

Nvidia CUDA



NVIDIA.[®]

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
- GPGPU

GPU – Graphics Processing Unit

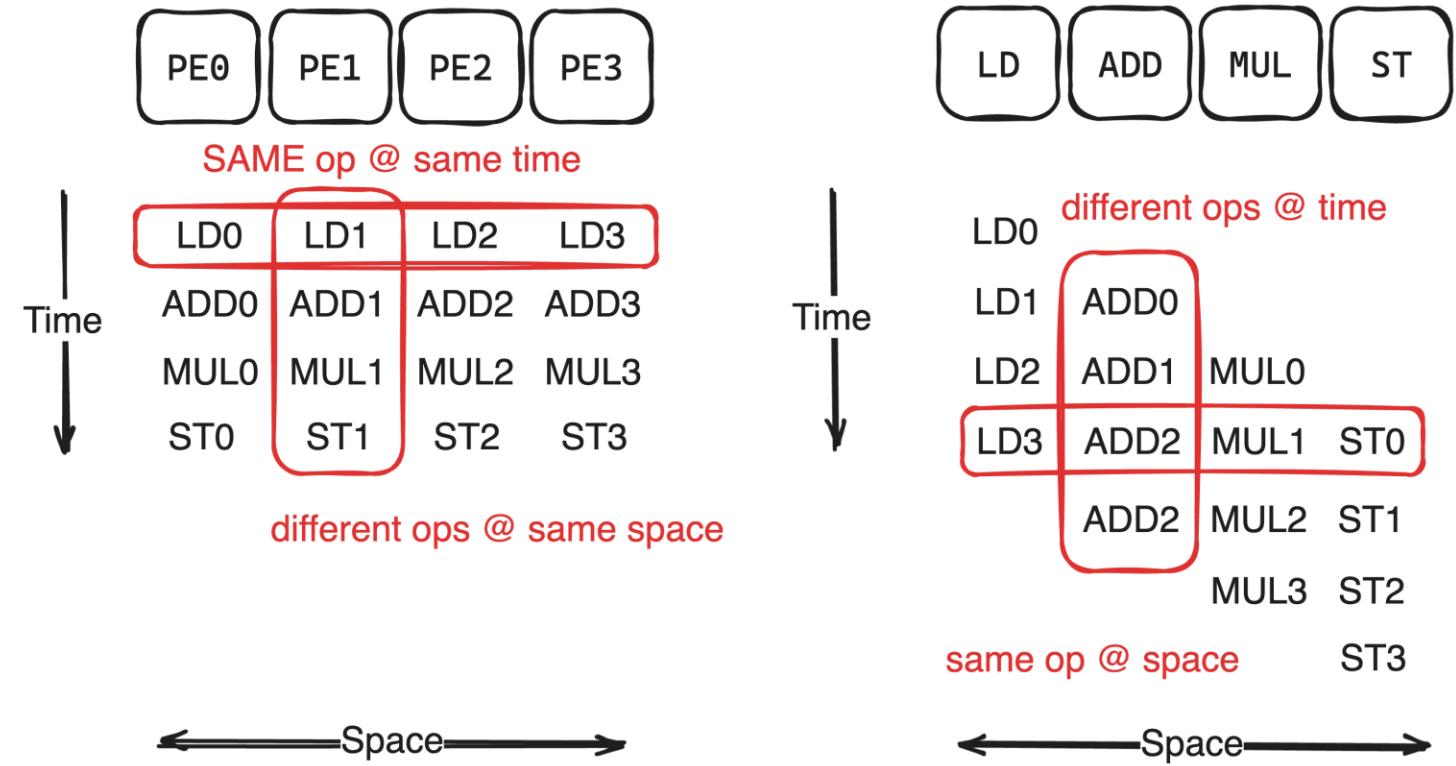
- A GPU is a specialized electronic circuit designed to handle multiple tasks simultaneously in an efficient manner
- It's equipped with many small efficient cores
 - o Many in-order cores
- Basically, a SIMD-like machine (Single Instruction Multiple Data) under the hood
- Limits the drawbacks of canonical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

SIMD (Single Instruction Multiple Data)

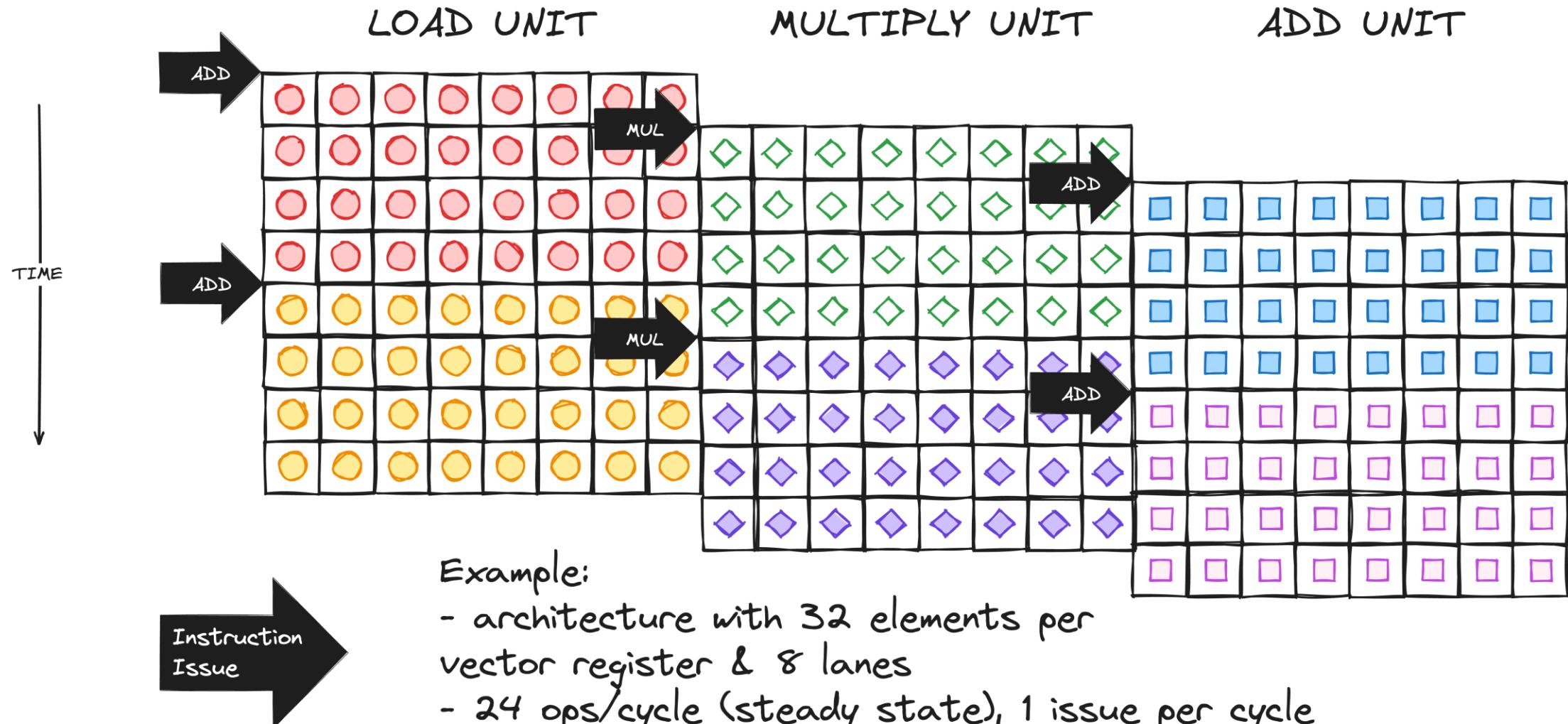
- A type of parallel processing that executes the same operation on multiple data
- Another approach to parallelism
 - o We studied ILP (Instruction Level Parallelism)
 - o GPUs & SIMD are based on DLP (Data Level Parallelism)
- ILP: Instruction level parallelism (Out-of-order execution for example)
- DLP: same operation across multiple different elements
- ARM NEON, Intel MMX/SSE/AVX, RISC-V P (DSP) Extension
- 2 types of processors
 - o Array processors
 - o 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



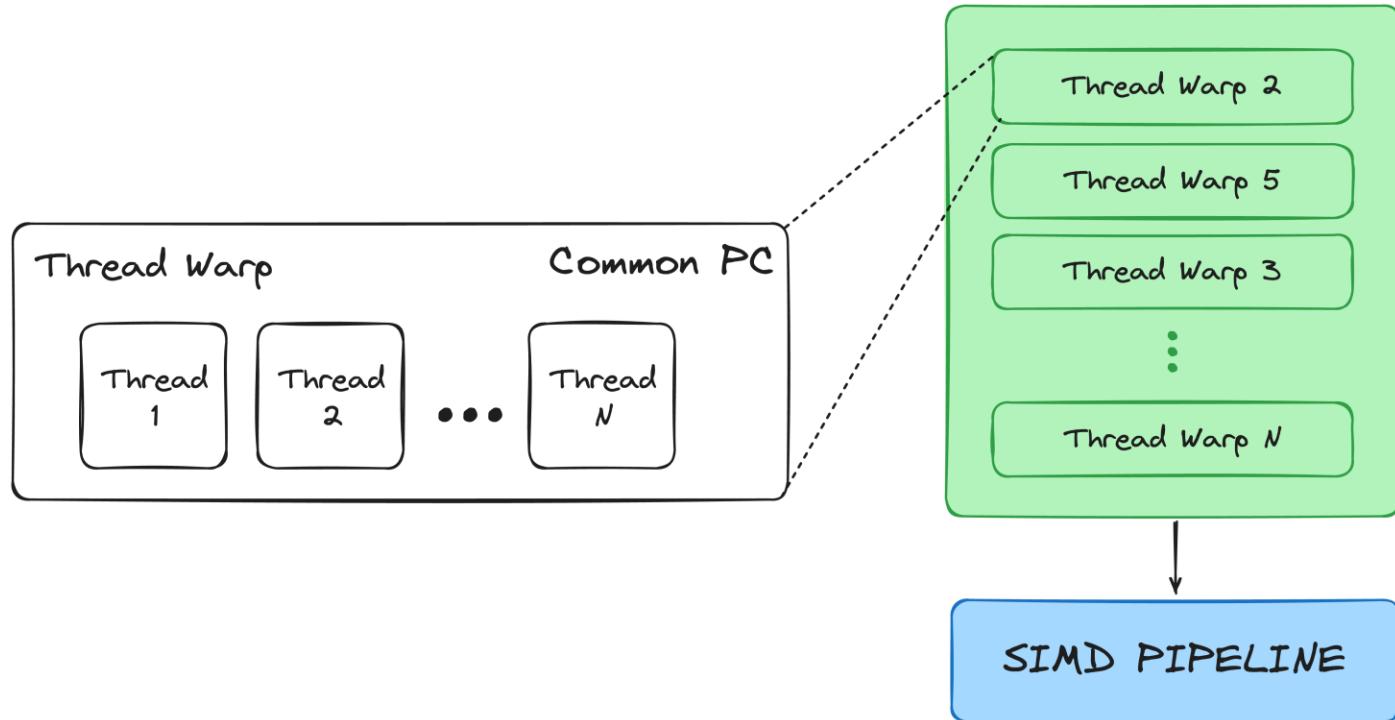
SIMD advantages & limitations

- + 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

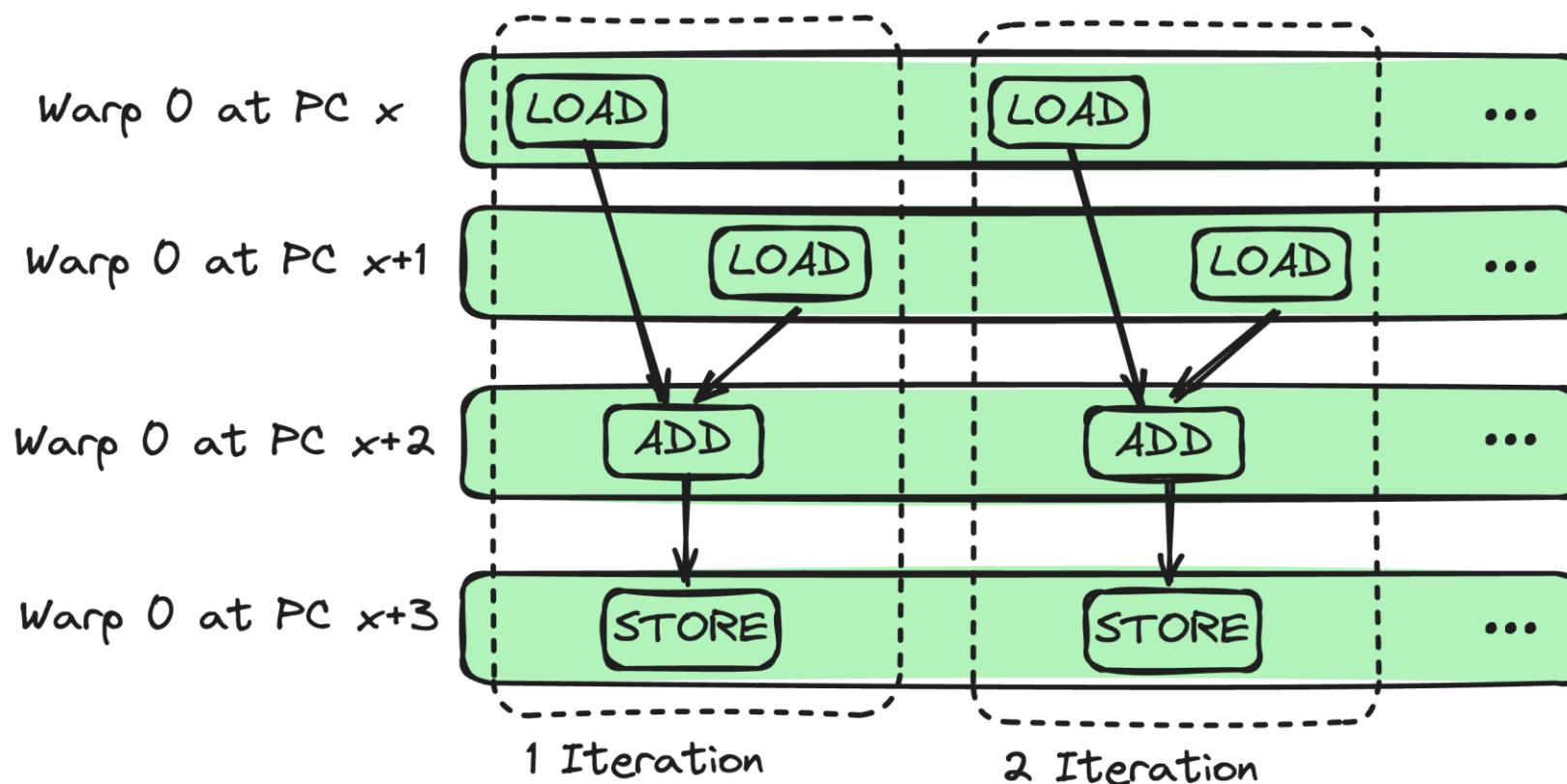
THREADS & WARPS



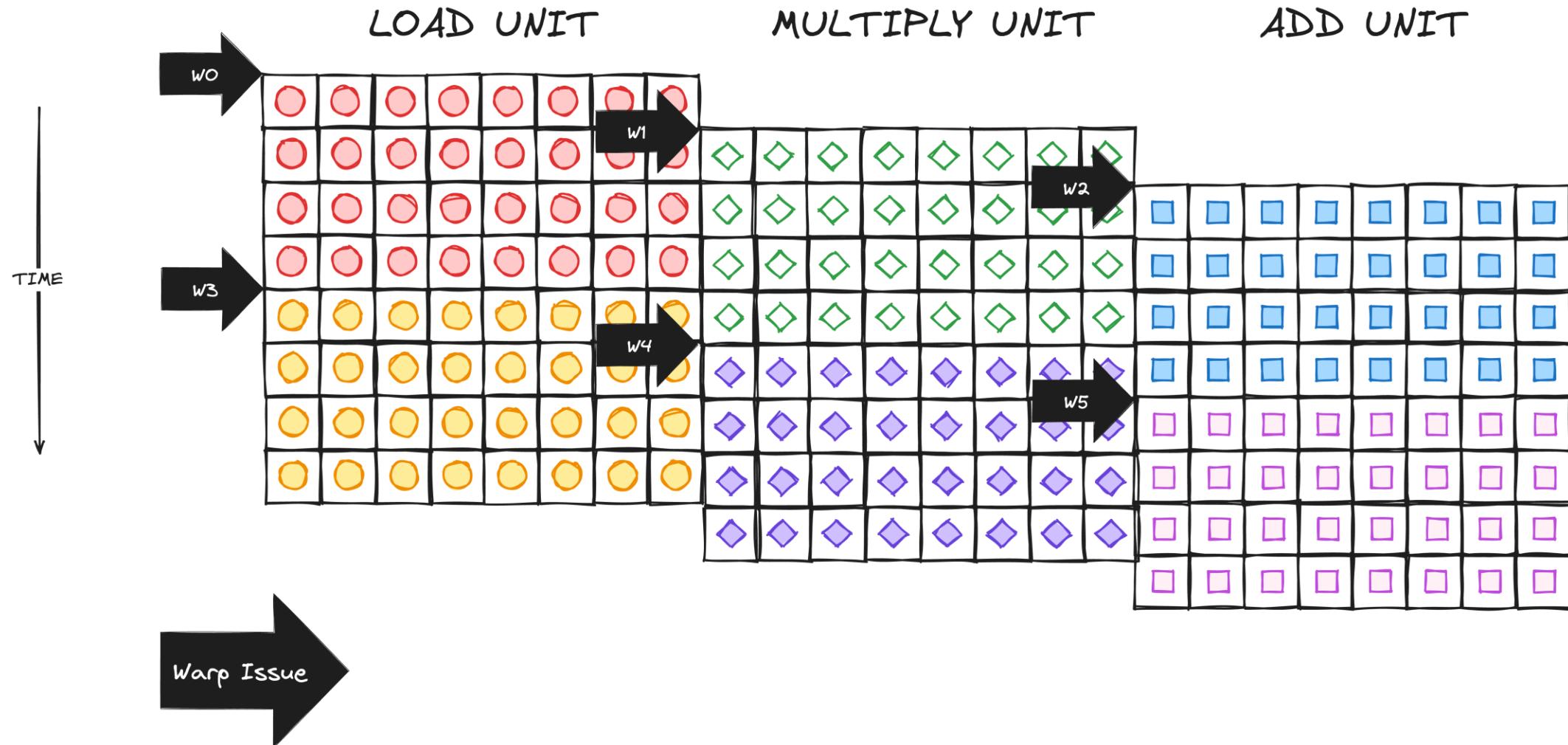
- Thread -> a single sequence of programmed instructions that operates on an independent set of data
- It is not a physical piece of hardware, it's a logical execution unit that the GPU hardware can manage and run
- Each thread in a warp share the same **program counter (PC)**
- A warp can be seen as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

Example

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

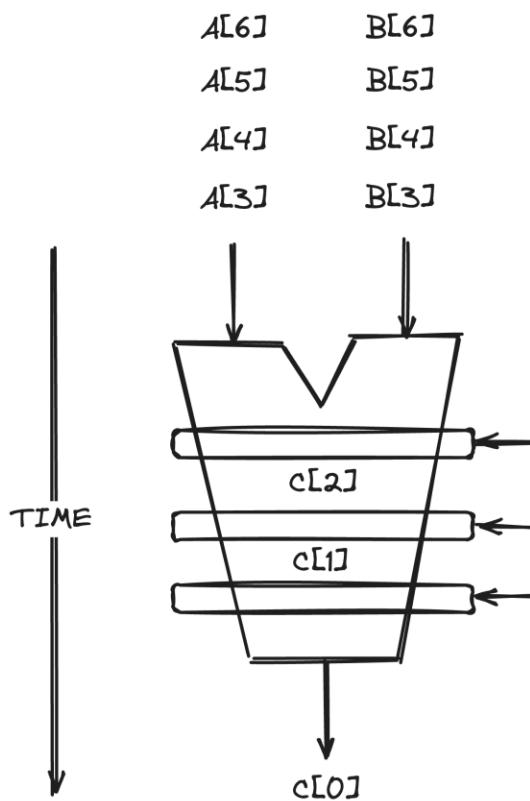


Example

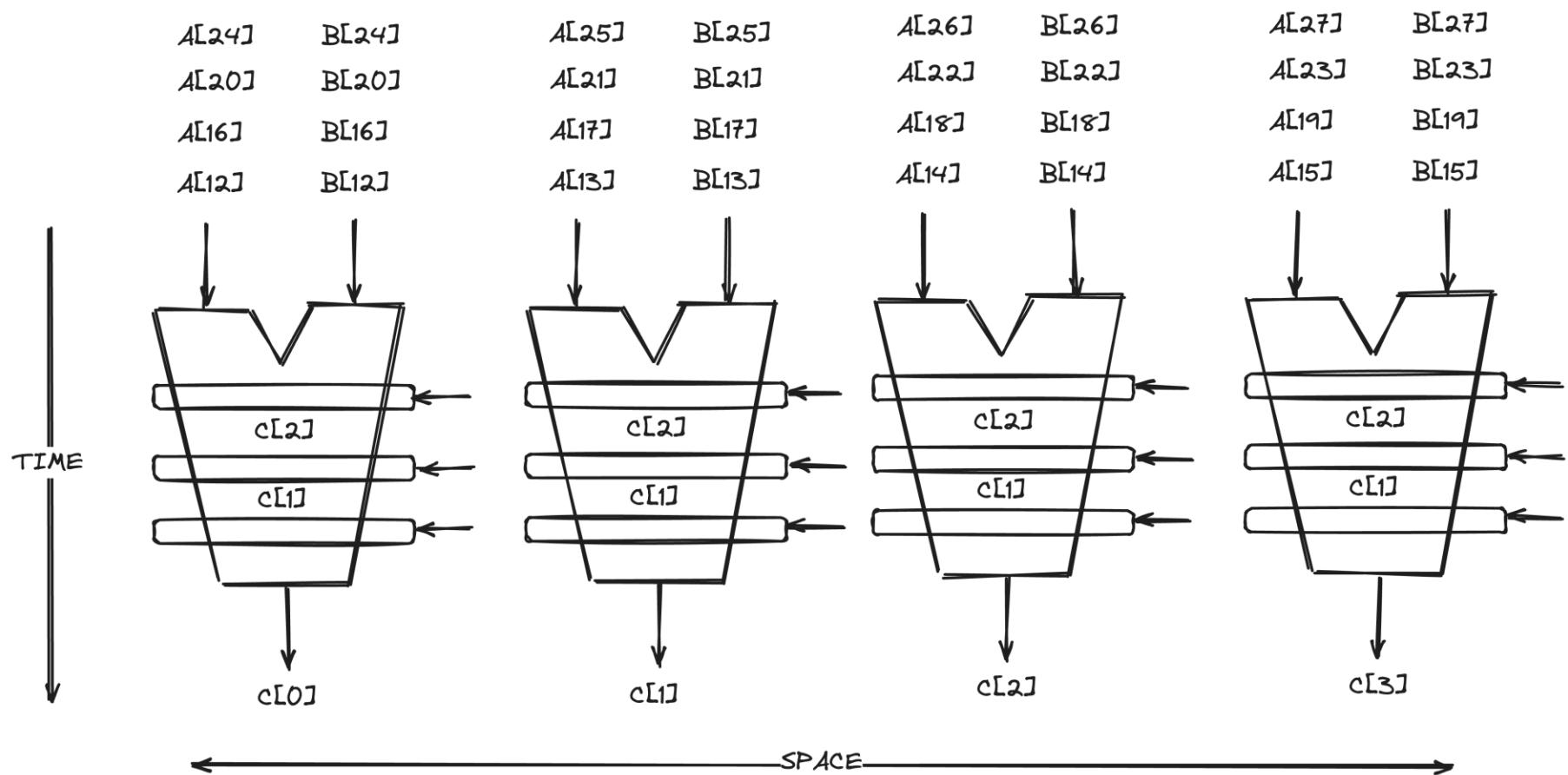


32-Thread warp executing ADD A[tid], B[tid] -> C[tid]

Execution with one pipelined FU

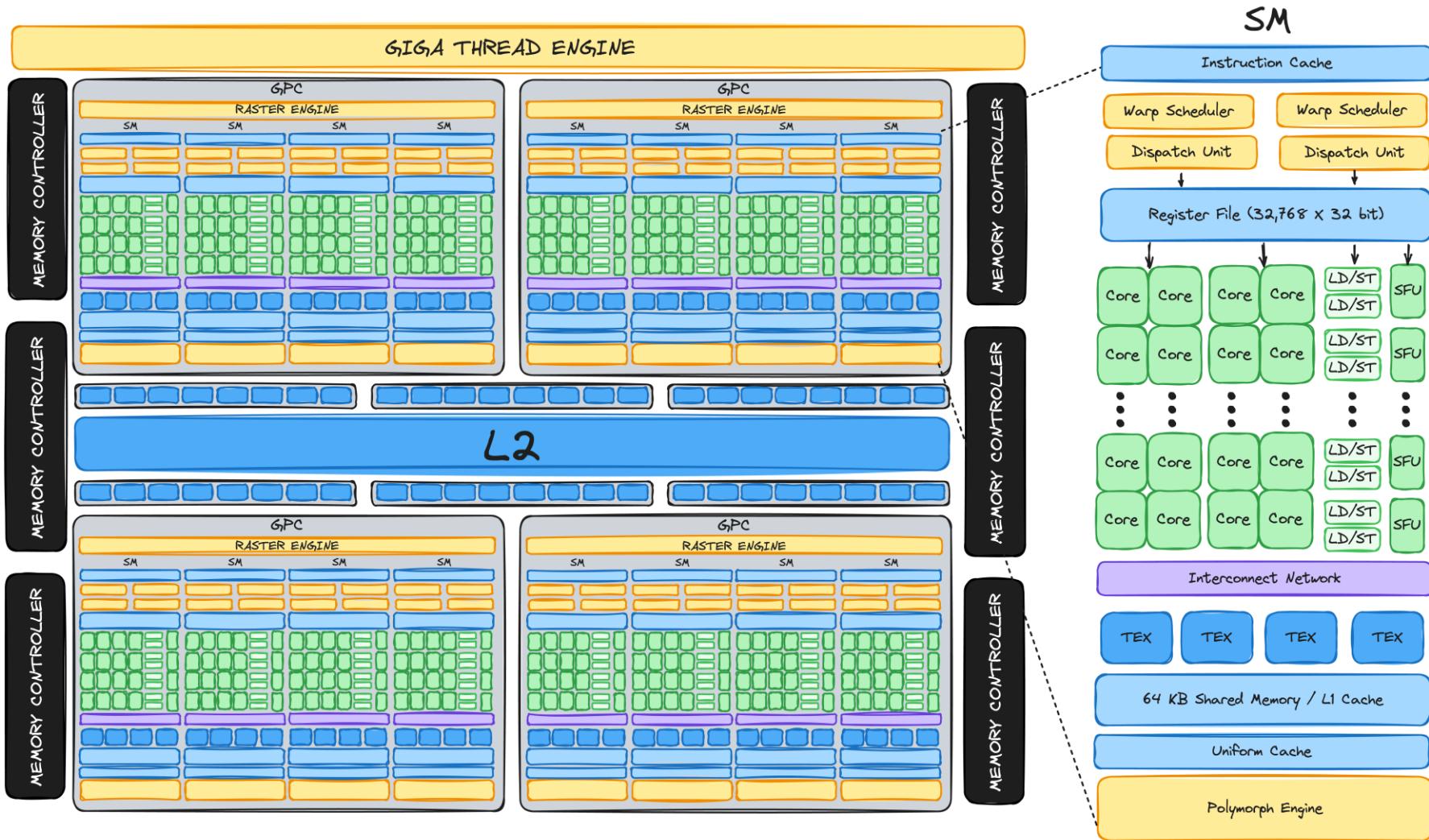


Execution with 4 pipelined FUs



But where do threads run?

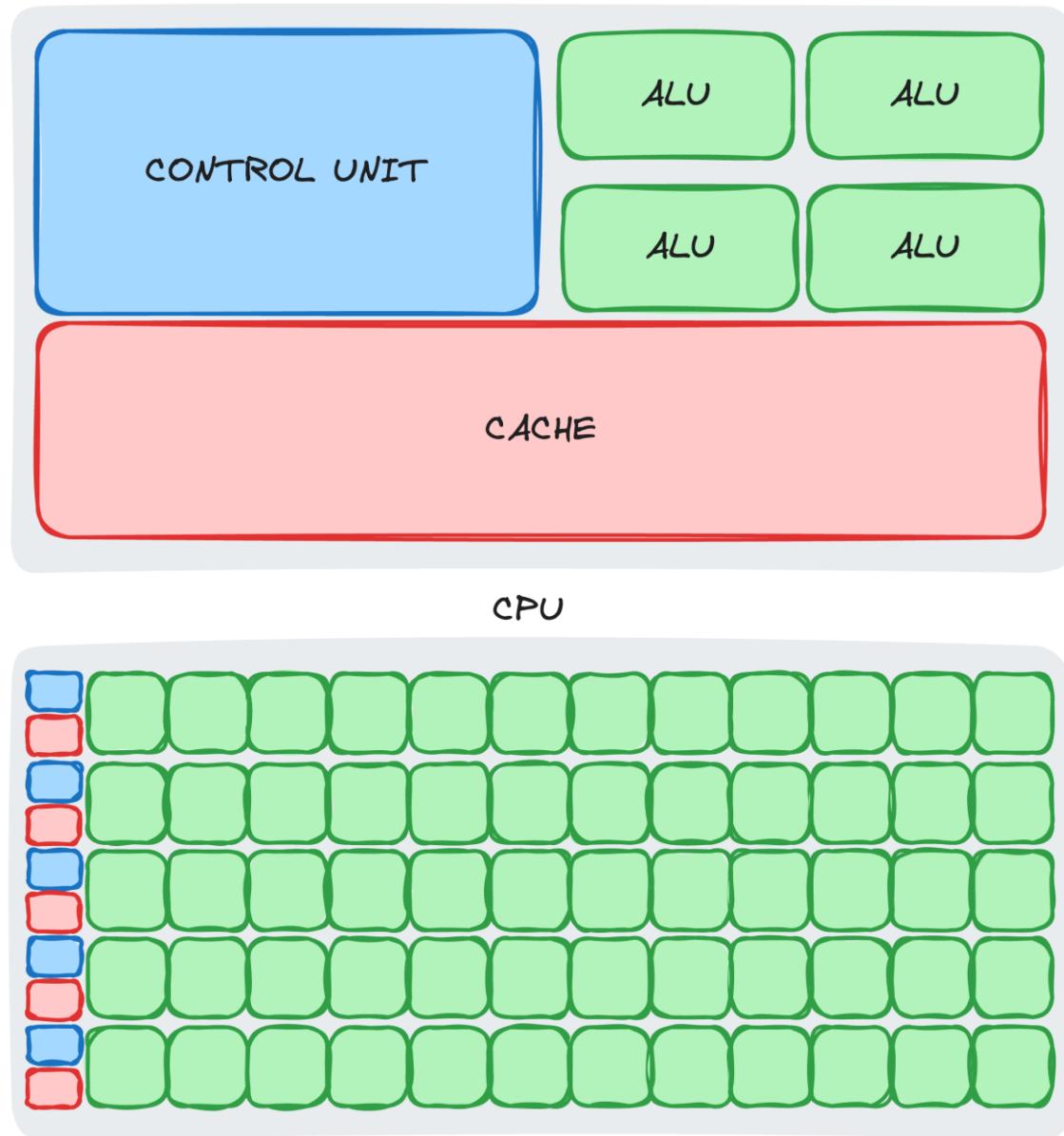
- Streaming Multiprocessor (SM)
- 32 cores in each
 - o Cuda core or Streaming Processor (SP)
 - o 1 integer multiplication per clock
- Register File of 32 KB
- 64 KB L1 Cache
- The Tesla (2006) & Fermi architectures are the foundation of modern Nvidia GPUs



FERMI ARCHITECTURE

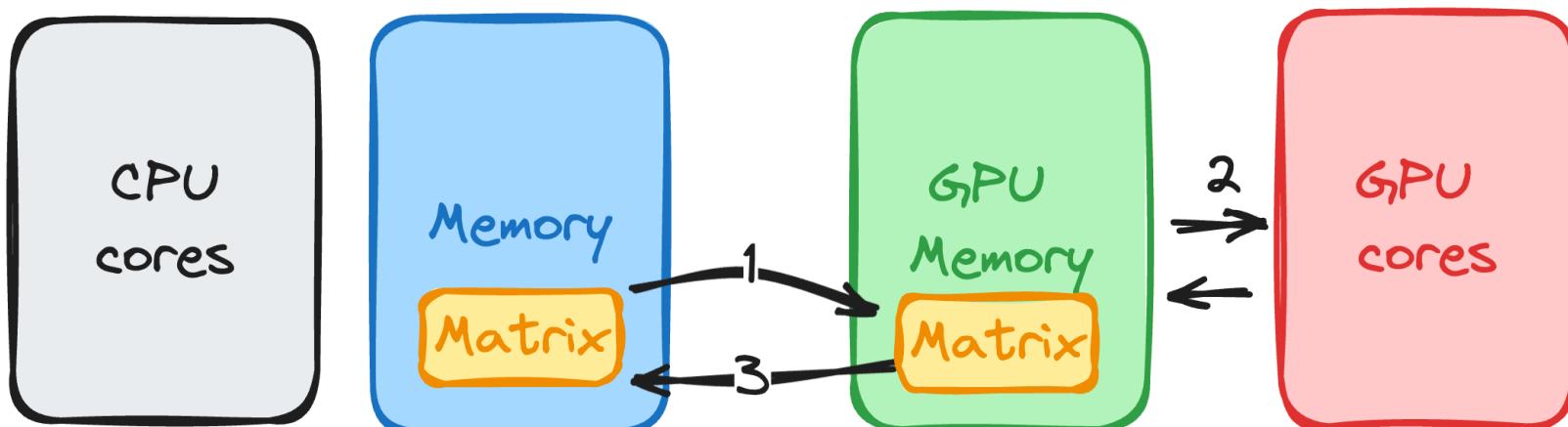
Comparison with CPU

- 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 FGM



GPU programming

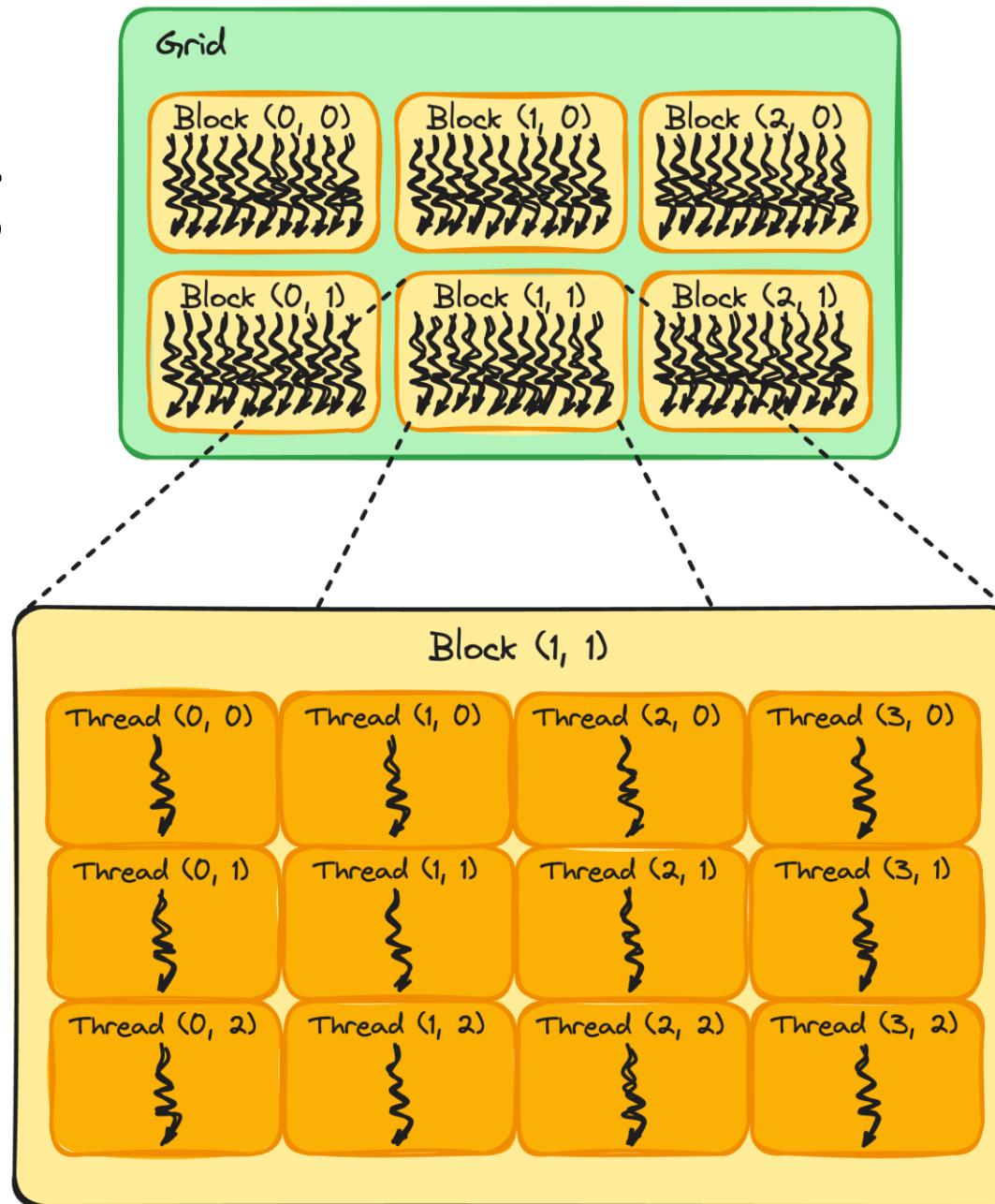
- The CPU is referred as **host**
- The GPU is referred as **device**
- Computation is offloaded to the GPU
- 3 steps:
 - CPU-GPU transfer
 - Latency & bandwidth must be considered
 - GPU kernel execution
 - Performance depends on how well the kernel is optimized for the GPU's architecture
 - GPU-CPU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



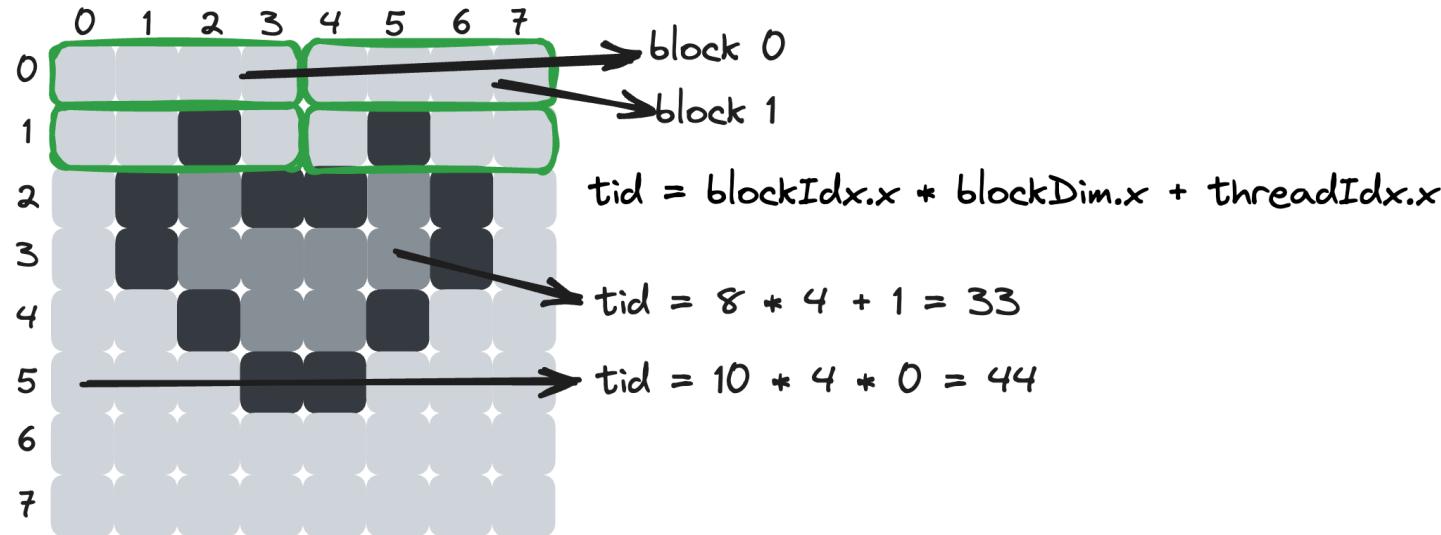
The basics

- 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()**

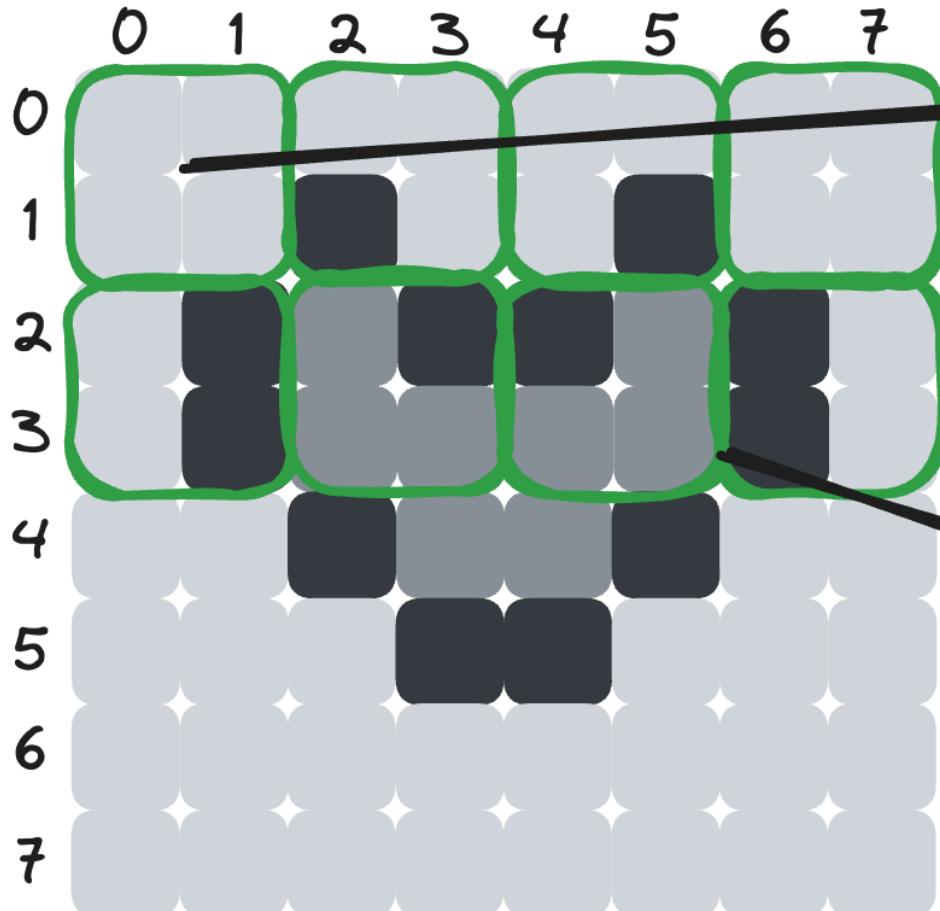
Data layout & Access

- 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)



block (0,0)

Row = blockIdx.y * blockDim.y + threadIdx.y
Col = blockIdx.x * blockDim.x + threadIdx.x

threadIdx.x = 1
threadIdx.y = 0
blockIdx.x = 2
blockIdx.y = 1

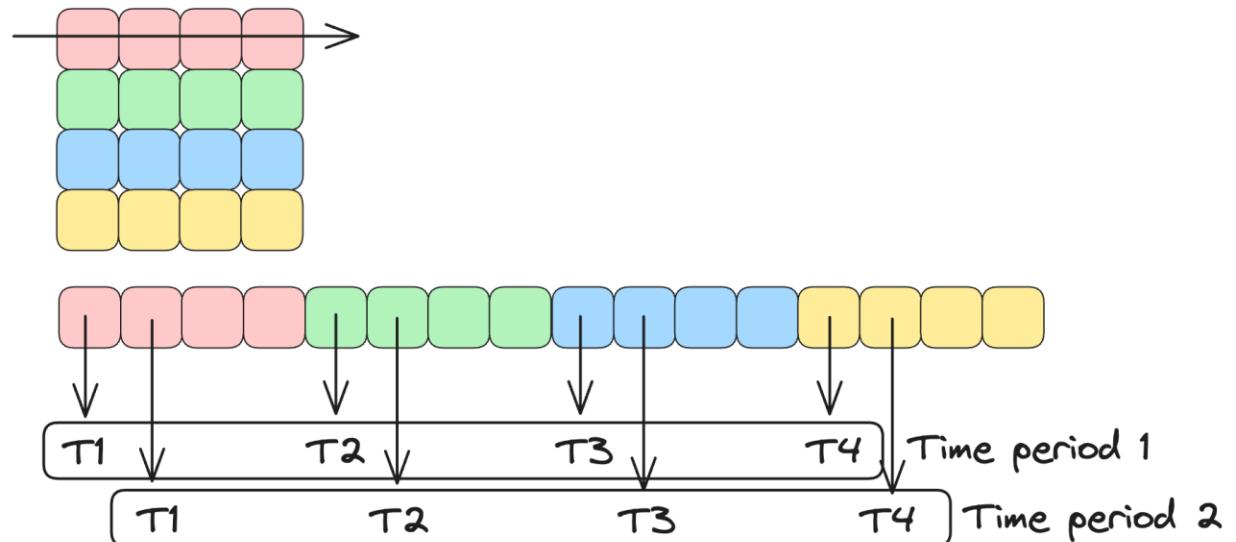
$$\text{Row} = 1 * 2 + 1 = 3$$

$$\text{Col} = 2 * 2 + 0 = 4$$

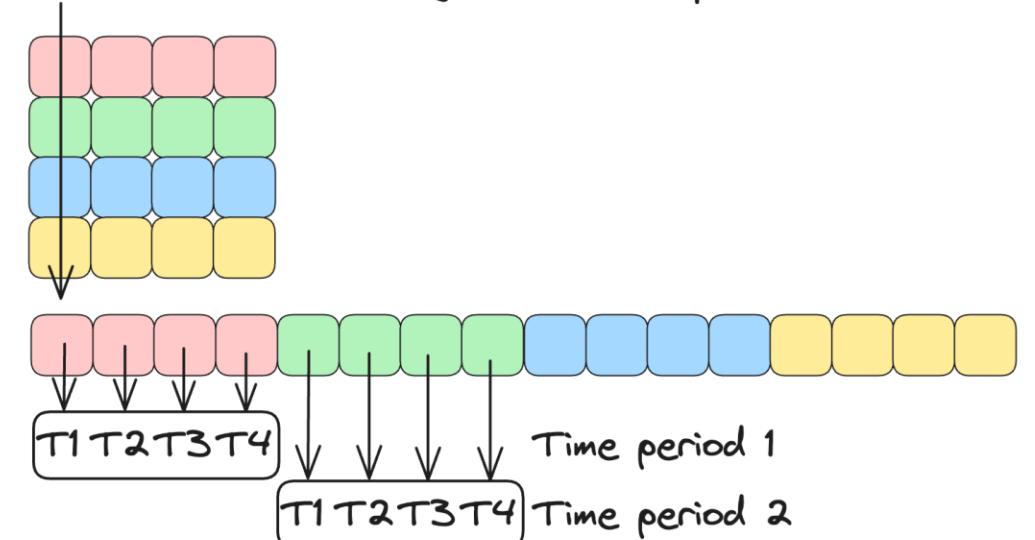
Memory access - Coalescing

- 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)

Not coalesced (row major) - 4 memory accesses

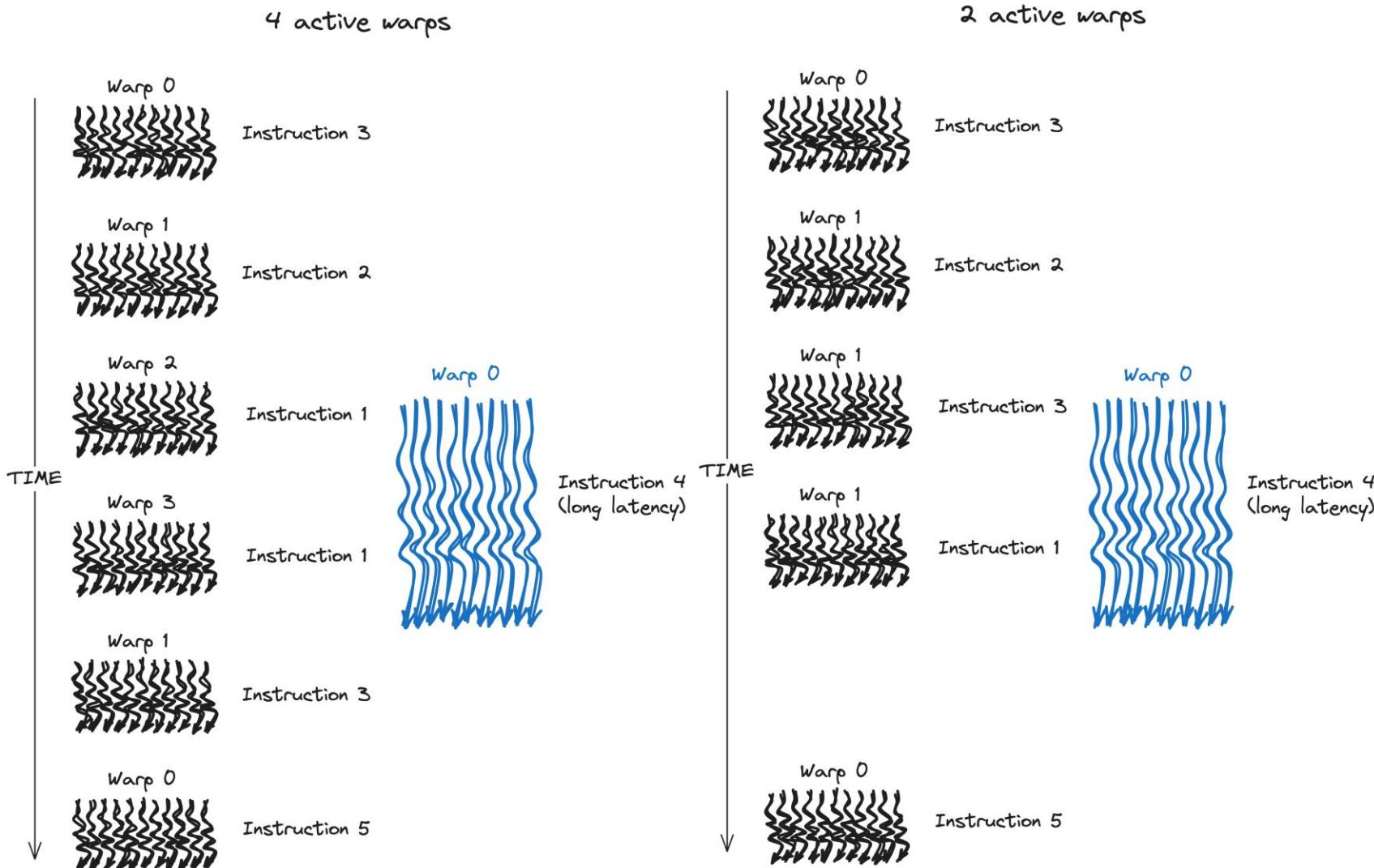


Coalesced (column major) - 1 memory access



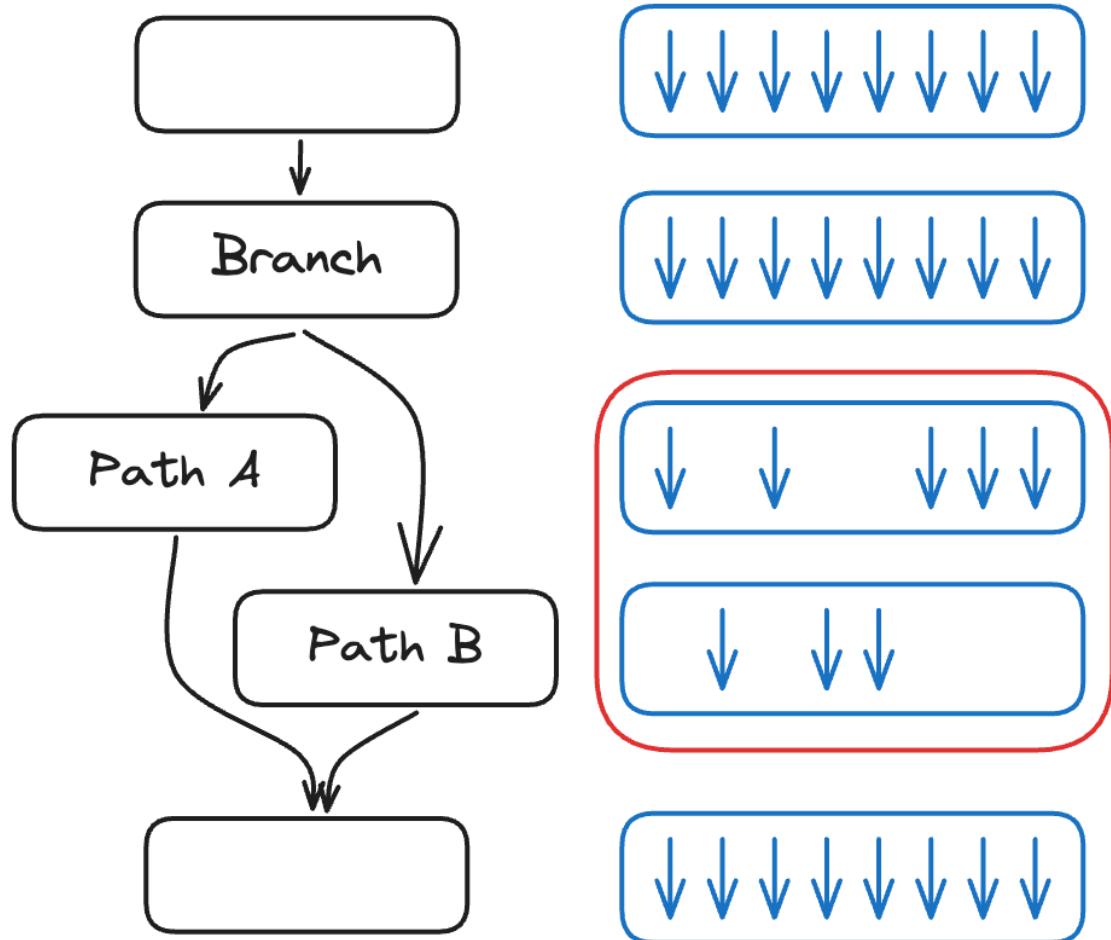
Memory access – latency hiding

- 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



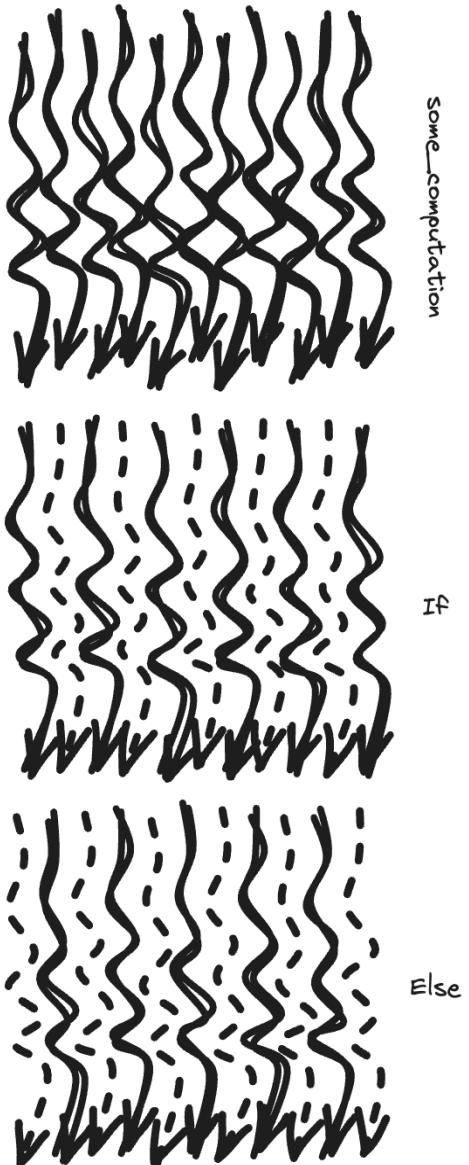
SIMD utilization - Branch divergence

- 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



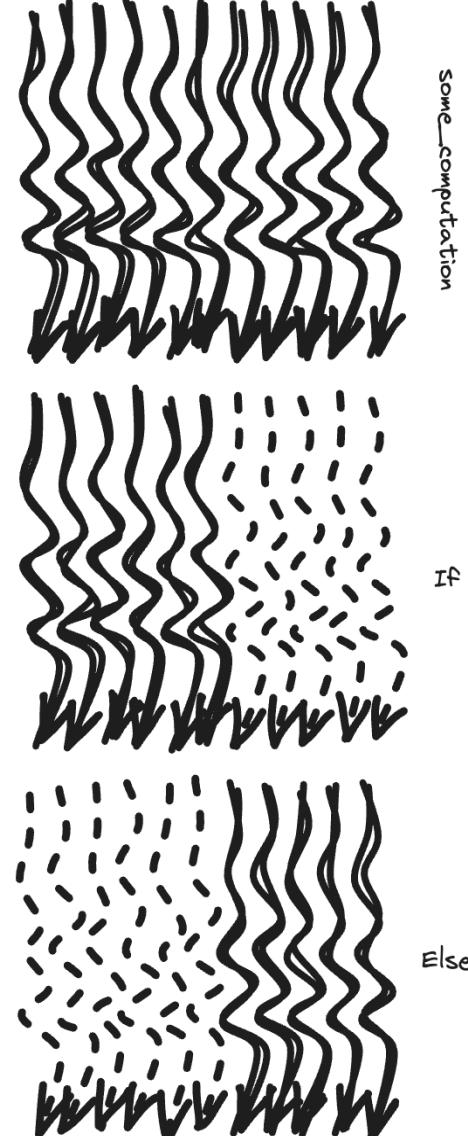
Intra warp divergence

```
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()
```

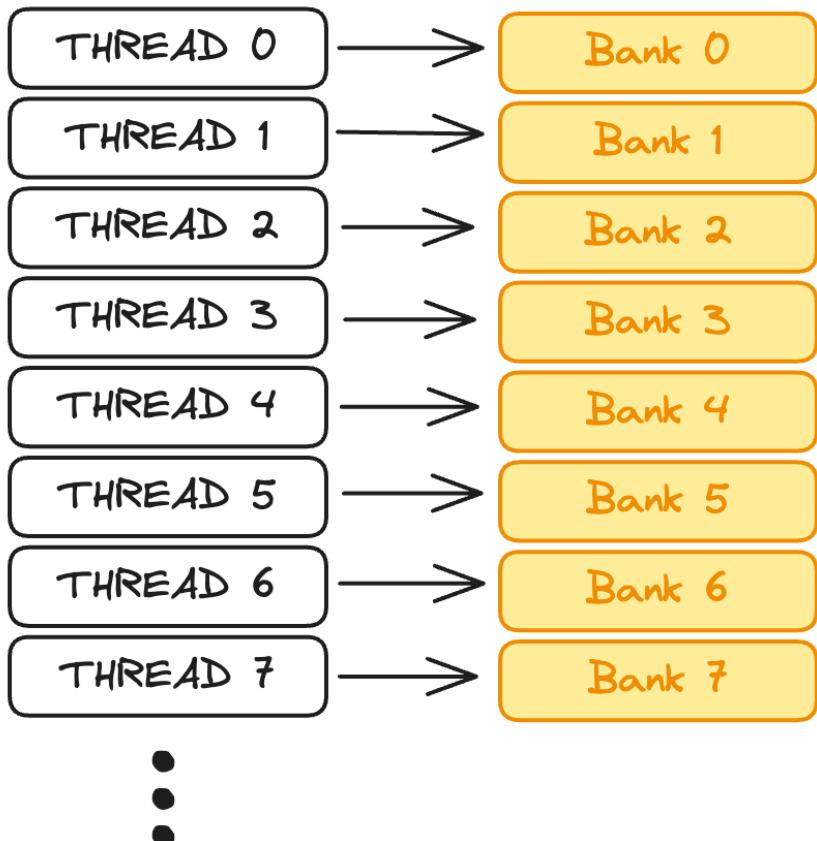


Shared Memory

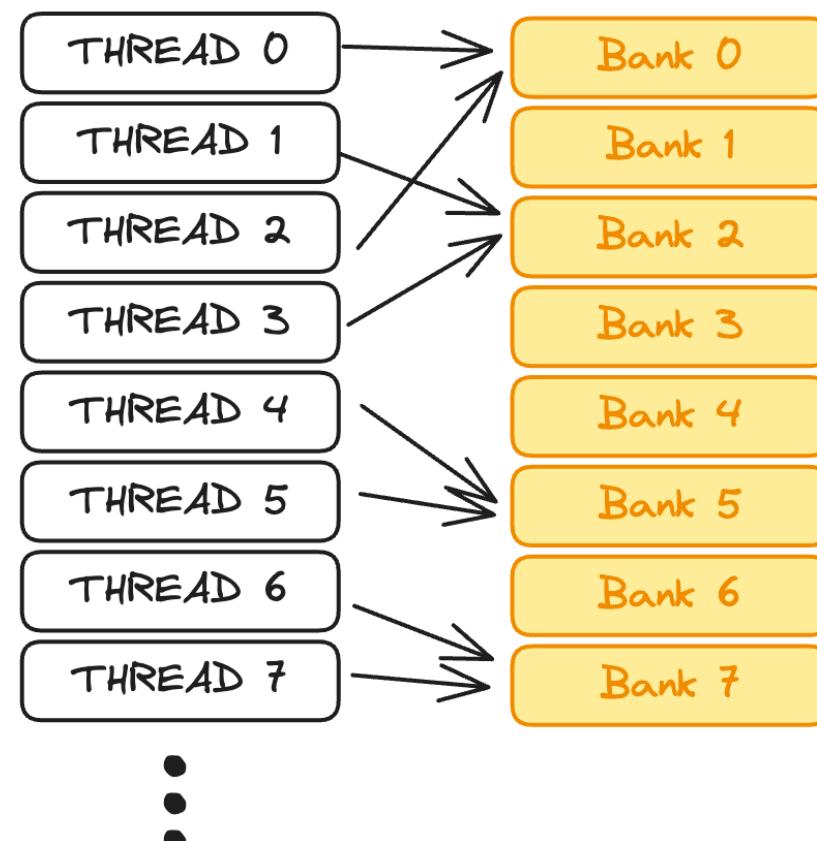
- It's a local cache shared between threads of the same block
- Manageable by the programmer via the `_shared_` directive
- Persists only until within the block's execution duration
- Typically organized into multiple banks to optimize the bandwidth
 - Each bank can service a memory request in one clock 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

CONFLICT FREE

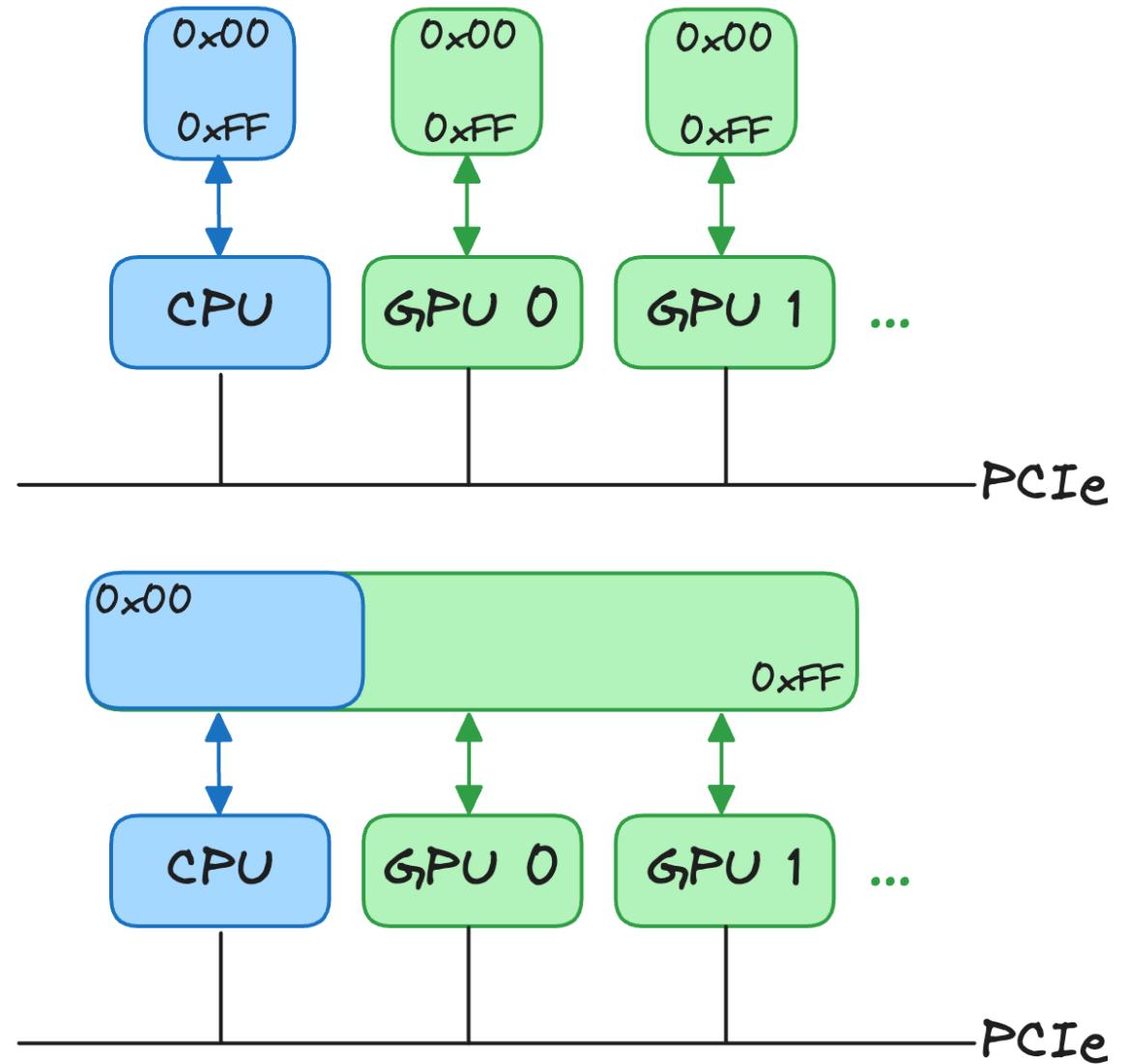


N-WAY CONFLICT



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 = 5000000000; // 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

- NVCC only converts the kernel into PTX code
- To compile the program to an executable
 - **nvcc gpu_nounifiedmem.cu**
 - This generates a .exe executable
- To view the generated PTX code of the kernel
 - **nvcc -ptx -o output.ptx gpu_nounifiedmem.cu**

```
.version 8.4      // ISA VERSION
.target sm_52      // Compute capability 5.2
.address_size 64  // 64-bit addresses

// .globl _Z5saxpyifPfS_

// saxpy function
.visible .entry _Z5saxpyifPfS_
    .param .u32 _Z5saxpyifPfS__param_0, // int n
    .param .f32 _Z5saxpyifPfS__param_1, // float a
    .param .u64 _Z5saxpyifPfS__param_2, // float* x
    .param .u64 _Z5saxpyifPfS__param_3 // float* y
{
    .reg .pred   %p<2>;
    .reg .f32    %f<5>;
    .reg .b32    %r<6>;
    .reg .b64    %rd<8>;

    // Load parameters
    ld.param.u32  %r2, [_Z5saxpyifPfS__param_0]; // int n
    ld.param.f32  %f1, [_Z5saxpyifPfS__param_1]; // float a
    ld.param.u64  %rd1, [_Z5saxpyifPfS__param_2]; // float* x
    ld.param.u64  %rd2, [_Z5saxpyifPfS__param_3]; // float* y

    // Calculate global thread index
    mov.u32    %r3, %ctaid.x;           // blockIdx.x
    mov.u32    %r4, %ntid.x;            // blockDim.x
    mov.u32    %r5, %tid.x;             // threadIdx.x
    mad.lo.s32 %r1, %r3, %r4, %r5;    // i = blockIdx.x * blockDim.x + threadIdx.x

    // if (i < n)
    setp.ge.s32  %p1, %r1, %r2;
    @%p1 bra     $L__BB0_2;

    // Body of the if condition
    cvta.to.global.u64 %rd3, %rd2;    // Convert y to global address space
    cvta.to.global.u64 %rd4, %rd1;    // Convert x to global address space
    mul.wide.s32  %rd5, %r1, 4;      // Compute byte offset (i * sizeof(float))
    add.s64     %rd6, %rd4, %rd5;    // Compute address of x[i]
    ld.global.f32 %f2, [%rd6];       // Load x[i]
    add.s64     %rd7, %rd3, %rd5;    // Compute address of y[i]
    ld.global.f32 %f3, [%rd7];       // Load y[i]
    fma.rn.f32  %f4, %f2, %f1, %f3; // Compute a * x[i] + y[i]
    st.global.f32 [%rd7], %f4;        // Store the result back to y[i]

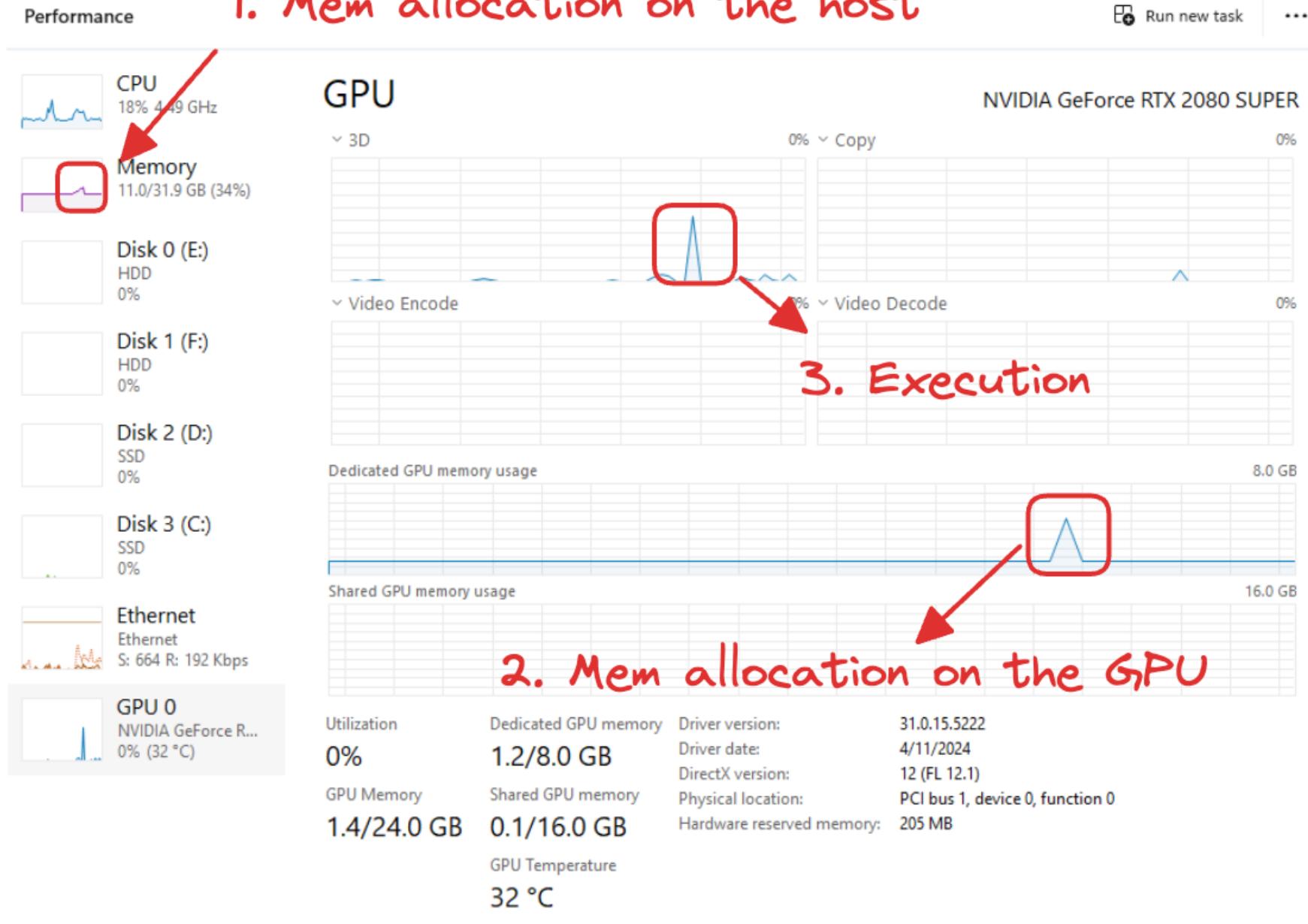
$L__BB0_2:
    ret;
}
```

Execution

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

- Machine specs:
 - RTX 2080 SUPER 8GB RAM
 - I7 7700k
 - 32GB RAM 3000Mhz

1. Mem allocation on the host



A comparison

- CPU-equivalent code
- Much slower, as expected

```
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 = 5000000000;  
        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