

CPSC/ECE 4780/6780

General-Purpose Computation on Graphical Processing Units (GPGPU)

Lecture 4: CUDA Threads

Recaps from Last Lecture

- What is CUDA?
- CUDA programing structure
- Processing flow of a CUDA program
- CUDA memory management and data transfer
- CUDA function declaration
- CUDA threads organization
- CUDA device properties

Vector Addition Using Blocks

- On the device:
 - Each block execute in parallel
 - blockIdx.x is used to index into the array

```
__global__ void add(int *a, int *b, int *c) {  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

- On the host:
 - Many blocks with one thread each

```
// Launch add() kernel on GPU with N blocks  
add<<<N, 1>>>>(d_a, d_b, d_c);
```

Vector Addition Using Threads

- On the device:
 - Parallel threads are used instead of parallel blocks
 - threadIdx.x is used to index into the array

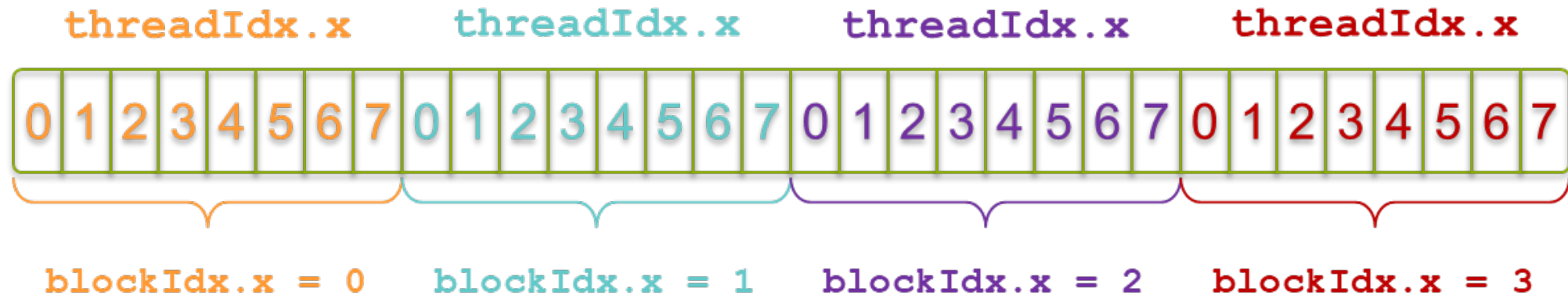
```
__global__ void add(int *a, int *b, int *c) {  
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];  
}
```

- On the host:
 - One block with many threads

```
// Launch add() kernel on GPU with N threads  
add<<<1, N>>>(d_a, d_b, d_c);
```

Indexing Arrays with Blocks and Threads

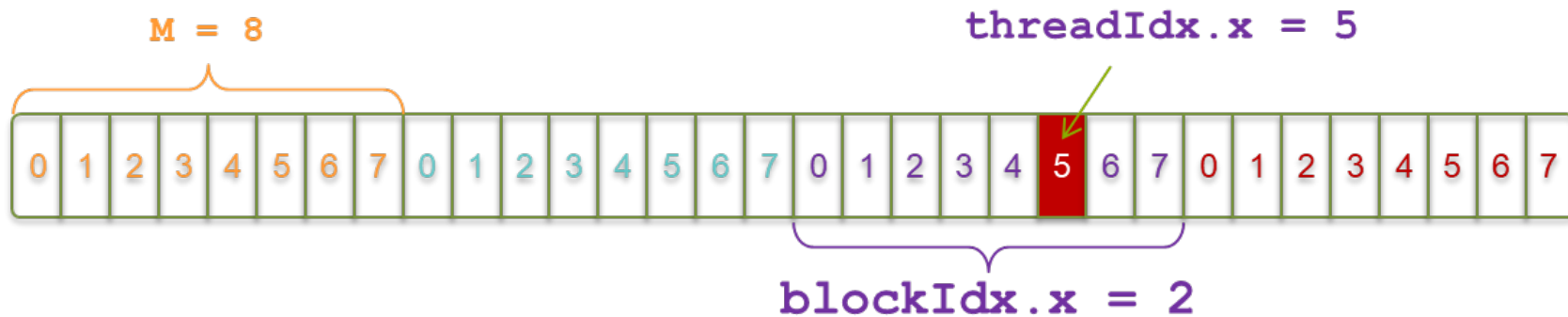
- No longer as simple as using `blockIdx.x` and `threadIdx.x`
 - Suppose we have a device with 4 blocks, and 8 threads/block



- Consider indexing an array (32 elements) with one element per thread
 - Neither `<<<32, 1>>>` nor `<<<1, 32>>` works

Indexing Arrays with Blocks and Threads

- Which thread will operate on the red element?



- With M threads/block, a unique index of each thread is given by:
$$\text{int index} = \text{threadIdx.x} + \text{blockIdx.x} * M;$$

Vector Addition Using Blocks and Threads

- On the device:
 - Use blockIdx.x to access block index
 - Use threadIdx.x to access thread index within block

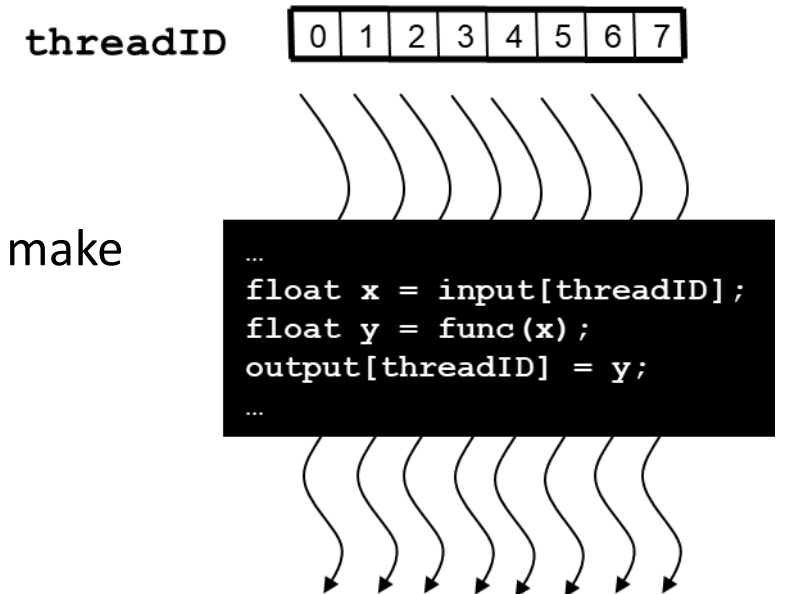
```
__global__ void add(int *a, int *b, int *c, int n) {  
    int index = threadIdx.x + blockIdx.x * blockDim.x;  
    if (index < n)  
        c[index] = a[index] + b[index];  
}
```

- On the host:
 - Multiple blocks with many threads

```
// Launch add() kernel on GPU  
add<<<(N + M - 1) / M, M>>>(d_a, d_b, d_c, N);
```

CUDA Thread Organization

- CUDA programs are a hierarchy of concurrent threads
 - Threads are grouped into thread blocks
 - All threads within a thread block run in the same SM
 - Threads of the same block can communicate
 - Thread blocks conform a grid
 - All threads in a grid execute the same kernel function
 - Each thread has an ID to compute memory address and make control decisions



Compute a Thread Index

- Coordinates of a thread

- blockIdx: 1D or 2D
- threadIdx: 1D, 2D, or 3D

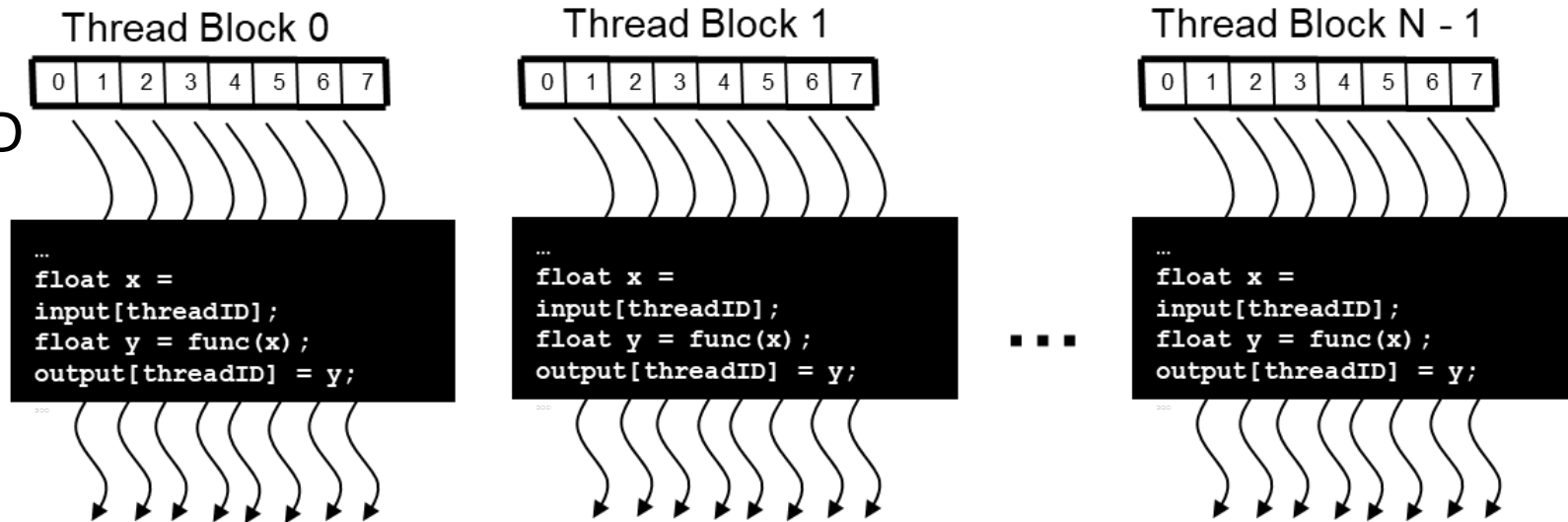
- Dimensions

- blockDim
- gridDim

- Thread index

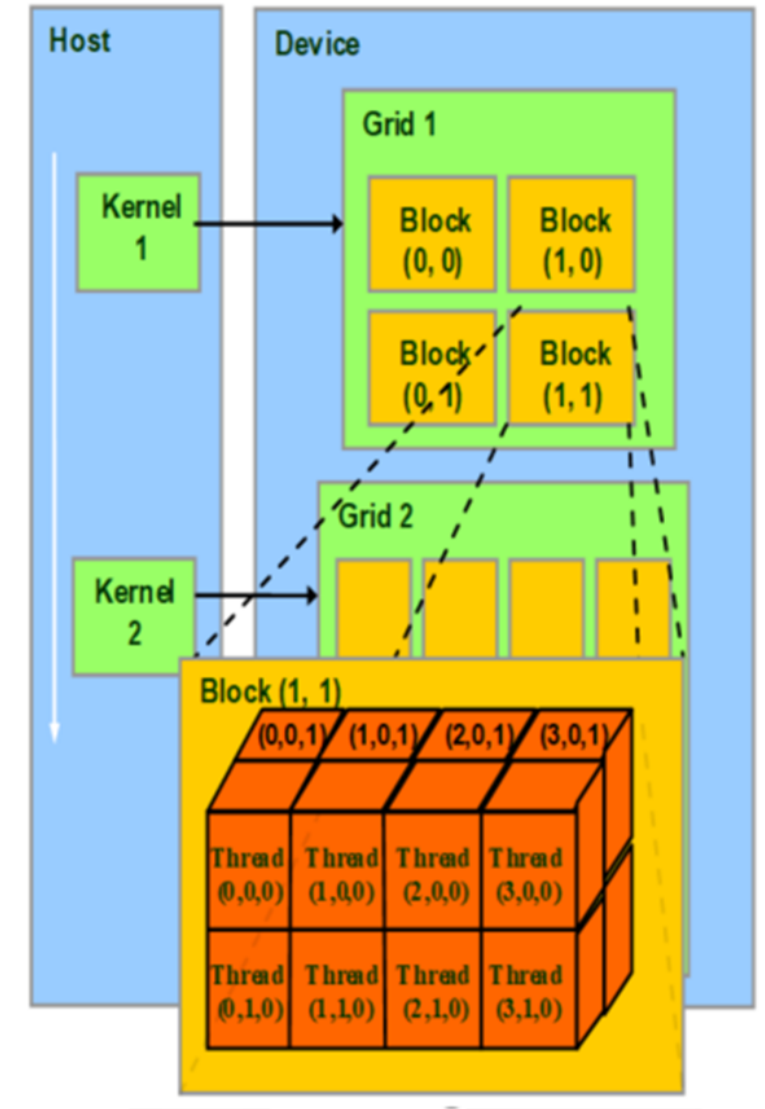
- $\text{threadID} = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$

- In general, a grid is organized as a 2D array of blocks; each block is organized into a 3D array of threads



Subdivision of a Grid

- Kernel Function `<<<dimGrid, dimBlock>>>(...);`
 - `dim3 dimGrid(128, 1, 1);`
 - `dim3 dimBlock(32, 1, 1);`
- 1D grid (Scalar values can be used if a grid or block has only one dimension)
 - Kernel function `<<<128, 32>>>(...);`
- 3D grid
 - `dim3 dimGrid(2, 2, 1);`
 - `dim3 dimBlock(4, 2, 2);`
 - Kernel function `<<<dimGrid, dimBlock>>>(...);`



Choice of Subdivision of a Grid

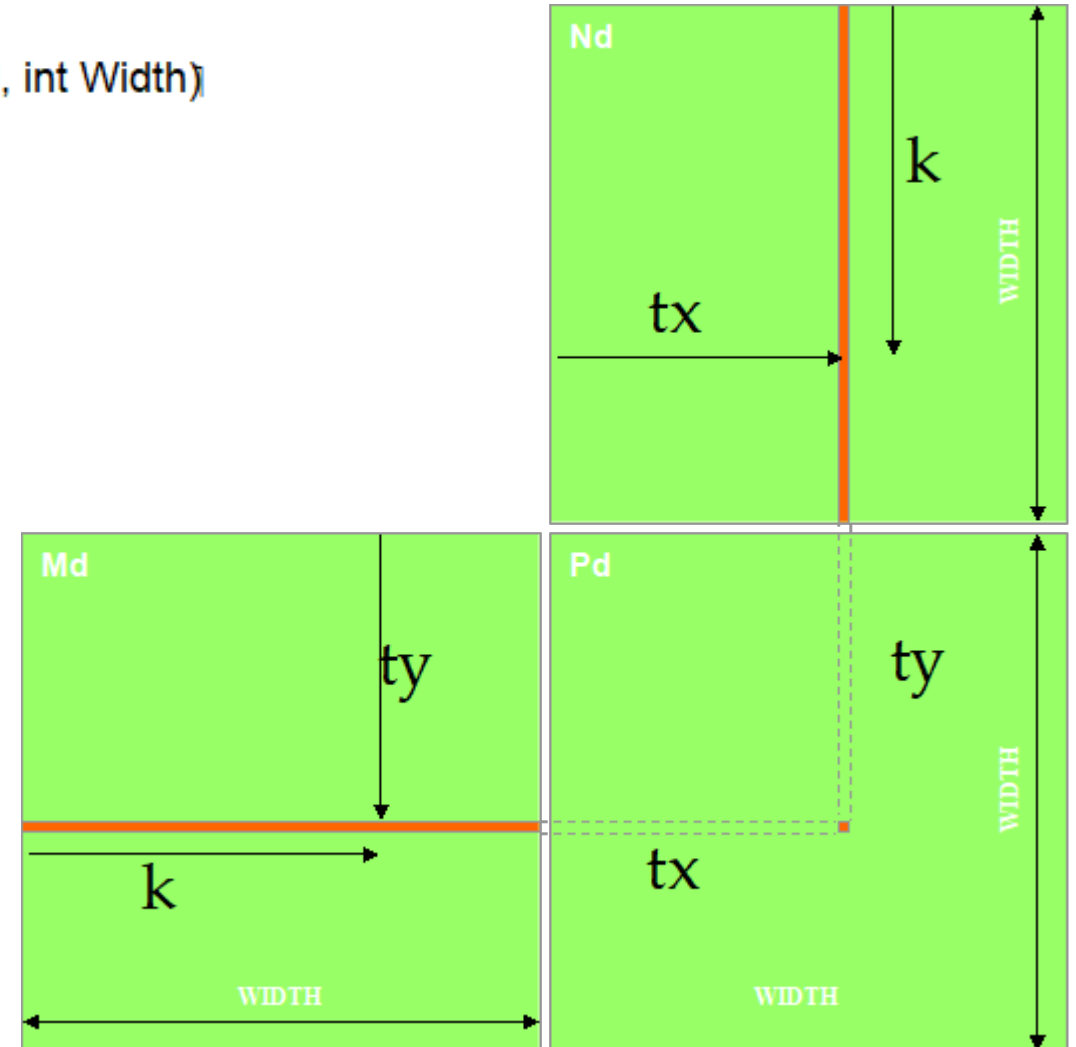
- Choice of subdivision is up to you, but with hardware limitations:
 - The values of `gridDim.x` and `gridDim.y` can range from 1 to **65,535**
 - The values of `blockIdx.x` range between 0 and `gridDim.x - 1`
 - The values of `blockIdx.y` range between 0 and `gridDim.y - 1`
 - The total size of a block is limited to **512** threads (depends on the GPU)
 - (512, 1, 1), (8, 16, 2), and (16,16,2) allowable
 - (32, 32, 1) not allowable
- You can also use `dataGetDeviceCount()` and `cudaGetDeviceProperties()` to learn more about the specifics of your system

```
Device 0: "Tesla K20m"
CUDA Driver Version / Runtime Version      9.2 / 8.0
CUDA Capability Major/Minor version number: 3.5
Total amount of global memory:              4744 MBytes (4974313472 bytes)
(13) Multiprocessors, (192) CUDA Cores/MP:  2496 CUDA Cores
GPU Max Clock rate:                         706 MHz (0.71 GHz)
Memory Clock rate:                          2600 Mhz
Memory Bus Width:                           320-bit
L2 Cache Size:                              1310720 bytes
Maximum Texture Dimension Size (x,y,z)      1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
Maximum Layered 1D Texture Size, (num) layers 1D=(16384), 2048 layers
Maximum Layered 2D Texture Size, (num) layers 2D=(16384, 16384), 2048 layers
Total amount of constant memory:             65536 bytes
Total amount of shared memory per block:     49152 bytes
Total number of registers available per block: 65536
Warp size:                                   32
Maximum number of threads per multiprocessor: 2048
Maximum number of threads per block:         1024
Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
Maximum memory pitch:                       2147483647 bytes
Texture alignment:                           512 bytes
Concurrent copy and kernel execution:        Yes with 2 copy engine(s)
Run time limit on kernels:                   Yes
Integrated GPU sharing Host Memory:          No
Support host page-locked memory mapping:     Yes
Alignment requirement for Surfaces:          Yes
Device has ECC support:                      Enabled
Device supports Unified Addressing (UVA):    Yes
Device PCI Domain ID / Bus ID / location ID: 0 / 36 / 0
Compute Mode:
    < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
```

Revisiting Matrix Multiplication

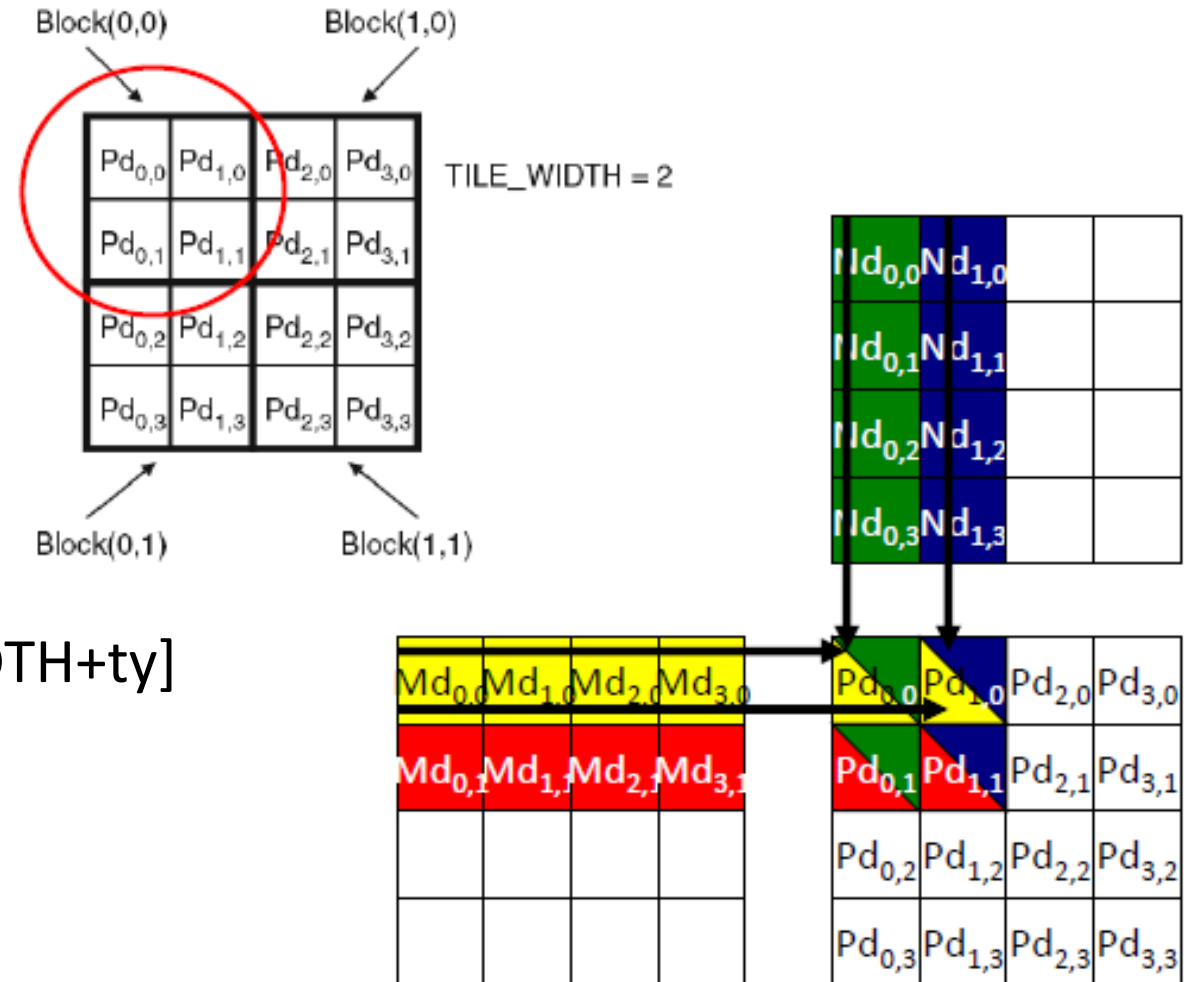
```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    float Pvalue = 0;
    for (int k = 0; k < Width; ++k) {
        float Melement = Md[threadIdx.y*Width+k];
        float Nelement = Nd[k*Width+threadIdx.x];
        Pvalue += Melement * Nelement;
    }
    Pd[threadIdx.y*Width+threadIdx.x] = Pvalue;
}
```

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



Solution: Use Multiple Thread Blocks

- Break Pd into square tiles
- Each block calculates one tile
 - Block size equals tile size
 - Each thread (tx, ty) in block (bx, by) calculates one element
- Pd element index:
 - $\text{Pd}[\text{bx} * \text{TILE_WIDTH} + \text{tx}][\text{by} * \text{TILE_WIDTH} + \text{ty}]$
- Row index of Md:
 - $\text{by} * \text{TILE_WIDTH} + \text{ty}$
- Col index of Nd:
 - $\text{bx} * \text{TILE_WIDTH} + \text{tx}$



Revised Matrix Multiplication Code

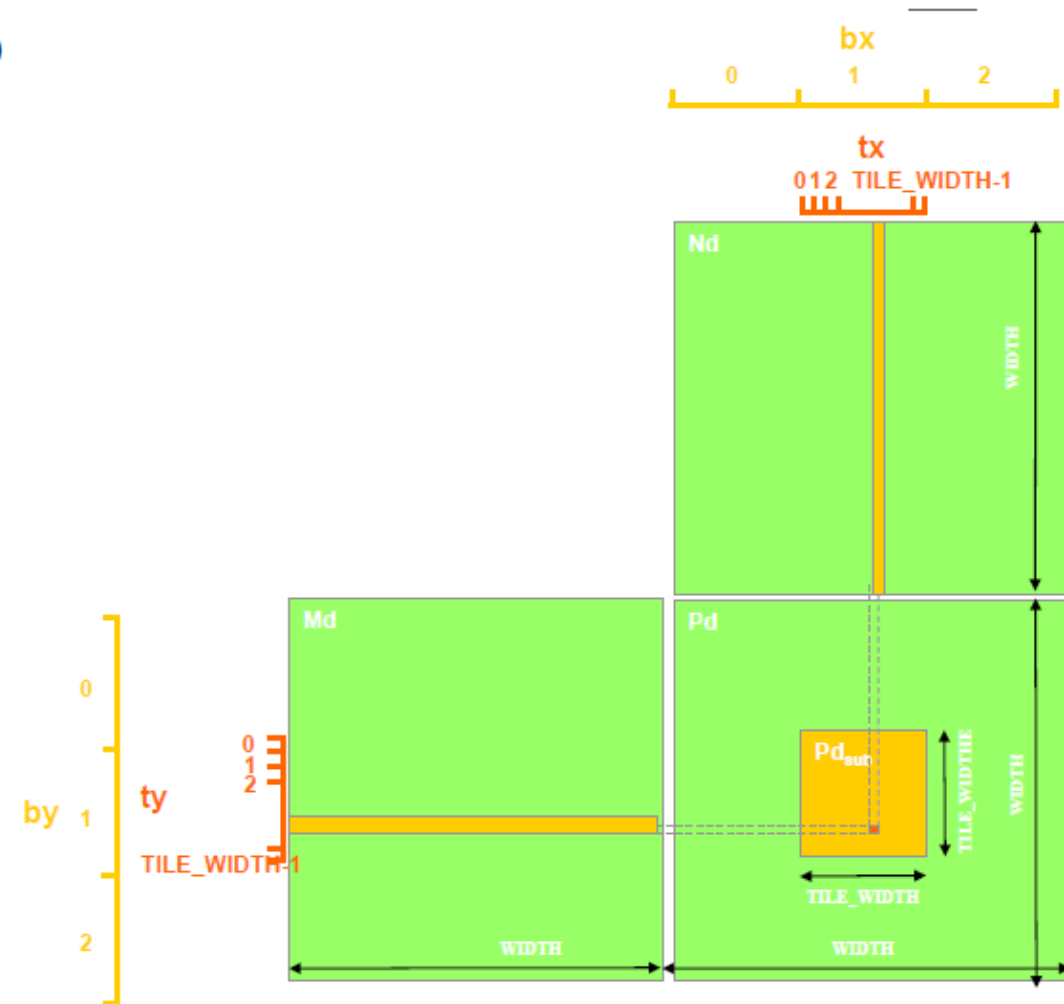
```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // Calculate the row index of the Pd element and M
    int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
    // Calculate the column index of Pd and N
    int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

    float Pvalue = 0;
    // each thread computes one 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;
}
```

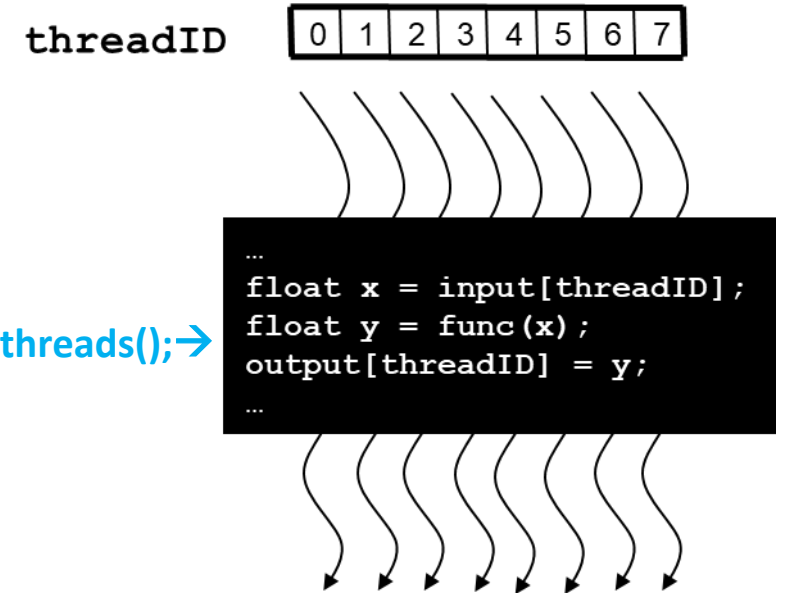
```
// Setup the execution configuration
dim3 dimGrid(Width/TILE_WIDTH, Width/TILE_WIDTH);
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);

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



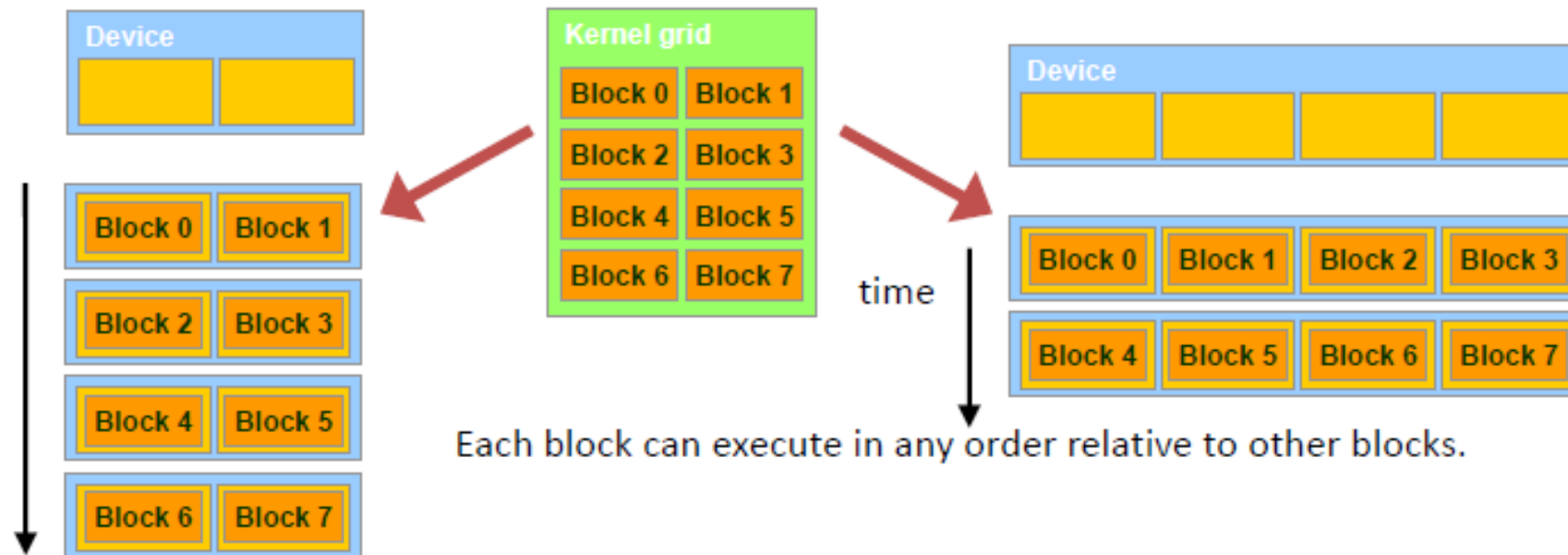
Thread Synchronization

- `__syncthreads()`
 - Called by a kernel function
 - Holds the thread which executes the function call at the calling location
 - Until every thread in the block reaches the location `__syncthreads();` →
- A `__syncthreads()` statement must be executed by all threads in a block!
- Execution constraints:
 - Threads in a block should execute in close time proximity to avoid excessively long waiting times
 - Threads in different blocks cannot synchronize → CUDA runtime system can execute blocks in any order



Transparent Scalability

- **Transparent scalability** is the ability to execute the same application code on hardware with different number of execution resources

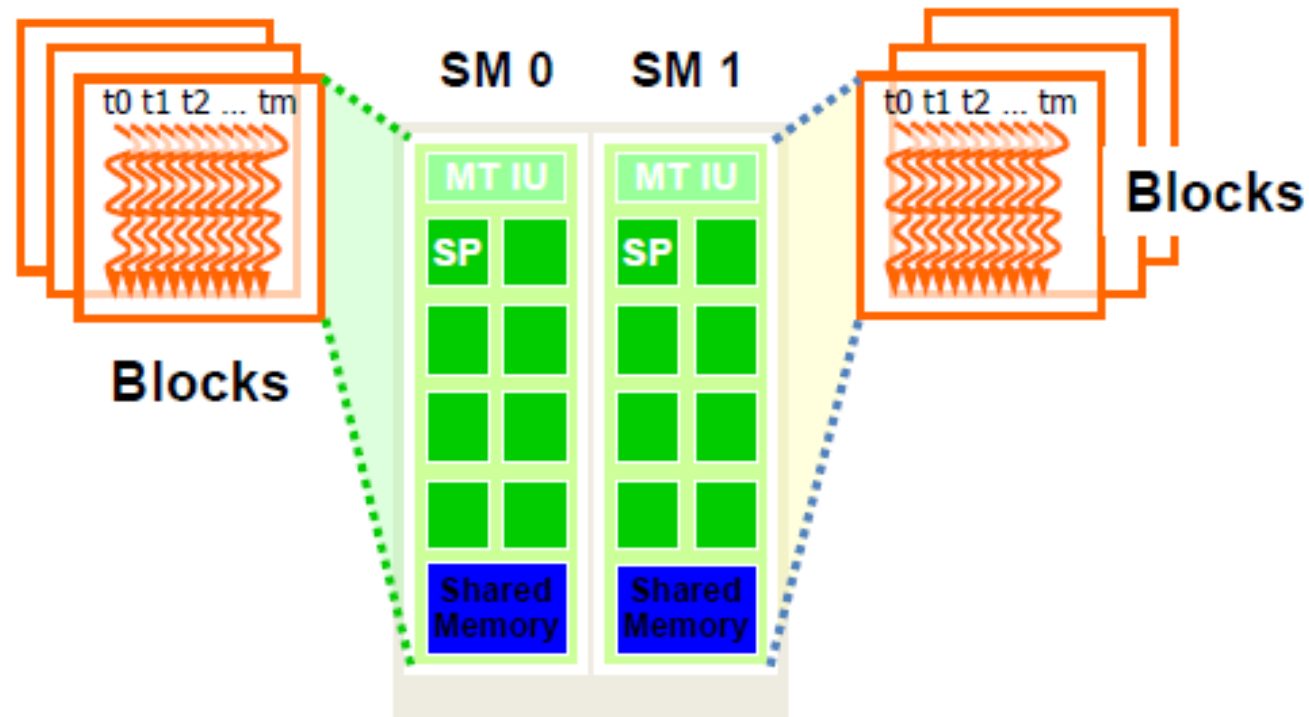


- It reduces the burden on application developers and improves the usability of application

Thread Assignment

- Threads are assigned to execution resources on a block-by-block basis
- Execution resources are organized into streaming multiprocessors (SMs)
 - E.g., NVIDIA GT200 implementation has 30 SMs with up to 8 blocks can be assigned to each SM
- CUDA runtime automatically reduces the number of blocks assigned to each SM until the resource usage is under the limit
- Runtime system:
 - Maintains a list of blocks that need to execute
 - Assigns new blocks to SM as they complete the execution of blocks previously assigned to them

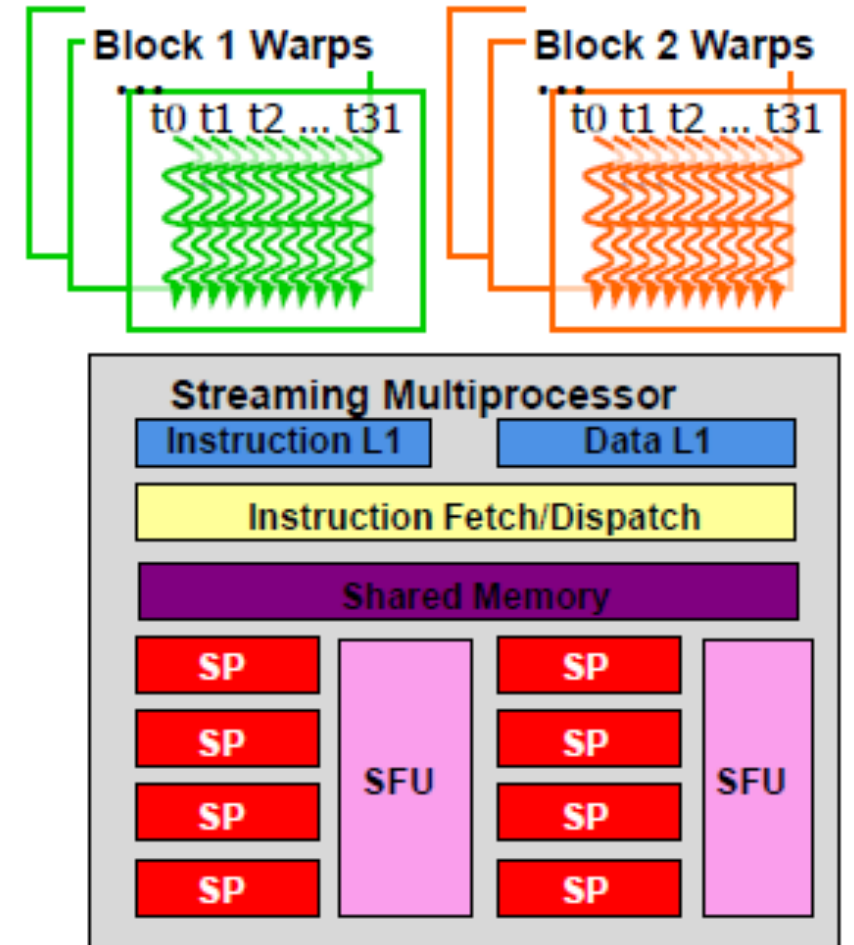
Thread Block Assignment to GT200 SMs



- NVIDIA GT200 has 30 SMs, up to 1024 threads can be assigned to each SM => up to 30,720 threads can be simultaneously residing in the SMs for execution
- NVIDIA GT200 can only accommodate up to 8 blocks/SM

Thread Scheduling

- Thread Scheduling is an implementation concept related to specific hardware implementation
- **Warps**
 - Once a block is assigned to an SM, it is further divided into thread units called warps
 - Thread IDs within a warp are consecutive and increasing
 - Warp 0 starts with Thread ID 0
 - Warp size is implementation specific; partitioning is always the same
 - E.g., GT200: 32-thread units
 - Warp is the unit of thread scheduling in SMs
 - Each warp is executed in a SIMD fashion



Latency Tolerance

- **Latency hiding:** filling the latency of expensive operations with work from other threads
 - Execute warp which is not waiting for long-latency operation
 - Use priority mechanism to schedule ready warps
- **Zero-overhead thread scheduling:** scheduling does not introduce idle time into the execution of timeline

A Simple Exercise

- The GT200 has the following specifications:
 - Up to 1024 threads/SM
 - Up to 8 blocks/SM
 - 32 threads/warp
 - Up to 512 threads/block
 - What is the best configuration for thread blocks to implement matrix multiplications on GT200? 8x8, 16x16, or 32x32?

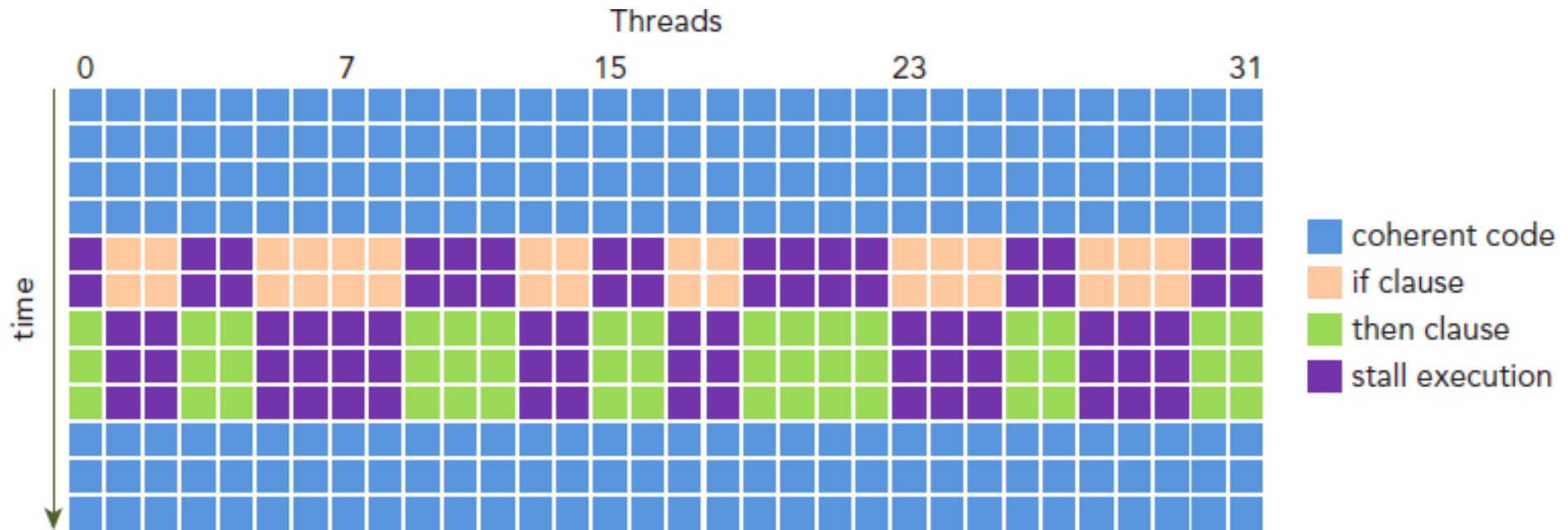
Tips: look for good thread capacity per SM to utilize SM's execution resources and large warp number per SM to schedule around long-latency operations

Warp Divergence

- CPUs include complex hardware to perform branch prediction
- GPUs are comparatively simple devices without complex branch prediction mechanisms
 - All threads in a warp must execute identical instructions
 - Warp divergence: threads in the same warp executing different instructions
 - E.g., if (cond) {
...
} else {
...
}

Warp Divergence

- If threads of a warp diverge, the warp serially executes each branch path, disabling threads that do not take that path -> significantly degrades performance



Warp Divergence

- Note that branch divergence occurs only within a warp
- Different conditional values in different warps do not cause warp divergence
- The warp assignment of threads in a thread block is deterministic -> possible to partition data in a way to ensure all threads in the same warp take the same control path in an application

Warp Divergence Example

```
__global__ void mathKernel1(float *c) {  
    int tid = blockIdx.x * blockDim.x + threadIdx.x;  
    float a, b;  
    a = b = 0.0f;  
    if (tid % 2 == 0) {  
        a = 100.0f;  
    } else {  
        b = 200.0f;  
    }  
    c[tid] = a + b;  
}
```

With warp divergence

```
__global__ void mathKernel1(float *c) {  
    int tid = blockIdx.x * blockDim.x + threadIdx.x;  
    float a, b;  
    a = b = 0.0f;  
    if ((tid / warpSize) % 2 == 0) {  
        a = 100.0f;  
    } else {  
        b = 200.0f;  
    }  
    c[tid] = a + b;  
}
```

Without warp divergence