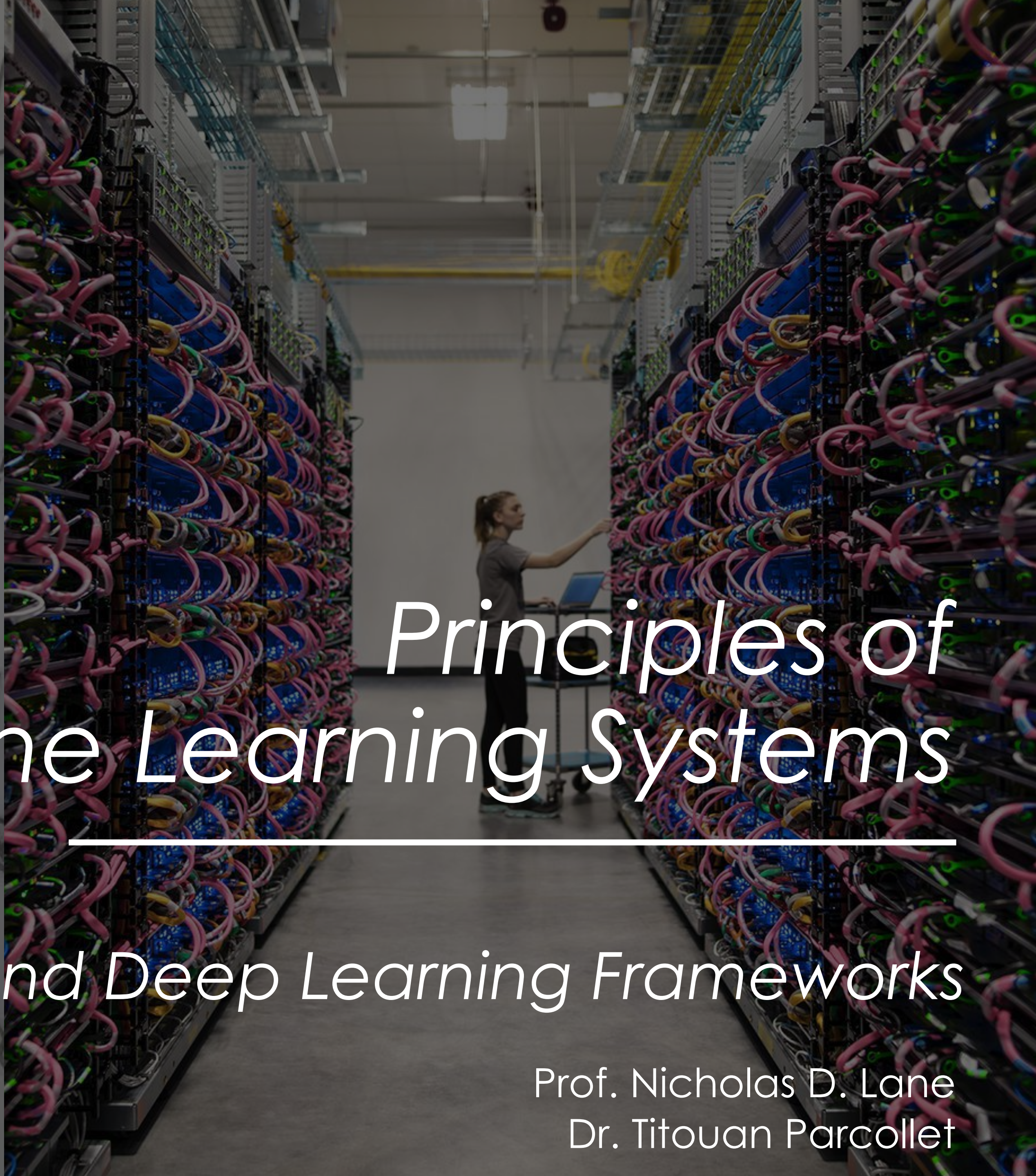
Principles of Machine Learning Systems

5: GPUs, CUDA and Deep Learning Frameworks



Prof. Nicholas D. Lane
Dr. Titouan Parcollet

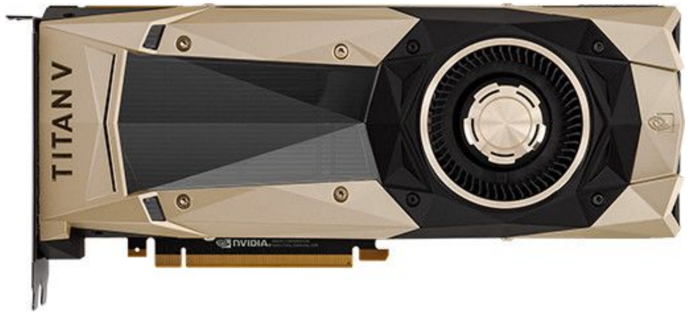
Roadmap for Today

1. Why do we need to understand GPUs?
2. GPU hardware and CUDA.
3. Practical CUDA optimisation example.
4. PyTorch CUDA bindings.

Roadmap for Today

1. **Why do we need to understand GPUs?**
2. GPU hardware and CUDA.
3. Practical CUDA optimisation example.
4. PyTorch CUDA bindings.

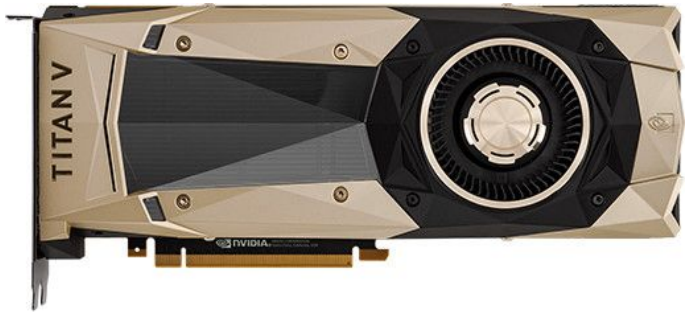
Why do we need to understand GPUs?




A vast majority of the DL models are trained with GPUs.

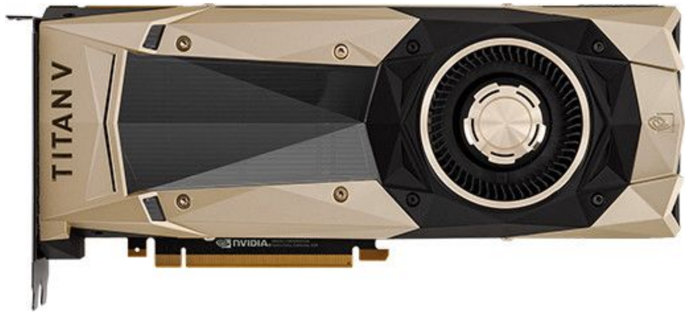
Most engineers do not know what it **means** to train on GPU.

Why do we need to understand GPUs?



As long as you are playing with MNIST or toy tasks, it does not matter.

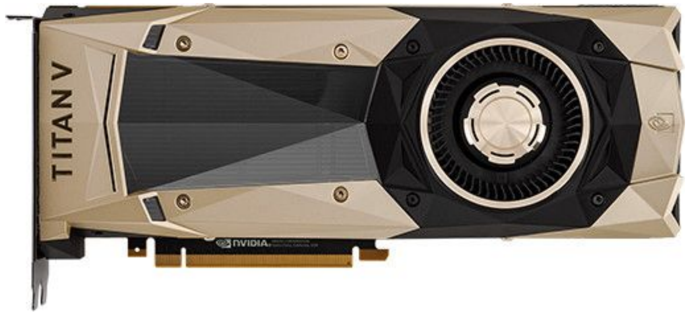
Why do we need to understand GPUs?



But the real world is different:

- Why is my training so slow while my GPU is worth £6,000?
- Can I train this 30B parameters Llama model on my RTX 3090?
- Why is my inference so slow while my GPU is equipped with Tensorcores?

Why do we need to understand GPUs?



Your hardware stack, e.g. your GPU, is your secondary tool — **learn to use it**.

The number of issues related to the lack of hardware knowledge is infinite.

Why do we need to understand GPUs?

Examples:

70x faster matmul with a proper cuda kernel.

Kernel	GFLOPs/s
1: Naive	309.0
2: GMEM Coalescing	1986.5
3: SMEM Caching	2980.3
4: 1D Blocktiling	8474.7
5: 2D Blocktiling	15971.7
6: Vectorized Mem Access	18237.3
9: Autotuning	19721.0
10: Warptiling	21779.3
0: cuBLAS	23249.6

https://siboehm.com/articles/22/CUDA-MMM

8x faster real training time of a RNN-based speech recogniser.

Forward pass:

Batch=16	fast SLi-GRU (CUDA+PyTorch)	slow SLi-GRU (PyTorch)
L=100	0.05 s	0.11 s
L=500	0.25 s	0.55 s
L=1000	0.50 s	1.11 s
L=2000	1.02 s	2.26 s
L=3000	1.55 s	3.39 s

Backward pass:

Batch=16	fast SLi-GRU (CUDA+PyTorch)	slow SLi-GRU (PyTorch)
L=100	0.15 s	0.25 s
L=500	0.63 s	1.29 s
L=1000	1.27 s	3.68 s
L=2000	2.65 s	11.87 s
L=3000	3.84 s	24.39 s

Moumen, A., & Parcollet, T. (2023, June). Stabilising and accelerating light gated recurrent units for automatic speech recognition. ICASSP 2023.

Roadmap for Today

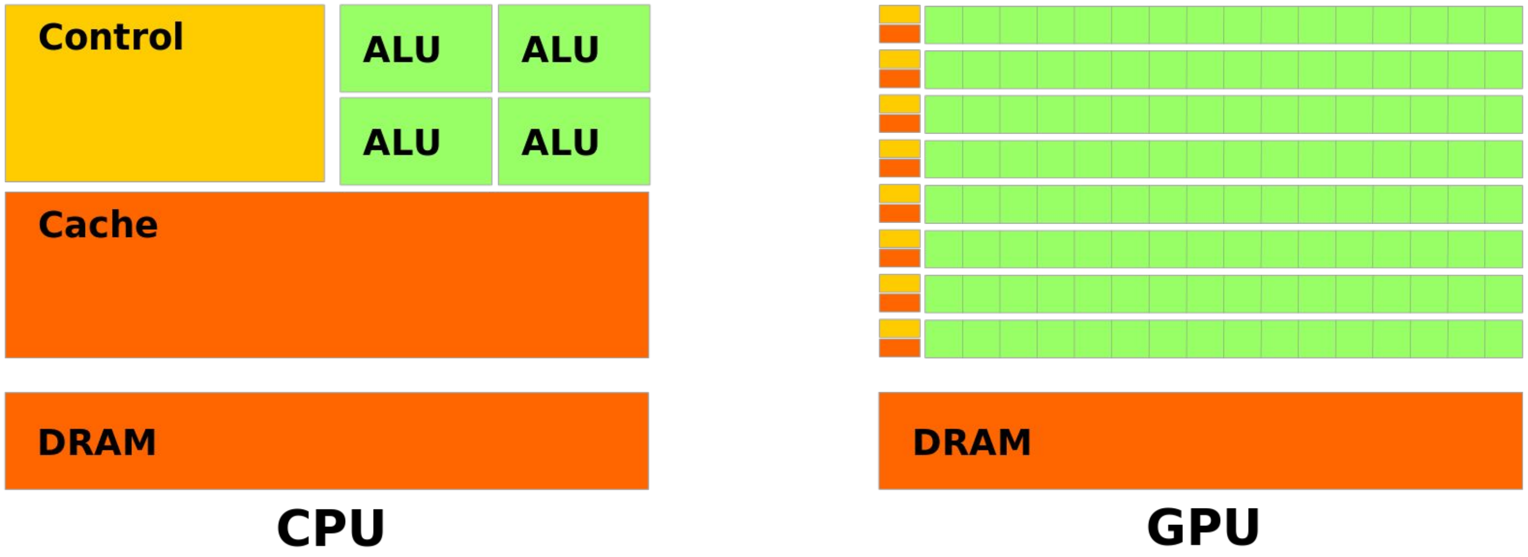
1. Why do we need to understand GPUs?
2. **GPU hardware and CUDA.**
3. Practical CUDA optimisation example.
4. PyTorch CUDA bindings.

GPU hardware and CUDA

Any idea of what are CUDA cores?

GPU hardware and CUDA

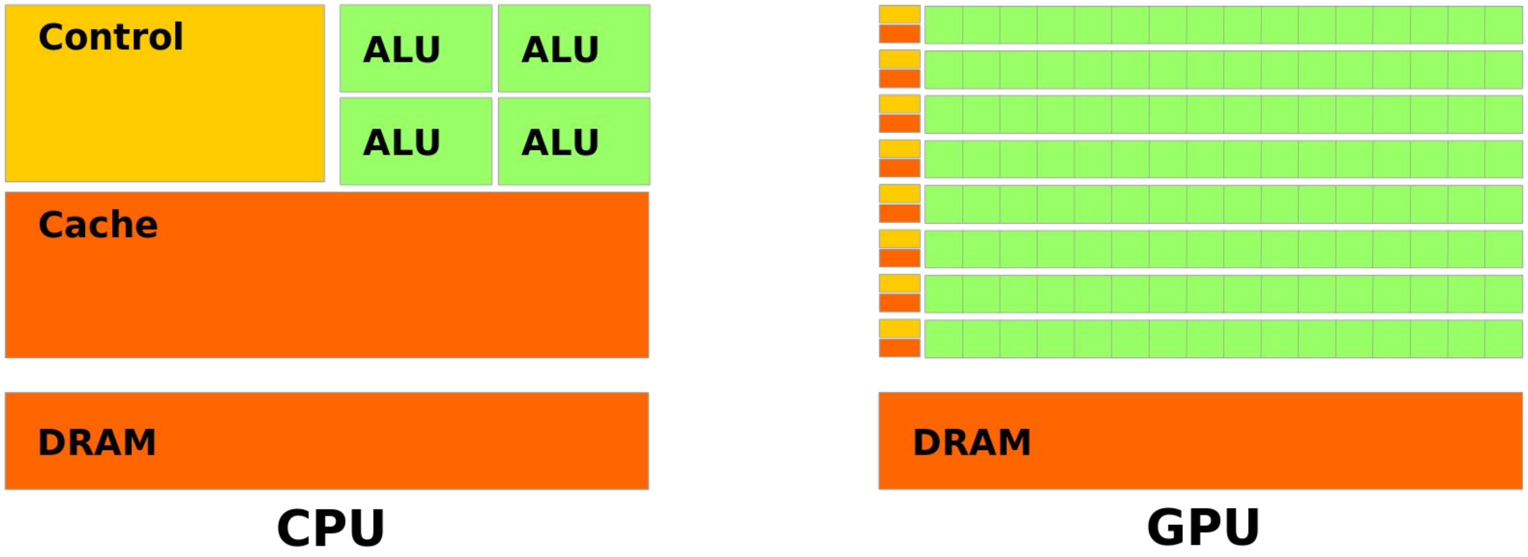
https://commons.wikimedia.org/wiki/File:Cpu-gpu.svg



Green = computational units   Orange = memory   Yellow = control

GPU hardware and CUDA

https://commons.wikimedia.org/wiki/File:Cpu-gpu.svg



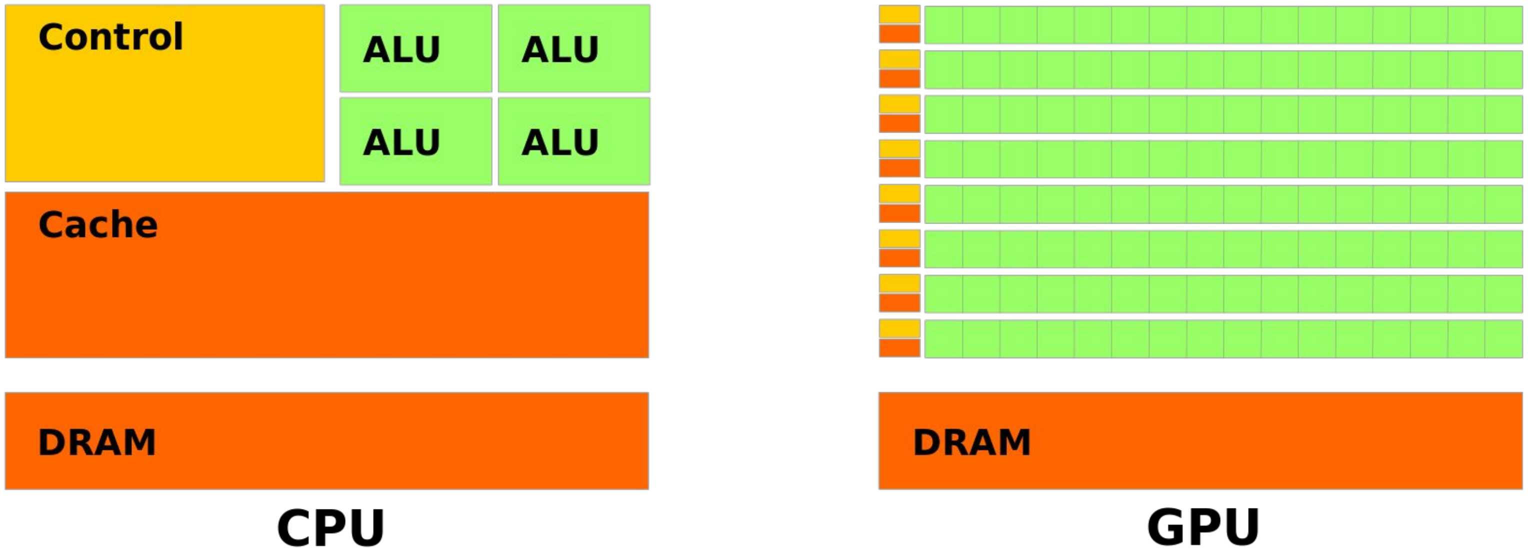
Green = computational units   Orange = memory   Yellow = control

Very basic view.

CPU computational units are bigger - “smarter”.
GPU computational units are smaller.
These units are called **“cores”**.

GPU hardware and CUDA

https://commons.wikimedia.org/wiki/File:Cpu-gpu.svg



Green = computational units   Orange = memory   Yellow = control

CPU cores must:

Perform non arithmetic ops well.
Manage out-of-order executions.

GPU cores must:

Perform arithmetic ops very well.
Stay simple and energy efficient.

Arithmetic intensity is maximised.

GPU hardware and CUDA

Arithmetic intensity is maximised.

Ampere architecture (GA102 — 10,496 CUDA cores).

GPU hardware and CUDA

Arithmetic intensity is maximised.

How are these cores managed and accessed?

Let's move one step back.

GPU hardware and CUDA

SM or Streaming Multiprocessors
Contains:

Set of cores.
Set of registers *(storing operands)*.
A chunk of shared memory *(cores of this SM)*.

GPU hardware and CUDA

The basic execution units is called **a warp**.
Contains:

32 cores.
They are executed **simultaneously** by an SM.

GPU hardware and CUDA

Nvidia Turing TU102 (e.g. RTX 2080 Ti — 4608 CUDA cores)

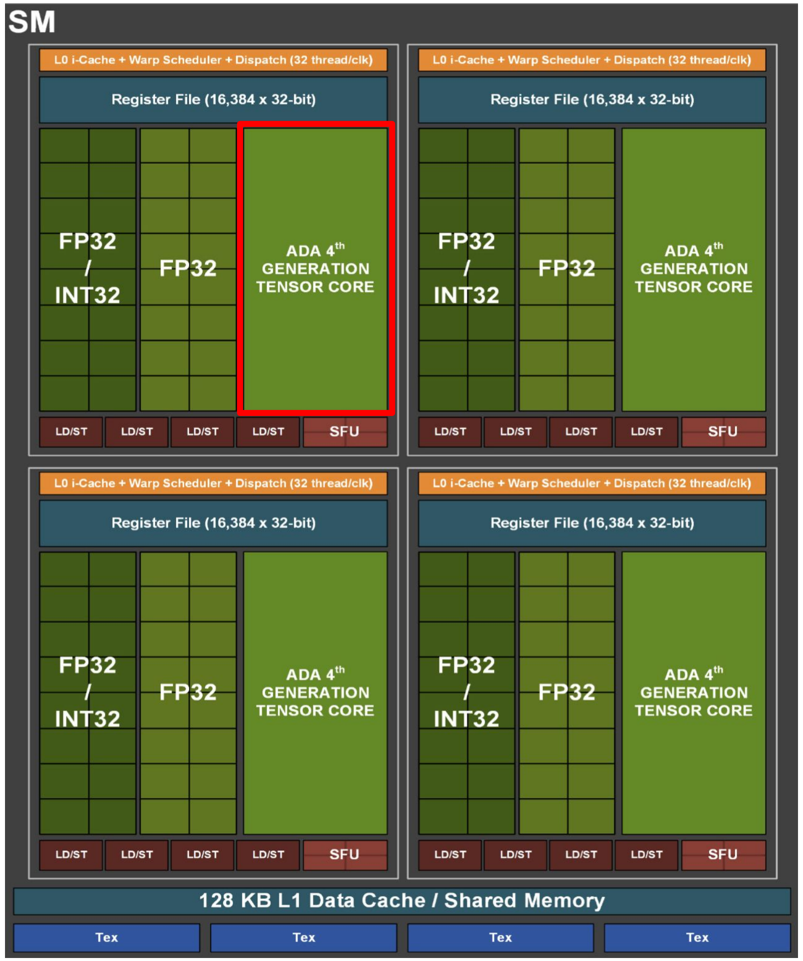
GPU hardware and CUDA

Nvidia Ampere GA102 (e.g. RTX 3090 — 10,496 CUDA cores)

GPU hardware and CUDA

Nvidia Ada Lovelace AD102 (e.g. H100 or RTX 4090 (smaller) — 18,432 CUDA cores)

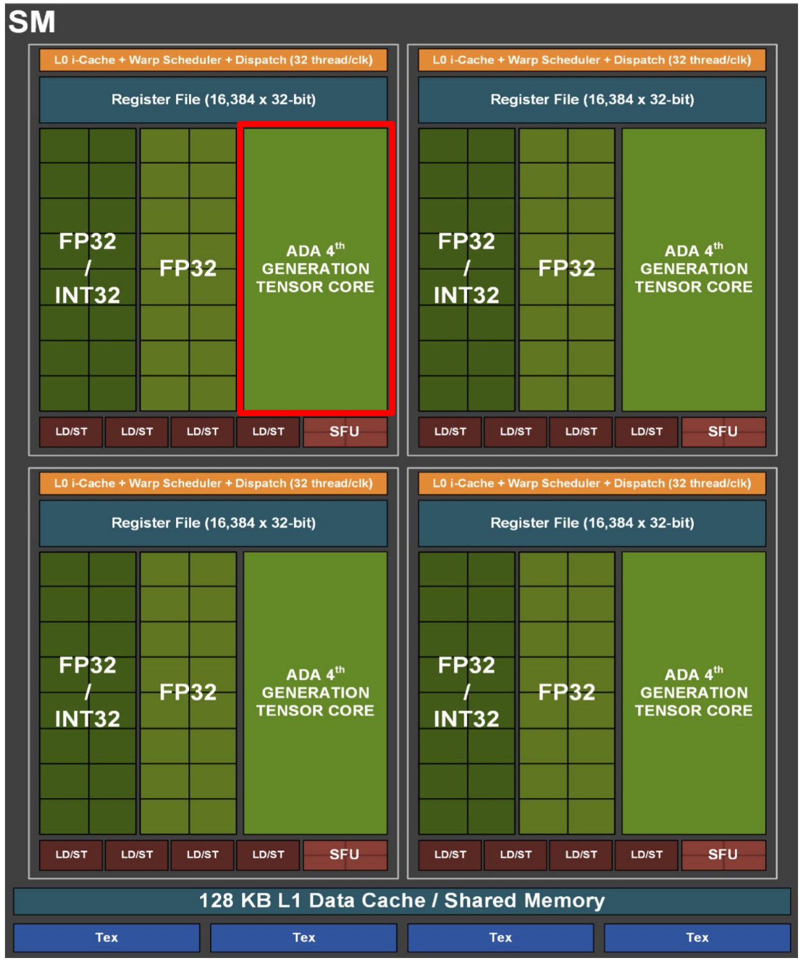
GPU hardware and CUDA



Ada Lovelace SM

Tensor cores are CUDA cores on steroïds.

GPU hardware and CUDA



Ada Lovelace SM
(4 tensor cores per SM)

Tensor cores are CUDA cores on steroïds.

In one GPU clock, a CUDA core can:
fp32 — x += y * z

In one GPU clock, a Tensor core can:
(Turing architecture)
fp16 — (4*4) x += y * z

Each tensor core can perform 1 matrix multiply-accumulate operation per GPU clock.

That's 16 times more operations per GPU clock.

GPU hardware and CUDA

What about CUDA programing?
And PyTorch?
And Tensorflow?

GPU hardware and CUDA

You favorite framework is just communicating with your GPU' SMs.

```python
# PyTorch
torch.mm(x,x.T)

# Tensorflow
tf.linalg.matmul(x, x, transpose_b=True)
```

GPU hardware and CUDA

CUDA or "Compute Unified Device Architecture", **merges** a **parallel computing platform** *(which we just saw)* with a **programming model** *(which we are about to see)*.

NVIDIA
CUDA

Hardware is nothing without a good software — right AMD?

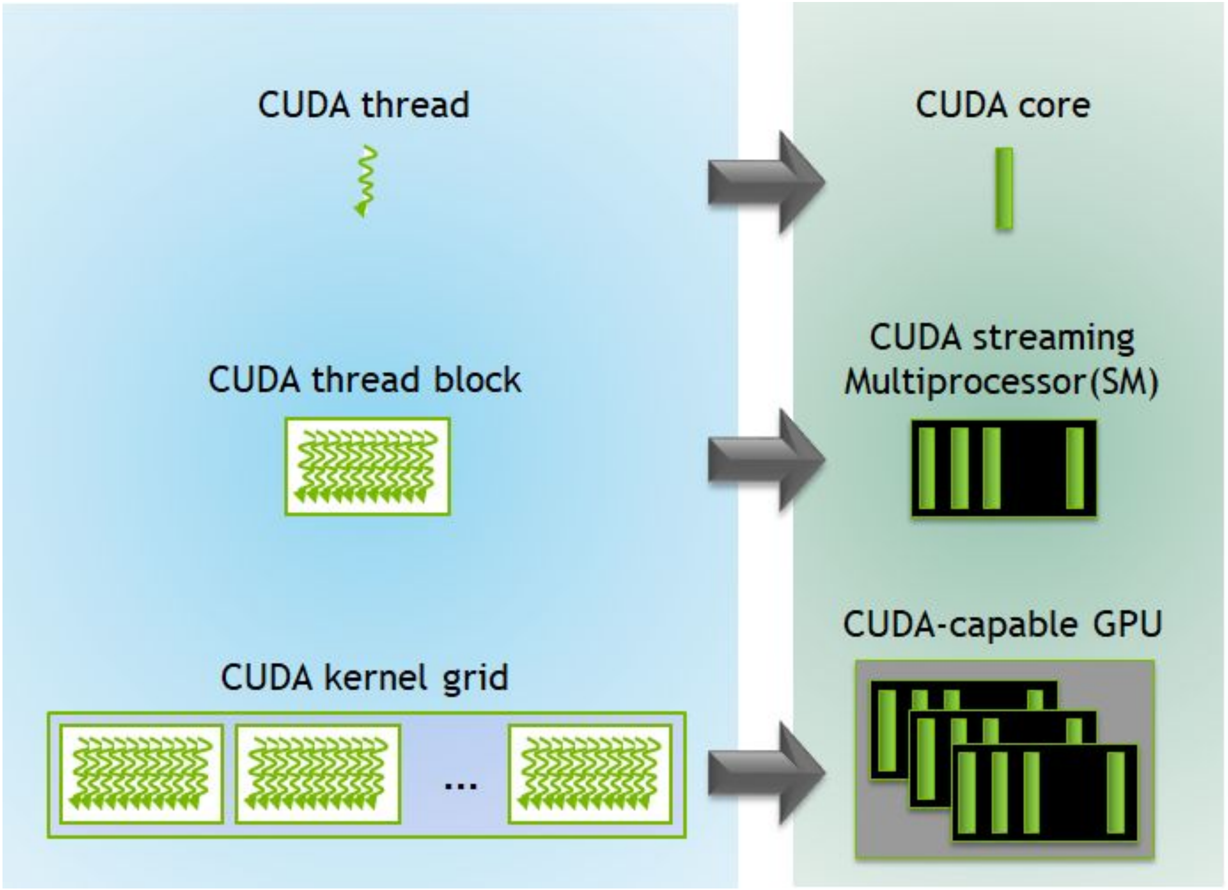
GPU hardware and CUDA

The CUDA programming model has three foundational concepts:

1. A hierarchy of thread groups (associated to kernels).
2. An ensemble of shared memories.
3. Barrier synchronization.

GPU hardware and CUDA

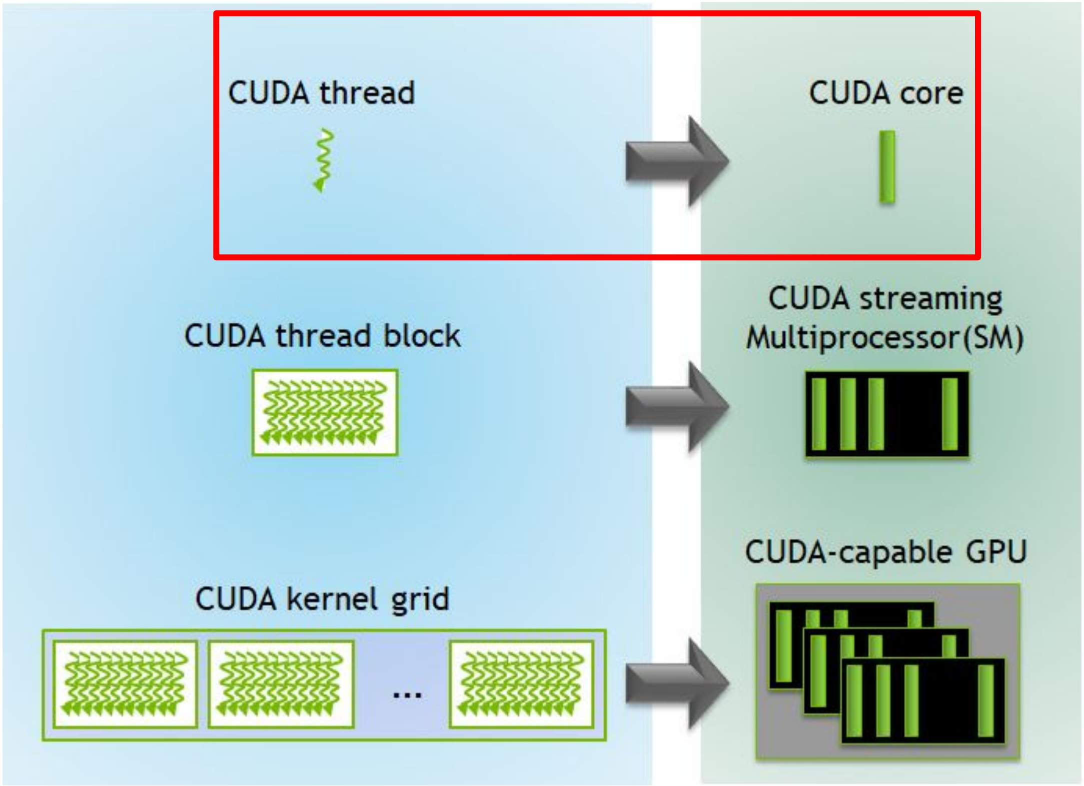
A hierarchy of thread groups.



https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/

GPU hardware and CUDA

A hierarchy of thread groups.



Not strictly true. A CUDA thread is an abstract entity that represents the execution of the kernel, it can represent a CUDA core or another logical unit.

GPU hardware and CUDA

A hierarchy of thread groups.

A kernel is a function that compiles to run on a special device.

In CUDA, a kernel is a function that will run on a certain configuration of grid / blocks / threads. These architecture information are given in the invocation of the function.

```
# Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    # Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
}
```

GPU hardware and CUDA

A hierarchy of thread groups.

threads can be identified in a 1D, 2D or 3D manner thanks to threadIdx.

This is particularly useful when manipulating vectors, matrices or volumes. This affects the corresponding thread block which also becomes 1D, 2D or 3D.

```
# Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    # Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);

}
```

GPU hardware and CUDA

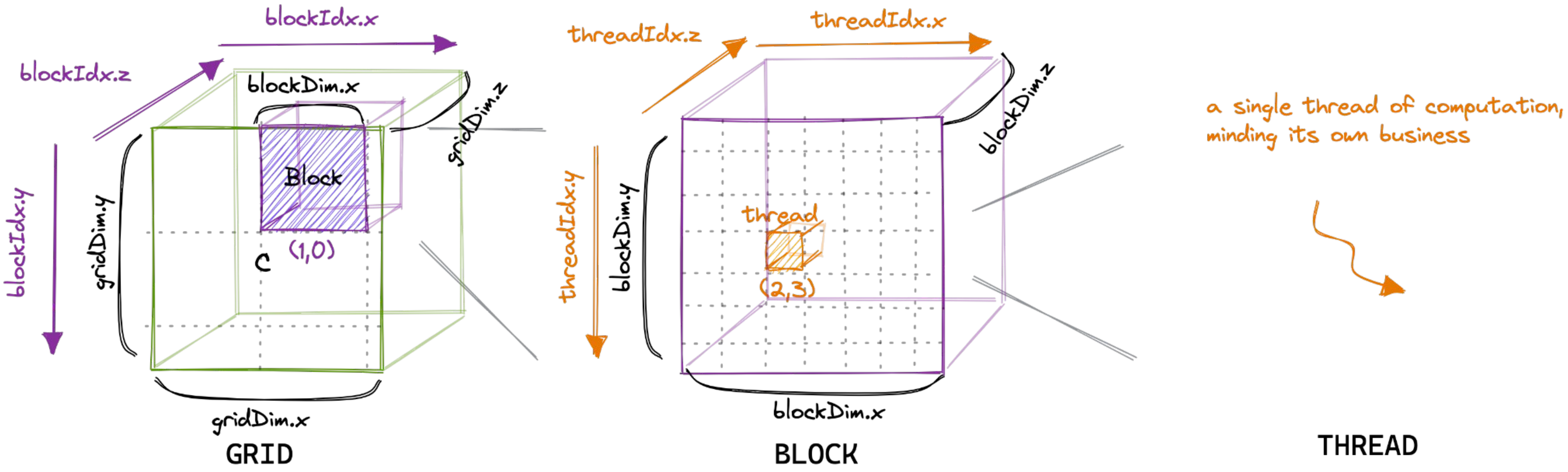
A hierarchy of thread groups.

All threads of a block resides on the same SM and share the same resources.
A single block **can't have more than 1,024 threads!**

But we can have **multiple blocks of 1,024 threads!**
Blocks are also organized in a 1D, 2D, 3D fashion (also called grids).

GPU hardware and CUDA

A hierarchy of thread groups **(summary)**.



GPU hardware and CUDA

A hierarchy of thread groups.

But we can have **multiple blocks of 1,024 threads!**

Blocks are also organized in a 1D, 2D, 3D fashion (also called grids).

```
# Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    # Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```

GPU hardware and CUDA

A hierarchy of thread groups.

But we can have **multiple blocks of 1,024 threads!**

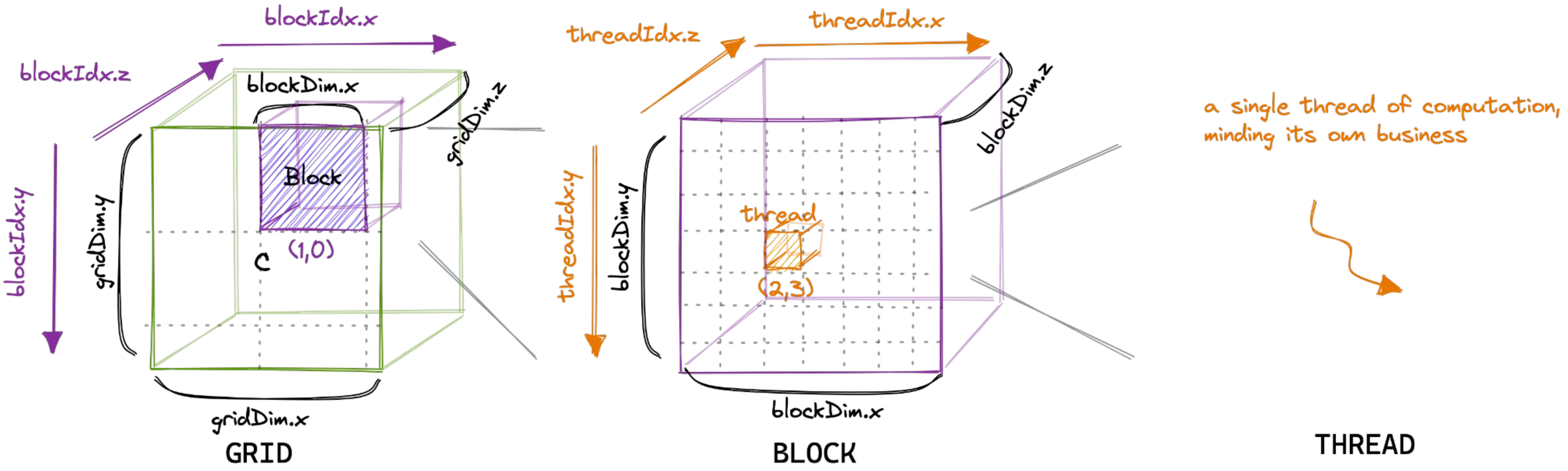
Blocks are also organized in a 1D, 2D, 3D fashion (also called grids).

```
# Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    # Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```

SegFault if N is not a multiple of 16!

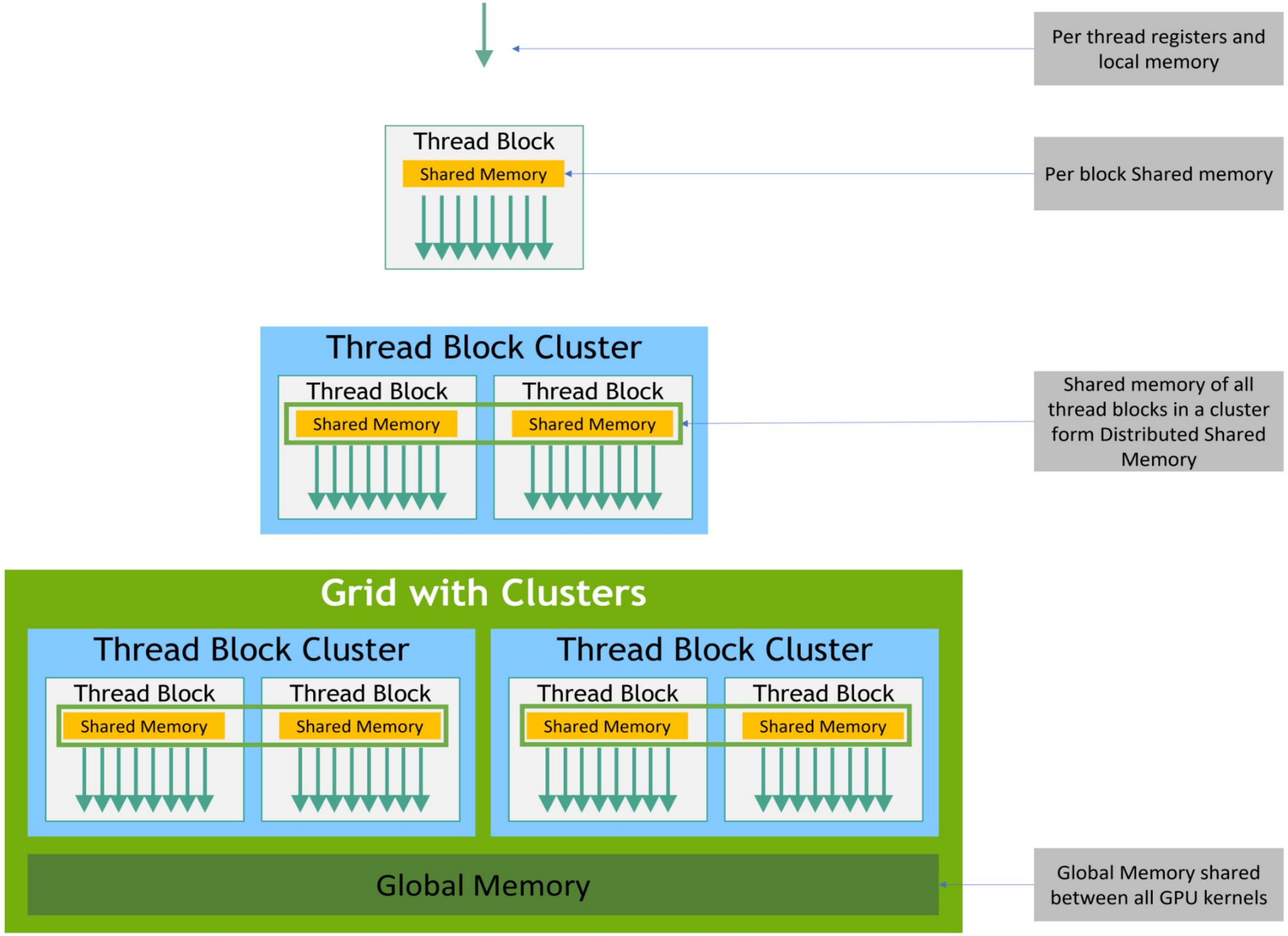
GPU hardware and CUDA

A hierarchy of thread groups **(summary)**.



GPU hardware and CUDA

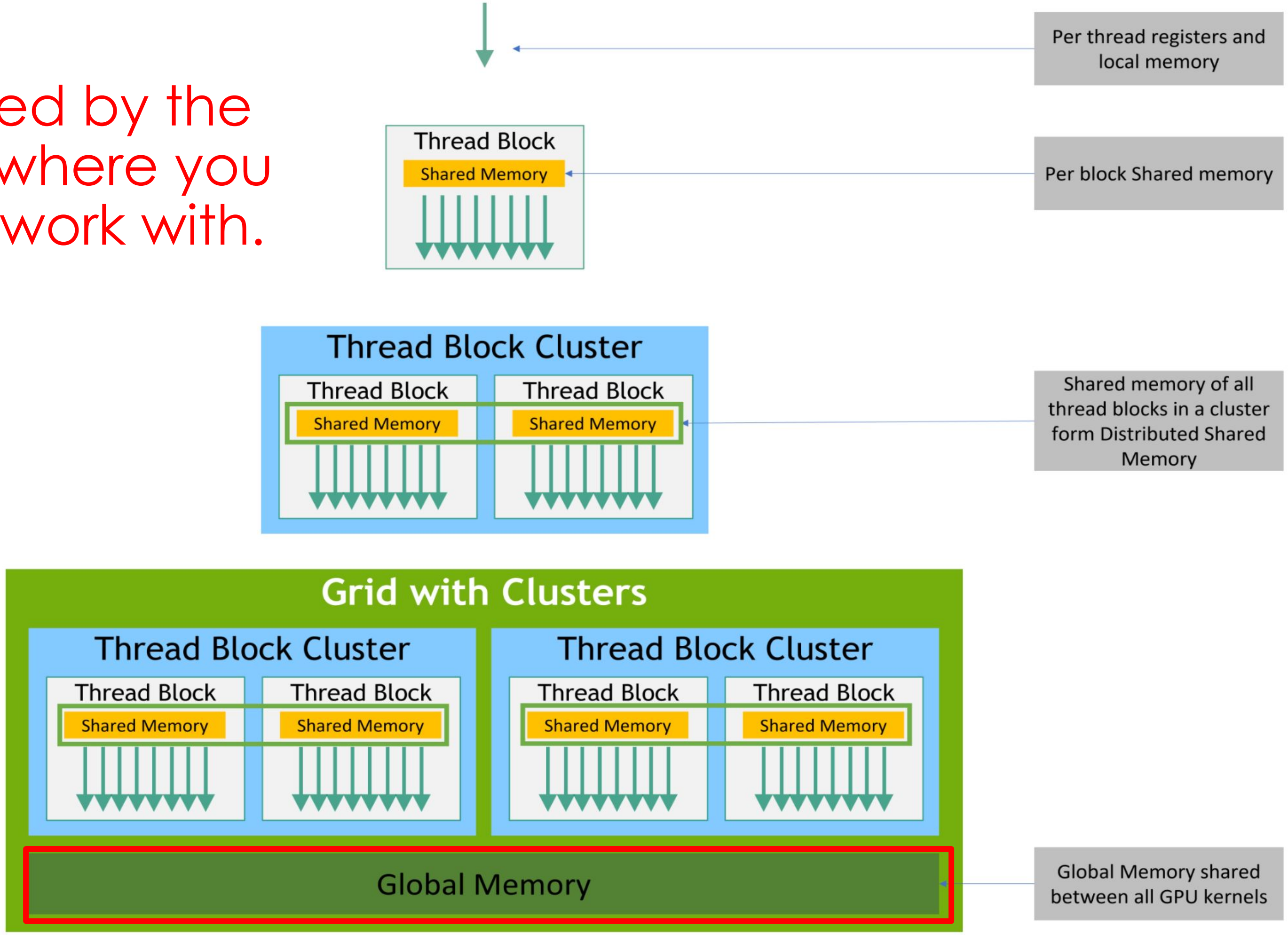
An ensemble of shared memories.



GPU hardware and CUDA

An ensemble of shared memories.

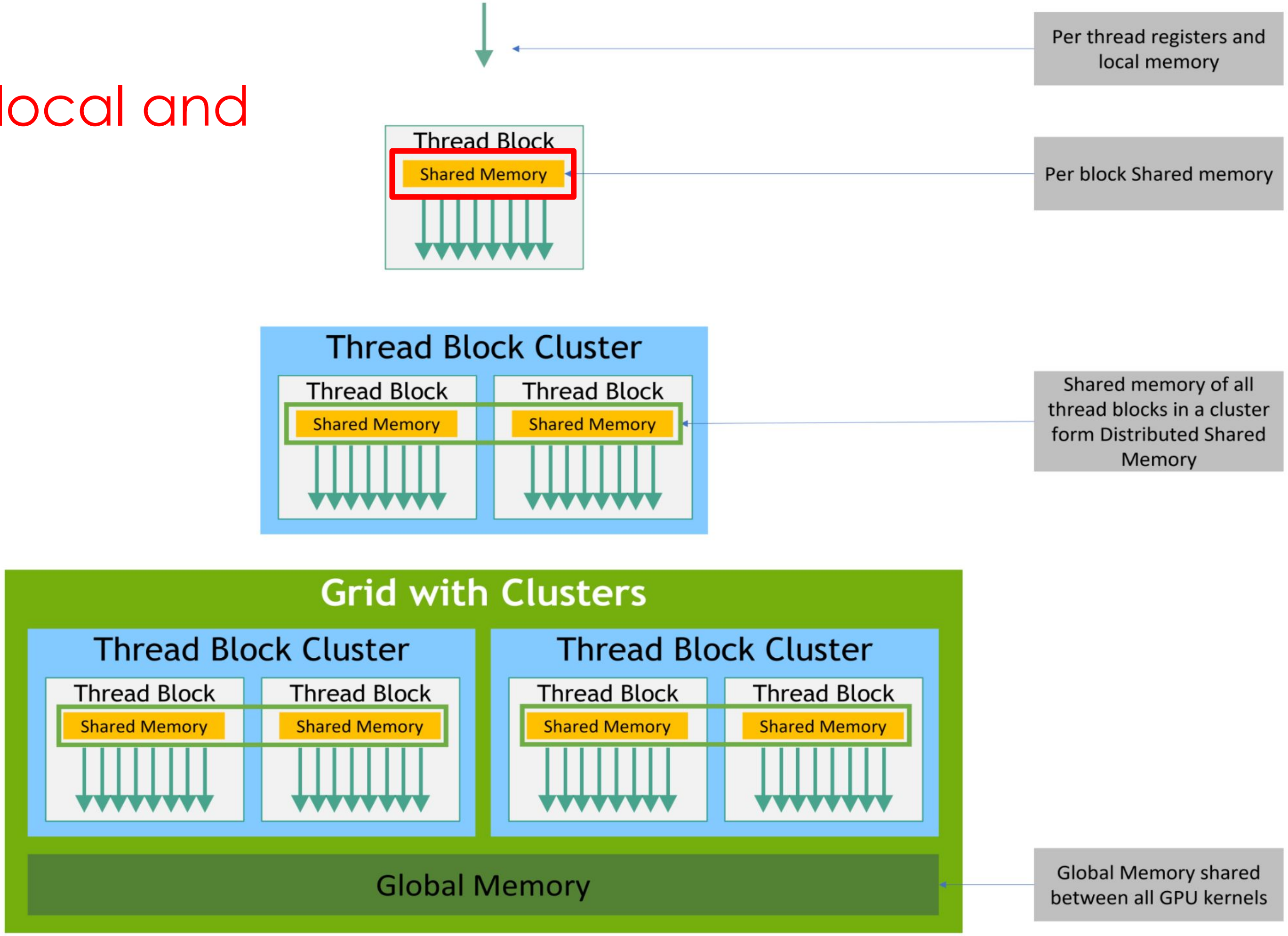
Usually manipulated by the Host (CPU). This is where you copy the data to work with.



GPU hardware and CUDA

An ensemble of shared memories.

Much faster than local and global memories.

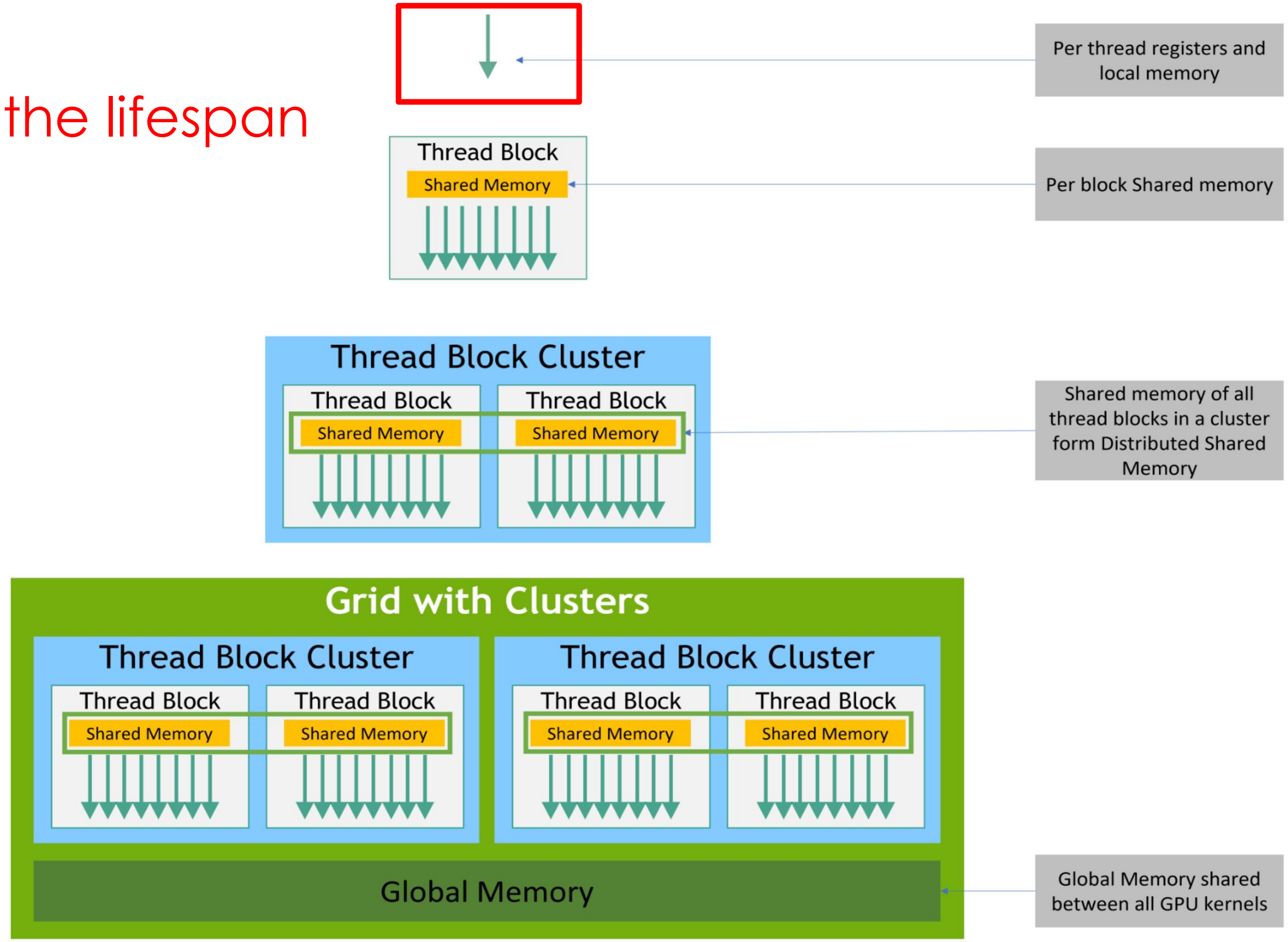


https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#shared-memory

GPU hardware and CUDA

An ensemble of shared memories.

Only exists during the lifespan of a thread.



GPU hardware and CUDA

Barrier synchronization.

All threads of a block are executed asynchronously.

You are guaranteed that all threads will finish before getting the result, but there is no guarantees on the order of execution.

GPU hardware and CUDA

Barrier synchronization.

All threads of a block are executed asynchronously.

You are guaranteed that all threads will finish before getting the result, but there is no guarantees on the order of execution.

What if we need to share partial results?

GPU hardware and CUDA

Barrier synchronization.

__syncthreads() acts as a barrier at the block level.

```
__global__ void globFunction(int *arr, int N)
{
    __shared__ int local_array[THREADS_PER_BLOCK];  # local block memory cache
    int idx = blockIdx.x* blockDim.x+ threadIdx.x;

    # ...calculate results
    local_array[threadIdx.x] = results;

    # synchronize the local threads writing to the local memory cache
    __syncthreads();

    # read the results of another thread in the current thread
    int val = local_array[(threadIdx.x + 1) % THREADS_PER_BLOCK];

    # write back the value to global memory
    arr[idx] = val;
}
```

Roadmap for Today

1. Why do we need to understand GPUs?
2. GPU hardware and CUDA.
3. **Practical CUDA optimisation example.**
4. PyTorch CUDA bindings.

Practical CUDA optimisation example

Implementing a Single precision GEneral Matrix Multiply (SGEMM).

$$C_{i,j} = \sum_{k=1}^{N} A_{i,k} \cdot B_{k,j}, \quad \forall i, j \in 1, N$$

https://siboehm.com/articles/22/CUDA-MMM

Practical CUDA optimisation example

Implementing a Single precision GEneral Matrix Multiply (SGEMM).
Naïve solution.

```
__global__ void sgemm_naive(int M, int N, int K, const float *A,
                            const float *B, float *C) {

  // compute position in C that this thread is responsible for
  const uint x = blockIdx.x * blockDim.x + threadIdx.x;
  const uint y = blockIdx.y * blockDim.y + threadIdx.y;

  // `if` condition is necessary for when M or N aren't multiples of 32.
  if (x < M && y < N) {
    float tmp = 0.0;
    for (int i = 0; i < K; ++i) {
      tmp += A[x * K + i] * B[i * N + y];
    }
    // C = α*(A@B)+β*C
    C[x * N + y] = tmp;
  }
}

int main(int argc, char *argv[]){
  // create as many blocks as necessary to map all of C
  dim3 gridDim(CEIL_DIV(M, 32), CEIL_DIV(N, 32), 1);
  // 32 * 32 = 1024 thread per block
  dim3 blockDim(32, 32, 1);
  // launch the asynchronous execution of the kernel on the device
  // The function call returns immediately on the host
  sgemm_naive<<<gridDim, blockDim>>>(M, N, K, A, B, C);
}
```

One thread is responsible for one element of C

Practical CUDA optimisation example

Implementing a Single precision GEneral Matrix Multiply (SGEMM).
Naïve solution.

Kernel	GFLOPs/s
1: Naive	309.0

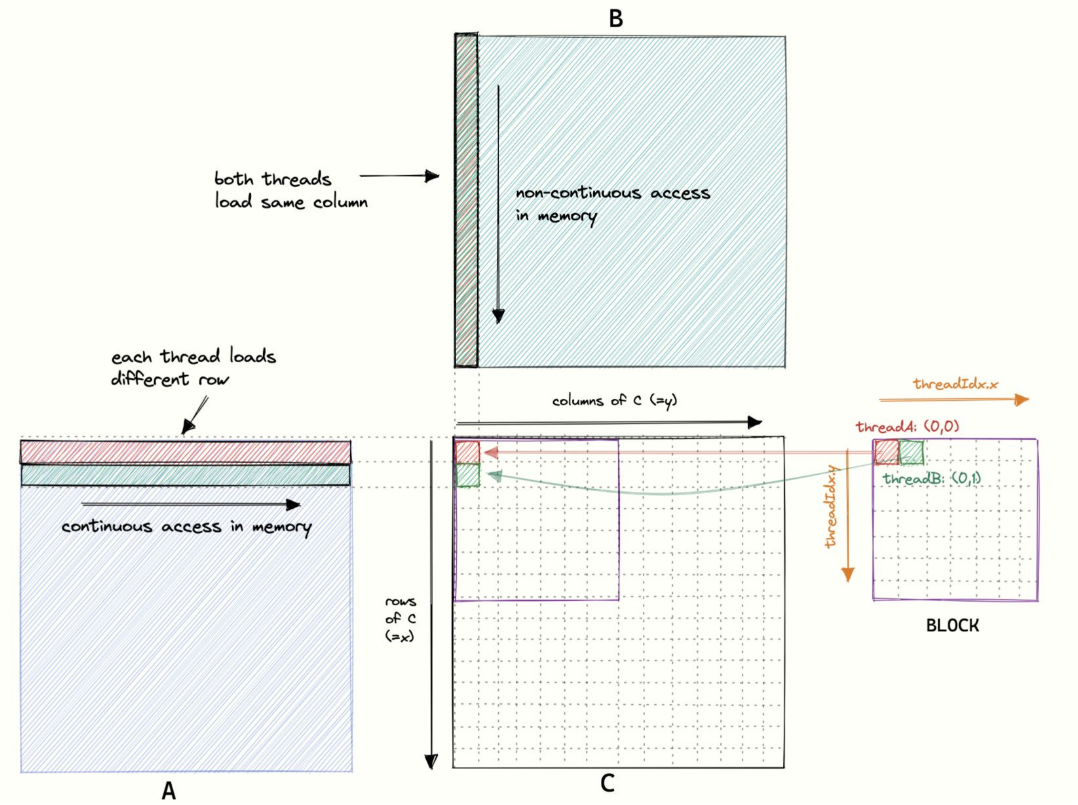
```
__global__ void sgemm_naive(int M, int N, int K, const float *A,
                            const float *B, float *C) {

  // compute position in C that this thread is responsible for
  const uint x = blockIdx.x * blockDim.x + threadIdx.x;
  const uint y = blockIdx.y * blockDim.y + threadIdx.y;

  // `if` condition is necessary for when M or N aren't multiples of 32.
  if (x < M && y < N) {
    float tmp = 0.0;
    for (int i = 0; i < K; ++i) {
      tmp += A[x * K + i] * B[i * N + y];
    }
    // C = α*(A@B)+β*C
    C[x * N + y] = tmp;
  }
}

int main(int argc, char *argv[]){
  // create as many blocks as necessary to map all of C
  dim3 gridDim(CEIL_DIV(M, 32), CEIL_DIV(N, 32), 1);
  // 32 * 32 = 1024 thread per block
  dim3 blockDim(32, 32, 1);
  // launch the asynchronous execution of the kernel on the device
  // The function call returns immediately on the host
  sgemm_naive<<<gridDim, blockDim>>>(M, N, K, A, B, C);
}
```

One thread is responsible for one element of C

Practical CUDA optimisation example

Implementing a Single precision GEneral Matrix Multiply (SGEMM).
Better memory access.

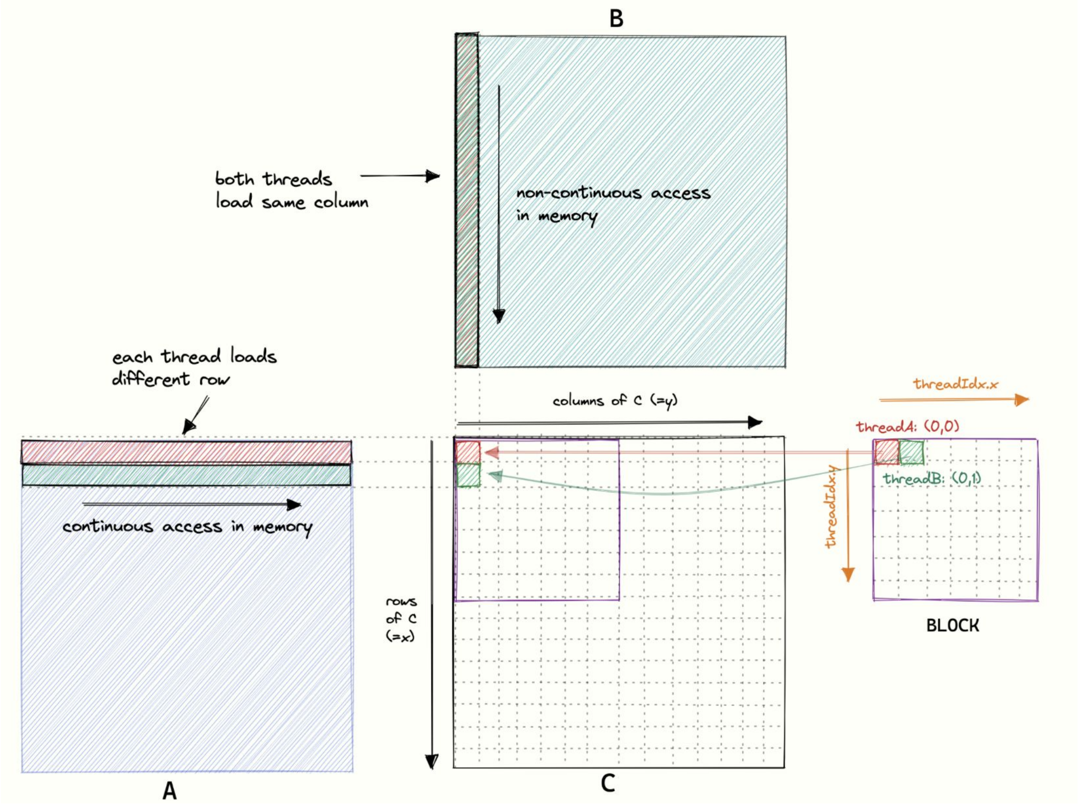
One of the memory access is non-continuous due to the storage of the matrix i.e. slow



Practical CUDA optimisation example

Implementing a Single precision GEneral Matrix Multiply (SGEMM).
Better memory access.

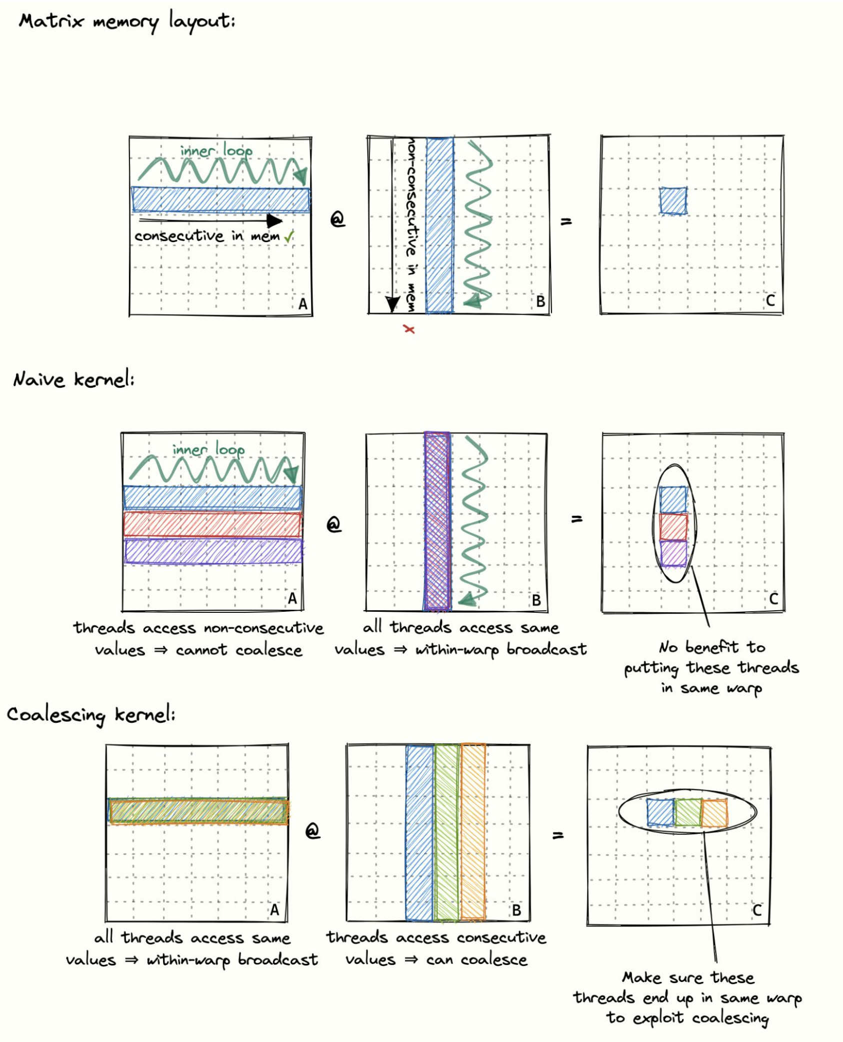
Sequential memory accesses by threads in a warp can be executed as one.



Practical CUDA optimisation example

Implementing a Single precision GEneral Matrix Multiply (SGEMM).

Better memory access.



Practical CUDA optimisation example

Implementing a Single precision GEneral Matrix Multiply (SGEMM).

Better memory access.

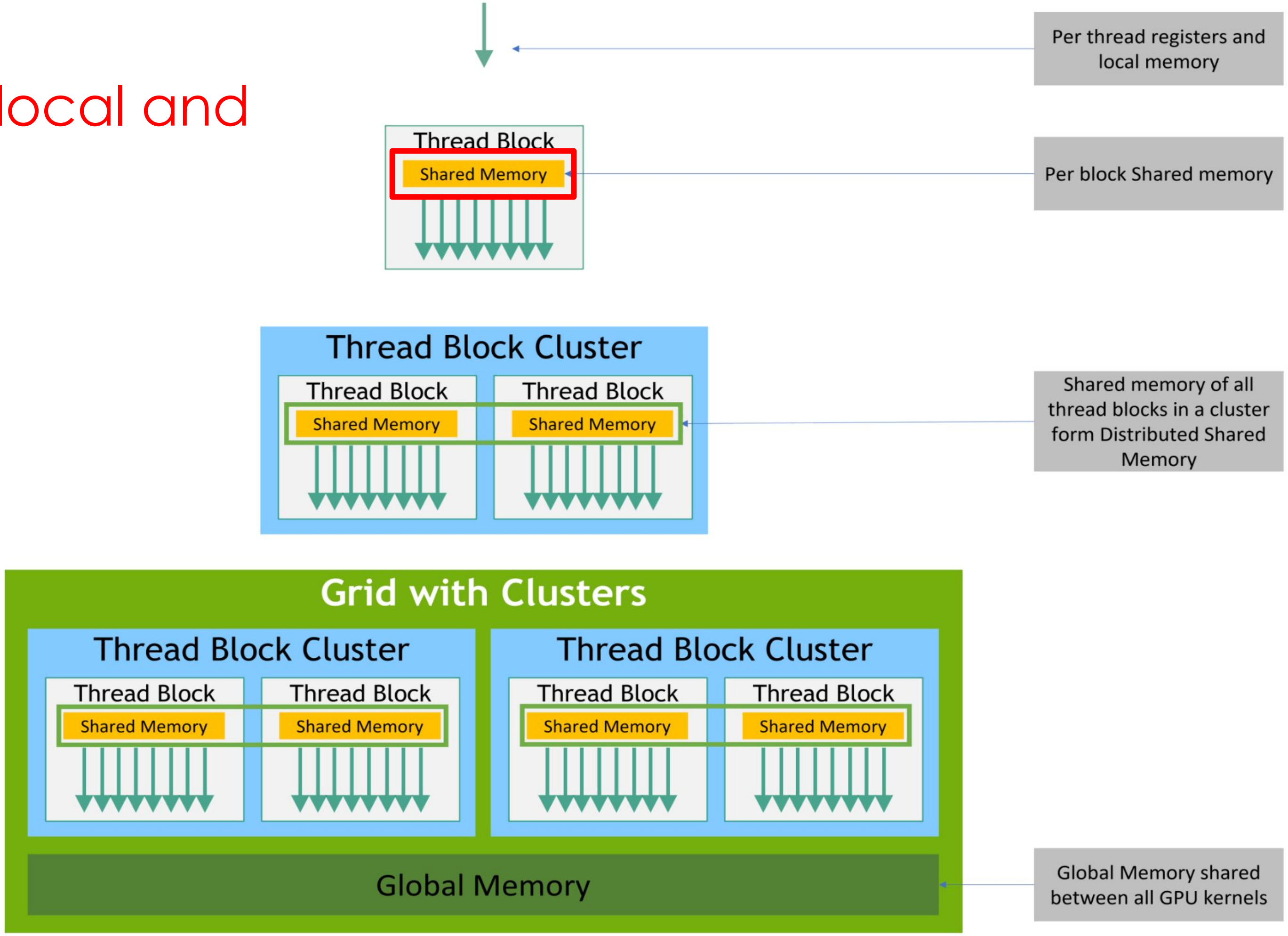
Kernel	GFLOPs/s
1: Naive	309.0
2: GMEM Coalescing	1986.5

```
__global__ void sgemm_naive(int M, int N, int K, const float *A,
                            const float *B, float *C) {

  const int x = blockIdx.x * BLOCKSIZE + (threadIdx.x / BLOCKSIZE);
  const int y = blockIdx.y * BLOCKSIZE + (threadIdx.x % BLOCKSIZE);

  if (x < M && y < N) {
    float tmp = 0.0;
    for (int i = 0; i < K; ++i) {
      tmp += A[x * K + i] * B[i * N + y];
    }
    C[x * N + y] = tmp
  }
}

int main(int argc, char *argv[]){
  // create as many blocks as necessary to map all of C
  dim3 gridDim(CEIL_DIV(M, 32), CEIL_DIV(N, 32), 1);
  // 32 * 32 = 1024 thread per block
  dim3 blockDim(32, 32, 1);
  // launch the asynchronous execution of the kernel on the device
  // The function call returns immediately on the host
  sgemm_naive<<<gridDim, blockDim>>>(M, N, K, A, B, C);
}
```

Practical CUDA optimisation example

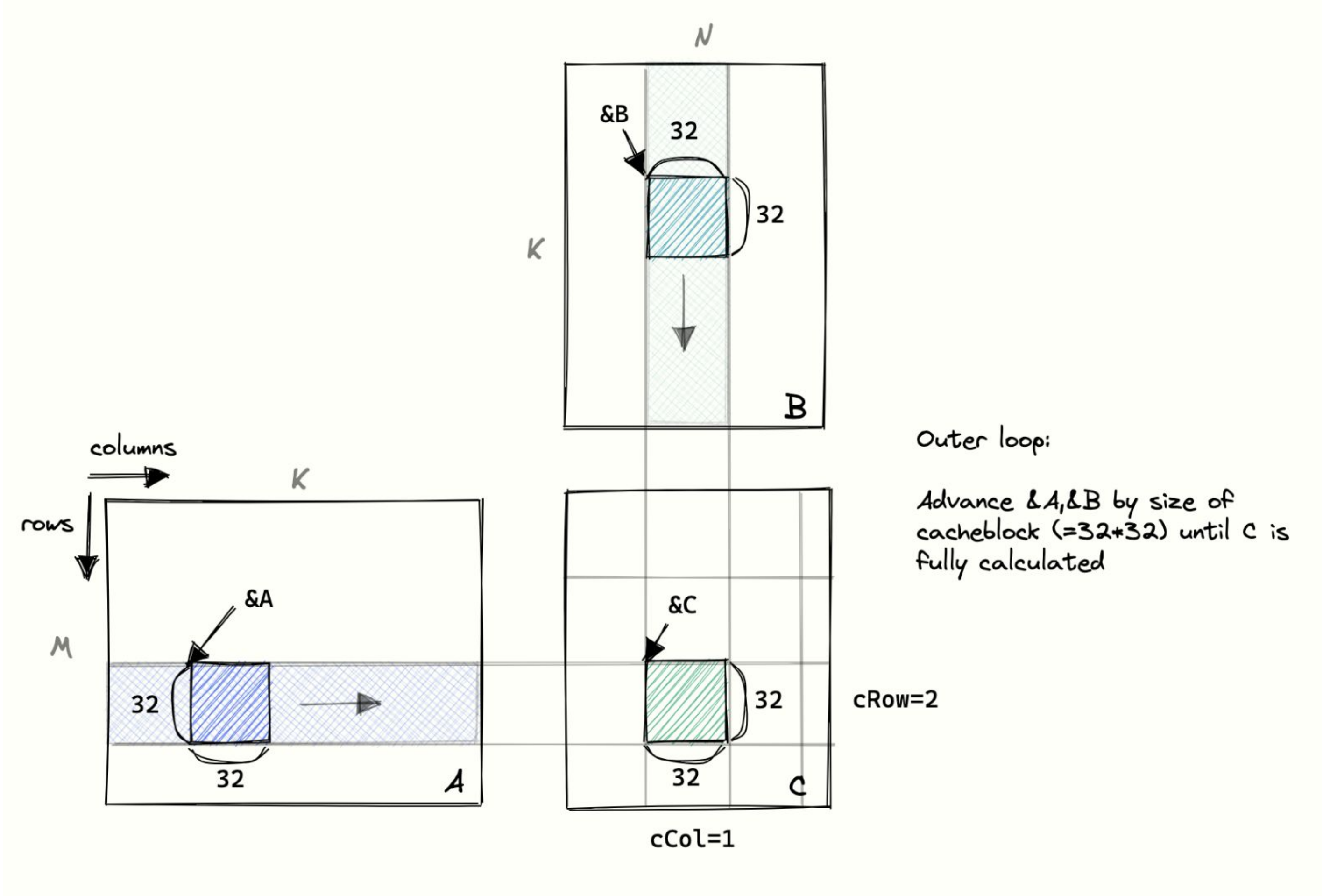
Implementing a Single precision GEneral Matrix Multiply (SGEMM).
Using shared memory.

Much faster than local and global memories.



Practical CUDA optimisation example

Implementing a Single precision GEneral Matrix Multiply (SGEMM).
Using shared memory.



Practical CUDA optimisation example

1. Allocate shared memory.

2. Copy from global to shared memory **using threads**.

3. Compute the product with shared memory elements.

```
__global__ void sgemm_shared_mem_block(int M, int N, int K,
                                       const float *A, const float *B,
                                       float *C) {
  // the output block that we want to compute in this threadblock
  const uint cRow = blockIdx.x;
  const uint cCol = blockIdx.y;

  // allocate buffer for current block in fast shared mem
  // shared mem is shared between all threads in a block
  __shared__ float As[BLOCKSIZE * BLOCKSIZE];
  __shared__ float Bs[BLOCKSIZE * BLOCKSIZE];

  // the inner row & col that we're accessing in this thread
  const uint threadCol = threadIdx.x % BLOCKSIZE;
  const uint threadRow = threadIdx.x / BLOCKSIZE;

  // advance pointers to the starting positions
  A += cRow * BLOCKSIZE * K;                    // row=cRow, col=0
  B += cCol * BLOCKSIZE;                        // row=0, col=cCol
  C += cRow * BLOCKSIZE * N + cCol * BLOCKSIZE; // row=cRow, col=cCol

  float tmp = 0.0;
  for (int bkIdx = 0; bkIdx < K; bkIdx += BLOCKSIZE) {
    // Have each thread load one of the elements in A & B
    // Make the threadCol (=threadIdx.x) the consecutive index
    // to allow global memory access coalescing
    As[threadRow * BLOCKSIZE + threadCol] = A[threadRow * K + threadCol];
    Bs[threadRow * BLOCKSIZE + threadCol] = B[threadRow * N + threadCol];

    // block threads in this block until cache is fully populated
    __syncthreads();
    A += BLOCKSIZE;
    B += BLOCKSIZE * N;

    // execute the dotproduct on the currently cached block
    for (int dotIdx = 0; dotIdx < BLOCKSIZE; ++dotIdx) {
      tmp += As[threadRow * BLOCKSIZE + dotIdx] *
             Bs[dotIdx * BLOCKSIZE + threadCol];
    }
    // need to sync again at the end, to avoid faster threads
    // fetching the next block into the cache before slower threads are done
    __syncthreads();
  }
  C[threadRow * N + threadCol] = tmp
}
```

Kernel	GFLOPs/s
1: Naive	309.0
2: GMEM Coalescing	1986.5
3: SMEM Caching	2980.3

Practical CUDA optimisation example

Kernel	GFLOPs/s
1: Naive	309.0
2: GMEM Coalescing	1986.5
3: SMEM Caching	2980.3
4: 1D Blocktiling	8474.7
5: 2D Blocktiling	15971.7
6: Vectorized Mem Access	18237.3
9: Autotuning	19721.0
10: Warptiling	21779.3
0: cuBLAS	23249.6

https://siboehm.com/articles/22/CUDA-MMM

Roadmap for Today

1. Why do we need to understand GPUs?
2. GPU hardware and CUDA.
3. Practical CUDA optimisation example.
4. **PyTorch CUDA bindings.**

PyTorch CUDA bindings.

PyTorch provides two ways of binding C++ code:
compilation ahead of time or **just in time (JIT)**.

1. Write your CUDA / C++ files.
2. Write the bindings to python with pybind11.
3. Use JIT or setuptool to compile.

PyTorch CUDA bindings.

1. Write your CUDA / C++ files.

```
#include <torch/extension.h>

// CUDA kernel. Each thread takes care of one element of c
__global__ void vecAdd(float *a, float *b, float *c, int n)
{
    // Get our global thread ID
    int id = blockIdx.x*blockDim.x+threadIdx.x;
    // Make sure we do not go out of bounds
    if (id < n)
        c[id] = a[id] + b[id];
}

torch::Tensor custom_add_vectors_cuda(torch::Tensor input1, torch::Tensor input2) {
  // Get the number of elements in the input vectors.
  int64_t num_elements = input1.numel();
  // Allocate a temporary output tensor.
  torch::Tensor output = torch::empty({num_elements}, torch::dtype(torch::kFloat32).device(torch::kCUDA, 0));
  // Launch the CUDA kernel to perform the vector addition operation.
  vecAdd<<<1, 1024>>>(input1.data_ptr<float>(), input2.data_ptr<float>(), output.data_ptr<float>(), num_elements);
  // Synchronize the GPU to ensure that the kernel has finished executing.
  torch::cuda::synchronize();
  // Return the output tensor.
  return output;
}

PYBIND11_MODULE(TORCH_EXTENSION_NAME, m) {
  m.def("custom_add_vectors_cuda", &custom_add_vectors_cuda);
}
```

PyTorch CUDA bindings.

2. Write the bindings to python with pybind11.

```
#include <torch/extension.h>

// CUDA kernel. Each thread takes care of one element of c
__global__ void vecAdd(float *a, float *b, float *c, int n)
{
    // Get our global thread ID
    int id = blockIdx.x*blockDim.x+threadIdx.x;
    // Make sure we do not go out of bounds
    if (id < n)
        c[id] = a[id] + b[id];
}

torch::Tensor custom_add_vectors_cuda(torch::Tensor input1, torch::Tensor input2) {
  // Get the number of elements in the input vectors.
  int64_t num_elements = input1.numel();
  // Allocate a temporary output tensor.
  torch::Tensor output = torch::empty({num_elements}, torch::dtype(torch::kFloat32).device(torch::kCUDA, 0));
  // Launch the CUDA kernel to perform the vector addition operation.
  vecAdd<<<1, 1024>>>(input1.data_ptr<float>(), input2.data_ptr<float>(), output.data_ptr<float>(), num_elements);
  // Synchronize the GPU to ensure that the kernel has finished executing.
  torch::cuda::synchronize();
  // Return the output tensor.
  return output;
}

PYBIND11_MODULE(TORCH_EXTENSION_NAME, m) {
  m.def("custom_add_vectors_cuda", &custom_add_vectors_cuda);
}
```

PyTorch CUDA bindings.

3. Use JIT or setuptool to compile.

```python
import torch
from torch.utils.cpp_extension import load
# Load the custom reduce sum operation.
custom_op = load(name="custom_add_vectors_cuda", sources=['reduce_sum.cu'])
# Create an input tensor.
input1 = torch.randn(100, dtype=torch.float32, device="cuda")
input2 = torch.randn(100, dtype=torch.float32, device="cuda")
# Compute the reduce sum of the input tensor.
output = custom_op.custom_add_vectors_cuda(input1, input2)
# Print the output tensor.
print(output)
```

In brief

1. GPUs or accelerators are our main tool in DL — we must know them.
2. Nvidia GPUs share the same overall architecture.
3. Nvidia GPUs are made of SM / warp / Arithmetic cores
4. GPU cores maximises arithmetic intensity.
5. CUDA merges a parallel computing platform with a programming model.
6. Key concepts are: hierarchy of threads and memory and synchronisation.
7. Optimising your code with CUDA may lead to massive improvements.
8. PyTorch (and Tensorflow) can handle custom CUDA code.

To go beyond the lecture

1. https://siboehm.com/articles/22/CUDA-MMM
2. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
3. https://pytorch.org/tutorials/advanced/cpp_extension.html