ECE 498AL

Applied Parallel Programming

Lecture 1: Introduction
Course Goals

- Learn how to program massively parallel processors and achieve
  - high performance
  - functionality and maintainability
  - scalability across future generations
- Acquire technical knowledge required to achieve the above goals
  - principles and patterns of parallel programming
  - processor architecture features and constraints
  - programming API, tools and techniques
Why Massively Parallel Processing?

- A quiet revolution and potential build-up
  - Calculation: TFLOPS vs. 100 GFLOPS
  - Memory Bandwidth: ~10x

- GPU in every PC—massive volume and potential impact

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE 498AL Spring 2010, University of Illinois, Urbana-Champaign
GeForce 8800 (2007)

16 highly threaded SM’s, >128 FPU’s, 367 GFLOPS, 768 MB DRAM, 86.4 GB/S Mem BW, 4GB/S BW to CPU
G80 Characteristics

- 367 GFLOPS peak performance (25-50 times of current high-end microprocessors)
- 265 GFLOPS sustained for apps such as VMD
- Massively parallel, 128 cores, 90W
- Massively threaded, sustains 1000s of threads per app
- 30-100 times speedup over high-end microprocessors on scientific and media applications: medical imaging, molecular dynamics

“I think they're right on the money, but the huge performance differential (currently 3 GPUs ~= 300 SGI Altix Itanium2s) will invite close scrutiny so I have to be careful what I say publically until I triple check those numbers.”

- John Stone, VMD group, Physics UIUC
Fermi (2010)

~1.5TFLOPS (SP) / ~800GFLOPS (DP)
230 GB/s DRAM Bandwidth
Future Apps Reflect a Concurrent World

- Exciting applications in future mass computing market have been traditionally considered “supercomputing applications”
  - Molecular dynamics simulation, Video and audio coding and manipulation, 3D imaging and visualization, Consumer game physics, and virtual reality products
  - These “Super-apps” represent and model physical, concurrent world

- Various granularities of parallelism exist, but…
  - programming model must not hinder parallel implementation
  - data delivery needs careful management
Stretching Traditional Architectures

• Traditional parallel architectures cover some super-applications
  – DSP, GPU, network apps, Scientific
• The game is to grow mainstream architectures “out” or domain-specific architectures “in”
  – CUDA is latter
## Previous Projects

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
<th>Source</th>
<th>Kernel</th>
<th>% time</th>
</tr>
</thead>
<tbody>
<tr>
<td>H.264</td>
<td>SPEC ‘06 version, change in guess vector</td>
<td>34,811</td>
<td>194</td>
<td>35%</td>
</tr>
<tr>
<td>LBM</td>
<td>SPEC ‘06 version, change to single precision and print fewer reports</td>
<td>1,481</td>
<td>285</td>
<td>&gt;99%</td>
</tr>
<tr>
<td>RC5-72</td>
<td>Distributed.net RC5-72 challenge client code</td>
<td>1,979</td>
<td>218</td>
<td>&gt;99%</td>
</tr>
<tr>
<td>FEM</td>
<td>Finite element modeling, simulation of 3D graded materials</td>
<td>1,874</td>
<td>146</td>
<td>99%</td>
</tr>
<tr>
<td>RPES</td>
<td>Rye Polynomial Equation Solver, quantum chem, 2-electron repulsion</td>
<td>1,104</td>
<td>281</td>
<td>99%</td>
</tr>
<tr>
<td>PNS</td>
<td>Petri Net simulation of a distributed system</td>
<td>322</td>
<td>160</td>
<td>&gt;99%</td>
</tr>
<tr>
<td>SAXPY</td>
<td>Single-precision implementation of saxpy, used in Linpack’s Gaussian elim. routine</td>
<td>952</td>
<td>31</td>
<td>&gt;99%</td>
</tr>
<tr>
<td>TRACF</td>
<td>Two Point Angular Correlation Function</td>
<td>536</td>
<td>98</td>
<td>96%</td>
</tr>
<tr>
<td>FDTD</td>
<td>Finite-Difference Time Domain analysis of 2D electromagnetic wave propagation</td>
<td>1,365</td>
<td>93</td>
<td>16%</td>
</tr>
<tr>
<td>MRI-Q</td>
<td>Computing a matrix Q, a scanner’s configuration in MRI reconstruction</td>
<td>490</td>
<td>33</td>
<td>&gt;99%</td>
</tr>
</tbody>
</table>

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE 498AL Spring 2010, University of Illinois, Urbana-Champaign
• GeForce 8800 GTX vs. 2.2GHz Opteron 248
• 10× speedup in a kernel is typical, as long as the kernel can occupy enough parallel threads
• 25× to 400× speedup if the function’s data requirements and control flow suit the GPU and the application is optimized
• “Need for Speed” Seminar Series organized by Patel and Hwu from Spring 2009
ECE 498AL

Lecture 2:
The CUDA Programming Model
Parallel Programming Basics

• Things we need to consider:
  – Control
  – Synchronization
  – Communication

• Parallel programming languages offer different ways of dealing with above
What is (Historical) GPGPU?

- General Purpose computation using GPU and graphics API in applications other than 3D graphics
  - GPU accelerates critical path of application

- Data parallel algorithms leverage GPU attributes
  - Large data arrays, streaming throughput
  - Fine-grain SIMD parallelism
  - Low-latency floating point (FP) computation

- Applications – see //GPGPU.org
  - Game effects (FX) physics, image processing
  - Physical modeling, computational engineering, matrix algebra, convolution, correlation, sorting

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

- Dealing with graphics API
  - Working with the corner cases of the graphics API
- Addressing modes
  - Limited texture size/dimension
- Shader capabilities
  - Limited outputs
- Instruction sets
  - Lack of Integer & bit ops
- Communication limited
  - Between pixels
  - Scatter \( a[i] = p \)
CUDA

“Compute Unified Device Architecture”

General purpose programming model
- User kicks off batches of threads on the GPU
- GPU = dedicated super-threaded, massively data parallel co-processor

Targeted software stack
- Compute oriented drivers, language, and tools

Driver for loading computation programs into GPU
- Standalone Driver - Optimized for computation
- Interface designed for compute – graphics-free API
- Data sharing with OpenGL buffer objects
- Guaranteed maximum download & readback speeds
- Explicit GPU memory management
An Example of Physical Reality Behind CUDA
Parallel Computing on a GPU

- 8-series GPUs deliver 25 to 200+ GFLOPS on compiled parallel C applications
  - Available in laptops, desktops, and clusters
- GPU parallelism is doubling every year
- Programming model scales transparently
- Programmable in C with CUDA tools
- Multithreaded SPMD model uses application data parallelism and thread parallelism
Overview

• CUDA programming model – basic concepts and data types

• CUDA application programming interface - basic

• Simple examples to illustrate basic concepts and functionalities

• Performance features will be covered later
CUDA – C with no shader limitations!

- Integrated host+device app C program
  - Serial or modestly parallel parts in **host** C code
  - Highly parallel parts in **device** SPMD kernel C code

```
Serial Code (host)

Parallel Kernel (device)
KernelA<<< nBlk, nTid >>>(args);

Serial Code (host)

Parallel Kernel (device)
KernelB<<< nBlk, nTid >>>(args);
```
CUDA Devices and Threads

• A compute device
  – Is a coprocessor to the CPU or host
  – Has its own DRAM (device memory)
  – Runs many threads in parallel
  – Is typically a GPU but can also be another type of parallel processing device

• Data-parallel portions of an application are expressed as device kernels which run on many 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
G80 – Graphics Mode

- The future of GPUs is programmable processing
- So – build the architecture around the processor

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE 498AL Spring 2010, University of Illinois, Urbana-Champaign
G80 CUDA mode – A Device Example
Extended C

• **Type Qualifiers**
  - global, device, shared, local, constant

• **Keywords**
  - threadIdx, blockIdx

• **Intrinsics**
  - __syncthreads

• **Runtime API**
  - Memory, symbol, execution management

• **Function launch**

```c
__device__ float filter[N];
__global__ void convolve (float *image) {
    __shared__ float region[M];
    ...
    region[threadIdx] = image[i];
    __syncthreads()
    ...
    image[j] = result;
}
```

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)

// 100 blocks, 10 threads per block
convolve<<<100, 10>>>(myimage);
Extended C

- Integrated source
  - (foo.cu)
- cudacc
  - EDG C/C++ frontend
  - Open64 Global Optimizer
- GPU Assembly
  - foo.s
- OCG
- G80 SASS
  - foo.sass
- CPU Host Code
  - foo.cpp
- gcc / cl

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

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

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

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

- The API is an extension to the ANSI C programming language
  - Low learning curve

- The hardware is designed to enable lightweight runtime and driver
  - High performance
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**

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

```c
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

```c
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
```

```c
cudaMemcpy(M, Md, size, cudaMemcpyDeviceToHost);
```
CUDA Keywords
### CUDA Function Declarations

<table>
<thead>
<tr>
<th></th>
<th>Executed on the:</th>
<th>Only callable from the:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__device__</code></td>
<td>device</td>
<td>device</td>
</tr>
<tr>
<td>float DeviceFunc()</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>__global__</code></td>
<td>device</td>
<td>host</td>
</tr>
<tr>
<td>void KernelFunc()</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>__host__</code></td>
<td>host</td>
<td>host</td>
</tr>
<tr>
<td>float HostFunc()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- `__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:

```c
__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 synth 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

- \( P = M \times N \) of size \( WIDTH \times WIDTH \)

- 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

\[
\begin{array}{cccc}
M_{0,0} & M_{0,1} & M_{0,2} & M_{0,3} \\
M_{1,0} & M_{1,1} & M_{1,2} & M_{1,3} \\
M_{2,0} & M_{2,1} & M_{2,2} & M_{2,3} \\
M_{3,0} & M_{3,1} & M_{3,2} & M_{3,3}
\end{array}
\]
// 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;
        }
}
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
       cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
       cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

    ... 

    1. // Allocate P on the device
       cudaMemcpy(Pd, P, size, cudaMemcpyHostToDevice);
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 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
Step 7: 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)!
Some Useful Information on Tools
Compiling a CUDA Program

- Parallel Thread eXecution (PTX)
  - Virtual Machine and ISA
  - Programming model
  - Execution resources and state

```
float4 me = gx[gtid];
me.x += me.y * me.z;
```

```
ld.global.v4.f32 {{f1,f3,f5,f7}, [$r9+0];
mad.f32 $f1, $f5, $f3, $f1;
```
Compilation

• Any source file containing CUDA language extensions must be compiled with NVCC

• NVCC is a compiler driver
  – Works by invoking all the necessary tools and compilers like cudacc, g++, cl, ...

• NVCC outputs:
  – C code (host CPU Code)
    • Must then be compiled with the rest of the application using another tool
  – PTX
    • Object code directly
    • Or, PTX source, interpreted at runtime
Linking

• Any executable with CUDA code requires two dynamic libraries:
  – The CUDA runtime library (cudart)
  – The CUDA core library (cuda)
Debugging Using the Device Emulation Mode

• An executable compiled in device emulation mode (nvcc -deviceemu) runs completely on the host using the CUDA runtime
  – No need of any device and CUDA driver
  – Each device thread is emulated with a host thread

• Running in device emulation mode, one can:
  – Use host native debug support (breakpoints, inspection, etc.)
  – Access any device-specific data from host code and vice-versa
  – Call any host function from device code (e.g. `printf`) and vice-versa
  – Detect deadlock situations caused by improper usage of `__syncthreads`
Device Emulation Mode Pitfalls

• Emulated device threads execute sequentially, so simultaneous accesses of the same memory location by multiple threads could produce different results.

• Dereferencing device pointers on the host or host pointers on the device can produce correct results in device emulation mode, but will generate an error in device execution mode.
Floating Point

• **Results of floating-point computations** will slightly differ because of:
  – Different compiler outputs, instruction sets
  – Use of extended precision for intermediate results
    • There are various options to force strict single precision on the host
Lecture 3: A Simple Example, Tools, and CUDA Threads
/ // Matrix multiplication on the (CPU) host
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;
        }
}
Step 2: Input Matrix Data Transfer (Host-side Code)

```c
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 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
Step 7: 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

You still need to put a loop around the kernel call for cases where \(WIDTH/TILE\_WIDTH\) is greater than max grid size (64K)!
Some Useful Information on Tools
Compiling a CUDA Program

- Parallel Thread eXecution (PTX)
  - Virtual Machine and ISA
  - Programming model
  - Execution resources and state

```
float4 me = gx[gtid];
me.x += me.y * me.z;
```

```
ld.global.v4.f32  {$f1,$f3,$f5,$f7}, [$r9+0];
mad.f32           $f1, $f5, $f3, $f1;
```
Compilation

- Any source file containing CUDA language extensions must be compiled with NVCC
- NVCC is a compiler driver
  - Works by invoking all the necessary tools and compilers like cudacc, g++, cl, ...
- NVCC outputs:
  - C code (host CPU Code)
    - Must then be compiled with the rest of the application using another tool
  - PTX
    - Object code directly
    - Or, PTX source, interpreted at runtime
Linking

• Any executable with CUDA code requires two dynamic libraries:
  – The CUDA runtime library (cudart)
  – The CUDA core library (cuda)
Debugging Using the Device Emulation Mode

• An executable compiled in device emulation mode (nvcc -deviceemu) runs completely on the host using the CUDA runtime
  - No need of any device and CUDA driver
  - Each device thread is emulated with a host thread

• Running in device emulation mode, one can:
  - Use host native debug support (breakpoints, inspection, etc.)
  - Access any device-specific data from host code and vice-versa
  - Call any host function from device code (e.g. printf) and vice-versa
  - Detect deadlock situations caused by improper usage of
Device Emulation Mode Pitfalls

• Emulated device threads execute sequentially, so simultaneous accesses of the same memory location by multiple threads could produce different results.

• Dereferencing device pointers on the host or host pointers on the device can produce correct results in device emulation mode, but will generate an error in device execution mode.
Floating Point

• Results of floating-point computations will slightly differ because of:
  – Different compiler outputs, instruction sets
  – Use of extended precision for intermediate results
    • There are various options to force strict single precision on the host
CUDA Threads
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
  - …
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)

Block(0,1)  Block(1,1)

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
A Small Example: Multiplication
Revised Matrix Multiplication Kernel using Multiple Blocks

```c
__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 index 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;
}
```

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

• Hardware is free to assigns blocks to any processor at any time
  – A kernel scales across any number of parallel processors

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
  - 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

Instruction:

<table>
<thead>
<tr>
<th></th>
<th>TB1 W1</th>
<th>TB2 W1</th>
<th>TB3 W1</th>
<th>TB1 W2</th>
<th>TB2 W2</th>
<th>TB3 W2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

TB = Thread Block, W = Warp
G80 Block Granularity Considerations

• For Matrix Multiplication using multiple blocks, should I use 8X8, 16X16 or 32X32 blocks?

  – For 8X8, we have 64 threads per Block. Since each SM can take up to 768 threads, there are 12 Blocks. However, each SM can only take up to 8 Blocks, only 512 threads will go into each SM!

  – For 16X16, we have 256 threads per Block. Since each SM can take up to 768 threads, it can take up to 3 Blocks and achieve full capacity unless other resource considerations overrule.

  – For 32X32, we have 1024 threads per Block. Not even one can fit into an SM!
Some Additional API Features
Application Programming Interface

- The API is an extension to the C programming language
- It consists of:
  - Language extensions
    - To target portions of the code for execution on the device
  - A runtime library split into:
    - A common component providing built-in vector types and a subset of the C runtime library in both host and device codes
    - A host component to control and access one or more devices from the host
Language Extensions: Built-in Variables

- \texttt{dim3\ gridDim};
  - Dimensions of the grid in blocks
    (\texttt{gridDim.z} unused)
- \texttt{dim3\ blockDim};
  - Dimensions of the block in threads
- \texttt{dim3\ blockIdx};
  - Block index within the grid
- \texttt{dim3\ threadIdx};
  - Thread index within the block
Common Runtime Component: Mathematical Functions

- `pow`, `sqrt`, `cbrt`, `hypot`
- `exp`, `exp2`, `expm1`
- `log`, `log2`, `log10`, `log1p`
- `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`
- `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`
- `ceil`, `floor`, `trunc`, `round`
- Etc.

- When executed on the host, a given function uses the C runtime implementation if available.
- These functions are only supported for scalar types, not vector types.
Device Runtime Component: Mathematical Functions

- Some mathematical functions (e.g. \( \sin(x) \)) have a less accurate, but faster device-only version (e.g. \( \_\_\sin(x) \))
  - \( \_\_\text{pow} \)
  - \( \_\_\log, \_\_\log2, \_\_\log10 \)
  - \( \_\_\exp \)
  - \( \_\_\sin, \_\_\cos, \_\_\tan \)
Host Runtime Component

• Provides functions to deal with:
  – **Device** management (including multi-device systems)
  – **Memory** management
  – **Error** handling

• Initializes the first time a runtime function is called

• A host thread can invoke device code on only one device
Device Runtime Component: Synchronization Function

• `void __syncthreads();`
• Synchronizes all threads in a block
• Once all threads have reached this point, execution resumes normally
• Used to avoid RAW / WAR / WAW hazards when accessing shared or global memory
• Allowed in conditional constructs only if the conditional is uniform across the entire thread block