Implementing Reductions I

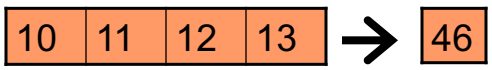
Nikos Hardavellas

Some slides/material from:
UToronto course by Andreas Moshovos
UIUC course by Wen-Mei Hwu and David Kirk
UCSB course by Andrea Di Blas
Universitat Jena by Waqar Saleem
NVIDIA by Simon Green, Mark Harris, and many others
Optimizations studied on G80 hardware

1

Reduction Operations

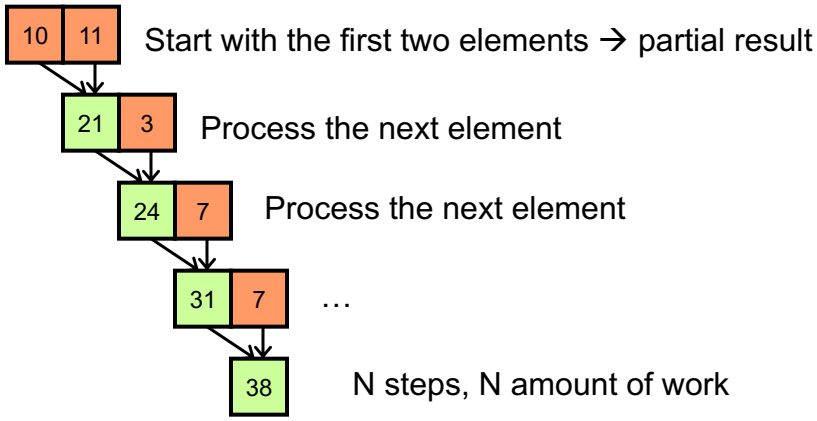
- Multiple values are reduced into a single value
  - ADD, MUL, AND, OR, ….



- Parallel reduction is a common and important data-parallel primitive
- Easy enough to implement in CUDA
  - But hard to get it right
  - Allows us to focus on optimization techniques
  - We'll walk through 7 different versions
  - Demonstrates several important optimization strategies

2

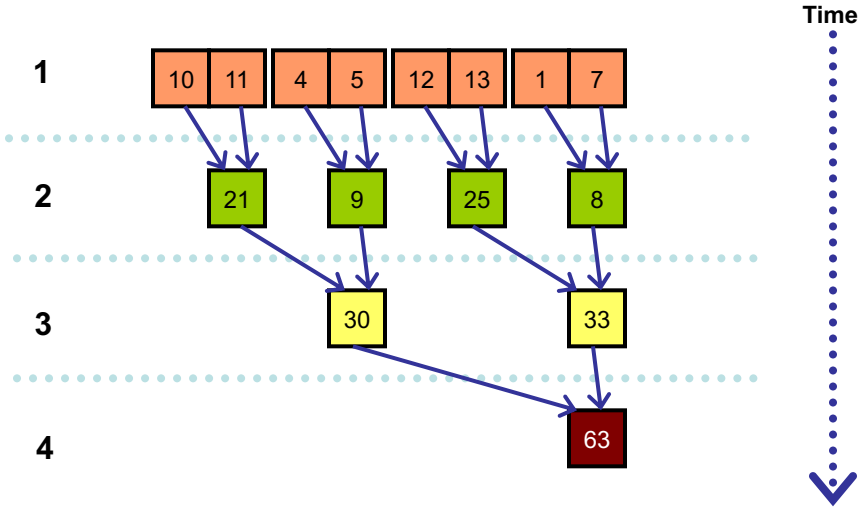
Sequential Reduction



3

Parallel Reduction

Pair-wise reduction in steps – Tree-like

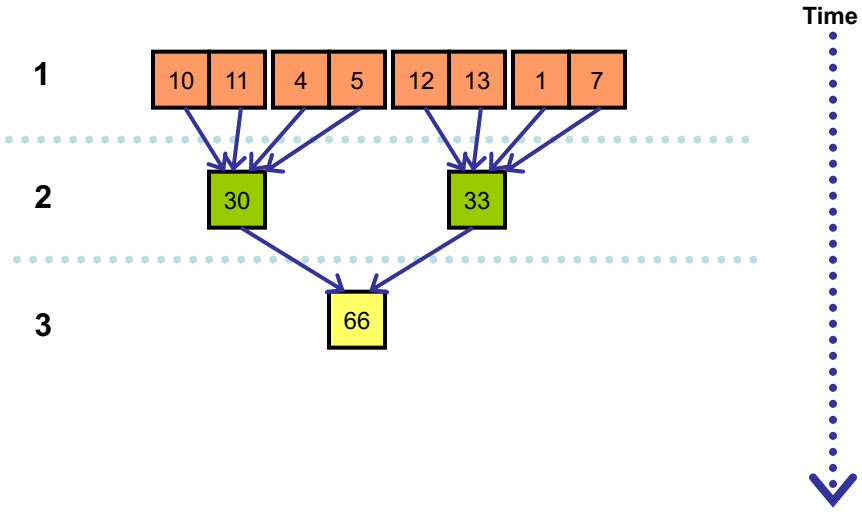


$\log_2 N$ steps, N amount of work

4

Parallel Reduction – Different Degree-Trees possible

Pair-wise reduction in steps – Tree-like



$\log_4 N$ steps, N amount of work

5

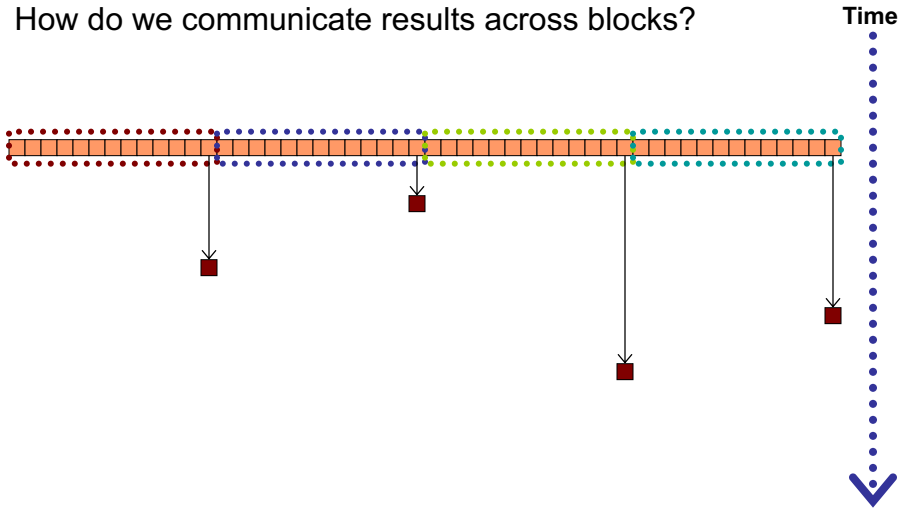
CUDA Strategy

- Single Block
  - Use Tree-Like approach
  - One thread can always process many elements
  - But, may suffer from low utilization
- Multiple Blocks
  - Use Tree-Like approach as well
  - Each block processes a portion of the input
  - Utilize GPU resources
  - Useful for large arrays

6

How about multiple blocks

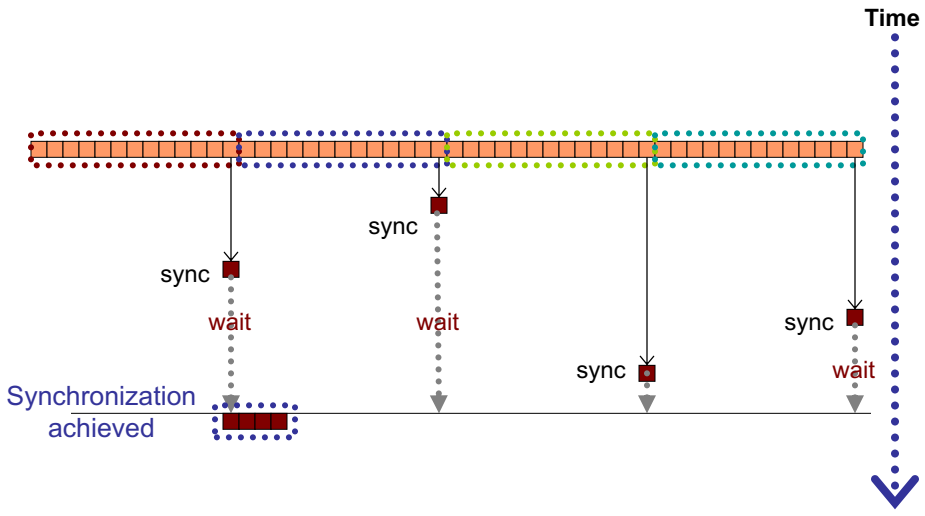
- How do we communicate results across blocks?



- The key problem is synchronization:
  - How do we know that each block has finished?

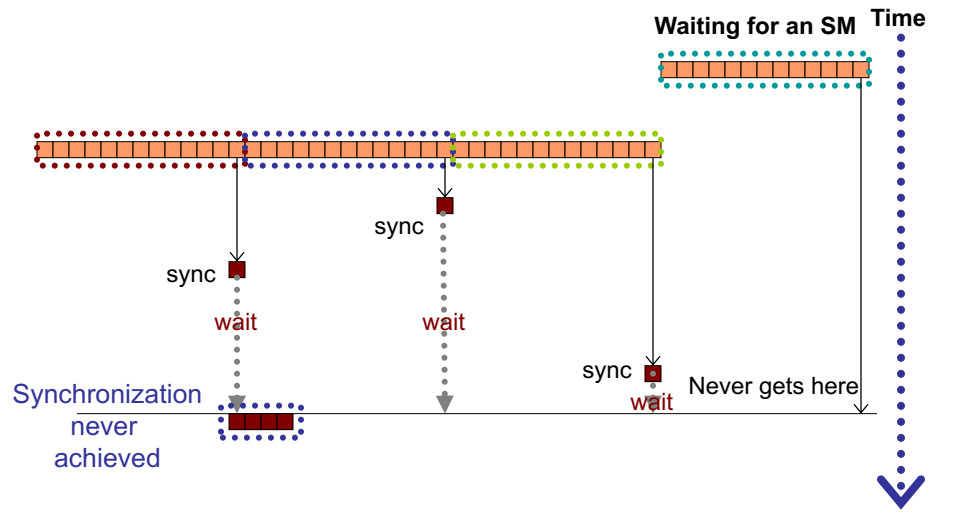
7

Global Synchronization



8

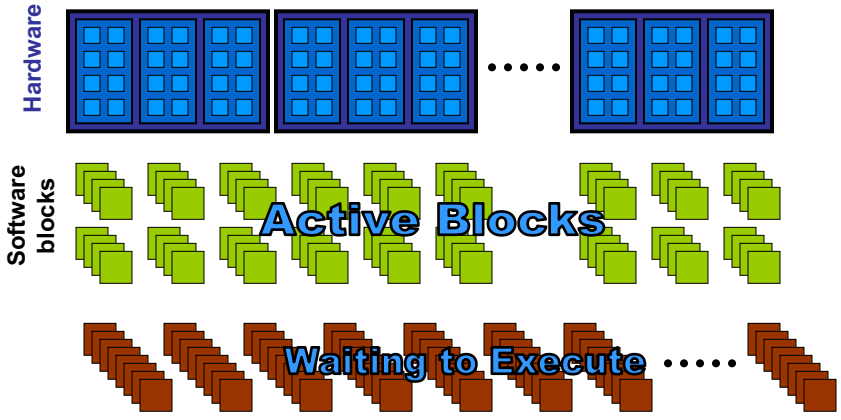
Deadlock



9

The Problem with Global Synchronization

- CUDA does not support it
  - One reason:
    - It's expensive to implement
  - Another reason:
    - "choose" between **limited blocks** or **deadlock**



10

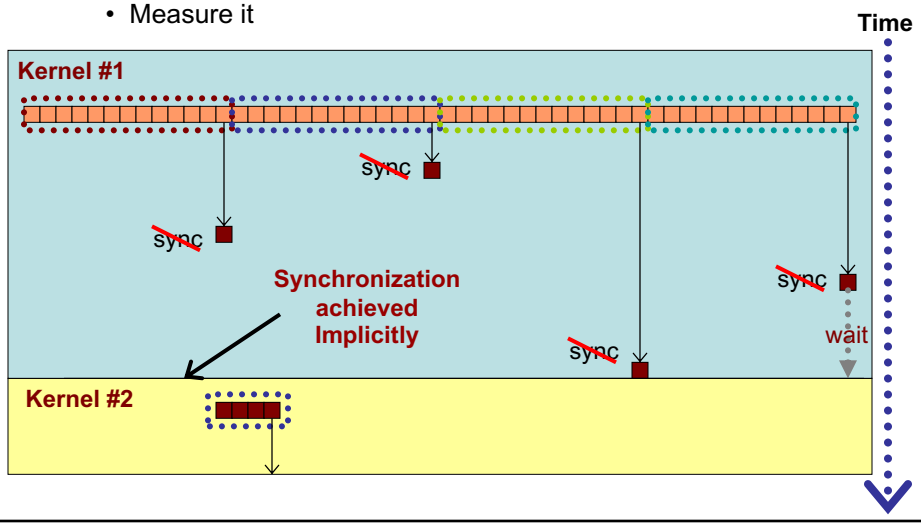
The Problem with Global Synchronization / Summary

- If there was global sync
  - Global sync after each block
  - Once all blocks are done, continue recursively
- CUDA does not have explicit global sync:
  - Expensive to support
  - Would limit the number of blocks
  - Otherwise deadlock will occur
    - Once a block gets assigned to an SM it stays there
    - Each SM can take only up to 16 blocks, 8 SMs in GTX680
    - At most 8 x 16 = 128 blocks could be active at any point of time
- Solution: Decompose into multiple kernels

11

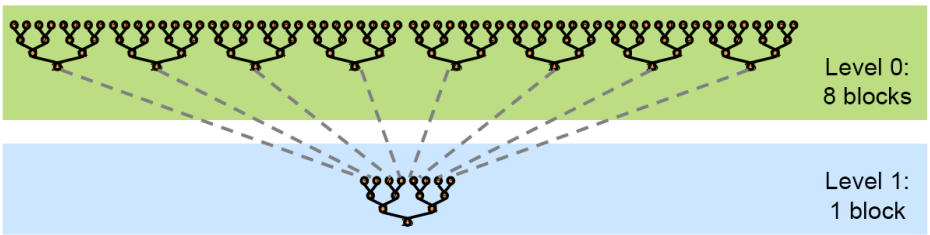
Decomposing into Multiple Kernels

- Implicit Synchronization between kernel invocations
  - Kernels need to launch within the same stream (or use events)
  - Overhead of launching a new kernel non-negligible
    - Don't know how much
    - Measure it



12

Reduction: Big Picture



- **The code for all levels is the same**
- **The same kernel code can be called multiple times**

13

Optimization Goal

- Get maximum GPU performance

- Two components:
  - Compute Bandwidth: GFLOPs
  - Memory Bandwidth: GB/s

14

Optimization Goals Cont.

- Reductions typically have low arithmetic intensity
  - FLOPs/element loaded from memory
- So, bandwidth will be the limiter

- For GTX280: **141.7 GB/s**
- For GTX680: **192.2 GB/s**

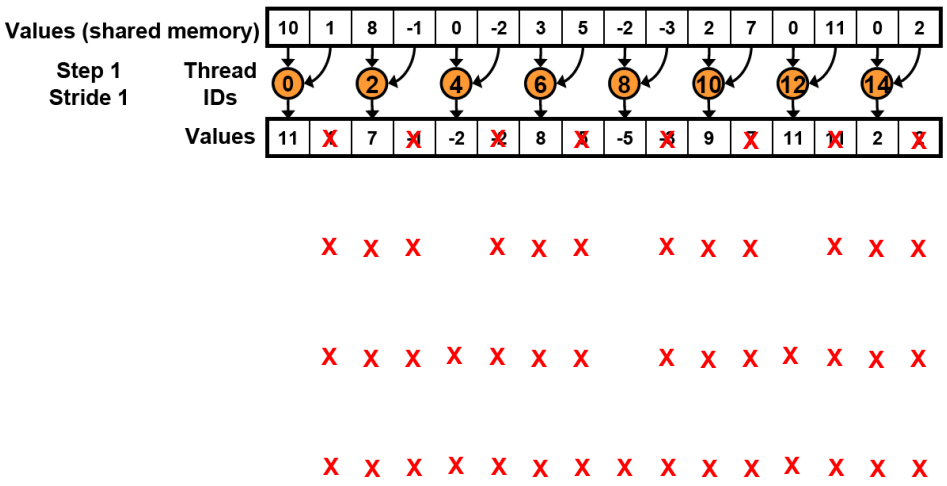
15

Reduction #1: Strategy

- Load data:
  - Each thread loads one element from global memory to shared memory
- Actual Reduction: Proceed in log*N* steps
  - A thread reduces two elements
    - The first two elements by the first thread
    - The next two by the next thread
    - And so, on
  - At the end of each step:
    - Deactivate half of the threads
  - Terminate: when one thread left
- Write back to global memory

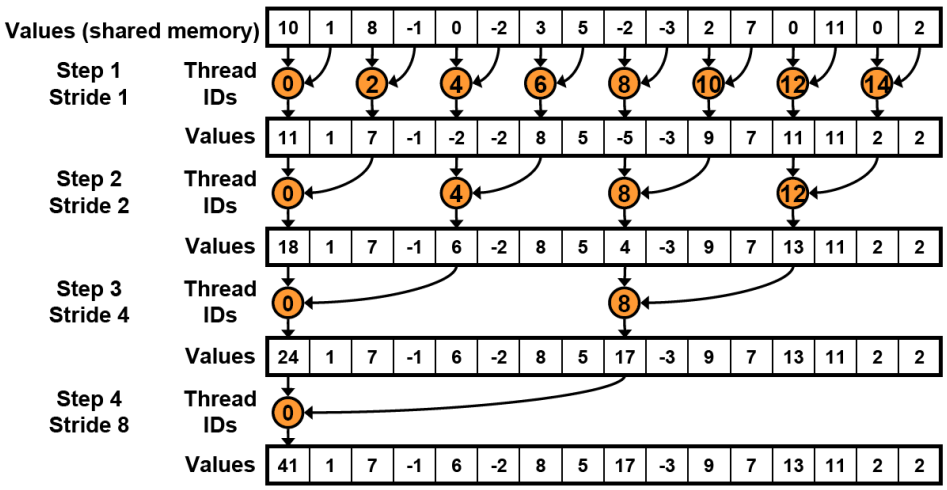
16

Reduction Steps



17

Reduction Steps



18

Reduction #1 Code: Interleaved Accesses

```
__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*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];
}
```

19

Performance for kernel #1 (target: 0.268 ms)

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

Note 1: Block size = 128 for all experiments

Note 2: results are for G80

Bandwidth calculation:
How many reads/writes per block?
Each block processes 128 elements and does:
128 reads & 1 write
We only care about global memory

Data reduction at each kernel/step:
N (element reads) + N / 128 (element writes)
Every kernel/step reduces input size by 128x
next set N = N / 128
So for:
N = 4194304 (4M floats)
Accesses = 4M+32K + 32K+256 + 256+2 + 2+1
Each access is four bytes

20

Reduction #1 Code: Interleaved Accesses

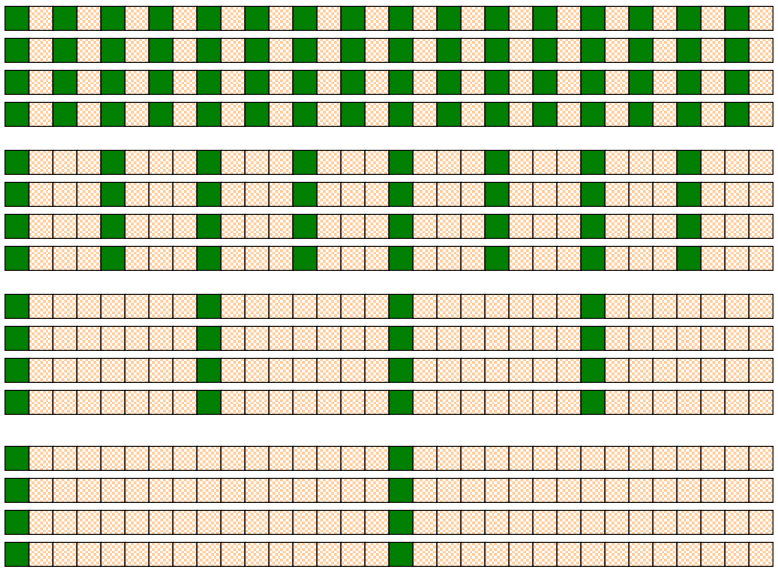
```
__global__ void reduce0(int *g_idata, int *g_odata) {
  extern __shared__ int sdata[];

  // 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*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 code leads to very poor performance

21

Divergent Branching: Warp Control Flow

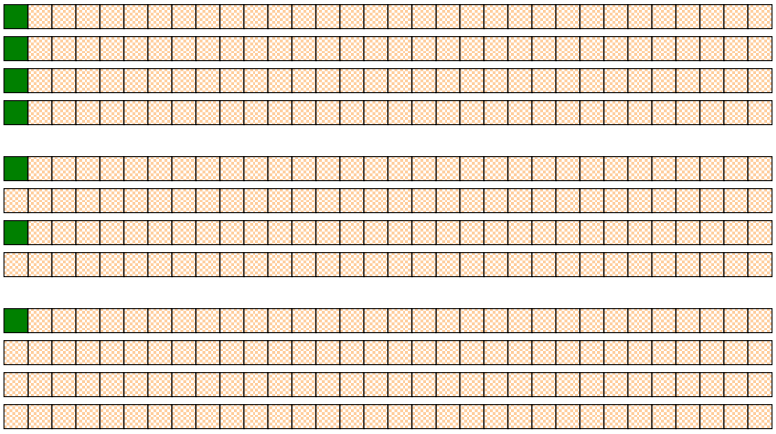
one square: data processed by one thread
green: active thread produces data; **yellow:** active thread produces no data



22

Divergent Branching: Warp Control Flow

one square: data processed by one thread
green: active thread produces data; **yellow:** active thread produces no data



23

Reduction #2: Interleaved Addr./non-divergent branching

- Replace the divergent branching code:

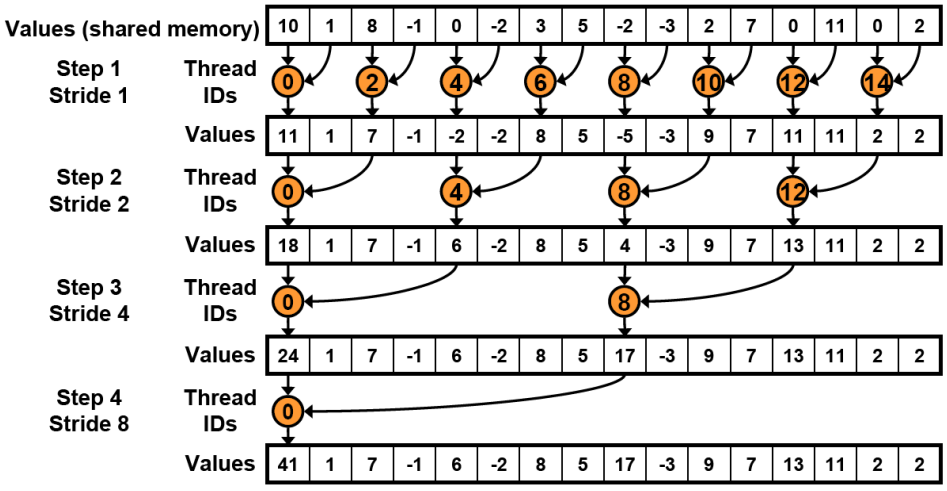
```
// do reduction in shared mem
for (unsigned int s=1; s < blockDim.x; s *= 2) {
     if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
     }
     __syncthreads();
}
```

- 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();
}
```

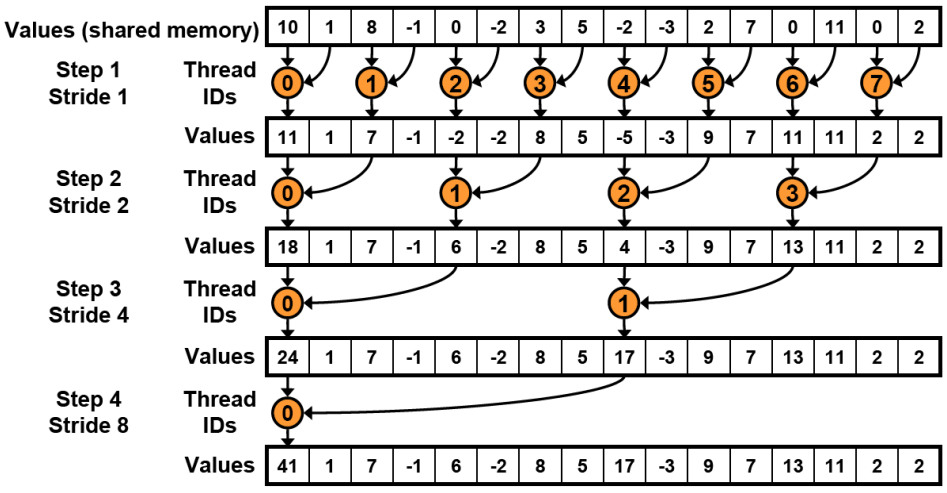
24

Reduction #1



25

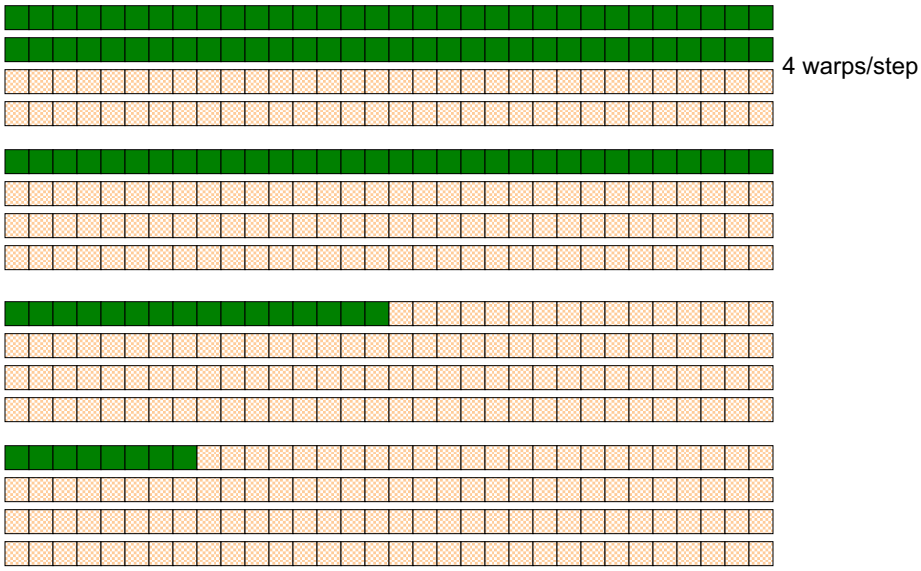
Reduction #2: Access Pattern



26

Reduction #2: Warp control flow

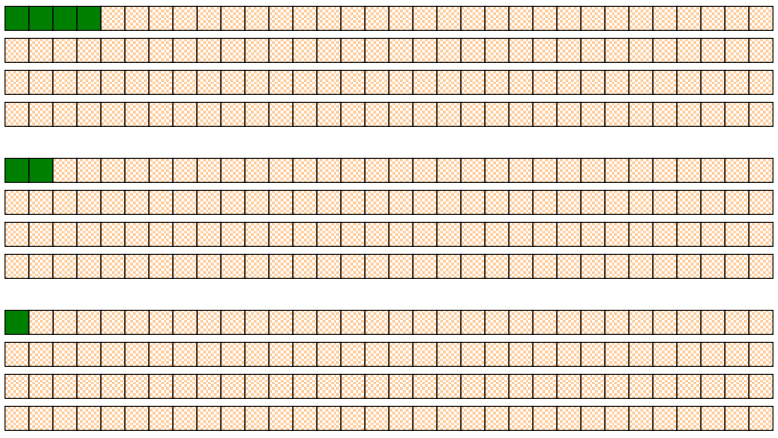
one square: data processed by one thread
green: active thread produces data; **yellow:** active thread produces no data



27

Reduction #2: Warp control flow

one square: data processed by one thread
green: active thread produces data; **yellow:** active thread produces no data



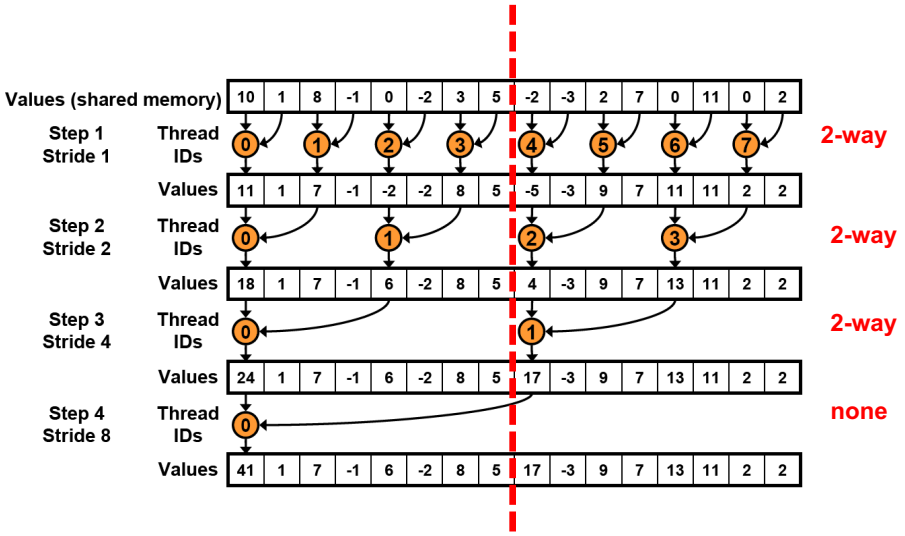
28

Performance for 4M element reduction (target: 0.268 ms)

	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**

29

Reduction #2: Problem



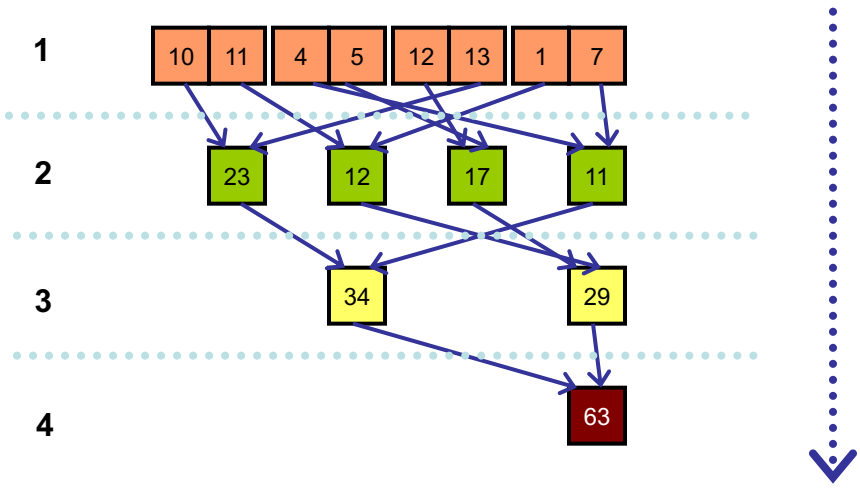
2-way bank conflicts at every step

Recall: 32 threads per warp, 32 banks of shared memory

To see the conflicts in the picture above (8 threads) assume banks = 8

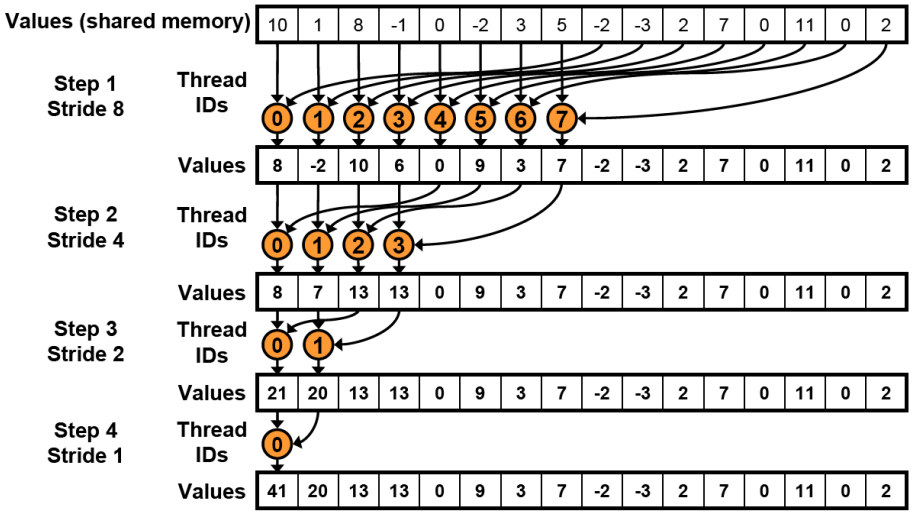
30

Observe: Arbitrary Unique Pairs OK



31

Reduction #3: Sequential Accesses



Eliminates bank conflicts

32

Reduction #3: Code Changes

- Replace stride indexing in the inner loop:

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

    if (index < blockDim.x == 0) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

- 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();
}
```

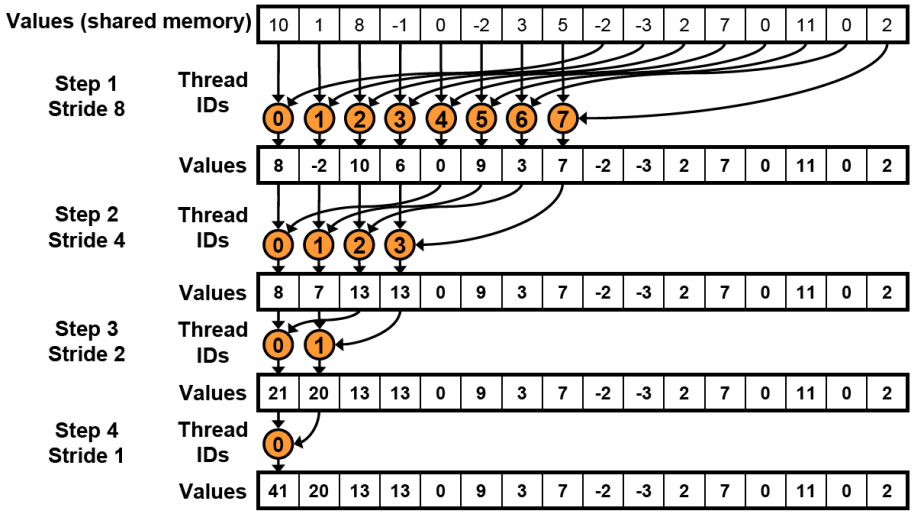
33

Performance for 4M element reduction (target: 0.268 ms)

	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**

34

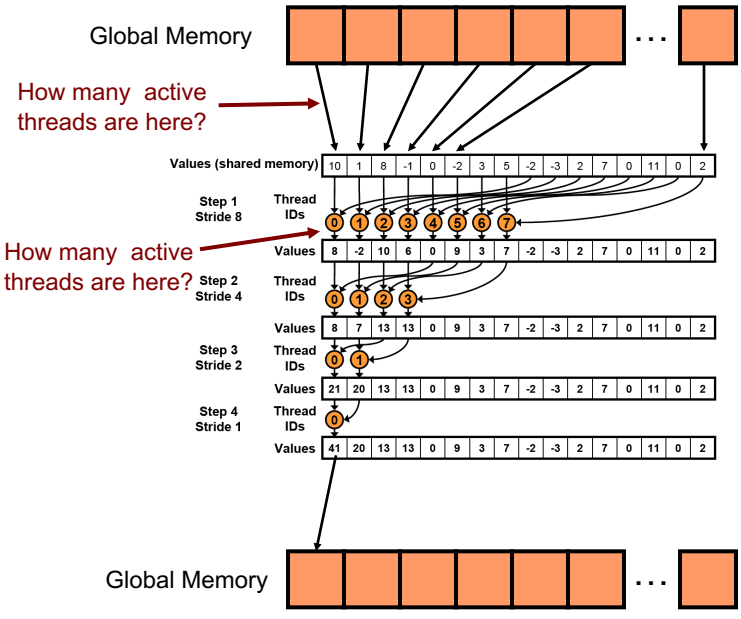
Reduction #3: Sequential Accesses



Why are we not there yet?

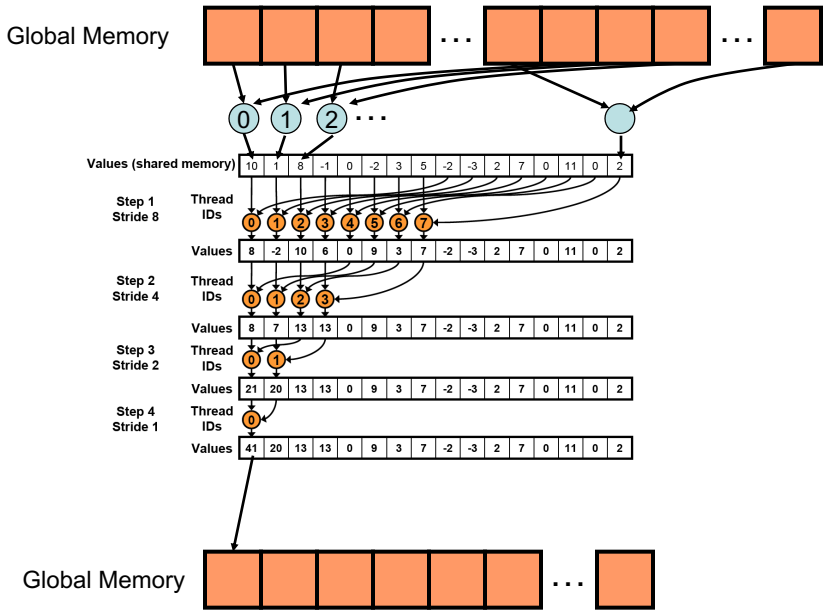
35

Remember the Big Picture



36

Reduction #4: First Step During Load



37

Reduction #4: First Step During Load

- Original: Each thread reads one element

```
// 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();
```

- Halve the number of blocks and replace the single load with two and the first step of the reduction:

```
// each thread loads 2 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();
```

38

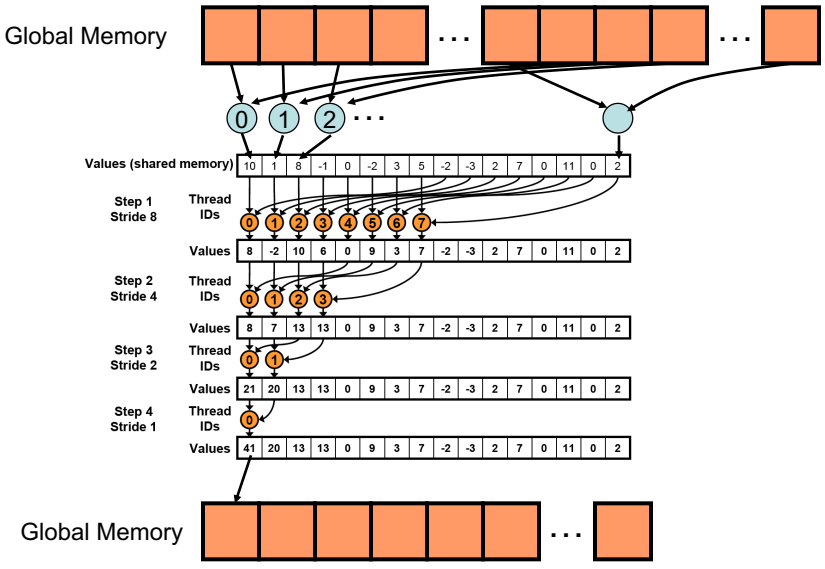
Performance for 4M element reduction (target: 0.268 ms)

	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**

Why are we not there yet?

39

Reduction #4: First Step During Load



Why are we not there yet?

40

Reduction #4: Low resource utilization

- What is the resource overhead here?

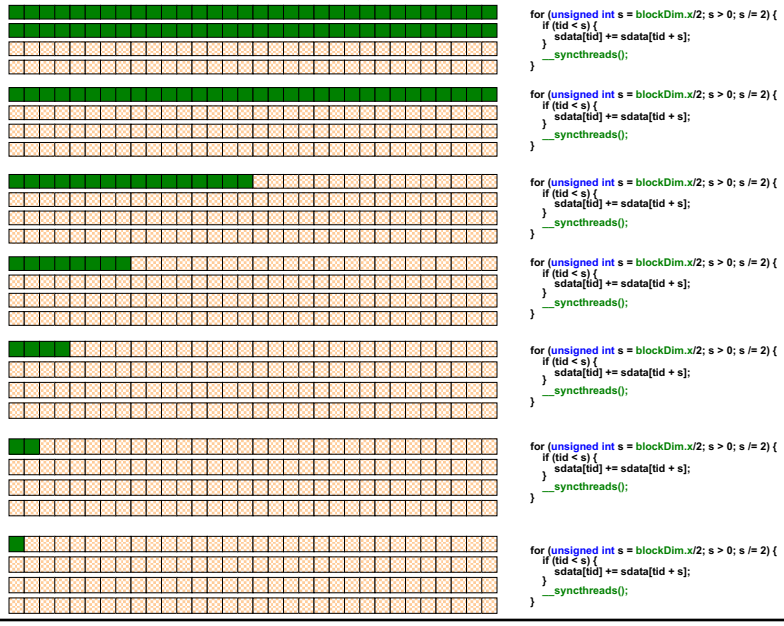
```
// do reduction in shared mem
for (unsigned int s = blockDim.x/2; s > 0; s /= 2) {

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

- Half the threads become idle at each step
- Low arithmetic intensity (too few arithmetic ops per load/store)
- Ancillary instructions
  (not useful computation, just bookkeeping)
- Instruction bottleneck
  - Address arithmetic and loop overhead
- Unroll loops to eliminate these “extra” instructions

41

Reduction #4: Warp control flow

one square: data processed by one thread
green: active thread produces data; **yellow:** active thread produces no data

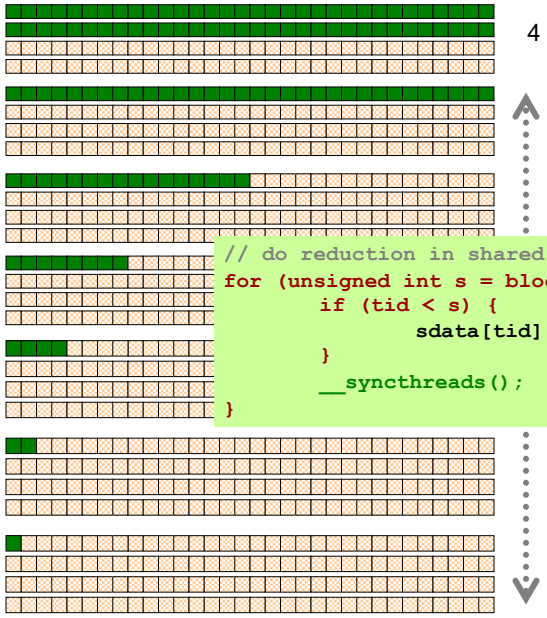


42

Reduction #4: Warp control flow

one square: data processed by one thread

green: active thread produces data; **yellow:** active thread produces no data




```
// 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();
}
```

43

Reduction #5: Unrolling the last warp

- At every step the number of active threads halves
  - When s <=32 there is only one warp left
- Instructions are SIMD-synchronous within a warp
  - They all happen in lock step
  - No need to use __syncthreads()
  - We don't need "if (tid < s)" since it does not save any work
    - All threads in a warp will "see" all instructions …
    - … regardless of whether they execute them or not
- Unroll the last 6 iterations of the inner loop
  - s <= 32

44

Reduction #5: Unrolling 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];
}
```

- This saves work in all warps not just the last one
  - Without unrolling all warps execute every iteration of the for loop

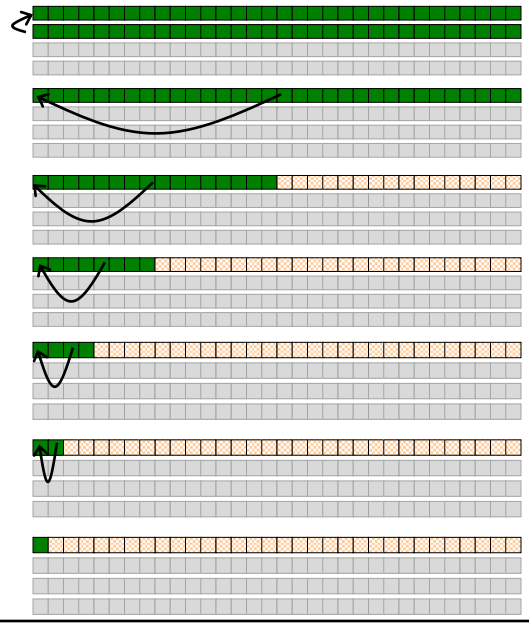
45

Reduction #5: wrap control flow

one square: data processed by one thread

green: active thread produces data; **yellow:** active thread produces no data

grey: inactive thread



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];

46

Unrolling the Last Warp: A closer look

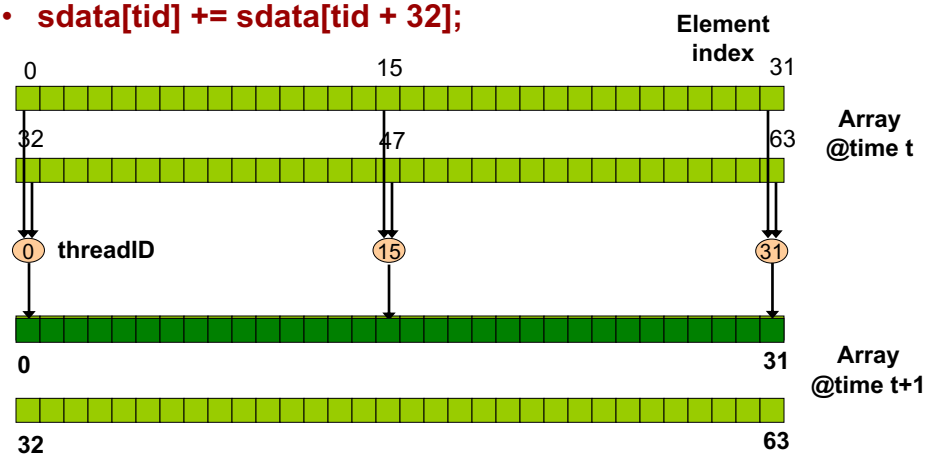
Keep in mind:

1. Warp execution proceeds in lock step for all threads
   - All threads execute the same instruction
   - So:
     - **sdata[tid] += sdata[tid + 32];**
   - Becomes:
     - Read into a register: sdata[tid]
     - Read into a register: sdate[tid+32]
     - Add the two
     - Write: sdata[tid]
2. Shared memory can provide up to 32 4-byte words per cycle
   - If we don't use this capability it just gets wasted

47

Unrolling the Last Warp: A Closer Look

- **sdata[tid] += sdata[tid + 32];**

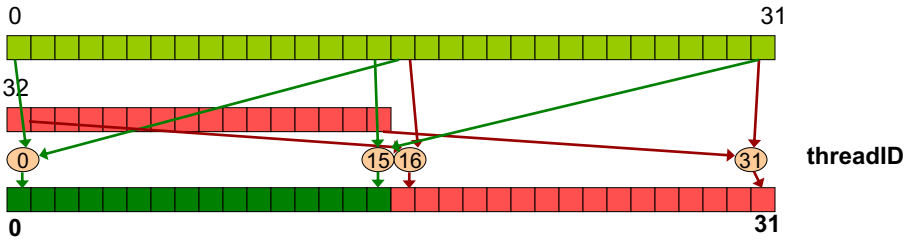


- **All threads doing useful work**

48

Unrolling the Last Warp: A Closer Look

- **sdata[tid] += sdata[tid + 16];**

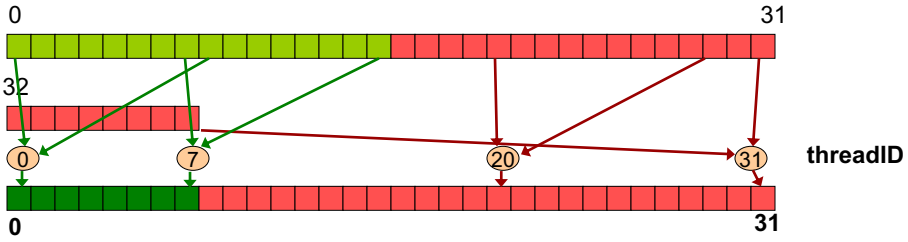


- Half of the threads, 16-31, are doing useless work
- At the end they "destroy" elements 16-31
- Elements 16-31 are inputs to threads 0-14
- But threads 0-15 read them before they get written by threads 16-31
  - All reads proceed in "parallel" first
  - All writes proceed in "parallel" last
- So, no correctness issues
- But, threads 16-31 are doing useless work
  - The units and bandwidth are there → no harm (only power)

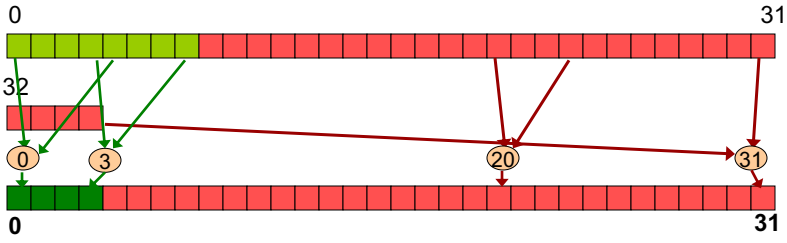
49

Unrolling the Last Warp: A Closer Look

sdata[tid] += sdata[tid + 8];



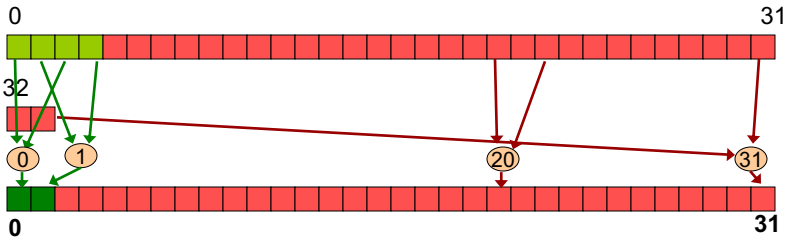
sdata[tid] += sdata[tid + 4];



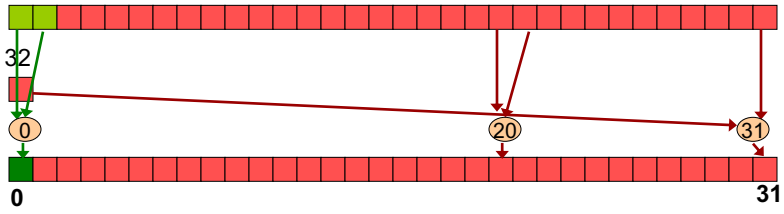
50

Unrolling the Last Warp: A Closer Look

sdata[tid] += sdata[tid + 2];



sdata[tid] += sdata[tid + 1];



51

Performance for 4M element reduction (target: 0.268 ms)

	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**

What else is left to optimize?

52