CME 213

Spring 2017

Eric Darve

Previously in CME213

- Global memory: coalesced access; warp requests and uses full 128-byte memory segment.
- Shared memory:
  - Avoid bank conflicts; stride should be an odd number
- Occupancy; occupancy calculator (spreadsheet)
- Impact of branching on performance
- Application to finite-difference stencil:
  - Roofline model: memory or compute bound?
  - Arithmetic intensity
  - Finite-difference is memory bound
  - Domain should be square to maximize data reuse and reduce memory traffic

Reductions and scans

- Reduction is a classic problem in parallel computing.
- We will use it to illustrate various computing challenges using CUDA.
- But before we get started:

Challenge!

- Form 2 teams.
- Each team will be a GPU processor.
- You need to calculate a cumulative sum. Example:

```
  3     5     6     2     4
  3     8    14    16    20
```

- Time your group! The fastest wins.

Step 1

- Pick a group: Group 1 or Group 2
- Each member of a group needs to get assigned a unique number or ID. They have to be in consecutive order:

```
1 2 3 4 5 6 7 …
```

- Download the code from **Canvas/Code/CUDA4**

generate_number.cpp

- Compile and run

```
$ g++ generate_number.cpp; ./a.out
```

- Enter your group number, then your ID.
- The code will return a random number.
- This is the number you should use for the cumulative sum.

Step 2

- Take a small piece of paper of size approximately 3in x 3in.
- Top left: write your group number and ID
- Top right: write your random number
- Center: write the result of the cumulative sum.

- Make sure you write the correct ID and random number. Any error will lead to the wrong cumulative sum at the end!

Step 3: winning team

The winning team will be determined based on a score computed using:

Score = Total time [second]
+ 20 second penalty for each incorrect answer

Advice: make sure your final result is correct. The verification can be easily done in parallel. You just need to check the sum of the previous student, add your number and check that it matches your cumulative sum. If there is a match for everyone, the result must be correct!

READY
SET
GO…

Discussion

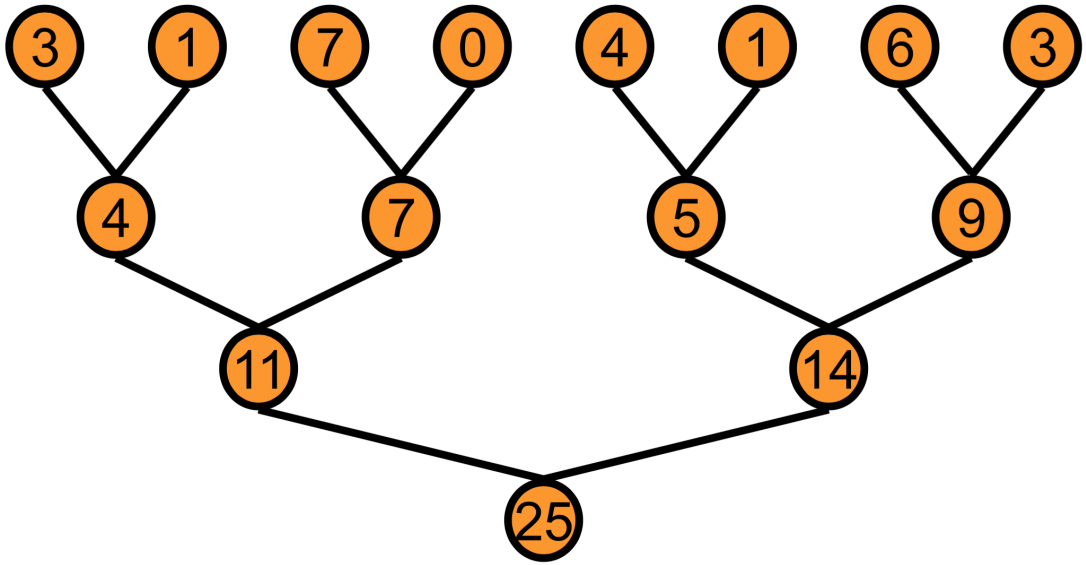
What was the best strategy?

What were the main bottlenecks?

How did you organize your group?

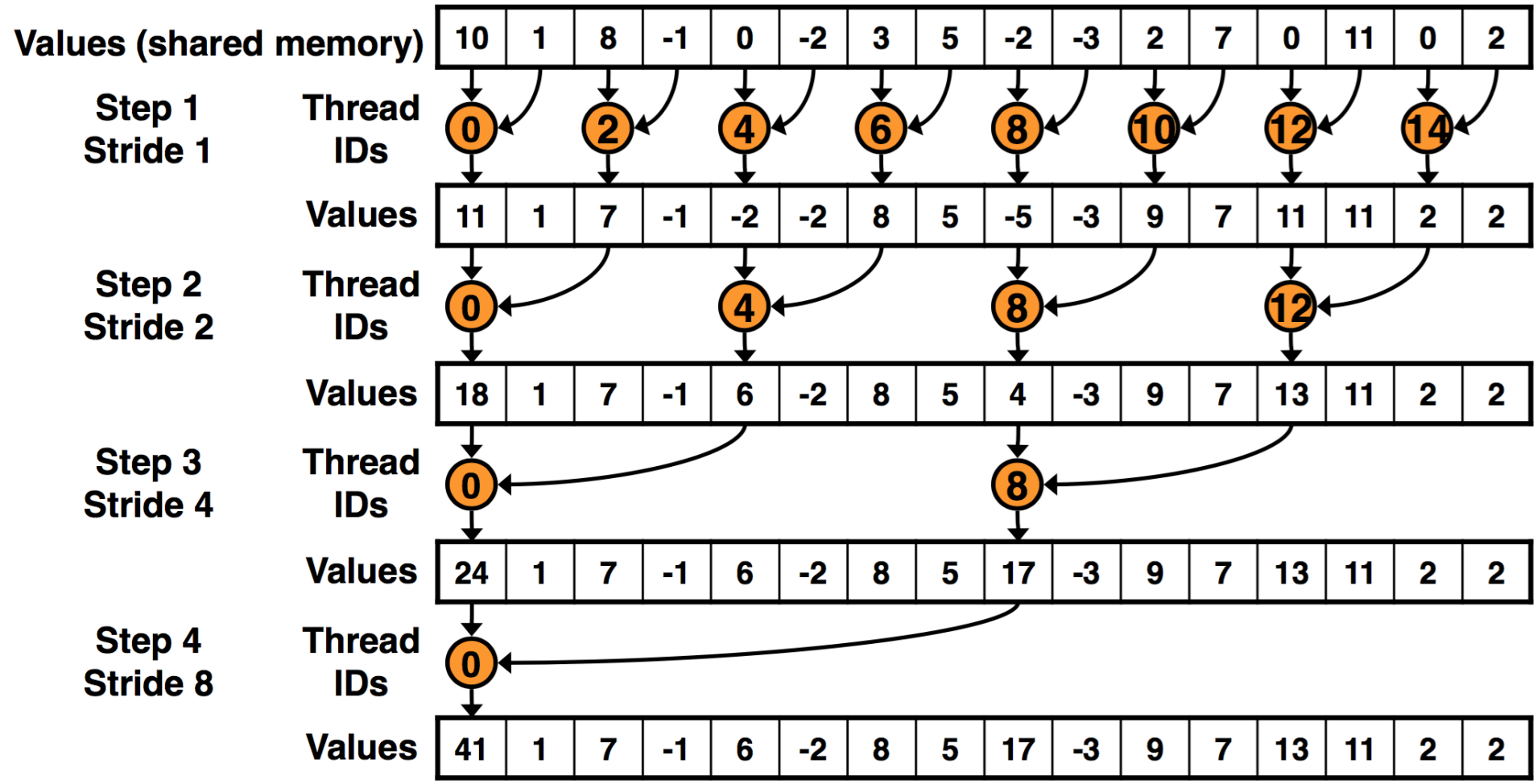
Parallel reduction

- We saw previously how this should be done in parallel: use a reduction tree.
- Let's look how this is going to work on a GPU.
- We need to account for several specific aspects of the hardware.



Kernel o

- Let's start with the simplest implementation.
- We launch a number of blocks.
- Each block performs a reduction.
- We will leave the problem of doing a reduction across multiple blocks to later.
- So now each kernel will output one partial sum per block.


Values (shared memory)
10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2
Step 1
Stride 1
Thread IDs
0 2 4 6 8 10 12 14
Values
11 1 7 -1 -2 -2 8 5 -5 -3 9 7 11 11 2 2
Step 2
Stride 2
Thread IDs
0 4 8 12
Values
18 1 7 -1 6 -2 8 5 4 -3 9 7 13 11 2 2
Step 3
Stride 4
Thread IDs
0 8
Values
24 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2
Step 4
Stride 8
Thread IDs
0
Values
41 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2

Algorithm

- Load data in shared memory
- Use shared memory to perform a tree reduction inside each block.
- Don't forget `__syncthreads` to make sure all threads are done before proceeding to the next stage.

```
template <class T>
__global__ void
reduce0(T* g_idata, T* g_odata, unsigned int n) {
  T* sdata = SharedMemory<T>();

  // load shared mem
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;

  sdata[tid] = (i < n) ? g_idata[i] : 0;

  __syncthreads();

  // do reduction in shared mem
  for(unsigned int s=1; s < blockDim.x; s *= 2) {
    // modulo arithmetic is slow!
    if((tid % (2*s)) == 0) {
      sdata[tid] += sdata[tid + s];
    }

    __syncthreads();
  }

  // write result for this block to global mem
  if(tid == 0) {
    g_odata[blockIdx.x] = sdata[0];
  }
}
```

Performance of kernel 0

Not that great!

```
Reducing array of type int

16777216 elements
512 threads (max)
32768 blocks

Reduction, Throughput = 6.9988 GB/s, Time = 0.00959 s, Size = 1677721

GPU result = 2139353471
CPU result = 2139353471

Test passed
```

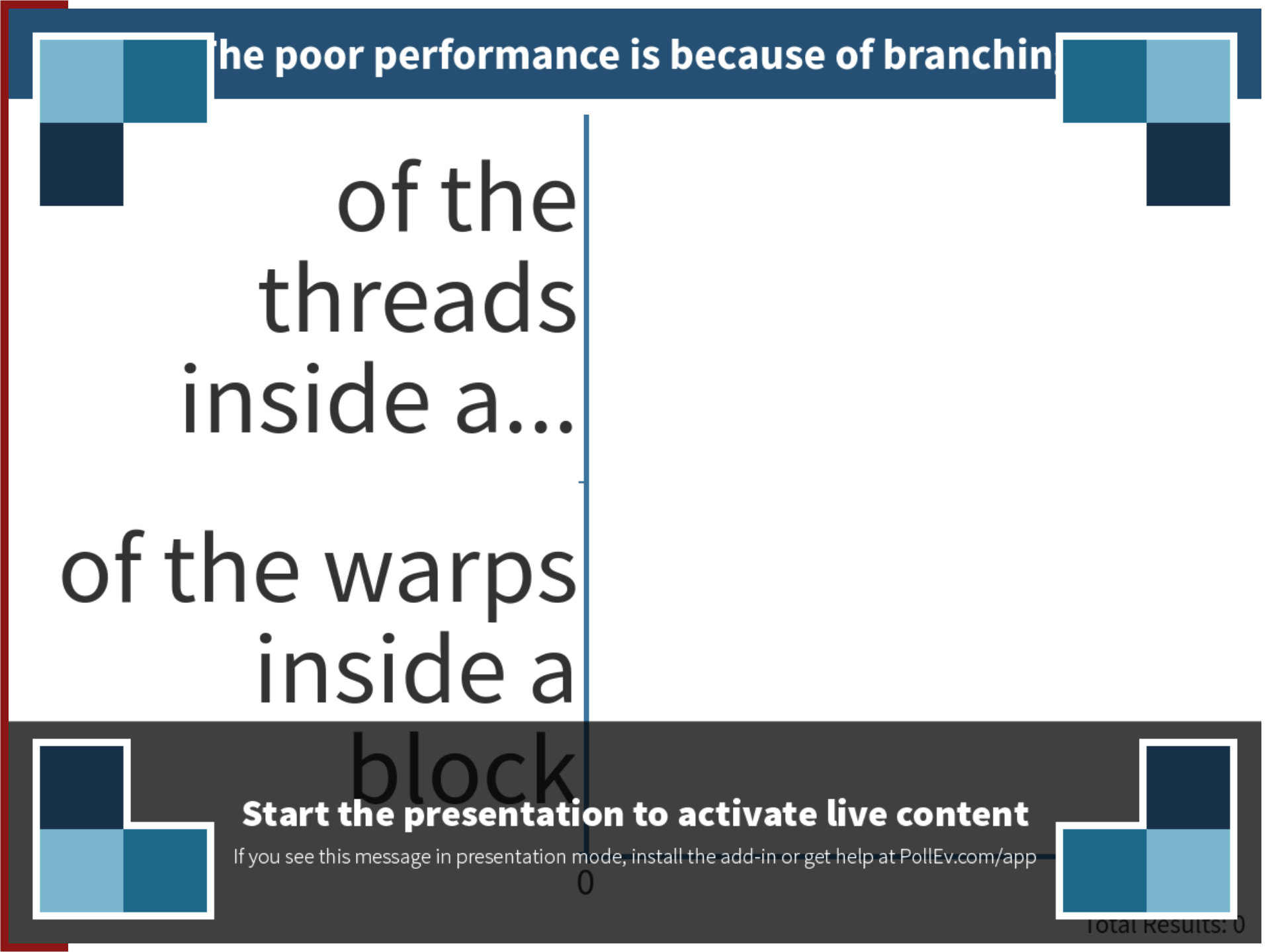
Issues

```
// do reduction in shared mem
for(unsigned int s=1; s < blockDim.x; s *= 2) {
  // modulo arithmetic is slow!
  if((tid % (2*s)) == 0) {
    sdata[tid] += sdata[tid + s];
  }
```

he poor performance is because of branchin

of the threads inside a...

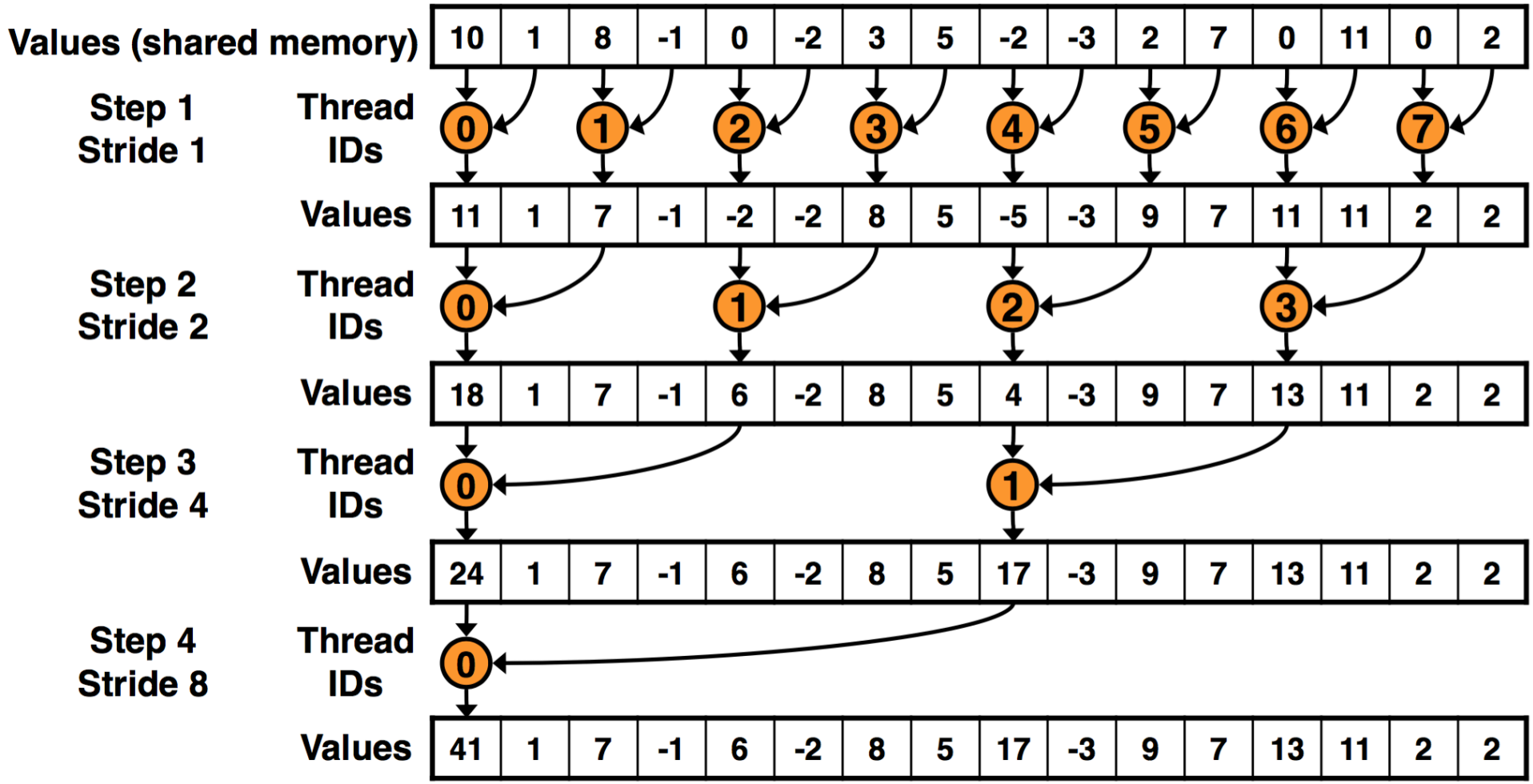
of the warps inside a block



Kernel 1

Let's try to have only a few warps doing most of the work.

This means less thread divergence.



```
// do reduction in shared mem
for(unsigned int s=1; s < blockDim.x; s *= 2) {
  int index = 2 * s * tid;

  if(index < blockDim.x) {
    sdata[index] += sdata[index + s];
  }

  __syncthreads();
}
```

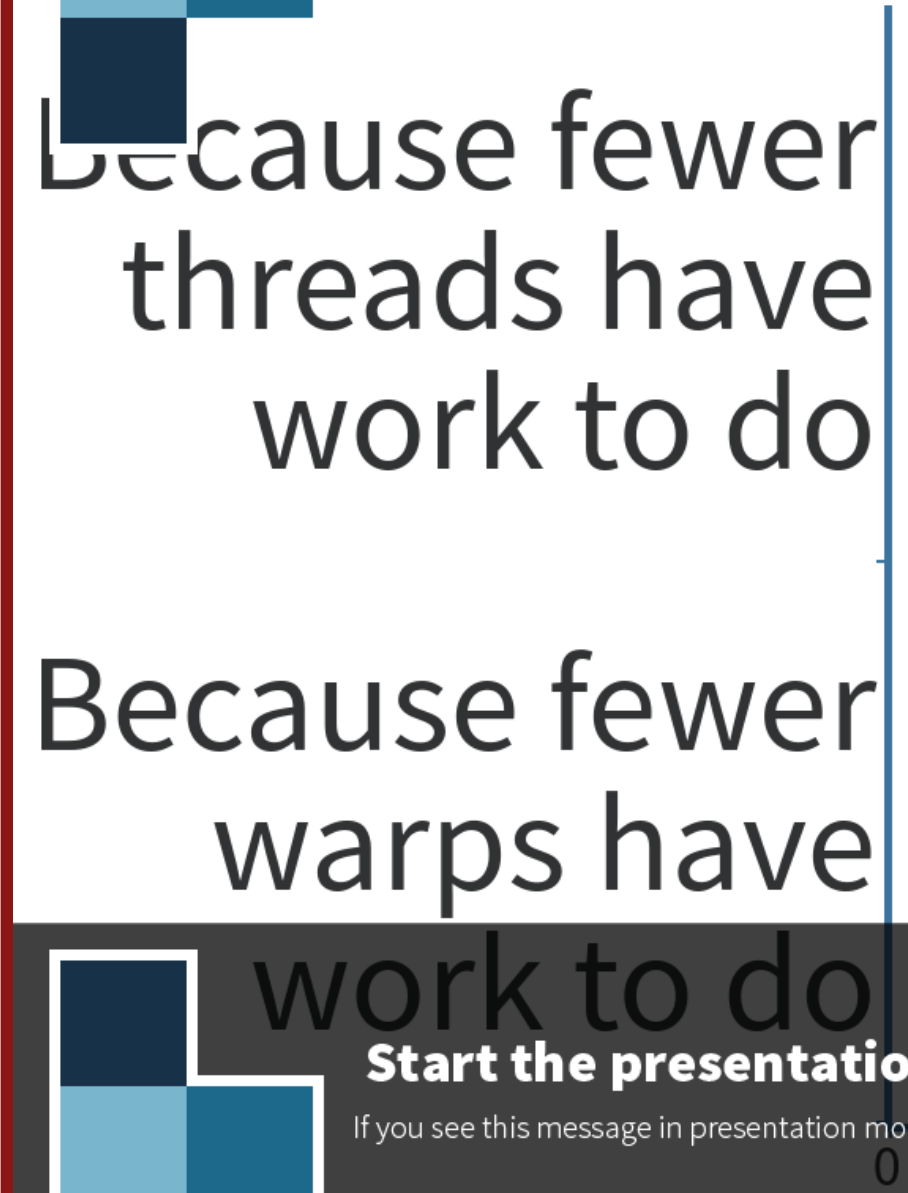
- Operations are performed in the same way in shared memory.
- However, these are done by different threads.
- Towards the end for example, only threads 0 and 1 do work, instead of 0 and 256 in the previous example.

	Throughput GB/s
Kernel 0	7
Kernel 1	9.7

Kernel 1 is faster

Because fewer threads have work to do

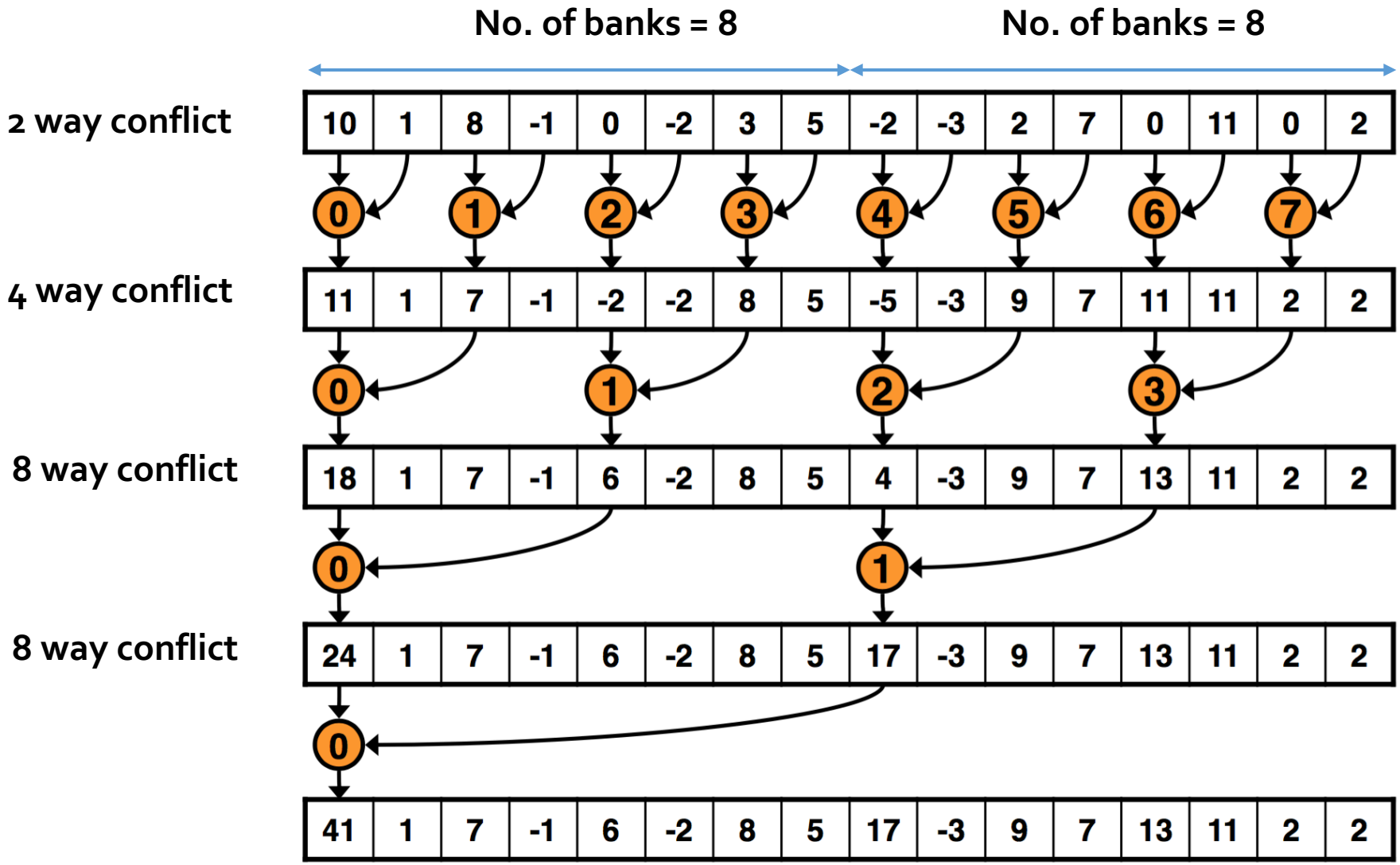
Because fewer warps have work to do



Start the presentation to activate live content

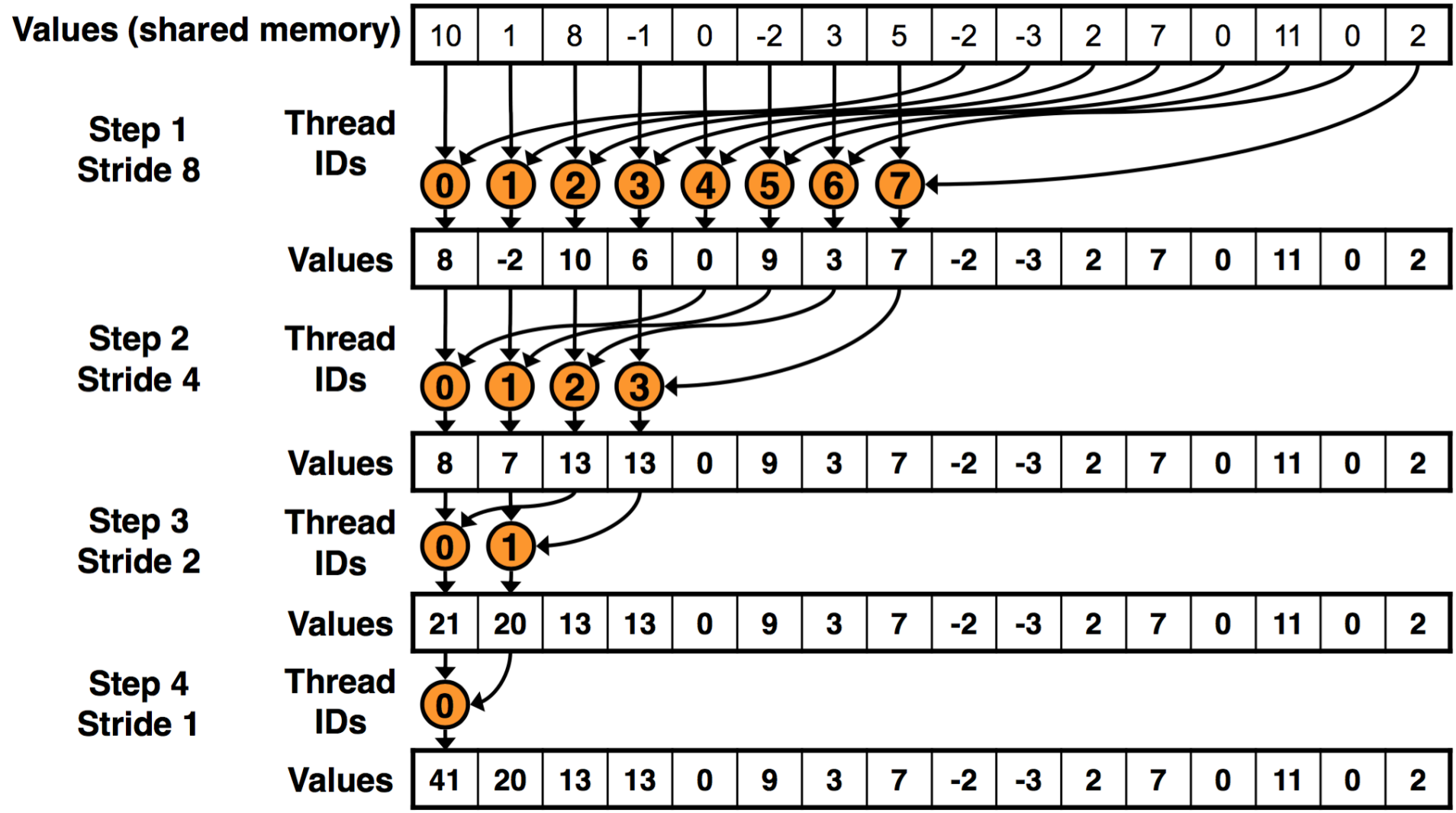
If you see this message in presentation mode, install the add-in or get help at PollEv.com/app

Total Results: 0


No. of banks = 8
No. of banks = 8
2 way conflict
10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2
0 1 2 3 4 5 6 7
4 way conflict
11 1 7 -1 -2 -2 8 5 -5 -3 9 7 11 11 2 2
0 1 2 3
8 way conflict
18 1 7 -1 6 -2 8 5 4 -3 9 7 13 11 2 2
0 1
8 way conflict
24 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2
0
41 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2

Kernel 2

Use sequential memory addressing in the shared memory.



```
// do reduction in shared mem
for(unsigned int s=blockDim.x/2; s>0; s>>=1) {
  if(tid < s) {
    sdata[tid] += sdata[tid + s];
  }

  __syncthreads();
}
```

	Throughput GB/s
Kernel 0	7
Kernel 1	9.7
Kernel 2	13.9

Half of the threads are idle!

```
// do reduction in shared mem
for(unsigned int s=blockDim.x/2; s>0; s>>=1) {
  if(tid < s) {
    sdata[tid] = mySum = mySum + sdata[tid + s];
  }

  __syncthreads();
}
```

- Only half of the threads have work to do.
- Let's change the code a little bit to make sure all threads do at least 1 addition.

```
T mySum = (i < n) ? g_idata[i] : 0;

if(i + blockDim.x < n) {
  mySum += g_idata[i+blockDim.x];
}

sdata[tid] = mySum;
__syncthreads();
```

	Throughput GB/s
Kernel 0	7
Kernel 1	9.7
Kernel 2	13.9
Kernel 3	29

Unrolling

- The next steps get more complicated.
- We need the compiler to be more effective.
- The last warp is a bit different.
- All threads in that warp are by definition synchronized.
- `__syncthreads()` is not needed.
- Let's just write a specialized routine for this.

Last warp

```
// do reduction in shared mem
for(unsigned int s=blockDim.x/2; s>32; s>>=1) {
  if(tid < s) {
    sdata[tid] += sdata[tid + s];
  }

  __syncthreads();
}

// fully unroll reduction within a single warp
if(tid < 32) {
  warpReduce<T,blockSize>(sdata, tid);
}
```

We write separate code for the very end.

```
template <class T, unsigned int blockSize>
__device__ void warpReduce(volatile T* sdata, int tid) {
  if(blockSize >= 64) {
    sdata[tid] += sdata[tid + 32];
  }

  if(blockSize >= 32) {
    sdata[tid] += sdata[tid + 16];
  }

  if(blockSize >= 16) {
    sdata[tid] += sdata[tid + 8];
  }

  if(blockSize >= 8) {
    sdata[tid] += sdata[tid + 4];
  }

  if(blockSize >= 4) {
    sdata[tid] += sdata[tid + 2];
  }

  if(blockSize >= 2) {
    sdata[tid] += sdata[tid + 1];
  }
}
```

- All the `if` statements are optimized away by the compiler.
- We manually write out all the operations so the compiler can produce better code.

Is branching causing a thread divergence?

Yes

No

Start the presentation to activate live content

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app

0

Total Results: 0

That was actually worth it

	Throughput GB/s
Kernel 0	7
Kernel 1	9.7
Kernel 2	13.9
Kernel 3	29
Kernel 4	45.7

Let's do this for the main loop as well

```
#pragma unroll
  for(unsigned int s=blockSize/2; s>32; s>>=1) {
    if(tid < s) {
      sdata[tid] += sdata[tid + s];
    }

    __syncthreads();
  }
```

- The loop size is known at compile time.
- So the compiler can completely unroll this.

One more step closer to heaven

	Throughput GB/s
Kernel 0	7
Kernel 1	9.7
Kernel 2	13.9
Kernel 3	29
Kernel 4	45.7
Kernel 5	59.8

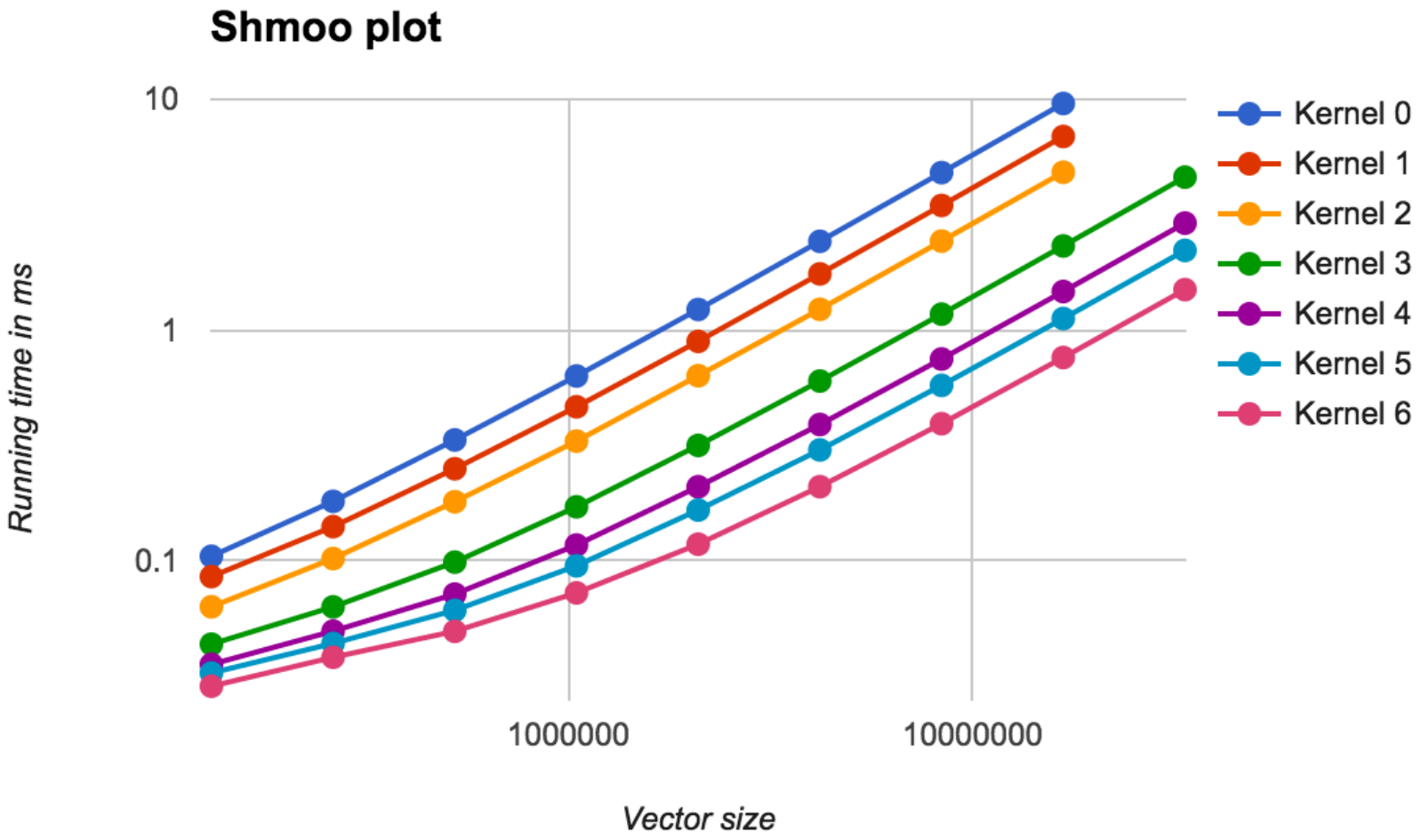
One more obvious optimization

- Right now, we are using one thread per entry in the vector (actually, no. of threads = n/2).
- However, it is more efficient to fix the total number of threads, and have each thread do a local reduction first.
- This produces better code because doing a local reduction is more efficient than doing everything using a reduction tree.
- In addition, this produces fewer partial sum values at the end. (Remember that so far we only have each block produce a partial sum.)

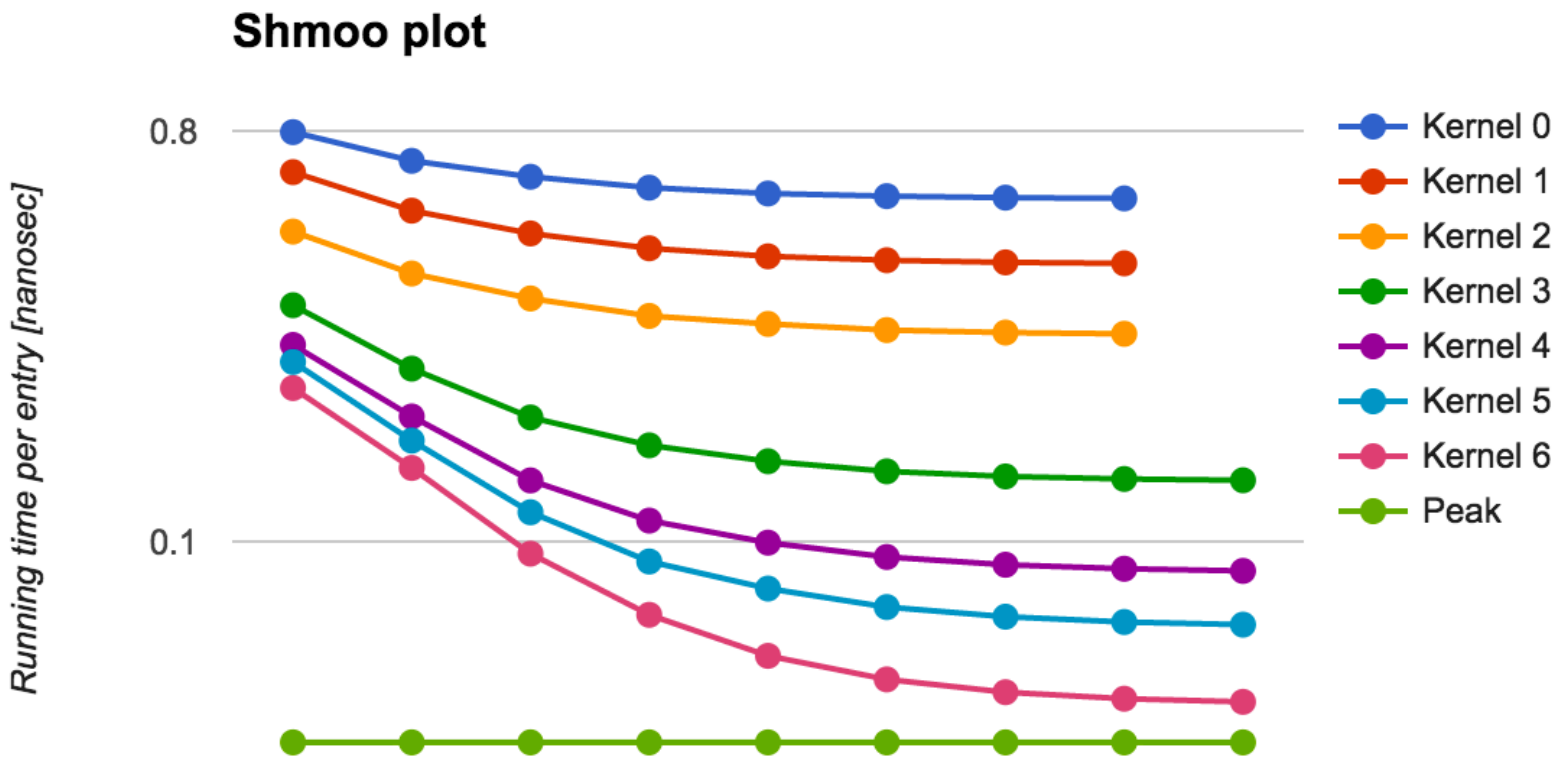
```
while(i < n) {
  mySum += g_idata[i];
  i += gridSize;
}

// each thread puts its local sum into shared memory
sdata[tid] = mySum;
__syncthreads();
```

	Throughput GB/s
Kernel 0	7
Kernel 1	9.7
Kernel 2	13.9
Kernel 3	29
Kernel 4	45.7
Kernel 5	59.8
Kernel 6	88.1

The great shmoo plot



Running time per vector entry



One final point

How should we do the reduction across the different blocks?

Different options:

1. Write another kernel that runs with a single block and does the final reduction. Time to run final kernel is negligible.
2. Send partial sum results back to CPU and sum on CPU. Also fast whenever possible.
3. Use atomics:

```
T atomicAdd(T* address, T val);
```

```
atomicAdd(g_odata, sdata[0]);
```

Atomics table

- `atomicAdd()`
- `atomicSub()`
- `atomicExch()` exchange
- `atomicMin()`
- `atomicMax()`
- `atomicInc()` increment integer
- `atomicDec()`
- `atomicCAS()` compare and exchange
- `atomicAnd()` bitwise operation
- `atomicOr()`
- `atomicXor()`