Parallel Computing with GPUs: Parallel Patterns

Dr Mozhgan Kabiri Chimeh
http://mkchimeh.staff.shef.ac.uk/teaching/COM4521

The University Of Sheffield.

NVIDIA GPU RESEARCH CENTER

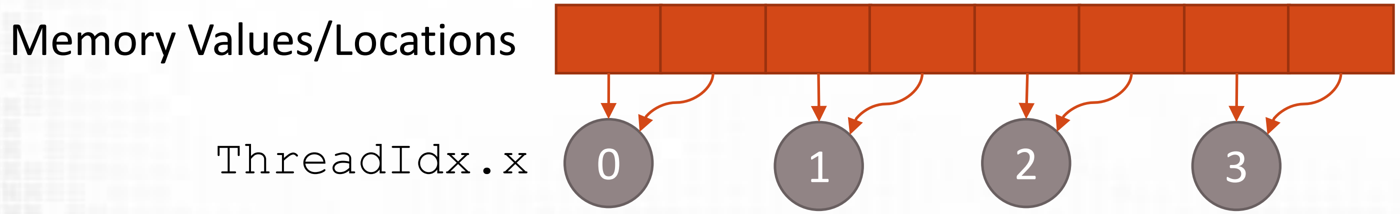
- ❑ Parallel Patterns Overview
- ❑ Reduction
- ❑ Scan

What are parallel Patterns

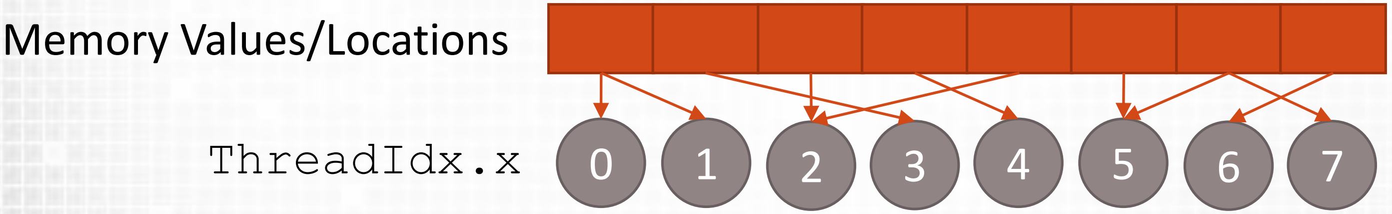
- Parallel patterns are high level building blocks that can be used to create algorithms
- Implementation is abstracted to give a higher level view
- Patterns describe techniques suited to parallelism
  - Allows algorithms to be built with parallelism from ground up
  - Top down approach might not parallelise very easily…
- Consider a the simplest parallel pattern: *Map*
  - Takes the input list *i*
  - Applies a function *f*
  - Writes the result list *o* by applying *f* to all members of *i*
  - Equivalent to a CUDA kernel where *i* and *o* are memory locations determined by `threadIdx` etc.

Gather

- ❑ Multiple inputs and single coalesced output
- ❑ Might have sequential loading or random access
  - ❑ Affect memory performance
- ❑ Differs to map due to multiple inputs



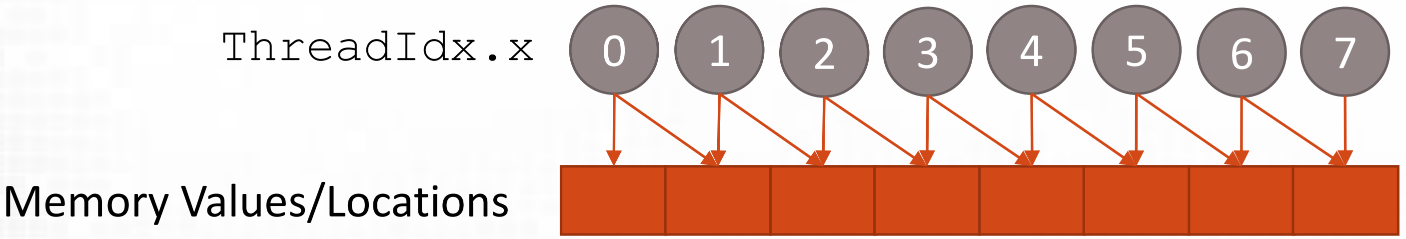
Gather operation
- ❑ Read from a number of locations



Gather operation
- ❑ Read from a number of locations
- ❑ Random access load

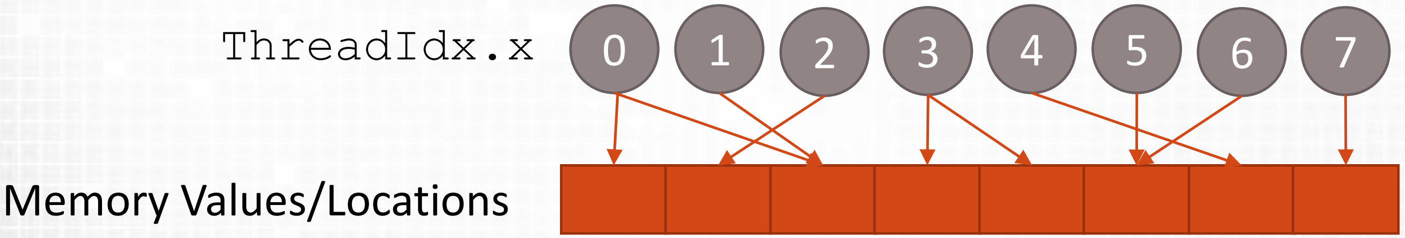
Scatter

- ❑Reads from a single input and writes to one or many
- ❑Can be implemented in CUDA using atomics
- ❑Write pattern will determine performance



Scatter operation

- ❑ Write to a number of locations
- ❑ Collision on write

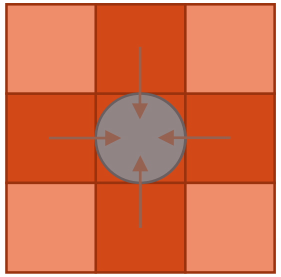


Scatter operation

- ❑ Write to a number of locations
- ❑ Random access write?

Other Parallel Patterns

- Stencil
  - Gather a fixed pattern, usually based on locality
  - See 2D shared memory examples
- **Reduce (this lecture)**
  - Reduce value to a single value or set of key value pairs
  - Combined with Map to form Map Reduce (often with intermediate shuffle or sort)
- **Scan (this lecture)**
  - Compute the sum of previous value in a set
- Sort (*later*)
  - Sort values or <value, key> pairs



Stencil Gather

- Parallel Patterns Overview
- Reduction
- Scan

Reduction

- A reduction is where **all** elements of a set have a common *binary associative operator* (⊕) applied to them to "reduce" the set to a single value
  - Binary associative = order in which operations is performed on set does not matter
    - E.g. `(1 + 2) + 3 + 4 == 1 + (2 + 3) + 4 == 10`
- Example operators
  - Most obvious example is addition (Summation)
  - Other examples, Maximum, Minimum, product
- Serial example is trivial but how does this work in parallel?

```
int data[N];
int i, r;
for (int i = 0; i < N; i++){
  r = reduce(r, data[i]);
}
```

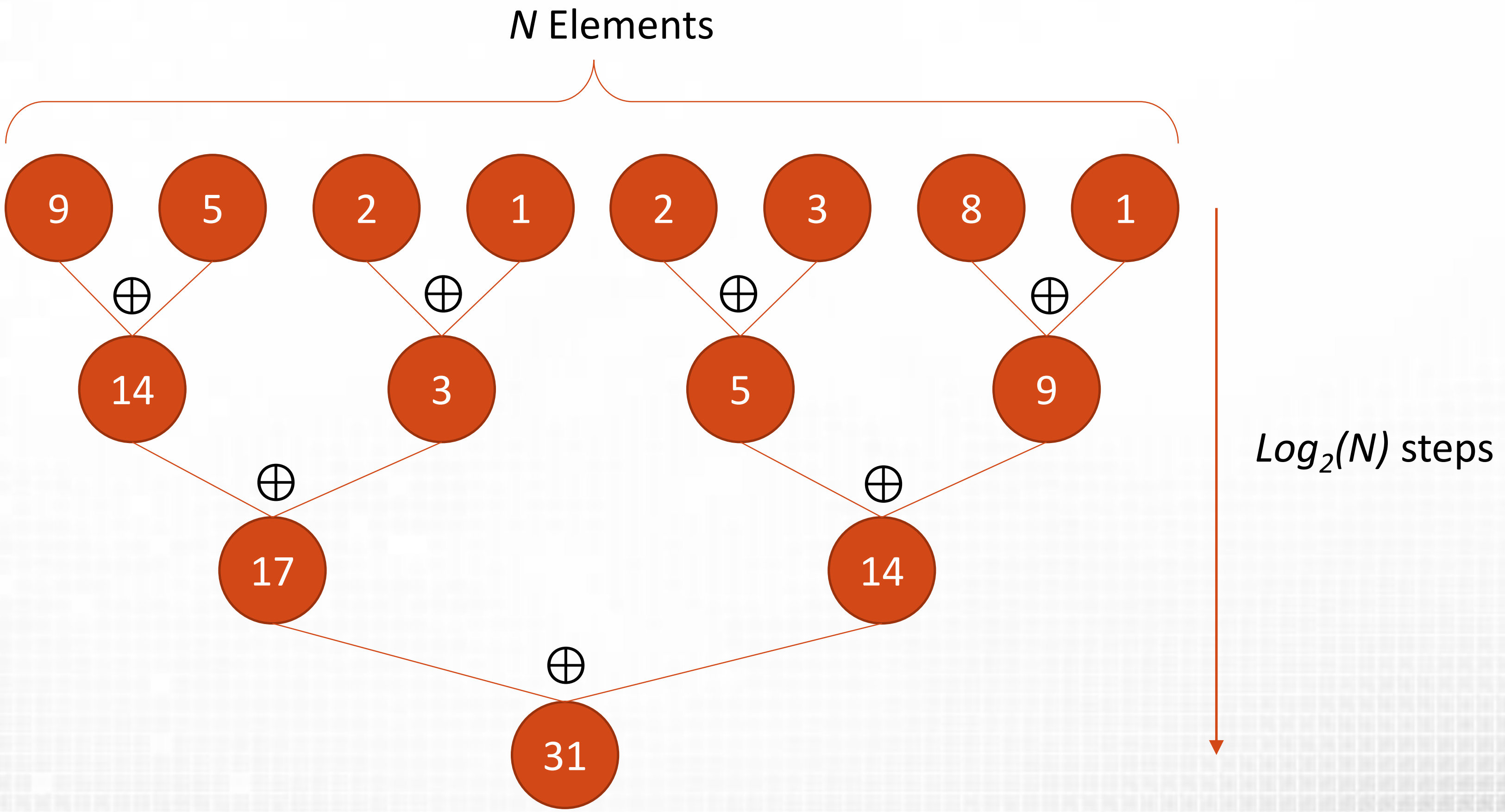
OR

```
int data[N];
int i, r;
for (int i = N-1; i >= 0; i--){
  r = reduce(r, data[i]);
}
```

```
int reduce(int r, int i){
  return r + i;
}
```

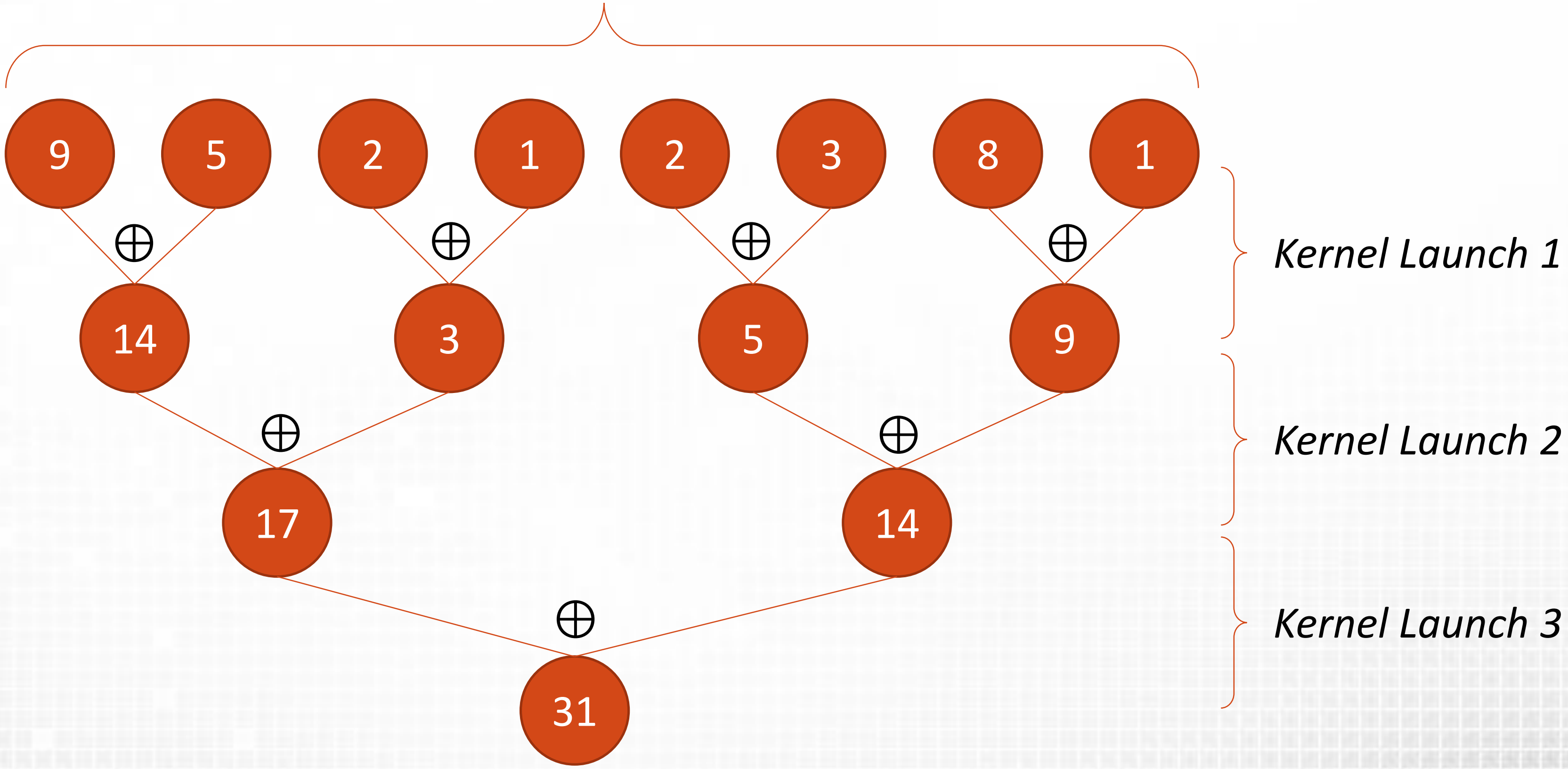
Parallel Reduction

- Order of operations does not matter so we don't have to think serially.
- A tree based approach can be used
  - At each step data is reduced by a factor of 2



Parallel Reduction in CUDA

- No global synchronisation so how do multiple blocks perform reduction?
- Split the execution into multiple stages
  - Recursive method



Recursive Reduction Problems

- What might be some problems with the following?

```
__global__ void sum_reduction(float *input, float *results){

  extern __shared__ int sdata[];
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;

  sdata[threadIdx.x] = input[i];
  __syncthreads();

  if (i % 2 == 0){
    results[i / 2] = sdata[threadIdx.x] + sdata[threadIdx.x+1]
  }

}
```

Recursive Reduction Problems

- High Launch Overhead
- Lots of reads/writes from global memory
  - Poor use of shared memory or caching
- Expensive `%` and `/` operators
- Divergent warps

```
__global__ void sum_reduction(float *input, float *results){

  extern __shared__ int sdata[];
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;

  sdata[threadIdx.x] = input[i];
  __syncthreads();

  if (i % 2 == 0){
    results[i / 2] = sdata[threadIdx.x] + sdata[threadIdx.x+1]
  }

}
```

The University Of Sheffield.

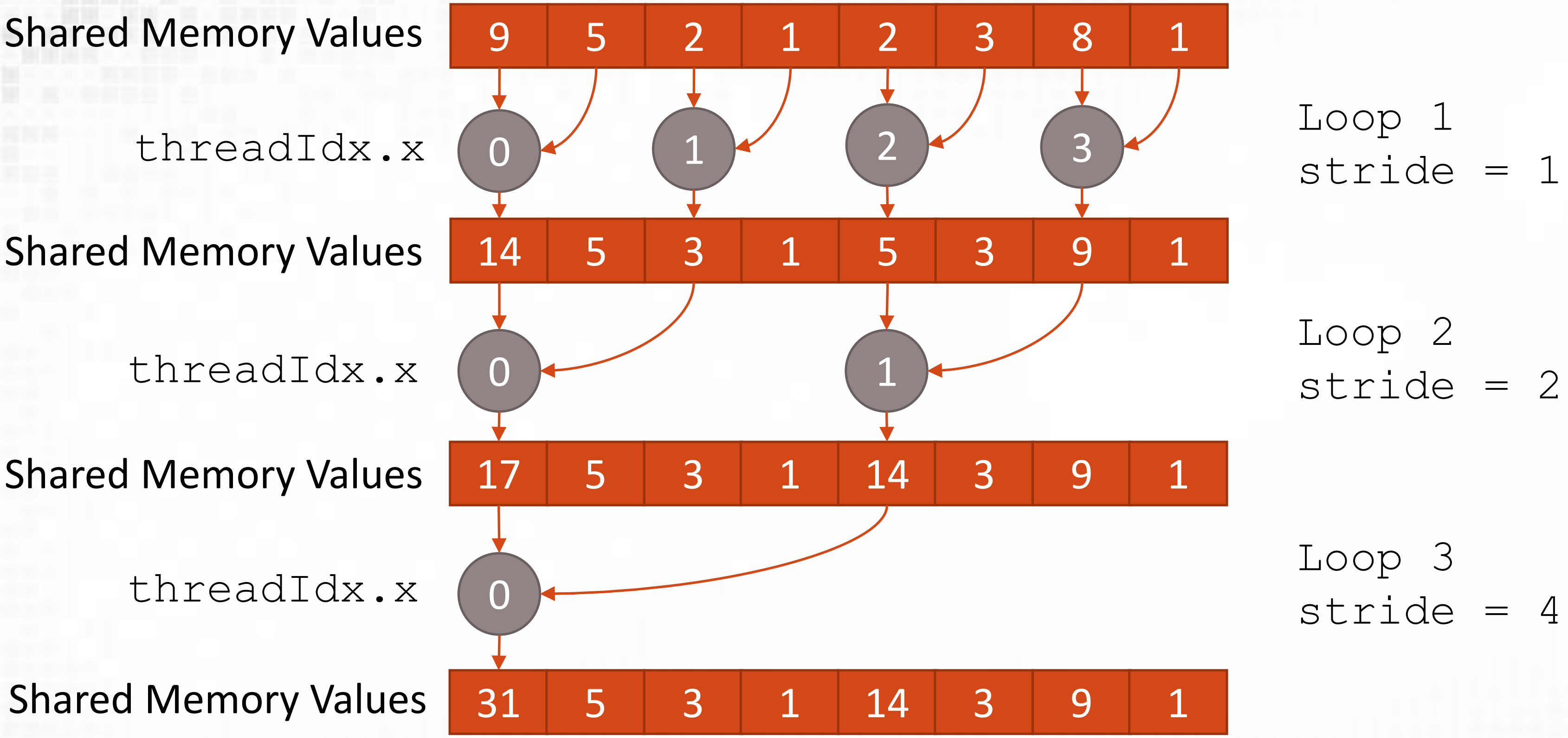
NVIDIA GPU RESEARCH CENTER

Block Level Reduction

- Lower launch overhead (reduction within block)
- Much better use of shared memory

```
__global__ void sum_reduction(float *input, float *block_results){
  extern __shared__ int sdata[];

  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  sdata[threadIdx.x] = input[i];
  __syncthreads();

  for (unsigned int stride = 1; stride < blockDim.x; stride*=2){
    unsigned int strided_i = threadIdx.x * 2 * stride;
    if (strided_i < blockDim.x){
      sdata[strided_i] += sdata[strided_i + stride]
    }
    __syncthreads();
  }

  if (threadIdx.x == 0)
    block_results[blockIdx.x] = sdata[0];
}
```

Block Level Recursive Reduction




```
for (unsigned int stride = 1; stride < blockDim.x; stride*=2){
  unsigned int strided_i = threadIdx.x * 2 * stride;
  if (strided_i < blockDim.x){
    sdata[strided_i] += sdata[strided_i + stride]
  }
  __syncthreads();
}
```

The University Of Sheffield.

NVIDIA GPU RESEARCH CENTER

Block Level Reduction

- Is this shared memory access pattern bank conflict free?

```
for (unsigned int stride = 1; stride < blockDim.x; stride*=2){
  unsigned int strided_i = threadIdx.x * 2 * stride;
  if (strided_i < blockDim.x){
    sdata[strided_i] += sdata[strided_i + stride]
  }
  __syncthreads();
}
```

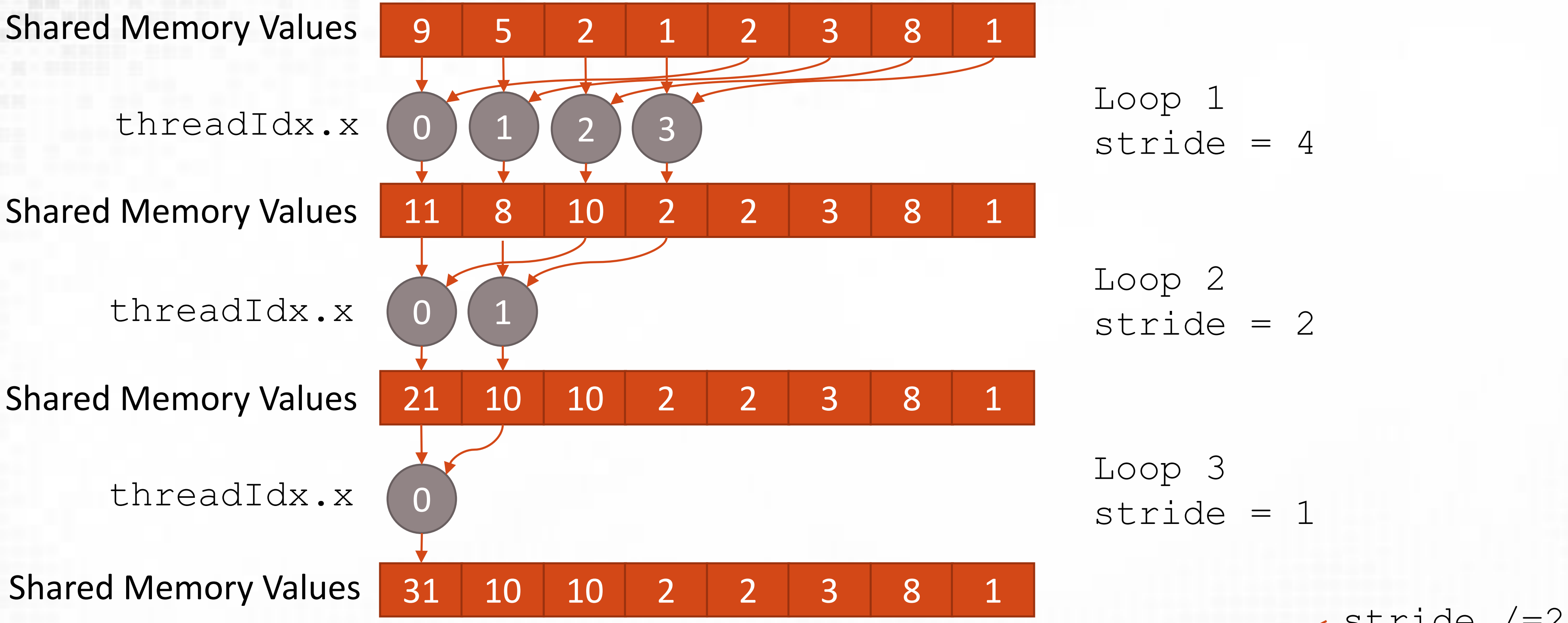
Block Level Reduction

- ❑ Is this shared memory access pattern conflict free? No
  - ❑ Each thread accesses SM using the following access pattern
    - ❑ `index = threadIdx.x * 2 * loop_stride + loop_stride`
  - ❑ Between each thread there is therefore strided access across SM banks
    - ❑ Try evaluating this using a spreadsheet
- ❑ To avoid bank conflicts stride between threads should be 1

```
for (unsigned int stride = 1; stride < blockDim.x; stride*=2){
  unsigned int strided_i = threadIdx.x * 2 * stride;
  if (strided_i < blockDim.x){
    sdata[strided_i] += sdata[strided_i + stride]
  }
  __syncthreads();
}
```

sm_stride	1			
loop stride	1			
threadIdx.x			index	bank
0			1	1
1			3	3
2			5	5
3			7	7
4			9	9
5			11	11
6			13	13
7			15	15
8			17	17
9			19	19
10			21	21
11			23	23
12			25	25
13			27	27
14			29	29
15			31	31
16			33	1
17			35	3
18			37	5
19			39	7
20			41	9
21			43	11
22			45	13
23			47	15
24			49	17
25			51	19
26			53	21
27			55	23
28			57	25
29			59	27
30			61	29
31			63	31
			Banks Used	16
			Max Conflicts	2

Block Level Reduction (Sequential Addressing)




```
for (unsigned int stride = blockDim.x/2; stride > 0; stride>>=1){
  if (threadIdx.x < stride){
    sdata[threadIdx.x] += sdata[threadIdx.x + stride]
  }
  __syncthreads();
}
```

The University Of Sheffield.

NVIDIA GPU RESEARCH CENTER

sm_stride	1			
loop stride	1			
threadIdx.x			index	bank
0			1	1
1			2	2
2			3	3
3			4	4
4			5	5
5			6	6
6			7	7
7			8	8
8			9	9
9			10	10
10			11	11
11			12	12
12			13	13
13			14	14
14			15	15
15			16	16
16			17	17
17			18	18
18			19	19
19			20	20
20			21	21
21			22	22
22			23	23
23			24	24
24			25	25
25			26	26
26			27	27
27			28	28
28			29	29
29			30	30
30			31	31
31			32	0
			Banks Used	32
			Max Conflicts	1

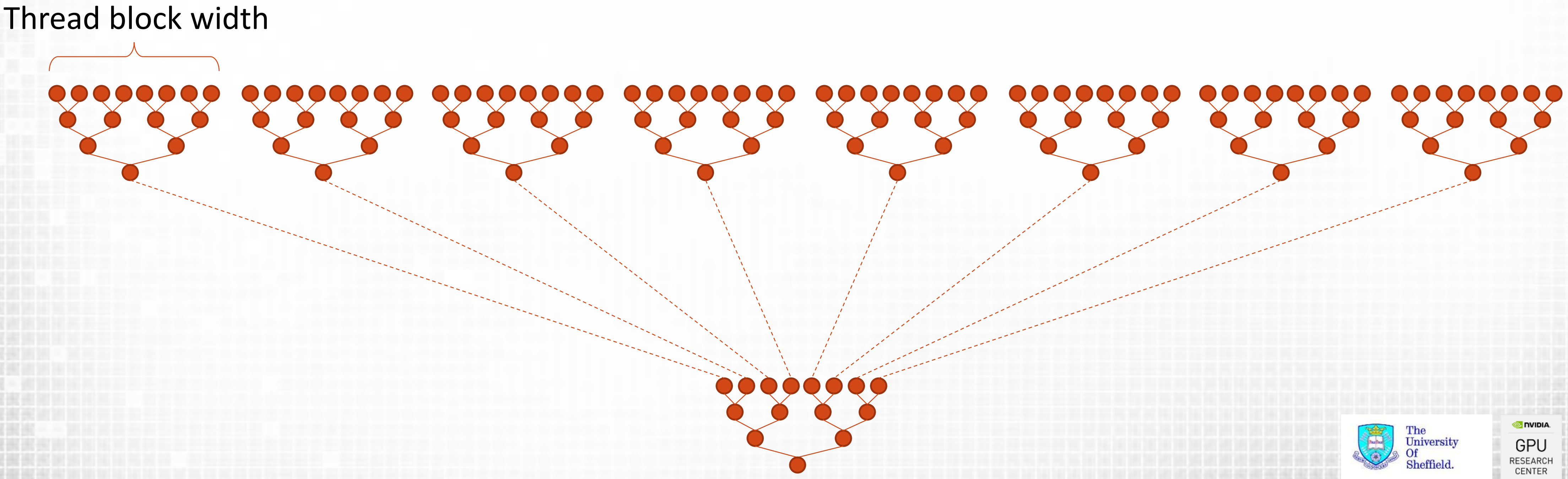
- Now conflict free regardless of the reduction loop stride
- The stride between shared memory variable accesses for threads is *always* sequential

- Careful: Two types of stride discussed

1. Loop stride (of algorithm)
2. SM *variable* stride (in 4 bytes)

Global Reduction Approach

- Use the recursive method
  - Our block level reduction can be applied to the result
  - At some stage it may be more effective to simply sum the final block on the CPU
- Or use atomics on block results



Global Reduction Atomics

```
__global__ void sum_reduction(float *input, float *result){
  extern __shared__ int sdata[];

  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  sdata[threadIdx.x] = input[i];
  __syncthreads();

  for (unsigned int stride = blockDim.x/2; stride > 0; stride>>=2){
    if (threadIdx.x < stride){
      sdata[threadIdx.x] += sdata[threadIdx.x + stride]
    }
    __syncthreads();
  }

  if (threadIdx.x == 0)
    atomicAdd(result, sdata[0]);
}
```

Further Optimisation?

- Can we improve our technique further?

```
__global__ void sum_reduction(float *input, float *result){
  extern __shared__ int sdata[];

  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  sdata[threadIdx.x] = input[i];
  __syncthreads();

  for (unsigned int stride = blockDim.x/2; stride > 0; stride>>=2){
    if (threadIdx.x < stride){
      sdata[threadIdx.x] += sdata[threadIdx.x + stride]
    }
    __syncthreads();
  }

  if (threadIdx.x == 0)
    atomicAdd(result, sdata[0]);
}
```

Further Optimisation?

- Can we improve our technique further? Yes

- **We could optimise for the warp level**
  - **Warp Level**: Shuffles for reduction (*see last lecture*)
  - **Thread Block Level**: Shared Memory reduction (or Maxwell SM atomics)
  - **Grid Block Level**: Recursive Kernel Launches or Global Atomics

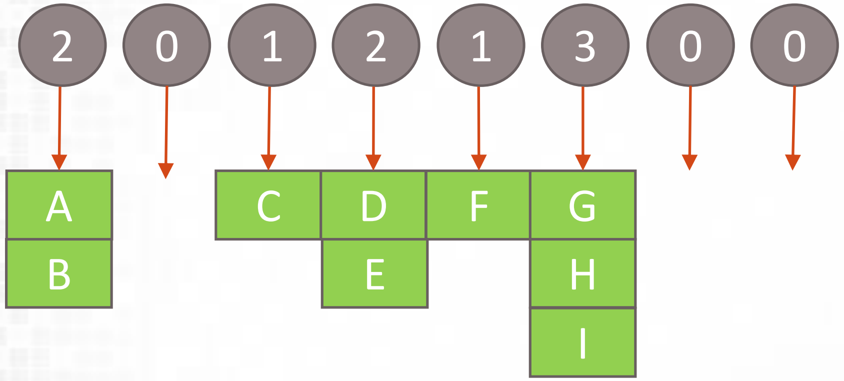
- Other optimisations
  - Loop unrolling
  - Increasing Thread Level Parallelism

- Different architectures may favour different implementations/optimisations

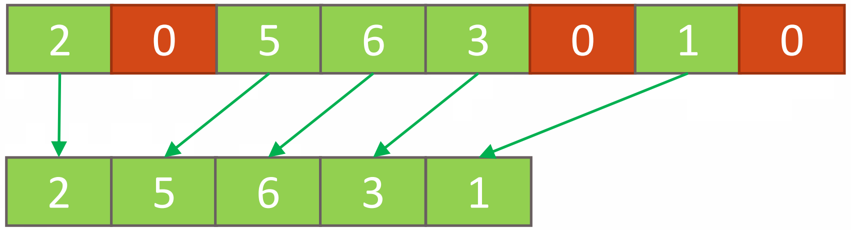
- Parallel Patterns Overview
- Reduction
- Scan

What is scan?

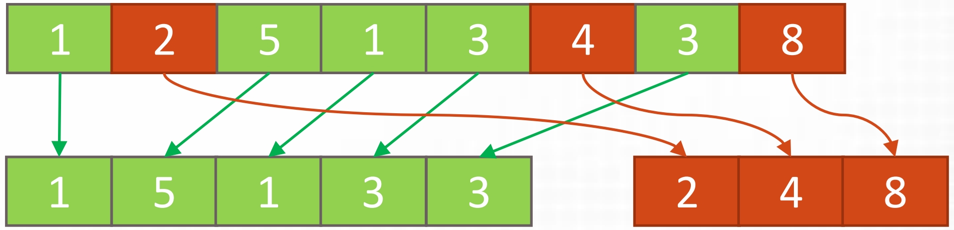
- Consider the following …



Output variable numbers of values per thread



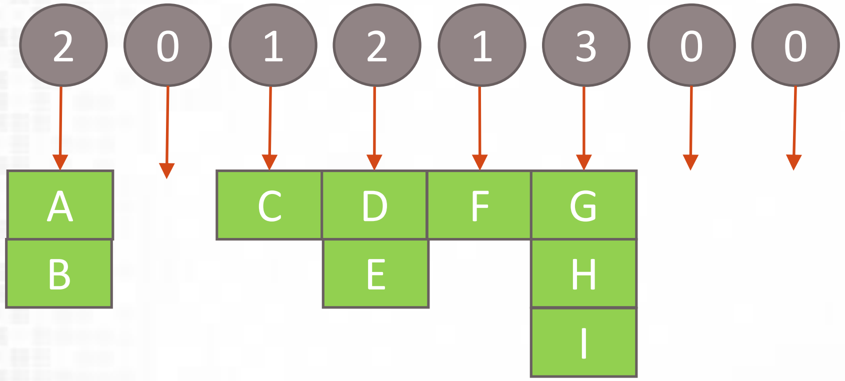
Remove empty elements from array (compact)



Split elements from array based on condition (split)

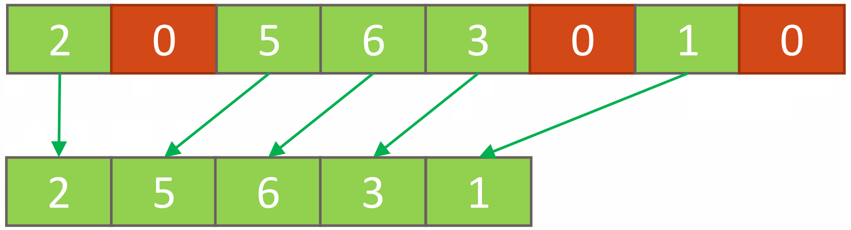
What is scan?

- Consider the following …

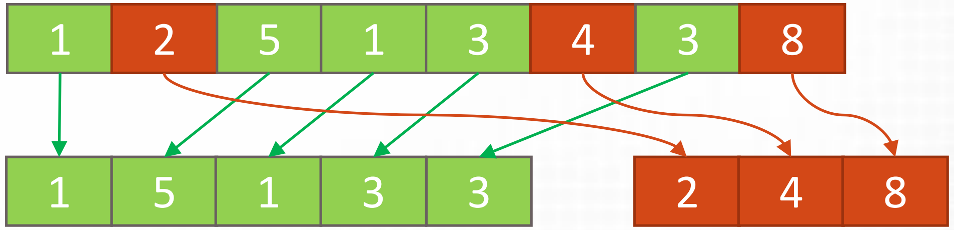


Output variable numbers of values per thread

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



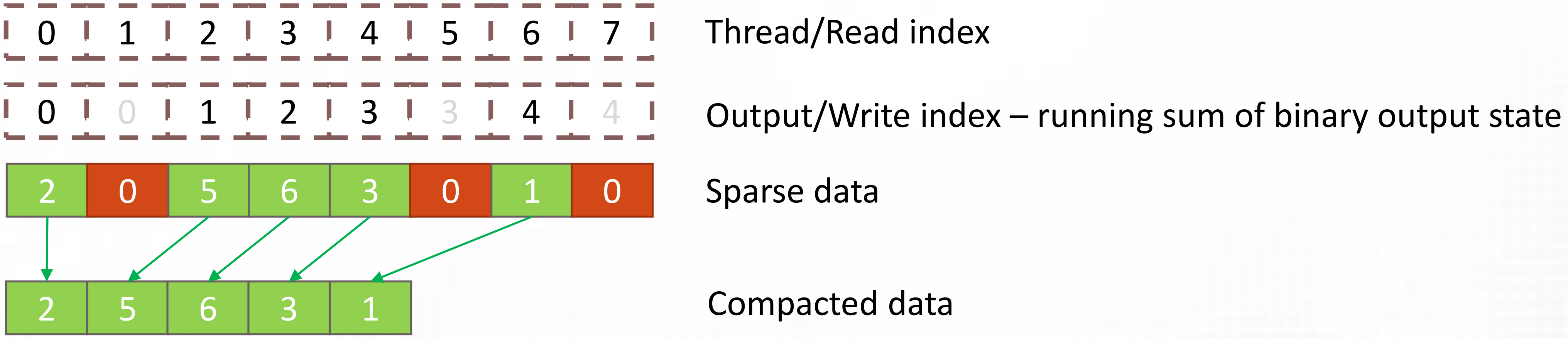
Remove empty elements from array (compact)



Split elements from array based on condition (split)

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.



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

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];
}
```

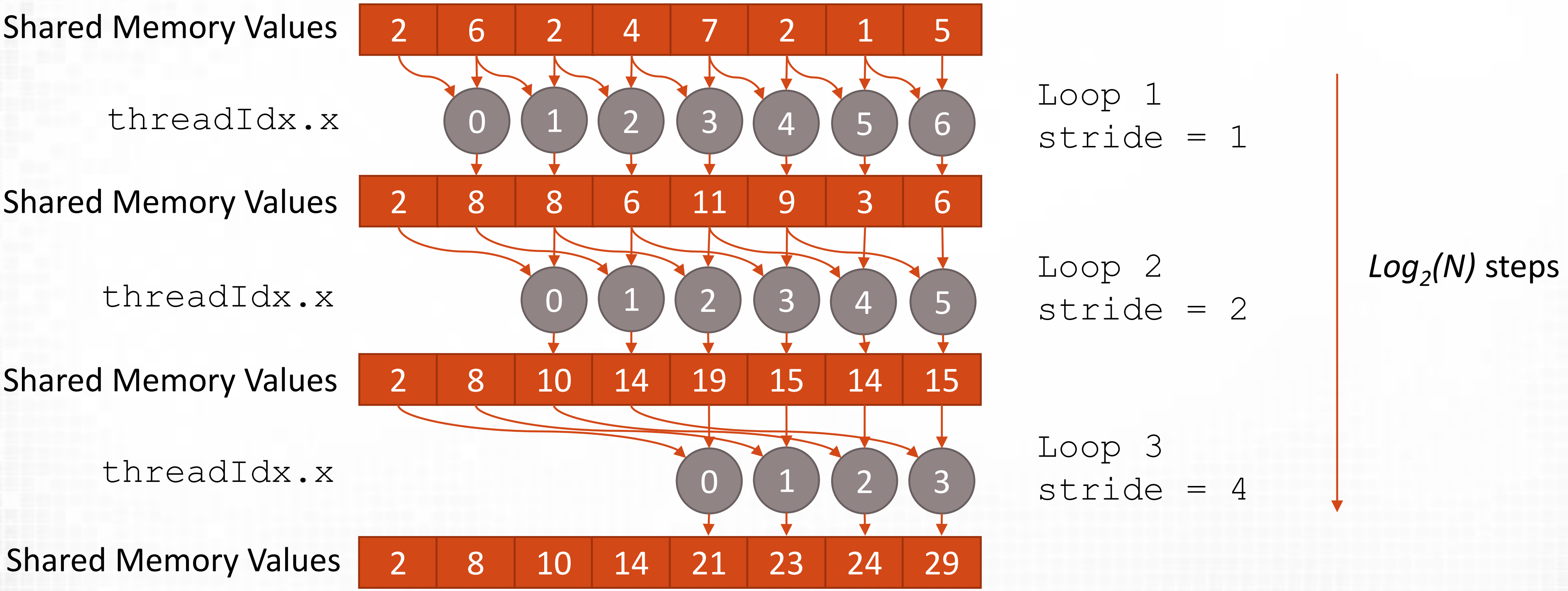
Serial Scan for Compaction

```
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_A[i];
    output[idx] = input;
  }
}
```

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

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

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

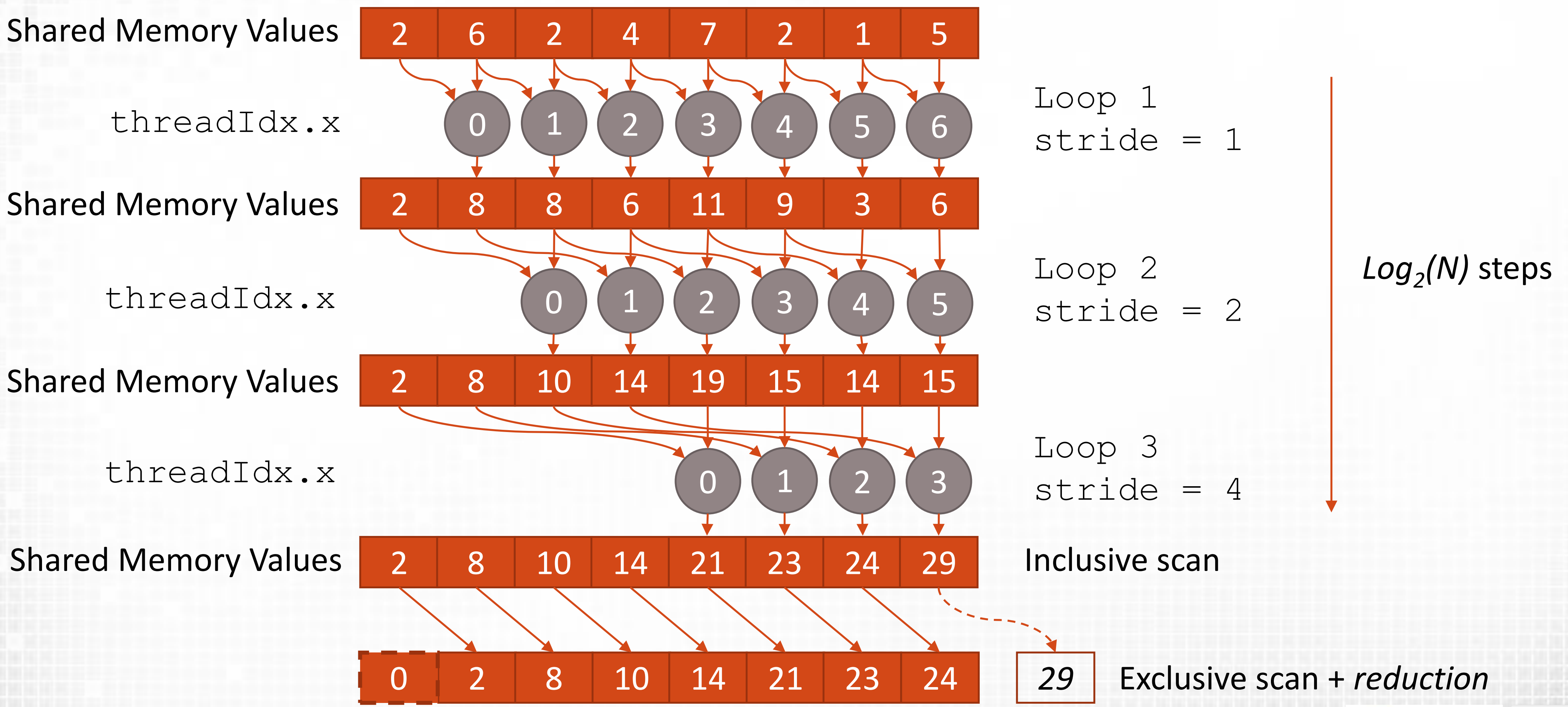
Parallel Local (Shared Memory) Scan

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



Inclusive Scan

Parallel Local Scan



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?
}
```

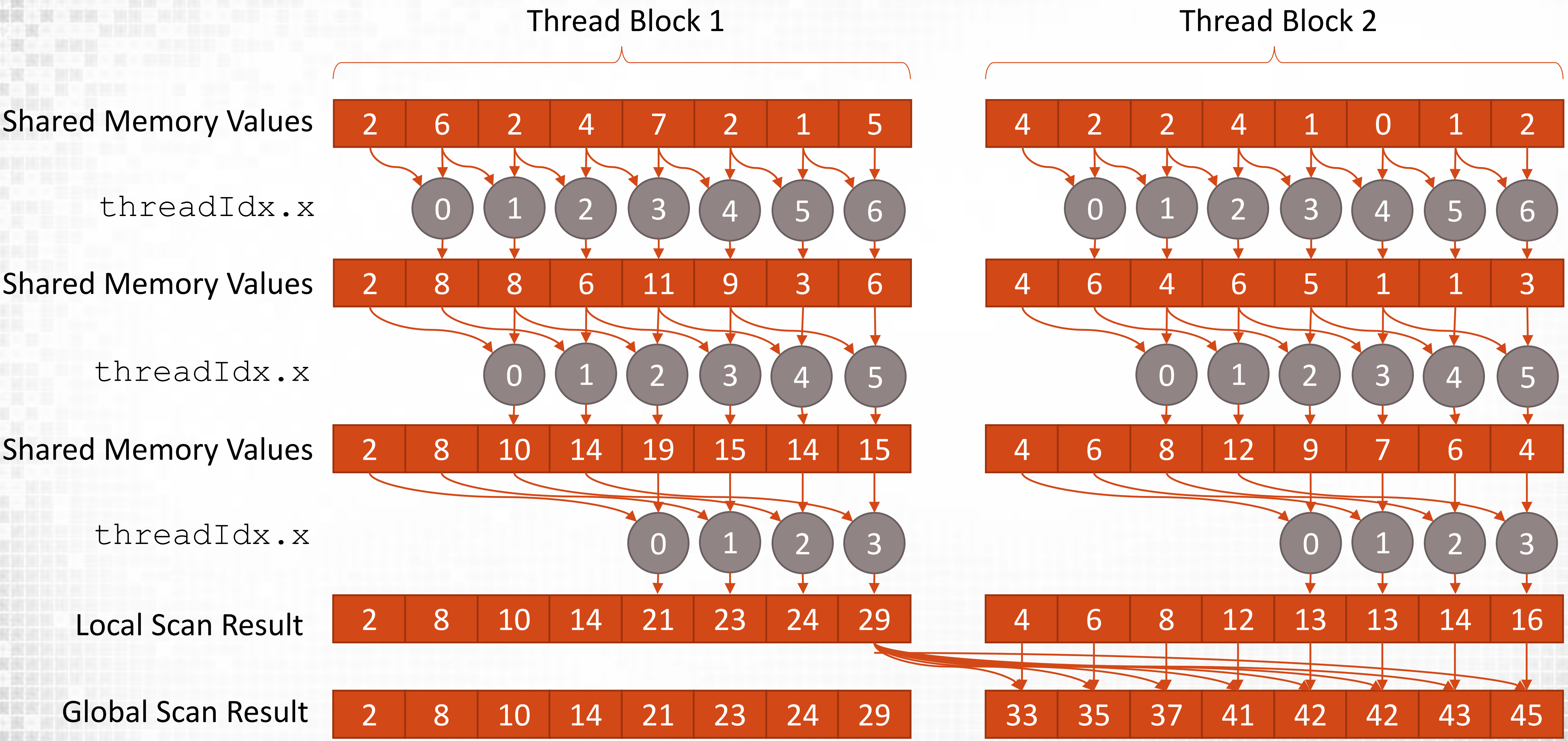
- No bank conflicts (stride of 1 between threads)
- Synchronisation required between read and write

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



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[GRID_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
  if (threadIdx.x == 0){
    for (int i=0; i<blockIdx.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

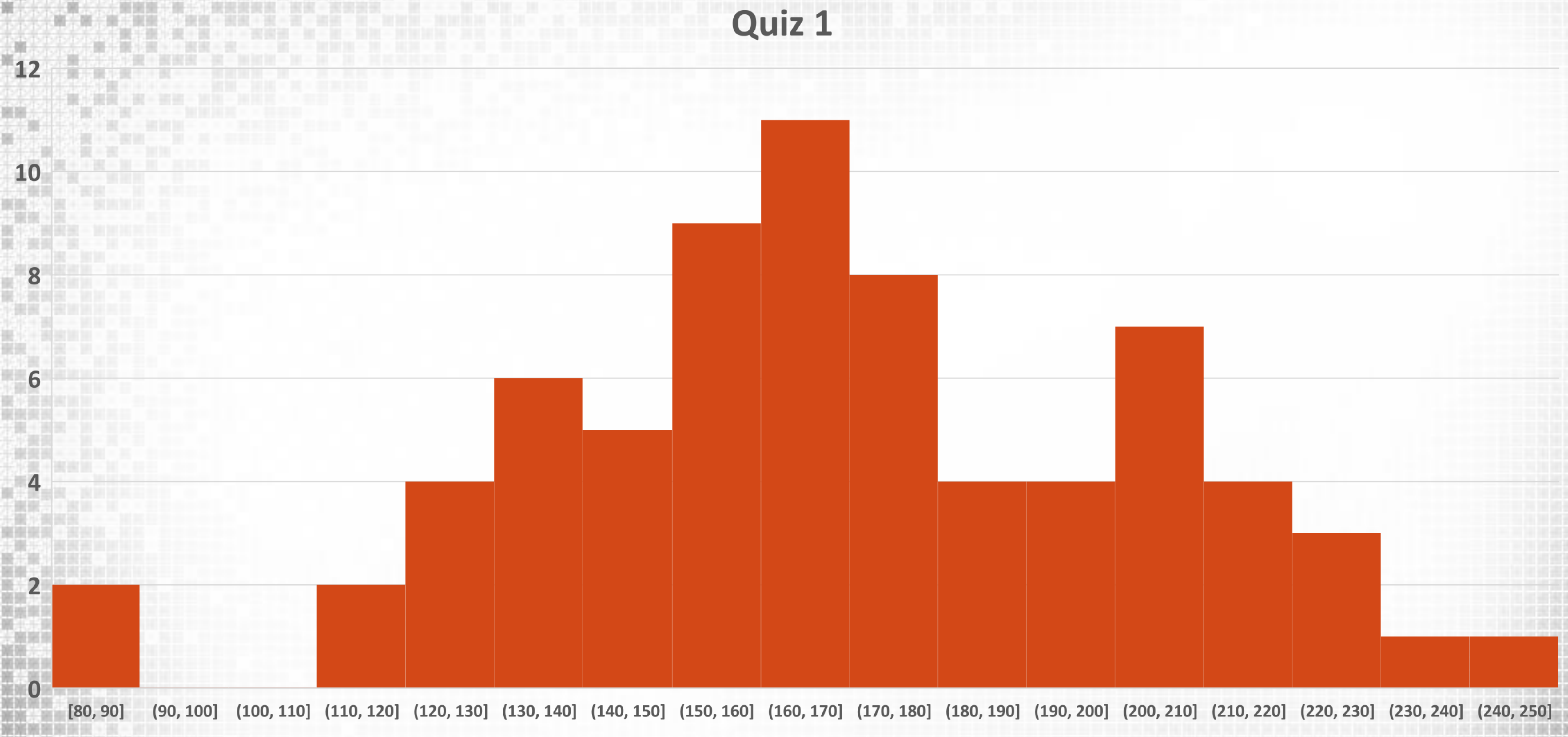
```
__device__ block_sums[BLOCK_DIM];

__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

- Parallel Patterns create a bottom up model for constructing algorithms from parallel building blocks
- Reduction can be implemented recursively however re-use of data avoid costly memory movement operations
- Scan is a building block for all kinds of problems
  - Can be used for compaction and split
- Parallel patterns can be optimised at warp, thread block and grid levels.
- Atomics can be used in the reduction or scan value summation between blocks or warps
- Lots of potential techniques to implement and evaluate
  - Fortunately in many cases libraries and examples already exist

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)



The University Of Sheffield.
NVIDIA GPU RESEARCH CENTER

PhD Position

- Interested in doing PhD?
- **Title:** Parallel computation of adaptive wavelet-based 2D flood models
- **Supervisors:** Georges Kesserwani, Paul Richmond
- **Deadline :** July 1st, 2019

Apply here:

http://tinyurl.com/y3fw5n8r

The University Of Sheffield. NVIDIA GPU RESEARCH CENTER