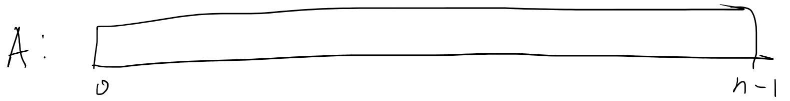
Parallel reduction

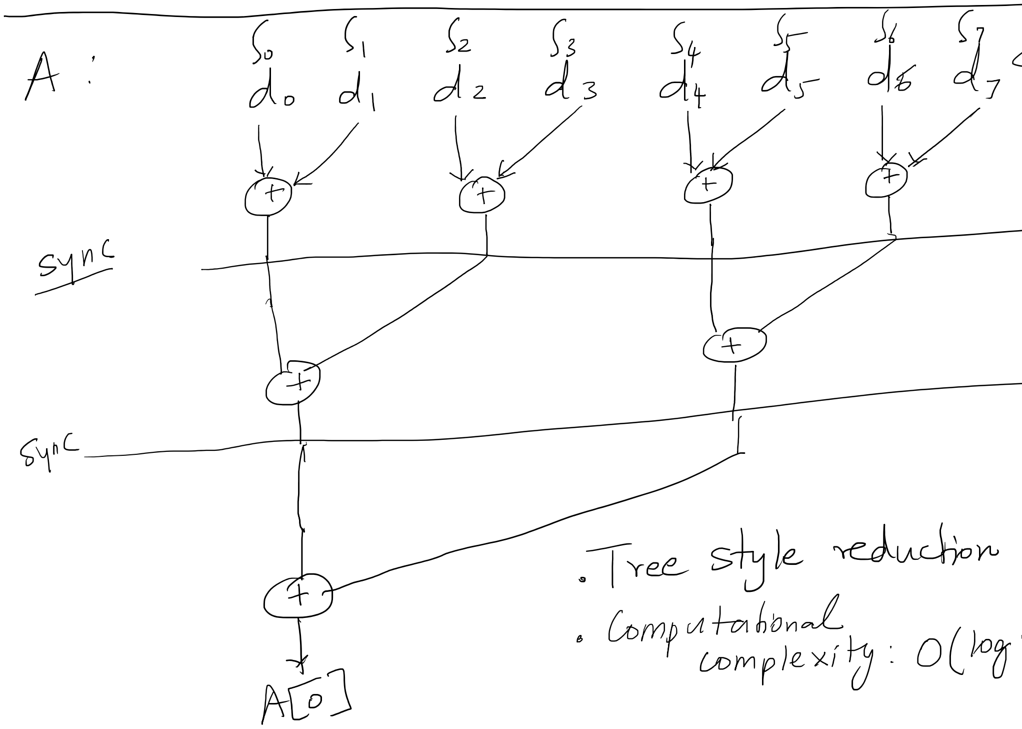
Tuesday, February 23, 2021 8:22 AM

- Code example: vector_reduction_small

$S = \sum_i a_i$



V1


A :
s_0 s_1 s_2 s_3 s_4 s_5 s_6 s_7
d_0 d_1 d_2 d_3 d_4 d_5 d_6 d_7
+
sync
sync
A[0]

loaded elements from global to shared memory

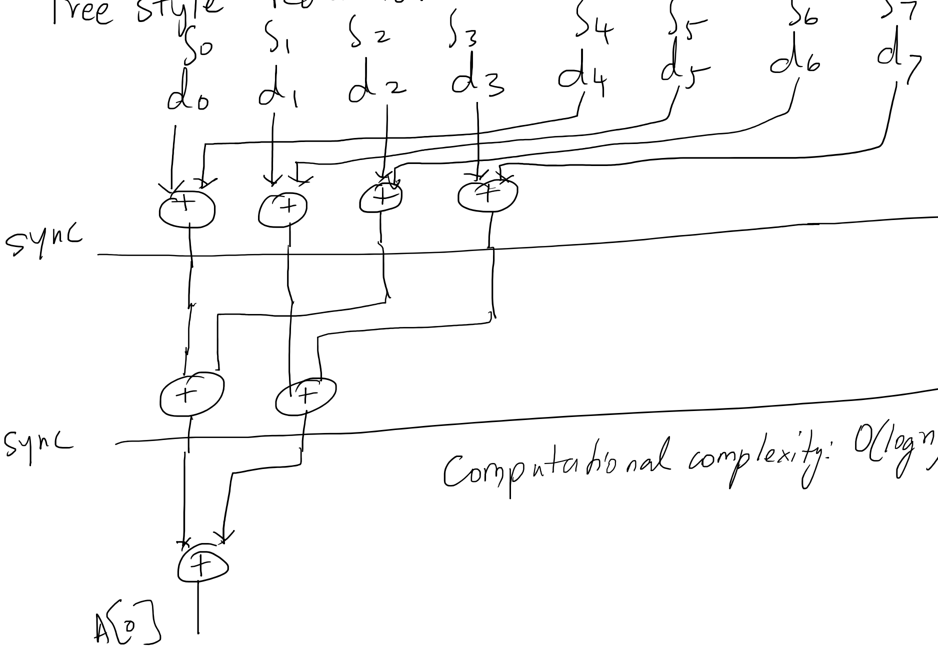
```
stride = 1
if (tid/2*stride == 0)
   A[tid] += A[tid+stride]
stride = 2 * stride = 2
if (tid/2*stride == 0)
   A[tid] += A[tid+stride]
stride = 2 * stride = 4
if (tid/2*stride == 0)
   A[tid] += A[tid+stride]
```

- Tree style reduction
- Computational complexity: O(log n)

Performance issue:

- Branch divergence within warp

V2 Tree style reduction

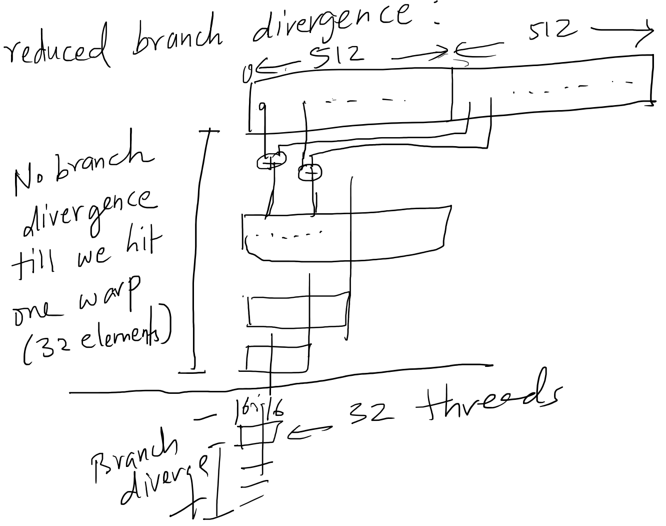

s_0 s_1 s_2 s_3 s_4 s_5 s_6 s_7
d_0 d_1 d_2 d_3 d_4 d_5 d_6 d_7
+
sync
sync
A[0]

```
stride = 8/2 = 4
if (tid < stride)
   A[tid] += A[tid+stride]
stride = stride/2 = 4/2 = 2
if (tid < stride)
   A[tid] += A[tid+stride]
stride = stride/2 = 2/2 = 1
if (tid < stride)
   A[tid] += A[tid+stride]
```

Computational complexity: O(log n)

Version V2 shows

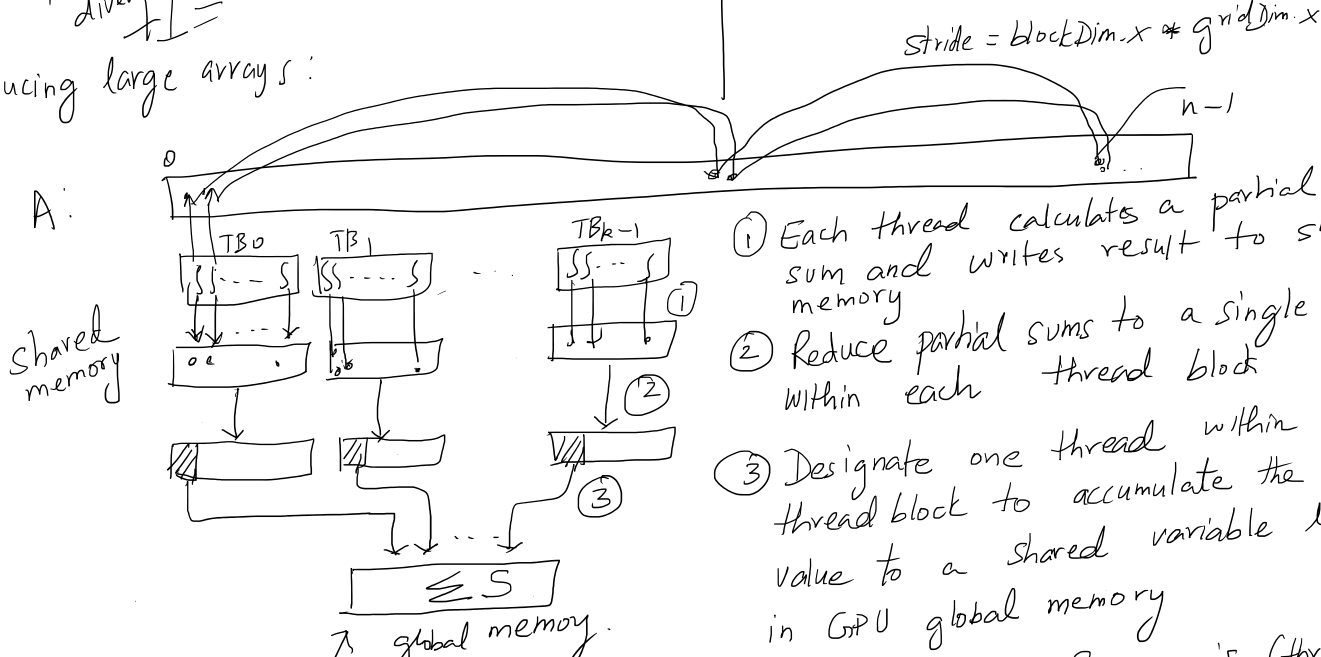
reduced branch divergence:



- # of elements must be power of 2 (since it is tree style)

- What if not power of 2?
  - Pad out shared memory area with zeros to nearest power of 2

Reducing large arrays:



① Each thread calculates a partial sum and writes result to shared memory

② Reduce partial sums to a single value within each thread block

③ Designate one thread within a thread block to accumulate the reduced value to a shared variable living in GPU global memory

Shared variable must be protected via a mutex or use atomic operations supported by NVIDIA's ISA.

```
if (threadIdx.x==0){

    atomicCAS(mutex);
    // accumulate into S
    mutex=0;
}
```

(or)

```
if (threadIdx.x==0)
    atomicAdd(S)
```

Parallel scan

Tuesday, February 23, 2021 8:22 AM

- Code example: scan

$\oplus$: scan operator
ex : addition

I : identity element

Scan:

input : $[a_0 \; a_1 \ldots a_{n-1}]$

Inclusive scan:

output : $[a_0, a_0 \oplus a_1 \;\; a_0 \oplus a_1 \oplus a_2, \ldots \; a_0 \oplus a_1 \oplus \cdots \oplus a_{n-1}]$

Exclusive scan:

output : $[I \;\; a_0 \;\; a_0 \oplus a_1 \;\; a_0 \oplus a_1 \oplus a_2 \cdots a_0 \oplus a_1 \oplus \cdots \oplus a_{n-2}]$

Example: Scan operator = +

Input :	[3 1 7 0 4 1 6 3]
IScan :	[3 4 11 11 15 16 22 25]
EScan :	[0 3 4 11 11 15 16 22]

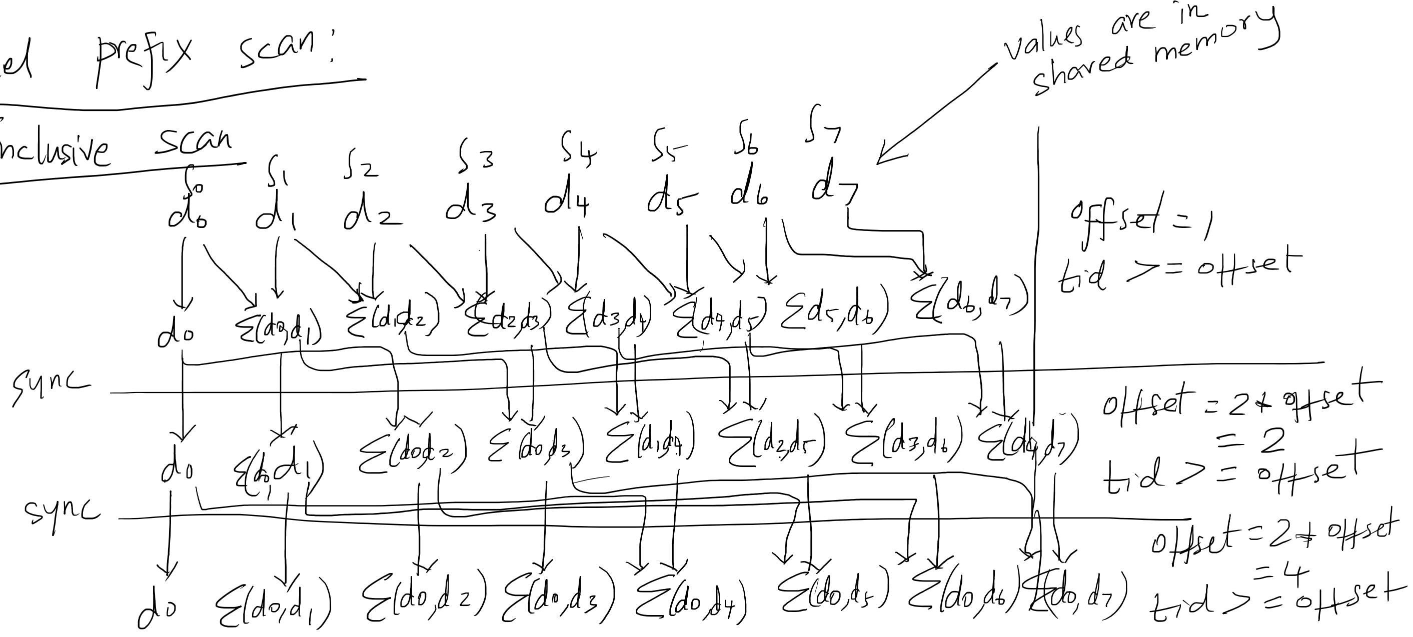
Serial code:

Escan :

```
scan(input, output, n) {
    output[0] = 0;
    for(i = 1; i < n; i++)
        output[i] = output[i-1] + input[i];
}
```

Parallel prefix scan:

Inclusive scan


values are in shared memory
s0 s1 s2 s3 s4 s5 s6 s7
d0 d1 d2 d3 d4 d5 d6 d7
offset = 1
tid >= offset
d0 Σ(d0,d1) Σ(d1,d2) Σ(d2,d3) Σ(d3,d4) Σ(d4,d5) Σ(d5,d6) Σ(d6,d7)
sync
offset = 2 * offset
= 2
tid >= offset
d0 Σ(d0,d1) Σ(d0,d2) Σ(d0,d3) Σ(d1,d4) Σ(d2,d5) Σ(d3,d6) Σ(d4,d7)
sync
offset = 2 * offset
= 4
tid >= offset
d0 Σ(d0,d1) Σ(d0,d2) Σ(d0,d3) Σ(d0,d4) Σ(d0,d5) Σ(d0,d6) Σ(d0,d7)

Use ping-pong buffers to store values between steps to prevent read after write hazards

Recap: atomics on the GPU

Thursday, February 25, 2021 8:14 AM

- Use *atomicCAS()* to achieve mutual exclusion to a critical section of code
  - Do not use *atomicCAS()* as a synchronization mechanism between threads belonging to the same thread warp --> may lead to deadlocks
  - Use *atomicCAS()* as synchronization mechanism between thread blocks
    - Ideally: designated one thread in each thread block, say threadidx.x = 0, to be the writer to the critical section:

```
If (threadIdx.x == 0) {
        Lock()
        /* Critical section code */
        Release()
}
```

- For operations such as addition, subtraction, etc., it is preferrable to use the atomic variants of these operations for better performance:
  - *atomicAdd(), atomicSub()*, …
  - See: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions

Recap: parallel scan

Thursday, February 25, 2021 8:23 AM

Scan : +

Input : [3 1 7 0 4 1 6 3]

iScan : [3 4 11 11 15 16 22 25]

eScan : [0 3 4 11 11 15 16 22]

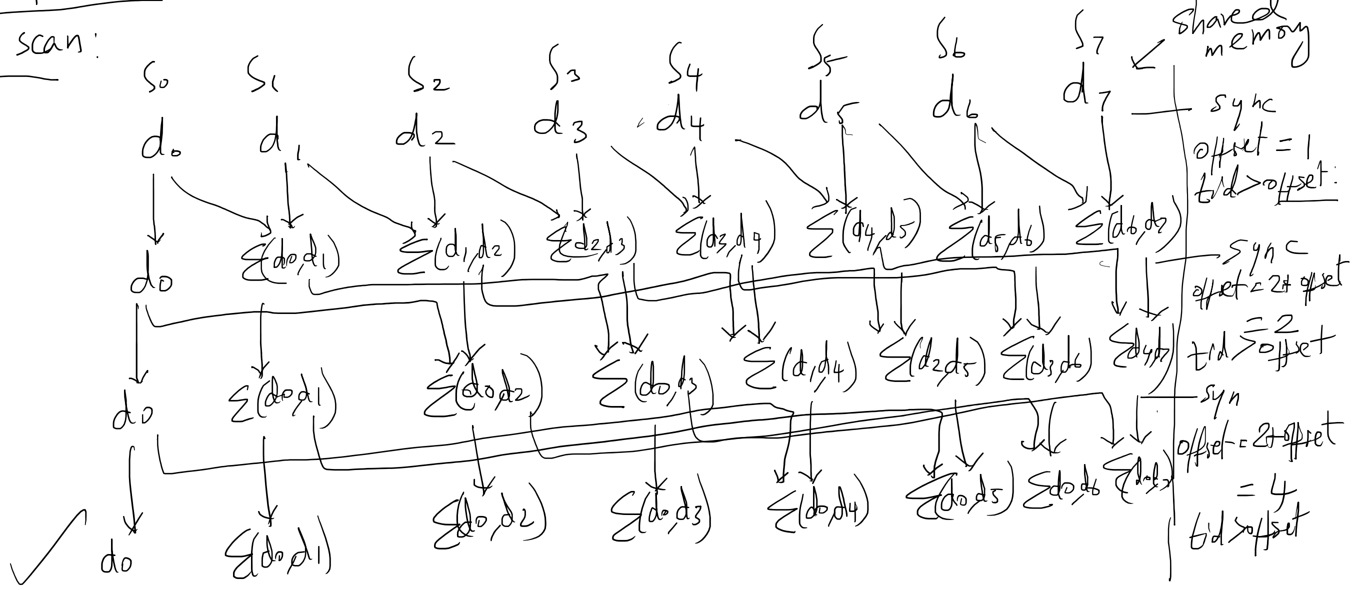
eScan :

```
output[0] = 0
for (i = 1; i < n; i++)
    output[i] = output[i-1] + input[i-1];
```

↑ loop carried dependency

Parallel prefix scan:

- Inclusive scan:

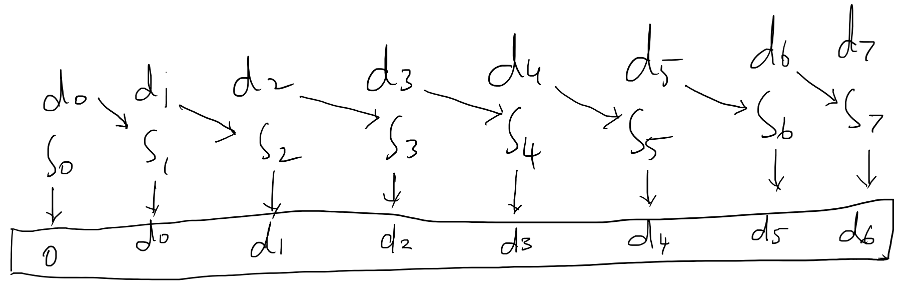


shared memory
s0 s1 s2 s3 s4 s5 s6 s7
d0 d1 d2 d3 d4 d5 d6 d7
sync
offset = 1
tid>offset
d0 Σ(d0,d1) Σ(d1,d2) Σ(d2,d3) Σ(d3,d4) Σ(d4,d5) Σ(d5,d6) Σ(d6,d7)
sync
offset = 2* offset
= 2
tid>offset
d0 Σ(d0,d1) Σ(d0,d2) Σ(d0,d3) Σ(d1,d4) Σ(d2,d5) Σ(d3,d6) Σ(d4,d7)
syn
offset = 2* offset
= 4
tid>offset
d0 Σ(d0,d1) Σ(d0,d2) Σ(d0,d3) Σ(d0,d4) Σ(d0,d5) Σ(d0,d6) Σ(d0,d7)
✓

- Exclusive scan:

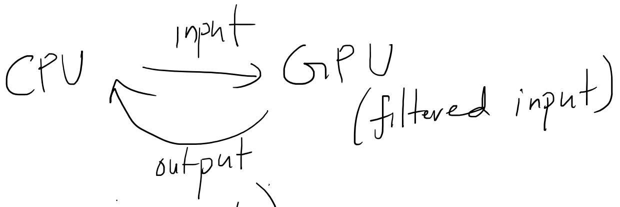
- Use same technique as above

- Threads load data into shared memory as ↗



Stream compaction

Thursday, February 25, 2021 8:23 AM



filter:
only +ve values
>0
$<= 0$ X

size(output) $\leq$ size(input)

Input: 1 −1 3 4 −6 5 −8 10
Output: 1 3 4 5 10

	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
input	1	−1	3	4	−6	5	−8	10
flag	1	0	1	1	0	1	0	1
escan(flag)	0	1	1	2	3	3	4	4

↑ locations in the output array for each thread.

Output:	1	3	4	5	10
	0	1	2	3	4

elements = 5

Counting sort

Thursday, February 25, 2021 8:54 AM

Counting sort : $D : \{d_0, d_1, \dots d_{n-1}\}$ $d_i \in$ Range

↑ integers ↖ for example, $[0, 255]$

- Non-comparison based
- Complexity : $O(n)$

input : 8 5 1 3 7 8 6 5 3 8

bin :	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8
	bin0	bin1	bin2	bin3	bin4	bin5	bin6	bin7	bin8
	0	1	0	2	0	2	1	1	3
escan(bin)	0	0	1	1	3	3	5	6	7

] create histogram

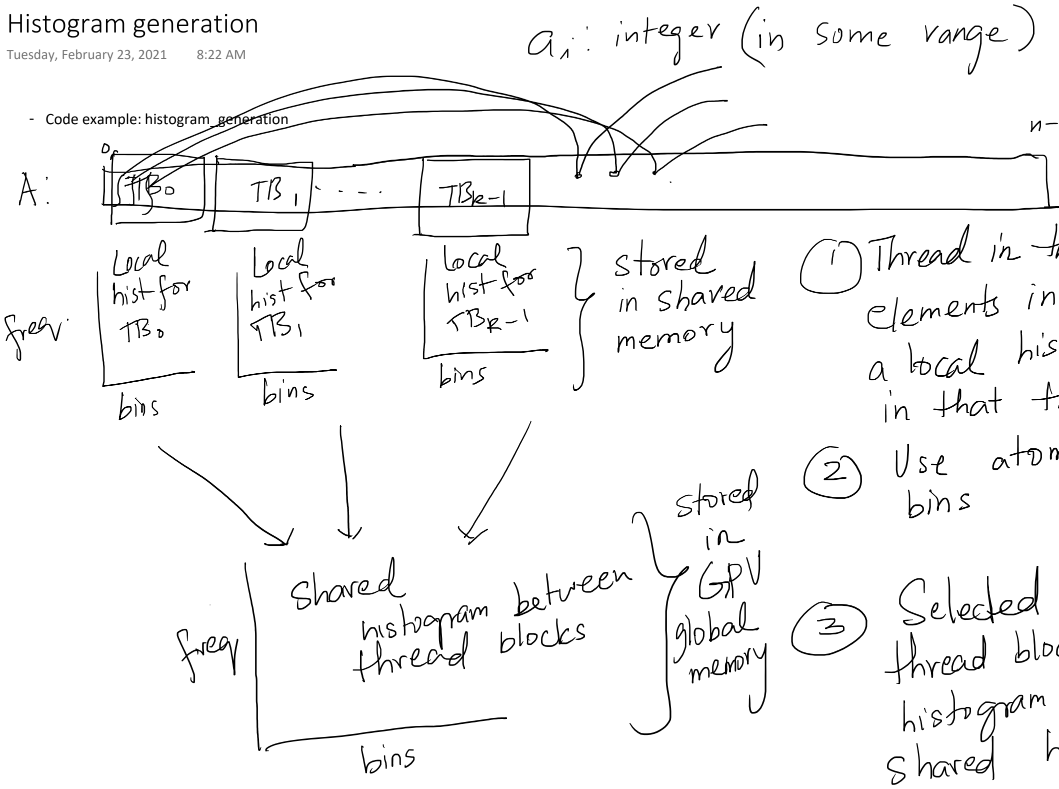
← generate starting address for each thread in sorted array

output	1	3	3	5	5	6	7	8	8	8
	0	1	2	3	4	5	6	7	8	9

Histogram generation

Tuesday, February 23, 2021 8:22 AM

- Code example: histogram_generation



1. Thread in thread block strides across elements in A and bins each in a local histogram shared by threads in that thread block
2. Use atomics when incrementing bins
3. Selected threads within each thread block accumulate local histogram bin values into the shared histogram in GPU global memory
4. Use atomics when accumulating bin values in the shared histogram