GPU - Graphics Processing Units

Intensive Computation

Annalisa Massini
2019-2020

Lecture 19

Programming Massively Parallel Processors

D.B. Kirk W. W. Hwu

- Chapter 3 - Introduction to Data Parallelism and CUDA C
  - **Sections 3.2 - 3.6**
- Chapter 4 - Data Parallel Execution Model
  - **Sections 4.5 - 4.7**
- Chapter 5 - CUDA Memories
  - **Sections 5.2 - 5.4**

Multicore and GPU Programming

G. Barlas

- Chapter 6 - GPU Programming
  - **Sections 6.2 - 6.7**

Computer Architecture - A Quantitative Approach, Fifth Edition

Hennessy Patterson

- Chapter 4 - Data-Level Parallelism in Vector, SIMD, and GPU Architectures
  - **Section 4.4 – Graphics Processing Units**

Graphics Processing Units

- GPUs and CPUs do **not** go back in computer architecture genealogy to a **common ancestor**
- The primary ancestors of GPUs are graphics accelerators
- Given the hardware invested to do graphics well architects ask:

how can be the design of GPUs used to improve the performance of a wider range of applications?

Graphics Processing Units

- The **challenge** for the GPU programmer
  - is **not** simply **getting good performance** on the GPU
  - but also in **coordinating** **the scheduling** of **computation** on the system processor and the GPU and the **transfer of data** between system memory and GPU memory

- GPUs have virtually **every type of parallelism** that can be captured by the programming environment:
  - multithreading
  - MIMD
  - SIMD
  - instruction-level

Programming the GPU

- NVIDIA developed a ***C-like language and programming environment*: CUDA - Compute Unified Device Architecture**

- CUDA produces C/C++ for the system processor - ***host*** - and a C and C++ dialect for the GPU - ***device*** (*D in CUDA*)

- **OpenCL** is a similar language, which several companies are developing as an independent multiple platforms language

Programming the GPU

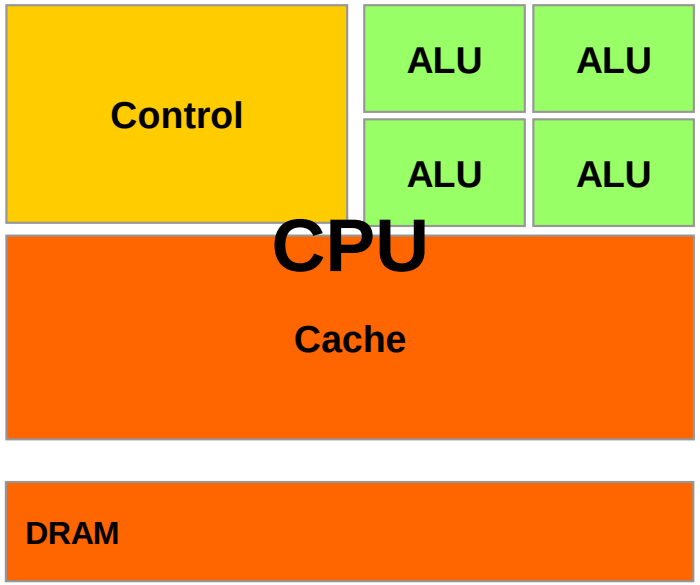
- NVIDIA unified all forms of parallelism in the ***CUDA Thread***
- The compiler and the hardware can gang thousands of CUDA threads together to utilize the various styles of parallelism within a GPU (multithreading, MIMD, SIMD, ILP)
- NVIDIA classifies the CUDA programming model as **Single Instruction, Multiple Thread** (***SIMT***)
- Threads are blocked together - ***Thread Block*** - and executed in **groups of 32** threads - ***warp***

CUDA Programming Model

- The GPU is viewed as a compute **device** with the following features:
  - It is a coprocessor to the CPU (**host**)
  - It has its own DRAM (**device memory**)
  - It runs many **threads in parallel**
- Data-parallel portions of an application are executed on the device as **kernels** which run in parallel 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

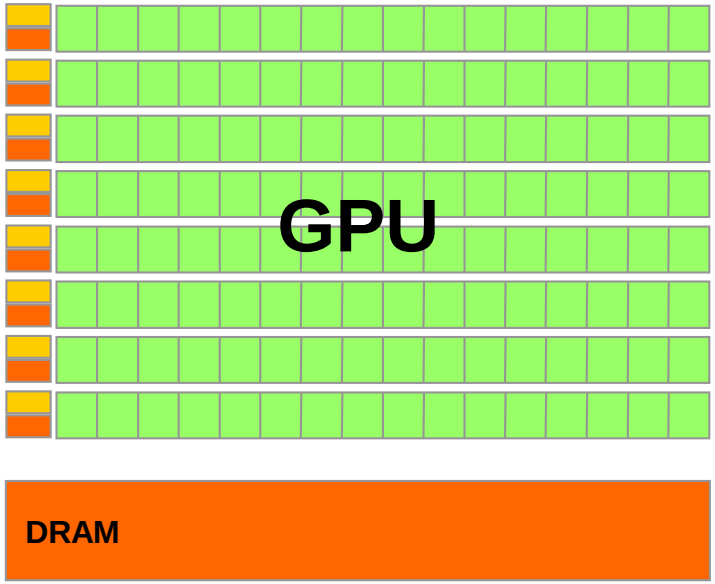
CPUs: Latency Oriented Design

- Large caches
  - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
  - Branch prediction for reduced branch latency
  - Data forwarding for reduced data latency
- Powerful ALU
  - Reduced operation latency



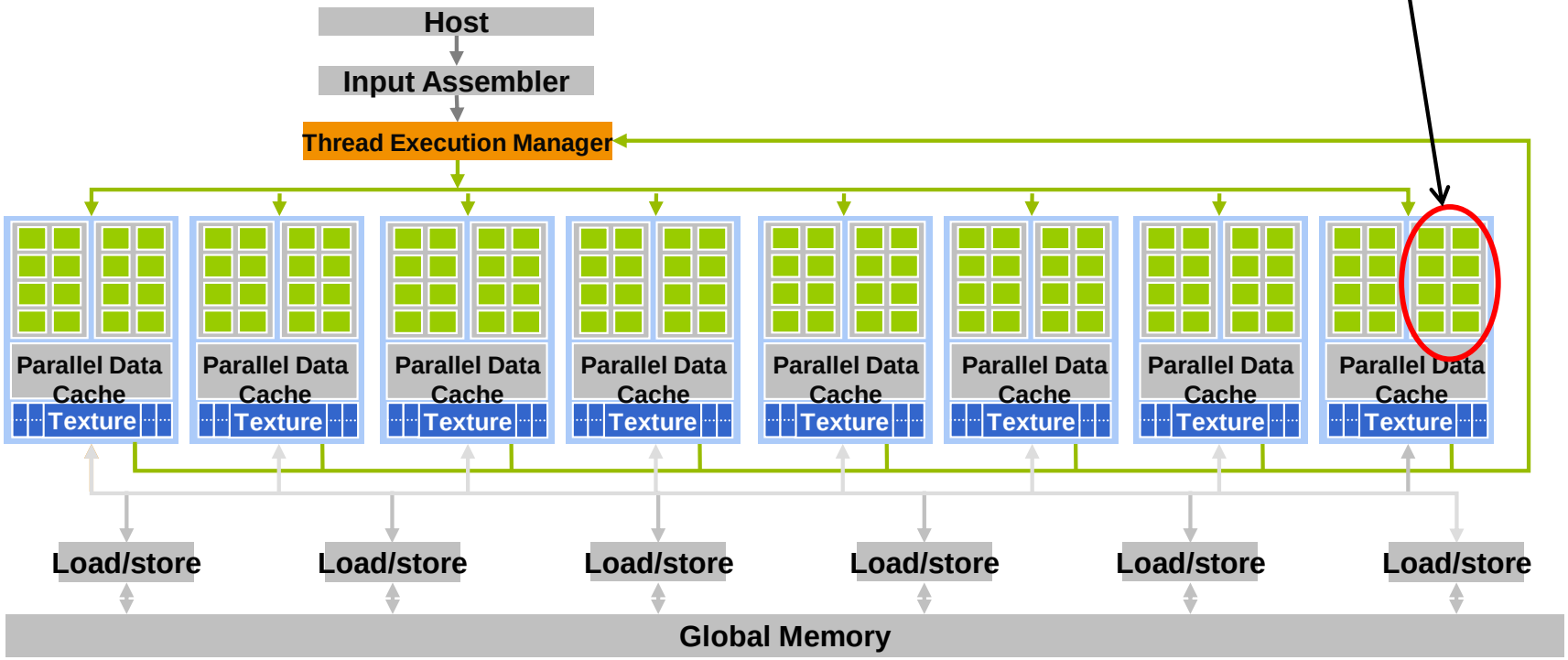
GPUs: Throughput Oriented Design

- Small caches
  - To boost memory throughput
- Simple control
  - No branch prediction
  - No data forwarding
- Energy efficient ALUs
  - Many, long latency but heavily pipelined for high throughput
- Require massive number of threads to tolerate latencies



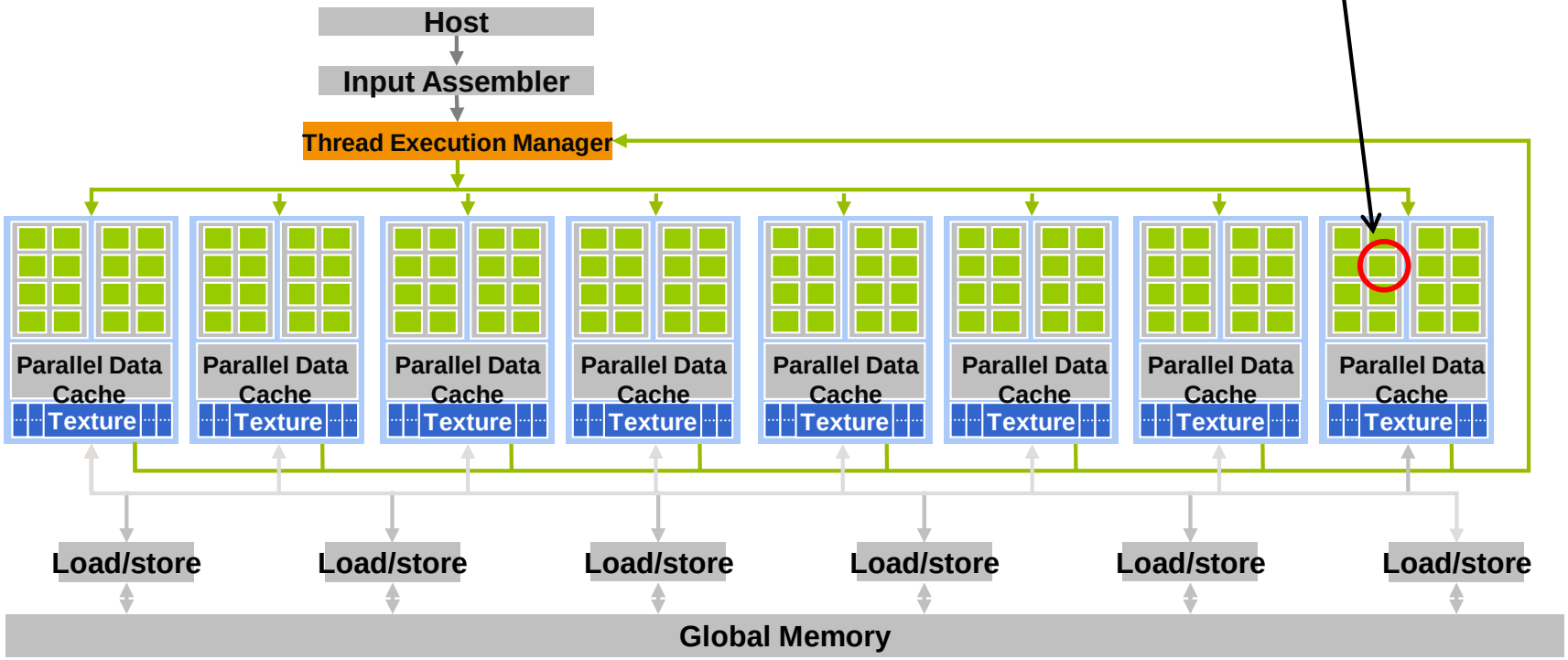
GPU Architecture

- A typical CUDA-capable GPU can be organized into
  - an array of highly threaded **streaming multiprocessors (SMs)**
  - two SMs form a building block
  - the number of SMs in a building block can vary from one generation of CUDA GPUs to another



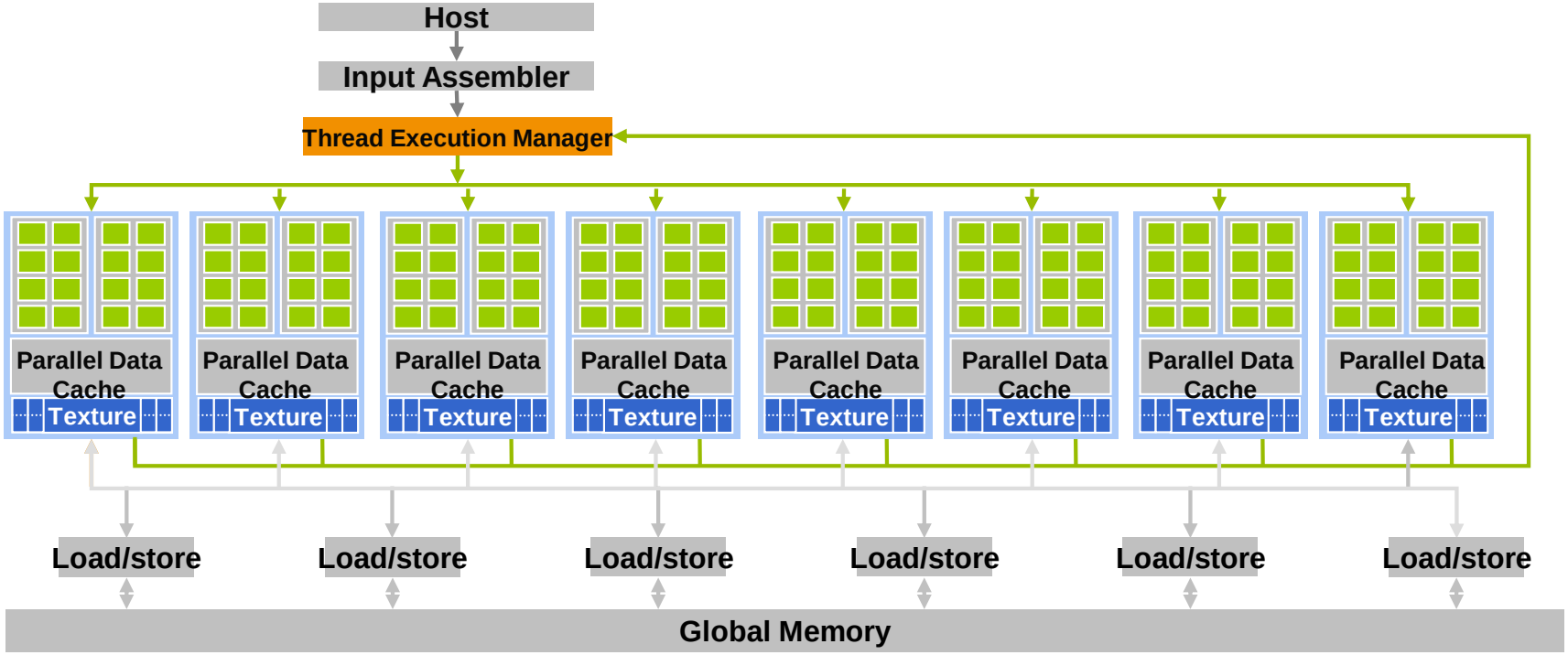
GPU Architecture

- Each SM has a number of **streaming processors (SPs)** that share control logic and instruction cache
- Each GPU currently comes with up to 4 gigabytes of graphics double data rate (GDDR) DRAM - **global memory**



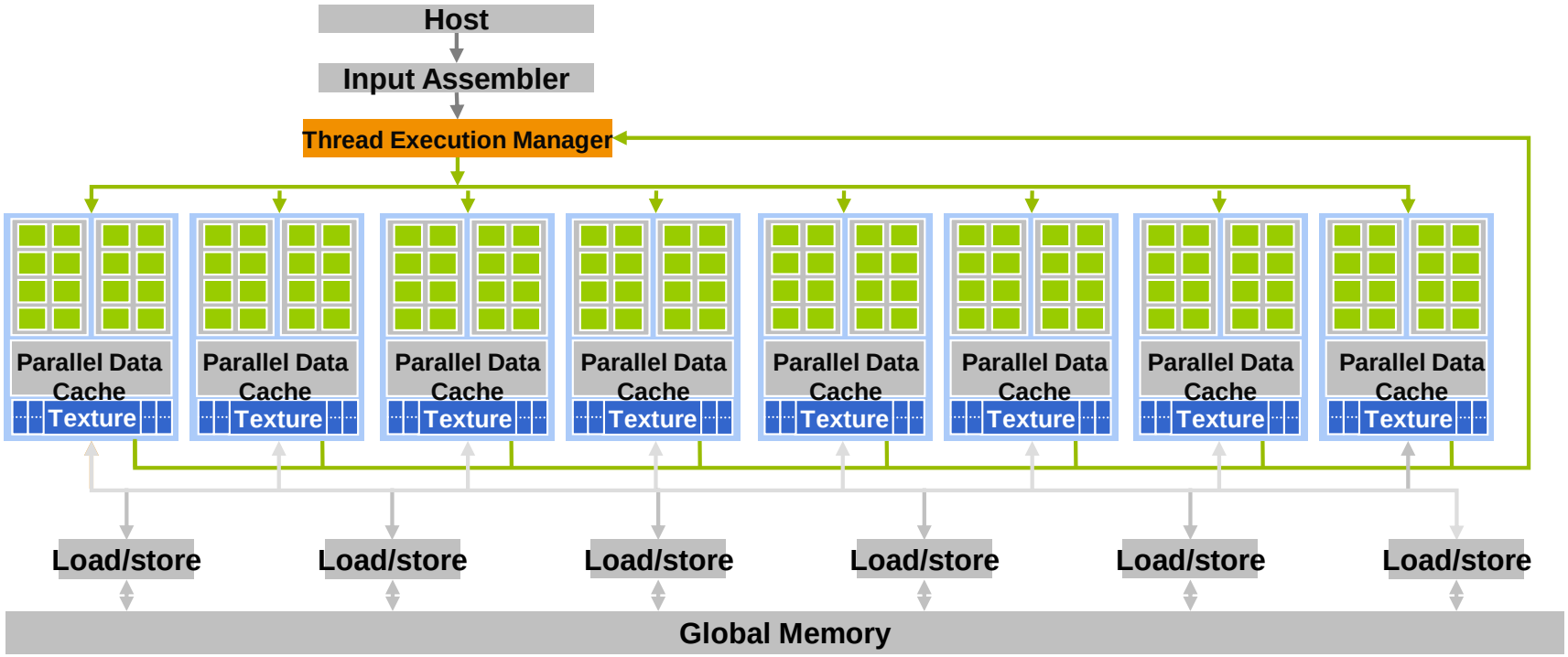
GPU Architecture

- The parallel **G80** chip has 128 SPs (16 SMs with 8 SPs)
- Each SP has a **multiply–add (MAD)** unit and an additional multiply unit
  - The **G80** (128 SPs) produces a total of **over 500 gigaflops**
  - The **GT200** (240 SPs) exceeds ***1 teraflops,*** the **GTX680** 1,5 **teraflops**



GPU Architecture

- The **G80** chip supports up to **768 threads** per SM, which sums up to about 12,000 threads for this chip
- The **GT200** supports **1024 threads** per SM and up to about 30,000 threads



CUDA Program Structure

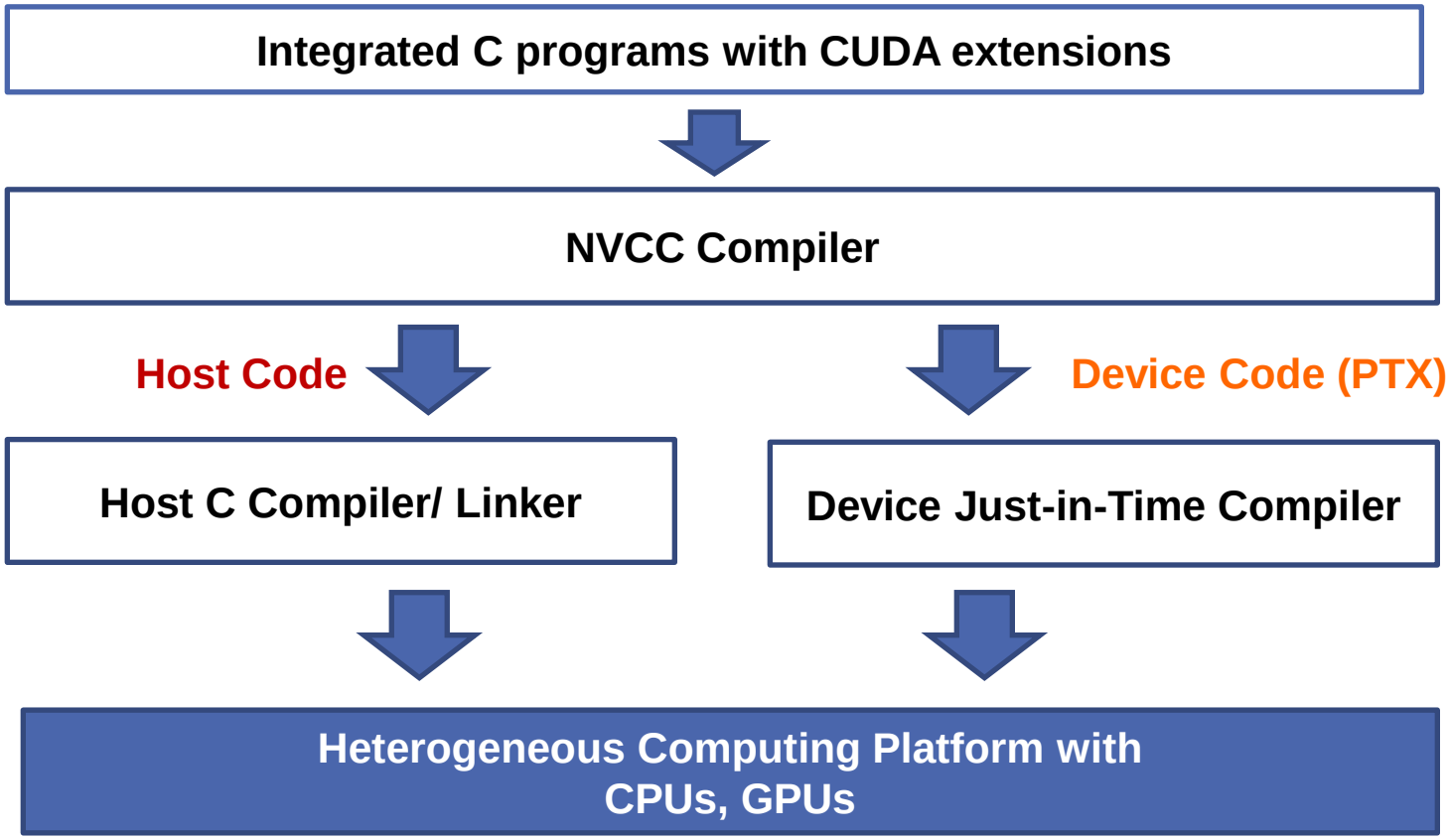
- The structure of a CUDA program reflects the computing system consisting of
  - a **host**, which is a traditional central processing unit (CPU)
  - one or more **devices** (GPUs)
- A **CUDA program** is a unified source code encompassing **both host and device code**
- The NVIDIA C compiler - **nvcc** - separates the two during the compilation process

CUDA Program Structure

- The **host code** is:
  - straight ANSI C code
  - it is further compiled with the host's standard C compilers and runs as an ordinary CPU process

- The **device code** is:
  - written using ANSI C extended with **keywords** for labeling data-parallel functions, called **kernels**, and their associated **data structures**
  - The device code is typically further compiled by the **nvcc** and executed on the GPU device

Compiling A CUDA Program



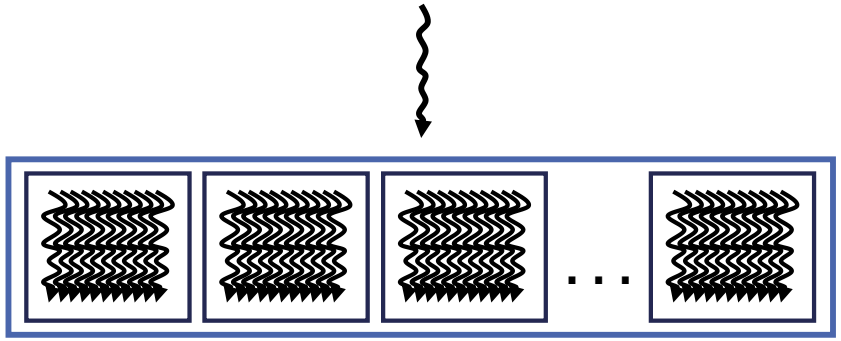
CUDA Execution Model

- The **execution starts with host (CPU) execution**
- When a **kernel function** is launched, the execution is moved to a device (GPU), where a large number of threads are generated to take advantage of abundant data parallelism

Serial Code (host)

Parallel Kernel (device)

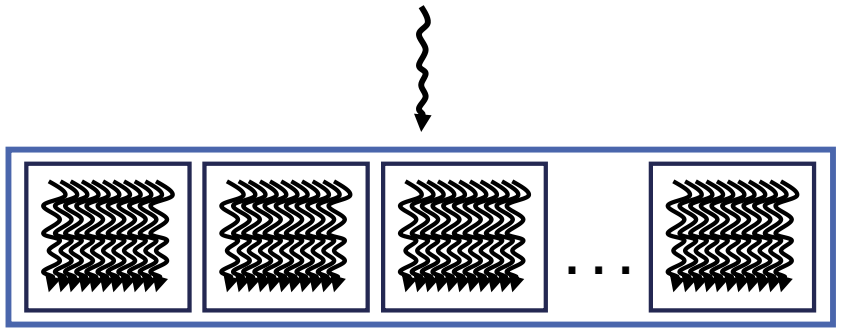
KernelA<<< nBlk, nTid >>>(args);



Serial Code (host)

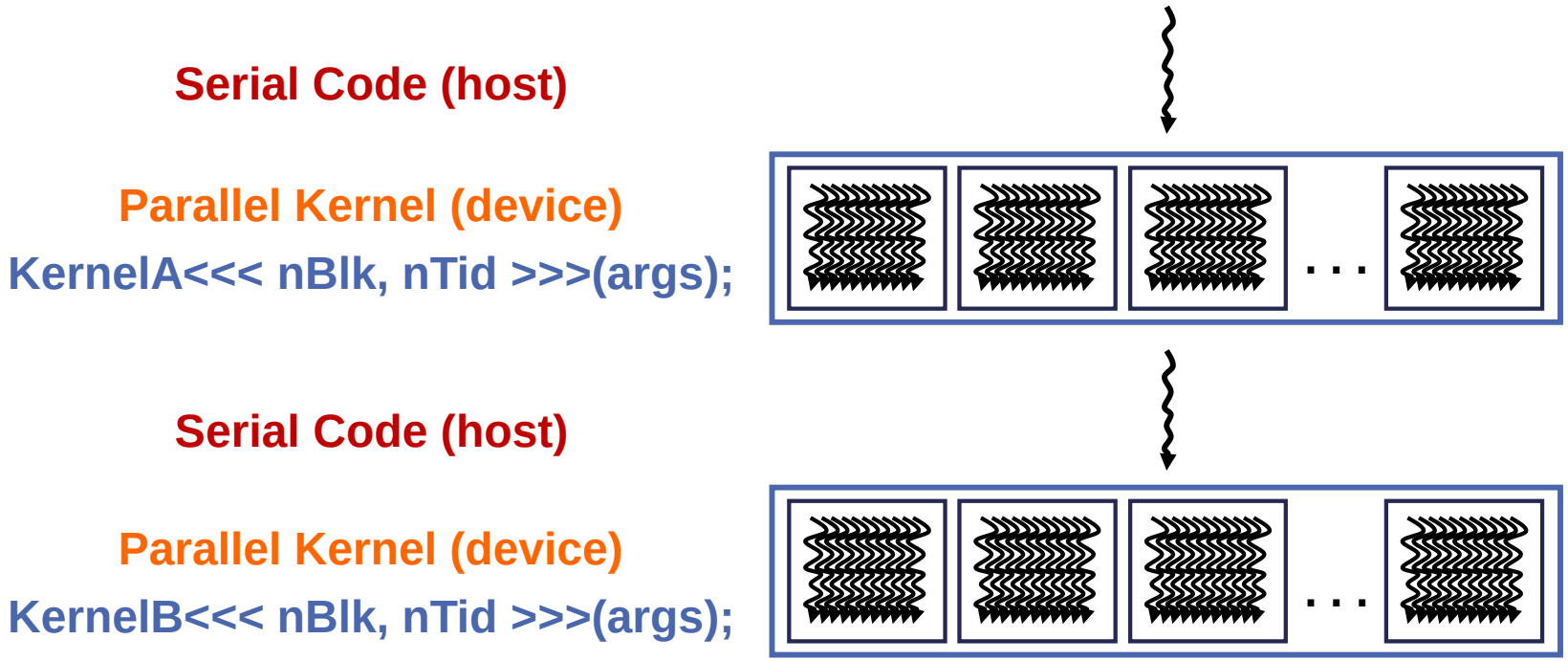
Parallel Kernel (device)

KernelB<<< nBlk, nTid >>>(args);



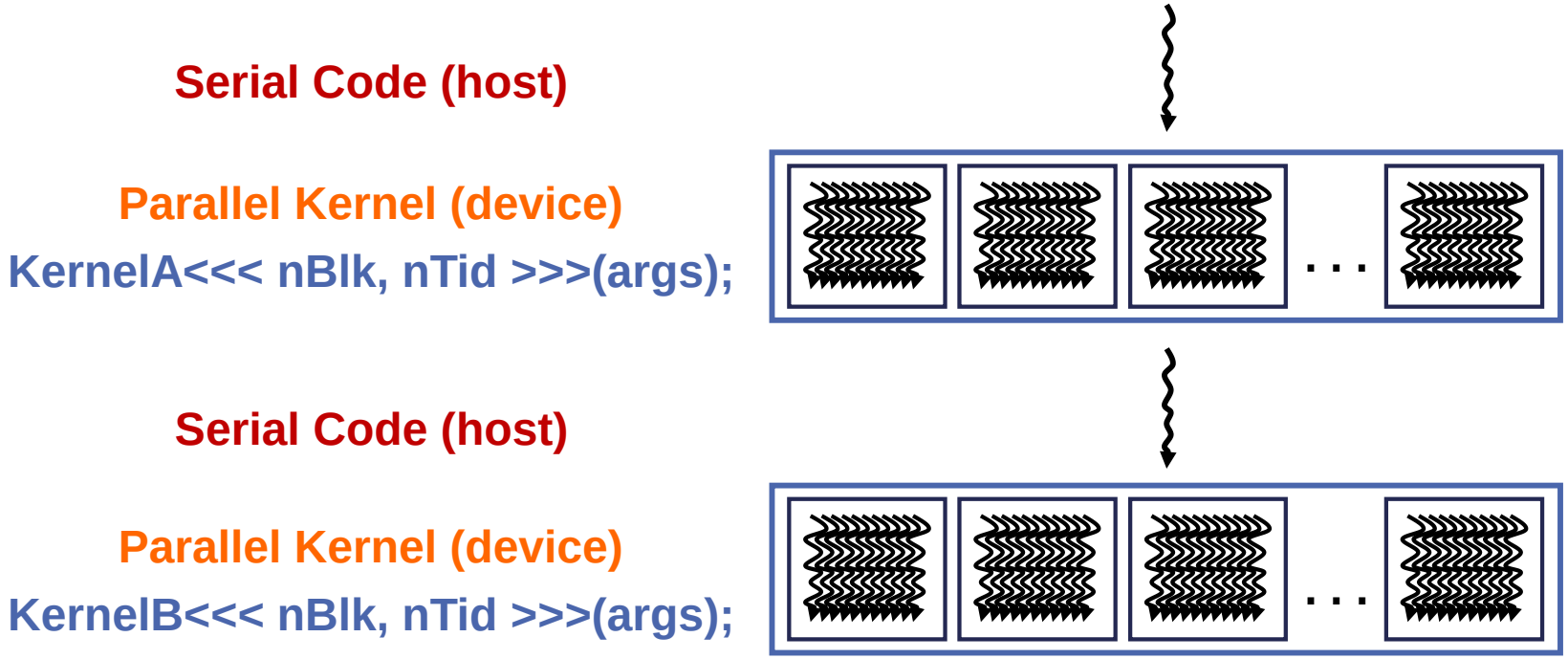
CUDA Execution Model

- All the threads that are generated by a kernel during an invocation are collectively called a **grid**
- Figure shows the execution of **two grids of threads**



CUDA Execution Model

- When all threads of a **kernel** complete their execution:
  - the corresponding **grid terminates**
  - the **execution continues on the host** until another kernel is invoked



Programming the GPU

- To distinguish between **functions** for the GPU (device) and functions for the system processor (host), CUDA uses:
  - `__device__` or `__global__` for the device
  - `__host__` for the processor

- CUDA **variables** declared as in the
  - `__device__` (or `__global__`) functions

  are allocated to the GPU Memory which is accessible by all multithreaded SIMD processors

Programming the GPU

- The call syntax for the function *name* that runs on the GPU

```
name<<<dimGrid, dimBlock>>>(... parameter list ...)
```

where **dimGrid** and **dimBlock** specify the dimensions of the grid (in blocks) and the dimensions of a block (in threads)

- CUDA provides keywords for:
  - the identifier for blocks per grid - **blockIdx** -
  - the identifier for threads per block - **threadIdx** -
  - the number of threads per block - **blockDim** - which comes from the **dimBlock** parameter

Example

- Consider the DAXPY example

```
// Invoke DAXPY
daxpy(n, 2.0, x, y);
// DAXPY in C
void daxpy(int n, double a, double *x, double *y)
{
      for (int i = 0; i < n; ++i)
            y[i] = a*x[i] + y[i];
}
```

Example

- In the CUDA version, we launch:
  - **n threads**, one per vector element
  - with **256 CUDA Threads per thread block**

```
// Invoke DAXPY with 256 threads per Thread Block
__host__
int nblocks = (n + 255) / 256;
        daxpy<<<nblocks, 256>>>(n, 2.0, x, y);
// DAXPY in CUDA
__device__
void daxpy(int n, double a, double *x, double *y)
{
        int i = blockIdx.x*blockDim.x + threadIdx.x;
        if (i < n) y[i] = a*x[i] + y[i];
}
```

Example

- The GPU function calculates the corresponding element **index i** based on the **block ID**, the **number of threads per block**, and the **thread ID**
- If this index is within the array (i < n), it performs the multiply and add

```
// Invoke DAXPY with 256 threads per Thread Block
__host__
int nblocks = (n+ 255) / 256;
        daxpy<<<nblocks, 256>>>(n, 2.0, x, y);
// DAXPY in CUDA
__device__
void daxpy(int n, double a, double *x, double *y)
{
        int i = blockIdx.x*blockDim.x + threadIdx.x;
        if (i < n) y[i] = a*x[i] + y[i];
}
```

Threads and Blocks

- The C version has:
  - A loop where each iteration is independent of the others
  - This allows the loop to be transformed into a parallel code
  - **Each loop iteration** becomes an **independent thread**

- The programmer determines the parallelism in CUDA explicitly by specifying
  - the **grid dimensions**
  - the **number of threads per Streaming Processor**
- By assigning a single thread to each element, there is **no need to synchronize** among threads when writing results to memory

Threads and Blocks

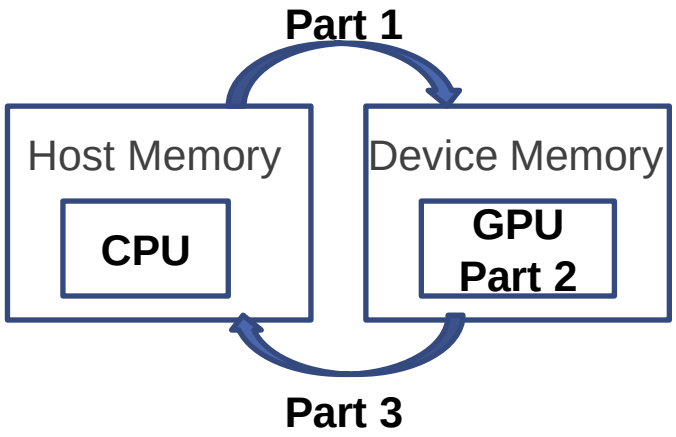
- A **thread** is associated with **each data element**
  - *CUDA threads*, with thousands of which for various styles of parallelism
- Threads are organized into **blocks**
  - ***Thread Blocks***: groups of up to 512 elements
  - ***Streaming Processor***: hardware that executes a whole thread block (32 elements executed per thread at a time)
- Blocks are organized into a **grid**
  - **Blocks are executed independently** and in any order
  - Different **blocks cannot communicate directly** but can ***coordinate*** using memory operations in GPU Global Memory

Vector Addition – Traditional C Code

```
// Compute vector sum C = A+B
void vecAdd(float* A, float* B, float* C, int n)
{
  for (i = 0, i < n, i++)
    C[i] = A[i] + B[i];
}

int main()
{
    // Memory allocation for A_h, B_h, and C_h
   // I/O to read A_h and B_h, N elements
   …
    vecAdd(A_h, B_h, C_h, N);
}
```

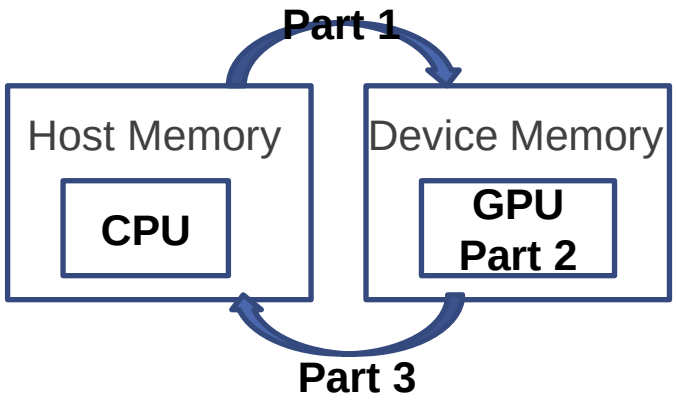
Vector Addition – Kernel

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    int size = n* sizeof(float);
    float* d_A, d_B, d_C;
    …
1.  // Allocate device memory for A, B, and C
    // Copy A and B to device memory

2.  // Kernel launch code – to have the device
    // to perform the actual vector addition

3.  // Copy C from the device memory
    // Free device vectors
}
```



Device Memory and Data Transfer

- The host and devices have separate memory spaces
- To execute a kernel on a device
  - the programmer needs to **allocate memory on the device**
  - **transfer data** from the host memory to the allocated device memory
  - this corresponds to *Part 1 of Figure*
- After device execution
  - the programmer needs to **transfer result data** from the device memory back to the host memory
  - **free up** the device memory
  - this corresponds to *Part 3 of Figure*

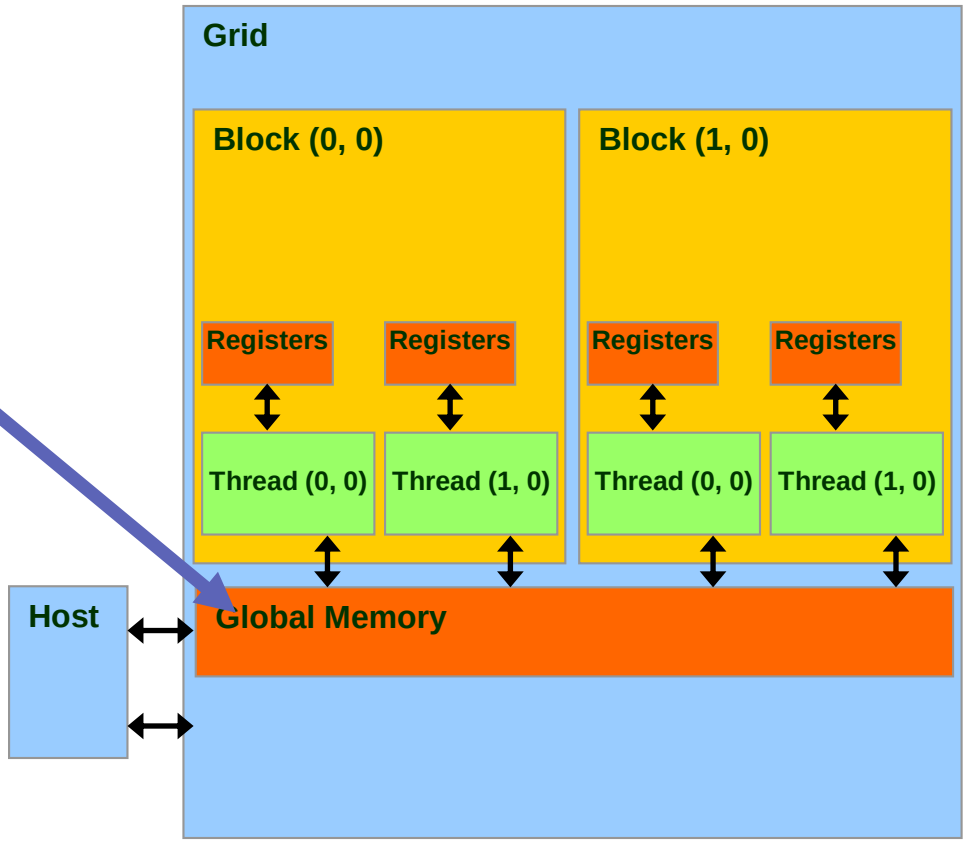


Device Memory and Data Transfer

- The CUDA memory model is supported by API functions that help programmers to manage data in memories
- The function **`cudaMalloc()`**:
  - called **from the host code** to allocate object in the device global memory
  - Two parameters:
    - **address of a pointer** variable to the allocated object after allocation
    - **size** of the allocated object in terms of bytes
- The function **`cudaFree()`** :
  - Frees object from device global memory
    - Pointer to freed object
- The function **`cudaMemcpy()`** for memory data transfer

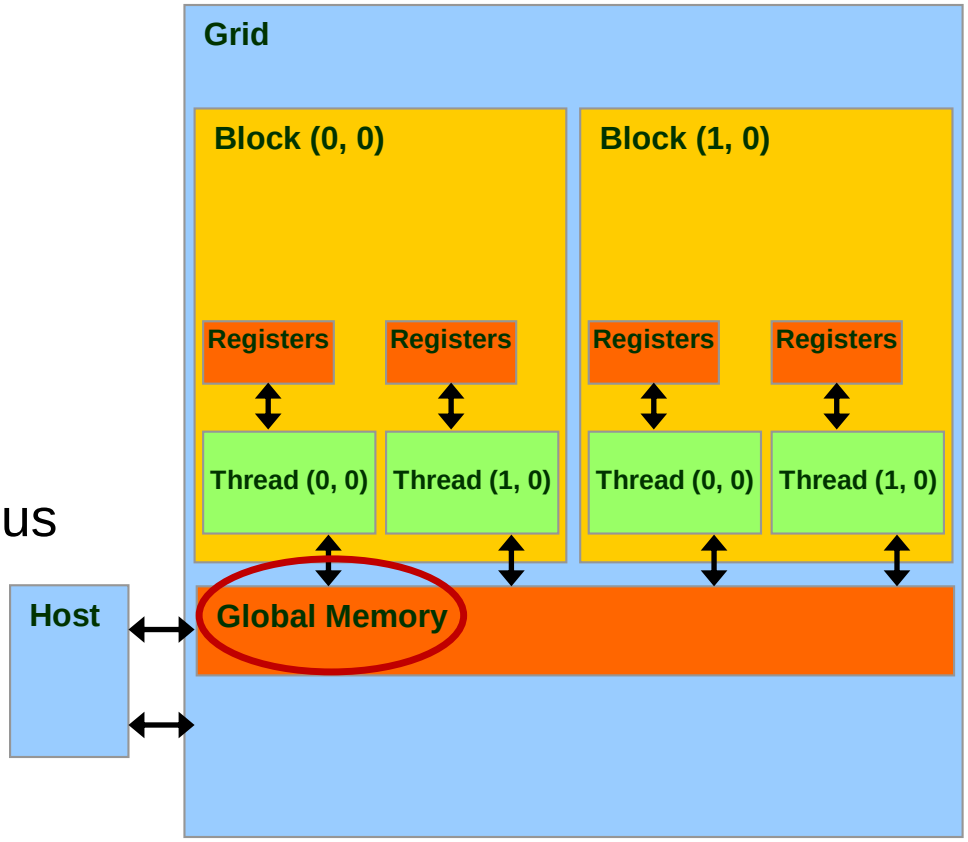
CUDA Device Memory Management API

- cudaMalloc()
  - Allocates object in the device ***global memory***
  - Two parameters
    - **Address of a pointer** to the allocated object
    - **Size** of allocated object in terms of bytes
- cudaFree()
  - Frees object from device global memory
    - **Pointer** to freed object



Host-Device Data Transfer API functions

- cudaMemcpy()
  - memory data transfer
  - requires four parameters
    - **Pointer** to destination
    - **Pointer** to source
    - Number of bytes copied
    - Type/Direction of transfer
  - Transfer to device is asynchronous

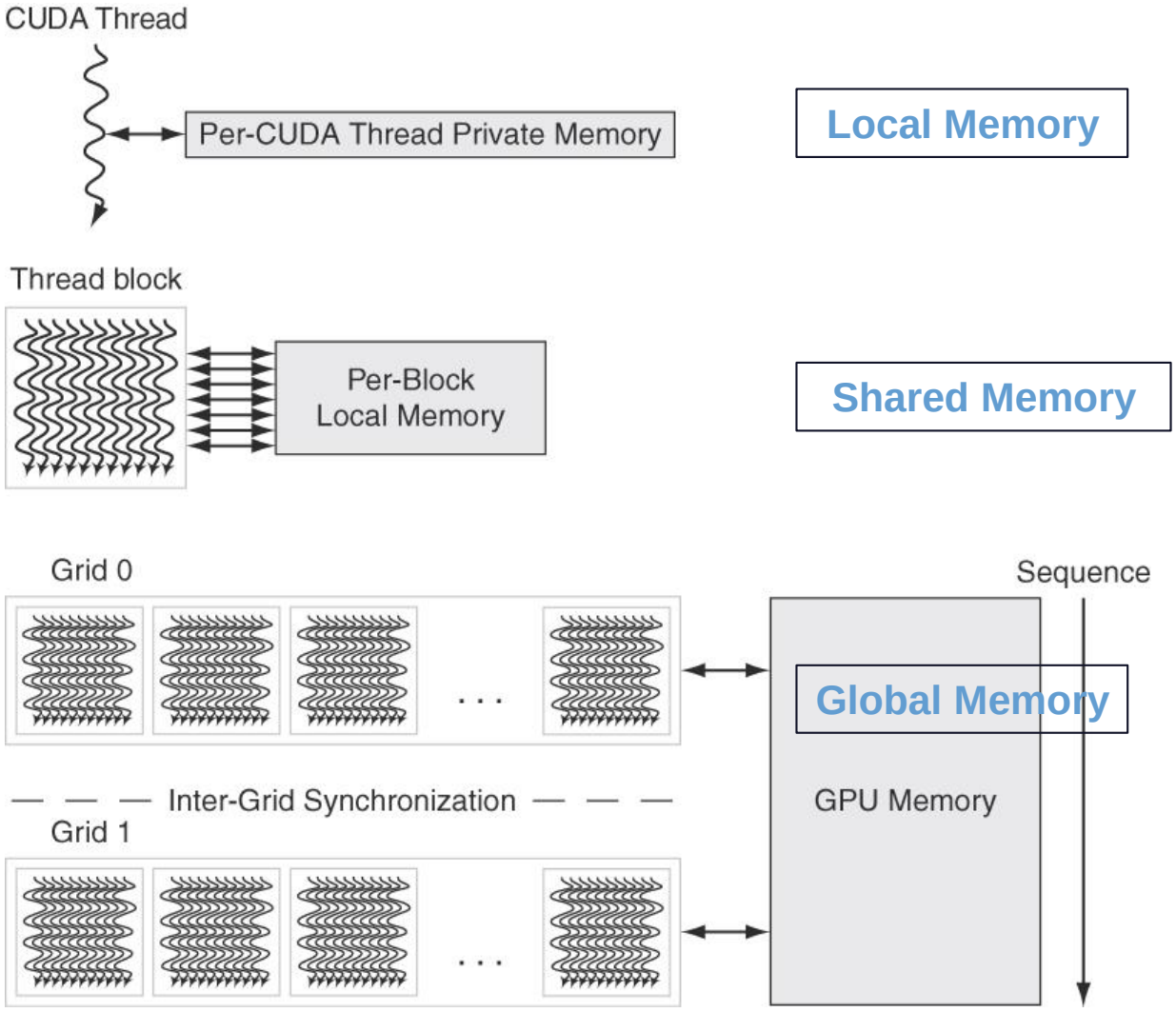


NVIDIA GPU Memory Structures

Local Memory is private to a single CUDA Thread

Shared Memory is shared by all threads of SIMD instructions within a thread block

GPU Global Memory is shared by all Grids

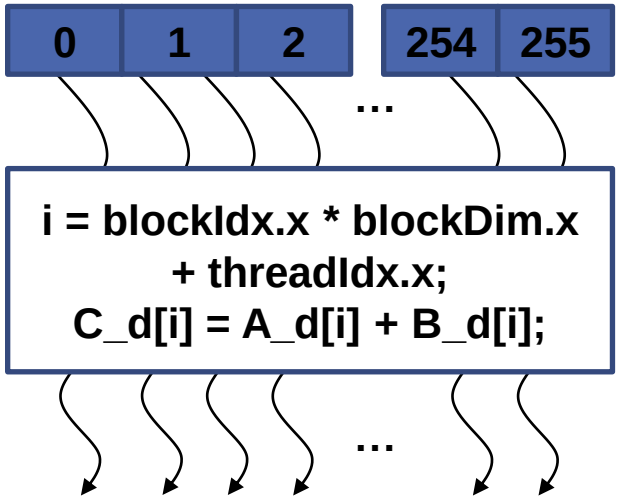


Vector Addition

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    int size = n * sizeof(float);
     float* d_A, d_B, d_C;

1. // Transfer A and B to device memory
     cudaMalloc((void **) &d_A, size);
     cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
     cudaMalloc((void **) &d_B, size);
     cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

   // Allocate device memory for C
      cudaMalloc((void **) &d_C, size);

2. // Kernel invocation code – to be shown later
      …
3. // Transfer C from device to host
      cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    // Free device memory for A, B, C
      cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

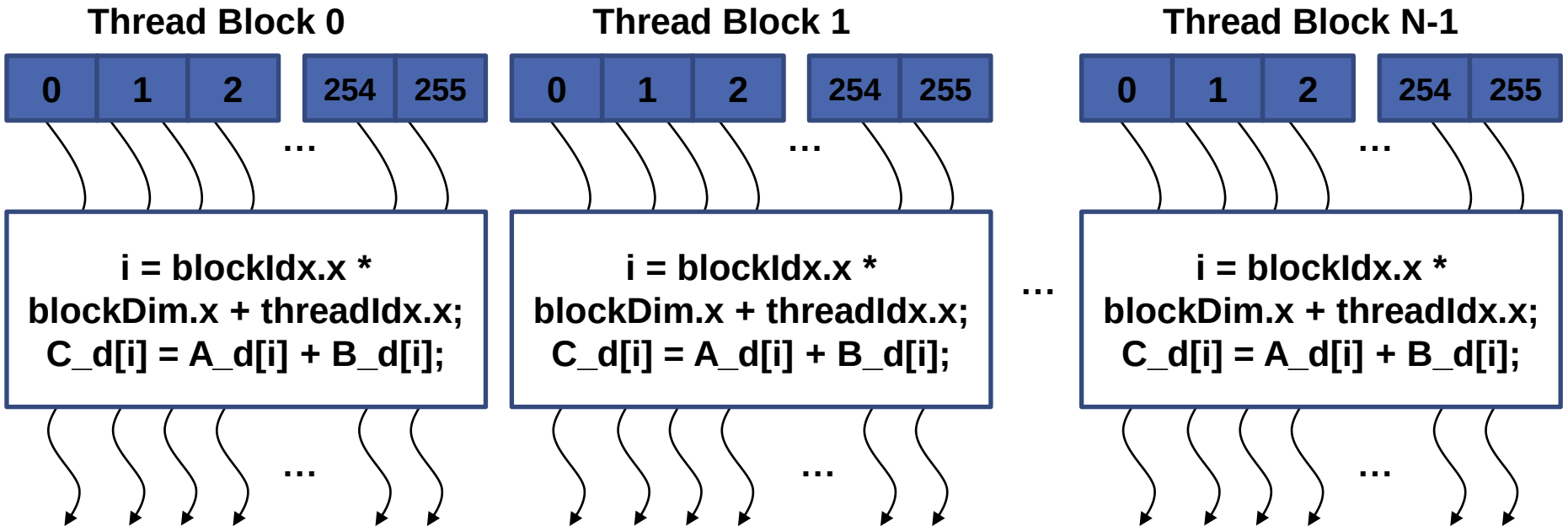
Arrays of Parallel Threads

- A **kernel** function specifies the code to be executed by all threads during a parallel phase
  - All of these threads execute the same code
- A CUDA kernel is executed by a grid (array) of threads
  - All threads in a grid run the same kernel code
  - Each thread has an index that it uses to compute memory addresses and make control decisions



Thread Blocks: Scalable Cooperation

- Thread array is divided into multiple blocks
  - Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization**
  - Threads in different blocks cannot cooperate

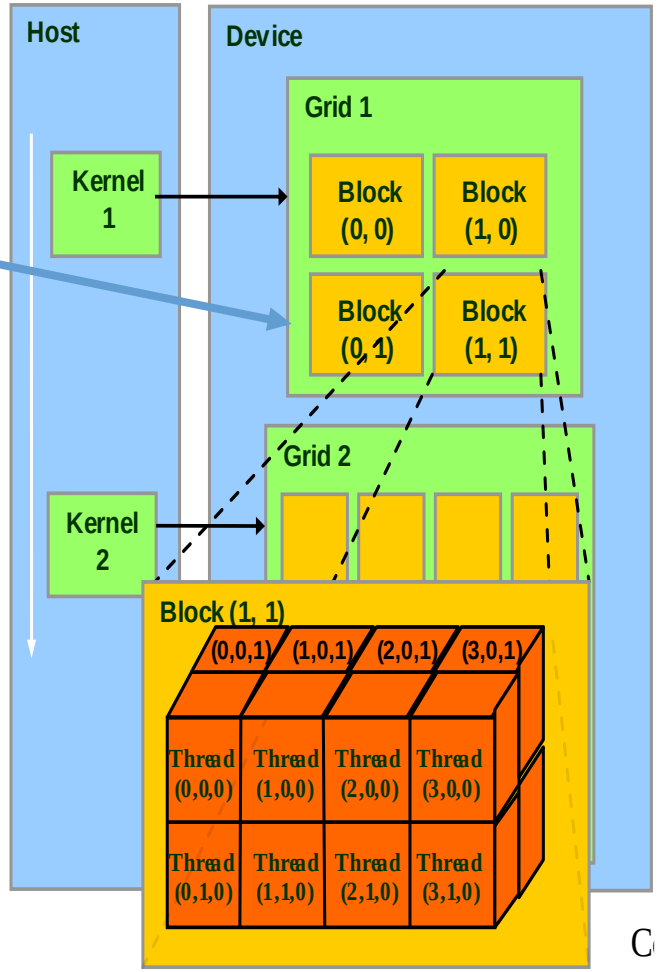


Arrays of Parallel Threads

- When a **kernel** is invoked, it is executed as **grid** of parallel threads
- Each CUDA thread grid typically is comprised of **thousands to millions** of lightweight GPU threads per kernel invocation
- Creating enough threads to fully utilize the hardware often requires a large amount of data parallelism

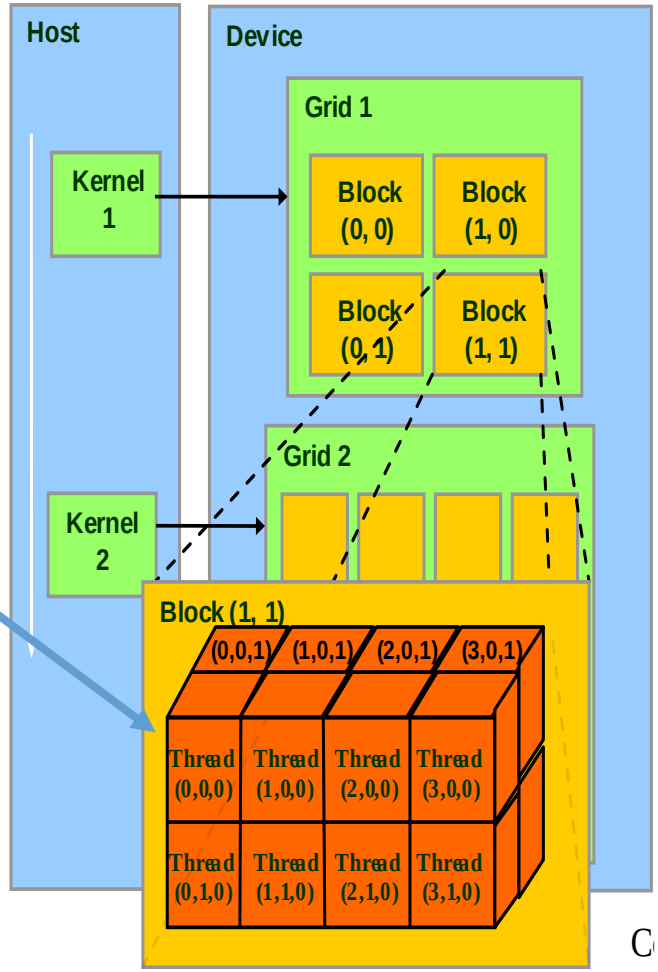
blockIdx and threadIdx

- Threads in a grid are organized into **a two-level hierarchy**
  - ***top level***, each **grid** consists of one or more thread blocks
  - All **blocks** in a grid have the **same number of threads** organized in the same manner
  - Each **grid** is organized as a as a **three-dimensional array** of blocks
  - Each **block** has a unique **three dimensional coordinate** given by the CUDA specific keywords **`blockIdx.x`**, **`blockIdx.y`** and **`blockIdx.z`**



blockIdx and threadIdx

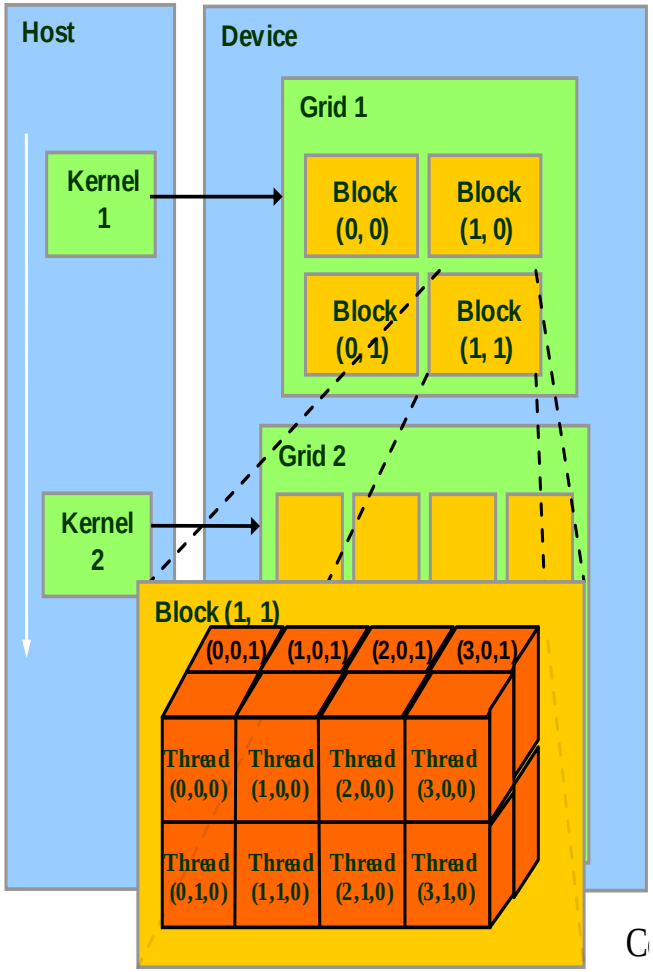
- Threads in a grid are organized into **a two-level hierarchy**
  - Each thread **block** is organized as a **three-dimensional array** of threads with a total size of up to 512 threads
  - The coordinates of threads in a block are uniquely defined by three thread indices: `threadIdx.x`, `threadIdx.y`, and `threadIdx.z`
  - Not all applications will use all three dimensions of a thread block



blockIdx and threadIdx

- Threads in a grid are organized into **a two-level hierarchy**
- In Figure
  - each thread block is organized into a 4x2x2 three-dimensional array of threads
  - this gives Grid 1 a total of 4x16 = 64 threads

- Each thread uses indices to decide what data to work on
  - blockIdx: 1D, 2D, or 3D (CUDA 4.0)
  - threadIdx: 1D, 2D, or 3D



CUDA Thread Organization

- When a thread executes the kernel function, references to the `blockIdx` and `threadIdx` variables return the **coordinates of the thread**
- Additional built-in variables, `gridDim` and `blockDim`, provide the **dimension of the grid** and the **dimension of each block**

- `threadID=blockIdx.x * blockDim.x + threadIdx` identifies the part of the input data to read from and the part of the output data structure to write to
  - **Example** Thread 3 of Block 0 has a threadID value of 0*M + 3 = 3
  - **Example** Thread 3 of Block 5 has a threadID value of 5*M + 3

CUDA threads, blocks and grids

- Nvidia use the **Compute Capability** specification to encode what each generation of GPU chips is capable of
- The Compute Capability (CC) of a GPU can be discovered by running the **`deviceQuery`** utility

	Compute Capability			
Item	**1.x**	**2.x**	**3.x**	**5.x**
Max. number of grid dimensions	2	3		
Grid maximum x-dimension	$2^{16}-1$		$2^{31}-1$	
Grid maximum y/z-dimension	$2^{16}-1$			
Max. number of block dimensions	3			
Block max. x/y-dimension	512	1024		
Block max. z-dimension	64			
Max. threads per block	512	1024		
GPU example (GTX family chips)	8800	480	780	980

CUDA Thread Organization

- The exact organization of a grid is determined by the **execution configuration** provided at kernel launch
  - The first parameter specifies the dimensions of the grid as # blocks
  - The second specifies the dimensions of each block as # threads
  - Each such parameter is a **`dim3`** type, a C struct with three unsigned integer fields: x, y, and z
  - Example

```
dim3 dimGrid(128, 1, 1);
dim3 dimBlock(32, 1, 1);
vecAddKernel<<<dimGrid, dimBlock>>>(. . .);
```

oppure

```
dim3 cat(128, 1, 1);
dim3 dog(32, 1, 1);
KernelFunction<<<cat, dog>>>(. . .);
```

Execution Configuration Examples

Assuming we have

```
dim3 b(3,3,3);
dim3 g(20,100);
```

Different grid-block combination are possible

- `foo<<<g,b>>>();` // Run a 20x100 grid made of 3x3x3 blocks
- `foo<<<10,b>>>();` // Run a 10-block grid, each block made by 3x3x3 threads
- `foo<<<g,256>>>();` // Run a 20x100 grid, made of 256 threads
- `foo<<<g,2048>>>();` // An invalid example: maximum block size is 1024 threads even for compute capability 5.x
- `foo<<<5,g>>>();` // Another invalid example, that specifies a block size of 20x100=2000 threads
- `foo<<<10,256>>>;` // simplified configuration for a 1D grid of 1D blocks

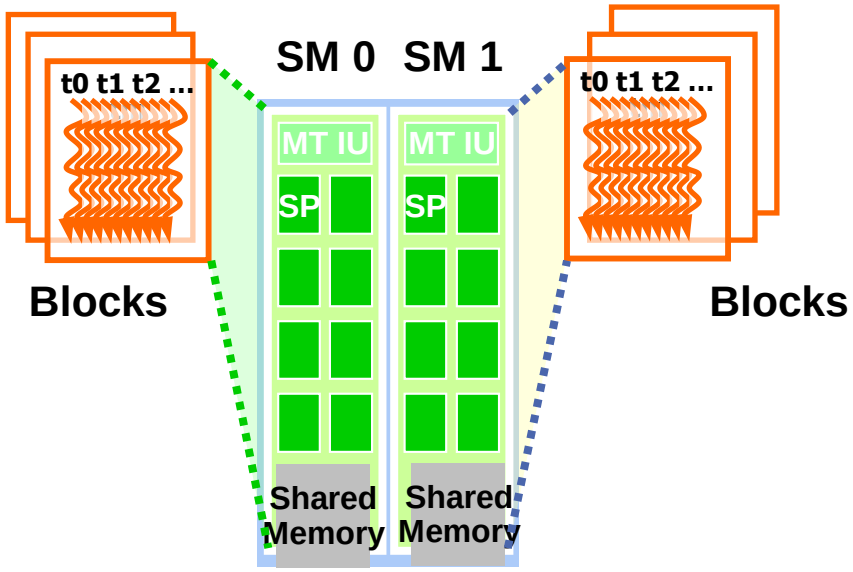
Synchronization

- CUDA allows threads in the same block to coordinate their activities using a barrier synchronization function, **`__syncthreads()`**
  - the thread that executes the function call will be held at the calling location until every thread in the block reaches the location

- A **`__syncthreads()`** statement must be executed by all threads in a block of the kernel before any moves on to the next phase

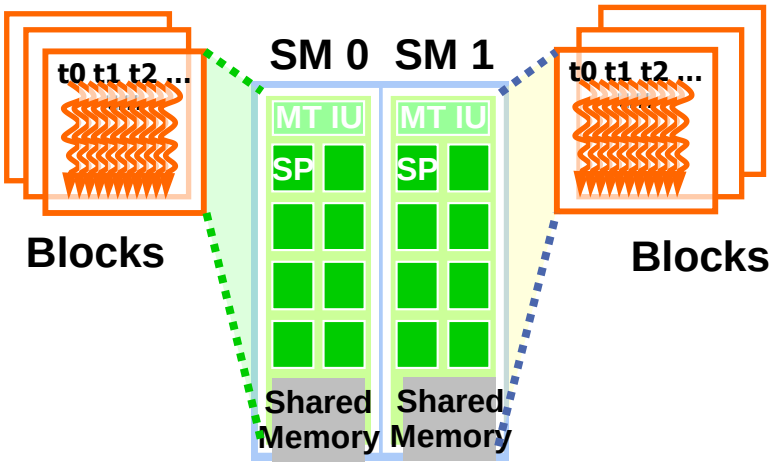
Thread and Block Assignment

- Once a **kernel is launched**, the CUDA runtime system generates the corresponding **grid of threads**
  - threads are assigned to execution resources on a block-by-block basis
- The execution resources are organized into streaming multiprocessors (SMs)
- Each device has a limit on the number of block that can be assigned to each SM



Thread and Block Assignment

- When an *insufficient* amount of any one or more types of resources needed for the simultaneous execution of blocks, the *CUDA runtime automatically reduces the number of blocks assigned to each SM*
- The runtime system maintains a list of blocks that need to execute and assigns new blocks to SMs as they complete the execution of blocks previously assigned to them



Thread Scheduling

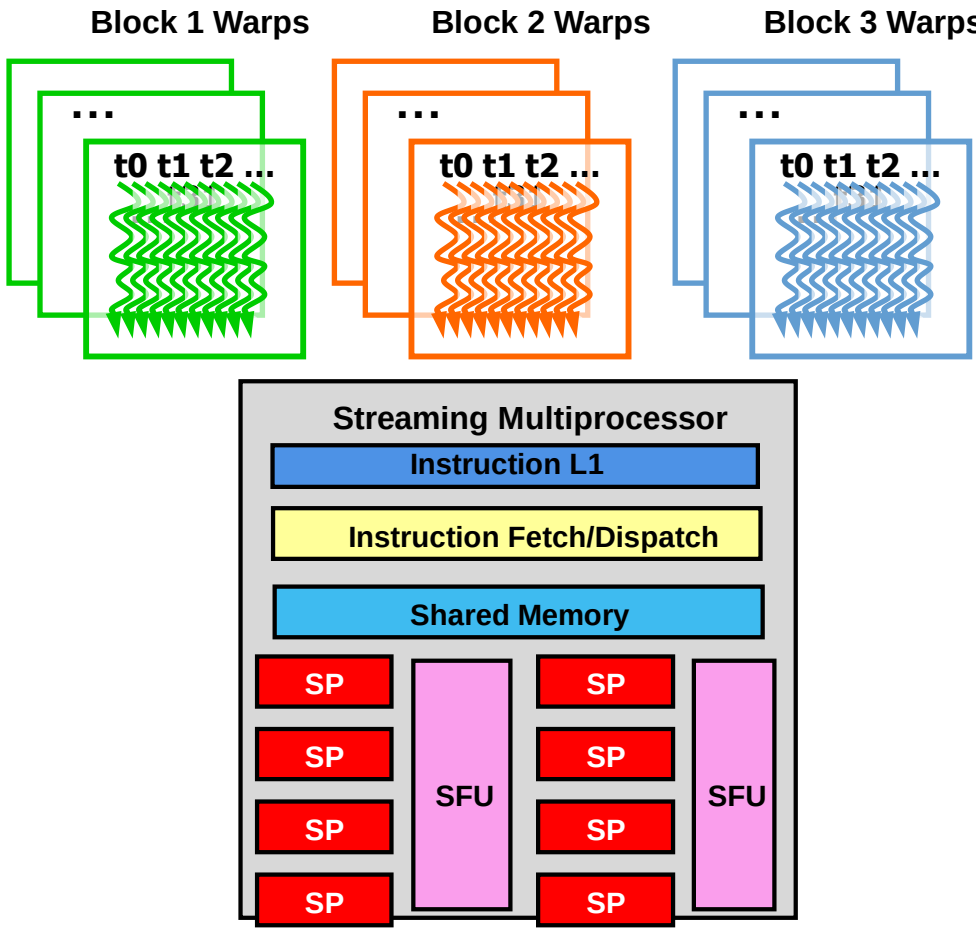
- Once a block is assigned to a streaming multiprocessor, it is further divided into 32-thread units called **warps**
- The warp is the **unit of thread scheduling** in SMs

- Each warp consists of 32 threads of consecutive threadIdx values:
  - threads 0 through 31 form the first warp
  - threads 32 through 63 the second warp, and so on

- We can calculate the number of warps that reside in an SM for a given block size and a given number of blocks assigned to each SM

Thread Scheduling

- Each Block is executed as 32-thread Warps
  - Warps are scheduling units in SM
- **Example** If 3 blocks are assigned to an SM and each block has 256 threads, how many **warps** are there in an SM?



- 3 blocks, each block 256 threads
- each block has 256/32 = 8 warps
- having 3 blocks in each SM, we have 8 x 3 = 24 warps in each SM

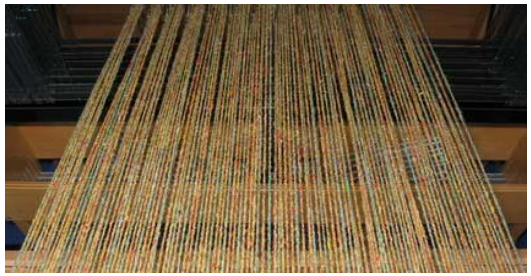
Thread Scheduling

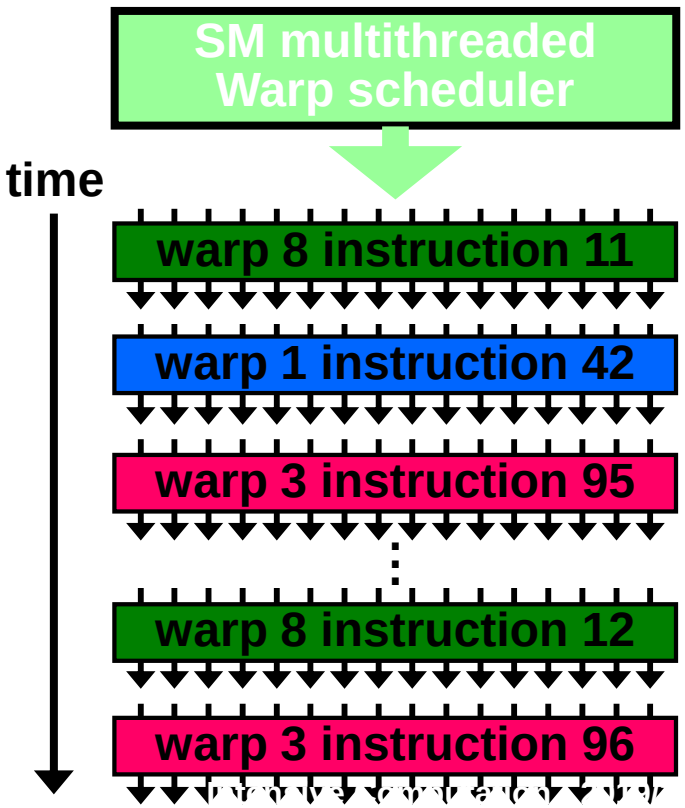
- **Why** do we need to have so many warps in an SM if there are only 8 SPs in an SM?
  - The answer is for **efficiently executing long-latency operations** such as global memory accesses
  - When an instruction executed by the threads in a warp needs to wait for the result of a previously initiated long-latency operation, the **warp is not selected for execution**
  - Another resident warp (that is no waiting for results) is selected for execution
  - If more than one warp is ready for execution, a **priority** mechanism is used to select one for execution
  - This mechanism of filling the latency of expensive operations with work from other threads is often referred to as ***latency hiding***

Thread Scheduling

- Note that **warp scheduling** is also used for tolerating other types of long latency operations such as *pipelined floating-point arithmetic* and *branch instructions*
- With enough warps around
  - the hardware will likely find a warp to execute at any point in time
  - full use of the execution hardware in spite of long-latency operations
- The selection of ready warps for execution
  - does not introduce any idle time into the execution timeline
  - ***zero-overhead thread scheduling***
- With warp scheduling, the long waiting time of warp instructions is hidden by executing instructions from other warps

SM Warp Scheduling





- SM hardware implements zero-overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - **Eligible Warps** are selected for execution on a prioritized scheduling policy
  - **All threads in a Warp execute the same instruction** when selected

- 4 clock cycles needed to dispatch the same instruction for all threads in a Warp in G80
  - If one global memory access is needed for every 4 instructions
  - A minimum of 13 Warps are needed to fully tolerate 200-cycle memory latency

Thread Scheduling

List of GPU chips and their SM capabilty

	compute capability					
Item	**1.0, 1.1**	**1.2, 1.3**	**2.x**	**3.0**	**3.5**	**5.0**
Concurrent kernels/device	1		16		32	
Max. resident blocks/SM	8			16		32
Max. resident warps/SM	24	32	48	64		
Max. resident threads/SM	768	1024	1536	2048		
32-bit registers/SM	8k	16k	32k	64k		
Max. registers/thread	128		63		255	

Exercise

Simple exercise (register and shared memory not considered)

- Assume a CUDA device allowing 8 blocks, 1024 threads per SM and 512 thread in each block
- For matrix multiplication, should we use 8x8, 16x16, or 32x32 thread blocks?
- Analyze the pros and cons of each choice:
  - If we use **8x8 blocks**, each block would have only 64 threads, and we will need 1024/64 = 12 blocks to fully occupy an SM
  - We are limited to 8 blocks in each SM, we will end up with only 64 x 8 = 512 threads in each SM
  - Then the SM execution resources will likely be underutilized because there will be fewer warps to schedule around long-latency operations

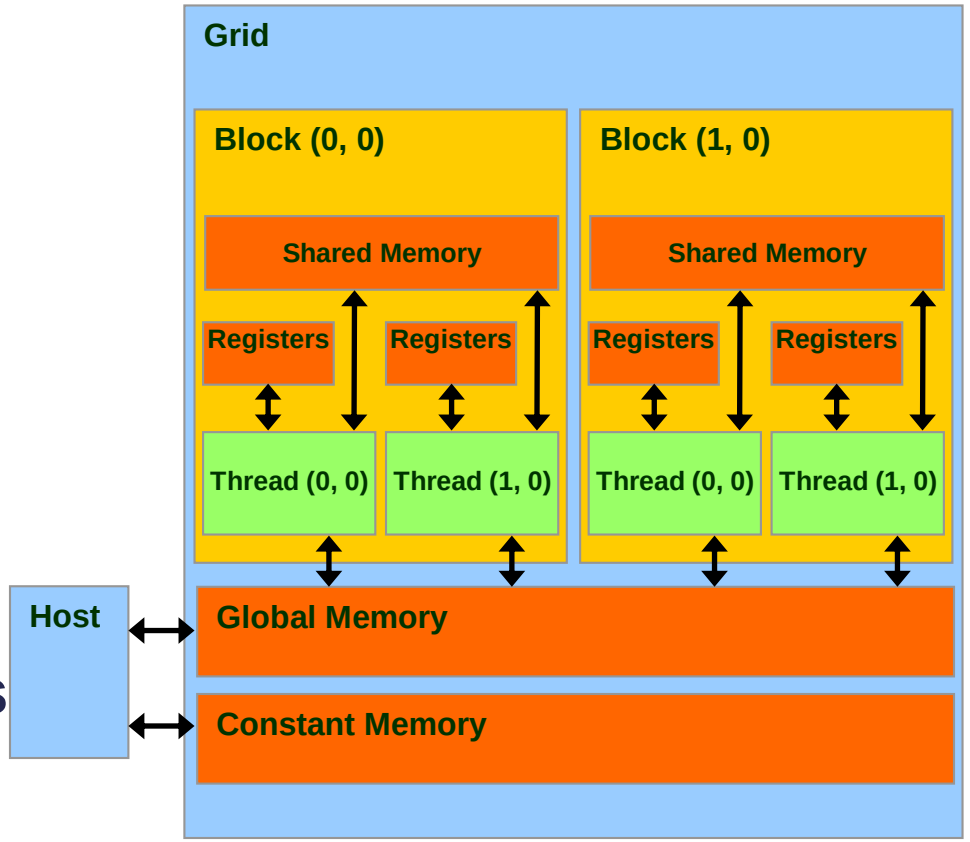
Exercise

Simple exercise (register and shared memory not considered)

- Assume a CUDA device allowing 8 blocks, 1024 threads per SM and 512 thread in each block
- For matrix multiplication, should we use 8x8, 16x16, or 32x32 thread blocks?
  - The **16x16** blocks give 256 threads per block.
  - This means that each SM can take 1024/256 = 4 blocks.
  - This is within the 8-block limitation.
  - **Good configuration**:
    - full thread capacity in each SM and the
    - maximal number of warps for scheduling around the long-latency oper.
  - The **32x32** blocks exceed the limitation of up to 512 threads per block

Programmer View of CUDA Memories

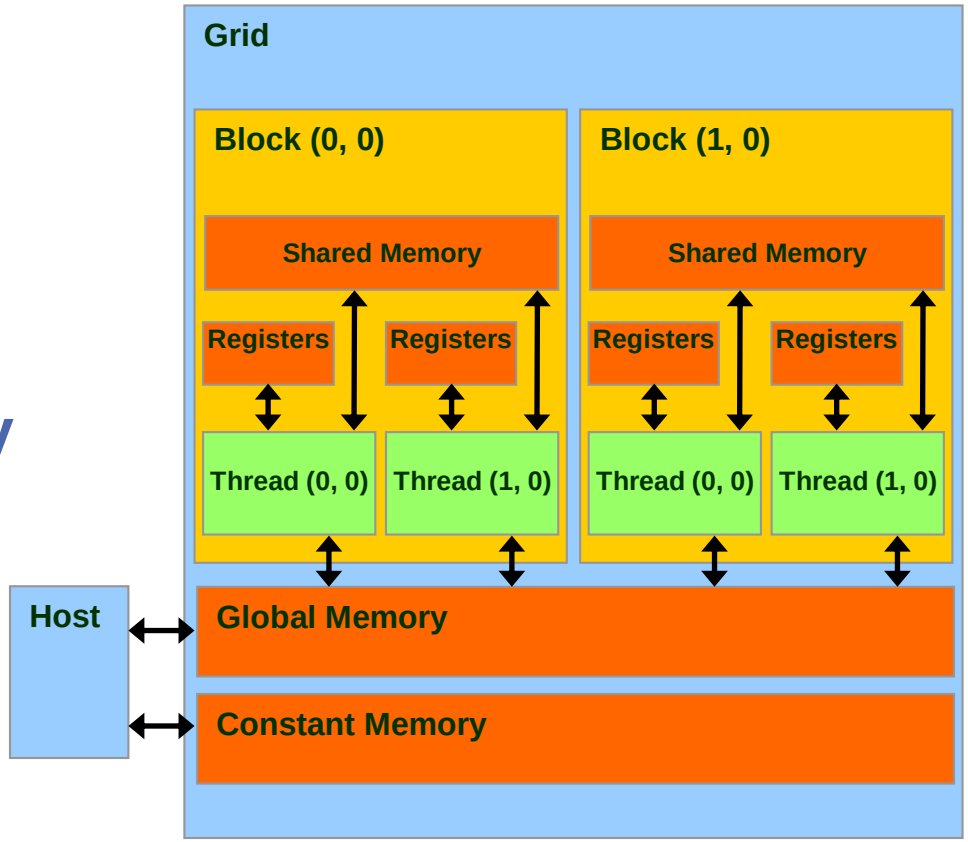
- At the bottom of the figure, we see global memory and constant memory
- These types of memory can be written (W) and read (R) by the host by calling API functions
- The **constant memory** supports short-latency, high-bandwidth, **read-only access by the device** when all threads simultaneously access the same location



Programmer View of CUDA Memories

Device code can:

- R/W per-thread **registers**
- R/W per-thread **local memory**
- R/W per-block **shared memory**
- R/W per-grid **global memory**
- Read-only per-grid **constant memory**

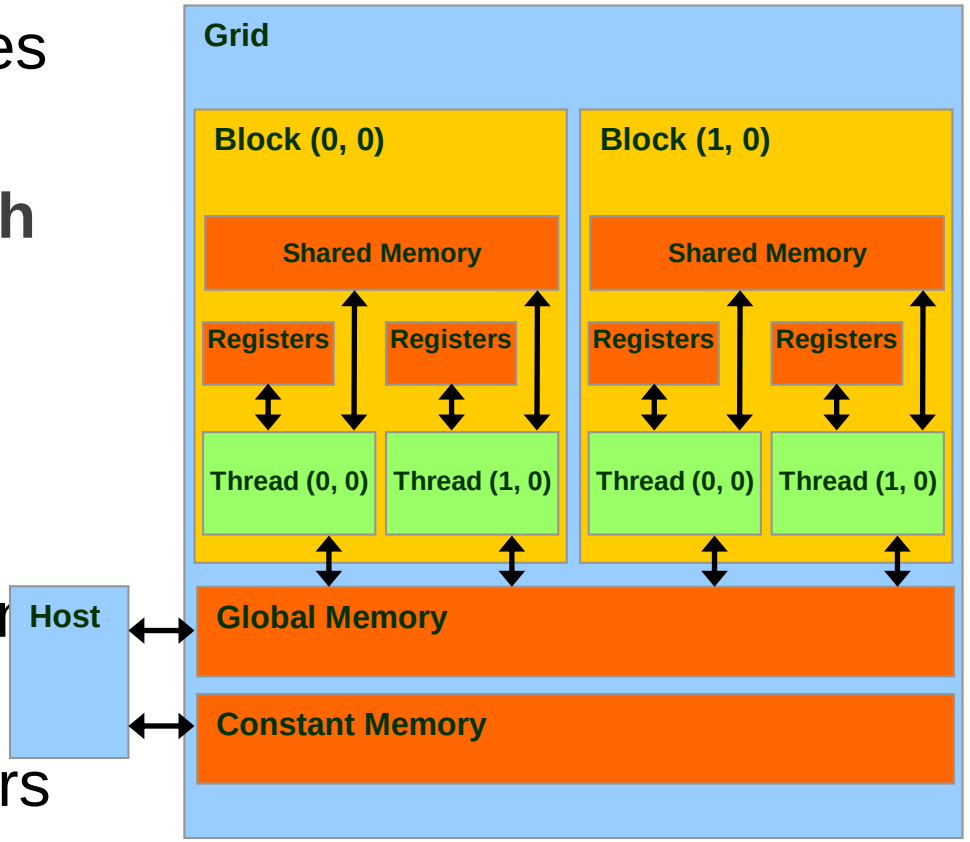


Host code can:

- Transfer data to/from per-grid global and constant memories

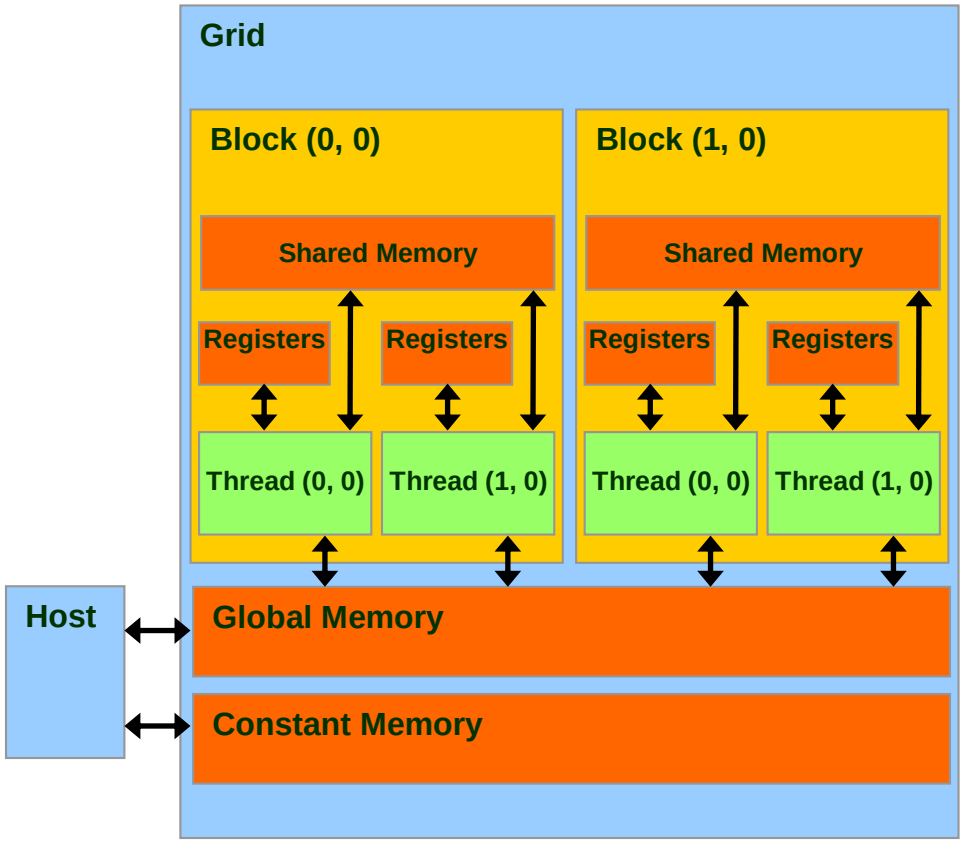
Programmer View of CUDA Memories

- **Registers** and **shared memory** are on-chip memories
- Variables on these memories can be **accessed at very high speed** in a highly parallel manner
- **Registers** are allocated to individual threads and each thread can only access its own registers
- A kernel function uses registers to hold frequently accessed variables private to each thread



Programmer View of CUDA Memories

- **Registers** and **shared memory** are on-chip memories
- **Shared memory** is allocated to thread blocks;
- **all threads in a block** can access variables in the **shared memory** locations allocated to the block
- Shared memory is used by threads to cooperate by sharing their input data and the intermediate results



Variables

- Table presents the CUDA syntax for declaring program variables into the various types of device memory
- Each declaration gives to CUDA variable:
  - A **scope** identifies the range of threads that can access the variable: single thread only, all threads of a block, or all threads of all grids
  - A **lifetime** specifies the portion of the program's execution duration when the variable is available for use: either within a kernel's invocation or throughout the entire application

Variable declaration	Memory	Scope	Lifetime
Automatic Variables	register	thread	kernel
`__device__ __shared__ int SharedVar;`	shared	block	kernel
`__device__ int GlobalVar;`	global	grid	application
`__device__ __constant__ int ConstantVar;`	constant	grid	application

A motivating example

- Lets assume that:
  - We have a kernel that requires 48 registers per thread
  - Target platform is a GTX 580 card (CC 2.0, 16SMs, 32k registers/SM)
  - Execution configuration is a grid of 4x5x3 blocks, each 100 threads
- Each block requires 100*48=4800 registers
- The grid is made of 4*5*3 = 60 blocks that need to be distributed to the 16 SMs of the card

A motivating example

- There will be 12 SMs that will receive 4 blocks and 4 SMs that will receive 3 blocks → Inefficient
- Additionally, each of the 100-thread blocks would be split into $\lceil \frac{100}{warpSize} \rceil = \lceil \frac{100}{32} \rceil = 4$ warps
- The first three warps would have 32 threads and the last would have 4 threads!
- So during the execution of the last warp of each block

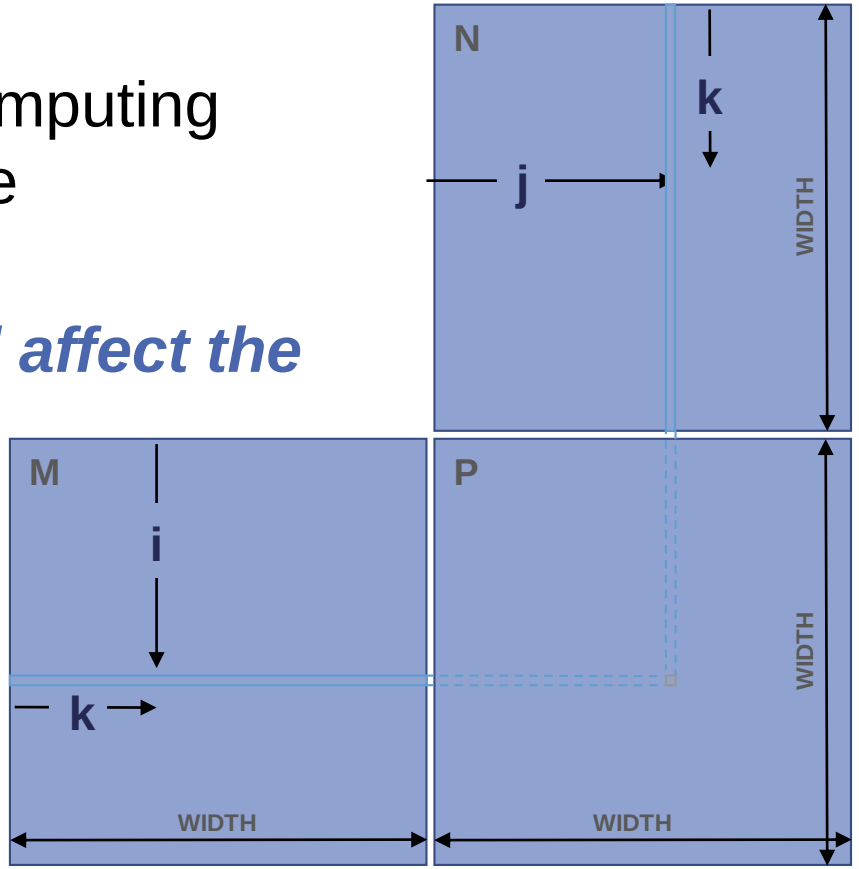
$$\frac{32 - 4}{32} = 87.5\%$$

of the SPs will be idle

MATRIX MULTIPLICATION EXAMPLE

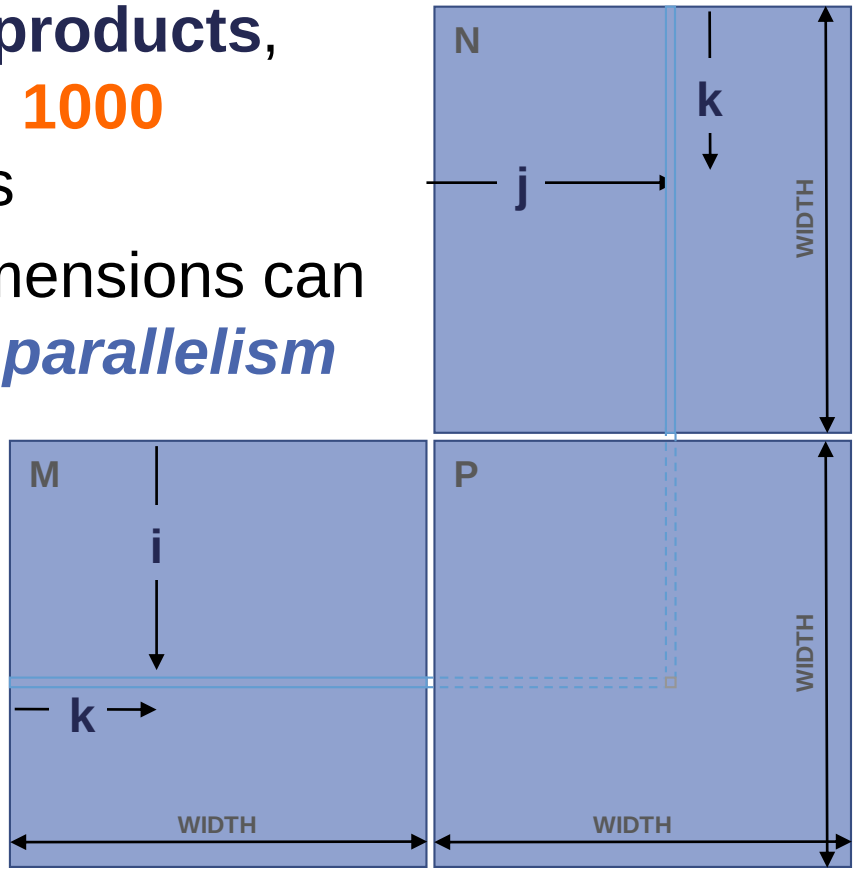
Matrix multiplication

- Each element of the product matrix P is generated by performing a **dot product** between a row of input matrix M and a column of input matrix N: **P = M x N**
- The dot product operations for computing different matrix P elements can be **simultaneously** performed
- ***None of these dot products will affect the results of each other***



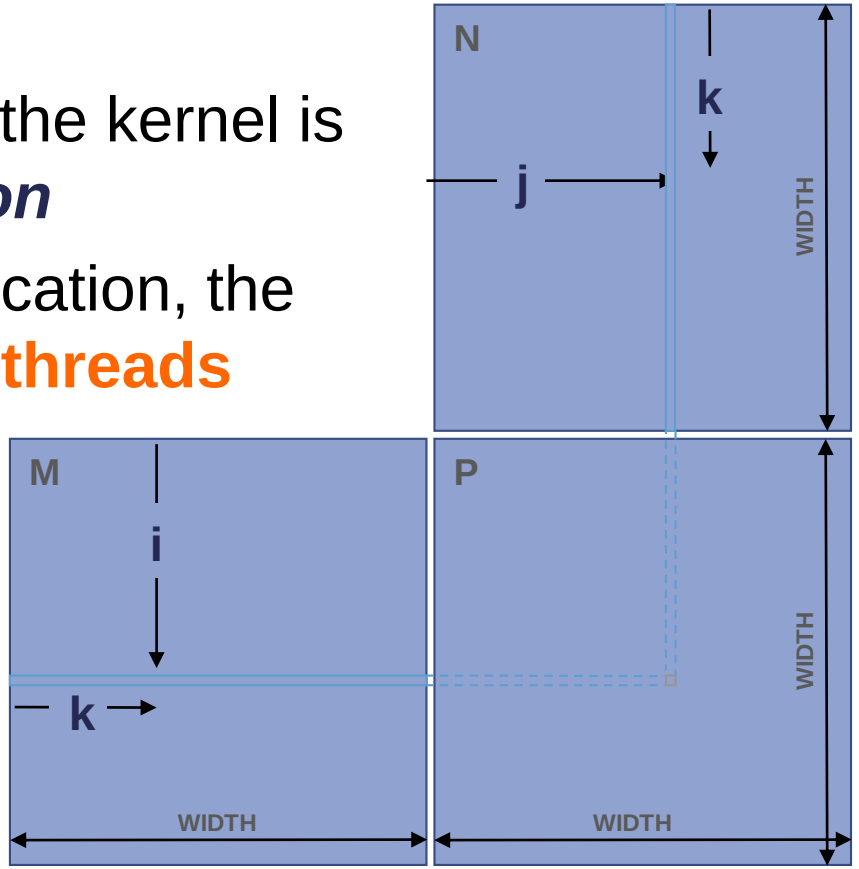
Matrix multiplication

- For large matrices, the number of dot products can be very large
- Example, a **1000 x 1000 matrix multiplication** has **1,000,000 independent dot products**, each involving **1000 multiply** and **1000 accumulate** arithmetic operations
- ***Matrix multiplication*** of large dimensions can have ***very large amount of data parallelism***



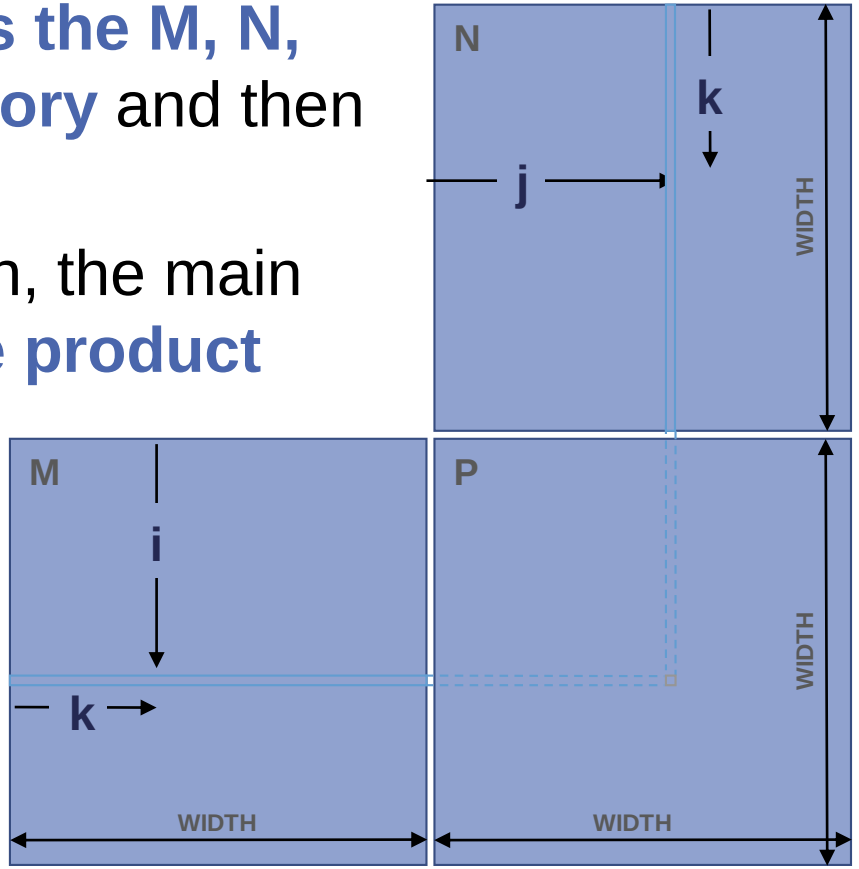
Matrix multiplication

- The entire **matrix multiplication computation** can be implemented as a **kernel**
- **Each thread** is used to compute **one element** of **output matrix P**
- The ***number of threads*** used by the kernel is a function of the ***matrix dimension***
- For a **1000 x 1000 matrix** multiplication, the kernel would generate **1,000,000 threads** when it is invoked



Matrix multiplication

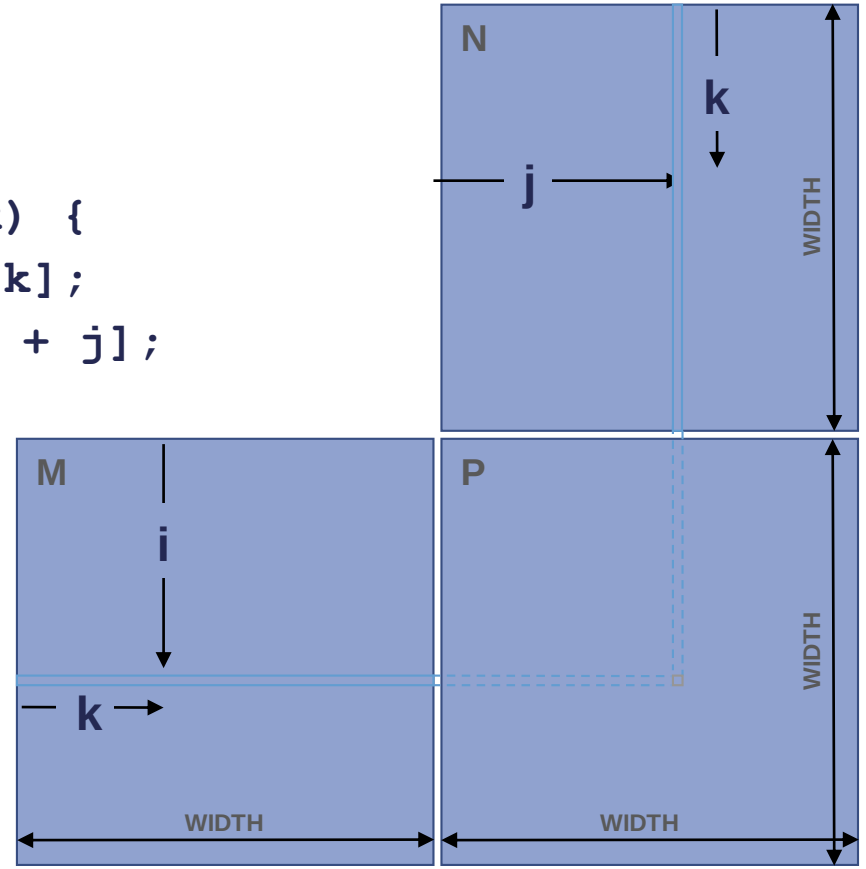
- Assume that the **matrices are square** in shape, and the dimension of each matrix is specified by the parameter **WIDTH**
- The **main program** first **allocates the M, N, and P matrices** in the **host memory** and then performs **I/O to read in M and N**
- **After** completing the multiplication, the main function performs **I/O to write the product matrix P and to free memory**



Matrix multiplication

CPU-only matrix multiplication function

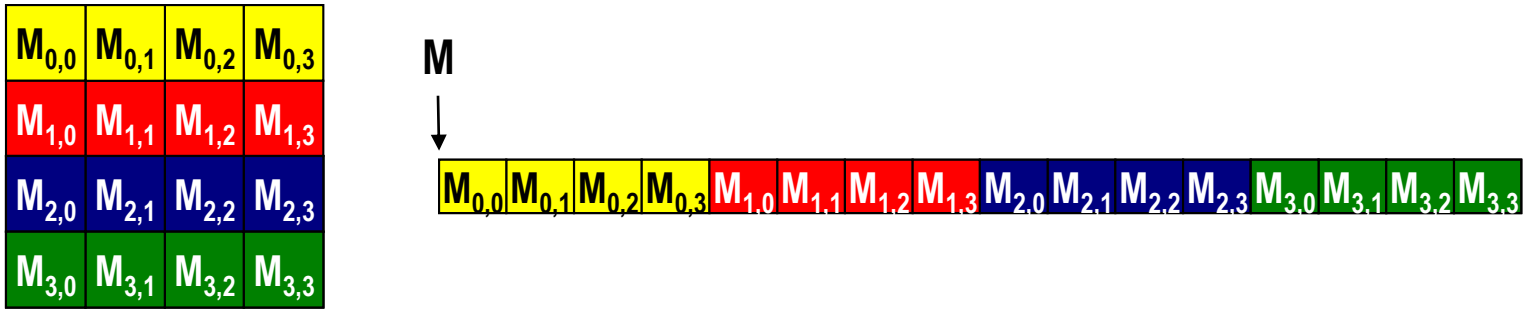
```
• void MatrixMulOnHost(float* M, float* N,
  float* P, int Width)
{
for (int i = 0; i < Width; ++i)
  for (int j = 0; j < Width; ++j) {
     float sum = 0;
     for (int k = 0; k < Width; ++k) {
          float a = M[i  * width + k];
              float b = N[k * width + j];
              sum +=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



Matrix multiplication

- The index used for accessing the M matrix in the innermost loop is **i * Width + k**

- The M matrix elements are placed into the system memory according to the row-major convention:
  - All elements of a row are placed into **consecutive memory locations**
  - The rows are then placed one after another



Matrix multiplication

- To port the matrix multiplication function into CUDA, we can modify the MatrixMultiplication() function to **move the bulk of the calculation to a CUDA device**

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
    int size = Width * Width * sizeof(float);
     float* Md, Nd, Pd;
   …
1.// Allocate device memory for M, N, P and
       // load M, N to device memory
2.// Kernel invocation code to have the device to perform
       // the actual matrix multiplication
3.// copy P from the device
          // Free device matrices
     }
```

Matrix multiplication

- To port the matrix multiplication function into CUDA, we can modify the MatrixMultiplication() fu of the calculation to a CUDA device

```
void MatrixMulOnDevice(float* M, fl                    dth)
{
    int size = Width * Wi
     float* Md, Nd
   …
1.// Allocate device memory for M, N, P and
       // load M, N to device memory
2.// Kernel invocation code to have the device to perform
       // the actual matrix multiplication
3.// copy P from the device
         // Free device matrices
     }
```

Part 1

- **allocates** device (GPU) memory to hold copies of the M, N, and P matrices,

- **copies** these matrices over to the device memory

Matrix multiplication

- To port the matrix multiplication function into CUDA, we can modify the MatrixMultiplication() fu ulk of the calculation to a CUDA device

```
void MatrixMulOnDevice(float* M, fl                t Width)
{
    int size = Width * Width *
     float* Md, Nd, Pd;
   …
1.// Allocate ice memory for M, N, P and
       // load M, N to device memory
2.// Kernel invocation code to have the device to perform
       // the actual matrix multiplication
3.// copy P from the device
         // Free device matrices
     }
```

Part 2

- invokes a kernel that launches parallel execution of the actual matrix multiplication on the device

Matrix multiplication

- To port the matrix multiplication function into CUDA, we can modify the MatrixMultiplication() fun… …lk of the calculation to a CUDA device

```
void MatrixMulOnDevice(float* M, flo…                    Width)
{
    int size = Width * Width *
     float* Md, Nd, Pd;
   …
1.// Allocate devi…emory for M, N, P and
       // load…, N to device memory
2.// Kern… invocation code to have the device to perform
       // the actual matrix multiplication
3.// copy P from the device
          // Free device matrices
     }
```

Part 3

- copies the product matrix P from the device memory back to the host memory

Matrix multiplication

Assume **M, N** and **P** are on the **host** and

Md, Nd and **Pd** on **device**

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
    int size = Width * Width * sizeof(float);
     float* Md, Nd, Pd;
    …
1.  // Allocate and Load M, N to device memory
      cudaMalloc(&Md, size);
      cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
      cudaMalloc(&Nd, size);
      cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);
      // Allocate P on the device
      cudaMalloc(&Pd, size);
```

Matrix multiplication

Assume M, N and P are on the **host** ar
device

```
void MatrixMulOnDevice(float* M, float
{
    int size = Width * Width * sizeof(f
     float* Md, Nd, Pd;
    …
1. // Allocate and Load M, N to device mem
     cudaMalloc(&Md, size);
     cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
     cudaMalloc(&Nd, size);
     cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);
     // Allocate P on the device
     cudaMalloc(&Pd, size);
```

The **two symbolic constants,** `cudaMemcpyHostToDevice` and `cudaMemcpyDeviceToHost` are **predefined constants** of the CUDA programming environment, **recognized by** `cudaMemcpy`

Matrix multiplication

```
2.    // Kernel invocation code – to be shown later
      …
3.     // Read P from the device
       cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
        // Free device matrices
       cudaFree(Md);
        cudaFree(Nd);
        cudaFree(Pd);
      }
```

-The product data is copied from device memory to host memory so the value will be available to main() by a call to the `cudaMemcpy()` function

- Then Md, Nd, and Pd are freed from the device memory by calls to the `cudaFree()` functions

Matrix multiplication

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int
  Width)
{
// 2D Thread ID
int tx = threadIdx.x;
int ty = threadIdx.y;
// Pvalue stores the Pd element that is computed by the thread
float Pvalue = 0;
for (int k = 0; k < Width; ++k)
   {
   float Mdelement = Md[ty * Md.width + k];
   float Ndelement = Nd[k * Nd.width + tx];
   Pvalue += Mdelement * Ndelement;
   }
// Write the matrix to device memory each thread writes one element
Pd[ty * Width + tx] = Pvalue;
}
```

Matrix multiplication

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd,
  int Width)
{
// 2D Thread ID
int tx = threadIdx.x;
int ty = threadIdx.y;
// Pvalue stores the Pd eleme
float Pvalue = 0;
for (int k = 0; k < Width; ++k)
   {
   float Mdelement = Md[ty * Md.widt
   float Ndelement = Nd[k * Nd.width
   Pvalue += Mdelement * Ndelement;
   }
// Write the matrix to device memor                    ment
Pd[ty * Width + tx] = Pvalue;
}
```

The CUDA-specific keyword `__global__` in front of the declaration of `MatrixMulKernel()` indicates that:
- the function is a kernel
- it can be called from a host functions to generate a grid of threads on a device

Matrix multiplication

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
// 2D Thread ID
int tx = threadIdx.x;
int ty = threadIdx.y;
// Pvalue stores the Pd el
float Pvalue = 0;
for (int k = 0; k < Width; ++k)
   {
   float Mdelement = Md[ty * Md.wid
   float Ndelement = Nd[k * Nd.width
   Pvalue += Mdelement * Ndelement;
   }
// Write the matrix to device memory
Pd[ty * Width + tx] = Pvalue;
}
```

- The keywords `threadIdx.x` and `threadIdx.y` refer to the thread indices of a thread
- The original loop variables i and j are now replaced with `threadIdx.x` and `threadIdx.y`
- The CUDA threading hardware generates all of the `threadIdx.x` and `threadIdx.y` values for each thread (instead of the loop increment the values of i and j for loop iteration)

Matrix multiplication

- Limitation of this simple code is the **size of matrices**: 16x16

- Infact the kernel function does not use **`blockIdx`**
- Then, we are limited to using **only one block of threads**
- Even if we used more blocks, threads from different blocks would calculate the same Pd element if they have the same threadIdx value

- The code can only calculate a product matrix of up to 512 elements → infact a thread block can have only up to 512 threads and each thread calculates one element of the matrix

Matrix multiplication

- For square matrices, 32x32 requires more than 512 threads per block → then we are limited to 16x16

- This is obviously **not acceptable**
- The product matrix must have millions of elements in order to have a sufficient amount of data parallelism to benefit from execution on a device

- Now we revise the matrix multiplication kernel function using **multiple blocks**

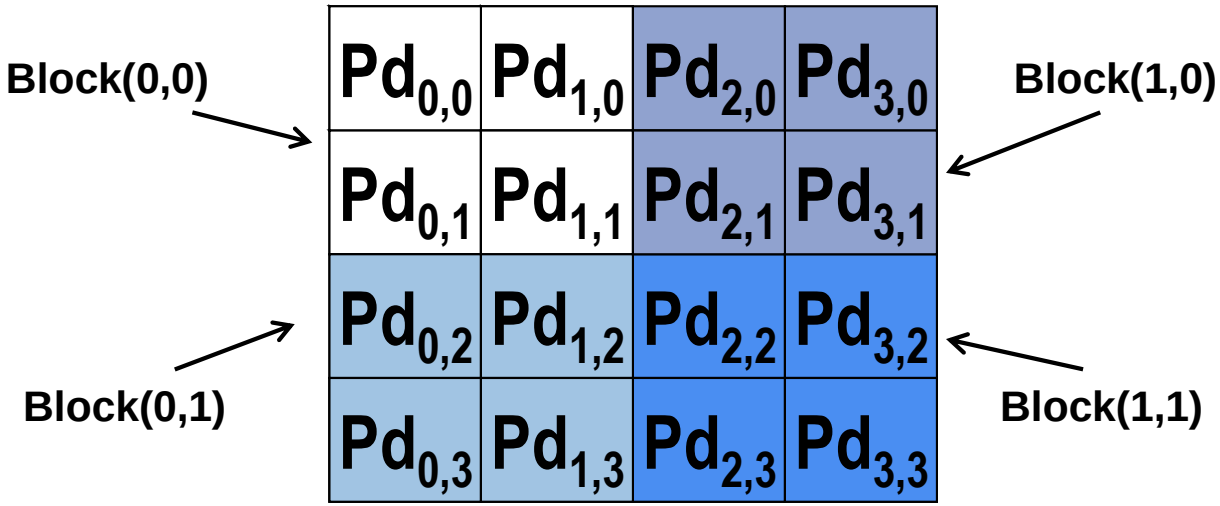
Matrix multiplication

- In order to accommodate larger matrices, we need to use **multiple thread blocks**
- Conceptually, **we break Pd into square tiles**
- All the Pd **elements of a tile** are computed by a **block of threads**
- By keeping the dimensions of these Pd tiles small, we keep the total number of threads in each block under **512**, the **maximal allowable block size**
- In the following, we abbreviate:
  - **`threadIdx.x`** and **`threadIdx.y`** as **`tx`** and **`ty`**
  - **`blockIdx.x`** and **`blockIdx.y`** as **`bx`** and **`by`**

Matrix multiplication

- Consider a very small matrix - 4x4 - and a very small TILE_WIDTH value - 2 - and divide the matrix into **4 tiles**

- We create **blocks** organized into **2x2 arrays of threads**
- **Each block calculates 4 Pd elements**



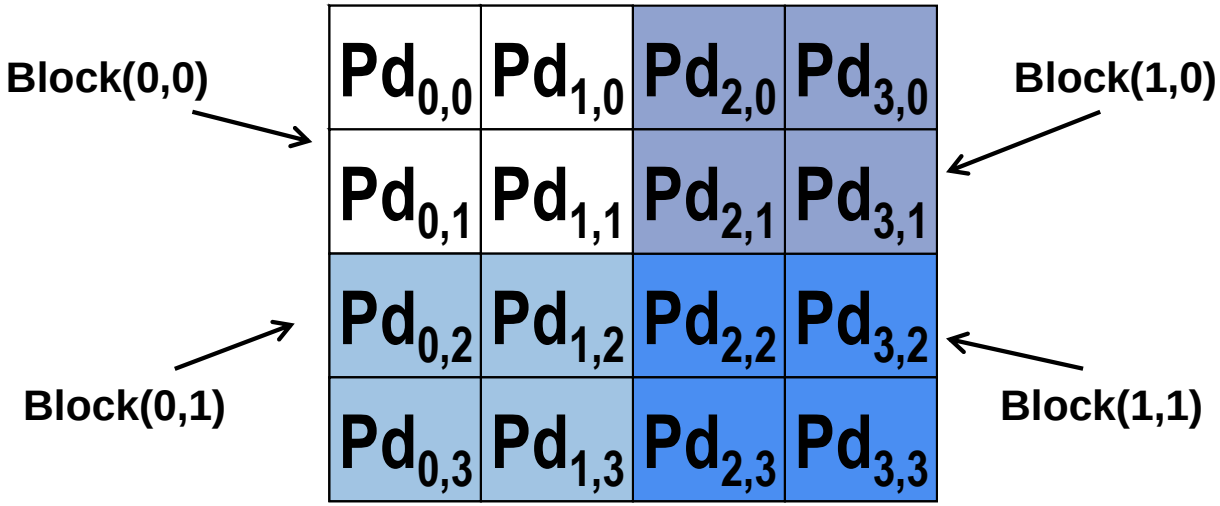
Matrix multiplication

- **thread (0, 0) of block (0, 0)** calculates $Pd_{0,0}$
- **thread (0, 0) of block (1, 0)** calculates $Pd_{2,0}$

The Pd element calculated by thread (0, 0) of block (1, 0) can be computed by:

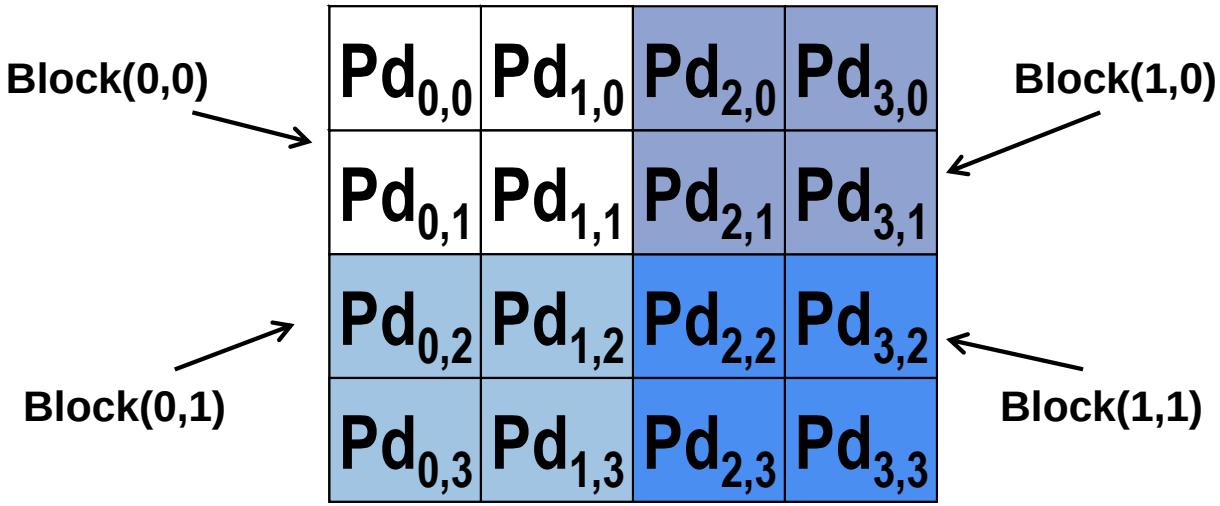
Pd[bx* TILE_WIDTH + tx] [by* TILE_WIDTH + ty] =

=Pd[1*2 + 0][0*2 + 0] = Pd[2][0]



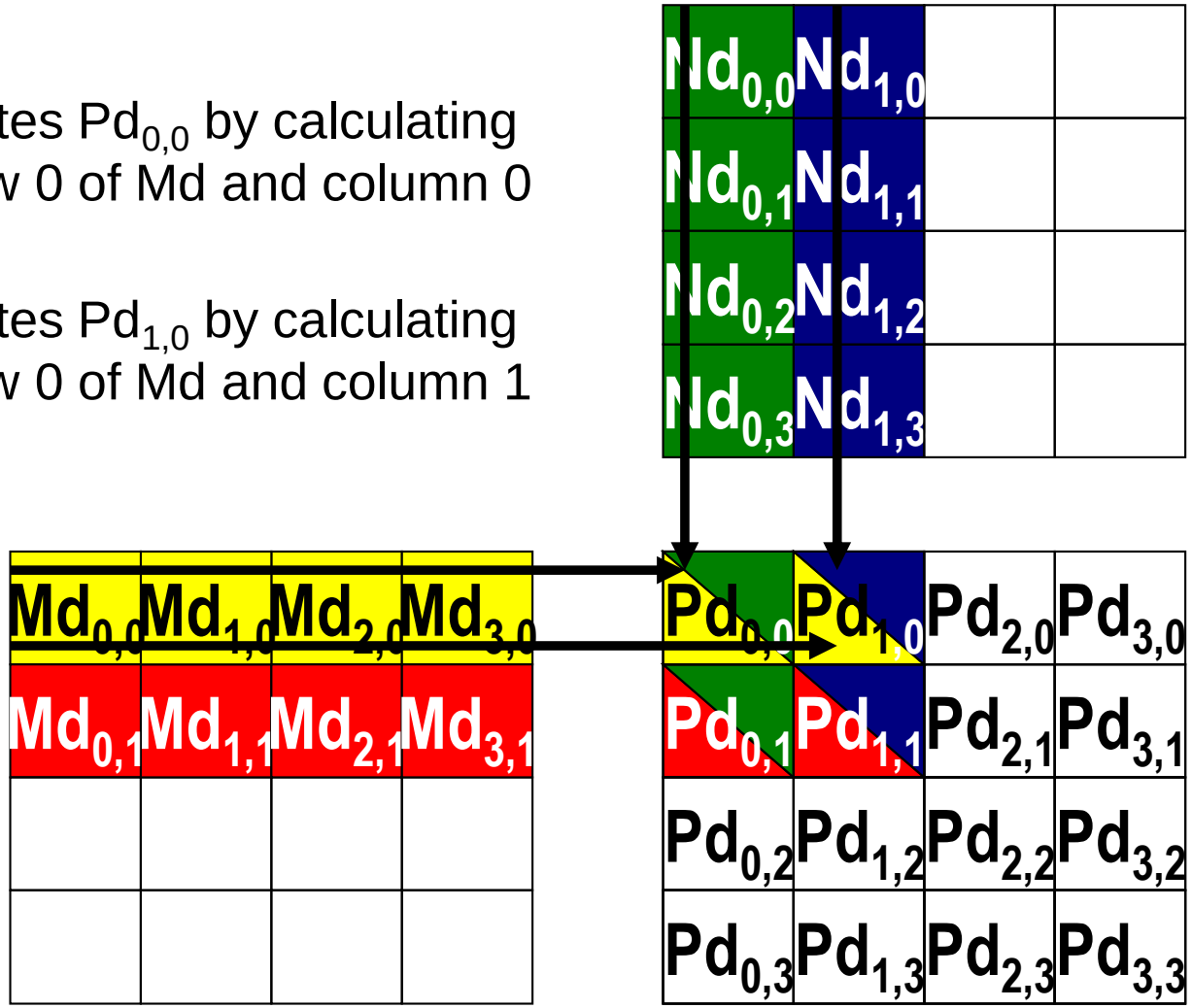
Matrix multiplication

- We also need the **row index y** of Md and the **column index x** of Nd for input values
- The **row index** of Md used by thread (tx, ty) of block (bx, by) is **(by*TILE_WIDTH + ty)**
- The **column index** of Nd used by the same thread is **(bx*TILE_WIDTH + tx)**



Matrix multiplication

- Threads in block (0, 0) produce four dot products:
  - Thread (0, 0) generates $Pd_{0,0}$ by calculating the dot product of row 0 of Md and column 0 of Nd
  - Thread (1, 0) generates $Pd_{1,0}$ by calculating the dot product of row 0 of Md and column 1 of Nd
  - …
- The arrows of $Pd_{0,0}$, and $Pd_{1,0}$ shows the row and column used for generating their result value



Matrix multiplication

Revised matrix multiplication kernel function **with blocks**

```
__global__ void MatrixMulKernel(float* Md, float* Nd,
    float* Pd, int Width)
{
// Calculate the row index of the Pd element and M
int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
// Calculate the column index of Pd and N
int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

float Pvalue = 0;
// each thread computes one ele
   matrix
for (int k = 0; k < Width; ++k)
  Pvalue += Md[Row*Width+k] * Nd[k*W

  Pd[Row*Width+Col] = Pvalue;
```

Each thread uses its `blockIdx` and `threadIdx` values to identify the row index - Row - and the column index - Col - of the Pd element

Matrix multiplication

- Revised matrix multiplication kernel fun

```
__global__ void MatrixMulKernel(floa
    Nd, float* Pd, int Width)
{
// Calculate the row index of the Pd
int Row = blockIdx.y*TILE_WIDTH + th
// Calculate the column index of Pd
int Col = blockIdx.x*TILE_WIDTH

float Pvalue = 0;
// each thread computes  element o
    sub-matrix
for (int k = 0; k < Width; ++k)
  Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];

  Pd[Row*Width+Col] = Pvalue;
```

Each thread performs:
- a dot product on the row of Md and column of Nd to generate the value of the Pd element
- eventually writes the Pd value to the appropriate global memory location

Matrix multiplication

- This kernel can handle matrices of up to 16 x 65,535 elements in each dimension
- In the situation where matrices larger than this new limit are to be multiplied, one can divide the Pd matrix into submatrices of a size permitted by the kernel
- All blocks can run in parallel with each other and will fully utilize parallel execution resources

Matrix multiplication

- **Revised host code** to be used in the **MatrixMultiplication()** **to launch the revised kernel MatrixMulKernel()** with multiple blocks
- Note that the **dimGrid** is **Width/TILE_WIDTH** for both the x dimension and the y dimension

```
// Set up the execution configuration
dim3 dimGrid(Width/TILE_WIDTH,Width/TILE_WIDTH)
dim3 dimBlock(TILE_WIDTH,TILE_WIDTH)

// launch the device computation thread
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md,Nd,Pd,Width);
```

Matrix multiplication

Memory access: global vs shared

- The table shows the global memory accesses done by all threads in block(0,0)
- The threads are listed in the horizontal direction, with the time of access increasing downward in the vertical direction

$P_{0,0}$ $thread_{0,0}$	$P_{1,0}$ $thread_{1,0}$	$P_{0,1}$ $thread_{0,1}$	$P_{1,1}$ $thread_{1,1}$
$M_{0,0} * N_{0,0}$	$M_{0,0} * N_{1,0}$	$M_{0,1} * N_{0,0}$	$M_{0,1} * N_{1,0}$
$M_{1,0} * N_{0,1}$	$M_{1,0} * N_{1,1}$	$M_{1,1} * N_{0,1}$	$M_{1,1} * N_{1,1}$
$M_{2,0} * N_{0,2}$	$M_{2,0} * N_{1,2}$	$M_{2,1} * N_{0,2}$	$M_{2,1} * N_{1,2}$
$M_{3,0} * N_{0,3}$	$M_{3,0} * N_{1,3}$	$M_{3,1} * N_{0,3}$	$M_{3,1} * N_{1,3}$

Access order ↓

Matrix multiplication

- Each thread accesses **4 elements of Md** and **4 elements of Nd** during its execution
- there is a significant **overlap** of the Md and Nd accesses:
  - **thread(0,0)** and **thread(1,0)** both access $Md_{1,0}$ as well as the rest of row 0 of Md
  - **thread(1,0)** and **thread(1,1)** both access $Nd_{1,0}$ as well as the rest of column 1 of Nd

$P_{0,0}$ $thread_{0,0}$	$P_{1,0}$ $thread_{1,0}$	$P_{0,1}$ $thread_{0,1}$	$P_{1,1}$ $thread_{1,1}$
$M_{0,0} * N_{0,0}$	$M_{0,0} * N_{1,0}$	$M_{0,1} * N_{0,0}$	$M_{0,1} * N_{1,0}$
$M_{1,0} * N_{0,1}$	$M_{1,0} * N_{1,1}$	$M_{1,1} * N_{0,1}$	$M_{1,1} * N_{1,1}$
$M_{2,0} * N_{0,2}$	$M_{2,0} * N_{1,2}$	$M_{2,1} * N_{0,2}$	$M_{2,1} * N_{1,2}$
$M_{3,0} * N_{0,3}$	$M_{3,0} * N_{1,3}$	$M_{3,1} * N_{0,3}$	$M_{3,1} * N_{1,3}$

Access order ↓

Matrix multiplication

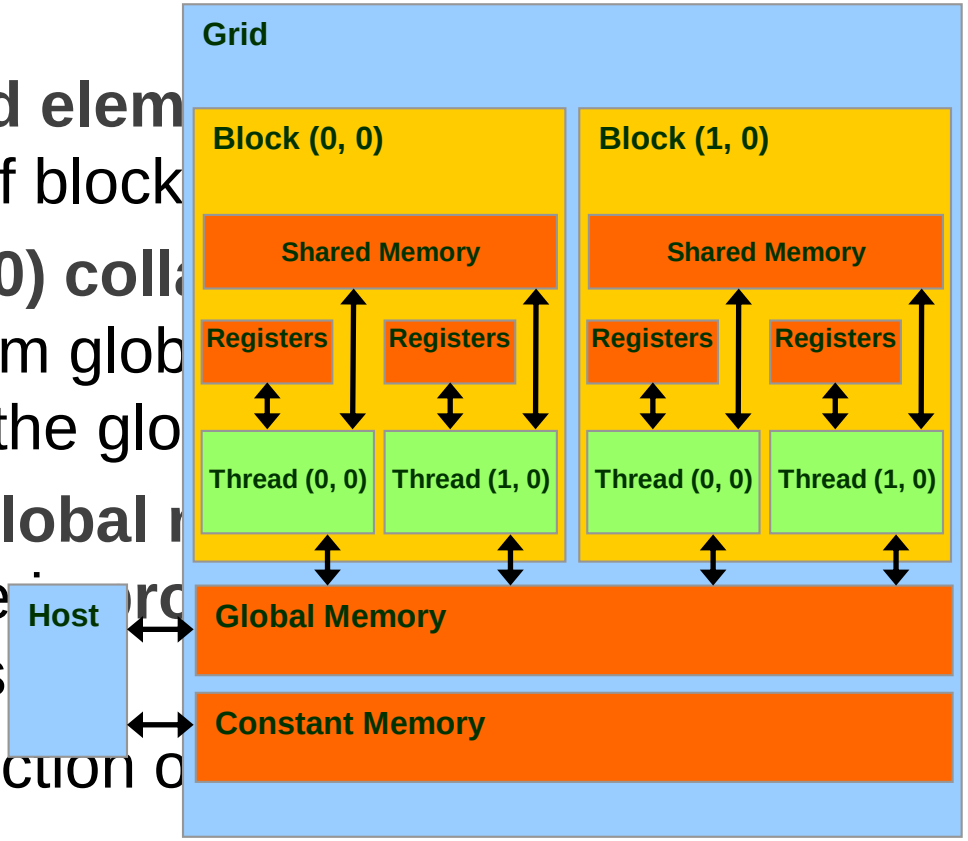
- The kernel is written so both thread(0,0) and thread(1,0) access these Md row 0 elements from the **global memory**

- In general, **every Md and Nd element** is accessed exactly **twice** during the execution of block(0,0)
- If **thread(0,0) and thread(1,0) collaborate** so that Md elements are only loaded from global memory once, the total number of accesses to the global memory reduced by half
- The potential **reduction in global memory traffic** in the matrix multiplication example is **proportional** to the **dimension of the blocks** used
- NxN blocks → potential reduction of global memory is N

Matrix multiplication

- The kernel is written so both thread(0,0) and thread(1,0) access these Md row 0 elements from the **global memory**

- In general, **every Md and Nd elem**
  twice during the execution of block
- If **thread(0,0) and thread(1,0) colla**
  elements are only loaded from glob
  total number of accesses to the glo
- The potential **reduction in global m**
  matrix multiplication example **i**
  dimension of the blocks us
- NxN blocks → potential reduction o



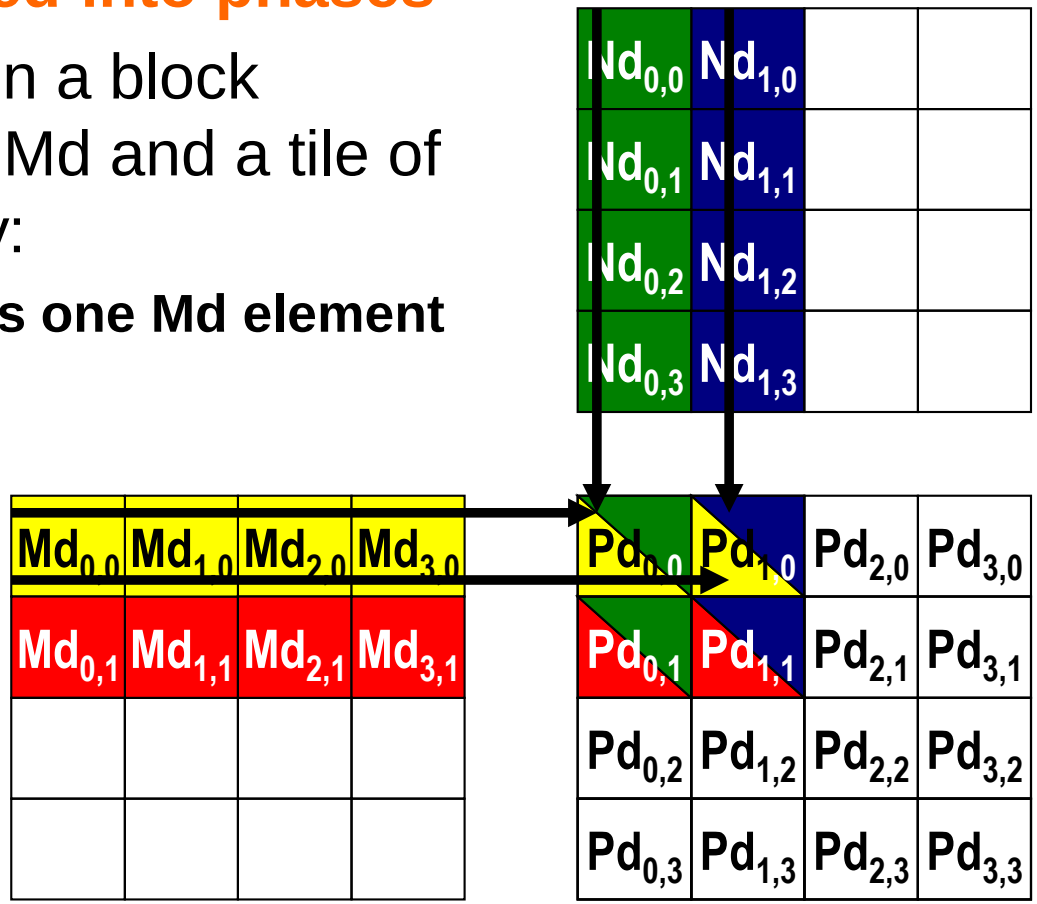
Matrix multiplication

- The multiplication algorithm where threads collaborate to **reduce the traffic to the global memory** exploits the **shared memory**
- Threads collaboratively load Md and Nd elements into the **shared memory** before they individually use these elements in their dot product calculation

- The size of the **shared memory is quite small**
- To no exceed the capacity of the shared memory when loading Md and Nd elements we consider Md and Nd matrices divided into **tiles**

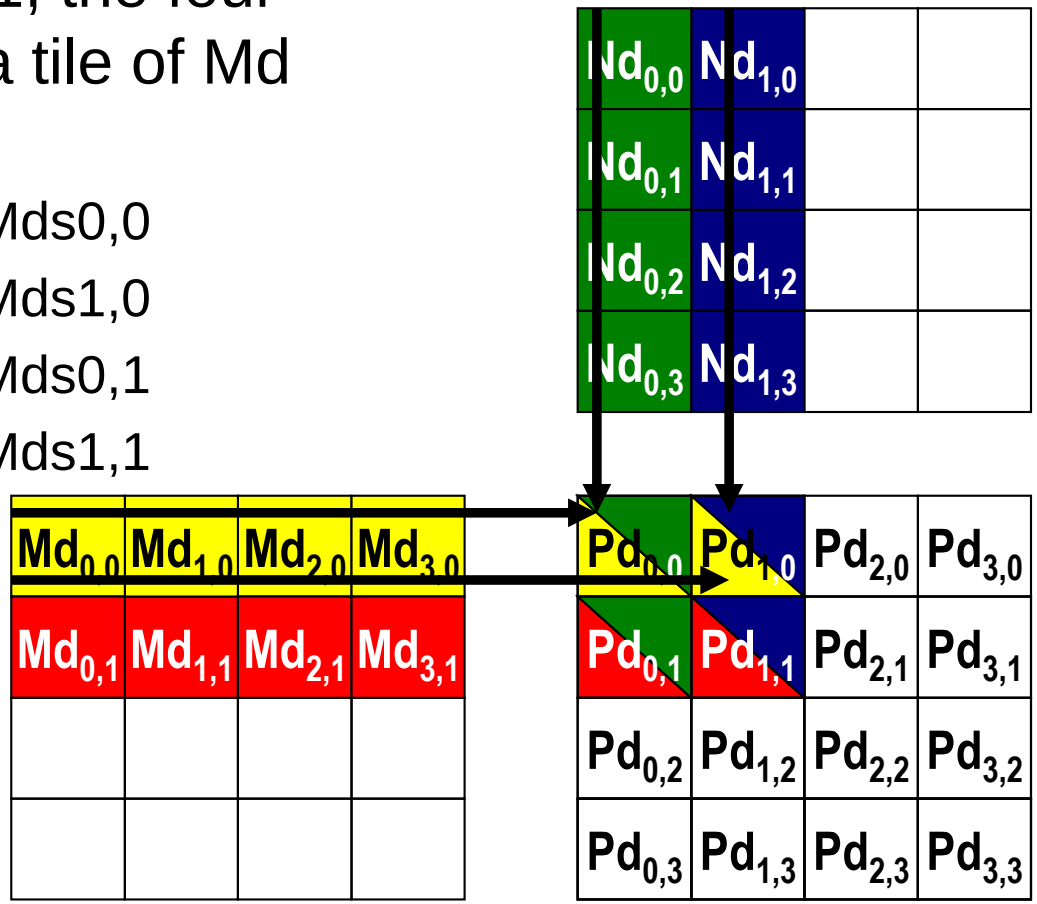
Matrix multiplication

- Md and Nd divided into 2x2 tiles
- The **dot product calculations** performed by each thread are now **divided into phases**
- In each phase, all threads in a block collaborate to load a tile of Md and a tile of Nd into the shared memory:
  - **every thread in a block loads one Md element and one Nd element**



Matrix multiplication

- Activities of threads in block(0,0) (other blocks are the same)
- At the beginning of Phase 1, the four threads of block(0,0) load a tile of Md into shared memory
  - thread(0,0) loads Md0,0 into Mds0,0
  - thread(1,0) loads Md1,0 into Mds1,0
  - thread(0,1) loads Md0,1 into Mds0,1
  - thread(1,1) loads Md1,1 into Mds1,1
- The shared memory array for the **Md** elements is **Mds**, and for the **Nd** elements is **Nds**



Matrix multiplication

- The shared memory array for the **Md** elements is **Mds**, and for the **Nd** elements is **Nds**

	Phase 1			Phase 2		
$T_{0,0}$	**$Md_{0,0}$** ↓ $Mds_{0,0}$	**$Nd_{0,0}$** ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$	**$Md_{2,0}$** ↓ $Mds_{0,0}$	**$Nd_{0,2}$** ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$
$T_{1,0}$	**$Md_{1,0}$** ↓ $Mds_{1,0}$	**$Nd_{1,0}$** ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$	**$Md_{3,0}$** ↓ $Mds_{1,0}$	**$Nd_{1,2}$** ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$
$T_{0,1}$	**$Md_{0,1}$** ↓ $Mds_{0,1}$	**$Nd_{0,1}$** ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{0,1}$	**$Md_{2,1}$** ↓ $Mds_{0,1}$	**$Nd_{0,3}$** ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{0,1}$
$T_{1,1}$	**$Md_{1,1}$** ↓ $Mds_{1,1}$	**$Nd_{1,1}$** ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$	**$Md_{3,1}$** ↓ $Mds_{1,1}$	**$Nd_{1,3}$** ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$

time →

Matrix multiplication

	Phase 1		
$T_{0,0}$	**$Md_{0,0}$** ↓ $Mds_{0,0}$	**$Nd_{0,0}$** ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$
$T_{1,0}$	**$Md_{1,0}$** ↓ $Mds_{1,0}$	**$Nd_{1,0}$** ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$
$T_{0,1}$	**$Md_{0,1}$** ↓ $Mds_{0,1}$	**$Nd_{0,1}$** ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{0,1}$
$T_{1,1}$	**$Md_{1,1}$** ↓ $Mds_{1,1}$	**$Nd_{1,1}$** ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$

- At the beginning of **Phase 1**, the four threads of block(0,0) load a tile of **Md** into the shared memory and a tile of **Nd**
- These values are used in the calculation of the dot product
- **Note that** each value in the shared memory is used **twice**

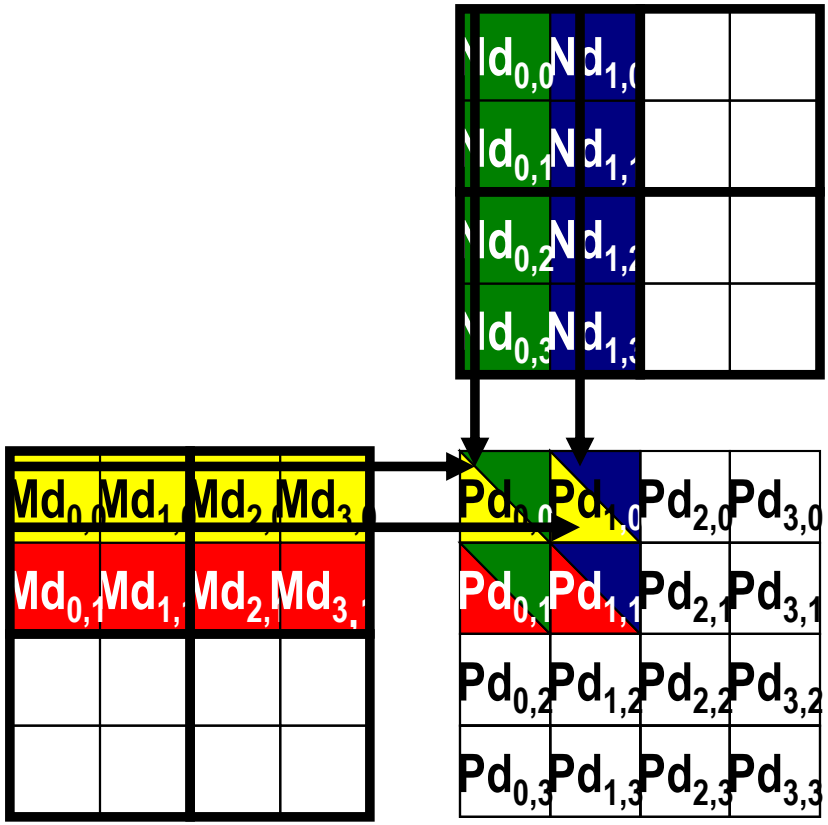
Example

- **$Md_{0,1}$** is loaded by $thread_{0,1}$ into $Mds_{0,1}$ and is used once by $thread_{0,1}$ and once by $thread_{1,1}$
- **$Nd_{1,0}$** is loaded by $thread_{1,0}$ into $Nds_{1,0}$ and is used once by $thread_{1,0}$ and once by $thread_{1,1}$

Matrix multiplication

	Phase2		
$T_{0,0}$	**$Md_{2,0}$** ↓ $Mds_{0,0}$	**$Nd_{0,2}$** ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$
$T_{1,0}$	**$Md_{3,0}$** ↓ $Mds_{1,0}$	**$Nd_{1,2}$** ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$
$T_{0,1}$	**$Md_{2,1}$** ↓ $Mds_{0,1}$	**$Nd_{0,3}$** ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{0,1}$
$T_{1,1}$	**$Md_{3,1}$** ↓ $Mds_{1,1}$	**$Nd_{1,3}$** ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$

- **Phase 2** is similar and it allow to complete the computation
- **Note that** the two phases use the same Mds e Nds.



Matrix multiplication

Tiled matrix multiplication **kernel** using **shared memories**

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float*
    Pd, int Width)
{
1.   __shared__float Mds[TILE_WIDTH][TILE_WIDTH];
2.   __shared__float Nds[TILE_WIDTH][TILE_WIDTH];

3.   int bx = blockIdx.x;  int by = blockIdx.y;
4.   int tx = threadIdx.x; int ty = threadIdx.y;

// Identify the row and column of the Pd element to work on
5.   int Row = by * TILE_WIDTH + ty;
6.   int Col = bx * TILE_WIDTH + tx;
```

Matrix multiplication

```
7.    float Pvalue = 0;
// Loop over the Md and Nd tiles required to compute
   the Pd element
8.    for (int m = 0; m < Width/TILE_WIDTH; ++m) {
// Collaborative loading of Md and Nd tiles into shared
   memory
9.     Mds[ty][tx] = Md[Row*Width+(m*TILE_WIDTH + tx)];
10.    Nds[ty][tx] = Nd[Col+(m*TILE_WIDTH + ty)*Width];
11.     __syncthreads();
12.     for (int k = 0; k < TILE_WIDTH; ++k)
13.         Pvalue += Mds[ty][k] * Nds[k][tx];
14.         synchthreads();
   }
15.  Pd[Row*Width+Col] = Pvalue;
   }
```