GPU Programming

Performance Considerations

Miaoqing Huang
University of Arkansas
Spring 2016
Outline

Control Flow Divergence

Memory Coalescing

Shared Memory Bank Conflicts

Occupancy

Loop Unrolling

Kernel Launch Overhead
Outline

Control Flow Divergence

Memory Coalescing

Shared Memory Bank Conflicts

Occupancy

Loop Unrolling

Kernel Launch Overhead
Control Flow

- Instructions are issued per 32 threads (warp)
- Divergent branches:
  - Threads within a single warp take different paths
    - if-else, ...
  - Different execution paths within a warp are serialized
- Different warps can execute different code

![Diagram showing control flow and conditional execution]

Not all ALUs do useful work!
Worst case: 1/8 performance
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

\begin{align*}
    T_0 & \quad T_1 & \quad T_2 & \quad \cdots & \quad T_{30} & \quad T_{31} & \quad T_{32} & \quad T_{33} & \quad \cdots & \quad T_{62} & \quad T_{63} & \quad T_{64}
\end{align*}
Partition the threads into warps
Avoid diverging within a warp
Example with divergence

```c
if (threadIdx.x > 2) {
    ...
} else {
    ...
}
```

- Branch granularity < warp size
Avoid diverging within a warp

Example with divergence

```c
if (threadIdx.x > 2) {
    ...
}
elcse {
    ...
}
```

- Branch granularity < warp size

Example without divergence

```c
if (((threadIdx.x / WARP_SIZE) > 2) {
    ...
}
elcse {
    ...
}
```

- Branch granularity is a whole multiple of warp size
Example: Divergent Iteration

```c
__global__ void per_thread_sum(int *indices, float *data, float *sums)
{
    ...
    // number of loop iterations is data dependent
    for(int j=indices[i]; j<indices[i+1]; j++) {
        sum += data[j];
    }
    sums[i] = sum;
}
```

- A single thread can drag a whole warp with it for a long time
- Know your data patterns
  - If data is unpredictable, try to flatten peaks by letting threads work on multiple data items
Execution of the Sum Reduction Kernel

[Diagram showing the execution of the sum reduction kernel with threads iterating through iterations 0 to 3, with operations (e.g., 0+1, 2+3) indicated.]
Outline

Control Flow Divergence

Memory Coalescing

Shared Memory Bank Conflicts

Occupancy

Loop Unrolling

Kernel Launch Overhead
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]

- For thread $i$:
  - 128 bytes
  - Accesses 128 bytes

- For thread $j$:
  - 128 bytes
  - Accesses 256 bytes
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
Coalesced Access to Global Memory

- 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
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
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 $N_d$
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 \( Md \) and one column of \( Nd \)
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 $N_d$
Matrix Data Access Pattern

- Directly access data from global memory
  - Each thread reads one row of $\mathbf{M}_d$ and one column of $\mathbf{N}_d$
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 $\text{Md}$ 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 $N_d$
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 $N_d$
Coalesced access to a matrix

Load iteration 0

Load iteration 1

...
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 Accesses

**Aligned accesses (sequential/ 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>
</table>

| Threads: | 0 | ... | 31 |

### Diagram

- **Blue arrows** indicate sequential accesses.
- **Red arrows** indicate non-sequential accesses.

This diagram illustrates the alignment of accesses across addresses with corresponding threads.
Aligned Accesses

Aligned accesses (sequential/ non-sequential)

Addresses: 96 128 160 192 224 256 288

Threads: 0 ... 31

Compute capability: 2.x, 3.x, 5.x

Memory transactions:

<table>
<thead>
<tr>
<th></th>
<th>Uncached</th>
<th>Cached</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x 32B at 128</td>
<td></td>
<td>1x 128B at 128</td>
</tr>
<tr>
<td>1x 32B at 160</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1x 32B at 192</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1x 32B at 224</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Misaligned Accesses

Mis-aligned accesses (sequential/ non-sequential)

Addresses:

Threads:

0  ...  31
**Misaligned Accesses**

**Mis-aligned accesses (sequential/ non-sequential)**

Addresses: 96 128 160 192 224 256 288

Threads: 0 ... 31

Compute capability:

<table>
<thead>
<tr>
<th>Memory transactions:</th>
<th>Uncached</th>
<th>Cached</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x 32B at 128</td>
<td>1x 128B at 128</td>
<td></td>
</tr>
<tr>
<td>1x 32B at 160</td>
<td>1x 128B at 128</td>
<td></td>
</tr>
<tr>
<td>1x 32B at 192</td>
<td>1x 128B at 256</td>
<td></td>
</tr>
<tr>
<td>1x 32B at 224</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1x 32B at 256</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Matrix Multiplication Kernel Using Multiple Blocks with Tile

__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
        __syncthreads();

        Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
        Nds[ty][tx] = Nd[(m*TILE_WIDTH + ty)*Width + Col];
        __syncthreads();

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

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

Control Flow Divergence

Memory Coalescing

Shared Memory Bank Conflicts

Occupancy

Loop Unrolling

Kernel Launch Overhead
Shared Memory

- Shared memory is banked (Fermi: 32 banks)
  - Consecutive 32-bit words are in different banks
  - Simultaneous & concurrent access
    - On Fermi, all the threads in the same wrap share the same request
- If two or more threads access the same bank but different words, get bank conflicts
  - Multiple threads access the same bank for same word → no bank conflict
Matrix Multiplication Kernel Using Multiple Blocks with Tile

```c
__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;
}
```

▶ Do we have a *shared memory bank conflict* problem here?
Outline

Control Flow Divergence

Memory Coalescing

Shared Memory Bank Conflicts

Occupancy

Loop Unrolling

Kernel Launch Overhead
Reminder: Thread Scheduling

- SM implements zero-overhead warp scheduling
  - At any time, only one of the warps is executed by SM
  - Warps whose next instruction has its inputs ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a warp execute the same instruction when selected

TB1, W1 stall
TB2, W1 stall
TB3, W2 stall

Instruction:

| 1 | 2 | 3 | 4 | 5 | 6 | 1 | 2 | 1 | 2 | 3 | 4 | 7 | 8 | 1 | 2 | 1 | 2 | 3 | 4 |
| TB1 | W1 | TB2 | W1 | TB3 | W1 | TB3 | W2 | TB2 | W1 | TB1 | W1 | TB1 | W2 | TB1 | W3 | TB1 | W2 | TB3 | W2 |

TB = Thread Block, W = Warp
Thread Scheduling

- What happens if all warps are stalled?
  - No instruction issued $\rightarrow$ performance lost
- Most common reason for stalling
  - Waiting on global memory
- If your code reads global memory every couple of instructions
  - Try to maximize occupancy
- What determines occupancy?
  - Register usage per thread
    - Registers are dynamically partitioned across all blocks assigned to the SM
    - Once assigned to a block, the register is NOT accessible by threads in other blocks
    - Each thread in the same block only access registers assigned to itself
  - Shared memory per thread block
Pool of registers and shared memory per SM

- Each thread block grabs registers & shared memory
- If one or the other is fully utilized → no more thread blocks
How do you know what you’re using?

- Use `nvcc --ptxas-options=-v` to get register & shared memory usage
- Plug those numbers into CUDA Occupancy Calculator
- How to influence how many registers you use?
  - Pass option `--maxrregcount=X` to nvcc
Outline

Control Flow Divergence

Memory Coalescing

Shared Memory Bank Conflicts

Occupancy

Loop Unrolling

Kernel Launch Overhead
How many instructions are required to be carried out in each iteration?

```cpp
for (int k = 0; k < BLOCK_SIZE; ++k) {
    Pvalue += Ms[ty][k] * Ns[k][tx];
}
```
Limited Processing Bandwidth of An SM

```c
for (int k = 0; k < BLOCK_SIZE; ++k) {
    Pvalue += Ms[ty][k] * Ns[k][tx];
}
```

- How many instructions are required to be carried out in each iteration?
  - Two floating-point arithmetic instructions
  - One loop branch instruction
  - Two address arithmetic instruction
  - One loop counter increment instruction

- Only 13 of the instructions executed are for real computation!!!
Limited Processing Bandwidth of An SM

```
for (int k = 0; k < BLOCK_SIZE; ++k) {
    Pvalue += Ms[ty][k] * Ns[k][tx];
}
```

- How many instructions are required to be carried out in each iteration?
  - Two floating-point arithmetic instructions
  - One loop branch instruction
  - Two address arithmetic instruction
  - One loop counter increment instruction
  - Only $\frac{1}{3}$ of the instructions executed are for real computation!!!
Loop Unrolling

// Assume BLOCK_SIZE = 16
Pvalue = Ms[ty][0] * Ns[0][tx] + Ms[ty][1] * Ns[1][tx] + ...
    + Ms[ty][15] * Ns[15][tx];

- Loop branch instructions $\rightarrow$ gone
- Loop counter increment instructions $\rightarrow$ gone
- Address arithmetic instructions $\rightarrow$ gone
  - Indices are constants
  - Compiler is able to eliminate address arithmetic instructions
- Only floating-point arithmetic instructions are still there
Loop Unrolling

// Assume BLOCK_SIZE = 16
Pvalue = Ms[ty][0] \times Ns[0][tx] + Ms[ty][1] \times Ns[1][tx] + ...  
+ Ms[ty][15] \times Ns[15][tx];

- Loop branch instructions → gone
- Loop counter increment instructions → gone
- Address arithmetic instructions → gone
  - Indices are constants
  - Compiler is able to eliminate address arithmetic instructions
- Only floating-point arithmetic instructions are still there
  - Close to peak performance!!!
Outline

Control Flow Divergence

Memory Coalescing

Shared Memory Bank Conflicts

Occupancy

Loop Unrolling

Kernel Launch Overhead
Kernel Launch Overhead

- Kernel launches are not free
  - A null kernel launch will take non-trivial time
  - Actual number changes with HW generations and driver software
  - If you are launching lots of small grids you will lose substantial performance due to this effect

- Independent kernel launches are cheaper than dependent kernel launches
  - Dependent launch: Some readback to the cpu

- If you are reading back data to the cpu for control decisions, consider doing it on the GPU
  - Even though the GPU is slow at serial tasks, can do surprising amounts of work before you used up kernel launch overhead