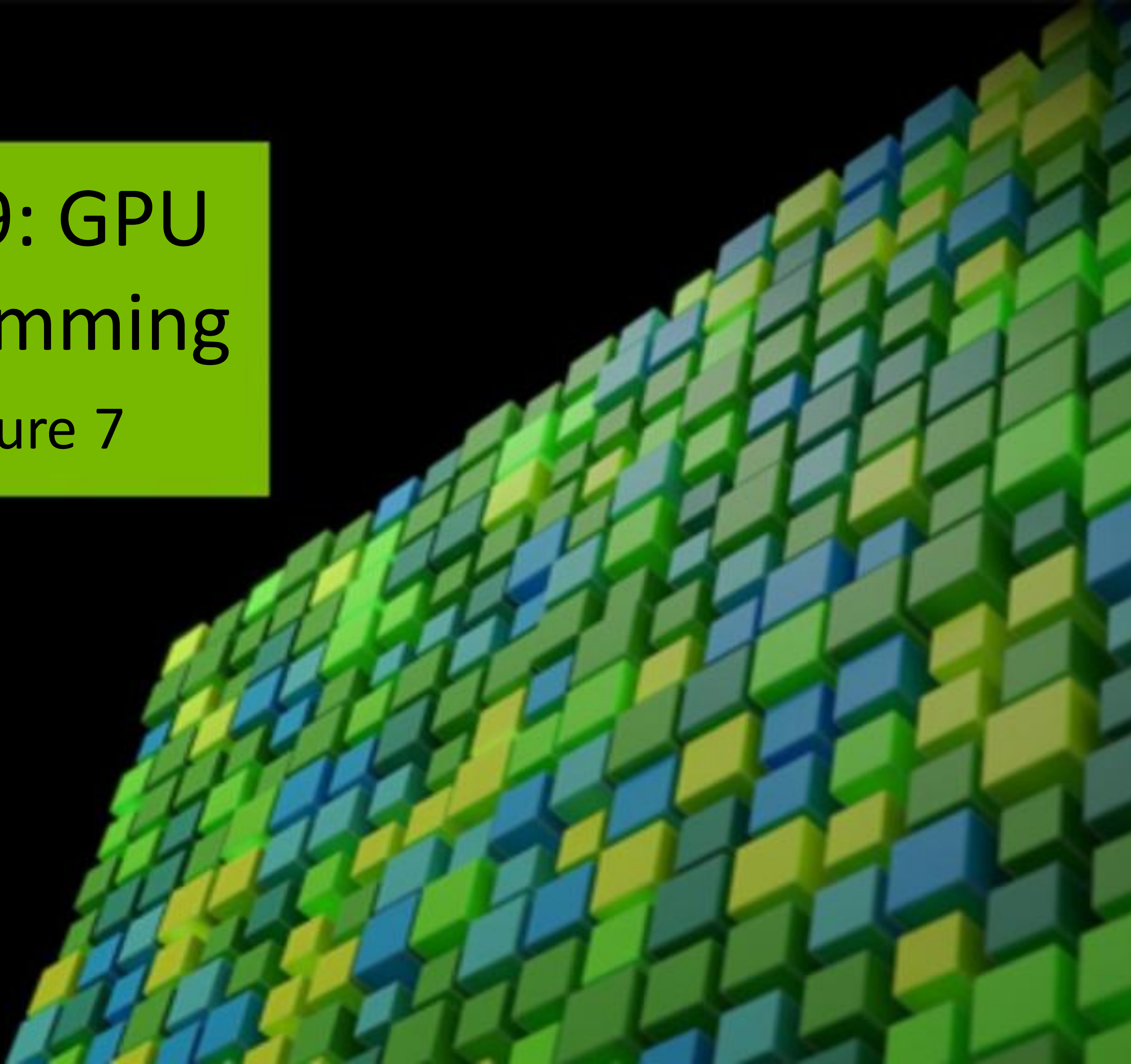
CS 179: GPU Programming

Lecture 7



Last Week

- Memory optimizations using different GPU caches
- Atomic operations
- Synchronization with __syncthreads()

This week, Week 3

- More advanced GPU-accelerable algorithms
- “Reductions” to parallelize problems that might not seem intuitively parallelizable
  - Not the same as reductions in complexity theory or machine learning!
  - Lots of technical meanings for “Reduction” – see https://en.wikipedia.org/wiki/Reduction
- See https://en.wikipedia.org/wiki/Reduction_Operator

This Lecture -- Outline

- Reductions for GPUs
- Examples of GPU-accelerable algorithms:
  - (To be used in combination for Quicksort!)
  - **Sum of array**
  - Prefix sum
  - Stream compaction
  - Sorting (quicksort)

GPU Reductions

- Again, see https://en.wikipedia.org/wiki/Reduction_Operator
- Commonly used in parallel programming to reduce all elements of an array to **single result**.
- Supposed to be associative and often (but not necessarily) commutative.
- Can be used in "Map Reduce" where reduction operator is applied (mapped) to all elements before they are reduced.
- Many reduction operators can be used for broadcasting, to distribute data to all processors.

Properties of Reduction Operator

- Allows many serial operations to be performed in parallel, reducing number of steps
- Helps break down full task into partial tasks. Calculates partial results to obtain final result.
- Stores results of partial tasks into “private copies” of the variable.
  - These private copies are then merged into a shared copy at the end.
- An operator is a reduction operator for example, if:
  - It can reduce an array to a single scalar value. (eg, adding all elements of array).
  - The final result should be obtainable from the results of the partial tasks that were created.
- Satisfied for commutative and associative operators that are applied to all array elements.
- Some operators which satisfy these requirements are integer addition, multiplication, and some logical operators (and, or, etc.).

MapReduce, more advanced…

- See https://en.wikipedia.org/wiki/MapReduce
- MapReduce is a framework for processing parallelizable problems across large datasets using a large number of computers
- Usually composed of three steps:
- **Map**: each worker node applies the map function to the local data, and writes the output to a temporary storage.
  - A master node ensures that only one copy of the redundant input data is processed.
- **Shuffle**: worker nodes redistribute data based on the output keys
  - (produced by the map function), such that all data belonging to one key is located on the same worker node.
- **Reduce**: worker nodes now process each group of output data, per key, in parallel.
- MapReduce allows for distributed processing of the map /reduction operations.
- Maps can be performed in parallel, when mapping operations are independent

Outline

- Examples of GPU-accelerable algorithms:
  - **Sum of array**
  - Prefix sum
  - Stream compaction
  - Sorting (quicksort)

Elementwise Integer Addition

Problem: C[i] = A[i] + B[i]

- CPU code:

```
float *C = malloc(N * sizeof(float));
for (int i = 0; i < N; i++)
    C[i] = A[i] + B[i];
```

- GPU code:

```
// assign device and host memory pointers, and allocate memory
in host. Adds simultaneously across thread indices!

int thread_index = threadIdx.x + blockIdx.x * blockDim.x;
while (thread_index < N) {
    C[thread_index] = A[thread_index] + B[thread_index];
    thread_index += blockDim.x * gridDim.x;
}
```

Simple Reduction Example

Problem: SUM(A[])

- CPU code:

```
float sum = 0.0;
for (int i = 0; i < N; i++)
    sum += A[i];
```

- GPU Pseudocode:

```
// set up device and host memory pointers
// create threads and get thread indices
// assign each thread a specific region to sum over
// wait for all threads to finish running ( __syncthreads; )
// combine all thread sums for final solution
```

Naive Reduction

- Suppose we wished to accumulate our results…

```
__global__ void
cudaSum_atomic_kernel(const float* const inputs,
                      unsigned int numberOfInputs,
                      const float* const c,
                      unsigned int polynomialOrder,
                      float* output) {

    //set inputIndex to initial thread index...


    float partial_sum = 0.0;

    while (inputIndex < numberOfInputs){

        //calculate polynomial value at inputs[inputIndex] and
        //add it to the partial sum...

        //increment input index to the next value...
    }

    output += partial_sum

}
```

Naive Reduction

- Race conditions! Could load old value before new one (from another thread) is written out

```
__global__ void
cudaSum_atomic_kernel(const float* const inputs,
                      unsigned int numberOfInputs,
                      const float* const c,
                      unsigned int polynomialOrder,
                      float* output) {

    //set inputIndex to initial thread index...


    float partial_sum = 0.0;

    while (inputIndex < numberOfInputs){

        //calculate polynomial value at inputs[inputIndex] and
        //add it to the partial sum...

        //increment input index to the next value...
    }

    output += partial_sum
}
```

Thread-unsafe!

Naive (but correct) Reduction

- We could do a bunch of atomic adds to our global accumulator…

```
__global__ void
cudaSum_atomic_kernel(const float* const inputs,
                      unsigned int numberOfInputs,
                      const float* const c,
                      unsigned int polynomialOrder,
                      float* output) {

    //set inputIndex to initial thread index...


    float partial_sum = 0.0;

    while (inputIndex < numberOfInputs){

        //calculate polynomial value at inputs[inputIndex] and
        //add it to the partial sum...

        //increment input index to the next value...
    }

    atomicAdd(output, partial_sum);

}
```

Naive (but correct) Reduction

- But then we lose a lot of our parallelism ☹

```
__global__ void
cudaSum_atomic_kernel(const float* const inputs,
                      unsigned int numberOfInputs,
                      const float* const c,
                      unsigned int polynomialOrder,
                      float* output) {

    //set inputIndex to initial thread index...


    float partial_sum = 0.0;

    while (inputIndex < numberOfInputs){

        //calculate polynomial value at inputs[inputIndex] and
        //add it to the partial sum...

        //increment input index to the next value...
    }

    atomicAdd(output, partial_sum);
}
```

Every thread needs to wait...

Shared memory accumulation

- Right now, the only parallelism we get is partial sums per thread
- Idea: store partial sums per thread in shared memory
- If we do this, we can accumulate partial sums per block in shared memory, and THEN atomically add a much larger sum to the global accumulator

Shared memory accumulation

```
__global__ void
cudaSum_linear_kernel(const float* const inputs,
                                   unsigned int numberOfInputs,
                                   const float* const c,
                                   unsigned int polynomialOrder,
                                   float * output) {

    extern __shared__ float partial_outputs[];

    //calculate partial_sum as before...



    //but this time, store the result in the partial_outputs[threadIndex]...


    //Make all threads in the block finish before continuing!
    syncthreads();
```

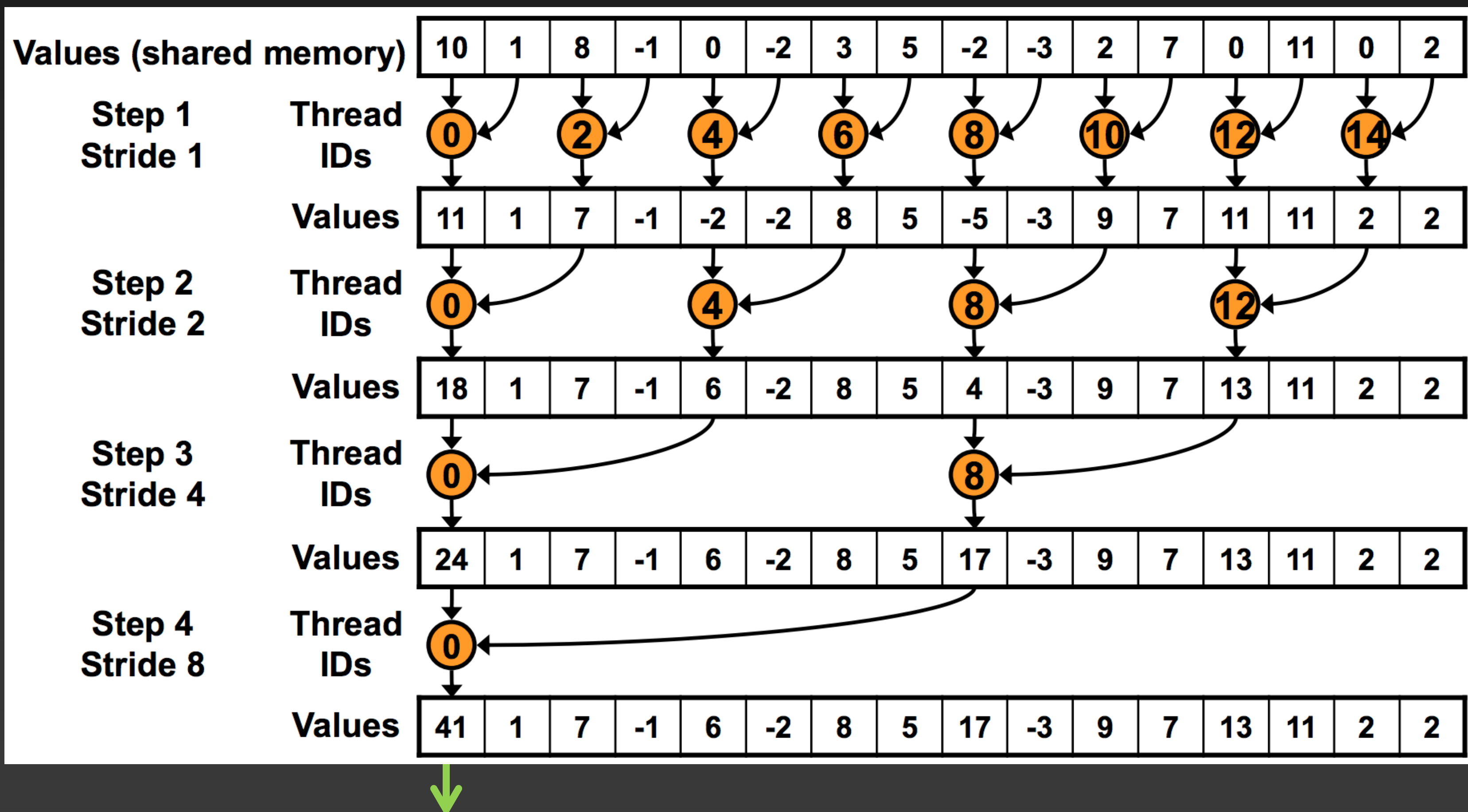
Shared memory accumulation

```
    //Use the first thread in the block to accumulate the results
    //of the other threads in said block
    if (threadIdx.x == 0) {
        for (unsigned int threadIndex = 1; threadIndex < blockDim.x;
                ++threadIndex){

            //Accumulate all the other partial sums into thread 0's
            //partial sum
            partial_sum += partial_outputs[threadIndex];
        }

        //Now we finally accumulate
        atomicAdd(output, partial_sum);
    }

}
```

Shared memory accumulation

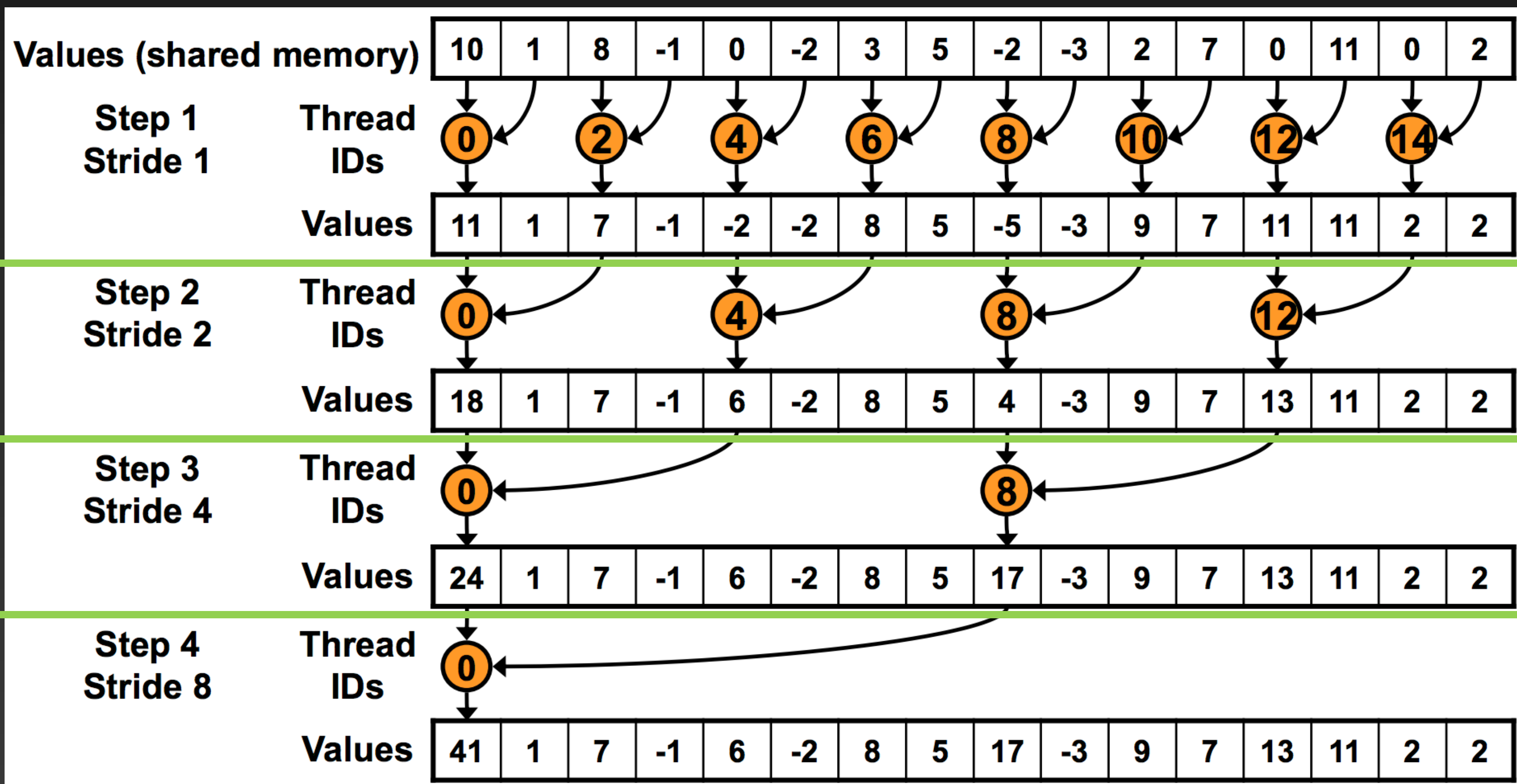
- It doesn’t seem particularly efficient to have one thread per block accumulate for the entire block…
- Can we do better?

“Binary tree” reduction



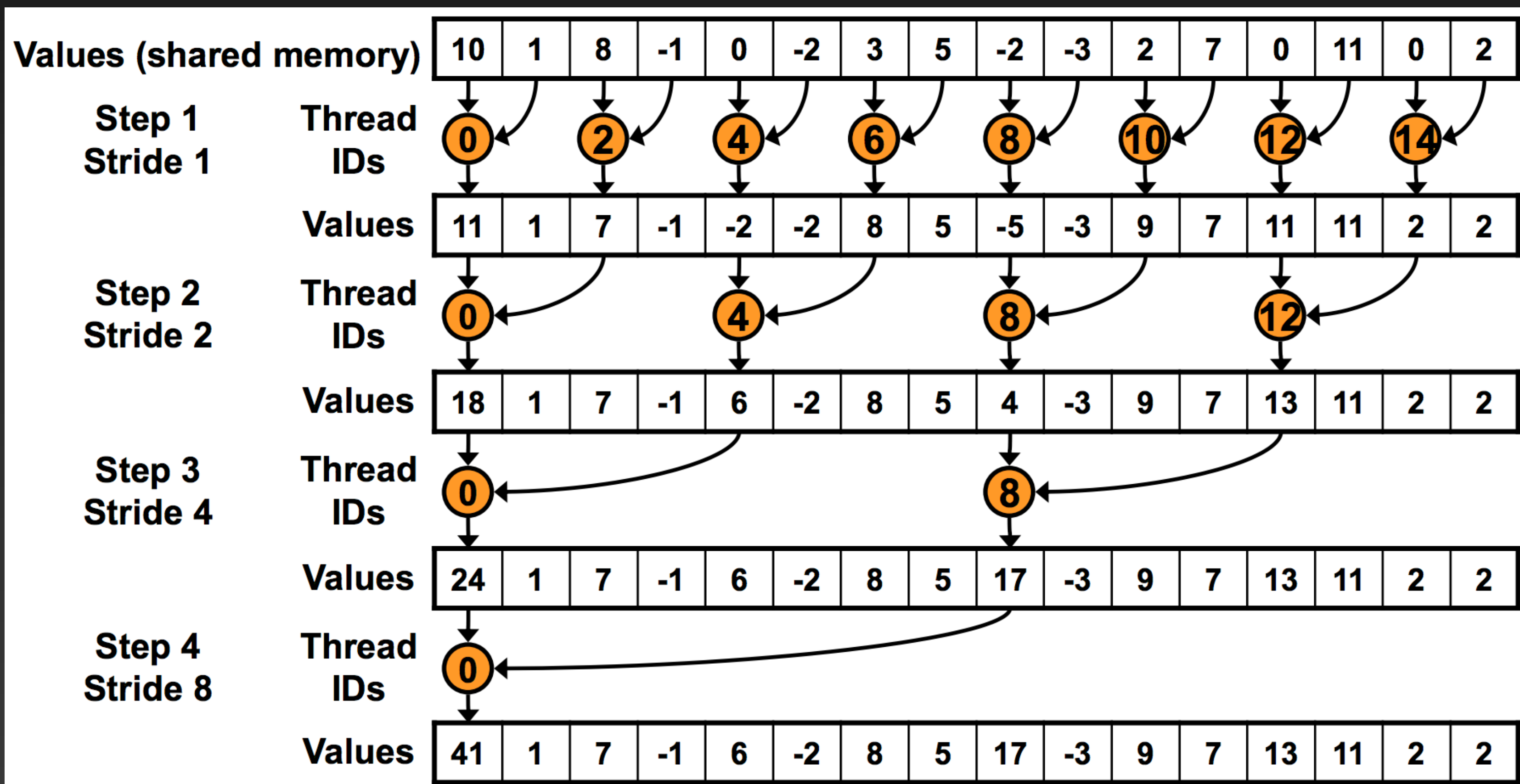
Thread 0 atomicAdd’s
this to global result

"Binary tree" reduction



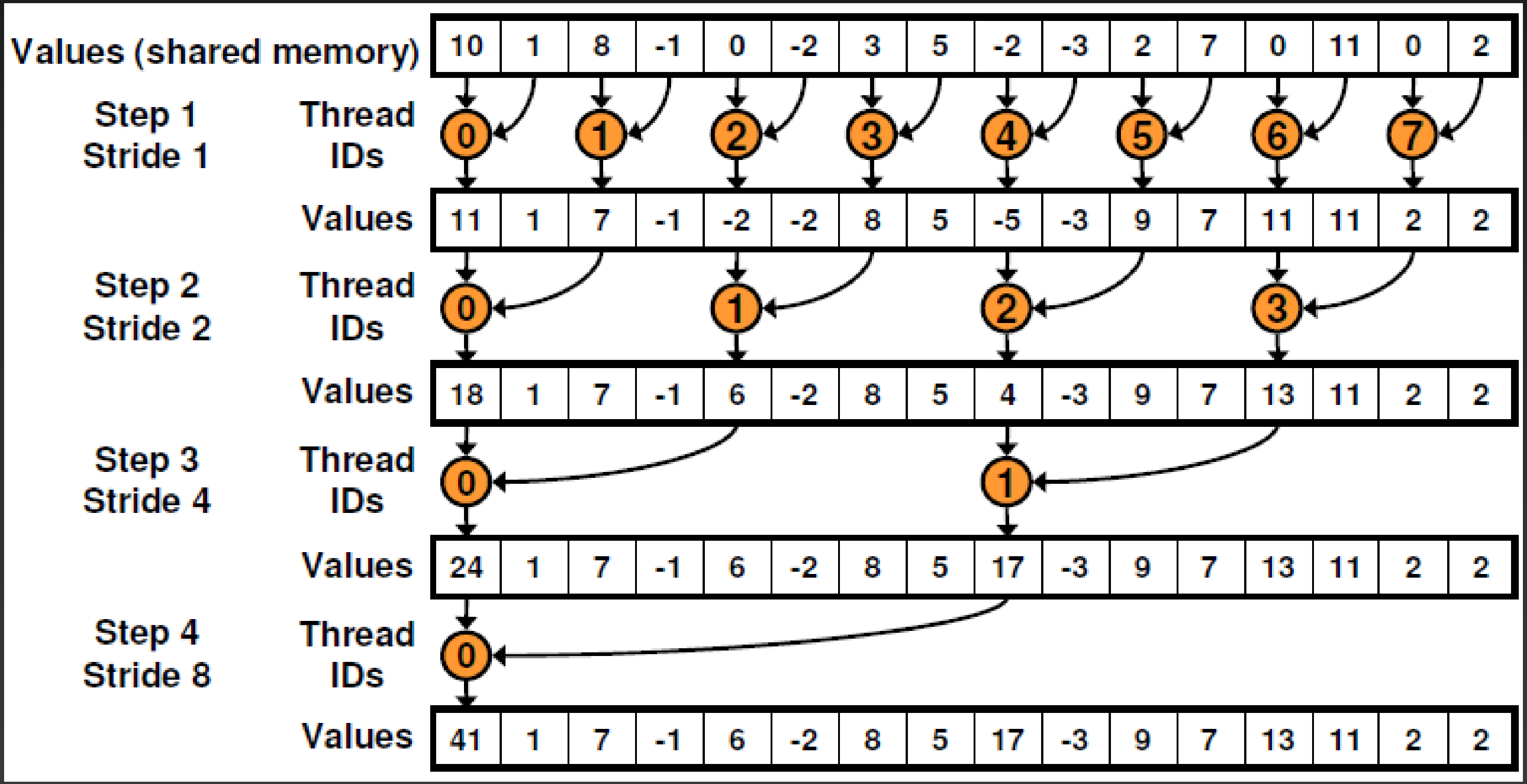
Use __syncthreads()
before proceeding!

“Binary tree” reduction

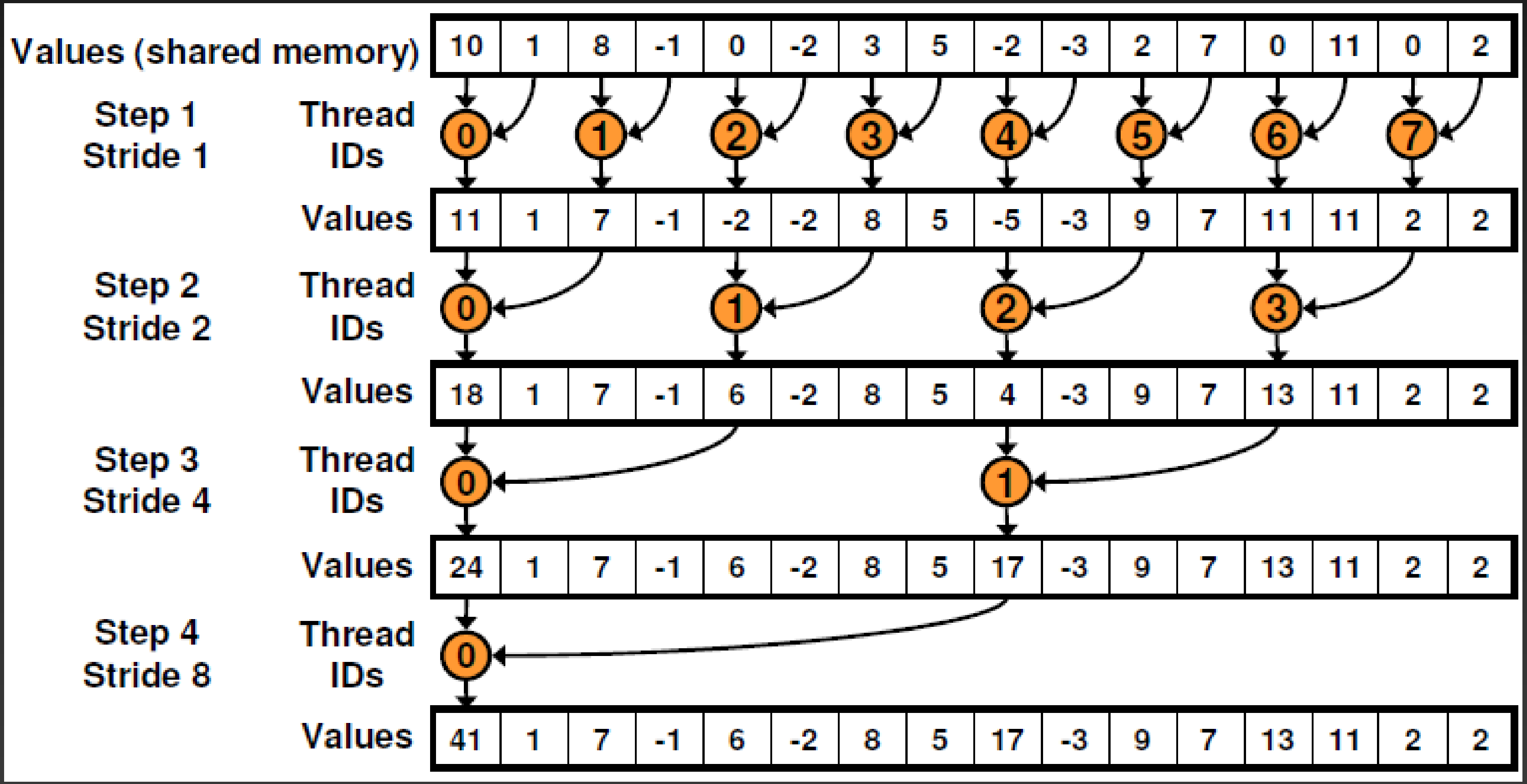


- Warp Divergence! Odd threads won’t even execute.

Non-divergent reduction

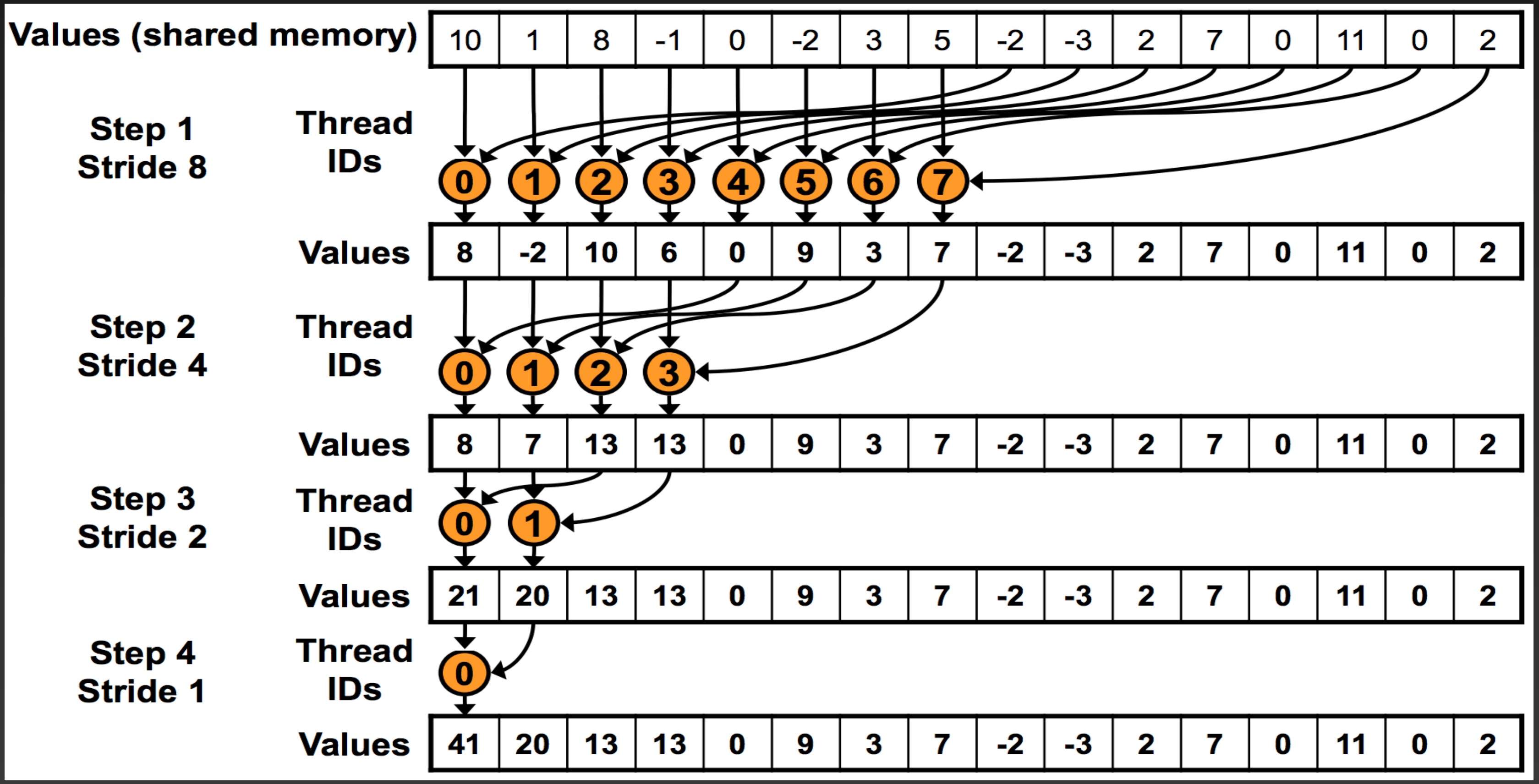


Non-divergent reduction



- Shared Memory Bank Conflicts!
  - 2-way on 1st iteration, 4-way on 2nd iteration, …

Sequential addressing



- Automatically resolves bank conflicts!

Sum Reduction

- More improvements possible (gets crazy!)
  - "Optimizing Parallel Reduction in CUDA" (Harris)
    - Code examples!

- Moral:
  - Different type of GPU-accelerated problems
    - Some are "parallelizable" in a different sense
  - More hardware considerations in play

Outline

- GPU-accelerated:
  - Sum of array
  - **Prefix sum – See https://en.wikipedia.org/wiki/Prefix_sum**
  - Stream compaction
  - Sorting (quicksort)

Prefix Sum

- Given input sequence x[n], produce sequence

$$y[n] = \sum_{k=0}^{n-1} x[k]$$

  – e.g. x[n] = (1, 1, 1, 1, 1, 1, 1)

    -> y[n] = (0, 1, 2, 3, 4, 5, 6)

  – e.g. x[n] = (1, 2, 3, 4, 5, 6)

    -> y[n] = (0, 1, 3, 6, 10, 15)

Prefix Sum

- Given input sequence x[n], produce sequence

$$y[n] = \sum_{k=0}^{n-1} x[k]$$

  - e.g. x[n] = (1, 2, 3, 4, 5, 6)

    -> y[n] = (0, 1, 3, 6, 10, 15)

- Recurrence relation:

$$y[n] = y[n-1] + x[n]$$

Prefix Sum

- Recurrence relation:

$$y[n] = y[n-1] + x[n]$$

  – Is it parallelizable? Is it GPU-accelerable?

- Recall:
  – $y[n] = x[n] + x[n-1] + \cdots + x[n-(K-1)]$
    » Easily parallelizable!
  – $y[n] = c \cdot x[n] + (1-c) \cdot y[n-1]$
    » Not so much

Prefix Sum

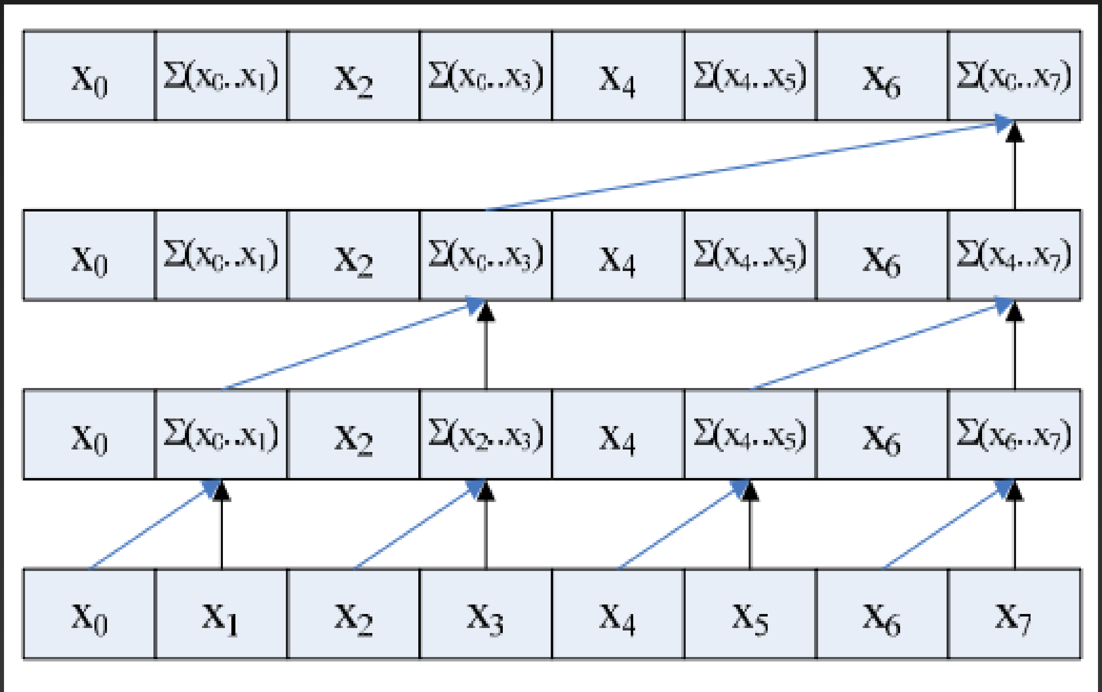
- Recurrence relation:

  $$y[n] = y[n-1] + x[n]$$

  – Is it parallelizable? Is it GPU-accelerable?

- Goal:
  – Parallelize using a “reduction-like” strategy

Prefix Sum sample code (up-sweep)



[1, 3, 3, 10, 5, 11, 7, 36]

[1, 3, 3, 10, 5, 11, 7, 26]

[1, 3, 3, 7, 5, 11, 7, 15]

Original array

[1, 2, 3, 4, 5, 6, 7, 8]

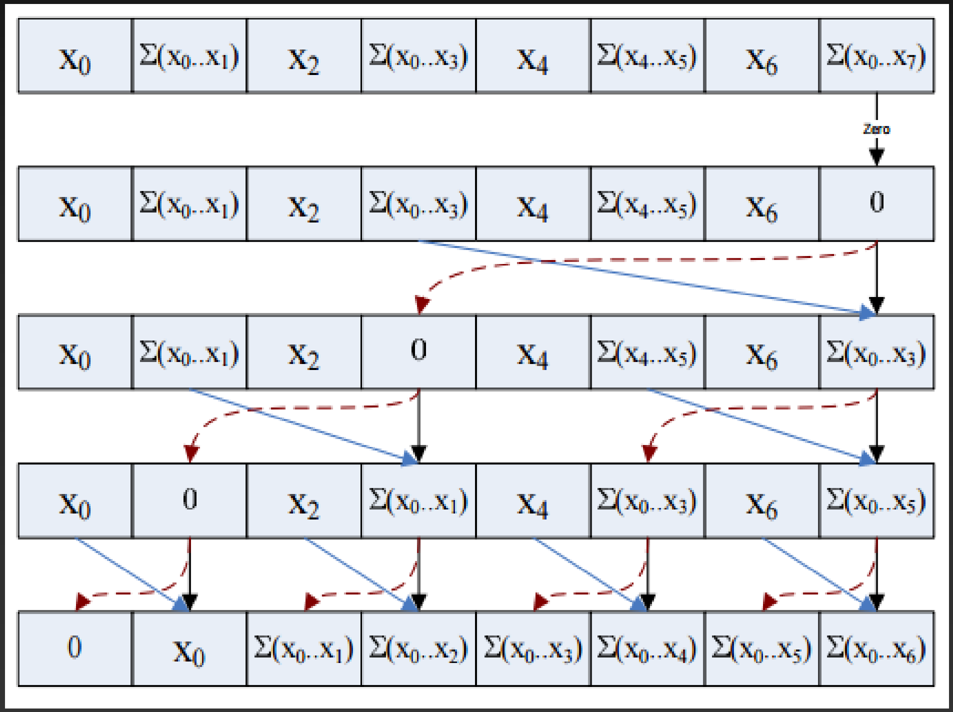
for d = 0 to $(\log_2 n) - 1$ do
  for all k = 0 to n-1 by 2^{d+1} in parallel do
    $x[k + 2^{d+1} - 1] = x[k + 2^d - 1] + x[k + 2^d]$

We want:

[0, 1, 3, 6, 10, 15, 21, 28]

(University of Michigan EECS,
http://www.eecs.umich.edu/courses/eecs570/hw/parprefix.pdf

Prefix Sum sample code (down-sweep)



x[n-1] = 0
for d = $\log_2(n) - 1$ down to 0 do
  for all k = 0 to n-1 by 2^d+1 in parallel do
    t = $x[k + 2^d - 1]$
    $x[k + 2^d - 1] = x[k + 2^d]$
    $x[k + 2^d] = t + x[k + 2^d]$

Original: [1, 2, 3, 4, 5, 6, 7, 8]

[1, 3, 3, 10, 5, 11, 7, 36]

[1, 3, 3, 10, 5, 11, 7, 0]

[1, 3, 3, 0, 5, 11, 7, 10]

[1, 0, 3, 3, 5, 10, 7, 21]

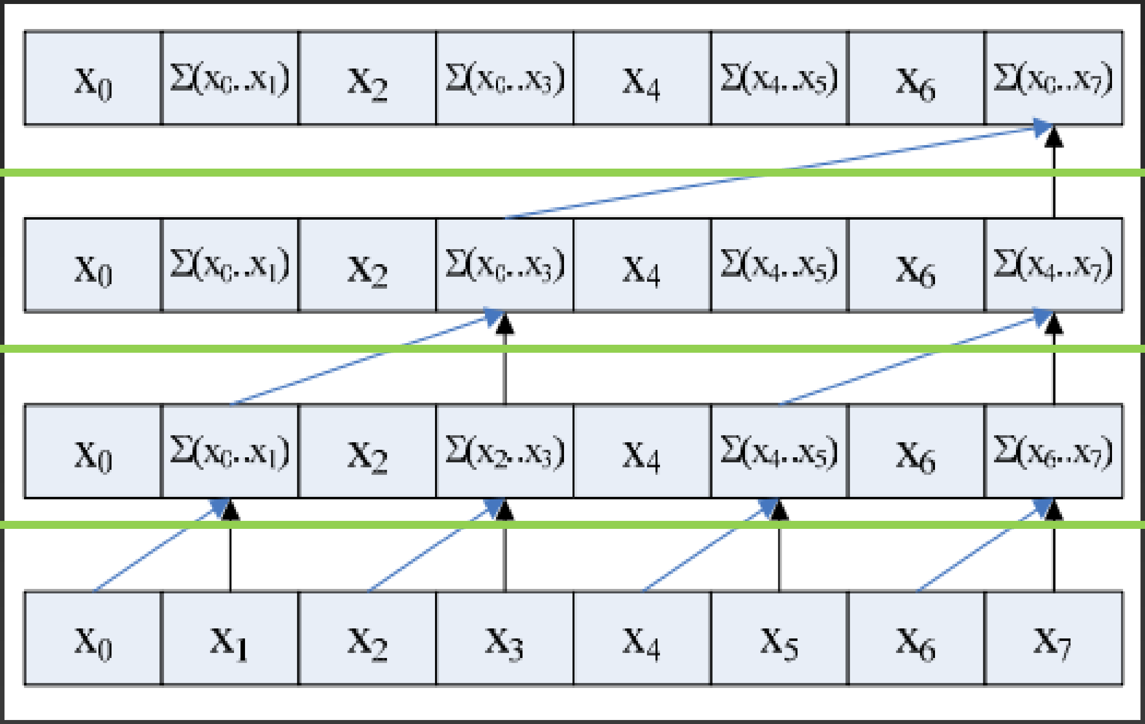
Final result

[0, 1, 3, 6, 10, 15, 21, 28]

(University of Michigan EECS, http://www.eecs.umich.edu/courses/eecs570/hw/parprefix.pdf

Prefix Sum (Up-Sweep)

Use __syncthreads() before proceeding!

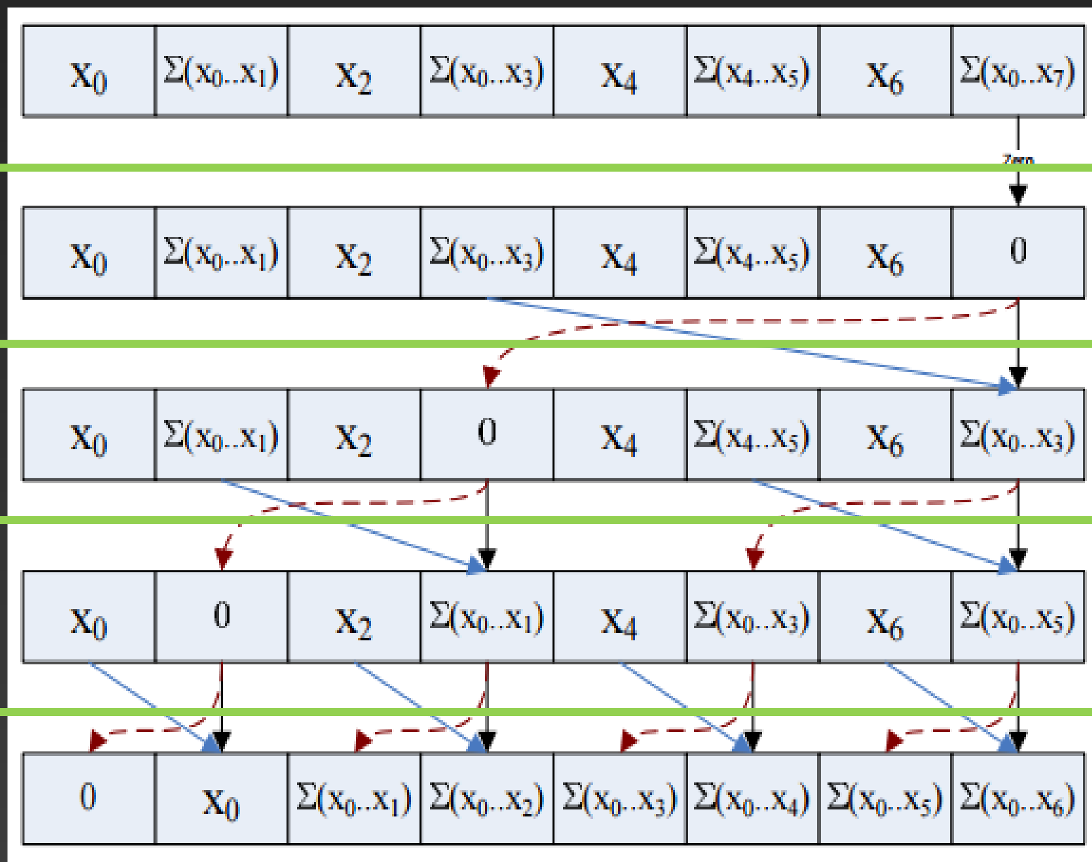


Original array →

(University of Michigan EECS, http://www.eecs.umich.edu/courses/eecs570/hw/parprefix.pdf

Prefix Sum (Down-Sweep)

Use __syncthreads() before proceeding!



Final result →

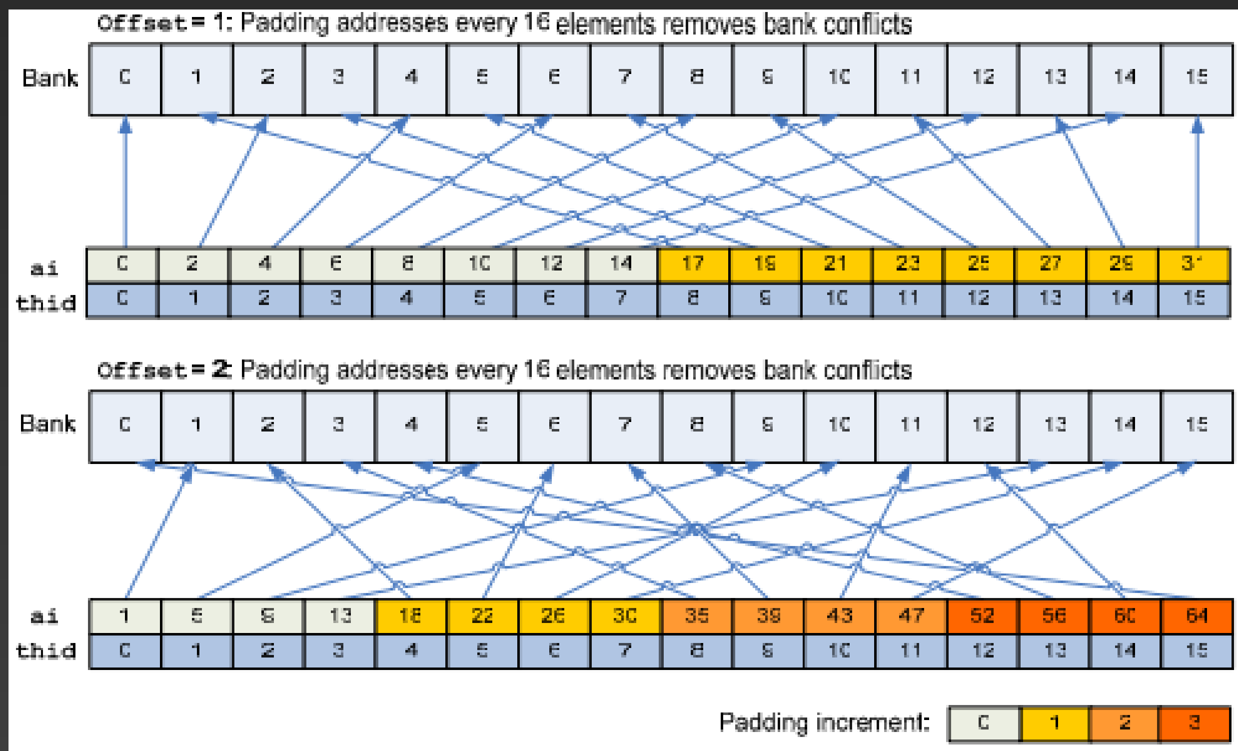
(University of Michigan EECS, http://www.eecs.umich.edu/courses/eecs570/hw/parprefix.pdf

Prefix Sum

- Bank conflicts galore!
  - 2-way, 4-way, …

Prefix Sum

- Bank conflicts!
  - 2-way, 4-way, …
  - Pad addresses!



(University of Michigan EECS, http://www.eecs.umich.edu/courses/eecs570/hw/parprefix.pdf

Prefix Sum

- https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda -- See Link for a More In-Depth Explanation of Up-Sweep and Down-Sweep
- All of GPU Gems 3 available to download here!
- See also Ch8 of textbook (Kirk and Hwu) for a more build-up and motivation for the up-sweep and down-sweep algorithm (like we did for the array sum)

Outline

- GPU-accelerated:
  - Sum of array
  - Prefix sum
  - **<u>Stream compaction</u> (to be used for Quicksort!)**
  - Sorting (quicksort)

Stream Compaction

- Problem:
  - Given array A, produce sub-array of A defined by Boolean condition
  - e.g. given array:

2	5	1	4	6	3

    - Produce array of numbers > 3

5	4	6

  - Will use for implementing Quicksort on GPUs!

Stream Compaction

- Given array A:

2	5	1	4	6	3

  – GPU kernel 1: Evaluate boolean condition,
    - Array M: 1 if true, 0 if false

0	1	0	1	1	0

  – GPU kernel 2: Cumulative sum of M (denote S)

0	1	1	2	3	3

  – GPU kernel 3: At each index,
    - if M[idx] is 1, store A[idx] in output at position (S[idx] - 1)

5	4	6

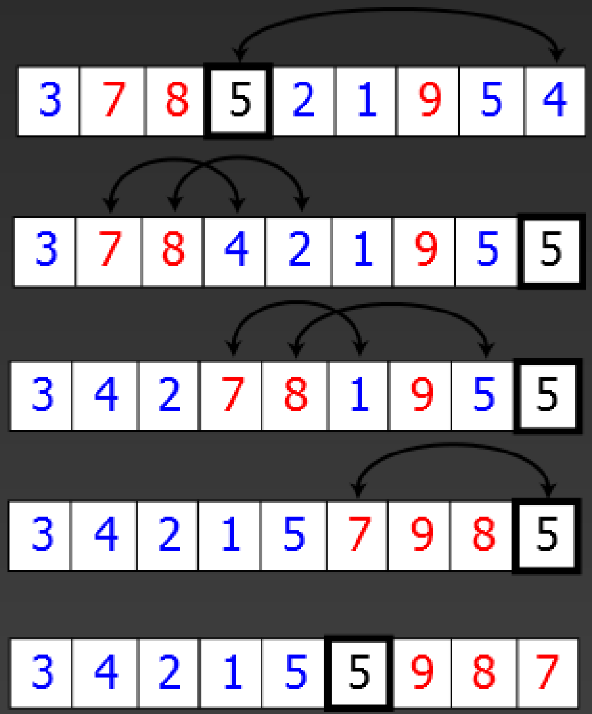
Outline

- GPU-accelerated:
  - Sum of array
  - Prefix sum
  - Stream compaction
  - **Sorting (quicksort)**
    - **See https://en.wikipedia.org/wiki/Quicksort**

GPU-accelerated quicksort

- Quicksort:
  - Divide-and-conquer algorithm
  - Partition array along chosen pivot point

Sequential partition



- Pseudocode:

```
quicksort(A, loIdx, hiIdx):
    if lo < hi:
        pIdx := partition(A, loIdx, hiIdx)
        quicksort(A, loIdx, pIdx - 1)
        quicksort(A, pIdx + 1, hiIdx)
```

GPU-accelerated partition

- Given array A:

2	5	1	4	6	3

  - Choose pivot (e.g. 3)
  - Stream compact on condition: ≤ 3

2	1				

  - Store pivot

2	1	3			

  - Stream compact on condition: > 3 (store with offset)

2	1	3	5	4	6

GPU acceleration details

- Synchronize between calls of the previous algorithm
- Continued partitioning/synchronization on sub-arrays results in sorted array

Final Thoughts

- "Less obviously parallelizable" problems
  - Hardware matters! (synchronization, bank conflicts, …)
- Resources:
  - GPU Gems, Vol. 3, Ch. 39
  - Highly Recommend Reading <u>This</u> Guide to CUDA Optimization, with a Reduction Example
  - Kirk and Hwu Chapters 7-12 for more parallel algorithms