ATOMICS, REDUCTIONS, WARP SHUFFLE

Bob Crovella, 5/13/2020

AGENDA

- Transformations vs. Reductions, Thread Strategy
- Atomics, Atomic Reductions
- Atomic Tips and Tricks
- Classical Parallel Reduction
- Parallel Reduction + Atomics
- Warp Shuffle, Reduction with Warp Shuffle
- Other Warp Shuffle Uses
- Further Study
- Homework

ATOMICS

MOTIVATING EXAMPLE

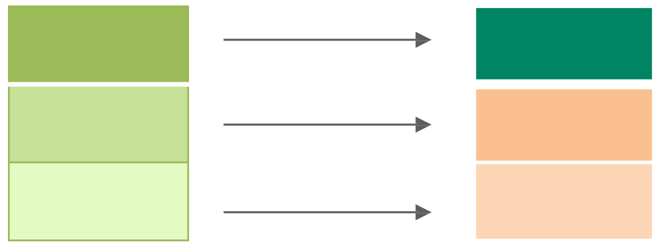
Sum - reduction

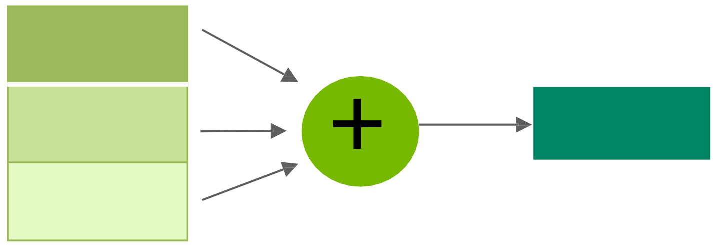
```
const int size = 100000;
float a[size] = {…};
float sum = 0;
for (int i = 0; i < size; i++)
        sum += a[i];
```

-> sum variable contains the sum of all the elements of array a

TRANSFORMATION VS. REDUCTION

May guide the **thread strategy**: what will each thread do?





Transformation:

e.g. c[i] = a[i] + 10;

CUDA thread strategy: one CUDA thread per output point

Reduction:

e.g. $*c = \Sigma\, a[i]$

CUDA thread strategy: ??

REDUCTION: NAÏVE THREAD STRATEGY

One thread per input point

```
*c += a[i];
```

(Doesn’t work.) Actual code the GPU executes:

```
LD R2, a[i]          (Thread independent)
 LD R1, c            (READ)
ADD R3, R1, R2       (MODIFY)
 ST c, R3            (WRITE)
```

But **every CUDA Thread** is trying to do this, potentially at the same time

The CUDA programming model does not enforce any order of thread execution

ATOMICS TO THE RESCUE

indivisible READ-MODIFY-WRITE

old = atomicAdd (&c, a[i]); https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions

Reads the 16-bit, 32-bit or 64-bit word old located at the address in global or shared memory, computes (old + val), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old.

```
LD R2, a[i]              (CUDA thread independent)

LD R1, c                 (READ)
ADD R3, R1, R2           (MODIFY)
ST R3, c                 (WRITE)
```

Becomes one indivisible operation/instruction:

RED.E.ADD.F32.FTZ.RN [c], R2;

Facilitated by special hardware in the L2 cache

May have performance implications

atomicAdd implementation

```
// Note that any atomic operation can be implemented based
on atomicCAS() (Compare And Swap). For example…….
  #if __CUDA_ARCH__ < 600
  __device__ double atomicAdd(double* address, double val)
  {
      unsigned long long int* address_as_ull = (unsigned long long int*)address;
      unsigned long long int old = *address_as_ull, assumed;

      do {
          assumed = old;
          old = atomicCAS(address_as_ull, assumed,
                          __double_as_longlong(val + __longlong_as_double(assumed)));

      // Note: uses integer comparison to avoid hang in case of NaN (since NaN != NaN)
      } while (assumed != old);

      return __longlong_as_double(old);
  }
  #endif
```

atomicCAS(int* address, int compare, int val): Reads the 16-bit, 32-bit or 64-bit word old located at the address in global or shared memory, computes (old == compare ? val : old) , and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old (Compare And Swap).

OTHER ATOMICS

- atomicMax/Min – choose the max (or min)
- atomicAdd/Sub – add to (or subtract from)
- atomicInc/Dec – increment (or decrement) and account for rollover/underflow
- atomicExch/CAS – swap values, or conditionally swap values
- atomicAnd/Or/Xor – bitwise ops
- atomics have different datatypes they can work on (e.g. int, unsigned, float, etc.)
- https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions
- **Reference textbook "The Art of Multiprocessor Programming" by MIT**

ATOMIC TIPS AND TRICKS

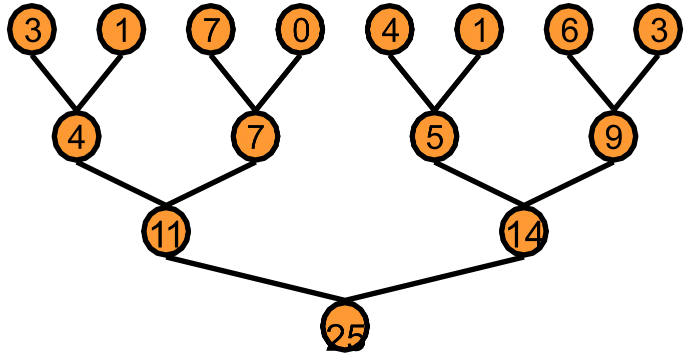
Determine my place in an order

- Could be used to determine next work item, queue slot, etc.
- int my_position = atomicAdd(order, 1);
- Most atomics return a value that is the “old” value that was in the location receiving the atomic update.

CLASSICAL
PARALLEL REDUCTION

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 streaming multiprocessors (SIMD processors) on the GPU busy
- Each thread block reduces a portion of the array

But how do we communicate partial results between thread blocks?

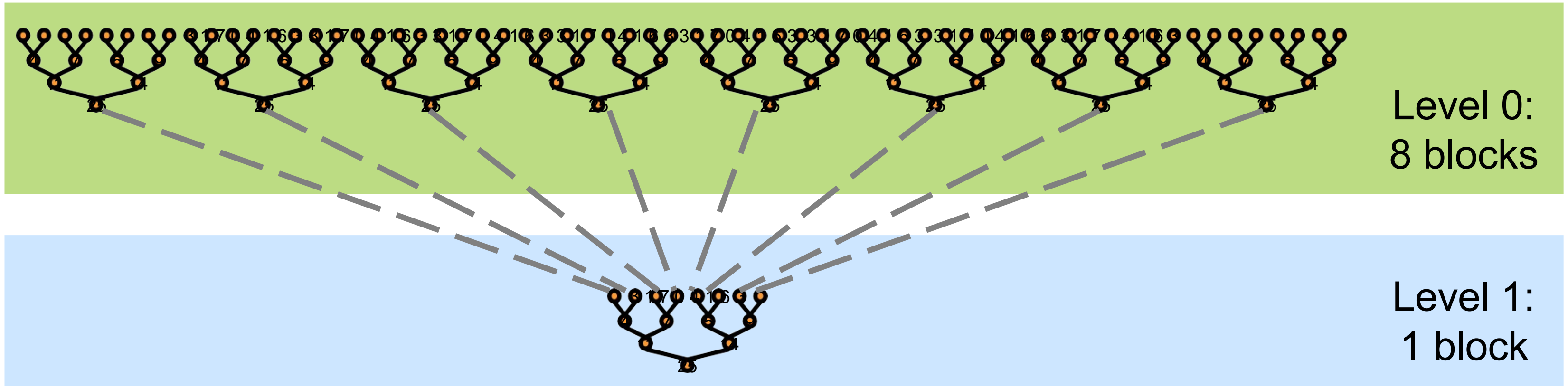
Problem: Global Synchronization

- 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 to avoid deadlock, which may reduce overall efficiency

- Solution: decompose into multiple kernels
  - Kernel launch serves as a global synchronization point
  - Kernel launch has HW overhead and 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?

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), limited by bandwidth

Therefore we should strive for peak bandwidth

Will use G80 GPU for this example

384-bit memory interface, 900 MHz DDR

384 * 800 * 2 / 8 = 86.4 GB/s

V100 GPU memory bandwidth 900 GB/s

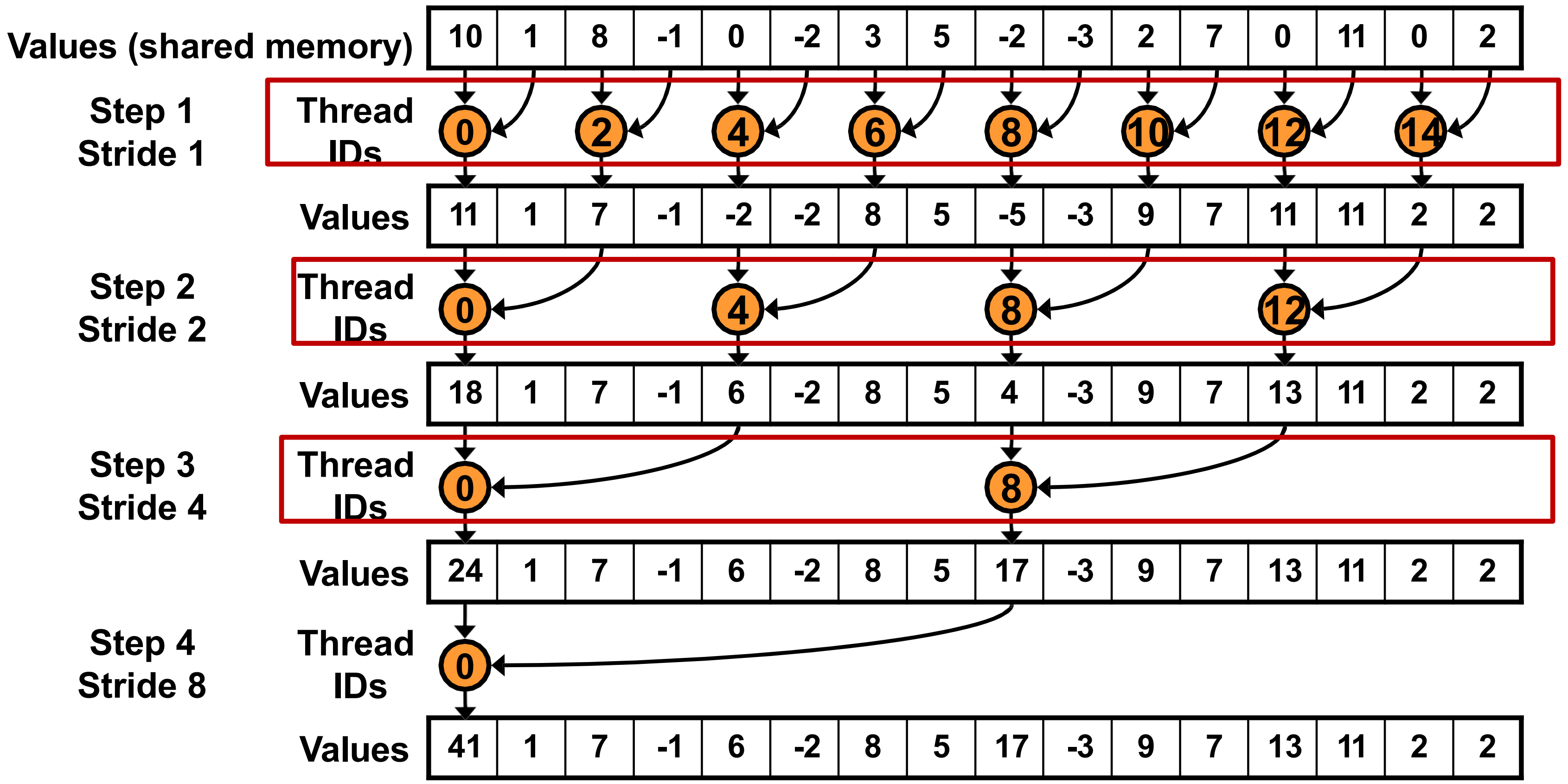
Reduction #0: Interleaved Addressing

```
__global__ void reduce0(int *g_idata, int *g_odata) {

// g_odata's size is the number of blocks,
//g_idata's size <= number of CUDA threads in grid.
    __shared__ int sdata[BLOCK_SIZE];
    // 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

Warp 0 Thread ID: 0, 2 , 4, 6, ……, 32, participate computing



Reduction #1: Interleaved Addressing

```
__global__ void reduce1(int *g_idata, int *g_odata)
{  __shared__ int sdata[BLOCK_SIZE];

   // 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 warp divergent branch in loop:

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

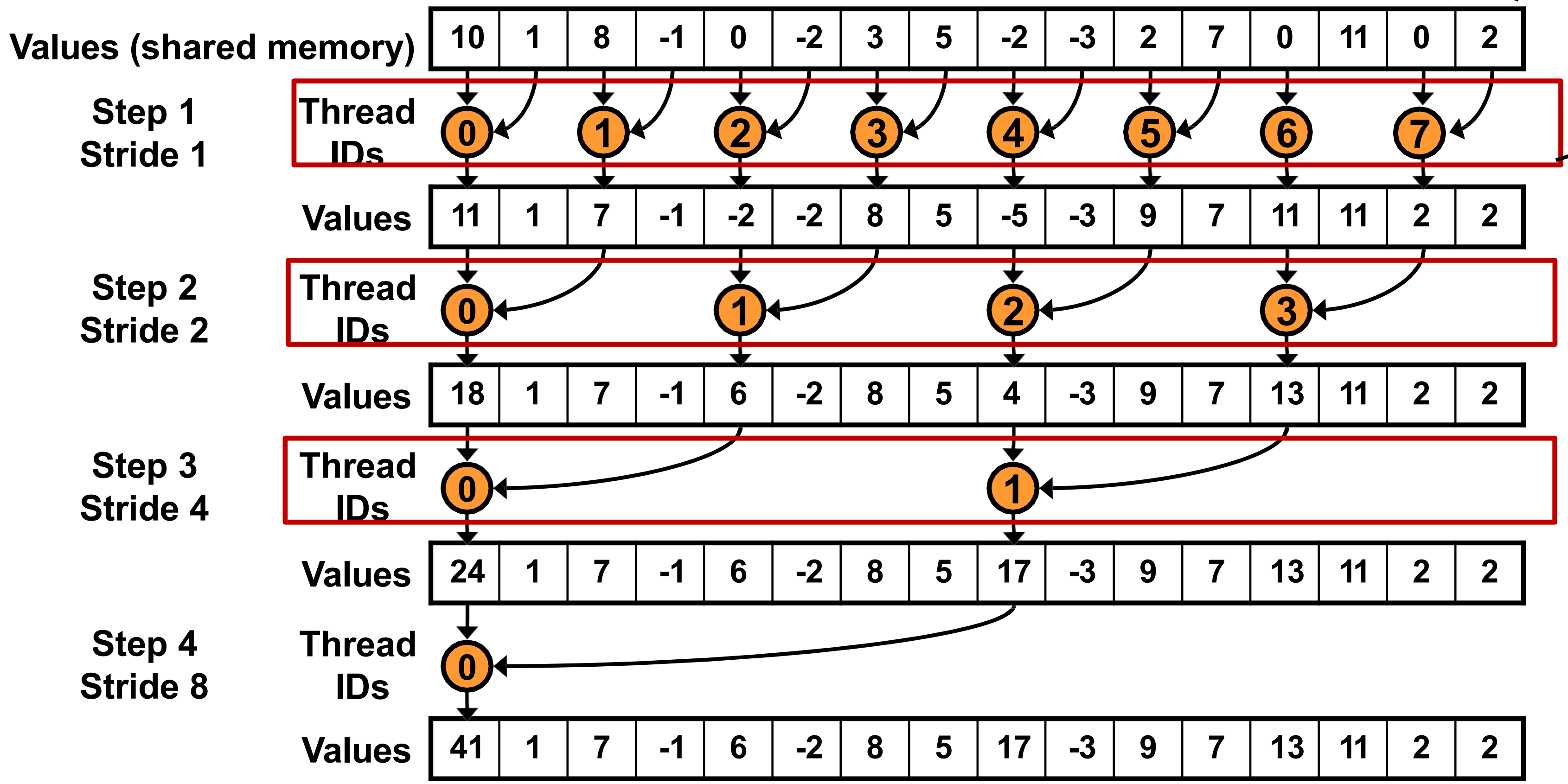
With strided index and warp non-divergent branch:

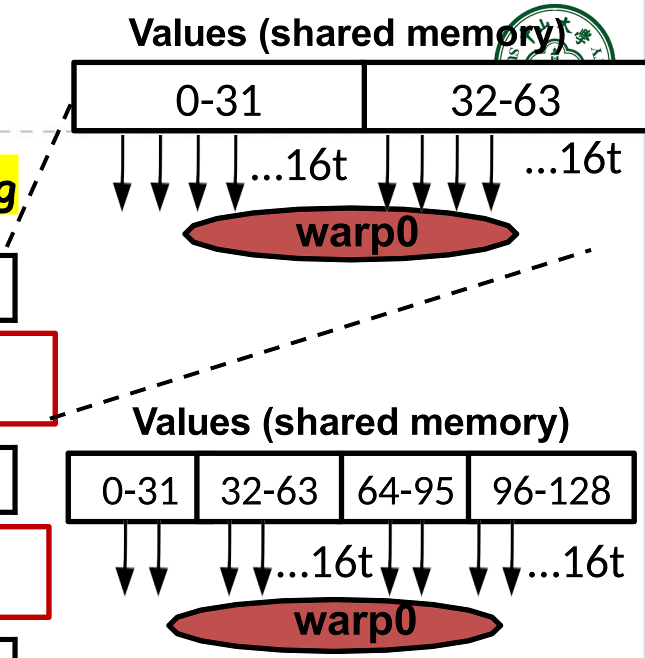
```
for (unsigned int s=1; s < blockDim.x; s *= 2)        {
    int index = 2 * s * tid;

    if (index < blockDim.x) {
    // warps are non-divergent, either a warp of 32
    // threads doing if code block, or skipping if.
        sdata[index] += sdata[index + s];
    }
    .       syncthreads();
}
```

Parallel Reduction: Interleaved Addressing

Warp 0 Thread ID: 0, 1, 2, 3, 4, 5, 6, ……, 31, participate computing





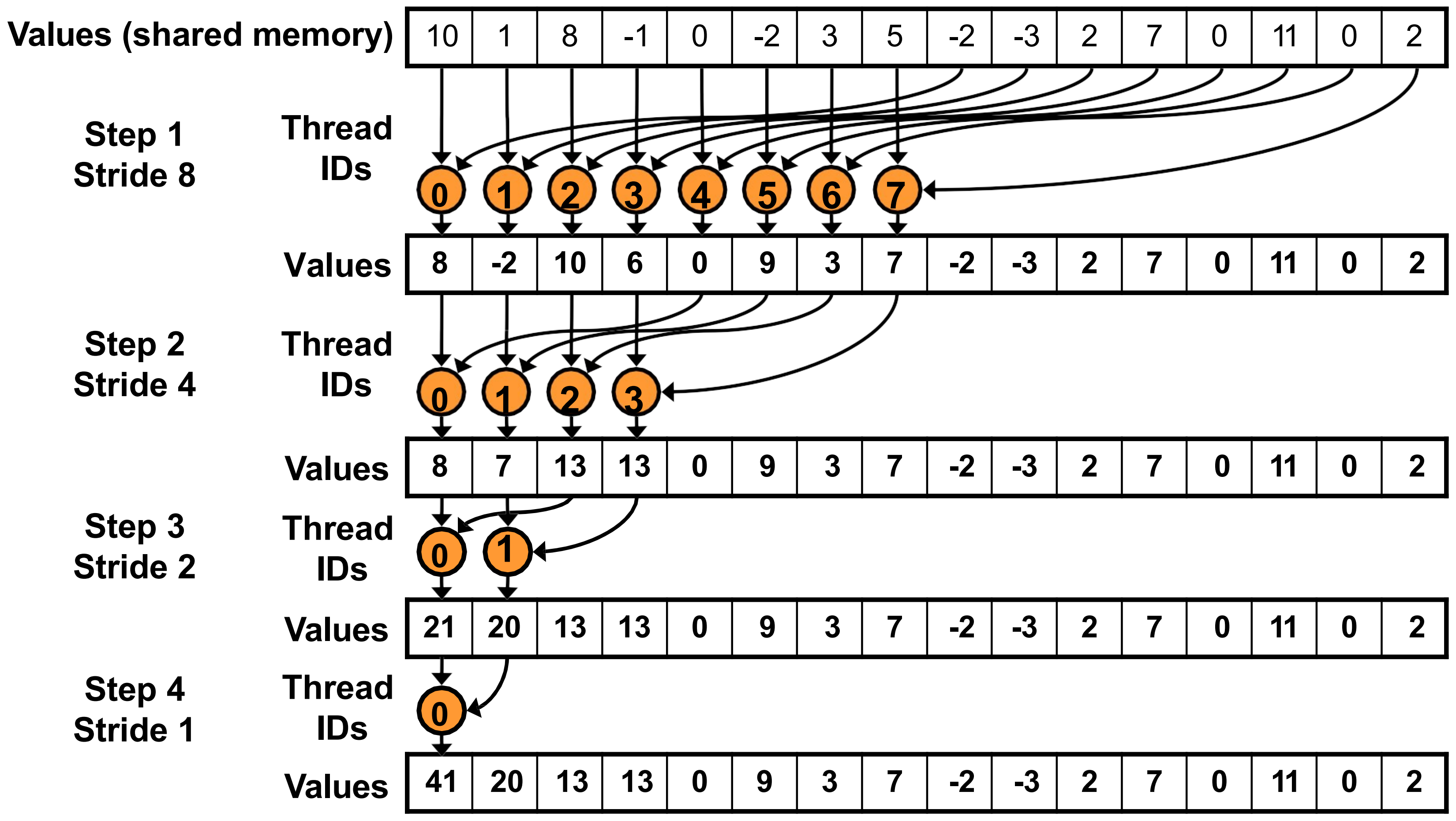
……

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)
                  {
// s represents stride
  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) {
// s represents stride
  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 1024 threads (Tesla V100)
  - 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];
      iif((bblloocckSSiizzee >>== 186)) ssddaattaa[[ttiidd]] ++== ssddaattaa[[ttiidd ++  4];
        if   (blockSize   4) sdata[tid] += sdata[tid +  2];
      8];>=             if   2) sdata[tid] += sdata[tid +  1];
 }    (blockSize >=
```

```
  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

Reduction #7: More Grids

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 in a CUDA thread:

```
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: More Grids

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 in a CUDA thread:

```
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();
```

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!

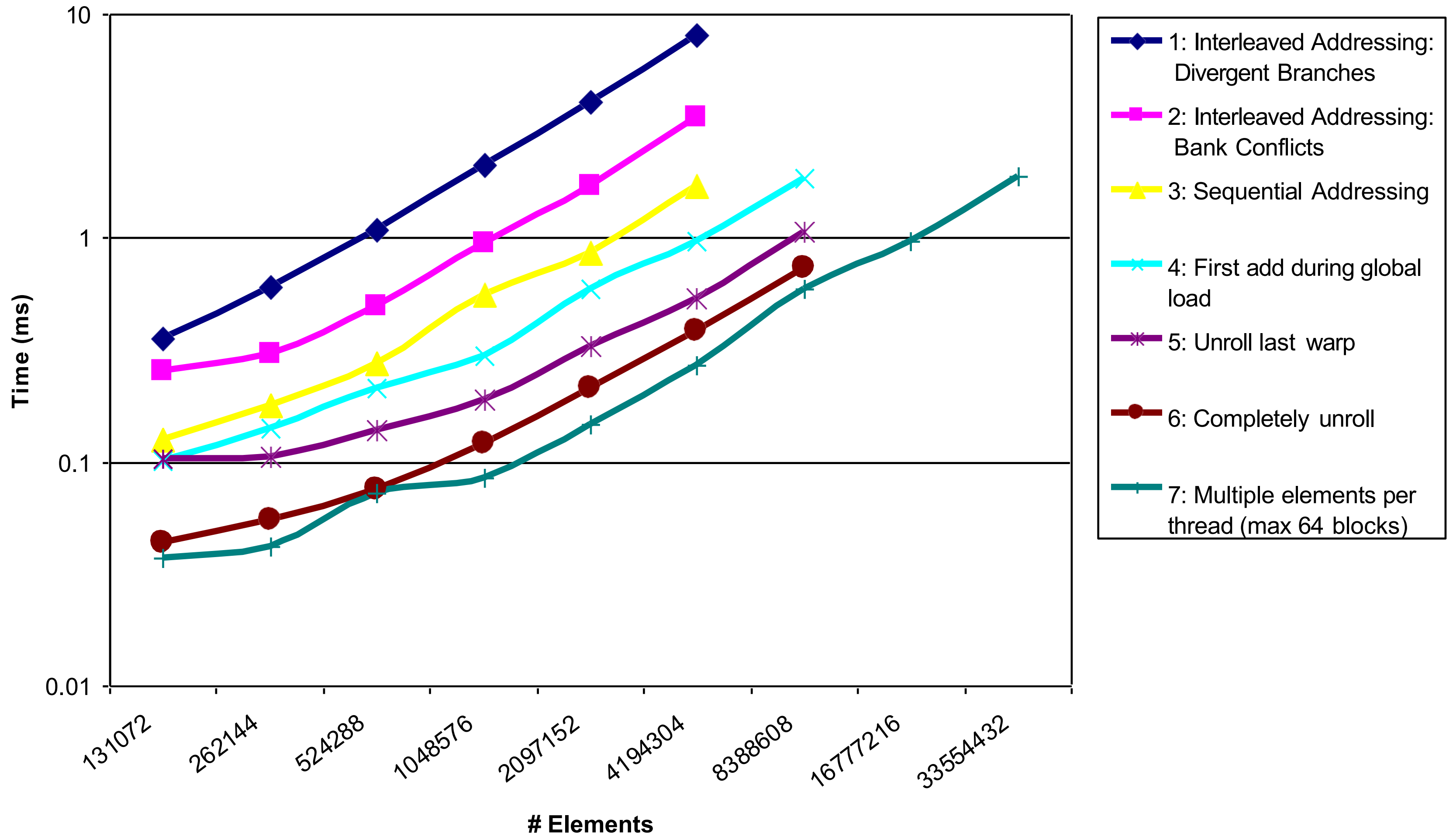
Final Optimized Kernel

```
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 2) sdata[tid] += sdata[tid +  1];
}   (blockSize >=
template <unsigned int blockSize>
    global      void reduce6(int *g_idata, int *g_odata, unsigned int n)
    {  __shared__ int sdata[BLOCK_SIZE];
    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

FUTURE SESSIONS

- Using Managed Memory
- Concurrency (streams, copy/compute overlap, multi-GPU)
- Analysis Driven Optimization
- Cooperative Groups

FURTHER STUDY

- Parallel reduction:
  - https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf
- Warp-shuffle and reduction:
  - https://devblogs.nvidia.com/faster-parallel-reductions-kepler/
- CUDA Cooperative Groups:
  - https://devblogs.nvidia.com/cooperative-groups/
- Grid-stride loops:
  - https://devblogs.nvidia.com/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/
- Floating point:
  - https://developer.nvidia.com/sites/default/files/akamai/cuda/files/NVIDIA-CUDA-Floating- Point.pdf
- CUDA Sample Codes:
  - Reduction, threadFenceReduction, reductionMultiBlockCG