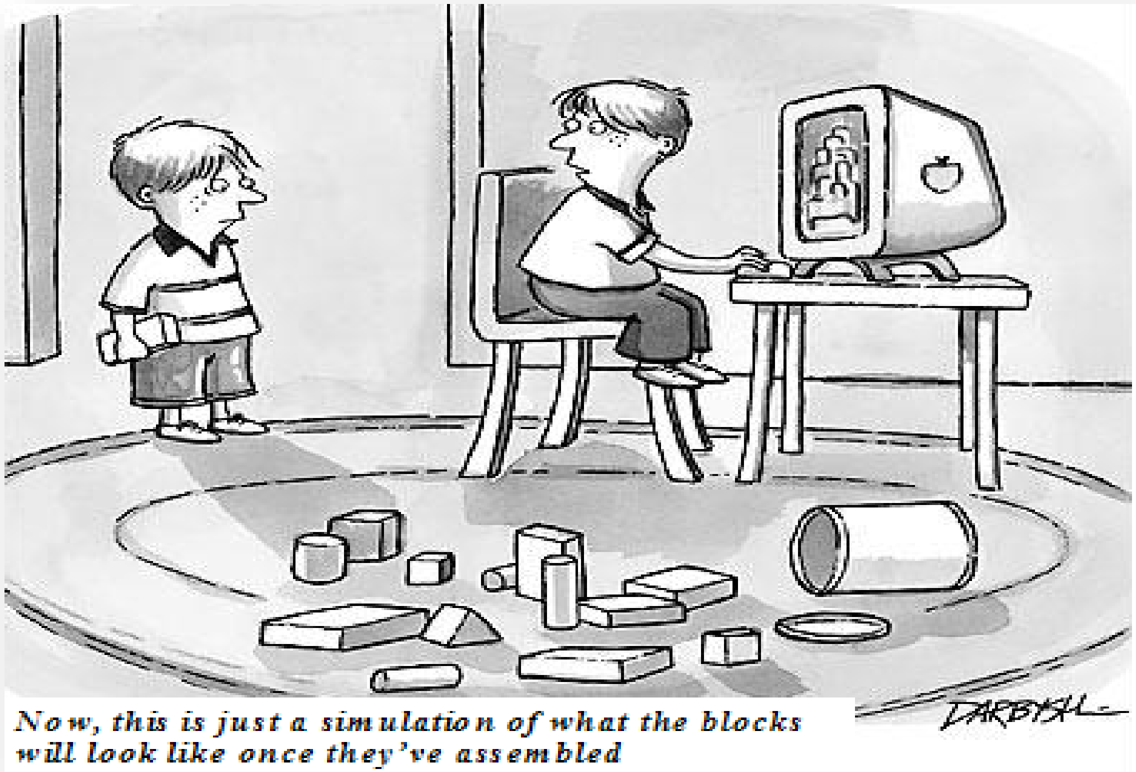
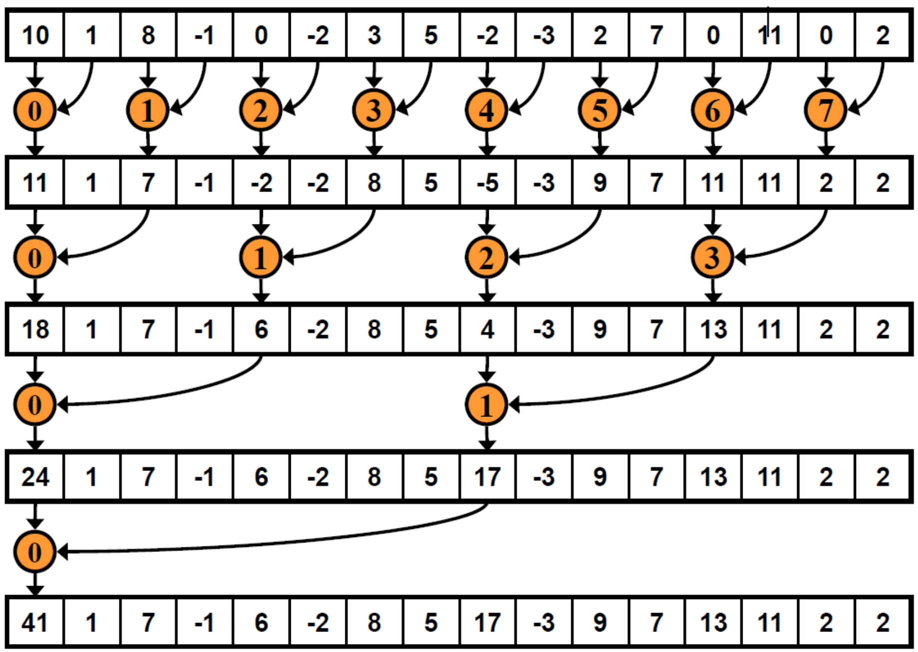
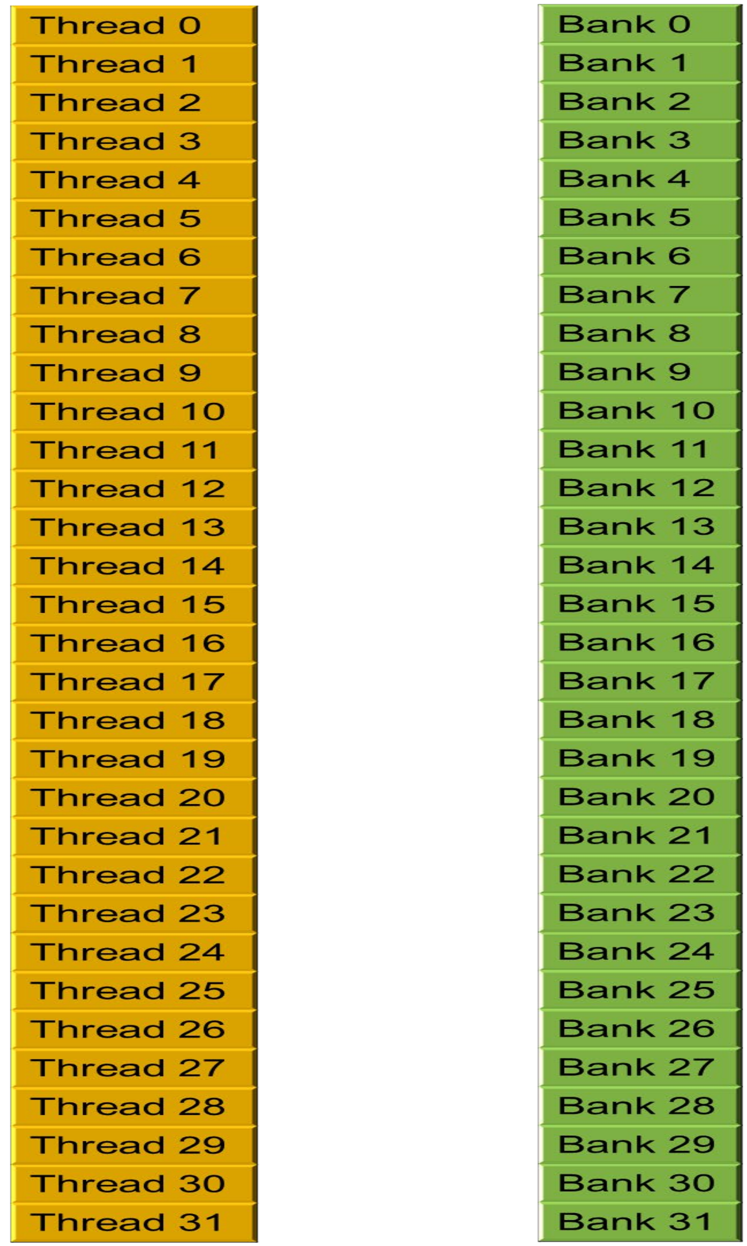
ECE569
Module 36



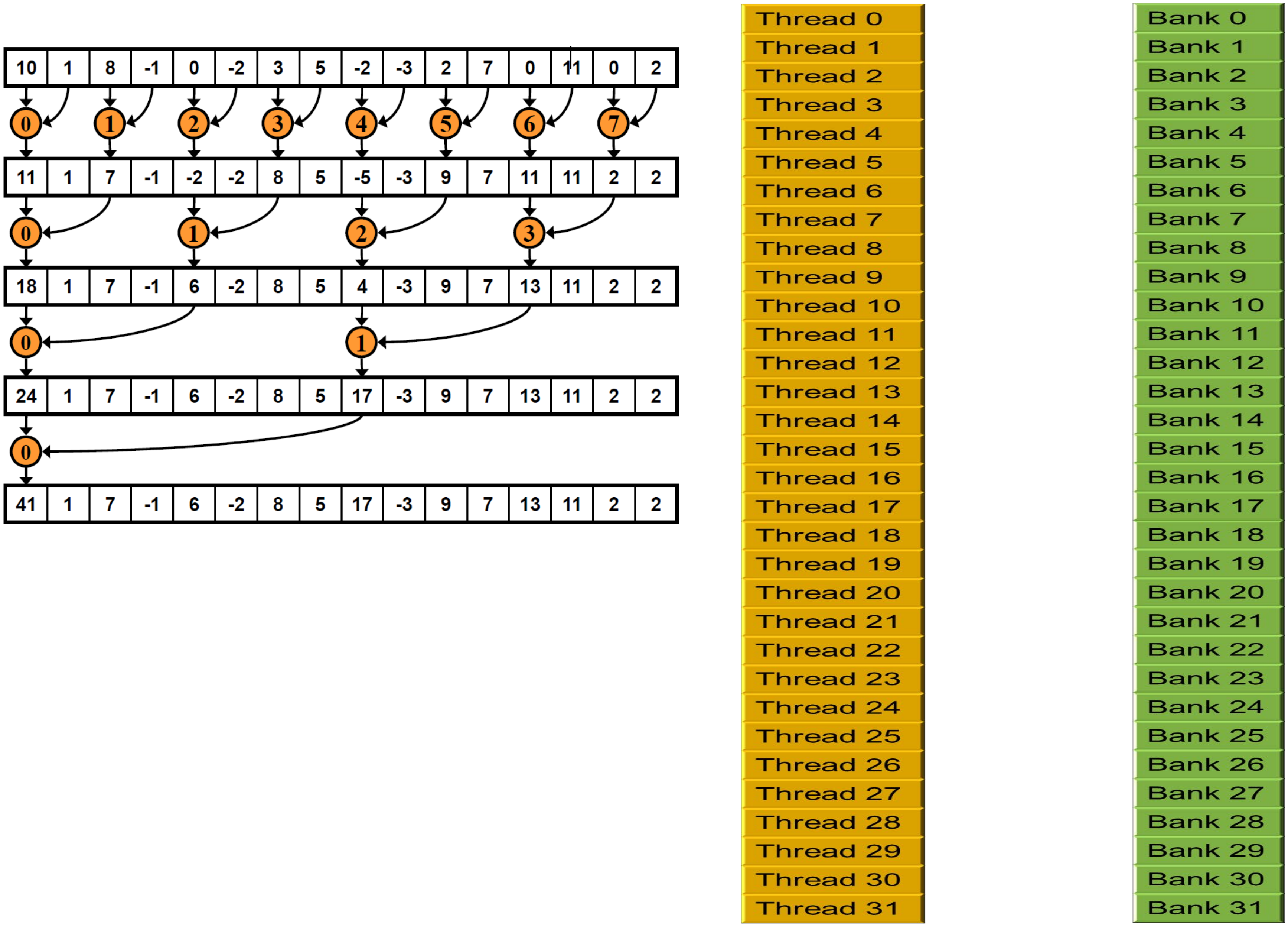
- Reduction – Shared Memory and Bank Conflicts Resolved

Kernel: Shared Memory – Bank Conflicts in Round 0: 2x stride (1024 elements)

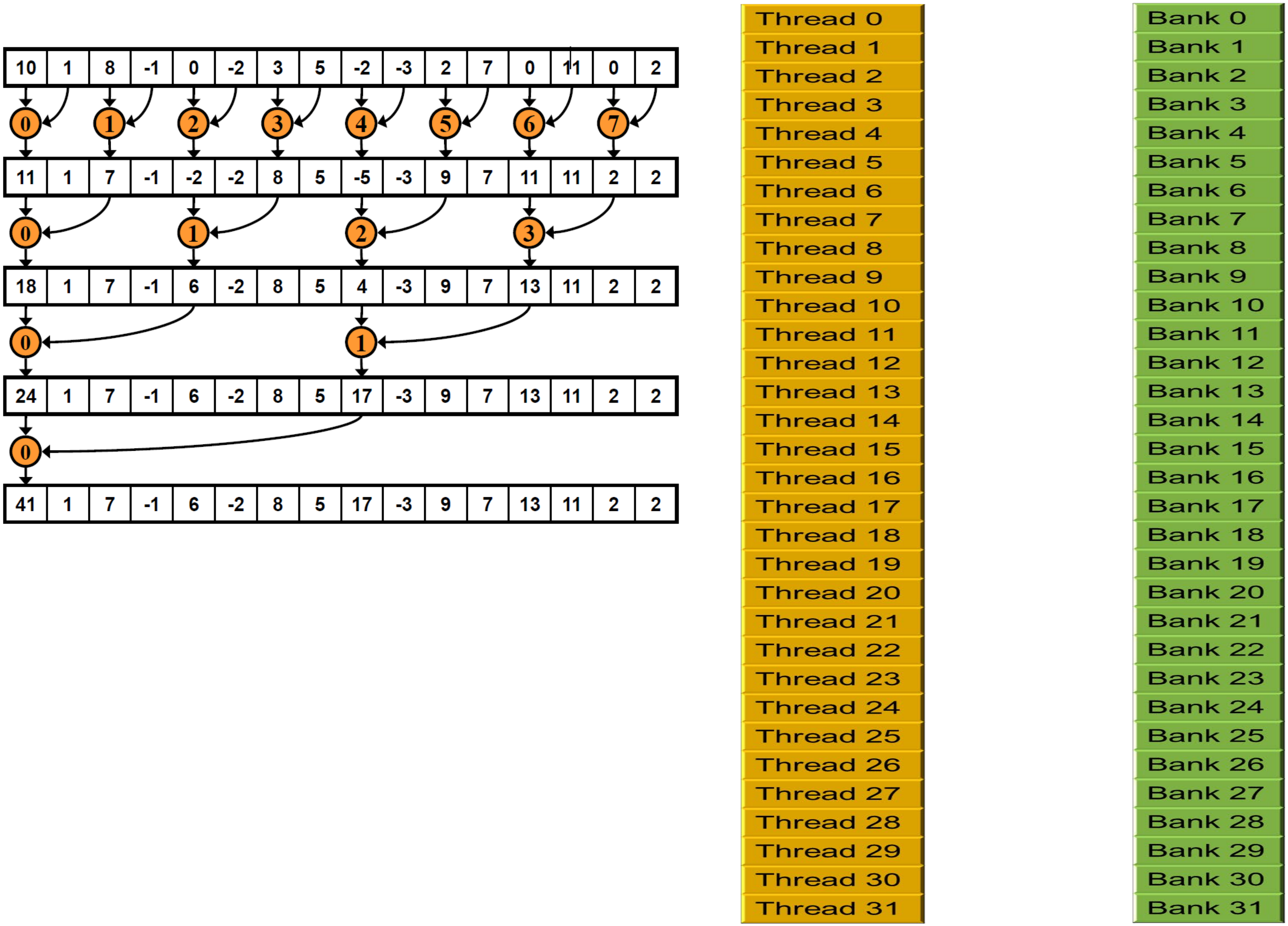




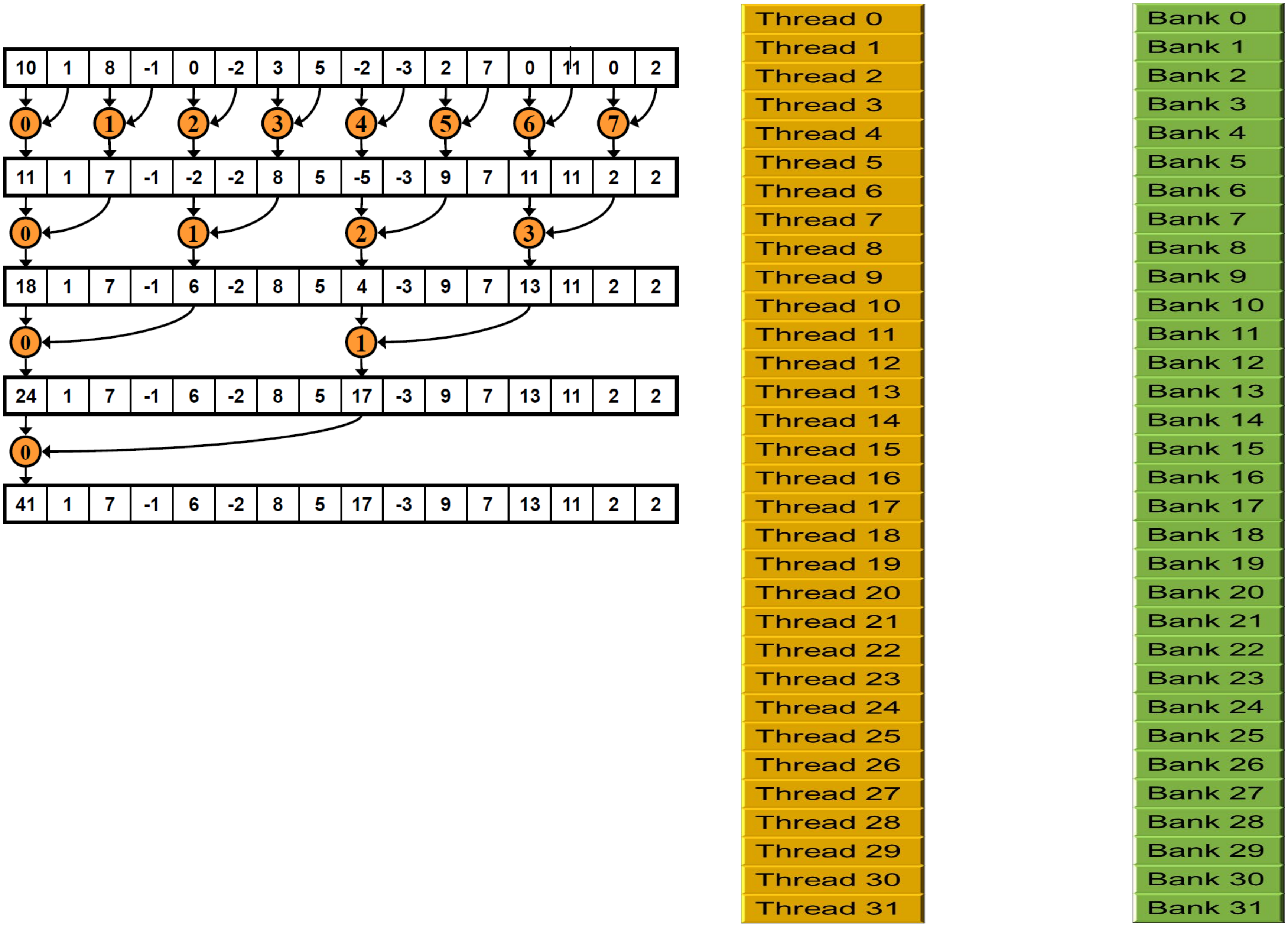
Kernel: Shared Memory – Bank Conflicts in Round 1: 4x stride (512 elements)



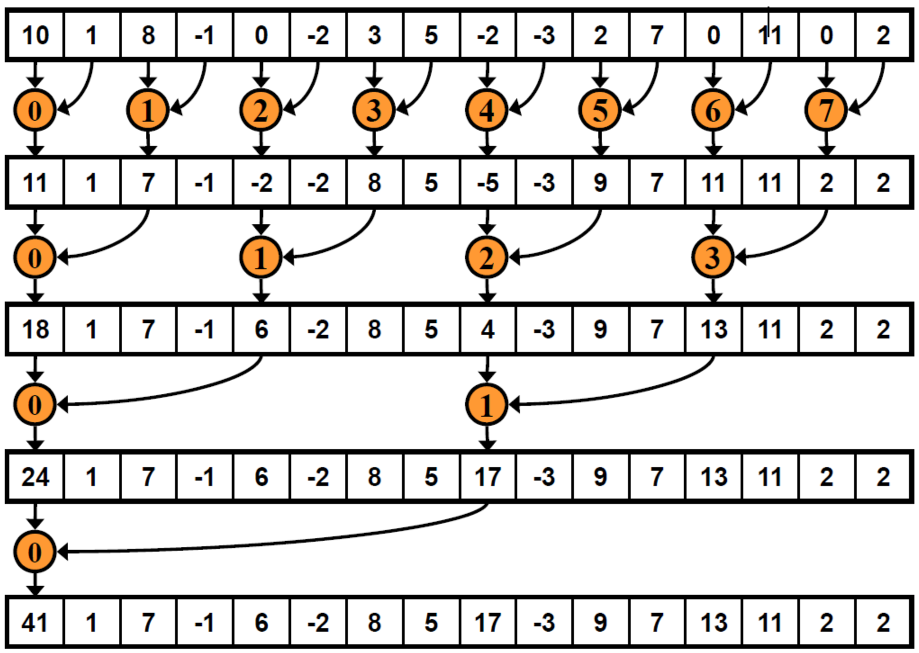
Kernel: Shared Memory – Bank Conflicts in Round 2: 8x stride (256 elements)

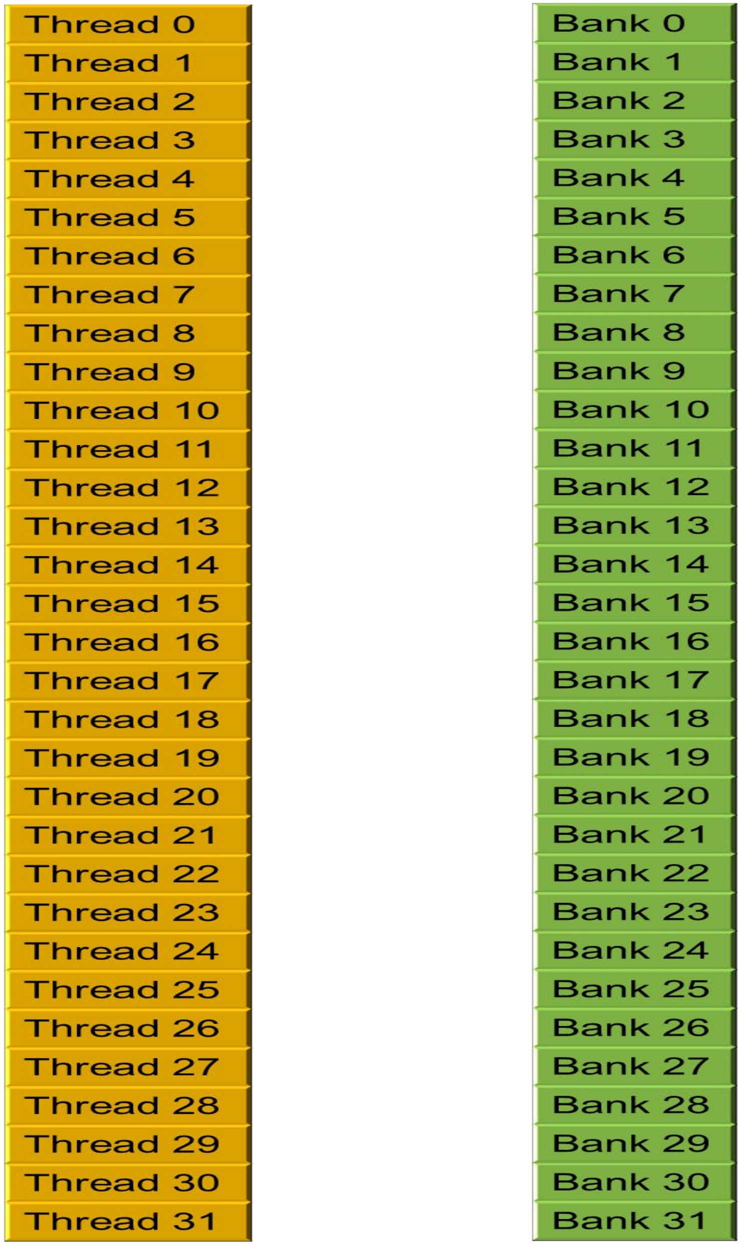


Kernel: Shared Memory – Bank Conflicts in Round 3: 16x (128 elements)



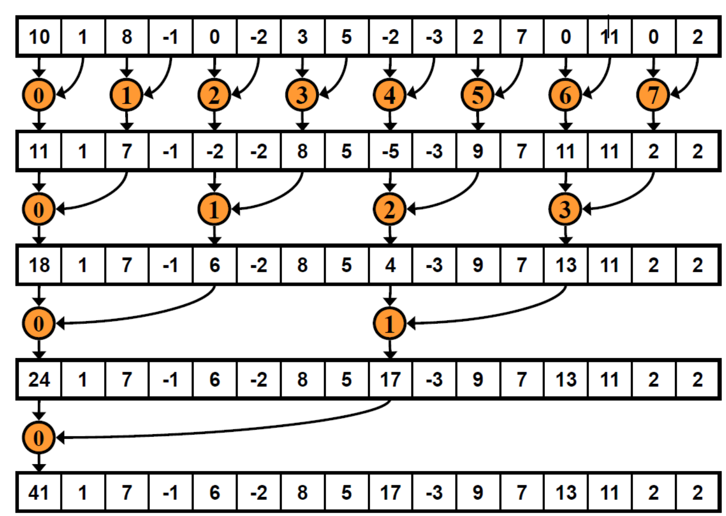
Kernel: Shared Memory – Bank Conflicts in Round 4: 32x (64 elements)





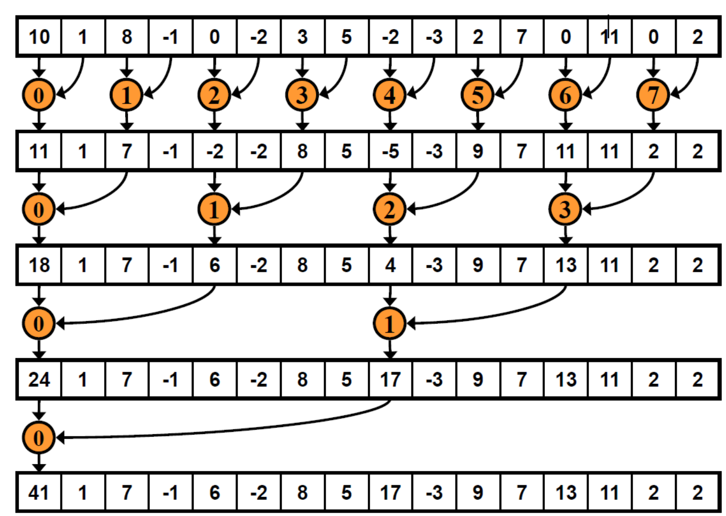
Kernel: Shared Memory – Resolving Bank Conflicts

- Need a new thread to data mapping function that will allow sequence of 32 threads access subsequent addresses
  - But workload per thread is two elements.



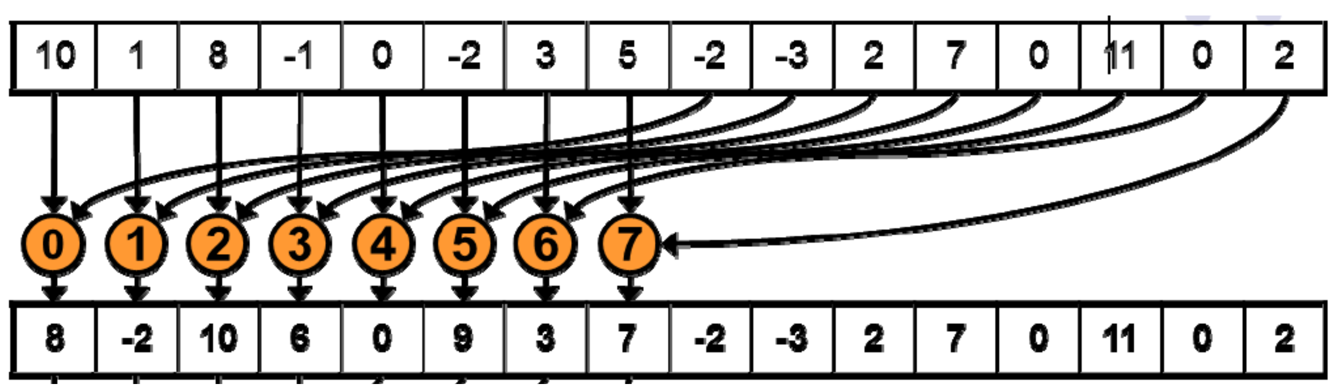
Kernel: Shared Memory – Resolving Bank Conflicts

- Need a new thread to data mapping function that will allow sequence of 32 threads access subsequent addresses
  – But workload per thread is two elements.
- Reduction is associative:
  – we can change index and stride



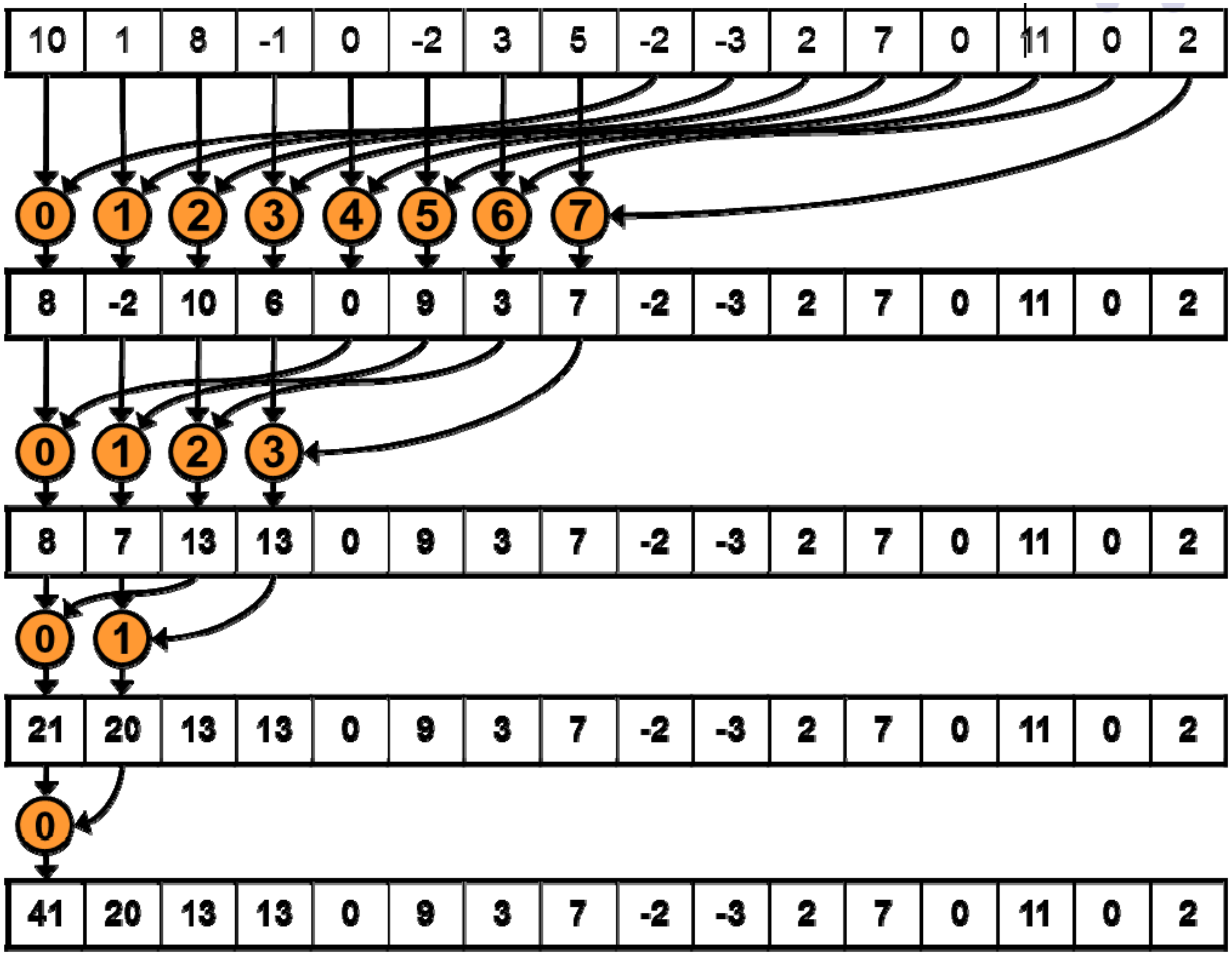
Kernel: Shared Memory – Resolving Bank Conflicts

- Need a new thread to data mapping function that will allow sequence of 32 threads access subsequent addresses
  - But workload per thread is two elements.
- Reduction is associative:
  - we can change index and stride
- What if we assign sequence of 32 elements as first inputs for each thread and feed second input data that is half of the thread block size distance away in round 0?



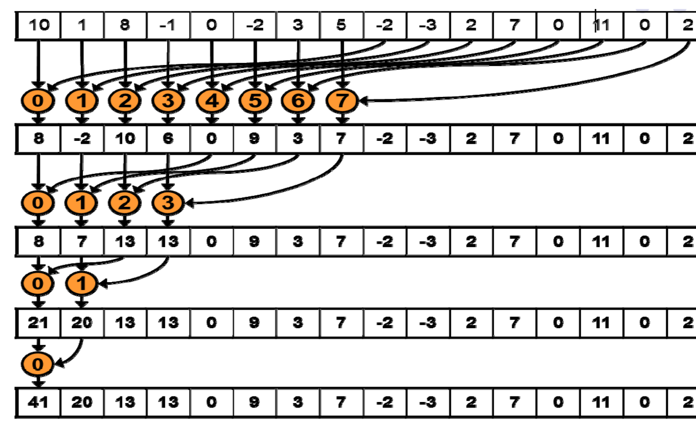
Next: Shared Memory – Reverse Stride Pattern

Reverse the access pattern and reduce the stride by half in each round



Kernel: Shared Memory – Reverse –Pattern

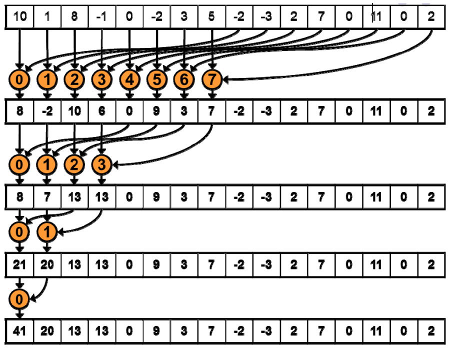
```
__global__ void shared_reduce_reverse(float* d_out, float* d_in){
  extern __shared__ float sdata[];
  // shared_reduce<<<blocks,threads,threads*sizeof(float)>>>
  int myId = threadIdx.x + blockDim.x * blockIdx.x;
  int tid  = threadIdx.x;
  // load shared mem from global mem
  sdata[tid] = d_in[myId];
  // make sure entire block is loaded!
  // do reduction in shared memory
```




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

Kernel: Shared Memory – Reverse Pattern

```
__global__ void shared_reduce_reverse(float* d_out, float* d_in){
  extern __shared__ float sdata[];
  // shared_reduce<<<blocks,threads,threads*sizeof(float)>>>
  int myId = threadIdx.x + blockDim.x * blockIdx.x;
  int tid  = threadIdx.x;
  // load shared mem from global mem
  sdata[tid] = d_in[myId];
  __synchthreads();
  // make sure entire block is loaded!
  // do reduction in shared memory
  for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1) {
        if (tid < s)  {
              sdata[tid] += sdata[tid + s];    }
      __syncthreads();
 }
 if (tid == 0){d_out[blockIdx.x] = sdata[tid]; }
}
```



Reduction - Tesla P100;compute v6.0;

Version	Time (ms)
serial	3.27400
global reduce stride – naïve	0.16450
shared stride reduce	0.15835
shared_reduce_stride_nodiverge	0.09081
shared_reduce_reverse	0.06675

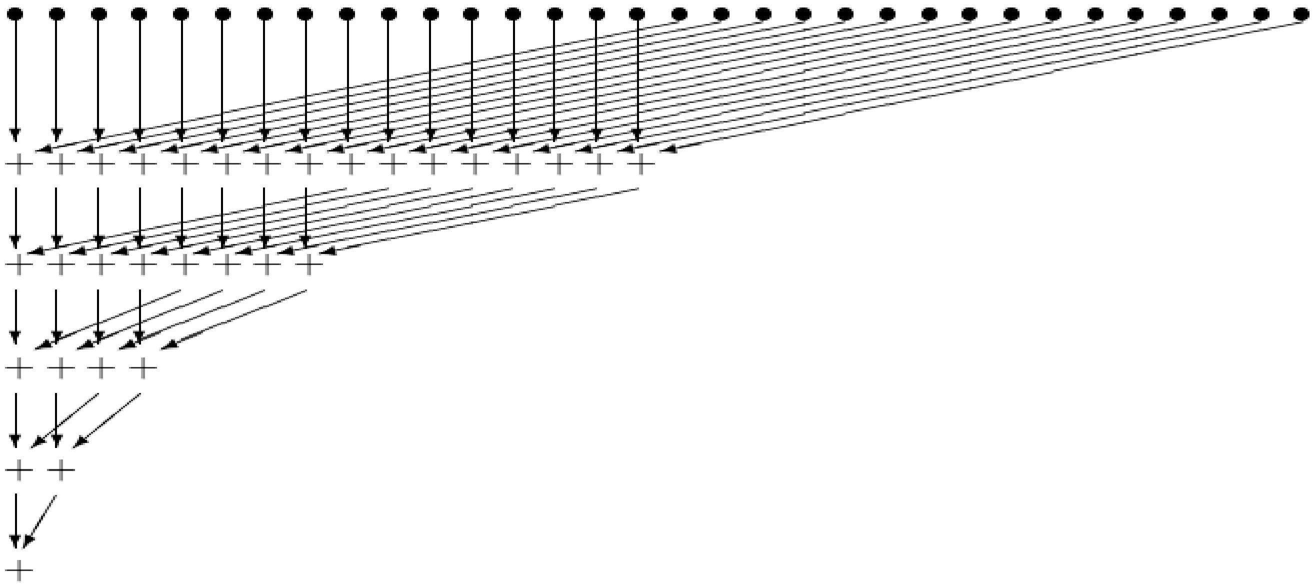
n: 1<<20

49X

1.36X! By resolving bank conflicts

Observations on the Reverse Stride Pattern

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

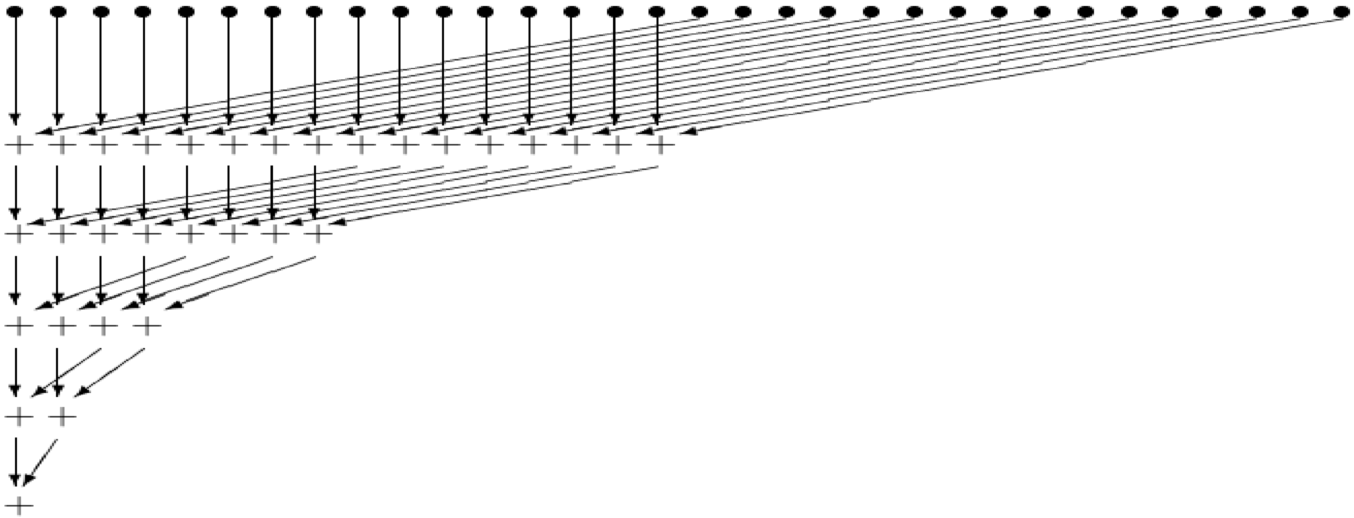


- Note that half of the threads are idle on first loop iteration!
  - This is wasteful…
- **What can we do?**

First Reduction

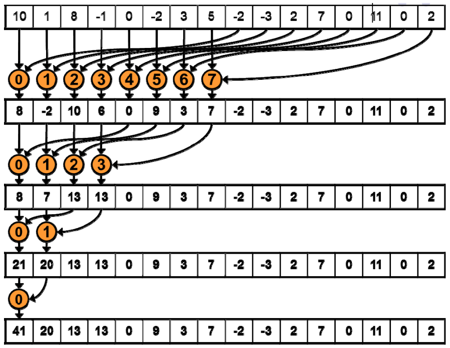
- **Use only half the blocks**
- **Do the first reduction during the load from the global memory**
  - Replace single load with two loads
  - Kernel launch will be:

```
shared_reverse_firstreduction <<<blocks/2, threads,
threads * sizeof(float)>>>
```



Kernel: Shared, Reverse, First reduction

```
__global__ void shared_reverse_firstreduction(float * d_out,
float * d_in){
  extern __shared__ float sdata[];
  // shared_reduce<<<blocks,threads,threads*sizeof(float)>>>
  int myId = threadIdx.x + blockDim.x * blockIdx.x;
  int tid  = threadIdx.x;
  // load shared mem from global mem
  sdata[tid] = d_in[myId];
  __synchthreads();
  // make sure entire block is loaded!
  // do reduction in shared memory
  for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1) {
        if (tid < s)  {
              sdata[tid] += sdata[tid + s];    }
      __synchthreads();
 }
 if (tid == 0){d_out[blockIdx.x] = sdata[tid]; }
}
```



Revise for first reduction

Kernel: Shared, Reverse, First reduction

```
__global__ void shared_reverse_firstreduction (float * d_out,
float * d_in){
  extern __shared__ float sdata[];
  // shared_reduce<<<blocks,threads,threads*sizeof(float)>>>
  int myId = threadIdx.x + blockDim.x * 2 * blockIdx.x;
  int tid  = threadIdx.x;
  // load shared mem from global mem
   sdata[tid] = d_in[myId]+d_in[myId+blockDim.x];
__syncthreads();
  // make sure entire block is loaded!
  // do reduction in shared memory
  for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1) {
        if (tid < s)  {
            sdata[tid] += sdata[tid + s];   }
      __syncthreads();
 }
 if (tid == 0){d_out[blockIdx.x] = sdata[tid]; }
}
```

blockIdx.x is half of the original range!

Reduction - Tesla P100;compute v6.0;

Version	Time (ms)
serial	3.27400
global reduce stride – naïve	0.16450
shared stride reduce	0.15835
shared_reduce_stride_nodiverge	0.09081
shared_reduce_reverse	0.06675
shared_reduce_reverse_first_reduction	0.03405

n: 1<<20

96X

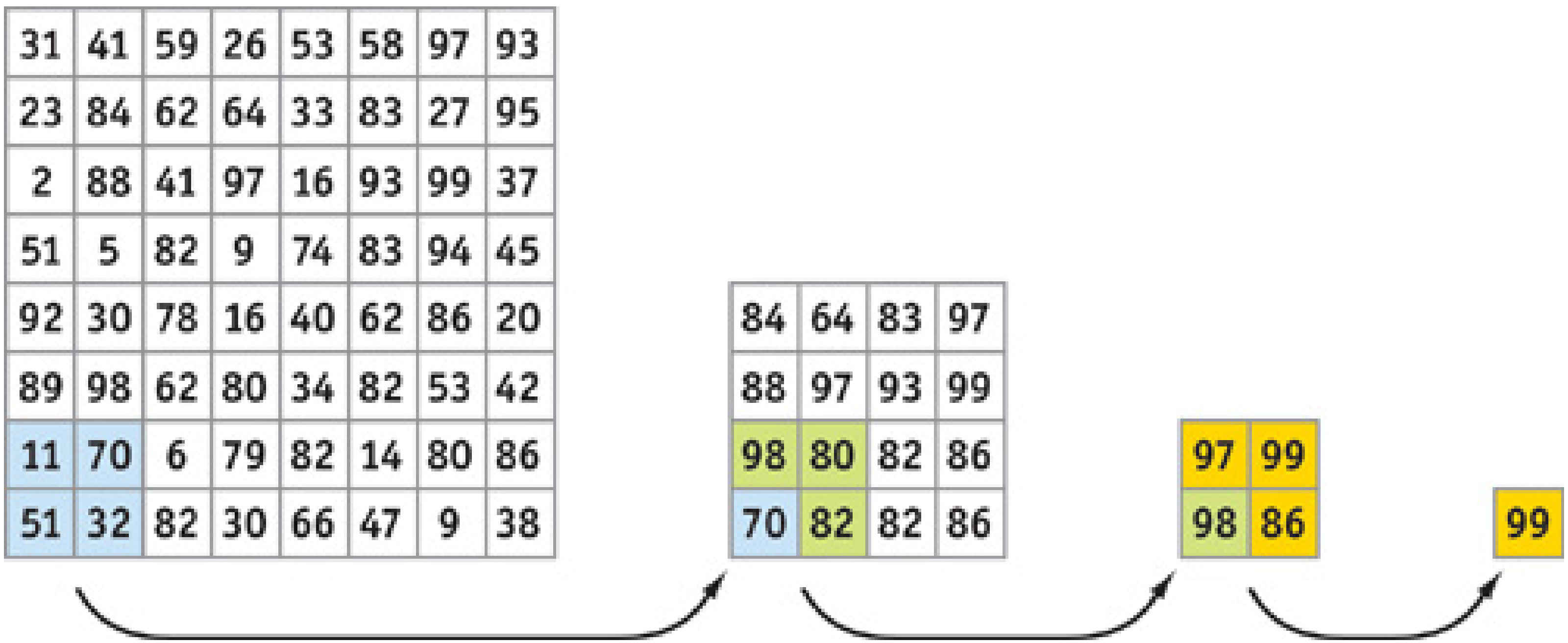
1.96X! With first reduction

Parallel Sum Reduction Summary

- Parallel implementation
  - Recursively halve # of threads, add two values per thread in each step
  - Takes log(n) steps for n elements, requires n/2 threads
- In-place reduction using shared memory
  - The original vector is in device global memory
  - The shared memory is used to hold a partial sum vector
  - Each step brings the partial sum vector closer to the sum
  - The final sum will be in element 0 of the partial sum vector
  - Reduces global memory traffic due to partial sum values
  - Thread block size limits n to be less than or equal to 2,048

2D Max Reduction Example

- Read four elements from four quadrants of the input buffer, such that the output size is halved in both dimensions at each step.



Next

- **Scan**