Objective

• To understand the organization and scheduling of threads
  – Resource assignment at the block level
  – Scheduling at the warp level
  – Implementation of SIMT execution
Reading Assignment

- Kirk and Hwu, “Programming Massively Parallel Processors: A Hands on Approach,”, Chapter 4
- CUDA Programming Guide

A Multi-Dimensional Grid Example
Built-In Variables

• 1D-3D Grid of thread blocks
  – Built-in: gridDim
    • gridDim.x, gridDim.y, gridDim.z
  – Built-in: blockDim
    • blockDim.x, blockDim.y, blockDim.z

Example
• dim3 dimGrid (32,2,2) - 3D grid of thread blocks
• dim3 dimGrid (2,2,1) - 2D grid of thread blocks
• dim3 dimGrid (32,1,1) - 1D grid of thread blocks
• dim3 dimGrid ((n/256.0),1,1) - 1D grid of thread blocks

my_kernel<<<dimGrid, dimBlock>>>(..)

Built-In Variables (2)

• 1D-3D grid of threads in a thread block
  – Built-in: blockIdx
    • blockIdx.x, blockIdx.y, blockIdx.z
  – Built-in: threadIdx
    • threadIdx.x, threadIdx.y, threadIdx.z
  – All blocks have the same thread configuration

Example
• dim3 dimBlock (4,2,2) - 3D grid of thread blocks
• dim3 dimBlock (2,2,1) - 2D grid of thread blocks
• dim3 dimBlock (32,1,1) - 1D grid of thread blocks

my_kernel<<<dimGrid, dimBlock>>>(..)
Built-In Variables (3)

- 1D-3D grid of threads in a thread block
  - Built-in: blockIdx
    - blockIdx.x, blockIdx.y, blockIdx.z
  - Built-in: threadIdx
    - threadIdx.x, threadIdx.y, threadIdx.z

- Initialized by the runtime through a kernel call
- Range fixed by the compute capability and target devices

Processing a Picture with a 2D Grid

16×16 blocks
72x62 pixels
Row-Major Layout in C/C++

\[
\begin{array}{cccc}
M_{0,0} & M_{0,1} & M_{0,2} & M_{0,3} \\
M_{1,0} & M_{1,1} & M_{1,2} & M_{1,3} \\
M_{2,0} & M_{2,1} & M_{2,2} & M_{2,3} \\
M_{3,0} & M_{3,1} & M_{3,2} & M_{3,3} \\
\end{array}
\]

\[\text{Row} \times \text{Width} + \text{Col} = 2 \times 4 + 1 = 9\]

Source Code of the Picture Kernel

```c
__global__ void PictureKernel(float* d_Pin, float* d_Pout, int n, int m) {
    // Calculate the row # of the d_Pin and d_Pout element to process
    int Row = blockIdx.y * blockDim.y + threadIdx.y;
    // Calculate the column # of the d_Pin and d_Pout element to process
    int Col = blockIdx.x * blockDim.x + threadIdx.x;

    // each thread computes one element of d_Pout if in range
    if ((Row < m) && (Col < n)) {
        d_Pout[Row*n+Col] = 2*d_Pin[Row*n+Col];
    }
}
```
Approach Summary

- Storage layout of data
- Assign unique ID
- Map IDs to Data (access)

Figure 4.5 Covering a 76×62 picture with 16×blocks.

A Simple Running Example
Matrix Multiplication

- A simple illustration of the basic features of memory and thread management in CUDA programs
  - Thread index usage
  - Memory layout
  - Register usage
  - Assume square matrix for simplicity
  - Leave shared memory usage until later

© David Kirk/NVIDIA and Wen-mei Hwu, 2007-2012     ECE408/CS483/ECE498sal, University of Illinois, Urbana-Champaign
Square Matrix-Matrix Multiplication

- \( P = M \times N \) of size \( WIDTH \times WIDTH \)
  - Each thread calculates one element of \( P \)
  - Each row of \( M \) is loaded \( WIDTH \) times from global memory
  - Each column of \( N \) is loaded \( WIDTH \) times from global memory

Row-Major Layout in C/C++

\[
\text{Row} \times \text{Width} + \text{Col} = \text{2} \times \text{4} + \text{1} = \text{9}
\]
Matrix Multiplication

A Simple Host Version in C

// Matrix multiplication on the (CPU) host in double precision
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; ++i)
        for (int j = 0; j < Width; ++j) {map
            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;
    }
}

Kernel Version: Functional Description

for (int k = 0; k < Width; ++k)
    Pvalue += d_M[Row*Width+k] *
        d_N[k*Width+Col];

• Which thread is at coordinate (row,col)?
• Threads self allocate
Kernel Function - A Small Example

• Have each 2D thread block to compute a \((\text{TILE\_WIDTH})^2\) sub-matrix (tile) of the result matrix
  – Each has \((\text{TILE\_WIDTH})^2\) threads
• Generate a 2D Grid of \((\text{WIDTH}/\text{TILE\_WIDTH})^2\) blocks

WIDTH = 4; TILE\_WIDTH = 2
Each block has \(2^2 = 4\) threads

WIDTH/TILE\_WIDTH = 2
Use \(2^2 = 4\) blocks

A Slightly Bigger Example

WIDTH = 8; TILE\_WIDTH = 2
Each block has \(2^2 = 4\) threads

WIDTH/TILE\_WIDTH = 4
Use \(4^2 = 16\) blocks
A Slightly Bigger Example (cont.)

WIDTH = 8; TILE_WIDTH = 4
Each block has 4*4 = 16 threads

WIDTH/TILE_WIDTH = 2
Use 2*2 = 4 blocks

---

Kernel Invocation (Host-side Code)

// Setup the execution configuration
// TILE_WIDTH is a #define constant

dim3 dimGrid(Width/TILE_WIDTH, Width/TILE_WIDTH, 1);
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);

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

// Matrix multiplication kernel – per thread code
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width) {
    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;

    Col = 0 * 2 + threadIdx.x
    Row = 0 * 2 + threadIdx.y

    Work for Block (0,0)
in a TILE_WIDTH = 2 Configuration

    blockDim.x
    blockDim.y

    Col = 0 * 2 + blockIdx.x
    Row = 0 * 2 + blockIdx.y

    Row = 0
    Row = 1

    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}

    N_{0,0} N_{0,1} N_{0,2} N_{0,3}
    N_{1,0} N_{1,1} N_{1,2} N_{1,3}
    N_{2,0} N_{2,1} N_{2,2} N_{2,3}
    N_{3,0} N_{3,1} N_{3,2} N_{3,3}

    P_{0,0} P_{0,1} P_{0,2} P_{0,3}
    P_{1,0} P_{1,1} P_{1,2} P_{1,3}
    P_{2,0} P_{2,1} P_{2,2} P_{2,3}
    P_{3,0} P_{3,1} P_{3,2} P_{3,3}
A Simple Matrix Multiplication Kernel

```c
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width) {
    int Row = blockIdx.y * blockDim.y + threadIdx.y;
    int Col = blockIdx.x * blockDim.x + threadIdx.x;

    if ((Row < Width) && (Col < Width)) {
        float Pvalue = 0;
        for (int k = 0; k < Width; ++k)
            Pvalue += d_M[Row * Width + k] * d_N[k * Width + Col];
        d_P[Row * Width + Col] = Pvalue;
    }
}
```
Kernel Execution Model

- Thread blocks and warps
- Synchronization
- Mapping to hardware execution units
- Scheduling and resource management
- Programmer’s view

CUDA Thread Block

- All threads in a block execute the same kernel program (SPMD)
- Programmer declares block:
  - Block size 1 to 1024 concurrent threads
  - Block shape 1D, 2D, or 3D
  - Block dimensions in threads
- Threads have thread index numbers within block
  - Kernel code uses thread index and block index to select work and address shared data
- Threads in the same block share data and synchronize while doing their share of the work

© David Kirk/NVIDIA and Wen-mei Hwu, 2007-2012 ECE408 / CS483 / ECE498al, University of Illinois, Urbana-Champaign
Execution Semantics

- Execution of a thread block partitioned into SIMD warps

Example: Vector Code

PTX (Assembly):

```plaintext
setp.lt.s32 %p, %r5, %rd4; // r5 = index, rd4 = N
@p bra L1;
bra L2;
L1:
ld.global.f32 %f1, [%r6]; // r6 = &a[index]
ld.global.f32 %f2, [%r7]; // r7 = &b[index]
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3; // r8 = &c[index]
L2:
ret;
```

© David Kirk/NVIDIA and Wen-mei Hwu, 2007-2012  ECE408/ CS483/ECE498al, University of Illinois, Urbana-Champaign
Execution Semantics (2)

- All threads must reach the barrier, otherwise indefinite wait
  - Note challenges with conditional statements
  - Note each barrier statement represents a distinct barrier
- Barriers only within a thread block
- No synchronization across thread blocks
  - Can be enforced via atomics
- Thread blocks can execute in any order
  - Loose synchronization can enhance scalability

Barrier Synchronization

- Impact on scalability
  - No constraints on execution order of thread blocks
- Impact on portability
  - Resource allocation on a per thread block basis
  - Launch thread blocks
Example: Executing Thread Blocks

• Threads are assigned to Streaming Multiprocessors in block granularity
  – Up to 8 blocks to each SM as resource allows
  – Fermi SM can take up to 1536 threads
    • Could be 256 (threads/block) * 6 blocks
    • Or 512 (threads/block) * 3 blocks, etc.

Threads run concurrently
SM maintains thread/block id #s
SM manages/schedules thread execution

Transparent Scalability

• Hardware is free to assign 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.
How thread blocks are partitioned

- Thread blocks are partitioned into warps
  - Thread IDs within a warp are consecutive and increasing
  - Warp 0 starts with Thread ID 0

- Partitioning is always the same
  - Thus you can use this knowledge in control flow
  - However, the exact size of warps may change from generation to generation
  - (Covered next)

- However, DO NOT rely on any ordering between warps
  - If there are any dependencies between threads, you must __syncthreads__() to get correct results (more later).

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
Example: Thread Scheduling (Cont.)

- SM implements zero-overhead warp scheduling
  - At any time, 1 or 2 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

Instructions:

```
  | T^1 | W^1 | T^2 | W^1 | T^3 | W^1 | T^2 | W^2 | T^1 | W^-1 | T^1 | W^2 | T^3 | W^2 |
--|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
  | 1   | 2   | 3   | 4   | 5   | 6   | 1   | 2   | 1   | 2   | 1   | 2   | 3   | 4   |
```

- Time

TB = Thread Block, W = Warp

Going back to the program

- Every instruction needs to be fetched from memory, decoded, then executed.
- Instructions come in three flavors: Operate, Data transfer, and Program Control Flow.
- An example instruction cycle is the following:

  Fetch | Decode | Execute | Memory
Control Flow Instructions

- Main performance concern with branching is divergence
  - Threads within a single warp take different paths
  - Different execution paths are serialized in current GPUs
    - The control paths taken by the threads in a warp are traversed one at a time until there is no more.
- A common case: avoid divergence when branch condition is a function of thread ID
  - Example with divergence:
    - If (threadIdx.x > 2) { }
    - This creates two different control paths for threads in a block
    - Branch granularity < warp size; threads 0, 1 and 2 follow different path than the rest of the threads in the first warp
  - Example without divergence:
    - If (threadIdx.x / WARP_SIZE > 2) { }
    - Also creates two different control paths for threads in a block
    - Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path

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 1536 threads, there are 24 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 1536 threads, it can take up to 6 Blocks and achieve full capacity unless other resource considerations overrule.
  - For 32X32, we would have 1024 threads per Block. Only one block can fit into an SM for Fermi. Using only 2/3 of the thread capacity of an SM. Also, this works for CUDA 3.0 and beyond but too large for some early CUDA versions.
Selecting Program Configurations

- Select parameters to maximize resource usage
  - Maximize #TBs or #threads? Equivalent?
  - Does this maximize performance?

- Which resource do you run out of first?

- Target-specific tuning of program configurations
  - Query device properties

Querying the Device

- Get device properties
  - #SMs
  - Max Threads/Block
  - Max Threads/Dim
  - Warp size

- Tune kernel properties to maximize efficiency of execution
Device Properties Example

```c
cudaError_t cudaGetDeviceProperties ( struct cudaDeviceProp * prop, 
int device )
```

Returns in *prop* the properties of device *dev*. The `cudaDeviceProp` structure is defined as:

```c
struct cudaDeviceProp {
    char name[256];
    size_t totalGlobalMem;
    size_t sharedMemPerBlock;
    int regsPerBlock;
    int warpSize;
    size_t memPitch;
    int maxThreadsPerBlock;
    int maxThreadsDim[3];
    int maxGridSize[3];
    int clockRate;
    size_t totalConstMem;
    int major;
    int minor;
    size_t texMemAlignment;
    size_t texturePitchAlignment;
    int deviceOverlap;
    int multiProcessorCount;
}
```


ANY MOE QUESTIONS?
READ CHAPTER 4!
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
    • A device component providing device-specific functions
Common Runtime Component: Mathematical Functions

- `pow`, `sqrt`, `cbrt`, `hypot`
- `exp`, `exp2`, `expml`
- `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)`)
  - `__pow`
  - `__log`, `__log2`, `__log10`
  - `__exp`
  - `__sin`, `__cos`, `__tan`