



Computation to Core Mapping

— Lessons learned from a simple application

A Simple Application

- **Matrix Multiplication**
 - Used as an example throughout the course
- **Goal for today:**
 - Show the concept of **“Computation-to-Core Mapping”**
 - *Block schedule, Occupancy, and thread schedule*
 - **Assumption**
 - Deal with square matrix for simplicity
 - Leave memory issues later
 - With global memory and local registers

The algorithm and CPU code

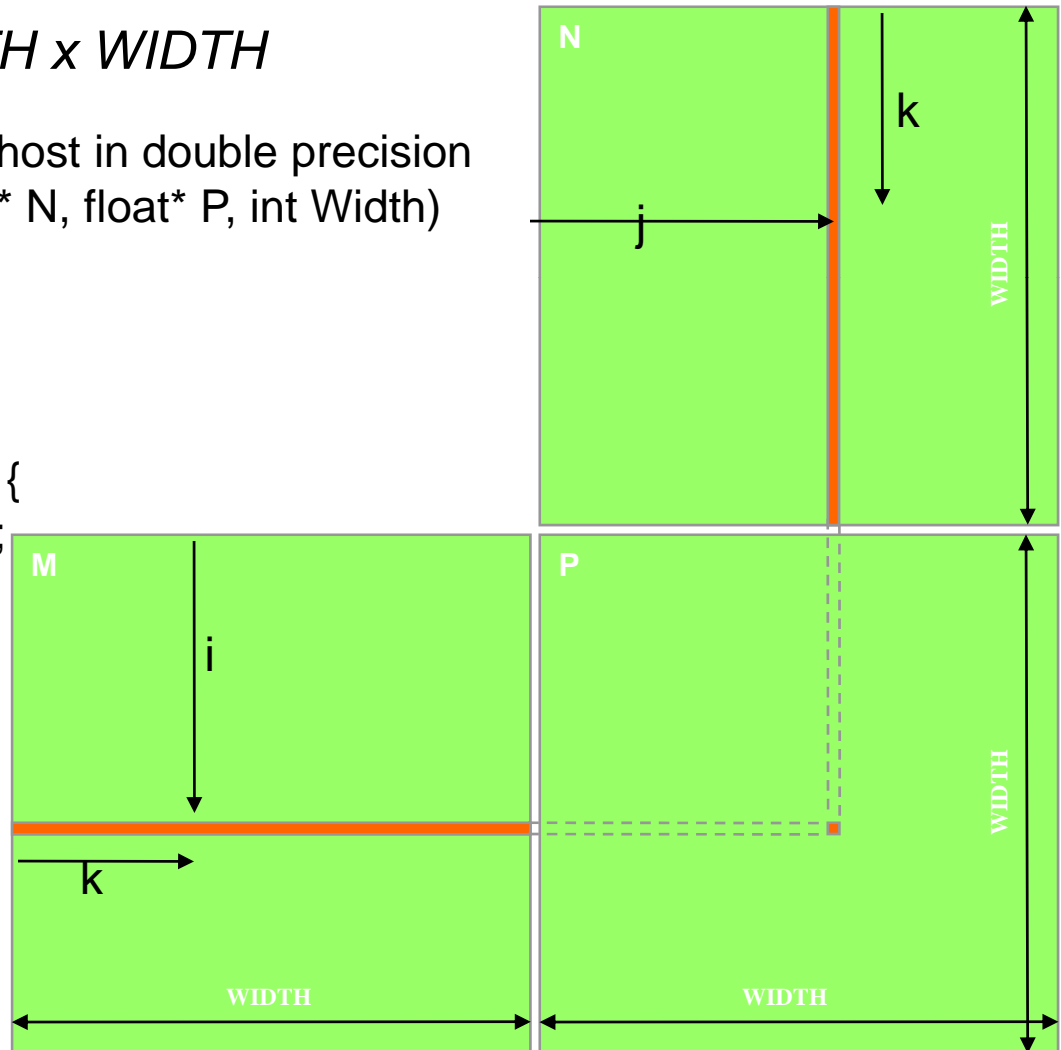
$P = M * N$ of size $WIDTH \times WIDTH$

// 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;
    }
}

```



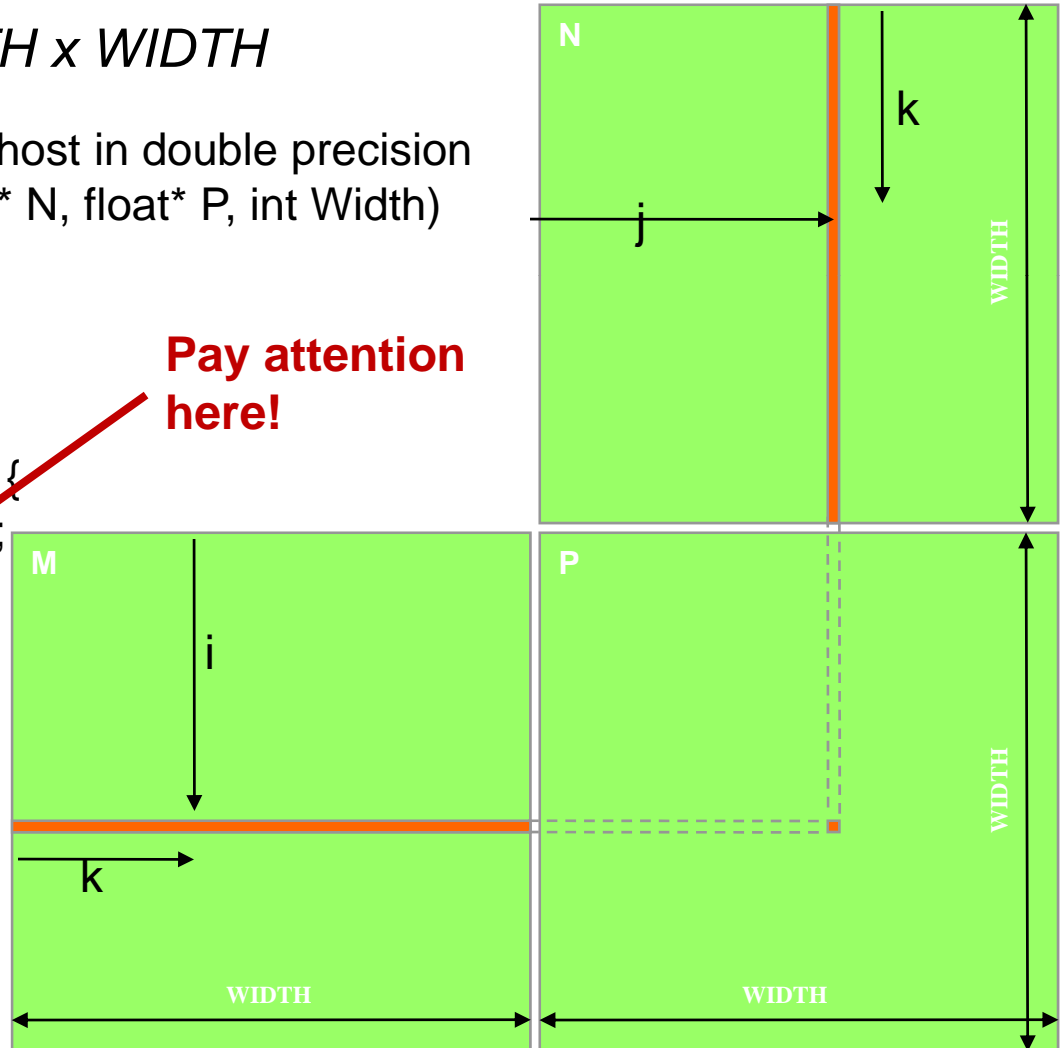
The algorithm and CPU code

$P = M * N$ of size $WIDTH \times WIDTH$

// Matrix multiplication on the (CPU) host in double precision
void MatrixMulOnHost(float* M, float* N, float* P, int Width)

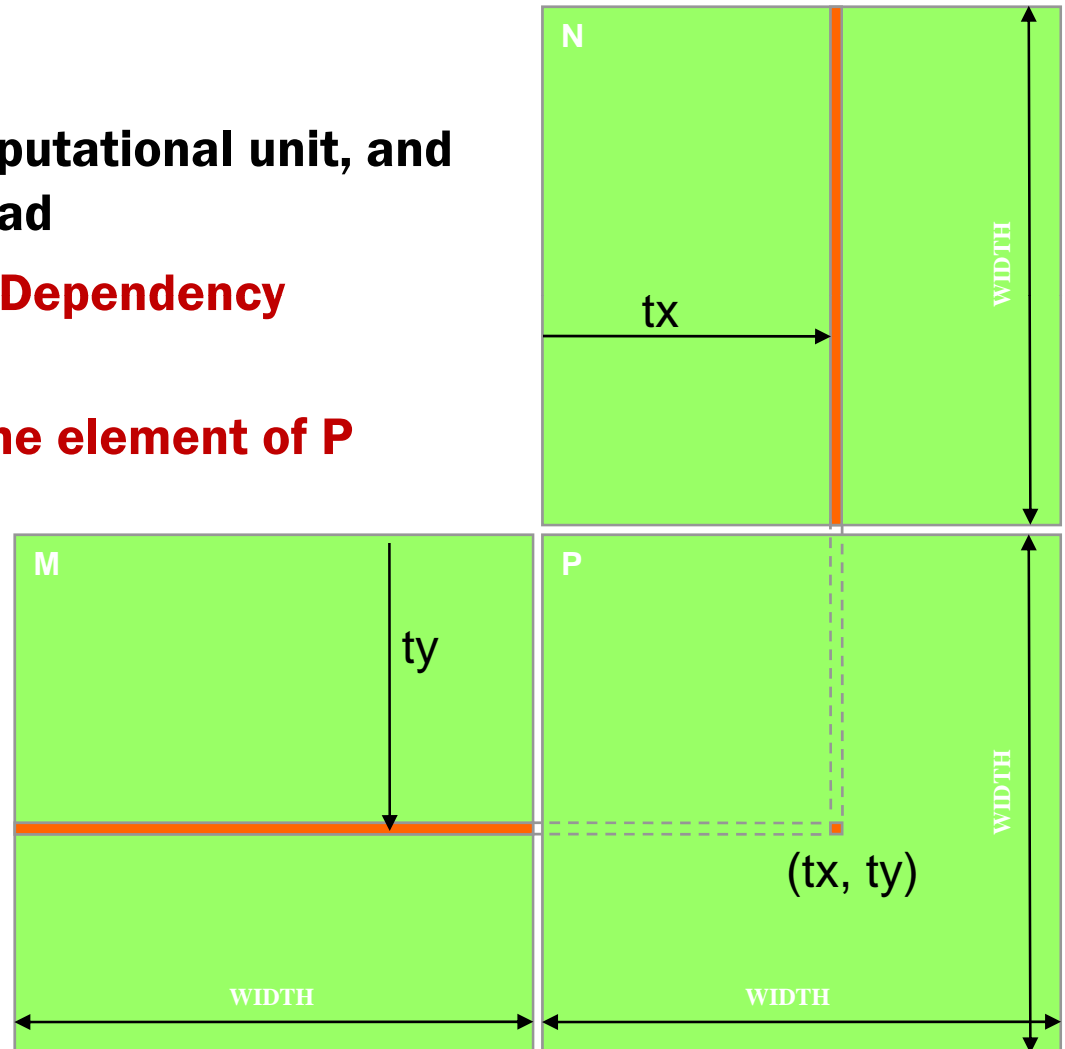
```
{
  for (int i = 0; i < Width; ++i)
    for (int j = 0; j < Width; ++j) {
      double sum = 0;
      for (int k = 0; k < Width; ++k) {
        double a = M[i * width + k];
        double b = N[k * width + j];
        sum += a * b;
      }
      P[i * Width + j] = sum;
    }
}
```

Pay attention here!



First Mapping Scheme

- **Thread mapping:**
 - Define the finest computational unit, and map it onto each thread
 - Main criterion : **None Dependency**
 - In our first scheme:
Unit: Calculation of one element of P
- **Block mapping:**
 - Simple: One block



Step 1: Memory layout

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

M (column#, row#)

M
↓

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

Step 2: Input Matrix Data Transfer (Host Code)

```
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);
```

Step 3: Output Matrix Data Transfer (Host Code)

```
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);
}
```


Step 4: Kernel Function

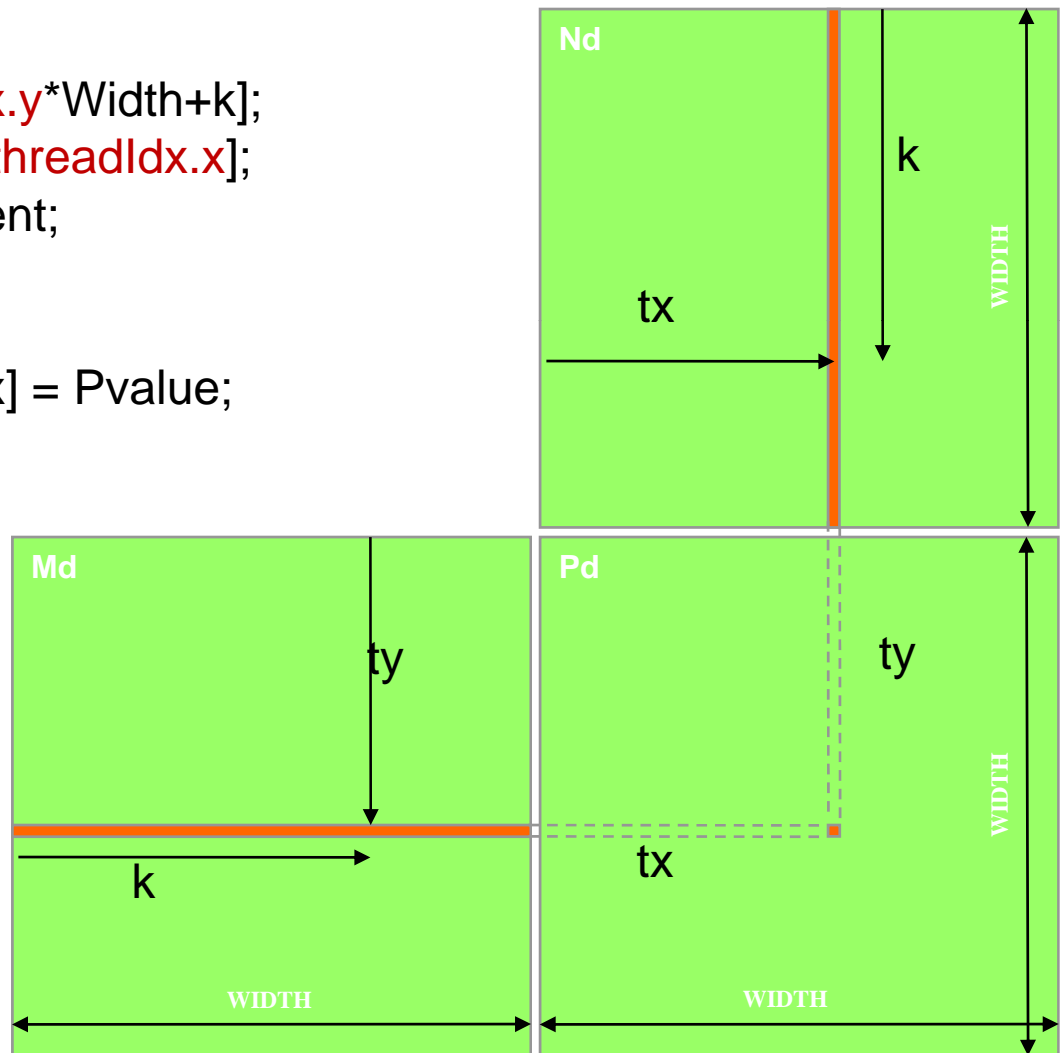
```
// Matrix multiplication kernel – per thread code
```

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;
```

Step 4: Kernel Function (cont.)

```
for (int k = 0; k < Width; ++k) {
    float Melement = Md[threadIdx.y*Width+k];
    float Nelement = Nd[k*Width+threadIdx.x];
    Pvalue += Melement * Nelement;
}
```

```
Pd[threadIdx.y*Width+threadIdx.x] = Pvalue;
}
```



Step 5: Kernel Invocation (Host Code)

```
// Setup the execution configuration
```

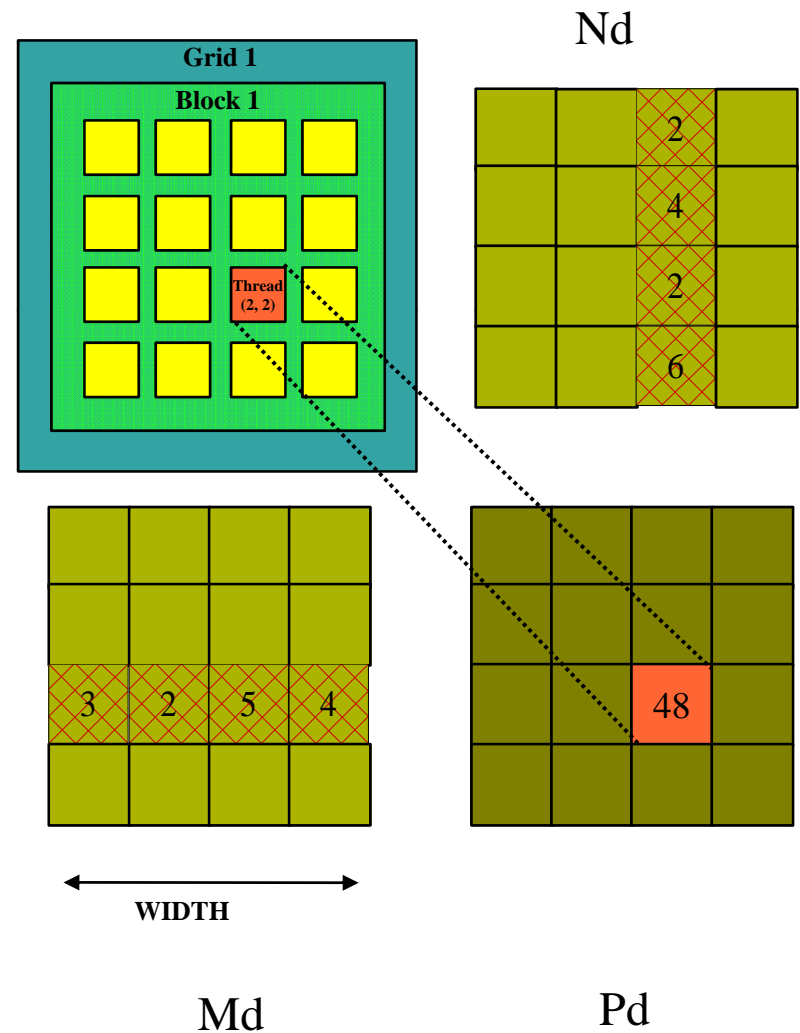
```
dim3 dimGrid(1, 1);  
dim3 dimBlock(Width, Width);
```

```
// Launch the device computation threads!
```

```
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

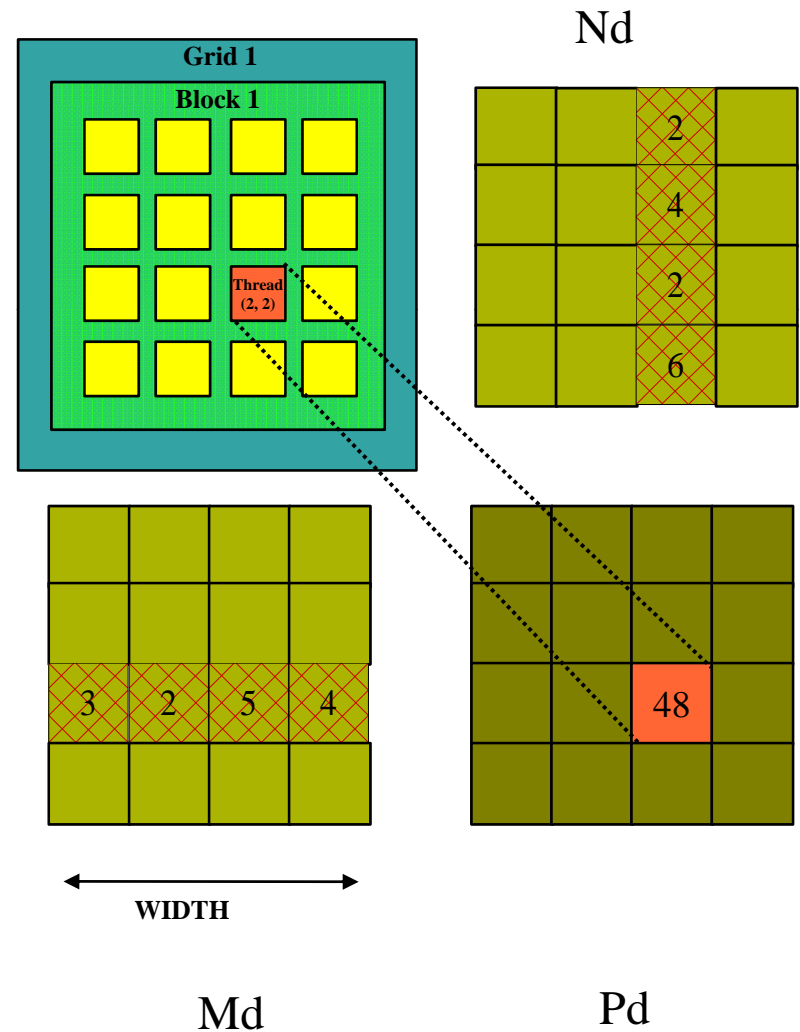
Issues with the First Mapping Scheme

- **One Block of threads compute matrix Pd**
 - **Other Multi-processors are not used.**



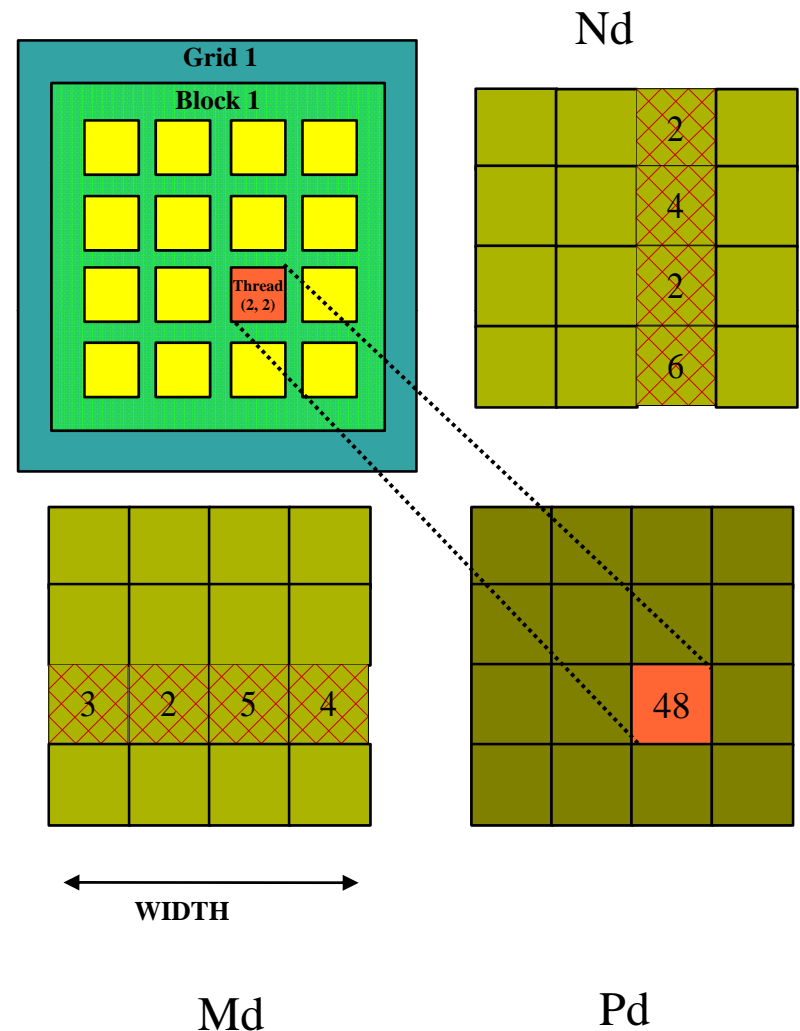
Issues with the First Mapping Scheme

- **Each thread**
 - Loads a row of matrix M_d
 - Loads a column of matrix N_d
 - Perform one multiply and addition for each pair of M_d and N_d elements
 - Compute to off-chip memory access ratio close to 1:1 (not very high)



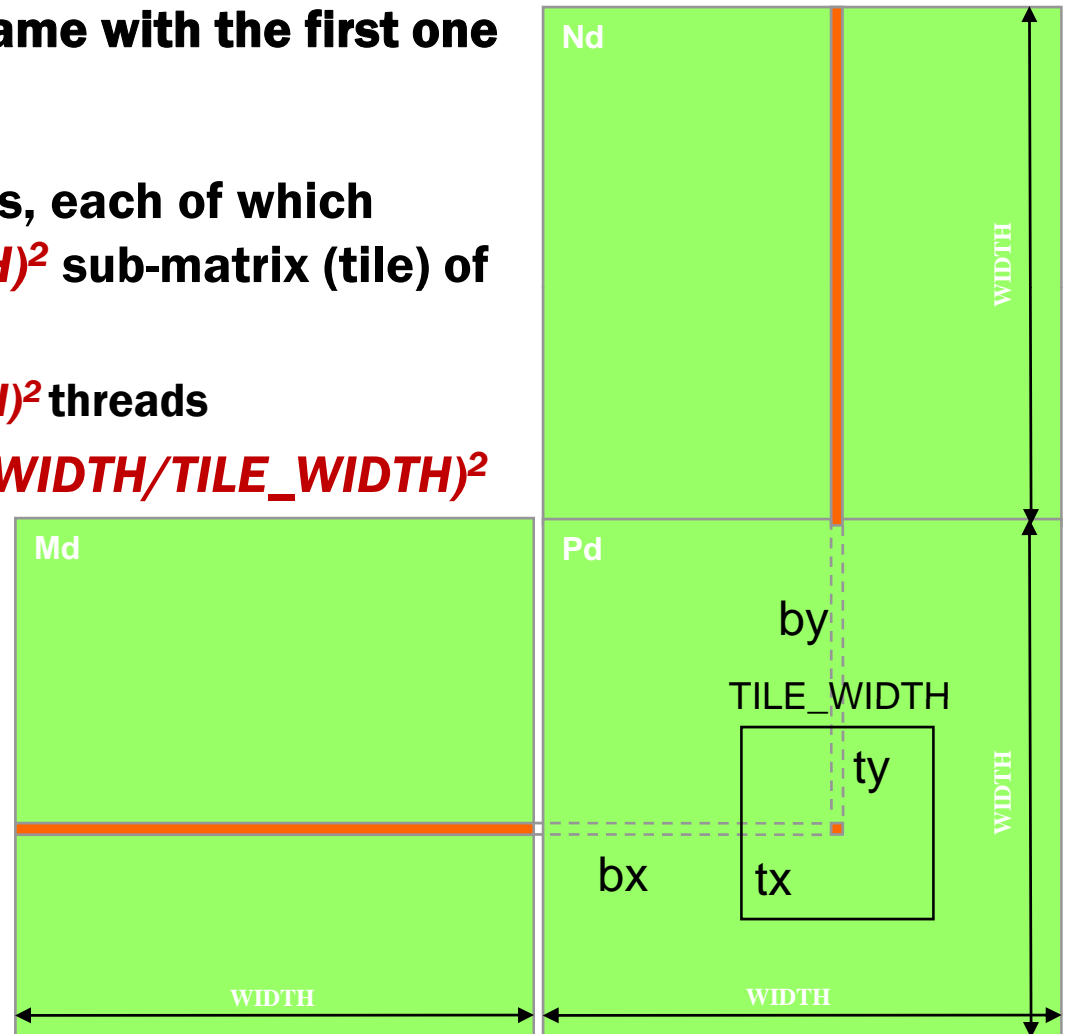
Issues with the First Mapping Scheme

- **Size of matrix limited by the number of threads allowed in a thread block**
 - **Maximum threads per block: 512**
 - **Can only do 22 x 22 matrix**
 - **You can put a loop around the kernel call for cases when Width > 22. But multiple kernel launch will cost you.**



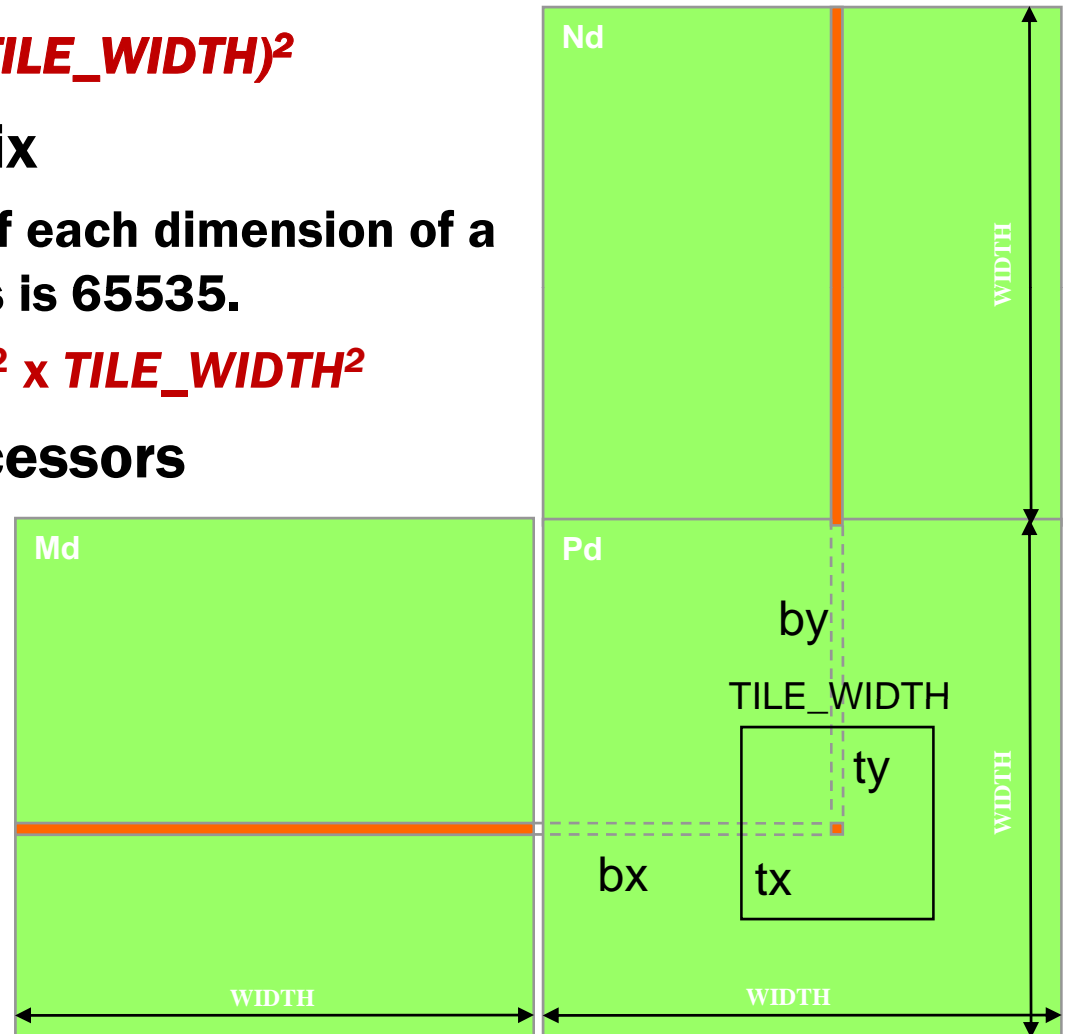
Solution: the Second Mapping Scheme

- **Thread mapping:** the same with the first one
- **Block mapping:**
 - Create 2D thread blocks, each of which 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



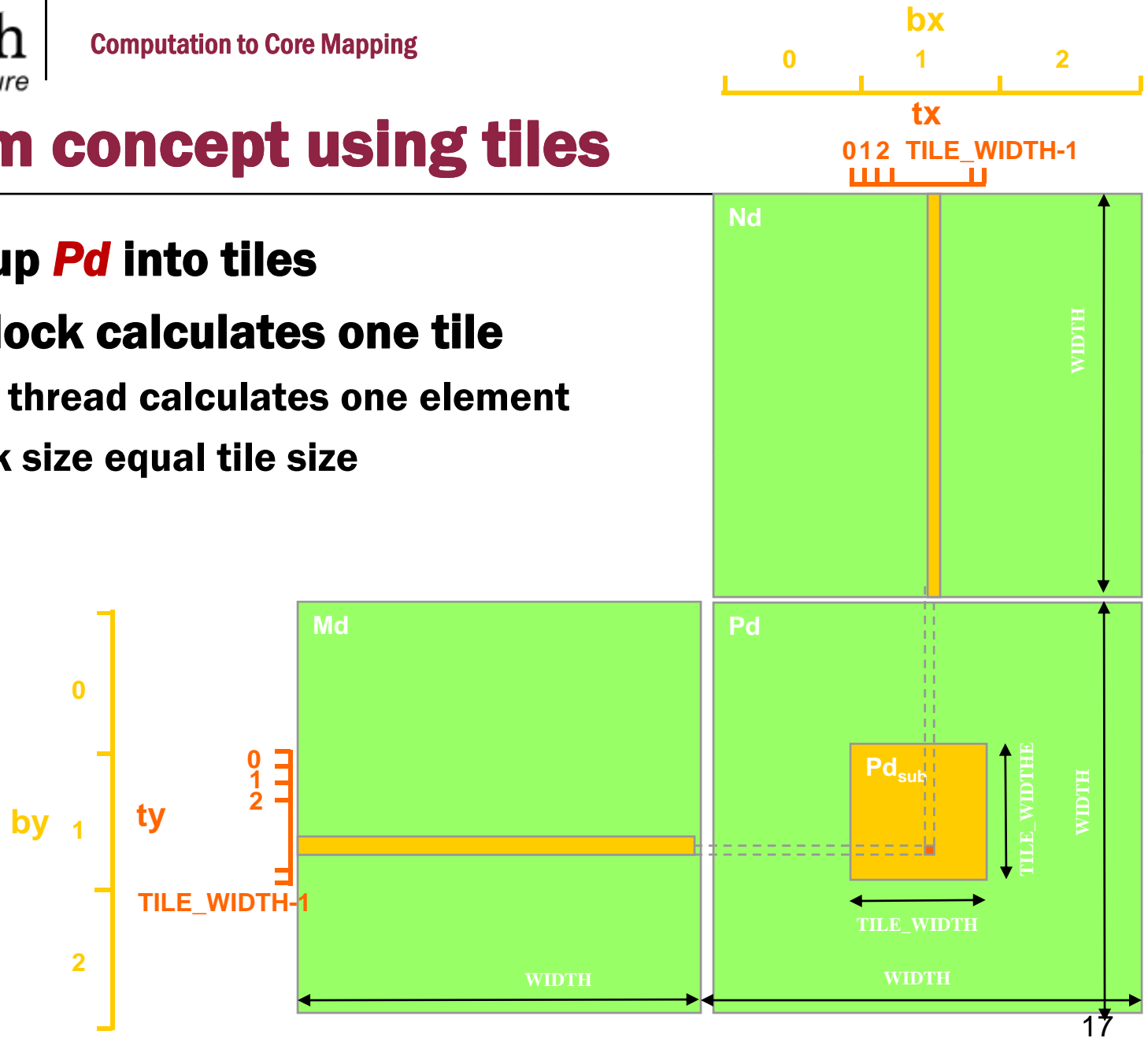
About the Second Mapping

- **More blocks** $(WIDTH/TILE_WIDTH)^2$
 - **Support larger matrix**
 - The maximum size of each dimension of a grid of thread blocks is 65535.
 - Max Width = $65535^2 \times TILE_WIDTH^2$
 - **Use more multi-processors**

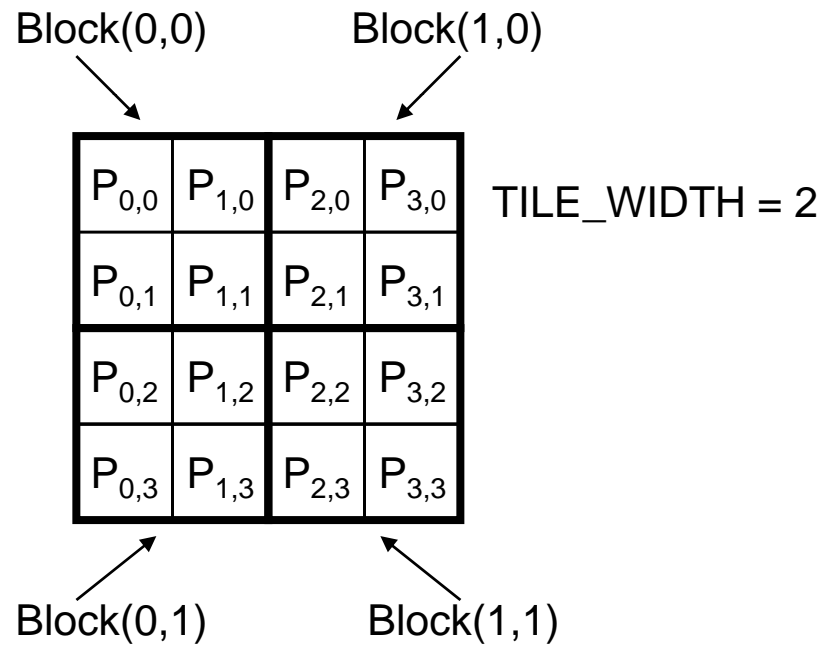


Algorithm concept using tiles

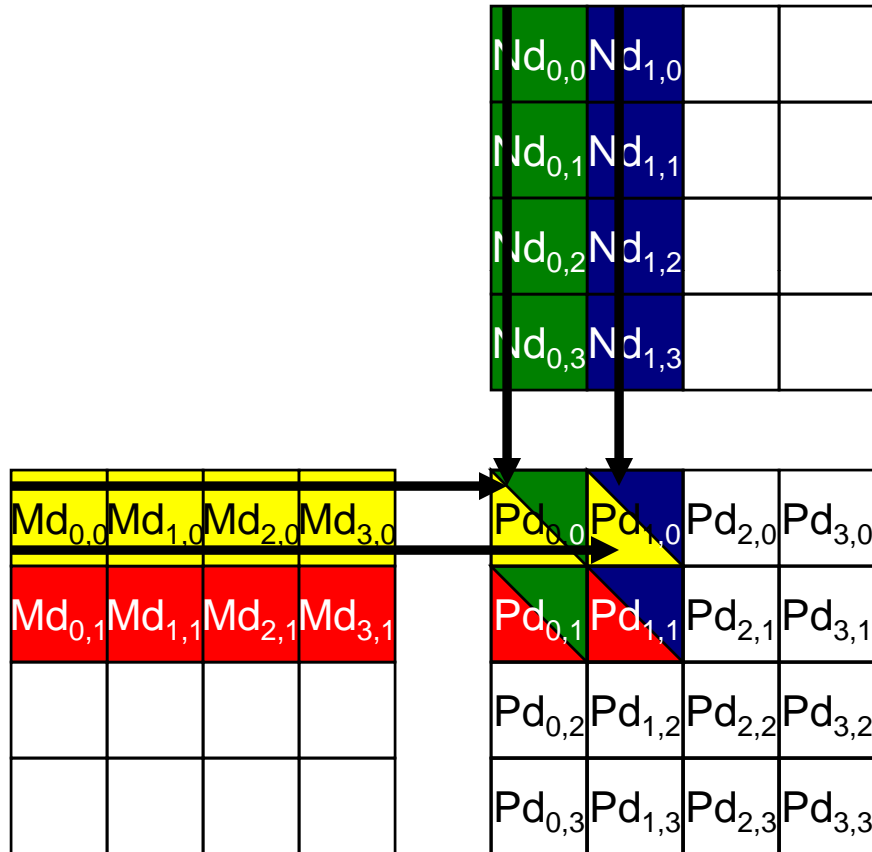
- **Break-up Pd into tiles**
- **Each block calculates one tile**
 - **Each thread calculates one element**
 - **Block size equal tile size**



Example



Block Computation



Kernel Code using Tiles

```
__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 idenx of Pd and N
    int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

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

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

Revised Kernel Invocation (Host Code)

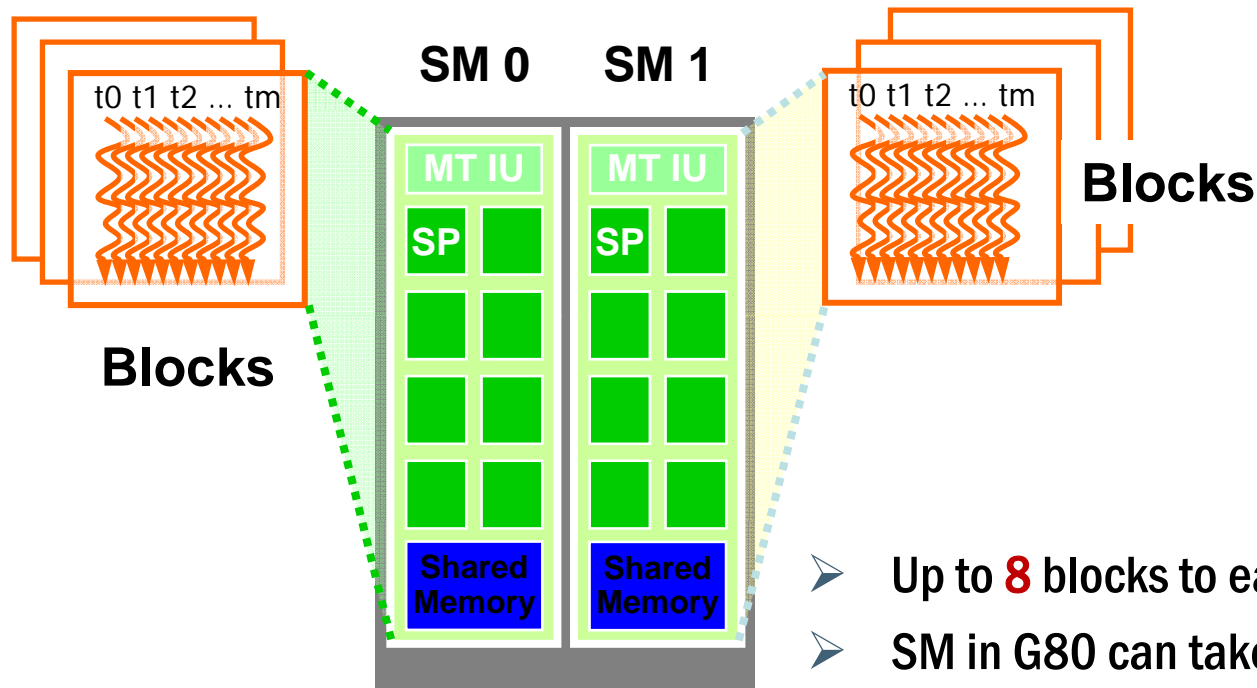
```
// Setup the execution configuration
    dim3 dimGrid (Width/TILE_WIDTH, Width/TILE_WIDTH);
    dim3 dimBlock (TILE_WIDTH, TILE_WIDTH);

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

Questions?

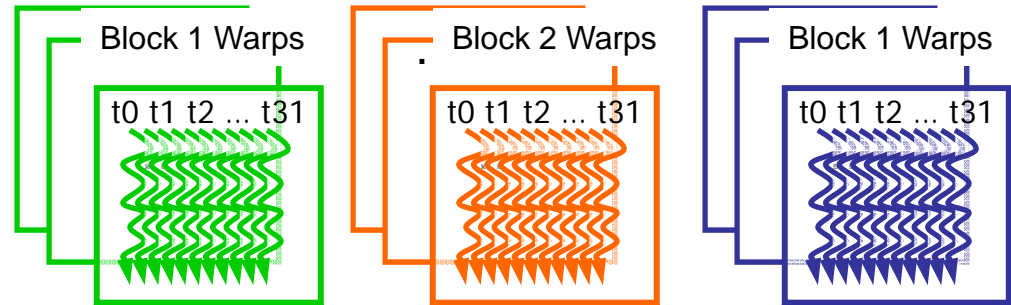
- **For Matrix Multiplication using multiple blocks, should I use **8X8**, **16X16** or **32X32** blocks?**
- **Why?**

Block Scheduling

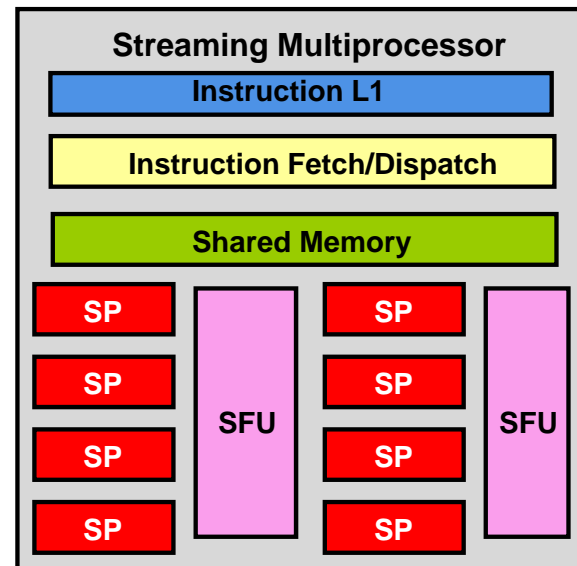


- Up to **8** blocks to each SM as resource allows
- SM in G80 can take up to **768** threads
 - Could be 256 (threads/block) * 3 blocks
 - Or 128 (threads/block) * 6 blocks, etc.
- SM in GT200 can take up to **1024** threads

Thread scheduling in Multiprocessing



- Each Block is executed as 32-thread Warps
- If 3 blocks are assigned to an SM and each block has 256 threads, how many Warps are there in an SM?
 - Each Block is divided into $256/32 = 8$ Warps
 - There are $8 * 3 = 24$ Warps



Occupancy of Multiprocessor

- **How much a Multiprocessor is occupied:**
Occupancy = Actually warps / Totally allowed
 - **GT200 SM allows 32 warps**
 - **G80 SM allow 24 warps**
- **For example:**
 - **One block per SM, 32 threads per block**
 - $(32/32) / 32 = 3.125\%$ (Very bad)
 - **4 blocks per SM, 256 threads per block**
 - $(256/32) * 4 / 32 = 100\%$ (Very good)

CUDA Occupancy Calculator

- **There are three factors:**
 - **Maximum number of warps**
 - **Maximum registers usage**
 - **Maximum share memory usage**
- **Demo**

Answers to Our Questions

- **For Matrix Multiplication using multiple blocks, should I use **8X8**, **16X16** or **32X32** blocks?**
- **For G80 GPU:**
 - **For 8X8, we have 64 threads per Block. Since each SM can take up to 768 threads, there are 12 Blocks. However, each SM can only take up to 8 Blocks, only 512 threads will go into each SM! (Occupancy = 66.6%)**
 - **For 16X16, we have 256 threads per Block. Since each SM can take up to 768 threads, it can take up to 3 Blocks and achieve full capacity unless other resource considerations overrule. (Occupancy = 100%)**
 - **For 32X32, we have 1024 threads per Block. Not even one can fit into an SM! (Can not support)**

Answers to Our Questions (Cont')

- **For Matrix Multiplication using multiple blocks, should I use **8X8**, **16X16** or **32X32** blocks?**
- **For **GT200** GPU:**
 - **For 8X8, we have 64 threads per Block. Since each SM can take up to 1024 threads, there are 16 Blocks. However, each SM can only take up to 8 Blocks, only 512 threads will go into each SM! (Occupancy = 50%)**
 - **For 16X16, we have 256 threads per Block. Each SM takes 4 Blocks and achieve full capacity unless other resource considerations overrule. (Occupancy = 100%)**
 - **For 32X32, we have 1024 threads per Block. Each SM takes 1 Block and achieve full capacity unless other resource considerations overrule. (Occupancy = 100%)**

Computation-to-Core Mapping

➤ **Step 1:**

- **Define your computational unit, map each unit to a thread**
 - **Avoid dependency**
 - **Increase compute to memory access ratio**

➤ **Step 2:**

- **Group your threads into blocks**
 - **Eliminate hardware limit**
 - **Take advantage of shared memory**