CUDA Performance Considerations
(1 of 2)

Patrick Cozzi
University of Pennsylvania
CIS 565 - Spring 2011

Administrivia
- Presentation topic due today
  - via email by 11:59pm
  - Or I pick your topic 😊
  - Not bonus day eligible
- Assignment 4 due Friday, 03/04

Agenda
- Warps Revisited
- Parallel Reduction Revisited
- Warp Partitioning
- Memory Coalescing
- Dynamic Partitioning of SM Resources
- Data Prefetching

Warps
- Recall a warp
  - Group of threads from a block
  - G80 / GT200 – 32 threads per warp
  - Each thread executes the same instruction on different data
  - The unit of scheduling
  - An implementation detail
    - Not if you want performance
Warps
- 32 threads per warp but 8 SPs per SM. What gives?

When an SM schedules a warp:
- Its instruction is ready
- 8 threads enter the SPs on the 1st cycle
- 8 more on the 2nd, 3rd, and 4th cycles
- Therefore, 4 cycles are required to execute any instruction

Question
- A kernel has
  - 1 global memory read (200 cycles)
  - 4 non-dependent multiples/adds
- How many warps are required to hide the memory latency?

Solution
- Each warp has 4 multiples/adds
  - 16 cycles
- We need to cover 200 cycles
  - \( \frac{200}{16} = 12.5 \)
  - \( \text{ceil}(12.5) = 13 \)
- 13 warps are required
Efficient data-parallel algorithms + Optimizations based on GPU Architecture = Maximum Performance

Parallel Reduction

- Recall Parallel Reduction (sum)

```
0 1 2 3 4 5 6 7
```

Parallel Reduction

```
0 1 2 3 4 5 6 7
```

Parallel Reduction

```
0 1 2 3 4 5 6 7
```

Parallel Reduction

```
0 1 2 3 4 5 6 7
```
Parallel Reduction

- Similar to brackets for a basketball tournament
- $\log(n)$ passes for $n$ elements
- How would you implement this in CUDA?

```c
__shared__ float partialSum[];
// ... load into shared memory
unsigned int t = threadIdx.x;
for (unsigned int stride = 1; stride < blockDim.x; stride *= 2) {
    __syncthreads();
    if (t % (2 * stride) == 0)
        partialSum[t] += partialSum[t + stride];
}
```

Code from: http://courses.engr.illinois.edu/ece498/al/Syllabus.html
__shared__ float partialSum[];
// ... load into shared memory
unsigned int t = threadIdx.x;
for (unsigned int stride = 1;
    stride < blockDim.x;
    stride *= 2) {
    __syncthreads();
    if (t % (2 * stride) == 0)
        partialSum[t] +=
            partialSum[t + stride];
}
Parallel Reduction

1st pass: threads 1, 3, 5, and 7 don’t do anything
Really only need \( n/2 \) threads for \( n \) elements

2nd pass: threads 2 and 6 also don’t do anything

3rd pass: thread 4 also doesn’t do anything

In general, number of required threads cuts in half after each pass
Parallel Reduction

- What if we *tweaked* the implementation?
Parallel Reduction

Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

```c
__shared__ float partialSum[]
// ... load into shared memory
unsigned int t = threadIdx.x;
for(unsigned int stride = blockDim.x >> 1;
    stride > 0;
    stride >>= 1) {
    __syncthreads();
    if (t < stride)
        partialSum[t] += partialSum[t + stride];
}
```

Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html
Parallel Reduction

- 1st pass: threads 4, 5, 6, and 7 don’t do anything
- Really only need \( n/2 \) threads for \( n \) elements

Parallel Reduction

- 2nd pass: threads 2 and 3 also don’t do anything

Parallel Reduction

- 3rd pass: thread 1 also doesn’t do anything

Parallel Reduction

- What is the difference?

\[ \text{stride} = 1, 2, 4, \ldots \]

\[ \text{stride} = 4, 2, 1, \ldots \]
Parallel Reduction

- What is the difference?

```
if (t % (2 * stride) == 0)
    partialSum[t] +=
    partialSum[t + stride];

if (t < stride)
    partialSum[t] +=
    partialSum[t + stride];
```

| stride = 1, 2, 4,... | stride = 4, 2, 1,... |

Warp Partitioning

- **Warp Partitioning**: how threads from a block are divided into warps
- Knowledge of warp partitioning can be used to:
  - Minimize divergent branches
  - Retire warps early

Understand warp partitioning ➔ make your code run faster

Warp Partitioning

- Partition based on *consecutive increasing* threadIdx
Warp Partitioning

- 1D Block
  - `threadIdx.x` between 0 and 512 (G80/GT200)
  - Warp $n$
    - Starts with thread $32n$
    - Ends with thread $32(n + 1) - 1$
  - Last warp is padded if block size is not a multiple of 32

Image from http://courses.engr.illinois.edu/ece498/al/textbook/Chapter5-CudaPerformance.pdf

- 2D Block
  - Increasing `threadIdx.x` means
    - Increasing `threadIdx.x`
    - Starting with row `threadIdx.y == 0`

- 2D Block
  - Linearized order

- 3D Block
  - Start with `threadIdx.z == 0`
  - Partition as a 2D block
  - Increase `threadIdx.z` and repeat
Warp Partitioning

Divergent branches are within a warp!

For any `warpSize`, does any warp have a divergent branch with this code:

```c
if (threadIdx.x > warpSize)
    // ...
```

Given knowledge of warp partitioning, which parallel reduction is better?

```c
if (t % (2 * stride) == 0)
    partialSum[t] += partialSum[t + stride];
```

```c
if (t < stride)
    partialSum[t] += partialSum[t + stride];
```

```
stride = 1, 2, 4, ...
```

```
stride = 4, 2, 1, ...
```
Warp Partitioning

- Pretend $\text{warpSize} == 2$

Strides:
- First pass: $1, 2, 4, ...$
- Second pass: $4, 2, 1, ...$

1st Pass

2nd Pass
Warp Partitioning

- 2nd Pass

Good partitioning also allows warps to be retired early.

- Better hardware utilization

```
if (t < stride)
    partialSum[t] += partialSum[t + stride];
```

Parallel Reduction

1st Pass
Warp Partitioning

1st Pass

Warp Partitioning

2nd Pass

Warp Partitioning

2nd Pass

Warp Partitioning

Memory Coalescing

Given a matrix stored row-major in global memory, what is a thread’s desirable access pattern?

\[
\begin{bmatrix}
M_{11} & M_{12} & M_{13} & M_{14} \\
M_{21} & M_{22} & M_{23} & M_{24} \\
M_{31} & M_{32} & M_{33} & M_{34} \\
M_{41} & M_{42} & M_{43} & M_{44}
\end{bmatrix}
\]
Memory Coalescing

- Given a matrix stored *row-major* in *global memory*, what is a thread's desirable access pattern?
  - a) column after column?
  - b) row after row?

---

Memory Coalescing

- Given a matrix stored *row-major* in *global memory*, what is a thread's desirable access pattern?
  - a) column after column
    - Individual threads read increasing, consecutive memory addresses
  - b) row after row
    - Adjacent threads read increasing, consecutive memory addresses
Memory Coalescing

Recall warp partitioning: if these threads are in the same warp, global memory addresses are increasing and consecutive across warps.

Memory Coalescing

- **Memory coalescing** – rearrange access patterns to improve performance
- Useful today but will be less useful with large on-chip caches

Memory Coalescing

- Global memory bandwidth (DRAM)
  - G80 – 86.4 GB/s
  - GT200 – 150 GB/s
- Achieve peak bandwidth by requesting large, consecutive locations from DRAM
  - Accessing random location results in much lower bandwidth

Memory Coalescing

- The GPU coalesce consecutive reads in a half-warp into a single read
  - What makes this possible?
  - **Strategy:** read global memory in a coalesce-able fashion into shared memory
    - Then access shared memory randomly at maximum bandwidth
      - Ignoring bank conflicts – next lecture

See Appendix G in the NVIDIA CUDA C Programming Guide for coalescing alignment requirements.
SM Resource Partitioning

- Recall a SM dynamically partitions resources:

  - Registers
  - Thread block slots
  - Thread slots
  - Shared memory

We can have:
- 8 blocks of 96 threads
- 4 blocks of 192 threads
- But not 8 blocks of 192 threads

SM Resource Partitioning

<table>
<thead>
<tr>
<th>G80 Limits</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
</tr>
<tr>
<td>768</td>
</tr>
<tr>
<td>8K registers / 32K memory</td>
</tr>
<tr>
<td>16K</td>
</tr>
</tbody>
</table>

SM Resource Partitioning

<table>
<thead>
<tr>
<th>G80 Limits</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
</tr>
<tr>
<td>768</td>
</tr>
<tr>
<td>8K registers / 32K memory</td>
</tr>
<tr>
<td>16K</td>
</tr>
</tbody>
</table>

SM Resource Partitioning

- We can have (assuming 256 thread blocks):
  - 768 threads (3 blocks) using 10 registers each
  - 512 threads (2 blocks) using 11 registers each

SM Resource Partitioning

<table>
<thead>
<tr>
<th>G80 Limits</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
</tr>
<tr>
<td>768</td>
</tr>
<tr>
<td>8K registers / 32K memory</td>
</tr>
<tr>
<td>16K</td>
</tr>
</tbody>
</table>
SM Resource Partitioning

- We can have (assuming 256 thread blocks)
  - 768 threads (3 blocks) using 10 registers each
  - 512 threads (2 blocks) using 11 registers each

- More registers decreases thread-level parallelism
- Can it ever increase performance?

### G80 Limits

<table>
<thead>
<tr>
<th>Thread block slots</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread slots</td>
<td>768</td>
</tr>
<tr>
<td>Registers</td>
<td>8K registers / 32K memory</td>
</tr>
<tr>
<td>Shared memory</td>
<td>16K</td>
</tr>
</tbody>
</table>

---

SM Resource Partitioning

- **Performance Cliff.** Increasing resource usage leads to a dramatic reduction in parallelism
  - For example, increasing the number of registers, unless doing so hides latency of global memory access

---

Data Prefetching

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

```plaintext
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

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

Data Prefetching

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

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

Data Prefetching

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

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

- 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();
}
```

Data Prefetching

- 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();
}
```

Data Prefetching

- Tiled matrix multiply with prefetch:

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

Data Prefetching

- Tiled matrix multiply with prefetch:

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