CS 179: GPU Programming

Lecture 7

Week 3

- Goals:
  - More involved GPU-accelerable algorithms
    - Relevant hardware quirks
  - CUDA libraries

Outline

- GPU-accelerated:
  - Reduction
  - Prefix sum
  - Stream compaction
  - Sorting (quicksort)

Reduction

- Find the sum of an array:
  - (Or any associative operator, e.g. product)

- CPU code:

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

- Add two arrays
  - A[] + B[] -> C[]
- CPU code:

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

- Find the sum of an array:
  - (Or any associative operator, e.g. product)
- CPU code:

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

Reduction vs. elementwise add

Add two arrays
(multithreaded pseudocode)

```
(allocate memory for C)

(create threads, assign
indices)

...

In each thread,
    for (i from beginning region of
    thread)
         C[i] <- A[i] + B[i]

Wait for threads to
synchronize...
```

Sum of an array
(multithreaded pseudocode)

```
(set sum to 0.0)

(create threads, assign
indices)

...

In each thread,
    (Set thread_sum to 0.0)

    for (i from beginning region of
    thread)
         thread_sum += A[i]

     "return" thread_sum

Wait for threads to
synchronize...

for j = 0,…,#threads-1:
    sum += (thread j's sum)
```

Reduction vs. elementwise add

Add two arrays

(multithreaded pseudocode)

```
(allocate memory for C)

(create threads, assign
indices)

...

In each thread,
    for (i from beginning region of
    thread)
         C[i] <- A[i] + B[i]

Wait for threads to
synchronize...
```

Sum of an array

(multithreaded pseudocode)

```
(set sum to 0.0)

(create threads, assign
indices)

...

In each thread,
    (Set thread_sum to 0.0)

    for (i from beginning region of
    thread)
         thread_sum += A[i]

    "return" thread_sum

Wait for threads to
synchronize...

for j = 0,…,#threads-1:
    sum += (thread j's sum)
```

Serial recombination!

Reduction vs. elementwise add

- Serial recombination has greater impact with more threads
  - CPU – no big deal
  - GPU – big deal

Sum of an array
(multithreaded pseudocode)

```
(set sum to 0.0)

(create threads, assign
indices)

...

In each thread,
    (Set thread_sum to 0.0)

    for (i from beginning region of
    thread)
         thread_sum += A[i]

     "return" thread_sum

Wait for threads to
synchronize...

for j = 0,…,#threads-1:
    sum += (thread j's sum)
```

Serial recombination!

Reduction vs. elementwise add (v2)

Add two arrays

(multithreaded pseudocode)

```
(allocate memory for C)

(create threads, assign
indices)

...

In each thread,
    for (i from beginning region of
    thread)
        C[i] <- A[i] + B[i]

Wait for threads to
synchronize...
```

Sum of an array

(multithreaded pseudocode)

```
(set sum to 0.0)

(create threads, assign
indices)

...

In each thread,
    (Set thread_sum to 0.0)

    for (i from beginning region of
    thread)
        thread_sum += A[i]

    Atomically add thread_sum to sum

Wait for threads to
synchronize...
```

Reduction vs. elementwise add (v2)

Add two arrays

(multithreaded pseudocode)

```
(allocate memory for C)

(create threads, assign
indices)

...

In each thread,
    for (i from beginning region of
    thread)
        C[i] <- A[i] + B[i]



Wait for threads to
synchronize...
```

Sum of an array

(multithreaded pseudocode)

```
(set sum to 0.0)

(create threads, assign
indices)

...

In each thread,
    (Set thread_sum to 0.0)

    for (i from beginning region of
    thread)
        thread_sum += A[i]

    Atomically add thread_sum to sum

Wait for threads to
synchronize...
```

Serialized access!

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

- 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

}
```

Thread-unsafe!

Naive (but correct) reduction

```
__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);

}
```

GPU threads in naive reduction

http://telegraph.co.uk/

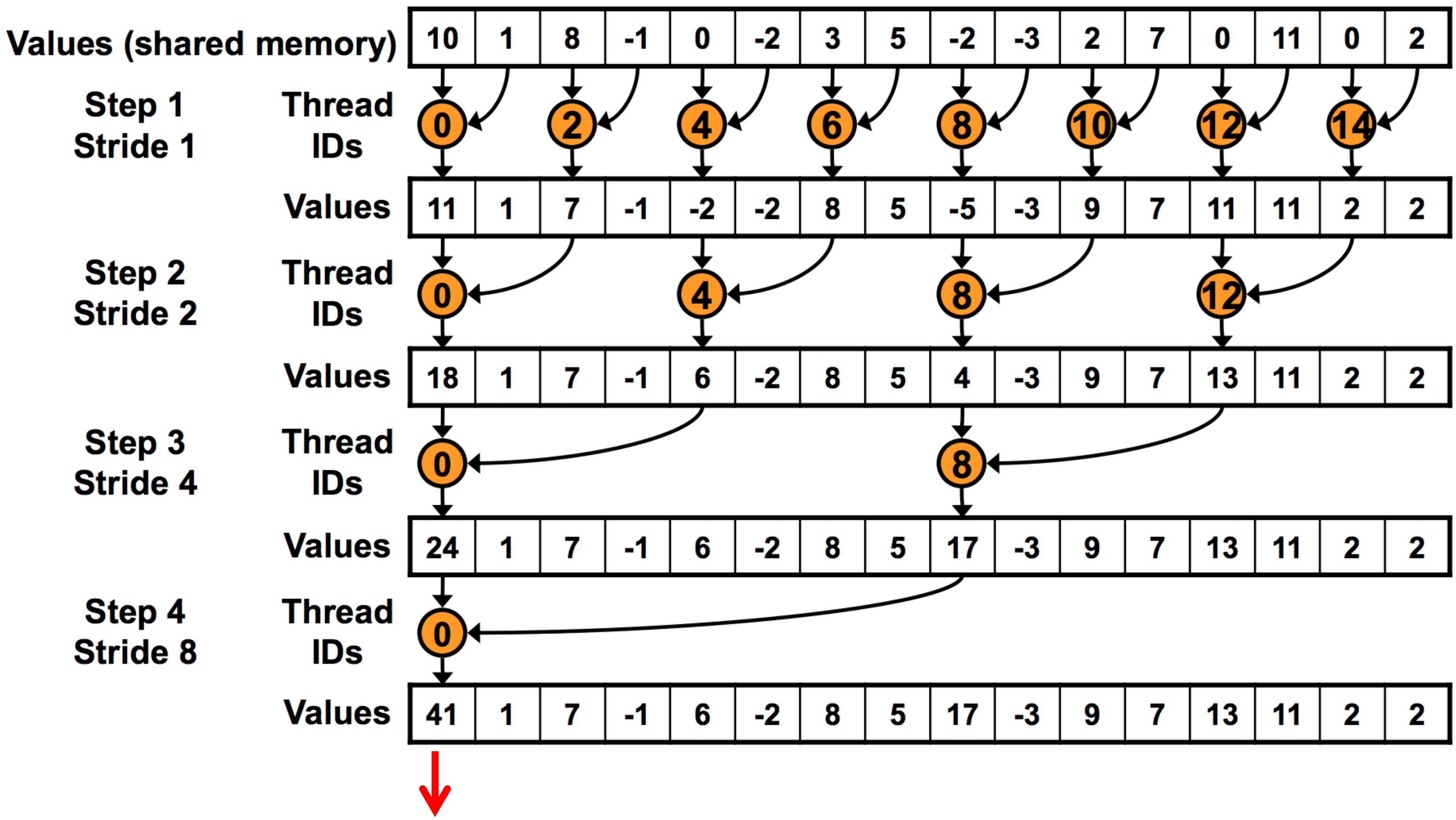
Shared memory accumulation

```
__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 (2)

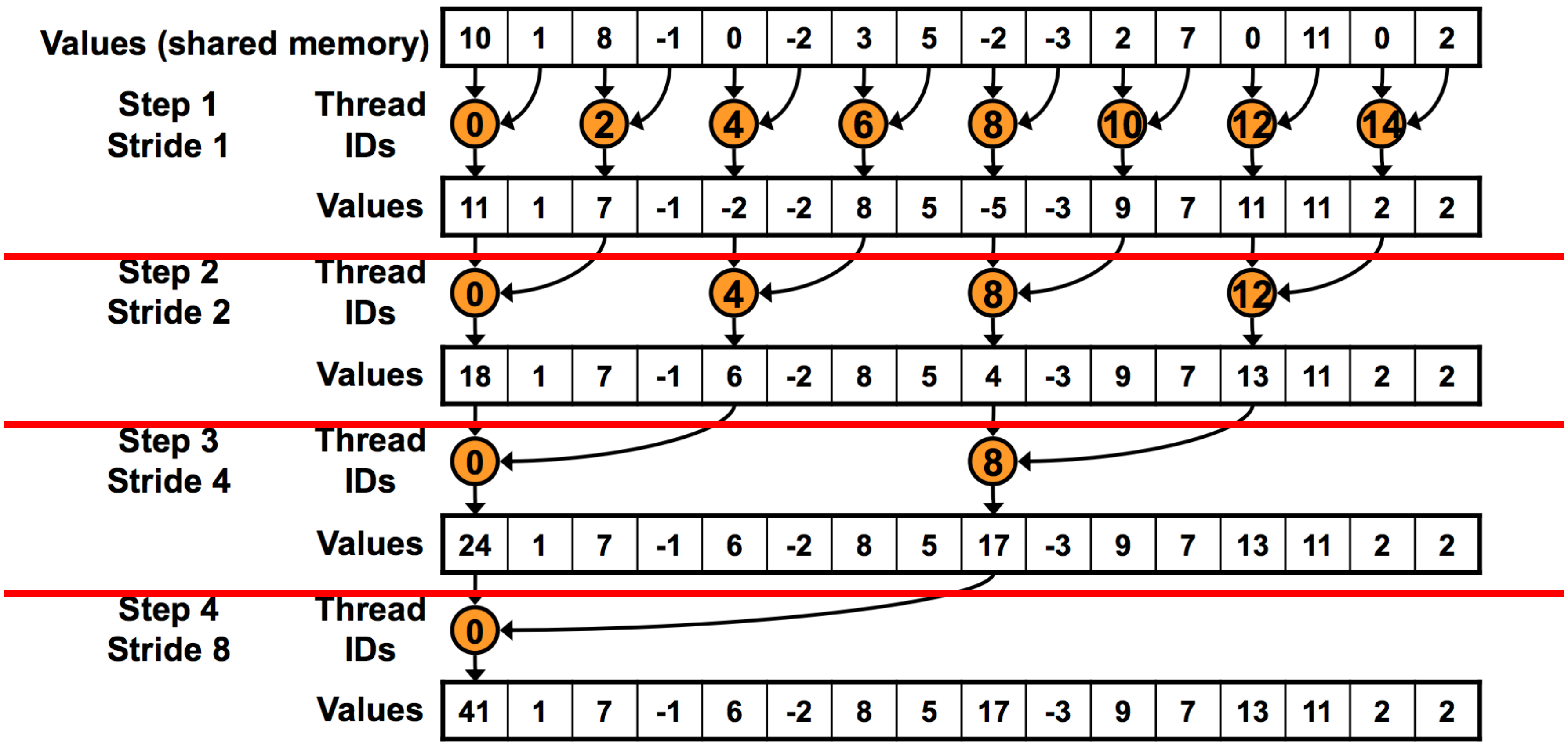
```
    //Use the first thread in the block to accumulate the results
    //of the other threads in said block
    if (threadIdx.x == 0) {
        for (unsigned int threadIndex = 1; threadIndex < blockDim.x;
                ++threadIndex){

            //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);
    }

}
```

“Binary tree” reduction



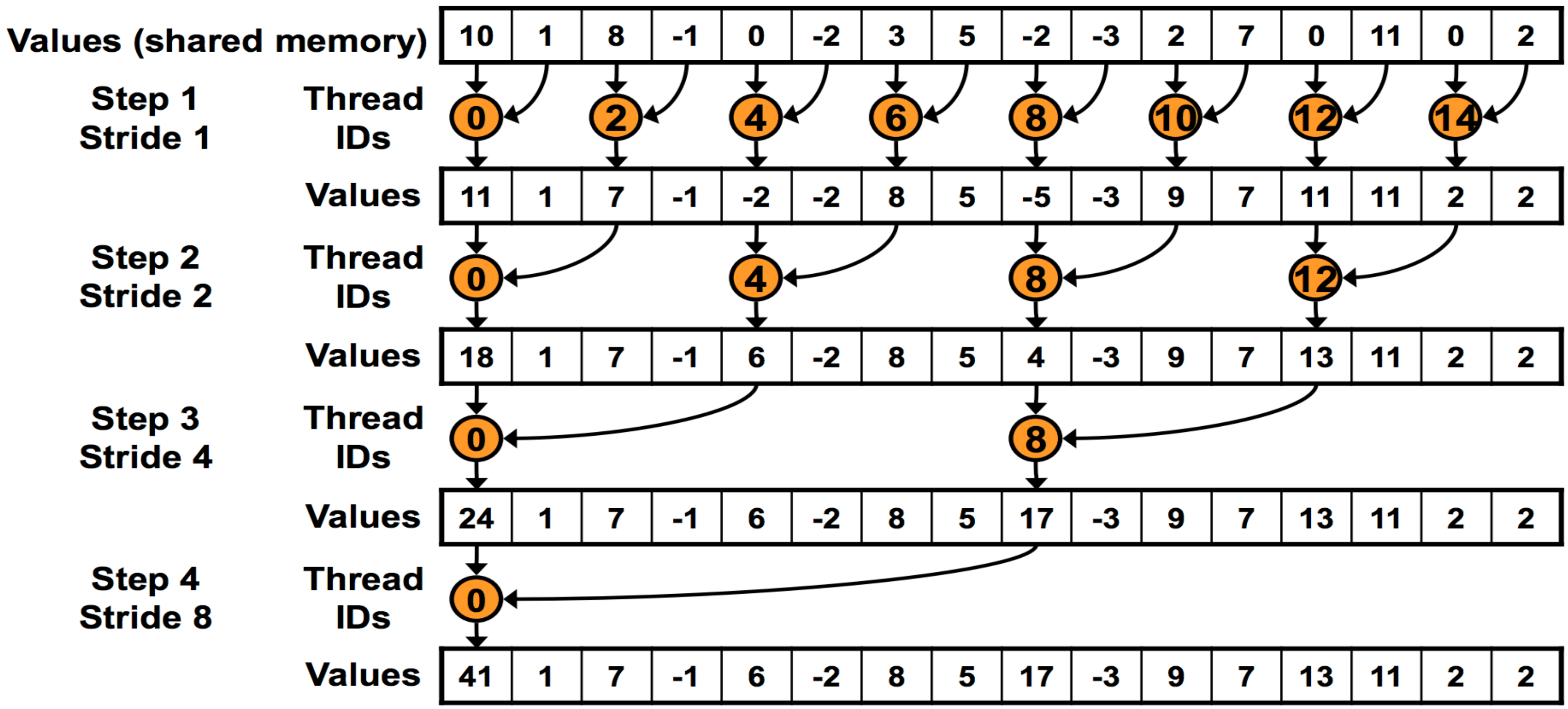
One thread atomicAdd’s
this to global result

“Binary tree” reduction



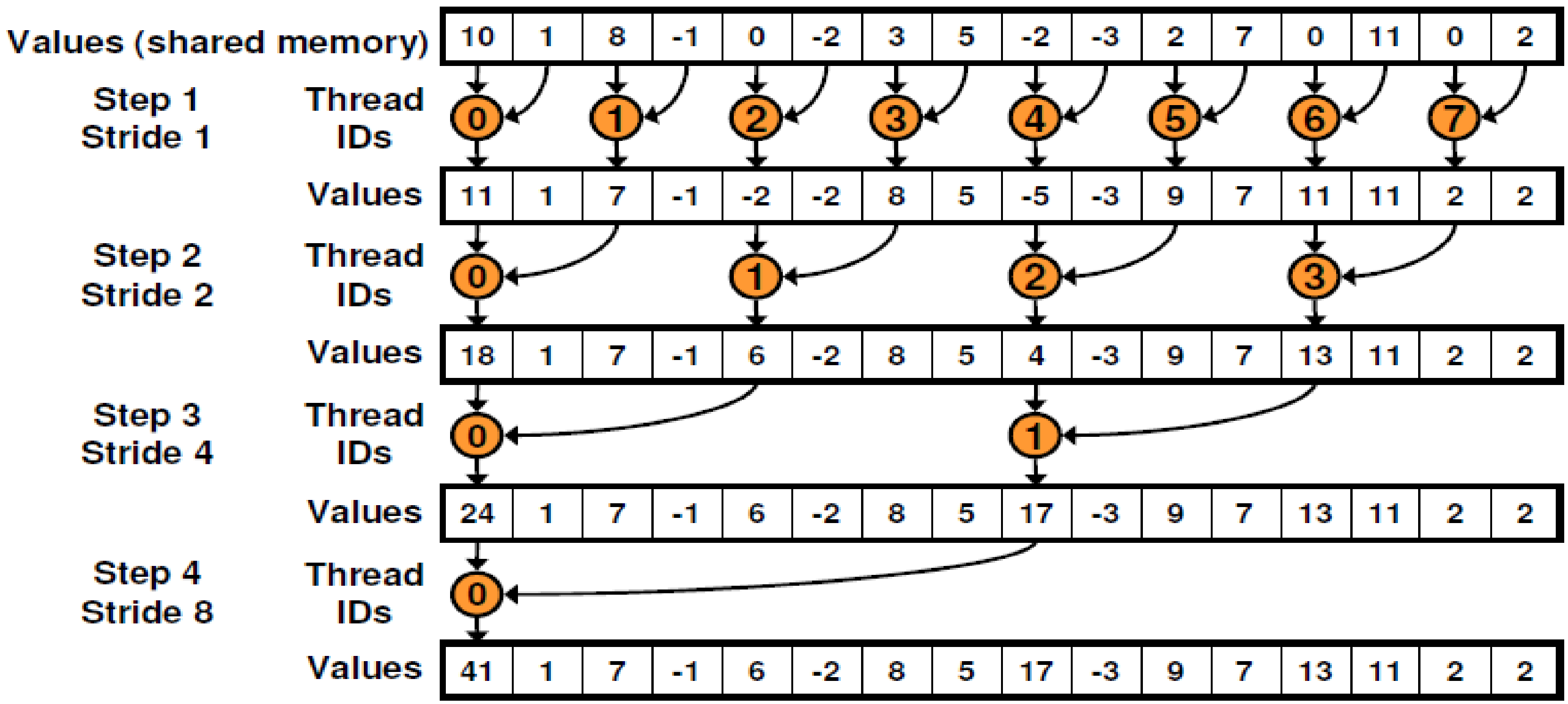
Use __syncthreads()
before proceeding!

“Binary tree” reduction

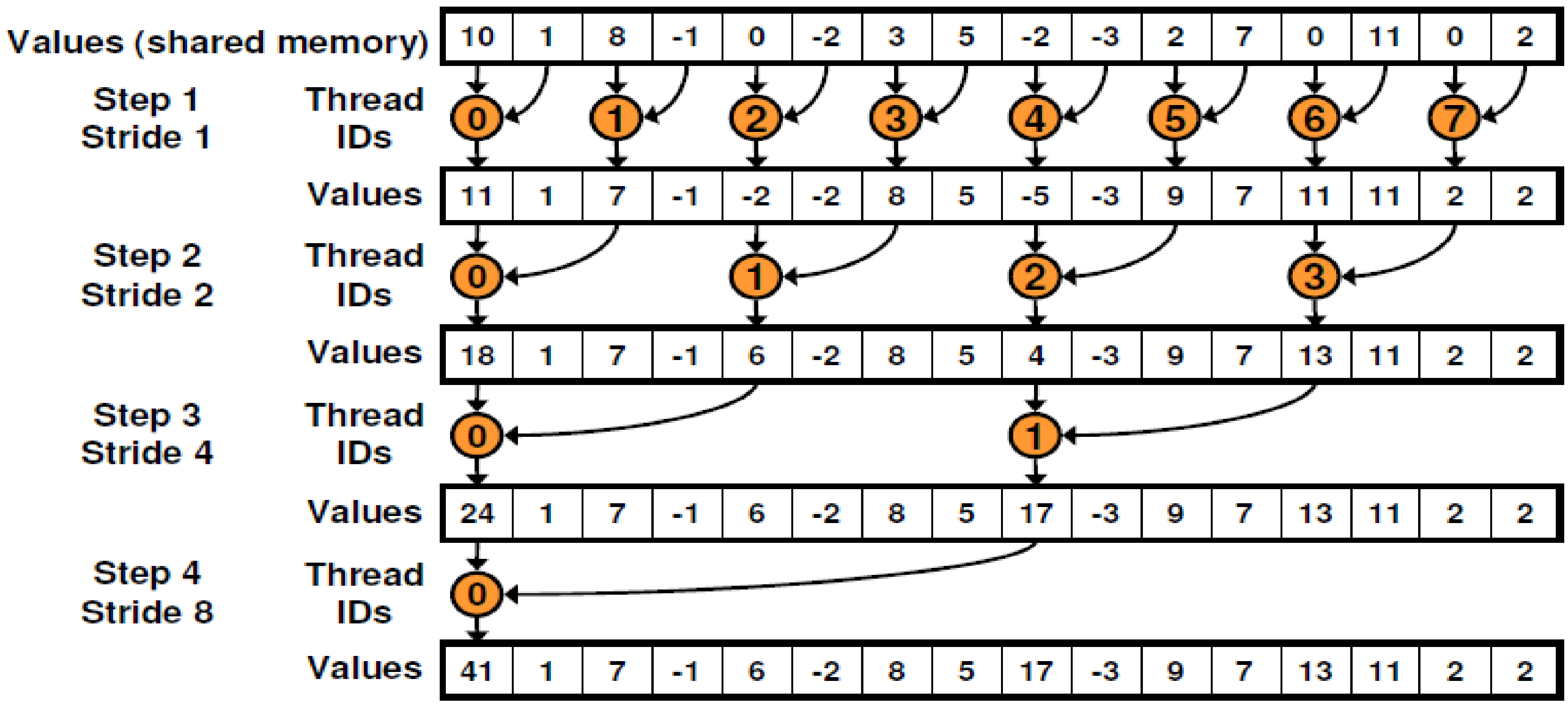


- Divergence!
  - Uses twice as many warps as necessary!

Non-divergent reduction

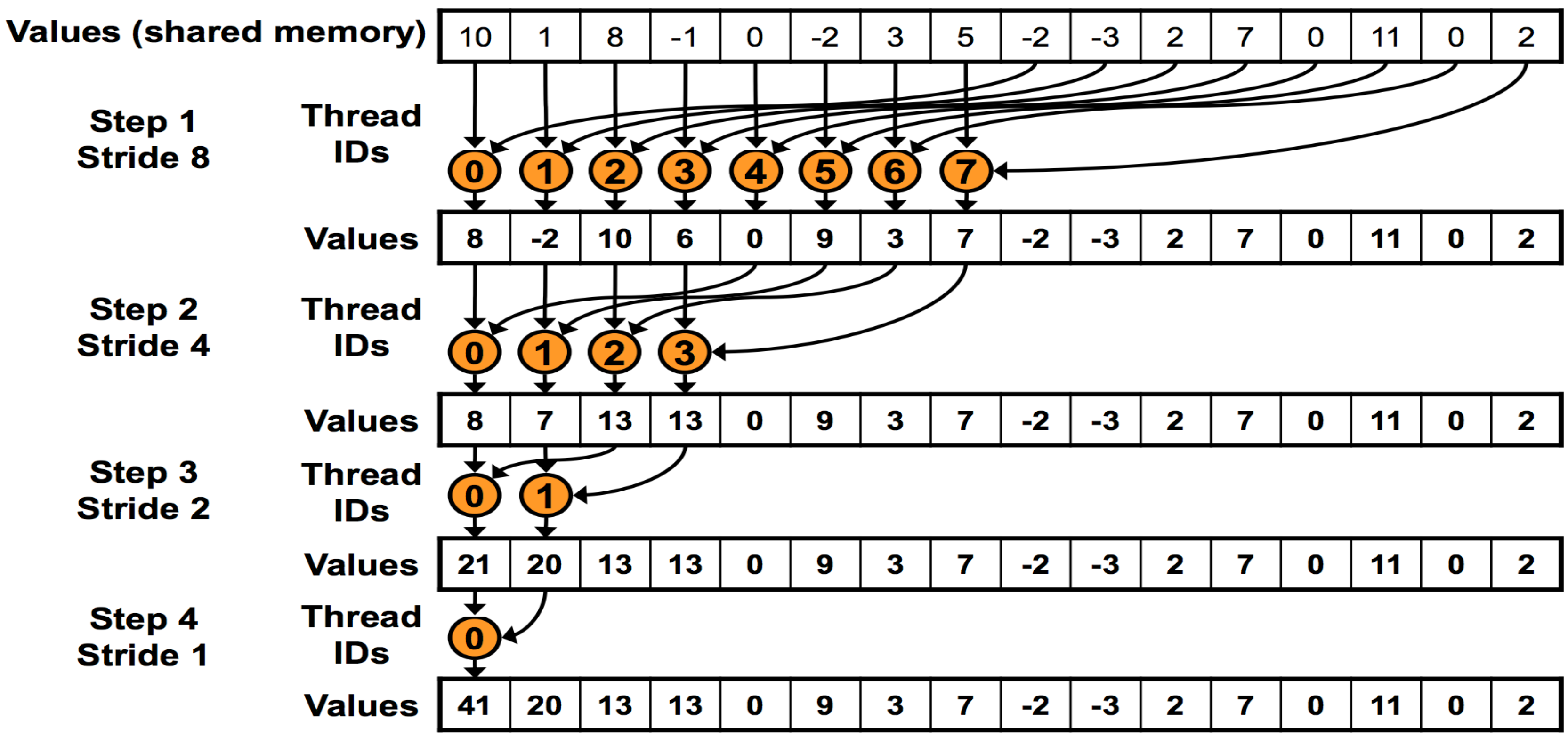


Non-divergent reduction



- Bank conflicts!
  - 1st iteration: 2-way,
  - 2nd iteration: 4-way (!), …

Sequential addressing



Reduction

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

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

Outline

- GPU-accelerated:
  - Reduction
  - Prefix sum
  - Stream compaction
  - Sorting (quicksort)

Prefix Sum

- Given input sequence x[n], produce sequence

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

  – e.g. x[n] = (1, 2, 3, 4, 5, 6)

    -> y[n] = (1, 3, 6, 10, 15, 21)

- Recurrence relation:

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

Prefix Sum

- Given input sequence x[n], produce sequence

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

  - e.g. x[n] = (1, 1, 1, 1, 1, 1, 1)

    -> y[n] = (1, 2, 3, 4, 5, 6, 7)

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

    -> y[n] = (1, 3, 6, 10, 15, 21)

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

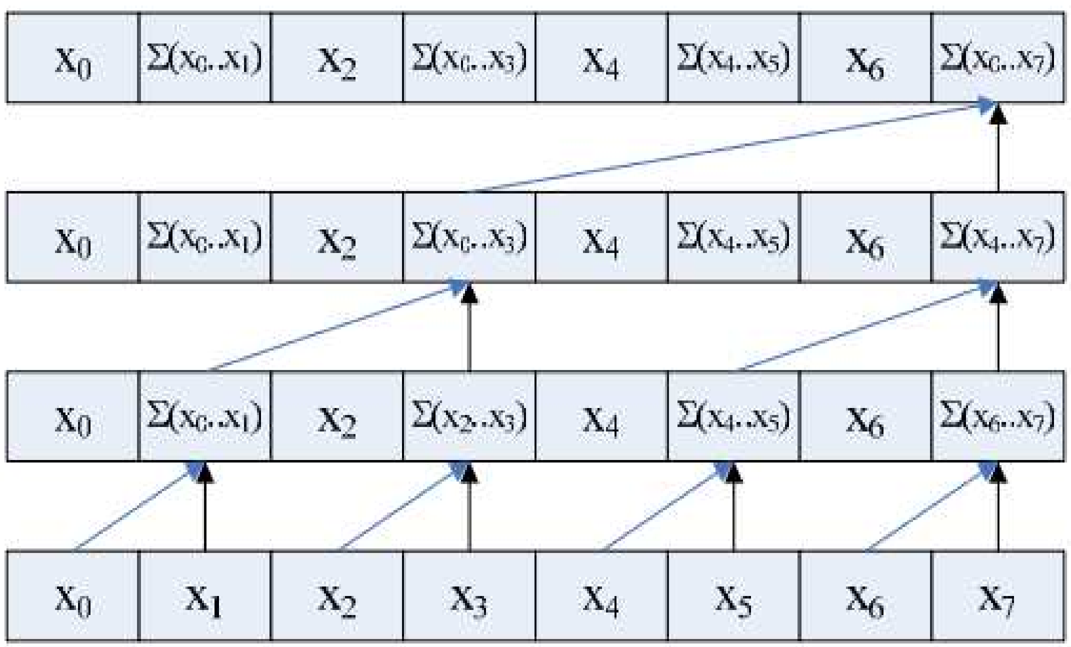
- Recurrence relation:

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

  - 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]

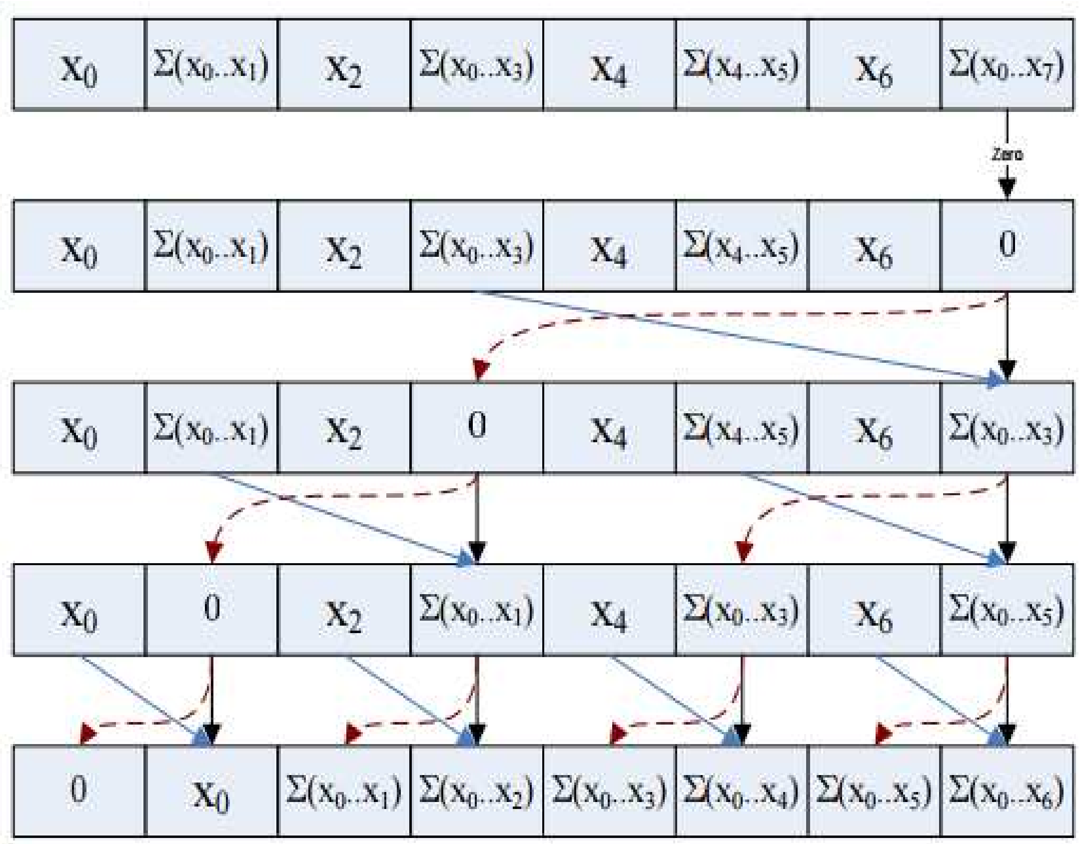
We want:

[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)

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]

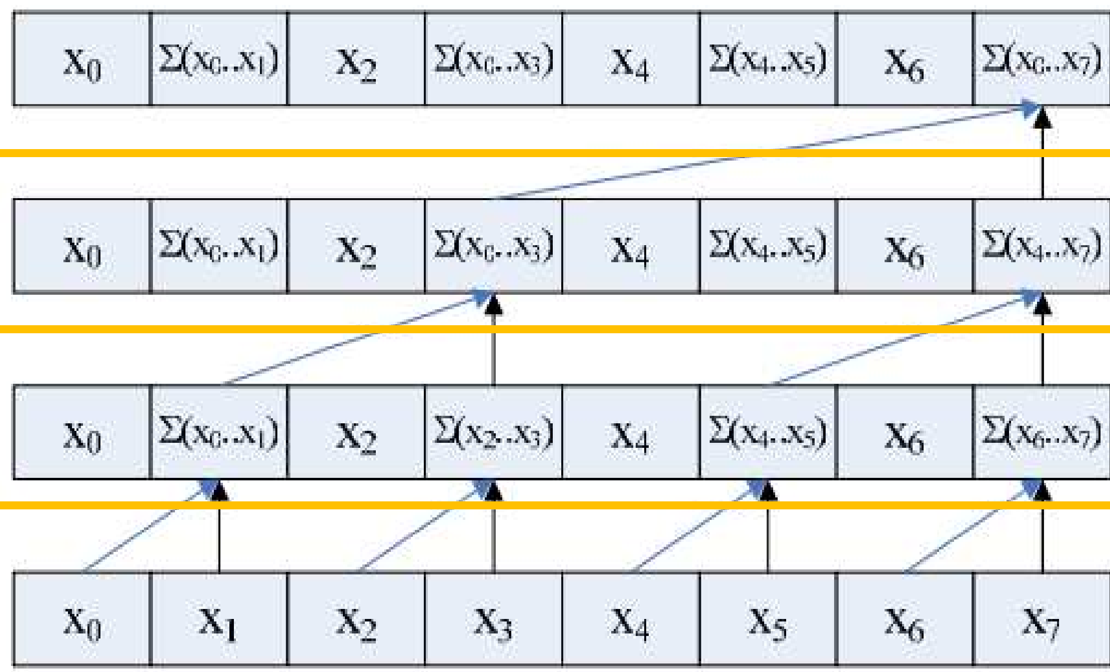
Final result

[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!

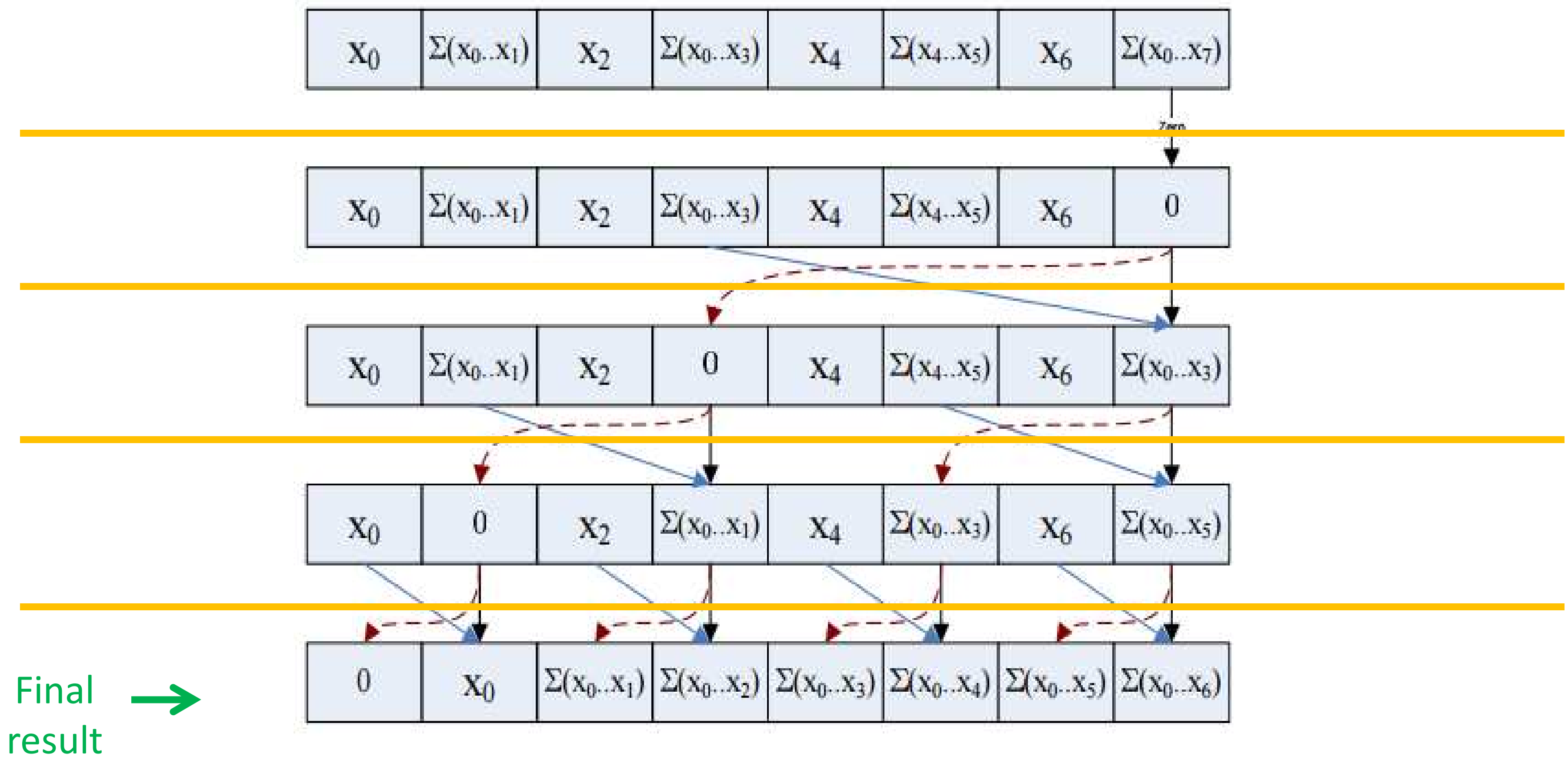


Original array

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

Prefix Sum (Down-Sweep)

Use __synchthreads() before proceeding!



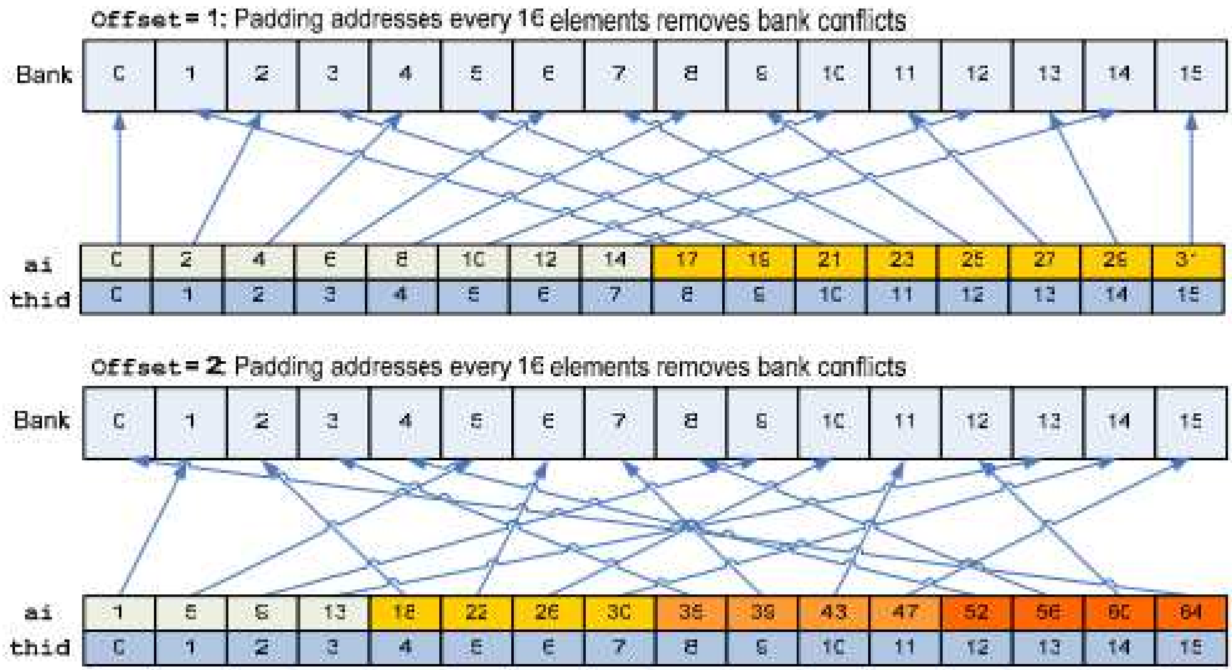
(University of Michigan EECS,
http://www.eecs.umich.edu/courses/eecs570/hw/parprefix.pdf

Prefix sum

- Bank conflicts!
  - 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

- Why does the prefix sum matter?

Outline

- GPU-accelerated:
  - Reduction
  - Prefix sum
  - Stream compaction
  - Sorting (quicksort)

Stream Compaction

- Problem:
  - Given array A, produce subarray of A defined by boolean condition

  - e.g. given array:

2	5	1	4	6	3

    - Produce array of numbers > 3

5	4	6

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

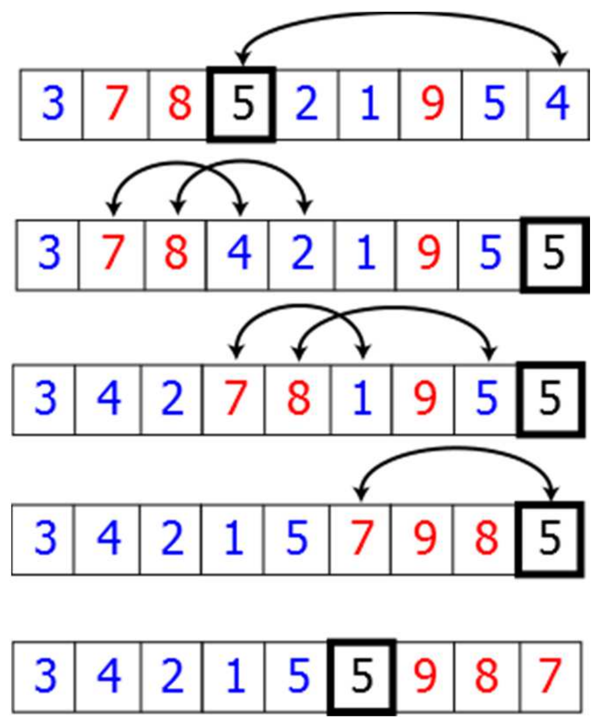
Outline

- GPU-accelerated:
  - Reduction
  - Prefix sum
  - Stream compaction
  - Sorting (quicksort)

GPU-accelerated quicksort

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

Sequential version



- Pseudocode:

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

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

- 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