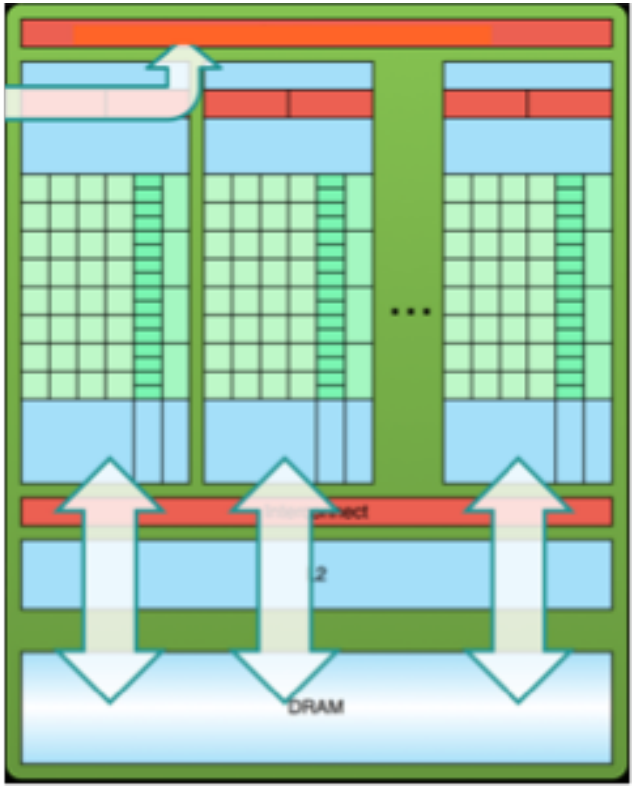
Programming Massively Parallel Hardware

Lab session 2
Basic blocks: Reduce and Scan in CUDA

Rasmus Fonseca

Synchronization

- Job is divided into subtasks
- Each subtask is solved by threads in a thread block
- Each thread block is put on a Streaming Multiprocessor (SM)



Synchronization

- A block contains many threads
- A block can run on more than one SM
- Threads in a block can cooperate
- Threads on the same SM can cooperate
- The programmer can specify the order two blocks (same kernel) will run
- The programmer can specify the order two blocks (different kernels) will run

Synchronization

- A block contains many threads
- A block can run on more than one SM
- Threads in a block can cooperate
- Threads on the same SM can cooperate
- The programmer can specify the order two blocks (same kernel) will run
- The programmer can specify the order two blocks (different kernels) will run

Synchronization

Collaborate in block using shared memory:

```
__global void kernelCall(...){
  extern __shared__ int sharedArr[];
}
…
kernelCall<<<blocks, threads, shared_mem_size>>>(...)
```

Avoid confusion within threads:

```
__syncthreads()
```

Thread collaboration in block

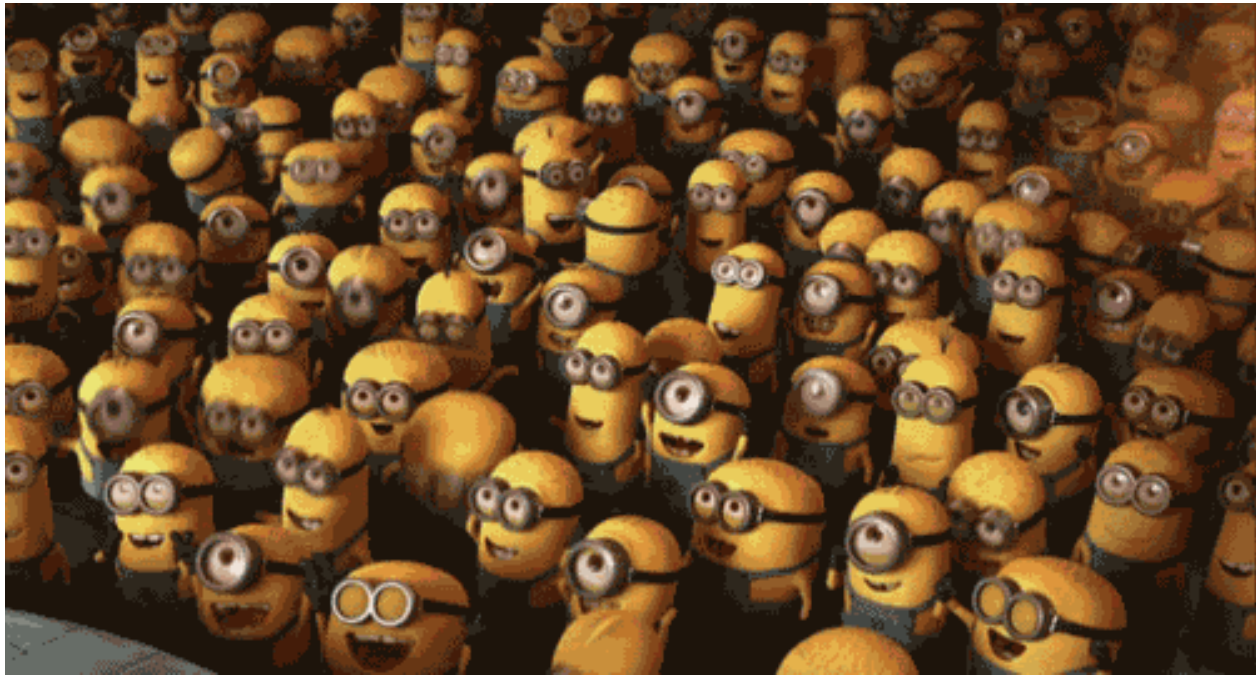
Assume we want to shift every element of an array left by one (put element 0 on the end):

```
__global__ void shiftLeftKernel(float *d_in, float *d_out)
{
    const unsigned int tid = threadIdx.x;
    extern __shared__ float s_arr[];

    //Copy to shared memory
    s_arr[tid] = d_in[tid];

    //Perform operation
    s_arr[tid] = s_arr[ tid+1 ];

    //Copy back to global memory
    d_out[tid] = s_arr[tid];
}
...
    shiftLeftKernel<<< 1, num_threads, shared_mem_size >>>(d_in, d_out);
```

Thread collaboration in block

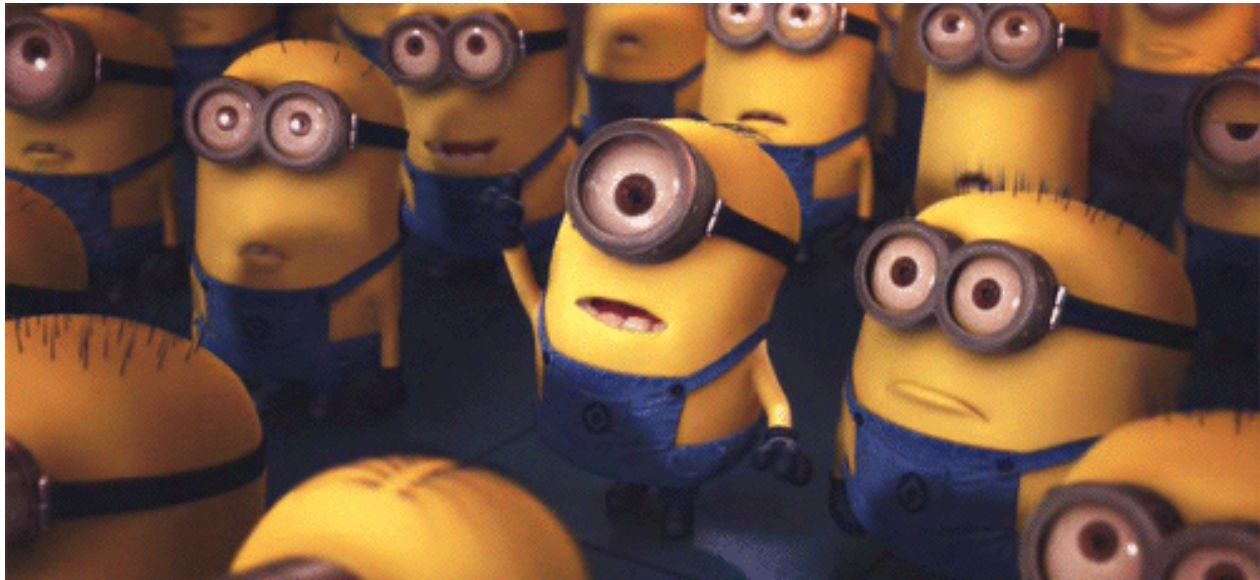
```
__global__ void shiftLeftKernel(float *d_in, float *d_out)
{
    const unsigned int tid = threadIdx.x;
    extern __shared__ float s_arr[];

    //Copy to shared memory
    s_arr[tid] = d_in[tid];

    __syncthreads();

    //Perform operation
    float newVal = s_arr[ (tid+1)%blockDim.x ];
    __syncthreads();
    s_arr[tid] = newVal;

    //Copy back to global memory
    d_out[tid] = s_arr[tid];
}
...
    shiftLeftKernel<<< 1, num_threads, shared_mem_size >>>(d_in, d_out);
```

Thats basically CUDA!



Thats basically CUDA!

(though it takes a bit more to use it efficiently)



Fundamental parallel algorithms

- ***Reduce***
- ***Scan (segmented, inclusive/exclusive)***
- ***Sort***
- ***Histogram***
- ***...***

Reduce

Reduce

Input: ?

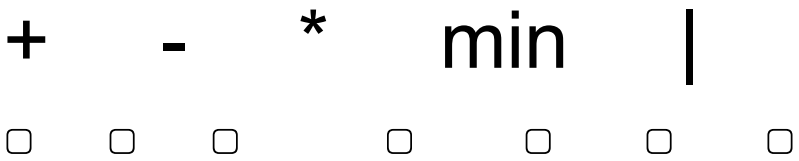
Output: ?

Reduce

Reduce

Input: List of elements, an associative binary operator on elements, and identity element

Output: Value of same type as element



Reduce

Reduce

Input: List of elements, an associative binary operator on elements, and identity element

Output: Value of all elements (and identity) operated on recursively.

+ - * min | & /

▣ ▢ ▣ ▣ ▣ ▣ ▢

Serial reduce

```
float reduce_sum(float *arr, float identity, int sz)
{
    float sum = identity;
    for(int i=0;i<sz;i++){
        sum+=arr[i];
    }
    return sum;
}
```

Work complexity: O(n)

Step complexity: O(n)

Parallel reduce

```
__global__ void reduceParallel_sum(float *d_in, float identity)
{
    const unsigned int threadId = threadIdx.x + blockDim.x*blockIdx.x;
    const unsigned int blockThreadId = threadIdx.x;
    for(int s=blockDim.x / 2; s>0; s/=2){
        if(blockThreadId<s){
            d_in[threadId] += d_in[threadId+s];
        }
        __syncthreads();
    }
}
```

Work complexity: O(n)

Step complexity: O(log(n))

Parallel reduce - shared mem

```
__global__ void reduceParallel_sum(float *d_in, float identity)
{
    extern __shared__ float s_data[];

    const unsigned int tid = threadIdx.x;

    s_data[tid] = d_in[tid];
    __syncthreads();

    for(int s=blockDim.x / 2; s>0; s/=2){
        if(tid<s){
            s_data[tid] += s_data[tid+s];
        }
        __syncthreads();
    }

    if(tid==0)
        d_in[0] = s_data[0];
}
```

Scan

Exclusive-Scan

Input: ?

Output: ?

Scan

Exclusive-Scan

Input: List of elements, binary associative operator, and neutral element

Output: List where each element is the reduction of all previous elements (and identity)

```
excl_scan([0102000010406], +, 0) =
          [00113333334488]
```

Scan

Exclusive-Scan

Input: List of elements, binary associative operator, and neutral element

Output: List where each element is the reduction of all previous elements (and identity)

```
excl_scan([01020000010406], +, 0) =
          [00113333334488]
```

Serial scan

```
void excl_sum_scan(float *arr, int sz, float neutral)
{
    float accumulator = neutral;
    for(int i=0;i<sz;i++){
        float tmp = arr[i];
        arr[i] = accumulator;
        accumulator+=tmp;
    }
}
```

Blelloch scan

Blelloch-Scan(arr, op, neutral):

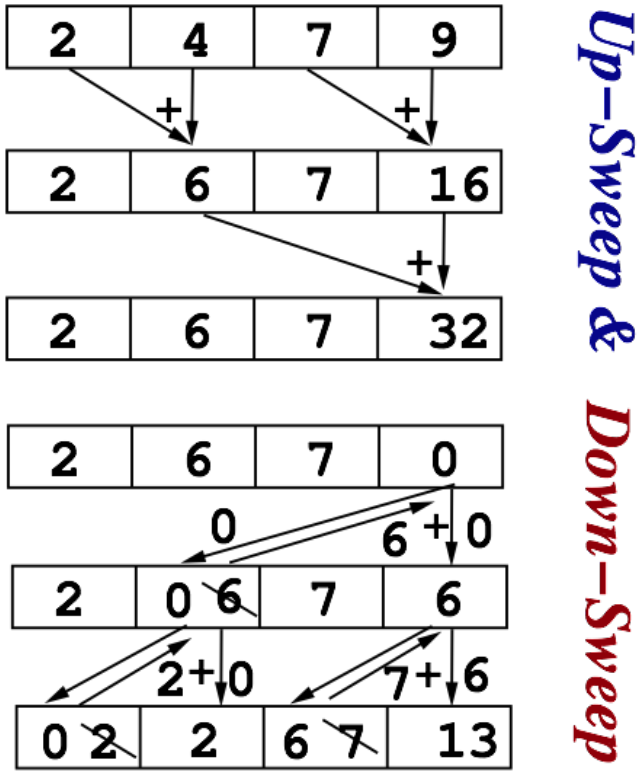
Up-Sweep *//Basically Reduce*

Replace last elem with neutral

Down-Sweep

//Apply ‘Down’ operation on pairs

//’Down’: Input=(L,R), output=(R,L+R)



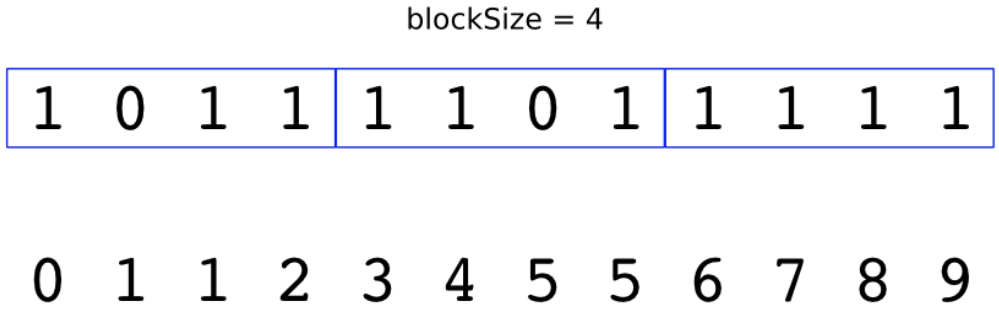
```
//Blelloch exclusive-sum-scan
__global__ void excl_sum_scan(float *d_in, float *d_out, float neutral)
{
    const unsigned int tid = threadIdx.x;
    extern __shared__ float s_arr[];

    //Copy to shared memory
    s_arr[tid] = d_in[tid];

    __syncthreads();

    //Up-sweep
    for(int d=1; d<blockDim.x; d*=2){
        if(  tid%(2*d)==(2*d-1)  ){
            s_arr[tid] = s_arr[tid-d]+s_arr[tid];
        }
        __syncthreads();
    }

    if(threadIdx.x==blockDim.x-1)
        s_arr[tid] = neutral;

    __syncthreads();

    //Down-sweep
    for(int d=blockDim.x/2; d>0; d/=2){
        if(  tid%(2*d)==(2*d-1)  ){
            float tmp = s_arr[tid-d];
            __syncthreads();
            s_arr[tid-d] = s_arr[tid];
            s_arr[tid] = tmp+s_arr[tid];
        }
        __syncthreads();
    }

    //Copy back to global memory
    d_out[tid] = s_arr[tid];
}
```

```cpp
//Blelloch exclusive-sum-scan
__global__ void excl_sum_scan(float *d_in, float *d_out, float neutral)
{
    const unsigned int tid = threadIdx.x;
    extern __shared__ float s_arr[];

    //Copy to shared memory
    s_arr[tid] = d_in[tid];

    __syncthreads();

    //Up-sweep
    for(int d=1; d<blockDim.x; d*=2){
        if(  tid%(2*d)==(2*d-1)  ){
            s_arr[tid] = s_arr[tid-d]+s_arr[tid];
        }
        __syncthreads();
    }

    if(threadIdx.x==blockDim.x-1)
        s_arr[tid] = neutral;

    __syncthreads();

    //Down-sweep
    for(int d=blockDim.x/2; d>0; d/=2){
        if(  tid%(2*d)==(2*d-1)  ){
            float tmp = s_arr[tid-d];
            __syncthreads();
            s_arr[tid-d] = s_arr[tid];
            s_arr[tid] = tmp+s_arr[tid];
        }
        __syncthreads();
    }

    //Copy back to global memory
    d_out[tid] = s_arr[tid];
}
```

```cpp
// execute the kernel
excl_sum_scan<<< 1, num_threads, shared_mem_size >>>(d_in, d_out, 0);
```

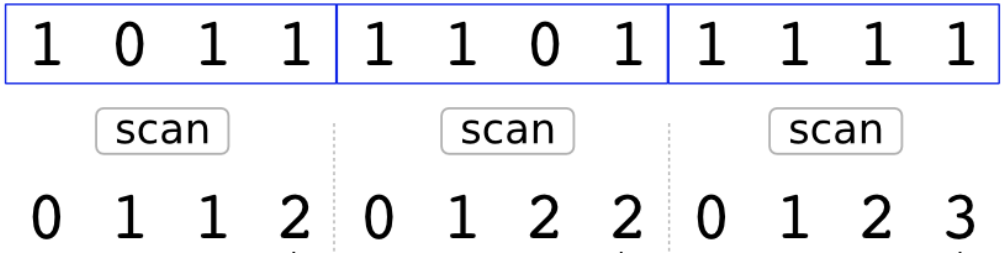
Does not allow for `num_threads>1024`

Exclusive Scan across blocks



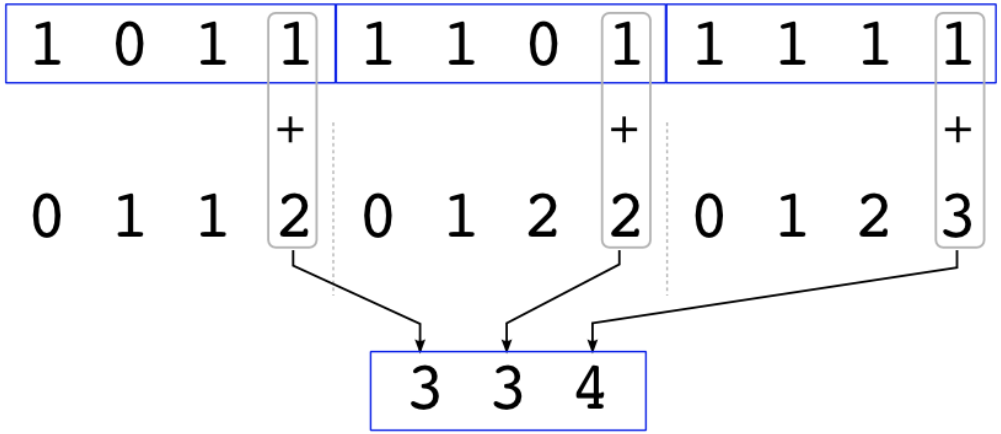
Exclusive Scan across blocks

- Perform ExclScan within each block



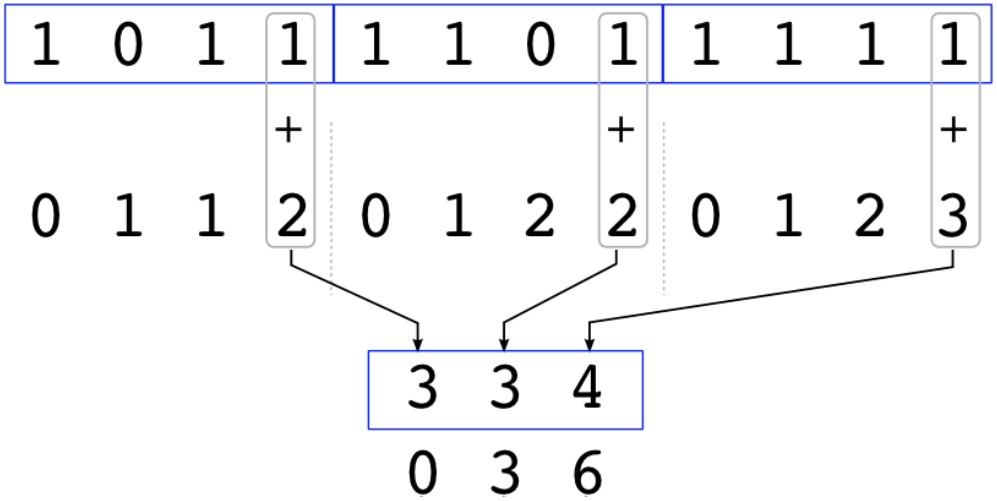
Exclusive Scan across blocks

- Perform ExclScan within each block
- Get list of reductions over each block (last elem)



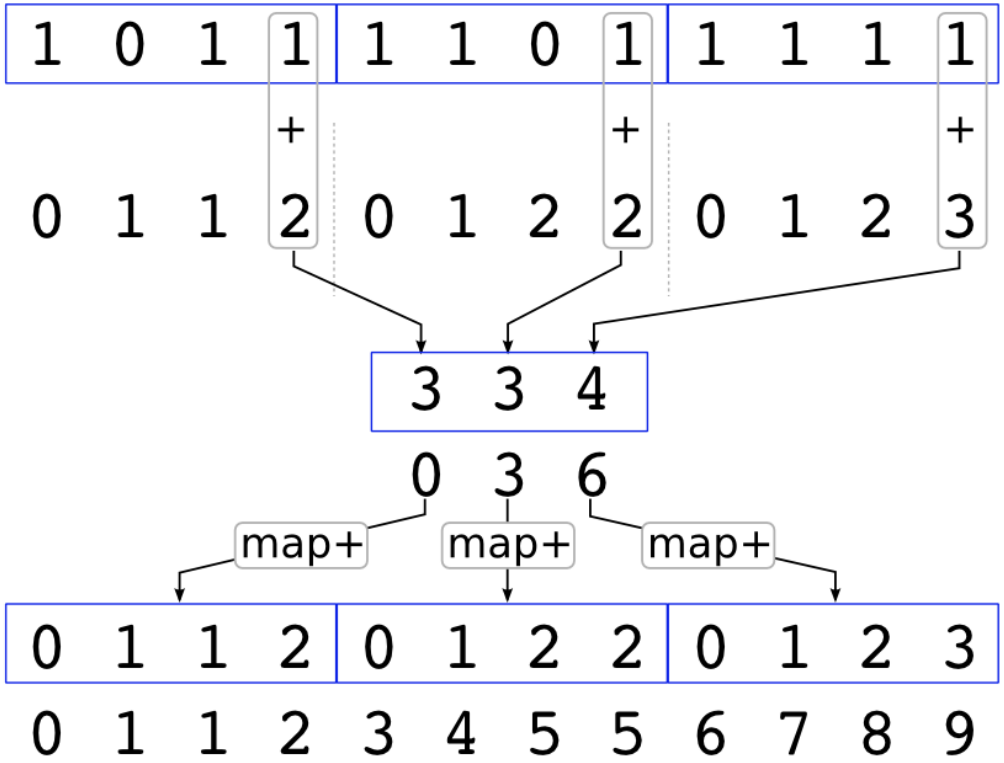
Exclusive Scan across blocks

- Perform ExclScan within each block
- Get list of reductions over each block (last elem)
- ExclScan reductions (recursively)

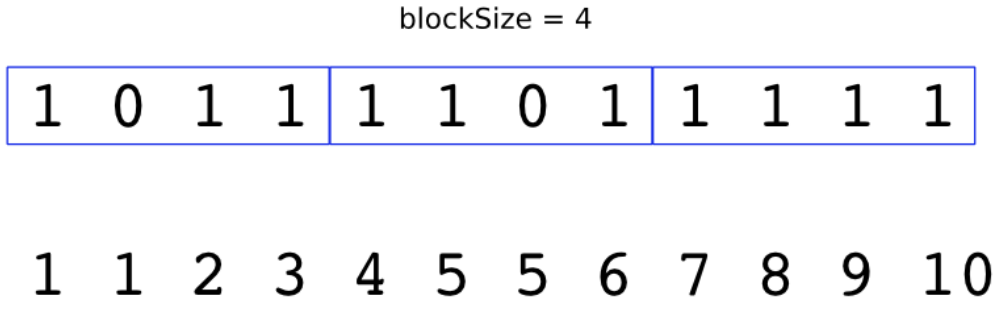


Exclusive Scan across blocks

- Perform ExclScan within each block
- Get list of reductions over each block (last elem)
- ExclScan reductions (recursively)
- Map to scanned blocks



Inclusive Scan across blocks



Inclusive Scan across blocks

- Perform InclScan within each block
- Get list of reductions over each block (last elem)
- ExclScan reductions
- Map to scanned blocks