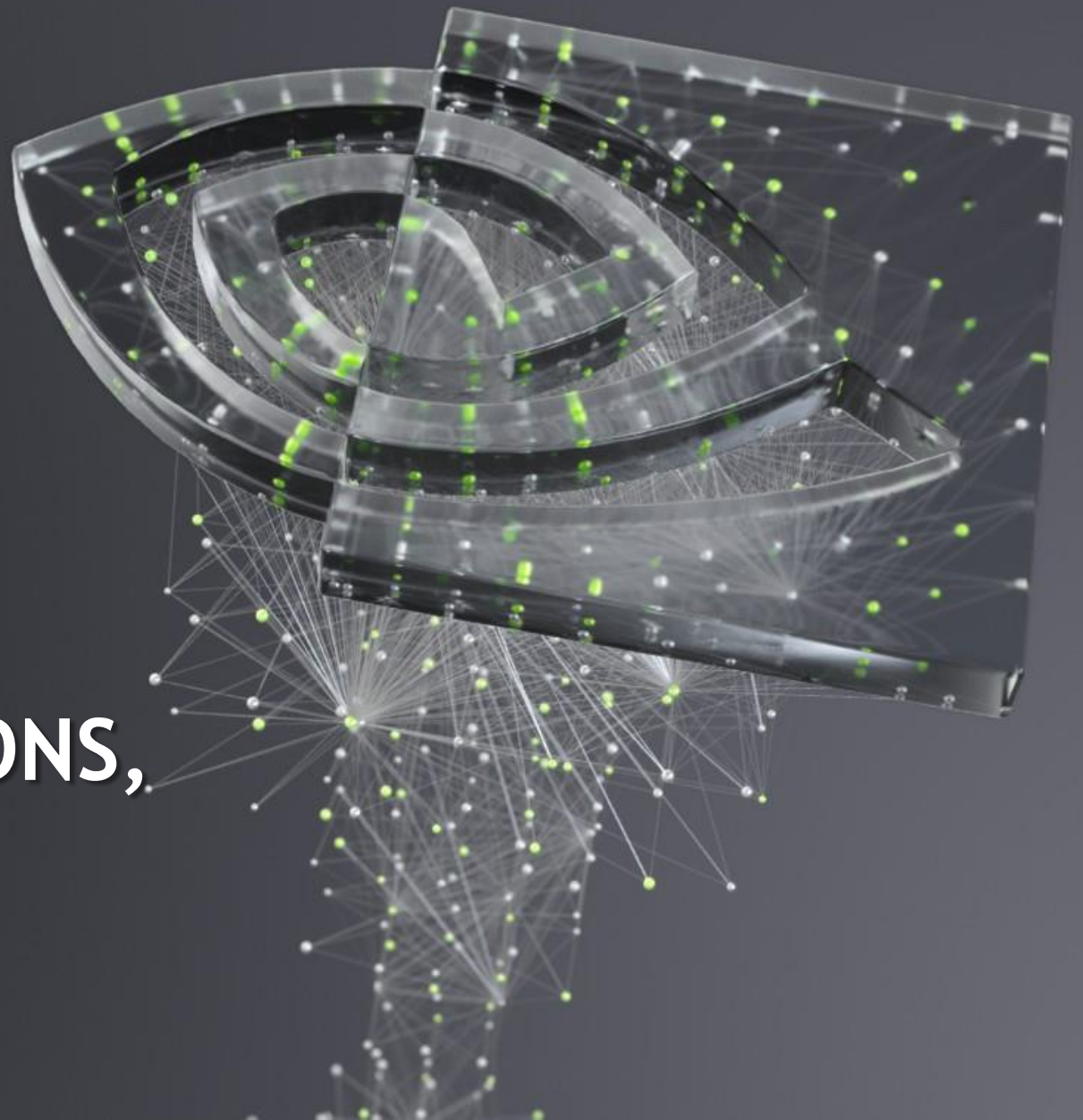


ATOMICS, REDUCTIONS, WARP SHUFFLE

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

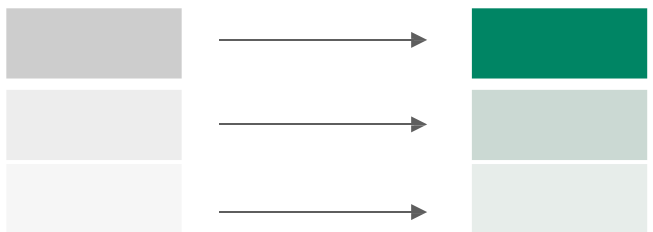
Sum - reduction

```
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

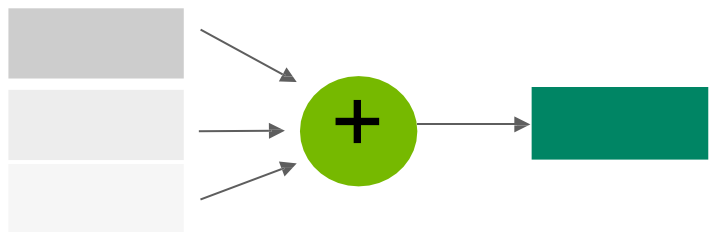
May guide the **thread strategy**: what will each thread do?



Transformation:

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

Thread strategy: one thread per output point



Reduction:

e.g. *c = Σ a[i]

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 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] (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

```
#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

ATOMIC TIPS AND TRICKS

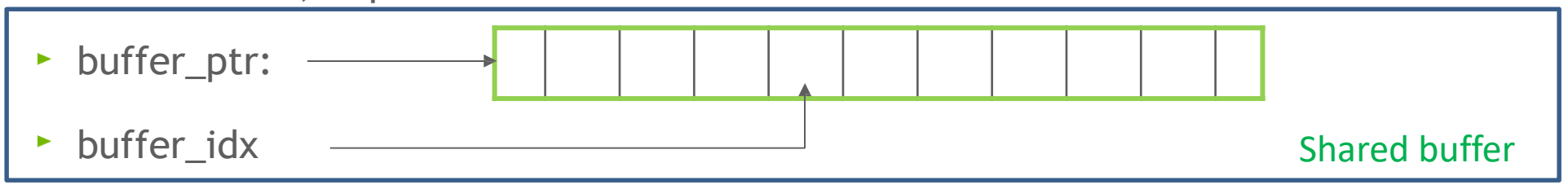
Determine my place in an order

- 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.

ATOMIC TIPS AND TRICKS

Reserve space in a buffer

- Each thread in my kernel may produce a variable amount of data. How to collect all of this in one buffer, in parallel?

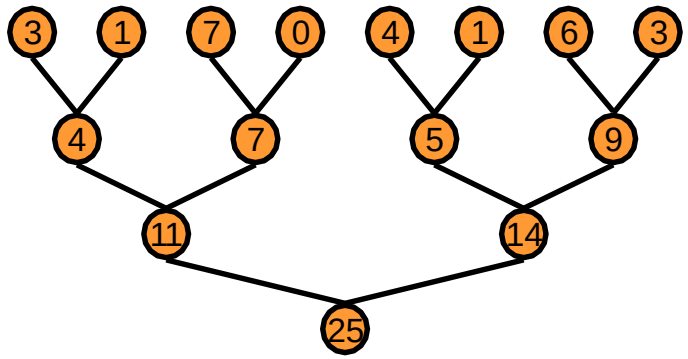


- int my_dsize = var;
- float local_buffer[my_dsize] = {…};
- int my_offset = atomicAdd(buffer_idx, my_dsize);
- // buffer_ptr+my_offset now points to the first reserved location, of length my_dsize
- memcpy(buffer_ptr+my_offset, local_buffer, my_dsize*sizeof(float));

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 multiprocessors on the GPU busy
- Each thread block reduces a portion of the array

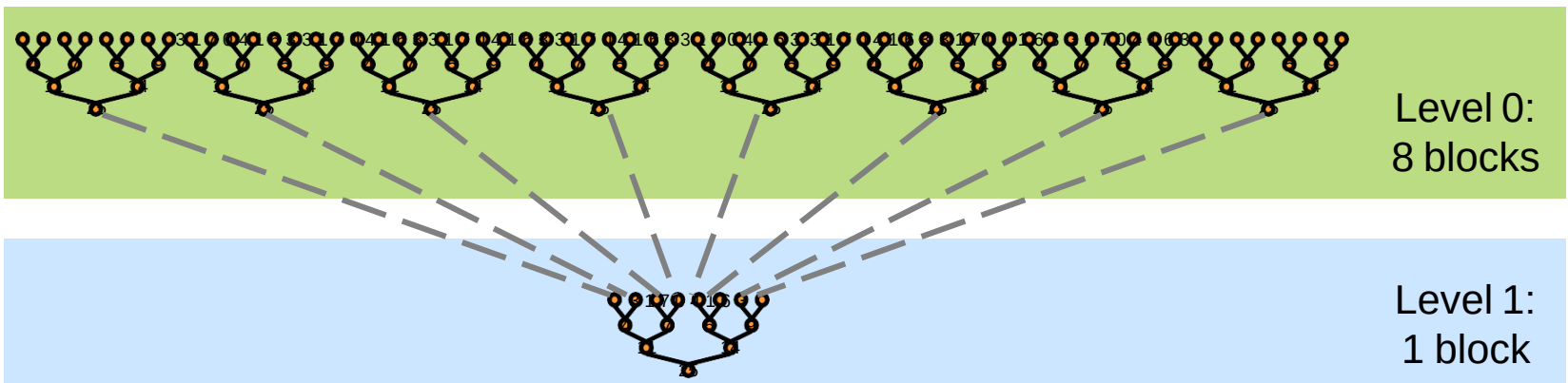
But how do we communicate partial results between thread blocks?

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

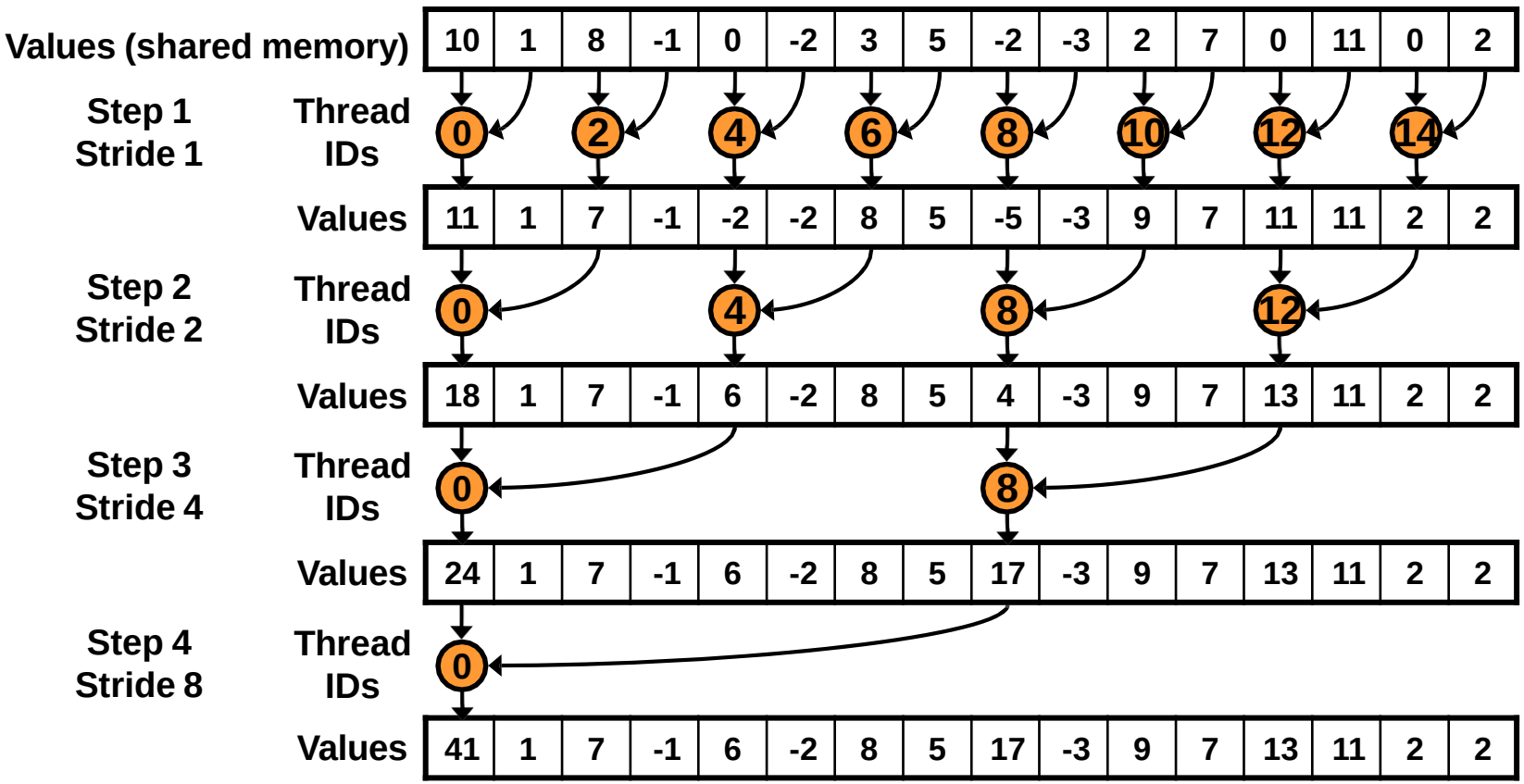
384 * 1800 / 8 = 86.4 GB/s

V100 GPU memory bandwidth 900 GB/s

Reduction #1: 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



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

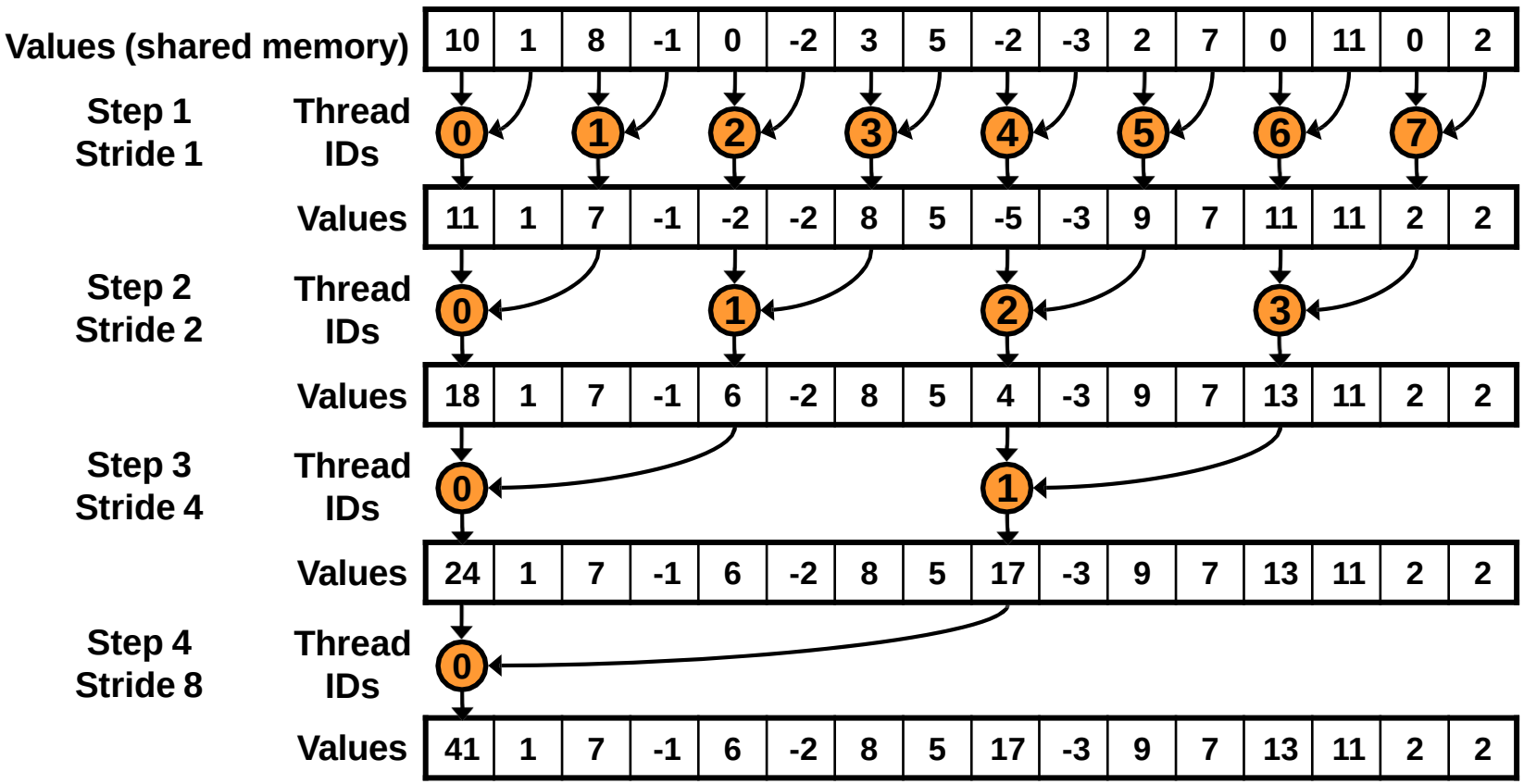
Just replace divergent branch in inner loop:

```
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

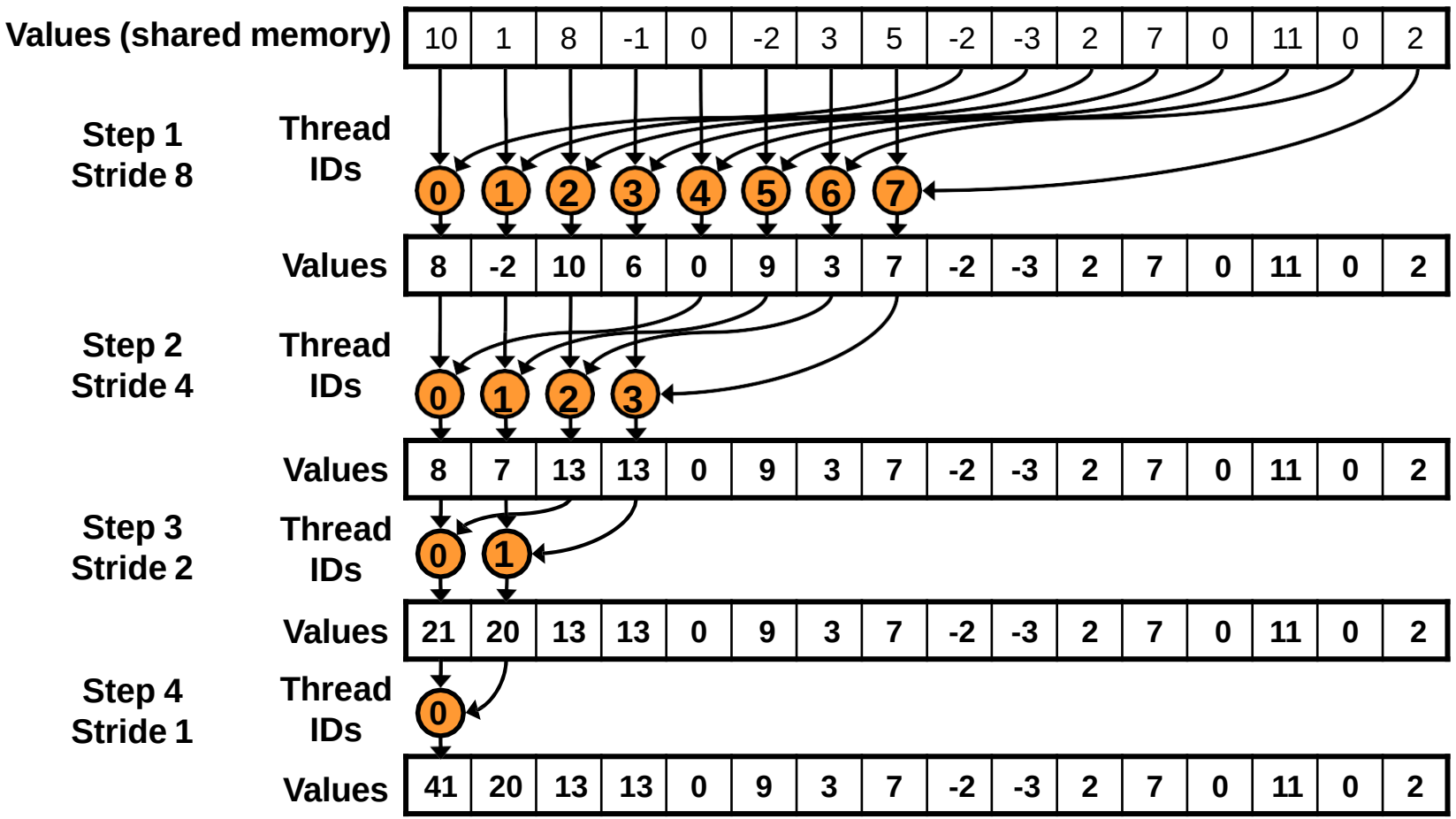


New Problem: Shared Memory Bank Conflicts

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)  {
// 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
- 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 = 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

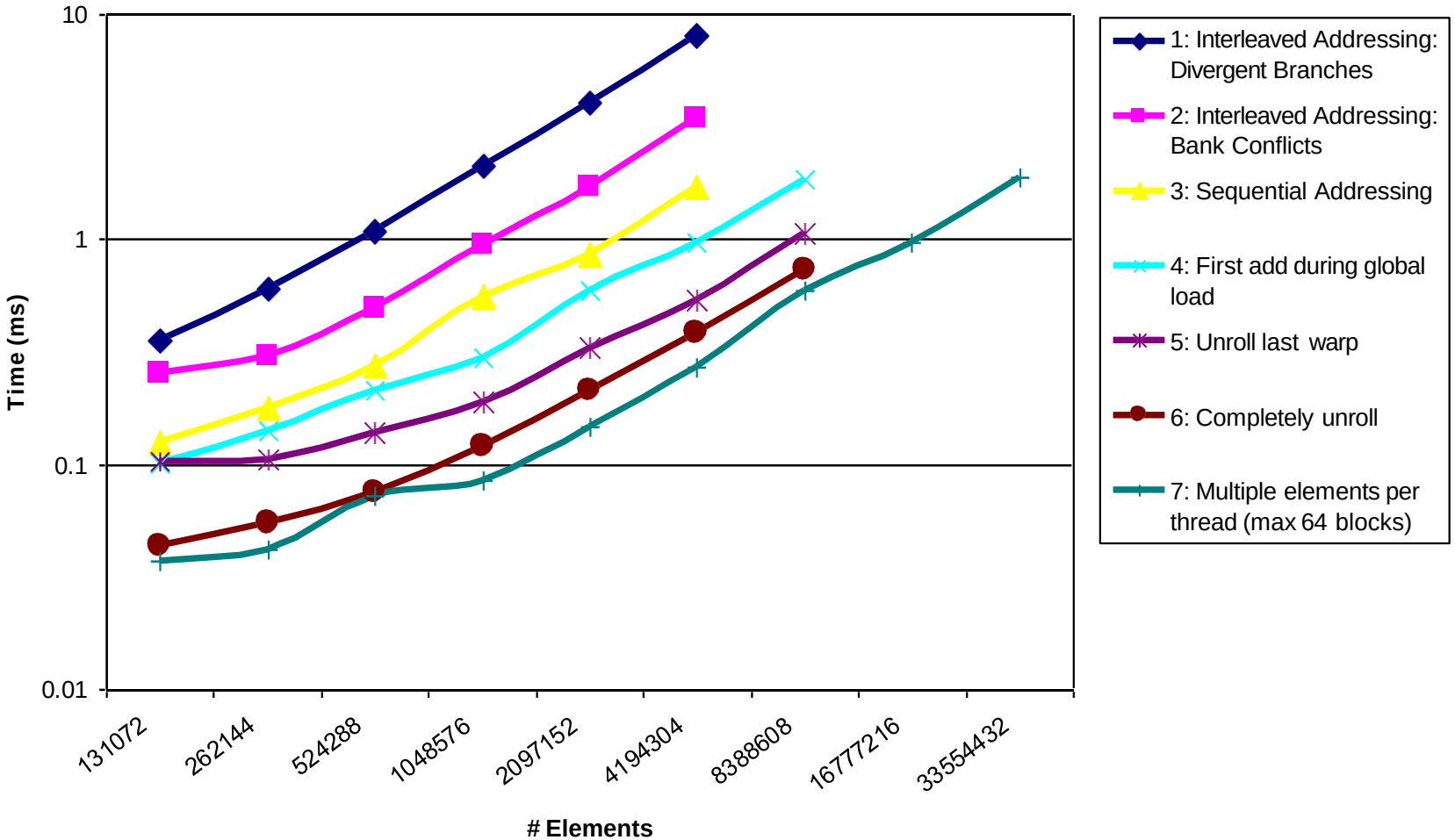
	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first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

```
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) {
    __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];
}
```

Final Optimized Kernel

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

REDUCE VERSION 8

How CUDA Reduce can process arbitrary data size?

- We'd like to be able to design kernels that load and operate on arbitrary data sizes efficiently
- Want to maintain coalesced loads/stores, efficient use of shared memory
- Choose number of blocks sized to the GPU
- **gdata**[0..N-1]:


```
int idx = threadIdx.x+blockDim.x*blockIdx.x;
while (idx < N) {
  sdata[tid] += gdata[idx];
  idx += gridDim.x*blockDim.x; // grid width
  }
```

REDUCE VERSION 8

```
__global__void reduce(float *gdata, float *out){
  __shared__float sdata[NUM_BLOCK];
  int tid = threadIdx.x;
  sdata[tid] = 0.0f;
  size_t idx = threadIdx.x+blockDim.x*blockIdx.x;

  while (idx < N) { // grid stride loop to load data
    sdata[tid] += gdata[idx];
    idx += gridDim.x*blockDim.x;
    }

  for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    __syncthreads();
    if (tid < s) // parallel sweep reduction
      sdata[tid] += sdata[tid + s];
    }
  if (tid == 0) out[blockIdx.x] = sdata[0];
 }
```

This version of Reduce puts the partial sum for each block in a different entry in a global array (sdata). These partial sums (sdata) can be transferred back to the host for the final summation

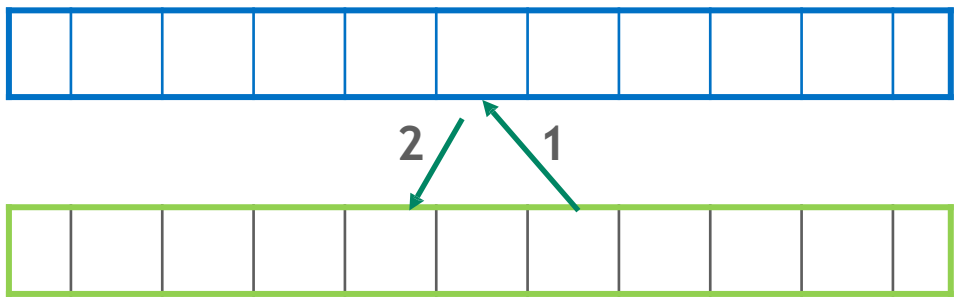
REDUCE VERSION 9

```
__global__void reduce_a(float *gdata, float *out){
   __shared__float sdata[NUM_ BLOCK];
   int tid = threadIdx.x;
   sdata[tid] = 0.0f;
   size_t idx = threadIdx.x+blockDim.x*blockIdx.x;

   while (idx < N) { // grid stride loop to load data
      sdata[tid] += gdata[idx];
      idx += gridDim.x*blockDim.x;
      }

   for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
      __syncthreads();
      if (tid < s) // parallel sweep reduction
          sdata[tid] += sdata[tid + s];
      }
   if (tid == 0) atomicAdd(out, sdata[0]);
}
```

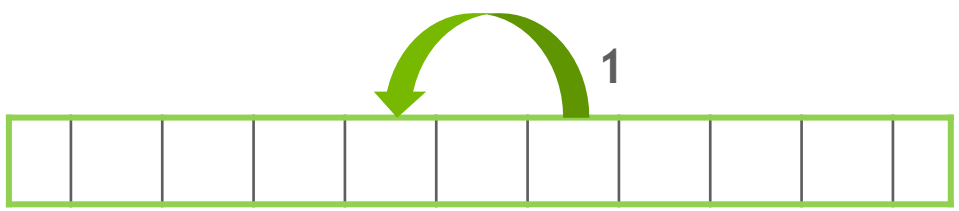
WARP SHUFFLE

INTER-THREAD COMMUNICATION: SO FAR

- Using **shared memory:**
- **Threads:**

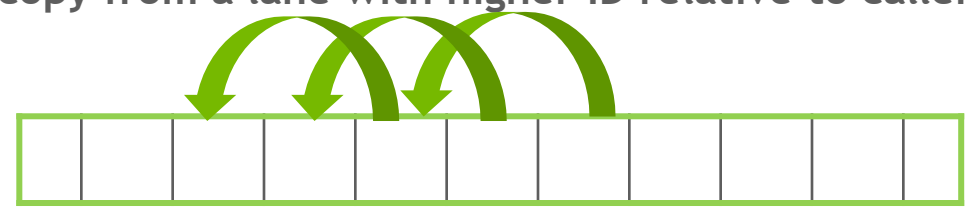


- Wouldn't this be convenient:
- **Threads:**



INTRODUCING WARP SHUFFLE

- Allows for intra-warp communication
- Various supported movement patterns:
  - **__shfl_sync(): copy from lane ID (arbitrary pattern)**
  - **__shfl_xor_sync(): copy from calculated lane ID (calculated pattern)**
  - **__shfl_up_sync(): copy from delta/offset lower lane**
  - **__shfl_down_sync(): copy from a lane with higher ID relative to caller:**



- Both source and destination threads in the warp must “participate”
- Sync “mask” used to identify and reconverge needed threads

Warp shuffles

```
__shfl_up_sync(unsigned mask, T var,
unsigned int delta);
```

`mask` controls which threads are involved — usually set to `-1` or `0xffffffff`, equivalent to all 1's

`var` is a local register variable (int, unsigned int, long long, unsigned long long, float or double)

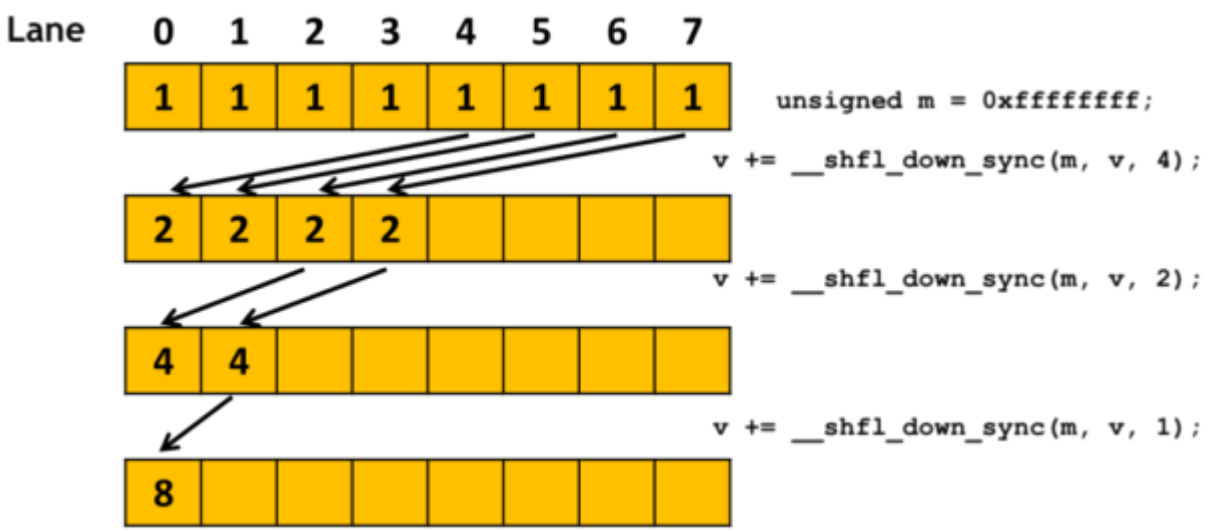
`delta` is the offset within the warp – if the appropriate thread does not exist (i.e. it's off the end of the warp) then the value is taken from the current thread

```
_shfl_down_sync(unsigned mask,    T var,
unsigned int delta);
```

defined similarly

Warp shuffles

```
#define FULL_MASK 0xffffffff
for (int offset = 16; offset > 0; offset /= 2)
    val += __shfl_down_sync(FULL_MASK, val, offset);
```



A warp comprises 32 lanes, with each CUDA thread occupying one lane. For a thread at lane X in the warp, __shfl_down_sync(FULL_MASK, val, offset) gets the value of the val variable from the CUDA thread at lane X+offset of the same warp. The data exchange is performed between **registers**, and more efficient than going through shared memory, which requires a load, a store and an extra register to hold the address.

WARP SHUFFLE REDUCTION

```
__global__void reduce_ws(float *gdata, float *out){
  __shared__float sdata[32];
  int tid = threadIdx.x;
  int idx = threadIdx.x+blockDim.x*blockIdx.x;
  float val = 0.0f;
  unsigned mask = 0xFFFFFFFFU;
  int lane = threadIdx.x % warpSize;
  int warpID = threadIdx.x / warpSize;
  while (idx < N) { // grid stride loop to load
    val += gdata[idx];
    idx += gridDim.x*blockDim.x;
    }

// 1st warp-shuffle reduction
  for (int offset = warpSize/2; offset > 0; offset >>= 1)
    val += __shfl_down_sync(mask, val, offset);
  if (lane == 0) sdata[warpID] = val;
 __syncthreads(); // put warp results in shared mem

// hereafter, just warp 0
  if (warpID == 0){
 // reload val from shared mem if warp existed
    val = (tid < blockDim.x/warpSize)?sdata[lane]:0;

 // final warp-shuffle reduction
    for (int offset = warpSize/2; offset > 0; offset >>= 1)
      val += __shfl_down_sync(mask, val, offset);

    if (tid == 0) atomicAdd(out, val);
  }
 }
```

WARP SHUFFLE BENEFITS

- Reduce or eliminate shared memory usage
- Single instruction vs. 2 or more instructions
- Reduce level of explicit synchronization

WARP SHUFFLE TIPS AND TRICKS

What else can we do with it?

- Broadcast a value to all threads in the warp in a single instruction
- Perform a warp-level prefix sum
- Atomic aggregation

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