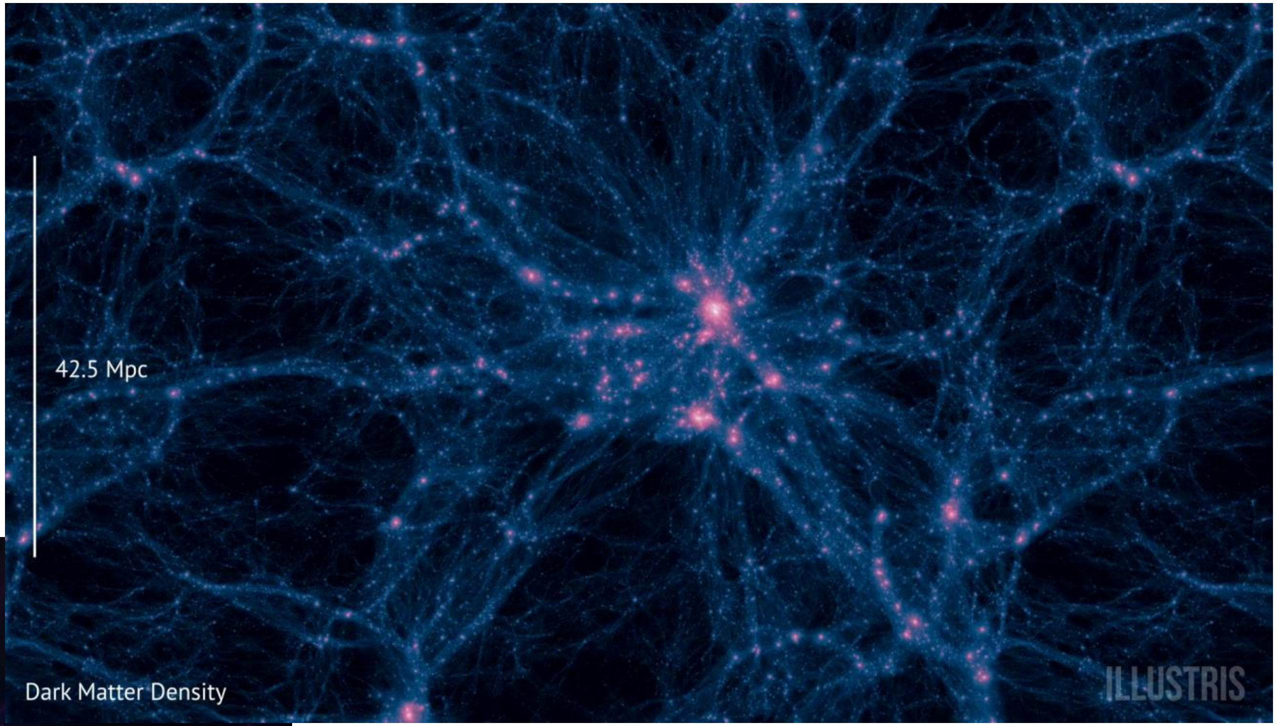
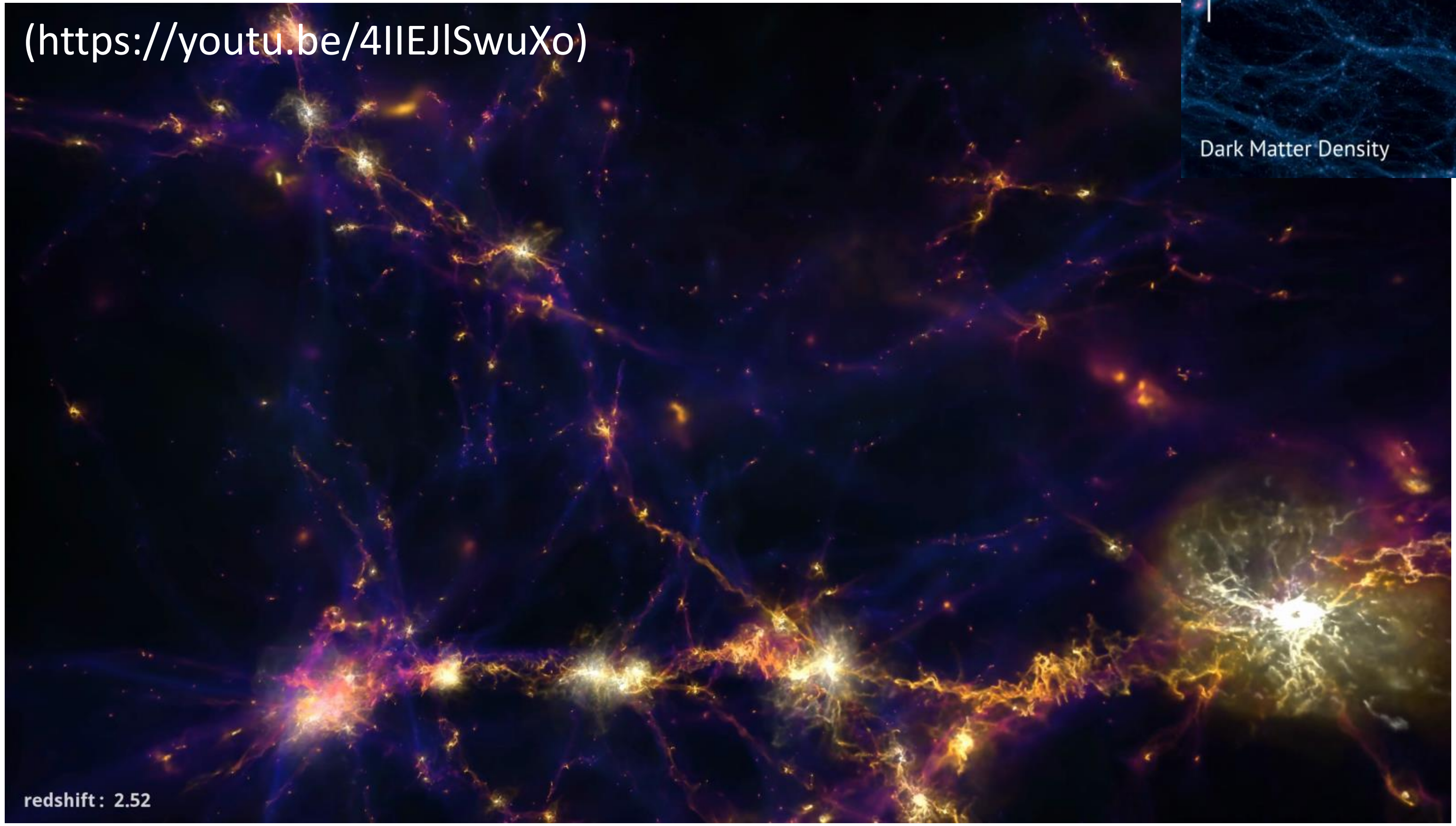
The architecture and programming model of graphics processing units

Christoph Riesinger | Rechnerarchitekturen | June 09, 2021

Motivation



(https://youtu.be/4IIEJlSwuXo)



(http://www.tng-project.org/media/)

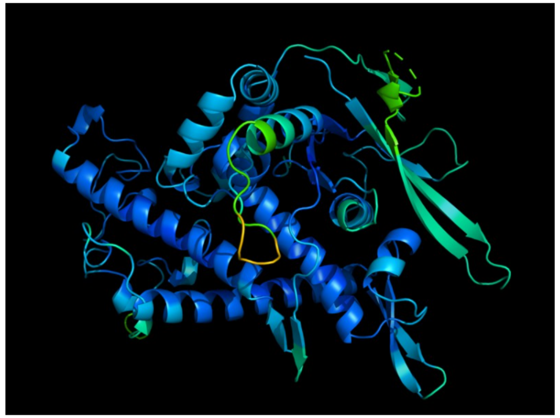
Motivation

NEWS · 30 NOVEMBER 2020

'It will change everything': DeepMind's AI makes gigantic leap in solving protein structures

Google's deep-learning program for determining the 3D shapes of proteins stands to transform biology, say scientists.

Ewen Callaway



A protein's function is determined by its 3D shape. Credit: DeepMind

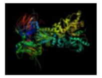
An artificial intelligence (AI) network developed by Google AI offshoot DeepMind has made a gargantuan leap in solving one of biology's grandest challenges — determining a protein's 3D shape from its amino-acid sequence.

DeepMind's program, called AlphaFold, outperformed around 100 other teams in a biennial protein-structure prediction challenge called CASP, short for Critical Assessment of Structure Prediction. The results were announced on 30 November, at the start of the conference — held virtually this year — that takes stock of the exercise.

"This is a big deal," says John Moult, a computational biologist at the University of Maryland in College Park, who co-founded CASP in 1994 to improve computational methods for accurately predicting protein structures. "In some sense the problem is solved."

Related Articles

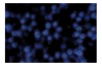
AI protein-folding algorithms solve structures faster than ever



The revolution will not be crystallized: a new method sweeps through structural biology

The computational protein designers

Revolutionary microscopy technique sees individual atoms for first time



Subjects

Computational biology and bioinformatics
Structural biology
Drug discovery

Sign up to Nature

(https://www.nature.com/articles/d41586-020-03348-4)

Motivation

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442,010.0	537,212.0	29,899
2	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
3	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
4	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
5	**Selene** - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband, Nvidia NVIDIA Corporation United States	555,520	63,460.0	79,215.0	2,646
6	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000, NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482
7	**JUWELS Booster Module** - Bull Sequana XH2000 , AMD EPYC 7402 24C 2.8GHz, NVIDIA A100, Mellanox HDR InfiniBand/ParTec ParaStation ClusterSuite, Atos Forschungszentrum Juelich (FZJ) Germany	449,280	44,120.0	70,980.0	1,764
8	**HPC5** - PowerEdge C4140, Xeon Gold 6252 24C 2.1GHz, NVIDIA Tesla V100, Mellanox HDR Infiniband, Dell EMC Eni S.p.A. Italy	669,760	35,450.0	51,720.8	2,252
9	**Frontera** - Dell C6420, Xeon Platinum 8280 28C 2.7GHz, Mellanox InfiniBand HDR, Dell EMC Texas Advanced Computing Center/Univ. of Texas United States	448,448	23,516.4	38,745.9	
10	**Dammam-7** - Cray CS-Storm, Xeon Gold 6248 20C 2.5GHz, NVIDIA Tesla V100 SXM2, InfiniBand HDR 100, HPE Saudi Aramco Saudi Arabia	672,520	22,400.0	55,423.6	

(https://www.top500.org/lists/top500/list/2020/11/)

Motivation

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442,010.0	537,212.0	29,899
2	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
3	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
4	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
5	**Selene** - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband, Nvidia NVIDIA Corporation United States	555,520	63,460.0	79,215.0	2,646
6	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000, NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482
7	**JUWELS Booster Module** - Bull Sequana XH2000 , AMD EPYC 7402 24C 2.8GHz, NVIDIA A100, Mellanox HDR InfiniBand/ParTec ParaStation ClusterSuite, Atos Forschungszentrum Juelich (FZJ) Germany	449,280	44,120.0	70,980.0	1,764
8	**HPC5** - PowerEdge C4140, Xeon Gold 6252 24C 2.1GHz, NVIDIA Tesla V100, Mellanox HDR Infiniband, Dell EMC Eni S.p.A. Italy	669,760	35,450.0	51,720.8	2,252
9	**Frontera** - Dell C6420, Xeon Platinum 8280 28C 2.7GHz, Mellanox InfiniBand HDR, Dell EMC Texas Advanced Computing Center/Univ. of Texas United States	448,448	23,516.4	38,745.9	
10	**Dammam-7** - Cray CS-Storm, Xeon Gold 6248 20C 2.5GHz, NVIDIA Tesla V100 SXM2, InfiniBand HDR 100, HPE Saudi Aramco Saudi Arabia	672,520	22,400.0	55,423.6	

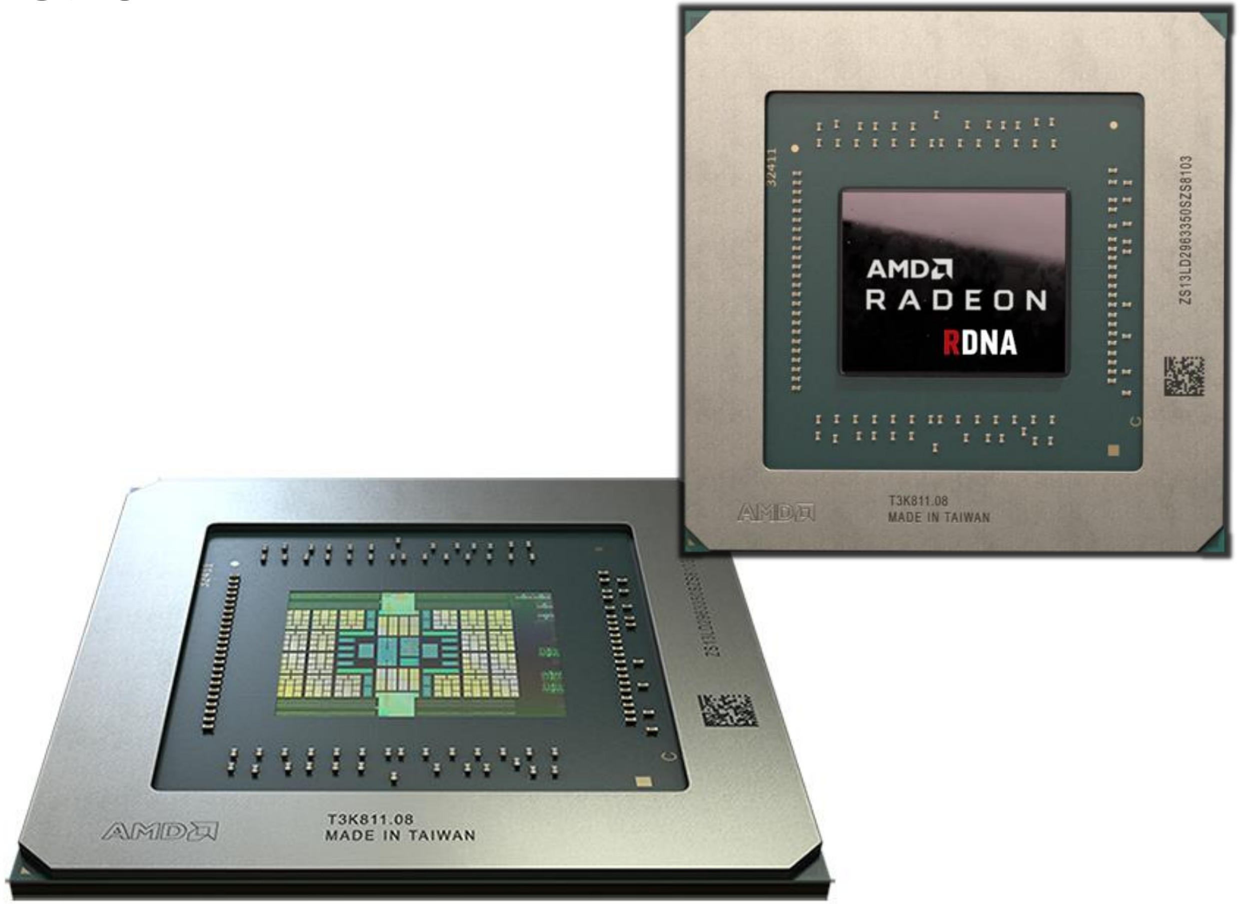
(https://www.top500.org/lists/top500/list/2020/11/)

Graphics card vs. GPU

Graphics card

(nvidia.com)

GPU



(amd.com)

Goal: Two main take-aways

Grasp the programming model for GPUs

Grasp the execution model of GPUs

Outline

- GPU architecture
  - Comparison to CPU
  - A closer look
- Memory hierarchy
- Programming model
- Execution model
- “Hello world!” program

GPU architecture: Comparison to CPU

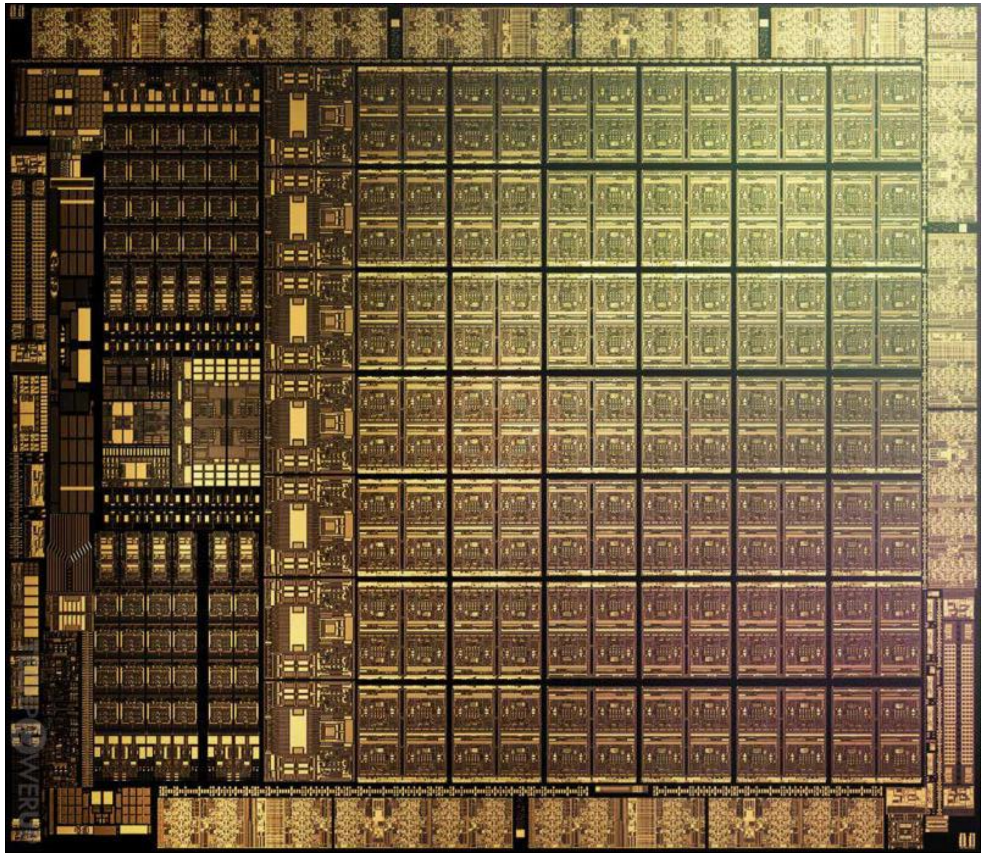
Comparison to CPU: Die shots

Intel Rocket Lake

(intel.com)

NVIDIA Ampere

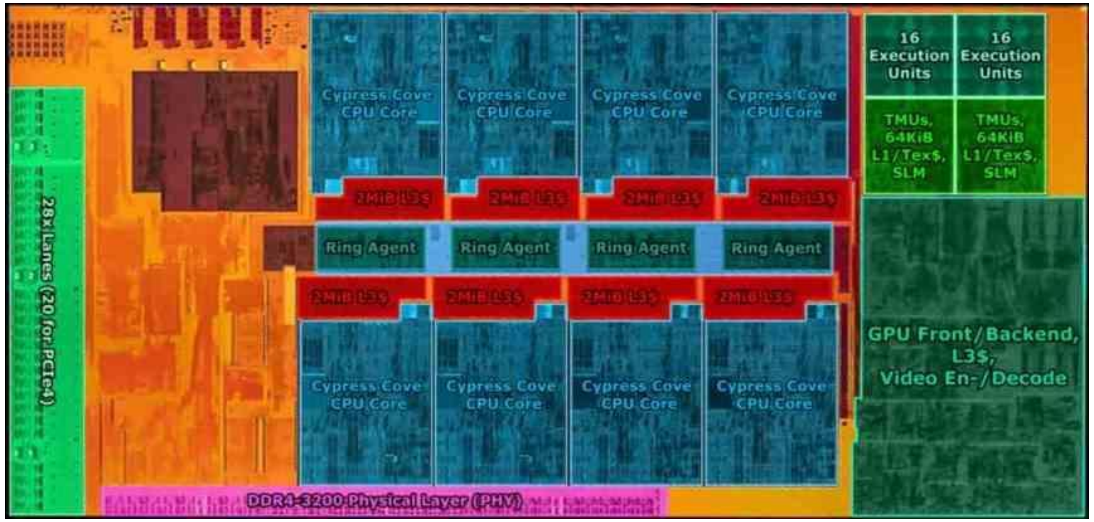
(https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/)



Comparison to CPU: Block diagrams

Intel Rocket Lake

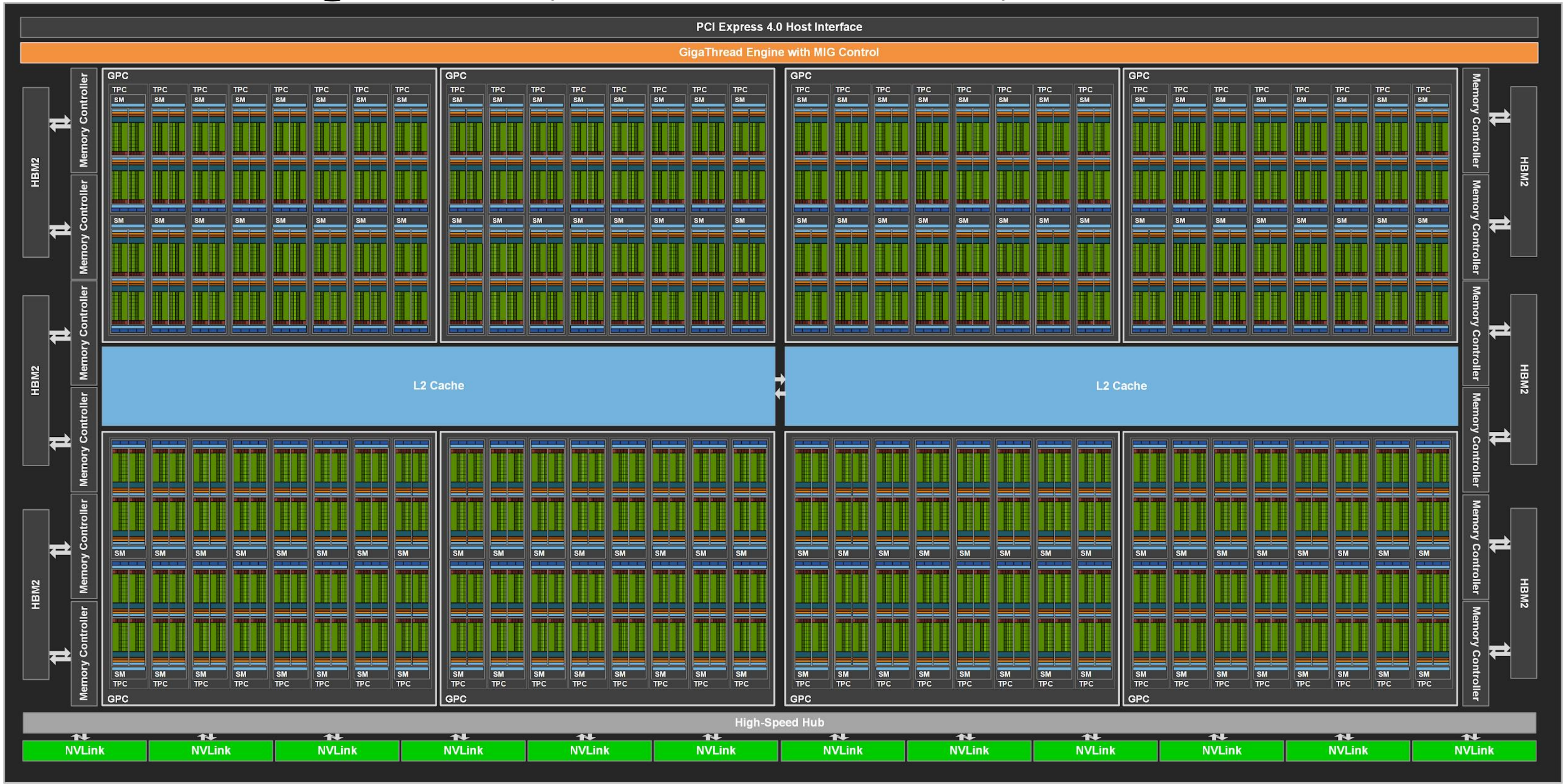
(https://itigic.com/intel-rocket-lake-s-architecture-specifications-and-features/)



NVIDIA Ampere

(https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/)

Comparison to CPU: Block diagrams (GPU detailed)



Comparison to CPU: Block diagrams (CPU core/multiprocessor)

Intel Rocket Lake

(https://en.wikichip.org/wiki/intel/microarchitectures/sunny_cove)

NVIDIA Ampere

(https://devblogs.nvidia.com/nvidia-turing-architecture-in-depth/)

Comparison to CPU: Key properties

	CPU	GPU
Purpose	General purpose device to deal with all kinds of workloads	Extremely specialized to render polygon-based 3D scenes
Single thread performance	Highly optimized: • Pipelining • Super scalarity • Out-of-order execution • Speculative execution/branch prediction • Register re-naming	Marginal: • In-order execution
Level of parallelism	Moderate: • Several cores • Vector units (512-bit)	High: • Thousands of execution units • Depending on vendor, also vector units
Programmability	Huge set of programming lanuages, models, and paradigms	Since „recently“, no programmability at all

Comparison to CPU: Numbers

		CPU (Intel Core i9-11900K)	GPU (NVIDIA A100 GPU accelerator)
# Cores		8	6912/108
Frequency in GHz		3.5	1.41
# Registers		2,784	$864 \cdot 2^{10}$ (= 27 MB)
Peak performance in TFLOPS	SP	1.344	19.5
	DP	0.672	9.7
Memory bandwidth in GB/s		50	1555
Die size in mm^2		~200	826
TDP in W		125	400

GPU architecture:
A closer look

A closer look: Nomenclature

AMD

intel

NVIDIA

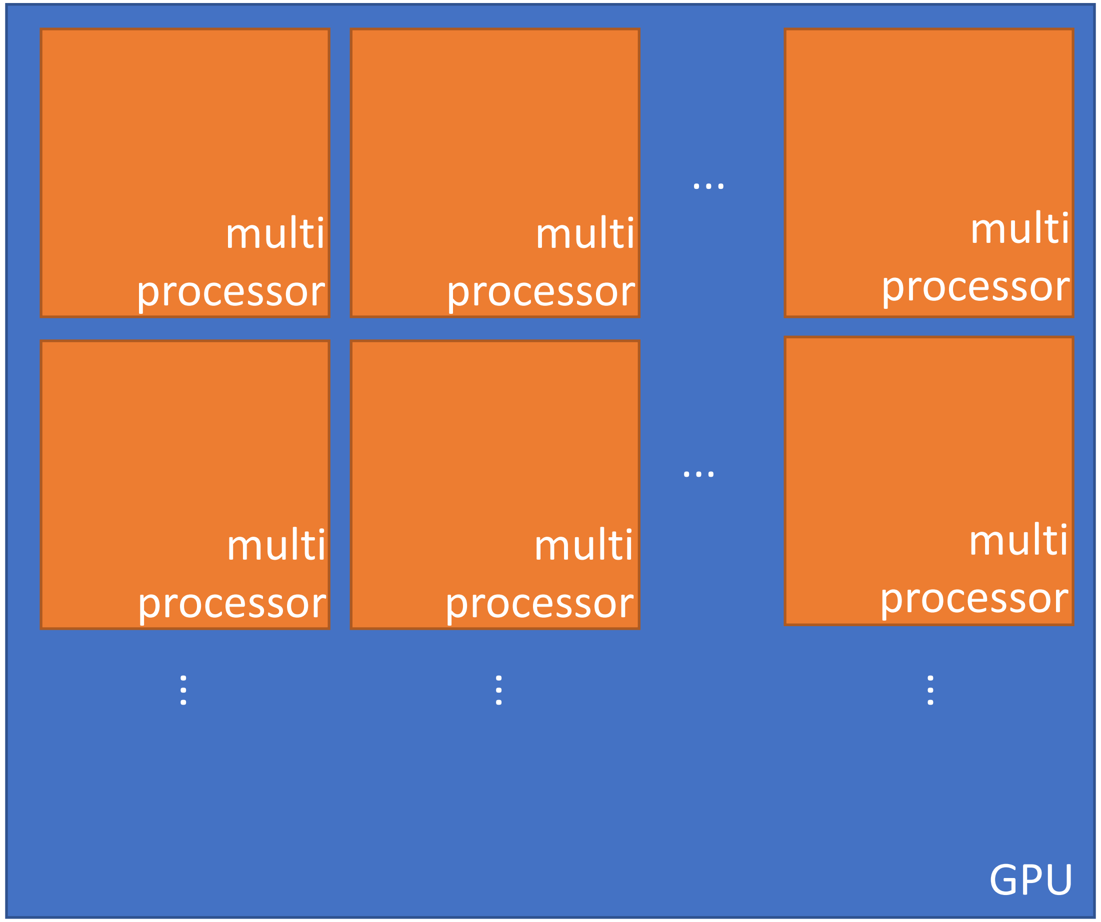
KHRONOS GROUP

A closer look: Nomenclature

A closer look: Two level parallelism

Higher level:

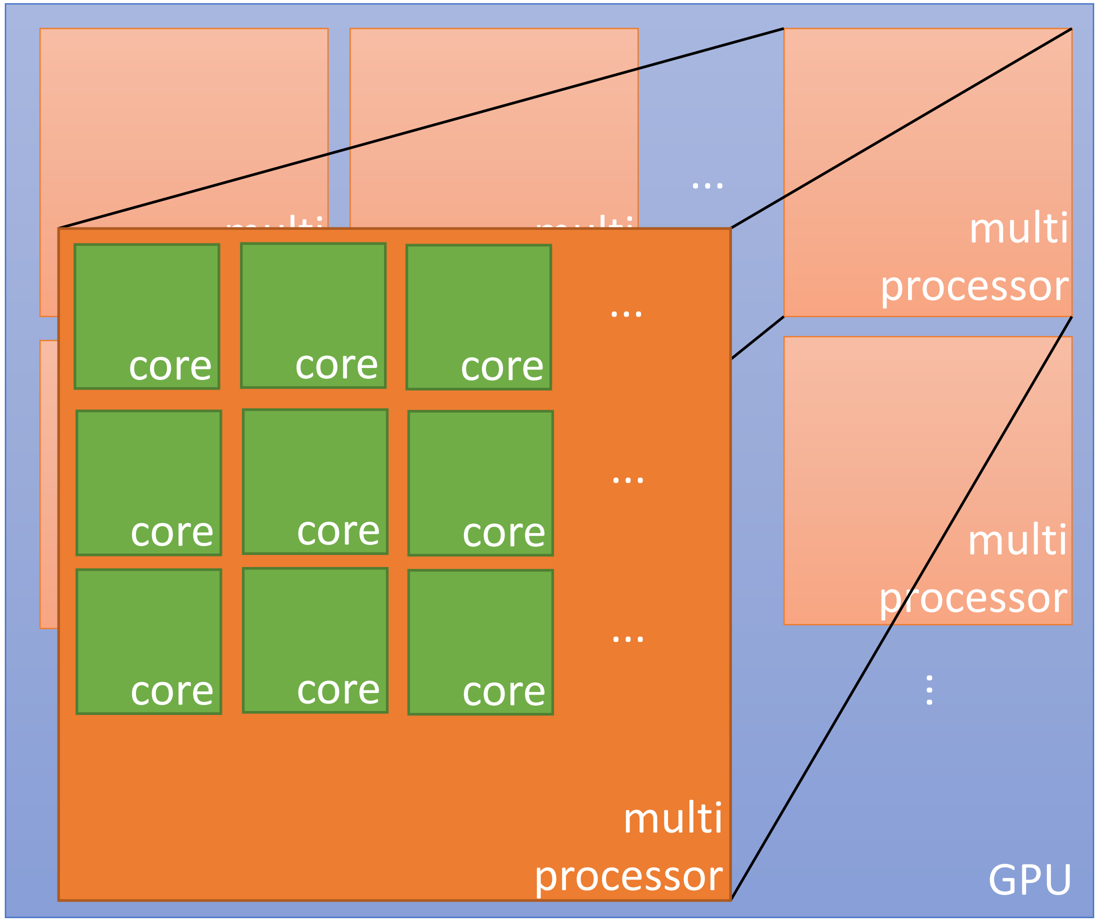
- The high level elements are called
  - *multiprocessor* (NVIDIA)
  - *compute unit* (AMD)
  - *subslices* (Intel)
- Typically, there are between one and 80 high level elements in one GPU



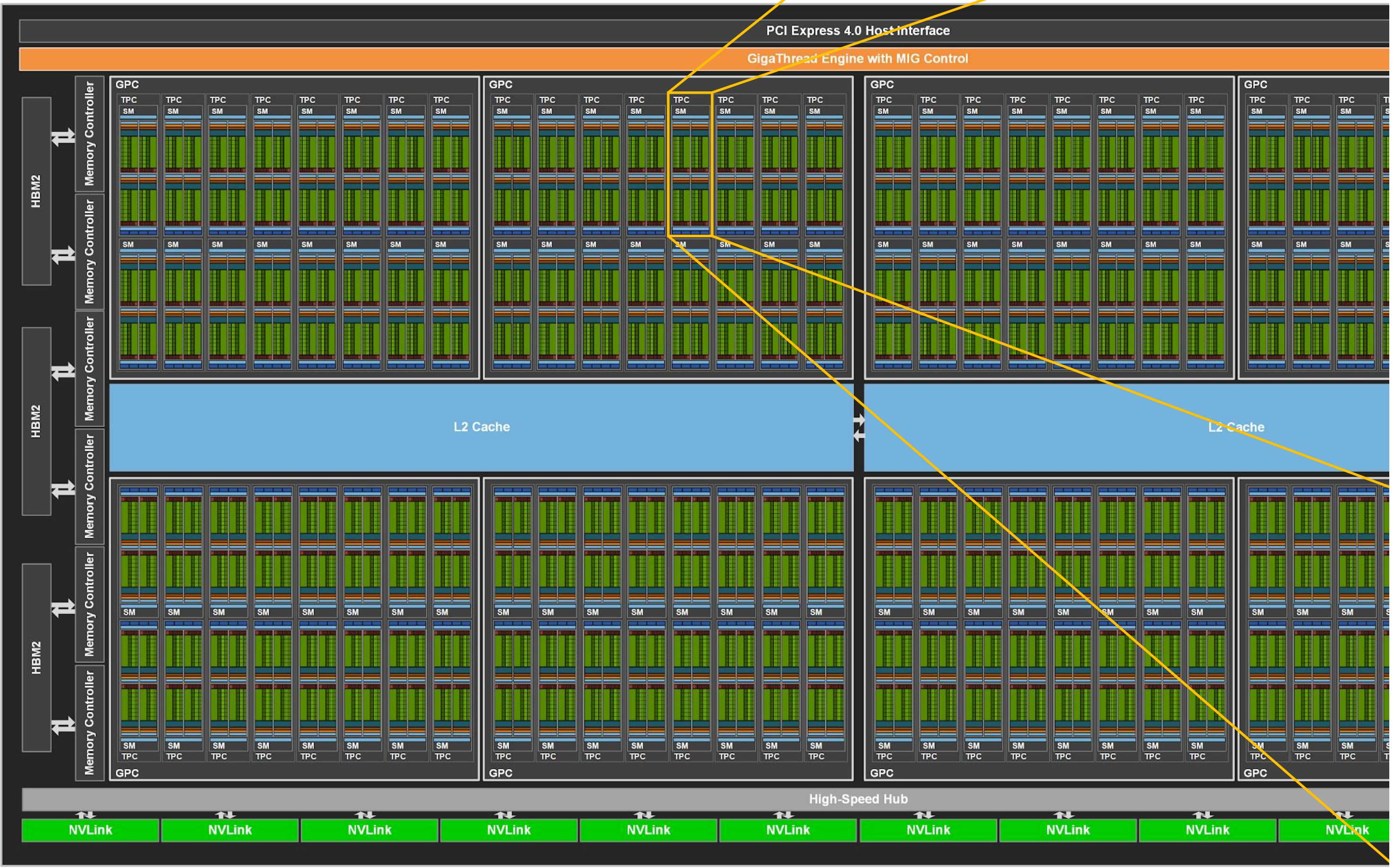
A closer look: Two level parallelism

Lower level:

- The low level elements are called
  - *cores* (NVIDIA)
  - *vector elements* (AMD)
  - *hardware threads* (Intel)
- Typically, there are between 32 and 192 low level elements in a high level element



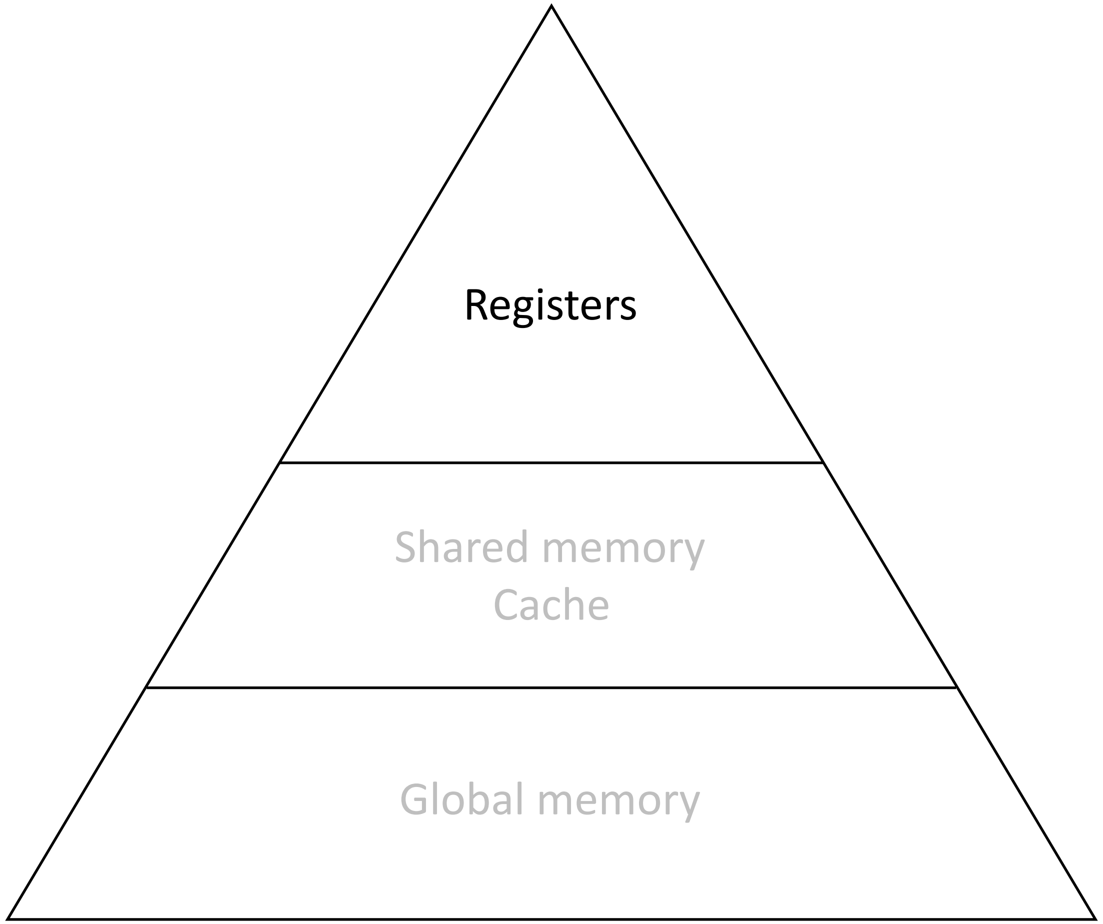
A closer look: Summary



Memory hierarchy

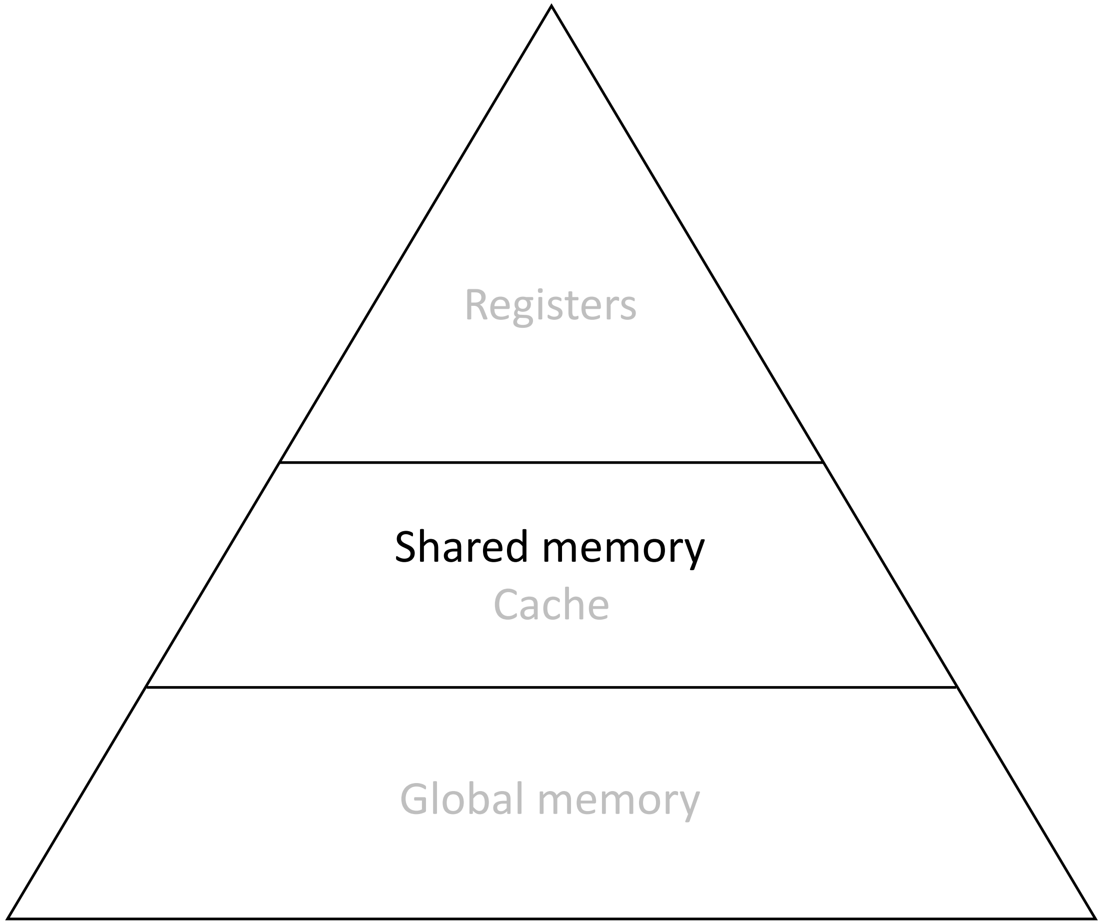
Memory hierarchy: Registers

Size	Several ten thousands per multi-processor
Latency	Low single digit clock ticks
Location	On-chip
Persistence	Runtime of the kernel



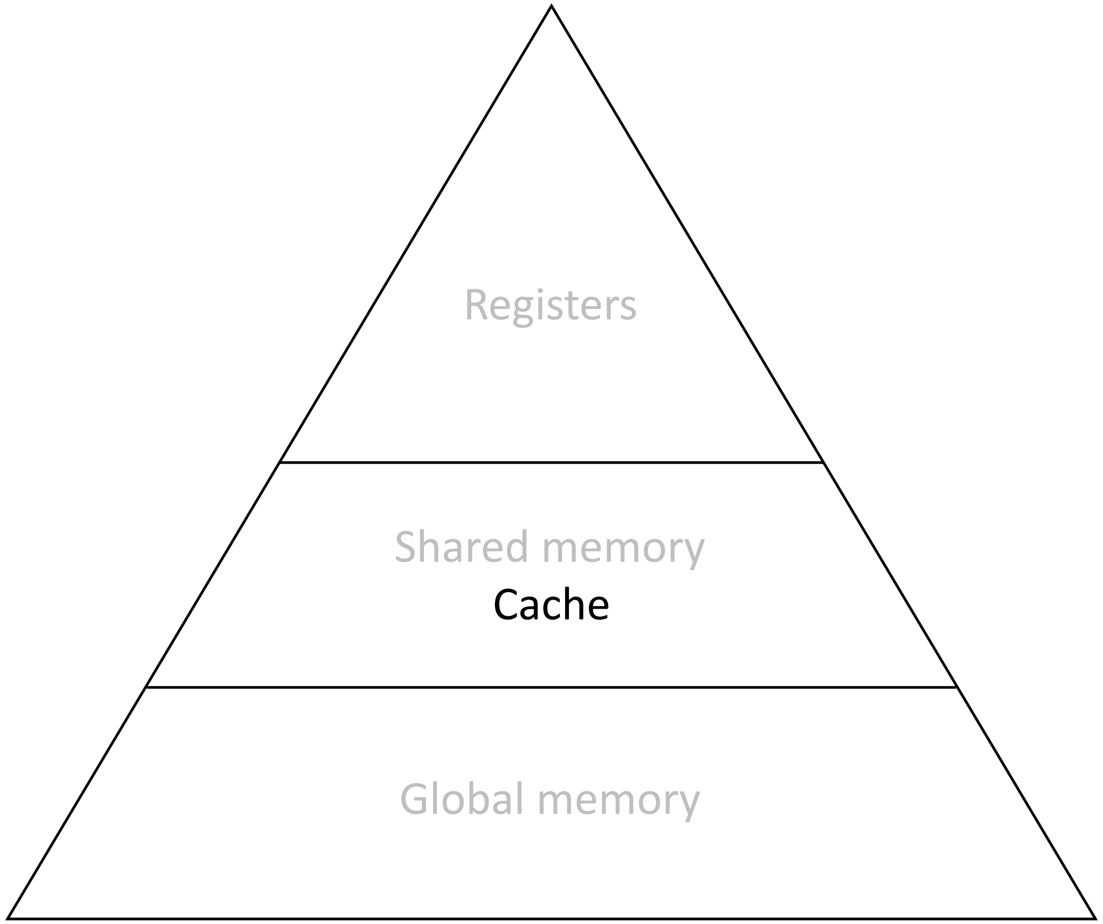
Memory hierarchy: Shared memory

Visibility	• Each multi-processor has its own dedicated shared memory • No other multi-precossor can access the shared memory of a multi-processor
Size	Several tens of KB
Latency	Low double digit clock ticks
Location	On-chip
Persistence	Runtime of the kernel



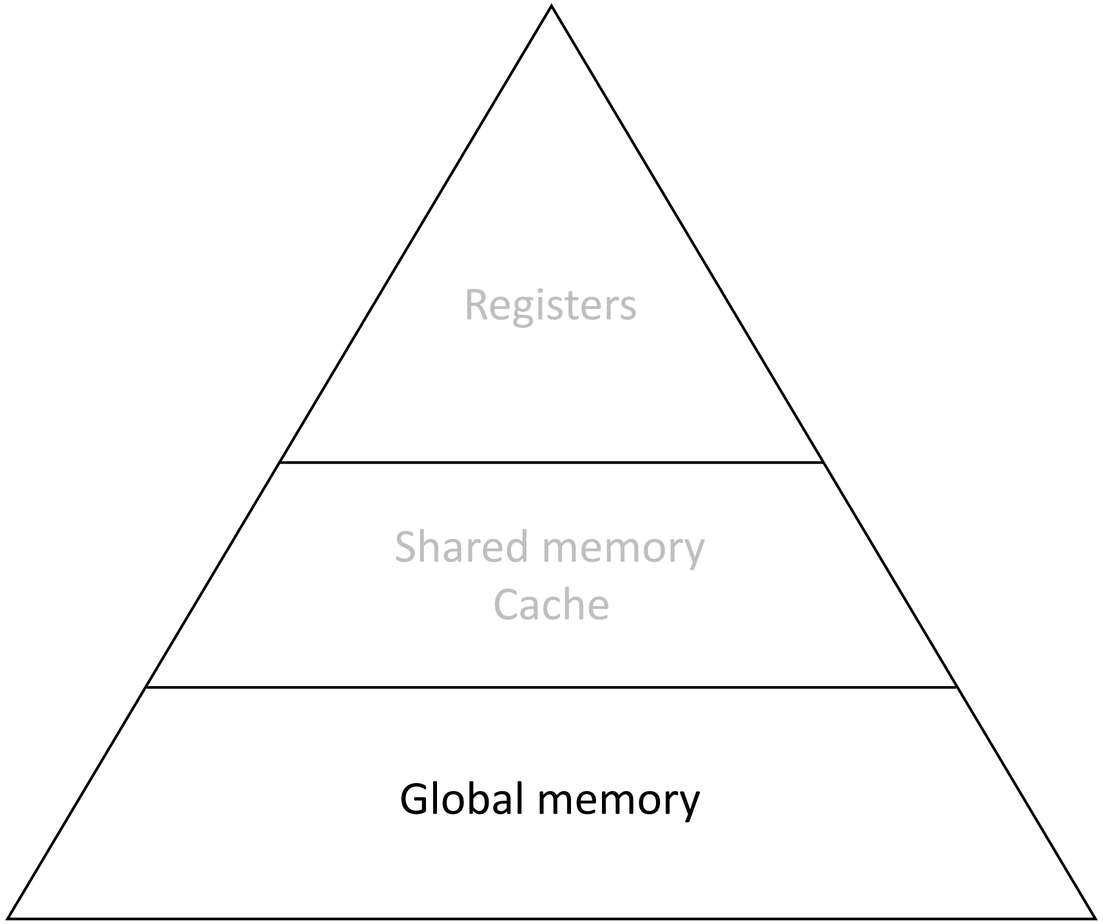
Memory hierarchy: Cache

	L1	L2
Visibility	All cores of a multiprocessor access the same L1	All cores of all multiprocessors access the same L2
Size	Several tens of KB	Several hundrets of KB
Latency	Low single digit clock ticks	Low double digit clock ticks
Location	On-chip	
Persistence	Runtime of the kernel	

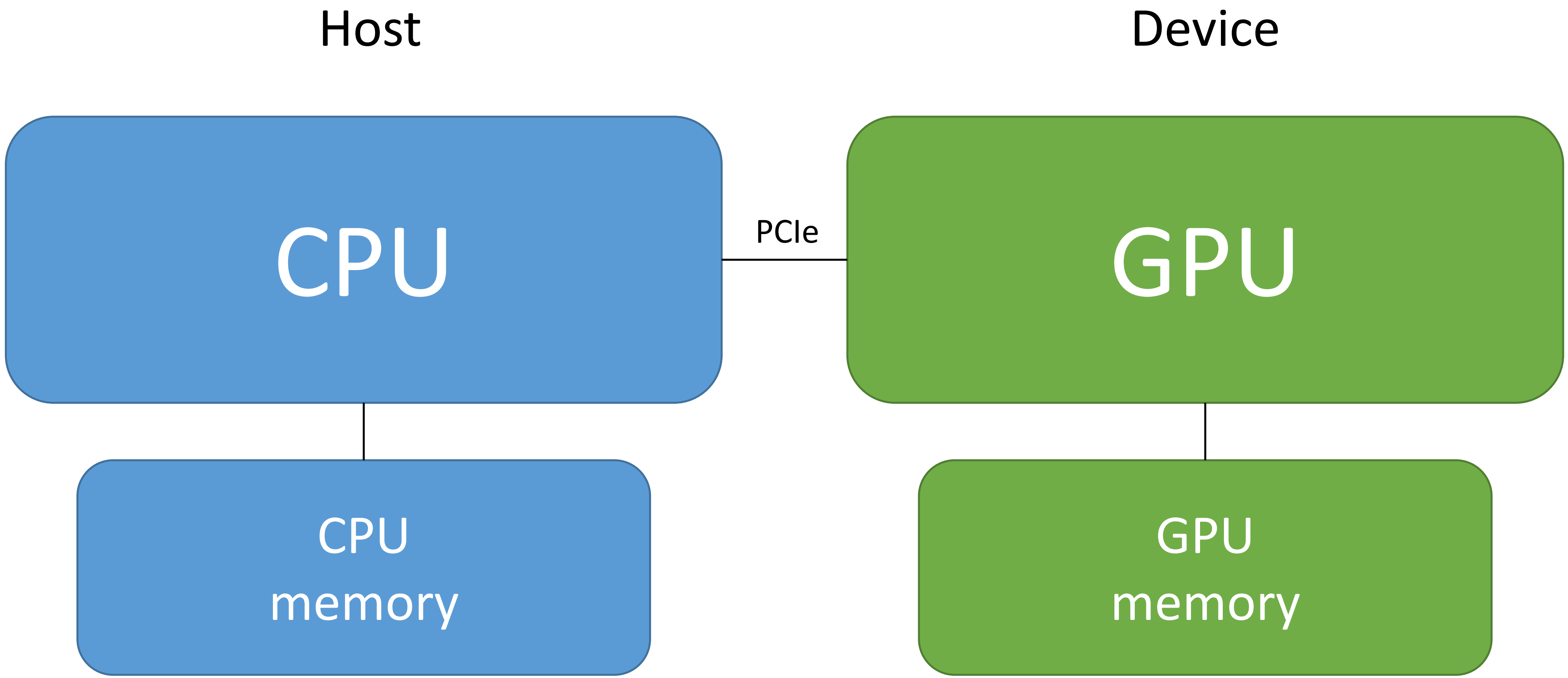


Memory hierarchy: Global memory

Visibility	All cores can access every single address
Size	Several GB
Latency	Low triple digit clock ticks
Location	Off-chip
Persistence	Runtime of the program
Remarks	Huge shared memory device



Memory hierarchy: Disjunct memory spaces



Programming model

Programming model: Programming platforms

Programming model: Programming platforms

Programming model: Kernel

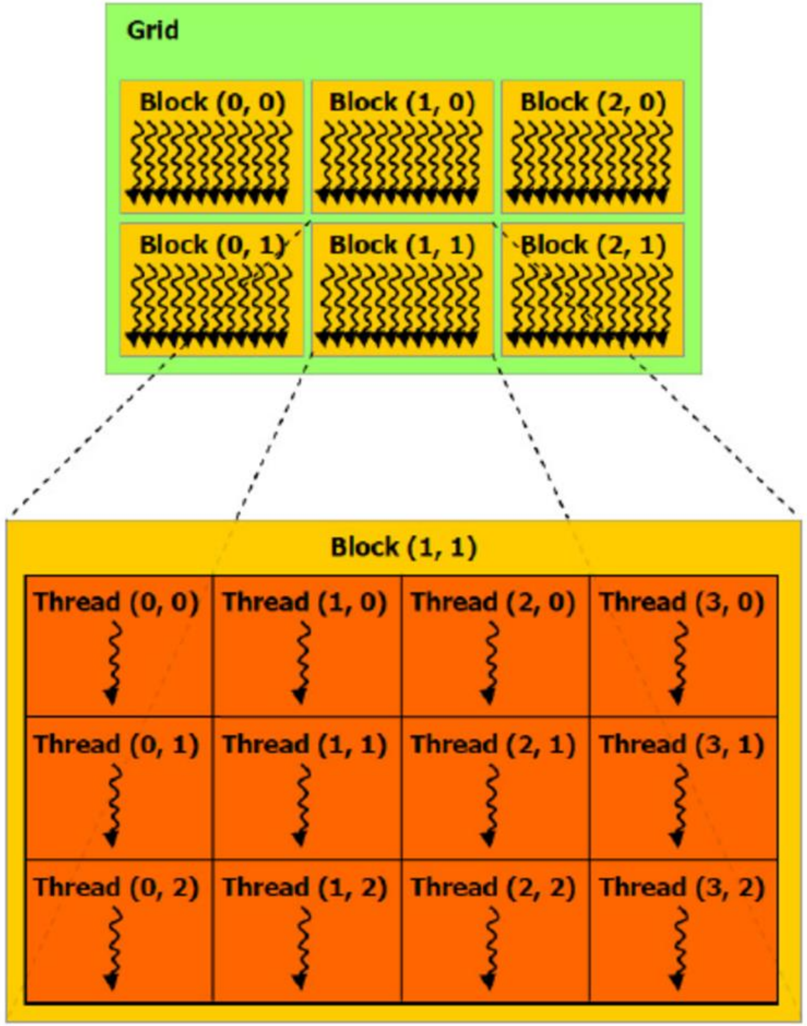
- Programs for the GPU are expressed as *kernels*
- A kernel is a C-like function:

```
__global__ void foo(type param1, type param2)
{
    // Do stuff on the GPU
}
```

- A kernel contains sequential scalar code
  - No multi-tasking statements
  - No multi-threading statements
  - No vectorization statements
- When called, a kernel is executed N times in parallel by N different *threads*

Programming model: Two level parallelism and invocation

- Threads organized by two level parallelism:
  - A *block* consists of multiple threads
  - The *grid* consists of multiple blocks
- Blocks and the grid can each be 1-, 2-, or 3-dimensional
- Call syntax:
  ```
  foo<<<gridConf,blockConf>>>
  (param1, param 2);
  ```
- Parallel setup stays constant



(https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html)

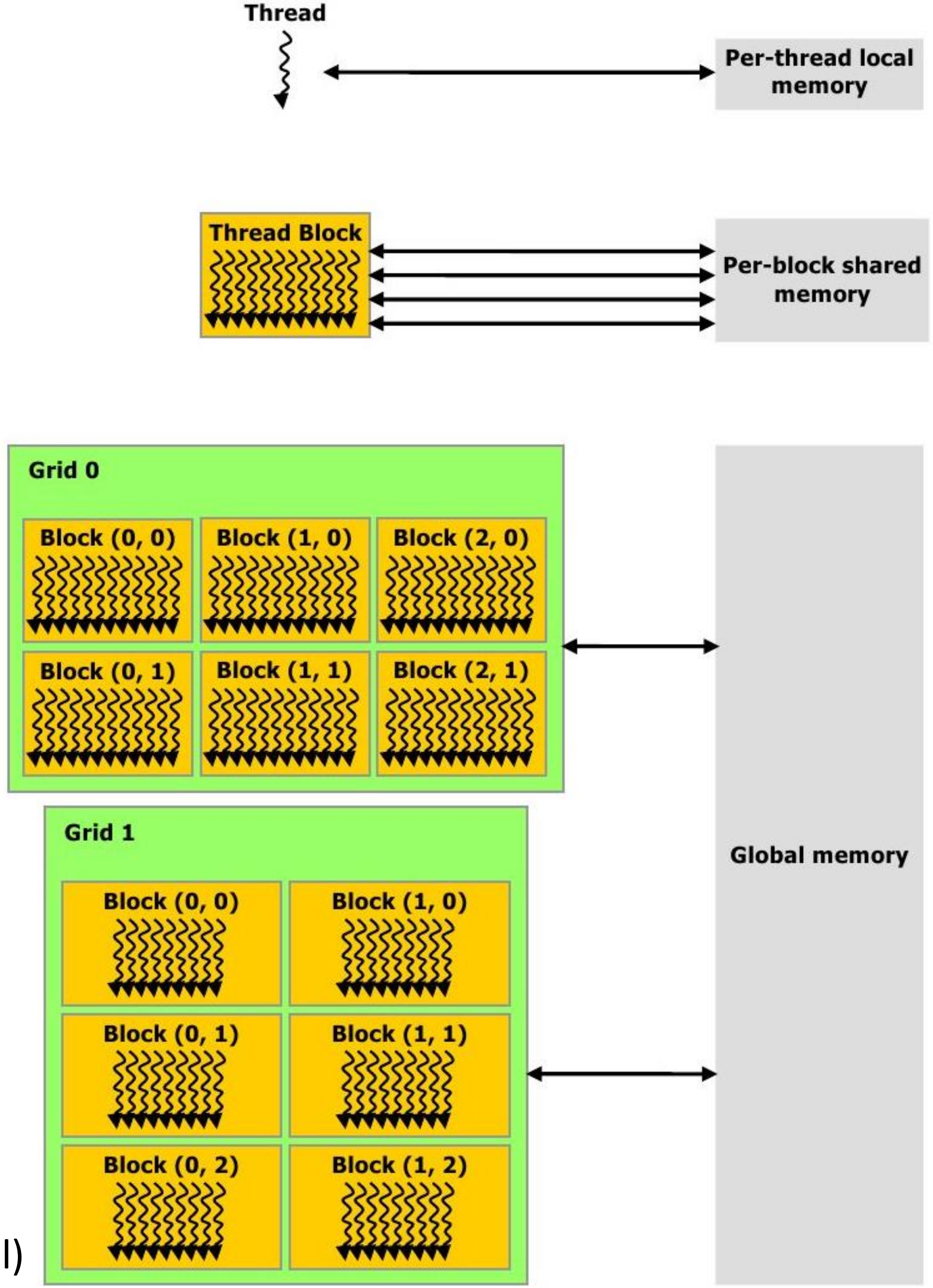
Goal: Two main take-aways

Grasp the programming model for GPUs

Grasp the execution model of GPUs

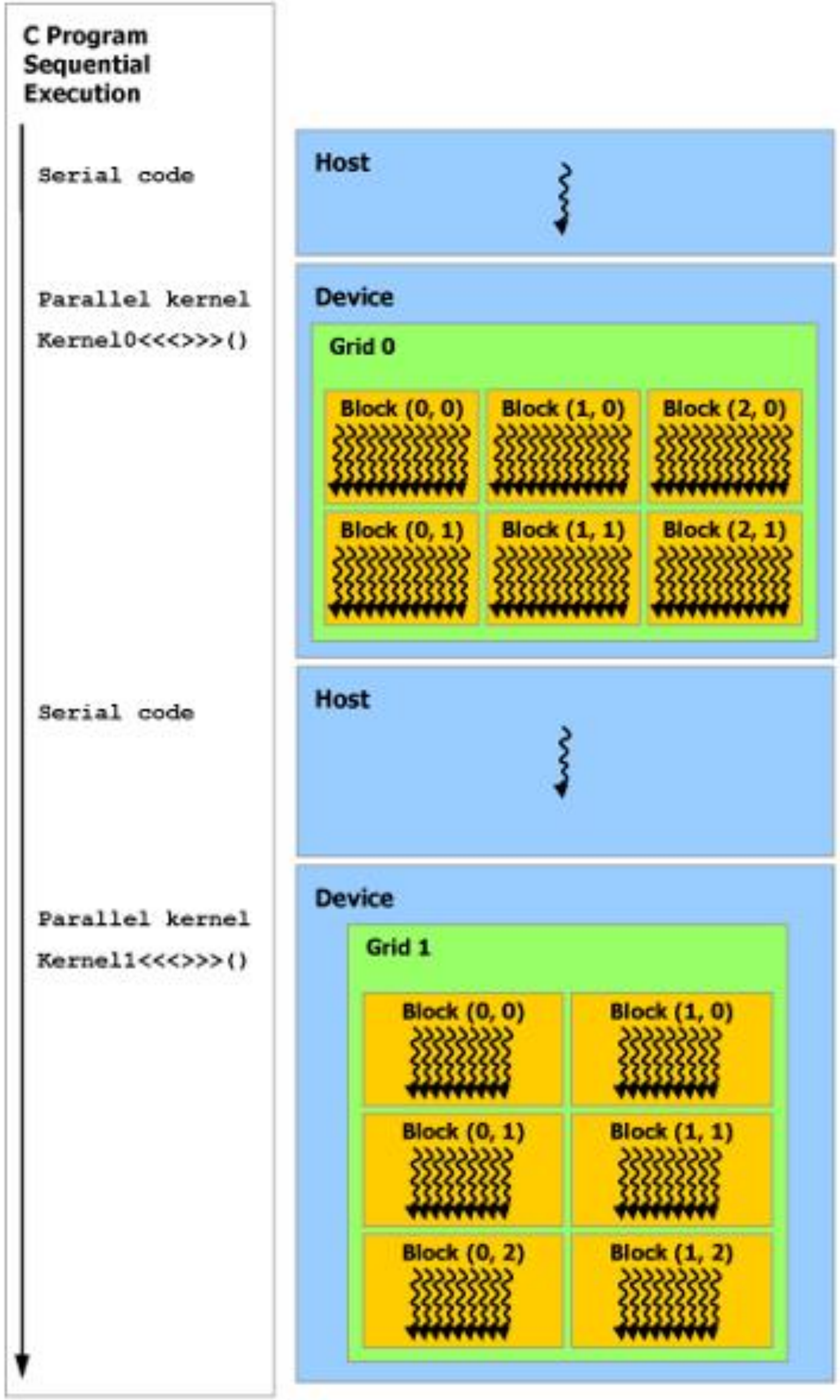
Programming model: Memory hierarchy revisited

- Each thread has its own registers
- Communication between threads within one particular block occurs via shared memory
- Communication between threads of different blocks happens via global memory
- Communication between differet kernels is done via global memory



(https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html)

Programming model: Workflow

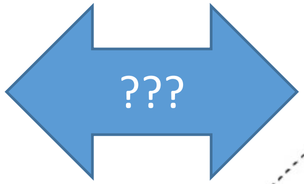


(https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html)

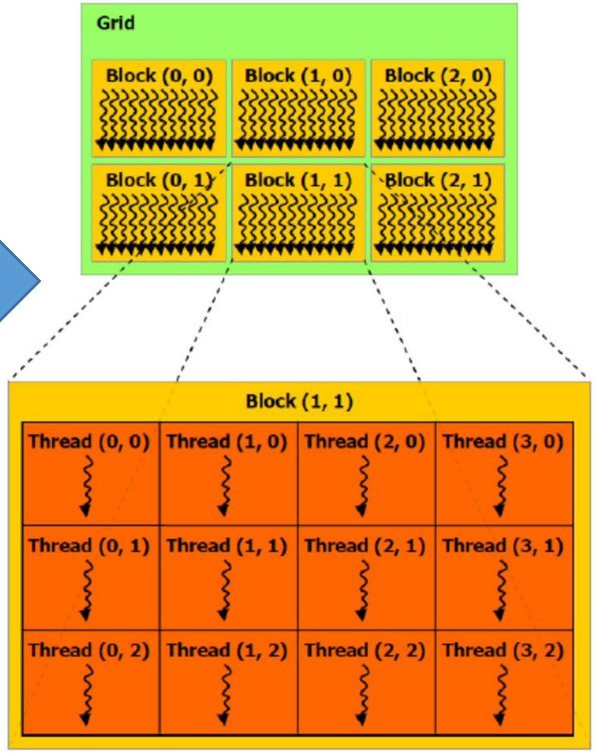
Execution model

Execution model: Mapping parallelism

Hardware (cores | multiprocessors)

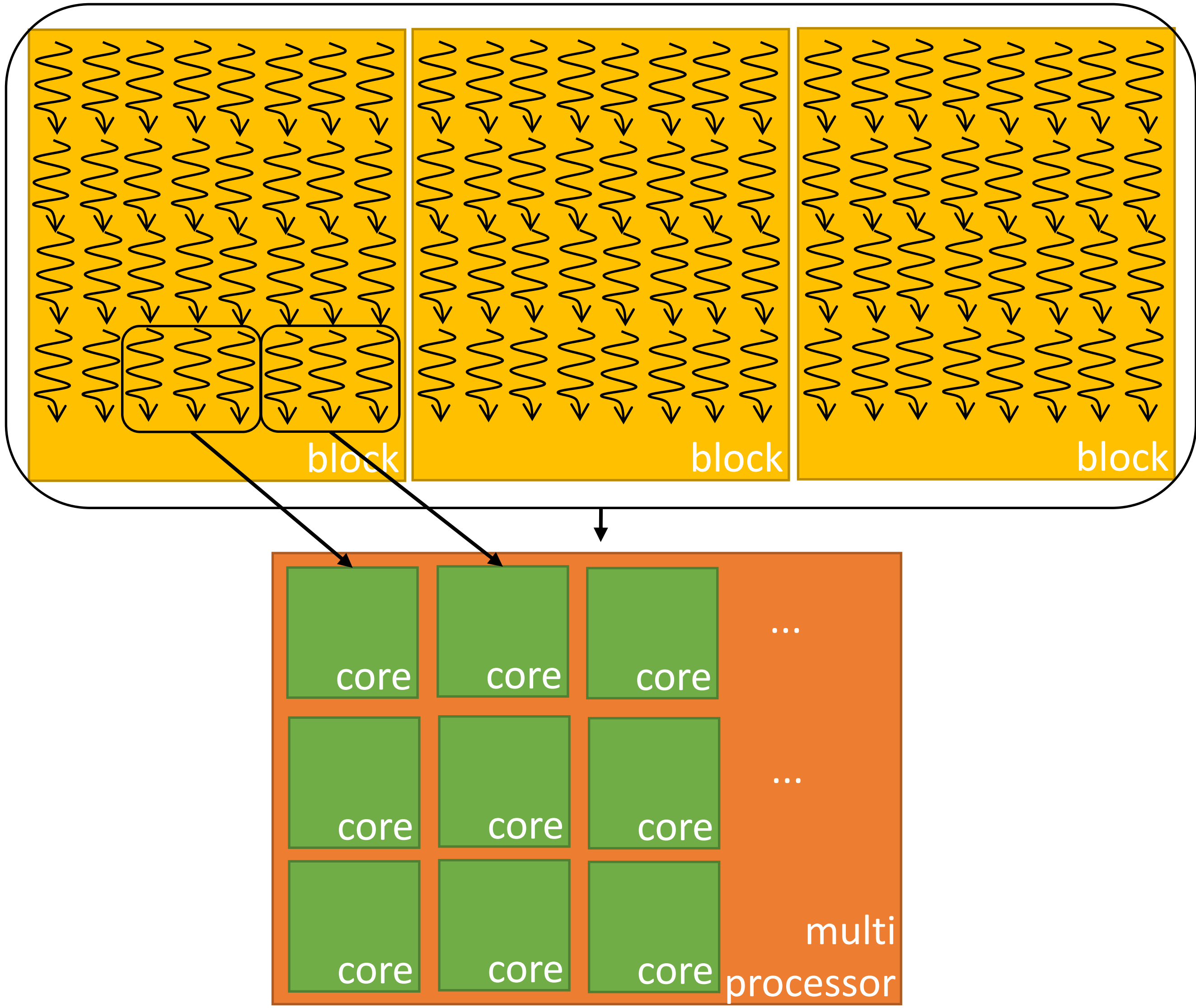


Programming model (threads | blocks)



Execution model: Mapping parallelism

- Multiple blocks are scheduled to one multiprocessor
- A particular block is not distributed over multiple multiprocessors
- Each thread runs on one core
- Parallel configuration is independent from underlying hardware making it very flexible and oblivious



Execution model: Efficiency by oversubscription

- GPUs can switch in threads with „zero overhead“
- There should be **much** more threads than cores
- This increases the likelihood of ready threads
- There should be „enough to schedule“
- Cores are fully occupied
- Communication is hidden by computations

Goal: Two main take-aways

Grasp the programming model for GPUs

Grasp the execution model of GPUs

Execution model: Occupancy

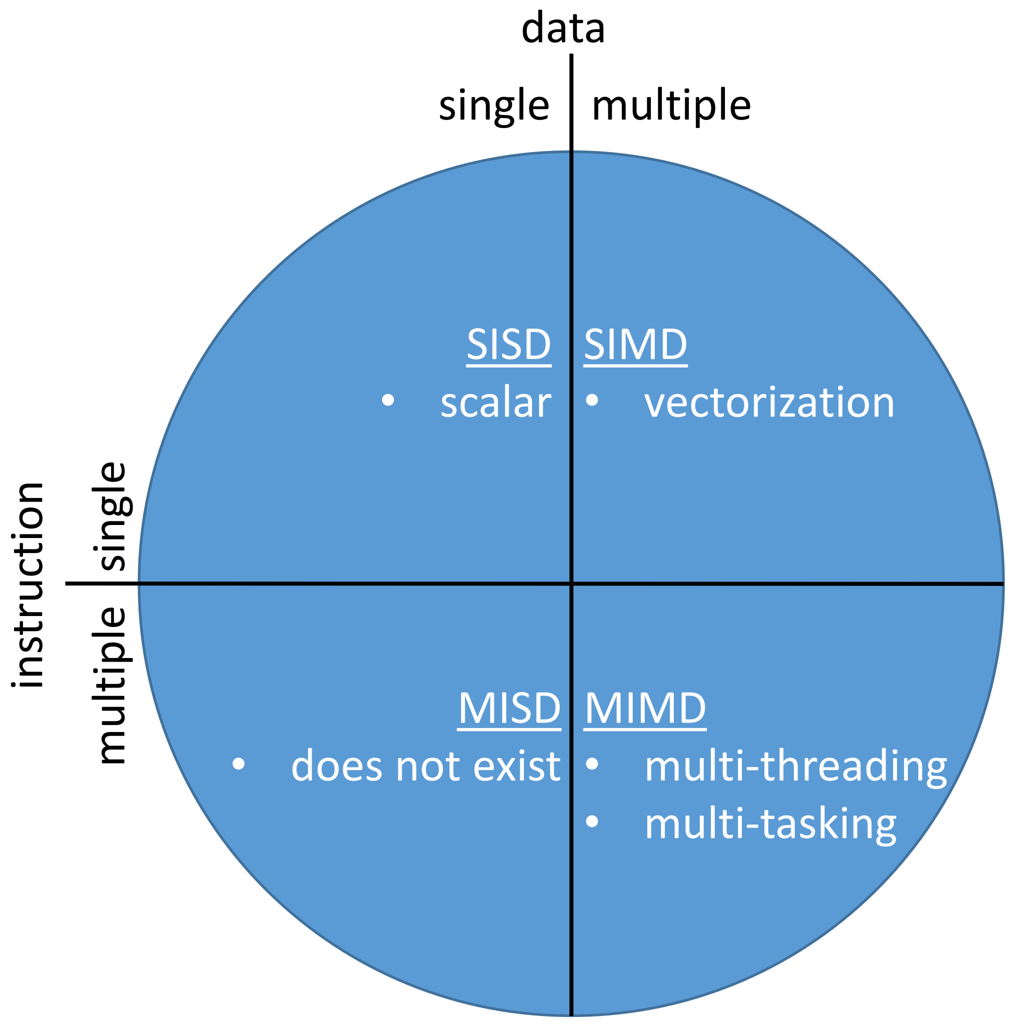
- Statement of „enough to schedule“ can be quantified by *occupancy* $\in \,]0,1]$
- Occupancy = $\min\left(1, \frac{\text{active threads on multiprocessor}}{\text{maximum number of active threads}}\right)$
  - Active threads on multiprocessor: Determined by register and shared memory consumption
  - Maximum number of active threads: Hardware-specific value
- Example (for limitation by registers):
  -

Number of registers per multiprocessor	$64 \cdot 2^{10}$
Register consumption per thread	37
Maximum number of resident threads per multiprocessor	2,048

  - Occupancy = $\frac{\frac{64 \cdot 2^{10}}{37}}{2{,}048} \approx 0.865$
- That‘s the reason why there are so many registers

Execution model: Warps and SIMT

- Flynn's taxonomy to classify computer architectures
- A *warp* is a pack of 32 threads
- Warps follow *single instruction multiple threads* (SIMT):
  - Each thread of a warp executes the same instruction at a time
  - Each thread can but does not has to operate on different data
  - Conditional statements can lead to warp divergence
- Instruction schedulers operate on the level of warps



Conclusion

- GPUs rely on a completely different architecture than CPUs
- Their massive parallelism...
  - ...favours a completely different programming model
  - ...requires a completely different execution model
- The huge potential of computational power is only exploitable for parallelizable workloads

Tutorial preparation

Tutorial preparation: Some necessary ingradients

- At the runtime of a kernel, every thread can determine its thread index within a thread block and its block index within the grid in each direction
  - `threadIdx.x/y/z`
  - `blockIdx.x/y/z`
- The same holds for the dimensions of the blocks and the grid
  - `blockDim.x/y/z`
  - `gridDim.x/y/z`
- Modifiers
  - `__global__`: Marks a kernel
  - `__shared__`: Marks shared memory
- All threads of a block are synchronized via `__syncthreads()`

Tutorial preparation: The „Hello World“ of CUDA

```
__global__ void vectorAdd(
  float* in_a,
  float* in_b,
  float* out_c)
{
  int globalThreadIdx =
    blockIdx.x * blockDim.x +
    threadIdx.x;

  out_c[globalThreadIdx] =
    in_a[globalThreadIdx] +
    in_b[globalThreadIdx];
}
```

```
int main()
{
  float *hA, *hB, *hC, *dA, *dB, *dC;

  // Allocate memory on the host and device
  hA = new float[N]; ...
  cudaMalloc(&dA, N * sizeof(float)); ...

  ... // Initiate vectors on the host

  // Copy input vectors to the device
  cudaMemcpy(dA, hA, N * sizeof(float), cudaMemcpyHostToDevice); ...

  // Invoke kernels on the device
  dim3 gridConf(BLOCKS_PER_GRID, 1, 1); dim3 blockConf(THREADS_PER_BLOCK, 1, 1);
  vectorAdd<<<gridConf, blockConf>>>(dA, dB, dC);

  // Copy result to the host
  cudaMemcpy(hC, dC, N * sizeof(float), cudaMemcpyDeviceToHost);

  ... // Clean up
}
```