KOÇ UNIVERSITY

Parallel Reduction in CUDA

Didem Unat

dunat@ku.edu.tr

http://parcorelab.ku.edu.tr

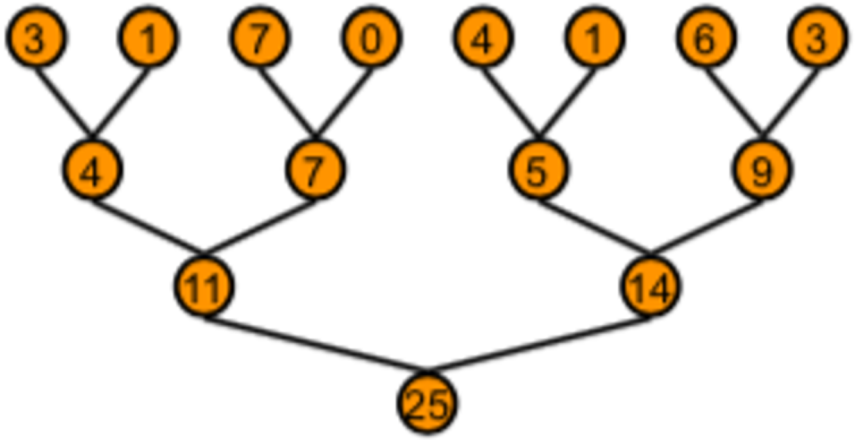
Parallel Reduction

- 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 different versions
  - Demonstrates several important optimization strategies

Reduce(+) = 3+ 1 + 7 + 0 + 4 + 1 + 6 + 3 = 25

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?

Global Synchronization

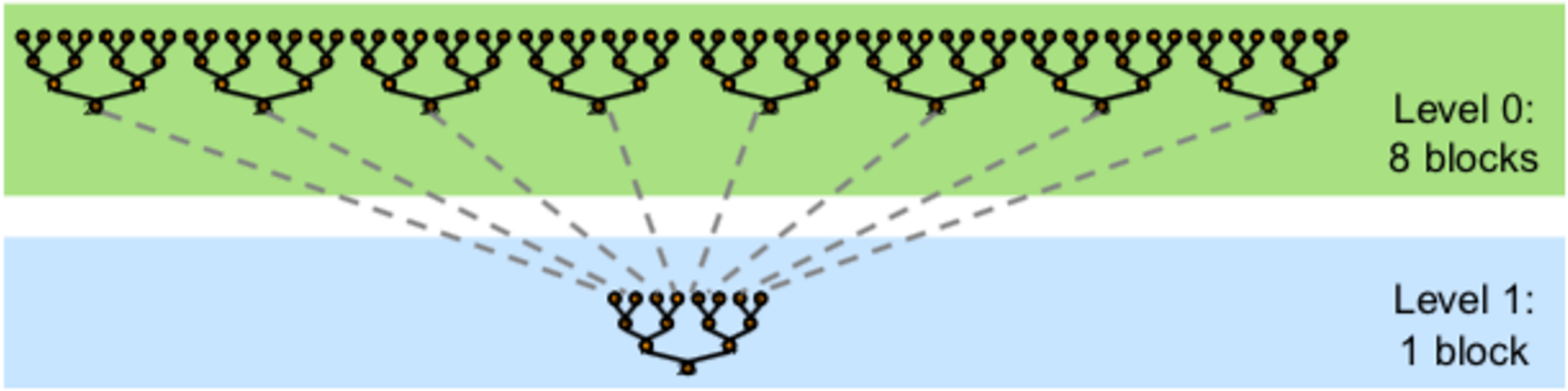
- If we could synchronize across all thread blocks, could easily reduce very large arrays
  - Global sync after each block produces its result
  - Once all blocks reach sync, continue recursively
- Global synchronization in CUDA
  - Using Cooperative Thread Blocks to synchronize within a kernel
  - Force programmer to run fewer blocks (no more than # multiprocessors) to avoid deadlock, which may reduce overall efficiency

Instead, we will use 2-step reduction in this implementation

- Kernel launch serves as a global synchronization point

Solution: Kernel Decomposition

- Avoid global sync by decomposing computation into multiple kernel invocations



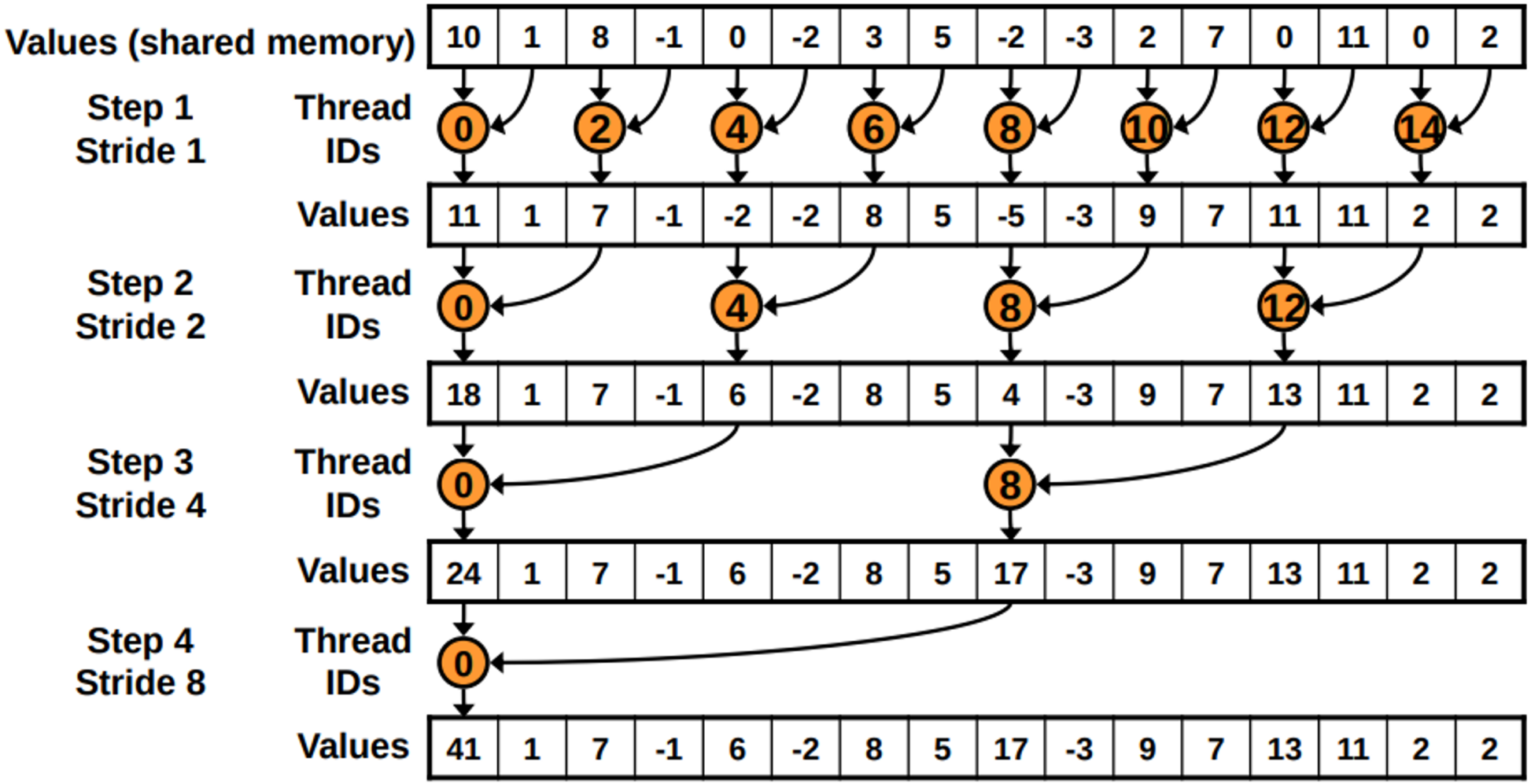
- In the case of reductions, code for all levels is the same
  - Recursive kernel invocation

Lab 9: Reduction

- 7 versions of the reduction kernel

- 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

Reduction #0: Interleaved Addressing



Reduction #0: 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];
}
```

Test Results

- Testing with 134 M elements
- 512 Threads in each Thread Block

Tesla V100	Effective Bandwidth (GB/s)
Kernel 0 (Interleaved Addressing)	131.9

This reduction interleaves which threads are active by using the modulo operator.

- This operator **is very expensive** on GPUs,
- The interleaved inactivity means that **no whole warps are active**, which is also very inefficient

Reduction #0: 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

Reduction #1: Interleaved Addressing

Just replace divergent branch in inner loop:

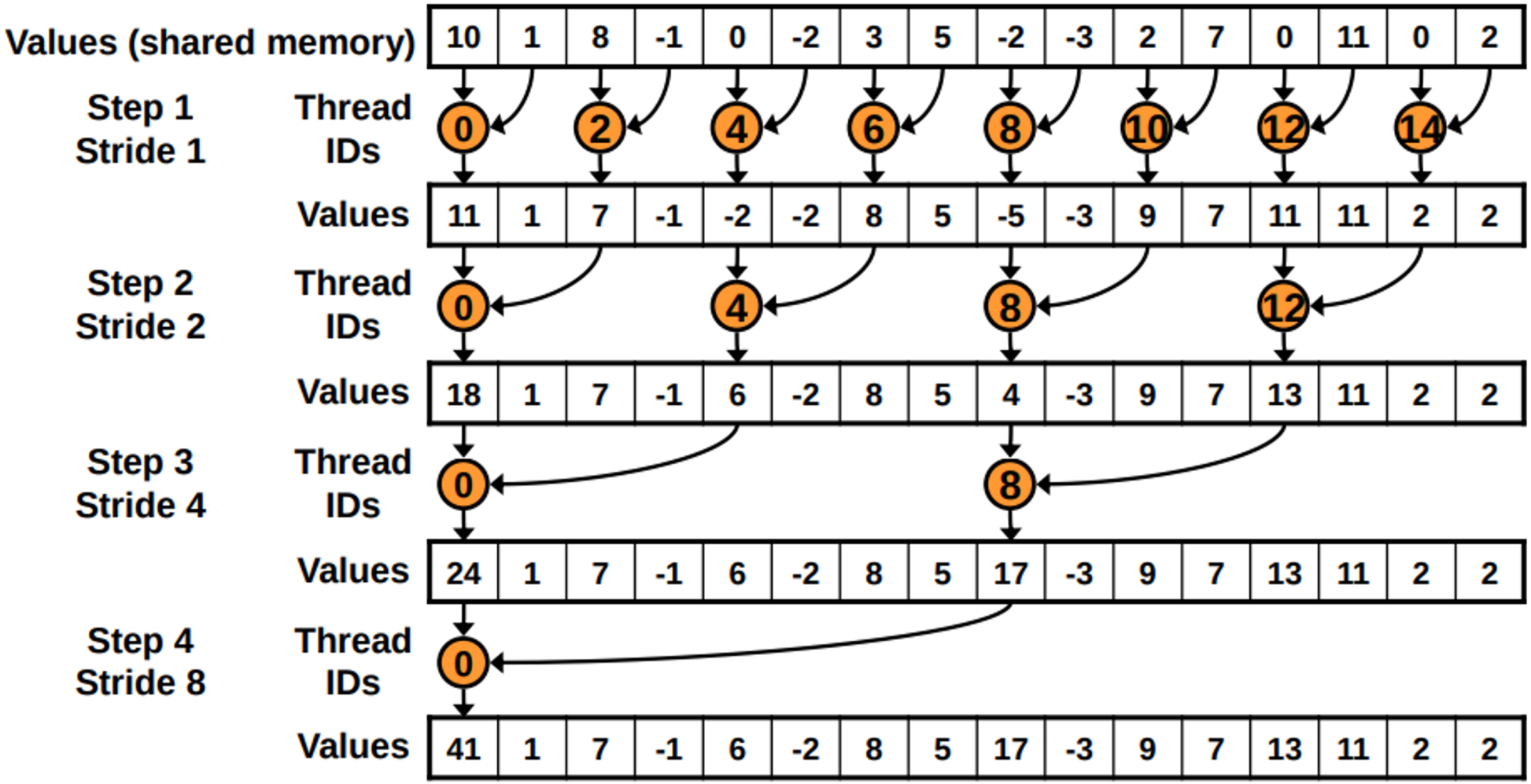
```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    if (tid % (2*s) == 0) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

With strided index and non-divergent branch:

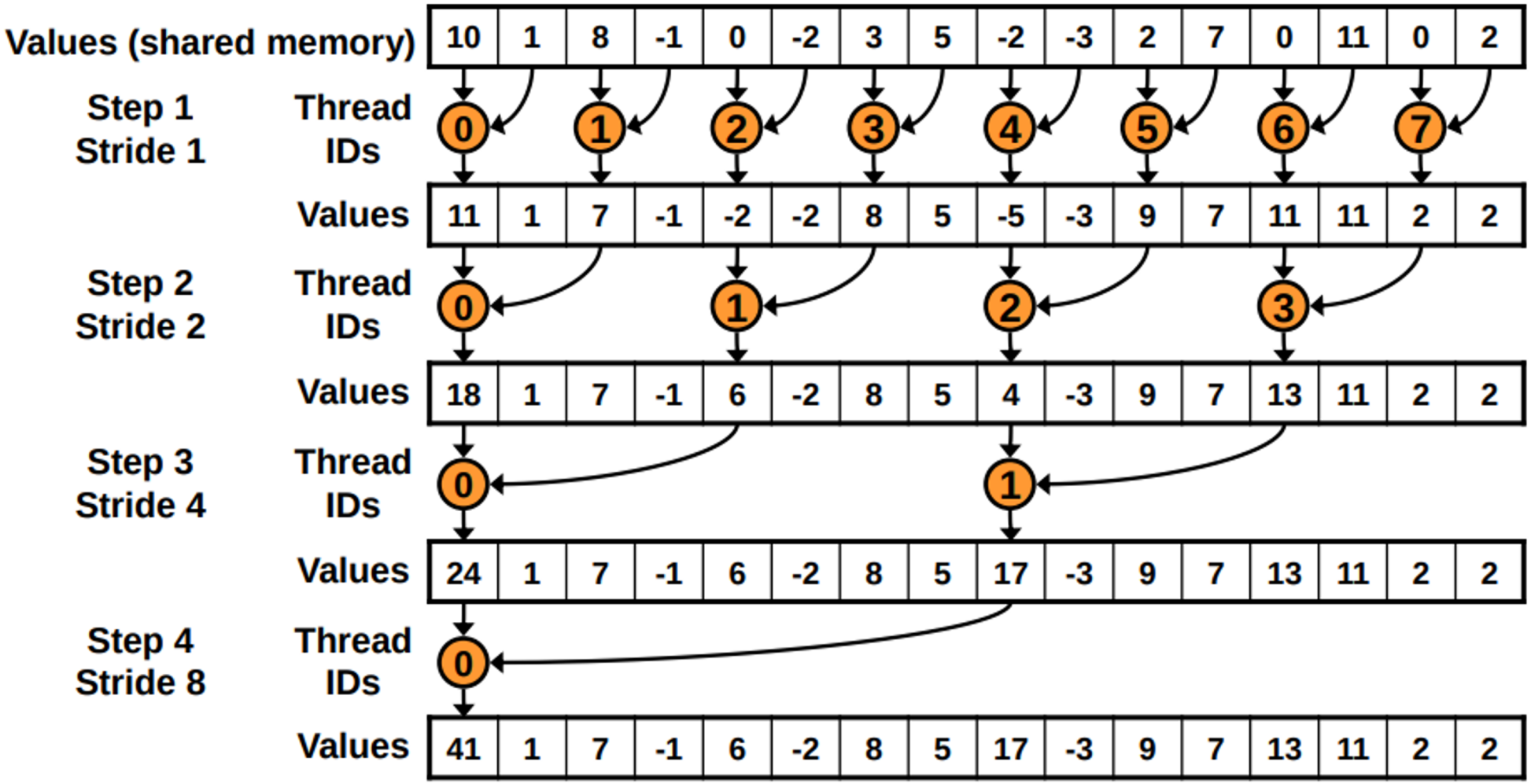
```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    int index = 2 * s * tid;

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

Reduction #0: Interleaved Addressing



Reduction #1: Interleaved Addressing



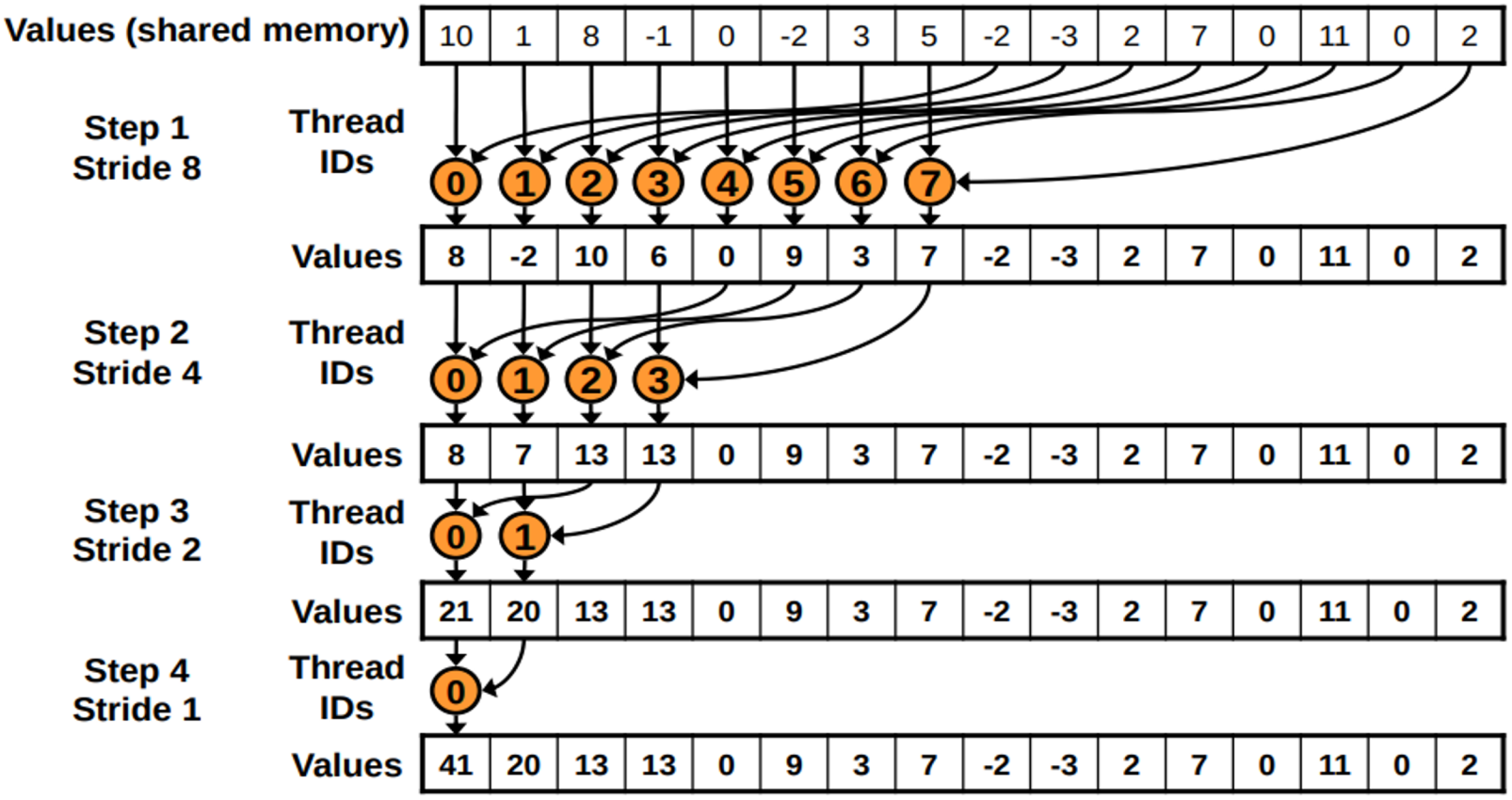
Test Results

- Testing with 134 M elements
- 512 Threads in each Thread Block

Tesla V100	Effective Bandwidth (GB/s)
Kernel 0 (Interleaved Addressing)	131.9
Kernel 1 (No divergent threads)	247.4

This version uses contiguous threads, but its interleaved addressing **results in many shared memory bank conflicts**.

Reduction #2: Sequential Addressing



Sequential addressing is conflict free

Reduction #2: 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();
}
```

Test Results

- Testing with 134 M elements
- 512 Threads in each Thread Block

Tesla V100	Effective Bandwidth (GB/s)
Kernel 0 (Interleaved Addressing)	131.9
Kernel 1 (No divergent threads)	247.4
Kernel 2 (Sequential Addressing)	277.1

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 #3: 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();
```

Test Results

- Testing with 134 M elements
- 512 Threads in each Thread Block

Tesla V100	Effective Bandwidth (GB/s)
Kernel 0 (Interleaved Addressing)	131.9
Kernel 1 (No divergent threads)	247.4
Kernel 2 (Sequential Addressing)	277.1
Kernel 3 (First add during load)	498.1

Reduce #4: Warp Reduce

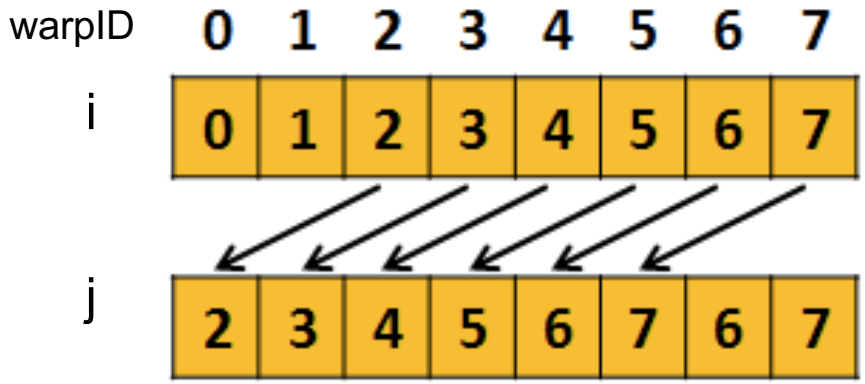
- On earlier versions, reduction uses shared memory, which involves writing data to shared memory, synchronizing, and then reading the data back from shared memory.

- Post Kepler architecture introduces **shuffle instruction (SHFL)** which enables a thread to directly read a register from another thread in the same **warp (32 threads)**

Reduce #4: Warp Reduce

```
int __shfl_down(int var, unsigned int delta,
                int width=warpSize);
```

```
//example
int i = threadIdx.x % 32;
int j = __shfl_down(i, 2, 8);
```



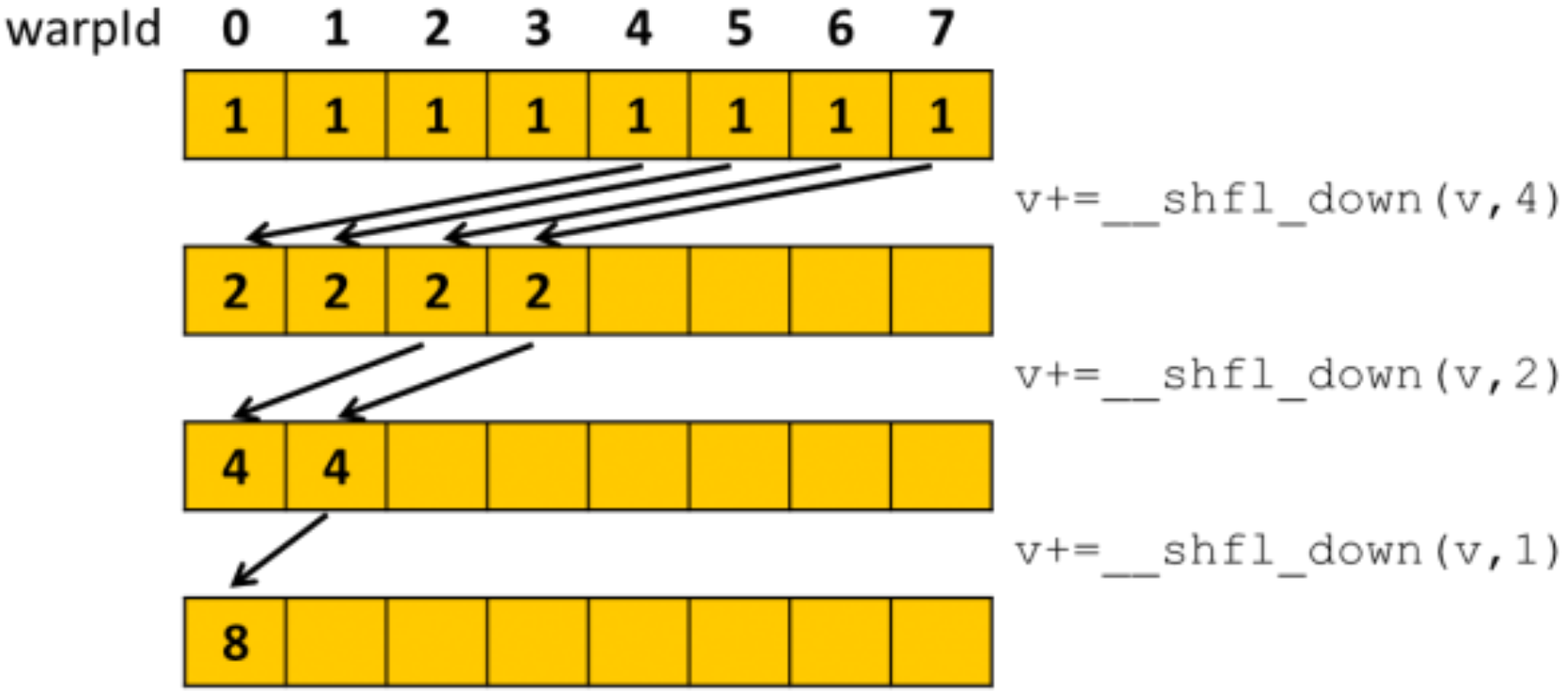
`threadIdx.x % 32`

`__shfl_down(i, 2, 8)`

The effect is values are shifted down by 2 threads.

For brevity, the diagrams that follow show only 8 threads in a warp even though the warp size of all current CUDA GPUs is 32.

Reduce #4: Shuffle Warp Reduce



- We can use shuffle down to build a reduction tree.
- After executing the three reductions thread 0 has the total reduced value in its variable v.
- For 32-thread warps, we need 5 steps to reduce.

Reduce #4: Warp Reduce

- Using the warp reduce, we can now easily build a reduction across the entire thread block.
- To do this we first reduce within warps.
- Then the first thread of each warp writes its partial sum to shared memory.
- Finally, after synchronizing, the first warp reads from shared memory and reduces again.

Test Results

- Testing with 134 M elements
- 512 Threads in each Thread Block

Tesla V100	Effective Bandwidth (GB/s)
Kernel 0 (Interleaved Addressing)	131.9
Kernel 1 (No divergent threads)	247.4
Kernel 2 (Sequential Addressing)	277.1
Kernel 3 (First add during load)	498.1
Kernel 4 (Warp Reduce)	648.2

Reduction #5: Unrolling Loops

- Let's unroll the last 6 iterations of the inner loop

```
 // do reduction in shared mem
  for (unsigned int s = blockDim.x / 2; s > 32; s >>= 1) {
    if (tid < s) {
      sdata[tid] = mySum = mySum + sdata[tid + s];
    }

    __syncthreads();
}
```

Reduction #5: Unrolling Loops

- Specify block size as a function template parameter:

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

```
switch (threads) {
    case 512:
      reduce5<T, 512>
          <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
      break;

    case 256:
      reduce5<T, 256>
          <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
      break;
 …..
```

Test Results

- Testing with 134 M elements
- 512 Threads in each Thread Block

Tesla V100	Effective Bandwidth (GB/s)
Kernel 0 (Interleaved Addressing)	131.9
Kernel 1 (No divergent threads)	247.4
Kernel 2 (Sequential Addressing)	277.1
Kernel 3 (First add during load)	498.1
Kernel 4 (Warp Reduce)	648.2
Kernel 5 (Unroll)	654.0

Reduction #6: 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 #6: 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!

Reduction #6: Multiple Adds

- This version adds multiple elements per thread sequentially.
- This reduces the overall cost of the algorithm while keeping the work complexity O(n) and the step complexity O(log n). (Brent's Theorem optimization)
- Brent's theorem says each thread should sum O(log n) elements
  - i.e. 1024 or 2048 elements per block vs. 256
- With larger input sizes, the reduction performance increases

Test Results

- Testing with 134 M elements
- 512 Threads in each Thread Block

Tesla V100	Effective Bandwidth (GB/s)
Kernel 0 (Interleaved Addressing)	131.9
Kernel 1 (No divergent threads)	247.4
Kernel 2 (Sequential Addressing)	277.1
Kernel 3 (First add during load)	498.1
Kernel 4 (Warp Reduce)	648.2
Kernel 5 (Unroll)	654.0
Kernel 6 (Multiple Adds)	784.7

Types of optimization

- Observations:
  - Each thread loads and sums multiple elements into shared memory
  - Tree-based reduction in shared memory

- Algorithmic optimizations
  - Changes to addressing, indexing
  - Loop unrolling
  - Warp reduction

Conclusion

- Understand CUDA performance characteristics:
  - Memory coalescing
  - Divergent branching
  - Bank conflicts
  - Latency hiding

- 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

References

- https://devblogs.nvidia.com/faster-parallel-reductions-kepler/
- Presentation from 2007 by Mark Harris