Lecture 11: Performance Guidelines II

Rui Wang
Parameter Space

- Number of threads per block
- Usage of GPU resources:
  - Shared memory
  - Register space
- Optimizations can change the resource usage:
  - Data prefetching
  - Loop unrolling
Dynamic Partitioning of Resources

- Resources on each multiprocessor (G80):
  - Registers: 8K (or 32KB)
  - Shared memory: 16KB
  - Threads slots: 768
  - Block slots: 8

- On GT200:
  - Registers: 16K (or 64KB)
  - Threads slots: 1024
Dynamic Partitioning of Resources

- Block is the minimum unit for scheduling on each MP.
- The number of blocks assigned to each MP depends on the bottleneck of the resource consumption.
- Example (G80):
  - 256 threads/block $\rightarrow$ 3 blocks (threads slots limit)
  - 10 regs/thread $\rightarrow$ 2560 regs/block $\rightarrow$ 3 blocks (reg limit)
  - 4KB smem/block $\rightarrow$ 4 blocks (smem limit)
- Result: 3 blocks will be scheduled on each MP
Dynamic Partitioning of Resources

- Block is the minimum unit for scheduling on each MP.
- The number of blocks assigned to each MP depends on the bottleneck of the resource consumption.
- Example (G80):
  - 256 threads/block → 3 blocks (threads slots limit)
  - 11 regs/thread → 2816 regs/block → 2 blocks (reg limit)
  - 4KB smem/block → 4 blocks (smem limit)
- Result: 2 blocks will be scheduled on each MP (number of threads drops from 768 to 512)
Thread Granularity

- Squeeze more work in each thread can reduce redundant work (take matrix mult as an example).
- Tiling can reduce the amount of GMEM accesses.
- However, must consider resource usage: given 3 blocks per MP, each block can consume slightly more than 5KB of SMEM.
Thread Granularity

- Results of performance, by varying block size, loop unrolling, data prefetching, thread granularity.
Thread Granularity

- Results of performance, by varying block size, loop unrolling, data prefetching, thread granularity.

18 GFLOPS to 120 GFLOPS
Thread Granularity

- Parameterizing your applications helps adaptation to different GPUs
- You can try to make applications self-tuning (like FFTW)
  - Experiment different configurations, discovers and saves the optimal one.
Performance Optimization Example

- Let's take another look at parallel reduction:

![Parallel Reduction Diagram]

Values (shared memory)

Step 1  Stride 8
Thread IDs: 0 1 2 3 4 5 6 7
Values: 10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2

Step 2  Stride 4
Thread IDs: 0 1 2 3
Values: 8 -2 10 6 0 9 3 7 -2 -3 2 7 0 11 0 2

Step 3  Stride 2
Thread IDs: 0 1
Values: 8 7 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2

Step 4  Stride 1
Thread IDs: 0
Values: 21 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2
Performance Optimization Example

- Let's take another look at parallel reduction:

```c
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
  if (tid < s) {
    sdata[tid] += sdata[tid + s];
  }
  __syncthreads();
}
```
Performance Optimization Example

• Let's take another look at parallel reduction:
  • As reduction proceeds, # of active threads decreases, especially when $s \leq 32$, only one warp
  • Instructions execute in lock-step within a warp
  • Therefore, when $s \leq 32$
    - We don't need `__syncthreads()` any more
    - No if $(tid < s)$ necessary because it doesn't save any work
Performance Optimization Example

- Unroll the last warp:

```c
for (unsigned int s=blockDim.x/2; s>32; s>>=1)
{
    if (tid < s)
        sdata[tid] += sdata[tid + s];
    __syncthreads();
}

if (tid < 32)
{
    sdata[tid] += sdata[tid + 32];
    sdata[tid] += sdata[tid + 16];
    sdata[tid] += sdata[tid + 8];
    sdata[tid] += sdata[tid + 4];
    sdata[tid] += sdata[tid + 2];
    sdata[tid] += sdata[tid + 1];
}
```
Performance Optimization Example

- Completely unroll the loop
  - We know that the block size is limited by the GPU to 512 threads, so we know the loop actually doesn't execute for many times
  - $512 \rightarrow 256 \rightarrow 128 \rightarrow 64 \rightarrow 32 \ldots \rightarrow 1$
  - So we can completely unroll the loop
Performance Optimization Example

- Completely unroll the loop:

```c
if (blockSize >= 512) {
    if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads();
}
if (blockSize >= 256) {
    if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads();
}
if (blockSize >= 128) {
    if (tid < 64) { sdata[tid] += sdata[tid + 64]; } __syncthreads();
}
if (tid < 32) {
    if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
    if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
    if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
    if (blockSize >= 8)  sdata[tid] += sdata[tid + 4];
    if (blockSize >= 4)  sdata[tid] += sdata[tid + 2];
    if (blockSize >= 2)  sdata[tid] += sdata[tid + 1];
}
```
Performance Optimization Example

- Completely unroll the loop:
  use templates to help remove the if's at compile time

```c
if (blockSize >= 512) {
    if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads();
}
if (blockSize >= 256) {
    if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads();
}
if (blockSize >= 128) {
    if (tid < 64) { sdata[tid] += sdata[tid + 64]; } __syncthreads();
}
if (tid < 32) {
    if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
    if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
    if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
    if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
    if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
    if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
}
```
## Performance Optimization Example

### Performance for 4M element reduction

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Addressing</th>
<th>Time (2^{22} ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel 1</td>
<td>interleaved addressing</td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>with divergent branching</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 2</td>
<td>interleaved addressing</td>
<td>3.456 ms</td>
<td>4.854 GB/s</td>
<td>2.33x</td>
<td>2.33x</td>
</tr>
<tr>
<td></td>
<td>with bank conflicts</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 3</td>
<td>sequential addressing</td>
<td>1.722 ms</td>
<td>9.741 GB/s</td>
<td>2.01x</td>
<td>4.68x</td>
</tr>
<tr>
<td>Kernel 4</td>
<td>first add during global load</td>
<td>0.965 ms</td>
<td>17.377 GB/s</td>
<td>1.78x</td>
<td>8.34x</td>
</tr>
<tr>
<td>Kernel 5</td>
<td>unroll last warp</td>
<td>0.536 ms</td>
<td>31.289 GB/s</td>
<td>1.8x</td>
<td>15.01x</td>
</tr>
<tr>
<td>Kernel 6</td>
<td>completely unrolled</td>
<td>0.381 ms</td>
<td>43.996 GB/s</td>
<td>1.41x</td>
<td>21.16x</td>
</tr>
<tr>
<td>Kernel 7</td>
<td>multiple elements per thread</td>
<td>0.268 ms</td>
<td>62.671 GB/s</td>
<td>1.42x</td>
<td>30.04x</td>
</tr>
</tbody>
</table>

Kernel 7 on 32M elements: 73 GB/s!