



Introduction to CUDA (2 of 2)

Patrick Cozzi
University of Pennsylvania
CIS 565 - Spring 2012

Announcements

- Homework 1 due anytime today
- Homework 2 released. Due 02/13
- Last day to add or drop courses

Agenda

- Built-ins and functions
- Synchronizing threads
- Scheduling threads
- Memory model
- Matrix multiply revisited
- Atomic functions

Functional Declarations

	Executed on the:	Only callable from the:
<code>__global__ void KernelFunc()</code>	device	host
<code>__device__ float DeviceFunc()</code>	device	device
<code>__host__ float HostFunc()</code>	host	host

See Appendix B.1 in the NVIDIA CUDA C Programming Guide for more details

Functional Declarations

- `__global__`
 - Must return `void`
- `__device__`
 - Inlined by default

See Appendix B.1 in the NVIDIA CUDA C Programming Guide for more details

Functional Declarations

- What do these do?
 - `__global__ __host__ void func()`
 - `__device__ __host__ void func()`

Functional Declarations

- What do these do?
 - `__global__ __host__ void func()`
 - `__device__ __host__ void func()`

```
__host__ __device__ func()
{
    #if __CUDA_ARCH__ == 100
        // Device code path for compute capability 1.0
    #elif __CUDA_ARCH__ == 200
        // Device code path for compute capability 2.0
    #elif !defined(__CUDA_ARCH__)
        // Host code path
    #endif
}
```

Code from http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf

Functional Declarations

- Global and device functions
 - No recursion (except Fermi)
 - No static variables
 - No `malloc()`
 - Careful with function calls through pointers
- We'll see similar constraints in GLSL

Vector Types

- `char[1-4]`, `uchar[1-4]`
- `short[1-4]`, `ushort[1-4]`
- `int[1-4]`, `uint[1-4]`
- `long[1-4]`, `ulong[1-4]`
- `longlong[1-4]`, `ulonglong[1-4]`
- `float[1-4]`
- `double1`, `double2`

Vector Types

- Available in host and device code
- Construct with `make_<type name>`

```
int2 i2 = make_int2(1, 2);  
float4 f4 = make_float4(  
    1.0f, 2.0f, 3.0f, 4.0f);
```

Vector Types

- Access with `.x`, `.y`, `.z`, and `.w`

```
int2 i2 = make_int2(1, 2);  
int x = i2.x;  
int y = i2.y;
```

- No `.r`, `.g`, `.b`, `.a`, etc. like GLSL

Math Functions

- `Double` and `float` overloads
 - No vector overloads
- On the host, functions use the C runtime implementation if available

See Appendix C in the NVIDIA CUDA C Programming Guide for a complete list of math functions

Math Functions

■ Partial list:

- `sqrt`, `rsqrt`
- `exp`, `log`
- `sin`, `cos`, `tan`, `sincos`
- `asin`, `acos`, `atan2`
- `trunc`, `ceil`, `floor`

See Appendix C in the NVIDIA CUDA C Programming Guide for a complete list of math functions

Math Functions

■ *Intrinsic* function

- Device only
- Faster, but less accurate
- Prefixed with `__`
- `__exp`, `__log`, `__sin`, `__pow`, ...

See Appendix C in the NVIDIA CUDA C Programming Guide for a complete list of math functions

Review: Thread Hierarchies

- A **thread block** is a batch of threads that can **cooperate** with each other by:
 - Synchronizing their execution
 - For hazard-free shared memory accesses
 - Efficiently sharing data through a low latency **shared memory**
- Two threads from two different blocks cannot cooperate

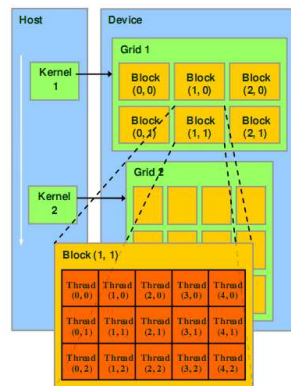


Image from: <http://courses.engr.illinois.edu/ece498/al/textbook/Chapter2-CudaProgrammingModel.pdf>

Review: Thread Hierarchies

```
int threadID = blockIdx.x *
    blockDim.x + threadIdx.x;
float x = input[threadID];
float y = func(x);
output[threadID] = y;
```

Review: Thread Hierarchies

```
int threadID = blockIdx.x *  
    blockDim.x + threadIdx.x;  
float x = input[threadID];  
float y = func(x);  
output[threadID] = y;
```

Use grid and block position to
compute a thread id

Review: Thread Hierarchies

```
int threadID = blockIdx.x *  
    blockDim.x + threadIdx.x;  
float x = input[threadID];  
float y = func(x);  
output[threadID] = y;
```

Use thread id to read from input

Review: Thread Hierarchies

```
int threadID = blockIdx.x *  
    blockDim.x + threadIdx.x;  
float x = input[threadID];  
float y = func(x);  
output[threadID] = y;
```

Run function on input: data-parallel!

Review: Thread Hierarchies

```
int threadID = blockIdx.x *  
    blockDim.x + threadIdx.x;  
float x = input[threadID];  
float y = func(x);  
output[threadID] = y;
```

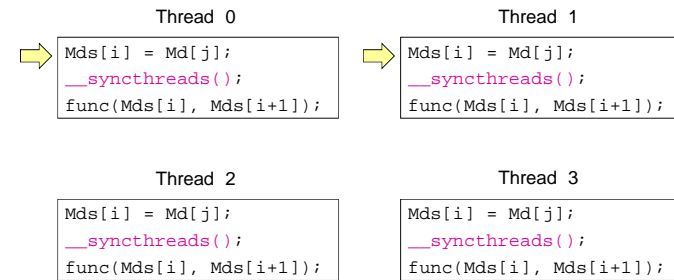
Use thread id to output result

Thread Synchronization

- Threads in a block can synchronize
 - call `__syncthreads()` to create a barrier
 - A thread waits at this call until all threads in the block reach it, then all threads continue

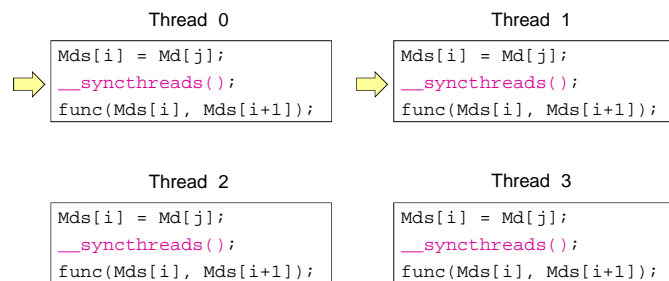
```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i + 1]);
```

Thread Synchronization



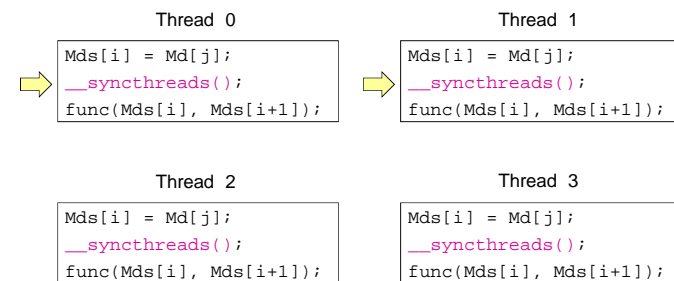
Time: 0

Thread Synchronization



Time: 1

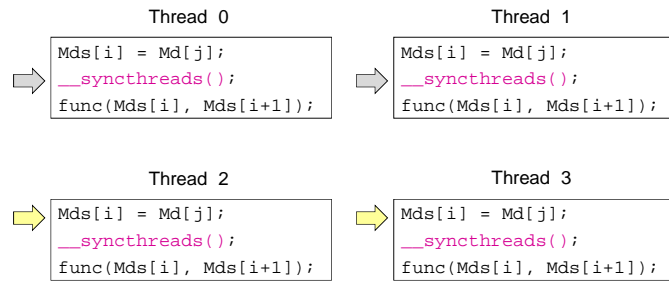
Thread Synchronization



Threads 0 and 1 are blocked at barrier

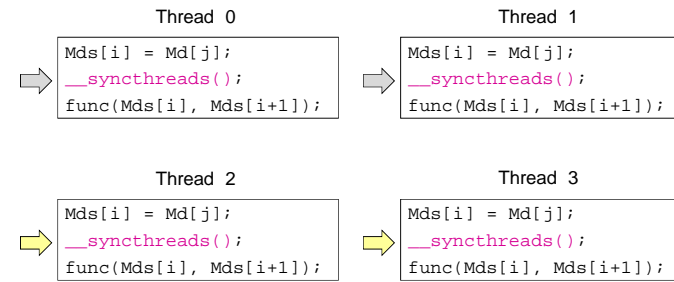
Time: 1

Thread Synchronization



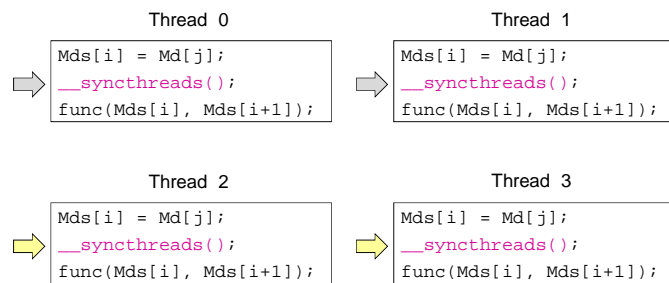
Time: 2

Thread Synchronization



Time: 3

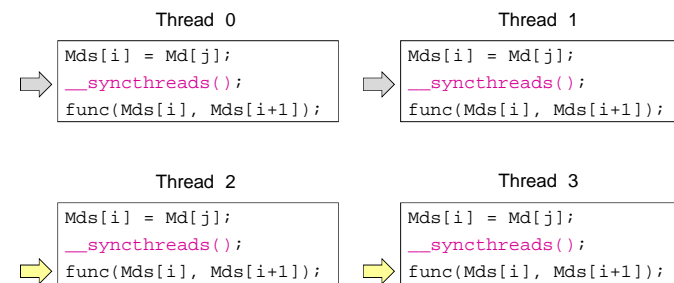
Thread Synchronization



All threads in block have reached barrier, any thread can continue

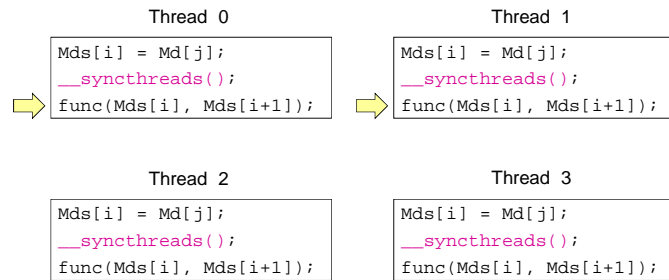
Time: 3

Thread Synchronization



Time: 4

Thread Synchronization



Time: 5

Thread Synchronization

- Why is it important that execution time be similar among threads?
- Why does it only synchronize within a block?

Thread Synchronization

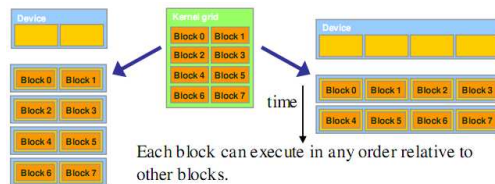


Figure 3.5 Lack of synchronization across blocks enables transparent scalability of CUDA programs

Image from <http://courses.engr.illinois.edu/ece498/al/textbook/Chapter3-CudaThreadingModel.pdf>

Thread Synchronization

- Can `__syncthreads()` cause a thread to hang?

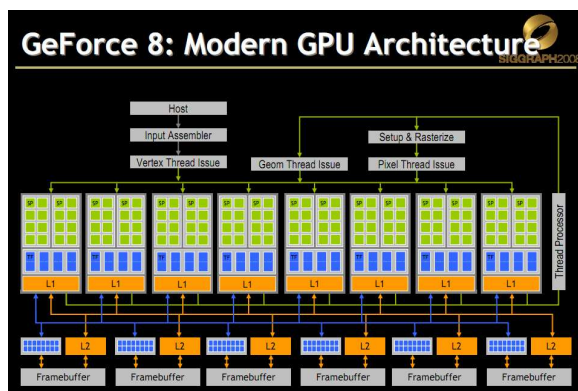
Thread Synchronization

```
if (someFunc())
{
    __syncthreads();
}
// ...
```

Thread Synchronization

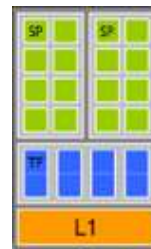
```
if (someFunc())
{
    __syncthreads();
}
else
{
    __syncthreads();
}
```

Scheduling Threads

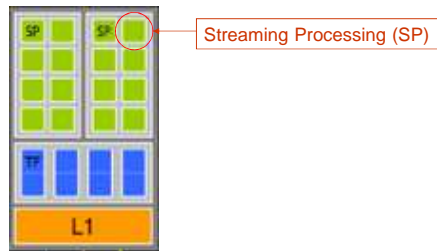


Slide from David Luebke: <http://s08.idav.ucdavis.edu/~luebke-nvidia-gpu-architecture.pdf>

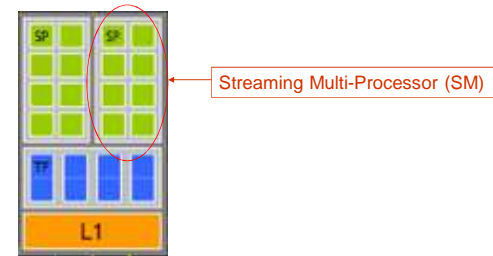
Scheduling Threads



Scheduling Threads



Scheduling Threads



Scheduling Threads

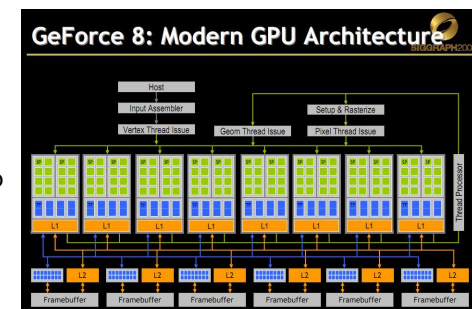


Look familiar?

Scheduling Threads

G80

- 16 SMs
- Each with 8 SPs
 - 128 total SPs
- Each SM hosts up to 768 threads
- Up to 12,288 threads in flight

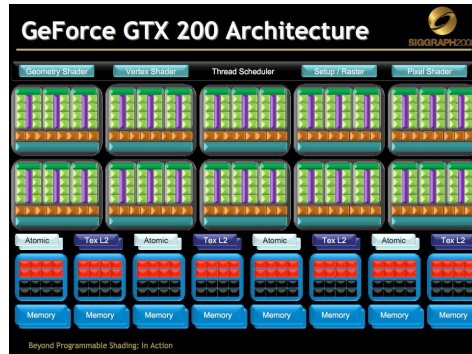


Slide from David Luebke: <http://s08.idav.ucdavis.edu/luebke-nvidia-gpu-architecture.pdf>

Scheduling Threads

GT200

- 30 SMs
- Each with 8 SPs
 - 240 total SPs
- Each SM hosts up to
 - 8 blocks, or
 - 1024 threads
- In flight, up to
 - 240 blocks, or
 - 30,720 threads



Slide from David Luebke: <http://s08.idav.ucdavis.edu/luebke-nvidia-gpu-architecture.pdf>

Scheduling Threads

- **Warp** – threads from a block
 - G80 / GT200 – 32 threads
 - Run on the same SM
 - Unit of thread scheduling
 - Consecutive `threadIdx` values
 - An implementation detail – in theory
 - `warpSize`

Scheduling Threads

- Warps for three blocks scheduled on the same SM.

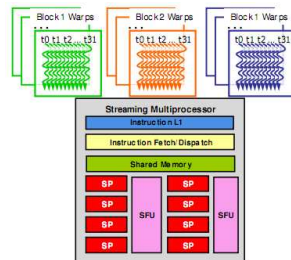
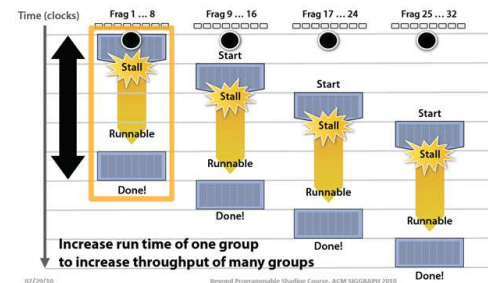


Image from: <http://courses.engr.illinois.edu/ece498/al/textbook/Chapter3-CudaThreadingModel.pdf>

Scheduling Threads

Remember this:



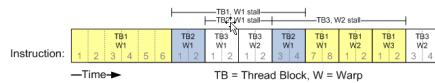
03/20/10

Beyond Programmable Shading Course, ACM SIGGRAPH 2010

Image from: http://bpsi10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraflop_BPS_SIGGRAPH2010.pdf

Scheduling Threads

- SM implements zero-overhead warp scheduling
 - 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



© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE498AL, University of Illinois, Urbana-Champaign

Slide from: <http://courses.engr.illinois.edu/ece498/al/Syllabus.html>

Scheduling Threads

- What happens if branches in a warp diverge?

Scheduling Threads

Remember this:

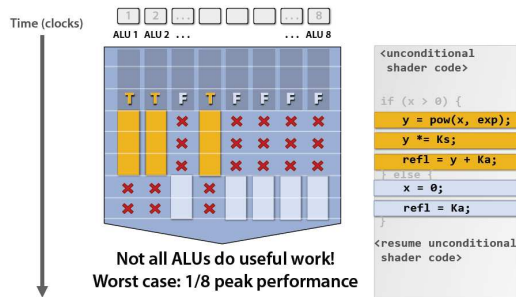


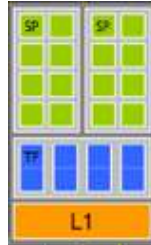
Image from: http://bps10.idav.ucdavis.edu/talks/03-fatahailan_gpuArchTeraflow_BPS_SIGGRAPH2010.pdf

Scheduling Threads

- If 3 blocks are assigned to an SM and each block has 256 threads, how many warps are there?
- A SM on GT200 can host up to 1024 threads, how many warps is that?

Scheduling Threads

- 32 threads per warp but 8 SPs per SM. What gives?



Scheduling Threads

- 32 threads per warp but 8 SPs per SM. What gives?
- When an SM schedules a warp:
 - Its instruction is ready
 - 8 threads enter the SPs on the 1st cycle
 - 8 more on the 2nd, 3rd, and 4th cycles
 - Therefore, 4 cycles are required to dispatch a warp

Scheduling Threads

- Question
 - A kernel has
 - 1 global memory read (200 cycles)
 - 4 non-dependent multiples/adds
 - How many warps are required to hide the memory latency?

Scheduling Threads

- Solution
 - Each warp has 4 multiples/adds
 - 16 cycles
 - We need to cover 200 cycles
 - $200 / 16 = 12.5$
 - $\text{ceil}(12.5) = 13$
 - 13 warps are required

Memory Model

Recall:

- Device code can:
 - R/W per-thread **registers**
 - R/W per-thread **local memory**
 - R/W per-block **shared memory**
 - R/W per-grid **global memory**
 - Read only per-grid **constant memory**
- Host code can
 - R/W per grid **global** and **constant** memories

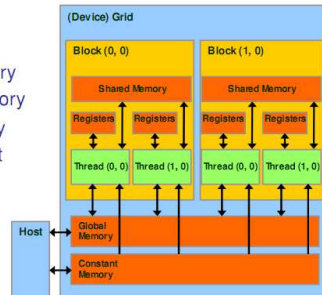
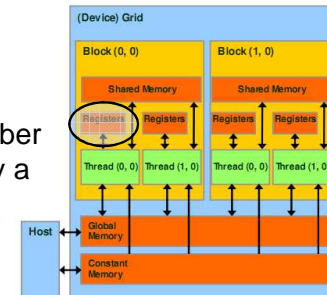


Image from: <http://courses.engr.illinois.edu/ece438/ai/textbook/Chapter2-CudaProgrammingModel.pdf>

Memory Model

■ Registers

- Per thread
- Fast, on-chip, read/write access
- Increasing the number of registers used by a kernel has what affect?



Memory Model

■ Registers - G80

- Per SM
 - Up to 768 threads
 - 8K registers
- How many registers per thread?

Memory Model

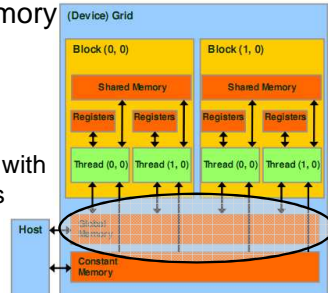
■ Registers - G80

- $8K / 768 = 10$ registers per thread
- Exceeding limit reduces threads by the block
- Example: Each thread uses 11 registers, and each block has 256 threads
 - How many threads can a SM host?
 - How many warps can a SM host?
 - What does having less warps mean?

Memory Model

■ Local Memory

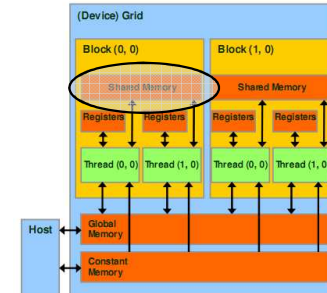
- Stored in global memory
 - Copy per thread
- Used for automatic arrays
 - Unless all accessed with only constant indices



Memory Model

■ Shared Memory

- Per block
- Fast, on-chip, read/write access
- Full speed random access



Memory Model

■ Shared Memory – G80

- Per SM
 - Up to 8 blocks
 - 16 KB
- How many KB per block

Memory Model

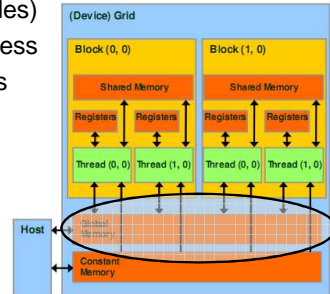
■ Shared Memory – G80

- 16 KB / 8 = 2 KB per block
- Example
 - If each block uses 5 KB, how many blocks can a SM host?

Memory Model

■ Global Memory

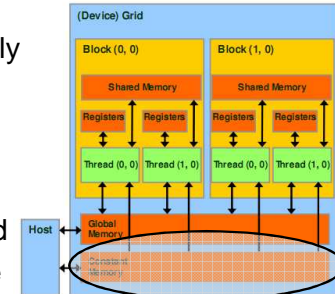
- Long latency (100s cycles)
- Off-chip, read/write access
- Random access causes performance hit
- Host can read/write
- GT200
 - 150 GB/s
 - Up to 4 GB
- G80 – 86.4 GB/s



Memory Model

■ Constant Memory

- Short latency, high bandwidth, read only access when all threads access the same location
- Stored in global memory but cached
- Host can read/write
- Up to 64 KB



Memory Model

Variable Declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	register	thread	kernel
Automatic array variables	local	thread	kernel
<code>__shared__ int sharedVar;</code>	shared	block	kernel
<code>__device__ int globalVar;</code>	global	grid	application
<code>__constant__ int constantVar;</code>	constant	grid	application

See Appendix B.2 in the NVIDIA CUDA C Programming Guide for more details

Memory Model

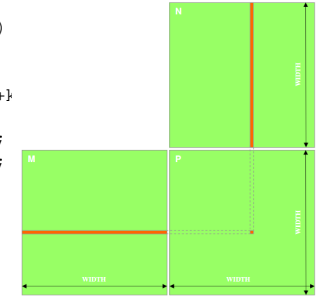
■ Global and constant variables

- Host can access with
 - `cudaGetSymbolAddress()`
 - `cudaGetSymbolSize()`
 - `cudaMemcpyToSymbol()`
 - `cudaMemcpyFromSymbol()`
- Constants must be declared outside of a function body

Let's revisit matrix multiple

Matrix Multiply: CPU Implementation

```
void MatrixMulOnHost(float* M, float* N, float* P, int width)
{
    for (int i = 0; i < width; ++i)
        for (int j = 0; j < width; ++j)
        {
            float sum = 0;
            for (int k = 0; k < width; ++k)
            {
                float a = M[i * width + k];
                float b = N[k * width + j];
                sum += a * b;
            }
            P[i * width + j] = sum;
        }
}
```



Code from: <http://courses.engr.illinois.edu/ece498/al/lectures/lecture3%20cuda%20threads%20spring%202010.ppt>

Matrix Multiply: CUDA Kernel

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue stores the Pd element that is computed by the thread
    float Pvalue = 0;

    for (int k = 0; k < Width; ++k)
    {
        float Mdelement = Md[ty * Md.width + k];
        float Ndelement = Nd[k * Nd.width + tx];
        Pvalue += Mdelement * Ndelement;
    }

    // Write the matrix to device memory each thread writes one element
    Pd[ty * Width + tx] = Pvalue;
}
```

Accessing a matrix, so using a 2D block

Code from: <http://courses.engr.illinois.edu/ece498/al/textbook/Chapter2-CudaProgrammingModel.pdf>

Matrix Multiply: CUDA Kernel

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue stores the Pd element that is computed by the thread
    float Pvalue = 0;

    for (int k = 0; k < Width; ++k)
    {
        float Mdelement = Md[ty * Md.width + k];
        float Ndelement = Nd[k * Nd.width + tx];
        Pvalue += Mdelement * Ndelement;
    }

    // Write the matrix to device memory each thread writes one element
    Pd[ty * Width + tx] = Pvalue;
}
```

Each kernel computes one output

Code from: <http://courses.engr.illinois.edu/ece498/al/textbook/Chapter2-CudaProgrammingModel.pdf>

Matrix Multiply: CUDA Kernel

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue stores the Pd element that is computed by the thread
    float Pvalue = 0;

    for (int k = 0; k < Width; ++k)
    {
        float Mdelement = Md[ty * Md.width + k];
        float Ndelement = Nd[k * Nd.width + tx];
        Pvalue += Mdelement * Ndelement;
    }

    // Write the matrix to device memory each thread writes one element
    Pd[ty * Width + tx] = Pvalue;
}
```

Where did the two outer for loops
in the CPU implementation go?

Code from: <http://courses.engr.illinois.edu/ece498/al/textbook/Chapter2-CudaProgrammingModel.pdf>

Matrix Multiply: CUDA Kernel

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue stores the Pd element that is computed by the thread
    float Pvalue = 0;

    for (int k = 0; k < Width; ++k)
    {
        float Mdelement = Md[ty * Md.width + k];
        float Ndelement = Nd[k * Nd.width + tx];
        Pvalue += Mdelement * Ndelement;
    }

    // Write the matrix to device memory each thread writes one element
    Pd[ty * Width + tx] = Pvalue;
}
```

No locks or synchronization, why?

Code from: <http://courses.engr.illinois.edu/ece498/al/textbook/Chapter2-CudaProgrammingModel.pdf>

Matrix Multiply

Problems

- Limited matrix size
 - Only uses one block
 - G80 and GT200 – up to 512 threads per block
- Lots of global memory access

Matrix Multiply

Remove size limitation

- Break Pd matrix into tiles
- Assign each tile to a block
- Use `threadIdx` and `blockIdx` for indexing

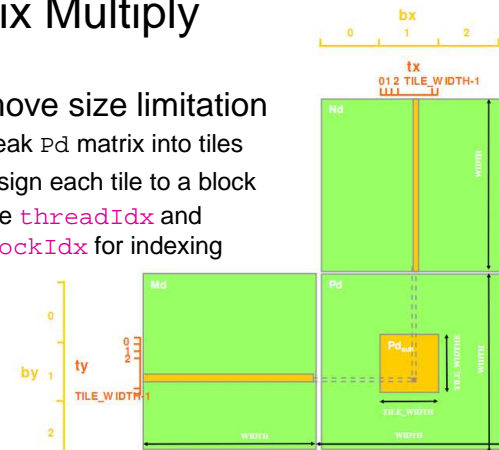


Image from <http://courses.engr.illinois.edu/ece498/al/textbook/Chapter3-CudaThreadingModel.pdf>

Matrix Multiply

■ Example

- Matrix: 4x4
- TILE_WIDTH = 2
- Block size: 2x2

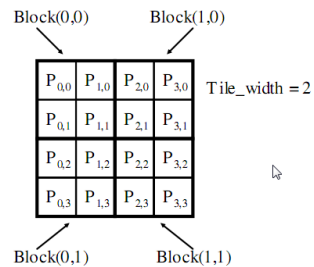


Image from <http://courses.engr.illinois.edu/ece498/al/textbook/Chapter3-CudaThreadingModel.pdf>

Matrix Multiply

■ Example

- Matrix: 4x4
- TILE_WIDTH = 2
- Block size: 2x2

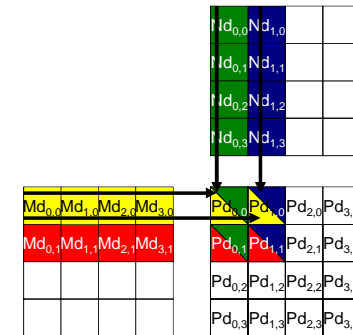


Image from <http://courses.engr.illinois.edu/ece498/al/Syllabus.html>

Matrix Multiply

```
__global__ void MatrixMulKernel(
    float* Md, float* Nd, float* Pd, int Width)
{
    int Row = blockIdx.y * TILE_WIDTH + threadIdx.y;
    int Col = blockIdx.x * TILE_WIDTH + threadIdx.x;

    float Pvalue = 0;
    for (int k = 0; k < Width; ++k)
        Pvalue += Md[Row * Width + k] * Nd[k * Width + Col];

    Pd[Row * Width + Col] = Pvalue;
}
```

Code from <http://courses.engr.illinois.edu/ece498/al/Syllabus.html>

Matrix Multiply

Calculate the row index of the Pd element and M

```
__global__ void MatrixMulKernel(
    float* Md, float* Nd, float* Pd, int Width)
{
    int Row = blockIdx.y * TILE_WIDTH + threadIdx.y;
    int Col = blockIdx.x * TILE_WIDTH + threadIdx.x;

    float Pvalue = 0;
    for (int k = 0; k < Width; ++k)
        Pvalue += Md[Row * Width + k] * Nd[k * Width + Col];

    Pd[Row * Width + Col] = Pvalue;
}
```

Code from <http://courses.engr.illinois.edu/ece498/al/Syllabus.html>

Matrix Multiply

Calculate the column index of Pd and N

```
__global__ void MatrixMulKernel(
    float* Md, float* Nd, float* Pd, int Width)
{
    int Row = blockIdx.y * TILE_WIDTH + threadIdx.y;
    int Col = blockIdx.x * TILE_WIDTH + threadIdx.x;

    float Pvalue = 0;
    for (int k = 0; k < Width; ++k)
        Pvalue += Md[Row * Width + k] * Nd[k * Width + Col];

    Pd[Row * Width + Col] = Pvalue;
}
```

Code from <http://courses.engr.illinois.edu/ece498/al/Syllabus.html>

Matrix Multiply

Each thread computes one element of the block sub-matrix

```
__global__ void MatrixMulKernel(
    float* Md, float* Nd, float* Pd, int Width)
{
    int Row = blockIdx.y * TILE_WIDTH + threadIdx.y;
    int Col = blockIdx.x * TILE_WIDTH + threadIdx.x;

    float Pvalue = 0;
    for (int k = 0; k < Width; ++k)
        Pvalue += Md[Row * Width + k] * Nd[k * Width + Col];

    Pd[Row * Width + Col] = Pvalue;
}
```

Code from <http://courses.engr.illinois.edu/ece498/al/Syllabus.html>

Matrix Multiply

■ Invoke kernel:

```
dim3 dimBlock(Width / TILE_WIDTH, Height / TILE_WIDTH);
dim3 dimGrid(TILE_WIDTH, TILE_WIDTH);

MatrixMulKernel<<<dimGrid, dimBlock>>>(
    Md, Nd, Pd, TILE_WIDTH);
```

What about
global memory
access?

Matrix Multiply

- Limited by global memory bandwidth
 - G80 peak GFLOPS: 346.5
 - Require 1386 GB/s to achieve this
 - G80 memory bandwidth: 86.4 GB/s
 - Limits code to 21.6 GFLOPS
 - In practice, code runs at 15 GFLOPS
 - Must drastically reduce global memory access

Matrix Multiply

- Each input element is read by `width` threads
- Use shared memory to reduce global memory bandwidth

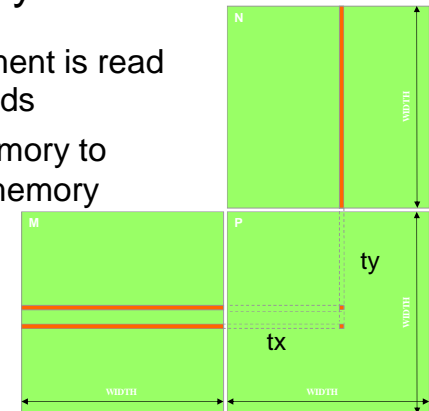


Image from <http://courses.engr.illinois.edu/ece498/al/Syllabus.html>

Matrix Multiply

- Break kernel into phases
 - Each phase accumulates P_d using a subset of M_d and N_d
 - Each phase has good data locality

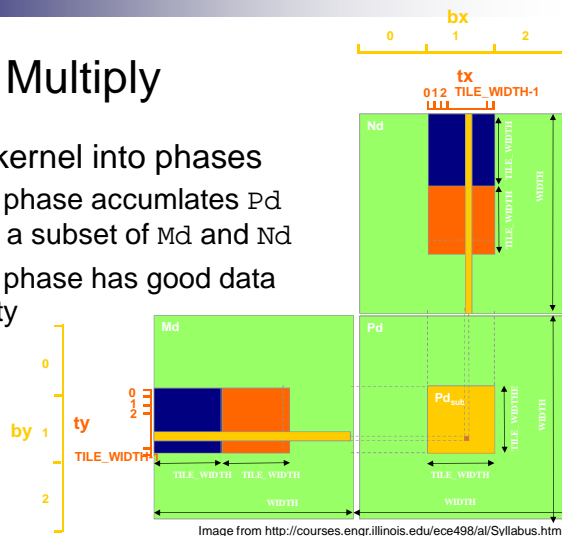


Image from <http://courses.engr.illinois.edu/ece498/al/Syllabus.html>

Matrix Multiply

- Each thread
 - loads one element of M_d and N_d in the tile into shared memory

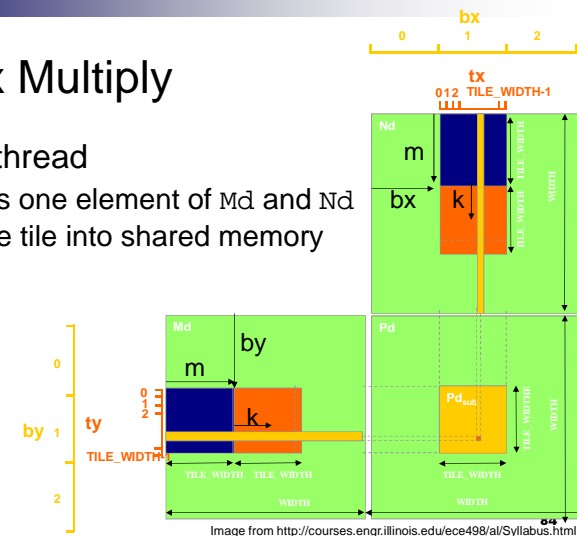


Image from <http://courses.engr.illinois.edu/ece498/al/Syllabus.html>

```

__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;

    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;

    float Pvalue = 0;
    for (int m = 0; m < Width/TILE_WIDTH; ++m) {
        Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
        Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
        __syncthreads();

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

```

Code from <http://courses.engr.illinois.edu/ece498/al/Syllabus.html>

```

__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;

    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;

    float Pvalue = 0;
    for (int m = 0; m < Width/TILE_WIDTH; ++m) {
        Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
        Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
        __syncthreads();

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

```

Shared memory for a subset of Md and Nd

Code from <http://courses.engr.illinois.edu/ece498/al/Syllabus.html>

```

__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;

    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;

    float Pvalue = 0;
    for (int m = 0; m < Width/TILE_WIDTH; ++m) {
        Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
        Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
        __syncthreads();

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

```

Width/TILE_WIDTH
• Number of phases
m
• Index for current phase

Code from <http://courses.engr.illinois.edu/ece498/al/Syllabus.html>

```

__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;

    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;

    float Pvalue = 0;
    for (int m = 0; m < Width/TILE_WIDTH; ++m) {
        Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
        Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
        __syncthreads();

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

```

Bring one element each from Md and Nd into shared memory

Code from <http://courses.engr.illinois.edu/ece498/al/Syllabus.html>

```

__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;

    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;

    float Pvalue = 0;
    for (int m = 0; m < Width/TILE_WIDTH; ++m) {
        Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
        Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
        __syncthreads();
        for (int k = 0; k < TILE_WIDTH; ++k)
            Pvalue += Mds[ty][k] * Nds[k][tx];
        __syncthreads();
    }
    Pd[Row*Width+Col] = Pvalue;
}

```

Wait for every thread in the block, i.e., wait for the tile to be in shared memory

Code from <http://courses.engr.illinois.edu/ece498/al/Syllabus.html>

```

__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;

    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;

    float Pvalue = 0;
    for (int m = 0; m < Width/TILE_WIDTH; ++m) {
        Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
        Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
        __syncthreads();
        for (int k = 0; k < TILE_WIDTH; ++k)
            Pvalue += Mds[ty][k] * Nds[k][tx];
        __syncthreads();
    }
    Pd[Row*Width+Col] = Pvalue;
}

```

Accumulate subset of dot product

Code from <http://courses.engr.illinois.edu/ece498/al/Syllabus.html>

```

__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;

    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;

    float Pvalue = 0;
    for (int m = 0; m < Width/TILE_WIDTH; ++m) {
        Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
        Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
        __syncthreads();
        for (int k = 0; k < TILE_WIDTH; ++k)
            Pvalue += Mds[ty][k] * Nds[k][tx];
        __syncthreads();
    }
    Pd[Row*Width+Col] = Pvalue;
}

```

Why?

Code from <http://courses.engr.illinois.edu/ece498/al/Syllabus.html>

```

__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;

    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;

    float Pvalue = 0;
    for (int m = 0; m < Width/TILE_WIDTH; ++m) {
        Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
        Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
        __syncthreads();
        for (int k = 0; k < TILE_WIDTH; ++k)
            Pvalue += Mds[ty][k] * Nds[k][tx];
        __syncthreads();
    }
    Pd[Row*Width+Col] = Pvalue;
}

```

Write final answer to global memory

Code from <http://courses.engr.illinois.edu/ece498/al/Syllabus.html>

Matrix Multiply

- How do you pick `TILE_WIDTH`?
 - How can it be too large?

Matrix Multiply

- How do you pick `TILE_WIDTH`?
 - How can it be too large?
 - By exceeding the maximum number of threads/block
 - G80 and GT200 – 512
 - Fermi – 1024

Matrix Multiply

- How do you pick `TILE_WIDTH`?
 - How can it be too large?
 - By exceeding the maximum number of threads/block
 - G80 and GT200 – 512
 - Fermi – 1024
 - By exceeding the shared memory limitations
 - G80: 16KB per SM and up to 8 blocks per SM
 - 2 KB per block
 - 1 KB for `Nds` and 1 KB for `Mds` ($16 * 16 * 4$)
 - `TILE_WIDTH` = 16
 - A larger `TILE_WIDTH` will result in less blocks

Matrix Multiply

- Shared memory tiling benefits
 - Reduces global memory access by a factor of `TILE_WIDTH`
 - 16x16 tiles reduces by a factor of 16
 - G80
 - Now global memory supports 345.6 GFLOPS
 - Close to maximum of 346.5 GFLOPS

First-order Size Considerations in G80

- Each **thread block** should have many threads
 - `TILE_WIDTH` of 16 gives $16 \times 16 = 256$ threads
- There should be many thread blocks
 - A 1024×1024 Pd gives $64 \times 64 = 4K$ Thread Blocks
- Each thread block perform $2 \times 256 = 512$ float loads from global memory for $256 * (2 \times 16) = 8K$ mul/add operations.
 - Memory bandwidth no longer a limiting factor

Slide from <http://courses.engr.illinois.edu/ece498/al/Syllabus.html>

Atomic Functions

- What is the value of `count` if 8 threads execute `++count`?

```
__device__ unsigned int count = 0;
// ...
++count;
```

Atomic Functions

- Read-modify-write atomic operation
 - Guaranteed no interference from other threads
 - No guarantee on order
- Shared or global memory
- Requires compute capability 1.1 (> G80)

See G.1 in the NVIDIA CUDA C Programming Guide for full compute capability requirements

Atomic Functions

- What is the value of `count` if 8 threads execute `atomicAdd` below?

```
__device__ unsigned int count = 0;
// ...
// atomic ++count
atomicAdd(&count, 1);
```

Atomic Functions

- How do you implement `atomicAdd`?

```
__device__ int atomicAdd(  
    int *address, int val);
```

Atomic Functions

- How do you implement `atomicAdd`?

```
__device__ int atomicAdd(  
    int *address, int val)  
{ // Made up keyword:  
    __lock (address) {  
        *address += value;  
    }  
}
```

Atomic Functions

- How do you implement `atomicAdd` **without** locking?

Atomic Functions

- How do you implement `atomicAdd` **without** locking?
- What if you were given an atomic compare and swap?

```
int atomicCAS(int *address, int  
    compare, int val);
```

Atomic Functions

■ `atomicCAS` pseudo implementation

```
int atomicCAS(int *address,
              int compare, int val)
{ // Made up keyword
  __lock(address) {
    int old = *address;
    *address = (old == compare) ? val : old;
    return old;
  }
}
```

Atomic Functions

■ `atomicCAS` pseudo implementation

```
int atomicCAS(int *address,
              int compare, int val)
{ // Made up keyword
  __lock(address) {
    int old = *address;
    *address = (old == compare) ? val : old;
    return old;
  }
}
```

Atomic Functions

■ `atomicCAS` pseudo implementation

```
int atomicCAS(int *address,
              int compare, int val)
{ // Made up keyword
  __lock(address) {
    int old = *address;
    *address = (old == compare) ? val : old;
    return old;
  }
}
```

Atomic Functions

■ Example:

```
*addr = 1;

atomicCAS(addr, 1, 2);
atomicCAS(addr, 1, 3);
atomicCAS(addr, 2, 3);
```

Atomic Functions

■ Example:

```
*addr = 1;
```

```
atomicCAS(addr, 1, 2); // returns 1  
atomicCAS(addr, 1, 3); // *addr = 2  
atomicCAS(addr, 2, 3);
```

Atomic Functions

■ Example:

```
*addr = 1;
```

```
atomicCAS(addr, 1, 2);  
atomicCAS(addr, 1, 3); // returns 2  
atomicCAS(addr, 2, 3); // *addr = 2
```

Atomic Functions

■ Example:

```
*addr = 1;
```

```
atomicCAS(addr, 1, 2);  
atomicCAS(addr, 1, 3);  
atomicCAS(addr, 2, 3); // returns 2  
                        // *addr = 3
```

Atomic Functions

■ Again, how do you implement `atomicAdd` given `atomicCAS`?

```
__device__ int atomicAdd(  
    int *address, int val);
```

Atomic Functions

```
__device__ int atomicAdd(int *address, int val)
{
    int old = *address, assumed;
    do {
        assumed = old;
        old = atomicCAS(address,
            assumed, val + assumed);
    } while (assumed != old);
    return old;
}
```

Atomic Functions

```
__device__ int atomicAdd(int *address, int val)
{
    int old = *address, assumed;
    do {
        assumed = old;
        old = atomicCAS(address,
            assumed, val + assumed);
    } while (assumed != old);
    return old;
}
```

Read original value at *address.

Atomic Functions

```
__device__ int atomicAdd(int *address, int val)
{
    int old = *address, assumed;
    do {
        assumed = old;
        old = atomicCAS(address,
            assumed, val + assumed);
    } while (assumed != old);
    return old;
}
```

If the value at *address didn't change, increment it.

Atomic Functions

```
__device__ int atomicAdd(int *address, int val)
{
    int old = *address, assumed;
    do {
        assumed = old;
        old = atomicCAS(address,
            assumed, assumed + val);
    } while (assumed != old);
    return old;
}
```

Otherwise, loop until atomicCAS succeeds.

The value of *address after this function returns is not necessarily the original value of *address + val, why?

Atomic Functions

- Lots of atomics:

// Arithmetic	// Bitwise
<code>atomicAdd()</code>	<code>atomicAnd()</code>
<code>atomicSub()</code>	<code>atomicOr()</code>
<code>atomicExch()</code>	<code>atomicXor()</code>
<code>atomicMin()</code>	
<code>atomicMax()</code>	
<code>atomicAdd()</code>	
<code>atomicDec()</code>	
<code>atomicCAS()</code>	

See B.10 in the NVIDIA CUDA C Programming Guide

Atomic Functions

- How can threads from different blocks work together?
- Use atomics sparingly. Why?