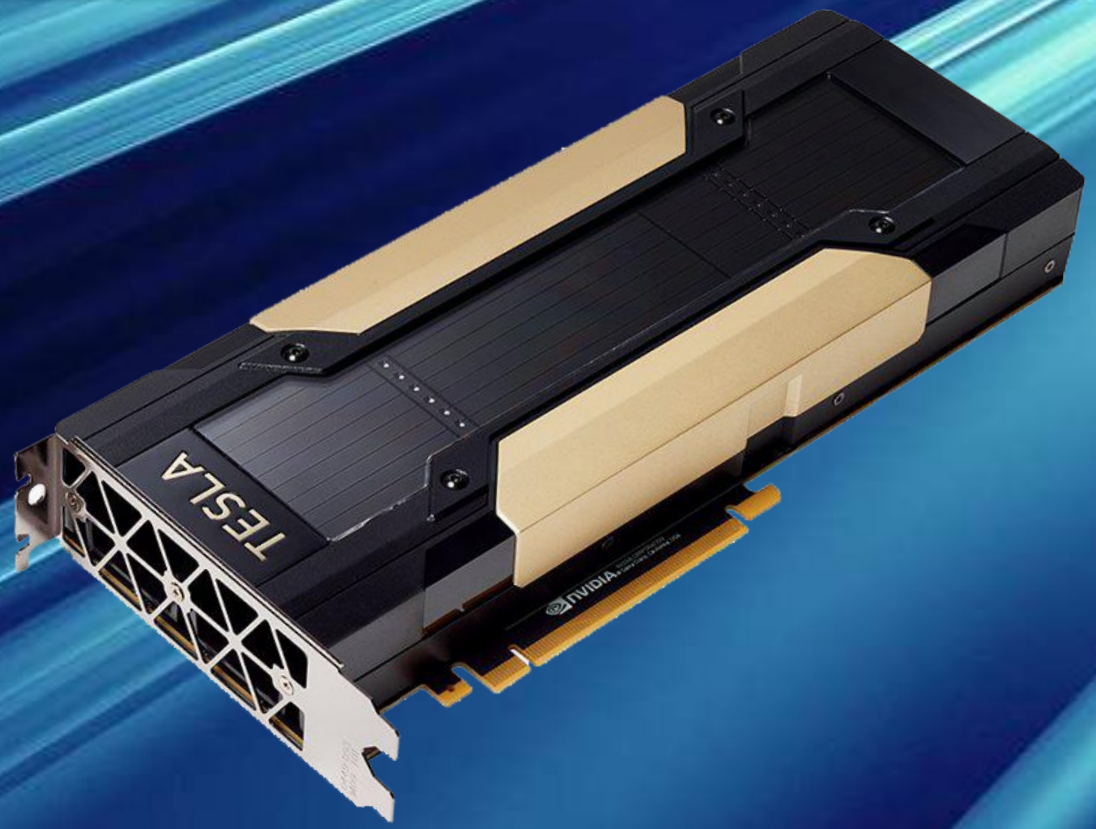
sciCORE
Center for scientific computing

Universität Basel

SIB
Swiss Institute of Bioinformatics



Introduction to GPU programming

Outline

Day 1

(09:00 – 13:00)

- ❑ Introduction
- ❑ Coding a CUDA kernel
- ❑ Example
  - ❑ Matrix Multiplication + Exercise
- ❑ Real world implementations
  - ❑ Genomes simulation
  - ❑ Random numbers generation
- ❑ Homework

Day 2

(09:00 – 13:00)

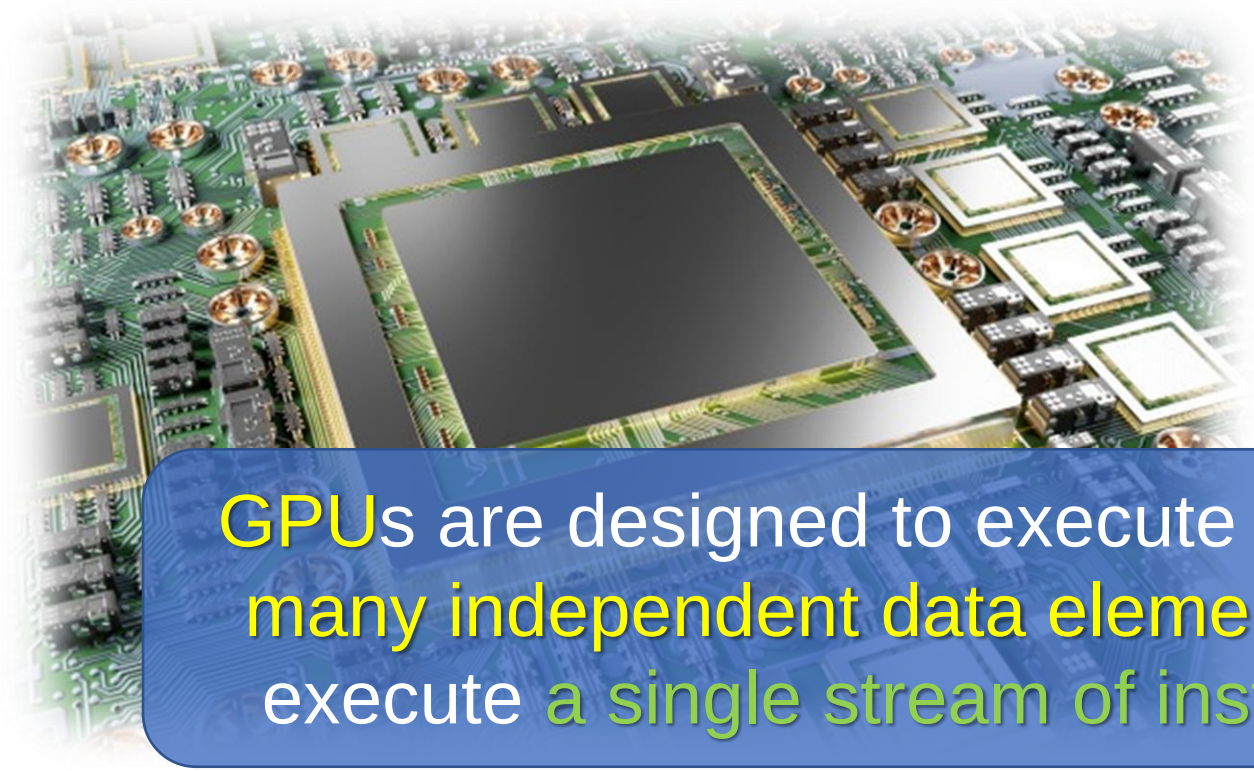
- ❑ Recap and homework solutions
- ❑ Common pitfalls
- ❑ Memory management
  - ❑ Image processing example
  - ❑ Sequence alignment
- ❑ Exercise
  - ❑ Area of the Mandelbrot set
- ❑ Monitoring and asynchronicity
  - ❑ Smoothed Particle Hydrodynamics
- ❑ Final remarks

Outline

Day 1

(09:00 – 13:00)

- ❑ Introduction
- ❑ Coding a CUDA kernel
- ❑ Example
  - ❑ Matrix Multiplication + Exercise
- ❑ Real world implementations
  - ❑ Genomes simulation
  - ❑ Random numbers generation
- ❑ Homework



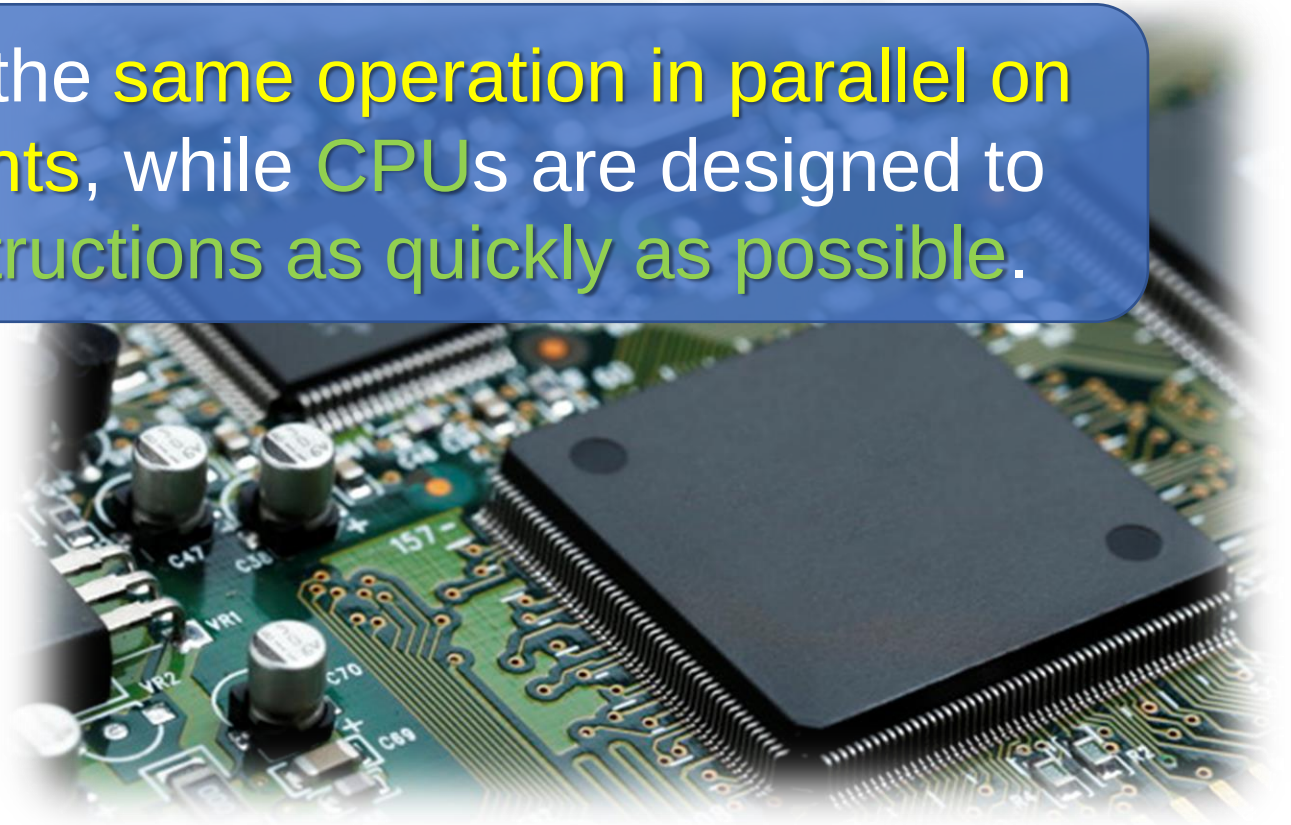
CPU

Central Processing Unit

GPUs are designed to execute the same operation in parallel on many independent data elements, while CPUs are designed to execute a single stream of instructions as quickly as possible.

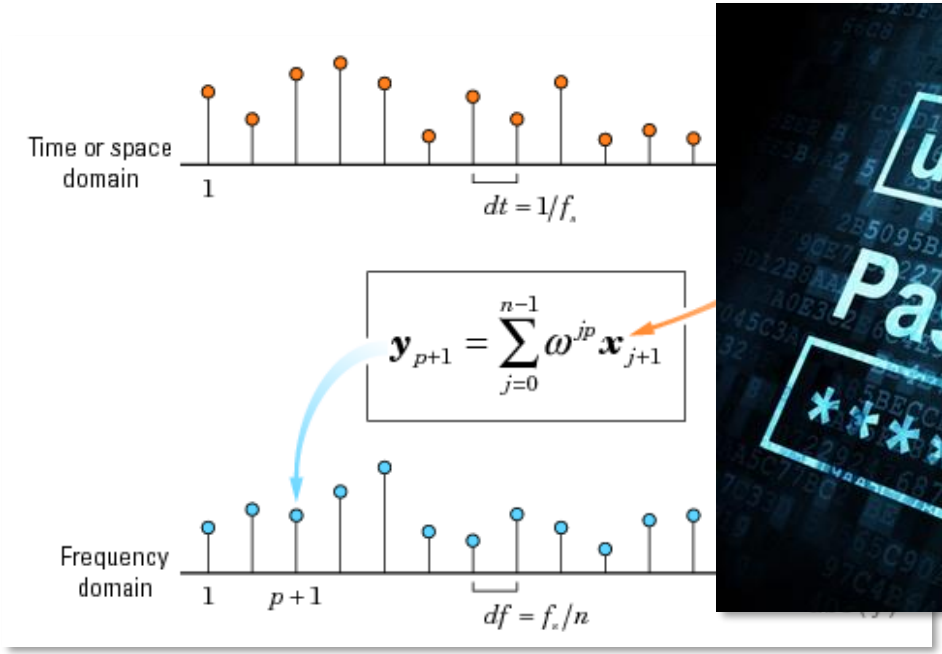
GPU

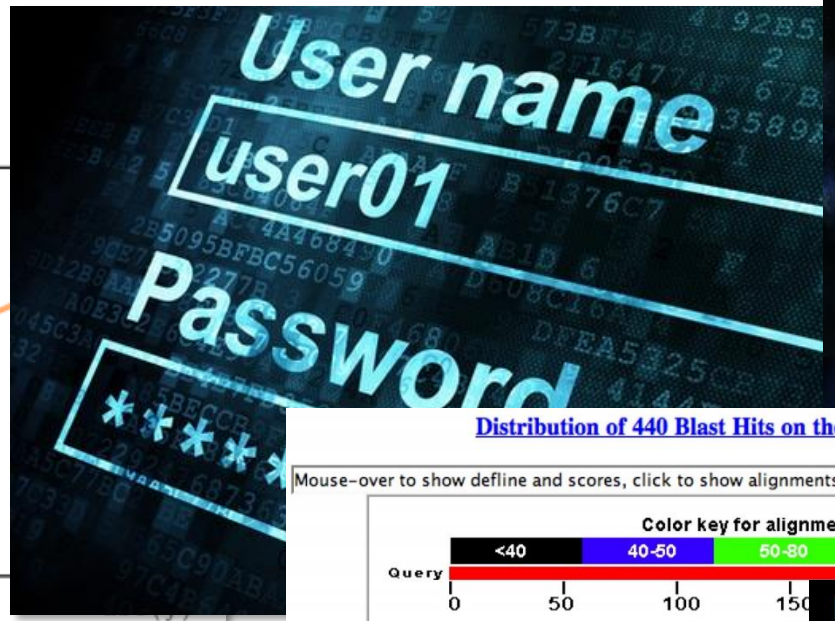
Graphics Processing Unit

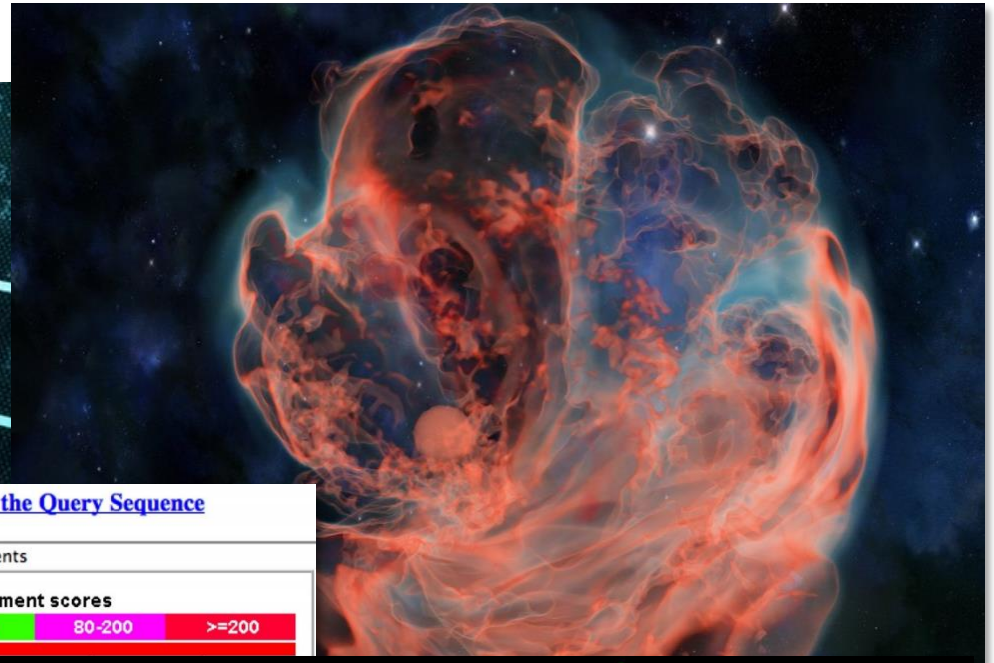


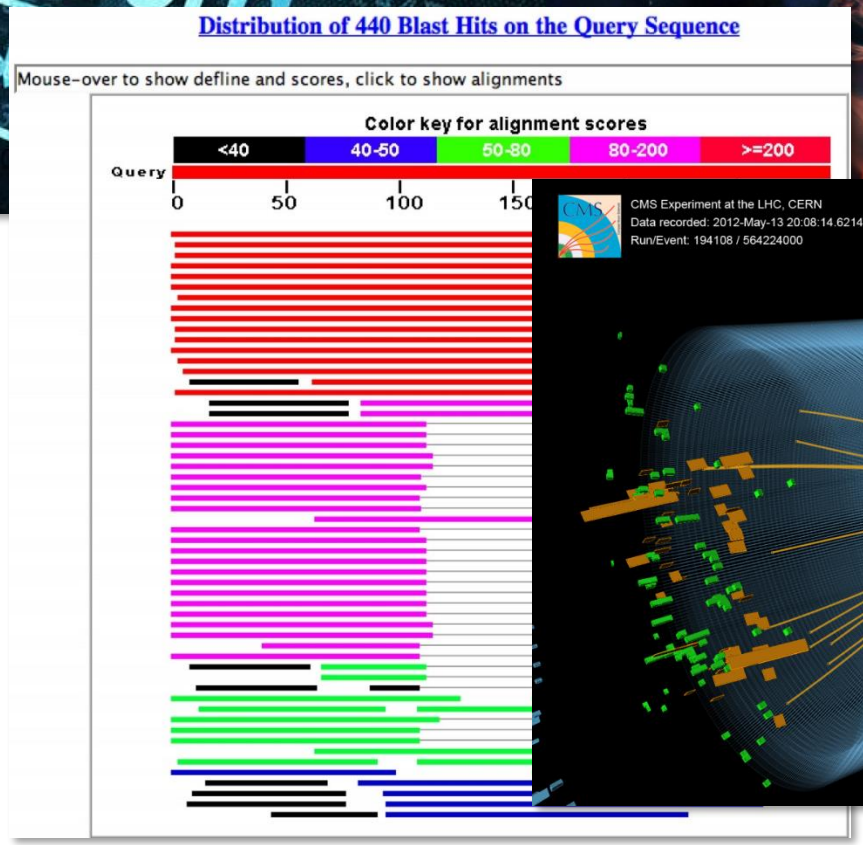
GPU

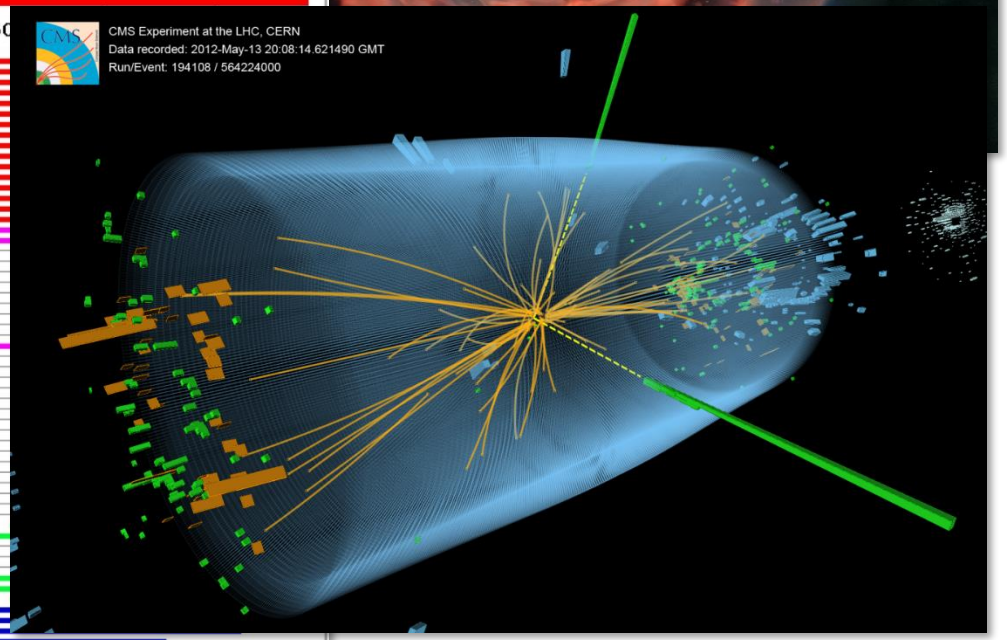
Graphics Processing Unit







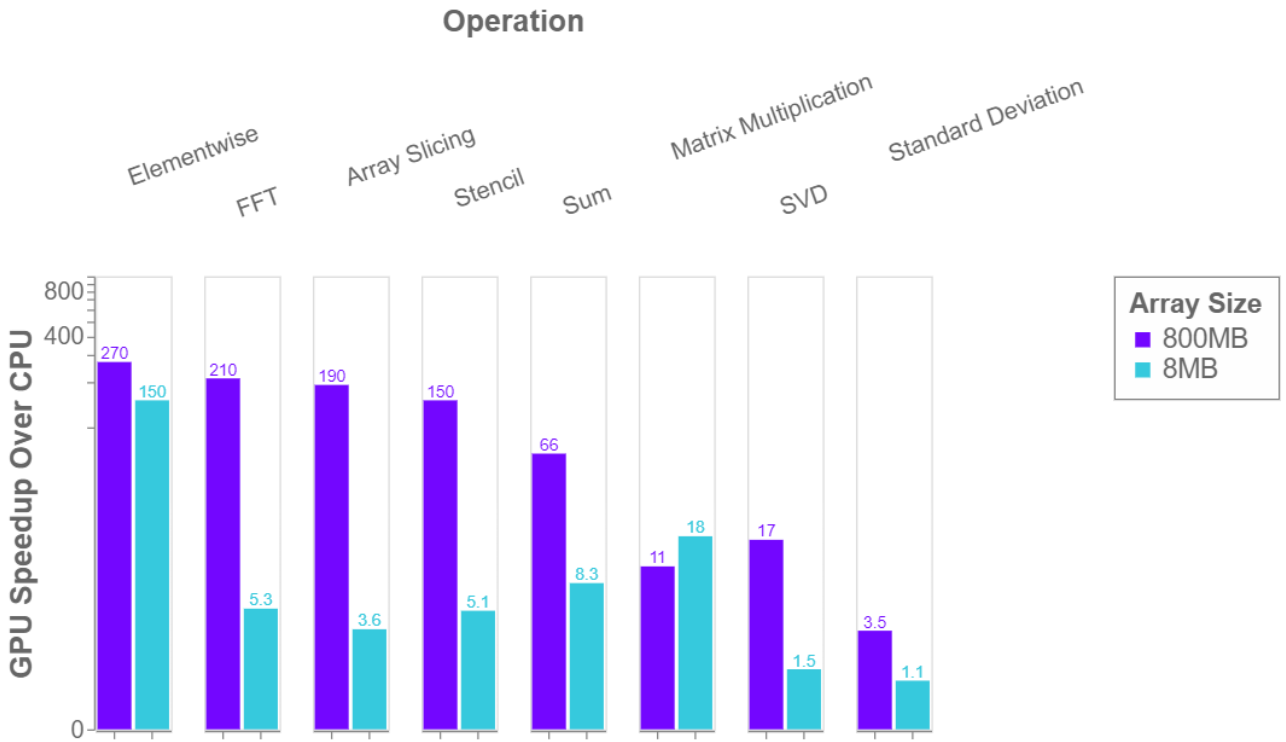




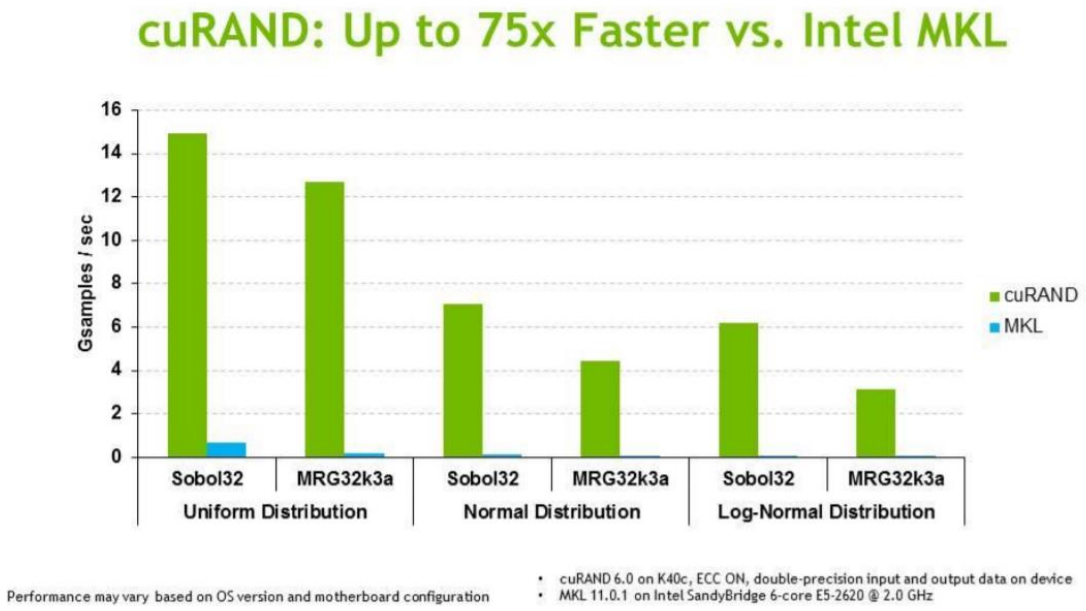
GPGPU

General Purpose

Graphics Processing Unit

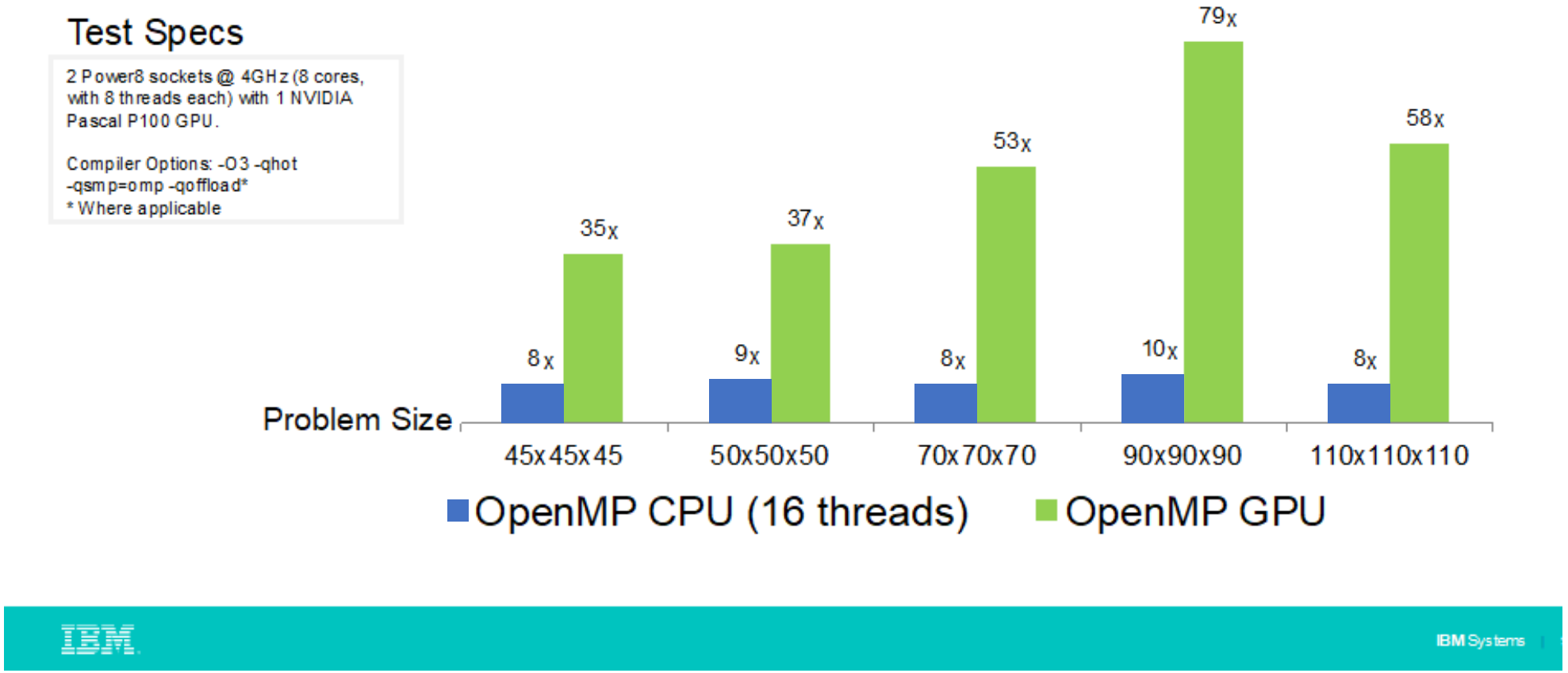


Single GPU using CuPy library
(GPU optimized library)
compared to numpy



Comparison between cuRAND
library (GPU optimize) and
MKL library (CPU optimize)

Source: https://blog.dask.org/2019/06/27/single-gpu-cupy-benchmarks
http://www.eurorisksystems.com/documents/speed_up_of_numeric_calculations_using_GPU.pdf

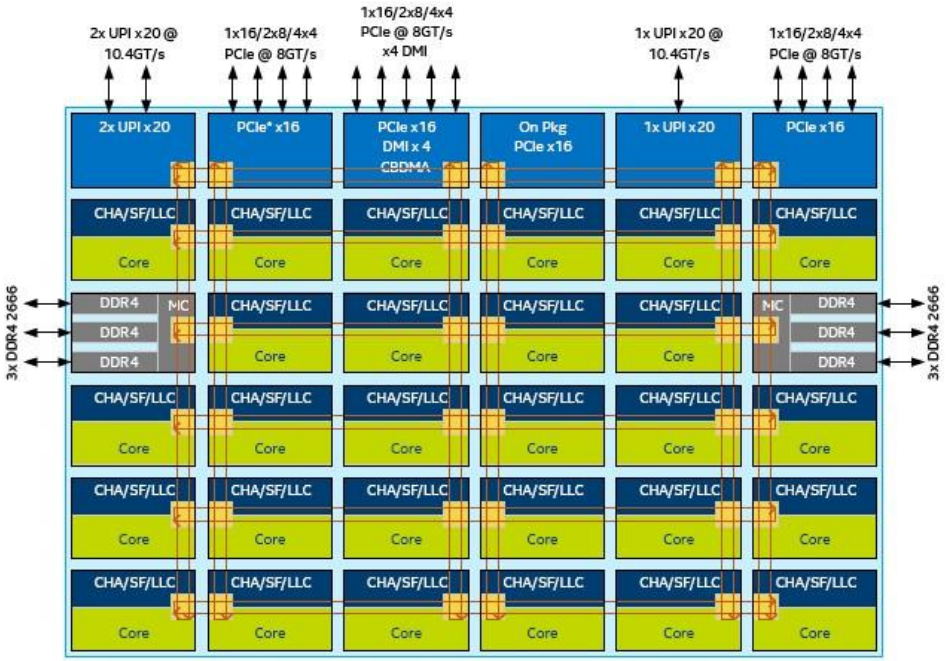


LULESH benchmark changing one single OMP directive.

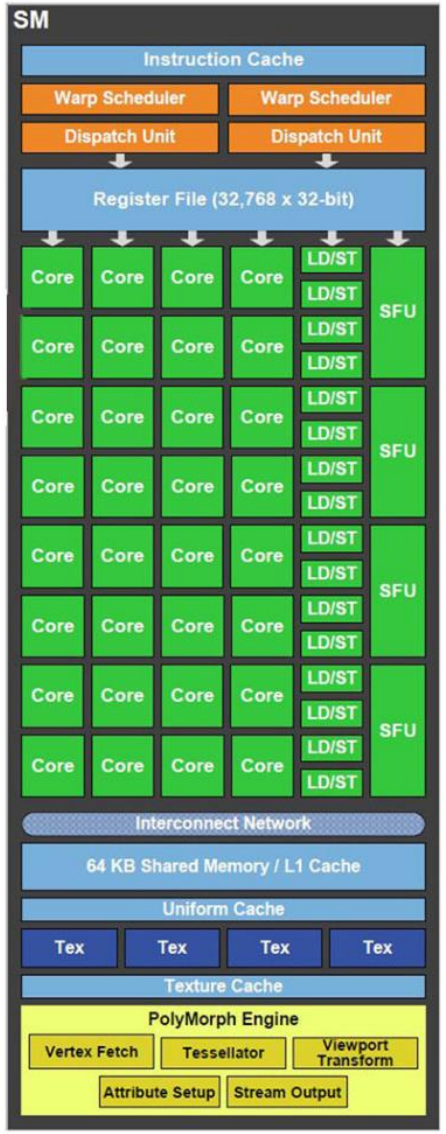
Source: https://www.openmp.org/updates/openmp-accelerator-support-gpus/

Streaming Multiprocessor

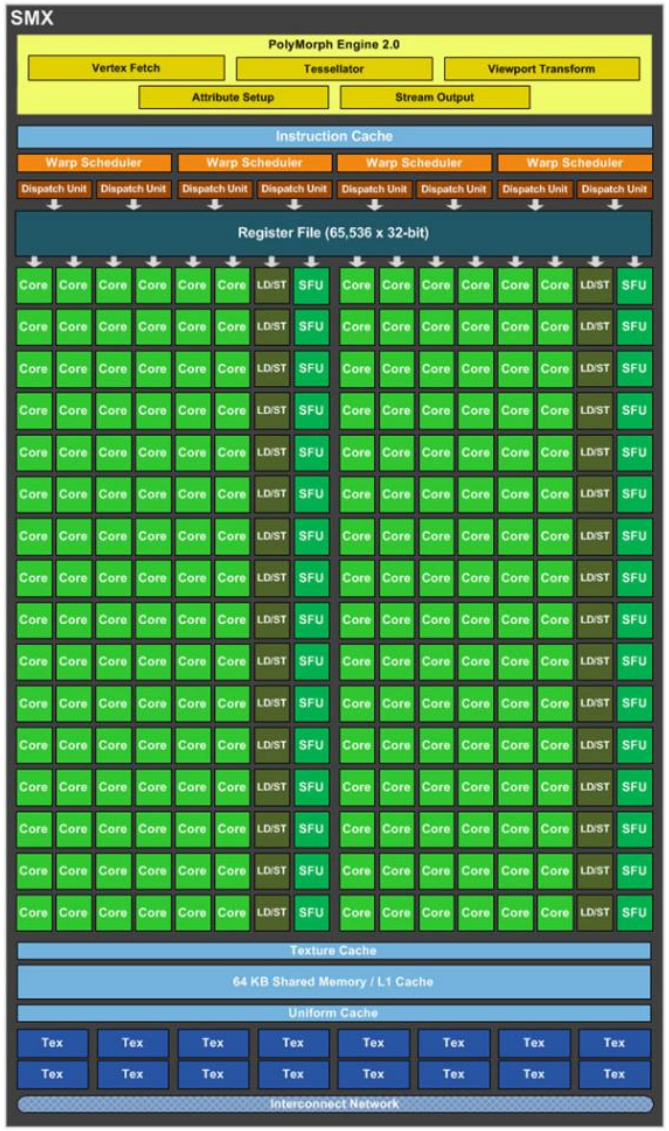
Intel Skylake (2017)



Tesla SM unit (2007)



Fermi SMX unit (2011)



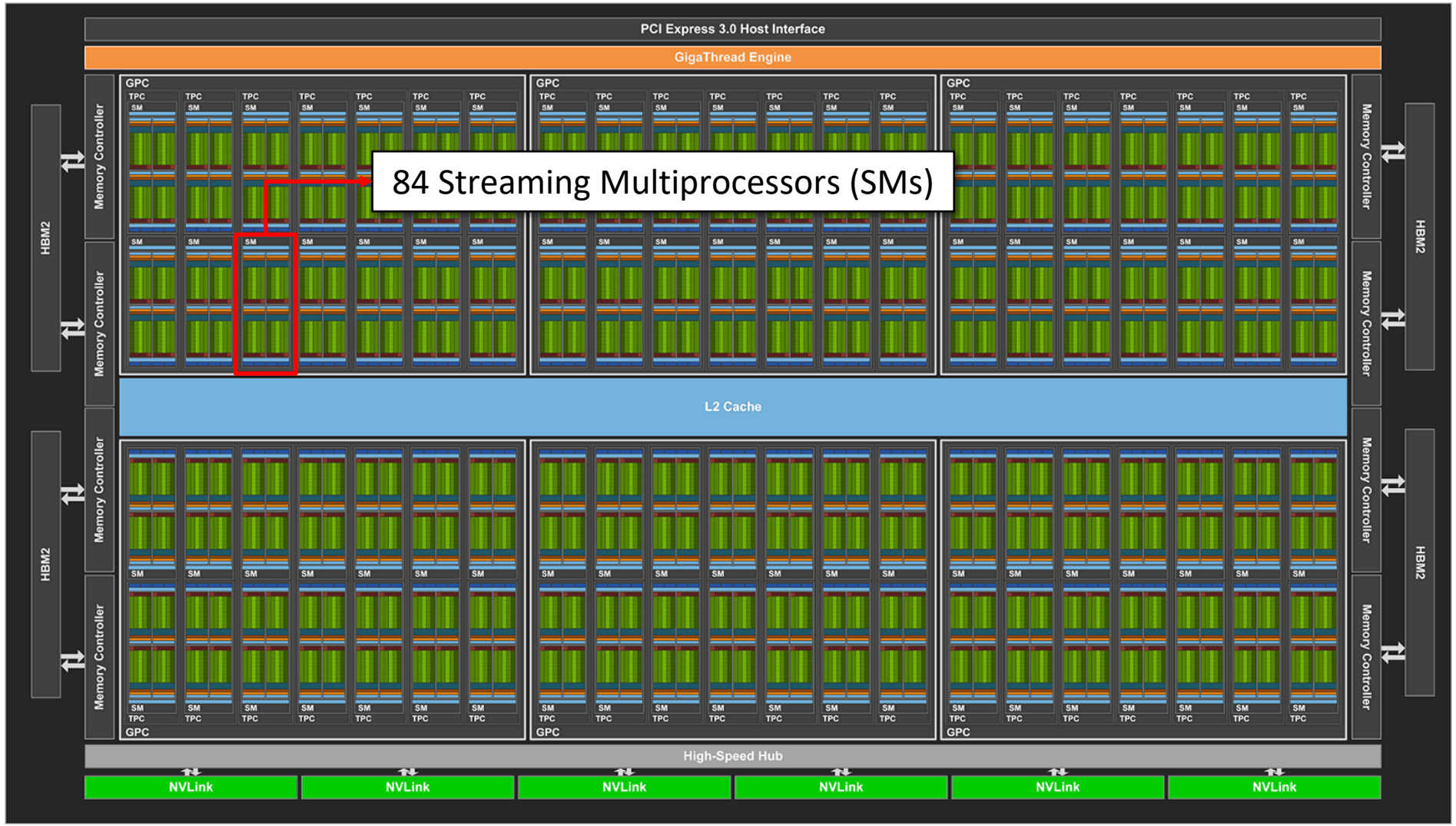
CPU	GPU
Large memory	Relatively small memory
Each core has its own independent control logic Allows independent execution	Groups of cores share control logic Saves space & power
Coherent caches between cores	Shared cache & sync only within groups
Tuned for serial execution of independent work MIMD	Tuned for parallel execution of the same work SIMD
Multiple independent threads	Threads work in lockstep (warp) within groups
It has branch prediction	It serializes codes with branches
Memory latency hidden by cache & prefetching Requires regular data access patterns	Memory latency hidden by scheduling stalled threads Requires 1000s of concurrent threads
Hyperthreading & Vectorization	None
Out of order execution	None

Mythbusters - Youtube

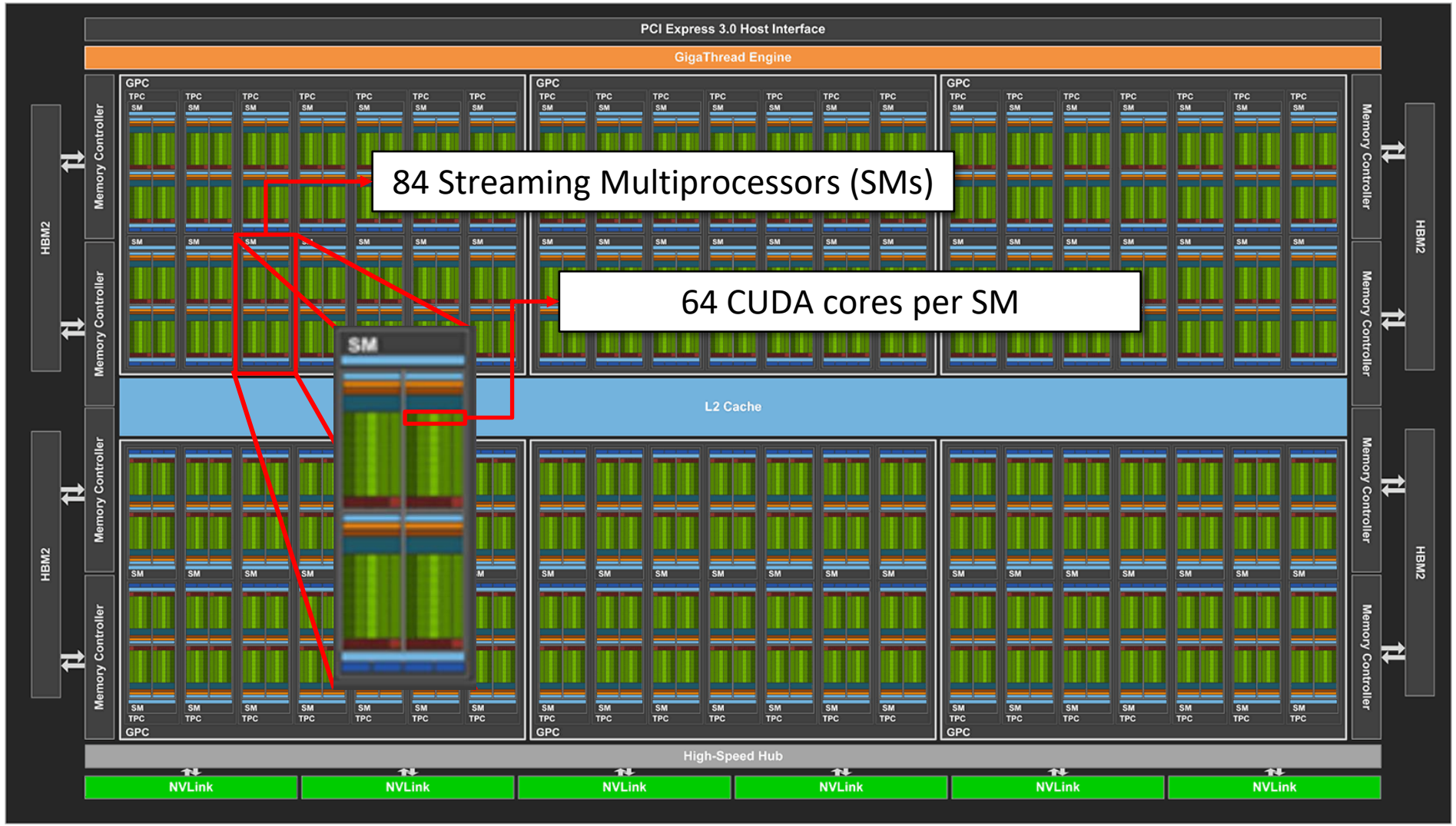
Volta (2017)


PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
L2 Cache
Memory Controller
HBM2
High-Speed Hub
NVLink

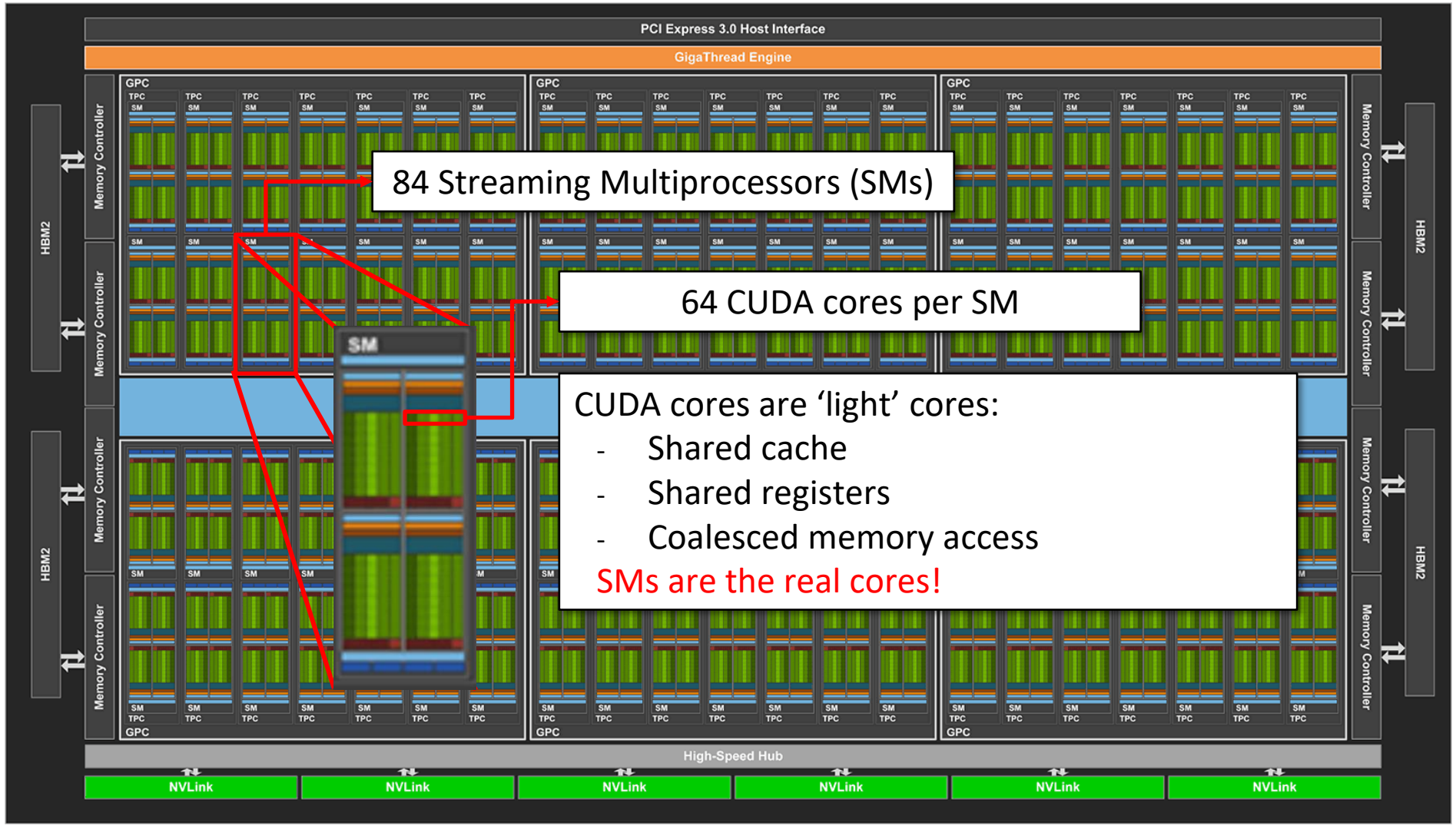
Volta (2017)


PCI Express 3.0 Host Interface
GigaThread Engine
84 Streaming Multiprocessors (SMs)
GPC
TPC
SM
Memory Controller
HBM2
L2 Cache
High-Speed Hub
NVLink

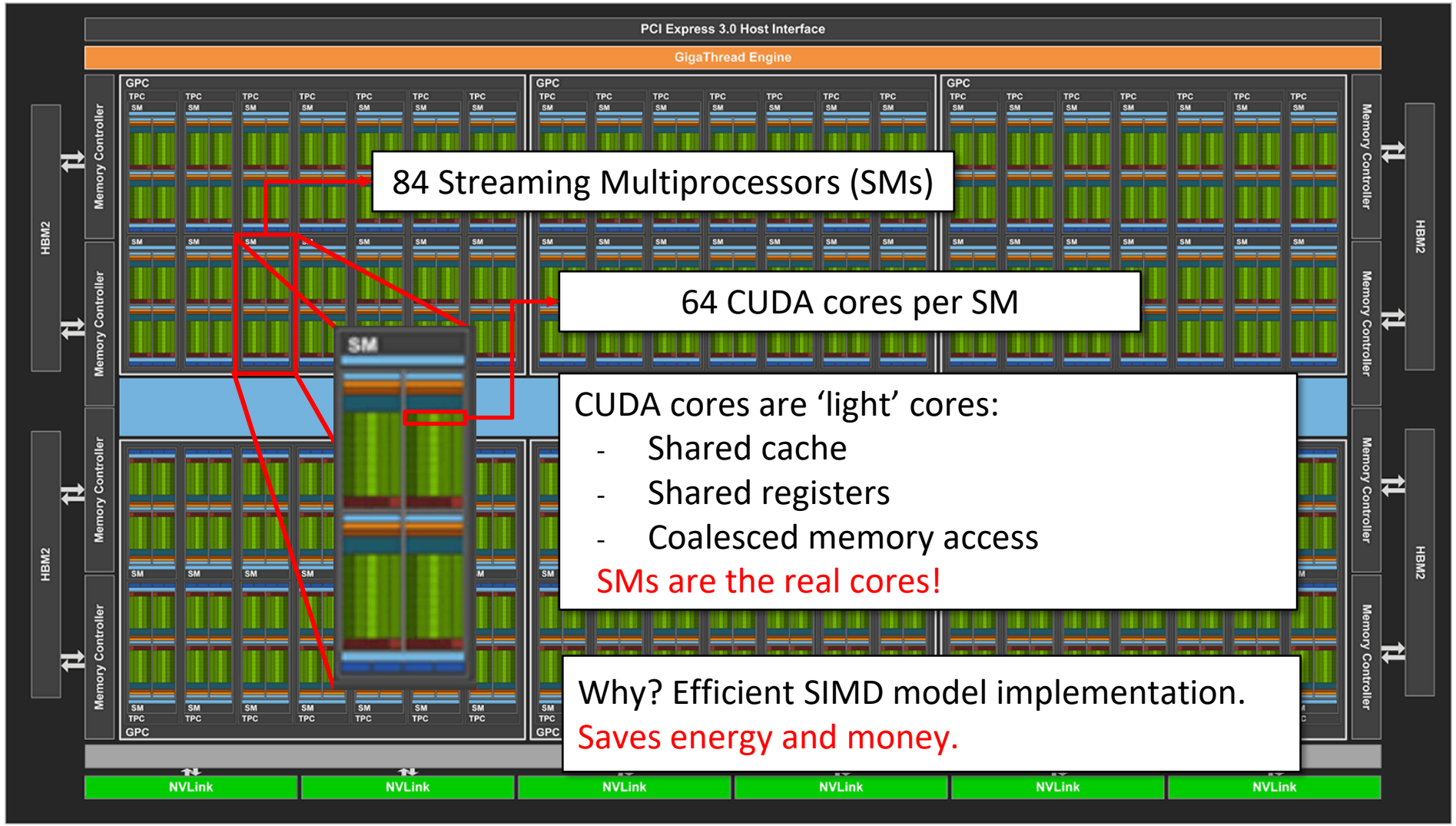
Volta (2017)


PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
84 Streaming Multiprocessors (SMs)
64 CUDA cores per SM
L2 Cache
Memory Controller
HBM2
High-Speed Hub
NVLink

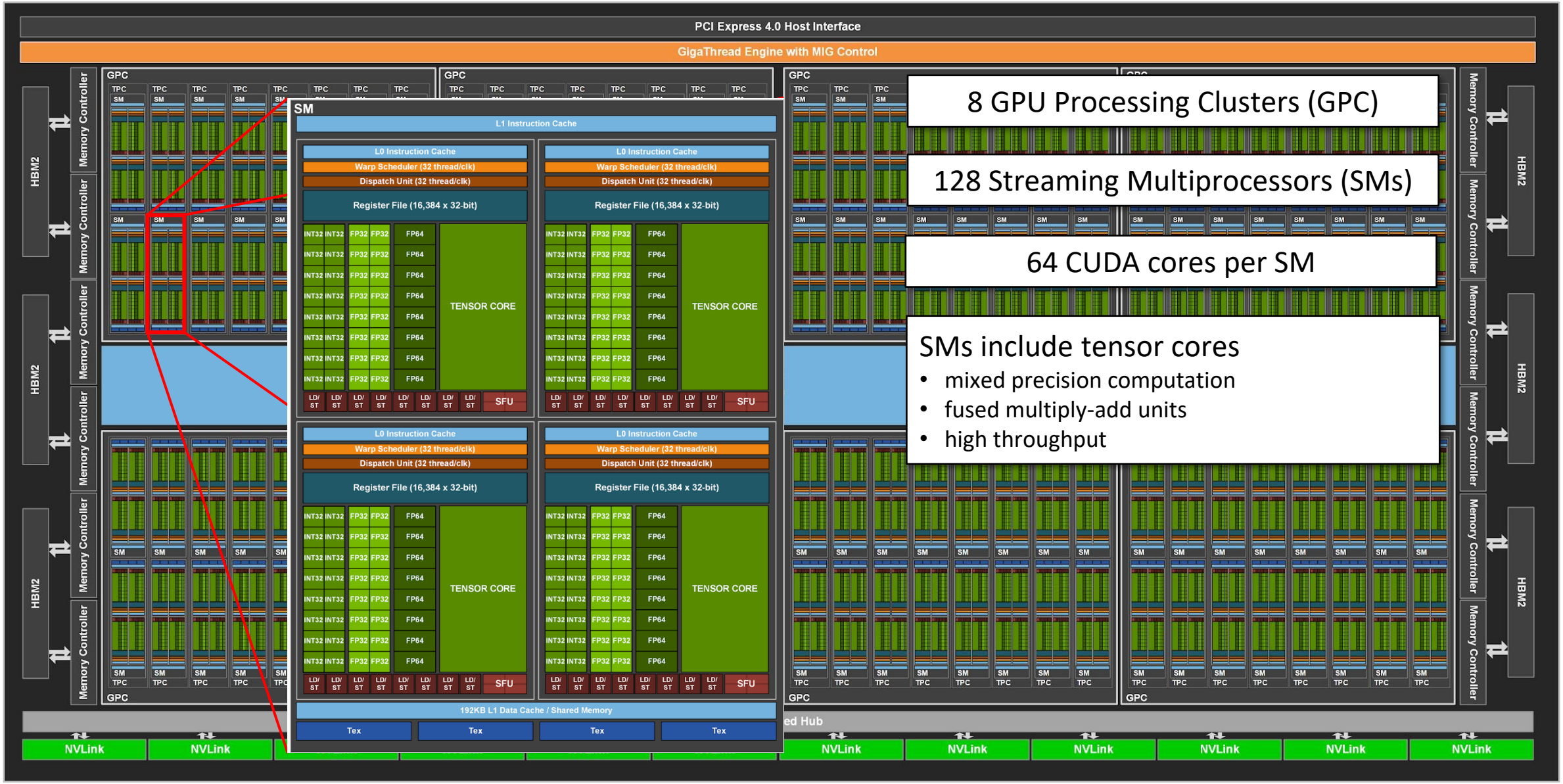
Volta (2017)


PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
Memory Controller
HBM2
84 Streaming Multiprocessors (SMs)
64 CUDA cores per SM
CUDA cores are 'light' cores:
- Shared cache
- Shared registers
- Coalesced memory access
SMs are the real cores!
High-Speed Hub
NVLink

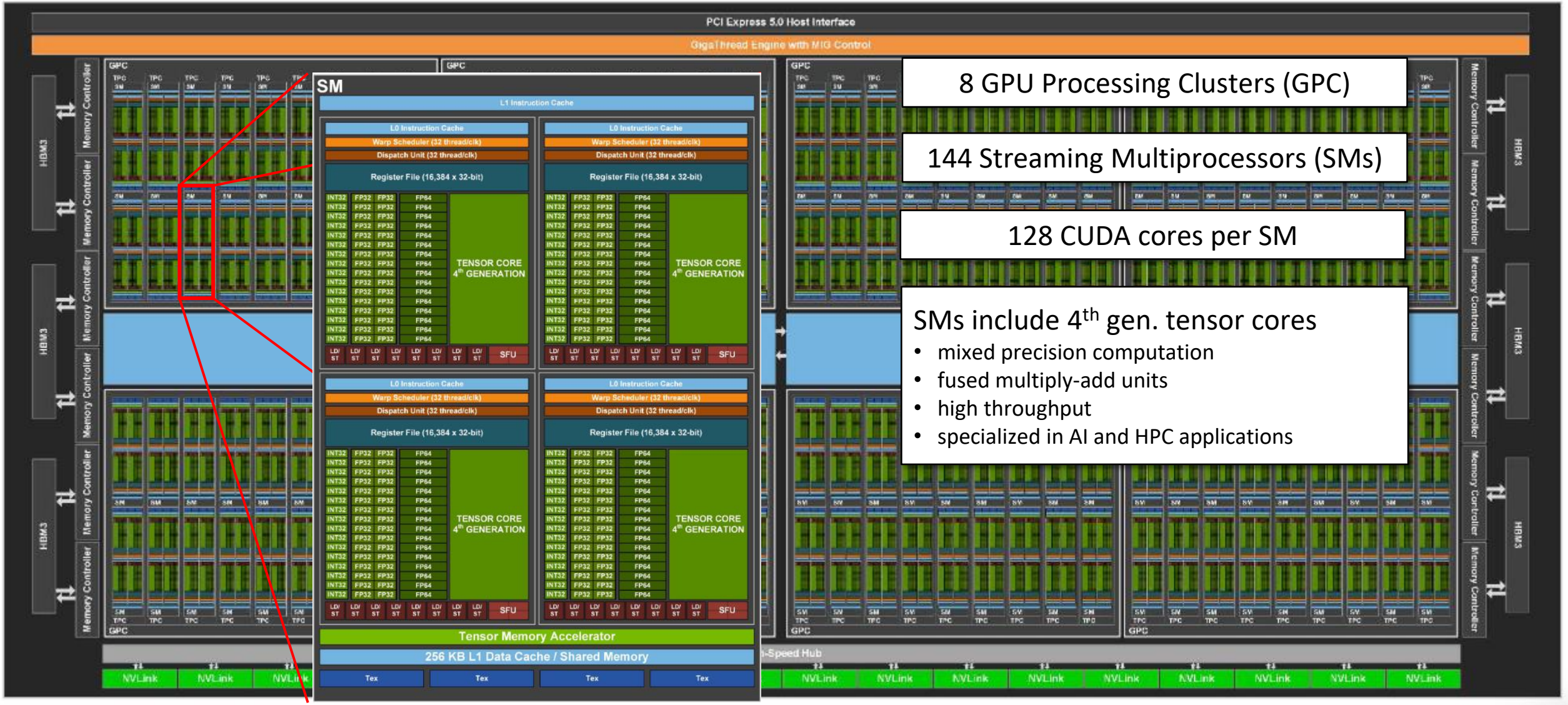
Volta (2017)


PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
Memory Controller
HBM2
NVLink
84 Streaming Multiprocessors (SMs)
64 CUDA cores per SM
CUDA cores are 'light' cores:
- Shared cache
- Shared registers
- Coalesced memory access
SMs are the real cores!
Why? Efficient SIMD model implementation.
Saves energy and money.

Ampere (2020)

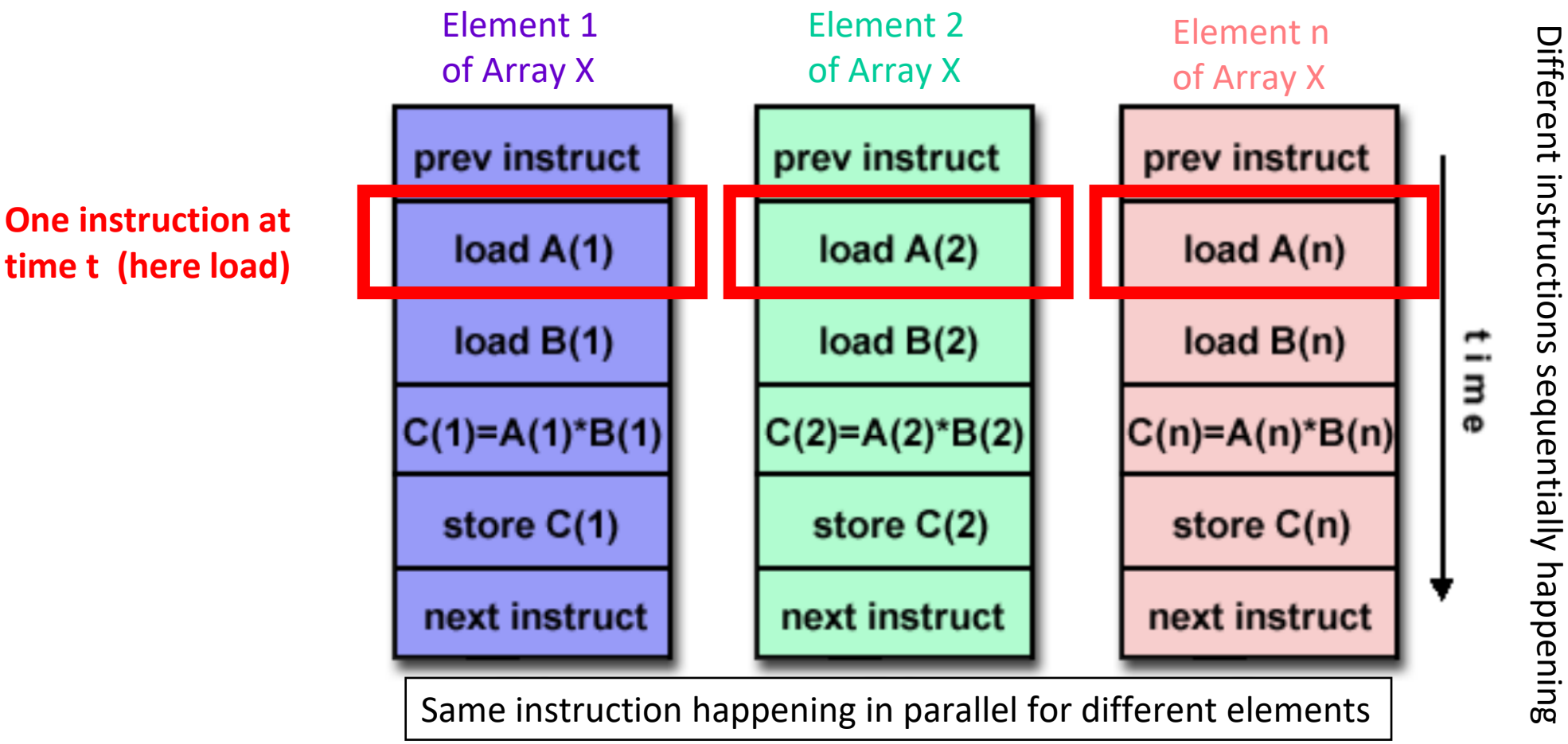

PCI Express 4.0 Host Interface
GigaThread Engine with MIG Control
GPC
TPC
SM
Memory Controller
HBM2
NVLink
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE
LD/ST
SFU
192KB L1 Data Cache / Shared Memory
Tex
8 GPU Processing Clusters (GPC)
128 Streaming Multiprocessors (SMs)
64 CUDA cores per SM
SMs include tensor cores
• mixed precision computation
• fused multiply-add units
• high throughput

Hopper (2022)


PCI Express 5.0 Host Interface
Gigathread Engine with MIG Control
GPC
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 4th GENERATION
LD/ST
SFU
Tensor Memory Accelerator
256 KB L1 Data Cache / Shared Memory
Tex
Memory Controller
HBM3
NVLink
8 GPU Processing Clusters (GPC)
144 Streaming Multiprocessors (SMs)
128 CUDA cores per SM
SMs include 4th gen. tensor cores
• mixed precision computation
• fused multiply-add units
• high throughput
• specialized in AI and HPC applications

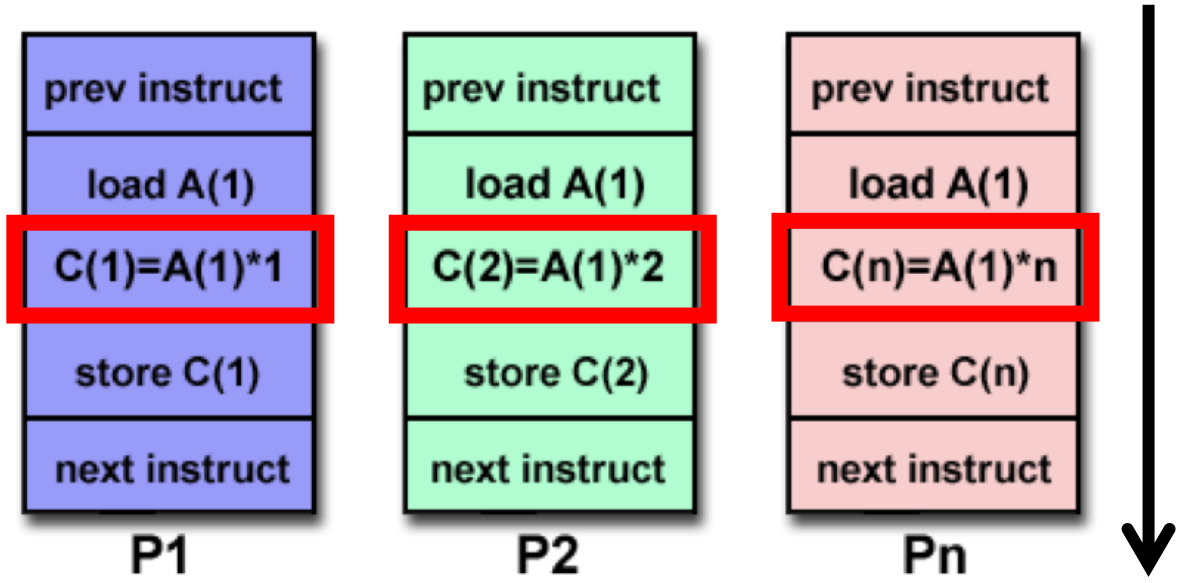
GPUs are SIMD oriented architectures

SIMD : Single Instruction Multiple Data



Source: https://www.cvg.ethz.ch/teaching/2011spring/gpgpu/Lec1-intro.pdf

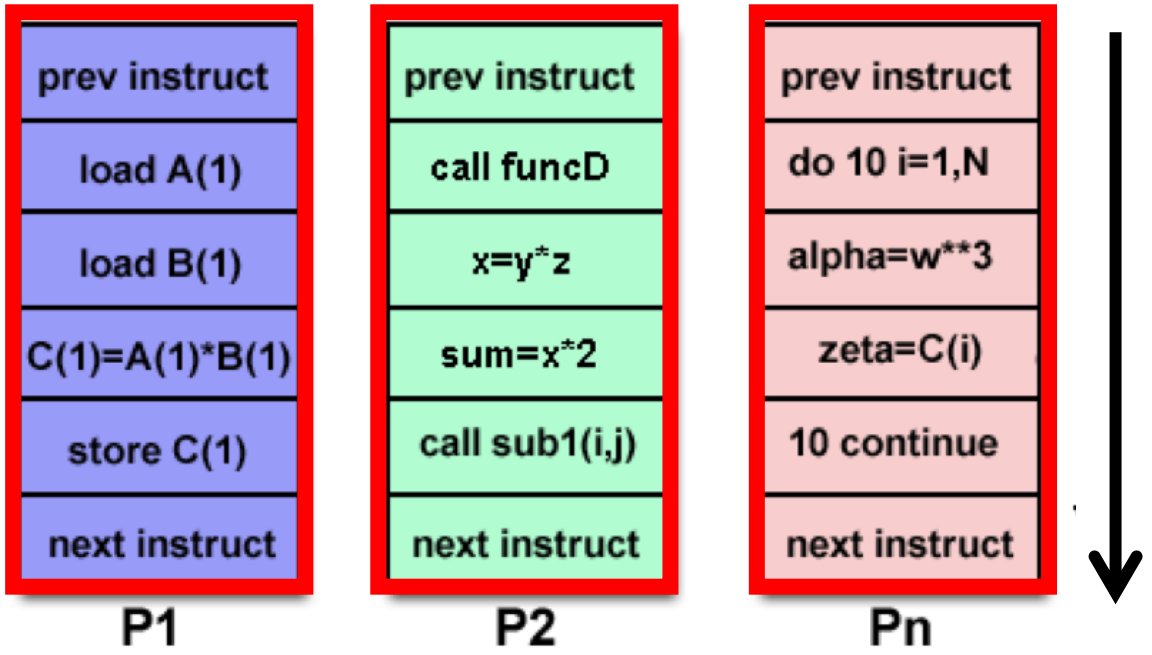
MISD : Multiple Instructions Single Data



GPUs are NOT MISD/MIMD **oriented** architectures

CPU can do MIMD & SIMD
GPU can do SIMD only

MIMD : Multiple Instructions Multiple Data



Source: https://www.cvg.ethz.ch/teaching/2011spring/gpgpu/Lec1-intro.pdf

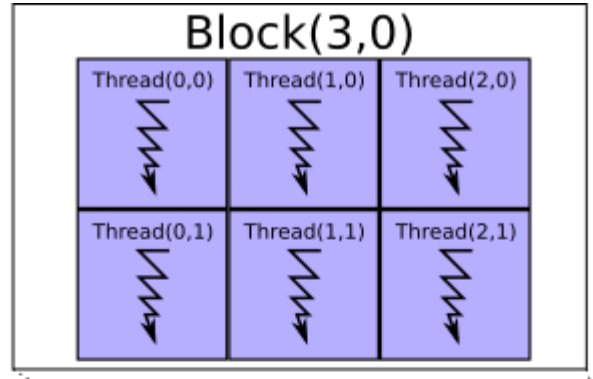
Mapping the hardware architecture into the programming architecture

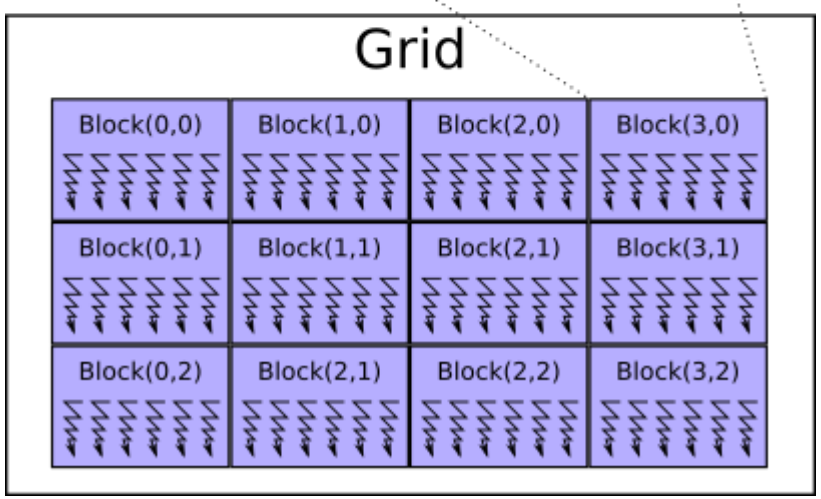
As a GPU programmer you will :

- define an instruction applied on an element of an array : **thread**.
- run many threads in parallel within a shared memory environment : **blocks**.

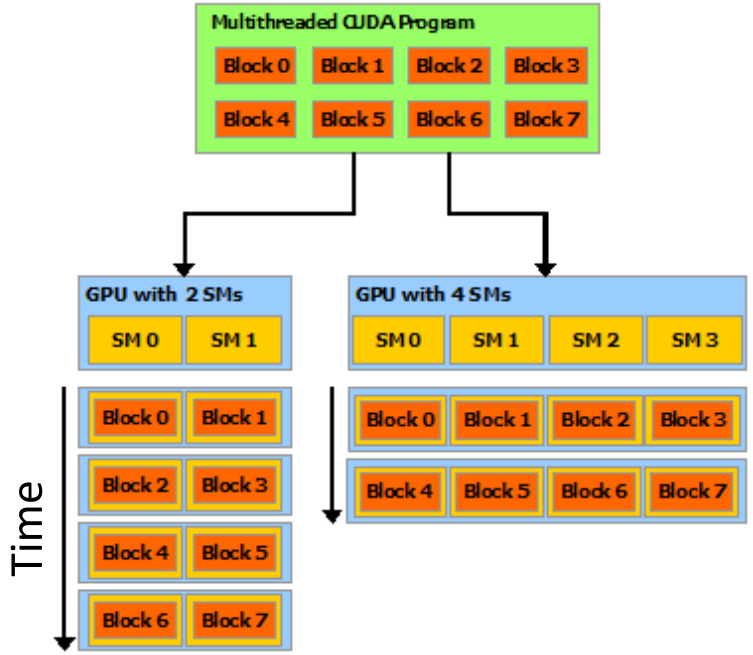
  (block = doing the same operation in parallel on a subset of elements of your array).

- manage many blocks in parallel (many blocks on the same SM, or many SM), or sequentially if your array exceed your GPU capacity.





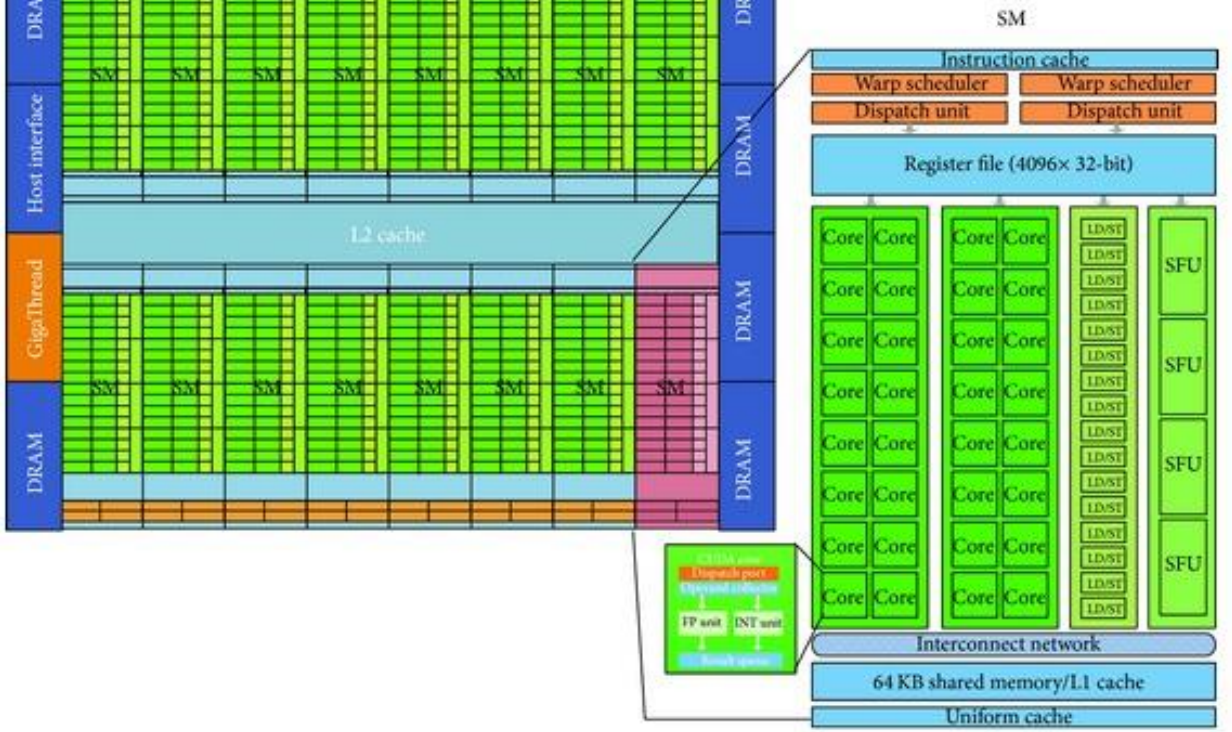
Warp: Minimum amount of threads (32) that will be executed simultaneously per SM



Source: http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf

How does it look like on a real architecture?

Nvidia Fermi GPU (2011)

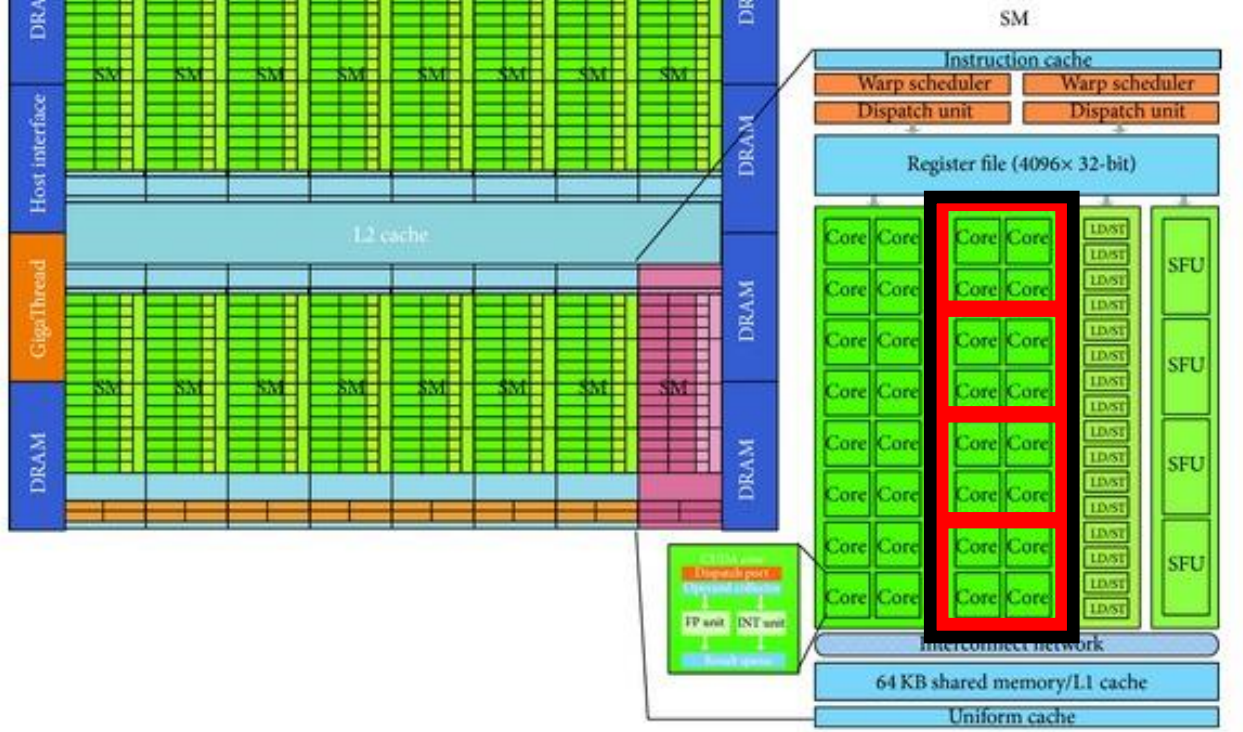


Here it is not a core, it is a "CUDA core" so not directly comparable to the CPU core

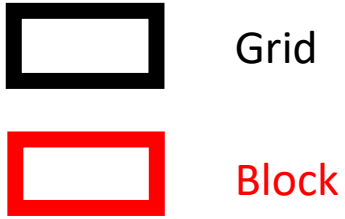
Source: 3D Data Denoising via Nonlocal Means Filter by Using Parallel GPU Strategies
DOI: 10.1155/2014/523862

How does it look like on a real architecture?

Nvidia Fermi GPU (2011)



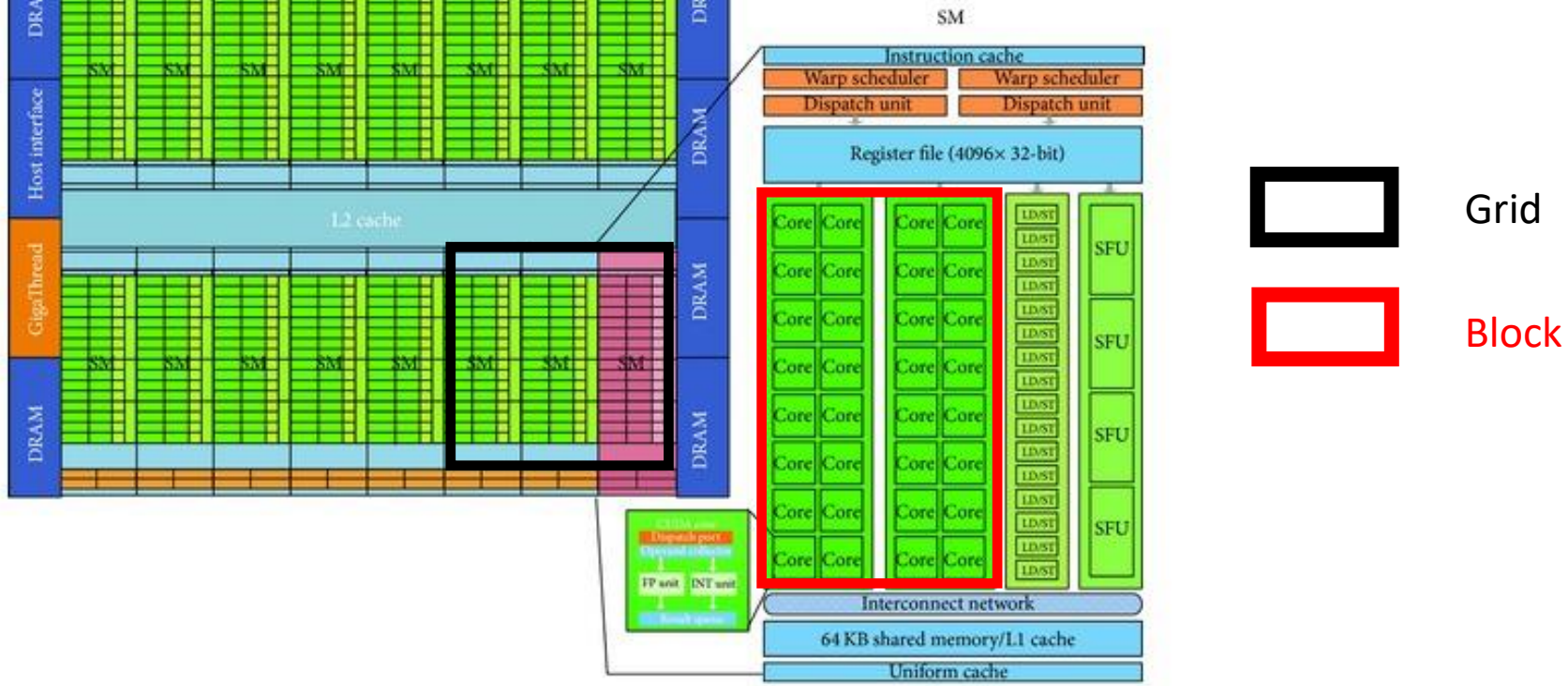
Here it is not a core, it is a "CUDA core" so not directly comparable to the CPU core



Source: 3D Data Denoising via Nonlocal Means Filter by Using Parallel GPU Strategies
DOI: 10.1155/2014/523862

How does it look like on a real architecture?

Nvidia Fermi GPU (2011)



Here it is not a core, it is a "CUDA core" so not directly comparable to the CPU core

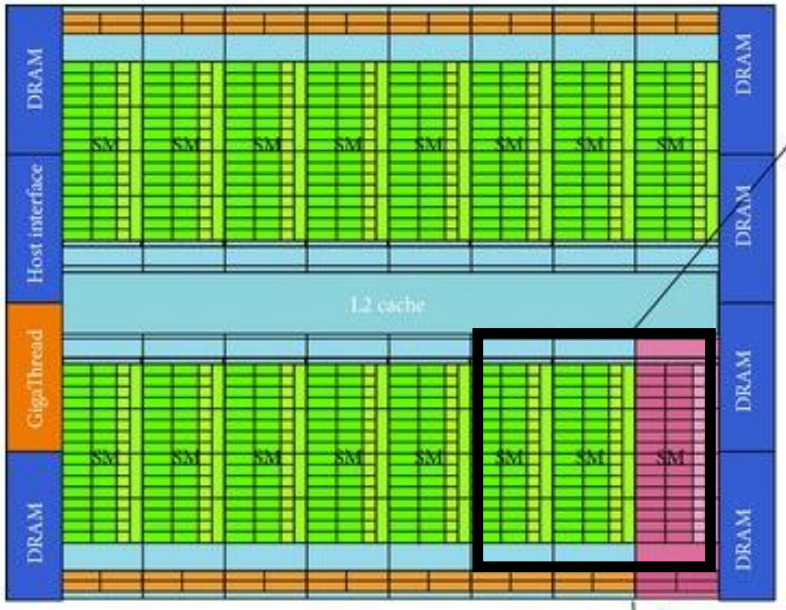
Grid

Block

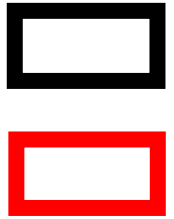
Source: 3D Data Denoising via Nonlocal Means Filter by Using Parallel GPU Strategies
DOI: 10.1155/2014/523862

How does it look like on a real architecture?

Nvidia Fermi GPU (2011)



Here it is not a core, it is a "CUDA core" so not directly comparable to the CPU core



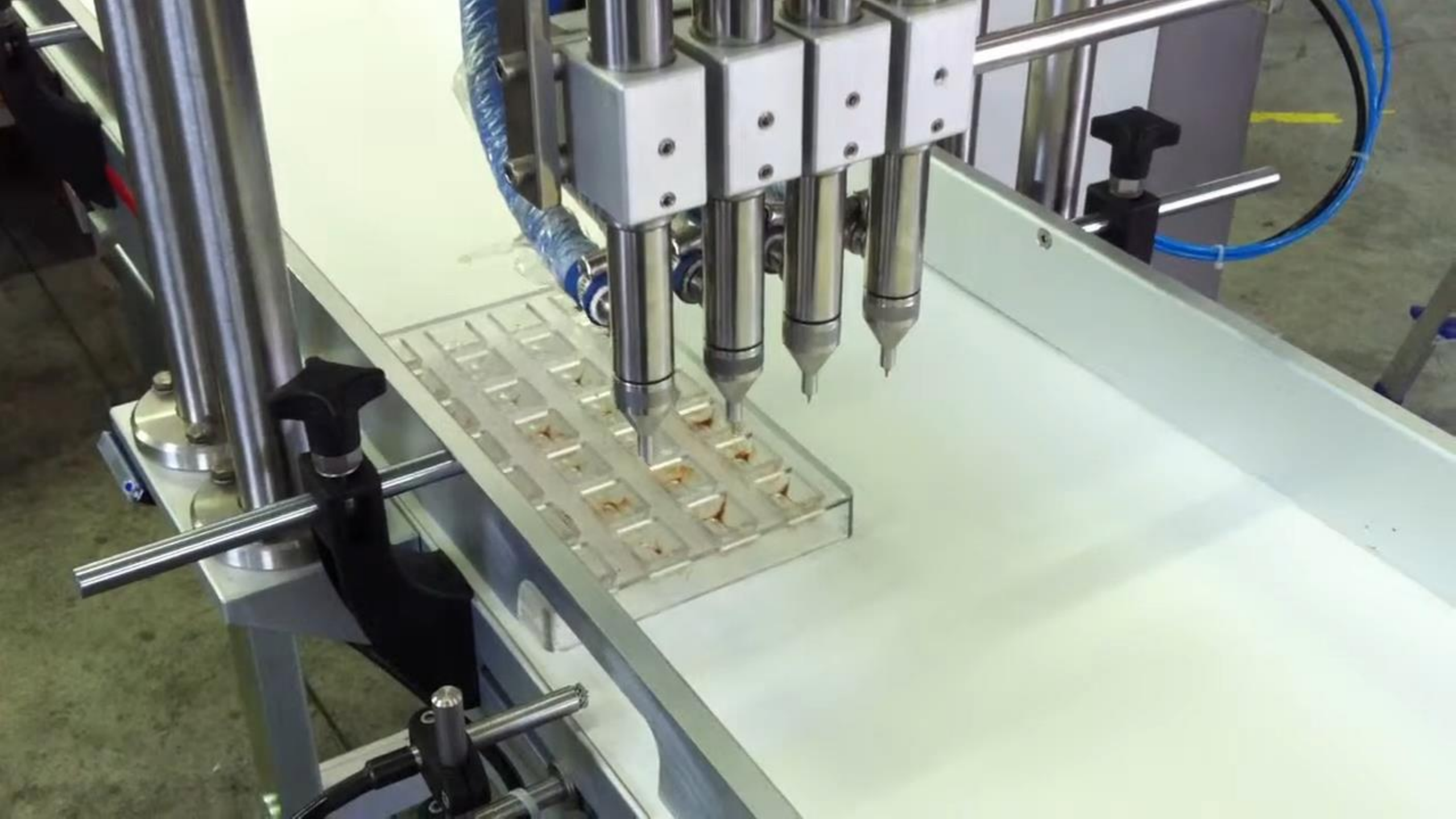
Grid

Block

A grid can encompass many SM but a block can't use more CUDA-cores that those in a SM since a block is a shared memory space.

You can also set the number of threads per block, but this number is not easily linkable to the number of CUDA cores.
Yet a **block size (number of threads per block) should be a multiple of 32**

Source: 3D Data Denoising via Nonlocal Means Filter by Using Parallel GPU Strategies
DOI: 10.1155/2014/523862



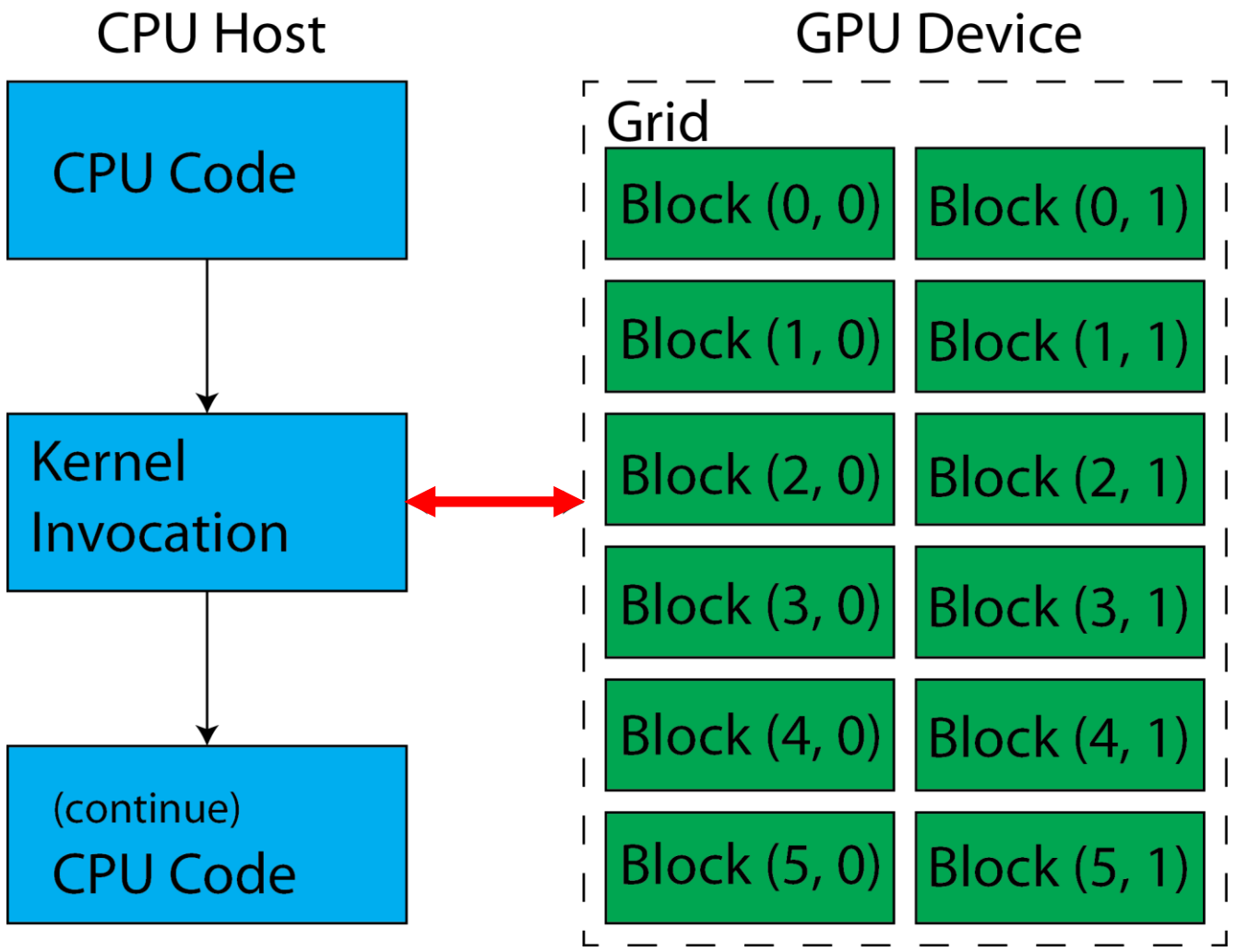
Outline

Day 1

(09:00 – 13:00)

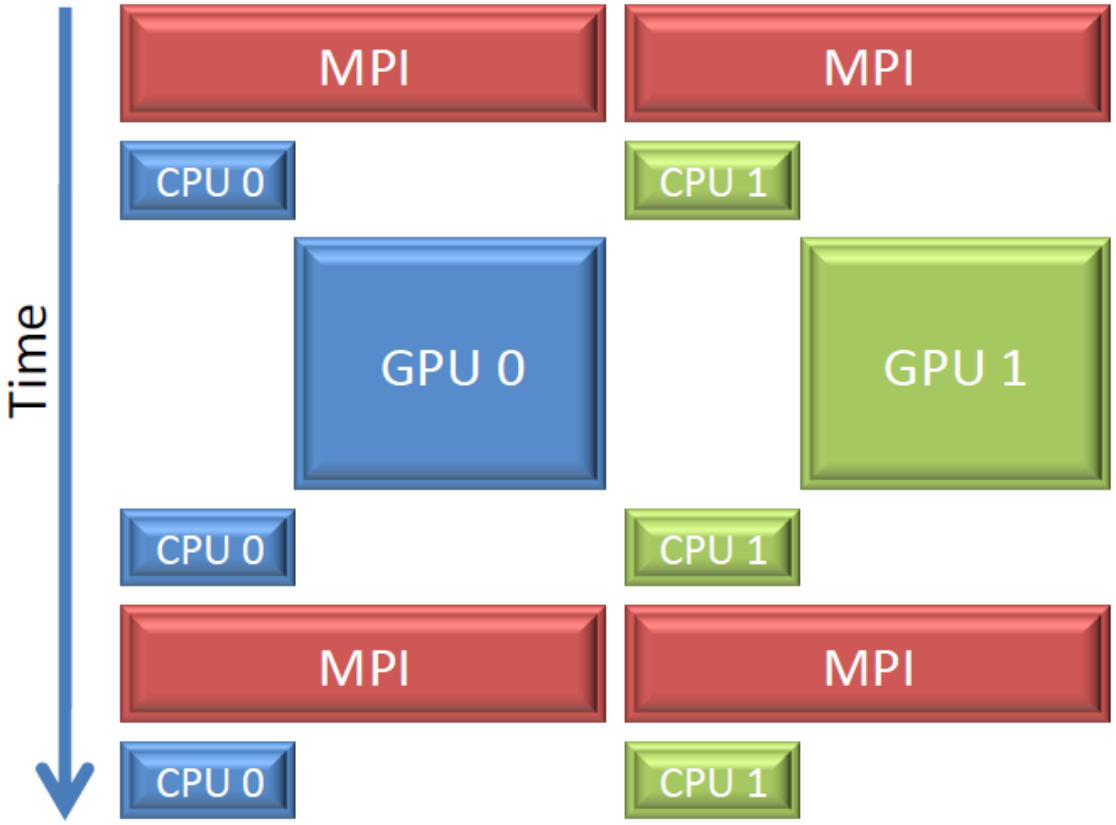
- ❑ Introduction
- ❑ Coding a CUDA kernel
- ❑ Example
  - ❑ Matrix Multiplication + Exercise
- ❑ Real world implementations
  - ❑ Genomes simulation
  - ❑ Random numbers generation
- ❑ Homework

A mapping at the core of CPU/GPU communication: the kernel



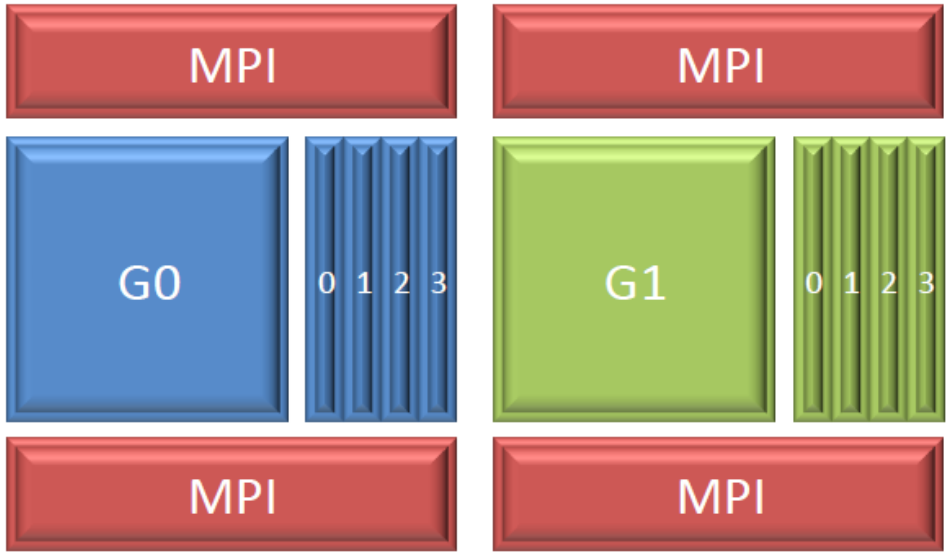
Source: https://cvw.cac.cornell.edu/GPU/structure

So-So Hybridization



- Neglects CPU
- Suffers from Amdahl's Law

Better Hybridization



- Overlap CPU/GPU work and data movement
- Even better if you can overlap MPI comms

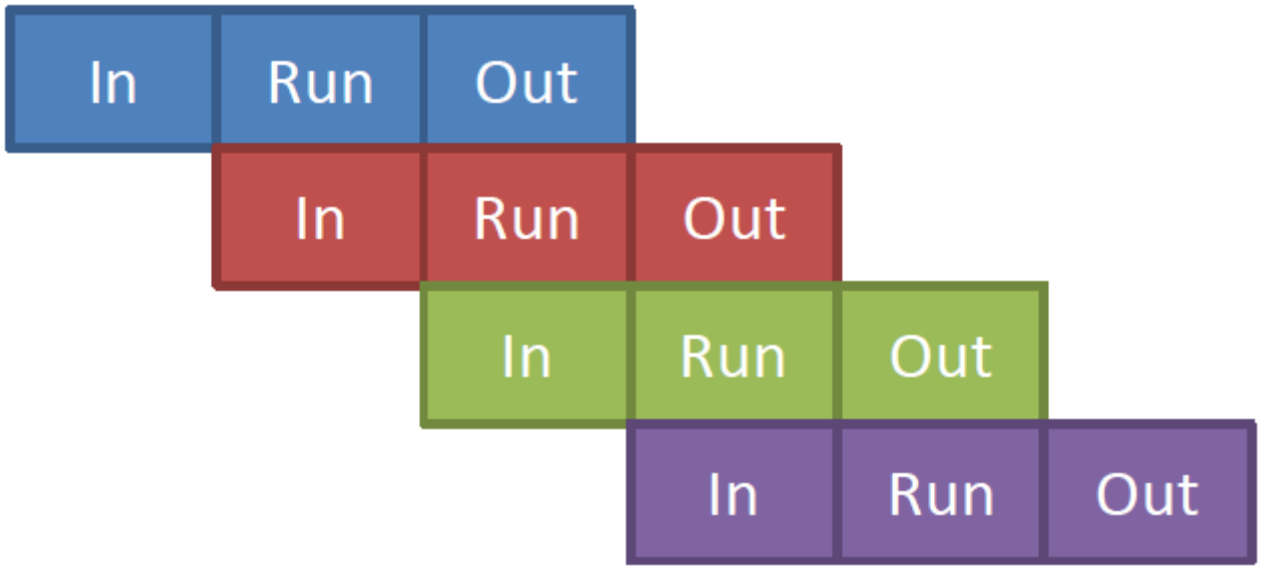
Most GPU operations are asynchronous from the CPU code

(i.e. the CPU can be busy doing other things!)

Source: Jeff Larkin NVIDIA

Synchronous execution (1 stream)

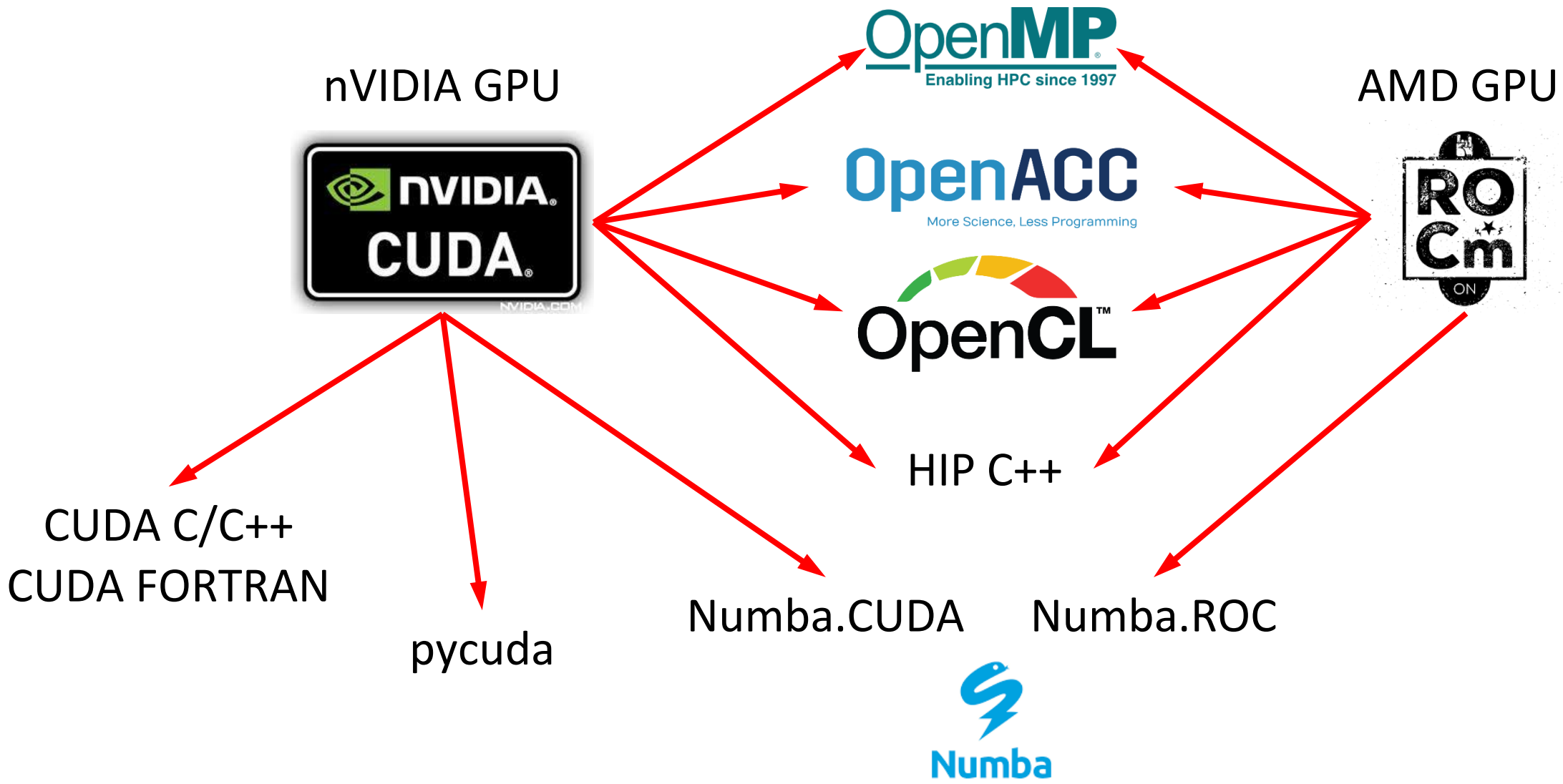
Asynchronous execution (3 streams)



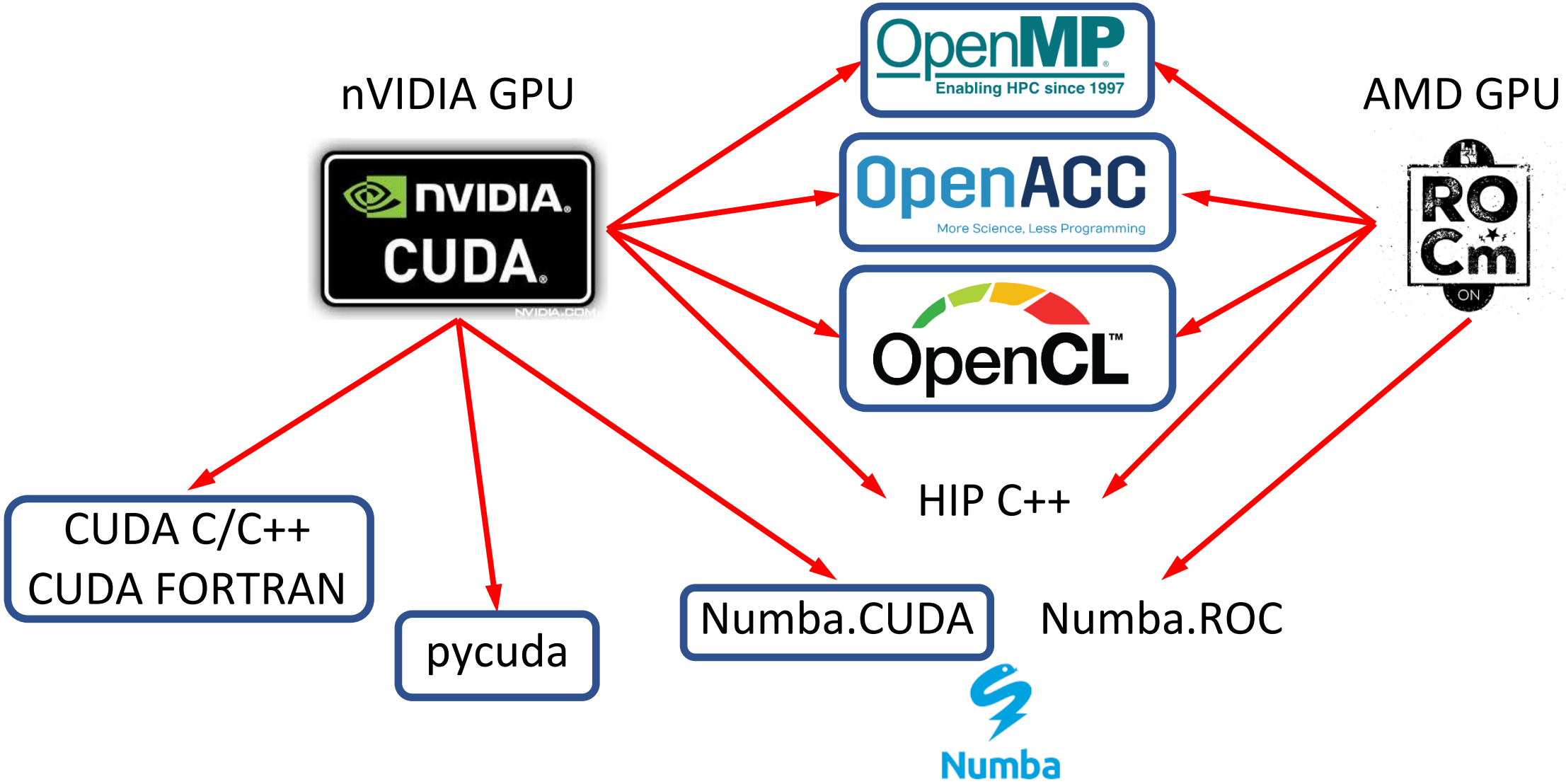
Asynchronous execution helps to hide the data transfer costs

Source: Jeff Larkin NVIDIA

Writing your kernel : GPU programming paradigms



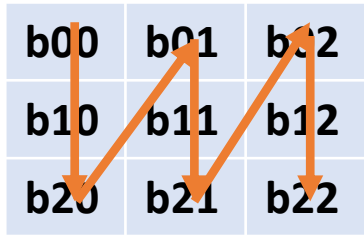
Writing your kernel : GPU programming paradigms



Sub-optimal memory access will kill the performance…

The less efficient way (column-major):

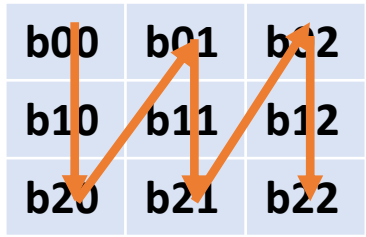
```
for(int i=0; i<n; ++i)
   for(int j=0; j<n; ++j)
      for(int k=0; k<n; ++k)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



Sub-optimal memory access will kill the performance…

The less efficient way (column-major):

```
for(int i=0; i<n; ++i)
   for(int j=0; j<n; ++j)
      for(int k=0; k<n; ++k)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



- Processors **load data into their cache** for fast reuse
- They always load a bunch of data at the same time
- They do **branch-prediction** and **pre-fetching**!

However here we are not helping …

Processor cache

Cache Misses: 0

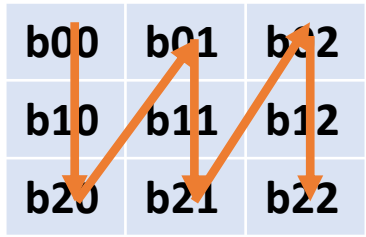
a00	a01	a02	a10	a11	a12	a20	a21	a22
b00	b01	b02	b10	b11	b12	b20	b21	b22
c00	c01	c02	c10	c11	c12	c20	c21	c22

i	j	k
0	0	0

Sub-optimal memory access will kill the performance…

The less efficient way (column-major):

```
for(int i=0; i<n; ++i)
   for(int j=0; j<n; ++j)
      for(int k=0; k<n; ++k)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



- Processors **load data into their cache** for fast reuse
- They always load a bunch of data at the same time
- They do **branch-prediction** and **pre-fetching**!

However here we are not helping …

Processor cache

Cache Misses: 1

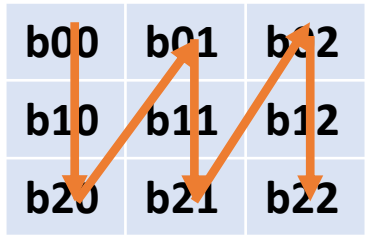
a00	a01	a02	a10	a11	a12	a20	a21	a22
b00	b01	b02	b10	b11	b12	b20	b21	b22
c00	c01	c02	c10	c11	c12	c20	c21	c22

i	j	k
0	0	1

Sub-optimal memory access will kill the performance…

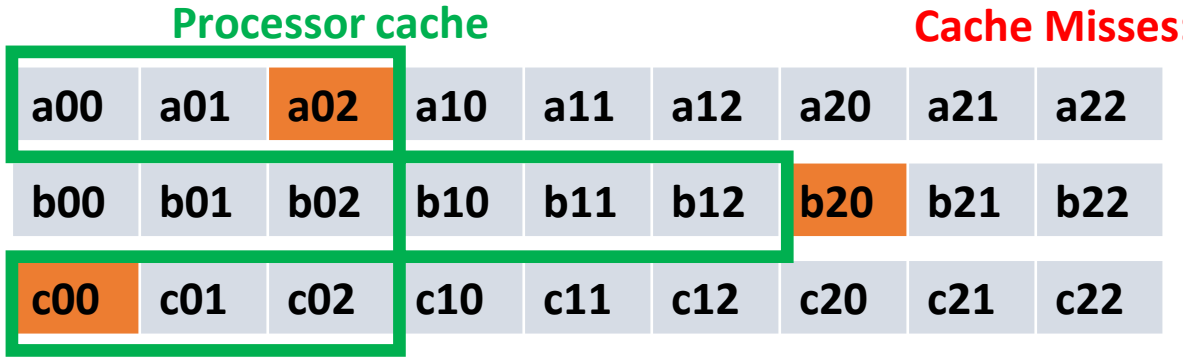
The less efficient way (column-major):

```
for(int i=0; i<n; ++i)
   for(int j=0; j<n; ++j)
      for(int k=0; k<n; ++k)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



- Processors **load data into their cache** for fast reuse
- They always load a bunch of data at the same time
- They do **branch-prediction** and **pre-fetching**!

However here we are not helping …

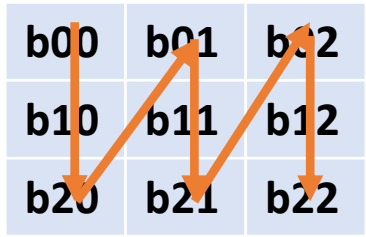


i	j	k
0	0	2

Sub-optimal memory access will kill the performance…

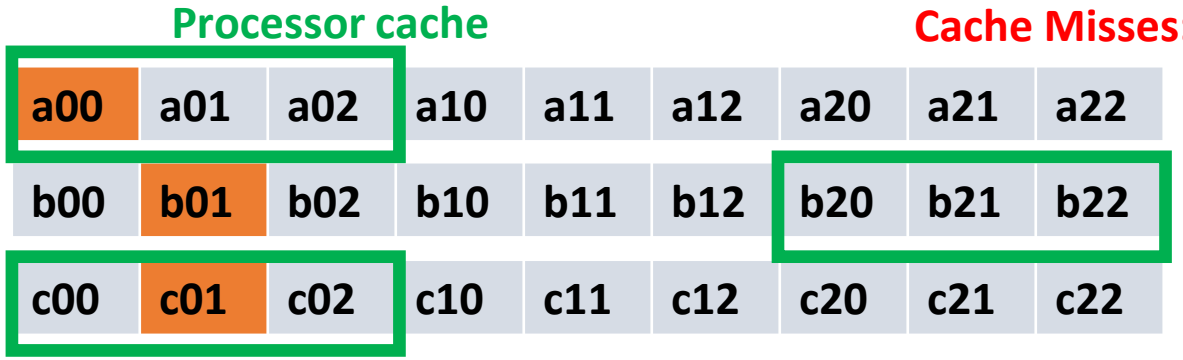
The less efficient way (column-major):

```
for(int i=0; i<n; ++i)
   for(int j=0; j<n; ++j)
      for(int k=0; k<n; ++k)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



- Processors **load data into their cache** for fast reuse
- They always load a bunch of data at the same time
- They do **branch-prediction** and **pre-fetching**!

However here we are not helping …

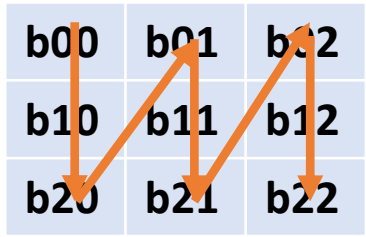


i	j	k
0	1	0

Sub-optimal memory access will kill the performance…

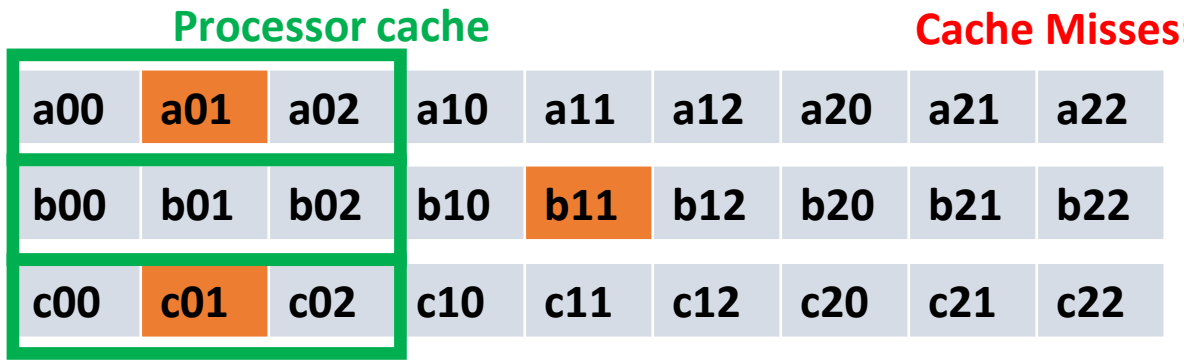
The less efficient way (column-major):

```
for(int i=0; i<n; ++i)
   for(int j=0; j<n; ++j)
      for(int k=0; k<n; ++k)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



- Processors **load data into their cache** for fast reuse
- They always load a bunch of data at the same time
- They do **branch-prediction** and **pre-fetching**!

However here we are not helping …

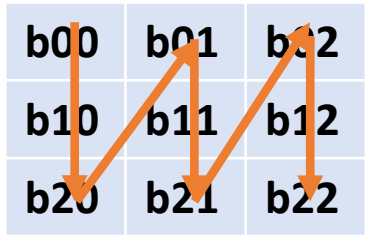


i	j	k
0	1	1

Sub-optimal memory access will kill the performance…

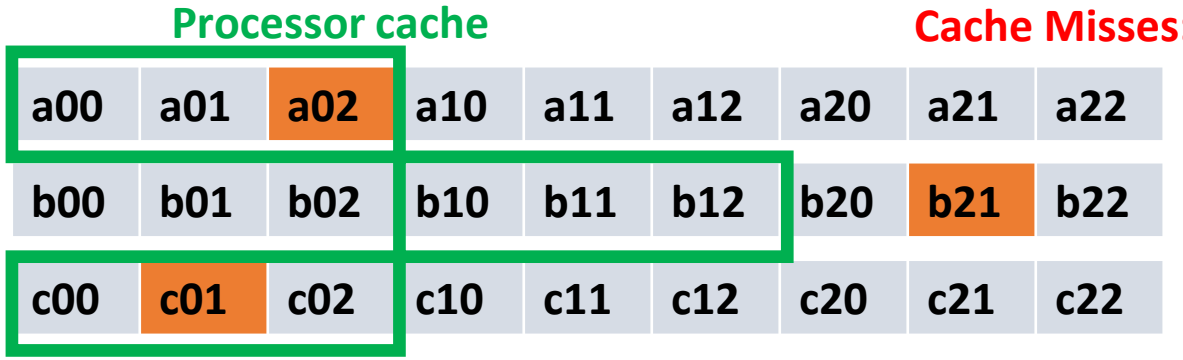
The less efficient way (column-major):

```
for(int i=0; i<n; ++i)
   for(int j=0; j<n; ++j)
      for(int k=0; k<n; ++k)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



- Processors **load data into their cache** for fast reuse
- They always load a bunch of data at the same time
- They do **branch-prediction** and **pre-fetching**!

However here we are not helping …

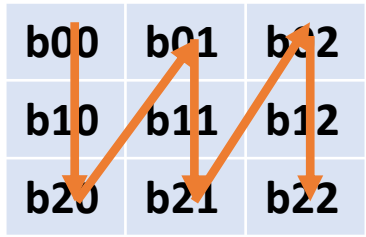


i	j	k
0	1	2

Sub-optimal memory access will kill the performance…

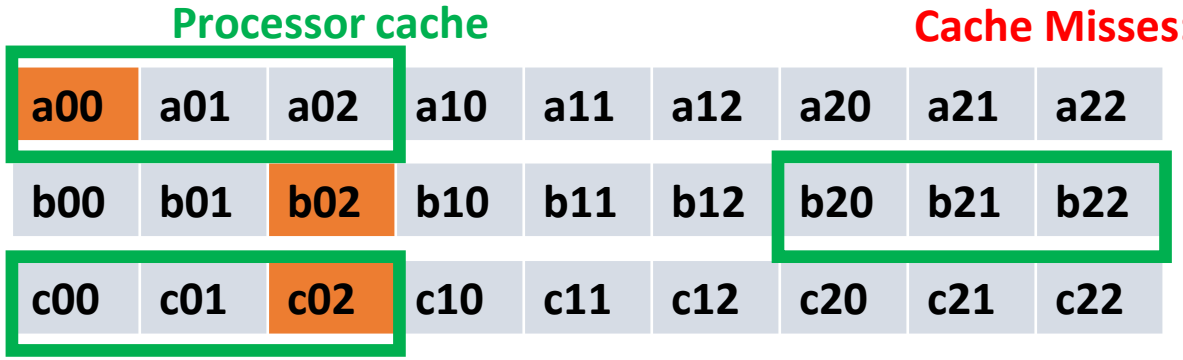
The less efficient way (column-major):

```
for(int i=0; i<n; ++i)
   for(int j=0; j<n; ++j)
      for(int k=0; k<n; ++k)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



- Processors **load data into their cache** for fast reuse
- They always load a bunch of data at the same time
- They do **branch-prediction** and **pre-fetching**!

However here we are not helping …

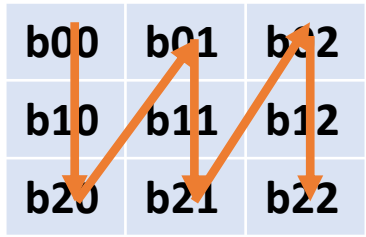


i	j	k
0	2	0

Sub-optimal memory access will kill the performance…

The less efficient way (column-major):

```
for(int i=0; i<n; ++i)
   for(int j=0; j<n; ++j)
      for(int k=0; k<n; ++k)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



- Processors **load data into their cache** for fast reuse
- They always load a bunch of data at the same time
- They do **branch-prediction** and **pre-fetching**!

However here we are not helping …

Processor cache

Cache Misses: 7

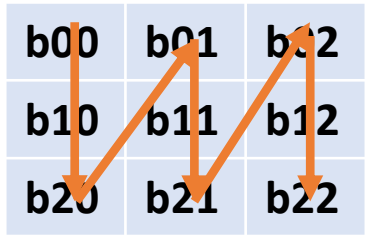
a00	a01	a02	a10	a11	a12	a20	a21	a22
b00	b01	b02	b10	b11	b12	b20	b21	b22
c00	c01	c02	c10	c11	c12	c20	c21	c22

i	j	k
0	2	1

Sub-optimal memory access will kill the performance…

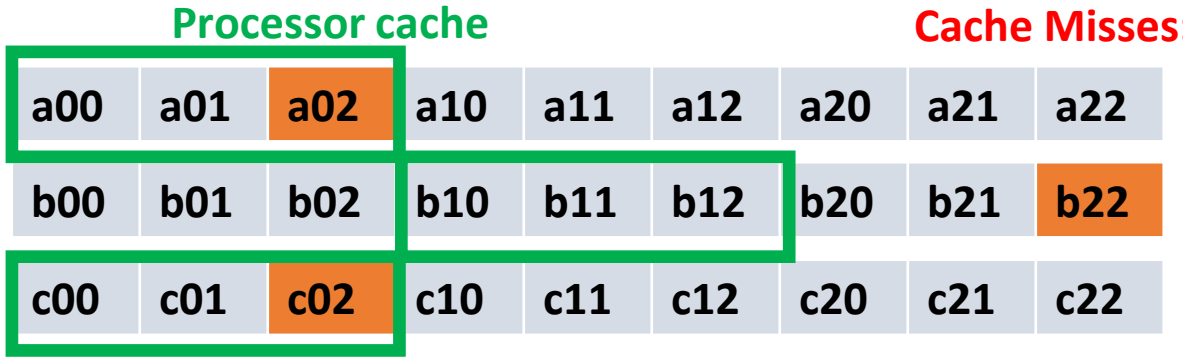
The less efficient way (column-major):

```
for(int i=0; i<n; ++i)
   for(int j=0; j<n; ++j)
      for(int k=0; k<n; ++k)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



- Processors **load data into their cache** for fast reuse
- They always load a bunch of data at the same time
- They do **branch-prediction** and **pre-fetching**!

However here we are not helping …

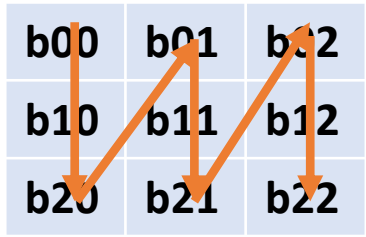


i	j	k
0	2	2

Sub-optimal memory access will kill the performance…

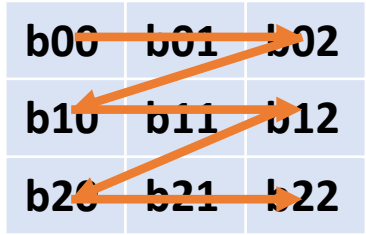
The less efficient way (column-major):

```
for(int i=0; i<n; ++i)
   for(int j=0; j<n; ++j)
      for(int k=0; k<n; ++k)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



The better way (row-major):

```
for(int i=0; i<n; ++i)
   for(int k=0; k<n; ++k)
       for(int j=0; j<n; ++j)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



Here, the most inner loop can be executed in parallel (without write conflicts on matrix c)

The better way (row-major):

```
for(int i=0; i<n; ++i)
   for(int k=0; k<n; ++k)
      for(int j=0; j<n; ++j)
         c[i*n+j] += a[i*n+k] * b[k*n+j];
```

Let's go…

Processor cache

Cache Misses: 0

a00	a01	a02	a10	a11	a12	a20	a21	a22
b00	b01	b02	b10	b11	b12	b20	b21	b22
c00	c01	c02	c10	c11	c12	c20	c21	c22

i	k	j
0	0	0

The better way (row-major):

```
for(int i=0; i<n; ++i)
   for(int k=0; k<n; ++k)
      for(int j=0; j<n; ++j)
         c[i*n+j] += a[i*n+k] * b[k*n+j];
```

Processor cache

Cache Misses: 0

a00	a01	a02	a10	a11	a12	a20	a21	a22
b00	b01	b02	b10	b11	b12	b20	b21	b22
c00	c01	c02	c10	c11	c12	c20	c21	c22

i	k	j
0	0	1

The better way (row-major):

```
for(int i=0; i<n; ++i)
   for(int k=0; k<n; ++k)
      for(int j=0; j<n; ++j)
         c[i*n+j] += a[i*n+k] * b[k*n+j];
```

Processor cache

Cache Misses: 0

a00	a01	a02	a10	a11	a12	a20	a21	a22
b00	b01	b02	b10	b11	b12	b20	b21	b22
c00	c01	c02	c10	c11	c12	c20	c21	c22

i	k	j
0	0	2

The better way (row-major):

```
for(int i=0; i<n; ++i)
   for(int k=0; k<n; ++k)
      for(int j=0; j<n; ++j)
         c[i*n+j] += a[i*n+k] * b[k*n+j];
```

Processor cache

Cache Misses: 1

a00	a01	a02	a10	a11	a12	a20	a21	a22
b00	b01	b02	b10	b11	b12	b20	b21	b22
c00	c01	c02	c10	c11	c12	c20	c21	c22

i	k	j
0	1	0

The better way (row-major):

```
for(int i=0; i<n; ++i)
   for(int k=0; k<n; ++k)
      for(int j=0; j<n; ++j)
         c[i*n+j] += a[i*n+k] * b[k*n+j];
```

Processor cache

Cache Misses: 1

a00	a01	a02	a10	a11	a12	a20	a21	a22
b00	b01	b02	b10	b11	b12	b20	b21	b22
c00	c01	c02	c10	c11	c12	c20	c21	c22

i	k	j
0	1	1

The better way (row-major):

```
for(int i=0; i<n; ++i)
   for(int k=0; k<n; ++k)
      for(int j=0; j<n; ++j)
         c[i*n+j] += a[i*n+k] * b[k*n+j];
```

Processor cache

Cache Misses: 1

a00	a01	a02	a10	a11	a12	a20	a21	a22
b00	b01	b02	b10	b11	b12	b20	b21	b22
c00	c01	c02	c10	c11	c12	c20	c21	c22

i	k	j
0	1	2

The better way (row-major):

```
for(int i=0; i<n; ++i)
   for(int k=0; k<n; ++k)
      for(int j=0; j<n; ++j)
         c[i*n+j] += a[i*n+k] * b[k*n+j];
```

Processor cache

Cache Misses: 2

a00	a01	a02	a10	a11	a12	a20	a21	a22
b00	b01	b02	b10	b11	b12	b20	b21	b22
c00	c01	c02	c10	c11	c12	c20	c21	c22

i	k	j
0	2	0

The better way (row-major):

```
for(int i=0; i<n; ++i)
   for(int k=0; k<n; ++k)
      for(int j=0; j<n; ++j)
         c[i*n+j] += a[i*n+k] * b[k*n+j];
```

Processor cache

Cache Misses: 2

a00	a01	a02	a10	a11	a12	a20	a21	a22
b00	b01	b02	b10	b11	b12	b20	b21	b22
c00	c01	c02	c10	c11	c12	c20	c21	c22

i	k	j
0	2	1

The better way (row-major):

```
for(int i=0; i<n; ++i)
   for(int k=0; k<n; ++k)
      for(int j=0; j<n; ++j)
         c[i*n+j] += a[i*n+k] * b[k*n+j];
```

Processor cache

Cache Misses: 2

a00	a01	a02	a10	a11	a12	a20	a21	a22
b00	b01	b02	b10	b11	b12	b20	b21	b22
c00	c01	c02	c10	c11	c12	c20	c21	c22

i	k	j
0	2	2

- We create as many kernel instances as we need!
- In this case, we divide the problems into 2D blocks of 16*16 threads each
- The grid of blocks is large enough to accommodate all the blocks

```
__global__ void cuda_mul(float* a, float* b, float* c, int size) {
    int row = blockIdx.y*blockDim.y+threadIdx.y;
    int col = blockIdx.x*blockDim.x+threadIdx.x;
    for (int i = 0; i < size; i++)
        c[row*size+col] += a[row*size+i] * b[i*size+col];
}
```

CUDA Kernel

```
cudaMemcpy(dm1, a, sizeof(float)*size*size, cudaMemcpyHostToDevice);
cudaMemcpy(dm2, b, sizeof(float)*size*size, cudaMemcpyHostToDevice);
cudaMemcpy(dm3, c, sizeof(float)*size*size, cudaMemcpyHostToDevice);

dim3 blockSize = dim3(16, 16);
dim3 gridSize = dim3(size / blockSize.x, size/ blockSize.y);

cuda_mul<<<gridSize, blockSize>>>(dm1, dm2, dm3, size);

cudaMemcpy(c, dm3, sizeof(float)*size*size, cudaMemcpyDeviceToHost);
```

Kernel Call

Each thread inside a 2D block receives:

- blockidx.x, blockidx.y
- threadidx.x, threadidx.y

This information is used to retrieve the row and col indices (formerly i and j).

This is the only information that differs between threads!

Remember, the data is not sent to individual threads

Instead, the data is mapped onto the grid of blocks

And threads know which data to access thanks to the block and thread ids.

```
__global__ void cuda_mul(float* a, float* b, float* c, int size) {
    int row = blockIdx.y*blockDim.y+threadIdx.y;
    int col = blockIdx.x*blockDim.x+threadIdx.x;
    for (int i = 0; i < size; i++)
        c[row*size+col] += a[row*size+i] * b[i*size+col];
}
```

CUDA Kernel

Each thread inside a 2D block receives:

- blockidx.x, blockidx.y
- threadidx.x, threadidx.y

This information is used to retrieve the row and col indices (formerly i and j).

This is the only information that differs between threads!

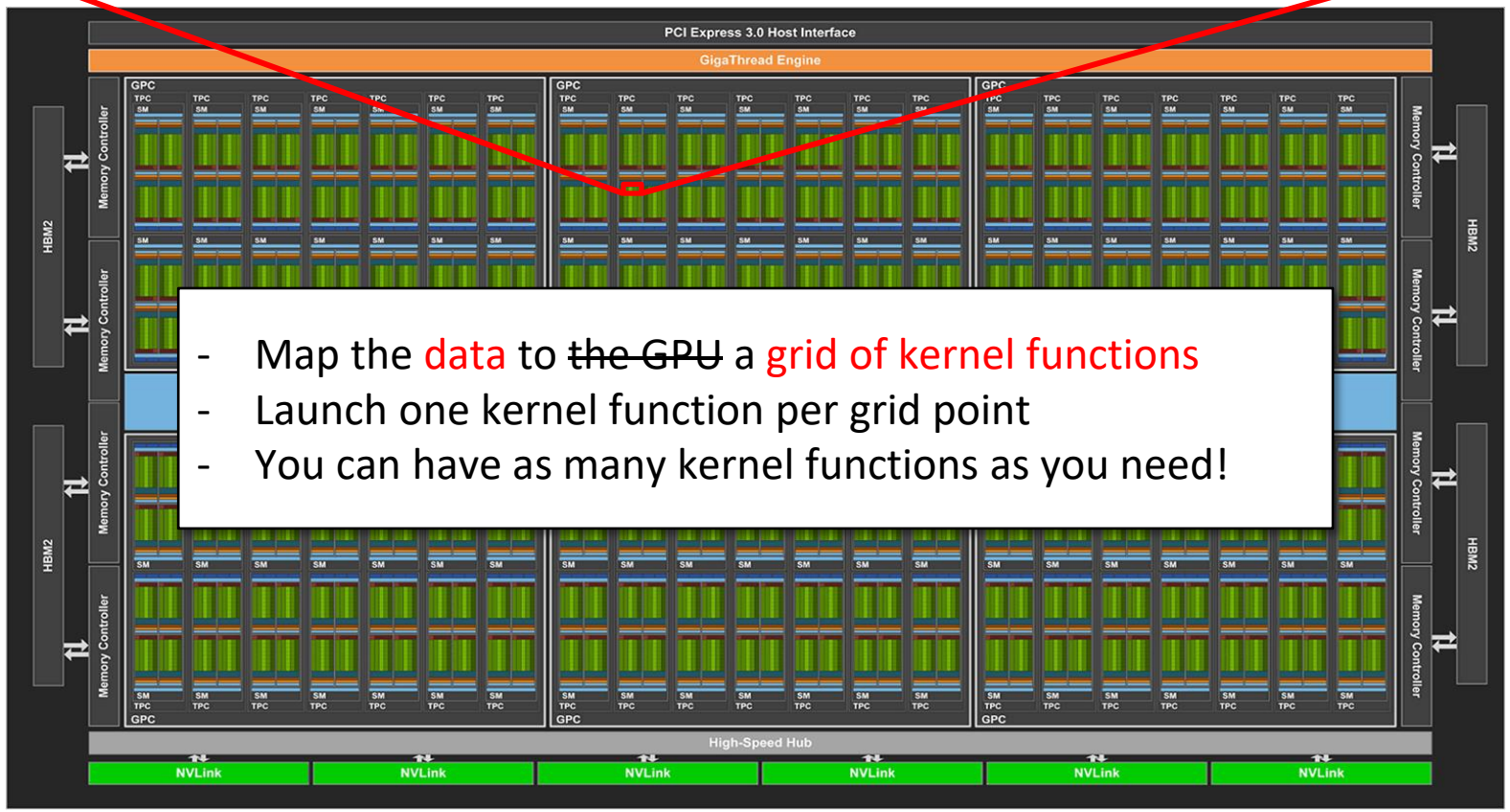
Remember, the data is not sent to individual threads

Instead, the data is mapped onto the grid of blocks

And threads know which data to access thanks to the block and thread ids.

```
__global__ void cuda_mul(float* a, float* b, float* c, int size) {
    int row = blockIdx.y*blockDim.y+threadIdx.y;
    int col = blockIdx.x*blockDim.x+threadIdx.x;
    for (int i = 0; i < size; i++)
        c[row*size+col] += a[row*size+i] * b[i*size+col];
}
```

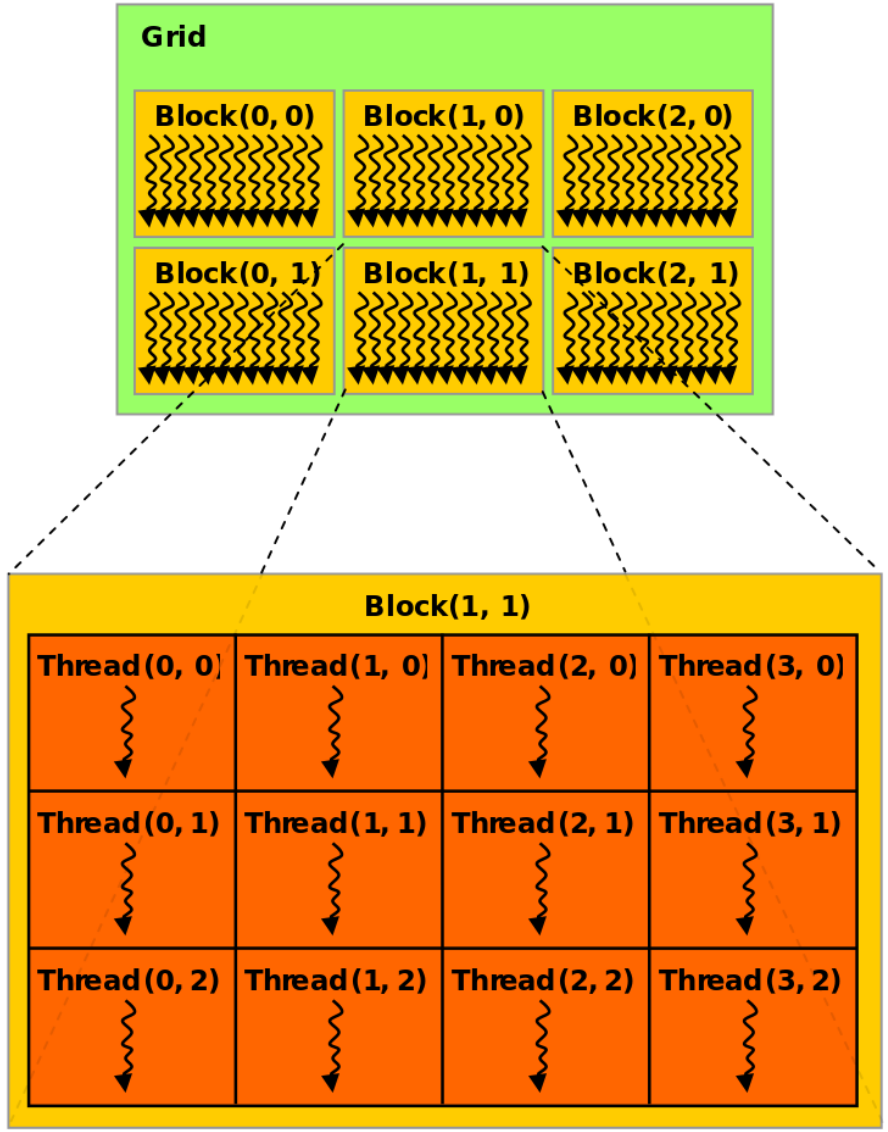
CUDA Kernel

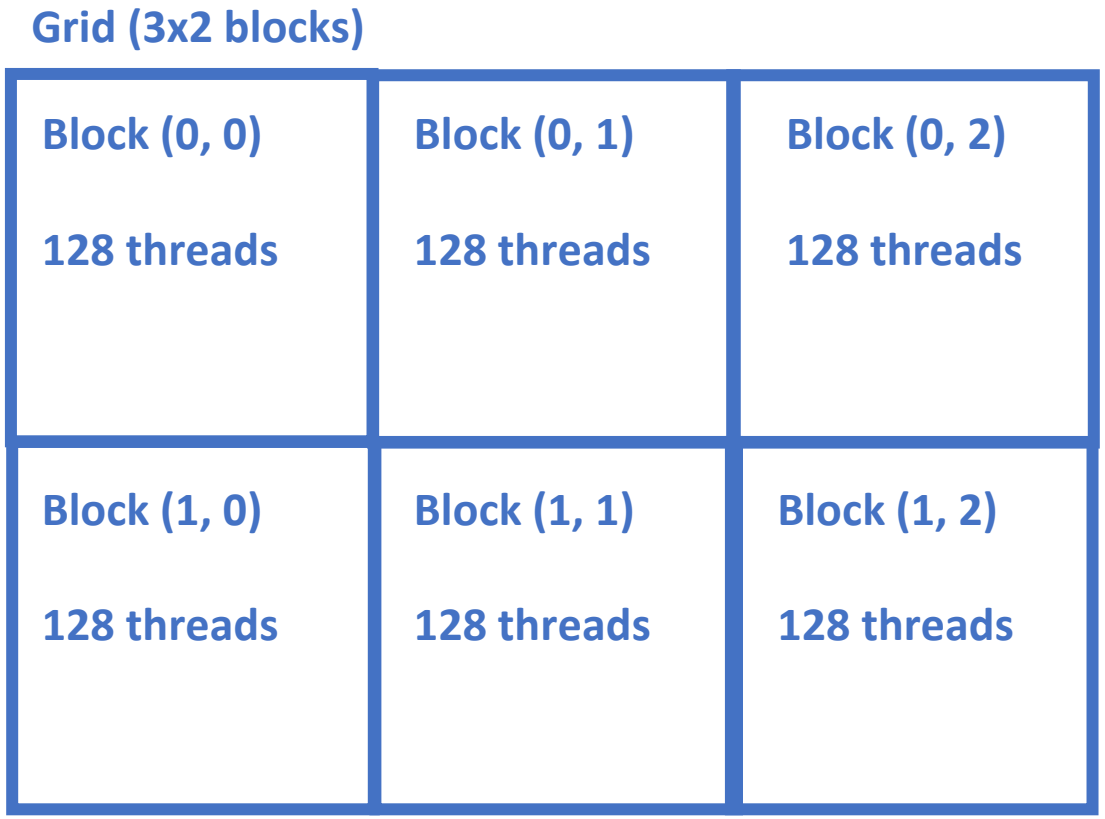


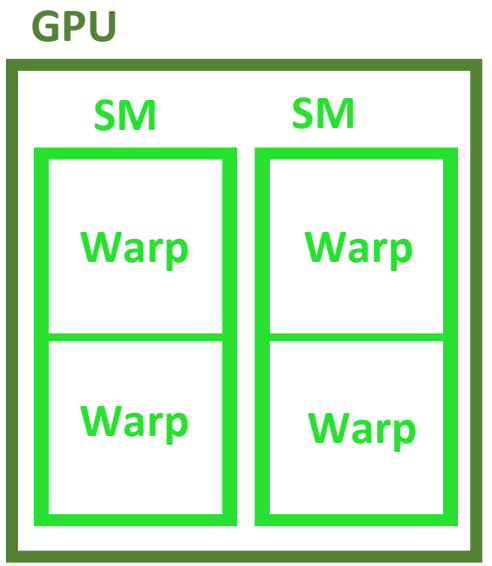
- Map the data to ~~the GPU~~ a grid of kernel functions
- Launch one kernel function per grid point
- You can have as many kernel functions as you need!

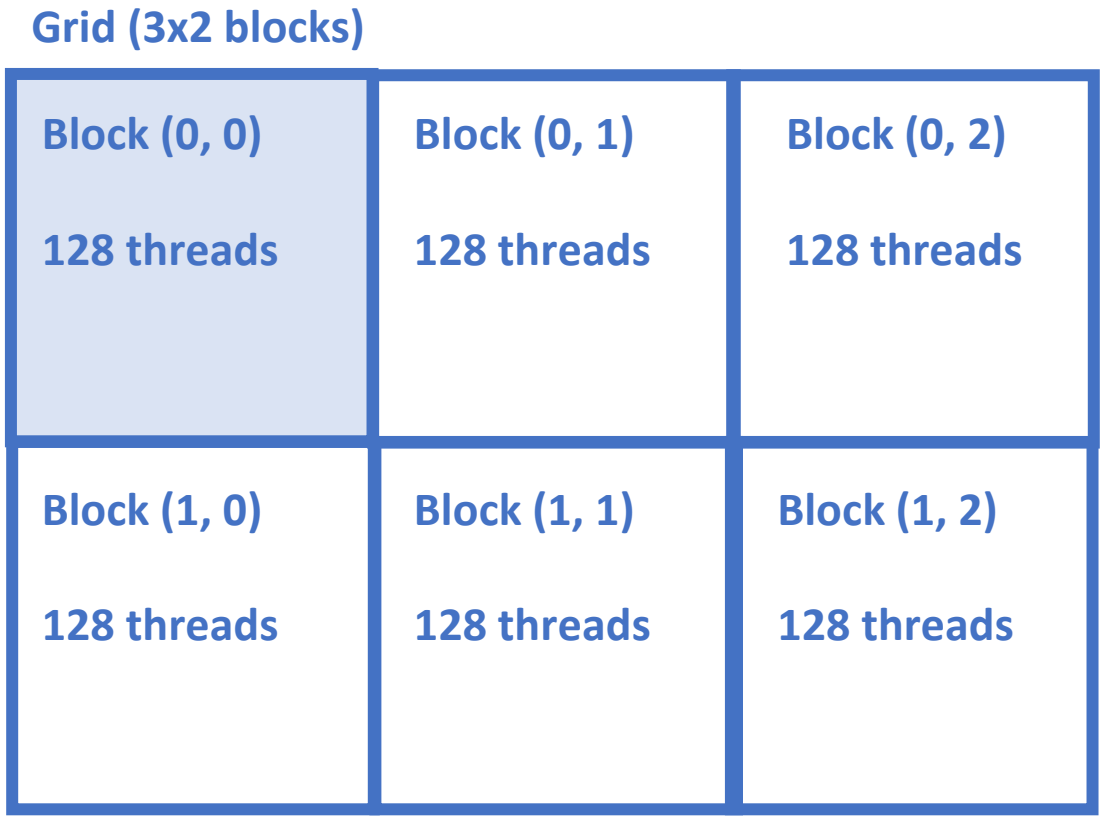
Reminder: Grids, blocks, threads and warps

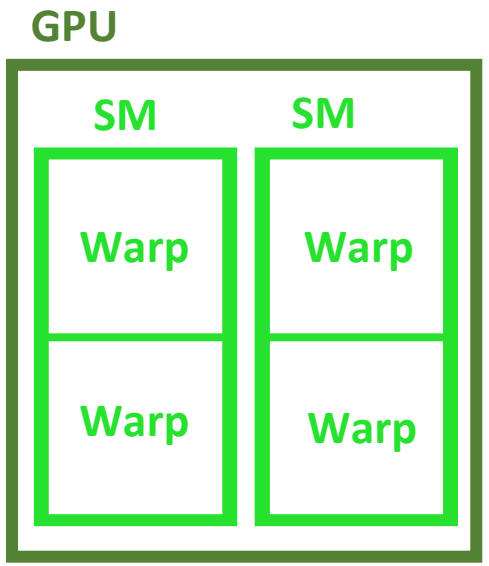
- Grid:
  - 1 to as many blocks as you want
- Block
  - 1 to 1024 threads
  - Should be a multiple of 32 threads
  - Execute on a single Streaming Multiprocessor (SM)
  - Possibility of having shared memory between threads
  - Is executed in **warps** of 32 threads
- Warps
  - Warps always execute 32 threads inside a SM
  - **All the 32 threads in a warp execute the same instruction** (lockstep)

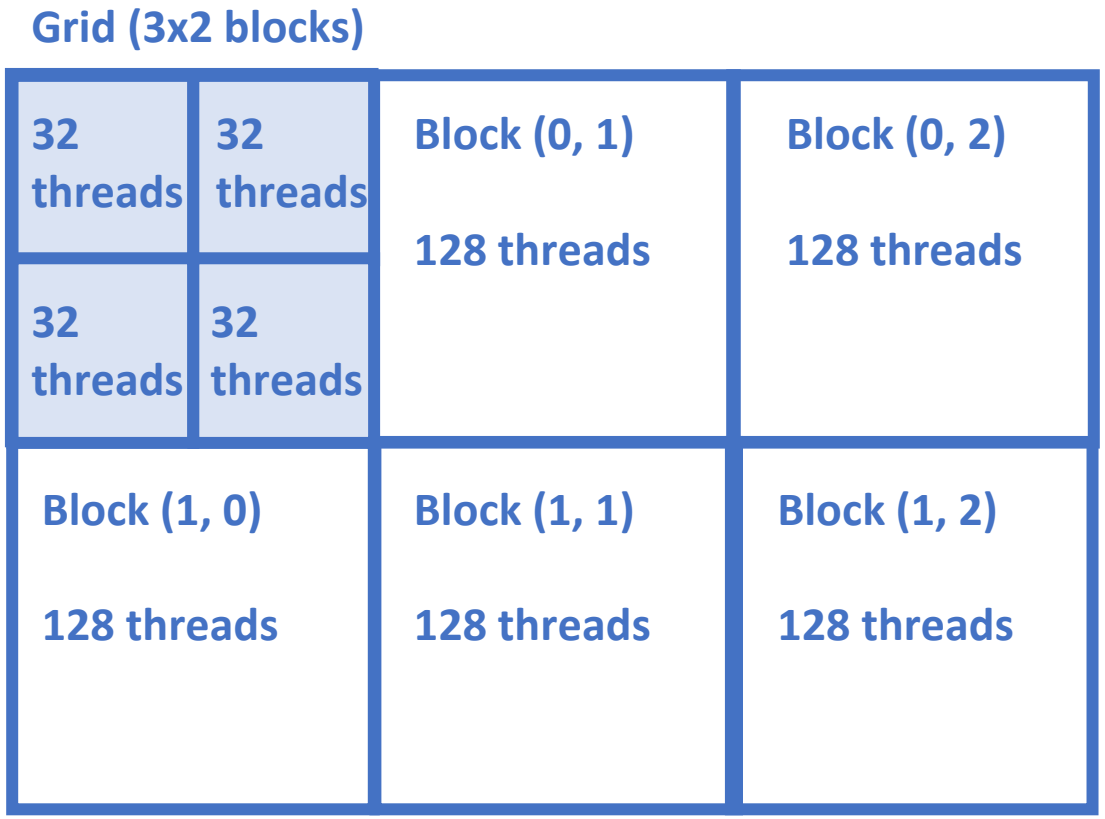


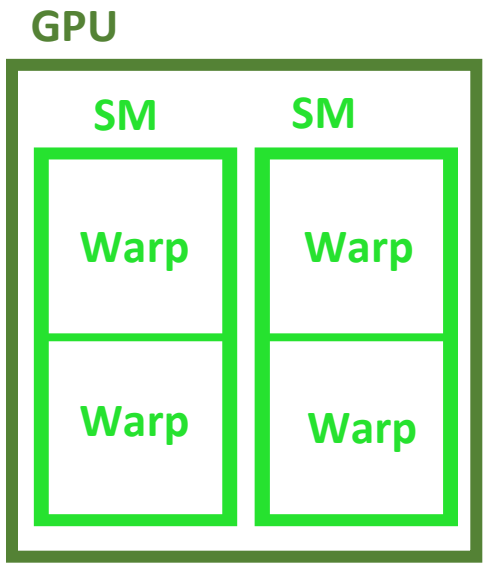

Grid (3x2 blocks)
Block (0, 0)
128 threads
Block (0, 1)
128 threads
Block (0, 2)
128 threads
Block (1, 0)
128 threads
Block (1, 1)
128 threads
Block (1, 2)
128 threads

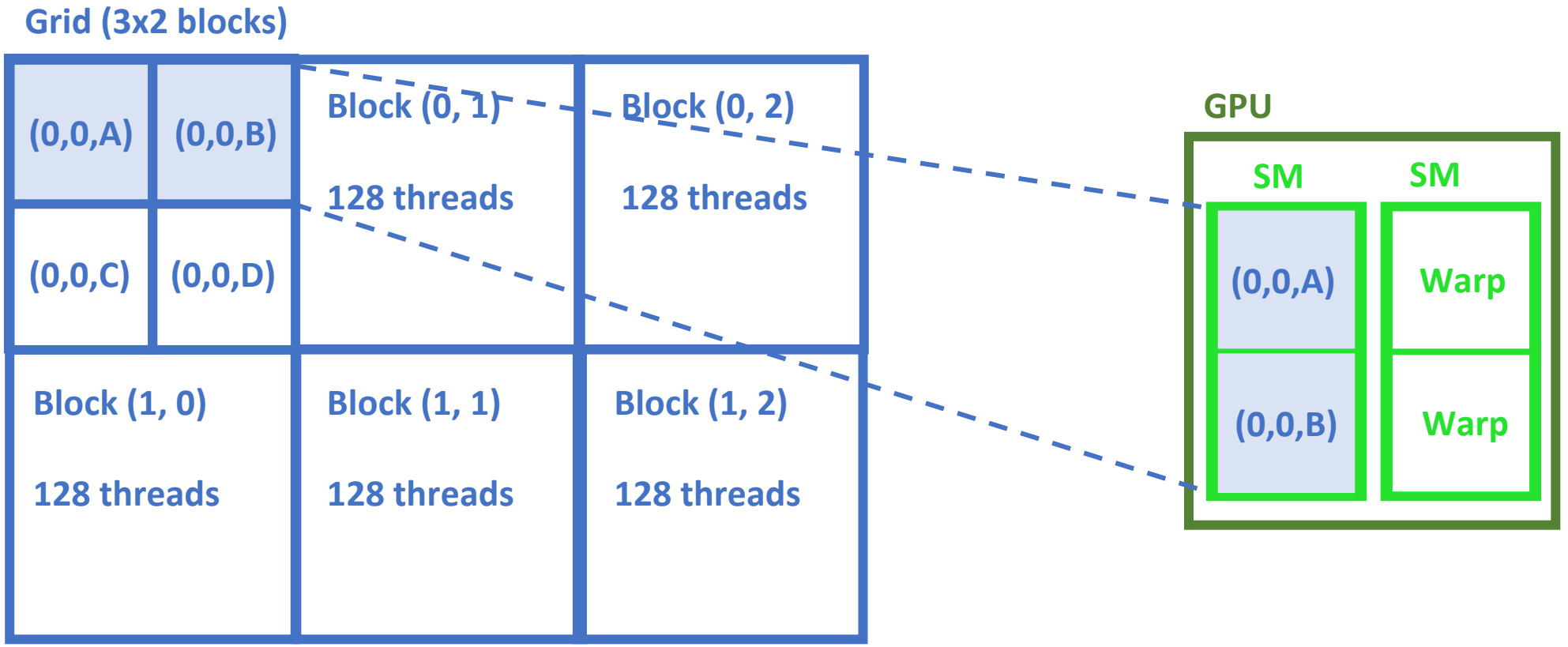

GPU
SM
SM
Warp
Warp
Warp
Warp


Grid (3x2 blocks)
Block (0, 0)
128 threads
Block (0, 1)
128 threads
Block (0, 2)
128 threads
Block (1, 0)
128 threads
Block (1, 1)
128 threads
Block (1, 2)
128 threads


GPU
SM
SM
Warp
Warp
Warp
Warp

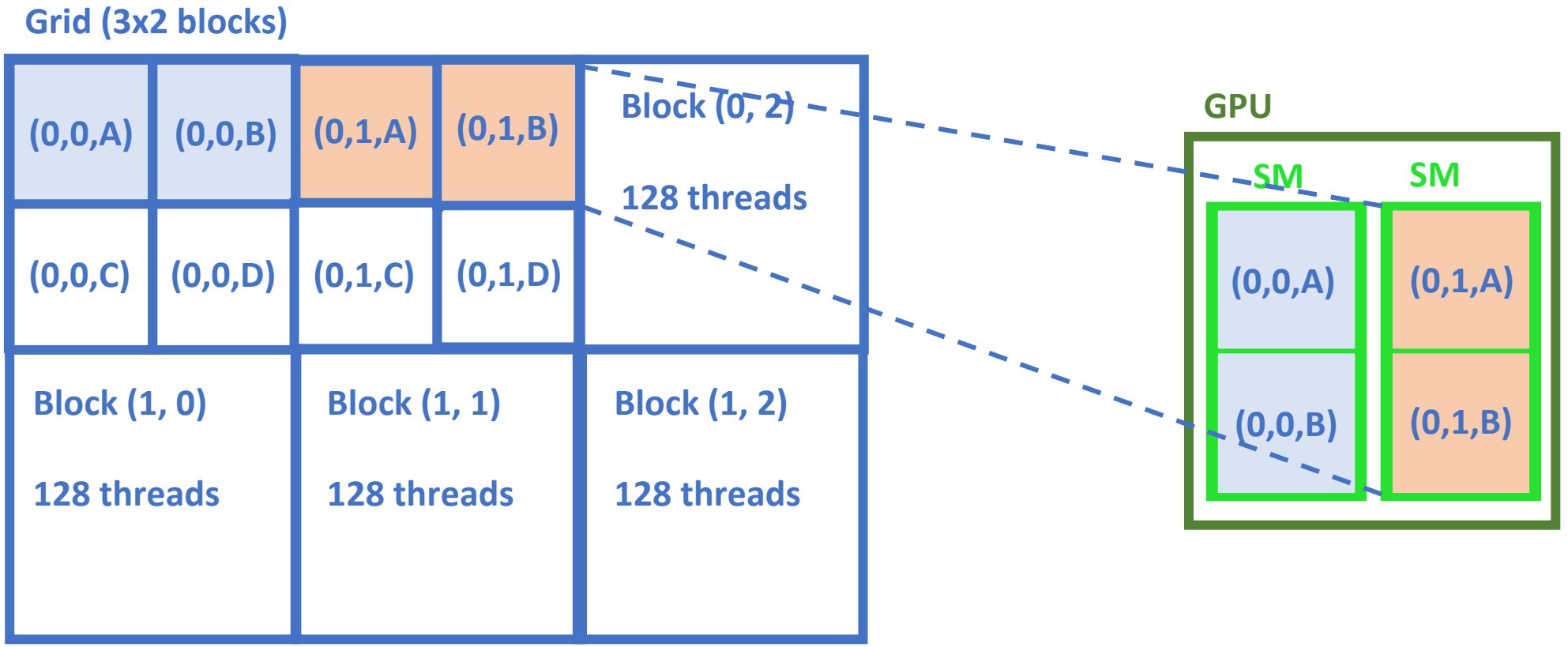

Grid (3x2 blocks)
32 threads
32 threads
32 threads
32 threads
Block (0, 1)
128 threads
Block (0, 2)
128 threads
Block (1, 0)
128 threads
Block (1, 1)
128 threads
Block (1, 2)
128 threads


GPU
SM
SM
Warp
Warp
Warp
Warp



Remember: a block can only be executed by a single SM!

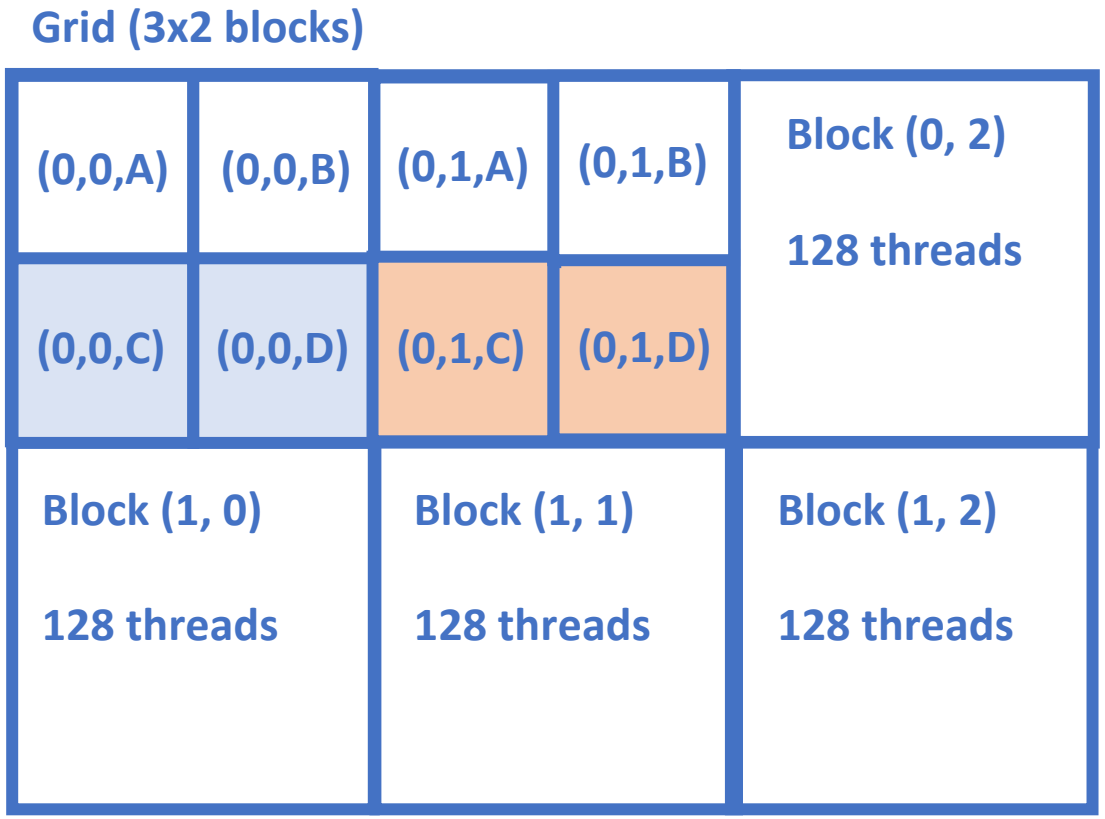
- The first block is scheduled on two warps of 32 threads on the first SM.

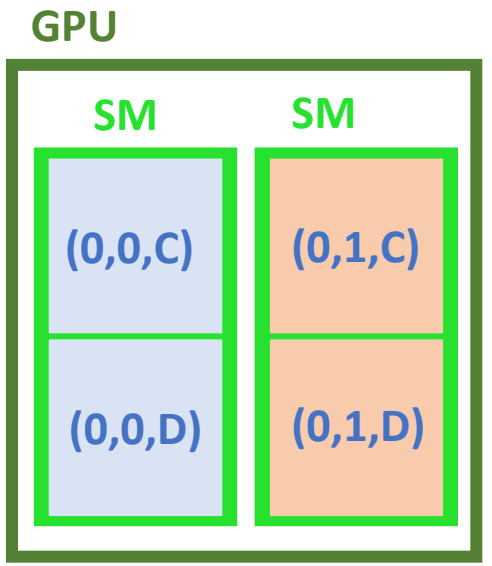


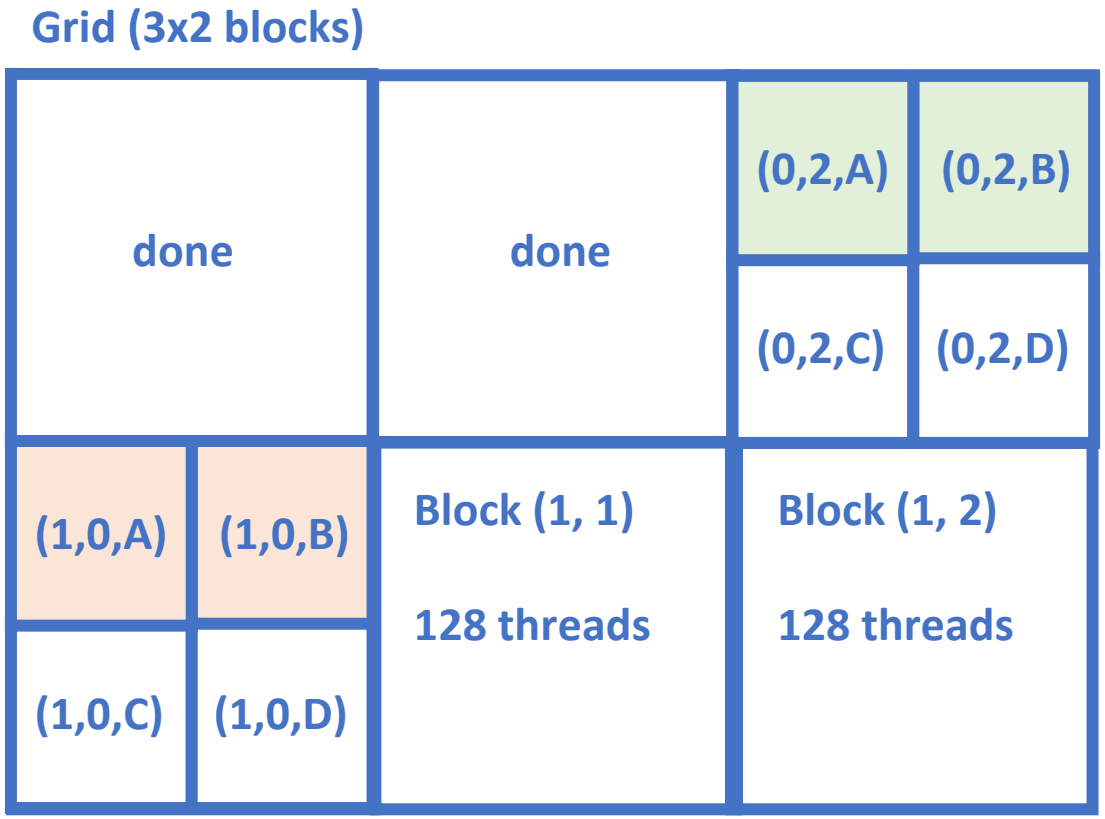
Remember: a block can only be executed by a single SM!

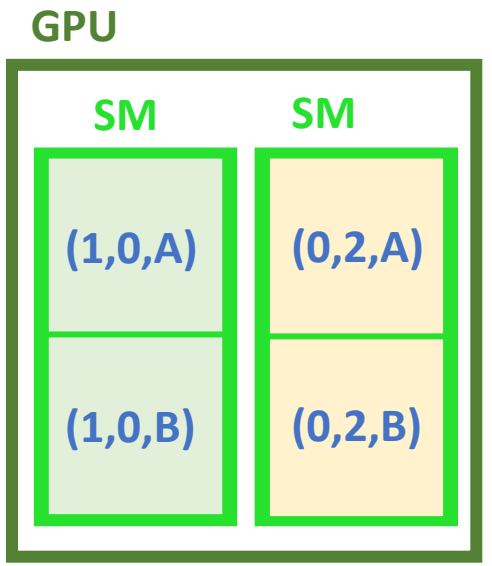
- The first block is scheduled on two warps of 32 threads on the first SM.
- The second block is scheduled on the second warp and the GPU is occupied at 100%

Execution of a block on a GPU

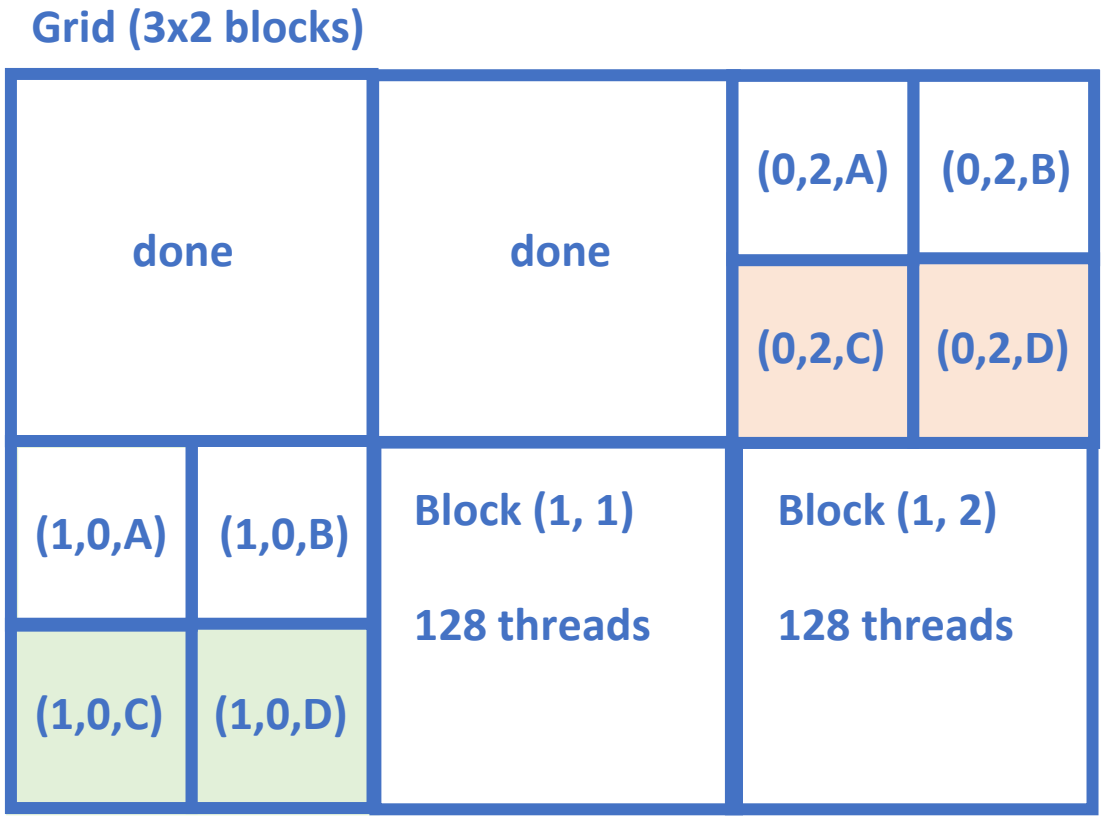


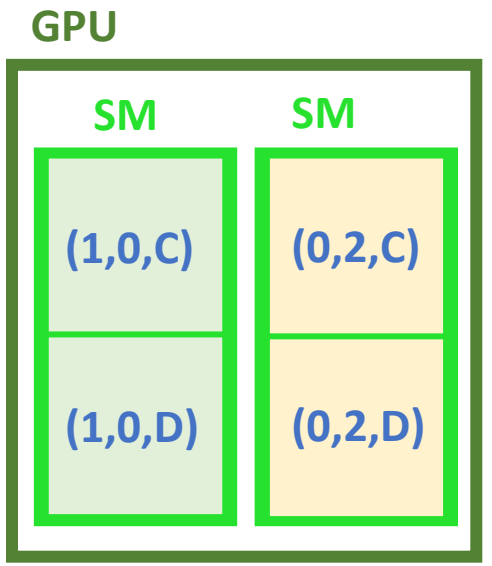




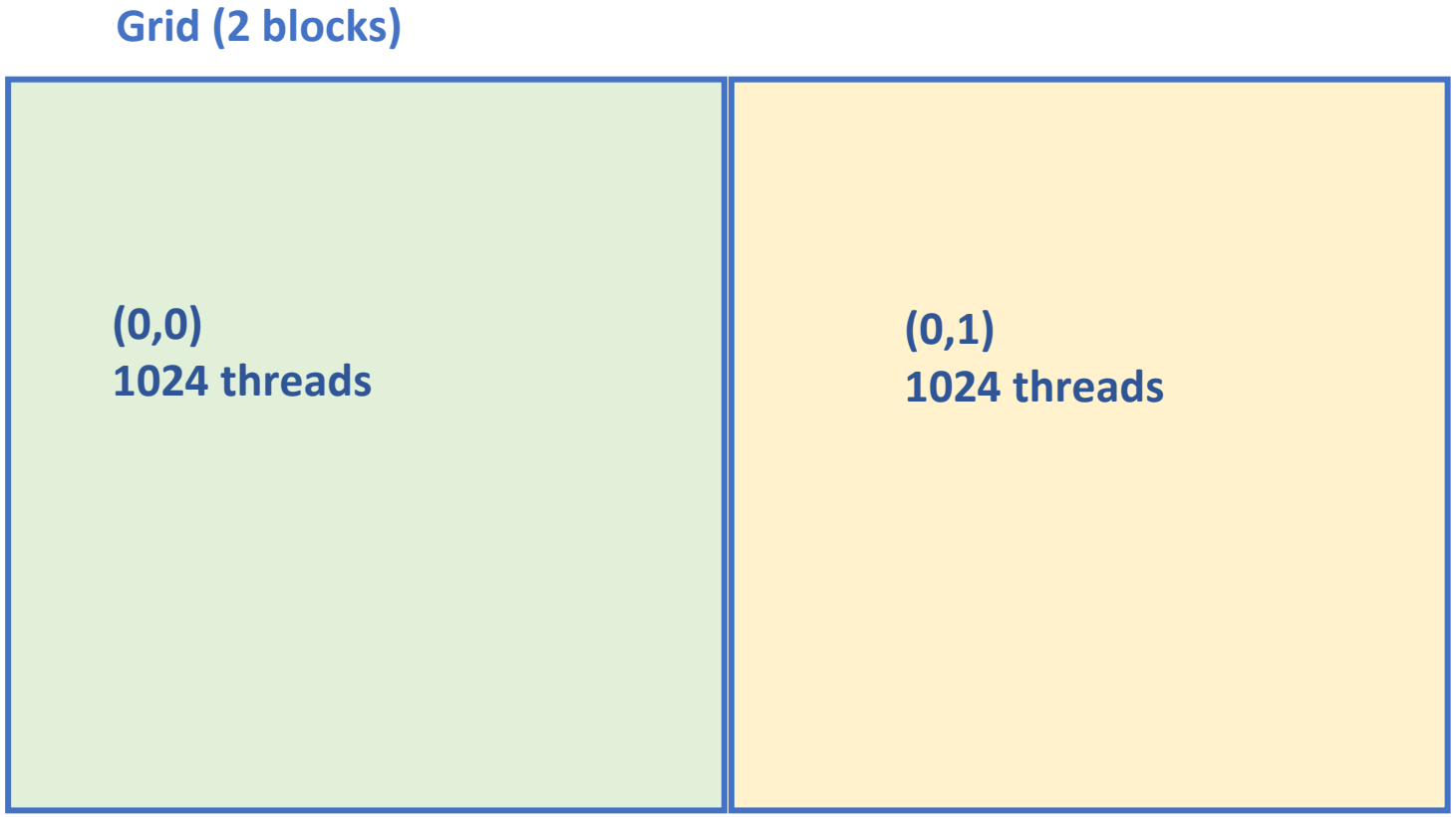


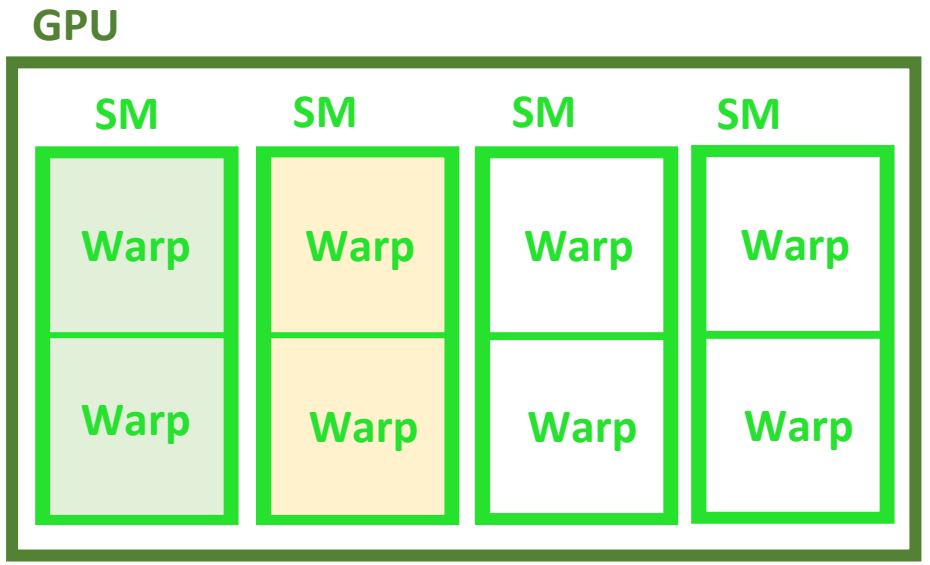
As soon as blocks (0,0) and (0,1) are done, the GPU will start executing the next blocks


Grid (3x2 blocks)
done
done
(0,2,A)
(0,2,B)
(0,2,C)
(0,2,D)
(1,0,A)
(1,0,B)
(1,0,C)
(1,0,D)
Block (1, 1)
128 threads
Block (1, 2)
128 threads


GPU
SM
SM
(1,0,C)
(0,2,C)
(1,0,D)
(0,2,D)

Execution of a block on a GPU





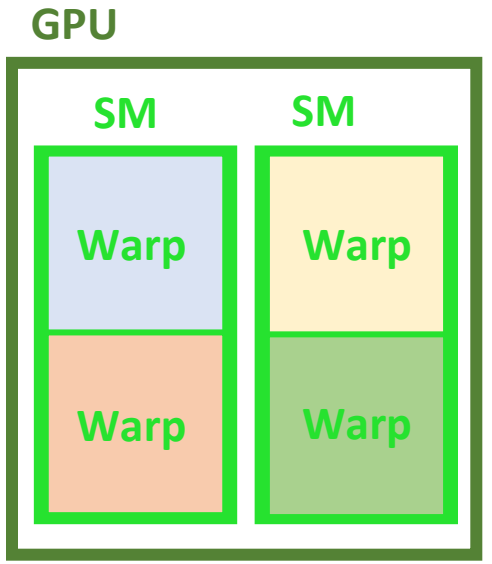
50% usage

Not enough blocks can result in underutilization!

Multiple blocks can execute in the same SM

This can happen if a block does not have enough warps to fill the SM

But remember: a block is always executed in a single SM.



Outline

Day 1

(09:00 – 13:00)

- ❑ Introduction
- ❑ Coding a CUDA kernel
- ❑ Example
  - ❑ Matrix Multiplication + Exercise
- ❑ Real world implementations
  - ❑ Genomes simulation
  - ❑ Random numbers generation
- ❑ Homework

Outline

Day 1

(09:00 – 13:00)

- ❑ Introduction
- ❑ Coding a CUDA kernel
- ❑ Example
  - ❑ Matrix Multiplication + Exercise
- ❑ Real world implementations
  - ❑ Genomes simulation
  - ❑ Random numbers generation
- ❑ Homework