CUDA is fast and efficient

CUDA enables efficient use of the massive parallelism of NVIDIA GPUs
- Direct execution of data-parallel programs
- Without the overhead of a graphics API

Using CUDA on Tesla GPUs can provide large speedups on data-parallel computations straight out of the box!

Even higher speedups are achievable by understanding and tuning for GPU architecture
- This presentation covers general performance, common pitfalls, and useful strategies
Outline

General optimization guidance
- Coalescing memory operations
- Latency Hiding
- Shared Memory Bank Conflicts

Example 1: transpose
- Coalescing and bank conflict avoidance

Example 2: efficient parallel reductions
- Using peak performance metrics to guide optimization
- Avoiding SIMD divergence & bank conflicts
- Loop unrolling
- Using template parameters to write general-yet-optimized code

Algorithmic strategy: Cost efficiency
Quick terminology review

**Thread:** concurrent code and associated state executed on the CUDA device (in parallel with other threads)
- The unit of parallelism in CUDA
- Note difference from CPU threads: creation cost, resource usage, and switching cost of GPU threads is much smaller

**Warp:** a group of threads executed *physically* in parallel (SIMD)
- **Half-warp:** the first or second half of a warp of threads

**Thread Block:** a group of threads that are executed together and can share memory on a single multiprocessor

**Grid:** a group of thread blocks that execute a single CUDA program *logically* in parallel

**Device:** GPU  
**Host:** CPU  
**SM:** Multiprocessor
CUDA Optimization Strategies

- Optimize Algorithms for the GPU
- Optimize Memory Access Coherence
- Take Advantage of On-Chip Shared Memory
- Use Parallelism Efficiently
Optimize Algorithms for the GPU

- Maximize independent parallelism
- Maximize arithmetic intensity (math/bandwidth)
- Sometimes it’s better to recompute than to cache
  - GPU spends its transistors on ALUs, not memory
- Do more computation on the GPU to avoid costly data transfers
  - Even low parallelism computations can sometimes be faster than transferring back and forth to host
Optimize Memory Coherence

- Coalesced vs. Non-coalesced = order of magnitude
  - Global/Local device memory

- Optimize for spatial locality in cached texture memory

- In shared memory, avoid high-degree bank conflicts
Take Advantage of Shared Memory

- Hundreds of times faster than global memory
- Threads can cooperate via shared memory

Use one / a few threads to load / compute data shared by all threads

- Use it to avoid non-coalesced access
  - Stage loads and stores in shared memory to re-order non-coalesceable addressing
  - Matrix transpose example later
Use Parallelism Efficiently

Partition your computation to keep the GPU multiprocessors equally busy
  - Many threads, many thread blocks

Keep resource usage low enough to support multiple active thread blocks per multiprocessor
  - Registers, shared memory
Global Memory Reads/Writes

- Highest latency instructions: 400-600 clock cycles
- Likely to be performance bottleneck
- Optimizations can greatly increase performance
  - Coalescing: up to 10x speedup
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 on G8X architecture:
- 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 being read

Exception: not all threads must be participating
- Predicated access, divergence within a halfwarp
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:
Timing Results

Experiment:
- Kernel: read a float, increment, write back
- 3M floats (12MB)
- Times averaged over 10K runs

12K blocks x 256 threads:
- 356µs – coalesced
- 357µs – coalesced, some threads don’t participate
- 3,494µs – permuted/misaligned thread access
```c
__global__ void accessFloat3(float3 *d_in, float3 d_out)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    float3 a = d_in[index];

    a.x += 2;
    a.y += 2;
    a.z += 2;

    d_out[index] = a;
}
```
Uncoalesced Access: float3 Case

- float3 is 12 bytes
- Each thread ends up executing 3 reads
  - sizeof(float3) ≠ 4, 8, or 12
  - Half-warp reads three 64B non-contiguous regions

First read
Coalescing float3 Access

Similarly, Step3 starting at offset 512
Coalesced Access: float3 Case

Use shared memory to allow coalescing

- Need `sizeof(float3)*(threads/block)` bytes of SMEM
- Each thread reads 3 scalar floats:
  - Offsets: 0, (threads/block), 2*(threads/block)
  - These will likely be processed by other threads, so sync

Processing

- Each thread retrieves its float3 from SMEM array
  - Cast the SMEM pointer to (float3*)
  - Use thread ID as index
- Rest of the compute code does not change!
**Coalesced float3 Code**

```c
__global__ void accessInt3Shared(float *g_in, float *g_out)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    __shared__ float s_data[256*3];
    s_data[threadIdx.x] = g_in[index];
    s_data[threadIdx.x+256] = g_in[index+256];
    s_data[threadIdx.x+512] = g_in[index+512];
    __syncthreads();
    float3 a = (float3*)s_data)[threadIdx.x];
    a.x += 2;
    a.y += 2;
    a.z += 2;
    ((float3*)s_data)[threadIdx.x] = a;
    __syncthreads();
    g_out[index] = s_data[threadIdx.x];
    g_out[index+256] = s_data[threadIdx.x+256];
    g_out[index+512] = s_data[threadIdx.x+512];
}
```

- **Read the input through SMEM**
- **Compute code is not changed**
- **Write the result through SMEM**
Coalescing: Structures of Size $\neq 4, 8, \text{ or } 16$ Bytes

- Use a structure of arrays instead of AoS
- If SoA is not viable:
  - Force structure alignment: `__align(X)`, where $X = 4, 8, \text{ or } 16$
  - Use SMEM to achieve coalescing
Coalescing: Timing Results

Experiment:
- Kernel: read a float, increment, write back
- 3M floats (12MB)
- Times averaged over 10K runs

12K blocks x 256 threads:
- 356µs – coalesced
- 357µs – coalesced, some threads don’t participate
- 3,494µs – permuted/misaligned thread access

4K blocks x 256 threads:
- 3,302µs – float3 uncoalesced
- 359µs – float3 coalesced through shared memory
Coalescing: Coalescing
Summary

- Coalescing greatly improves throughput
- Critical to small or memory-bound kernels
- Reading structures of size other than 4, 8, or 16 bytes will break coalescing:
  - Prefer Structures of Arrays over AoS
  - If SoA is not viable, read/write through SMEM
- Futureproof code: coalesce over whole warps
- Additional resources:
  - Aligned Types CUDA SDK Sample
Data Transfers

- Device memory to host memory bandwidth much lower than device memory to device bandwidth
  - 4GB/s peak (PCI-e x16) vs. 80 GB/s peak (Quadro FX 5600)

- Minimize transfers
  - Intermediate data structures can be allocated, operated on, and deallocated without ever copying them to host memory

- Group transfers
  - One large transfer much better than many small ones
Page-Locked Memory Transfers

cudaMallocHost() allows allocation of page-locked host memory
Enables highest cudaMemcpy performance
  3.2 GB/s+ common on PCI-express x16
  ~4 GB/s measured on nForce 680i motherboards (overclocked PCI-e)

See the “bandwidthTest” CUDA SDK sample

Use with caution
  Allocating too much page-locked memory can reduce overall system performance
  Test your systems and apps to learn their limits
Thread instructions executed sequentially, executing other warps is the only way to hide latencies and keep the hardware busy.

**Occupancy** = Number of warps running concurrently on a multiprocessor divided by maximum number of warps that can run concurrently.

- **Minimize occupancy requirements** by minimizing latency.
- **Maximize occupancy** by optimizing threads per multiprocessor.
Minimize Occupancy Requirements

- Optimize global memory access:
  - 400-600 cycle latency

- Maximize arithmetic intensity (math/bandwidth)

- Follow all the global memory optimizations described before!
Register Pressure

- Solution to latency issues = more threads per SM
- Limiting Factors:
  - Number of registers per kernel
    - 8192 per SM, partitioned among concurrent threads
  - Amount of shared memory
    - 16KB per SM, partitioned among concurrent threadblocks
- Check .cubin file for # registers / kernel
- Use \texttt{--maxrregcount=N} flag to NVCC
  - \texttt{N} = desired maximum registers / kernel
  - At some point “spilling” into LMEM may occur
    - Reduces performance – LMEM is slow
    - Check .cubin file for LMEM usage
Determining resource usage

Compile the kernel code with the -cubin flag to determine register usage.

Open the .cubin file with a text editor and look for the “code” section.

architecture {sm_10}
abiversion {0}
modname {cubin}
code {
  name = BlackScholesGPU
  lmem = 0
  smem = 68
  reg = 20
  bar = 0
  bincode {
    0xa0004205 0x04200780 0x40024c09 0x00200780
    ...
  }

- per thread local memory
- per thread block shared memory
- per thread registers
CUDA GPU Occupancy Calculator

1. Select a GPU from the list (click here for help).
2. Enter your resource usage:
   - Threads Per Block: [value]
   - Registers Per Thread: [value]
   - Shared Memory Per Block (Bytes): [value]
3. The GPU Occupancy Data is displayed here and in the graphs:
   - Active Threads per Multiprocessor: [value]
   - Active Warps per Multiprocessor: [value]
   - Active Thread Blocks per Multiprocessor: [value]
   - Occupancy of each Multiprocessor: [value]
   - Maximum Simultaneous Blocks per GPU: [value]
4. Note: This assumes there are at least this many blocks.
5. Physical Limits for GPU:
   - Multiprocessors per GPU: [value]
   - Threads Per Warp: [value]
   - Multiprocessor: [value]
   - Threads Per Multiprocessor: [value]
   - Thread Blocks Per Multiprocessor: [value]
   - Total on-2D half registers per Multiprocessor: [value]
   - Memory / Multiprocessor (Bytes): [value]
6. Allocation Per Thread Block:
   - Warp: [value]
   - Register: [value]
   - Shared Memory: [value]
   - These data are used in computing the occupancy data in blue.
7. Maximum Thread Blocks Per Multiprocessor: [value]
8. Limited by Max Warps Per Multiprocessor: [value]
9. Limited by Register / Multiprocessor: [value]
10. Limited by Shared Memory / Multiprocessor: [value]
11. Thread Block Limit Per Multiprocessor is the minimum of these three.

For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Varying Block Size

Varying Register Count

Varying Shared Memory Usage
Grid/Block Size Heuristics

- # of blocks / # of multiprocessors > 1
  - So all multiprocessors have at least one block to execute
  - Per-block resources at most half of total available
    - Shared memory and registers
    - Multiple blocks can run concurrently in a multiprocessor
    - If multiple blocks coexist that aren’t all waiting at a
      __syncthreads(), machine can stay busy

- # of blocks / # of multiprocessors > 2
  - So multiple blocks run concurrently in a multiprocessor

- # of blocks > 100 to scale to future devices
  - Blocks stream through machine in pipeline fashion
  - 1000 blocks per grid will scale across multiple generations
Optimizing threads per block

- Choose threads per block as a multiple of warp size
  - Avoid wasting computation on under-populated warps
- More threads per block == better memory latency hiding
- But, more threads per block == fewer registers per thread
  - Kernel invocations can fail if too many registers are used

Heuristics
- Minimum: 64 threads per block
  - Only if multiple concurrent blocks
- 128 to 256 threads a better choice
  - Usually still enough regs to compile and invoke successfully
- This all depends on your computation!
  - Experiment!
Occupancy != Performance

- Increasing occupancy does not necessarily increase performance

  BUT...

- Low-occupancy multiprocessors cannot adequately hide latency on memory-bound kernels

  (It all comes down to arithmetic intensity and available parallelism)
Parameterize Your Application

Parameterization helps adaptation to different GPUs
GPUs vary in many ways
- # of multiprocessors
- Memory bandwidth
- Shared memory size
- Register file size
- Threads per block

You can even make apps self-tuning (like FFTW and ATLAS)
- “Experiment” mode discovers and saves optimal configuration
Register Pressure

Solution to latency issues = more threads per SM

Limiting Factors:
- Number of registers per kernel
  - 8192 per SM, partitioned among concurrent threads
- Amount of shared memory
  - 16KB per SM, partitioned among concurrent threadblocks

Check .cubin file for # registers / kernel

Use –maxrregcount=N flag to NVCC
- N = desired maximum registers / kernel

At some point “spilling” into LMEM may occur
- Reduces performance – LMEM is slow
- Check .cubin file for LMEM usage
Latency Hiding: Global Memory

- Global memory access: **400-600 cycle latency**
  - Blocks dependent instructions in the same thread

Remedy:
- More threads!
- Instructions in other threads are not blocked
- Maximize occupancy

Same idea as pipelining:
- 4 sequential reads take at least $4 \times 400 = 1,600$ cycles

4 threads, one read each, take: $400 + 1 + 1 + 1 = 403$ cycles
Latency Hiding: Register Dependency

Read-after-write register dependency
- Instruction’s result can be read 11 cycles later
- Scenarios:
  - CUDA:
    - $x = y + 5$
    - $z = x + 3$
  - PTX:
    - `add.f32 $f3, $f1, $f2`
    - `add.f32 $f5, $f3, $f4`
    - `ld.shared.f32 $f3, [$r31+0]`
    - `add.f32 $f3, $f3, $f4`

To completely hide the latency:
- Run at least 192 threads (6 warps) per multiprocessor
- At least 25% occupancy
- Threads do not have to belong to the same thread block
Latency Hiding: Synchronization

- Thread synchronization (__syncthreads)
- More threads per block = higher latency
  - Waiting on threads in other warps to reach the sync point
- Smaller thread blocks will reduce latency
- BUT: usually not really a problem
In a parallel machine, many threads access memory. Therefore, memory is divided into banks. Essential to achieve high bandwidth.

Each bank can service one address per cycle. A memory can service as many simultaneous accesses as it has banks.

Multiple simultaneous accesses to a bank result in a bank conflict. Conflicting accesses are serialized.
Bank Addressing Examples

No Bank Conflicts
- Linear addressing
  stride == 1

- Random 1:1 Permutation

No Bank Conflicts
Bank Addressing Examples

2-way Bank Conflicts
  Linear addressing
  stride == 2

8-way Bank Conflicts
  Linear addressing
  stride == 8
How addresses map to banks on G80

- Bandwidth of each bank is 32 bits per 2 clock cycles
- Successive 32-bit words are assigned to successive banks
- G80 has 16 banks
  - So \( \text{bank} = \text{address} \mod 16 \)
  - Same as the size of a half-warp
    - No bank conflicts between different half-warps, only within a single half-warp
Shared memory bank conflicts

Shared memory is as fast as registers if there are no bank conflicts.

The fast case:
- If all threads of a half-warp access different banks, there is no bank conflict.
- If all threads of a half-warp read the identical address, there is no bank conflict (broadcast).

The slow case:
- Bank Conflict: multiple threads in the same half-warp access the same bank.
- Must serialize the accesses.
- Cost = max # of simultaneous accesses to a single bank.
Optimization Example 1:
Matrix Transpose
Matrix Transpose

SDK Sample ("transpose")

Illustrates:

- Coalescing
- Avoiding SMEM bank conflicts
- Speedups for even small matrices

```
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16
```

```
1 5 9 13
2 6 10 14
3 7 11 15
4 8 12 16
```
Uncoalesced Transpose

```c
__global__ void transpose_naive(float *odata, float *idata, int width, int height) {
    unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;
    if (xIndex < width && yIndex < height) {
        unsigned int index_in  = xIndex + width * yIndex;
        unsigned int index_out = yIndex + height * xIndex;
        odata[index_out] = idata[index_in];
    }
}
```
Uncoalesced Transpose

Reads input from GMEM

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>0,1</td>
<td>0,2</td>
<td>0,15</td>
</tr>
<tr>
<td>1,0</td>
<td>1,1</td>
<td>1,2</td>
<td>1,15</td>
</tr>
<tr>
<td>15,0</td>
<td>15,1</td>
<td>15,2</td>
<td>15,15</td>
</tr>
</tbody>
</table>

Write output to GMEM

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>1,0</td>
<td>2,0</td>
<td>15,0</td>
</tr>
<tr>
<td>0,1</td>
<td>1,1</td>
<td>2,1</td>
<td>15,1</td>
</tr>
<tr>
<td>0,15</td>
<td>1,15</td>
<td>2,15</td>
<td>15,15</td>
</tr>
</tbody>
</table>

Stride = 16, uncoalesced

Stride = 1, coalesced
Coalesced Transpose

Assumption: matrix is partitioned into square tiles

Threadblock \((bx, by)\):
- Read the \((bx,by)\) input tile, store into SMEM
- Write the SMEM data to \((by,bx)\) output tile
  - Transpose the indexing into SMEM

Thread \((tx,ty)\):
- Reads element \((tx,ty)\) from input tile
- Writes element \((tx,ty)\) into output tile

Coalescing is achieved if:
- Block/tile dimensions are multiples of 16
Coalesced Transpose

Reads from GMEM

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>0,1</td>
<td>0,2</td>
</tr>
<tr>
<td>1,0</td>
<td>1,1</td>
<td>1,2</td>
</tr>
<tr>
<td>15,0</td>
<td>15,1</td>
<td>15,2</td>
</tr>
</tbody>
</table>

Write to SMEM

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>0,1</td>
<td>0,2</td>
</tr>
<tr>
<td>1,0</td>
<td>1,1</td>
<td>1,2</td>
</tr>
<tr>
<td>15,0</td>
<td>15,1</td>
<td>15,2</td>
</tr>
</tbody>
</table>

Reads from SMEM

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>1,0</td>
<td>2,0</td>
</tr>
<tr>
<td>0,1</td>
<td>1,1</td>
<td>2,1</td>
</tr>
<tr>
<td>0,15</td>
<td>1,15</td>
<td>2,15</td>
</tr>
</tbody>
</table>

Write to GMEM

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>0,1</td>
<td>0,2</td>
</tr>
<tr>
<td>1,0</td>
<td>1,1</td>
<td>1,2</td>
</tr>
<tr>
<td>15,0</td>
<td>15,1</td>
<td>15,2</td>
</tr>
</tbody>
</table>

---

NVIDIA
SMEM Optimization

Reads from SMEM

 Threads read SMEM with stride = 16
  - Bank conflicts

Solution
  - Allocate an “extra” column
  - Read stride = 17
  - Threads read from consecutive banks
__global__ void transpose(float *odata, float *idata, int width, int height)
{
1. __shared__ float block[(BLOCK_DIM+1)*BLOCK_DIM];

2. unsigned int xBlock = __mul24(blockDim.x, blockIdx.x);
3. unsigned int yBlock = __mul24(blockDim.y, blockIdx.y);
4. unsigned int xIndex = xBlock + threadIdx.x;
5. unsigned int yIndex = yBlock + threadIdx.y;
6. unsigned int index_out, index_transpose;

7. if (xIndex < width && yIndex < height)
    {
8.    unsigned int index_in = __mul24(width, yIndex) + xIndex;
9.    unsigned int index_block = __mul24(threadIdx.y, BLOCK_DIM+1) + threadIdx.x;
10.   block[index_block] = idata[index_in];
11.   index_transpose = __mul24(threadIdx.x, BLOCK_DIM+1) + threadIdx.y;
12.   index_out = __mul24(height, xBlock + threadIdx.y) + yBlock + threadIdx.x;
    }
13. __syncthreads();

14. if (xIndex < width && yIndex < height)
15.    odata[index_out] = block[index_transpose];
}
### Transpose Timings

**Speedups with coalescing and SMEM optimization:**
- 128x128: 0.011ms vs. 0.022ms (2.0X speedup)
- 512x512: 0.07ms vs. 0.33ms (4.5X speedup)
- 1024x1024: 0.30ms vs. 1.92ms (6.4X speedup)
- 1024x2048: 0.79ms vs. 6.6ms (8.4X speedup)

**Coalescing without SMEM optimization:**
- 128x128: 0.014ms
- 512x512: 0.101ms
- 1024x1024: 0.412ms
- 1024x2048: 0.869ms
Optimization Example 2: Parallel Reduction
Parallel Reduction

- Common and important data parallel primitive
- Easy to implement in CUDA
  - 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

```
  3 1 7 0 4 1 6 3
   4 7 5 9
     11 14
       25
```

- 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 blocks, could easily reduce very large arrays, right?
- Global sync after each block produces its result
- Once all blocks reach sync, continue recursively

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

Solution: decompose into multiple kernels
- Kernel launch serves as a global synchronization point
- Kernel launch has negligible HW overhead, low SW overhead
Solution: Kernel Decomposition

Avoid global sync by decomposing computation into multiple kernel invocations

In the case of reductions, code for all levels is the same

Recursive kernel invocation

Level 0: 8 blocks
Level 1: 1 block
What is Our Optimization Goal?

- We should strive to reach GPU peak performance
- Choose the right metric:
  - GFLOP/s: for compute-bound kernels
  - Bandwidth: for memory-bound kernels
- Reductions have very low arithmetic intensity
  - 1 flop per element loaded (bandwidth-optimal)
- Therefore we should strive for peak bandwidth

Will use G80 GPU for this example
- 384-bit memory interface, 900 MHz DDR
- \(384 \times 1800 / 8 = 86.4 \text{ GB/s}\)
Reduction #1: Interleaved Addressing

```c
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

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

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

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

Values (shared memory):

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

Step 1
Stride 1

Values:

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

Step 2
Stride 2

Values:

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

Step 3
Stride 4

Values:

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

Step 4
Stride 8

Values:

<table>
<thead>
<tr>
<th>Values</th>
<th>Thread IDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>41</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
</tr>
<tr>
<td>-3</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
__global__ void reduce1(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

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

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

    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
### Performance for 4M element reduction

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

Note: Block Size = 128 threads for all tests
Reduction #2: Interleaved Addressing

Just replace divergent branch in inner loop:

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

With strided index and non-divergent branch:

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

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

<table>
<thead>
<tr>
<th>Kernel 1: interleaved addressing with divergent branching</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>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Kernel 2: interleaved addressing with bank conflicts     | 3.456 ms            | 4.854 GB/s| 2.33x        | 2.33x              |
Parallel Reduction: Sequential Addressing

 Sequential addressing is conflict free
Reduction #3: Sequential Addressing

Just replace strided indexing in inner loop:

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

With reversed loop and threadID-based indexing:

```cpp
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```
## Performance for 4M element reduction

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Time (2⁰² ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel 1: interleaved</td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addressing with divergent</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>branching</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 2: interleaved</td>
<td>3.456 ms</td>
<td>4.854 GB/s</td>
<td>2.33x</td>
<td>2.33x</td>
</tr>
<tr>
<td>addressing with bank</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>conflicts</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 3: sequential</td>
<td>1.722 ms</td>
<td>9.741 GB/s</td>
<td>2.01x</td>
<td>4.68x</td>
</tr>
<tr>
<td>addressing</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Idle Threads

Problem:

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

Half of the threads are idle on first loop iteration!

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

Halve the number of blocks, and replace single load:

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

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 = 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>Description</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 addressing with divergent branching</td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>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>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>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>
</tbody>
</table>
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 \texttt{__syncthreads()}\n  - 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
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];
}

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>Kernel</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>Kernel 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>Kernel 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>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>
</tbody>
</table>
Complete Unrolling

If we knew 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.

- But we need to be generic – how can we unroll for block sizes that we don’t know at compile time?

Templates to the rescue!

- CUDA supports C++ template parameters on device and host functions.
Unrolling with Templates

Specify block size as a function template parameter:

```cpp
template <unsigned int blockSize>
__global__ void reduce5(int *g_idata, int *g_odata)
```
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.

Results in a very efficient inner loop!
Invoking Template Kernels

Don’t we still need block size at compile time?

Nope, just a switch statement for 10 possible block sizes:

```c
switch (threads)
{
    case 512:
        reduce5<512><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 256:
        reduce5<256><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 128:
        reduce5<128><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 64:
        reduce5<64><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 32:
        reduce5<32><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 16:
        reduce5<16><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 8:
        reduce5<8><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 4:
        reduce5<4><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 2:
        reduce5<2><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 1:
        reduce5<1><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
}
```
## Performance for 4M element reduction

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Access Pattern</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 with divergent branching</td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 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>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>
</tbody>
</table>
Parallel Reduction Complexity

- **Log($N$)** parallel steps, each step $S$ does $N/2^S$ independent ops
  - Step Complexity is $O(\log N)$

- For $N=2^D$, performs $\sum_{S \in [1..D]} 2^{D-S} = N-1$ operations
  - Work Complexity is $O(N)$ – It is work-efficient
    - i.e. does not perform more operations than a sequential algorithm

- With $P$ threads physically in parallel ($P$ processors), time complexity is $O(N/P + \log N)$
  - Compare to $O(N)$ for sequential reduction
  - In a thread block, $N=P$, so $O(\log N)$
What About Cost?

Cost of a parallel algorithm is processors × time complexity
- Allocate threads instead of processors: $O(N)$ threads
- Within a block, time complexity is $O(\log N)$, so cost is $O(N \log N)$: not cost efficient!

Brent’s theorem suggests $O(N/\log N)$ threads
- Each thread does $O(\log N)$ sequential work
- Then all $O(N/\log N)$ threads cooperate for $O(\log N)$ steps
- Cost = $O((N/\log N) \times \log N) = O(N)$

Sometimes called algorithm cascading
- Can lead to significant speedups in practice
Algorithm Cascading

- Combine sequential and parallel reduction
  - Each thread loads and sums multiple elements into shared memory
  - Tree-based reduction in shared memory
- Brent’s theorem says each thread should sum O(log n) elements
  - i.e. 1024 or 2048 elements per block vs. 256
- In my experience, beneficial to push it even further
  - Possibly better latency hiding with more work per thread
  - More threads per block reduces levels in tree of recursive kernel invocations
  - High kernel launch overhead in last levels with few blocks
- On G80, best perf with 64-256 blocks of 128 threads
  - 1024-4096 elements per thread
Replace load and add of two elements:

```c
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

With a while loop to add as many as necessary:

```c
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
unsigned int gridSize = blockSize*2*gridDim.x;
sdata[tid] = 0;

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

Replace load and add of two elements:

```c
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

With a while loop to add as many as necessary:

```c
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockSize*2 + threadIdx.x;
unsigned int gridSize = blockSize*2*gridDim.x;
sdata[tid] = 0;
while (i < n) {
    sdata[tid] += g_idata[i] + g_idata[i+blockSize];
    i += gridSize;
}
__syncthreads();
```

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

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Description</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 with divergent branching</td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td>1.41x</td>
<td>1.42x</td>
</tr>
<tr>
<td>Kernel 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>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: 72 GB/s!
template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n)
{
    extern __shared__ int sdata[];

    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockSize*2) + tid;
    unsigned int gridSize = blockSize*2*gridDim.x;
    sdata[tid] = 0;

    do { sdata[tid] += g_idata[i] + g_idata[i+blockSize]; i += gridSize; } while (i < n);
__syncthreads();

    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];
    }

    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
Performance Comparison

Graph showing the relationship between the number of elements and time (ms) for different addressing and optimization techniques:

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)
Types of optimization

Interesting observation:

Algorithmic optimizations
  Changes to addressing, algorithm cascading
  11.84x speedup, combined!

Code optimizations
  Loop unrolling
  2.54x speedup, combined
Conclusion

- Understand CUDA performance characteristics
  - Memory coalescing
  - Divergent branching
  - Bank conflicts
  - Occupancy and Latency hiding
- Use peak performance metrics to guide optimization
- Understand parallel algorithm complexity theory
- 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