# Parallel reduction (reduce)

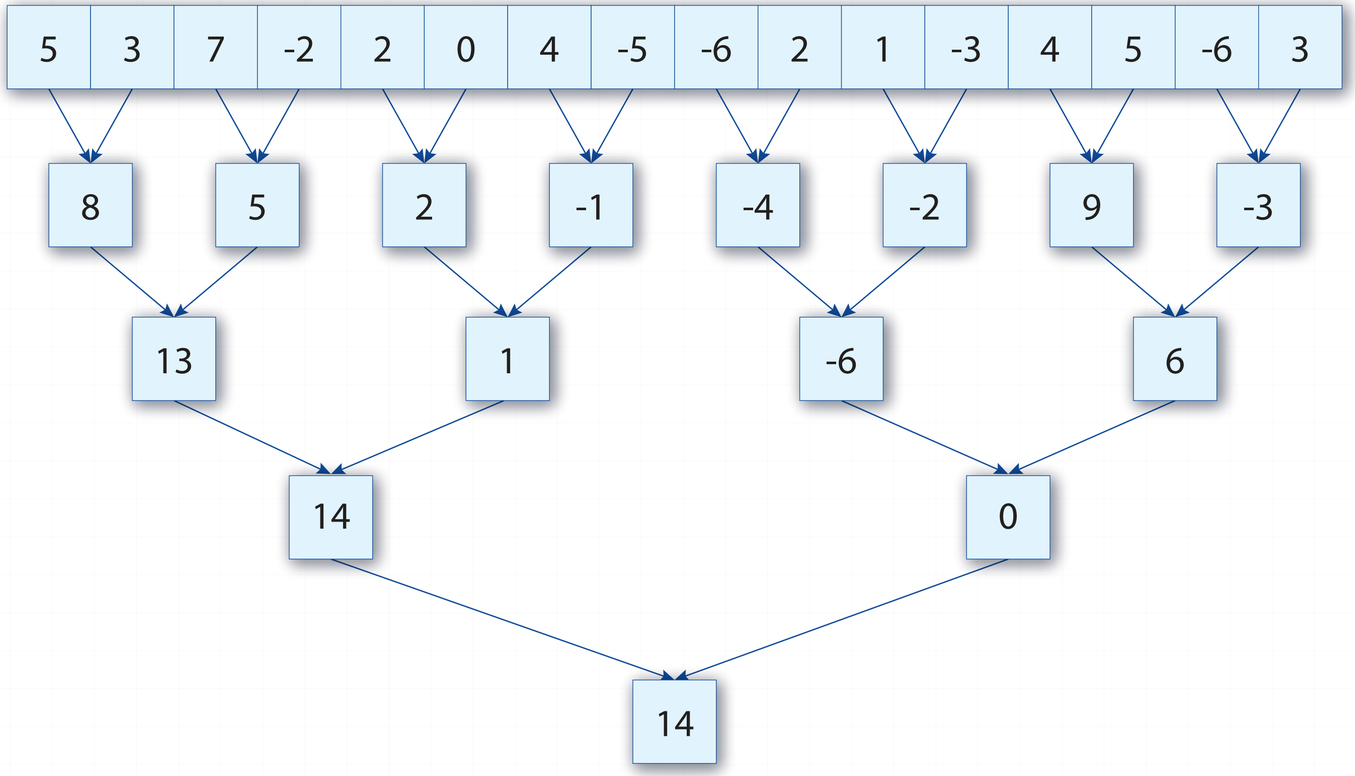
- Input:
  - Array of elements $a_0, a_1, \ldots, a_{n-1}$
  - Binary associative operation ‘+’
- Problem: calculate $A = a_0 + a_1 + \ldots + a_{n-1}$
- Bottleneck: memory accesses
- Instead of “+” there also could be *min*, *max*, etc

```
int sum = 0;
for (int i = 0; i < n; i++)
sum += a [i];
```

# Reduce implementation

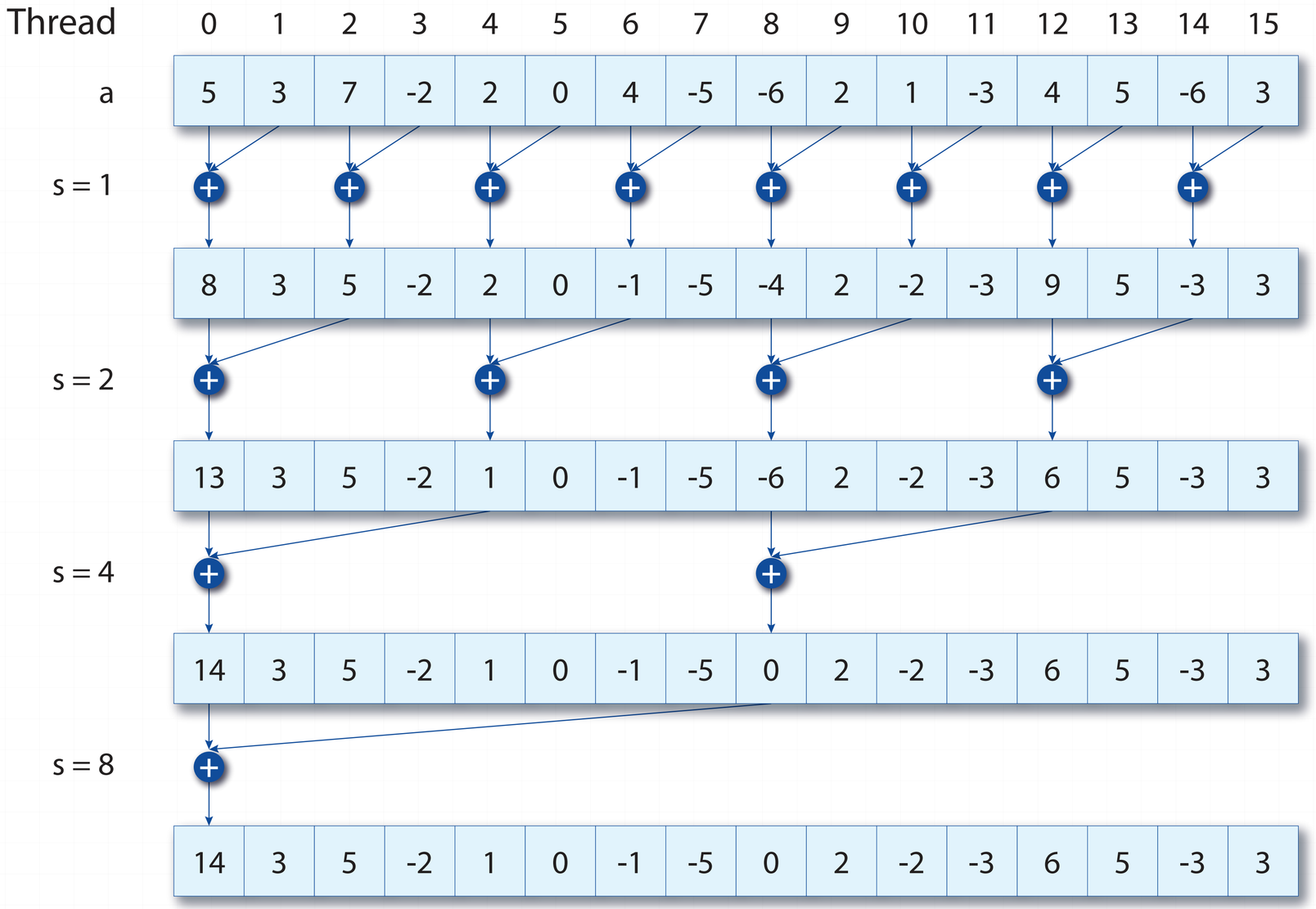
- Each threads block processes its own part of input array
- Each block performs:
  - Copying data into *shared* memory
  - Hierarchical summing in *shared* memory
  - Stores partial result in global memory

# Hierarchical summing




- Hierarchical array summing algorithm utilizes many threads in parallel
- Requires *log(N)* steps

# Reduce, version 1

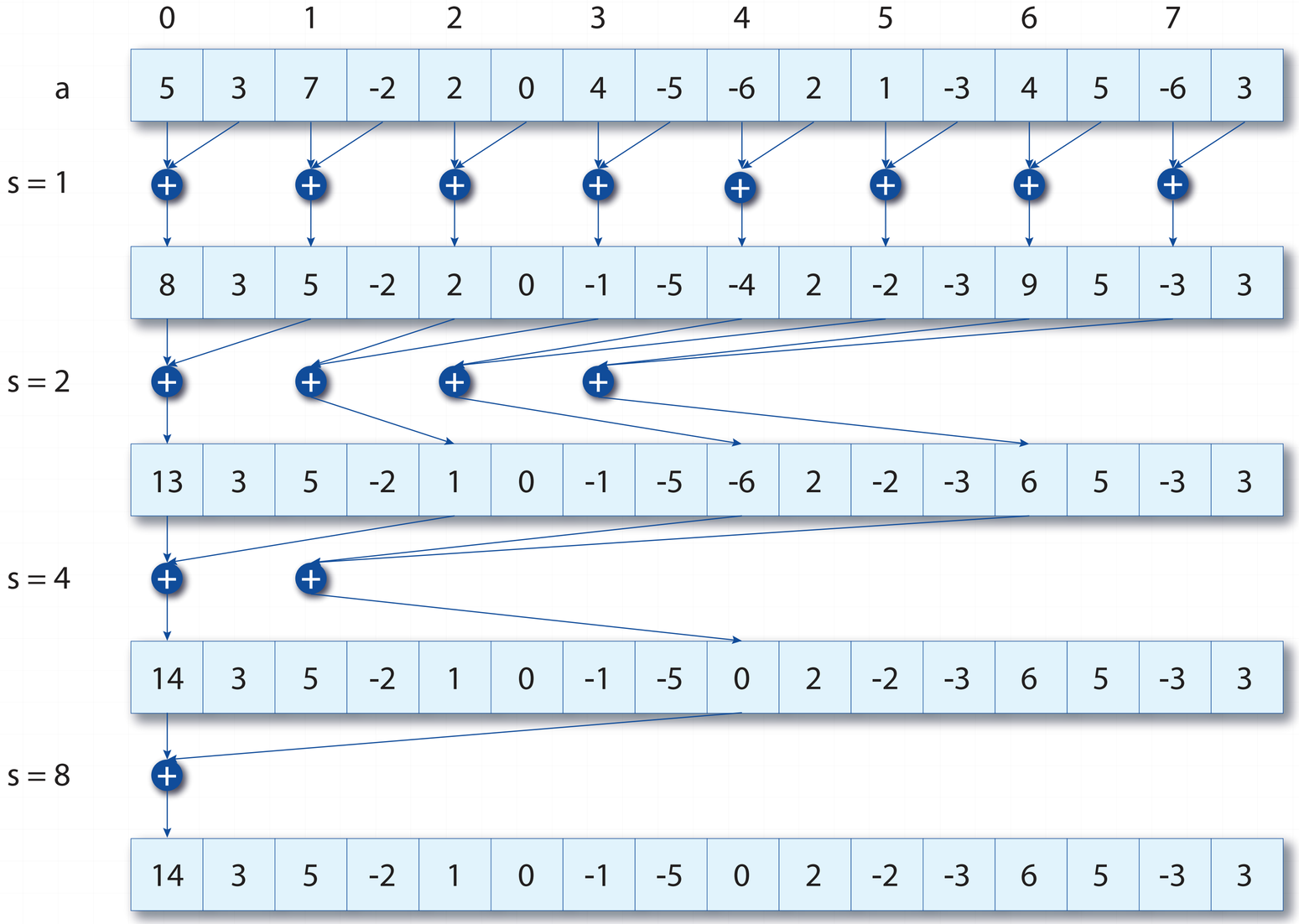


# Reduce, version 1

```
__global__ void reduce1 ( int * inData, int * outData )
{
  __shared__ int data [BLOCK_SIZE];
  int tid = threadIdx.x;
  int i   = blockIdx.x * blockDim.x + threadIdx.x;

  data [tid] = inData [i];     // load into shared memory
  __syncthreads ();
  for ( int s = 1; s < blockDim.x; s *= 2 ) {
    if ( tid % (2*s) == 0 )    // heavy branching !!!
      data [tid] += data [tid + s];
    __syncthreads ();
  }
  if ( tid == 0 )          // write result of block reduction
    outData[blockIdx.x] = data [0];
}
```

# Reduce, version 2



# Reduce, version 2

```
__global__ void reduce2 ( int * inData, int * outData )
{
    __shared__ int data [BLOCK_SIZE];
    int tid = threadIdx.x;
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    data [tid] = inData [i]; // load into shared memory
    __syncthreads ();
    for ( int s = 1; s < blockDim.x; s <<= 1 )
{
    int index = 2 * s * tid; // better replace with >>
    if ( index < blockDim.x )
    data [index] += data [index + s];
    __syncthreads ();
}
if ( tid == 0 ) // write result of block reduction
    outData [blockIdx.x] = data [0];
}
```

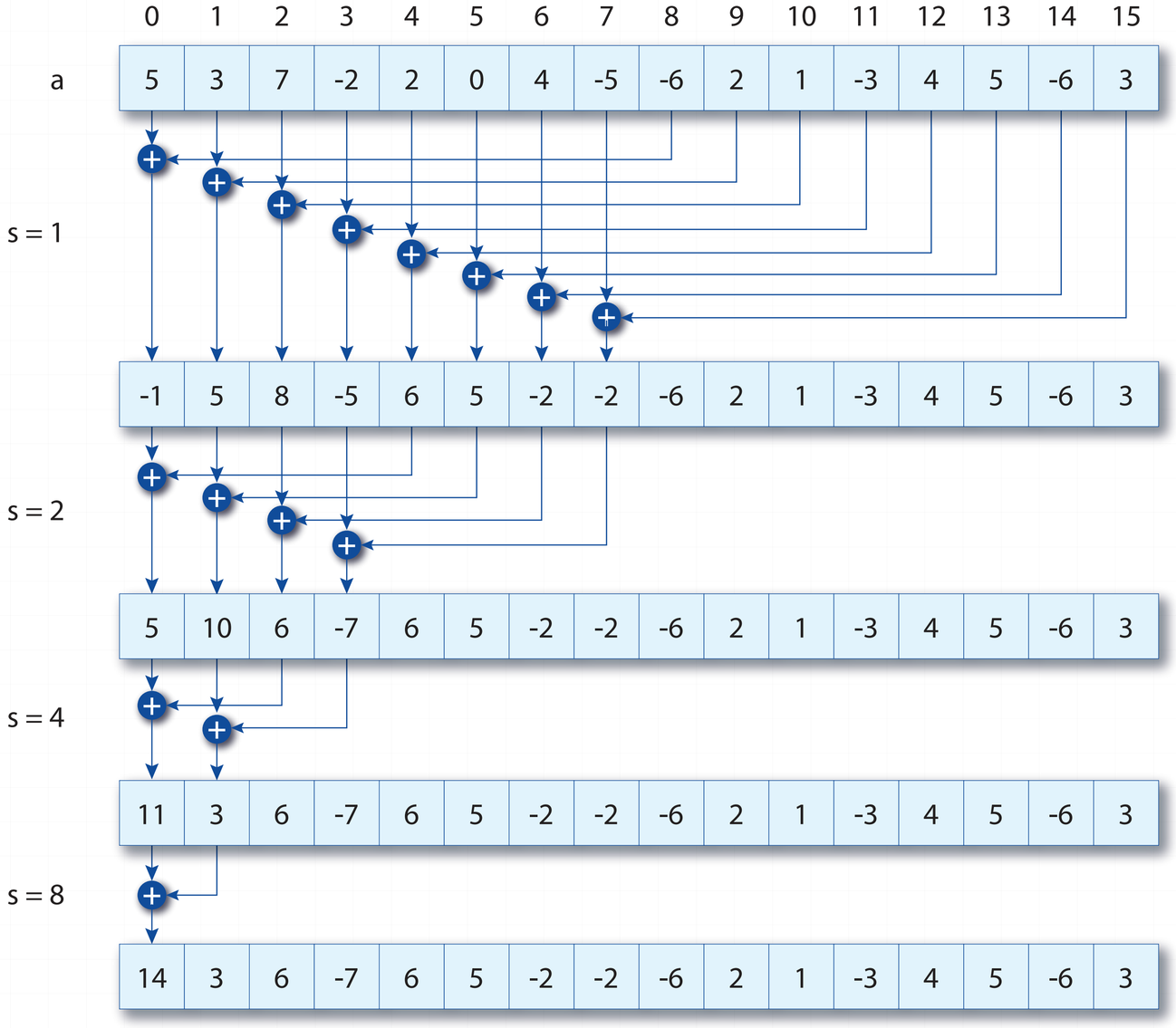
# Reduce, version 2

- Branching is almost eliminated
- This version still has a lot of bank conflicts with increasing order

# Reduce, version 3

- In previous version summing was performed between nearest elements with distance increasing by 2x on every new step
- Let's invert the order of summing:
  - summing starts from the most distant pairs *(dimBlock.x/2)*, and distance will be reduced by 2x on every new step

# Reduce, version 3



# Reduce, version 3

```
__global__ void reduce3 ( int * inData, int * outData )
{
  __shared__ int data [BLOCK_SIZE];
  int tid = threadIdx.x;
  int i   = blockIdx.x * blockDim.x + threadIdx.x;

  data [tid] = inData [i];
  __syncthreads ();
  for ( int s = blockDim.x / 2; s > 0; s >>= 1 )
  {
    if ( tid < s )
      data [tid] += data [tid + s];
    __syncthreads ();
  }
  if ( tid == 0 )
    outData [blockIdx.x] = data [0];
}
```

# Reduce, version 3

- No bank conflicts
- No branching
- But on first iteration half of threads are not used
  - Let's combine first step with data loading

# Reduce, version 4

```
__global__ void reduce4 ( int * inData, int * outData )
{
  __shared__ int data [BLOCK_SIZE];
  int tid = threadIdx.x;
  int i   = 2 * blockIdx.x * blockDim.x + threadIdx.x;

  data [tid] = inData [i] + inData [i+blockDim.x]; // sum
  __syncthreads ();
  for ( int s = blockDim.x / 2; s > 0; s >>= 1 )
  {
    if ( tid < s )
      data [tid] += data [tid + s];
    __syncthreads ();
  }
  if ( tid == 0 )
    outData [blockIdx.x] = data [0];
}
```

# Reduce, version 5

- In case of *s<=32* each block contains only single warp, thus:
  - No need for synchronization
  - No need for *tid<s* check
- Let's inline loop for *s<=32*

# Reduce, version 5

```
for ( int s = blockDim.x / 2; s > 32; s >>= 1 )
{
  if ( tid < s )
    data [tid] += data [tid + s];
  __syncthreads ();
}

if ( tid < 32 ) // unroll last iterations
{                 // compile can be “oversmart here”
  volatile float * smem = data;

  smem [tid] += smem [tid + 32];
  smem [tid] += smem [tid + 16];
  smem [tid] += smem [tid + 8];
  smem [tid] += smem [tid + 4];
  smem [tid] += smem [tid + 2];
  smem [tid] += smem [tid + 1];
}
```

# Reduce performance (Tesla C2070, 16 mln elements)

| Algorithm version | Execution time (milliseconds) |
|---|---|
| reduction1 | 5.28 |
| reduction2 | 2.52 |
| reduction3 | 1.88 |
| reduction4 | 0.99 |
| reduction5 | 0.65 |
| reduction OpenMP (dual Xeon E5620) | 2.34 |