CUDA Performance Considerations (2 of 2)

Patrick Cozzi
University of Pennsylvania
CIS 565 - Spring 2011

Administrivia

- Friday 03/04, 11:59pm
- Assignment 4 due
- Presentation date change due via email
  - Not bonus day eligible
- Course Networking
  - Monday 03/14 or
  - Friday 04/29

Survey

- What are you interested in?
  - More performance
  - More parallel algorithms, e.g., sorting
  - OpenCL
  - Fermi, e.g., NVIDIA GeForce GTX 480

Agenda

- Data Prefetching
- Loop Unrolling
- Thread Granularity
- Bank Conflicts
- Review Final Project
Data Prefetching

- Independent instructions between a global memory read and its use can hide memory latency

```c
float m = Md[i];
float f = a * b + c * d;
float f2 = m * f;
```

Data Prefetching

- Independent instructions between a global memory read and its use can hide memory latency

```c
float m = Md[i];  // Read global memory
float f = a * b + c * d;
float f2 = m * f;
```

Data Prefetching

- Independent instructions between a global memory read and its use can hide memory latency

```c
float m = Md[i];
float f = a * b + c * d;
float f2 = m * f;  // Use global memory after the above line executes in enough warps hide the memory latency
```
Data Prefetching

- Prefetching data from global memory can effectively increase the number of independent instructions between global memory read and use.

Recall tiled matrix multiply:

```c
for (/* ... */)
{
    // Load current tile into shared memory
    __syncthreads();
    // Accumulate dot product
    __syncthreads();
}
```

Tiled matrix multiply with prefetch:

```c
for (/* ... */)
{
    // Load first tile into registers
    // Load current tile into shared memory
    __syncthreads();
    // Accumulate dot product
    __syncthreads();
}
```
Tiled matrix multiply with prefetch:

```c
// Load first tile into registers
for (/* ... */)
{
    // Deposit registers into shared memory
    __syncthreads();
    // Load next tile into registers
    // Accumulate dot product
    __syncthreads();
}
```

Cost
- Added complexity
- More registers – what does this imply?

Loop Unrolling

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

Instructions per iteration
- One floating-point multiple
- One floating-point add
- What else?
Loop Unrolling

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

- Other instructions per iteration
  - Update loop counter

- Instruction Mix
  - 2 floating-point arithmetic instructions
  - 1 loop branch instruction
  - 2 address arithmetic instructions
  - 1 loop counter increment instruction
Loop Unrolling

- Only 1/3 are floating-point calculations
  - But I want my full theoretical 346.5 GFLOPs (G80)
  - Consider loop unrolling

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

- No more loop
  - No loop count update
  - No branch
  - Constant indices – no address arithmetic instructions

Thread Granularity

- How much work should one thread do?
  - Parallel Reduction
    - Reduce two elements?
  - Matrix multiply
    - Compute one element of Pd?
Thread Granularity

- Matrix Multiple
  - Both elements of $P_d$ require the same row of $M_d$
  - Compute both $P_d$ elements in the same thread
    - Reduces global memory access by $\frac{1}{4}$
    - Increases number of independent instructions
      - What is the benefit?
    - New kernel uses more registers and shared memory
      - What does that imply?

Matrix Multiply

- What improves performance?
  - Prefetching?
  - Loop unrolling?
  - Thread granularity?
- For what inputs?

Matrix Multiply

- What improves performance?
  - Prefetching?
  - Loop unrolling?
  - Thread granularity?
- For what inputs?
Matrix Multiply

8x8 Tiles
- Coarser thread granularity helps
- Prefetching doesn’t
- Loop unrolling doesn’t

16x16 Tiles
- Coarser thread granularity helps
- Full loop unrolling can help
- Prefetch helps for 1x1 tiling

Image from http://courses.engr.illinois.edu/ece498/Book/Chapter5-CudaPerformance.pdf
Bank Conflicts

- Shared memory is the same speed as registers...\textit{usually}
  - Registers – per thread
  - Shared memory – per block

- Shared memory access patterns can affect performance. Why?

<table>
<thead>
<tr>
<th>Thread block slots</th>
<th>G80 Limits</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>768</td>
<td></td>
</tr>
<tr>
<td>8K registers</td>
<td></td>
</tr>
<tr>
<td>16K</td>
<td></td>
</tr>
</tbody>
</table>

Bank Conflicts

- G80 Banks
  - 16 banks. Why?
  - Per-bank bandwidth: 32-bits per two cycles
  - Successive 32-bit words are assigned to successive banks
    - Bank = address \% 16
    - Why?

- Banks
  - Each bank can service one address per two cycle
  - \textit{Bank Conflict}: Two simultaneous accesses to the same bank, but not the same address
    - Serialized
Bank Conflicts

- Linear addressing
  - stride == 1

- Random 1:1 Permutation

- Linear addressing
  - stride == 2

- Linear addressing
  - stride == 8

Fast Path 1
- All threads in a half-warp access different banks

Fast Path 2
- All threads in a half-warp access the same address
Bank Conflicts

- Slow Path
  - Multiple threads in a half-warp access the same bank
  - Access is serialized
  - What is the cost?

```
__shared__ float shared[256];
// ...
float f = shared[index + s * threadIdx.x];
```

- For what values of \( s \) is this conflict free?
  - Hint: The G80 has 16 banks

Without using a profiler, how can we tell what kind of speedup we can expect by removing bank conflicts?

What happens if more than one thread in a warp writes to the same shared memory address (non-atomic instruction)?