Synchronization

Synchronization

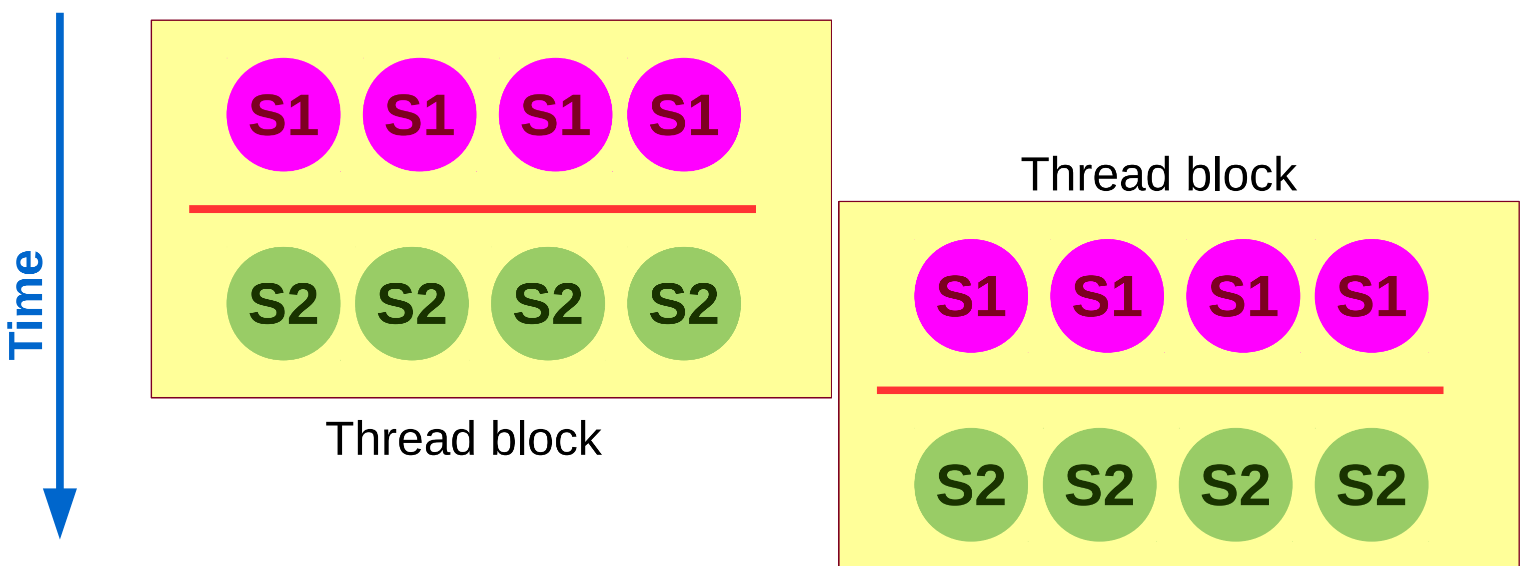
- Control + data flow
- Atomics
- Barriers
- ...

Barriers - Recap

- A barrier is a program point where all threads need to reach before any thread can proceed.
- End of kernel is an implicit barrier for all GPU threads (global barrier).
- ~~There is no explicit global barrier supported in CUDA~~. *grid*.sync() is now supported (from CUDA 9).
- Threads in a thread-block can synchronize using __syncthreads().
- How about barrier within warp-threads?

Barriers

```
__global__ void dkernel(unsigned *vector, unsigned vectorsize) {
    unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
    vector[id] = id;                    S1
    __syncthreads();
    if (id < vectorsize - 1 && vector[id + 1] != id + 1)                    S2
       printf("syncthreads does not work.\n");
}
```



Barriers

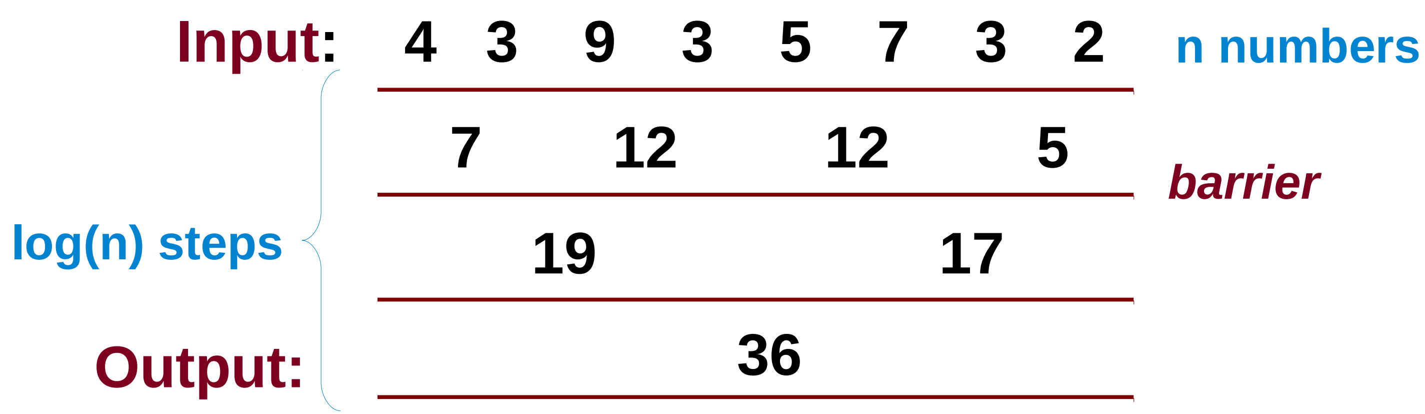
- __syncthreads() is not only about control synchronization, it also has data synchronization mechanism.
- It performs a **memory fence** operation.
  - A memory fence ensures that the writes from a thread are made visible to other threads.
  - __syncthreads() executes a fence for all the block-threads.
- There is a separate __threadfence_block() instruction also. Then, there is __threadfence().

Reductions

- Converting a set of values to few values (typically 1)
- Computation must be *reducible*.
  - Must satisfy associativity property (a.(b.c) = (a.b).c).
  - Min, Max, Sum, XOR, ...
- Can be often implemented using atomics
  - atomicAdd(&sum, a[i]);
  - atomicMin(&min, a[i]);
  - But adds sequentiality.
- Reductions allow improving parallelism.

Reductions

- Converting a set of values to few values (typically 1)
- Computation must be *reducible*.
  - Must satisfy associativity property (a.(b.c) = (a.b).c).
  - Min, Max, Sum, XOR, ...
- Complexity measures

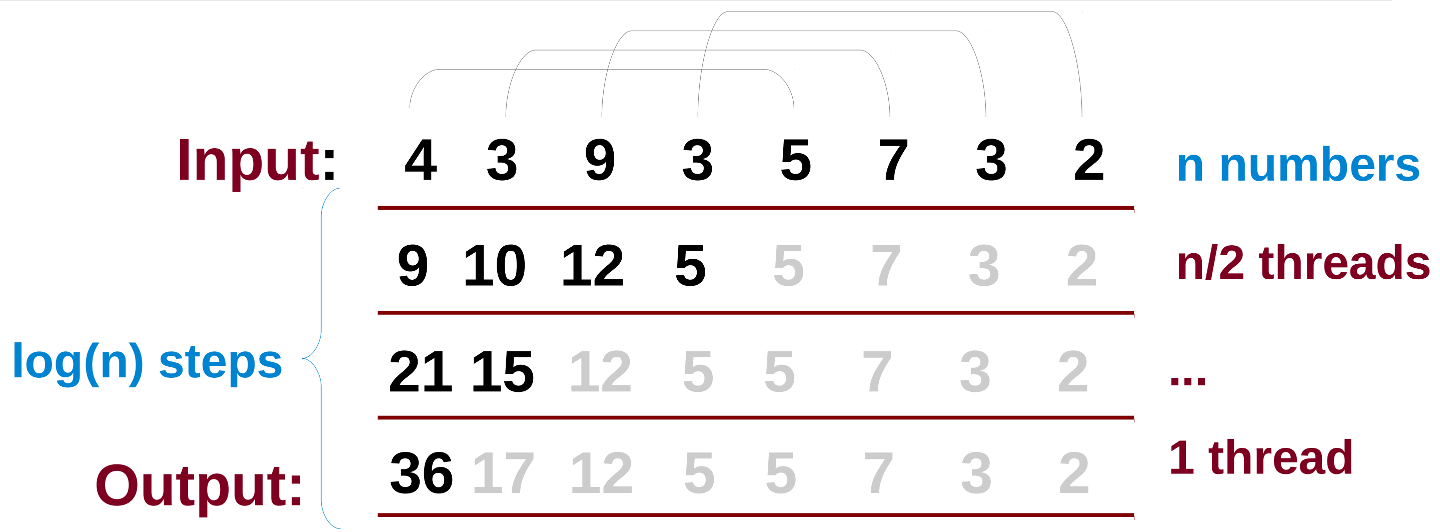


Classwork: Write the reduction code.

Reductions

n must be a power of 2

```
for (int off = n/2; off; off /= 2) {
        if (threadIdx.x < off) {
                a[threadIdx.x] += a[threadIdx.x + off];
        }
        __syncthreads();
}
```



Reductions

```
for (int off = n/2; off; off /= 2) {
    if (threadIdx.x < off) {
        a[threadIdx.x] += a[threadIdx.x + off];
    }
    __syncthreads();
}
```

Write the reduction as: **4 3 9 3 5 7 3 2**

```
for (int off = n/2; off; off /= 2) {
  if (threadIdx.x < off) {
      a[threadIdx.x] += a[2 * off - threadIdx.x - 1];
  }
  __syncthreads();
}
```

Reductions

- Let’s go back to our first diagram.

Input:	**4**	**3**	**9**	**3**	**5**	**7**	**3**	**2**	**n numbers**
	7		**12**		**12**		**5**		***barrier***
log(n) steps		**19**				**17**			
Output:				**36**					

- This can be implemented as

Input:	**4**	**3**	**9**	**3**	**5**	**7**	**3**	**2**	**n numbers**
	7	**12**	**12**	**5**	5	7	3	2	**n/2 threads**
log(n) steps	**19**	**17**	12	5	5	7	3	2	**...**
Output:	**36**	17	12	5	5	7	3	2	**1 thread**

Reductions

- A challenge in the implementation is:
  - a[1] is read by thread 0 and written by thread 1.
  - This is a data-race.
  - Can be resolved by separating R and W.
  - This requires another barrier and a temporary.

Homework: Try this out.

Input:	**4**	**3**	**9**	**3**	**5**	**7**	**3**	**2**	**n numbers**
	7	**12**	**12**	**5**	5	7	3	2	**n/2 threads**
log(n) steps	**19**	**17**	12	5	5	7	3	2	**...**
Output:	**36**	17	12	5	5	7	3	2	**1 thread**

Prefix Sum

- Imagine threads wanting to push work-items to a central worklist.
- Each thread pushes different number of work-items.
- This can be computed using atomics or prefix sum (also called as *scan*).

Input:	**4**	**3**	**9**	**3**	**5**	**7**	**3**	**2**
Output:	**4**	**7**	**16**	**19**	**24**	**31**	**34**	**36**
OR								
Output:	**0**	**4**	**7**	**16**	**19**	**24**	**31**	**34**

Classwork: Write the prefix-sum code.

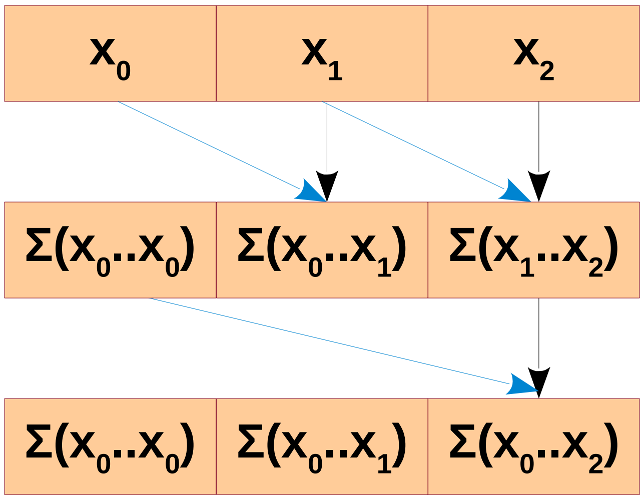
Prefix Sum

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7

$\Sigma(x_0..x_0)$	$\Sigma(x_0..x_1)$	$\Sigma(x_0..x_2)$	$\Sigma(x_0..x_3)$	$\Sigma(x_0..x_4)$	$\Sigma(x_0..x_5)$	$\Sigma(x_0..x_6)$	$\Sigma(x_0..x_7)$

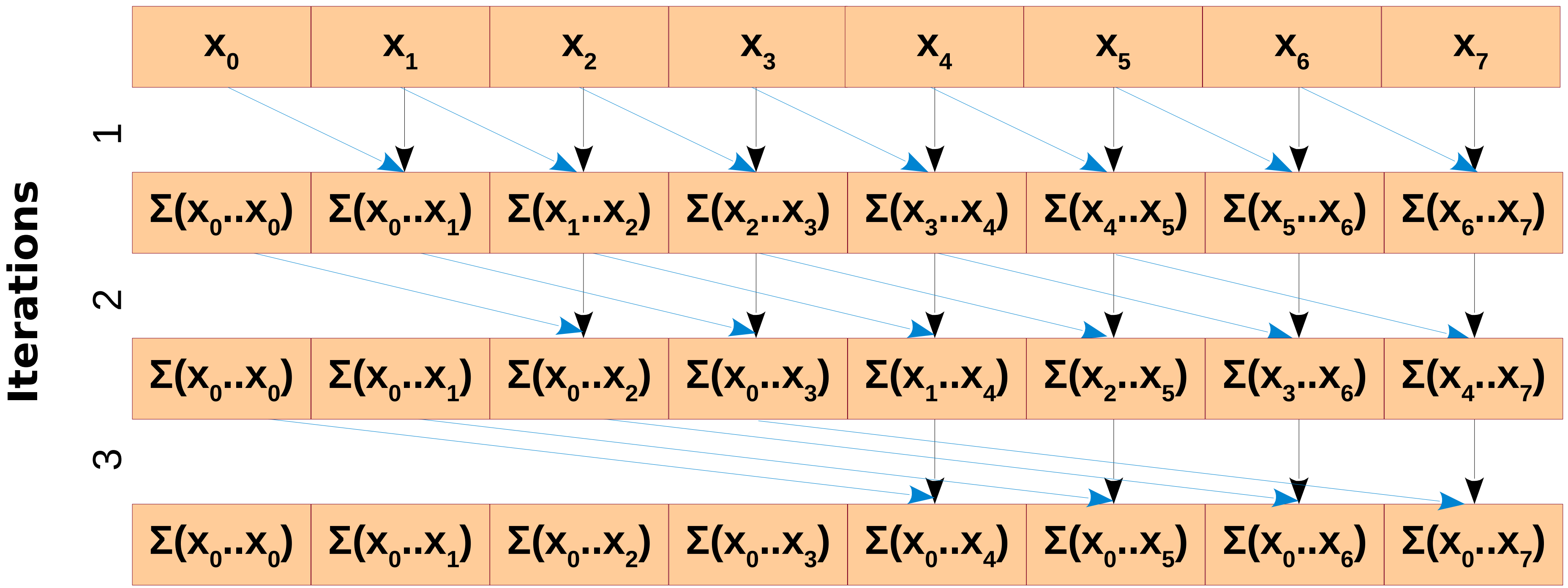
Input: 4 3 9 3 5 7 3 2
Output: 4 7 16 19 24 31 33 35
OR
Output: 0 4 7 16 19 24 31 33

Prefix Sum



Input:	**4**	**3**	**9**	**3**	**5**	**7**	**3**	**2**
Output:	**4**	**7**	**16**	**19**	**24**	**31**	**33**	**35**
OR								
Output:	**0**	**4**	**7**	**16**	**19**	**24**	**31**	**33**

Prefix Sum



Input: 4 3 9 3 5 7 3 2
Output: 4 7 16 19 24 31 33 35
OR
Output: 0 4 7 16 19 24 31 33

Prefix Sum

```
for (int off = 1; off < n; off *= 2) {
    if (threadIdx.x > off) {
        a[threadIdx.x] += a[threadIdx.x - off];
    }
    __syncthreads();
}
```

Datarace

v1

```
for (int off = 1; off < n; off *= 2) {
    if (threadIdx.x > off) {
        tmp = a[threadIdx.x – off];
            __syncthreads();
            a[threadIdx.x] += tmp;
    }
    __syncthreads();
}
```

Separating R and W in time

v2

Prefix Sum

```
for (int off = 1; off < n; off *= 2) {
    if (threadIdx.x >= off) {
        tmp = a[threadIdx.x - off];
    }
    __syncthreads();

    if (threadIdx.x >= off) {
        a[threadIdx.x] += tmp;
    }
    __syncthreads();
}
```

v3

Application of Prefix Sum

- Assuming that you have the prefix sum kernel, insert elements into the worklist.
  - Each thread inserts nelem[tid] many elements.
  - The order of elements is not important.
  - You are forbidden to use atomics.
- Computing cumulative sum
  - Histogramming
  - Area under the curve
  - Fenwick Tree (Binary Indexed Tree)

nelem

Input:	**4**	**3**	**9**	**3**	**5**	**7**	**3**	**2**
Output:	**0**	**4**	**7**	**16**	**19**	**24**	**31**	**33**

Start offset