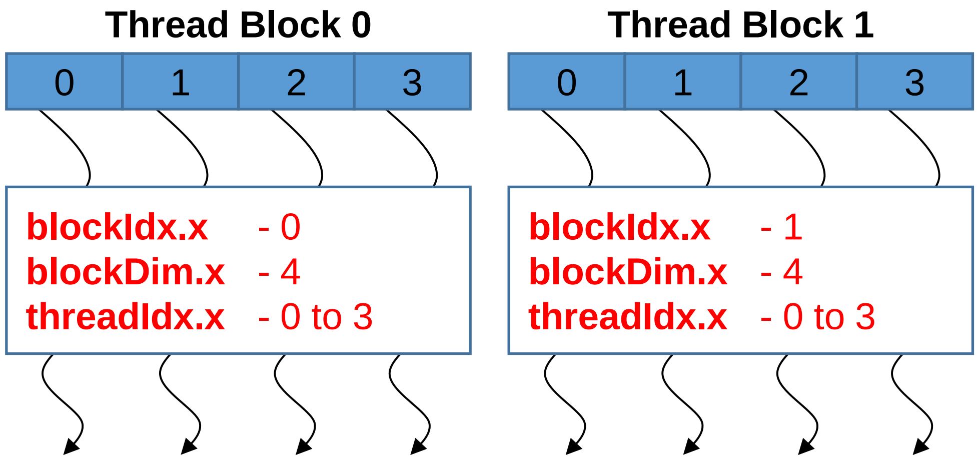
- **__global__** defines a kernel function

Host code – kernel execution

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

Grid dimension = # of blocks

Block dimension = # of threads per block



Thread Block 0:
blockIdx.x - 0
blockDim.x - 4
threadIdx.x - 0 to 3

Thread Block 1:
blockIdx.x - 1
blockDim.x - 4
threadIdx.x - 0 to 3

Kernel Code

```
__global__ void say_hello()
{
  int global_index  = blockIdx.x * blockDim.x + threadIdx.x;
  int total_threads = blockDim.x * gridDim.x;
  printf("Hello from thread %d,
          block %d,
          my global index is %d,
          total number of threads is %d\n",
    threadIdx.x,
    blockIdx.x,
    global_index,
    total_threads);
}
```

Each thread uses indices to decide what data to work on

- **blockIdx.x** – block index in x direction
- **threadIdx.x** – thread index in x direction
- **blockDim.x** – block size (# of threads per block) in x dir.

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

Lunch break

EPICURE

Unlocking European-level HPC Support

Hands-On
Hello world in CUDA

Hands-On
Hello world in CUDA

- Start simple with a classic hello world
- `tasks/hello_world/hello_world.cu`
- Print info in each thread
  - Thread index, number of threads in block (=block size)
  - Block index, number of blocks (=grid size)
  - Global index of thread, total number of threads (need to calculate first)
- Compile using
  - `nvcc hello_world.cu -o hello_world.x`
- And run with
  - `./hello_world.x`

Sample output (might be in different order):

```
Launching the kernel with 2 blocks, each with 4 threads
Kernel was launched, waiting for its completion
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
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
Kernel execution completed
```

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

EPICURE

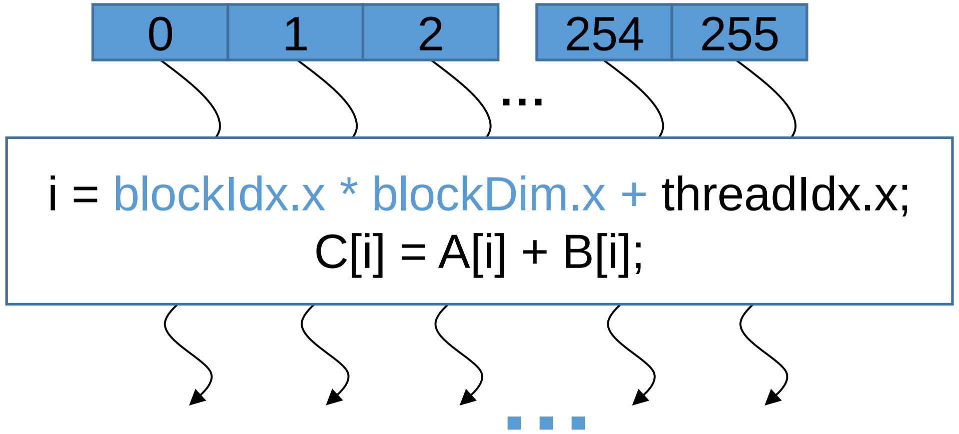
Unlocking European-level HPC Support

CUDA Programming cont.

CUDA programming
Arrays of Parallel Threads

A CUDA kernel is executed by a grid (array) of threads

- All threads in a grid run the same kernel code (Single Program Multiple Data)
- Each thread has indexes that it uses to compute memory addresses and make control decisions



```
i = blockIdx.x * blockDim.x + threadIdx.x;
C[i] = A[i] + B[i];
```

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

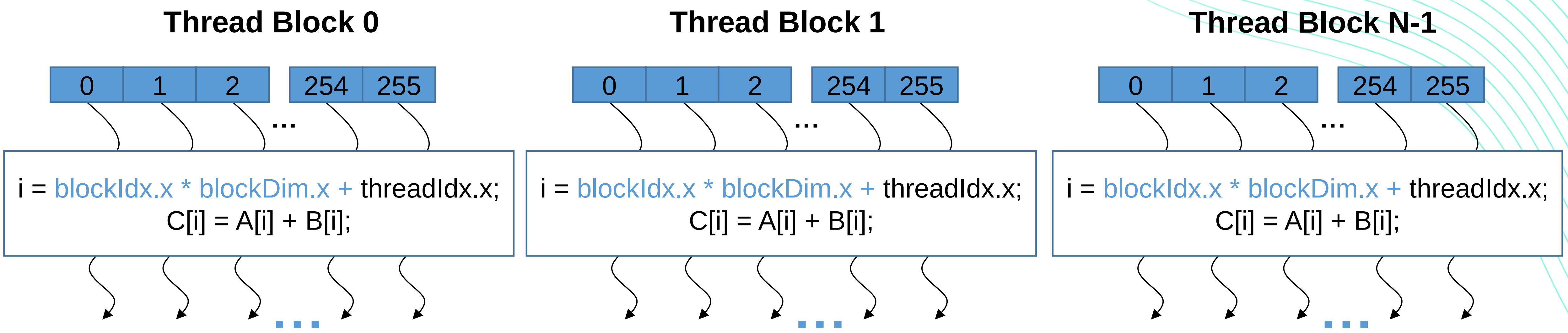
CUDA programming
Thread Blocks

Divide thread array into multiple blocks

- **Threads within a block cooperate** via
  - shared memory,
  - atomic operations and
  - barrier synchronization
- **Threads in different blocks do not interact**

blockIdx and threadIdx

- Each thread uses indices to decide what data to work on
  - blockIdx: 1D, 2D, or 3D
  - threadIdx: 1D, 2D, or 3D



Thread Block 0

```
i = blockIdx.x * blockDim.x + threadIdx.x;
C[i] = A[i] + B[i];
```

Thread Block 1

```
i = blockIdx.x * blockDim.x + threadIdx.x;
C[i] = A[i] + B[i];
```

Thread Block N-1

```
i = blockIdx.x * blockDim.x + threadIdx.x;
C[i] = A[i] + B[i];
```

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CUDA programming
Vector Addition Kernel

Device code or kernel

- compute vector sum C = A + B
- each thread performs one pair-wise addition

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

__global__ defines a kernel function

- each “__” consists of two underscore characters
- kernel function must return void

Each thread uses indices to decide what data to work on

- **blockIdx.x** – block index in x direction
- **threadIdx.x** – thread index in x direction
- **blockDim.x** – block size (# of threads per block) in x dir.
- Note: 1D indexing uses .x only, 2D uses .x, .y and 3D uses .x, .y, .z

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CUDA programming
Vector Addition Kernel Launch

Host code

- Kernel execution – host code that launches kernel
- GPU hardware creates a grid of threads
- each thread executes the kernel function from previous slide

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
 // d_A, d_B, d_C allocations and memory copies are done
 //      x y z direction
 dim3 DimGrid (2, 1, 1);   // number of blocks per grid to be launched
 dim3 DimBlock(4, 1, 1);   // number of threads per block to be launched
 vecAddKernel<<<DimGrid,DimBlock>>>(d_A, d_B, d_C, n);
}
```

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CUDA programming
Vector Addition Kernel Launch

Host code

- Kernel execution – host code that launches kernel
- GPU hardware creates a grid of threads
- each thread executes the kernel function from previous slide

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
 // d_A, d_B, d_C allocations and memory copies are done
 // launches 2 block in a grid and 4 threads per block
 vecAddKernel<<<2,4>>>(d_A, d_B, d_C, n);}
}
```

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CUDA programming
Vector Addition Kernel Launch

Host code

- Executes ceil(n/256.0) blocks of 256 threads each
- the ceiling function makes sure that there are enough threads to cover all elements.

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
 // d_A, d_B, d_C allocations and memory copies are done
 vecAddKernel<<<ceil(n/256.0),256>>>(d_A, d_B, d_C, n);
}
```

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CUDA programming
Vector Addition Kernel Launch

Host code

- This is an equivalent way to express the ceiling function.

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
 // d_A, d_B, d_C allocations and memory copies are done
 dim3 DimGrid((n-1)/256 + 1, 1, 1);
 dim3 DimBlock(256, 1, 1);
 vecAddKernel<<<DimGrid,DimBlock>>>(d_A, d_B, d_C, n);
}
```

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CUDA programming
Vector Addition Kernel Launch

- Host: launches „extra“ block to cover all elements – ensures that there is enough threads to process all elements
- Kernel: controls that thread does not read unallocated memory

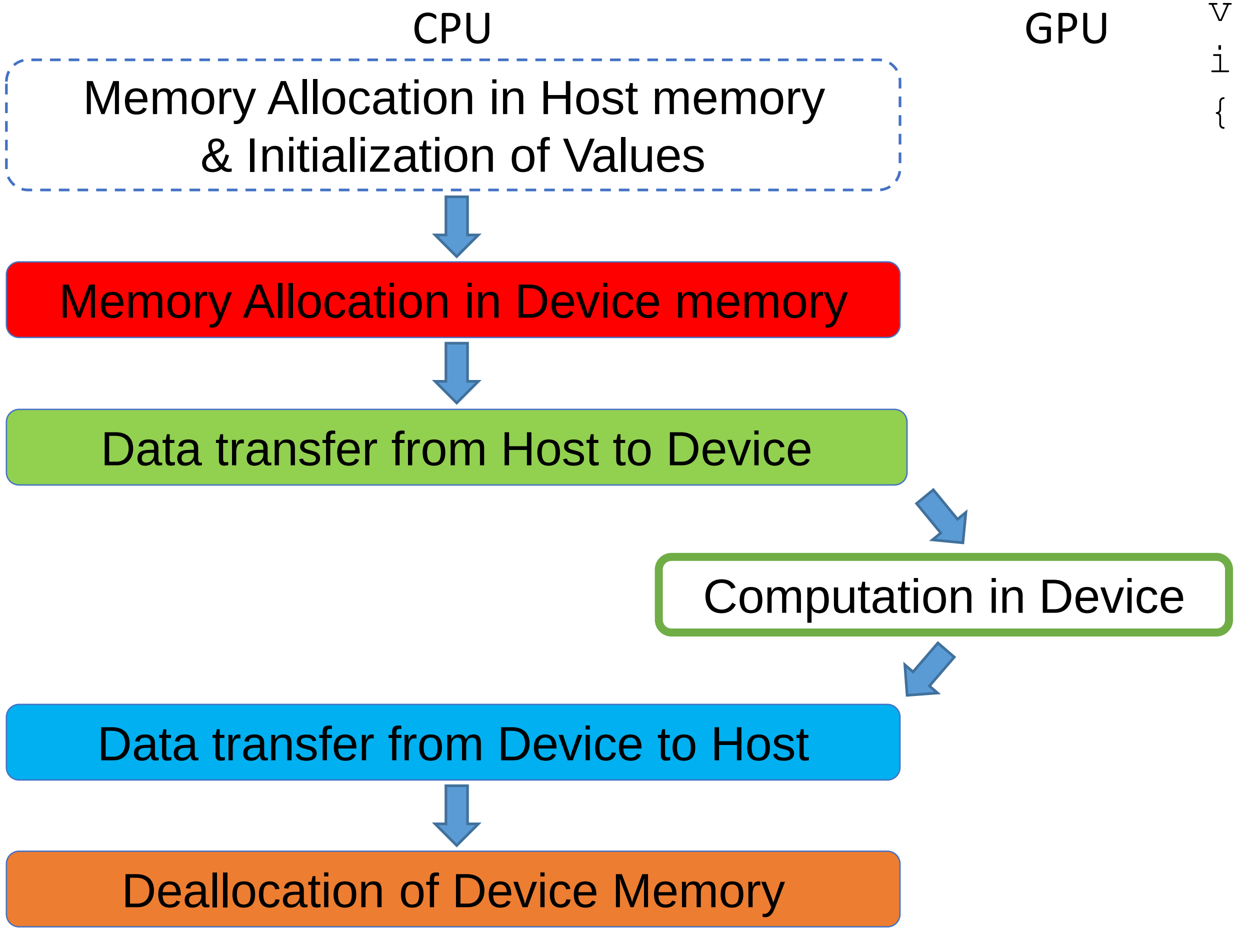
- Host: DimBlock equals to
- Kernel: blockDim

- Kenel: threadIdx is in range <0, DimBlock)
- Kenel: blockIdx is in range <0, DimGrid)

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
 dim3 DimGrid( ceil(n/256.0) , 1, 1);
 dim3 DimBlock(256, 1, 1);
 vecAddKernel<<<DimGrid,DimBlock>>>(d_A, d_B, d_C, n);
}

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

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CUDA programming
Vector Addition – with kernel exec.



CPU

GPU

Memory Allocation in Host memory & Initialization of Values

Memory Allocation in Device memory

Data transfer from Host to Device

Computation in Device

Data transfer from Device to Host

Deallocation of Device Memory

```
void vecAdd(float *h_A, float *h_B, float *h_C,
int n)
{
   int size = n * sizeof(float);
   float *d_A, *d_B, *d_C;
   cudaMalloc((void **) &d_A, size);
   cudaMalloc((void **) &d_B, size);
   cudaMalloc((void **) &d_C, size);

   cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

   vecAddKernel<<<ceil(n/256.0),256>>>
     (d_A, d_B, d_C, n);

   cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

   cudaFree(d_A);
   cudaFree(d_B);
   cudaFree(d_C);
}
```

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CUDA programming
Kernel timing using events

Use CUDA **Events** for timing CUDA related execution time.

- Works as "markers" in execution queue
- Besides timing, they are crucial for GPU synchronization
- Important! In order to compute elapsed time correctly. Both events must "happen". That is, they need to reach the end of execution queue
- Can be ensured by waiting for the event to "happen" using `cudaEventSynchronize()` or synchronization with entire GPU by `cudaDeviceSynchronize()`

```
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
   …
   float timeInMs;
   cudaEvent_t startEvent, endEvent;

   cudaEventCreate(&startEvent);
   cudaEventCreate(&endEvent);

   cudaEventRecord(startEvent);

   vecAddKernel<<<ceil(n/256.0),256>>>
    (d_A, d_B, d_C, n);

   cudaEventRecord(endEvent);

   cudaDeviceSynchronize();
   cudaEventElapsedTime
   (&timeInMs, startEvent, endEvent);

   cudaEventDestroy(endEvent);
   cudaEventDestroy(startEvent);
   …
}
```

EPICURE
Unlocking European-level HPC Support

2x Hands-on Vector Addition

manual memcpy; managed memory

CUDA programming
Error checking

- CUDA functions return error code (cudaError_t)

```
cudaError_t cudaMalloc ( void** devPtr, size_t size );
```

- We should check it, so it won’t be silently ignored
- Complicated to do error checking with every CUDA function call
- Create a macro (already provided in our examples)

```
#define CUDACHECK(err) do { cuda_check((err), __FILE__, __LINE__); } while(false)
inline void cuda_check(cudaError_t error_code, const char *file, int line)
{
    if (error_code != cudaSuccess)
    {
        fprintf(stderr, "CUDA Error %d: %s. In file '%s' on line %d\n", error_code, cudaGetErrorString(error_code), file, line);
        exit(error_code);
    }
}
```

- Wrap every function call in the macro and after kernel launch, also check for errors

```
CUDACHECK(cudaMalloc( ... ));
```

```
my_kernel<<< ... >>>(...);
CUDACHECK(cudaPeekAtLastError());
```

Hands-on: vector add with manual memory transfers

- `tasks/vector_add_classic/vector_add_classic.cu`
- C = A + B
- Implement the vector add on GPU yourself
  - Vector already initialized on CPU
  - Allocate memory on GPU
  - Copy vectors to GPU
  - Implement and launch kernel
  - Copy result vector back to CPU
  - Free the allocated GPU memory
  - Use the CUDACHECK() macro to do error checking

Sample output:

```
Input A:
  0.000   1.000   2.000   3.000   4.000   5.000   6.000   7.000   8.000   9.000
Input B:
  0.000  10.000  20.000  30.000  40.000  50.000  60.000  70.000  80.000  90.000
Output C:
  0.000  11.000  22.000  33.000  44.000  55.000  66.000  77.000  88.000  99.000
The result is CORRECT!
```

BONUS for those who are finished: vector_add_pinned

Hands-on: vector add with managed memory

- `tasks/vector_add_managed`
- C = A + B
- Implement the vector add on GPU again, but use managed memory
  - Copy your solution from the previous hands-on
  - Kernel and its launch stays the same
  - Only memory management changes
  - No host and device array, only single array in managed memory
  - Use cudaMallocManaged to allocate the arrays
  - No cudaMemcpy needed
  - cudaDeviceSynchronize required now

Sample output:

```
Input A:
  0.000   1.000   2.000   3.000   4.000   5.000   6.000   7.000   8.000   9.000
Input B:
  0.000  10.000  20.000  30.000  40.000  50.000  60.000  70.000  80.000  90.000
Output C:
  0.000  11.000  22.000  33.000  44.000  55.000  66.000  77.000  88.000  99.000
The result is CORRECT!
```

BONUS for those who are finished: vector_add_thread_mapping

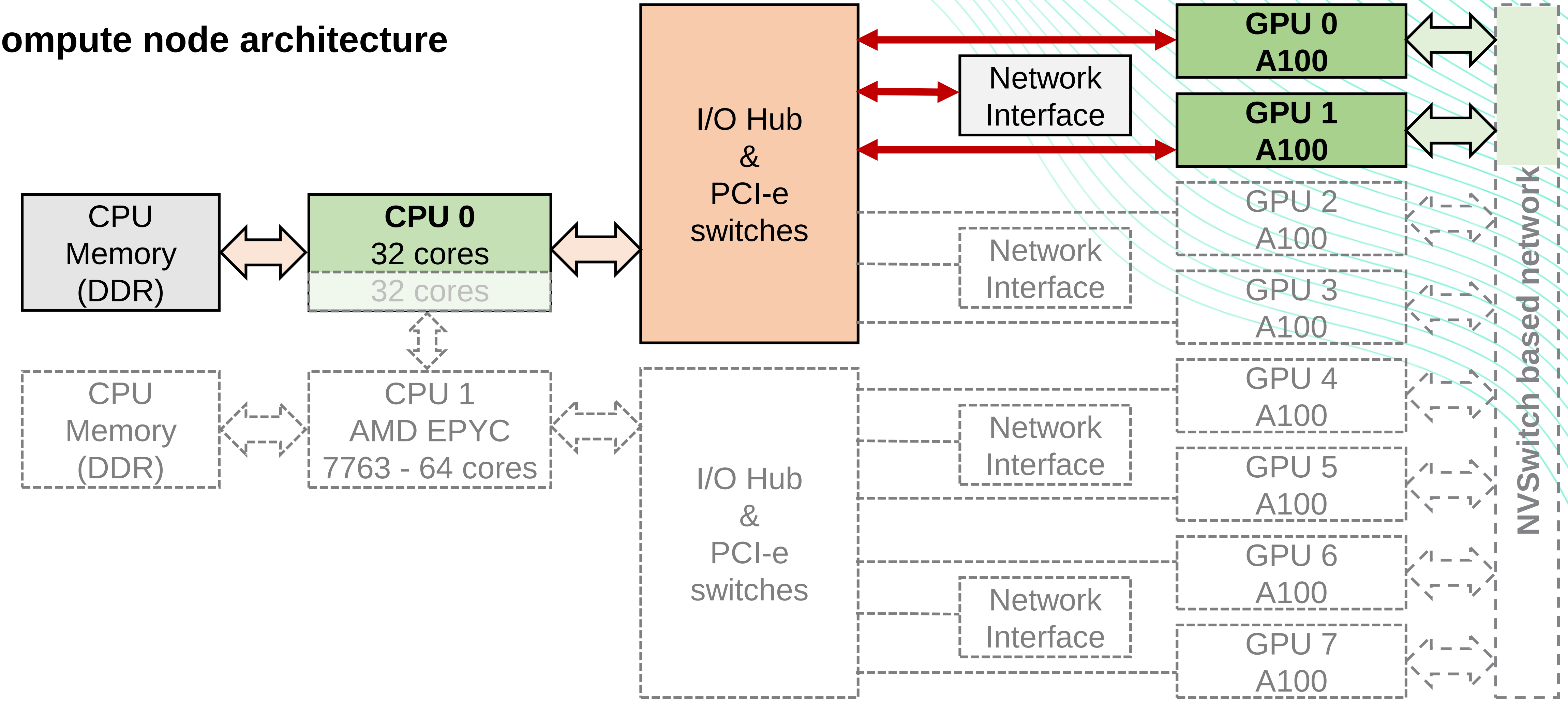
EPICURE

Unlocking European-level HPC Support

Multi-GPU Programming

CUDA programming
MultiGPU programing basics

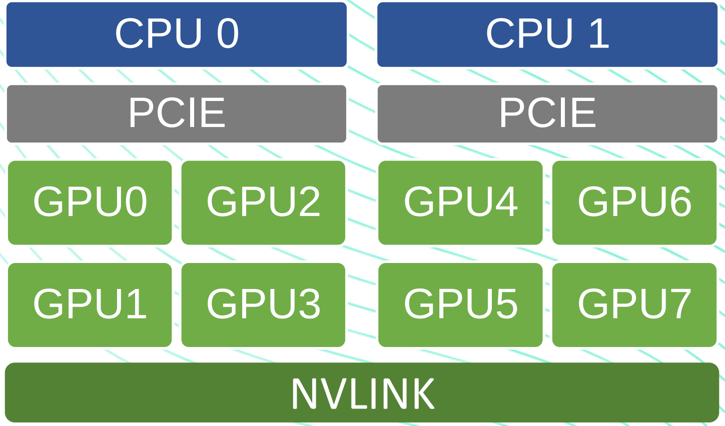
Compute node architecture



CUDA programming
MultiGPU programing basics

Multi-GPU system

- GPU's are numbered from 0 to n-1, where n is the number of GPU’s.
- The CUDA driver always starts with a default active device.
- There are two broad types of Multi GPU communication:
  - Through the PCIE bus
  - Through NVLINK



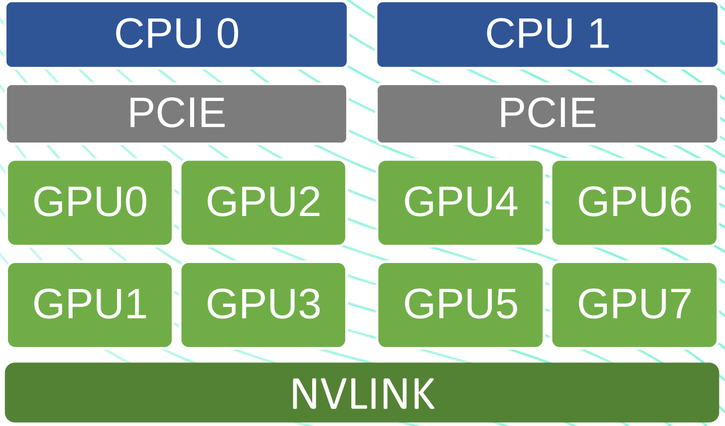
```
$ nvidia-smi topo -m

        GPU0    GPU1    GPU2    GPU3    GPU4    GPU5    GPU6    GPU7    mlx5_0    mlx5_1    mlx5_2    mlx5_3    CPU Affinity    NUMA Affinity
GPU0     X      NV12    NV12    NV12    NV12    NV12    NV12    NV12    SYS       PXB       SYS       SYS       48-63           3
GPU1    NV12     X      NV12    NV12    NV12    NV12    NV12    NV12    SYS       PXB       SYS       SYS       48-63           3
GPU2    NV12    NV12     X      NV12    NV12    NV12    NV12    NV12    PXB       SYS       SYS       SYS       16-31           1
GPU3    NV12    NV12    NV12     X      NV12    NV12    NV12    NV12    PXB       SYS       SYS       SYS       16-31           1
GPU4    NV12    NV12    NV12    NV12     X      NV12    NV12    NV12    SYS       SYS       SYS       PXB       112-127         7
GPU5    NV12    NV12    NV12    NV12    NV12     X      NV12    NV12    SYS       SYS       SYS       PXB       112-127         7
GPU6    NV12    NV12    NV12    NV12    NV12    NV12     X      NV12    SYS       SYS       PXB       SYS       80-95           5
GPU7    NV12    NV12    NV12    NV12    NV12    NV12    NV12     X      SYS       SYS       PXB       SYS       80-95           5
mlx5_0  SYS     SYS     PXB     PXB     SYS     SYS     SYS     SYS      X        SYS       SYS       SYS
mlx5_1  PXB     PXB     SYS     SYS     SYS     SYS     SYS     SYS     SYS        X        SYS       SYS
mlx5_2  SYS     SYS     SYS     SYS     SYS     SYS     PXB     PXB     SYS       SYS        X        SYS
mlx5_3  SYS     SYS     SYS     SYS     PXB     PXB     SYS     SYS     SYS       SYS       SYS        X
```

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CUDA programming
CUDA host API calls for Multi GPU's

cudaSetDevice()

- Set GPU device to use for device code execution on the active host thread.
- Requires one parameter:
    - An int with the device id number
- This function doesn't affect other host threads, meaning that setting the device on one thread will not set the device in other host threads. Also doesn't affect previous async calls.



cudaGetDevice()

- Get GPU device being currently used by the active host thread
- Requires one parameter:
    - An int pointer to store the device id

cudaGetDeviceCount()

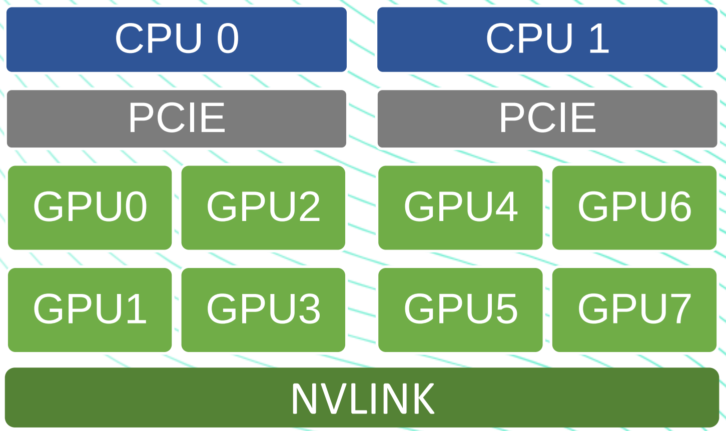
- Get the number of CUDA-capable devices in the system.
- Requires one parameter:
    - An int pointer to store the device count

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CUDA programming
CUDA host API calls for Multi GPU's

cudaSetDevice()

- Set GPU device to use for device code execution on the active host thread.
- Requires one parameter:
  - An int with the device id number
- This function doesn’t affect other host threads, meaning that setting the device on one thread will not set the device in other host threads. Also doesn’t affect previous async calls.



Memory allocation

To allocate or associate memory with a specific device using non-Managed CUDA-API calls, **it’s necessary to call cudaSetDevice() before doing the allocation call**.

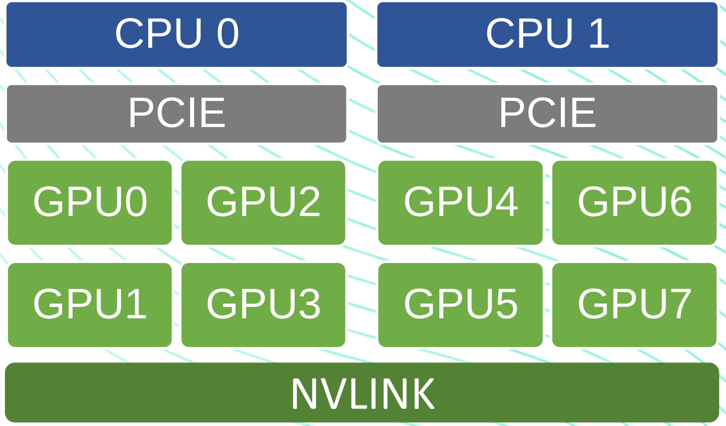
- cudaMalloc() - allocates an object in the device global memory
- cudaHostAlloc(), cudaMallocHost() - allocates pinned memory on the host

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CUDA programming

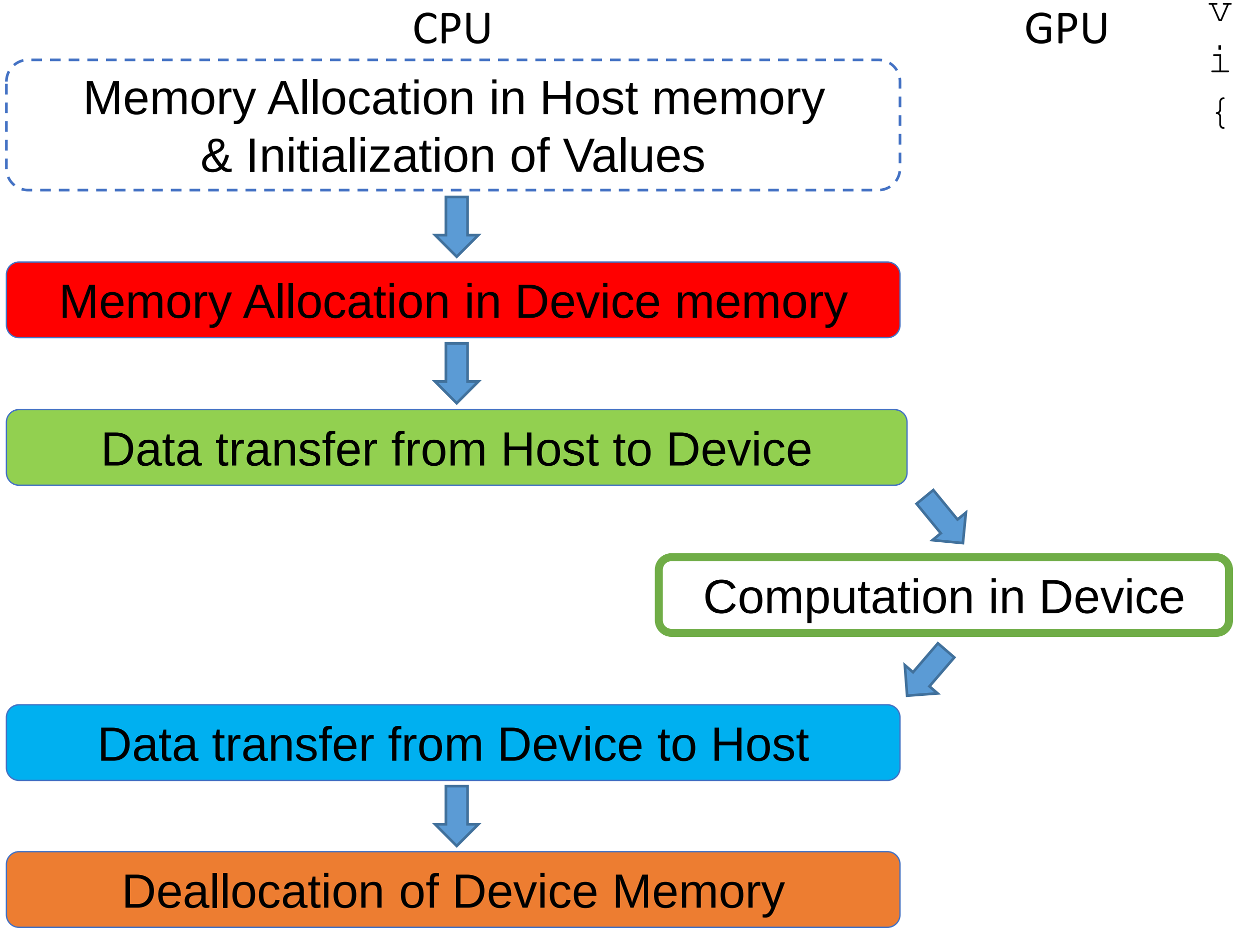
CUDA runtime calls affected by cudaSetDevice

- If **cudaSetDevice()** was called before a kernel launching call, the kernel will execute in the active device.
  - It’s crucial that every non managed memory being used in the kernel resides in the active device, otherwise an error will occur.
- If **cudaSetDevice()** was called before a **cudaStreamCreate()**, then the stream will be associated with the active device.
- The synchronization functions: **cudaDeviceSynchronize()**, **cudaStreamSynchronize()** are also affected by **cudaSetDevice()**, synchronizing tasks only for the active device on the active host thread



Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

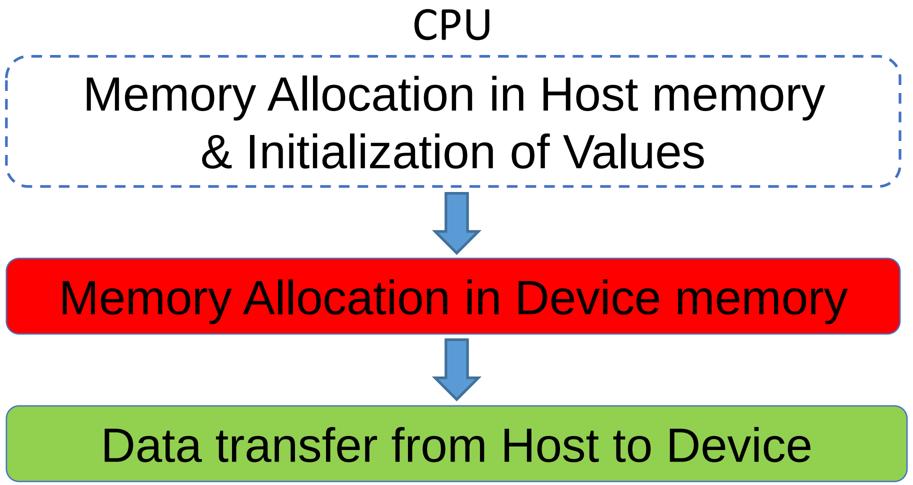
CUDA programming
Vector Addition – with kernel exec.


CPU
GPU
Memory Allocation in Host memory & Initialization of Values
Memory Allocation in Device memory
Data transfer from Host to Device
Computation in Device
Data transfer from Device to Host
Deallocation of Device Memory

```
void vecAdd(float *h_A, float *h_B, float *h_C,
int n)
{
  int size = n * sizeof(float);
  float *d_A, *d_B, *d_C;
  cudaMalloc((void **) &d_A, size);
  cudaMalloc((void **) &d_B, size);
  cudaMalloc((void **) &d_C, size);

  cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
   cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

  vecAddKernel<<<ceil(n/256.0),256>>>
   (d_A, d_B, d_C, n);

  cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

  cudaFree(d_A);
  cudaFree(d_B);
  cudaFree(d_C);
}
```

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CUDA programming
Multi-GPU Vector Addition – Part 1



CPU

Memory Allocation in Host memory & Initialization of Values

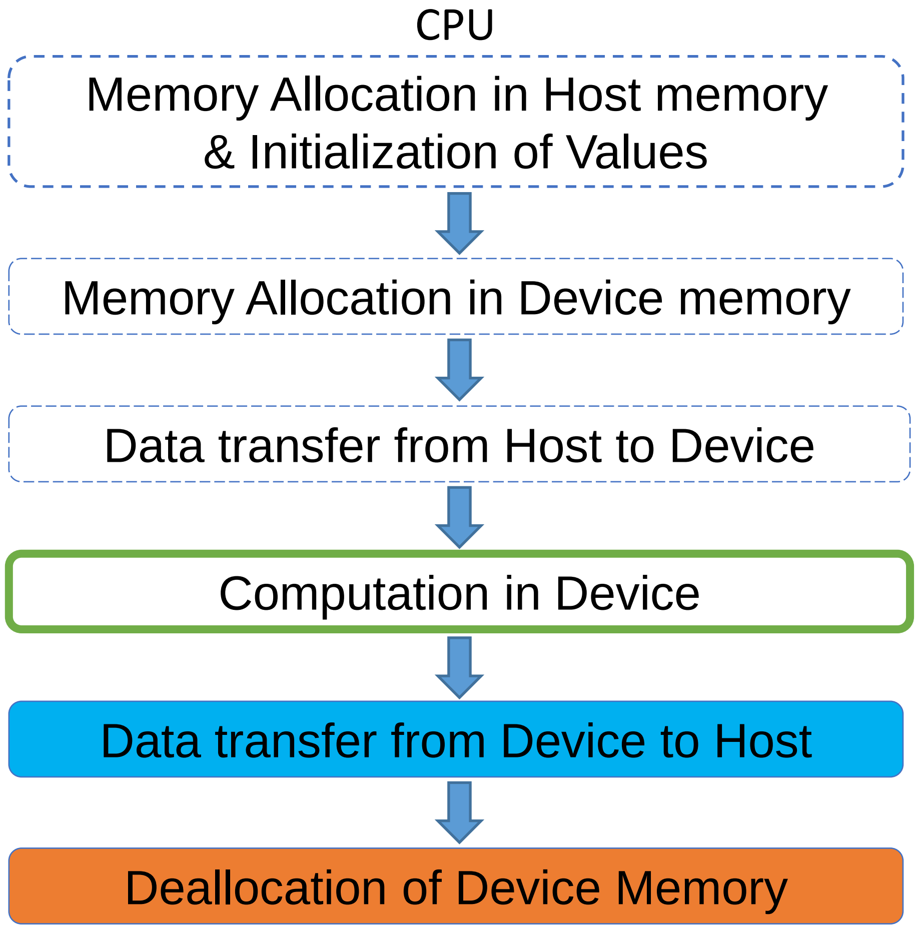
Memory Allocation in Device memory

Data transfer from Host to Device

```
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
 int n0 = n / 2;
 int n1 = n - n0;
 int size0 = n0 * sizeof(float);
 int size1 = n1 * sizeof(float);
 float *d_A0, *d_B0, *d_C0;
 float *d_A1, *d_B1, *d_C1;

 cudaSetDevice(0);
 cudaMalloc((void **) &d_A0, size0);
 cudaMalloc((void **) &d_B0, size0);
 cudaMalloc((void **) &d_C0, size0);
 cudaMemcpy(d_A0, &h_A[0], size0, cudaMemcpyHostToDevice);
 cudaMemcpy(d_B0, &h_B[0], size0, cudaMemcpyHostToDevice);

 cudaSetDevice(1);
 cudaMalloc((void **) &d_A1, size1);
 cudaMalloc((void **) &d_B1, size1);
 cudaMalloc((void **) &d_C1, size1);
 cudaMemcpy(d_A1, &h_A[n0], size1, cudaMemcpyHostToDevice);
 cudaMemcpy(d_B1, &h_B[n0], size1, cudaMemcpyHostToDevice);
```

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CUDA programming
Multi-GPU Vector Addition – Part 2



CPU

Memory Allocation in Host memory & Initialization of Values

Memory Allocation in Device memory

Data transfer from Host to Device

Computation in Device

Data transfer from Device to Host

Deallocation of Device Memory

```
int n0 = floor(n/2.0);
int n1 = ceil(n/2.0);
int size0 = n0 * sizeof(float);
int size1 = n1 * sizeof(float);

cudaSetDevice(0);
vecAddKernel<<<ceil(n0/256.0),256>>> (d_A0, d_B0, d_C0, n0);

cudaSetDevice(1);
vecAddKernel<<<ceil(n1/256.0),256>>> (d_A1, d_B1, d_C1, n1);

cudaMemcpy(&h_C[0], d_C0, size, cudaMemcpyDeviceToHost);
cudaMemcpy(&h_C[n0],d_C1, size, cudaMemcpyDeviceToHost);

cudaFree(d_A0); cudaFree(d_A1);
cudaFree(d_B0); cudaFree(d_B1);
cudaFree(d_C0); cudaFree(d_C1);
}
```

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CUDA programming
GPU selection

Environment variable controlling devices visibility

- Useful for selecting or restricting the set of available GPUs for specific application even without the access to the source code
- Execute `export CUDA_VISIBLE_DEVICES=<comma separated list of GPU IDs>` before running the app
- To list all available GPU IDs run `nvidia-smi` from command line

- Single GPU applications:

```
export CUDA_VISIBLE_DEVICES=0 ./app
```

- Multi GPU applications:

```
export CUDA_VISIBLE_DEVICES=0,1 ./app
```

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

EPICURE

Unlocking European-level HPC Support

Hands-on
Multi-GPU Vector Addition

Hands-on: vector add, multi-GPU

- `tasks/vector_add_multigpu`
- Use multiple GPUs to perform the vector add operation
- 2 GPUs available today
- Start from the solution to the vector_add_managed hands-on
- Launch the kernel 2 times, once for each GPU
  - Calculate indexes where each GPU should start and end
  - Don’t modify the kernel, just pass it different arguments
- Use cudaSetDevice() to set the currently used GPU device
- Try to write the code for any number of GPUs
  - Use cudaGetDeviceCount() to query the number of GPUs
  - Loop through every GPU on the system

EPICURE
Unlocking European-level HPC Support

Coffee break ☕

EPICURE

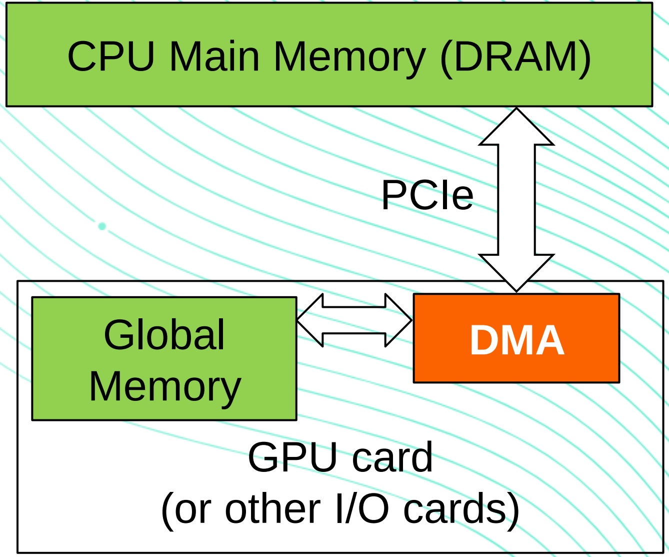
Unlocking European-level HPC Support

Efficient Host-Device Data Transfer and CUDA Streams

CPU-GPU Data Transfer using DMA

CPU-GPU Data Transfer using DMA

- DMA (Direct Memory Access) hardware is used by **cudaMemcpy()** for better efficiency
  - CPU is not used and perform useful calculations
  - DMA is hardware unit used to transfer given number of bytes
    - between physical memory address space regions
  - uses system interconnect: in current systems PCI-Express



Virtual Memory Management

- **Problem for DMA**: *not all variables and data structures are always located in the physical memory*

Data Transfer and Virtual Memory

- DMA uses **ONLY** physical addresses
- when cudaMemcpy() copies an array, it is implemented as one or more DMA transfers

Solution: Pinned Memory

- pinned memory are virtual memory pages that are specially selected, and they cannot be paged out (removed from physical memory)
- pinned memory is allocated with a special system API function call

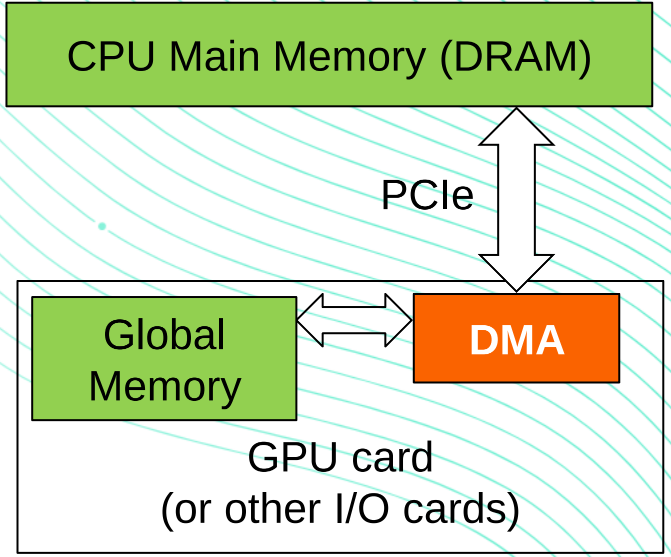
CPU memory that serve as the source or destination of a DMA transfer must be allocated as pinned memory

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CPU-GPU Data Transfer using DMA

CUDA data transfer uses pinned memory.

- the DMA used by *cudaMemcpy()* requires that any source or destination in the host memory is allocated as pinned memory
- if a source or destination of a *cudaMemcpy()* in the host memory is not allocated in pinned memory, it needs to be first copied to a pinned memory – extra overhead
- *cudaMemcpy()* is faster if the host memory source or destination is allocated in pinned memory since no extra copy is needed



Using Pinned Memory in CUDA

- use the allocated pinned memory and its pointer the same way as those returned by malloc();
- the only difference is that the allocated memory cannot be paged by the OS
- the cudaMemcpy() function should be about 2X faster with pinned memory
- pinned memory is a limited resource
- over-subscription can have serious consequences

Allocate/Free Pinned Memory

cudaHostAlloc(), three parameters

- Address of pointer to the allocated memory
- Size of the allocated memory in bytes
- Option – use cudaHostAllocDefault for now

cudaFreeHost(), one parameter

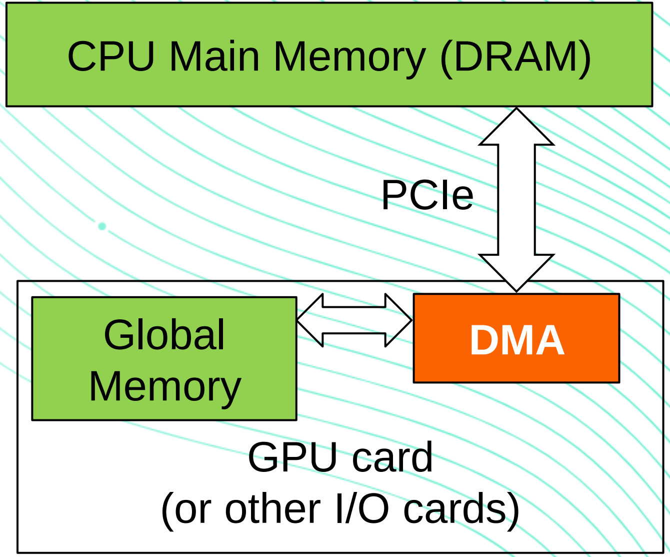
- Pointer to the memory to be freed

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

Pinned Memory

Example: Vector Addition Host Code

```
int main()
{
  float *h_A, *h_B, *h_C;
  …
  cudaHostAlloc((void **) &h_A, N* sizeof(float), cudaHostAllocDefault);
  cudaHostAlloc((void **) &h_B, N* sizeof(float), cudaHostAllocDefault);
  cudaHostAlloc((void **) &h_C, N* sizeof(float), cudaHostAllocDefault);
  …
  // cudaMemcpy() runs 2X faster
}
```



Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

Concurrency using CUDA Streams

System can perform multiple CUDA operations simultaneously:

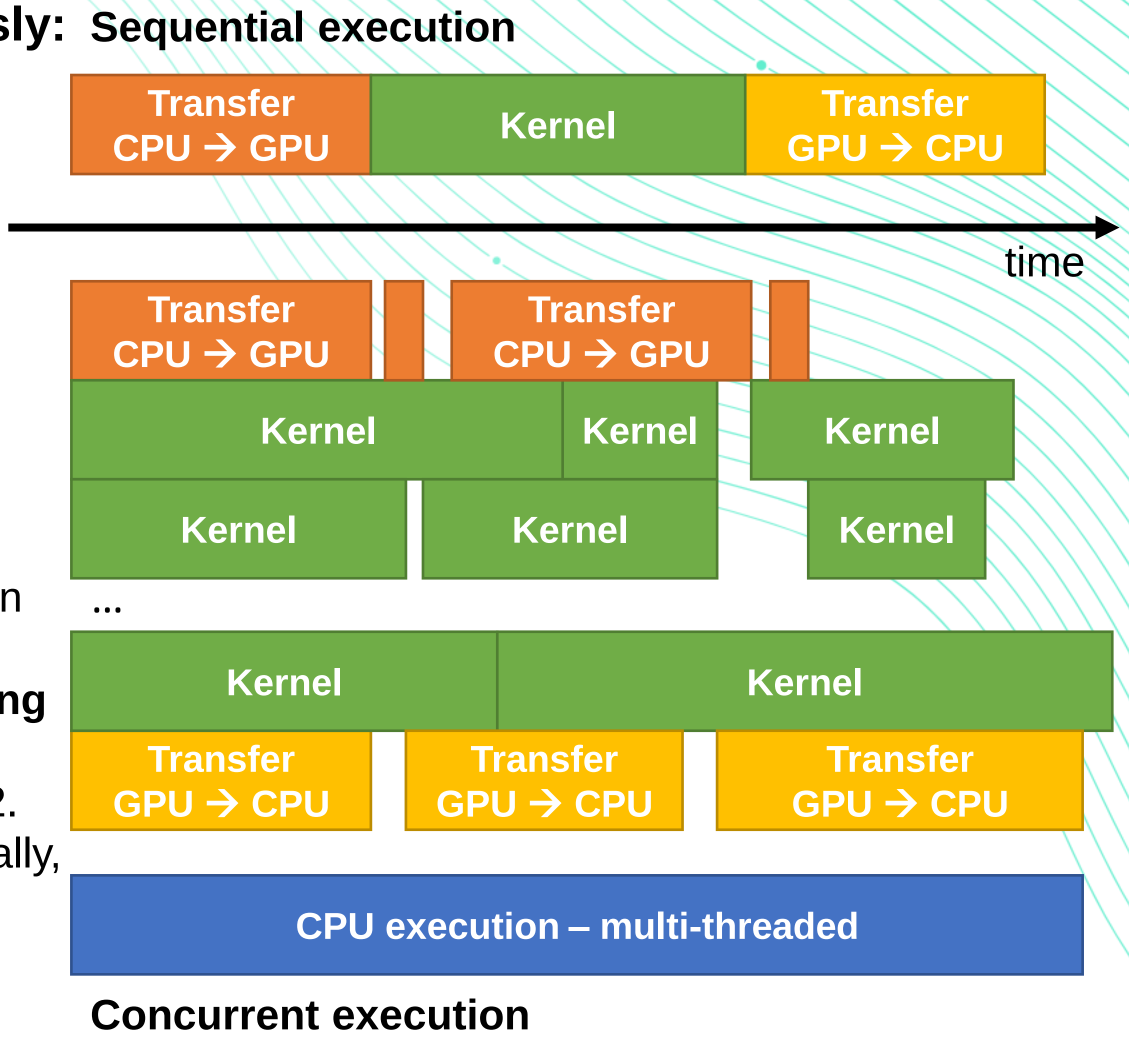
- multiple CUDA kernels on GPU
- one cudaMemcpyAsync from Host to Device
- one cudaMemcpyAsync from Device to Host
- computation on the CPU

CUDA Stream

- **a sequence of operations that execute in issue-order on the GPU**

Stream Semantics

- **Two operations issued into the same stream will execute in issue-order.** Operation B issued after Operation A will not begin to execute until Operation A has completed.
- **Two operations issued into separate streams have no ordering prescribed by CUDA.** Operation A issued into stream 1 may execute before, during, or after Operation B issued into stream 2.
- Operation: Usually, cudaMemcpyAsync or a kernel call. More generally, most CUDA API calls that take a stream parameter, as well as stream callbacks.



Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

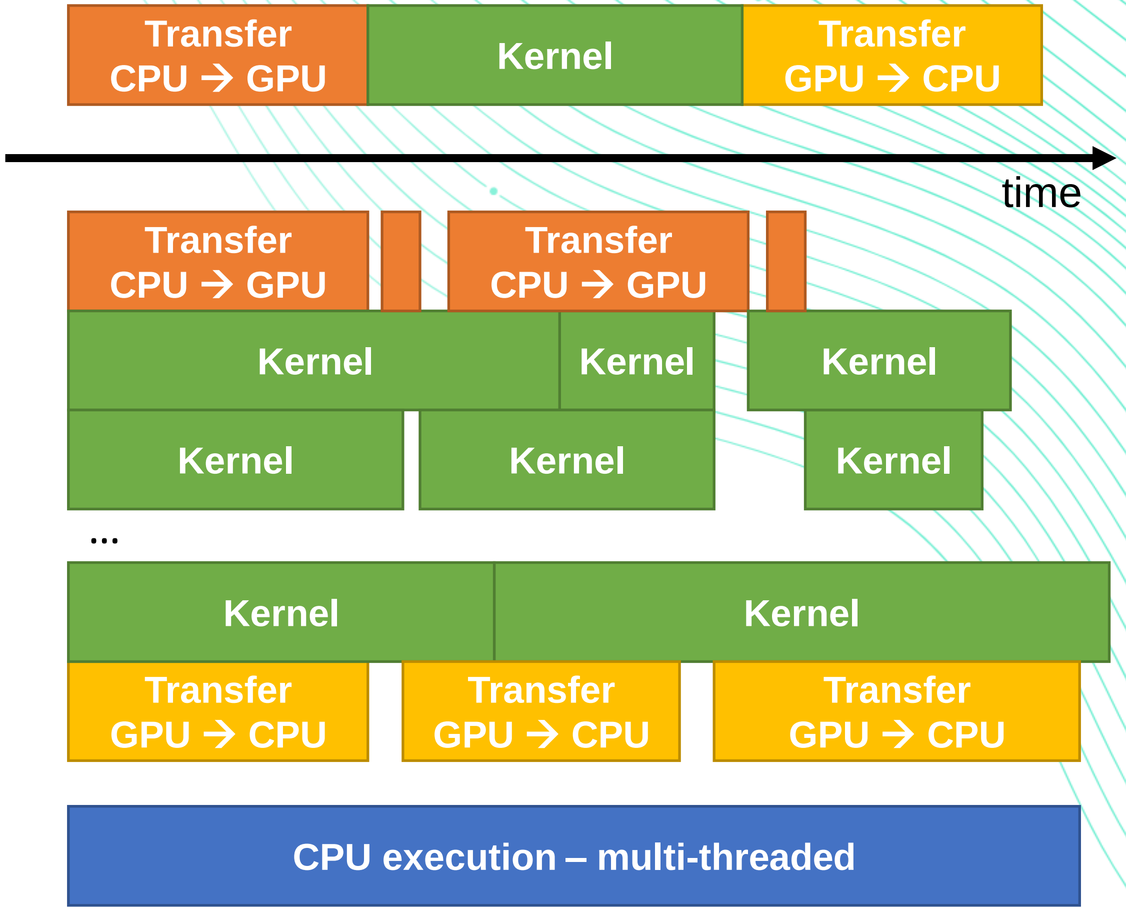
Concurrency using CUDA Streams

Default Stream (aka Stream '0')

- Stream used when no stream is specified
- Completely synchronous w.r.t. host and device
  - As if cudaDeviceSynchronize() inserted before and after every CUDA operation
- Exceptions – asynchronous w.r.t.
  - hostKernel launches in the default stream
  - cudaMemcpy*Async
  - cudaMemset*Async
  - cudaMemcpy within the same device
  - H2D cudaMemcpy of 64kB or less

Requirements for Concurrency

Sequential execution



Transfer CPU → GPU

Kernel

Transfer GPU → CPU

time

Transfer CPU → GPU

Transfer CPU → GPU

Kernel

Kernel

Kernel

Kernel

Kernel

Kernel

...

Kernel

Kernel

Transfer GPU → CPU

Transfer GPU → CPU

Transfer GPU → CPU

CPU execution – multi-threaded

Concurrent execution

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

Concurrency using CUDA Streams

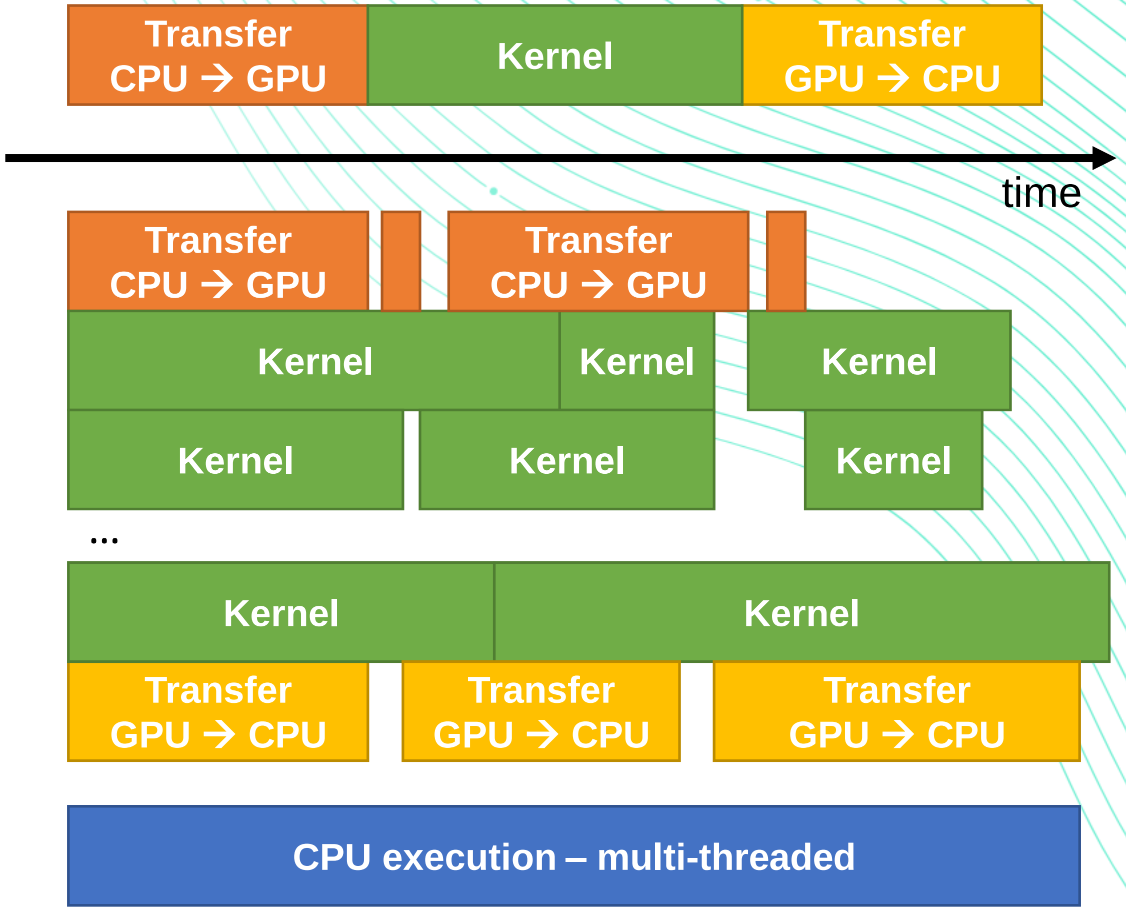
CUDA Streams – How to use them?

- **Create/Destroy**
  - cudaStream_t **stream**;
  - cudaStreamCreate(&**stream**);
  - cudaStreamDestroy(**stream**);

- **Launch**
  - my_kernel<<<grid,block,0,**stream**>>>(...);
  - cudaMemcypAsync(..., **stream**);

- **Synchronize**
  - cudaStreamSynchronize(**stream**);

Sequential execution



Concurrent execution

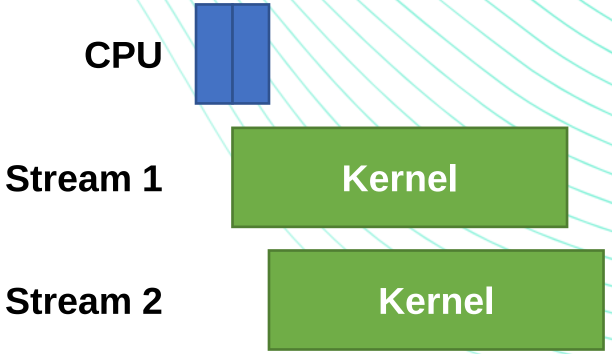
Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

Concurrency using CUDA Streams

Basic Example 1: KERNEL CONCURRENCY

- assume foo only utilizes 50% of the GPU
- using user streams

```
cudaStream_t stream1, stream2;

cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

foo<<<blocks,threads,0,stream1>>>();
foo<<<blocks,threads,0,stream2>>>();

cudaStreamDestroy(stream1);
cudaStreamDestroy(stream2);
```



Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

Concurrency using CUDA Streams

Basic Example 2: CONCURRENT MEMORY COPIES

- assume pinned memory

Synchronous

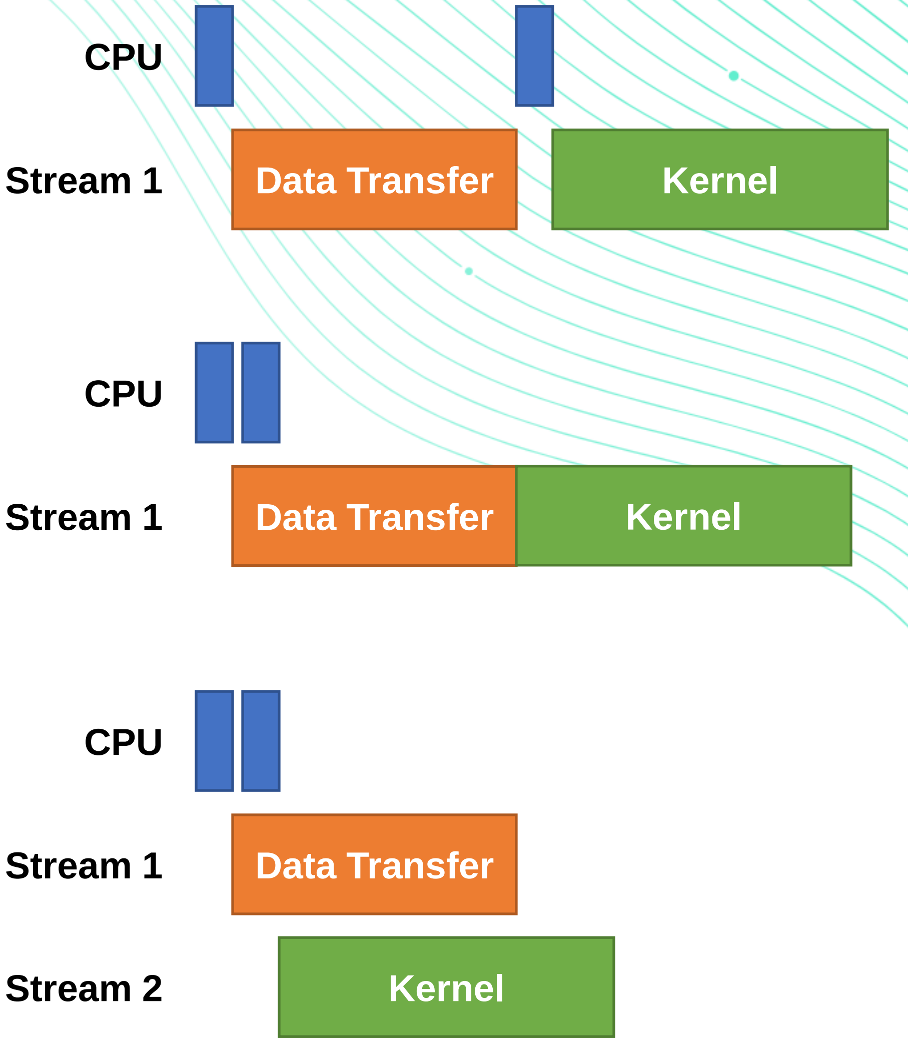
```
cudaMemcpy(...);
foo<<<...>>>();
```

Asynchronous Same Stream

```
cudaMemcpyAsync(...,stream1);
foo<<<...,stream1>>>();
```

Asynchronous Different Streams

```
cudaMemcpyAsync(...,stream1);
foo<<<...,stream2>>>();
```



Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CPU-GPU Data Transfer using DMA

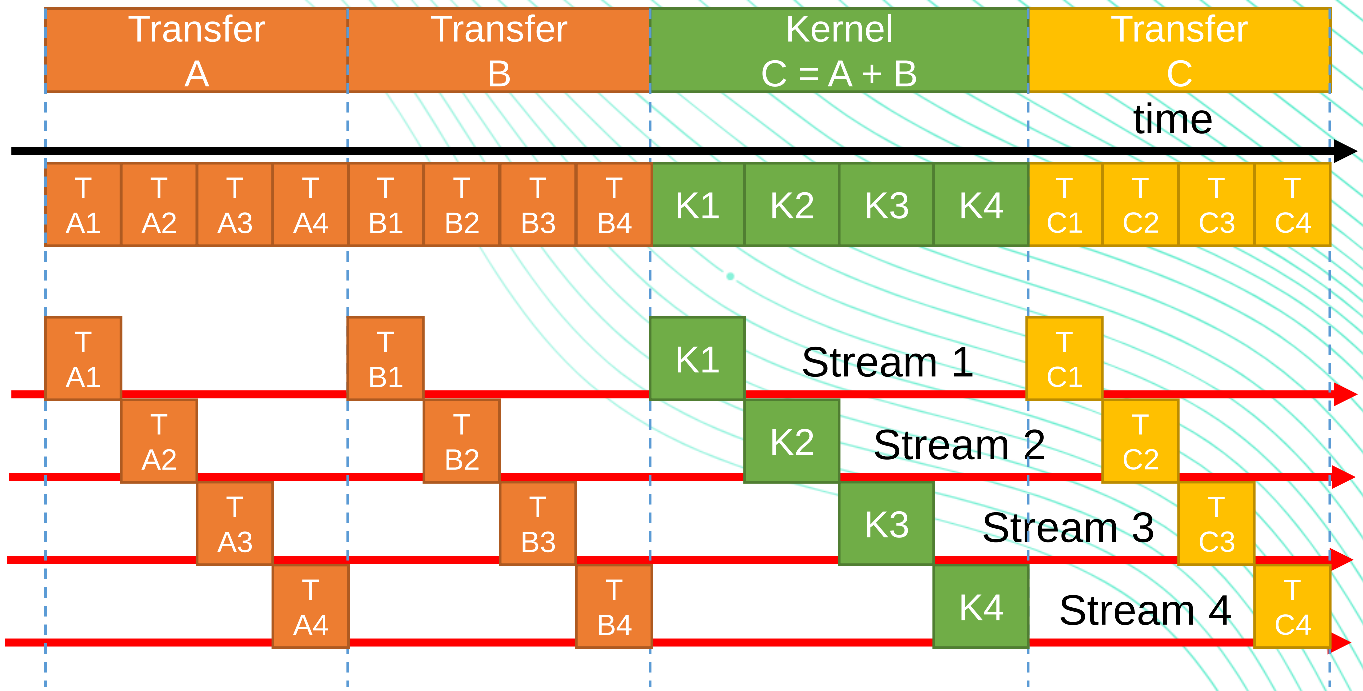
Serialized Data Transfer and Computation

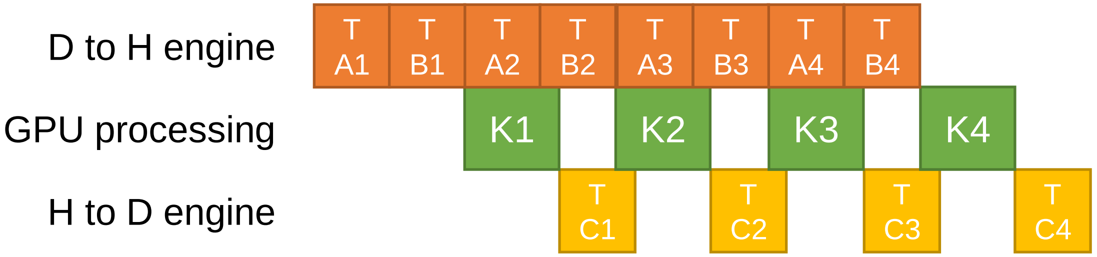
- So far, the way we use cudaMemcpy serializes data transfer and GPU computation for VecAddKernel()

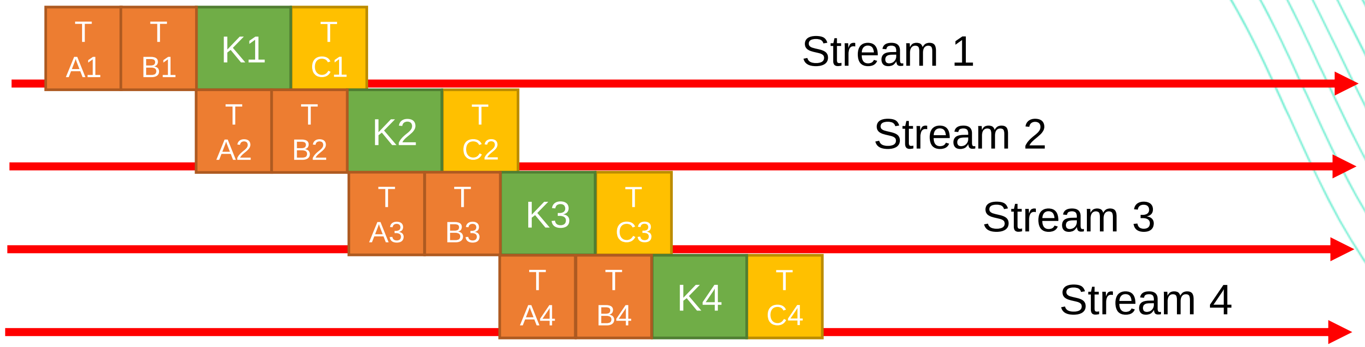
Ideal, Pipelined Timing

- Divide large vectors into segments
- Overlap transfer and compute of adjacent segments

Let CUDA devices overlap transfers and kernels execution







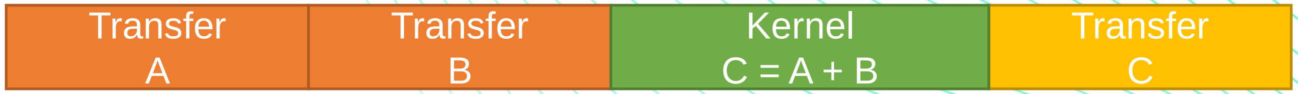
Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CPU-GPU Data Transfer using DMA

Serialized Data Transfer and Computation

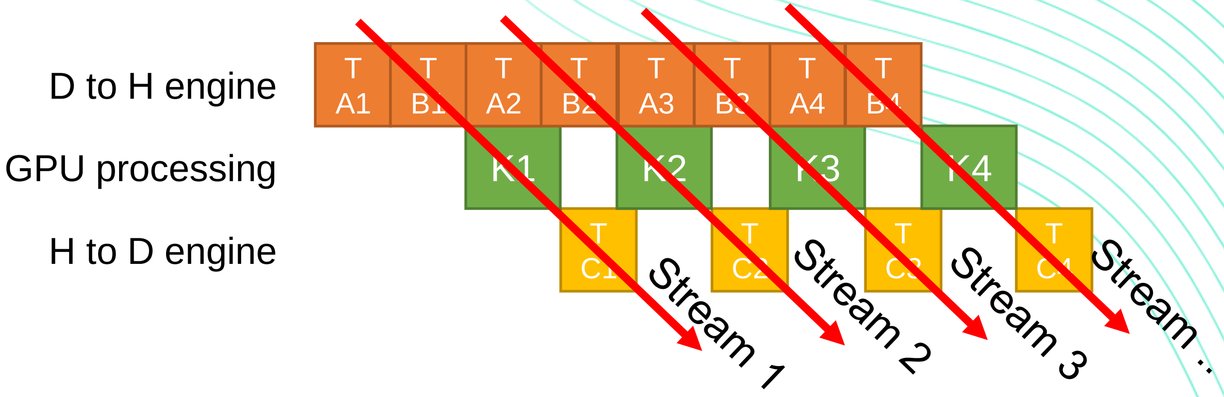
//non-streamed version

```
cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);
Kernel<<<b, t>>>(d_a, d_b, d_c, N);
cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);
```



//streamed version

```
// c    – number of pipeline phases
// ns   – total number of streams used
// size – size of input arrays
cudaStream_t stream[ns];
for (int i = 0; i < ns; ++i)
 cudaStreamCreate(&stream[i]);

for (int i = 0, i<c; i++){
 size_t off = (size/c)*i;
 cudaMemcpyAsync(d_a+off, h_a+off, size/c, cudaMemcpyHostToDevice, stream[i%ns]);
 cudaMemcpyAsync(d_b+off, h_b+off, size/c, cudaMemcpyHostToDevice, stream[i%ns]);
 Kernel<<<b/c, t, 0, stream[i%ns]>>>(d_a+off, d_b+off, d_c+off, N/c);
 cudaMemcpyAsync(h_c+off, d_c+off, size/c, cudaMemcpyDeviceToHost, stream[i%ns]);
}
```



Nvidia: https://www.olcf.ornl.gov/wp-content/uploads/2020/07/07_Concurrency.pdf

EPICURE

Unlocking European-level HPC Support

Hands On:
Asynchronous execution, streams, pipelining

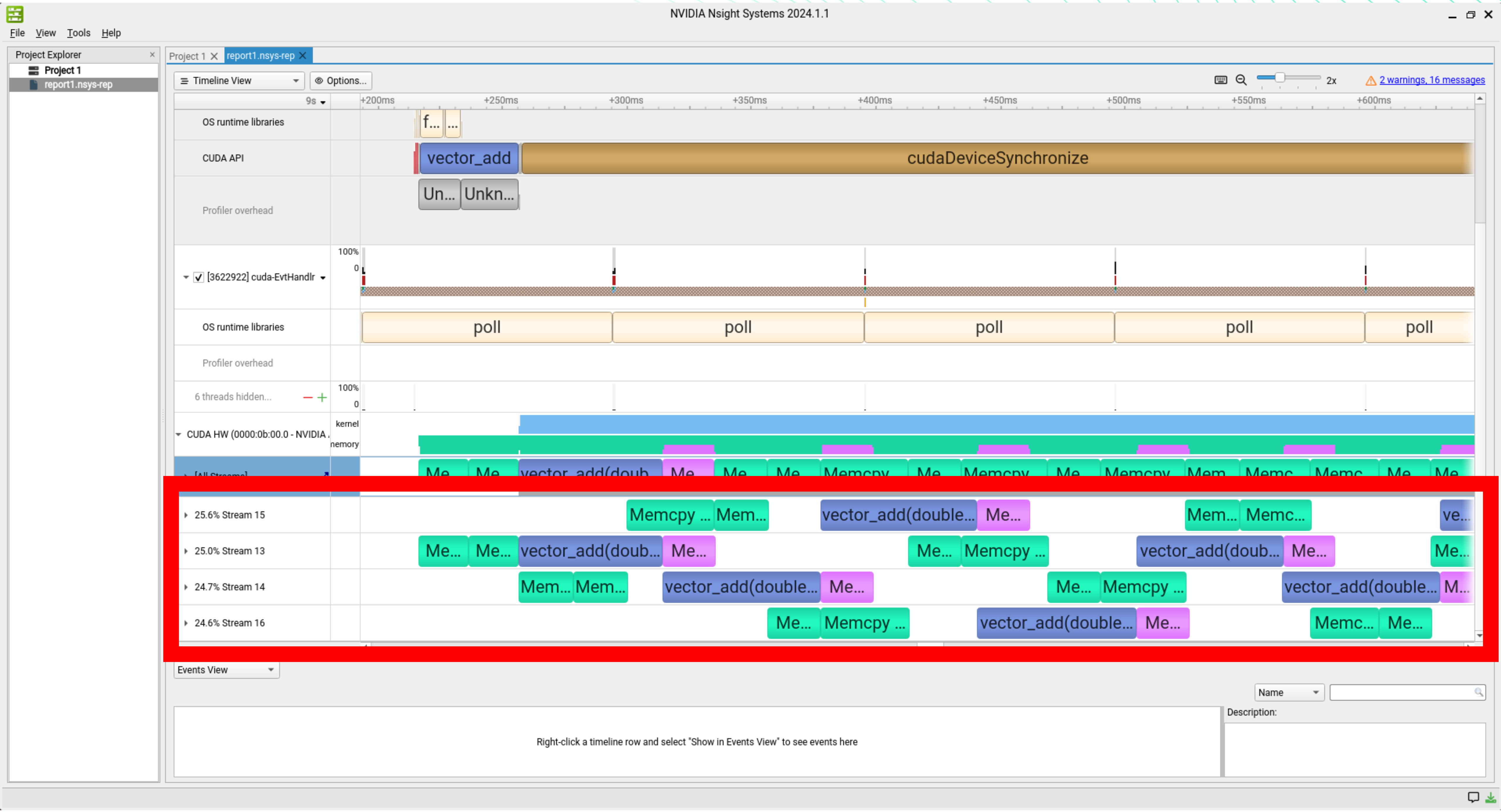
Hands-on: asynchronous execution

- `tasks/vector_add_stream`
- Copy the solution of the vector add pinned task
  - E.g. `cp ../../solution/vector_add_pinned/vector_add_pinned.cu .`
  - It just allocates the CPU array differently
- Modify the code such that GPU operations run asynchronously
  - Use cudaMemcpyAsync(…) for copy
  - Kernel is already asynchronous
- Create a cuda stream and use it to launch all the GPU operations
  - cudaStreamCreate(), cudaStreamDestroy()
  - Additional stream parameter in cudaMemcpyAsync() and in <<< >>>
- Increase the `count` and add some `printf` to observe the asychronicity

Hands-on: pipelining

- `tasks/vector_add_overlap`
- More efficient vector add – overlap computation with memory transfer
- Copy the solution of the previous task utilizing streams
- Modify the kernel so that the computation takes longer --->
- Use the pipelining described before
  - Create an array of 4 streams
  - Split the vector into 20 sections (not really, just pointer arithmetic)
  - For each (i-th) section: copyin, compute, copyout in (i%4)-th stream
- Measure the time using e.g. cuda events (or std::chrono, or omp_get_wtime())
  - Measure only the copy+compute time, not malloc etc.
- Vary the number of sections and streams and observe the timing differences

```
if(idx < count)
{
    for(int r = 0; r < 200; r++)
    {
        c[idx] = a[idx] + b[idx];
    }
}
```



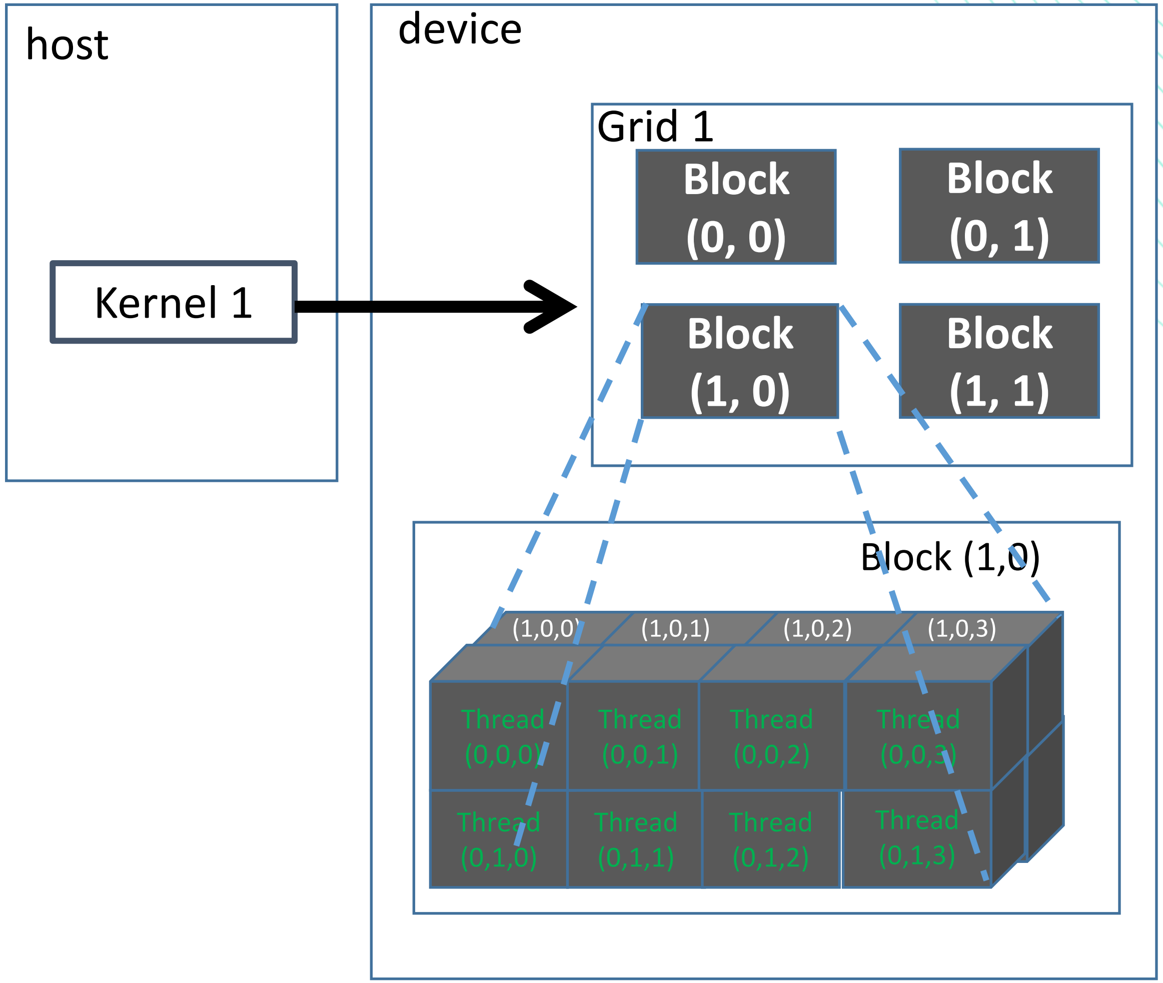
Right-click a timeline row and select "Show in Events View" to see events here

EPICURE

Unlocking European-level HPC Support

Multi-Dimensional Grids

CUDA programming
Multi-Dimensional Grid



Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

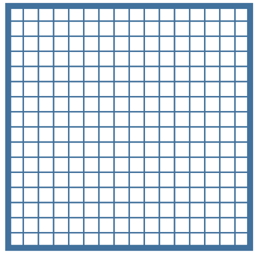
CUDA programming
Processing a Picture with a 2D Grid

Work distribution

- image will be addressed in 2D blocks of size
    - 16x16 threads
- some threads, highlighted in orange, will be idle

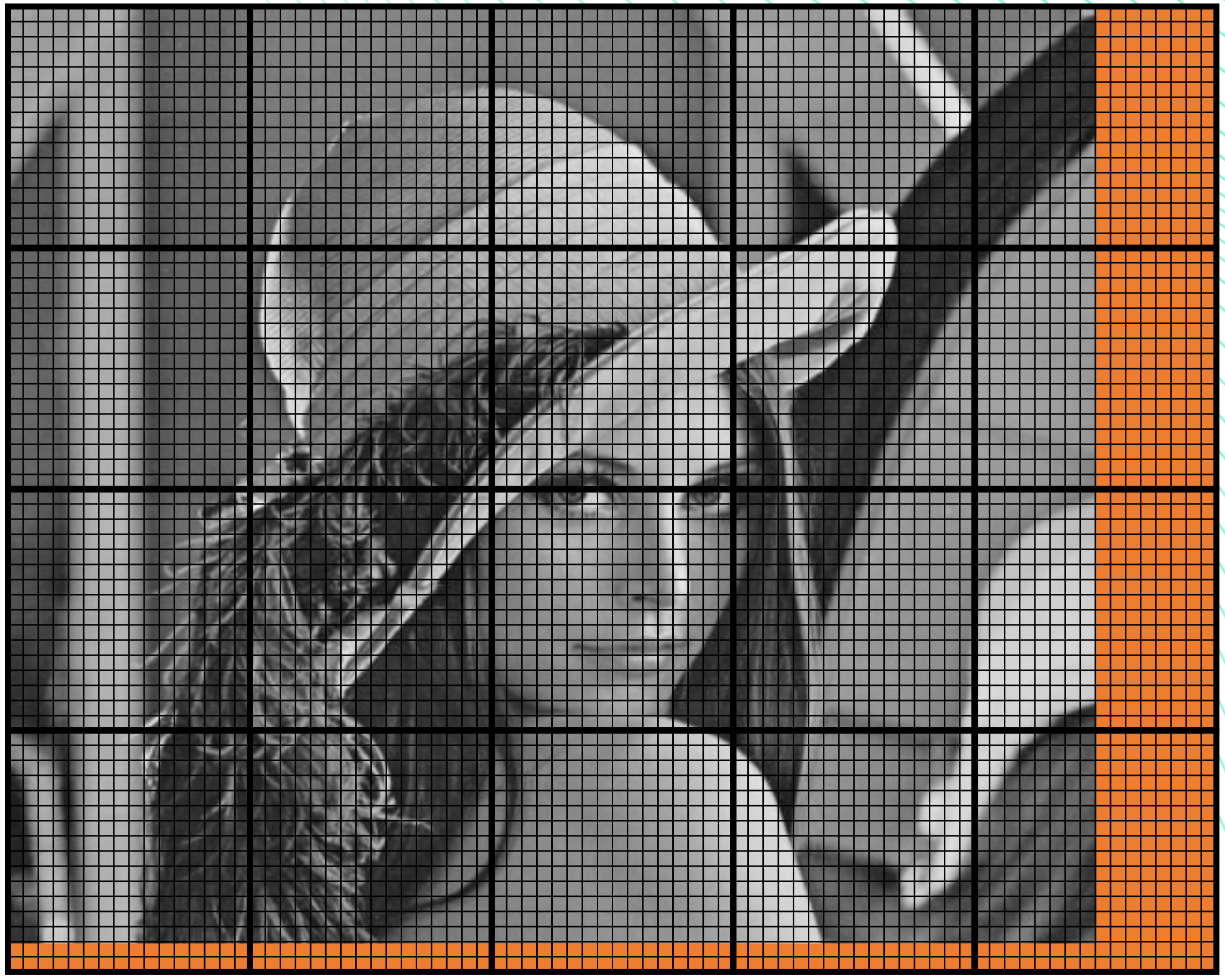
Control flow divergence

- not all threads in a Block will follow the same control flow path



1 block: 16×16 threads per block

62×76 picture



Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

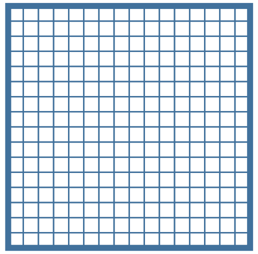
CUDA programming
Processing a Picture with a 2D Grid

Work distribution

- image will be addressed in 2D blocks of size
  - 16x16 threads
- some threads, highlighted in orange, will be idle

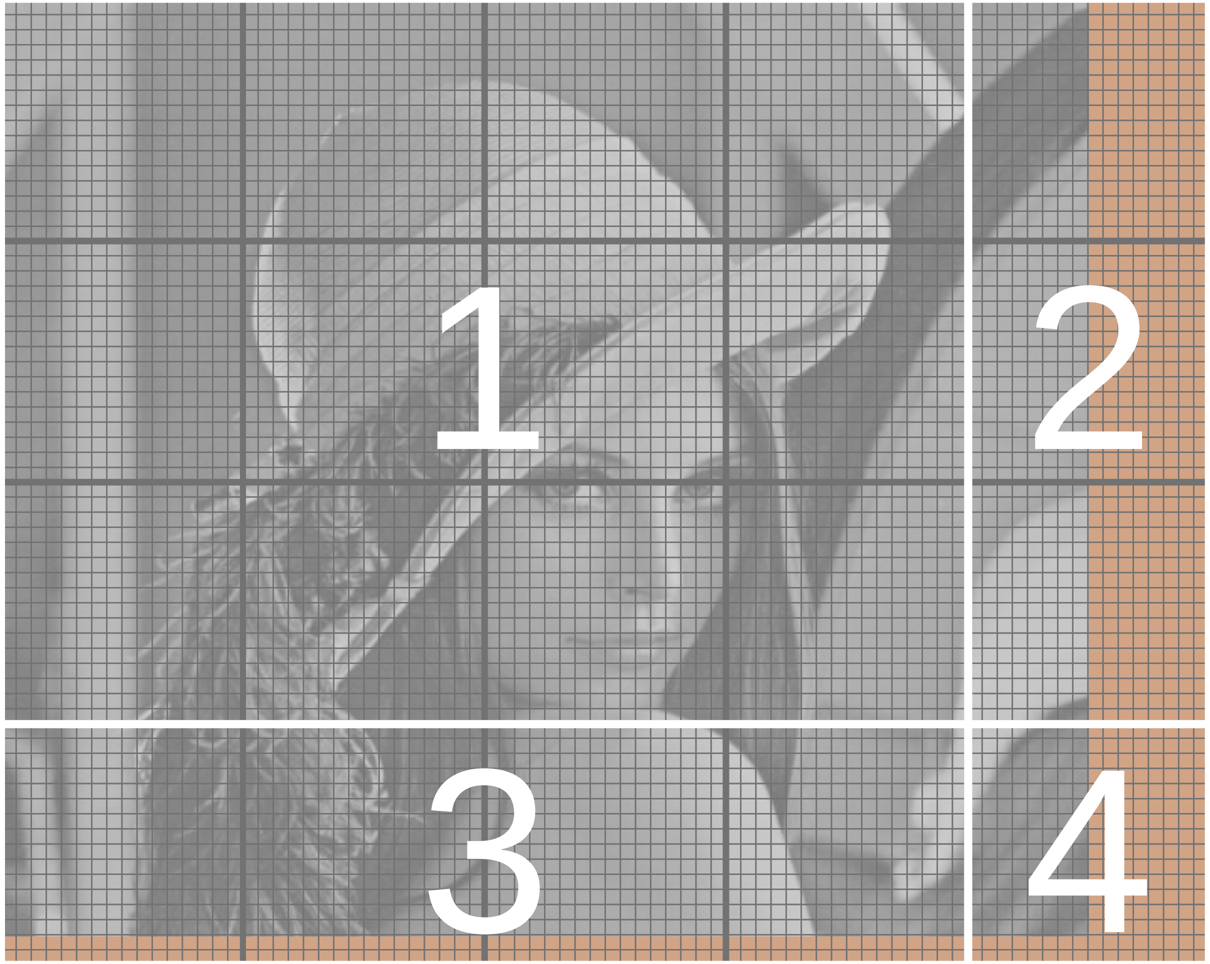
Control flow divergence

- not all threads in a block will follow the same control flow path
- 4 different paths in this case



1 block: 16×16 threads per block

62×76 picture



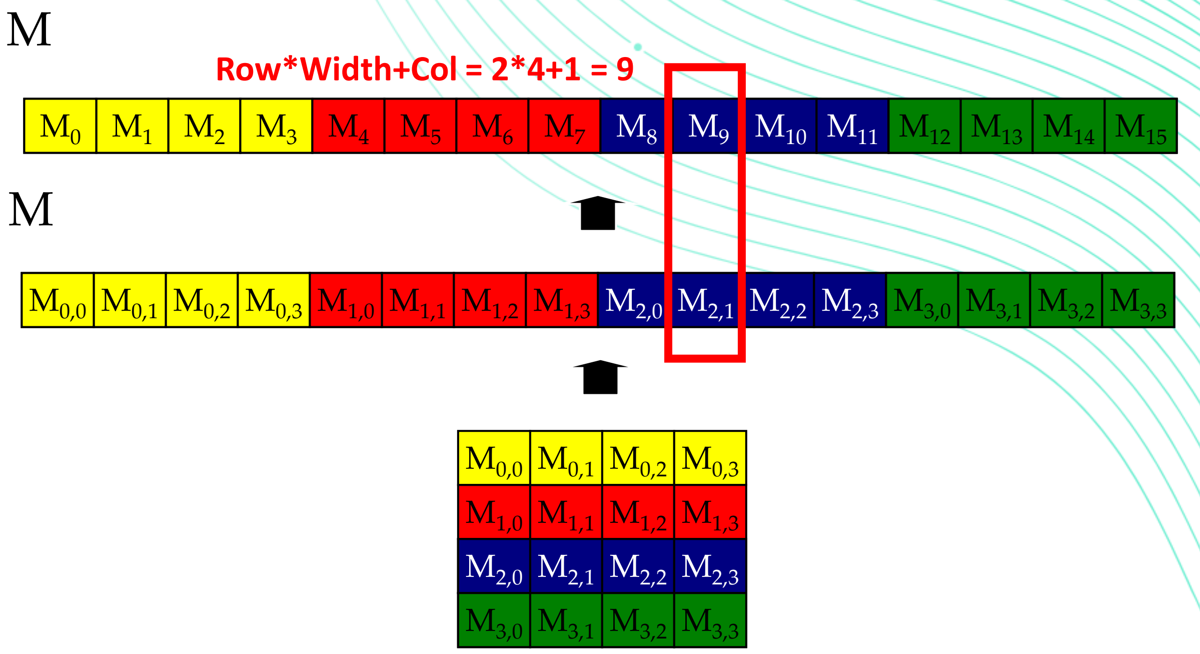
Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CUDA programming
Processing a Picture with a 2D Grid

Kernel

```
__global__ void PictureKernel(float* d_Pin,
                              float* d_Pout,
                              int height,
                              int width)

{
 // Calculate the row # of
 // the d_Pin and d_Pout element
 int Row = blockIdx.y*blockDim.y + threadIdx.y;

 // Calculate the column # of
 // the d_Pin and d_Pout element
 int Col = blockIdx.x*blockDim.x + threadIdx.x;

 // each thread computes one
 // element of d_Pout if in range
 if ((Row < height) && (Col < width)) {
  d_Pout[Row*width+Col] = 2.0*d_Pin[Row*width+Col];
 }
}
```

Row-Major Layout in C/C++



M
Row*Width+Col = 2*4+1 = 9
M_0 M_1 M_2 M_3 M_4 M_5 M_6 M_7 M_8 M_9 M_{10} M_{11} M_{12} M_{13} M_{14} M_{15}
M
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$ $M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$ $M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$ $M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CUDA programming
Processing a Picture with a 2D Grid

Kernel

```
__global__ void PictureKernel(float* d_Pin,
                            float* d_Pout,
                            int height,
                            int width)
{
 // Calculate the row # of
 // the d_Pin and d_Pout element
 int Row = blockIdx.y*blockDim.y + threadIdx.y;

 // Calculate the column # of
 // the d_Pin and d_Pout element
 int Col = blockIdx.x*blockDim.x + threadIdx.x;

 // each thread computes one
 // element of d_Pout if in range
 if ((Row < height) && (Col < width)) {
  d_Pout[Row*width+Col] = 2.0*d_Pin[Row*width+Col];
 }
}
```

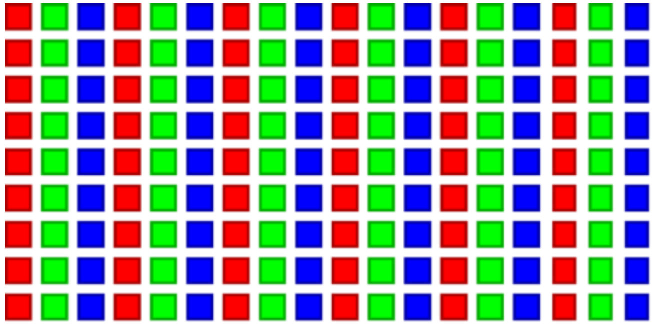
Host Code for Launching 2D kernel

- assume that the picture is **m × n**, (height × width)
- m pixels in y dimension and n pixels in x dimension
- input d_Pin has been allocated on and copied to device
- output d_Pout has been allocated on device

```
dim3 DimGrid((n-1)/16 + 1, (m-1)/16+1, 1);

dim3 DimBlock(16, 16, 1);

PictureKernel<<<DimGrid,DimBlock>>>(d_Pin, d_Pout, m, n);
```

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CUDA programming
Converting color image to grayscale



RGB color image

- 3 values per pix
  - r - red
  - g - green
  - b - blue

Grayscale image

- only intesity

RGB Kernel:

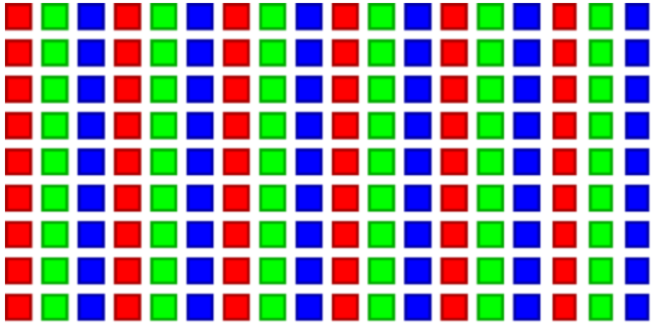
```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__ void colorConvert(unsigned char * grayImage,
                             unsigned char * rgbImage,
                        int width, int height) {

 int col = threadIdx.x + blockIdx.x * blockDim.x;
 int row = threadIdx.y + blockIdx.y * blockDim.y;

 if (col < width && row < height) {
  // get 1D coordinate for the grayscale image
  int grayOffset = row*width + col;











 }
}
```

Host code for launching the kernel is the same as in previou s slide.

grayPixel[I,J] = 0.21*r + 0.71*g + 0.07*b

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CUDA programming
Converting color image to grayscale



RGB color image

- 3 values per pix
  - r - red
  - g - green
  - b - blue

Grayscale image

- only intesity

RGB Kernel:

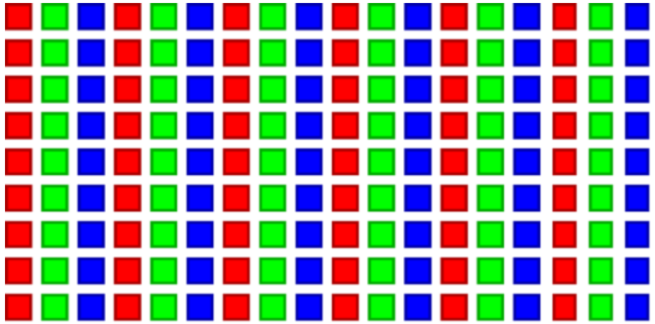
```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__ void colorConvert(unsigned char * grayImage,
             unsigned char * rgbImage,
       int width, int height) {

  int col = threadIdx.x + blockIdx.x * blockDim.x;
  int row = threadIdx.y + blockIdx.y * blockDim.y;

  if (col < width && row < height) {
   // get 1D coordinate for the grayscale image
   int grayOffset = row*width + col;
   // one can think of the RGB image having
   // CHANNEL times columns than the gray scale image
   int  rgbOffset = grayOffset*CHANNELS;
   unsigned char r = rgbImage[rgbOffset + 0]; // red value for pix
   unsigned char g = rgbImage[rgbOffset + 1]; // green value for pix
   unsigned char b = rgbImage[rgbOffset + 2]; // blue value for pix
```

grayPixel[I,J] = 0.21*r + 0.71*g + 0.07*b

Host code for launching the kernel is the same as in previou s slide.

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CUDA programming
Converting color image to grayscale



RGB color image

- 3 values per pix
  - r - red
  - g - green
  - b - blue

Grayscale image

- only intesity

grayPixel[I,J] = 0.21*r + 0.71*g + 0.07*b

RGB Kernel:

```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__ void colorConvert(unsigned char * grayImage,
                             unsigned char * rgbImage,
                   int width, int height) {

 int col = threadIdx.x + blockIdx.x * blockDim.x;
 int row = threadIdx.y + blockIdx.y * blockDim.y;

 if (col < width && row < height) {
  // get 1D coordinate for the grayscale image
  int grayOffset = row*width + col;
  // one can think of the RGB image having
  // CHANNEL times columns than the gray scale image
  int  rgbOffset = grayOffset*CHANNELS;
  unsigned char r = rgbImage[rgbOffset + 0]; // red value for pix
  unsigned char g = rgbImage[rgbOffset + 1]; // green value for pix
  unsigned char b = rgbImage[rgbOffset + 2]; // blue value for pix
  // perform the rescaling and store it
  // We multiply by floating point constants
  grayImage[grayOffset] = (unsigned char)(0.21f*r + 0.71f*g + 0.07f*b);
 }
}
```

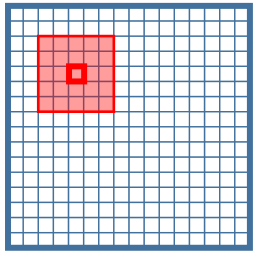
Host code for launching the kernel is the same as in previou s slide.

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

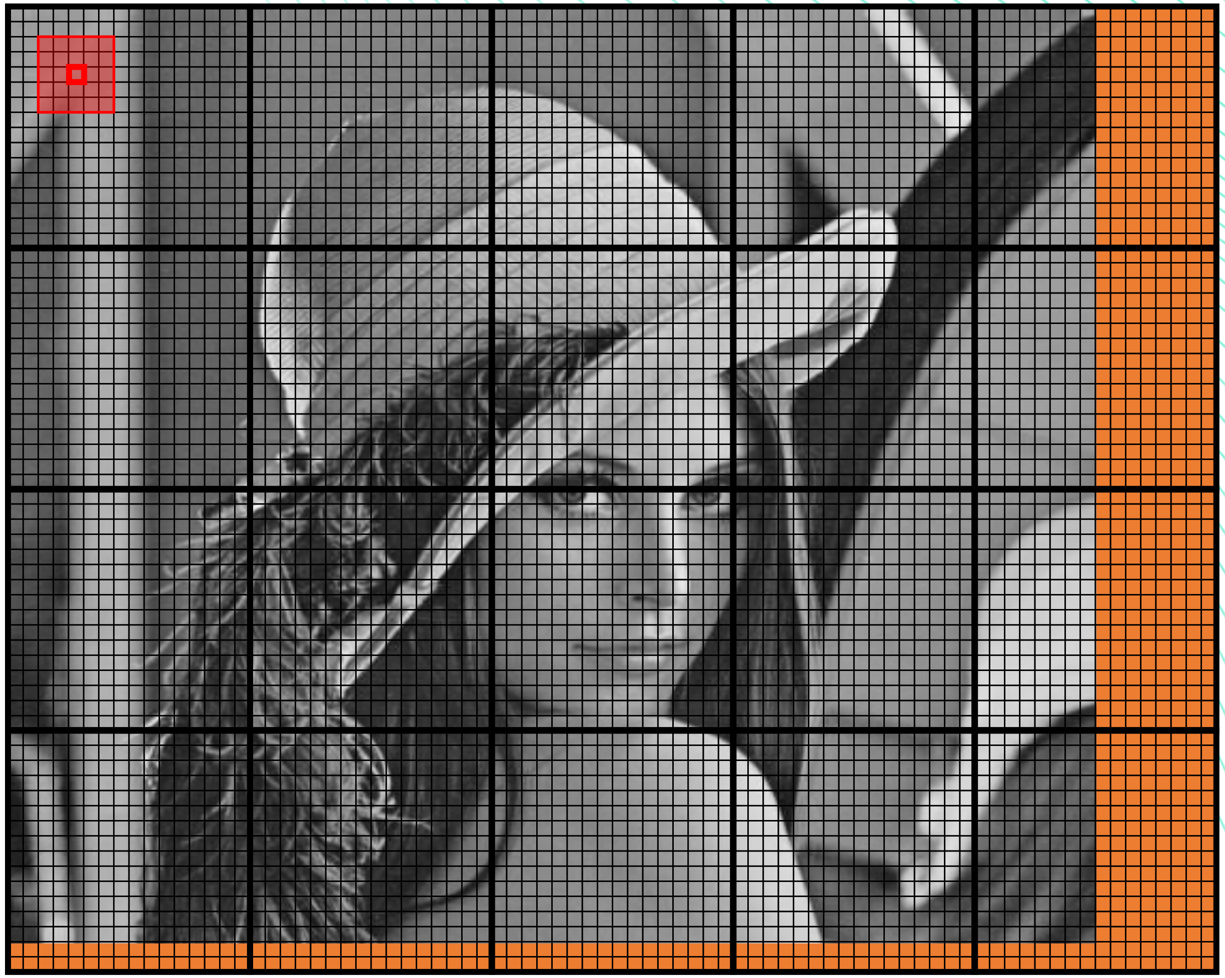
CUDA programming
Image Blur

Blur Filter

- calculates average value inside the mask
    - `BLUR_SIZE` **value**



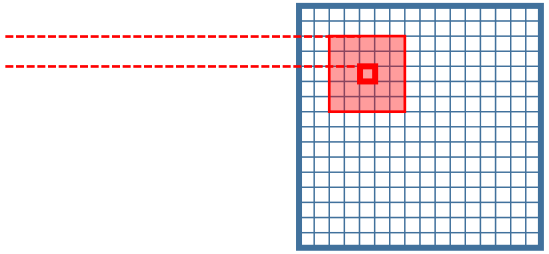
1 block: 16×16 threads per block



BlurPixel[I,J] = Average value of all pixel in a mask

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

Hands-on
Image Blur


BLUR_SIZE=2

1 block:
16×16
threads
per block

BlurPixel[I,J] = Average value of all pixel in a mask

```
// we have 1 channel, therefore a grayscale image
// The input image is encoded as unsigned characters [0, 255]
__global__ void BlurKernel(unsigned char * inImage,
                          unsigned char * outImage,
                     int width, int height) {
 int col = threadIdx.x + blockIdx.x * blockDim.x;
 int row = threadIdx.y + blockIdx.y * blockDim.y;
 if (col < width && row < height) {
  int pixVal = 0; int pixels = 0;
  // Get the average of the surrounding 2xBLUR_SIZE x 2xBLUR_SIZE box
  for(int blurRow =                                     ) {
   for(int blurCol =                                     ) {
    int curRow =               ;
    int curCol =               ;

    // Verify we have a valid image pixel
    if(curRow >    && curRow <   && curCol >    && curCol <  ) {
     pixVal =
     pixels =           ; // Total number of accumulated pixels
    }
   }
  }
  // Write our new pixel value out
  outImage[                  ] = (unsigned char)(pixVal        ); }
}
```

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

EPICURE

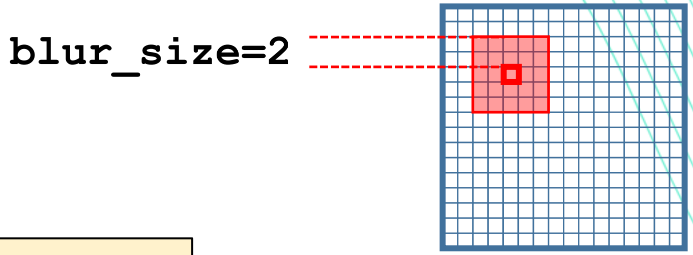
Unlocking European-level HPC Support

Hands-on Image Blur

Hands-on: image blur

- `tasks/image_blur`
- Complete the TODO tasks
    - Allocate managed memory for original (input) and blurred (output) image
        - height*width, unsigned char
    - Implement and launch the kernel
        - Use 2D blocks and grid, and the dim3 type
    - Here, blur_size should be a kernel parameter instead of a global macro

    - There is no actual image
        - Just a pattern that is easy to check for correctness



Correct output:

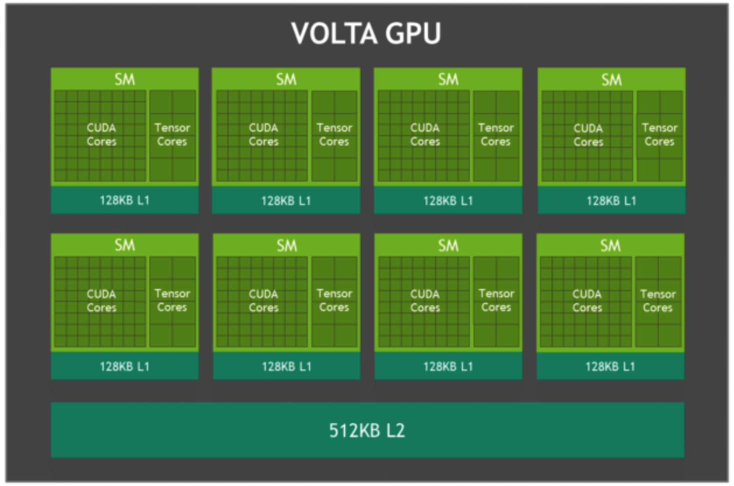
```
Everything seems OK
```

Thread Execution

Thread Execution

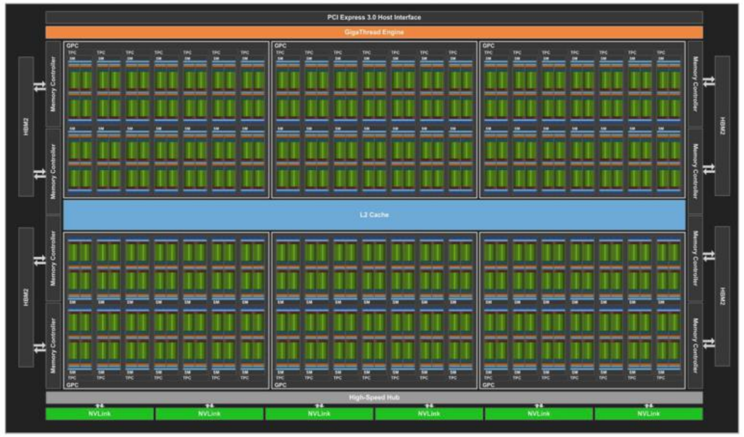
Transparent scaling of GPU kernels

- Kernel execution is broken in Grid of Blocks
  - blocks can be executed in any order relative to others
  - hardware is free to assign blocks to any Streaming Multiprocessor (SM) at any time
  - **a kernel scales to any number of parallel processors**
- **this property ensures correct execution on GPUs with**
  - **different number of Streaming Multiprocessors (different performance, different model of GPU accelerators (A100, A40, ...)**
  - **different GPU architectures (Pascal, Volta, Ampere, …)**



NVIDIA Jetson AGX Xavier

- ARM based embedded single board computer with on-chip GPU
- GPU with 8 SMs



NVIDIA V100

- HPC accelerator
- GPU with 80 SMs

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

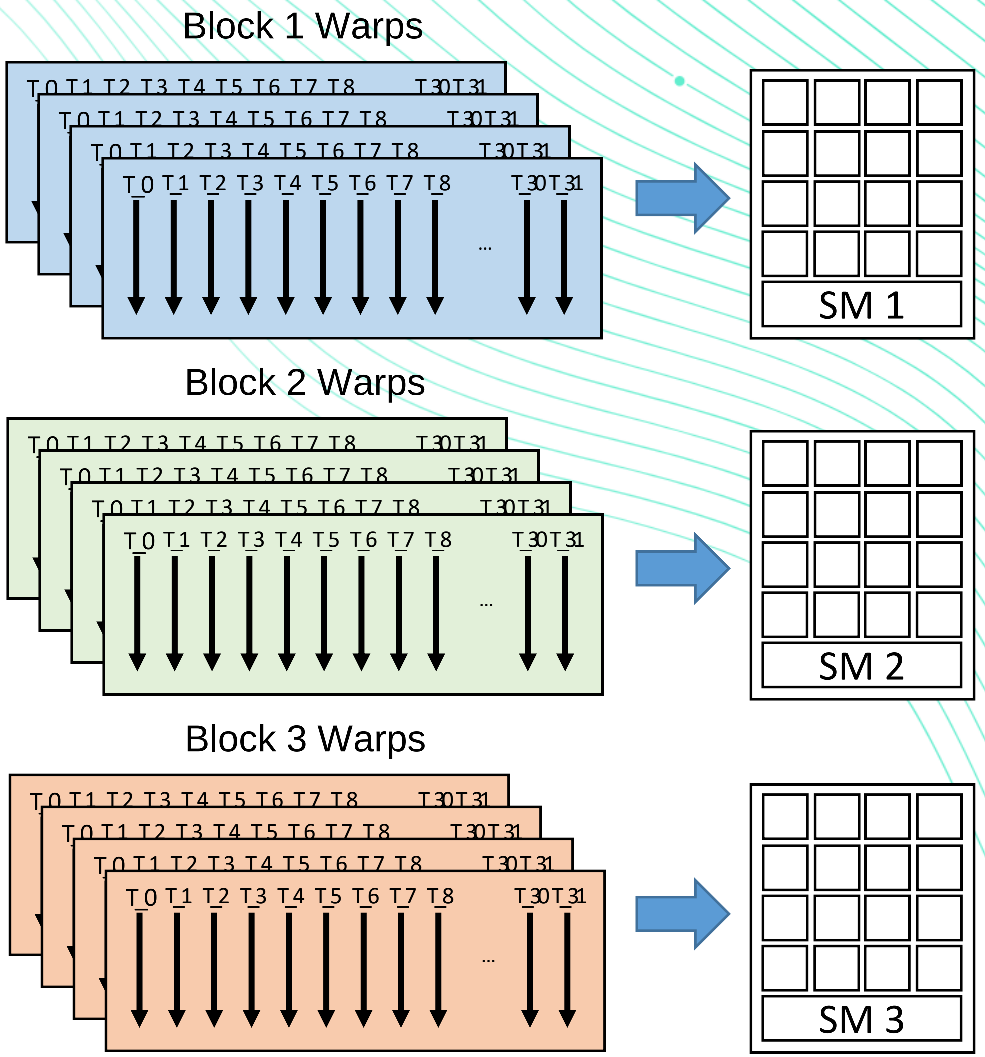
Thread Execution and Warps

Thread Execution

- blocks are assigned to Streaming Multiprocessors (SM)
  - up to 32 blocks can be assigned to one SM as resources allow
  - Ampere generation SM can take up to 2048 threads
    - could be 256 (threads/block) * 8 blocks
    - or 512 (threads/block) * 4 blocks, etc.
- SM maintains thread/block idx #s
- SM manages/schedules thread execution

Warps as Scheduling Units

- each Block is divided and executed as 32-thread Warps
  - an implementation decision, not part of the CUDA programming model
- warps are scheduling units in SM
- threads in a warp execute in SIMD fashion
- future GPUs may have different number of threads in each warp
  - for instance, AMD GPUs have warp size 64 threads



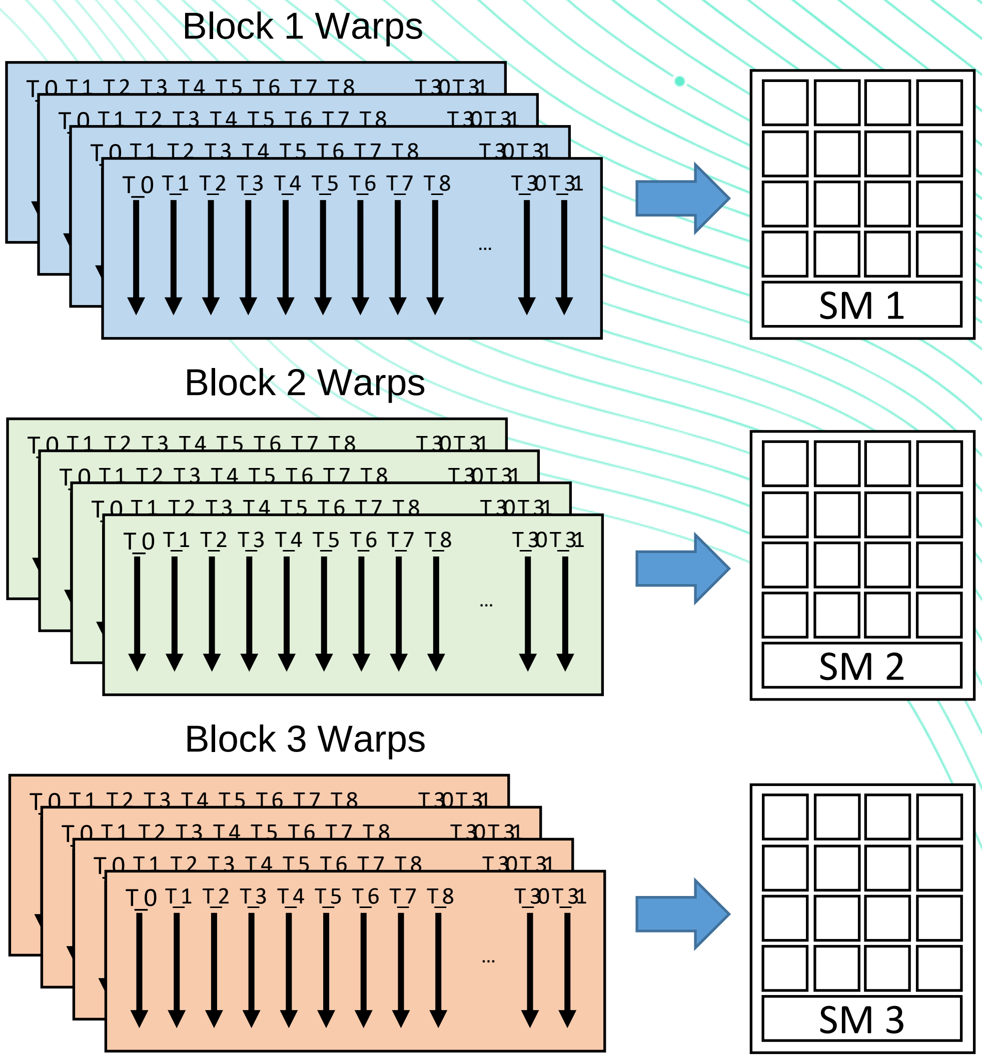
Block 1 Warps
T0 T1 T2 T3 T4 T5 T6 T7 T8 … T30 T31
SM 1
Block 2 Warps
T0 T1 T2 T3 T4 T5 T6 T7 T8 … T30 T31
SM 2
Block 3 Warps
T0 T1 T2 T3 T4 T5 T6 T7 T8 … T30 T31
SM 3

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

Thread Execution and Warps

Thread Execution cont.

- SM implements zero-overhead warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution based on a prioritized scheduling policy
  - All threads in a warp execute the same instruction when selected

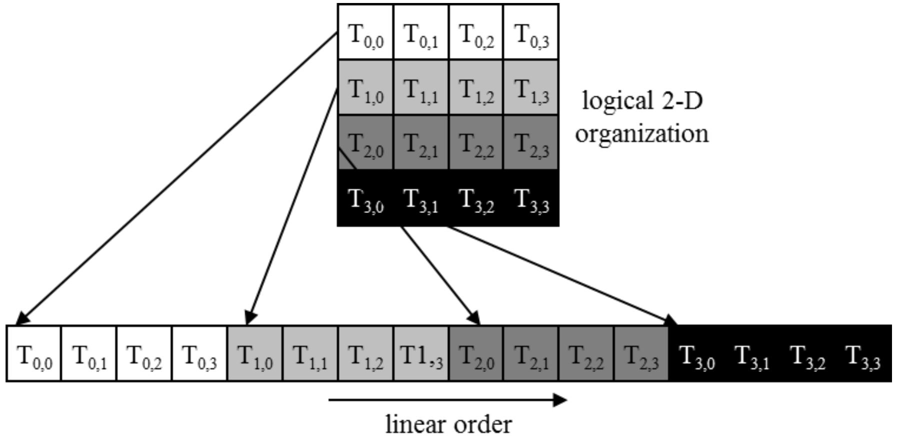

Block 1 Warps
T 0 T 1 T 2 T 3 T 4 T 5 T 6 T 7 T 8 … T 30 T 31
SM 1
Block 2 Warps
T 0 T 1 T 2 T 3 T 4 T 5 T 6 T 7 T 8 … T 30 T 31
SM 2
Block 3 Warps
T 0 T 1 T 2 T 3 T 4 T 5 T 6 T 7 T 8 … T 30 T 31
SM 3

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

Thread Execution and Warps

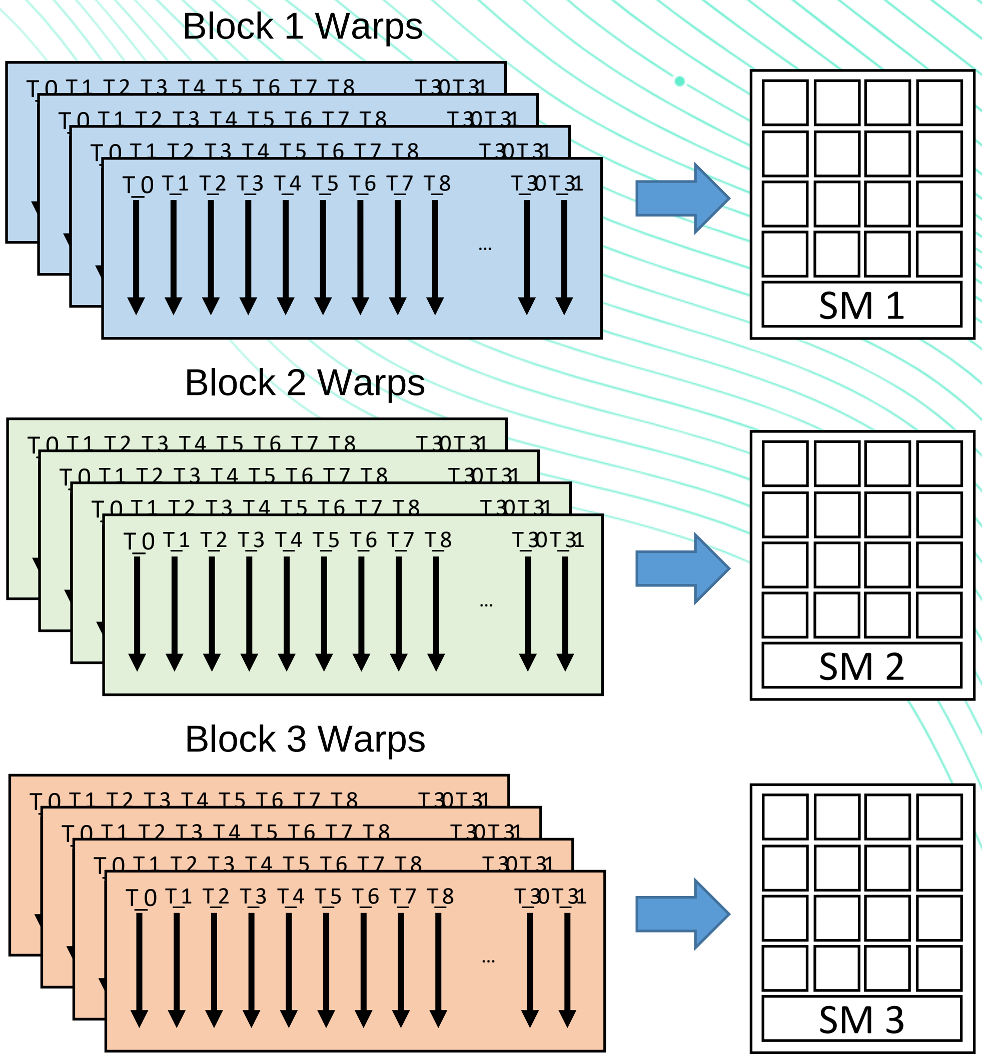
Warps in Multi-dimensional Thread Blocks

- The thread blocks are first linearized into 1D in row major order
- In x-dimension first, y-dimension next, and z-dimension last



- Linearized thread blocks are partitioned in warps
  - Thread indices within a warp are consecutive and increasing
  - Warp 0 starts with Thread 0

- **DO NOT rely on any ordering within or between warps**
  - If there are any dependencies between threads, you must `__syncthreads()` to get correct results (more later)



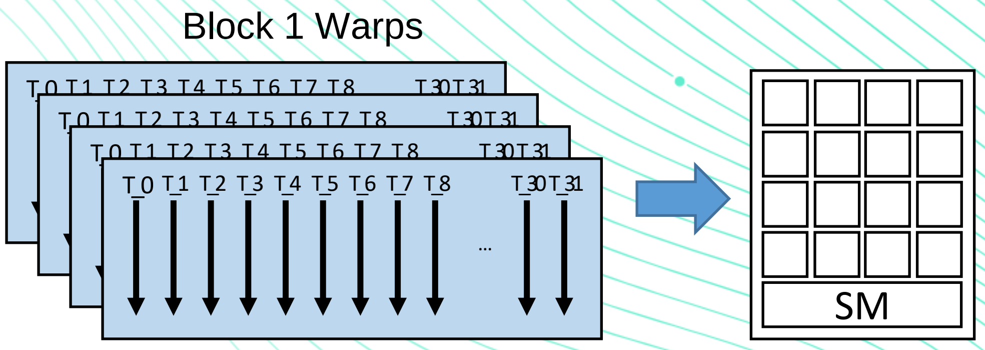
Block 1 Warps
T 0 T 1 T 2 T 3 T 4 T 5 T 6 T 7 T 8 ... T 30 T 31
SM 1
Block 2 Warps
T 0 T 1 T 2 T 3 T 4 T 5 T 6 T 7 T 8 ... T 30 T 31
SM 2
Block 3 Warps
T 0 T 1 T 2 T 3 T 4 T 5 T 6 T 7 T 8 ... T 30 T 31
SM 3

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

Thread Execution and Warps

SIMD Execution Among Threads in a Warp

- All threads in a warp must execute the same instruction at any point in time
- This works efficiently if all threads follow the same control flow path
  - All if-then-else statements make the same decision
  - All loops iterate the same number of times



SMs are SIMD Processors

Example of a SIMD code:

```
__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;

  C[i] = A[i] + B[i];
}
```

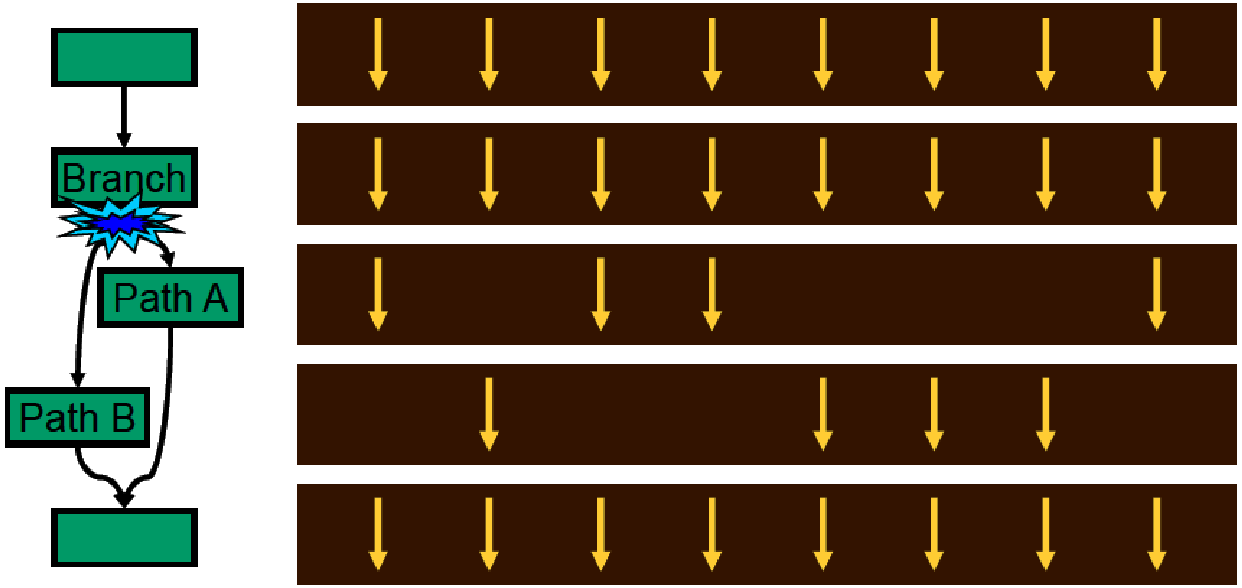
Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

Thread Execution and Warps

Control Divergence

- control divergence occurs when threads in a warp take different control flow paths by making different control decisions
    - some take the then-path and others take the else-path of an if-statement
    - some threads take different number of loop iterations than others
- The execution of threads taking different paths are serialized in current GPUs
    - the control paths taken by the threads in a warp are traversed one at a time until there is no more
    - during the execution of each path, all threads taking that path will be executed in parallel

```
if(foo(threadIdx.x))
   {
           do_A();
   }
   else
   {
           do_B();
   }
```

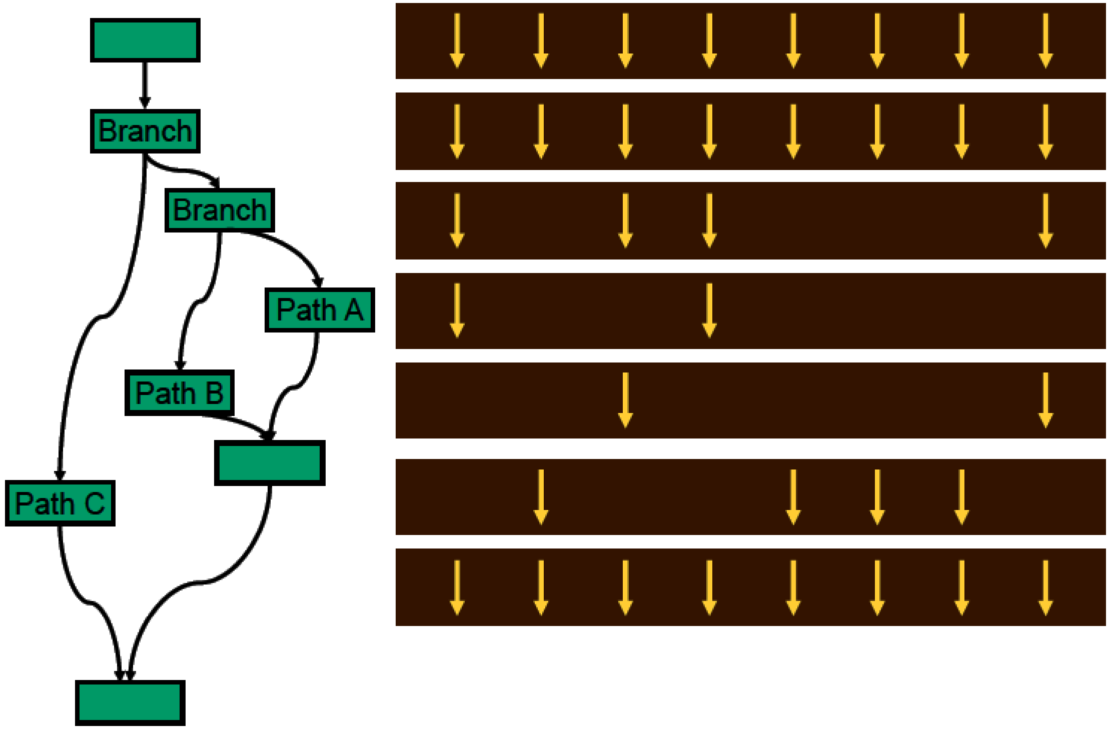


Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

Thread Execution and Warps

Control Divergence

The number of different paths can be large when considering nested control flow statements.

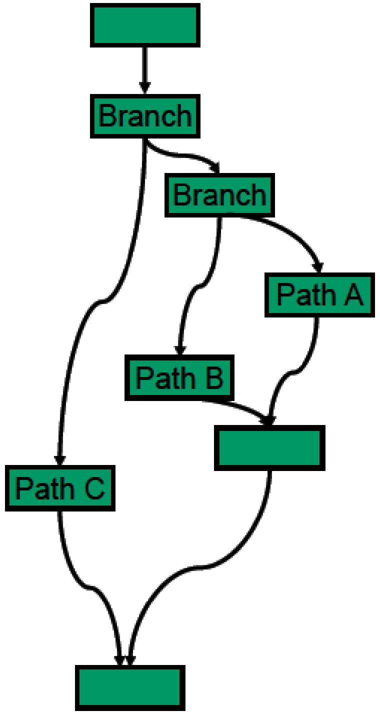


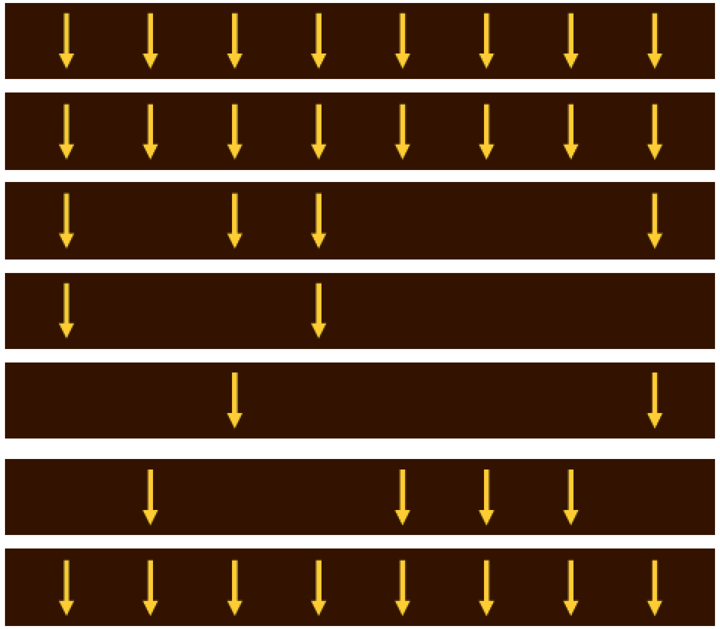
Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

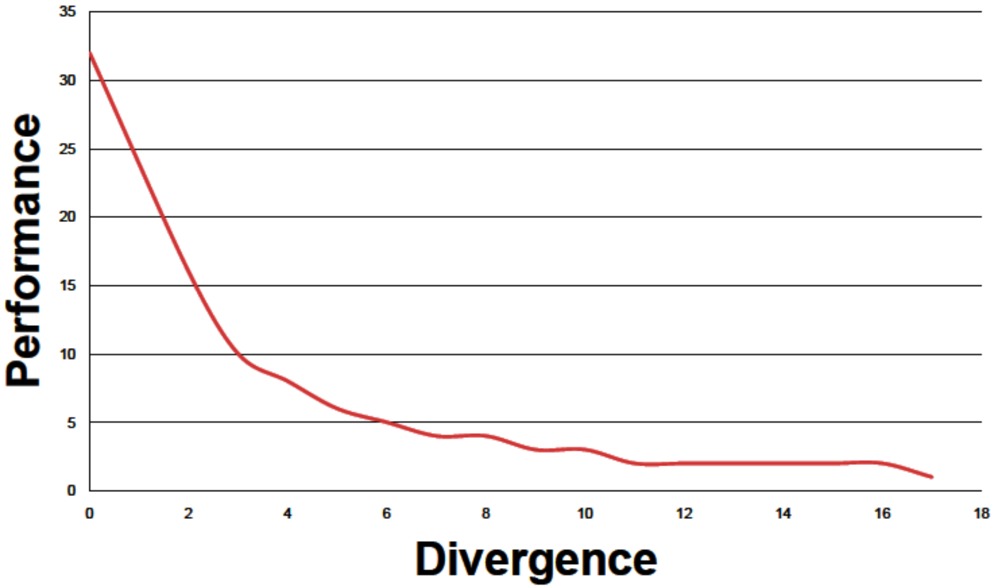
Thread Execution and Warps

Control Divergence

The number of different paths can be large when considering nested control flow statements.







The control diverges is problem only among threads within a warp.

The control divergence among warps is perfectly fine as long as all threads within a warp execute the same instruction.

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

Thread Execution and Warps

Divergence can arise when branch or loop condition is a function of thread indices

```
__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;

  if(i<n) C[i] = A[i] + B[i];
}
```

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

EPICURE

Unlocking European-level HPC Support

End of Day 1