ECE 498AL

Programming Massively Parallel Processors

Lecture 6: CUDA Memories
Part 2
Tiled Multiply

- Break up the execution of the kernel into phases so that the data accesses in each phase is focused on one subset (tile) of Md and Nd
A Small Example
Every Md and Nd Element is used exactly twice in generating a 2X2 tile of P

<table>
<thead>
<tr>
<th>M_{0,0} \times N_{0,0}</th>
<th>M_{0,0} \times N_{1,0}</th>
<th>M_{0,1} \times N_{0,0}</th>
<th>M_{0,1} \times N_{1,0}</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_{1,0} \times N_{0,1}</td>
<td>M_{1,0} \times N_{1,1}</td>
<td>M_{1,1} \times N_{0,1}</td>
<td>M_{1,1} \times N_{1,1}</td>
</tr>
<tr>
<td>M_{2,0} \times N_{0,2}</td>
<td>M_{2,0} \times N_{1,2}</td>
<td>M_{2,1} \times N_{0,2}</td>
<td>M_{2,1} \times N_{1,2}</td>
</tr>
<tr>
<td>M_{3,0} \times N_{0,3}</td>
<td>M_{3,0} \times N_{1,3}</td>
<td>M_{3,1} \times N_{0,3}</td>
<td>M_{3,1} \times N_{1,3}</td>
</tr>
</tbody>
</table>

Access order
Breaking Md and Nd into Tiles

- Break up the inner product loop of each thread into phases
- At the beginning of each phase, load the Md and Nd elements that everyone needs during the phase into shared memory
- Everyone access the Md and Nd elements from the shared memory during the phase
Each phase of a Thread Block uses one tile from Md and one from Nd

<table>
<thead>
<tr>
<th>Phase 1</th>
<th>Phase 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_{0,0} )</td>
<td>( T_{1,0} )</td>
</tr>
<tr>
<td>( \text{Md}_{0,0} )</td>
<td>( \text{Md}_{1,0} )</td>
</tr>
<tr>
<td>( \downarrow )</td>
<td>( \downarrow )</td>
</tr>
<tr>
<td>( \text{Mds}_{0,0} )</td>
<td>( \text{Mds}_{1,0} )</td>
</tr>
<tr>
<td>( \text{Nd}_{0,0} )</td>
<td>( \text{Nd}_{1,0} )</td>
</tr>
<tr>
<td>( \downarrow )</td>
<td>( \downarrow )</td>
</tr>
<tr>
<td>( \text{Nds}_{0,0} )</td>
<td>( \text{Nds}_{1,0} )</td>
</tr>
<tr>
<td>( \text{PValue}<em>{0,0} += ) ( \text{Mds}</em>{0,0} \cdot \text{Nds}<em>{0,0} + \text{Mds}</em>{1,0} \cdot \text{Nds}_{1,0} )</td>
<td>( \text{PValue}<em>{1,0} += ) ( \text{Mds}</em>{0,0} \cdot \text{Nds}<em>{0,0} + \text{Mds}</em>{1,0} \cdot \text{Nds}_{1,1} )</td>
</tr>
<tr>
<td>( \text{Md}_{2,0} )</td>
<td>( \text{Md}_{3,0} )</td>
</tr>
<tr>
<td>( \downarrow )</td>
<td>( \downarrow )</td>
</tr>
<tr>
<td>( \text{Mds}_{0,0} )</td>
<td>( \text{Mds}_{1,0} )</td>
</tr>
<tr>
<td>( \text{Nd}_{0,2} )</td>
<td>( \text{Nd}_{1,2} )</td>
</tr>
<tr>
<td>( \downarrow )</td>
<td>( \downarrow )</td>
</tr>
<tr>
<td>( \text{Nds}_{0,0} )</td>
<td>( \text{Nds}_{1,0} )</td>
</tr>
<tr>
<td>( \text{PValue}<em>{0,0} += ) ( \text{Mds}</em>{0,0} \cdot \text{Nds}<em>{0,0} + \text{Mds}</em>{1,0} \cdot \text{Nds}_{0,1} )</td>
<td>( \text{PValue}<em>{1,0} += ) ( \text{Mds}</em>{0,0} \cdot \text{Nds}<em>{1,0} + \text{Mds}</em>{1,0} \cdot \text{Nds}_{1,1} )</td>
</tr>
<tr>
<td>( \text{Md}_{2,1} )</td>
<td>( \text{Md}_{3,1} )</td>
</tr>
<tr>
<td>( \downarrow )</td>
<td>( \downarrow )</td>
</tr>
<tr>
<td>( \text{Mds}_{0,1} )</td>
<td>( \text{Mds}_{1,1} )</td>
</tr>
<tr>
<td>( \text{Nd}_{0,3} )</td>
<td>( \text{Nd}_{1,3} )</td>
</tr>
<tr>
<td>( \downarrow )</td>
<td>( \downarrow )</td>
</tr>
<tr>
<td>( \text{Nds}_{0,1} )</td>
<td>( \text{Nds}_{1,1} )</td>
</tr>
<tr>
<td>( \text{PValue}<em>{0,1} += ) ( \text{Mds}</em>{0,1} \cdot \text{Nds}<em>{0,0} + \text{Mds}</em>{1,1} \cdot \text{Nds}_{0,1} )</td>
<td>( \text{PValue}<em>{1,1} += ) ( \text{Mds}</em>{0,1} \cdot \text{Nds}<em>{1,0} + \text{Mds}</em>{1,1} \cdot \text{Nds}_{1,1} )</td>
</tr>
</tbody>
</table>

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE498AL, University of Illinois, Urbana Champaign
Tiled Matrix Multiplication Kernel

```c
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
  __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

  int bx = blockIdx.x;  int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;

  // Identify the row and column of the Pd element to work on
  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;

  float Pvalue = 0;
  // Loop over the Md and Nd tiles required to compute the Pd element
  for (int m = 0; m < Width/TILE_WIDTH; ++m) {
    // Collaborative loading of Md and Nd tiles into shared memory
    Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
    Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
    __syncthreads();

    for (int k = 0; k < TILE_WIDTH; ++k)
      Pvalue += Mds[ty][k] * Nds[k][tx];
    __syncthreads();

    Pd[Row*Width+Col] = Pvalue;
  }
}
```

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE498AL, University of Illinois, Urbana Champaign
CUDA Code – Kernel Execution Configuration

// Setup the execution configuration

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

First-order Size Considerations in G80

• Each **thread block** should have many threads
  – TILE_WIDTH of 16 gives 16*16 = 256 threads

• There should be many thread blocks
  – A 1024*1024 Pd gives 64*64 = 4096 Thread Blocks
  – TILE_WIDTH of 16 gives each SM 3 blocks, 768 threads (full capacity)

• Each thread block perform 2*256 = 512 float loads from global memory for 256 * (2*16) = 8,192 mul/add operations.
  – Memory bandwidth no longer a limiting factor
Tiled Multiply

- Each **block** computes one square sub-matrix $P_{d_{sub}}$ of size $TILE\_WIDTH$
- Each **thread** computes one element of $P_{d_{sub}}$
G80 Shared Memory and Threading

- Each SM in G80 has 16KB shared memory
  - SM size is implementation dependent!
  - For TILE_WIDTH = 16, each thread block uses 2*256*4B = 2KB of shared memory.
  - The shared memory can potentially have up to 8 Thread Blocks actively executing
    - This allows up to 8*512 = 4,096 pending loads. (2 per thread, 256 threads per block)
    - The threading model limits the number of thread blocks to 3 so shared memory is not the limiting factor here
  - The next TILE_WIDTH 32 would lead to 2*32*32*4B = 8KB shared memory usage per thread block, allowing only up to two thread blocks active at the same time

- Using 16x16 tiling, we reduce the accesses to the global memory by a factor of 16
  - The 86.4B/s bandwidth can now support (86.4/4)*16 = 347.6 GFLOPS!
Tiled Matrix Multiplication Kernel

__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x; int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

    // Identify the row and column of the Pd element to work on
    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;

    float Pvalue = 0;
    // Loop over the Md and Nd tiles required to compute the Pd element
    for (int m = 0; m < Width/TILE_WIDTH; ++m) {
        // Collaborative loading of Md and Nd tiles into shared memory
        Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
        Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
        __syncthreads();

        // Pvalue += Mds[ty][k] * Nds[k][tx];
        Synchthreads();
    }
    Pd[Row*Width+Col] = Pvalue;
}

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE498AL, University of Illinois, Urbana Champaign
Tiling Size Effects
Summary - Typical Structure of a CUDA Program

- **Introduction**
- **Main Sections**
  - **CUDA Architecture**
  - **Memory Model**
  - **Scheduling**
  - **CUDA Programming Model**
  - **CUDA Libraries**
  - **CUDA Example**

- **Conclusion**

*repeat as needed*