Parallel Computing with GPUs

Parallel Patterns
Part 1 – Parallel Patterns Overview

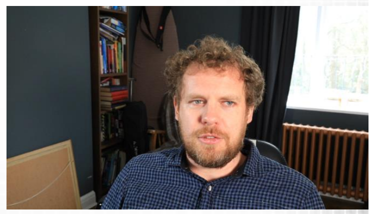
Dr Paul Richmond
http://paulrichmond.shef.ac.uk/teaching/COM4521/

This Lecture (learning objectives)

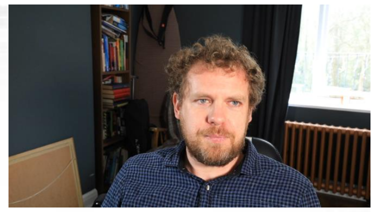
- ❑Parallel Patterns
  - ❑Define a parallel pattern as building blocks for parallel applications
  - ❑Give examples of common patterns

What are parallel Patterns

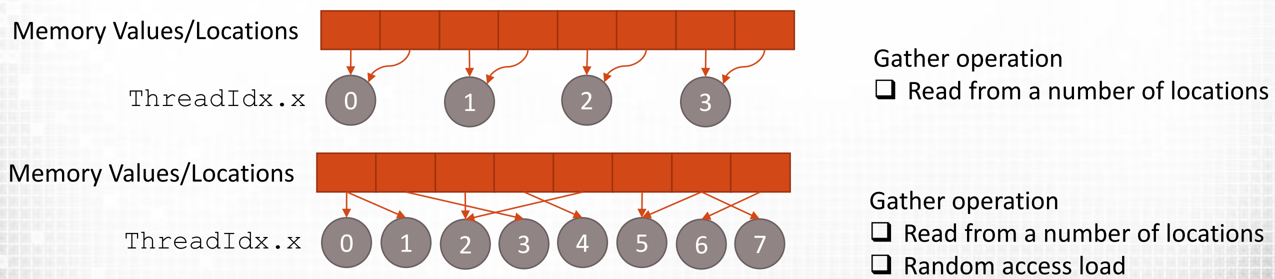
- ❑Parallel patterns are high level building blocks that can be used to create algorithms
- ❑Implementation is abstracted to give a higher level view
- ❑Patterns describe techniques suited to parallelism
  - ❑Allows algorithms to be built with parallelism from ground up
  - ❑Top down approach might not 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

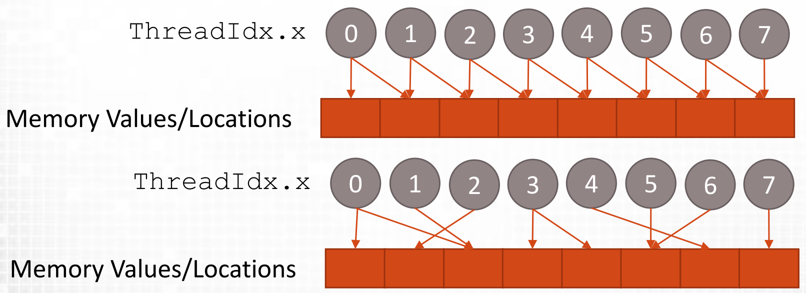


- ❑Multiple inputs and single coalesced output
- ❑Might have sequential loading or random access
  - ❑Affect memory performance
- ❑Differs to map due to multiple inputs



Scatter

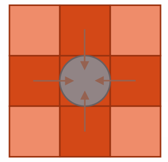
- ❑Reads from a single input and writes to one or many
- ❑Can be implemented in CUDA using atomics or block/warp co-operation
- ❑Write pattern will determine performance



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

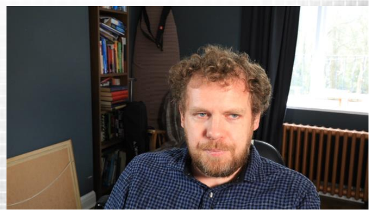


Stencil Gather

- ❑Stencil
  - ❑Gather a fixed pattern, usually based on locality
  - ❑See 2D shared memory examples
- ❑**Reduce**
  - ❑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**
  - ❑Compute the sum of previous value in a set
- ❑Sort (*later*)
  - ❑Sort values or <value, key> pairs

Summary

- ❑Parallel Patterns
  - ❑Define a parallel pattern as building blocks for parallel applications
  - ❑Give examples of common patterns
- ❑Next Lecture: Reduction

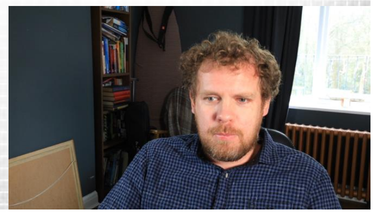


Parallel Computing with GPUs

Parallel Patterns
Part 2 - Reduction

The University Of Sheffield.

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



This Lecture (learning objectives)

- Reduction
  - Present the process of performing parallel reduction
  - Explore the performance implications of parallel reduction implementations
  - Analyze block level and atomic approaches for reduction

Reduction

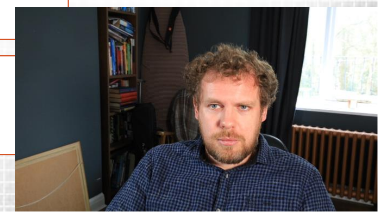
- A reduction is where **all** elements of a set have a common *binary associative operator* (⨁) applied to them to "reduce" the set to a single value
  - Binary associative = order in which operations is performed on set does not matter
    - E.g. `(1 + 2) + 3 + 4 == 1 + (2 + 3) + 4 == 10`
- Example 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

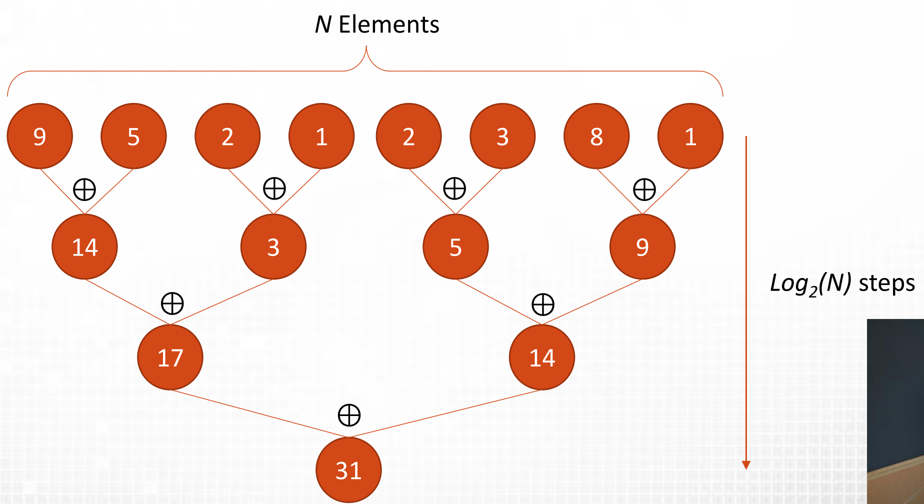
```
int data[N];
int i, r;
for (int i = N-1; i >= 0; i--){
  r = reduce(r, data[i]);
}
```

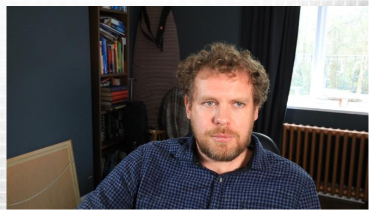
```
int reduce(int r, int i){
  return r + i;
}
```



Parallel Reduction

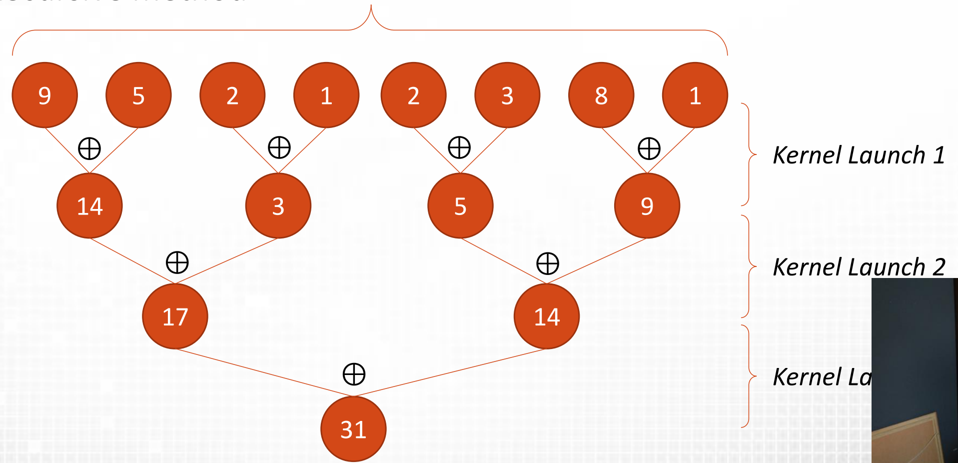
- Order of operations does not matter so we don't have to think serially.
- A tree based approach can be used
  - At each step data is reduced by a factor of 2





Parallel Reduction in CUDA

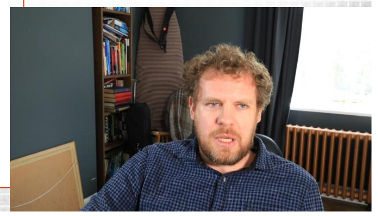
- No global synchronisation so how do multiple blocks perform reduction?
- Split the execution into multiple stages
  - Recursive method



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

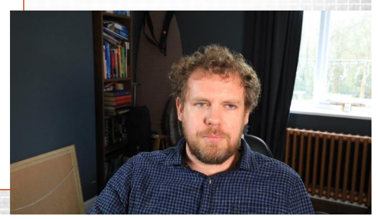
- High Launch Overhead
- Lots of reads/writes from global memory
  - Poor use of shared memory or caching
- Expensive `%` and `/` operators
- Divergent warps

```
__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]
  }

}
```

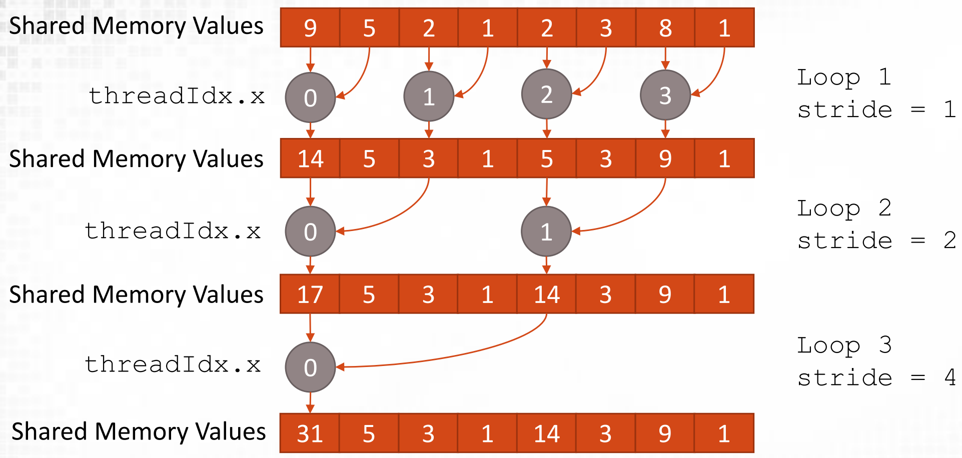
Block Level Reduction

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

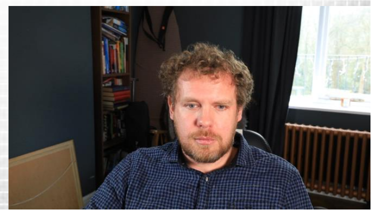
```
__global__ void sum_reduction(float *input, float *block_results){
  extern __shared__ int sdata[];

  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();
}
```



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();
}
```

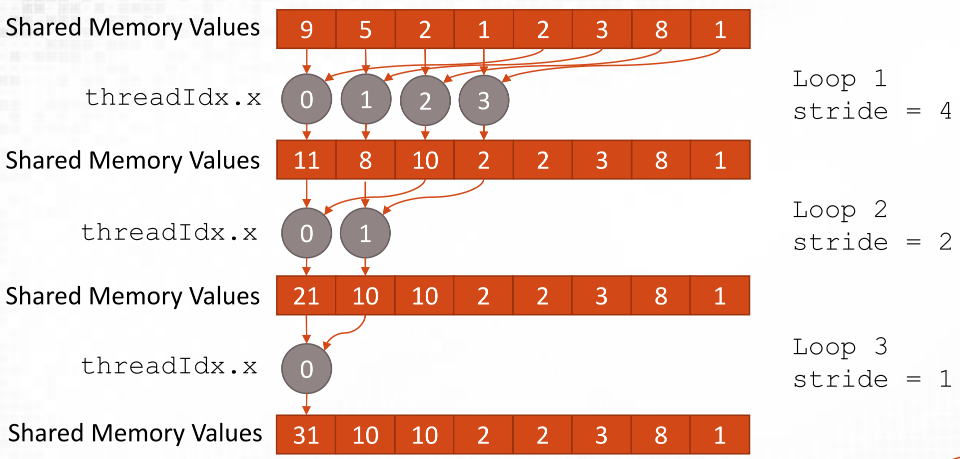
Block Level Reduction

- Is this shared memory access pattern conflict free? No
  - Each thread accesses SM bank using the following
    - sm_bank = (threadIdx.x * 2 * stride + word_size) % 32
  - Between each thread there is therefore strided access across SM banks
    - Try evaluating this using a spreadsheet
- To avoid bank conflicts SM 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();
}
```

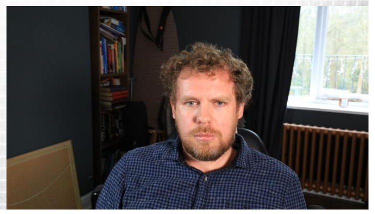
Word_size	1	
stride	1	
threadIdx.x	**index**	bank
0	1	1
1	3	3
2	5	5
3	7	7
4	9	9
5	11	11
6	13	13
7	15	15
8	17	17
9	19	19
10	21	21
11	23	23
12	25	25
13	27	27
14	29	29
15	31	31
16	33	1
17	35	3
18	37	5
19	39	7
20	41	9
21	43	11
22	45	13
23	47	15
24	49	17
25	51	19
26	53	21
27	55	23
28	57	25
29	59	27
30	61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

Block Level Reduction (Sequential Addressing)



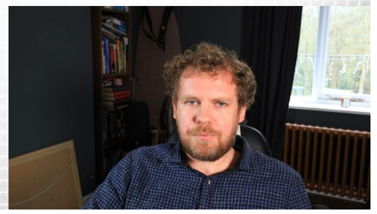
stride /=2

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



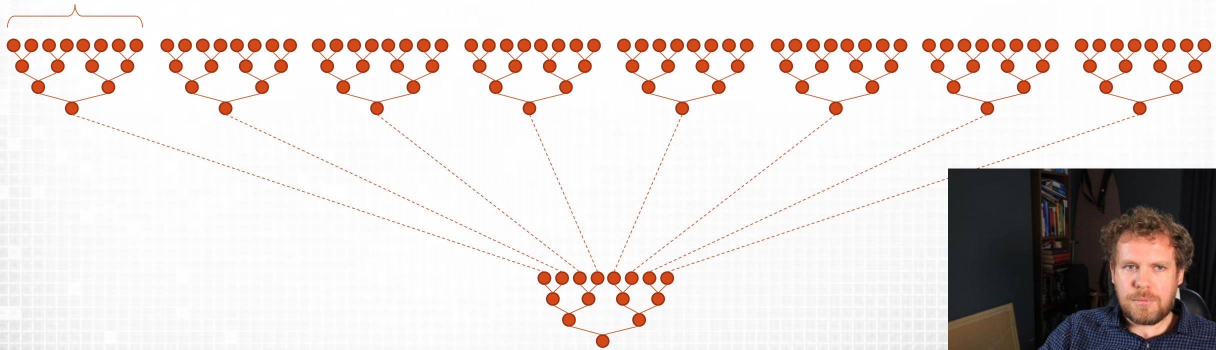
word size	1	
loop stride	16	
threadIdx.x	index	bank
0	16	16
1	17	17
2	18	18
3	19	19
4	20	20
5	21	21
6	22	22
7	23	23
8	24	24
9	25	25
10	26	26
11	27	27
12	28	28
13	29	29
14	30	30
15	31	31
16	32	0
17	33	1
18	34	2
19	35	3
20	36	4
21	37	5
22	38	6
23	39	7
24	40	8
25	41	9
26	42	10
27	43	11
28	44	12
29	45	13
30	46	14
31	47	15
	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



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



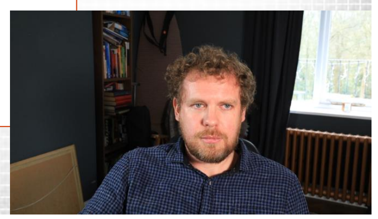
Global Reduction Atomics

```
__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

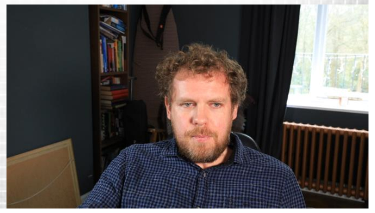
- **We could optimise for the warp level**
  - **Warp Level**: Shuffles for reduction (*see last lecture*)
  - **Thread Block Level**: Shared Memory reduction (or Maxwell SM atomics)
  - **Grid Block Level**: Recursive Kernel Launches or Global Atomics

- Other optimisations
  - Loop unrolling
  - Increasing Thread Level Parallelism

- Different architectures may favour different implementations/optimisations

Summary

- ❑Reduction
  - ❑Present the process of performing parallel reduction
  - ❑Explore the performance implications of parallel reduction implementations
  - ❑Analyze block level and atomic approaches for reduction
- ❑Next Lecture: Scan



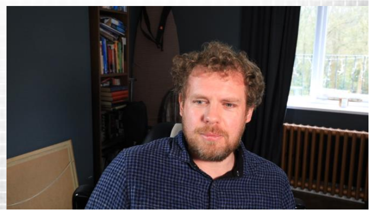
Parallel Computing with GPUs

Parallel Patterns Part 3 – Scan

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

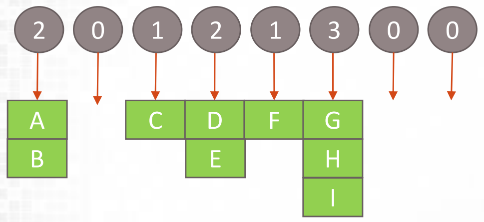
This Lecture (learning objectives)

- ❑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

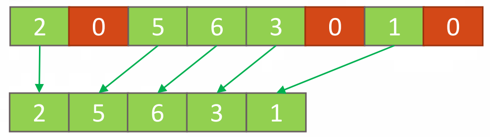


What is scan?

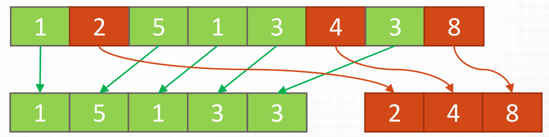
- ❑Consider the following …



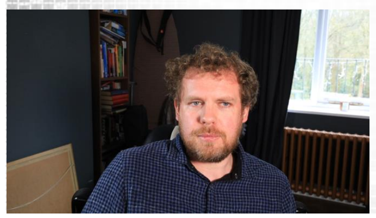
Output variable numbers of values per thread



Remove empty elements from array (compact)

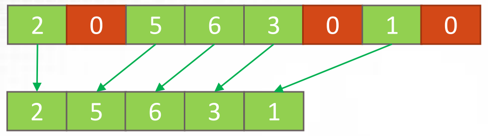


Split elements from array based on condition (split)

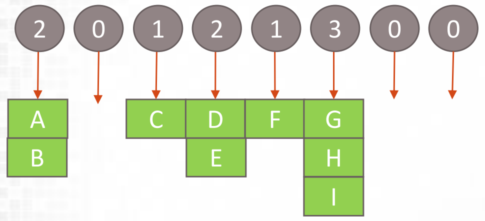


What is scan?

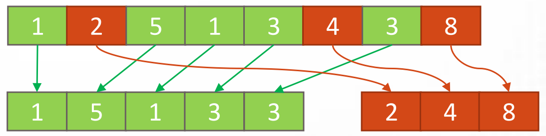
- Consider the following …



Remove empty elements from array (compact)



Output variable numbers of values per thread

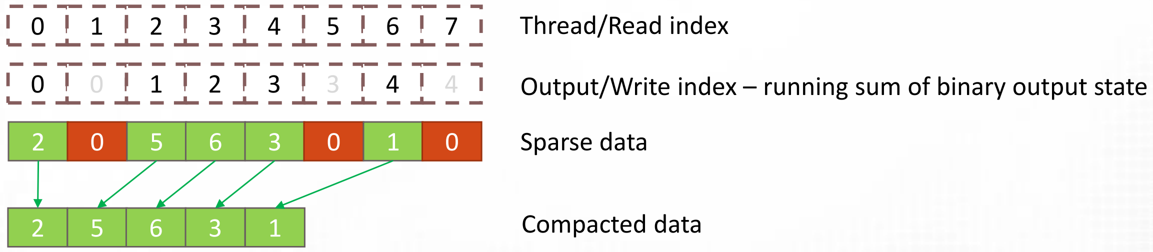


Split elements from array based on condition (split)

- 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.



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

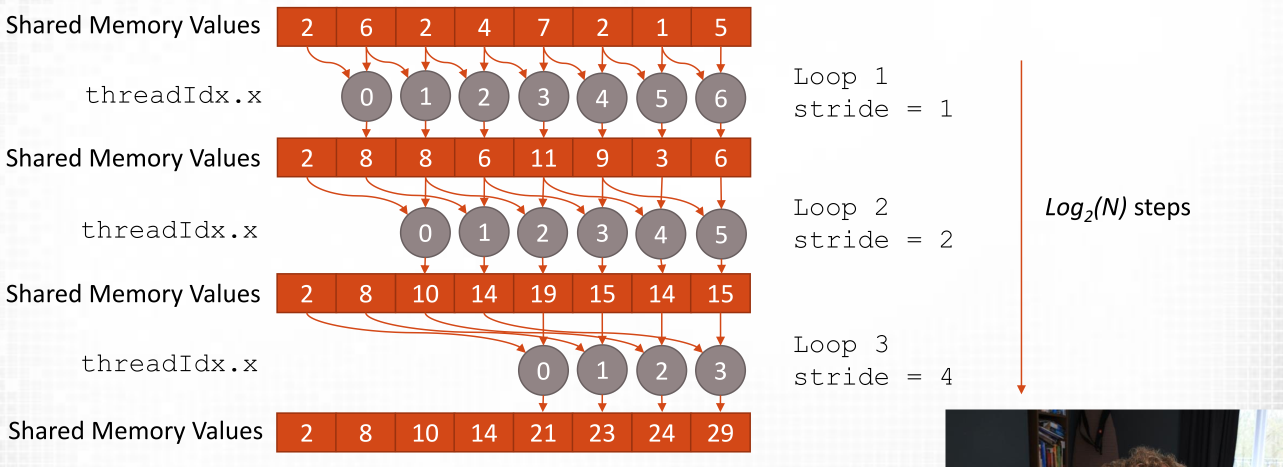
Serial Scan for Compaction

```
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){              // generate scan input
  A[i] = Input>0;                          // A = {1, 0, 1, 1, 1, 0, 1, 0}
}

for (int i = 0; i < 8; ++i){
  scan_A[i] = running_sum;                 // scan
  running_sum += A[i];                     // scan_A = {0, 1, 1, 2, 3, 4, 4, 5}
}

for (int i = 0; i < 8; ++i){
  int input = Input[i];
  if (input > 0){                          // scattered write
    int idx = scan[i];                     // output = {2, 5, 6, 3, 1}
    output[idx] = input;
  }
}
```

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

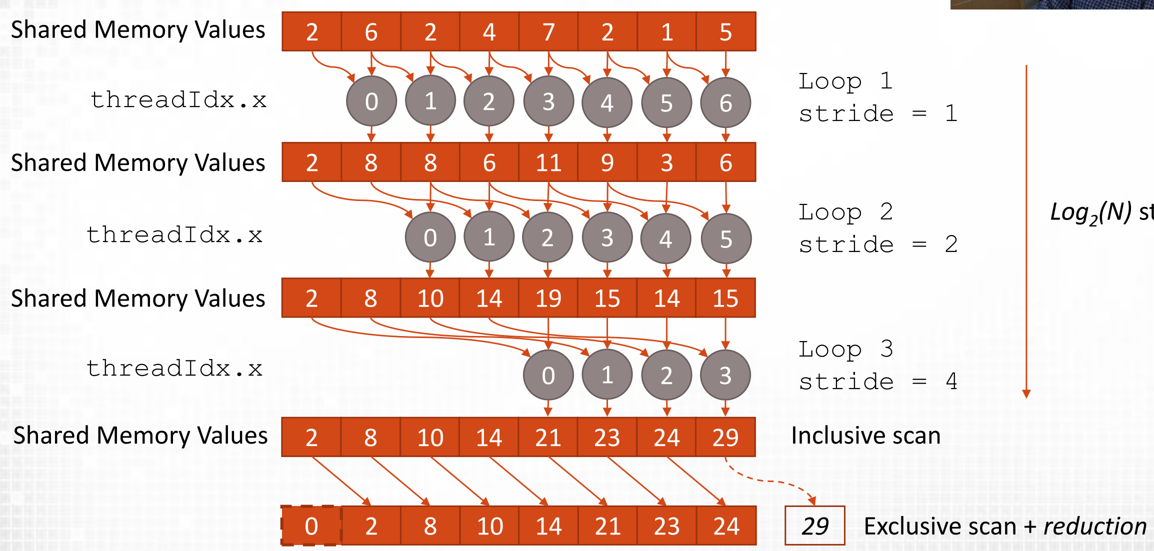
Parallel Local (Shared Memory) Scan

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



Inclusive Scan

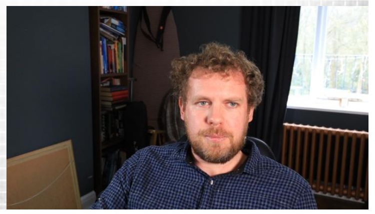
Parallel Local Scan



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?
}
```

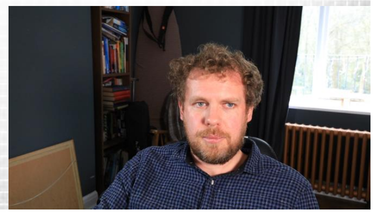
- ❑No bank conflicts (stride of 1 between threads)
- ❑Synchronisation required between read and write



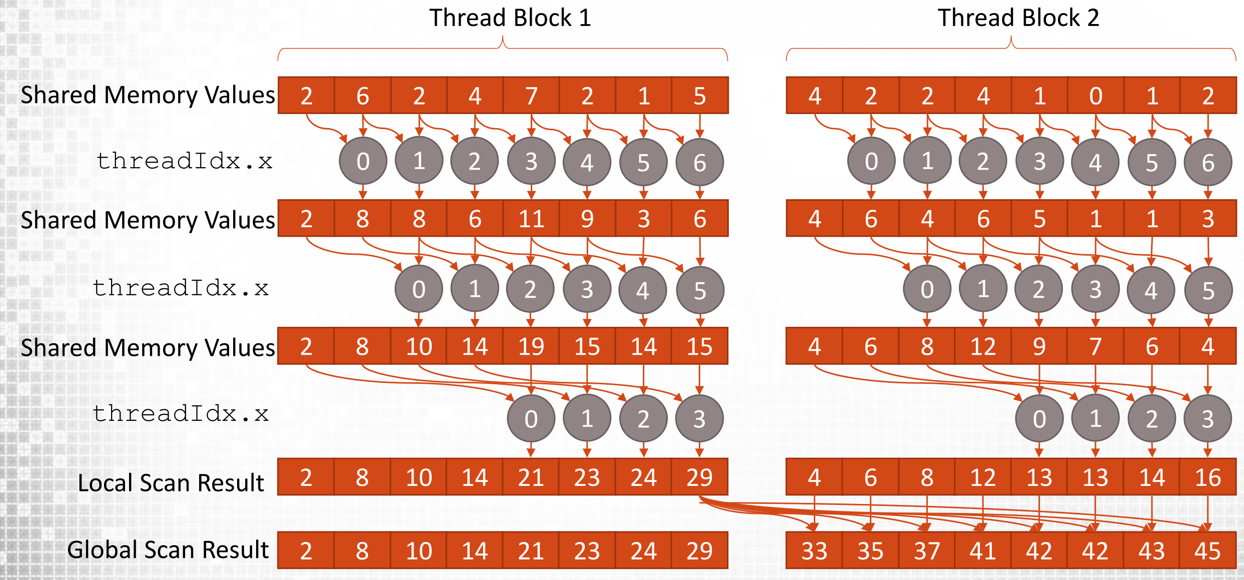
Implementing Local Scan (at warp level)

```
__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

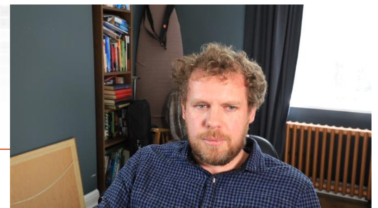


Implementing scan at Grid Level

- ❑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)

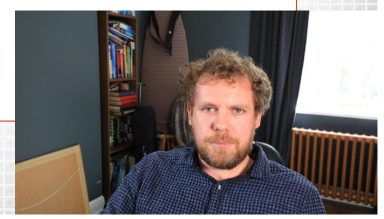


```
__device__ block_sums[BLOCK_DIM];

__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)

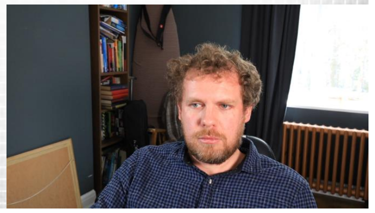
- ❑After completion of the first kernel, block sums are all synchronised
- ❑Use first thread in block to load block total into shared memory
- ❑Increment local result

```
__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)