P&S Heterogeneous Systems

GPU Memory Hierarchy

Dr. Juan Gómez Luna

Prof. Onur Mutlu

ETH Zürich

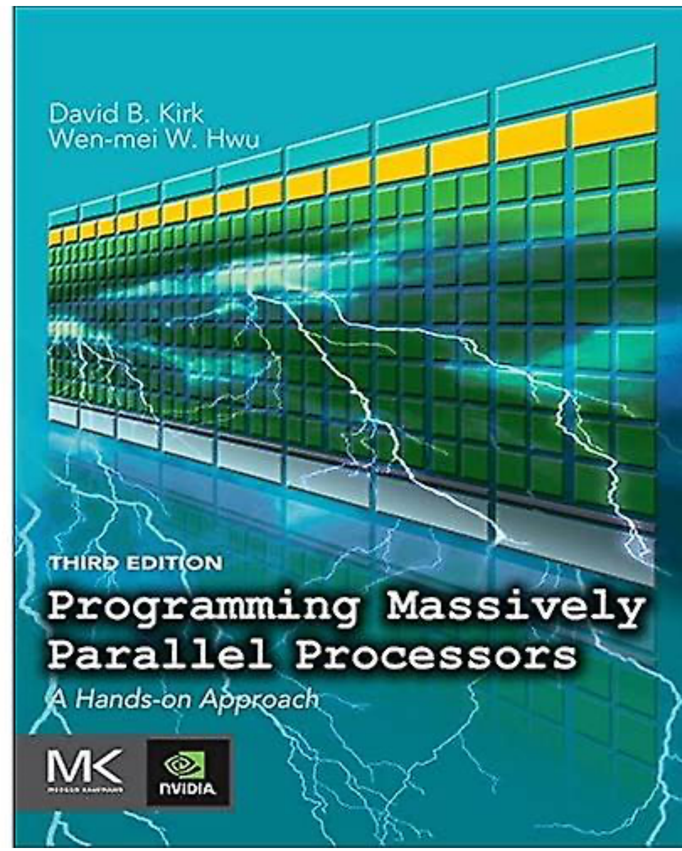
Fall 2021

28 October 2021

GPU Programming

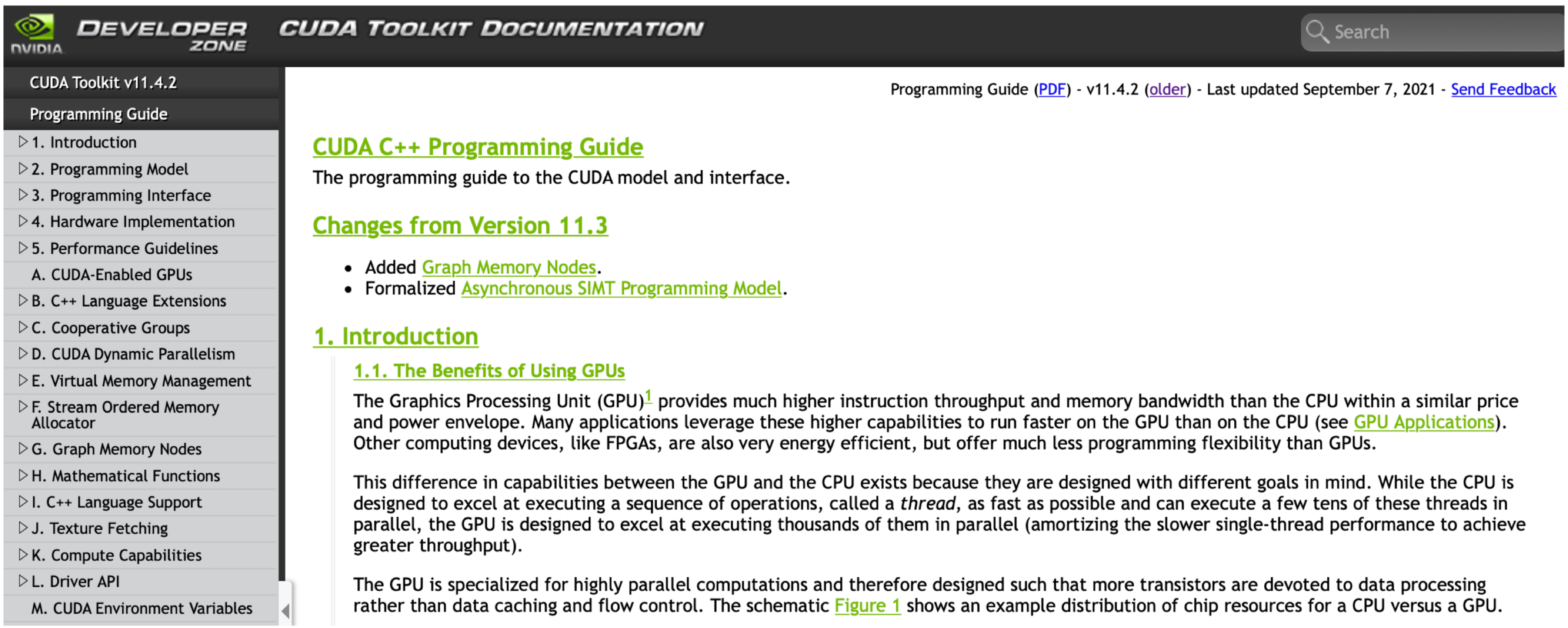
Recommended Readings (I)

- 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



DEVELOPER ZONE — CUDA TOOLKIT DOCUMENTATION

Search

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

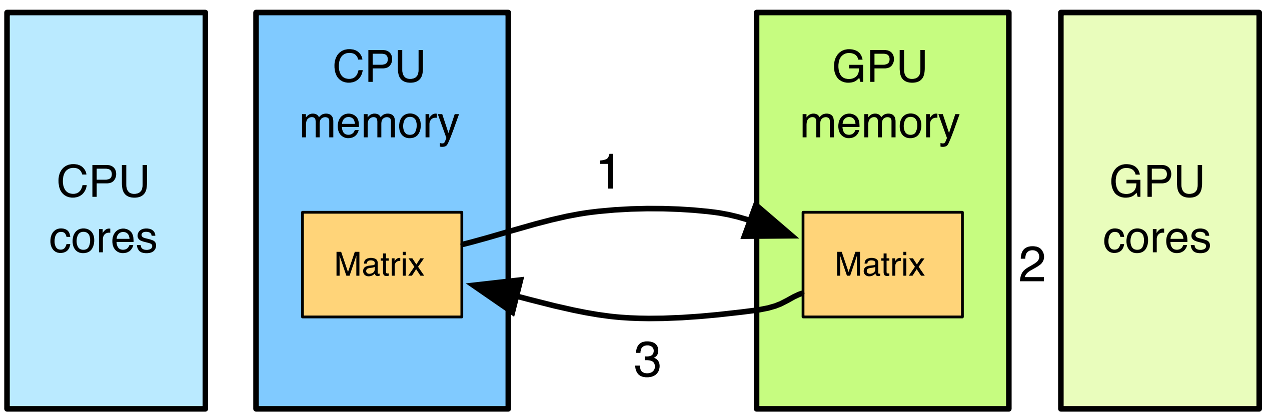
The Graphics Processing Unit (GPU)[1] provides much higher instruction throughput and memory bandwidth than the CPU within a similar price and power envelope. Many applications leverage these higher capabilities to run faster on the GPU than on the CPU (see GPU Applications). Other computing devices,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

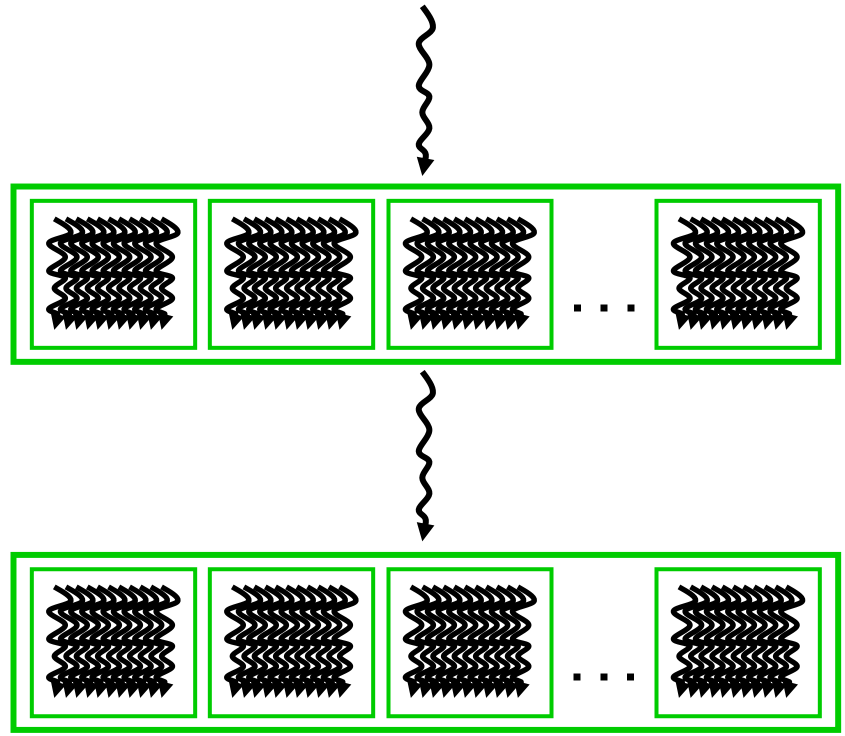
Parallel Kernel (device)

```
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();`

Slide credit: Hwu & Kirk

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

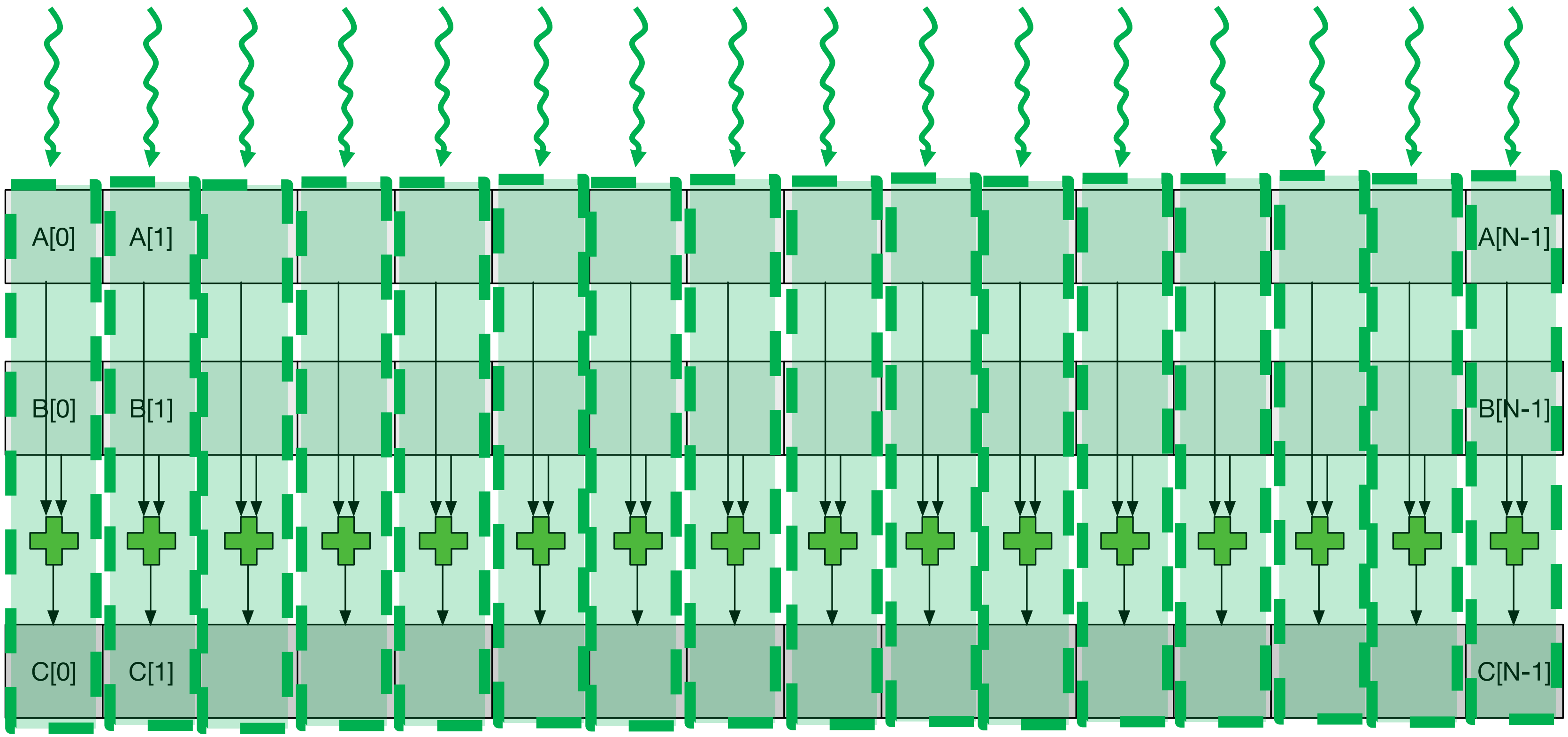
```
cudaFree(d_in);
```

- Explicit synchronization

```
cudaDeviceSynchronize();
```

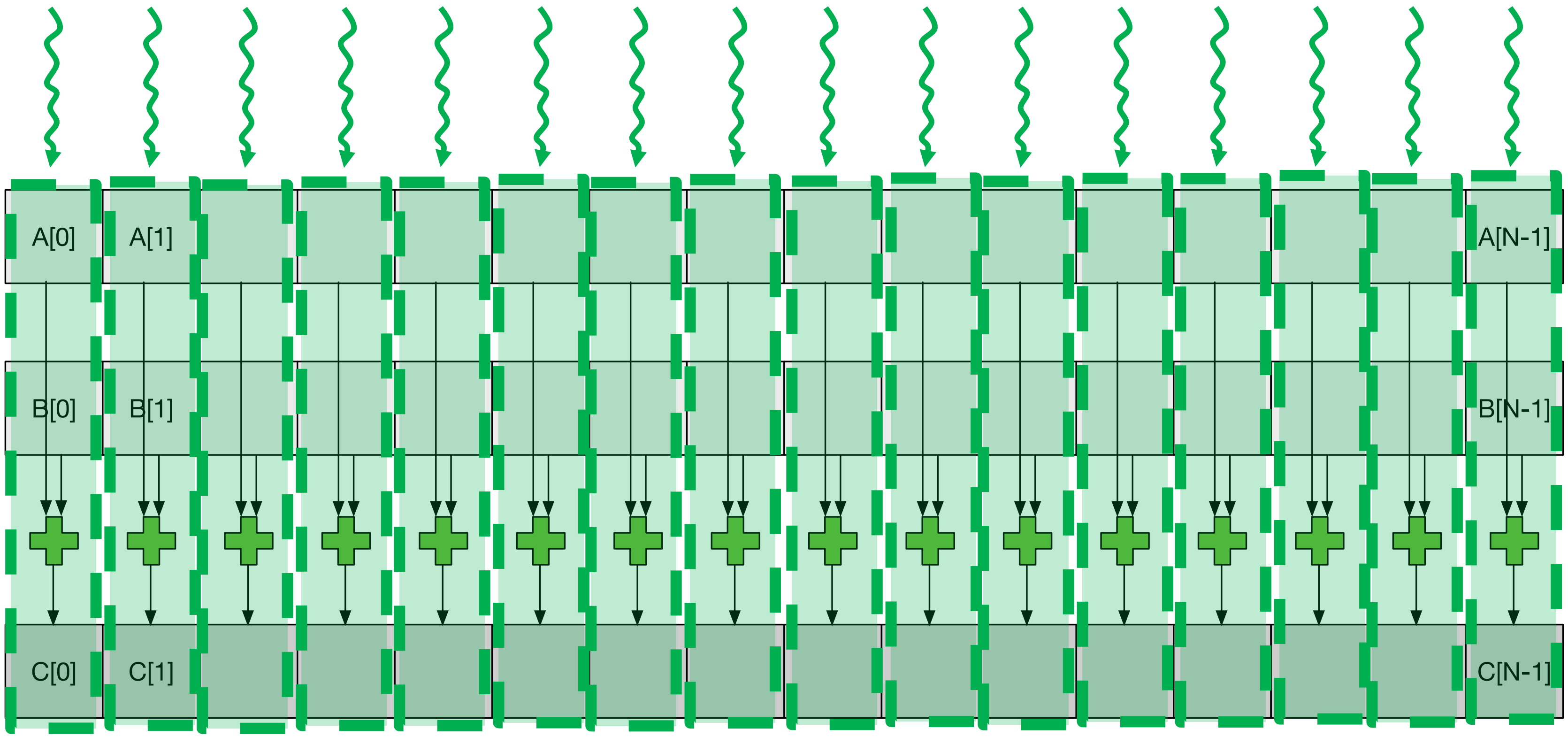
Vector Addition (I)

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



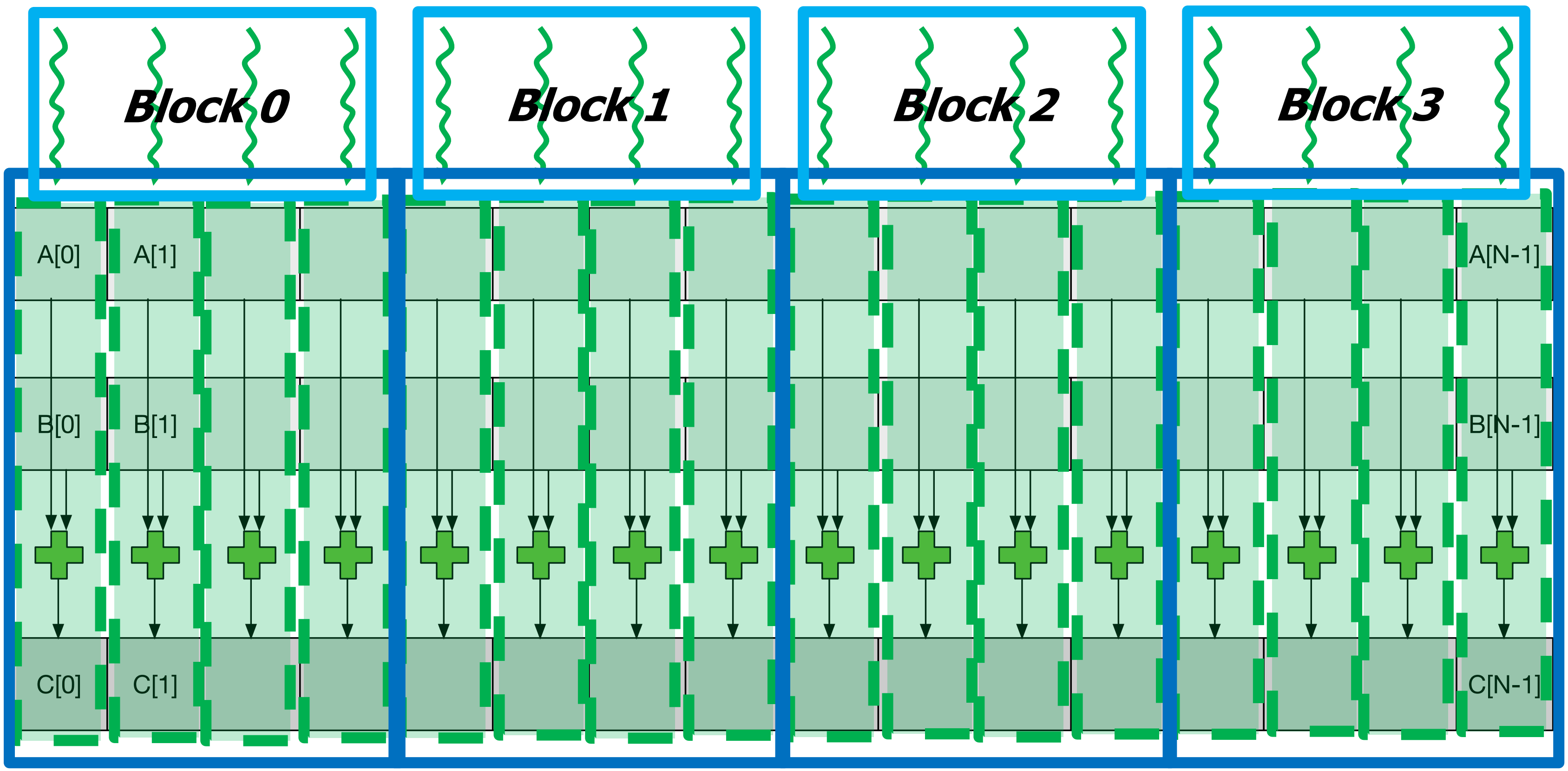
Vector Addition (II)

- 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);
}
```


Slide credit: Izzat El Hajj

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
  - Image[j][i], where 0 ≤ j < height, and 0 ≤ i < width

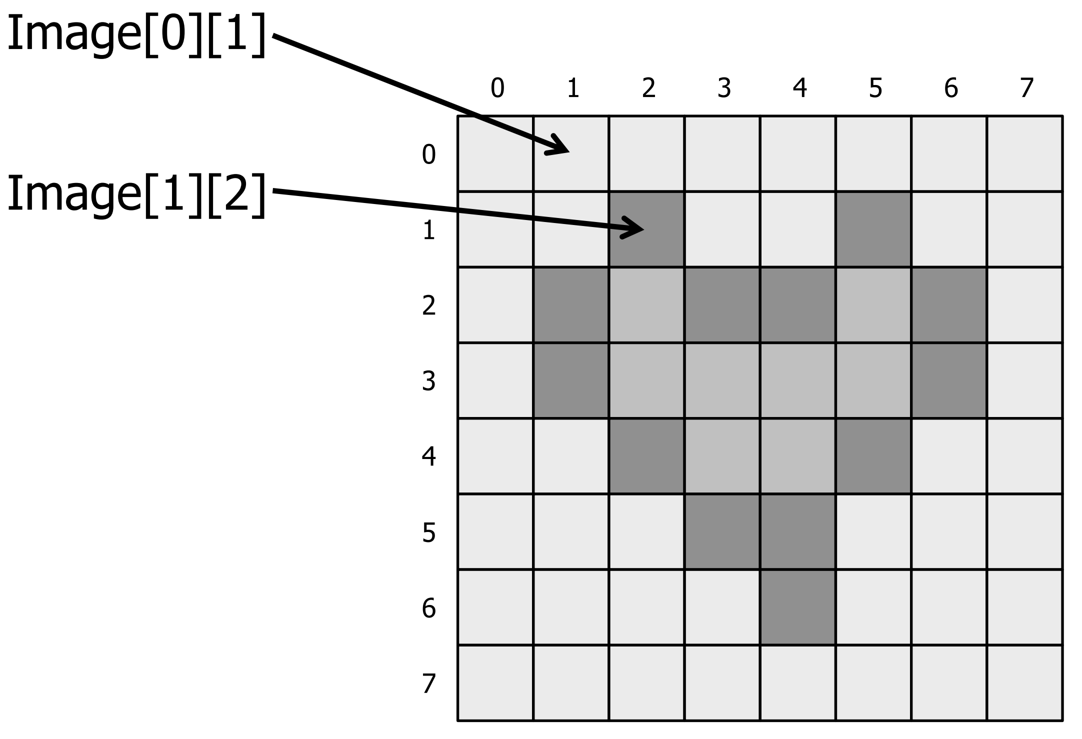
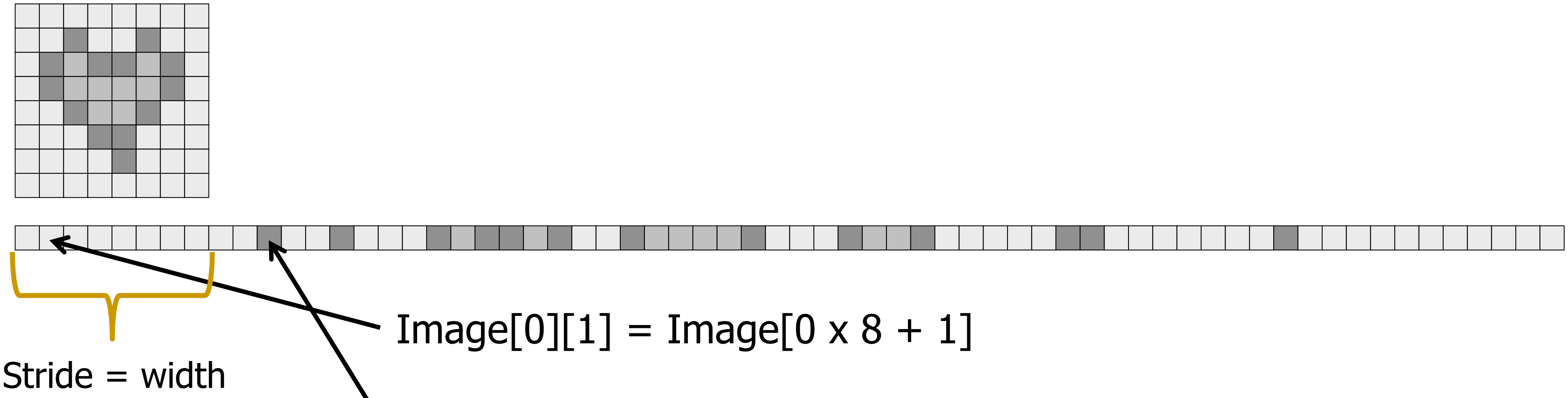


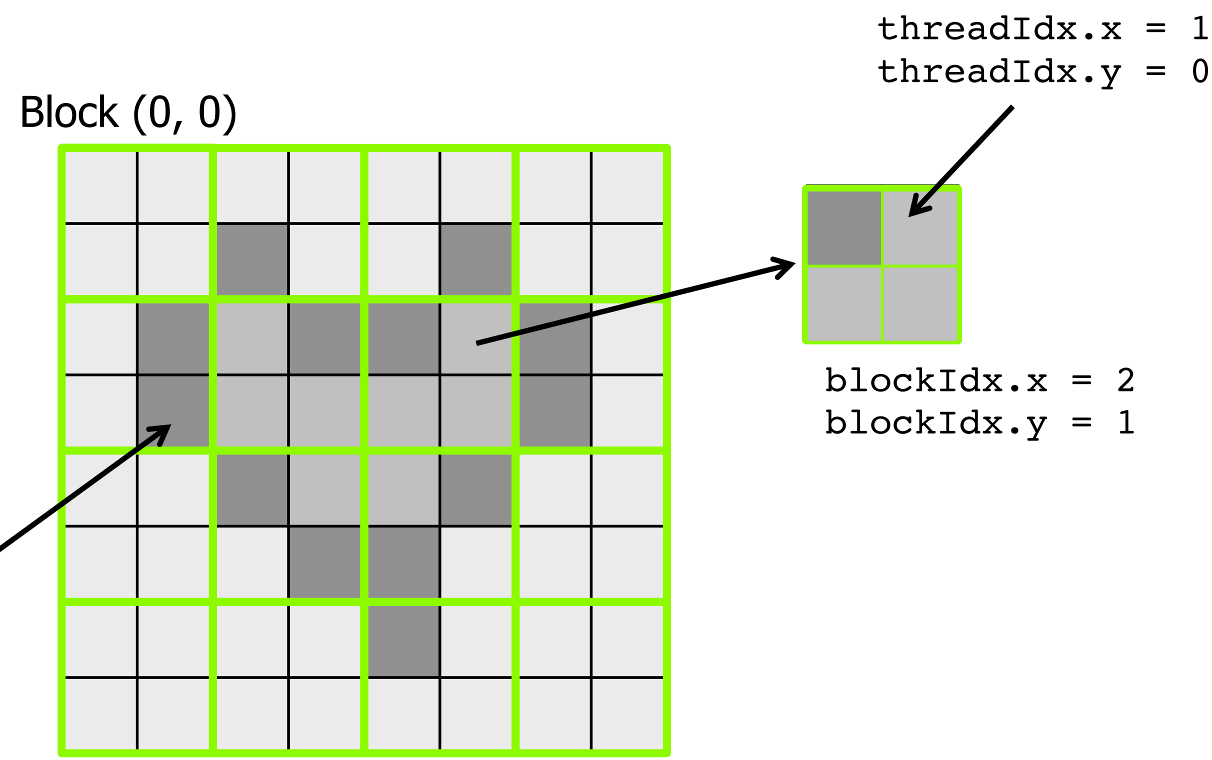
Image Layout in Memory

- Row-major layout
- Image[j][i] = Image[j x width + i]



Indexing and Memory Access: 2D Grid

- 2D blocks
  - `gridDim.x, gridDim.y`




```
Row = blockIdx.y *
blockDim.y + threadIdx.y

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

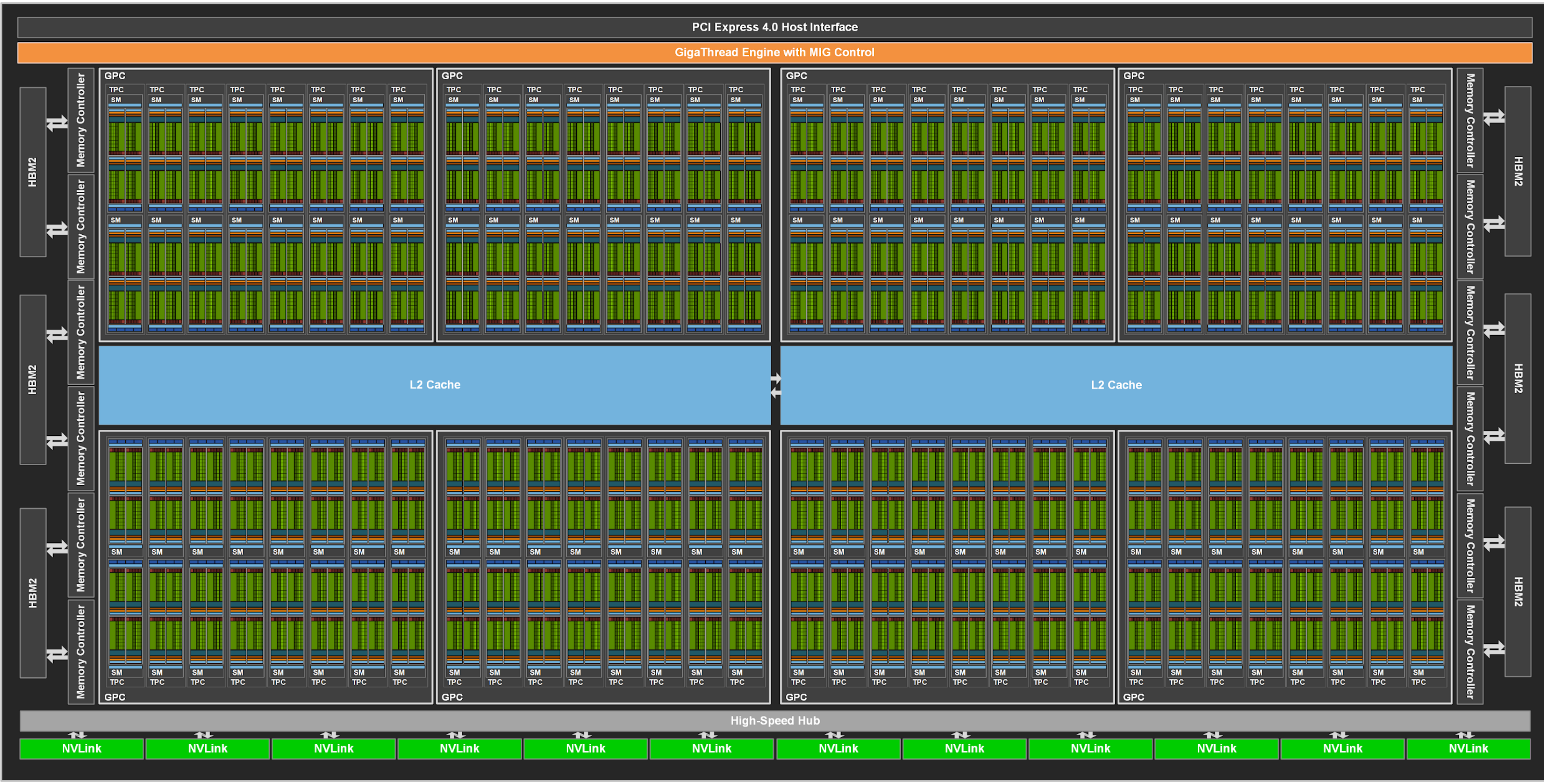
Row = 1 * 2 + 1 = 3

Col = 0 * 2 + 1 = 1

Image[3][1] = Image[3 * 8 + 1]

GPU Memories

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)

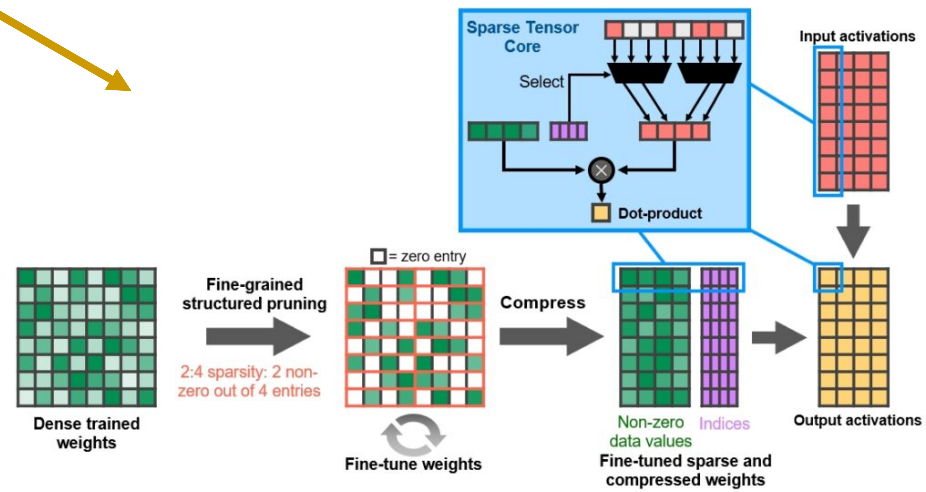
40MB L2 cache

NVIDIA A100 Core

19.5 TFLOPS Single Precision

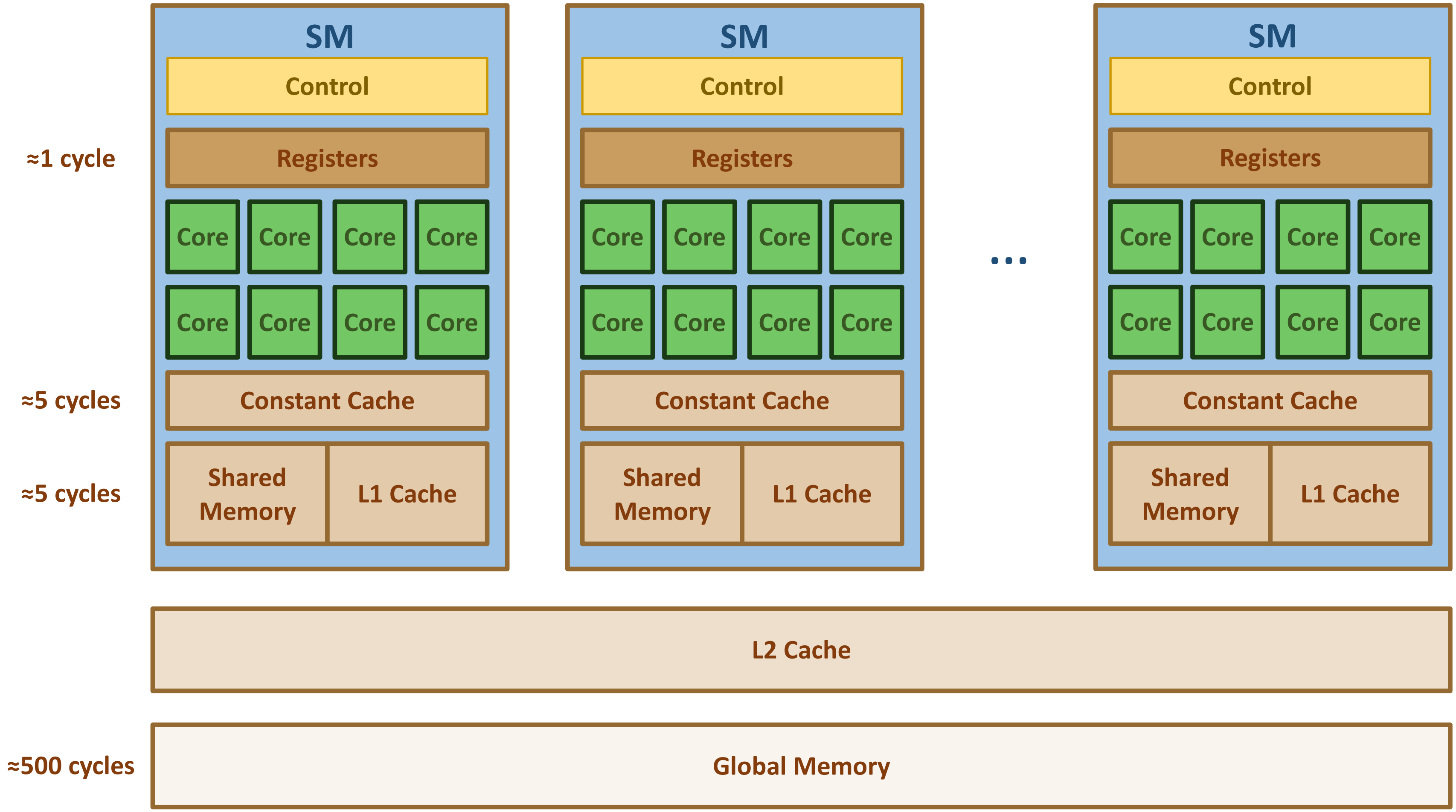
9.7 TFLOPS Double Precision

312 TFLOPS for Deep Learning (Tensor cores)



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

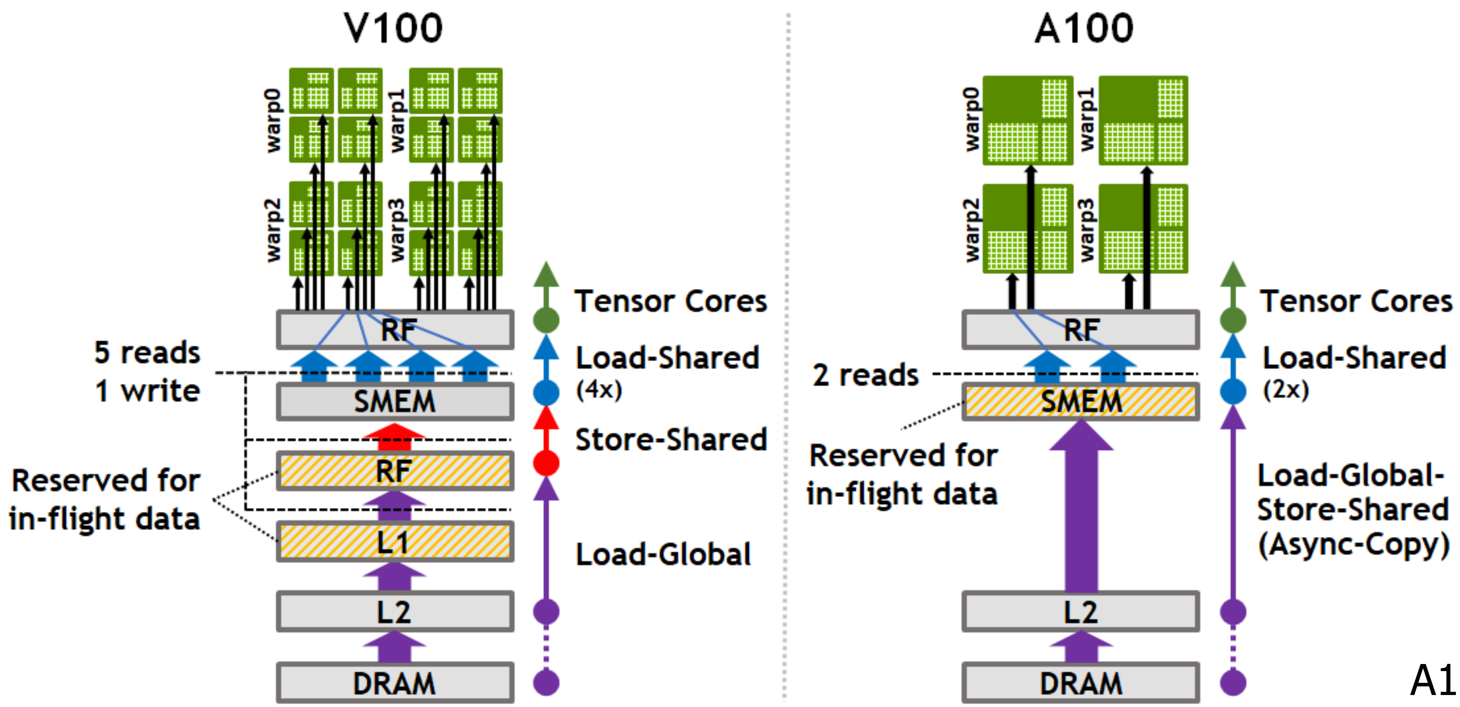
Memory in the GPU Architecture



Slide credit: Izzat El Hajj

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.

https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

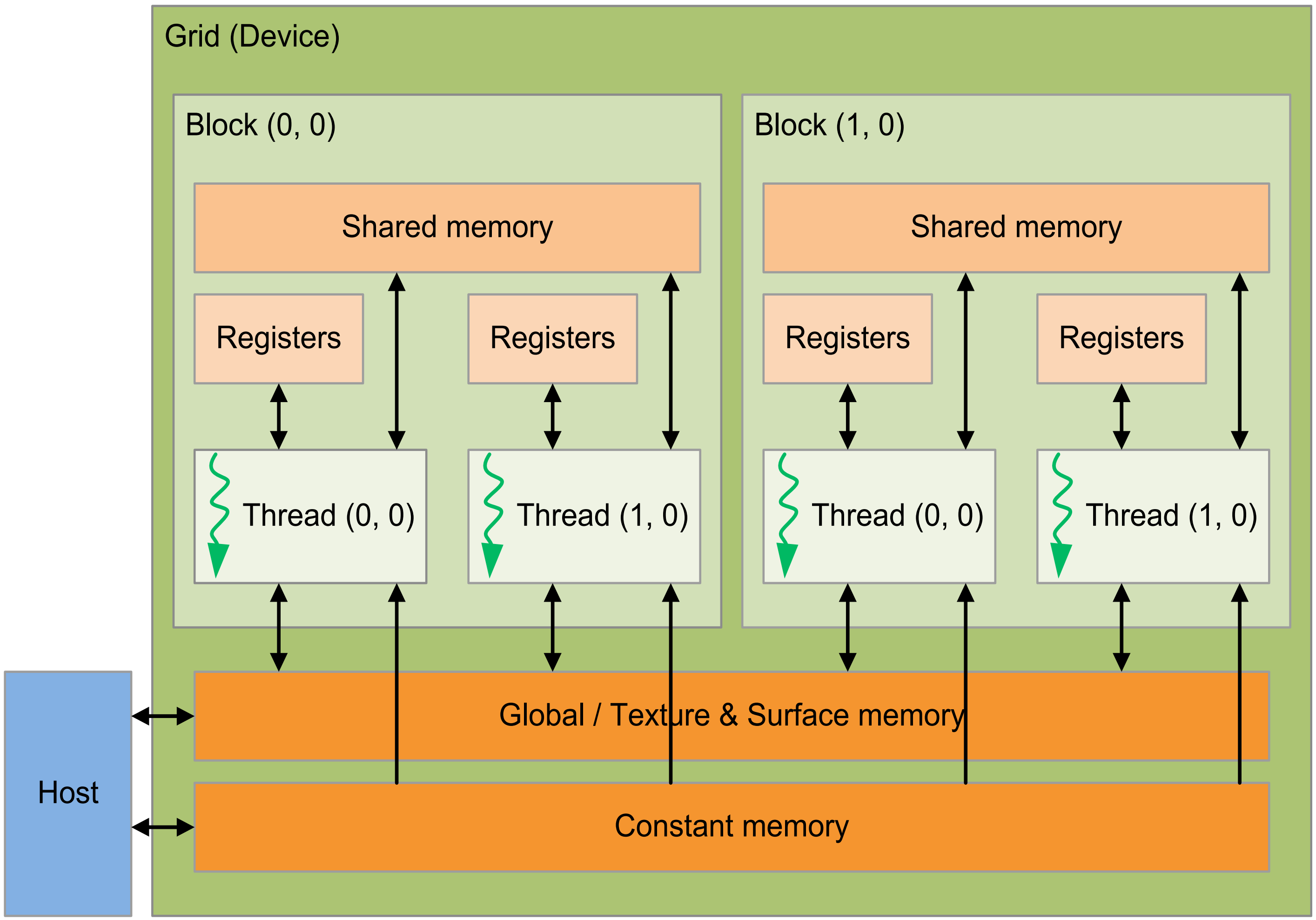
CUDA Variable Type Qualifiers

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

- `__device__` is optional when used with `__shared__`, or `__constant__`

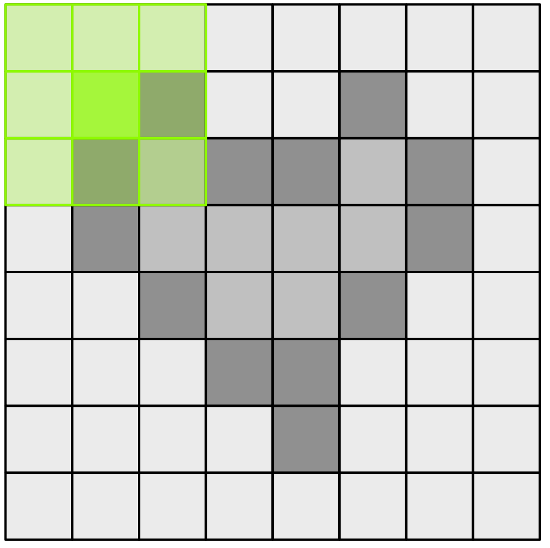
- Recall `cudaMalloc(…)` allocates memory from the host
  - Constant memory can also be allocated and initialized from the host
- Automatic variables without any qualifier reside in a register
  - Except arrays that reside in global memory

Memory Hierarchy in CUDA Programs



Data Reuse

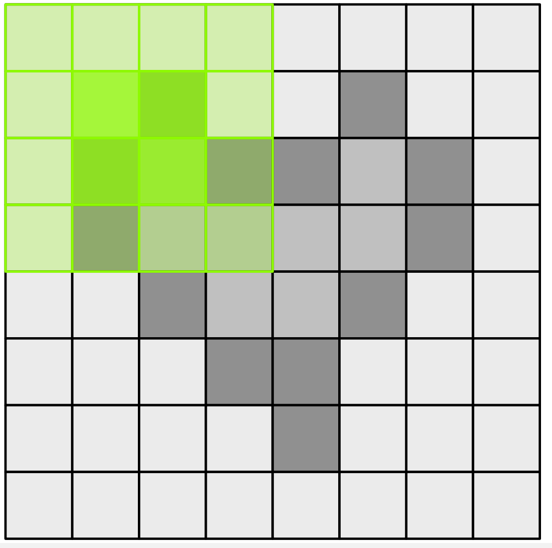
- Same memory locations accessed by neighboring threads



```
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

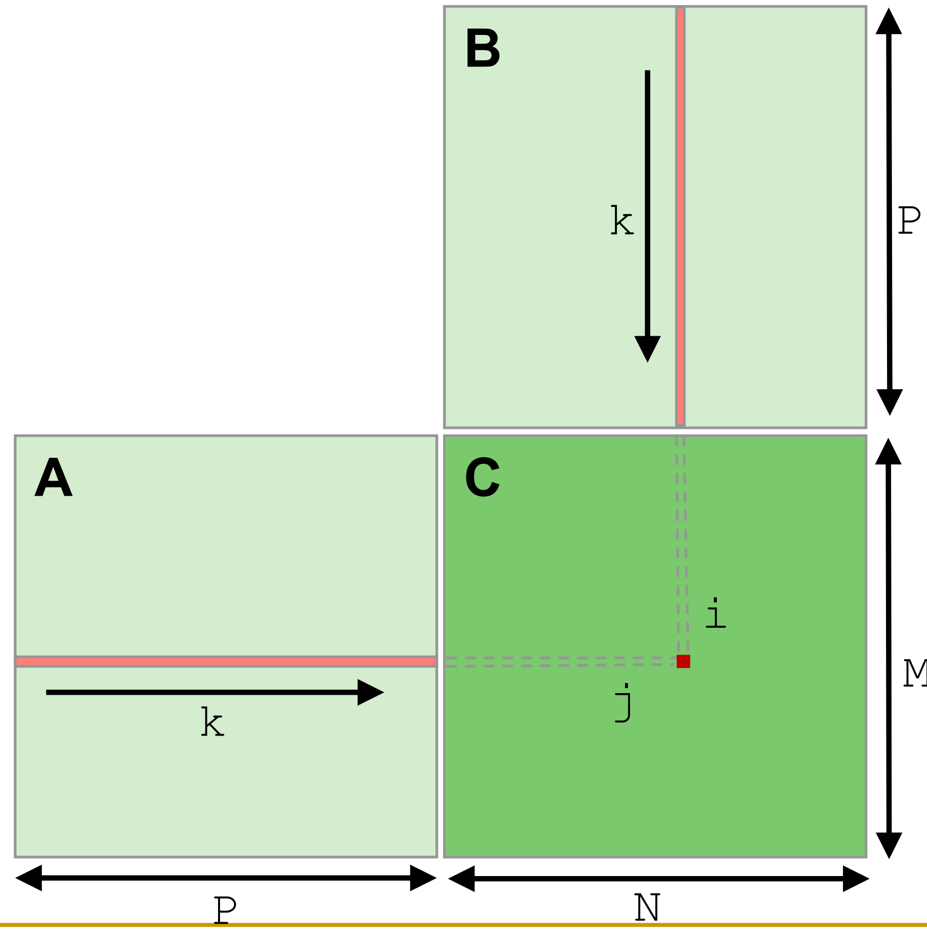
- **`void __syncthreads();`**
- Synchronizes all threads in a block
- Once all threads in a block have reached this point, execution resumes normally
- Used to avoid RAW / WAR / WAW hazards when accessing shared or global memory

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 x B
- Consider two input matrices A and B in row-major layout
  - A size is M x P
  - B size is P x N
  - C size is M x N

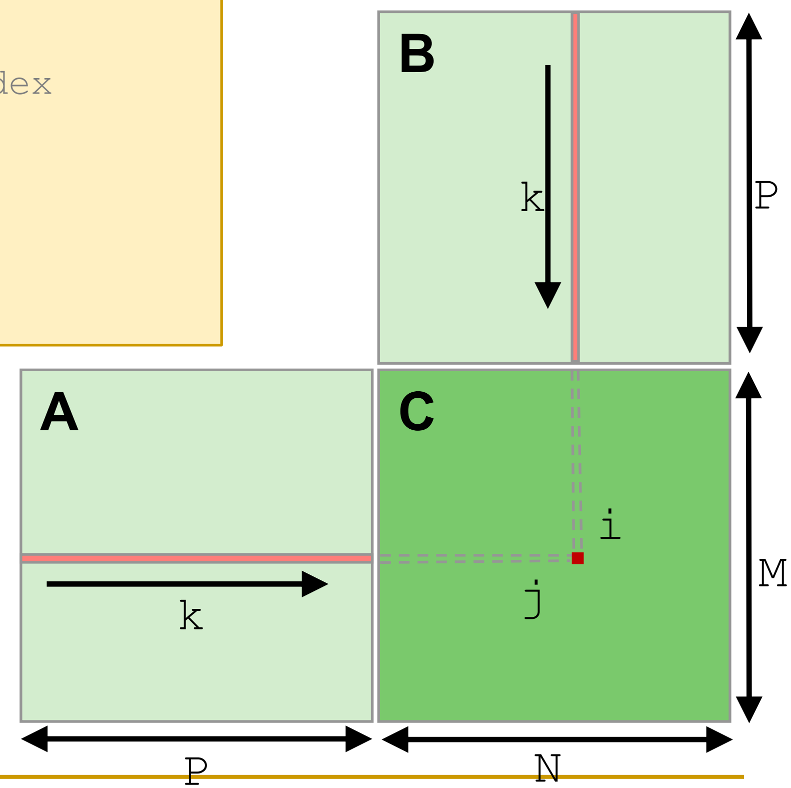


Naïve Matrix Multiplication (II)

- Naïve implementation of matrix multiplication has poor cache locality

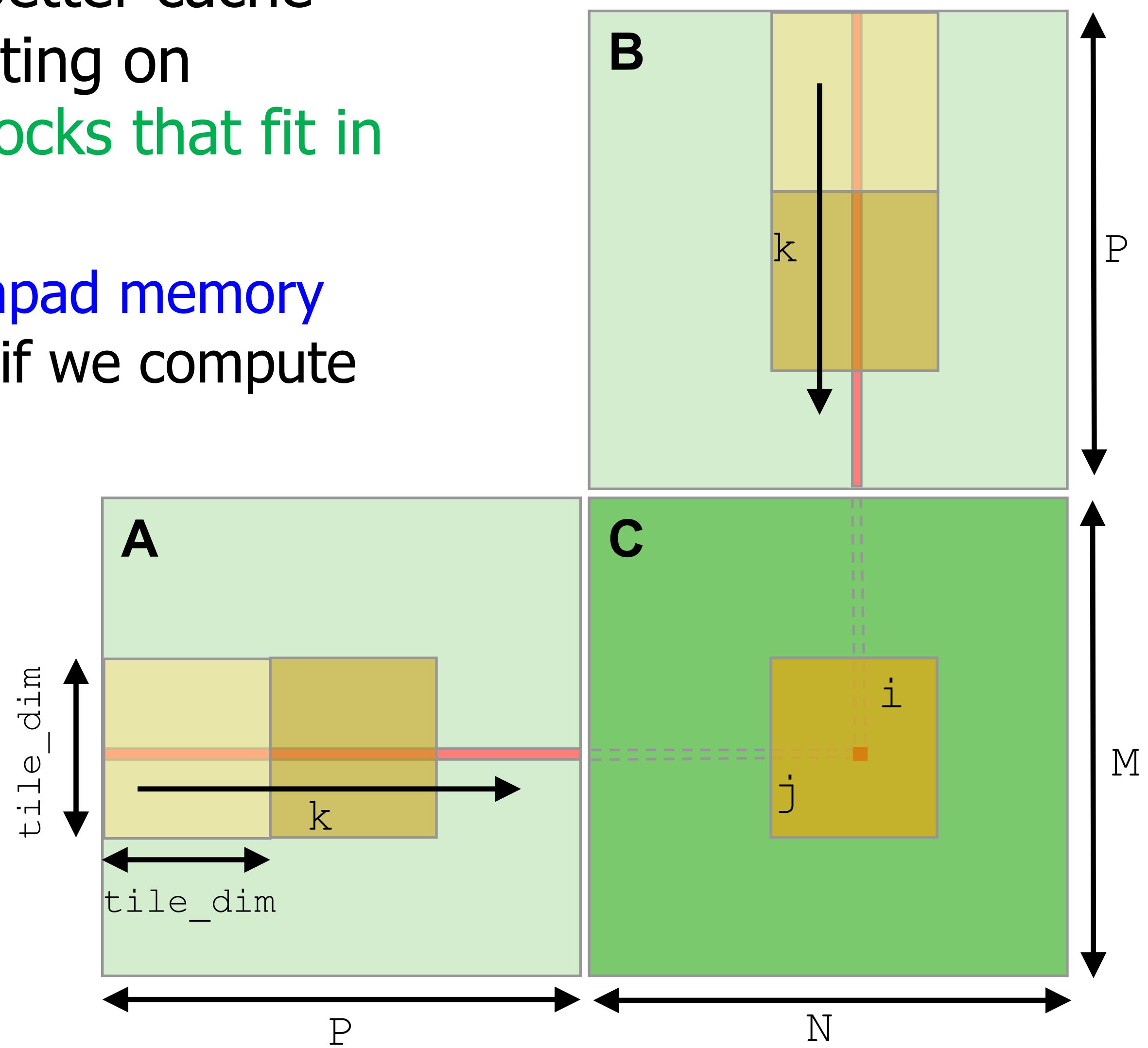
```
#define A(i,j) matrix_A[i * P + j]
#define B(i,j) matrix_B[i * N + j]
#define C(i,j) matrix_C[i * N + j]

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



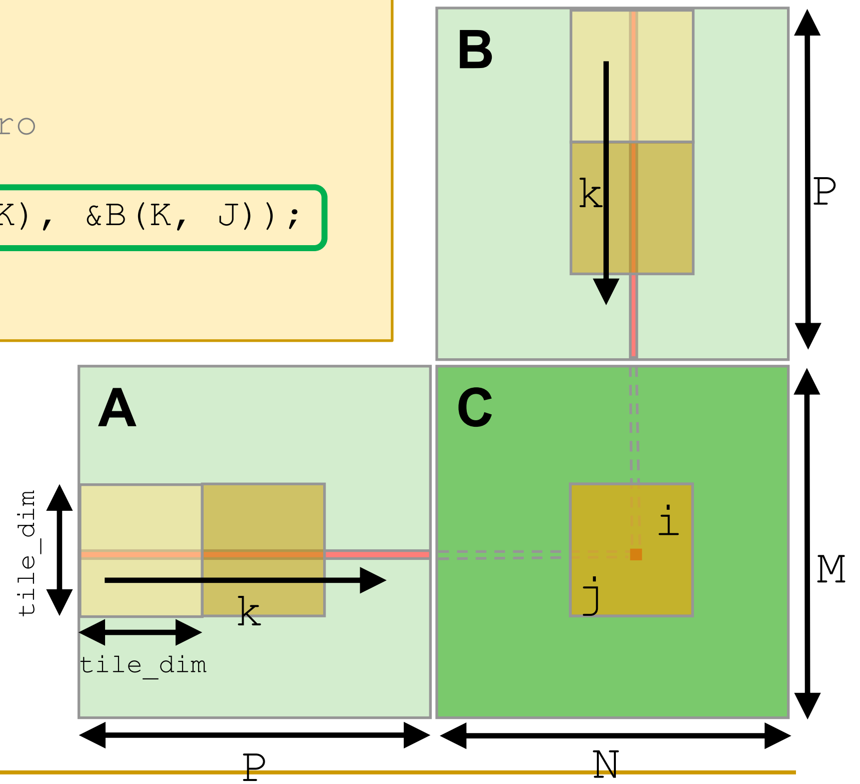
Lam+, "The cache performance and optimizations of blocked algorithms," ASPLOS 1991. https://doi.org/10.1145/106972.106981
Bansal+, "Chapter 15 - Fast Matrix Computations on Heterogeneous Streams," in "High Performance Parallelism Pearls", 2015. https://doi.org/10.1016/B978-0-12-803819-2.00011-2
Kirk & Hwu, "Chapter 5 - Performance considerations," in "Programming Massively Parallel Processors (Third Edition)", 2017. https://doi.org/10.1016/B978-0-12-811986-0.00005-4

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`



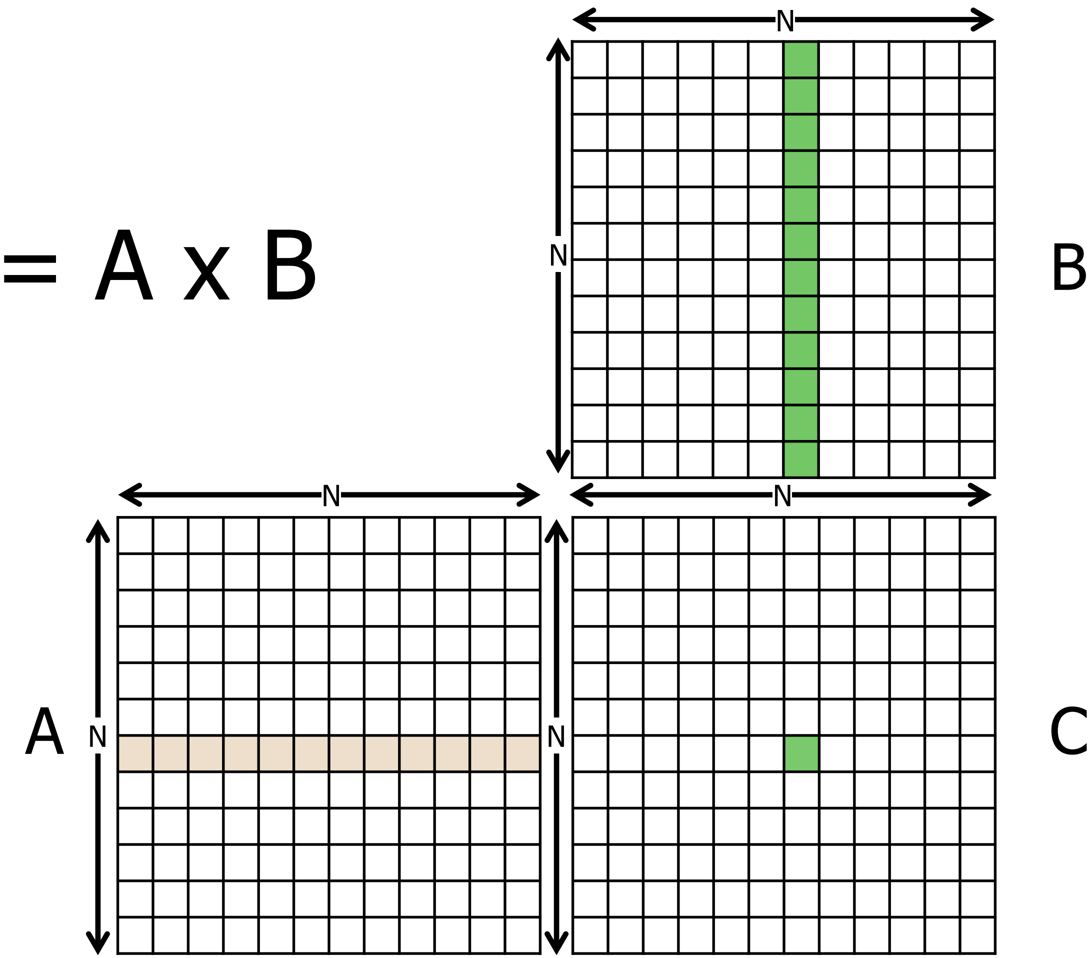
Lam+, "The cache performance and optimizations of blocked algorithms," ASPLOS 1991. https://doi.org/10.1145/106972.106981
Bansal+, "Chapter 15 - Fast Matrix Computations on Heterogeneous Streams," in "High Performance Parallelism Pearls", 2015. https://doi.org/10.1016/B978-0-12-803819-2.00011-2
Kirk & Hwu, "Chapter 5 - Performance considerations," in "Programming Massively Parallel Processors (Third Edition)", 2017. https://doi.org/10.1016/B978-0-12-811986-0.00005-4

Lecture on Advanced Caches

DDCA - Lecture 24: Advanced Caches (Spring 2021) https://youtu.be/89Q7OdhmQ9o

Example: Matrix-Matrix Multiplication (I)

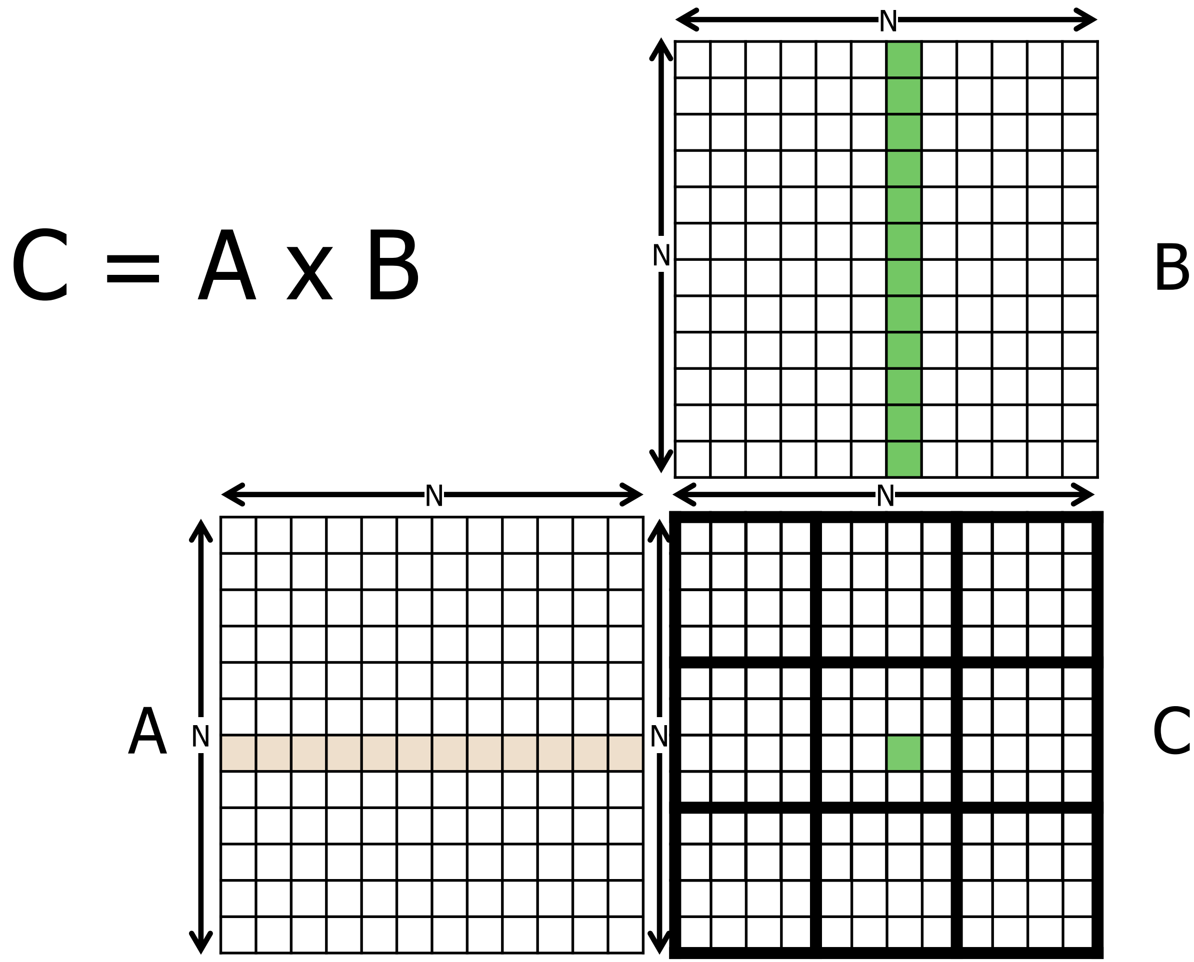
C = A x B



Slide credit: Izzat El Hajj

Example: Matrix-Matrix Multiplication (II)

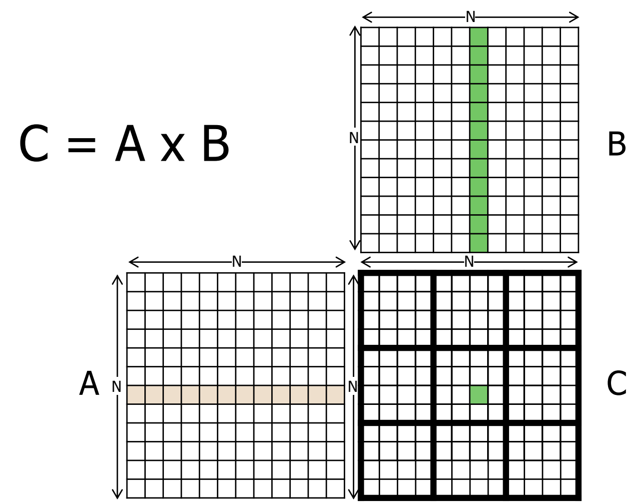
Parallelization approach: assign one thread to each element in the output matrix (C)



Slide credit: Izzat El Hajj

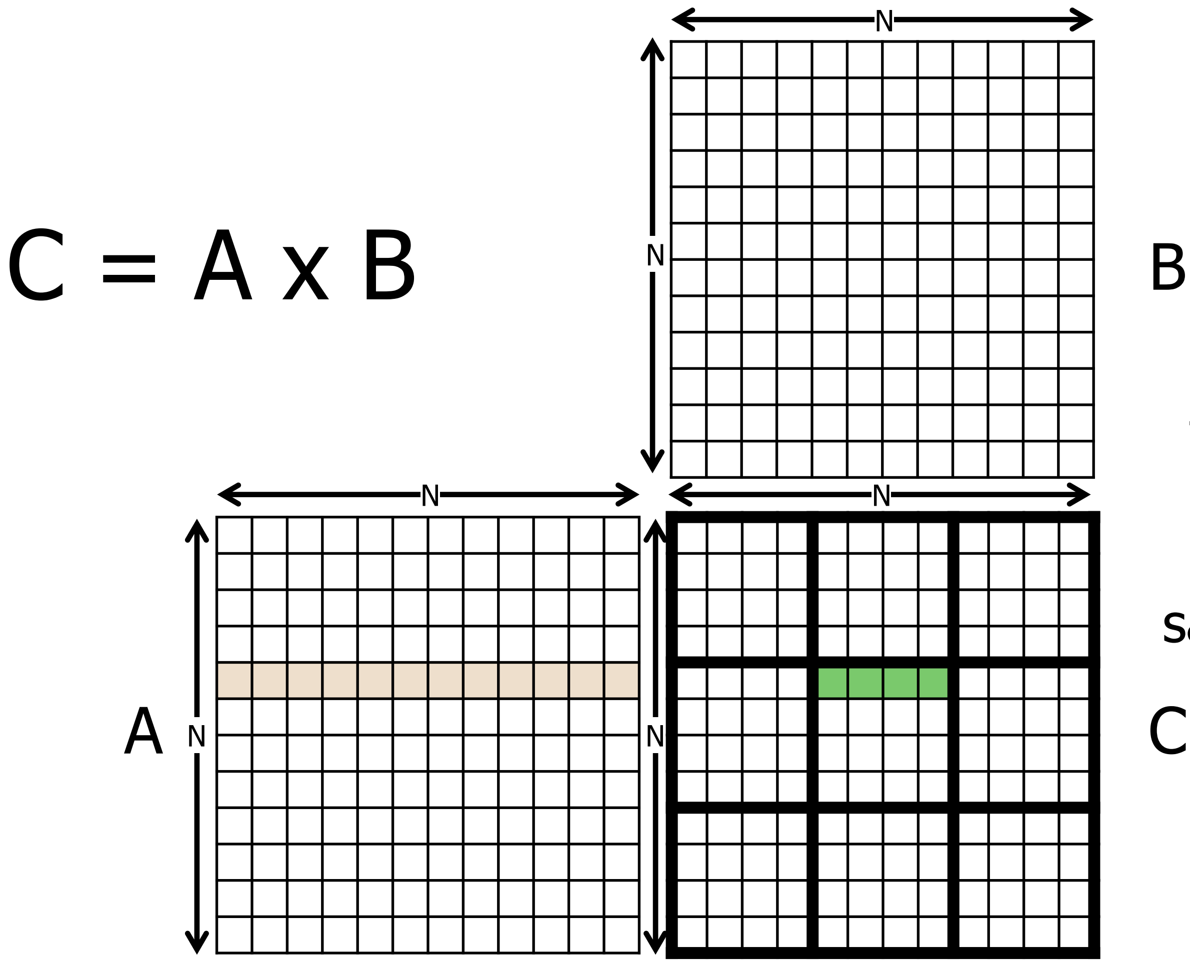
Example: Matrix-Matrix Multiplication (III)

```
__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;

}
```



Slide credit: Izzat El Hajj

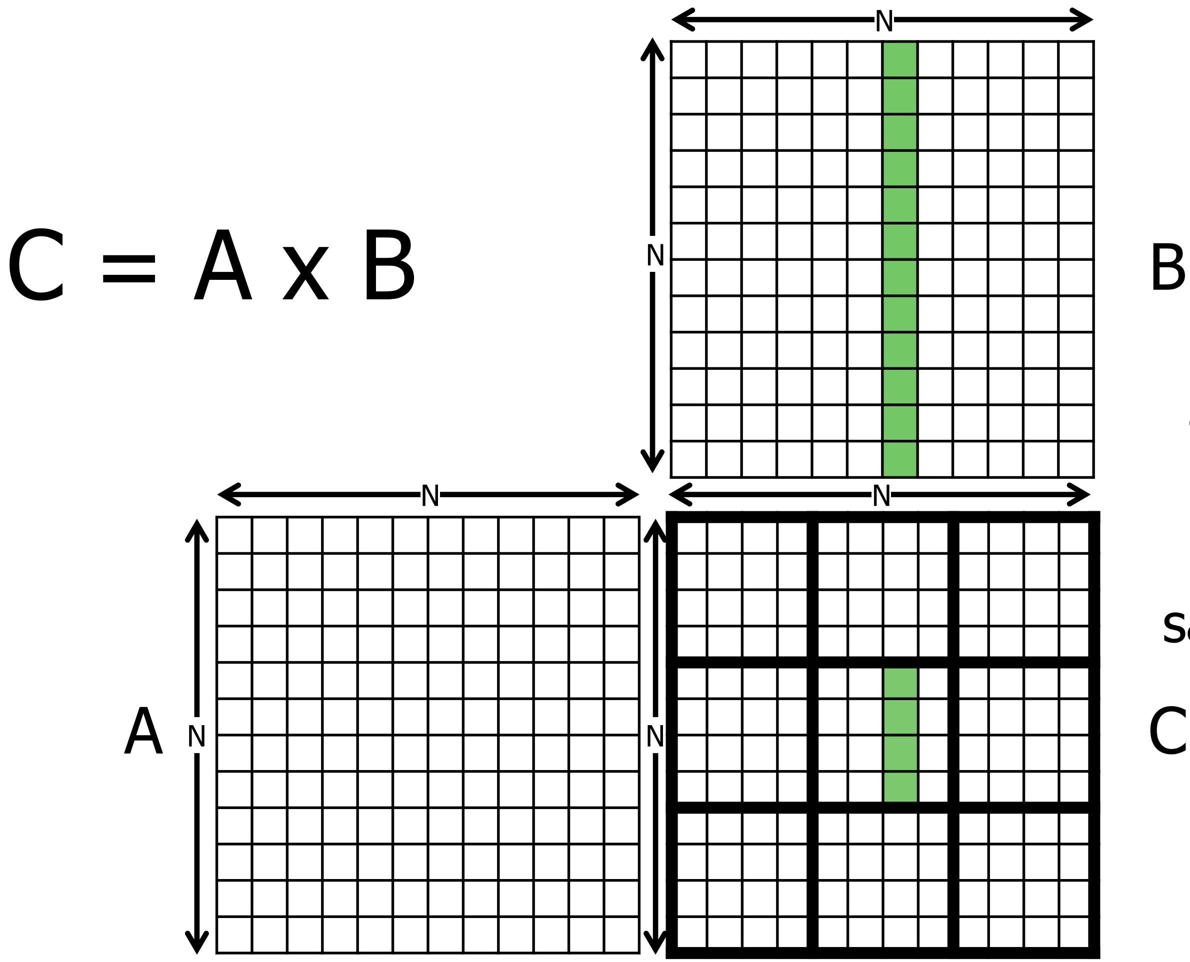
Reuse in Matrix-Matrix Multiplication (I)



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

Slide credit: Izzat El Hajj

Reuse in Matrix-Matrix Multiplication (II)



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

Slide credit: Izzat El Hajj

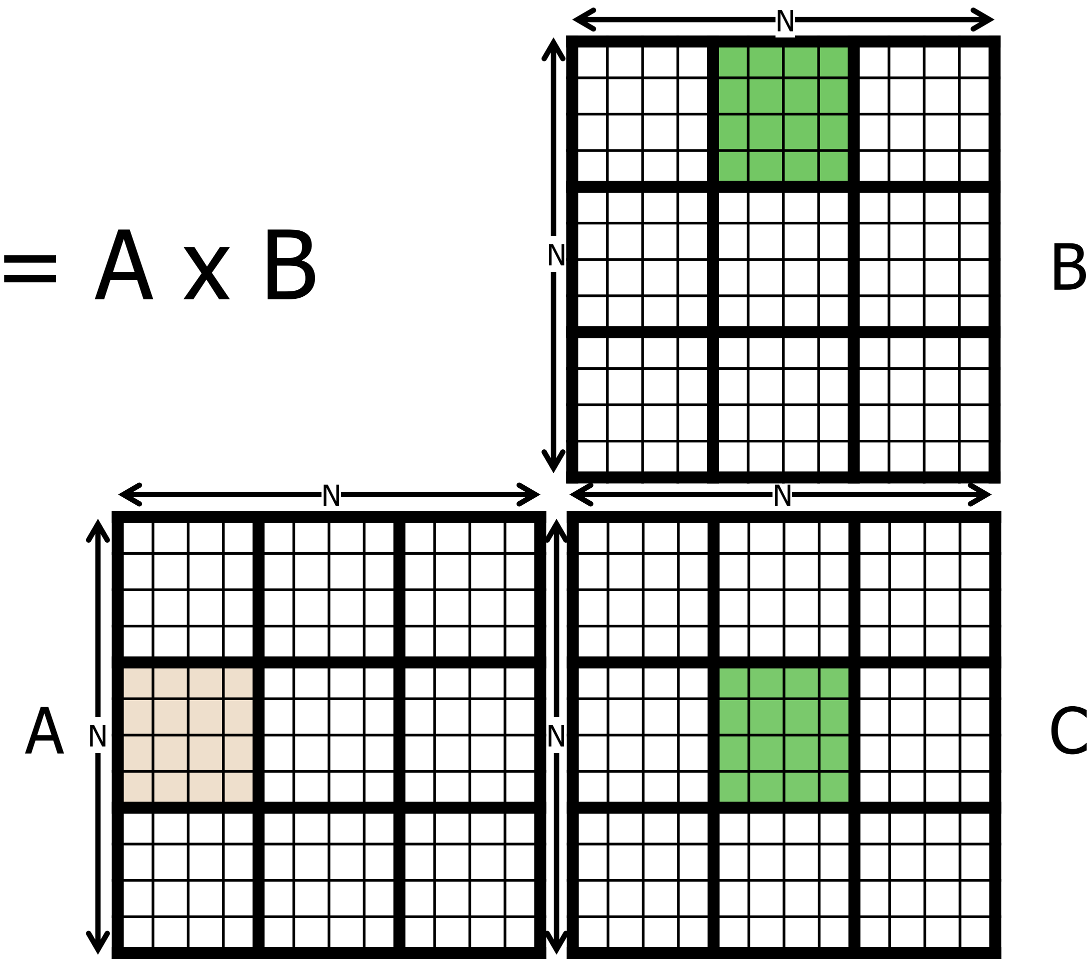
Reuse in Matrix-Matrix Multiplication (III)

- 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

Slide credit: Izzat El Hajj

Tiled Matrix-Matrix Multiplication (I)

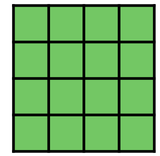
C = A x B



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

Slide credit: Izzat El Hajj

Tiled Matrix-Matrix Multiplication (II)

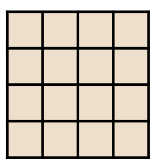


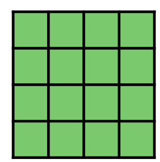
B_{tile}

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

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

A_{tile}



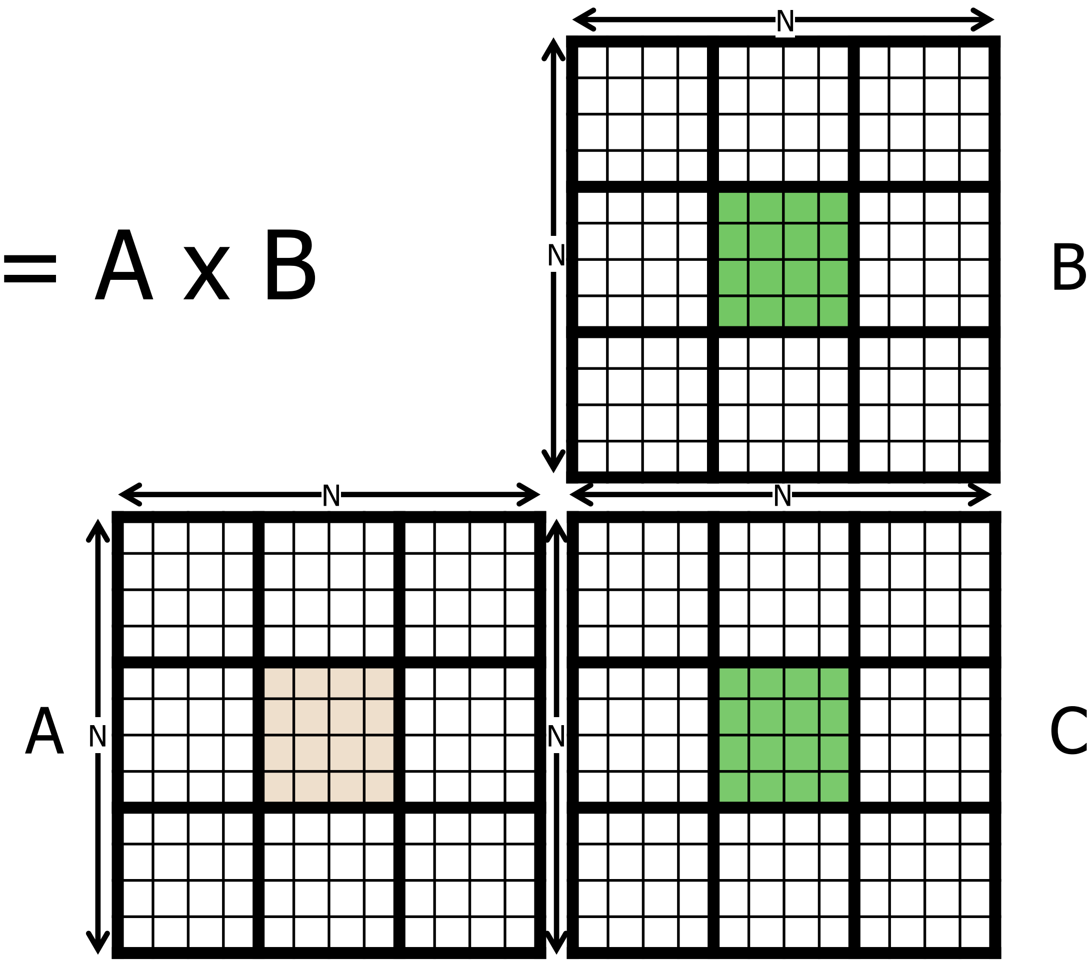


C_{tile}

Slide credit: Izzat El Hajj

Tiled Matrix-Matrix Multiplication (III)

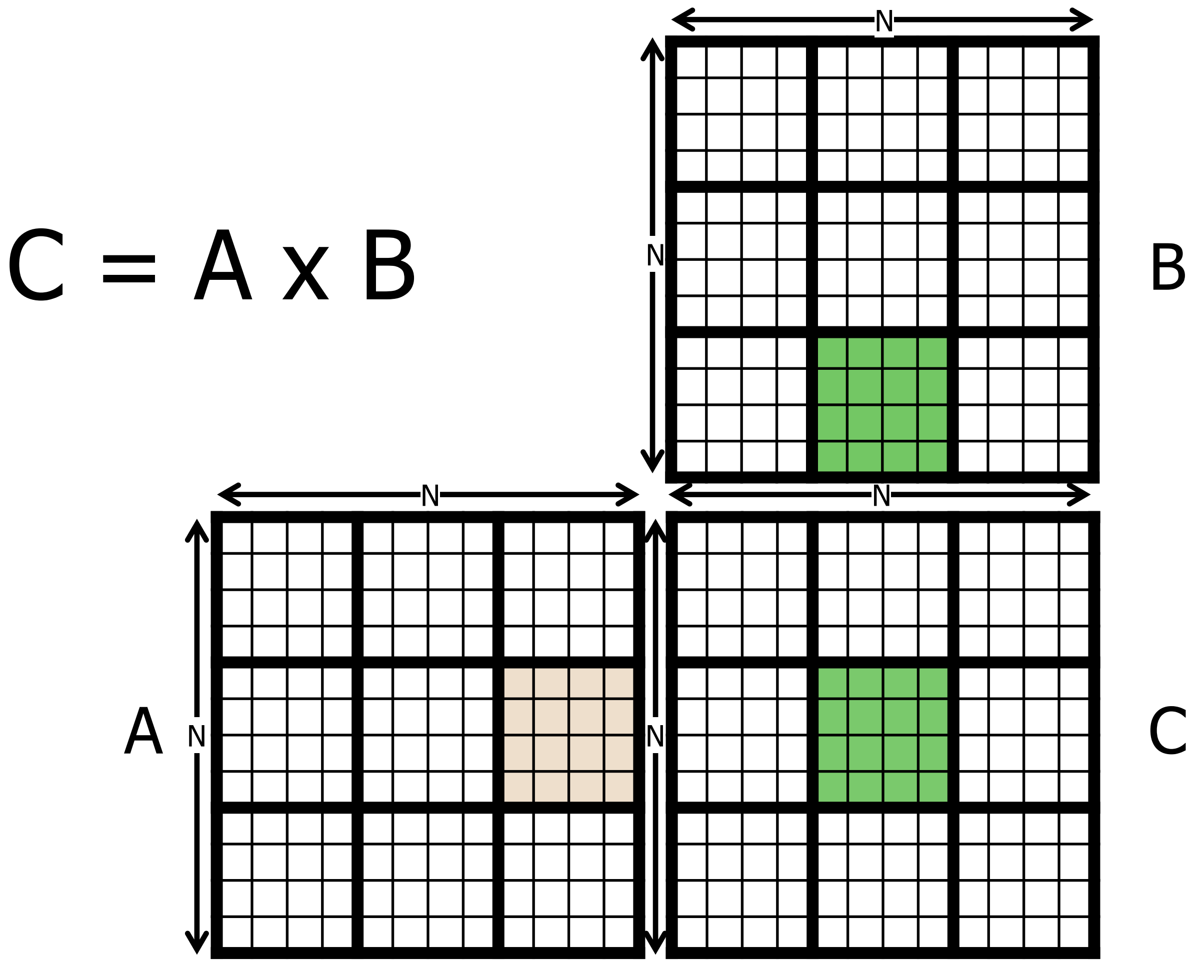
$C = A \times B$



...repeat for the next tile

Slide credit: Izzat El Hajj

Tiled Matrix-Matrix Multiplication (IV)



...and the next tile

Slide credit: Izzat El Hajj

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

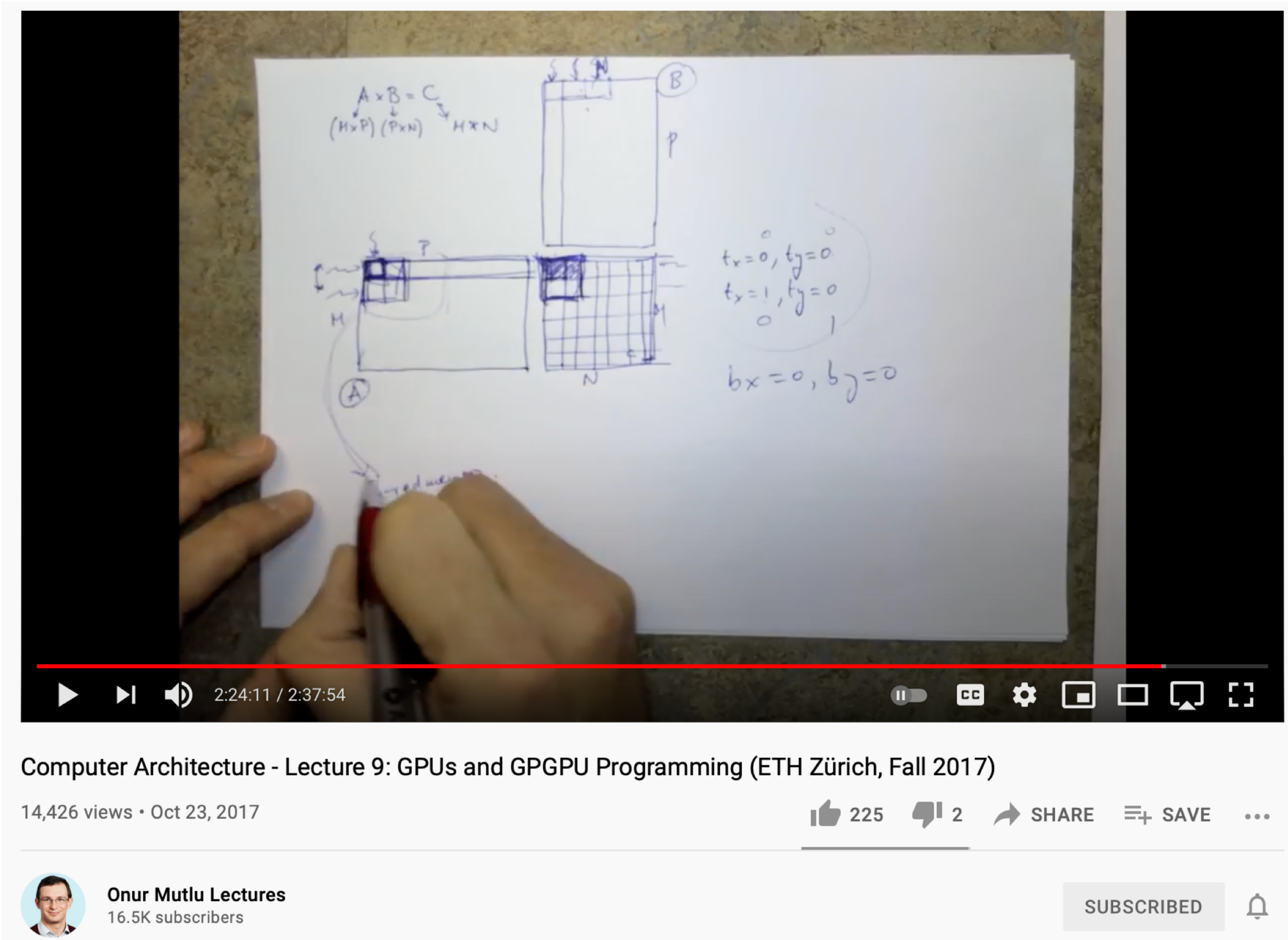
```
    // Compute with tile
    for(unsigned int i = 0; i < TILE_DIM; ++i) {
        sum += A_s[threadIdx.y][i]*B_s[i][threadIdx.x];
    }
    __syncthreads();
```

Threads wait for each other to finish computing before loading

```
}

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


Slide credit: Izzat El Hajj

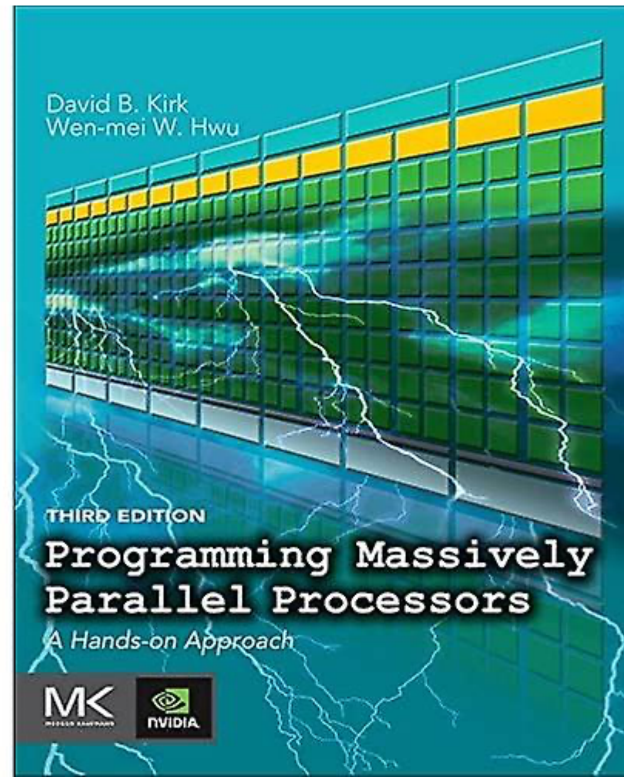
Tiled Matrix Multiplication on GPU



Computer Architecture - Lecture 9: GPUs and GPGPU Programming (Fall 2017) https://youtu.be/mgtlbEqn2dA?t=8157

Recommended Readings

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



P&S Heterogeneous Systems

GPU Memory Hierarchy

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zürich

Fall 2021

28 October 2021