Parallel Computing with GPUs

Parallel Patterns
Part 3 - Scan

The University Of Sheffield.

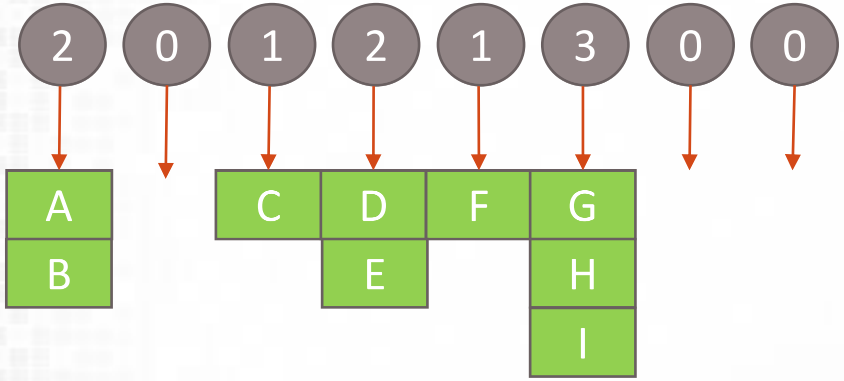
Dr Paul Richmond
http://paulrichmond.shef.ac.uk/teaching/COM4521/

This Lecture (learning objectives)

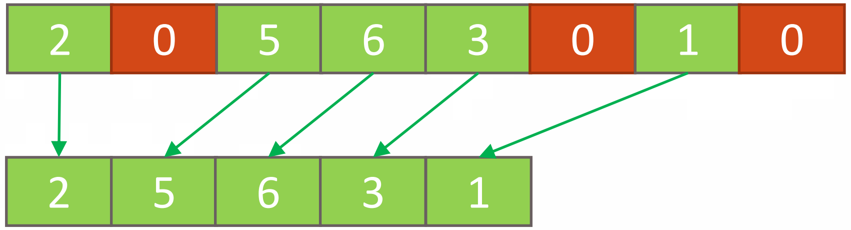
- ❑Scan
    - ❑Give motivating examples of parallel prefix sum (scan)
    - ❑Describe the serial and parallel approaches towards scan
    - ❑Compare block level and atomic approaches to the parallel prefix sum algorithm

What is scan?

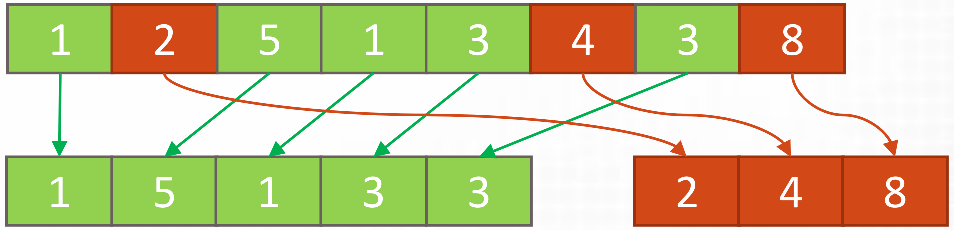
- Consider the following …



Output variable numbers of values per thread



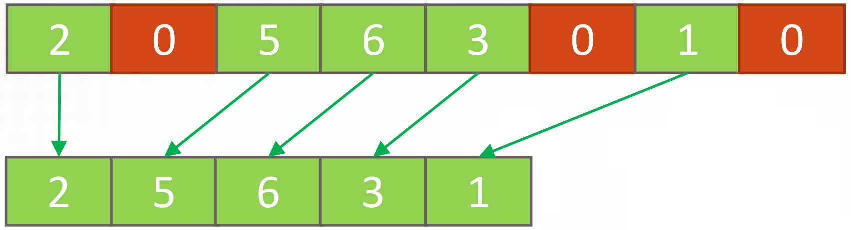
Remove empty elements from array (compact)



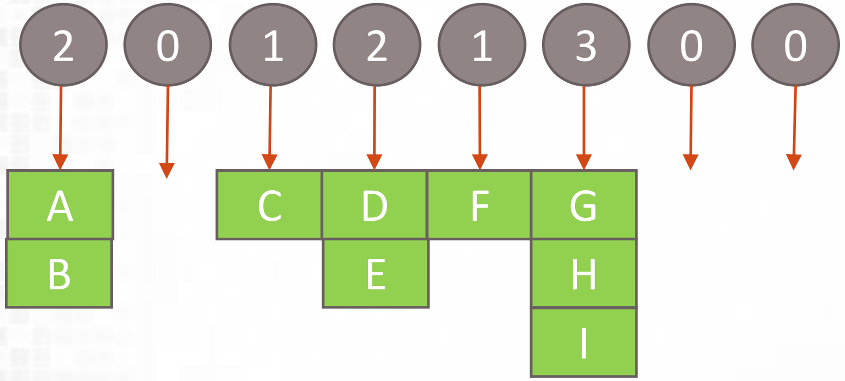
Split elements from array based on condition (split)

What is scan?

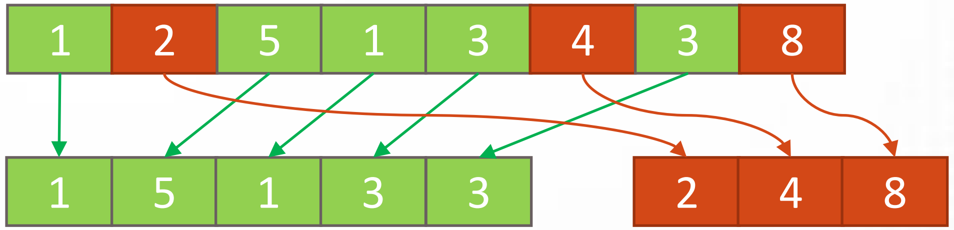
- Consider the following …



Remove empty elements from array (compact)



Output variable numbers of values per thread

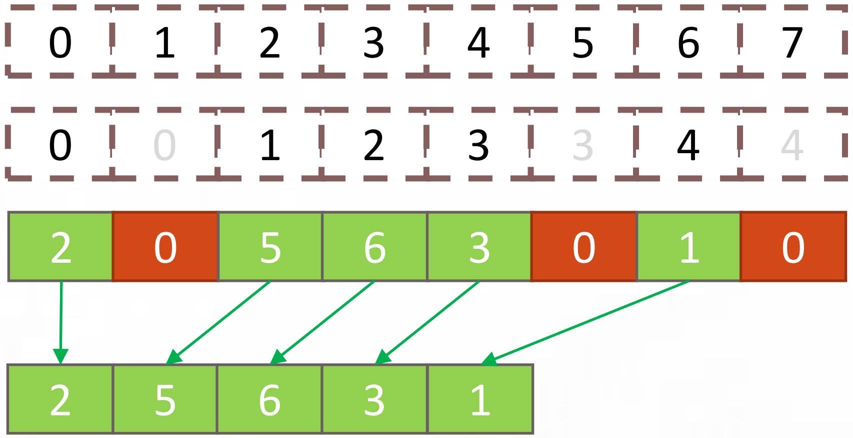


Split elements from array based on condition (split)

- Each has the same problem
  - Not even considered for sequential programs!
- Where to write output in parallel?

Parallel Prefix Sum (scan)

- Where to write output in parallel?
  - Each threads needs to know the output location(s) it can write to avoid conflicts.



0	1	2	3	4	5	6	7	Thread/Read index
0	0	1	2	3	3	4	4	Output/Write index – running sum of binary output state
2	0	5	6	3	0	1	0	Sparse data
2	5	6	3	1				Compacted data

- The solution is a parallel prefix sum (or scan)
  - Given the inputs $A = [a_0, a_1, …, a_{n-1}]$ and binary associate operator $\oplus$
  - $\mathrm{Scan}(A) = [0, a0, (a_0 \oplus a_1), …, (a_0 \oplus a_1 \oplus … \oplus a_{n-1})]$

Serial Parallel Prefix Sum Example

- E.g. Given the input and the addition operator
  - A= [2, 6, 2 ,4, 7, 2 ,1, 5]
  - Scan(A) = [0, 2, 2+6, 2+6+2, 2+6+2+4, …]
  - Scan(A) = [0, 2, 8, 10, 14, 21, 23, 24]
- More generally a serial implementation of an additive scan using a running sum looks like…

```
int A[8] = { 2, 6, 2, 4, 7, 2, 1, 5 };
int scan_A[8];
int running_sum = 0;
for (int i = 0; i < 8; ++i)
{
  scan_A[i] = running_sum;
  running_sum += A[i];
}
```

Serial Scan for Compaction

```
int Input[8] = { 2, 0, 5, 6, 3, 0, 1, 0 };
int A[8] =     { 2, 0, 5, 6, 3, 0, 1, 0 };
int scan_A[8];
int output[5]
int running_sum = 0;

for (int i = 0; i < 8; ++i){
  A[i] = Input>0;
}

for (int i = 0; i < 8; ++i){
  scan_A[i] = running_sum;
  running_sum += A[i];
}

for (int i = 0; i < 8; ++i){
  int input = Input[i];
  if (input > 0){
    int idx = scan[i];
    output[idx] = input;
  }
}
```

```
// generate scan input
// A = {1, 0, 1, 1, 1, 0, 1, 0}
```

```
// scan
// scan_A = {0, 1, 1, 2, 3, 4, 4, 5}
```

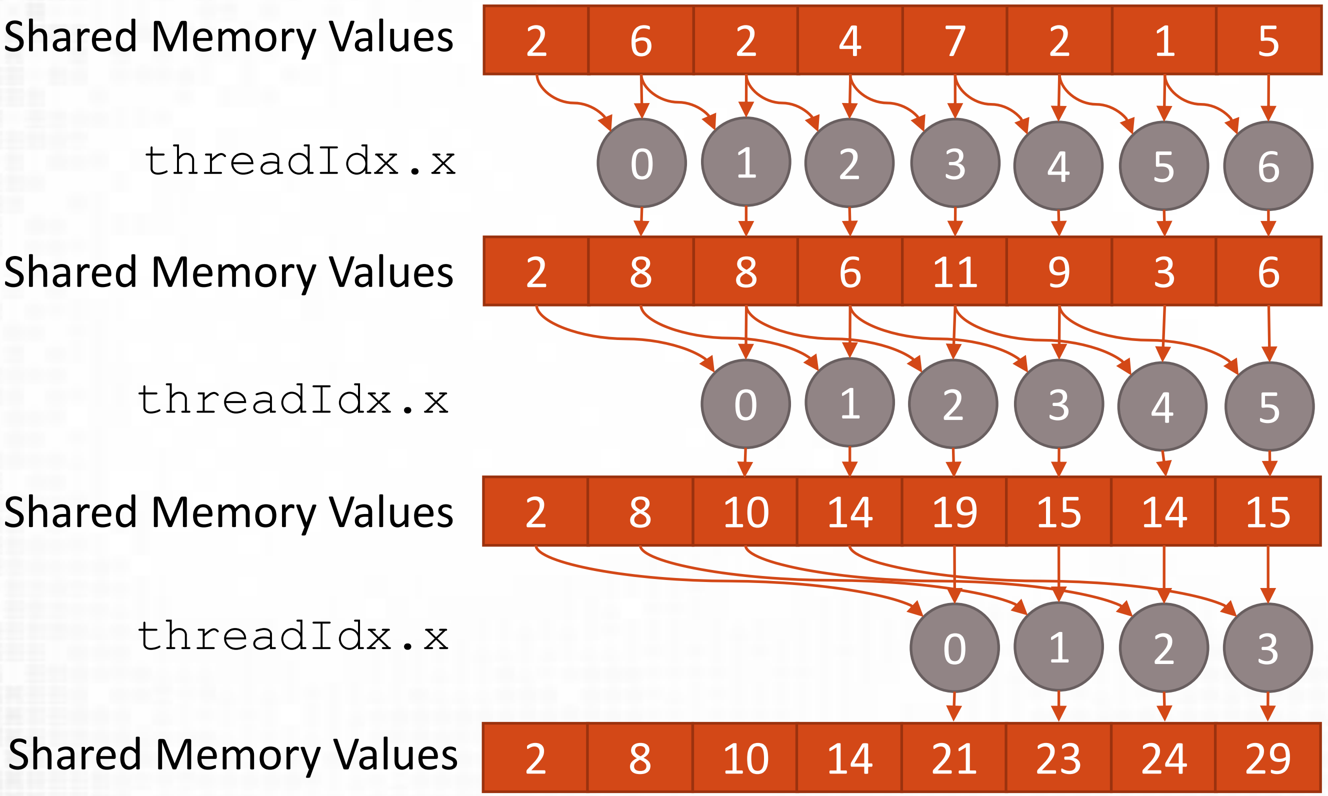
We could test either Input[i] or A[i] to find empty values

```
// scattered write
// output = {2, 5, 6, 3, 1}
```

The University Of Sheffield.

Parallel Local (Shared Memory) Scan

After Log(N) loops each sum has local plus preceding 2^n-1 values



Shared Memory Values: 2 6 2 4 7 2 1 5

threadIdx.x: 0 1 2 3 4 5 6

Shared Memory Values: 2 8 8 6 11 9 3 6

threadIdx.x: 0 1 2 3 4 5

Shared Memory Values: 2 8 10 14 19 15 14 15

threadIdx.x: 0 1 2 3

Shared Memory Values: 2 8 10 14 21 23 24 29

```
Loop 1
stride = 1
```

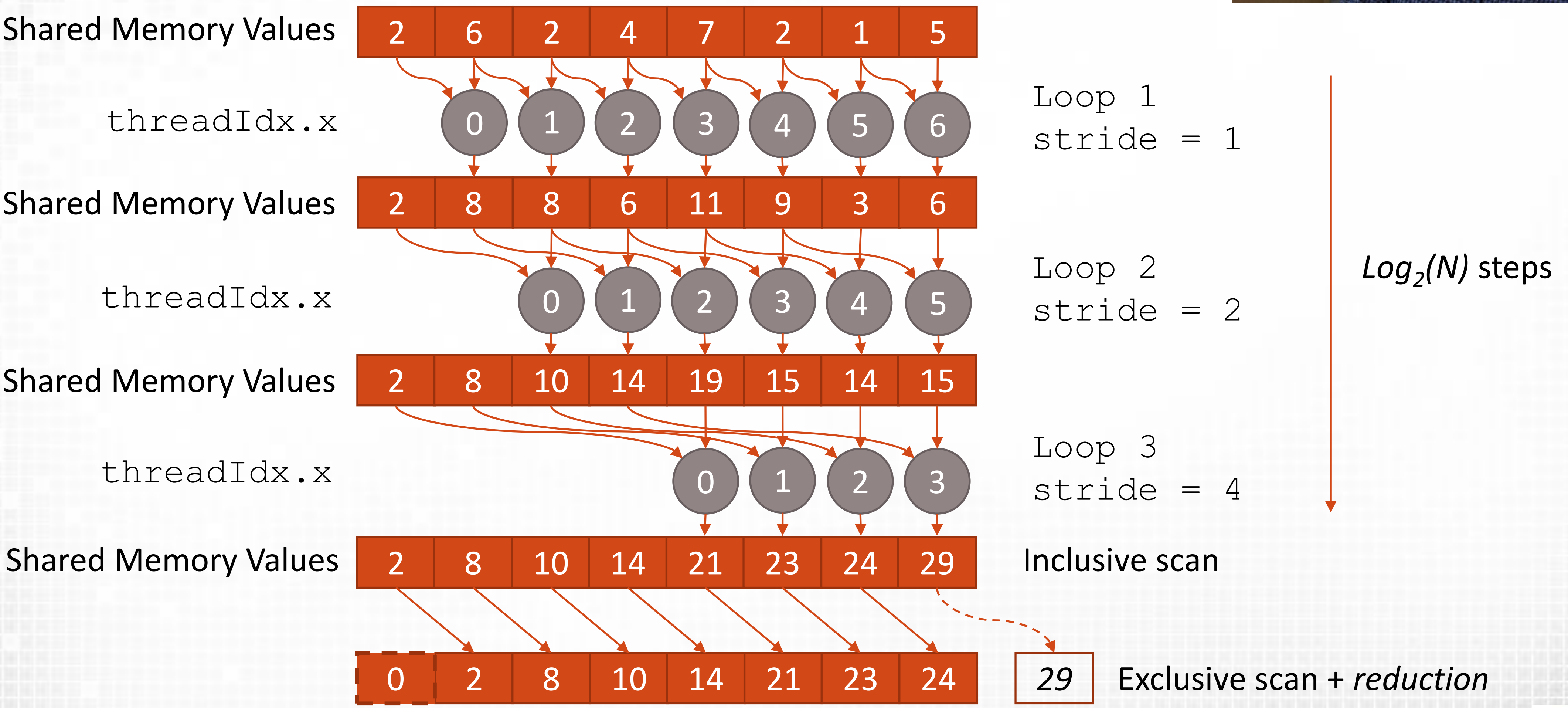
```
Loop 2
stride = 2
```

```
Loop 3
stride = 4
```

$Log_2(N)$ steps

Inclusive Scan

Parallel Local Scan



Shared Memory Values: 2 | 6 | 2 | 4 | 7 | 2 | 1 | 5

threadIdx.x: 0 | 1 | 2 | 3 | 4 | 5 | 6

Loop 1
stride = 1

Shared Memory Values: 2 | 8 | 8 | 6 | 11 | 9 | 3 | 6

threadIdx.x: 0 | 1 | 2 | 3 | 4 | 5

Loop 2
stride = 2

Shared Memory Values: 2 | 8 | 10 | 14 | 19 | 15 | 14 | 15

threadIdx.x: 0 | 1 | 2 | 3

Loop 3
stride = 4

Log$_2$(N) steps

Shared Memory Values: 2 | 8 | 10 | 14 | 21 | 23 | 24 | 29

Inclusive scan

0 | 2 | 8 | 10 | 14 | 21 | 23 | 24

29 Exclusive scan + *reduction*

Implementing Local Scan with Shared Memory

```
__global__ void scan(float *input) {
  extern __shared__ float s_data[];
  s_data[threadIdx.x] = input[threadIdx.x + blockIdx.x*blockDim.x];

  for (int stride = 1; stride<blockDim.x; stride<<=1) {
    __syncthreads();
    float s_value = (threadIdx.x >= stride) ? s_data[threadIdx.x - stride] : 0;
    __syncthreads();
    s_data[threadIdx.x] += s_value;
  }

  //something with global results?
}
```

❑No bank conflicts (stride of 1 between threads)

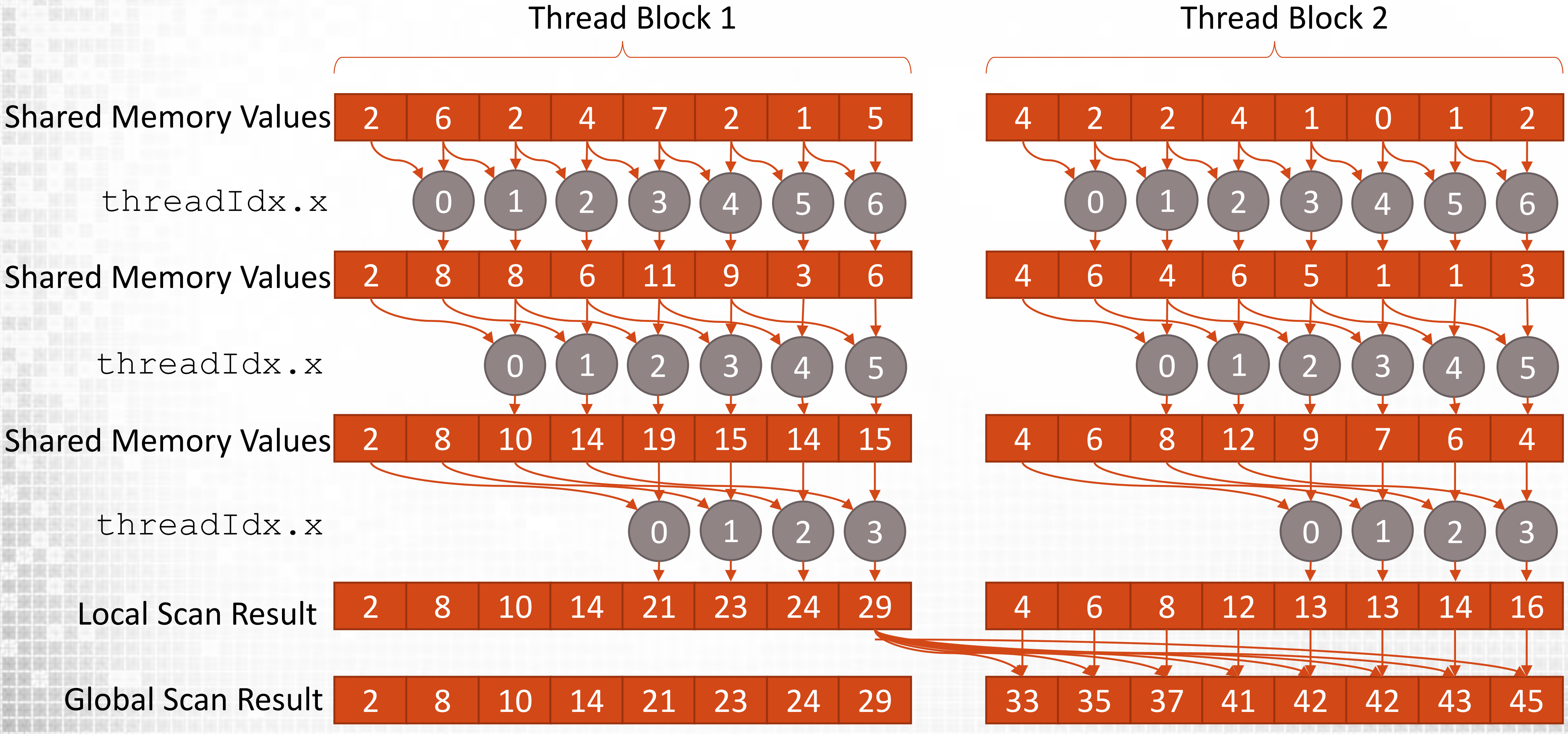
❑Synchronisation required between read and write

Implementing Local Scan (at warp level)

```
__global__ void scan(float *input) {
  __shared__ float s_data[32];
  float val1, val2;

  val1 = input[threadIdx.x + blockIdx.x*blockDim.x];

  for (int s = 1; s < 32; s <<= 1) {
    val2 = __shfl_up(val1, s);
    if (threadIdx.x % 32 >= s)
      val1 += val2;
  }

  //store warp level results}
```

- Exactly the same as the block level technique but at warp level
- Warp prefix sum is in `threadIdx.x%32==31`
- Either use shared memory to reduce between warps
  - Or consider the following global scan approaches.

Implementing scan at Grid Level



	Thread Block 1								Thread Block 2							
Shared Memory Values	2	6	2	4	7	2	1	5	4	2	2	4	1	0	1	2
threadIdx.x		0	1	2	3	4	5	6		0	1	2	3	4	5	6
Shared Memory Values	2	8	8	6	11	9	3	6	4	6	4	6	5	1	1	3
threadIdx.x			0	1	2	3	4	5			0	1	2	3	4	5
Shared Memory Values	2	8	10	14	19	15	14	15	4	6	8	12	9	7	6	4
threadIdx.x					0	1	2	3					0	1	2	3
Local Scan Result	2	8	10	14	21	23	24	29	4	6	8	12	13	13	14	16
Global Scan Result	2	8	10	14	21	23	24	29	33	35	37	41	42	42	43	45

Implementing scan at Grid Level

- ❑Same problem as reduction when scaling to grid level
    - ❑Each block is required to add the reduction value from proceeding blocks

- ❑Global scan therefore requires either;
    1. Recursive scan kernel on results of local scan
        - ❑ Additional kernel to add sums of proceeding blocks
    2. **Atomic Increments (next slides)**
        - ❑ **Increment a counter for block level results**
        - ❑ **Additional kernel to add sums of proceeding blocks to each value**

Global Level Scan (Atomics Part 1)

```
__device__ block_sums[BLOCK_DIM];

__global__ void scan(float *input, float *local_result) {
  extern __shared__ float s_data[];
  s_data[threadIdx.x] = input[threadIdx.x + blockIdx.x*blockDim.x];

  for (int stride = 1; stride<blockDim.x; stride<<=1) {
    __syncthreads();
    float s_value = (threadIdx.x >= stride) ? s_data[threadIdx.x - stride] : 0;
    __syncthreads();
    s_data[threadIdx.x] += s_value;
  }

  //store local scan result to each thread
  local_result[threadIdx.x + blockIdx.x*blockDim.x] = s_data[threadIdx.x];

  //atomic store to all proceeding block totals (e.g. blocks after this block)
  if (threadIdx.x == 0){
    for (int i=blockIdx.x+1; i<gridDim.x; i++)
      atomicAdd(&block_sums[i], s_data[blockDim.x-1]);
  }
}
```

Global Level Scan (Atomics Part 2)

❑After completion of the first kernel, block sums are all synchronised

❑Use first thread in block to load block total into shared memory

❑Increment local result

```
__device__ block_sums[BLOCK_DIM];

__global__ void scan_update(float *local_result, float *global_result) {
  extern __shared__ float block_total;
  int idx = threadIdx.x + blockIdx.x*blockDim.x;

  if (threadIdx.x == 0)
    block_total = block_sums[blockIdx.x];

  __syncthreads();

  global_result[idx] = local_result[idx]+block_total;
}
```

Summary

- ❑Scan
  - ❑Give motivating examples of parallel prefix sum (scan)
  - ❑Describe the serial and parallel approaches towards scan
  - ❑Compare block level and atomic approaches to the parallel prefix sum algorithm

Acknowledgements and Further Reading

- ❑ https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/
    - ❑ All about application of warp shuffles to reduction
- ❑ https://stanford-cs193g-sp2010.googlecode.com/svn/trunk/lectures/lecture_6/parallel_patterns_1.ppt
    - ❑ Scan material based loosely on this lecture
- ❑ http://docs.nvidia.com/cuda/samples/6_Advanced/reduction/doc/reduction.pdf
    - ❑ Reduction material is based on this fantastic lecture by Mark Harris (NVIDIA)