

# P&S Heterogeneous Systems

## GPU Software Hierarchy:

Grids, Blocks, Threads

Dr. Juan Gómez Luna

Prof. Onur Mutlu

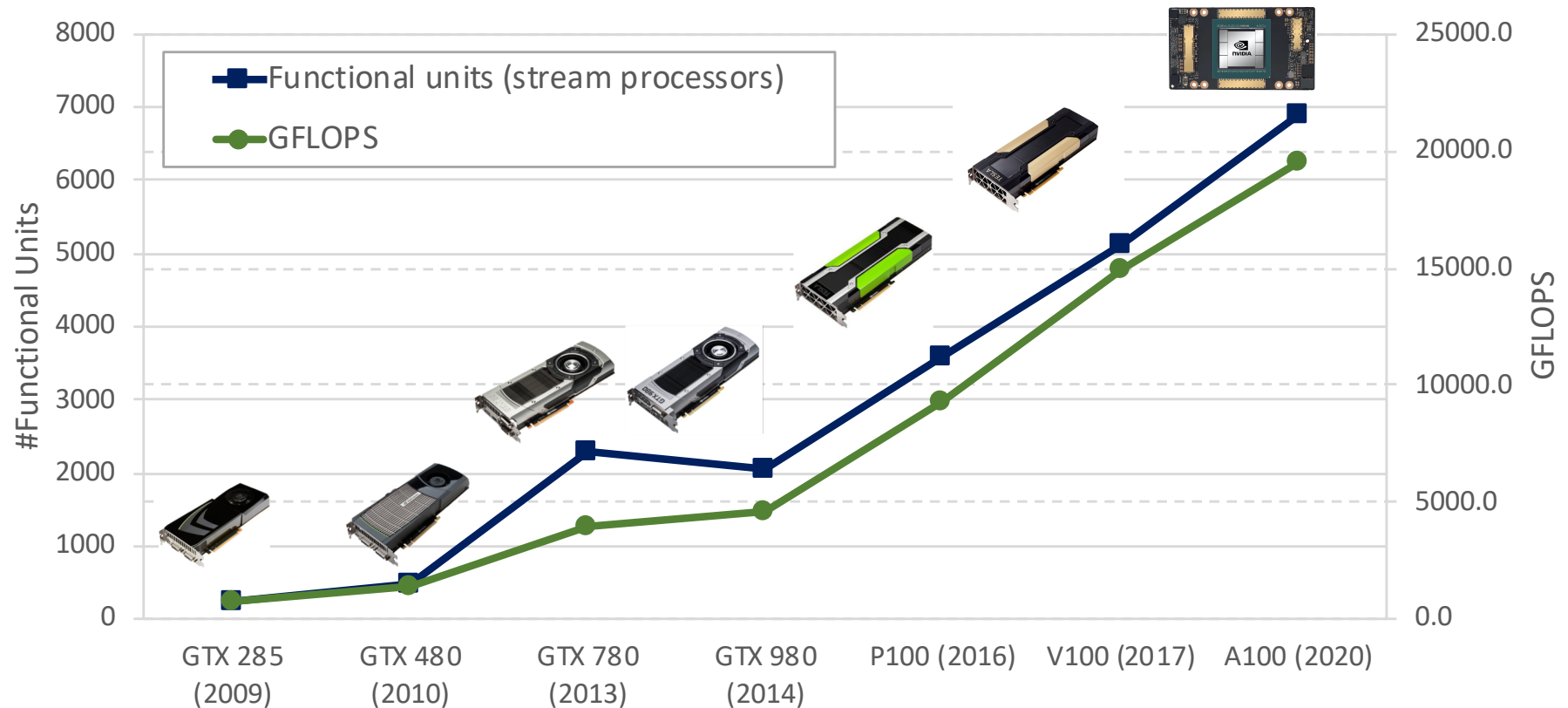
ETH Zürich

Fall 2021

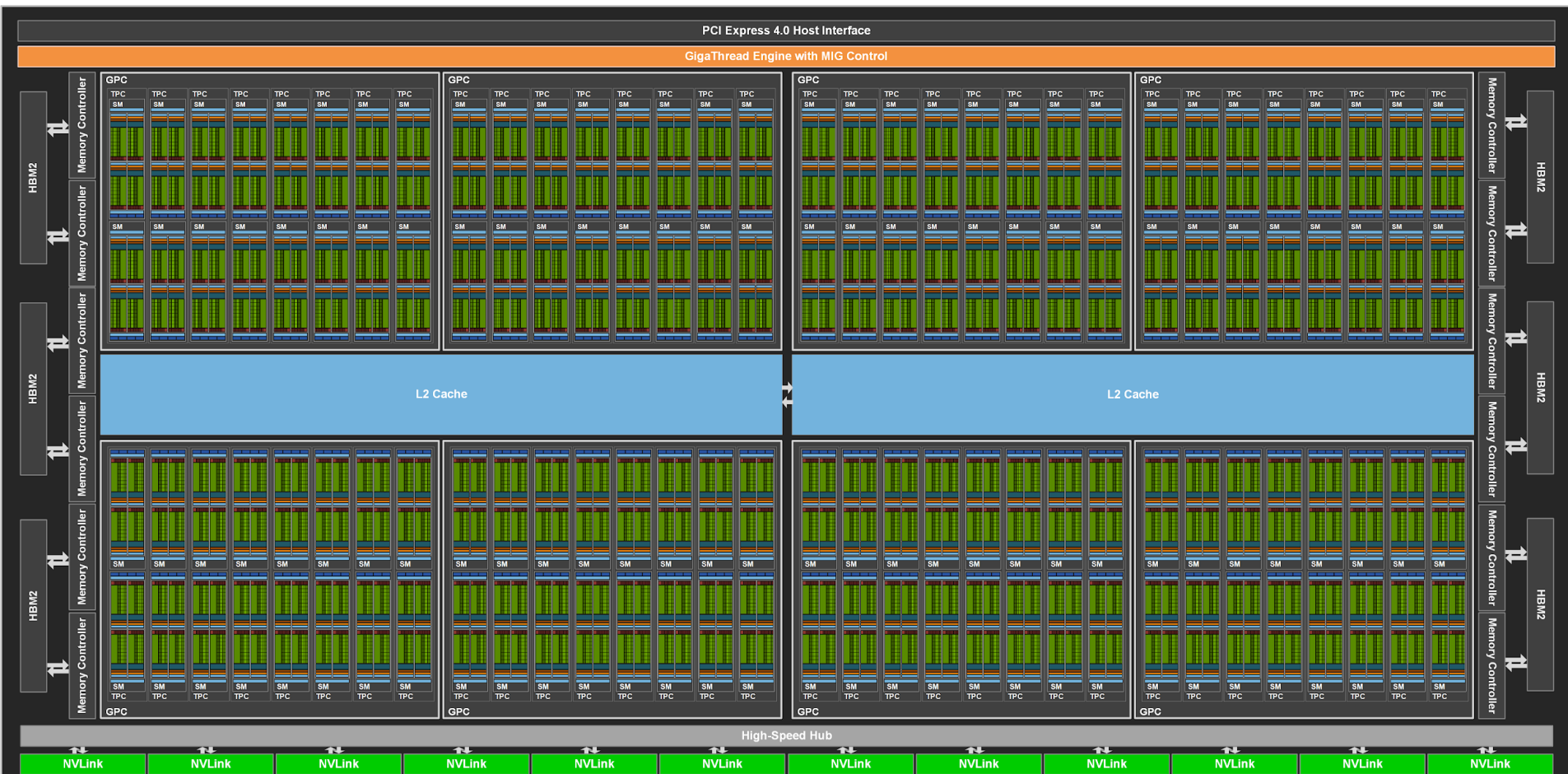
21 October 2021

# GPUs are SIMD Engines Underneath

# Evolution of NVIDIA GPUs



# NVIDIA A100 Block Diagram



<https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/>

## 108 cores on the A100

(Up to 128 cores in the full-blown chip)

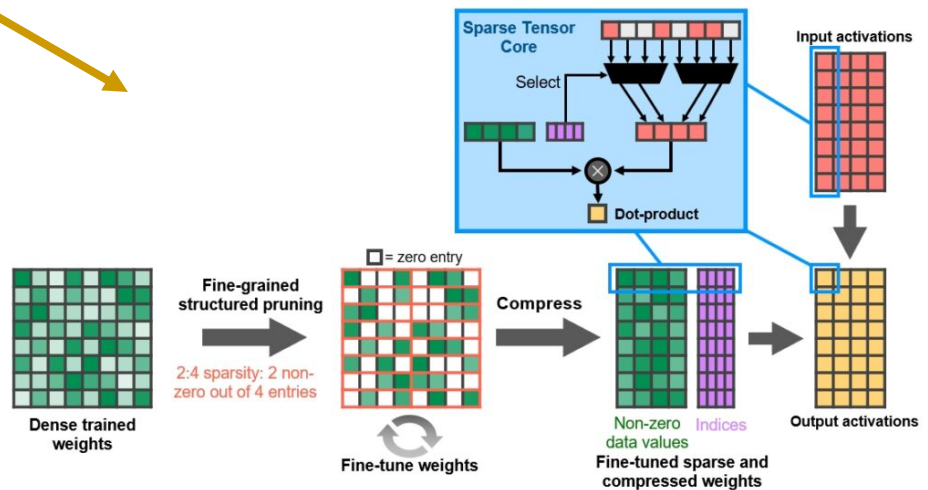
## 40MB L2 cache



# NVIDIA A100 Core



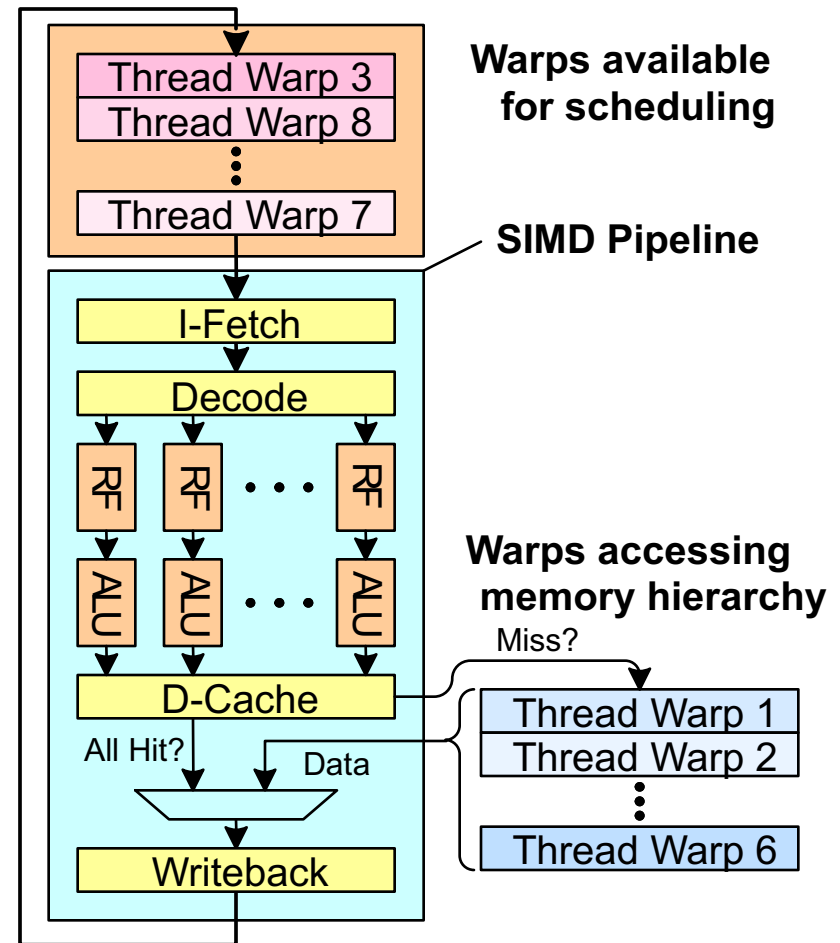
19.5 TFLOPS Single Precision  
9.7 TFLOPS Double Precision  
312 TFLOPS for Deep Learning (Tensor cores)



<https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/>

# Recall: Latency Hiding via Warp-Level FGMT

- Warp: A set of threads that execute the same instruction (on different data elements)
- Fine-grained multithreading
  - One instruction per thread in pipeline at a time (No interlocking)
  - Interleave warp execution to hide latencies
- Register values of all threads stay in register file
- FGMT enables long latency tolerance
  - Millions of pixels

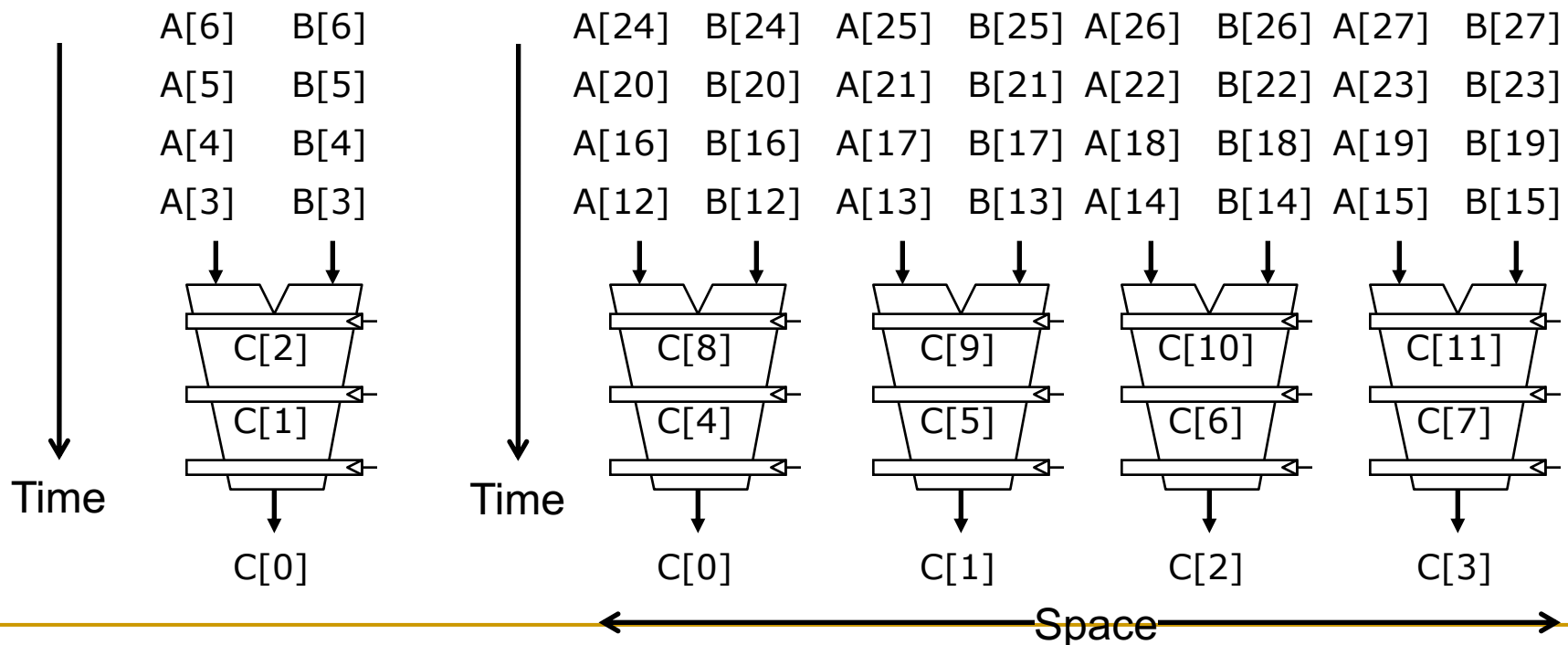


# Recall: Warp Execution

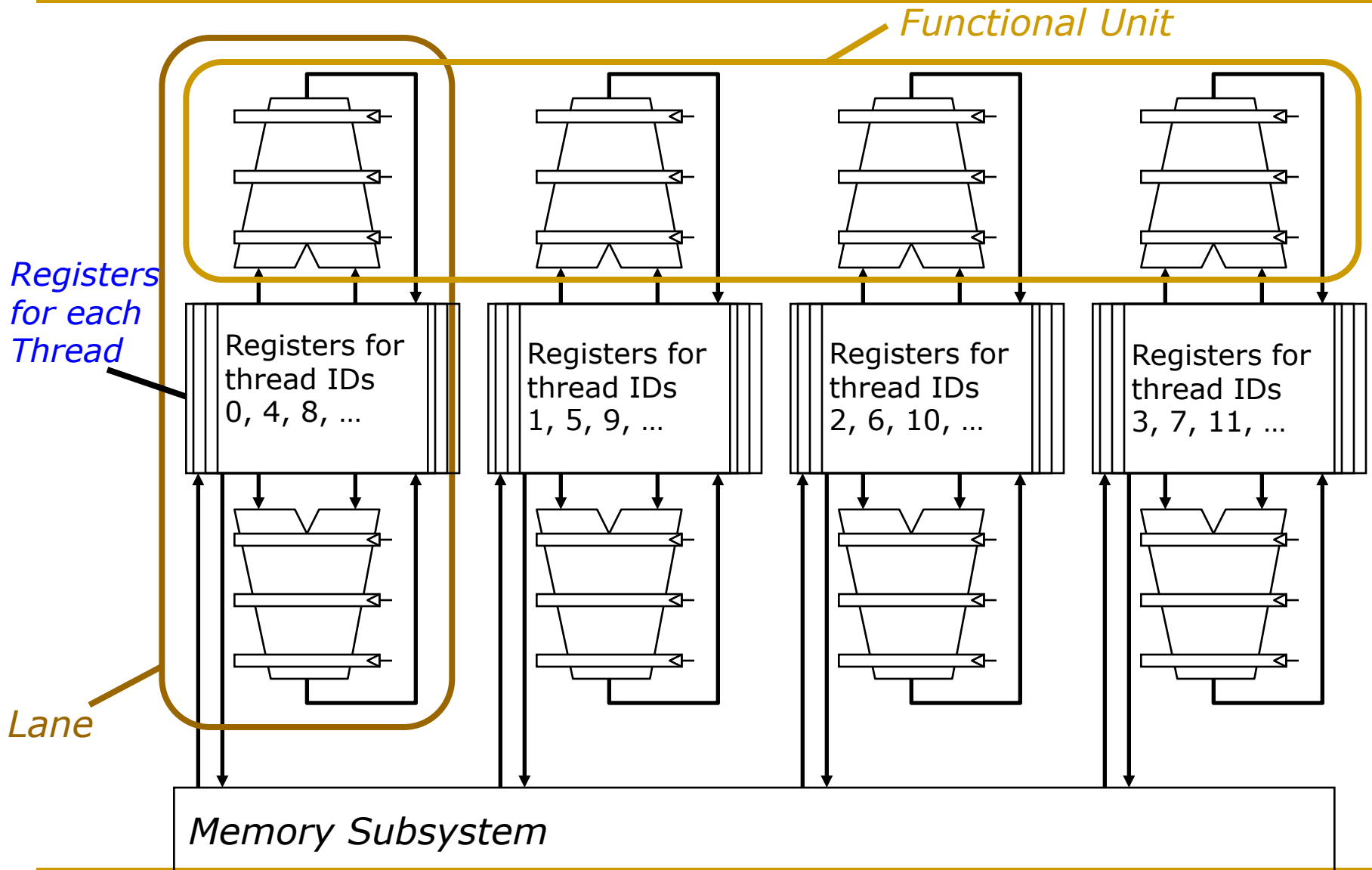
32-thread warp executing  $\text{ADD } A[\text{tid}], B[\text{tid}] \rightarrow C[\text{tid}]$

*Execution using  
one pipelined  
functional unit*

*Execution using  
four pipelined  
functional units*



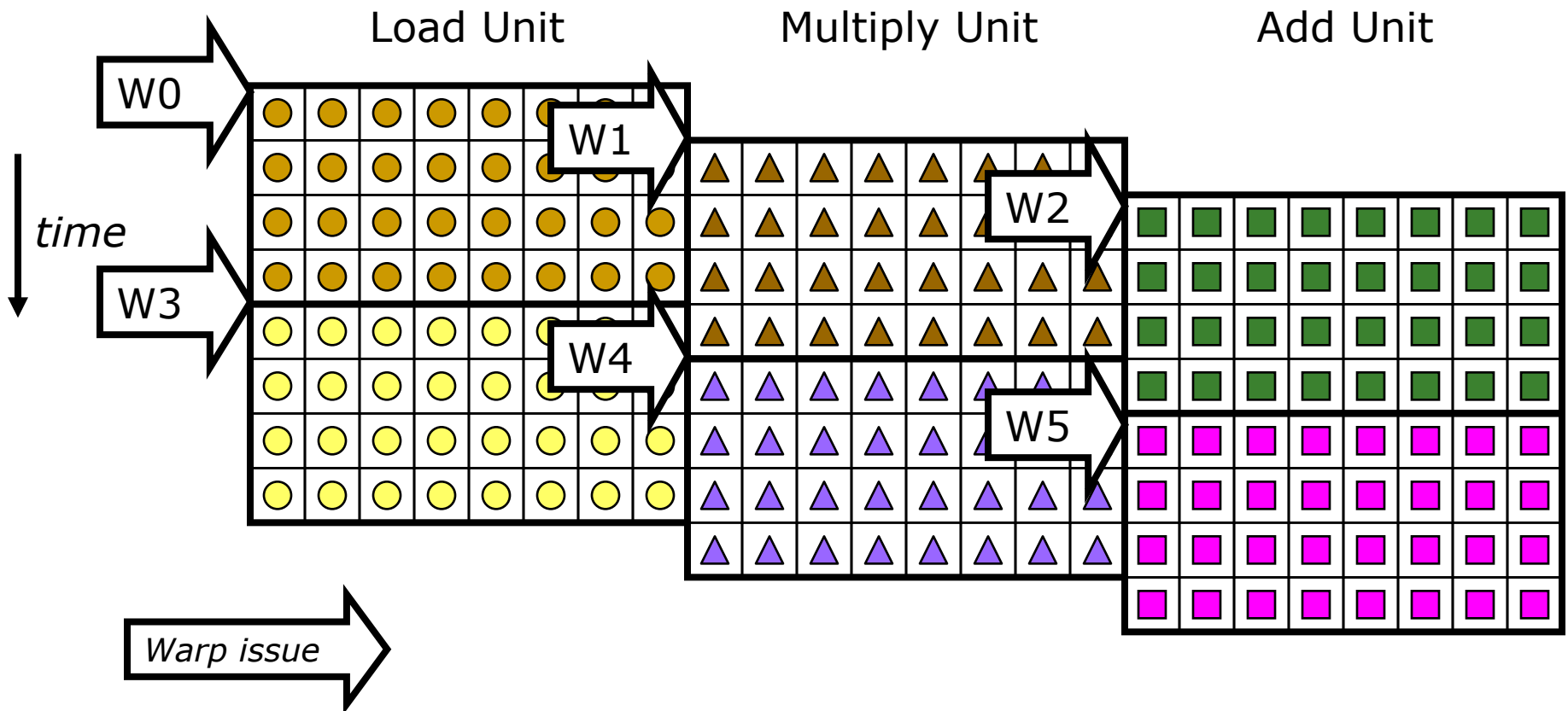
# Recall: SIMD Execution Unit Structure



# Recall: Warp Instruction Level Parallelism

Can overlap execution of multiple instructions

- Example machine has 32 threads per warp and 8 lanes
- Completes 24 operations/cycle while issuing 1 warp/cycle



# GPU Programming

# Recall: Vector Processor Disadvantages

---

- Works (only) if parallelism is regular (data/SIMD parallelism)
  - ++ Vector operations
  - Very inefficient if parallelism is irregular
    - How about searching for a key in a linked list?

To program a vector machine, the compiler or hand coder must make the data structures in the code fit nearly exactly the regular structure built into the hardware. That's hard to do in first place, and just as hard to change. One tweak, and the low-level code has to be rewritten by a very smart and dedicated programmer who knows the hardware and often the subtleties of the application area. Often the rewriting is

# General Purpose Processing on GPU

---

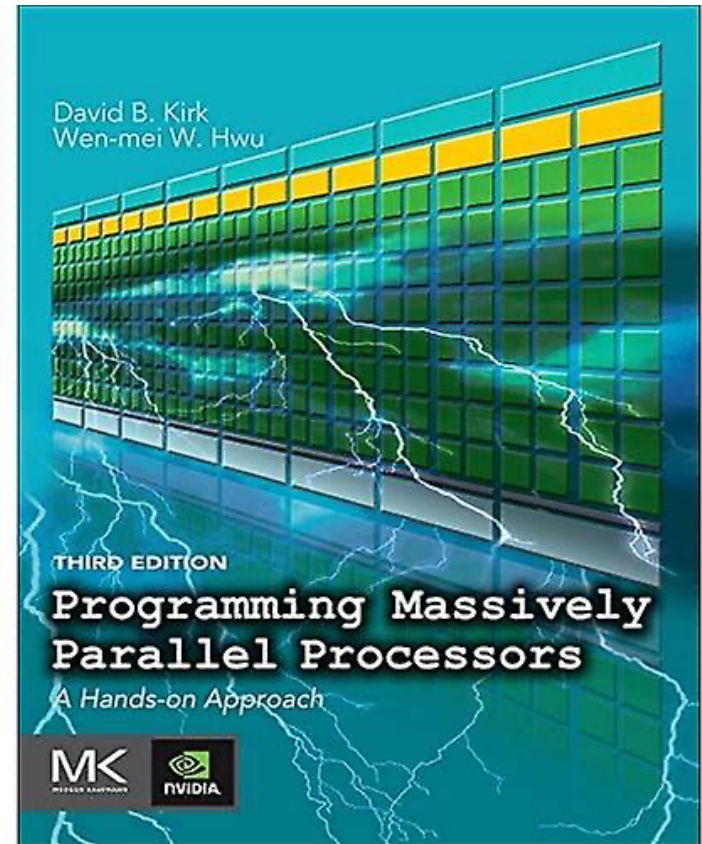
- Easier programming of SIMD processors with SPMD
  - GPUs have democratized High Performance Computing (HPC)
  - Great FLOPS/\$, massively parallel chip on a commodity PC
- Many workloads exhibit inherent parallelism
  - Matrices
  - Image processing
  - Deep neural networks
- However, this is not for free
  - New programming model
  - Algorithms need to be re-implemented and rethought
- Still some bottlenecks
  - CPU-GPU data transfers (PCIe, NVLINK)
  - DRAM memory bandwidth (GDDR5, GDDR6, HBM2)
    - Data layout



# Recommended Readings (I)

---

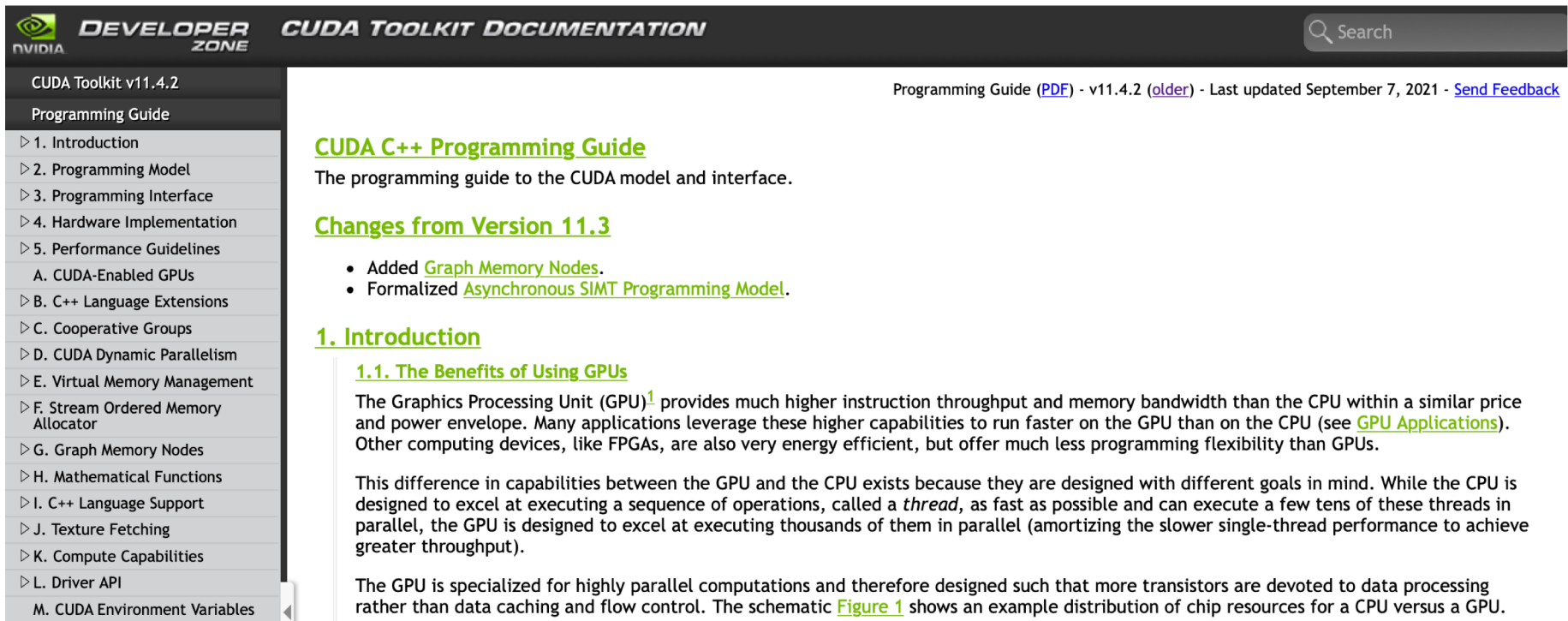
- Hwu and Kirk, “**Programming Massively Parallel Processors,**” Third Edition, 2017



# Recommended Readings (II)

## ■ CUDA Programming Guide

- <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>



The screenshot shows the NVIDIA Developer Zone interface for the CUDA Toolkit v11.4.2 documentation. The left sidebar contains a navigation menu with the following items: Introduction, Programming Model, Programming Interface, Hardware Implementation, Performance Guidelines (with sub-items A. CUDA-Enabled GPUs, B. C++ Language Extensions, C. Cooperative Groups, D. CUDA Dynamic Parallelism, E. Virtual Memory Management, F. Stream Ordered Memory Allocator, G. Graph Memory Nodes, H. Mathematical Functions, I. C++ Language Support, J. Texture Fetching, K. Compute Capabilities, L. Driver API, and M. CUDA Environment Variables), and a search bar. The main content area is titled "CUDA C++ Programming Guide" and includes a sub-header "Changes from Version 11.3" with a bulleted list of updates: "Added Graph Memory Nodes" and "Formalized Asynchronous SIMT Programming Model". Below this is the "1. Introduction" section, which contains a sub-section "1.1. The Benefits of Using GPUs". The text in 1.1 explains that the GPU provides higher instruction throughput and memory bandwidth than the CPU, and that it is designed to execute many threads in parallel. It also mentions that the GPU is specialized for highly parallel computations.

**DEVELOPER ZONE** **CUDA TOOLKIT DOCUMENTATION**

CUDA Toolkit v11.4.2

Programming Guide

- ▷ 1. Introduction
- ▷ 2. Programming Model
- ▷ 3. Programming Interface
- ▷ 4. Hardware Implementation
- ▷ 5. Performance Guidelines
  - A. CUDA-Enabled GPUs
  - ▷ B. C++ Language Extensions
  - ▷ C. Cooperative Groups
  - ▷ D. CUDA Dynamic Parallelism
  - ▷ E. Virtual Memory Management
  - ▷ F. Stream Ordered Memory Allocator
  - ▷ G. Graph Memory Nodes
  - ▷ H. Mathematical Functions
  - ▷ I. C++ Language Support
  - ▷ J. Texture Fetching
  - ▷ K. Compute Capabilities
  - ▷ L. Driver API
  - ▷ M. CUDA Environment Variables

Programming Guide ([PDF](#)) - v11.4.2 ([older](#)) - Last updated September 7, 2021 - [Send Feedback](#)

## CUDA C++ Programming Guide

The programming guide to the CUDA model and interface.

### Changes from Version 11.3

- Added [Graph Memory Nodes](#).
- Formalized [Asynchronous SIMT Programming Model](#).

## 1. Introduction

### 1.1. The Benefits of Using GPUs

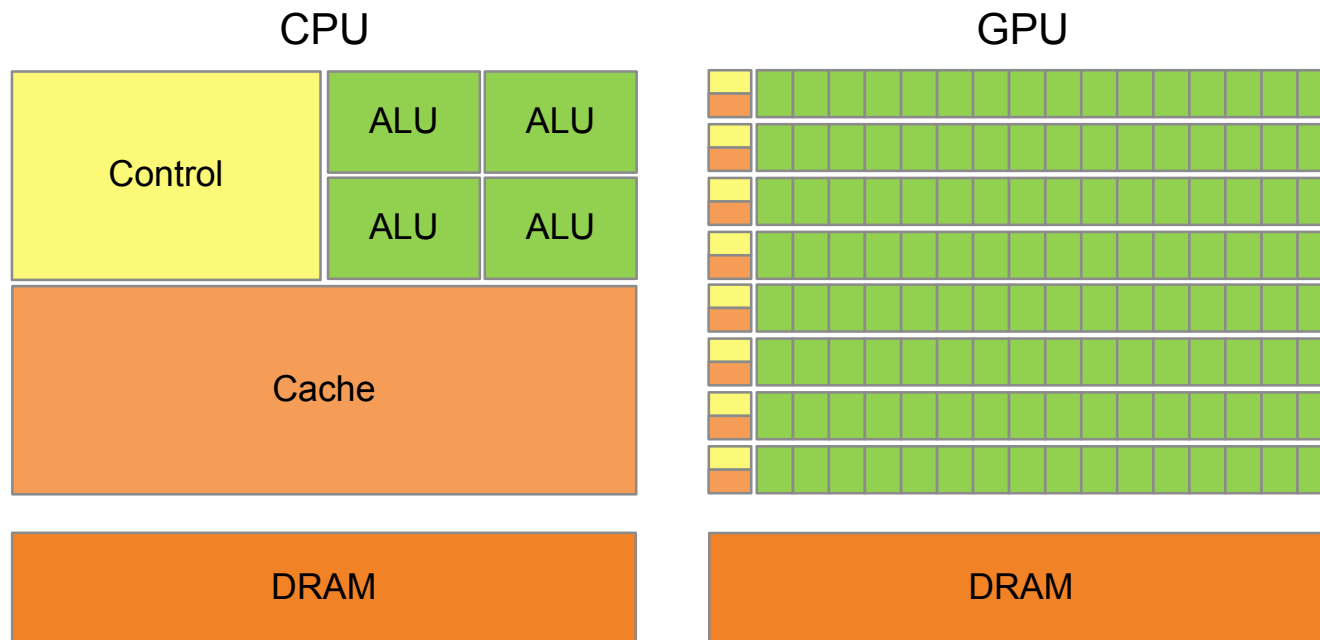
The Graphics Processing Unit (GPU)<sup>1</sup> provides much higher instruction throughput and memory bandwidth than the CPU within a similar price and power envelope. Many applications leverage these higher capabilities to run faster on the GPU than on the CPU (see [GPU Applications](#)). Other computing devices, like FPGAs, are also very energy efficient, but offer much less programming flexibility than GPUs.

This difference in capabilities between the GPU and the CPU exists because they are designed with different goals in mind. While the CPU is designed to excel at executing a sequence of operations, called a *thread*, as fast as possible and can execute a few tens of these threads in parallel, the GPU is designed to excel at executing thousands of them in parallel (amortizing the slower single-thread performance to achieve greater throughput).

The GPU is specialized for highly parallel computations and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control. The schematic [Figure 1](#) shows an example distribution of chip resources for a CPU versus a GPU.

# CPU vs. GPU

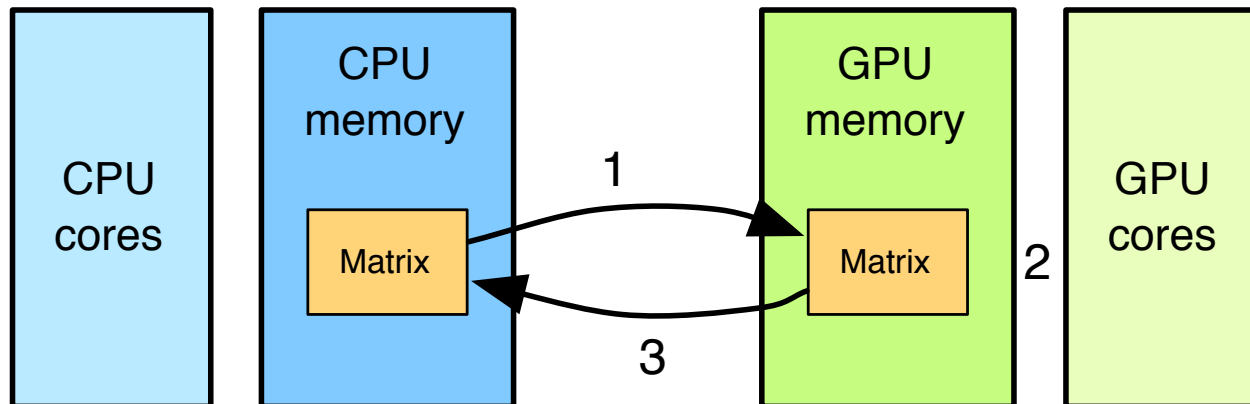
- Different design philosophies
  - ❑ CPU: A few out-of-order cores
  - ❑ GPU: Many in-order FGMT cores



# GPU Computing

---

- Computation is **offloaded to the GPU**
- Three steps
  - ❑ CPU-GPU data transfer (1)
  - ❑ GPU kernel execution (2)
  - ❑ GPU-CPU data transfer (3)



# Traditional Program Structure

- CPU threads and GPU kernels
  - Sequential or modestly parallel sections on CPU
  - Massively parallel sections on GPU

Serial Code (host)

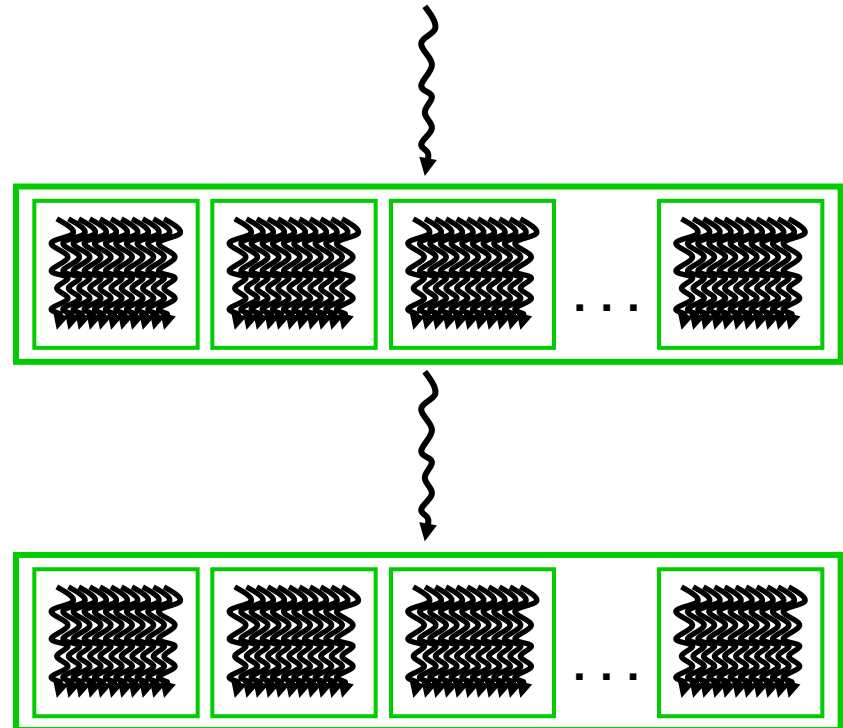
Parallel Kernel (device)

```
KernelA<<< nBlk, nThr >>>(args);
```

Serial Code (host)

Parallel Kernel (device)

```
KernelB<<< nBlk, nThr >>>(args);
```



# Recall: SPMD

---

- Single procedure/program, multiple data
  - This is a programming model rather than computer organization
- Each processing element executes the same procedure, except on different data elements
  - Procedures can synchronize at certain points in program, e.g. barriers
- Essentially, multiple instruction streams execute the same program
  - Each program/procedure 1) works on different data, 2) can execute a different control-flow path, at run-time
  - Many scientific applications are programmed this way and run on MIMD hardware (multiprocessors)
  - Modern GPUs programmed in a similar way on a SIMD hardware

# CUDA/OpenCL Programming Model

---

- SIMT or SPMD
- Bulk synchronous programming
  - Global (coarse-grain) synchronization between kernels
- The host (typically CPU) allocates memory, copies data, and launches kernels
- The device (typically GPU) executes kernels
  - Grid (NDRange)
  - Block (work-group)
    - Within a block, shared memory, and synchronization
  - Thread (work-item)


# Traditional Program Structure in CUDA

## ■ Function prototypes

```
float serialFunction(...);  
__global__ void kernel(...);
```

## ■ main( )

- ❑ 1) **Allocate memory** space on the device – `cudaMalloc(&d_in, bytes);`
- ❑ 2) Transfer data from **host to device** – `cudaMemcpy(d_in, h_in, ...);`
- ❑ 3) Execution configuration setup: `#blocks` and `#threads`
- ❑ 4) **Kernel call** – `kernel<<<execution configuration>>>(args...);`
- ❑ 5) Transfer results from **device to host** – `cudaMemcpy(h_out, d_out, ...);`



repeat  
as needed

## ■ Kernel – `__global__ void kernel(type args,...)`

- ❑ Automatic variables transparently assigned to **registers**
- ❑ **Shared memory**: `__shared__`
- ❑ Intra-block **synchronization**: `__syncthreads( )`;



# CUDA Programming Language

---

- Memory allocation

```
cudaMalloc((void**)&d_in, #bytes);
```

- Memory copy

```
cudaMemcpy(d_in, h_in, #bytes, cudaMemcpyHostToDevice);
```

- Kernel launch

```
kernel<<< #blocks, #threads >>>(args);
```

- Memory deallocation

```
cudaFree(d_in);
```

- Explicit synchronization

```
cudaDeviceSynchronize();
```

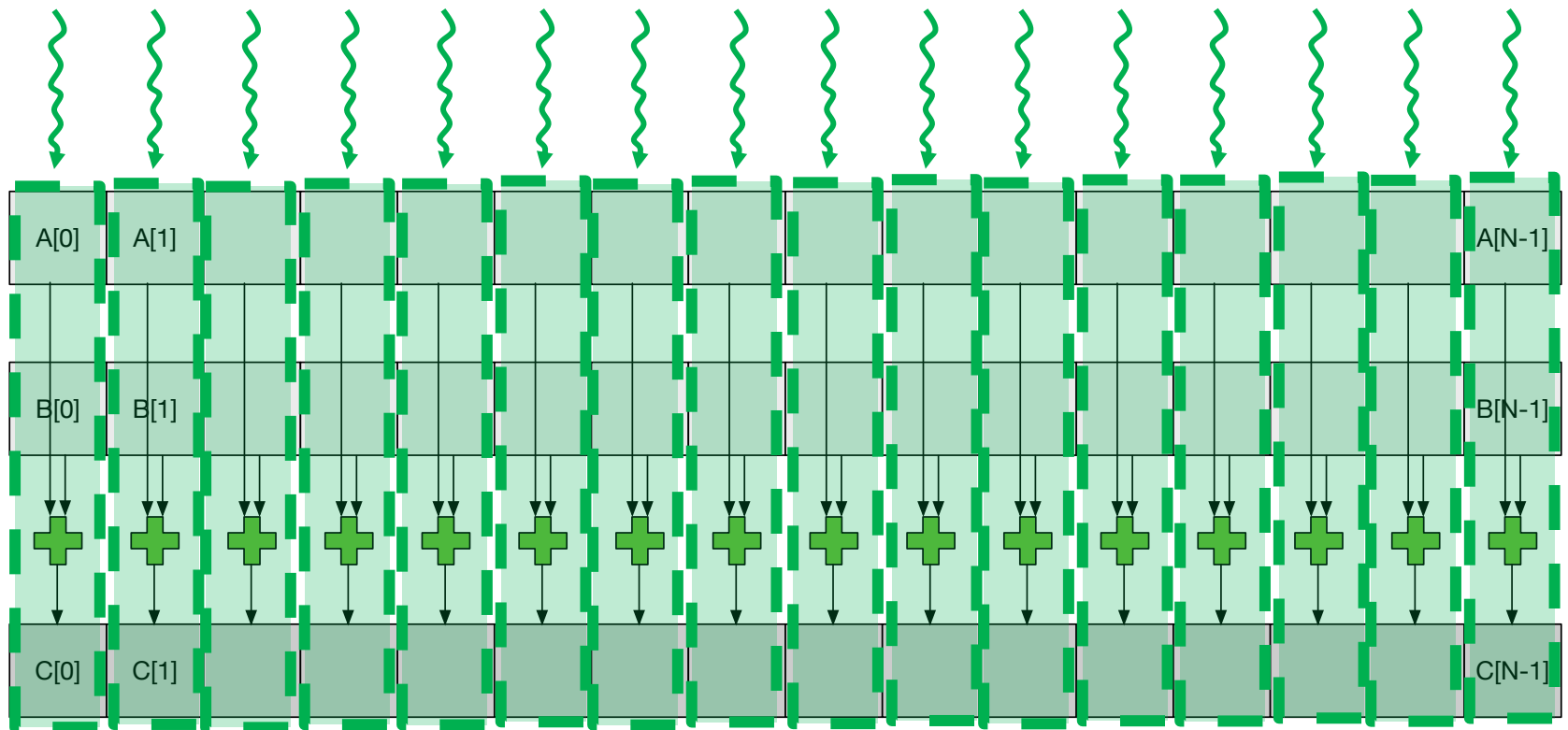
# Host Code Example: Vector Addition

---

```
void vecadd(float* A, float* B, float* C, int N) {  
  
    // Allocate GPU memory  
    float *A_d, *B_d, *C_d;  
    cudaMalloc((void**) &A_d, N*sizeof(float));  
    cudaMalloc((void**) &B_d, N*sizeof(float));  
    cudaMalloc((void**) &C_d, N*sizeof(float));  
  
    // Copy data to GPU memory  
    cudaMemcpy(A_d, A, N*sizeof(float), cudaMemcpyHostToDevice);  
    cudaMemcpy(B_d, B, N*sizeof(float), cudaMemcpyHostToDevice);  
  
    // Perform computation on GPU  
    ...  
  
    // Copy data from GPU memory  
    cudaMemcpy(C, C_d, N*sizeof(float), cudaMemcpyDeviceToHost);  
  
    // Deallocate GPU memory  
    cudaFree(A_d);  
    cudaFree(B_d);  
    cudaFree(C_d);  
}
```

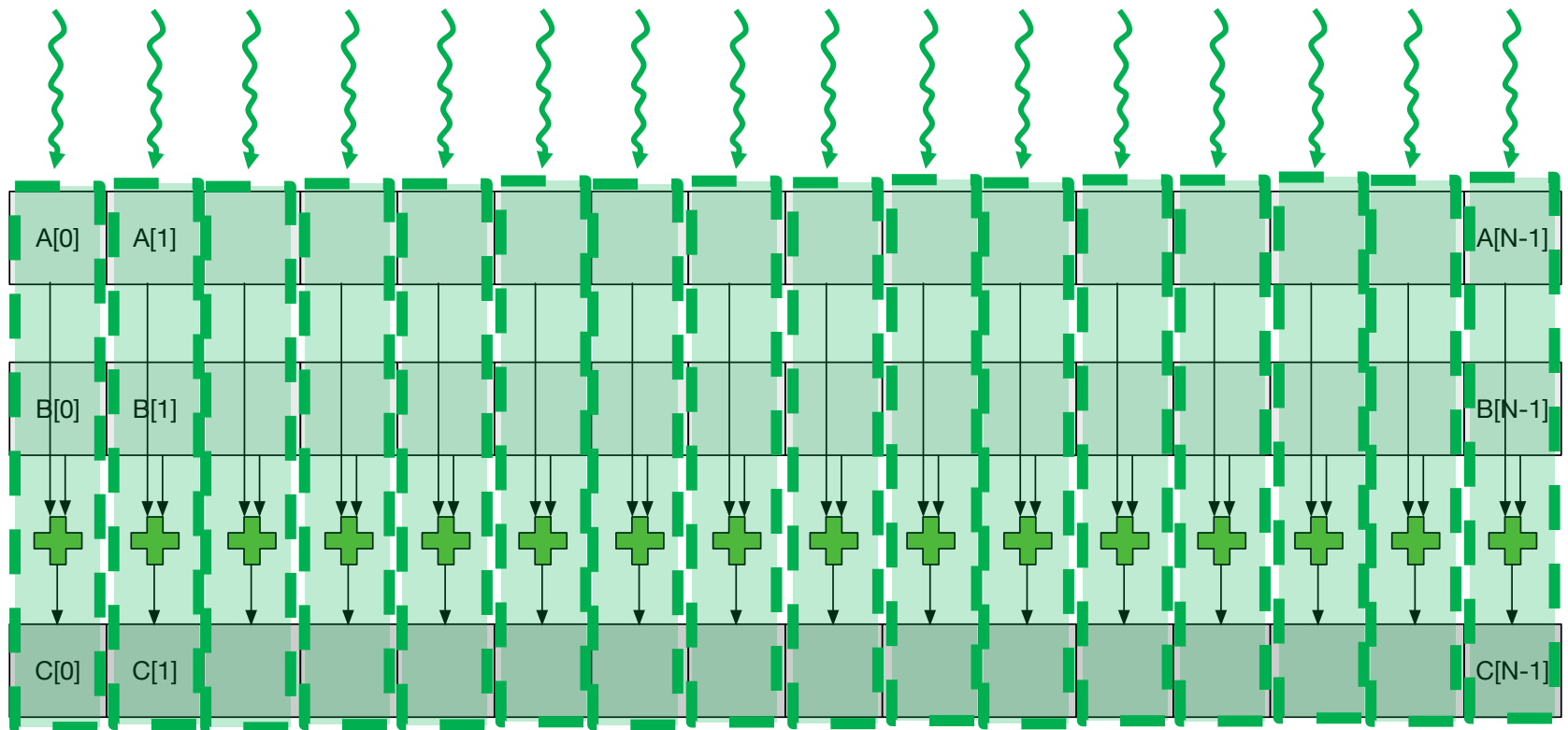
# Vector Addition (I)

- Our first GPU programming example
- We assign **one GPU thread to each element-wise addition**



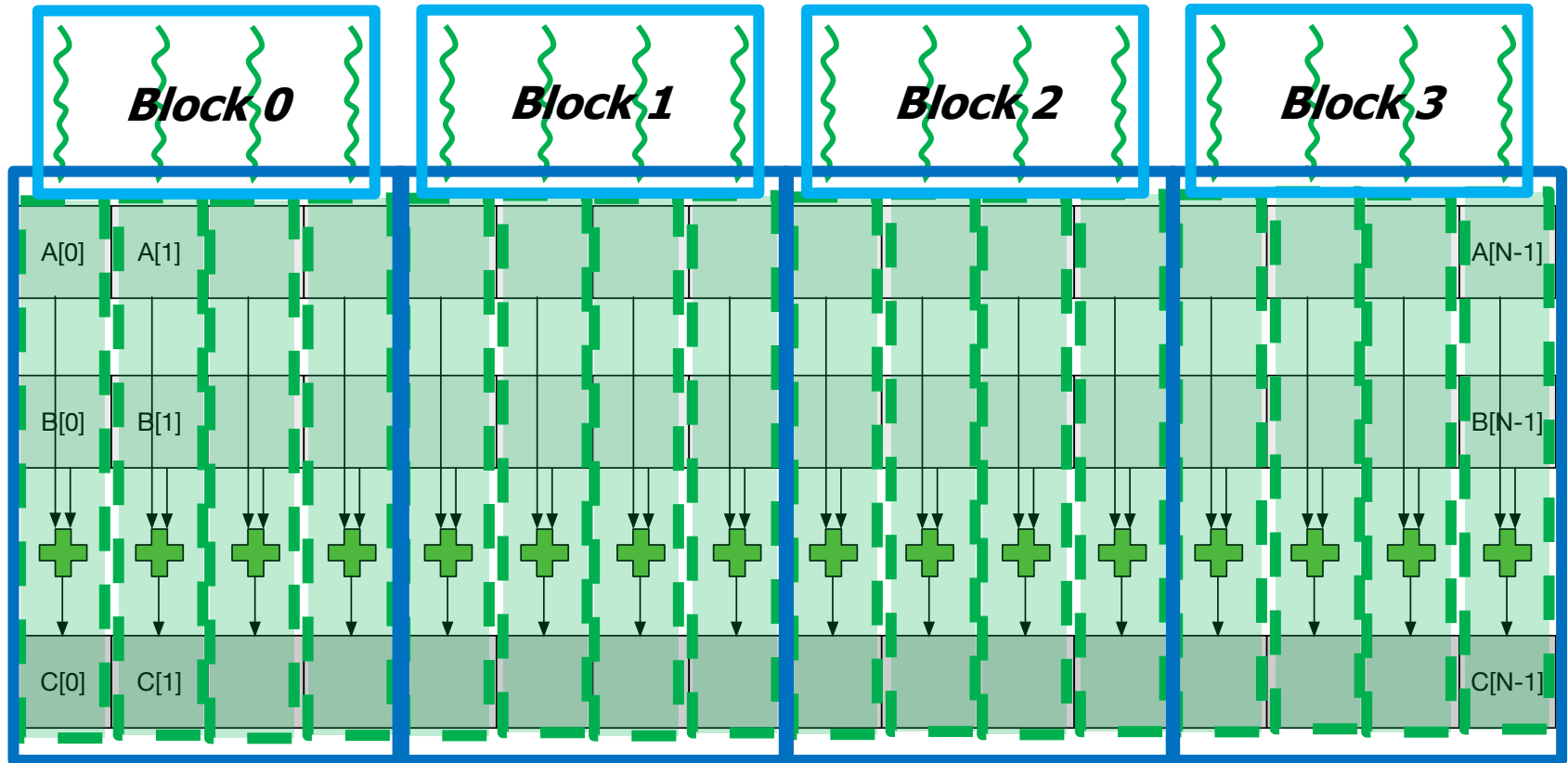
# Vector Addition (II)

- The whole set of threads is called a **grid**
- We need a way to assign threads to GPU cores



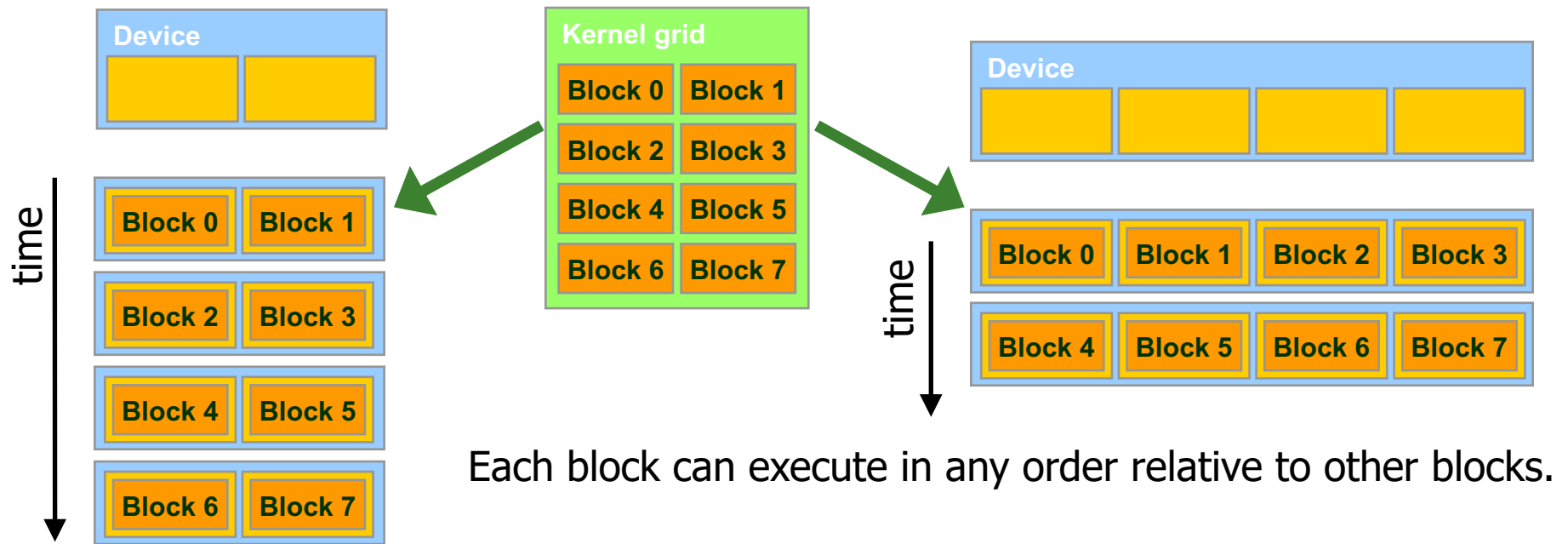
# Vector Addition (III)

- We group threads into **blocks**



# Transparent Scalability

- Hardware is **free to schedule** thread blocks



# Launching a Grid

---

- Threads in the same grid execute the same function known as a **kernel**
- A grid can be launched by calling a kernel and configuring it with appropriate grid and block sizes

```
const unsigned int numThreadsPerBlock = 512;  
const unsigned int numBlocks = N/numThreadsPerBlock;  
  
vecadd_kernel<<<numBlocks, numThreadsPerBlock>>>(A_d, B_d, C_d, N);
```

# Host Code Example: Vector Addition

---

```
void vecadd(float* A, float* B, float* C, int N) {  
  
    // Allocate GPU memory  
    float *A_d, *B_d, *C_d;  
    cudaMalloc((void**) &A_d, N*sizeof(float));  
    cudaMalloc((void**) &B_d, N*sizeof(float));  
    cudaMalloc((void**) &C_d, N*sizeof(float));  
  
    // Copy data to GPU memory  
    cudaMemcpy(A_d, A, N*sizeof(float), cudaMemcpyHostToDevice);  
    cudaMemcpy(B_d, B, N*sizeof(float), cudaMemcpyHostToDevice);  
  
    // Perform computation on GPU  
    const unsigned int numThreadsPerBlock = 512;  
    const unsigned int numBlocks = N/numThreadsPerBlock;  
  
    vecadd_kernel<<<numBlocks, numThreadsPerBlock>>>(A_d, B_d, C_d, N);  
    // Copy data from GPU memory  
    cudaMemcpy(C, C_d, N*sizeof(float), cudaMemcpyDeviceToHost);  
  
    // Deallocate GPU memory  
    cudaFree(A_d);  
    cudaFree(B_d);  
    cudaFree(C_d);  
}
```



# Sample GPU SIMT Code (Simplified)

---

CPU code

```
for (ii = 0; ii < 100000; ++ii) {  
    C[ii] = A[ii] + B[ii];  
}
```



CUDA code

```
// there are 100000 threads  
__global__ void KernelFunction(...) {  
    int tid = blockDim.x * blockIdx.x + threadIdx.x;  
    int varA = aa[tid];  
    int varB = bb[tid];  
    C[tid] = varA + varB;  
}
```

# Vector Addition Kernel

---

- It is preceded by the keyword `__global__` to indicate that it is a GPU kernel
- It uses special keywords to distinguish different threads from each other
  - Block index (`blockIdx.x`), block size (`blockDim.x`), thread index (`threadIdx.x`)

```
__global__ void vecadd_kernel(float* A, float* B, float* C, int N) {  
    int i = blockDim.x * blockIdx.x + threadIdx.x;  
    C[i] = A[i] + B[i];  
}
```

# Boundary Conditions

---

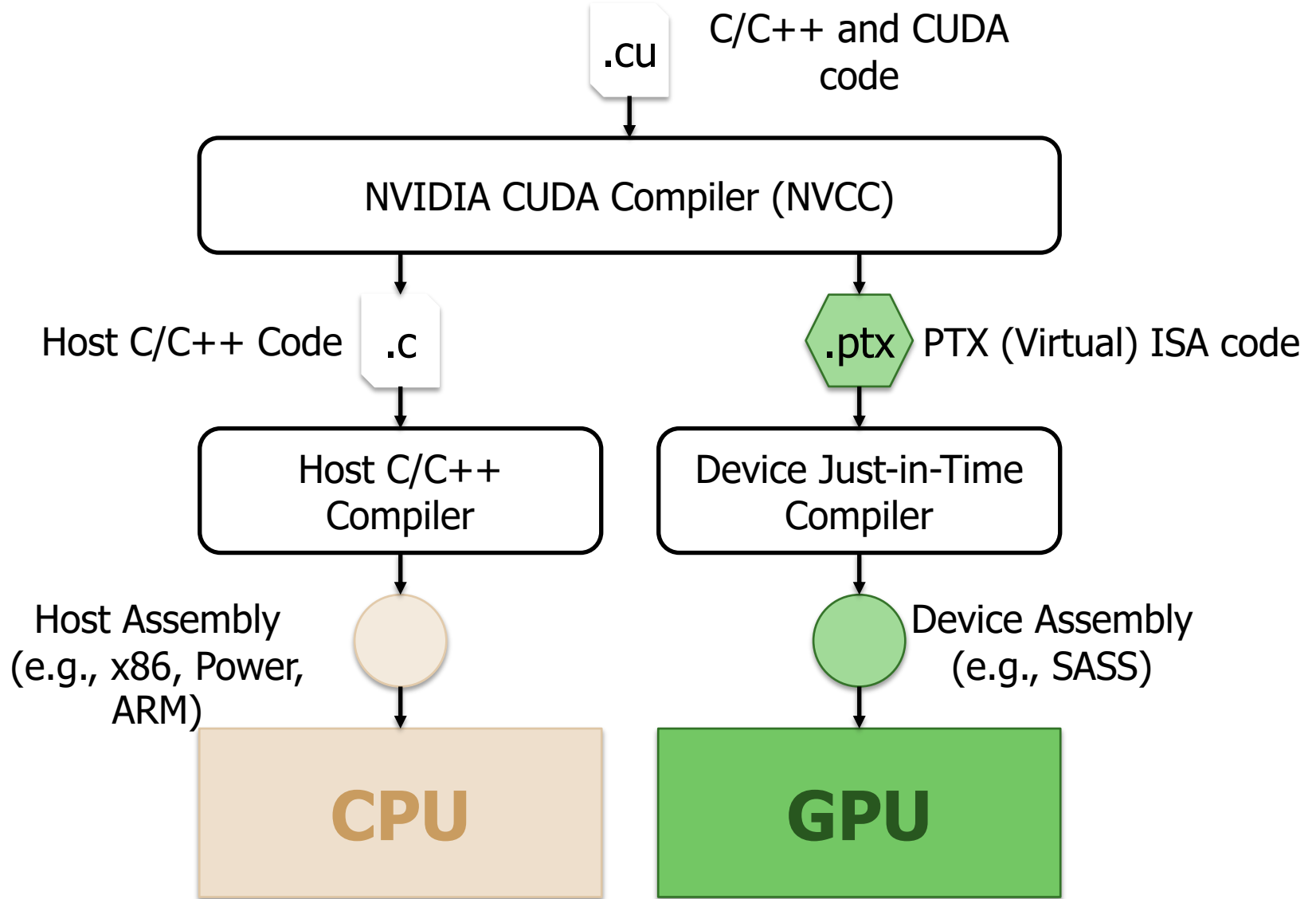
- What if the **size of the input is not a multiple of the number of threads** per block?
  - ❑ Solution: use the ceiling to launch extra threads then omit the threads after the boundary

```
const unsigned int numBlocks = (N + numThreadsPerBlock - 1) / numThreadsPerBlock;
```

## ■ Kernel code

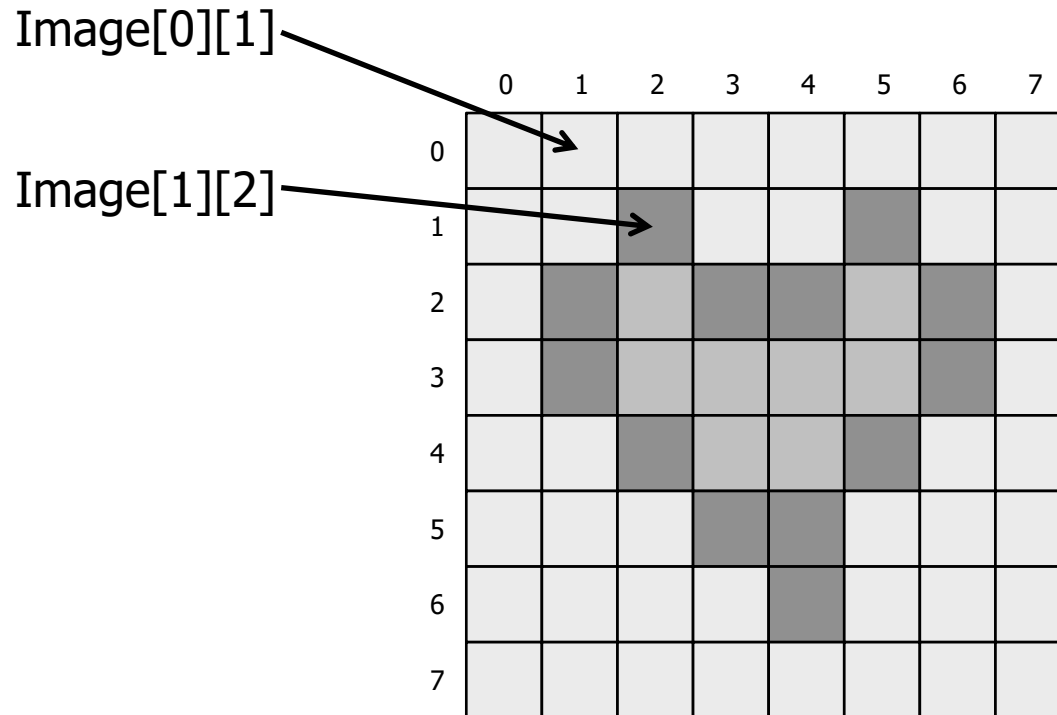
```
__global__ void vecadd_kernel(float* A, float* B, float* C, int N) {  
  
    int i = blockDim.x * blockIdx.x + threadIdx.x;  
  
    if(i < N) {  
        C[i] = A[i] + B[i];  
    }  
}
```

# Compilation



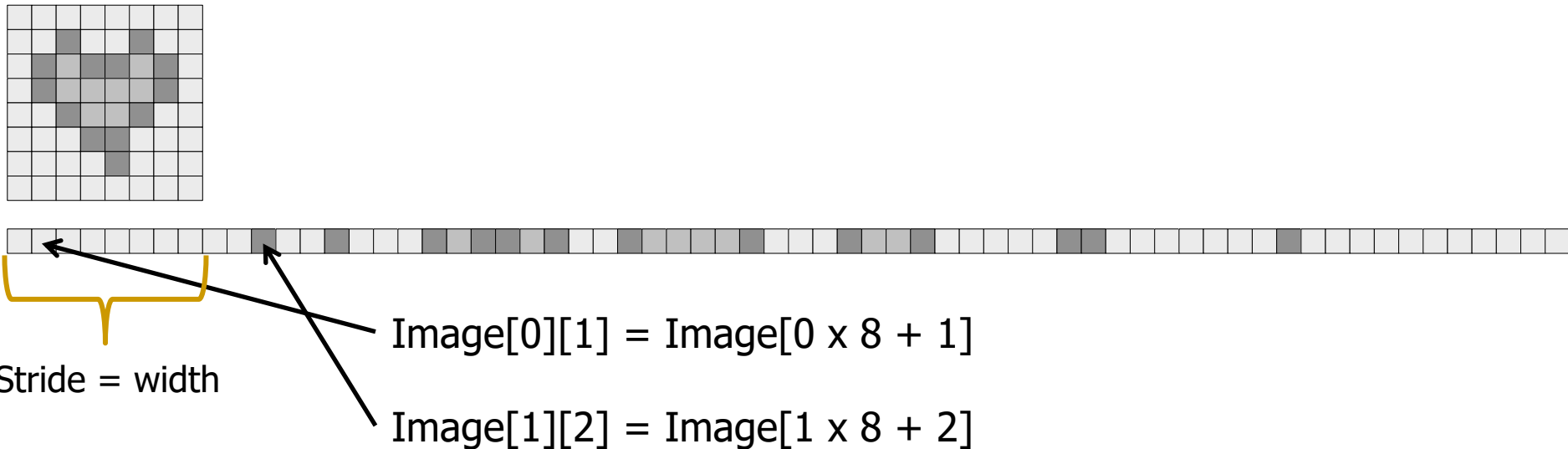
# Indexing and Memory Access

- Images are 2D data structures
  - height x width
  - `Image[j][i]`, where  $0 \leq j < \text{height}$ , and  $0 \leq i < \text{width}$



# Image Layout in Memory

- Row-major layout
- $\text{Image}[j][i] = \text{Image}[j \times \text{width} + i]$

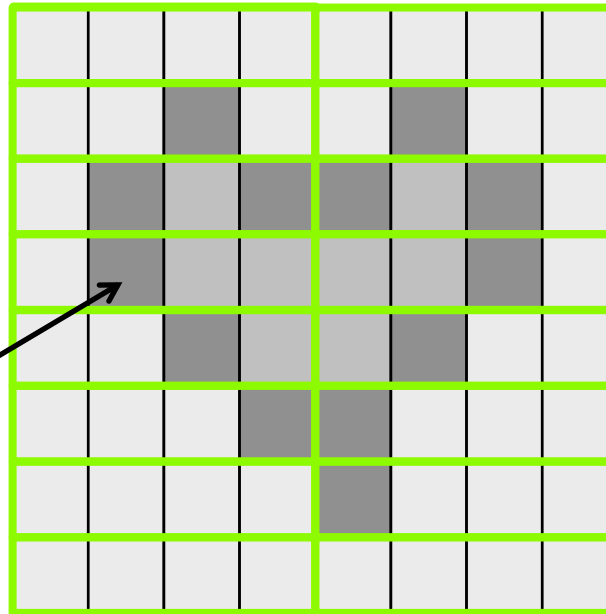


# Indexing and Memory Access: 1D Grid

- One GPU thread per pixel
- Grid of Blocks of Threads
  - `gridDim.x`, `blockDim.x`
  - `blockIdx.x`, `threadIdx.x`

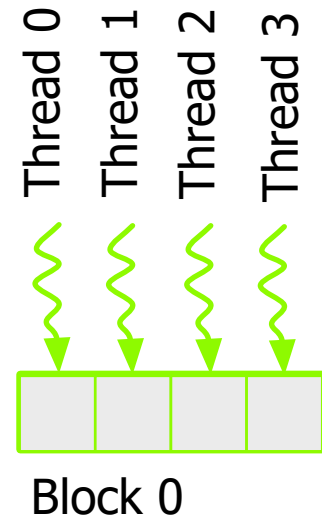
`blockIdx.x`  
`threadIdx.x`

Block 0



$$6 * 4 + 1 = 25$$

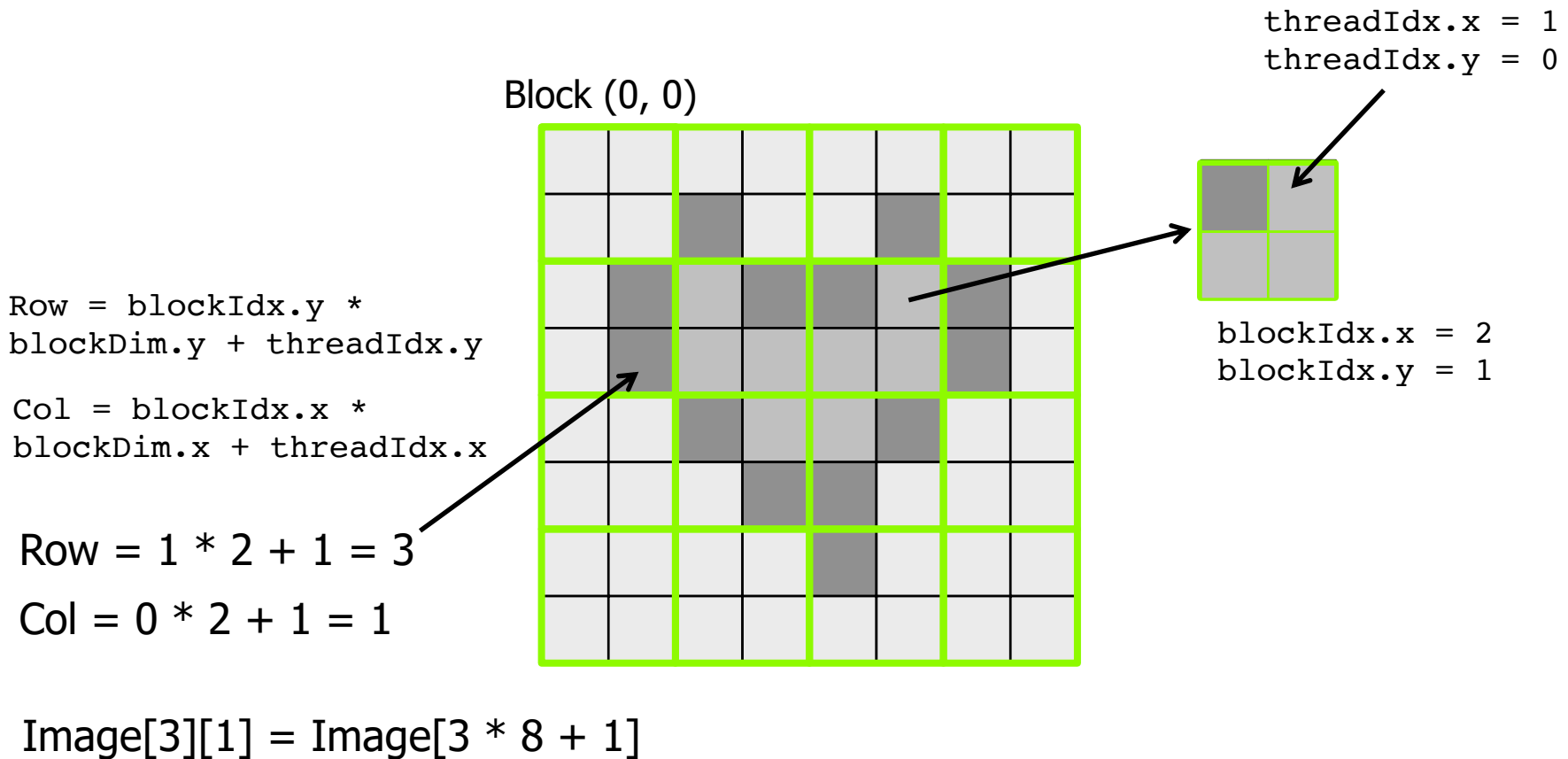
`blockIdx.x * blockDim.x + threadIdx.x`



# Indexing and Memory Access: 2D Grid

## ■ 2D blocks

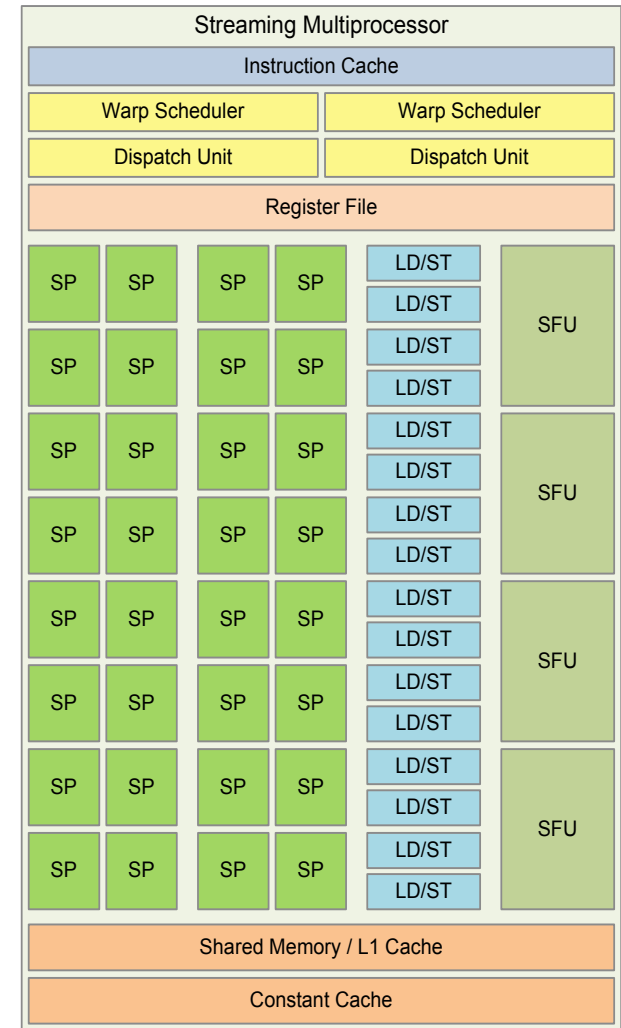
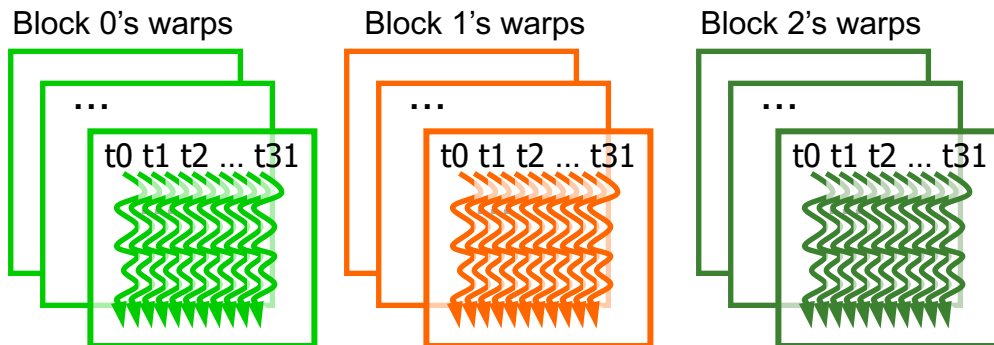
□ `gridDim.x`, `gridDim.y`





# Recall: From Blocks to Warps

- GPU cores: SIMD pipelines
  - ❑ Streaming Multiprocessors (SM)
  - ❑ Streaming Processors (SP)
- Blocks are divided into **warps**
  - ❑ SIMD unit (32 threads)

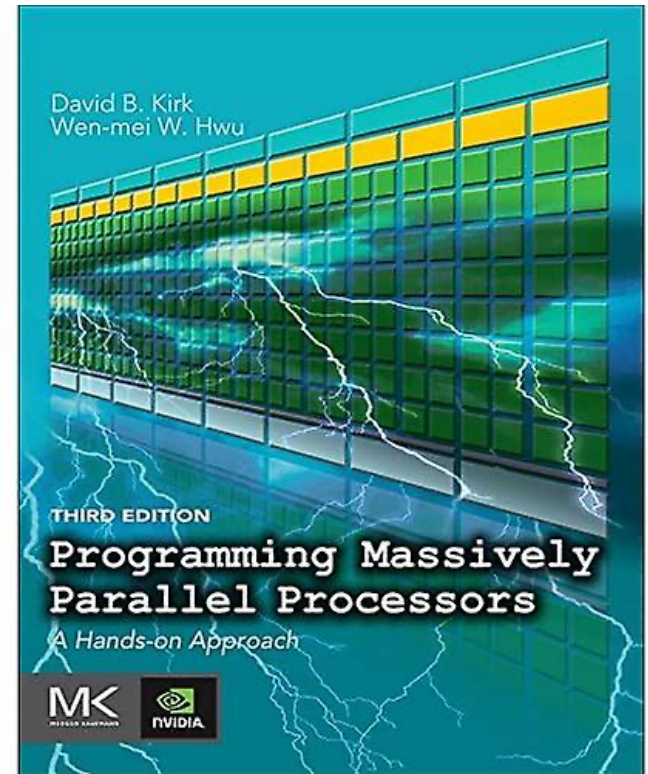


NVIDIA Fermi architecture

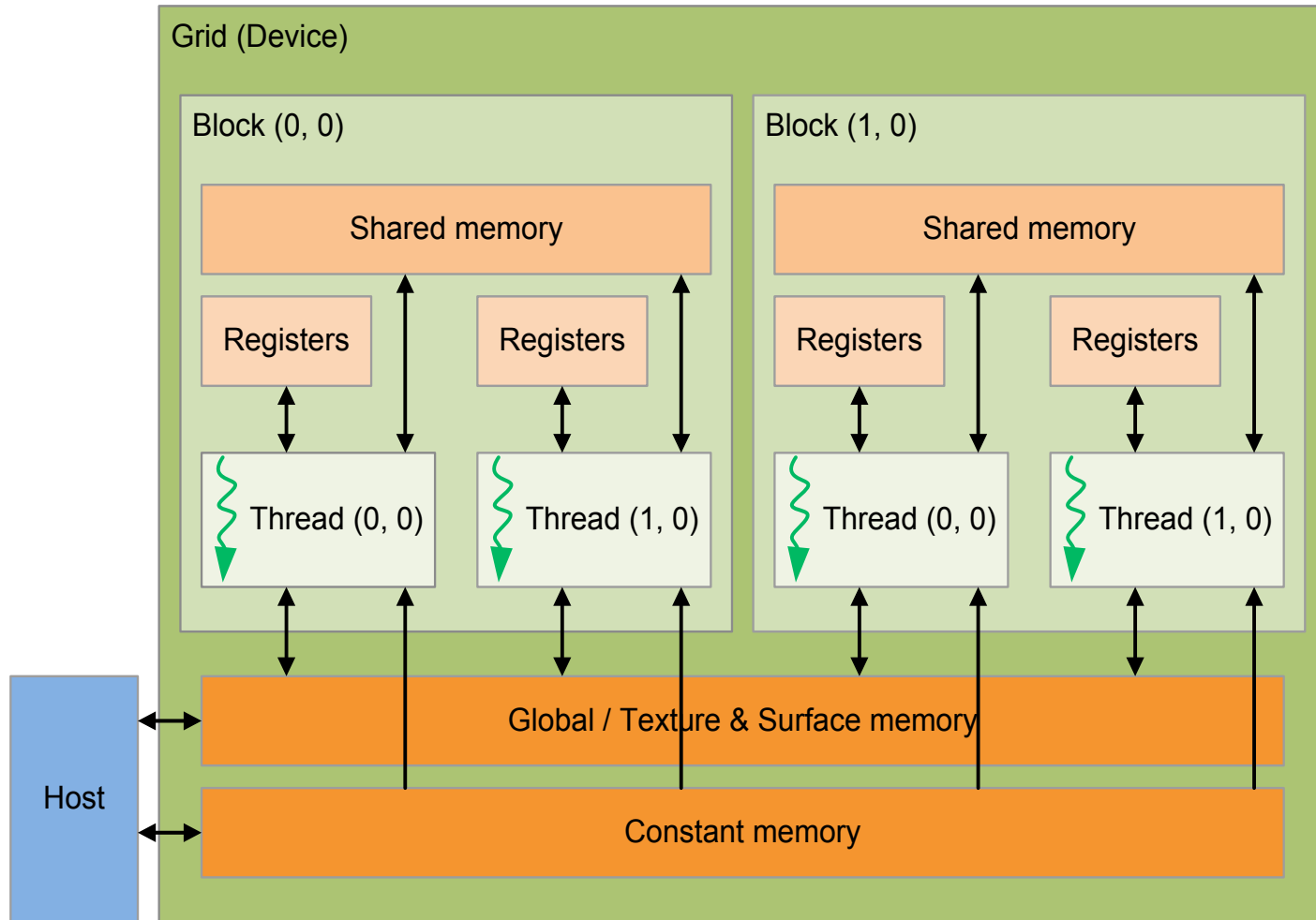
# Recommended Readings

---

- Hwu and Kirk, “**Programming Massively Parallel Processors,**” Third Edition, 2017
  - Chapter 1: Introduction
  - Chapter 2: Data parallel computing



# Memory Hierarchy



# P&S Heterogeneous Systems

## GPU Software Hierarchy:

Grids, Blocks, Threads

Dr. Juan Gómez Luna

Prof. Onur Mutlu

ETH Zürich

Fall 2021

21 October 2021