Objective

- A more detailed look at kernel execution
  - Data to thread assignment
- To understand the organization and scheduling of threads
  - Resource assignment at the block level
  - Scheduling at the warp level
  - Basics of SIMT execution
Reading Assignment

- Kirk and Hwu, “Programming Massively Parallel Processors: A Hands on Approach,”, Chapter 4
- Kirk and Hwu, “Programming Massively Parallel Processors: A Hands on Approach,”, Chapter 6.3

- Reference: CUDA Programming Guide

A Multi-Dimensional Grid Example

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

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

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

\texttt{my\_kernel<<<dimGrid, dimBlock>>>(..)}

Built-In Variables (2)

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

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

\texttt{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
  – You can query the device (later)

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

2D Examples
Processing a Picture with a 2D Grid

16×16 blocks
72x62 pixels

Row-Major Layout in C/C++

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

Square Matrix-Matrix Multiplication

- \( P = M \times N \) of size \( \text{WIDTH} \times \text{WIDTH} \)
  - Each thread calculates one element of \( P \)
  - Each row of \( M \) is loaded \( \text{WIDTH} \) times from global memory
  - Each column of \( N \) is loaded \( \text{WIDTH} \) times from global memory
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 \((TILE\_WIDTH)^2\) sub-matrix (tile) of the result matrix
  - Each has \((TILE\_WIDTH)^2\) threads
- Generate a 2D Grid of \((WIDTH/TILE\_WIDTH)^2\) blocks

© David Kirk/NVIDIA and Wen-mei W. Hwu, Urbana, July 9-13
A Slightly Bigger Example

WIDTH = 8; TILE_WIDTH = 2
Each block has 2\times2 = 4 threads

WIDTH/TILE_WIDTH = 4
Use \(4 \times 4 = 16\) blocks

A Slightly Bigger Example (cont.)

WIDTH = 8; TILE_WIDTH = 4
Each block has 4\times4 = 16 threads

WIDTH/TILE_WIDTH = 2
Use 2\times2 = 4 blocks
// 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);

// 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;

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

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

Work for Block (0,1)

Col = 1 * 2 + threadIdx.x
Row = 0 * 2 + threadIdx.y
A Simple Matrix Multiplication Kernel

__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width)
{
    // Calculate the row index of the d_P element and d_M
    int Row = blockIdx.y*blockDim.y+threadIdx.y;
    // Calculate the column index of d_P and d_N
    int Col = blockIdx.x*blockDim.x+threadIdx.x;

    if ((Row < Width) && (Col < Width)) {
        float Pvalue = 0;
        // each thread computes one element of the block sub-matrix
        for (int k = 0; k < Width; ++k)
            Pvalue += d_M[Row*Width+k] * d_N[k*Width+Col];
        d_P[Row*Width+Col] = Pvalue;
    }
}

QUESTIONS?