Objective

• To understand the organization and scheduling of threads
  – Resource assignment at the block level
  – Scheduling at the warp level
  – SIMD implementation of SIMT execution
A Multi-Dimensional Grid Example

host

Kernel 1

device

Grid 1

Block (0, 0)

Block (1, 0)

Block (0, 1)

Block (1, 1)

Grid 2

Block (1, 1)

Thread (0,0,0)

Thread (0,0,1)

Thread (0,1,0)

Thread (0,1,1)

Thread (0,0,2)

Thread (0,0,3)

Thread (0,1,2)

Thread (0,1,3)
Processing a Picture with a 2D Grid

16×16 blocks
Multidimensional (Nested) Arrays

- **Declaration**
  
  \[ T \ A[R][C]; \]
  
  - 2D array of data type \( T \)
  - \( R \) rows, \( C \) columns
  - Type \( T \) element requires \( K \) bytes

- **Array Size**
  
  - \( R \times C \times K \) bytes

- **Arrangement**
  
  - Row-Major Ordering

\[
\begin{array}{cccc}
A[0][0] & \cdots & A[0][C-1] \\
\vdots & & \vdots \\
A[R-1][0] & \cdots & A[R-1][C-1]
\end{array}
\]
Nested Array Row Access

• Row Vectors
  – \( A[i] \) is array of \( C \) elements
  – Each element of type \( T \) requires \( K \) bytes
  – Starting address \( A + i \times (C \times K) \)

```c
int A[R][C];
```

\[
\begin{array}{c}
\text{A[0]} \\
\mid \mid \\
\hline \\
A [0] [0] & \ldots & A [0] [C-1] \\
\mid \mid \\
\hline \\
A \\
\end{array}
\]

\[
\begin{array}{c}
\text{A[i]} \\
\mid \mid \\
\hline \\
A [i] [0] & \ldots & A [i] [C-1] \\
\mid \mid \\
\hline \\
A + i \times C \times 4 \\
\end{array}
\]

\[
\begin{array}{c}
\text{A[R-1]} \\
\mid \mid \\
\hline \\
A [R-1] [0] & \ldots & A [R-1] [C-1] \\
\mid \mid \\
\hline \\
A + (R-1) \times C \times 4 \\
\end{array}
\]
Strange Referencing Examples

- Reference Address

Value Guaranteed?

ec[3][3]
ec[2][5]
ec[2][−1]
ec[4][−1]
ec[0][19]
ec[0][−1]

Will disappear
__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];
    }
}

Source Code of the PictureKernel
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
Row-Major Layout in C/C++

Row*Width+Col = 2*4+1 = 9
/ * Matrix multiplication on the (CPU) host *
void MatrixMulOnHost(double* M, double* N,
    double* 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;
        }
    }
}
Kernel Function - A Small Example

- Main strategy: 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

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

WIDTH = 4; TILE\_WIDTH = 2
Each block has 2*2 = 4 threads

WIDTH/TILE\_WIDTH = 2
Use 2* 2 = 4 blocks

- What if matrix is not square?
- What if width is not a multiple of TILE\_WIDTH?
A Slightly Bigger Example

Each block has $2 \times 2 = 4$ threads

$\text{WIDTH} = 8$;  $\text{TILE WIDTH} = 2$

Use $4 \times 4 = 16$ blocks
A Slightly Bigger Example (cont.)

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

WIDTH / TILE_WIDTH = 2
Use $2 \times 2 = 4$ blocks

<table>
<thead>
<tr>
<th>P_{0,0}</th>
<th>P_{0,1}</th>
<th>P_{0,2}</th>
<th>P_{0,3}</th>
<th>P_{0,4}</th>
<th>P_{0,5}</th>
<th>P_{0,6}</th>
<th>P_{0,7}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P_{1,0}</td>
<td>P_{1,1}</td>
<td>P_{1,2}</td>
<td>P_{1,3}</td>
<td>P_{1,4}</td>
<td>P_{1,5}</td>
<td>P_{1,6}</td>
<td>P_{1,7}</td>
</tr>
<tr>
<td>P_{2,0}</td>
<td>P_{2,1}</td>
<td>P_{2,2}</td>
<td>P_{2,3}</td>
<td>P_{2,4}</td>
<td>P_{2,5}</td>
<td>P_{2,6}</td>
<td>P_{2,7}</td>
</tr>
<tr>
<td>P_{3,0}</td>
<td>P_{3,1}</td>
<td>P_{3,2}</td>
<td>P_{3,3}</td>
<td>P_{3,4}</td>
<td>P_{3,5}</td>
<td>P_{3,6}</td>
<td>P_{3,7}</td>
</tr>
<tr>
<td>P_{4,0}</td>
<td>P_{4,1}</td>
<td>P_{4,2}</td>
<td>P_{4,3}</td>
<td>P_{4,4}</td>
<td>P_{4,5}</td>
<td>P_{4,6}</td>
<td>P_{4,7}</td>
</tr>
<tr>
<td>P_{5,0}</td>
<td>P_{5,1}</td>
<td>P_{5,2}</td>
<td>P_{5,3}</td>
<td>P_{5,4}</td>
<td>P_{5,5}</td>
<td>P_{5,6}</td>
<td>P_{5,7}</td>
</tr>
<tr>
<td>P_{6,0}</td>
<td>P_{6,1}</td>
<td>P_{6,2}</td>
<td>P_{6,3}</td>
<td>P_{6,4}</td>
<td>P_{6,5}</td>
<td>P_{6,6}</td>
<td>P_{6,7}</td>
</tr>
<tr>
<td>P_{7,0}</td>
<td>P_{7,1}</td>
<td>P_{7,2}</td>
<td>P_{7,3}</td>
<td>P_{7,4}</td>
<td>P_{7,5}</td>
<td>P_{7,6}</td>
<td>P_{7,7}</td>
</tr>
</tbody>
</table>
// 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(double* d_M, double* d_N, double* 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

Row = 0
Row = 1
Work for Block (0,1)

Col \(= 1 \times 2 + \text{threadIdx.x}\)
Row \(= 0 \times 2 + \text{threadIdx.y}\)

Row = 0
Row = 1
A Simple Matrix Multiplication Kernel

```c
__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;
    }
}
```
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
- Threads in different blocks cannot cooperate
  - Each block can execute in any order relative to other blocks!

Courtesy: John Nickolls, NVIDIA
History of parallelism

• 1\textsuperscript{st} gen - Instructions are executed sequentially in program order, one at a time.

• Example:

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction1</td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td>Memory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Fetch</td>
<td>Decode</td>
</tr>
</tbody>
</table>
• **2\textsuperscript{nd} gen** - Instructions are executed sequentially, in program order, in an assembly line fashion. (pipeline)

• **Example:**

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction 1</td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td>Memory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction 2</td>
<td></td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td>Memory</td>
<td></td>
</tr>
<tr>
<td>Instruction 3</td>
<td></td>
<td></td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td>Memory</td>
</tr>
</tbody>
</table>
History – Instruction Level Parallelism

- $3^{rd}$ gen - Instructions are executed in parallel
- Example code 1:
  
  \[
  \begin{align*}
  c &= b + a; \\
  d &= c + e;
  \end{align*}
  \]

  Non-parallelizable

- Example code 2:
  
  \[
  \begin{align*}
  a &= b + c; \\
  d &= e + f;
  \end{align*}
  \]

  Parallelizable
Instruction Level Parallelism (Cont.)

- Two forms of ILP:
  - Superscalar: At runtime, fetch, decode, and execute multiple instructions at a time. Execution may be out of order.
    
    | Cycle | 1     | 2    | 3     | 4     | 5     |
    |-------|-------|------|-------|-------|-------|
    |       | Fetch | Decode | Execute | Memory |       |
    | Instruction 1 |       |       |       |       |       |
    | Instruction 2 |       |       |       |       |       |
    | Instruction 3 |       | Fetch | Decode | Execute | Memory |
    | Instruction 4 |       | Fetch | Decode | Execute | Memory |

  - VLIW: At compile time, pack multiple, independent instructions in one large instruction and process the large instructions as the atomic units.
History – Cont’d

- 4th gen – Multi-threading: multiple threads are executed in an alternating or simultaneous manner on the same processor/core. (will revisit)

- 5th gen - Multi-Core: Multiple threads are executed simultaneously on multiple processors
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.
The diagram illustrates the allocation of threads to Streaming Multiprocessors (SMs) in block granularity. Threads are assigned to SMs in blocks, with each SM being capable of handling up to 1536 threads. The number of threads per block can be varied, for example:

- 256 threads per block * 6 blocks
- 512 threads per block * 3 blocks

Threads run concurrently, with each SM maintaining and scheduling thread execution. The diagram showcases the structure of threads and blocks, along with the distribution of shared memory and Multi-Thread Instruction Units (MT IU).
## Configuration of Fermi and Kepler

<table>
<thead>
<tr>
<th></th>
<th>FERMI GF100</th>
<th>FERMI GF104</th>
<th>KEPLER GK104</th>
<th>KEPLER GK110</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Capability</td>
<td>2.0</td>
<td>2.1</td>
<td>3.0</td>
<td>3.5</td>
</tr>
<tr>
<td>Threads / Warp</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Max Warps / Multiprocessor</td>
<td>48</td>
<td>48</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Max Threads / Multiprocessor</td>
<td>1536</td>
<td>1536</td>
<td>2048</td>
<td>2048</td>
</tr>
<tr>
<td>Max Thread Blocks / Multiprocessor</td>
<td>8</td>
<td>8</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>32-bit Registers / Multiprocessor</td>
<td>32768</td>
<td>32768</td>
<td>65536</td>
<td>65536</td>
</tr>
<tr>
<td>Max Registers / Thread</td>
<td>63</td>
<td>63</td>
<td>63</td>
<td>255</td>
</tr>
<tr>
<td>Max Threads / Thread Block</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
</tr>
<tr>
<td>Shared Memory Size Configurations (bytes)</td>
<td>16K</td>
<td>16K</td>
<td>16K</td>
<td>16K</td>
</tr>
<tr>
<td></td>
<td>48K</td>
<td>48K</td>
<td>32K</td>
<td>32K</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>48K</td>
<td>48K</td>
</tr>
<tr>
<td>Max X Grid Dimension</td>
<td>$2^{\text{16-1}}$</td>
<td>$2^{\text{16-1}}$</td>
<td>$2^{\text{32-1}}$</td>
<td>$2^{\text{32-1}}$</td>
</tr>
<tr>
<td>Hyper-Q</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Dynamic Parallelism</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Compute Capability of Fermi and Kepler GPUs
The Von-Neumann Model

- Memory
- I/O
- Processing Unit
  - ALU
  - Reg File
- Control Unit
  - PC
  - IR
The Von-Neumann Model with SIMD units

Memory

I/O

Processing Unit

Reg File

ALU

Control Unit

PC

IR
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
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
Operate Instructions

• Example of an operate instruction:
  ADD R1, R2, R3

• Instruction cycle for an operate instruction:
  Fetch | Decode | Execute | Memory
Data Transfer Instructions

• Examples of data transfer instruction:
  LDR R1, R2, #2
  STR R1, R2, #2

• Instruction cycle for an data transfer instruction:
  Fetch | Decode | Execute | Memory
Control Flow Operations

• Example of control flow instruction:
  BRp #\(-4\)
  if the condition is positive, jump back four instructions

• Instruction cycle for an arithmetic instruction:
  Fetch | Decode | Execute | Memory
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 use __syncthreads() to get correct results (more later).
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:
  - \( \text{If (threadIdx.x > 2)} \) { }
  - 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:
  - \( \text{If (threadIdx.x / WARP_SIZE > 2)} \) { }
  - 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
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

TB = Thread Block, W = Warp
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.
ANY MORE 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`, `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. $\text{__sin}(x)$)
  - $\text{__pow}$
  - $\text{__log, __log2, __log10}$
  - $\text{__exp}$
  - $\text{__sin, __cos, __tan}$