CSE 591/CSE 392: GPU Programming

Threads

Klaus Mueller

Computer Science Department

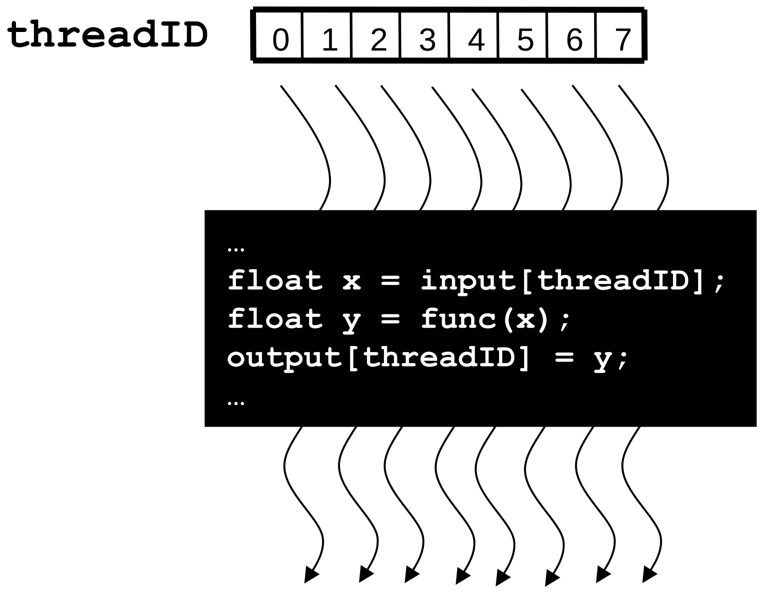
Stony Brook University

CUDA Threads

- 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

Arrays of Parallel Threads

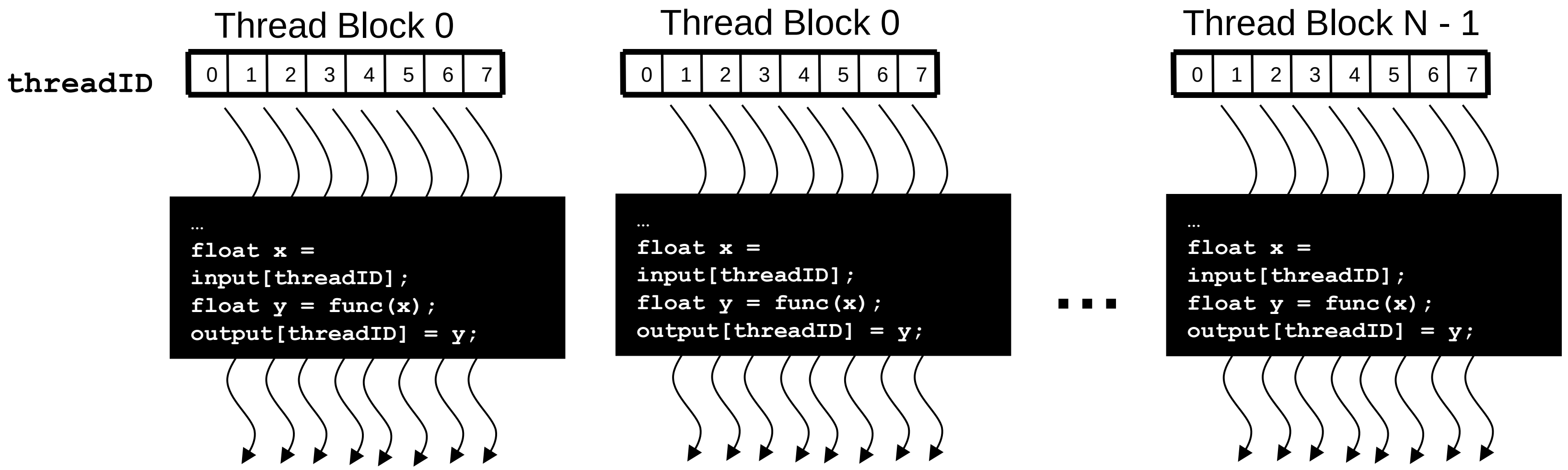
- A CUDA kernel is executed by an array of threads
  - All threads run the same code (SPMD)
  - Each thread has an ID that it uses to compute memory addresses and make control decisions



```
…
float x = input[threadID];
float y = func(x);
output[threadID] = y;
…
```

Thread Blocks: Scalable Cooperation

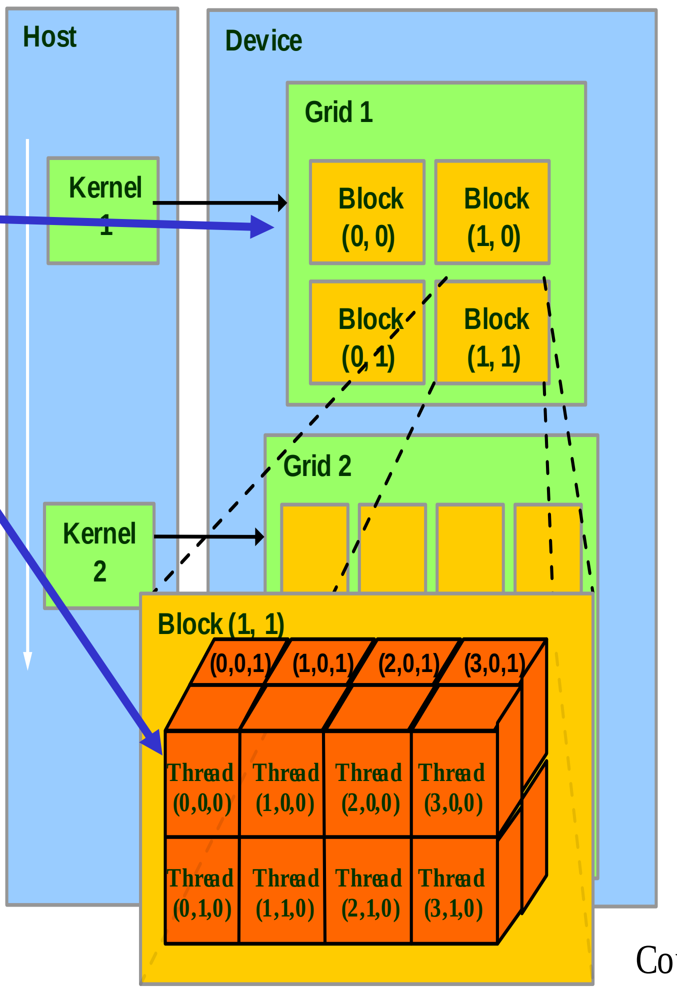
- Divide monolithic thread array into multiple blocks
  - Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization**
  - Threads in different blocks cannot cooperate



```
…
float x =
input[threadID];
float y = func(x);
output[threadID] = y;
```

Block IDs and Thread IDs

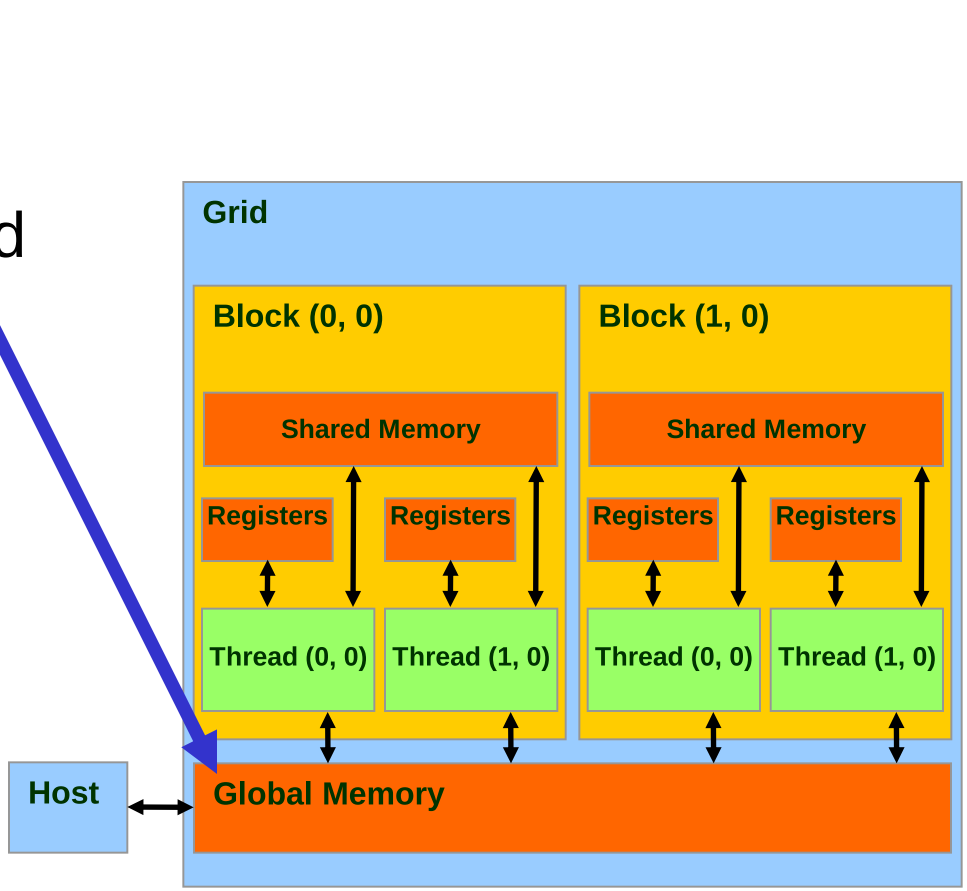
- Each thread uses IDs to decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



Courtesy: NDVIA

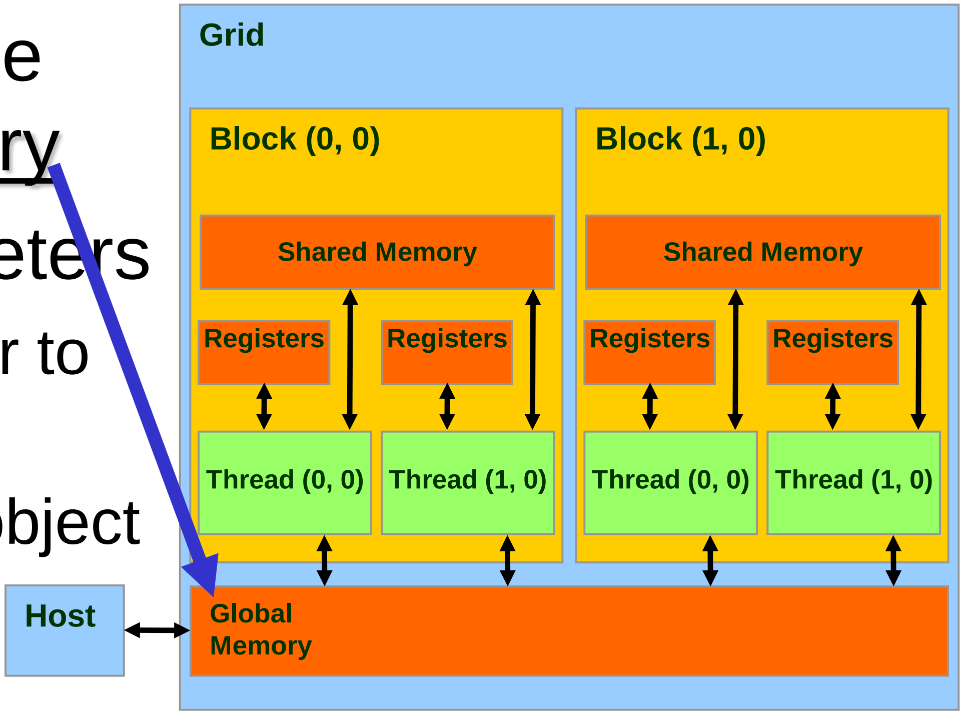
CUDA Memory Model Overview

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
  - Long latency access
- We will focus on global memory for now
  - Constant and texture memory will come later



CUDA Device Memory Allocation

- cudaMalloc()
  - Allocates object in the device Global Memory
  - Requires two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** of allocated object
- cudaFree()
  - Frees object from device Global Memory
    - Pointer to freed object



CUDA Device Memory Allocation (cont.)

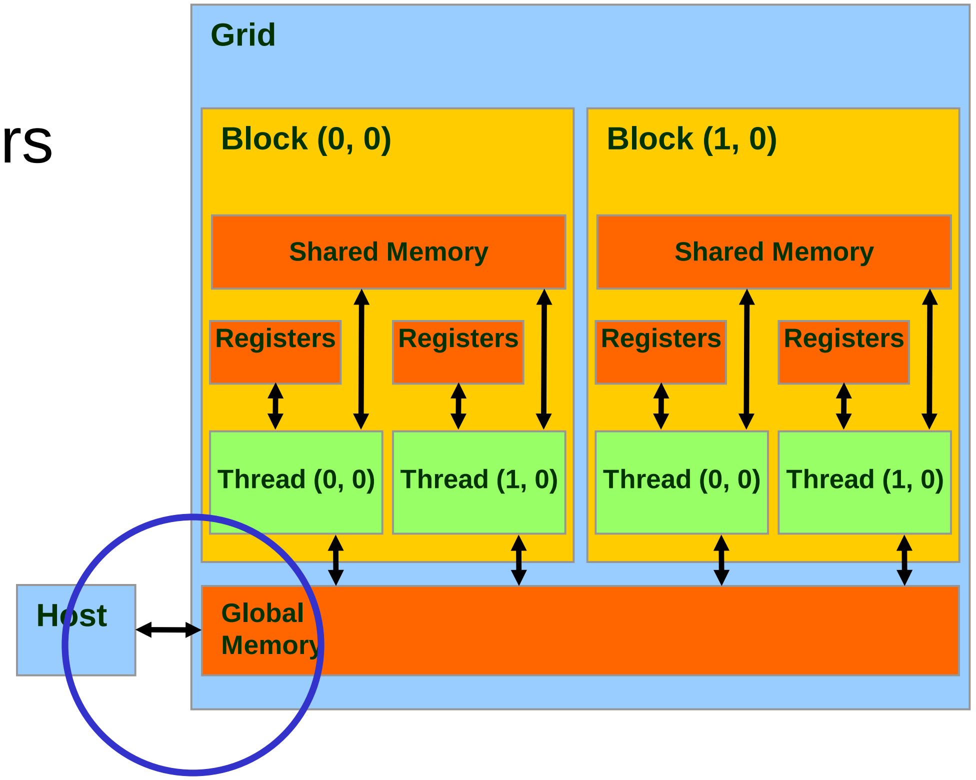
- Code example:
  - Allocate a 64 * 64 single precision float array
  - Attach the allocated storage to Md
  - “d” is often used to indicate a device data structure

```
TILE_WIDTH = 64;
Float* Md
int size = TILE_WIDTH * TILE_WIDTH * sizeof(float);
```

```
cudaMalloc((void**)&Md, size);
cudaFree(Md);
```

CUDA Host-Device Data Transfer

- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
- Asynchronous transfer



CUDA Host-Device Data Transfer (cont.)

- Code example:
  - Transfer a 64 * 64 single precision float array
  - M is in host memory and Md is in device memory
  - cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost are symbolic constants

cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

cudaMemcpy(M, Md, size, cudaMemcpyDeviceToHost);

CUDA Keywords

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

Calling a Kernel Function – Thread Creation

- A kernel function must be called with an execution configuration:

```
__global__ void KernelFunc(...);
dim3   DimGrid(100, 50);    // 5000 thread blocks
dim3   DimBlock(4, 8, 8);   // 256 threads per
   block
size_t SharedMemBytes = 64; // 64 bytes of shared
   memory
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes
   >>>(...);
```

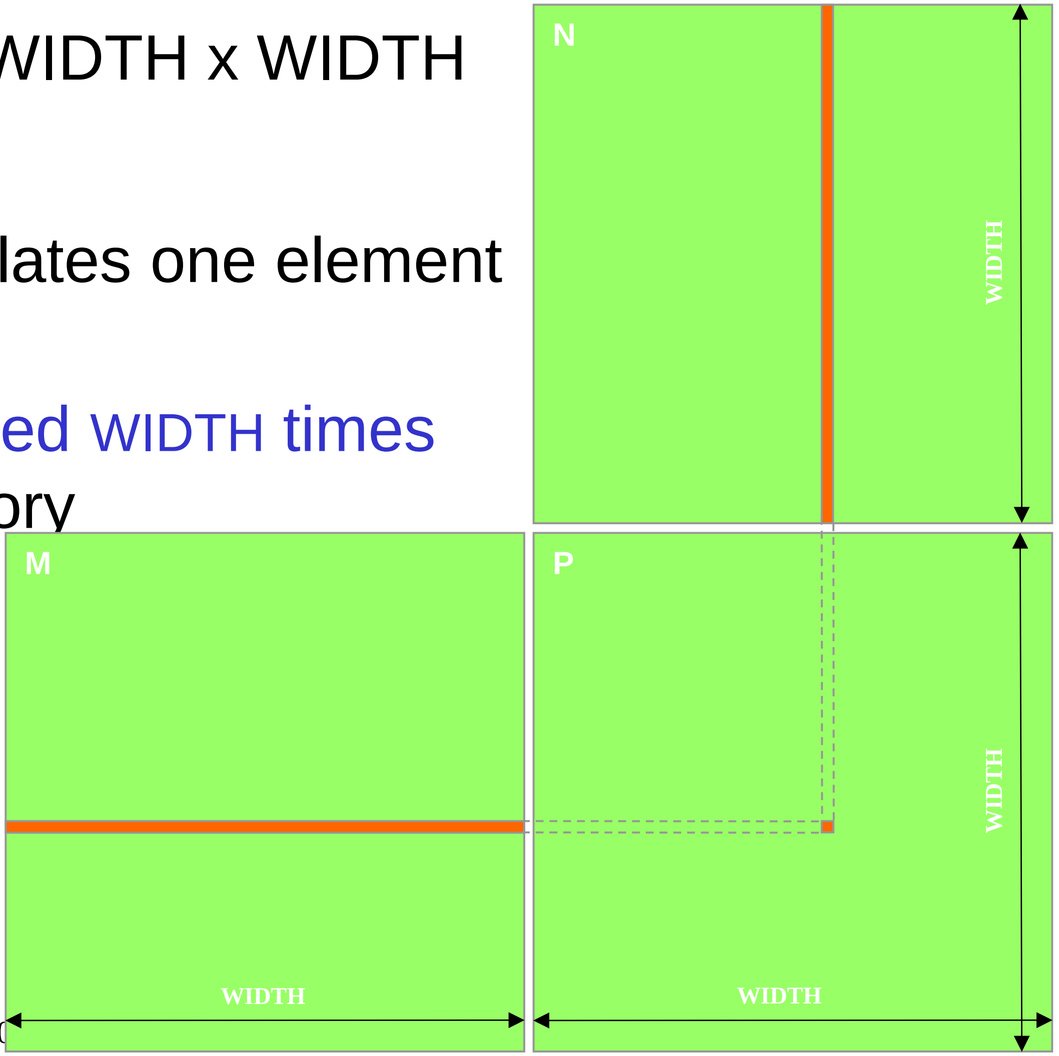
- Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit synch needed for blocking

A Simple Running Example Matrix Multiplication

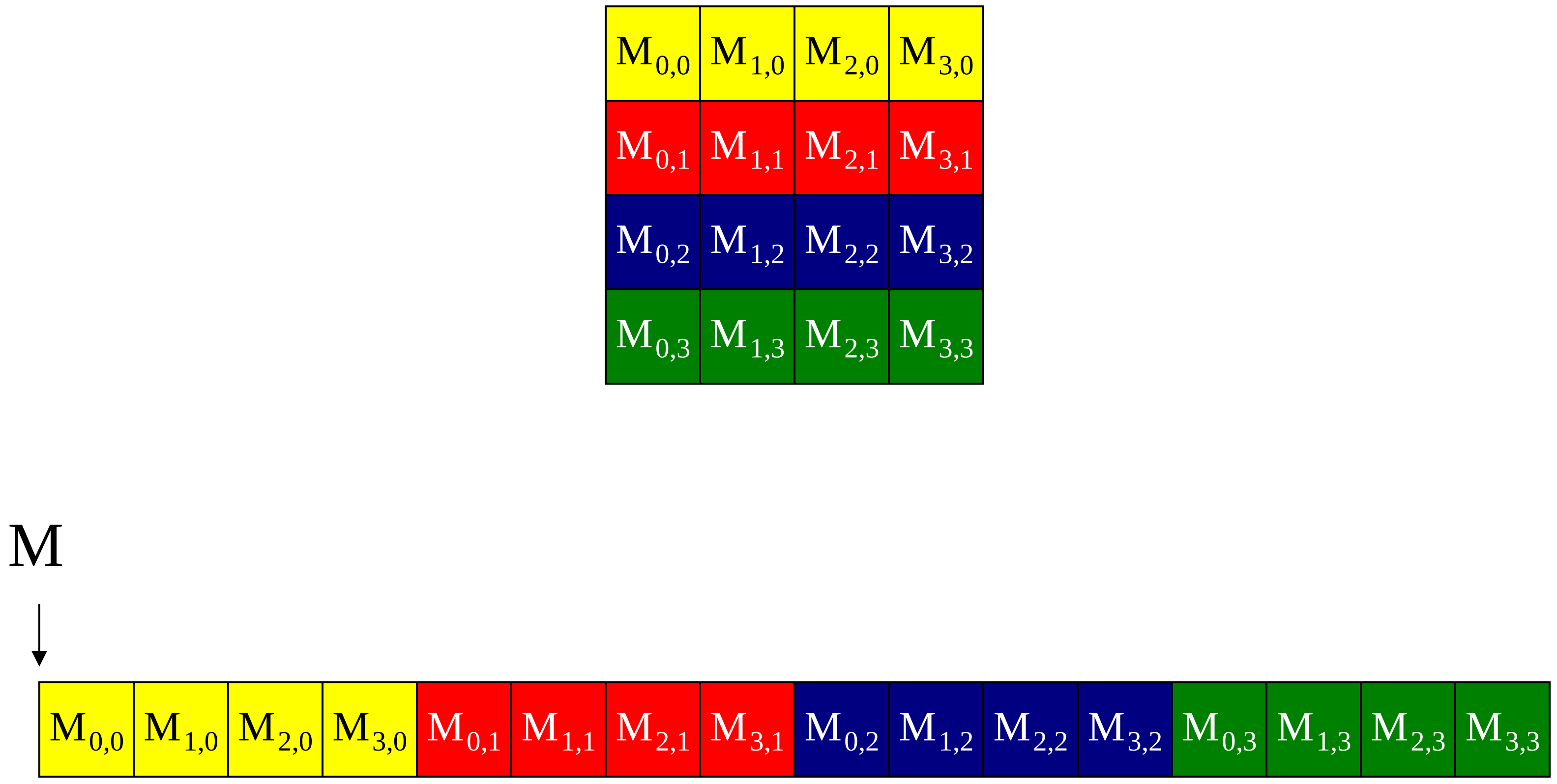
- A simple matrix multiplication example that illustrates the basic features of memory and thread management in CUDA programs
  - Leave shared memory usage until later
  - Local, register usage
  - Thread ID usage
  - Memory data transfer API between host and device
  - Assume square matrix for simplicity

Programming Model: Square Matrix Multiplication Example

- P = M * N of size WIDTH x WIDTH
- Without tiling:
  - One thread calculates one element of P
  - M and N are loaded WIDTH times from global memory

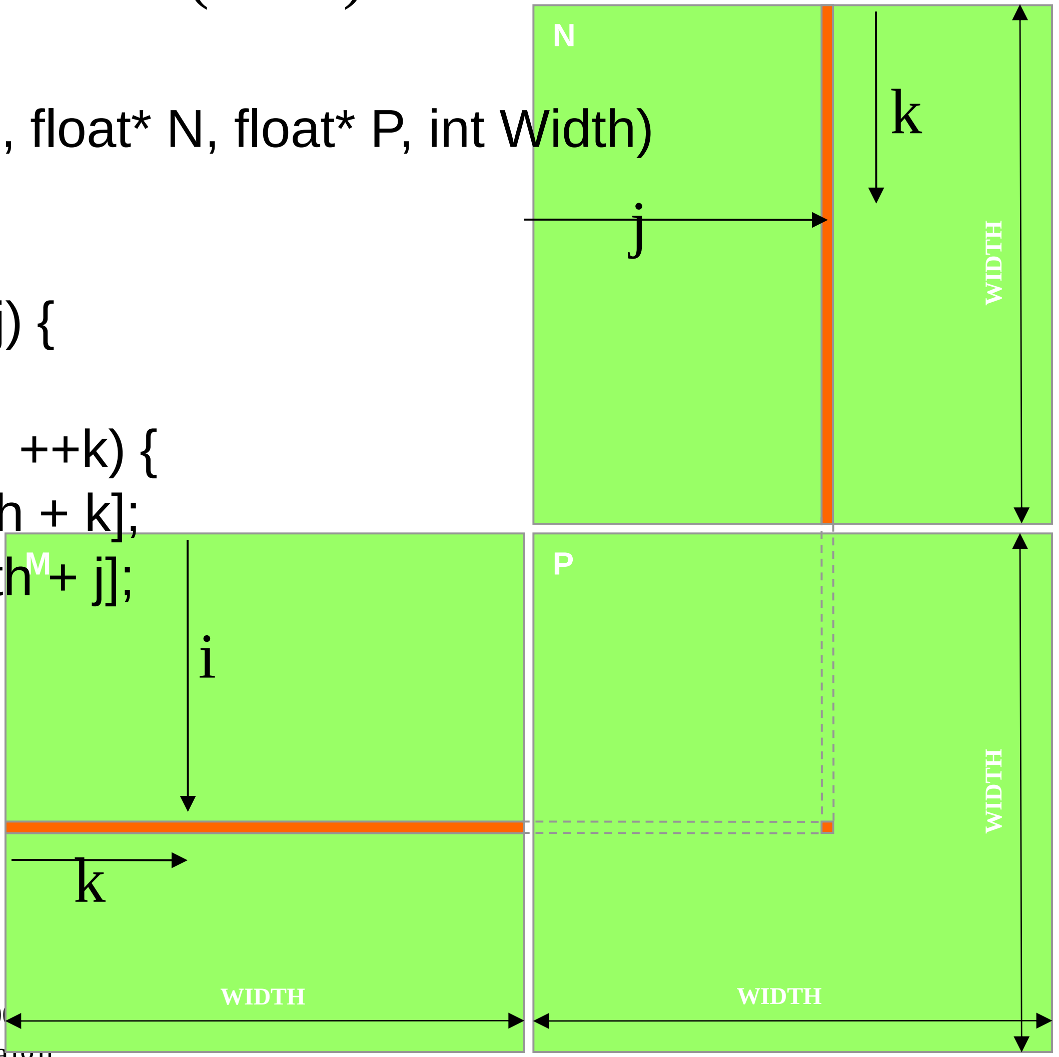


Memory Layout of a Matrix in C


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

Step 1: Matrix Multiplication A Simple Host Version in C

```
// Matrix multiplication on the (CPU) host in double precision
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; ++i)
        for (int j = 0; j < Width; ++j) {
            double sum = 0;
            for (int k = 0; k < Width; ++k) {
                double a = M[i * width + k];
                double b = N[k * width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
```



Step 2: Input Matrix Data Transfer (Host-side Code)

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
   int size = Width * Width * sizeof(float);
   float* Md, Nd, Pd;
   …
1. // Allocate and Load M, N to device memory
    cudaMalloc(&Md, size);
    cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

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

     // Allocate P on the device
    cudaMalloc(&Pd, size);
```

Step 3: Output Matrix Data Transfer (Host-side Code)

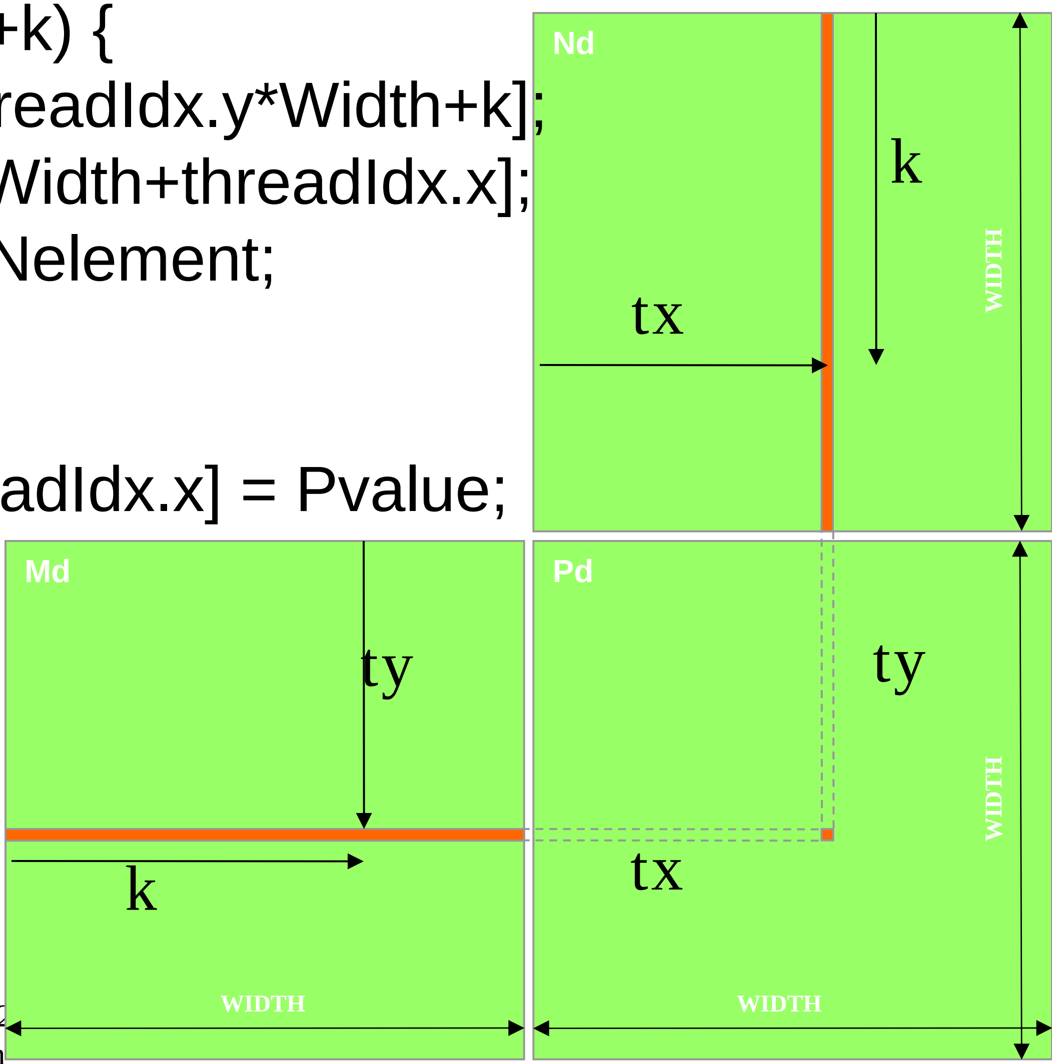
```
2.  //  Kernel invocation code – to be shown later
    …

3.  //  Read P from the device
    cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);

    //  Free device matrices
    cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
   }
```

Step 4: Kernel Function

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

__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{

    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;
```

Step 4: Kernel Function (cont.)

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

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

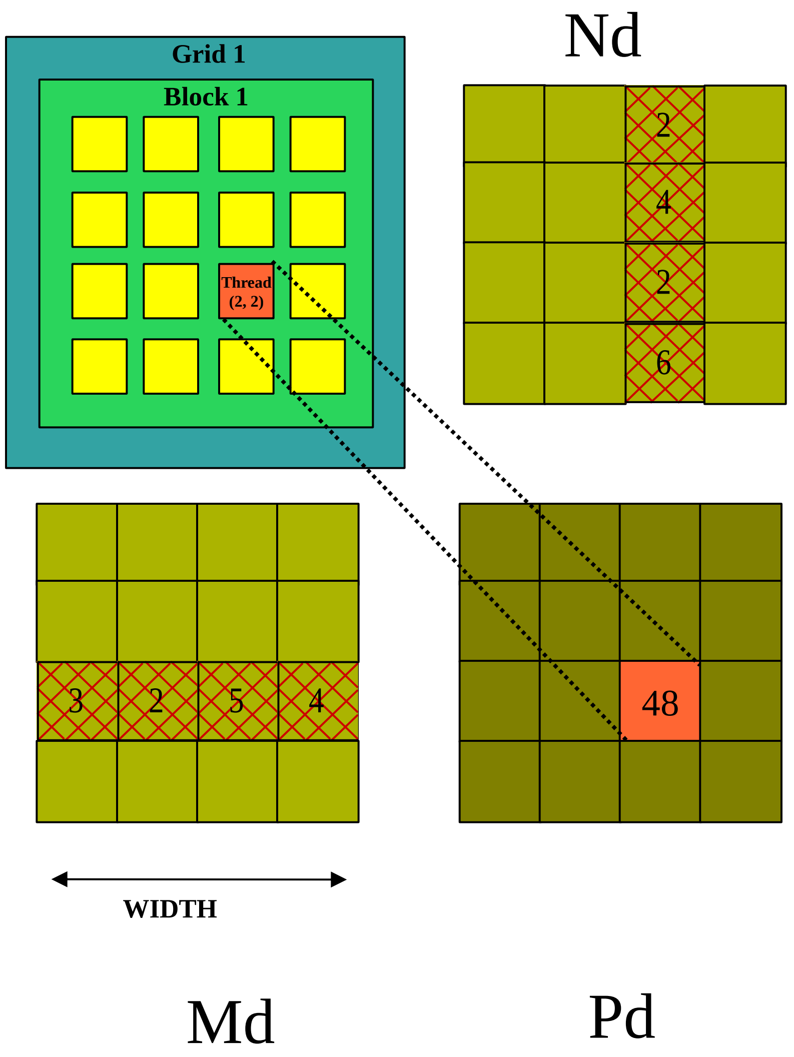


Step 5: Kernel Invocation (Host-side Code)

```
// Setup the execution configuration
   dim3 dimGrid(1, 1);
   dim3 dimBlock(Width, Width);

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

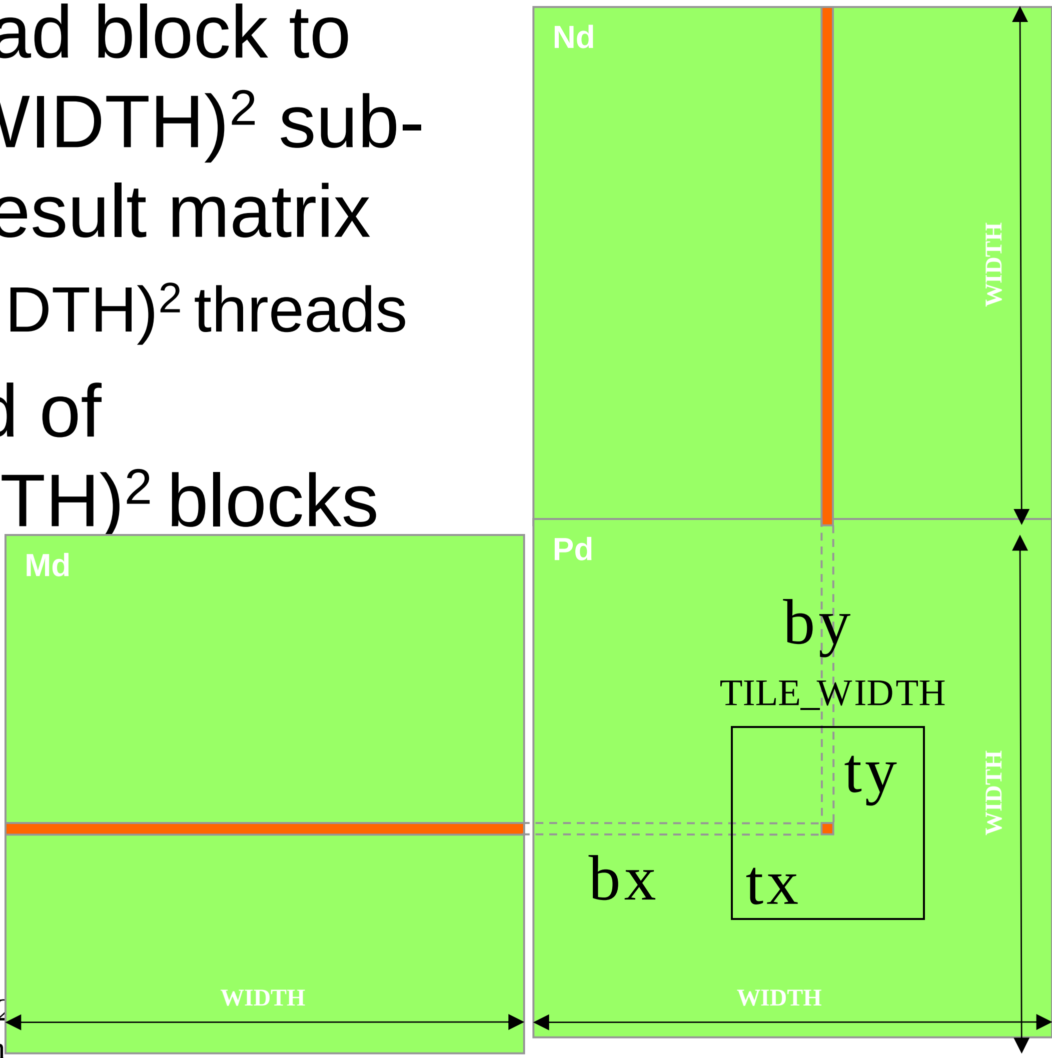
Only One Thread Block Used

- One Block of threads computes matrix Pd
  - Each thread computes one element of Pd
- Each thread
  - Loads a row of matrix Md
  - Loads a column of matrix Nd
  - Performs 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



Next: Handling Arbitrary Sized Square Matrices

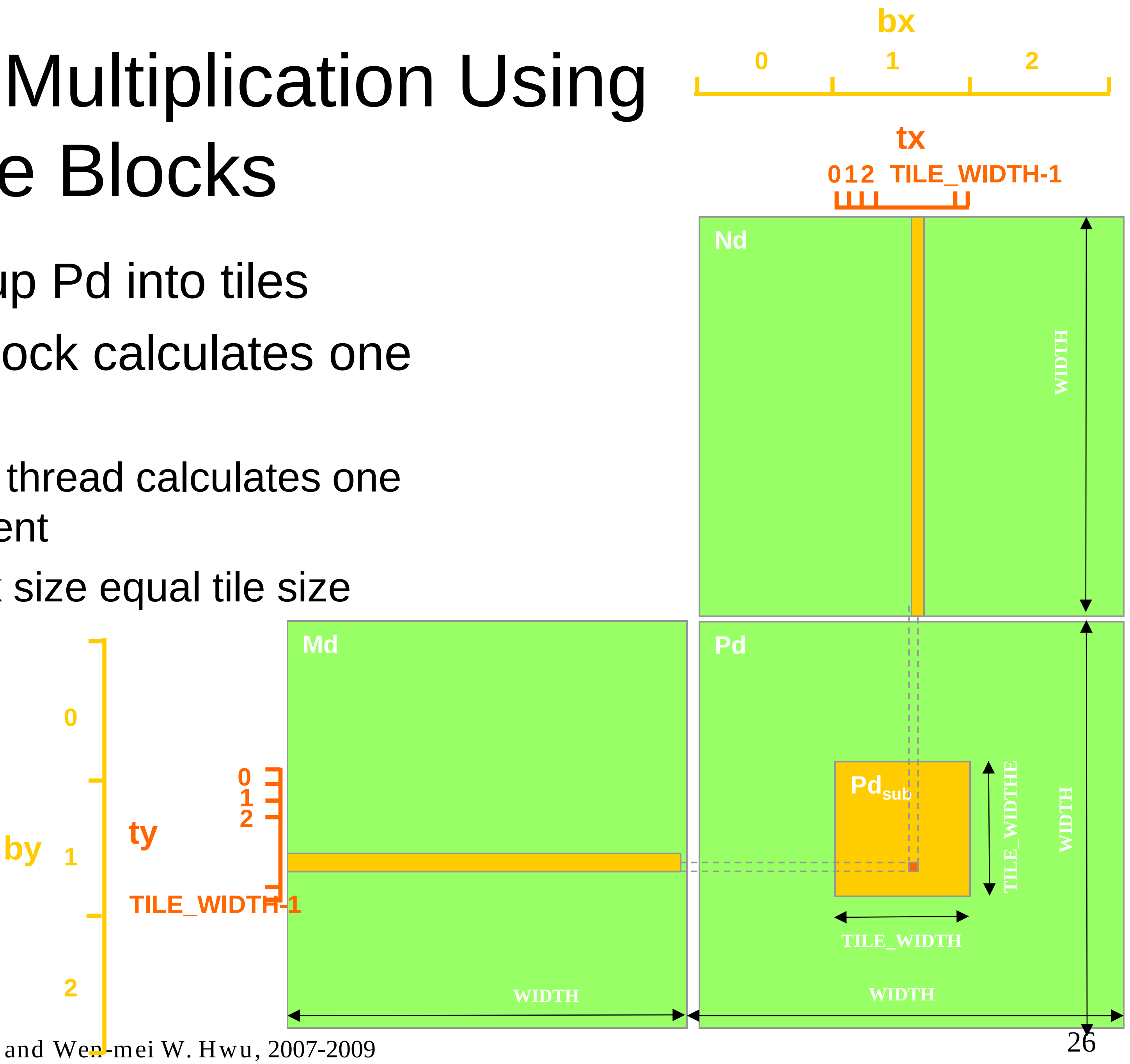
- Have each 2D thread block to compute a (TILE_WIDTH)^2 sub-matrix (tile) of the result matrix
  - Each has (TILE_WIDTH)^2 threads
- Generate a 2D Grid of (WIDTH/TILE_WIDTH)^2 blocks



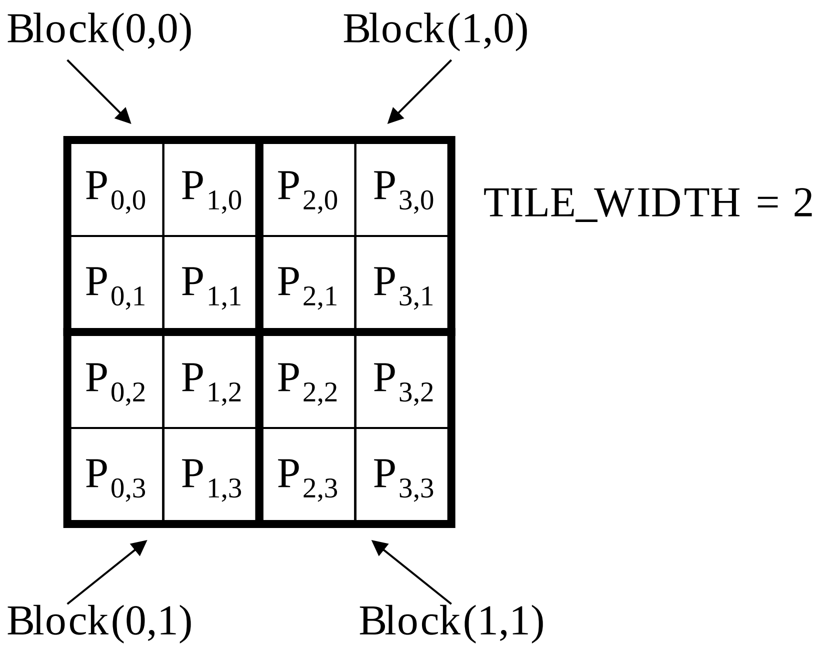
You still need to put a loop around the kernel call for cases where WIDTH/ TILE_WIDTH is greater than max grid size (64K)!

Matrix Multiplication Using Multiple Blocks

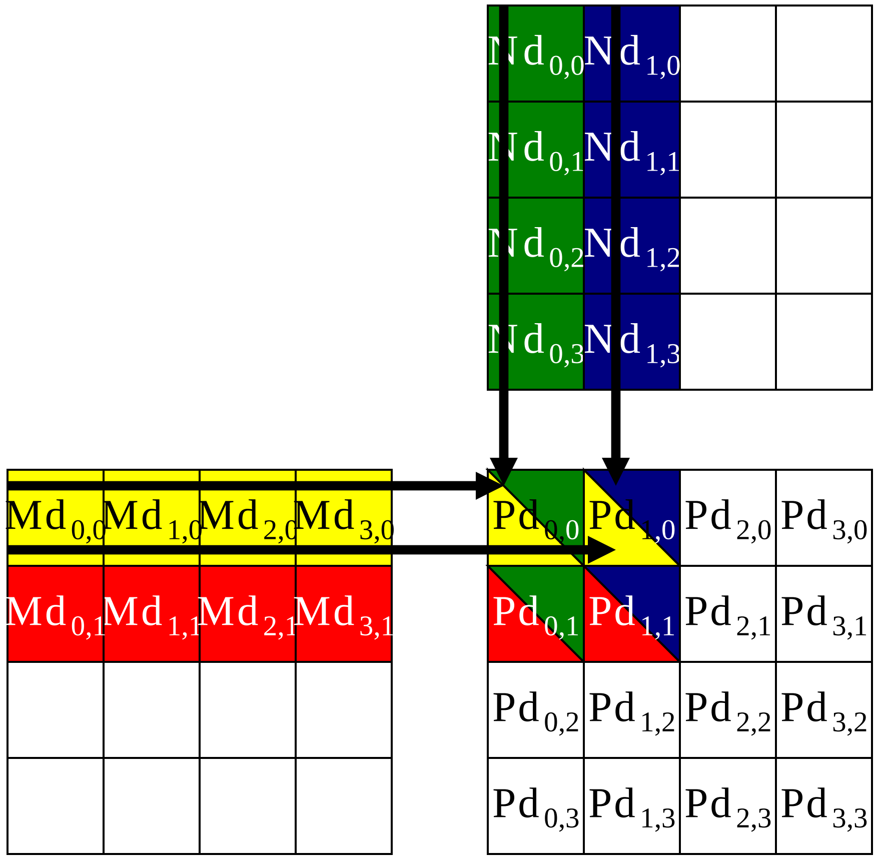
- Break-up Pd into tiles
- Each block calculates one tile
  - Each thread calculates one element
  - Block size equal tile size



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



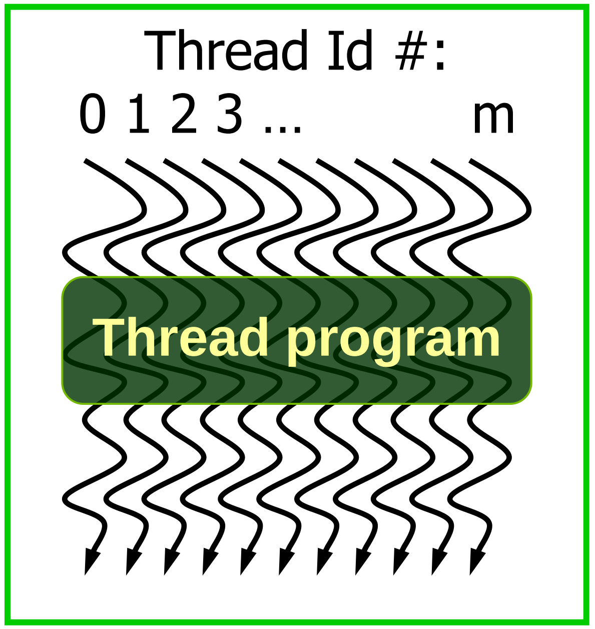
Revised Matrix Multiplication Kernel using Multiple Blocks

```
__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 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;
}
```

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!

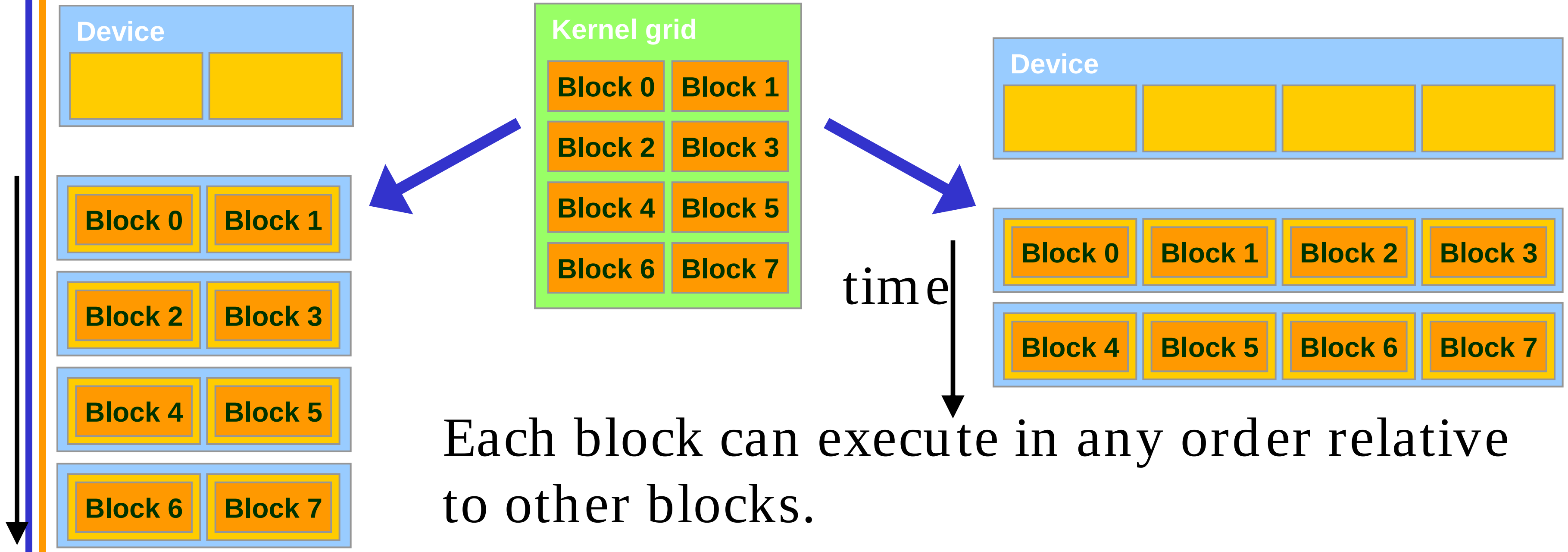
CUDA Thread Block



Courtesy: John Nickolls, NVIDIA

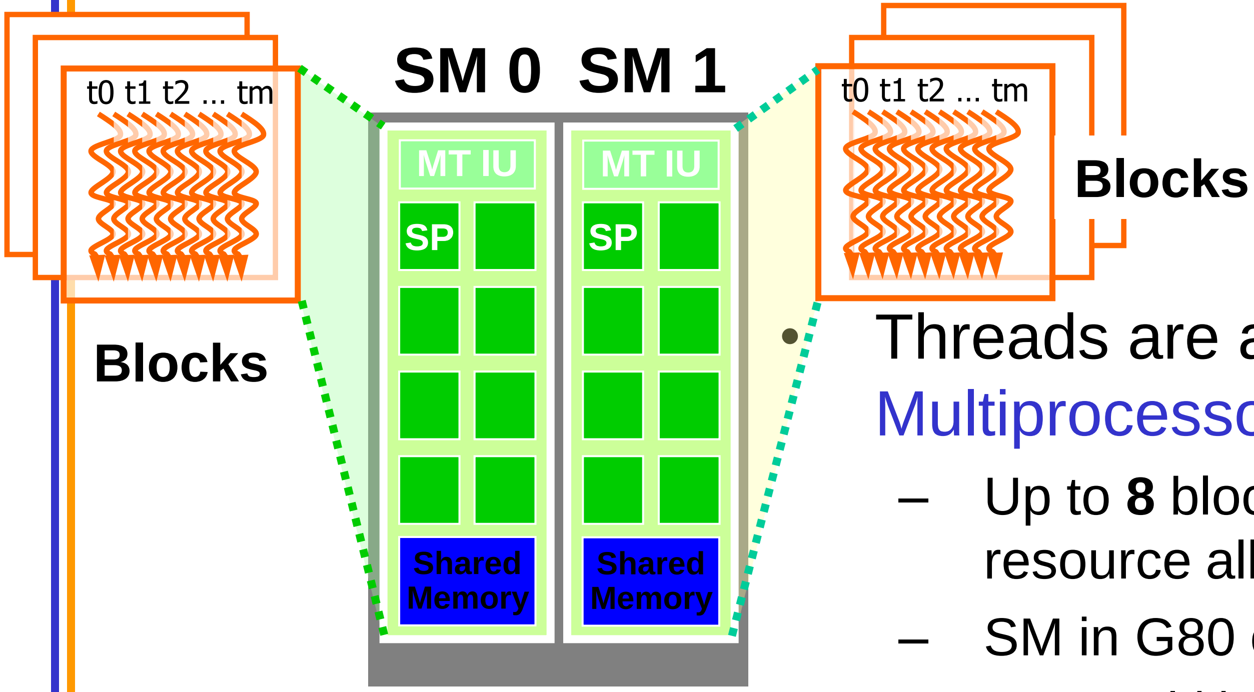
Transparent Scalability

- Hardware is free to assign blocks to any processor at any time, given the resources
  - A kernel scales across any number of parallel processors
  - When less resources are available, hardware will reduce the number of blocks run in parallel (compare right with left block assignment below)



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

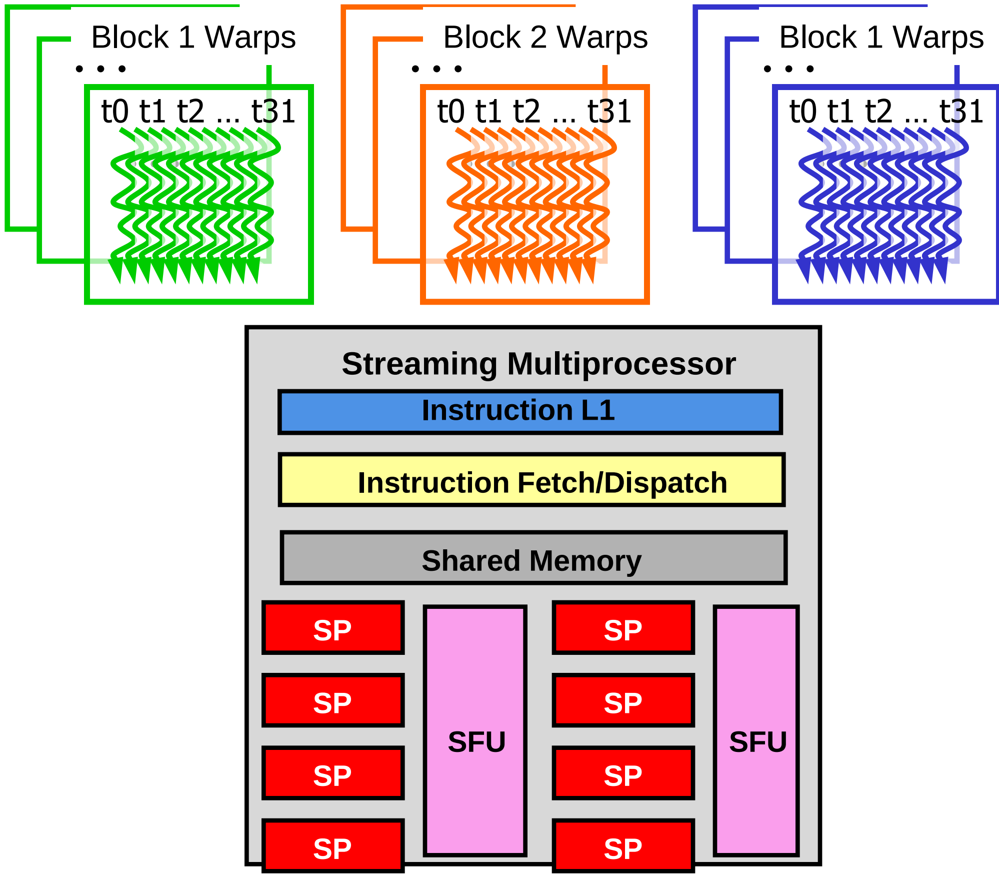
G80 Example: Executing Thread Blocks



- 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

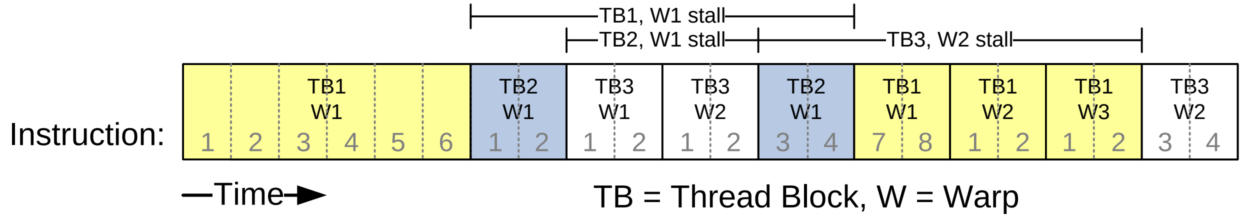
G80 Example: Thread Scheduling

- 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
  - Effectively provides for *latency hiding* (memory waits, etc.)
  - At any time, only one of the warps is executed by SM
  - 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! This will lead to under-utilization (bad for latency hiding).
  - 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!

Number of Threads

- All threads on an SM must run in lock-step
  - if one thread is delayed because of memory load then all threads in a warp must wait
  - a new warp is scheduled
  - so it is good to have more threads per block
  - however, there is a limit on the number of threads per SM
  - 768, 1024, 1536, 2048 depending on compute capability
  - this is a function of the maximum number of warps (24, 32, 48, 64) of 32 threads each

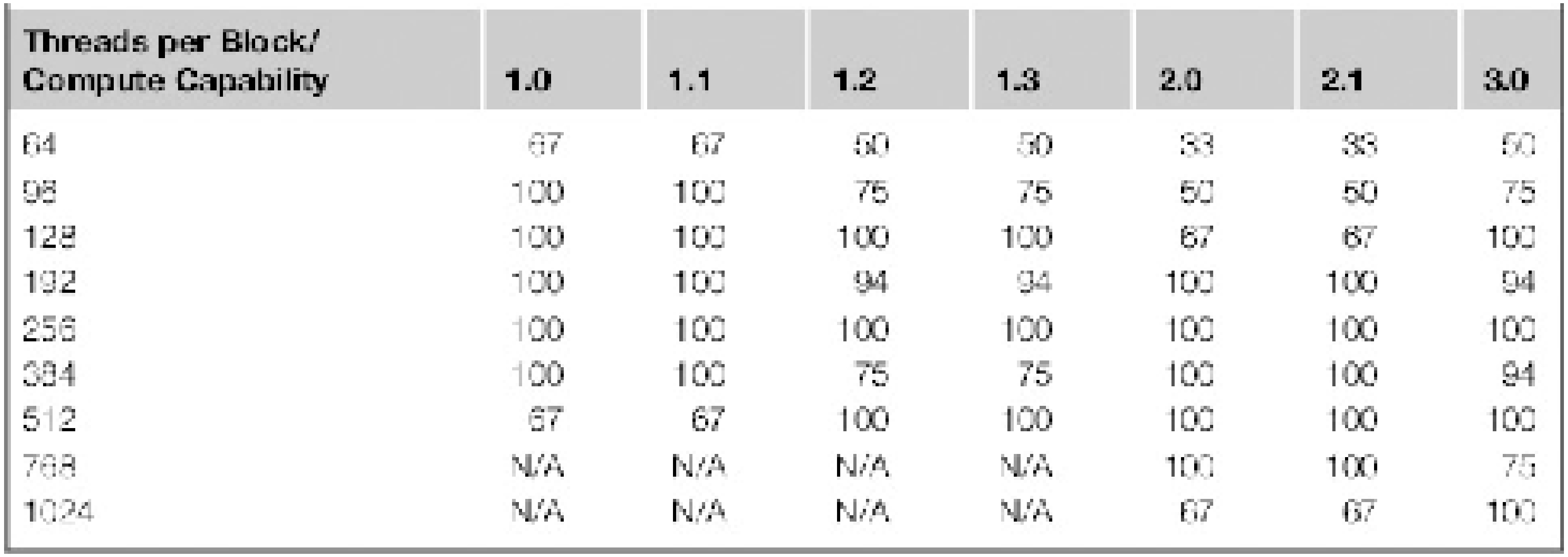
Number of Blocks

- Workload of threads is not always uniform
  - all threads must complete before a new block can be scheduled
  - if the slow thread is part of a large block the idle time is high
  - so it is better to have smaller blocks
  - however, there is a limit on the number of blocks per SM (8 or less, depending on compute capability)

GPU Utilization

- Goal is to allocate as many threads per SM as maximum limit
- Here take into account:
  - max number of blocks
  - modulus of warps
- So to achieve 100 % utilization depends on compute capability and threads/block

GPU Utilization



Threads per Block/ Compute Capability	1.0	1.1	1.2	1.3	2.0	2.1	3.0
64	67	67	50	50	33	33	50
96	100	100	75	75	50	50	75
128	100	100	100	100	67	67	100
192	100	100	94	94	100	100	94
256	100	100	100	100	100	100	100
384	100	100	75	75	100	100	94
512	67	67	100	100	100	100	100
768	N/A	N/A	N/A	N/A	100	100	75
1024	N/A	N/A	N/A	N/A	67	67	100

- So 256 threads per block is safest across all compute capabilities

Shane Cook “CUDA Programming”

Practical Example

- Histogram computation

```
for (unsigned int i=0; i< max; i++)
{
        bin[array[i]]++;
}
```

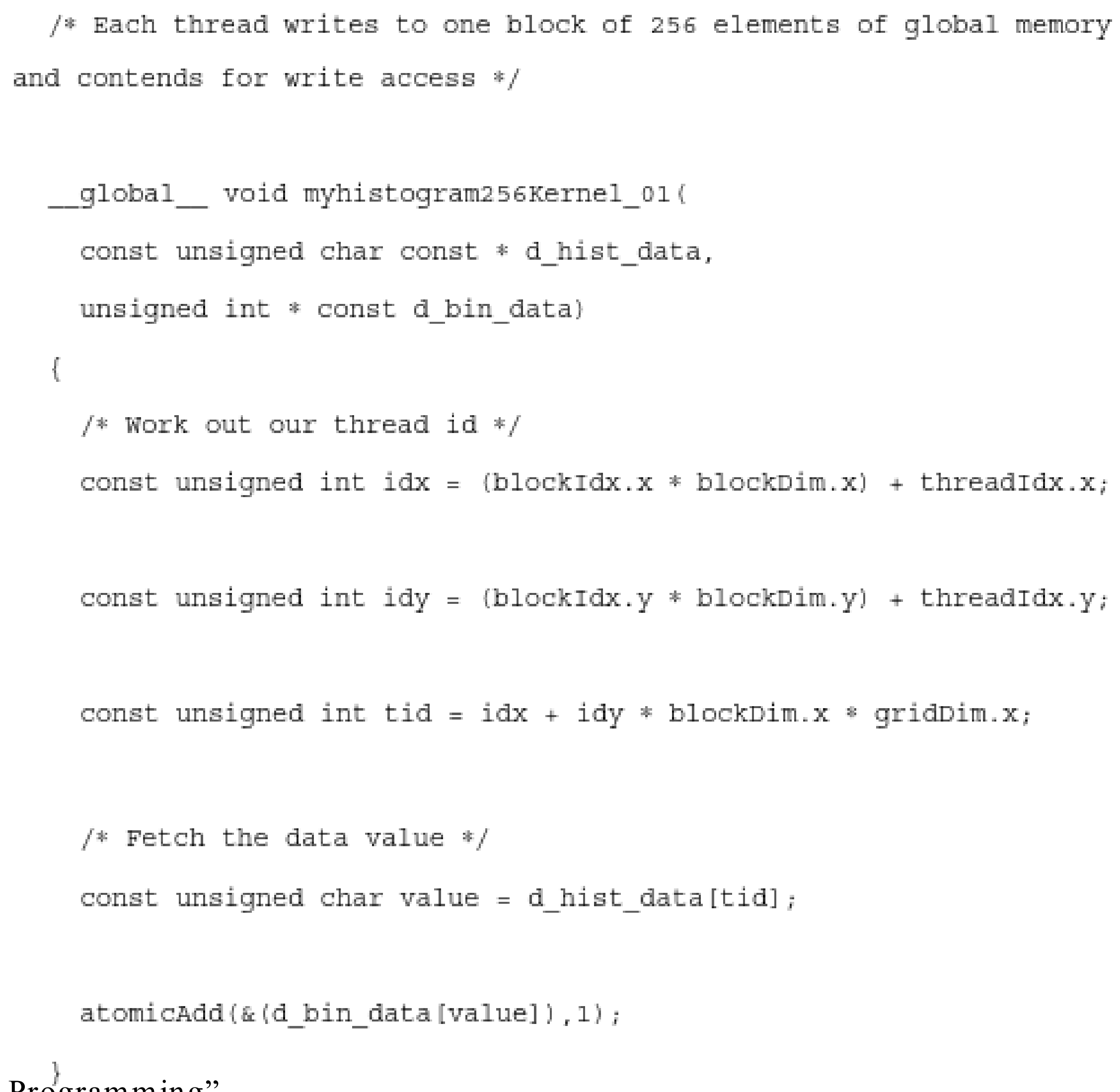
CPU Algorithm

1. Read the value from the array into a register
2. Work out the base address and offset to the correct bin element
3. Fetch the existing bin value
4. Increment the bin value by one
5. Write the new bin value back to the bin in memory

Steps 3, 4, 5 are not atomic

- OK for CPU since serial but not for GPU
- use *atomicAdd(&value)* on the GPU

GPU Algorithm 1

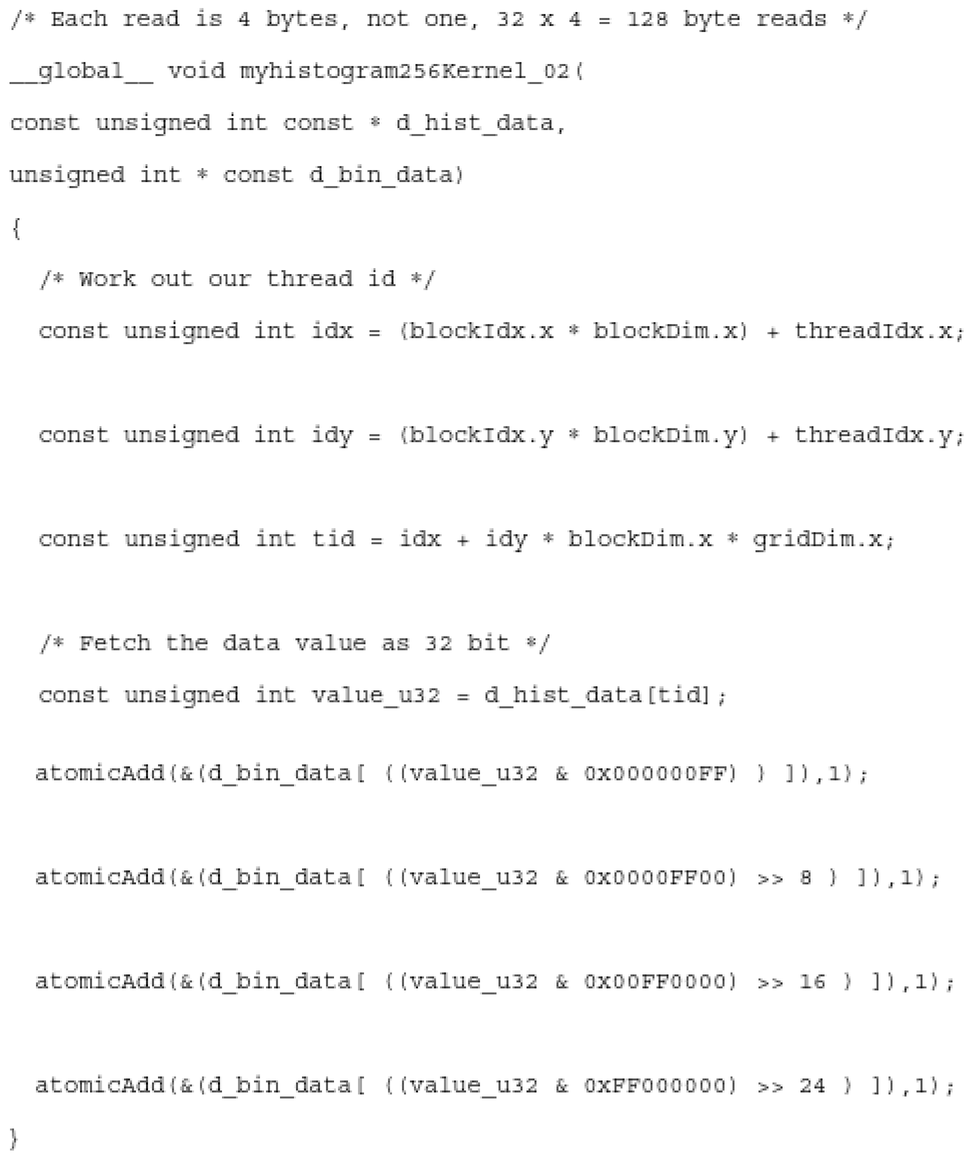


```
  /* Each thread writes to one block of 256 elements of global memory
and contends for write access */

  __global__ void myhistogram256Kernel_01(
    const unsigned char const * d_hist_data,
    unsigned int * const d_bin_data)
  {
    /* Work out our thread id */
    const unsigned int idx = (blockIdx.x * blockDim.x) + threadIdx.x;

    const unsigned int idy = (blockIdx.y * blockDim.y) + threadIdx.y;

    const unsigned int tid = idx + idy * blockDim.x * gridDim.x;

    /* Fetch the data value */
    const unsigned char value = d_hist_data[tid];

    atomicAdd(&(d_bin_data[value]),1);
  }
```

GPU Algorithm 1

- Not overly fast
- Why?
  - each thread only fetches 1 byte
  - half warp fetches 16 bytes
  - maximal supported size is 128 bytes
  - hence memory bandwidth is heavily underused

GPU Algorithm 2



```
/* Each read is 4 bytes, not one, 32 x 4 = 128 byte reads */
__global__ void myhistogram256Kernel_02(
const unsigned int const * d_hist_data,
unsigned int * const d_bin_data)
{
  /* Work out our thread id */
  const unsigned int idx = (blockIdx.x * blockDim.x) + threadIdx.x;

  const unsigned int idy = (blockIdx.y * blockDim.y) + threadIdx.y;

  const unsigned int tid = idx + idy * blockDim.x * gridDim.x;

  /* Fetch the data value as 32 bit */
  const unsigned int value_u32 = d_hist_data[tid];
  atomicAdd(&(d_bin_data[ ((value_u32 & 0x000000FF) ) ]),1);

  atomicAdd(&(d_bin_data[ ((value_u32 & 0x0000FF00) >> 8 ) ]),1);

  atomicAdd(&(d_bin_data[ ((value_u32 & 0x00FF0000) >> 16 ) ]),1);

  atomicAdd(&(d_bin_data[ ((value_u32 & 0xFF000000) >> 24 ) ]),1);
}
```

GPU Algorithm 2

- In fact, achieves zero speedup
  - no improvements in memory bandwidth
  - advanced compute capability already does good coalescing
  - need to look for other culprit

GPU Algorithm 2

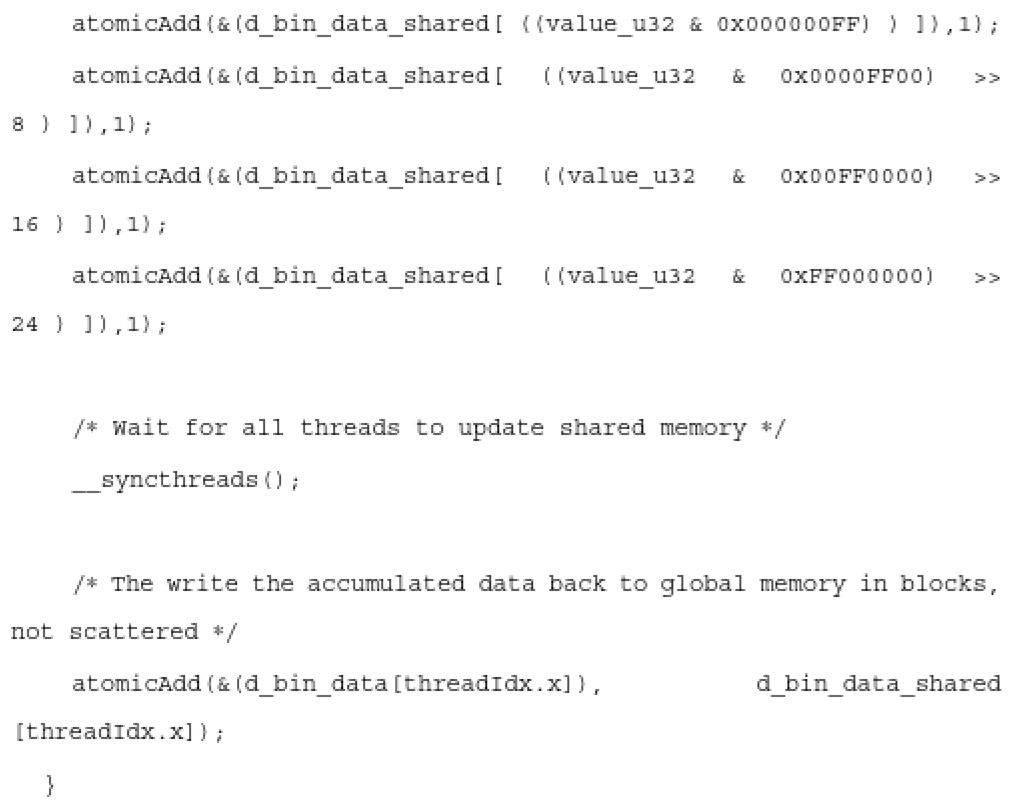
- In fact, achieves zero speedup
  - no improvements in memory bandwidth
  - advanced compute capability already does good coalescing
  - need to look for other culprit

  - maybe reduce the amount of global number of atomic writes?

GPU Algorithm 3

```
__shared__ unsigned int d_bin_data_shared[256];

/* Each read is 4 bytes, not one, 32 x 4 = 128 byte reads */
__global__ void myhistogram256Kernel_03(
const unsigned int const * d_hist_data,
unsigned int * const d_bin_data)
{
  /* Work out our thread id */
  const unsigned int idx = (blockIdx.x * blockDim.x) + threadIdx.x;
  const unsigned int idy = (blockIdx.y * blockDim.y) + threadIdx.y;
  const unsigned int tid = idx + idy * blockDim.x * gridDim.x;

  /* Clear shared memory */
  d_bin_data_shared[threadIdx.x] = 0;

  /* Fetch the data value as 32 bit */
  const unsigned int value_u32 = d_hist_data[tid];

  /* Wait for all threads to update shared memory */
  __syncthreads();
```

GPU Algorithm 3



```
    atomicAdd(&(d_bin_data_shared[ ((value_u32 & 0x000000FF) ) ]),1);
    atomicAdd(&(d_bin_data_shared[  ((value_u32  &  0x0000FF00)  >>
8 ) ]),1);
    atomicAdd(&(d_bin_data_shared[  ((value_u32  &  0x00FF0000)  >>
16 ) ]),1);
    atomicAdd(&(d_bin_data_shared[  ((value_u32  &  0xFF000000)  >>
24 ) ]),1);

    /* Wait for all threads to update shared memory */
    __syncthreads();

    /* The write the accumulated data back to global memory in blocks,
not scattered */
    atomicAdd(&(d_bin_data[threadIdx.x]),          d_bin_data_shared
[threadIdx.x]);
  }
```

GPU Algorithm 3

- results in a 6 fold speedup
- but could still reduce global memory traffic
- what can we do?

GPU Algorithm 3

- results in a 6 fold speedup
- but could still reduce global memory traffic
- what can we do?

- compute more than one histogram per thread

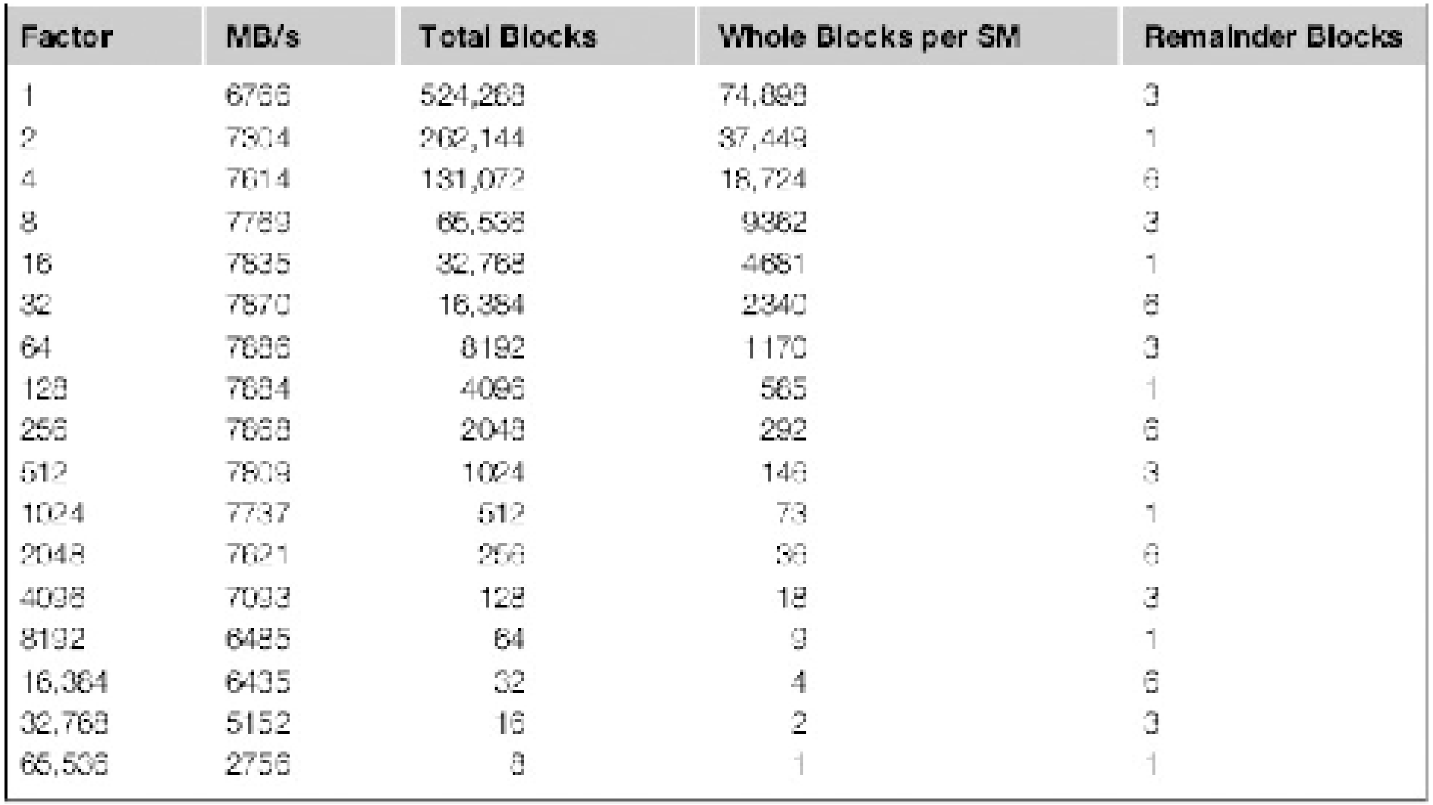
GPU Algorithm 3

```
  /* Each read is 4 bytes, not one, 32 x 4 = 128 byte reads */
  /* Accumulate into shared memory N times */
  __global__ void myhistogram256Kernel_07(const unsigned int const *
d_hist_data,
                                          unsigned   int   *   const
d_bin_data,
                                          unsigned int N)
  {
    /* Work out our thread id */
    const  unsigned  int  idx  =  (blockIdx.x  *  (blockDim.x*N)  )  +
threadIdx.x;
    const unsigned int idy = (blockIdx.y * blockDim.y ) + threadIdx.y;
    const unsigned int tid = idx + idy * (blockDim.x*N) * (gridDim.x);
    /* Clear shared memory */
    d_bin_data_shared[threadIdx.x] = 0;

    /* Wait for all threads to update shared memory */
    __syncthreads();
```

GPU Algorithm 3

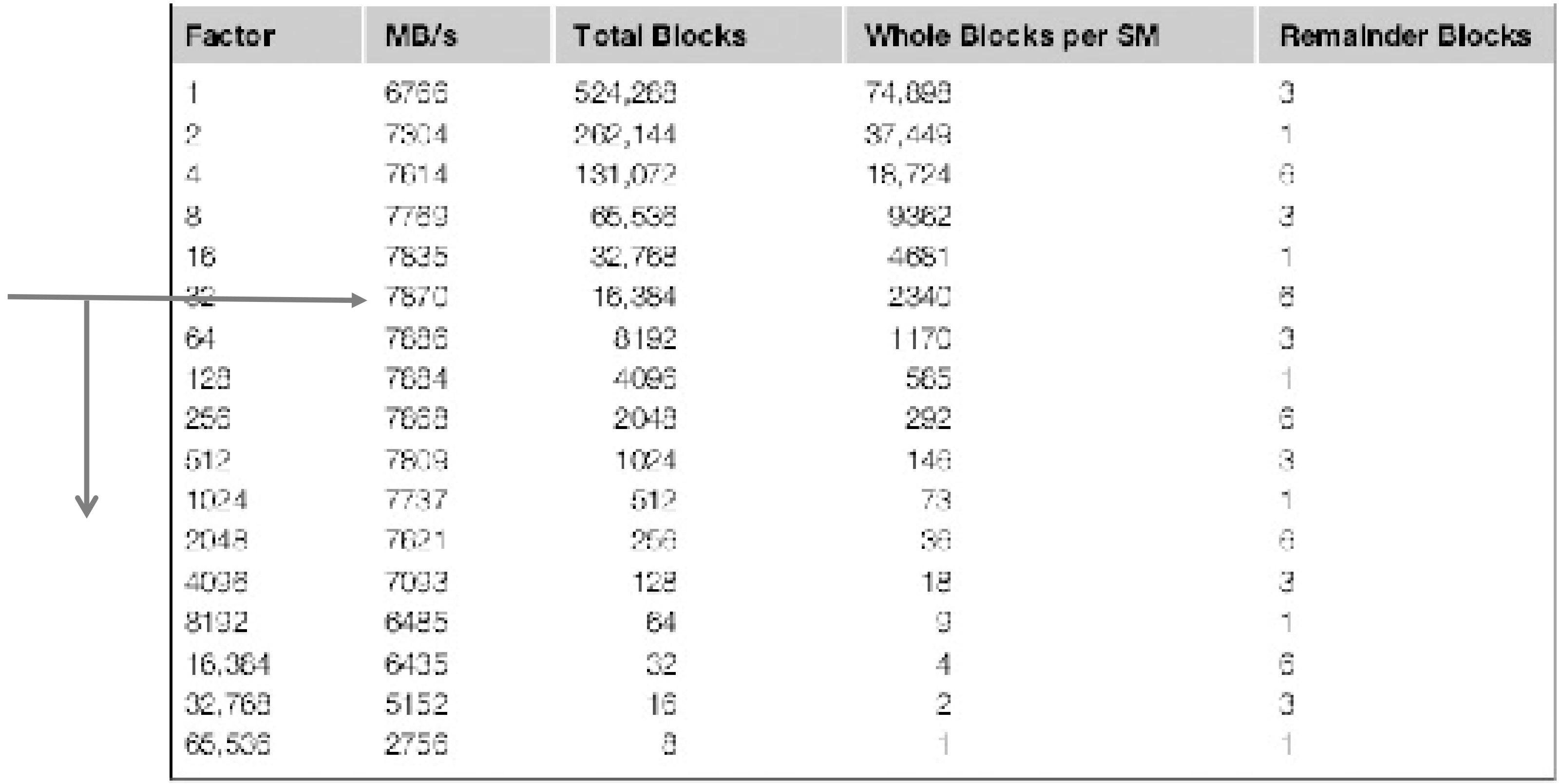
```
    for (unsigned int i=0, tid_offset=0; i< N; i++, tid_offset+=256)
    {
     const unsigned int value_u32 = d_hist_data[tid+tid_offset];

     atomicAdd(&(d_bin_data_shared[ ((value_u32 & 0x000000FF) ) ]),1);
     atomicAdd(&(d_bin_data_shared[  ((value_u32  &  0x0000FF00)  >>
8 ) ]),1);
     atomicAdd(&(d_bin_data_shared[  ((value_u32  &  0x00FF0000)  >>
16 ) ]),1);
     atomicAdd(&(d_bin_data_shared[  ((value_u32  &  0xFF000000)  >>
24 ) ]),1);
    }
    /* Wait for all threads to update shared memory */
    __syncthreads();

    /* The write the accumulated data back to global memory in blocks,
not scattered */
    atomicAdd(&(d_bin_data[threadIdx.x]),           d_bin_data_shared
[threadIdx.x]);
  }
```

GPU Algorithm 3



Factor	MB/s	Total Blocks	Whole Blocks per SM	Remainder Blocks
1	6766	524,288	74,898	3
2	7304	262,144	37,449	1
4	7614	131,072	18,724	6
8	7769	65,536	9362	3
16	7835	32,768	4681	1
32	7870	16,384	2340	6
64	7886	8192	1170	3
128	7884	4096	585	1
256	7868	2048	292	6
512	7809	1024	146	3
1024	7737	512	73	1
2048	7621	256	36	6
4096	7093	128	18	3
8192	6485	64	9	1
16,384	6435	32	4	6
32,768	5152	16	2	3
65,536	2756	8	1	1

GPU Algorithm 3



Factor	MB/s	Total Blocks	Whole Blocks per SM	Remainder Blocks
1	6766	524,288	74,898	3
2	7304	262,144	37,449	1
4	7614	131,072	18,724	6
8	7769	65,536	9362	3
16	7835	32,768	4681	1
32	7870	16,384	2340	6
64	7886	8192	1170	3
128	7884	4096	585	1
256	7868	2048	292	6
512	7809	1024	146	3
1024	7737	512	73	1
2048	7621	256	36	6
4096	7093	128	18	3
8192	6485	64	9	1
16,384	6435	32	4	6
32,768	5152	16	2	3
65,536	2756	8	1	1

not much growth in bandwidth after N=32 due to other factors impeding growth (atomics adds in this case)

Shane Cook “CUDA Programming”