Optimizing Parallel Reduction in CUDA

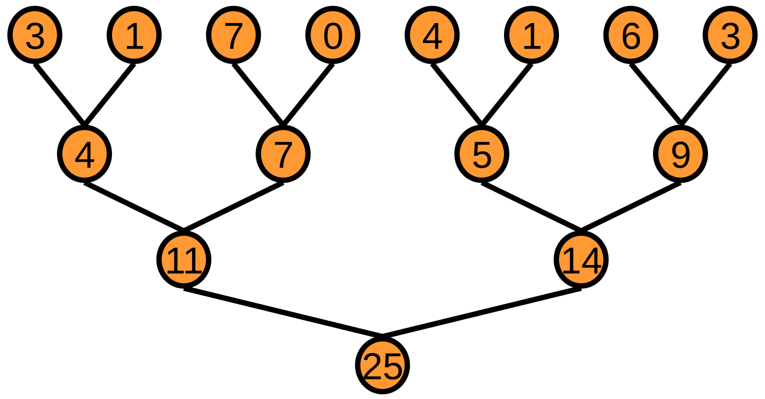
Mark Harris
NVIDIA Developer Technology

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**



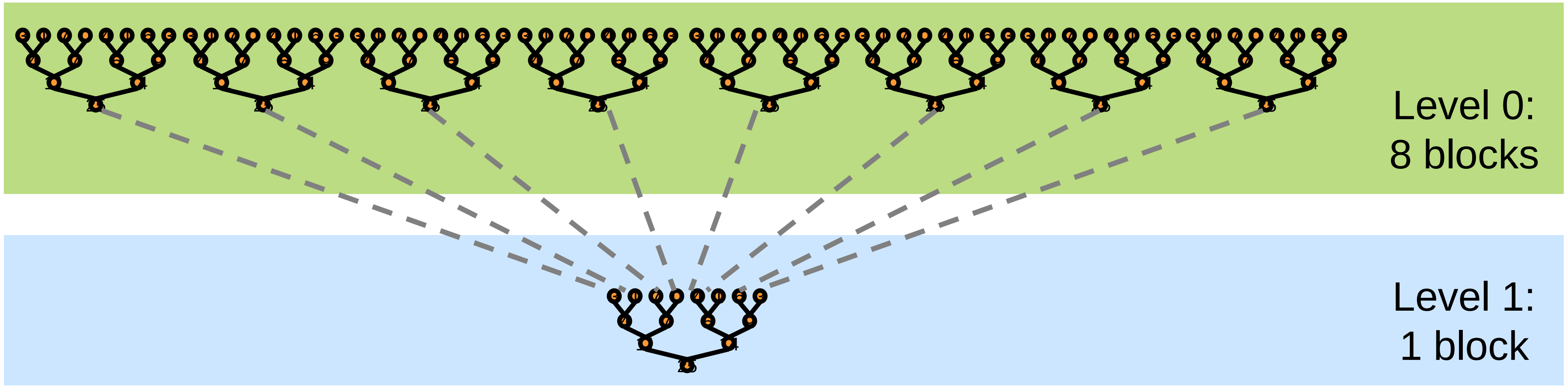
- **Need to be able to use multiple thread blocks**
  - **To process very large arrays**
  - **To keep all multiprocessors on the GPU busy**
  - **Each thread block reduces a portion of the array**
- **But how do we communicate partial results between thread blocks?**

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**

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 * 1800 / 8 = 86.4 GB/s

Reduction #1: Interleaved Addressing

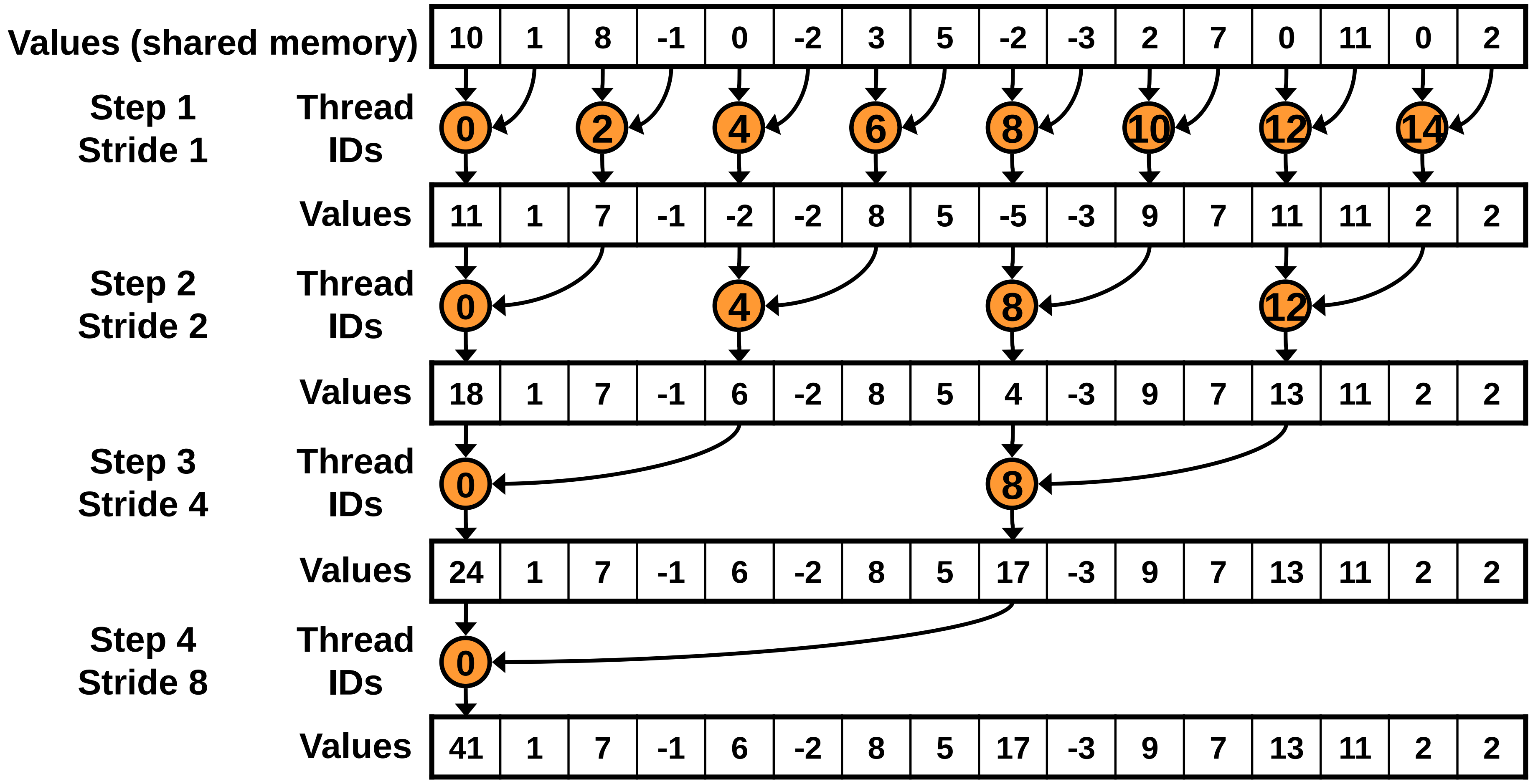
```
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // 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



Reduction #1: Interleaved Addressing

```
__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();
    }

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

Problem: highly divergent warps are very inefficient, and % operator is very slow

Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s

Note: Block Size = 128 threads for all tests

Reduction #2: Interleaved Addressing

Just replace divergent branch in inner loop:

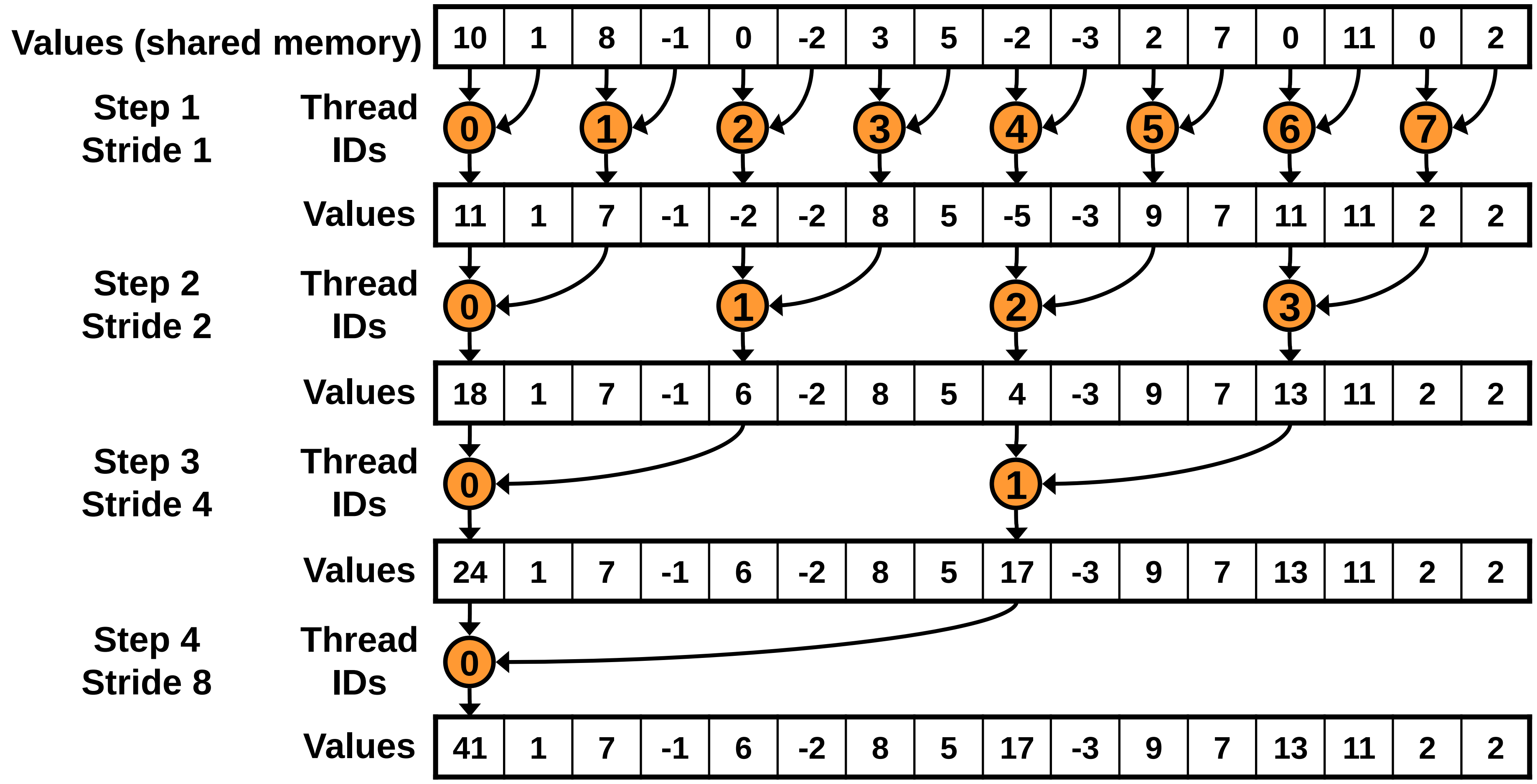
```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    if (tid % (2*s) == 0) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

With strided index and non-divergent branch:

```
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();
}
```

Parallel Reduction: Interleaved Addressing

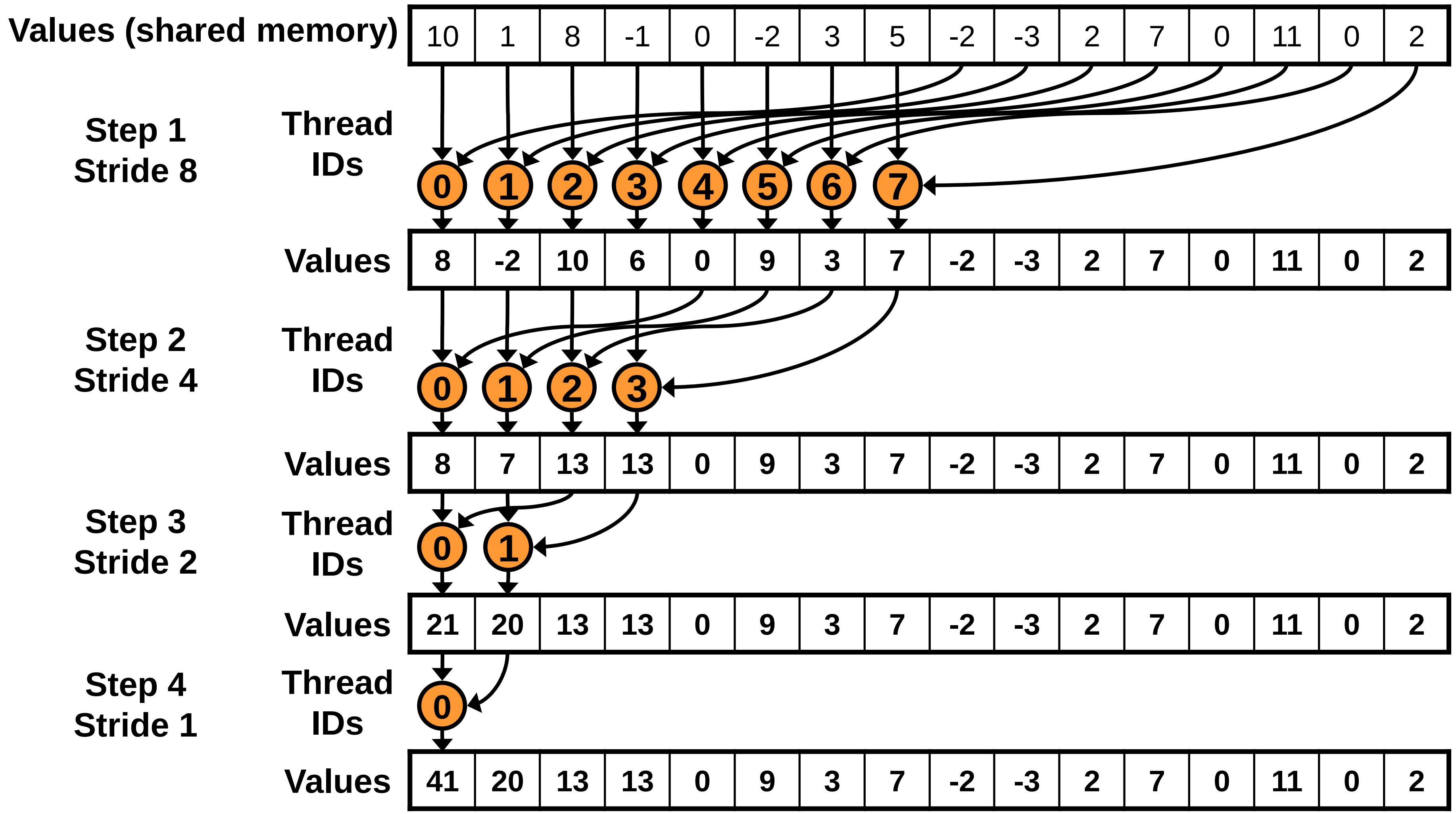


New Problem: Shared Memory Bank Conflicts

Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
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:

```
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:

```
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

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x

Idle Threads

Problem:

```
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:

```
// 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:

```
// 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

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x

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 <= 32, we have only one warp left
- Instructions are SIMD synchronous within a warp
- That means when s <= 32:
  - We don’t need to __syncthreads()
  - 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

```
__device__ void warpReduce(volatile int* sdata, int tid) {
        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];
  }
```

IMPORTANT:
For this to be correct,
we must use the
"volatile" keyword!

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

  if (tid < 32) warpReduce(sdata, tid);
```

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

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x

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:

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

Reduction #6: Completely Unrolled

```
Template <unsigned int blockSize>
__device__ void warpReduce(volatile int* sdata, int tid) {
      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 (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) warpReduce<blockSize>(sdata, tid);
```

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:**

```
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

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x
Kernel 6: completely unrolled	0.381 ms	43.996 GB/s	1.41x	21.16x

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
  - 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*) * log *N*) = O(*N*) → cost efficient
- 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*

Reduction #7: Multiple Adds / Thread

Replace load and add of two elements:

```
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:

```
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();
```

Reduction #7: Multiple Adds / Thread

Replace load and add of two elements:

```
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:

```
unsigned int tid = th
unsigned int i = blo
unsigned int gridSi
sdata[tid] = 0;

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

Note: gridSize loop stride to maintain coalescing!

Performance for 4M element reduction

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

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

Final Optimized Kernel

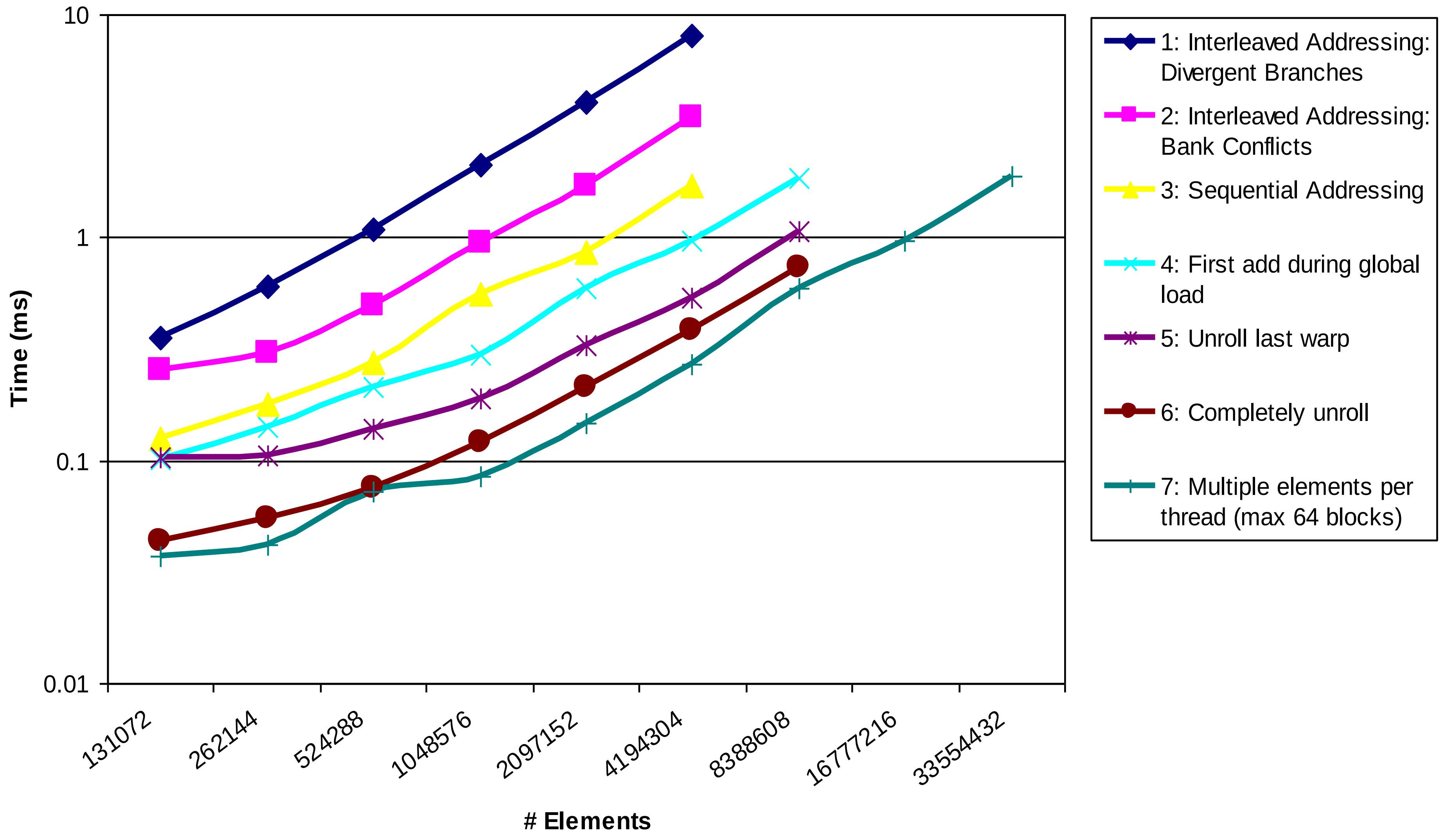
```
template <unsigned int blockSize>
__device__ void warpReduce(volatile int *sdata, unsigned int tid) {
    if (blockSize >=  64) sdata[tid] += sdata[tid + 32];
    if (blockSize >=  32) sdata[tid] += sdata[tid + 16];
    if (blockSize >=  16) sdata[tid] += sdata[tid +  8];
    if (blockSize >=    8) sdata[tid] += sdata[tid +  4];
    if (blockSize >=    4) sdata[tid] += sdata[tid +  2];
    if (blockSize >=    2) sdata[tid] += sdata[tid +  1];
}
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;

    while (i < n) { sdata[tid] += g_idata[i] + g_idata[i+blockSize];  i += gridSize;  }
    __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) warpReduce(sdata, tid);
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Performance Comparison



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
  - 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

- Questions: mharris@nvidia.com