Distributed and Parallel Computing

Lecture 04

Alan P. Sexton

University of Birmingham

Spring 2018

Prefix Sum, or Scan

Prefix sum, also known as *Scan*, is an operation to that takes a binary associative operator (e.g. addition, multiplication, maximum, etc.) and applies the operator to calculate a cumulative output vector from an input vector.

A sequential version where the operator used is addition might be implemented as follows:

```
void sequential_scan(float *x, float *y, int len)
{
    y[0] = x[0];
    for (int i = 1; i < len ; i++)
        y[i] = y[i-1] + x[i];
}
```

When applied to:

$$\begin{bmatrix}1, & 2, & 3, & 2, & 3, & 1, & 4, & 5\end{bmatrix}$$

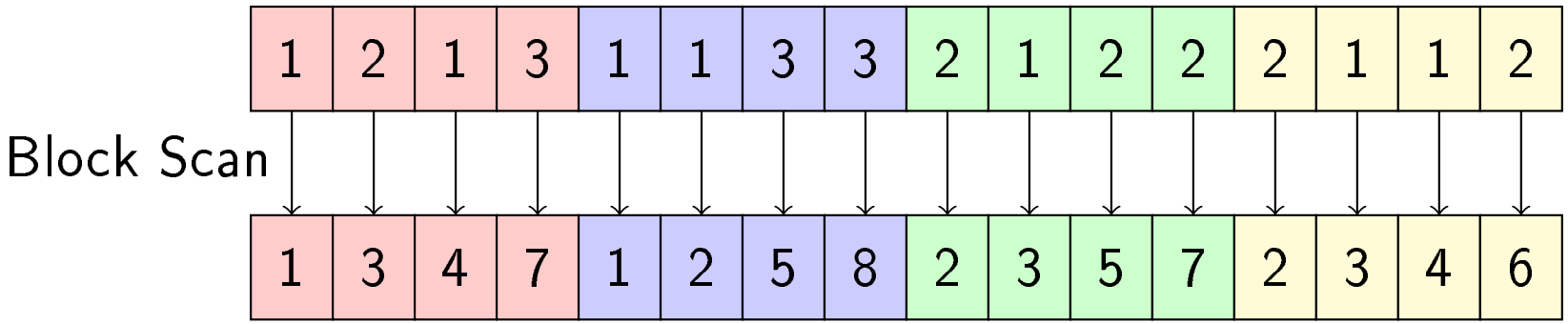
the result would be:

$$\begin{bmatrix}1, & 3, & 6, & 8, & 11, & 12, & 16, & 21\end{bmatrix}$$

Block Prefix Sum, or Block Scan

We will start with algorithms that compute the scan correctly **within each block** but which does not propagate the scan across block boundaries. We shall return later to the problem of how to complete the block scan to a full scan.

An example execution of block scan where the block size is 4 and the length of the vector is 16 is as follows:



Sequential implementation of Block Scan

A sequential version of a block scan where the operator used is addition might be implemented as follows:

```
#define BLOCK_SIZE 1024

void sequential_block_scan(float *x, float *y, int len)
{
    int num_blocks = 1 + (len-1)/BLOCK_SIZE;
    for (blk = 0 ; blk < num_blocks ; blk ++)
    {
        int blk_start = blk * BLOCK_SIZE ;
        int blk_end = blk_start + BLOCK_SIZE ;
        if (blk_end > len)
            blk_end = len;
        y[blk_start] = x[blk_start];
        for (int i = blk_start + 1; i < blk_end; i++)
            y[i] = y[i-1] + x[i];
    }
}
```

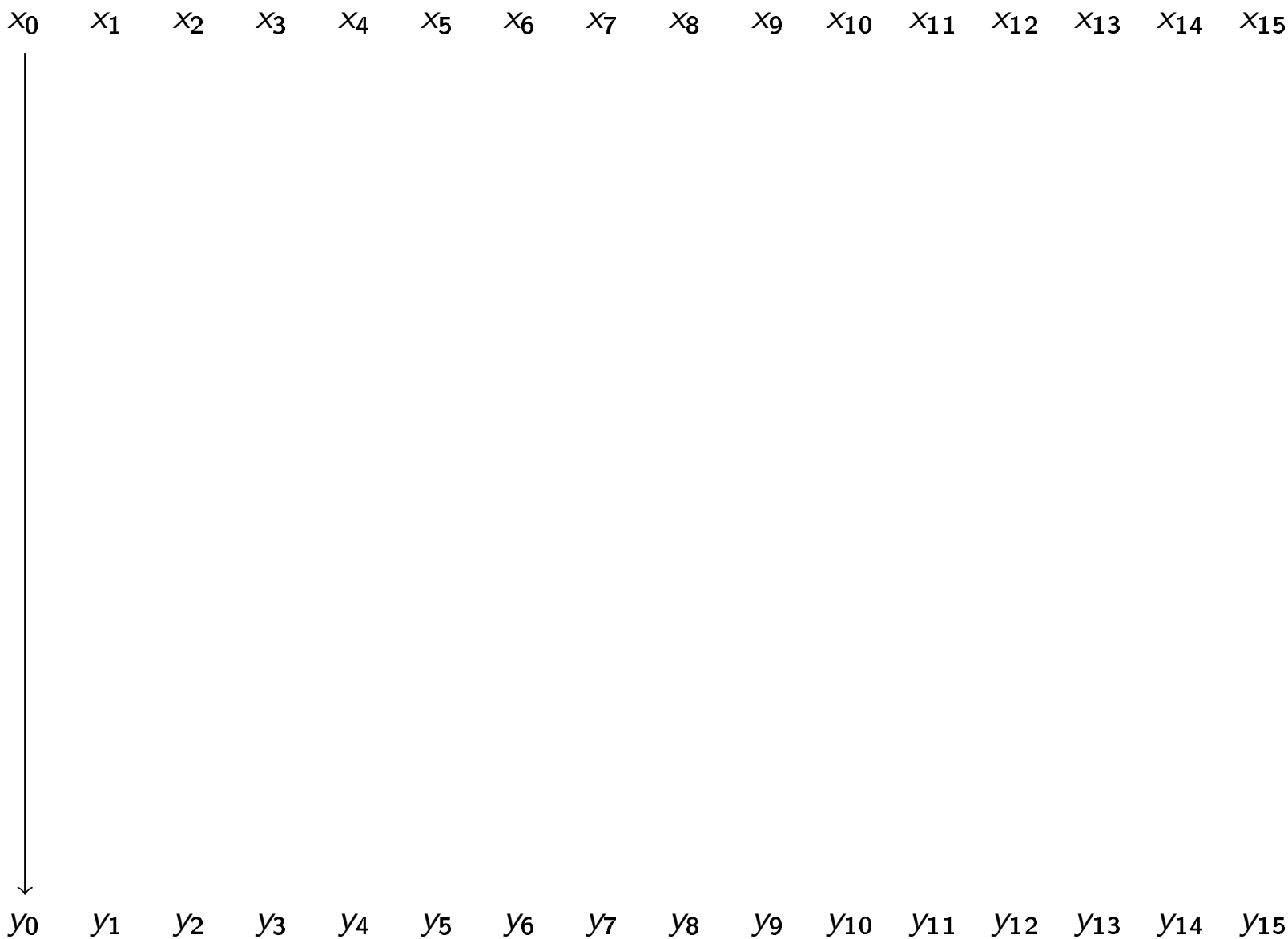
Cost of Sequential Scan

The cost of sequential scan is easy to calculate:

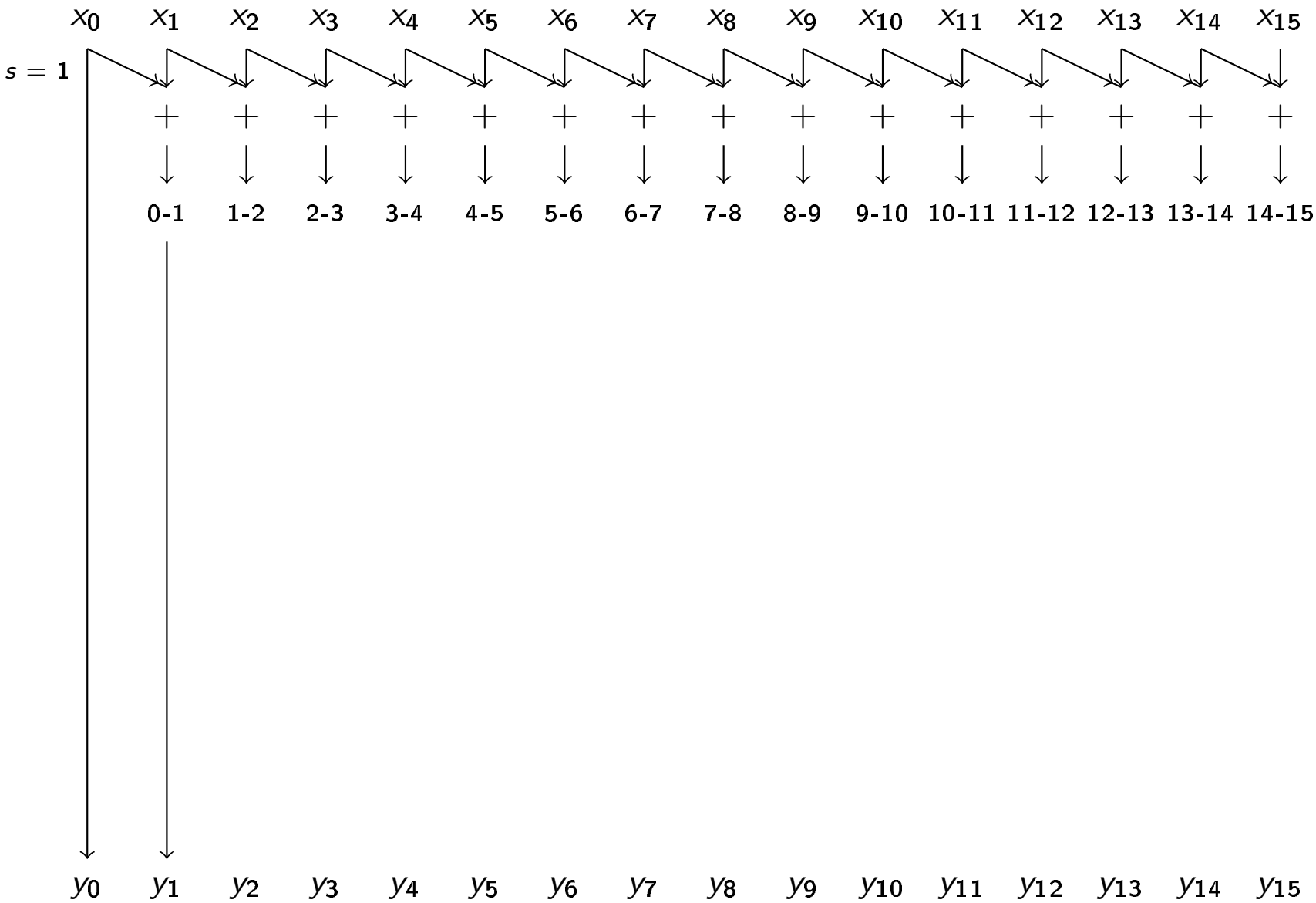
- for N elements, there are $N-1$ floating point additions

$$N = 1024 \Rightarrow \mathrm{Cost}_{\mathrm{SS}} = 1024 - 1 = 1023$$

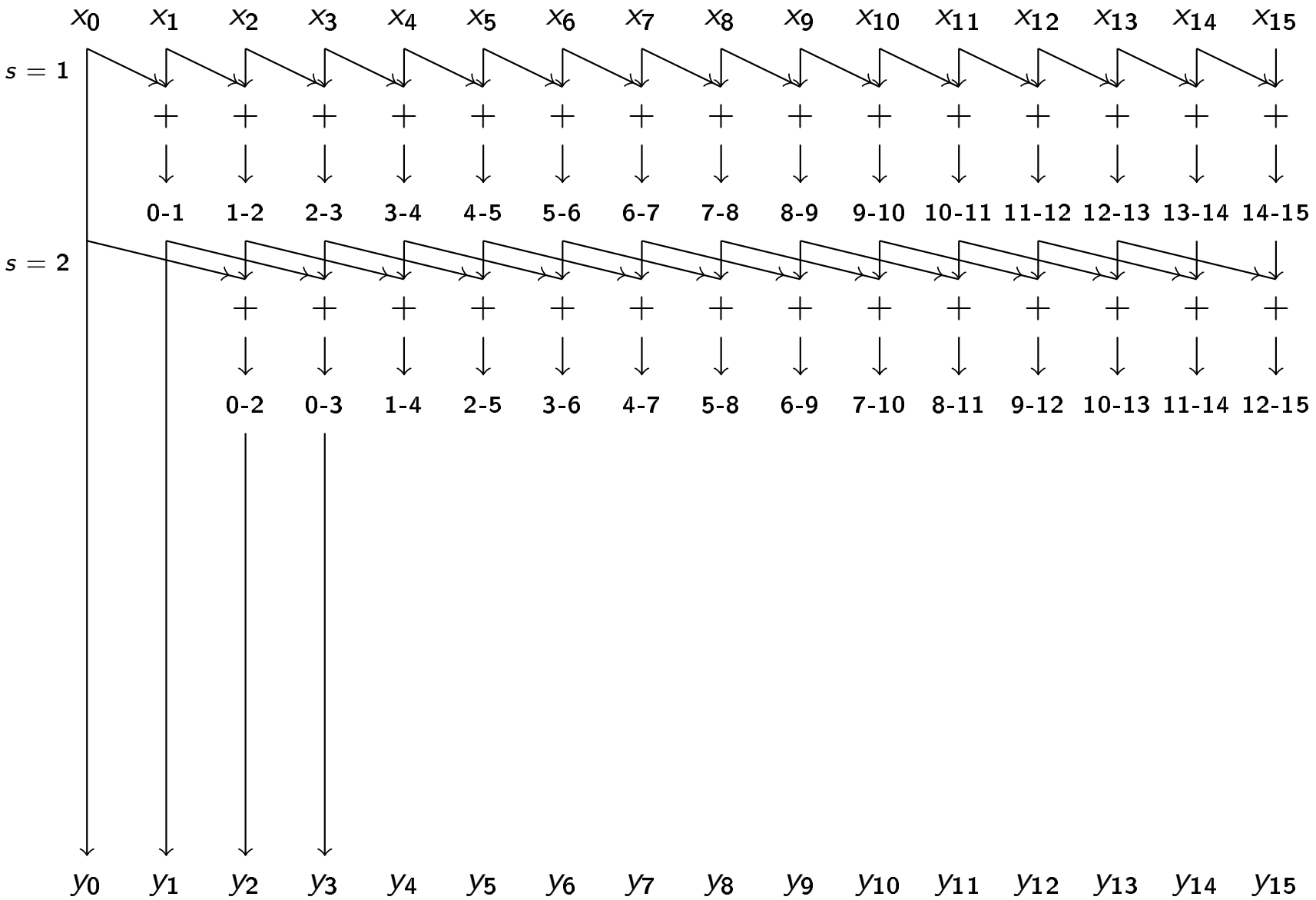
First Attempt at Scan: Hillis Steele Horn (Inefficient)



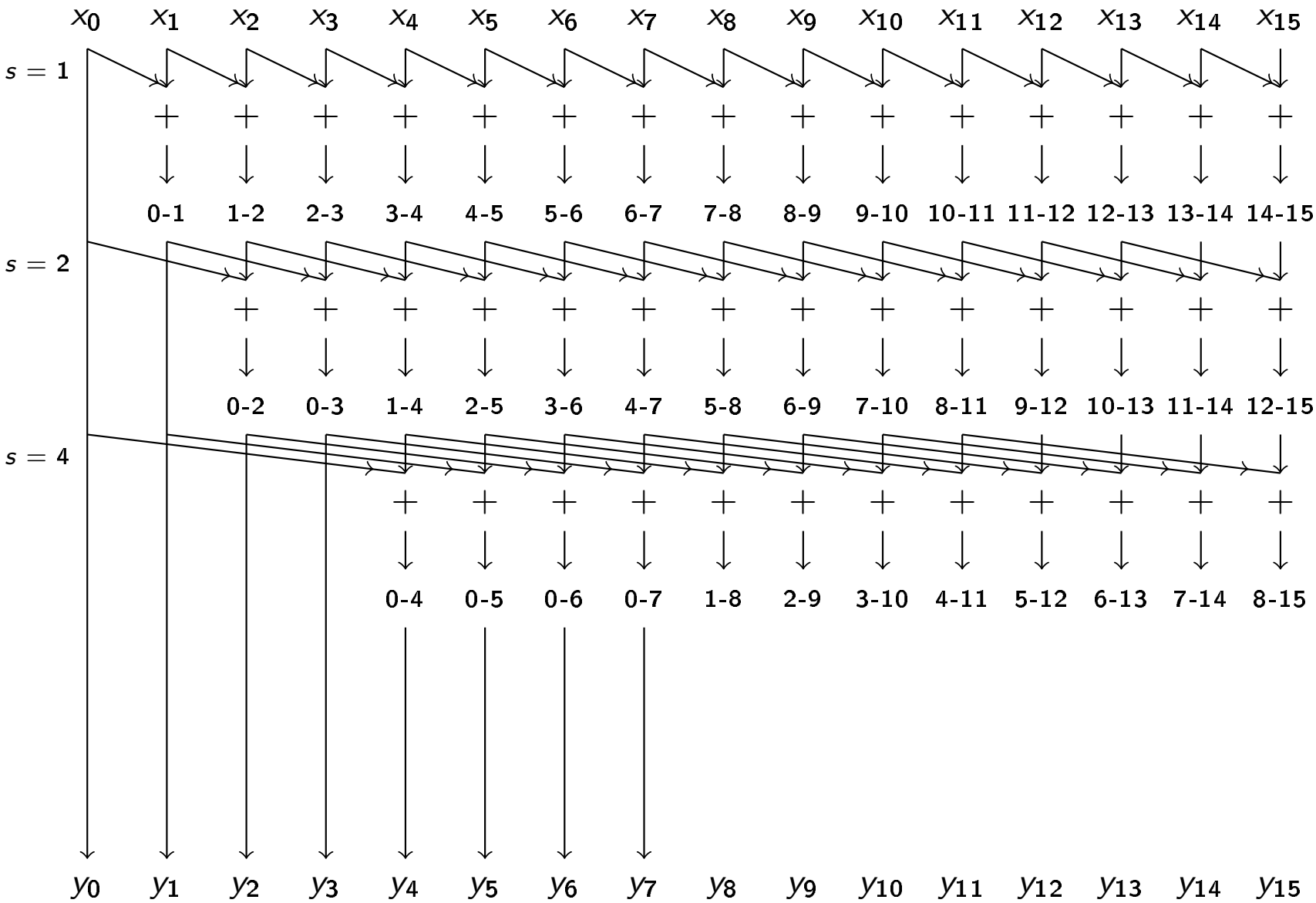
First Attempt at Scan: Hillis Steele Horn (Inefficient)



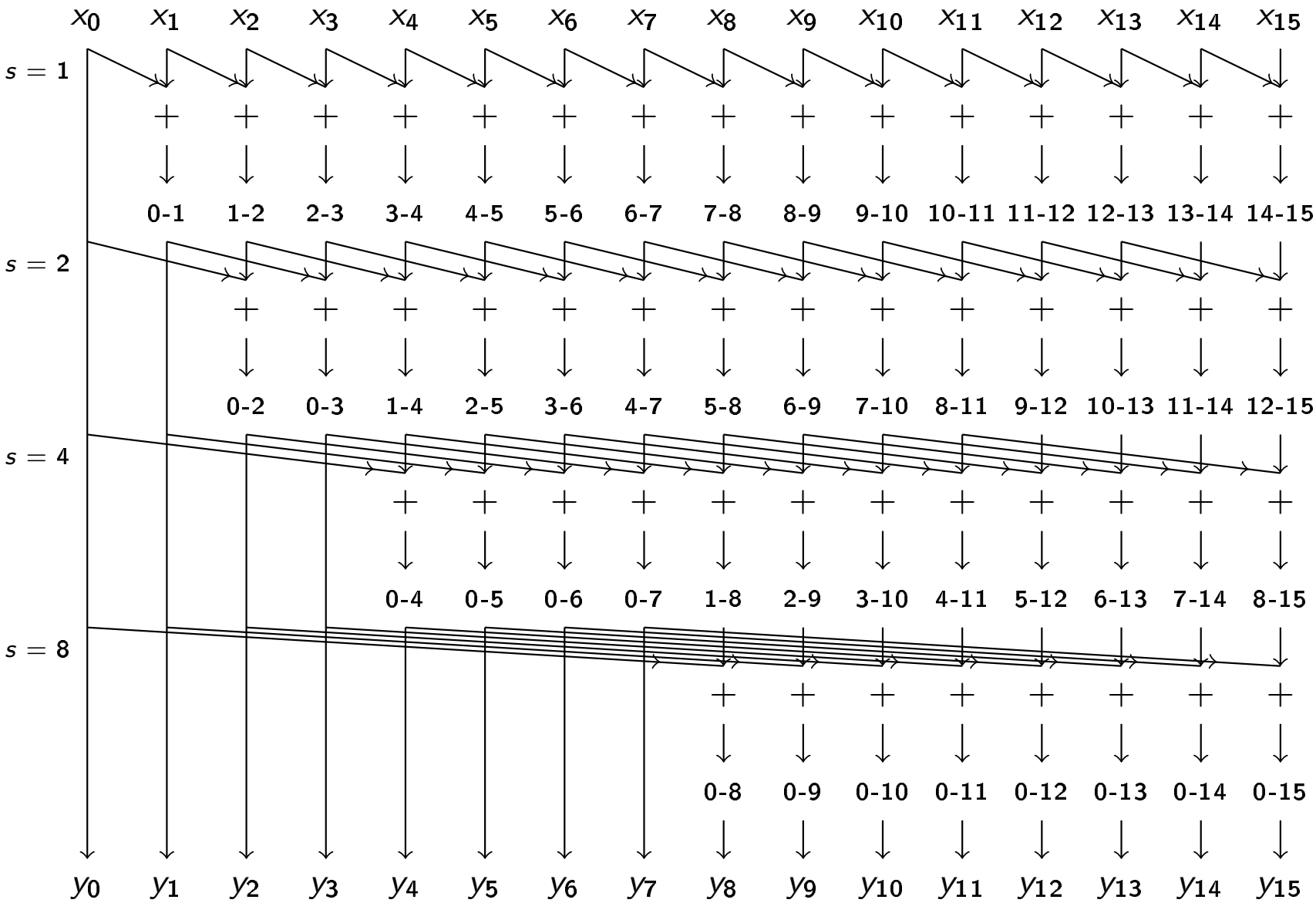
First Attempt at Scan: Hillis Steele Horn (Inefficient)



First Attempt at Scan: Hillis Steele Horn (Inefficient)



First Attempt at Scan: Hillis Steele Horn (Inefficient)



Code for HSH Scan (len $\leq$ block size) - with errors

```
#define BLOCK_SIZE 1024 // the actual configured block size

__global__ void hsh_scan(float *X, float *Y, len)
{
    __shared__ float XY[BLOCK_SIZE];

    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < len)
        XY[threadIdx.x] = X[i];

    for(uint stride = 1; stride <= threadIdx.x; stride *= 2)
    {
        __syncthreads();
        XY[threadIdx.x] += XY[threadIdx.x-stride];
    }
    if (i < len)
        Y[i] = XY[threadIdx.x];
}
```

Errors in Previous Code for HSH Scan

The previous code had 2 errors. Consider the line:

```
XY[threadIdx.x] += XY[threadIdx.x-stride];
```

- The `__syncthreads` ensures that all threads in one block finish one iteration before any starts the next but . . .
- . . . there is no guaranteed order in which the different threads execute this statement within the same iteration:
- If two threads are from different warps, they can execute in arbitrary sequential orders relative to each other
- Thus we could have a thread write to an element of `XY` either **before** or **after** a different thread reads from it.
- This is called a *read/write race*
- To fix it we need to *double buffer*: double the size of `XY` and, in each iteration, read and write from different halves.

Errors in Previous Code for HSH Scan

The previous code had 2 errors. Consider the line:

```
XY[threadIdx.x] += XY[threadIdx.x-stride];
```

- The `__syncthreads` ensures that all threads in one block finish one iteration before any starts the next but ...
- ...there is no guaranteed order in which the different threads execute this statement within the same iteration:
- If two threads are from different warps, they can execute in arbitrary sequential orders relative to each other
- Thus we could have a thread write to an element of `XY` either **before** or **after** a different thread reads from it.
- This is called a *read/write race*
- To fix it we need to *double buffer*: double the size of `XY` and, in each iteration, read and write from different halves.

Worse: since different threads exit the loop at different times, the barrier synchronise may deadlock!

Corrected Code for HSH Scan

```
#define BLOCK_SIZE 1024 // the actual configured block size
__global__ void hsh_scan(float *X, float *Y, len)
{
    __shared__ float XY[BLOCK_SIZE*2]; // 2 buffers
    int rBuf = 0, wBuf = BLOCK_SIZE ;
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < len)
        XY[wBuf + threadIdx.x] = X[i];

    // note: now All threads execute in ALL iterations
    for(uint s=1; s < BLOCK_SIZE; s *= 2)
    {
        __syncthreads();
        wBuf = BLOCK_SIZE - wBuf; rBuf = BLOCK_SIZE - rBuf;
        if (threadIdx.x >= s)
            XY[wBuf+threadIdx.x] =
              XY[rBuf+threadIdx.x-s] + XY[rBuf+threadIdx.x];
        else // if not adding, thread should copy
            XY[wBuf+threadIdx.x] = XY[rBuf+threadIdx.x];
    }
    if (i < len)
        Y[i] = XY[wBuf + threadIdx.x];
}
```

Cost of HSH Scan

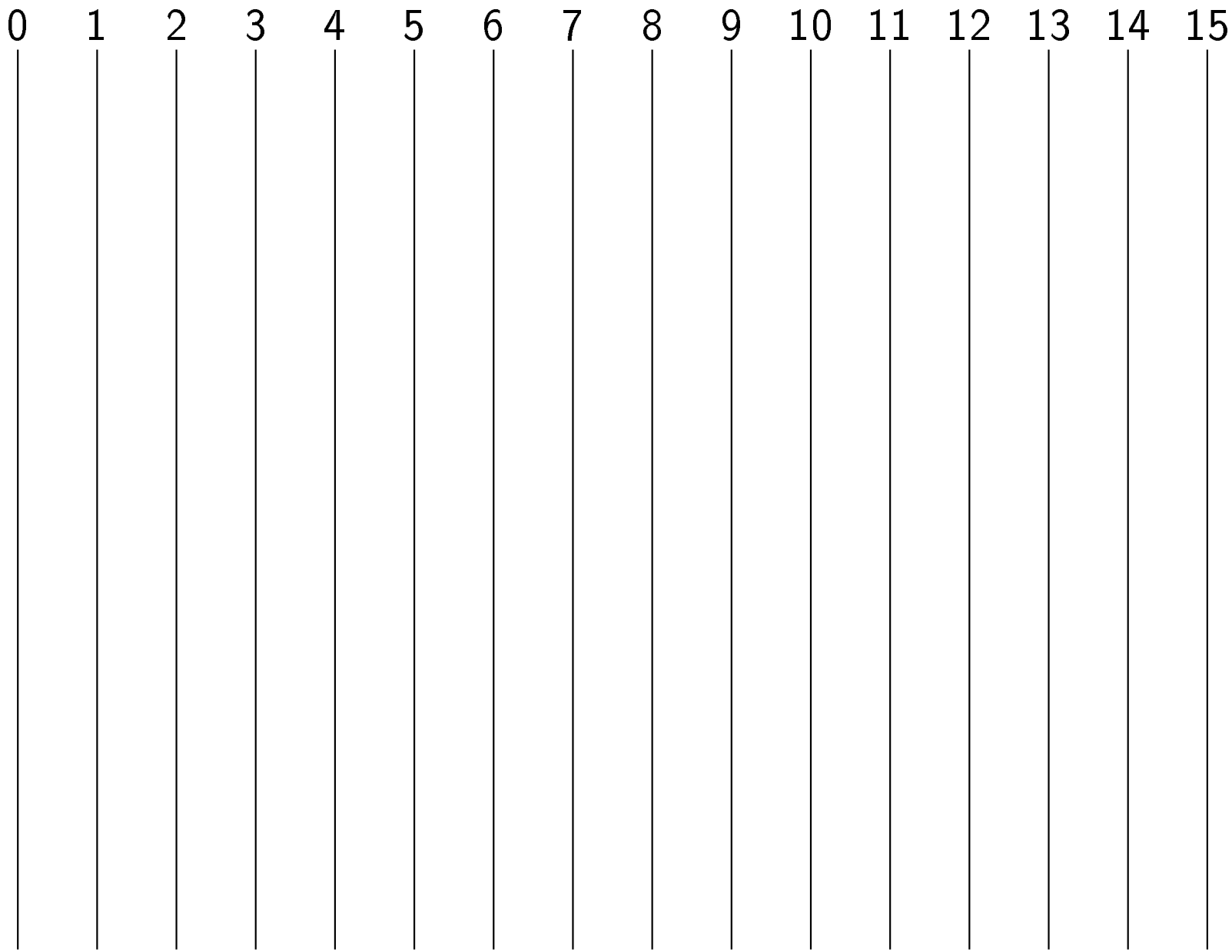
Given N elements:

- There will be $\log_2(N)$ iterations
- In each iteration, all except the *stride* number of threads is doing an addition
- In the first iteration, *stride* is 1
- In the second iteration, *stride* is 2
- In the third iteration, *stride* is 4
- ...

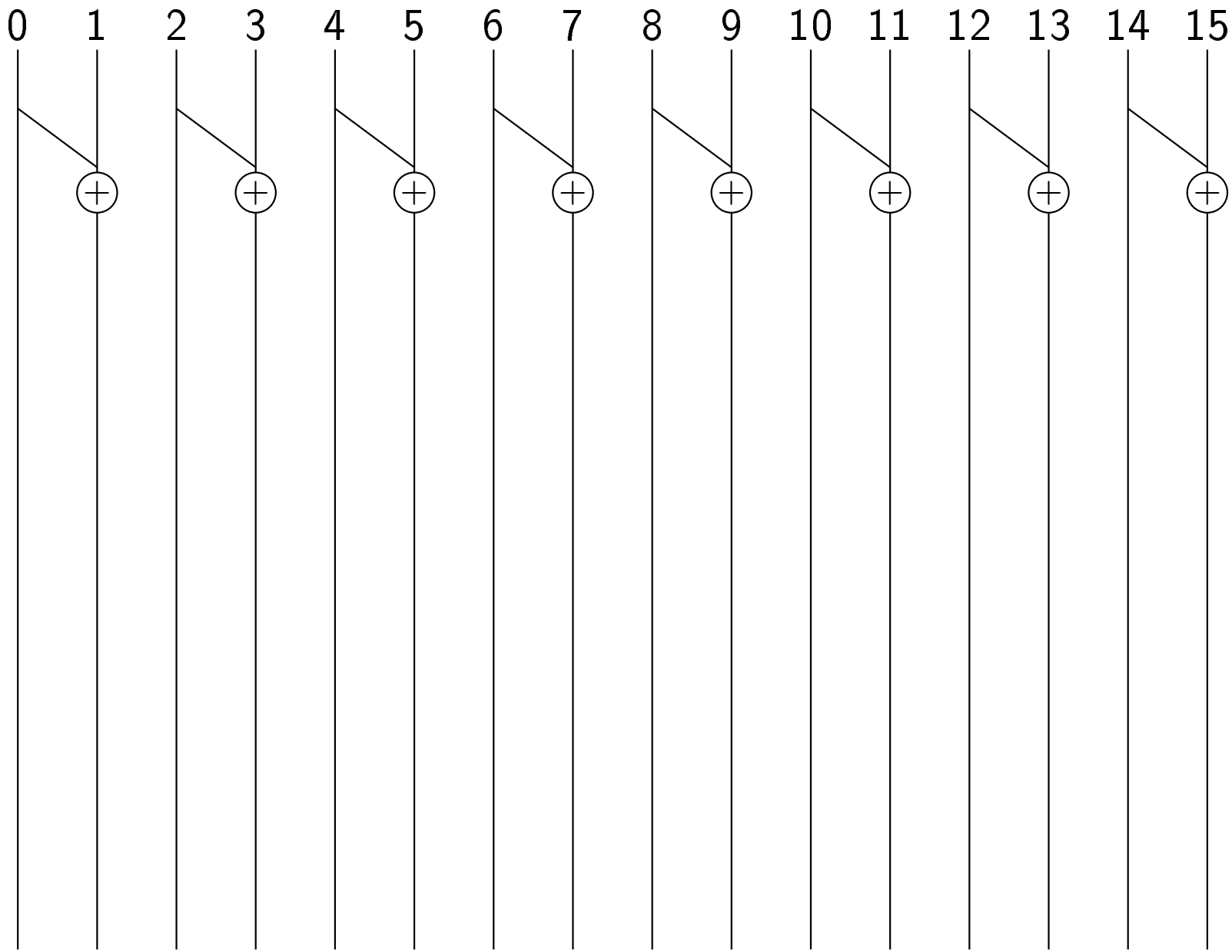
$$\begin{aligned}\mathrm{Cost}_{\mathrm{IS}} &= \sum_{i=1}^{\log_2(N)} (N - 2^{i-1}) = \left(\sum_{i=1}^{\log_2(N)} N\right) - \left(\sum_{i=1}^{\log_2(N)} 2^{i-1}\right) \\ &= N\log_2(N) - (1 + 2 + \cdots + N/2) \\ &= N\log_2(N) - (N - 1)\end{aligned}$$

$$N = 1024 \Rightarrow \begin{cases} \mathrm{Cost}_{\mathrm{HSH}} = 1024 \times 10 - (1024 - 1) & = 9217 \\ \mathrm{Cost}_{\mathrm{SS}} = 1024 - 1 & = 1023 \end{cases}$$

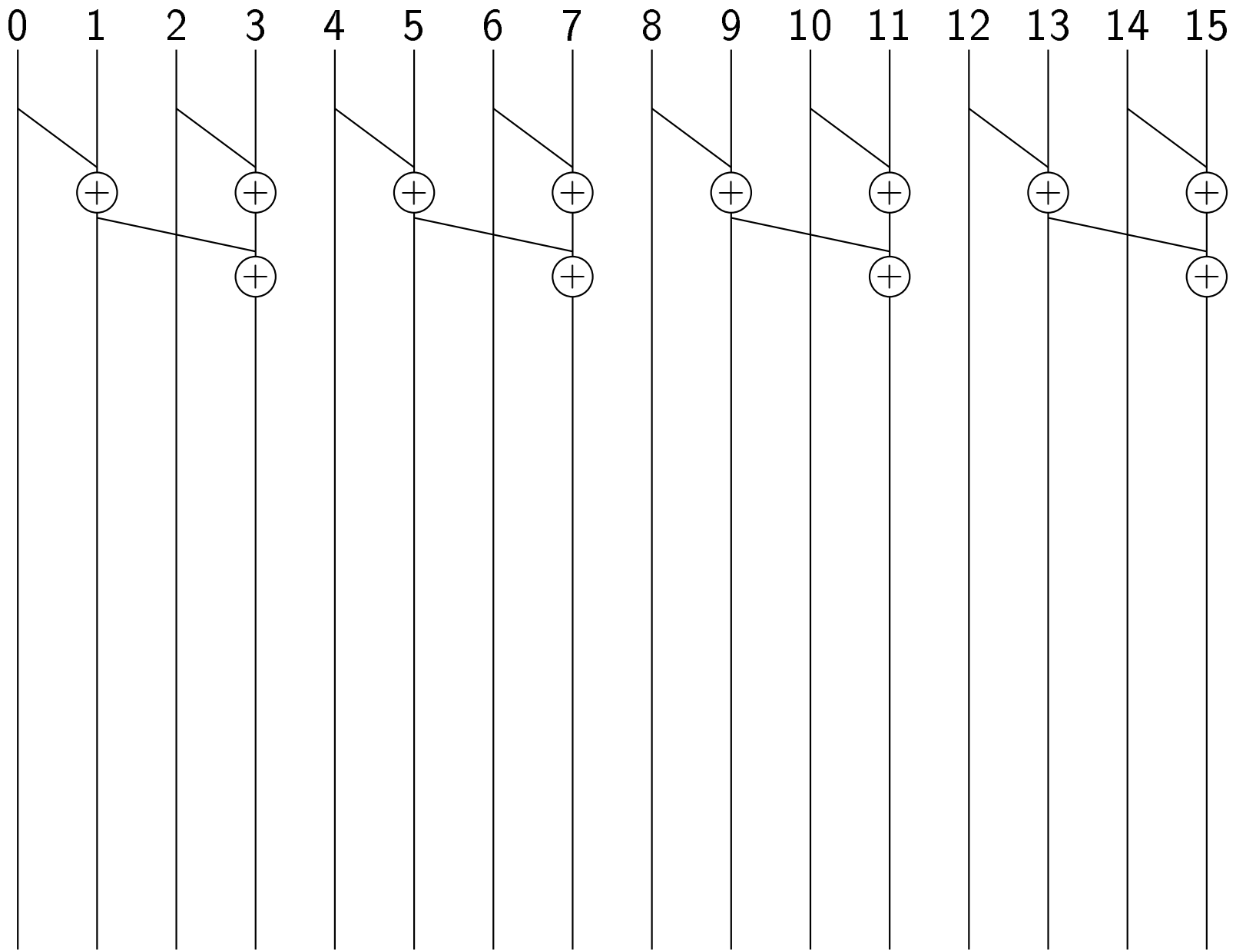
Blelloch Scan [Efficient]



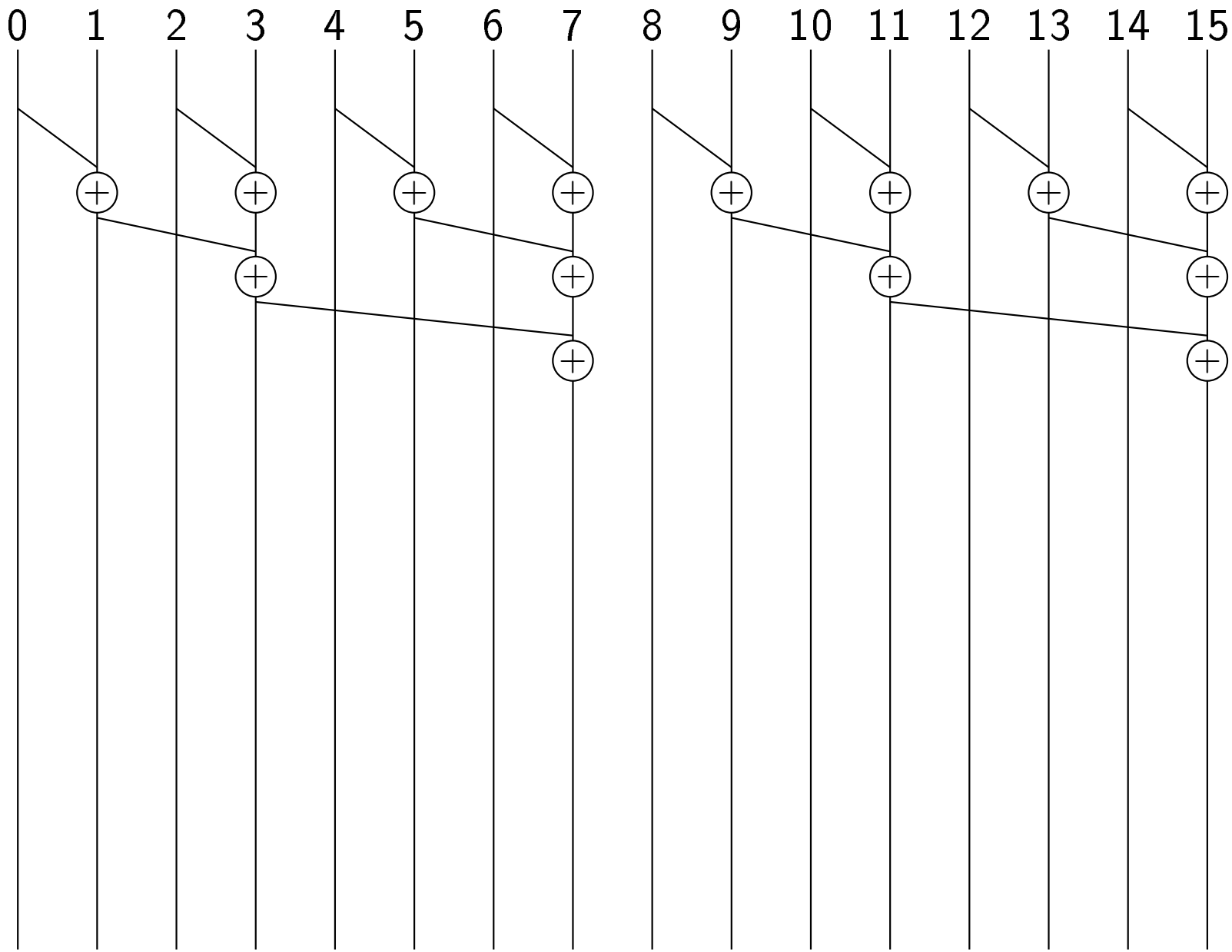
Blelloch Scan [Efficient]



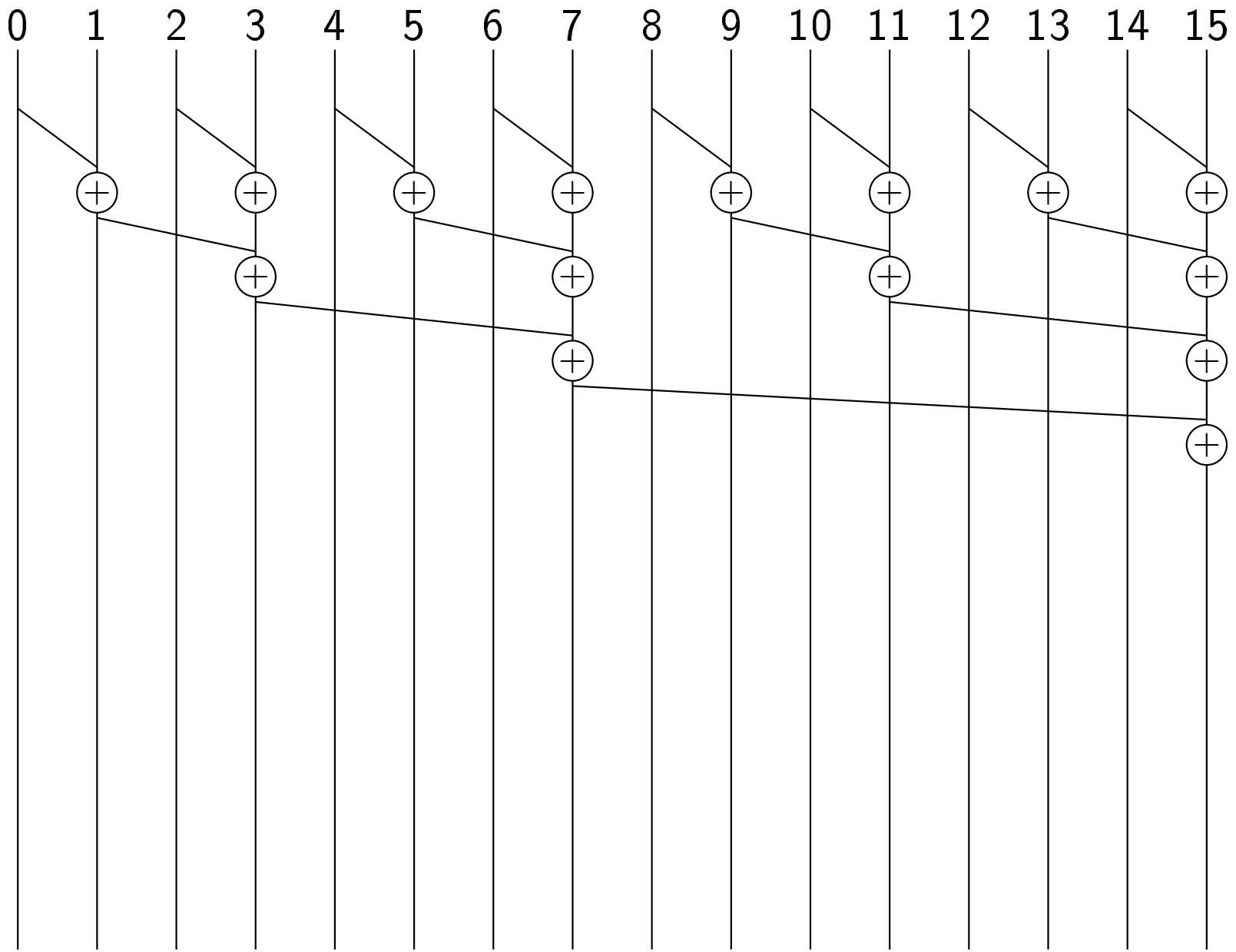
Blelloch Scan [Efficient]



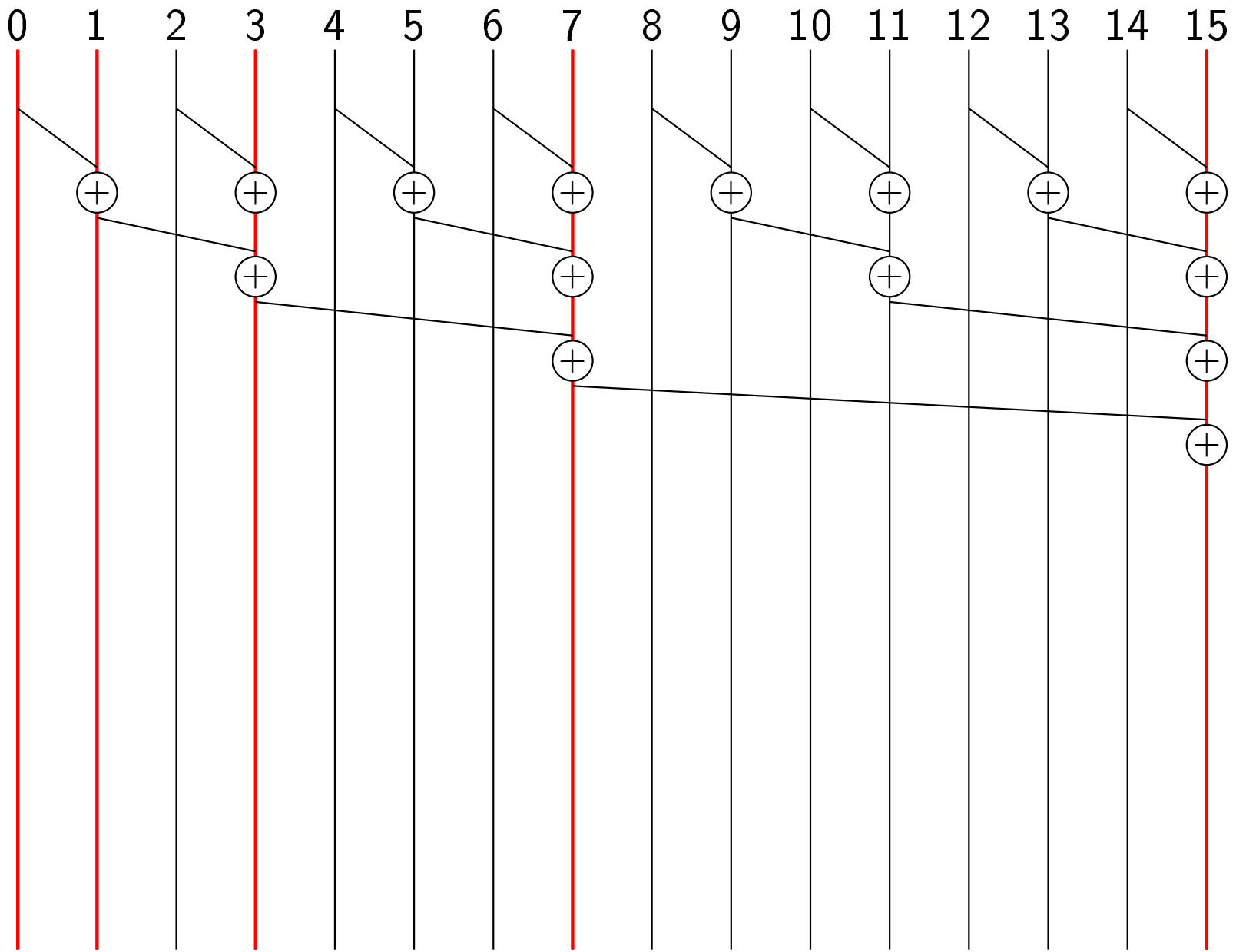
Blelloch Scan [Efficient]



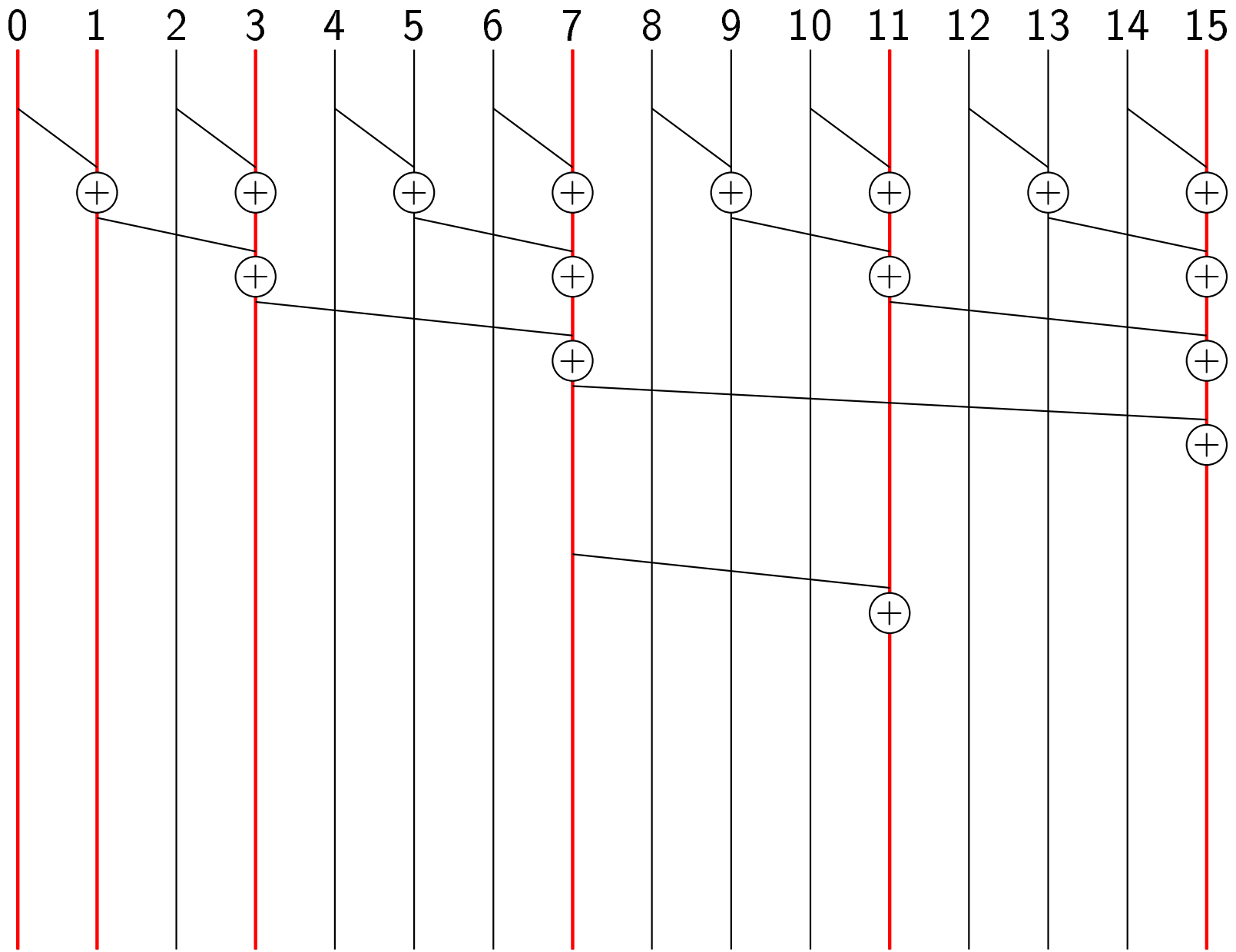
Blelloch Scan [Efficient]



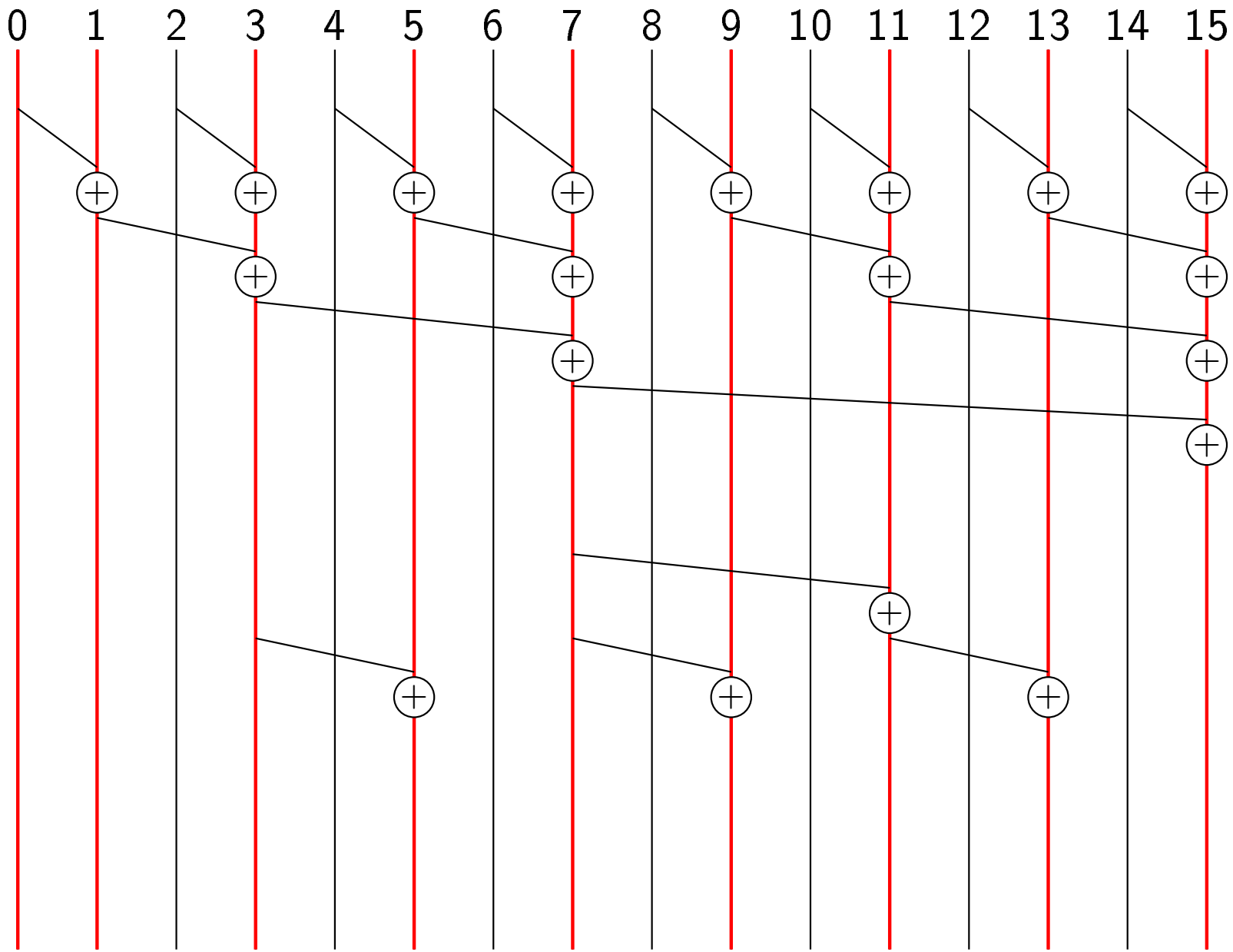
Blelloch Scan [Efficient]



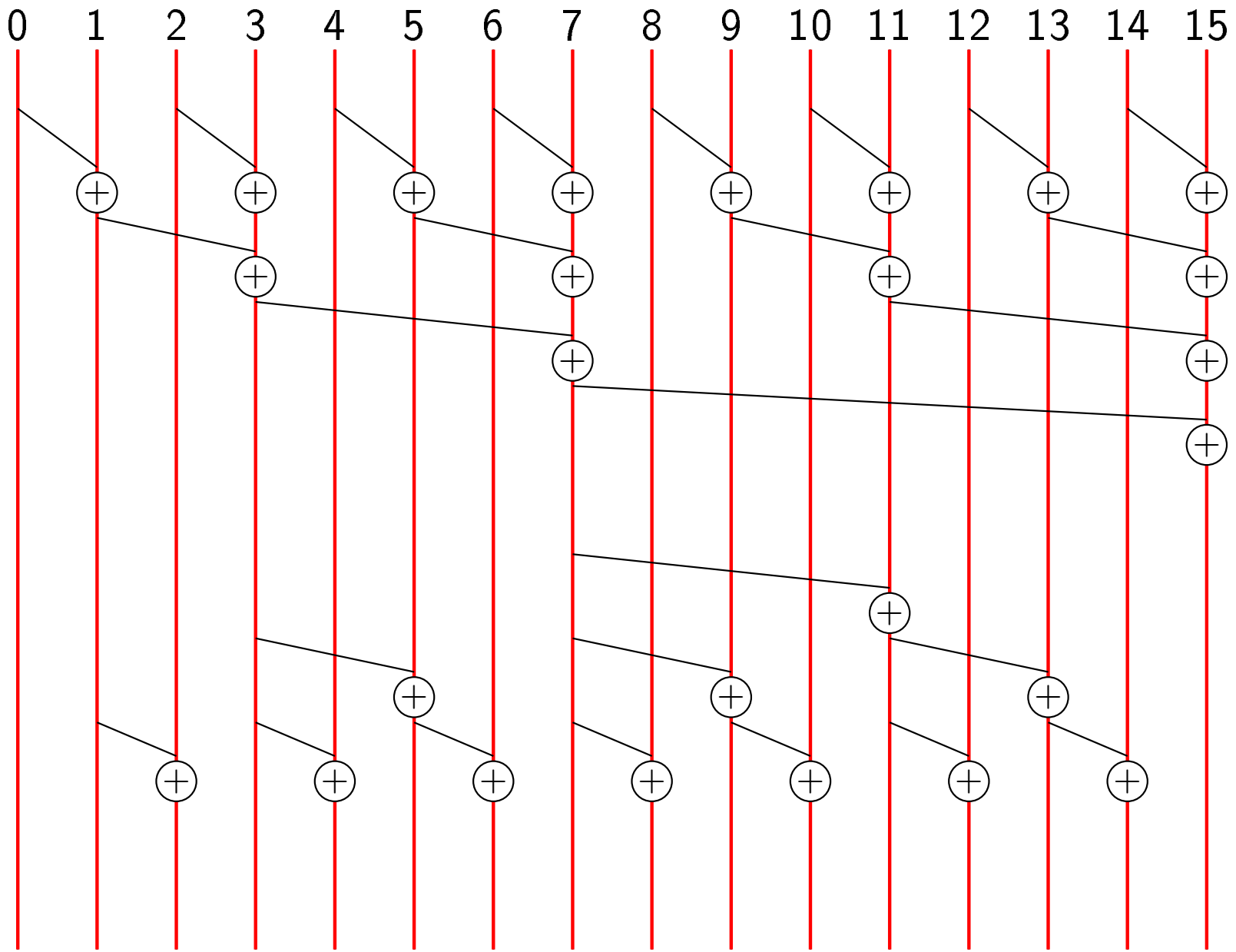
Blelloch Scan [Efficient]



Blelloch Scan [Efficient]



Blelloch Scan [Efficient]



Structure of Blelloch Scan

The basic structure will be:

- Copy `X` from global memory to shared memory `XY`
- Carry out reduction phase
- Carry out distribution phase
- Copy `XY` from shared memory to global memory `Y`

Code for Blelloch Scan: Copying

```
#define BLOCK_SIZE 1024 // the actual configured block size

__global void blelloch_scan(float *X, float *Y, int len)
{
    __shared__ float XY[BLOCK_SIZE];

    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < len)
        XY[threadIdx.x] = X[i];

    // Do Reduction phase here

    // Do Distribution phase here

    __syncthreads();

    if (i < len)
        Y[i] = XY[threadIdx.x];
}
```

Code for Blelloch Scan — Reduction

Reduction phase:

```
for(uint stride = 1; stride < blockDim.x; stride *= 2)
{
    __syncthreads();
    if ((threadIdx.x + 1) % (2*stride) == 0)
        XY[threadIdx.x] += XY[threadIdx.x - stride];
}
```

- All threads sync every iteration $\Rightarrow$ no deadlock
- No read/write races $\Rightarrow$ no double buffering required
  - No thread writes to a location that another thread reads from within the same iteration

Code for Blelloch Scan — Reduction

Reduction phase:

```
for(uint stride = 1; stride < blockDim.x; stride *= 2)
{
    __syncthreads();
    if ((threadIdx.x + 1) % (2*stride) == 0)
        XY[threadIdx.x] += XY[threadIdx.x - stride];
}
```

- All threads sync every iteration ⇒ no deadlock
- No read/write races ⇒ no double buffering required
  - No thread writes to a location that another thread reads from within the same iteration
- Iteration 1: threads 1, 3, 5,. . . do the addition

Code for Blelloch Scan — Reduction

Reduction phase:

```
for(uint stride = 1; stride < blockDim.x; stride *= 2)
{
    __syncthreads();
    if ((threadIdx.x + 1) % (2*stride) == 0)
        XY[threadIdx.x] += XY[threadIdx.x - stride];
}
```

- All threads sync every iteration ⇒ no deadlock
- No read/write races ⇒ no double buffering required
  - No thread writes to a location that another thread reads from within the same iteration
- Iteration 1: threads 1, 3, 5,. . . do the addition
- Iteration 2: threads 3, 7, 11, . . . do the addition

Code for Blelloch Scan — Reduction

Reduction phase:

```
for(uint stride = 1; stride < blockDim.x; stride *= 2)
{
    __syncthreads();
    if ((threadIdx.x + 1) % (2*stride) == 0)
        XY[threadIdx.x] += XY[threadIdx.x - stride];
}
```

- All threads sync every iteration $\Rightarrow$ no deadlock
- No read/write races $\Rightarrow$ no double buffering required
  - No thread writes to a location that another thread reads from within the same iteration
- Iteration 1: threads 1, 3, 5,. . . do the addition
- Iteration 2: threads 3, 7, 11, . . . do the addition
- **LOTS OF DIVERGENCE**

Code for Blelloch Scan — Reduction, minimising divergence

Reduction phase:

```
for(uint stride = 1; stride < blockDim.x; stride *= 2)
{
    __syncthreads();
    uint index = (threadIdx.x + 1) * stride * 2 - 1 ;
    if (index < blockDim.x)
        XY[index] += XY[index - stride];
}
```

Code for Blelloch Scan — Reduction, minimising divergence

Reduction phase:

```
for(uint stride = 1; stride < blockDim.x; stride *= 2)
{
    __syncthreads();
    uint index = (threadIdx.x + 1) * stride * 2 - 1 ;
    if (index < blockDim.x)
        XY[index] += XY[index - stride];
}
```

- Iteration 1: thread 0 uses index 1, $1 \rightarrow 3$, $2 \rightarrow 5$,...

Code for Blelloch Scan — Reduction, minimising divergence

Reduction phase:

```
for(uint stride = 1; stride < blockDim.x; stride *= 2)
{
    __syncthreads();
    uint index = (threadIdx.x + 1) * stride * 2 - 1 ;
    if (index < blockDim.x)
        XY[index] += XY[index - stride];
}
```

- Iteration 1: thread 0 uses index 1, $1 \rightarrow 3$, $2 \rightarrow 5$,...
- Iteration 2: thread 0 uses index 3, $1 \rightarrow 7$, $2 \rightarrow 11$,

Code for Blelloch Scan — Reduction, minimising divergence

Reduction phase:

```
for(uint stride = 1; stride < blockDim.x; stride *= 2)
{
    __syncthreads();
    uint index = (threadIdx.x + 1) * stride * 2 - 1 ;
    if (index < blockDim.x)
        XY[index] += XY[index - stride];
}
```

- Iteration 1: thread 0 uses index 1, $1 \rightarrow 3$, $2 \rightarrow 5$,...
- Iteration 2: thread 0 uses index 3, $1 \rightarrow 7$, $2 \rightarrow 11$,
- Thus the working threads are contiguous, starting at 0 $\Rightarrow$ **minimal divergence**

Code for Blelloch Scan — Distribution

Distribution phase:

```
for(uint stride = BLOCK_SIZE/4; stride > 0; stride /= 2)
{
    __syncthreads();
    uint index = (threadIdx.x + 1) * stride * 2 - 1 ;
    if (index + stride < BLOCK_SIZE)
        XY[index + stride] += XY[index];
}
```

- All threads sync every iteration ⇒ no deadlock
- No read/write races ⇒ no double buffering required
- `stride` divides by 2 from `BLOCK_SIZE/4` each iteration
- Each iteration we *push* the `XY` values:
  - from locations: 1 less than multiples of twice `stride`
  - *to* locations: 1 stride above the *from*

Code for Blelloch Scan — Distribution

Distribution phase:

```
for(uint stride = BLOCK_SIZE/4; stride > 0; stride /= 2)
{
    __syncthreads();
    uint index = (threadIdx.x + 1) * stride * 2 - 1 ;
    if (index + stride < BLOCK_SIZE)
        XY[index + stride] += XY[index];
}
```

- All threads sync every iteration ⇒ no deadlock
- No read/write races ⇒ no double buffering required
- `stride` divides by 2 from `BLOCK_SIZE/` `4` each iteration
- Each iteration we *push* the `XY` values:
  - from locations: 1 less than multiples of twice `stride`
  - *to* locations: 1 stride above the *from*
- Assuming `BLOCK_SIZE = 16`, to match our diagram:
- Iteration 1: thread 0 uses index 7, 1 → 15, 2 → 23,...

Code for Blelloch Scan — Distribution

Distribution phase:

```
for(uint stride = BLOCK_SIZE/4; stride > 0; stride /= 2)
{
    __syncthreads();
    uint index = (threadIdx.x + 1) * stride * 2 - 1 ;
    if (index + stride < BLOCK_SIZE)
        XY[index + stride] += XY[index];
}
```

- All threads sync every iteration ⇒ no deadlock
- No read/write races ⇒ no double buffering required
- `stride` divides by 2 from `BLOCK_SIZE/` `4` each iteration
- Each iteration we *push* the `XY` values:
  - from locations: 1 less than multiples of twice `stride`
  - *to* locations: 1 stride above the *from*
- Assuming `BLOCK_SIZE = 16`, to match our diagram:
- Iteration 1: thread 0 uses index 7, 1 → 15, 2 → 23,...
- Iteration 2: thread 0 uses index 3, 1 → 7, 2 → 11,

Code for Blelloch Scan — Distribution

Distribution phase:

```
for(uint stride = BLOCK_SIZE/4; stride > 0; stride /= 2)
{
    __syncthreads();
    uint index = (threadIdx.x + 1) * stride * 2 - 1 ;
    if (index + stride < BLOCK_SIZE)
        XY[index + stride] += XY[index];
}
```

- All threads sync every iteration ⇒ no deadlock
- No read/write races ⇒ no double buffering required
- `stride` divides by 2 from `BLOCK_SIZE/` `4` each iteration
- Each iteration we *push* the `XY` values:
  - from locations: 1 less than multiples of twice `stride`
  - *to* locations: 1 stride above the *from*
- Assuming `BLOCK_SIZE = 16`, to match our diagram:
- Iteration 1: thread 0 uses index 7, 1 → 15, 2 → 23,...
- Iteration 2: thread 0 uses index 3, 1 → 7, 2 → 11,
- Thus the working threads are contiguous, starting at 0 ⇒ **minimal divergence**

Threads vs Vector Elements

- In the previous code, we dealt with a number of vector elements equal to the block size
- But, at maximum, we only use half that number of threads
- Better to deal with a number of vector elements equal to twice the block size and use all the threads
- Make sure to choose the block size so that `XY` fits within the shared memory per block limit
- Easy modification of code to make this happen (exercise for reader!)

Cost of Blelloch Scan

Given N elements:

- During the Reduction phase:
  - Iteration 1: N/2 floating point additions
  - Iteration 2: N/4 floating point additions
  - ...
  - Last iteration: 1 floating point addition
- During the Distribution phase:
  - Last iteration: N/2 - 1 floating point additions
  - 2^{nd} last iteration: N/4 - 1 floating point operations

$$\begin{aligned}\text{COST}_{\text{ES}} &= 2 \times (N/2 + N/4 + \ldots 1) - \log_2(N/2) \\ &= 2(N-1) - (\log_2(N) - 1) \\ &= 2N - log_2(N) - 1\end{aligned}$$

$$N = 1024 \Rightarrow \begin{cases} \text{Cost}_{\text{BS}} = 2 \times 1024 - 10 - 1 & = 2037 \\ \text{Cost}_{\text{HSH}} = 1024 \times 10 - (1024 - 1) & = 9217 \\ \text{Cost}_{\text{SS}} = 1024 - 1 & = 1023 \end{cases}$$

Why is HSH scan slower?

- The Hillis Steele Horn scan (HSH) uses more additions than the Blelloch scan (B), but takes fewer steps (iterations). Shouldn't it be faster?

Why is HSH scan slower?

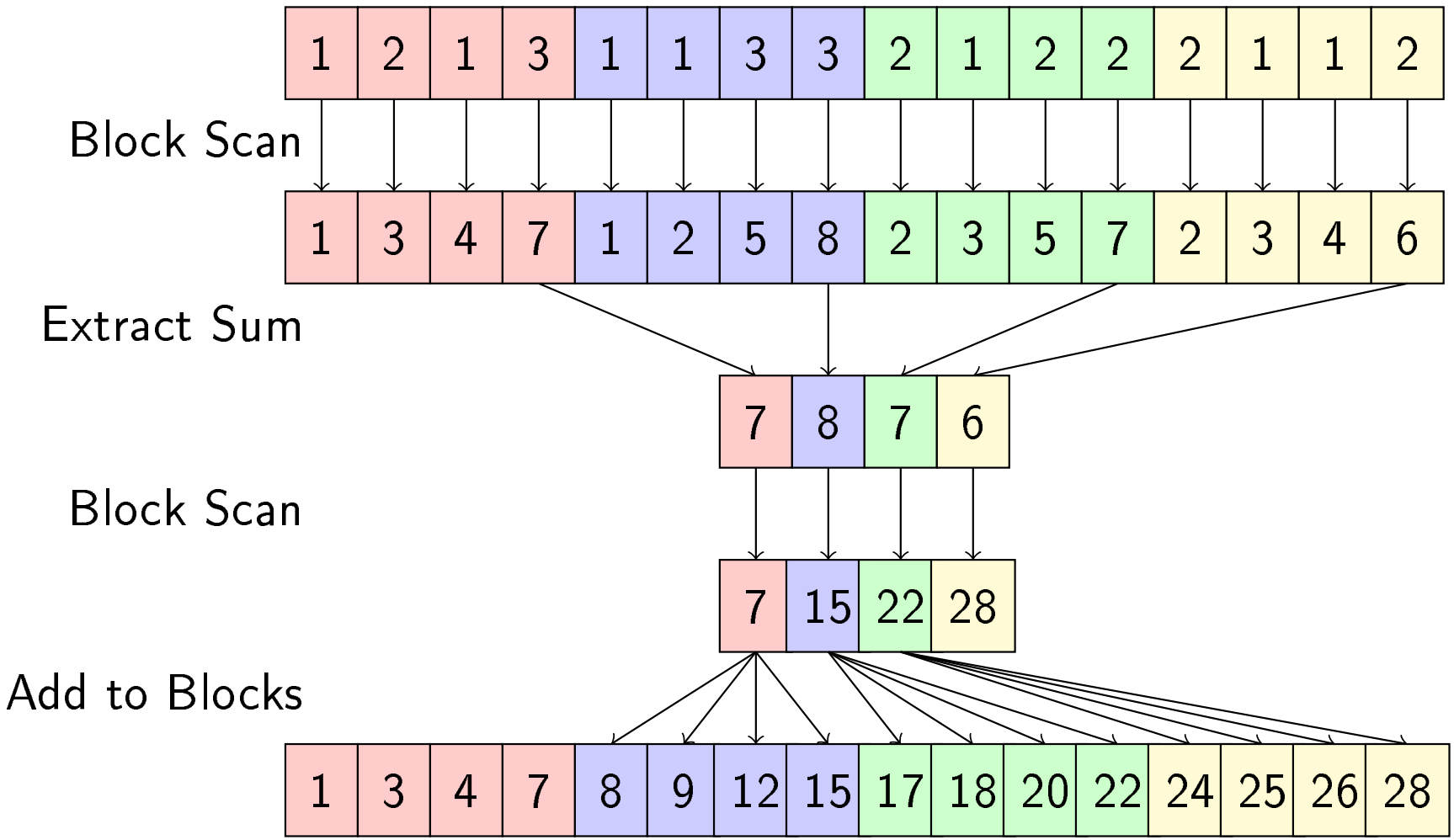
- The Hillis Steele Horn scan (HSH) uses more additions than the Blelloch scan (B), but takes fewer steps (iterations). Shouldn't it be faster?
- Yes IS takes fewer steps but ...
- Remember that the gpu can only execute a small number of warps simultaneously (although it can hold as many warps "*ready to switch in*" as it has cores)
- actual execution of (groups of) warps happens in an interleaved/sequential fashion
- so, in general, more work requires more warps which takes more time

Scaling up to Large Vectors

Until now we have assumed that whole vector fits in a single block. We need to make it work on large vectors that require many blocks.

- Run the scan kernel on the whole vector: each block has been scanned independently into the Y blocks, but the block results have not been propagated between blocks.
- Run a second kernel (this is the same kernel as in the first step, just with different data) that runs a scan on the last elements of each Y block and puts the result into a new vector S
- Run a third kernel that adds each element of S to all elements of the corresponding block of Y.

Large Vector Scan



Maximum Vector Size

Assume the maximum block size is 1024 threads. Then the Blelloch block scan can handle segments of the vector of size 2048 words

- Level 1 & 2 scans only $\Rightarrow$
  - Level 2 scans a single segment = 2K words
  - Level 1 scans 2K segments = 4M words
- Level 1, 2 & 3 scans $\Rightarrow$
  - Level 3 scans a single segment = 2K words
  - Level 2 scans 2K segments = 4M words
  - Level 1 scans 4M segments = 8G words = 32G bytes

But lab machines (GTX 960) have 2G byte global memory $\Rightarrow$ for very large vectors, scans need to be iterated by the host over maximum parallel GPU scans

Code structure for 3 level scan

Pseudocode for a 3 level scan of a vector `d_X` of size `N` to `d_Y`, using a block size of 1024 (= segment size of 2048) might look like this:

```
Allocate globals
    d_X, d_Y both of size N
    d_Sum1, d_Sum1_scanned both of size ceil(N/2048)
    d_Sum2, d_Sum2_scanned both of size 1

Copy host h_X to gpu d_X

block_scan<<<...>>> (d_X, d_Y, N)
extract_sum<<<...>>> (d_Y, d_Sum1)
block_scan<<<...>>> (d_Sum1, d_Sum1_scanned, ceil(N/2048))
extract_sum<<<...>>> (d_Sum1_scanned, d_Sum2)
block_scan<<<...>>> (d_Sum2, d_Sum2_scanned, 1)
block_add<<<...>>> (d_Sum2_scanned, d_Sum1_scanned, ...)
block_add<<<...>>> (d_Sum1_scanned, d_Y, ...)

Copy gpu d_Y to host h_Y
```

- no host/gpu copies between kernels
- `block_scan` and `extract_sum` can be merged into one kernel (with extra bool parameter to request it)