Introduction to CUDA 1 of 2

Patrick Cozzi
University of Pennsylvania
CIS 565 - Fall 2012

Announcements

- Project 0 due Tues 09/18
- Project 1 released today. Due Friday 09/28
  - Starts student blogs
  - Get approval for all third-party code

Acknowledgements

- Many slides are from David Kirk and Wen-mei Hwu's UIUC course:
  - http://courses.engr.illinois.edu/ece498/al/

GPU Architecture Review

- GPUs are specialized for
  - Compute-intensive, highly parallel computation
  - Graphics!
- Transistors are devoted to:
  - Processing
  - Not:
    - Data caching
    - Flow control

GPU Architecture Review

Transistor Usage

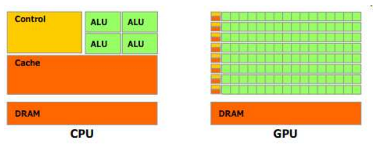


Figure 1-2. The GPU Devotes More Transistors to Data Processing

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

Let's program this thing!

GPU Computing History

- 2001/2002 – researchers see GPU as data-parallel coprocessor
  - The *GPGPU* field is born
- 2007 – NVIDIA releases CUDA
  - *CUDA* – Compute Uniform Device Architecture
  - GPGPU shifts to *GPU Computing*
- 2008 – Khronos releases *OpenCL* specification

CUDA Abstractions

- A hierarchy of thread groups
- Shared memories
- Barrier synchronization

CUDA Terminology

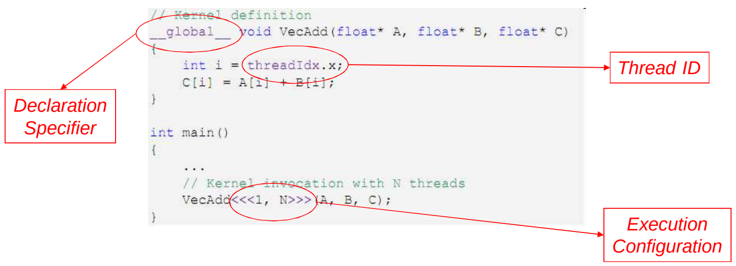
- *Host* – typically the CPU
  - Code written in ANSI C
- *Device* – typically the GPU (data-parallel)
  - Code written in *extended* ANSI C
- Host and device have separate memories
- CUDA Program
  - Contains both host and device code

CUDA Terminology

- *Kernel* – data-parallel function
  - Invoking a kernel creates lightweight threads on the device
    - Threads are generated and scheduled with hardware
- Similar to a *shader* in OpenGL?

CUDA Kernels

- Executed N times in parallel by N different *CUDA threads*



```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
}
```

CUDA Program Execution

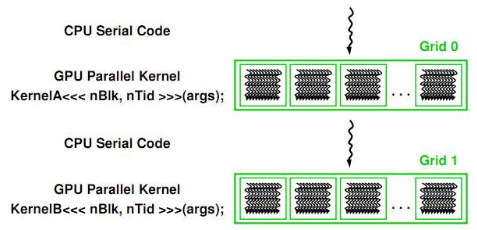


Image from: http://courses.engr.illinois.edu/ece498/al/textbook/Chapter2-CudaProgrammingModel.pdf

Thread Hierarchies

- *Grid* – one or more thread blocks
  - 1D or 2D
- *Block* – array of threads
  - 1D, 2D, or 3D
  - Each block in a grid has the same number of threads
  - Each thread in a block can
    - Synchronize
    - Access shared memory

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

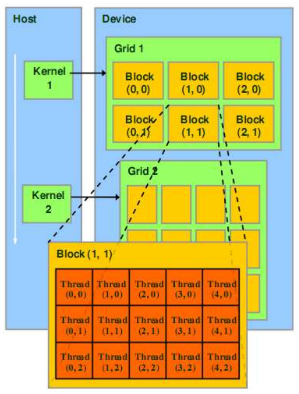
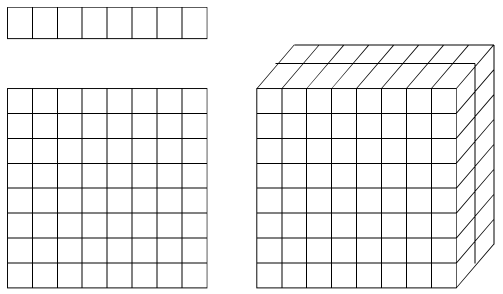


Image from: http://courses.engr.illinois.edu/ece498/al/textbook/Chapter2-CudaProgrammingModel.pdf

Thread Hierarchies

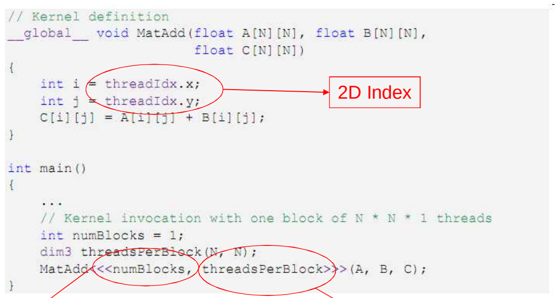
- *Block* – 1D, 2D, or 3D
  - Example: Index into vector, matrix, volume



Thread Hierarchies

- *Thread ID*: Scalar thread identifier
- Thread Index: `threadIdx`
- 1D: Thread ID == Thread Index
- 2D with size (D_x, D_y)
  - Thread ID of index (x, y) == $x + y D_y$
- 3D with size (D_x, D_y, D_z)
  - Thread ID of index (x, y, z) == $x + y D_y + z D_x D_y$

Thread Hierarchies



```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```

2D Index

1 Thread Block

2D Block

Thread Hierarchies

- Thread Block
  - Group of threads
    - G80 and GT200: Up to 512 threads
    - Fermi: Up to 1024 threads
  - Reside on same processor core
  - Share memory of that core

Thread Hierarchies

- Thread Block
  - Group of threads
    - G80 and GT200: Up to 512 threads
    - Fermi: Up to 1024 threads
  - Reside on same processor core
  - Share memory of that core

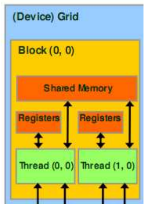
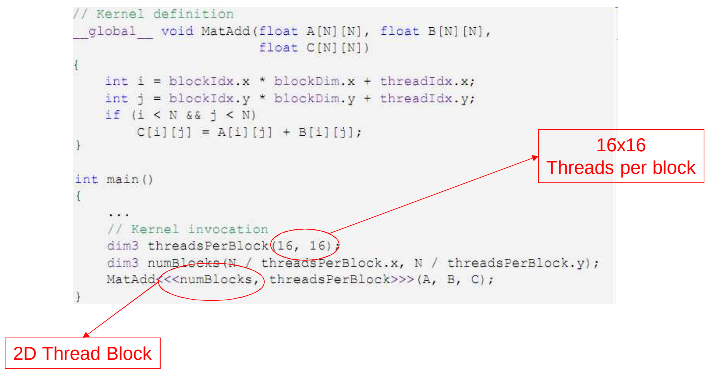


Image from: http://courses.engr.illinois.edu/ece498/al/textbook/Chapter2-CudaProgrammingModel.pdf

Thread Hierarchies

- Block Index: `blockIdx`
- Dimension: `blockDim`
  - 1D or 2D

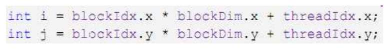
Thread Hierarchies



```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```

Thread Hierarchies

- Example: N = 32
  - 16x16 threads per block (independent of N)
    - `threadIdx` ([0, 15], [0, 15])
  - 2x2 thread blocks in grid
    - `blockIdx` ([0, 1], [0, 1])
    - `blockDim` = 16



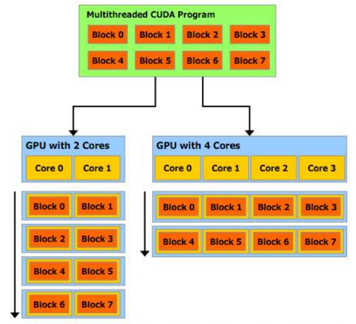
```
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
```

- `i = [0, 1] * 16 + [0, 15]`

Thread Hierarchies

- Thread blocks execute independently
  - In any order: parallel or series
  - Scheduled in any order by any number of cores
    - Allows code to scale with core count

Thread Hierarchies



A multithreaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more cores will automatically execute the program in less time than a GPU with fewer cores.

Figure 1-4. Automatic Scalability

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

Thread Hierarchies

- Threads in a block
  - Share (limited) low-latency memory
  - Synchronize execution
    - To coordinate memory accesses
    - `__syncThreads()`
      - Barrier – threads in block wait until all threads reach this
      - Lightweight

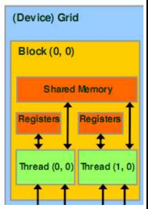


Image from: http://courses.engr.illinois.edu/ece498/al/textbook/Chapter2-CudaProgrammingModel.pdf

CUDA Memory Transfers

- 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

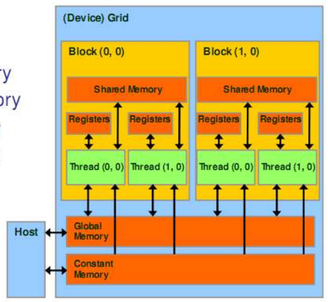


Image from: http://courses.engr.illinois.edu/ece498/al/textbook/Chapter2-CudaProgrammingModel.pdf

CUDA Memory Transfers

- Host can transfer to/from device
  - *Global* memory
  - *Constant* memory

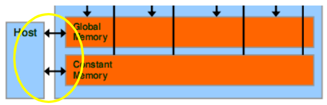


Image from: http://courses.engr.illinois.edu/ece498/al/textbook/Chapter2-CudaProgrammingModel.pdf

CUDA Memory Transfers

- `cudaMalloc()`
  - Allocate global memory on device
- `cudaFree()`
  - Frees memory

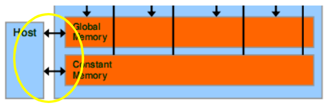


Image from: http://courses.engr.illinois.edu/ece498/al/textbook/Chapter2-CudaProgrammingModel.pdf

CUDA Memory Transfers

```
float *Md
int size = Width * Width * sizeof(float);

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

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

CUDA Memory Transfers

```
float *Md
int size = Width * Width * sizeof(float);

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

Pointer to device memory

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

CUDA Memory Transfers

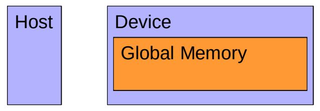
```
float *Md
int size = Width * Width * sizeof(float);

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

Size in bytes

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

CUDA Memory Transfers

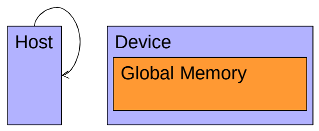
- `cudaMemcpy()`
  - Memory transfer
    - Host to host
    - Host to device
    - Device to host
    - Device to device



- Similar to *buffer objects* in OpenGL

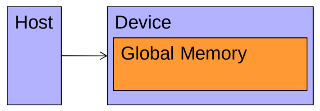
CUDA Memory Transfers

- `cudaMemcpy()`
  - Memory transfer
    - Host to host
    - Host to device
    - Device to host
    - Device to device



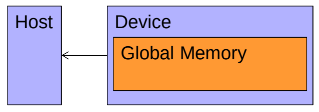
CUDA Memory Transfers

- `cudaMemcpy()`
  - Memory transfer
    - Host to host
    - Host to device
    - Device to host
    - Device to device



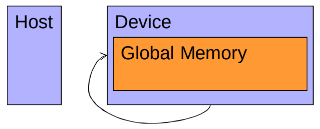
CUDA Memory Transfers

- `cudaMemcpy()`
  - Memory transfer
    - Host to host
    - Host to device
    - Device to host
    - Device to device



CUDA Memory Transfers

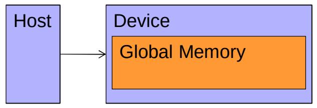
- `cudaMemcpy()`
  - Memory transfer
    - Host to host
    - Host to device
    - Device to host
    - Device to device



CUDA Memory Transfers

Host to device

```
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
```

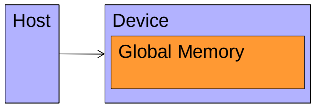


Code from: http://courses.engr.illinois.edu/ece498/al/textbook/Chapter2-CudaProgrammingModel.pdf

CUDA Memory Transfers

Destination (device) Source (host)

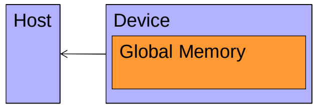
```
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
```



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

CUDA Memory Transfers

```
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
```

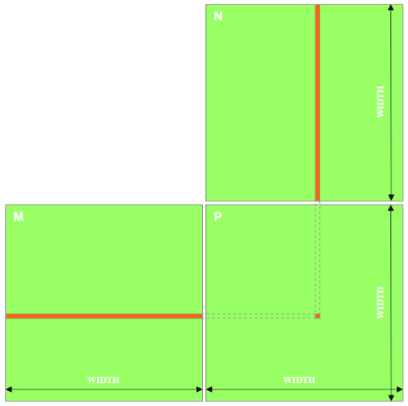


Code from: http://courses.engr.illinois.edu/ece498/al/textbook/Chapter2-CudaProgrammingModel.pdf

Matrix Multiply Reminder

- Vectors
- Dot products
- Row major or column major?
- Dot product per output element

Matrix Multiply



- P = M * N
- Assume M and N are square for simplicity

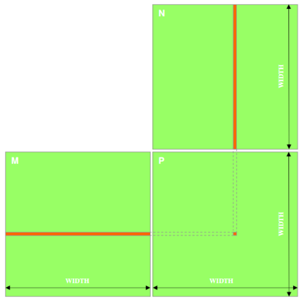
Image from: http://courses.engr.illinois.edu/ece498/al/textbook/Chapter2-CudaProgrammingModel.pdf

Matrix Multiply

- 1,000 x 1,000 matrix
  - 1,000,000 dot products
    - Each 1,000 multiples and 1,000 adds

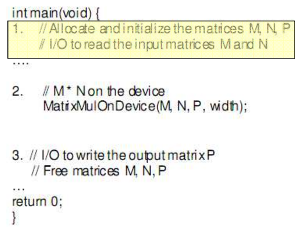
Matrix Multiply: CPU Implementation

```
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 Skeleton



```
int main(void) {
1.    // Allocate and initialize the matrices M, N, P
      // I/O to read the input matrices M and N
....

2.     // M * N on the device
       MatrixMulOnDevice(M, N, P, width);


3. // I/O to write the output matrix P
   // Free matrices M, N, P
...
return 0;
}
```

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

Matrix Multiply: CUDA Skeleton

```
int main(void) {
1.    // Allocate and initialize the matrices M, N, P
      // I/O to read the input matrices M and N
....
2.    // M * N on the device
      MatrixMulOnDevice(M, N, P, width);

3. // I/O to write the output matrix P
   // Free matrices M, N, P
...
return 0;
}
```

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

Matrix Multiply: CUDA Skeleton

```
int main(void) {
1.    // Allocate and initialize the matrices M, N, P
      // I/O to read the input matrices M and N
....
2.    // M * N on the device
      MatrixMulOnDevice(M, N, P, width);

3. // I/O to write the output matrix P
   // Free matrices M, N, P
...
return 0;
}
```

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

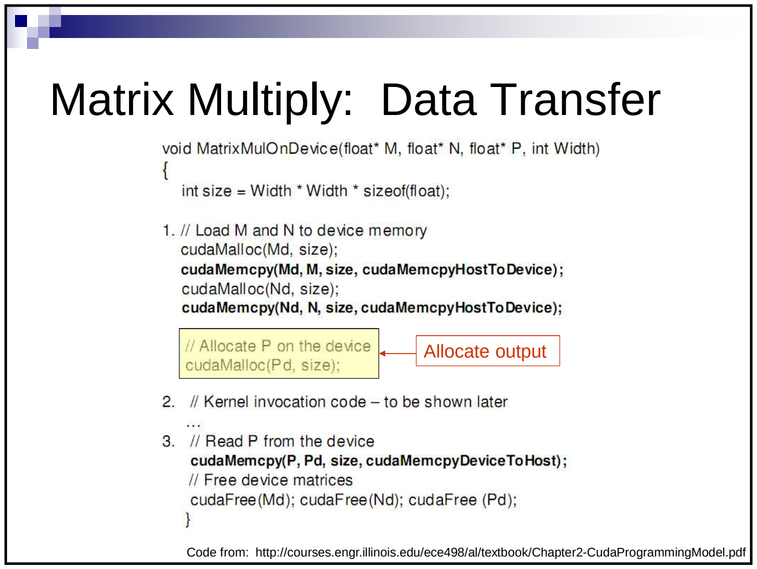
Matrix Multiply

- Step 1
  - Add *CUDA memory transfers* to the skeleton

Matrix Multiply: Data Transfer

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
   int size = Width * Width * sizeof(float);

1. // Load M and N to device memory
   cudaMalloc(Md, size);
   cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);     <- Allocate input
   cudaMalloc(Nd, size);
   cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

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

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);
    }
```

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

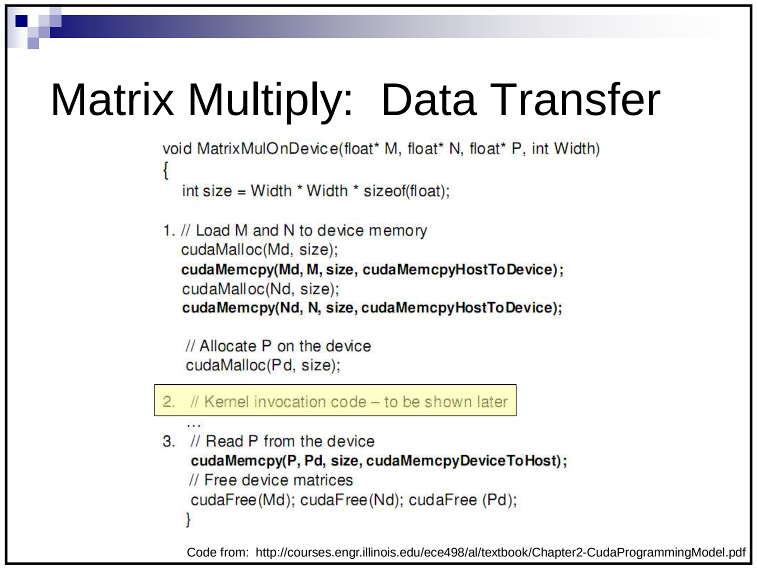


Matrix Multiply: Data Transfer

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
   int size = Width * Width * sizeof(float);

1. // Load M and 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);

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);
    }
```

Allocate output

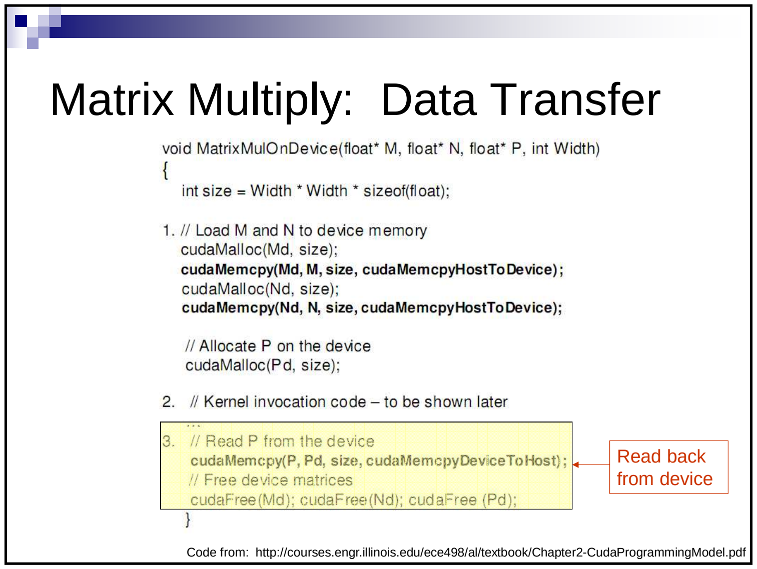
Code from: http://courses.engr.illinois.edu/ece498/al/textbook/Chapter2-CudaProgrammingModel.pdf



Matrix Multiply: Data Transfer

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
   int size = Width * Width * sizeof(float);

1. // Load M and 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);

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);
    }
```

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

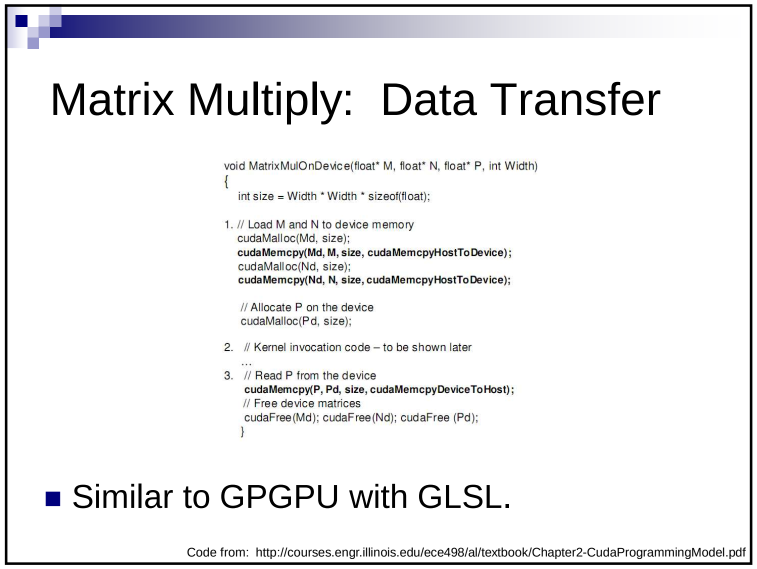


Matrix Multiply: Data Transfer

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
   int size = Width * Width * sizeof(float);

1. // Load M and 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);

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);
    }
```

Read back from device

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



Matrix Multiply: Data Transfer

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
   int size = Width * Width * sizeof(float);

1. // Load M and 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);

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);
    }
```

- Similar to GPGPU with GLSL.

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

Matrix Multiply

- Step 2
  - Implement the *kernel* in CUDA C

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

- Step 3
  - Invoke the *kernel* in CUDA C

Matrix Multiply: Invoke Kernel

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

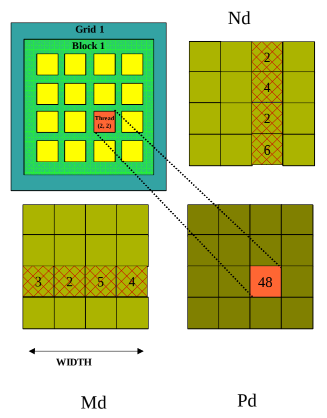
One block with width by width threads

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

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

Matrix Multiply

- 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



© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE 498AL Spring 2010, University of Illinois, Urbana-Champaign

60

Slide from: http://courses.engr.illinois.edu/ece498/al/lectures/lecture2%20cuda%20spring%2009.ppt

Matrix Multiply

- What is the major performance problem with our implementation?
- What is the major limitation?