Nvidia CUDA

https://github.com/ilcors-dev/unibo-nvidia-cuda-research

WHY

- GPUs have never really been a central subject of study in the courses in both BS / MS
- How do they work?
- Why are they so needed nowadays?
- How can we program them?

CUDA

- Stands for Compute Unified Device Architecture
- An extension of C/C++ languages
- Gives access to the power of GPUs, increasing the performance of certain types of programs
- GPUs excel in handling multiple operations simultaneously, making them ideal for computations that can be parallelized

GPU – Graphics Processing Unit

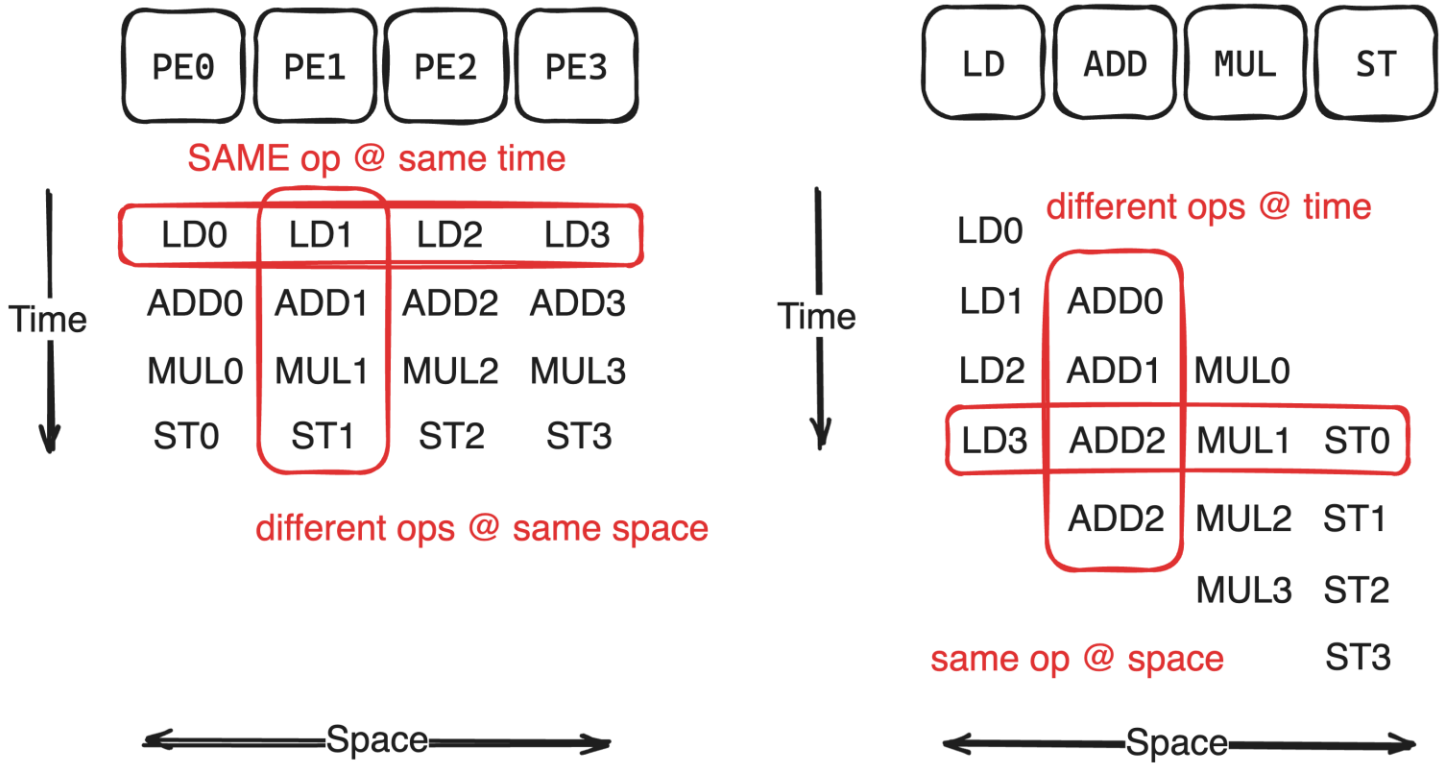
- A GPU is a specialized electronic circuit designed to handle multiple tasks simultaneously in an efficient manner
- It's equipped with many small efficient cores
  - Many in-order cores
- Basically, a SIMD-like machine (Single Instruction Multiple Data) under the hood
- Limits the drawbacks of canonical SIMD
- Fundamental in applications like:
  - Deep learning, Graphics processing, Video processing, Scientific computing

SIMD (Single Instruction Multiple Data)

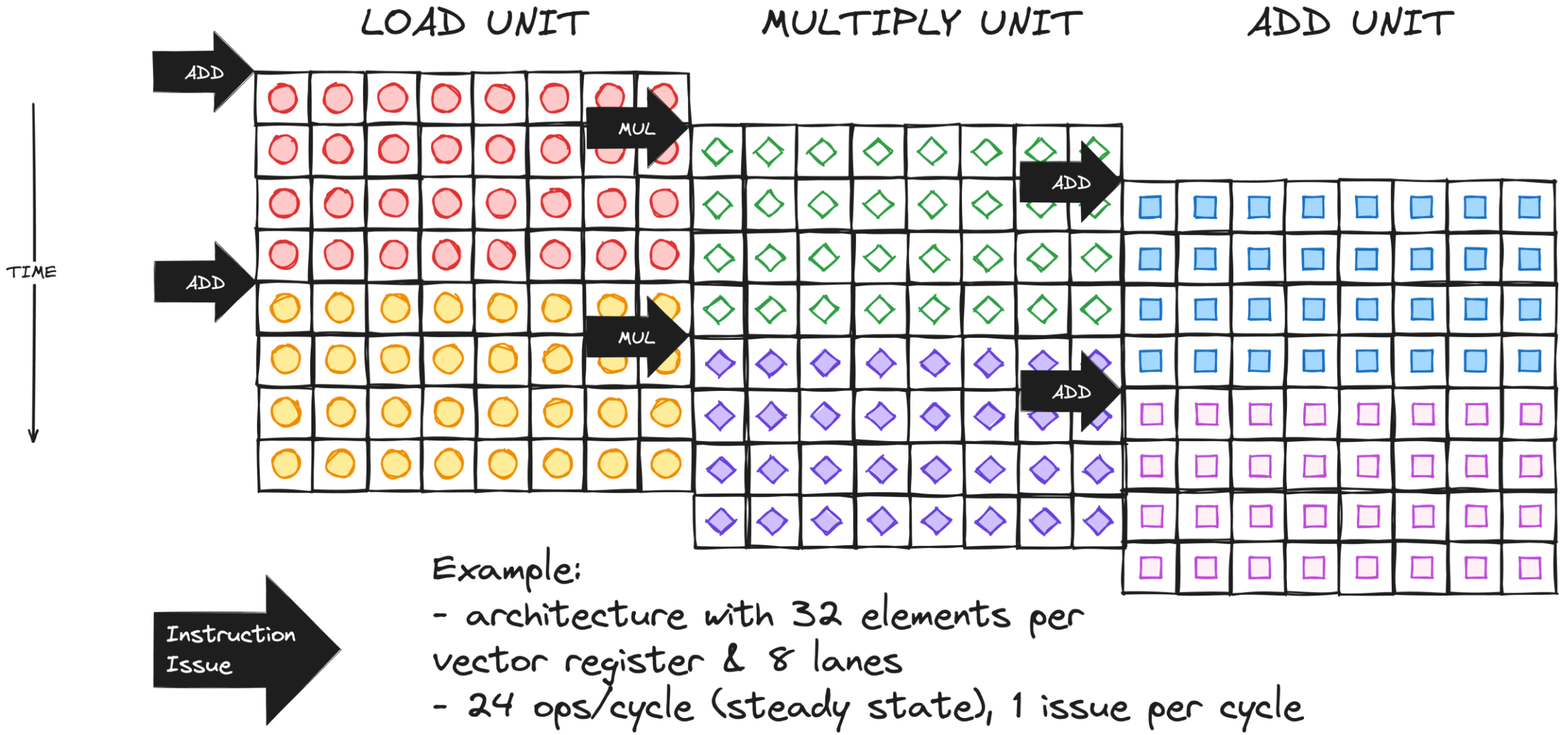
- A type of parallel processing that executes the same operation on multiple data
- Another approach to parallelism
  - We studied ILP (Instruction Level Parallelism)
  - GPUs & SIMD are based on DLP (Data Level Parallelism)
- ILP: Instruction level parallelism (Out-of-order execution for example)
- DLP: same operation across multiple different elements
- ARM NEON, Intel MMX/SSE/AVX, RISC-V P (DSP) Extension
- 2 types of processors
  - Array processors
  - Vector processors
- Data independence is taken for granted -> no need to check data dependencies

Array Processors VS Vector Processors

- **Array processors**: instructions operate on multiple data elements simultaneously using different processing elements
- **Vector processors**: execute one instruction on multiple data elements using a single processing unit
- This abstraction does not really exist..



SIMD


LOAD UNIT
MULTIPLY UNIT
ADD UNIT
ADD
MUL
ADD
ADD
MUL
ADD
TIME
Instruction Issue
Example:
- architecture with 32 elements per vector register & 8 lanes
- 24 ops/cycle (steady state), 1 issue per cycle

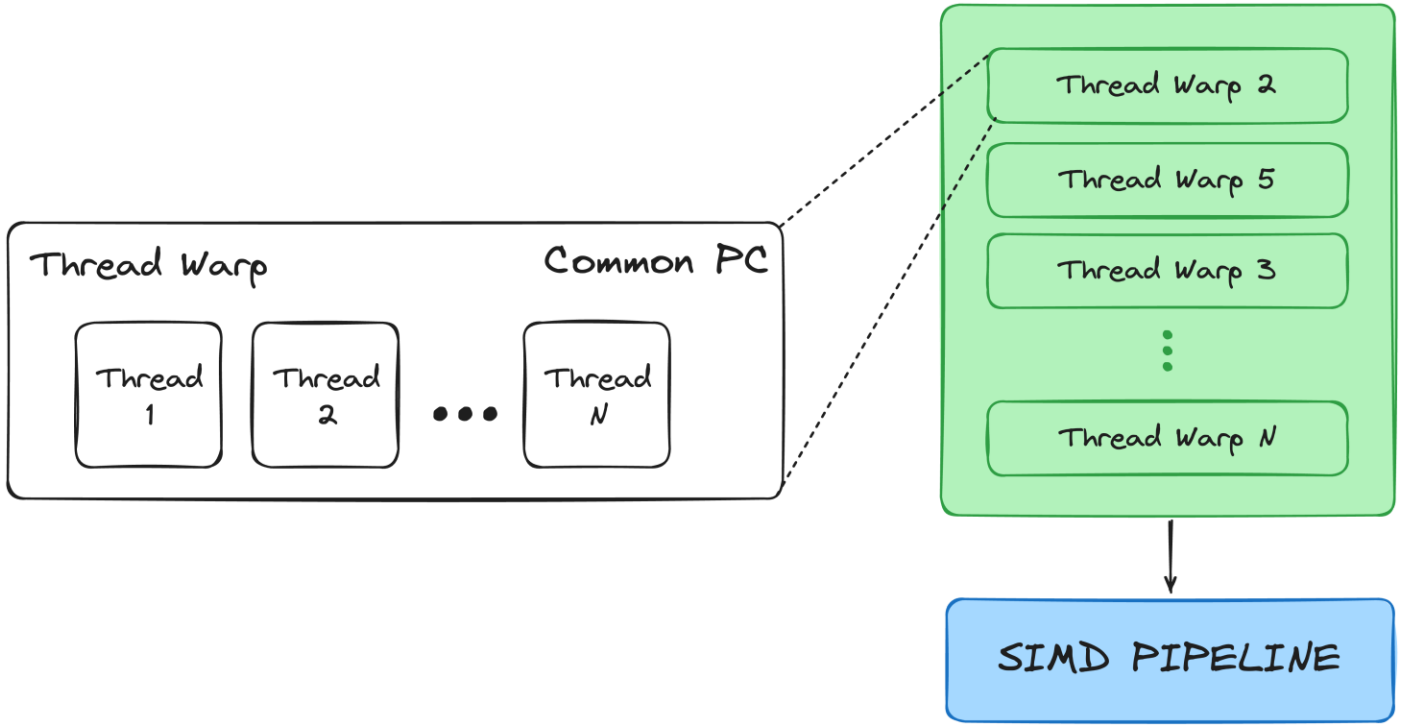
SIMD advantages & limitations

+ Energy efficient, every instruction generates a lot of work = less idle time
+ Vectors data are dependency-free
    + deeper pipelines
+ Less branches, especially for loops
- Difficult to program
    - Require the programmers to re-think the algorithms and to know the specific architecture they operate on
    - Data must be dependency free
    - Highly inefficient if data is not regular

In GPUs

- GPUs are a mix of vector & array processors
- Implement SIMD via SIMT (Single Instruction Multiple Thread)
  - Instructions are run on THREADs
    - Each thread executes the same code on different data
    - Each thread maintains its own context which is independent from others -> no lock-step
      - Threads can be synced at barriers with **__syncthreads()**
  - Threads are grouped together in WARPs (NVIDIA terminology)
    - Typically, in a warp -> 32 threads

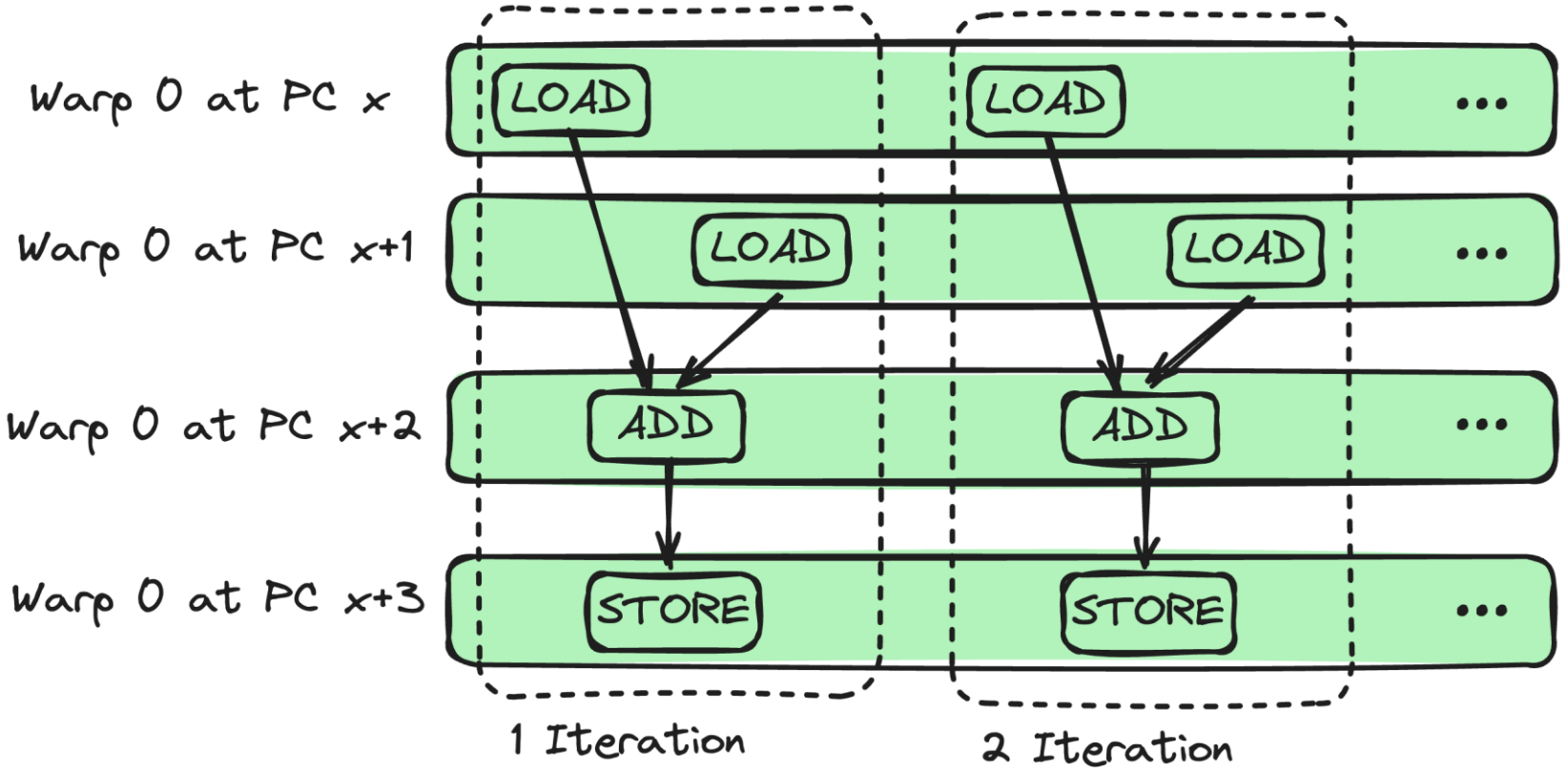
THREADS & WARPS



- Thread -> a single sequence of programmed instructions that operates on an independent set of data
- It is not a physical piece of hardware, it's a logical execution unit that the GPU hardware can manage and run
- Each thread in a warp share the same **program counter (PC)**
- A warp can be seen as a SIMD instruction formed by the hardware

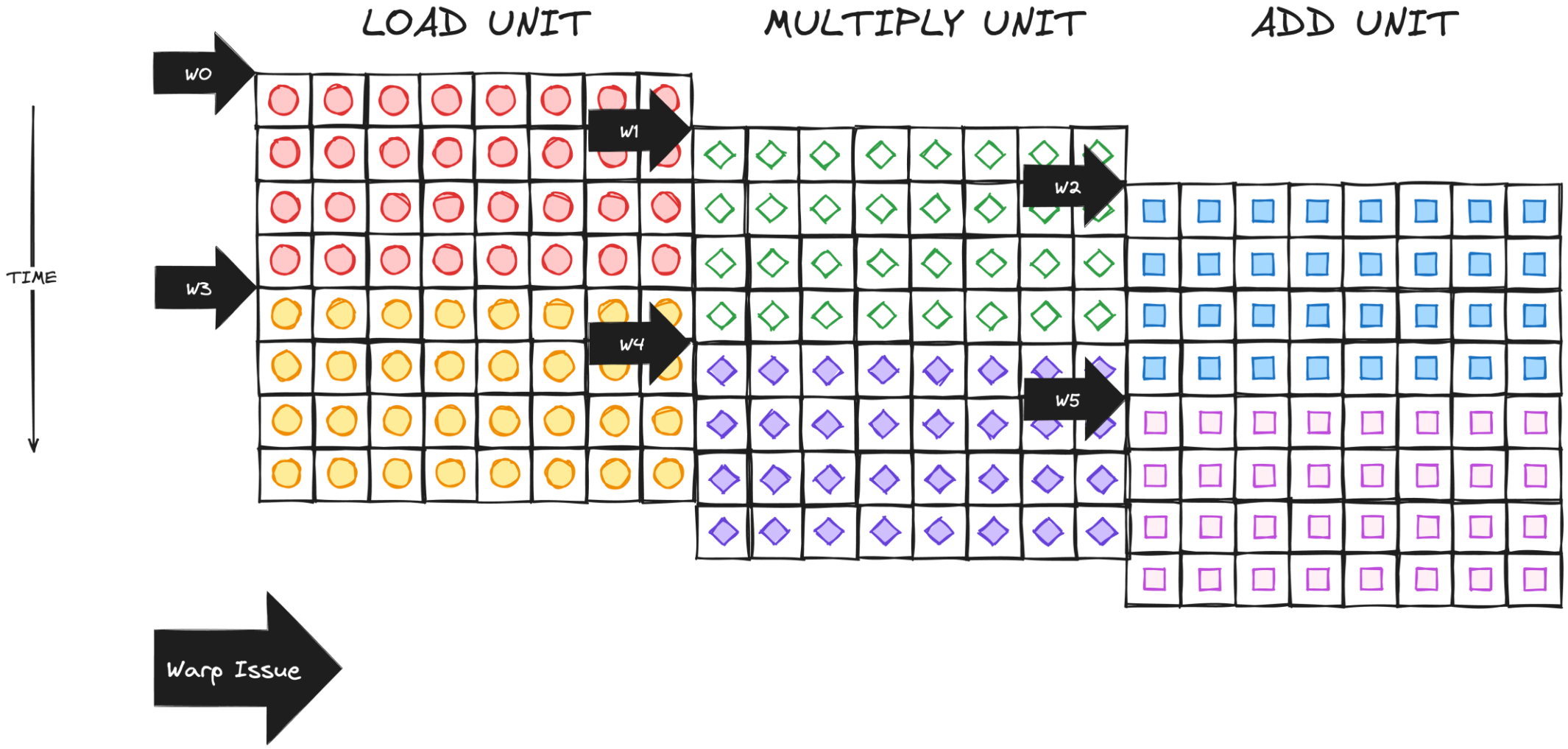
Example

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```



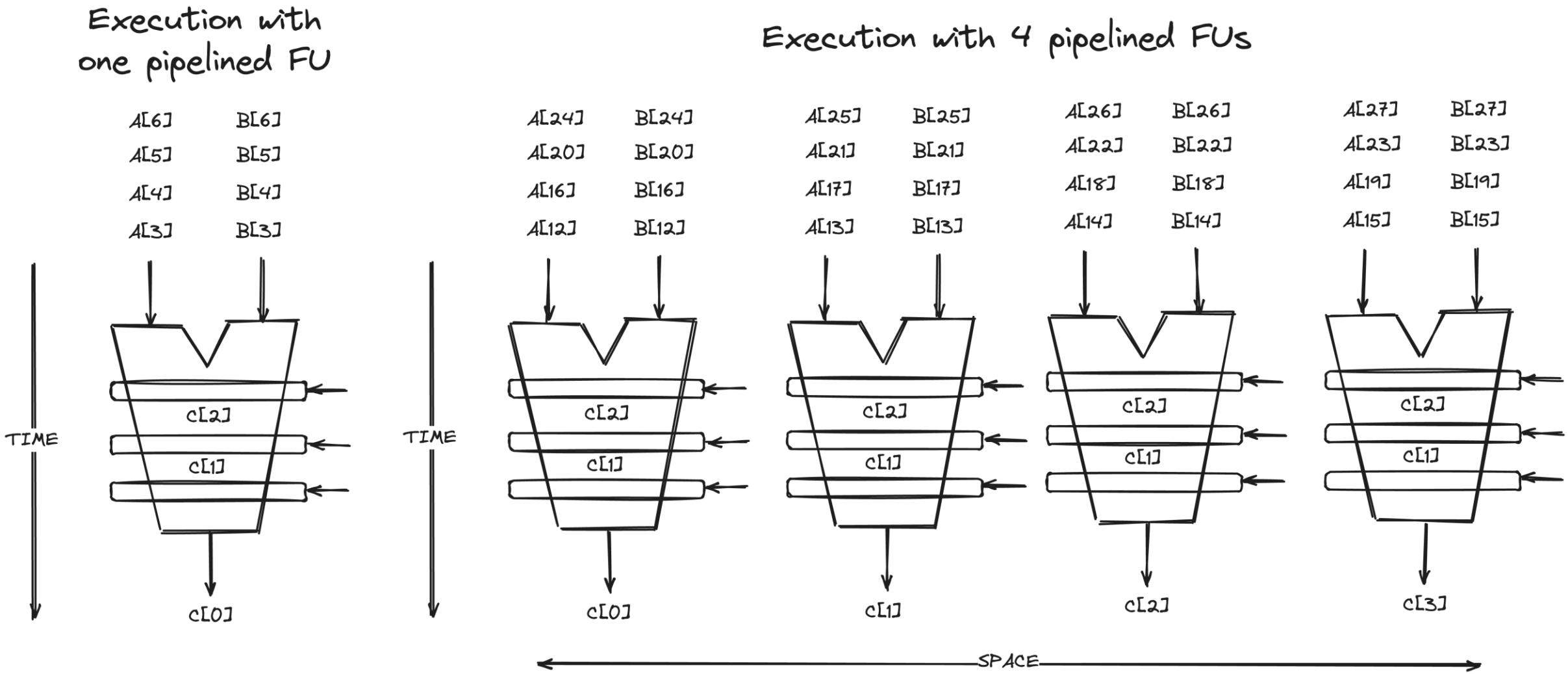
https://github.com/ilcors-dev/unibo-nvidia-cuda-research

Example



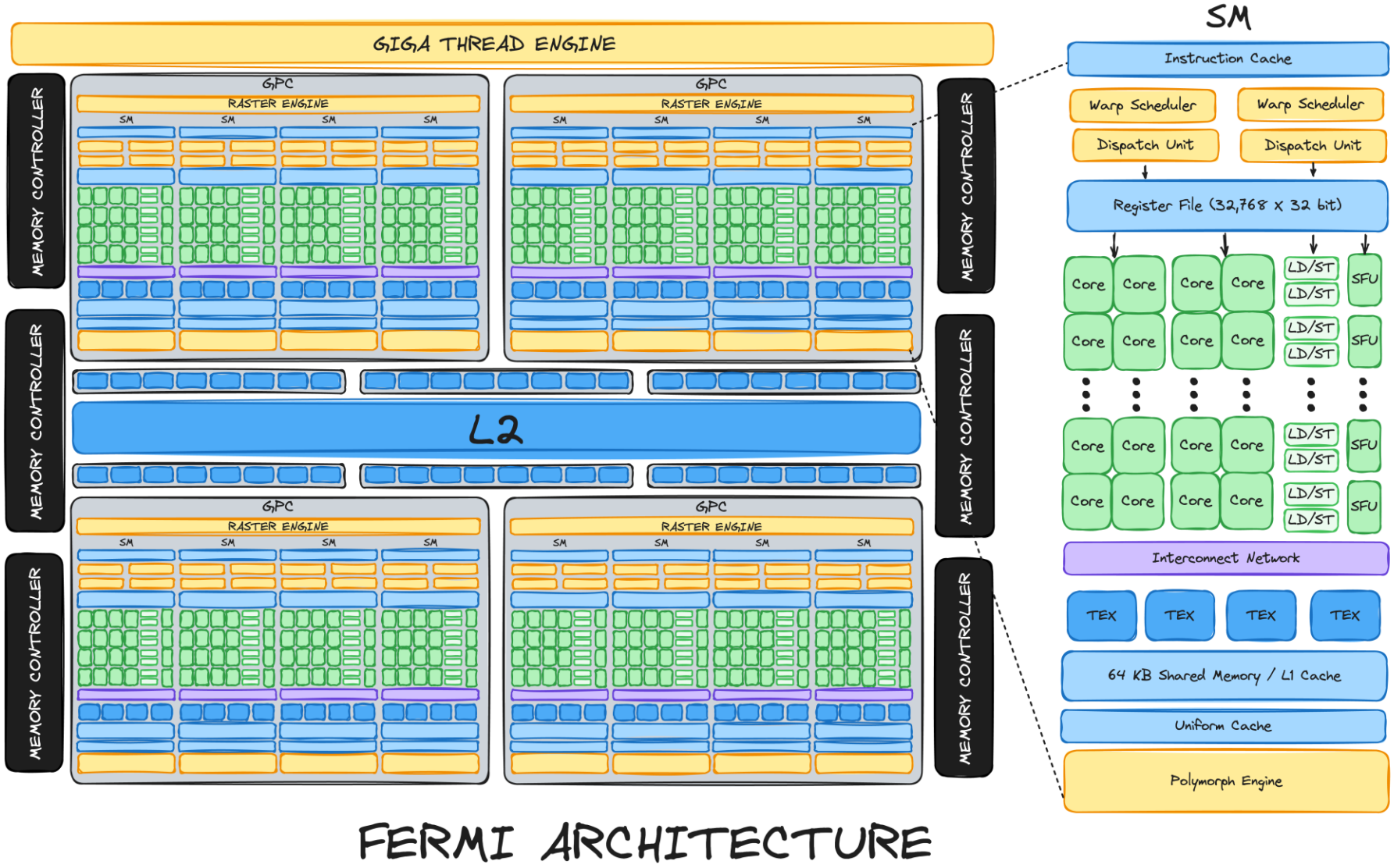
https://github.com/ilcors-dev/unibo-nvidia-cuda-research

32-Thread warp executing ADD A[tid], B[tid] -> C[tid]



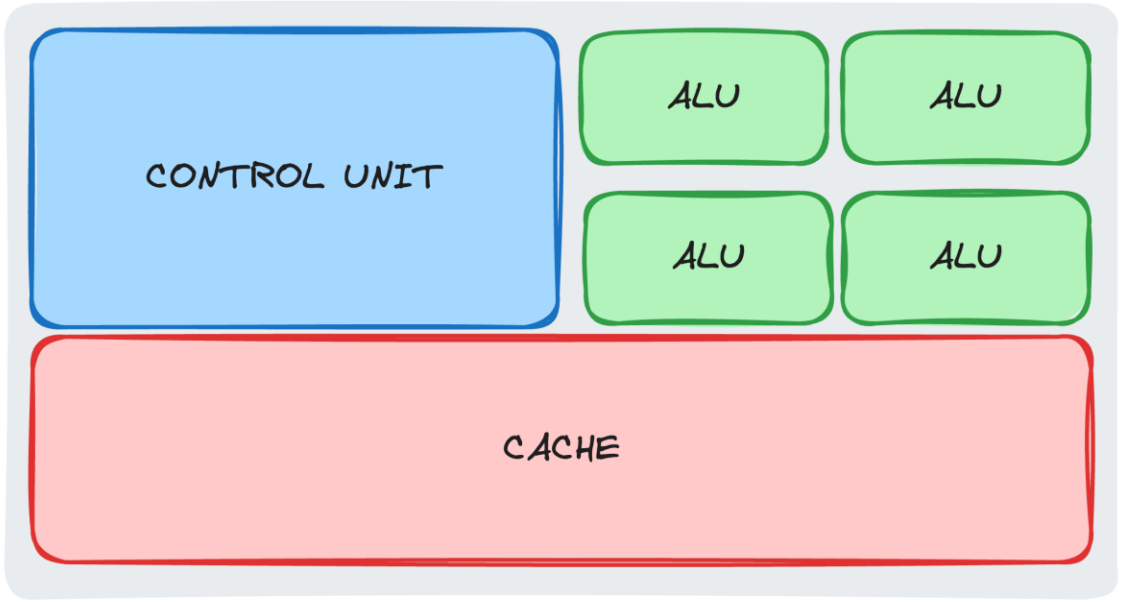
But where do threads run?

- Streaming Multiprocessor (SM)
- 32 cores in each
  - Cuda core or Streaming Processor (SP)
  - 1 integer multiplication per clock
- Register File of 32 KB
- 64 KB L1 Cache
- The Tesla (2006) & Fermi architectures are the foundation of modern Nvidia GPUs

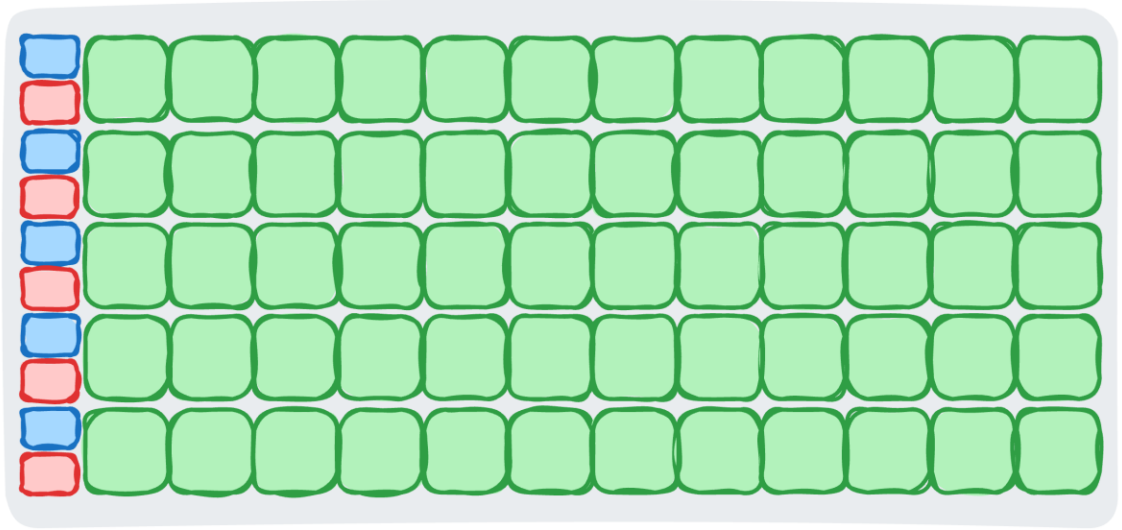


Comparison with CPU

- CPU
  - A few **out-of-order** powerful cores
  - General purpose computing
  - Able to handle dependencies
  - Complex hardware
  - Hide latency with caching & prediction
- GPU
  - Many **in-order** cores
  - Focus on throughput rather than latency
  - Optimized for parallelism
  - Hide latency with FGMT



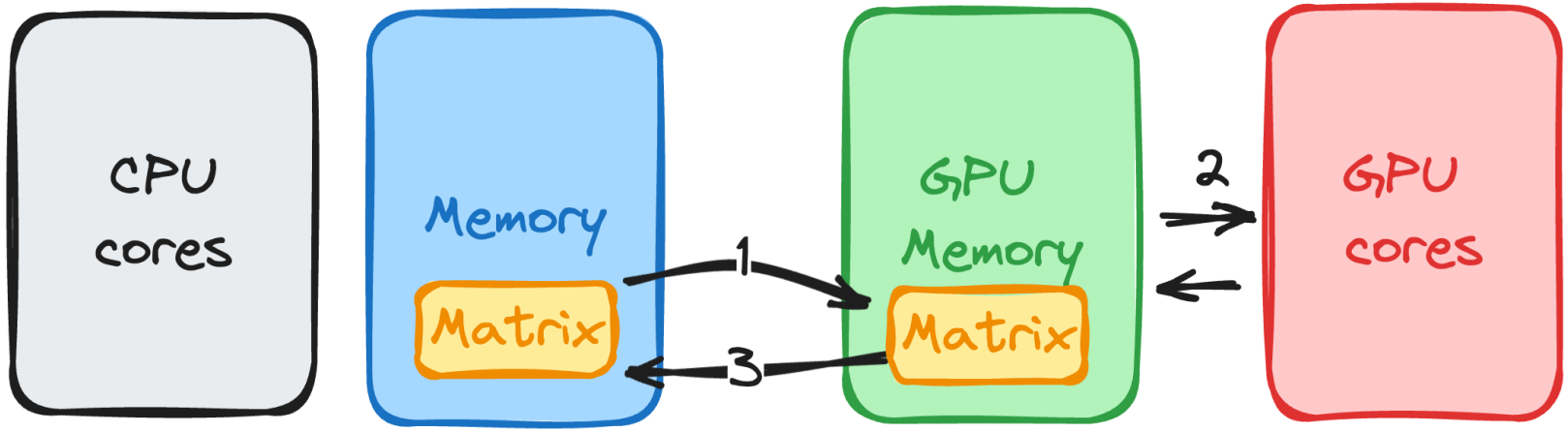
CPU



GPU

GPU programming

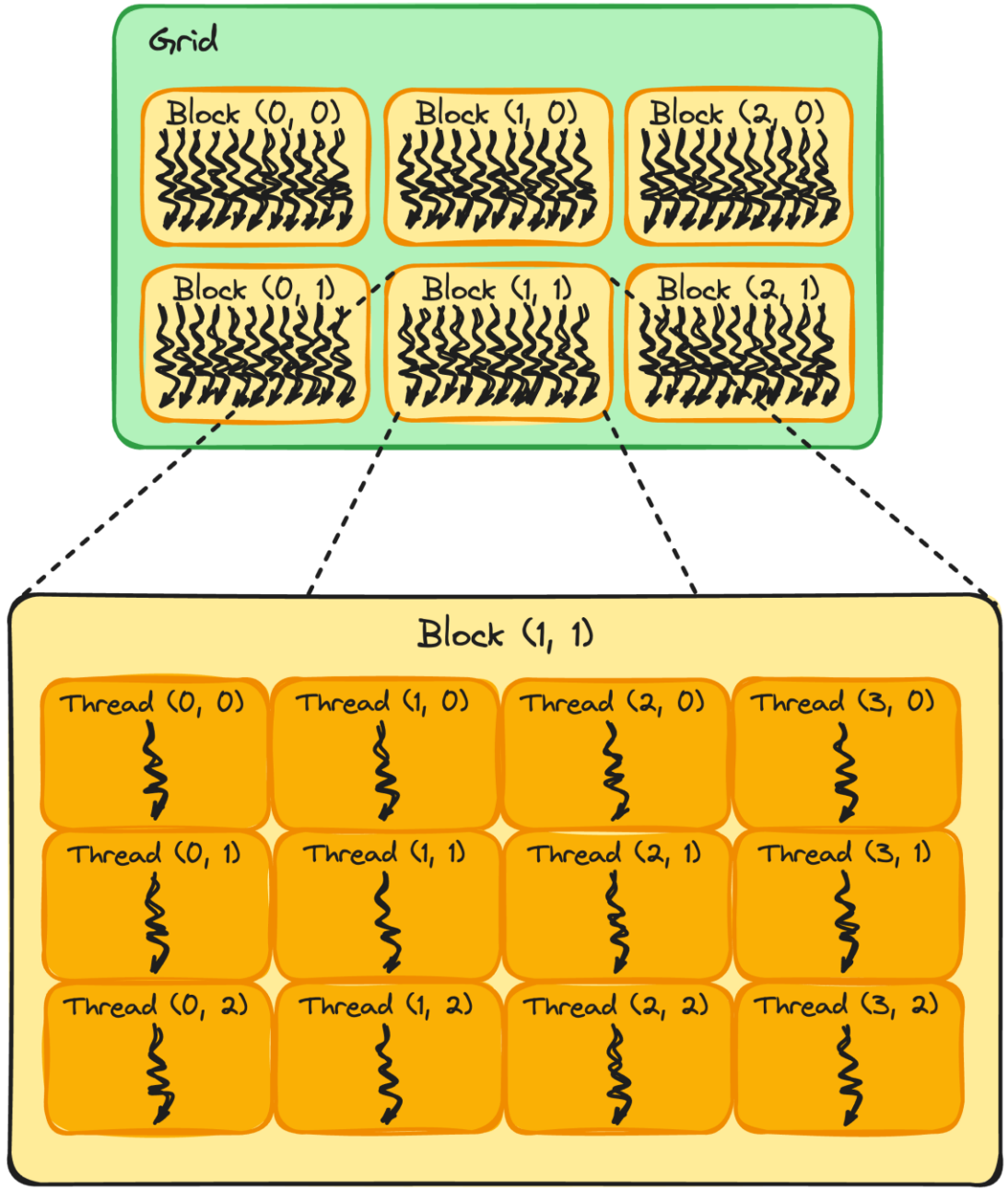
- The CPU is referred as **host**
- The GPU is referred as **device**
- Computation is offloaded to the GPU
- 3 steps:
  - CPU-GPU transfer
    - Latency & bandwidth must be considered
  - GPU kernel execution
    - Performance depends on how well the kernel is optimized for the GPU's architecture
  - GPU-CPU transfer



CUDA programming model

- Fundamental abstractions:
  - **Thread:** smallest execution unit
    - Executed on a CUDA core
  - **Block:** a collection of threads
    - Abstraction of a Streaming Multiprocessor
    - Schedules the execution of its threads
    - Shares a small fast local memory accessible by its threads
  - **Grid:** A collection of blocks that execute a kernel
    - Spreads across multiple SMs
    - Managed by the GPU scheduler

CUDA programming model



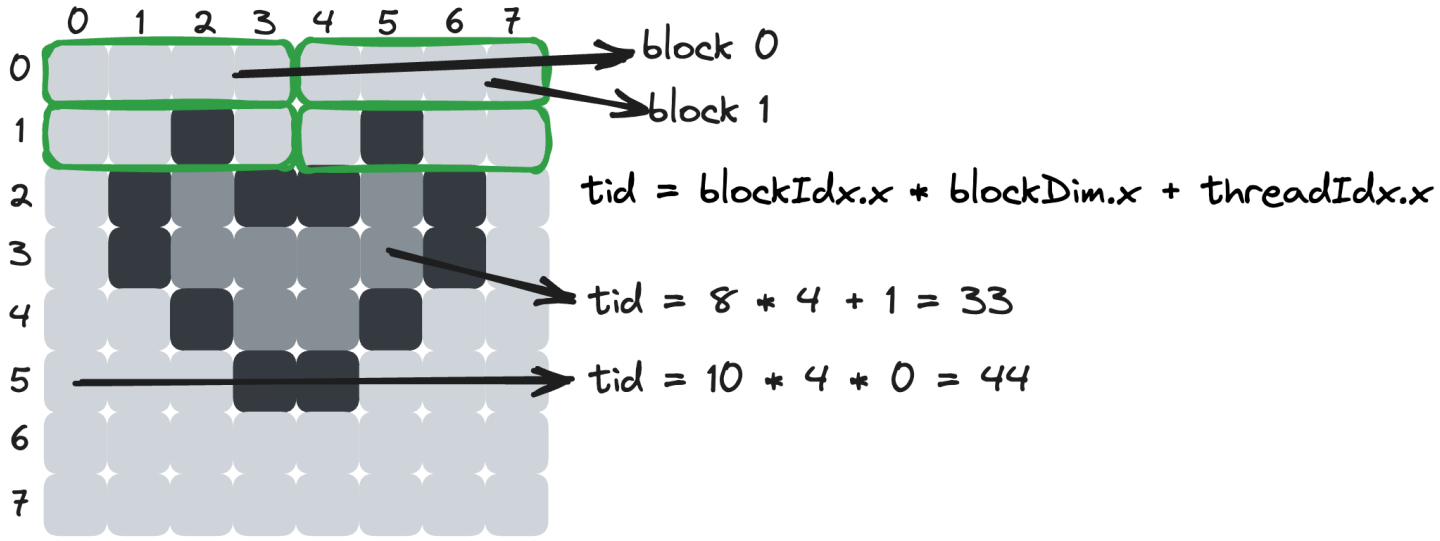
The basics

- A **kernel** is defined using the **__global__** macro
- To allocate memory on the device -> **cudaMalloc(&d_in, bytes)**
- Transfer data from **host to device** -> **cudaMemCpy(d_in, h_in, bytes, cudaMemcpyHostToDevice)**
- To launch a kernel -> **kernel<<<numBlocks, numThreadsPerBlock>>>(d_in, d_out)**
- Transfer data back from **device to host** -> **cudaMemCpy(h_out, d_out, bytes, cudaMemcpyDeviceToHost)**
- To free memory on the device -> **cudaFree(d_in)**
- To synchronize threads -> **cudaDeviceSynchronize()**

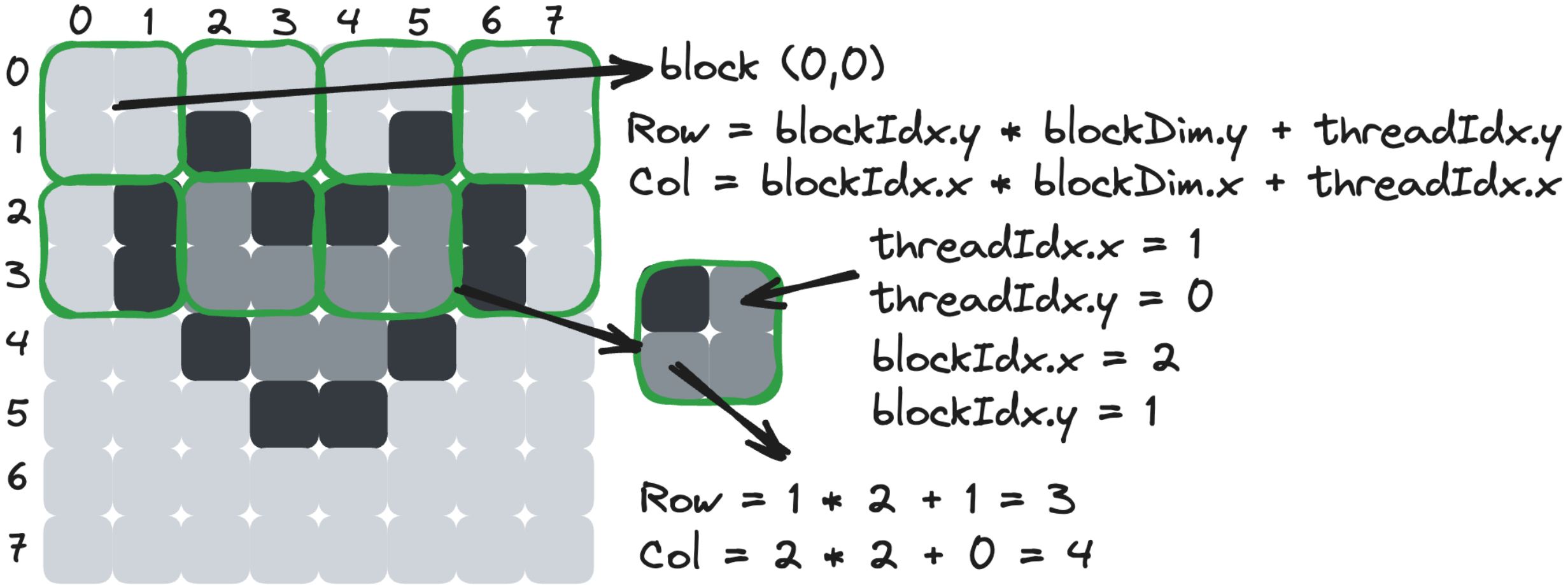
Data layout & Access

- The position of a thread in the global grid depends on how data is layered down
  - 1D, 2D, 3D
  - The programmer can choose it
- **threadIdx** the index of a thread within its block
- **blockIdx** the index of a block within its grid
- **blockDim** how many threads there are in each x, y, z directions
- **gridDim** how many blocks there are in each x, y, z directions

1 dimension (vectors)

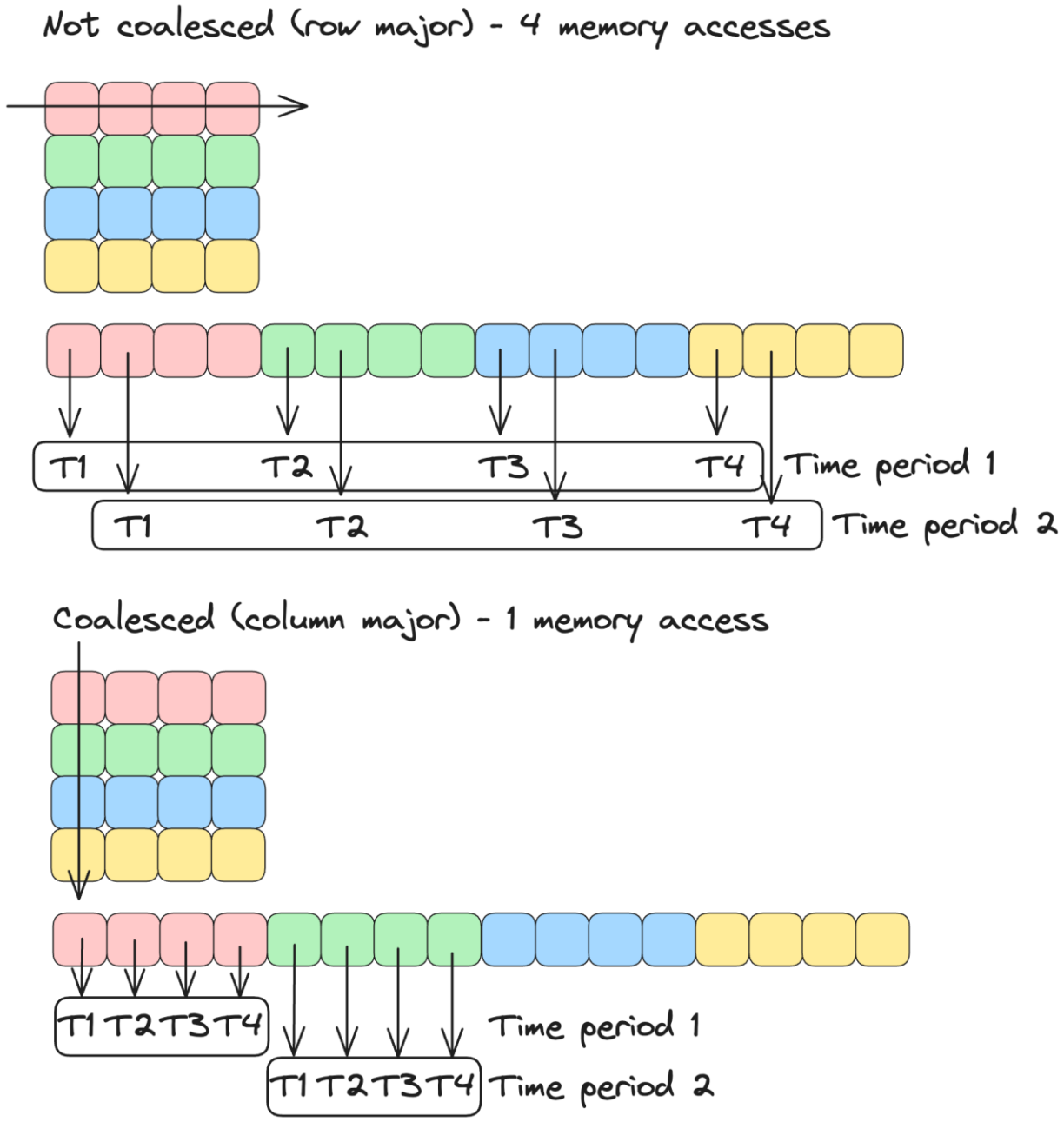


2 dimension (matrixes)



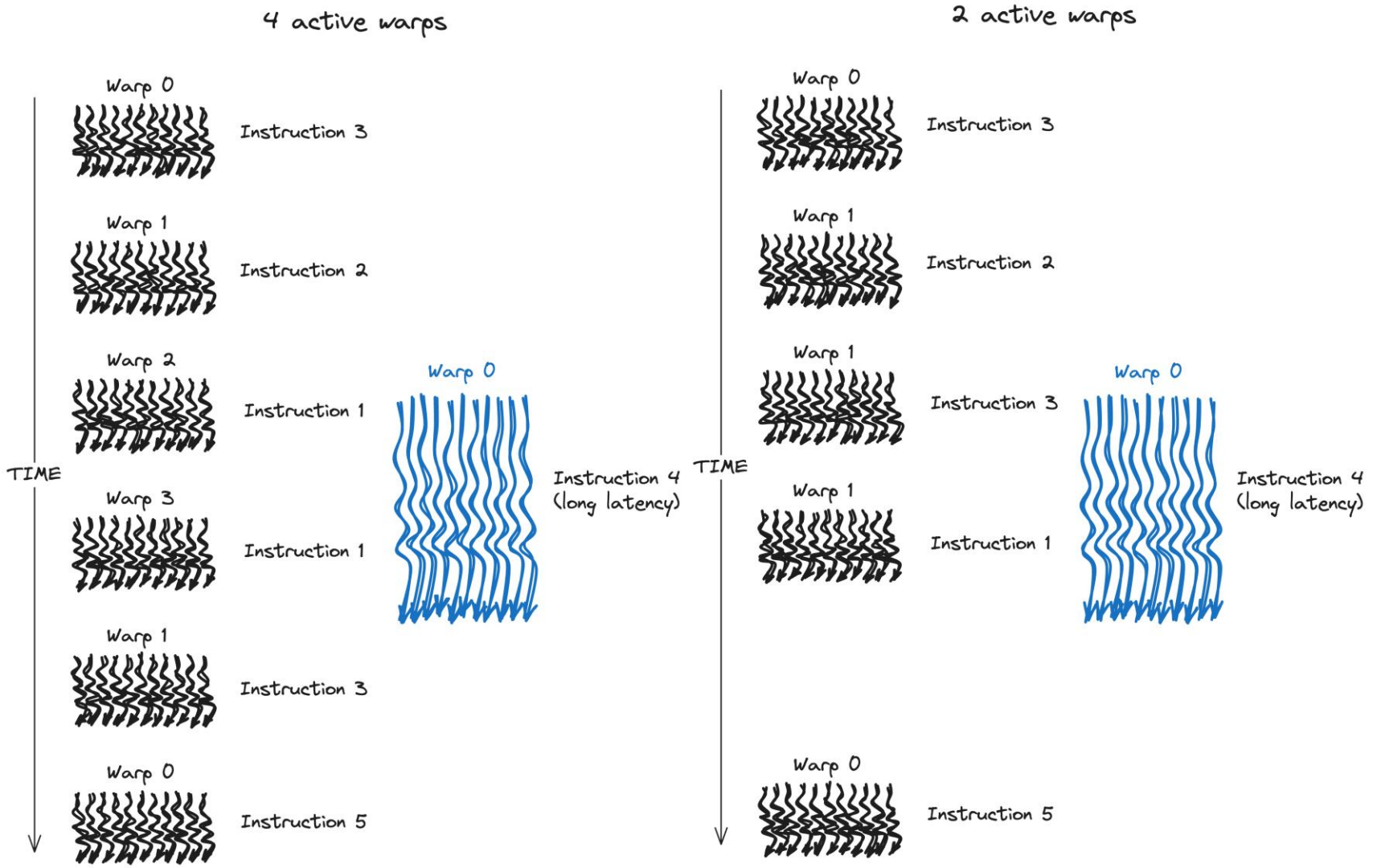
Memory access - Coalescing

- Crucial to improve memory performance & increase bandwidth
- In a CPU we would prefer the row-major layout, to take advantage of data locality
- For example
  - CPUs prefer Array Of Structures (AoS)
  - GPUs prefer Structure Of Array (SoA)



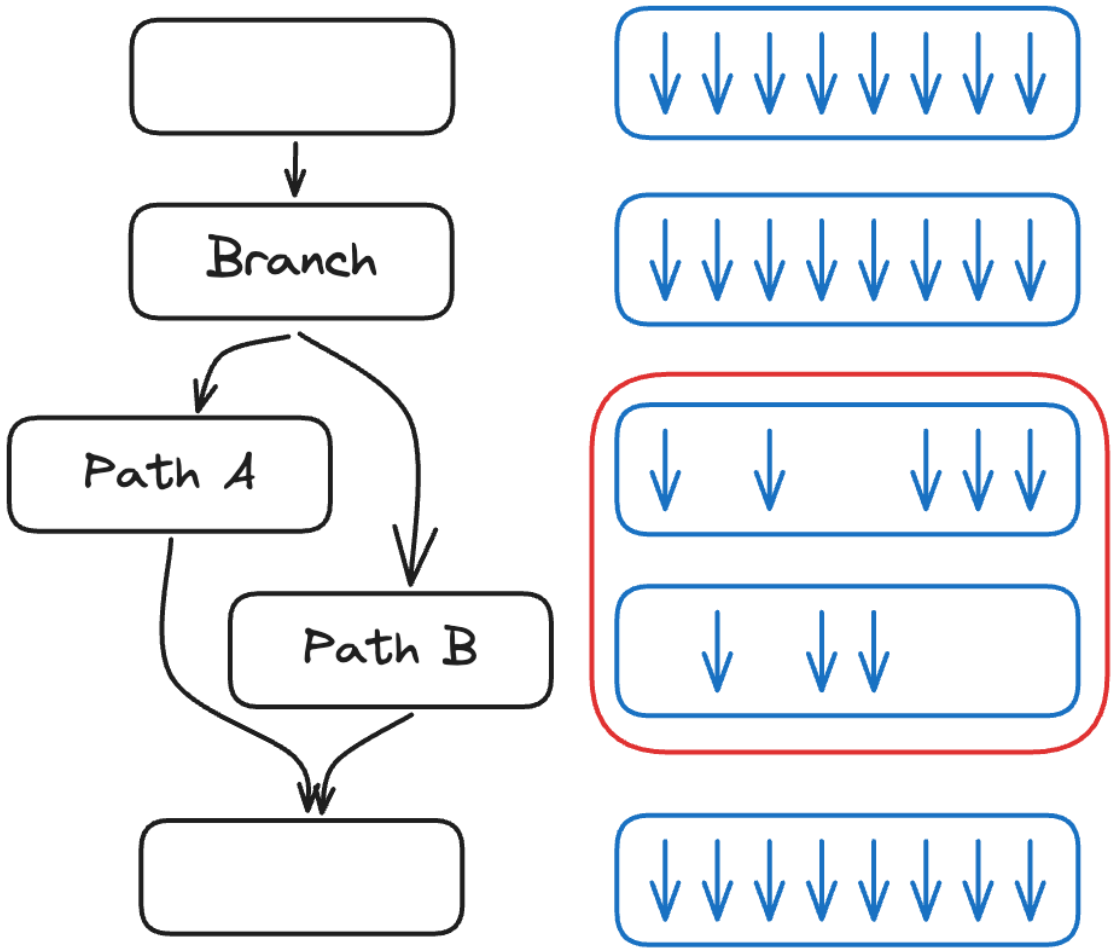
Memory access – latency hiding

- Hides long memory accessing by interleaving the execution of warps
  - Can occur when accessing global memory
- Better occupancy: ratio of active warps / #max per SM



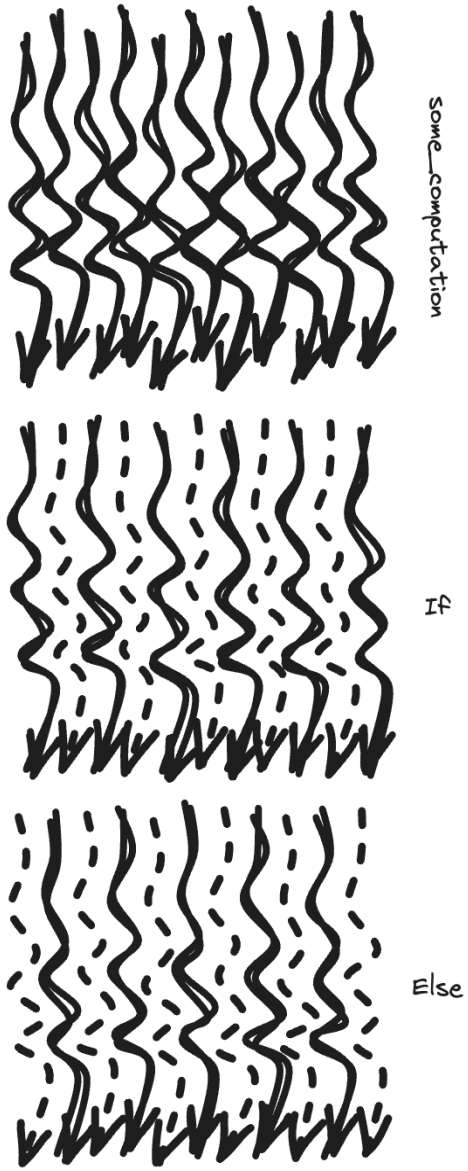
SIMD utilization - Branch divergence

- Branches may lead to lower GPU efficiency
- One of the solutions is to organize the code in a way to
    1. Minimize the branches
    2. Split the condition for threads in different warps, without causing intra-warp divergence



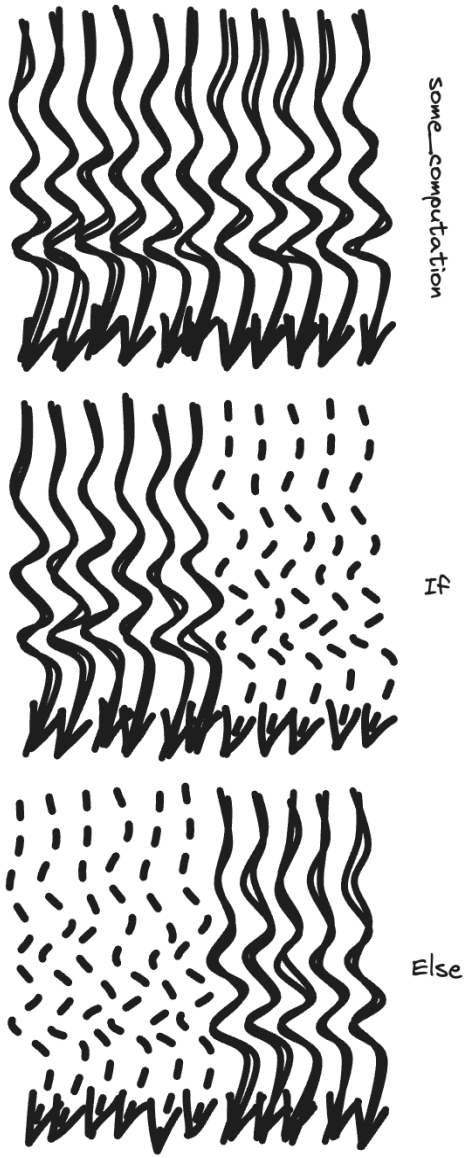
Intra warp divergence

```
some_computation()

if (threadIdx.x % 2 == 0) {
    do_this(threadIdx.x);
} else {
    do_that(threadIdx.x);
}

some_computation()
```



Divergence-free

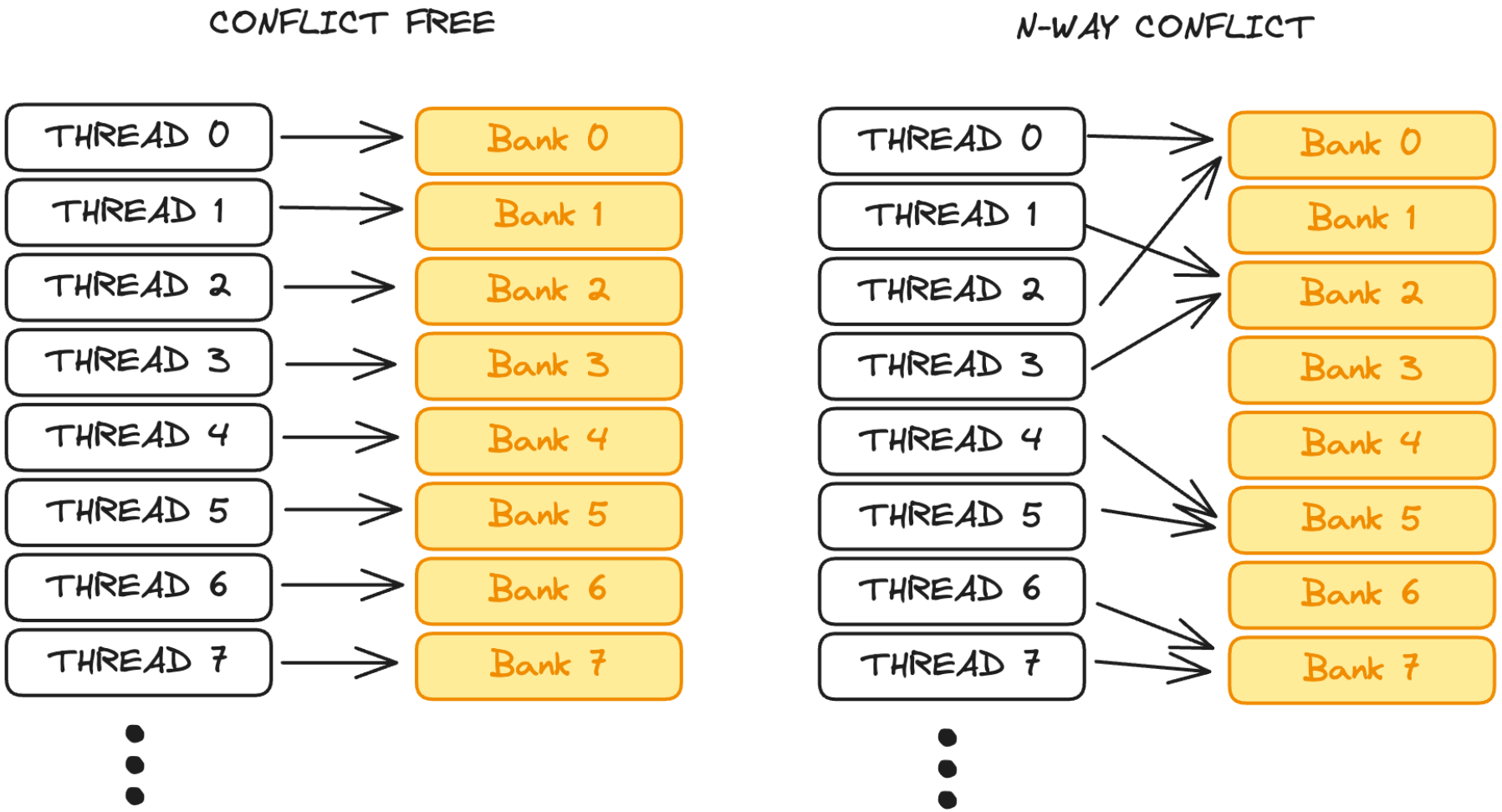
```
some_computation()

if (threadIdx.x < 32) {
    do_this(threadIdx.x);
} else {
    do_that((threadIdx.x));
}

some_computation()
```



Shared Memory

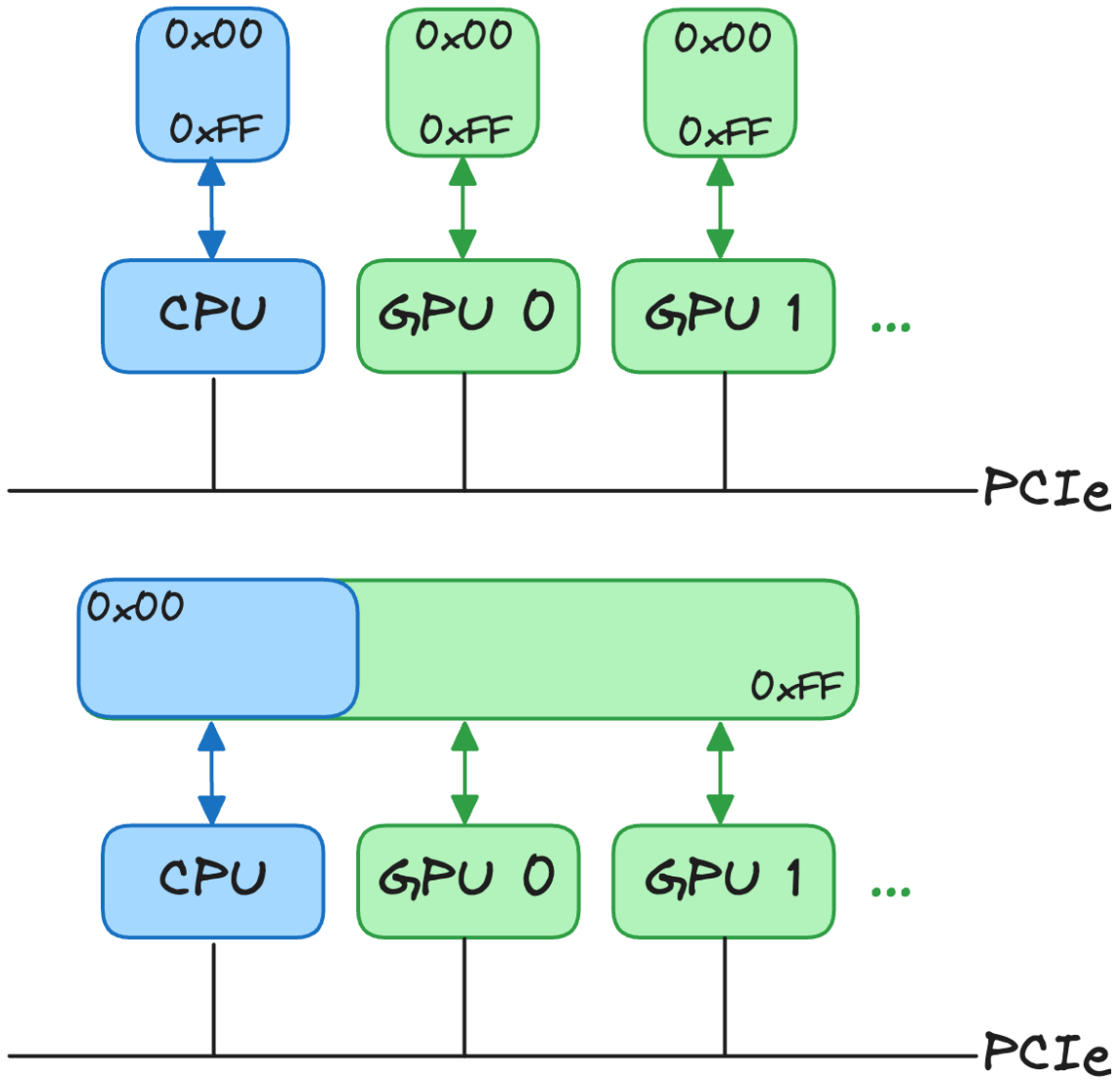
- It's a local cache shared between threads of the same block
- Persists only until within the block's execution duration
- Typically organized into multiple banks to optimize the bandwidth
  - Each bank can service a memory request in one clock cycle, in absence of conflicts
- Conflicts can be
  - 2 way, where 2 threads access the same bank
  - N-way, where N threads access the same bank
- To avoid them, some techniques are used:
  - Padding
  - Randomized mapping
  - Hashing
- Not easy to optimize as it requires deep knowledge of the architecture

Shared Memory



Unified memory

- Introduced in CUDA 6 ~2013
- Unified memory is a managed memory model in which both the CPU & the GPUs share the same memory space
  - Not to be confused with shared memory
  - Simplifies the code
  - Simplifies the data transfer
- **cudaMallocManaged()**



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

An example

- The following code performs a Single Precision A*X+Y operation
- **__global__** specifies that the code will be run on the GPU
- **int i = blockIdx.x * blockDim.x + threadIdx.x;** calculate the index of the thread withing the block & grid
- **if (i < n)** prevents out-of-bound error that may occur

The host code

- Allocates memory on the GPU
- Copies the data into the GPU memory
- Sets the CUDA kernel
  - Number of threads per block
  - Number of blocks to cover every data point
- Executes the kernels
- Once finished, the result is copied back to the host memory
- No use of unified memory

```
int main() {
    int n = 500000000; // Smaller for testing
    float a = 2.0;

    std::vector<float> x(n, 1.0);
    std::vector<float> y(n, 1.0);

    float* x_dev, * y_dev;
    // Allocate the required memory on the GPU
    cudaMalloc(&x_dev, n * sizeof(float));
    cudaMalloc(&y_dev, n * sizeof(float));

    // Transfer the data from the CPU to the GPU
    cudaMemcpy(x_dev, x.data(), n * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(y_dev, y.data(), n * sizeof(float), cudaMemcpyHostToDevice);
    // How many threads there may be in a block
    int blockSize = 256;
    // How many blocks are necessary to cover all the datapoints
    int numBlocks = (n + blockSize - 1) / blockSize;
    // Launch the kernel
    saxpy<<<numBlocks, blockSize>>>(n, a, x_dev, y_dev);
    // Wait for every thread to finish
    cudaDeviceSynchronize();
    // Copy the result back to the host (CPU)
    cudaMemcpy(y.data(), y_dev, n * sizeof(float), cudaMemcpyDeviceToHost);

    cudaFree(x_dev);
    cudaFree(y_dev);
}
```

Compiling a CUDA program

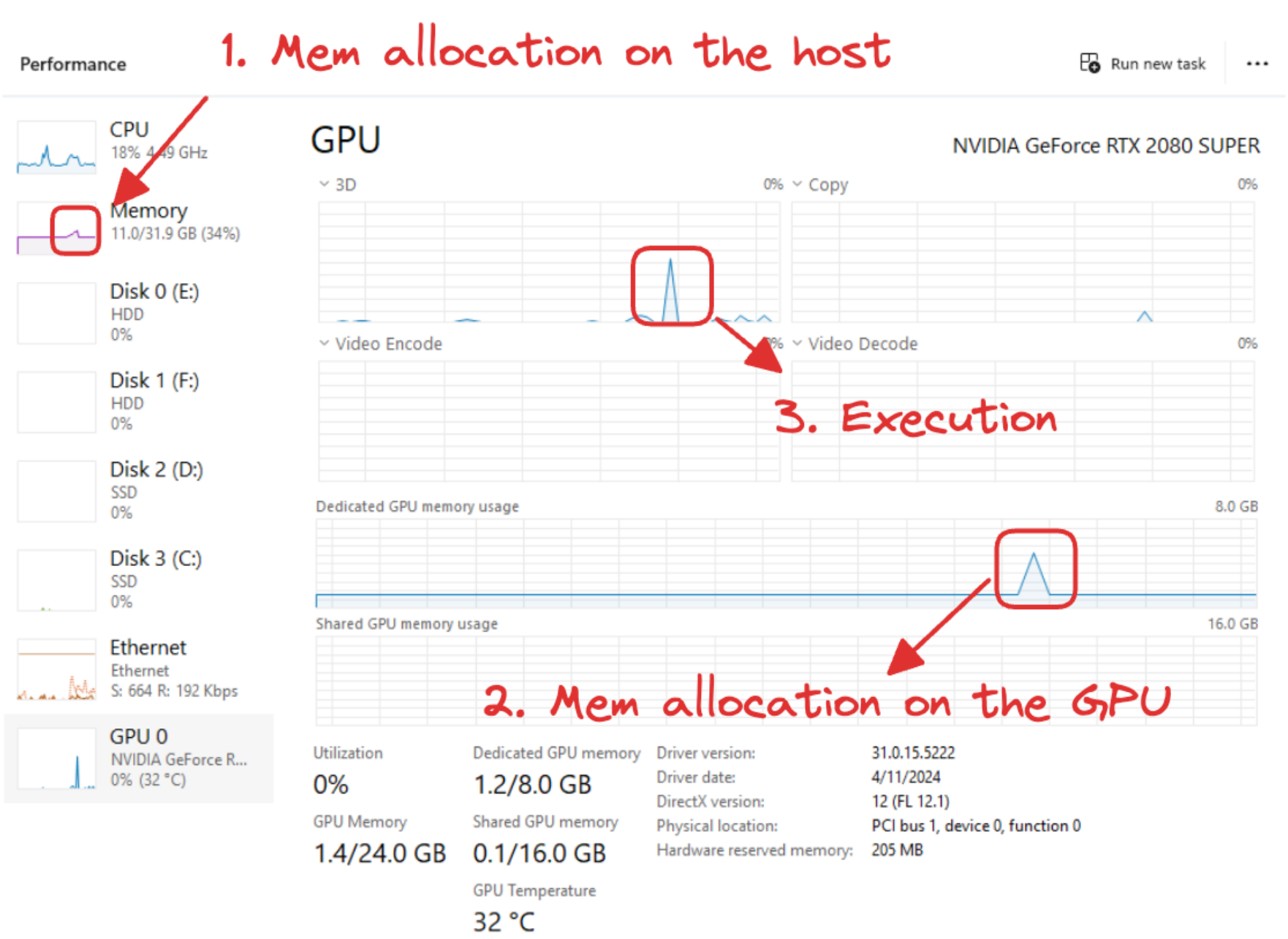
- Everything starts from a .cu file, which contains c++ like code with both code for the CPU & the GPU
- Compilation is done via NVCC (Nvidia CUDA compiler)
  - The .cu gets compiled down to PTX
  - The PTX gets compiled by the GPU driver down to SASS assembly language
- The PTX is an intermediate virtual machine code compatible with multiple GPU's architecture
- SASS is the core assembly language specific to a certain GPU architecture

Compilation

- TODO, insert the PTX code

Execution

```
C:\Users\luca\Desktop\NetworkShared\cuda - Copy>gpu_nounifiedmem.exe
for 500000000 elements, 1953125 blocks are needed
15ms
```



https://github.com/ilcors-dev/unibo-nvidia-cuda-research

A comparison

- CPU-equivalent code
- Much slower, as expected

```cpp
void saxpy(int n, float a, std::vector<float>& x, std::vector<float>& y) {
    for (int i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}

int main() {
    int n = 500000000;
    float a = 2.0;

    // Vectors x and y
    std::vector<float> x(n, 1.0); // Initialize x with 1.0
    std::vector<float> y(n, 2.0); // Initialize y with 2.0

    // Perform SAXPY operation
    saxpy(n, a, x, y);
}
```

```
C:\Users\luca\Desktop\NetworkShared\cuda - Copy>cpu.exe
2711ms
```

Conclusion

+ CUDA provides an API to execute code on NVIDIA GPUs

+ Programmers still code in with the same programming model

+ The GPU allows massive throughput exploiting the SIMD execution model

- Not suited for all types of computations
- Programmers need to understand:
  - GPUs architecture
  - know the CUDA specs
  - memory management