Nvidia CUDA

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

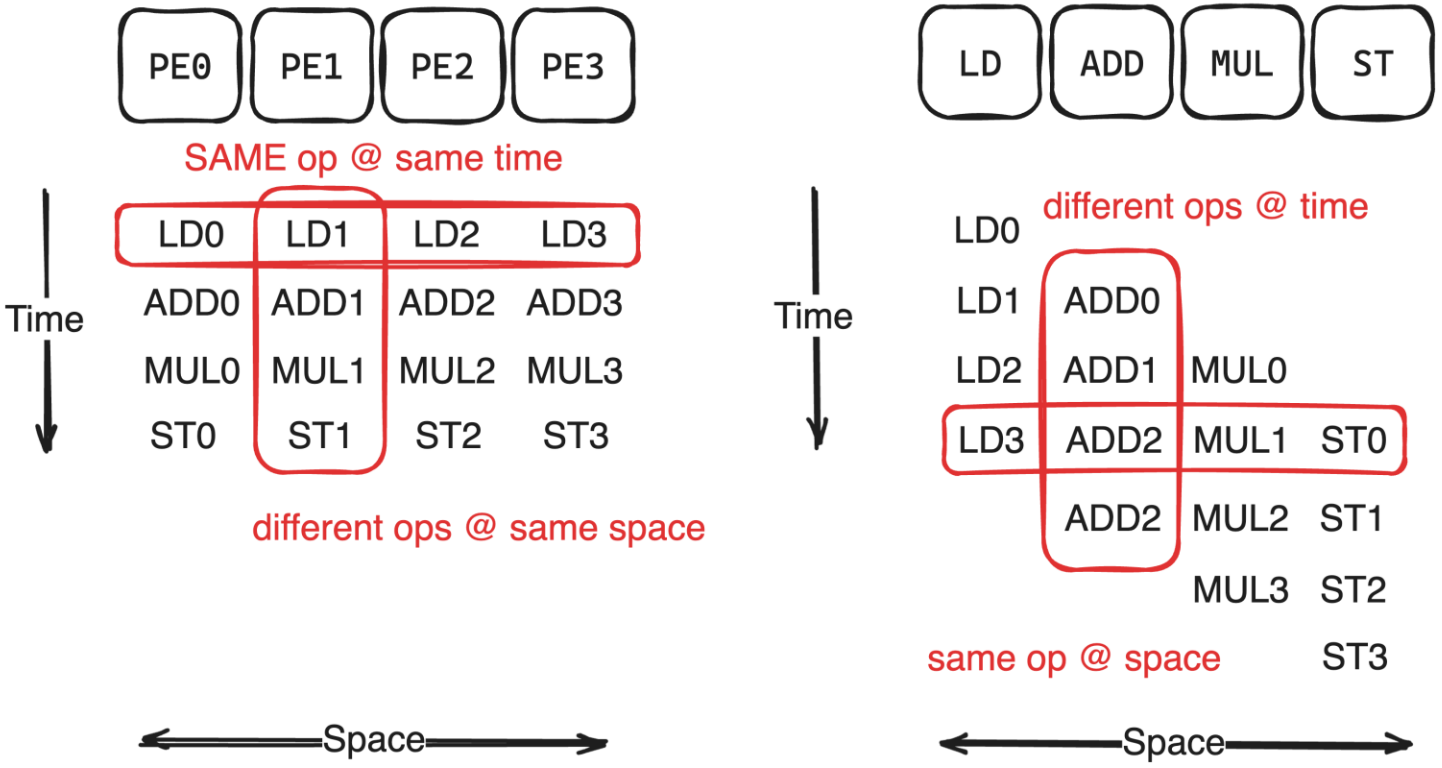
- A GPU is a specialized electronic circuit designed to handle multiple tasks simultaneously in an efficient manner
- It's equipped with many small efficient cores
  - Many in-order cores
- Basically, a SIMD-like machine (Single Instruction Multiple Data) under the hood
- Limits the drawbacks of canonical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

SIMD (Single Instruction Multiple Data)

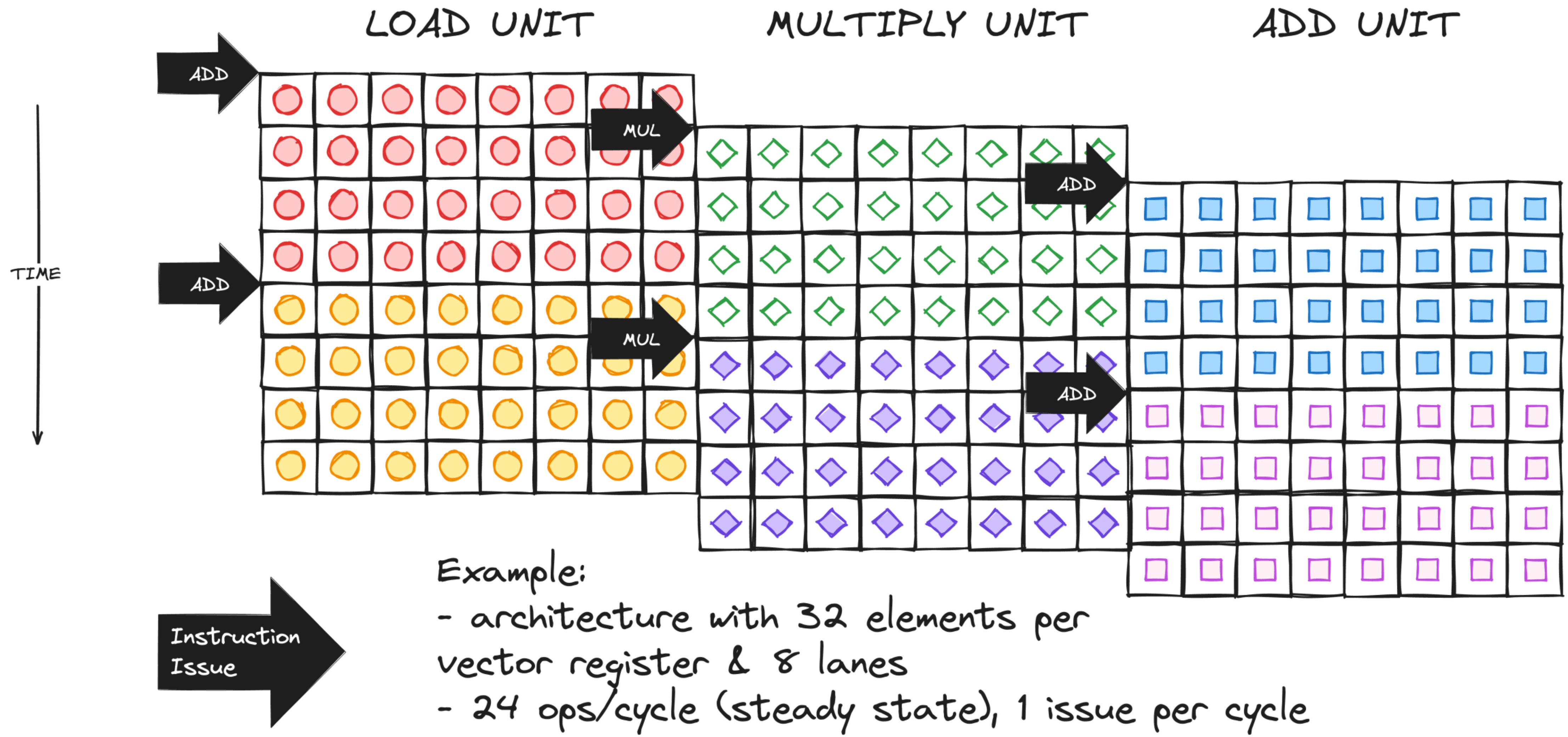
- A type of parallel processing that executes the same operation on multiple data
- Another approach to parallelism
  - We studied ILP (Instruction Level Parallelism)
  - GPUs & SIMD are based on DLP (Data Level Parallelism)
- ILP: Instruction level parallelism (Out-of-order execution for example)
- 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



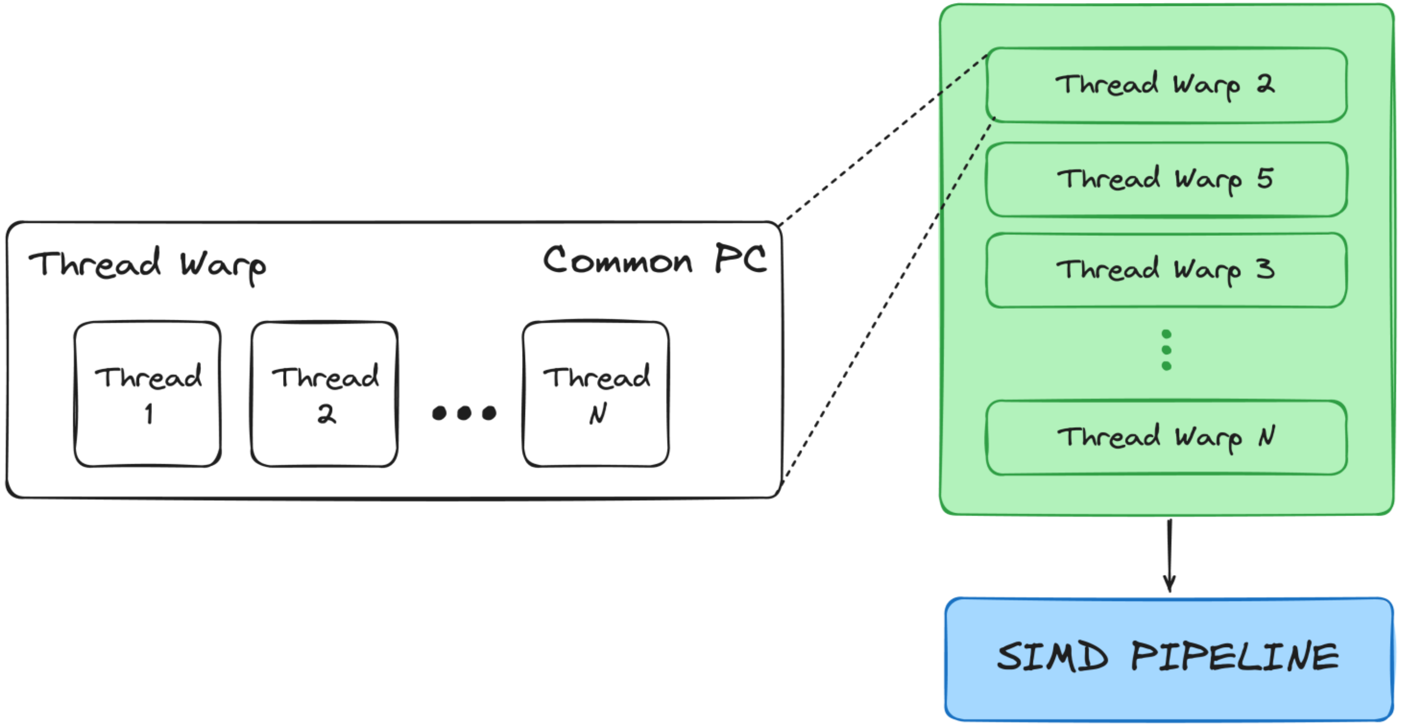
Example:
- architecture with 32 elements per vector register & 8 lanes
- 24 ops/cycle (steady state), 1 issue per cycle

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

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

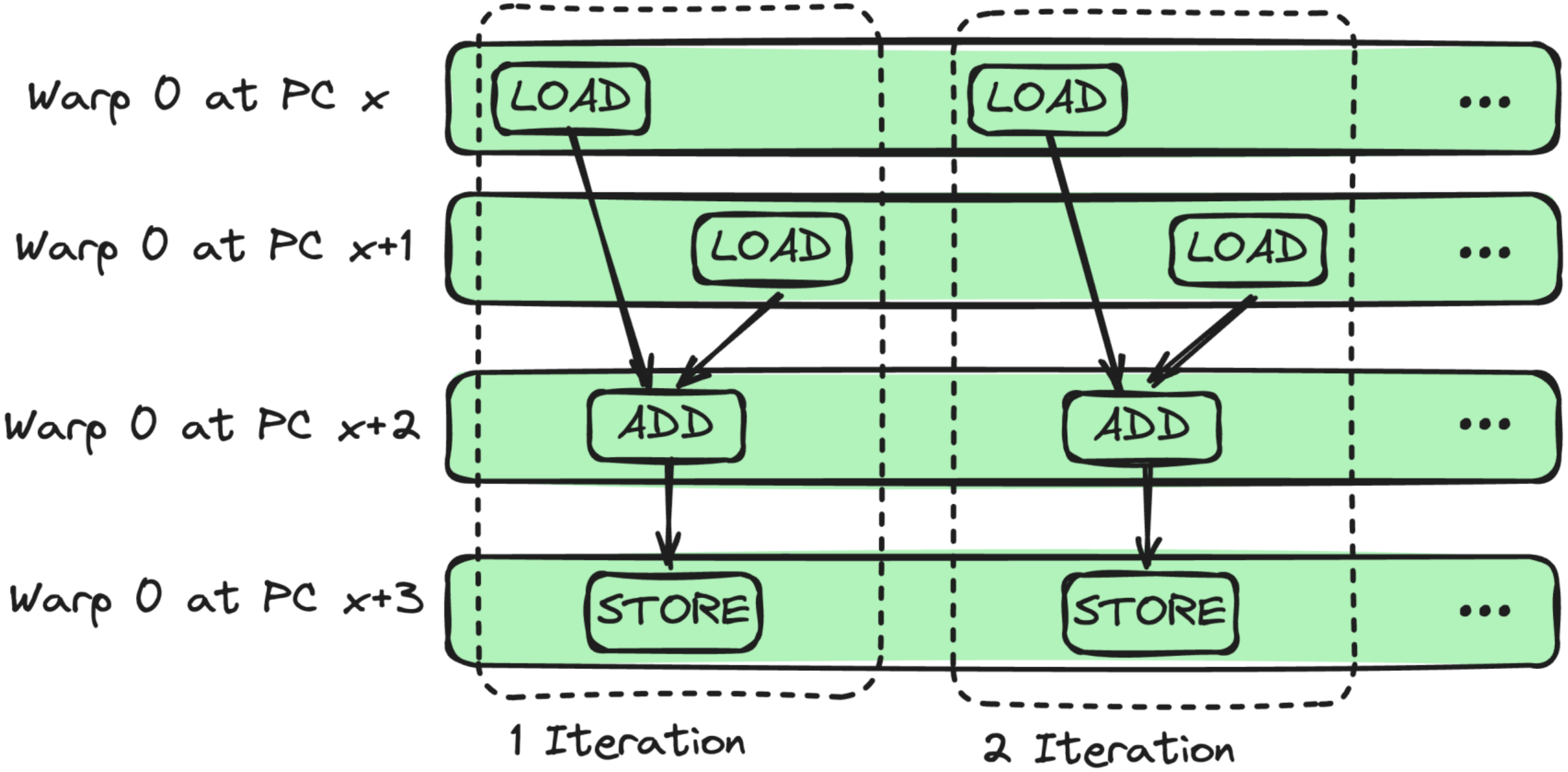
THREADS & WARPS



- 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

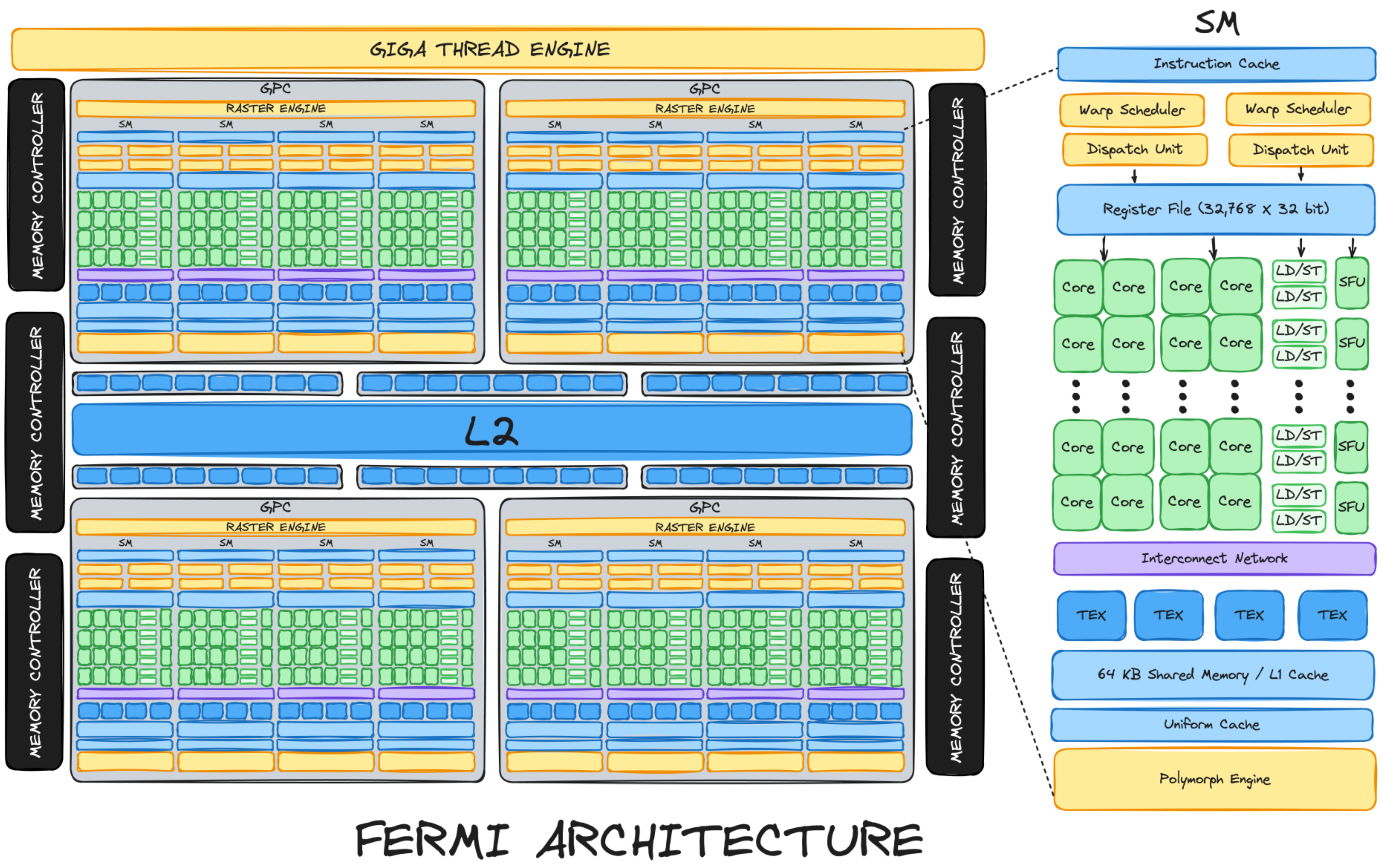
```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```



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

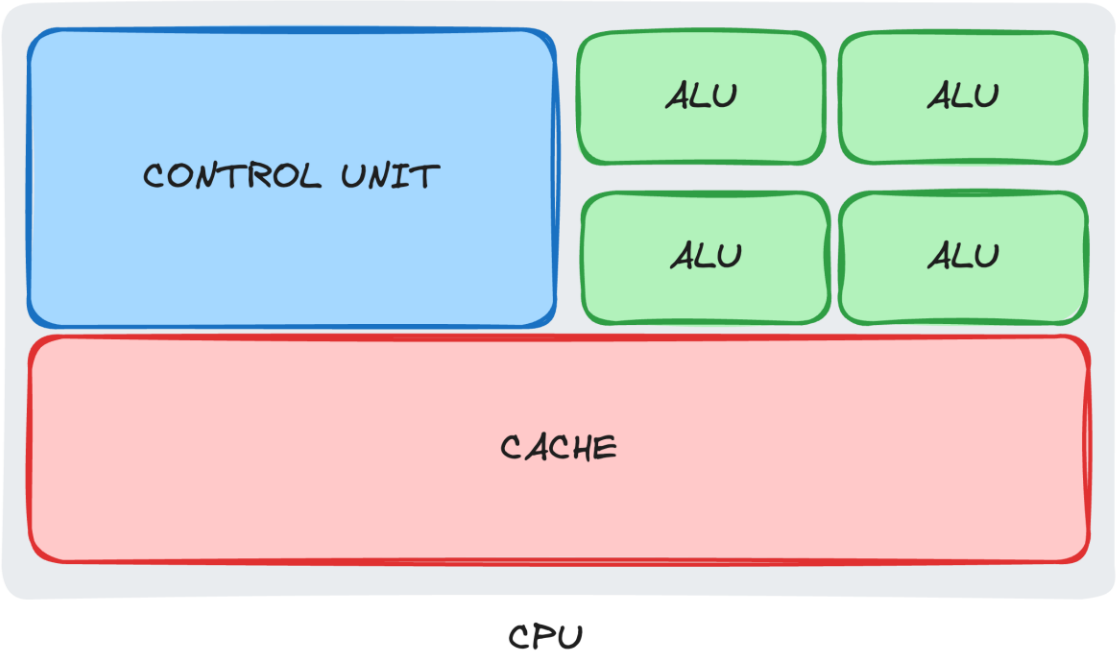
But where do threads run?

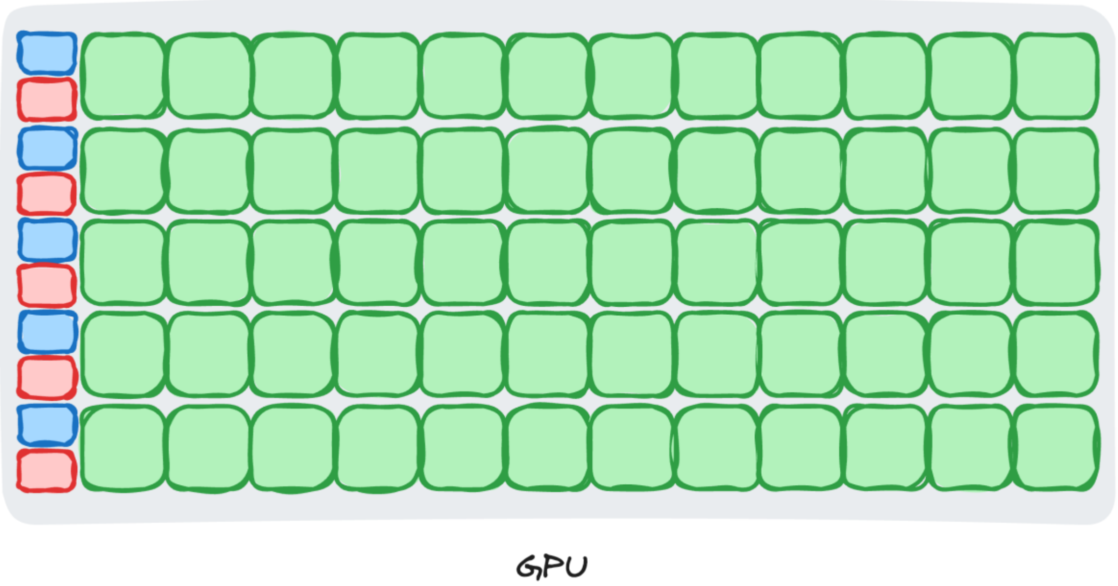
- Streaming Multiprocessor (SM)
- 32 cores in each
  - Cuda core or Streaming Processor (SP)
  - 1 integer multiplication per clock
- Register File of 32 KB
- 64 KB L1 Cache
- The Tesla (2006) & Fermi architectures are the foundation of modern Nvidia GPUs



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

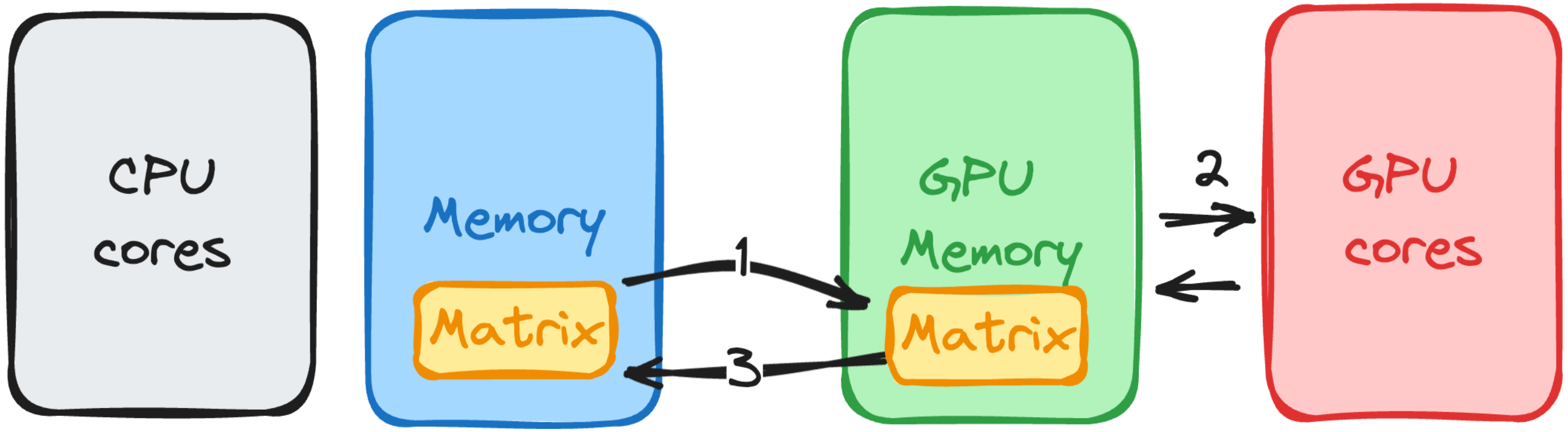




https://github.com/ilcors-dev/unibo-nvidia-cuda-research

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

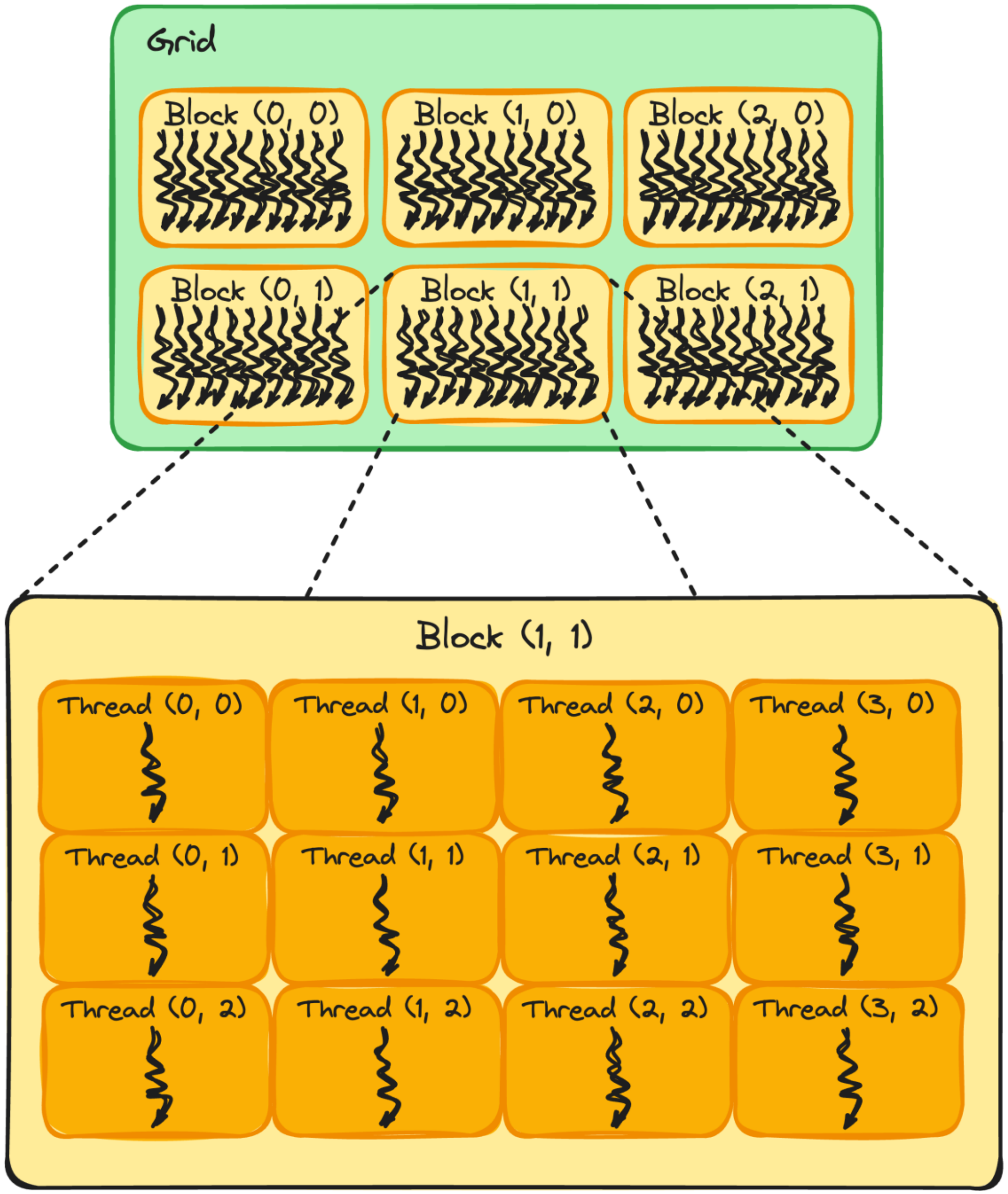


https://github.com/ilcors-dev/unibo-nvidia-cuda-research

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



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

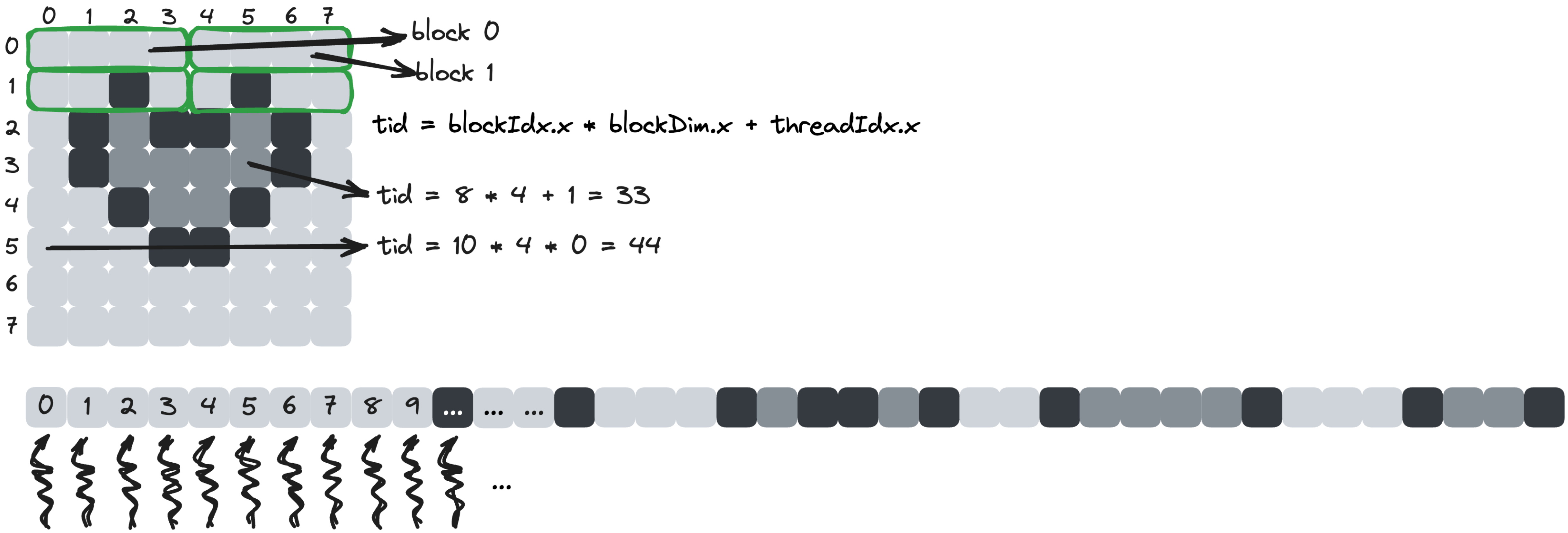
The basics

- 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

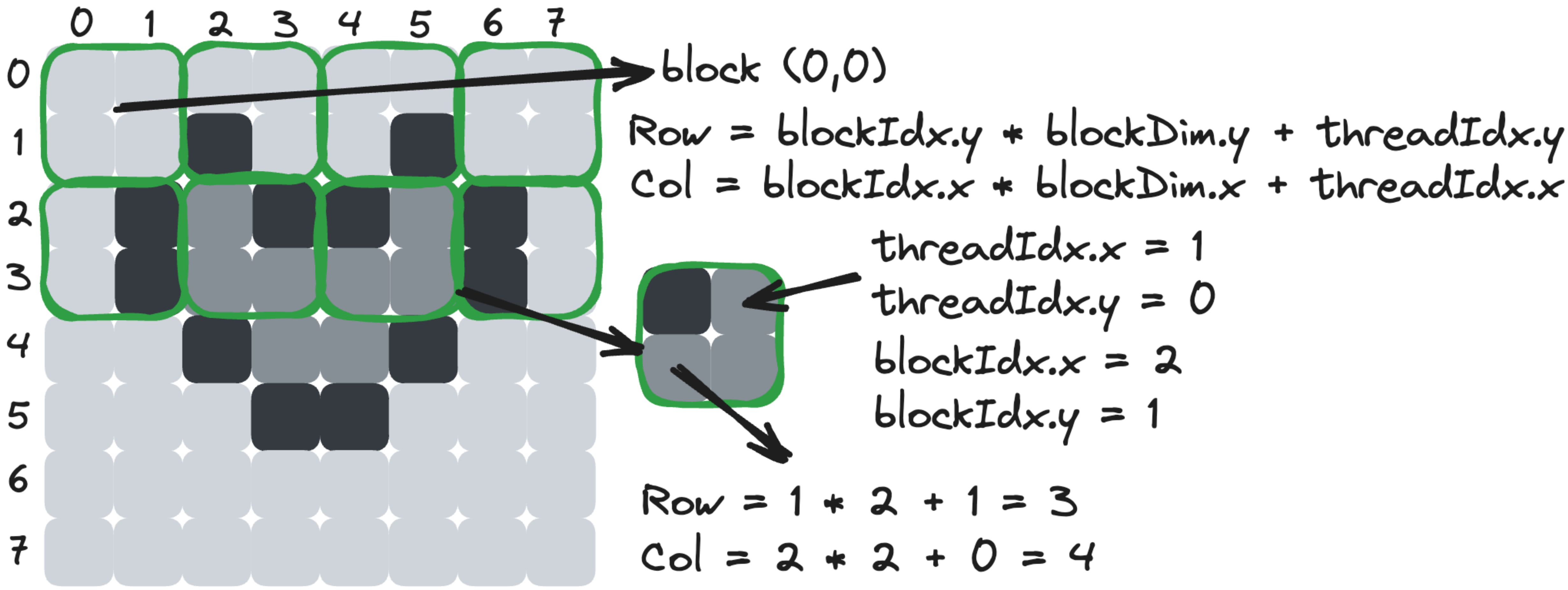
- The position of a thread in the global grid depends on how data is layered down
  - 1D, 2D, 3D
  - The programmer can choose it
- **threadIdx** the index of a thread within its block
- **blockIdx** the index of a block within its grid
- **blockDim** how many threads there are in each x, y, z directions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

1 dimension (vectors)



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

2 dimension (matrixes)



```
__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
- Monolithic Kernel: 1 thread per data element (we could also use a stride-grid loop)

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;
    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;
}
```

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

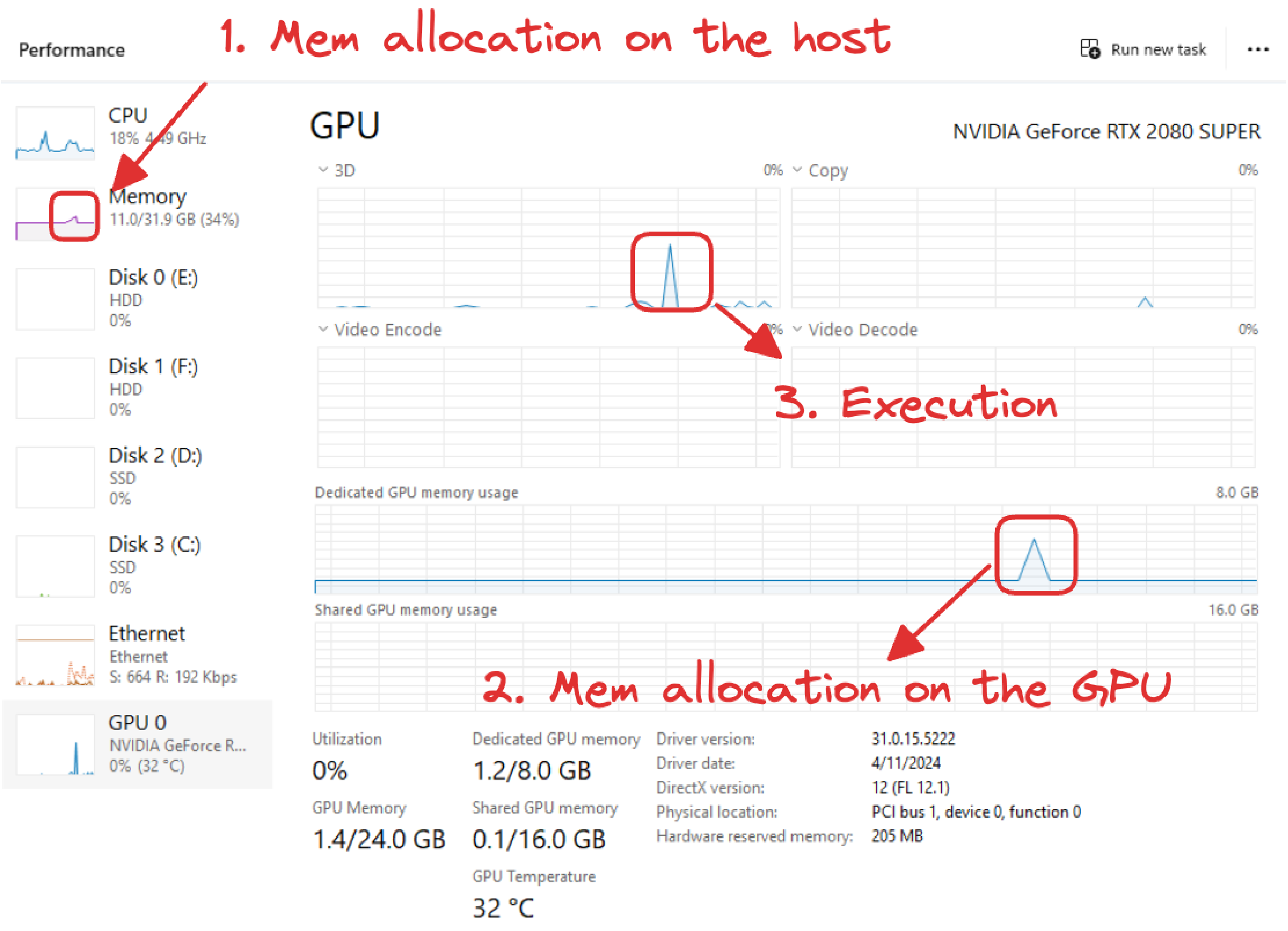
Execution

```
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

Profiling

```
PS C:\Users\luca\Desktop\NetworkShared\cuda - Copy> nvprof .\gpu_nounifiedmem.exe
==18500== NVPROF is profiling process 18500, command: .\gpu_nounifiedmem.exe
for 500000000 elements, 1953125 blocks are needed
14ms
==18500== Profiling application: .\gpu_nounifiedmem.exe
==18500== Warning: 13 API trace records have same start and end timestamps.
This can happen because of short execution duration of CUDA APIs and low timer resolution on the underlying operating system.
==18500== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   65.25%  337.79ms         2  168.90ms  167.91ms  169.88ms  [CUDA memcpy HtoD]
                   31.99%  165.60ms         1  165.60ms  165.60ms  165.60ms  [CUDA memcpy DtoH]
                    2.76%  14.276ms         1  14.276ms  14.276ms  14.276ms  saxpy(int, float, float*, float*)
      API calls:   78.02%  584.91ms         3  194.97ms  165.71ms  251.22ms  cudaMemcpy
                   14.23%  106.70ms         2  53.351ms  9.3110ms  97.392ms  cudaMalloc
                    3.14%  23.546ms         2  11.773ms  10.496ms  13.050ms  cudaFree
                    2.64%  19.768ms         1  19.768ms  19.768ms  19.768ms  cuDevicePrimaryCtxRelease
                    1.91%  14.291ms         1  14.291ms  14.291ms  14.291ms  cudaDeviceSynchronize
                    0.06%  458.40us         1  458.40us  458.40us  458.40us  cudaLaunchKernel
                    0.00%  35.200us         1  35.200us  35.200us  35.200us  cuLibraryUnload
                    0.00%  16.100us       114     141ns       0ns  1.7000us  cuDeviceGetAttribute
                    0.00%  2.6000us         3     866ns     100ns  2.3000us  cuDeviceGetCount
                    0.00%  2.0000us         1  2.0000us  2.0000us  2.0000us  cuModuleGetLoadingMode
                    0.00%  1.7000us         2     850ns     200ns  1.5000us  cuDeviceGet
                    0.00%     400ns         1     400ns     400ns     400ns  cuDeviceGetName
                    0.00%     300ns         1     300ns     300ns     300ns  cuDeviceGetLuid
                    0.00%     200ns         1     200ns     200ns     200ns  cuDeviceTotalMem
                    0.00%     200ns         1     200ns     200ns     200ns  cuDeviceGetUuid
                    0.00%     200ns         1     200ns     200ns     200ns  cudaGetLastError
```


1. Mem allocation on the host
Performance
Run new task
CPU
18% 4.49 GHz
Memory
11.0/31.9 GB (34%)
Disk 0 (E:)
HDD
0%
Disk 1 (F:)
HDD
0%
Disk 2 (D:)
SSD
0%
Disk 3 (C:)
SSD
0%
Ethernet
Ethernet
S: 664 R: 192 Kbps
GPU 0
NVIDIA GeForce R...
0% (32 °C)
GPU
NVIDIA GeForce RTX 2080 SUPER
3D
0%
Copy
0%
Video Encode
Video Decode
0%
3. Execution
Dedicated GPU memory usage
8.0 GB
Shared GPU memory usage
16.0 GB
2. Mem allocation on the GPU
Utilization
0%
Dedicated GPU memory
1.2/8.0 GB
GPU Memory
1.4/24.0 GB
Shared GPU memory
0.1/16.0 GB
GPU Temperature
32 °C
Driver version: 31.0.15.5222
Driver date: 4/11/2024
DirectX version: 12 (FL 12.1)
Physical location: PCI bus 1, device 0, function 0
Hardware reserved memory: 205 MB

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 = 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

    // Measure performance
    auto start = std::chrono::high_resolution_clock::now();

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

    auto end = std::chrono::high_resolution_clock::now();

    std::cout << std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count() << "ms\n";

    return 0;
}
```

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

The tip of the iceberg

- CUDA provides many libraries such as
  - cuBLAS (CUDA Basic Linear Algebra Subroutines)
  - cuDNN (CUDA Deep Neural Network library)
  - cuFFT (CUDA Fast Fourier Transform library)
  - ...
- NVVP profiling
- Coalesced access
- Intra warp divergence handling
- Shared memory
- Asynchronous Execution:
  - Overlap computation with data transfer
  - **cudaStreamCreate, cudaMemcpyAsync, cudaMemcpyAsync**

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 SIMT execution model

- Not suited for all types of computations
- Knowledge needed:
  - GPUs architecture
  - know the CUDA specs
  - memory management