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>Access order</th>
<th>P₀,₀ thread₀,₀</th>
<th>P₁,₀ thread₁,₀</th>
<th>P₀,₁ thread₀,₁</th>
<th>P₁,₁ thread₁,₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>M₀,₀ * N₀,₀</td>
<td>M₀,₀ * N₁,₀</td>
<td>M₀,₁ * N₀,₀</td>
<td>M₀,₁ * N₁,₀</td>
<td></td>
</tr>
<tr>
<td>M₁,₀ * N₀,₁</td>
<td>M₁,₀ * N₁,₁</td>
<td>M₁,₁ * N₂,₀</td>
<td>M₁,₁ * N₁,₁</td>
<td></td>
</tr>
<tr>
<td>M₂,₀ * N₀,₂</td>
<td>M₂,₀ * N₁,₂</td>
<td>M₂,₁ * N₂,₀</td>
<td>M₂,₁ * N₁,₂</td>
<td></td>
</tr>
<tr>
<td>M₃,₀ * N₀,₃</td>
<td>M₃,₀ * N₁,₃</td>
<td>M₃,₁ * N₀,₃</td>
<td>M₃,₁ * N₁,₃</td>
<td></td>
</tr>
</tbody>
</table>
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>
</table>
| $T_{0,0}$ | $Md_{0,0}$  
$\downarrow$  
$Mds_{0,0}$ | $Nd_{0,0}$  
$\downarrow$  
$Nds_{0,0}$ | $PValue_{0,0} +=$  
$Mds_{0,0} \times Nds_{0,0} +$  
$Mds_{1,0} \times Nds_{1,1}$ | $Md_{2,0}$  
$\downarrow$  
$Mds_{0,0}$ | $Nd_{0,2}$  
$\downarrow$  
$Nds_{0,0}$ | $PValue_{0,0} +=$  
$Mds_{0,0} \times Nds_{0,0} +$  
$Mds_{1,0} \times Nds_{1,1}$ |
| $T_{1,0}$ | $Md_{1,0}$  
$\downarrow$  
$Mds_{1,0}$ | $Nd_{1,0}$  
$\downarrow$  
$Nds_{1,0}$ | $PValue_{1,0} +=$  
$Mds_{0,0} \times Nds_{1,0} +$  
$Mds_{1,0} \times Nds_{1,1}$ | $Md_{3,0}$  
$\downarrow$  
$Mds_{1,0}$ | $Nd_{1,2}$  
$\downarrow$  
$Nds_{1,0}$ | $PValue_{1,0} +=$  
$Mds_{0,0} \times Nds_{1,0} +$  
$Mds_{1,0} \times Nds_{1,1}$ |
| $T_{0,1}$ | $Md_{0,1}$  
$\downarrow$  
$Mds_{0,1}$ | $Nd_{0,1}$  
$\downarrow$  
$Nds_{1,1}$ | $PdValue_{0,1} +=$  
$Mds_{0,1} \times Nds_{0,0} +$  
$Mds_{1,1} \times Nds_{1,1}$ | $Md_{2,1}$  
$\downarrow$  
$Mds_{0,1}$ | $Nd_{0,3}$  
$\downarrow$  
$Nds_{0,1}$ | $PdValue_{0,1} +=$  
$Mds_{0,1} \times Nds_{0,0} +$  
$Mds_{1,1} \times Nds_{1,1}$ |
| $T_{1,1}$ | $Md_{1,1}$  
$\downarrow$  
$Mds_{1,1}$ | $Nd_{1,1}$  
$\downarrow$  
$Nds_{1,1}$ | $PdValue_{1,1} +=$  
$Mds_{0,1} \times Nds_{1,0} +$  
$Mds_{1,1} \times Nds_{1,1}$ | $Md_{3,1}$  
$\downarrow$  
$Mds_{1,1}$ | $Nd_{1,3}$  
$\downarrow$  
$Nds_{1,1}$ | $PdValue_{1,1} +=$  
$Mds_{0,1} \times Nds_{1,0} +$  
$Mds_{1,1} \times Nds_{1,1}$ |
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;

  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;

  float Pvalue = 0;
  for (int m = 0; m < Width/TILE_WIDTH; ++m) {
    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;
}
```

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 $16 \times 16 = 256$ threads

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

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

- Each **block** computes one square sub-matrix $P_{d_{sub}}$ of size $\text{TILE}_W$IDTH
- 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!

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;
}
```

ECE498AL, University of Illinois, Urbana Champaign
Tiling Size Effects

ECE498AL, University of Illinois, Urbana Champaign
Summary- Typical Structure of a CUDA Program

- Global variables declaration
  - __host__
  - __device__... __global__, __constant__, __texture__

- Function prototypes
  - __global__ void kernelOne(...)
  - float handyFunction(...)

- Main()
  - allocate memory space on the device – cudaMalloc(&d_GlblVarPtr, bytes)
  - transfer data from host to device – cudaMemcpy(d_GlblVarPtr, h_GlblVarPtr, bytes)
  - execution configuration setup
  - kernel call – kernelOne<<<execution configuration>>>( args... );
  - transfer results from device to host – cudaMemcpy(h_GlblVarPtr, d_GlblVarPtr,...)
  - optional: compare against golden (host computed) solution

- Kernel – void kernelOne(type args,...)
  - variables declaration - __local__, __shared__
    - automatic variables transparently assigned to registers or local memory
  - syncthreads()...

- Other functions
  - float handyFunction(int inVar...);