VSCSE summer school - short course

Introduction to CUDA

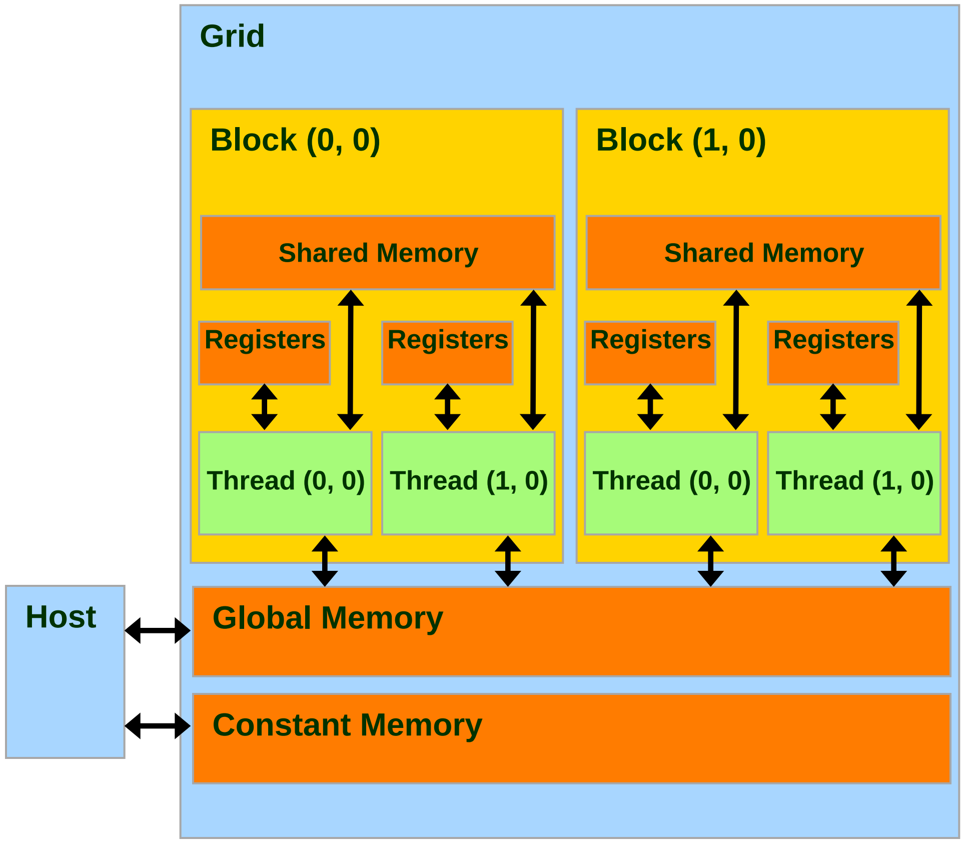
Lecture 4
CUDA Memory Model

Joshua A. Anderson

© 2009 David Kirk/NVIDIA and Wen-mei W. Hwu
Recorded for the Virtual School of Computational Science and Engineering

G80 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**



© 2009 David Kirk/NVIDIA and Wen-mei W. Hwu
Recorded for the Virtual School of Computational Science and Engineering

CUDA Variable Type Qualifiers

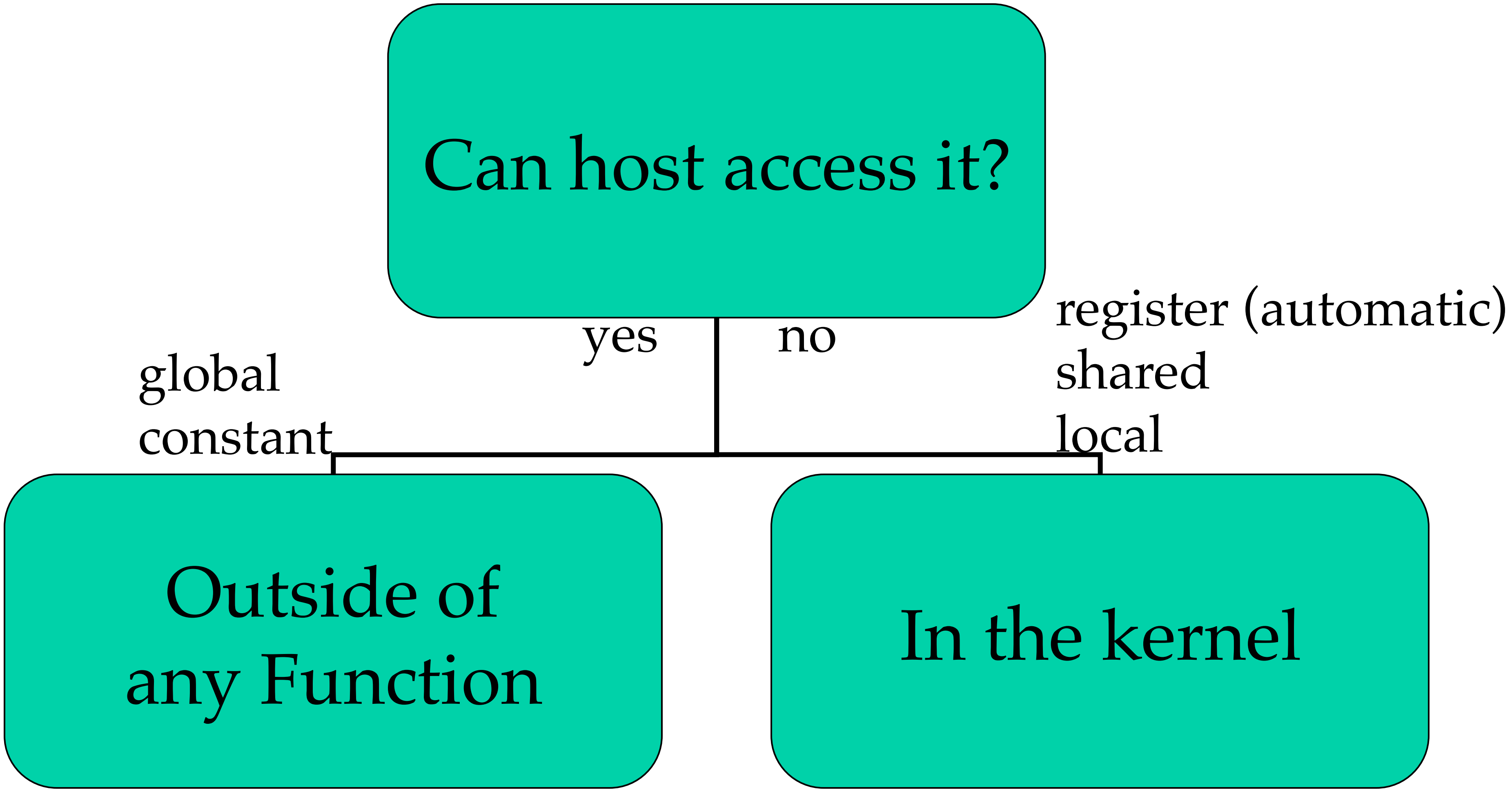
Variable declaration	Memory	Scope	Lifetime
Automatic assignment by compiler	local	thread	thread
`__device__ __shared__ int SharedVar;`	shared	block	block
`__device__ int GlobalVar;`	global	grid	application
`__device__ __constant__ int ConstantVar;`	constant	grid	application

- **`__device__`** is implied when used with **`__shared__`**, or **`__constant__`**

- Automatic variables without any qualifier reside in a register
  - Except arrays that reside in local memory

© 2009 David Kirk/NVIDIA and Wen-mei W. Hwu
Recorded for the Virtual School of Computational Science and Engineering

Where to Declare Variables?



© 2009 David Kirk/NVIDIA and Wen-mei W. Hwu
Recorded for the Virtual School of Computational Science and Engineering

Variable Type Restrictions

- Pointers typically 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;
    ```

  - Or obtained from cudaMalloc()
- Can also point into shared memory
- Pointers into constant memory are possible on Fermi

© 2009 David Kirk/NVIDIA and Wen-mei W. Hwu
Recorded for the Virtual School of Computational Science and Engineering

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

© 2009 David Kirk/NVIDIA and Wen-mei W. Hwu
Recorded for the Virtual School of Computational Science and Engineering

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, broadcast, data
- Carefully divide data according to access patterns
  - R/Only $\rightarrow$ constant memory (very fast if in cache)
  - R/W shared within Block $\rightarrow$ shared memory (very fast)
  - R/W within each thread $\rightarrow$ registers (very fast)
  - R/W inputs/results $\rightarrow$ global memory (very slow)

For texture memory usage, see courses.ece.uiuc.edu/ece498/al.

© 2009 David Kirk/NVIDIA and Wen-mei W. Hwu
Recorded for the Virtual School of Computational Science and Engineering

Matrix Multiplication using Shared Memory

© 2009 David Kirk/NVIDIA and Wen-mei W. Hwu
Recorded for the Virtual School of Computational Science and Engineering

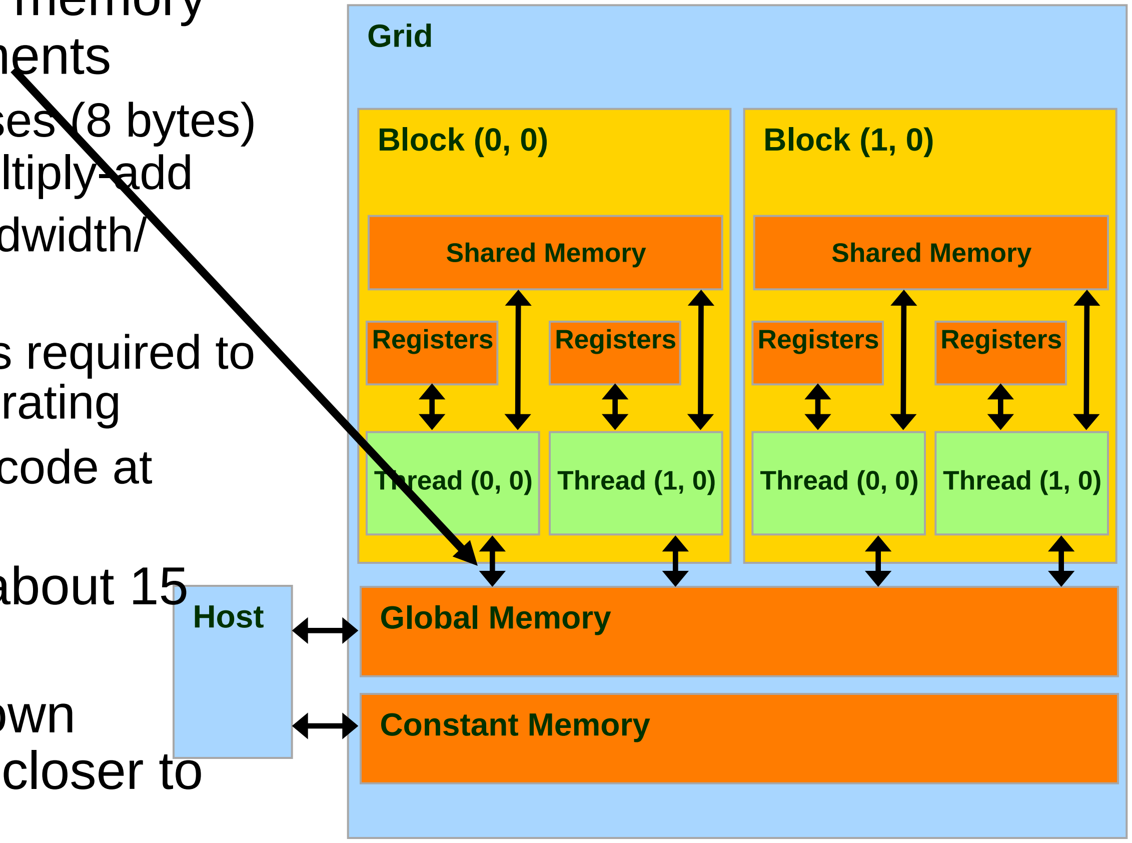
Revised Matrix Multiplication Kernel using Multiple Blocks

```
__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][k] * Nd[k][Col];

Pd[Row][Col] = Pvalue;
}
```

© 2009 David Kirk/NVIDIA and Wen-mei W. Hwu
Recorded for the Virtual School of Computational Science and Engineering

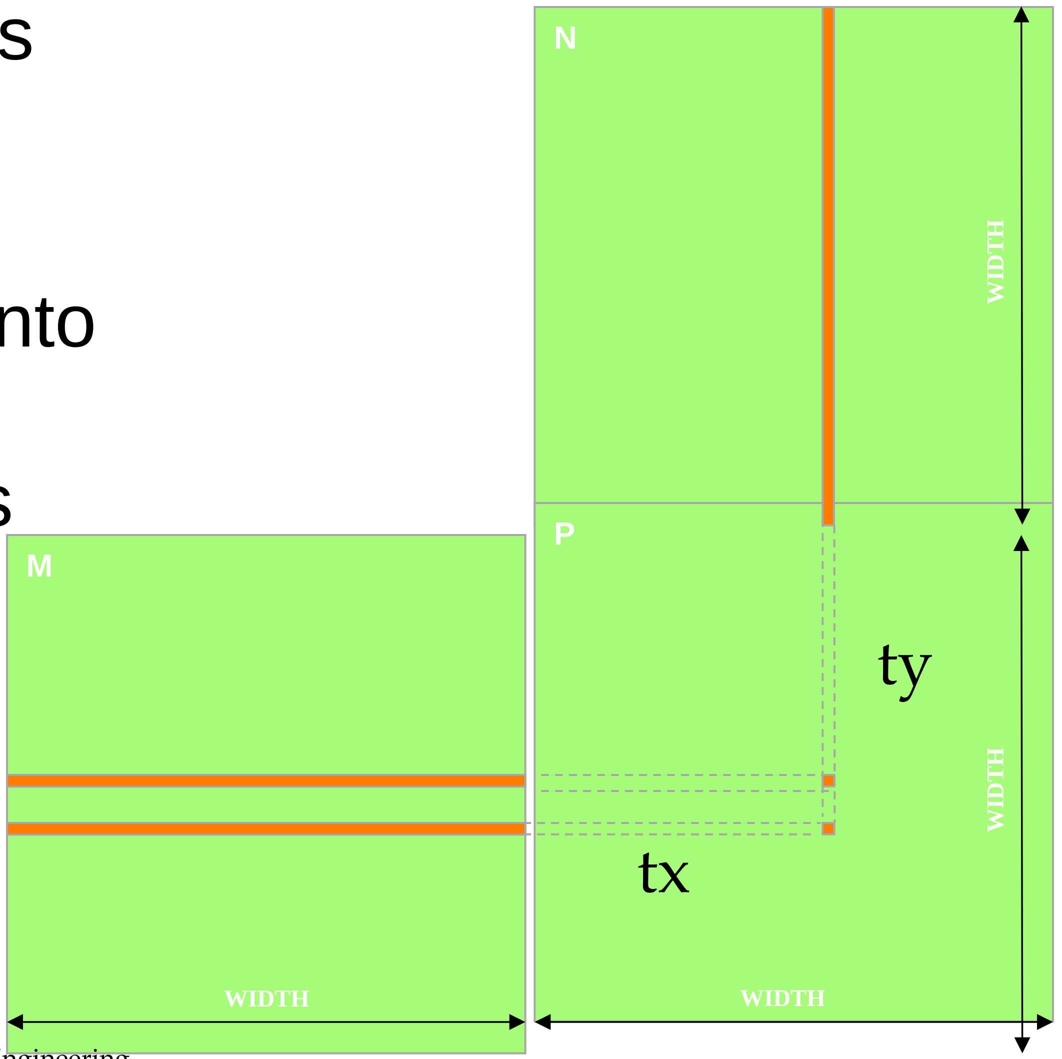
How about performance on G80?

- 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



Idea: Use Shared Memory to reuse global memory data

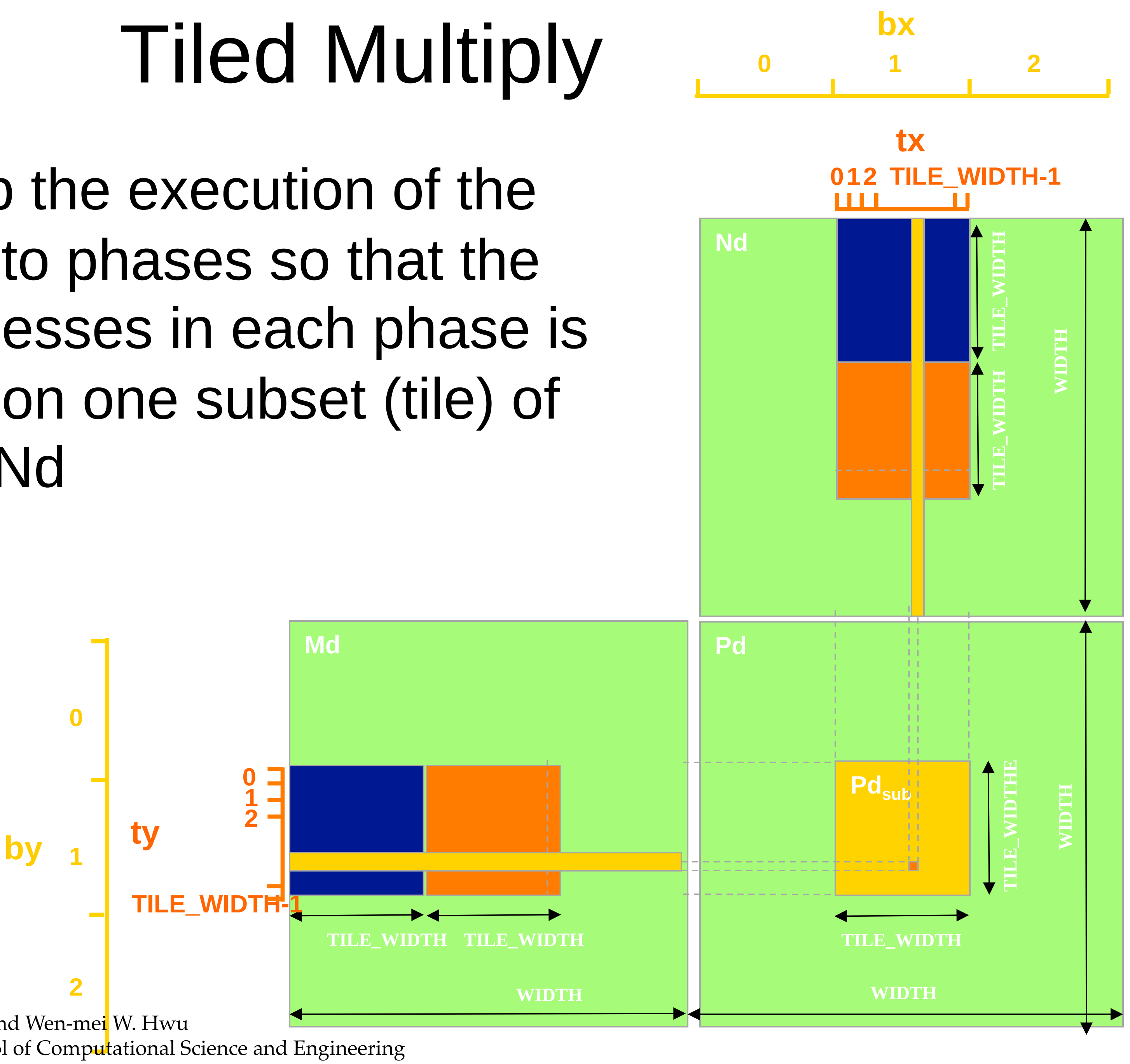
- Each input element is read by WIDTH threads.
- Load each element into Shared Memory and have several threads use the local version reduce the memory bandwidth
  - Tiled algorithms



© 2009 David Kirk/NVIDIA and Wen-mei W. Hwu
Recorded for the Virtual School of Computational Science and Engineering

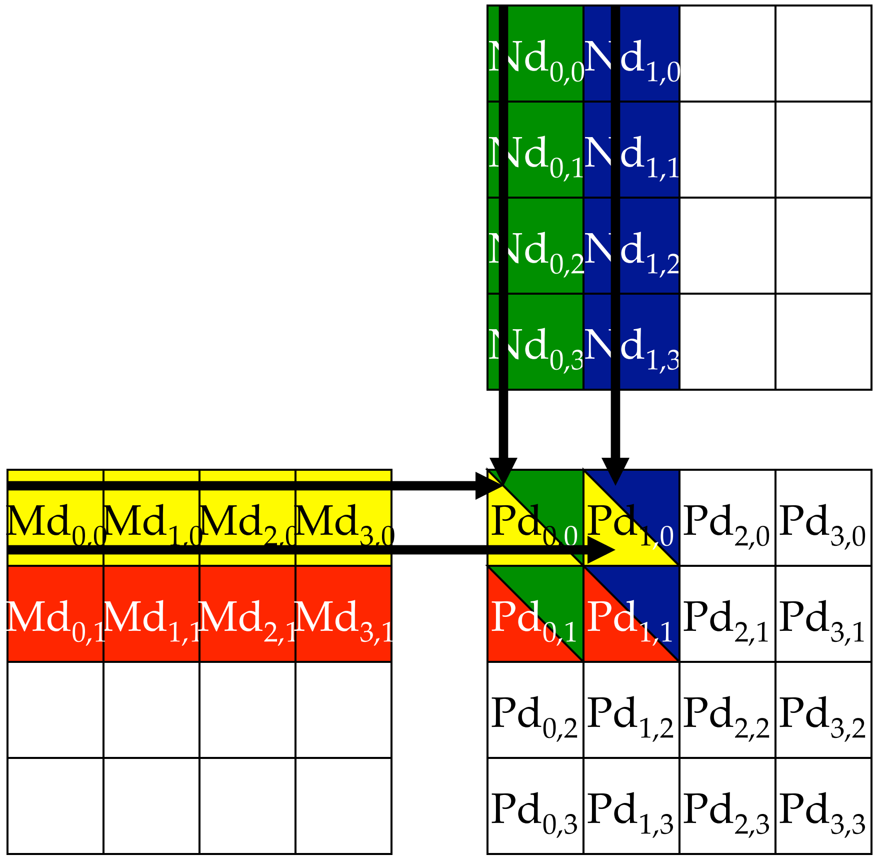
Tiled Multiply

- 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



© 2009 David Kirk/NVIDIA and Wen-mei W. Hwu
Recorded for the Virtual School of Computational Science and Engineering

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$_{3,3}$

© 2009 David Kirk/NVIDIA and Wen-mei W. Hwu
Recorded for the Virtual School of Computational Science and Engineering

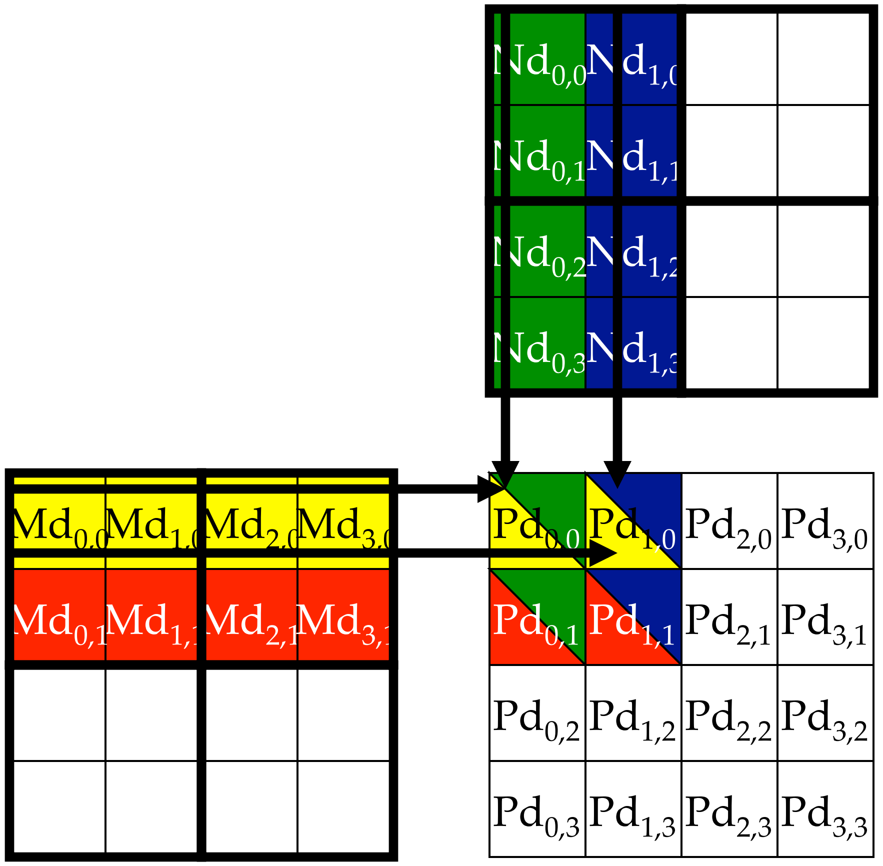
Every M and N Element is used exactly twice in generating a 2X2 tile of P

Access order

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

© 2009 David Kirk/NVIDIA and Wen-mei W. Hwu
Recorded for the Virtual School of Computational Science and Engineering

Breaking Md and Nd into Tiles



© 2009 David Kirk/NVIDIA and Wen-mei W. Hwu
Recorded for the Virtual School of Computational Science and Engineering

Each phase uses one tile from Md and one from Nd

	Phase 1			Phase 2		
$T_{0,0}$	**$Md_{0,0}$** ↓ $Mds_{0,0}$	**$Nd_{0,0}$** ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$	**$Md_{2,0}$** ↓ $Mds_{0,0}$	**$Nd_{0,2}$** ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$
$T_{1,0}$	**$Md_{1,0}$** ↓ $Mds_{1,0}$	**$Nd_{1,0}$** ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$	**$Md_{3,0}$** ↓ $Mds_{1,0}$	**$Nd_{1,2}$** ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$
$T_{0,1}$	**$Md_{0,1}$** ↓ $Mds_{0,1}$	**$Nd_{0,1}$** ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{0,1}$	**$Md_{2,1}$** ↓ $Mds_{0,1}$	**$Nd_{0,3}$** ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{0,1}$
$T_{1,1}$	**$Md_{1,1}$** ↓ $Mds_{1,1}$	**$Nd_{1,1}$** ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$	**$Md_{3,1}$** ↓ $Mds_{1,1}$	**$Nd_{1,3}$** ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$

time $\longrightarrow$

© 2009 David Kirk/NVIDIA and Wen-mei W. Hwu
Recorded for the Virtual School of Computational Science and Engineering

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
- 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

© 2009 David Kirk/NVIDIA and Wen-mei W. Hwu
Recorded for the Virtual School of Computational Science and Engineering

CUDA Code – Kernel Execution Configuration

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

© 2009 David Kirk/NVIDIA and Wen-mei W. Hwu
Recorded for the Virtual School of Computational Science and Engineering

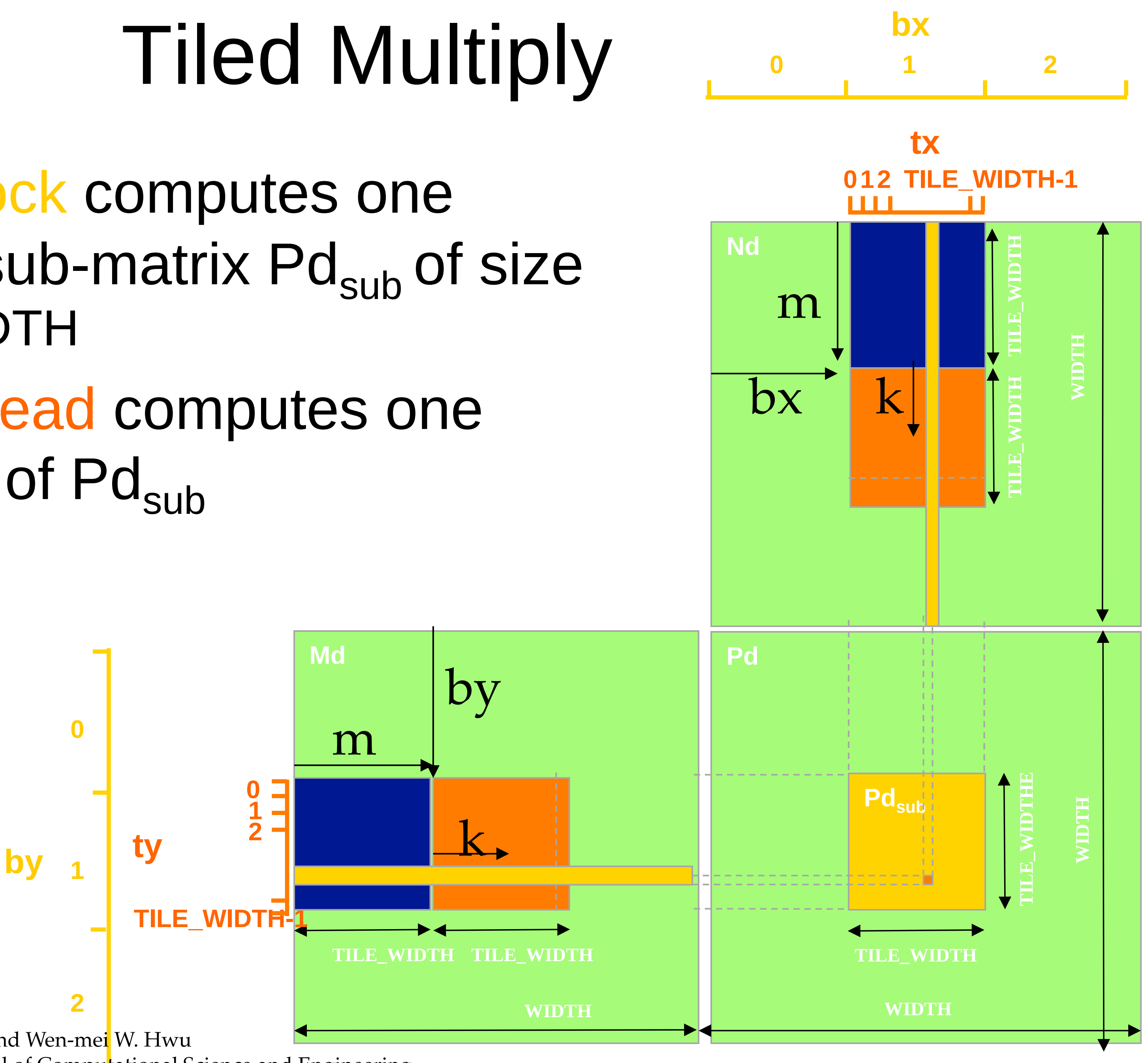
Tiled Matrix Multiplication Kernel

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

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

© 2009 David Kirk/NVIDIA and Wen-mei W. Hwu
Recorded for the Virtual School of Computational Science and Engineering

Tiled Multiply

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



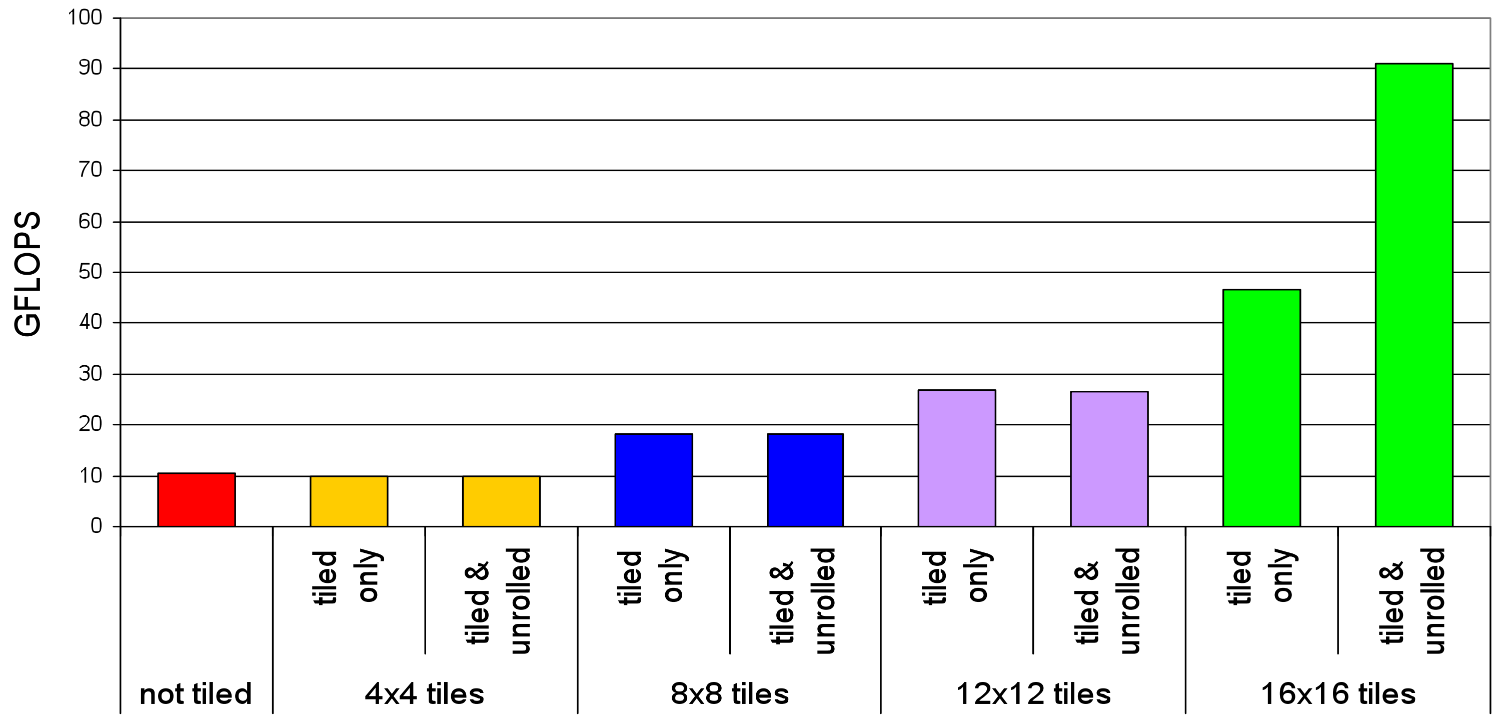
© 2009 David Kirk/NVIDIA and Wen-mei W. Hwu
Recorded for the Virtual School of Computational Science and Engineering

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.
  - 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 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!

© 2009 David Kirk/NVIDIA and Wen-mei W. Hwu
Recorded for the Virtual School of Computational Science and Engineering

Tiling Size Effects



© 2009 David Kirk/NVIDIA and Wen-mei W. Hwu
Recorded for the Virtual School of Computational Science and Engineering

Summary- Typical Structure of a CUDA Program

- Global variables declaration
  - __host__
  - __device__... __global__, __constant__, texture<>
- Function prototypes
  - __global__ void kernelOne(…)
  - float handyFunction(…)
- Main ()
  - allocate memory space on the device – cudaMalloc(&d_GlblVarPtr, bytes)
  - transfer data from host to device – cudaMemCpy(d_GlblVarPtr, h_Gl…)
  - execution configuration setup
  - kernel call – kernelOne<<<execution configuration>>>(args…);
  - transfer results from device to host – cudaMemCpy(h_GlblVarPtr,…)
  - optional: compare against golden (host computed) solution
- Kernel – void kernelOne(type args,…)
  - variables declaration - __shared__
    - automatic variables transparently assigned to registers or local memory
  - syncthreads()…
- Other functions
  - float handyFunction(int inVar…);

repeat as needed

© 2009 David Kirk/NVIDIA and Wen-mei W. Hwu
Recorded for the Virtual School of Computational Science and Engineering