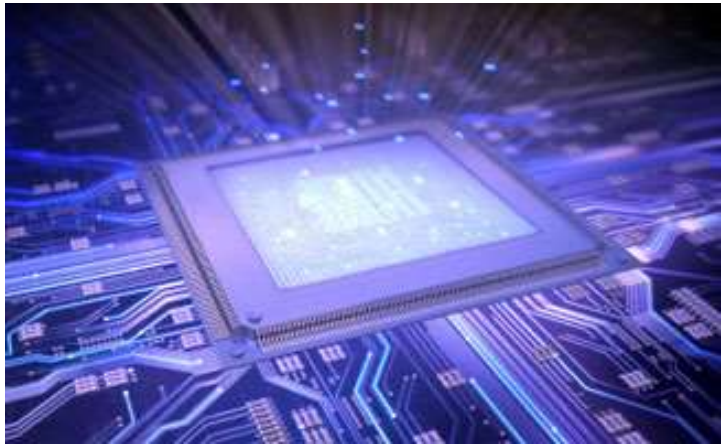
NYU
New York University

# Parallel Computing

## CUDA II

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com



# Software <-> Hardware

- From a programmer's perspective:
  - Blocks
  - Kernel
  - Threads
  - Grid
- Hardware Implementation:
  - SMs
  - SPs (per SM)
  - Warps

# Some Restrictions First

- All threads in a grid execute the same kernel code.
- A grid is organized as a 1D, 2D, or 3D array of blocks (gridDim.x, gridDim.y, and gridDim.z)
- Each block is organized as 1D, 2D, or3D array of threads (blockDim.x, blockDim.y, and blockDim.z)
- Once a kernel is launched, its dimensions cannot change.
- All blocks in a grid have the same dimension.
- The total size of a block, in terms of number of threads, has an upper bound
- Once assigned to an SM, the block must execute in its entirety by the SM

# Compute Capability

- It is a number in the form of x.y
- A standard way to expose hardware resources to applications.
- CUDA compute capability starts with 1.0 and latest one is 8.x (as of today)
- API: cudaGetDeviceProperties()

```
cudaError_t cudaGetDeviceProperties(
        struct cudaDeviceProp * prop,
        int device)
```

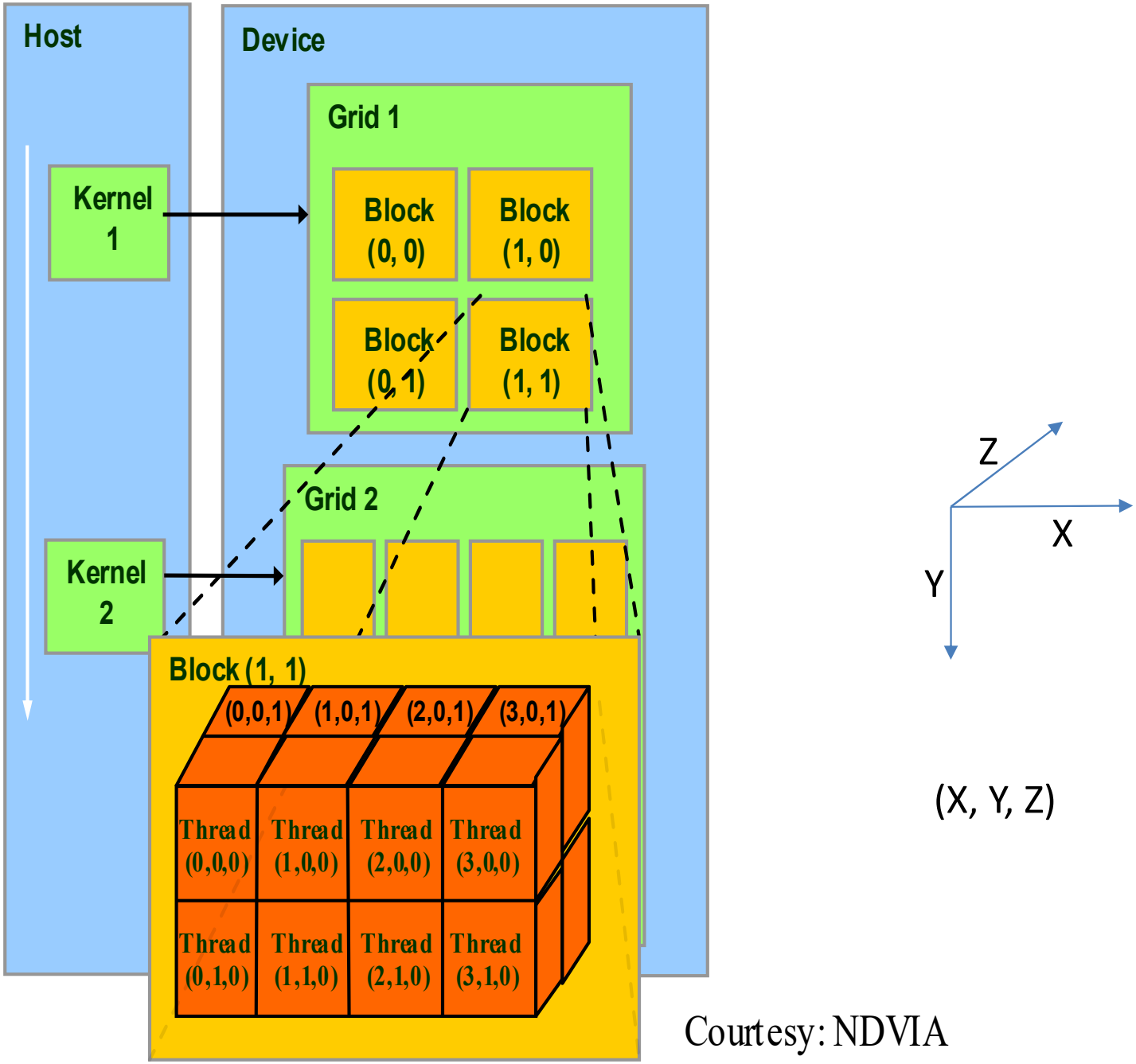
```
struct cudaDeviceProp {
        char name[256];
        size_t  totalGlobalMem;  /* in bytes */
        size_t sharedMemPerBlock;  /* in bytes */
        int  regsPerBlock;
        int warpSize;
        int maxThreadsPerBlock;
        int maxThreadsDim[3];
        int maxGridSize[3];
        int clockRate;  /* in KHz */
        size_t totalConstMem;
        int major;    int minor;
        int multiProcessorCount;
        int concurrentKernels;
        int unifiedAddressing;
        int memoryClockRate;
        int memoryBusWidth;
        int l2CacheSize;
        int maxThreadsPerMultiProcessor;
        … and a lot of other stuff}
```

```
cudaError_t
    cudaGetDeviceCount(
            int * count )
```

# Compute Capability Example

Table 1. A Comparison of Maxwell GM107 to Kepler GK107

| GPU | GK107 (Kepler) | GM107 (Maxwell) |
|---|---|---|
| CUDA Cores | 384 | 640 |
| Base Clock | 1058 MHz | 1020 MHz |
| GPU Boost Clock | N/A | 1085 MHz |
| GFLOP/s | 812.5 | 1305.6 |
| Compute Capability | 3.0 | 5.0 |
| Shared Memory / SM | 16KB / 48 KB | 64 KB |
| Register File Size / SM | 256 KB | 256 KB |
| Active Blocks / SM | 16 | 32 |
| Memory Clock | 5000 MHz | 5400 MHz |
| Memory Bandwidth | 80 GB/s | 86.4 GB/s |
| L2 Cache Size | 256 KB | 2048 KB |
| TDP | 64W | 60W |
| Transistors | 1.3 Billion | 1.87 Billion |
| Die Size | 118 $mm^2$ | 148 $mm^2$ |
| Manufactoring Process | 28 nm | 28 nm |




Courtesy: NDVIA

# Revisiting 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 * Width + k];
      float Ndelement = Nd[k * Width + tx];
      Pvalue += Mdelement * Ndelement;
   }

   // Write the matrix to device memory each thread writes one element
   Pd[ty * Width + tx] = Pvalue;
}
```

This is what we did before…
What is the main shortcoming??

# Revisiting Matrix Multiplication

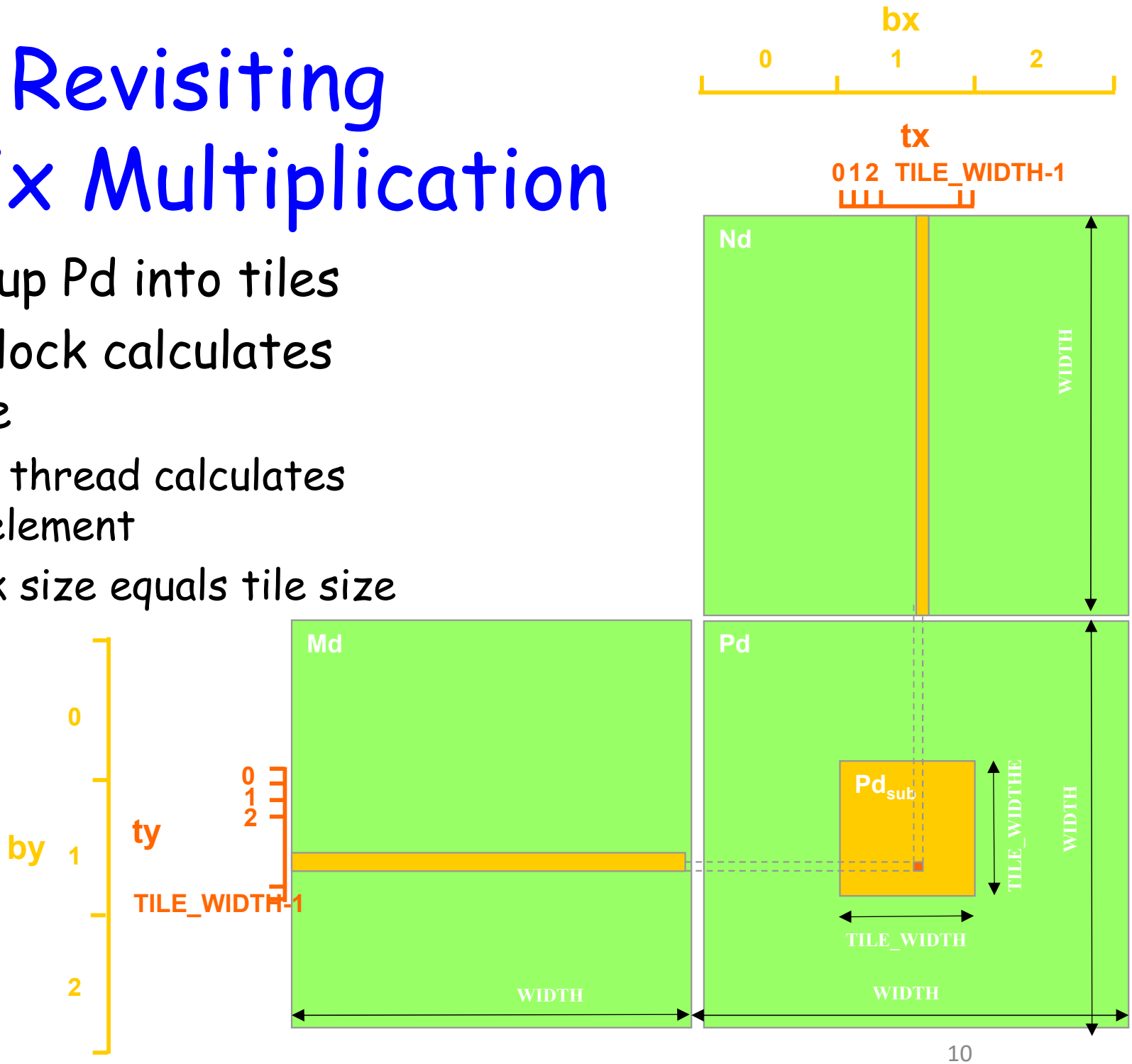
```
// Matrix multiplication kernel - thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
   // 2D Thread ID
   int tx = threadIdx.x;
   int ty = threadIdx.y;

   // 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 * Width + k];
      float Ndelement = Nd[k * Width + tx];
      Pvalue += Mdelement * Ndelement;
   }

   // Write the matrix to device memory each thread writes one element
   Pd[ty * Width + tx] = Pvalue;
}
```

Can only handle 16 elements in each dimension!

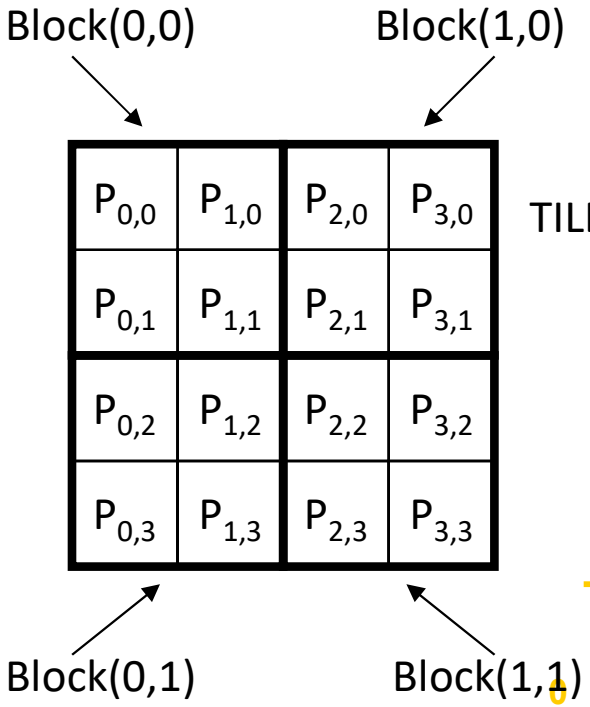
**Reason:**
**We used 1 block, and a block in most GPUs is limited to 512 threads (1024 in newer GPUs)**

# Revisiting Matrix Multiplication

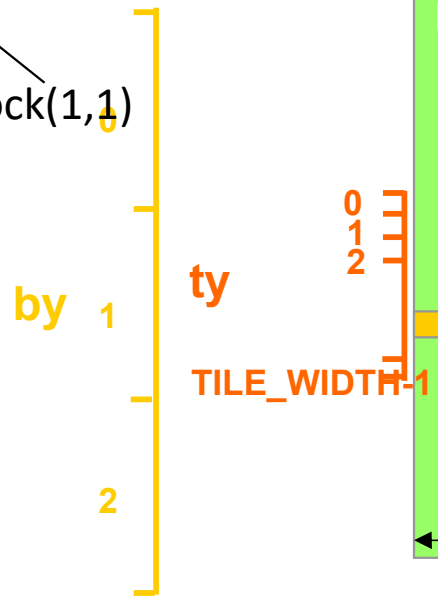
- Break-up Pd into tiles
- Each block calculates one tile
  - Each thread calculates one element
  - Block size equals tile size

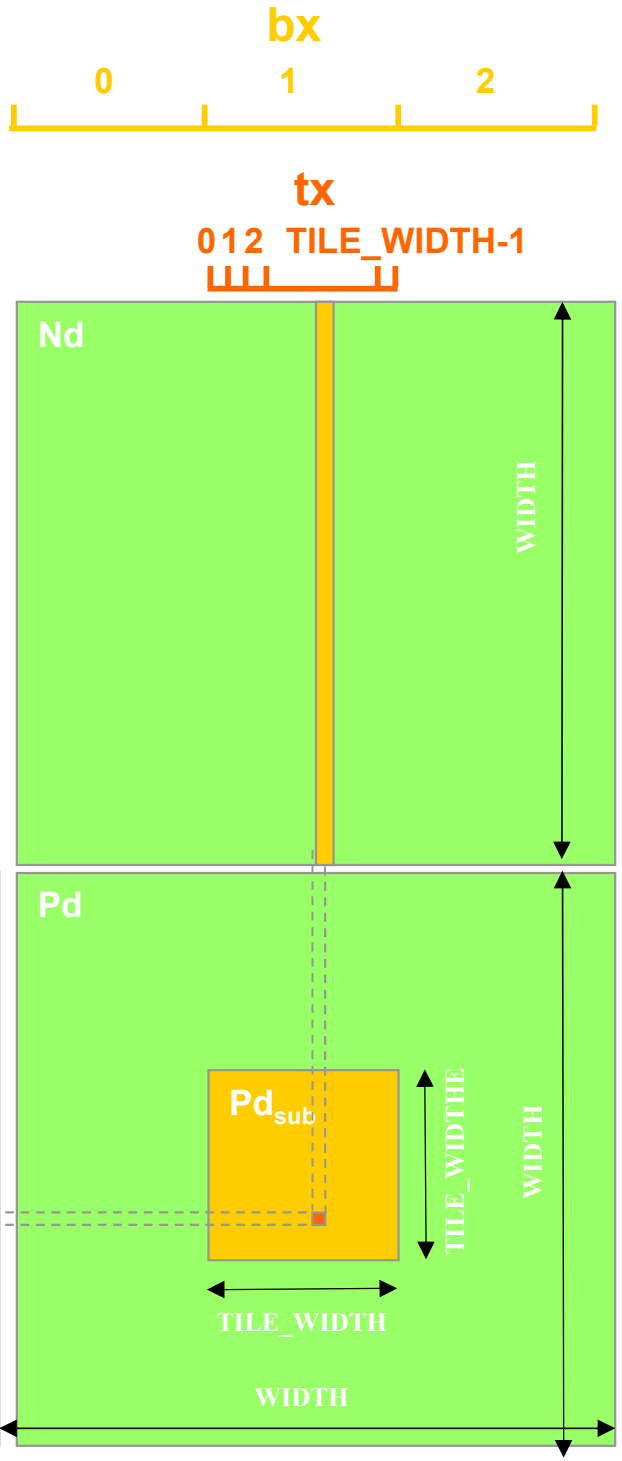


# Revisiting Matrix Multiplication




TILE_WIDTH = 2



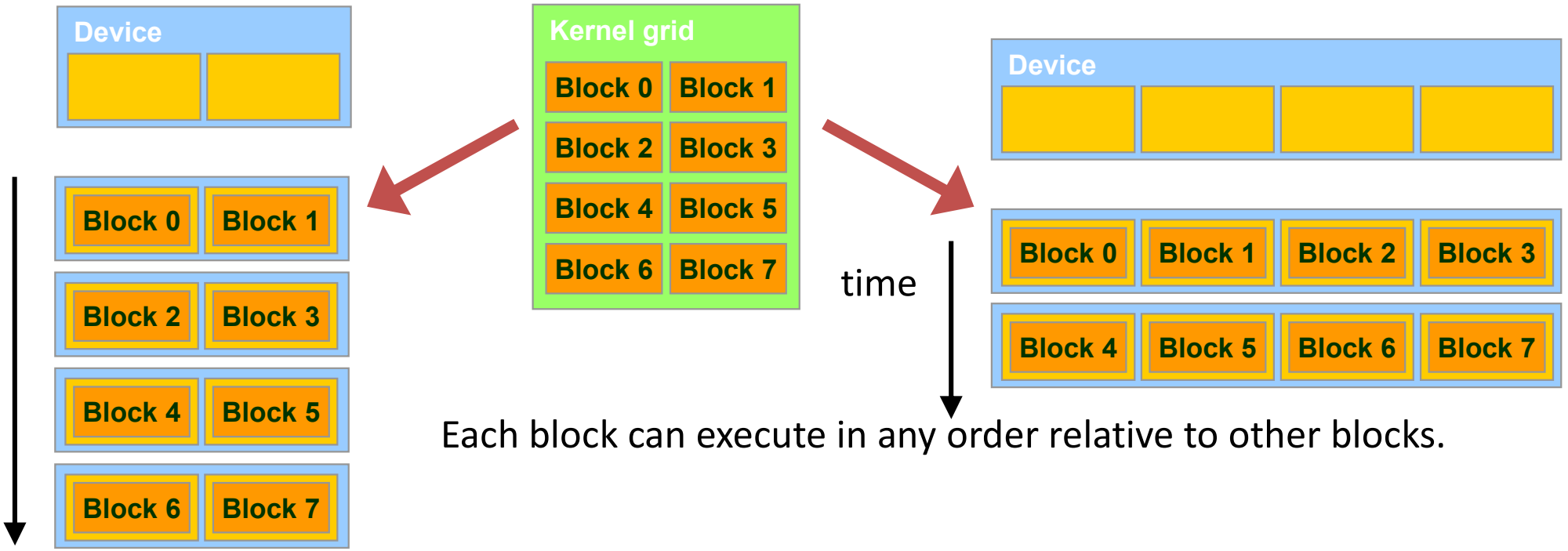
# Revisiting Matrix Multiplication

```
// 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);


__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;
}
```

# Synchronization

__syncthreads()

- called by a kernel function
- The thread that makes the call will be held at the calling location until every thread in the block reaches the location
- **Beware** of if-then-else
- **Threads in different blocks cannot synchronize** -> CUDA runtime system can execute blocks in any order




Each block can execute in any order relative to other blocks.

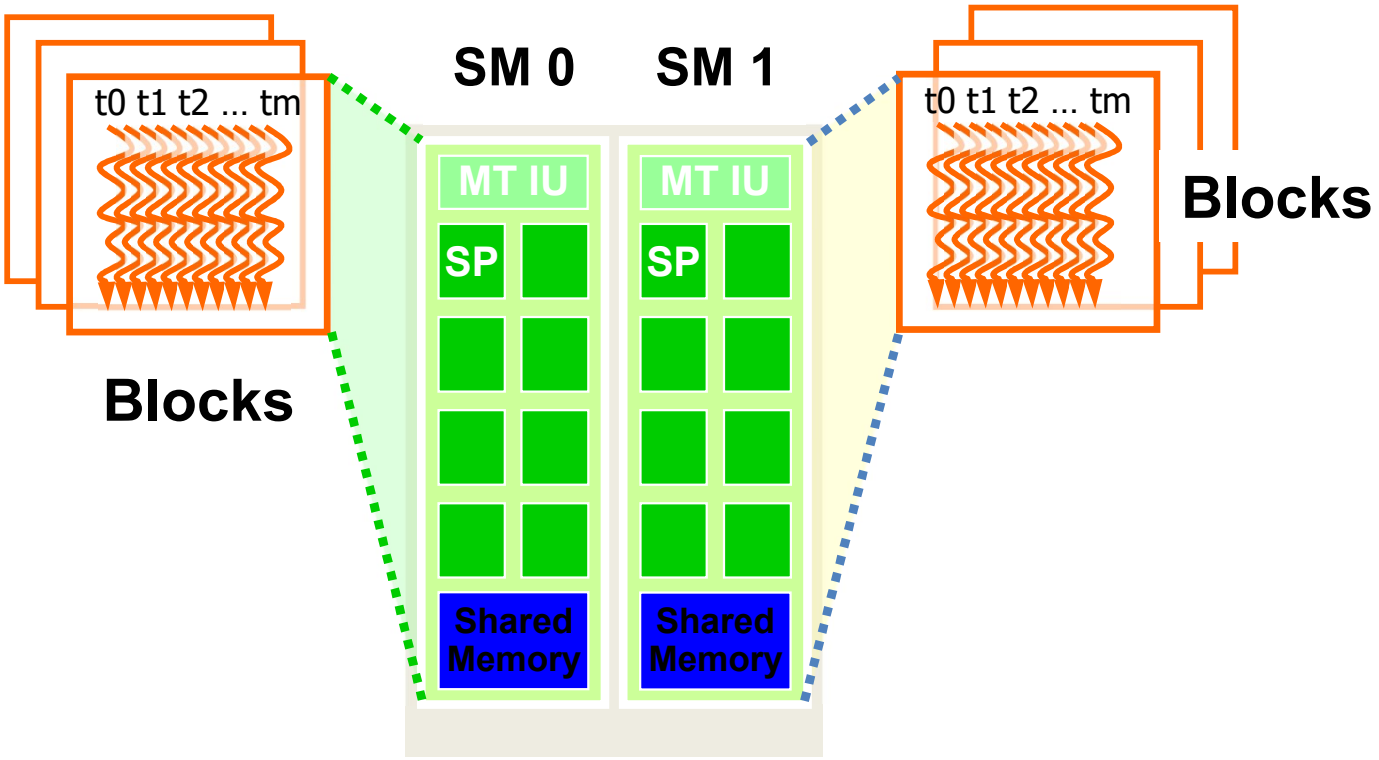
The ability to execute the same application code on hardware with different number of execution resources is called **transparent scalability**

# Scheduling of Blocks

- To be assigned to an SM, a block needs to have all its resources (registers, shared memory, number of threads, …) assigned beforehand.
- CUDA runtime automatically reduces number of blocks assigned to each SM until resource usage is under limit.
- Runtime system:
  - maintains a list of blocks that need to execute
  - assigns new blocks to SM as they compute previously assigned blocks

# What Is a Resource?

- Characteristics of a resource:
  - Must be inside the SM
  - Must be determined before kernel launch
- Example of SM resources
  - number of threads that can be simultaneously tracked and scheduled.
  - Registers
  - Shared memory




GT200 can accommodate 8 blocks/SM and up to 1024 threads can be assigned to an SM.

What are our choices for number of blocks and number of threads/block?

**Thread scheduling is an implementation concept.**
**So, outside the programmer's control.**

# Warps

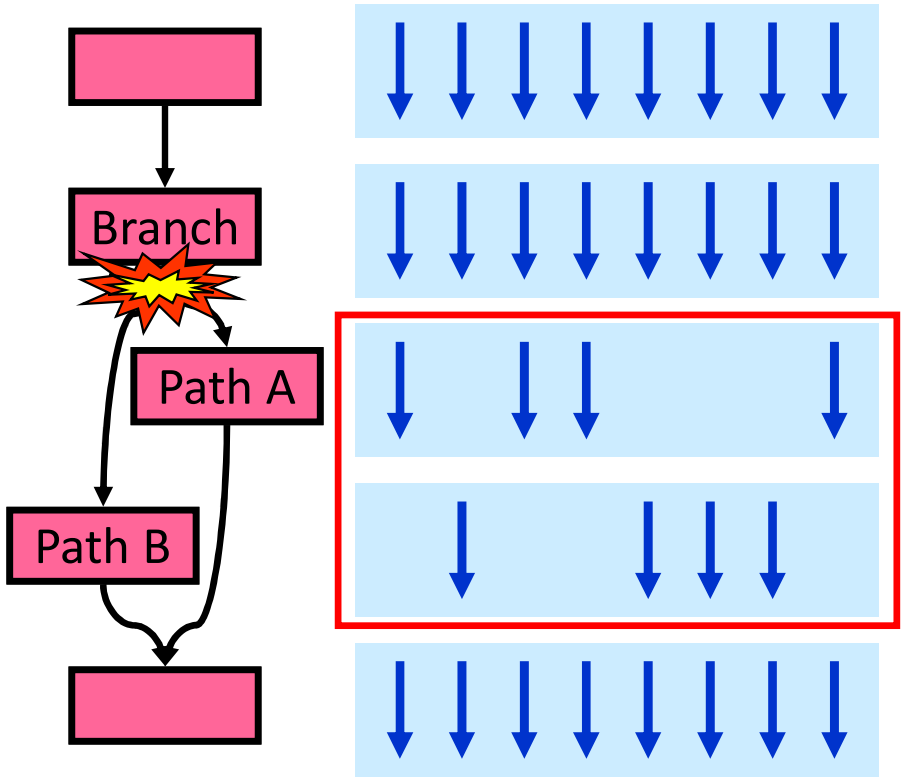
- Once a block is assigned to an SM, it is divided into units called warps.
  - Thread IDs within a warp are consecutive and increasing
    - Warp 0 starts with Thread ID 0 till thread ID 31, and so one.
- Warp size is implementation specific.
  - But so far all NVIDIA GPUs have warp = 32 threads.
- Warp is unit of thread scheduling in SMs

# Warps

- Partitioning is always the same
- We cannot determine which warp finishes first.
- Each warp is executed in a SIMD fashion (i.e. all threads within a warp must execute the same instruction at any given time).
  - Problem: branch divergence

# Branch Divergence in Warps

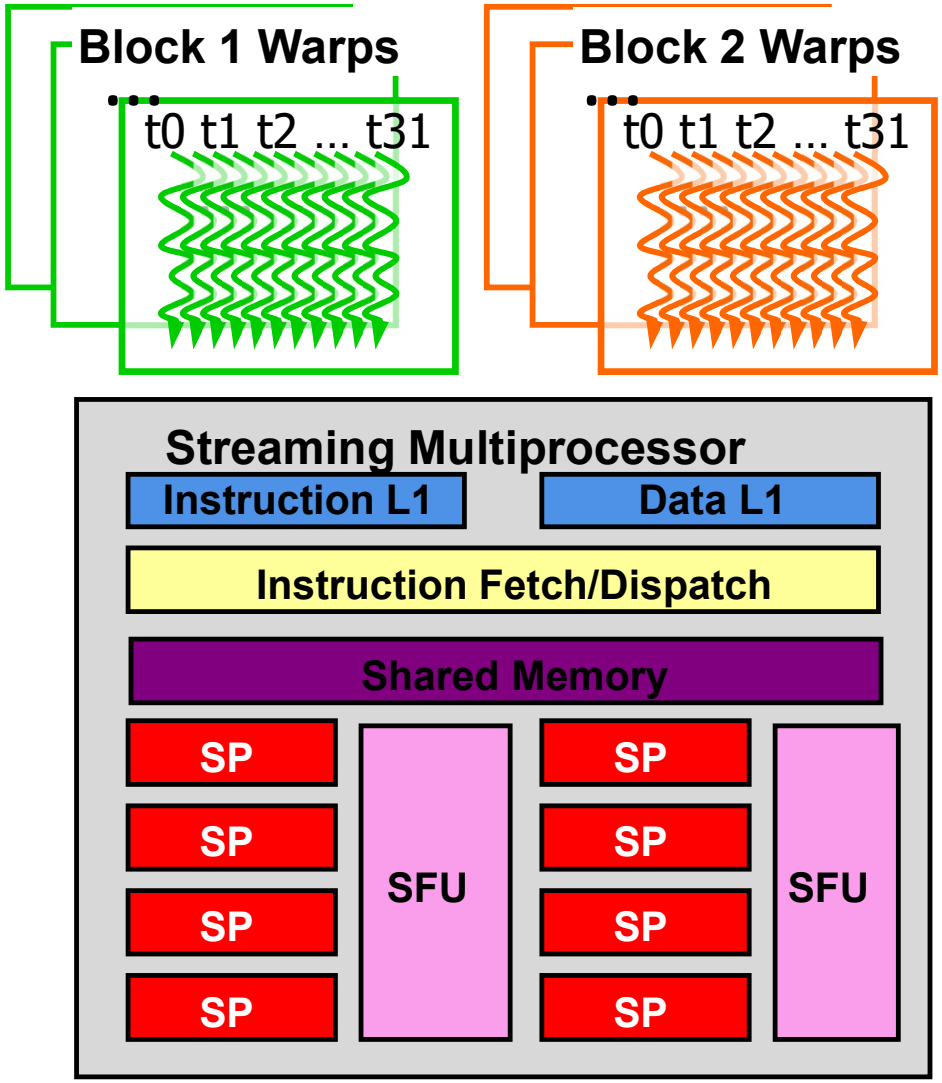
- occurs when threads inside warps branches to different execution paths.




**50% performance loss**

# Latency Tolerance

- When an instruction executed by the threads in a warp must wait for the result of a previously initiated long-latency operation, the warp is not selected for execution -> latency hiding
- Scheduling does not introduce idle time -> zero-overhead thread scheduling
- Scheduling is used for tolerating long-latency operations.




This ability of tolerating long-latency operation is the main reason why GPUs do not dedicate as much chip area to cache memory and branch prediction mechanisms as traditional CPUs.

# Exercise

The GT200 has the following specs (maximum numbers):

- 512 threads/block
- 1024 threads/SM
- 8 blocks/SM
- 32 threads/warp

What is the best configuration for thread blocks to implement matrix multiplications

8x8, 16x16, or 32x32?

# Myths About CUDA

- GPUs have very wide (1000s) SIMD machines
  - No, a CUDA Warp is only 32 threads
- Branching is not possible on GPUs
  - Incorrect.
- GPUs are power-inefficient
  - Nope, performance per watt is quite good
- CUDA is only for C or C++ programmers
  - Not true, there are third party wrappers for Java, Python, and more

# Conclusion

- We must be aware of the restrictions imposed by hardware:
  - threads/SM
  - blocks/SM
  - threads/blocks
  - threads/warps
- The only safe way to synchronize threads in different blocks is to terminate the kernel and start a new kernel for the activities after the synchronization point