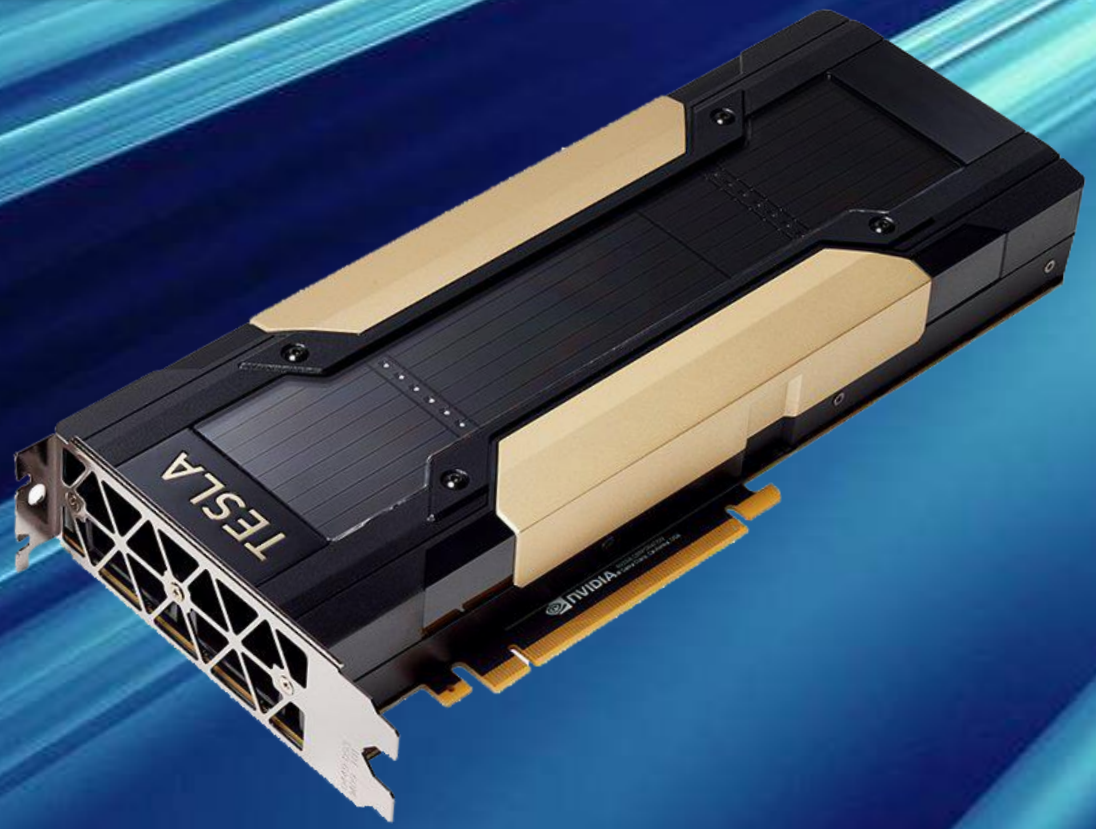
sciCORE
Center for scientific computing

Universität Basel

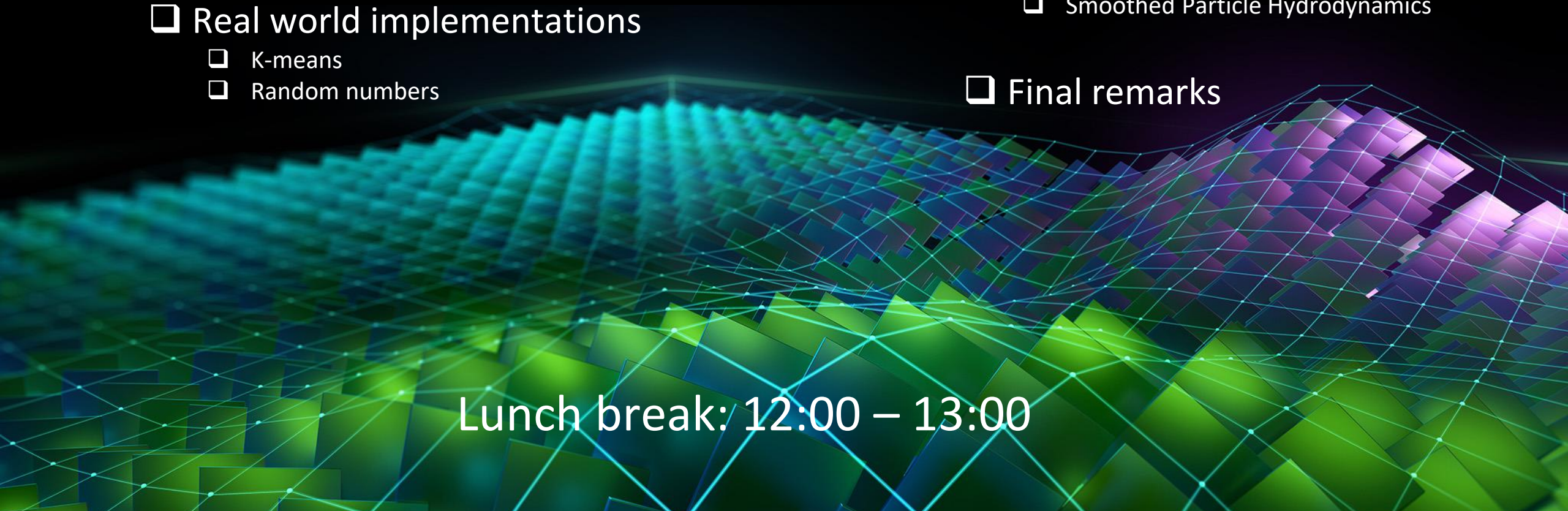
SIB
Swiss Institute of Bioinformatics



Introduction to GPU programming

Outline

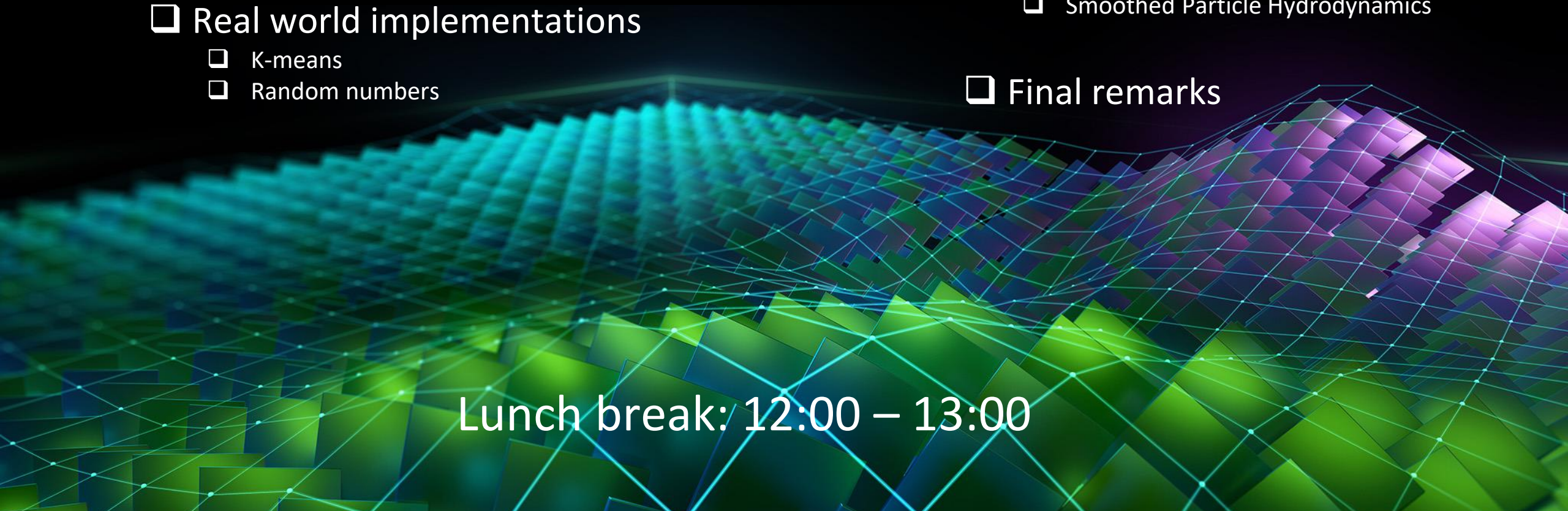
- ❑ Introduction
- ❑ Example
  - ❑ Matrix Multiplication
- ❑ Real world implementations
  - ❑ K-means
  - ❑ Random numbers
- ❑ Common pitfalls
- ❑ Memory management
  - ❑ Image processing example
- ❑ Monitoring and asynchronicity
  - ❑ Smoothed Particle Hydrodynamics
- ❑ Final remarks



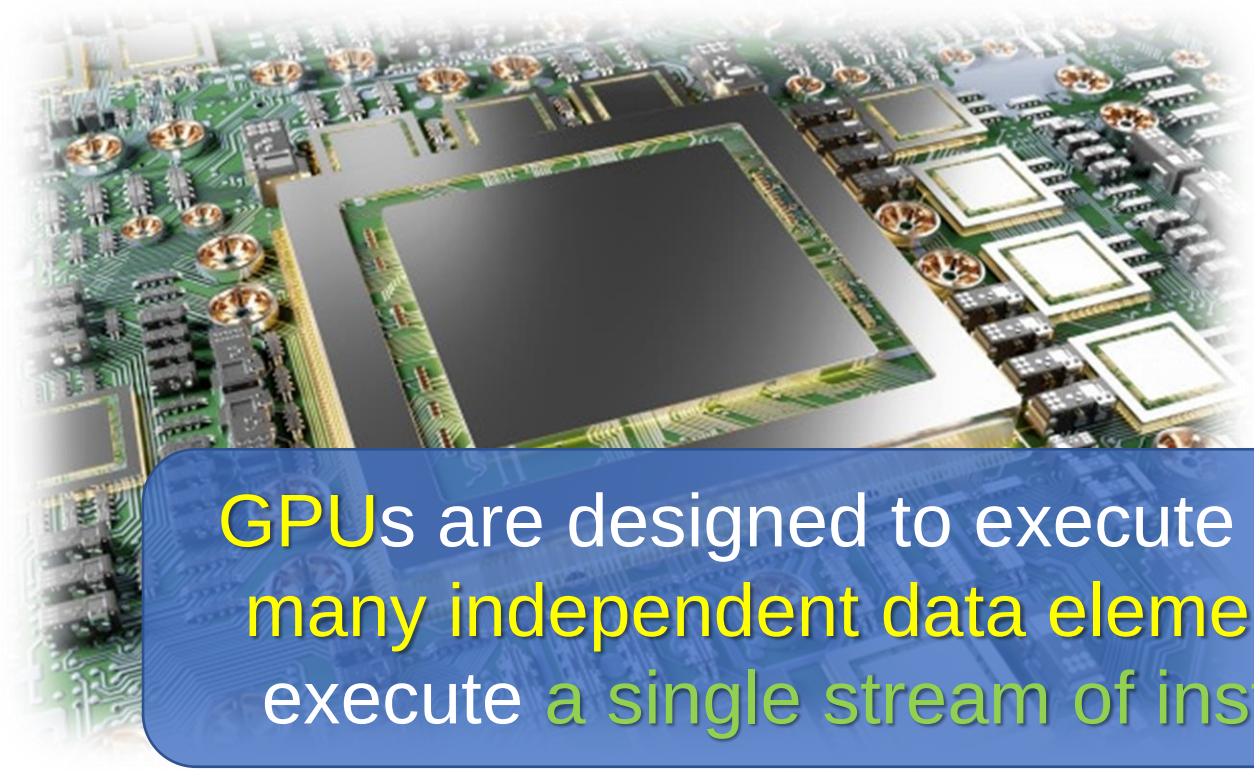
Lunch break: 12:00 – 13:00

Outline

- ❑ Introduction
- ❑ Example
  - ❑ Matrix Multiplication
- ❑ Real world implementations
  - ❑ K-means
  - ❑ Random numbers
- ❑ Common pitfalls
- ❑ Memory management
  - ❑ Image processing example
- ❑ Monitoring and asynchronicity
  - ❑ Smoothed Particle Hydrodynamics
- ❑ Final remarks



Lunch break: 12:00 – 13:00



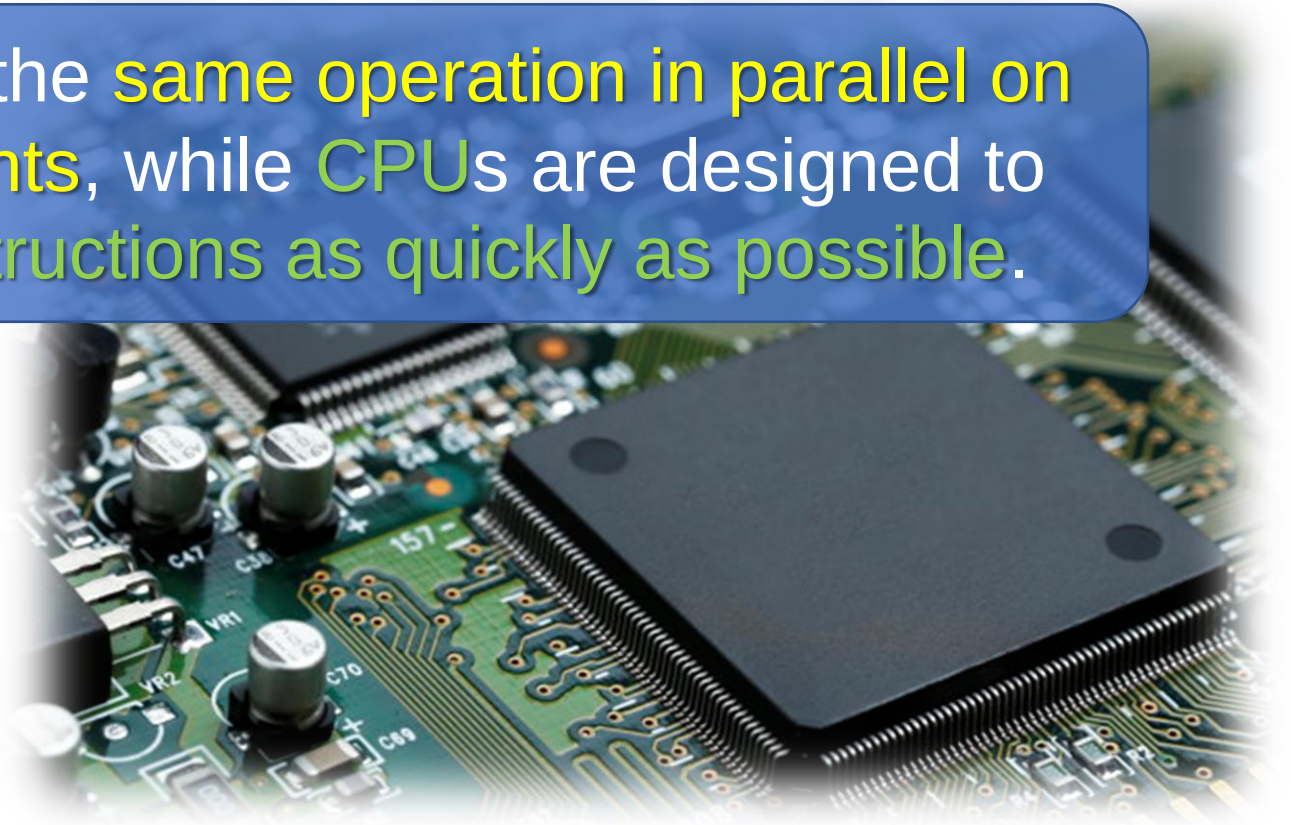
CPU

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.

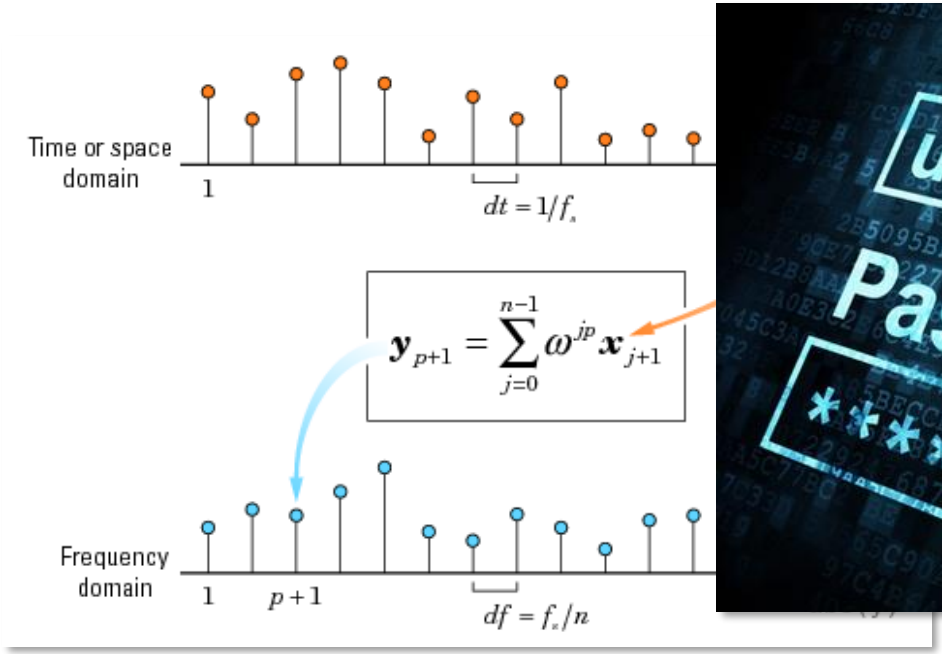
GPU

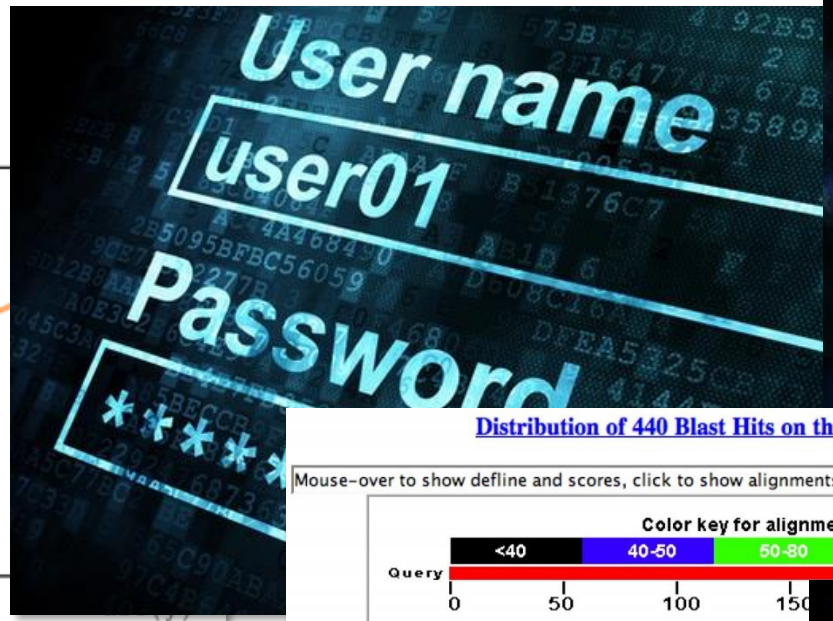
Graphics Processing Unit

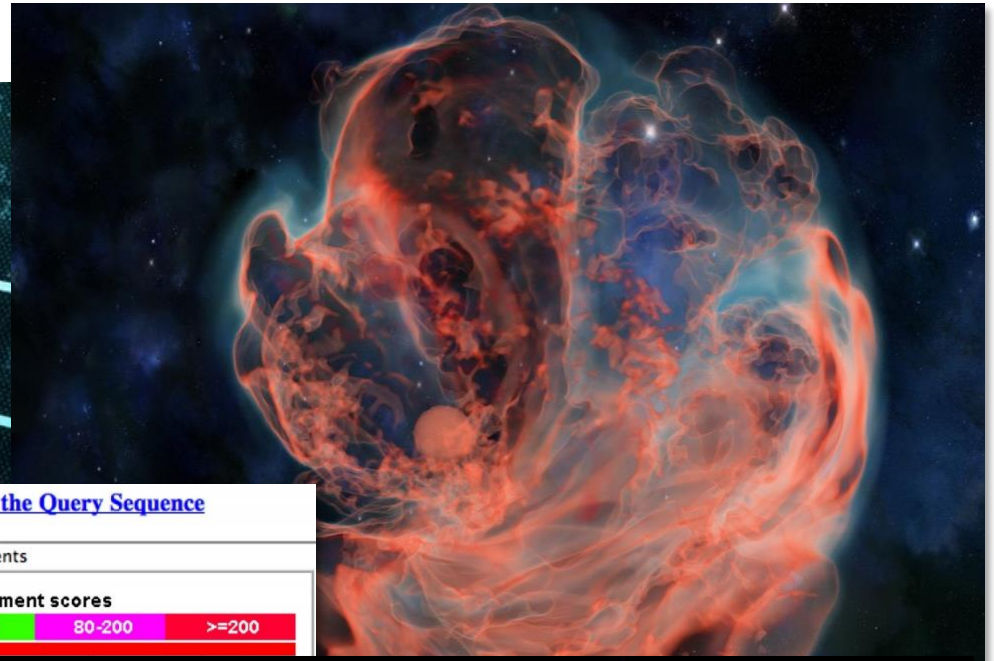


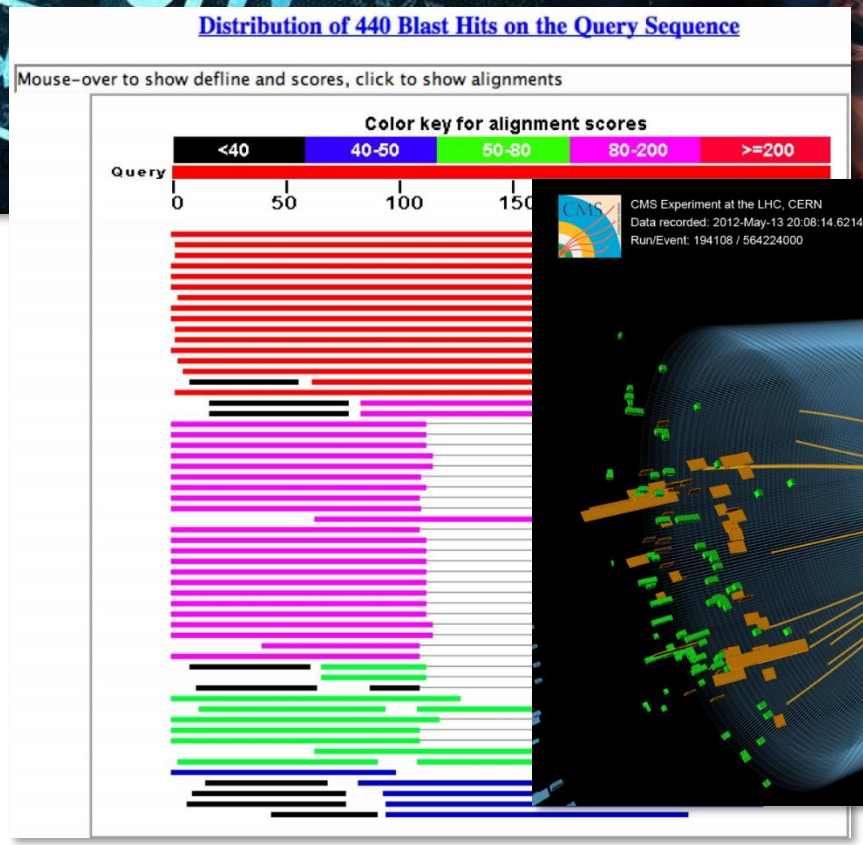
GPU

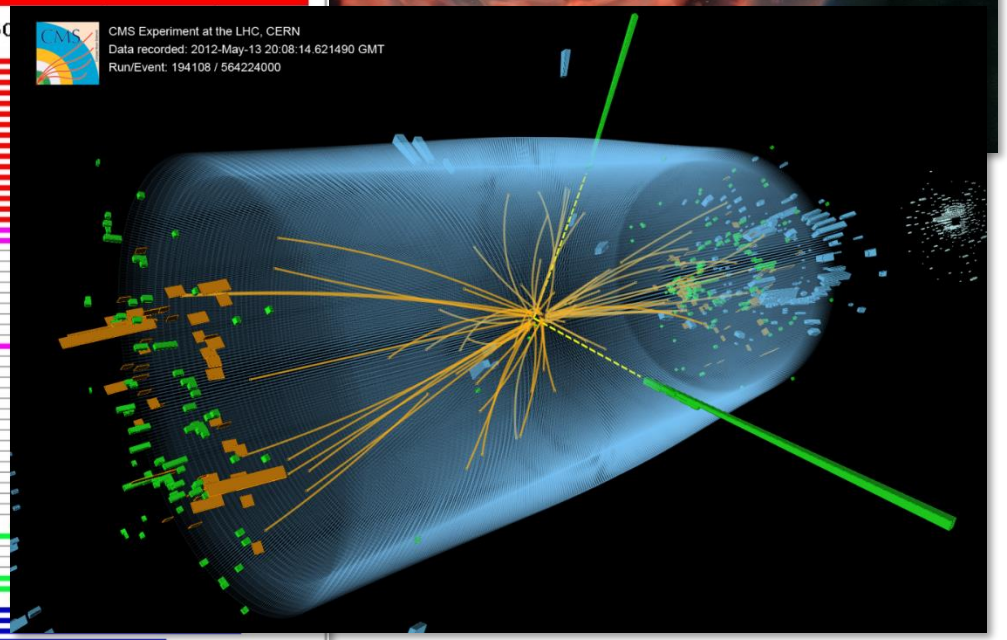
Graphics Processing Unit







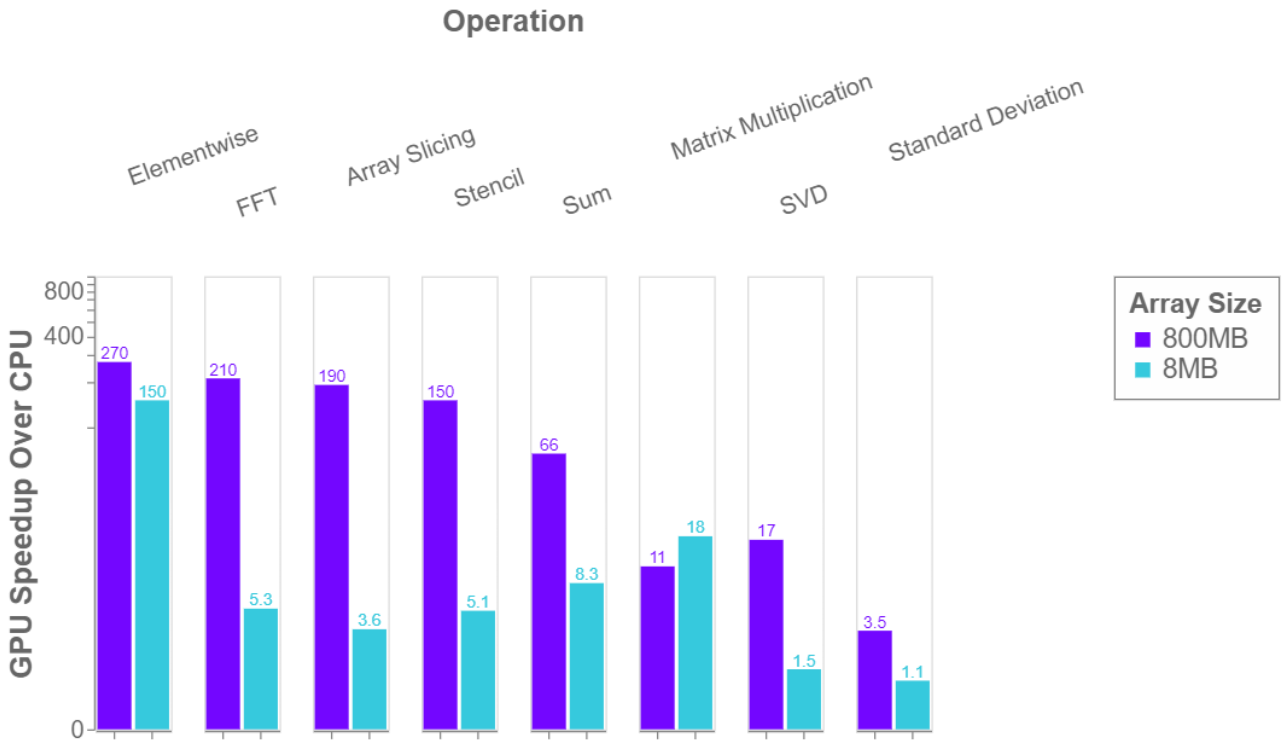




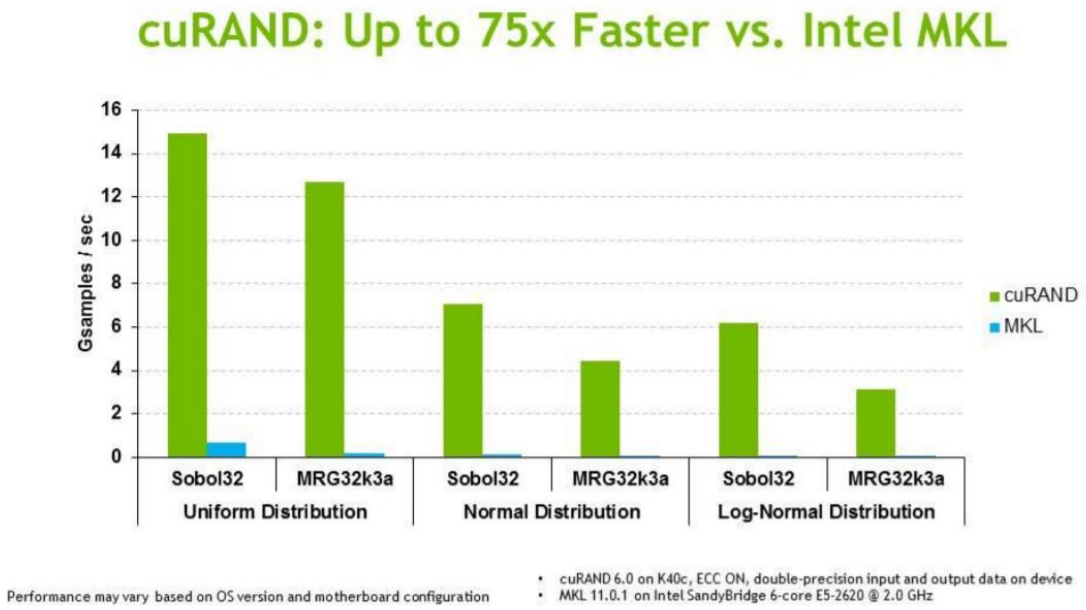
GPGPU

General Purprose

Graphics Processing Unit

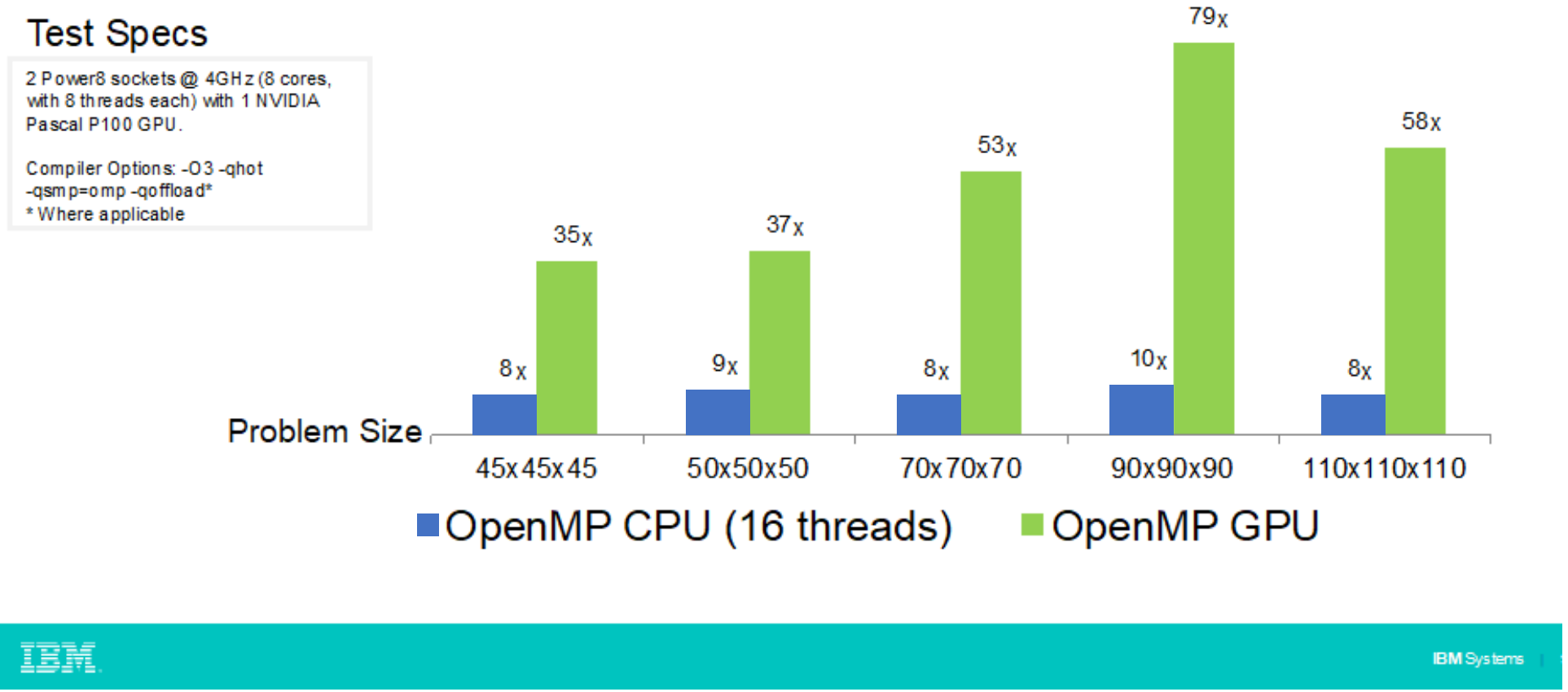


Single GPU using CuPy library (GPU optimized library) compared to numpy



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

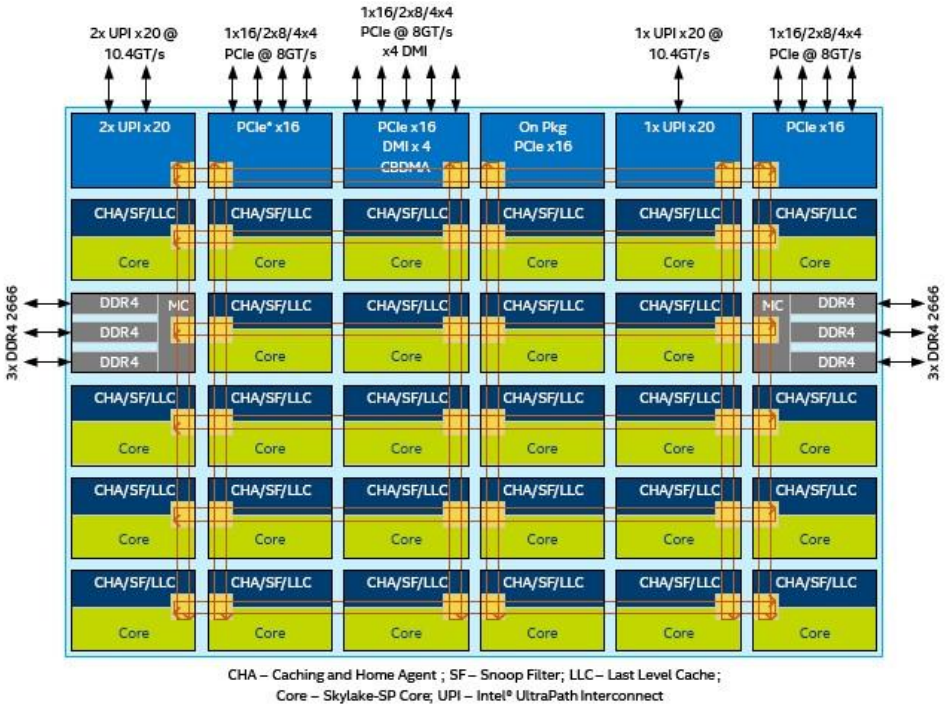
Source: https://blog.dask.org/2019/06/27/single-gpu-cupy-benchmarks
http://www.eurorisksystems.com/documents/speed_up_of_numeric_calculations_using_GPU.pdf



LULESH benchmark changing one single OMP directive.

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

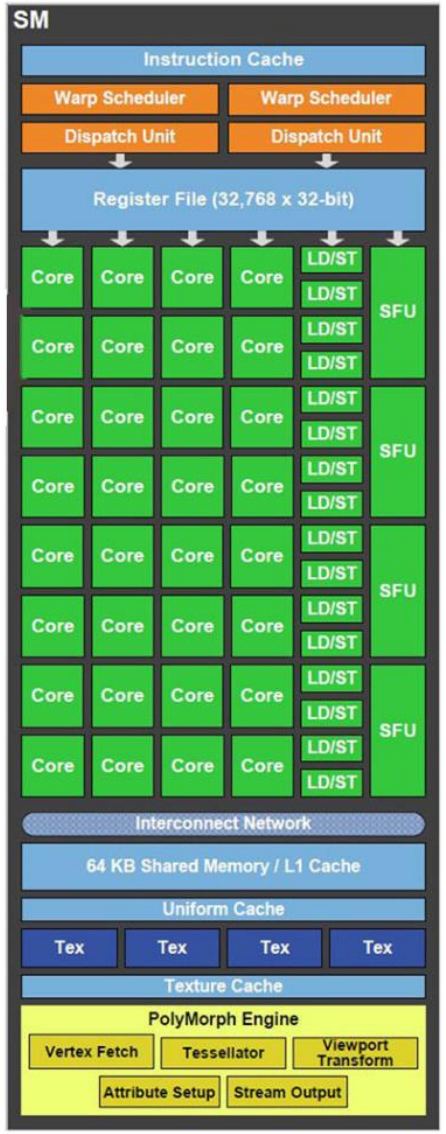
Intel Skylake (2017)



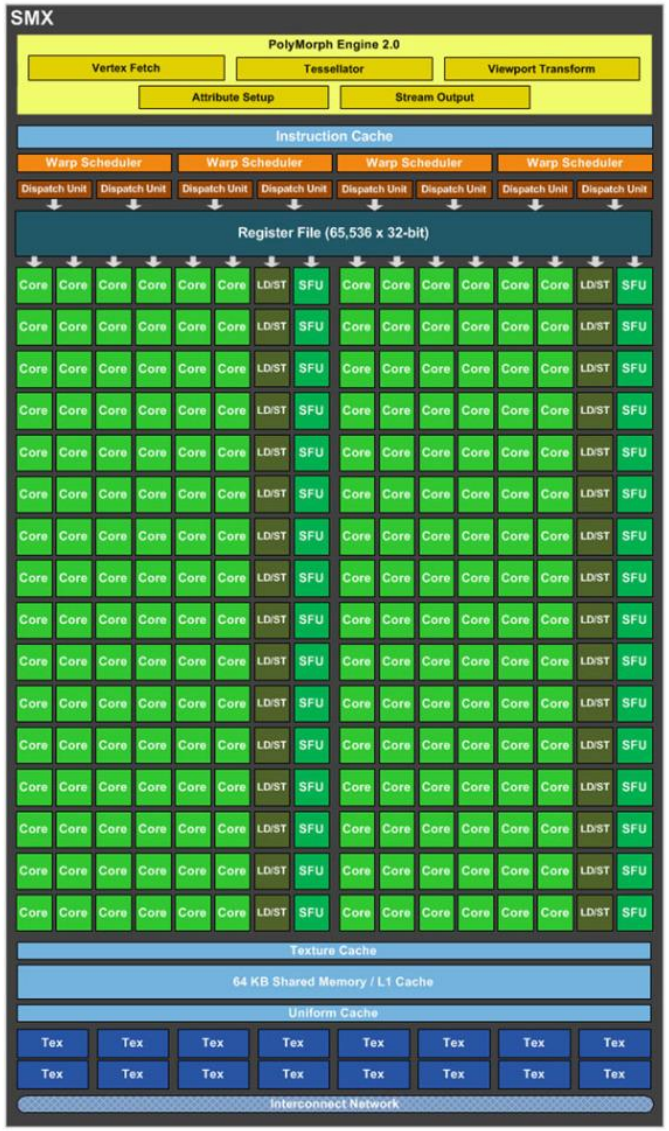
CHA – Caching and Home Agent ; SF – Snoop Filter; LLC – Last Level Cache ;
Core – Skylake-SP Core; UPI – Intel® UltraPath Interconnect

Streaming Multiprocessor

Tesla SM unit (2007)



Fermi SMX unit (2011)

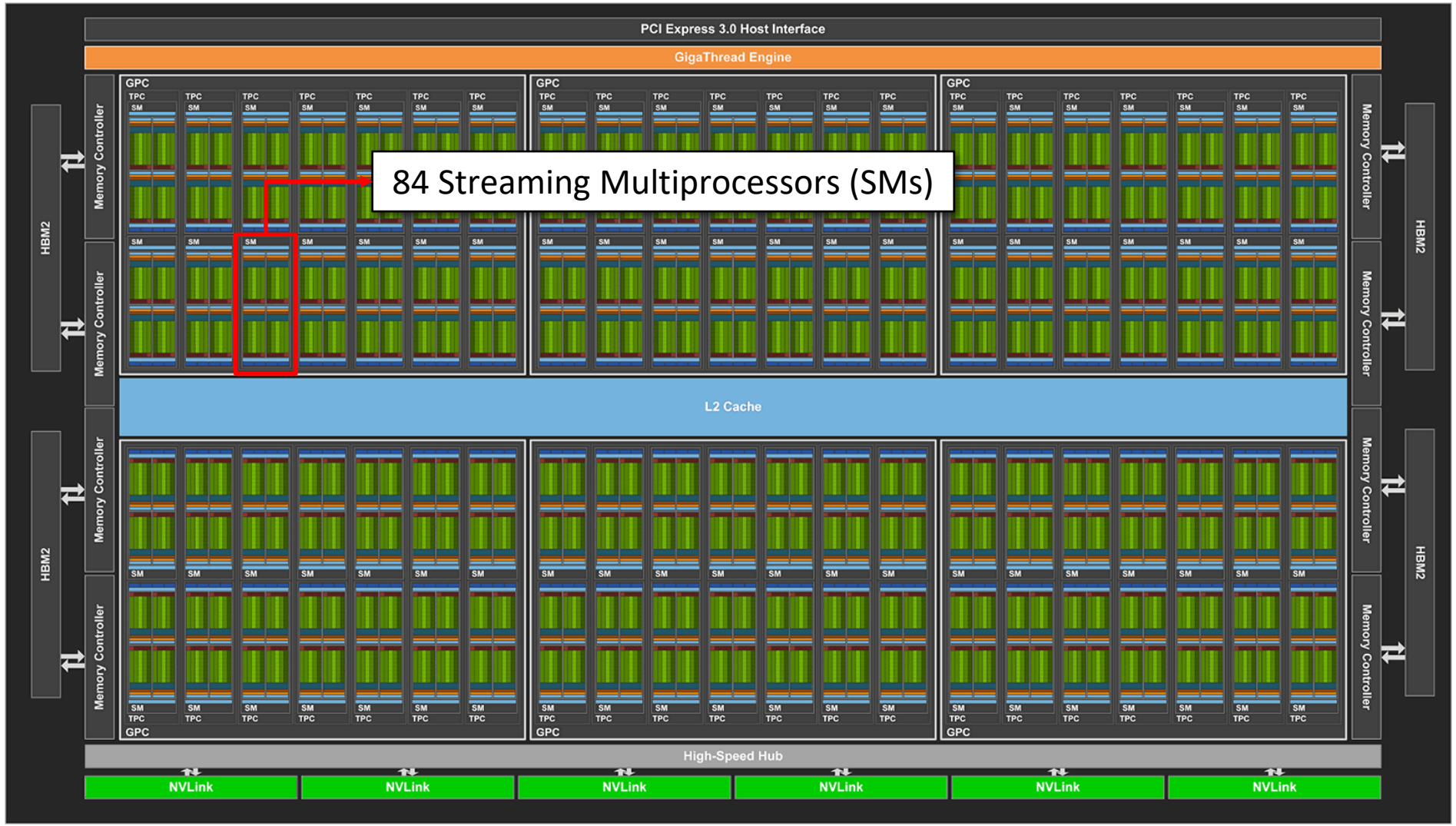


CPU	GPU
Large memory	Relatively small memory
Each core has its own independent control logic Allows independent execution	Groups of cores share control logic Saves space & power
Coherent caches between cores	Shared cache & sync only within groups
Tuned for serial execution of independent work MIMD	Tuned 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

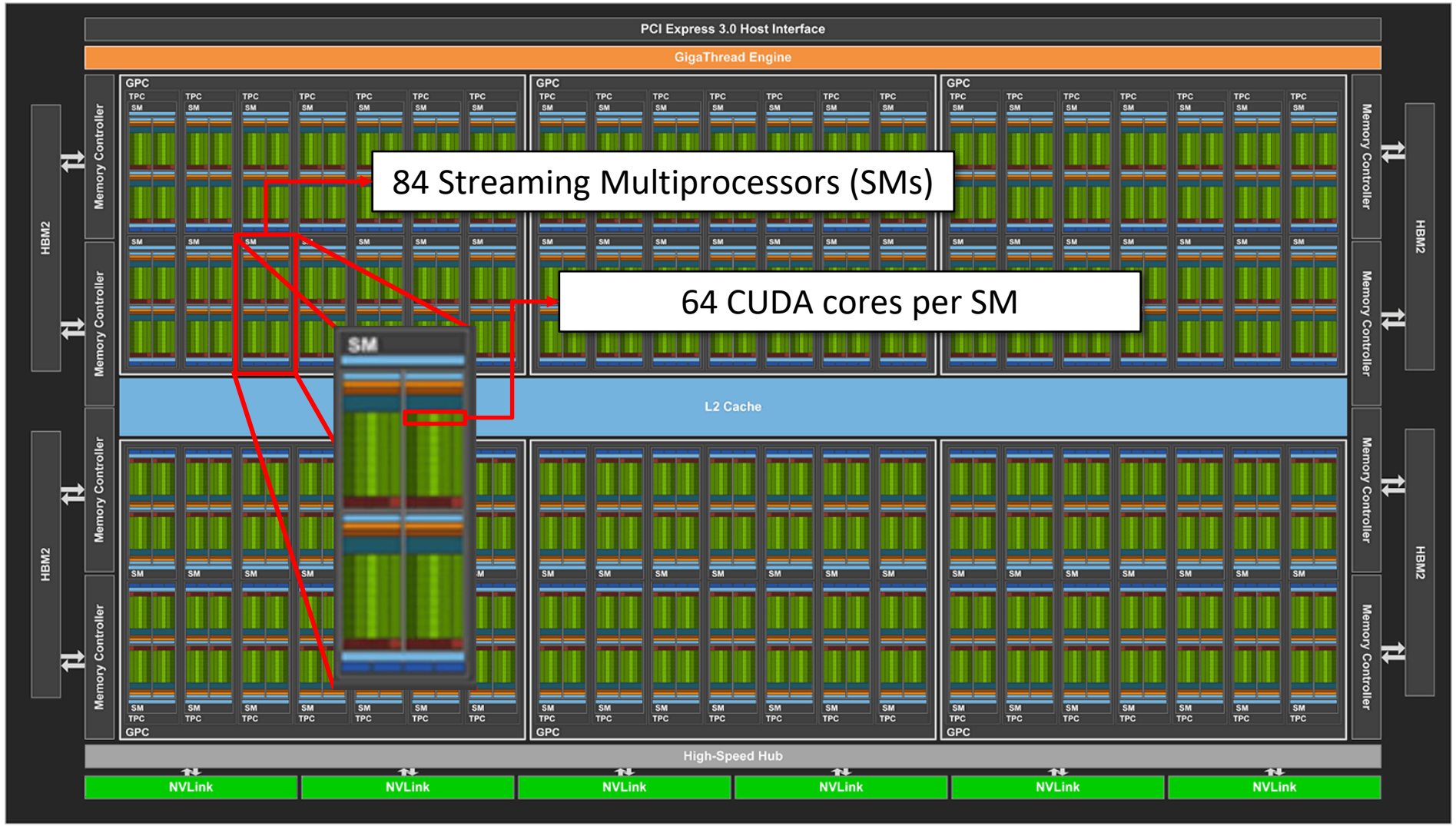
Volta (2017)


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

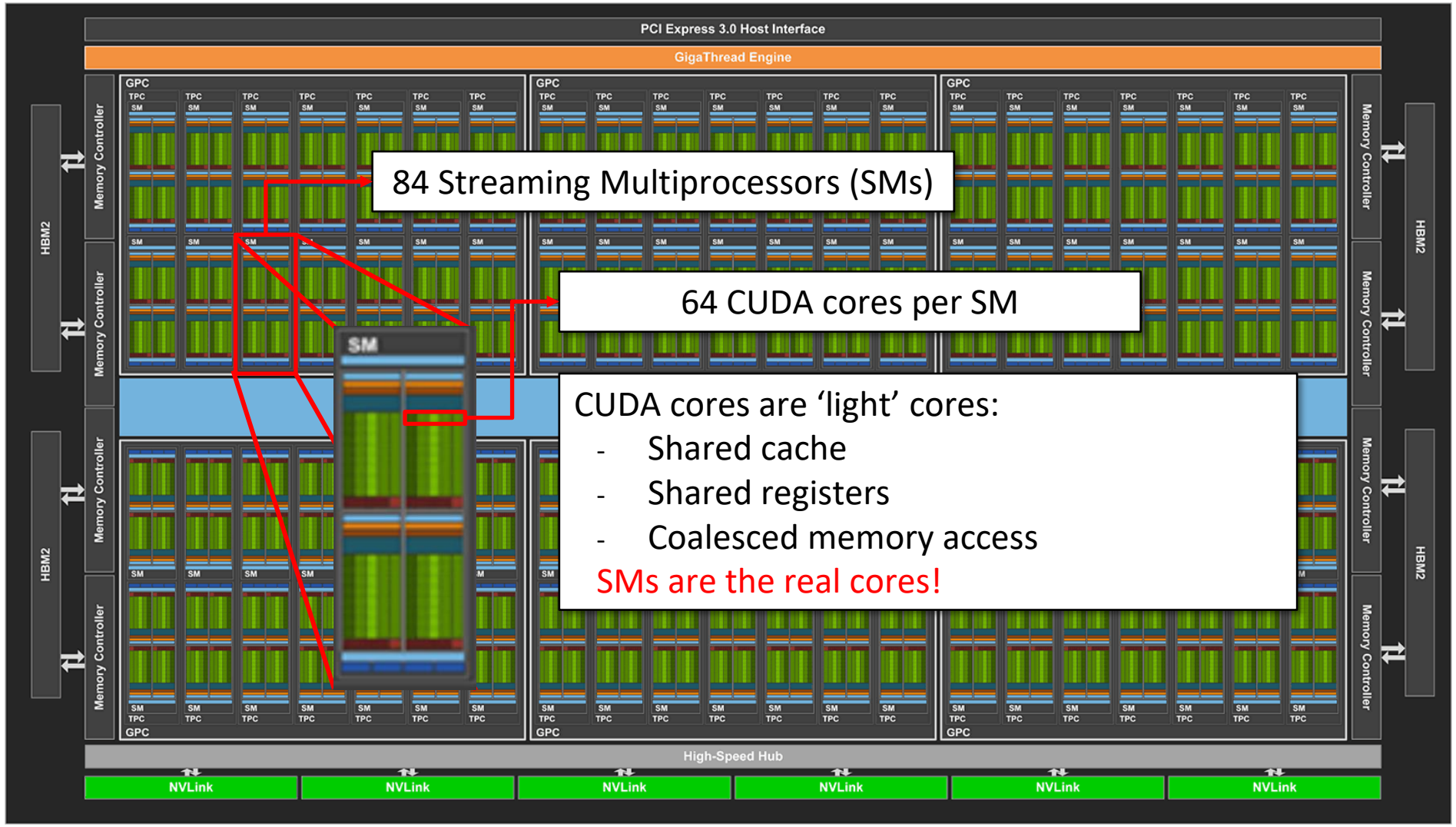
Volta (2017)


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

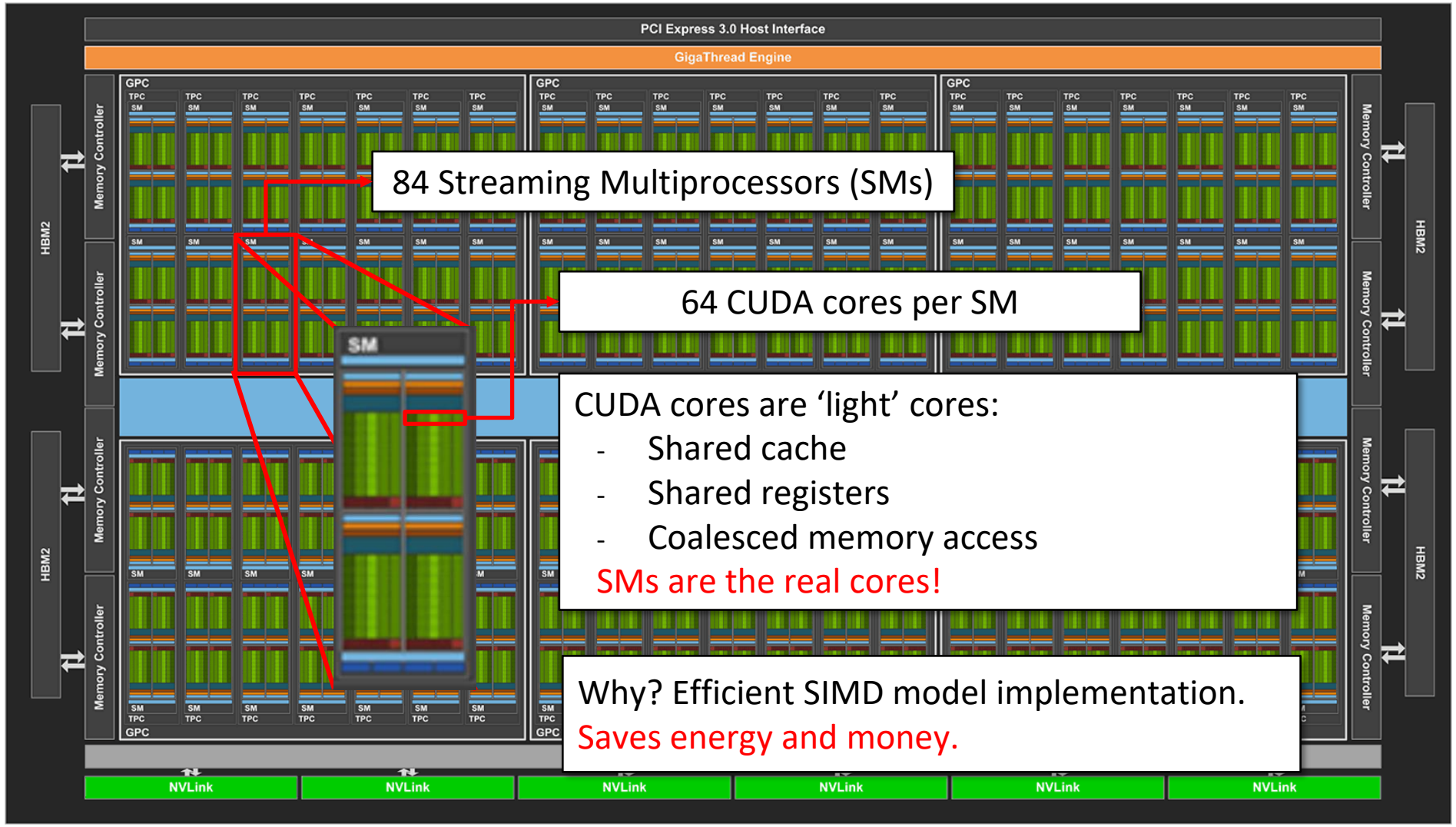
Volta (2017)


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

Volta (2017)

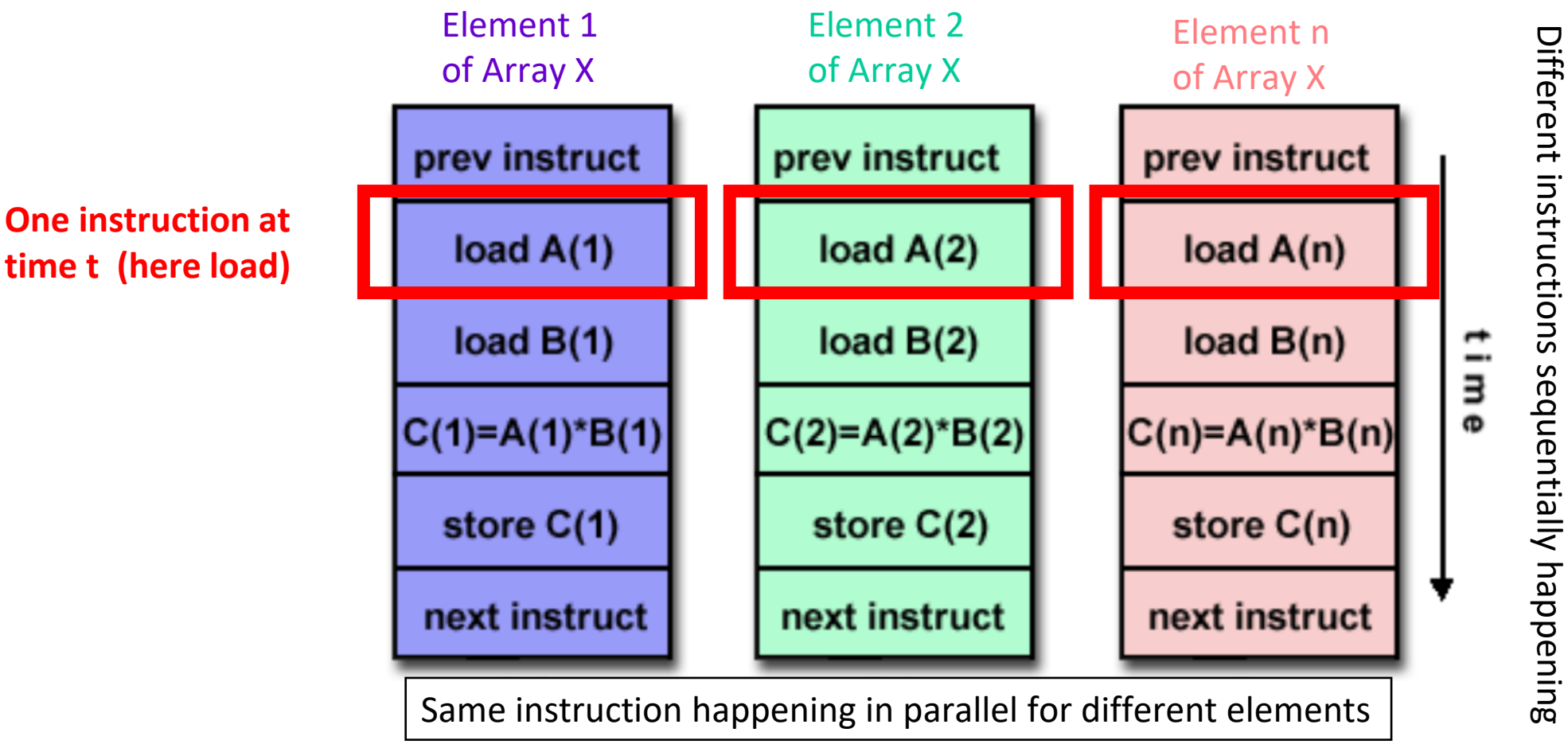

PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
Memory Controller
HBM2
84 Streaming Multiprocessors (SMs)
64 CUDA cores per SM
CUDA cores are 'light' cores:
- Shared cache
- Shared registers
- Coalesced memory access
SMs are the real cores!
High-Speed Hub
NVLink

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.

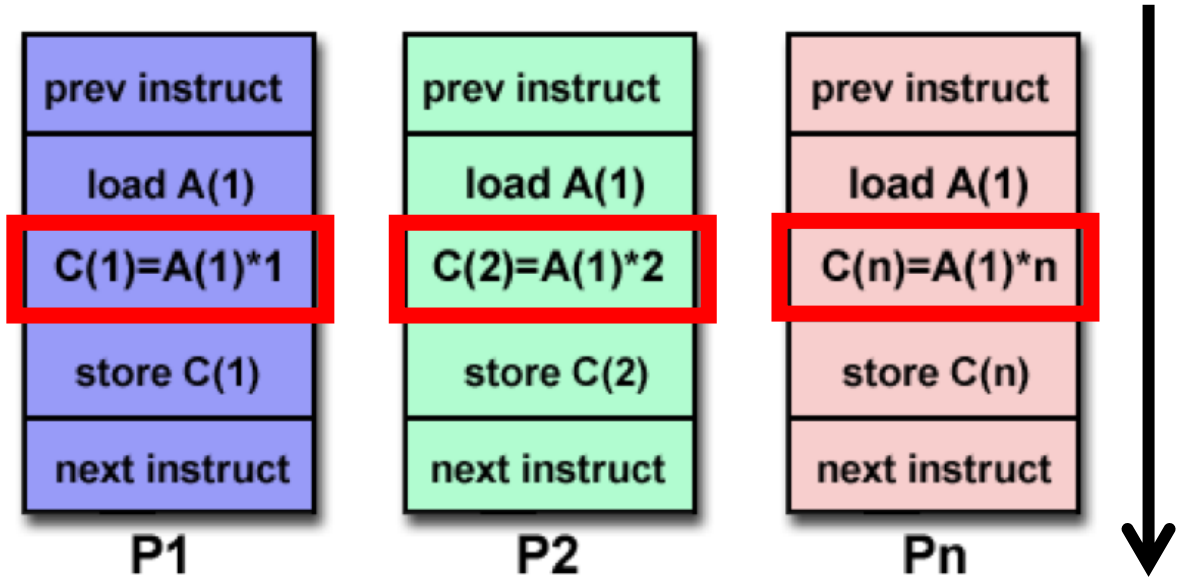
GPUs are SIMD oriented architectures

SIMD : Single Instruction Multiple Data



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

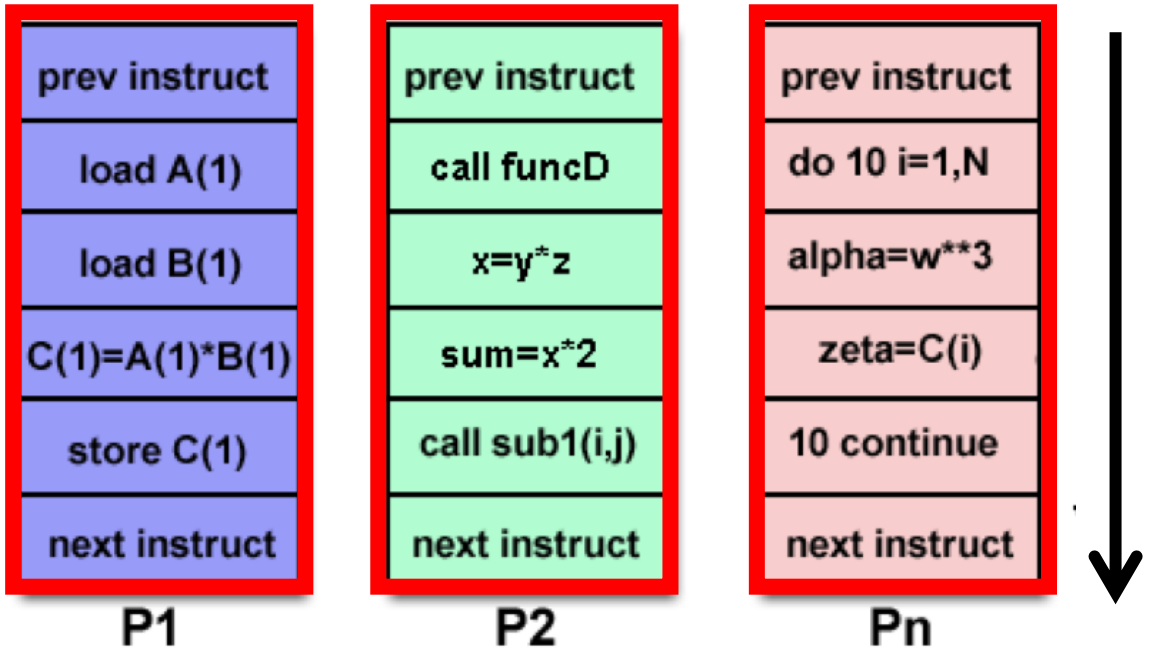
MISD : Multiple Instructions Single Data



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

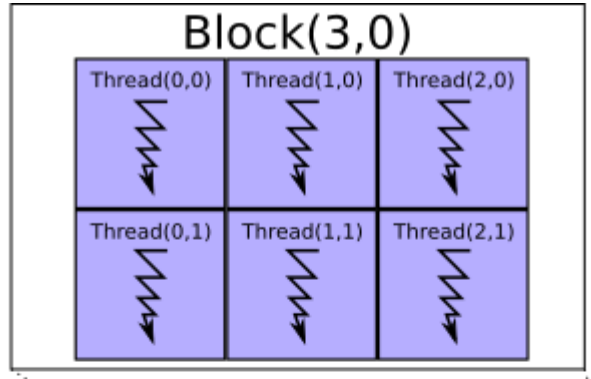
Mapping the hardware architecture into the programming architecture

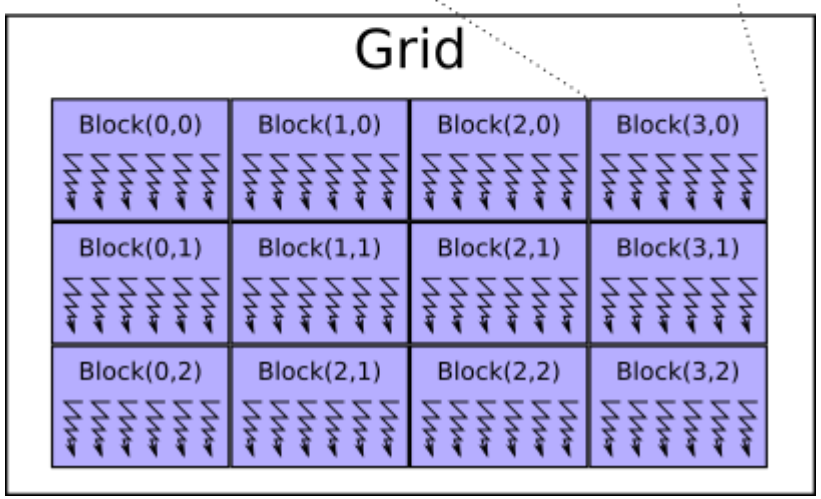
As a GPU programmer you will :

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

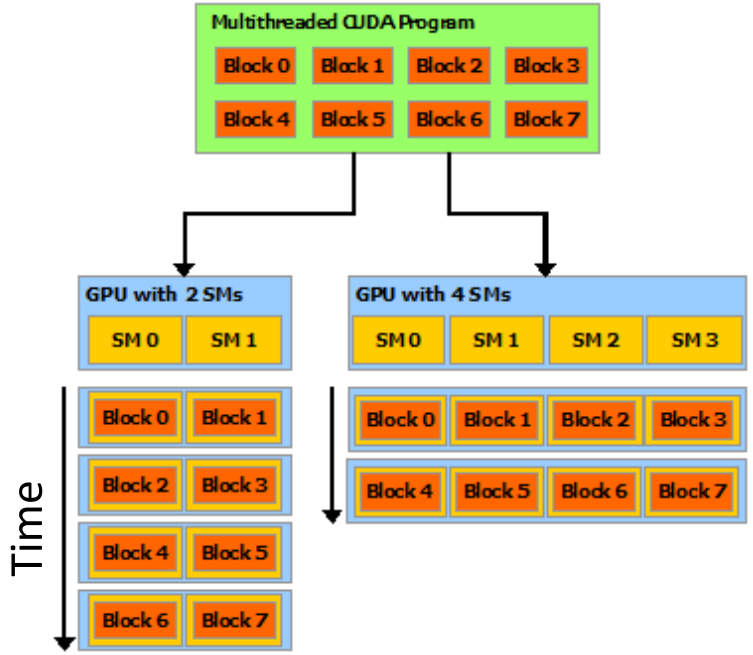
  (block = doing the same operation in parallel on a subset of elements of your array).

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





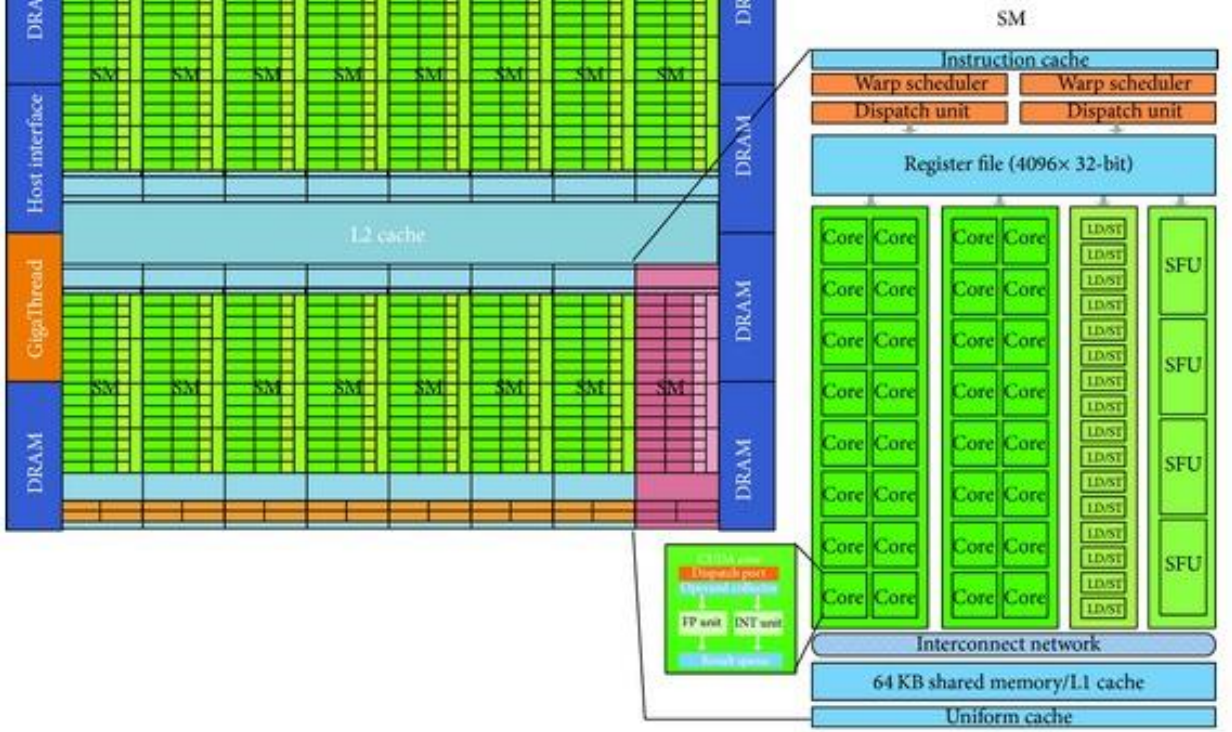
Warp: Minimum amount of threads (32) that will be executed simultaneously per SM



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)

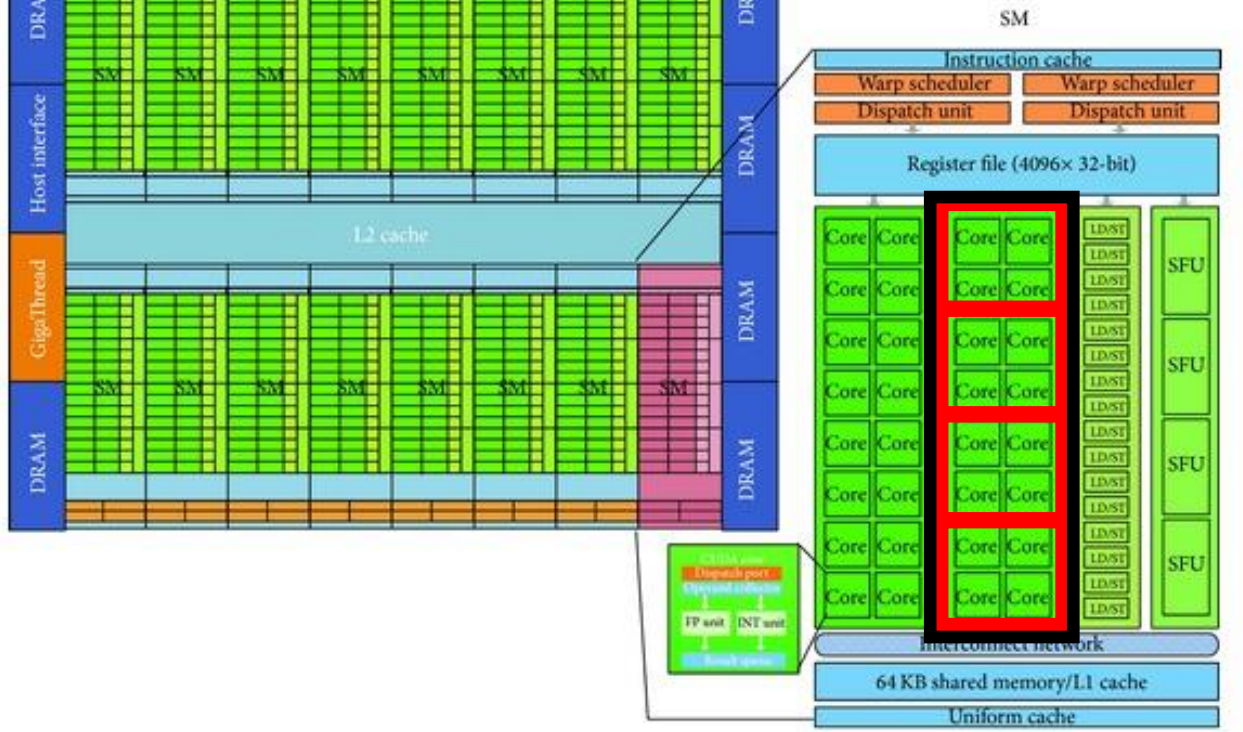


Here it is not a core, it is a "CUDA core" so not directly comparable to the CPU core

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

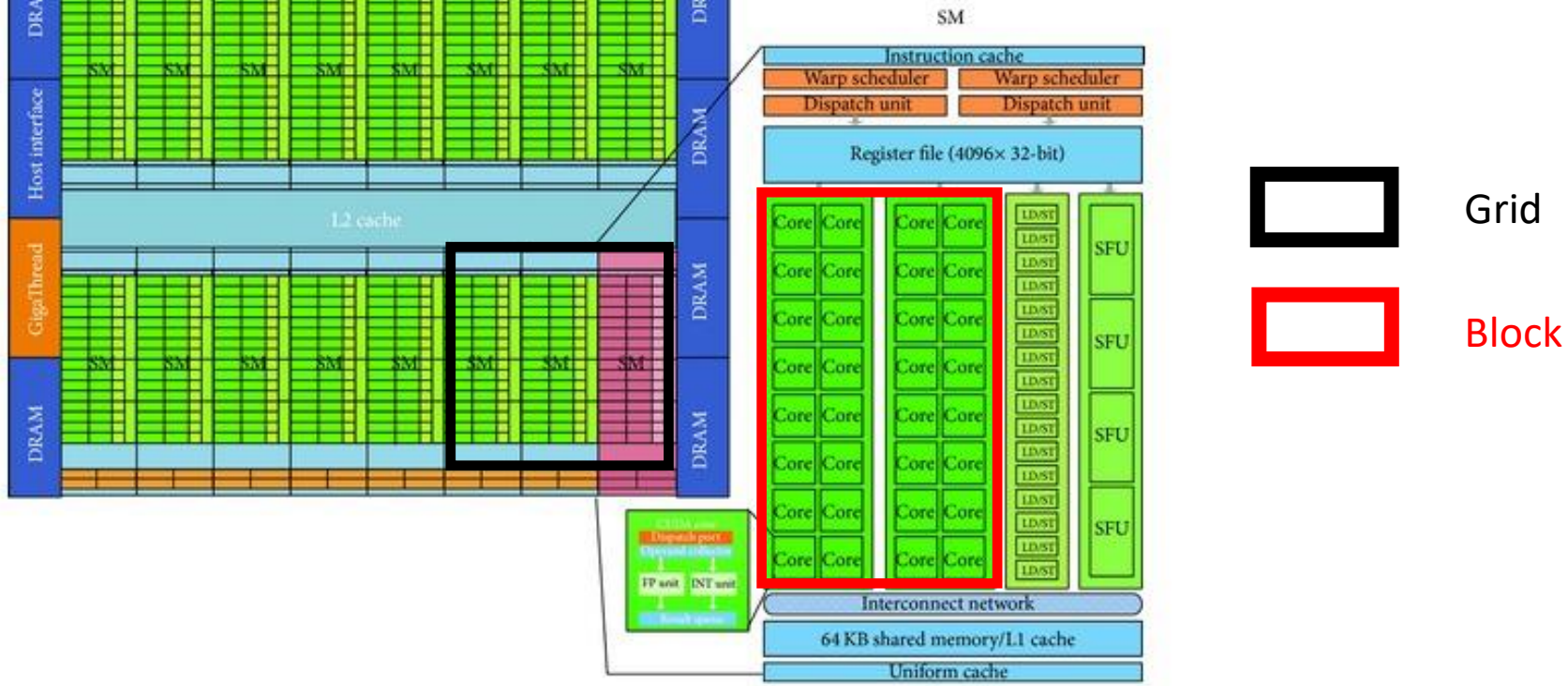
Grid

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

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

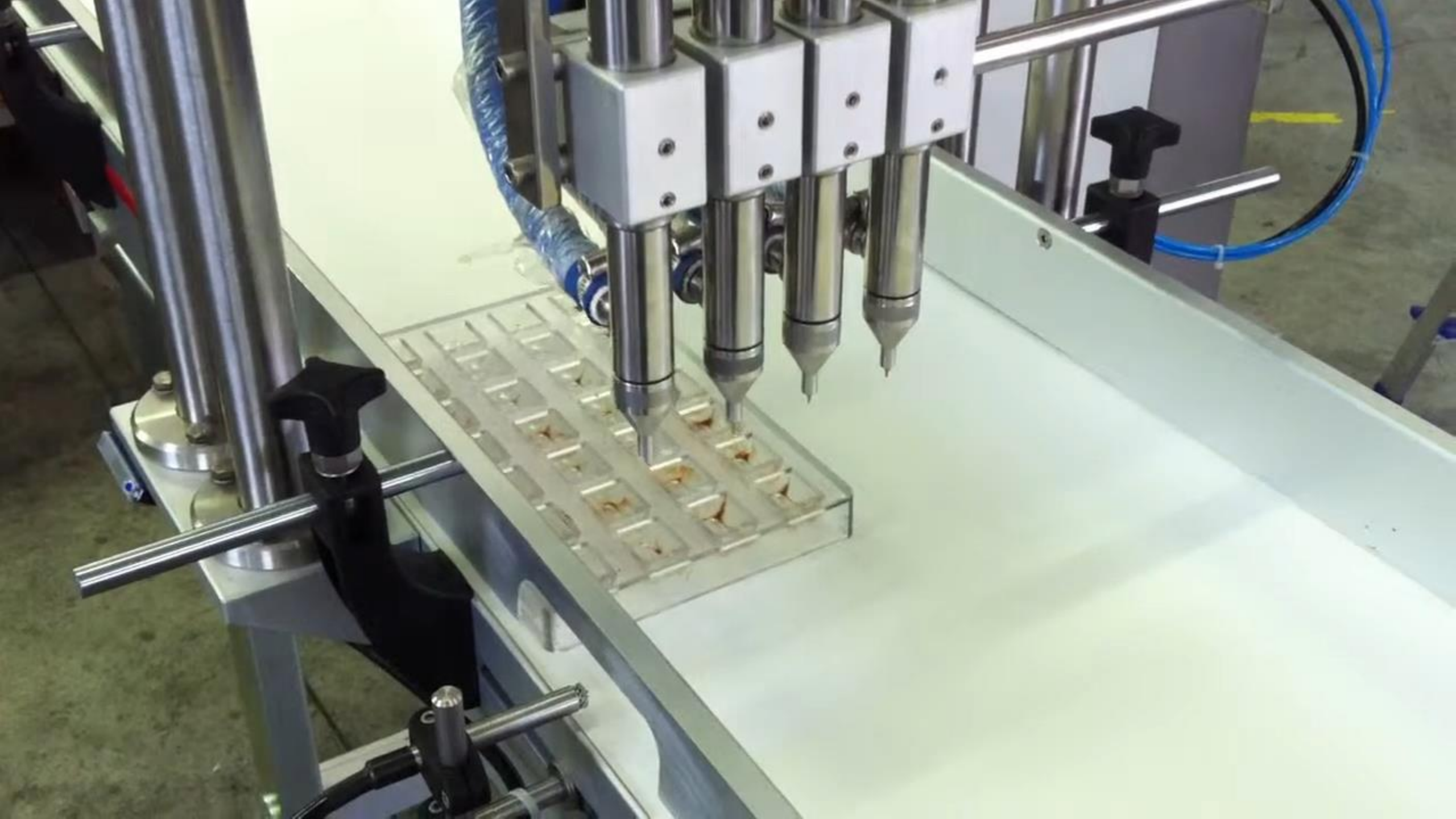
Grid

Block

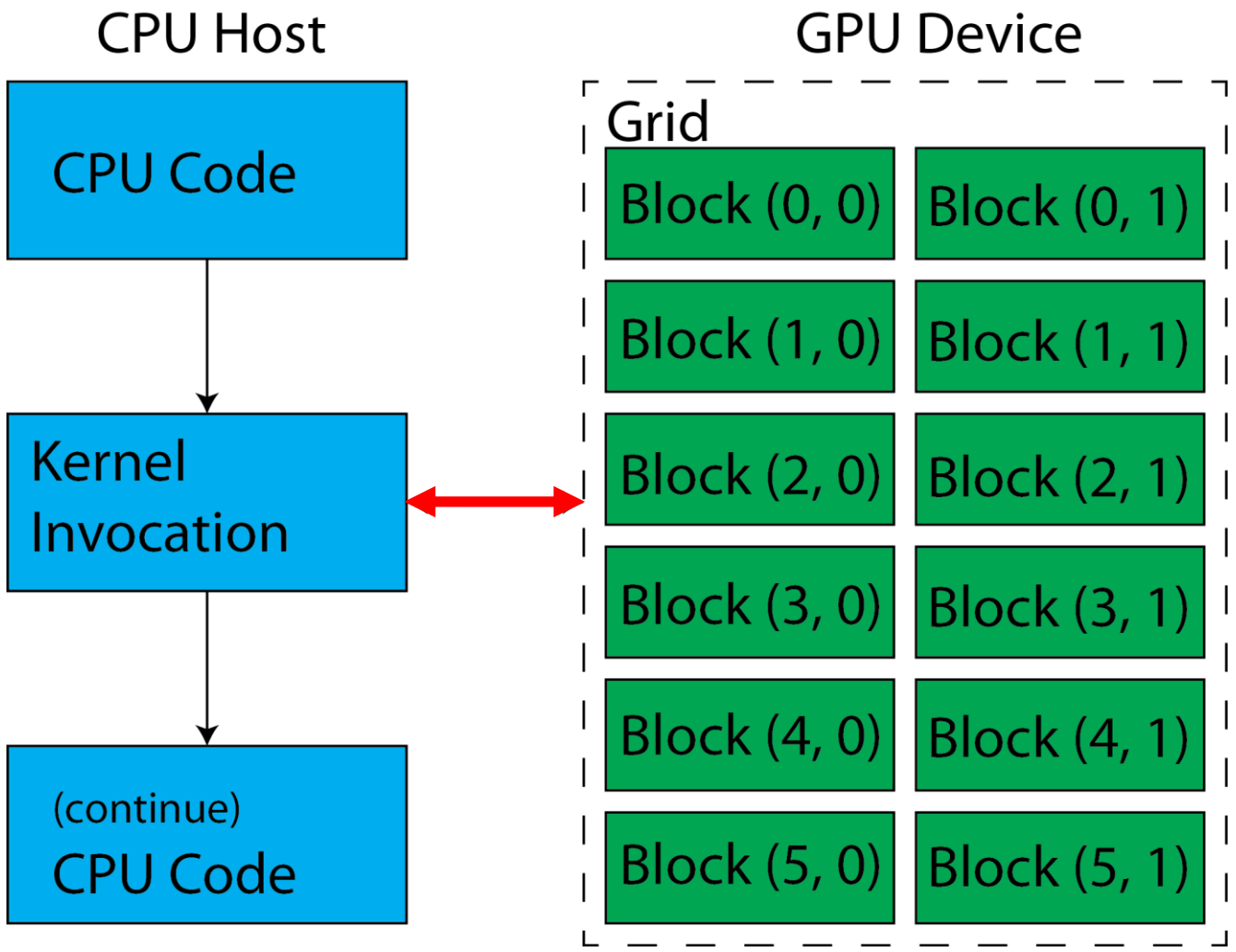
A grid can encompass many SM but a block can not be larger than a SM since a block is a shared memory space.

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

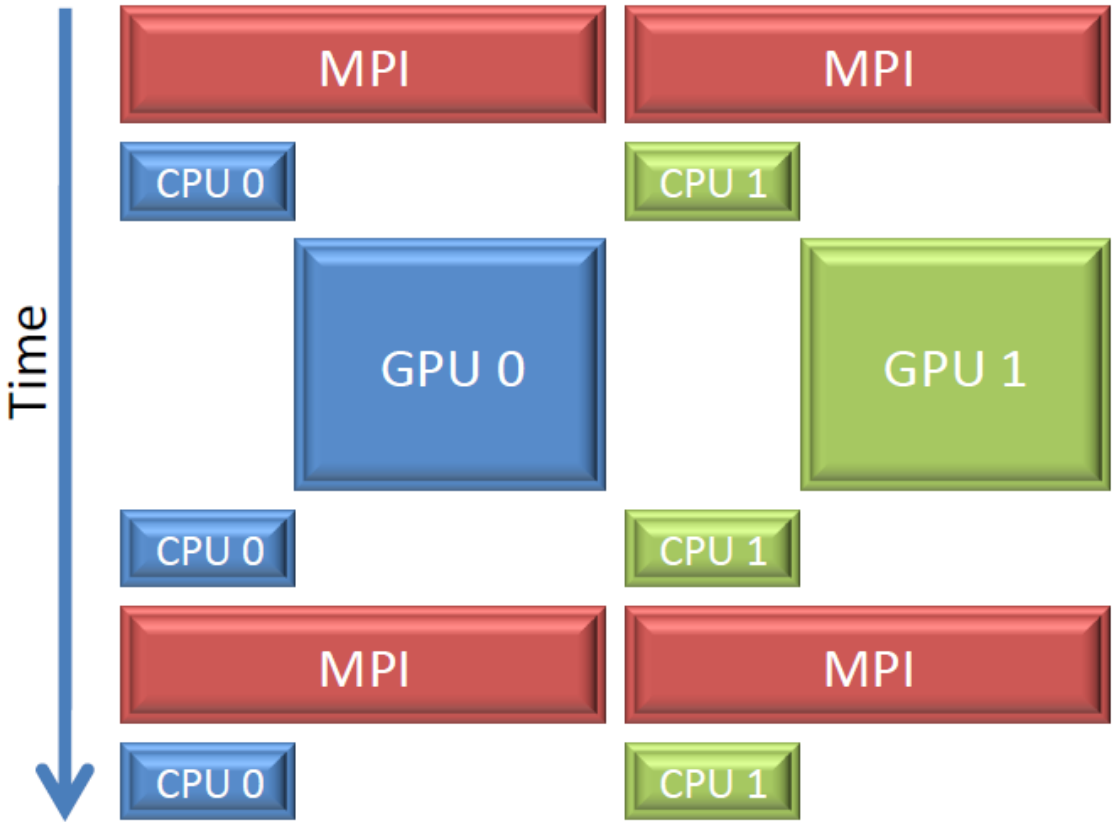


A mapping at the core of CPU/GPU communication: the kernel



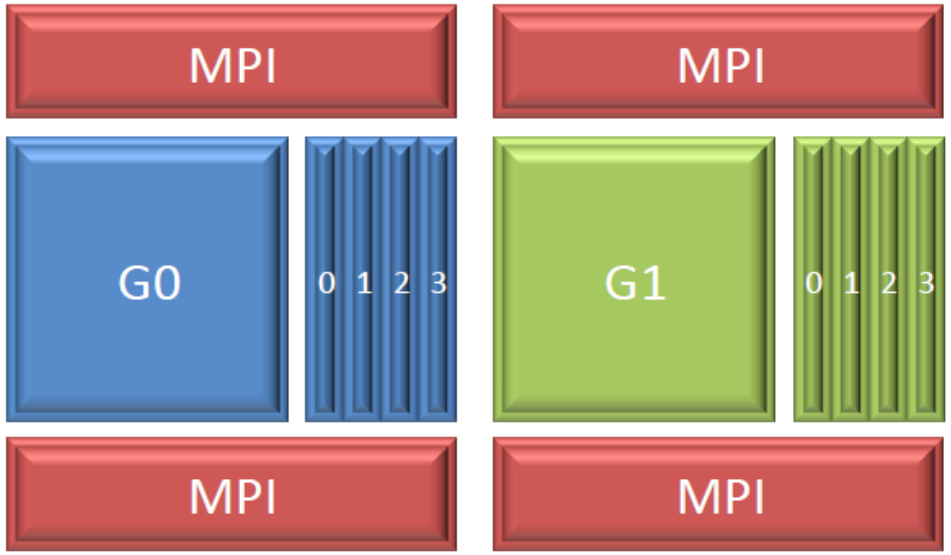
Source: https://cvw.cac.cornell.edu/GPU/structure

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

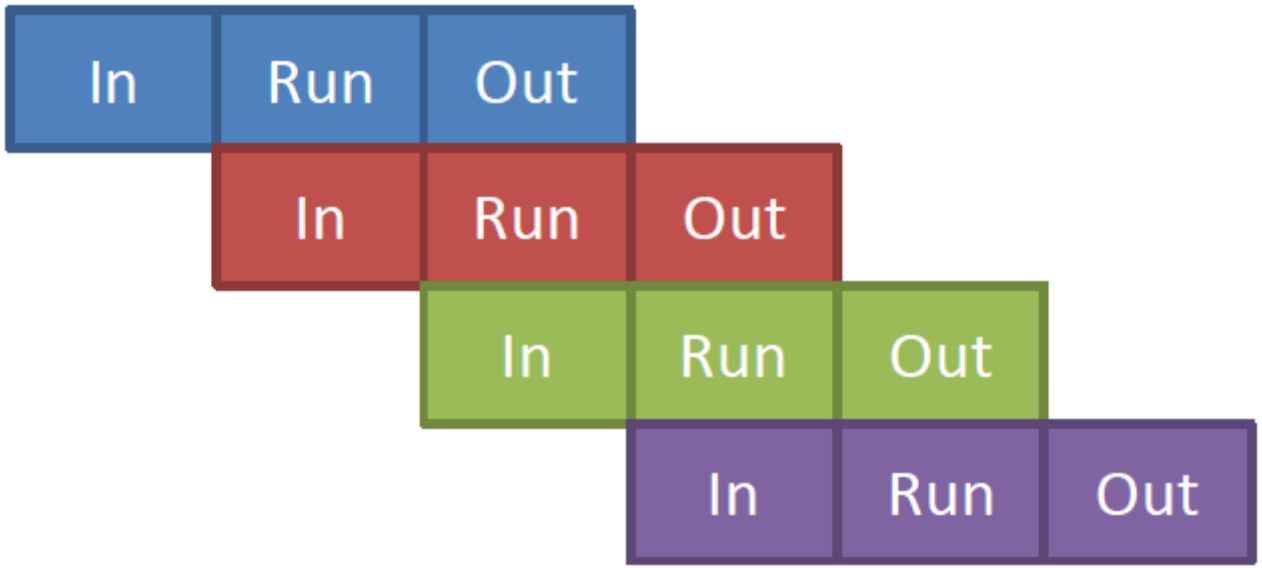
Most GPU operations are asynchronous from the CPU code

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

Source: Jeff Larkin NVIDIA

Synchronous execution (1 stream)

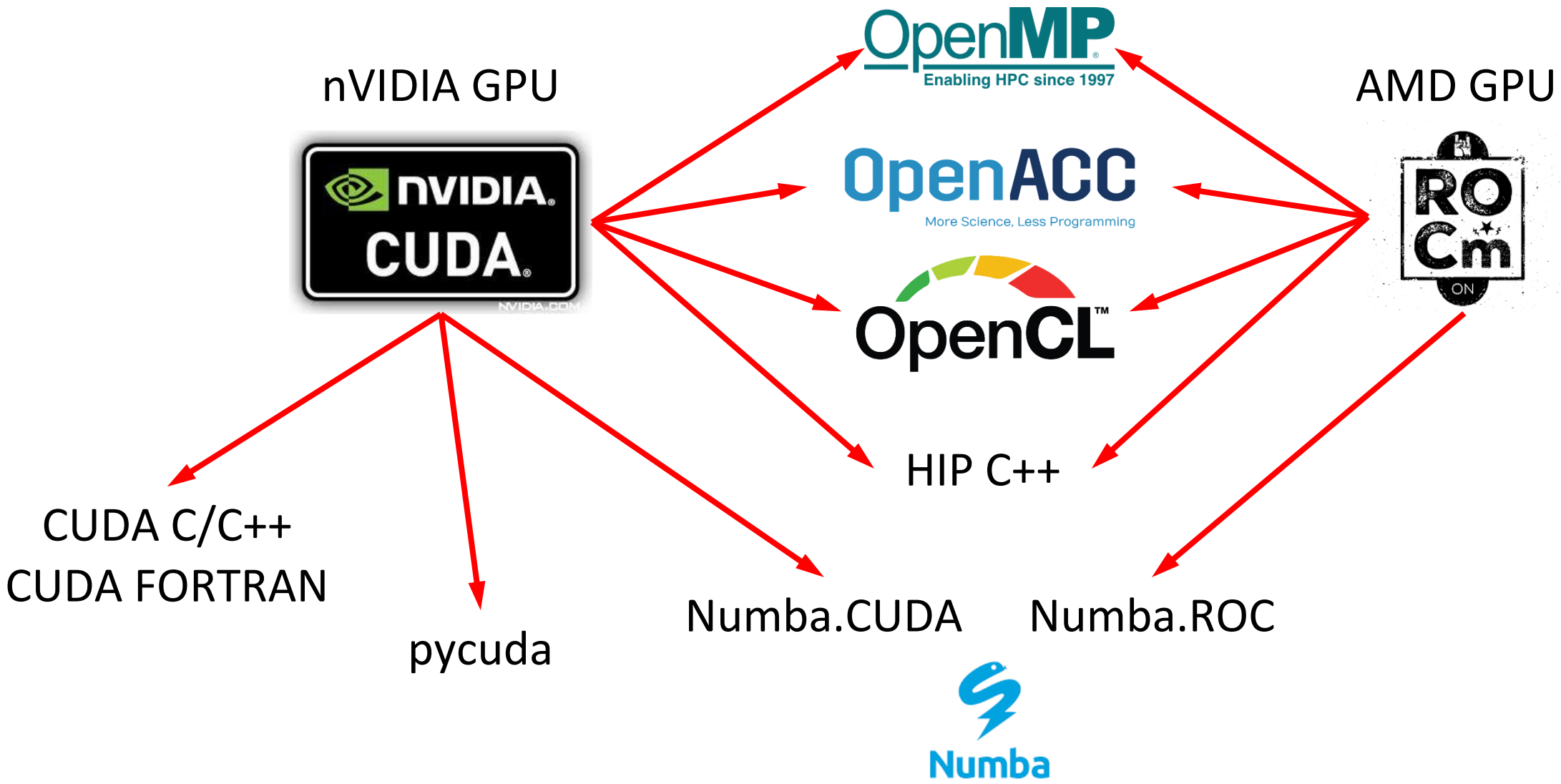
Asynchronous execution (3 streams)



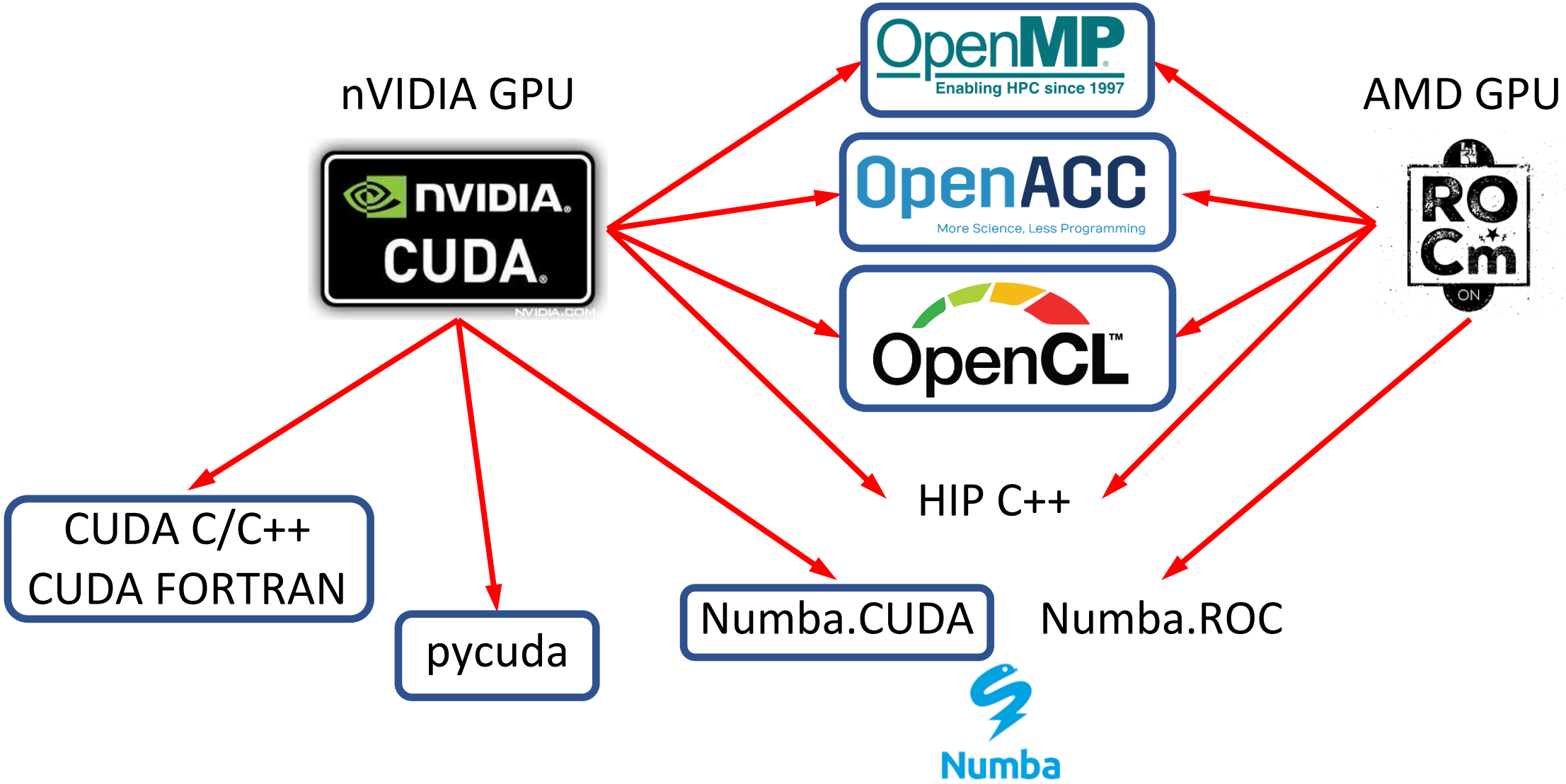
Asynchronous execution helps to hide the data transfer costs

Source: Jeff Larkin NVIDIA

Writing your kernel : GPU programming paradigms



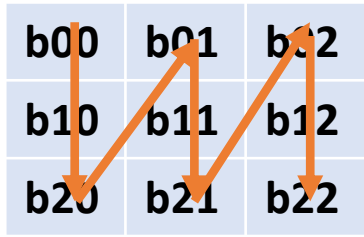
Writing your kernel : GPU programming paradigms



Sub-optimal memory access will kill the performance…

The less efficient way (column-major):

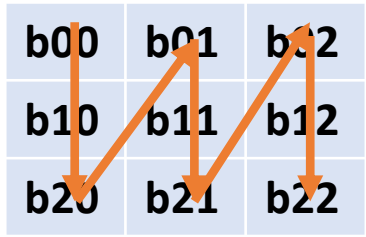
```
for(int i=0; i<n; ++i)
   for(int j=0; j<n; ++j)
      for(int k=0; k<n; ++k)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



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

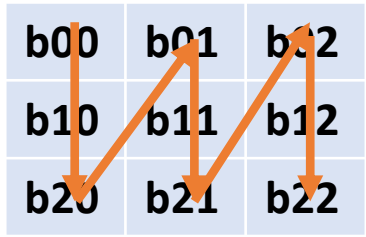
a00	a01	a02	a10	a11	a12	a20	a21	a22
b00	b01	b02	b10	b11	b12	b20	b21	b22
c00	c01	c02	c10	c11	c12	c20	c21	c22

i	j	k
0	0	0

Sub-optimal memory access will kill the performance…

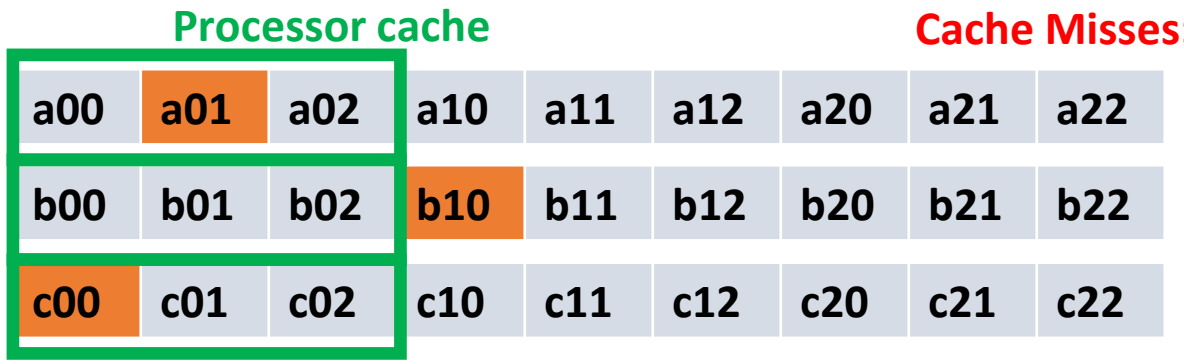
The less efficient way (column-major):

```
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 …

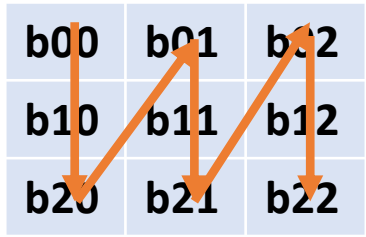


i	j	k
0	0	1

Sub-optimal memory access will kill the performance…

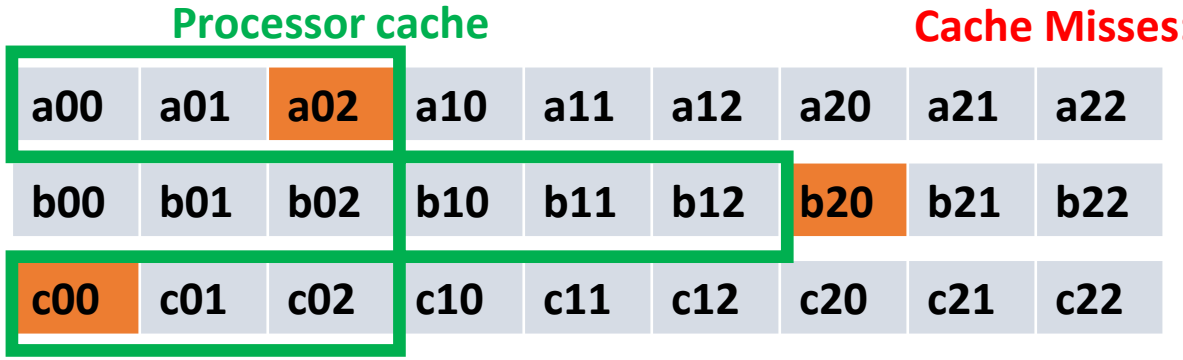
The less efficient way (column-major):

```
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 …

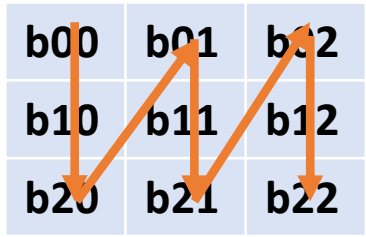


i	j	k
0	0	2

Sub-optimal memory access will kill the performance…

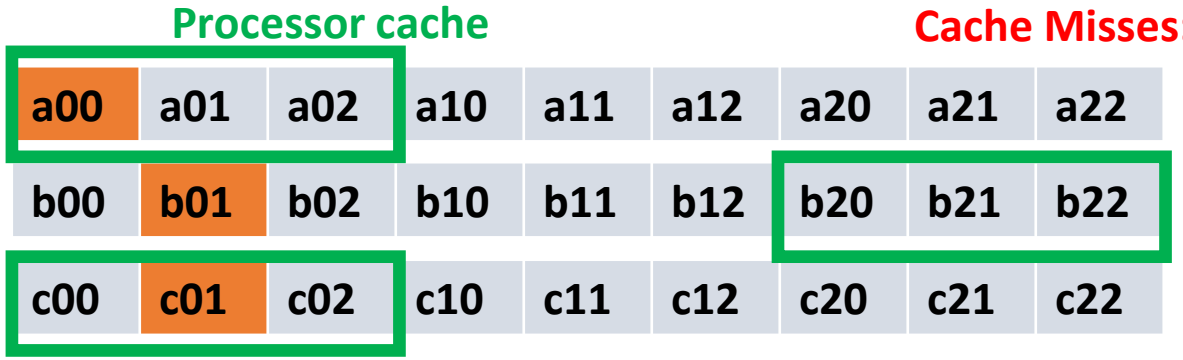
The less efficient way (column-major):

```
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 …

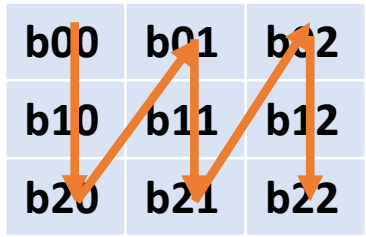


i	j	k
0	1	0

Sub-optimal memory access will kill the performance…

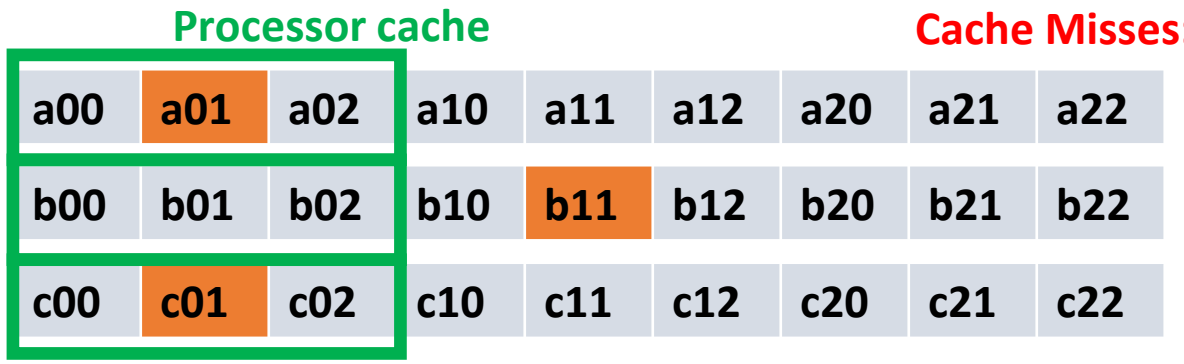
The less efficient way (column-major):

```
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 …

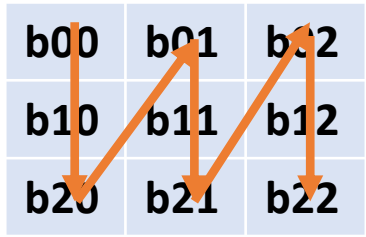


i	j	k
0	1	1

Sub-optimal memory access will kill the performance…

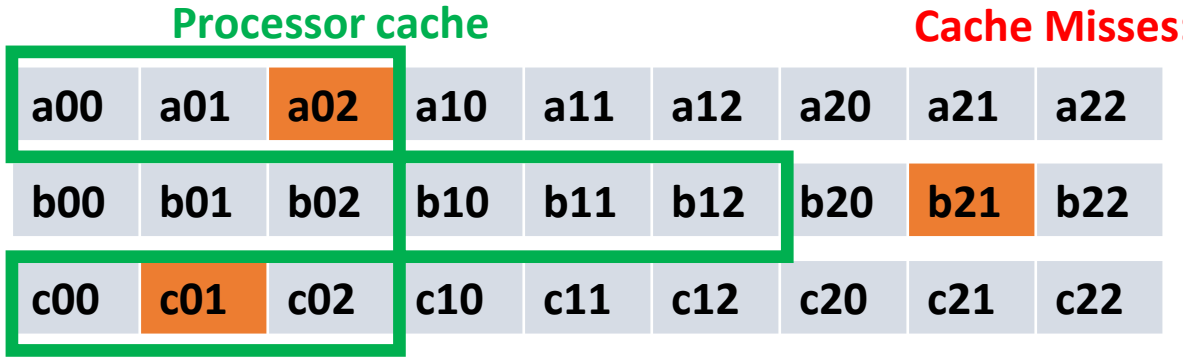
The less efficient way (column-major):

```
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 …

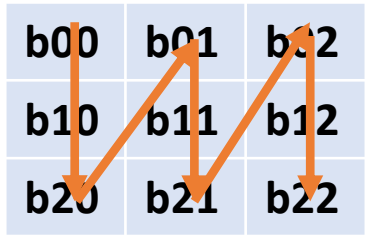


i	j	k
0	1	2

Sub-optimal memory access will kill the performance…

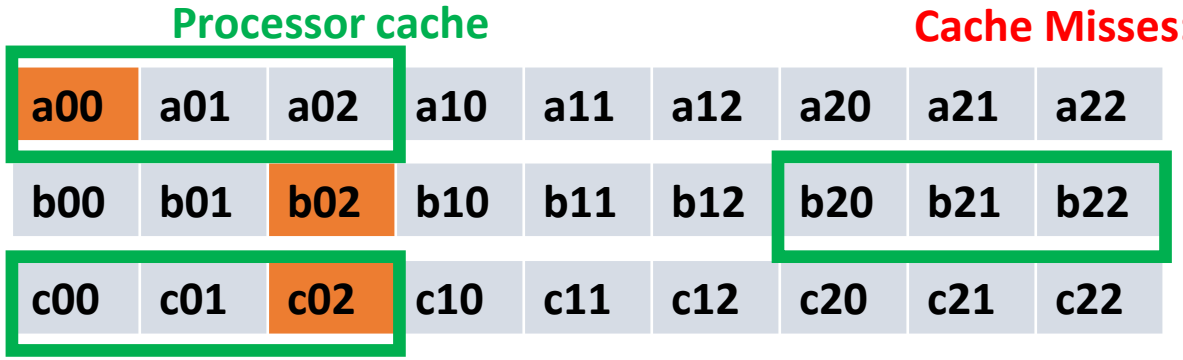
The less efficient way (column-major):

```
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 …

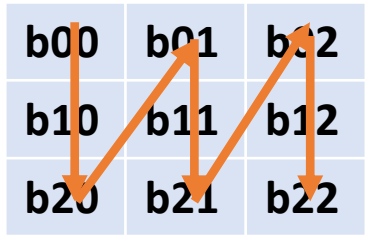


i	j	k
0	2	0

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

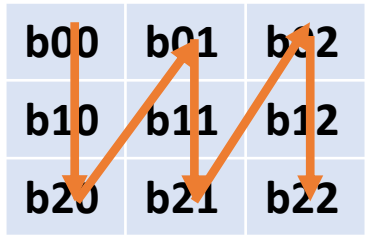
a00	a01	a02	a10	a11	a12	a20	a21	a22
b00	b01	b02	b10	b11	b12	b20	b21	b22
c00	c01	c02	c10	c11	c12	c20	c21	c22

i	j	k
0	2	1

Sub-optimal memory access will kill the performance…

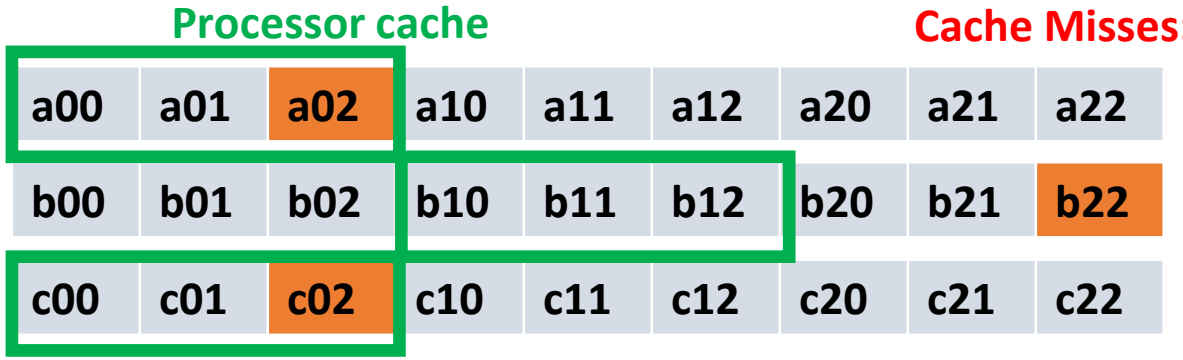
The less efficient way (column-major):

```
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 …

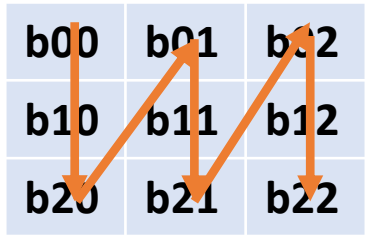


i	j	k
0	2	2

Sub-optimal memory access will kill the performance…

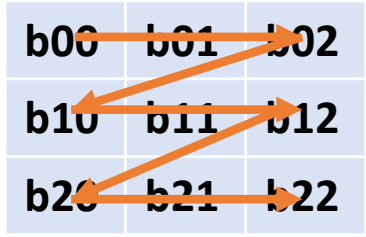
The less efficient way (column-major):

```
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!

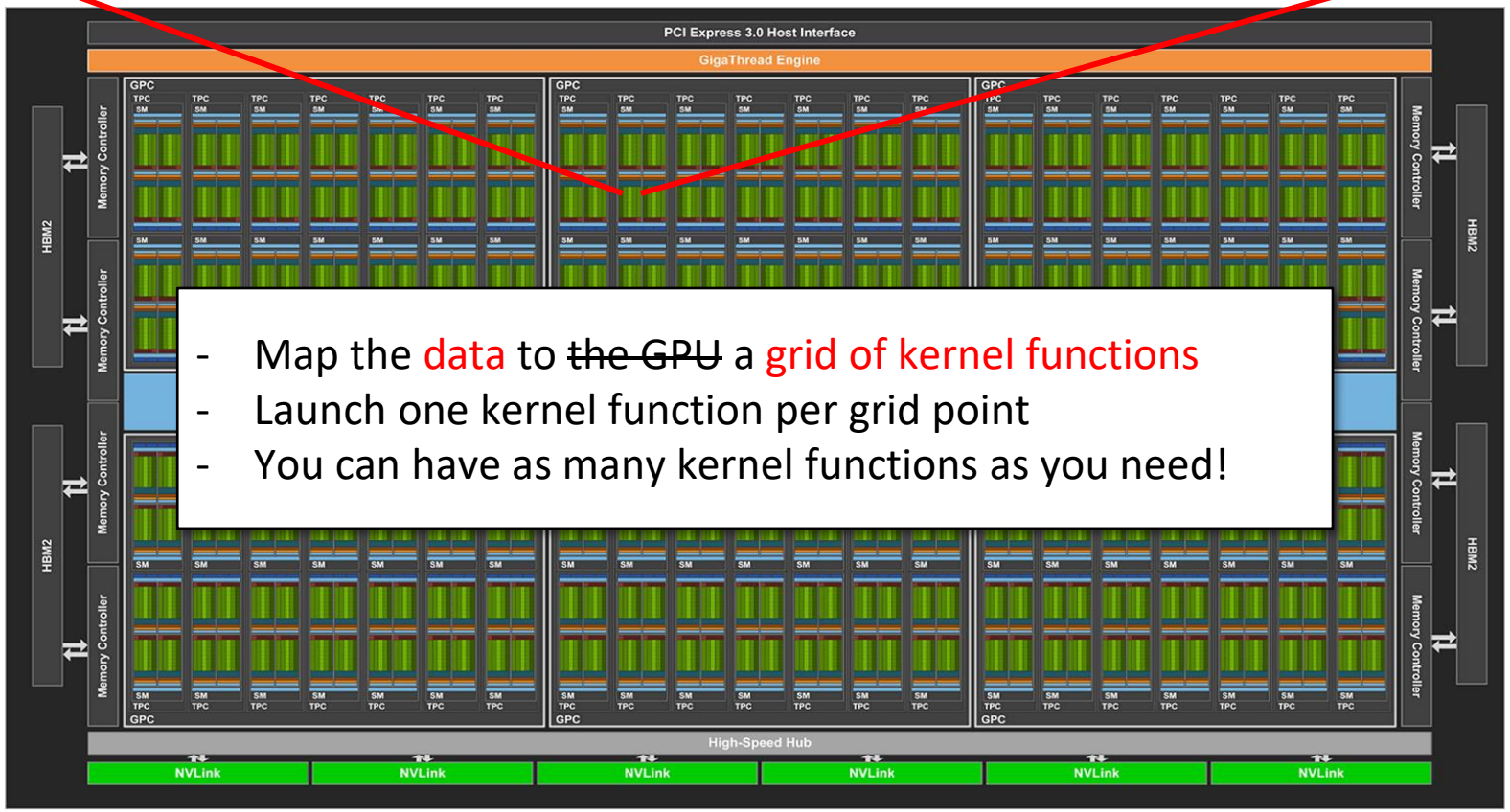
Remember, the data is not sent to individual threads

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];
}
```

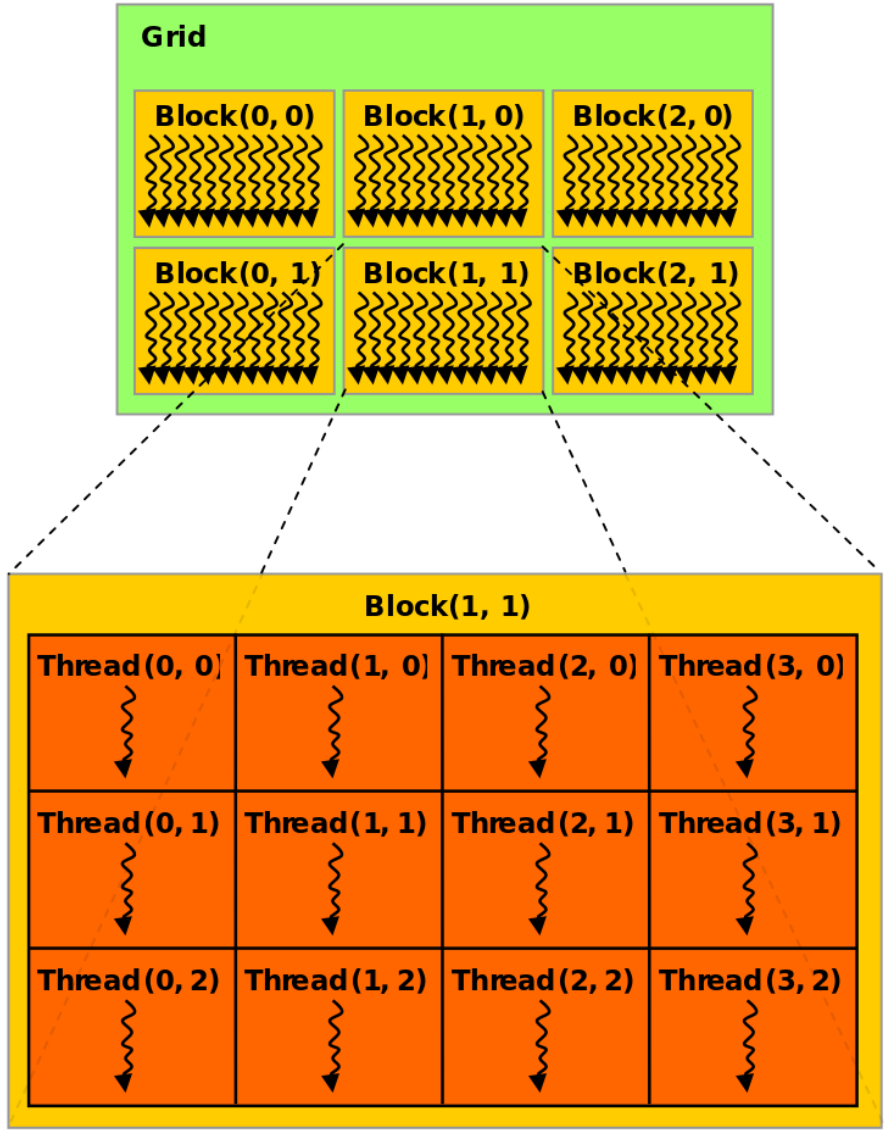
CUDA Kernel



- 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!

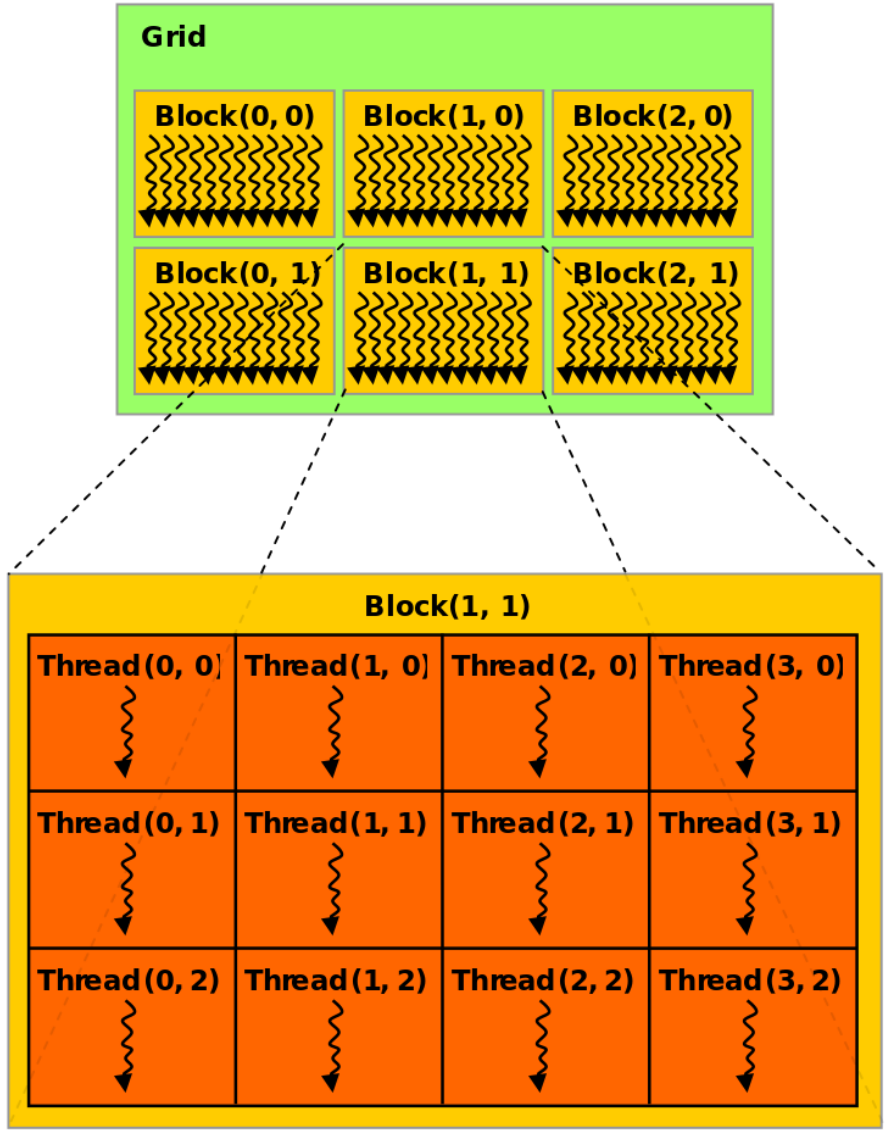
Reminder: Grids, blocks, threads and warps

- Grid: has 1 to as many blocks as you want in 1D, 2D, or 3D
- Block: has 1 to 1024 threads
- Blocks are mapped to 1 or more **warps**.
- Warps always execute 32 threads inside a SM.
- All the 32 threads in a warp execute the same instruction

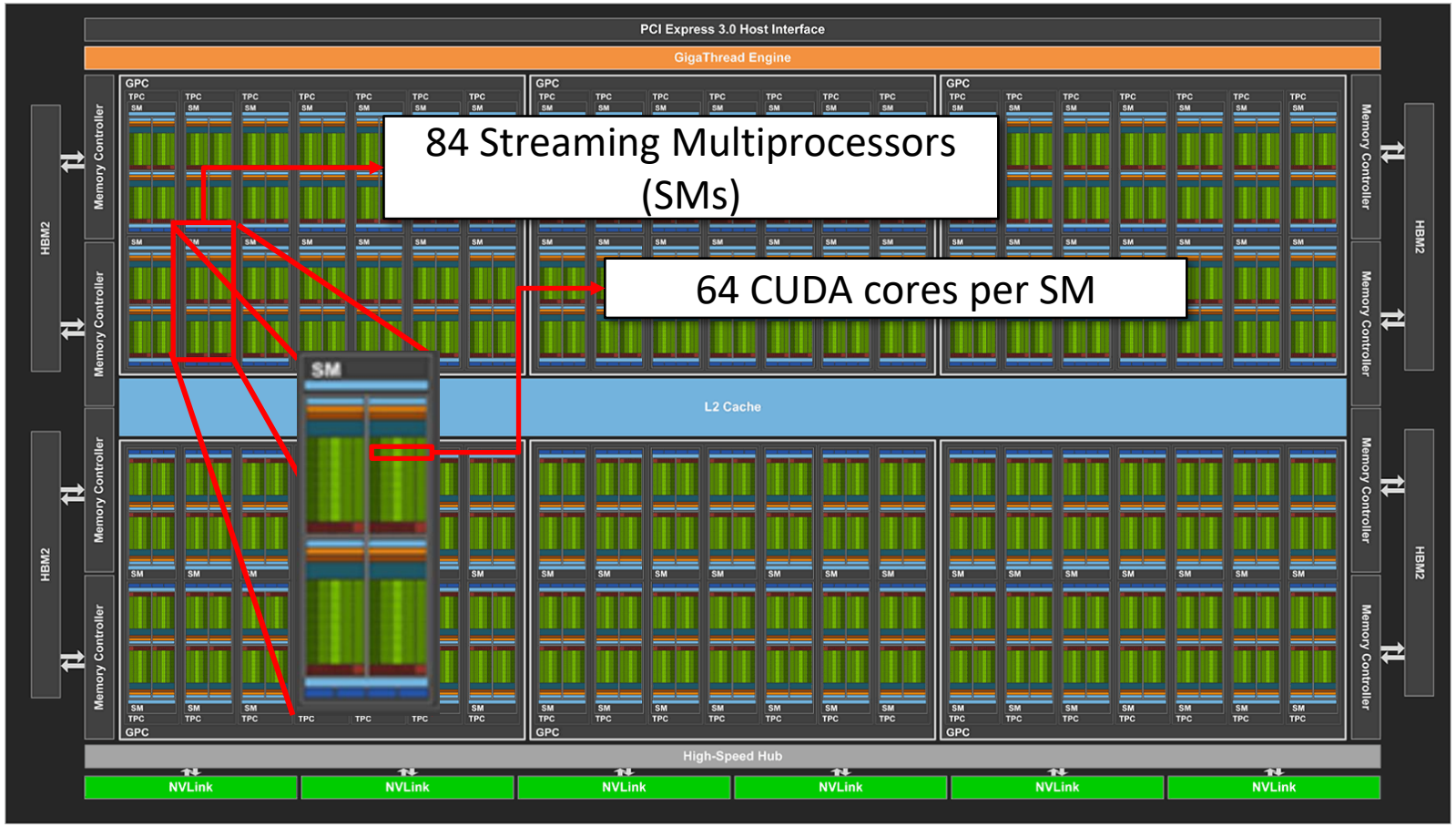


Reminder: Grids, blocks, threads and warps

- The threads inside a block are selected serially by the Streaming Multiprocessor.
- When you launch a grid containing a single block with one thread, you launch 1 warp. This warp contains 31 "dummy" threads which are masked off, and a single live thread. If you launch a single block with two threads, you still launch 1 warp, but now the single warp contains 2 active threads.
- When you launch two blocks containing a single thread each, it results in two warps, each of which contains 1 active thread. Because all scheduling and execution is done on a per warp basis, you now have two separate entities (warps) which the hardware can schedule and execute independently. This allows more latency hiding and less instruction pipeline stalls, and the code runs faster as a result.

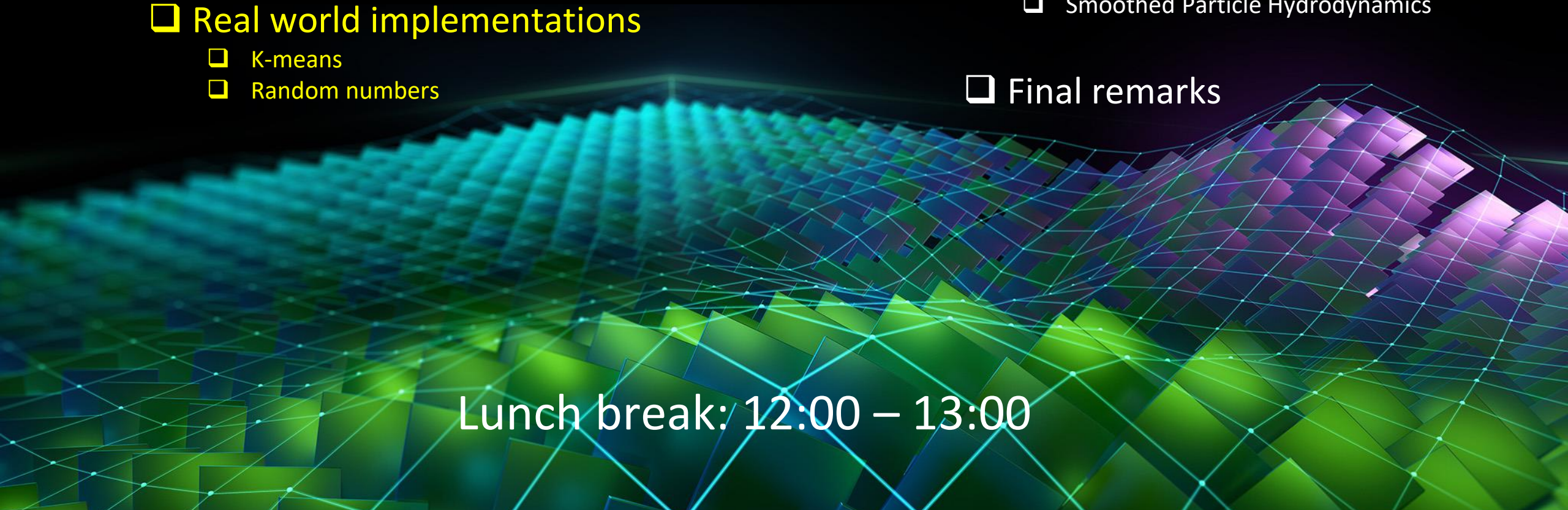


- All the 32 threads in a warp execute the same instruction
- All the warps inside a SM must execute the same instruction.
- **Example 1:** A block with 32 threads fits in a single warp. The SM can execute up to 64 CUDA cores. No other block/warp can execute inside that SM. Max occupancy is 50%.
- **Example 2**: A block with 64 threads fits in a single warp. The SM can execute both warps simultaneously. Max occupancy is 100%!



Outline

- ❑ Introduction
- ❑ Example
  - ❑ Matrix Multiplication + Exercise
- ❑ Real world implementations
  - ❑ K-means
  - ❑ Random numbers
- ❑ Common pitfalls
- ❑ Memory management
  - ❑ Image processing example
- ❑ Monitoring and asynchronicity
  - ❑ Smoothed Particle Hydrodynamics
- ❑ Final remarks

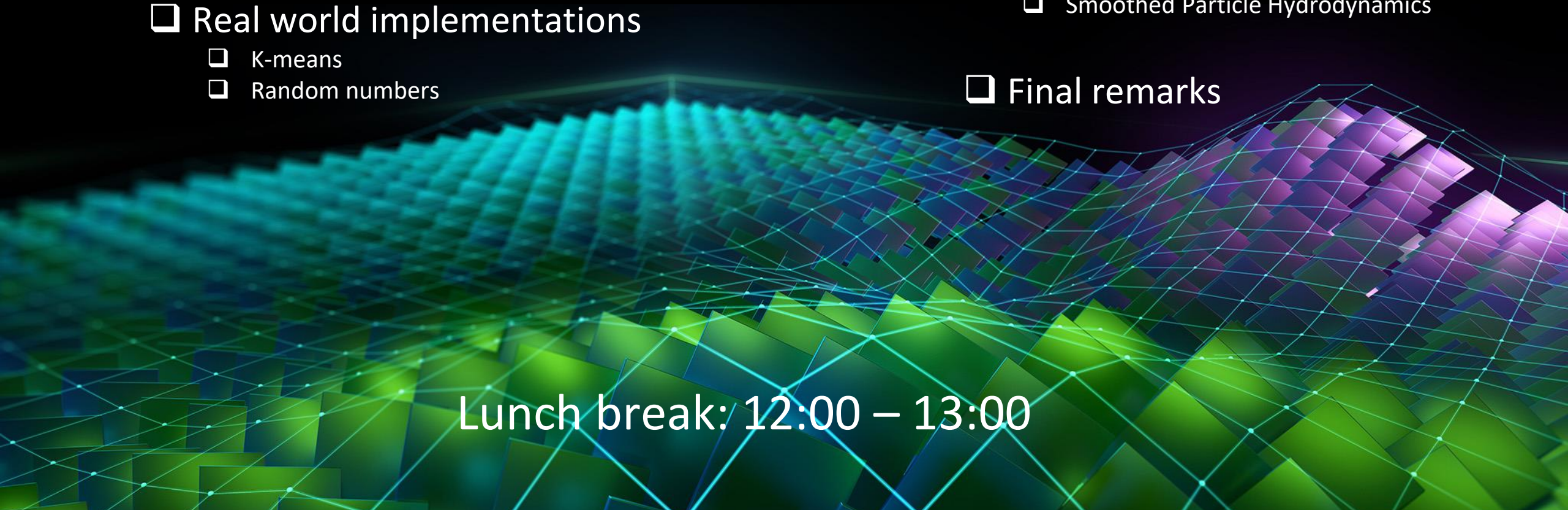


Lunch break: 12:00 – 13:00


Resume at:
13:00

Outline

- ❑ Introduction
- ❑ Example
  - ❑ Matrix Multiplication + Exercise
- ❑ Real world implementations
  - ❑ K-means
  - ❑ Random numbers
- ❑ Common pitfalls
- ❑ Memory management
  - ❑ Image processing example
- ❑ Monitoring and asynchronicity
  - ❑ Smoothed Particle Hydrodynamics
- ❑ Final remarks



Lunch break: 12:00 – 13:00

Warp divergence (aka Execution divergence)

Threads are executed in warps of 32, with all threads in the warp executing the same instruction at the same time.

What happens if different threads in a warp need to do different things?

This is called **warp divergence**. CUDA will generate correct code to handle this, but to understand the performance you need to understand what CUDA does with it.

All the threads in a warp execute both conditional branches

If the condition evaluate to false for a given thread, it will **remain idle** and wait for the other threads to finish.

=> **potentially large loss of performance.**

```
11    if (x < 0.0)
12        z = x-2.0;
13    else
14        z = sqrt(x);
```

Deadlock

Again, all the threads in a warp execute both branches of the condition.

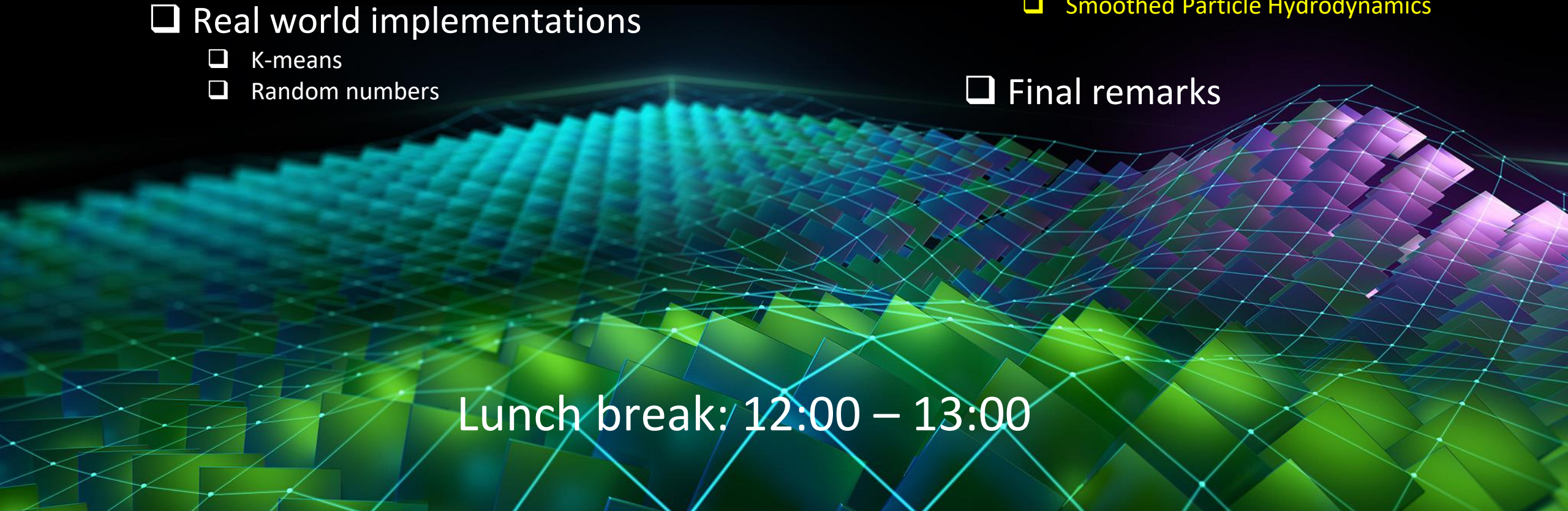
Example: a single block of 32 threads is mapped to a single warp, where the 32 threads **will execute the same instruction**.

- 50% of the threads (with threaded.x < 16) execute the first branch, **the other 50% are idle.**
- The first 50% reach the __syncthread() instruction. It will never return because the other threads cannot execute the second branch yet.

```
28    if (threadidx.x < 16)
29    {
30        myFunc_then();
31        __syncthread();
32    }
33    else if (threadid.x >= 16)
34    {
35        myFunc_else();
36        __syncthread();
37    }
38
```

Outline

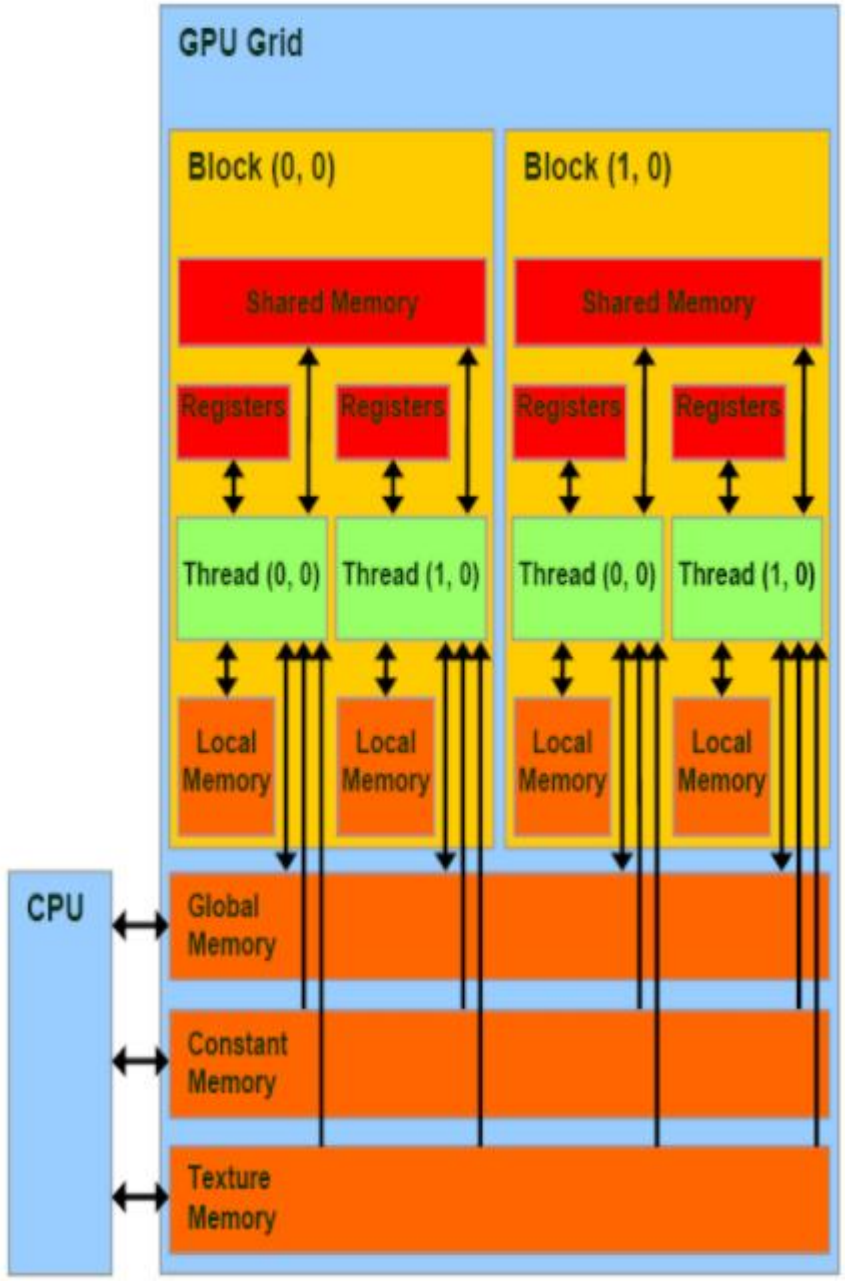
- Introduction
- Example
  - Matrix Multiplication + Exercise
- Real world implementations
  - K-means
  - Random numbers
- Common pitfalls
- Memory management
  - Image processing example
- Monitoring and asynchronicity
  - Smoothed Particle Hydrodynamics
- Final remarks



Lunch break: 12:00 – 13:00

There are 6 types of memories in a GPU.

Memory type	Speed	Location	Size	Comments
Register	+++++	In-chip	256 KB/SM	Visible only to the thread that wrote it and it has the same lifespan as that thread
Shared	++++	In-chip	48 KB/SM	Visible to all threads within the block and it hs the same lifespan as that block
Constant	+++	Off-chip	64 KB	≈8 KB/SM cached. Read-only. Used for data that will not change over the course of a kernel execution
Texture	++	Off-chip	≈ GB	Read-only. Useful when all reads in a warp are physically adjacent. Taken out from Global memory.
Local	+	Off-chip	512 KB/Thread	Visible only to the thread that wrote it and it has the same lifespan as that thread. Taken out from Global memory.
Global	+	Off-chip	≈ GB	Visible to all threads in the application and it has the same lifespan as the host allocation

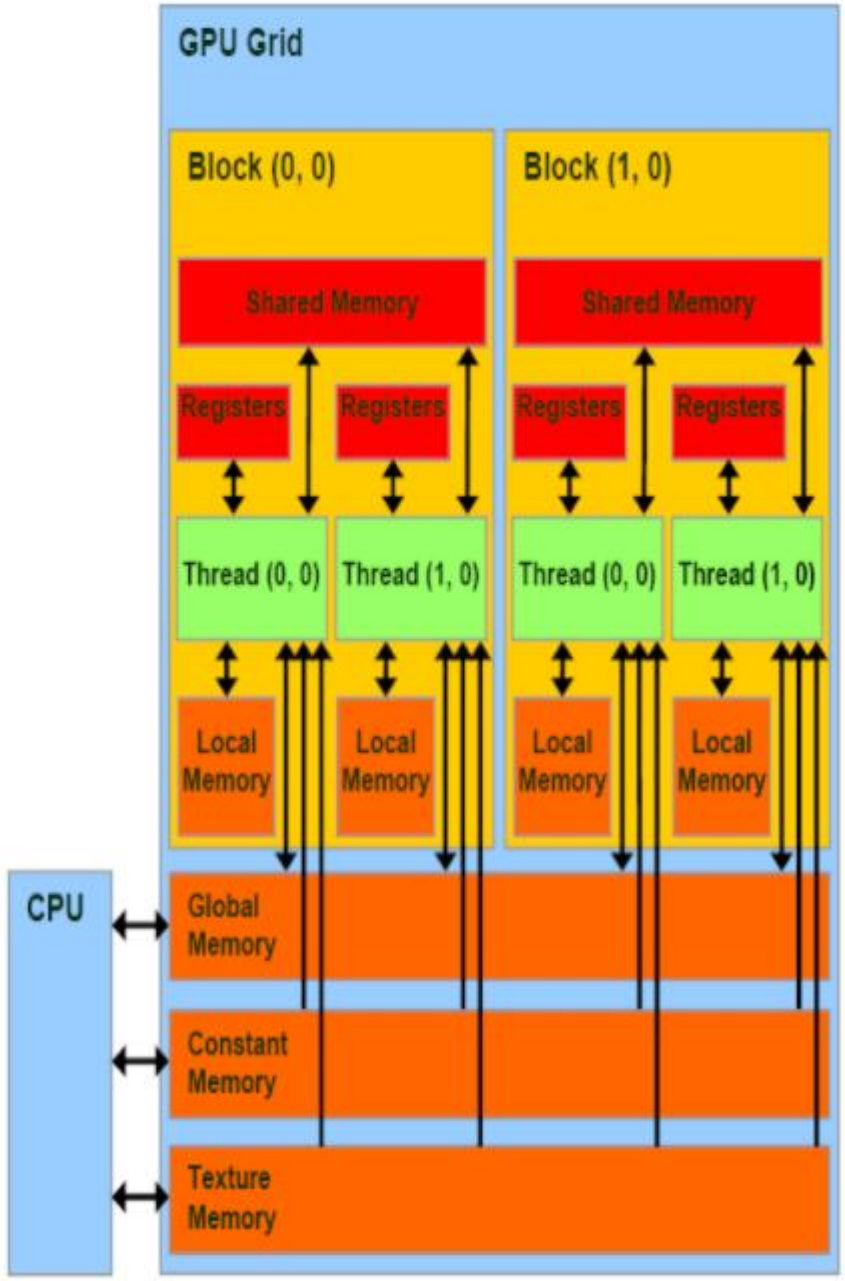


Source: cuda-programming.blogspot.com

There are 6 types of memories in a GPU.

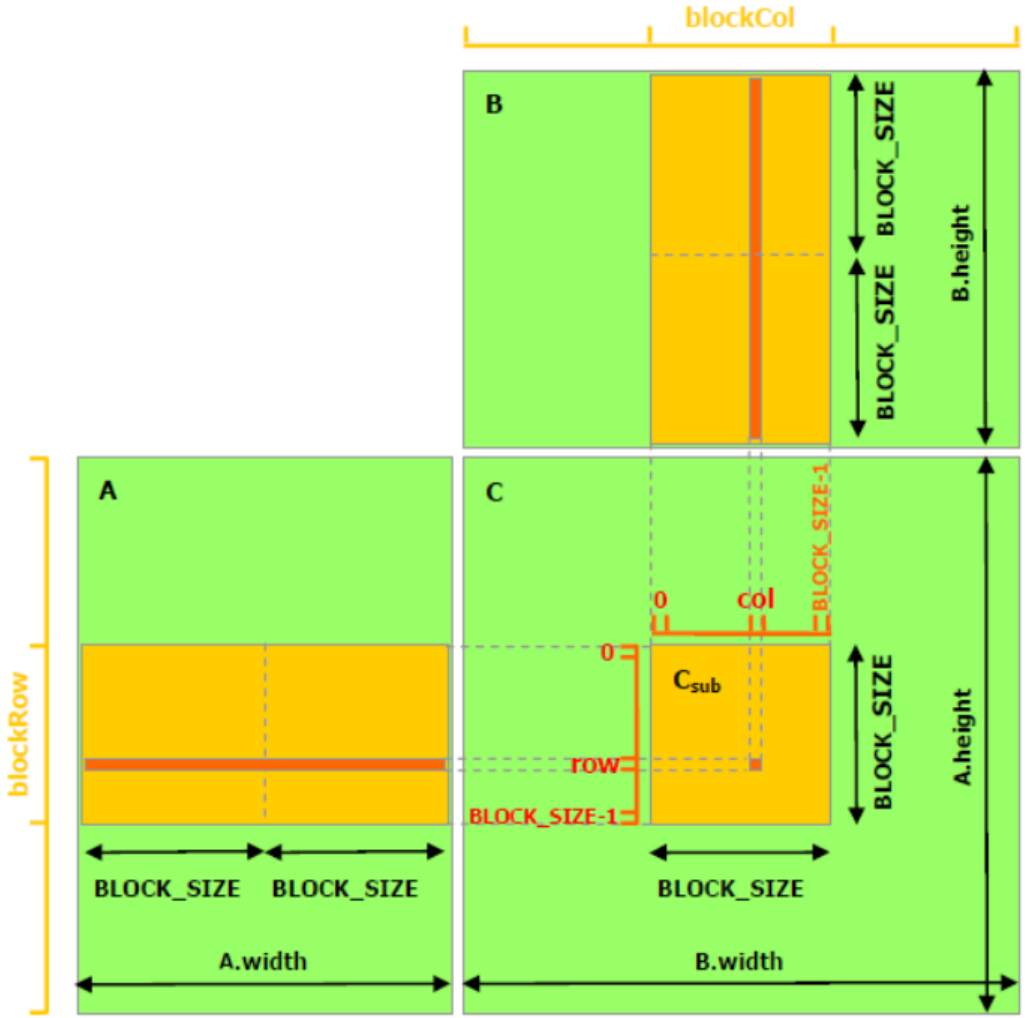
Memory type	Speed	Location	Size	Comments
Register	+++++	In-chip	256 KB/SM	Visible only to the thread that wrote it and it has the same lifespan as that thread
Shared	++++	In-chip	48 KB/SM	Visible to all threads within the block and it hs the same lifespan as that block
Constant	+++	Off-chip	64 KB	≈8 KB/SM cached. Read-only. Used for data that will not change over the course of a kernel execution
Texture	++	Off-chip	≈ GB	Read-only. Useful when all reads in a warp are physically adjacent. Taken out from Global memory.
Local	+	Off-chip	512 KB/Thread	Visible only to the thread that wrote it and it has the same lifespan as that thread. Taken out from Global memory.
Global	+	Off-chip	≈ GB	Visible to all threads in the application and it has the same lifespan as the host allocation

Recommendation: Try to make use of shared memory wherever possible.
Global memory can be 150x slower!



Source: cuda-programming.blogspot.com

Up to now we have linearized the 2D matrices and mapped the multiplication onto the GPU.
Now we are going to perform the multiplication by blocks:

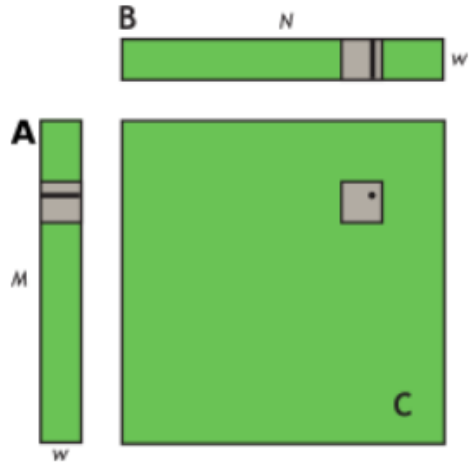


- Each threadblock computes a sub-matrix C_{sub}
- Each thread within the block computes one element of C_{sub}

BLOCK_SIZE = 16 (or 32), so that the # of threads/block is a multiple of the warp size (w) and remains below the max # of threads/block.

```
__global__ void simpleMultiply(float *a, float* b, float *c, int N)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;
    for (int i = 0; i < TILE_DIM; i++) {
         sum += a[row*TILE_DIM+i] * b[i*N+col];
    }
    c[row*N+col] = sum;
}
```

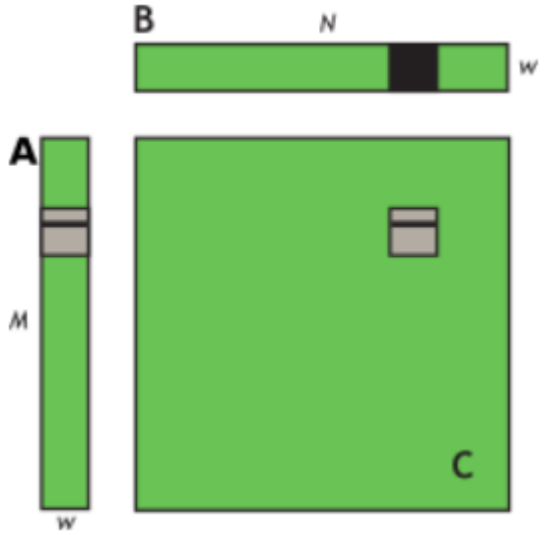
blockDim.x, blockDim.y, and TILE_DIM = w



Effective BW: 119,9 GB/s on Tesla V100

Source: Nvidia CUDA documentation

Let's analyze how data is accessed by the warps:



Each row of C_{sub} needs a row of the A tile and all the columns of the B tile.
For each iteration of i, the warp reads a full row of the B tile (**Good**), but all threads in the warp read the same value from global memory from the A tile (**Bad**): wasted memory BW and likely cache miss.

It would be better to read the A tile from shared memory:

```
__global__ void simpleMultiply(float *a, float* b, float *c, int N)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;
    for (int i = 0; i < TILE_DIM; i++) {
         sum += a[row*TILE_DIM+i] * b[i*N+col];
    }
    c[row*N+col] = sum;
}
```

Effective BW: 119,9 GB/s on Tesla V100

```
__global__ void coalescedMultiply(float *a, float* b, float *c, int N)
{
    __shared__ float aTile[TILE_DIM][TILE_DIM];

    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;

    aTile[threadIdx.y][threadIdx.x] = a[row*TILE_DIM+threadIdx.x];

    __syncwarp();

    for (int i = 0; i < TILE_DIM; i++) {
         sum += aTile[threadIdx.y][i]* b[i*N+col];
    }
    c[row*N+col] = sum;
}
```

Effective BW: 144,4 GB/s on Tesla V100

Each element of tile A is read once w/o BW waste to shared memory.

Source: Nvidia CUDA documentation

Because shared memory is in-chip, it has much higher bandwidth and lower latency than accessing global memory. But the speed gain depends on not having bank conflicts between threads.

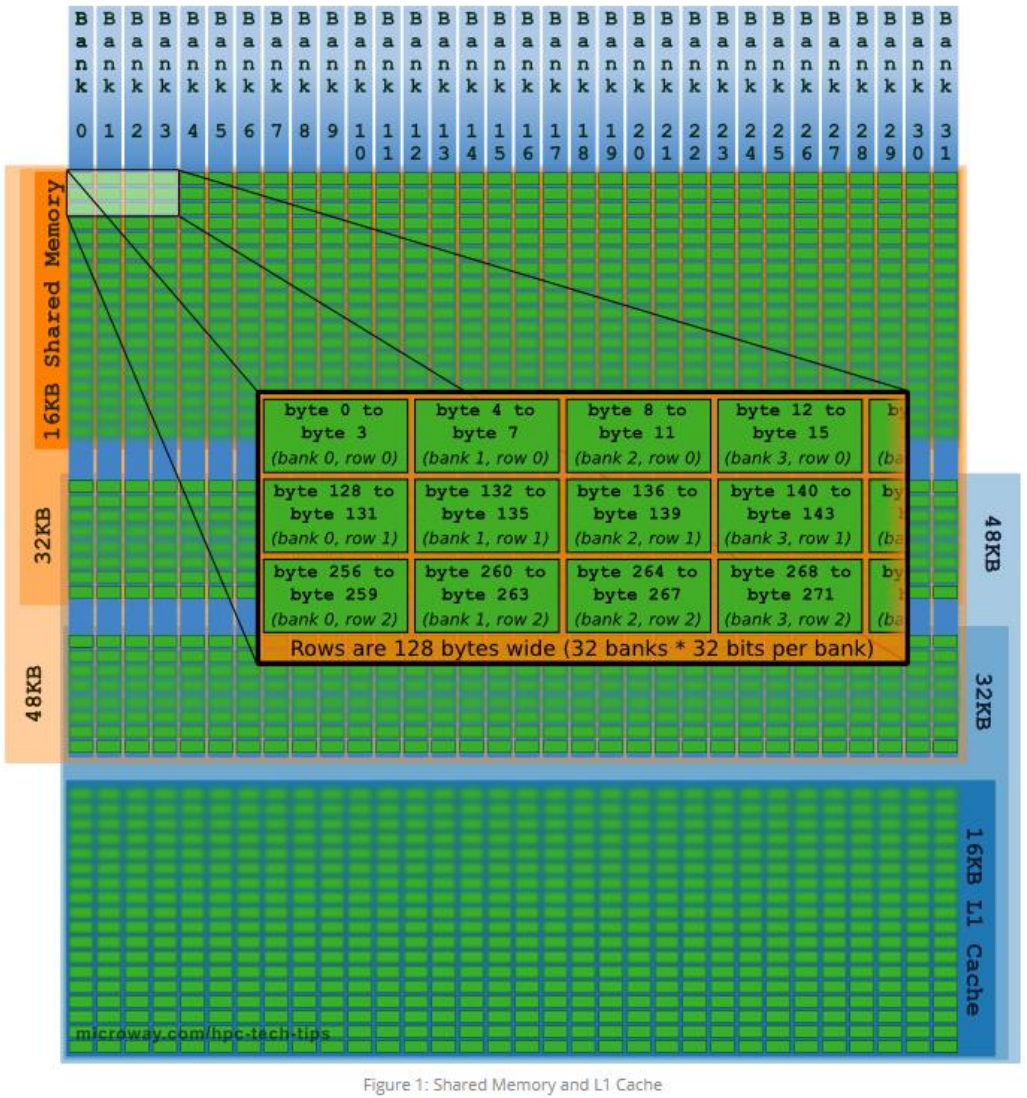
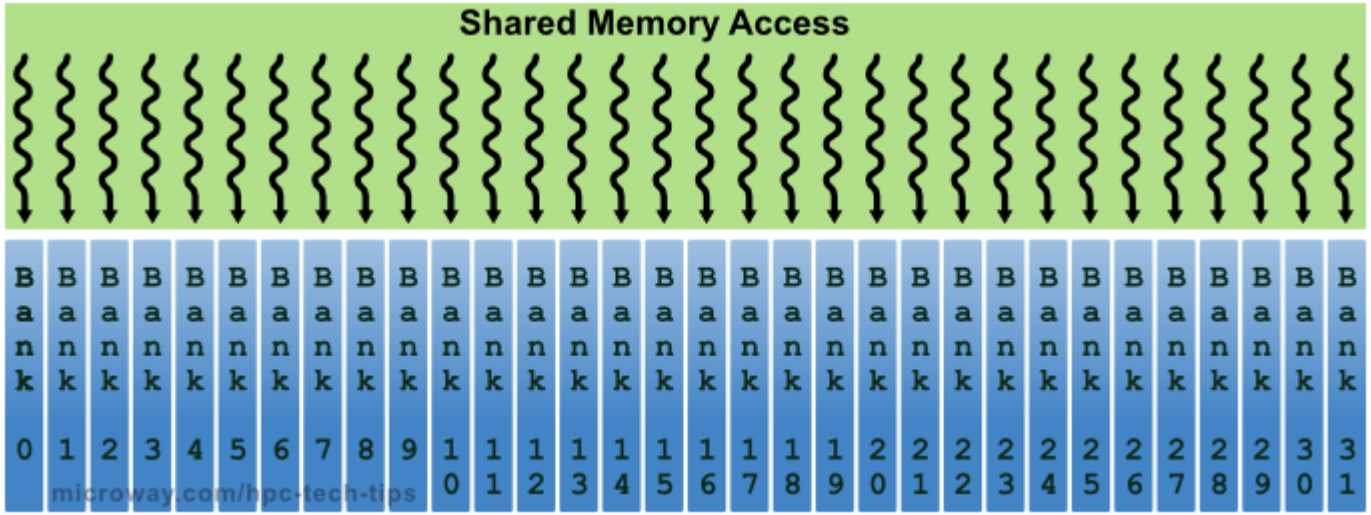


Figure 1: Shared Memory and L1 Cache

In order to effectively service concurrent memory access, shared memory is divided into 32 equal memory Banks.



Source: microway.com

Because shared memory is in-chip, it has much higher bandwidth and lower latency than accessing global memory. But the speed gain depends on not having bank conflicts between threads.

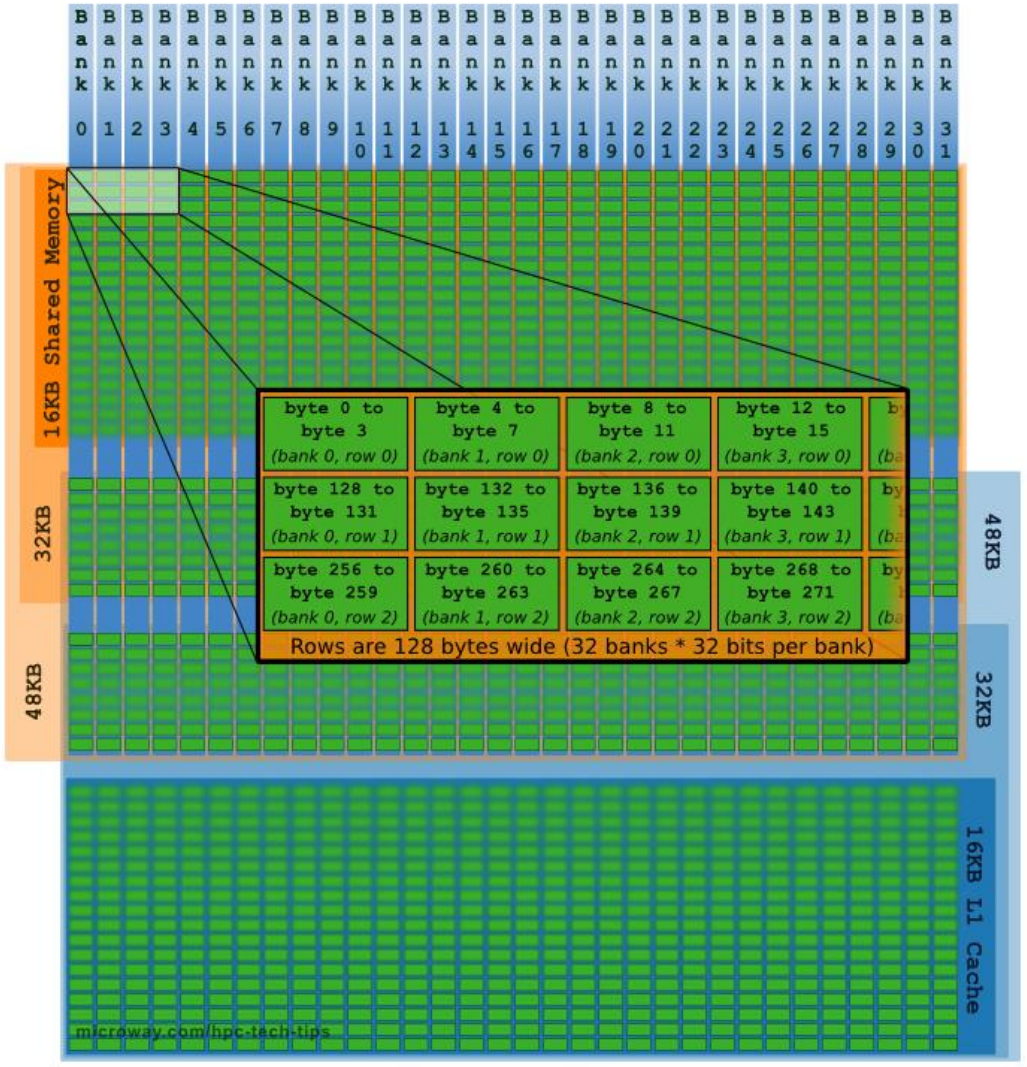
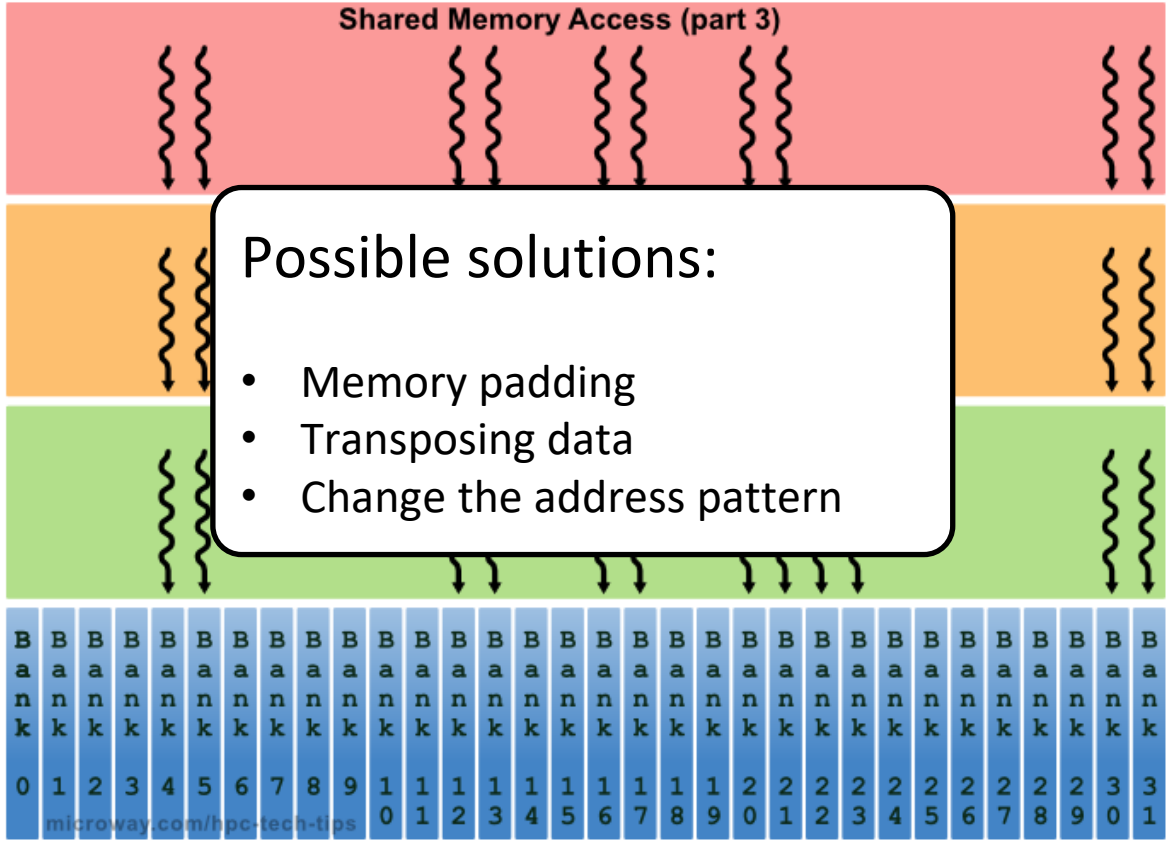


Figure 1: Shared Memory and L1 Cache

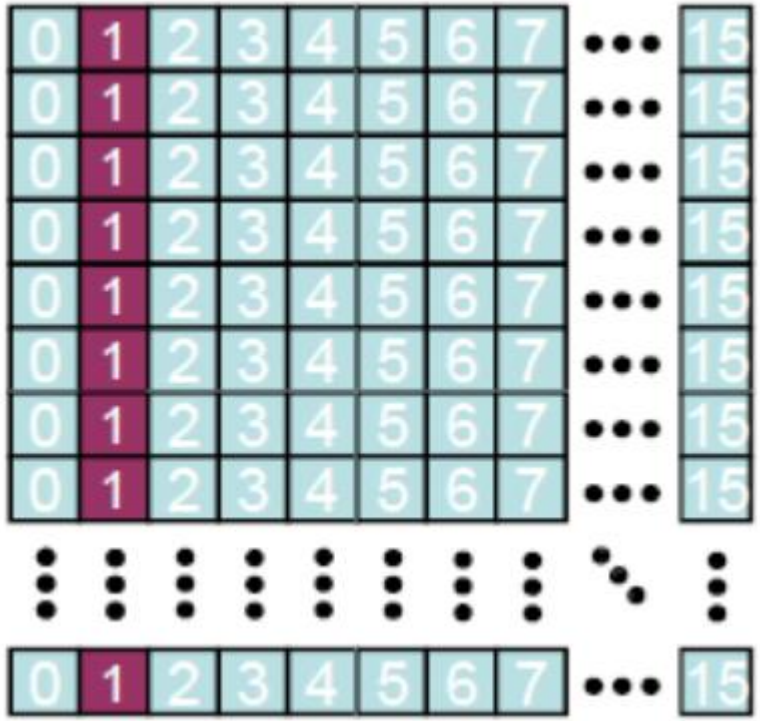
But if multiple thread requests map to the same bank, the accesses are serialized with as many accesses as to ensure there a no conflicts.



Source: microway.com

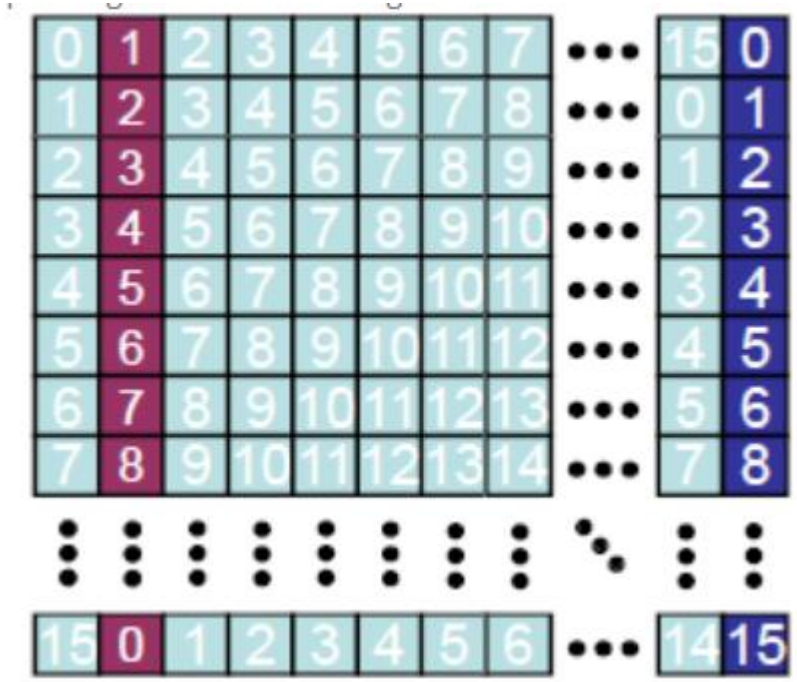
Example of bank conflict:

Let's assume process a 2D array (16x16) and we assing each thread to process one row.



With memory padding we add an extra column with zeros.
Now there is no bank conflicts!

Instead of: _shared_ int shared[TILE_WIDTH][TILE_HEIGHT];
use: _shared_ int shared[TILE_WIDTH+1][TILE_HEIGHT];

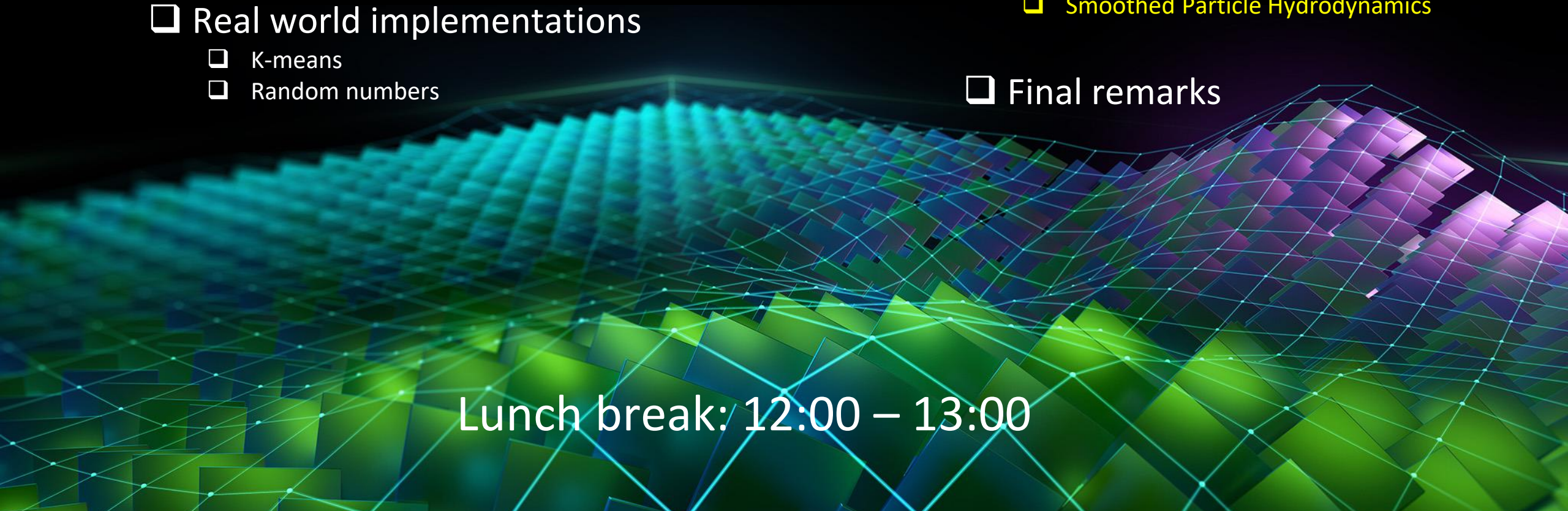


The number is the bank in which data is stored.
All threads move to the right (like the purple cells) having a 16-way bank conflict.

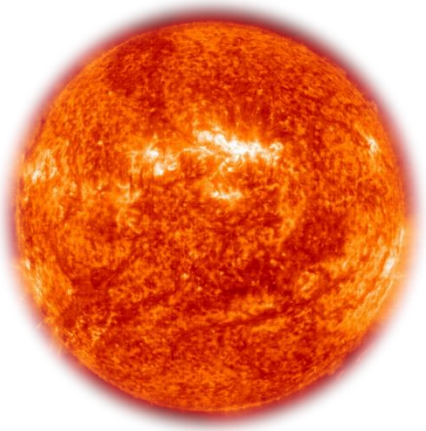
Source: cuda-programming-blogspot.com

Outline

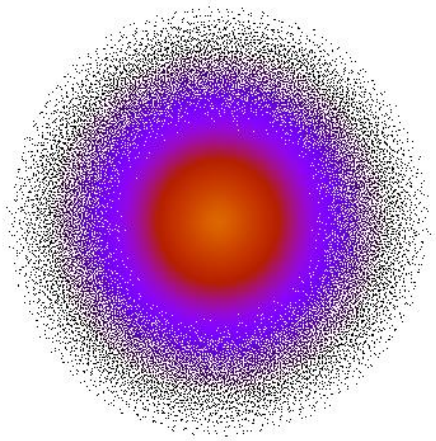
- ❑ Introduction
- ❑ Example
  - ❑ Matrix Multiplication + Exercise
- ❑ Real world implementations
  - ❑ K-means
  - ❑ Random numbers
- ❑ Common pitfalls
- ❑ Memory management
  - ❑ Image processing example
- ❑ Monitoring and asynchronicity
  - ❑ Smoothed Particle Hydrodynamics
- ❑ Final remarks



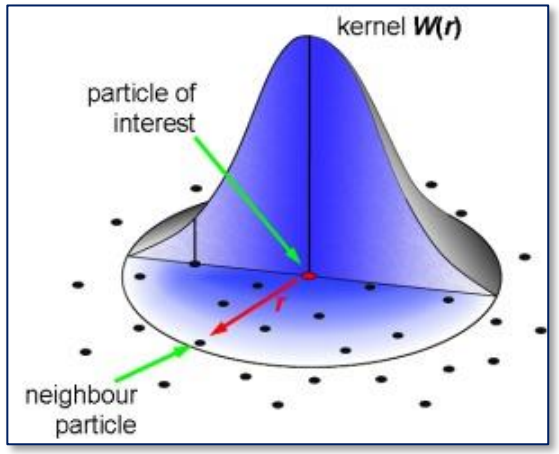
Lunch break: 12:00 – 13:00



Fluid



SPH particles



$$< f(\boldsymbol{r}) > = \int f(\boldsymbol{r}')W(\boldsymbol{r}' - \boldsymbol{r})d\boldsymbol{r}'$$

Loop over particles (a):
Loop over neighbors (b):

$$f_{\boldsymbol{a}}(\boldsymbol{r}) = \sum_{b=1}^{n_v} \frac{m_b}{\rho_b} f_b(\boldsymbol{r}) W(\boldsymbol{r}_{ab}, h)$$

$f(\boldsymbol{r}) = \rho$

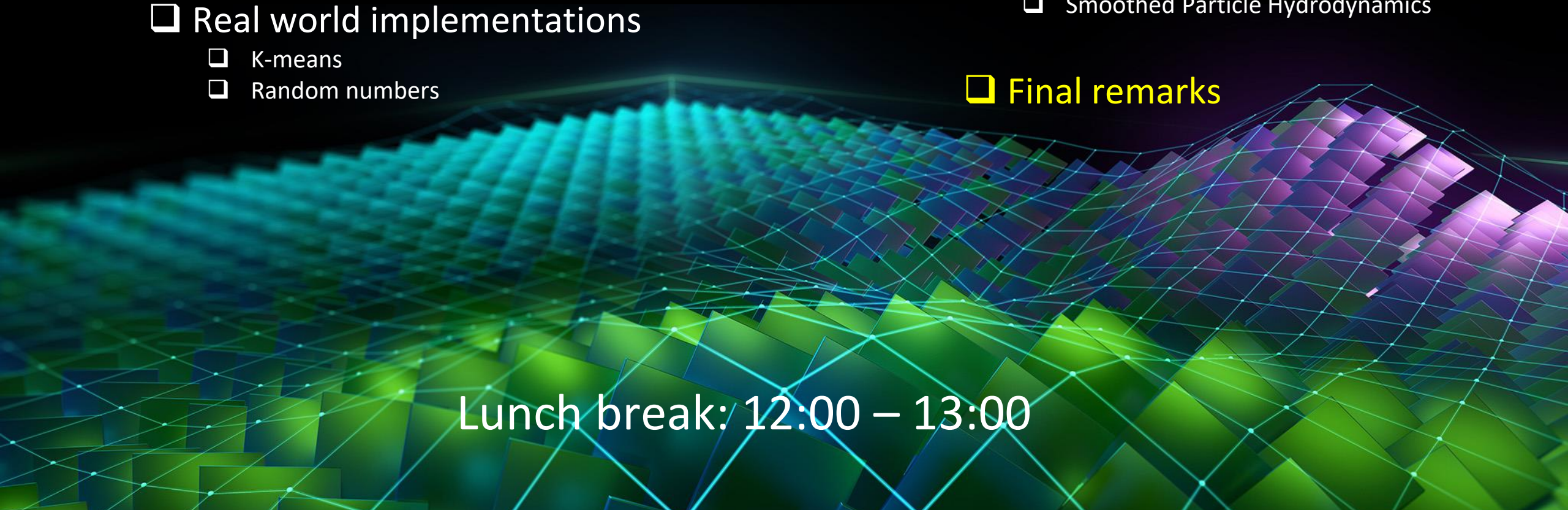
$$\rho_a = \sum_b m_b W_{ab}$$

SPH interpolation kernel

$$W_n^H(v,h) = B_n(h)\begin{cases} 1 & , v = 0 \\ \left\{\operatorname{sinc}\left(\frac{\pi}{2}v\right)\right\}^n & , 0 < v \le 2 \\ 0 & , v > 2 \end{cases}$$

Outline

- ❑ Introduction
- ❑ Example
  - ❑ Matrix Multiplication + Exercise
- ❑ Real world implementations
  - ❑ K-means
  - ❑ Random numbers
- ❑ Common pitfalls
- ❑ Memory management
  - ❑ Image processing example
- ❑ Monitoring and asynchronicity
  - ❑ Smoothed Particle Hydrodynamics
- ❑ Final remarks



Lunch break: 12:00 – 13:00

GPU	Nodes	GPUs/node	Slurm Partition	Group
K80	sgi01	4	k80	Unavailable
Titanx	sgi[21-25]	6	pascal	Scicore
Titanx	sgi[21-30]	6	titanx	BIOPZ-A-RG-TM01-Group,stahlberg,basler

```
#!/bin/bash

#SBATCH --job-name=GPU_JOB
#SBATCH --time=01:00:00
#SBATCH --qos=6hours
#SBATCH --mem-per-cpu=1G
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --partition=pascal     # or k80 / titanx
#SBATCH --gres=gpu:1           # --gres=gpu:2 for two GPU,...

module load CUDA
.....
```

Where to go from here?

Official documentation:

CUDA: https://docs.nvidia.com/
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
Numba: https://numba.readthedocs.io/en/stable/index.html
PyCUDA: https://documen.tician.de/pycuda/
OpenMP: https://www.openmp.org/specifications/
OpenCL: https://github.com/KhronosGroup/OpenCL-Guide
OpenACC: https://www.openacc.org/resources

CUDA:

nVidia CUDA training : https://developer.nvidia.com/accelerated-computing-training
3rd-party CUDA training: https://developer.nvidia.com/educators/existing-courses
Numba CUDA: https://nyu-cds.github.io/python-numba/05-cuda/

OpenMP:

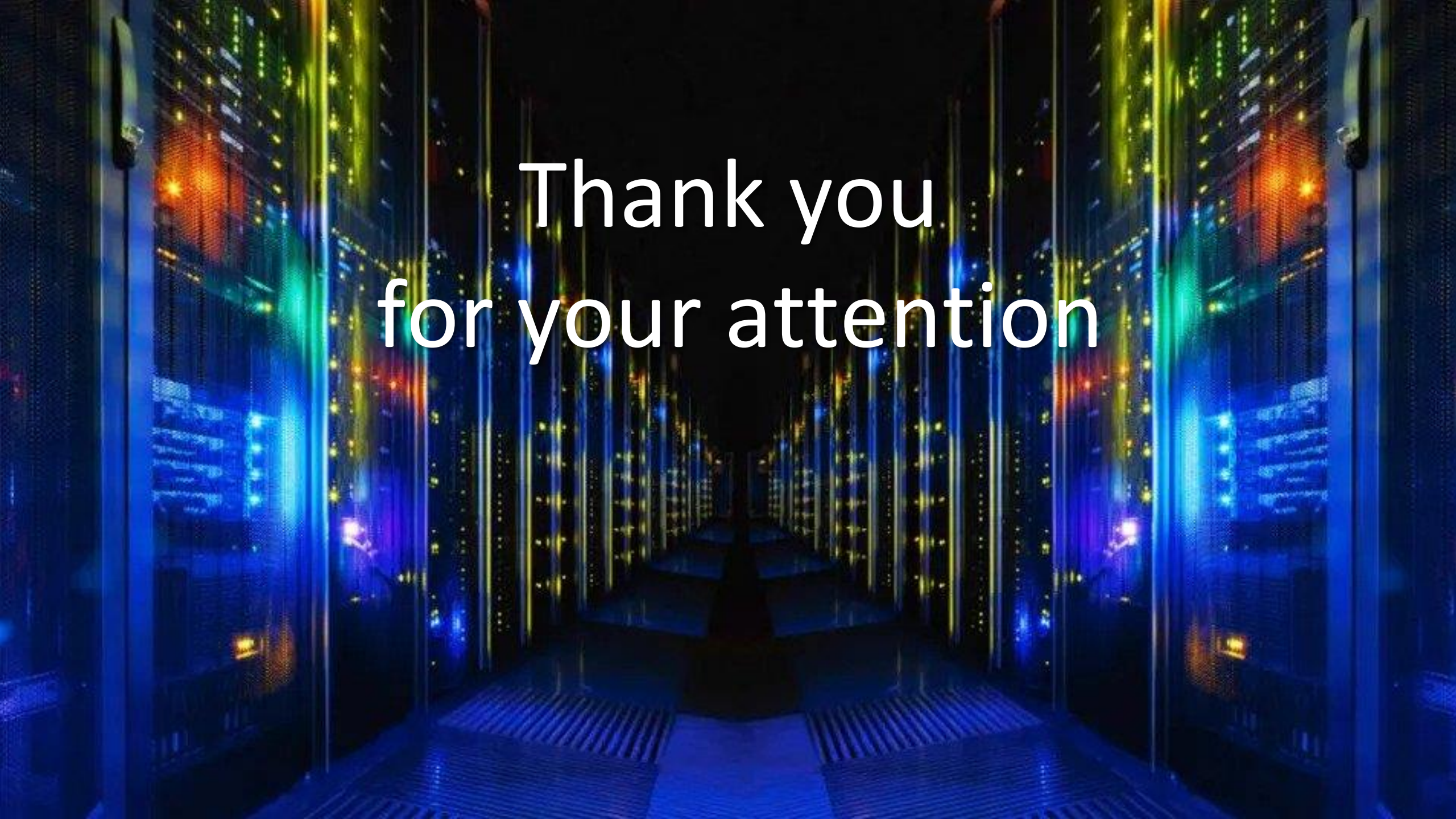
Offloading support in GCC: https://gcc.gnu.org/wiki/Offloading
OpenMP offloading talk: https://www.youtube.com/watch?v=ypRBx31e8GA

Numba:

Numba intro: https://www.youtube.com/watch?v=6oXedk2tGfk
Numba pitfalls: https://www.youtube.com/watch?v=x58W9A2lnQc

Optimization in GPUs:

GPU optimization strategies: https://www.paranumal.com/single-post/2018/02/26/basic-gpu-optimization-strategies
Memory types: https://www.microway.com/hpc-tech-tips/gpu-memory-types-performance-comparison/
Using shared memory: https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/


Thank you
for your attention