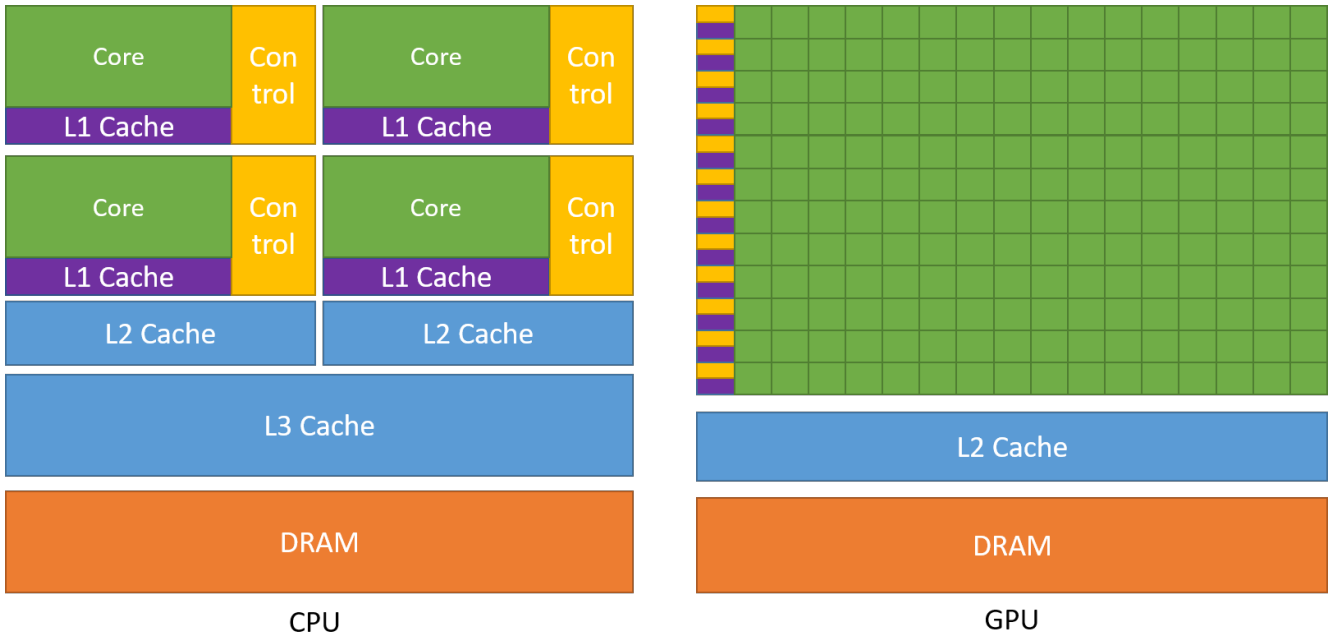
Introduction to CUDA programming

General GPU architecture

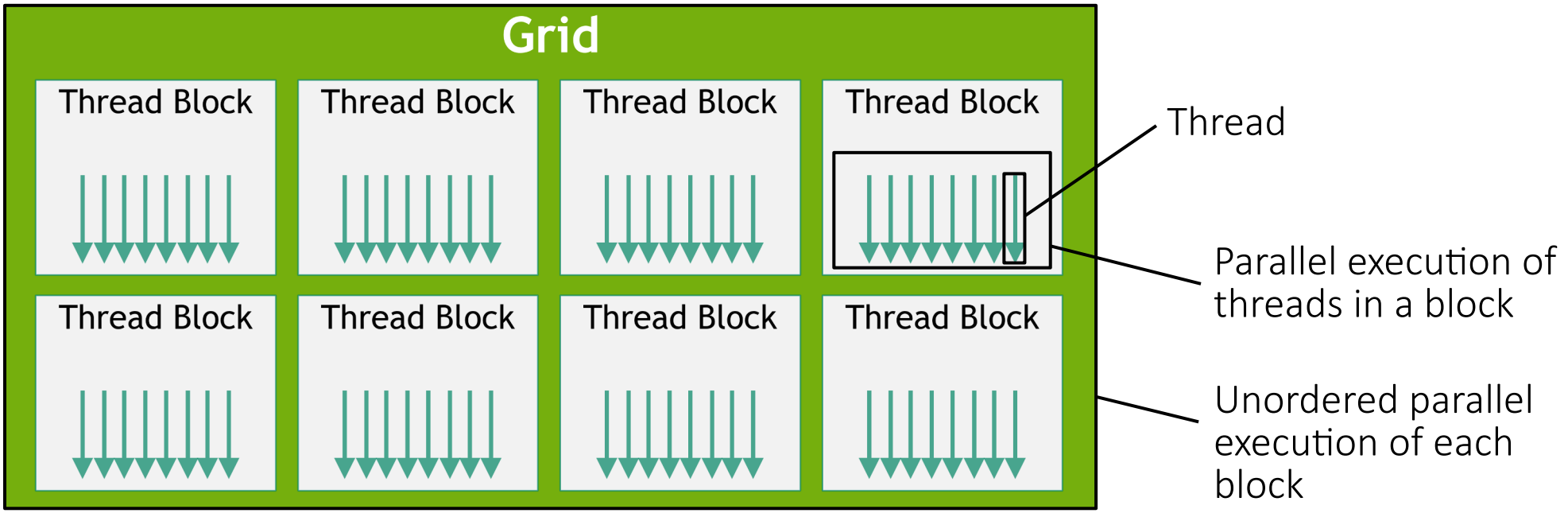
- *Many* simple cores, organised through several layers of hierarchy, with shared controls
  - RTX 4090: 16384 CUDA cores



- Efficient for highly/“embarrassingly” parallel tasks
  - Historically for 2D/3D graphics applications
  - GPGU: intensive scientific computing, deep learning…

CUDA programming model

- CUDA: **C**ompute **U**nified **D**evice **A**rchitecture
  - NVIDIA's parallel computing architecture & API
- High-level CUDA thread hierarchy:
  - ***Thread***: executed sequence of instructions
  - ***Block***: group of multiple threads
  - ***Grid***: set of blocks being executed on the GPU



C++ example: vector addition

```cpp
// main.cpp
#include <stdlib.h>
#include <stdio.h>

void add_vec(float *a, float *b, float *c, int N) {
    for(int i = 0; i < N; i++){
        c[i] = a[i] + b[i];
    }
}

int main() {
    int N = 32; // vectors of size 32
    float *a = (float*)malloc(N * sizeof(float));
    float *b = (float*)malloc(N * sizeof(float));
    float *c = (float*)malloc(N * sizeof(float));

    for(int i = 0; i < N; i++){ // a and b are {0, 1, ..., N-1}
        a[i] = (float)i;
        b[i] = (float)i;
    }

    add_vec(a, b, c, N);

    for(int i = 0; i < N; i++){
        printf("%.0f ", c[i]);
    }
}
```

independant index-based operations: can be parallelized

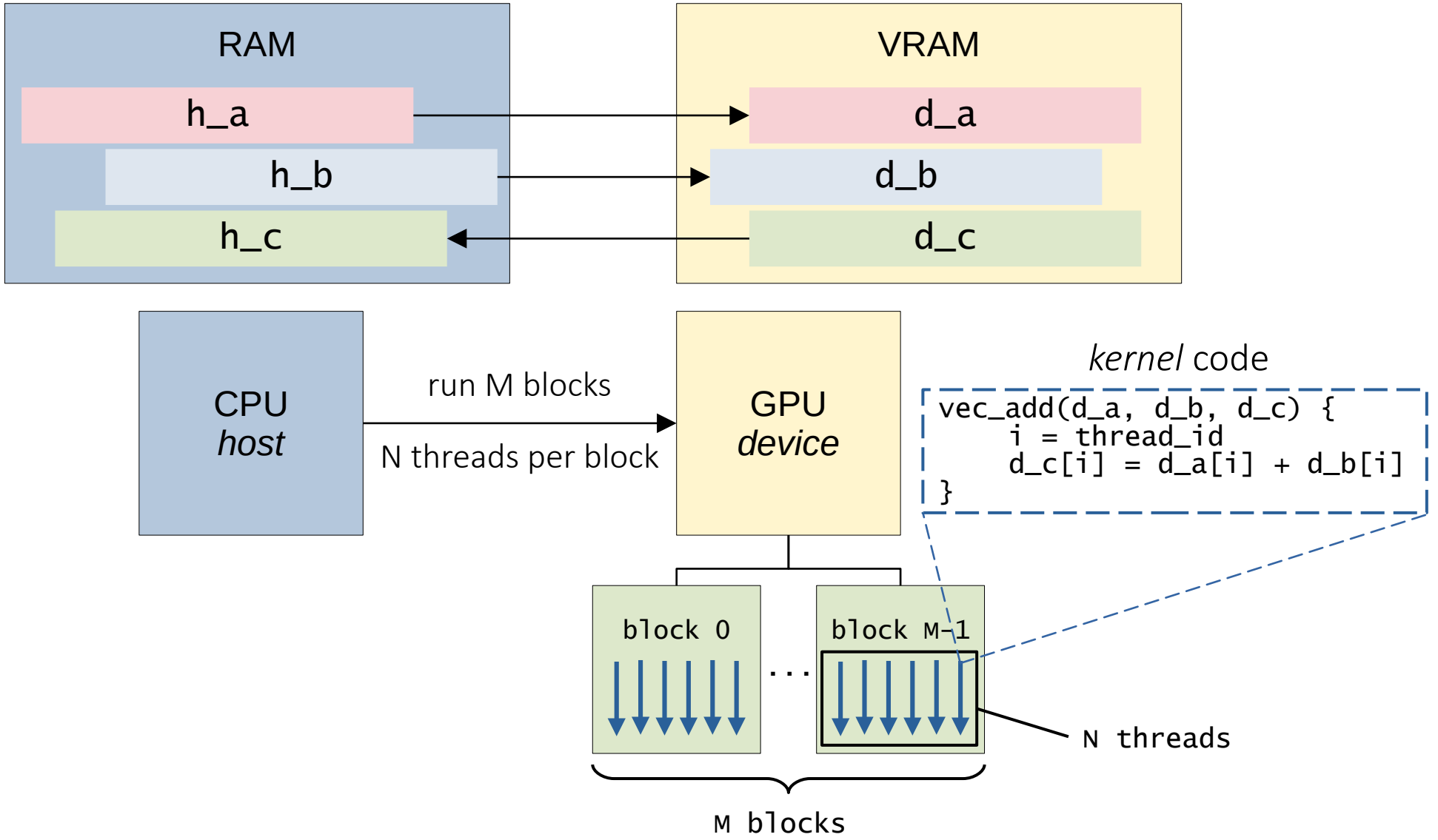
N *sequential* operations

```
c[0] = a[0] + b[0];
c[1] = a[1] + b[1];
...
```

```
$ g++ main.cpp -o main
$ ./main
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50
52 54 56 58 60 62
```

From C++ to CUDA: vector addition

- We want to execute `add_vec` on the GPU in parallel for each index of the arrays



From C++ to CUDA: vector addition

Called from the host, executed on the device

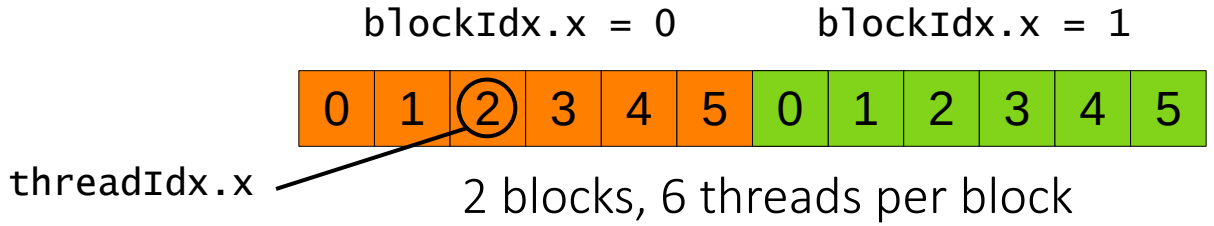
Grid/blocks can be multidimensional

```
// main.cu
__global__ void add_vec_kernel(float *a, float *b, float *c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    c[i] = a[i] + b[i];
}
```

Index of the block of this thread

Number of threads per block

Index of the thread *within* its block



From C++ to CUDA: vector addition

```
int main() {
    int N = 32;
    float *h_a = (float*)malloc(N * sizeof(float));
    float *h_b = (float*)malloc(N * sizeof(float));
    float *h_c = (float*)malloc(N * sizeof(float));
    for(int i = 0; i < N; i++){
        h_a[i] = (float)i; h_b[i] = (float)i;
    }
    float *d_a, *d_b, *d_c; // device pointers
    // Allocate memory on the device
    cudaMalloc(&d_a, N * sizeof(float));
    cudaMalloc(&d_b, N * sizeof(float));
    cudaMalloc(&d_c, N * sizeof(float));
    // Copy host arrays a and b to the device
    cudaMemcpy(d_a, h_a, N * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, N * sizeof(float), cudaMemcpyHostToDevice);
    // Run 1 block of N = 32 threads executing vec_add_kernel
    int blocks = 1;
    int threads_per_block = N;
    add_vec_kernel<<<blocks, threads_per_block>>>(d_a, d_b, d_c, N);
    // Execution returns once all threads terminate

    // Copy the result array back from device to host
    cudaMemcpy(h_c, d_c, N * sizeof(float), cudaMemcpyDeviceToHost);

    for(int i = 0; i < N; i++){
        printf("%.0f ", h_c[i]);
    }

    // free memory both from device and host
    free(h_a); free(h_b); free(h_c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
}
```

From C++ to CUDA: vector addition

```
$ nvcc main.cu -o main
$ ./main
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50
52 54 56 58 60 62
```

- NVCC (**Nv**idia **C**UDA **C**ompiler) can interpret mix of C++ and CU files
  - Supports all features of modern C++ in host code (calls g++/cl compiler)
  - Possible object-oriented code, higher order functions in device code, but **cannot** directly use C++ standard library in a kernel
- Extra libraries provided by CUDA's toolkit:
  - `cuda_fp16.h`, `cuda_bf16.h`: defines binary16 (`half`) and bfloat16 types
  - `cuda.h`, `cuda_runtime.h`: driver and runtime libraries used in host-only code (i.e. compiled with g++, cl...)

CUDA host, global and device functions

- `__global__` : called from the host, runs on the device, **must be `void`**
- `__device__` : called from the device, executed on the device, **GPU-only functions**
- `__host__` : called from the host, executed on the host
  - Regular CPU function, qualifier can be omitted

```
__device__ float my_add(float a, float b) {
    return a + b;
}

__global__ void add_vec_kernel(float *a, float *b, float *c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    c[i] = my_add(a[i], b[i]);
}
```

- Limitations:
  - Maximum of **1024 threads per block**
  - Grid size limits along each dimension
    - Specifications per compute capabilities

Total number of threads

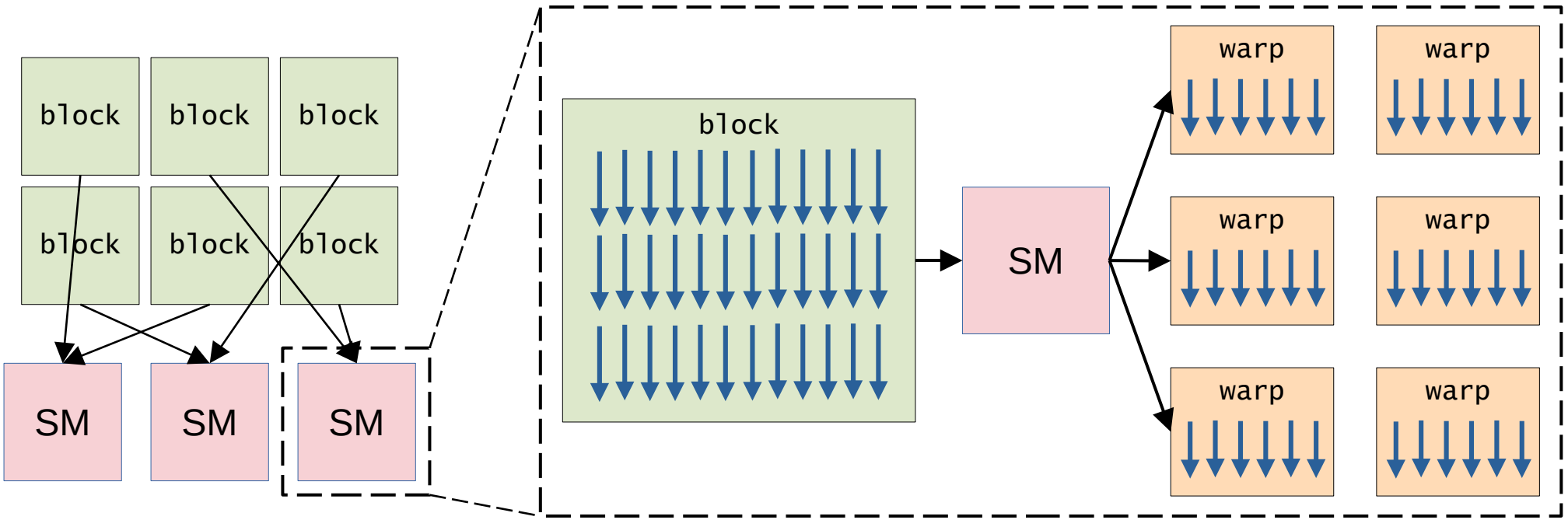
- What if you cannot evenly distribute the same number of threads in each block?
  - Make larger blocks or add an extra block
  - Adds more threads than the problem size
  - Must ignore threads which can cause an out-of-bound indexing: "thread check"

```
// main.cu
#include <stdlib.h>
#include <stdio.h>

__global__ void add_vec_kernel(float *a, float *b, float *c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    // Ignore threads out of the expected range
    if(i < N) {
        c[i] = a[i] + b[i];
    }
}
```

- It is often useful for performance to have block size be a multiple of 32 (see next section on warps)

Warps and lower-level architecture

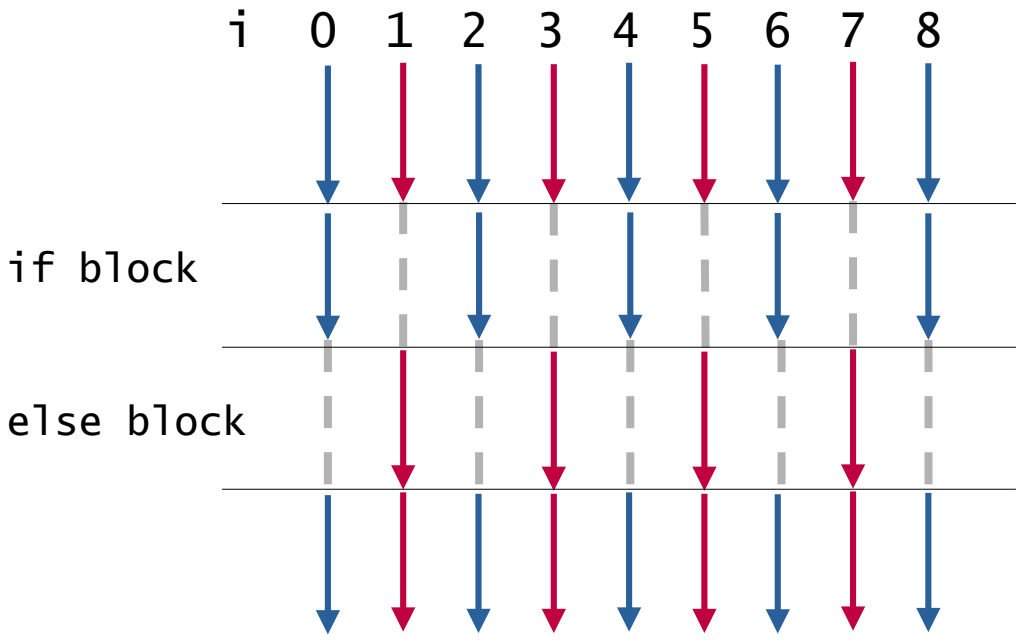
- Each block is assigned to a *streaming multiprocessor* (SM)
- Threads in a block are split in groups of **32** threads (*warp*)
- Scheduling and ordering of each warp is transparent to the programmer
  - The programmer writes a kernel as if each thread in a block is running in parallel



Warp execution and branching

- GPU executes threads in Single Instruction, Multiple Threads (SIMT) manner
- *Lock-step* execution: threads of a warp execute the same instruction at a given instant
- Divergent branching at warp-level can cause slow-downs

```
int i = blockIdx.x * blockDim.x
      + threadIdx.x;

if(i % 2 == 0) {
    // something...
} else {
    // something else...
}
```



PyTorch and CUDA

- CUDA code is loaded with `torch.utils.cpp_extension.load`
  - Calls `nvcc` and creates a python module.
- PyTorch `Tensor`: represents a multidimensional array
  - 1D vector, 2D matrix, 3D tensor…
- Data is stored as a contiguous (ensure with `contiguous()`) row-major array of size `shape[0] * shape[1] * … * shape[n] = numel()`
- Raw access to the array with `data_ptr()`
  - e.g: `float* array_data = tensor.data_ptr<float>();`
  - If `tensor.is_cuda()`, `data_ptr()` is a device pointer
    - No memory transfer with `cudaMemcpy` required
- Call custom kernels directly by passing pointers to raw data

PyTorch example: square tensor values

```
// square.h
void square_tensor(Tensor input, Tensor output);
```

```
// square.cu

__global__ void square_kernel(float *input, float *output, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i < N)
        output[i] = input[i] * input[i];
}

void square_tensor(Tensor input, Tensor output) {
    // Assume input and output have the right shape
    int N = input.numel();
    int threads_per_block = 128;
    // ceiled division, adds an extra block if necessary
    int blocks = N / threads_per_block + (N % threads_per_block > 0);
    float *input_arr = input.data_ptr<float>();
    float *output_arr = output.data_ptr<float>();
    square_kernel<<<blocks, threads_per_block>>>(input_arr, output_arr);
}
```

```
// pybind.cpp
#include <pybind11/pybind11.h>
#include "square.h"

PYBIND11_MODULE(TORCH_EXTENSION_NAME, m) {
    m.def("square_tensor", &square_tensor, "Squares a tensor");
}
```

PyTorch example: square tensor values

```python
# my_module.py

import torch
import torch.utils.cpp_extension as cpp_extension

module = cpp_extension.load(
    name="my_loaded_module",
    sources=["square.cu", "pybind.cpp"]
    verbose=False
)


# define a wrapper around the C++ function
def square(tensor):
    # create the output tensor, must be on the same device
    output = torch.zeros(tensor.shape, device=tensor.device)
    # call the loaded C++ function
    module.square_tensor(tensor, output)
    return output
```

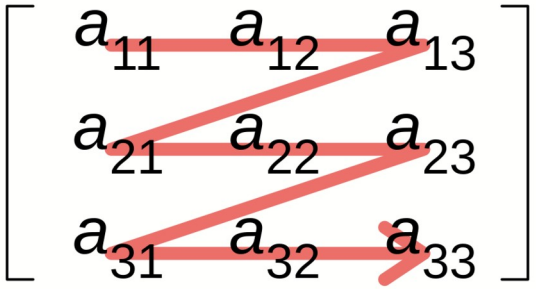
```
>>> import torch
>>> from my_module import square
>>>
>>> x = (torch.rand(3, 4)*10).floor()
>>> x
tensor([[7., 7., 9., 0.],
        [6., 6., 3., 9.],
        [1., 6., 1., 3.]])
>>> square(x)
tensor([[49., 49., 81.,  0.],
        [36., 36.,  9., 81.],
        [ 1., 36.,  1.,  9.]])
```

Matrix storage

- Matrices (and tensors) are stored as a single contiguous row-major or column-major array

PyTorch's convention

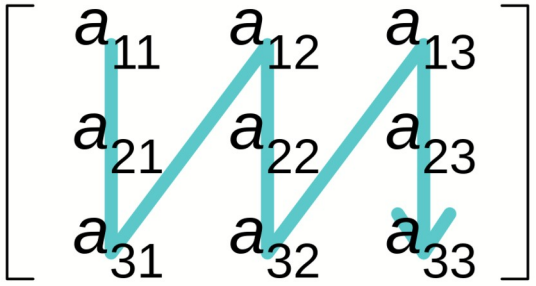
Row-major order



$\{a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{23}, a_{31}, a_{32}, a_{33}\}$

```
index = row * num_columns + column
```

Column-major order



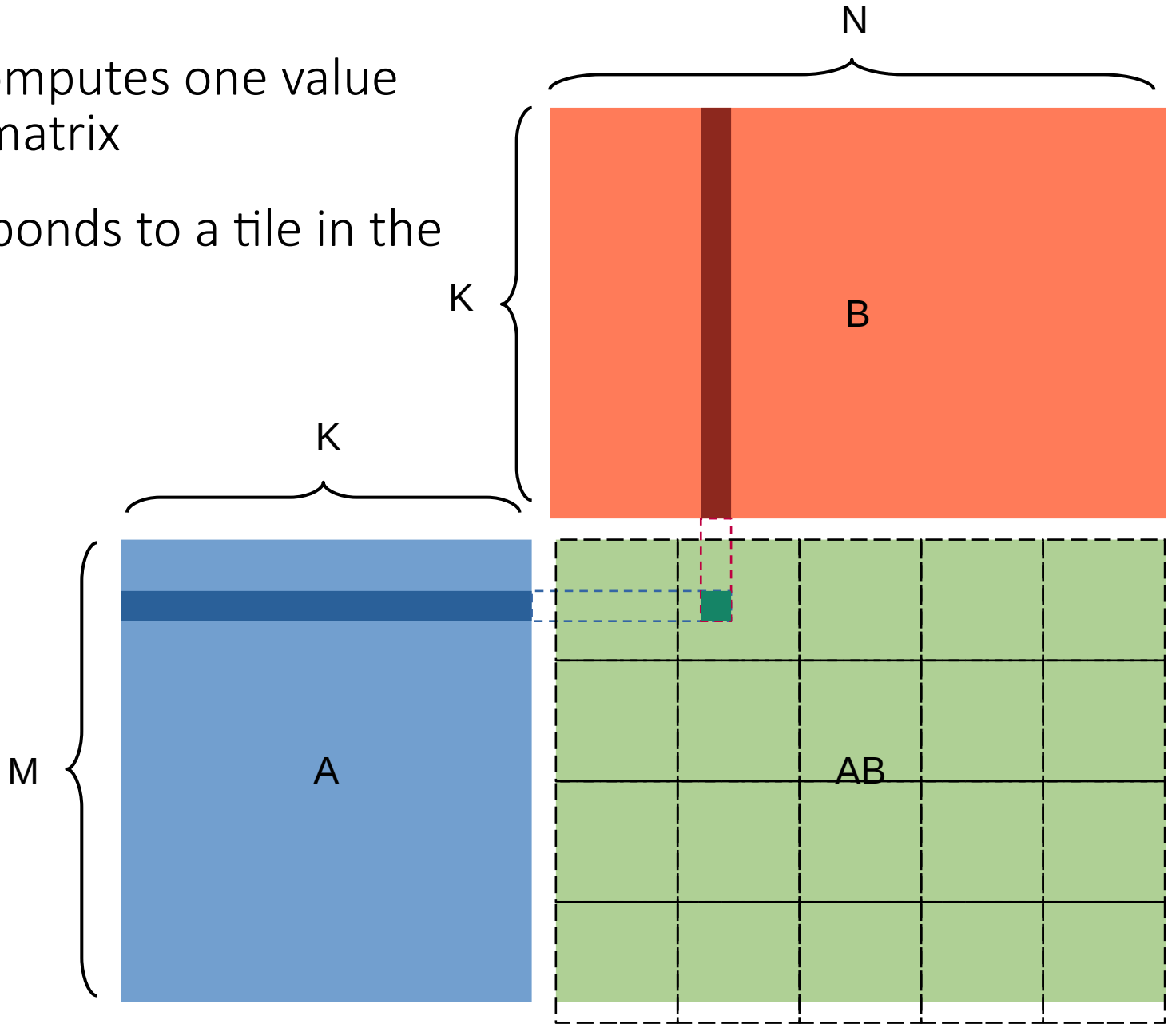
$\{a_{11}, a_{21}, a_{31}, a_{12}, a_{22}, a_{32}, a_{13}, a_{23}, a_{33}\}$

```
index = column * num_rows + row
```

- Ordering format can impact performance (memory coalescing)

Simple CUDA matrix multiply kernel

- Each thread computes one value in the output matrix
- A block corresponds to a tile in the output matrix



Simple CUDA matrix multiply kernel

```
__global__ void matmul_kernel(
    const float* a, // MxK, row-major
    const float* b, // KxN, row-major
    float* out, // MxN, row-major
    int M, int N, int K
) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if(i >= M || j >= N) return;

    float acc = 0.0f;
    for(int k = 0; k < K; k++) {
        acc += a[row * K + k] * b[k * N + col]
    }
    out[row * N + col] = acc;
}

void matmul(Tensor a, Tensor b, Tensor out) {
    // Assume a, b and out are matrices with matching shapes
    int M = a.size(0);
    int K = a.size(1);
    int N = b.size(1);
    // 2D tiled threads
    dim2 block_size(16, 16); // 16x16 tiles
    dim2 blocks(divceil(N, 16), divceil(M, 16));
    float *a_arr = a.data_ptr<float>();
    float *b_arr = b.data_ptr<float>();
    float *out_arr = out.data_ptr<float>();
    matmul_kernel<<<blocks, block_size>>>(
        a_arr, b_arr, out_arr, M, N, K
    );
}
```

Further resources

- https://en.wikipedia.org/wiki/CUDA
- https://docs.nvidia.com/cuda/cuda-c-programming-guide/
- https://developer.nvidia.com/blog/even-easier-introduction-cuda/
- https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html