Optimizing Parallel Reduction in CUDA

Mark Harris
NVDIA Developer Technology

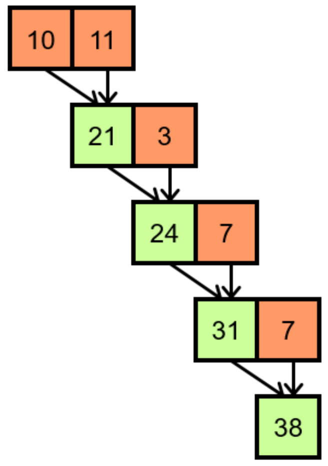
2016. 04. 14
Ju Youn, Park

Contents

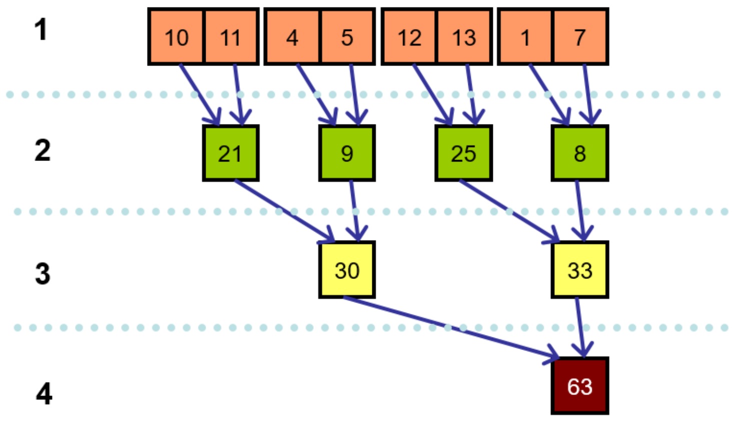
- Parallel Reduction
  - Global Synchronization
  - Kernel Decomposition
- Optimization
- Reduction #1 to #7
- Performance Comparison
- Conclusion

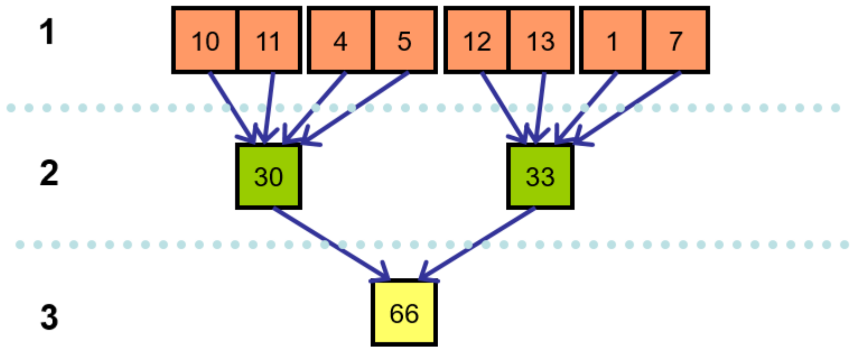
Parallel Reduction

- Reduction into primitive operations
- **Sequential Reduction**
  - O(N)



- **Parallel Reduction** (Tree-Like approach)
  - O(logN) – depth of tree

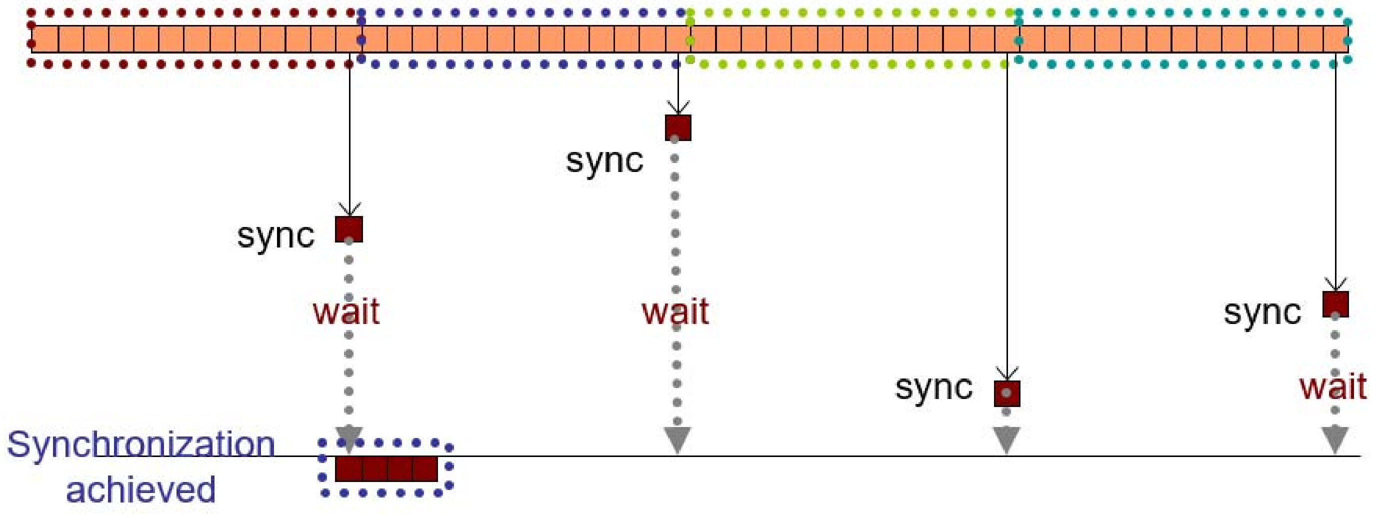




Global Synchronization

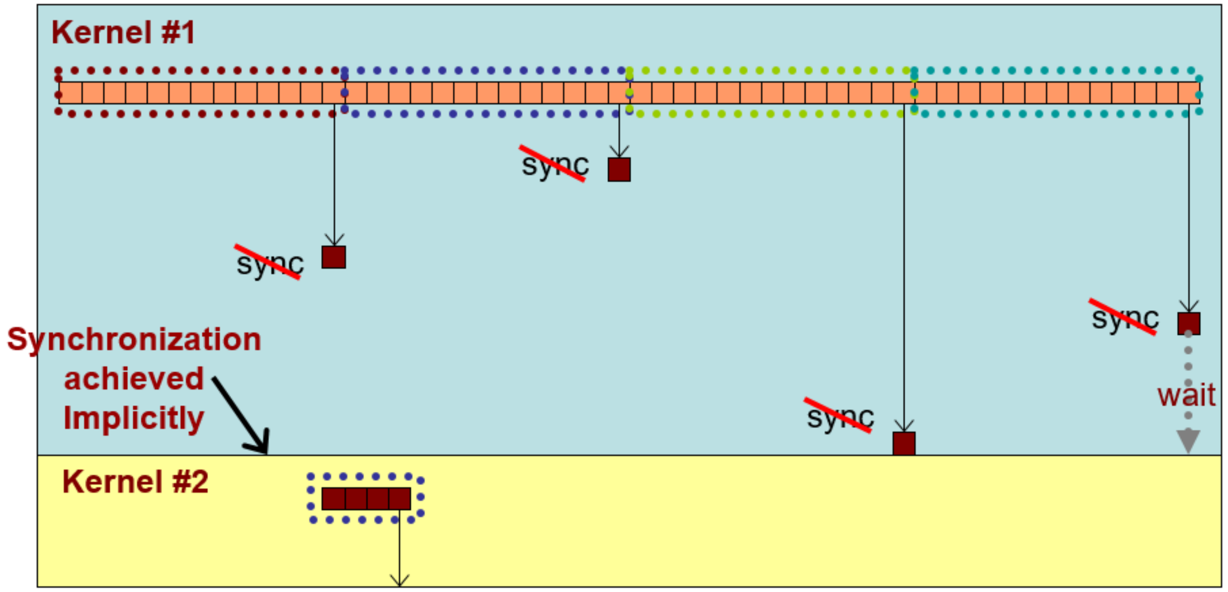
- Single Block
  - Tree-Like approach

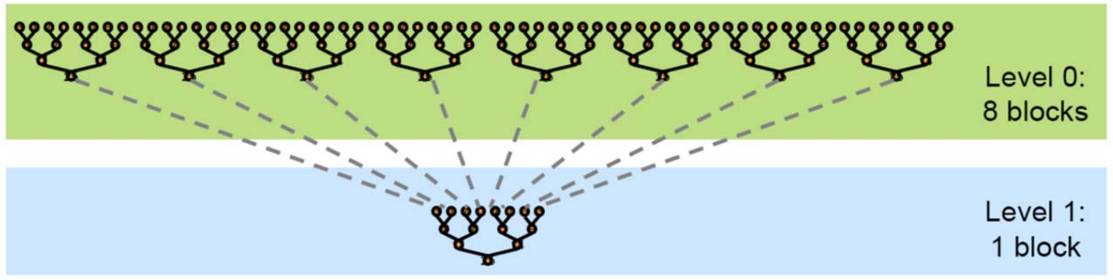
- Multiple Blocks
  - Synchronization should be considered – to communicate partial results
  - But, CUDA does not support **Global Synchronization**
    - Expensive to support
    - Would limit the number of blocks to avoid deadlock
  - ▶ Solution: **Decompose into multiple kernels**



Kernel Decomposition

- **Kernel** launch serves as a global synchronization point
  - Same kernel code can be called multiple times
  - Recursive kernel invocation





Optimization

- Metrics for GPU performance
  - **GFLOP/s** for compute-bound kernels
    - One billion floating-point operations per second
  - **Bandwidth** for memory-bound kernels
    - The rate at which data can be read from or stored into a semiconductor memory by a processor

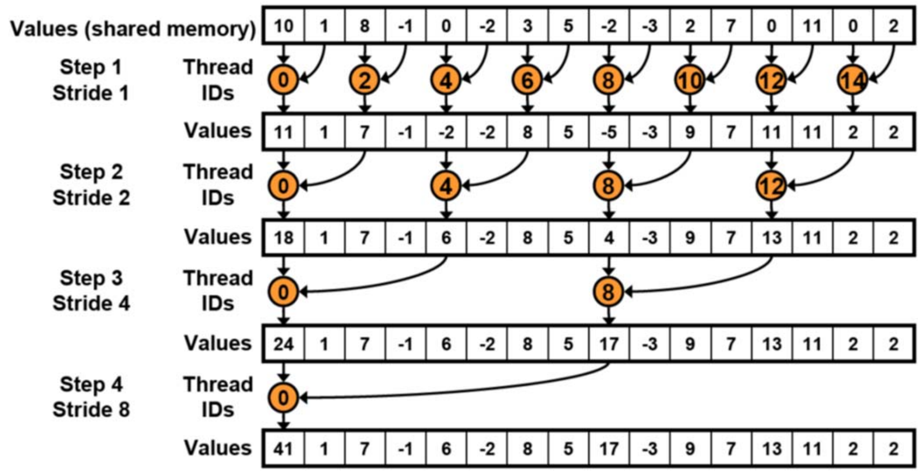
- Reductions have very **low arithmetic intensity**
  - The ratio of arithmetic operations to memory operations is low

► **Bandwidth will be the limiter**
  - Should be considered for optimization

Reduction #1: Interleaved Addressing

- Each thread loads one element from global memory to shared memory
- A thread adds two elements (**Reduction**)
- Half of the threads is deactivated at the end of each step

```
__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) { // step = s x 2
        if (tid % (2*s) == 0) { // only threadIDs divisible by the step participate
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

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



Values (shared memory) 10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2
Step 1 Stride 1
Thread IDs 0 2 4 6 8 10 12 14
Values 11 1 7 -1 -2 -2 8 5 -5 -3 9 7 11 11 2 2
Step 2 Stride 2
Thread IDs 0 4 8 12
Values 18 1 7 -1 6 -2 8 5 4 -3 9 7 13 11 2 2
Step 3 Stride 4
Thread IDs 0 8
Values 24 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2
Step 4 Stride 8
Thread IDs 0
Values 41 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2

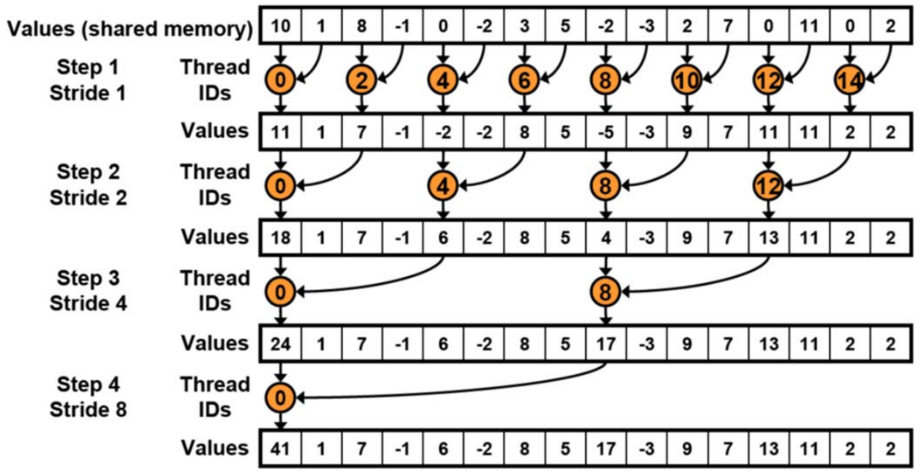
	Time (2^{22} ints)	Bandwidth
Kernel 1: interleaved addressing with divergent branching	**8.054 ms**	**2.083 GB/s**

Reduction #1: Interleaved Addressing

- Each thread loads one element from global memory to shared memory
- A thread adds two elements (**Reduction**)
- Half of the threads is deactivated at the end of each step

```
__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) { // step = s x 2
        if (tid % (2*s) == 0) { // only threadIDs divisible by the step participate
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

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

Highly divergent branching



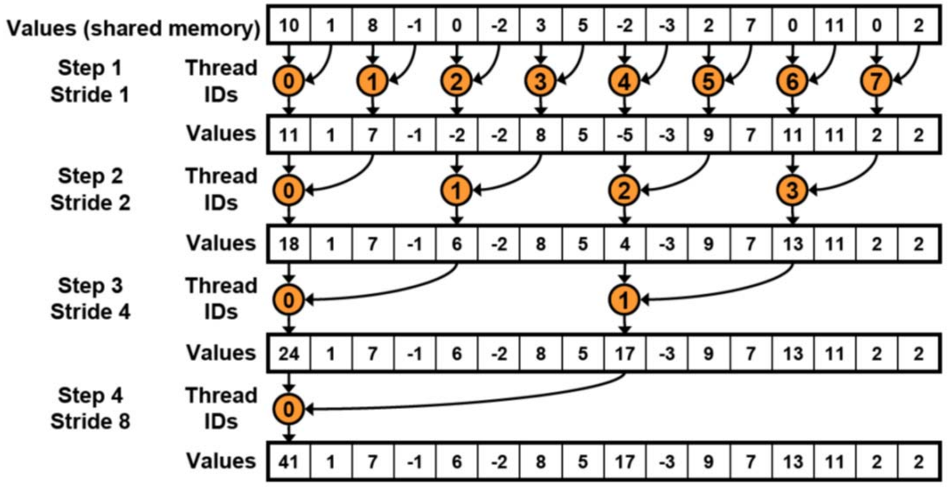
Values (shared memory) 10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2
Step 1 Stride 1
Thread IDs 0 2 4 6 8 10 12 14
Values 11 1 7 -1 -2 -2 8 5 -5 -3 9 7 11 11 2 2
Step 2 Stride 2
Thread IDs 0 4 8 12
Values 18 1 7 -1 6 -2 8 5 4 -3 9 7 13 11 2 2
Step 3 Stride 4
Thread IDs 0 8
Values 24 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2
Step 4 Stride 8
Thread IDs 0
Values 41 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2

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

Reduction #2: Interleaved Addressing

- Replace divergent branch with **strided index** and **non-divergent branch**

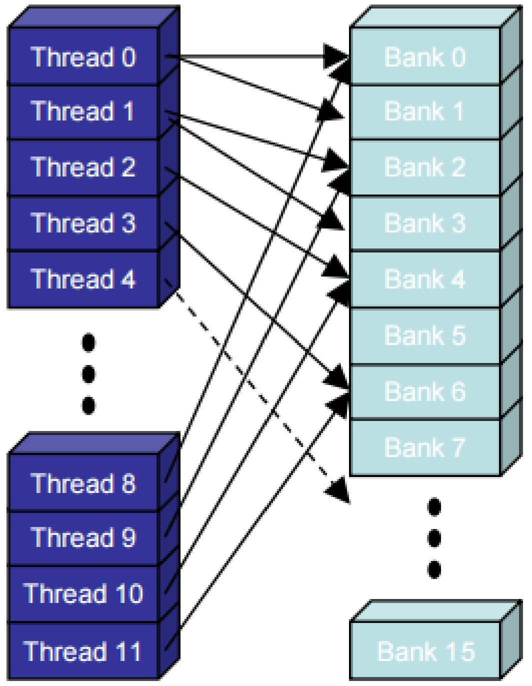
```
// do reduction in shared mem
for (unsigned int s=1; s < blockDim.x; s *= 2) {
        int index  = 2 * s * tid;

        if (index < blockDim.x / s) {
                sdata[index] += sdata[index + s];
        }
        __syncthreads();
}
```

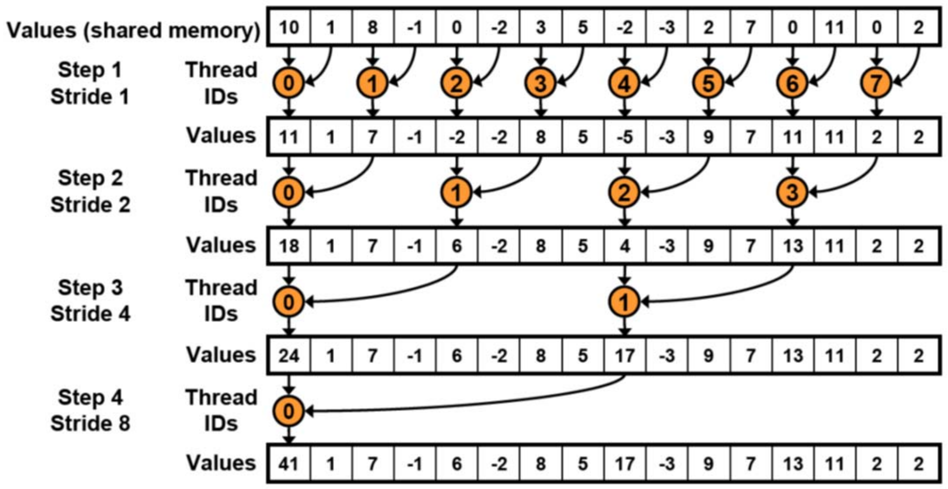


	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing non-divergent branching	3.456 ms	4.854 GB/s	2.33x	2.33x

Reduction #2: Interleaved Addressing

- Replace divergent branch with **strided index** and **non-divergent branch**





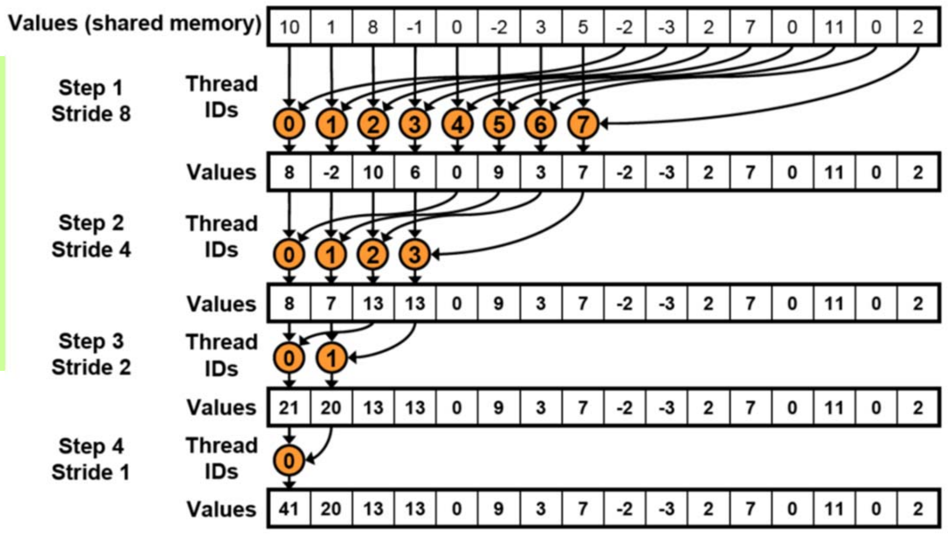
Values (shared memory)	10	1	8	-1	0	-2	3	5	-2	-3	2	7	0	11	0	2
Step 1 Stride 1 Thread IDs	0		1		2		3		4		5		6		7	
Values	11	1	7	-1	-2	-2	8	5	-5	-3	9	7	11	11	2	2
Step 2 Stride 2 Thread IDs	0				1				2				3			
Values	18	1	7	-1	6	-2	8	5	4	-3	9	7	13	11	2	2
Step 3 Stride 4 Thread IDs	0								1							
Values	24	1	7	-1	6	-2	8	5	17	-3	9	7	13	11	2	2
Step 4 Stride 8 Thread IDs	0															
Values	41	1	7	-1	6	-2	8	5	17	-3	9	7	13	11	2	2

- **Shared Memory Bank Conflicts** (2-way bank conflicts)
  - The total number of banks is fixed → multiple addresses of a memory request map to the same memory bank (**Bank Conflict**) → the hardware splits a memory request **decreasing the effective bandwidth**

Reduction #3: Sequential Addressing

- Replace stride indexing with reversed loop and threadId-based indexing

```
// do reduction in shared mem
for (unsigned int s = blockDim.x/2; s > 0; s /= 2) {

        if (tid < s) {
                sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
}
```



Values (shared memory) 10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2
Step 1 Stride 8
Thread IDs 0 1 2 3 4 5 6 7
Values 8 -2 10 6 0 9 3 7 -2 -3 2 7 0 11 0 2
Step 2 Stride 4
Thread IDs 0 1 2 3
Values 8 7 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2
Step 3 Stride 2
Thread IDs 0 1
Values 21 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2
Step 4 Stride 1
Thread IDs 0
Values 41 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2

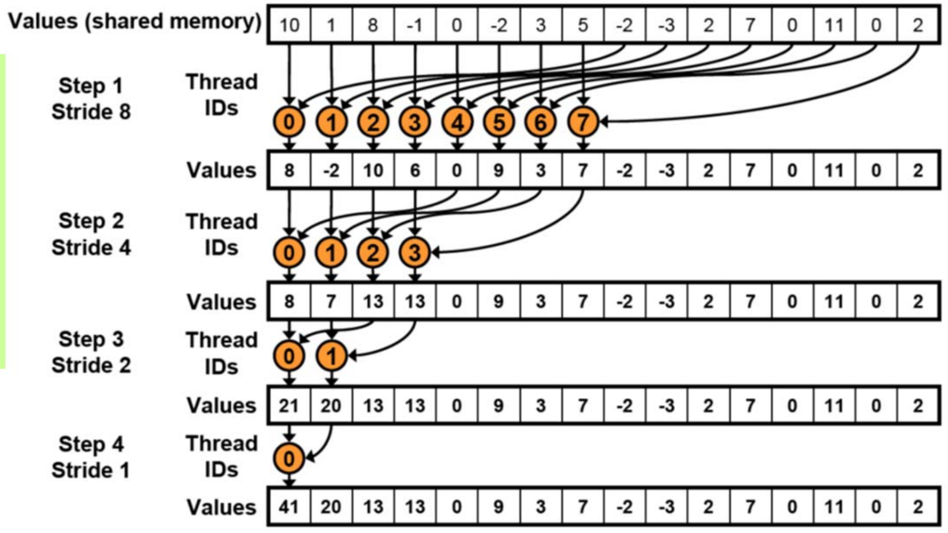
	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing non-divergent branching	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x

Reduction #3: Sequential Addressing

- Replace stride indexing with reversed loop and threadId-based indexing

```
// do reduction in shared mem
for (unsigned int s = blockDim.x/2; s > 0; s /= 2) {

        if (tid < s) {
                sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
}
```

Half of the threads becomes idle for each step



	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 non-divergent branching	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

Reduction #4: First Add During Load

- Read two elements from global memory when allocating shared memory
- The allocation process performs the first reduction

```
// each thread loads two elements from global to shared mem
// end performs the first step of the reduction
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();
```

	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 non-divergent branching	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 step during global load	0.965 ms	17.377 GB/s	1.78x	8.34x

Reduction #4: First Add During Load

- Memory bandwidth is still underutilized
  - 86.4GB/s for tested device
  - Low arithmetic density
- Potential bottleneck is instruction overhead
  - **Ancillary**(보조적인) **instructions** that are not loads, stores, or arithmetic for the core computation
  - **Address arithmetic and loop overhead**

▶ **Unroll Loops**

- Instructions are SIMD synchronous within a warp
- When the number of active threads is less than 32
  - **__synchthreads()** is not needed
  - **if(tid < s)** is not needed

Reduction #5: Unroll the Last Warp

- Unroll the last 6 iterations

```
// do reduction in shared mem
for (unsigned int s = blockDim.x/2; s > 32; s /= 2) {

        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];
}
```

	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 non-divergent branching	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 step 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

Reduction #5: Unroll the Last Warp

- Unroll the last 6 iterations

```
// do reduction in shared mem
for (unsigned int s = blockDim.x/2; s > 32; s /= 2) {

    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

Still have iterations

```
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];
}
```

	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 non-divergent branching	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 step 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

Reduction #6: Completely Unrolled

- Complete unrolling
  - Block size is limited to 512(1024) threads and power-of-2 block sizes
  - CUDA supports C++ template parameters on device and host functions
  - → Block size can be specified as a function template parameter

```
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];
}
```

	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 non-divergent branching	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 step 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: Complete Unroll	0.381 ms	43.996 GB/s	1.41x	21.16x

Reduction #7: Multiple Adds / Thread

- **Algorithm cascading**
  - Combine sequential and parallel reduction
    - Each thread loads and sums multiple elements into shared memory
    - Tree-based reduction in shared memory
  - Replace load and add two elements with a 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();
```

gridSize steps to achieve coalescing

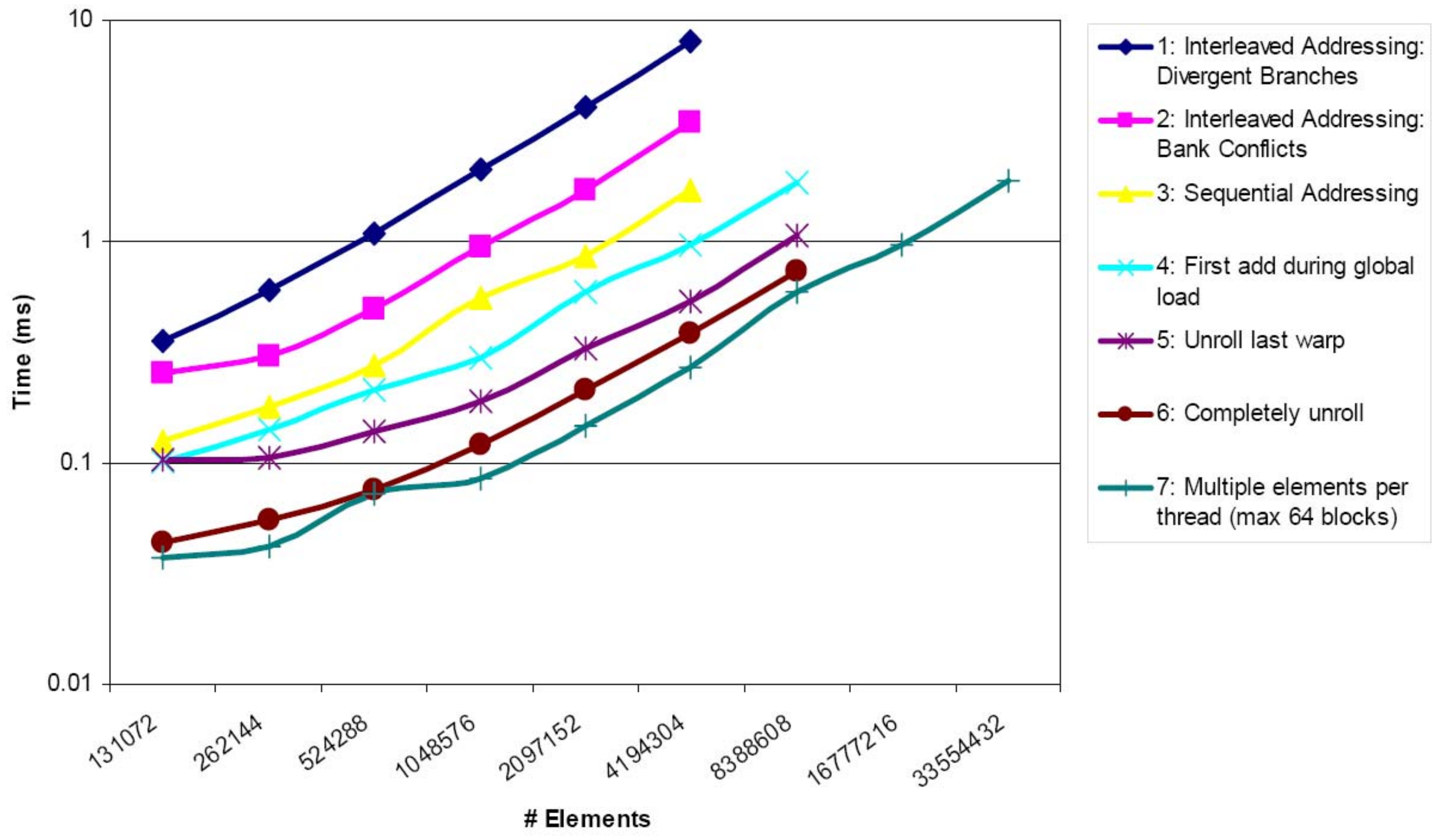
	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing non-divergent branching	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first step 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: Complete Unroll	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

Reduction #7: Multiple Adds / Thread

- **Final Optimized Kernel**

```
template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n)
{
     __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



Conclusion

- **Algorithmic optimizations**
  - Changes to addressing, algorithm cascading (Reduction #1~4, #7)
  - 11.84x speedup
- **Code optimizations**
  - Loop unrolling (Reduction #5, 6)
  - 2.54x speedup
- **From the optimizing process…**
  - 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
  - Optimize your algorithm, then unroll loops
  - Use template parameters to generate optimal code