

# Introduction to GPU programming

# Outline

- ❑ Introduction
- ❑ Example
  - ❑ Matrix Multiplication + Exercise
- ❑ Real world implementations
  - ❑ Genomes simulation
  - ❑ Random numbers generation
- ❑ 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 + Exercise

- ❑ Real world implementations

- ❑ Genomes simulation
  - ❑ Random numbers generation

- ❑ Common pitfalls

- ❑ Memory management

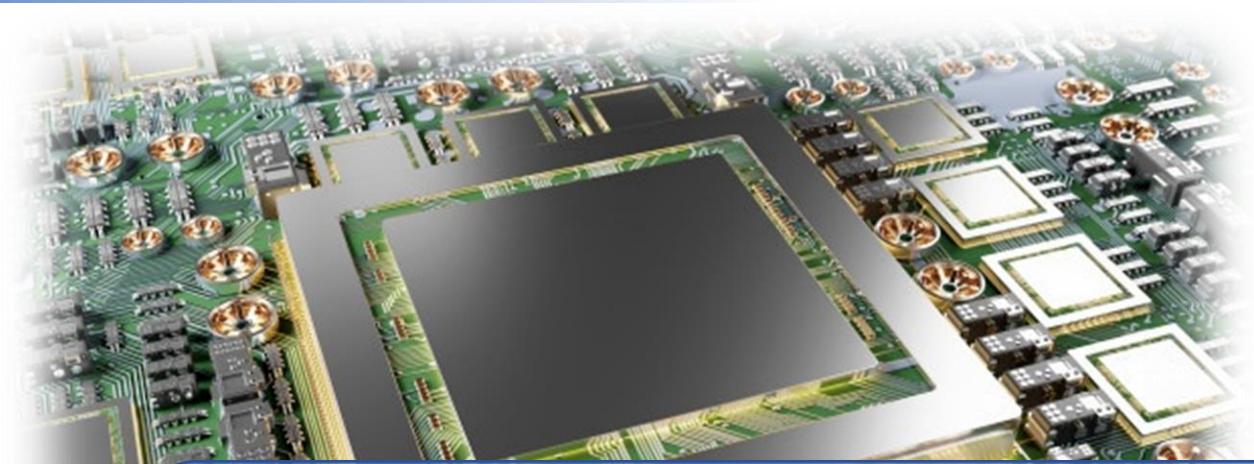
- ❑ Image processing example

- ❑ 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

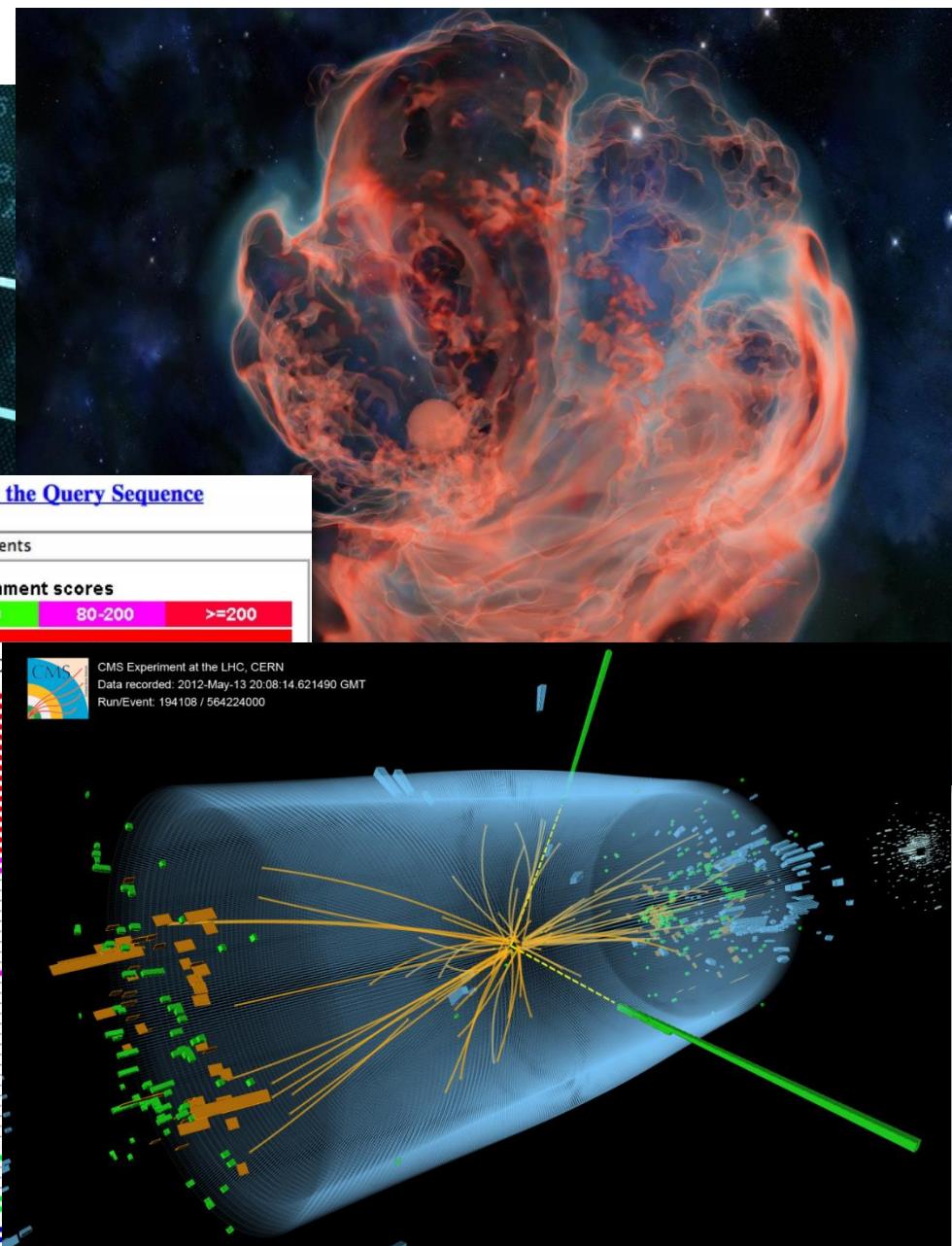
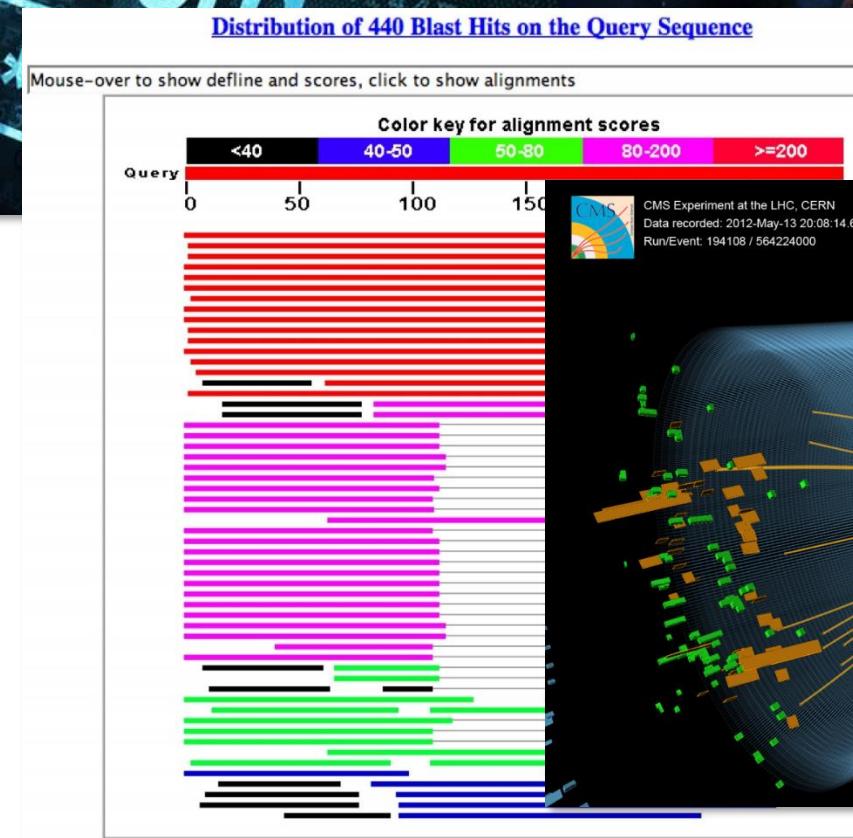
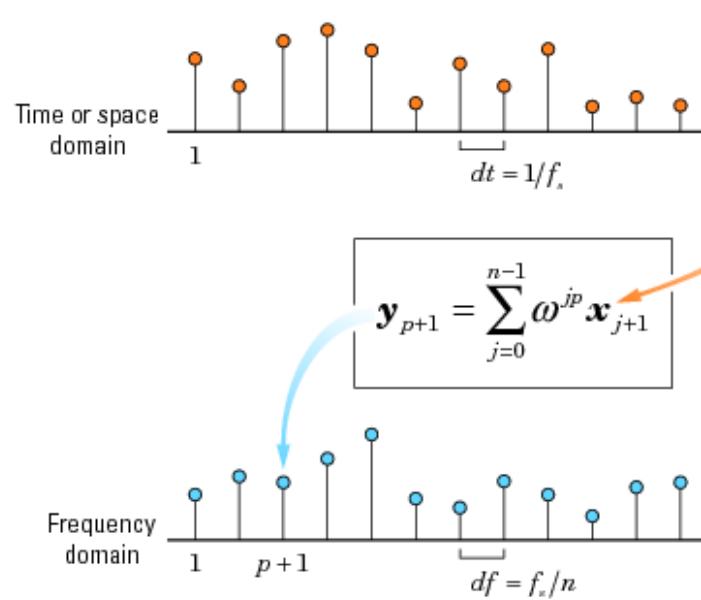




# GPU

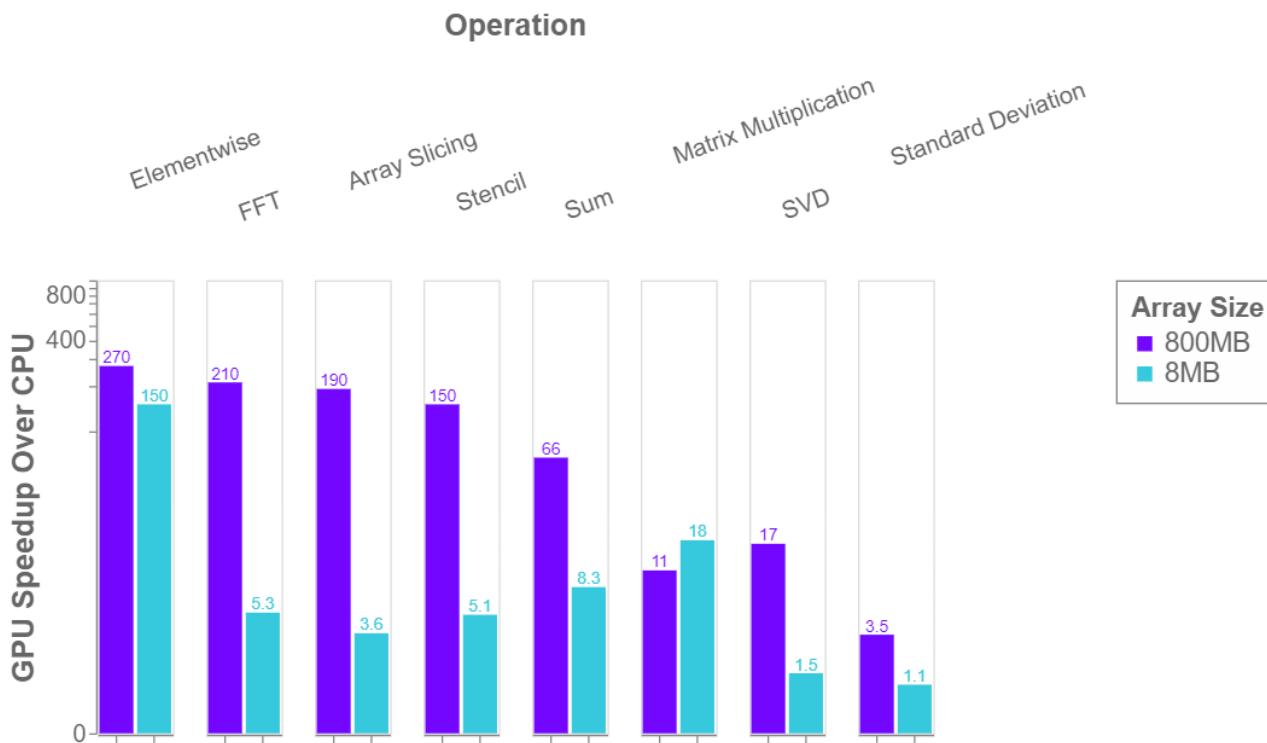
## Graphics Processing Unit





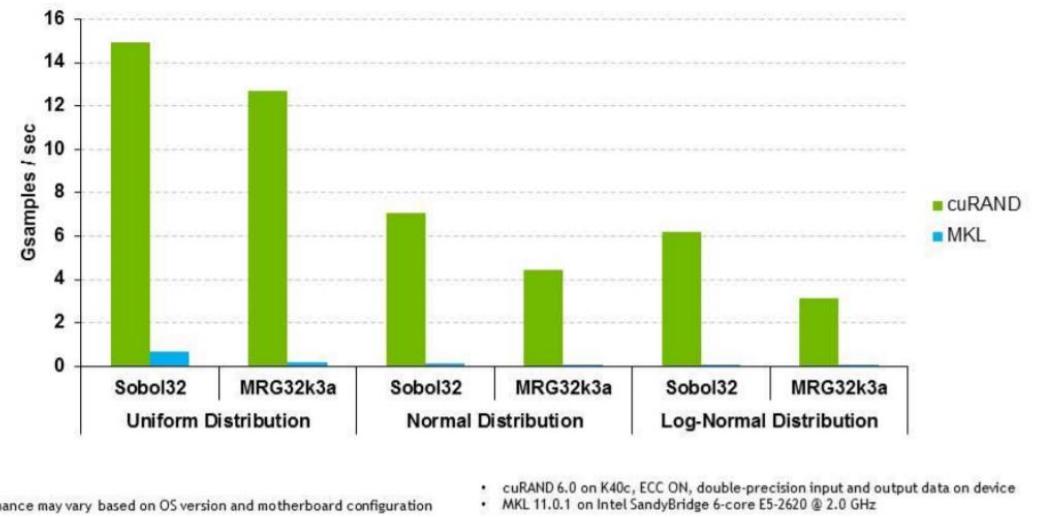
**GPGPU**  
General Purpose  
Graphics Processing Unit

# Introduction



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

## cuRAND: Up to 75x Faster vs. Intel MKL



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

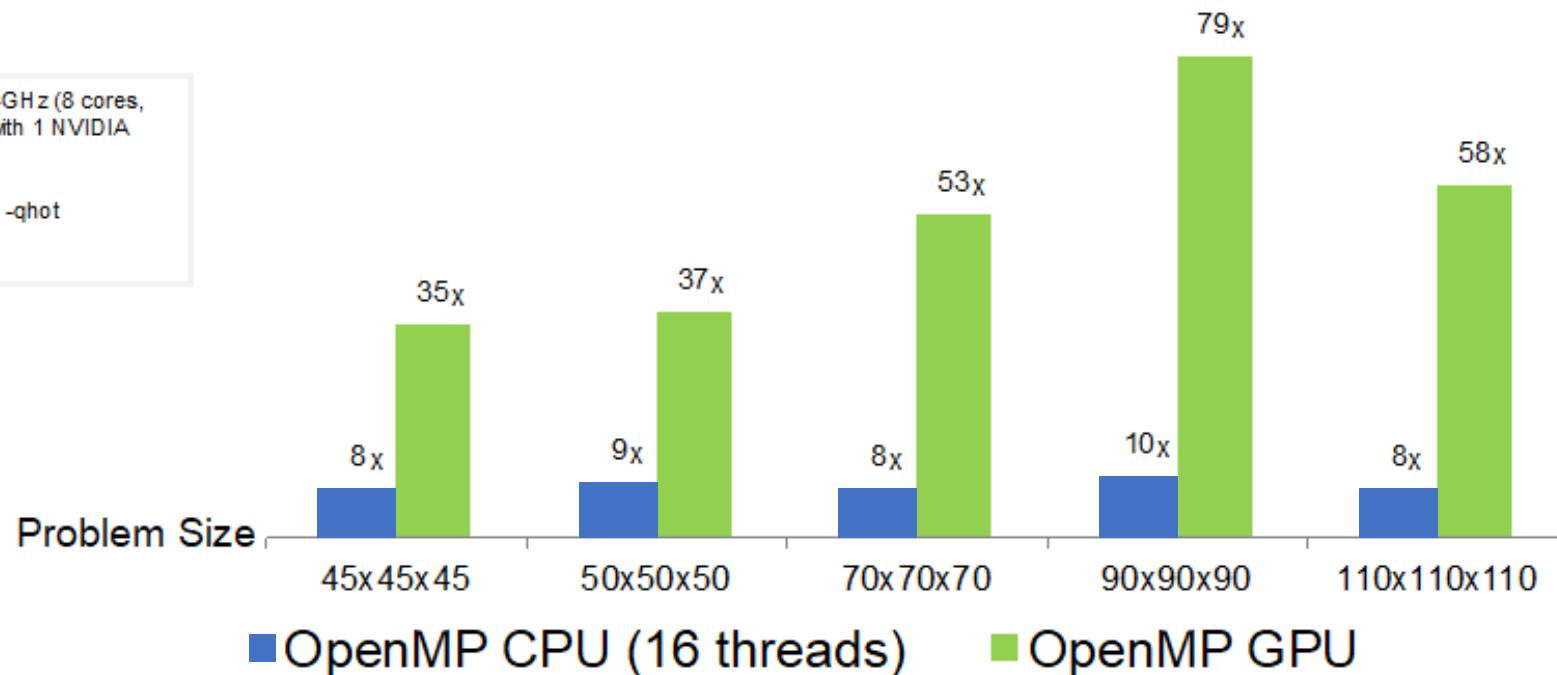
Source: <https://blog.dask.org/2019/06/27/single-gpu-cupy-benchmarks>

[http://www.eurorisksystems.com/documents/speed\\_up\\_of\\_numeric\\_calculations\\_using\\_GPU.pdf](http://www.eurorisksystems.com/documents/speed_up_of_numeric_calculations_using_GPU.pdf)

## Test Specs

2 Power8 sockets @ 4GHz (8 cores, with 8 threads each) with 1 NVIDIA Pascal P100 GPU.

Compiler Options: -O3 -qhot  
-qsmp=omp -qoffload\*  
\* Where applicable



IBM Systems

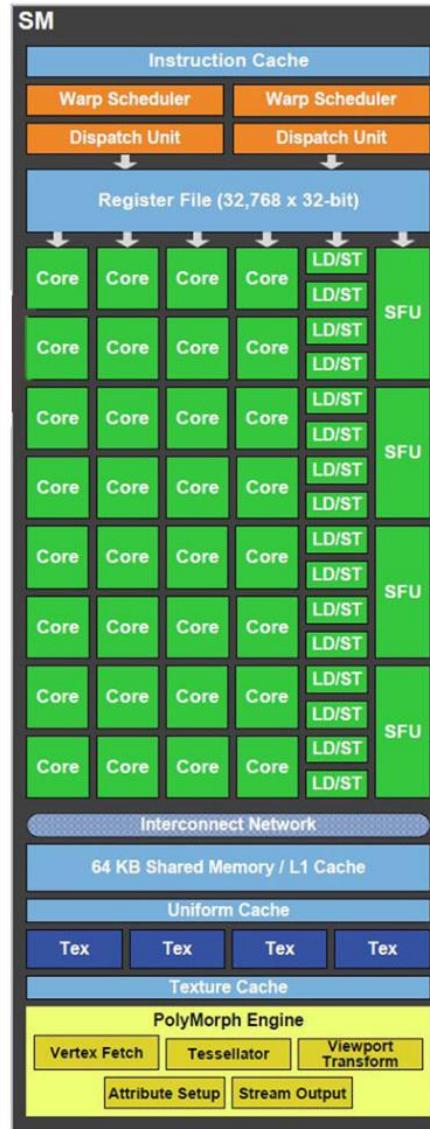
LULESH benchmark changing one single OMP directive.

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

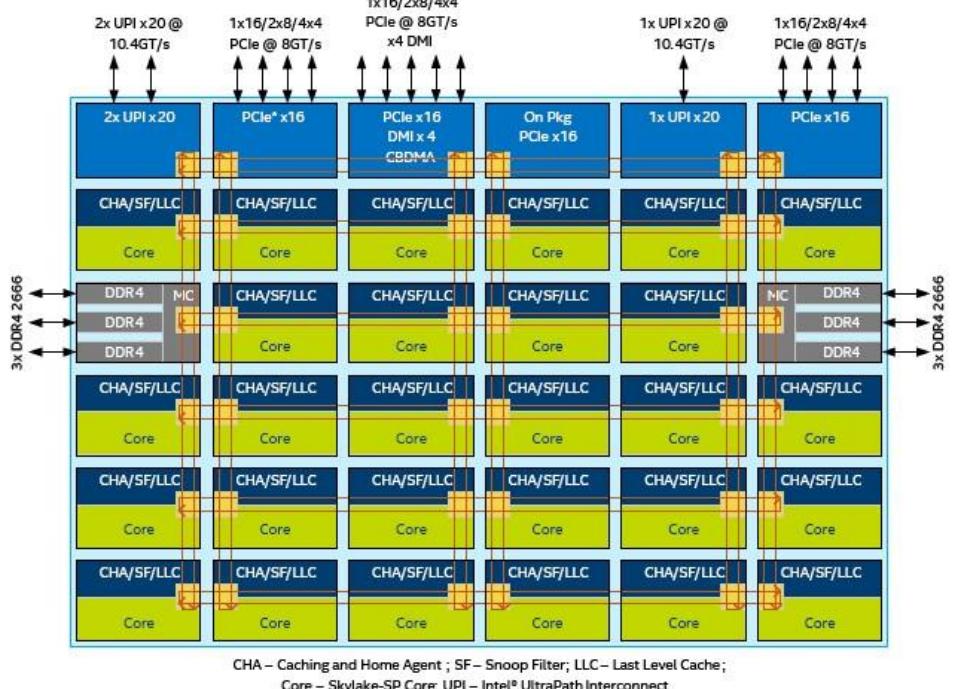
# Introduction

## Streaming Multiprocessor

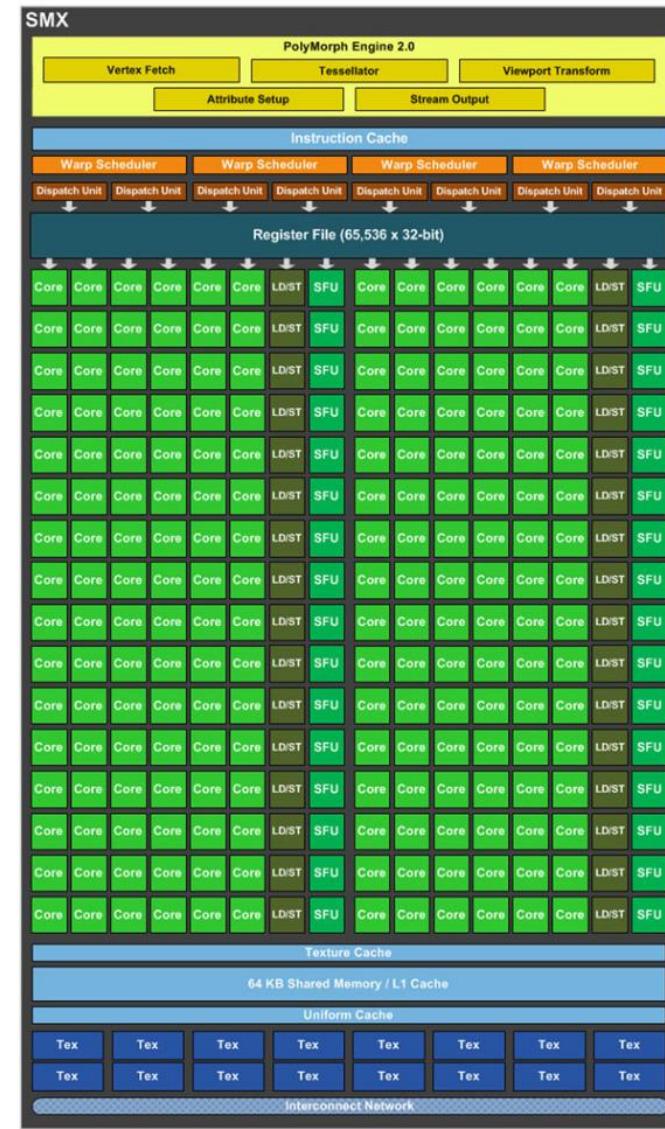
Tesla SM unit (2007)



Intel Skylake (2017)



Fermi SMX unit (2011)



CPU	GPU
Large memory	Relatively small memory
Each core has its own independent control logic <i>Allows independent execution</i>	Groups of cores share control logic <i>Saves space &amp; power</i>
Coherent caches between cores	Shared cache & sync only within groups
Tuned for serial execution of independent work <i>MIMD</i>	Tuned for parallel execution of the same work <i>SIMD</i>
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 <i>Requires regular data access patterns</i>	Memory latency hidden by scheduling stalled threads <i>Requires 1000s of concurrent threads</i>
Hyperthreading & Vectorization	None
Out of order execution	None



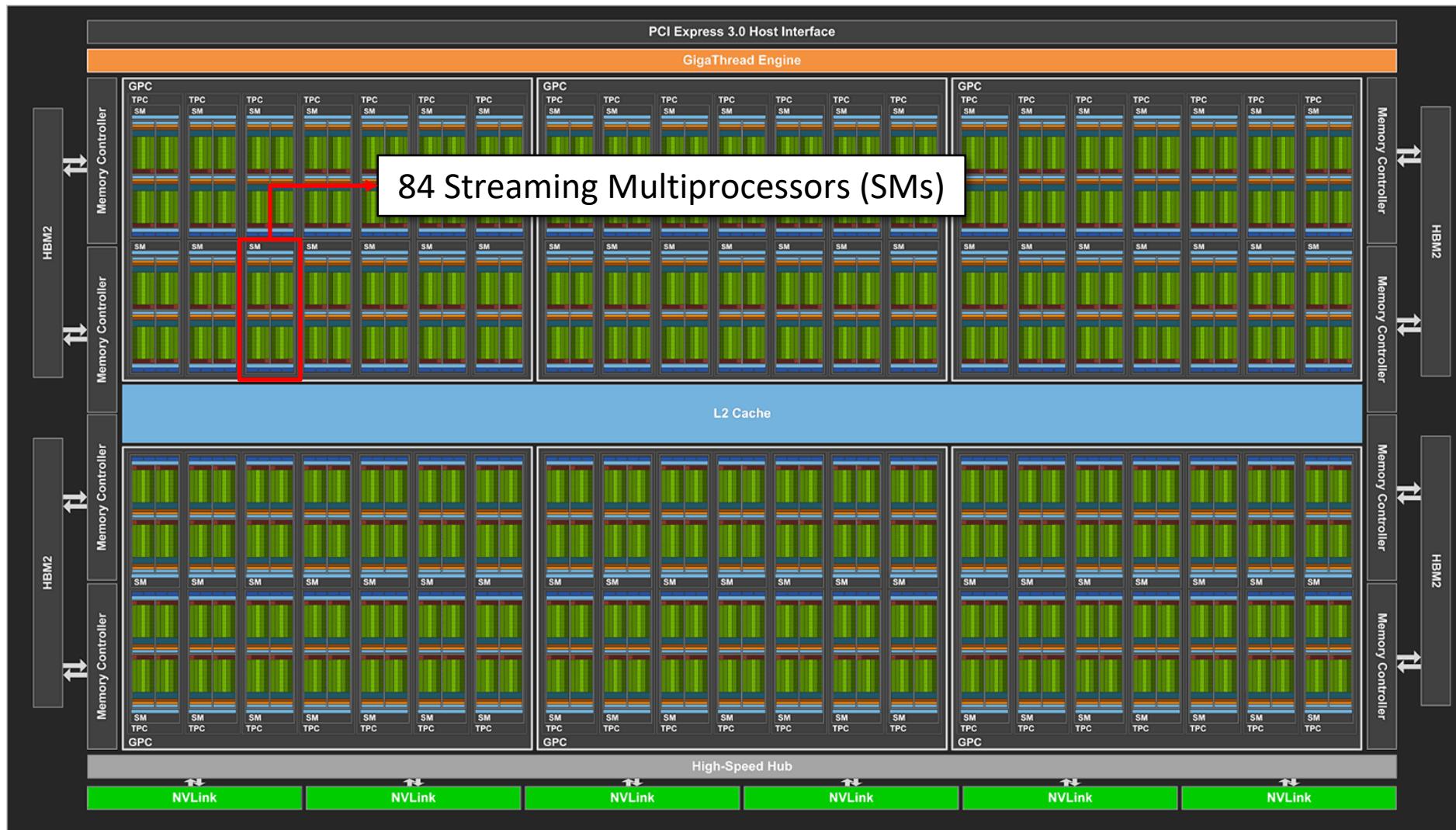
# Introduction

Volta (2017)

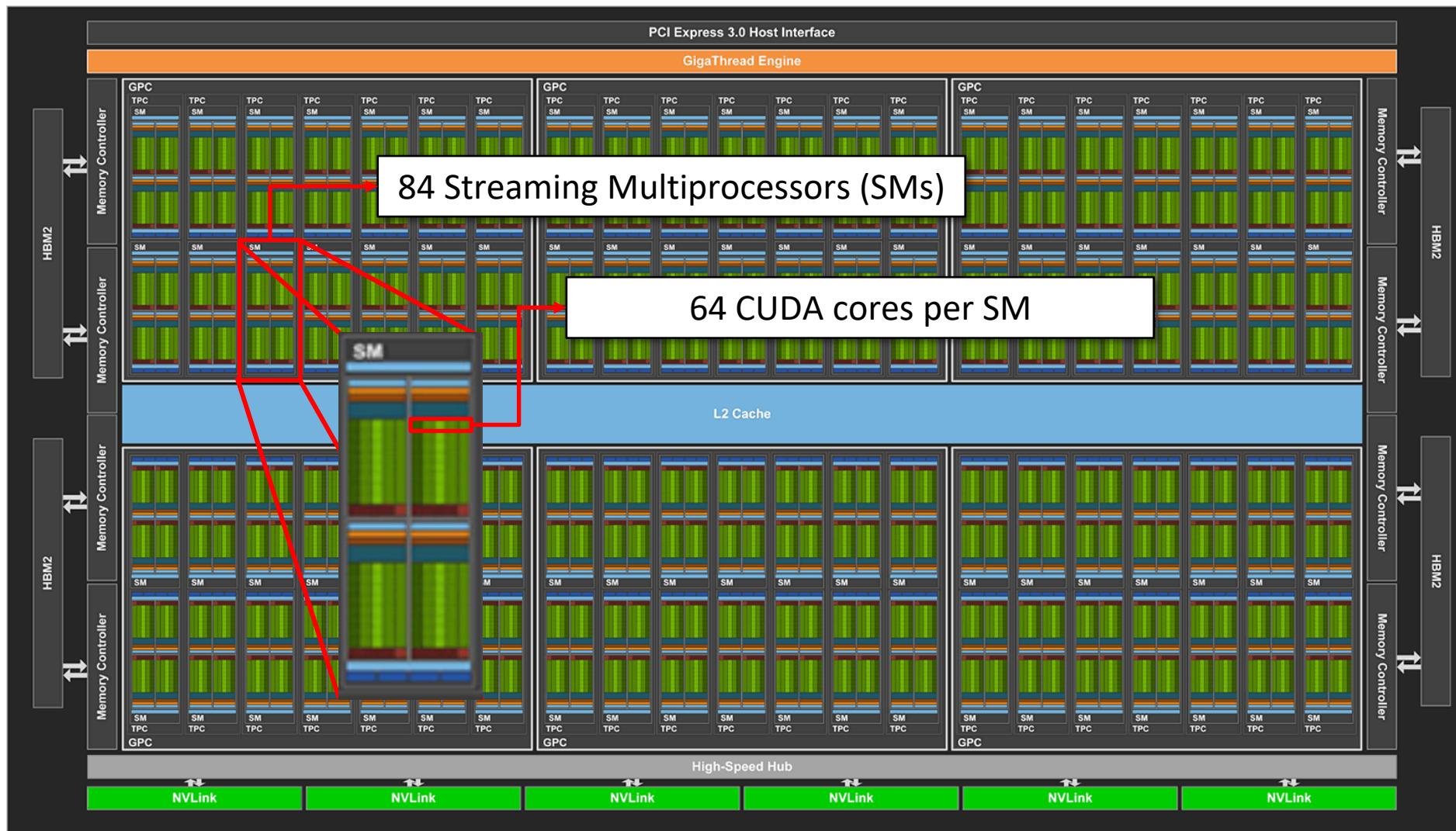


# Introduction

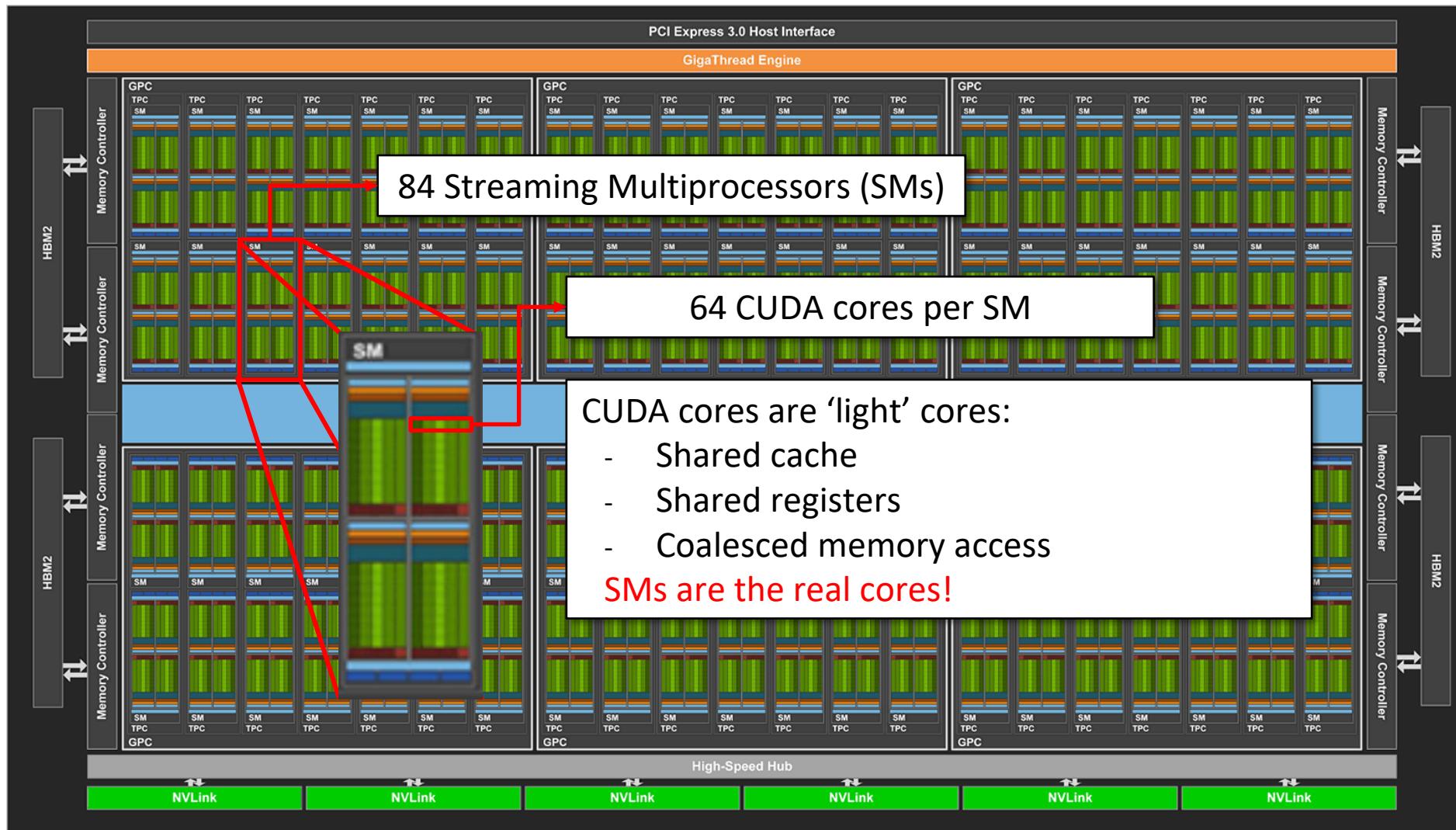
Volta (2017)



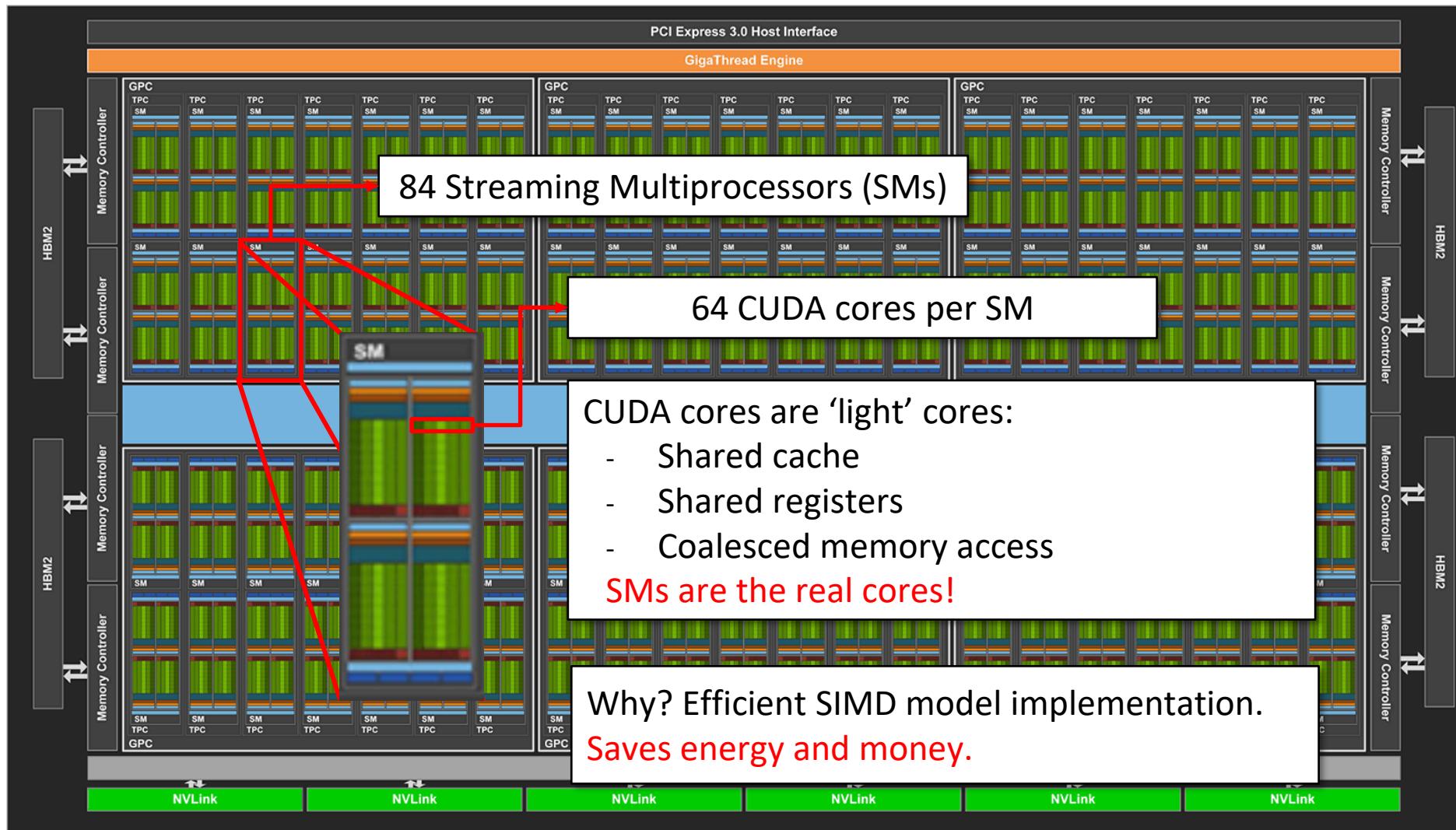
Volta (2017)



Volta (2017)

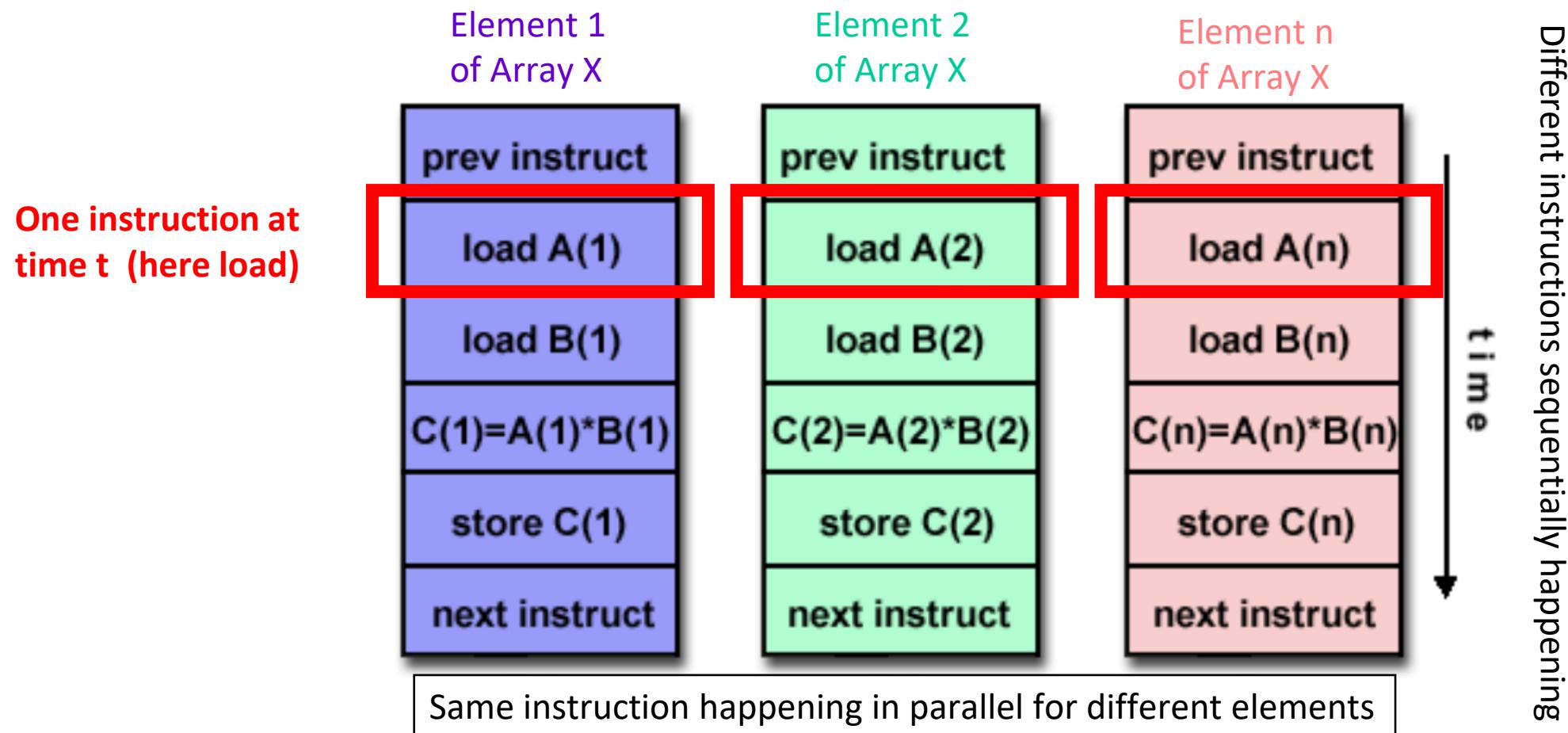


Volta (2017)

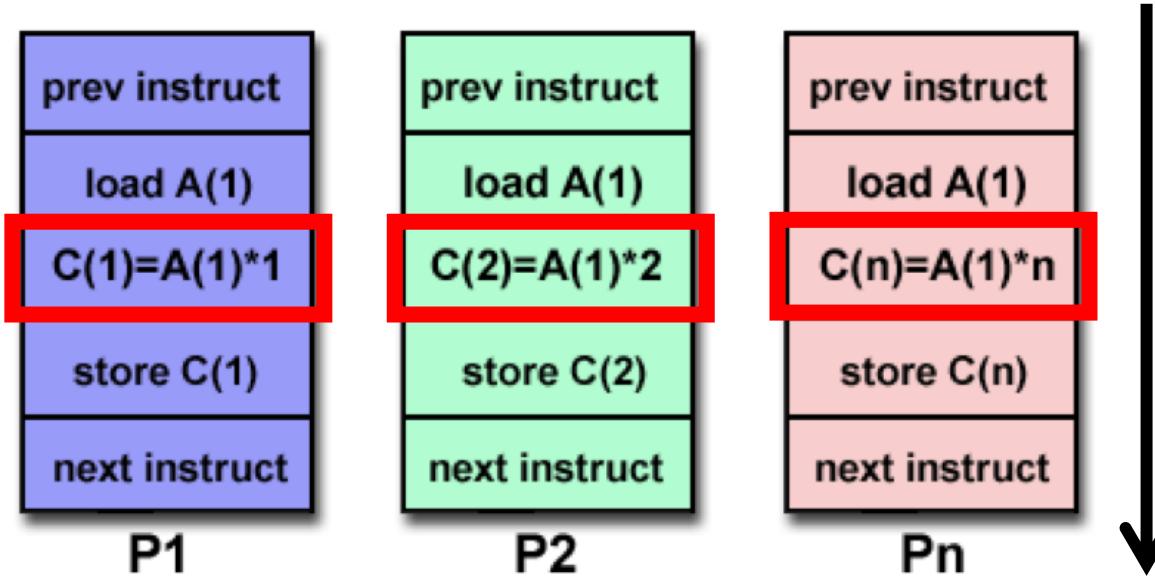


# GPUs are SIMD oriented architectures

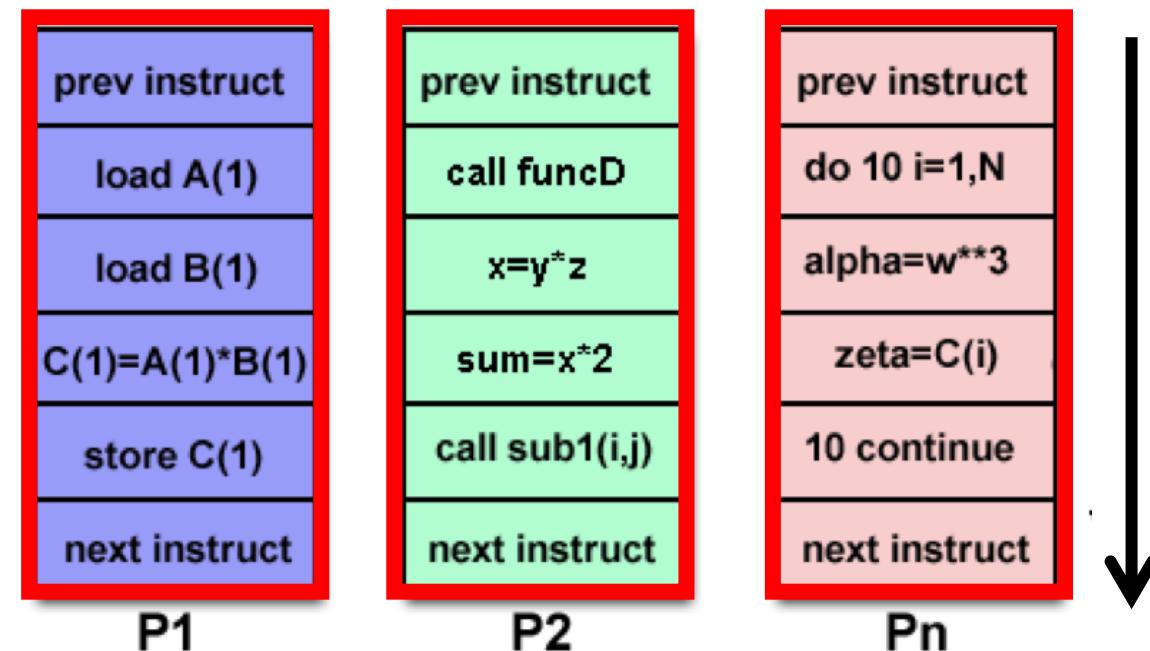
## SIMD : Single Instruction Multiple Data



# MISD : Multiple Instructions Single Data



# MIMD : Multiple Instructions Multiple Data



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

CPU can do MIMD & SIMD

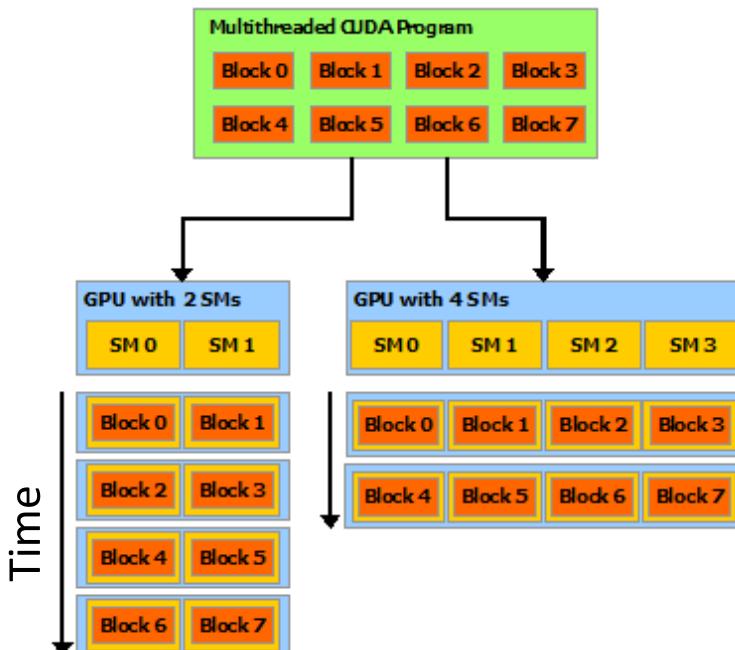
GPU can do SIMD only

# Introduction

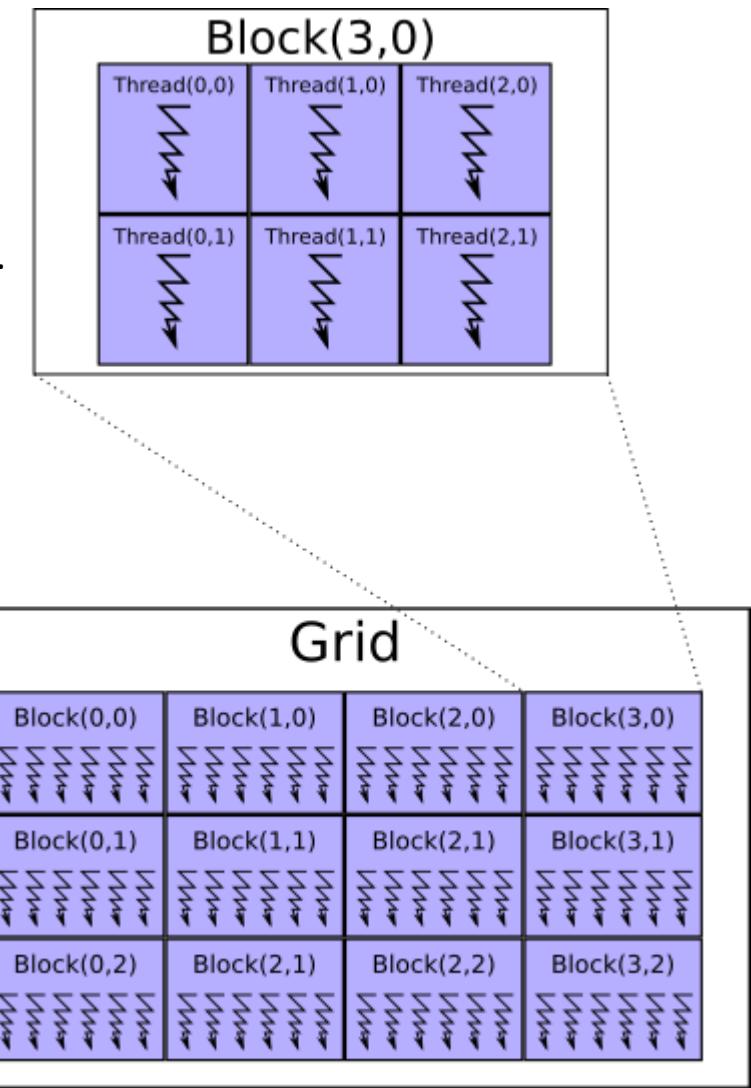
Mapping the hardware architecture into the programming architecture

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.



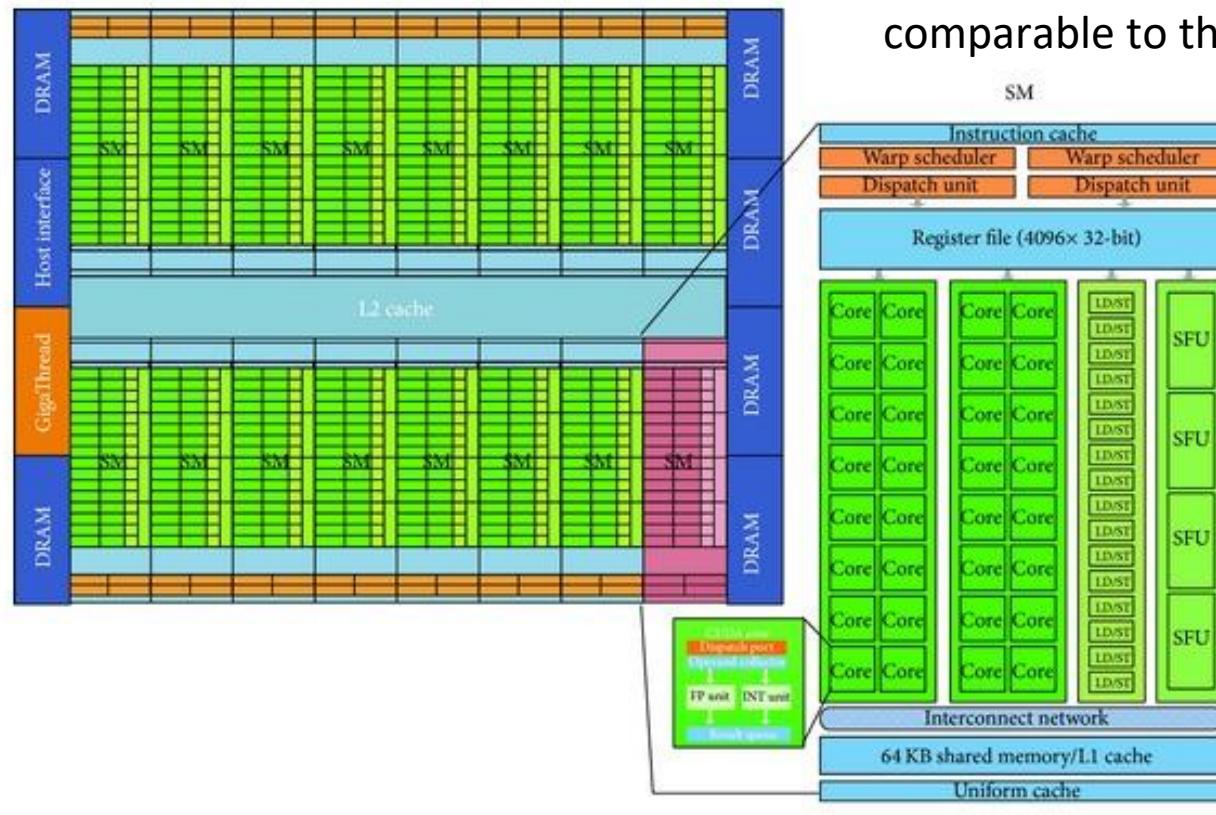
**Warp:** Minimum amount of threads (32) that will be executed simultaneously per SM



# Introduction

How does it look like on a real architecture?

Nvidia Fermi GPU (2011)

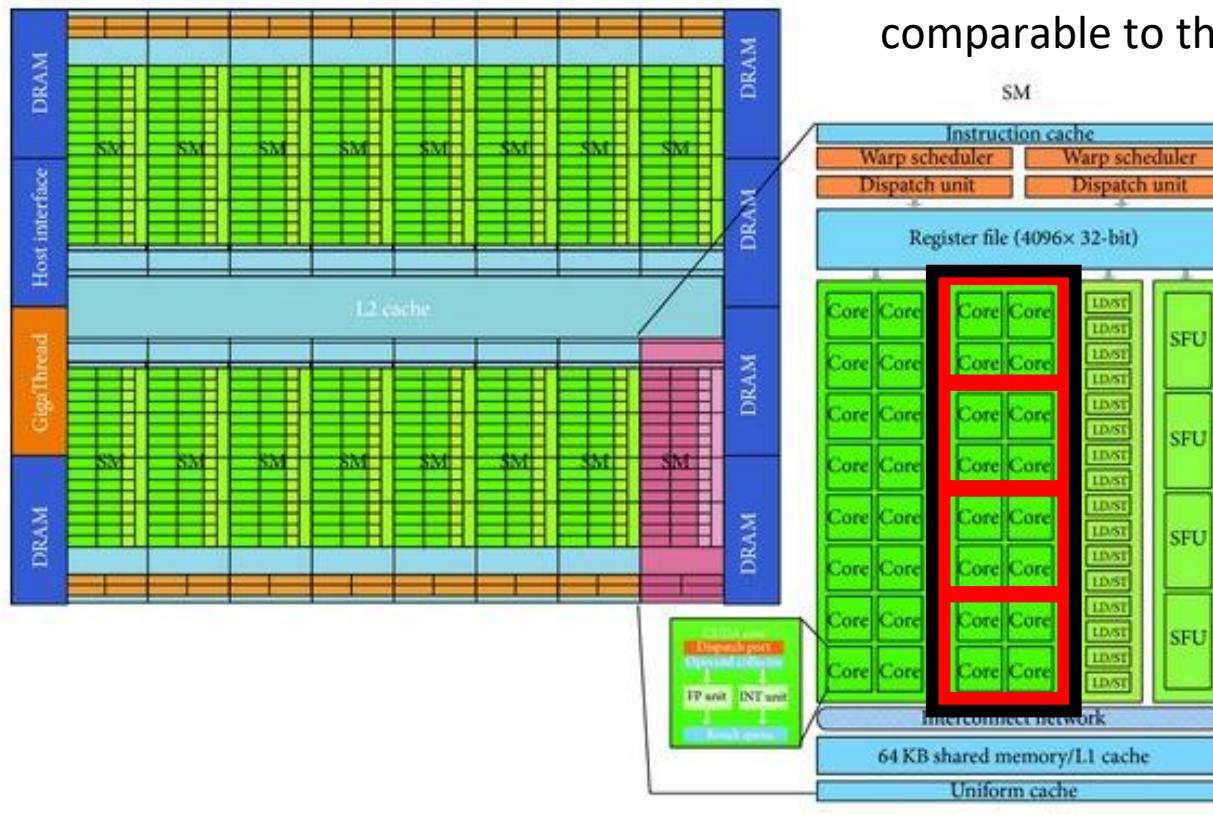


Here it is not a core, it is a  
"CUDA core" so not directly  
comparable to the CPU core

# Introduction

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

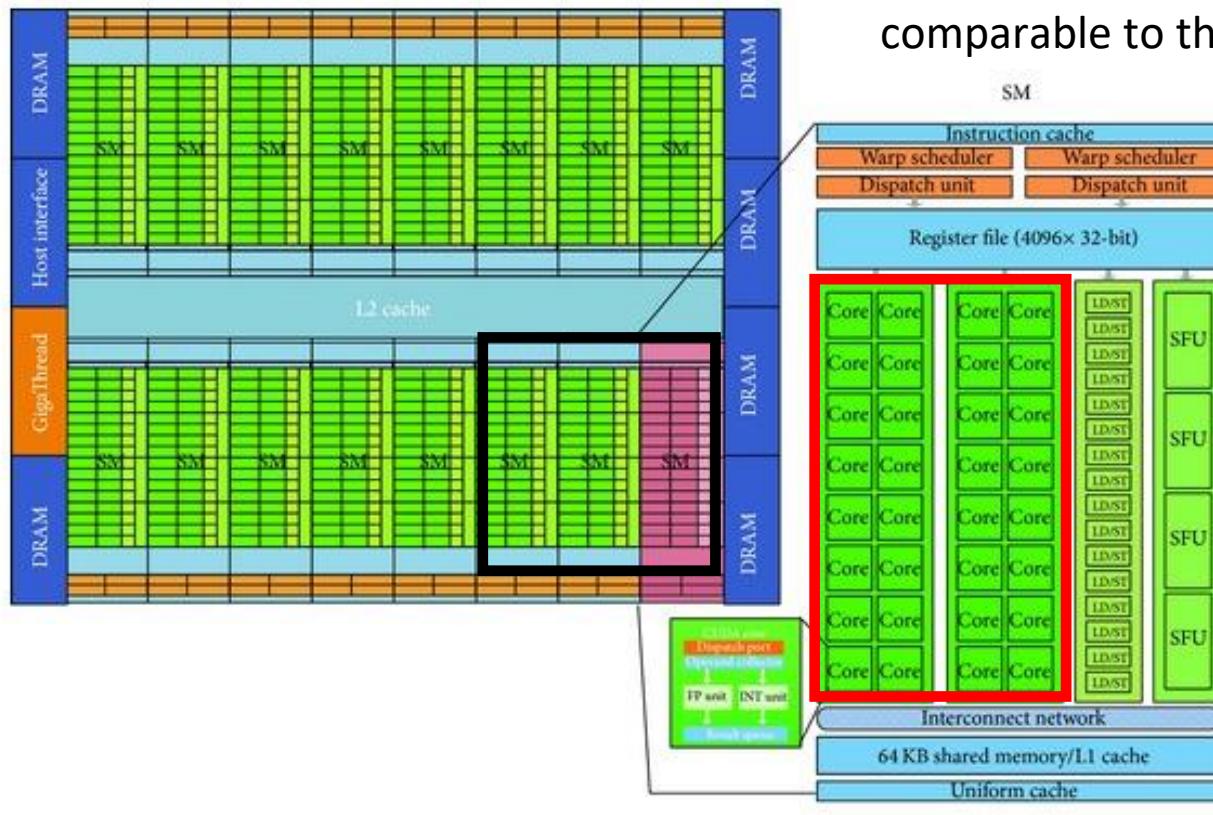
Block

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

# Introduction

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

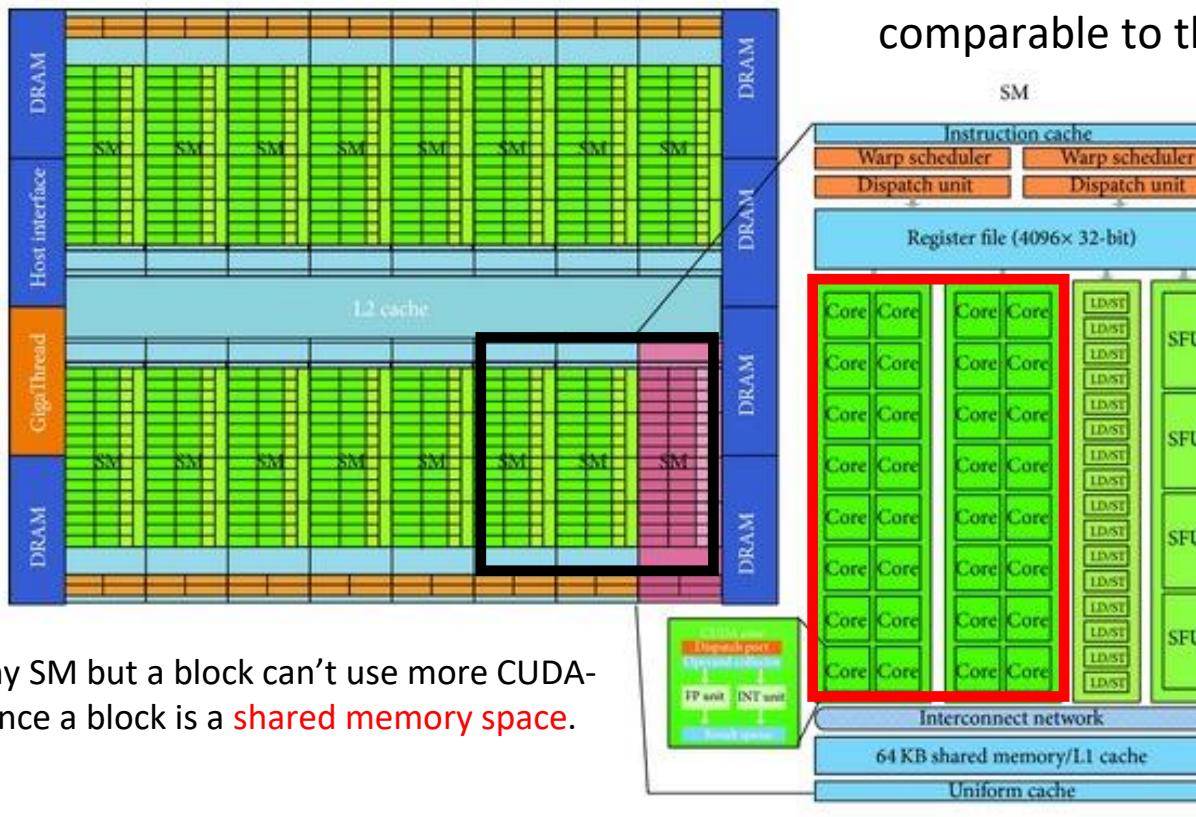
Block

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

# Introduction

How does it look like on a real architecture?

Nvidia Fermi GPU (2011)

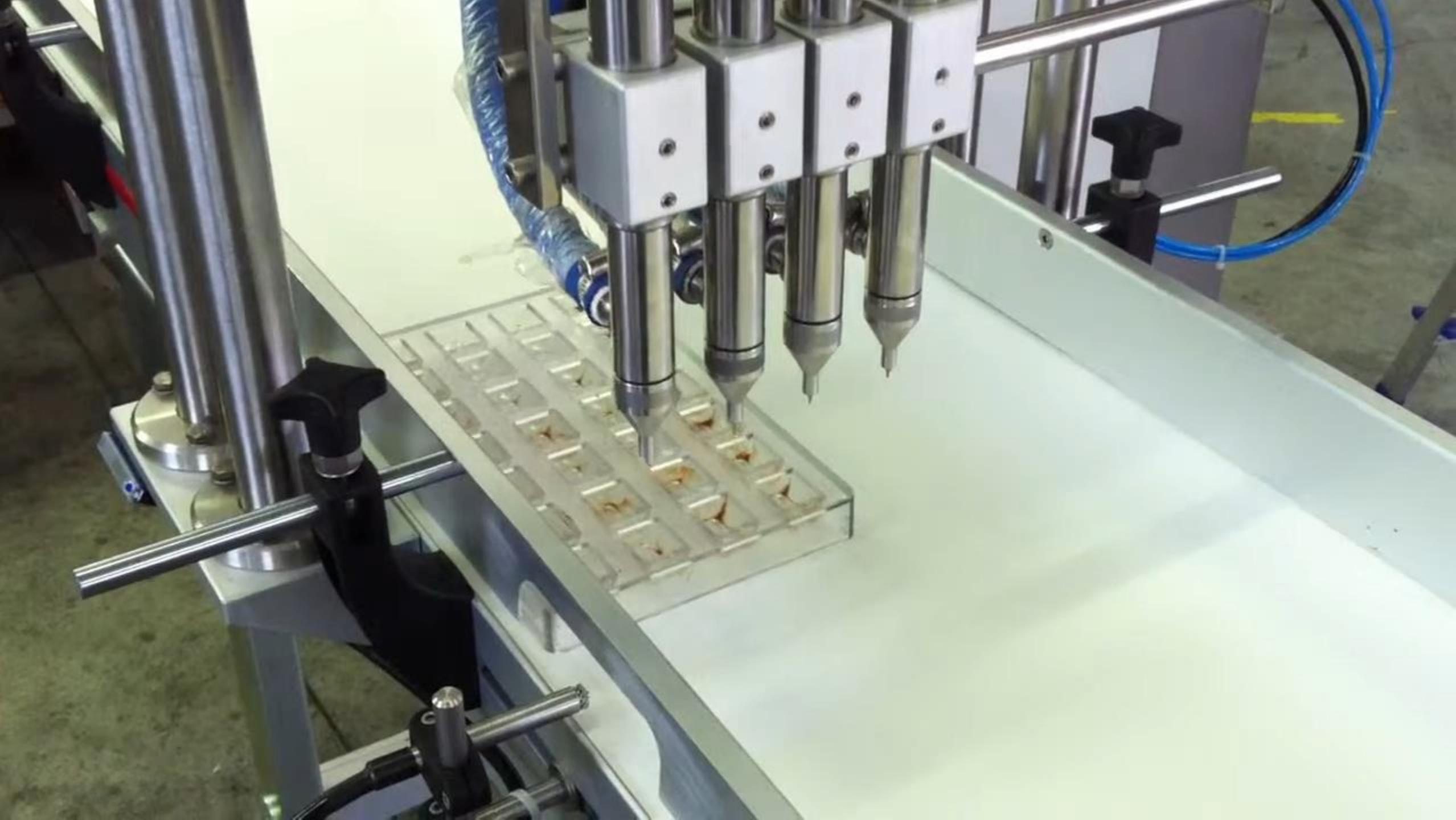


Here it is not a core, it is a "CUDA core" so not directly comparable to the CPU core

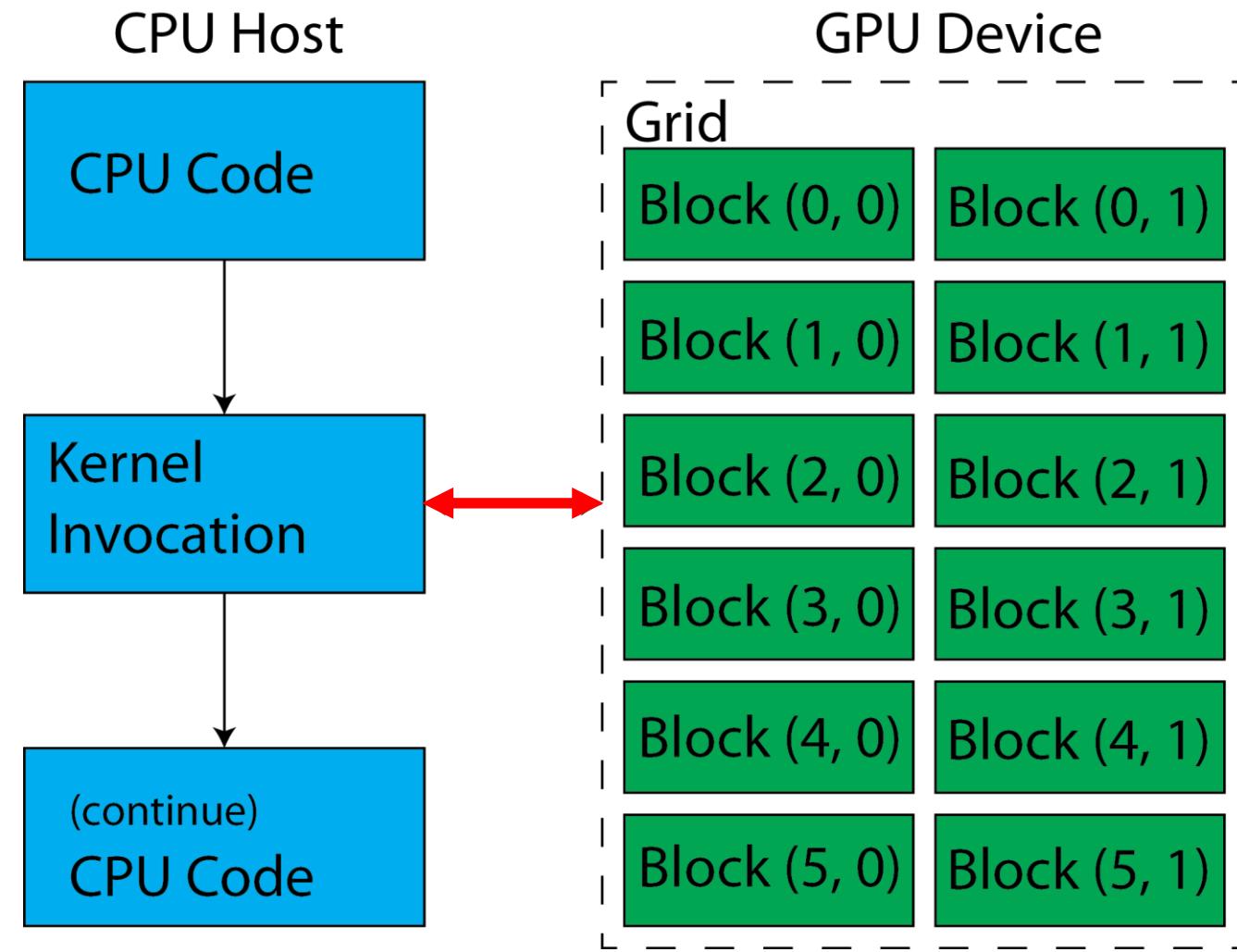
Grid

Block

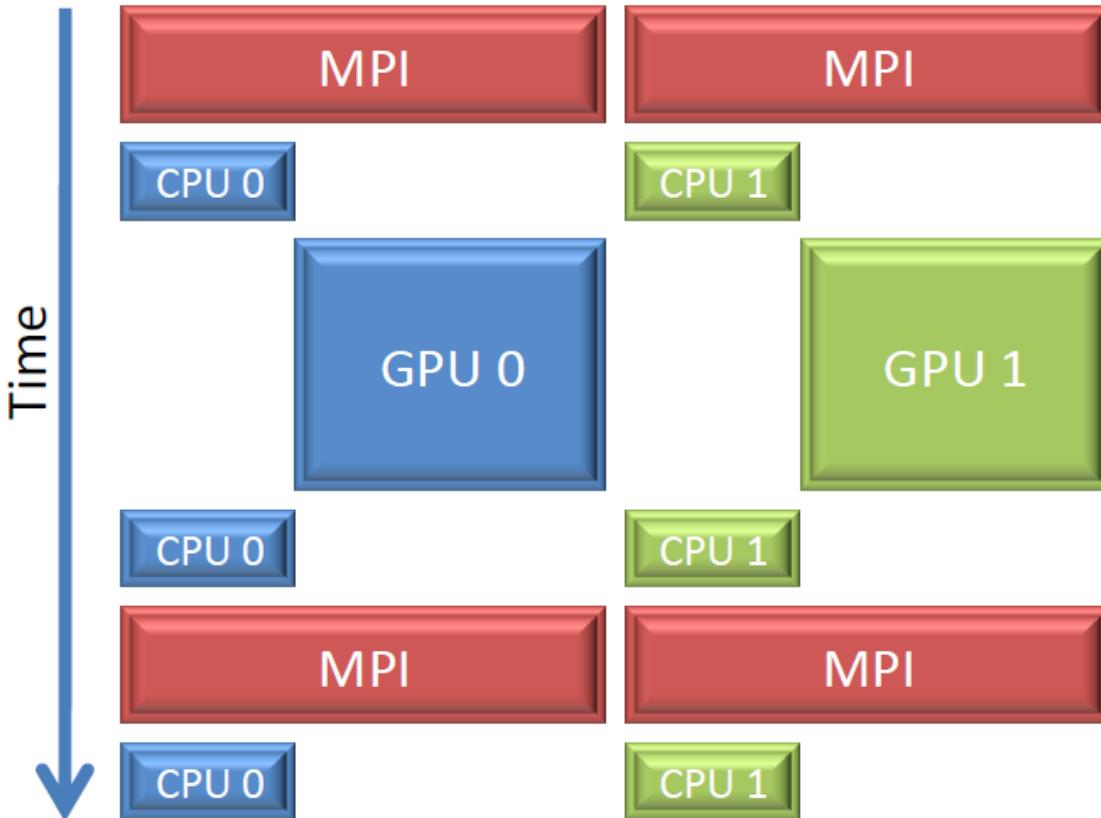
You can also set the number of threads per block, but this number is not easily linkable to the number of CUDA cores.  
Yet a **block size (number of threads per block)** should be a multiple of 32



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

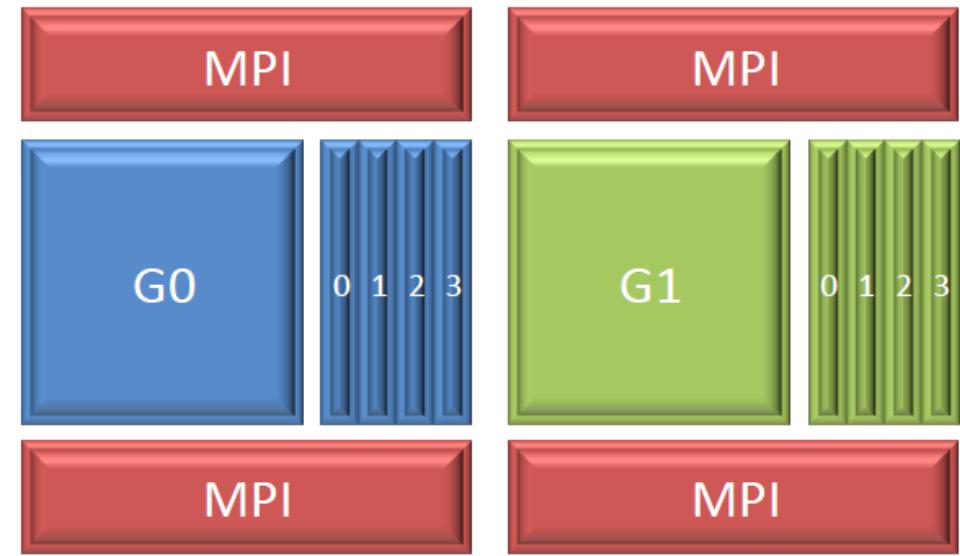


## So-So Hybridization



- Neglects CPU
- Suffers from Amdahl's Law

## Better Hybridization



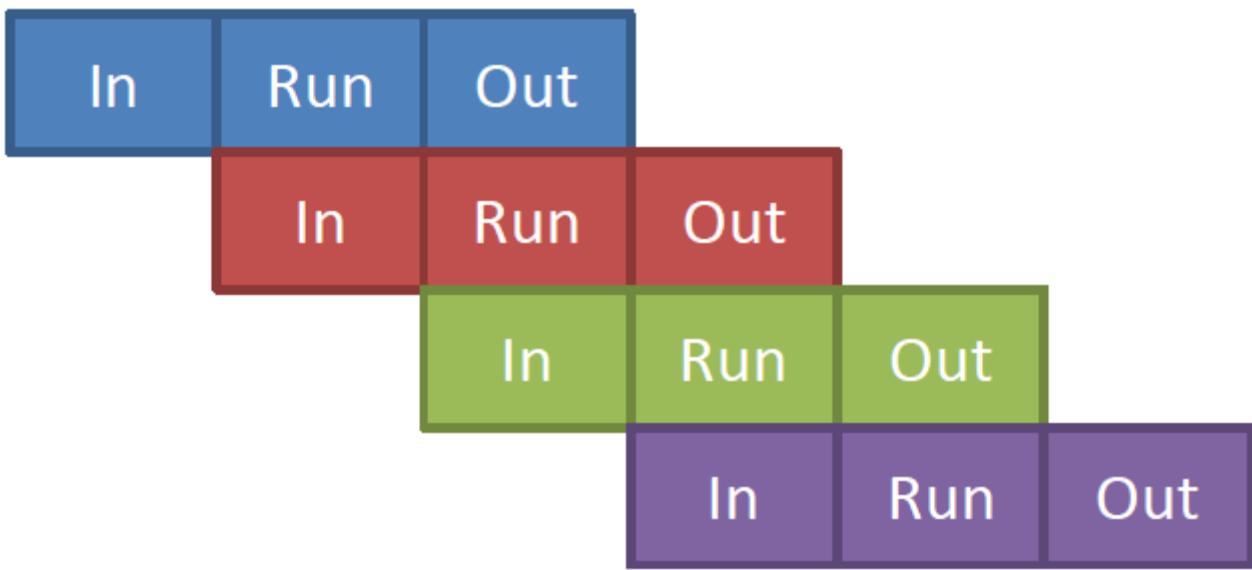
- Overlap CPU/GPU work and data movement
- Even better if you can overlap MPI comms

Most GPU operations are asynchronous from the CPU code  
(i.e. the CPU can be busy doing other things!)

Synchronous execution (1 stream)

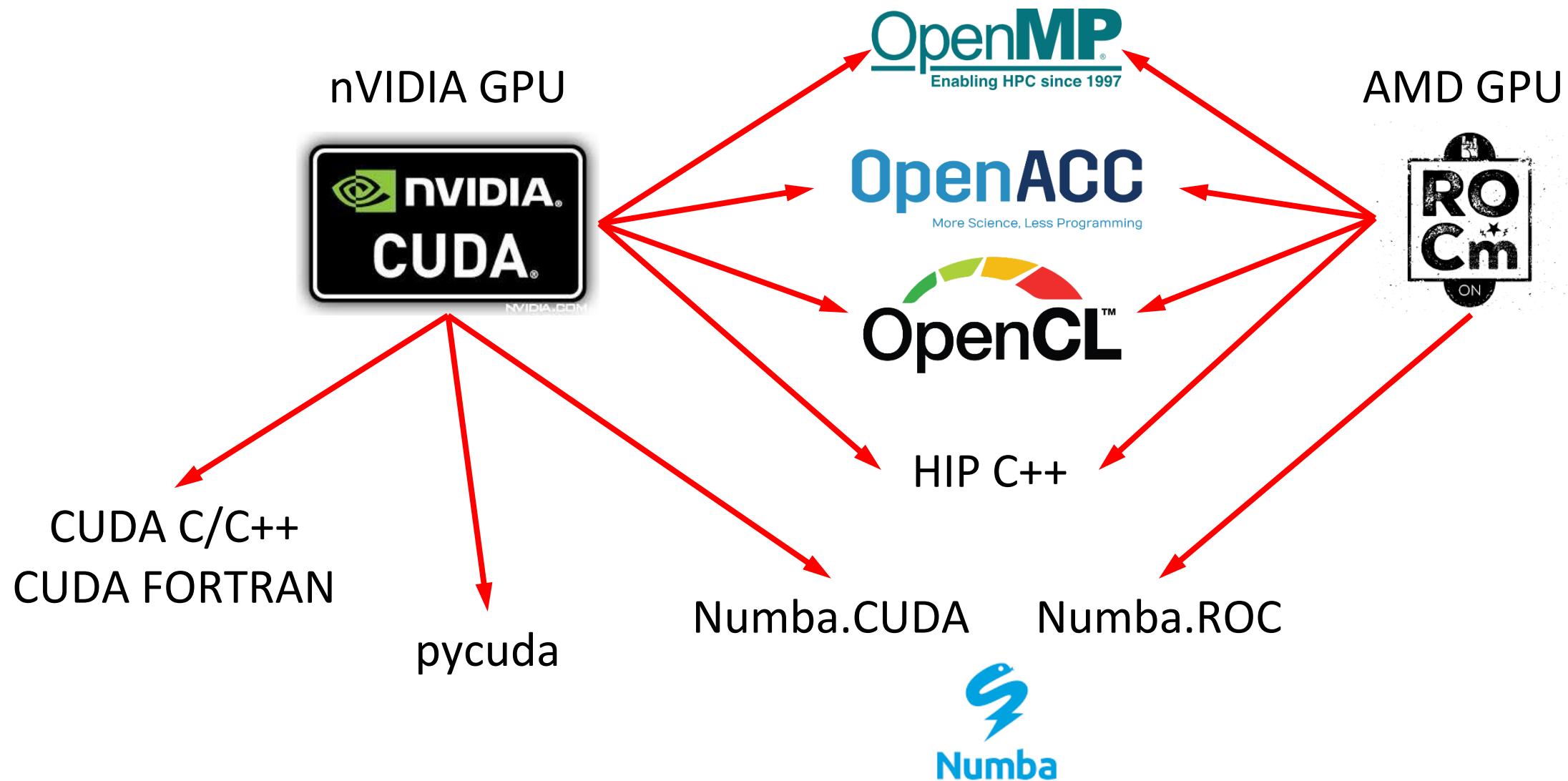


Asynchronous execution (3 streams)

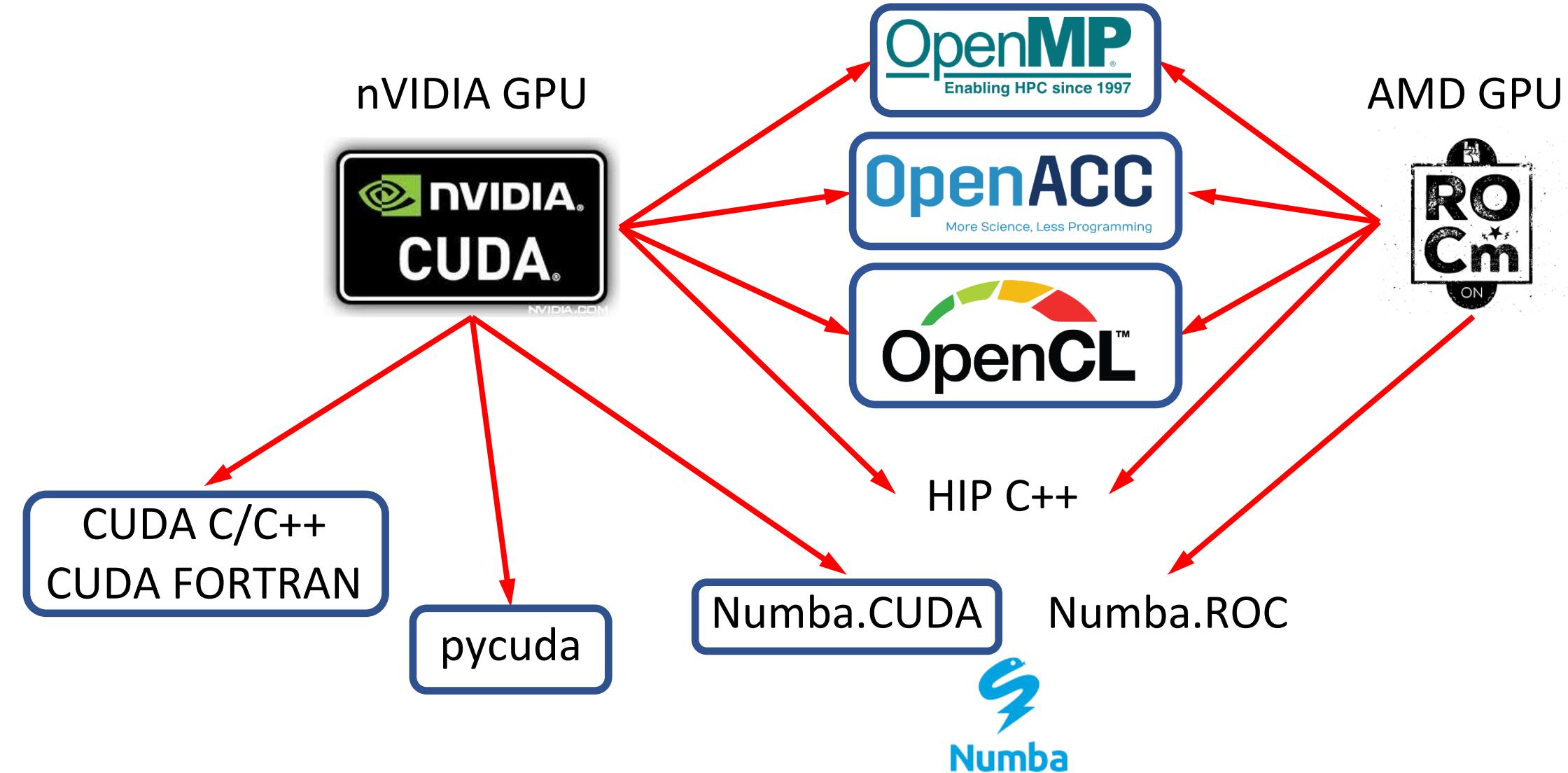


Asynchronous execution helps  
to hide the data transfer costs

## Writing your kernel : GPU programming paradigms



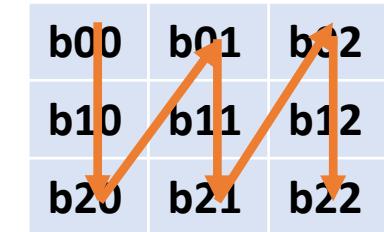
## Writing your kernel : GPU programming paradigms



**Sub-optimal memory access will kill the performance...**

The less efficient way (column-major):

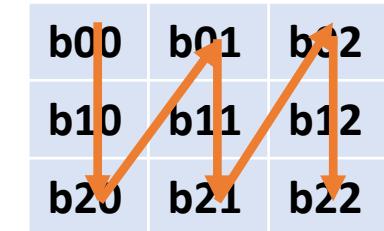
```
for(int i=0; i<n; ++i)
    for(int j=0; j<n; ++j)
        for(int k=0; k<n; ++k)
            c[i*n+j] += a[i*n+k] * b[k*n+j];
```



**Sub-optimal memory access will kill the performance...**

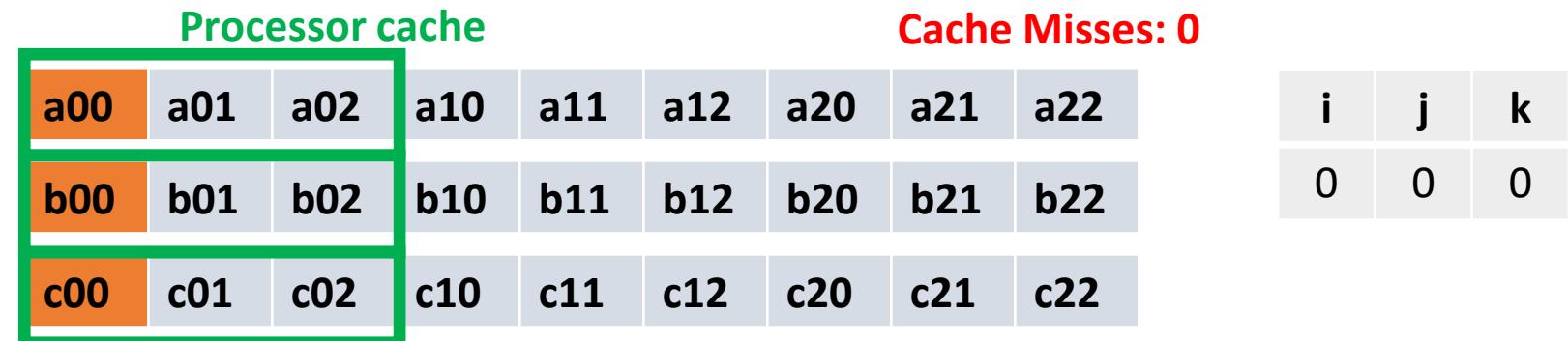
The less efficient way (column-major):

```
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**!

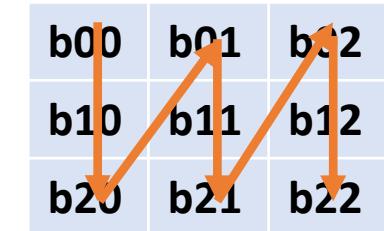
However here we are not helping ...



**Sub-optimal memory access will kill the performance...**

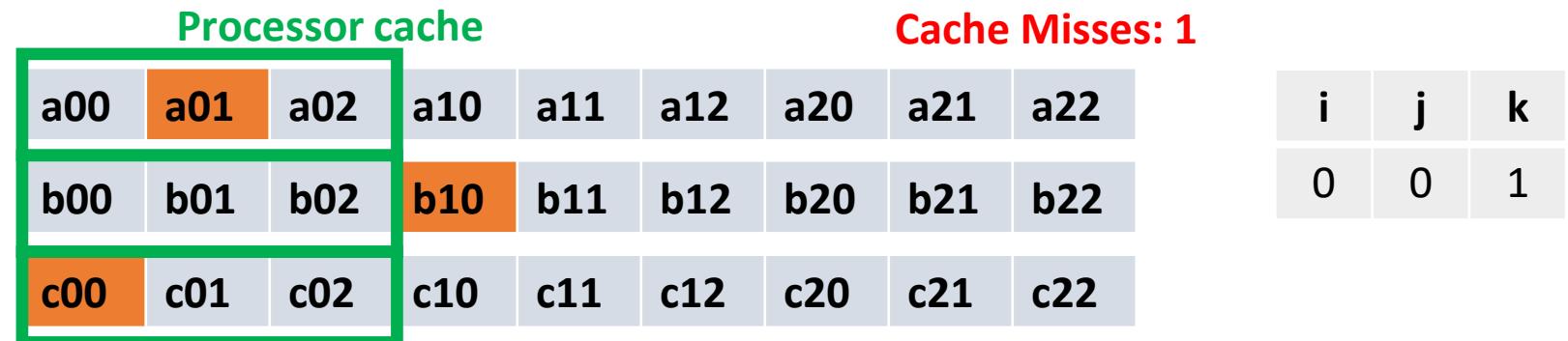
The less efficient way (column-major):

```
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**!

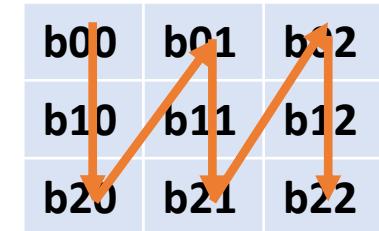
However here we are not helping ...



**Sub-optimal memory access will kill the performance...**

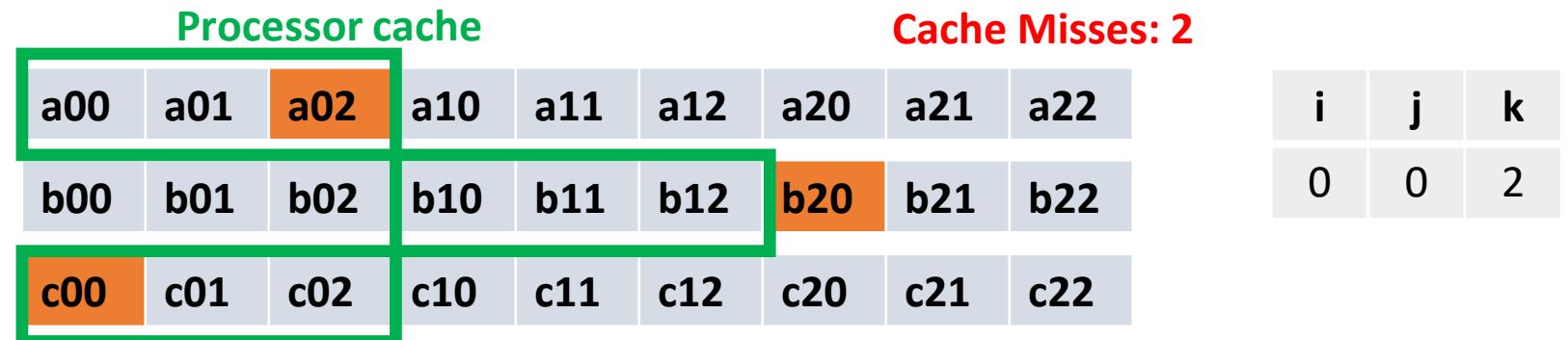
The less efficient way (column-major):

```
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**!

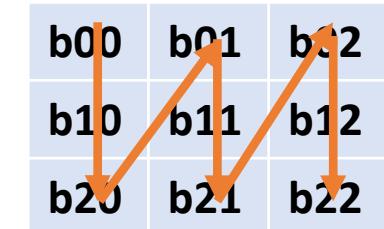
However here we are not helping ...



**Sub-optimal memory access will kill the performance...**

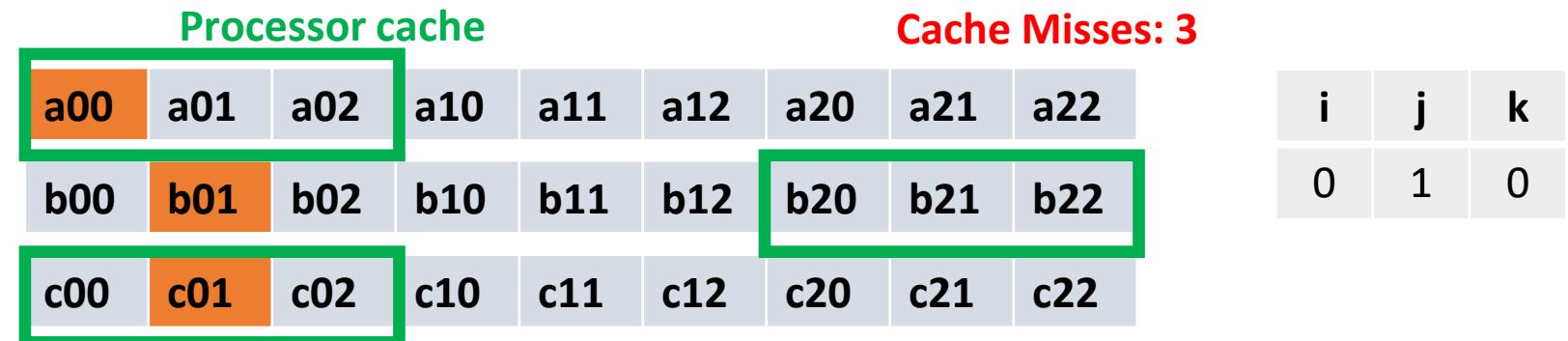
The less efficient way (column-major):

```
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**!

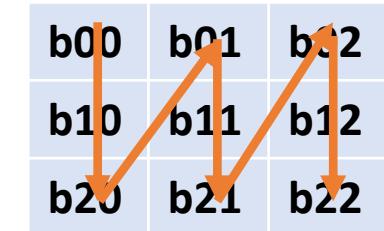
However here we are not helping ...



**Sub-optimal memory access will kill the performance...**

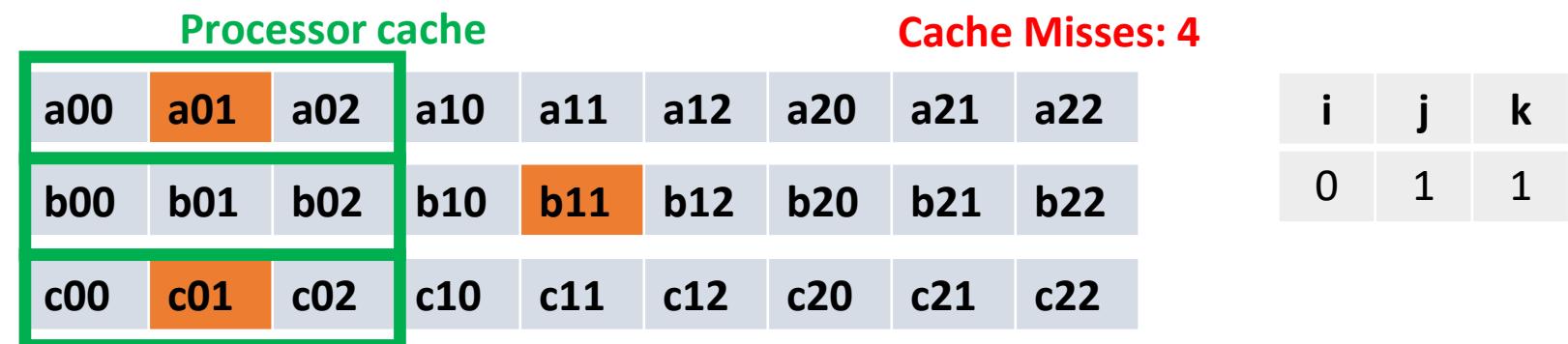
The less efficient way (column-major):

```
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**!

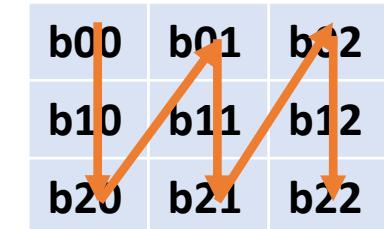
However here we are not helping ...



**Sub-optimal memory access will kill the performance...**

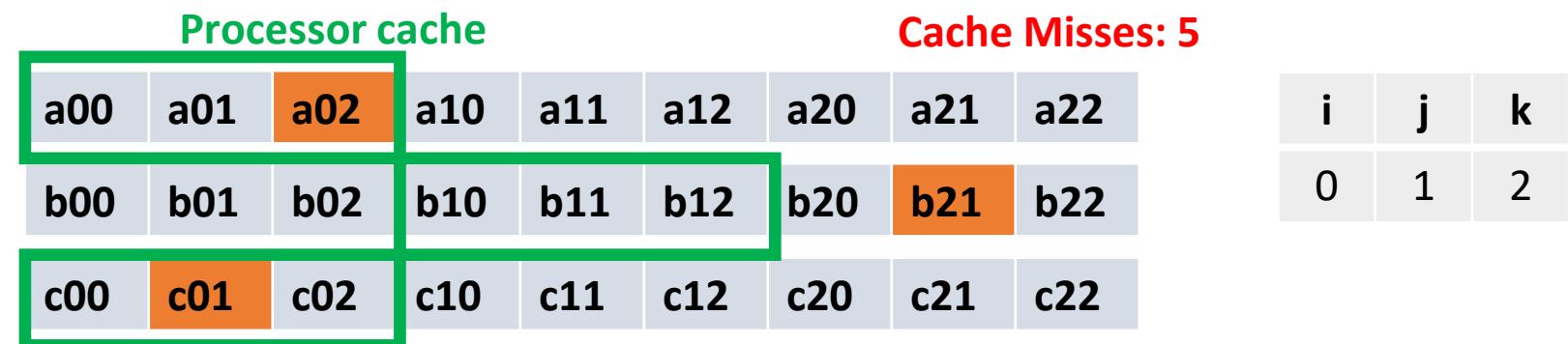
The less efficient way (column-major):

```
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**!

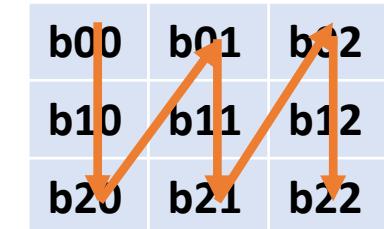
However here we are not helping ...



**Sub-optimal memory access will kill the performance...**

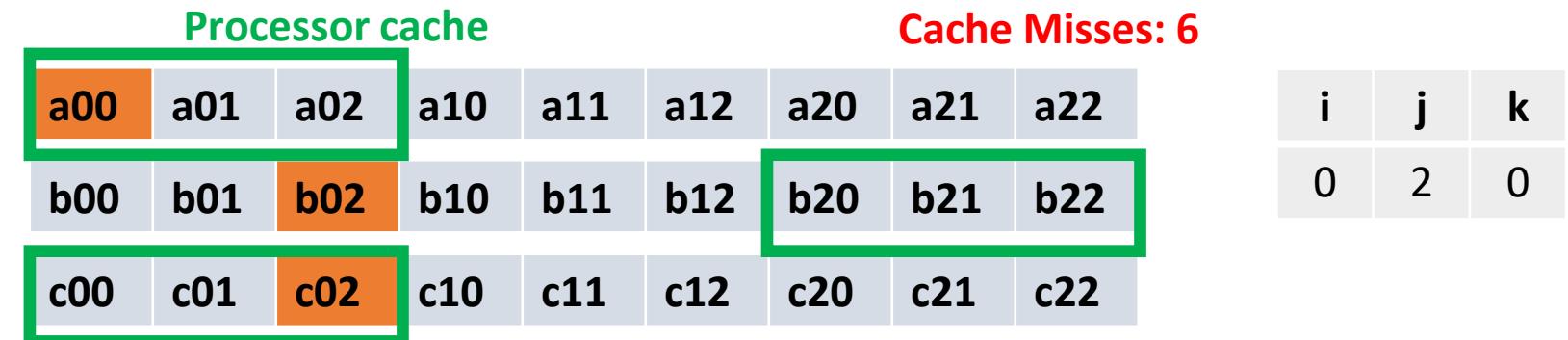
The less efficient way (column-major):

```
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**!

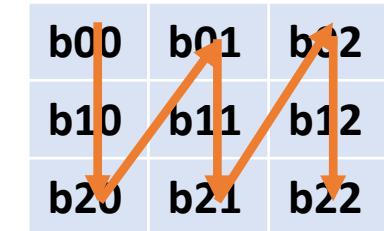
However here we are not helping ...



**Sub-optimal memory access will kill the performance...**

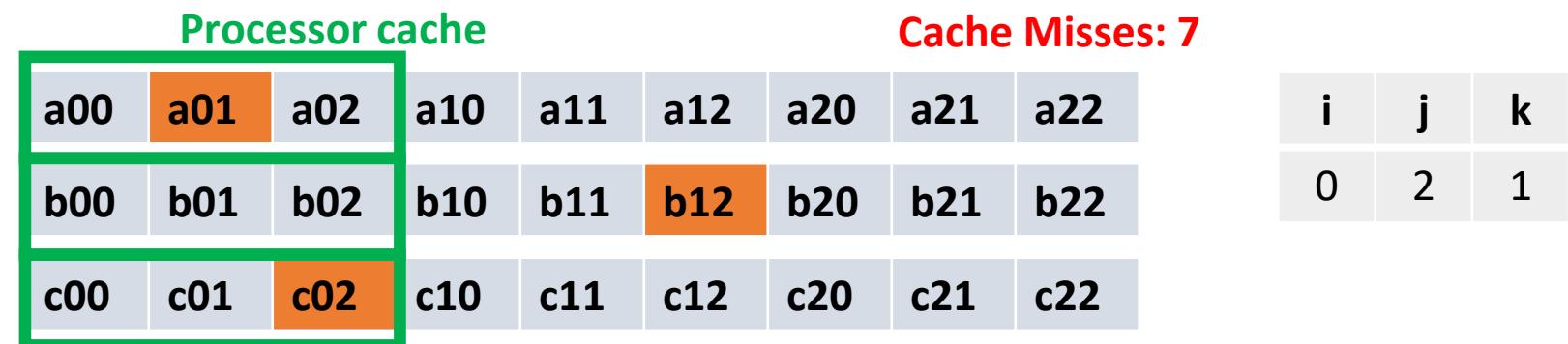
The less efficient way (column-major):

```
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**!

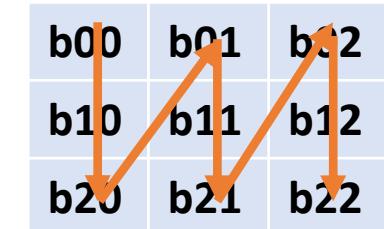
However here we are not helping ...



**Sub-optimal memory access will kill the performance...**

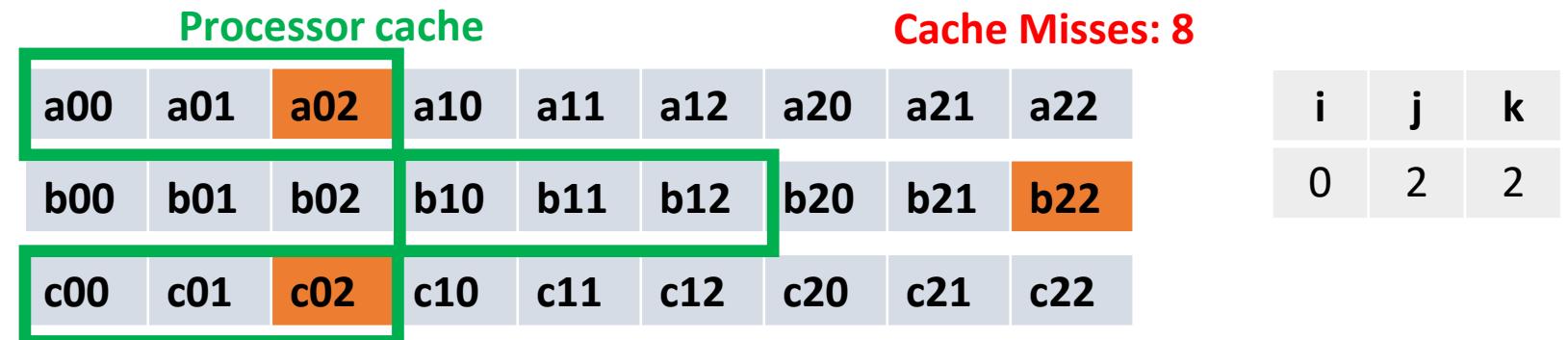
The less efficient way (column-major):

```
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**!

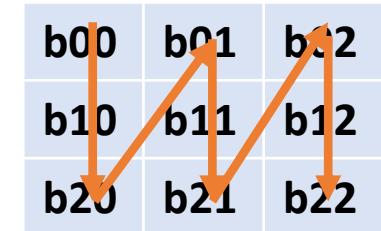
However here we are not helping ...



**Sub-optimal memory access will kill the performance...**

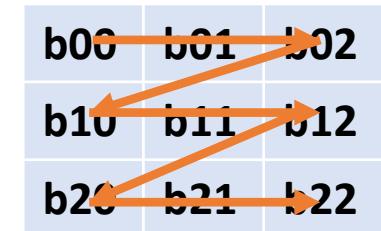
The less efficient way (column-major):

```
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):

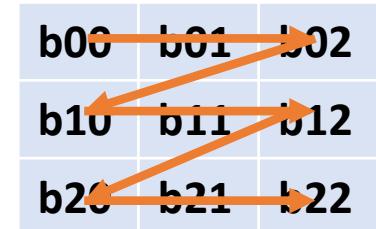
```
for(int i=0; i<n; ++i)
    for(int k=0; k<n; ++k)
        for(int j=0; j<n; ++j)
            c[i*n+j] += a[i*n+k] * b[k*n+j];
```



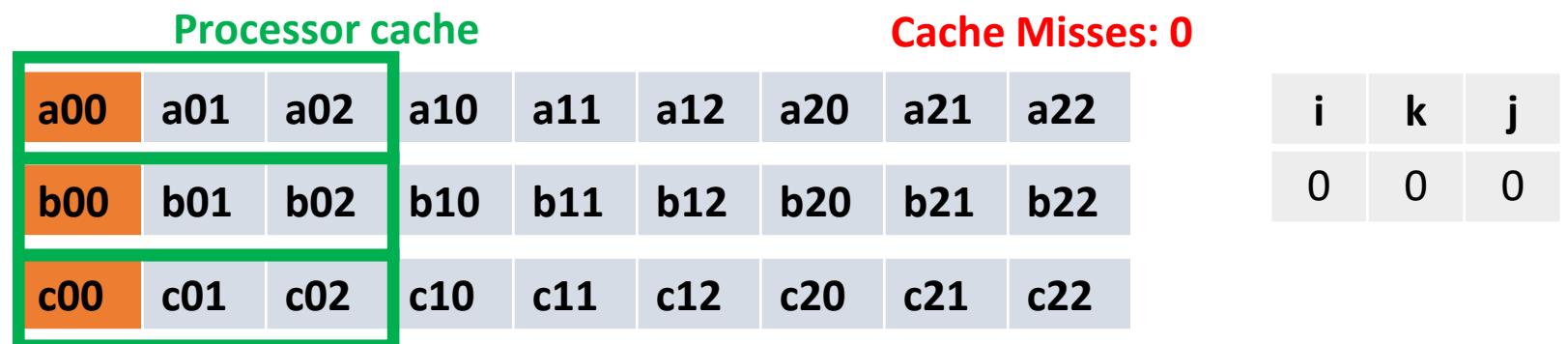
**Here, the most inner loop can  
be executed in parallel** (without  
write conflicts on matrix c)

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];
```

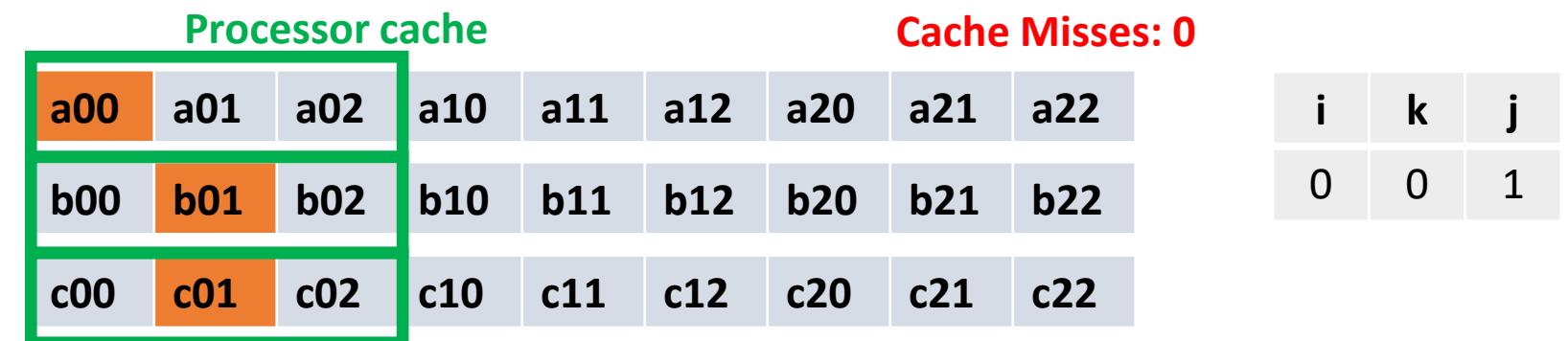
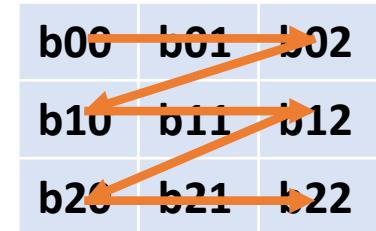


Let's go...



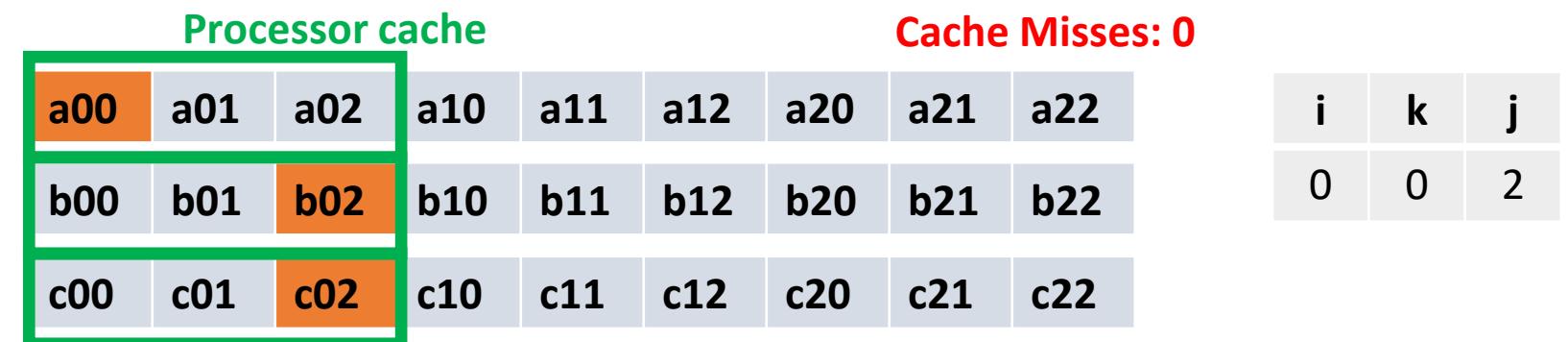
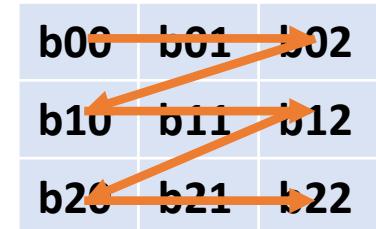
The better way (row-major):

```
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];
```



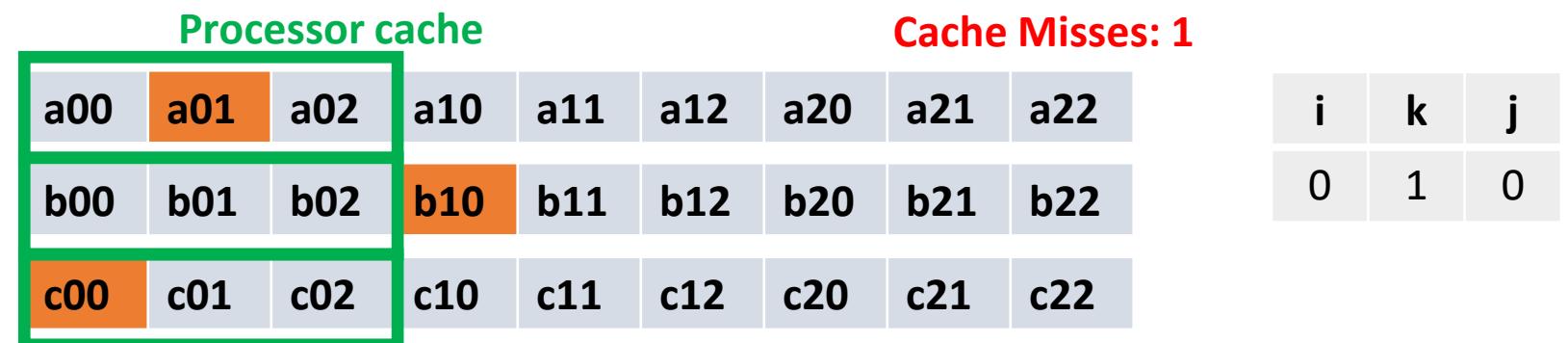
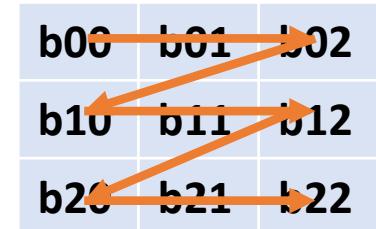
The better way (row-major):

```
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];
```



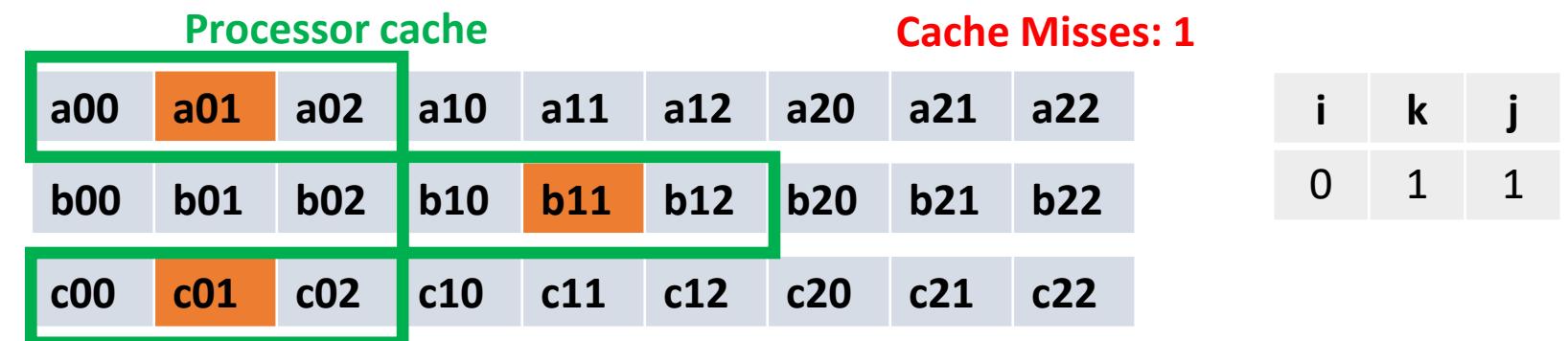
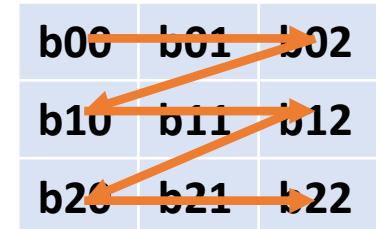
The better way (row-major):

```
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];
```



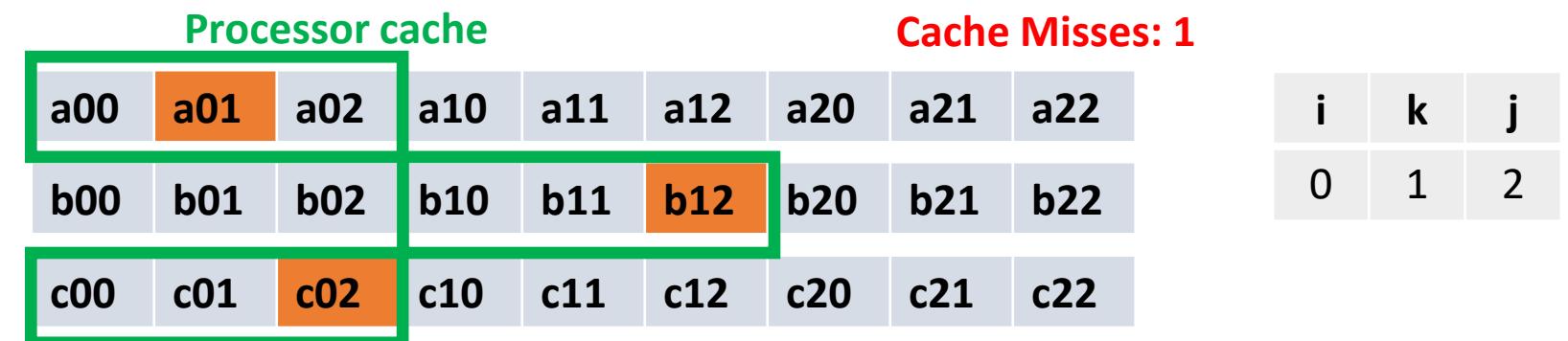
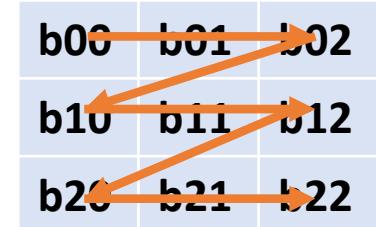
The better way (row-major):

```
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];
```



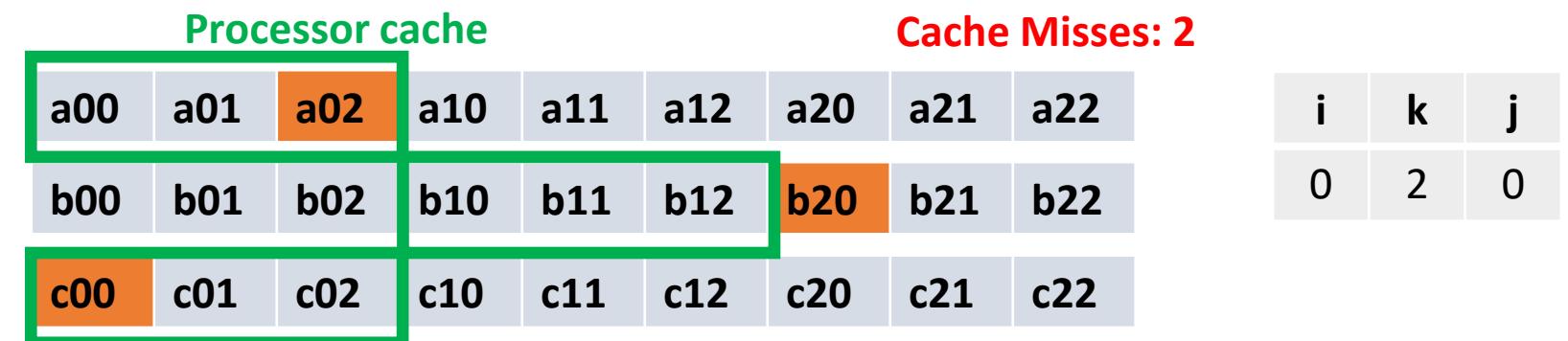
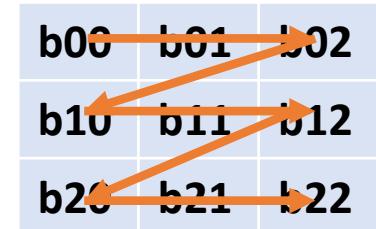
The better way (row-major):

```
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];
```



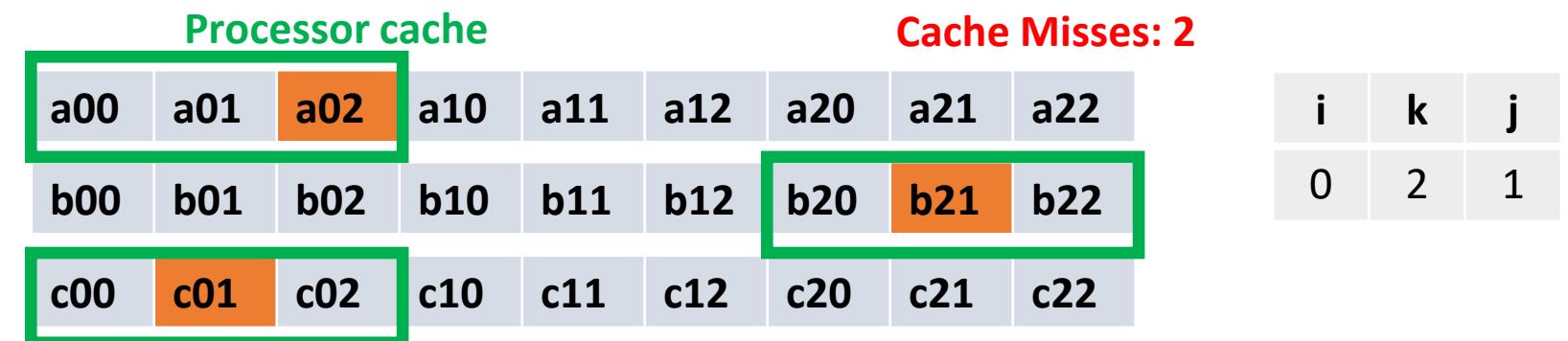
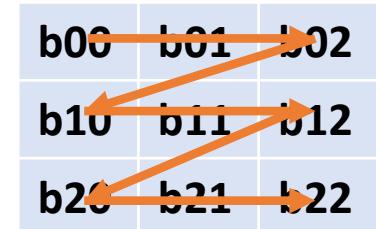
The better way (row-major):

```
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];
```



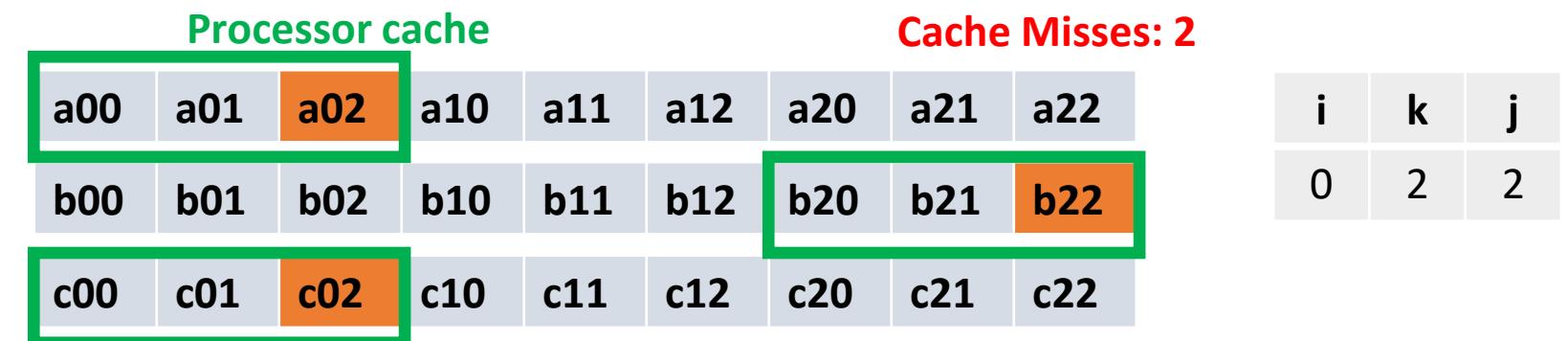
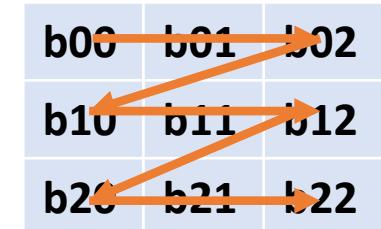
The better way (row-major):

```
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];
```



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];
```



# Matrix multiplication example

- 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

# Matrix multiplication example

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



# Matrix multiplication example

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!

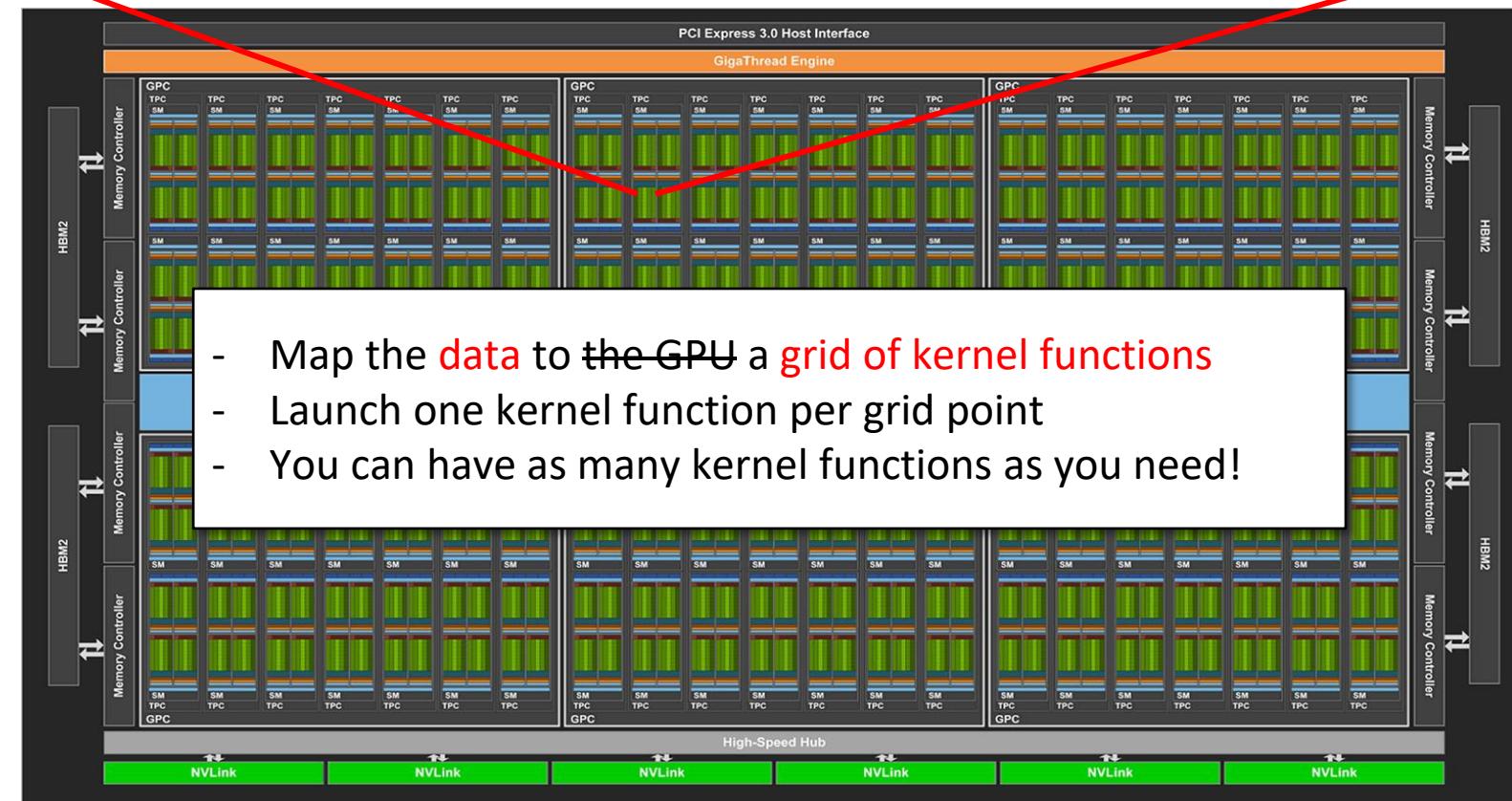
Remember, the data is not sent to individual threads

Instead, the data is mapped onto the grid of blocks

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

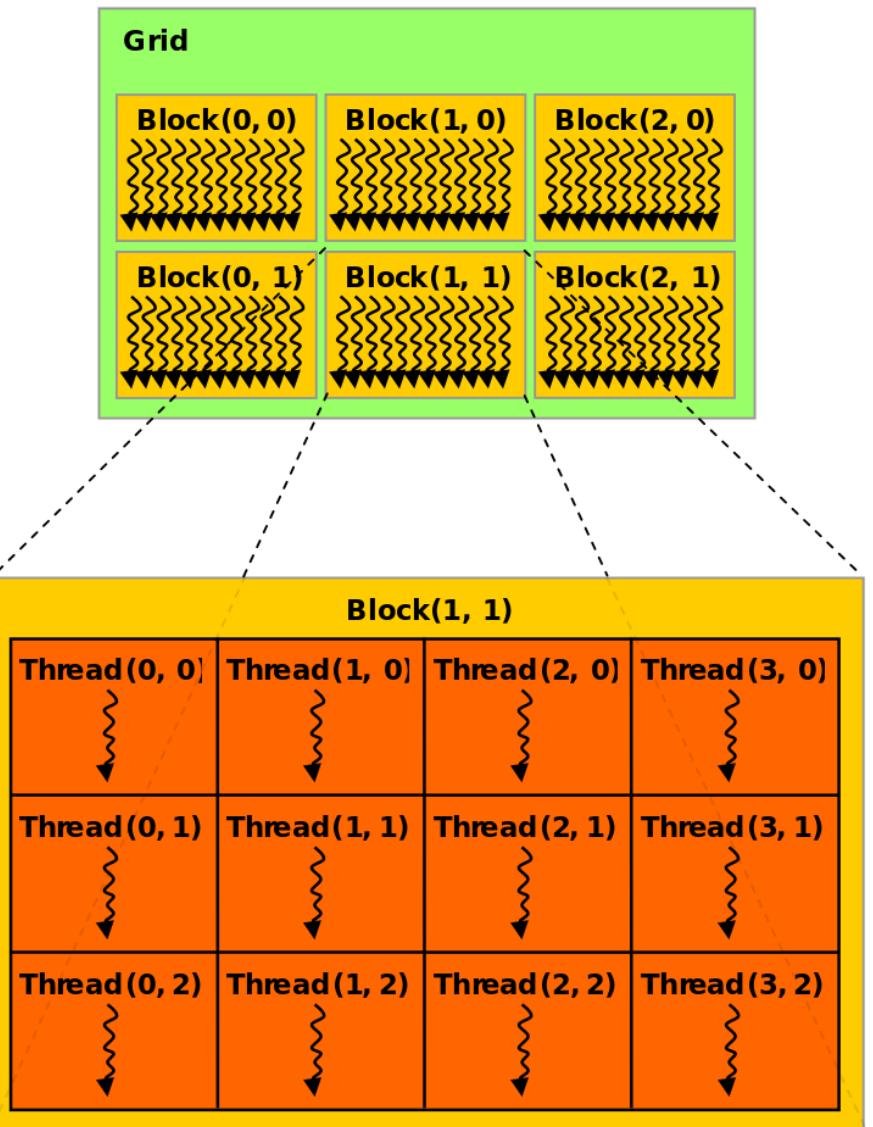
```
__global__ void cuda_mul(float* a, float* b, float* c, int size) {  
    int row = blockIdx.y*blockDim.y+threadIdx.y;  
    int col = blockIdx.x*blockDim.x+threadIdx.x;  
    for (int i = 0; i < size; i++)  
        c[row*size+col] += a[row*size+i] * b[i*size+col];  
}
```

CUDA Kernel



**Reminder:** Grids, blocks, threads and warps

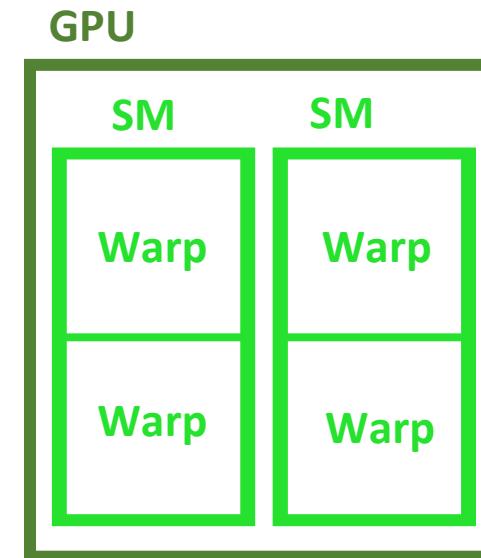
- Grid:
  - 1 to as many blocks as you want
- Block
  - 1 to 1024 threads
  - Should be a multiple of 32 threads
  - Execute on a single Streaming Multiprocessor (SM)
  - Possibility of having shared memory between threads
  - Is executed in **warps** of 32 threads
- Warps
  - Warps always execute 32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



# Execution of a block on a GPU

Grid (3x2 blocks)

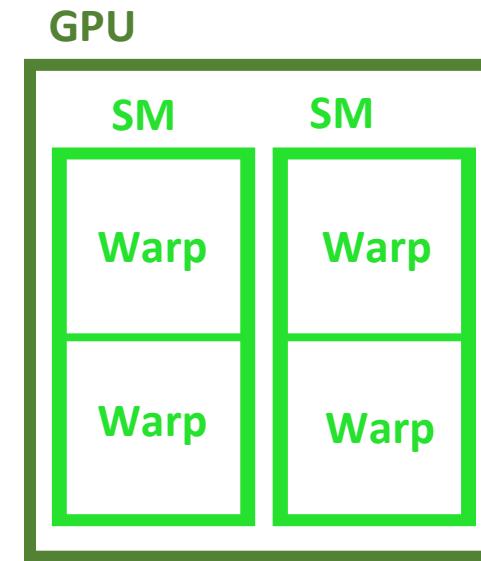
Block (0, 0) 128 threads	Block (0, 1) 128 threads	Block (0, 2) 128 threads
Block (1, 0) 128 threads	Block (1, 1) 128 threads	Block (1, 2) 128 threads



# Execution of a block on a GPU

Grid (3x2 blocks)

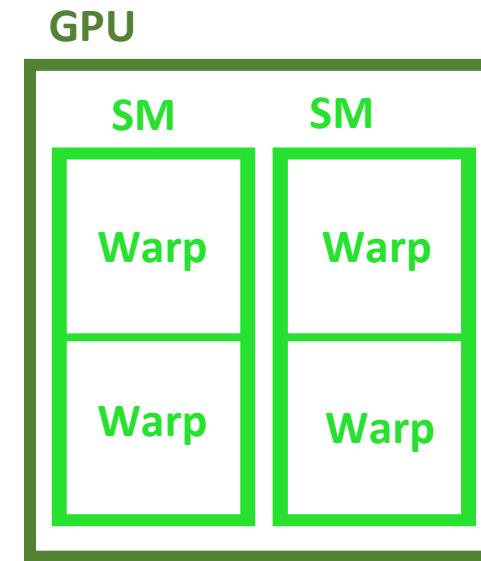
Block (0, 0) 128 threads	Block (0, 1) 128 threads	Block (0, 2) 128 threads
Block (1, 0) 128 threads	Block (1, 1) 128 threads	Block (1, 2) 128 threads



# Execution of a block on a GPU

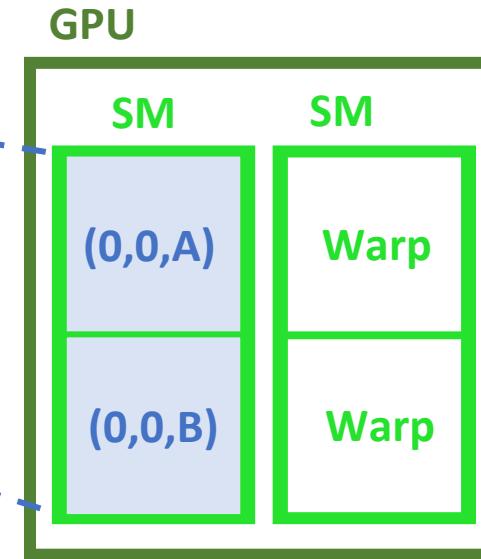
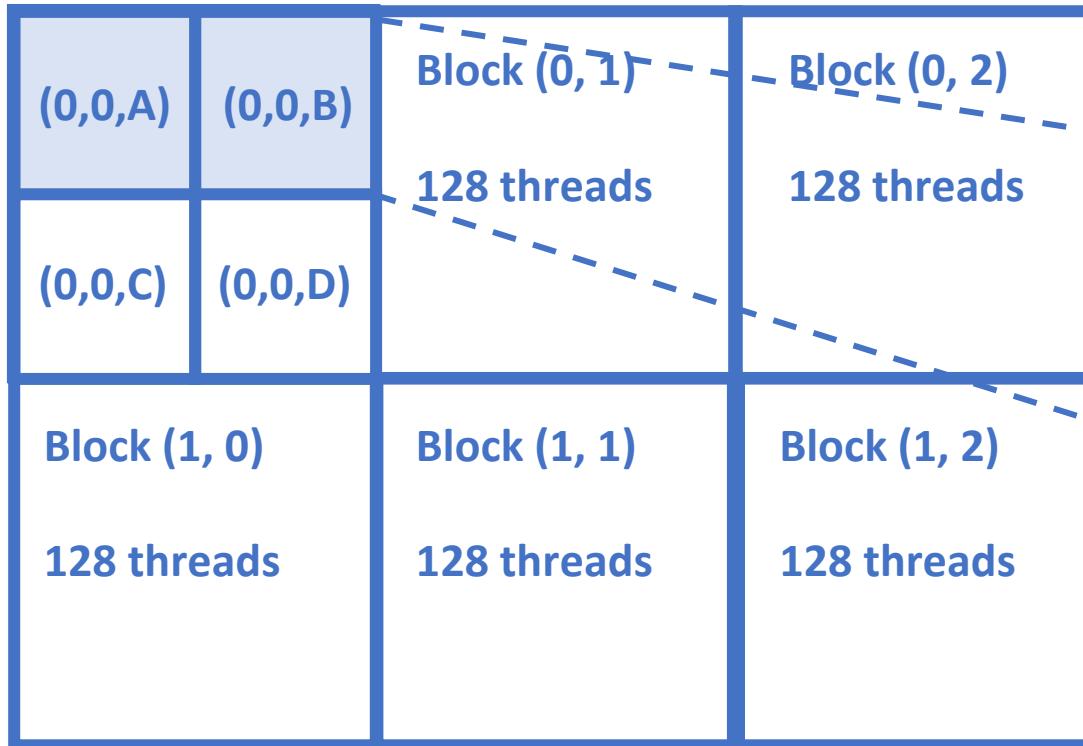
Grid (3x2 blocks)

32 threads	32 threads	Block (0, 1) 128 threads	Block (0, 2) 128 threads
32 threads	32 threads		
Block (1, 0) 128 threads	Block (1, 1) 128 threads	Block (1, 2) 128 threads	



# Execution of a block on a GPU

Grid (3x2 blocks)

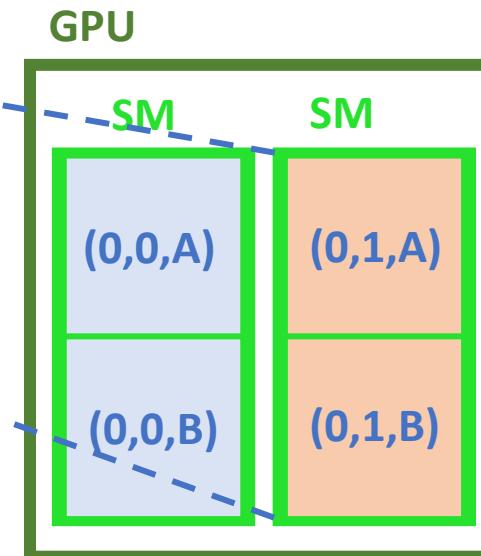
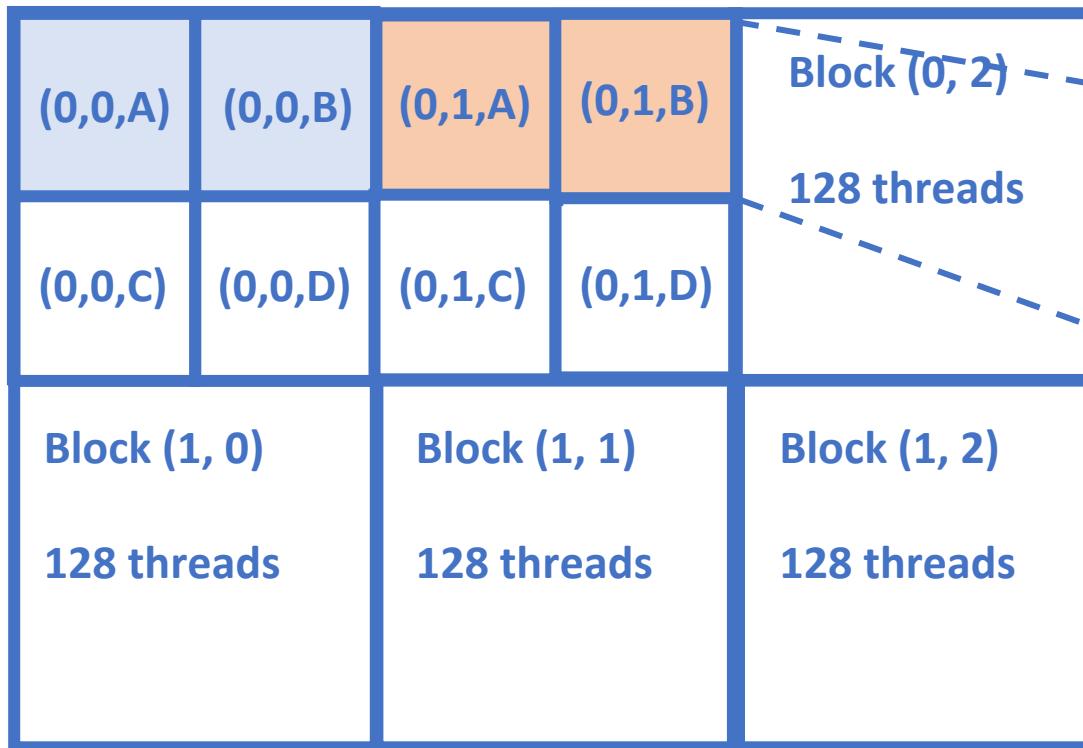


**Remember: a block can only be executed by a single SM!**

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

# Execution of a block on a GPU

Grid (3x2 blocks)

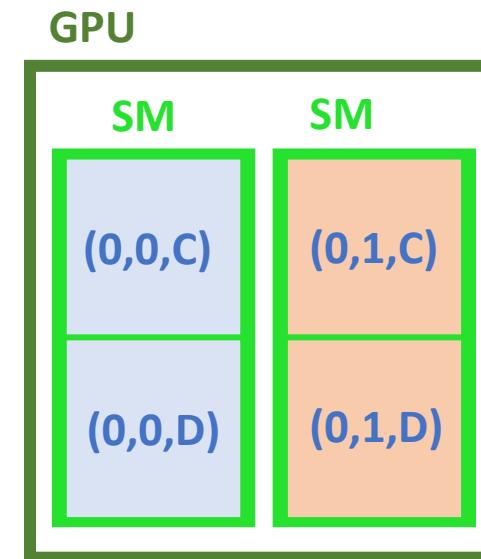
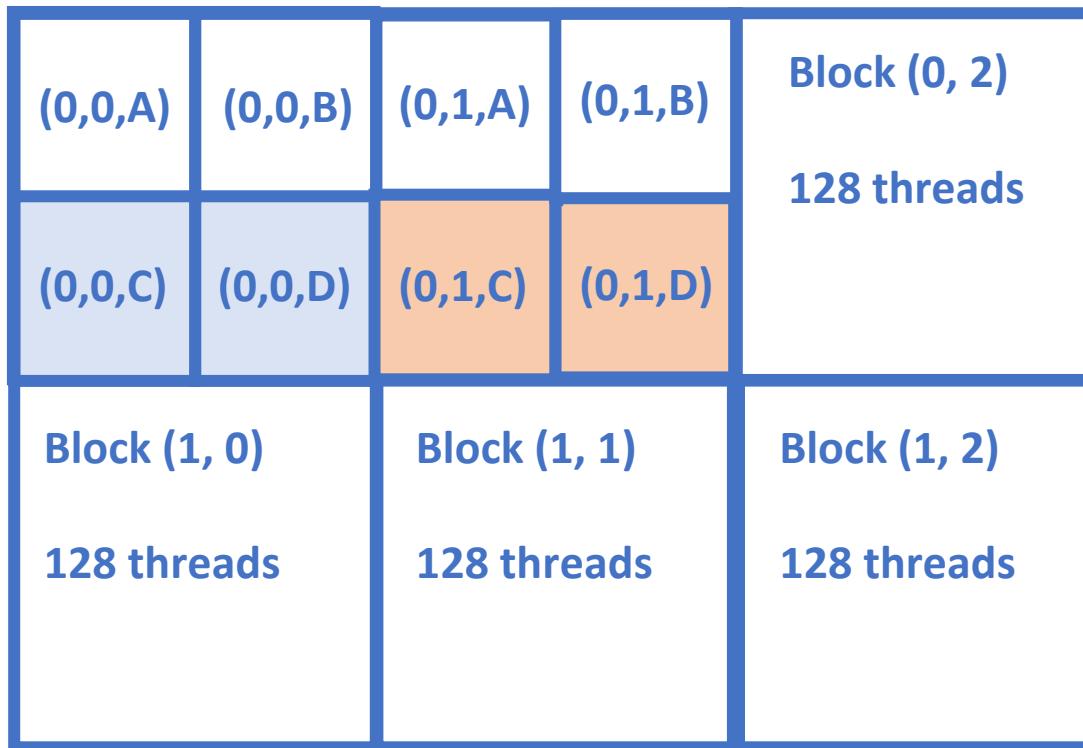


**Remember: a block can only be executed by a single SM!**

- 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%

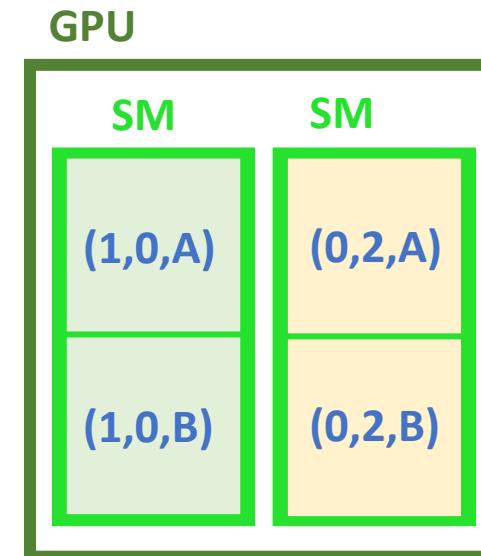
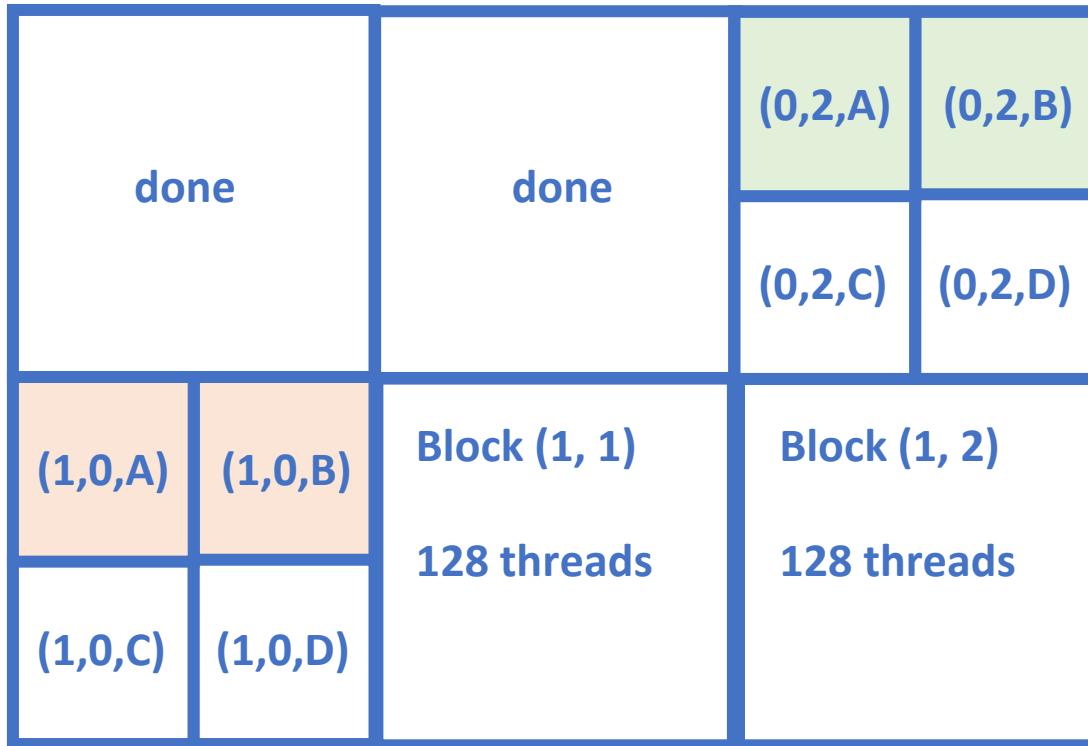
# Execution of a block on a GPU

Grid (3x2 blocks)



# Execution of a block on a GPU

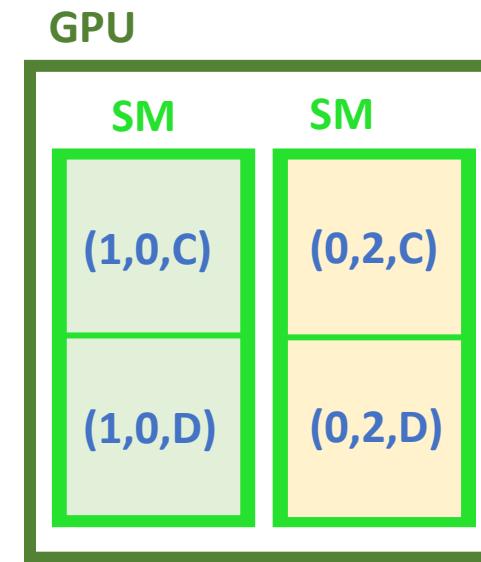
Grid (3x2 blocks)



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

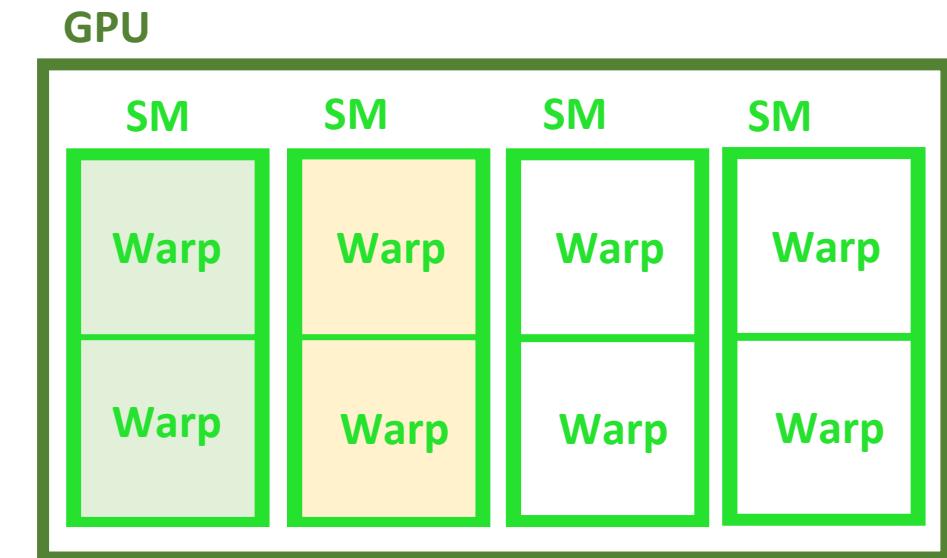
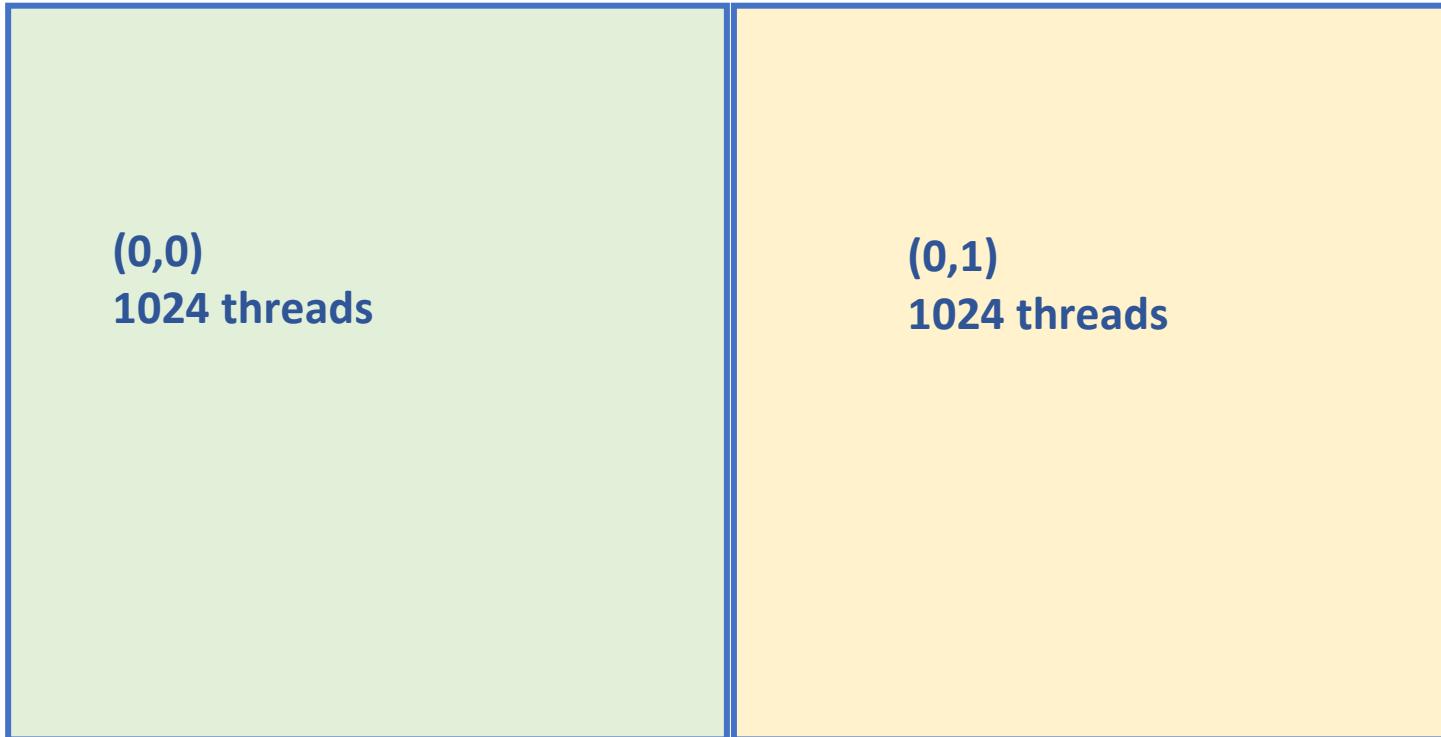
# Execution of a block on a GPU

Grid (3x2 blocks)



# Execution of a block on a GPU

## Grid (2 blocks)



**50% usage**

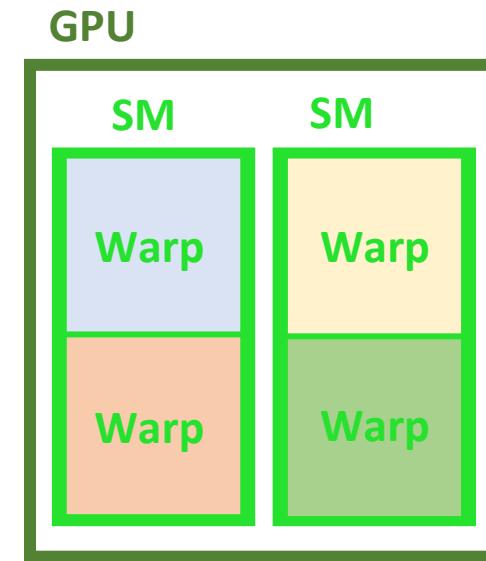
**Not enough blocks can result in underutilization!**

# Execution of a block on a GPU

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

- ❑ Introduction

- ❑ Example

- ❑ Matrix Multiplication + Exercise

- ❑ Real world implementations

- ❑ Genomes simulation
  - ❑ Random numbers generation

- ❑ 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
  - ❑ Genomes simulation
  - ❑ Random numbers generation
- ❑ Common pitfalls
- ❑ Memory management
  - ❑ Image processing example
- ❑ Monitoring and asynchronicity
  - ❑ Smoothed Particle Hydrodynamics
- ❑ Final remarks



## 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 `threadidx.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 (threadidx.x >= 16)
34 {
35     myFunc_else();
36     __syncthread();
37 }
38
```

# Outline

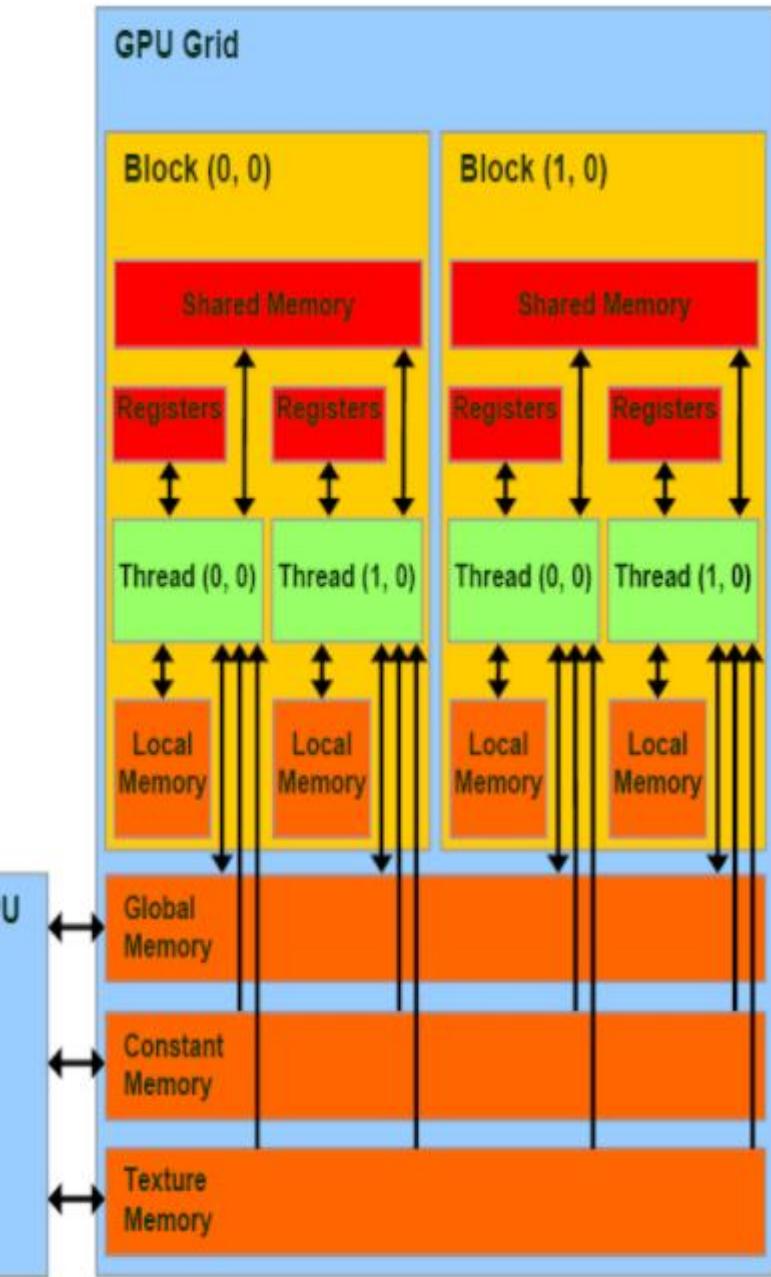
- ❑ Introduction
- ❑ Example
  - ❑ Matrix Multiplication + Exercise
- ❑ Real world implementations
  - ❑ Genomes simulation
  - ❑ Random numbers generation
- ❑ Common pitfalls
- ❑ Memory management
  - ❑ Image processing example
- ❑ Monitoring and asynchronicity
  - ❑ Smoothed Particle Hydrodynamics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



# Memory management

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 has 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



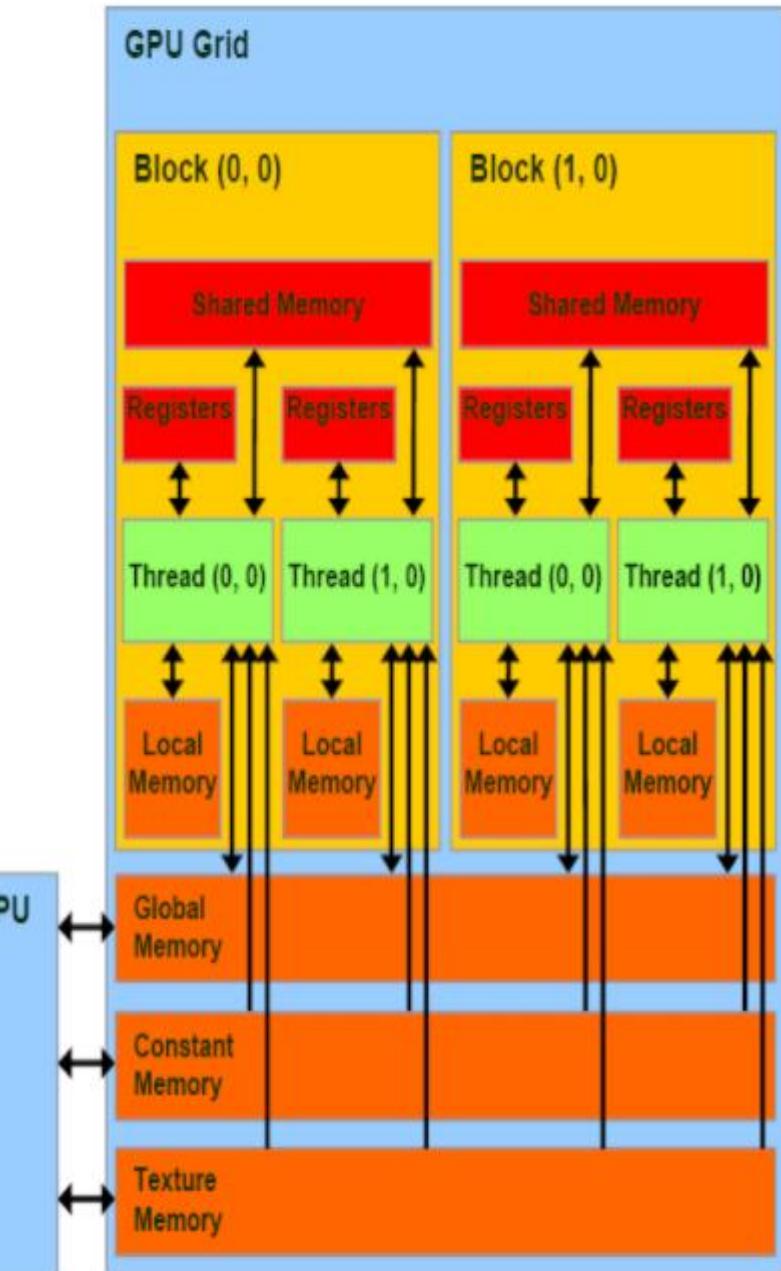
Source: cuda-programming.blogspot.com

# Memory management

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 has 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

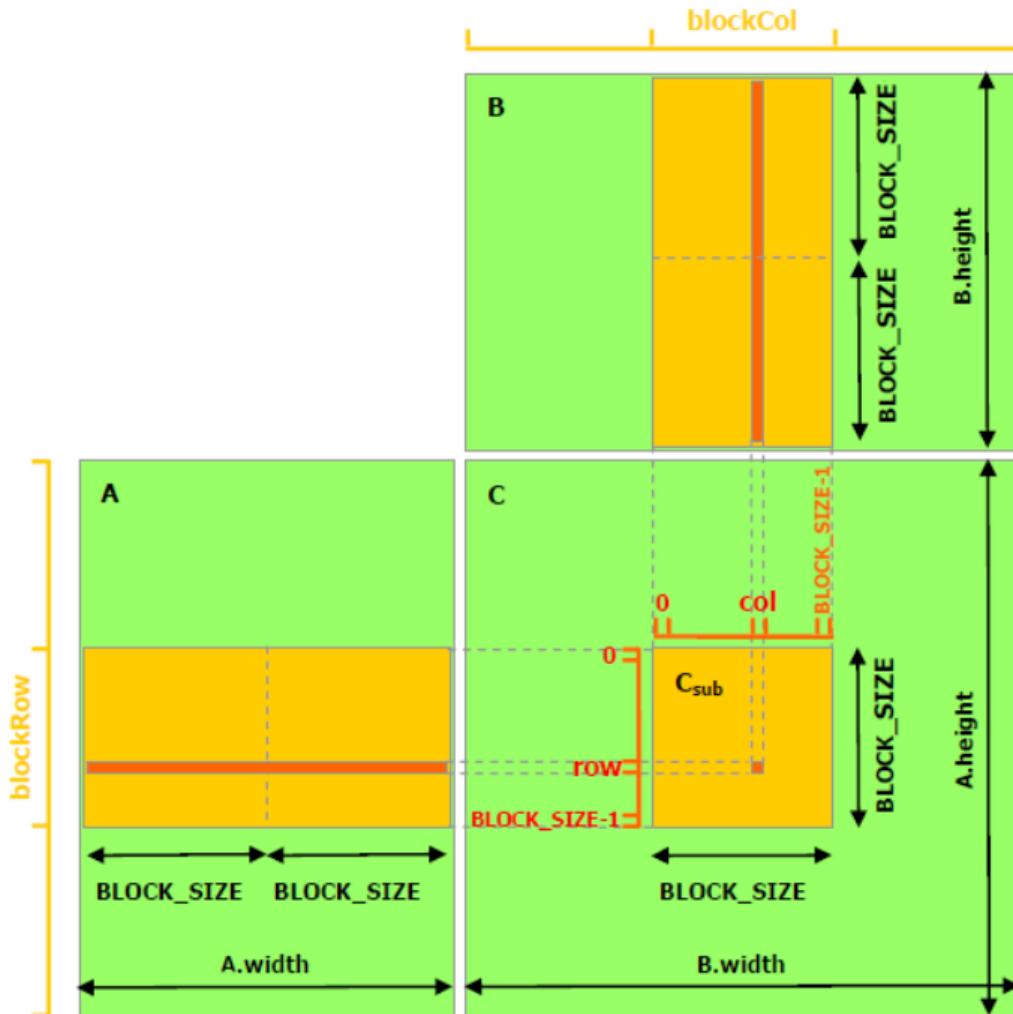
**Recommendation:** Try to make use of shared memory wherever possible.  
Global memory can be 150x slower!



Source: [cuda-programming.blogspot.com](http://cuda-programming.blogspot.com)

# Memory management

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:



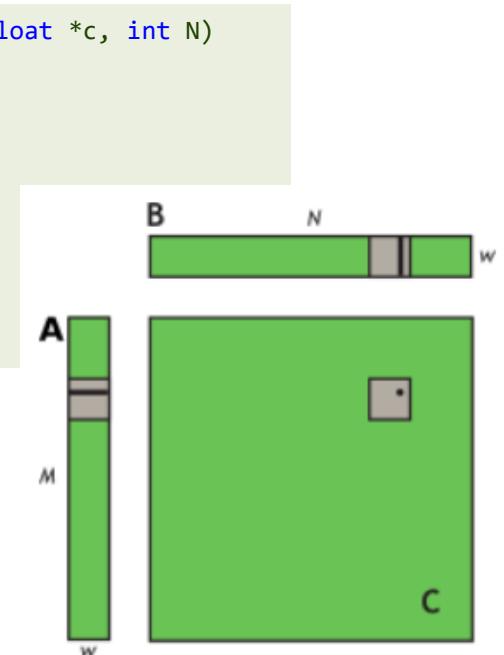
- Each threadblock computes a sub-matrix  $C_{sub}$
- Each thread within the block computes one element of  $C_{sub}$

$BLOCK\_SIZE = 16$  (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;
}
```

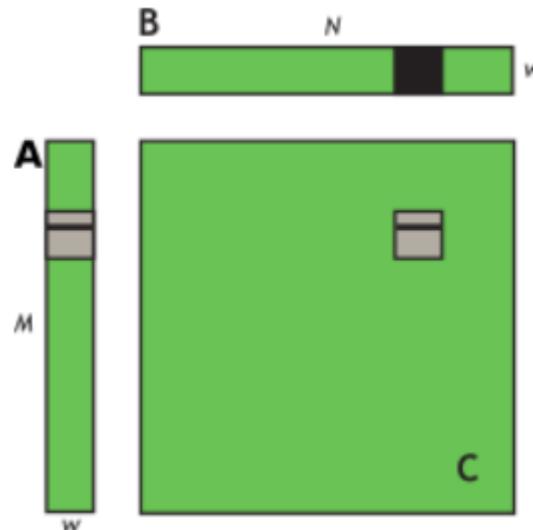
$blockDim.x$ ,  $blockDim.y$ , and  $TILE\_DIM = w$

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



# Memory management

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



Each row of  $C_{\text{sub}}$  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.

# Memory management

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.

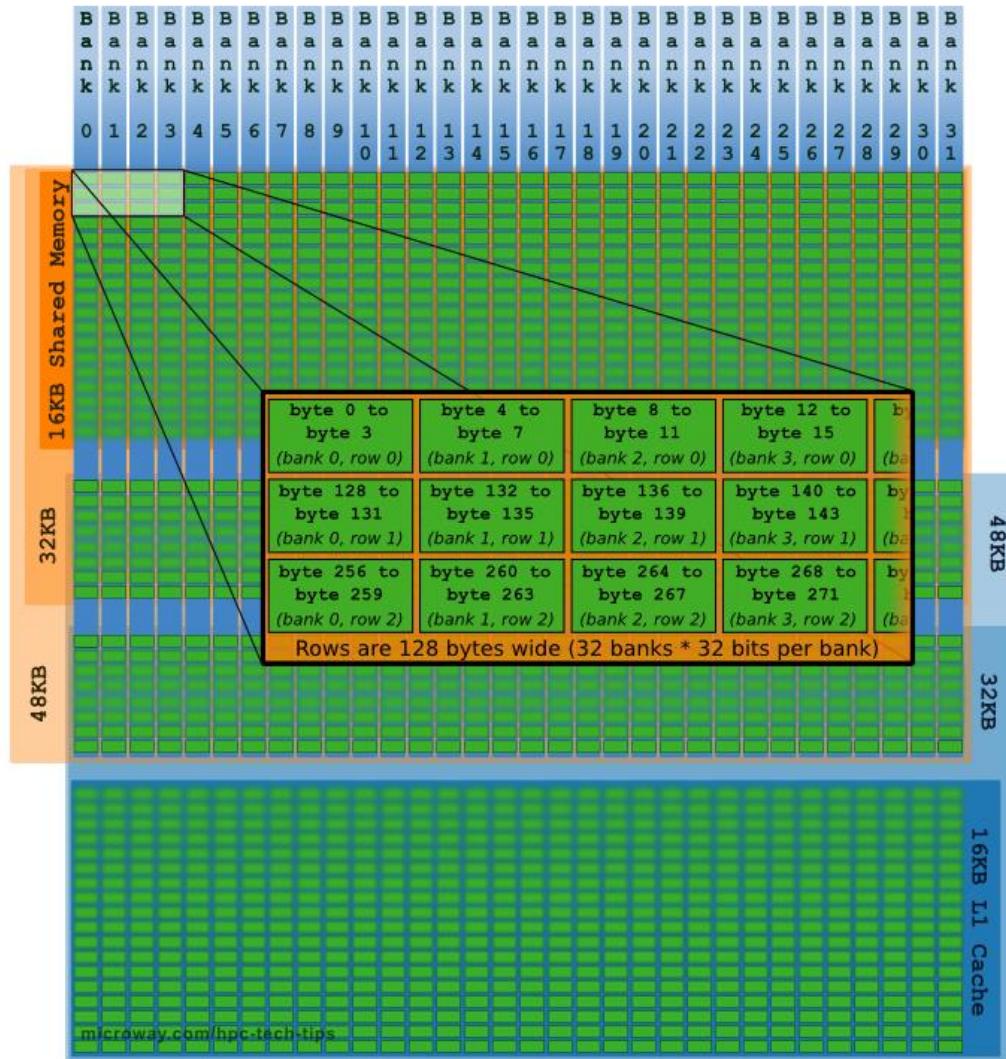
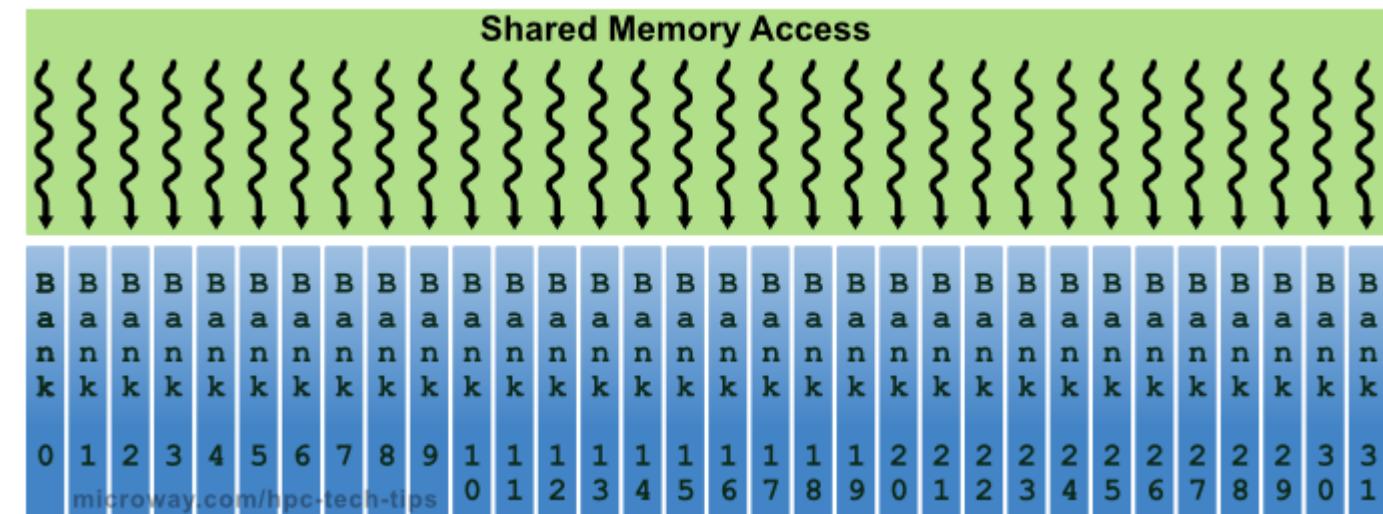


Figure 1: Shared Memory and L1 Cache

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



Source: microway.com

# Memory management

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.

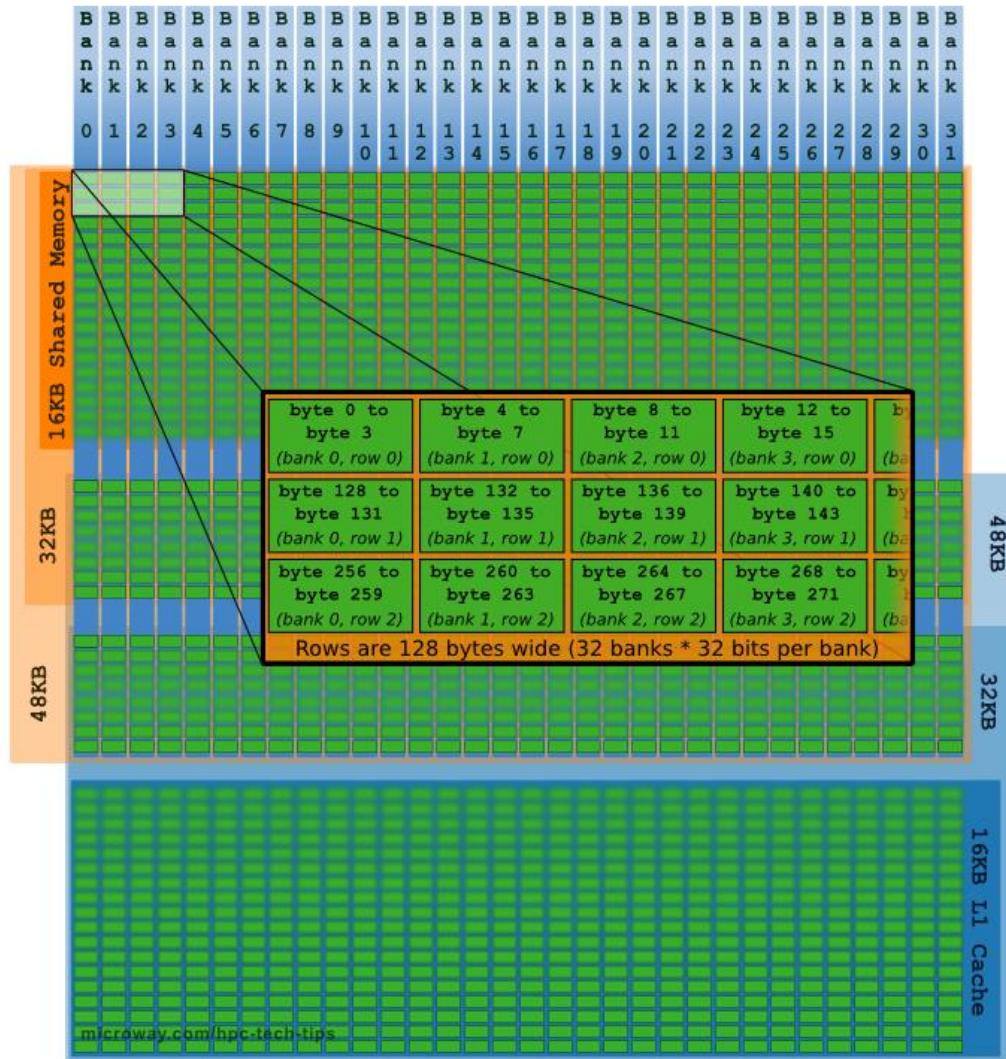
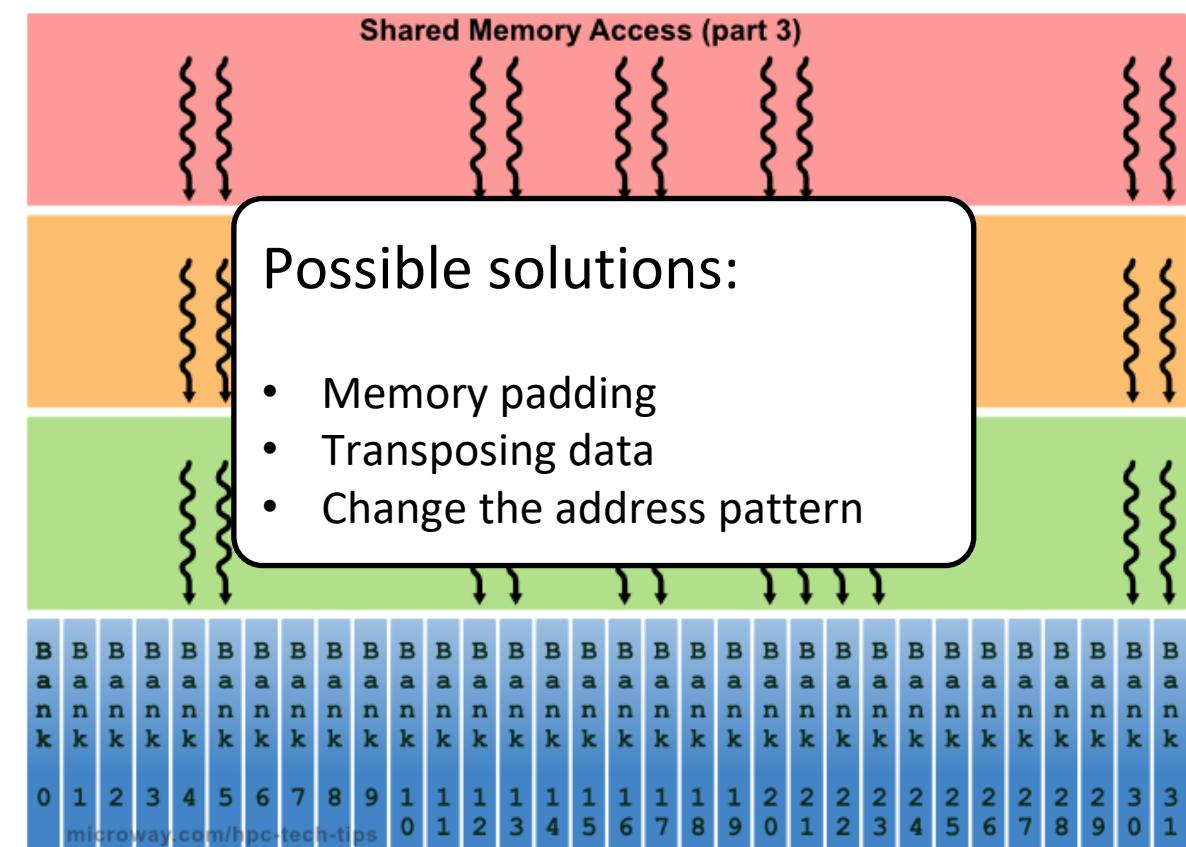


Figure 1: Shared Memory and L1 Cache

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

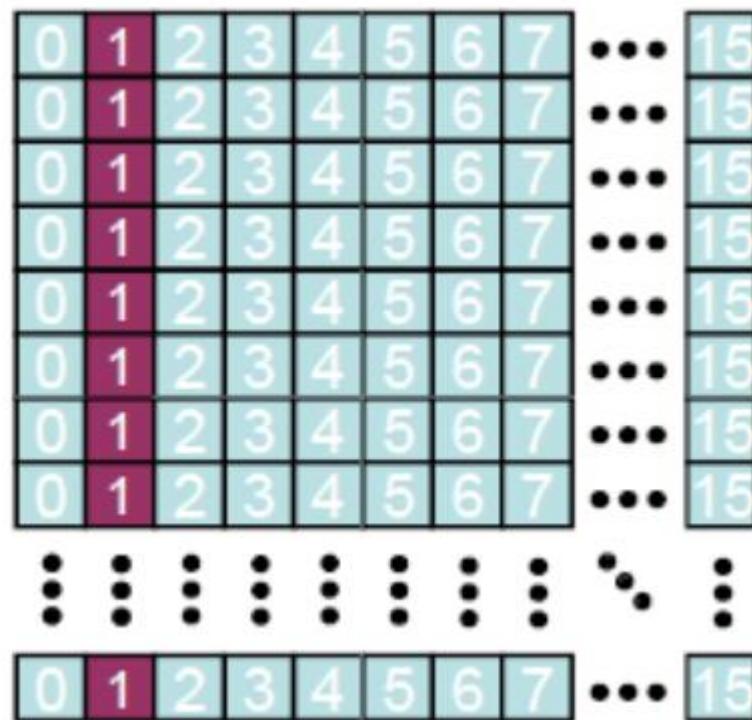


Source: microway.com

# Memory management

Example of bank conflict:

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

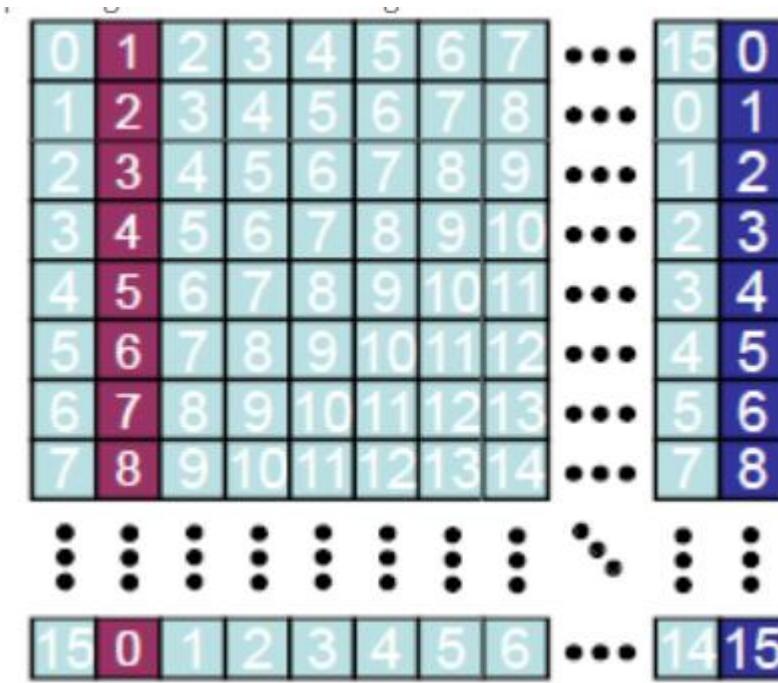


The number is the bank in which data is stored.

All threads move to the right (like the purple cells) having a 16-way bank conflict.

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];`



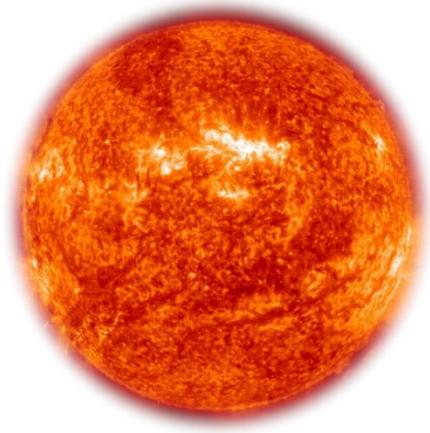
Source: [cuda-programming.blogspot.com](http://cuda-programming.blogspot.com)

# Outline

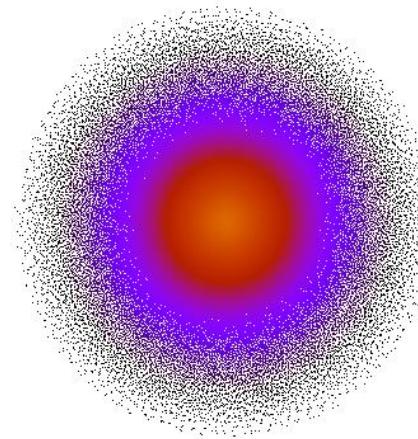
- ❑ Introduction
- ❑ Example
  - ❑ Matrix Multiplication + Exercise
- ❑ Real world implementations
  - ❑ Genomes simulation
  - ❑ Random numbers generation
- ❑ Common pitfalls
- ❑ Memory management
  - ❑ Image processing example
- ❑ Monitoring and asynchronicity
  - ❑ Smoothed Particle Hydrodynamics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



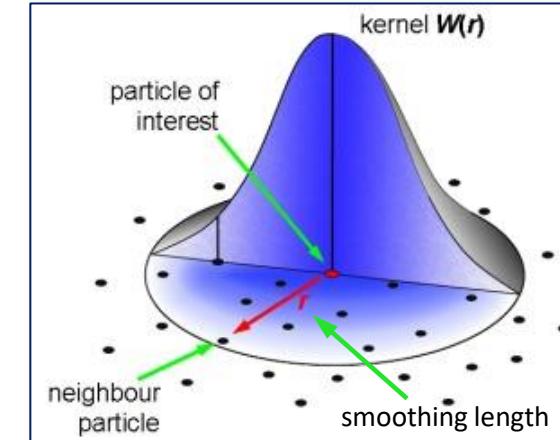
# Smoothed Particle Hydrodynamics



Fluid



SPH particles



$$\langle f(\mathbf{r}) \rangle = \int f(\mathbf{r}') W(\mathbf{r}' - \mathbf{r}) d\mathbf{r}'$$

SPH interpolation kernel

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

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

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

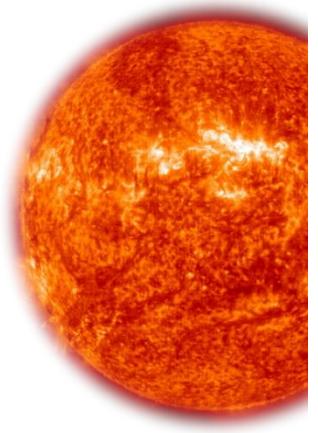
$$f(\mathbf{r}) = \rho$$



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



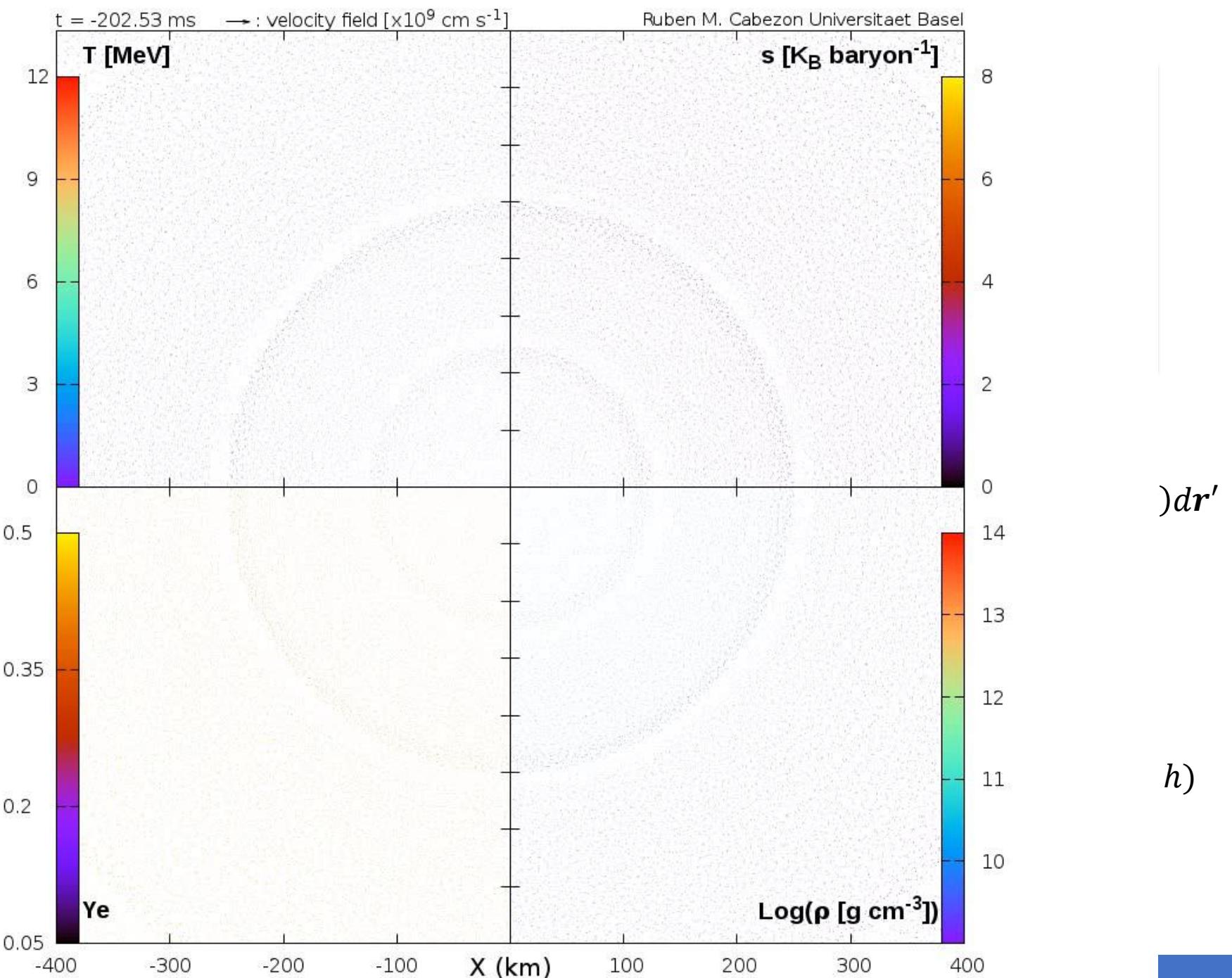
# Smoothed Particle



Fluid

SPH inter

$$W_n^H(v, h) = B_n$$

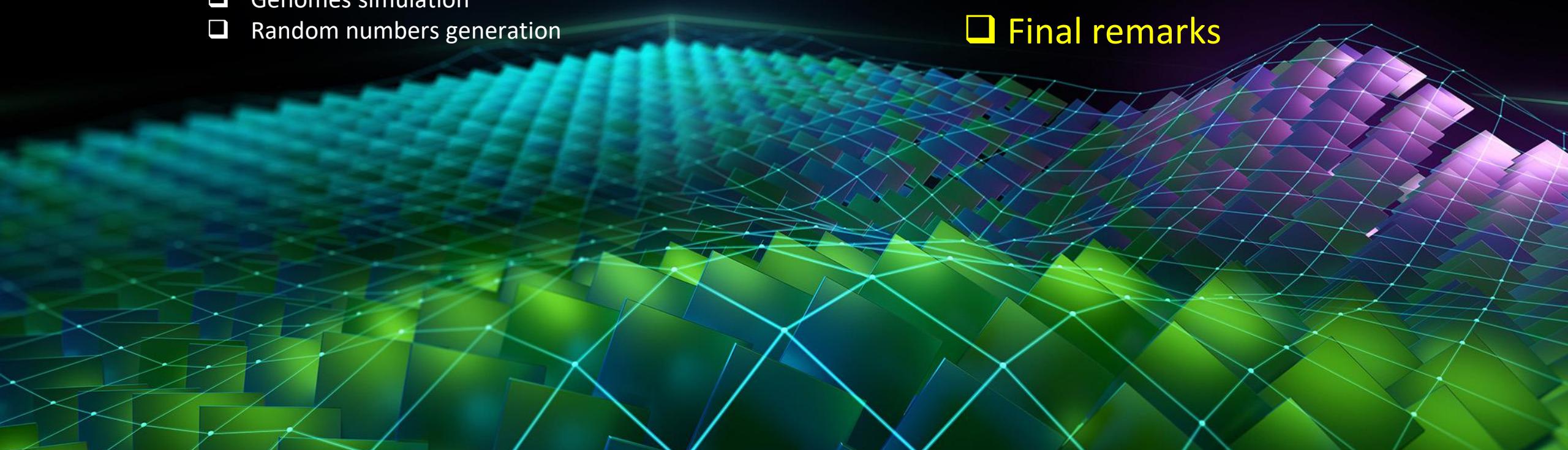


)dr'

h)

# Outline

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



GPU	Nodes	GPUs/node	Slurm Partition	Group
Titanx	sgi[26-29]	6	pascal	All users
Titanx	sgi[21-29]	6	titanx	BIOPZ-A-RG-TM01-Group,stahlberg,basler
RTX3090	sgi[41-42]	4	rtx3090	All users, BIOPZ-A-RG-TM01-Group,maiert,a-maiert-cs2,perez,a-perez-cs2
RTX8000	sgi[51-52]	2	rtx8000	All users
A100	sgi[61-67]	4	a100	All users

```

#!/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 titanx / rtx3090 / ...
#SBATCH --gres=gpu:1            # --gres=gpu:2 for two GPU, ...

module load CUDA
.....

```

You can also use OpenOnDemand to reserve one GPU at sciCORE.

<https://ood.scicore.unibas.ch/>

Useful for developing and debugging, or running short jobs interactively.

Interactive Apps

- Desktops
- sciCORE desktop**
- Servers
- Jupyter Notebook
- RStudio-Server

**sciCORE desktop** version: v1.0.3

This app will launch an interactive desktop on one or more compute nodes. You will have full access to the resources these nodes provide. This is analogous to an interactive batch job.

Instance size

Medium (14 core, 90GB RAM)

**Note** With large and full node instances job allocation wait times might be longer depending on current cluster load

- **Small** (4 core, 26GB RAM)
- **Medium** (14 core, 90GB RAM)
- **Large** (28 core, 180GB RAM)
- **XLarge** (42 core, 270GB RAM)

Runtime (in hours)

GPU

Yes

Partition

a100

Use default account

Yes

Select no only if you need to run the job with a different account

Especial resource reservation granted by sciCORE (optional)

**Launch**

# 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>

## OpenMP:

Offloading support in GCC: <https://gcc.gnu.org/wiki/Offloading>

OpenMP offloading talk: <https://www.youtube.com/watch?v=ypRBx31e8GA>

## 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/>

## CUDA:

nVidia CUDA training : <https://developer.nvidia.com/accelerated-computing-training>

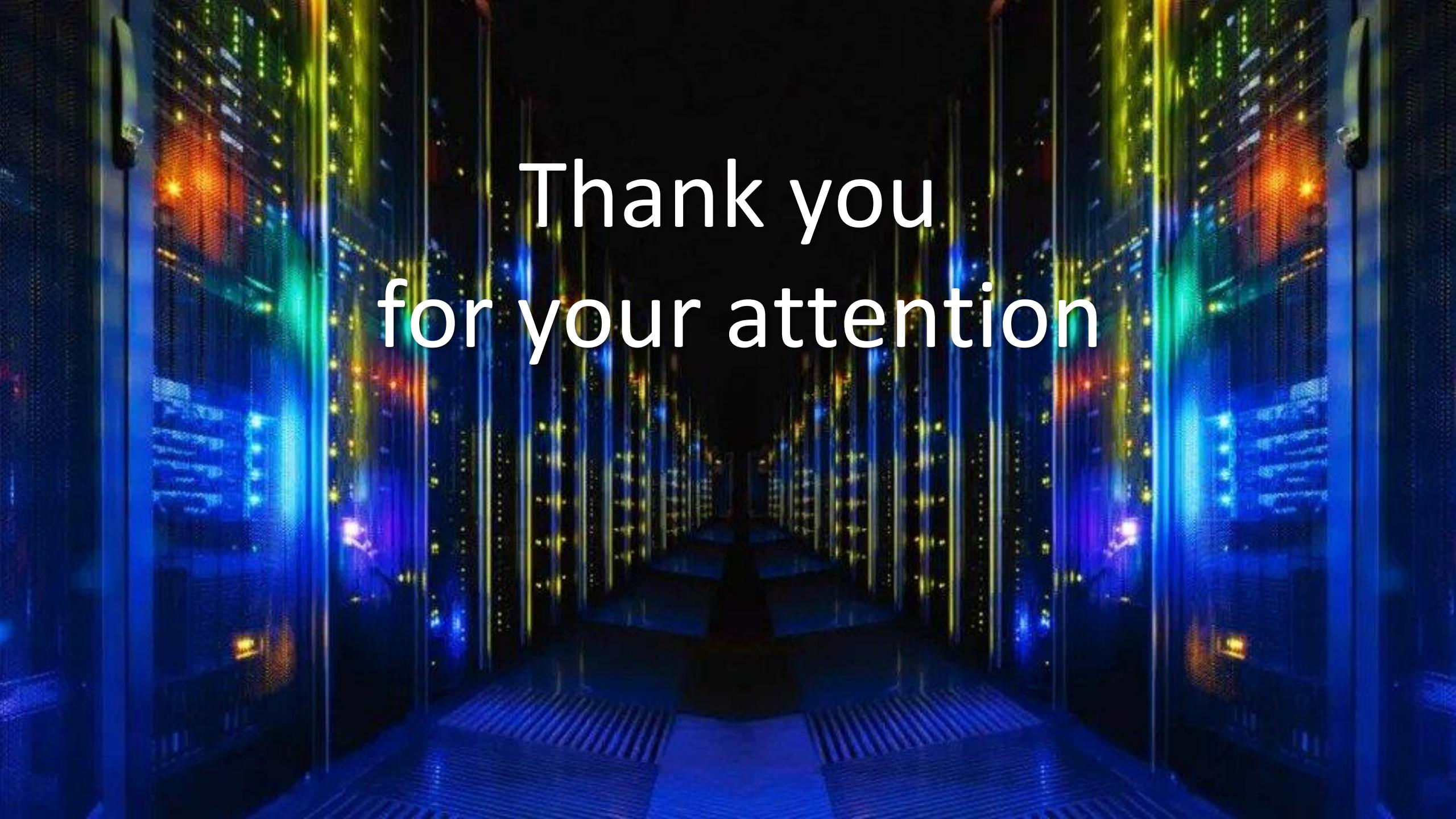
3rd-party CUDA training: <https://developer.nvidia.com/educators/existing-courses>

Numba CUDA: <https://nyu-cds.github.io/python-numba/05-cuda/>

## Numba:

Numba intro: <https://www.youtube.com/watch?v=6oXedk2tGfk>

Numba pitfalls: <https://www.youtube.com/watch?v=x58W9A2lnQc>

A perspective view of a server room filled with rows of server racks, illuminated by colorful lights.

Thank you  
for your attention