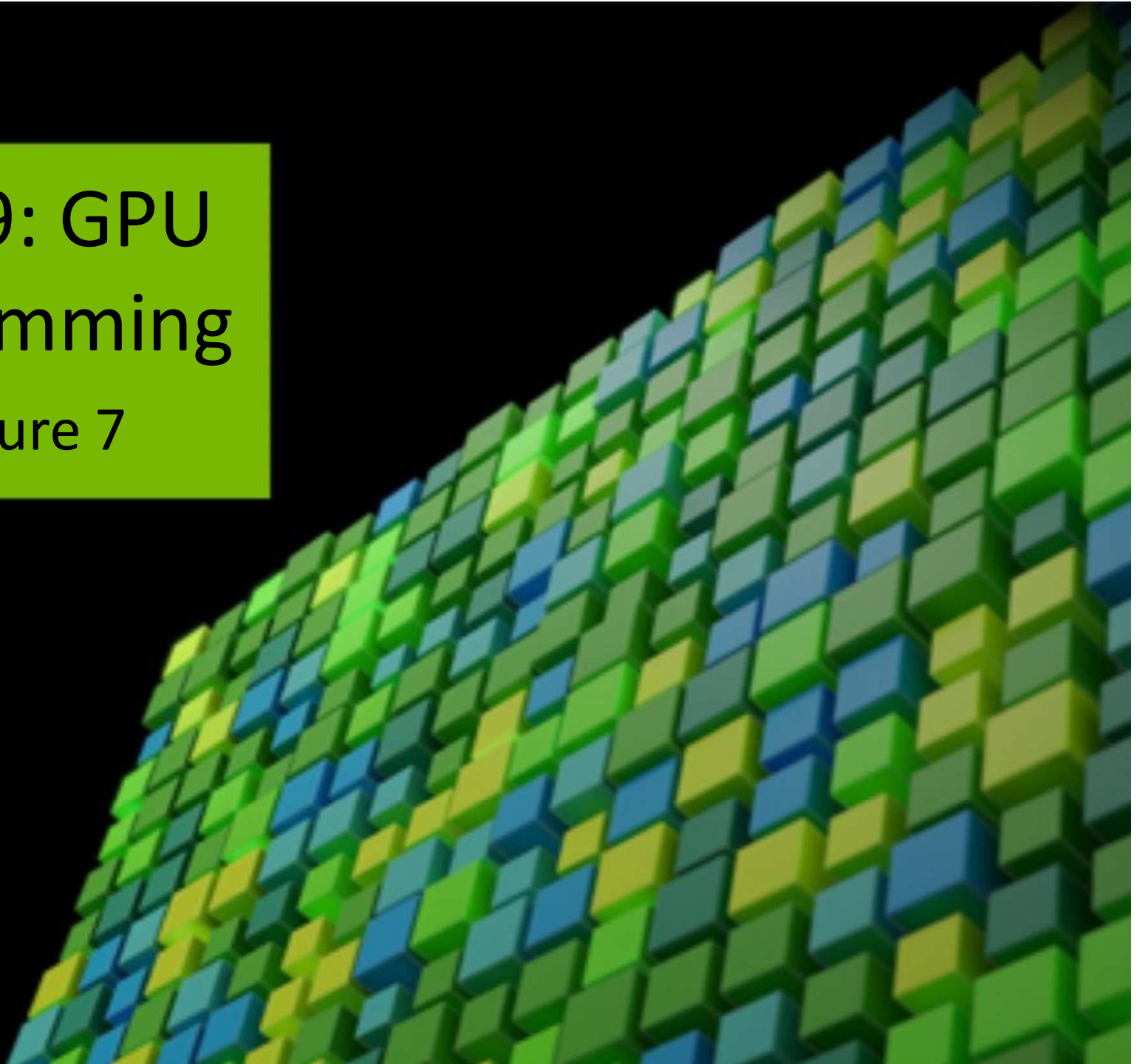


CS 179: GPU Programming

Lecture 7



Week 3

- Goals:
 - Advanced GPU-accelerated algorithms
 - CUDA libraries and tools

This Lecture

- GPU-acceleratable algorithms:
 - **Reduction**
 - Prefix sum
 - Stream compaction
 - Sorting (quicksort)

Elementwise 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

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;
}
```

Reduction Example

Problem: SUM(A[])

- CPU code:

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

- GPU Pseudocode:

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

Naïve Reduction

Problem: Sum of Array

- Serial Recombination causes speed reduction with GPUs, especially with higher number of threads
- GPU must use atomic functions for mutex

- atomicCAS
- atomicAdd

```
__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);
}
```

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);
}
```

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();
}
```

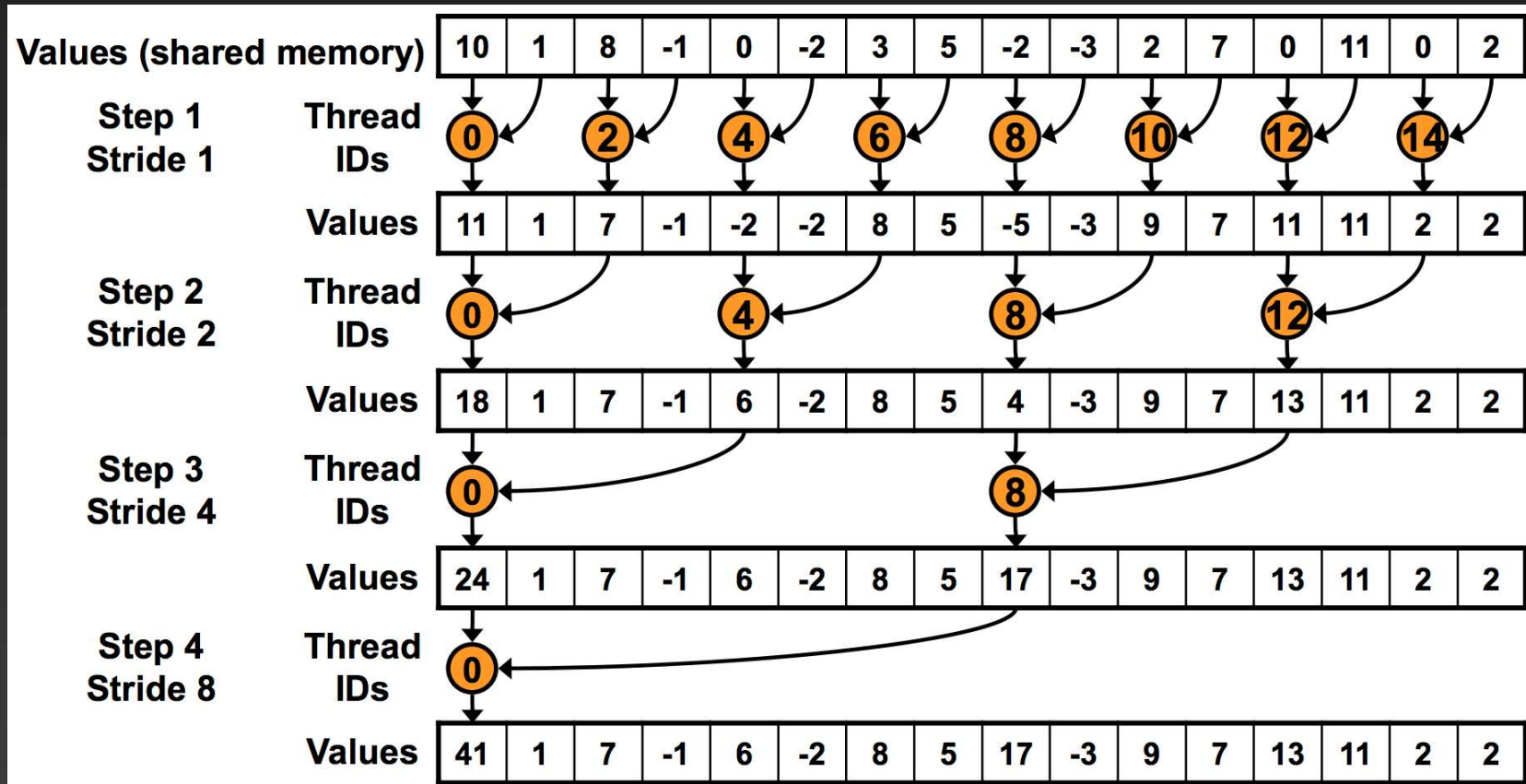
Shared memory accumulation (2)

```
//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 threadIdx = 1; threadIdx < blockDim.x;
        ++threadIdx){

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

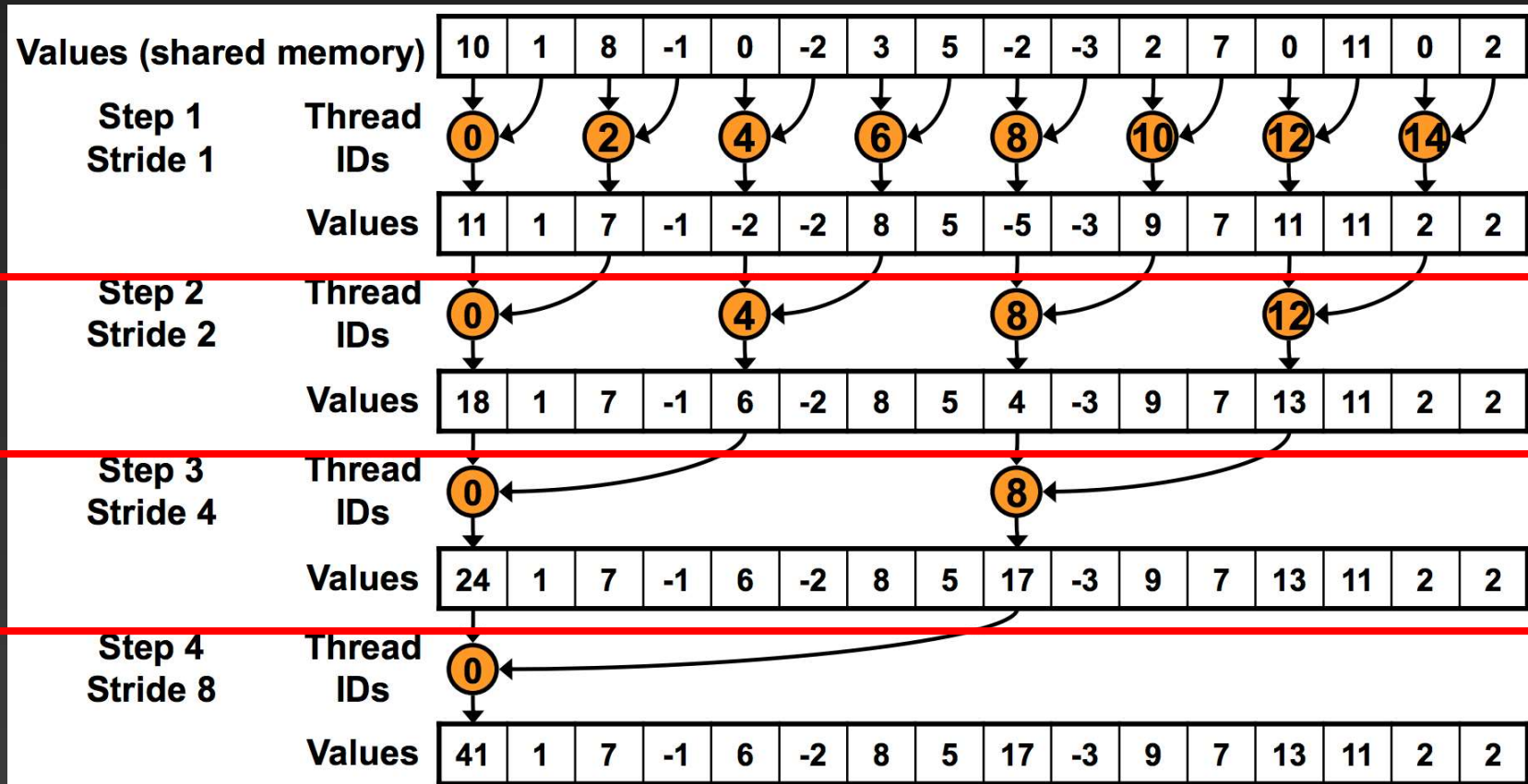
    //Now we finally accumulate
    atomicAdd(output, partial_sum);
}
}
```

“Binary tree” reduction



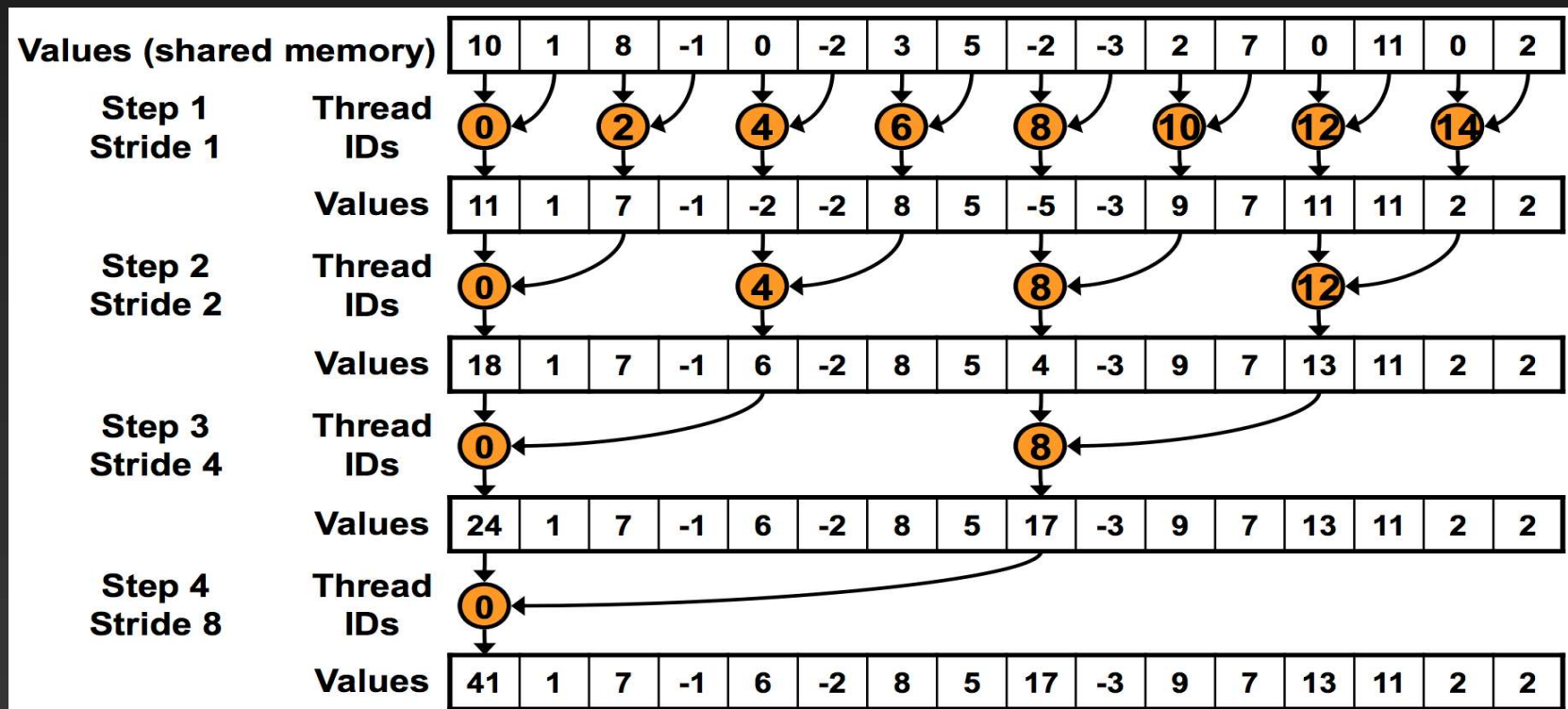
One thread atomicAdd's
this to global result

“Binary tree” reduction



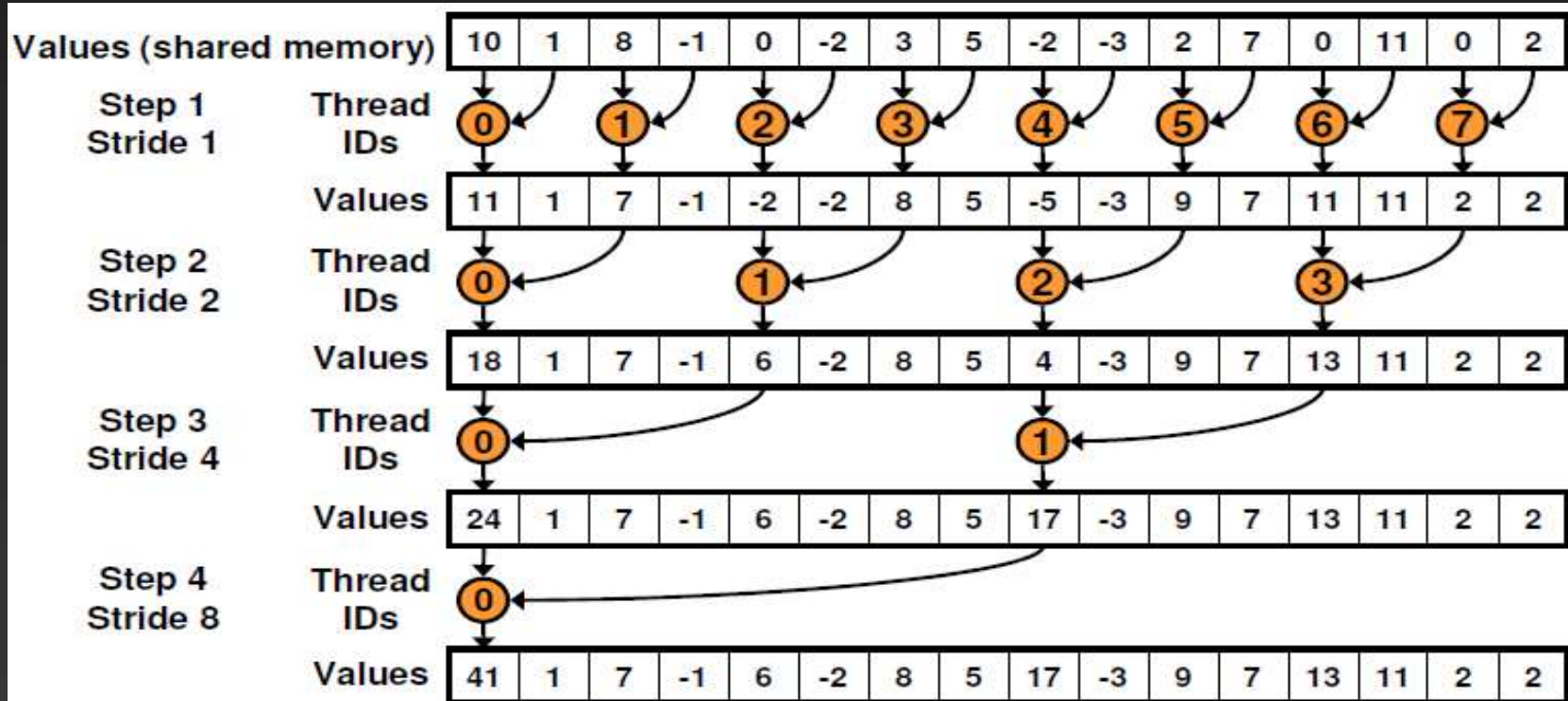
Use `__syncthreads()`
before proceeding!

“Binary tree” reduction

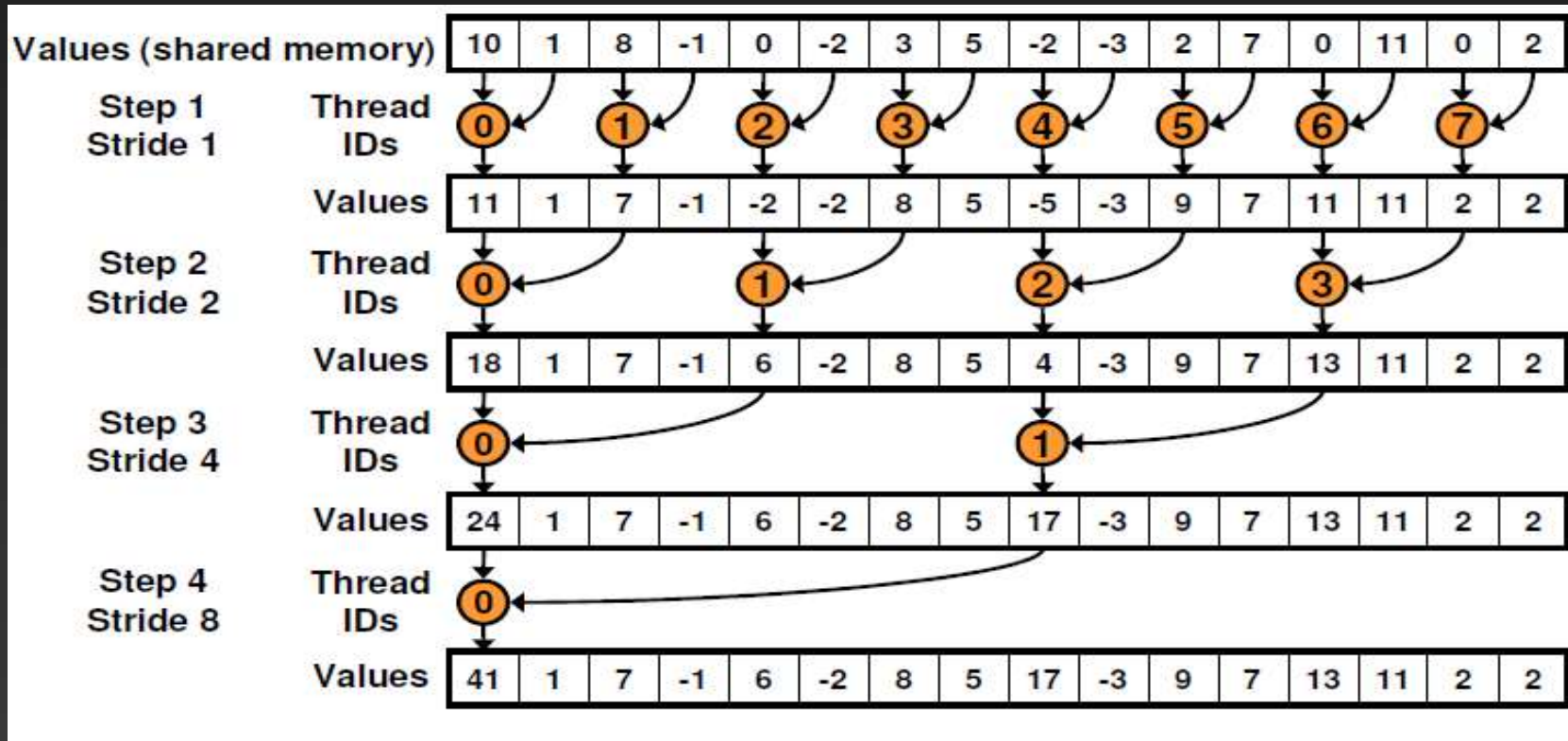


- Warp Divergence!
 - Odd threads won't even execute

Non-divergent reduction

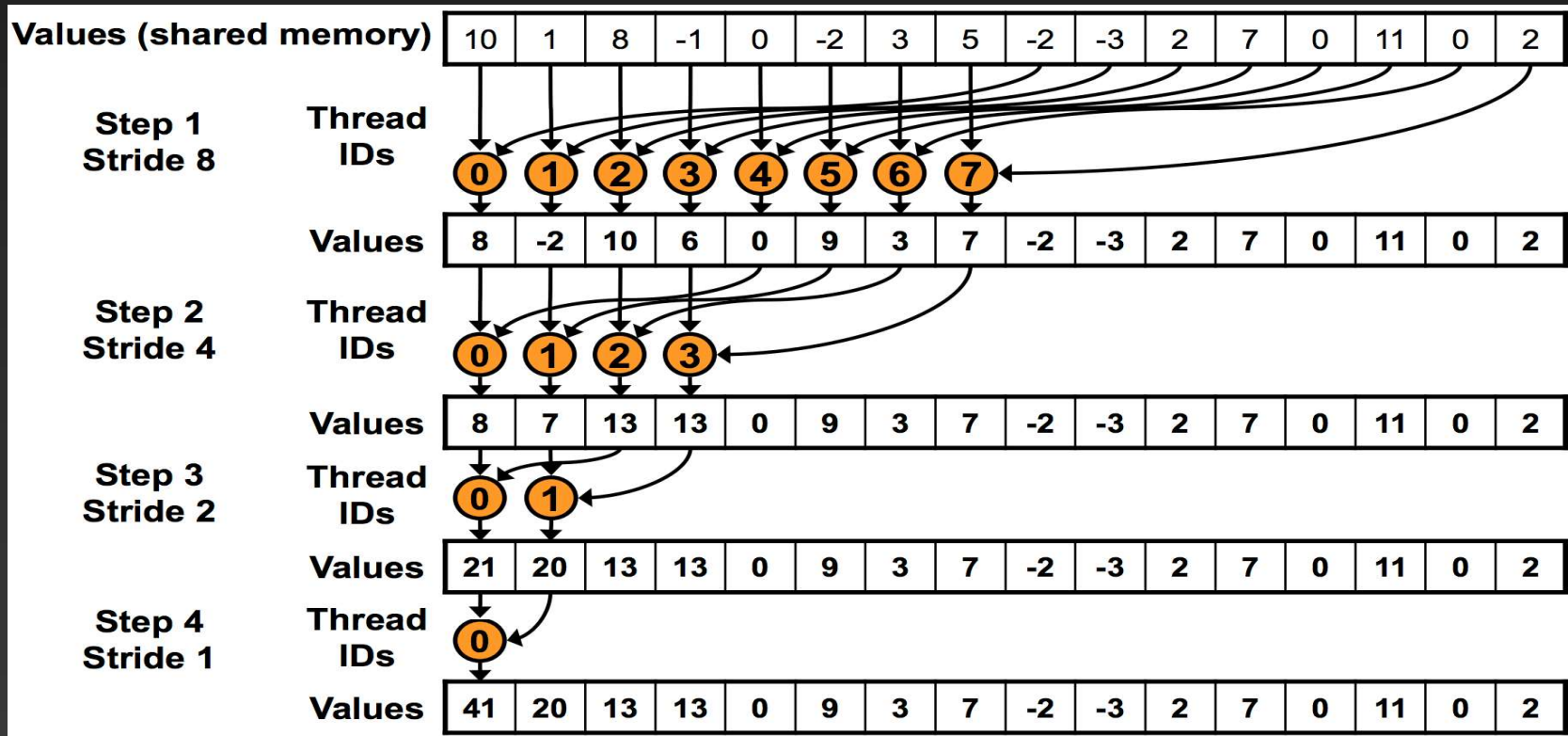


Non-divergent reduction



- Shared Memory Bank Conflicts!
 - 1st iteration: 2-way,
 - 2nd iteration: 4-way (!), ...

Sequential addressing



Sequential Addressing Automatically Resolves Bank Conflict Problems

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-acceleratable?

- Recall:

– $y[n] = x[n] + x[n - 1] + \dots + x[n - (K - 1)]$

» Easily parallelizable!

– $y[n] = c \cdot x[n] + (1 - c) \cdot y[n - 1]$

» Not so much

Prefix Sum

- Recurrence relation:

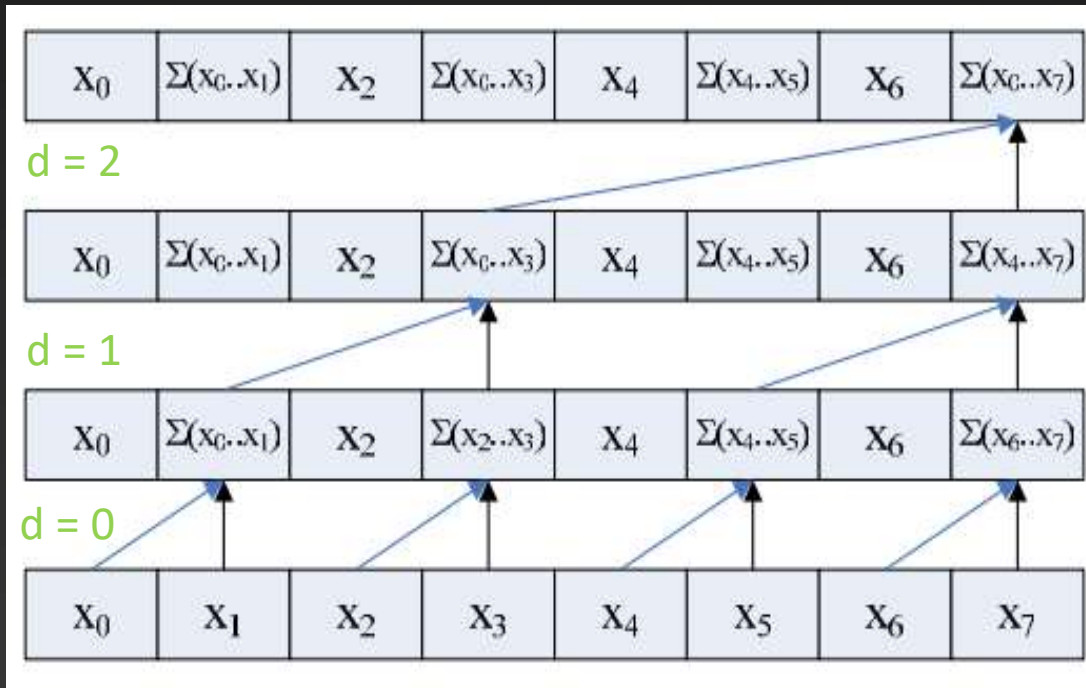
$$y[n] = y[n - 1] + x[n]$$

- 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:

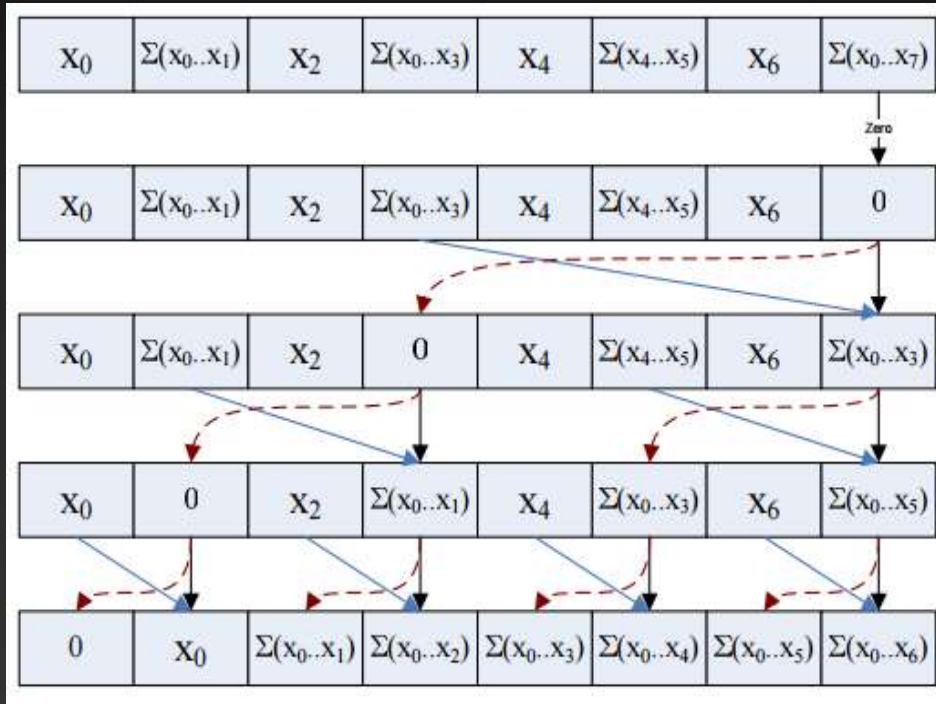
[0, 1, 3, 6, 10, 15, 21, 28]


```

4:   t = x[k + 2d - 1]
5:   x[k + 2d - 1] = x[k + 2d + 1 - 1]
6:   x[k + 2d + 1 - 1] = t + x[k + 2d + 1 - 1]

```

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]

$x[n-1] = 0$

for $d = \log_2(n) - 1$ down to 0 do

for all $k = 0$ to $n-1$ by 2^d+1 in parallel do

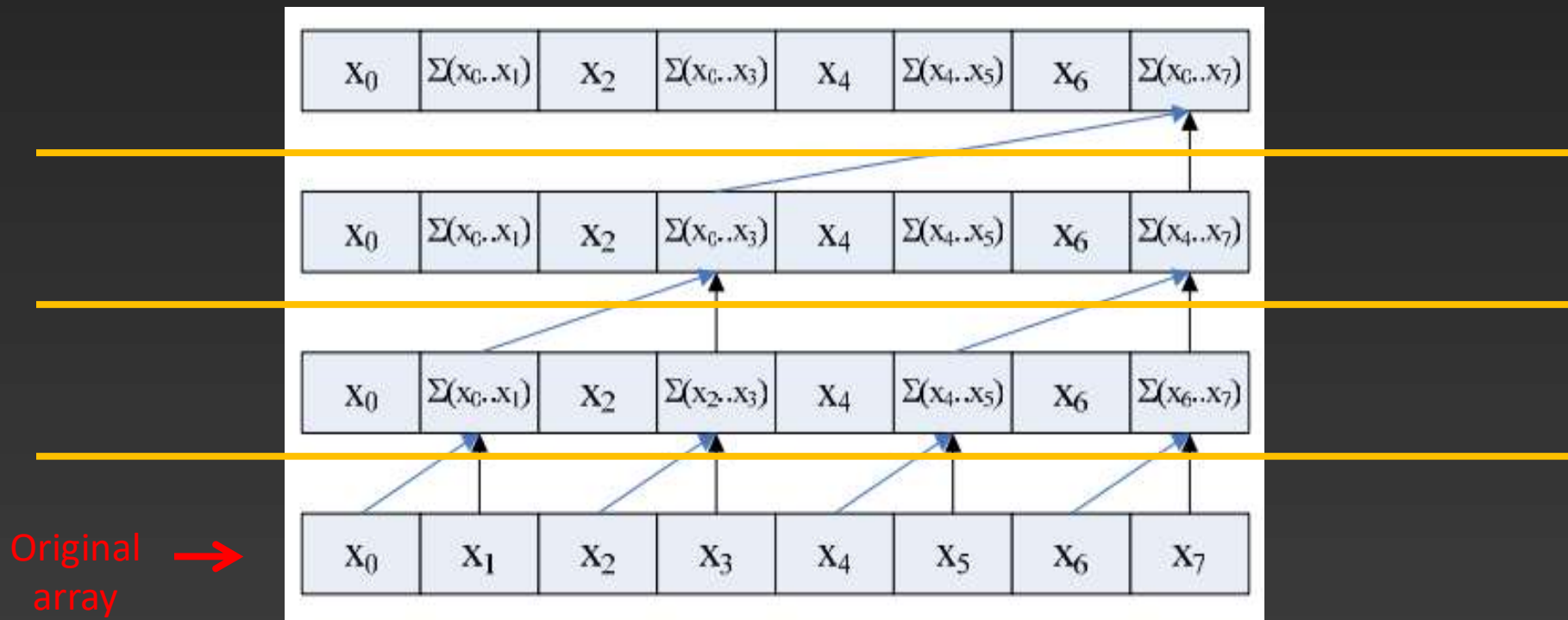
$t = x[k + 2^d - 1]$

$x[k + 2^d - 1] = x[k + 2^d]$

$x[k + 2^d] = t + x[k + 2^d]$

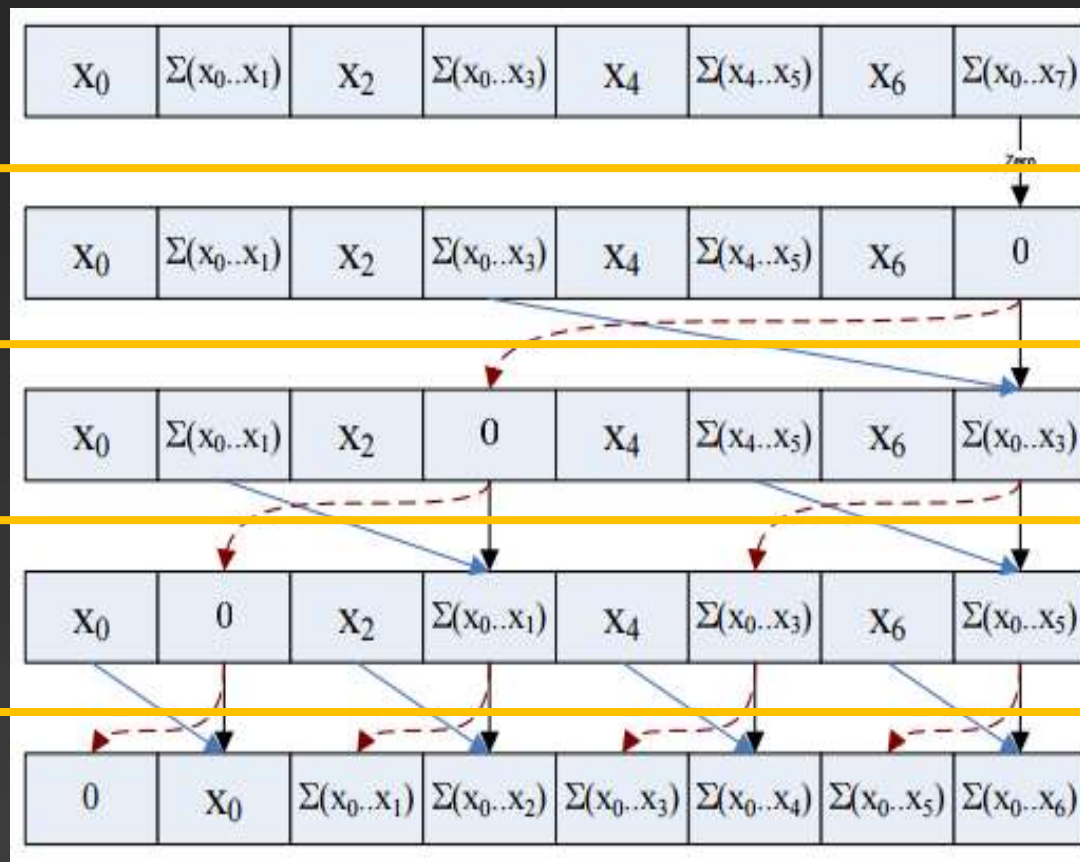
Prefix Sum (Up-Sweep)

Use `__syncthreads()`
before proceeding!



Prefix Sum (Down-Sweep)

Use `__syncthreads()`
before proceeding!



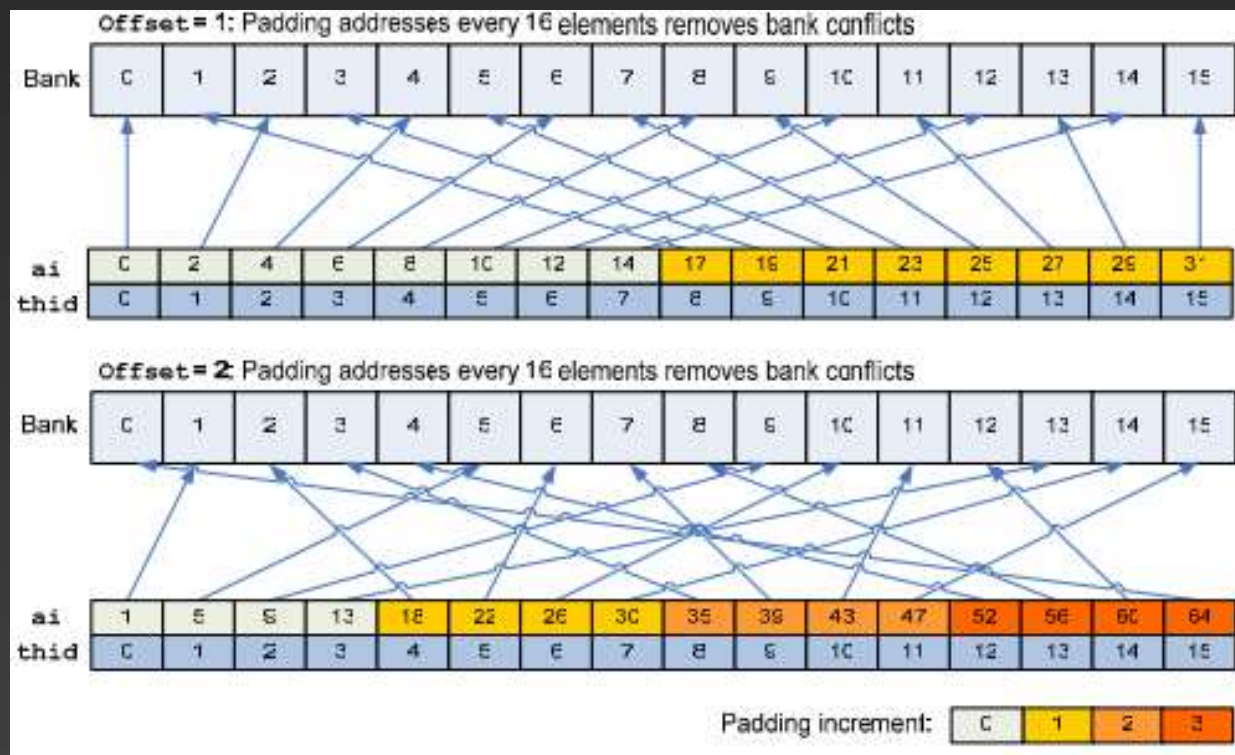
Final result →

Prefix sum

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

Prefix sum

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



Prefix Sum

- http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html -- See Link for a More In-Depth Explanation of Up-Sweep and Down-Sweep

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

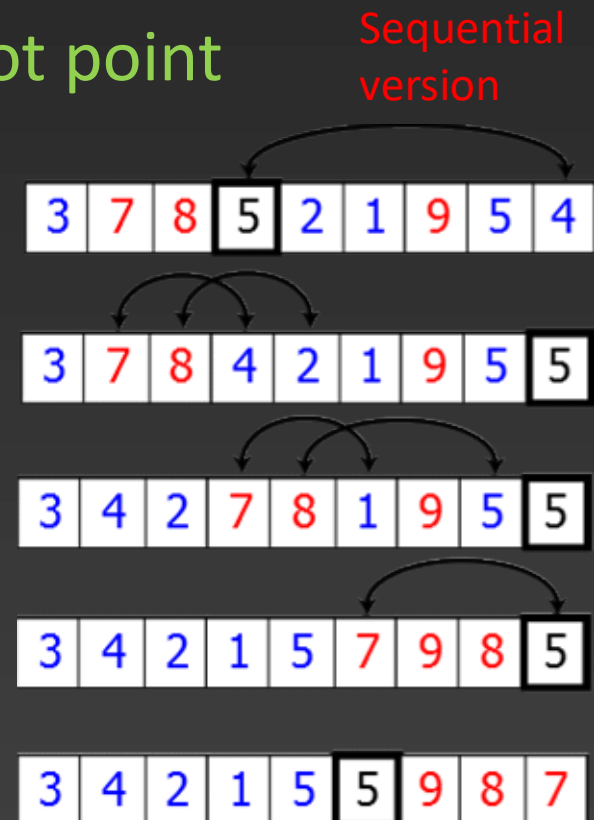
- GPU-accelerated:
 - Reduction
 - Prefix sum
 - Stream compaction
 - **Sorting (quicksort)**

GPU-accelerated quicksort

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

- 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
 - Highly Recommend Reading [This](#) Guide to CUDA Optimization, with a Reduction Example