GPU PROGRAMMING

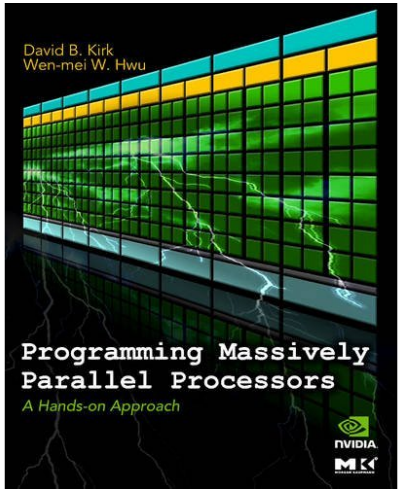
Assignment 4

- Consists of two programming assignments
  - Concurrency
  - GPU programming
    - Requires a computer with a CUDA/OpenCL/DirectCompute compatible GPU
- Due Jun 07
  - We have no final exams

GPU Resources

- Download CUDA toolkit from the web

- Very good text book:
  - Programming Massively Parallel Processors
    - Wen-mei Hwu and David Kirk
  - Available at
    - http://courses.engr.illinois.edu/ece498/al/Syllabus.html

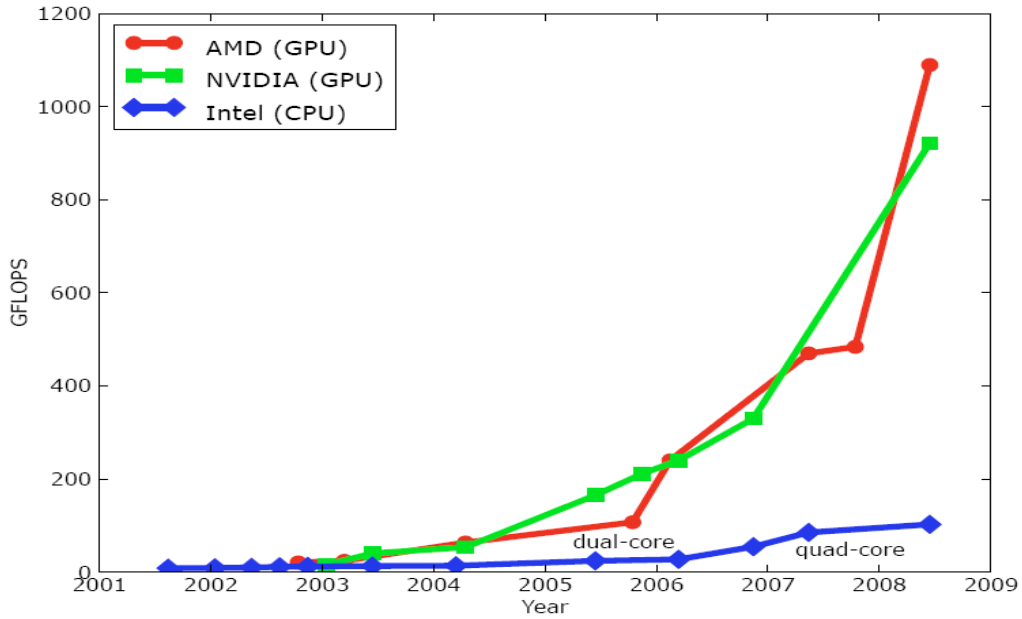


Acknowledgments

- Slides and material from
  - Wen-mei Hwu (UIUC) and David Kirk (NVIDIA)

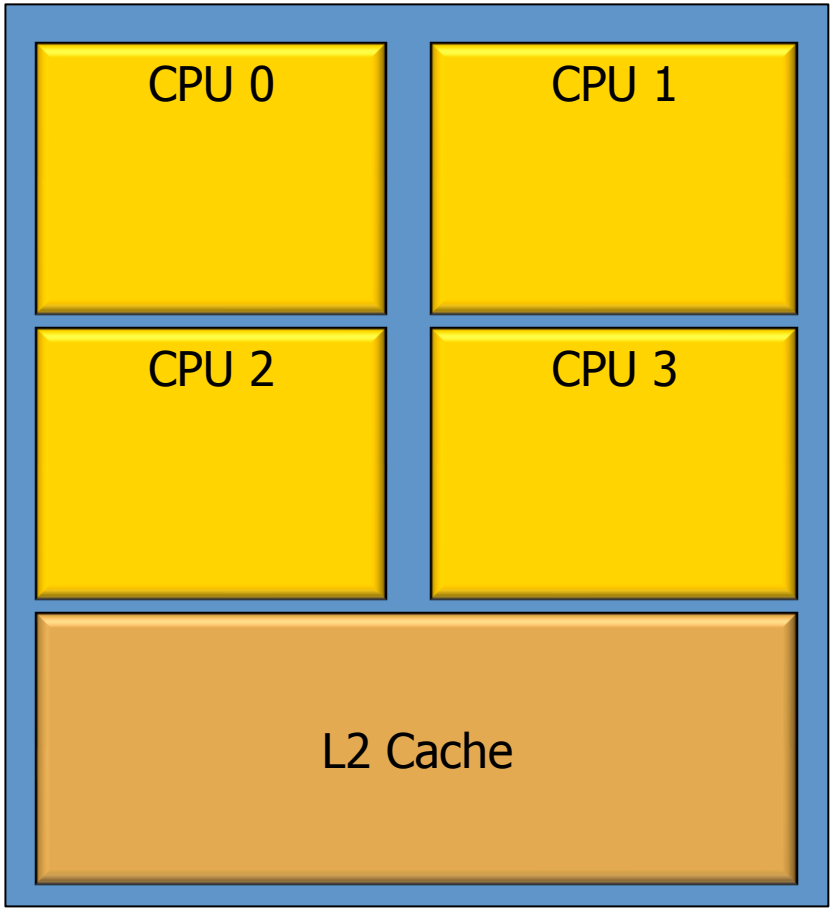
Why GPU Programming

- More processing power + higher memory bandwidth



- GPU in every PC and workstation – massive volume and potential impact

Current CPU



4 Cores

4 float wide SIMD

3GHz

48-96GFlops

2x HyperThreaded

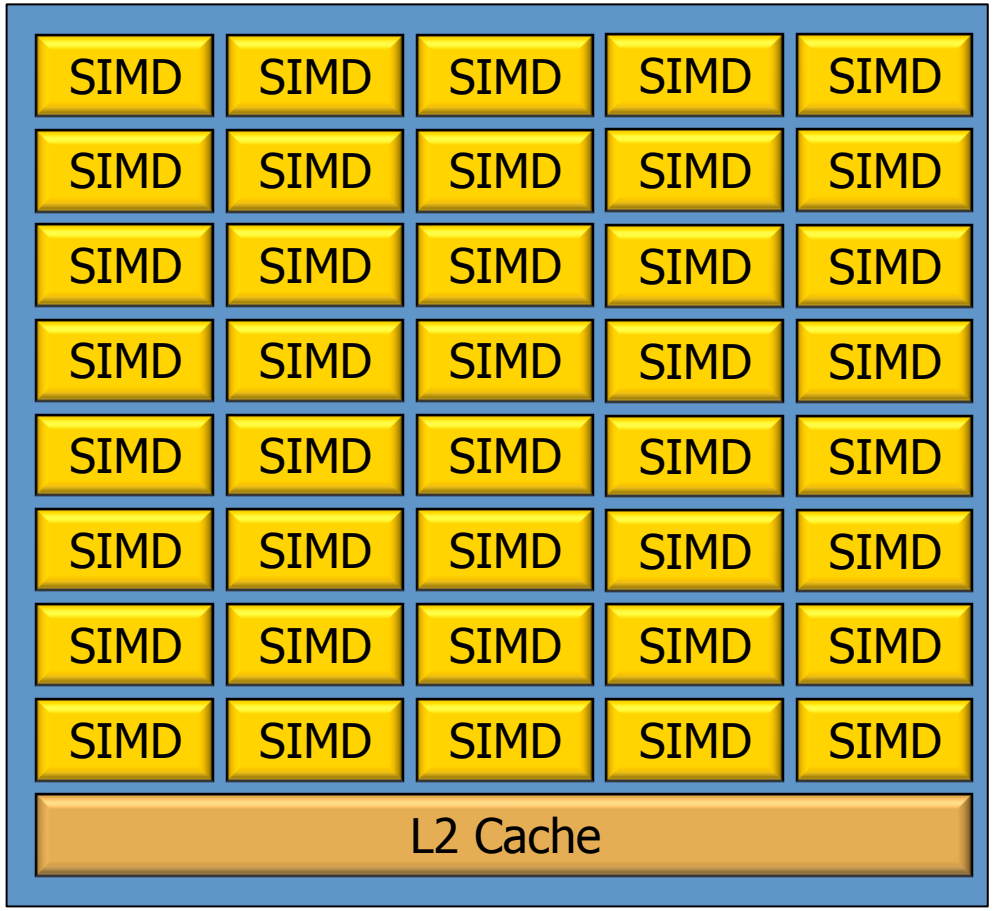
64kB $L1/core

20GB/s to Memory

$200

200W

Current GPU



32 Cores

32 Float wide

1GHz

1TeraFlop

32x “HyperThreaded”

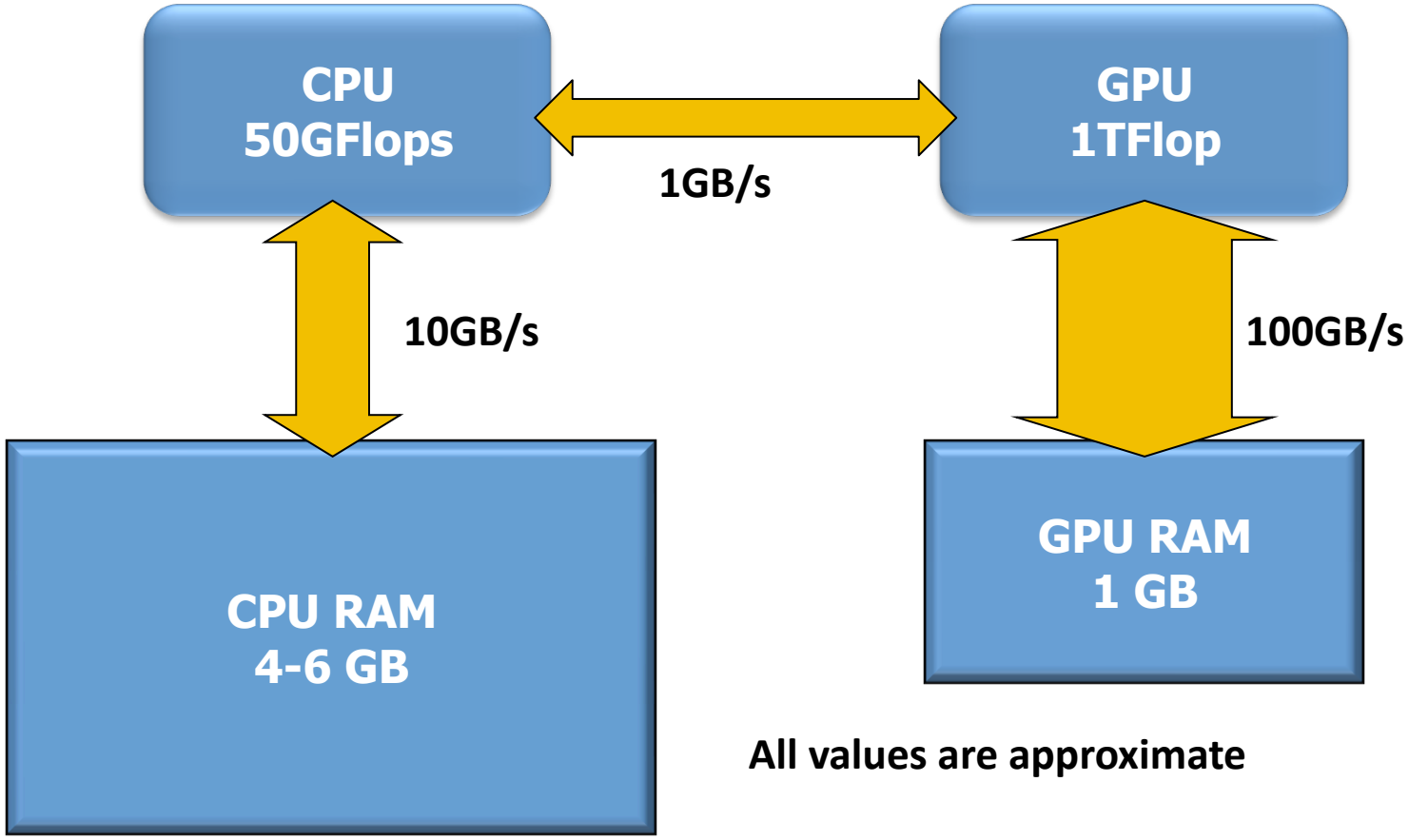
64kB $L1/Core

150GB/s to Mem

$200,

200W

Bandwidth and Capacity



All values are approximate

CUDA

- "Compute Unified Device Architecture"
- General purpose programming model
  - User kicks off batches of threads on the GPU
  - GPU = dedicated super-threaded, massively data parallel co-processor
- Targeted software stack
  - Compute oriented drivers, language, and tools
- Driver for loading computation programs into GPU

Languages with Similar Capabilities

- CUDA
- OpenCL
- DirectCompute

- You are free to use any of the above for assignment 4
- I will focus on CUDA for the rest of the lecture
  - Same abstractions present in all three with different (and confusing) names

CUDA Programming Model:

- The GPU = compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
- GPU program = kernel
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

A CUDA Program

1. Host performs some CPU computation
2. Host copies input data into the device
3. Host instructs the device to execute a kernel
4. Device executes the kernel produces results
5. Host copies the results
6. Goto step 1

CUDA Kernel is a SPMD program

- SPMD = Single Program Multiple Data
- All threads run the same code
- Each thread uses its id to
  - Operate on different memory addresses
  - Make control decisions

```
Kernel:
…
i = input[tid];
o = f(i);
output[tid] = o;
…
```

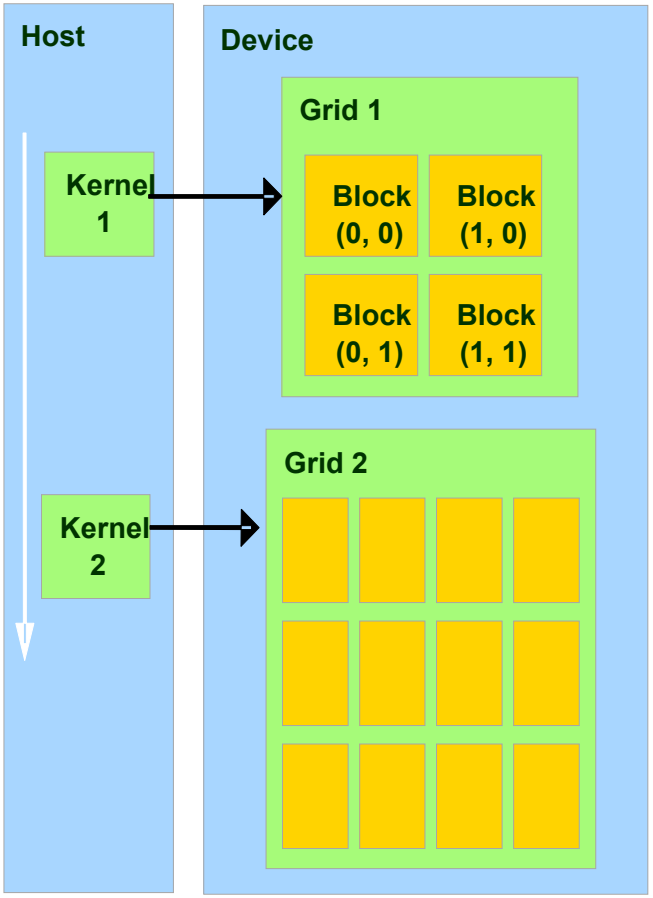
CUDA Kernel is a SPMD program

- SPMD = Single Program Multiple Data
- All threads run the same code
- Each thread uses its id to
  - Operate on different memory addresses
  - Make control decisions
- Difference with SIMD
  - Threads can execute different control flow
  - At a performance cost

```
Kernel:
…
i = input[tid];
if(i%2 == 0)
  o = f(i);
else
  o = g(i);
output[tid] = o;
…
```

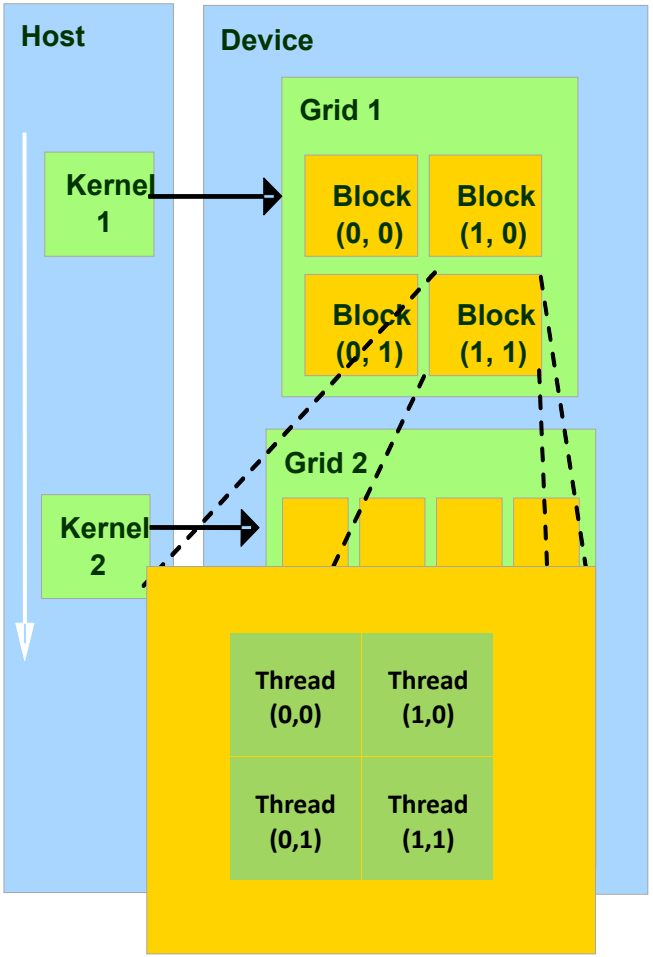
Threads Organization

- Kernel threads
  = Grid of Thread Blocks
  (1D or 2D)
- Thread Block
  = Array of Threads
  (1D or 2D or 3D)
- Simplifies memory addressing for multidimensional data



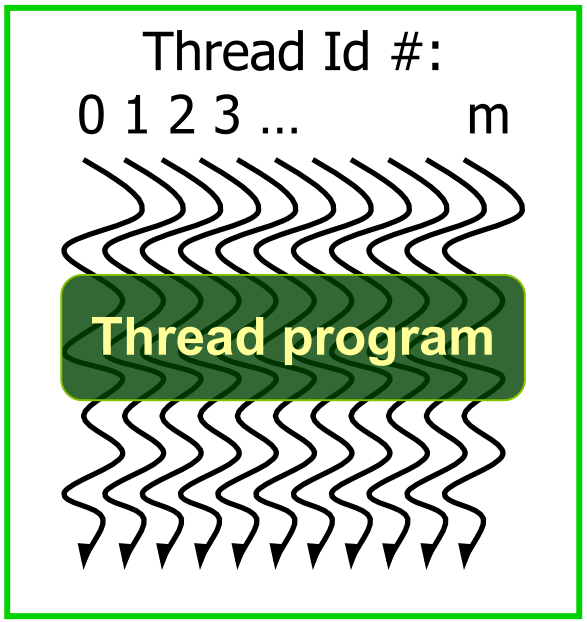
Threads Organization

- Kernel threads
  = Grid of Thread Blocks
  (1D or 2D)
- Thread Block
  = Array of Threads
  (1D or 2D or 3D)
- Simplifies memory addressing for multidimensional data



Threads within a Block

- Execute in lock step
- Can share memory
- Can synchronize with each other



Courtesy: John Nickolls, NVIDIA

CUDA Function Declarations

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- `__global__` defines a kernel function
  - Must return `void`
- `__device__` and `__host__` can be used together

CUDA Function Declarations (cont.)

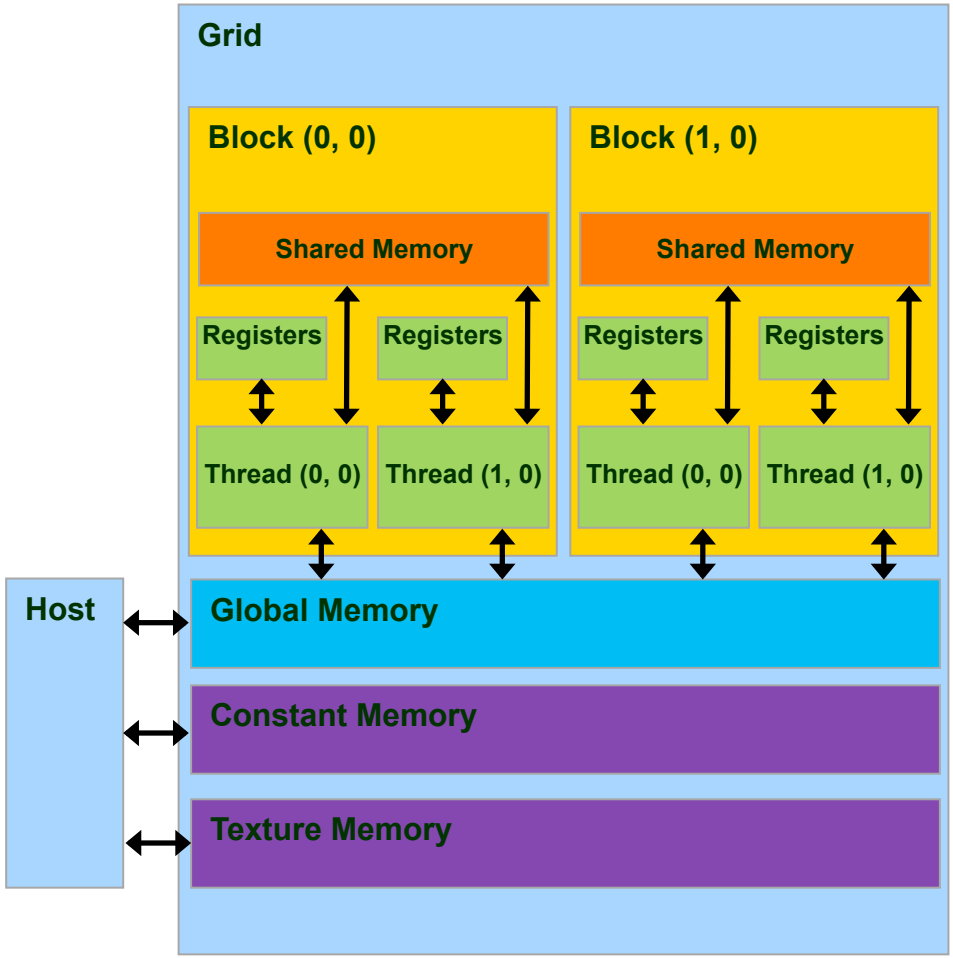
- `__device__` functions cannot have their address taken
- For functions executed on the device:
  - No recursion
  - No static variable declarations inside the function
  - No variable number of arguments

Putting it all together

```
__global__ void KernelFunc(…)
dim3   DimGrid(100, 50);
dim3   DimBlock(4, 8, 8);

KernelFunc<<< DimGrid, DimBlock >>>(...);
```

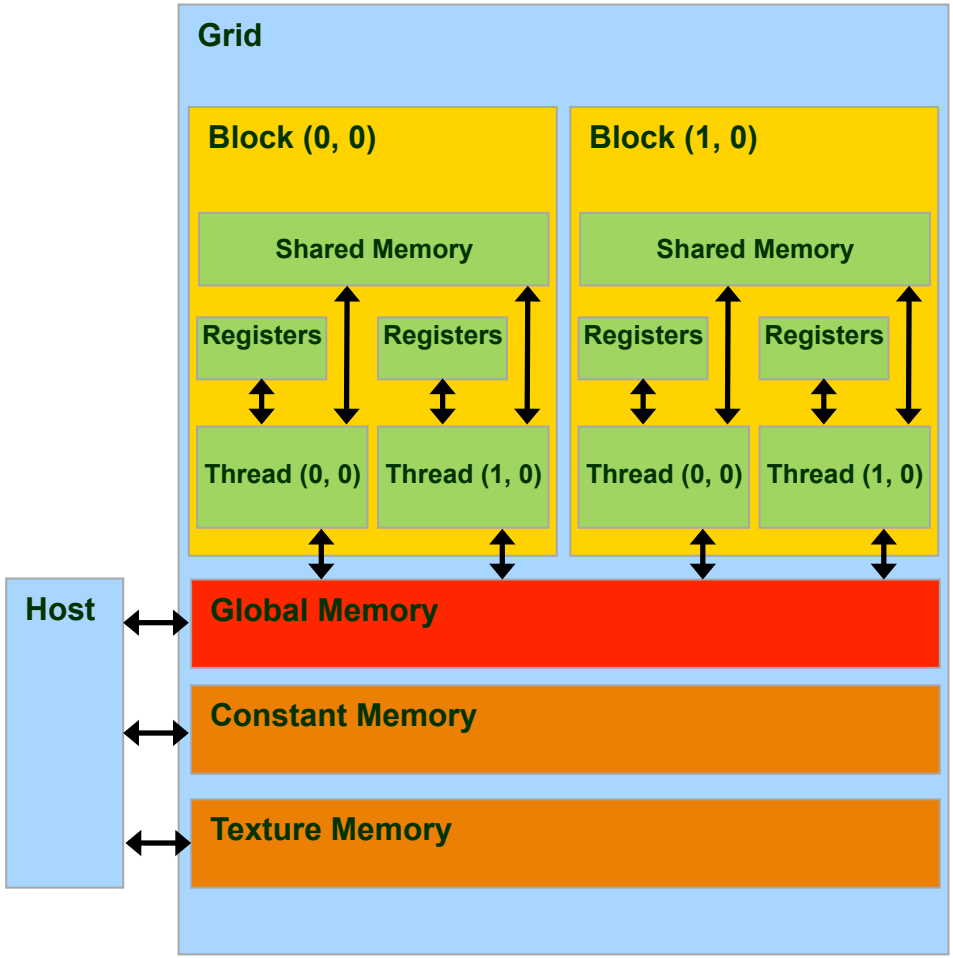
CUDA Memory Model

- Registers
  - Read/write per thread
- Local memory
  - Read/write per thread
- Shared memory
  - Read/write per block
- Global memory
  - Read/write per grid
- Constant memory
  - Read only, per grid
- Texture memory
  - Read only, per grid



Memory Access Efficiency

- Registers
  - Fast
- Local memory
  - Not cached -> Slow
  - Registers spill into local memory
- Shared memory
  - On chip -> Fast
- Global memory
  - Not cached -> Slow
- Constant memory
  - Cached – Fast if good reuse
- Texture memory
  - Cached – Fast if good reuse



CUDA Variable Type Qualifiers

Variable declaration	Memory	Scope	Lifetime
`__device__ __local__ int LocalVar;`	local	thread	thread
`__device__ __shared__ int SharedVar;`	shared	block	block
`__device__ int GlobalVar;`	global	grid	application
`__device__ __constant__ int ConstantVar;`	constant	grid	application

- **`__device__`** is optional when used with **`__local__`**, **`__shared__`**, or **`__constant__`**
- Automatic variables without any qualifier reside in a register
  - Except arrays that reside in local memory

Variable Type Restrictions

- Pointers can only point to memory allocated or declared in global memory:
  - Allocated in the host and passed to the kernel:

```
__global__ void KernelFunc(float* ptr)
```

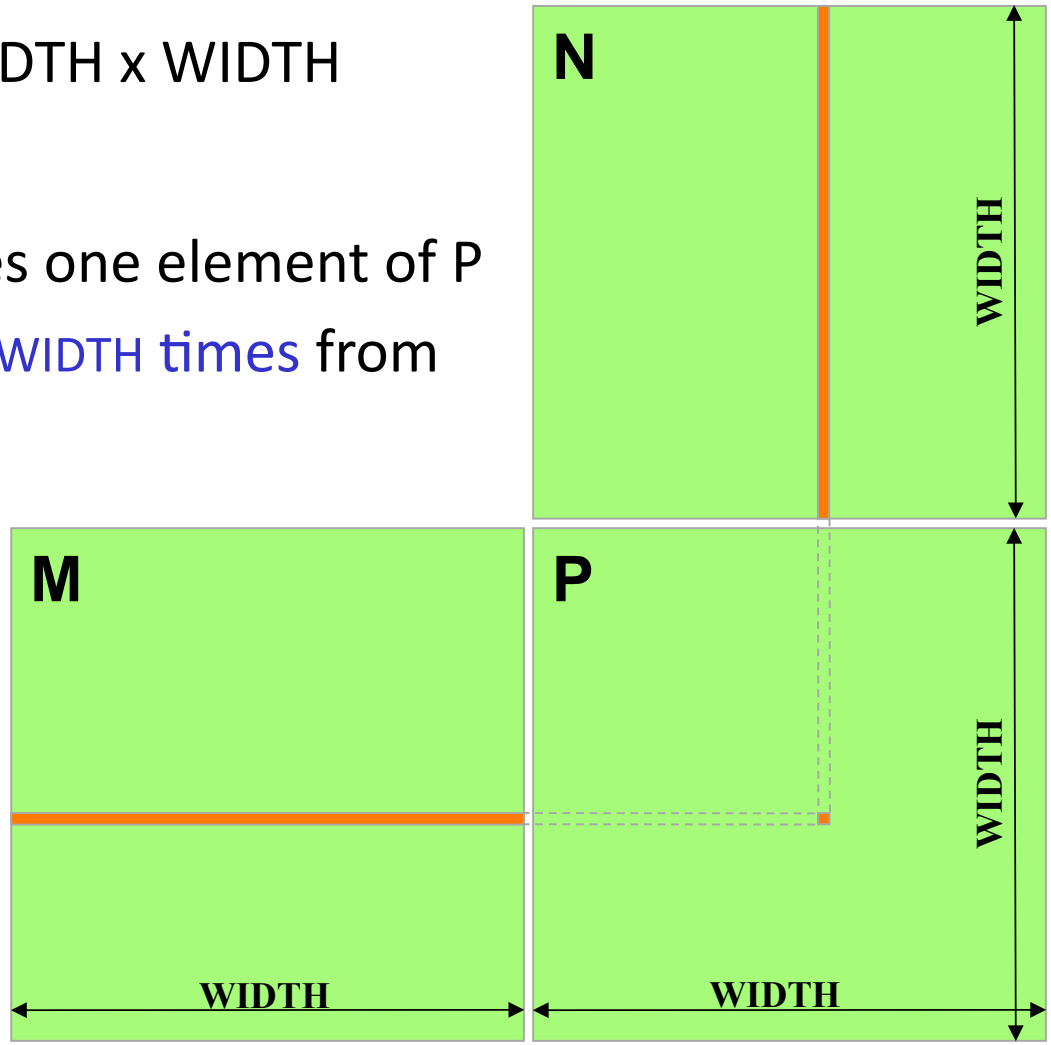
  - Obtained as the address of a global variable:

```
float* ptr = &GlobalVar;
```

Simple Example: Matrix Multiplication

Matrix Multiplication

- P = M * N of size WIDTH x WIDTH
- Simple strategy
  - One thread calculates one element of P
  - M and N are loaded WIDTH times from global memory



GPU Matrix Multiplication: Host

```
float *M, *N, *P; int width;
int size = width * width * sizeof(float);

cudaMalloc(&Md, size);
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
```

GPU Matrix Multiplication: Host

```
float *M, *N, *P; int width;
int size = width * width * sizeof(float);

cudaMalloc(&Md, size);
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

cudaMalloc(&Nd, size);
cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

cudaMalloc(&Pd, size);
```

GPU Matrix Multiplication: Host

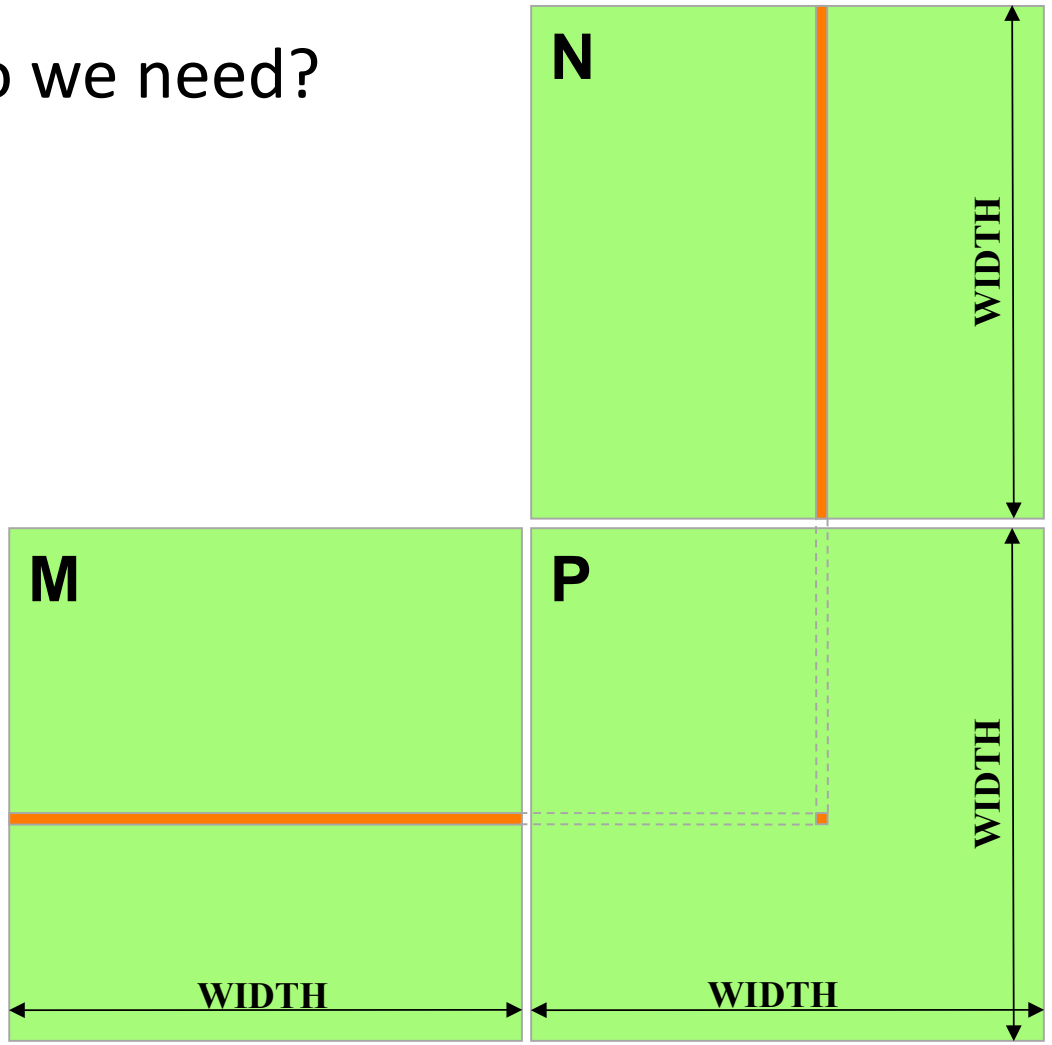
```
float *M, *N, *P; int width;
int size = width * width * sizeof(float);

cudaMalloc(&Md, size);
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

cudaMalloc(&Nd, size);
cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

cudaMalloc(&Pd, size);

// call kernel

cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
```

GPU Matrix Multiplication: Host

```
float *M, *N, *P; int width;
int size = width * width * sizeof(float);

cudaMalloc(&Md, size);
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

cudaMalloc(&Nd, size);
cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

cudaMalloc(&Pd, size);

// call kernel

cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
cudaFree(Md); cudaFree(Nd); cudaFree(Pd);
```

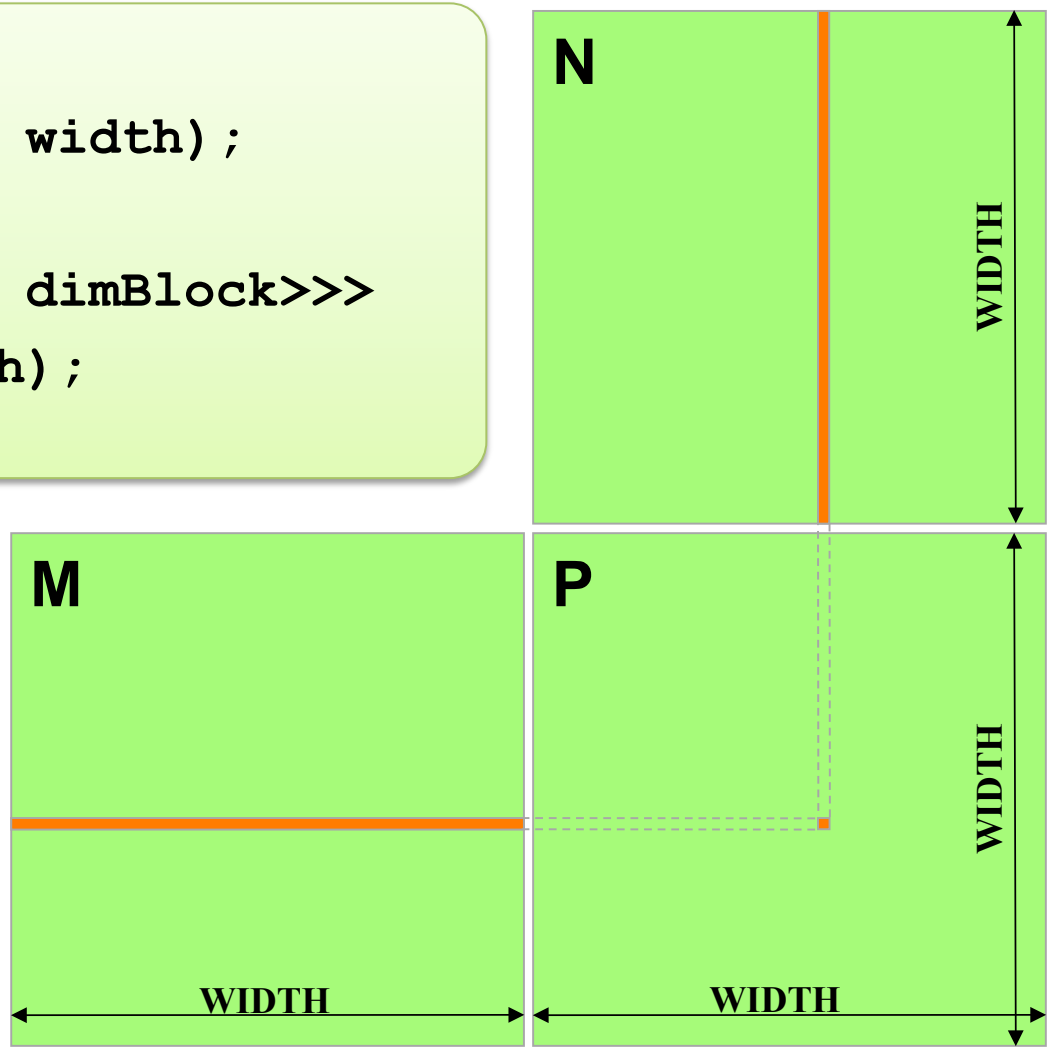
GPU Matrix Multiplication: Host

- How many threads do we need?



GPU Matrix Multiplication: Host

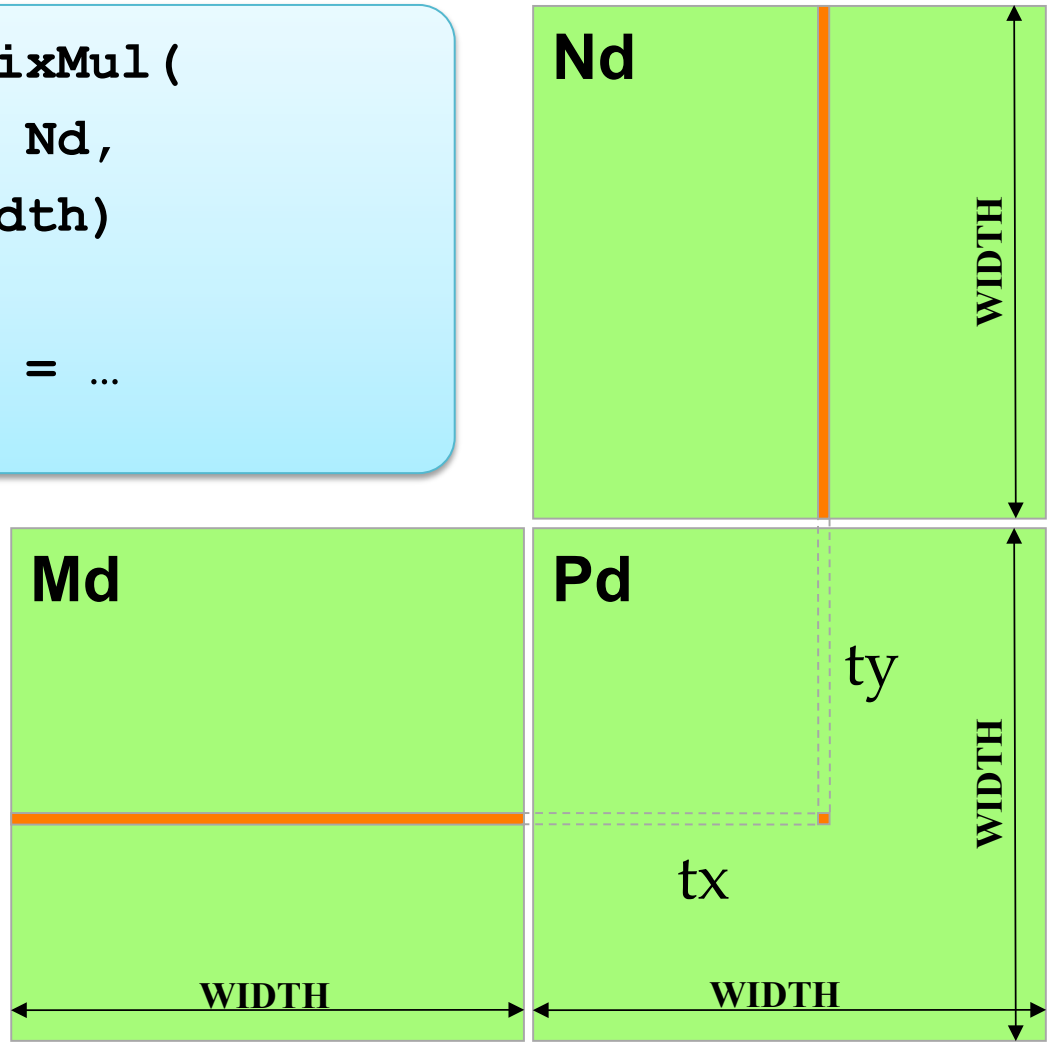
```
dim3 dimGrid(1,1);
dim3 dimBlock(width, width);

MatrixMul<<<dimGrid, dimBlock>>>
    (Md, Nd, Pd, width);
```



GPU Matrix Multiplication: Kernel

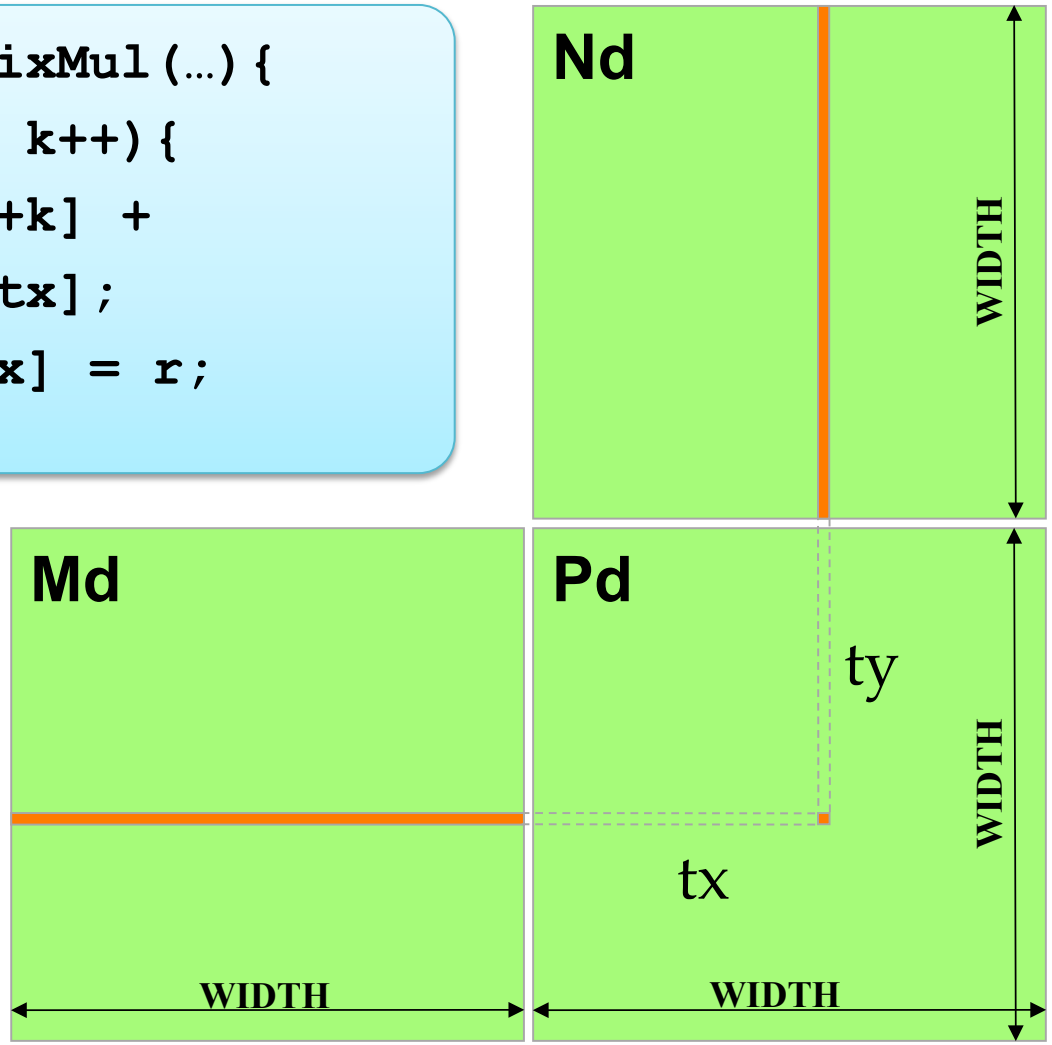
```
__global__ void MatrixMul(
    float* Md, float* Nd,
    float* Pd, int width)
{
    Pd[ty*width + tx] = …
}
```

```
short forms:
tx = threadIdx.x;
ty = threadIdx.y;
```



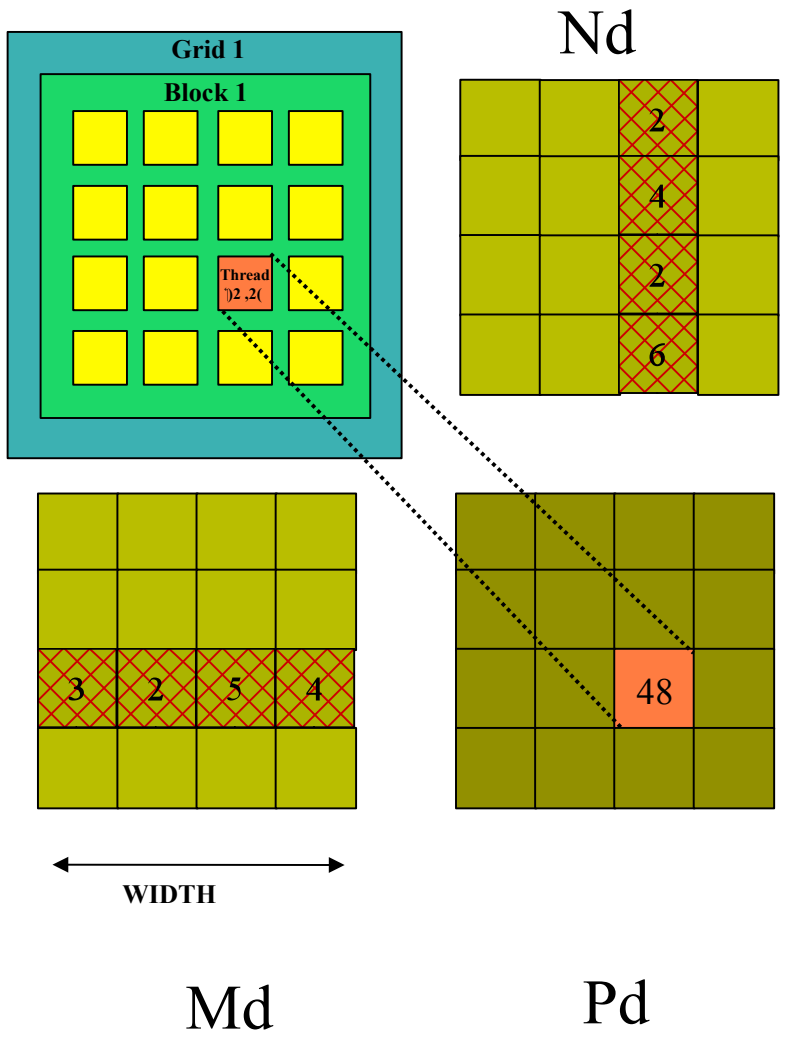
GPU Matrix Multiplication: Kernel

```
__global__ void MatrixMul(…){
   for(k=0; k<width; k++){
     r = Md[ty*width+k] +
         Nd[k*width+tx];
     Pd[ty*width + tx] = r;
  }}
```



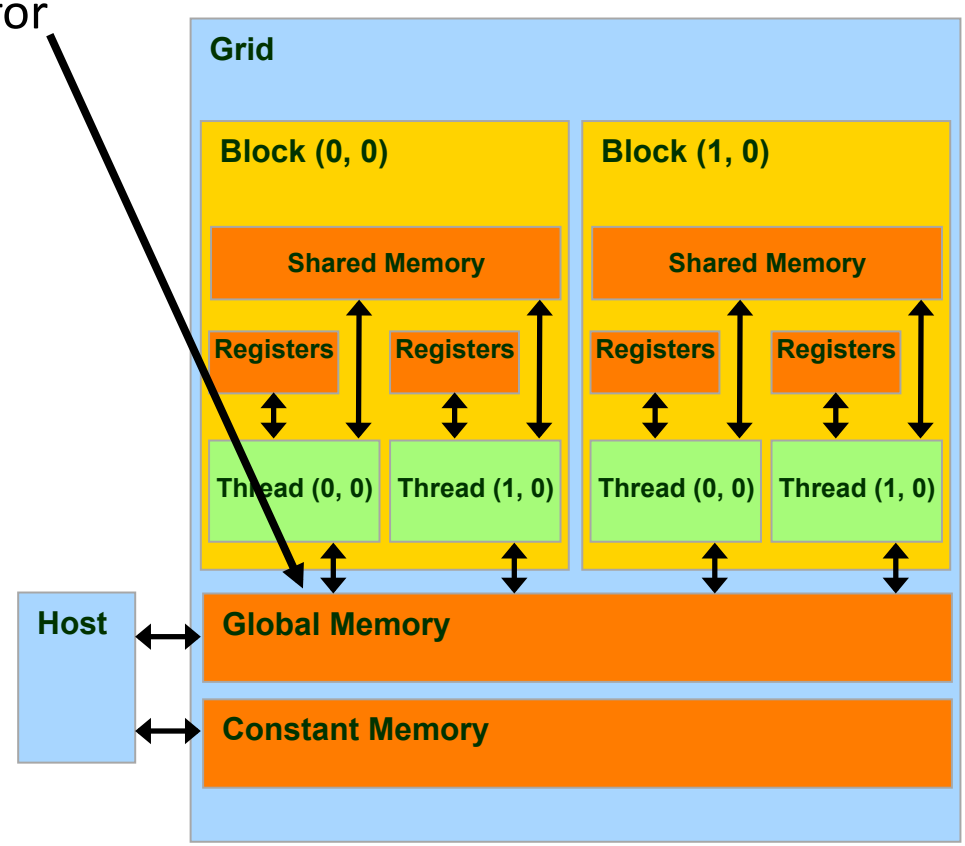
Only One Thread Block Used

- One Block of threads compute matrix Pd
  - Each thread computes one element of Pd
- Each thread
  - Loads a row of matrix Md
  - Loads a column of matrix Nd
  - Perform one multiply and addition for each pair of Md and Nd elements
  - Compute to off-chip memory access ratio close to 1:1 (not very high)
- Size of matrix limited by the number of threads allowed in a thread block



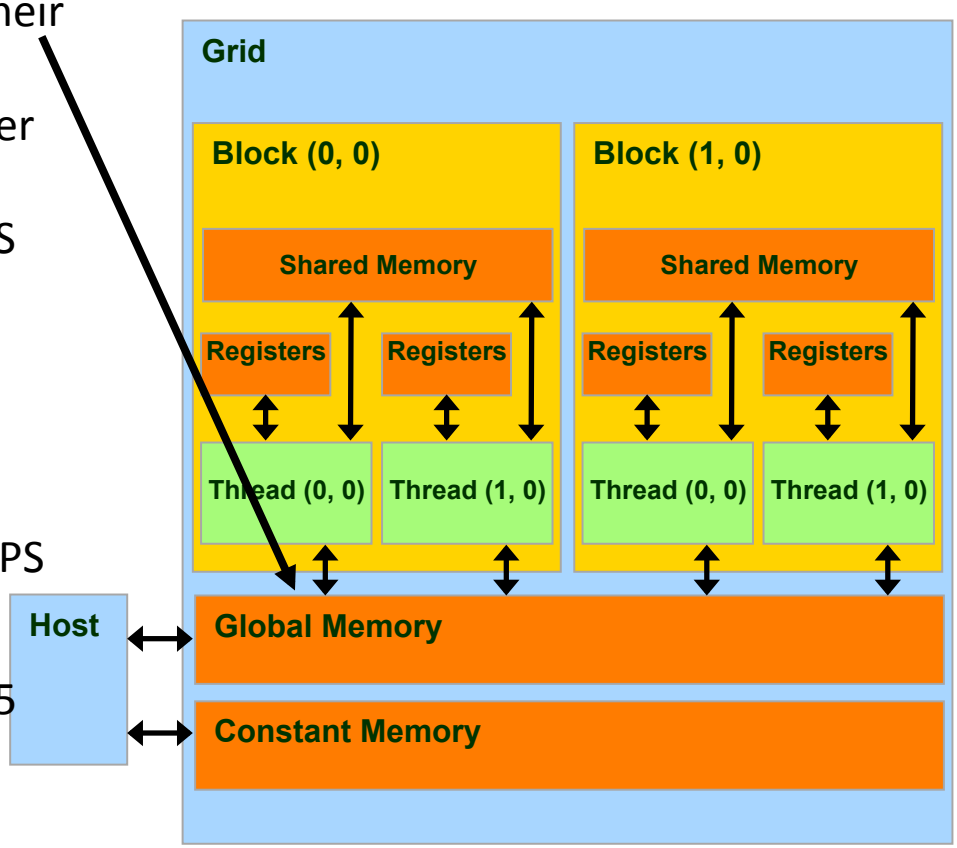
How about performance on G80?

- All threads access global memory for their input matrix elements
- Compute: 346.5 GFLOPS
- Memory bandwidth: 86.4 GBps

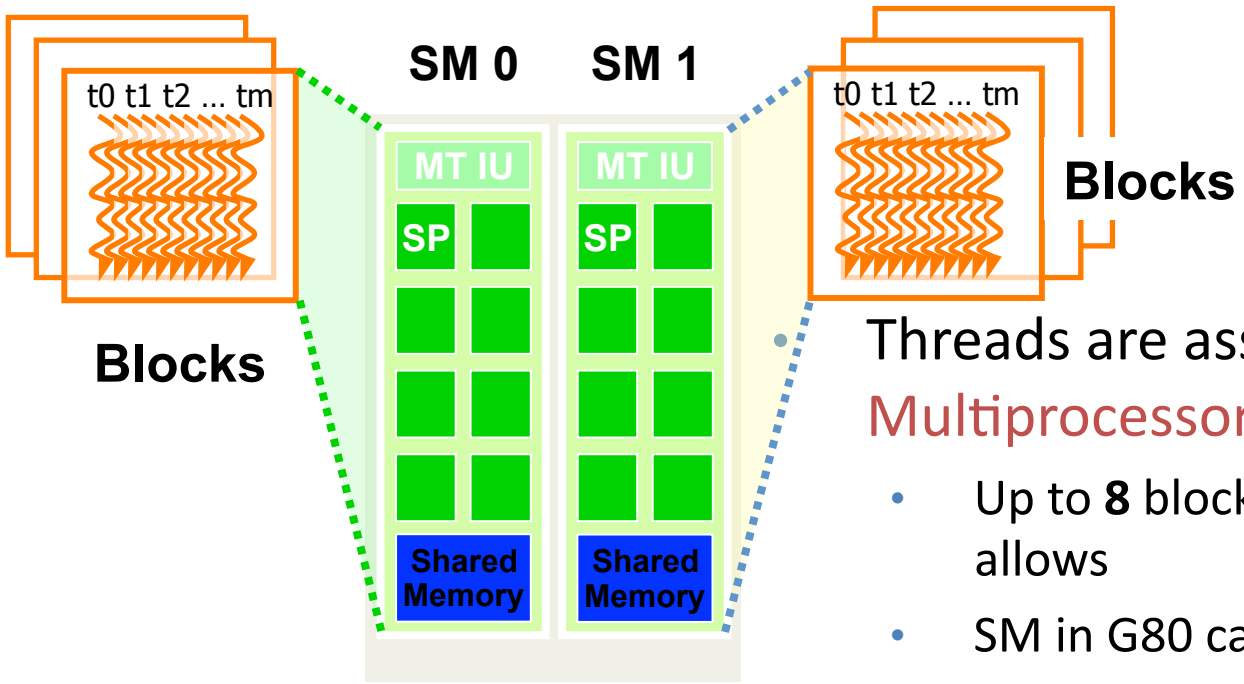


How about performance on G80?

- 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



G80 Example: Executing Thread Blocks

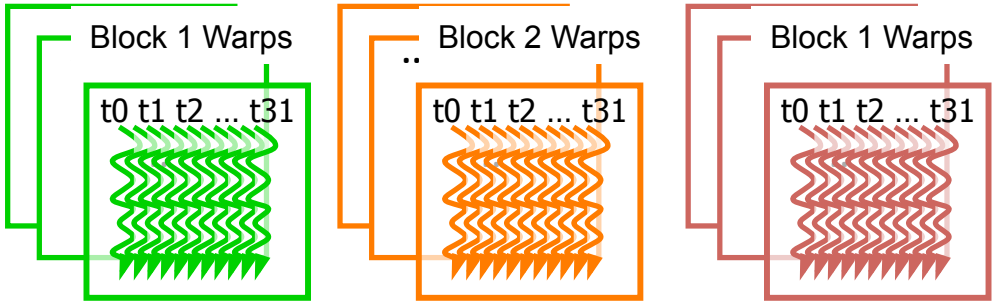


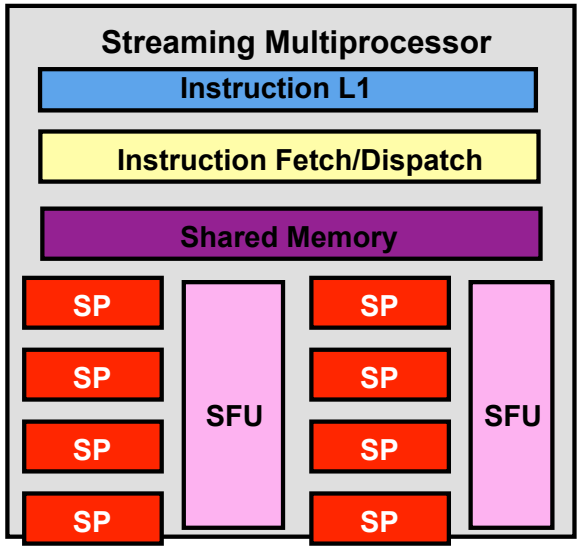
- Threads are assigned to Streaming Multiprocessors in block granularity
  - Up to **8** blocks to each SM as resource allows
  - SM in G80 can take up to **768** threads
    - Could be 256 (threads/block) * 3 blocks
    - Or 128 (threads/block) * 6 blocks, etc.
- Threads run concurrently
  - SM maintains thread/block id #s
  - SM manages/schedules thread execution

G80 Example: Thread Scheduling

- Each Block is executed as 32-thread Warps
  - Warps are scheduling units in SM

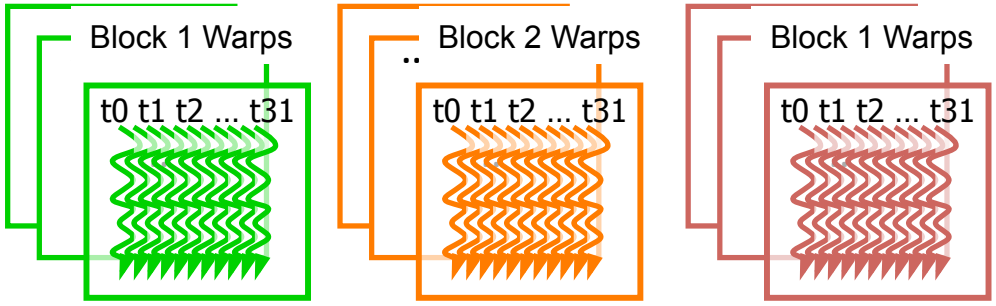
- If 3 blocks are assigned to an SM and each block has 256 threads, how many Warps are there in an SM?



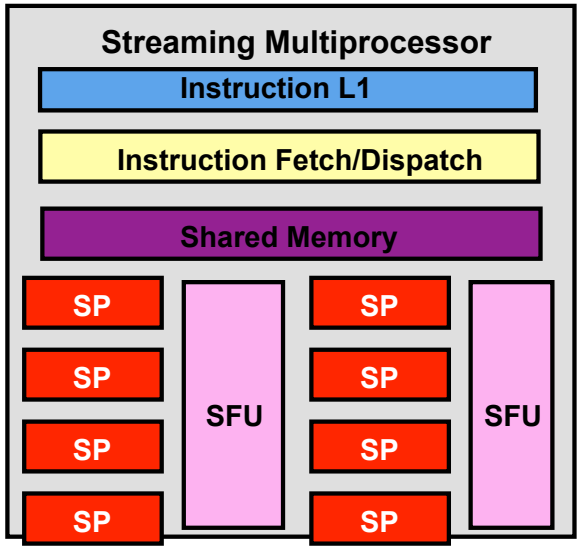


G80 Example: Thread Scheduling

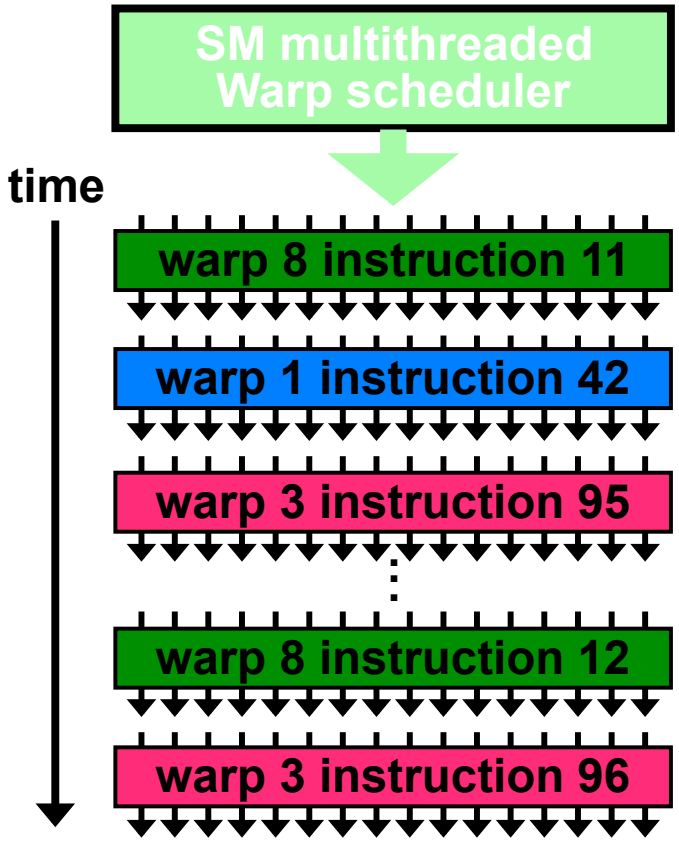
- Each Block is executed as 32-thread Warps
  - Warps are scheduling units in SM



- 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



SM Warp Scheduling



- SM hardware 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?
  - Each SM can take max 8 blocks and max 768 threads

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!

A Common Programming Strategy

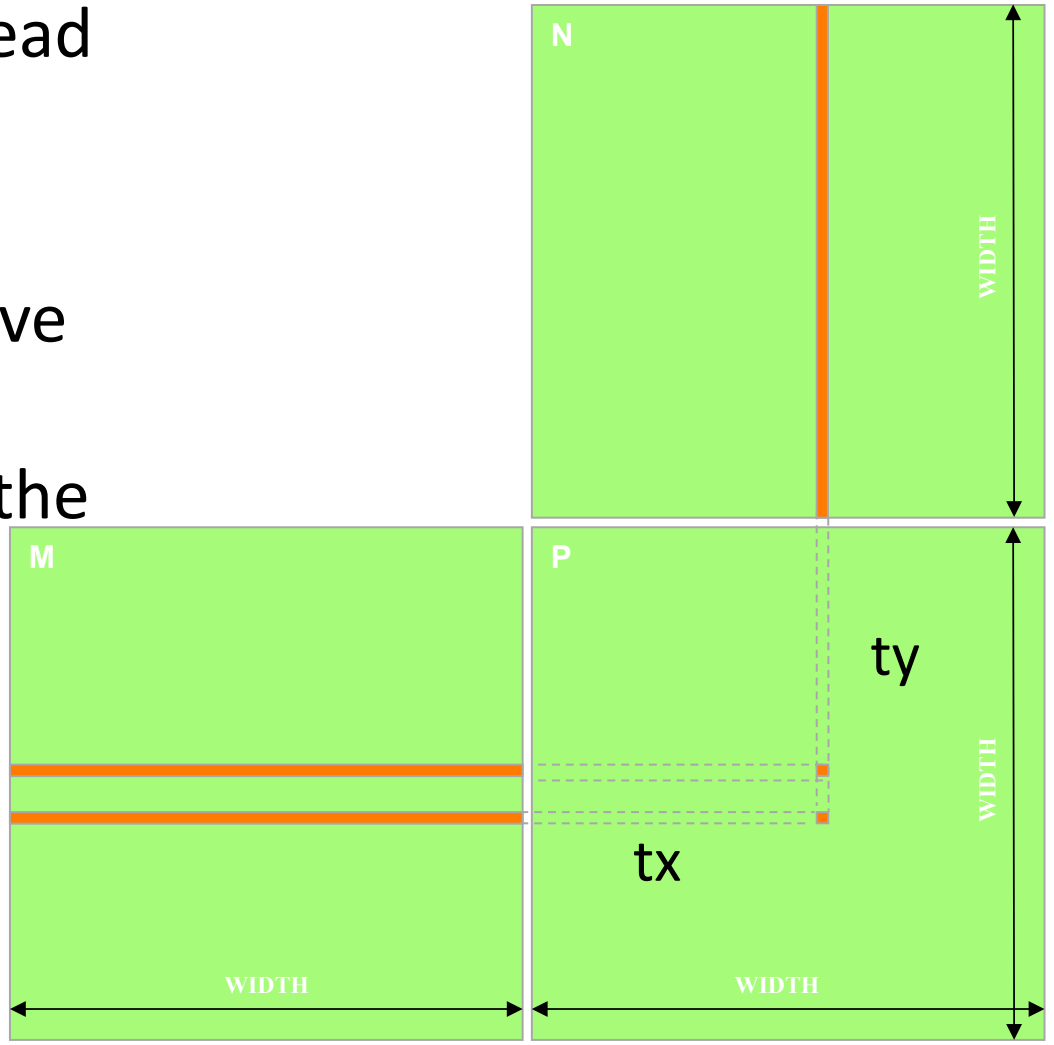
- Global memory resides in device memory (DRAM) - much slower access than shared memory
- So, a profitable way of performing computation on the device is to tile data to take advantage of fast shared memory:
  - Partition data into subsets that fit into shared memory
  - Handle each data subset with one thread block by:
    - Loading the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism
    - Performing the computation on the subset from shared memory; each thread can efficiently multi-pass over any data element
    - Copying results from shared memory to global memory

A Common Programming Strategy (Cont.)

- Constant memory also resides in device memory (DRAM) - much slower access than shared memory
  - But… cached!
  - Highly efficient access for read-only data
- Carefully divide data according to access patterns
  - R/Only → constant memory (very fast if in cache)
  - R/W shared within Block → shared memory (very fast)
  - R/W within each thread → registers (very fast)
  - R/W inputs/results → global memory (very slow)

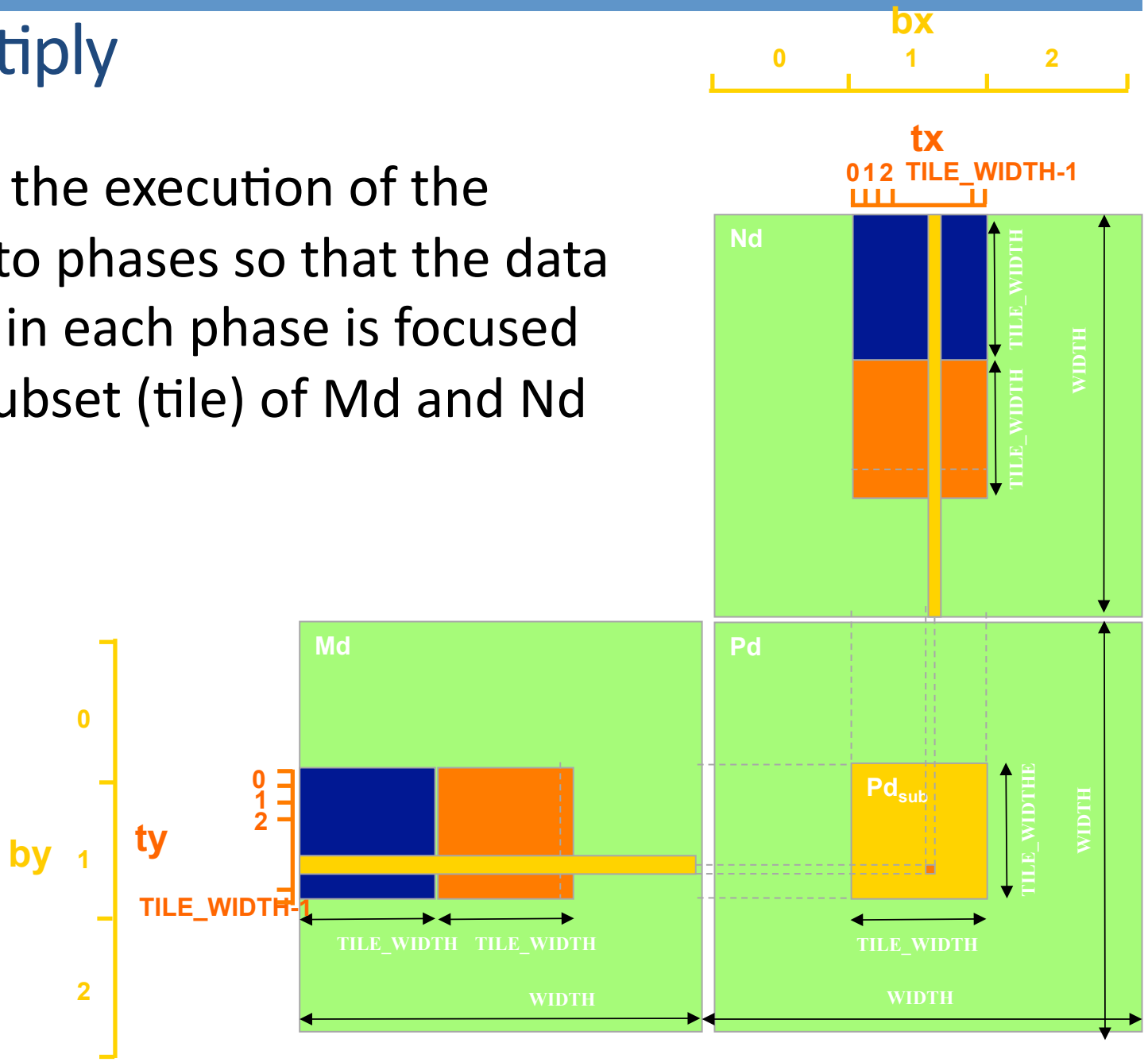
Idea: Use Shared Memory to reuse global memory data

- 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

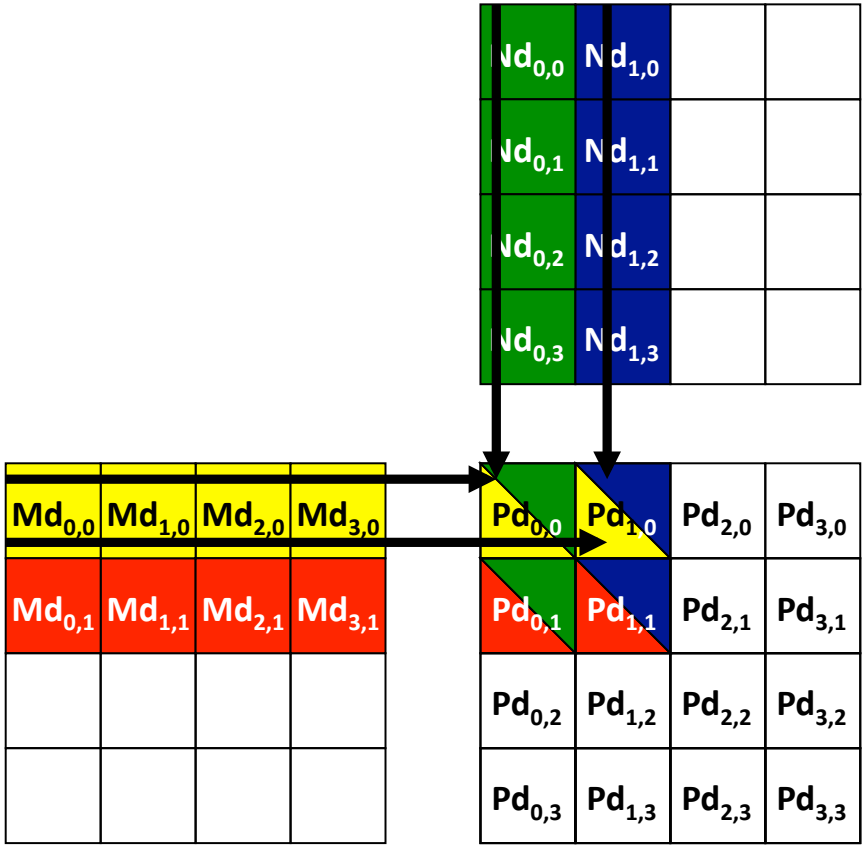


Tiled Multiply

- 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: 2X2 Tiling of P



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

Access order

$P_{0,0}$ $thread_{0,0}$	$P_{1,0}$ $thread_{1,0}$	$P_{0,1}$ $thread_{0,1}$	$P_{1,1}$ $thread_{1,1}$
$M_{0,0} * N_{0,0}$	$M_{0,0} * N_{1,0}$	$M_{0,1} * N_{0,0}$	$M_{0,1} * N_{1,0}$
$M_{1,0} * N_{0,1}$	$M_{1,0} * N_{1,1}$	$M_{1,1} * N_{0,1}$	$M_{1,1} * N_{1,1}$
$M_{2,0} * N_{0,2}$	$M_{2,0} * N_{1,2}$	$M_{2,1} * N_{0,2}$	$M_{2,1} * N_{1,2}$
$M_{3,0} * N_{0,3}$	$M_{3,0} * N_{1,3}$	$M_{3,1} * N_{0,3}$	$M_{3,1} * N_{1,3}$

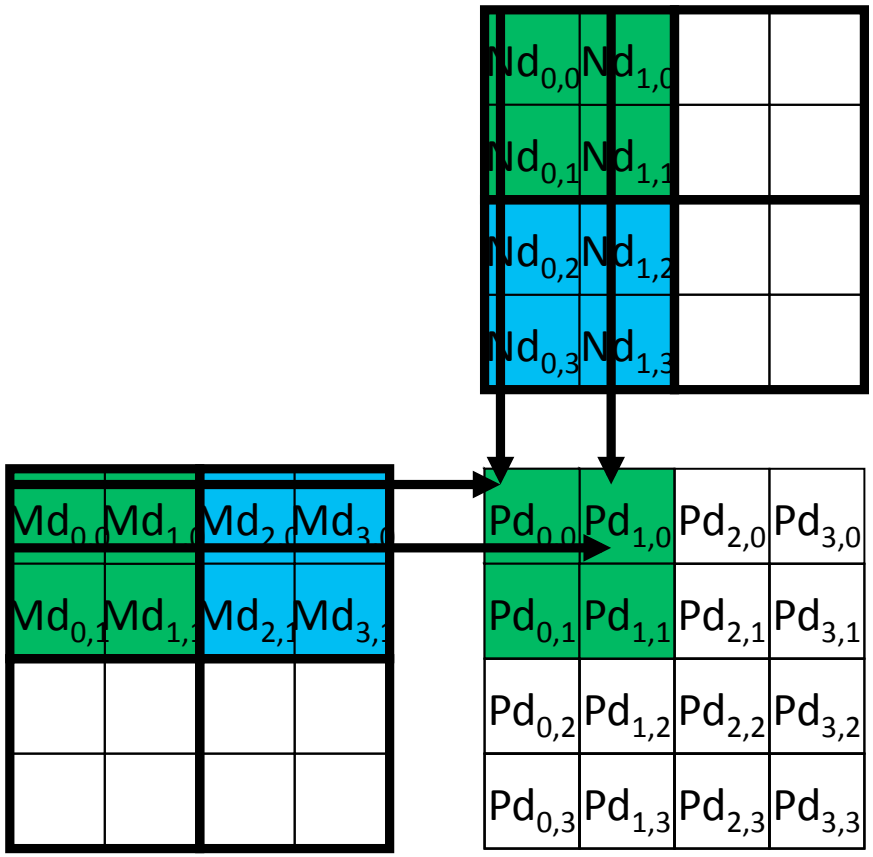
Every Md and Nd Element is used exactly twice in generating a 2X2 tile of P

Access order

$P_{0,0}$ $thread_{0,0}$	$P_{1,0}$ $thread_{1,0}$	$P_{0,1}$ $thread_{0,1}$	$P_{1,1}$ $thread_{1,1}$
$M_{0,0} * N_{0,0}$	$M_{0,0} * N_{1,0}$	$M_{0,1} * N_{0,0}$	$M_{0,1} * N_{1,0}$
$M_{1,0} * N_{0,1}$	$M_{1,0} * N_{1,1}$	$M_{1,1} * N_{0,1}$	$M_{1,1} * N_{1,1}$
$M_{2,0} * N_{0,2}$	$M_{2,0} * N_{1,2}$	$M_{2,1} * N_{0,2}$	$M_{2,1} * N_{1,2}$
$M_{3,0} * N_{0,3}$	$M_{3,0} * N_{1,3}$	$M_{3,1} * N_{0,3}$	$M_{3,1} * N_{1,3}$

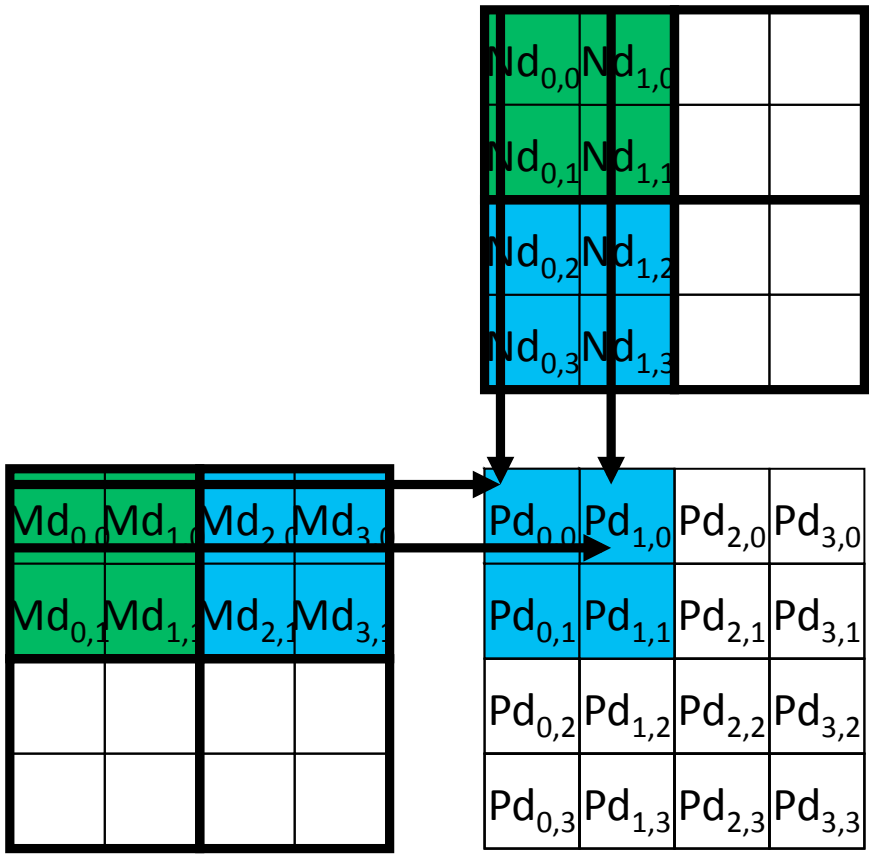
Breaking Md and Nd into Tiles

- 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 access the Md and Nd elements from the shared memory during the phase



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 access the Md and Nd elements from the shared memory during the phase



Tiled Kernel

```
__global__
void Tiled(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;
    // compute Pvalue
    Pd[Row*Width + Col] = Pvalue;
}
```

Tiled Kernel: Computing Pvalue

```
//…
float Pvalue = 0;
// Loop over the Md and Nd tiles required
for (int m = 0; m < Width/TILE_WIDTH; ++m) {
   // Collaborative loading of Md and Nd tiles   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

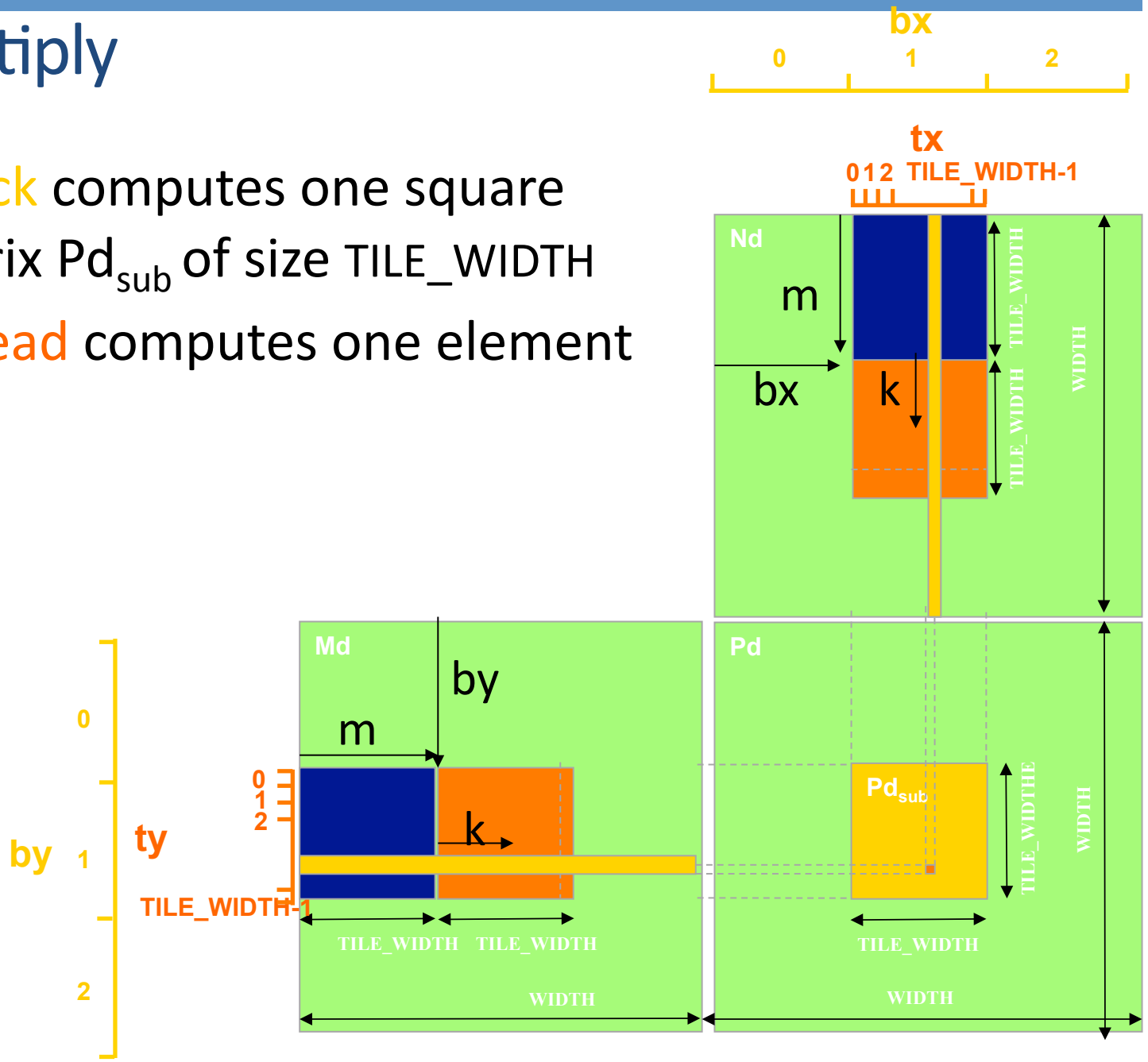
```
// Setup the execution configuration
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);
dim3 dimGrid(Width  / TILE_WIDTH,
             Width /  TILE_WIDTH);
```

First-order Size Considerations in G80

- 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 perform 2*256 = 512 float loads from global memory for 256 * (2*16) = 8,192 mul/add operations.
  - 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}



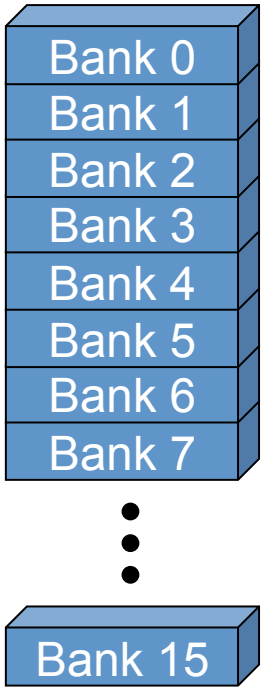
G80 Shared Memory and Threading

- 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 shared memory 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!

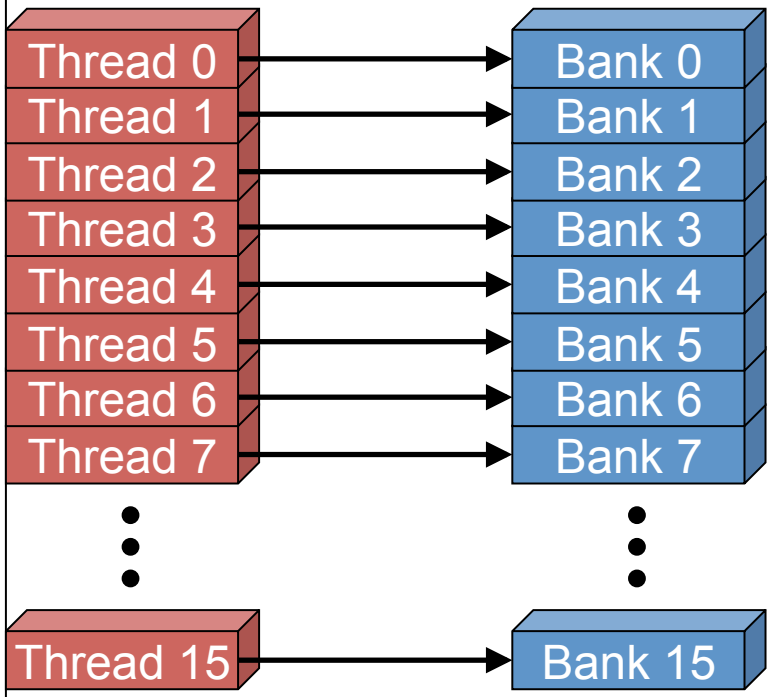
Parallel Memory Architecture

- In a parallel machine, many threads access memory
  - Therefore, memory is divided into banks
  - Essential to achieve high bandwidth
- Each bank can service one address per cycle
  - A memory can service as many simultaneous accesses as it has banks
- Multiple simultaneous accesses to a bank result in a bank conflict
  - Conflicting accesses are serialized

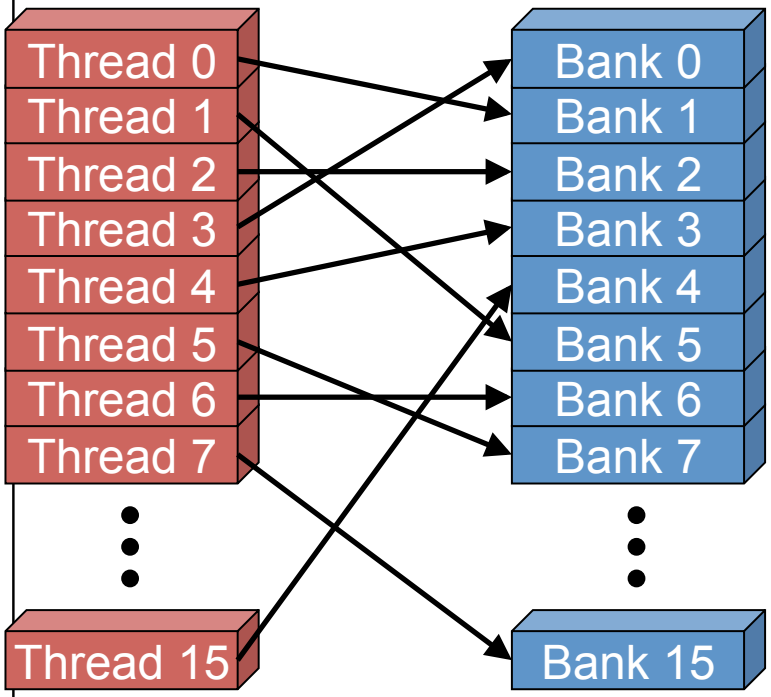


Bank Addressing Examples

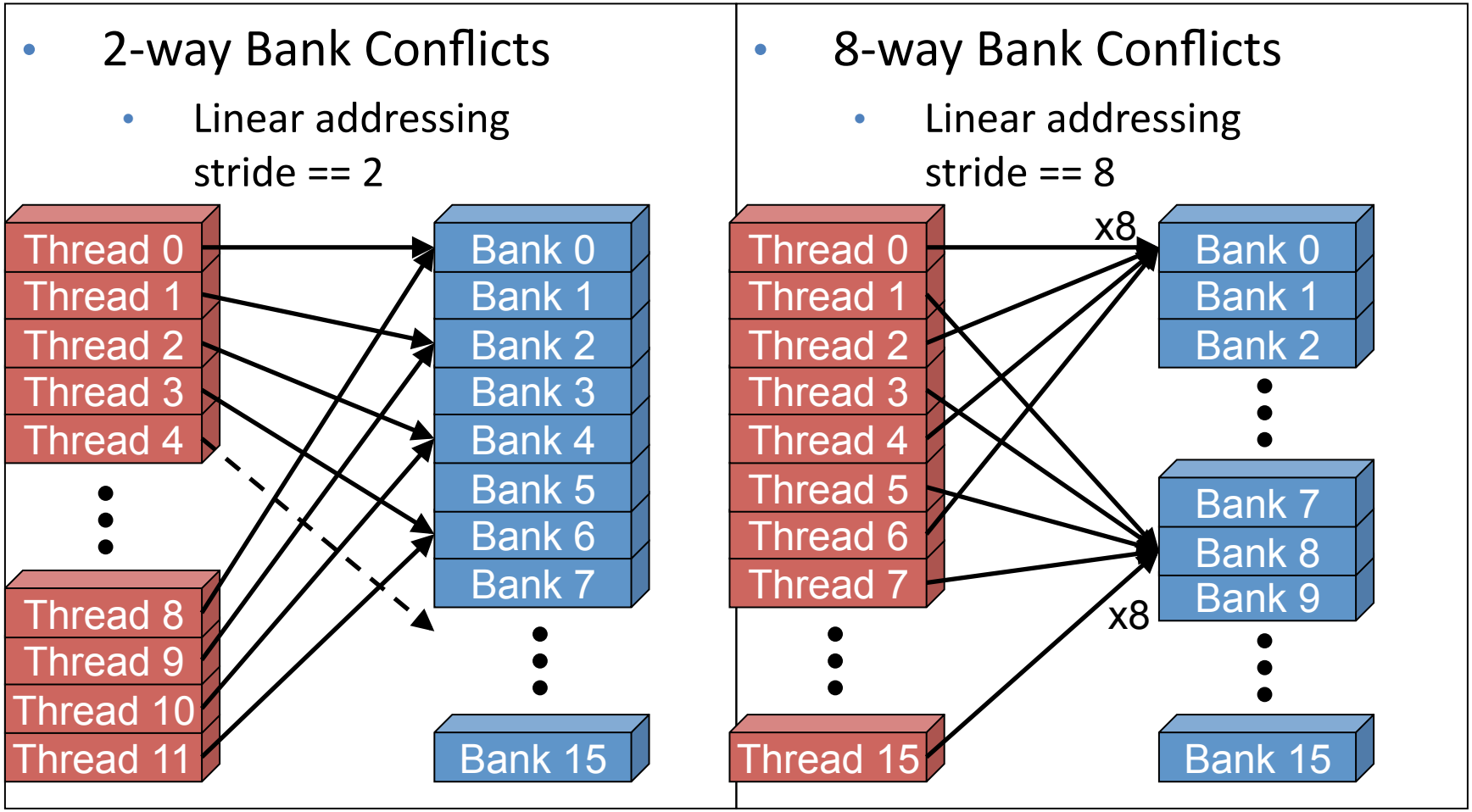
- No Bank Conflicts
  - Linear addressing stride == 1



- No Bank Conflicts
  - Random 1:1 Permutation



Bank Addressing Examples



How addresses map to banks on G80

- Each bank has a bandwidth of 32 bits per clock cycle
- Successive 32-bit words are assigned to successive banks
- G80 has 16 banks
  - So bank = address % 16
  - Same as the size of a half-warp
    - No bank conflicts between different half-warps, only within a single half-warp

Shared memory bank conflicts

- Shared memory is as fast as registers if there are no bank conflicts

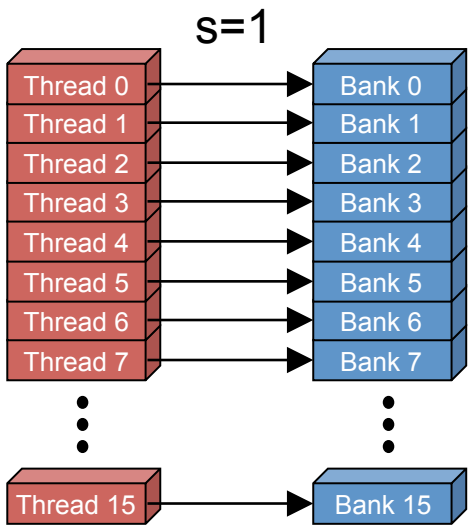
- The fast case:
  - If all threads of a half-warp access different banks, there is no bank conflict
  - If all threads of a half-warp access the identical address, there is no bank conflict (broadcast)
- The slow case:
  - Bank Conflict: multiple threads in the same half-warp access the same bank
  - Must serialize the accesses
  - Cost = max # of simultaneous accesses to a single bank

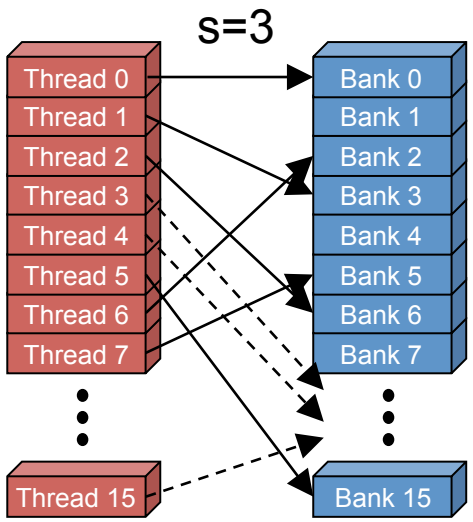
Linear Addressing

- Given:

```
__shared__ float shared[256];
float foo =
  shared[baseIndex + s *
   threadIdx.x];
```

- This is only bank-conflict-free if s shares no common factors with the number of banks
  - 16 on G80, so `s` must be odd





Control Flow Instructions

- Main performance concern with branching is divergence
  - Threads within a single warp take different paths
  - Different execution paths are serialized in G80
    - The control paths taken by the threads in a warp are traversed one at a time until there is no more.
- A common case: avoid divergence when branch condition is a function of thread ID
  - Example with divergence:
    - `If (threadIdx.x > 2) { }`
    - This creates two different control paths for threads in a block
    - Branch granularity < warp size; threads 0, 1 and 2 follow different path than the rest of the threads in the first warp
  - Example without divergence:
    - `If (threadIdx.x / WARP_SIZE > 2) { }`
    - Also creates two different control paths for threads in a block
    - Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path