Parallel Computing with GPUs

Parallel Patterns
Part 2 - Reduction

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]);
}
```

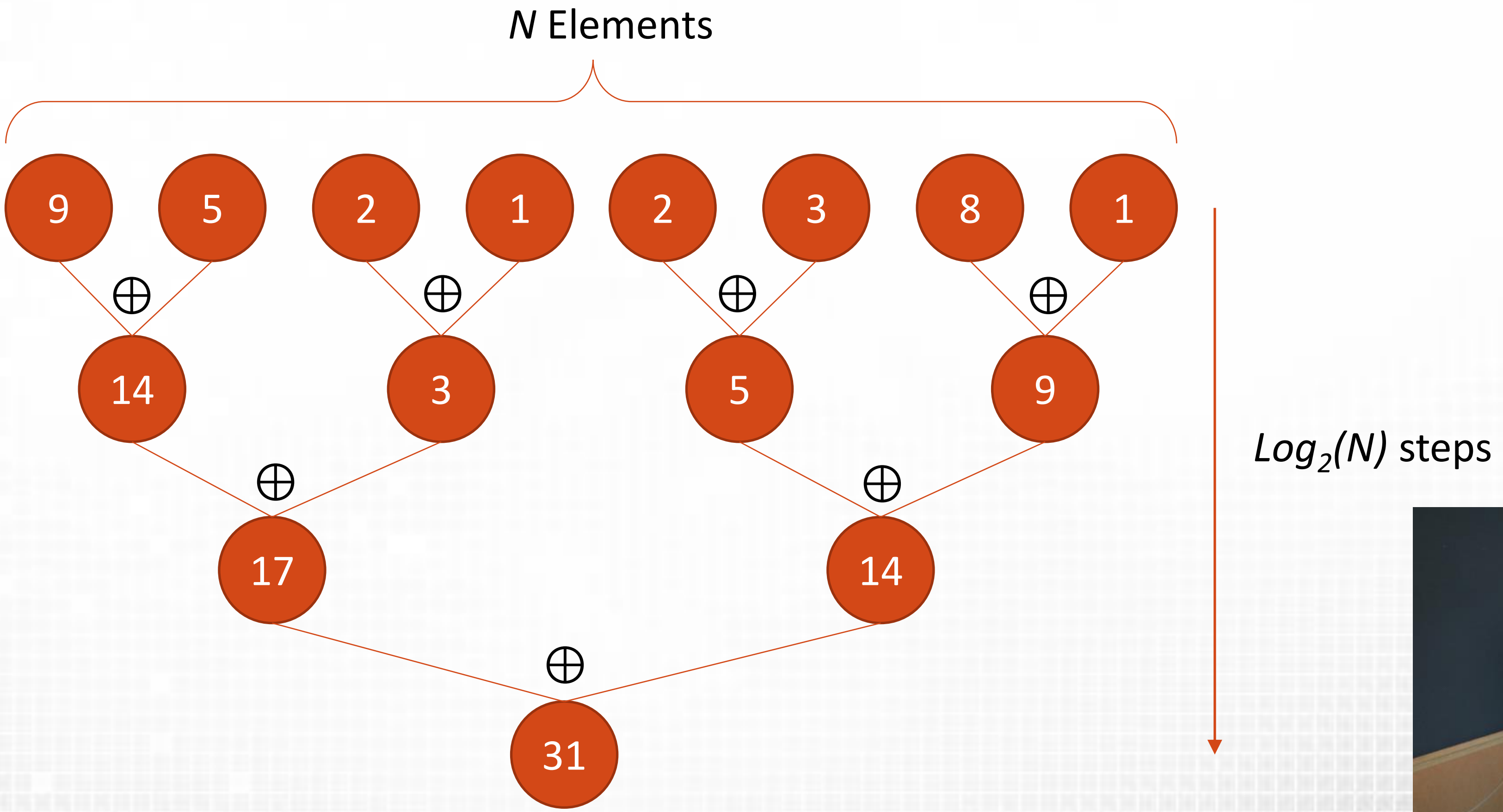
OR

```
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;
}
```

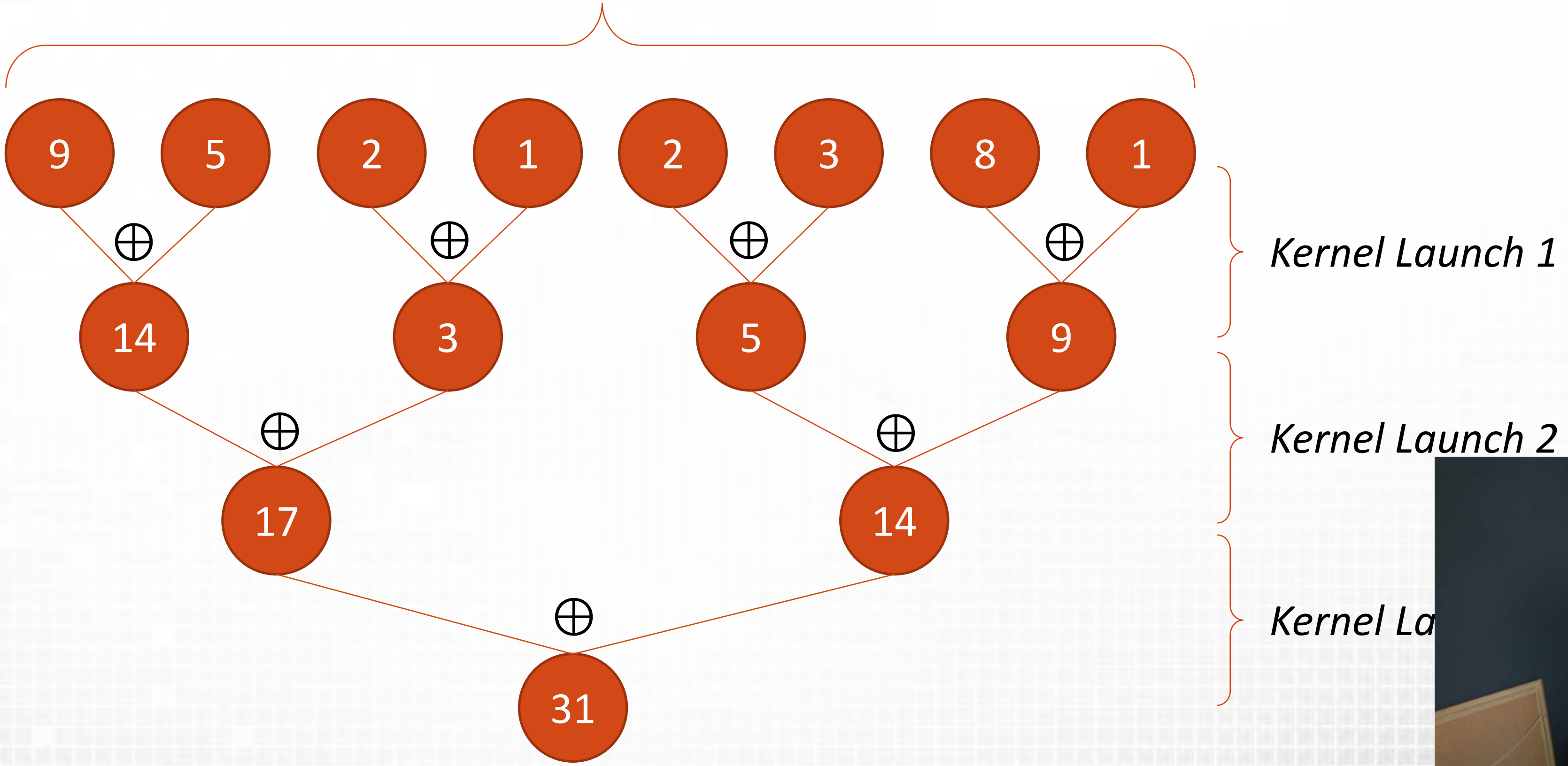
Parallel Reduction

- 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

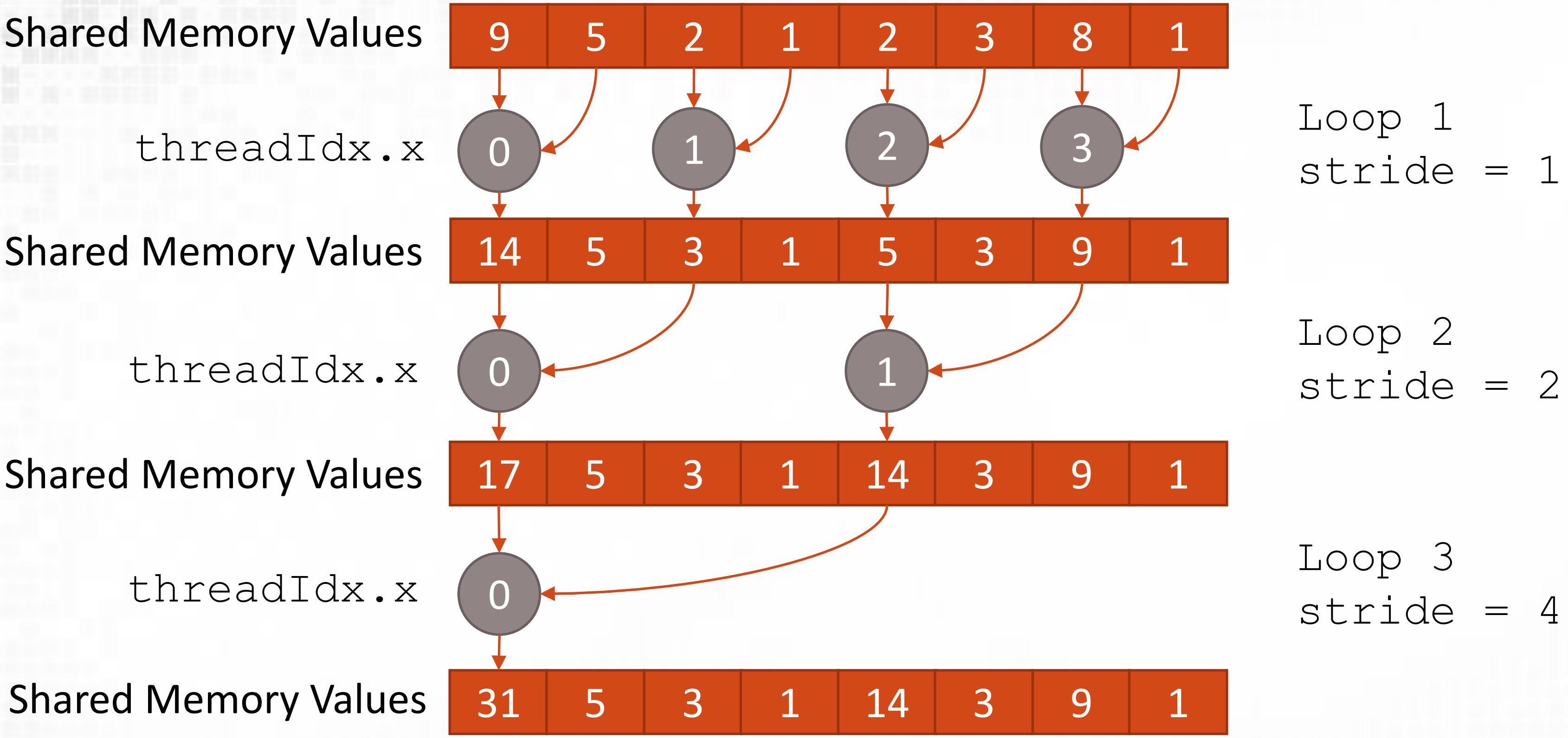
```
__global__ void sum_reduction(float *input, float *block_results){
  extern __shared__ int sdata[];

  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();
}
```

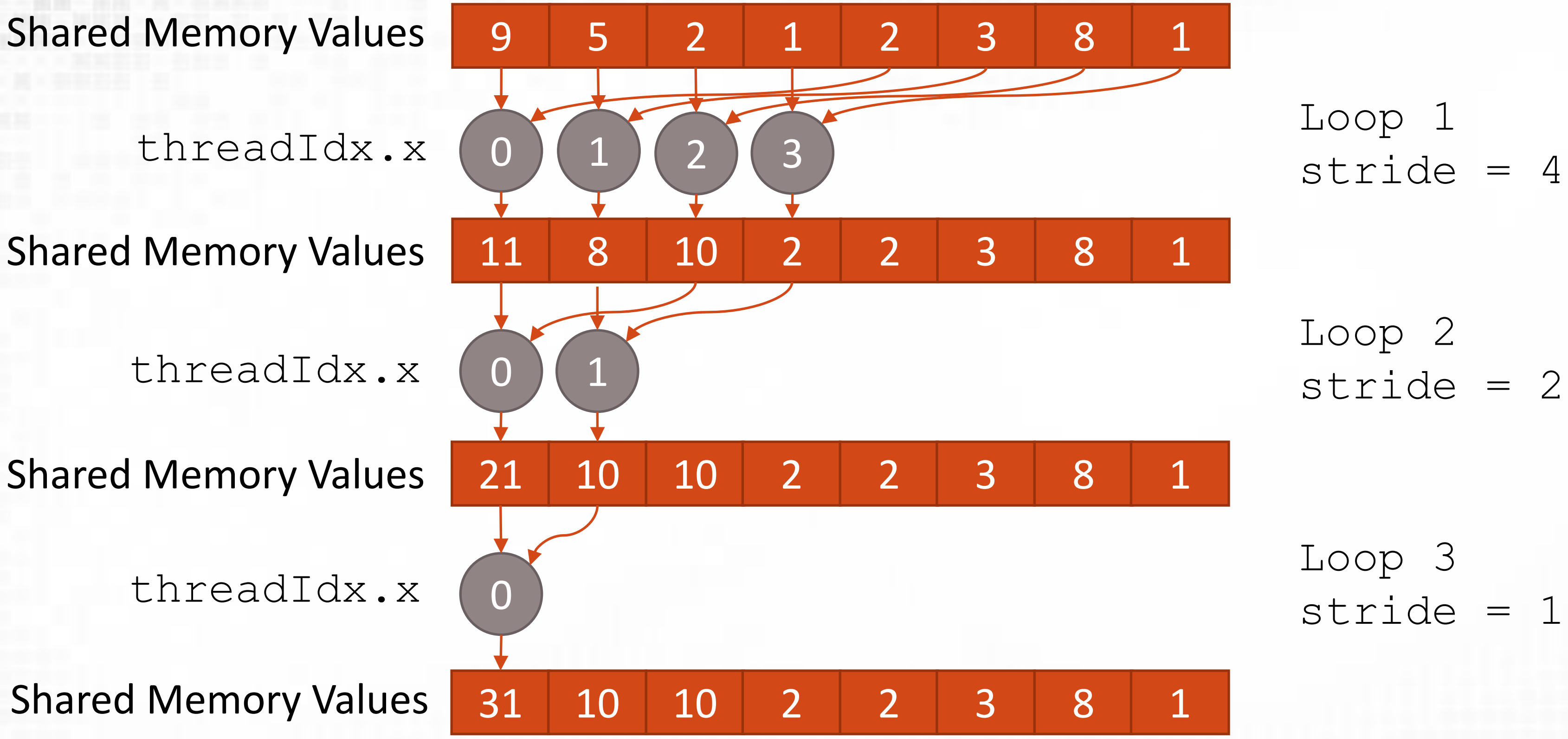
Block Level Reduction

- ❑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

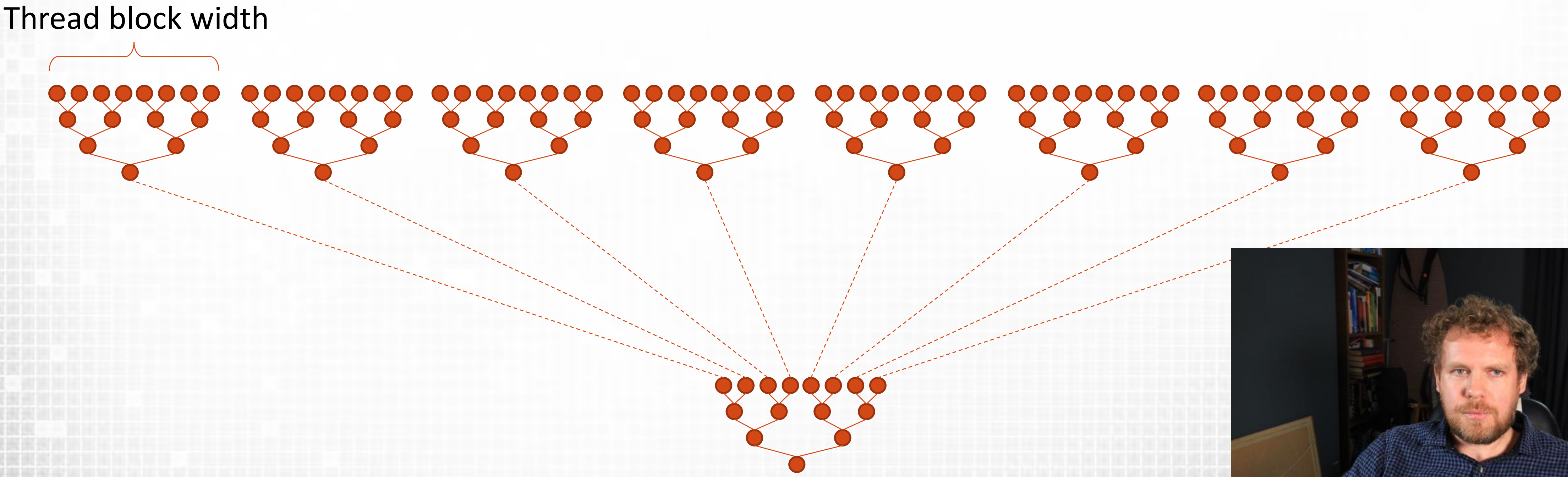
```
for (unsigned int stride = blockDim.x/2; stride > 0; stride>>=1){
  if (threadIdx.x < stride){
    sdata[threadIdx.x] += sdata[threadIdx.x + stride];
  }
  __syncthreads();
}
```

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