



ECE408/CS483/CSE408 Fall 2022

Applied Parallel Programming

Lecture 4:
Memory Model

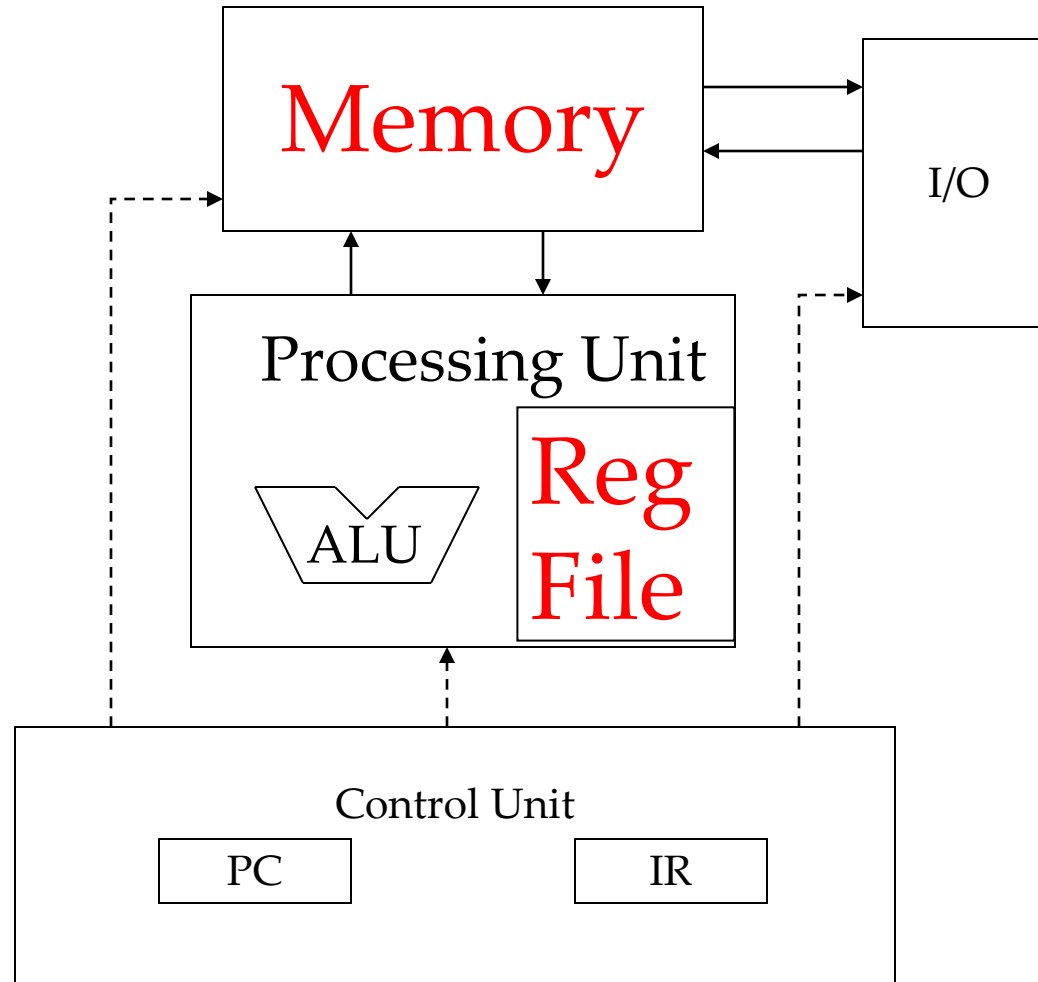
Course Reminders

- Lab 1 submission deadline is coming up
 - Instructions about how to submit will be posted any time now
 - Make sure to submit it by the deadline, no late submissions will be accepted
 - Make sure to submit the code **AND** only then answer the questions
- Lab 2 will be out soon, it is due next Friday
- The course staff only replies to questions posted on Campuswire
- Check out office hours schedule, there are many options

Objective

- To learn the basic features of the memories accessible by CUDA threads
- To prepare for MP-2 - basic matrix multiplication
- To learn to evaluate the performance implications of global memory accesses

The Von-Neumann Model



Instructions are Stored in Memory

- Every instruction needs to be fetched from memory, decoded, then executed.

- Instruction processing breaks into steps:

Fetch | Decode | Execute | Memory

- Instructions come in three flavors:
Operate, Data Transfer, and Control Flow.

Example: Processing an Add Instruction

- Example of an (LC-3) operate instruction:

ADD R1, R2, R3

- meaning:
 - read R2 and R3
 - add them as unsigned/2's complement
 - write sum to R1
- Instruction processing for an operate instruction:

Fetch | Decode | Execute | Memory

Example: Processing a Load Instruction

- Example of an (LC-3) data transfer instruction:

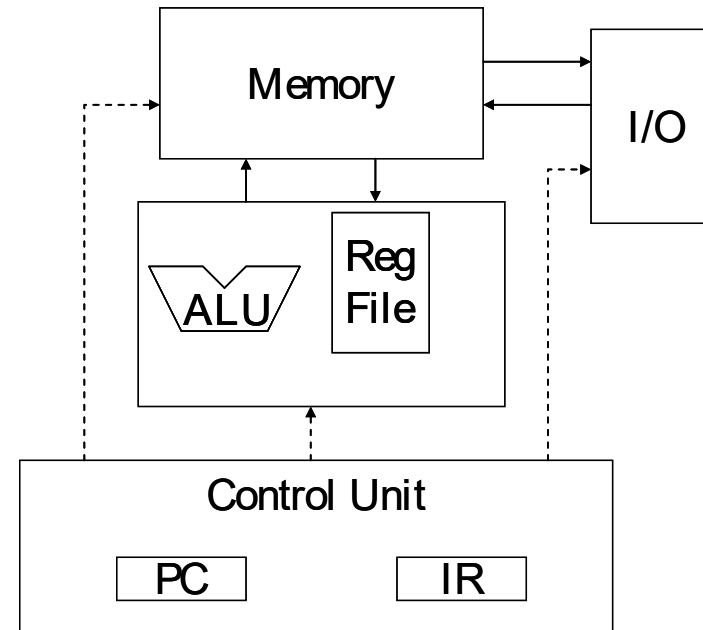
LDR R4, R6, #3 ; a load

- meaning:
 - read R6
 - add the number 3 to it
 - load the contents of memory at the resulting address
 - write the bits to R4
- Instruction processing for a load instruction:

Fetch | Decode | Execute | Memory

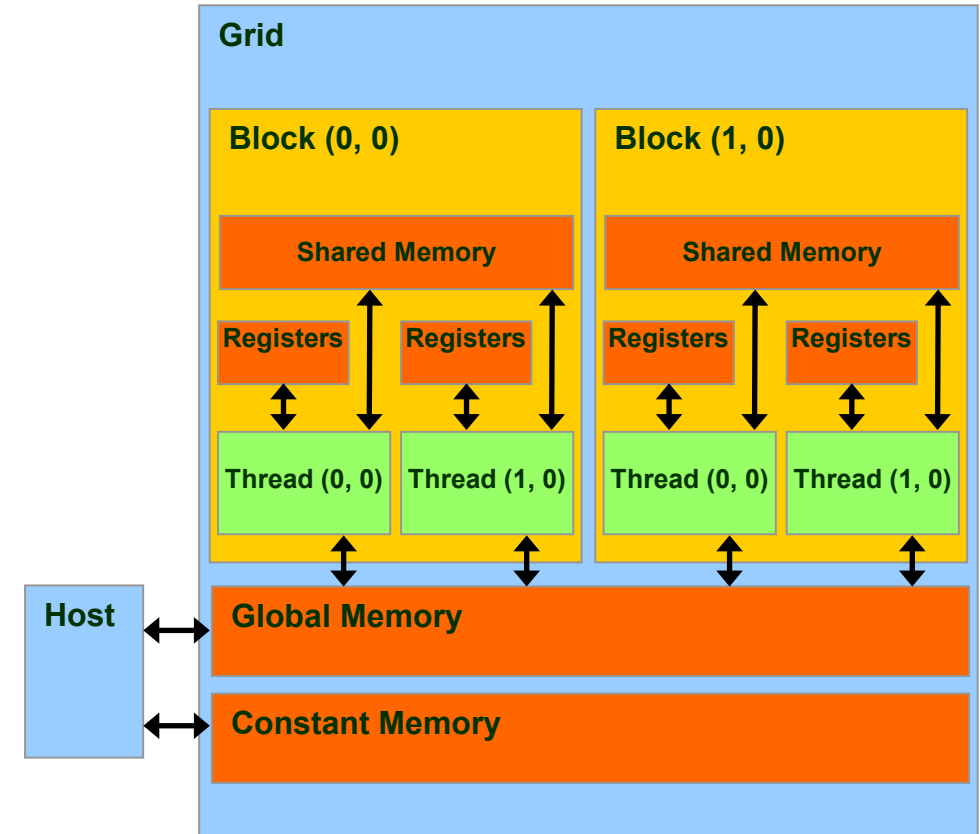
Registers vs Memory

- Registers
 - Fast: 1 cycle; no memory access required
 - Few: hundreds for CPU, $O(10k)$ for GPU SM
- Memory
 - Slow: hundreds of cycles
 - Huge: GB or more



Programmer View of CUDA Memories

- Each thread can:
 - read/write per-thread **registers** (~1 cycle)
 - read/write per-block **shared memory** (~5 cycles)
 - read/write per-grid **global memory** (~500 cycles)
 - read/only per-grid **constant memory** (~5 cycles with caching)



CUDA Variable Type Qualifiers

Variable declaration	Memory	Scope	Lifetime
<code>int LocalVar;</code>	register	thread	thread
<code>__device__ __shared__ int SharedVar;</code>	shared	block	block
<code>__device__ int GlobalVar;</code>	global	app.	application
<code>__device__ __constant__ int ConstantVar;</code>	constant	app.	application

- `__device__`
 - optional with `__shared__` or `__constant__`
 - not allowed by itself within functions
- Automatic variables with no qualifiers
 - in registers for primitive types and structures
 - in global memory for per-thread arrays

Next Application: Matrix Multiplication

- Given two Width \times Width matrices, M and N,
 - we can multiply M by N
 - to compute a third Width \times Width matrix, P:
 - $P = MN$

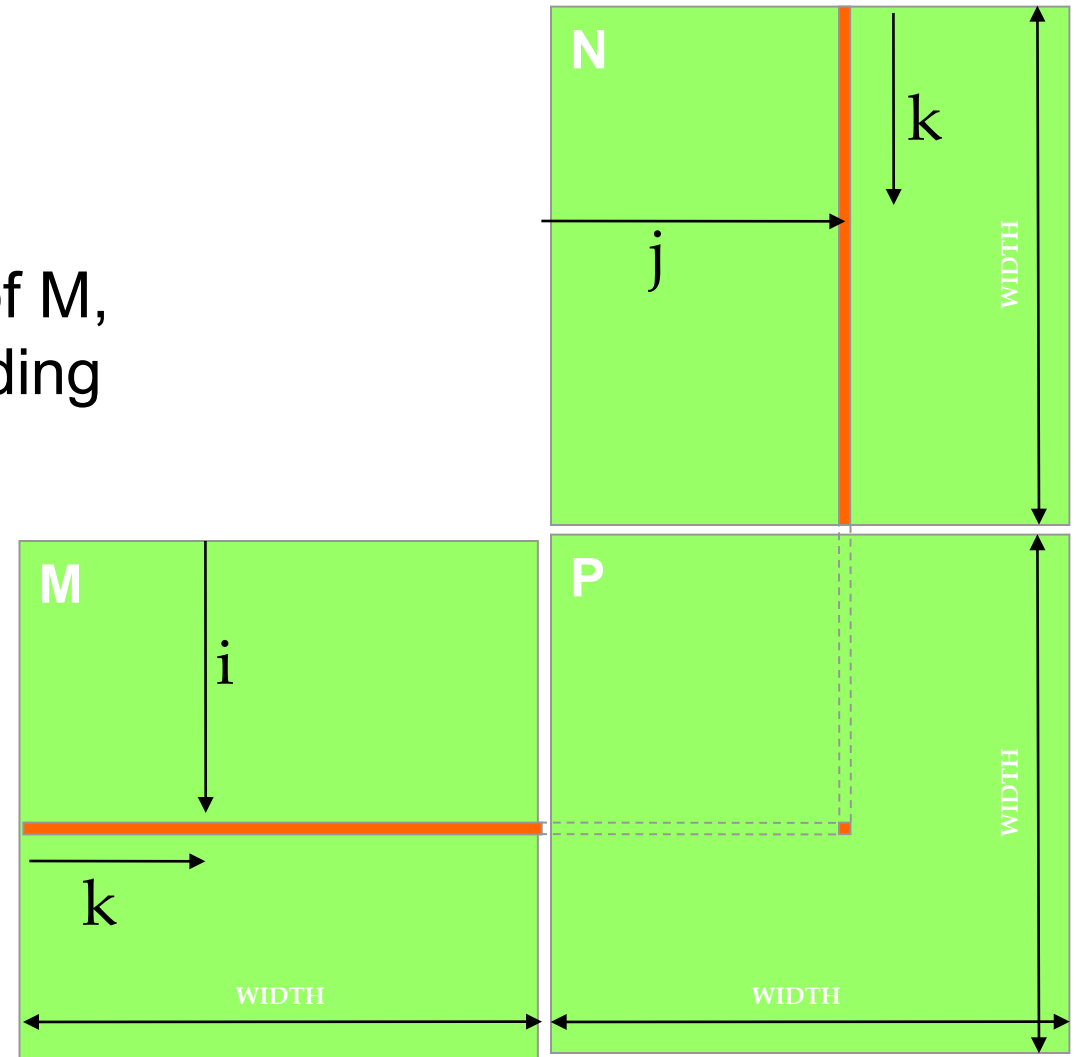
In terms of the elements of P, matrix multiplication implies computing...

$$P_{ij} = \sum_{k=1}^{Width} M_{ik} N_{kj}$$

Matrix Multiplication

$$P_{ij} = \sum_{k=1}^{Width} M_{ik} N_{kj}$$

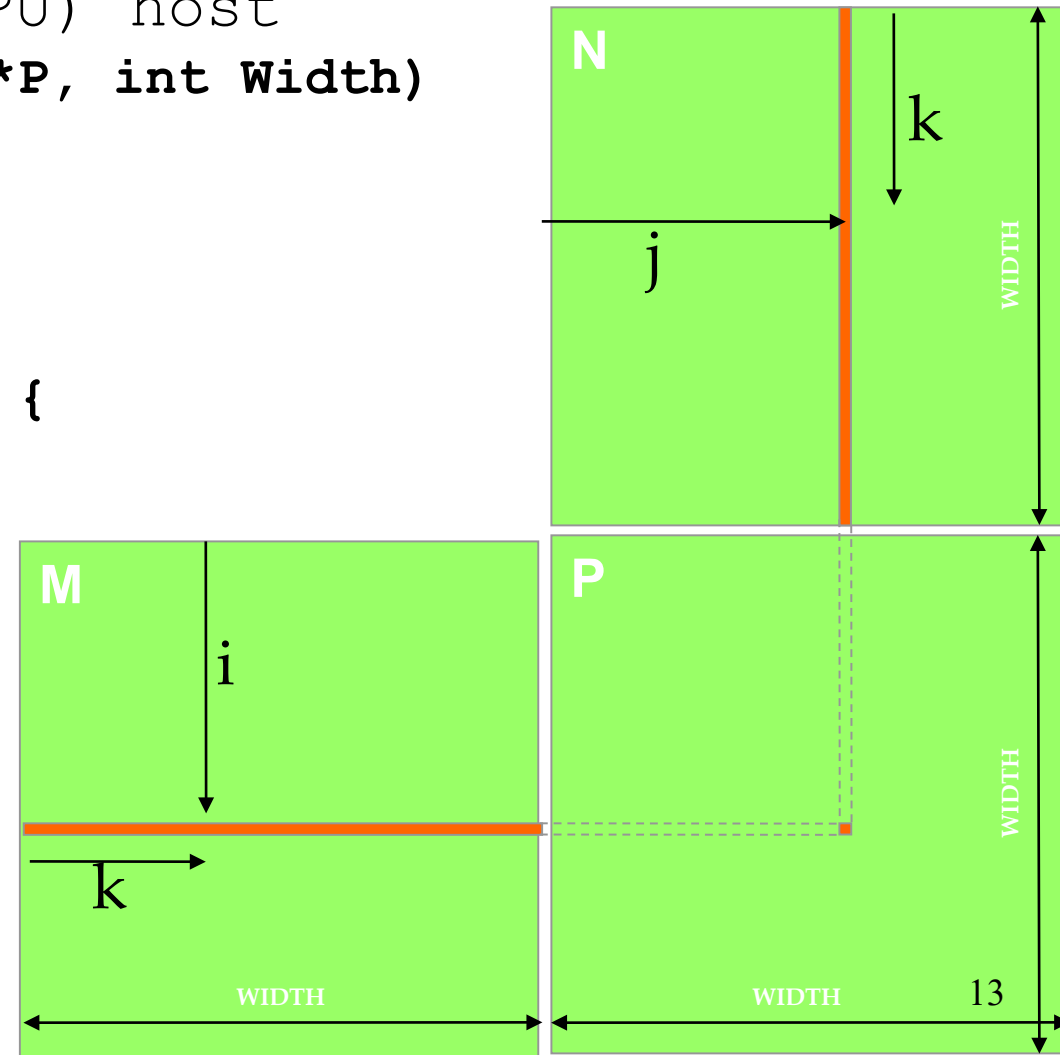
- Graphically, imagine
 - taking each element in a row of M,
 - multiplying it by the corresponding element in a column of N, and
 - summing up the products.
- Do that for every row and every column to produce P.



Matrix Multiplication Example

A Simple Host Version in C

```
// Matrix multiplication on the (CPU) host
void MatrixMul(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;
        }
}
```



Parallelize Elements of P

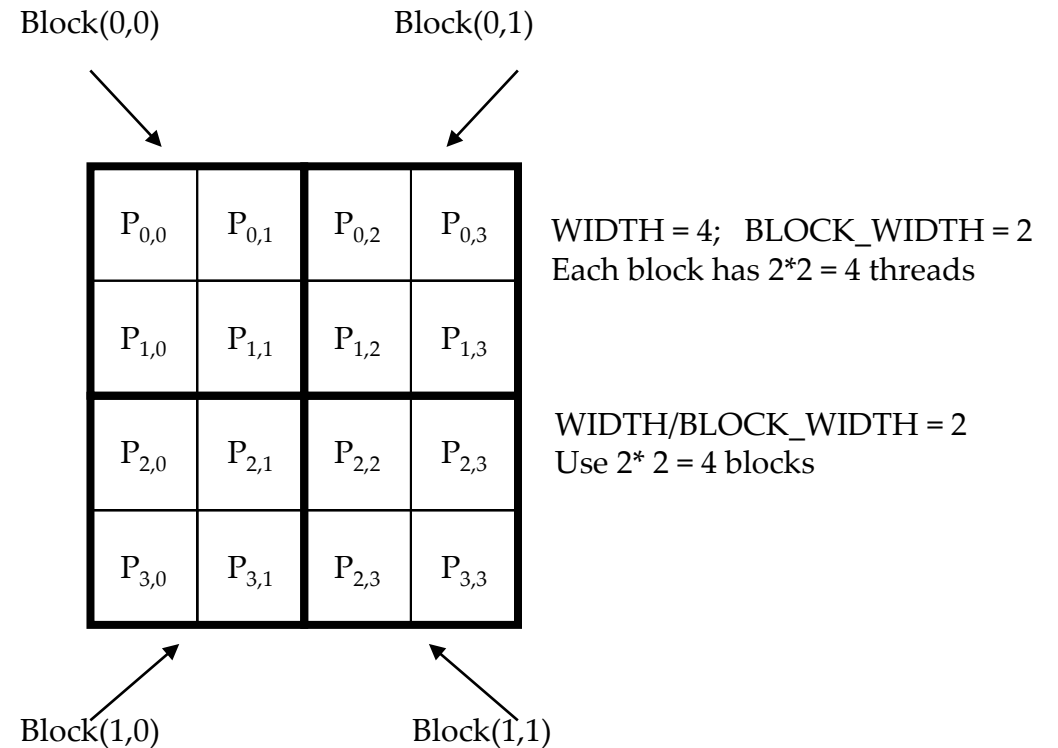
- What can we parallelize?
 - start with the **two outer loops**
 - parallelize **computation of elements of P**
- What about the inner loop?
 - Technically, floating-point is NOT associative.
 - The **parallel sum** is called a **reduction**—we'll come back to it in a few weeks.
 - For now, **use a single thread for each P_{ij}** .

Compute Using 2D Blocks in a 2D Grid

- **P** is 2D, so organize threads in 2D as well:
 - Split the output **P** into square **tiles**
 - of size **$\text{TILE_WIDTH} \times \text{TILE_WIDTH}$**
 - (a preprocessor constant).
 - **Each thread block produces one tile** of TILE_WIDTH^2 elements.
 - Create **$\lceil \text{Width} / \text{TILE_WIDTH} \rceil^2$** thread **blocks** to cover the output matrix.

Kernel Function - A Small Example

- Have each 2D thread block to compute a $(\text{BLOCK_WIDTH})^2$ sub-matrix of the result matrix
 - Each block has $(\text{BLOCK_WIDTH})^2$ threads
- Generate a 2D Grid of $(\text{WIDTH}/\text{BLOCK_WIDTH})^2$ blocks
- This concept is called **tiling**. Each block represents a **tile**.



Example: Width 8, TILE_WIDTH 2

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$	$P_{0,4}$	$P_{0,5}$	$P_{0,6}$	$P_{0,7}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$	$P_{1,4}$	$P_{1,5}$	$P_{1,6}$	$P_{1,7}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$	$P_{2,4}$	$P_{2,5}$	$P_{2,6}$	$P_{2,7}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$	$P_{3,4}$	$P_{3,5}$	$P_{3,6}$	$P_{3,7}$
$P_{4,0}$	$P_{4,1}$	$P_{4,2}$	$P_{4,3}$	$P_{4,4}$	$P_{4,5}$	$P_{4,6}$	$P_{4,7}$
$P_{5,0}$	$P_{5,1}$	$P_{5,2}$	$P_{5,3}$	$P_{5,4}$	$P_{5,5}$	$P_{5,6}$	$P_{5,7}$
$P_{6,0}$	$P_{6,1}$	$P_{6,2}$	$P_{6,3}$	$P_{6,4}$	$P_{6,5}$	$P_{6,6}$	$P_{6,7}$
$P_{7,0}$	$P_{7,1}$	$P_{7,2}$	$P_{7,3}$	$P_{7,4}$	$P_{7,5}$	$P_{7,6}$	$P_{7,7}$

Each block has
 $2 \times 2 = 4$ threads.

$\text{WIDTH}/\text{TILE_WIDTH} = 4$
Use $4 \times 4 = 16$ blocks.

Example: Same Matrix, Larger Tiles

(Width 8, TILE_WIDTH 4)

P _{0,0}	P _{0,1}	P _{0,2}	P _{0,3}	P _{0,4}	P _{0,5}	P _{0,6}	P _{0,7}
P _{1,0}	P _{1,1}	P _{1,2}	P _{1,3}	P _{1,4}	P _{1,5}	P _{1,6}	P _{1,7}
P _{2,0}	P _{2,1}	P _{2,2}	P _{2,3}	P _{2,4}	P _{2,5}	P _{2,6}	P _{2,7}
P _{3,0}	P _{3,1}	P _{3,2}	P _{3,3}	P _{3,4}	P _{3,5}	P _{3,6}	P _{3,7}
P _{4,0}	P _{4,1}	P _{4,2}	P _{4,3}	P _{4,4}	P _{4,5}	P _{4,6}	P _{4,7}
P _{5,0}	P _{5,1}	P _{5,2}	P _{5,3}	P _{5,4}	P _{5,5}	P _{5,6}	P _{5,7}
P _{6,0}	P _{6,1}	P _{6,2}	P _{6,3}	P _{6,4}	P _{6,5}	P _{6,6}	P _{6,7}
P _{7,0}	P _{7,1}	P _{7,2}	P _{7,3}	P _{7,4}	P _{7,5}	P _{7,6}	P _{7,7}

Each block has
 $4 \times 4 = 16$ threads.

$WIDTH/TILE_WIDTH = 2$
Use $2 \times 2 = 4$ blocks.

Kernel Invocation (Host-side Code)

```
// TILE_WIDTH is a #define constant
dim3 dimGrid(ceil((1.0*Width)/TILE_WIDTH),
             ceil((1.0*Width)/TILE_WIDTH), 1);

dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);

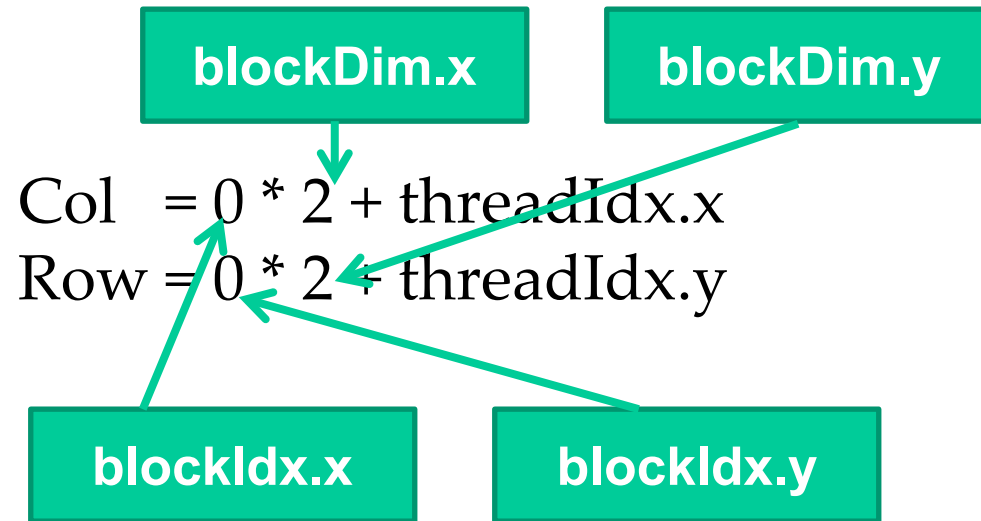
// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

Kernel Function

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

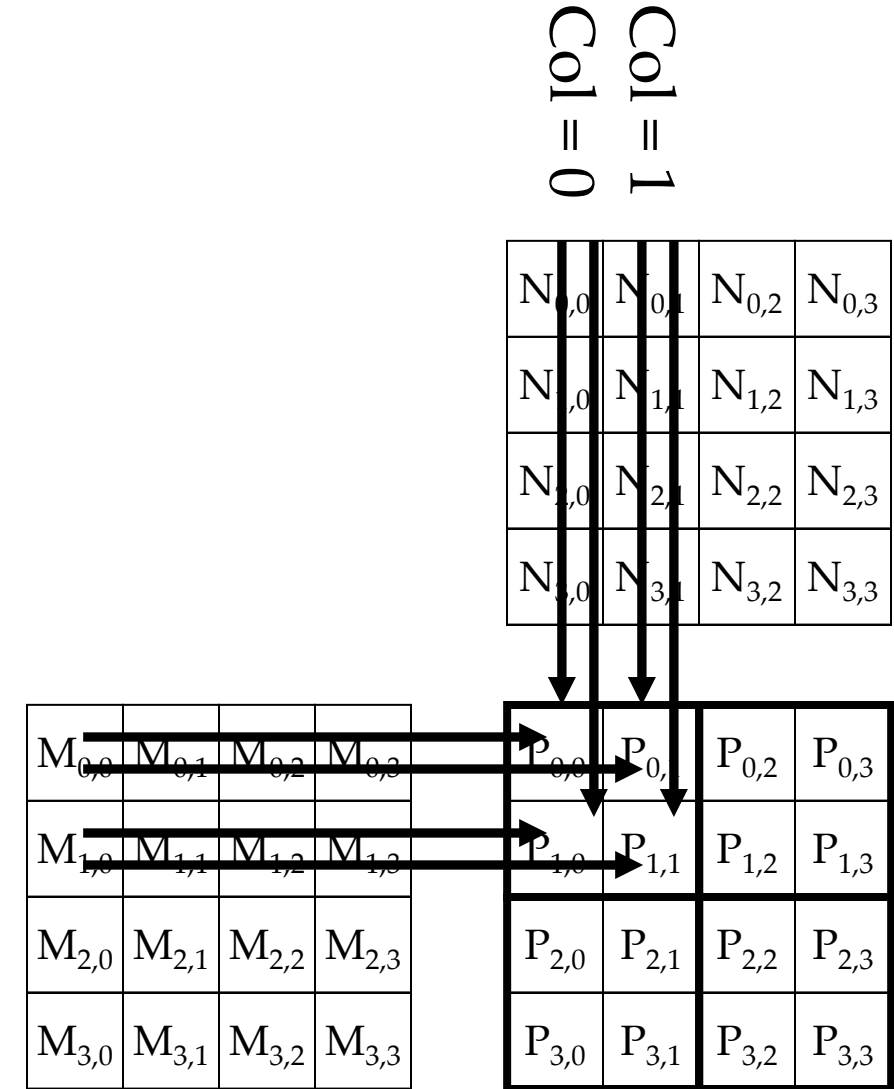
```
__global__  
void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width)  
{  
  
    // Pvalue is used to store the element of the matrix  
    // that is computed by the thread  
    float Pvalue = 0;  
    ...  
    ...  
    d_P[  ] = Pvalue;  
}
```

Work for Block (0,0) with TILE_WIDTH 2



Row = 0

Row = 1



Work for Block (0,1)

$$\text{Col} = 1 * 2 + \text{threadIdx.x}$$

$$\text{Row} = 0 * 2 + \text{threadIdx.y}$$

blockIdx.x

blockIdx.y

Row = 0

Row = 1

Col = 2

Col = 3

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	$N_{0,3}$
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$
$N_{3,0}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$	$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$	$P_{0,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$	$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$	$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$

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;
    }
}
```

Memory Bandwidth is Overloaded!

- That's a **simple implementation**:
 - GPU kernel is the **CPU code** with the **outer loops replaced with** per-thread **index calculations**!
- Unfortunately, performance is quite bad.
- Why?
- With the given approach,
 - global **memory bandwidth can't** supply enough data to **keep the SMs busy**!

Where Do We Access Global Memory?

```
__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 index 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;
    }
}
```

accesses
to global
memory

Each Thread Requires 4B of Data per FLOP

- Each threads access global memory
 - for elements of **M** and **N**:
 - 4B each**, or **8B per pair**.
 - (And once TOTAL to **P** per thread—ignore it.)
- With each pair of elements,
 - a thread does a single multiply-add,
 - 2 FLOP**—floating-point operations.
- So for every FLOP,
 - a thread needs** 4B from memory:
 - 4B / FLOP**.

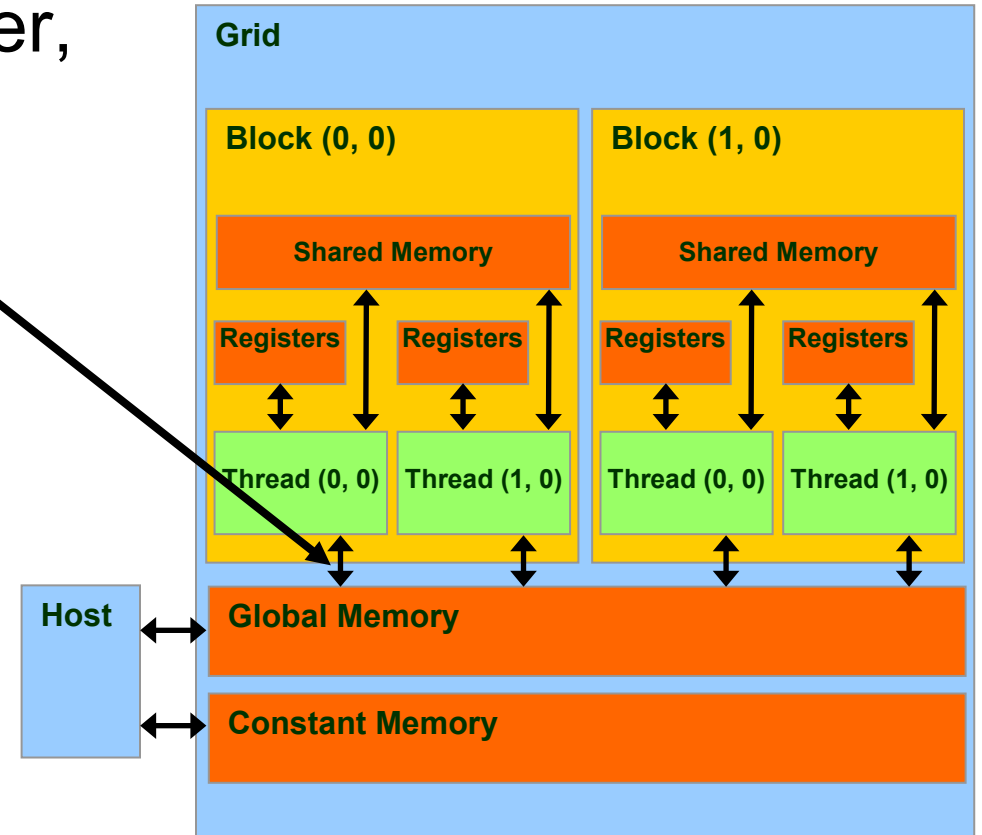
150 GB/s Bandwidth Implies 37.5 GFLOPs

- One generation of GPUs:
 - **1,000 GFLOP/s** of compute power,
 - and
 - **150 GB/s** of memory bandwidth.

- Dividing bandwidth by memory requirements:

$$\frac{150 \text{ GB/s}}{4 \text{ B/FLOP}} = 37.5 \text{ GFLOP/s}$$

which **limits computation!**



What to Do? Reuse Memory Accesses!

But **37.5 GFLOPs** is a limit.

In an **actual execution**,

- memory is not busy all the time, and
- the code **runs at** about **25 GFLOPs**.

To get closer to 1,000 GFLOPs

- we **need to** drastically **cut down**
- **accesses to global memory**.

Two vertical lines, one blue and one orange, are positioned on the left side of the slide.

**ANY MORE QUESTIONS?
READ CHAPTER 4!**