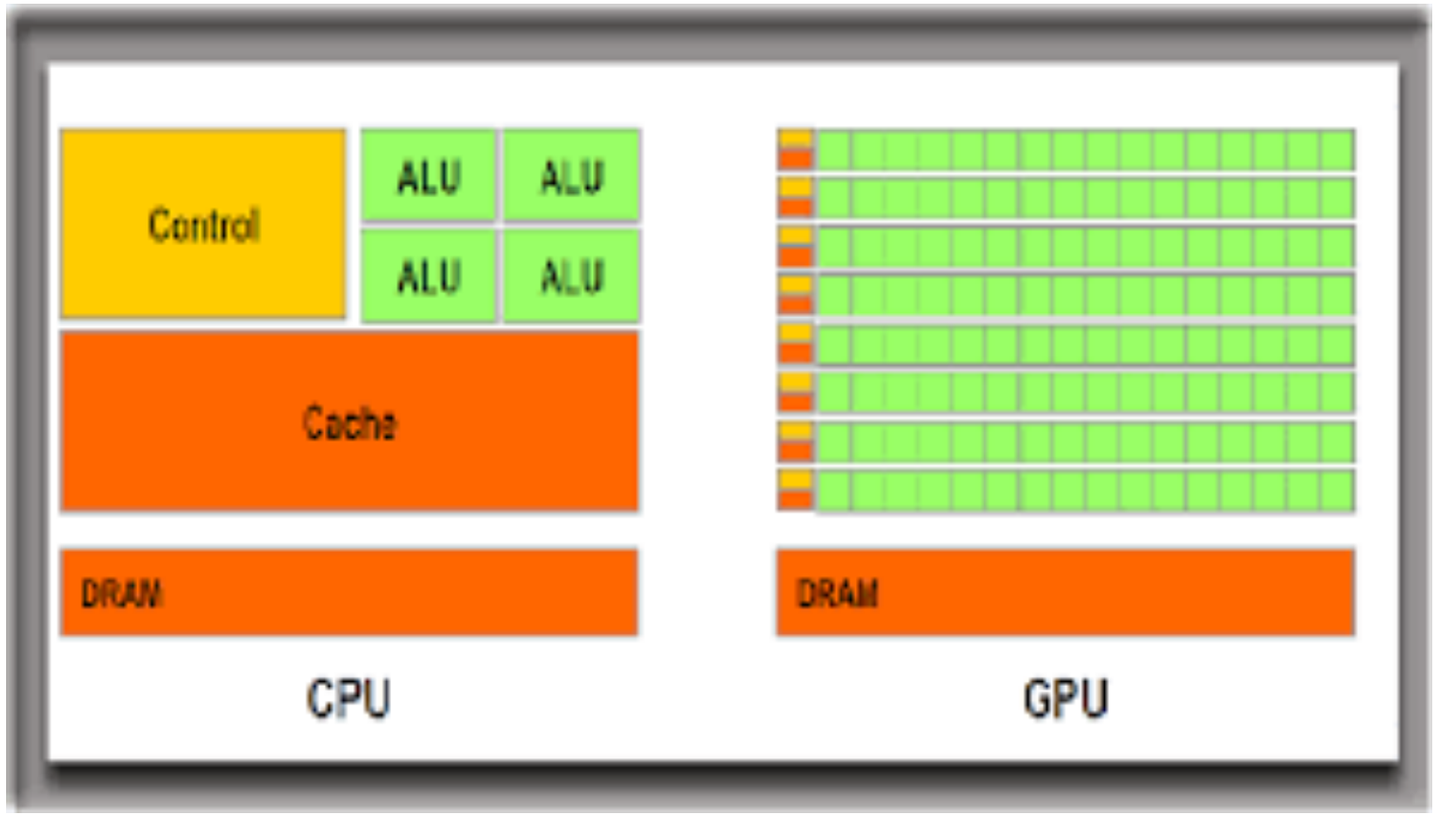
Introduction to Cuda

Olivier Mattelaer
UCLouvain
CP3 & CISM

Program of this lecture

- Difference between CPU and GPU
  - ➡ Why and when to use a GPU?
- What is CUDA?
  - ➡ When/where can I use cuda?
- Structure of a GPU program
  - ➡ Nomenclature
- First example of CUDA programming
- First step in optimisation of a CUDA program
  - ➡ Managing memory transfer

CPU versus GPU



CPU	GPU
Central Processing Unit	Graphics Processing Unit
Several cores	Many cores
Low latency	High throughput
Good for serial processing	Good for parallel processing
Can do a handful of operations at once	Can do thousands of operations at once

Speed versus Latency

- Speed: number of operation per second
- Latency: delay in the first operation

➡ $T = L + vD$

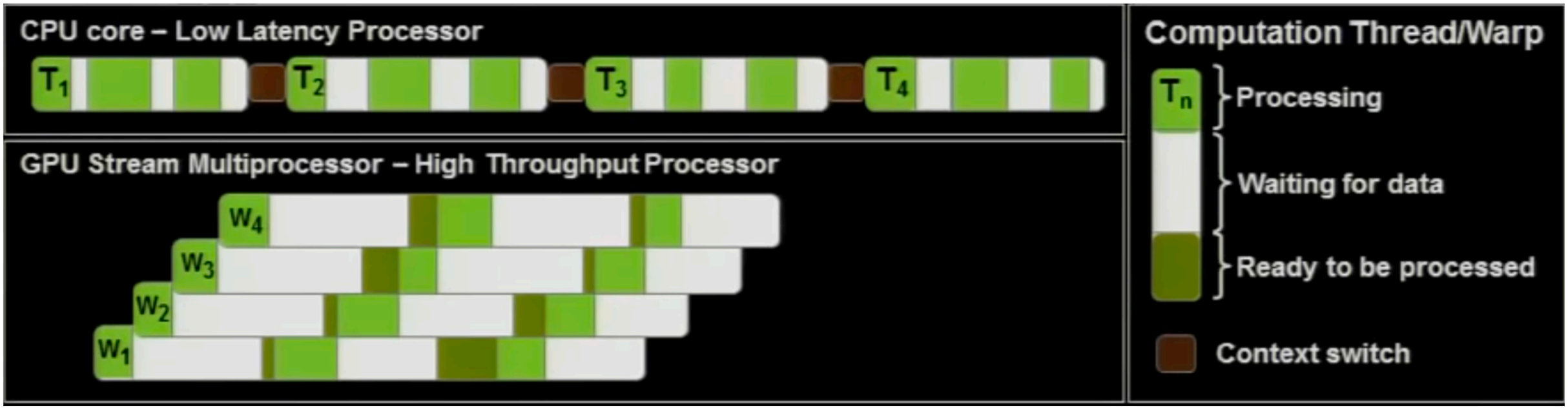
- How amazon transfer data from one cluster to another

- Speed: Large bandwidth
  - Fiber connection: Gb
- Latency: time of the travel between the two cluster.

- Latency is “reactivity”

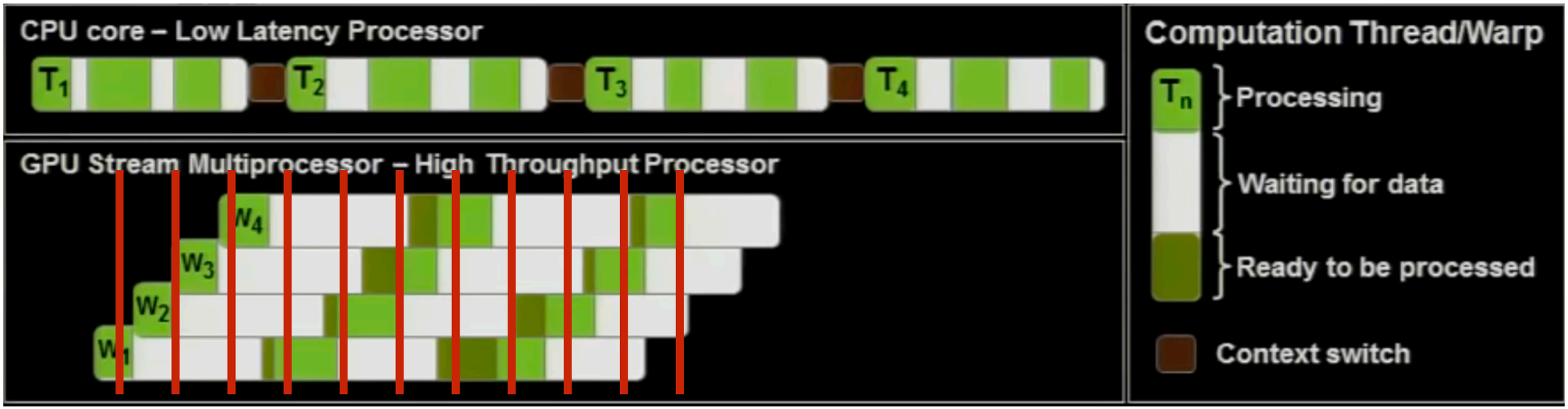
GPU versus CPU

- CPU minimizes latency ➡ $T = L + NvD$
- GPU hides latency by overlapping computation



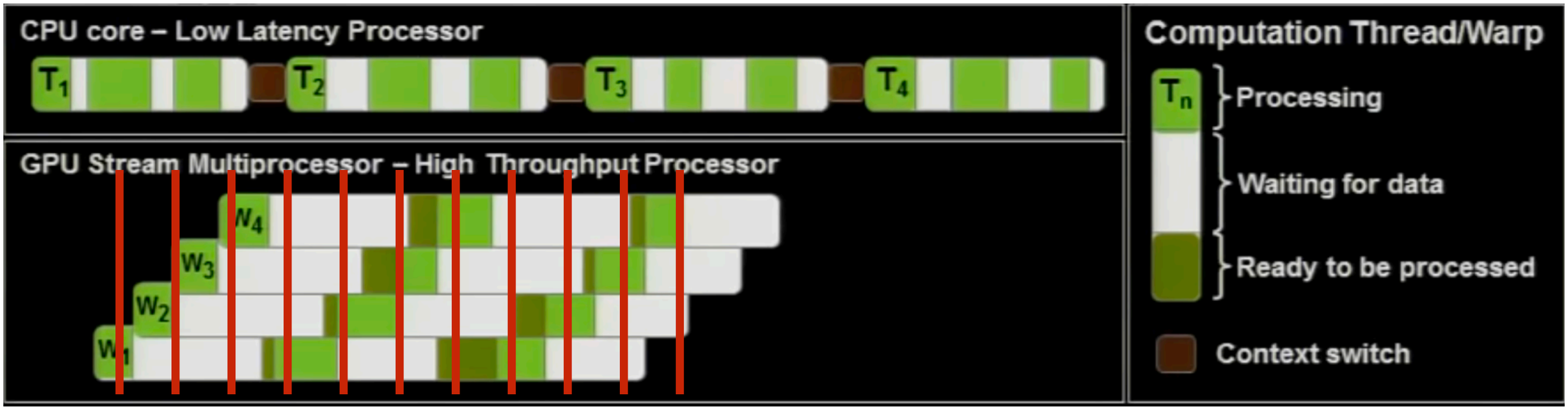
GPU versus CPU

- CPU minimizes latency ➡ $T = L + NvD$
- GPU hides latency by overlapping computation



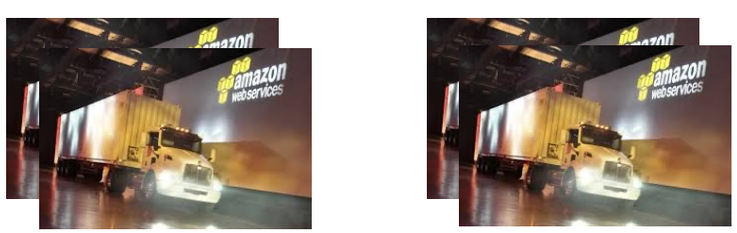
GPU versus CPU

- CPU minimizes latency ➡ $T = L + NvD$
- GPU hides latency by overlapping computation



- Moving data

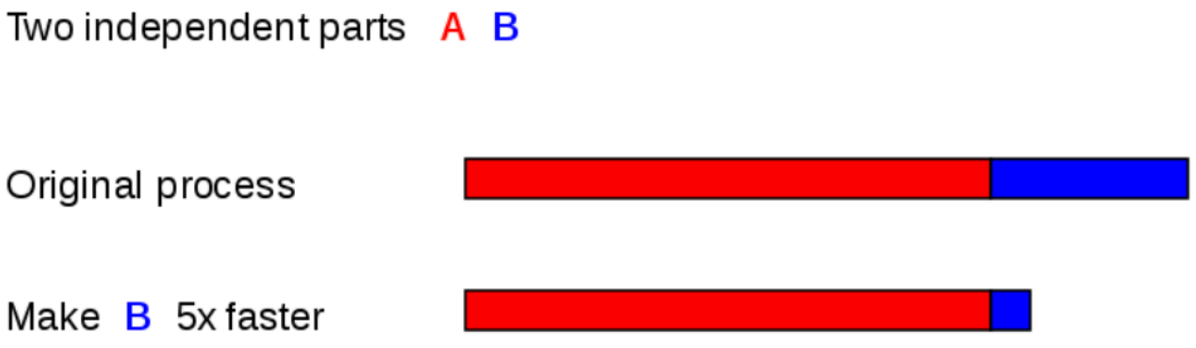
- Transit



- Moving data

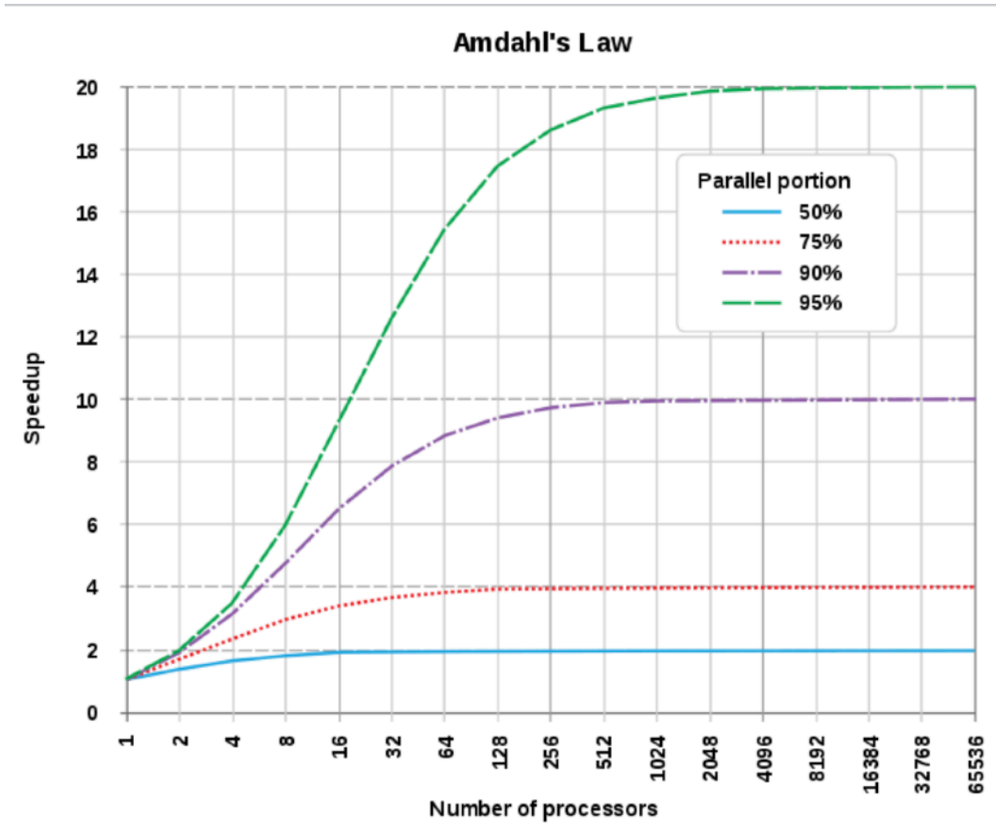
Amdahl's law

- A cpu has 8 core a GPU 2056 core
  - ➡ Should my code should be 200 faster?



- It depends which fraction of your code can use parallelism

- This is Amdahl's law given theoretical speed-up of your code

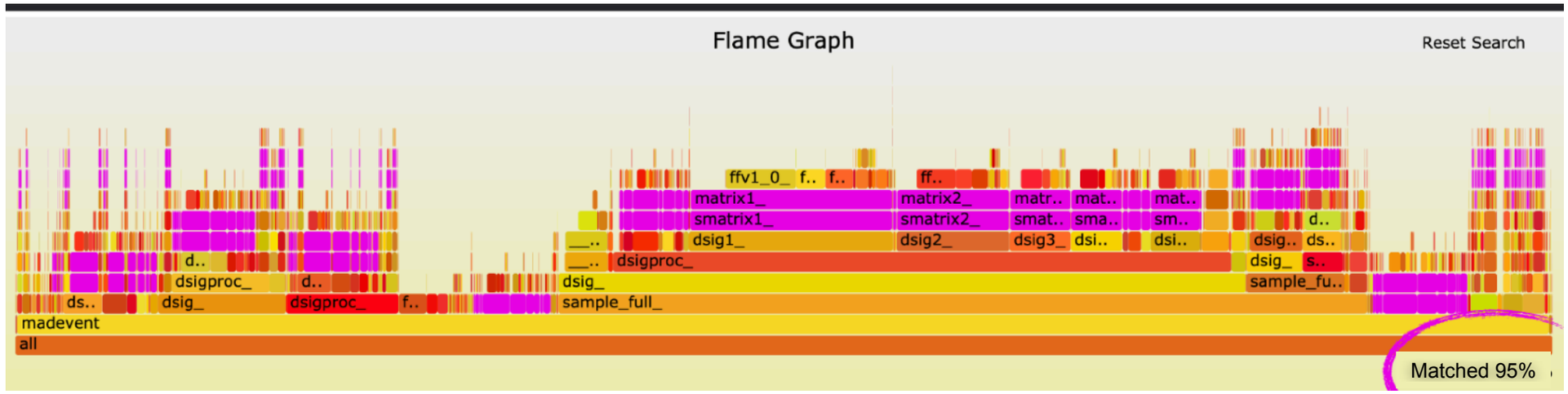


Speed-up in practise

- Comparing speed of code between cpu and gpu are not really fair
  - ➡ Cost of the GPU/CPU
  - ➡ Huge speed-up typically means "bad" denominator
- A "normal" is around 5-20
  - ➡ Much higher number reported in some cases.
- GPU clock is slower than CPU clock
  - ➡ GPU ~ mhz
  - ➡ CPU ~ Ghz

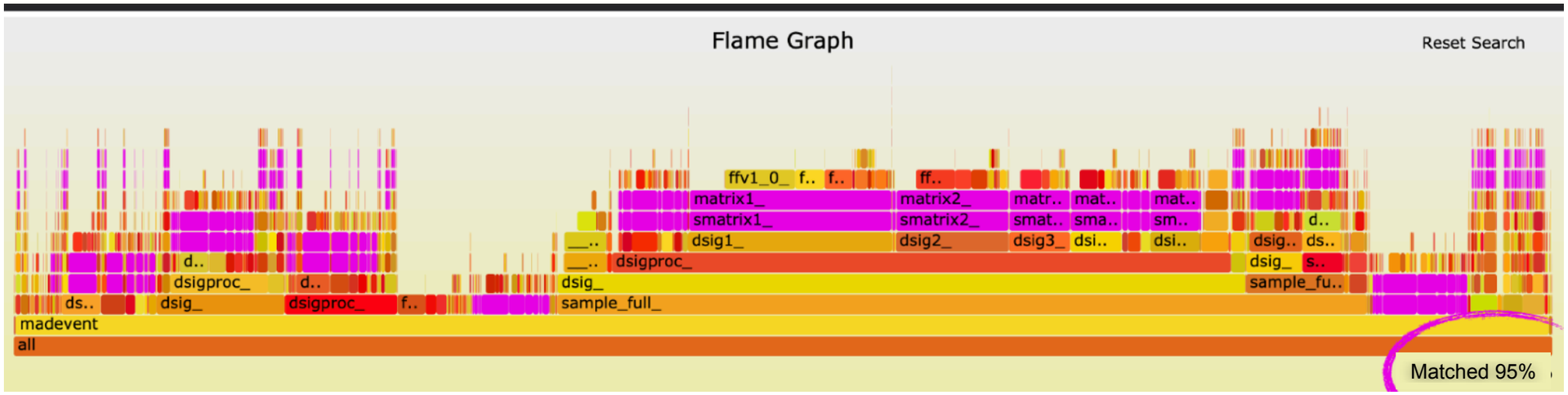
Real case example

- CPU
- 95%

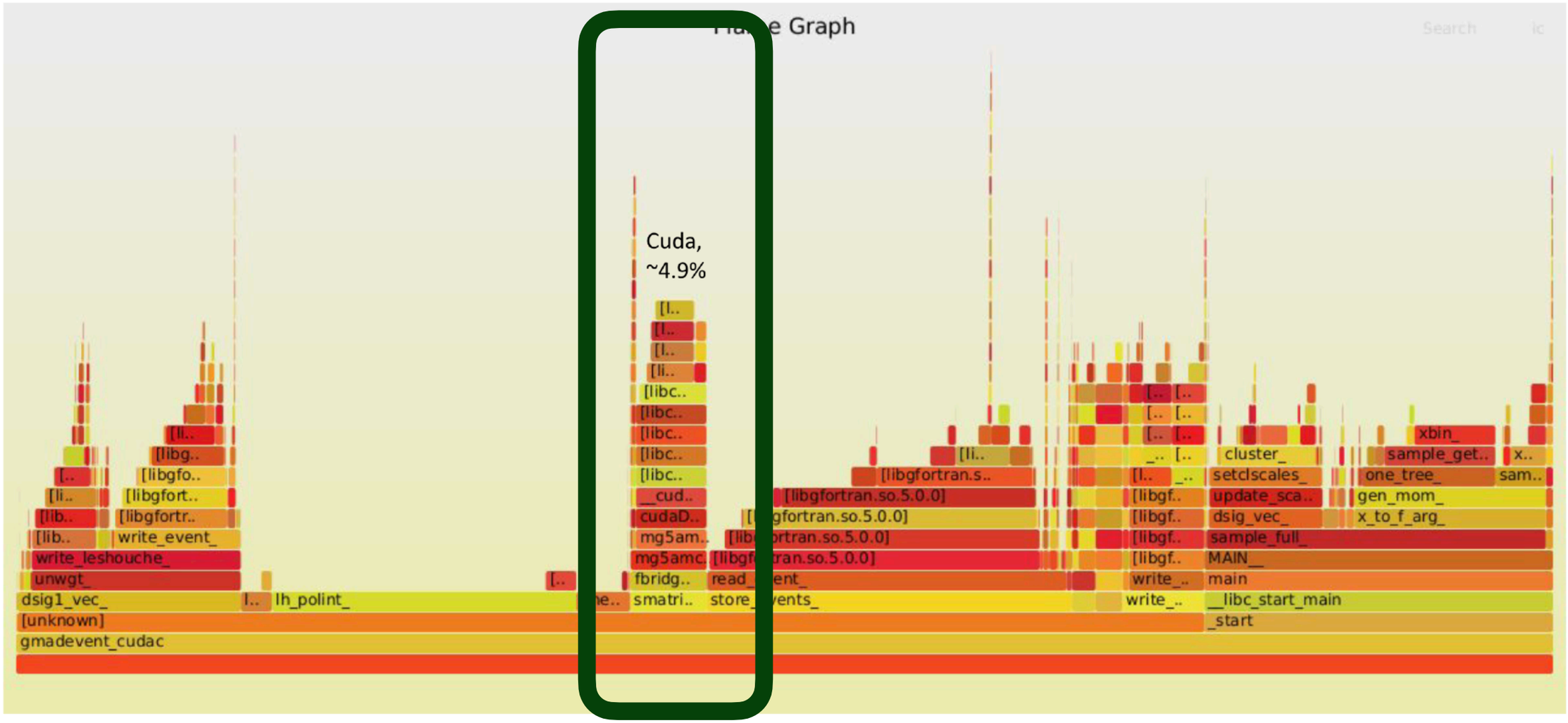


Real case example

- CPU
- 95%

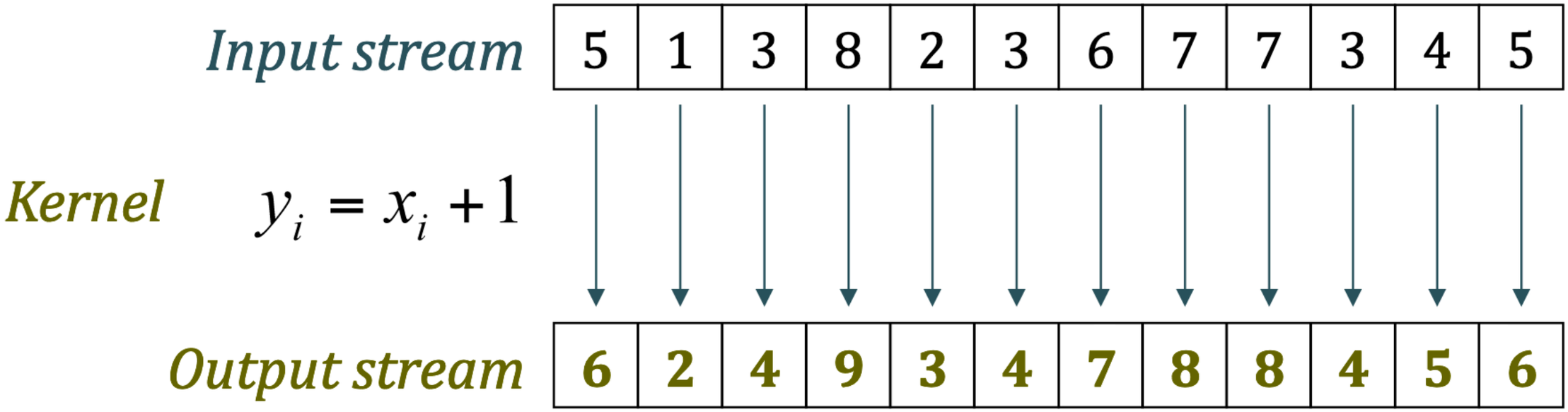


- GPU
- 5%



Stream computing

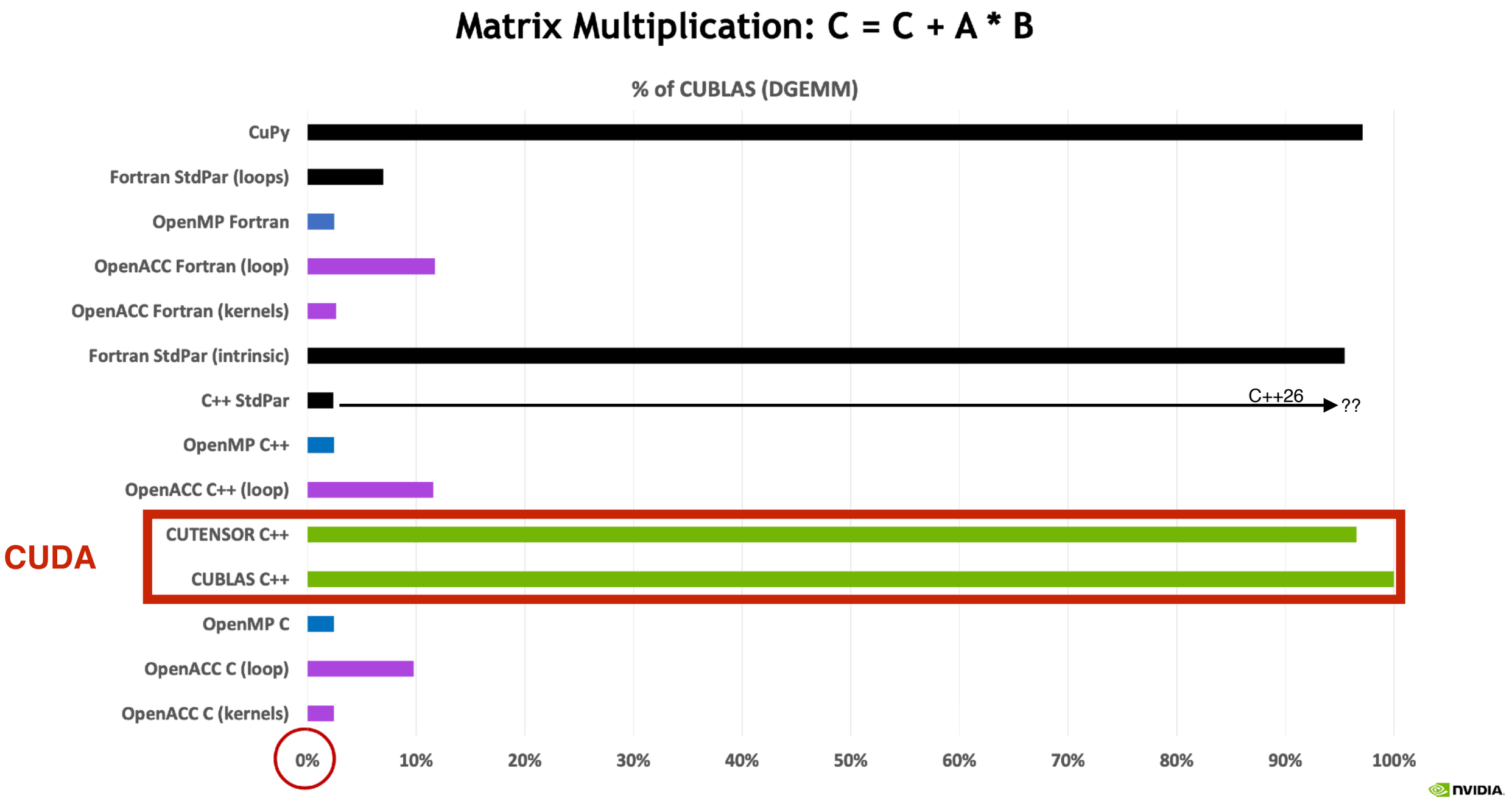
- The idea of GPU are
  - ➡ "multiple data"
  - ➡ SAME operation
- Same as vectorisation on CPU (but different scale)



- You can have synchronisation between threads

- As for CPU, you do not want to code at assembler level
- First released in 2006
  - ➡ Restricted to nvidia GPU
  - ➡ Expose the raw computation power
    - ✦ No need of graphical knowledge

Do you need cuda?



https://youtu.be/nlo_8x54Syo?t=1304

GPU availability

- Dragon 2:
  - ➡ Two machines with two Nvidia V100
- Namur
  - ➡ Gaming GPU
- Manneback (UCL only)
  - ➡ Nvidia V100/ A100
  - ➡ Bunch of gaming GPU
- (Future) Lumi European computer (EUROHPC)
  - ➡ Not Nvidia GPU machine
  - ➡ Cuda code need to be converted to HIP
    - ✦ Alternative: OpenACC, OneAPI, Sycl, kokos,…

SLURM FOR GPU

- Check ressource
  - ➡ sinfo --format="%N %.6D %P %G"

```
mb-bro080      1 cp3-gpu gpu:TeslaK80:2,localscratch:156
mb-cas101      1 gpu gpu:TeslaV100:2,mps:TeslaV100:100,localscratch:172
mb-cas102      1 gpu gpu:TeslaV100:2,mps:TeslaV100:100,localscratch:411
mb-sab040      1 gpu gpu:TeslaM10:4,localscratch:46
```

- First run interactively
  - ➡ srun -p gpu --gres=gpu:TeslaV100:1 --pty bash
- Check module on the machine
  - ➡ module av
- Check that you have access to the GPU
  - ➡ nvidia-smi

SLURM FOR GPU

- Check ressource
  - ➡ sinfo --format="%N %.6D %P %G"

```
NODELIST  NODES PARTITION GRES
drg2-w[001-017]     17 batch* (null)
drg2-w[001-017]     17 long (null)
drg2-w[018-019]      2 gpu gpu:2
drg2-w[001-017]     17 debug (null)
```

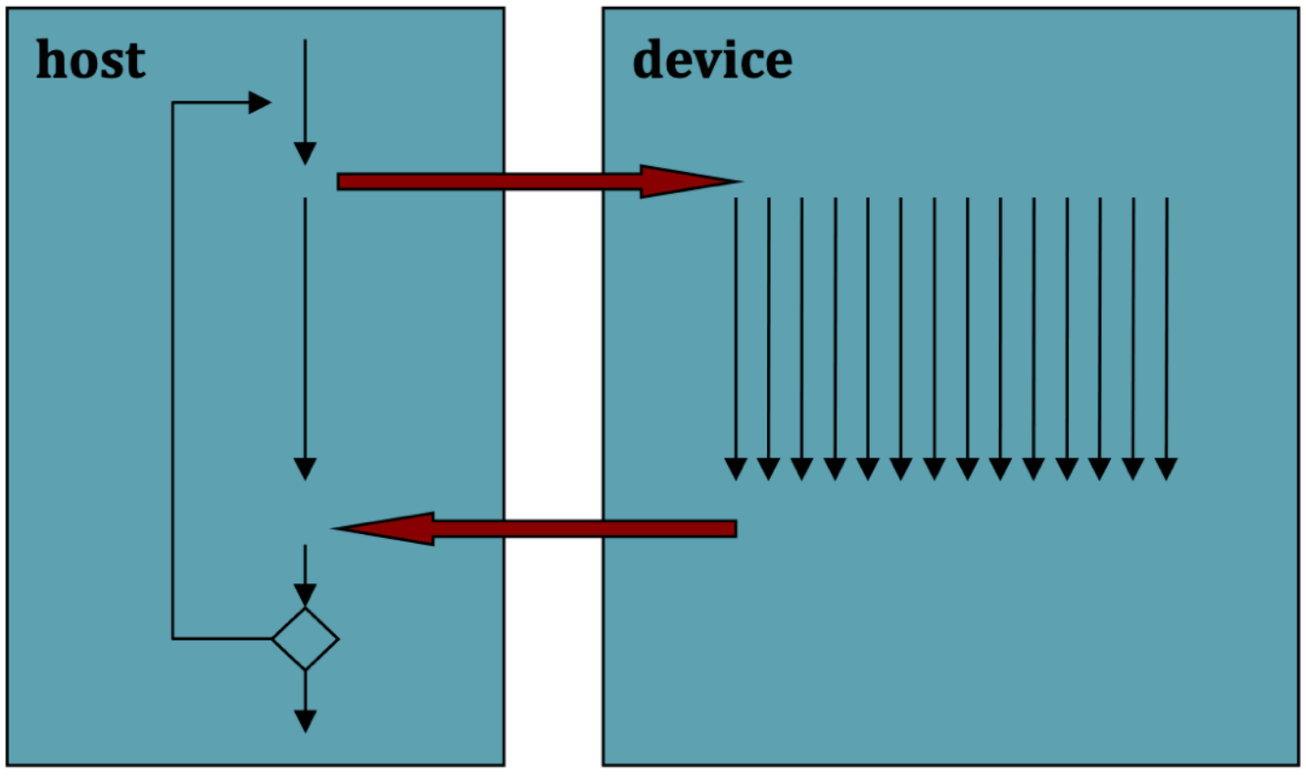
- First run interactively
  - ➡ srun -p gpu --gres=gpu:1 --pty bash
- Check module on the machine
  - ➡ module av
- Check that you have access to the GPU
  - ➡ nvidia-smi

Virtual GPU : MIG (A100)

- If you say "A100 is too big for me to fill it. I prefer to have multiple small GPU"
- Then MIG is for you.
  - ➡ MIG split your A100 in multiple virtual smaller GPU that are seen as different independent GPU
  - ➡ Many splitting are technically possible but we offer only one (splitting in 7)
  - ➡ Add in slurm "--comment=MIG"
- Only for A100 GPU (so not dragon2)

Cuda Programming model

- A GPU needs to be controlled by a CPU.
  - ➡ All programs start by the CPU
  - ➡ Data are prepared on the CPU and moved to the GPU
  - ➡ GPU is crunching data
  - ➡ Data moved back to the cpu
  - ➡ Programs end



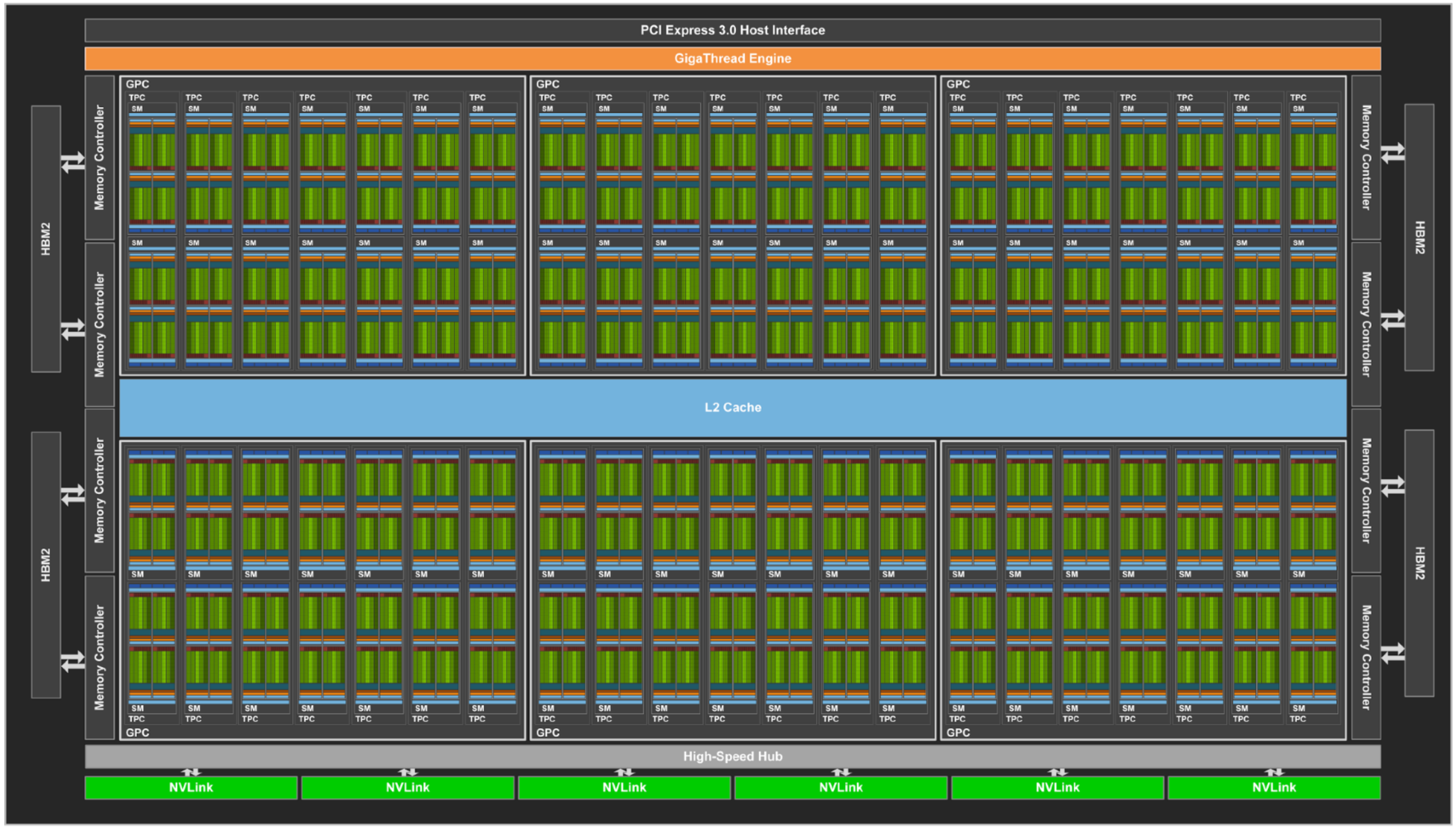
Kernel execution is asynchronous

Asynchronous memory transfers also available

Cuda Programming model

- The cpu is called the **“host”**
- The gpu is called the **“device”**
  - ➡ Viewed as a co-processor
- Function executed on gpu are called **kernel**
  - ➡ Executed in parallel on different data element
- Both the host/device have their own memory
  - ➡ Memory management is handle by the host
  - ➡ Automatic management is possible

Multi-processor/block/thread

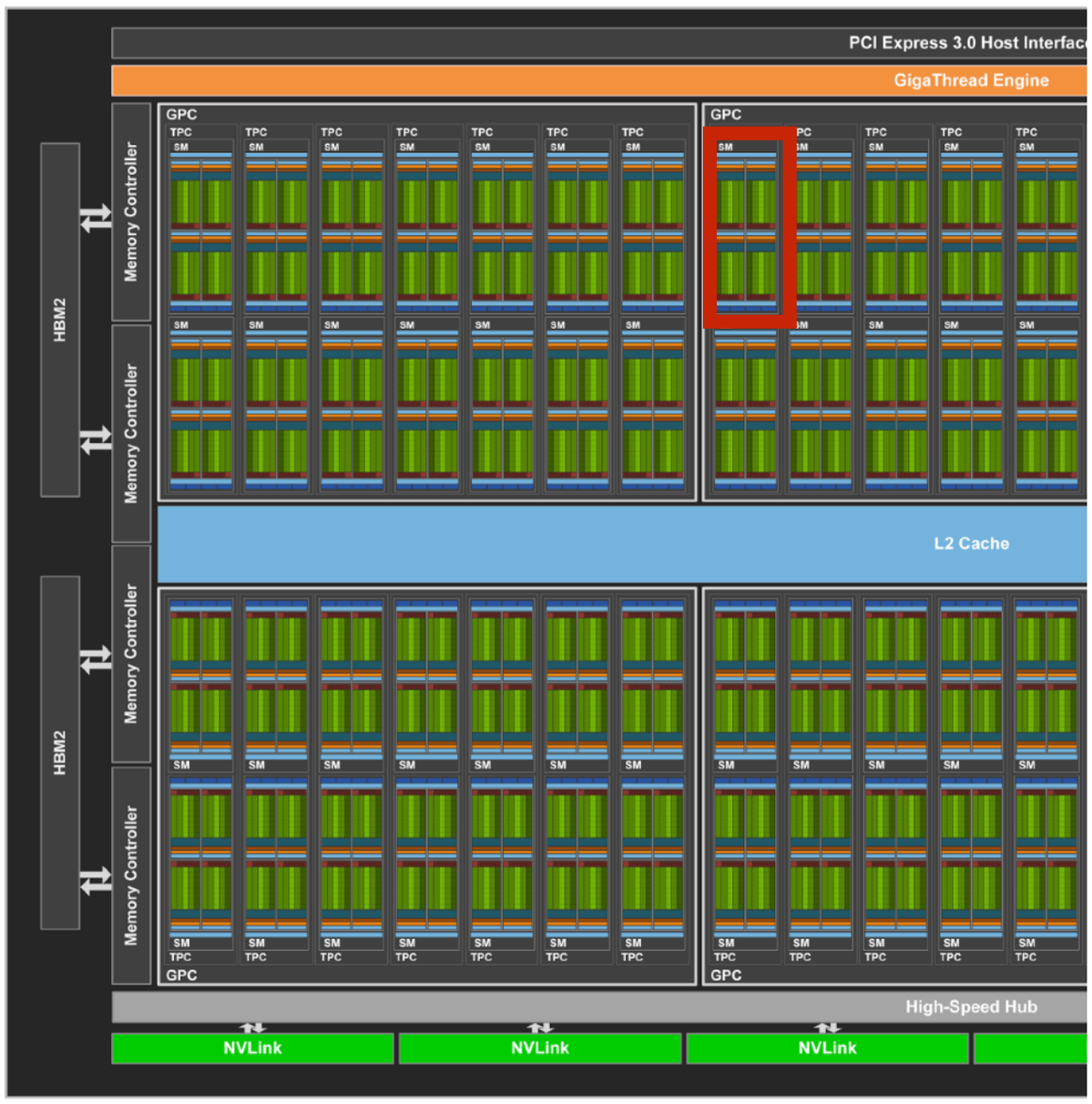


- Main component
  - ➡ Memory
  - ➡ Streaming Multiprocessor (84 of them here)

Multi-processor/block/thread

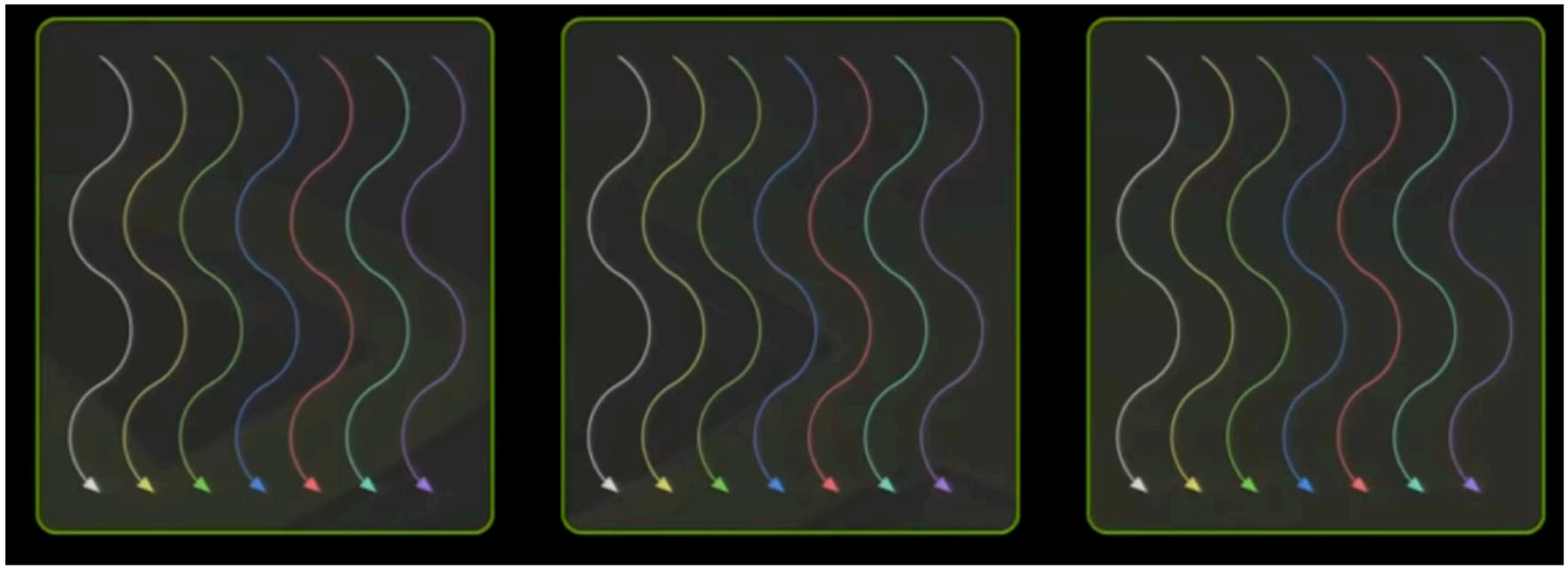
- Main component
  - ➡ Memory
  - ➡ Streaming Multiprocessor (84 of them here)

Multi-processor/block/thread



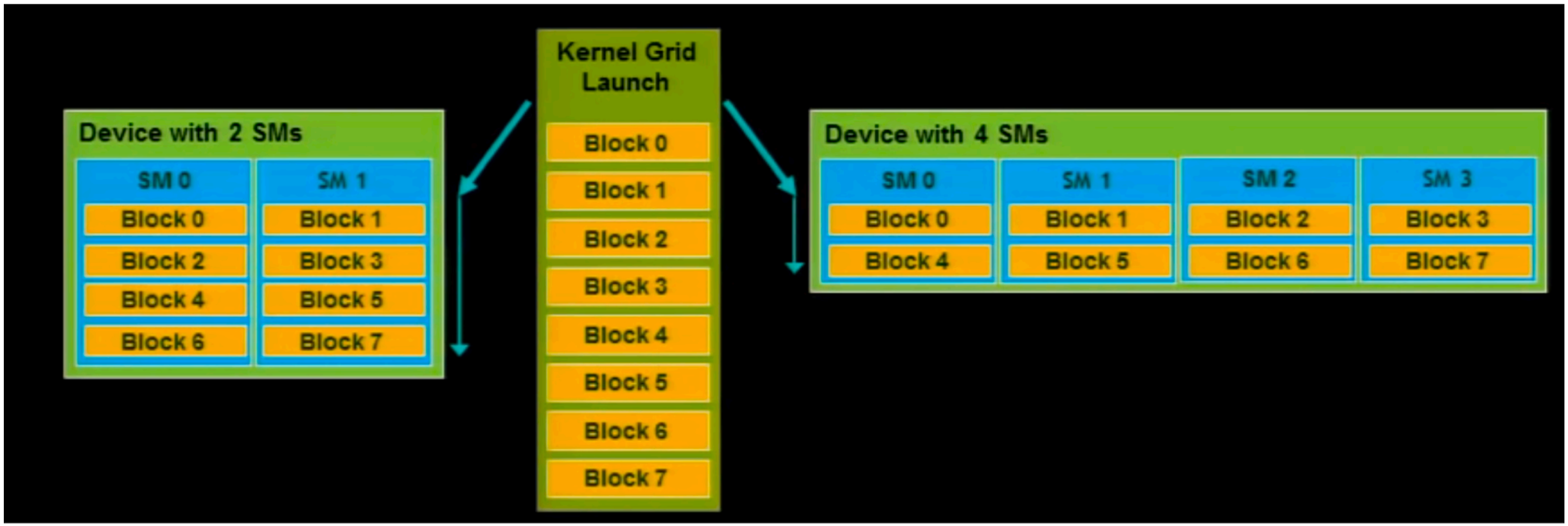
- Main component
  - ➡ Memory
  - ➡ Streaming Multiprocessor (84 of them here)

Block



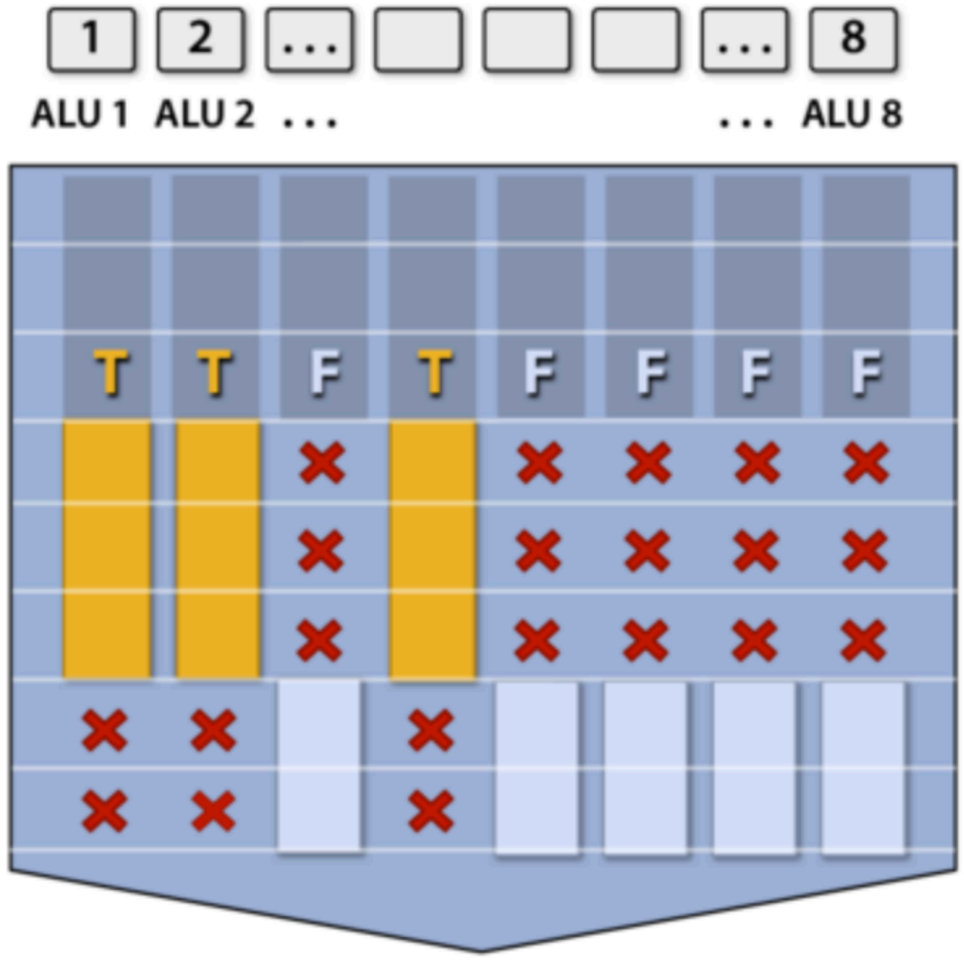
- Thread are grouped by block
  - ➡ Collaboration of thread (syncronization, shared memory)
- Up to 2048 thread per block
- Block are fully independent
  - ➡ Can be executed in any order
  - ➡ Can be executed on different GPU

- Separation into block allow you to adapt to various GPU in an easy way.



Wrap

- block are organised in wrap of 32 thread
  - ➡ Correspond to an hardware configuration
- Those 32 threads are working in lock step
  - ➡ Run the same command at the same time
  - ➡ If statement slows down the code



(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

```
<unconditional code>

float x = A[i];

if (x > 0) {
   float tmp = exp(x,5.f);
   tmp *= kMyConst1;
   x = tmp + kMyConst2;
} else {
   float tmp = kMyConst1;
   x = 2.f * tmp;
}

<resume unconditional code>
```

Let's port this function to GPU

```cpp
#include <stdio.h>
#include <stdlib.h>
#include <algorithm>
#include <cmath>

void saxpy_cpu(float* vecX, float* vecY, float alpha, int n)
{
  for (int i=0; i < n; i++){
    vecY[i] = alpha * vecX[i] + vecY[i];
  }
}

int main(){

  int N = 1<<20; // 2^20 = 1,048,576
  float* x;
  float* y;
  x = (float *) malloc(N*sizeof(float));
  y = (float *) malloc(N*sizeof(float));
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }
  saxpy_cpu(x, y, 2.f, N);

  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, std::abs(y[i]-4.0f));
  printf("Max error: %f\n", maxError);


}
```

Let's port this function to GPU

```cpp
#include <stdio.h>
#include <stdlib.h>
#include <algorithm>
#include <cmath>

void saxpy_cpu(float* vecX, float* vecY, float alpha, int n)
{
  for (int i=0; i < n; i++){
    vecY[i] = alpha * vecX[i] + vecY[i];
  }
}

int main(){

  int N = 1<<20; // 2^20 = 1,048,576
  float* x;
  float* y;
  x = (float *) malloc(N*sizeof(float));
  y = (float *) malloc(N*sizeof(float));
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }
  saxpy_cpu(x, y, 2.f, N);

  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, std::abs(y[i]-4.0f));
  printf("Max error: %f\n", maxError);


}
```

- $\vec{y} = a\vec{x} + \vec{y}$

Let's port this function to GPU

```cpp
#include <stdio.h>
#include <stdlib.h>
#include <algorithm>
#include <cmath>

void saxpy_cpu(float* vecX, float* vecY, float alpha, int n)
{
  for (int i=0; i < n; i++){
    vecY[i] = alpha * vecX[i] + vecY[i];
  }
}

int main(){

  int N = 1<<20; // 2^20 = 1,048,576
  float* x;
  float* y;
  x = (float *) malloc(N*sizeof(float));
  y = (float *) malloc(N*sizeof(float));
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }
  saxpy_cpu(x, y, 2.f, N);

  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, std::abs(y[i]-4.0f));
  printf("Max error: %f\n", maxError);


}
```

- $\vec{y} = a\vec{x} + \vec{y}$
- float* is used here for passing an array

Let's port this function to GPU

```cpp
#include <stdio.h>
#include <stdlib.h>
#include <algorithm>
#include <cmath>

void saxpy_cpu(float* vecX, float* vecY, float alpha, int n)
{
  for (int i=0; i < n; i++){
    vecY[i] = alpha * vecX[i] + vecY[i];
  }
}

int main(){

  int N = 1<<20; // 2^20 = 1,048,576
  float* x;
  float* y;
  x = (float *) malloc(N*sizeof(float));
  y = (float *) malloc(N*sizeof(float));
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }
  saxpy_cpu(x, y, 2.f, N);

  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, std::abs(y[i]-4.0f));
  printf("Max error: %f\n", maxError);


}
```

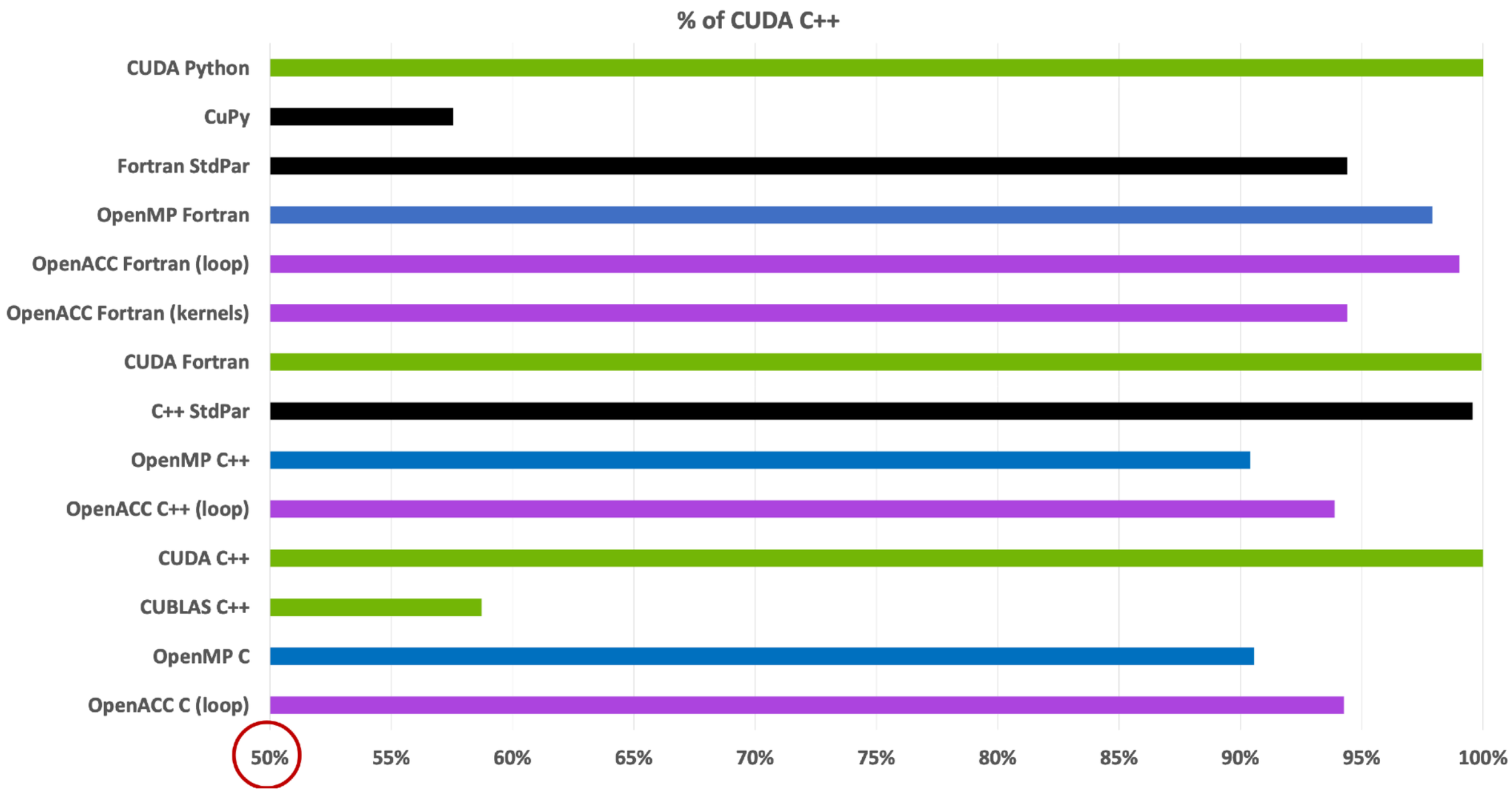
- $\vec{y} = a\vec{x} + \vec{y}$
- float* is used here for passing an array
- that array is assigned dynamically (malloc)

Let's port this function to GPU

```cpp
#include <stdio.h>
#include <stdlib.h>
#include <algorithm>
#include <cmath>

void saxpy_cpu(float* vecX, float* vecY, float alpha, int n)
{
  for (int i=0; i < n; i++){
    vecY[i] = alpha * vecX[i] + vecY[i];
  }
}

int main(){

  int N = 1<<20; // 2^20 = 1,048,576
  float* x;
  float* y;
  x = (float *) malloc(N*sizeof(float));
  y = (float *) malloc(N*sizeof(float));
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }
  saxpy_cpu(x, y, 2.f, N);

  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, std::abs(y[i]-4.0f));
  printf("Max error: %f\n", maxError);


}
```

- $\vec{y} = a\vec{x} + \vec{y}$
- float* is used here for passing an array
- that array is assigned dynamically (malloc)
- We explicitly loop over the data element

Vector Addition: Z = a * X + Y



The limitation here is the **time to move data**, all parallelism language are efficient for this.

-> Fast code can be achieved with StdPar

https://youtu.be/nIo_8x54Syo

Cuda version: kernel

GPU

```
__global__ void saxpy(float *d_VecX, float *d_VecY, float alpha, int n)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) d_VecY[i] = alpha*d_VecX[i] + d_VecY[i];
}
```

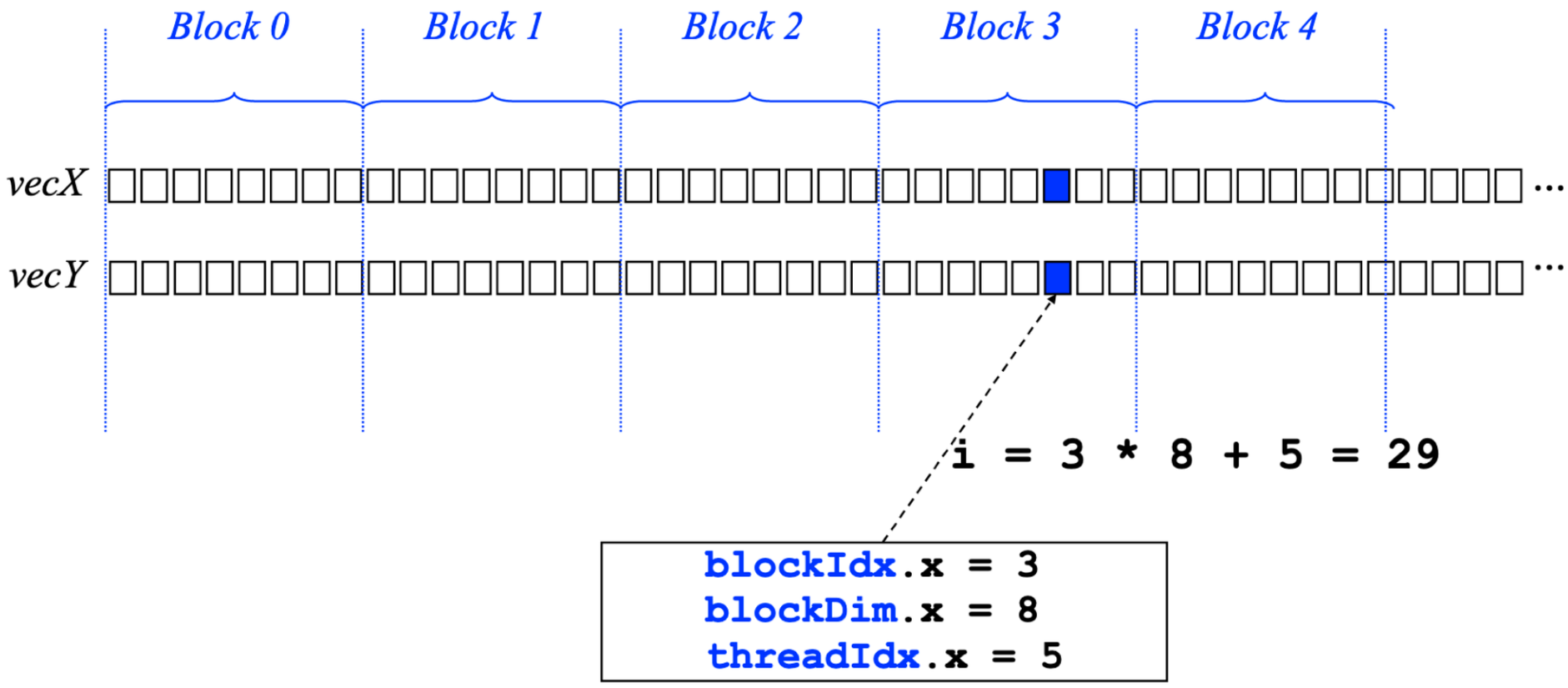
CPU

```
void saxpy_cpu(float* vecX, float* vecY, float alpha, int n)
{
  for (int i=0; i < n; i++){
    vecY[i] = alpha * vecX[i] + vecY[i];
  }
}
```

- No loop anymore !!
  - ➡ Each thread will take care of one data
  - ➡ Need to compute which element each thread has to handle.
  - ➡ Various variable defined for that
    - ✦ blockIdx.x (.y/ .z if 2D and 3D): id of the current block
    - ✦ blockDim.x: number thread in Block (for that dimension)
    - ✦ threadIdx.x: id of the current thread inside the block

Index

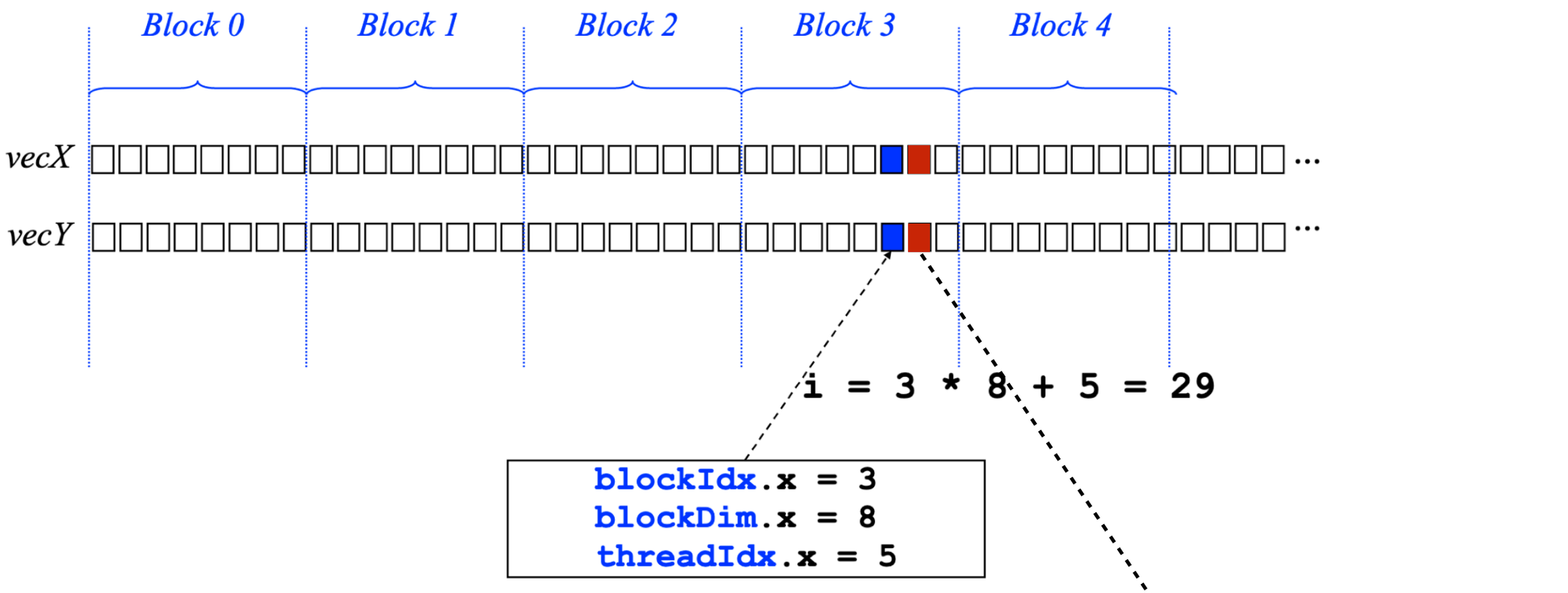
- Let's give an example:



- Super Important - coalesced memory:
  - Reading (global) memory should be from adjacent memory address for the threads

Index

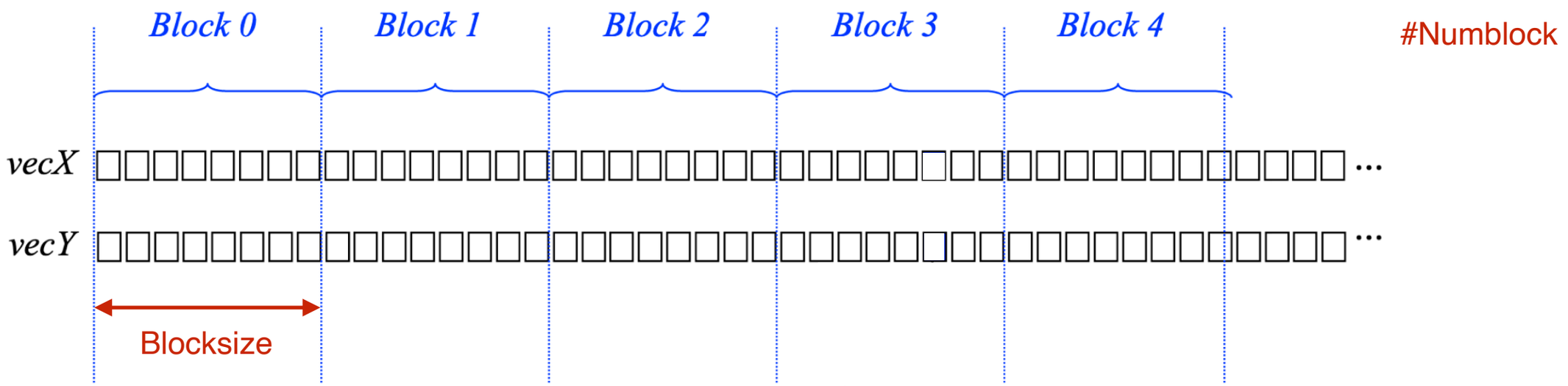
- Let's give an example:



- Super Important - coalesced memory:
  - Reading (global) memory should be from adjacent memory address for the threads

Kernel call

- How do you call a kernel?
  - ➡ saxpy<<<numblock, blocksize>>>(d_x, d_y, a, n)



- blocksize: number of thread in a block
  - Should be multiple of 32 (due to wrap)
  - Maximum of 2048
    - depends of the GPU capabilities

A complete GPU code

- Code steps in more details:
  1. Initialise GPU
  2. Initialise variable on the host (cpu)
  3. Allocate memory on the device (gpu)
  4. Move data from host to device
  5. Execute kernel on device
  6. Move back results
  7. Clean up (deallocation)

A complete GPU code

1. Initialise GPU
2. Initialise variable on the host (cpu)

```
int main(void)
{
  int N = 1<<20;
  float *x, *y, *d_x, *d_y;
  x = (float*)malloc(N*sizeof(float));
  y = (float*)malloc(N*sizeof(float));

  cuInit(0);
```

- culnit(0) is NOT required for the code to work
  - Will be called automatically at first cuda function call
  - Nice to use for profiling
    - Otherwise first call much slower than expected

A complete GPU code

3. Allocate memory on the device (gpu)

```
float *d_x, *d_y;
cudaMalloc(&d_x, N*sizeof(float));
cudaMalloc(&d_y, N*sizeof(float));
```

- cudaMalloc does NOT follow the exact same syntax as a malloc:

```
x = (float*)malloc(N*sizeof(float));
```

  - The cuda rule for any function is to return an error code
  - So the cuda malloc does not return a pointer but has one more argument (pointer of pointer)
- Here we use "d_" prefix to indicated device pointer.
  - Useful convention for code clarity

A complete GPU code

4. Move data from host to device

```
cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);
```

A complete GPU code

4. Move data from host to device

```
cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);
```

Device pointer

A complete GPU code

4. Move data from host to device

```
cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);
```

Device pointer

Host pointer

A complete GPU code

4. Move data from host to device

```
cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);
```

Device pointer

Host pointer

Transfer direction

A complete GPU code

4. Move data from host to device

```
cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);
```

Device pointer

Host pointer

Transfer direction

- Quite slow transfer but 2 tricks:

1. For simple initialisation/value
   - ➡ cudaMemSet(d_x, 0, N*sizeof(xxxxx))
2. Used hosted pinned memory for host
   - ➡ cudaMallocHost(&&x_host, size)
   - ➡ Slower allocation on host

A complete GPU code

5. Execute kernel on device

```
// Perform SAXPY on 1M elements
int   blocksize = 512;
int nblock = N/blocksize + ( n % blocksize > 0 ? 1: 0 );
saxpy<<<nblock, blocksize>>>( d_x, d_y, 2.0f, N);
```

A complete GPU code

5. Execute kernel on device

```
// Perform SAXPY on 1M elements
int   blocksize = 512;
int nblock = N/blocksize + ( n % blocksize > 0 ? 1: 0 );
saxpy<<<nblock, blocksize>>>( d_x, d_y, 2.0f, N);
```

- Computing the number of block needed
  - Special <<<A, B, C, D >>> syntax
    - A: number of block
    - B: number of thread per block
    - C: dynamically allocated shared memory
    - D: which stream to use

A complete GPU code

5. Execute kernel on device

```
// Perform SAXPY on 1M elements
int   blocksize = 512;
int nblock = N/blocksize + ( n % blocksize > 0 ? 1: 0 );
saxpy<<<nblock, blocksize>>>( d_x, d_y, 2.0f, N);
```

- Computing the number of block needed
  - Special <<<A, B, C, D >>> syntax
    - A: number of block
    - B: number of thread per block
    - C: dynamically allocated shared memory
    - D: which stream to use

```
__global__ void saxpy(float *d_VecX, float *d_VecY, float alpha, int n)
```

- __global__ to use for kernel called from the host
- __device__ for GPU function call from a kernel

A complete GPU code

6. Move back results
7. Clean up (deallocation)

```
cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);

float maxError = 0.0f;
for (int i = 0; i < N; i++)
  maxError = max(maxError, abs(y[i]-4.0f));
printf("Max error: %f\n", maxError);

cudaFree(d_x);
cudaFree(d_y);
free(x);
free(y);
```

A complete GPU code

6. Move back results
7. Clean up (deallocation)

Device pointer

```
cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);

float maxError = 0.0f;
for (int i = 0; i < N; i++)
  maxError = max(maxError, abs(y[i]-4.0f));
printf("Max error: %f\n", maxError);

cudaFree(d_x);
cudaFree(d_y);
free(x);
free(y);
```

A complete GPU code

6. Move back results
7. Clean up (deallocation)

Host pointer

Device pointer

```
cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);

float maxError = 0.0f;
for (int i = 0; i < N; i++)
  maxError = max(maxError, abs(y[i]-4.0f));
printf("Max error: %f\n", maxError);

cudaFree(d_x);
cudaFree(d_y);
free(x);
free(y);
```

Full code

```
#include <stdio.h>

__global__ void saxpy(float *d_VecX, float *d_VecY, float alpha, int n)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) d_VecY[i] = alpha*d_VecX[i] + d_VecY[i];
}

int main(void)
{
  int N = 1<<20;
  float *x, *y;
  x = (float*)malloc(N*sizeof(float));
  y = (float*)malloc(N*sizeof(float));

  cuInit(0);

  float *d_x, *d_y;
  cudaMalloc(&d_x, N*sizeof(float));
  cudaMalloc(&d_y, N*sizeof(float));

  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

  // Perform SAXPY on 1M elements
  int   blocksize = 512;
  int nblock = N/blocksize + ( n % blocksize > 0 ? 1: 0 );
  saxpy<<<nblock, blocksize>>>( d_x, d_y, 2.0f, N);

  cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);

  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = max(maxError, abs(y[i]-4.0f));
  printf("Max error: %f\n", maxError);

  cudaFree(d_x);
```

- How to compile it?

Compilation of cuda code

- Module load CUDA
- nvcc -arch=sm_70 saxpy.cu -o saxpy
  - ➡ You can have additional flags for C++ par of the code (library linking, -O3,…)
  - ➡ Arch allows to have a minimum target gpu
  - ➡ No dedicated flag for additional GPU optimisation
  - ➡ GPU does support multiple file source code
    - ✦ But seriously limit optimisation
      - Cuda11 starts supports for that but still limited.

Is GPU always faster?

- GPU

```
[omatt@mb-cas102 omatt]$ time ./saxpy
Max error: 0.000000

real    0m2.401s
user    0m0.752s
sys     0m0.555s
```

- CPU

```
[omatt@mb-cas102 omatt]$ time ./saxpy_cpu
Max error: 0.000000

real    0m0.803s
user    0m0.704s
sys     0m0.097s
```

- 10 kernel

```
real    0m4.029s
user    0m2.284s
sys     0m1.091s
```

- 100 kernel

```
real    0m25.084s
user    0m23.164s
sys     0m1.302s
```

- 10 kernel

```
real    0m5.265s
user    0m5.202s
sys     0m0.057s
```

- 100 kernel

```
real    0m45.636s
user    0m45.524s
sys     0m0.067s
```

- The gpu initialisation is large for simple problem like this one
- You need to optimise your GPU code to hide the latency, data transfer, …
  - This GPU code can be speed-up quite a lot

Type of Memory available

- You have to manage memory: Plenty of type of memory on the GPU
  - Inside each SM:

Type of Memory available

- You have to manage memory: Plenty of type of memory on the GPU

- Inside each SM:
  - Register
    - Fastest memory
    - Thread specific
    - Very limited amount
    - Overflow goes to L1

Type of Memory available

- You have to manage memory: Plenty of type of memory on the GPU

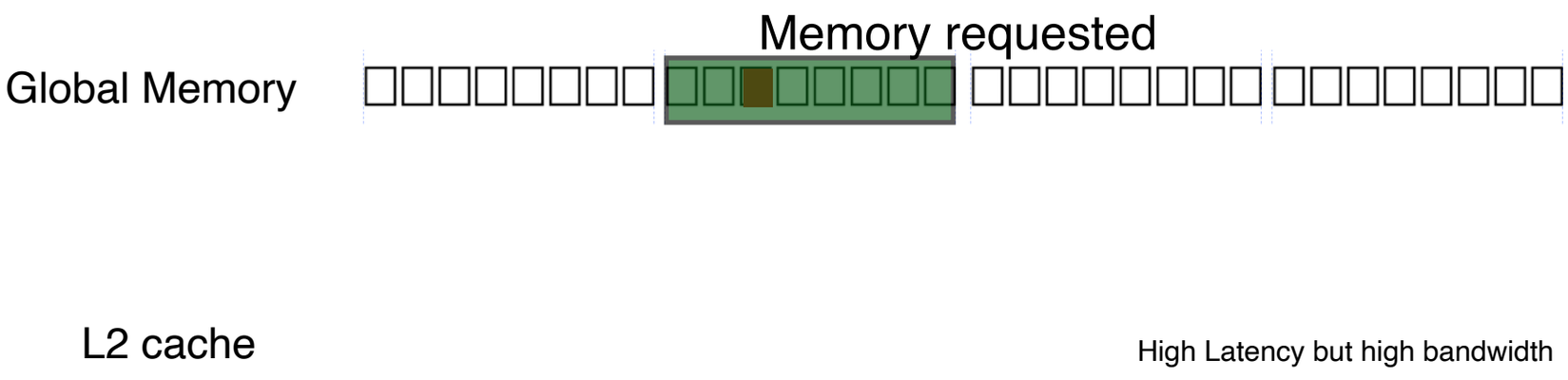
- Inside each SM:
  - Register
    - Fastest memory
    - Thread specific
    - Very limited amount
    - Overflow goes to L1
  - Shared memory
    - Limited amount
    - Block wide memory
    - __shared__

Type of Memory available

- You have to manage memory: Plenty of type of memory on the GPU
- Outside the SM
    - Global memory
        - High bandwidth (900Gb/s) but High latency
        - High number of thread need to hide this latency
        - Default memory for cpu/gpu pointer

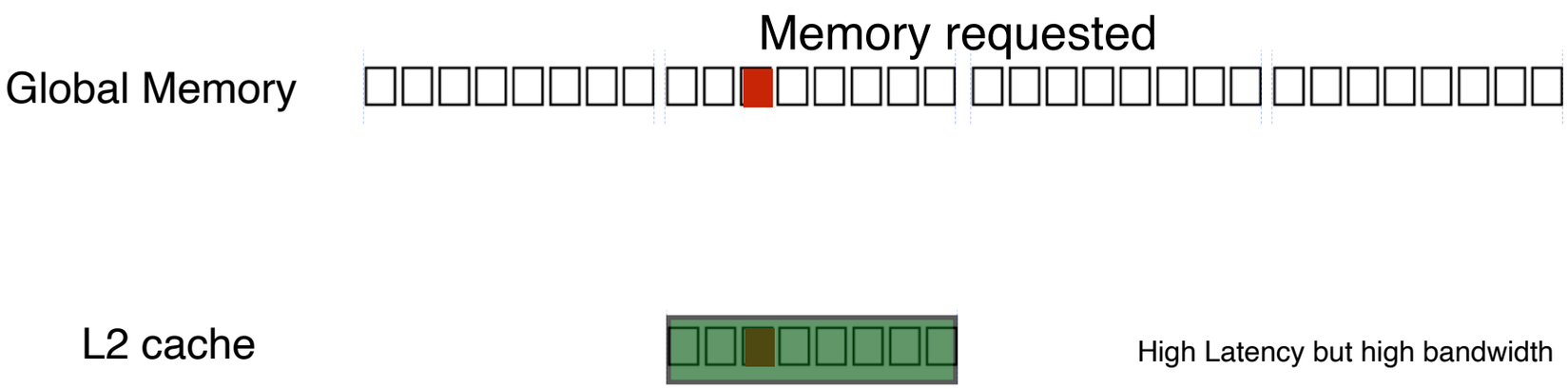
Type of Memory available

- You have to manage memory: Plenty of type of memory on the GPU
- Outside the SM
    - Global memory
        - High bandwidth (900Gb/s) but High latency
        - High number of thread need to hide this latency
        - Default memory for cpu/gpu pointer



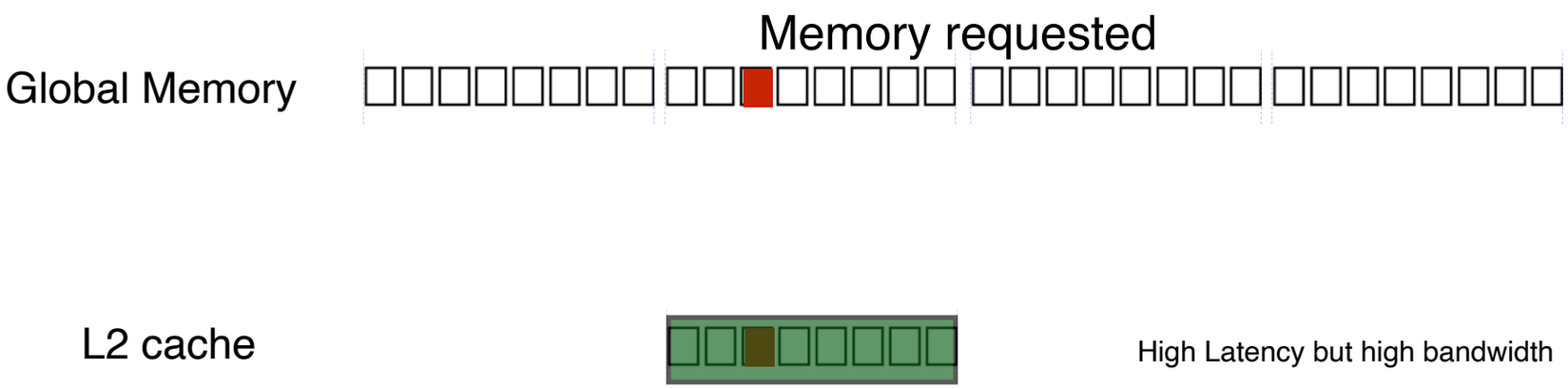
Type of Memory available

- You have to manage memory: Plenty of type of memory on the GPU
- Outside the SM
    - Global memory
        - High bandwidth (900Gb/s) but High latency
        - High number of thread need to hide this latency
        - Default memory for cpu/gpu pointer



Type of Memory available

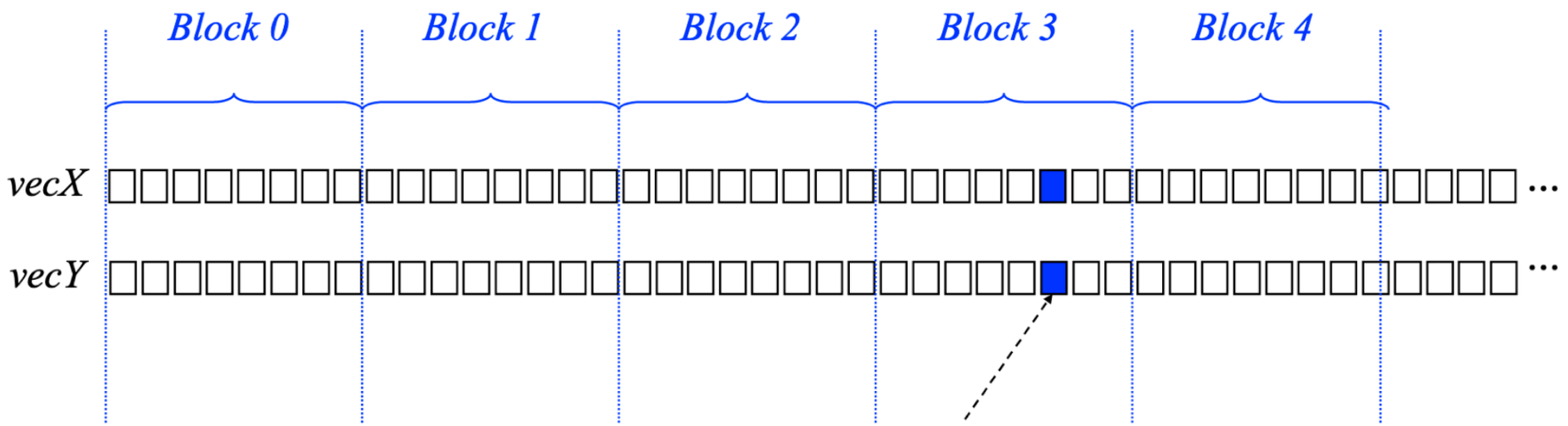
- You have to manage memory: Plenty of type of memory on the GPU
- Outside the SM
    - Global memory
        - High bandwidth (900Gb/s) but High latency
        - High number of thread need to hide this latency
        - Default memory for cpu/gpu pointer



Efficiency in memory transfer = maximise the usage of the data transfer, need to use all data in a block before needing a new block transfer

Index

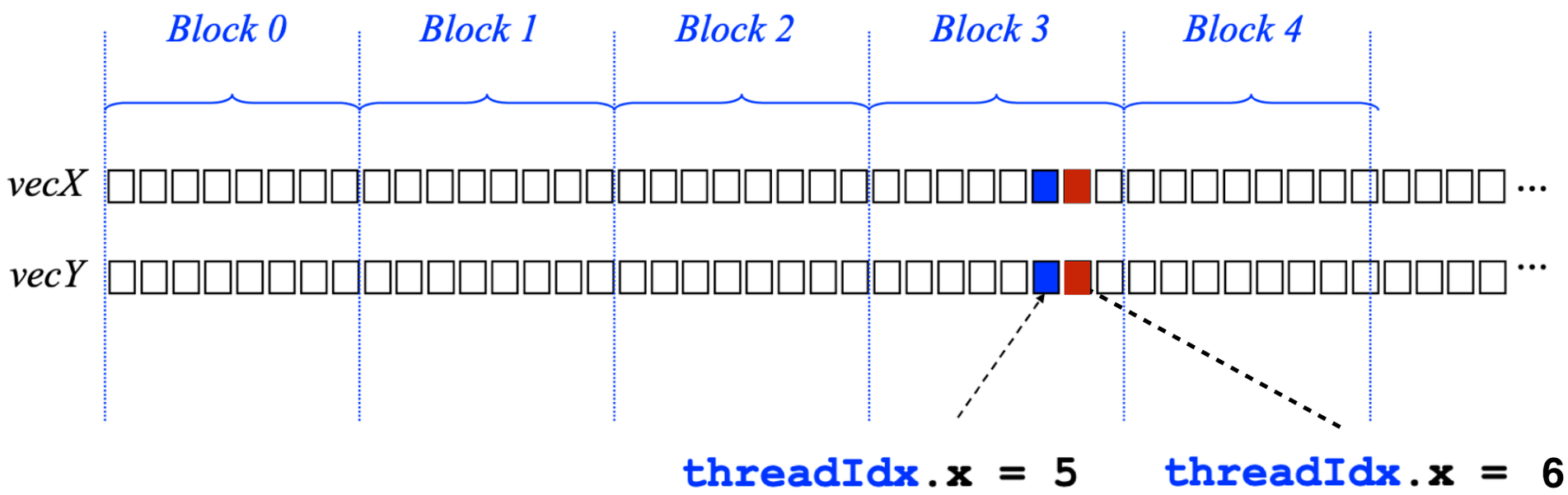
- This is how the memory should be read/write by the various thread



- You need to be careful with 2D array to be sure that you follow that pattern

Index

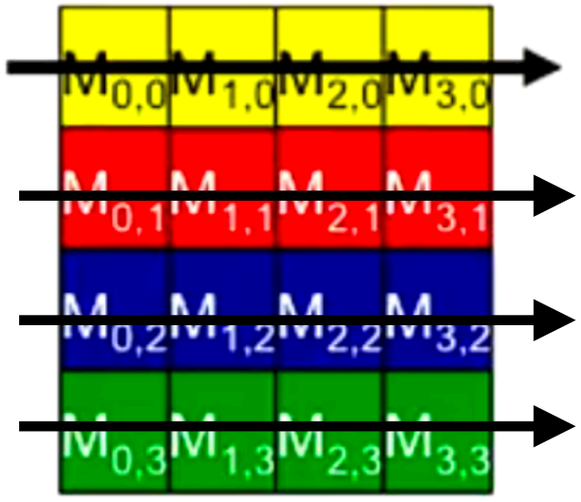
- This is how the memory should be read/write by the various thread

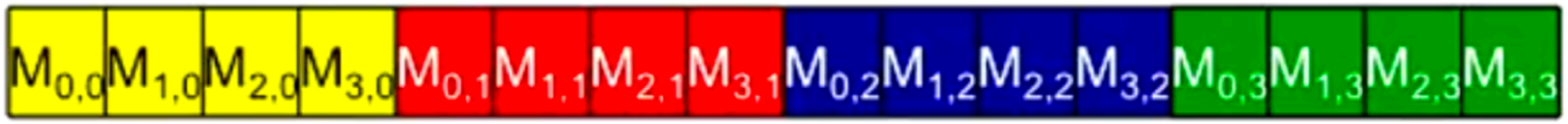


- You need to be careful with 2D array to be sure that you follow that pattern

Uncoalesced Memory

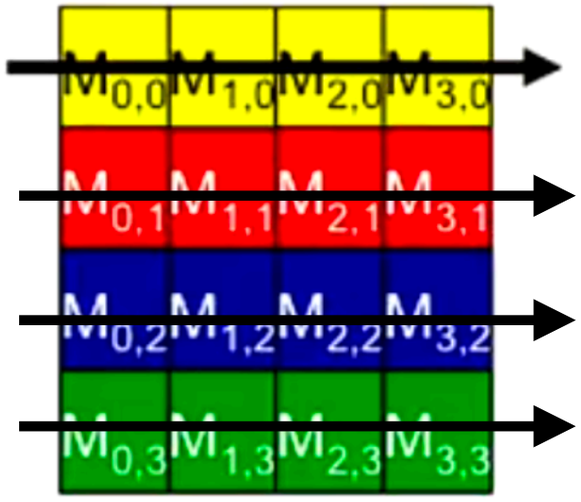
Access direction in Kernel code

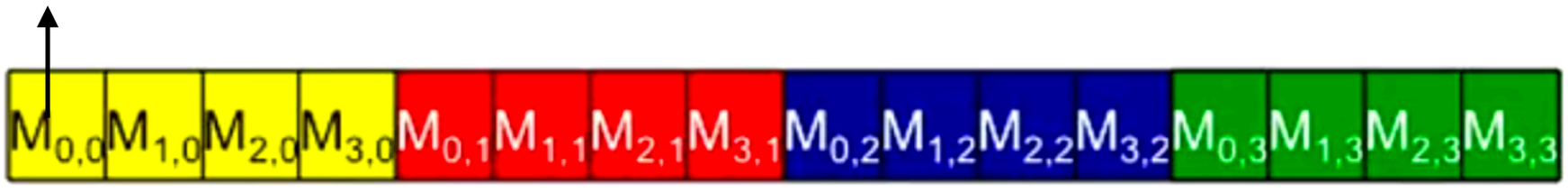




Uncoalesced Memory

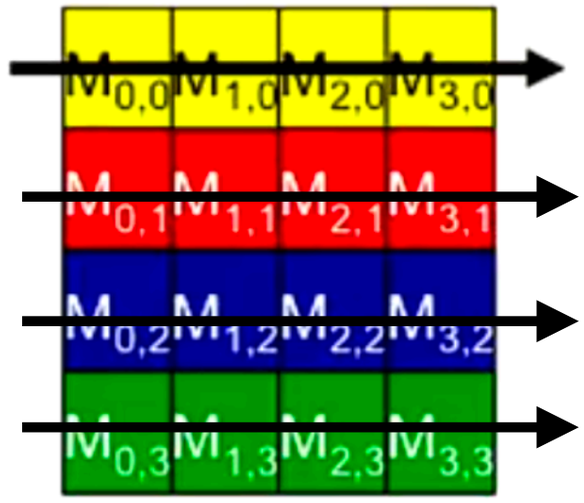
Access direction in Kernel code

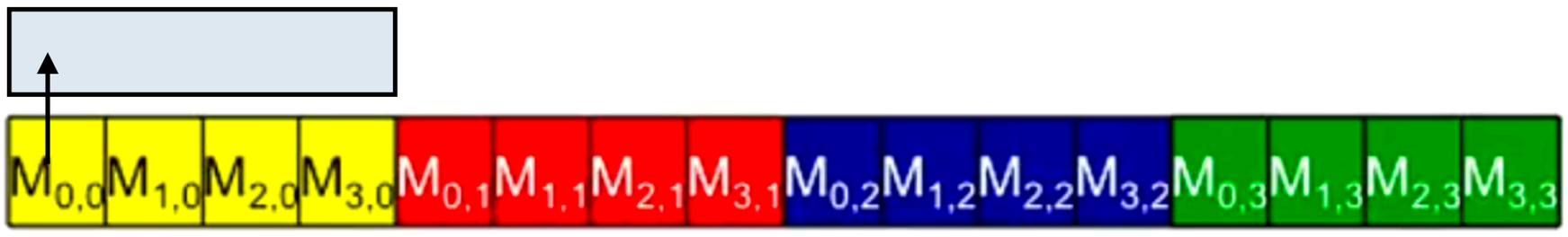




Uncoalesced Memory

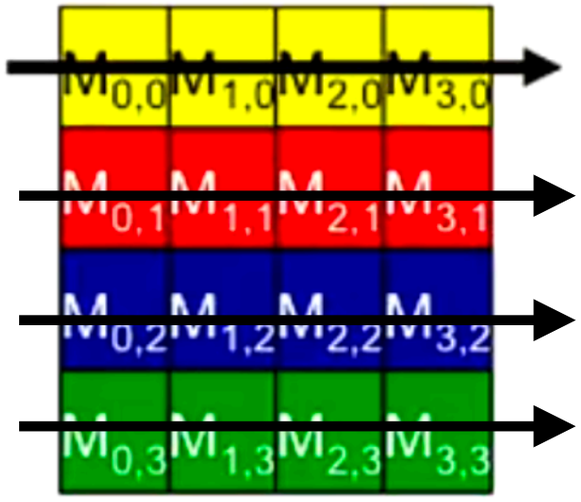
Access direction in Kernel code

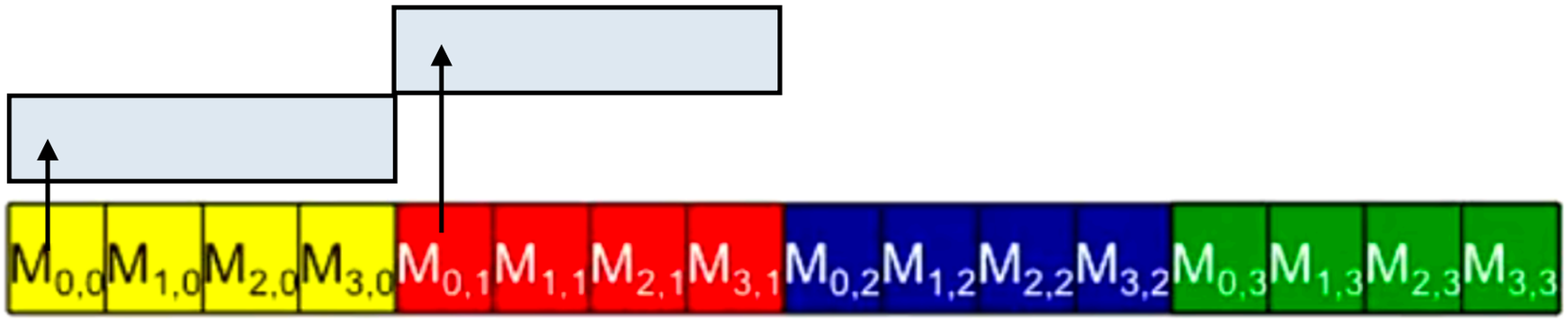




Uncoalesced Memory

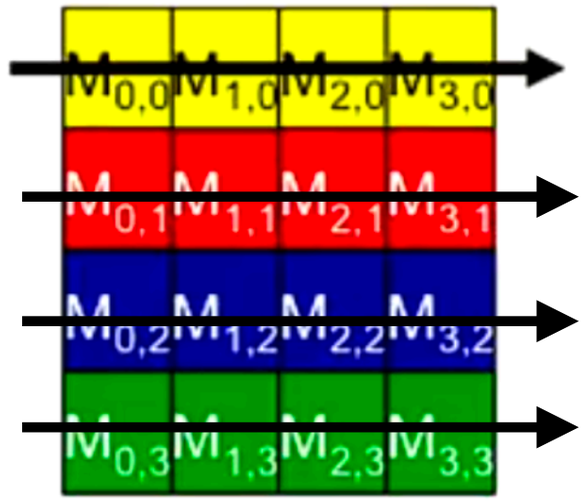
Access direction in Kernel code

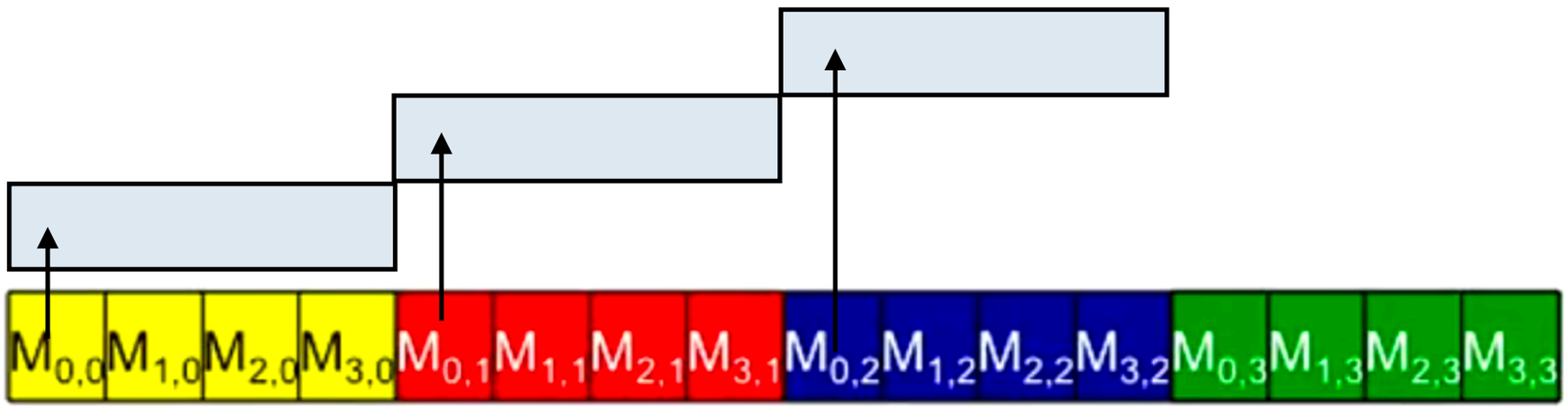




Uncoalesced Memory

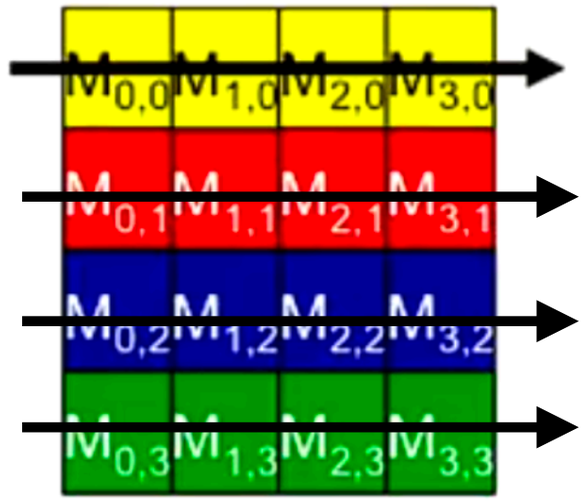
Access direction in Kernel code

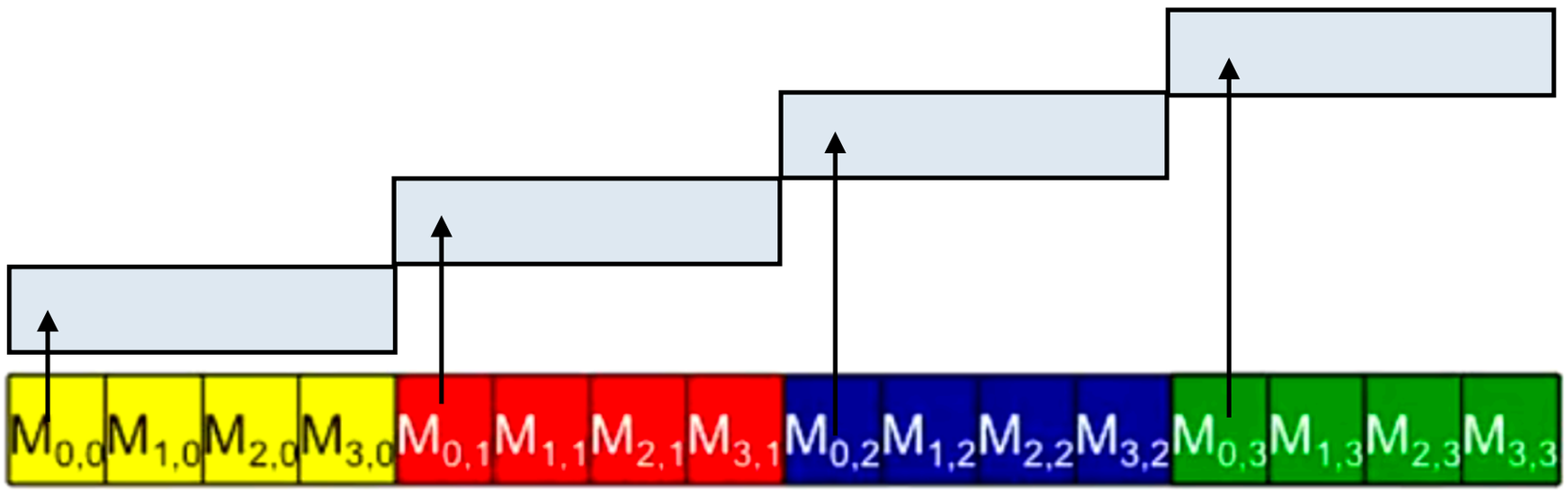




Uncoalesced Memory

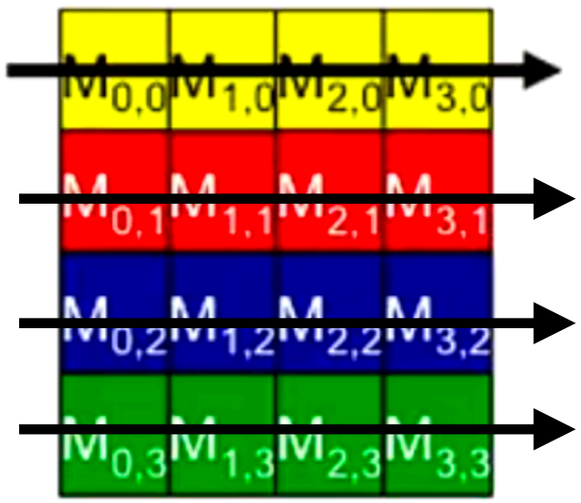
Access direction in Kernel code





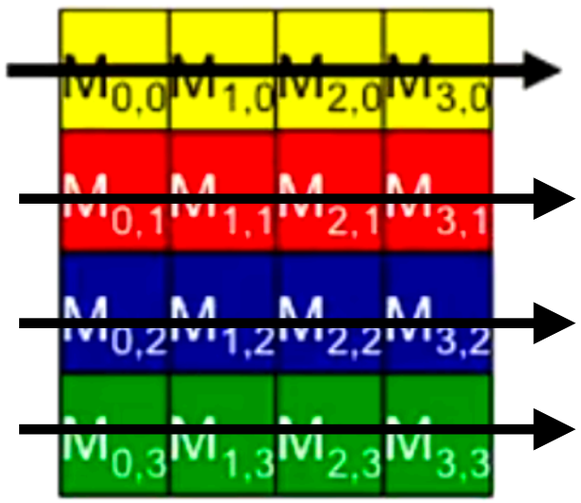
Uncoalesced Memory

Access direction in Kernel code

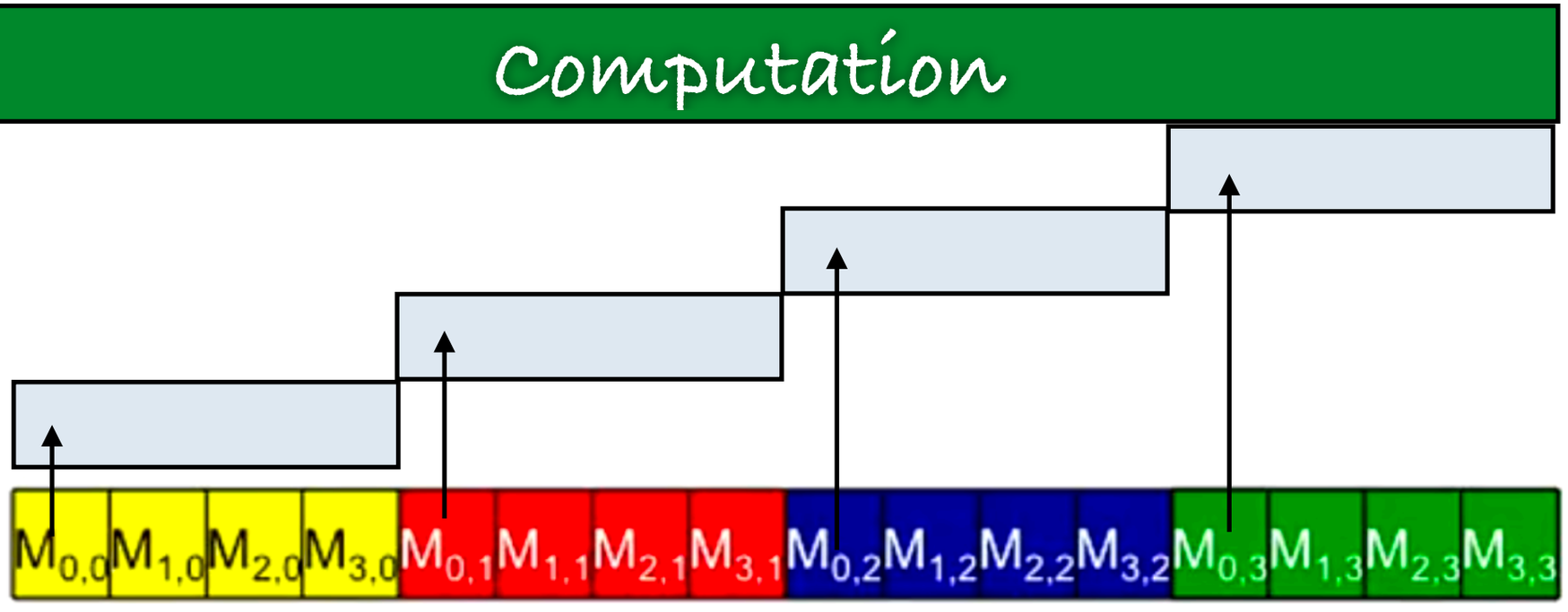


Uncoalesced Memory

Access direction in Kernel code

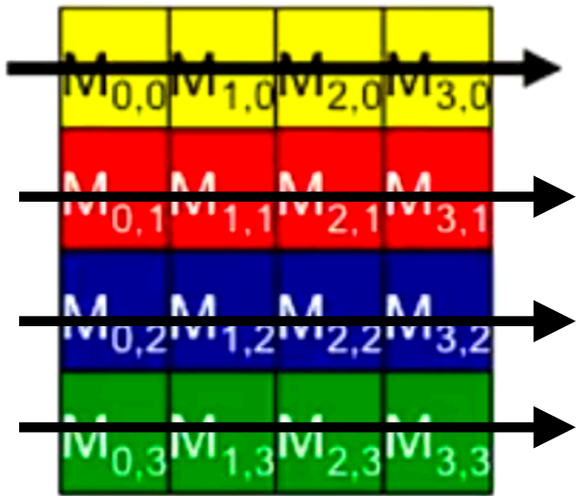


Time to move from global memory to register
4 times the latency



Uncoalesced Memory

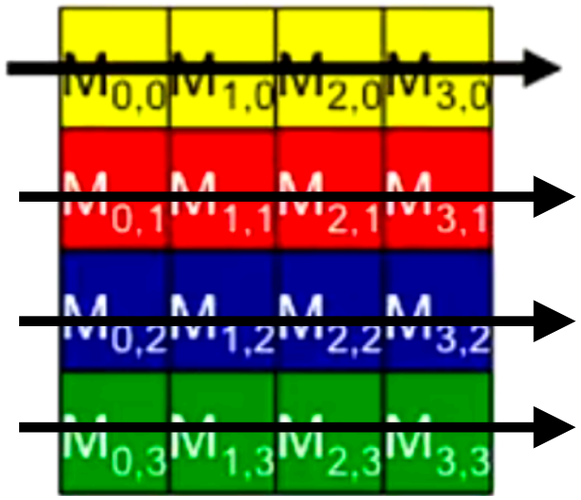
Access direction in Kernel code



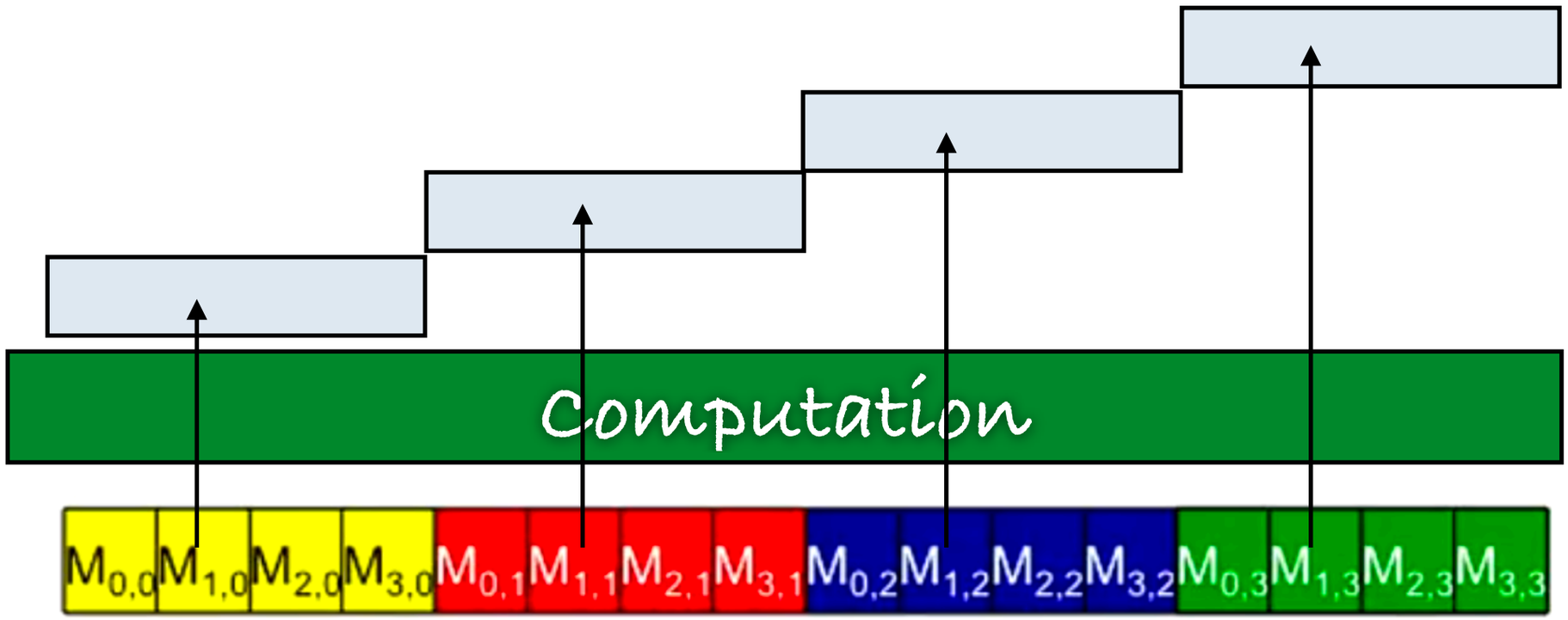
Computation

Uncoalesced Memory

Access direction in Kernel code

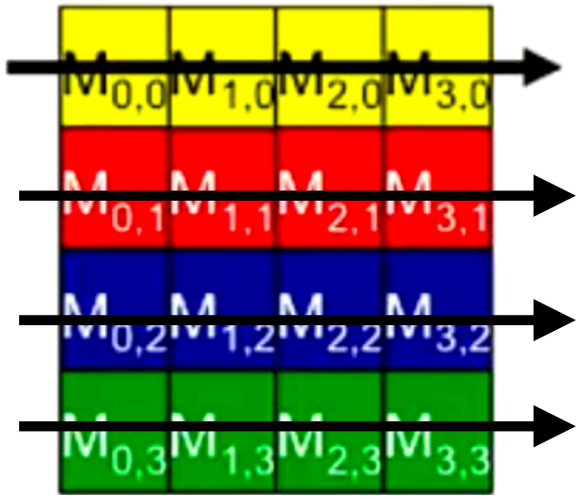


Likely not in cache anymore
-> need to extract
-> four times the latency

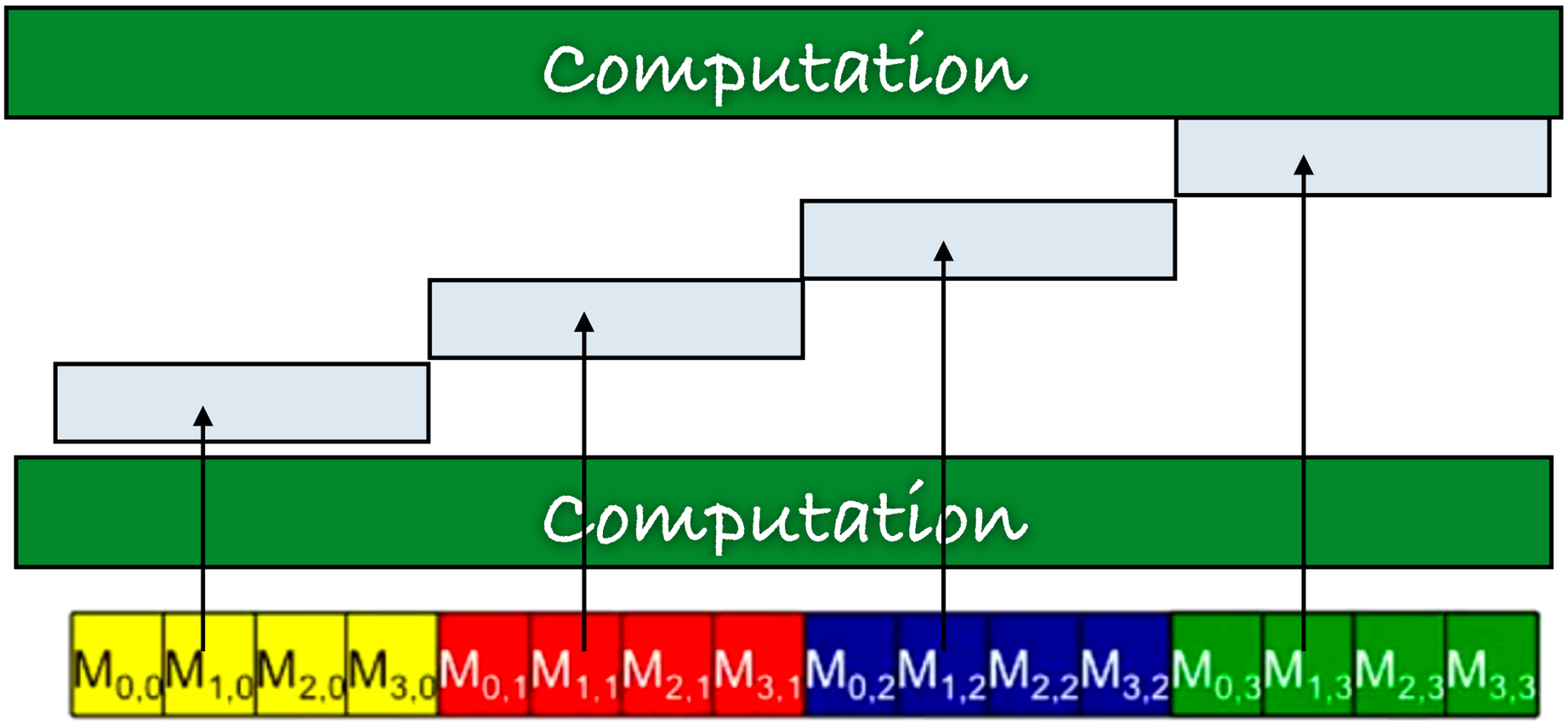


Uncoalesced Memory

Access direction in Kernel code

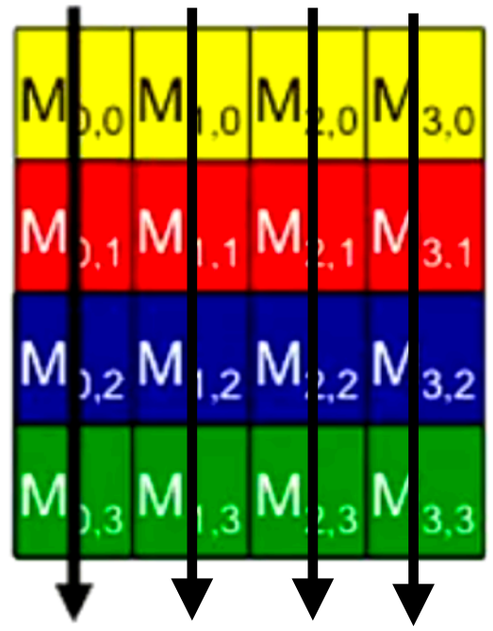


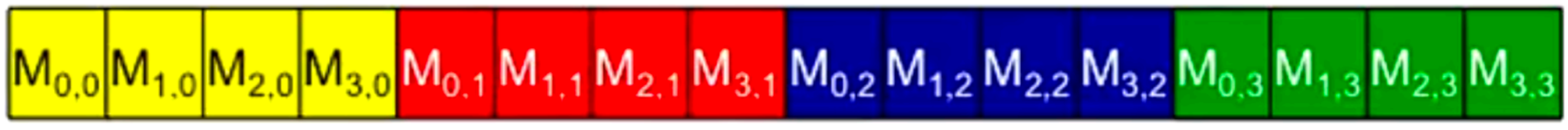
Likely not in cache anymore
-> need to extract
-> four times the latency



Coalesced memory

Access direction in Kernel code

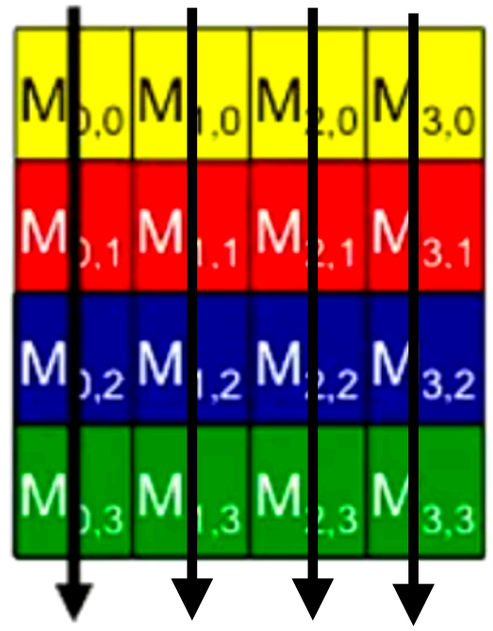


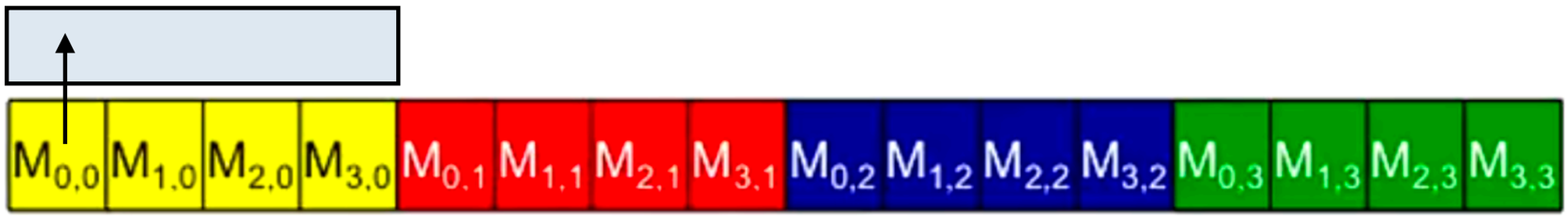


Slide credit: Hwu & Kirk

Coalesced memory

Access direction in Kernel code

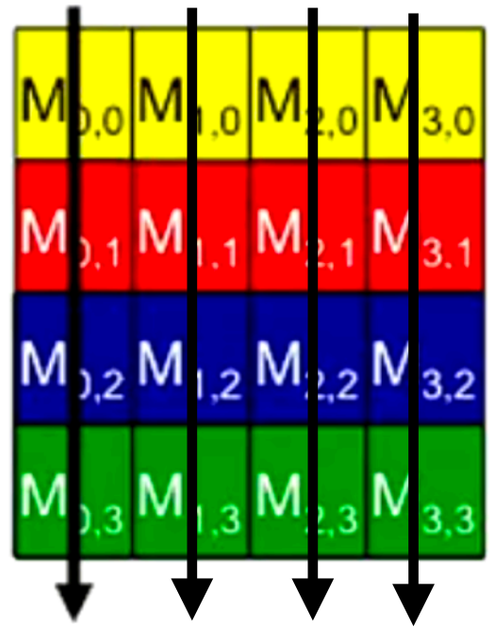


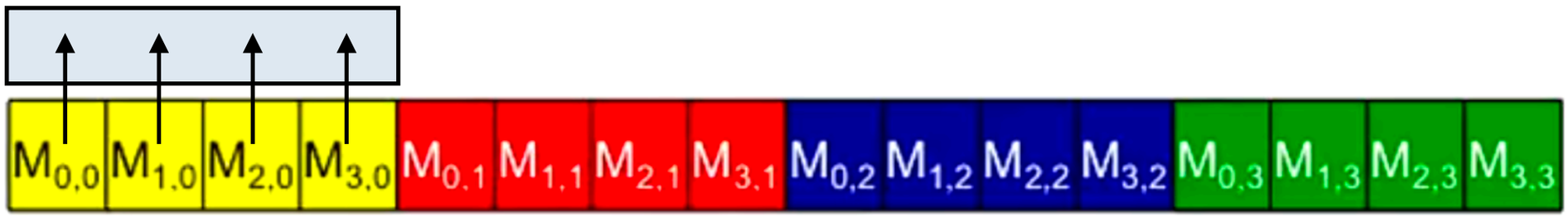


Slide credit: Hwu & Kirk

Coalesced memory

Access direction in Kernel code

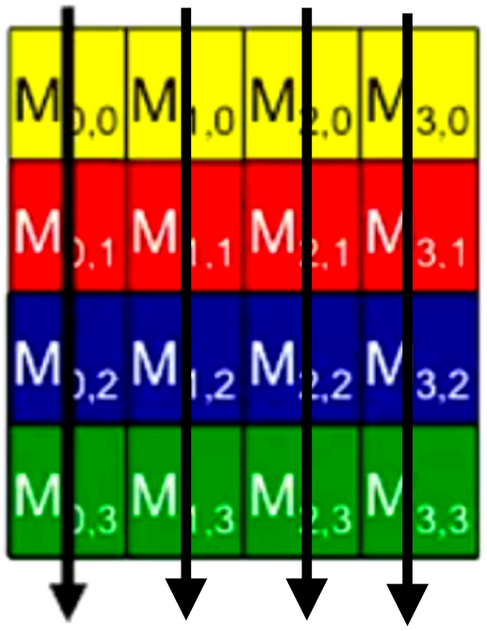


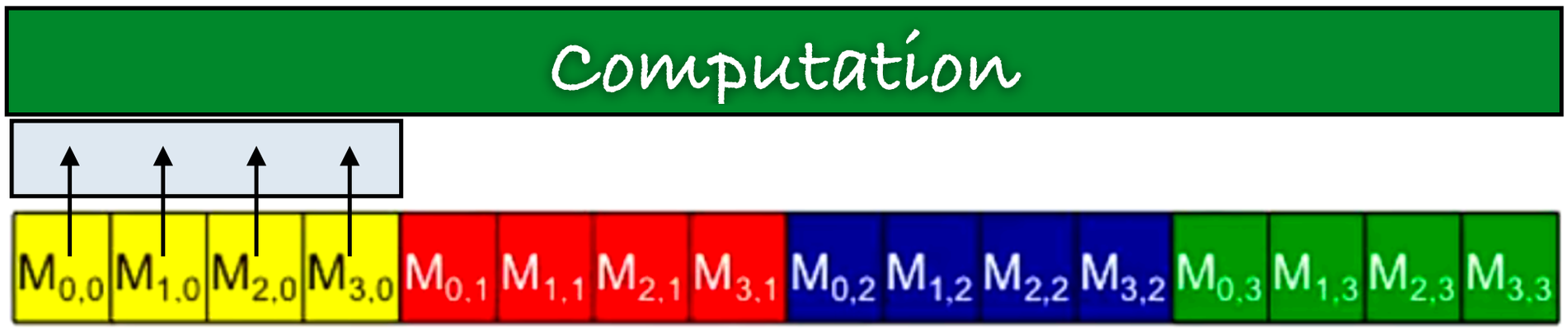


Slide credit: Hwu & Kirk

Coalesced memory

Access direction in Kernel code

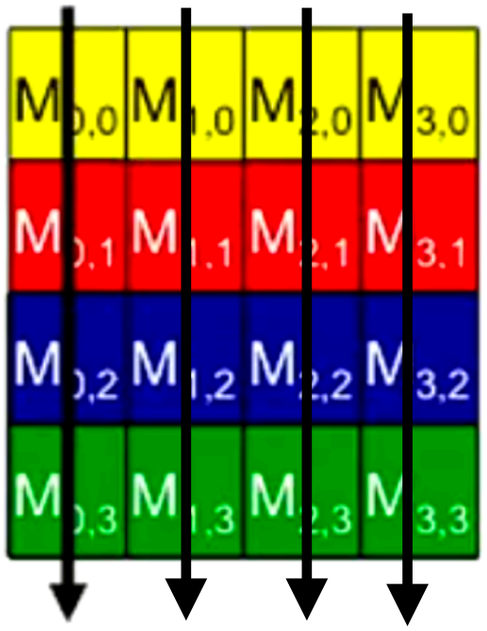


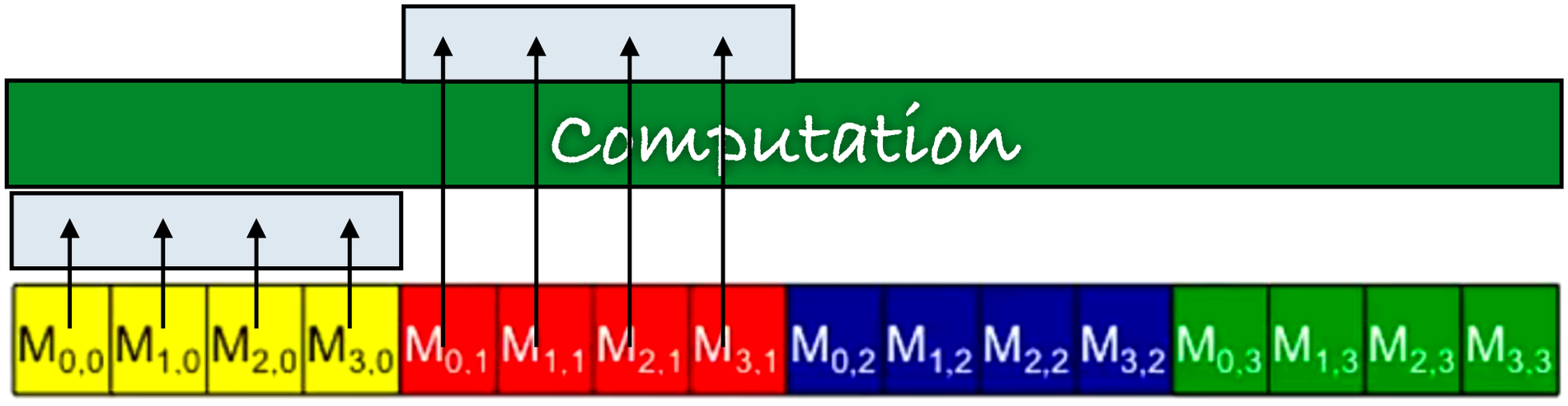


Slide credit: Hwu & Kirk

Coalesced memory

Access direction in Kernel code

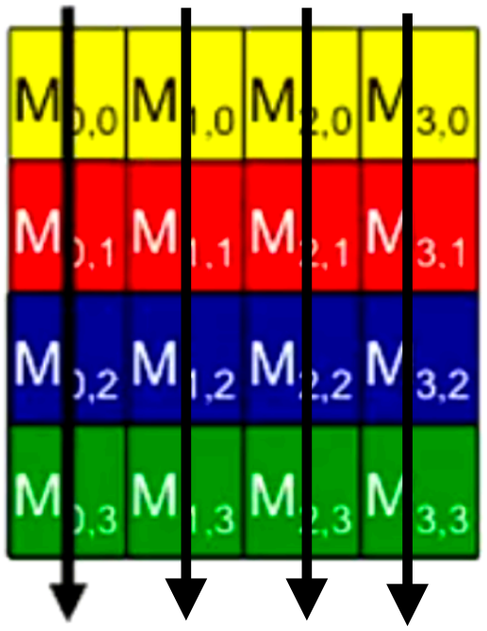


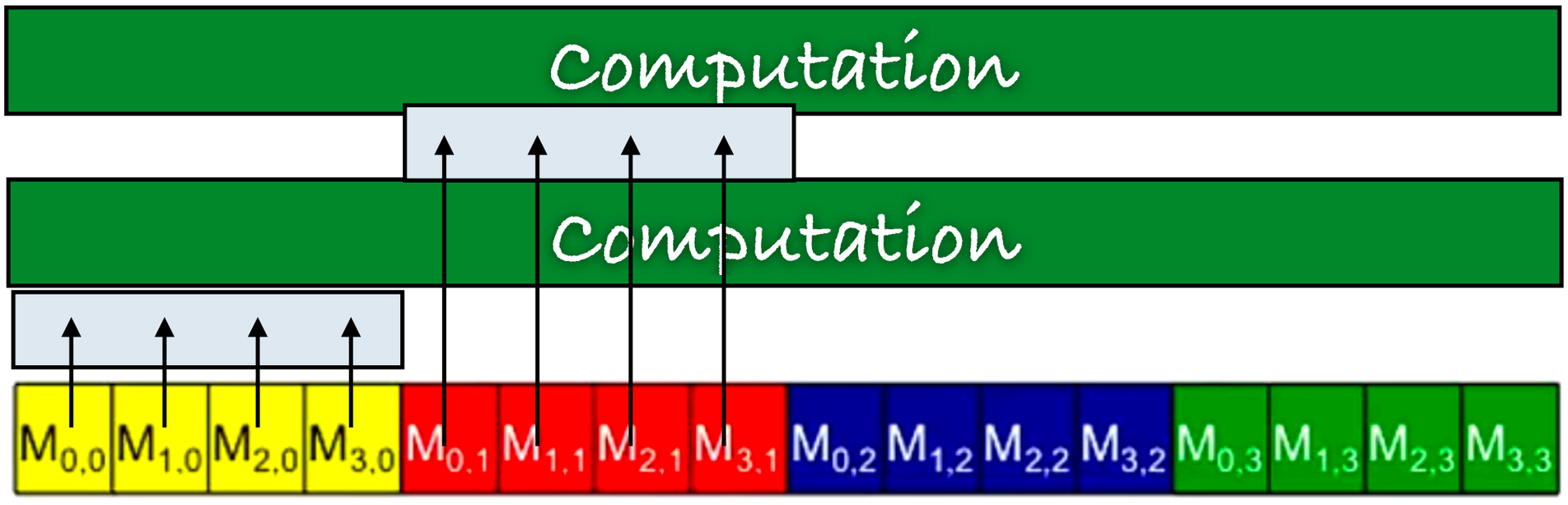


Slide credit: Hwu & Kirk

Coalesced memory

Access direction in Kernel code

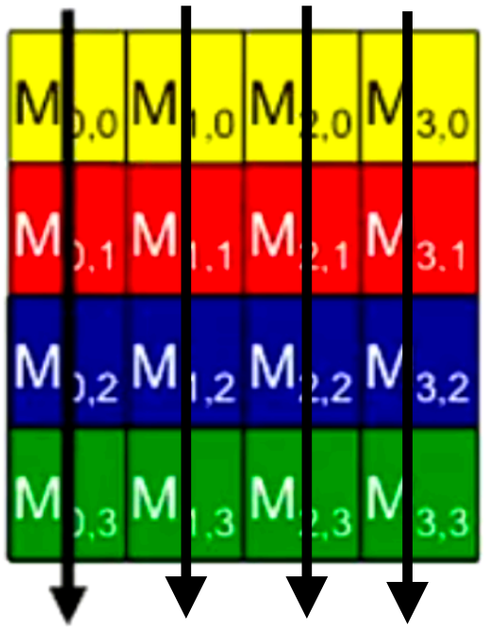


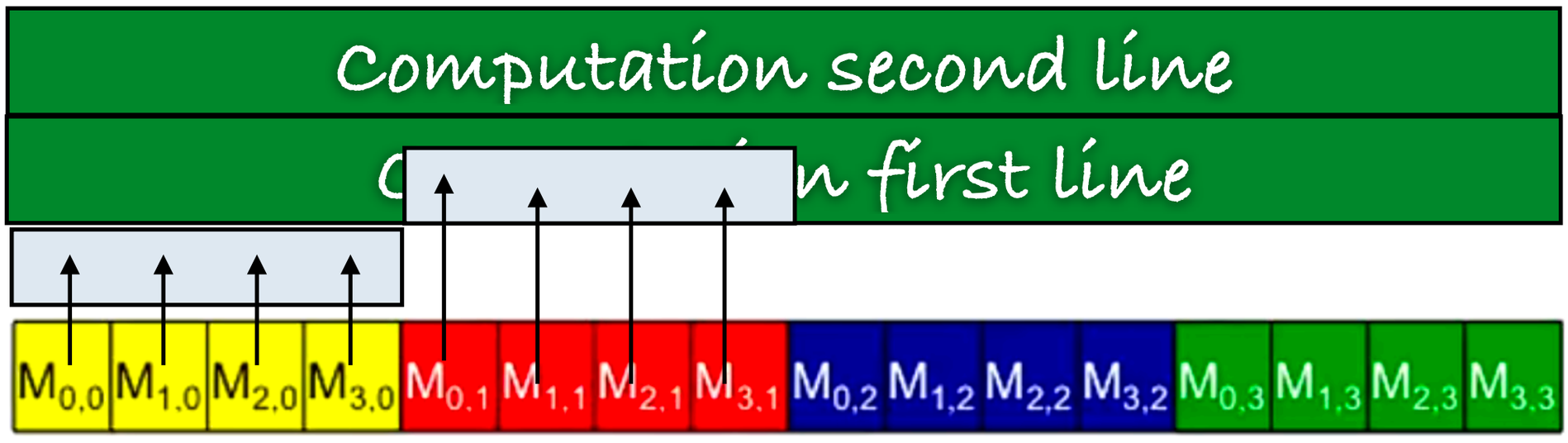


Slide credit: Hwu & Kirk

Coalesced memory

Access direction in Kernel code





Slide credit: Hwu & Kirk

Structure of array

- AoS vs. SoA

Structure of Arrays (SoA)

```
struct foo{
  float a[8];
  float b[8];
  float c[8];
  int d[8];
} A;
```

Better for GPU/ vectorised CPU

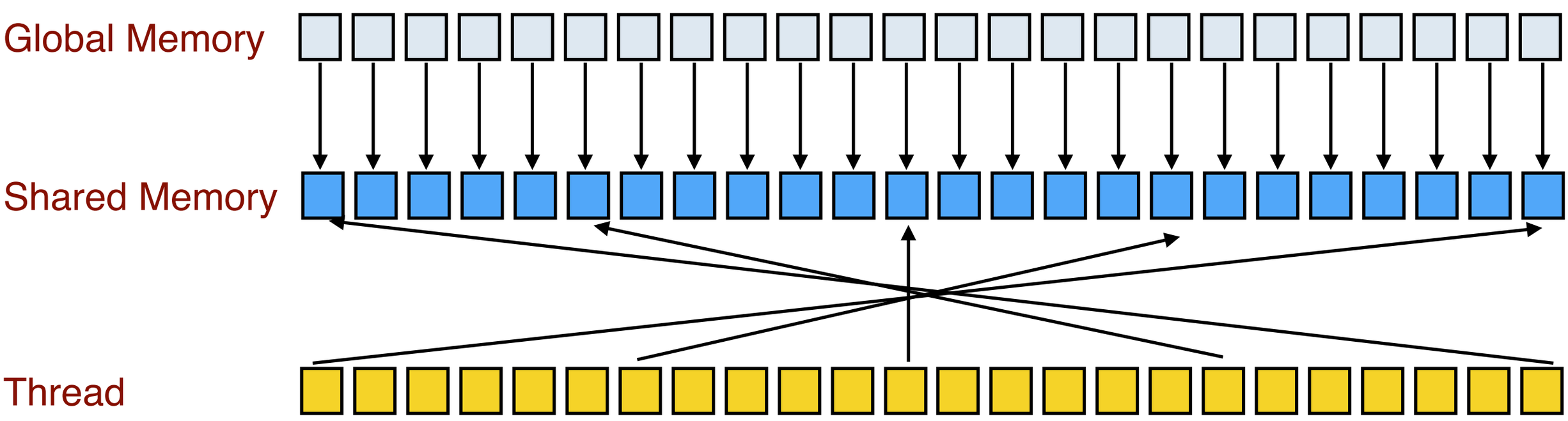
Array of Structures (AoS)

```
struct foo{
  float a;
  float b;
  float c;
  int d;
} A[8];
```

Better for non-vectorised operation for CPU

Coalesced access

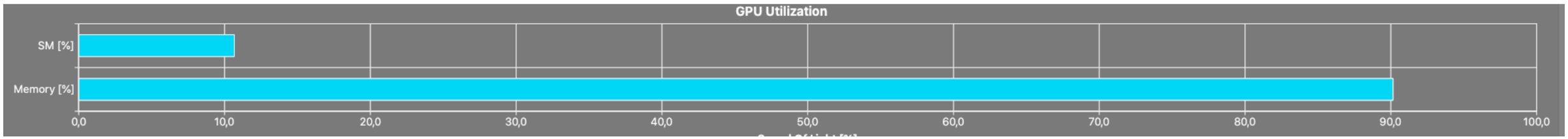
- Coalesced access not possible?
  - ➡ Use shared memory as a cache



CUDA profiler

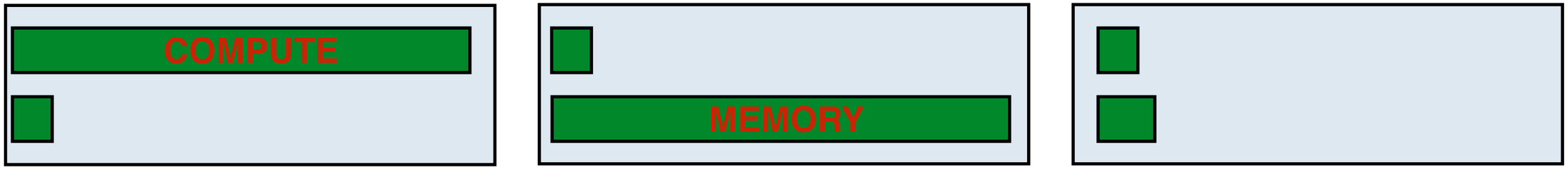
- nv-nsight-cu-cli -o profile --target-processes all ./ saxpy
  - ➡ Executable is also sometimes "ncu"
  - ➡ The more convenient is to download back that profile on your laptop and use "nsight compute" to visualise the data (do not need a GPU on that machine)
- On cluster mode, you need to be sudo to run those command. Contact us if needed.

What is the limitation of your kernel?



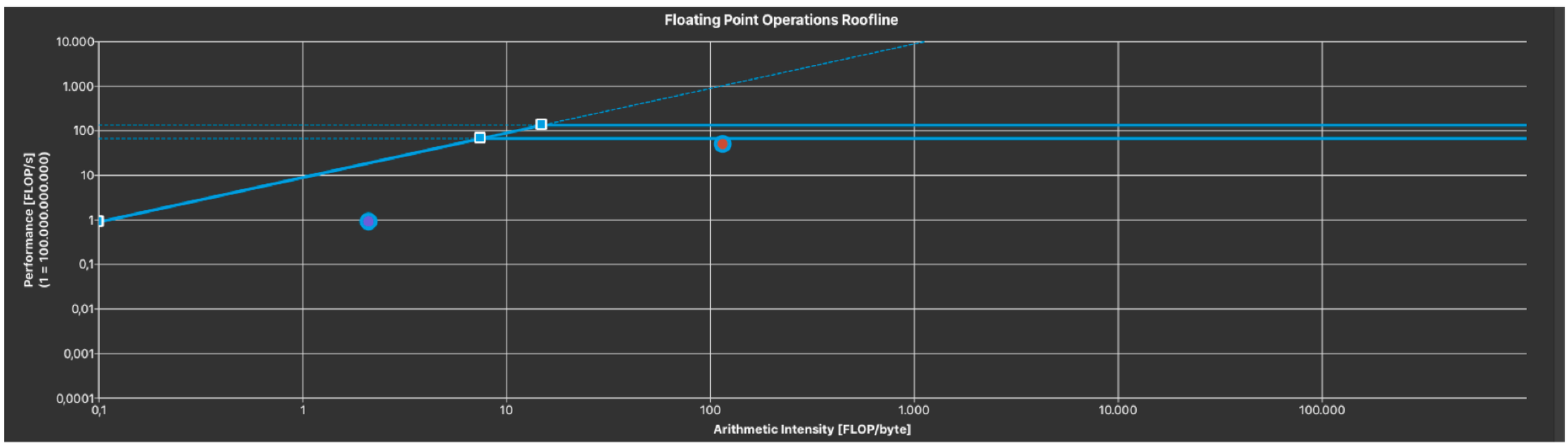
- Here two metric
  - ➡ How much the code compute (here 10%)
  - ➡ How much memory you use (here 90%)
- This indicates what limit your computation
  - ➡ Here we are Memory bound

What is the limitation of your kernel?



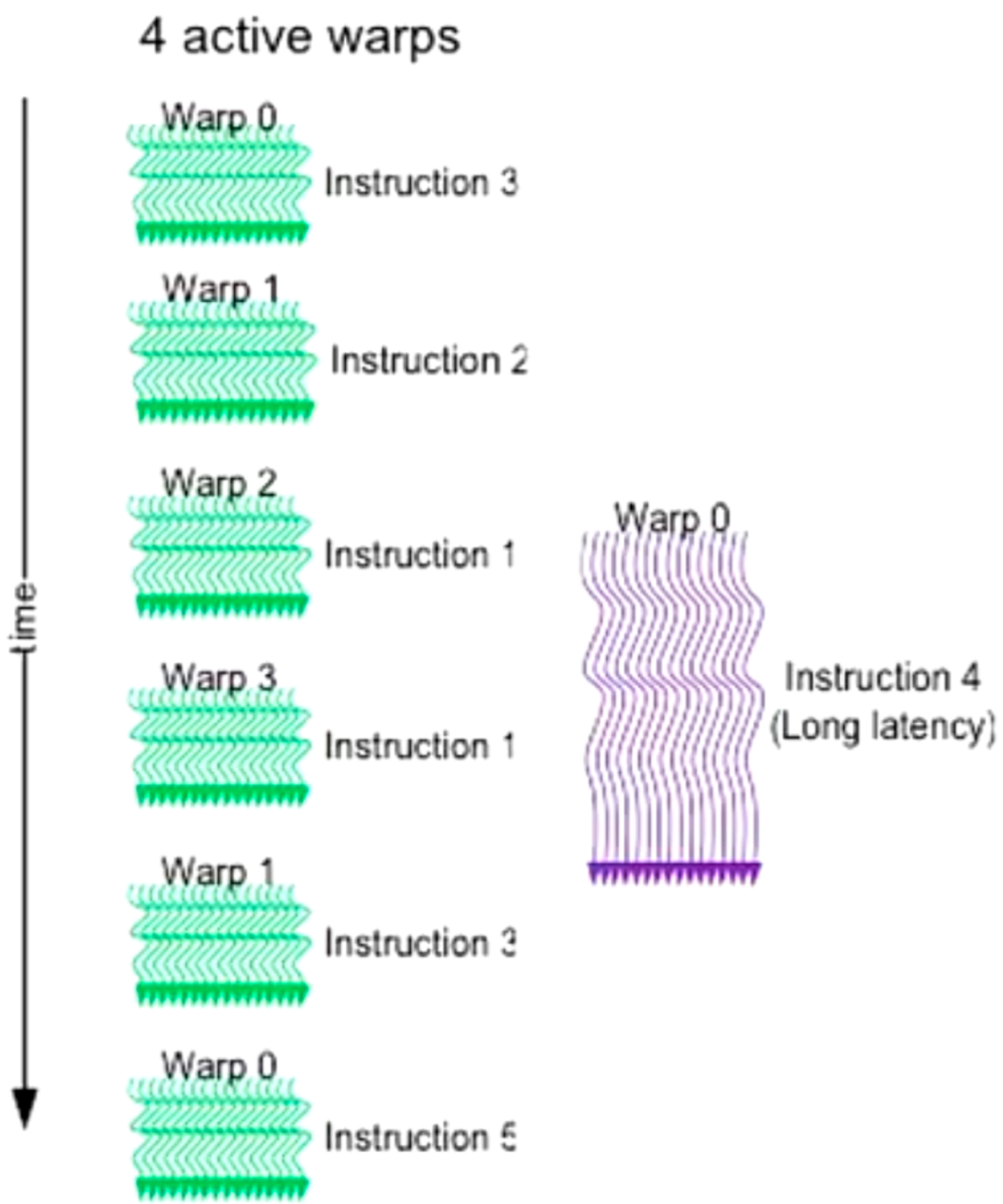
COMPUTE BOUND | MEMORY BOUND | LATENCY BOUND

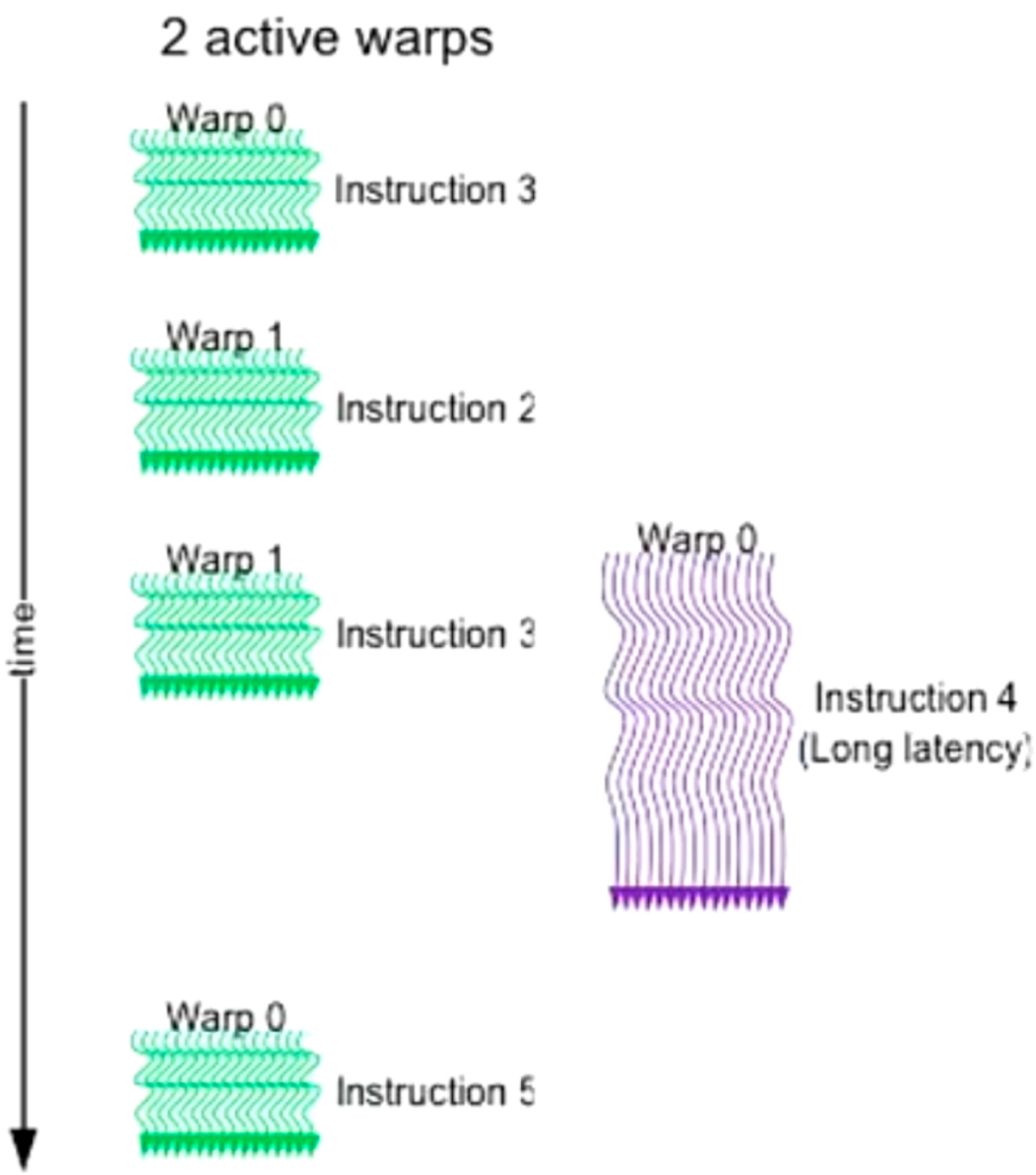
- Ideal case: compute AND memory bound
- If you are latency bound you need to allow more parallelism



Effect of occupancy

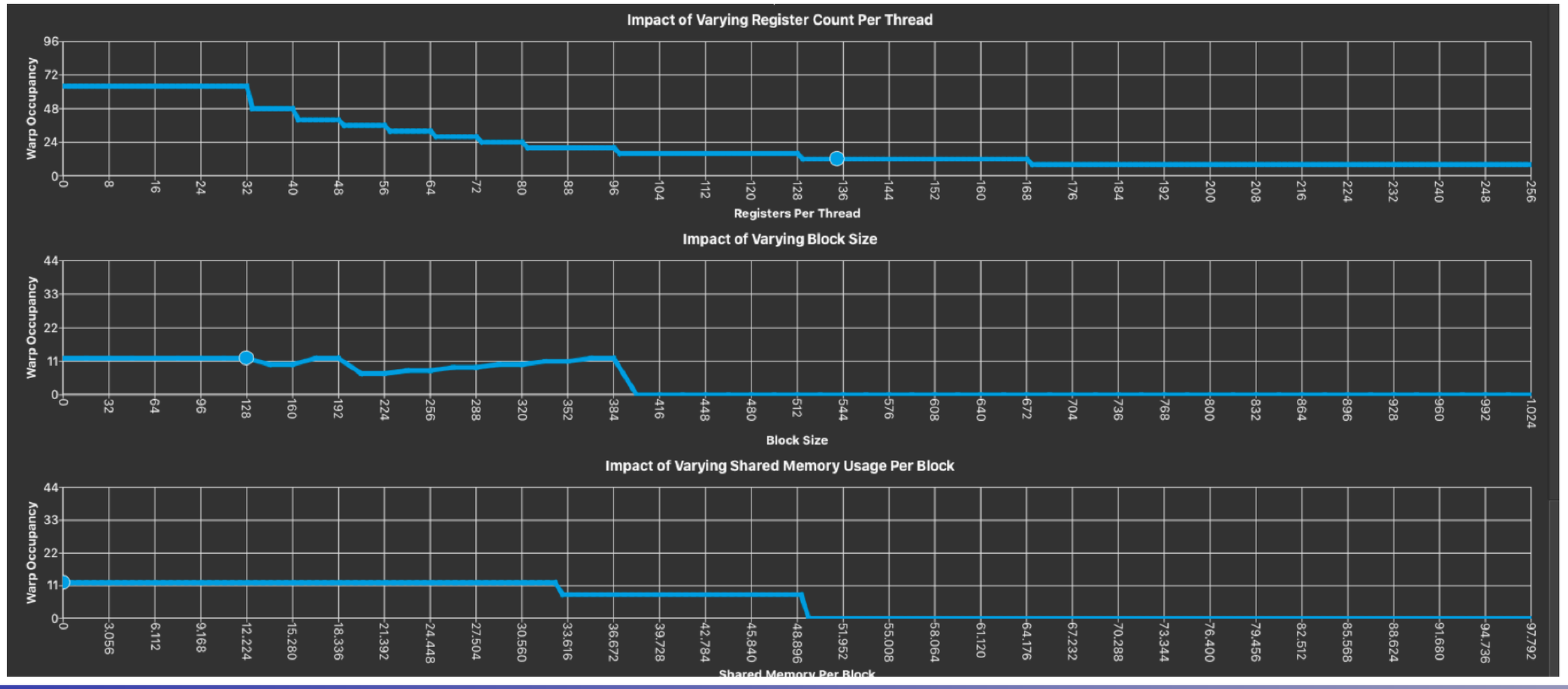
- Hide latency with other wrap





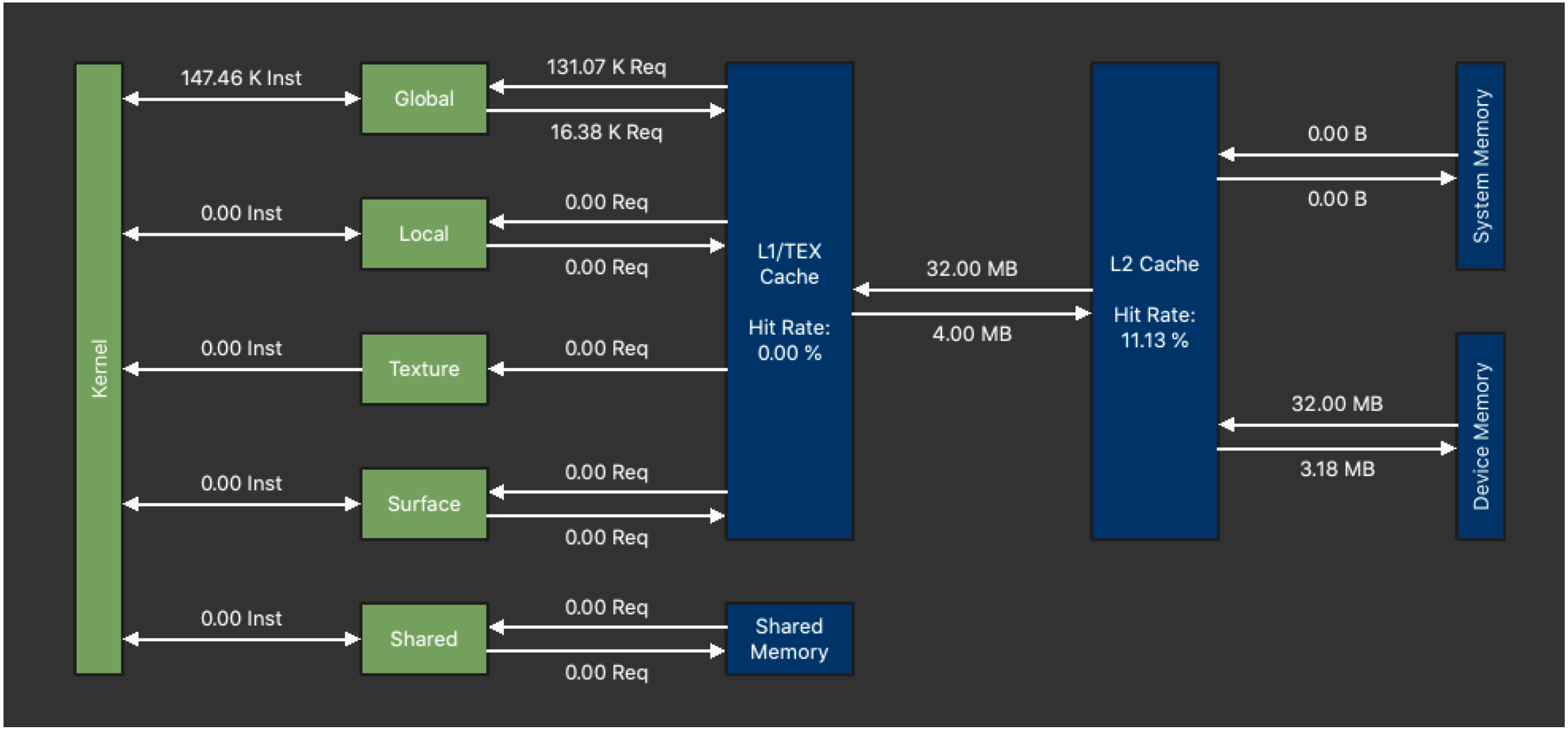
Occupancy

- Occupancy is limited
  - ➡ Each SM has limited ressources
    - ✦ Maximum number of wrap (64)
    - ✦ Maximum number of block (32)
    - ✦ Register usage (256Kb)
    - ✦ Shared memory usage (64Kb)



Checking memory

- You should also check where your memory bottleneck are



Nice series of tutorial:

- https://developer.nvidia.com/blog/even-easier-introduction-cuda/
  - ➡ At the bottom of the page you have a list of quite progressive tutorial
- Nice video on Cuda 5: https://www.youtube.com/watch?v=irvhW7oSNeQ&list=PLGvfHSgImk4aAt3R3XKvUMIv_RFOzSnWz&index=2
- Nice presentation: https://cac-staff.github.io/summer-school-2018/files/cuda_day1_summer_school_2018.pdf

GPU Features	NVIDIA Tesla P100	NVIDIA Tesla V100	NVIDIA A100
GPU Codename	GP100	GV100	GA100
GPU Architecture	NVIDIA Pascal	NVIDIA Volta	NVIDIA Ampere
Compute Capability	6.0	7.0	8.0
Threads / Warp	32	32	32
Max Warps / SM	64	64	64
Max Threads / SM	2048	2048	2048
Max Thread Blocks / SM	32	32	32
Max 32-bit Registers / SM	65536	65536	65536
Max Registers / Block	65536	65536	65536
Max Registers / Thread	255	255	255
Max Thread Block Size	1024	1024	1024
FP32 Cores / SM	64	64	64
Ratio of SM Registers to FP32 Cores	1024	1024	1024
Shared Memory Size / SM	64 KB	Configurable up to 96 KB	Configurable up to 164 KB

Conclusion

- GPU is a high throughput
  - ➡ High latency
- Various level of parralelism
  - ➡ thread/wrap/block
- Various type of memory
  - ➡ register/shared memory/global memory
- Optimization for the hardware is key
  - ➡ Coalesced memory ->Array of structure
  - ➡ Shared memory