Performance Considerations – I

Miaoqing Huang
University of Arkansas
Fall 2011
1 Memory Coalescing
Memory Coalescing

- Off-chip memory is accessed in chunks
  - Even if you read only a single word
  - If you don’t use whole chunk, bandwidth is wasted
- Chunks are aligned to multiples of 128 bytes on Fermi GPU
Memory Coalescing

- Off-chip memory is accessed in chunks
  - Even if you read only a single word
  - If you don’t use whole chunk, bandwidth is wasted
- Chunks are aligned to multiples of 128 bytes on Fermi GPU

![Diagram showing memory coalescing with threads accessing memory in chunks.](image-url)
How is the global memory access of the threads in a warp coalesced?
- On Fermi, global memory loads and stores by threads of a warp (i.e., 32 threads) are coalesced.

How is the coalesced memory access aligned into segments?
- On Fermi, the segment size is always 128 Bytes.

Thread blocks are partitioned into warps based on thread indices.
- Each warp contains threads of consecutive and increasing thread IDs with the first warp containing thread 0.
Memory Coalescing

Project the threads into a linear order

- Line up the row with larger \( y \) and \( z \) coordinates after those with lower ones
Partition the threads into warps

\[ T_0 \quad T_1 \quad T_2 \quad \ldots \quad T_{30} \quad T_{31} \quad T_{32} \quad T_{33} \quad \ldots \quad T_{62} \quad T_{63} \quad T_{64} \]
Memory Coalescing

Partition the threads into warps

<table>
<thead>
<tr>
<th>$T_{0,0}$</th>
<th>$T_{1,0}$</th>
<th>$T_{2,0}$</th>
<th>$T_{3,0}$</th>
<th>$T_{4,0}$</th>
<th>$T_{5,0}$</th>
<th>$T_{6,0}$</th>
<th>$T_{7,0}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{0,1}$</td>
<td>$T_{1,1}$</td>
<td>$T_{2,1}$</td>
<td>$T_{3,1}$</td>
<td>$T_{4,1}$</td>
<td>$T_{5,1}$</td>
<td>$T_{6,1}$</td>
<td>$T_{7,1}$</td>
</tr>
<tr>
<td>$T_{0,2}$</td>
<td>$T_{1,2}$</td>
<td>$T_{2,2}$</td>
<td>$T_{3,2}$</td>
<td>$T_{4,2}$</td>
<td>$T_{5,2}$</td>
<td>$T_{6,2}$</td>
<td>$T_{7,2}$</td>
</tr>
<tr>
<td>$T_{0,3}$</td>
<td>$T_{1,3}$</td>
<td>$T_{2,3}$</td>
<td>$T_{3,3}$</td>
<td>$T_{4,3}$</td>
<td>$T_{5,3}$</td>
<td>$T_{6,3}$</td>
<td>$T_{7,3}$</td>
</tr>
<tr>
<td>$T_{0,4}$</td>
<td>$T_{1,4}$</td>
<td>$T_{2,4}$</td>
<td>$T_{3,4}$</td>
<td>$T_{4,4}$</td>
<td>$T_{5,4}$</td>
<td>$T_{6,4}$</td>
<td>$T_{7,4}$</td>
</tr>
<tr>
<td>$T_{0,5}$</td>
<td>$T_{1,5}$</td>
<td>$T_{2,5}$</td>
<td>$T_{3,5}$</td>
<td>$T_{4,5}$</td>
<td>$T_{5,5}$</td>
<td>$T_{6,5}$</td>
<td>$T_{7,5}$</td>
</tr>
<tr>
<td>$T_{0,6}$</td>
<td>$T_{1,6}$</td>
<td>$T_{2,6}$</td>
<td>$T_{3,6}$</td>
<td>$T_{4,6}$</td>
<td>$T_{5,6}$</td>
<td>$T_{6,6}$</td>
<td>$T_{7,6}$</td>
</tr>
<tr>
<td>$T_{0,7}$</td>
<td>$T_{1,7}$</td>
<td>$T_{2,7}$</td>
<td>$T_{3,7}$</td>
<td>$T_{4,7}$</td>
<td>$T_{5,7}$</td>
<td>$T_{6,7}$</td>
<td>$T_{7,7}$</td>
</tr>
</tbody>
</table>
Simple Memory Access Pattern

- Coalesced access in which all threads but a few access the word in a segment (on Fermi)
  - Not all threads in a warp need to access the memory
  - The access by threads can be permuted

![Diagram showing coalesced access pattern with 128B segment]
Misaligned Access Pattern

- Misaligned sequential addresses that fall within two 128-byte segments
Misaligned Access Pattern

- Misaligned sequential addresses that fall within two 128-byte segments
Misaligned Access Pattern

- Misaligned sequential addresses that fall within two 128-byte segments
Matrix Data Access Pattern

- Directly access data from global memory
  - Each thread reads one row of $M_d$ and one column of $N_d$
Memory Coalescing

Matrix Data Access Pattern

- Directly access data from global memory
  - Each thread reads one row of $M_d$ and one column of $N_d$
Matrix Data Access Pattern

- Directly access data from global memory
- Each thread reads one row of $\text{Md}$ and one column of $\text{Nd}$
Matrix Data Access Pattern

- Directly access data from global memory
  - Each thread reads one row of $M_d$ and one column of $N_d$
Matrix Data Access Pattern

- Directly access data from global memory
  - Each thread reads one row of $M_d$ and one column of $N_d$
Memory Coalescing

Matrix Data Access Pattern

- Directly access data from global memory
  - Each thread reads one row of $M_d$ and one column of $N_d$

Md

Nd
Matrix Data Access Pattern

- Directly access data from global memory
  - Each thread reads one row of $\mathbf{Md}$ and one column of $\mathbf{Nd}$
Memory Coalescing

Matrix Data Access Pattern

- Directly access data from global memory
  - Each thread reads one row of $M_d$ and one column of $N_d$
Matrix Data Access Pattern

- Directly access data from global memory
- Each thread reads one row of $M_d$ and one column of $N_d$
Matrix Data Access Pattern

- Directly access data from global memory
- Each thread reads one row of $M_d$ and one column of $N_d$
Matrix Data Access Pattern

- Directly access data from global memory
- Each thread reads one row of $M_d$ and one column of $N_d$
Matrix Data Access Pattern

- Directly access data from global memory
  - Each thread reads one row of $M_d$ and one column of $N_d$
Matrix Data Access Pattern

- Directly access data from global memory
- Each thread reads one row of $M_d$ and one column of $N_d$
### Coalesced access to a matrix

The image illustrates the concept of coalesced access in memory. The matrix $M$ is divided into time periods, with each period containing elements labeled $T_1, T_2, T_3, T_4$. The diagram shows how elements are accessed in a coalesced manner, grouping together elements that are spatially close in memory to reduce latency.

#### Example:

- **Time Period 1:**
  - Elements: $T_1, T_2, T_3, T_4$

- **Time Period 2:**
  - Elements: $T_1, T_2, T_3, T_4$

- **Time Period 3:**
  - ...
Uncoalesced access to a matrix
Use Shared Memory to Improve Coalescing

Original Access Pattern

Tiled Access Pattern

Copy into shared memory

Perform multiplication with data in shared memory
## Aligned and Sequential

<table>
<thead>
<tr>
<th>Addresses:</th>
<th>96</th>
<th>128</th>
<th>160</th>
<th>192</th>
<th>224</th>
<th>256</th>
<th>288</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads:</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

**Aligned and sequential**
# Memory Coalescing

## Aligned and Sequential

<table>
<thead>
<tr>
<th>Addresses:</th>
<th>96</th>
<th>128</th>
<th>160</th>
<th>192</th>
<th>224</th>
<th>256</th>
<th>288</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads:</td>
<td>0</td>
<td>…</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compute capability:</td>
<td>1.0 and 1.1</td>
<td>1.2 and 1.3</td>
<td>2.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory transactions:</td>
<td>Uncached</td>
<td>Uncached</td>
<td>Cached</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 x 64B at 128</td>
<td>1 x 64B at 128</td>
<td>1 x 128B at 128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 x 64B at 192</td>
<td>1 x 64B at 192</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Aligned and Non-sequential

<table>
<thead>
<tr>
<th>Addresses:</th>
<th>96</th>
<th>128</th>
<th>160</th>
<th>192</th>
<th>224</th>
<th>256</th>
<th>288</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads:</td>
<td>0</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

The diagram shows memory addresses and corresponding threads, illustrating the concept of aligned and non-sequential memory coalescing.
## Aligned and Non-sequential

### Memory Transactions

<table>
<thead>
<tr>
<th>Addresses:</th>
<th>96</th>
<th>128</th>
<th>160</th>
<th>192</th>
<th>224</th>
<th>256</th>
<th>288</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads:</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>31</td>
</tr>
<tr>
<td>Compute capability:</td>
<td>1.0 and 1.1</td>
<td>1.2 and 1.3</td>
<td>2.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory transactions:</td>
<td>Uncached</td>
<td>Cached</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 x 32B at 128</td>
<td>1 x 64B at 128</td>
<td>1 x 128B at 128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 x 32B at 160</td>
<td>1 x 64B at 192</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 x 32B at 192</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 x 32B at 224</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Misaligned and Sequential

<table>
<thead>
<tr>
<th>Addresses:</th>
<th>96</th>
<th>128</th>
<th>160</th>
<th>192</th>
<th>224</th>
<th>256</th>
<th>288</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Threads:</th>
<th>0</th>
<th>...</th>
<th>31</th>
</tr>
</thead>
</table>

### Memory Coalescing

The diagram above illustrates the concept of memory coalescing in the context of misaligned and sequential addresses. The addresses shown are 96, 128, 160, 192, 224, 256, and 288. The threads are represented by numbers 0 to 31, with an ellipsis indicating sequential numbering. The arrows indicate the flow or access pattern of the threads across these addresses.
## Memory Coalescing

### Misaligned and Sequential

<table>
<thead>
<tr>
<th>Addresses:</th>
<th>96</th>
<th>128</th>
<th>160</th>
<th>192</th>
<th>224</th>
<th>256</th>
<th>288</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads:</td>
<td>0</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>31</td>
</tr>
<tr>
<td>Compute capability:</td>
<td>1.0 and 1.1</td>
<td>1.2 and 1.3</td>
<td>2.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory transactions:</td>
<td>Uncached</td>
<td>Cached</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7 x 32B at 128</td>
<td>1 x 128B at 128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8 x 32B at 160</td>
<td>1 x 64B at 192</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8 x 32B at 192</td>
<td>1 x 32B at 256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8 x 32B at 224</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 x 32B at 256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 x 128B at 128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 x 128B at 256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Matrix Multiplication Using Multiple Blocks with Tile
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int width)
{
1. __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
2. __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

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

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

7. float Pvalue = 0;

// Loop over the Md and Nd tiles required to compute the Pd element
8. for (int m = 0; m < Width/TILE_WIDTH; ++m) {

   // Collaborative loading of Md and Nd tiles into shared memory
9.      Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
   Nds[ty][tx] = Nd[(m*TILE_WIDTH + ty)*Width + Col];
   __syncthreads();

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

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