CS 677: Parallel Programming for Many-core Processors
Lecture 2

Instructor: Philippos Mordohai

Overview

- Simple encryption example
- Blocks, threads and warps
- CUDA memory types
- Matrix Multiplication using Shared Memory
- Thread Execution and Divergence
- Atomics

Encryption Example

```
# include <iostream>
# include <cutil.h>

using namespace std;

__global__ void cuda_encrypt(char* m, int m_len, int shift)
{
   for (int i = 0; i < m_len; i++)
       m[i] = (((m[i] -'a') + shift) % 26) + 'a';
}
```

Courtesy of Werner Backes

```
int main()
{
   char message[255];
   int message_len, shift;
   char* dev_message;

   cin >> message;
   cin >> shift;
   cout << "plaintext: " << message << endl;
   message_len = strlen(message);

   cudaMalloc(&dev_message, message_len+1);
   cudaMemcpy(dev_message, message, message_len+1,
        cudaMemcpyHostToDevice);
   cuda_encrypt<<<1,1>>>(dev_message, message_len, shift);
   cudaMemcpy(message, dev_message, message_len+1,
        cudaMemcpyDeviceToHost);

   cout << "ciphertext: " << message << endl;
   return 0;
}
```

Compilation and Execution

- Compile the example program hello world.cu using the CUDA compiler nvcc.
  - nvcc -I. hello_world.cu -o hello_world
  - The option -I is used to add an include path
  - nvcc --help outputs all available compiler options

- Output:
  - Execute ./hello_world
    ```
    helloworld
    3
    plaintext: helloworld
    ciphertext: khoorzruog
    ```

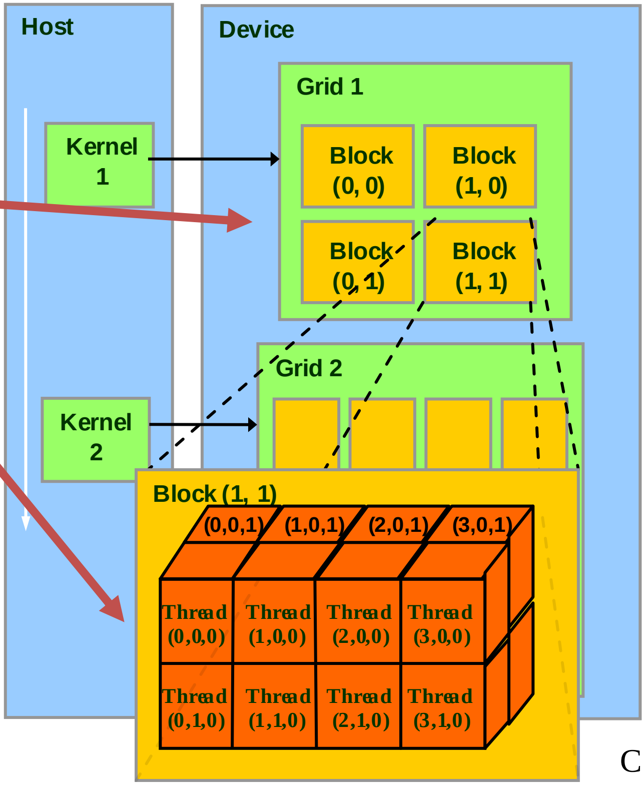
Parallel Encryption Example

```
# include <iostream>
# include <cutil.h>

using namespace std;

__global__ void cuda_encrypt(char* m, int m_len, int shift)
{
   int tid = blockIdx.x * blockDim.x + threadIdx.x;
   if (tid < m_len)
        m[tid] = (((m[tid] -'a') + shift) % 26) + 'a';
}
```

```
int main()
{
   char message[255];
   int message_len, shift;
   char* dev_message;

   cin >> message;
   cin >> shift;
   cout << "plaintext: " << message << endl;
   message_len = strlen(message);

   cudaMalloc(&dev_message, message_len+1);
   cudaMemcpy(dev_message, message, message_len+1,
        cudaMemcpyHostToDevice);
   cuda_encrypt<<<(message_len/32)+1,32>>>(dev_message, message_len,
        shift);
   cudaMemcpy(message, dev_message, message_len+1,
        cudaMemcpyDeviceToHost);

   cout << "ciphertext: " << message << endl;
   return 0;
}
```

Block IDs and Thread IDs

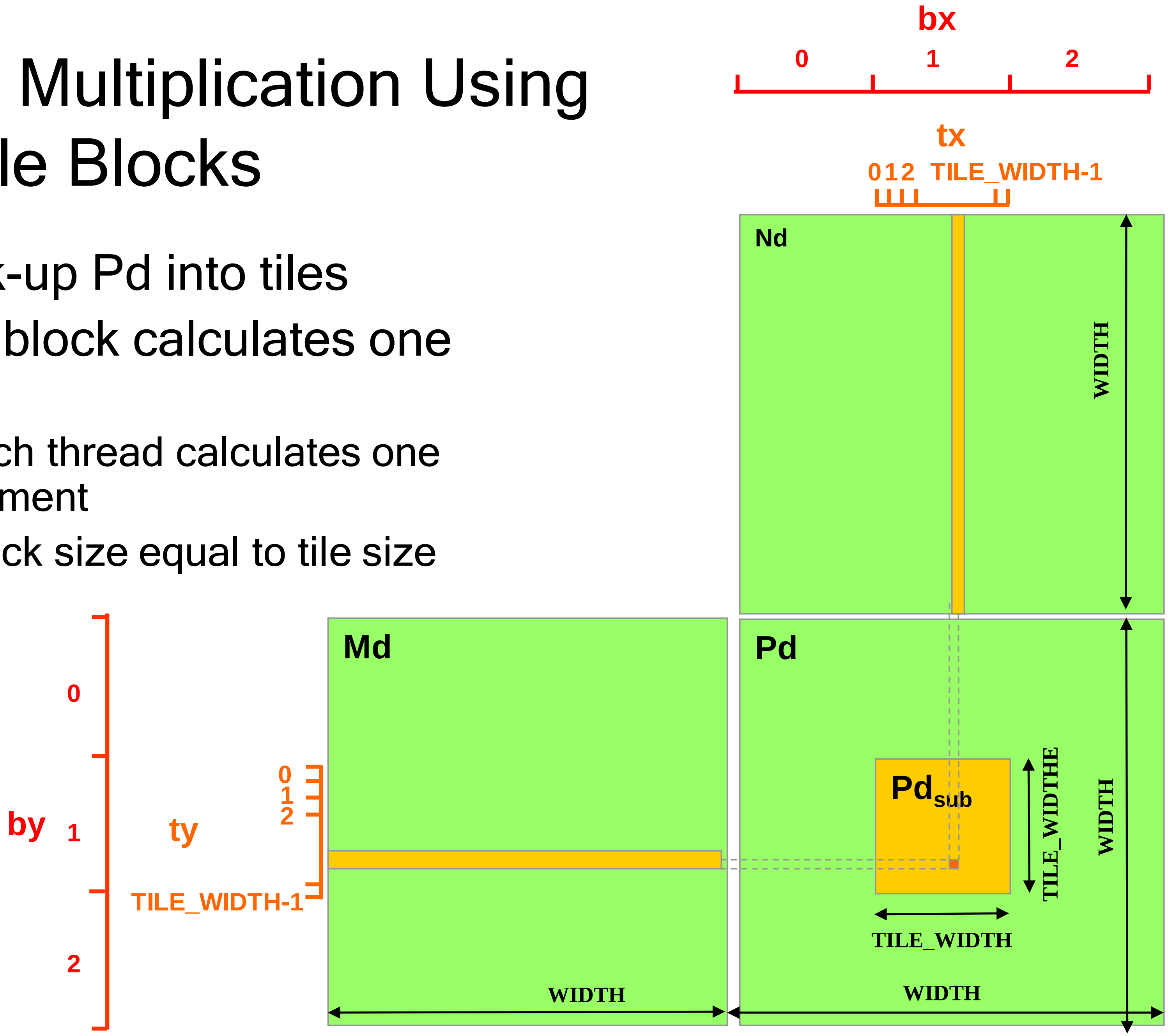
- Each thread uses IDs to decide what data to work on
  - Block ID: 1D, 2D or 3D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



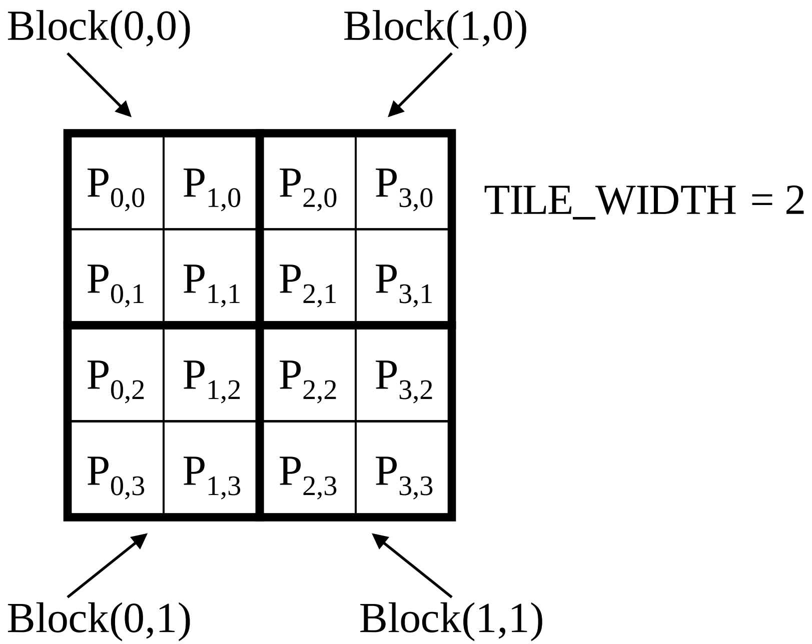
Courtesy: NDVIA

Matrix Multiplication Using Multiple Blocks

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



A Small Example



Block(0,0)

Block(1,0)

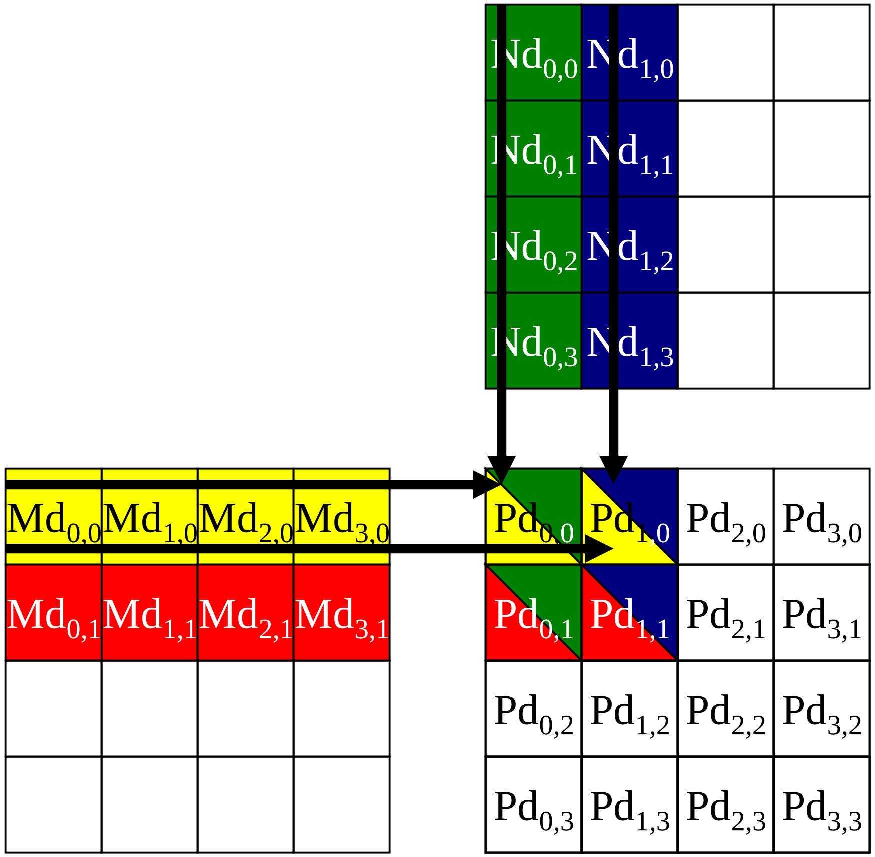
$P_{0,0}$	$P_{1,0}$	$P_{2,0}$	$P_{3,0}$
$P_{0,1}$	$P_{1,1}$	$P_{2,1}$	$P_{3,1}$
$P_{0,2}$	$P_{1,2}$	$P_{2,2}$	$P_{3,2}$
$P_{0,3}$	$P_{1,3}$	$P_{2,3}$	$P_{3,3}$

TILE_WIDTH = 2

Block(0,1)

Block(1,1)

A Small Example: Multiplication



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

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

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

Revised Matrix Multiplication Kernel using Multiple Blocks

```
__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;
}
```

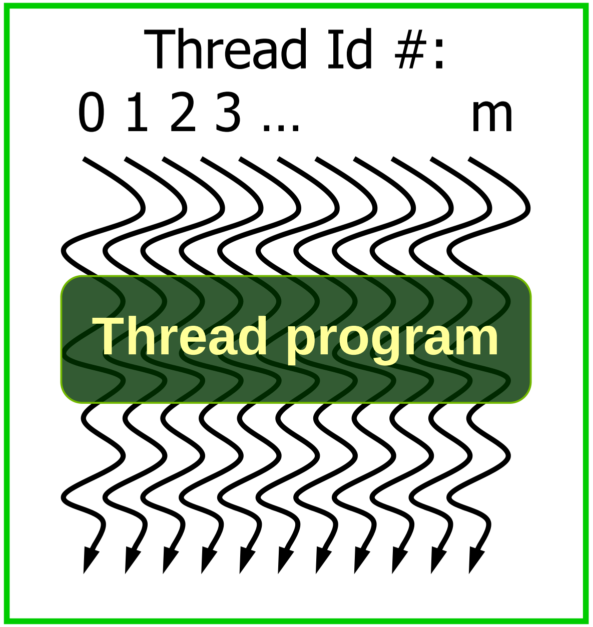
Revised Step 5: Kernel Invocation (Host-side Code)

```
// 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);
```

CUDA Thread Block

- All threads in a block execute the same kernel program (SPMD)
- Programmer declares block:
  - Block size 1 to **512** concurrent threads
  - Block shape 1D, 2D, or 3D
  - Block dimensions in threads
- Threads have thread id numbers within block
  - Thread program uses thread id to select work and address shared data
- Threads in the same block share data and synchronize while doing their share of the work
- Threads in different blocks cannot cooperate
  - Each block can execute in any order relative to other blocs!

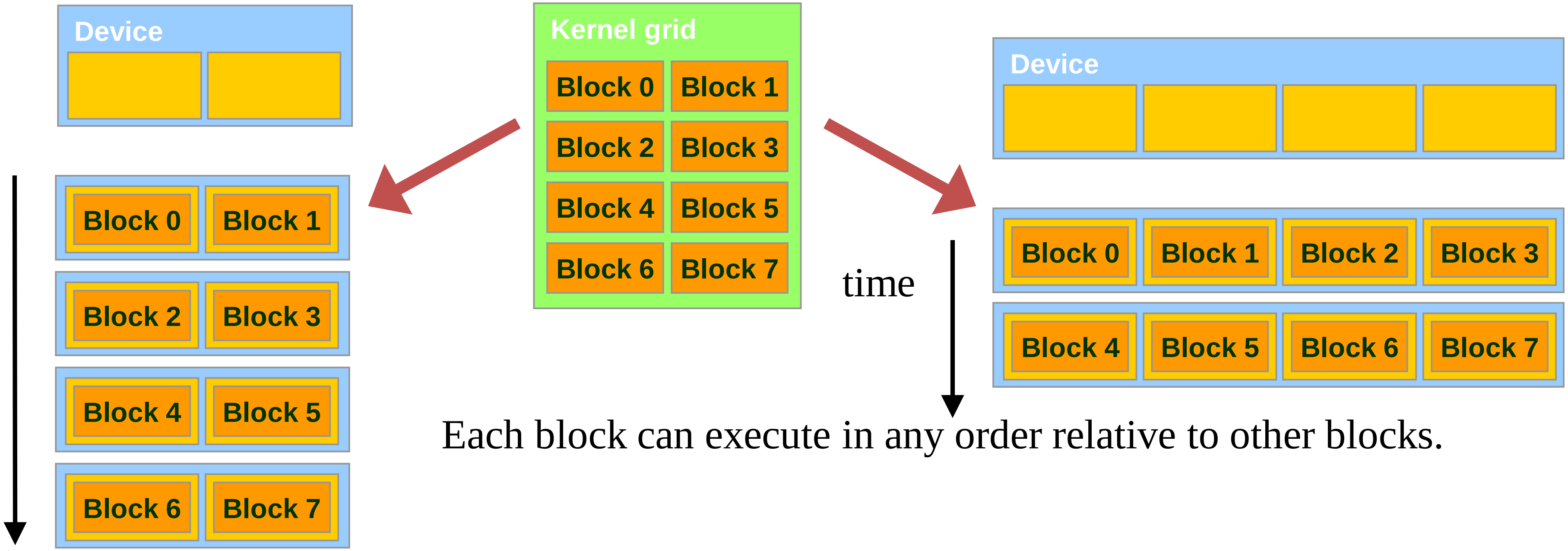
CUDA Thread Block



Courtesy: John Nickolls, NVIDIA

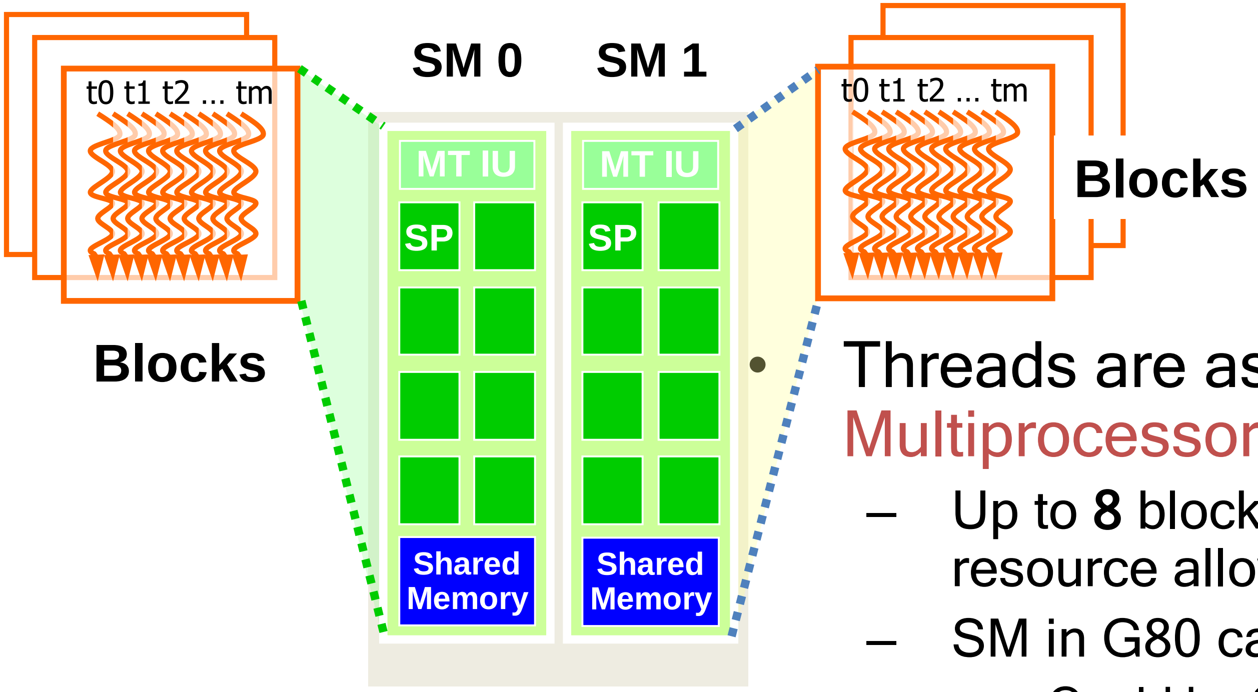
Transparent Scalability

- Hardware is free to assign blocks to any processor at any time
  - A kernel scales across any number of parallel processors



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

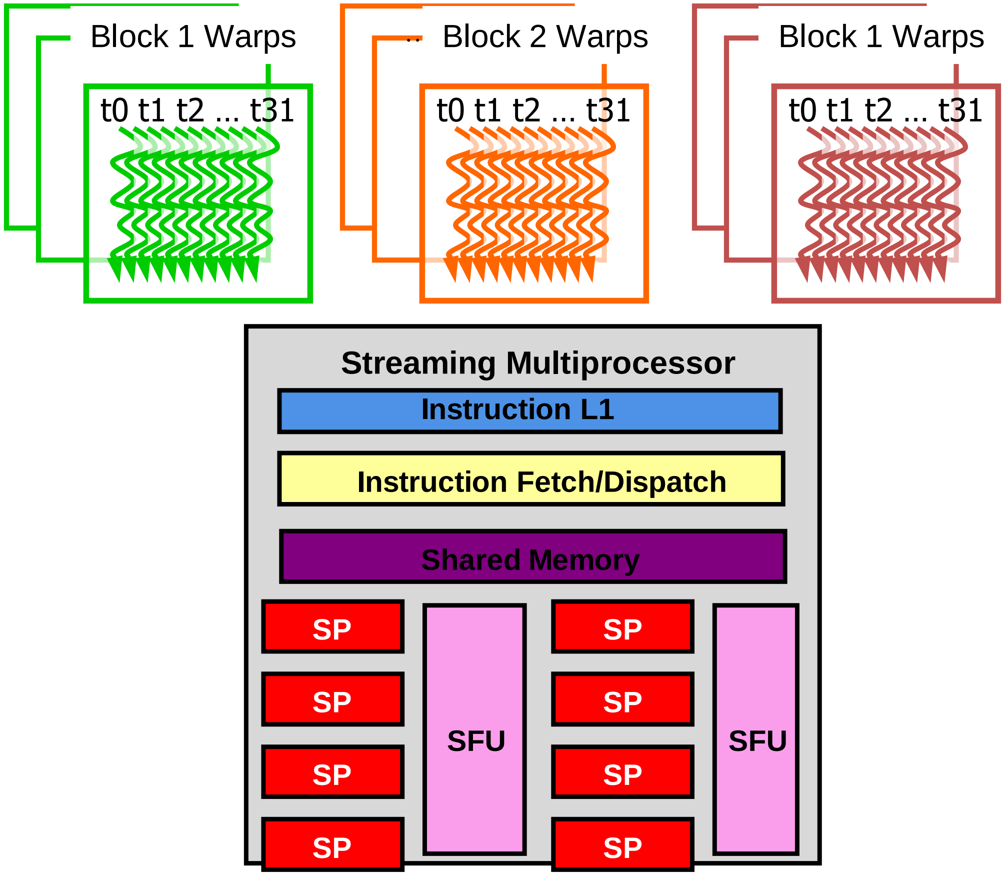
G80 Example: Executing Thread Blocks



- Threads are assigned to Streaming Multiprocessors in block granularity
  - 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.
- Threads run concurrently
  - SM maintains thread/block id #s
  - SM manages/schedules thread execution

G80 Example: Thread Scheduling

- Each Block is executed as 32-thread Warps
  - An implementation decision, not part of the CUDA programming model
  - Warps are scheduling units in SM
- 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



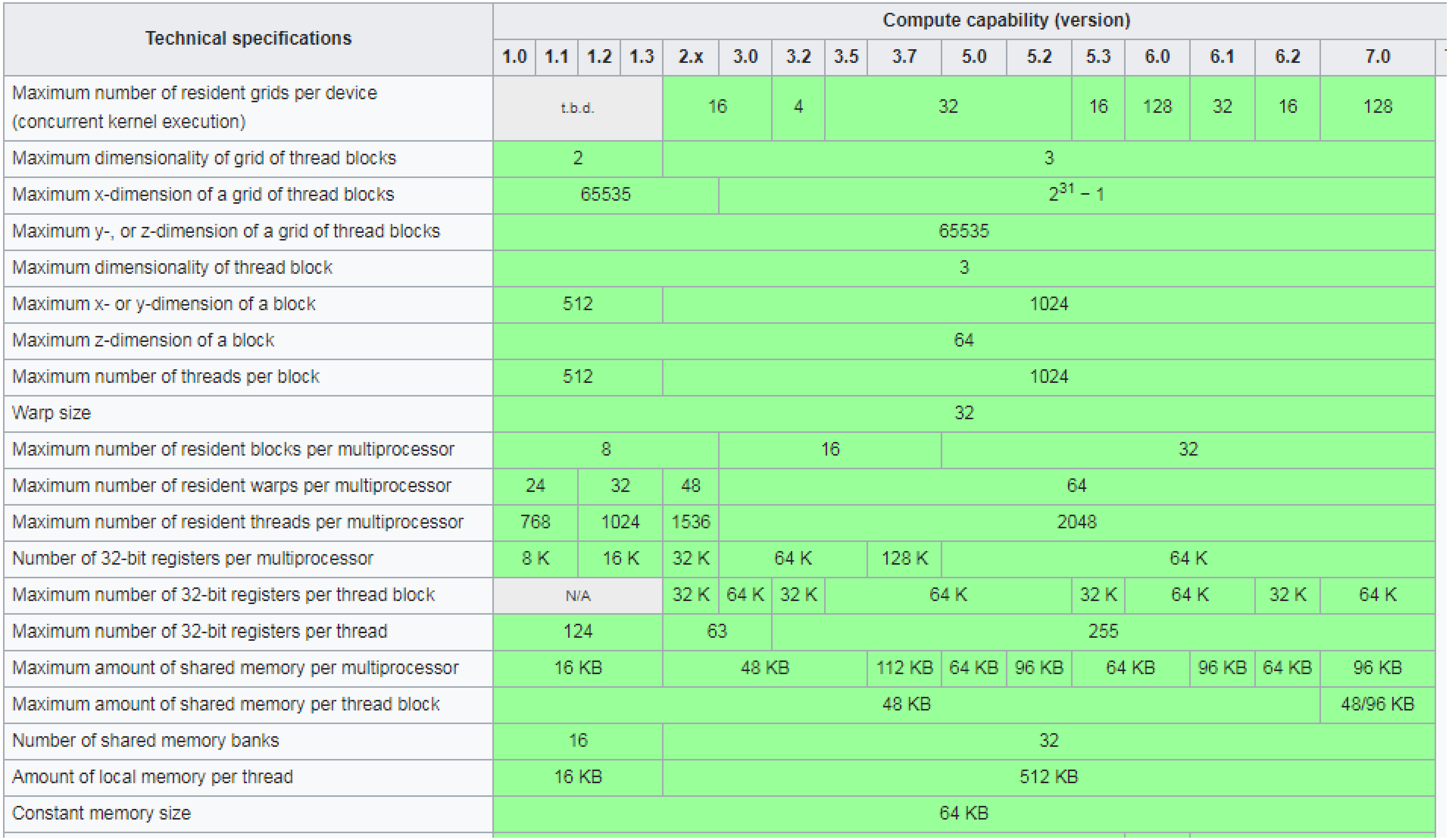
G80 Example: Thread Scheduling (Cont.)

- SM implements zero-overhead warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a warp execute the same instruction when selected

G80 Block Granularity Considerations

- For Matrix Multiplication using multiple blocks, should I use 8X8, 16X16 or 32X32 blocks?
  - 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!
  - 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.
  - For 32X32, we have 1024 threads per Block. Not even one can fit into an SM!

Technical Specifications per Compute Capability



Technical specifications	Compute capability (version)															
	1.0	1.1	1.2	1.3	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0
Maximum number of resident grids per device (concurrent kernel execution)	t.b.d.				16		4	32				16	128	32	16	128
Maximum dimensionality of grid of thread blocks	2				3											
Maximum x-dimension of a grid of thread blocks	65535					$2^{31} - 1$										
Maximum y-, or z-dimension of a grid of thread blocks	65535															
Maximum dimensionality of thread block	3															
Maximum x- or y-dimension of a block	512				1024											
Maximum z-dimension of a block	64															
Maximum number of threads per block	512				1024											
Warp size	32															
Maximum number of resident blocks per multiprocessor	8					16				32						
Maximum number of resident warps per multiprocessor	24		32		48	64										
Maximum number of resident threads per multiprocessor	768		1024		1536	2048										
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K	64 K			128 K	64 K						
Maximum number of 32-bit registers per thread block	N/A				32 K	64 K	32 K	64 K				32 K	64 K		32 K	64 K
Maximum number of 32-bit registers per thread	124				63		255									
Maximum amount of shared memory per multiprocessor	16 KB				48 KB				112 KB	64 KB	96 KB	64 KB		96 KB	64 KB	96 KB
Maximum amount of shared memory per thread block	48 KB															48/96 KB
Number of shared memory banks	16				32											
Amount of local memory per thread	16 KB				512 KB											
Constant memory size	64 KB															

Source: Wikipedia

More Details of API Features

Application Programming Interface

- The API is an extension to the C programming language
- It consists of:
  - Language extensions
    - To target portions of the code for execution on the device
  - A runtime library split into:
    - A common component providing built-in vector types and a subset of the C runtime library in both host and device code
    - A host component to control and access one or more devices from the host
    - A device component providing device-specific functions

Language Extensions: Built-in Variables

- **`dim3 gridDim;`**
  - Dimensions of the grid in blocks
- **`dim3 blockDim;`**
  - Dimensions of the block in threads
- **`dim3 blockIdx;`**
  - Block index within the grid
- **`dim3 threadIdx;`**
  - Thread index within the block

Common Runtime Component: Mathematical Functions

- `pow, sqrt, cbrt, hypot`
- `exp, exp2, expm1`
- `log, log2, log10, log1p`
- `sin, cos, tan, asin, acos, atan, atan2`
- `sinh, cosh, tanh, asinh, acosh, atanh`
- `ceil, floor, trunc, round`
- Etc.
  - When executed on the host, a given function uses the C runtime implementation if available
  - These functions are only supported for scalar types, not vector types

Device Runtime Component: Mathematical Functions

- Some mathematical functions (e.g. `sin(x)`) have a less accurate, but faster device-only version (e.g. `__sin(x)`)
  - `__pow`
  - `__log`, `__log2`, `__log10`
  - `__exp`
  - `__sin`, `__cos`, `__tan`

Host Runtime Component

- Provides functions to deal with:
  - Device management (including multi-device systems)
  - Memory management
  - Error handling

- Initializes the first time a runtime function is called

- A host thread can invoke device code on only one device
  - Multiple host threads required to run on multiple devices

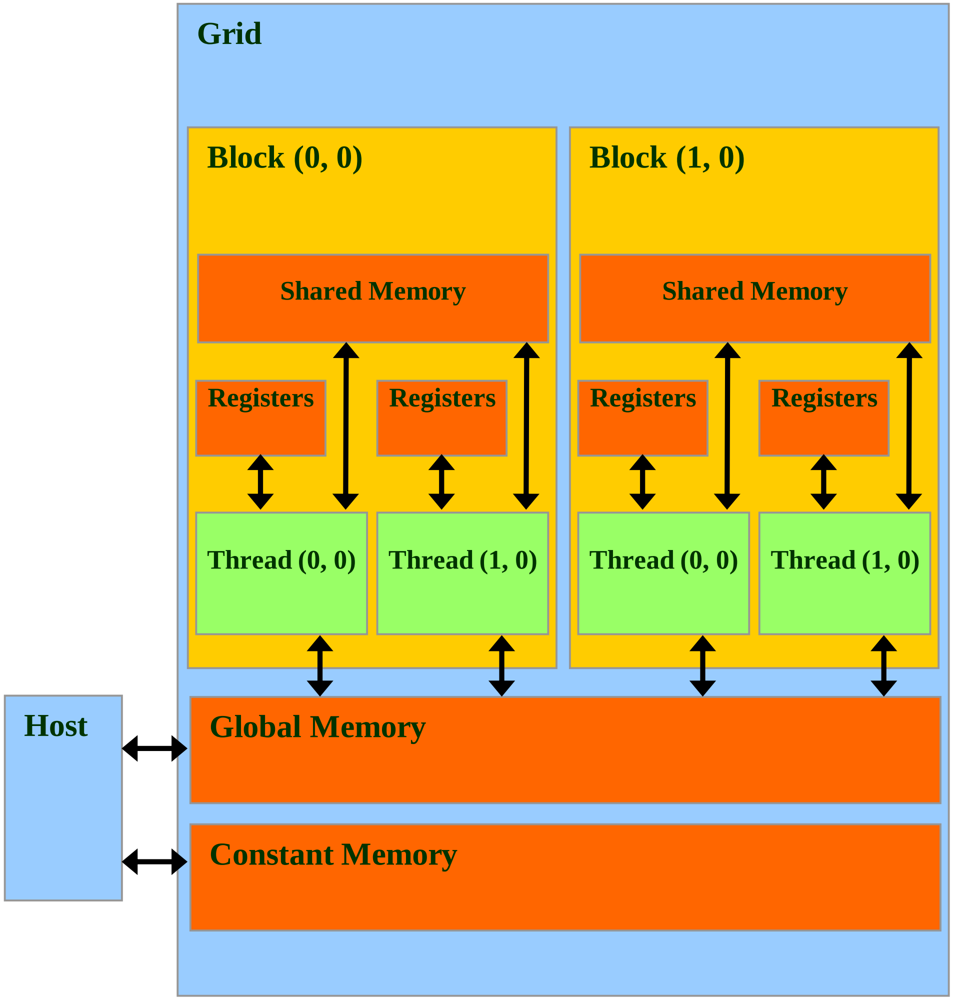
Device Runtime Component: Synchronization Function

- `void __syncthreads();`
- Synchronizes all threads in a block
- Once all threads have reached this point, execution resumes normally
- Used to avoid RAW / WAR / WAW hazards when accessing shared or global memory
- Allowed in conditional constructs only if the conditional is uniform across the entire thread block

CUDA Memories

Hardware Implementation of CUDA Memories

- Each thread can:
  - Read/write per-thread registers
  - Read/write per-thread local memory
  - Read/write per-block shared memory
  - Read/write per-grid global memory
  - Read/only per-grid constant memory



CUDA Variable Type Qualifiers

Variable declaration	Memory	Scope	Lifetime
`int var;`	register	thread	thread
`int array_var[10];`	local	thread	thread
`__shared__ int shared_var;`	shared	block	block
`__device__ int global_var;`	global	grid	application
`__constant__ int constant_var;`	constant	grid	application

- "automatic" scalar variables without qualifier reside in a register
  - compiler will spill to thread local memory
- "automatic" array variables without qualifier reside in thread local memory

CUDA Variable Type Performance

Variable declaration	Memory	Penalty
`int var;`	register	1x
`int array_var[10];`	local	100x
`__shared__ int shared_var;`	shared	1x
`__device__ int global_var;`	global	100x
`__constant__ int constant_var;`	constant	1x

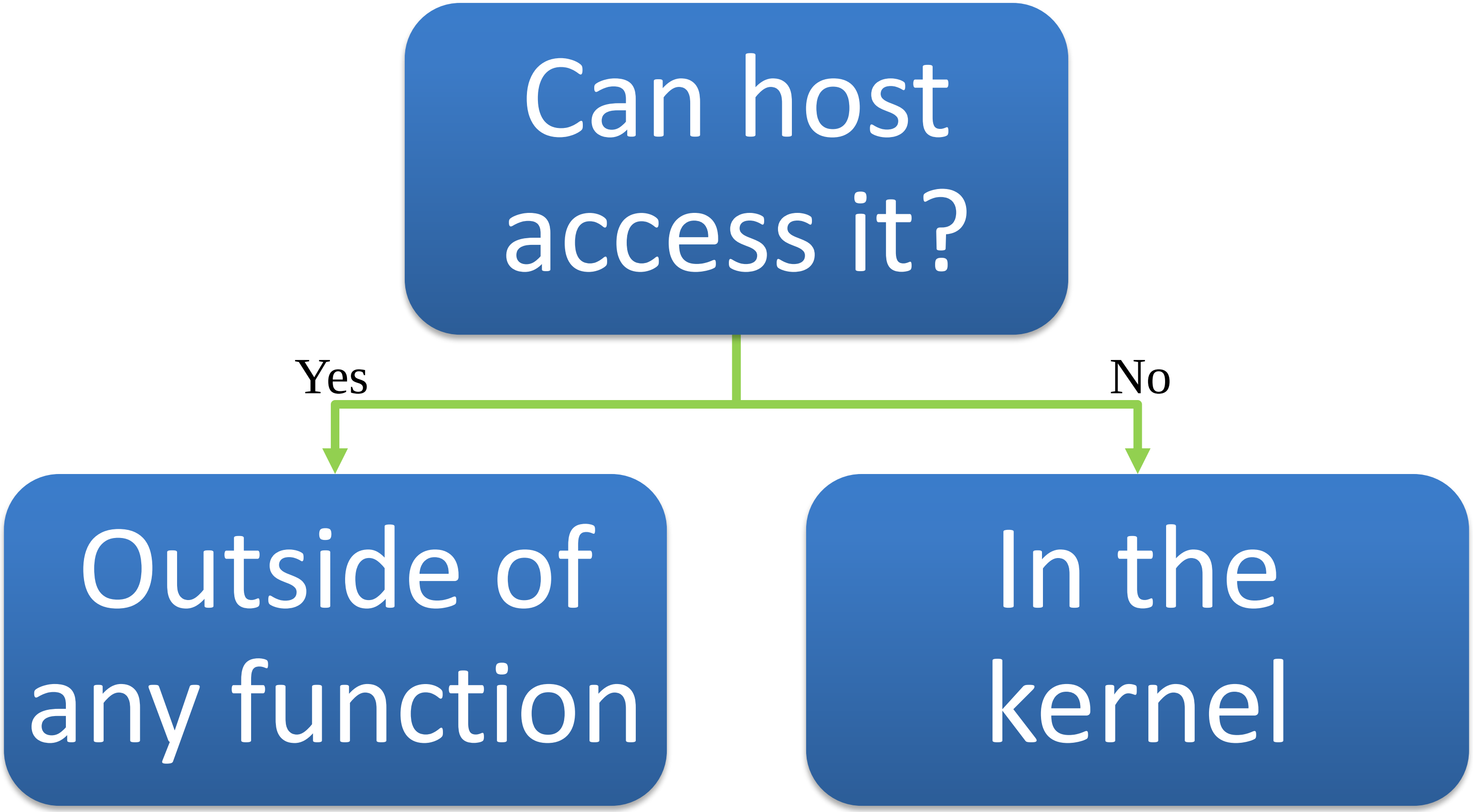
- scalar variables reside in fast, on-chip registers
- shared variables reside in fast, on-chip memories
- thread-local arrays & global variables reside in uncached off-chip memory
  - Cache is now available, but there is still a significant drop off in speed
- constant variables reside in cached off-chip memory

CUDA Variable Type Scale

Variable declaration	Instances	Visibility
`int var;`	100,000s	1
`int array_var[10];`	100,000s	1
`__shared__ int shared_var;`	100s	100s
`__device__ int global_var;`	1	100,000s
`__constant__ int constant_var;`	1	100,000s

- 100Ks per-thread variables, R/W by 1 thread
- 100s shared variables, each R/W by 100s of threads
- 1 global variable is R/W by 100Ks threads
- 1 constant variable is readable by 100Ks threads

Where to declare variables?



```
__constant__ int constant_var;
__device__   int global_var;
```

```
int var;
int array_var[10];
__shared__   int shared_var;
```

Example - thread-local variables

```
// Ten Nearest Neighbors application
__global__ void ten_nn(float2 *result, float2 *ps, float2 *qs,
                       size_t num_qs)
{
  // p goes in a register
  float2 p = ps[threadIdx.x];

  // per-thread heap goes in off-chip memory
  float2 heap[10];

  // read through num_qs points, maintaining
  // the nearest 10 qs to p in the heap
  ...
  // write out the contents of heap to result
  ...
}
```

Example - shared variables

```
// motivate shared variables with
// Adjacent Difference application:
// compute result[i] = input[i] – input[i-1]
__global__ void adj_diff_naive(int *result, int *input)
{
  // compute this thread’s global index
  unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

  if(i > 0)
  {

    int x_i = input[i];
    int x_i_minus_one = input[i-1];

    result[i] = x_i – x_i_minus_one;
  }
}
```

Example - shared variables

```
// motivate shared variables with
// Adjacent Difference application:
// compute result[i] = input[i] – input[i-1]
__global__ void adj_diff_naive(int *result, int *input)
{
  // compute this thread’s global index
  unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

  if(i > 0)
  {
    // what are the bandwidth requirements of this kernel?
    int x_i = input[i];
    int x_i_minus_one = input[i-1];

    result[i] = x_i – x_i_minus_one;
  }
}
```

Two loads

Example - shared variables

```
// motivate shared variables with
// Adjacent Difference application:
// compute result[i] = input[i] – input[i-1]
__global__ void adj_diff_naive(int *result, int *input)
{
  // compute this thread’s global index
  unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

  if(i > 0)
  {
    // How many times does this kernel load input[i]?
    int x_i = input[i]; // once by thread i
    int x_i_minus_one = input[i-1];   // again by thread i+1

    result[i] = x_i – x_i_minus_one;
  }
}
```

Example - shared variables

```
// optimized version of adjacent difference
__global__ void adj_diff(int *result, int *input)
{
  // shorthand for threadIdx.x
  int tx = threadIdx.x;
  // allocate a __shared__ array, one element per thread
  __shared__ int s_data[BLOCK_SIZE];
  // each thread reads one element to s_data
  unsigned int i = blockDim.x * blockIdx.x + tx;
  s_data[tx] = input[i];

  // avoid race condition: ensure all loads
  // complete before continuing
  __syncthreads();
  ...
```

Example - shared variables

```
 if(tx > 0)
  result[i] = s_data[tx] - s_data[tx-1];
 else if(i > 0)
 {
  // handle thread block boundary
  result[i] = s_data[tx] - input[i-1];
 }
}
```

Example - shared variables

```
// when the size of the array isn't known at compile time...
__global__ void adj_diff(int *result, int *input)
{
  // use extern to indicate a __shared__ array will be
  // allocated dynamically at kernel launch time
  extern __shared__ int s_data[];
  ...
}

// pass the size of the per-block array, in bytes, as the third
// argument to the triple chevrons
adj_diff<<<num_blocks, block_size, block size * sizeof(int)>>>(r,i);
```

- Only one extern shared array can be declared
  - See CUDA programming guide for work-around

About Pointers - Outdated but Useful

- Yes, you can use them!
- You can point to any memory space:

```
__device__ int my_global_variable;
__constant__ int my_constant_variable = 13;

__global__ void foo(void)
{
  __shared__ int my_shared_variable;

  int *ptr_to_global = &my_global_variable;
  const int *ptr_to_constant = &my_constant_variable;
  int *ptr_to_shared = &my_shared_variable;
  ...
  *ptr_to_global = *ptr_to_shared;
}
```

About Pointers – Outdated but Useful

- Pointers aren't typed on memory space
  - `__shared__ int *ptr;`
  - Where does `ptr` point?
  - `ptr` is a `__shared__` pointer variable, not a pointer to a `__shared__` variable!

Don’t confuse the compiler!

```
__device__ int my_global_variable;
__global__ void foo(int *input)
{
  __shared__ int my_shared_variable;

  int *ptr = 0;
  if(input[threadIdx.x] % 2)
    ptr = &my_global_variable;
  else
    ptr = &my_shared_variable;
  // where does ptr point?
}
```

Advice

- Prefer dereferencing pointers in simple, regular access patterns
- Avoid propagating pointers
- Avoid pointers to pointers
  - The GPU would rather not pointer chase
  - Linked lists will not perform well
- Pay attention to compiler warning messages
  - `Warning: Cannot tell what pointer points to, assuming global memory space`
  - Crash waiting to happen

Unified Virtual Address Space

- The location of any memory on the host or on any of the devices which use the unified address space, can be determined from the value of the pointer using `cudaPointerGetAttributes()`
- When copying, the `cudaMemcpyKind` parameter of `cudaMemcpy*()` can be set to `cudaMemcpyDefault` to determine locations from the pointers. This also works for host pointers not allocated through CUDA, as long as the current device uses unified addressing.

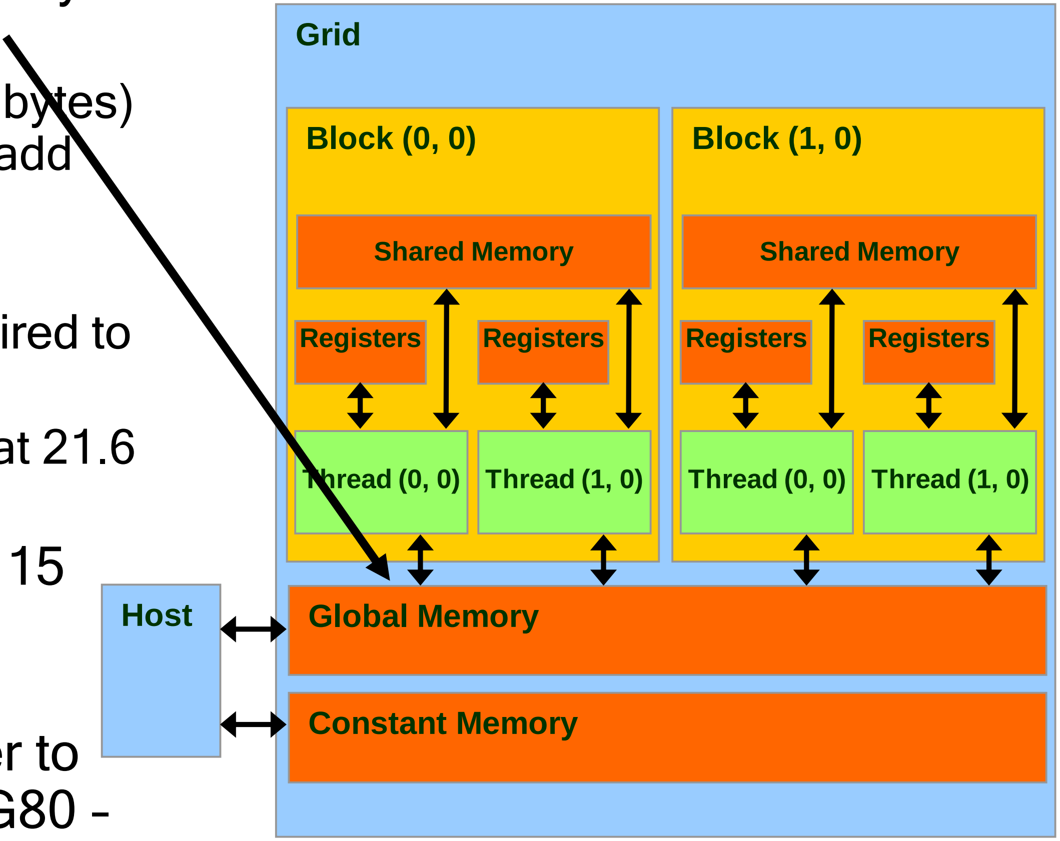
Matrix Multiplication using Shared Memory

Review: Matrix Multiplication Kernel using Multiple Blocks

```
__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;
}
```

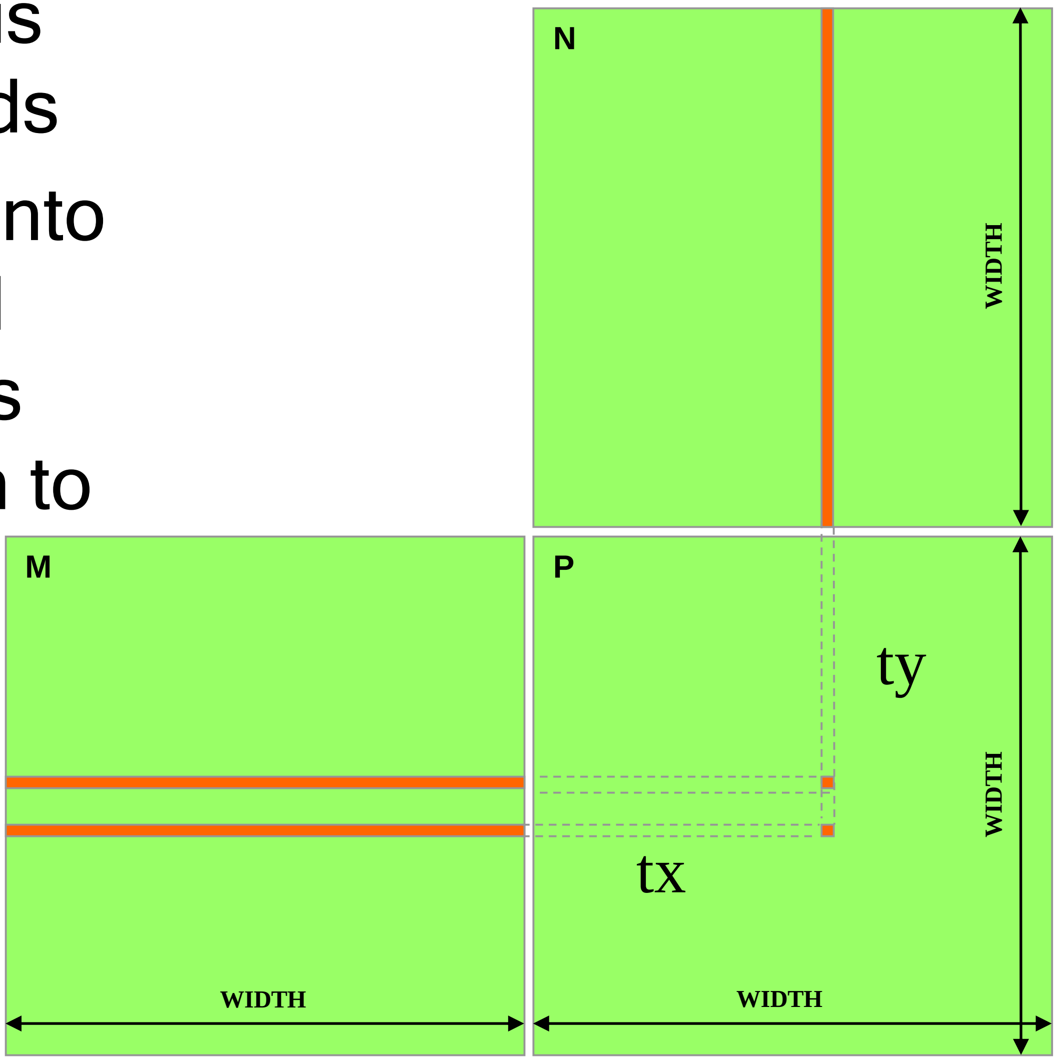
How about performance on GPU?

- All threads access global memory for their input matrix elements
  - Two memory accesses (8 bytes) per floating point multiply-add
  - 4B/s of memory bandwidth/FLOPS
  - 4*346.5 = 1386 GB/s required to achieve peak FLOP rating
  - 86.4 GB/s limits the code at 21.6 GFLOPS
- The actual code runs at about 15 GFLOPS
- Need to drastically cut down memory accesses to get closer to the peak 346.5 GFLOPS (on G80 - ignore specific numbers)



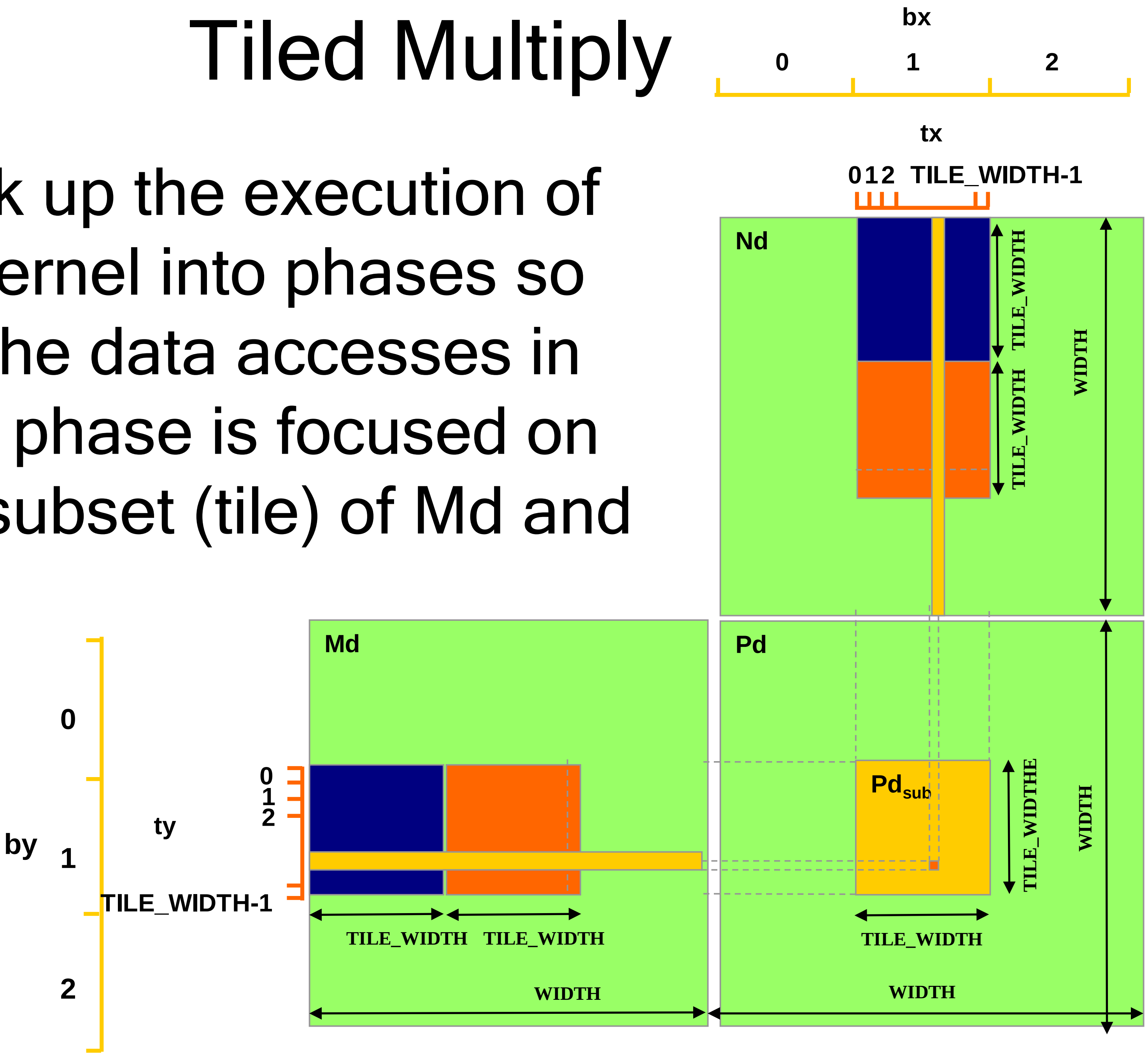
Idea: Use Shared Memory to reuse global memory data

- Each input element is read by Width threads
- Load each element into Shared Memory and have several threads use the local version to reduce the memory bandwidth
  - – Tiled algorithms

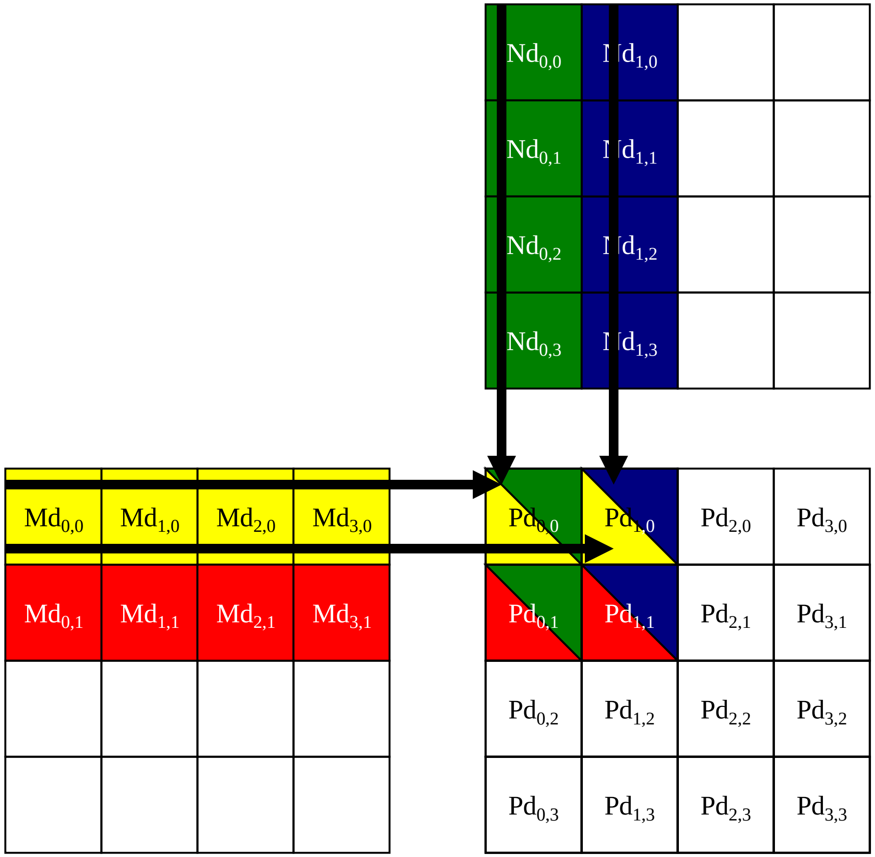


Tiled Multiply

- Break up the execution of the kernel into phases so that the data accesses in each phase is focused on one subset (tile) of Md and Nd



A Small Example



$Nd_{0,0}$ $Nd_{1,0}$

$Nd_{0,1}$ $Nd_{1,1}$

$Nd_{0,2}$ $Nd_{1,2}$

$Nd_{0,3}$ $Nd_{1,3}$

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

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

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

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

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

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

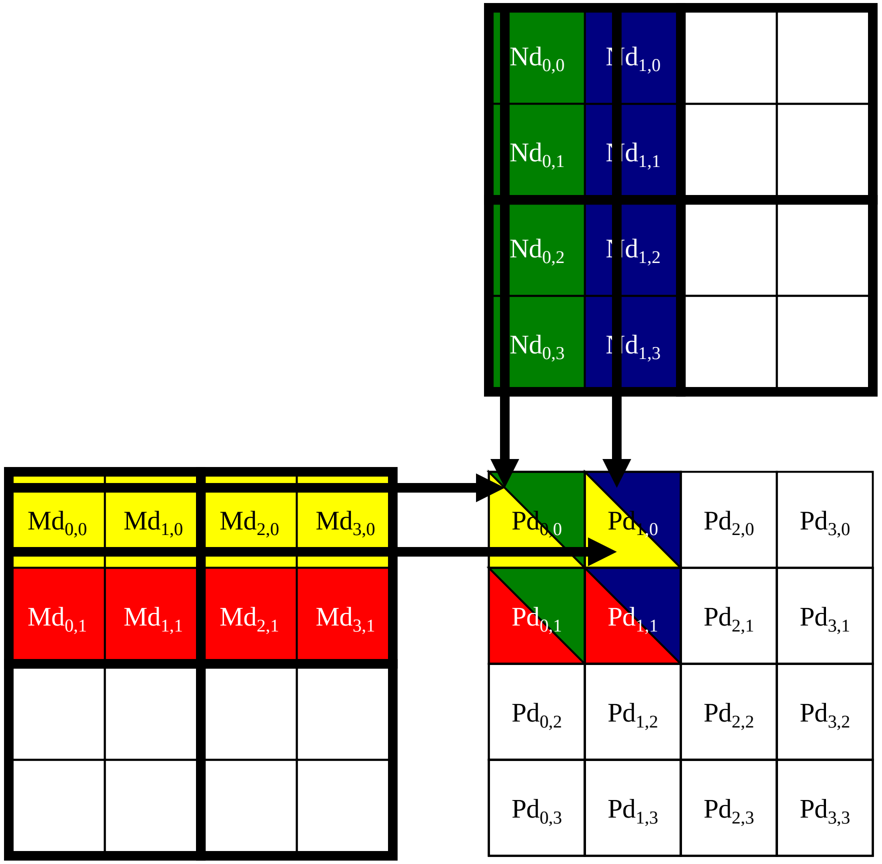
Every Md and Nd Element is used exactly twice in generating a 2X2 tile of P

Access order

$P_{0,0}$ $\text{thread}_{0,0}$	$P_{1,0}$ $\text{thread}_{1,0}$	$P_{0,1}$ $\text{thread}_{0,1}$	$P_{1,1}$ $\text{thread}_{1,1}$
$M_{0,0} * N_{0,0}$	$M_{0,0} * N_{1,0}$	$M_{0,1} * N_{0,0}$	$M_{0,1} * N_{1,0}$
$M_{1,0} * N_{0,1}$	$M_{1,0} * N_{1,1}$	$M_{1,1} * N_{0,1}$	$M_{1,1} * N_{1,1}$
$M_{2,0} * N_{0,2}$	$M_{2,0} * N_{1,2}$	$M_{2,1} * N_{0,2}$	$M_{2,1} * N_{1,2}$
$M_{3,0} * N_{0,3}$	$M_{3,0} * N_{1,3}$	$M_{3,1} * N_{0,3}$	$M_{3,1} * N_{1,3}$

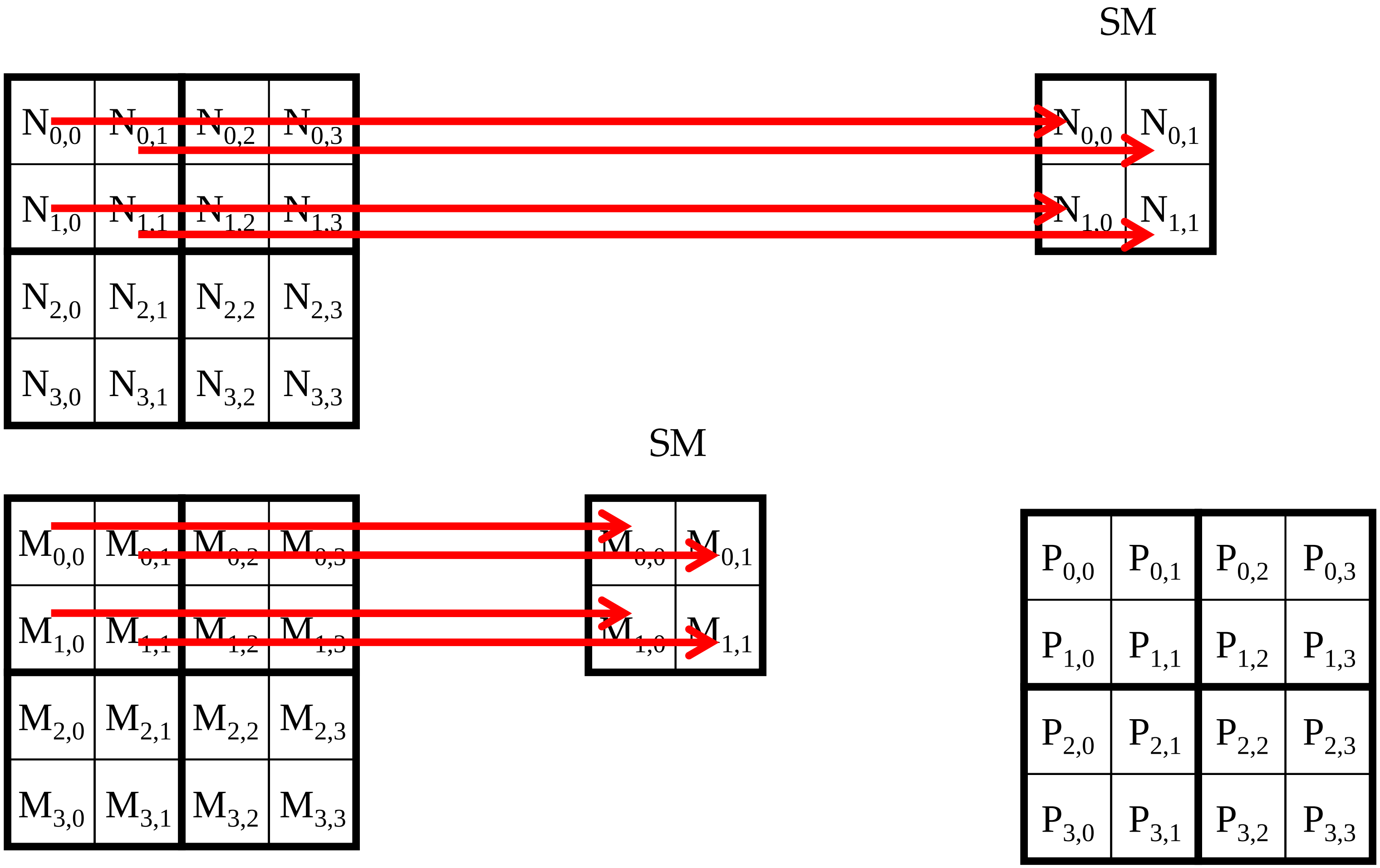
Breaking Md and Nd into Tiles

- Break up the inner product loop of each thread into phases
- At the beginning of each phase, load the Md and Nd elements that everyone needs during the phase into shared memory
- Everyone accesses the Md and Nd elements from shared memory during the phase



© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2010
ECE408, University of Illinois, Urbana Champaign

Work for Block (0,0)



SM

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

$N_{0,0}$	$N_{0,1}$
$N_{1,0}$	$N_{1,1}$

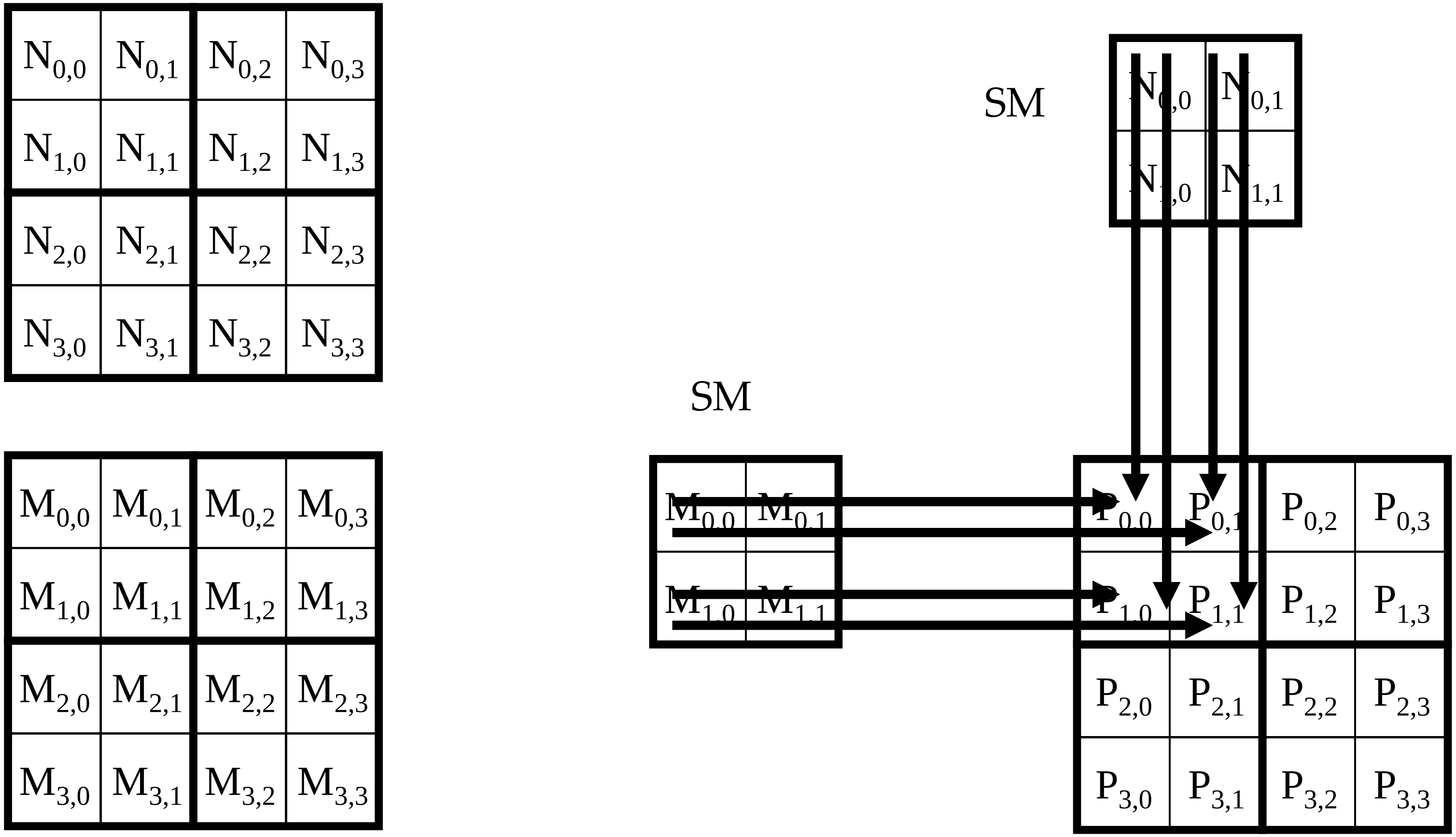
SM

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

$M_{0,0}$	$M_{0,1}$
$M_{1,0}$	$M_{1,1}$

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

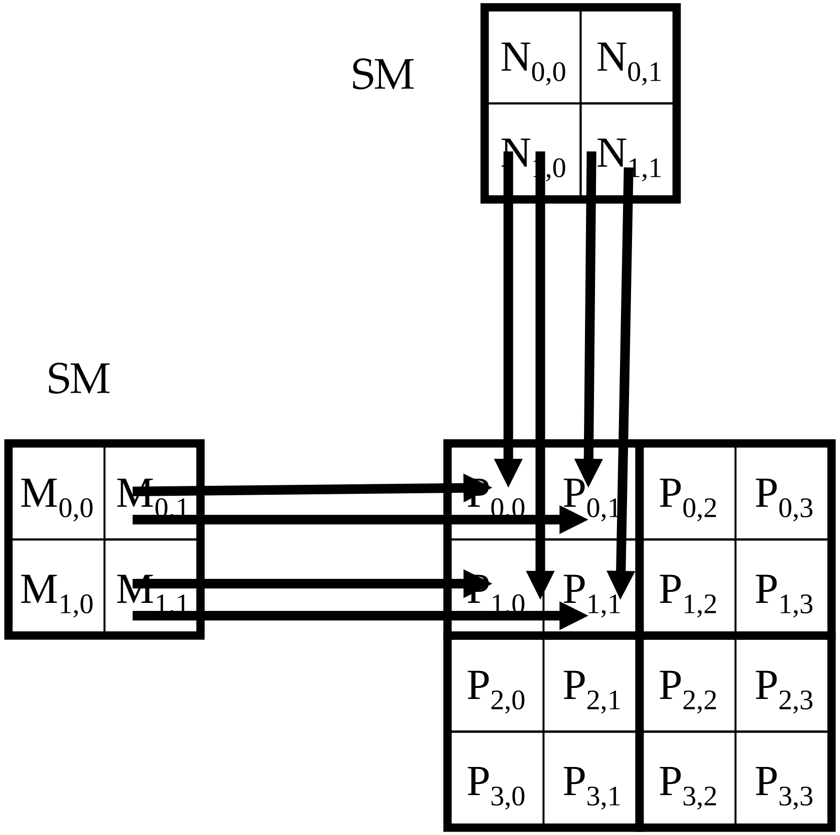
Work for Block (0,0)


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

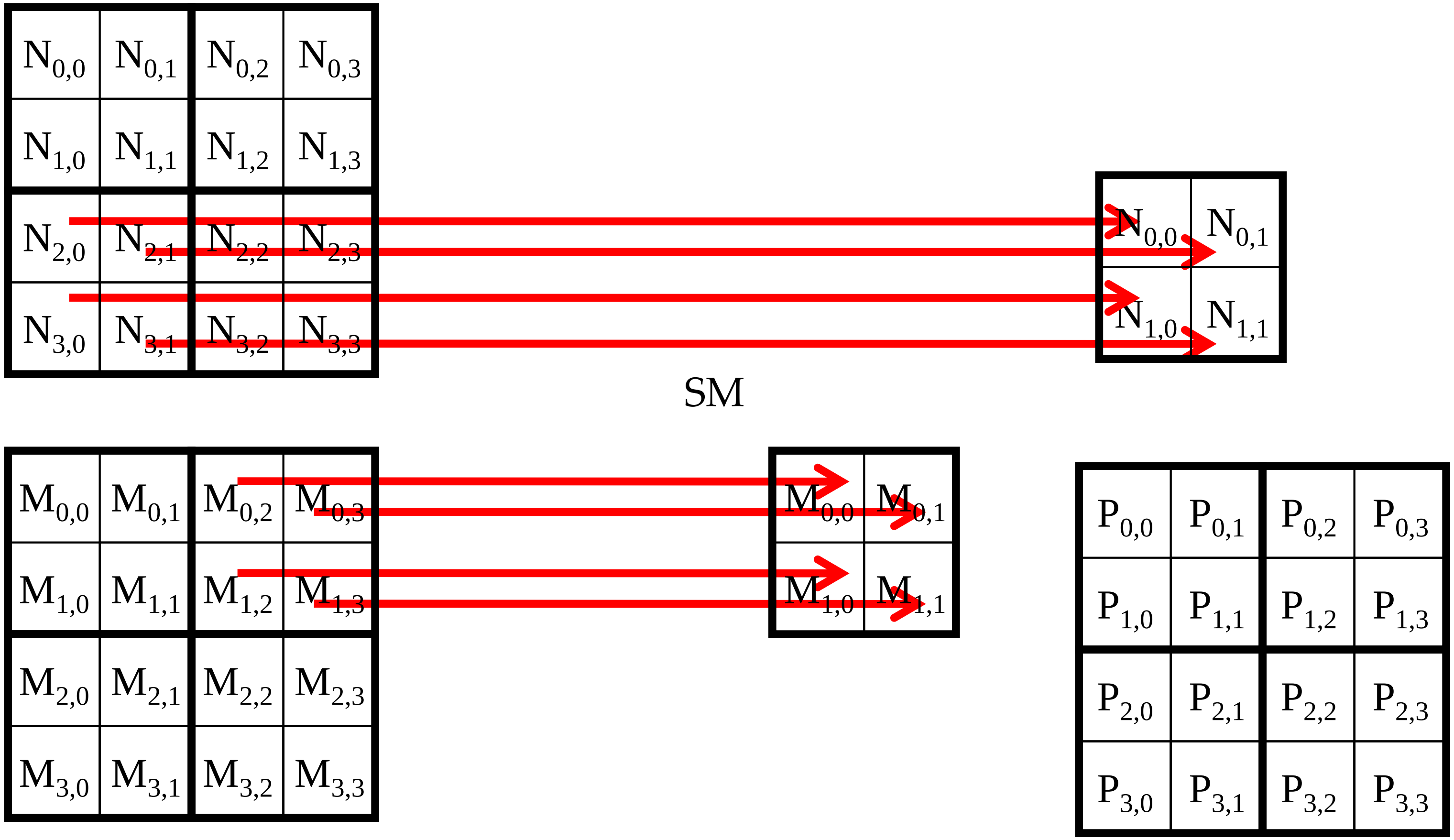
Work for Block (0,0)

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

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



Work for Block (0,0)



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

$N_{0,0}$	$N_{0,1}$
$N_{1,0}$	$N_{1,1}$

SM

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

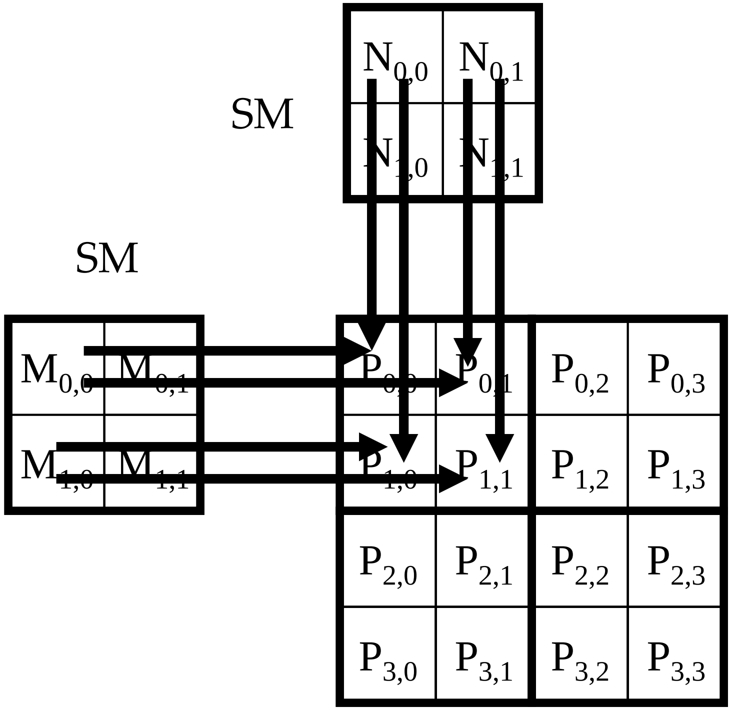
$M_{0,0}$	$M_{0,1}$
$M_{1,0}$	$M_{1,1}$

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

Work for Block (0,0)

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

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


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

Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
   __shared __float Mds[TILE_WIDTH][TILE_WIDTH];
   __shared __float Nds[TILE_WIDTH][TILE_WIDTH];

   int bx = blockIdx.x;  int by = blockIdx.y;
   int tx = threadIdx.x; int ty = threadIdx.y;

// Identify the row and column of the Pd element to work on
   int Row = by * TILE_WIDTH + ty;
   int Col = bx * TILE_WIDTH + tx;

   float Pvalue = 0;
// Loop over the Md and Nd tiles required to compute the Pd element
   for (int m = 0; m < Width/TILE_WIDTH; ++m) {

// Collaborative loading of Md and Nd tiles into shared memory
       Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
       Nds[ty][tx] = Nd[(m*TILE_WIDTH + ty)*Width + Col];
       __syncthreads();

      for (int k = 0; k < TILE_WIDTH; ++k)
         Pvalue += Mds[ty][k] * Nds[k][tx];
      __syncthreads();
   }
   Pd[Row*Width + Col] = Pvalue;
}
```

CUDA Code – Kernel Execution Configuration

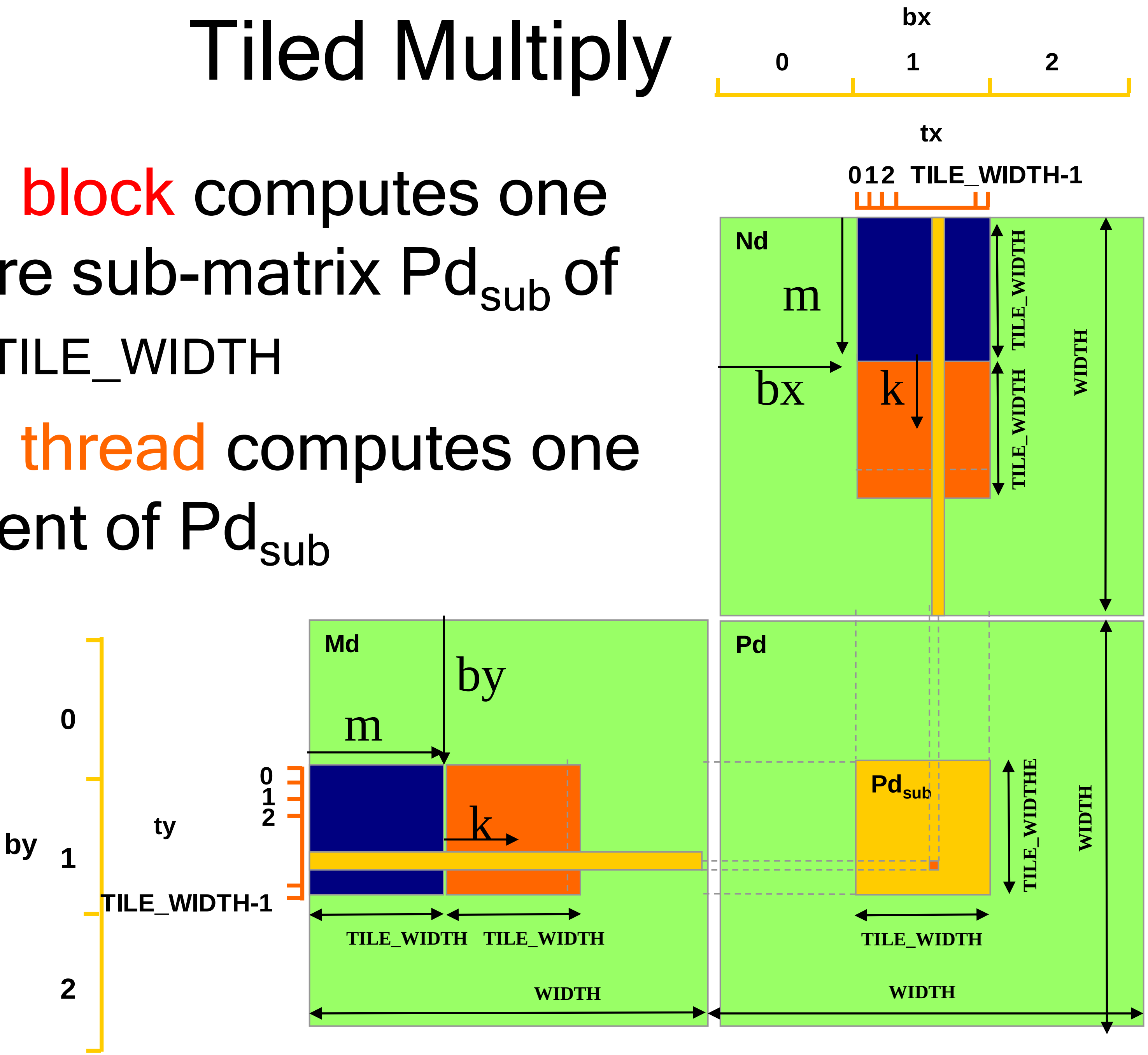
```
// Setup the execution configuration
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);
dim3 dimGrid(Width  / TILE_WIDTH,
             Width /  TILE_WIDTH);
```

First-order Size Considerations

- Each thread block should have many threads
  - TILE_WIDTH of 16 gives 16*16 = 256 threads
- There should be many thread blocks
  - A 1024*1024 Pd gives 64*64 = 4096 Thread Blocks
  - TILE_WIDTH of 16 gives each SM 3 blocks, 768 threads (full capacity)
- Each thread block performs 2*256 = 512 float loads from global memory for 256 * (2*16) = 8,192 mul/add operations (lines 9-14)
  - Memory bandwidth no longer a limiting factor

Tiled Multiply

- Each block computes one square sub-matrix Pd_{sub} of size TILE_WIDTH
- Each thread computes one element of Pd_{sub}



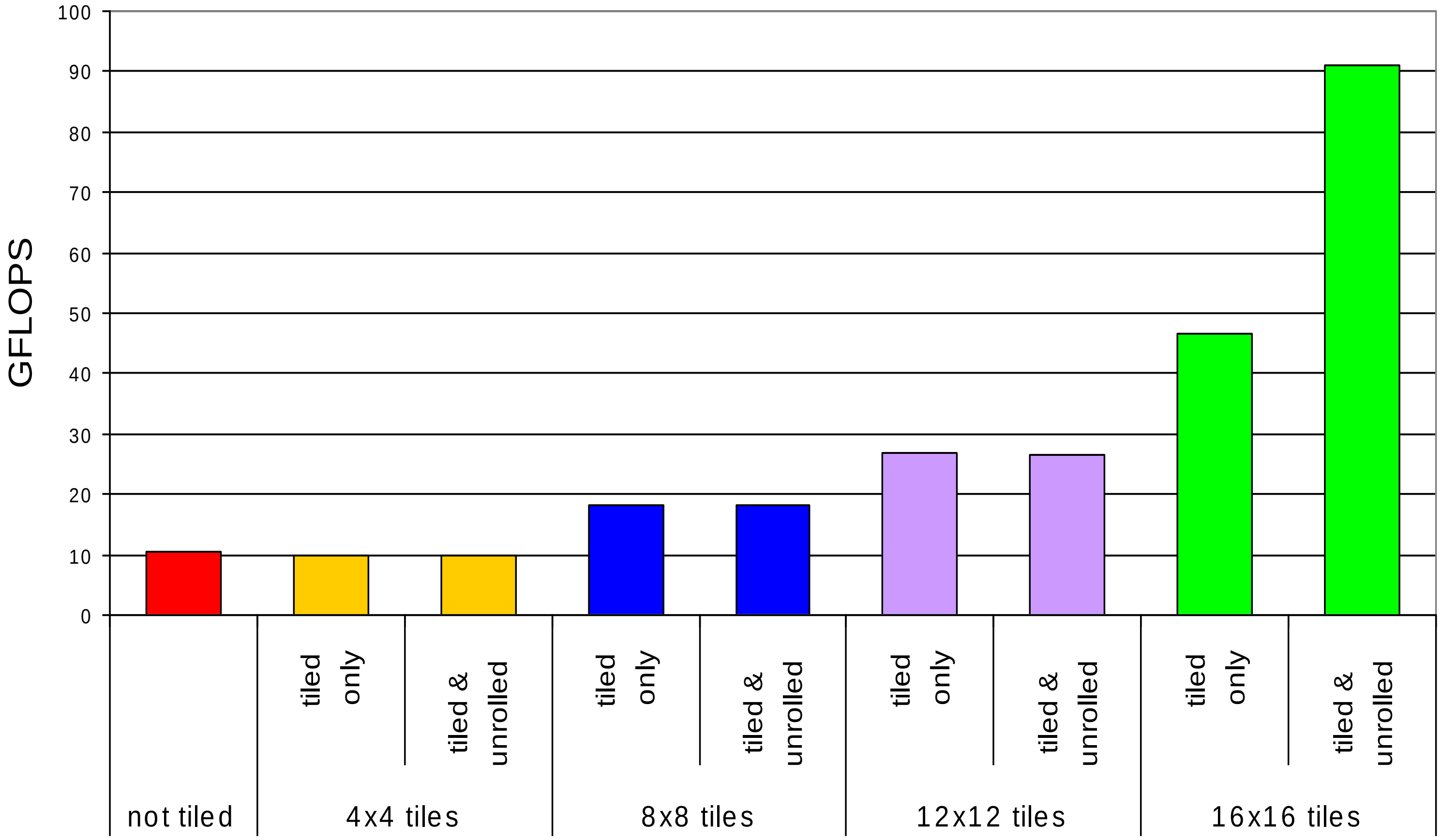
Shared Memory and Threading

- Each SM in G80 has 16KB shared memory
  - SM size is implementation-dependent!
  - For TILE_WIDTH = 16, each thread block uses 2*256*4B = 2KB of shared memory.
  - The SM can potentially have up to 8 Thread Blocks actively executing
    - This allows up to 8*512 = 4,096 pending loads. (2 per thread, 256 threads per block)
    - The threading model limits the number of thread blocks to 3 so shared memory is not the limiting factor here
  - The next TILE_WIDTH 32 would lead to 2*32*32*4B= 8KB shared memory usage per thread block, allowing only up to two thread blocks active at the same time

- Using 16x16 tiling, we reduce the accesses to the global memory by a factor of 16
  - The 86.4B/s bandwidth can now support (86.4/4)*16 = 347.6 GFLOPS

- Each SM in Fermi has 16KB or 48KB shared memory
  - Configurable vs L1 cache, total 64KB

Tiling Size Effects



Memory Resources as Limit to Parallelism

Resource	Per GT200 SM	Full Occupancy on GT200
Registers	16384	<= 16384 / 768 threads = 21 per thread
`__shared__` Memory	16KB	<= 16KB / 8 blocks = 2KB per block

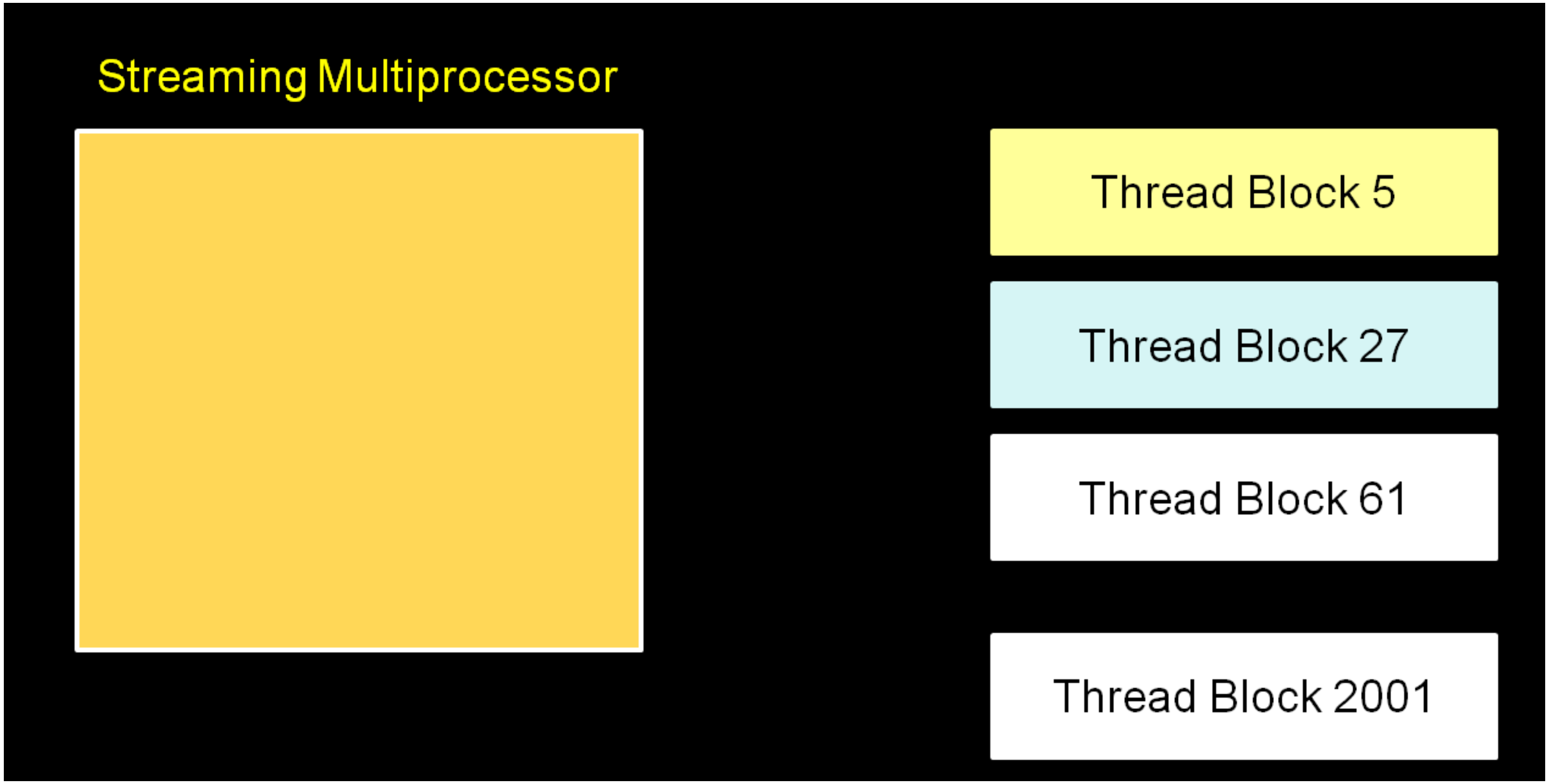
- Effective use of different memory resources reduces the number of accesses to global memory
- These resources are finite!
- The more memory locations each thread requires → the fewer threads an SM can accommodate → what if each thread required 22 registers and each block had 256 threads?

Final Thoughts on Memory

- Effective use of CUDA memory hierarchy decreases bandwidth consumption to increase throughput
- Use `__shared__` memory to eliminate redundant loads from global memory
  - Use `__syncthreads` barriers to protect `__shared__` data
  - Use atomics if access patterns are sparse or unpredictable
- Optimization comes with a development cost
- Memory resources ultimately limit parallelism

Thread Execution and Divergence

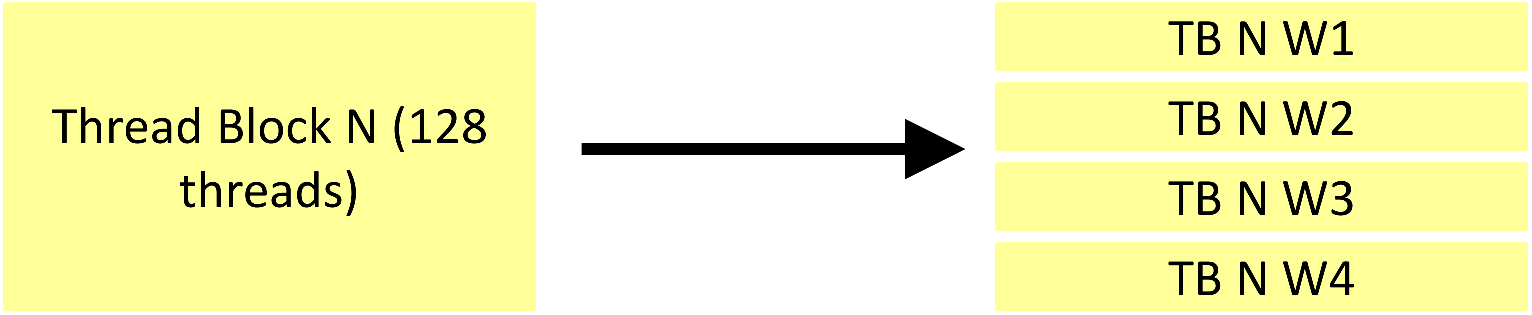
Scheduling Blocks onto SMs



- HW Schedules thread blocks onto available SMs
  - No guarantee of ordering among thread blocks
  - HW will schedule thread blocks as soon as a previous thread block finishes

Mapping of Thread Blocks

- Each thread block is mapped to one or more warps
- The hardware schedules each warp independently



Thread Block N (128 threads)

TB N W1

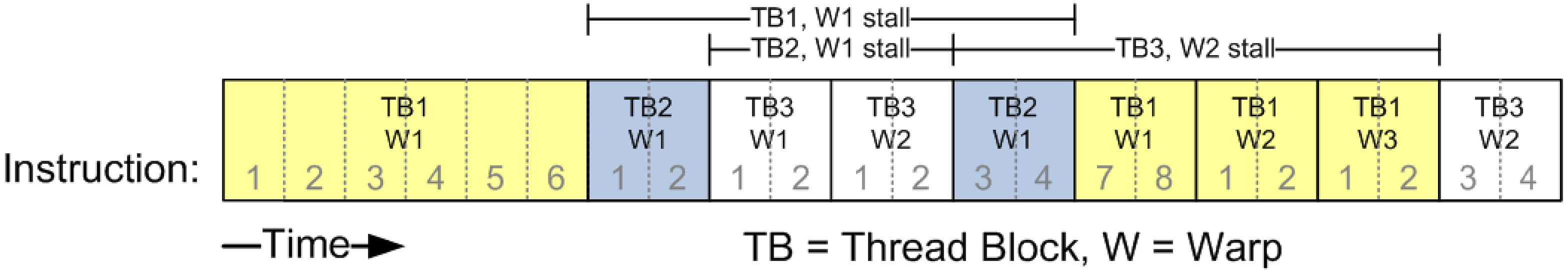
TB N W2

TB N W3

TB N W4

Thread Scheduling Example

- SM implements zero-overhead warp scheduling
  - At any time, only one of the warps is executed by SM
  - Warps whose next instruction has its inputs ready for consumption are eligible for execution
  - Eligible warps are selected for execution on a prioritized scheduling policy
  - All threads in a warp execute the same instruction when selected



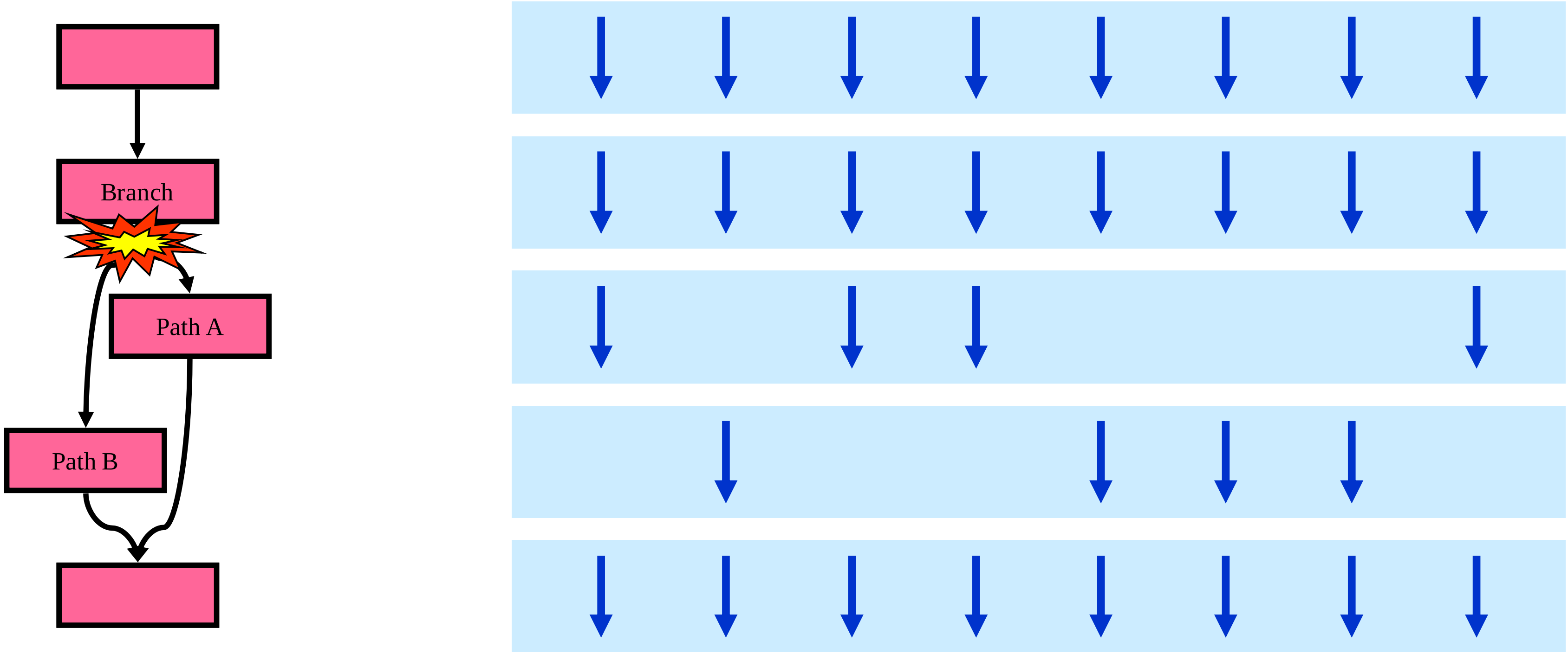
TB = Thread Block, W = Warp

Control Flow Divergence

- What happens if you have the following code?

```
if(foo(threadIdx.x))
{
  do_A();
}
else
{
  do_B();
}
```

Control Flow Divergence



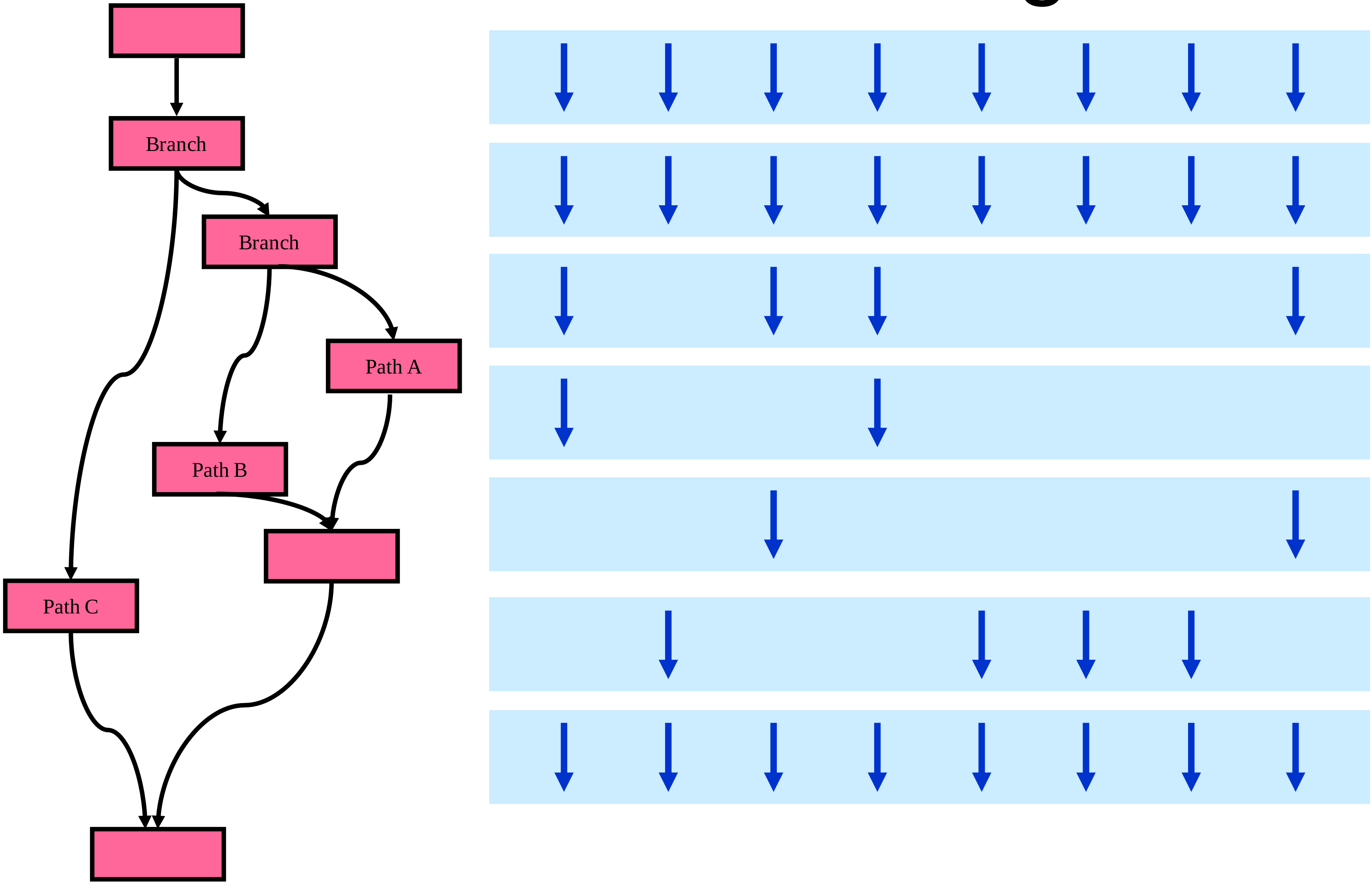
From Fung et al. MICRO ‘07

Control Flow Divergence

- Nested branches

```
if(foo(threadIdx.x))
{
  if(bar(threadIdx.x))
    do_A();
  else
    do_B();
}
else
  do_C();
```

Control Flow Divergence



Control Flow Divergence

- You don’t have to worry about divergence for correctness (*)
- You might have to think about it for performance
  - Depends on your branch conditions

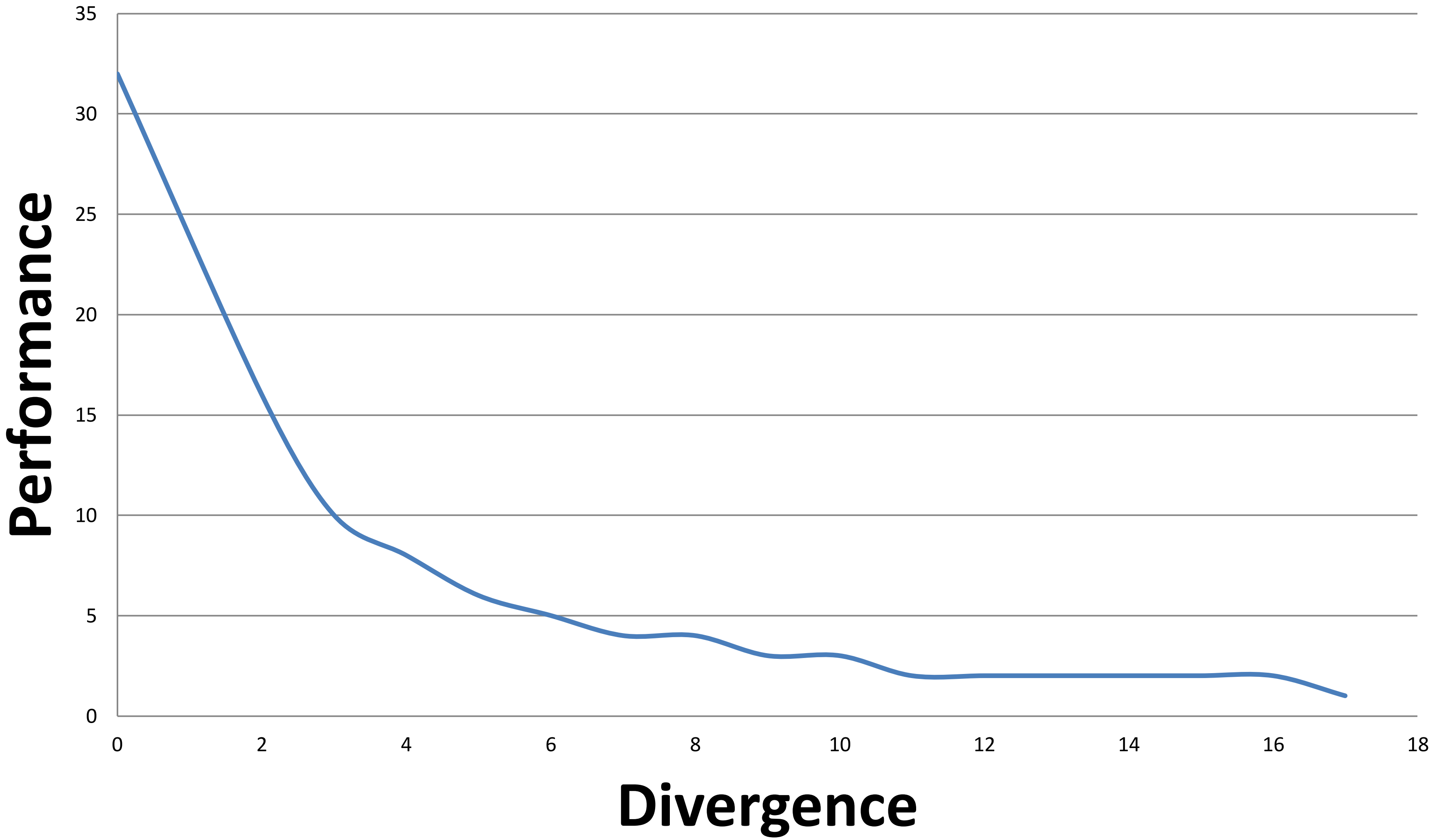
* Mostly true, except corner cases (for example intra-warp locks)

Control Flow Divergence

- Performance drops off with the degree of divergence

```
switch(threadIdx.x % N)
{
  case 0:
    ...
  case 1:
    ...
}
```

Divergence


Performance
35
30
25
20
15
10
5
0
0
2
4
6
8
10
12
14
16
18
Divergence

Atomics

The Problem

- How do you do global communication?
- Finish a grid and start a new one

Global Communication

- Finish a kernel and start a new one
- All writes from all threads complete before a kernel finishes

```
step1<<<grid1,blk1>>>(...);
// The system ensures that all
// writes from step1 complete.
step2<<<grid2,blk2>>>(...);
```

Global Communication

- Would need to decompose kernels into before and after parts

Race Conditions

- Or, write to a predefined memory location
    - Race condition! Updates can be lost

Race Conditions

```
threadId:0                   threadId:1917
     // vector[0] was equal to 0
vector[0] += 5;              vector[0] += 1;
...                          ...
a = vector[0];               a = vector[0];
```

- What is the value of `a` in thread 0?
- What is the value of `a` in thread 1917?

Race Conditions

- Thread 0 could have finished execution before 1917 started
- Or the other way around
- Or both are executing at the same time

- Answer: not defined by the programming model, can be arbitrary
- CUDA provides atomic operations to deal with this problem

Atomics

- An atomic operation guarantees that only a single thread has access to a piece of memory while an operation completes
- The name atomic comes from the fact that it is uninterruptable
- No dropped data, but ordering is still arbitrary
- Different types of atomic instructions
- `atomic{Add, Sub, Exch, Min, Max, Inc, Dec, CAS, And, Or, Xor}`
- More types in newer architectures

Compare and Swap

```
int compare_and_swap(int* register,
  int oldval, int newval)
{
  int old_reg_val = *register;
  if(old_reg_val == oldval)
     *register = newval;

  return old_reg_val;
}
```

- Most general type of atomic
- Can emulate all others with CAS

Example: Histogram

```
// Determine frequency of colors in a picture
// colors have already been converted into ints
// Each thread looks at one pixel and increments
// a counter atomically
__global__ void histogram(int* color,
                          int* buckets)
{
  int i = threadIdx.x
        + blockDim.x * blockIdx.x;
  int c = colors[i];
  atomicAdd(&buckets[c], 1);
}
```

Example: Workqueue

```
// For algorithms where the amount of work per item
// is highly non-uniform, it often makes sense
// to continuously grab work from a queue
__global__
void workq(int* work_q, int* q_counter,
           int* output, int queue_max)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  int q_index = atomicInc(q_counter, queue_max);
  int result = do_work(work_q[q_index]);
  output[i] = result;
}
```

Atomics

- Atomics are slower than normal load/store
- You can have the whole machine queuing on a single location in memory
- Atomics unavailable on G80

Example: Global Min/Max (Naive)

```
// If you require the maximum across all threads
// in a grid, you could do it with a single global
// maximum value, but it will be VERY slow
__global__
void global_max(int* values, int* gl_max)
{
  int i = threadIdx.x
        + blockDim.x * blockIdx.x;
  int val = values[i];
  atomicMax(gl_max,val);
}
```

Example: Global Min/Max (Better)

```
// introduce intermediate maximum results, so that
// most threads do not try to update the global max
__global__
void global_max(int* values, int* max,
                int *regional_maxes,
                int num_regions)
{
  // i and val as before …
  int region = i % num_regions;
  if(atomicMax(&reg_max[region],val) < val)
  {
    atomicMax(max,val);
  }
}
```

Global Min/Max

- Single value causes serial bottleneck
- Create hierarchy of values for more parallelism
- Performance will still be slow, so use judiciously

Atomics - Summary

- Can’t use normal load/store for inter-thread communication because of race conditions

- Use atomic instructions for sparse and/or unpredictable global communication

- Decompose data (very limited use of single global sum/max/min/etc.) for more parallelism