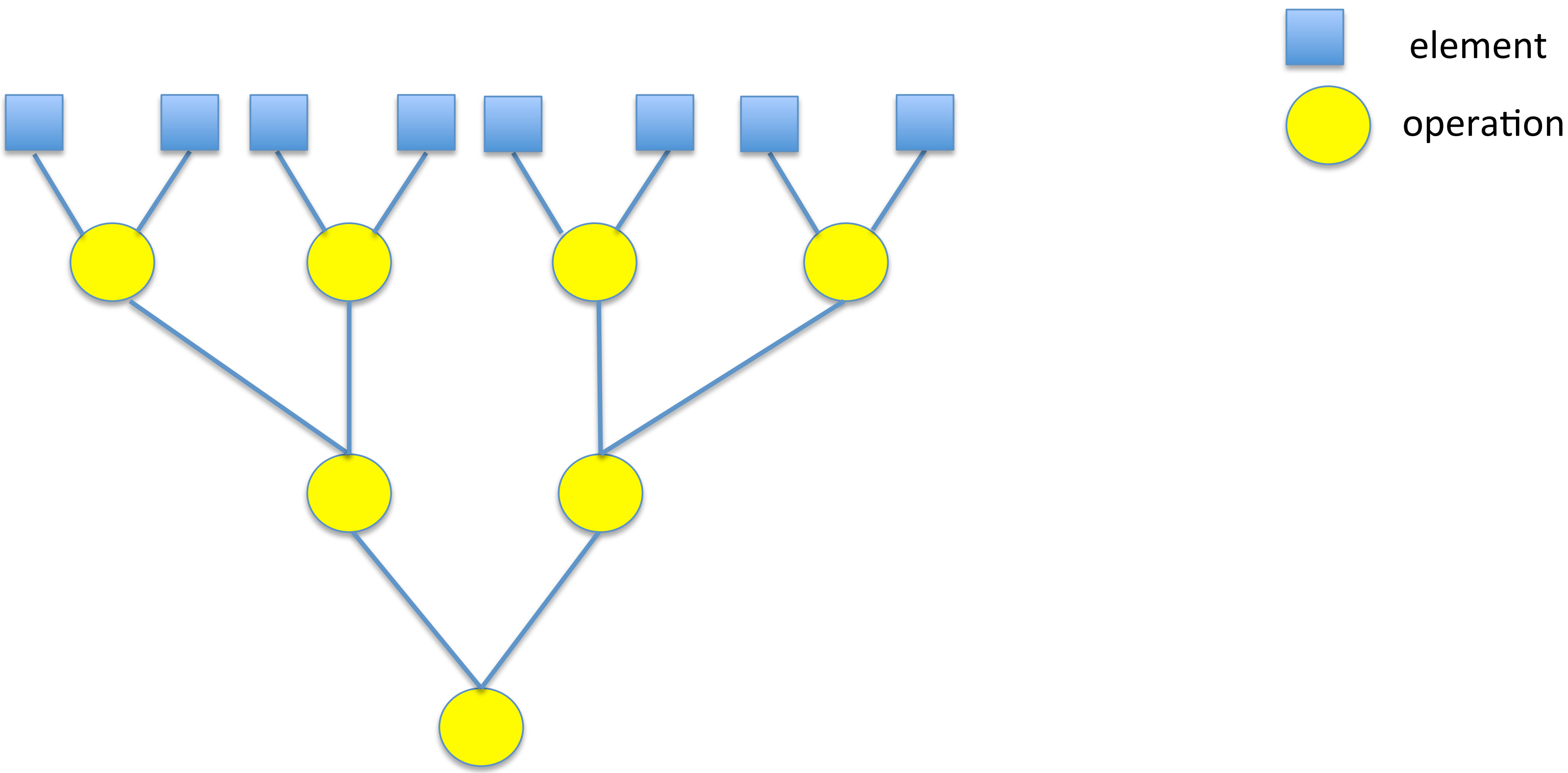
High Performance Computing

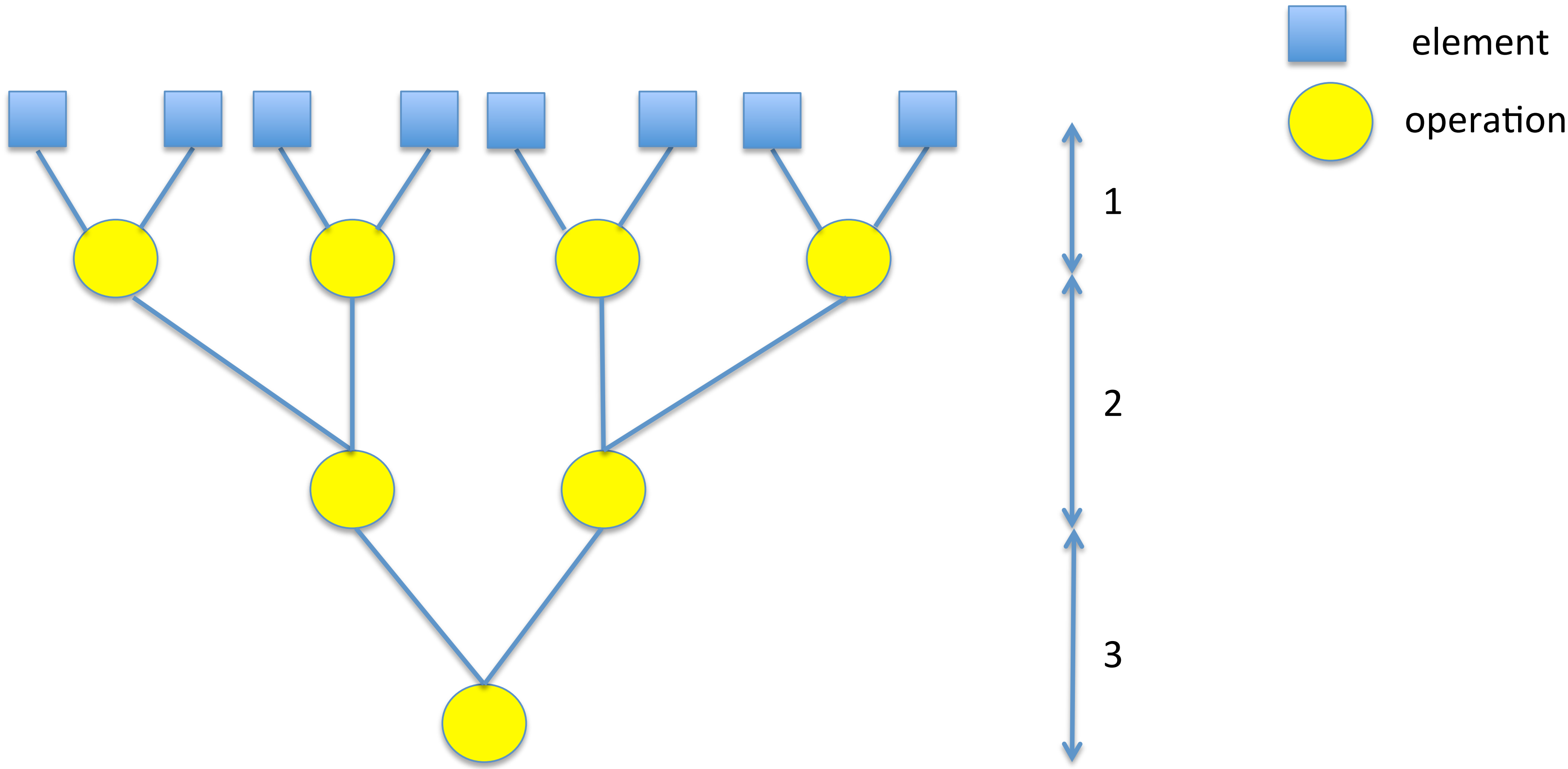
Term 4 2018/2019

Lecture 9

Step complexity vs work complexity



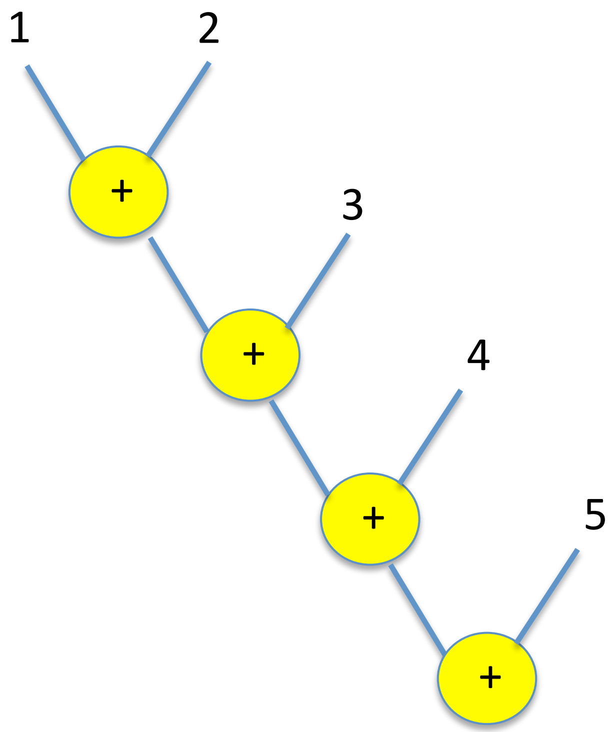
Step complexity vs work complexity



Step complexity = 3
Work complexity = 7

Serial reduce

1 + 2 + 3 + 4 + 5

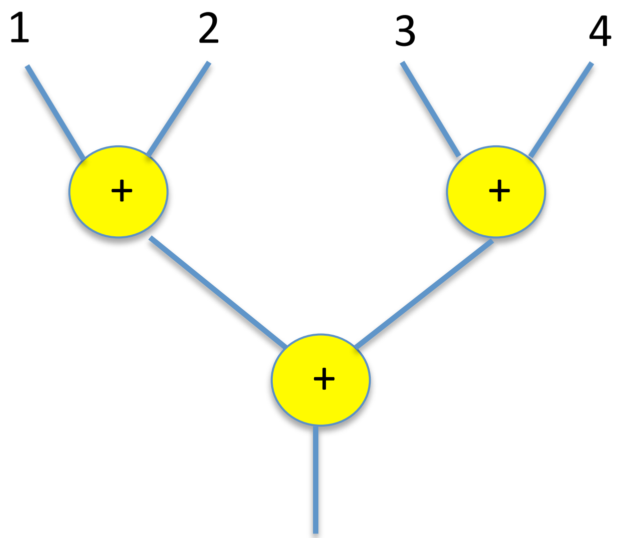


Reduce:

1) set of elements
2) reduction operator (binary and associative)

step complexity: 4
work complexity: 4

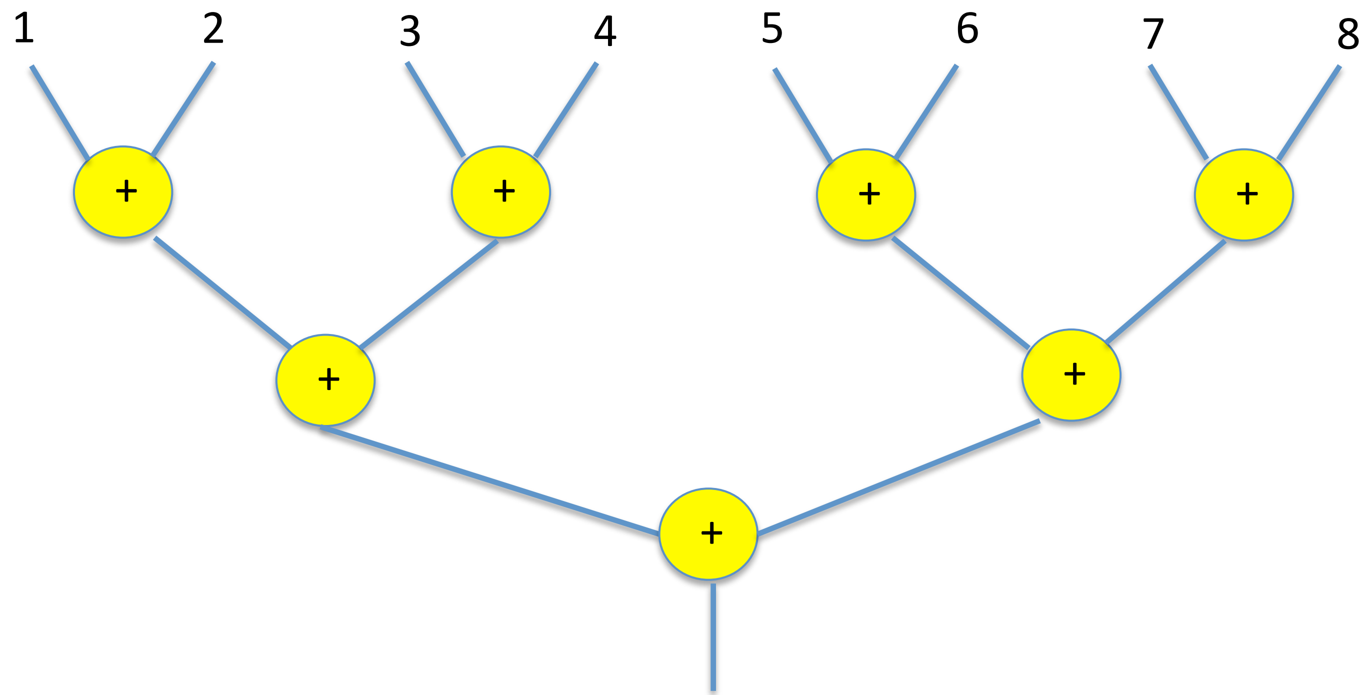
Parallel reduce



step complexity: 2
work complexity: 3
number of elements: 4

Parallel reduce

1 + 2 + 3 + 4 + 5



step complexity: 3
work complexity: 7
number of elements: 8

$\log_2 n$ steps

CUDA parallel reduce (naive)

```
__global__ void reduce(float * d_out, float * d_in)
{
    int myId = threadIdx.x + blockDim.x * blockIdx.x;
    int tid = threadIdx.x;

    unsigned int s = blockDim.x / 2;

    while(s>0)
    {
        if (tid<s)
        {
            d_in[myId]+=d_in[myId+s];
        }
        __syncthreads();
        s=(unsigned int)s/2;
    }

    if (tid == 0)
    {
        d_out[blockIdx.x] = d_in[myId];
        printf("%d\t%f\n", blockIdx.x, d_in[myId]);   // try without printf to see the speedup
    }

}
```

CUDA parallel reduce (shared mem)

```
__global__ void shmem_reduce(float * d_out, float * d_in)
{
     extern float * sdata[];

     int myId = threadIdx.x + blockDim.x * blockIdx.x;
     int tid = threadIdx.x;

     sdata[tid]=d_in[myId];

     unsigned int s = blockDim.x / 2;

     while(s>0)
     {
          if (tid<s)
          {
               sdata[tid]+=sdata[tid+s];
          }
          __syncthreads();
          s=(unsigned int)s/2;
     }

     if (tid == 0)
     {
          d_out[blockIdx.x] =sdata[0];
     }

}
```

kernel call:

reduce<<<1024,1024,**1024*sizeof(float)**>>>(d_intermediate, d_in);

Scan

1 2 3 4 5

cumulative sum (inclusive scan):

1, 3, 6, 10, 15,...

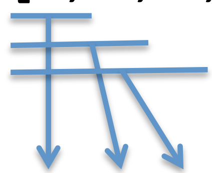
exclusive scan:

0, 1, 3, 6, 10,...

you need a binary operator and identity element (I)

Parallel Scan using reduce

In: [1, 2, 3, 4, 5, 6, 7, 8]

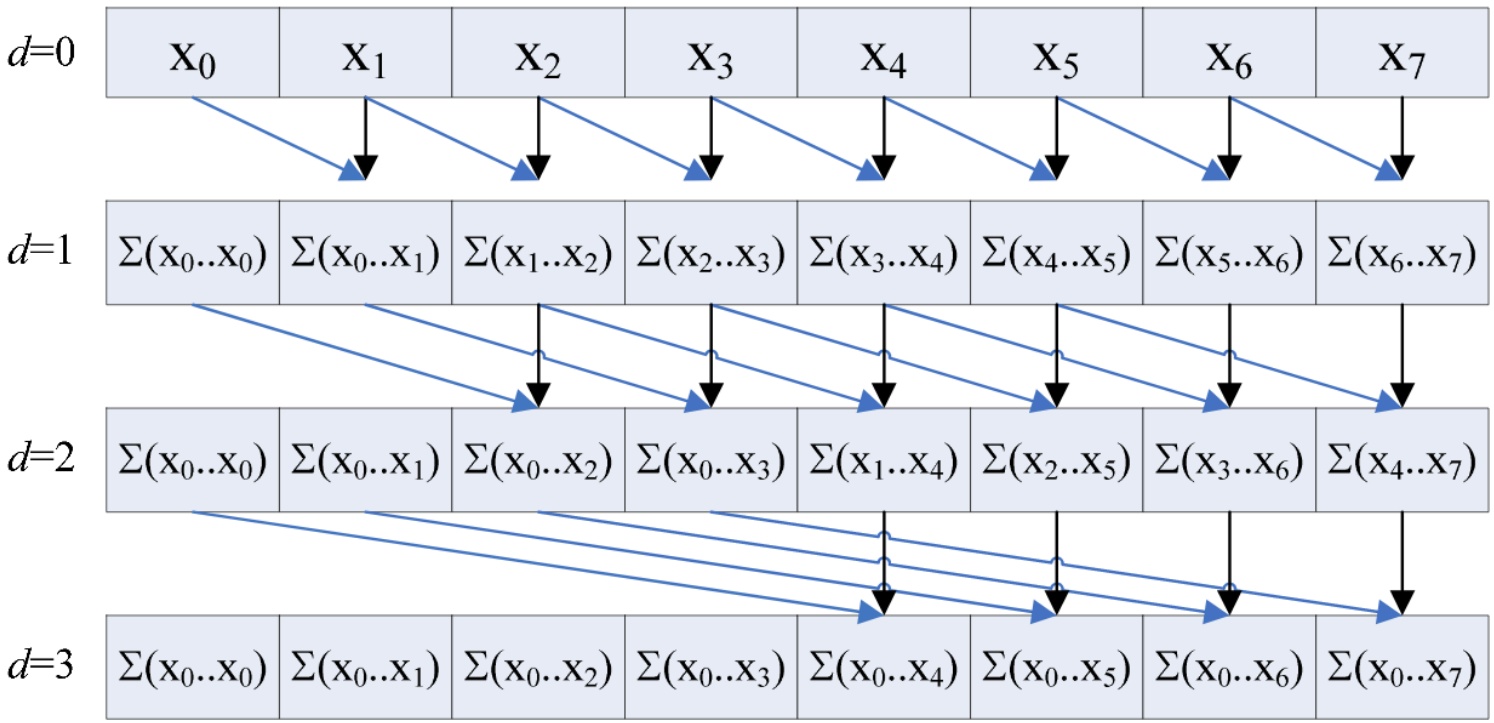


Out: [1, 3, 6, 10, 15, 21, 28, 36]

step complexity: log(n)
work complexity: $O(n^2)$

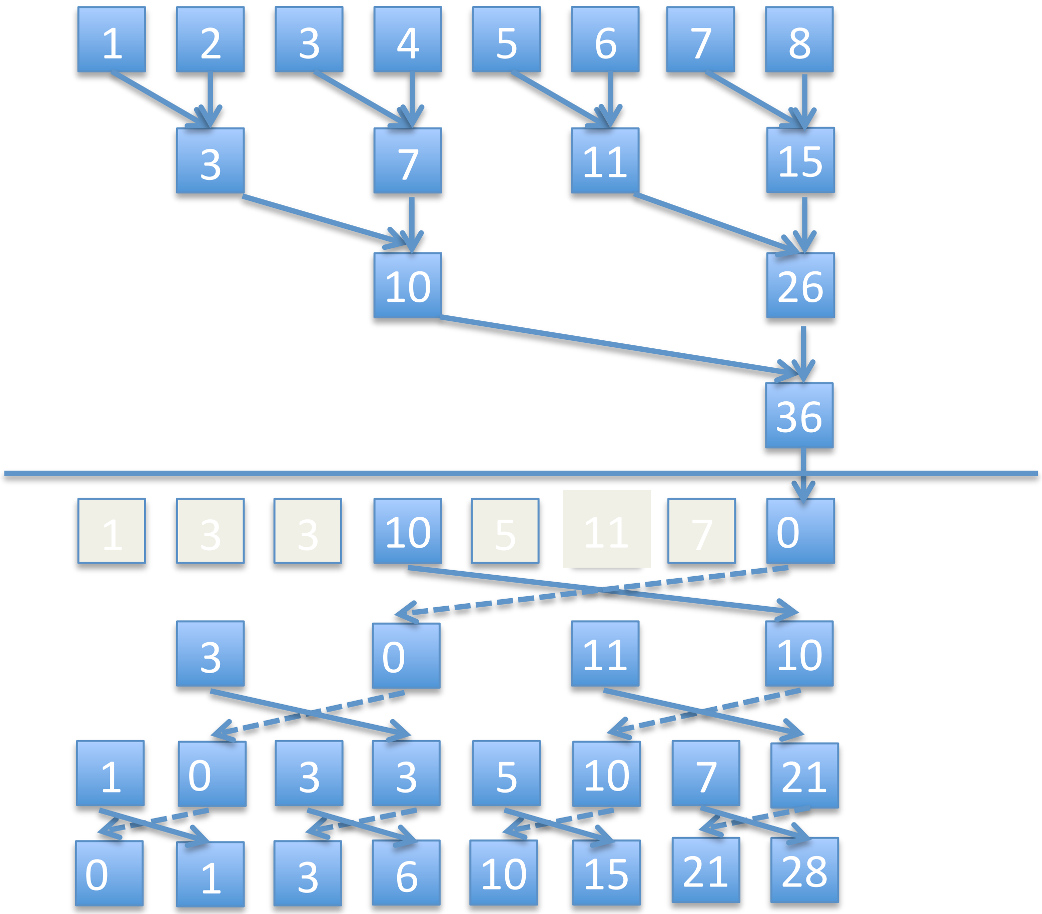
Parallel Scan Hillis-Steele

- Note that a implementation of the algorithm shown in picture requires two buffers of length n (shown is the case $n=8=2^3$)
- Assumption: the number n of elements is a power of 2: $n=2^M$



steps: log(n)
work: O(n log(n))

Parallel Prescan Blelloch



reduction

downsweep

steps: 2 log(n)
work: O(n)

Histogram

```
__global__ void naive_histo(int *d_bins, const int *d_in, const int BIN_COUNT)
{
    int myId = threadIdx.x + blockDim.x * blockIdx.x;
    int myItem = d_in[myId];
    int myBin = myItem % BIN_COUNT;
    d_bins[myBin]++;
}
```

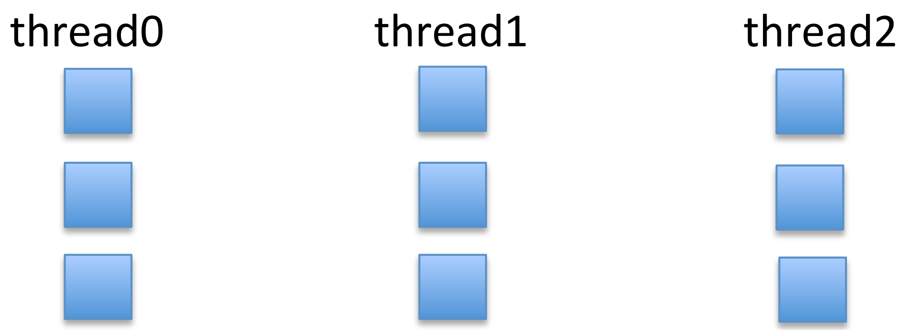
Histogram (atomics)

```
__global__ void naive_histo(int *d_bins, const int *d_in, const int BIN_COUNT)
{
    int myId = threadIdx.x + blockDim.x * blockIdx.x;
    int myItem = d_in[myId];
    int myBin = myItem % BIN_COUNT;
    d_bins[myBin]++;
}
```

```
__global__ void simple_histo(int *d_bins, const int *d_in, const int BIN_COUNT)
{
    int myId = threadIdx.x + blockDim.x * blockIdx.x;
    int myItem = d_in[myId];
    int myBin = myItem % BIN_COUNT;
    //d_bins[myBin]++;
    atomicAdd(&(d_bins[myBin]), 1);
}
```

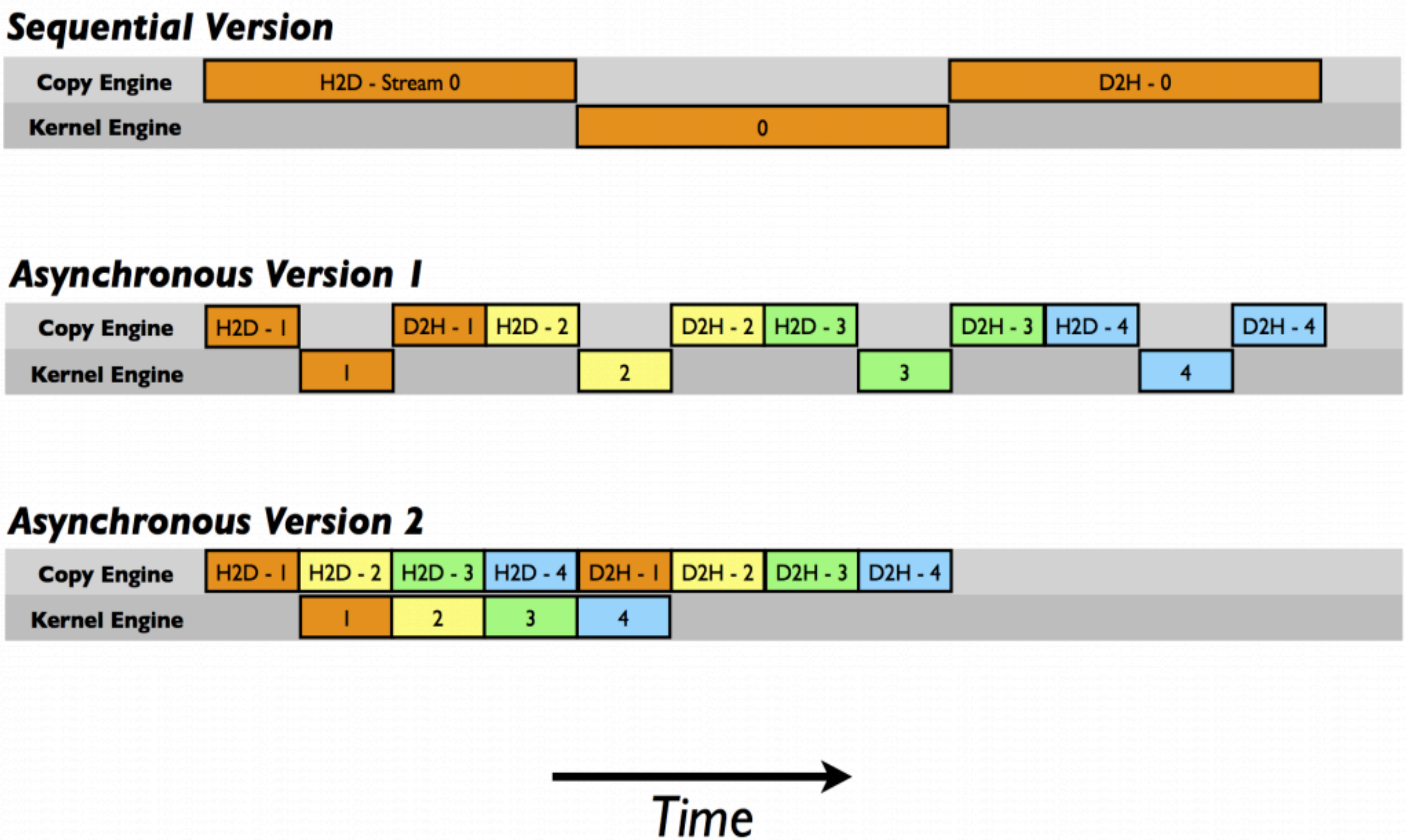
Histogram (reduce-based)

1. Each thread calculates its own histogram



2. Reduce the histogram element by element

CUDA Streams



CUDA Streams

```
cudaStream_t mystream;
cudaError_t result;


// create and destroy stream
result = cudaStreamCreate(&mystream);
result = cudaStreamDestroy(mystream);


// asynchronously copy from host to device
result = cudaMemcpyAsync(
   arr_dev, arr_host, size, cudaMemcpyHostToDevice, mystream
);


// launch kernel
increment<<<gridSize, blockSize, shSize, mystream>>>(arr_dev);
```

PyCuda

- wrapper of CUDA in Python
- you still write your kernel in C and send it as a string
- try it yourself ☺