Objective

- To learn the basic features of the memories accessible by CUDA threads
- To prepare for MP-2 - basic matrix multiplication
- To learn to evaluate the performance implications of global memory accesses

Programmer View of CUDA Memories

- Each thread can:
  - Read/write per-thread registers (~1 cycle)
  - Read/write per-block shared memory (~5 cycles)
  - Read/write per-grid global memory (~500 cycles)
  - Read/only per-grid constant memory (~5 cycles with caching)
Going back to the program

- Every instruction needs to be fetched from memory, decoded, then executed.
  - The decode stage typically accesses register file
- 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
  R1 – dest, R2, R3 - src
- Instruction cycle for an operate instruction:
  Fetch | Decode | Execute | Memory

Memory Access Instructions

- Examples of memory access instruction:
  LDR R1, R2, #2
  STR R1, R2, #2
- Instruction cycle for an operate instruction:
  Fetch | Decode | Execute | Memory

Registers vs Memory

- Registers are “free”
  - No additional memory access instruction
  - Very fast to use, however, there are very few of them
- Memory is expensive (slow), but very large
CUDA Variable Type Qualifiers

<table>
<thead>
<tr>
<th>Variable declaration</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>int LocalVar;</td>
<td>register</td>
<td>thread</td>
<td>thread</td>
</tr>
<tr>
<td><strong>device</strong> <strong>shared</strong> int SharedVar;</td>
<td>shared</td>
<td>block</td>
<td>block</td>
</tr>
<tr>
<td><strong>device</strong> <strong>shared</strong> int GlobalVar;</td>
<td>global</td>
<td>grid</td>
<td>application</td>
</tr>
<tr>
<td><strong>device</strong> <strong>constant</strong> int ConstantVar;</td>
<td>constant</td>
<td>grid</td>
<td>application</td>
</tr>
</tbody>
</table>

• __device__ is optional when used with __shared__ or __constant__

• Automatic variables without any qualifier reside in a register
  – Except per-thread arrays that reside in global memory

Matrix Multiplication Example
A Simple Host Version in C

// Matrix multiplication on the (CPU) host in single 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) {
            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;
        }
}

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

A Slightly Bigger Example
(TILE_WIDTH =2)

WIDTH = 8; TILE_WIDTH =2
Each block has 2*2 = 4 threads

WIDTH/TILE_WIDTH = 4
Use 4*4 = 16 blocks
**A Slightly Bigger Example (cont.)**
(TILE_WIDTH = 4)

```
<table>
<thead>
<tr>
<th>P0,0</th>
<th>P1,0</th>
<th>P0,1</th>
<th>P1,1</th>
<th>P0,2</th>
<th>P1,2</th>
<th>P0,3</th>
<th>P1,3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

WIDTH = 8; TILE_WIDTH = 4
Each block has 4*4 = 16 threads
WIDTH/TILE_WIDTH = 2
Use 2* 2 = 4 blocks
```

---

**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;

---

**Kernel Invocation (Host-side Code)**

// Setup the execution configuration
// TILE_WIDTH is a #define constant
dim3 dimGrid(ceil((1.0*Width)/TILE_WIDTH),
            ceil((1.0*Width)/TILE_WIDTH), 1);
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);

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

---

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

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

```
M0,0 M0,1 M0,2 M0,3
M1,0 M1,1 M1,2 M1,3
M2,0 M2,1 M2,2 M2,3
M3,0 M3,1 M3,2 M3,3
```
Work for Block (0,1)

\[
\begin{align*}
\text{Col} &= 1 \times 2 + \text{threadIdx.x} \\
\text{Row} &= 0 \times 2 + \text{threadIdx.y}
\end{align*}
\]

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;
    }
}
```

How about performance on a device with 150 GB/s memory bandwidth?

- All threads access global memory for their input matrix elements
  - Two memory accesses (8 bytes) per floating point multiply-add
  - 48B/s of memory bandwidth/FLOPS
  - 150 GB/s limits the code at 37.5 GFLOPS
- The actual code runs at about 25 GFLOPS
- Need to drastically cut down memory accesses to get closer to the peak of more than 1,000 GFLOPS

ANY MORE QUESTIONS?
READ CHAPTER 4!