DirectCompute
Optimizations and Best Practices
Eric Young - NVIDIA Corporation
San Jose, CA | September 20th, 2010
Contents

- Introduction
- Best Practices for GPU Optimization
- Parallel Reduction Optimization Example
Why GPUs?

• GPUs are throughput oriented processors
  – GPUs a lot of parallel processing units (FLOPs)
  – Memory latencies are covered with more workload

• Problems with a lot of work can achieve good speedups

• Must provide enough work to GPUs for efficiency and performance

• DirectCompute is an API allowing Compute Shaders on the GPU hardware efficiently
What Is DirectCompute?

- Microsoft’s standard GPU-Computing platform
  - For Windows Vista and Windows 7
  - On DX10 and DX11 hardware generations

- Another realization of the CUDA architecture
  - Sibling API to OpenCL and CUDA C
  - Shares many concepts, idioms, algorithms and optimizations
Advantages of DirectCompute

- DirectCompute allows general purpose computation on CUDA GPUs via Compute Shaders
- DirectCompute:
  - Interoperates with Direct3D resources
  - Includes all texture features (cube maps, mip-maps)
  - Similar to HLSL (DirectX Shaders)
  - Single API across all GPU vendors, on Windows
  - Some guarantees of identical results across different hardware
GPU Programming Model

DirectCompute programs decompose parallel work into groups of threads, and dispatch many thread groups to solve a problem.

**Dispatch:** 3D grid of thread groups. Hundreds of thousands of threads.

**Thread Group:** 3D grid of threads. Tens or hundreds of threads.

**numThreads** nX, nY, nZ

**Thread:** One invocation of a shader.

SV_DispatchThreadID, SV_GroupThreadID, SV_GroupID
Parallel Execution Model

- Threads in the same group run concurrently
- Threads in different groups may run concurrently
DirectCompute
Best Practices for GPU Optimizations
Memory Coalescing

- A coordinated read by a half-warp (16 threads)
- A contiguous region of global memory:
  - 64 bytes - each thread reads a word: int, float, ...
  - 128 bytes - each thread reads a double-word: int2, float2, ...
  - 256 bytes - each thread reads a quad-word: int4, float4, ...
- Additional restrictions:
  - Starting address for a region must be a multiple of region size
  - The $k$th thread in a half-warp must access the $k$th element in a block
- Exception: not all threads must be participating
  - Predicated access, divergence within a half warp
Coalesced Access: Reading floats

All threads participate

Some Threads Do Not Participate
Uncoalesced Access: Reading floats

Permuted Access by Threads

Misaligned Starting Address (not a multiple of 64)
Coalescing (Compute 1.2+ GPUs)

- Much improved coalescing capabilities in 10-series architecture

- Hardware combines addresses within a half-warp into one or more aligned segments
  - 32, 64, or 128 bytes

- All threads with addresses within a segment are serviced with a single memory transaction
  - Regardless of ordering or alignment within the segment
Compute 1.2+ *Coalesced* Access: Reading floats

Permuted access by threads

Misaligned starting address (not a multiple of 64)
Compute 1.2+ Coalesced Access: Reading floats

32-byte segment

64-byte segment

Misaligned starting address (not a multiple of 64)

Transaction size recursively reduced to minimize size
Thread Group Shared Memory (TGSM)

- Fast memory shared across threads *within a group*
  - *Not* shared across thread groups!
  - `groupshared` float2 MyArray[16][32];
  - Not persistent between `Dispatch()` calls
- Used to reduce computation
  - Use neighboring calculations by storing them in TGSM
  - E.g. Post-processing texture instructions
TGSM Performance (contd.)

- Reduce access whenever possible
  - E.g. Pack data into \texttt{uint} instead of \texttt{float4}
  - But watch out for increased ALUs!
- Basically try to read/write once per TGSM address
  - Copy to temp array can help if it can avoid duplicate accesses!
  - Ensure that you perform a thread synchronization immediately after loading your data to shared memory
- Unroll loops accessing shared mem
  - Helps compiler hide latency
Shared Memory Bank Addressing

- No Bank Conflicts
  - Linear addressing stride == 1

- No Bank Conflicts
  - Random 1:1 Permutation
Shared Memory Bank Addressing

- **2-way Bank Conflicts**
  - Linear addressing
  - stride == 2

- **8-way Bank Conflicts**
  - Linear addressing
  - stride == 8
What is Occupancy?

- GPUs typically run 1000 to 10,000’s of threads concurrently.
- Higher occupancy = More efficient utilization of the HW.
- Parallel code executed in HW through warps (32 threads) running concurrently at any given moment of time.
- Thread instructions are executed sequentially, by executing other warps, we can hide instruction and memory latencies in the HW.
- Maximizing “Occupancy”, with occupancy of 1.0 the best scenario.

\[
\text{Occupancy} = \frac{\# \text{ of resident warps}}{\text{Max possible } \# \text{ of resident warps}}
\]
Maximizing HW Occupancy

- One or more thread groups resides on a single shader unit
- Occupancy limited by resource usage:
  - Thread group size declaration
  - Thread Group Shared Memory usage
  - Number of registers used by thread group

- **Example**: HW shader unit:
  - 8 thread groups max
  - 48KB total shared memory
  - 1536 threads max

- Shader launched with a thread group size of 256 threads and uses 32KB of shared memory
  - → Can only run 1 thread group per HW shader unit
- We are limited by shared memory.
Maximizing HW Occupancy

• Registers used per thread affects occupancy:
  – “Register Pressure”
  – You have little control over this
  – Rely on drivers to do the right thing 😊

• Experimentation and tuning needed to find the right balance
  – Store different presets for best performance across a variety of GPUs
 Dispatch/Thread Group Size Heuristics

• \# of thread groups > \# of multiprocessors
  – So all multiprocessors have at least one thread group to execute

• \# of thread groups / \# of multiprocessors > 2
  – Multiple thread groups can run concurrently in a multiprocessor
  – Thread groups that aren’t waiting at a barrier keep the hardware busy
  – Subject to resource availability - registers, shared memory

• \# of thread groups > 100 to scale to future devices
  – Thread groups executed in pipeline fashion
  – 1000 groups per dispatch will scale across multiple generations

• \# threads / threadgroup a multiple of warp size
  – All threads in a warp doing work
DirectCompute Optimization Example
Parallel Reduction
Parallel Reduction

- Common and important data parallel primitive
  - (e.g. find the sum of an array)

- Easy to implement in compute shaders
  - Harder to get it right

- Serves as a great optimization example
  - We’ll walk step by step through 7 different versions
  - Demonstrates several important optimization strategies
Parallel Reduction

- Tree-based approach used within each thread block

- Need to be able to use multiple thread blocks
  - To process very large arrays
  - To keep all multiprocessors on the GPU busy
  - Each thread block reduces a portion of the array

- But how do we communicate partial results between thread blocks?
Problem: Global Synchronization

- If we could synchronize across all thread groups we can run reduce on a very large array
  - A global sync after each group produces its result
  - Once all groups reach sync, continue recursively

- But GPUs have no global synchronization. Why?
  - Expensive to build in hardware for GPUs with high processor count
  - Would force programmer to run fewer groups (no more than \# multiprocessors * \# resident groups / multiprocessor) to avoid deadlock, which may reduce overall efficiency

- Solution: decompose into multiple shader dispatches
  - A dispatch() call serves as a global synchronization point
  - Dispatch() has negligible HW overhead, low SW overhead
Solution: Shader Decomposition

- Avoid global sync by decomposing computation into multiple dispatches

- In the case of reductions, code for all levels is the same
  - Implement with recursive dispatches
What is Our Optimization Goal?

- We should strive to reach GPU peak performance
- Choose the right metric:
  - GFLOP/s: for compute-bound shaders
  - Bandwidth: for memory-bound shaders
- Reductions have very low arithmetic intensity
  - 1 flop per element loaded (bandwidth-optimal)
- Therefore we should strive for peak bandwidth
- We use a G80 GPU for this Optimization
  - 384-bit memory interface, 900 MHz DDR
  - \[ 384 \times 1800 / 8 = 86.4 \text{ GB/s} \]
  - *Optimization techniques are equally applicable to newer GPUs*
Reduction #1: Interleaved Addressing

```
RWStructuredBuffer<float> g_data;
#define groupDim_x 128
groupshared float sdata[groupDim_x];
[numthreads( groupDim_x, 1, 1)]
void reduce1( uint3 threadIdx : SV_GroupThreadID,
              uint3 groupIdx : SV_GroupID)
{
    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = groupIdx.x*groupDim_x + threadIdx.x;
    sdata[tid] = g_data[i];
    GroupMemoryBarrierWithGroupSync();

    // do reduction in shared mem
    for(unsigned int s=1; s < groupDim_x; s *= 2) {
        if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        GroupMemoryBarrierWithGroupSync();
    }

    // write result for this block to global mem
    if (tid == 0) g_data[groupIdx.x] = sdata[0];
}
```
## Parallel Reduction: Interleaved Addressing

<table>
<thead>
<tr>
<th>Values (shared memory)</th>
<th>10</th>
<th>1</th>
<th>8</th>
<th>-1</th>
<th>0</th>
<th>-2</th>
<th>3</th>
<th>5</th>
<th>-2</th>
<th>-3</th>
<th>2</th>
<th>7</th>
<th>0</th>
<th>11</th>
<th>0</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Step 1</strong></td>
<td>Thread IDs</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stride 1</td>
<td>Values</td>
<td>11</td>
<td>1</td>
<td>7</td>
<td>-1</td>
<td>-2</td>
<td>-2</td>
<td>8</td>
<td>5</td>
<td>-5</td>
<td>-3</td>
<td>9</td>
<td>7</td>
<td>11</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td><strong>Step 2</strong></td>
<td>Thread IDs</td>
<td>0</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stride 2</td>
<td>Values</td>
<td>18</td>
<td>1</td>
<td>7</td>
<td>-1</td>
<td>6</td>
<td>-2</td>
<td>8</td>
<td>5</td>
<td>4</td>
<td>-3</td>
<td>9</td>
<td>7</td>
<td>13</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td><strong>Step 3</strong></td>
<td>Thread IDs</td>
<td>0</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stride 4</td>
<td>Values</td>
<td>24</td>
<td>1</td>
<td>7</td>
<td>-1</td>
<td>6</td>
<td>-2</td>
<td>8</td>
<td>5</td>
<td>17</td>
<td>-3</td>
<td>9</td>
<td>7</td>
<td>13</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td><strong>Step 4</strong></td>
<td>Thread IDs</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stride 8</td>
<td>Values</td>
<td>41</td>
<td>1</td>
<td>7</td>
<td>-1</td>
<td>6</td>
<td>-2</td>
<td>8</td>
<td>5</td>
<td>17</td>
<td>-3</td>
<td>9</td>
<td>7</td>
<td>13</td>
<td>11</td>
<td>2</td>
</tr>
</tbody>
</table>
Performance for 4M element reduction

<table>
<thead>
<tr>
<th>Shader 1: interleaved addressing with divergent branching</th>
<th>Time ($2^{22}$ ints)</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
</tr>
</tbody>
</table>

Note: Block Size = 128 threads for all tests
Reduction #1: Interleaved Addressing

```c
void reduce1( uint3 threadIdx : SV_GroupIndex,
              uint3 groupIdx : SV_GroupID )
{
    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = groupIdx.x*groupDim_x + threadIdx.x;
    sdata[tid] = g_idata[i];
    GroupMemoryBarrierWithGroupSync();

    // do reduction in shared mem
    for(unsigned int s=1; s < groupDim_x; s *= 2) {
        if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        GroupMemoryBarrierWithGroupSync();
    }

    // write result for this block to global mem
    if (tid == 0) g_data[groupIdx.x] = sdata[0];
}
```

Problem: highly divergent branching results in very poor performance!
What is Thread Divergence?

- Divergence is the main performance concern when branching
  - Threads within a single warp take different paths
  - Different execution paths must be serialized

- Avoid divergence when branch condition is a function of thread ID
  - Example with divergence:
    - Branch granularity < warp size
    - if (threadIdx.x > 2) {}
  - Example without divergence:
    - Branch granularity is a whole multiple of warp size
    - if (threadIdx.x / WARP_SIZE > 2) {}
Reduction #2: Interleaved Addressing

Replace divergent branch in inner loop:

```c
for (unsigned int s=1; s < groupDim_x; s *= 2) {
    if (tid % (2*s) == 0) {
        sdata[tid] += sdata[tid + s];
    }
    GroupMemoryBarrierWithGroupSync();
}
```

With strided index and non-divergent branch:

```c
for (unsigned int s=1; s < groupDim_x; s *= 2) {
    int index = 2 * s * tid;
    if (index < groupDim_x) {
        sdata[index] += sdata[index + s];
    }
    GroupMemoryBarrierWithGroupSync();
}
```
Parallel Reduction: Interleaved Addressing

Shared Memory

Step 1
Stride 1
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 2
Thread IDs
0
1
2
3

Values
11
1
7
-1
-2
-2
8
5
-5
-3
9
7
11
11
2
2

Step 3
Stride 4
Thread IDs
0
1

Values
18
1
7
-1
6
-2
8
5
4
-3
9
7
13
11
2
2

Step 4
Stride 8
Thread IDs
0

Values
24
1
7
-1
6
-2
8
5
17
-3
9
7
13
11
2
2

New Problem: Shared Memory Bank Conflicts
## Performance for 4M element reduction

<table>
<thead>
<tr>
<th>Shader</th>
<th>Time (2^{22} ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shader 1:</td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>interleaved addressing with divergent branching</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shader 2:</td>
<td>3.456 ms</td>
<td>4.854 GB/s</td>
<td>2.33x</td>
<td>2.33x</td>
</tr>
<tr>
<td>interleaved addressing with bank conflicts</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Thread Group Shared Memory Bank Conflicts

- 32 banks example (each address is 32-bits)
- Banks are arranged linearly with addresses:

<table>
<thead>
<tr>
<th>Address:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
<th>31</th>
<th>32</th>
<th>33</th>
<th>34</th>
<th>35</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bank:</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>...</td>
<td>31</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>...</td>
</tr>
</tbody>
</table>

- TGSM addresses that are 32 DWORD apart use the same bank
- Accessing those addresses from multiple threads will create a bank conflict
- Declare TGSM 2D arrays as MyArray[Y][X], and increment X first, then Y
  - Essential if X is a multiple of 32!
- Padding arrays/structures to avoid bank conflicts can help
  - E.g. MyArray[16][33] instead of [16][32]
Parallel Reduction: Sequential Addressing

Sequential addressing is conflict free

<table>
<thead>
<tr>
<th>Shared Memory</th>
<th>Thread IDs</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2</td>
<td>0 1 2 3 4 5 6 7</td>
<td>8 -2 10 6 0 9 3 7 -2 -3 2 7 0 11 0 2</td>
</tr>
<tr>
<td><strong>Step 1</strong></td>
<td><strong>Stride 8</strong></td>
<td></td>
</tr>
<tr>
<td>Thread IDs</td>
<td>Values</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Thread IDs</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1</td>
<td>21 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2</td>
</tr>
<tr>
<td><strong>Step 3</strong></td>
<td><strong>Stride 2</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread IDs</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>41 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2</td>
</tr>
<tr>
<td><strong>Step 4</strong></td>
<td><strong>Stride 1</strong></td>
</tr>
</tbody>
</table>
Reduction #3: Sequential Addressing

Replace strided indexing in inner loop:

```c
for (unsigned int s=1; s < groupDim_x; s *= 2) {
    int index = 2 * s * tid;
    if (index < groupDim_x) {
        sdata[index] += sdata[index + s];
    }
    GroupMemoryBarrierWithGroupSync();
}
```

With reversed loop and threadID-based indexing:

```c
for (unsigned int s=groupDim_x/2; s>0; s>>=1) { 
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    GroupMemoryBarrierWithGroupSync();
}
```
### Performance for 4M element reduction

<table>
<thead>
<tr>
<th>Shader 1:</th>
<th>Time ($2^{12}$ ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>interleaved addressing with divergent branching</td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shader 2:</th>
<th>Time ($2^{12}$ ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>interleaved addressing with bank conflicts</td>
<td>3.456 ms</td>
<td>4.854 GB/s</td>
<td>2.33x</td>
<td>2.33x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shader 3:</th>
<th>Time ($2^{12}$ ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential addressing</td>
<td>1.722 ms</td>
<td>9.741 GB/s</td>
<td>2.01x</td>
<td>4.68x</td>
</tr>
</tbody>
</table>
Idle Threads

Problem:

```c
for (unsigned int s=groupDim_x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    GroupMemoryBarrierWithGroupSync();
}
```

Half of the threads are idle on first loop iteration!

This is wasteful…
Reduction #4: First Add During Load

Halve the number of groups, and replace single load:

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

With two loads and first add of the reduction:

```c
// perform first level of reduction,
// reading from global memory, writing to shared memory
unsigned int tid = threadIdx.x;
unsigned int i = groupIdx.x*(groupDim_x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+groupDim_x];
GroupMemoryBarrierWithGroupSync();
```
### Performance for 4M element reduction

<table>
<thead>
<tr>
<th>Shader</th>
<th>Time ($2^{22}$ ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shader 1: interleaved addressing with divergent branching</td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shader 2: interleaved addressing with bank conflicts</td>
<td>3.456 ms</td>
<td>4.854 GB/s</td>
<td>2.33x</td>
<td>2.33x</td>
</tr>
<tr>
<td>Shader 3: 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>Shader 4: 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>
</tbody>
</table>
Instruction Bottleneck

- At 17 GB/s, we’re far from bandwidth bound
  - And we know reduction has low arithmetic intensity

- Therefore a likely bottleneck is instruction overhead
  - Ancillary instructions that are not loads, stores, or arithmetic for the core computation
  - In other words: address arithmetic and loop overhead

- Strategy: unroll loops
Unrolling the Last Warp

- As reduction proceeds, # “active” threads decreases
  - When $s \leq 32$, we have only one warp left
- Instructions are SIMD synchronous within a warp
- That means when $s \leq 32$:
  - We don’t need a `barrier()` call
  - We don’t need “if (tid < s)” because it doesn’t save any work

- Let’s unroll the last 6 iterations of the inner loop
Reduction #5: Unroll the Last Warp

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

Note: This saves useless work in all warps, not just the last one!

Without unrolling, all warps execute every iteration of the for loop and if statement.
### Performance for 4M element reduction

<table>
<thead>
<tr>
<th>Shader</th>
<th>Addressing Method</th>
<th>Time (2^{22} ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>interleaved</td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>addressing with</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>divergent</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>interleaved</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>addressing with</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bank conflicts</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>sequential</td>
<td>1.722 ms</td>
<td>9.741 GB/s</td>
<td>2.01x</td>
<td>4.68x</td>
</tr>
<tr>
<td></td>
<td>addressing</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>first add during</td>
<td>0.965 ms</td>
<td>17.377 GB/s</td>
<td>1.78x</td>
<td>8.34x</td>
</tr>
<tr>
<td></td>
<td>global load</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>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>
</tbody>
</table>
Complete Unrolling

- Assuming we know the number of iterations at compile time, we could completely unroll the reduction
  - Luckily, the block size is limited by the GPU to 512 threads
  - Also, we are sticking to power-of-2 block sizes

- So we can easily unroll for a fixed block size
Reduction #6: Completely Unrolled

```c
#define groupDim_x ...
if (groupDim_x >= 512) {
    if (tid < 256) { sdata[tid] += sdata[tid + 256]; } GroupMemoryBarrierWithGroupSync();
}
if (groupDim_x >= 256) {
    if (tid < 128) { sdata[tid] += sdata[tid + 128]; } GroupMemoryBarrierWithGroupSync();
}
if (groupDim_x >= 128) {
    if (tid <  64) { sdata[tid] += sdata[tid +  64]; } GroupMemoryBarrierWithGroupSync();
}
if (tid < 32) {
    if (groupDim_x >=  64) sdata[tid] += sdata[tid + 32];
    if (groupDim_x >=  32) sdata[tid] += sdata[tid + 16];
    if (groupDim_x >=  16) sdata[tid] += sdata[tid +  8];
    if (groupDim_x >=   8) sdata[tid] += sdata[tid +   4];
    if (groupDim_x >=   4) sdata[tid] += sdata[tid +   2];
    if (groupDim_x >=   2) sdata[tid] += sdata[tid +   1];
}
```
## Performance for 4M element reduction

<table>
<thead>
<tr>
<th>Shader</th>
<th>Instruction Details</th>
<th>Time ($2^{22}$ ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shader 1:</td>
<td>interleaved addressing with divergent branching</td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shader 2:</td>
<td>interleaved addressing with bank conflicts</td>
<td>3.456 ms</td>
<td>4.854 GB/s</td>
<td>2.33x</td>
<td>2.33x</td>
</tr>
<tr>
<td>Shader 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>Shader 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>Shader 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>Shader 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>
</tbody>
</table>
Reduction #7: Multiple Adds / Thread

Replace load and add of two elements:

```c
unsigned int tid = threadIdx.x;
unsigned int i = groupIdx.x*(groupDim_x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+groupDim_x];
GroupMemoryBarrierWithGroupSync();
```

With a while loop to add as many as necessary:

```c
unsigned int tid = threadIdx.x;
unsigned int i = groupIdx.x*(groupDim*2) + threadIdx.x;
unsigned int dispatchSize = groupDim *2*gridDim.x;
sdata[tid] = 0;

while (i < n) {
    sdata[tid] += g_idata[i] + g_idata[i+groupDim_x];
    i += dispatchSize;
}
GroupMemoryBarrierWithGroupSync();
```
Reduction #7: Multiple Adds / Thread

Replace load and add of two elements:

```c
unsigned int tid = threadIdx.x;
unsigned int i = groupIdx.x*(groupDim_x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+groupDim_x];
GroupMemoryBarrierWithGroupSync();
```

With a while loop to add as many as necessary:

```c
unsigned int tid = threadIdx.x;
unsigned int i = groupIdx.x*(groupDim_x*2) + threadIdx.x;
unsigned int dispatchSize = groupDim_x*2*gridDim.x;
sdata[tid] = 0;

while (i < n) {
    sdata[tid] += g_idata[i] + g_idata[i+groupDim_x];
    i += dispatchSize;
}
GroupMemoryBarrierWithGroupSync();
```

Note: dispatchSize loop stride to maintain coalescing!
## Performance for 4M element reduction

<table>
<thead>
<tr>
<th>Shader</th>
<th>Time ($2^{22}$ ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2:</td>
<td>3.456 ms</td>
<td>4.854 GB/s</td>
<td>2.33x</td>
<td>2.33x</td>
</tr>
<tr>
<td>3:</td>
<td>1.722 ms</td>
<td>9.741 GB/s</td>
<td>2.01x</td>
<td>4.68x</td>
</tr>
<tr>
<td>4:</td>
<td>0.965 ms</td>
<td>17.377 GB/s</td>
<td>1.78x</td>
<td>8.34x</td>
</tr>
<tr>
<td>5:</td>
<td>0.536 ms</td>
<td>31.289 GB/s</td>
<td>1.8x</td>
<td>15.01x</td>
</tr>
<tr>
<td>6:</td>
<td>0.381 ms</td>
<td>43.996 GB/s</td>
<td>1.41x</td>
<td>21.16x</td>
</tr>
<tr>
<td>7:</td>
<td>0.268 ms</td>
<td>62.671 GB/s</td>
<td>1.42x</td>
<td>30.04x</td>
</tr>
</tbody>
</table>

Shader 7 on 32M elements: 73 GB/s!
Final Optimized Compute Shader

```c
Cbuffers consts {
    uint n;
    uint dispatchDim_x;
};
groupsshared float sdata[groupDim_x];
[numthreads( groupDim_x, 1, 1)]
void reduce6(uint tid : SV_GroupIndex,
    uint3 groupIdx : groupID )
{
    unsigned int i = groupIdx.x*(groupDim_x*2) + tid;
    unsigned int dispatchSize = groupDim_x*2*dispatchDim_x;
sdata[tid] = 0;
    do {  sdata[tid] += g_idata[i] + g_idata[i+groupDim_x];  i += dispatchSize; } while (i < n);
    GroupMemoryBarrierWithGroupSync();
    if (groupDim_x >= 512) {  if (tid < 256) { sdata[tid] += sdata[tid + 256]; } GroupMemoryBarrierWithGroupSync(); }
    if (groupDim_x >= 256) {  if (tid < 128) { sdata[tid] += sdata[tid + 128]; } GroupMemoryBarrierWithGroupSync(); }
    if (groupDim_x >= 128) {  if (tid < 64) { sdata[tid] += sdata[tid + 64]; } GroupMemoryBarrierWithGroupSync(); }
    if (tid < 32) {
        if (groupDim_x >= 64) sdata[tid] += sdata[tid + 32];
        if (groupDim_x >= 32) sdata[tid] += sdata[tid + 16];
        if (groupDim_x >= 16) sdata[tid] += sdata[tid + 8];
        if (groupDim_x >= 8) sdata[tid] += sdata[tid + 4];
        if (groupDim_x >= 4) sdata[tid] += sdata[tid + 2];
        if (groupDim_x >= 2) sdata[tid] += sdata[tid + 1];
    }
    if (tid == 0) g_odata[groupIdx.x] = sdata[0];
}
```
Performance Comparison

- 1: Interleaved Addressing: Divergent Branches
- 2: Interleaved Addressing: Bank Conflicts
- 3: Sequential Addressing
- 4: First add during global load
- 5: Unroll last warp
- 6: Completely unroll
- 7: Multiple elements per thread (max 64 blocks)

Graph showing time (ms) vs. number of elements for different methods.
Questions?

- Eric Young
  - eyoung@nvidia.com
Extra Slides
Multi-GPU

- Multiple GPUs can be used in a single system for task or data parallel GPU processing
- Host explicitly manages I/O and workload for each GPU
- Choose the best split to minimize inter-GPU communication (must occur via host memory)
Memory Coalescing (Matrix Multiply)

Each iteration, threads access the same element in A. Un-coalesced in CC $\leq 1.1$. 
## Matrix Multiplication (cont.)

<table>
<thead>
<tr>
<th>Optimization</th>
<th>GeForce GTX 280</th>
<th>GeForce GTX 8800</th>
</tr>
</thead>
<tbody>
<tr>
<td>No optimization</td>
<td>8.8 GBps</td>
<td>0.7 GBps</td>
</tr>
<tr>
<td>Coalesced using local memory to store a tile of A</td>
<td>14.3 GBps</td>
<td>8.2 GBps</td>
</tr>
<tr>
<td>Using thread group shared memory to eliminate redundant reads of a tile of B</td>
<td>29.7 GBps</td>
<td>15.7 GBps</td>
</tr>
</tbody>
</table>
## Matrix Multiplication (cont.)

<table>
<thead>
<tr>
<th>Optimization</th>
<th>GeForce GTX 280</th>
<th>GeForce GTX 8800</th>
</tr>
</thead>
<tbody>
<tr>
<td>No optimization</td>
<td>8.8 GBps</td>
<td>0.7 GBps</td>
</tr>
<tr>
<td>Coalesced using local memory to store a tile of A</td>
<td>14.3 GBps</td>
<td>8.2 GBps</td>
</tr>
<tr>
<td>Using thread group shared memory to eliminate redundant reads of a tile of B</td>
<td>29.7 GBps</td>
<td>15.7 GBps</td>
</tr>
</tbody>
</table>