Case Study: Reduction

General-purpose Programming of Massively Parallel Graphics Processors
Shiraz University, Spring 2010
Instructor: Reza Azimi

The slides are primarily adapted from:
Andreas Moshovos’ Course at the University of Toronto

Reduction Operations

- Multiple values are reduced into a single value
  - ADD, MUL, AND, OR, ....

  10 11 12 13 → 46

- Useful primitive in parallel computing
  - Many tasks generating intermediate results
  - Reduction must be done to combine the intermediate results into final results

- Easy enough to allow us to focus on optimization techniques
### Sequential Reduction

```java
sum = array[0];
for (k = 1; k < N; k++) {
    sum += array[i];
}
```

- Start with the first two elements --> partial result
- Process the next element
- $O(N)$

### Parallel Reduction

$log_2(N)$, where $N$ is the number of elements
Trees with Larger Degrees

log_4(N), where N is the number of elements

CUDA Impl.: Single Thread Block

```c
__global__ void reduce(float *g_idata, float *g_odata, int n) {
    int tid = blockIdx.x;

    // copy input data into output data
    g_odata[tid] = g_idata[tid];

    for (int s = 1; s < blockDim.x; s *= 2) {
        if ((tid % (2*s)) == 0) {
            g_odata[tid] += g_odata[tid + s];
        }
    }
    __syncthreads();
}
```

Why do we need this?
Using Shared Memory

```c
__global__ void reduce(float *g_idata, float *g_odata, int n)
{
    int tid = blockIdx.x;
    __shared__ float s_data[BLOCK_DIM];
    // copy input data into output data
    s_data[tid] = g_idata[tid];
    __syncthreads();

    for (int s = 1; s < blockDim.x; s *= 2) {
        if ((tid % (2*s)) == 0) {
            s_data[tid] += s_data[tid + s];
        }
        __syncthreads();
    }
    g_odata[tid] = s_data[tid];
}
```

Reduction Steps

<table>
<thead>
<tr>
<th>Step 1</th>
<th>Stride 1</th>
<th>Thread IDs</th>
<th>Values (shared memory)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>10 1 8 -1 0 2 3 5 -2 -3 2 7 0 11 0 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 2</th>
<th>Stride 2</th>
<th>Thread IDs</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>11 1 7 -1 -2 -2 8 5 -5 -3 9 7 11 2 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 3</th>
<th>Stride 4</th>
<th>Thread IDs</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>18 1 7 -1 6 -2 8 5 4 -3 9 7 13 11 2 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 4</th>
<th>Stride 8</th>
<th>Thread IDs</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>24 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 5</th>
<th>Stride 16</th>
<th>Thread IDs</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>41 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2</td>
</tr>
</tbody>
</table>
```
How about Multiple Thread Blocks

- How do we communicate results across blocks?

- The key problem is **synchronization**:
  - How do we know that each block has finished?

Global Synchronization

- **IF** there was such a thing:
CUDA Doesn’t Support Global Synchronization

- Expensive to implement
- We have to choose between limited blocks or deadlock

Deadlock

Synchronization Never Achieved
Global Synchronization: Summary

- If there was global sync
  - Global sync after each block
  - Once all blocks are done, continue recursively

- CUDA does not have global sync:
  - Expensive to support
  - Would limit the number of blocks
  - Otherwise deadlock will occur
    - Once a block gets assigned to an SM it stays there
    - Each SM can take only 8 blocks
    - So, at most #SMs x 8 blocks could be active at any given point of time

- Solution: Decompose into multiple kernels

Implicit Synchronization between Kernels

Overhead of launching a new kernel non-negligible: measure it!
Reduction: Big Picture

- The code for all levels is the same
- The same kernel code can be called multiple times

Optimization Goal: Get Max Performance

- Two components:
  - Compute Bandwidth: GFLOPs
  - Memory Bandwidth: GB/s

- Reductions typically have low arithmetic intensity
  - FLOPs/element loaded from memory

- Bandwidth will be the limiter

- For the GT280
  - 512-bit interface, 1.1GHz DDR3
  - \( (512 / 8) \times 1.1 \times 2 = 140 \text{ GB/s} \)
Performance for kernel #1

<table>
<thead>
<tr>
<th>Time</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.29 ms</td>
<td>3.91 GB/s</td>
</tr>
</tbody>
</table>

kernel #1

Thread Block size = 128 for all experiments

Bandwidth Calculation:
- Each block processes 128 elements and does:
  - 128 global memory reads & 1 global memory write
  - Each Access is 4 bytes
- At each kernel:
  - \( N \) (element reads) + \( \frac{N}{128} \) (element writes)
  - For the next kernel \( N = \frac{N}{128} \)

Kernel #1 Access Pattern

Values (shared memory)
Kernel #1: Divergent Branch

```c
__global__ void reduce1(float *g_idata, float *g_odata) {
    extern __shared__ float sdata[];
    int tid = threadIdx.x;
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();
    // do reduction in shared mem
    for (int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) { Highly Divergent Branch
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }
    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Divergent Branching: Warp Control Flow

Step 1
Step 2
Step 3
Step 4
Divergent Branching: Warp Control Flow

Removing the Divergent Branch

```c
for (int s=1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}

for (int s=1; s < blockDim.x; s *= 2) {
    int index = 2 * s * tid;

    if (index < blockDim.x == 0) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```
Reduction #2: Access Pattern

Reduction #2: Warp control flow
Reduction #2: Warp control flow

Performance for 4M element reduction

<table>
<thead>
<tr>
<th>Time (2^{22} ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel 1: Original</td>
<td>4.29 ms</td>
<td>3.91 GB/s</td>
<td></td>
</tr>
<tr>
<td>Kernel 2: Divergent Branch Removed</td>
<td>1.94 ms</td>
<td>8.62 GB/s</td>
<td>2.21x</td>
</tr>
</tbody>
</table>
**Detour: Specifying Shared Mem Size**

```c
__global__ void
reduce(float *g_idata, float *g_odata, int n) {
    int tid = blockIdx.x;
    __shared__ float s_data[BLOCK_DIM];

    // copy input data into output data
    s_data[tid] = g_idata[tid];
    __syncthreads();
    ...
```

This requires us to know the size of the thread block in advance, but we don’t!

---

**Detour: Specifying Shared Mem Size**

```c
__global__ void
reduce(float *g_idata, float *g_odata, int n) {
    int tid = blockIdx.x;
    extern __shared__ float s_data[];

    // copy input data into output data
    s_data[tid] = g_idata[tid];
    __syncthreads();

    using extern keyword allows the size of the shared memory to be specified at runtime

    SMemSize = threads * sizeof(float);
    reduce <<<blocks, threads, SMemSize>>>(d_idata, d_odata, n);
```

Kernel Launch:
Problem: Shared Memory Bank Conflicts

Values (shared memory)

<table>
<thead>
<tr>
<th>Step 1</th>
<th>Thread IDs</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stride 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>-1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2-way</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 2</td>
<td>Thread IDs</td>
<td>Values</td>
</tr>
<tr>
<td>Stride 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2-way</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 3</td>
<td>Thread IDs</td>
<td>Values</td>
</tr>
<tr>
<td>Stride 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2-way</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 4</td>
<td>Thread IDs</td>
<td>Values</td>
</tr>
<tr>
<td>Stride 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>none</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2-way bank conflicts at every step
Reminder: There are 16 threads in each half-warp

Reduction #3: Sequential Accesses

Values (shared memory)

<table>
<thead>
<tr>
<th>Step 1</th>
<th>Thread IDs</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stride 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>-1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Step 2</td>
<td>Thread IDs</td>
<td>Values</td>
</tr>
<tr>
<td>Stride 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Step 3</td>
<td>Thread IDs</td>
<td>Values</td>
</tr>
<tr>
<td>Stride 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>13</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Step 4</td>
<td>Thread IDs</td>
<td>Values</td>
</tr>
<tr>
<td>Stride 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>20</td>
<td>13</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Step 5</td>
<td>Thread IDs</td>
<td>Values</td>
</tr>
<tr>
<td>Stride 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>20</td>
<td>13</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
Reduction #3: Sequential Access

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

Performance for 4M element reduction

<table>
<thead>
<tr>
<th>Kernel</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: Original</td>
<td>4.29 ms</td>
<td>3.91 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 2: Divergent Branch Removed</td>
<td>1.94 ms</td>
<td>8.62 GB/s</td>
<td>2.21x</td>
<td>2.21x</td>
</tr>
<tr>
<td>Kernel 3: Bank Conflicts Removed</td>
<td>0.97 ms</td>
<td>17.2 GB/s</td>
<td>2.00x</td>
<td>4.42x</td>
</tr>
</tbody>
</table>
Problem: Bad Resource Utilization

- Threads are idle
  - At each step half of the remaining threads become idle

- Why did we create so many threads?
  - For loading the data from global memory to shared memory
  - Each thread reads one element from global memory to shared memory

Reduction #4: Threads Read Two Elements

```c
// each thread loads one element from global to shared mem
int tid = threadIdx.x;
int i = blockIdx.x * blockDim.x + threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();
...
```

```c
// each thread loads two elements from global to shared mem
// end performs the first step of the reduction
int tid = threadIdx.x;
int i = blockIdx.x * blockDim.x * 2 + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i + blockDim.x];
__syncthreads();
...
```
Performance for 4M element reduction

<table>
<thead>
<tr>
<th>Kernel</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: Original</td>
<td>4.29 ms</td>
<td>3.91 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 2: Divergent Branch Removed</td>
<td>1.94 ms</td>
<td>8.62 GB/s</td>
<td>2.21 x</td>
<td>2.21 x</td>
</tr>
<tr>
<td>Kernel 3: Bank Conflicts Removed</td>
<td>0.97 ms</td>
<td>17.2 GB/s</td>
<td>2.00 x</td>
<td>4.42 x</td>
</tr>
<tr>
<td>Kernel 4: Doubling the Threads to Load from Global Memory</td>
<td>0.55 ms</td>
<td>30.2 GB/s</td>
<td>1.76 x</td>
<td>7.84 x</td>
</tr>
</tbody>
</table>

Reduction #4: Still way off

- Memory bandwidth is still underutilized
  - We know that reductions have low arithmetic density

- What is the potential bottleneck?
  - Instructions that are not loads, stores, or arithmetic for the core computation
  - Address arithmetic and loop overhead

- Unroll loops to eliminate these “extra” instructions
Loop Unrolling

for (i=0; i<1000; i++)
  a[i] = b[i] + c[i];

for (i=0; i<1000; i+=4) {
  a[i] = b[i] + c[i];
  a[i+1] = b[i+1] + c[i+1];
  a[i+2] = b[i+2] + c[i+2];
  a[i+3] = b[i+3] + c[i+3];
}

.L2:
  movl b(,%edx,4), %eax
  addl %eax, a(,%edx,4)
  addl $1, %edx
  cmpl $1000, %edx
  jne .L2

.L4:
  movl b(,%edx,4), %eax
  addl %eax, a(,%edx,4)
  addl %eax, a+4(,%edx,4)
  movl b+4(,%edx,4), %eax
  addl %eax, a+4(,%edx,4)
  movl b+8(,%edx,4), %eax
  addl %eax, a+8(%edx,4)
  movl b+12(,%edx,4), %eax
  addl %eax, a+12(%edx,4)
  addl $4, %edx
  cmpl $1000, %edx
  jne .L4

Unrolling the Loop for the Last Warp

- At every step the number of active threads halves
  - When \( s \leq 32 \) there is only one warp left

- Instructions are SIMD-synchronous within a warp
  - They all happen in lock step
  - No need to use __syncthreads()
  - We don’t need “if (tid < s)” since it does not save any work
    - All threads in a warp will “see” all instructions whether they execute them or not

- Unroll the last 6 iterations of the inner loop
  - \( s \leq 32 \)
Reduction #3: Warp control flow

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

Reduction #5: Unrolling the Warp

```c
// do reduction in shared mem
for (int s = blockDim.x/2; s > 32; s /= 2) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}

// there is only one warp left, so no need for
// __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];
}
```
**Unrolling the Last Warp: A Closer Look**

- Warp execution proceeds in lock step for all threads
  - All threads execute the same instruction

```c
sdata[tid] += sdata[tid + 32];
```

1. Read into a register: `sdata[tid]`
2. Read into a register: `sdata[tid+32]`
3. Add the two
4. Write: `sdata[tid]`

- Shared memory can provide up to 16 words per cycle
  - If we don’t use this capability it just gets wasted

---

**Performance for 4M element reduction**

<table>
<thead>
<tr>
<th>Kernel</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: Original</td>
<td>4.29 ms</td>
<td>3.91 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 2: Divergent Branch Removed</td>
<td>1.94 ms</td>
<td>8.62 GB/s</td>
<td>2.21x</td>
<td>2.21x</td>
</tr>
<tr>
<td>Kernel 3: Bank Conflicts Removed</td>
<td>0.97 ms</td>
<td>17.2 GB/s</td>
<td>2.00x</td>
<td>4.42x</td>
</tr>
<tr>
<td>Kernel 4: Doubling the Threads to Load from Global Memory</td>
<td>0.55 ms</td>
<td>30.2 GB/s</td>
<td>1.76x</td>
<td>7.84x</td>
</tr>
<tr>
<td>Kernel 5: Unrolling the last warp</td>
<td>0.33 ms</td>
<td>50.18 GB/s</td>
<td>1.66x</td>
<td>13.06x</td>
</tr>
</tbody>
</table>
Reduction #6: Complete Unrolling

- If we knew the number of iterations at compile time, we could completely unroll the reduction
  - Block size is limited to 512
  - We can restrict our attention to powers-of-two block sizes

- We can easily unroll for a fixed block size
  - But we need to be generic
  - How can we unroll for block sizes we don’t know at compile time?

- C++ Templates
  - CUDA supports C++ templates on device and host functions

Unrolling Using Templates

- Specify block size as a function template parameter:

```c++
template <unsigned int blockSize>
__global__ void reduce5(int *g_data, int *g_odata)
{
...
}
```

- Kernel Invocation:

```
reduce5<512><<blocks, threads, SMemSize>>(...
```
Reduction #6: Completely Unrolled

```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];
}
```

Note: all code in Red will be evaluated at compile time.

If block size isn’t known at compile time?

```c
switch (threads) {
    case 512:
        reduce5<512><<< blocks, 512, smemSize >>>(d_idata, d_odata);
        break;
    case 256:
        reduce5<256><<< blocks, 256, smemSize >>>(d_idata, d_odata);
        break;
    case 128:
        reduce5<128><<< blocks, 128, smemSize >>>(d_idata, d_odata);
        break;
    case 64:
        reduce5<64><<< blocks, 64, smemSize >>>(d_idata, d_odata);
        break;
    case 32:
        reduce5<32><<< blocks, 32, smemSize >>>(d_idata, d_odata);
        break;
    ...
```
Performance for 4M element reduction

<table>
<thead>
<tr>
<th>Kernel 1: Original</th>
<th>Time (2^22 ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4.29 ms</td>
<td>3.91 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 2: Divergent Branch Removed</td>
<td>1.94 ms</td>
<td>8.62 GB/s</td>
<td>2.21x</td>
<td>2.21x</td>
</tr>
<tr>
<td>Kernel 3: Bank Conflicts Removed</td>
<td>0.97 ms</td>
<td>17.2 GB/s</td>
<td>2.00x</td>
<td>4.42x</td>
</tr>
<tr>
<td>Kernel 4: Doubling the Threads to Load from Global Memory</td>
<td>0.55 ms</td>
<td>30.2 GB/s</td>
<td>1.76x</td>
<td>7.84x</td>
</tr>
<tr>
<td>Kernel 5: Unrolling the last warp</td>
<td>0.33 ms</td>
<td>50.18 GB/s</td>
<td>1.66x</td>
<td>13.06</td>
</tr>
<tr>
<td>Kernel 6: Complete Unrolling</td>
<td>0.26 ms</td>
<td>62.50 GB/s</td>
<td>1.23x</td>
<td>16.07</td>
</tr>
</tbody>
</table>

Summary

- Understand CUDA performance characteristics
  - Memory coalescing
  - Divergent branching
  - Bank conflicts
  - Latency hiding
- Use peak performance metrics to guide optimization
- Know how to identify type of bottleneck
  - e.g., memory, core computation, or instruction overhead
- Optimize your algorithm, then unroll loops
- Use template parameters to generate optimal code