CME 213

Spring 2019

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 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

- Each group of players is assigned a unique group ID
- Download the code from class web page:

```
generate_sequence.cpp
```

- Compile and run

```
$ g++ -std=c++11 generate_sequence.cpp; ./a.out
```

- Enter your group number
- The code returns a sequence of random numbers.
- This is the sequence you will use for the cumulative sum.

```
[darve@omp:~$ ./a.out
Enter your group number (an integer greater or equal to 1)
4
Selected group ID: 4
Row 1: index
Row 2: random value

Index  1 to 10
1           2           3           4           5           6           7           8           9           10
879         403         503         630         459         682         973         970         843         677

Index 11 to 20
11          12          13          14          15          16          17          18          19          20
150         986         512         853         140         529         150         196         492         529

Index 21 to 30
21          22          23          24          25          26          27          28          29          30
941         285         271         694         122         802         819         966         796         501

Index 31 to 40
31          32          33          34          35          36          37          38          39          40
580         331         423         573         423         700         231         607         849         756
```

Step 2

Take small pieces of paper of size approximately 3in x 3in.

- **Top left:** write the group number and your index
- **Top right:** write the corresponding random number
- **Center:** write the result of the cumulative sum.

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

Step 3: rules!

- A **memloc** is a piece of paper with a single number written on it. You can strike a number to write a new one. A memloc is full when it has a number on it, and empty otherwise.
- There are 3 types of players: mem, net, pu. Each player has only one type. You can have as many players of each type as you want.
- **mem player:** can copy and strikethrough numbers.
- **net player:** can take/give memlocs to **mem** and **pu**. Cannot hold more than 3 memlocs at a time.
- **pu player:** can take two full memlocs and one empty memloc, add the numbers on the full memlocs (may use a calculator) and write the result on the empty memloc. Can take/give memlocs to **net**. Cannot hold more than 3 memlocs at a time.

Step 4: pick a team name!

Games

- Game 1: players can locate themselves anywhere
- Game 2: there must be a distance of at least 30 meters between mem and pu players.
- Think about how many mem, net, and pu players you want to have on your team.

Team

Time:

Errors:

Team

Time:

Errors:

Team

Time:

Errors:

Team

Time:

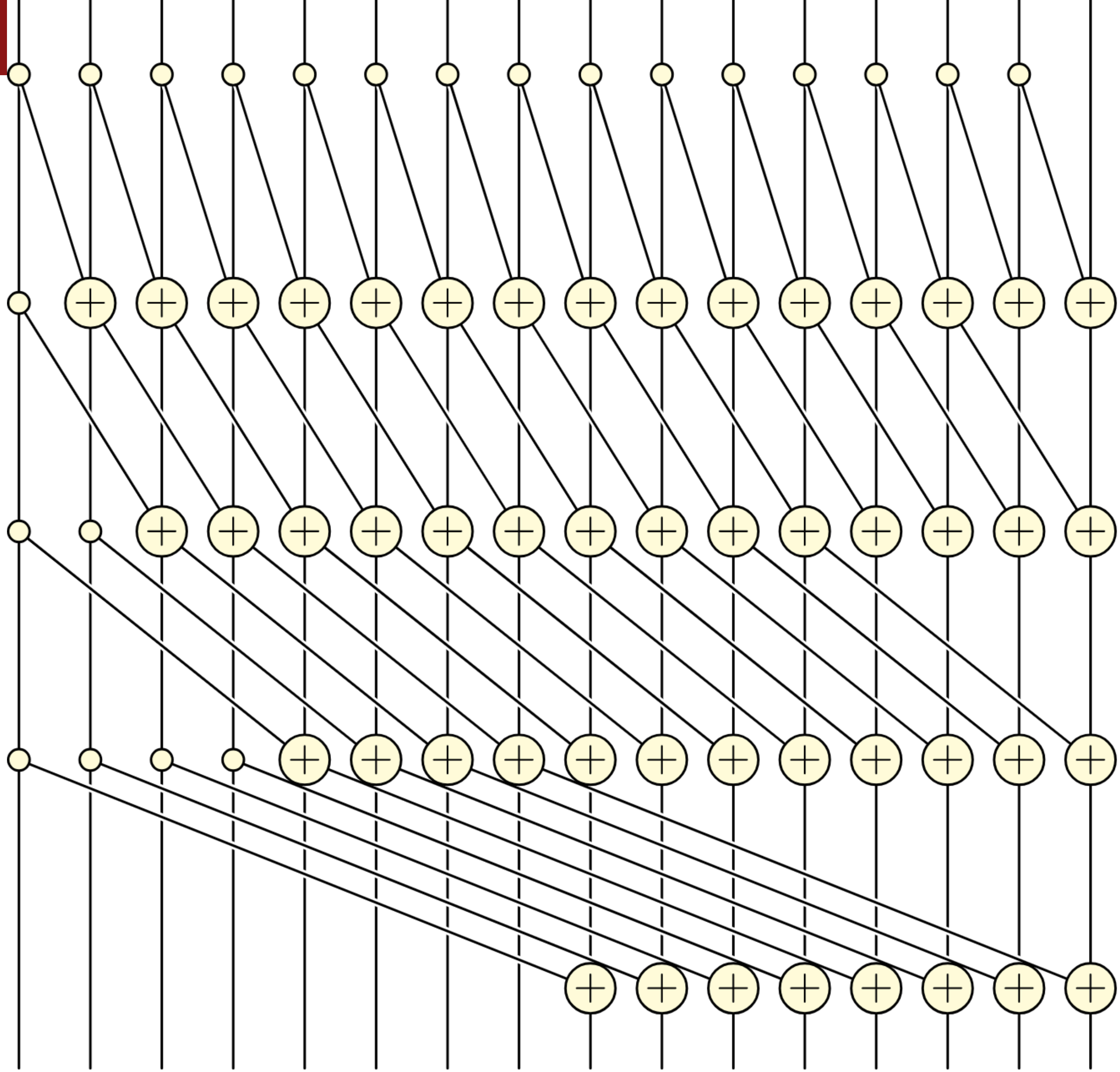
Errors:

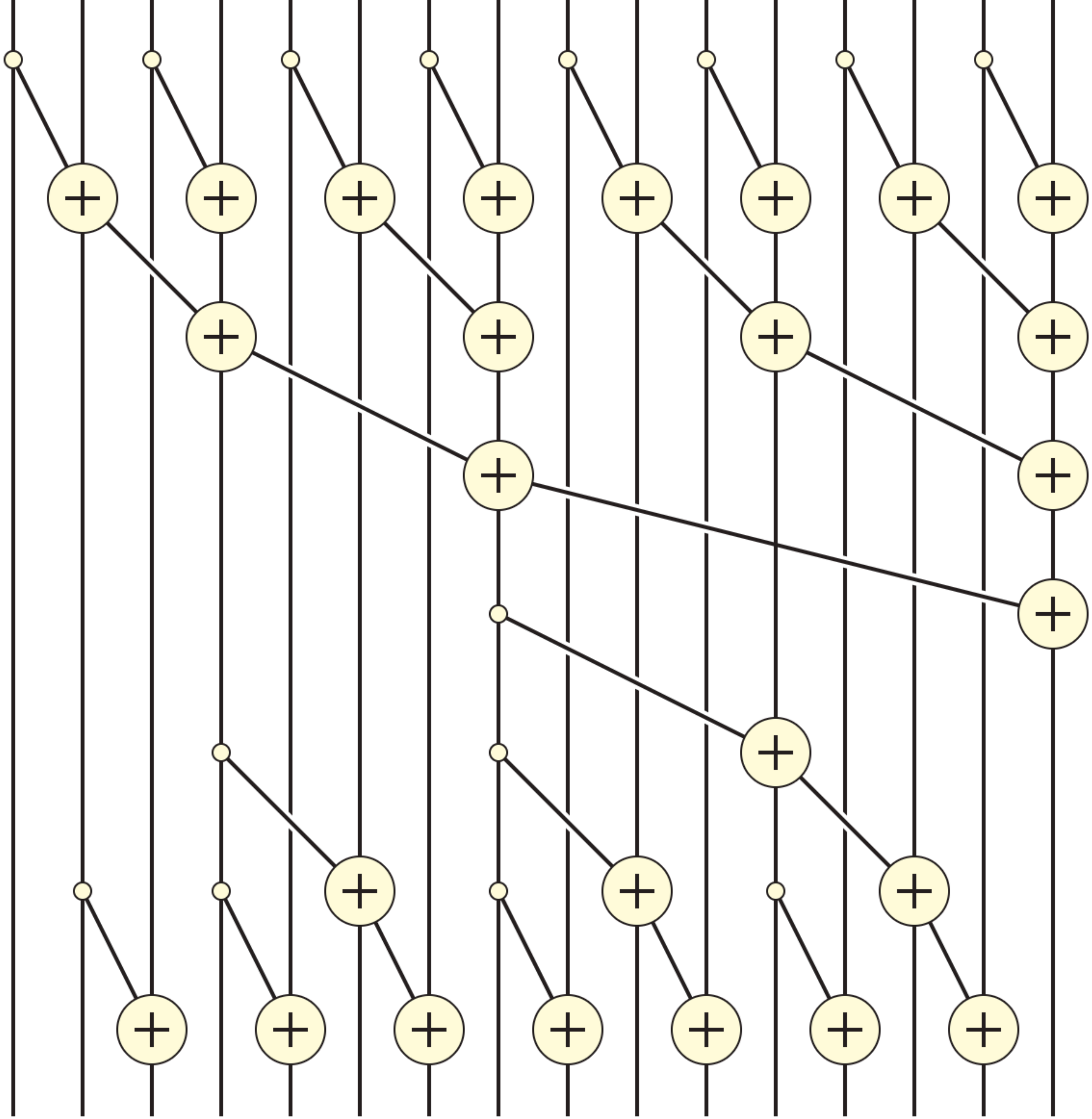
Discussion

What was the best strategy?

What were the main bottlenecks?

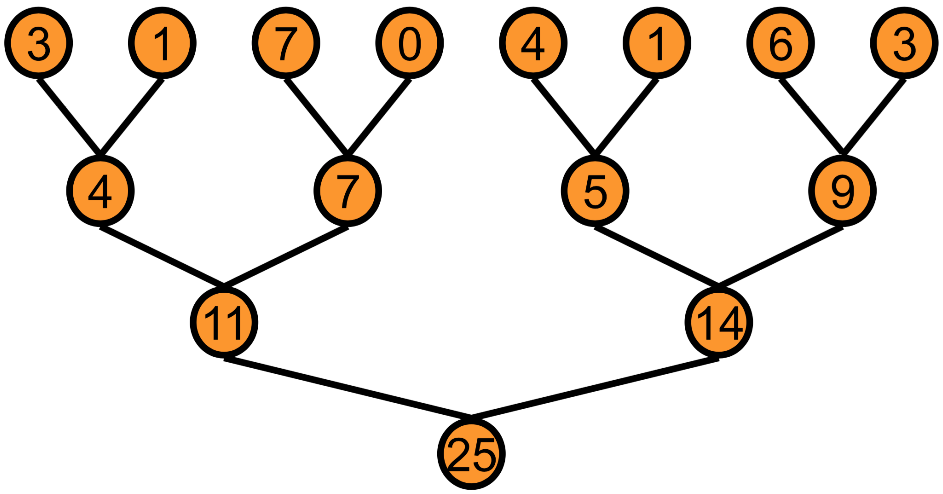
How did you organize your group?





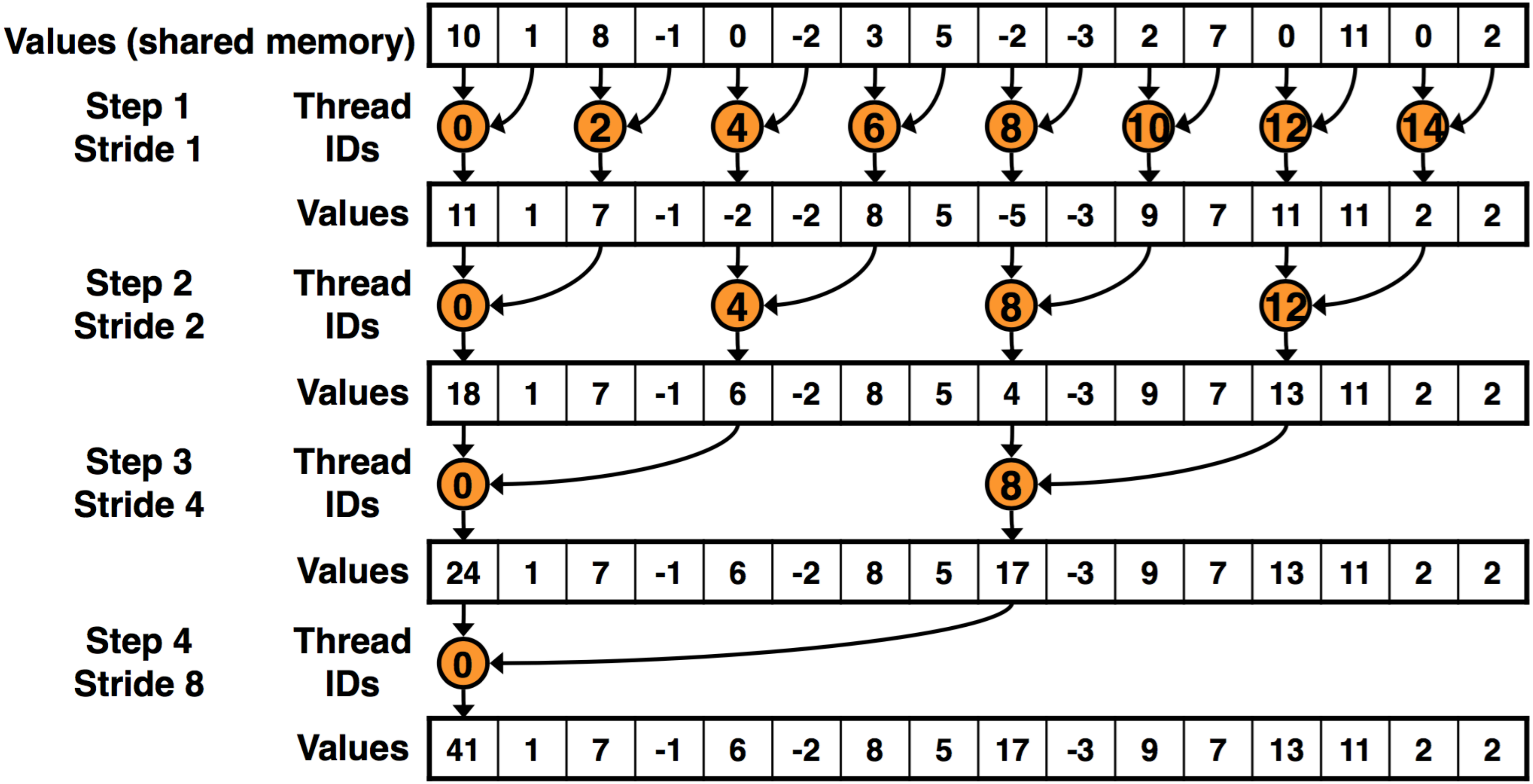
Parallel reduction

- 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 0

- 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!

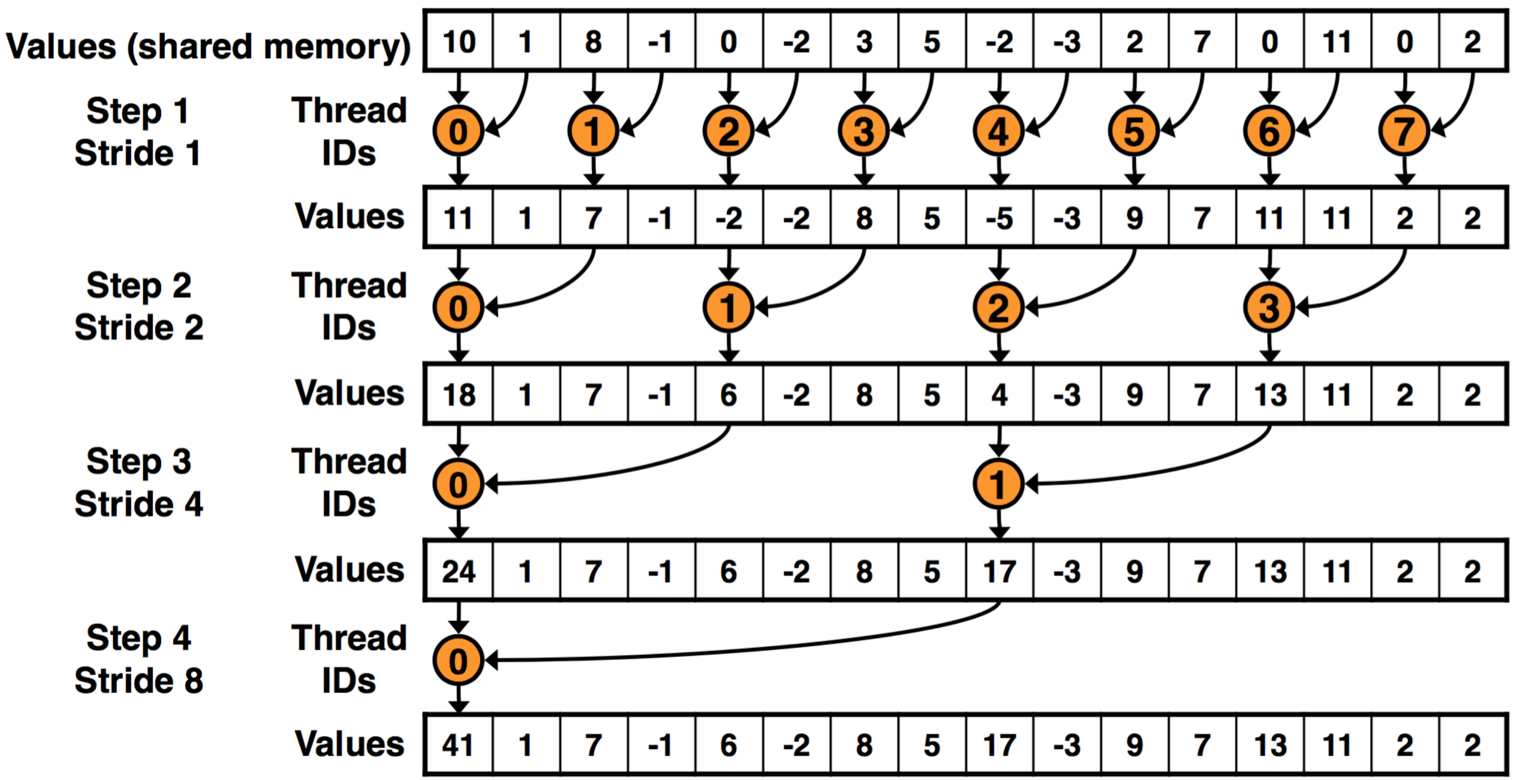
```
Reduction, Throughput = 12.1446 GB/s, Time = 0.00553 s, Size

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];
    }

    __syncthreads();
}
```

Kernel 1

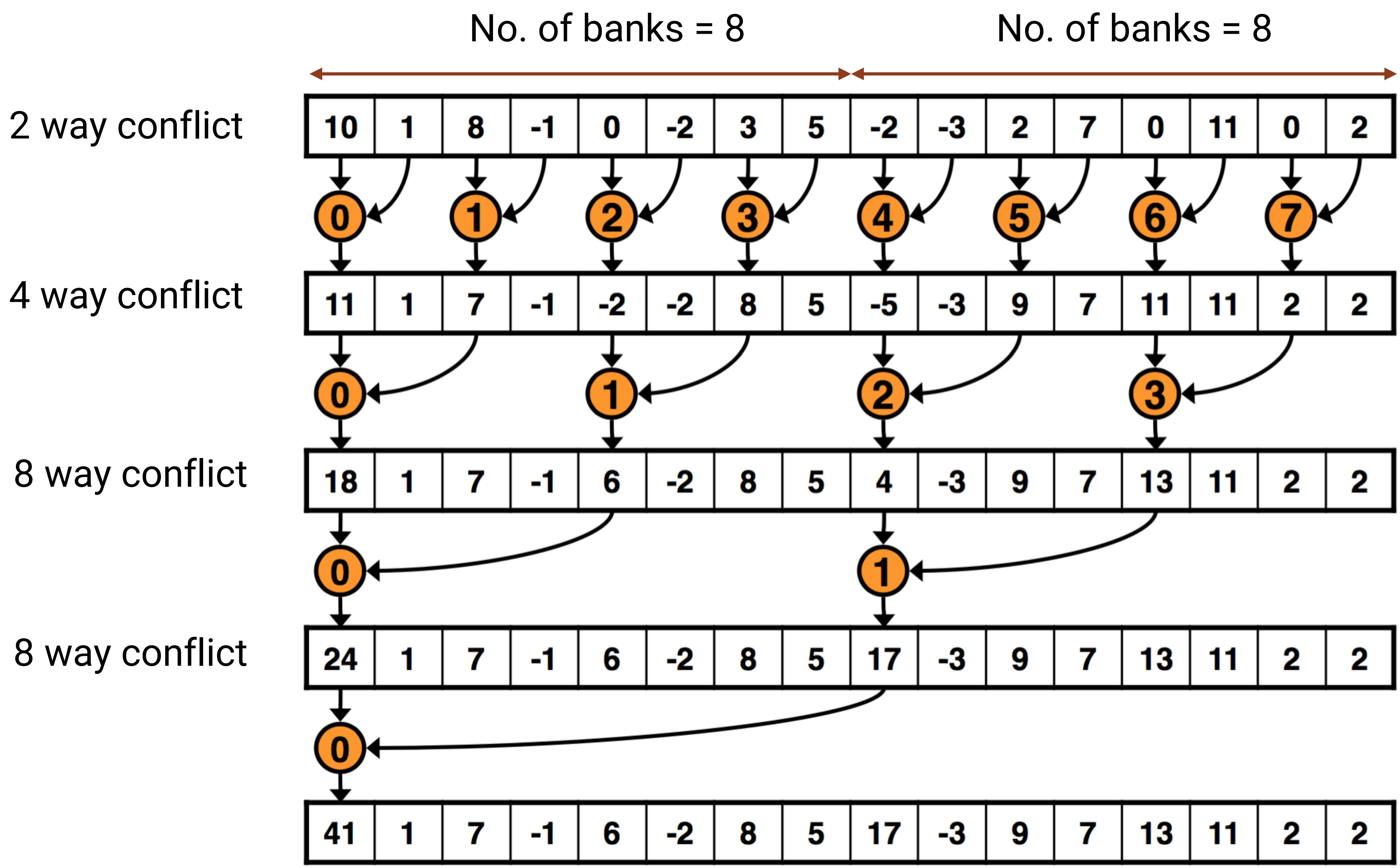
Let's try to have only a few warps doing most of the work.
This means less thread divergence.



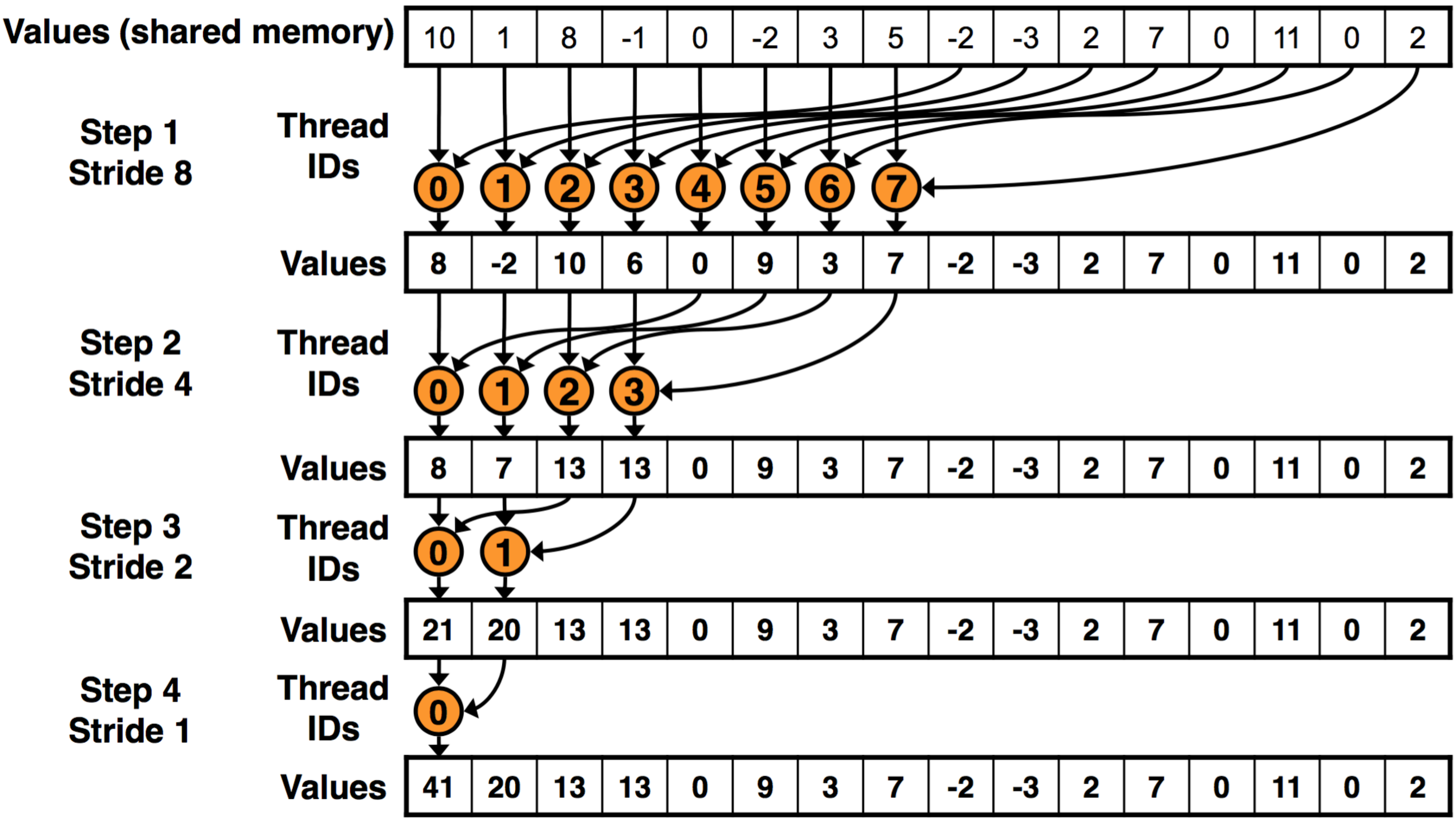
```
// do reduction in shared mem
for(unsigned int s=1; s < blockDim.x; s *= 2) {
    int index = 2 * s * tid;

    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	16
Kernel 1	20


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


Values (shared memory)
10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2
Step 1
Stride 8
Thread IDs
0 1 2 3 4 5 6 7
Values
8 -2 10 6 0 9 3 7 -2 -3 2 7 0 11 0 2
Step 2
Stride 4
Thread IDs
0 1 2 3
Values
8 7 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2
Step 3
Stride 2
Thread IDs
0 1
Values
21 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2
Step 4
Stride 1
Thread IDs
0
Values
41 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2

```
// 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	16
Kernel 1	20
Kernel 2	31

Half of the threads are idle!

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.

```
// 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();
}
```

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

if(i + blockSize < n) {
    mySum += g_idata[i+blockSize];
}

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

	Throughput GB/s
Kernel 0	16
Kernel 1	20
Kernel 2	31
Kernel 3	63

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

We write separate code for the very end.

```
// 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);
}
```

```
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.

That was actually worth it

	Throughput GB/s
Kernel 0	16
Kernel 1	20
Kernel 2	31
Kernel 3	63
Kernel 4	70

Let’s do this for the main loop as well

```
    // do reduction in shared mem
#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	16
Kernel 1	20
Kernel 2	31
Kernel 3	63
Kernel 4	70
Kernel 5	74

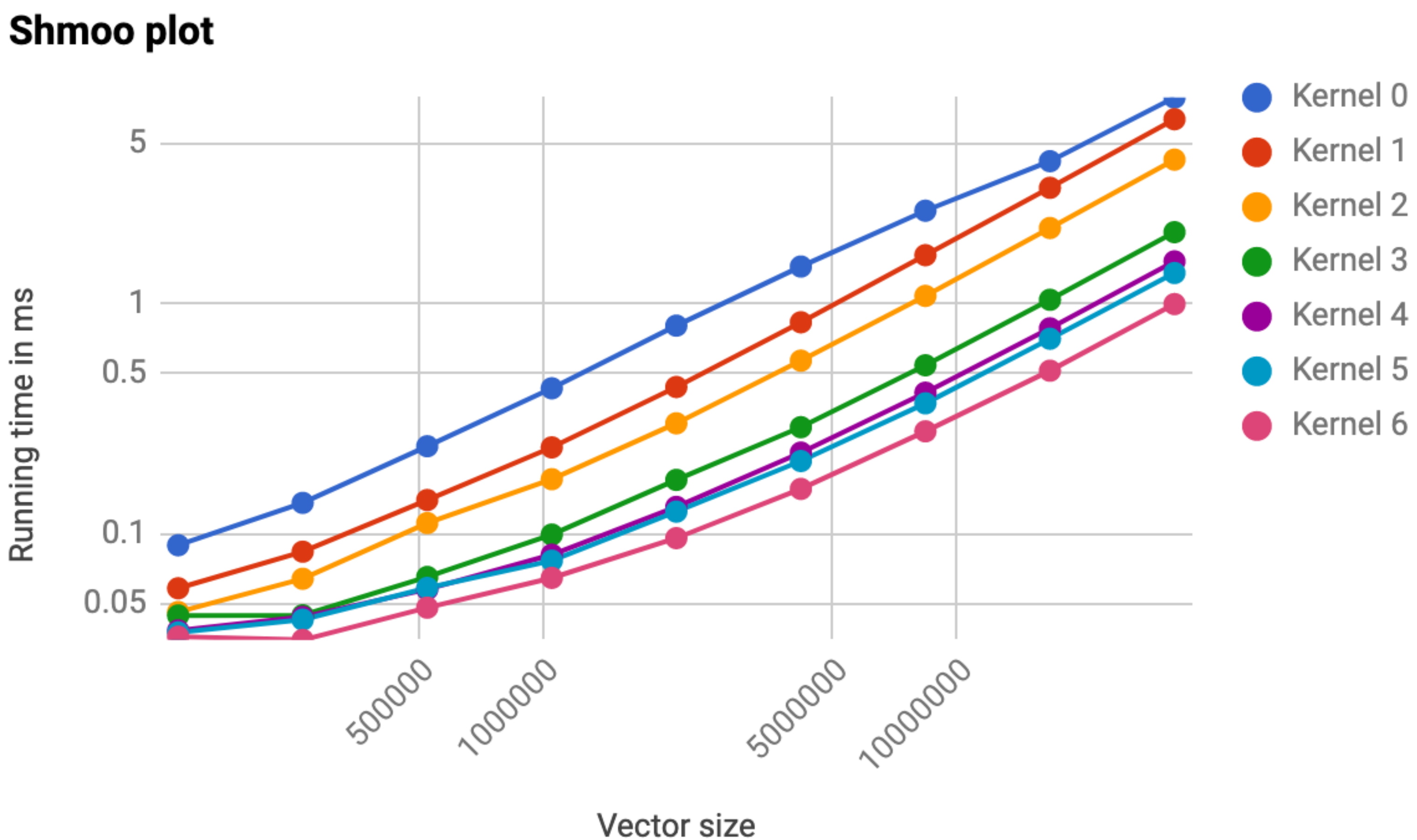
One more obvious optimization

- 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.)

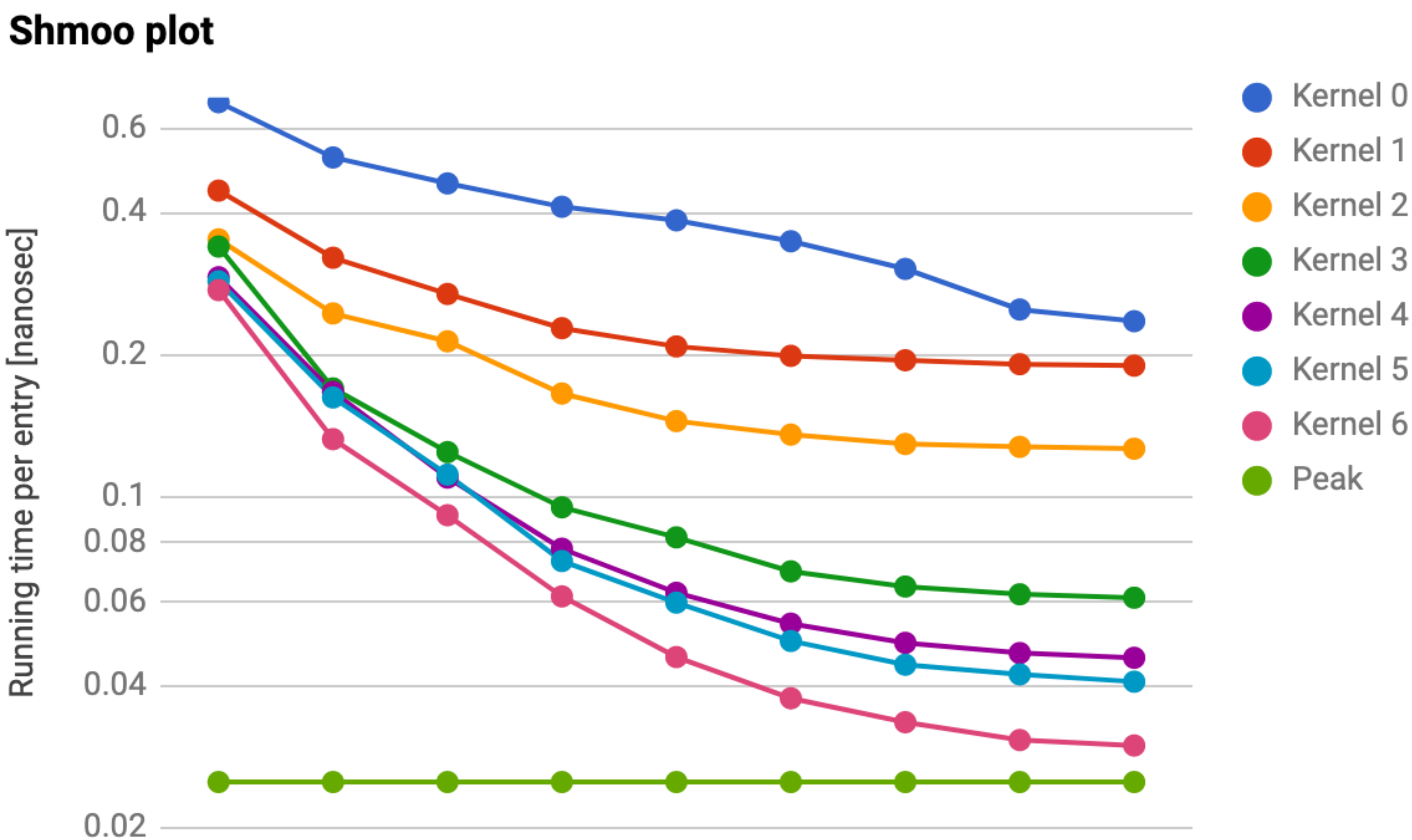
```
while(i < n) {
    mySum += g_idata[i];
    i += gridSize;
}

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

	Throughput GB/s
Kernel 0	16
Kernel 1	20
Kernel 2	31
Kernel 3	63
Kernel 4	70
Kernel 5	74
Kernel 6	105

The great shmoo plot



Running time per vector entry



One final point

How should we do the reduction across the different blocks?

Different options:

- Write another kernel that runs with a single block and does the final reduction. Time to run final kernel is negligible.
- Send partial sum results back to CPU and sum on CPU. Also fast whenever possible.
- 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()`	