

P&S Heterogeneous Systems

GPU Memory Hierarchy

Dr. Juan Gómez Luna

Prof. Onur Mutlu

ETH Zürich

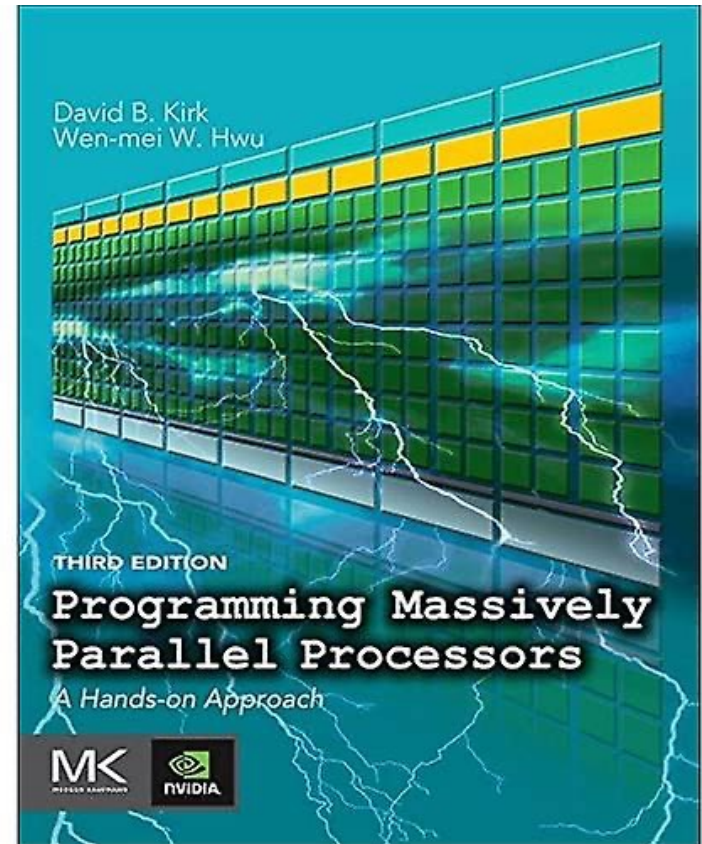
Fall 2022

24 October 2022

GPU Programming

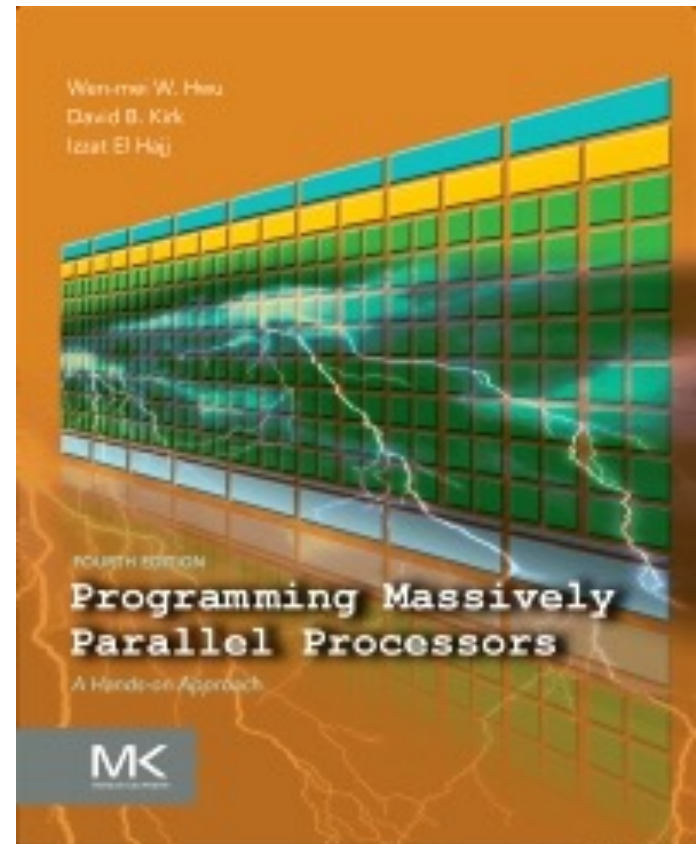
Recommended Readings (I)

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



Recommended Readings (II)

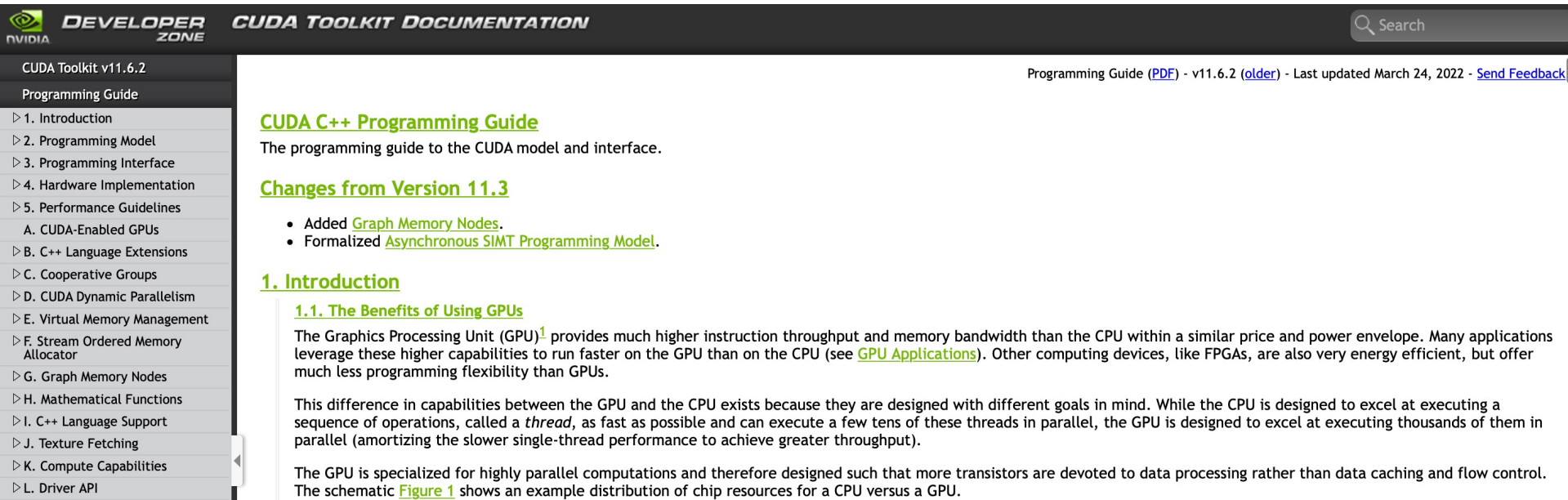
- Hwu, Kirk, El Hajj , “**Programming Massively Parallel Processors**,” Fourth Edition, 2022



Recommended Readings (III)

■ CUDA Programming Guide

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



The screenshot shows the NVIDIA Developer Zone documentation page for the CUDA Toolkit v11.6.2. The page title is "CUDA Toolkit v11.6.2" and the main heading is "CUDA C++ Programming Guide". 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), C++ Language Support, Texture Fetching, Compute Capabilities, and Driver API. The main content area is titled "CUDA C++ Programming Guide" and includes a sub-section "Changes from Version 11.3" with two bullet points: "Added Graph Memory Nodes" and "Formalized Asynchronous SIMT Programming Model". Below this is the "1. Introduction" section, which includes a sub-section "1.1. The Benefits of Using GPUs". The text in 1.1 states that the GPU provides higher instruction throughput and memory bandwidth than the CPU, and that it is designed to execute thousands of threads in parallel. A final paragraph explains that the GPU is specialized for highly parallel computations and that its architecture is designed to devote more transistors to data processing than to data caching and flow control, as shown in Figure 1.

DEVELOPER ZONE NVIDIA CUDA TOOLKIT DOCUMENTATION

Search

CUDA Toolkit v11.6.2

Programming Guide

Programming Guide (PDF) - v11.6.2 (older) - Last updated March 24, 2022 - 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

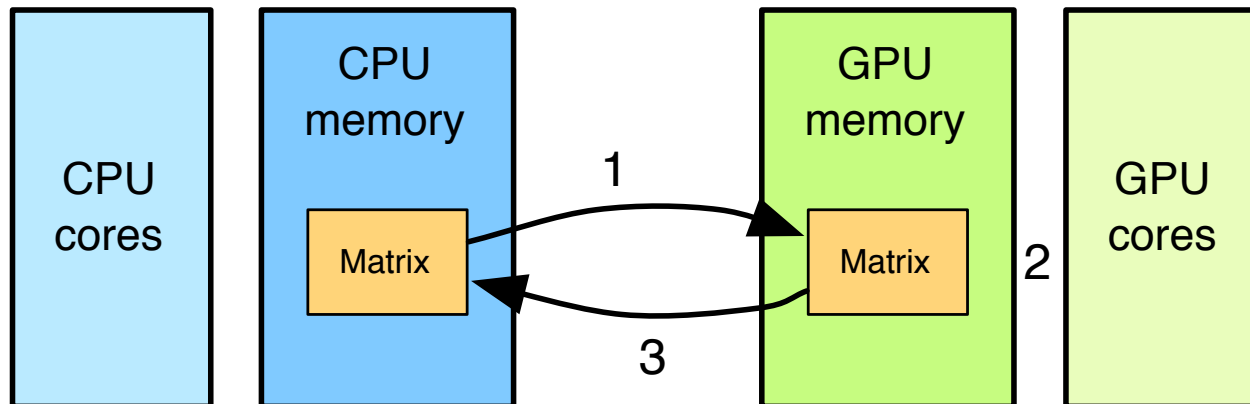
The Graphics Processing Unit (GPU)¹ provides much higher instruction throughput and memory bandwidth than the CPU within a similar price and power envelope. Many applications leverage these higher capabilities to run faster on the GPU than on the CPU (see [GPU Applications](#)). Other computing devices, 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.

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)

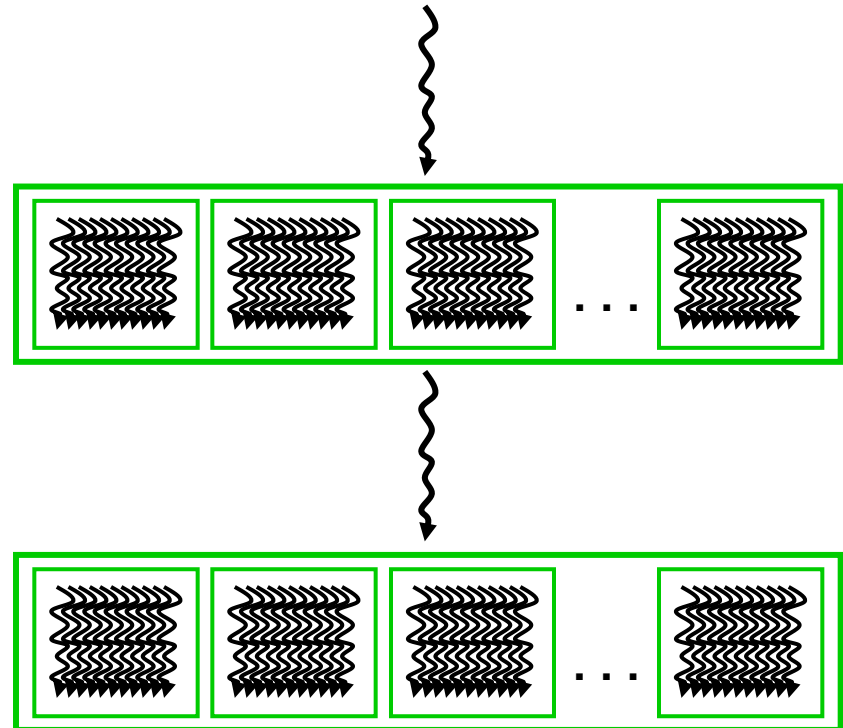
Parallel Kernel (device)

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

Serial Code (host)

Parallel Kernel (device)

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




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**

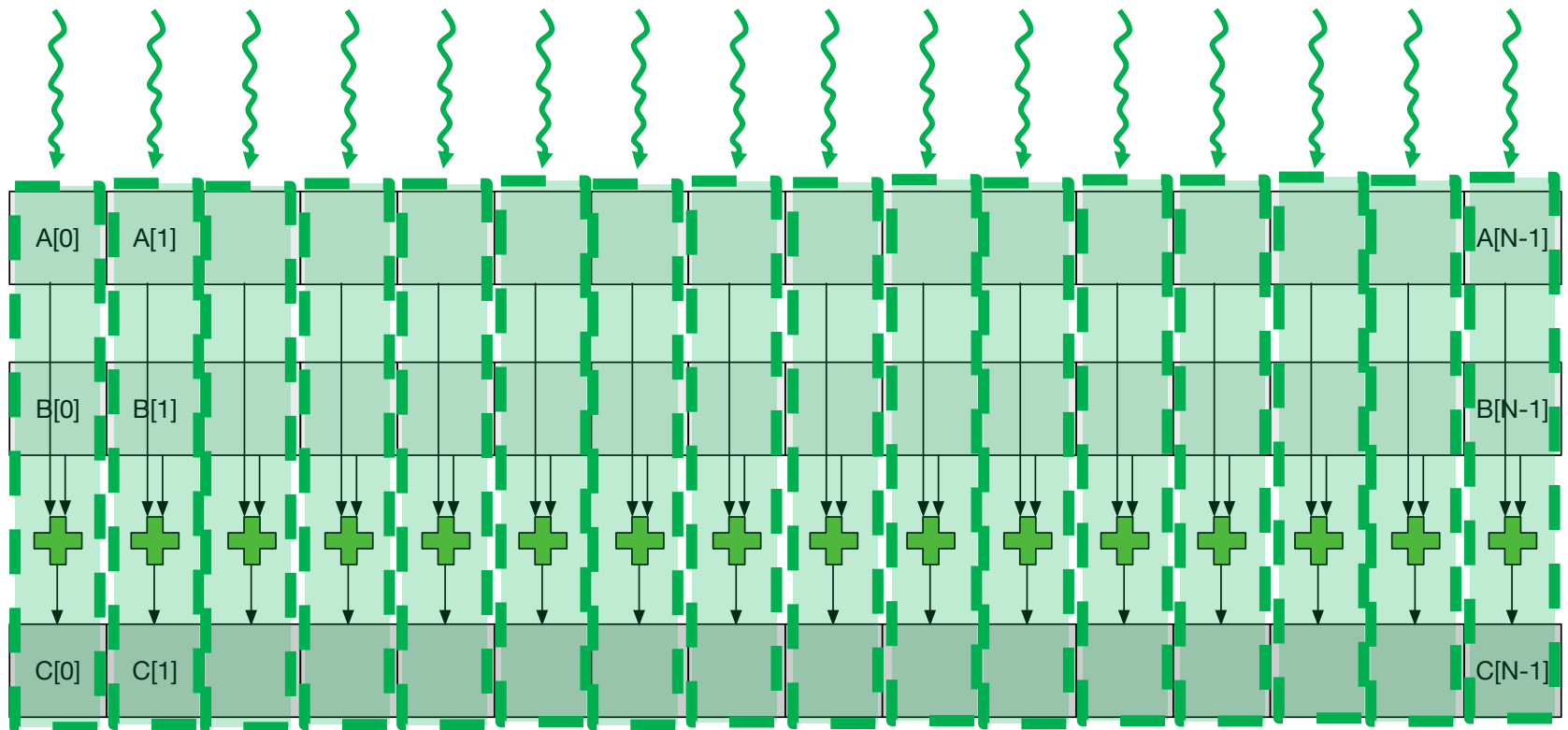
```
cudaFree(d_in);
```

- **Explicit synchronization**

```
cudaDeviceSynchronize();
```

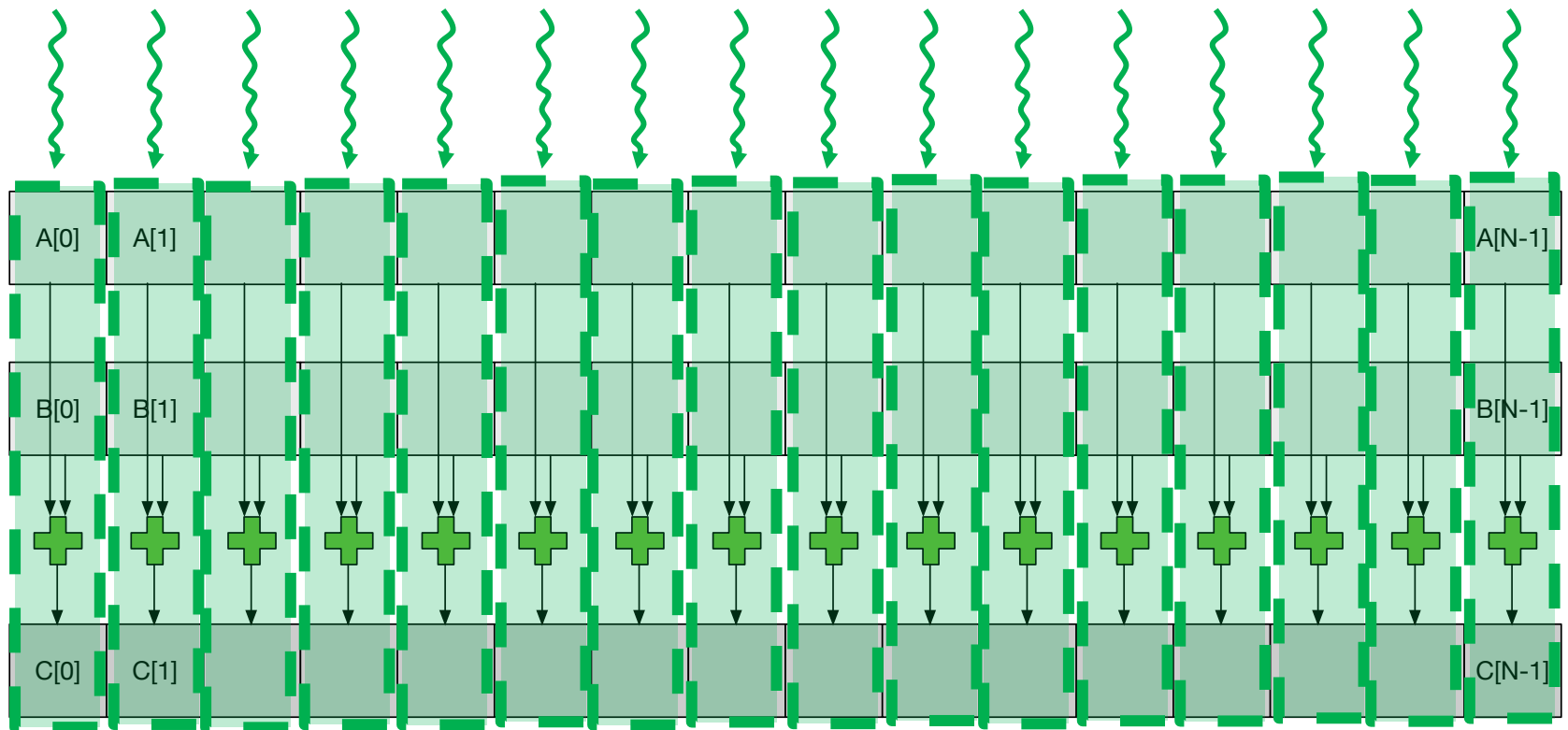
Vector Addition (I)

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



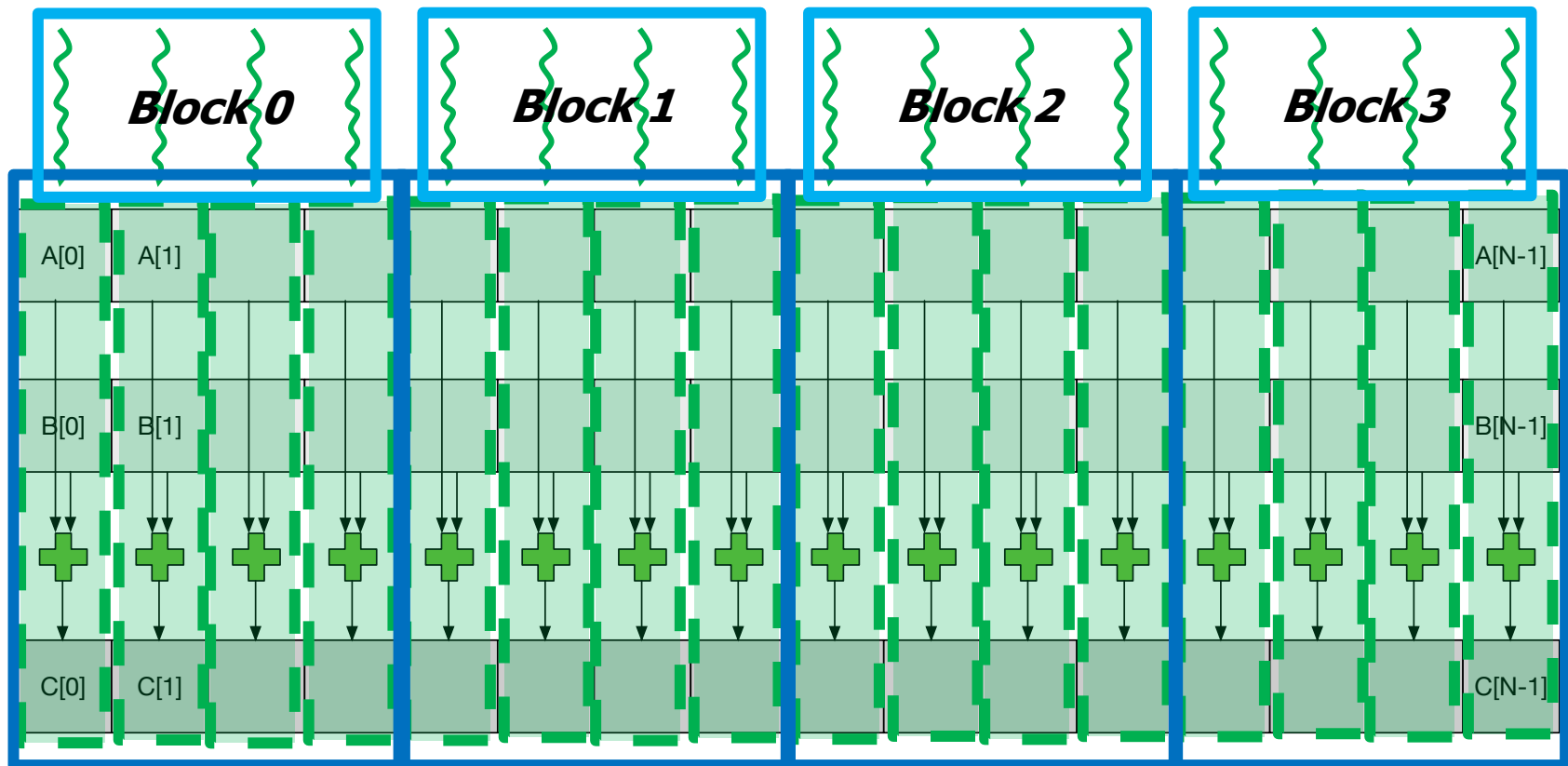
Vector Addition (II)

- 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**



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);  
}
```

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];  
    }  
}
```

Indexing and Memory Access

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

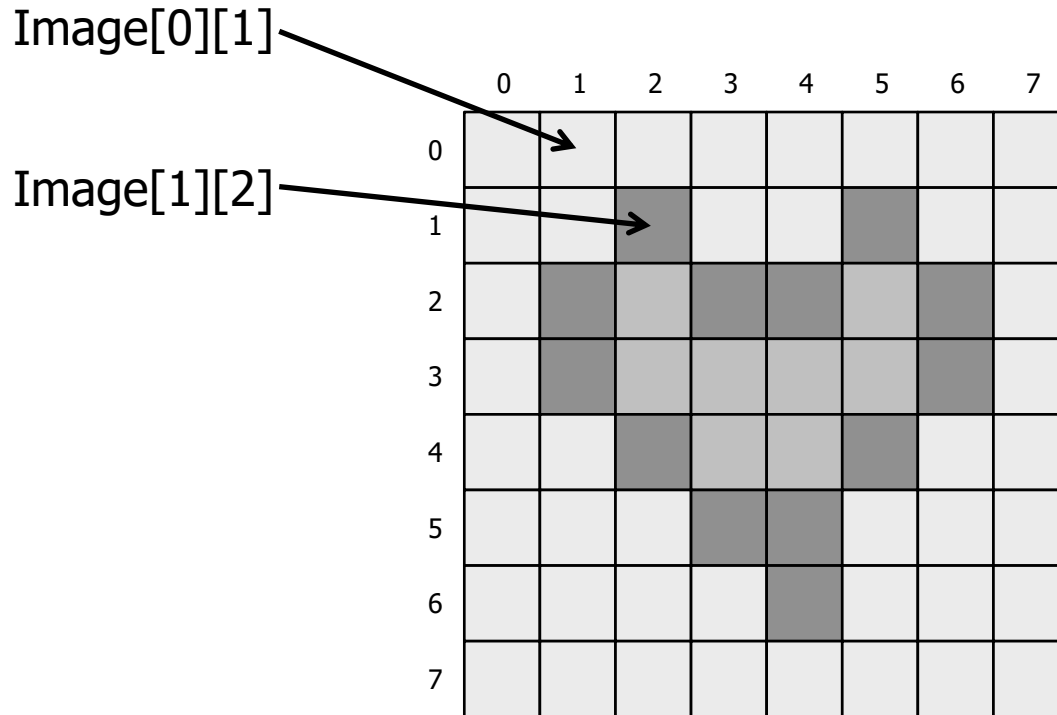
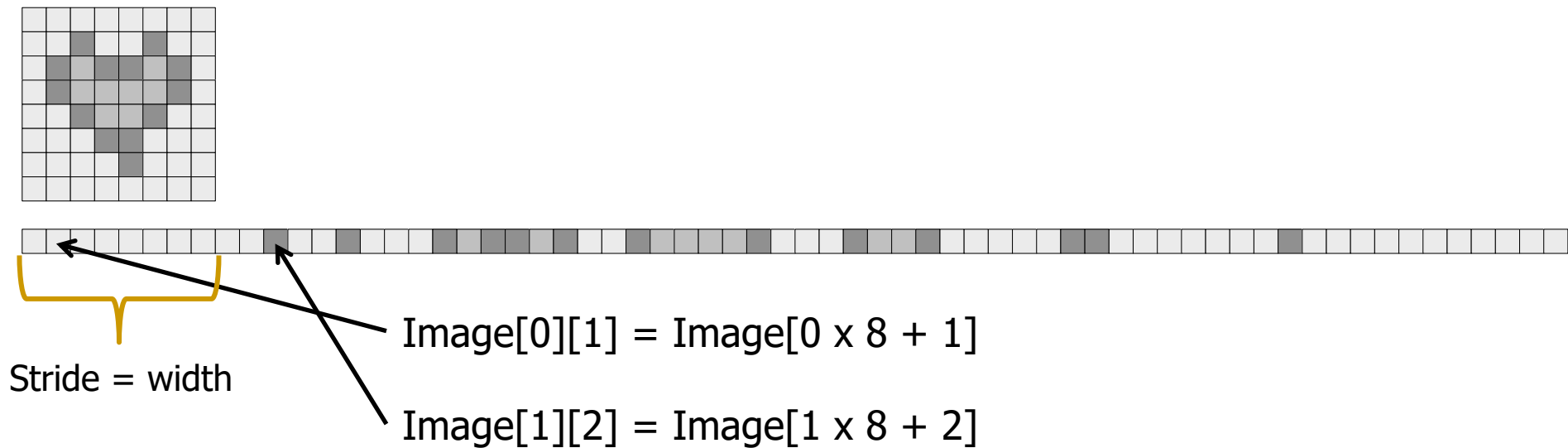


Image Layout in Memory

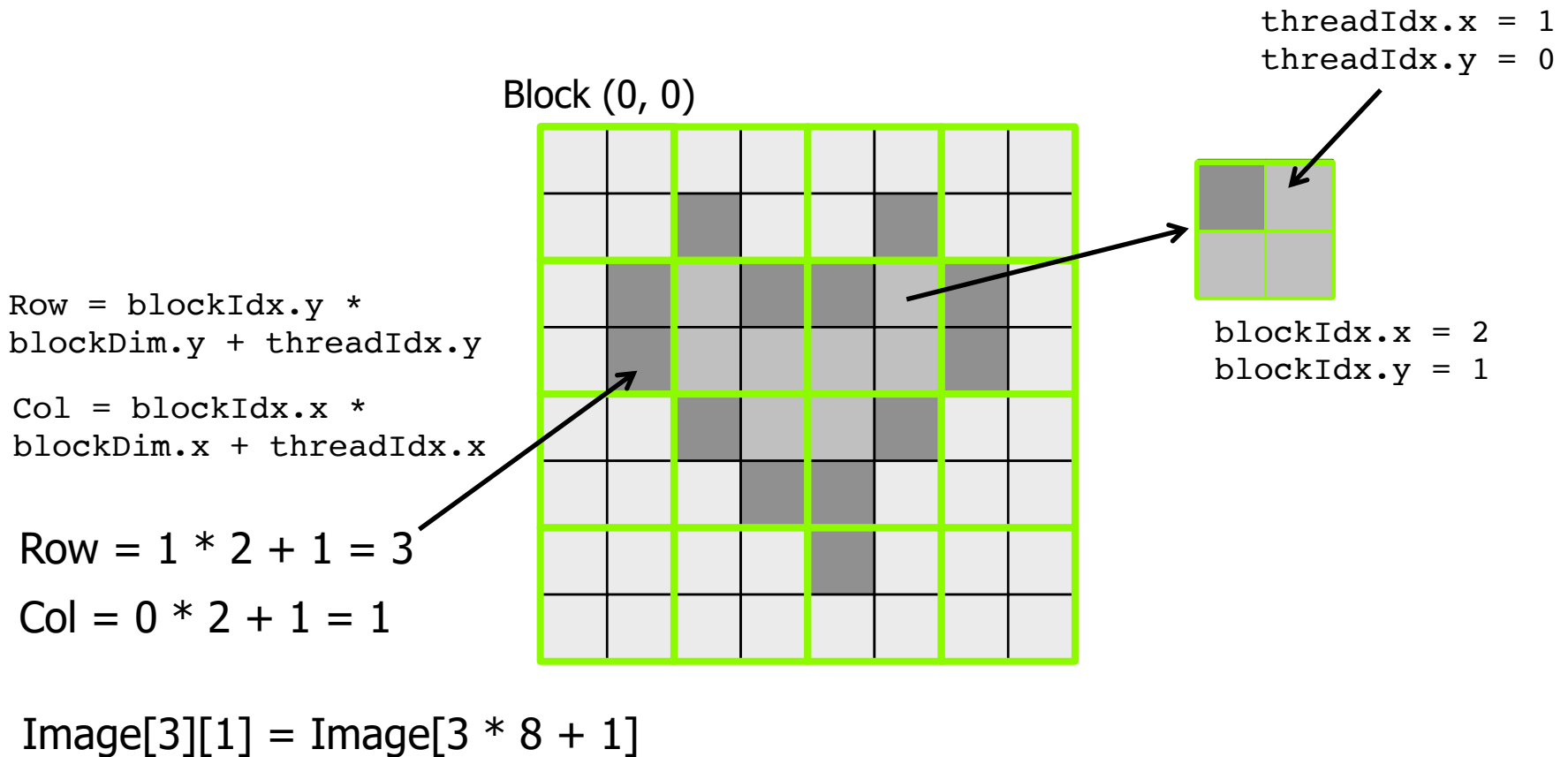
- Row-major layout
- $\text{Image}[j][i] = \text{Image}[j \times \text{width} + i]$



Indexing and Memory Access: 2D Grid

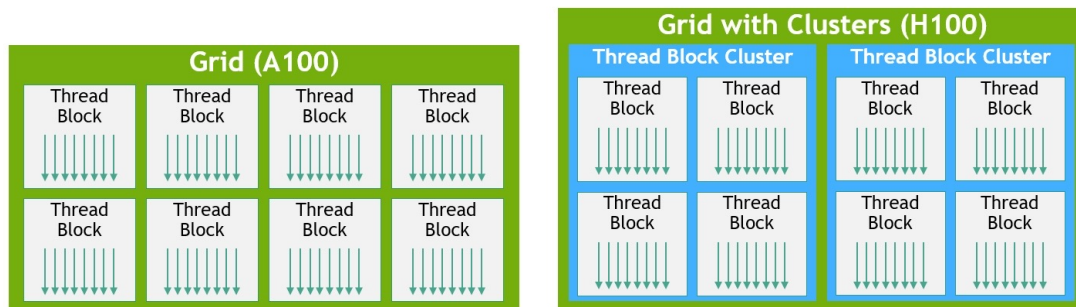
■ 2D blocks

- `gridDim.x`, `gridDim.y`



NVIDIA H100: Thread Block Clusters

- GPUs grow beyond 100 GPU cores (SMs): a new level in the software hierarchy can improve execution efficiency
 - Programmatic control of locality at a granularity larger than a single thread block on a single SM
- Thread blocks in the same cluster can synchronize and exchange data
- Thread blocks in the same cluster are guaranteed to be concurrently scheduled
 - Thread blocks in the same cluster run on the SMs within a GPU Processing Cluster (GPC)



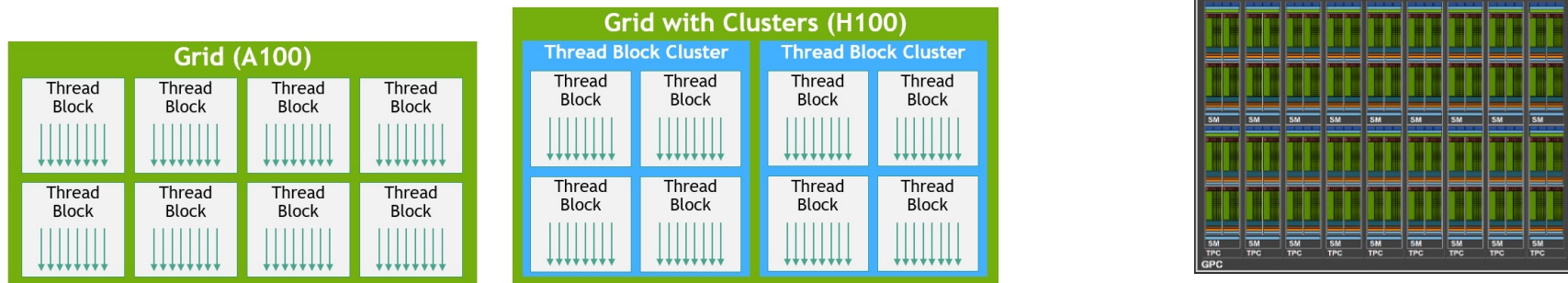
NVIDIA H100: Thread Block Clusters

- GPUs grow beyond 100 GPU cores (SMs): a new level in the software hierarchy can improve execution efficiency
 - Programmatic control of locality at a granularity larger than a single thread block on a single SM



NVIDIA H100: Thread Block Clusters

- GPUs grow beyond 100 GPU cores (SMs): a new level in the software hierarchy can improve execution efficiency
 - Programmatic control of locality at a granularity larger than a single thread block on a single SM
- Thread blocks in the same cluster can synchronize and exchange data
- Thread blocks in the same cluster are guaranteed to be concurrently scheduled
 - Thread blocks in the same cluster run on the SMs within a GPU Processing Cluster (GPC)
 - Data sharing via SM-to-SM network in a GPC



Thread - Thread block - Thread block cluster - Grid

GPU Memories

NVIDIA H100 Block Diagram



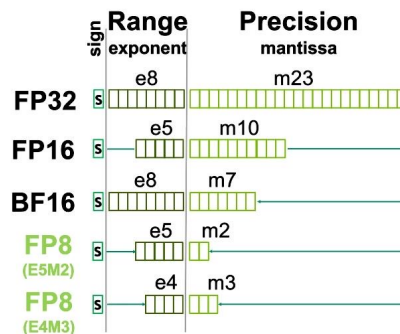
<https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/>

144 cores on the full GH100
60MB L2 cache

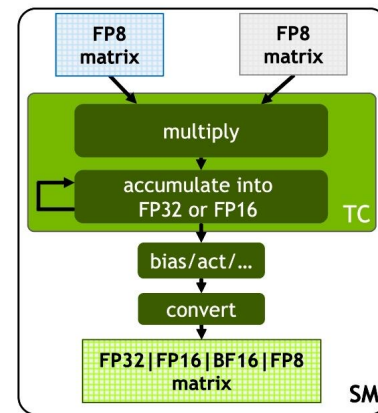
NVIDIA H100 Core



48 TFLOPS Single Precision*
 24 TFLOPS Double Precision*
 800 TFLOPS (FP16, Tensor Cores)*

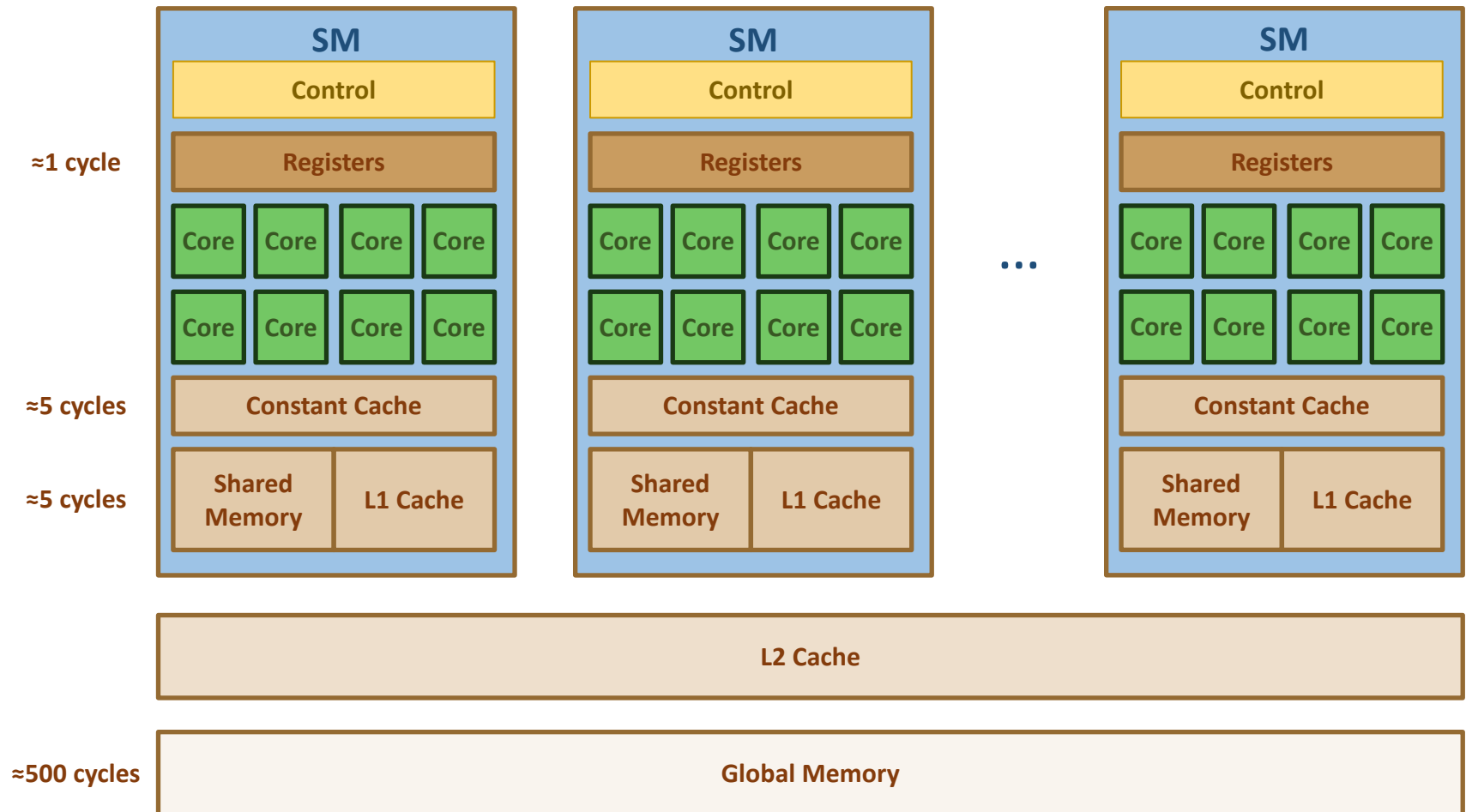


Allocate 1 bit to either range or precision



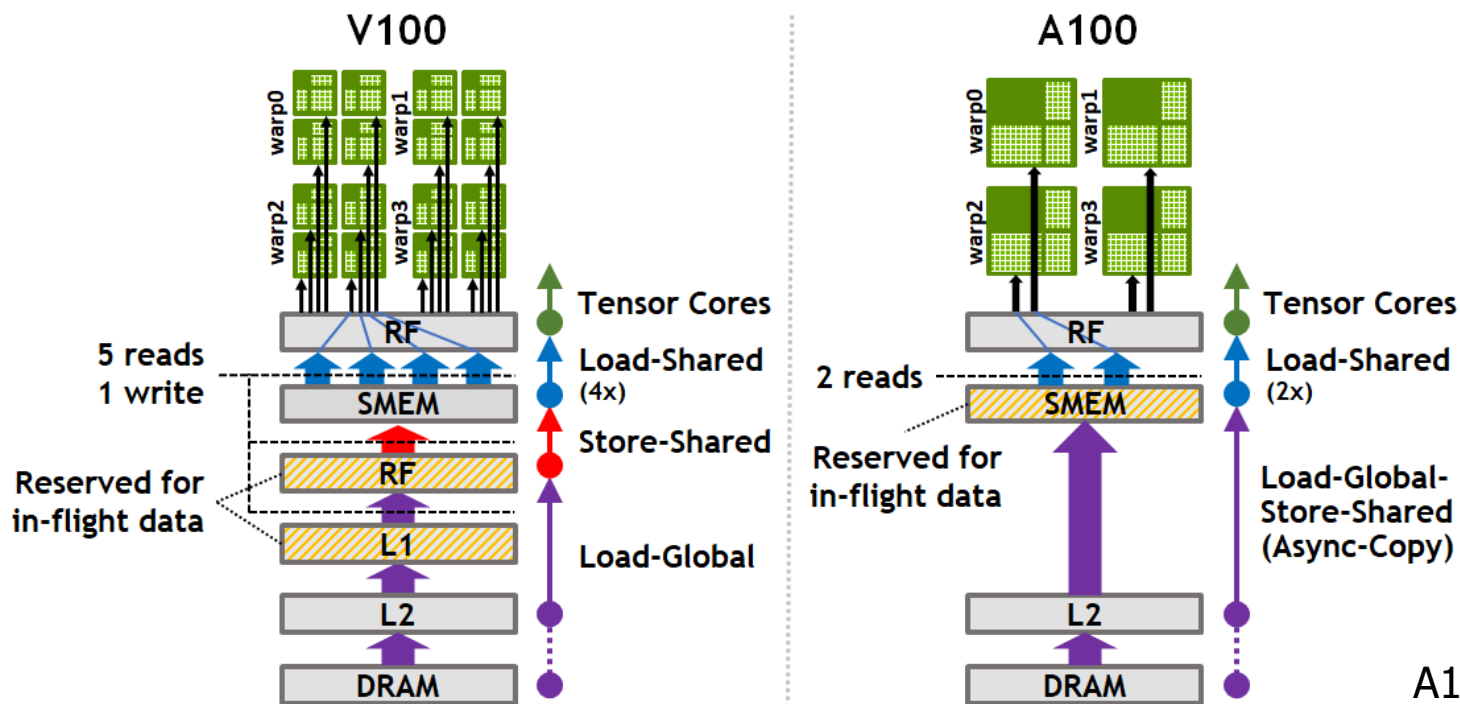
Support for multiple accumulator and output types

Memory in the GPU Architecture



NVIDIA V100 & A100 Memory Hierarchy

- Example of data movement between GPU global memory (DRAM) and GPU cores.

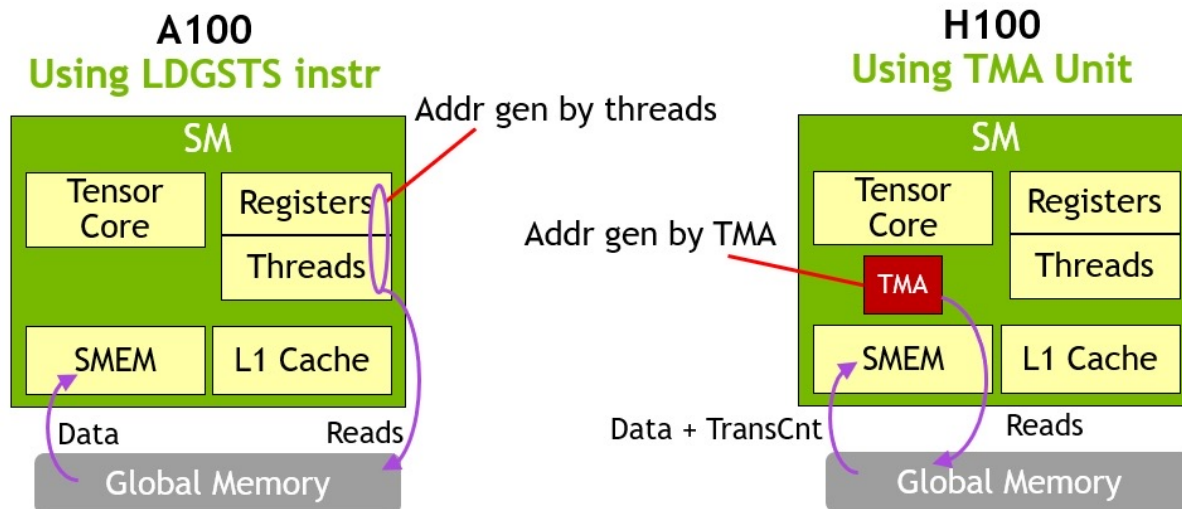


A100 improves SM bandwidth efficiency with a new load-global-store-shared asynchronous copy instruction that bypasses L1 cache and register file (RF). Additionally, A100's more efficient Tensor Cores reduce shared memory (SMEM) loads.

A100 feature:
Direct copy from L2 to scratchpad, bypassing L1 and register file.

NVIDIA H100 Tensor Memory Accelerator

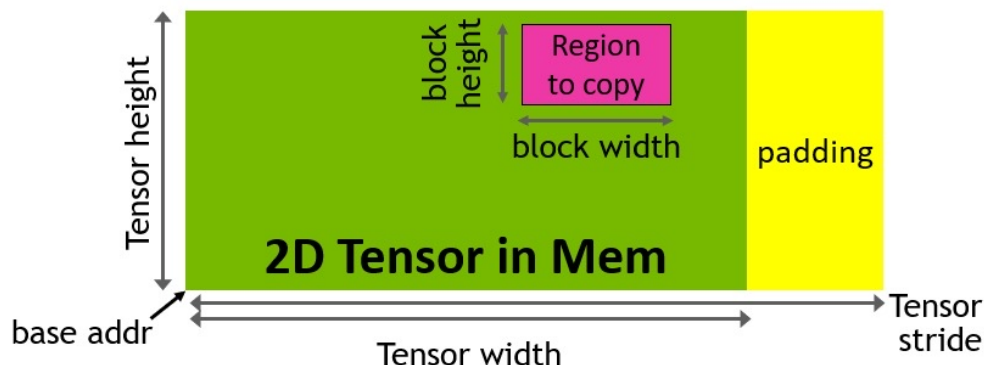
- Asynchronous memory copy with LDGSTS instruction vs. TMA



TMA unit reduces addressing overhead

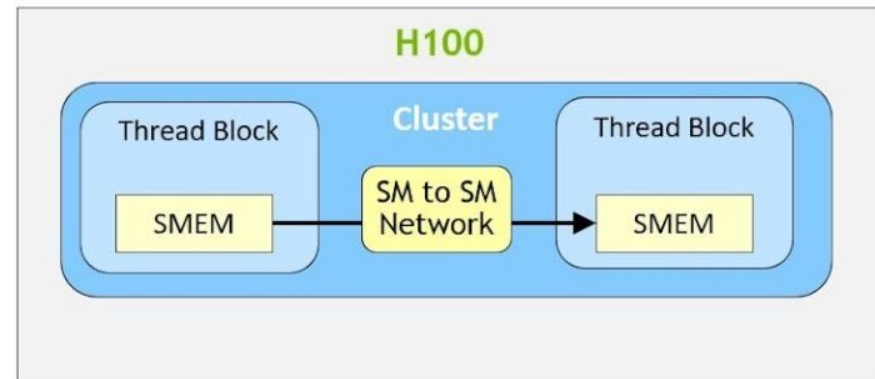
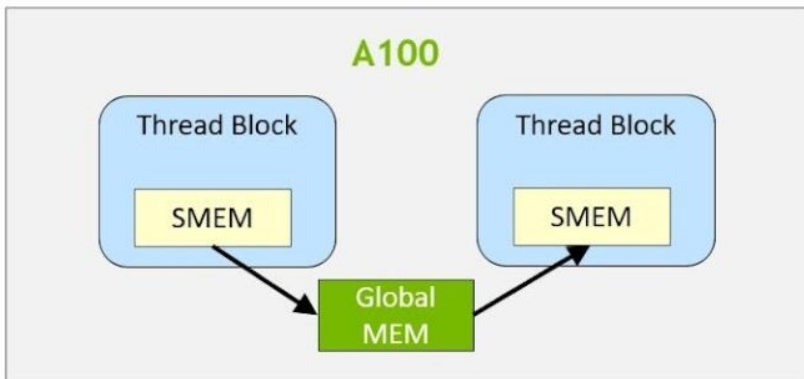
A single thread per warp issues the TMA operation

Support for different tensor layouts (1D-5D)



NVIDIA H100 Distributed Shared Memory

- Shared memory virtual address space distributed across the blocks of a cluster
- Load, store, and atomic operations to other SM's shared memory

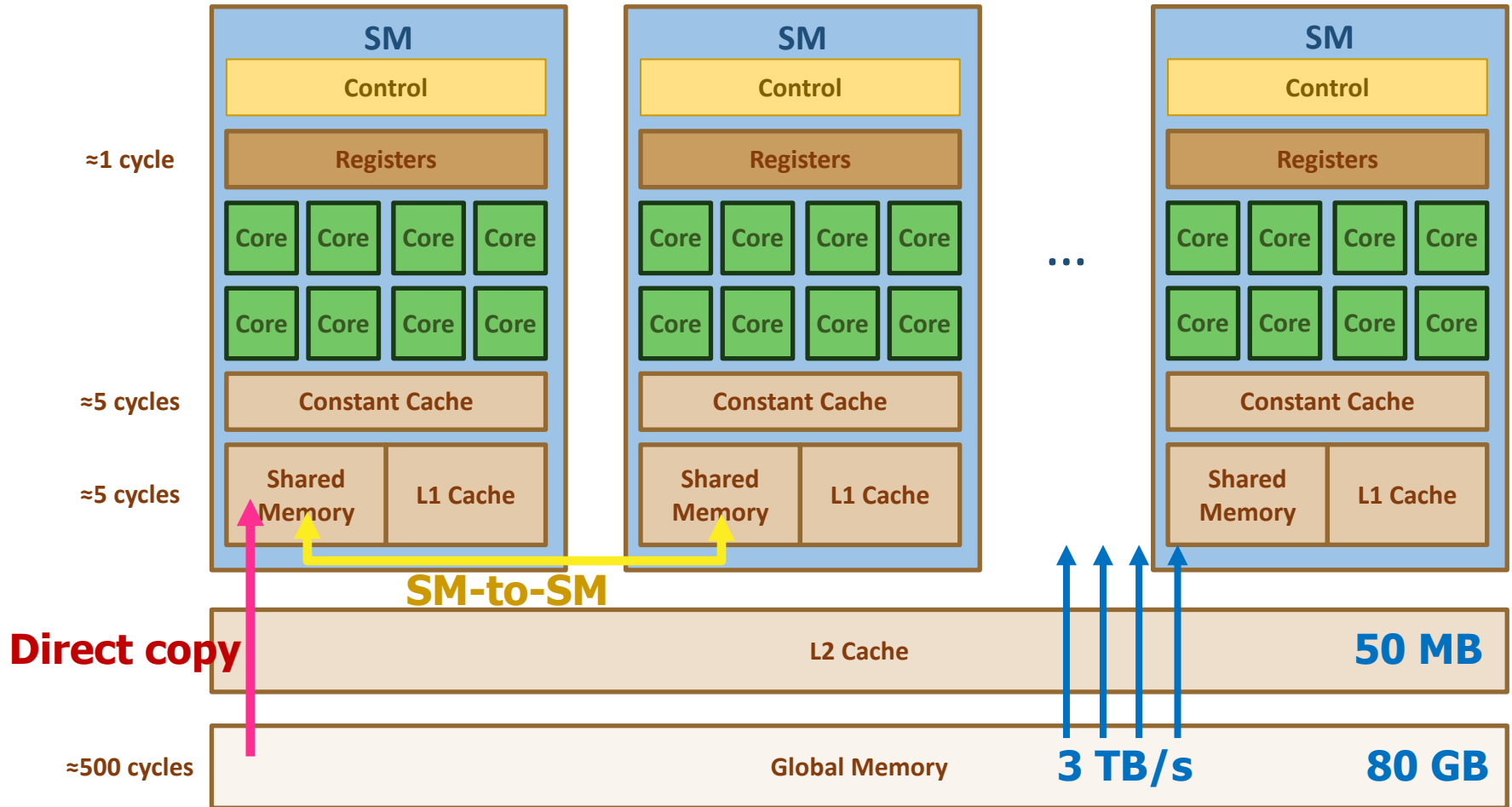


Thread block clusters and distributed shared memory (DSMEM) are leveraged via `cooperative_groups` API

TMA unit supports copies across thread blocks in a cluster

Asynchronous transaction barriers

Memory in the H100 GPU Architecture

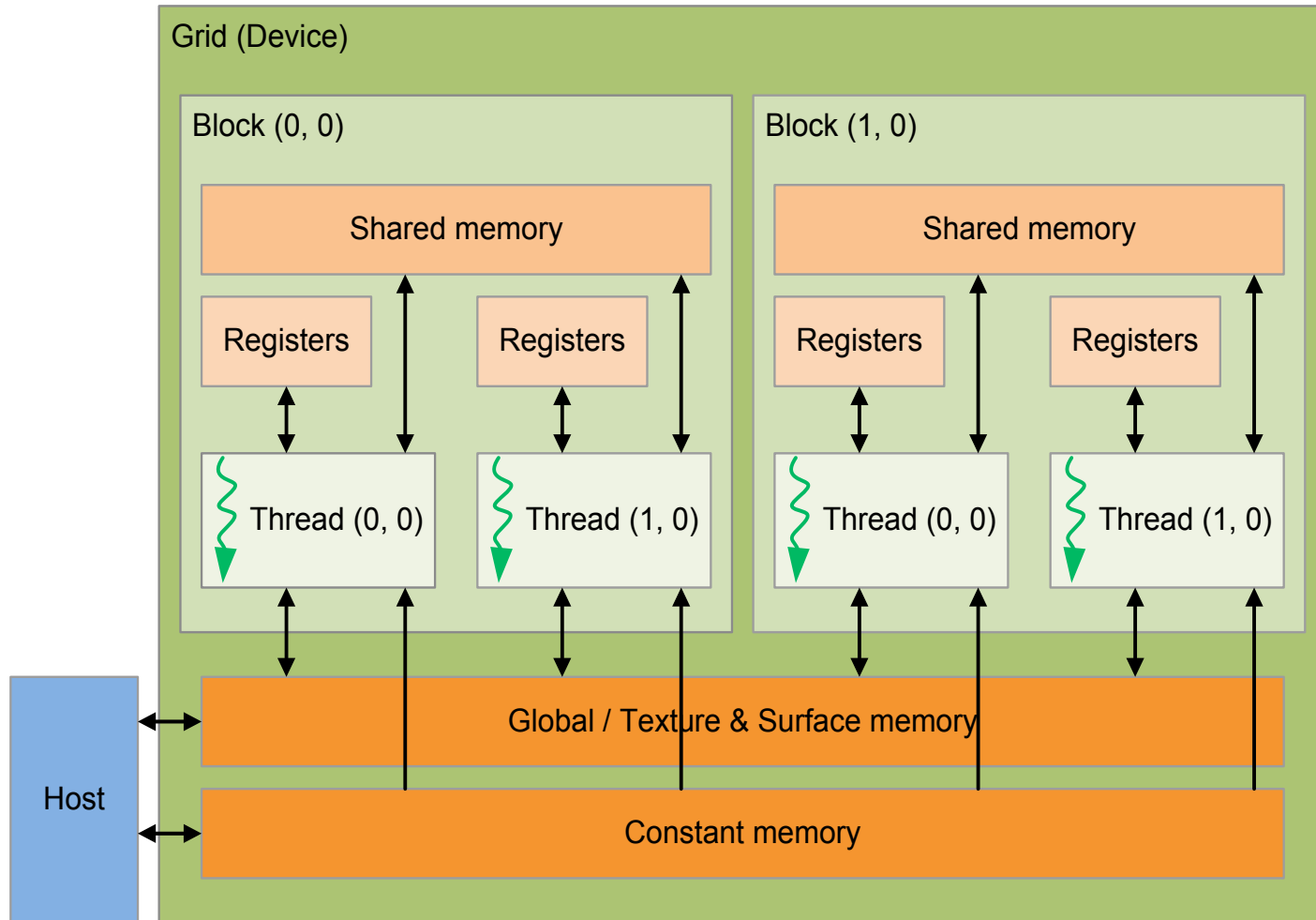


CUDA Variable Type Qualifiers

Variable declaration	Memory	Scope	Lifetime
<code>int LocalVar;</code>	register	thread	thread
<code>int localArr[N];</code>	global	thread	thread
<code>__device__ __shared__ int SharedVar;</code>	shared	block	block
<code>__device__ int GlobalVar;</code>	global	grid	application
<code>__device__ __constant__ int ConstantVar;</code>	constant	grid	application

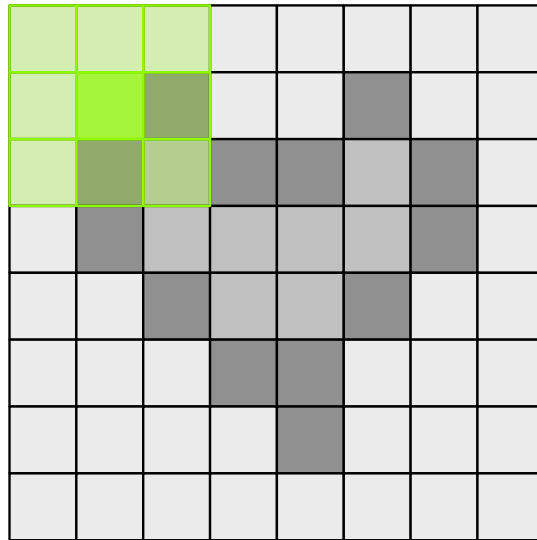
- `__device__` is optional when used with `__shared__`, or `__constant__`
- Recall `cudaMalloc(...)` allocates memory from the host
 - Constant memory can also be allocated and initialized from the host
- Automatic variables without any qualifier reside in a register
 - Except arrays that reside in global memory

Memory Hierarchy in CUDA Programs



Data Reuse

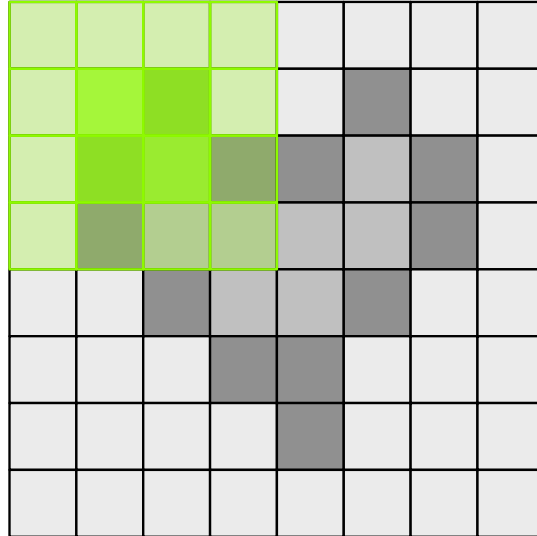
- Same memory locations accessed by neighboring threads



```
for (int i = 0; i < 3; i++){
    for (int j = 0; j < 3; j++){
        sum += gauss[i][j] * Image[(i+row-1)*width + (j+col-1)];
    }
}
```

Data Reuse: Tiling

- To take advantage of data reuse, we divide the input into **tiles** that can be loaded into **shared memory**



```
__shared__ int l_data[(L_SIZE+2)*(L_SIZE+2)];  
...  
Load tile into shared memory  
__syncthreads();  
for (int i = 0; i < 3; i++){  
    for (int j = 0; j < 3; j++){  
        sum += gauss[i][j] * l_data[(i+l_row-1)*(L_SIZE+2)+j+l_col-1];  
    }  
}
```


Synchronization Function

- `void __syncthreads () ;`
- Synchronizes all threads in a block
- Once all threads in a block have reached this point, execution resumes normally
- Used to avoid RAW / WAR / WAW hazards when accessing shared or global memory

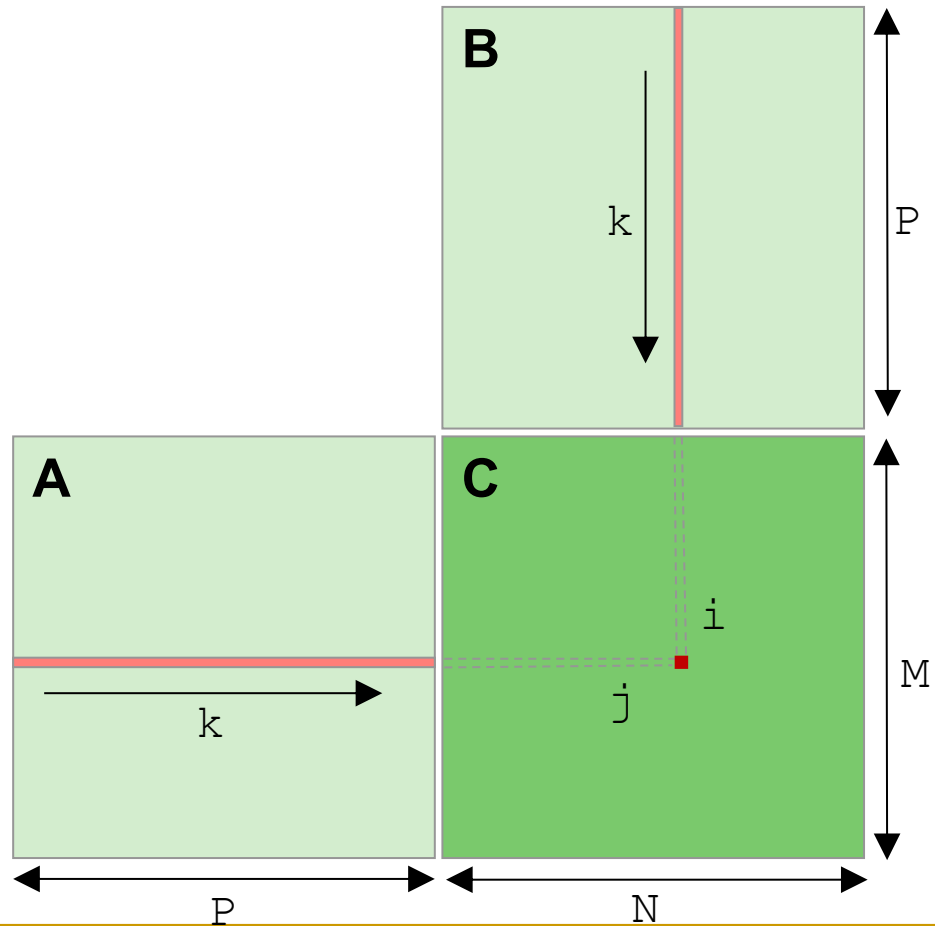
Tiling/Blocking in On-chip Memories

■ Tiling or Blocking

- Divide loops operating on arrays into computation chunks so that each chunk can hold its data in the cache (or other on-chip memory, e.g., scratchpad)
- Avoids cache conflicts between different chunks of computation
- Essentially: Divide the working set so that each piece fits in the cache
- Let's first see an example for CPUs

Naïve Matrix Multiplication (I)

- Matrix multiplication: $C = A \times B$
- Consider two input matrices A and B in row-major layout
 - A size is $M \times P$
 - B size is $P \times N$
 - C size is $M \times N$



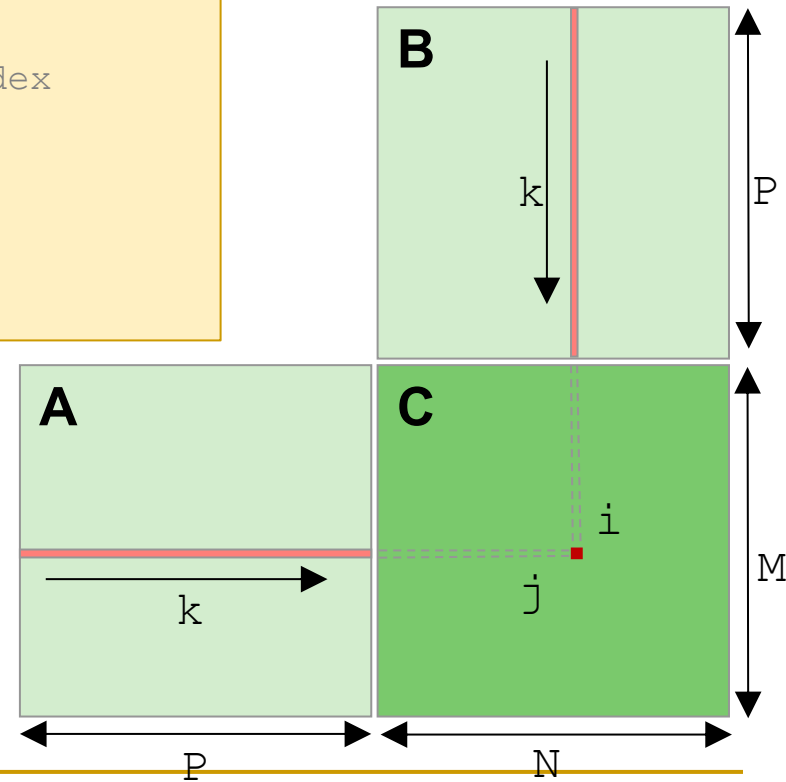
Naïve Matrix Multiplication (II)

- Naïve implementation of matrix multiplication has **poor cache locality**

```
#define A(i,j) matrix_A[i * P + j]
#define B(i,j) matrix_B[i * N + j]
#define C(i,j) matrix_C[i * N + j]

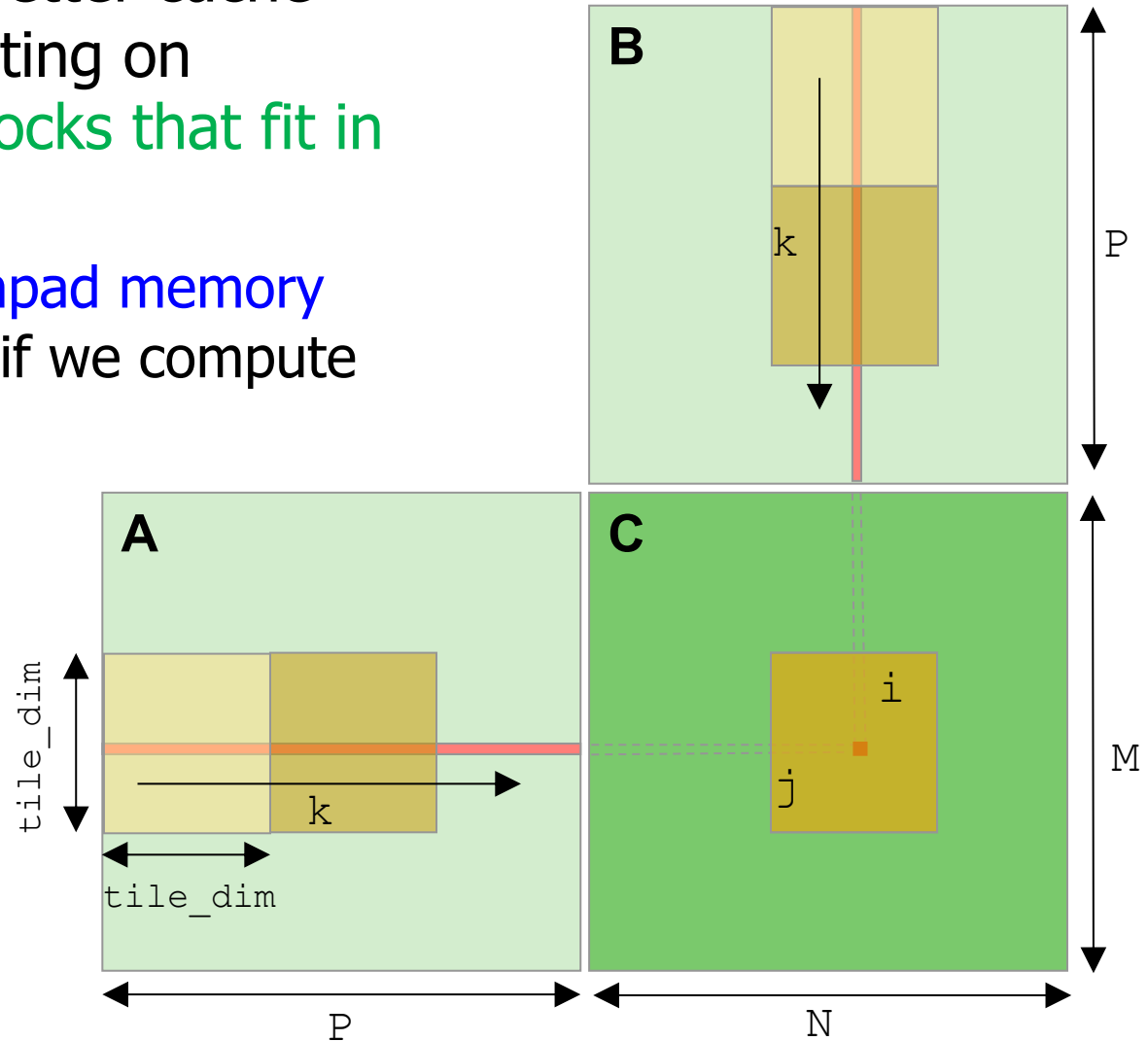
for (i = 0; i < M; i++){ // i = row index
    for (j = 0; j < N; j++){ // j = column index
        C(i, j) = 0; // Set to zero
        for (k = 0; k < P; k++) // Row x Col
            C(i, j) += A(i, k) * B(k, j);
    }
}
```

Consecutive accesses to B are far from each other, in **different cache lines**.
Every access to B is likely to cause a **cache miss**



Tiled Matrix Multiplication (I)

- We can achieve better cache locality by computing on smaller tiles or blocks that fit in the cache
 - Or in the scratchpad memory and register file if we compute on a GPU



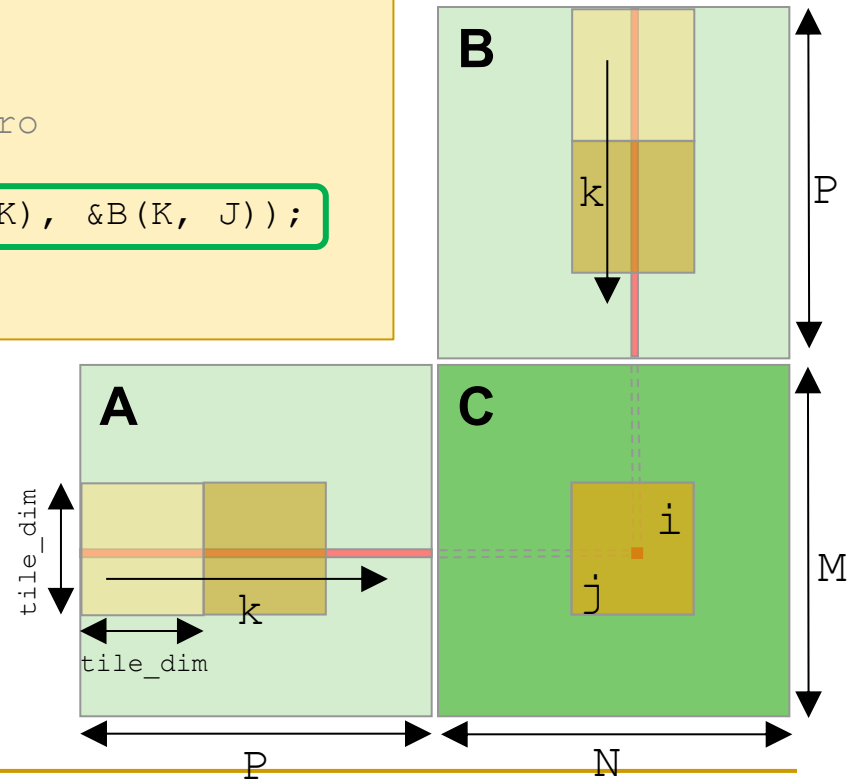
Tiled Matrix Multiplication (II)

- **Tiled implementation** operates on submatrices (tiles or blocks) that fit fast memories (cache, scratchpad, RF)

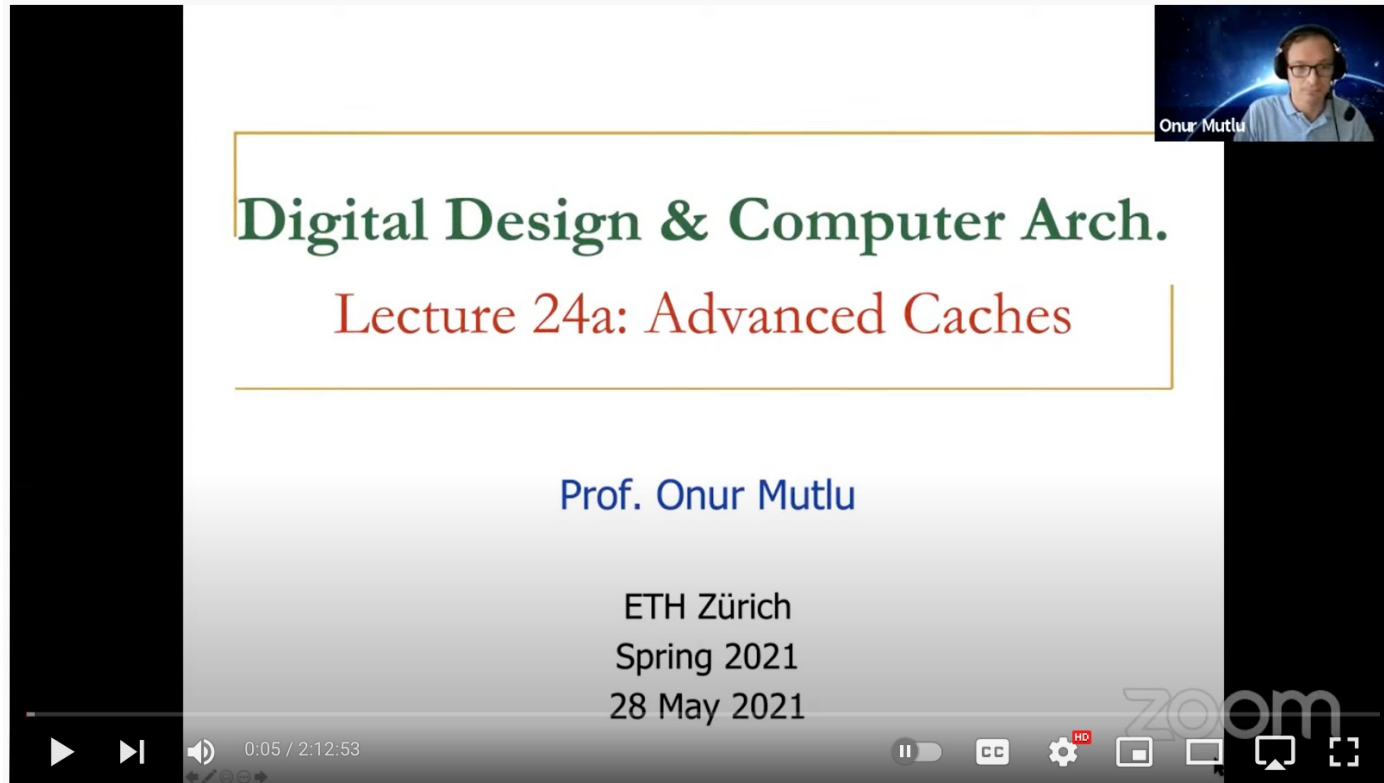
```
#define A(i,j) matrix_A[i * P + j]
#define B(i,j) matrix_B[i * N + j]
#define C(i,j) matrix_C[i * N + j]

for (I = 0; I < M; I += tile_dim){
  for (J = 0; J < N; J += tile_dim){
    Set_to_zero(&C(I, J)); // Set to zero
    for (K = 0; K < P; K += tile_dim)
      Multiply_tiles(&C(I, J), &A(I, K), &B(K, J));
  }
}
```

Multiply small submatrices (tiles or blocks) of size `tile_dim` x `tile_dim`



Lecture on Advanced Caches



The image shows a Zoom video player interface. In the top right corner, there is a small video thumbnail of Prof. Onur Mutlu. The main content area displays a slide with the following text:

Digital Design & Computer Arch.
Lecture 24a: Advanced Caches

Prof. Onur Mutlu

ETH Zürich
Spring 2021
28 May 2021

At the bottom of the slide, there is a 'zoom' watermark. Below the slide, the video player controls are visible, showing a progress bar at 0:05 / 2:12:53 and various control icons like play, volume, and full screen.

DEPARTMENT OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING (D-ITET)

Digital Design & Comp. Arch. - Lecture 24: Advanced Caches (ETH Zürich, Spring 2021) - Onur Mutlu

2,958 views • Streamed live on May 28, 2021

75 1 SHARE SAVE ...



Onur Mutlu Lectures

19.6K subscribers

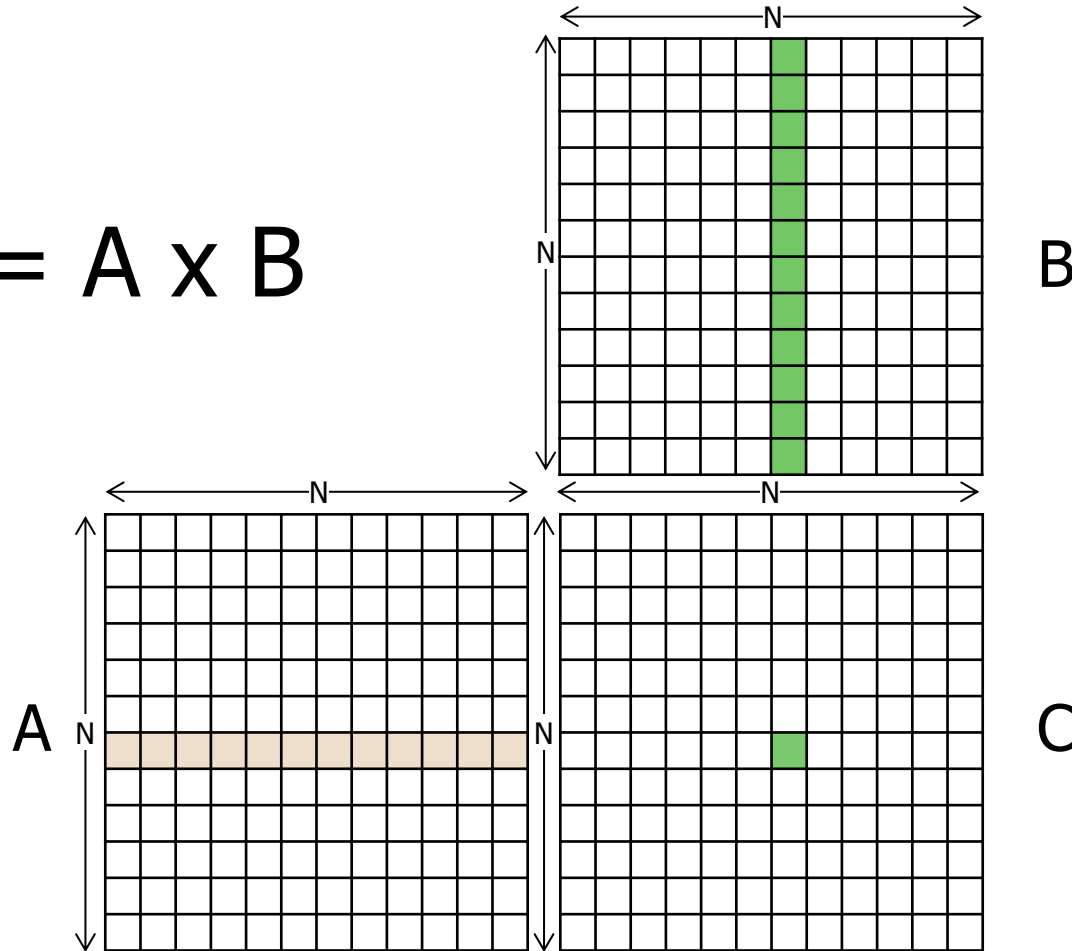
ANALYTICS

EDIT VIDEO

Digital Design and Computer Architecture, ETH Zürich, Spring 2021 (
<https://safari.ethz.ch/digitaltechnik...>)

Example: Matrix-Matrix Multiplication (I)

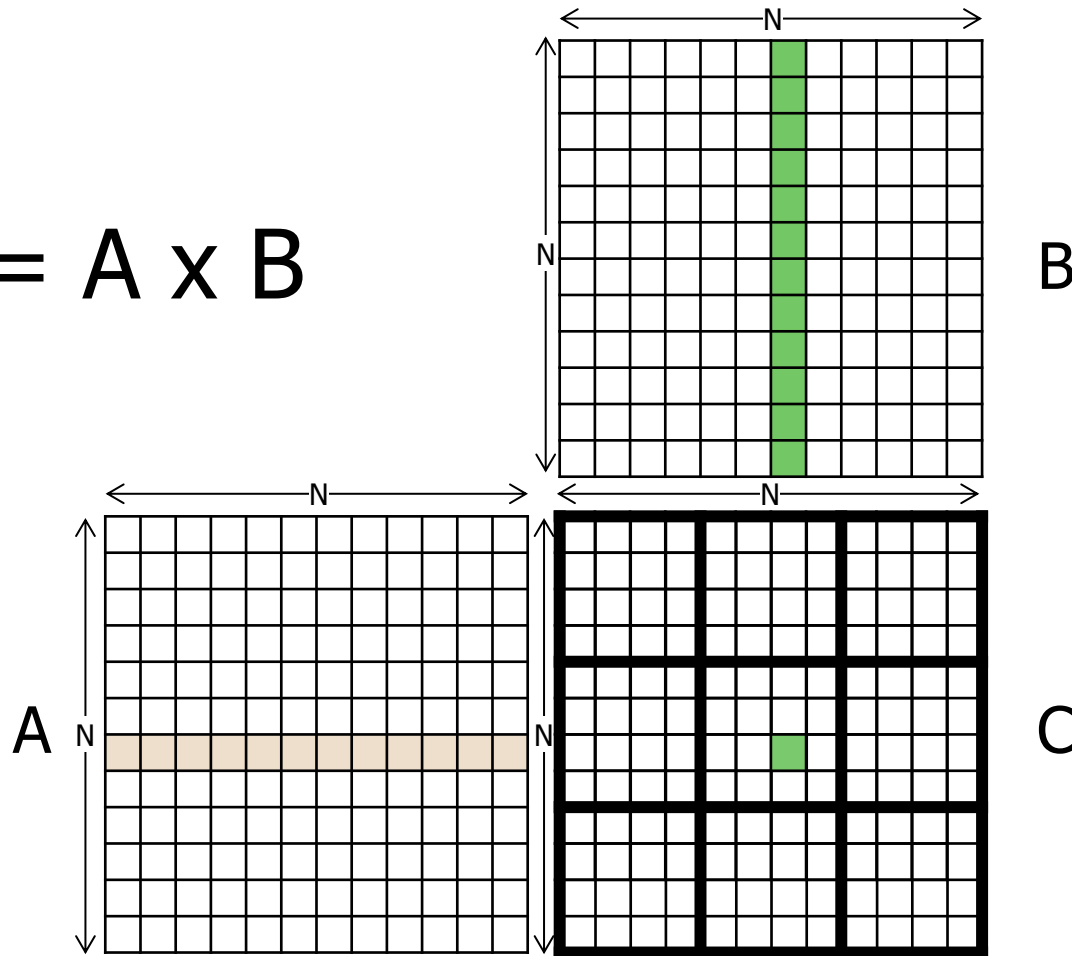
$$C = A \times B$$



Example: Matrix-Matrix Multiplication (II)

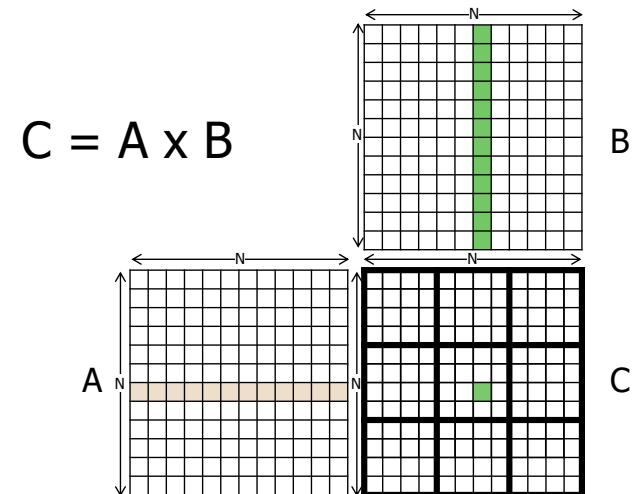
Parallelization approach: assign one thread to each element in the output matrix (C)

$$C = A \times B$$



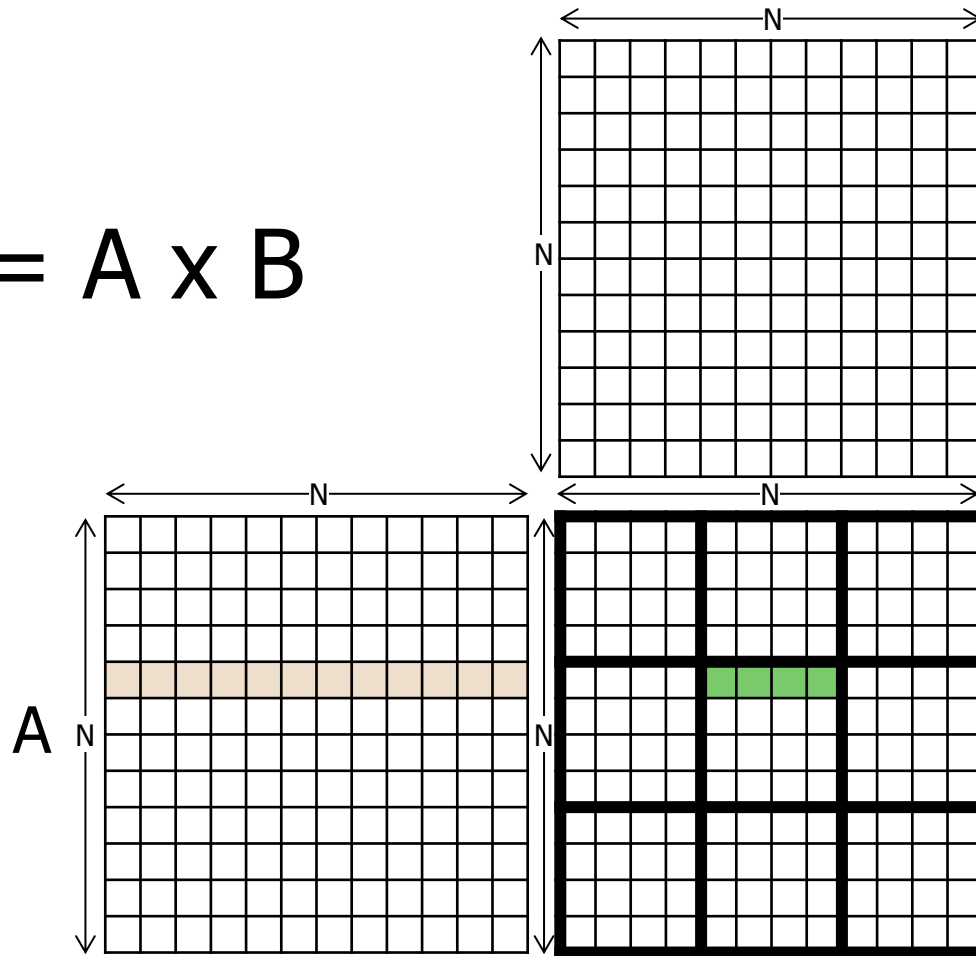
Example: Matrix-Matrix Multiplication (III)

```
__global__ void mm_kernel(float* A, float* B, float* C, unsigned int N) {  
  
    unsigned int row = blockIdx.y*blockDim.y + threadIdx.y;  
    unsigned int col = blockIdx.x*blockDim.x + threadIdx.x;  
  
    float sum = 0.0f;  
    for(unsigned int i = 0; i < N; ++i) {  
        sum += A[row*N + i]*B[i*N + col];  
    }  
    C[row*N + col] = sum;  
  
}
```



Reuse in Matrix-Matrix Multiplication (I)

$$C = A \times B$$



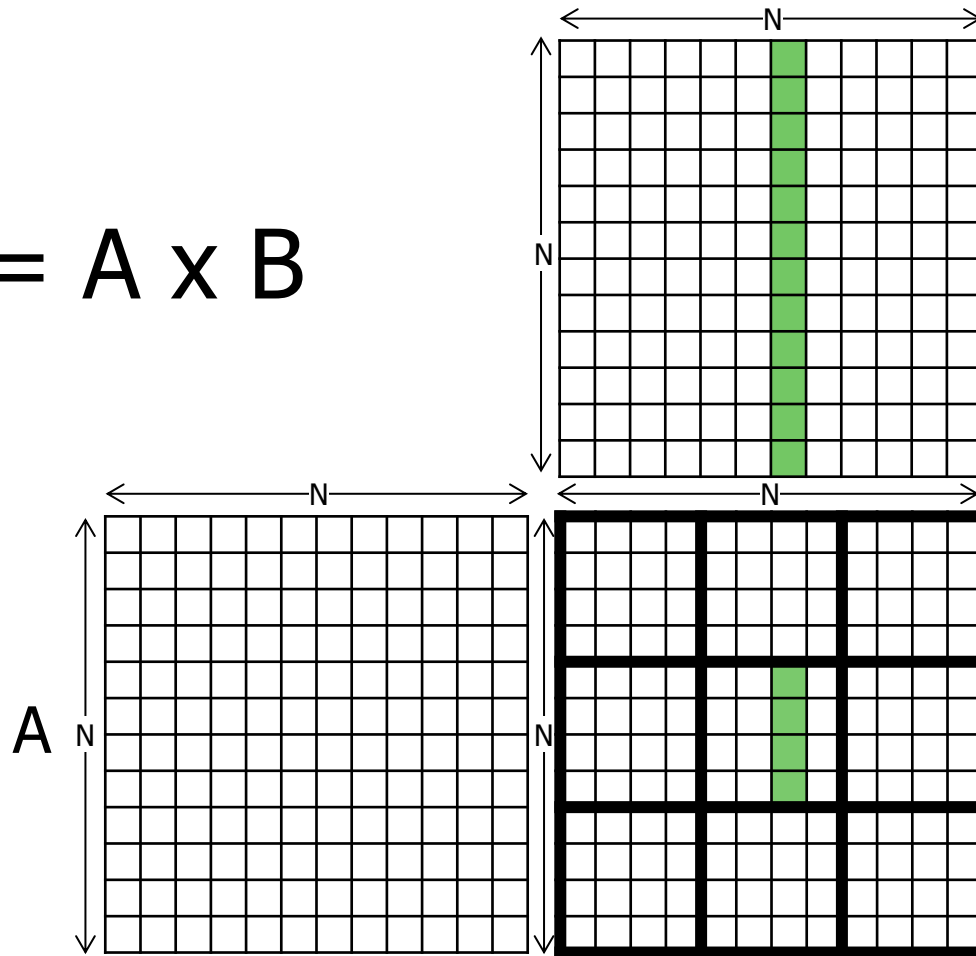
B

Some of the threads in the same thread block use the same input data

C

Reuse in Matrix-Matrix Multiplication (II)

$$C = A \times B$$



B

Some of the threads in the same thread block use the same input data

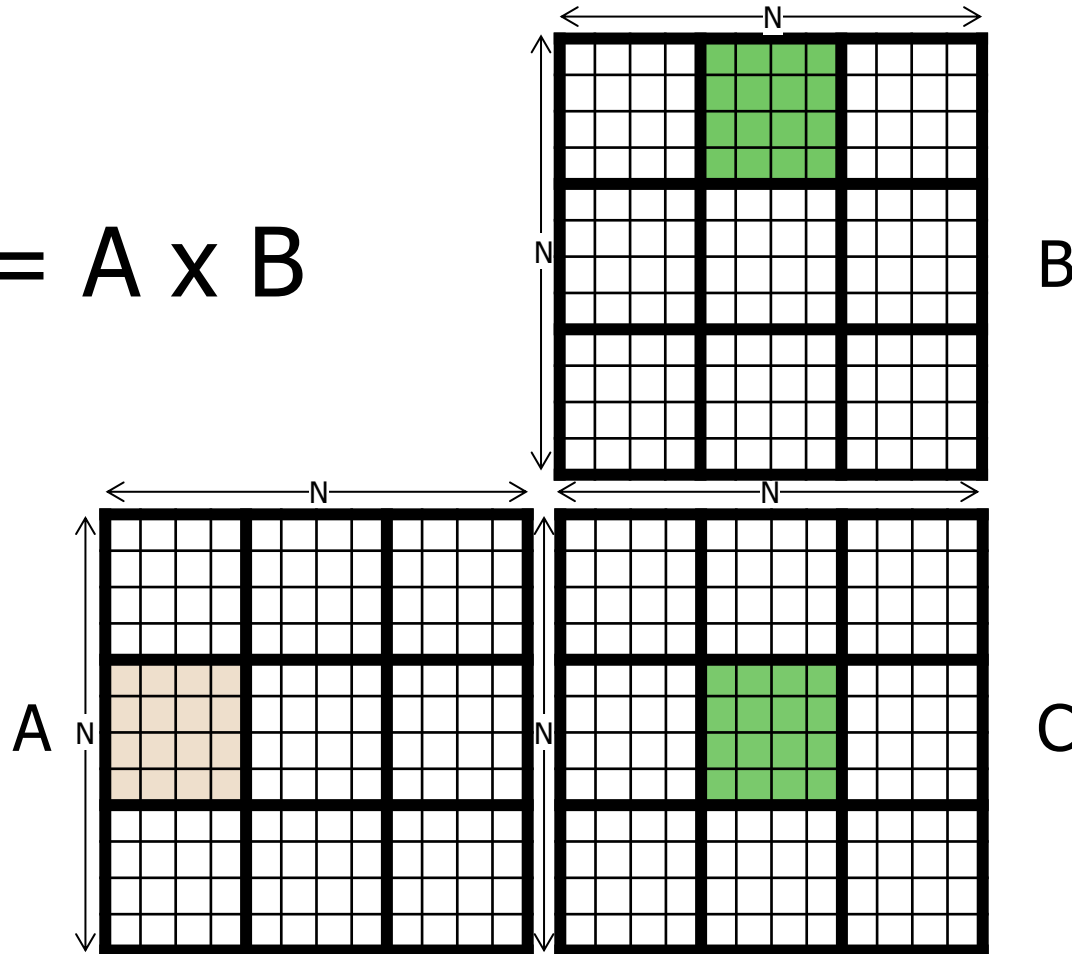
C

Reuse in Matrix-Matrix Multiplication (III)

- Sometimes, we are lucky:
 - The thread finds the data in the L1 cache because it was recently loaded by another thread
- Sometimes, we are not lucky:
 - The data gets evicted from the L1 cache before another thread tries to load it
- Solution:
 - Let the threads work together to load part of the data and ensure that all threads that need it use it before loading more data
 - Use shared memory to ensure data stays close
 - Optimizing called tiling because divides input to tiles

Tiled Matrix-Matrix Multiplication (I)

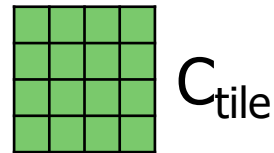
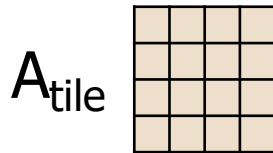
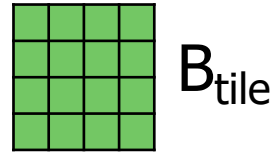
$$C = A \times B$$



Step 1: Load the first tile of each input matrix to shared memory (each thread loads one element)

Tiled Matrix-Matrix Multiplication (II)

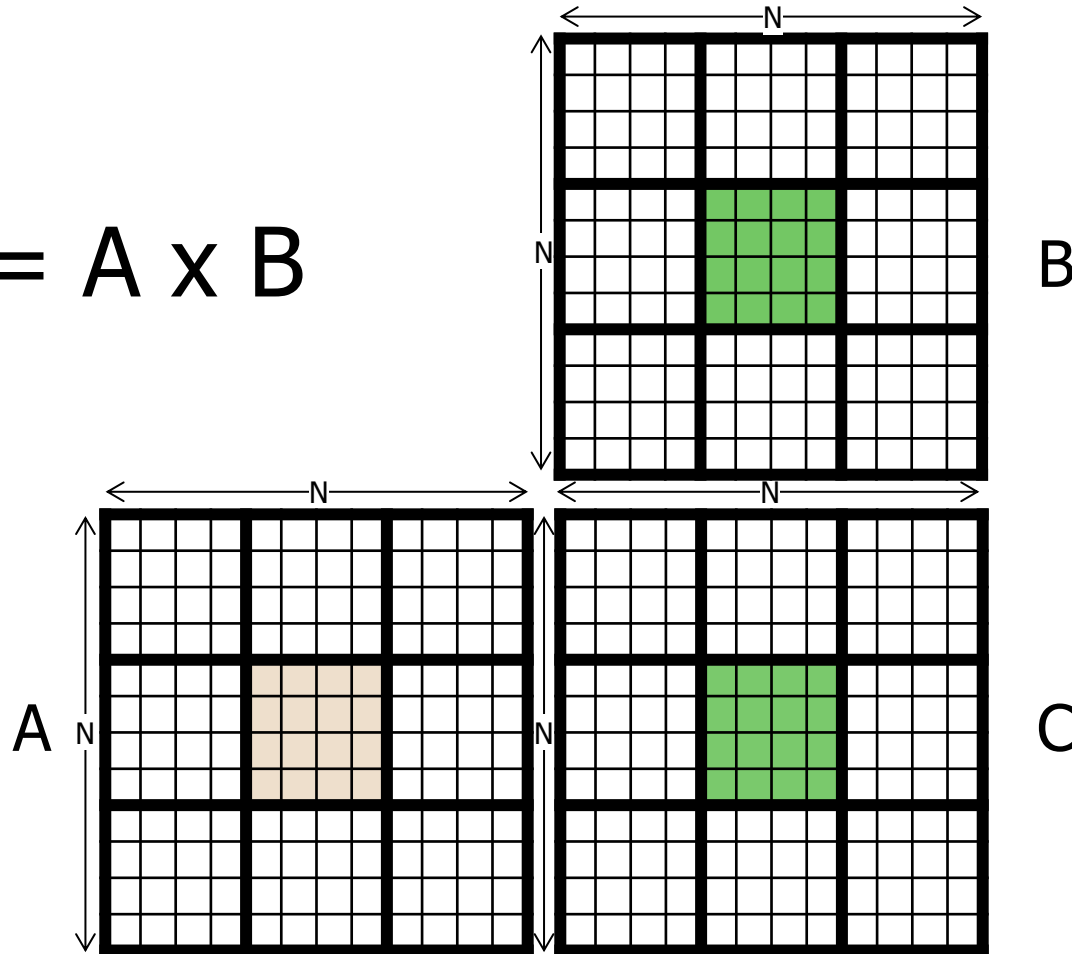
$$C_{\text{tile}} = A_{\text{tile}} \times B_{\text{tile}}$$



Step 2: Each thread computes its partial sum from the tiles in shared memory (threads wait for each other to finish)

Tiled Matrix-Matrix Multiplication (III)

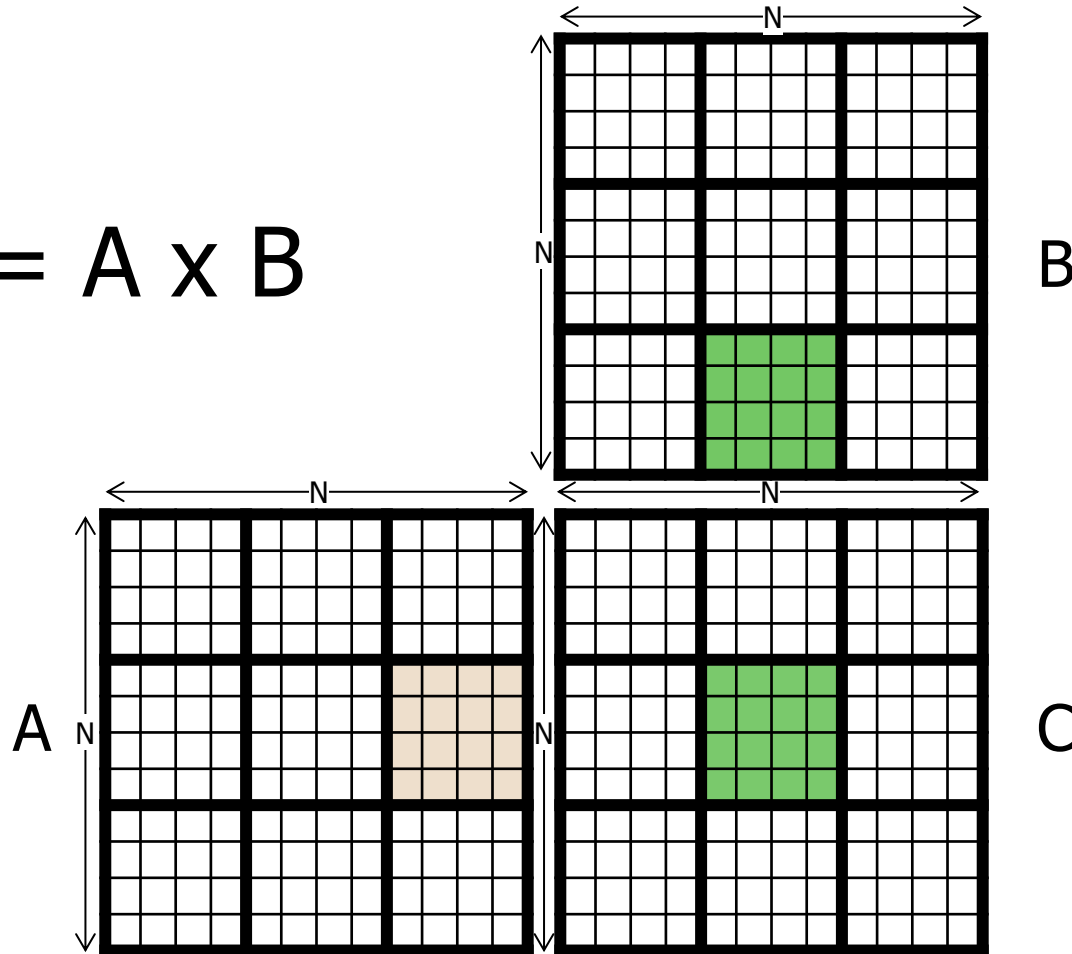
$$C = A \times B$$



...repeat for
the next tile

Tiled Matrix-Matrix Multiplication (IV)

$$C = A \times B$$



...and the next tile

Tiled Matrix-Matrix Multiplication (V)

```
__shared__ float A_s[TILE_DIM][TILE_DIM];  
__shared__ float B_s[TILE_DIM][TILE_DIM];
```

————— Declare arrays in shared memory

```
unsigned int row = blockIdx.y*blockDim.y + threadIdx.y;  
unsigned int col = blockIdx.x*blockDim.x + threadIdx.x;
```

```
float sum = 0.0f;
```

```
for(unsigned int tile = 0; tile < N/TILE_DIM; ++tile) {
```

```
    // Load tile to shared memory
```

```
    A_s[threadIdx.y][threadIdx.x] = A[row*N + tile*TILE_DIM + threadIdx.x];  
    B_s[threadIdx.y][threadIdx.x] = B[(tile*TILE_DIM + threadIdx.y)*N + col];  
    __syncthreads();
```

————— Threads wait for each other to finish loading before computing

```
    // Compute with tile
```

```
    for(unsigned int i = 0; i < TILE_DIM; ++i) {  
        sum += A_s[threadIdx.y][i]*B_s[i][threadIdx.x];  
    }
```

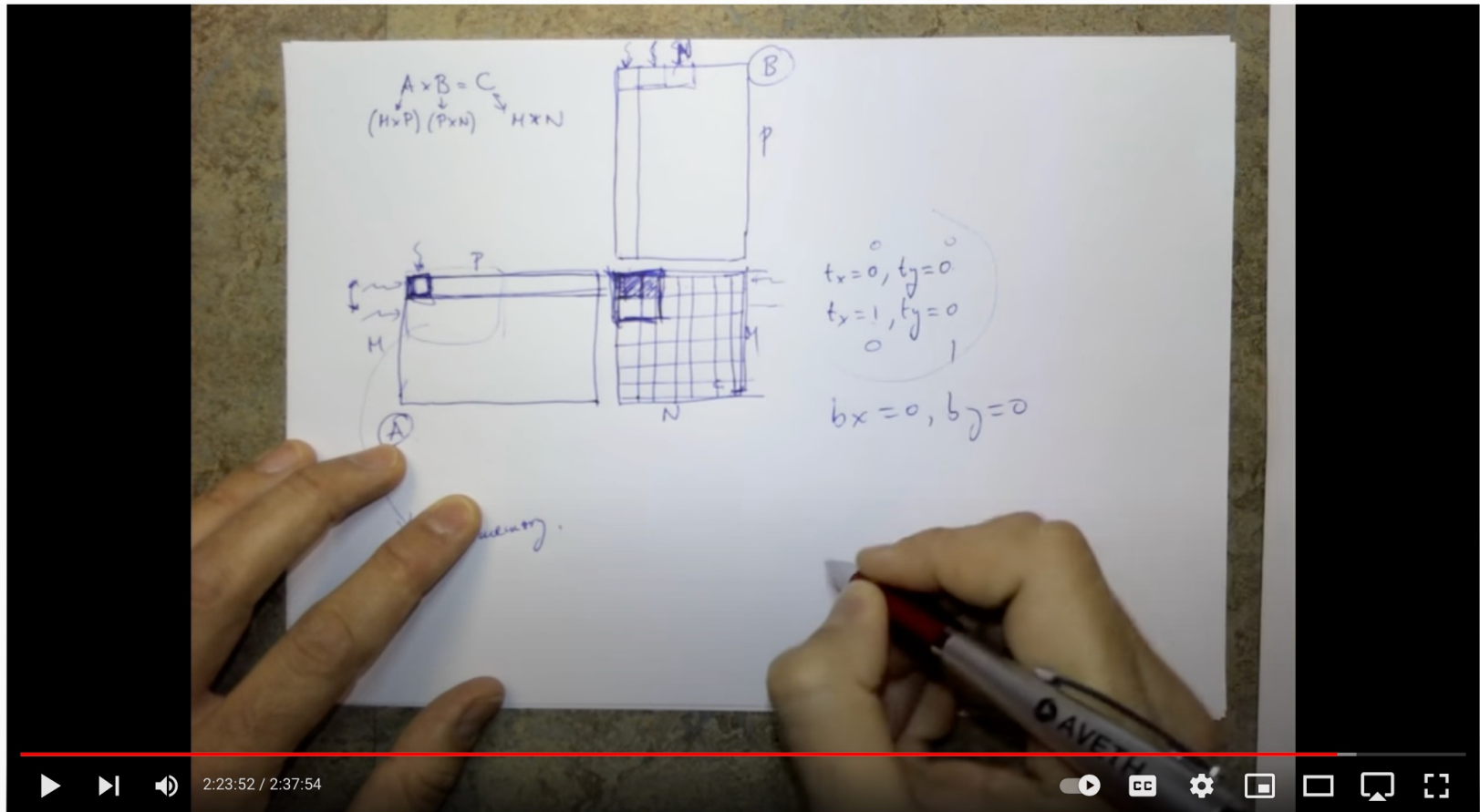
```
    __syncthreads();
```

————— Threads wait for each other to finish computing before loading

```
}
```

```
C[row*N + col] = sum;
```

Tiled Matrix Multiplication on GPU



Computer Architecture - Lecture 9: GPUs and GPGPU Programming (ETH Zürich, Fall 2017)

20,878 views • Oct 23, 2017

358 DISLIKE SHARE CLIP SAVE ...



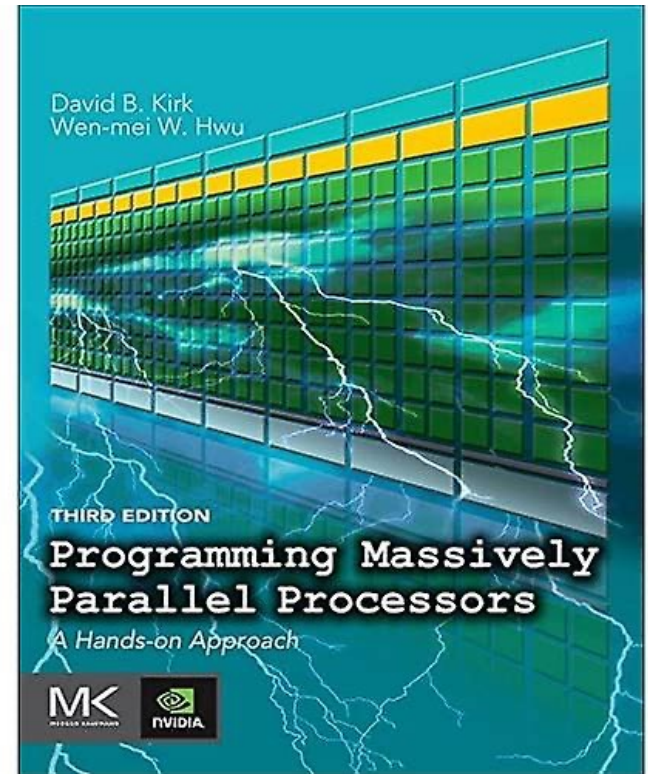
Onur Mutlu Lectures
23.7K subscribers

SUBSCRIBED



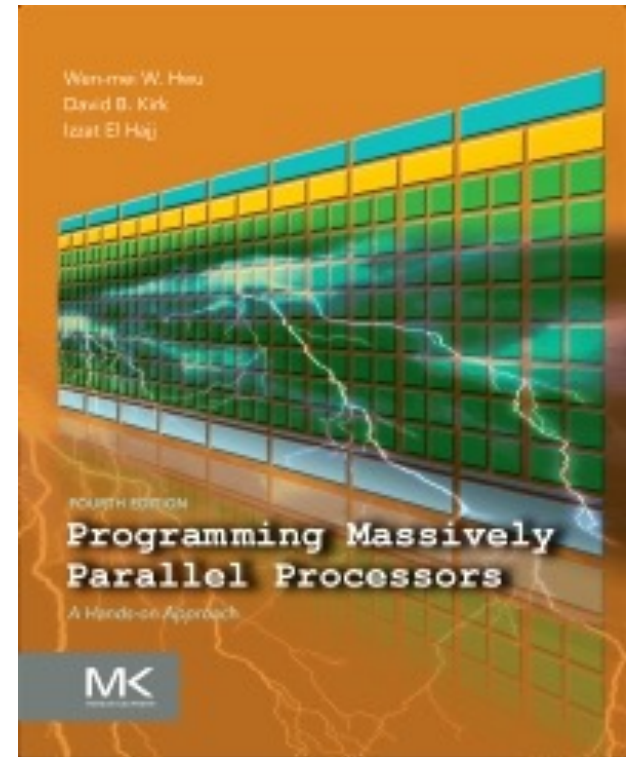
Recommended Readings (I)

- Hwu and Kirk, “**Programming Massively Parallel Processors,**”
Third Edition, 2017
 - Chapter 4: Memory and data locality



Recommended Readings (II)

- Hwu and Kirk and El Hajj, “[Programming Massively Parallel Processors](#),” Fourth Edition, 2022
 - Chapter 5 - Memory architecture and data locality



P&S Heterogeneous Systems

GPU Memory Hierarchy

Dr. Juan Gómez Luna

Prof. Onur Mutlu

ETH Zürich

Fall 2022

24 October 2022