PASI Summer School

Advanced Algorithmic Techniques for GPUs

Lecture 3: Blocking/Tiling for Locality
Objective

• Reuse each data accessed from the global memory multiple times
  – Across threads – shared memory blocking
  – Within a thread - register tiling

• Register tiling is also often used to re-use computation results for increased efficiency.
Shared Memory Blocking Basic Idea

Thread 1

Global Memory

Thread 2

On-chip Memory

©Wen-mei W. Hwu and David Kirk, NVIDIA, Chile, January 5–7, 2011
Basic Concept of Blocking/Tiling

• In a congested traffic system, significant reduction of vehicles can greatly improve the delay seen by all vehicles
  – Carpooling for commuters
  – Blocking/Tiling for global memory accesses
    • drivers = threads,
    • cars = data

©Wen-mei W. Hwu and David Kirk/NVIDIA, Chile, January 5–7, 2011
Some computations are more challenging to block/tile than others.

• Some carpools may be easier than others
  – More efficient if neighbors are also classmates or co-workers
  – Some vehicles may be more suitable for carpooling

• Similar variations exist in blocking/tiling
Carpools need synchronization.

• Good – when people have similar schedule

Worker A  sleep  work  dinner
Time
Worker B  sleep  work  dinner

• Bad – when people have very different schedule

Worker A  party  sleep  work
  time
Worker B  sleep  work  dinner
Same with Blocking/Tiling

- Good – when threads have similar access timing
- Bad – when threads have very different timing
Outline of Technique

• Identify a block/tile of global memory content that are accessed by multiple threads
• Load the block/tile from global memory into on-chip memory
• Have the multiple threads to access their data from the on-chip memory
• Move on to the next block/tile
Tiled Matrix Multiply

- Each row of Md is accessed by multiple threads
- Problem: some threads can be much further along than others
  - An entire row may need to be in on-chip memory
  - Not enough on-chip memory for large input matrices
A Small Example

- Can we use two on-chip memory locations to reduce the number of M accesses by the two threads?
  - Not if the two threads can have very different timing!
Every M and N Element is used exactly twice in generating a 2X2 tile of P.

<table>
<thead>
<tr>
<th></th>
<th>$P_{0,0}$ thread$_{0,0}$</th>
<th>$P_{1,0}$ thread$_{1,0}$</th>
<th>$P_{0,1}$ thread$_{0,1}$</th>
<th>$P_{1,1}$ thread$_{1,1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_{0,0} \times N_{0,0}$</td>
<td>$M_{0,0} \times N_{1,0}$</td>
<td>$M_{0,1} \times N_{0,0}$</td>
<td>$M_{0,1} \times N_{1,0}$</td>
<td></td>
</tr>
<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>
<td></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>
<td></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>
<td></td>
</tr>
</tbody>
</table>

Access order:

©Wen-mei W. Hwu and David Kirk/NVIDIA, Chile, January 5–7, 2011
Breaking Md and Nd into Tiles

Phase 1
Breaking Md and Nd into Tiles (cont.)
Each phase 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><strong>T(_{0,0})</strong></td>
<td><strong>T(_{0,0})</strong></td>
</tr>
<tr>
<td>Md(_{0,0})</td>
<td>Md(_{2,0})</td>
</tr>
<tr>
<td>(\downarrow)</td>
<td>(\downarrow)</td>
</tr>
<tr>
<td>Mds(_{0,0})</td>
<td>Mds(_{0,0})</td>
</tr>
<tr>
<td>Nd(_{0,0})</td>
<td>Nd(_{0,2})</td>
</tr>
<tr>
<td>(\downarrow)</td>
<td>(\downarrow)</td>
</tr>
<tr>
<td>Nds(_{0,0})</td>
<td>Nds(_{0,0})</td>
</tr>
</tbody>
</table>

\[ PValue\(_{0,0}\) += Mds\(_{0,0}\) \times Nds\(_{0,0}\) + Mds\(_{1,0}\) \times Nds\(_{0,1}\) \]

<table>
<thead>
<tr>
<th><strong>T(_{1,0})</strong></th>
<th><strong>T(_{1,0})</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Md(_{1,0})</td>
<td>Md(_{3,0})</td>
</tr>
<tr>
<td>(\downarrow)</td>
<td>(\downarrow)</td>
</tr>
<tr>
<td>Mds(_{1,0})</td>
<td>Mds(_{1,0})</td>
</tr>
<tr>
<td>Nd(_{1,0})</td>
<td>Nd(_{1,2})</td>
</tr>
<tr>
<td>(\downarrow)</td>
<td>(\downarrow)</td>
</tr>
<tr>
<td>Nds(_{1,0})</td>
<td>Nds(_{1,0})</td>
</tr>
</tbody>
</table>

\[ PValue\(_{1,0}\) += Mds\(_{0,0}\) \times Nds\(_{1,0}\) + Mds\(_{1,1}\) \times Nds\(_{1,1}\) \]

<table>
<thead>
<tr>
<th><strong>T(_{0,1})</strong></th>
<th><strong>T(_{0,1})</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Md(_{0,1})</td>
<td>Md(_{2,1})</td>
</tr>
<tr>
<td>(\downarrow)</td>
<td>(\downarrow)</td>
</tr>
<tr>
<td>Mds(_{0,1})</td>
<td>Mds(_{0,1})</td>
</tr>
<tr>
<td>Nd(_{0,1})</td>
<td>Nd(_{0,3})</td>
</tr>
<tr>
<td>(\downarrow)</td>
<td>(\downarrow)</td>
</tr>
<tr>
<td>Nds(_{0,1})</td>
<td>Nds(_{0,1})</td>
</tr>
</tbody>
</table>

\[ PdValue\(_{0,1}\) += Mds\(_{0,1}\) \times Nds\(_{0,0}\) + Mds\(_{1,1}\) \times Nds\(_{0,1}\) \]

<table>
<thead>
<tr>
<th><strong>T(_{1,1})</strong></th>
<th><strong>T(_{1,1})</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Md(_{1,1})</td>
<td>Md(_{3,1})</td>
</tr>
<tr>
<td>(\downarrow)</td>
<td>(\downarrow)</td>
</tr>
<tr>
<td>Mds(_{1,1})</td>
<td>Mds(_{1,1})</td>
</tr>
<tr>
<td>Nd(_{1,1})</td>
<td>Nd(_{1,3})</td>
</tr>
<tr>
<td>(\downarrow)</td>
<td>(\downarrow)</td>
</tr>
<tr>
<td>Nds(_{1,1})</td>
<td>Nds(_{1,1})</td>
</tr>
</tbody>
</table>

\[ PdValue\(_{1,1}\) += Mds\(_{0,1}\) \times Nds\(_{1,0}\) + Mds\(_{1,1}\) \times Nds\(_{1,1}\) \]

©Wen-mei W. Hwu and David Kirk/NVIDIA, Chile, January 5–7, 2011
Tiled Multiply – Large Matrices

- Make sure that tiles are all loaded in vertical patterns from the global memory.
- Md data can then be accessed from shared memory in horizontal direction.
First-order Size Considerations

• Assume
  – TILE_WIDTH of 16 gives 16*16 = 256 threads
  – A 1024*1024 Pd gives 64*64 = 4096 Thread Blocks
• 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
  – Could use thread coarsening to further reduce traffic
• Each thread block can have up to 1024 threads
  – Can use 32*32 tiles to further reduce traffic
Memory Access Pattern (Corner Turning)

Original Access Pattern

Tiled Access Pattern

Copy into scratchpad memory

Perform multiplication with scratchpad values
Memory Layout of a Matrix in C

Access direction in Kernel code

Time Period 1
T₁ T₂ T₃ T₄

Time Period 2
T₁ T₂ T₃ T₄

…
Memory Layout of a Matrix in C

Access
direction in
Kernel code

Time Period 2

T₁  T₂  T₃  T₄

Time Period 1

T₁  T₂  T₃  T₄

©Wen-mei W. Hwu and David Kirk/NVIDIA,
Chile, January 5–7, 2011
Loading a Tile

• All threads in a block participate
  – Each thread loads one Md element and one Nd element in based tiled code

• Assign the loaded element to each thread such that the accesses within each warp is coalesced
CUDA Code – Kernel Execution Configuration

// Setup the execution configuration

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

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

©Wen-mei W. Hwu and David Kirk/NVIDIA, Chile, January 5–7, 2011
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[tx][ty] = Md[Row*Width + m*TILE_WIDTH + tx];
    Nds[tx][ty] = Nd[(m*TILE_WIDTH + ty) * Width + Col]);
    __syncthreads();
    for (int k = 0; k < TILE_WIDTH; ++k)
      Pvalue += Mds[tx][k] * Nds[k][ty];
    __syncthreads();
  }
  Pd[Row*Width+Col] = Pvalue;
}

©Wen-mei W. Hwu and David Kirk/NVIDIA, Chile, January 5-7, 2011
Shared Memory and Threading

• Each SM in Fermi has 64KB on-chip SRAM, partitioned into 48KB L1 cache and 16KB shared memory, or vice versa
  – SM shared memory size is implementation dependent!
  – For TILE_WIDTH = 16, each thread block uses $2 \times 256 \times 4B = 2KB$ of shared memory.
  – Can potentially have up to 8 Thread Blocks actively executing
    • This allows up to $8 \times 512 = 4,096$ pending loads. (2 per thread, 256 threads per block)
    – The next TILE_WIDTH 32 would lead to $2 \times 32 \times 32 \times 4B = 8KB$ shared memory usage per thread block, allowing 2 or 6 thread blocks active at the same time (Problem with earlier GPUs!)
• Using 16x16 tiling, we reduce the accesses to the global memory by a factor of 16
  – A 150GB/s bandwidth can now support $(150/4) \times 16 = 600$ GFLOPS!
ANY MORE QUESTIONS?