GPU Programming

Farshad Khunjush

Department of Computer Science and Engineering
Shiraz University
Fall 2025

Intoduction to GPUs
Part 2

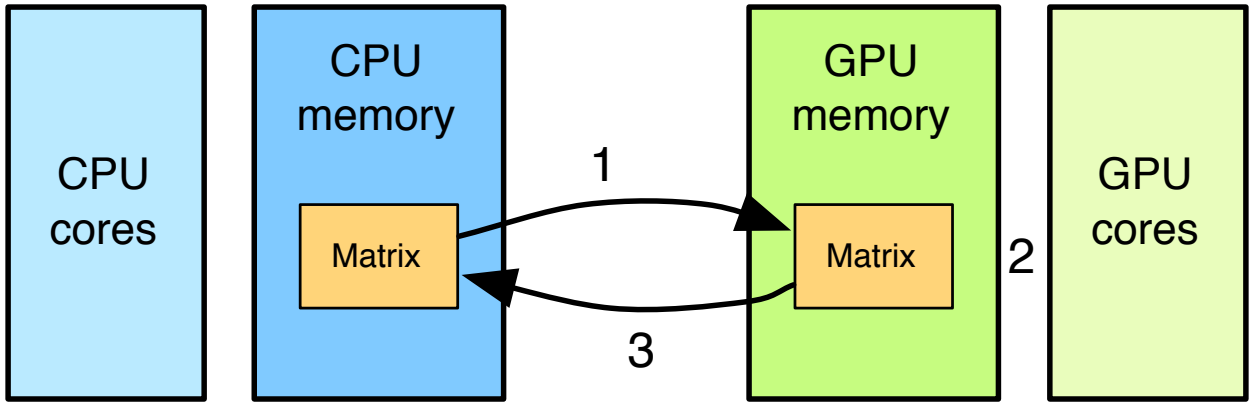
Farshad Khunjush

Some slides come form

Professor Babak Falsafi @ EPFL

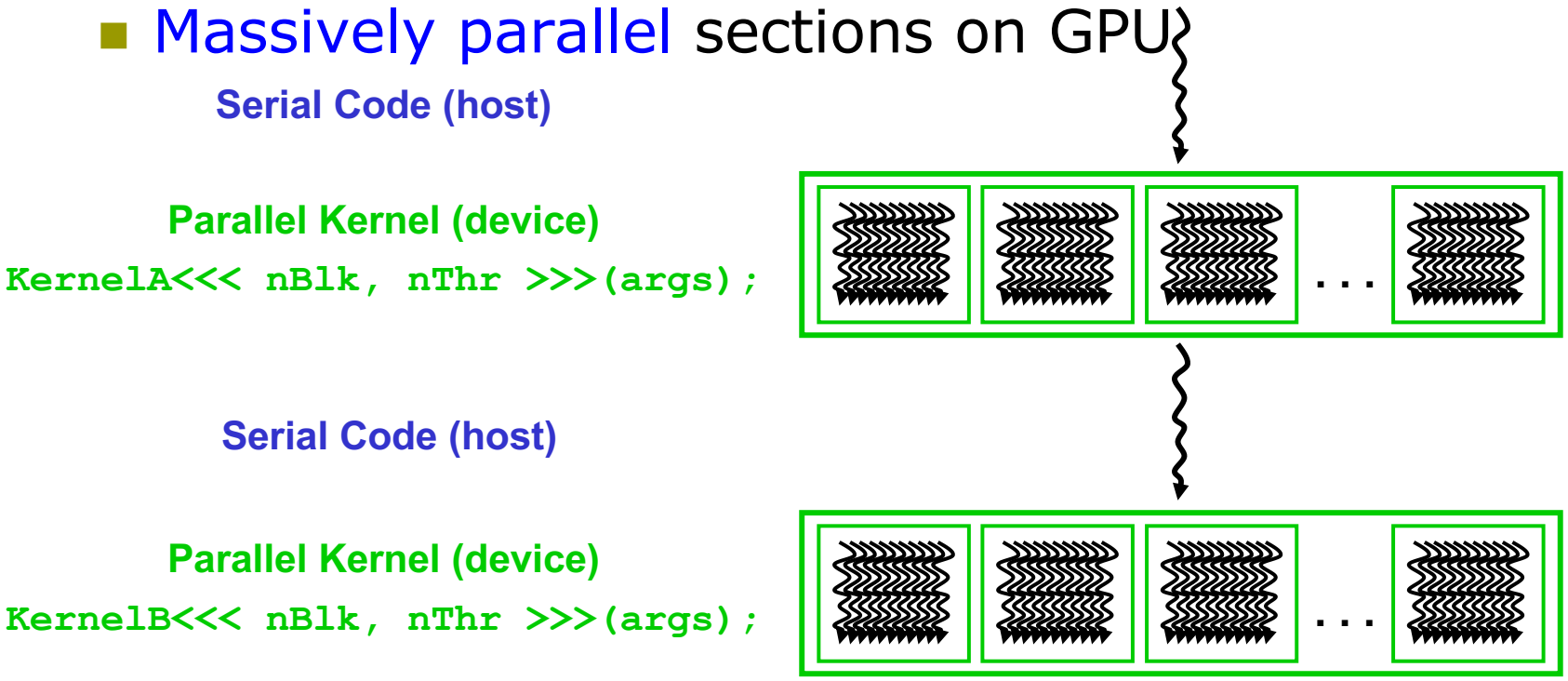
GPU Computing

- 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

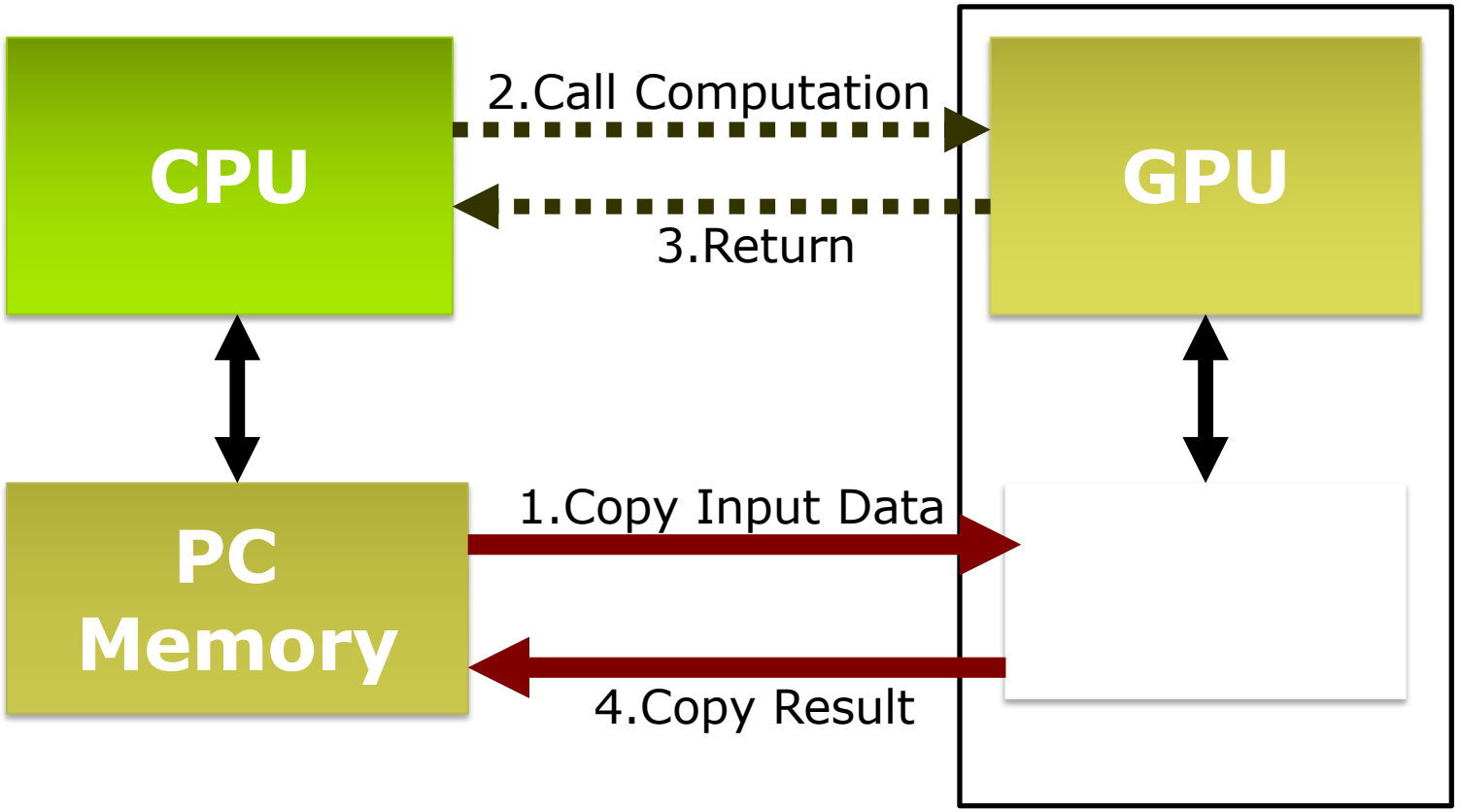


Slide credit: Hwu & Kirk

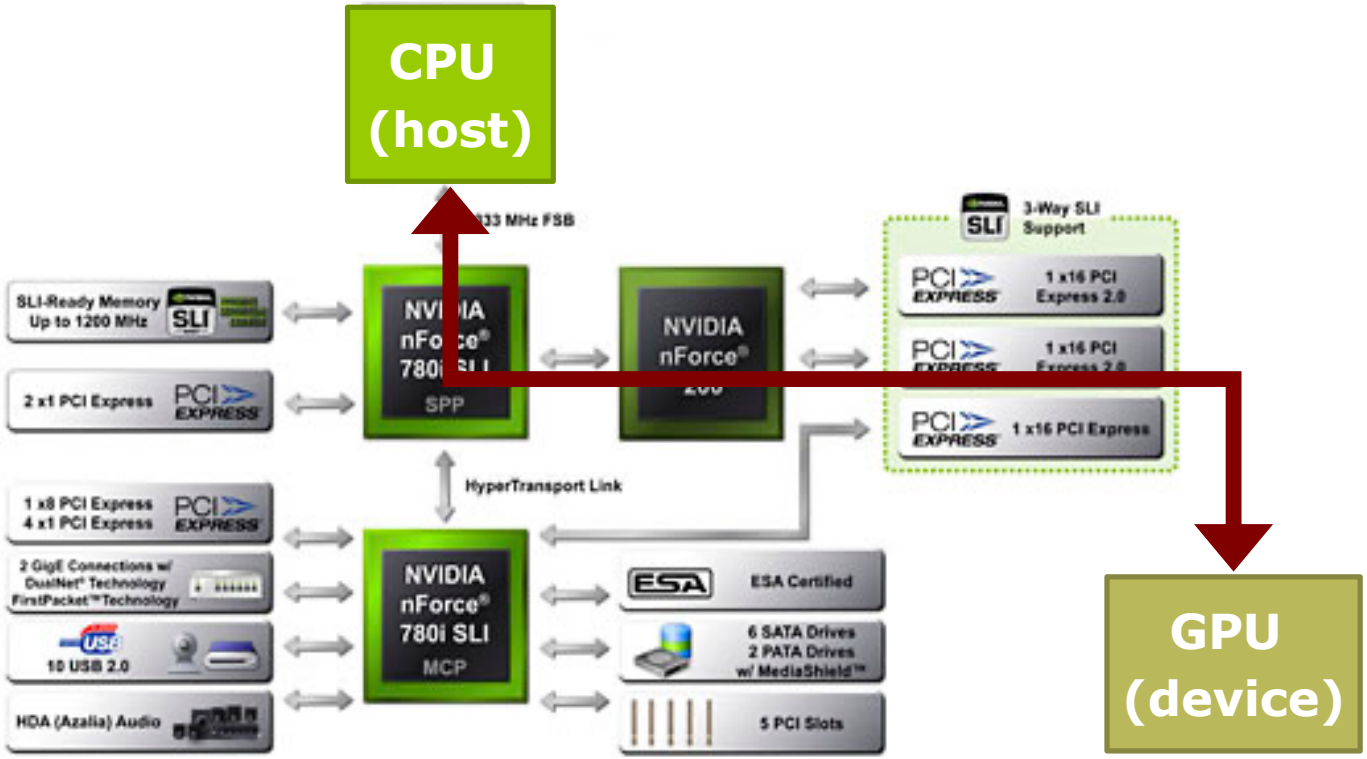
CUDA
(Compute Unified Device Architecture)

- General purpose programming model developed by NVIDIA.
  - It is designed to support joint CPU/GPU execution of an application.
- It is accessible to software developers through standard programming languages.
- Driver for loading computation programs into GPU
  - Standalone Driver - Optimized for computation
  - Interface designed for compute – graphics-free API
  - Explicit GPU memory management

How does it work?



CPU-GPU Communication Path

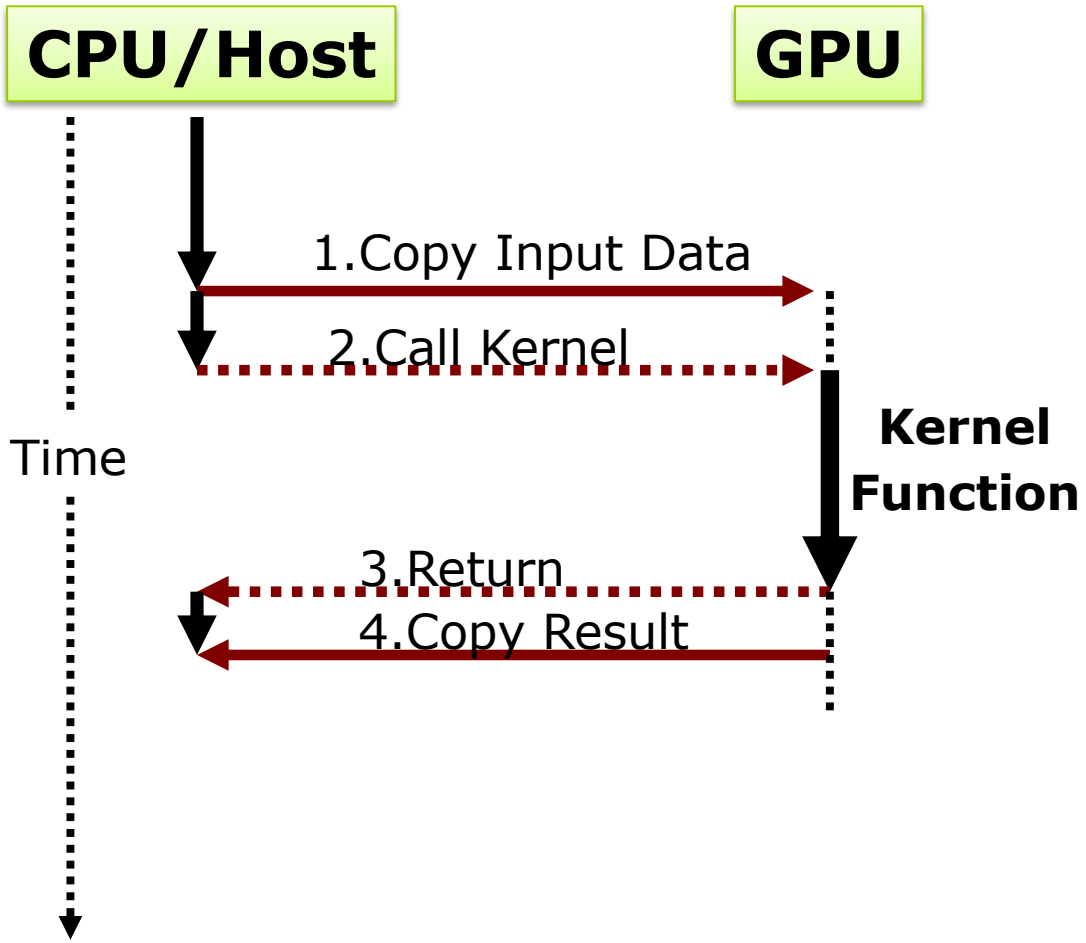


- ~1000 cycles latency
- ~10GB/s bandwidth limitation
- Only coarse-grained offloading

Source:NVIDIA

CPU-GPU Interaction

- GPU:
  - Runs a number of compute-intensive ***Kernels***
- CPU:
  - Provides data for GPU and manages it
  - Runs the rest of the computation



CUDA Devices and Threads

- A Compute device
  - A coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
  - Is typically a GPU but can also be another type of parallel processing device
- Data-parallel portions of an application are expressed as device kernels which run on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

CUDA Language: Extended C

- Type Qualifiers

```
__global__ void KernelFunc(...) // kernel function,          runs on device

__device__ int GlobalVar;       //variable in device        memory

__shared__ int SharedVar;       //variable in shared              memory
```

CUDA Language: Extended C

- Special variables for thread identification in kernels

  dim3 threadIdx; dim3 blockIdx;

  dim3 blockDim; dim3 gridDim;

- Intrinsic that expose specific operations in kernel code

  __synchthreads(); //barrier synchronization within kernel

CUDA Language: Extended C

- Extend function invocation syntax for parallel kernel launch

  KernelFunc**<<<500, 128>>>**(…) //launch 500 blocks with 128 threads each

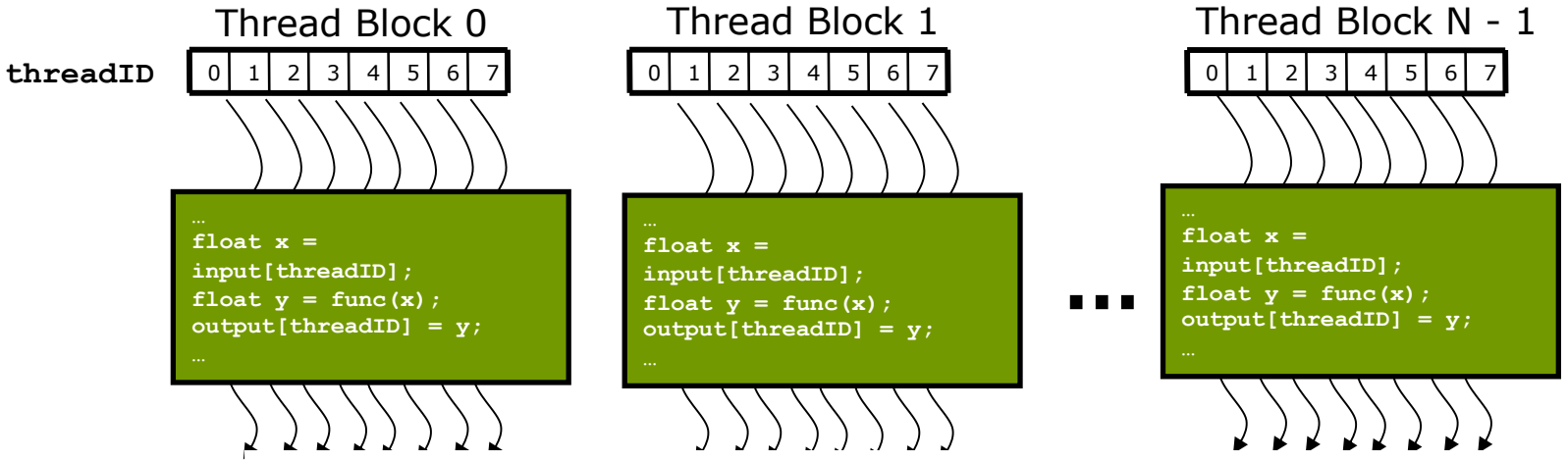
CUDA Kernel

- A piece of code executed on GPU (device)
- A CUDA kernel is executed by an array of parallel threads
- All threads run the same code (SPMD model)
- Each thread has a unique ID

Thread Blocks: Scalable Cooperation

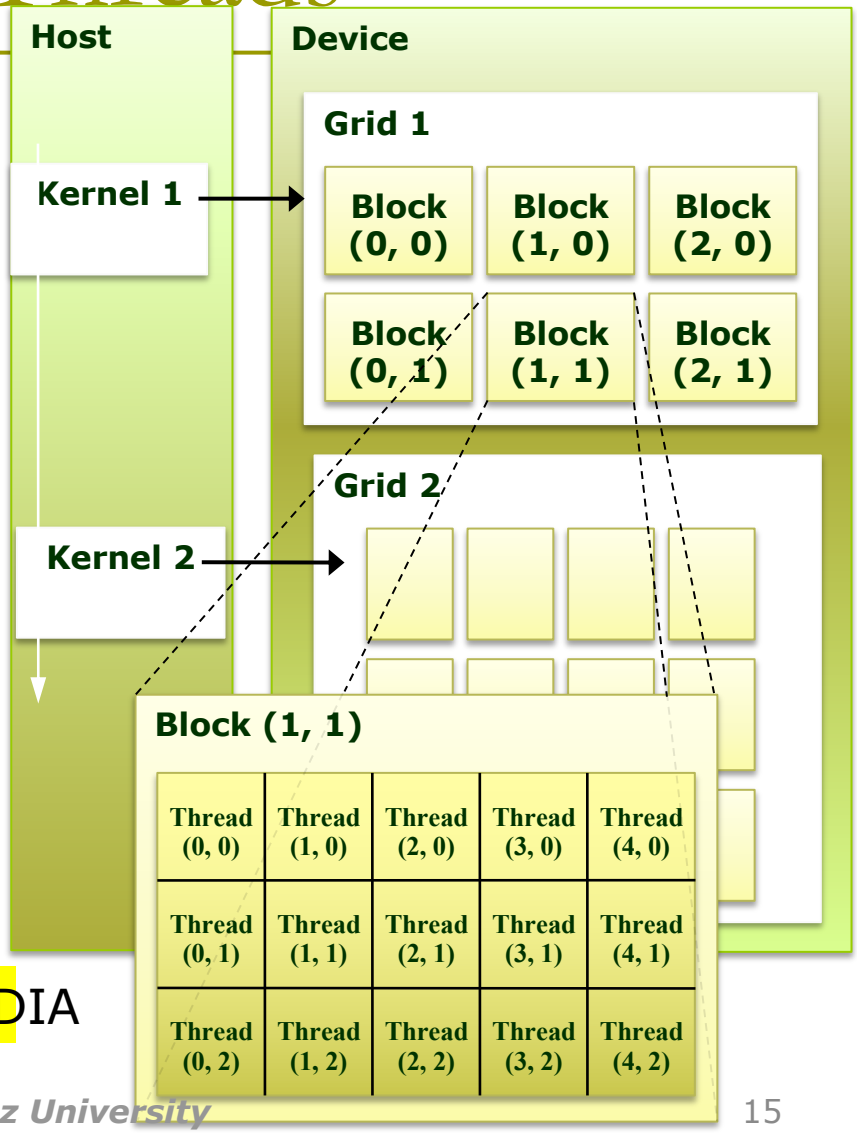
Divide monolithic thread array into multiple blocks

- Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization**
- Threads in different blocks cannot cooperate



Grids of Blocks of Threads

- Grid is launched on an array of Streaming Multiprocessors
- Thread Blocks are serially distributed to all the SM's
  - Potentially >1 Thread Block per SM



Source: NVIDIA

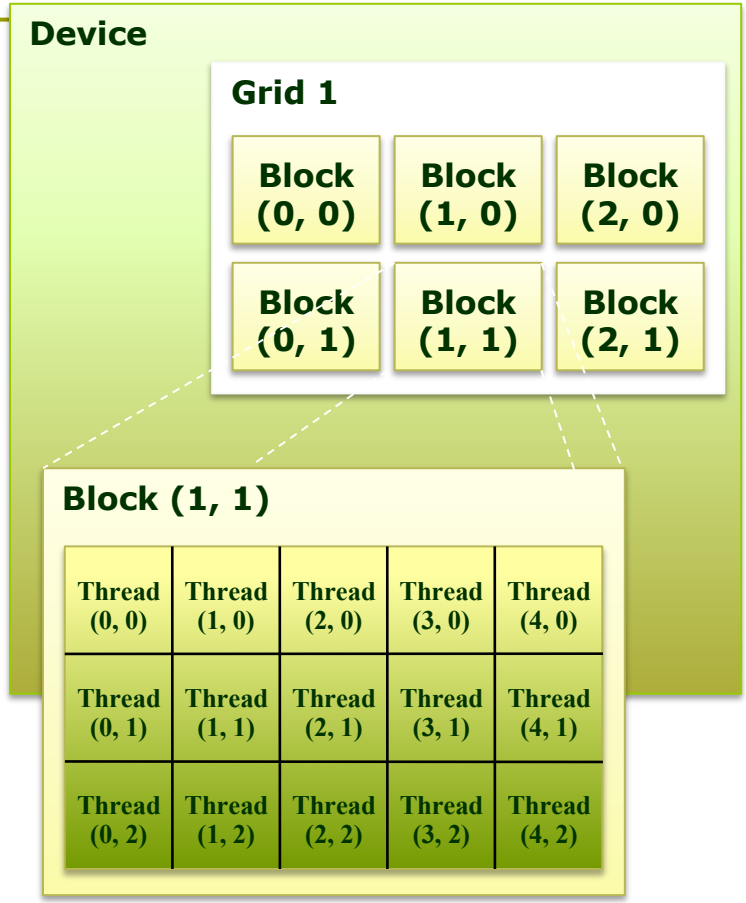
Dimension Limits

- Grid of Blocks 1D, 2D, or 3D
  - Max x: 65535
  - Max y: 65535
- Block of Threads: 1D, 2D, or 3D
  - Max number of threads: 1024
  - Max x: 1024
  - Max y: 1024
  - Max z: 64
- Limits apply to Compute Capability 1.0, 1.1, 1.2,1.3, and 2.0
  - GTX480 = 2.0

Block and Thread IDs

- Threads and blocks have IDs
  - So each thread can decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data

IDs and dimensions are easily accessible through predefined "variables", e.g., **`blockDim.x`** and **`threadIdx.x`**



source: NVIDIA

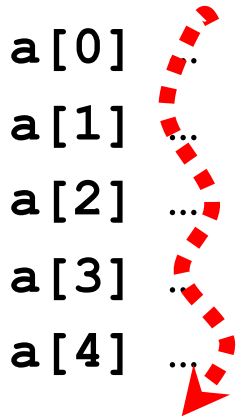
Designing a Kernel: Basic Example

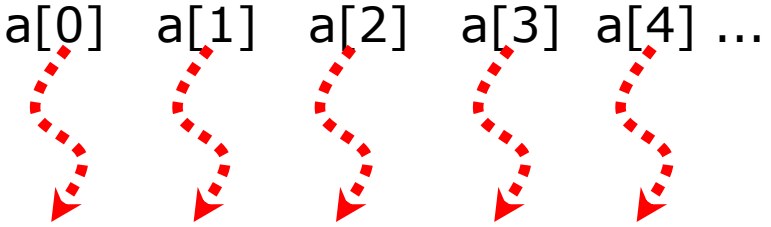
Sequential Code

```
  /* N is large */
for (i = 0; i < N; i++)
{
  a[i] = a[i] * c;
}
```

Parallel Code

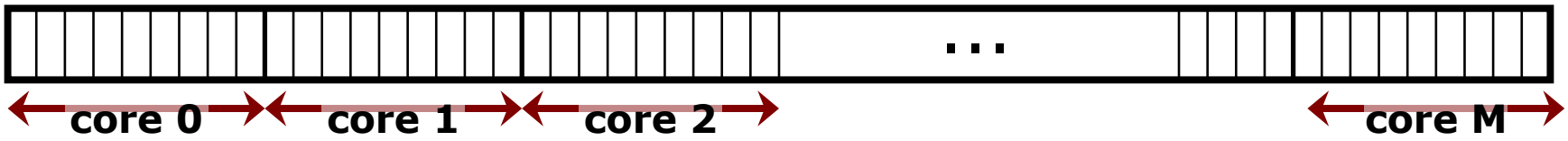
```
/* assuming you have one
core per array entry */
i = coreIndex;
a[i] = a[i] * c;
```





But we don't have a million cores!

Partitioning Input Data




```
/* Code executed by each core */
partitionSize = N / numberOfCores;
start = myCoreIndex * partitionSize;
end = start + partitionSize -1;

for (i = start; i < end; i++) {
  a[i] = a[i] * c;
}
```

Provided by the system

CUDA Execution Model

- All blocks are identical
  - Same structure, same number of threads
- Block execution order is undefined
  - Thread block 1 can execute **before** thread block 0 or vice versa
- Synchronization
  - Threads within the same block can synchronize through shared memory
  - Threads from different blocks cannot communicate => cannot synchronize => cannot cooperate
- Block-to-Processor assignment
  - Multiple thread blocks can be assigned to the same SM
  - Thread blocks do not migrate from one SM to another during kernel execution

CUDA Example

- Kernel Candidate:

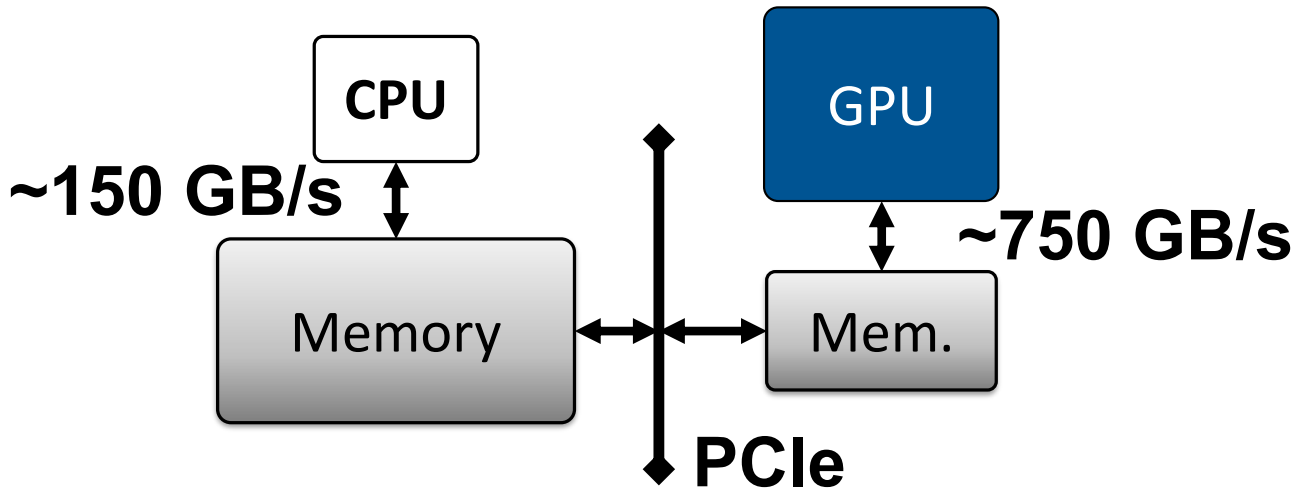
```
  /* N is large */
for (i = 0; i < N; i++) {
  a[i] = a[i] * c;
}
```

- Steps:

1. Memory Allocation on CPU and GPU
2. Initialize Data on CPU
3. Copy Data from CPU to GPU
4. Define *Execution Configuration*
5. Run Kernel
6. CPU synchronizes with GPU
7. Copy Data from GPU to CPU
8. Deallocate GPU and CPU memory

GPU Memory Allocation

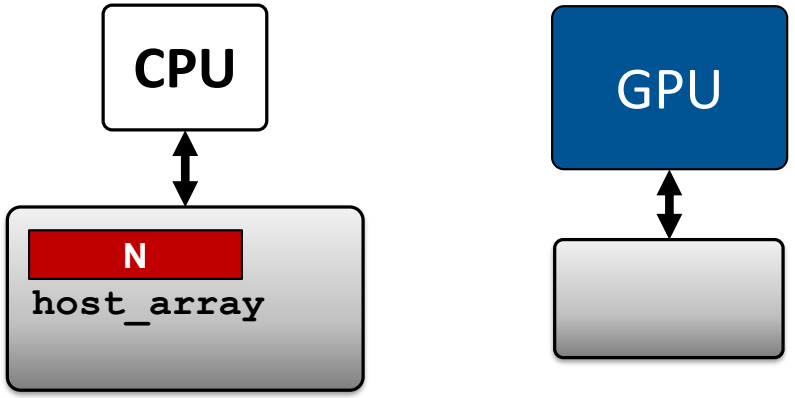
- GPUs don't run an operating system
  - No unified "memory manager", up to the programmer
  - Corollary: your code can step on other people's code!
- CPU allocates, copies, and deallocates



GPU Memory Allocation

- Allocation and Deallocation is similar to using C
  - cudaMalloc(...) and cudaFree(...)
- Example:

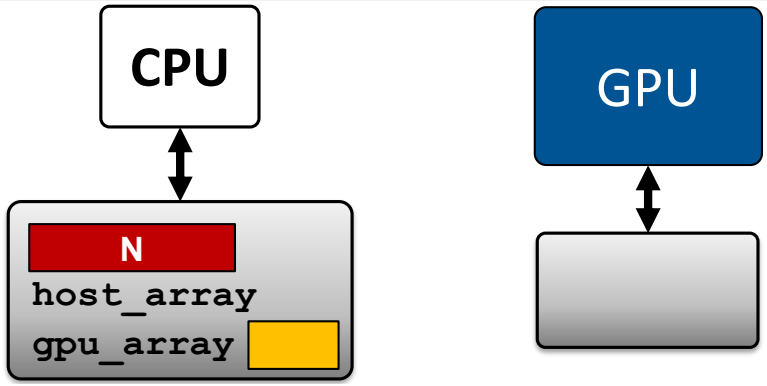
```
int main(...)
{
    int host_array[N];
    int* gpu_array;
    cudaMalloc( (void**)&gpu_array, SIZE_N);
}
```



GPU Memory Allocation

- Allocation and Deallocation is similar to using C
  - cudaMalloc(...) and cudaFree(...)
- Example:

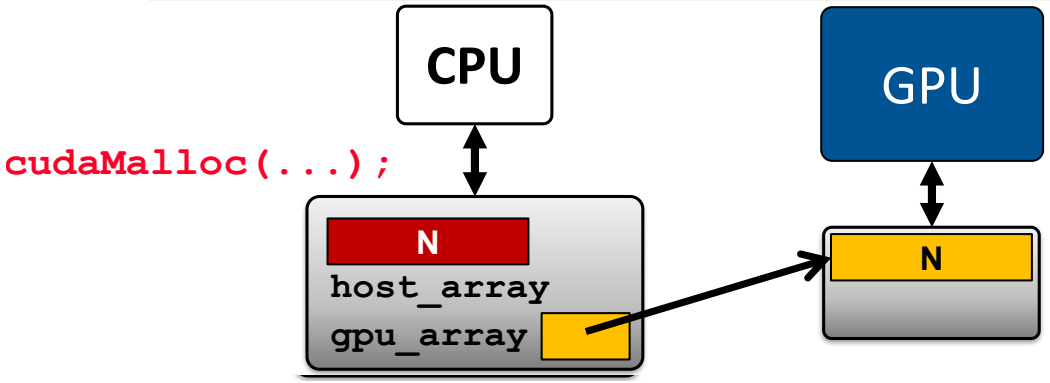
```
int main(...)
{
    int host_array[N];
    int* gpu_array;
    cudaMalloc( (void**)&gpu_array, SIZE_N);
}
```



GPU Memory Allocation

- Allocation and Deallocation is similar to using C
  - cudaMalloc(...) and cudaFree(...)
- Example:

```
int main(...)
{
     int host_array[N];
     int* gpu_array;
     cudaMalloc( (void**)&gpu_array, SIZE_N);
}
```



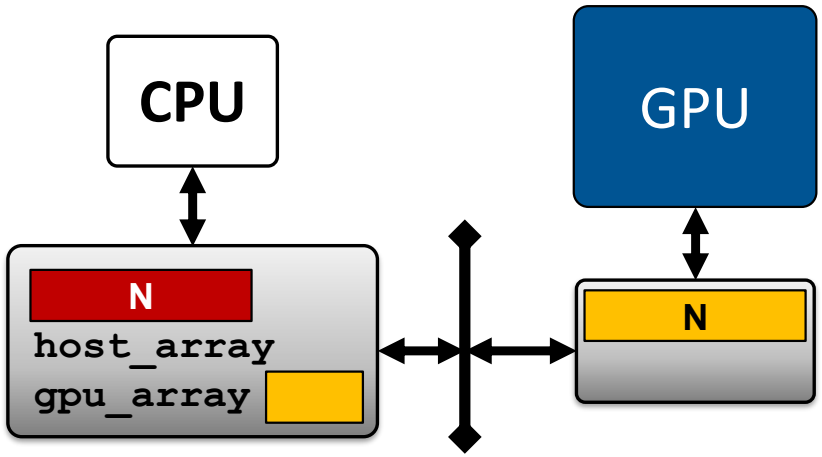
How Does gpu_array Pointer Work?

- Disclaimer: It is **not** a regular C pointer!
  - If you dereference it, the behavior is undefined
- The host **pointer** is on the CPU's stack
  - But, it serves as a simple name for the CUDA library
  - When calling cudaMalloc(...), device driver tells the GPU to "please allocate SIZE_N bytes of memory"
- Don't forget, you must copy to/from the GPU!

Copying Data to the GPU

- Now that both arrays are set up, invoke copy
- Example:

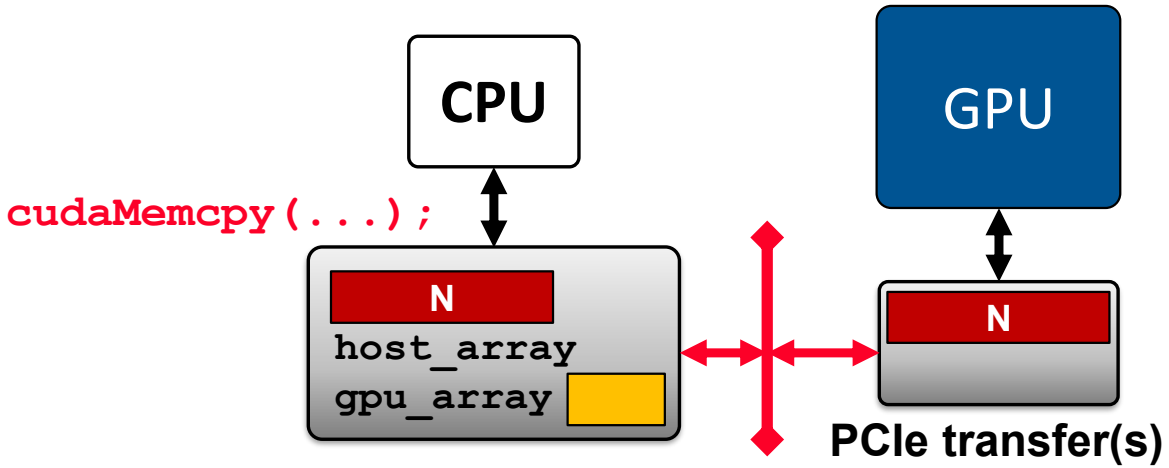
```
int main(...)
{
    ...
    cudaMemcpy((void*)gpu_array,       /* DEST */
               (void*)host_array,      /* SRC */
               SIZE_N,                 /* NBYTES */
               cudaMemcpyHostToDevice /* DIR. */);
}
```



Copying Data to the GPU

- Now that both arrays are set up, invoke copy
- Example:

```
int main(...)
{
    ...
    cudaMemcpy((void*)gpu_array,       /* DEST */
               (void*)host_array,      /* SRC */
               SIZE_N,                 /* NBYTES */
               cudaMemcpyHostToDevice /* DIR. */);
}
```



Copying Data to the GPU (3)

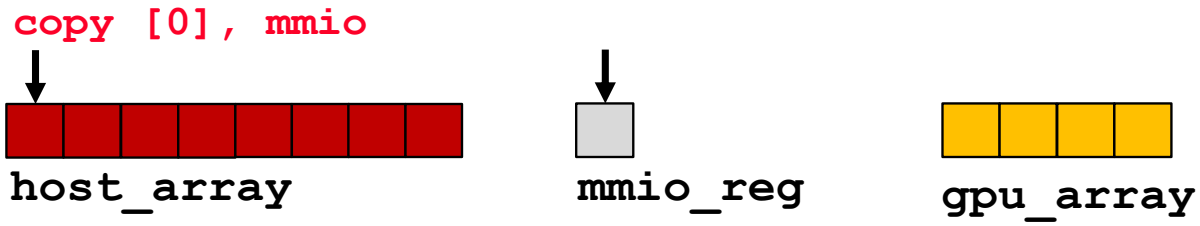
- Things to remember about transferring memory:
- The copy **direction** matters for your results
- (You will probably want 1 of the first 2)

```
enum MemCpyKind {
    cudaMemcpyHostToDevice,
    cudaMemcpyDeviceToHost,
    cudaMemcpyDeviceToDevice
};
```

- When copying data to the device, cudaMemcpy() is an **asynchronous** function, CPU keeps running
- How?? Why??

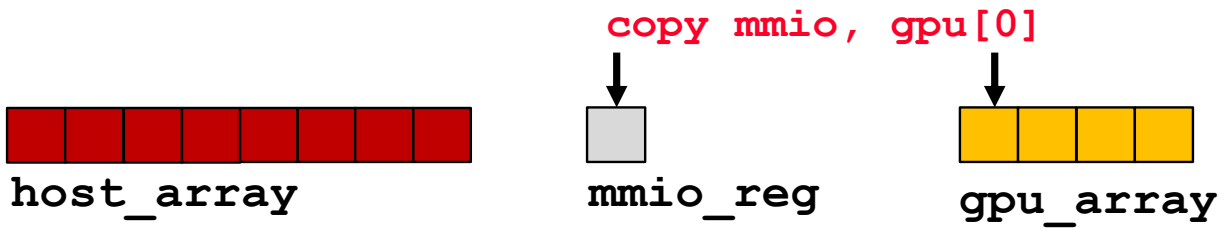
Copying Data to the GPU (3)

- CudaMemcpy() is an **asynchronous** call, why?
  - Answer: **It is a DMA (Direct Memory Access)**
- CPUs can perform I/O in 2 common ways
  - Memory Mapped I/O (MMIO) and DMA
- MMIO is inefficient as it means every piece of data goes through the CPU first
  - In this case, imagine where N = 1GB! CPU would issue 230 / 4B = 256K memory read/writes.



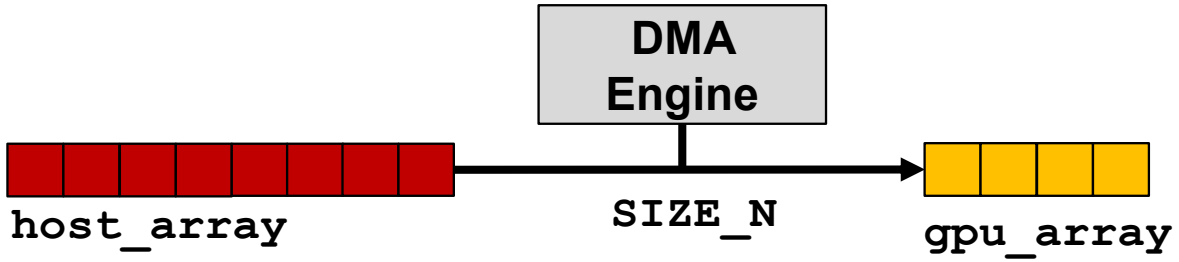
Copying Data to the GPU (3)

- CudaMemcpy() is an **asynchronous** call, why?
  - Answer: **It is a DMA (Direct Memory Access)**
- CPUs can perform I/O in 2 common ways
  - Memory Mapped I/O (MMIO) and DMA
- MMIO is inefficient as it means every piece of data goes through the CPU first
  - In this case, imagine where N = 1GB! CPU would issue 230 / 4B = 256K memory read/writes.



Copying Data to the GPU (3)

- CudaMemcpy() is an **asynchronous** call, why?
  - Answer: **It is a DMA (Direct Memory Access)**
- CPUs can perform I/O in 2 common ways
  - Memory Mapped I/O (MMIO) and DMA
- MMIO is inefficient as it means every piece of data goes through the CPU first
  - In this case, imagine where N = 1GB! CPU would issue 230 / 4B = 256K memory read/writes.



Step 1: Memory Allocation

- Allocating host memory

```
float *host_array;
host_array = (float *)malloc(sizeof(float) * N);
if (host_array == NULL) {
        printf("ERROR: cannot allocate memory \n");
        exit(-1);
 }
```

- Allocating device (GPU) memory

```
cudaError_t error;
float *device_array;
error = cudaMalloc((void *)&device_array,
        sizeof(float) * N);
if (error != cudaSuccess) {
        printf("ERROR: cannot allocate memory \n");
        ...
```

Step 2: Initialization

```
for (j = 0 ; j < N; j++) {
     host_array[j] = j;
}
```

- No direct access to the GPU memory
  - Distributed Memory (as opposed to shared memory)
  - Access only through **`cudaxxxx`** functions

Step 3: Copy Data From CPU to GPU

```
float *host_array;
float *device_array;
...

cudaMemcpy((void *)device_array,     // Destination
           (void *)host_array,       // Source
            sizeof(float) * N,       // Size
           cudaMemcpyHostToDevice); // Direction
```

Host/Device Data Transfers

- The host initiates all transfers:

```
cudaMemcpy(void *dst, void *src,
           size_t nbytes,
           enum cudaMemcpyKind direction);
```

- Asynchronous from the CPU's perspective
  - CPU thread continues

- In-order processing with other CUDA requests

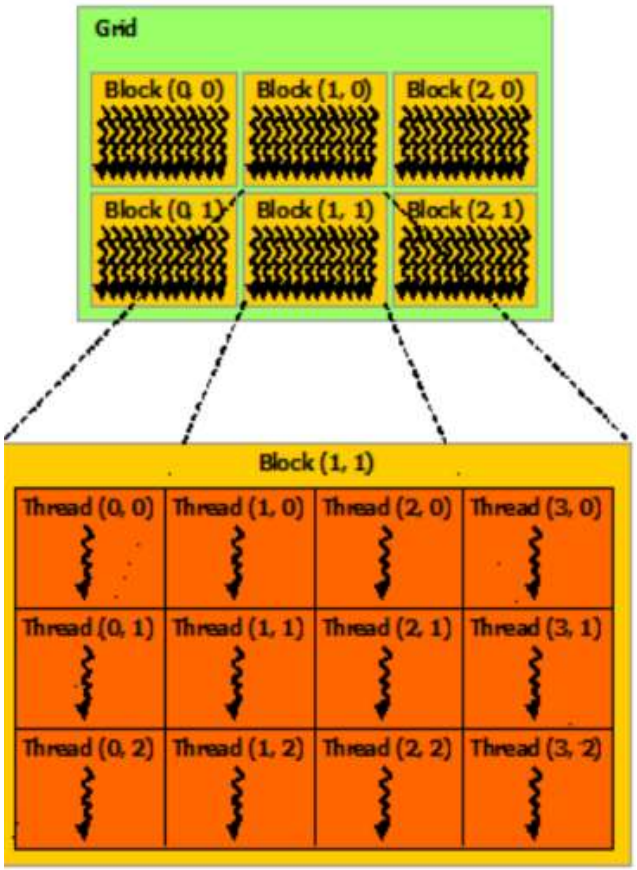
```
enum cudaMemcpyKind {
    cudaMemcpyHostToDevice,
    cudaMemcpyDeviceToHost,
    cudaMemcpyDeviceToDevice
    }
```

CUDA Thread Organization

- Organize threads into "blocks" (TBs)
  - Each TB contains a bunch of threads
  - The TBs are organized as a **grid**
  - Fun indexing – the blocks can be 1D, 2D, or 3D!
- When you call a kernel, you specify:
  - Number of TBs, and threads per TB

Example 2D Thread Organization

- Walk through:
  - Grid = 2 x 3
  - Each block has 12 threads
    - Layout = 3 x 4
- Each thread knows where it sits in layout
  - Accessed by variables in the kernel (coming up!)
- Size limitations online:
  - **CUDA Prog. Guide**



Example 2D Thread Organization

- Pick size based on input data and choice of work division
  - e.g., 1 element per thread, or 4 elements per thread with 1/4 threads?
- Syntax:
  - Specify dimensions in variables typed as int or dim3 (special)
- Code below launches 6 blocks of 12 threads each

```
int main(...) {
    dim3 thrsPerBlock(3,4); // 3x4
    dim3 nBlks( 2,3 ); // 2x3
    image_fade <<< nBlks, thrsPerBlock >>> ( );
}
```

Choosing a Thread Organization

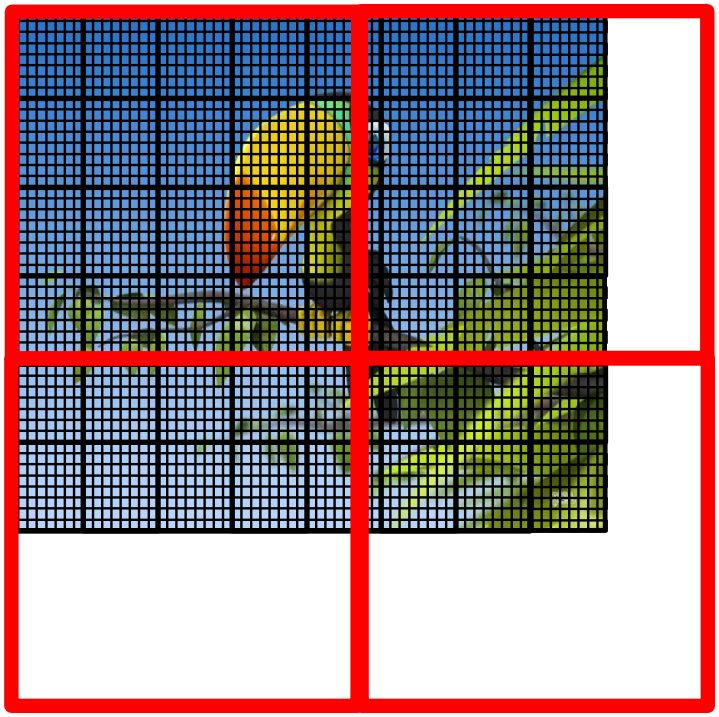
- Fundamental step in GPU programming
  - Dividing up the work between thread blocks
- For max perf., GPU should be fully occupied
  - Every cycle should have a warp ready to issue
  - Requires you to think about your kernel's operations
    - Which ones are long latency?
    - How many warps can be swapped in while this one is waiting?

Occupancy Example

- Assume the image is small (64x48)
  - One thread per pixel means we need ~2k threads
  - But, how to arrange them?
- Example numbers for K40 GPUs
  - 15 SMs, max 64 warps per SM
  - 2048 threads max per SM
  - 16 thread blocks max per SM
  - 1 warp is 32 threads
  - 960 concurrently scheduled warps/GPU
    - You can launch more, but won't start until others finish

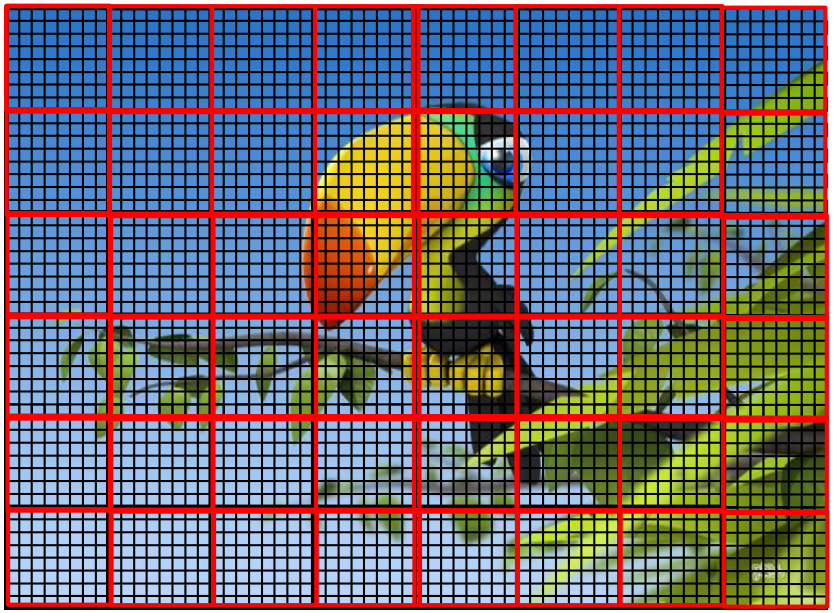
Occupancy Example: Big Blocks

- 4 large thread blocks of 1024 threads each
  - Only 4 of 15 SMs occupied
  - Absolute max occupancy is 26%, not using all HW



Occupancy Example: Smaller Blocks

- 48 thread blocks of 64 threads each
  - SMs have 3 or 4 TB's, each TB happens to be 2 warps
  - Whole image happens in 8 warps (~16 clock cycles!)



Step 4: Define Execution Configuration

```
// we'll see how to find a
// good value for this later
int blk_size = 64;
int num_blks = N / blk_size;

// adding one more block if N is
// not a multiple of thread_block
if (num_blks % N != 0)
{
     num_blks++;
}
```

Step 5: Run the Kernel

```
d_add <<<num_blks, blk_size>>>(device_array, 3.85, N);
```

- Kernel Name: **`d_add`**
- Execution Configuration:

  `<<<num_blks, blk_size>>>`
  - We'll talk more about this later
- Arguments: **`(device_array, 3.85, N)`**

Step6: CPU-GPU Synchronization

- CPU does not block on cuda…() calls
  - Kernel/requests are queued and processed in-order
  - Control returns to CPU immediately
- Good if there is other work to be done
  - e.g., preparing for the next kernel invocation
- Eventually, CPU must know when GPU is done
- Then it can safely copy the GPU results
- **`cudaThreadSynchronize ()`**
  - Block CPU until **all** preceding cuda…() and kernel requests have completed

Step 7: Copy Results from GPU to CPU

```
float *host_array;
float *device_array;
...

cudaMemcpy((void *)host_array,        // Destination
           (void *)device_array,      // Source
            sizeof(float) * N,        // Size
           cudaMemcpyDeviceToHost); // Direction
```

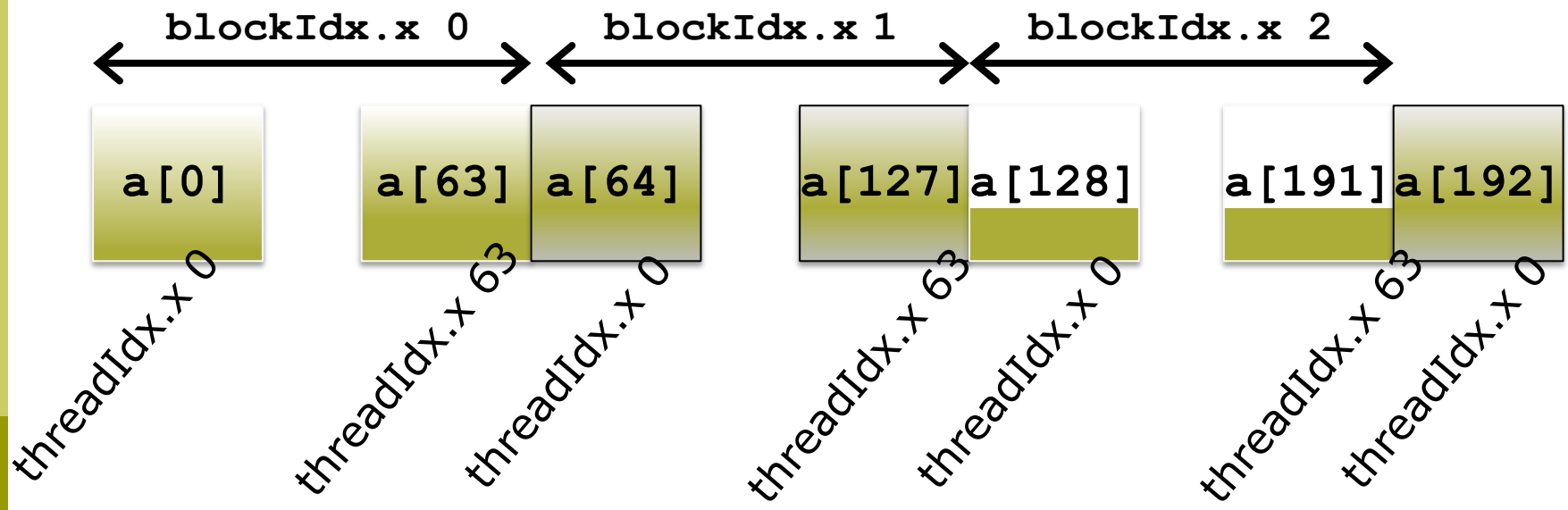
The GPU Kernel

```
__global__ d_add (float *da, float x, int N)
{
  int i;
  i = blockIdx.x * blockDim.x + threadIdx.x;

  if (i < N)
device_array[i] = device_array[i] * x;
}
```

- **BlockIdx**: unique block ID (0,1, …)
- **ThreadIdx**: per block thread ID (0,1, …)
- **BlockDim**: Dimensions of the blocks
  - **BlockDim.x, BlockDim.y, BlockDim.z**)
  - Unused dimensions default to 0

Array Index Calculation Example

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



blockDim.x = 64

source: A. Moshovos

Indexing and Memory Access

- Images are 2D data structures
  - height x width
  - Image[j][i], where 0 ≤ j < height, and 0 ≤ i < width

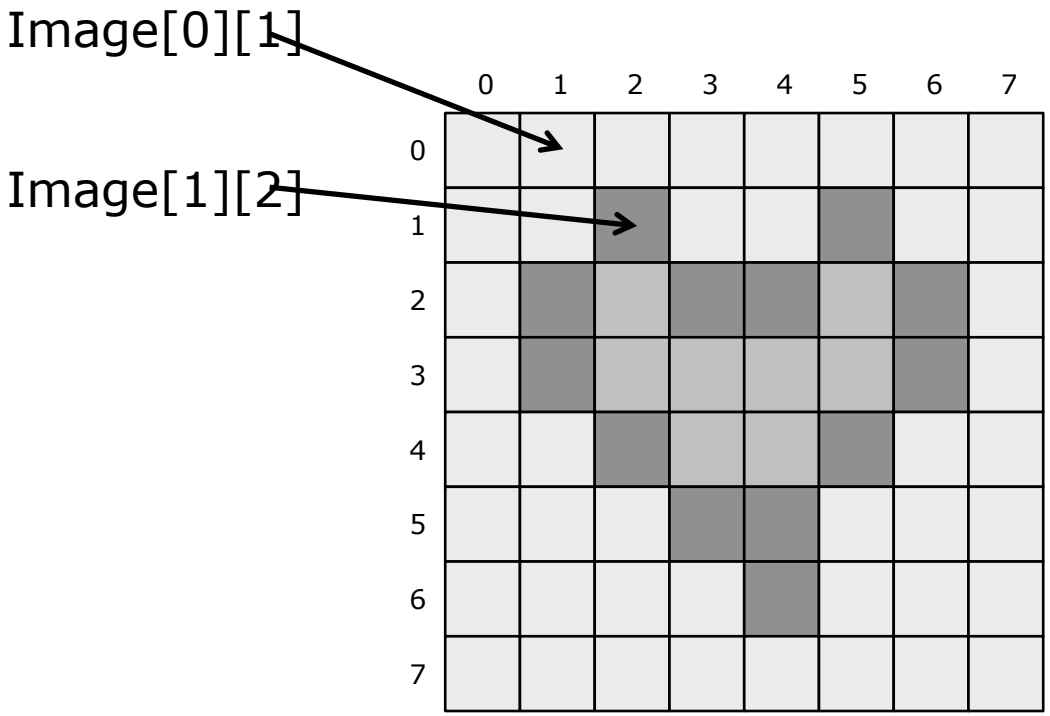
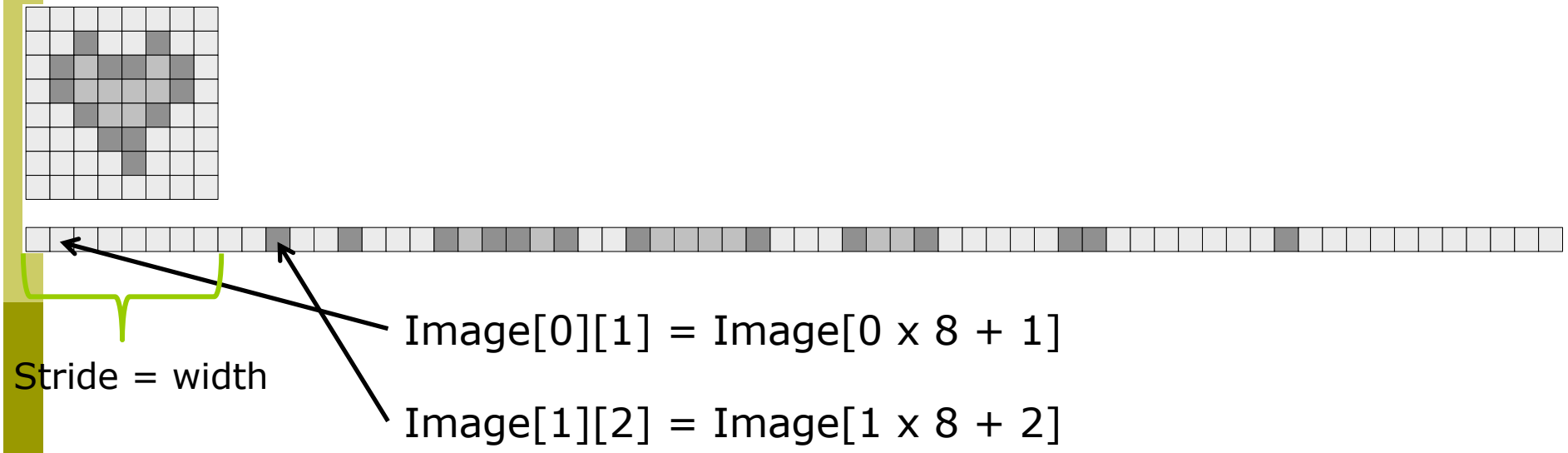


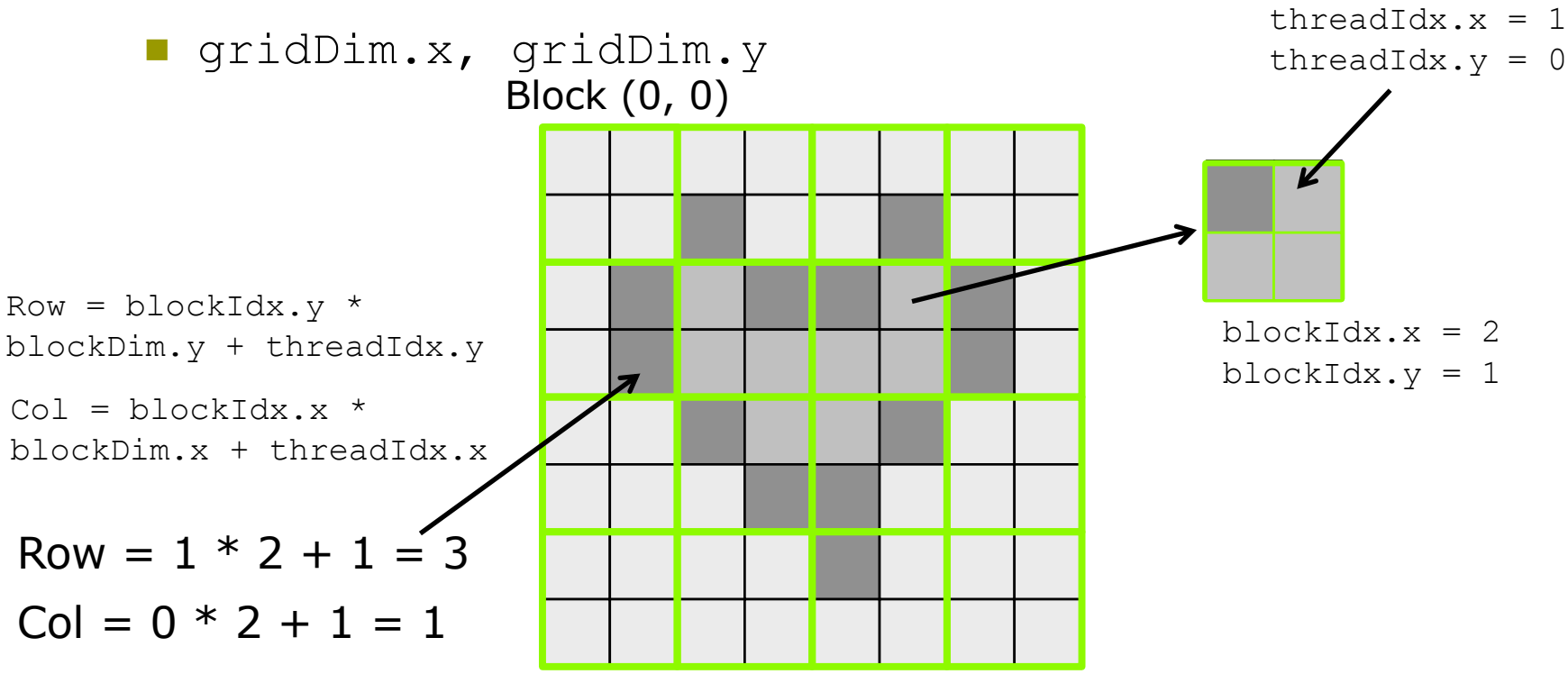
Image Layout in Memory

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



Indexing and Memory Access: 2D Grid

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



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

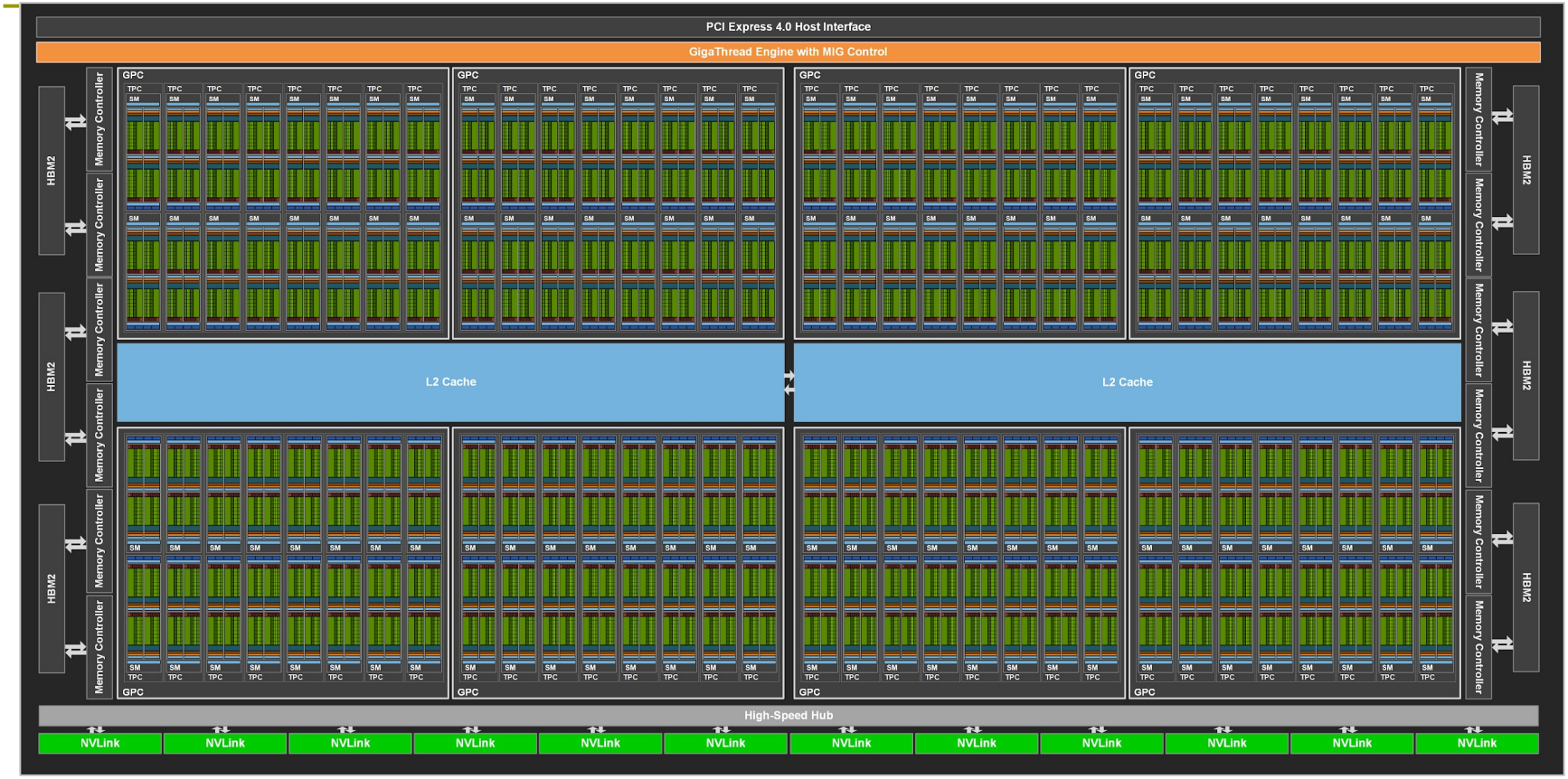
CUDA Function Declarations

	Executed on the:	Only callable from the:
__device__ float DeviceFunc()	device	device
__global__ void KernelFunc()	device	host
__host__ float HostFunc()	host	host

- __global__ defines a kernel function
  - Must return void
  - Can only call __device__ functions
- __device__ and __host__ can be used together

GPU Memories

NVIDIA A100 Block Diagram



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

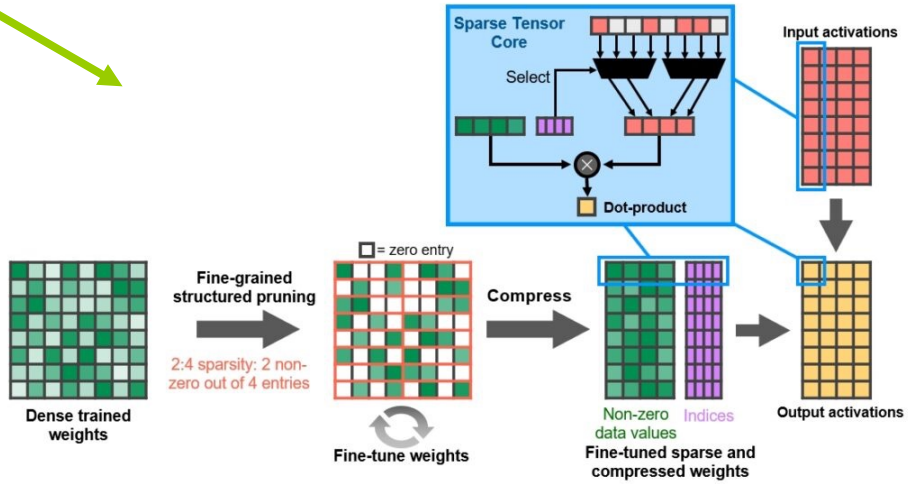
108 cores on the A100

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

40MB L2 cache

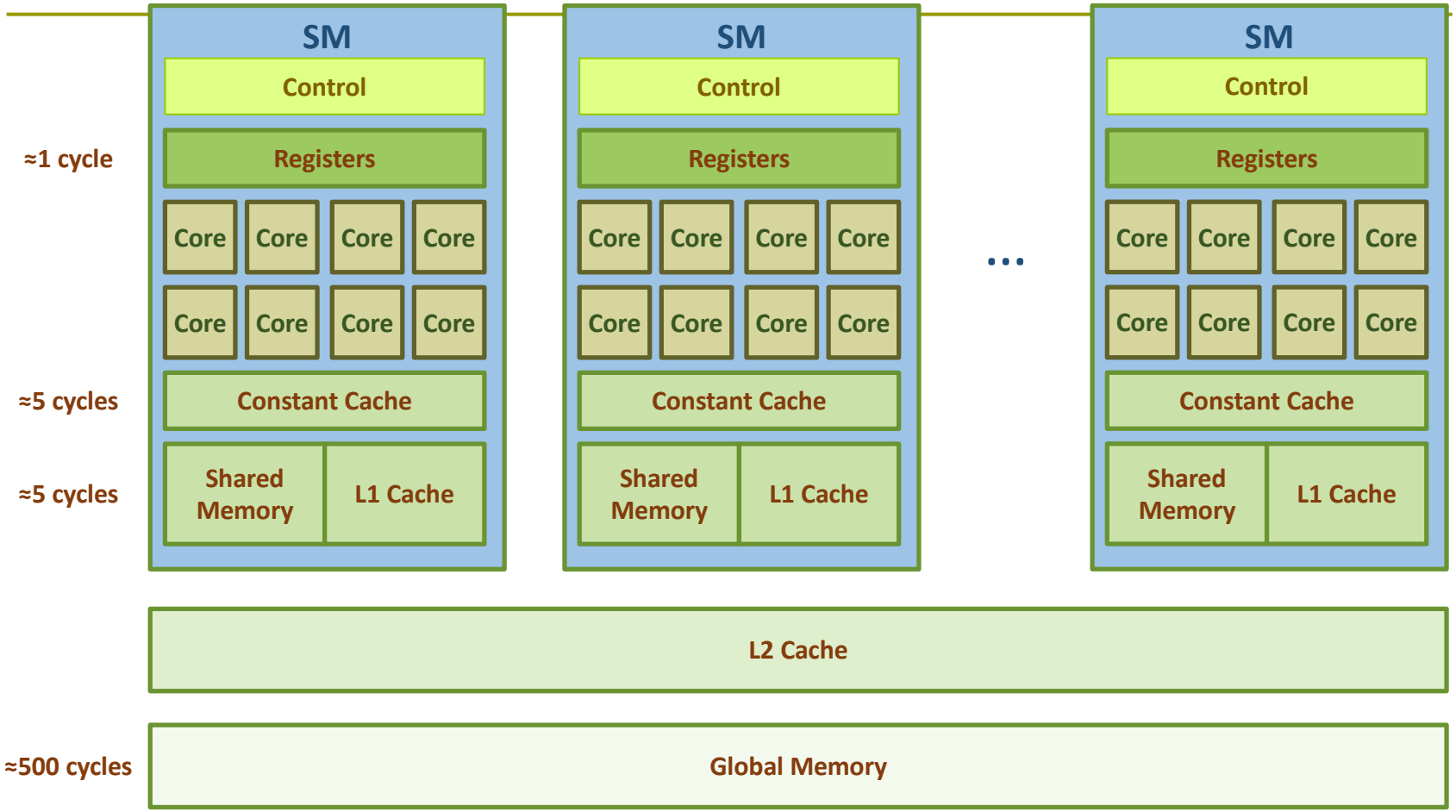
NVIDIA A100 Core

- 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/

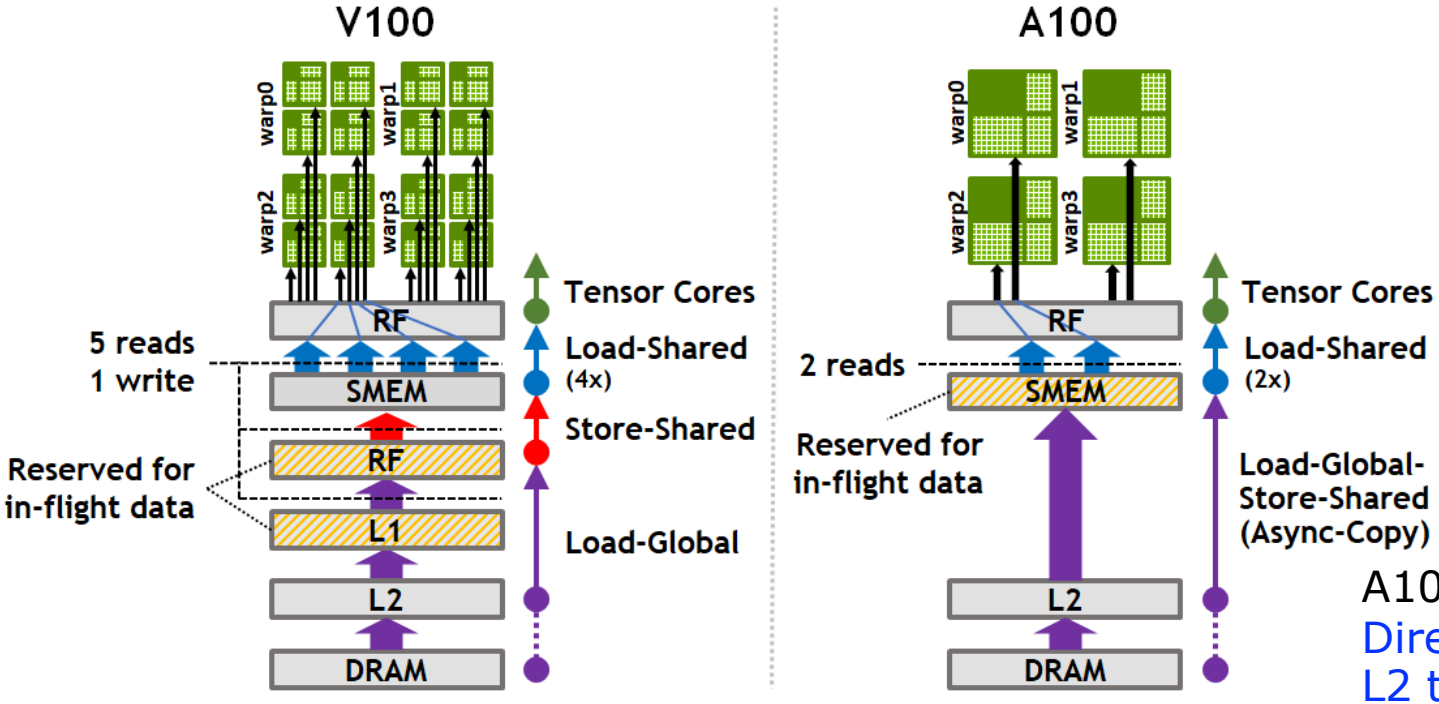
Memory in the GPU Architecture



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.

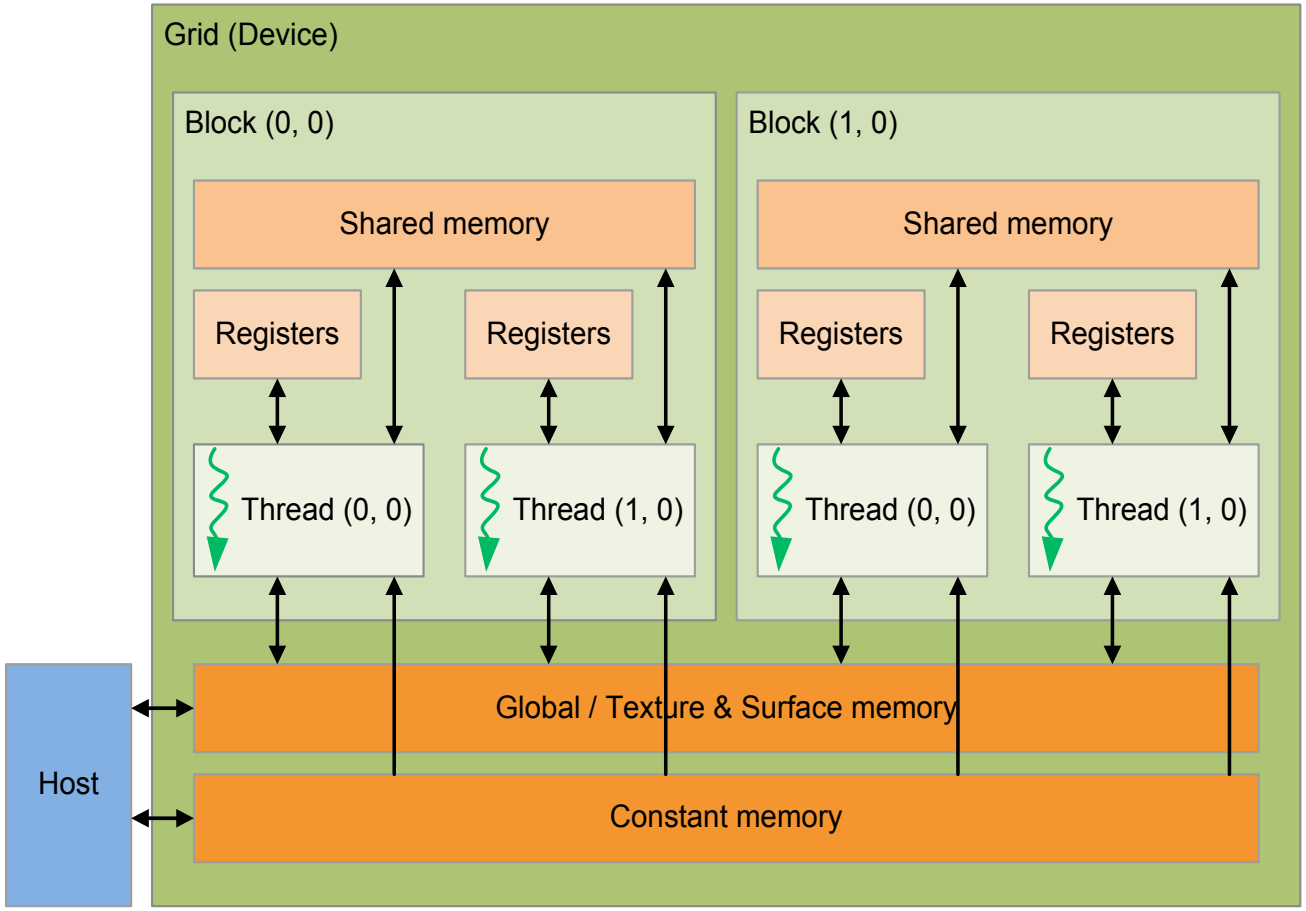
https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

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

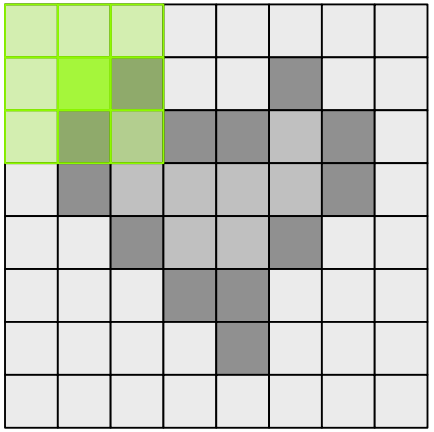
- `__device__` is optional when used with `__shared__`, or `__constant__`
- Recall `cudaMalloc(…)` allocates memory from the host
  - Constant memory can also be allocated and initialized from the host
- Automatic variables without any qualifier reside in a register
  - Except arrays that reside in global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

Memory Hierarchy in CUDA Programs



Data Reuse

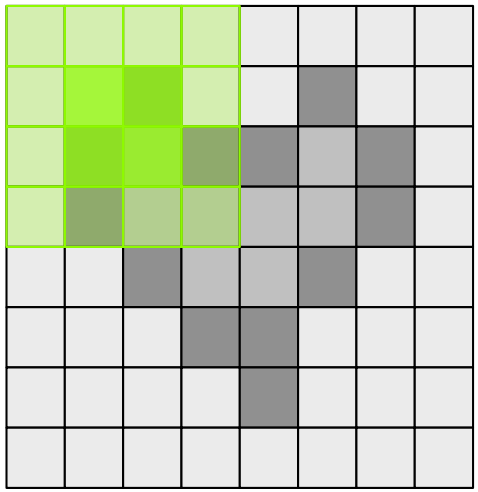
- Same memory locations accessed by neighboring threads



```
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

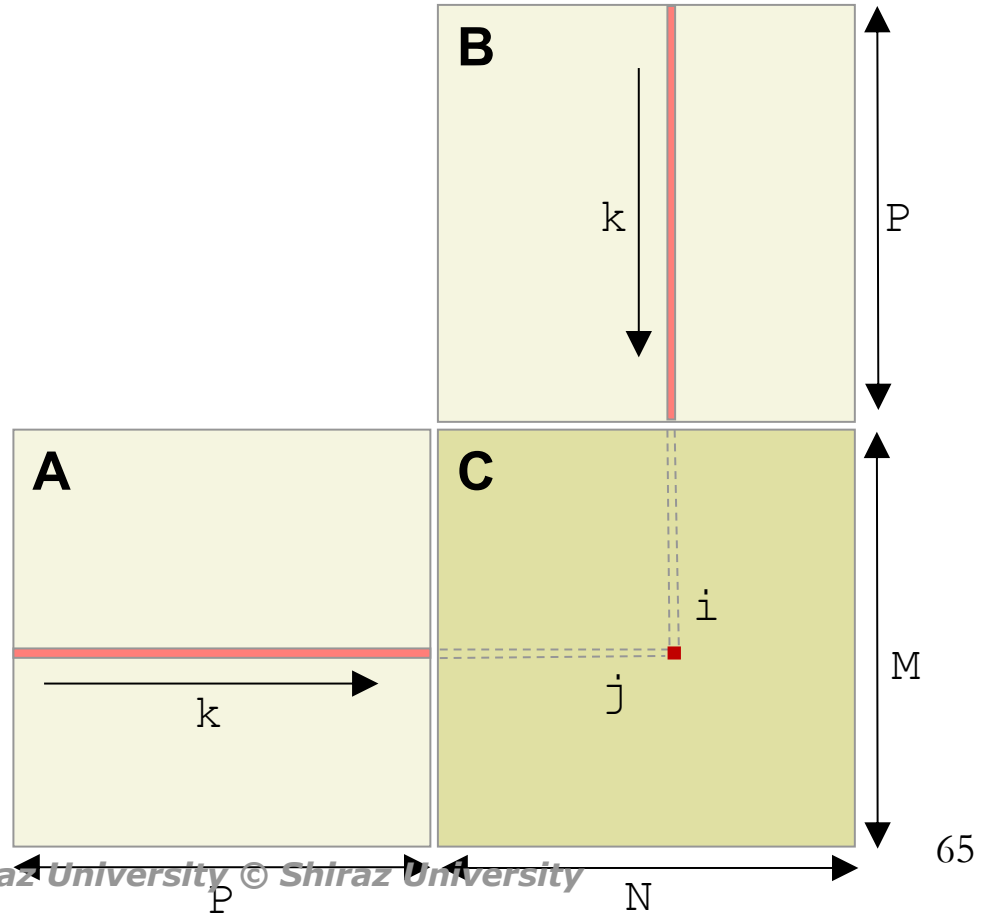
- `void __syncthreads();`
- Synchronizes all threads in a block
- Once all threads in a block have reached this point, execution resumes normally
- Used to avoid RAW / WAR / WAW hazards when accessing shared or global memory

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)

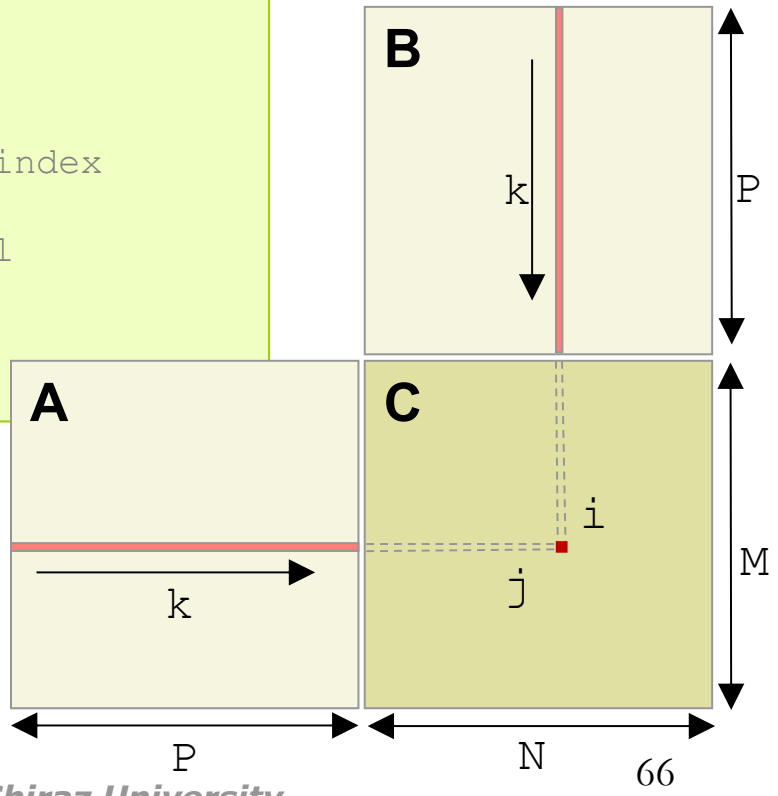
- Matrix multiplication: C = A x B
- Consider two input matrices A and B in row-major layout
  - A size is M x P
  - B size is P x N
  - C size is M x N



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);
    }
}
```

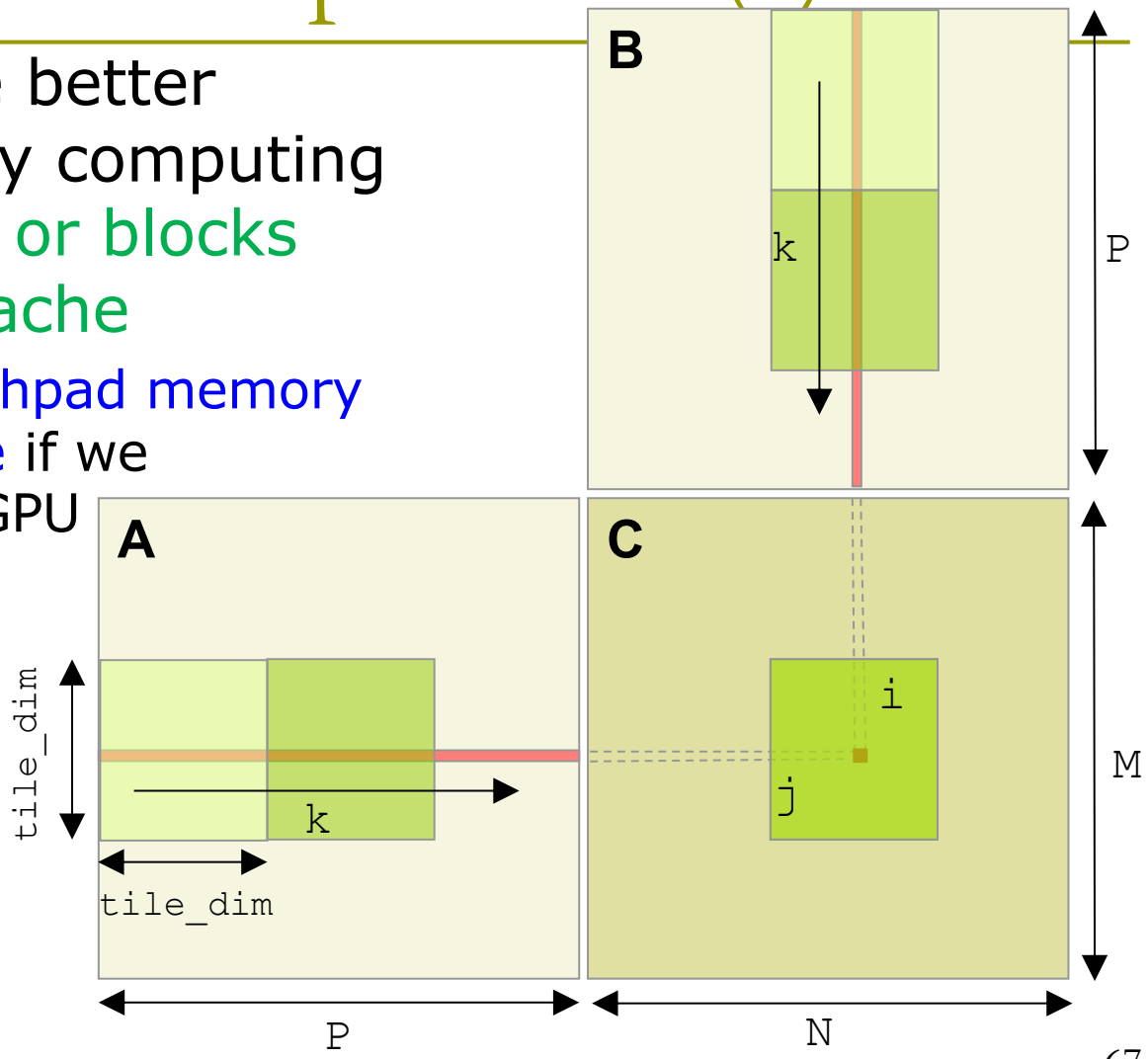


Consecutive accesses to B are far from each other, in different cache lines.

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



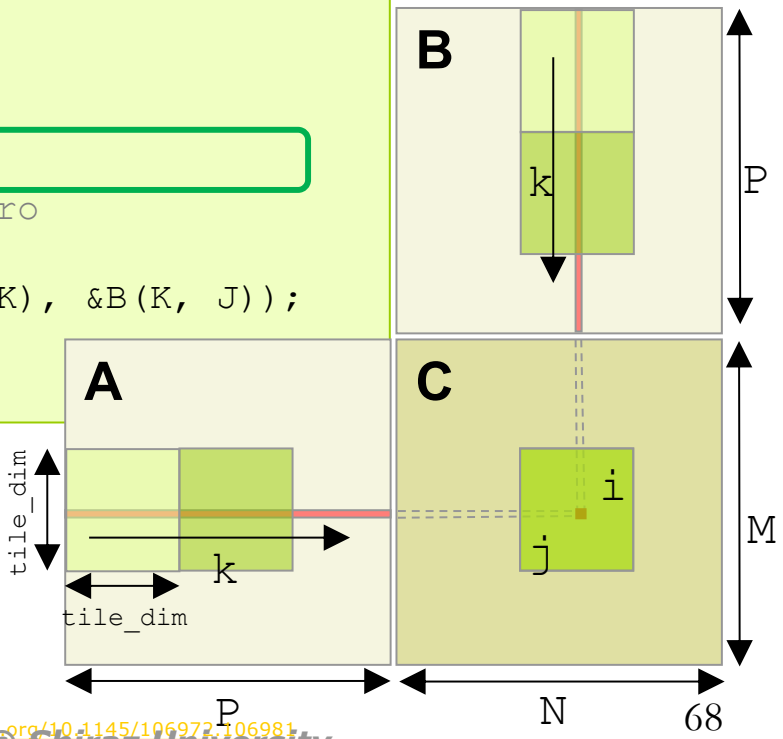
Lam+, "The cache performance and optimizations of blocked algorithms," ASPLOS 1991. https://doi.org/10.1145/106972.106981
Bansal+, "Chapter 15 - Fast Matrix Computations on Heterogeneous Streams," in "High Performance Parallelism Pearls", 2015. https://doi.org/10.1016/B978-0-12-803819-2.00011-2
Kirk & 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)

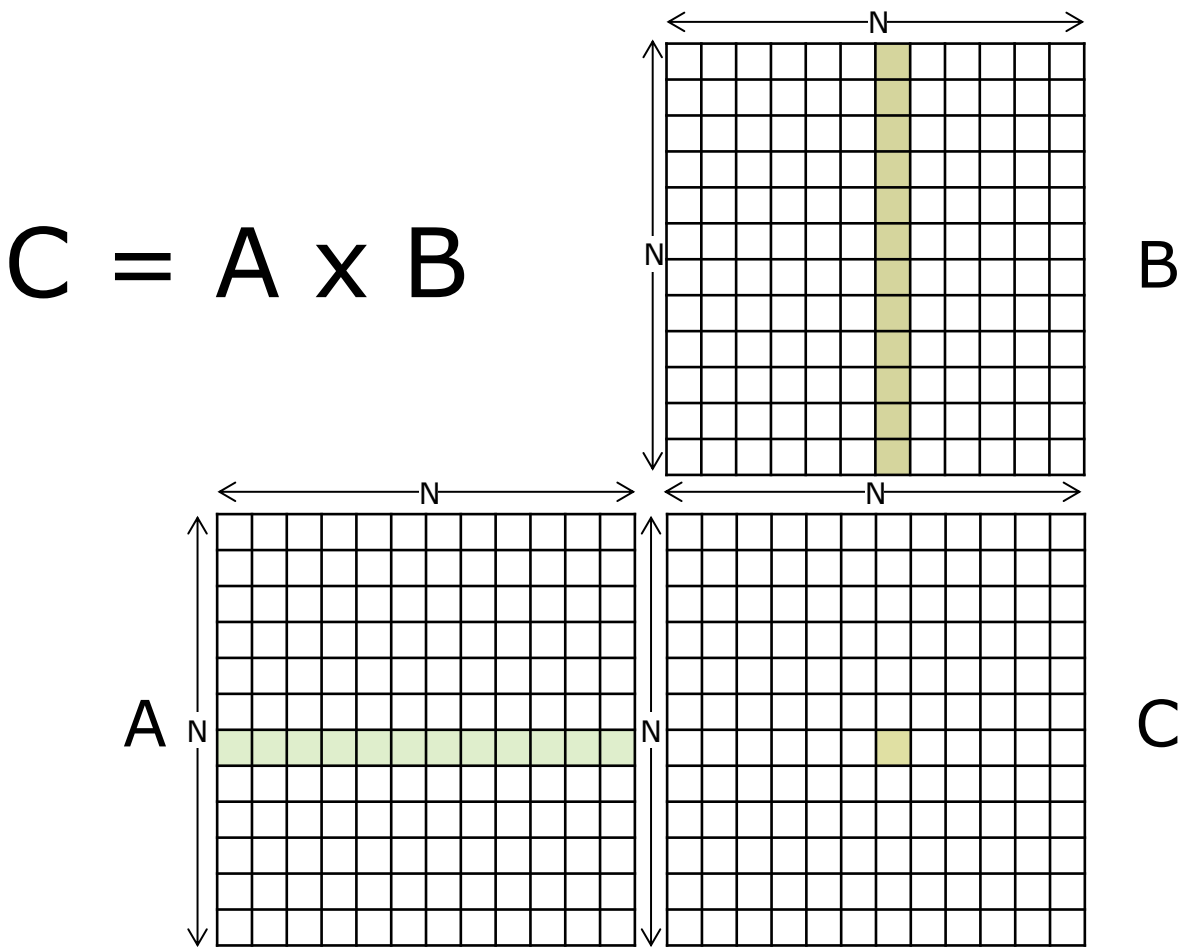
```
#define A(i,j) matrix_A[i * P + j]
#define B(i,j) matrix_B[i * N + j]
#define C(i,j) matrix_C[i * N + j]

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

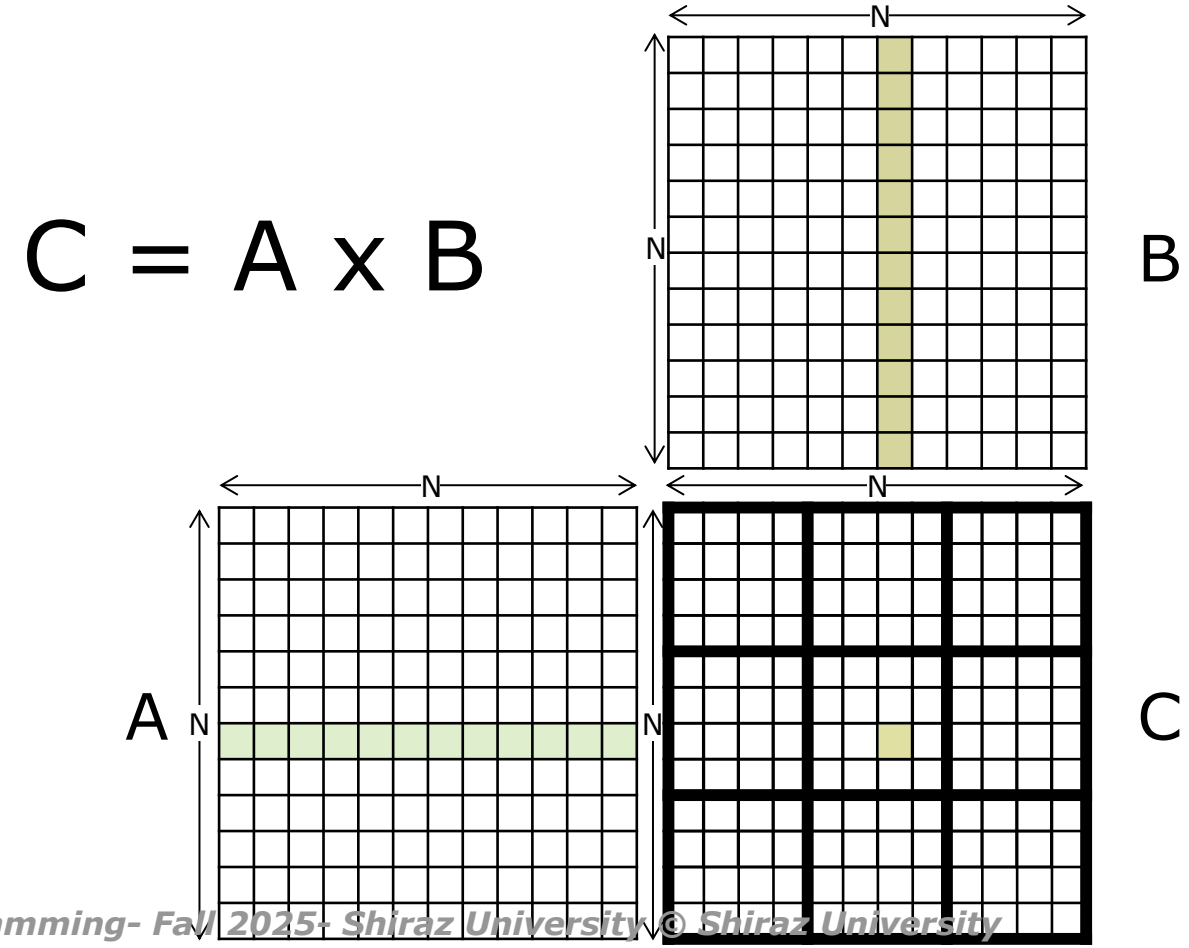
Example: Matrix-Matrix Multiplication (I)



Slide credit: Izzat El Hajj

Example: Matrix-Matrix Multiplication (II)

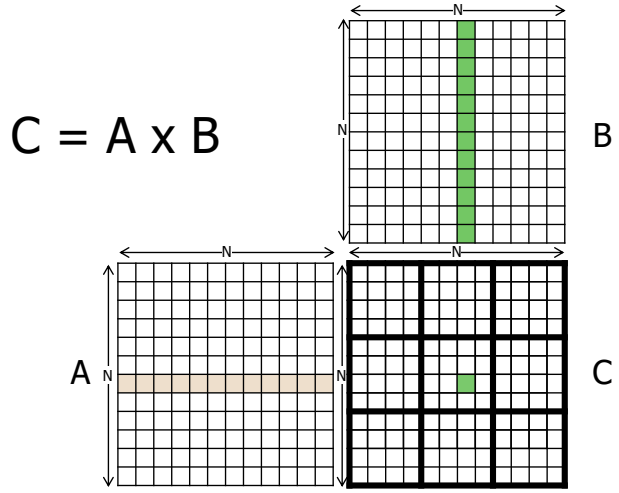
Parallelization approach: assign one thread to each element in the output matrix (C)



Slide credit: Izzat El Hajj

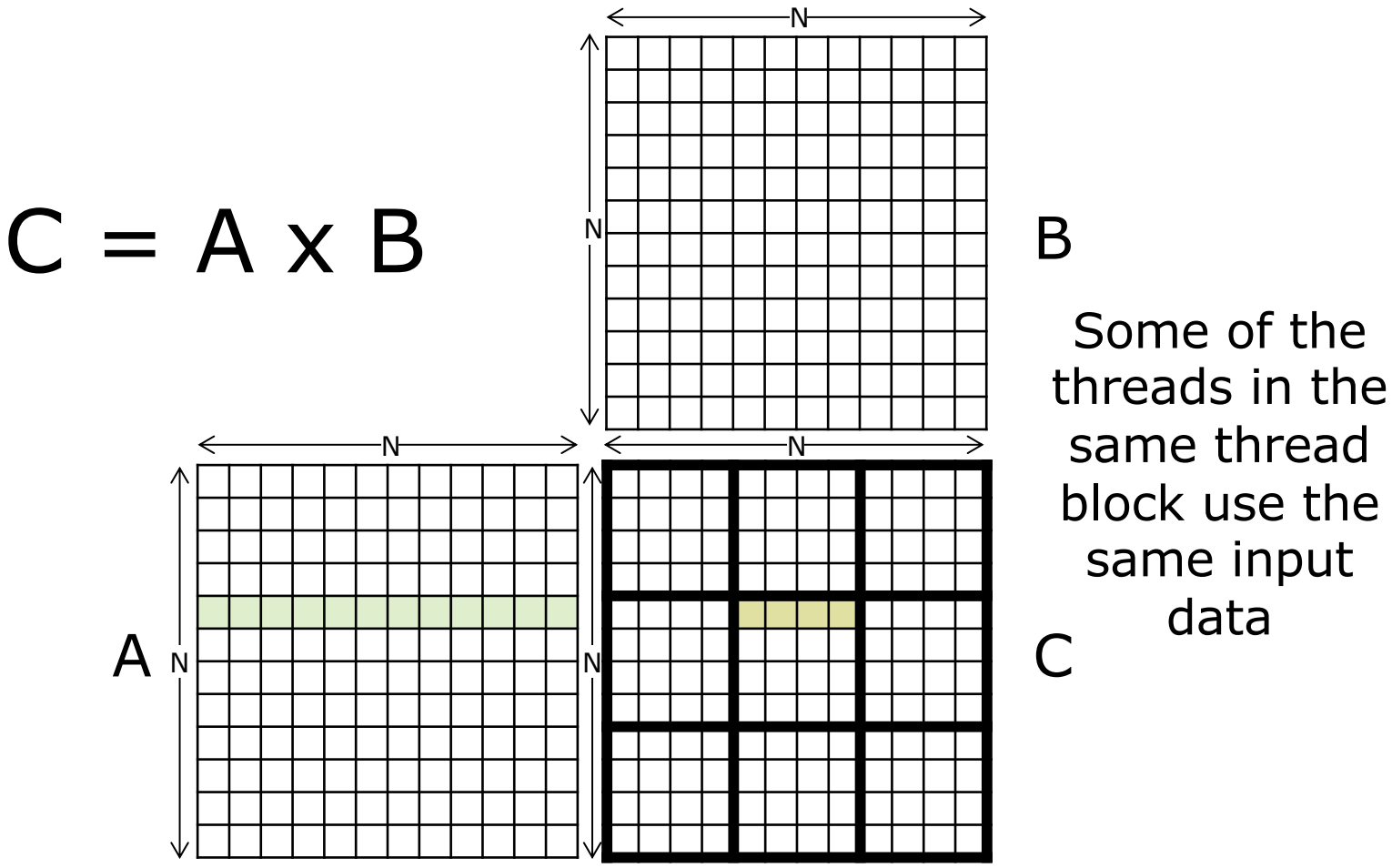
Example: Matrix-Matrix Multiplication (III)

```
__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

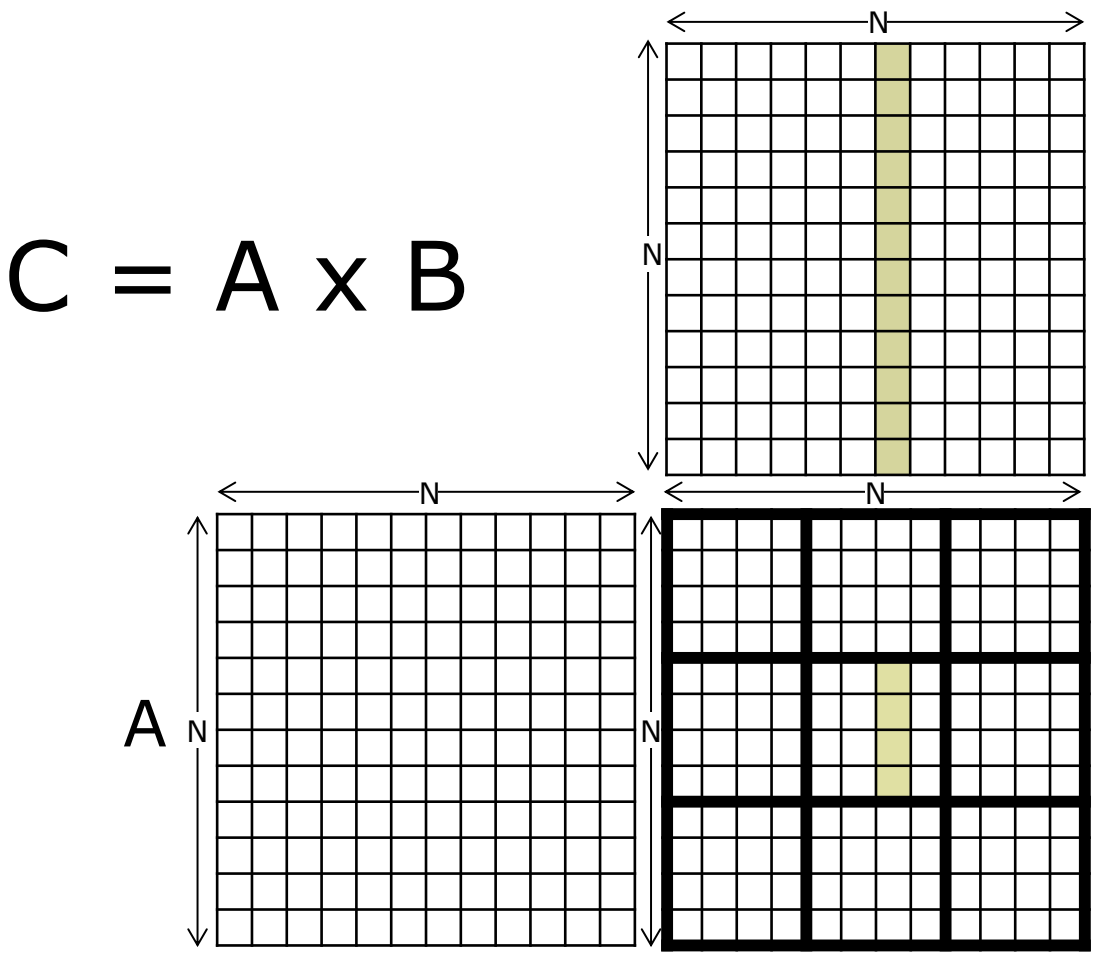
Reuse in Matrix-Matrix Multiplication (I)



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

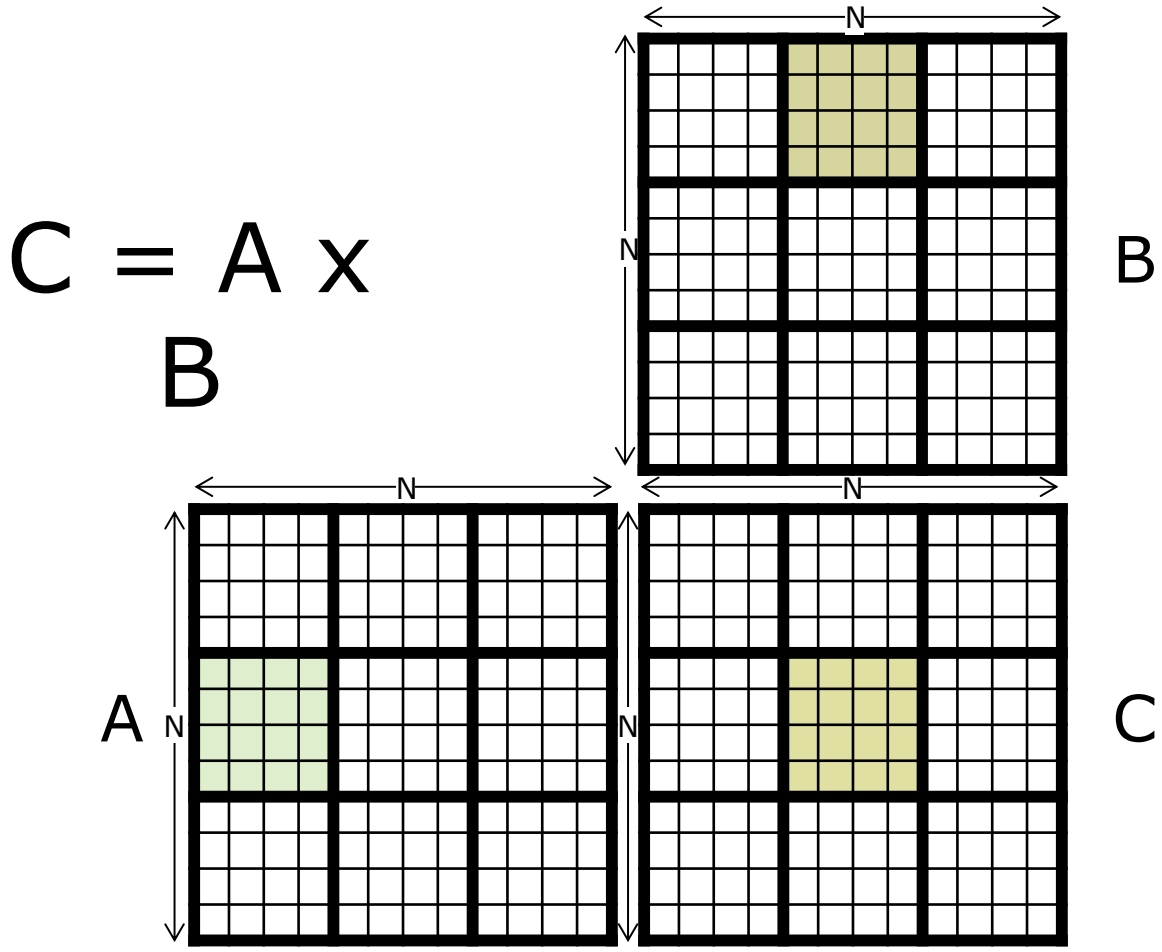
Slide credit: Izzat El Hajj

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

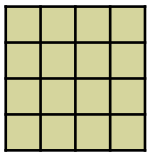
Tiled Matrix-Matrix Multiplication (I)



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

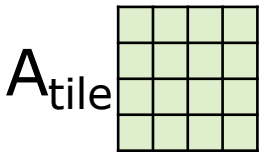
Slide credit: Izzat El Hajj

Tiled Matrix-Matrix Multiplication (II)

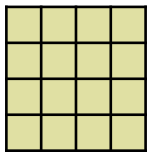


B_{tile}

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



A_{tile}

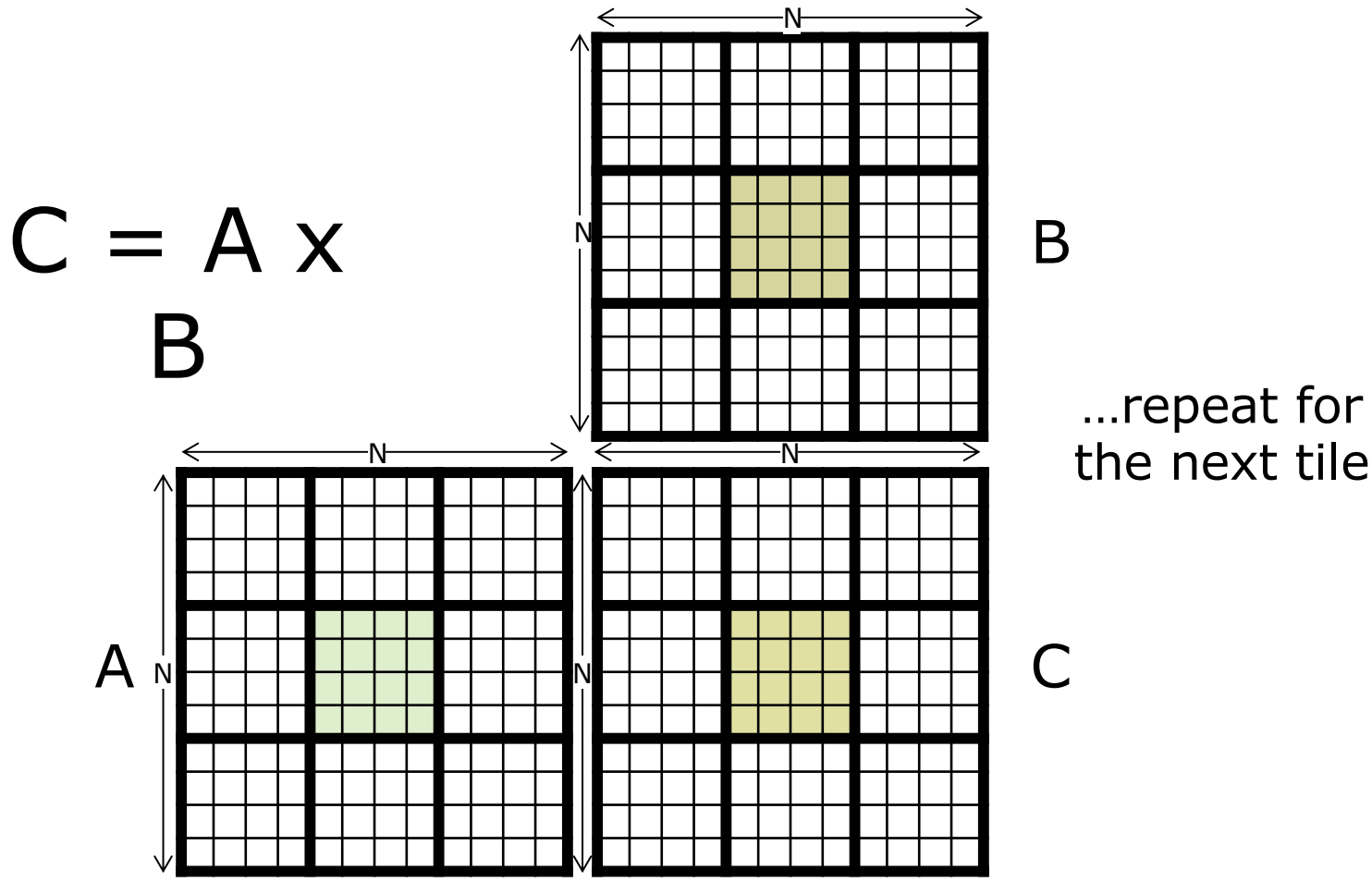


C_{tile}

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

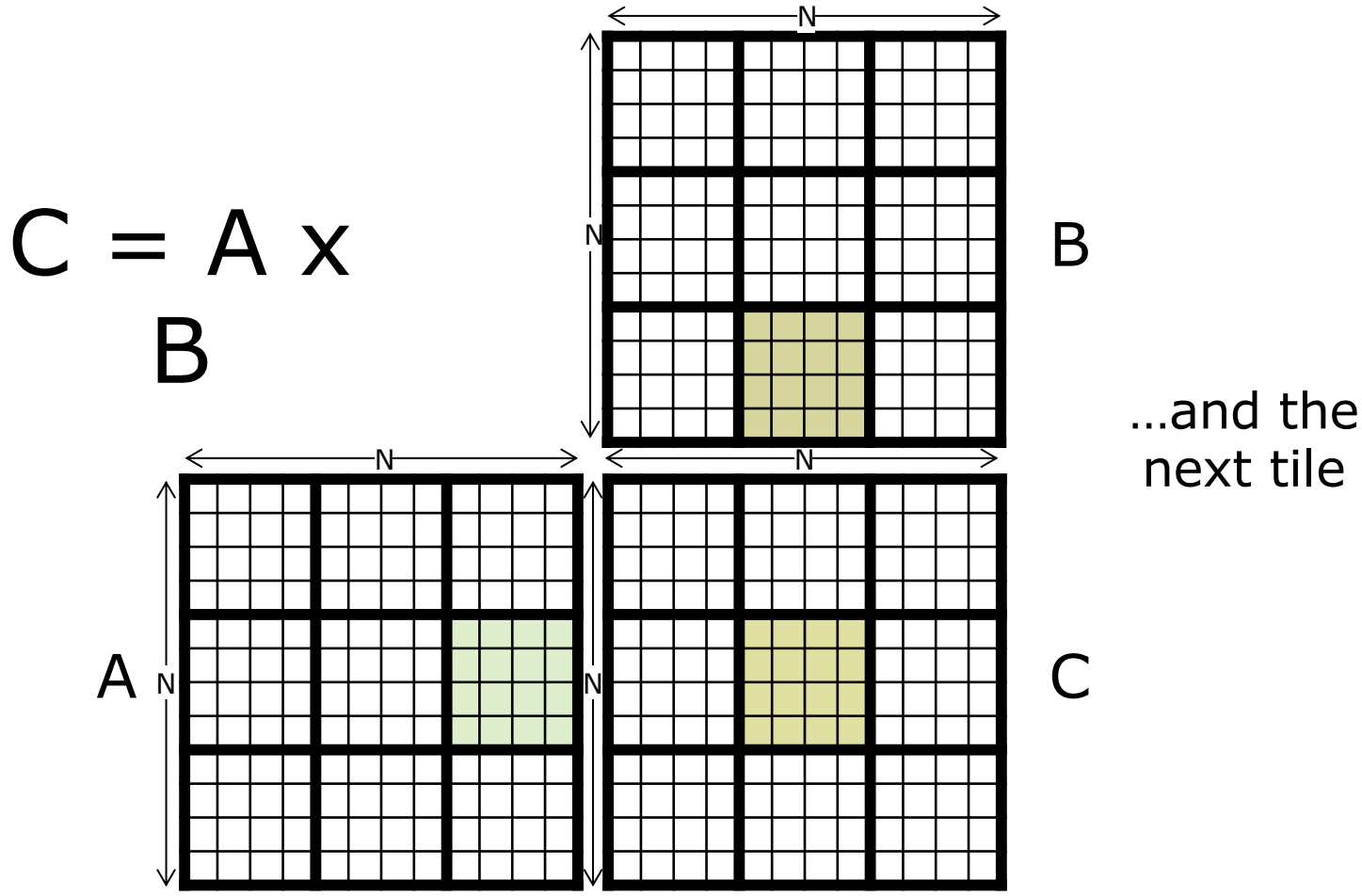
Slide credit: Izzat El Hajj

Tiled Matrix-Matrix Multiplication (III)



Slide credit: Izzat El Hajj

Tiled Matrix-Matrix Multiplication (IV)



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];

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();

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

}

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

Declare arrays in shared memory

Threads wait for each other to finish loading before computi

Threads wait for each other to finish computing before loadi

Slide credit: Izzat El Hajj

Basic Example: Matrix Multiplication using CUDA

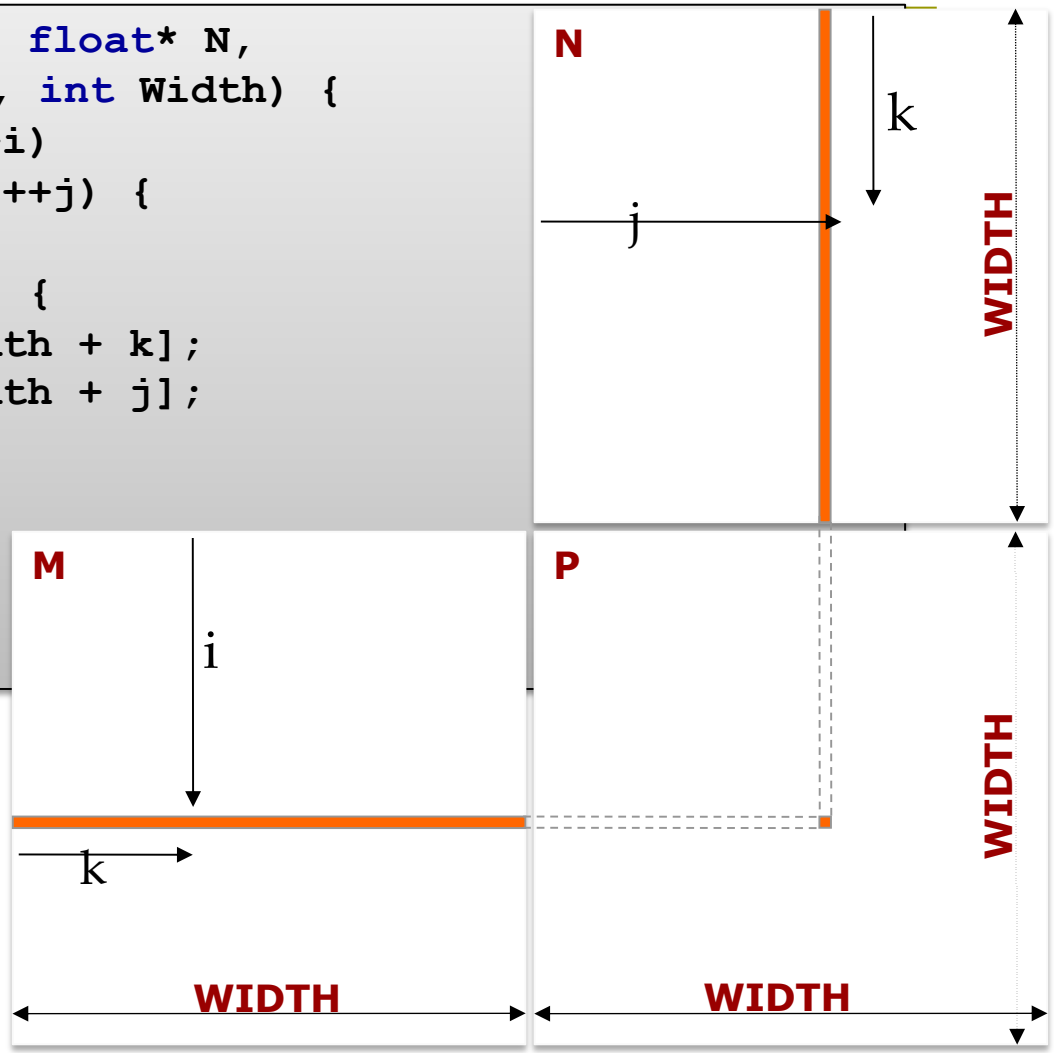
Some materials/slides are adapted from:

Andreas Moshovos' Course at the University of Toronto

UIUC course by Wen-Mei Hwu and David Kirk

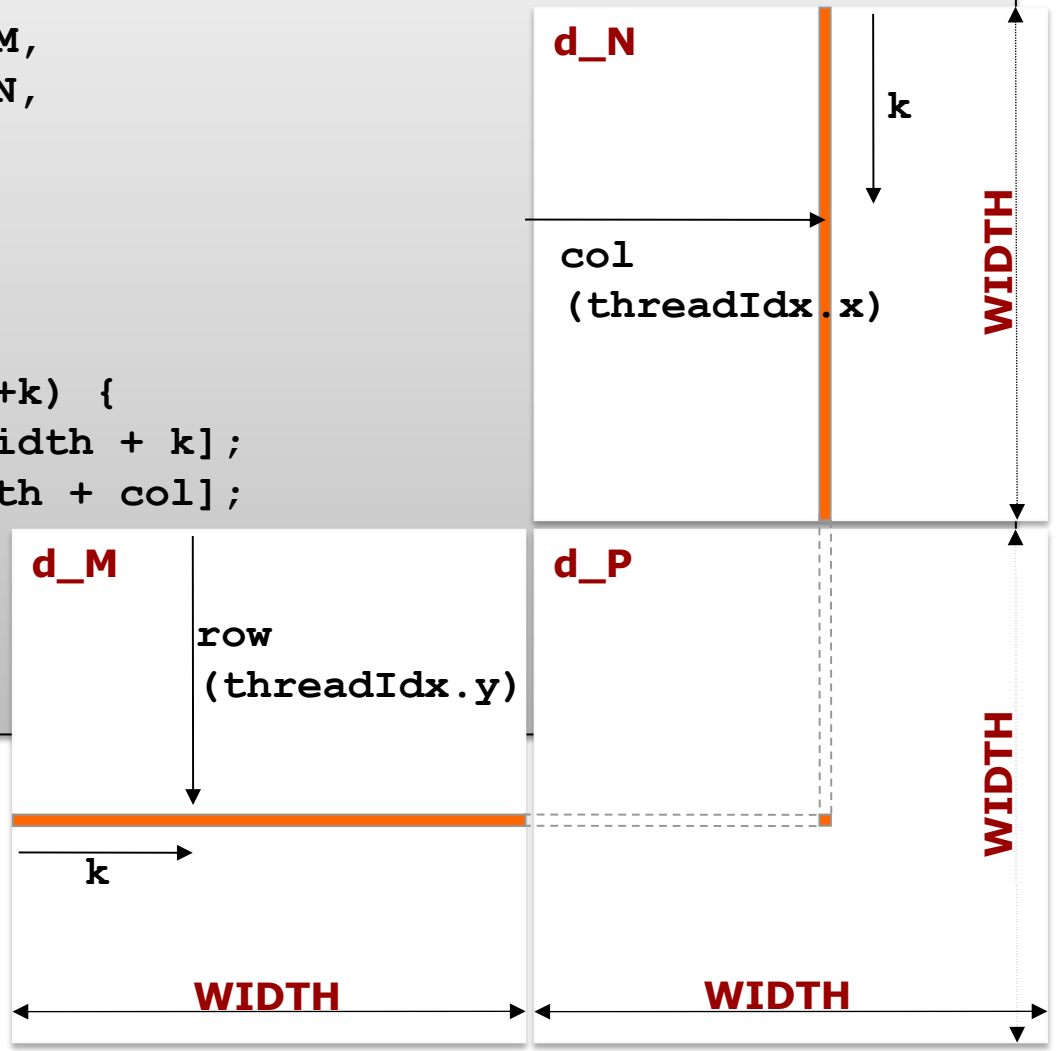
A Simple Host Version in C

```
void MatrixMulOnHost( float* M, float* N,
                      float* P, int Width) {
  for (int i = 0; i < Width; ++i)
    for (int j = 0; j < Width; ++j) {
float sum = 0;
for (int k = 0; k < Width; ++k) {
             float a = M[i * Width + k];
             float b = N[k * Width + j];
             sum += a * b;
         }
         P[i * Width + j] = sum;
    }
}
```



GPU Kernel Function

```
__global__
void MatrixMulKernel(float* d_M,
                     float* d_N,
                float* d_P,
                int Width) {
  int row = threadIdx.y;
  int col = threadIdx.x;
  float P_val = 0;
  for (int k = 0; k < Width; ++k) {
    float M_elem = d_M[row * Width + k];
    float N_elem = d_N[k * Width + col];
    P_val += M_elem * N_elem;
  }
  d_p[row*Width+col] = P_val;
}
```



Input Data Allocation and Transfer

```
void MatrixMulOnDevice(float* M,
                       float* N,
                       float* P,
                       int Width)
{
    int matrix_size = Width * Width * sizeof(float);
    float *d_M, *d_N, *d_P;

    // Allocate and Load M and N to device memory
    cudaMalloc(&d_M, matrix_size);
    cudaMemcpy(d_M, M, matrix_size, cudaMemcpyHostToDevice);

    cudaMalloc(&d_N, matrix_size);
    cudaMemcpy(d_N, N, matrix_size, cudaMemcpyHostToDevice);

    // Allocate P on the device
    cudaMalloc(&d_P, matrix_size);
```

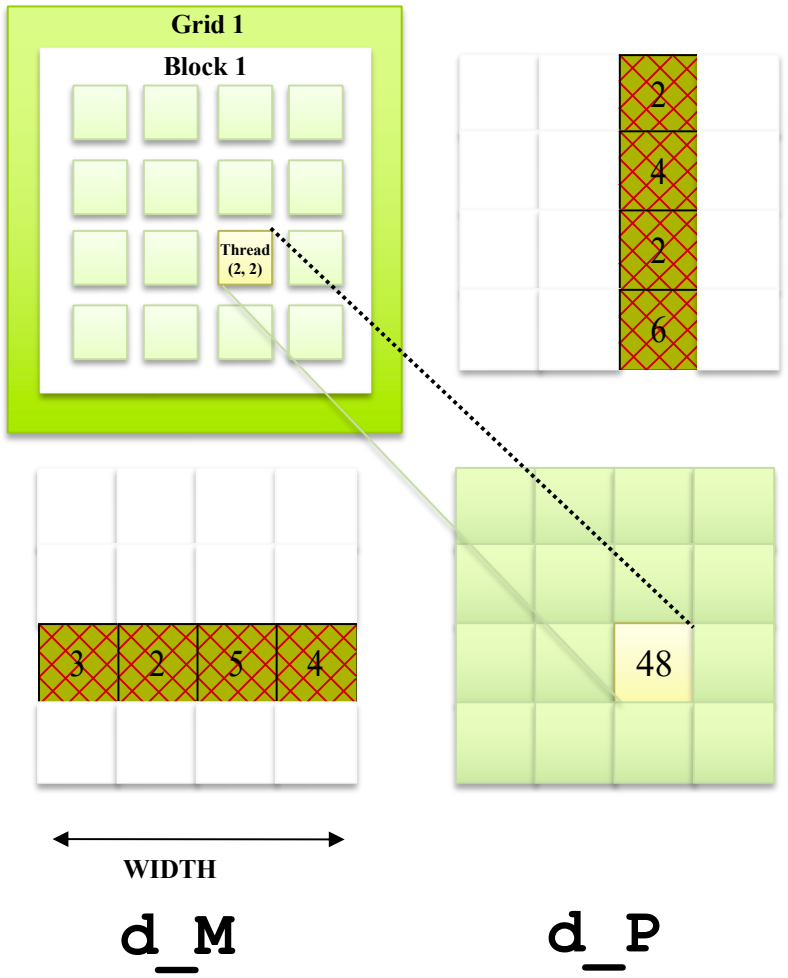
Kernel Invocation and Copy Results

```
// Setup the execution configuration
dim3 dimGrid(1, 1);
dim3 dimBlock(Width, Width);


// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>> (d_M, d_N, d_P,
                                        Width);

 // Copy back the results from device to host
cudaMemcpy(P, d_P, matrix_size, cudaMemcpyDeviceToHost);

 // Free up the device memory matrices
cudaFree(d_P);
cudaFree(d_M);
cudaFree(d_N);
}
```

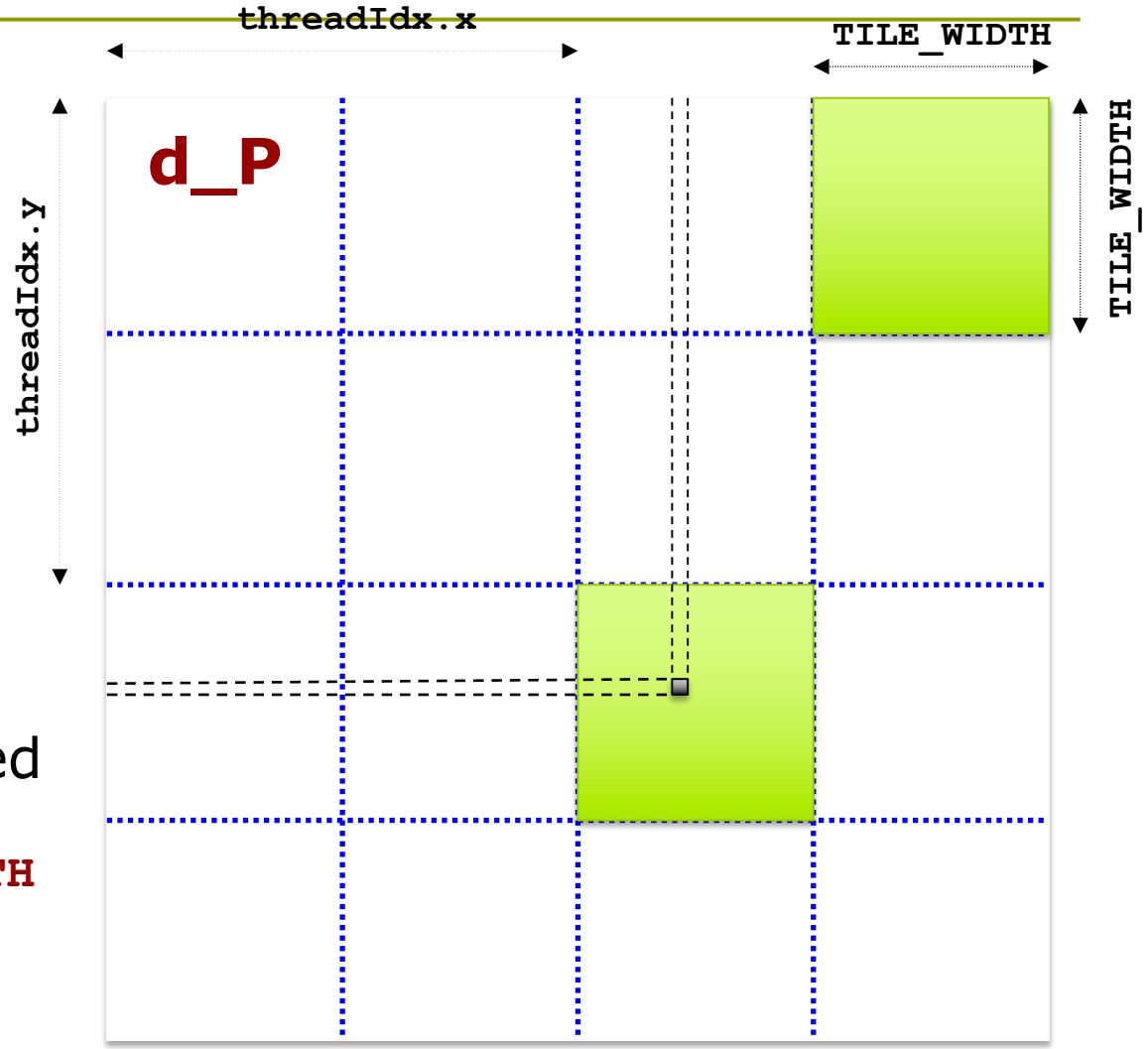
Only One Thread Block Used

- One Block of threads compute matrix `d_P`
- Each thread
  - Loads a row of matrix `d_M`
  - Loads a column of matrix `d_N`
  - Perform one multiply and addition for each pair of `d_M` and `d_N` elements
  - Computes one element of `d_P`

Size of matrix limited by the number of threads allowed in a thread block



Solution 1: Give Each Thread More Work



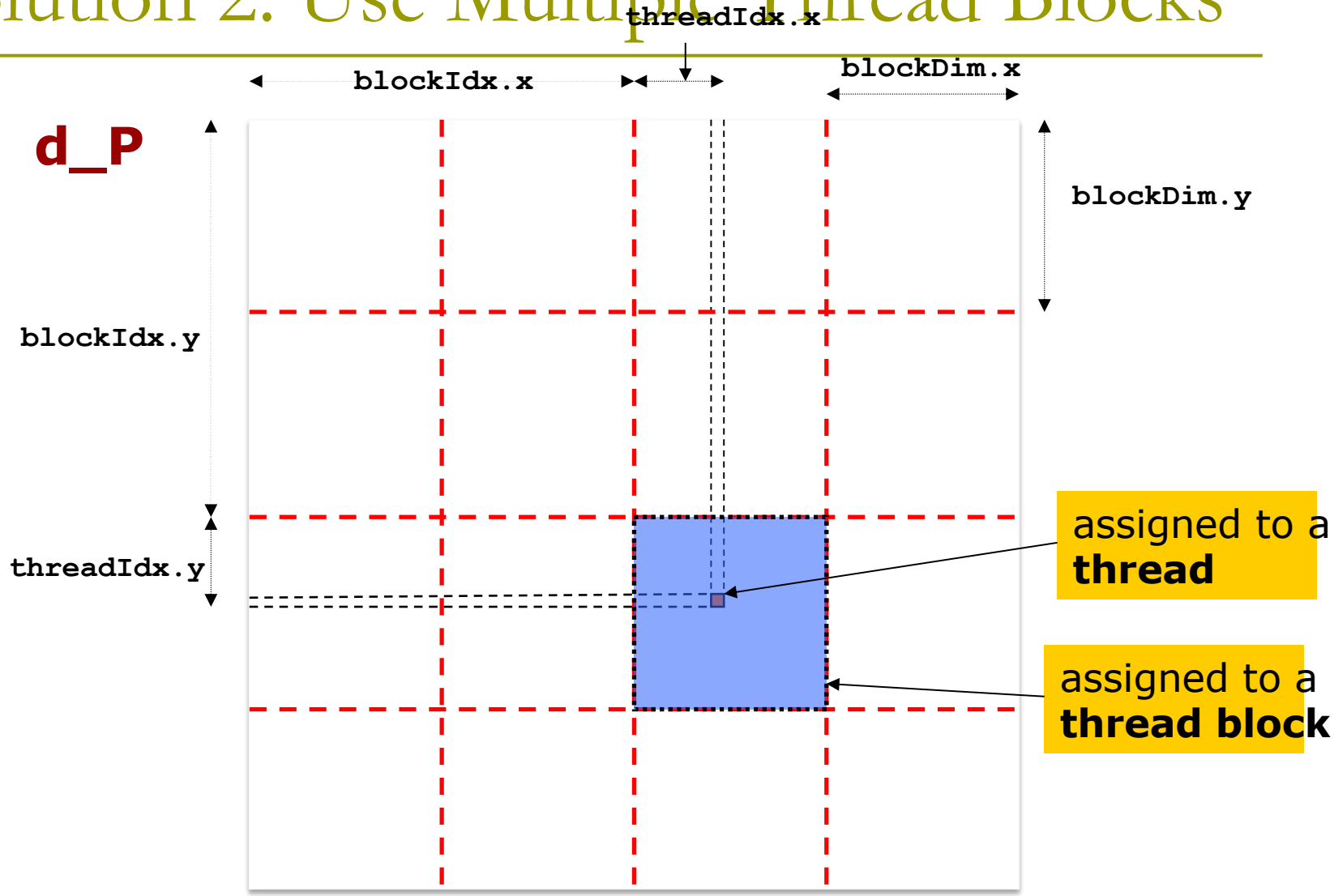
Each thread is assigned to a ***Tile*** of `TILE_WIDTH x TILE_WIDTH` Entries`

Solution 1: Give Each Thread More Work

```
__global__ void MatrixMulKernel(float* d_M,
                                float* d_N,
                          float* d_P,
                          int Width) {
  int start_row = threadIdx.y * TILE_WIDTH;
  int end_row = start_row + TILE_WIDTH;
  int start_col = threadIdx.x * TILE_WIDTH;
  int end_col = start_col + TILE_WIDTH;

  for (int row = start_row; row < end_row; row++) {
     for(int col = start_col; col < end_col; col++) {
        float P_val = 0;
        for (int k = 0; k < Width; ++k) {
           float M_elem = d_M[row * Width + k];
           float N_elem = d_N[k * Width + col];
           P_val += M_elem * N_elem;
        }
        d_p[row*Width+col] = P_val;
     }
  }
}
```

With one block we utilize only one multiprocessor!

Solution 2: Use Multiple Thread Blocks



Solution 2: Use Multiple Thread Blocks

```
__global__
void MatrixMulKernel(float* d_M,
                     float* d_N,
                float* d_P,
                int Width) {
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  float P_val = 0;

  for (int k = 0; k < Width; ++k) {
    float M_elem = d_M[row * Width + k];
    float N_elem = d_N[k * Width + col];
    P_val += M_elem * N_elem;
  }
  d_p[row*Width+col] = P_val;
}
```

Predefined Vector Data Types

- Max Vector Size

```
int3 i3;        uint3 vect;
ulong4 list;    float3 f3;
short2 s2;      char4 cs;
```

- Can be used both in host and in device code.
- Structures accessed with **.x, .y, .z, .w** fields

```
i3.x = 10;  i3.y = 20; i3.z = 5;
printf(" %f %f %f \n", f3.x, f3.y, f3.z);
```

- default constructors, "`make_TYPE (…)`":
  - `float4 f4 = make_float4 (1f, 10f, 1.2f, 0.5f);`

- **dim3**
  - type built on uint3
  - Used to specify dimensions
  - Default value is (1, 1, 1)

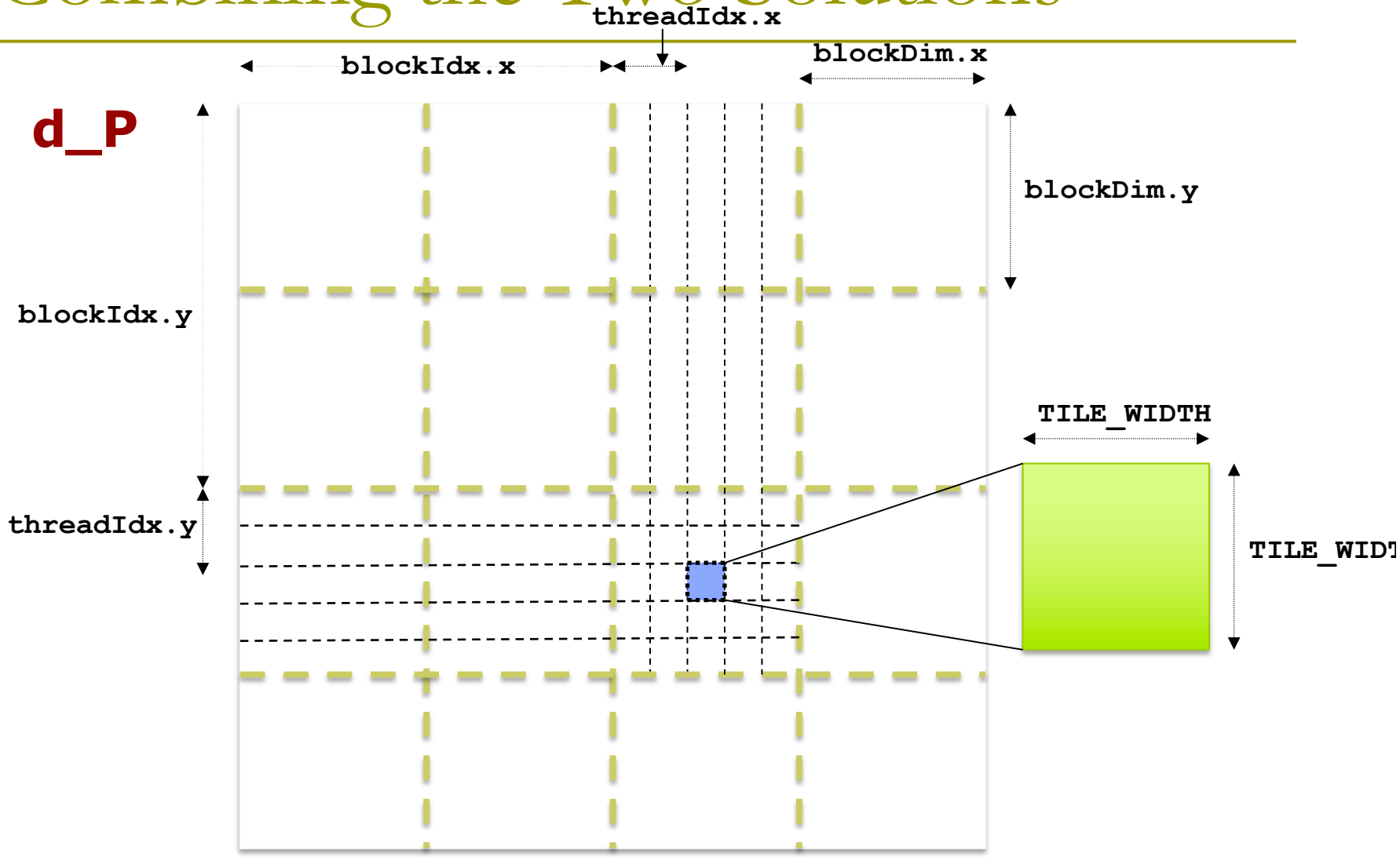
Adapted From: A. Moshovos

Kernel Invocation and Copy Results

```
    int block_size = 64;

    // Setup the execution configuration
    dim3 dimGrid(Width/block_size, Width/block_size);
    dim3 dimBlock(block_size, block_size);


    // Launch the device computation threads!
   MatrixMulKernel<<<dimGrid, dimBlock>>>(d_M, d_N, d_P,
Width);

     …
```

Size of matrix limited by the number of threads allowed on a device

Combining the Two Solutions



Combining the Two Solutions

```
__global__ void MatrixMulKernel(float* d_M,
                                float* d_N,
                          float* d_P,
                          int Width) {
  int start_row = blockDim.y * blockIdx.y + threadIdx.y * TILE_WIDTH;
  int end_row = start_row + TILE_WIDTH;
  int start_col = blockDim.x * blockIdx.x + threadIdx.x * TILE_WIDTH;
  int end_col = start_col + TILE_WIDTH;

  for (int row = start_row; row < end_row; row++) {
     for(int col = start_col; col < end_col; col++) {
        float P_val = 0;
        for (int k = 0; k < Width; ++k) {
           float M_elem = d_M[row * Width + k];
           float N_elem = d_N[k * Width + col];
           P_val += M_elem * N_elem;
        }
        d_p[row*Width+col] = P_val;
     }
  }
}
```

Execution Configuration

- Must specify when calling a `__global__` function:
  `<<< Dg, Db [, Ns [, S]] >>>`
- **`dim3 Dg`**: grid dimensions in blocks
- **`dim3 Db`**: block dimensions in threads
- **`size_t Ns:`** per block additional number of shared memory bytes to allocate
  - optional, defaults to 0
- **`cudaStream_t S`**: request stream(queue)
  - optional, default to 0.
  - Used to have multiple kernels active simultaneously
  - Compute capability >= 1.1

Adapted From: A. Moshovos

Built-in Variables

- **`dim3 gridDim`**
  - Number of blocks per grid, in 2D (.z always 1)
- **`uint3 blockIdx`**
  - Block ID, in 2D (blockIdx.z = 1 always)
- **`dim3 blockDim`**
  - Number of threads per block, in 3D
- **`uint3 threadIdx`**
  - Thread ID in block, in 3D

Execution Configuration Examples

- 1D grid / 1D blocks

```
dim3 gd(1024)
dim3 bd(64)
akernel<<<gd, bd>>>(...)
```

```
gridDim.x = 1024, gridDim.y = 1,
blockDim.x = 64, blockDim.y = 1, blockDim.z = 1
```

- 2D grid / 3D blocks

```
dim3 gd(4, 128)
dim3 bd(64, 16, 4)
akernel<<<gd, bd>>>(...)
```

```
gridDim.x = 4, gridDim.y = 128,
blockDim.x = 64, blockDim.y = 16, blockDim.z = 4
```

Error Handling

- Most cuda…() functions return a **`cudaError_t`**
  - If **`cudaSuccess`**: Request completed without a problem

- **`cudaGetLastError()`**:
  - returns the last error to the CPU
  - Use with **`cudaThreadSynchronize()`**:

```
cudaError_t code;
cudaThreadSynchronize ();
code = cudaGetLastError ();
```

- **`char *cudaGetErrorString(cudaError_t code);`**
  - returns a human-readable description of the error code

Adapted From: A. Moshovos

Error Handling Utility Function

```
void
cudaDie (const char *msg)
{
  cudaError_t err;
  cudaThreadSynchronize ();
  err = cudaGetLastError();

  if (err == cudaSuccess) return;
  fprintf (stderr, "CUDA error: %s: %s.\n",
           msg,
           cudaGetErrorString (err));
  exit(EXIT_FAILURE);
}
```

adapted from: http://www.ddj.com/hpc-high-performance-computing/207603131

CUDA_SAFE_CALL

- A proprocessor macro

  CUDA_SAFE_CALL (*cuda call*)

- Example:

```
CUDA_SAFE_CALL (cudaMemcpy (a_h, a_d, arr_size,
cudaMemcpyDeviceToHost) );
```

- Must define **#define _DEBUG**
  - No code emitted when undefined: Performance
- Use **make dbg=1** under NVIDIA_CUDA_SDK

Adapted From: A. Moshovos

Measuring Time

- On Host
  - `gettimeofday`
  - CUDA Timers
- On Device
  - GPU Clock

UNIX/Linux **gettimeofday**

```
#include <sys/time.h>
#include <time.h>

struct timeval start, end;
unsigned long cpu_time;

gettimeofday (&start, NULL);

THE CODE WE'RE INTERSTED IN

gettimeofday (&end, NULL);


cpu_time = (end.tv_sec - start.tv_sec) * 1000000 +
              (end.tv_usec - start.tv_usec);
```

CUDA Timer Utility

```
#include <cuda.h>
#include <cutil.h>

unsigned int htimer;
float start, end;

cutCreateTimer (&htimer);

start = cutGetTimerValue(htimer);
cutStartTimer(htimer);

WHAT WE ARE INTERESTED IN
cudaThreadSynchronize();

cutStopTimer(htimer);
end = cutGetTimerValue(htimer);

printf ("execution time in millisecond: %f\n", end-start);
```

Using CUDA clock ()

- Can be used in device code
- Returns per multiprocessor counter which is incremented every clock cycle
- Note: threads are time-sliced
- Using clock():
  - Every thread measures start and end
  - Then must find min start and max end
  - Accurate
- Take a look at the clock example in the CUDA SDK

```
uint start, end;

start = clock();
kernel computation
end = clock();

if (end > start)
    time = end - start;
else
    time = end + (0xffffffff - start)
```

Adapted From: A. Moshovos

Measurement Methodology

- You will not get exactly the same time measurements every time
  - Other processes running / external events (e.g., network activity)
  - Cannot control
  - "Non-determinism"
- Must take sufficient samples
  - say 10 or more
  - There is theory on what the number of samples must be
- Measure average

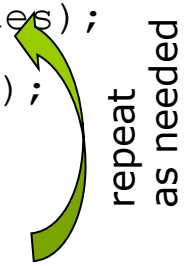
Adapted From: A. Moshovos

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, …);`



- 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

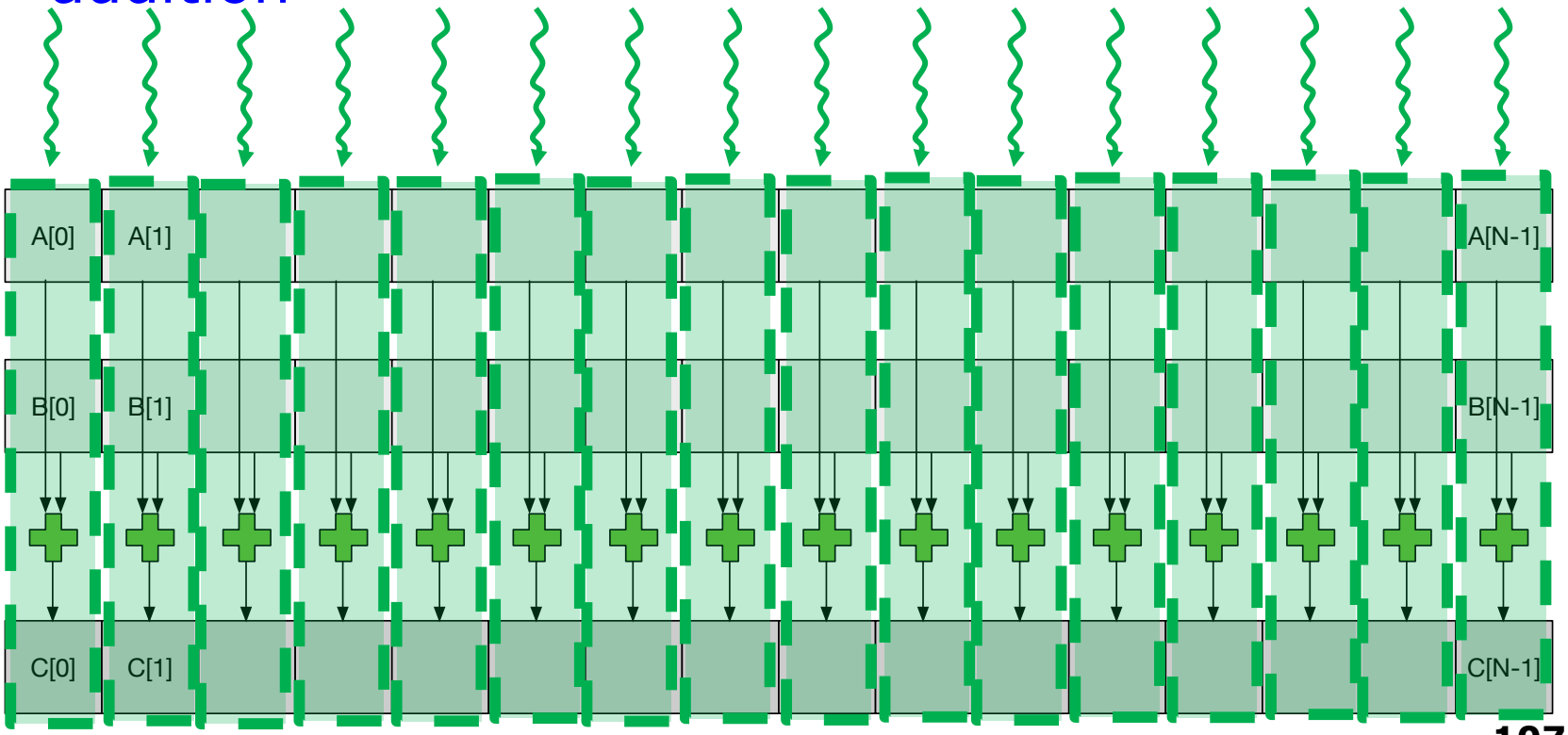
```
cudaFree(d_in);
```

- Explicit synchronization

```
cudaDeviceSynchronize();
```

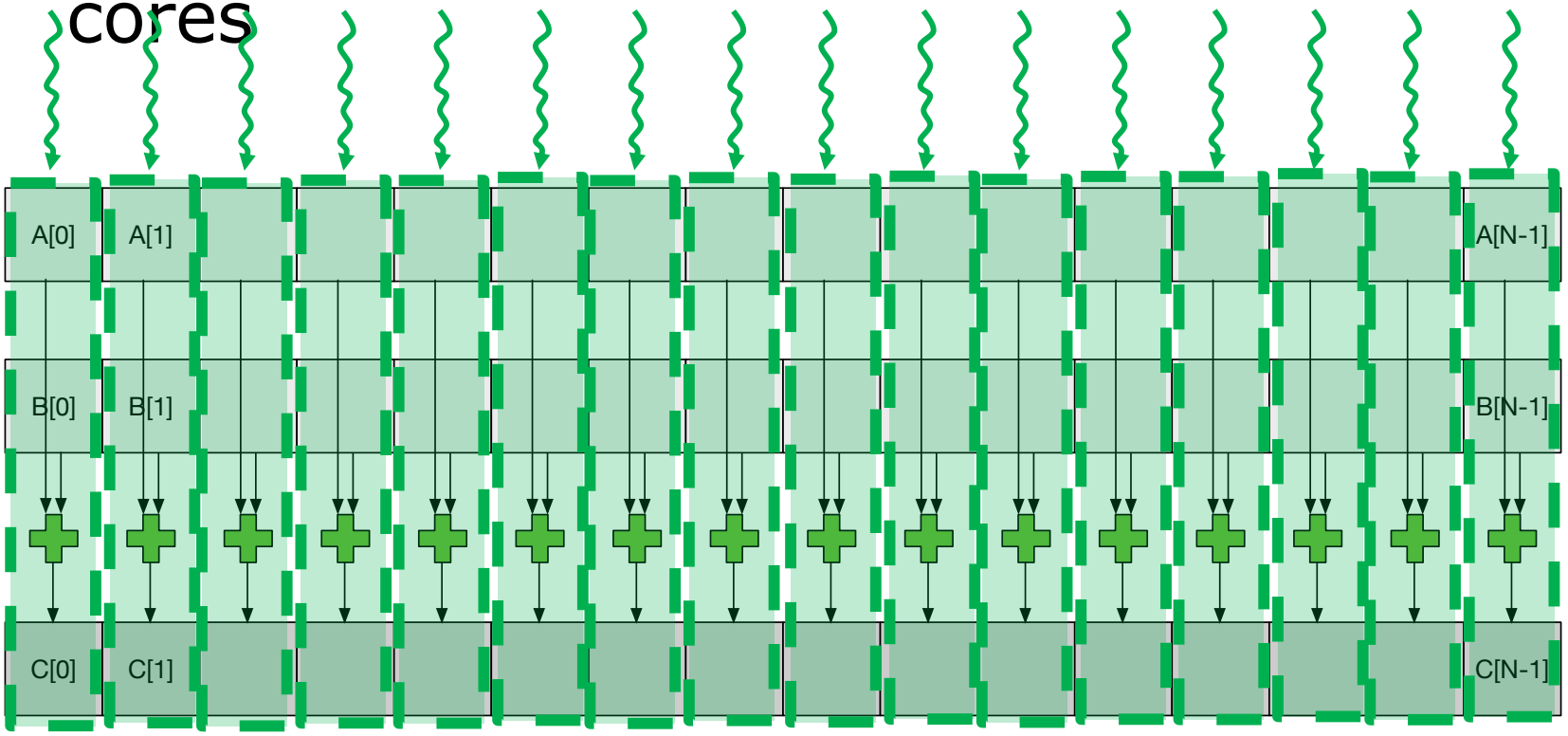
Vector Addition (I)

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



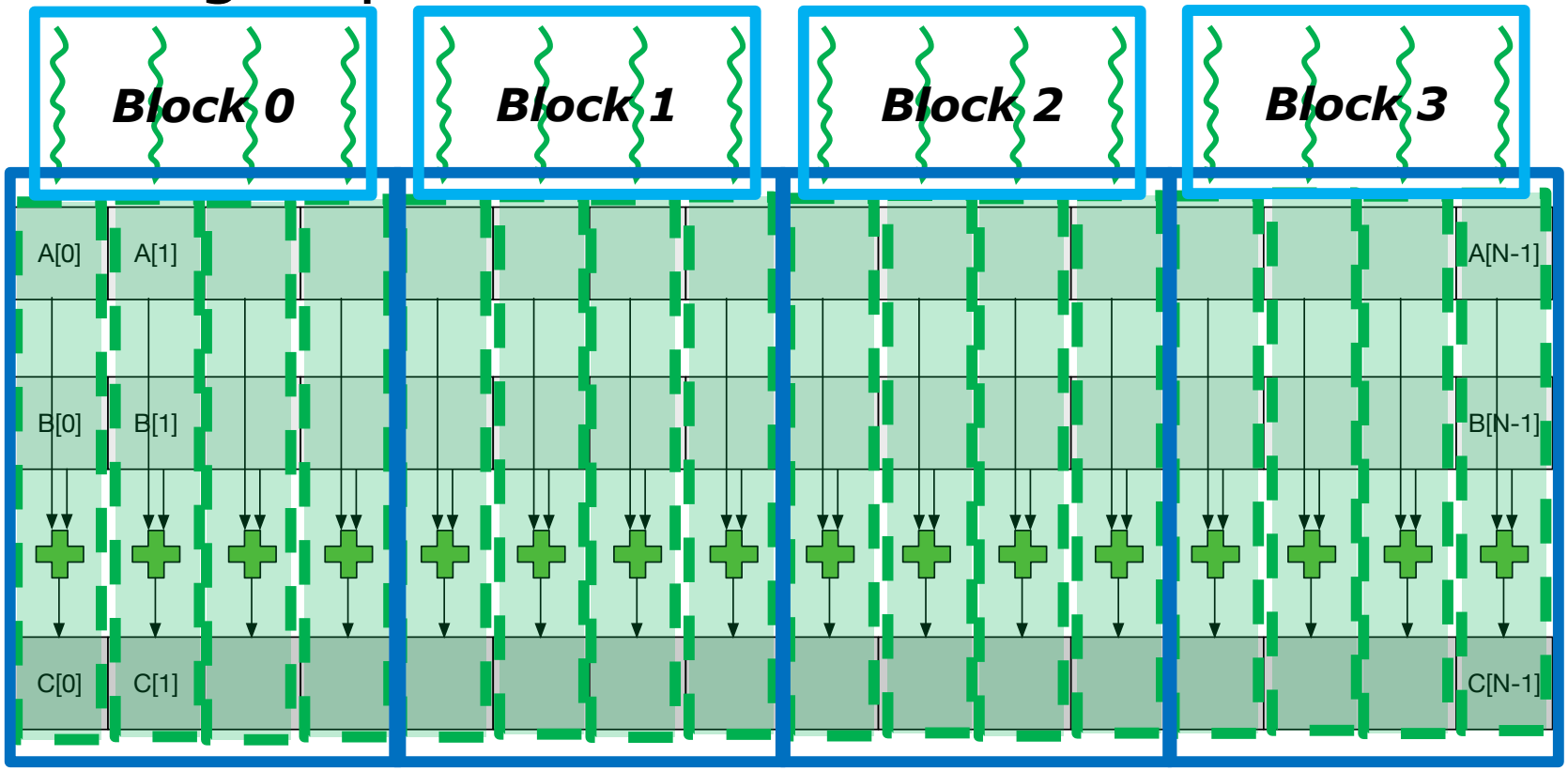
Vector Addition (II)

- 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
- Kernel code

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

__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

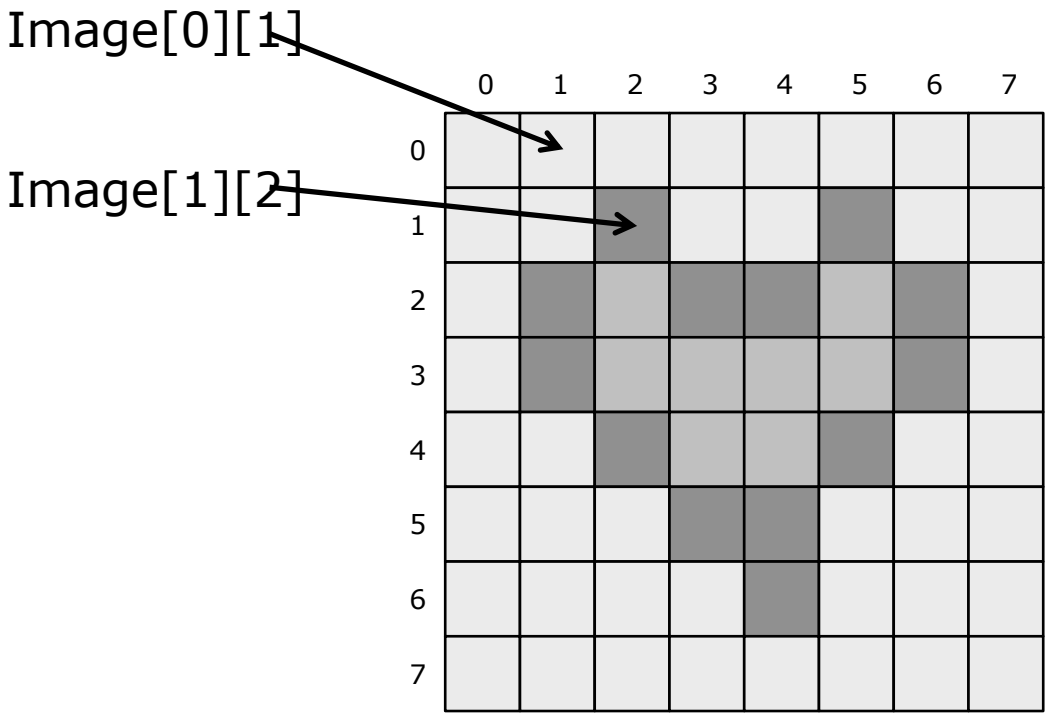
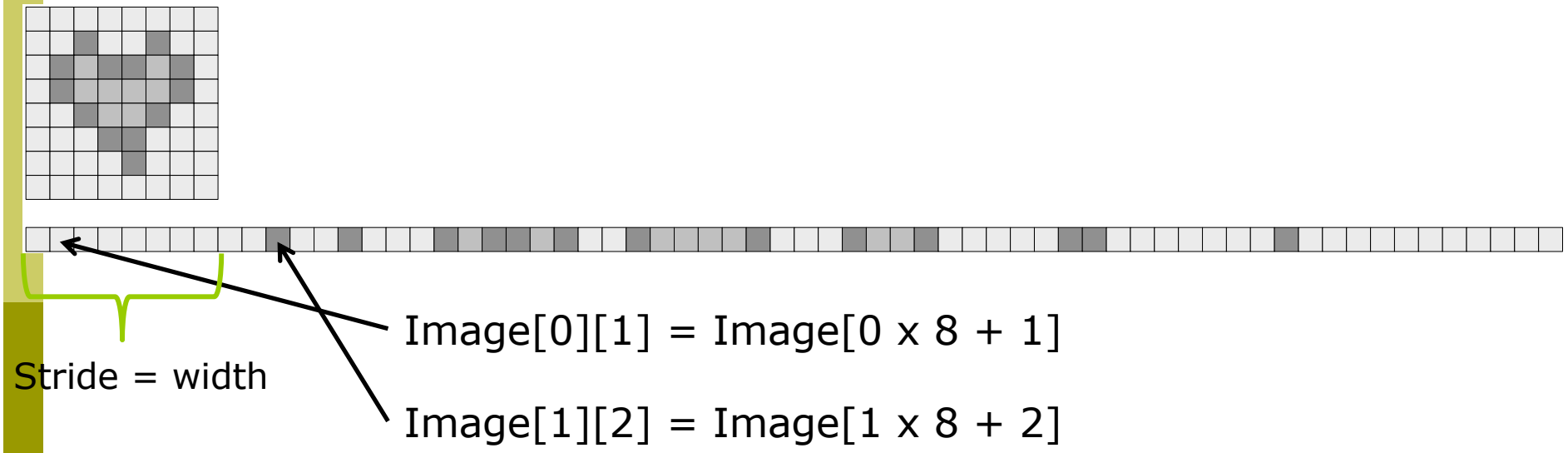


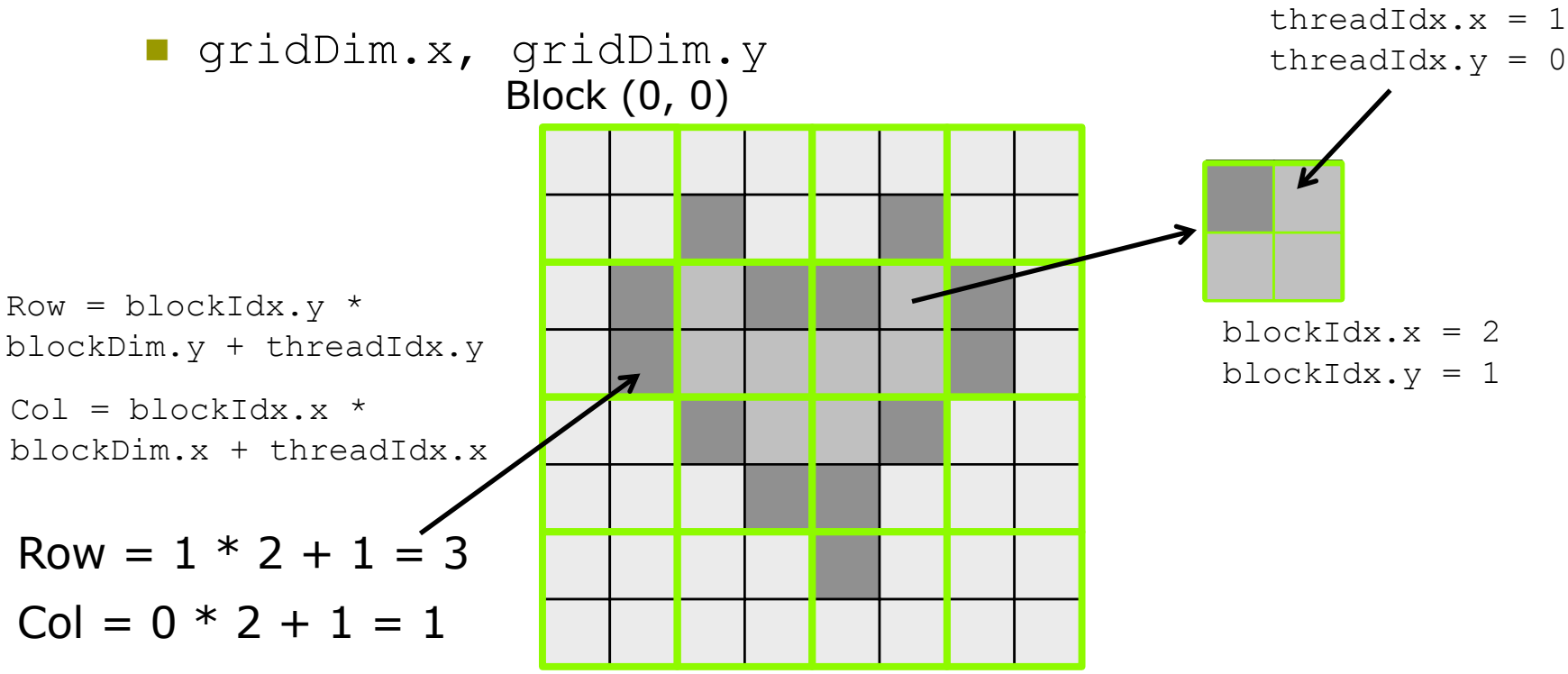
Image Layout in Memory

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



Indexing and Memory Access: 2D Grid

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



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