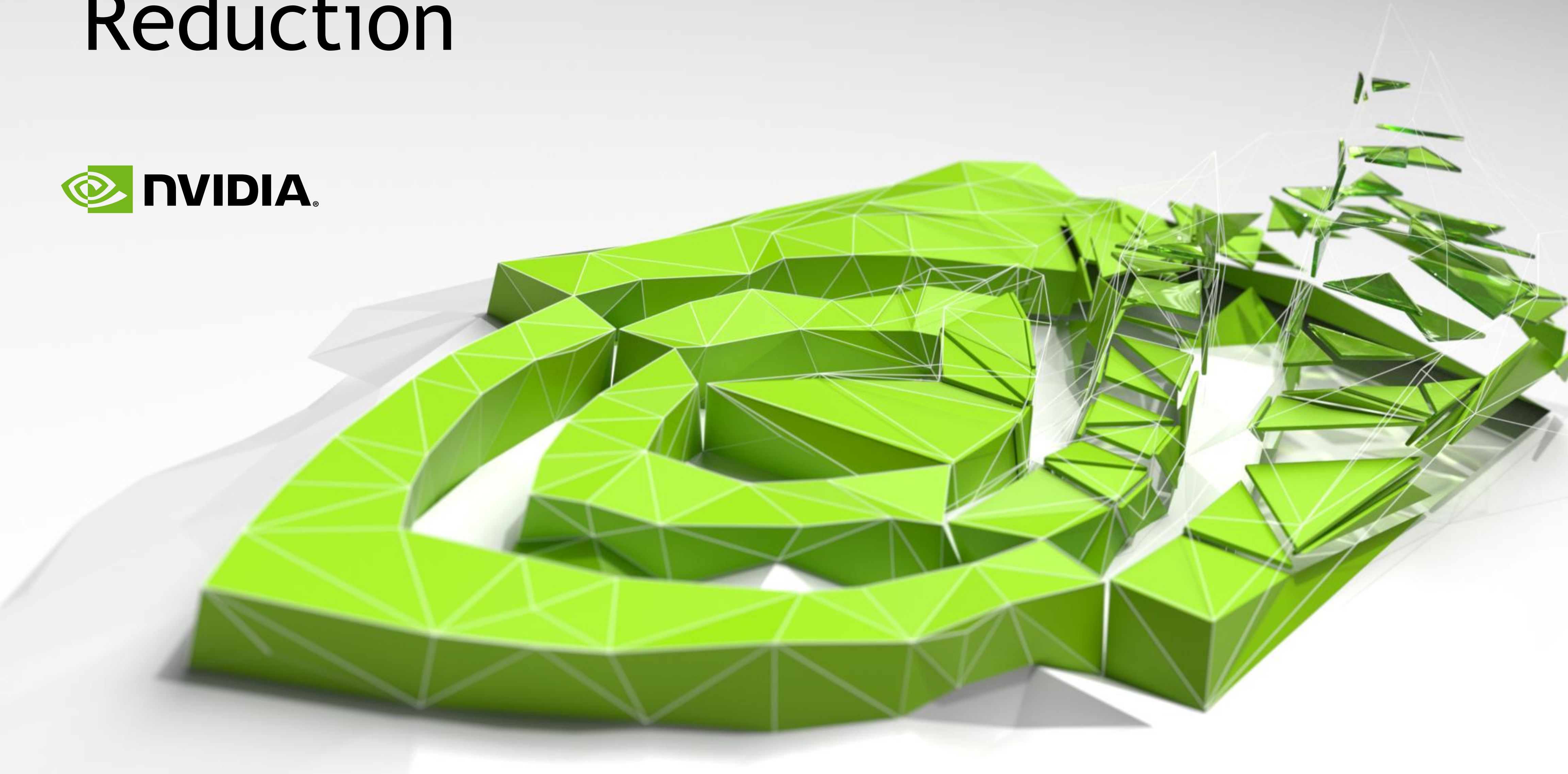
Reduction



Parallel Reduction

Adapted from Mark Harris, Chief Technologist GPU Computing software

- Common and important data primitive
  - Apply a binary operator to pairs of data to reduce the data to one final value
  - SUM, MIN, MAX, PROD, etc.
- Easy to implement on a GPU
  - Harder to get it right (where “right” == “fast”)
- Great optimization example
  - We’ll walk through 7 different versions using SUM reduction
  - Various strategies including code optimization and use of libraries

Performance Goal

- We should strive to reach GPU peak performance
- Choose the right metric
  - Compute-bound kernels—often measured in Gflop/s
  - Memory-bound kernels—often measured in GB/s
- Reduction loads two elements from memory to perform one computation
  - Therefore low computation:communication so we’ll optimize for bandwidth
- GRID K520, 256bit memory bus, 2500 GHz clock, DDR (double data rate)
  - 256 * 2500 / 8 = 160 GB/s

Serial reduction

- Open **`exercises/cuda/reduction_naive/kernel.cu`**
- Finish the kernel by replacing **`FIXME`** with code.
    - Single thread, single thread block
        - Loop will be serial only. No parallel component yet.
    - Performance will be abysmal, but that's ok.
    - **`FLOATTYPE_T`** can be **`float`** or **`double`**. Choose **`float`** for now.
    - PASS/FAIL (tested against CPU) and performance info will be printed.
- Measure performance in GB/s, how does it compare to serial CPU perf?

Serial reduction

```
/* loop for calculating the result */
  for( int i = 0; i < N; i++ )
  {
    result += in[i];
  } /* end for */

/* write the result to global memory */

  *sum = result;
  return;
```

Parallel Reduction

- How to use multiple threads and result with only one final value?
- Atomics!
  - One thread gets exclusive access to a location in global memory.
  - Updates the global memory value with a binary operation and then releases the lock.
- Algorithm
  - Each thread calculates a local sum.
  - Each thread atomically updates the global sum with it’s local value

Grid-Stride looping

Short segue into writing data size independent kernels

- Kernels launch one thread per data element. Not always a good idea:
  - If data size is larger than total threads GPU can launch, kernel will fail.
  - Often one thread operating on multiple elements is better for memory utilization
- Solution is writing Grid-stride loops inside the kernel

```
for( int i = globalThreadIndex; i < n; i += blockDim.x * gridDim.x )
  {
       //do work here on data array with index i
  } /* end for
```

Parallel Reduction with Atomics

- exercises/cuda/reduction_atomic/kernel.cu
- Finish the kernel
- Use **`atomicAdd`**
  - **`float atomicAdd(float* address, float val);`**
  - **`val`** is the thread local value, **`address`** is the location in global memory where the global sum will be accumulated. Return value is original value in *address
- Measure performance again. Any improvement over serial version?

Reduction with atomicAdd

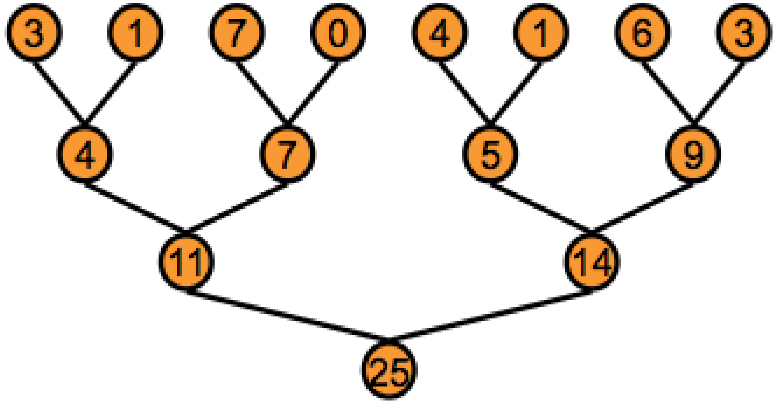
```
/* grid stride loop where array is larger than number of threads
 * launched, using atomics
 */

  for( int i = globalIndex; i < n; i += blockDim.x * gridDim.x )
  {
      atomicAdd( sum, in[i] );
  } /* end for */
```

Parallel Reduction

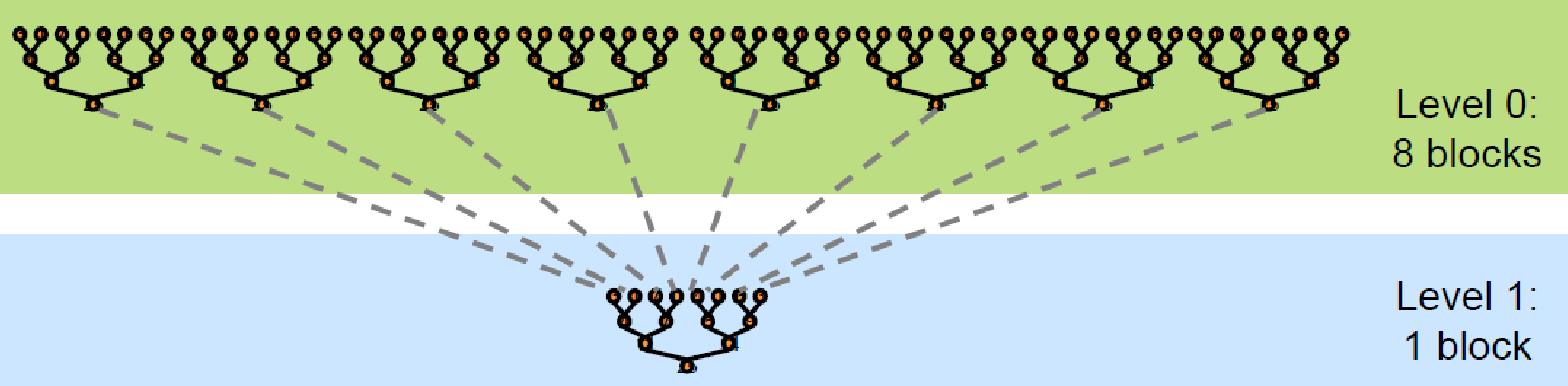
- So far two functionally-correct programs but performance is bad.
    - Atomics have performance penalty and hopefully can be used sparingly.
- Need to consider fully parallel algorithm.
- If the array is size N, we could do N/2 parallel, independent sums
    - This would reduce the data size to N/2.
    - Now we could do N/4 independent sums and the data would be reduced to N/4.
    - Continue on in this fashion until data size is reduced to one value.
    - Requires ability for threads to share data between themselves.

Tree Reduction



- Within a thread block we can use a tree algorithm with shared memory.
  - At each iteration there are N values so we use N/2 threads to reduce to N/2 values
  - Finally we are left with a single thread calculating the final value.
- Want to use many thread blocks. Each block produces a final value.
  - How to aggregate the results from each thread block?

Tree Reduction with multiple kernels



- Launch the kernel twice.
  - First time called with many threadblocks to get one value per threadblock.
  - Second time called with only one threadblock to compute the final value.

Parallel Reduction

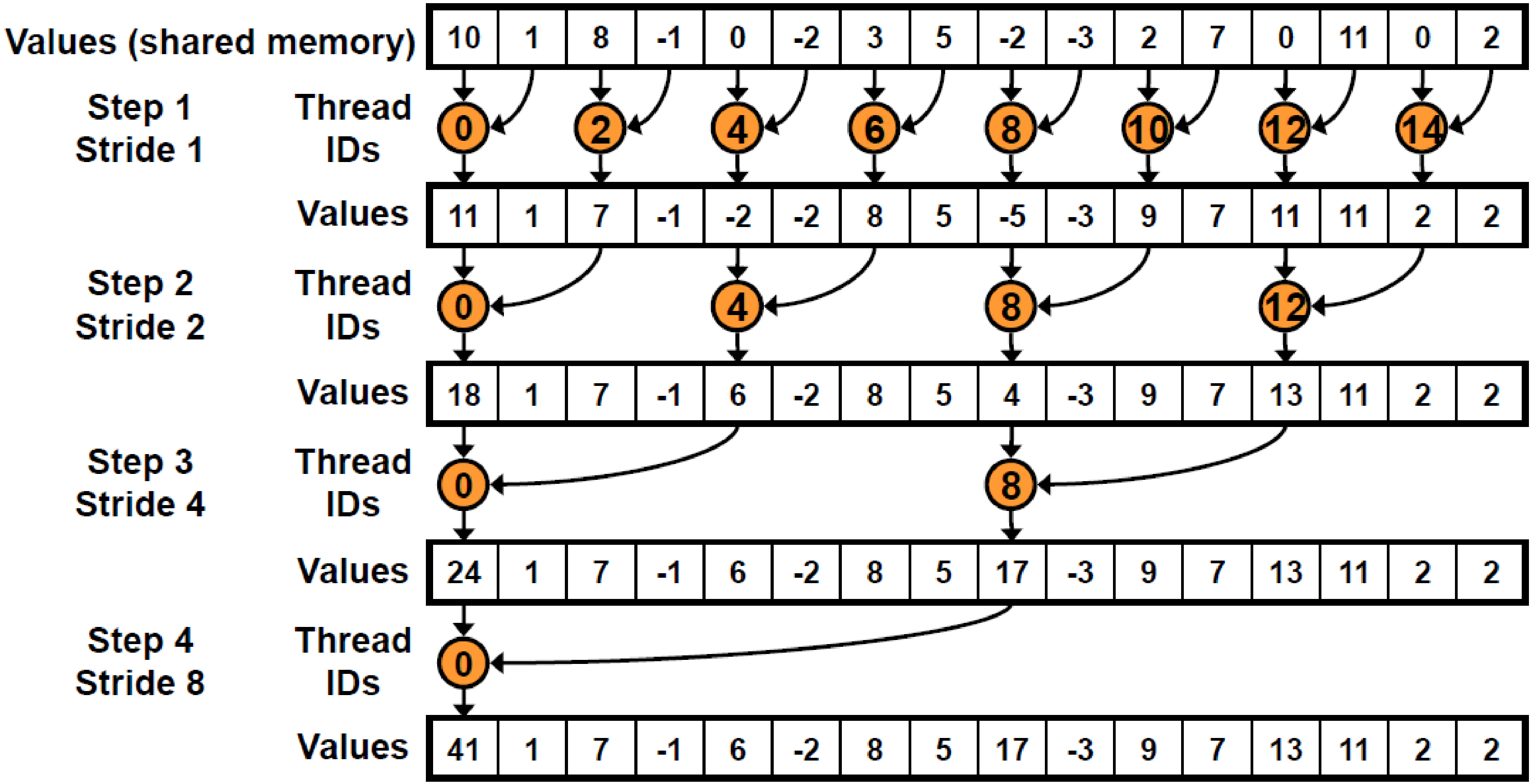
High level

- Each thread grid-stride loops through the array in global memory and accumulates a per-thread local sum
- Each thread writes its per-thread sum into SMEM
- Using some reduction algorithm, each threadblock collaboratively reduces the values in SMEM to one value and writes this value to global memory.
- The same reduction kernel is called again, with input array equal to output array of prior kernel launch.
- Result is one final reduced value.

Blockwide tree reduction

- How to decide good algorithm to reduce the values in SMEM?
- One option is to use every other thread add two adjacent values together, storing them in SMEM.
- Call this method Interleaved Addressing.

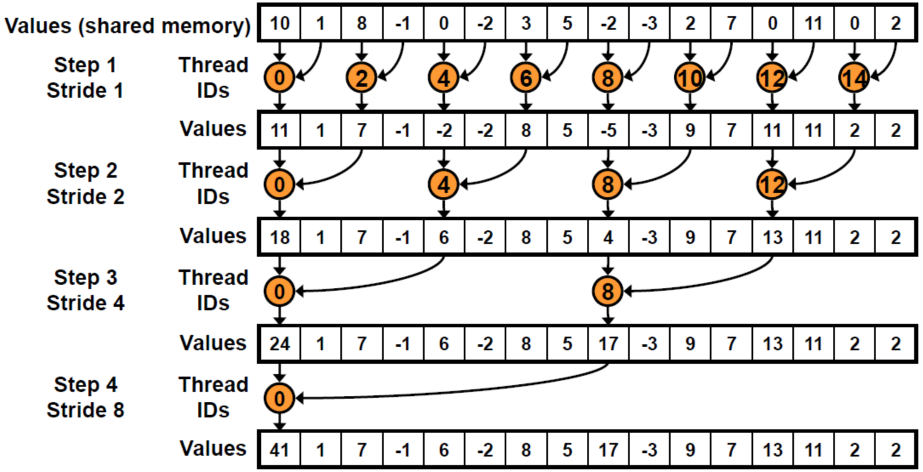
Parallel Reduction with interleaved addressing



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

Interleaved Addressing

- **`exercises/cuda/reduction3/kernel.cu`**
- Finish kernel using interleaved addressing.
- Assume powers of 2 for data and threads.
- What kind of performance can you achieve?
- **You are using SMEM. What should you add??**



	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

Interleaved Addressing

```
 for( int i = globalIndex; i < n; i += blockDim.x * gridDim.x )
  {
    sArray[threadIdx.x] += in[i];
  } /* end for */
  __syncthreads();
/* do the final reduction in SMEM */
  for( int i = 1; i < blockDim.x; i = 2*i )
  {
    if( threadIdx.x % (2*i) == 0 )
    {
      sArray[threadIdx.x] += sArray[threadIdx.x + i];
    } /* end if */
  __syncthreads();
  } /* end for */

/* thread0 writes the thread block reduced value back to global memory
*/
  if( threadIdx.x == 0 ) out[blockIdx.x] = sArray[0];
```

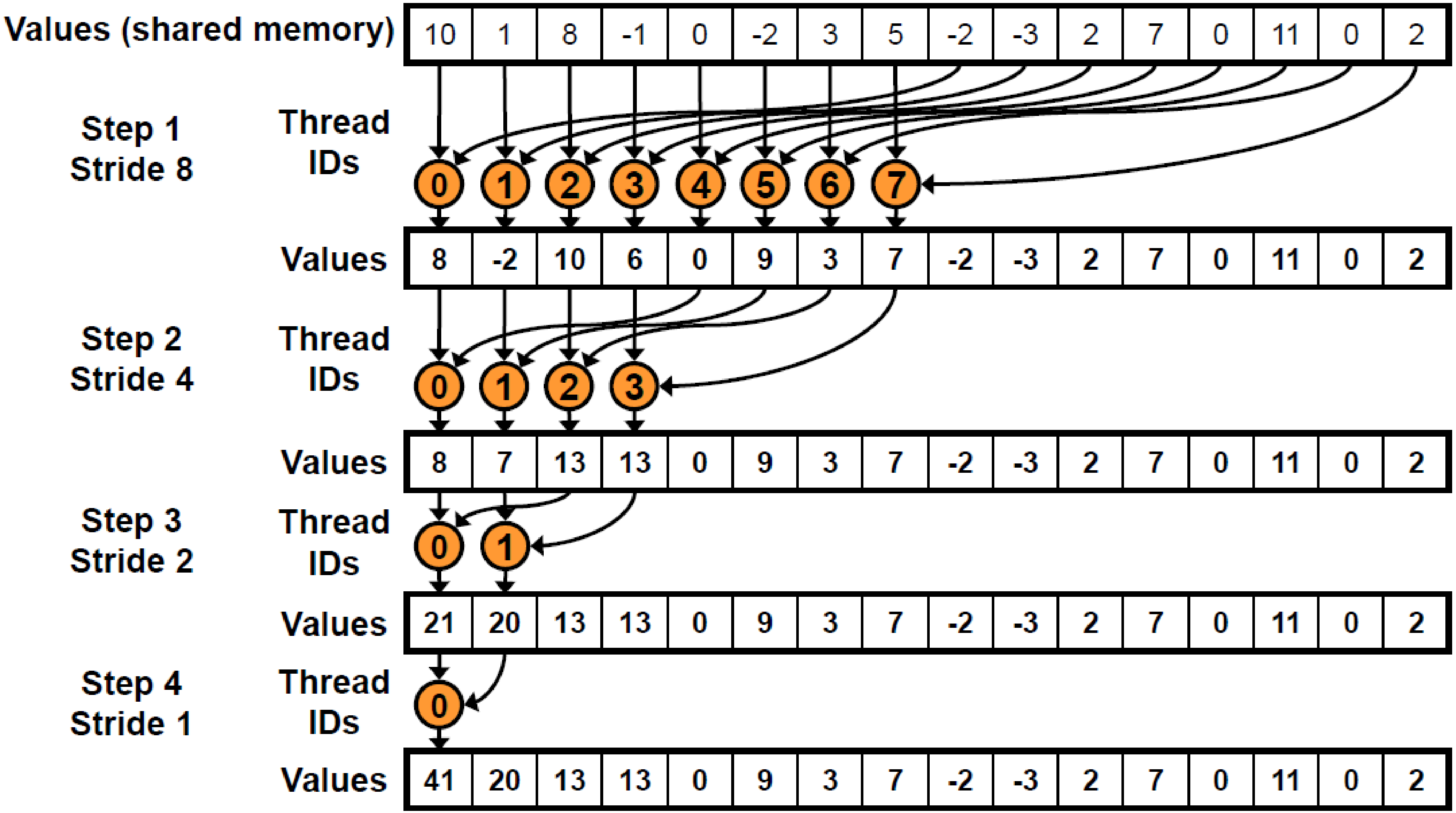
Reduction

- Pretty reasonable performance, where might we improve?
- Due to the nature of how warps operate we might conclude that as the tree reduction proceeds, we'll have lots of active warps but only a few threads in each warp active.
    - Can we aggregate the work such that entire warps can have as many active threads as possible?

Sequential Addressing

- Instead of working on adjacent data elements, each thread can work on elements that are far away from each other in shared memory.
- In each iteration the upper half of the threads will then drop off and not participate, but the lower half of the threads will remain and so we'll have contiguous threads (therefore contiguous warps also) continuing to work.

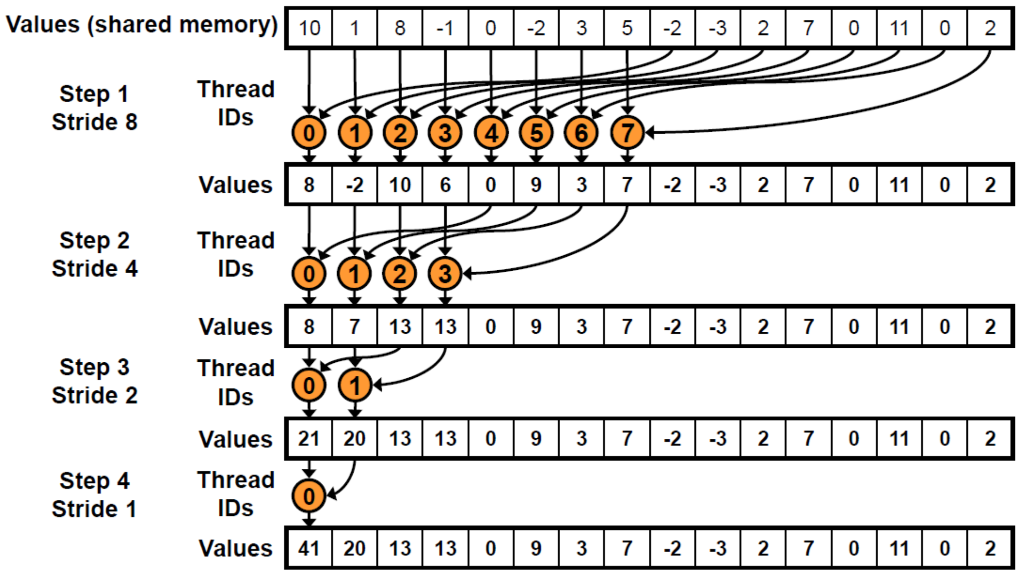
Parallel Reduction with sequential addressing



	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

Sequential Addressing

- **`exercises/cuda/reduction4/kernel.cu`**
- Finish kernel using sequential addressing.
- Assume powers of 2
- **Using SMEM, what should you include?**
- What perf can you achive?



	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

Sequential Addressing

```
 for( int i = globalIndex; i < n; i += blockDim.x * gridDim.x )
  {
    sArray[threadIdx.x] += in;
  } /* end for */
  __syncthreads();

/* do the final reduction in SMEM */
  for( int i = blockDim.x/2; i > 0; i = i / 2 )
  {
    if( threadIdx.x < i )
    {
      sArray[threadIdx.x] += sArray[threadIdx.x + i];
    } /* end if */
  __syncthreads();
  } /* end for */

/* thread0 of each threadblock writes the result to global memory */
  if( threadIdx.x == 0 ) out[blockIdx.x] = sArray[0];
```

Further Optimizations

- We could continue making incremental performance changes.
- As we attempt to squeeze more performance out, the changes we make would likely become more hardware-specific and even data type specific.
  - Not a bad thing, but does result in less portable code depending on use cases.
- Can we leverage a library that already exists?

CUB (CUDA unbound)

http://nvlabs.github.io/cub/

- NVIDIA-provided library of performance portable primitives for use in kernels.
    - Warp-wide primitives—prefix-scan, reduction, etc.
    - Block-wide primitives—sort, scan, reduction, histogram, etc.
    - Device-wide primitives—parallel sort, reduction, histogram, etc.
- Great choice to use. Insert function call and let NVIDIA engineers worry about architecture and hardware-specific performance!

CUB block-wide reduction

http://nvlabs.github.io/cub/index.html

- **`exercises/cuda/reduction_cub_block`**
- Keep our kernel grid stride loop and call CUB to compute the SMEM reduction.
- Uses C++ templates.
  - If you're not familiar with them, just treat the stuff in the <> as arguments.
- Requires us to allocate some temporary SMEM storage for CUB to use.
  - **`__shared__ cub::BlockReduce<typename T, BLOCK_DIM_X>::TempStorage t1;`**
- Once that is complete, we can call the function as follows
  - **`sum = cub::BlockReduce<T, BLOCK_DIM_X>(t1).Sum(tempResult);`**

CUB block wide reduction

```
/* Allocate shared memory for cub::BlockReduce */
   __shared__ cub::BlockReduce<FLOATTYPE_T,
                THREADS_PER_BLOCK>::TempStorage sArray;
```

…

```
/* Compute the block-wide sum for thread0 */
   FLOATTYPE_T blockSum = cub::BlockReduce<FLOATTYPE_T,
                 THREADS_PER_BLOCK>(sArray).Sum(tempResult);

/* write the result back to global memory */
  if( threadIdx.x == 0 ) out[blockIdx.x] = blockSum;
```

CUB device-wide reduction

http://nvlabs.github.io/cub/index.html

- Performance improved when using CUB block-wide reduction
- CUB also has device-wide functions
- Instead of calling functions inside a kernel, if ALL we want to do is reduction, we can call the CUB reduction on the entire data set, instead of calling a kernel.
- This will allow CUB to attempt to use best algorithm for entire reduction, rather than only the SMEM portion.

CUB device-wide reduction

http://nvlabs.github.io/cub/index.html

- **`exercises/cuda/reduction_cub_device`**
- No kernel at all! Replace kernel call with CUB function call.
- Requires us to allocate some temporary global mem storage for CUB to use.
  - First call the function with temp_storage=NULL to calculate temp_storage_bytes
  - cub::DeviceReduce::Sum(temp_storage, temp_storage_bytes, input, sum, numItems)
- Once that is complete, we can call the function as follows
  - cub::DeviceReduce::Sum(temp_storage, temp_storage_bytes, input, sum, numItems)

CUB device-wide reduction

```
 void *d_temp_storage = NULL;
  size_t temp_storage_bytes = 0;
  cub::DeviceReduce::Sum( d_temp_storage, temp_storage_bytes, d_in,
d_out,
    size );

  checkCUDA( cudaMalloc( &d_temp_storage, temp_storage_bytes ) );

…

/* launch the kernel on the GPU */
  cub::DeviceReduce::Sum( d_temp_storage, temp_storage_bytes, d_in,
d_out,
    size );
```

Thrust

https://developer.nvidia.com/Thrust

- NVIDIA also has a C++ STL like library
- Higher level than writing CUDA C/C++
- Handles data allocation/deallocation and memcopy at a higher level.
  - `thrust::host_vector<float> h_vec( size );`
    - creates vector of float type and size elements
  - `thrust::device_vector<float> d_vec = h_vec;`

Thrust

https://developer.nvidia.com/Thrust

- **`exercises/cuda/reduction_thrust/kernel.cu`**
- Finish the thrust calls.
- What’s the performance?

Thrust

```
/* create the host array */
  thrust::host_vector<FLOATTYPE_T> h_vec( size );

…

/* transfer data to the device */
  thrust::device_vector<FLOATTYPE_T> d_vec = h_vec;

…

/* reduce data on the device */
  FLOATTYPE_T devResult = thrust::reduce( d_vec.begin(), d_vec.end() );
```

Performance

GRID K520, 2^27 floats

	Time	Bandwidth
Naïve kernel	11.424s	0.0470 GB/s
AtomicAdd	0.505s	1.062 GB/s
Interleaved	0.00552s	97.2 GB/s
Sequential	0.00537s	105.9 GB/s
CUB block	0.00485s	110.6 GB/s
CUB device	0.00432s	124.4 GB/s
Thrust	0.00493	108.9 GB/s

Summary

- Understand important GPU performance characteristics
    - Atomics
    - Thread divergence within warps.
- Use correct performance metric to guide optimization: Gflop/s or GB/s?
- What operation are you doing? If it’s common, there may be a library!
    - CUB--block and device-wide primitives, optimized to all current arch’s
    - Thrust—STL-like library allows high-level GPU programming

NVIDIA