ICCS School

Advanced GPU Programming for Science

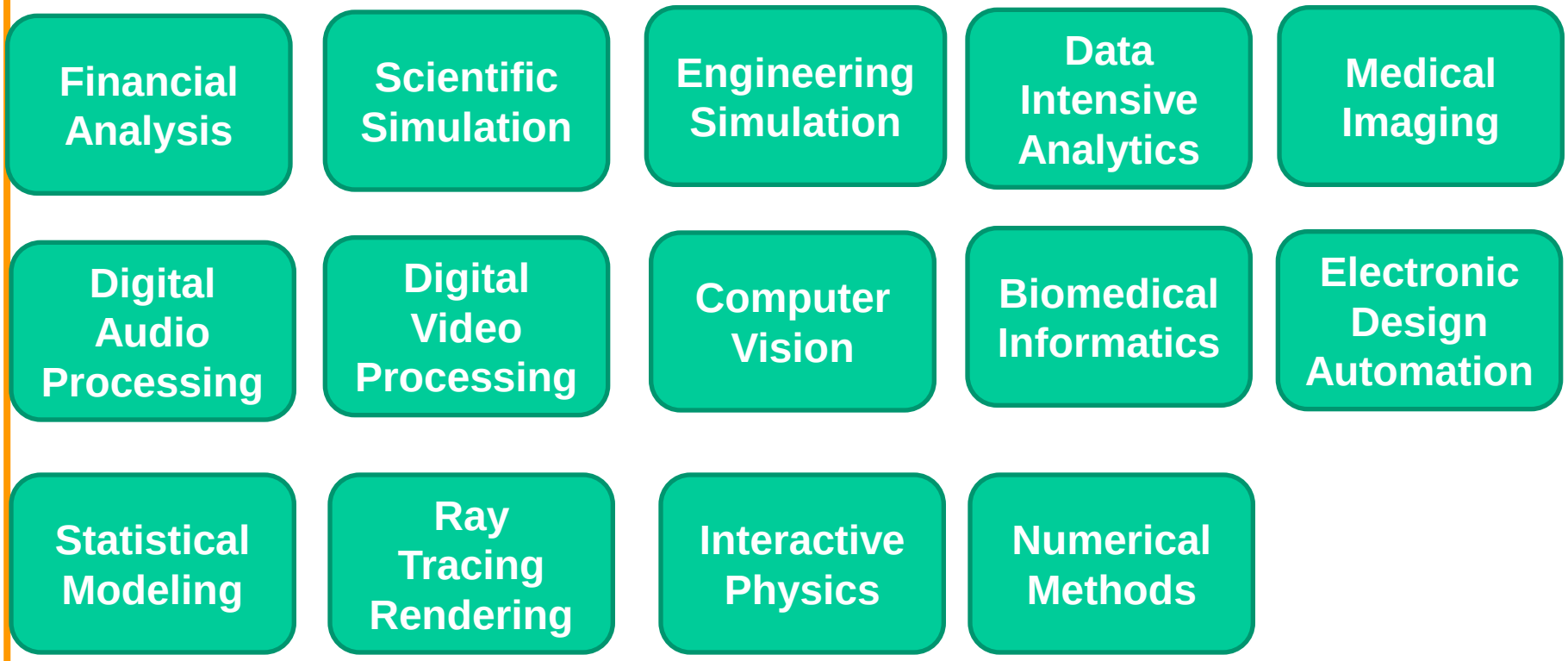
Lecture 1: Introduction to Scalability

Course Objective

- To master the most commonly used algorithm techniques and computational thinking skills needed for many-core GPU programming
  - Especially the simple ones!

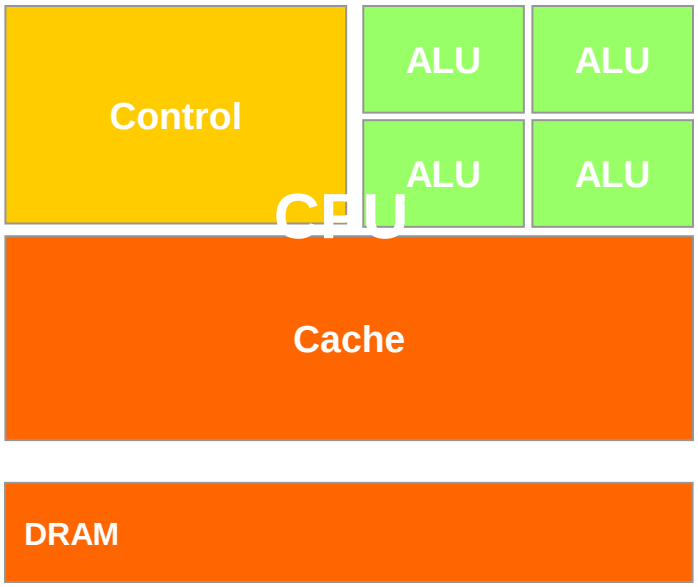
- In particular, to understand
  - Many-core hardware limitations and constraints
  - Desirable and undesirable computation patterns
  - Importance of controlling computational complexity
  - Commonly used algorithm techniques to convert undesirable computation patterns into desirable ones

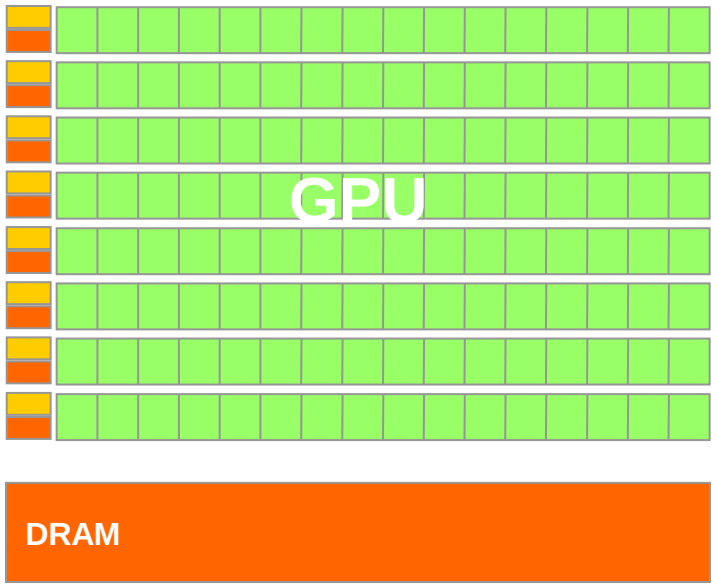
GPU computing is catching on.



- 280 submissions to GPU Computing Gems
  - More than 80 articles included in two volumes

CPUs and GPUs have fundamentally different design philosophies.



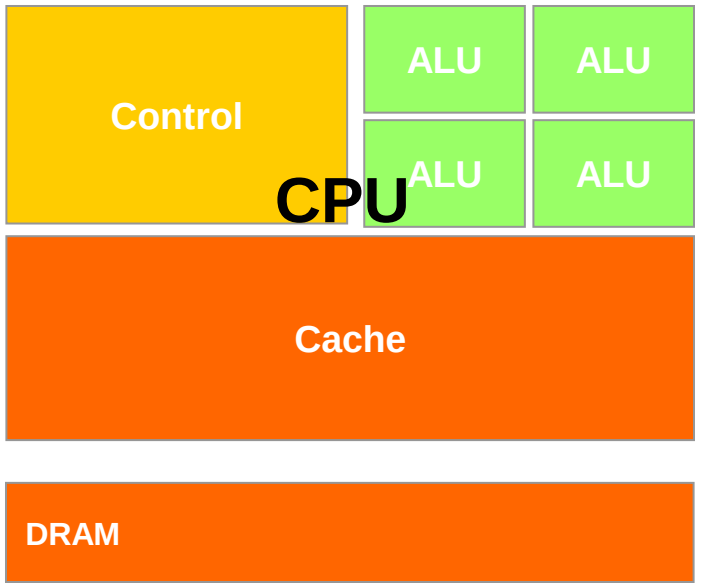


0.4 TFLOPS, 30 GB/s
16-32 threads

2 TFLOPS, 150 GB/s
30K threads

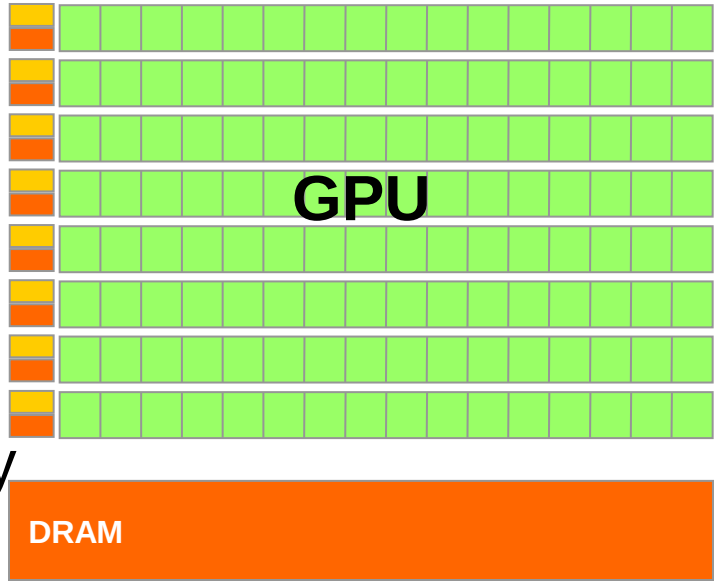
CPUs: Latency Oriented Design

- 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 ALUs
  - 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



Winning Applications Use Both CPU and GPU

- CPUs for sequential parts where latency matters
  - CPUs can be 10+X faster than GPUs for sequential code
- GPUs for parallel parts where throughput wins
  - GPUs can be 10+X faster than CPUs for parallel code

CPUs help the GPUs to overcome load balance, control divergence, and memory bandwidth challenges.

A Common GPU Usage Pattern

- A desirable approach considered impractical
  - Due to excessive computational requirement
  - But demonstrated to achieve domain benefit
  - Convolution filtering (e.g. bilateral Gaussian filters), De Novo gene assembly, etc.
- Use GPUs to accelerate the most time-consuming aspects of the approach
  - GPU Kernels in CUDA
  - Refactor host code to better support kernels
  - Use CPU to improve the input data characteristics for GPU kernels
- Rethink the domain problem

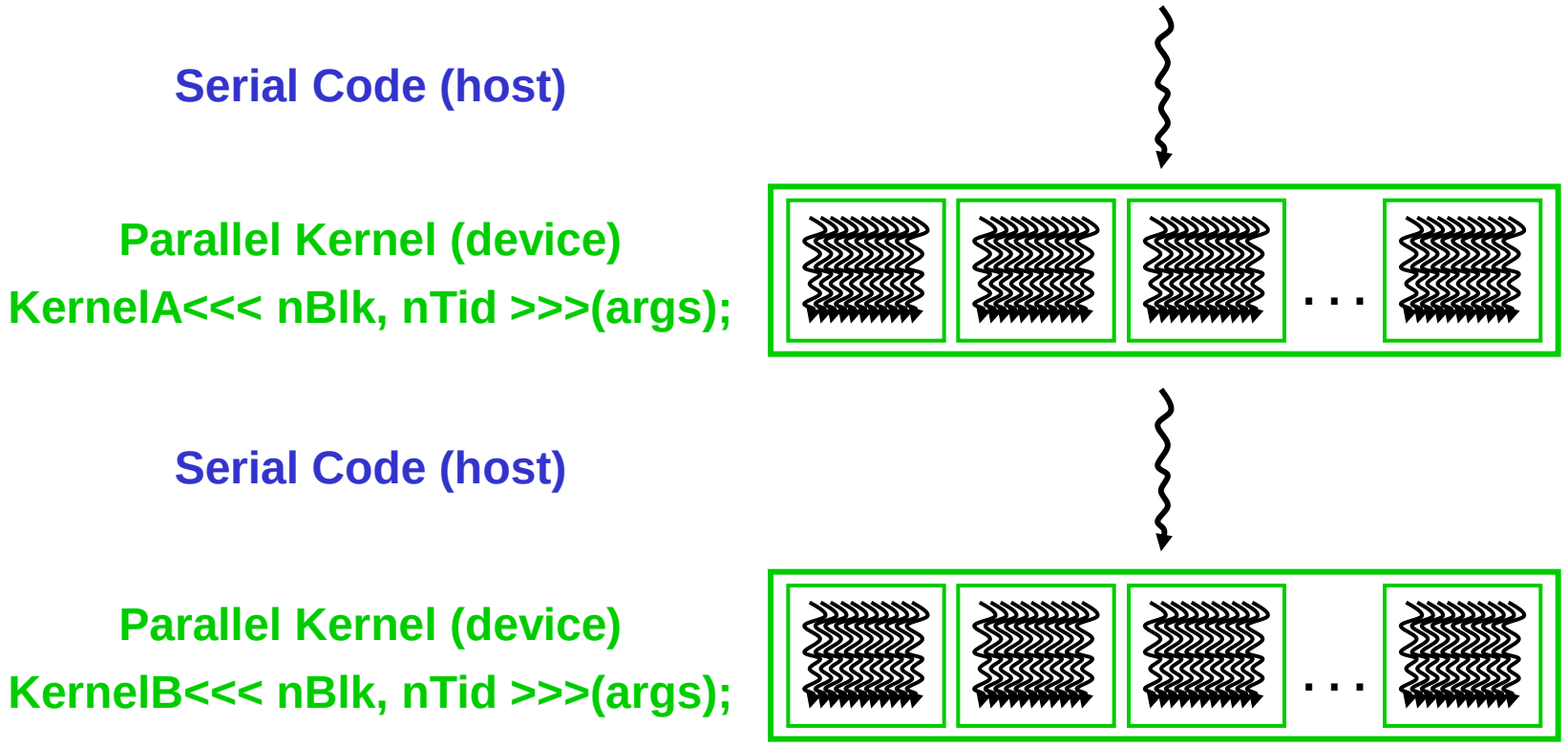
EcoG - One of the Most Energy Efficient Supercomputers in the World

- #3 of the Nov 2010 Green 500 list
- 128 nodes
- One Fermi GPU per node
- About 1 GFLOPS/Watt
- 33.6 TFLOPS DP Linpack

- Built by Illinois students and NVIDIA researchers

CUDA /OpenCL – Execution Model

- Integrated host+device app C program
  - Serial or modestly parallel parts in **host** C code
  - Highly parallel parts in **device** SPMD kernel C code

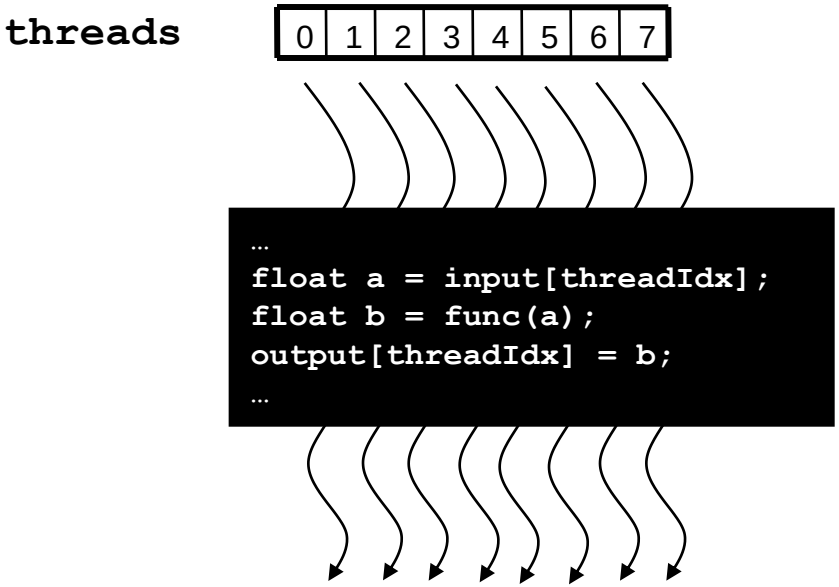


CUDA Devices and Threads

- A compute device
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads (work elements for OpenCL) 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

Arrays of Parallel Threads

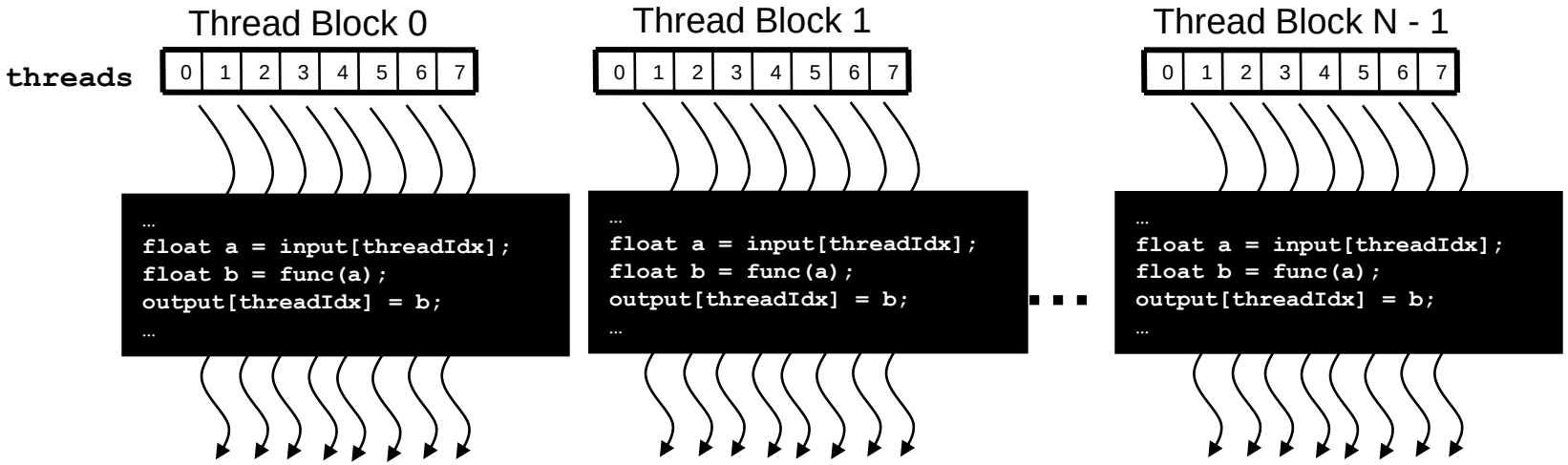
- A CUDA kernel is executed by an array of threads
  - All threads run the same code (SPMD)
  - Each thread has an index that it uses to compute memory addresses and make control decisions



12

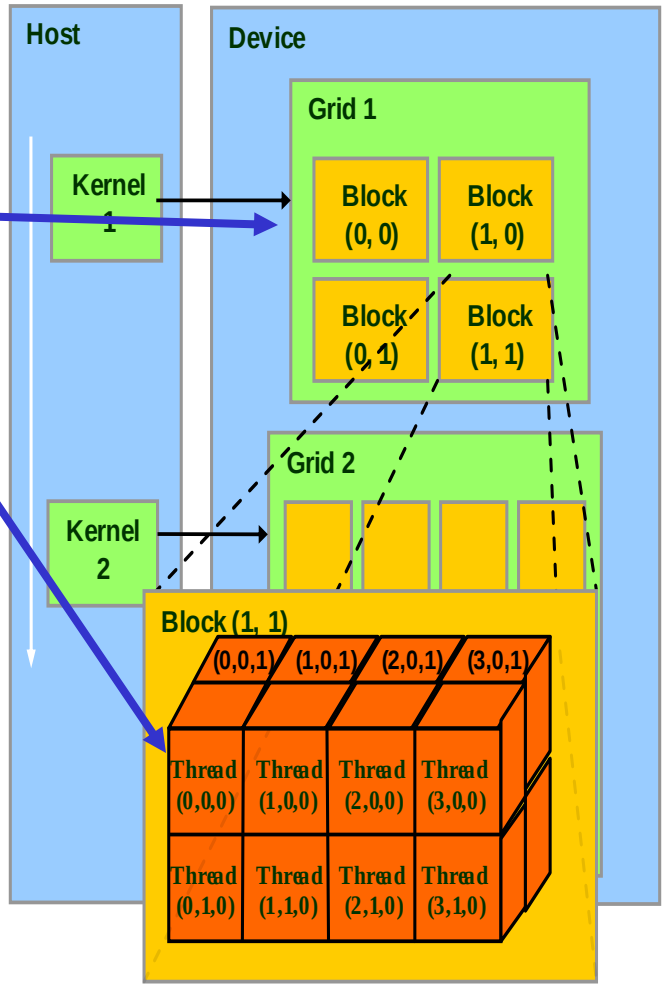
Thread Blocks: Scalable Cooperation

- 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



blockIdx and threadIdx

- Each thread uses indices to decide what data to work on
  - blockIdx: 1D or 2D, or 3D
  - threadIdx: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



Example: Vector Addition Kernel

Device Code

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAdd(float* A, float* B, float* C, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C[i] = A[i] + B[i];
}

int main()
{
    // Run ceil(N/256) blocks of 256 threads each
    vecAdd<<<ceil(N/256), 256>>>(d_A, d_B, d_C, n);
}
```

Example: Vector Addition Kernel

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAdd(float* A, float* B, float* C, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C[i] = A[i] + B[i];
}

int main()
{
    // Run ceil(N/256) blocks of 256 threads each
    vecAdd<<<ceil(N/256), 256>>>(d_A, d_B, d_C, N);
}
```

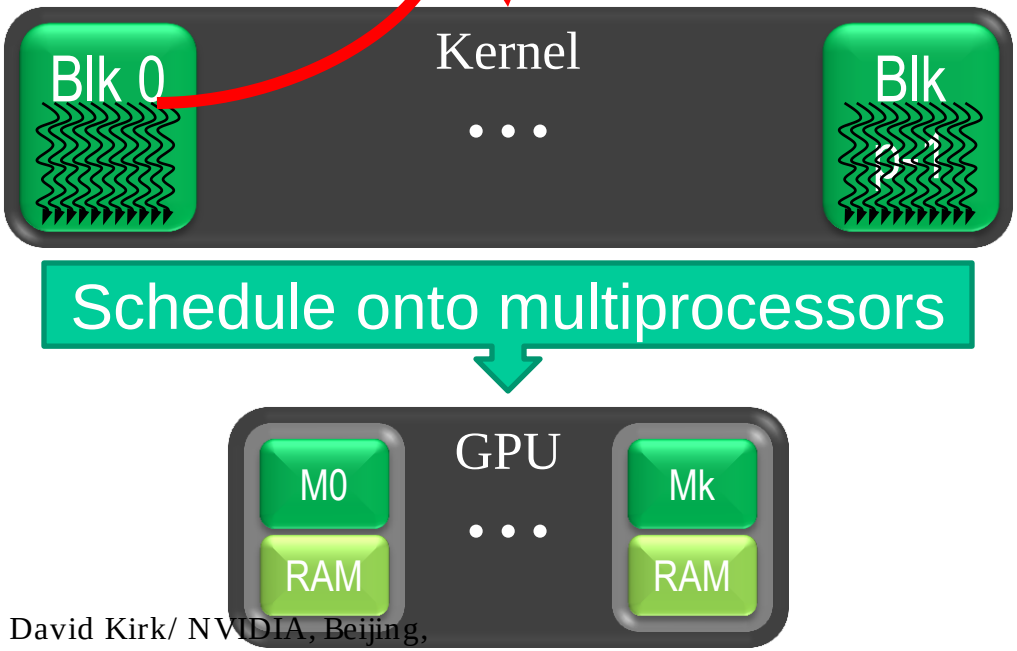
Host Code

Kernel execution in a nutshell

__host__ __global__

```
vecAdd<<<P,B>>>(n,a,x,y);
```

```
blockIdx.x   blockDim.x
    threadIdx.x
```



Harvesting Performance Benefit of Many-core GPU Requires

- Massive parallelism in application algorithms
  - Data parallelism
- Regular computation and data accesses
  - Similar work for parallel threads
- Avoidance of conflicts in critical resources
  - Off-chip DRAM (Global Memory) bandwidth
  - Conflicting parallel updates to memory locations
- Control algorithm complexity for data scalability

Massive Parallelism - Regularity







Main Hurdles to Overcome

- Serialization due to conflicting use of critical resources
- Over subscription of Global Memory bandwidth
- Load imbalance among parallel threads

Computational Thinking Skills

- The ability to translate/formulate domain problems into computational models that can be solved efficiently by available computing resources
  - Understanding the relationship between the domain problem and the computational models
  - **Understanding the strength and limitations of the computing devices**
  - **Defining problems and models to enable efficient computational solutions**

DATA ACCESS CONFLICTS

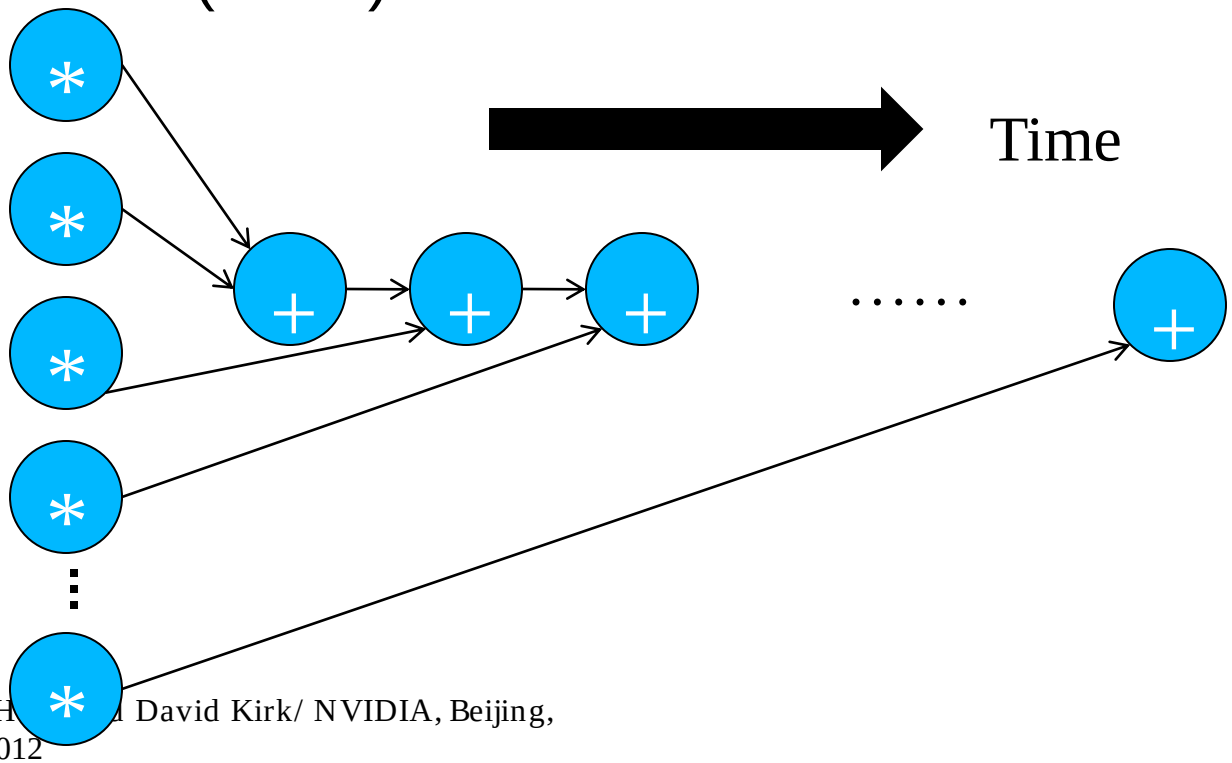
Conflicting Data Accesses Cause Serialization and Delays

- Massively parallel execution cannot afford serialization

- Contentions in accessing critical data causes serialization

A Simple Example

- A naïve inner product algorithm of two vectors of one million elements each
  - All multiplications can be done in time unit (parallel)
  - Additions to a single accumulator in one million time units (serial)



How much can conflicts hurt?

- Amdahl's Law
  - If fraction X of a computation is serialized, the speedup can not be more than 1/(1-X)
- In the previous example, X = 50%
  - Half the calculations are serialized
  - No more than 2X speedup, no matter how many computing cores are used

GLOBAL MEMORY BANDWIDTH

Global Memory Bandwidth

Ideal

Reality

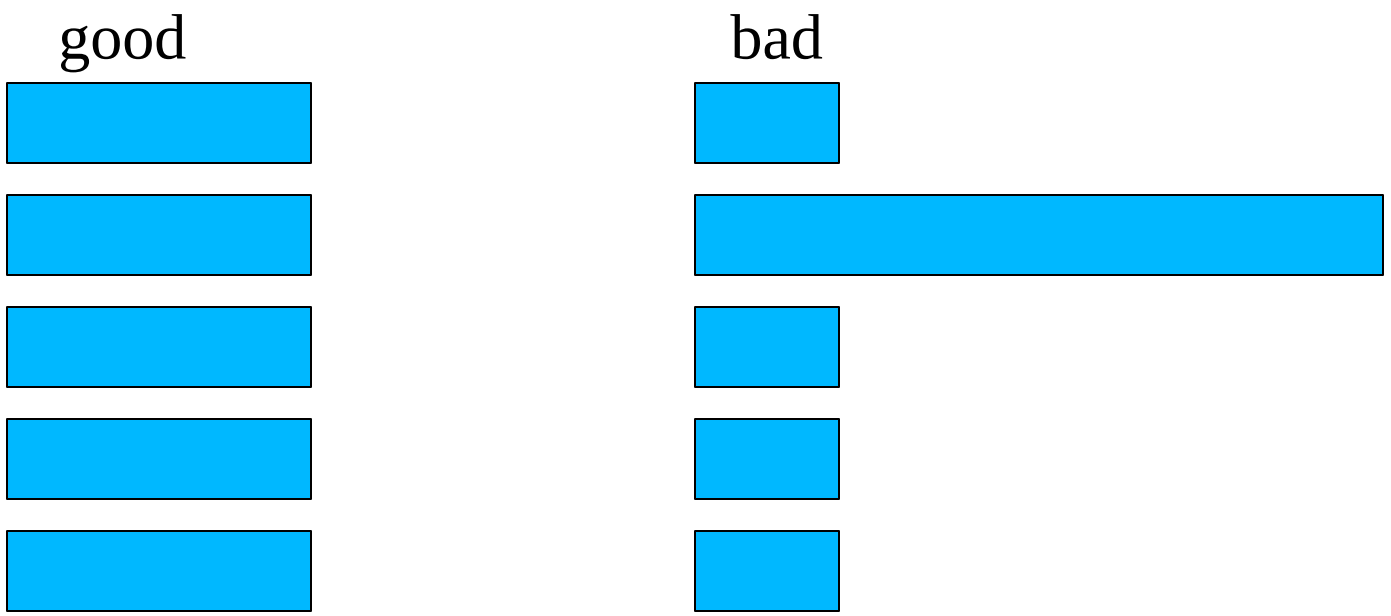
Global Memory Bandwidth

- Many-core processors have limited off-chip memory access bandwidth compared to peak compute throughput
- Fermi
  - 1 TFLOPS SPFP peak throughput
  - 0.5 TFLOPS DPFP peak throughput
  - 144 GB/s peak off-chip memory access bandwidth
    - 36 G SPFP operands per second
    - 18 G DPFP operands per second
  - To achieve peak throughput, a program must perform 1,000/36 = ~28 FP arithmetic operations for each operand value fetched from off-chip memory

LOAD BALANCE

Load Balance

- The total amount of time to complete a parallel job is limited by the thread that takes the longest to finish

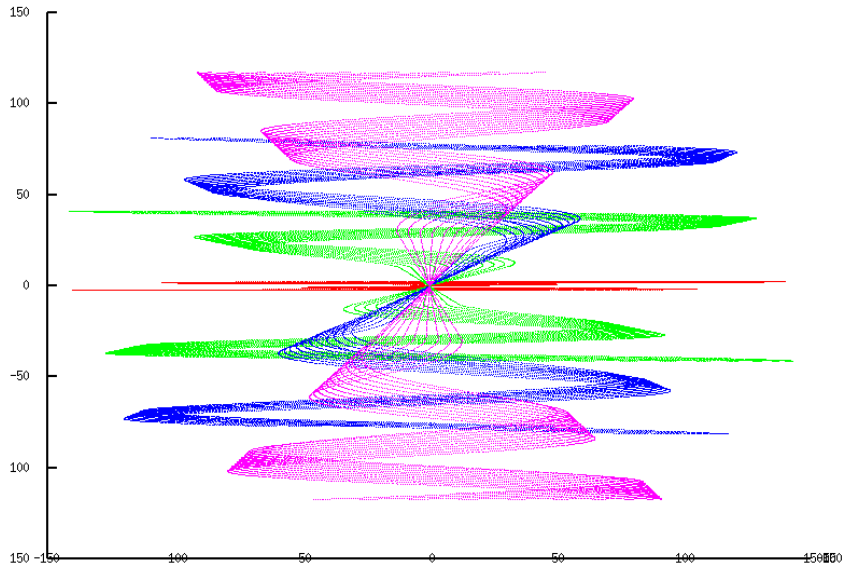


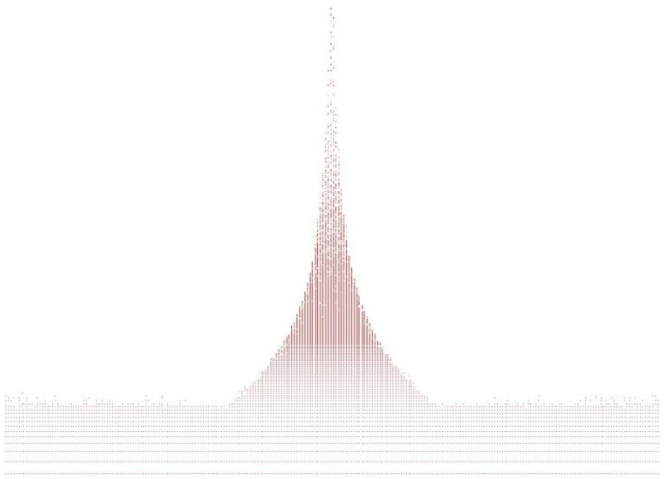
How bad can it be?

- Assume that a job takes 100 units of time for one person to finish
  - If we break up the job into 10 parts of 10 units each and have fo10 people to do it in parallel, we can get a 10X speedup
  - If we break up the job into 50, 10, 5, 5, 5, 5, 5, 5, 5, 5 units, the same 10 people will take 50 units to finish, with 9 of them idling for most of the time. We will get no more than 2X speedup.

How does imbalance come about?

- Non-uniform data distributions
  - Highly concentrated spatial data areas
  - Astronomy, medical imaging, computer vision, rendering, …
- If each thread processes the input data of a given spatial volume unit, some will do a lot more work than others

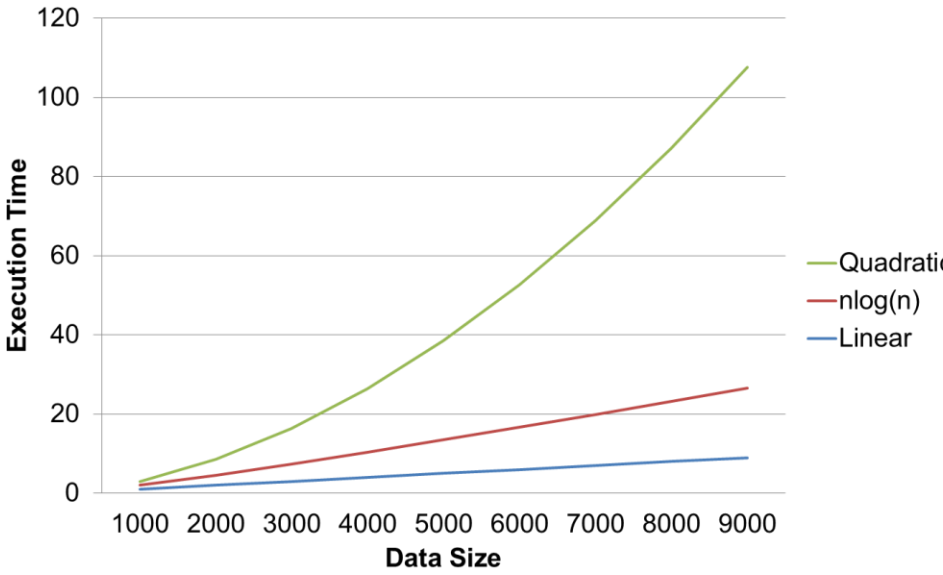




ALGORITHM COMPLEXITY

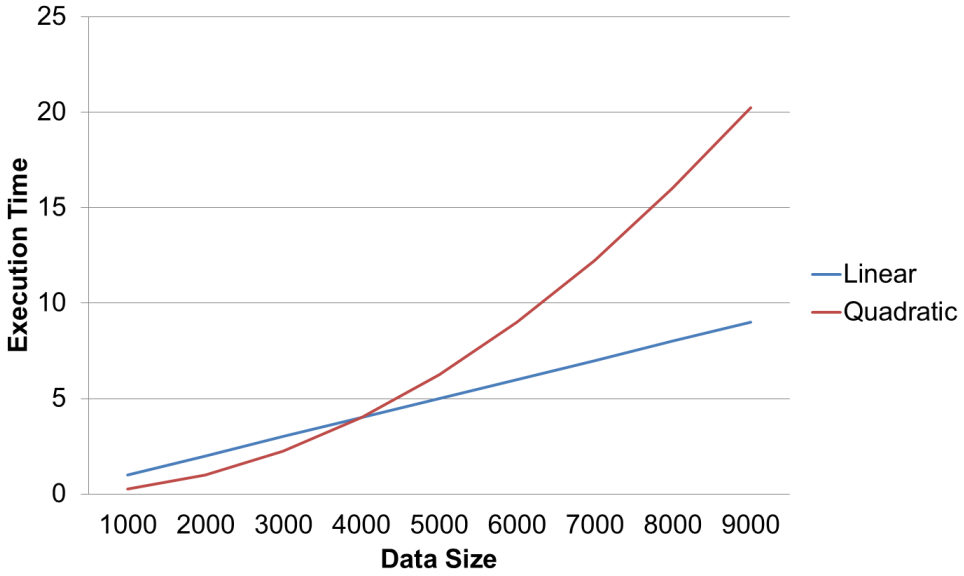
Algorithm Complexity

- Classic CS Topic
- The rate at which the number of operations performed by an algorithm grows as the data size increases



A Common Parallel Algorithm Pitfall

- A sequential algorithm is of linear complexity
- A scalable parallel algorithm is of higher complexity, say quadratic
- For small data sets, parallel wins
- As data size grows, sequential wins



But, processing large data sets is a major motivation for using GPUs!

Complexity Example: Tri-diagonal Solvers

- Classic Gaussian elimination based algorithms are of linear complexity
  - But sequential for each system being solved
- Cyclic Reduction based algorithms are of n*log(n) complexity
  - Use divide and concur to create parallelism for a system being solved

For a large system, one can uses cyclic reduction to create multiple smaller systems and then use traditional Gaussian elimination based sequential algorithm on each.

Eight Algorithmic Techniques

Technique	Contention	Bandwidth	Locality	Efficiency	Load Imbalance	CPU Leveraging
Tiling		X	X			
Privatization	X		X			
Regularization				X	X	X
Compaction		X				
Binning		X	X	X		X
Data Layout Transformation	X		X			
Thread Coarsening	X	X	X	X		
Scatter to Gather Conversion	X					

http://courses.engr.illinois.edu/ece598/hk/

You can do it.

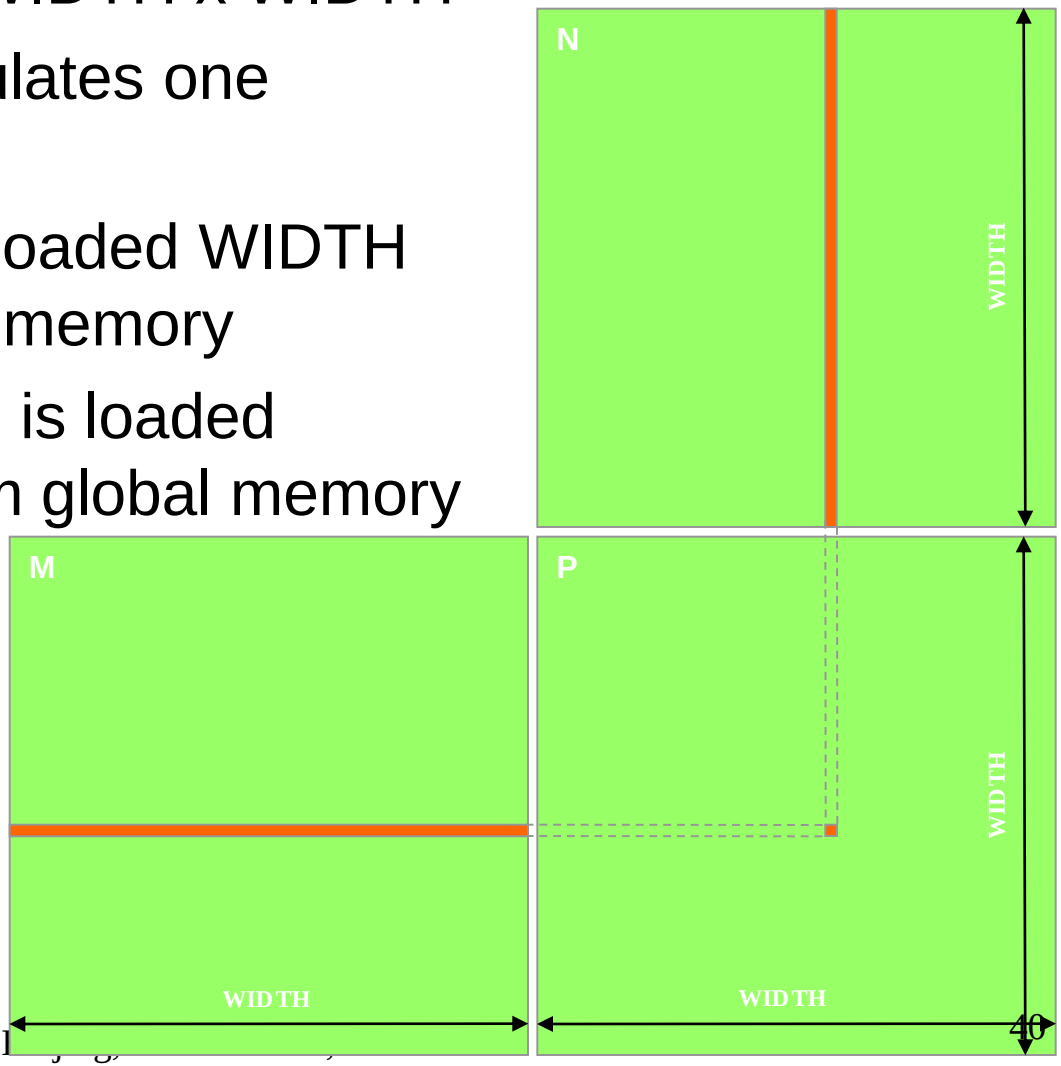
- Computational thinking is not as hard as you may think it is.
  - Most techniques have been explained, if at all, at the level of computer experts.
  - The purpose of the course is to make them accessible to domain scientists and engineers.

A Simple Running Example Matrix Multiplication

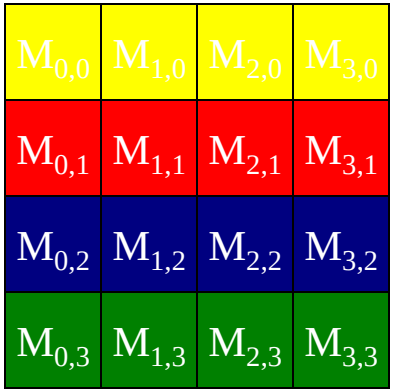
- A simple illustration of the basic features of memory and thread management in CUDA programs
  - Thread index usage
  - Memory layout
  - Register usage
  - Assume square matrix for simplicity
  - Leave shared memory usage until later

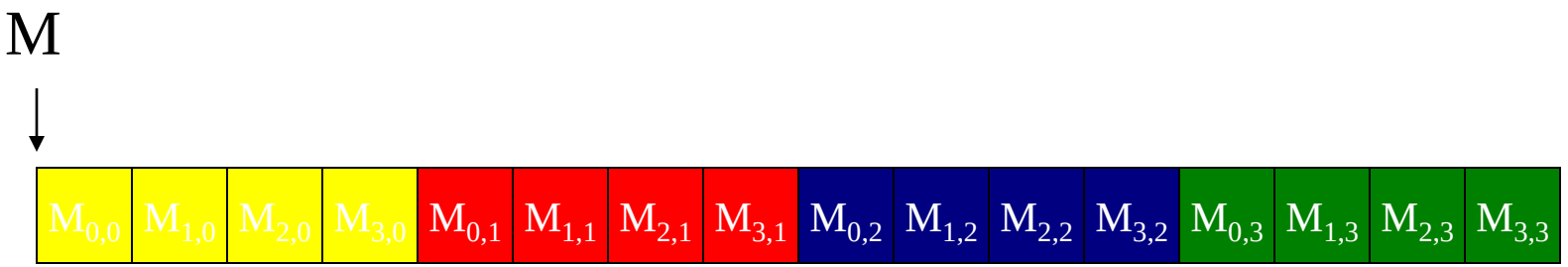
Square Matrix-Matrix Multiplication

- P = M * N of size WIDTH x WIDTH
  - Each thread calculates one element of P
  - Each row of M is loaded WIDTH times from global memory
  - Each column of N is loaded WIDTH times from global memory



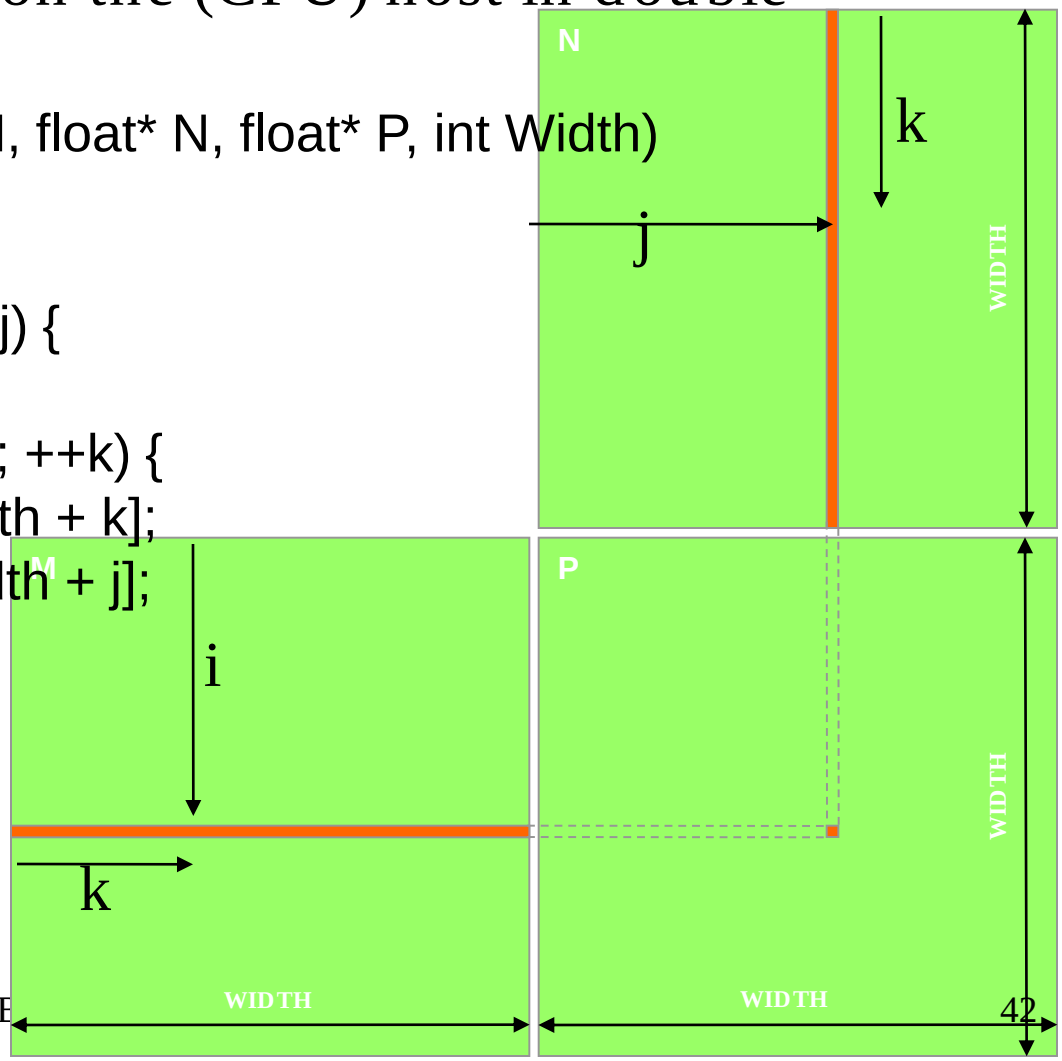
Memory Layout of a Matrix in C





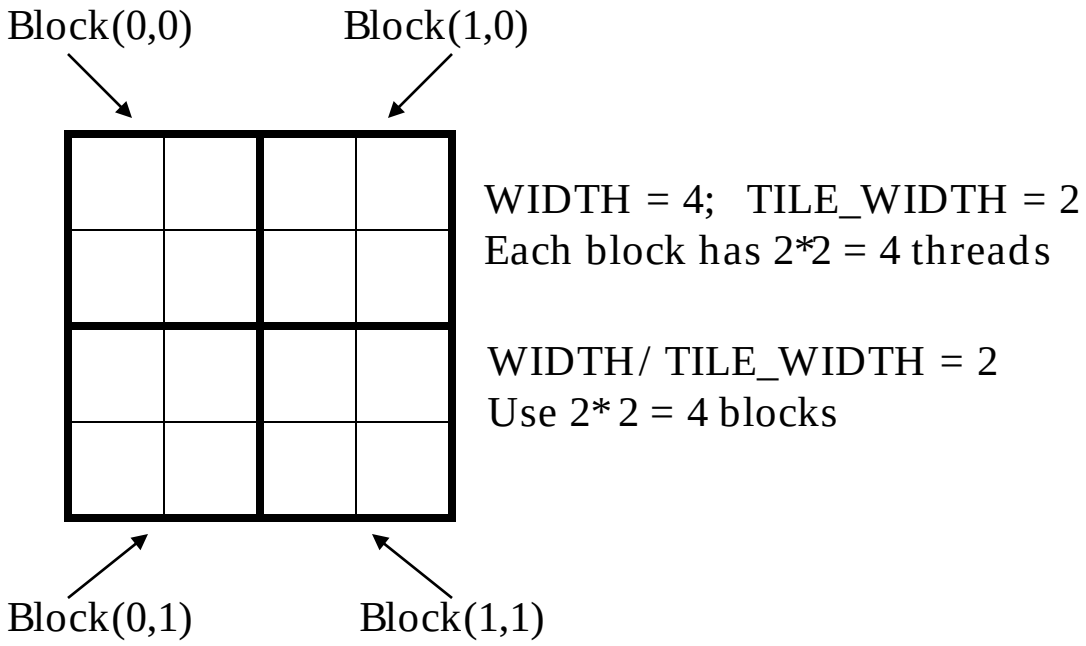
Matrix Multiplication
A Simple Host Version in C

```
// Matrix multiplication on the (CPU) host in double precision
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; ++i)
        for (int j = 0; j < Width; ++j) {
            double sum = 0;
            for (int k = 0; k < Width; ++k) {
                double a = M[i * Width + k];
                double b = N[k * Width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
```

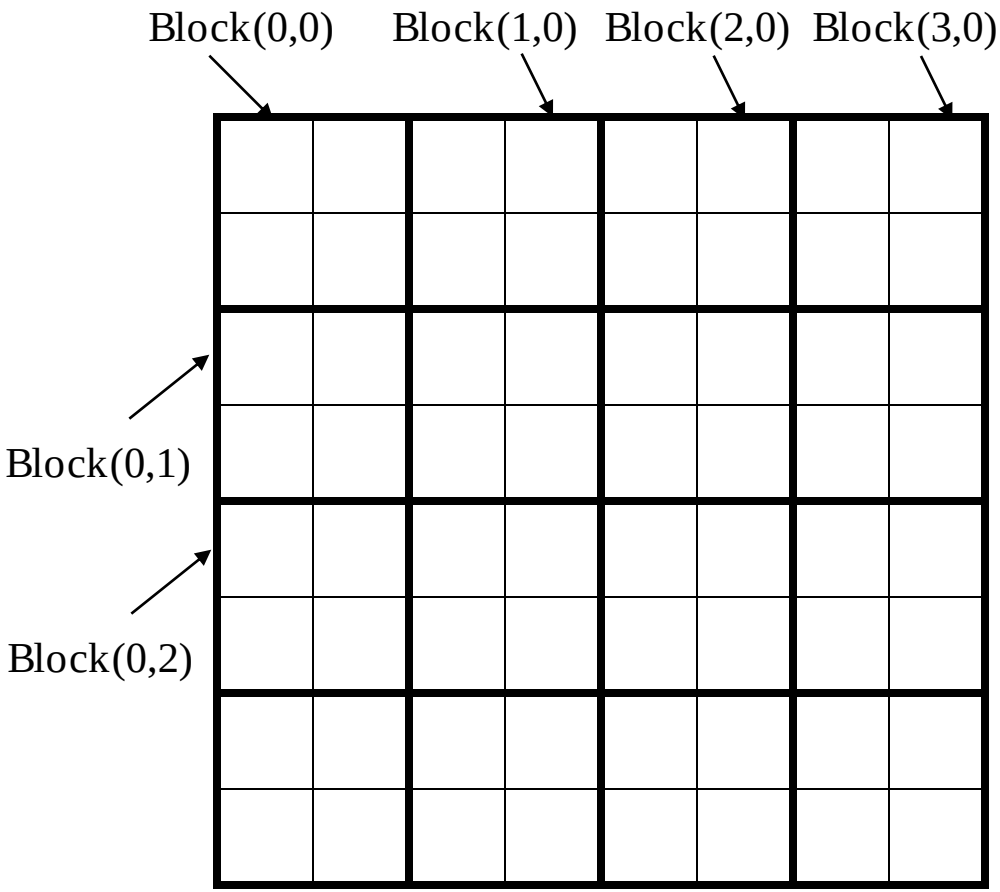


Kernel Function - A Small Example

- Have each 2D thread block to compute a $(TILE\_WIDTH)^2$ sub-matrix (tile) of the result matrix
  - Each has $(TILE\_WIDTH)^2$ threads
- Generate a 2D Grid of $(WIDTH/TILE\_WIDTH)^2$ blocks



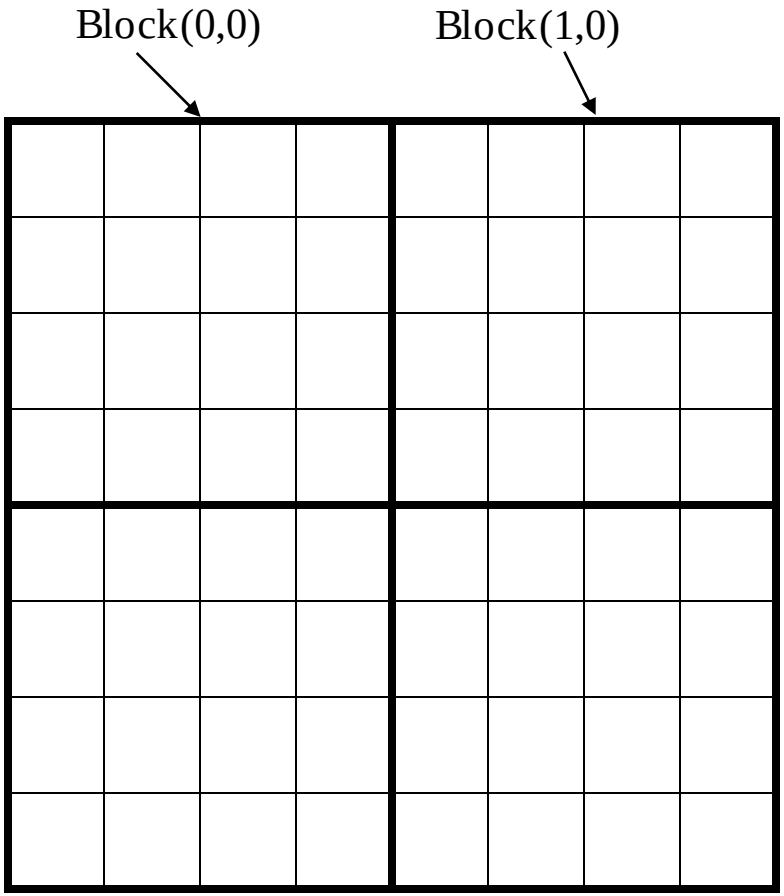
A Slightly Bigger Example



WIDTH = 8; TILE_WIDTH = 2
Each block has 2*2 = 4 threads

WIDTH/ TILE_WIDTH = 4
Use 4* 4 = 16 blocks

A Slightly Bigger Example (cont.)



WIDTH = 8; TILE_WIDTH = 4
Each block has 4*4 =16 threads

WIDTH/ TILE_WIDTH = 2
Use 2* 2 = 4 blocks

Kernel Invocation (Host-side Code)

```
// Setup the execution configuration
// TILE_WIDTH is a #define constant
    dim3 dimGrid(Width/TILE_WIDTH, Width/TILE_WIDTH, 1);
   dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);

// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

Kernel Function

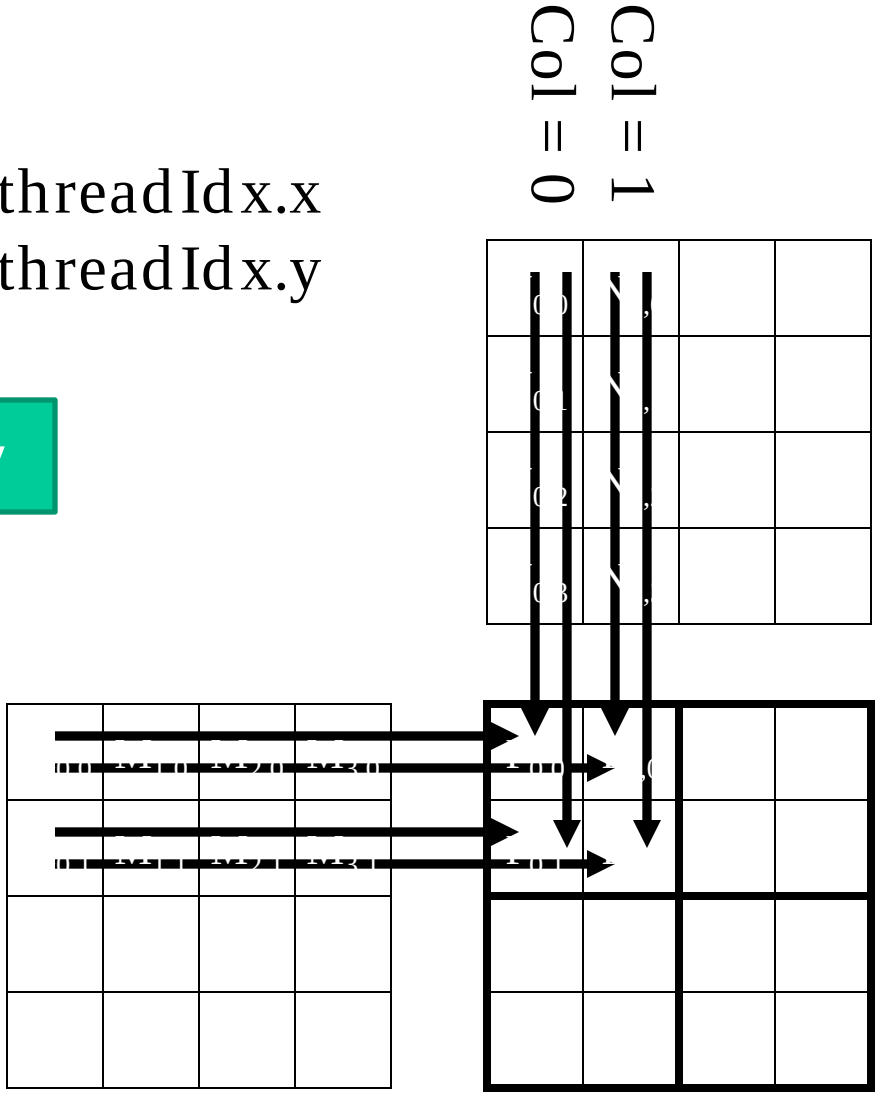
```
// Matrix multiplication kernel – per thread code

__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width)
{

    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;
```

Thread Mapping for Block (0,0) in a TILE_WIDTH = 2 Configuration

Col = 0 * (TILE_WIDTH) + threadIdx.x
Row = 0 * (TILE_WIDTH) + threadIdx.y

blockIdx.x

blockIdx.y

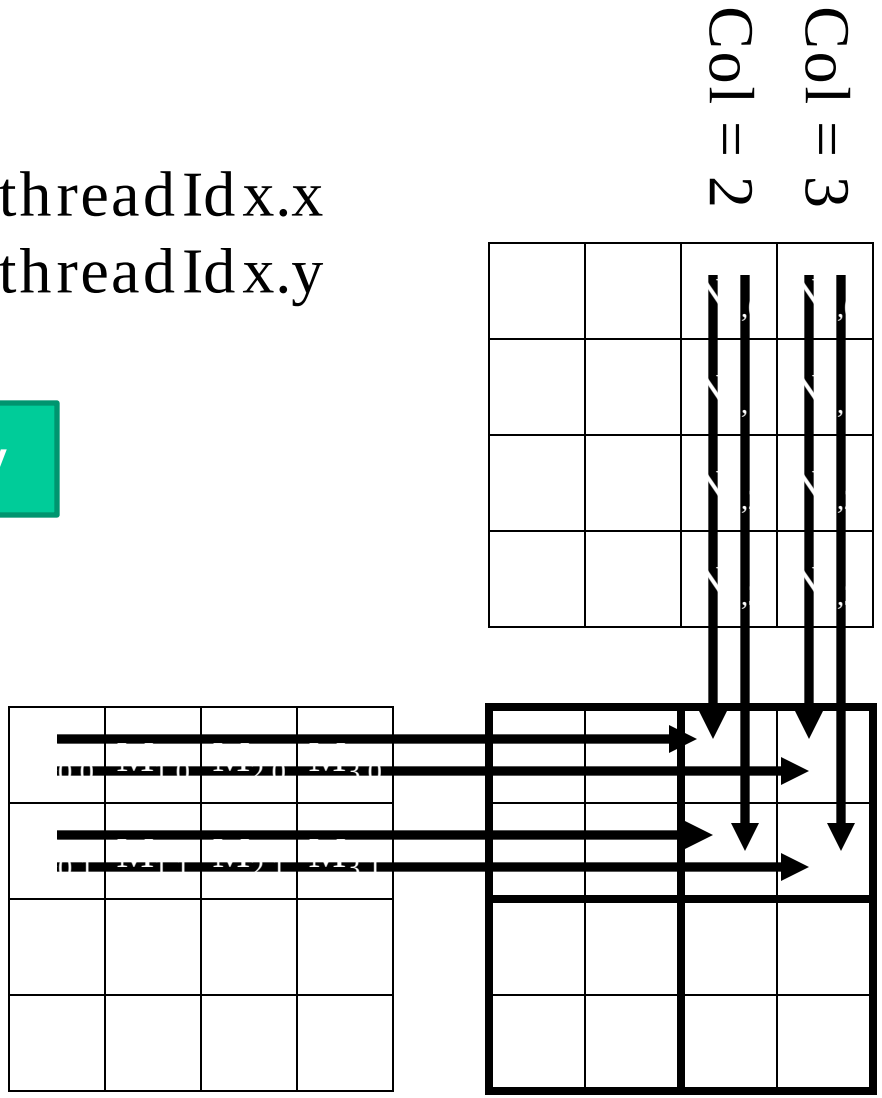


Work for Block (1,0)

Col = 1 * (TILE_WIDTH) + threadIdx.x
Row = 0 * (TILE_WIDTH) + threadIdx.y

blockIdx.x

blockIdx.y

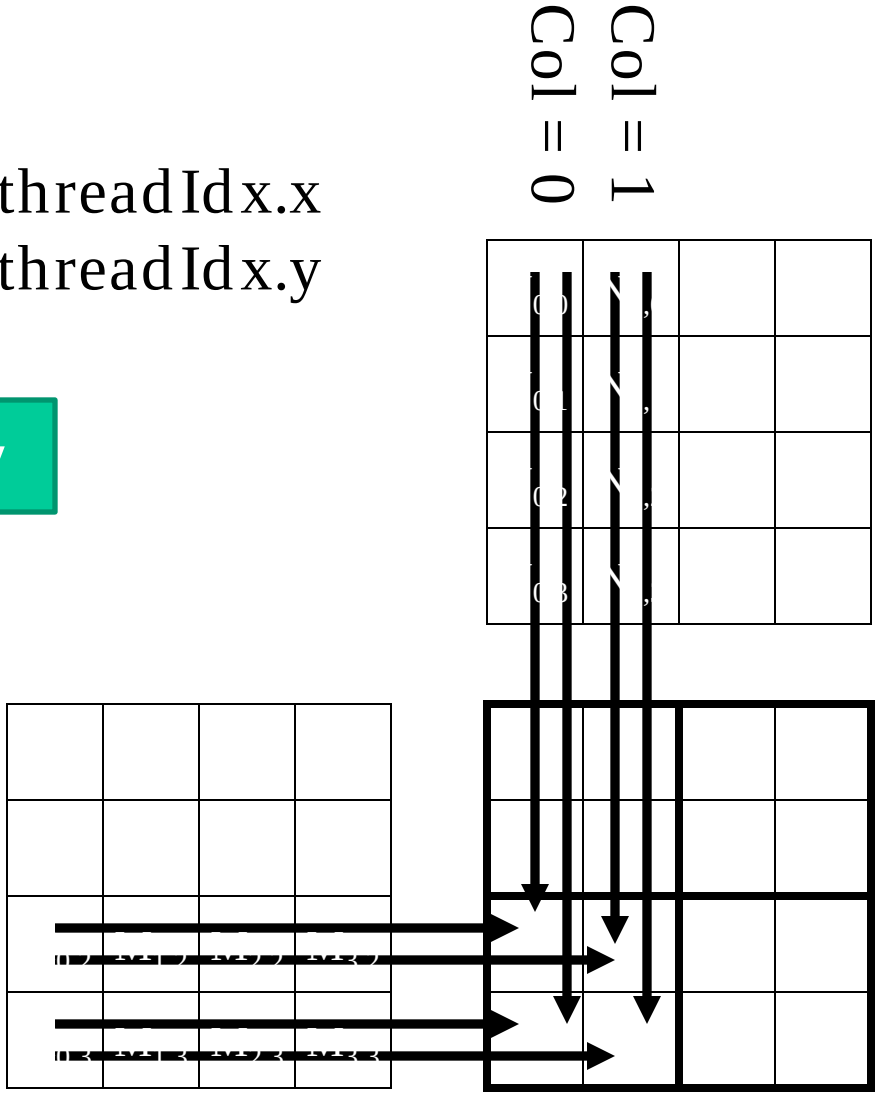


Work for Block (0,1)

Col = 0 * (TILE_WIDTH) + threadIdx.x
Row = 1 * (TILE_WIDTH) + threadIdx.y

blockIdx.x

blockIdx.y



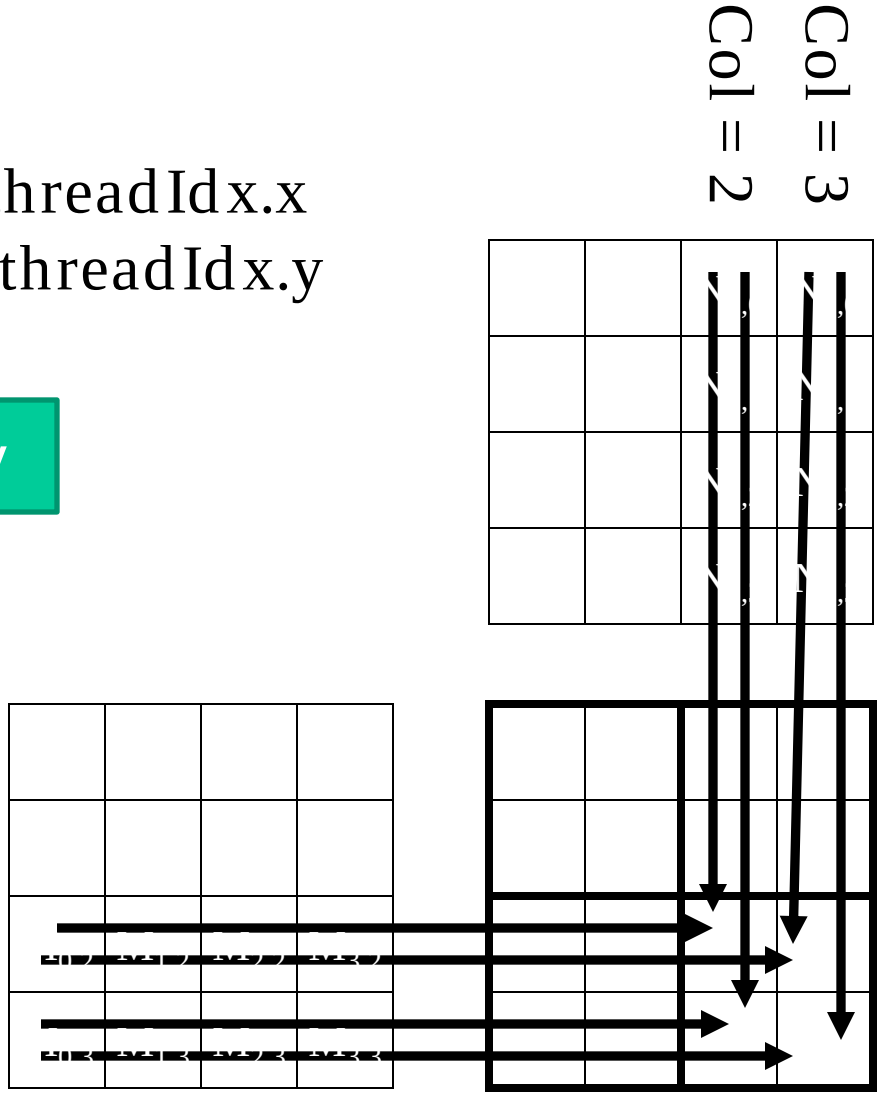
Work for Block (1,1)

Col = 1 * (TILE_WIDTH) + threadIdx.x

Row = 1 * (TILE_WIDTH) + threadIdx.y

blockIdx.x

blockIdx.y



A Simple Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width)
{
// Calculate the row index of the d_P element and d_M
 int Row = blockIdx.y*blockDim.y+threadIdx.y;
// Calculate the column idenx of d_P and d_N
 int Col = blockIdx.x*blockDim.x+threadIdx.x;

 if ((Row < Width) && (Col < Width)) {
   float Pvalue = 0;
// each thread computes one element of the block sub-matrix
   for (int k = 0; k < Width; ++k)
     Pvalue += d_M[Row*Width+k]*
               d_N[k*Width+Col];
    d_P[Row*Width+Col] = Pvalue;
  }
}
```

ANY MORE QUESTIONS?