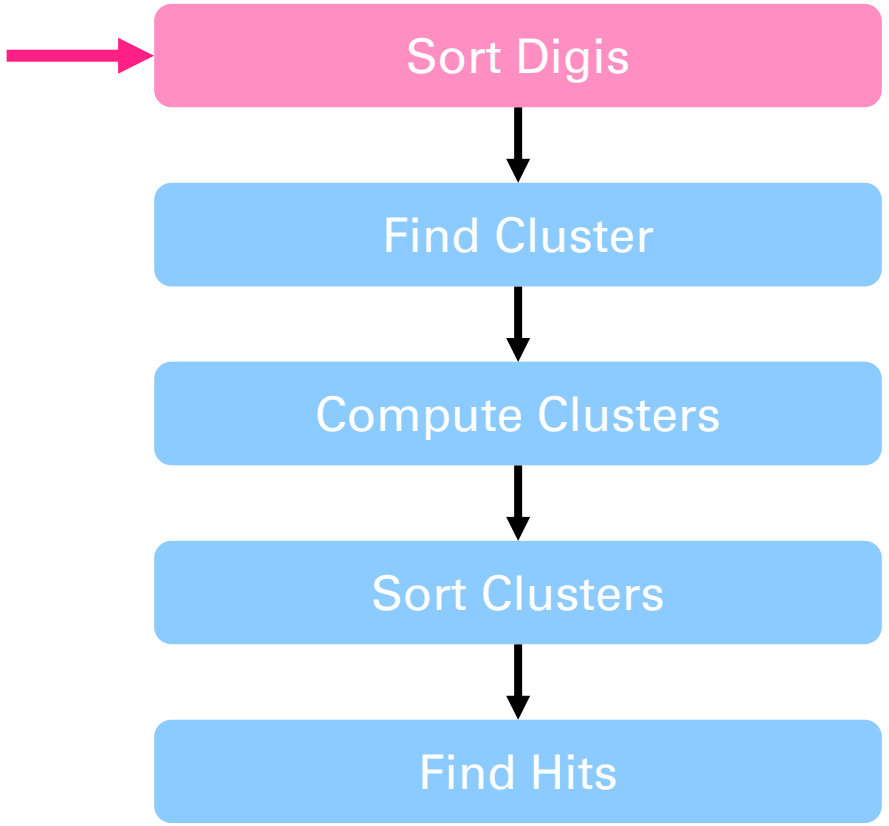
EFFICIENT STS-DIGI TIME SORTING

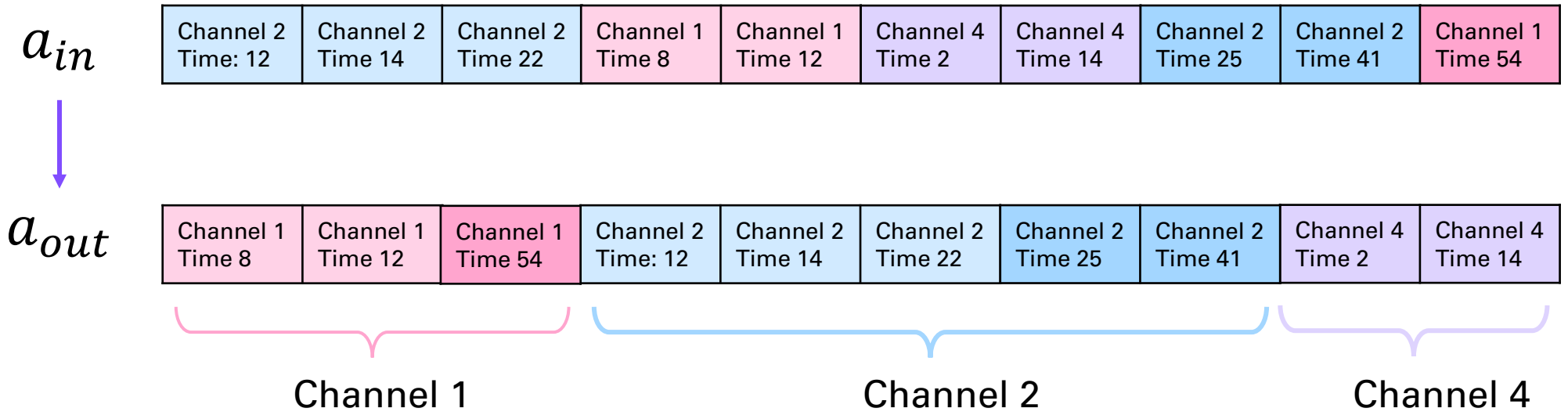
Saman Sedighi Rad
Masterstudent

We are here right now



A variation of Mergesort used on GPU, with running time of $\sim\Theta(n \log n) + \Theta(n)$ additional temporary memory

See: https://github.com/fweig/xpu/blob/master/src/xpu/driver/hip_cuda/device.h




```
digis = digiSort.sort(digis, nDigis, digisTmp, [](const CbmStsDigi a) {
   return ((unsigned long int) a.fChannel) << 32 | (unsigned long int) (a.fTime);
});
```

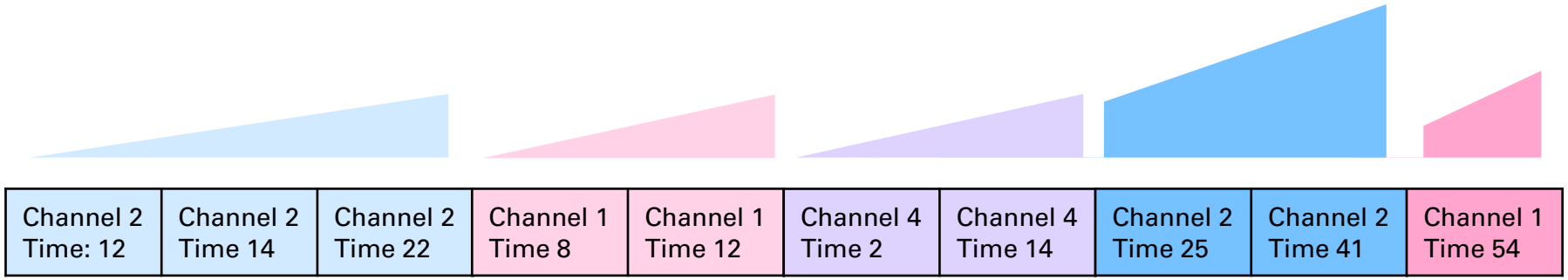
See, Felix merge request:
https://git.cbm.gsi.de/fweig/cbmroot/-/blob/gpu-sts-reco-mr/reco/detectors/sts/experimental/CbmStsGpuHitFinder.cxx#L25

Approach: Adaptive sorting algorithm

- Tailored to the specific data, not for arbitrary inputs
- Incorporating two information for this approach:
    1. Timeslices consist of monotonically time-sorted sub-sequences (Jan)
    2. Each Digi has already the information in which channel it belongs (Sergey)

Premise:

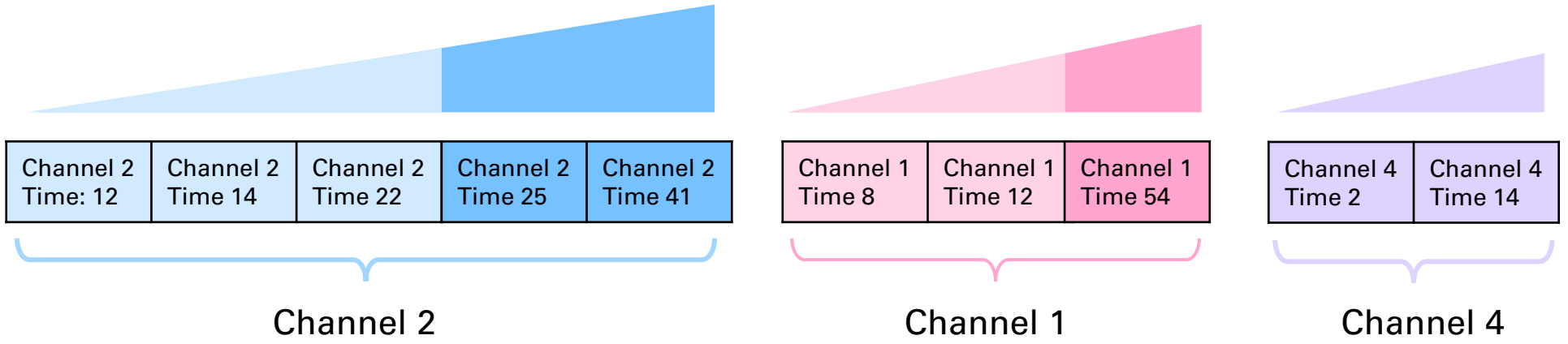
Each timeslice consist of monotonically time-sorted sub-sequences



Timeslice

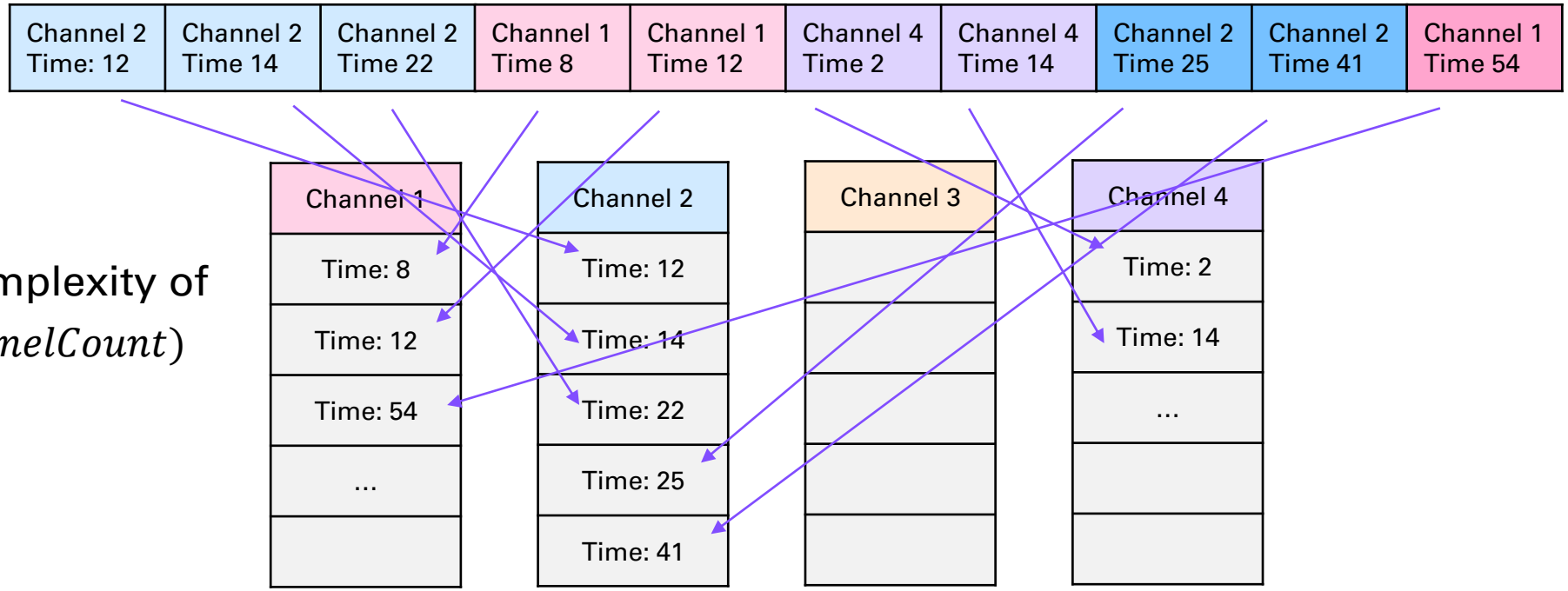
Premise:

Concatenating channels, results again in a monotonic sequence



Sergeys Hint:

Digi already contain the Channel number!



Space complexity of

$O(n * channelCount)$

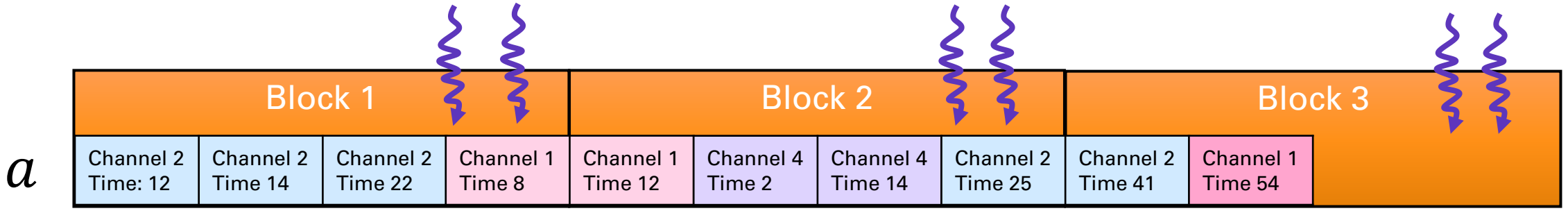
Semi-Parallel solution:

1. Step: Count how many elements each channel contains in $O(n/p + k)$ k=channelCount

```
…
const int channelSize = channelCount * sizeof(int);
hipMalloc((void**)&digisPerChannel, channelSize);
hipMemset(digisPerChannel, 0, channelSize);
countKernel<<<…,…>>>(a, digisPerChannel, n);

// … copy result to host
```

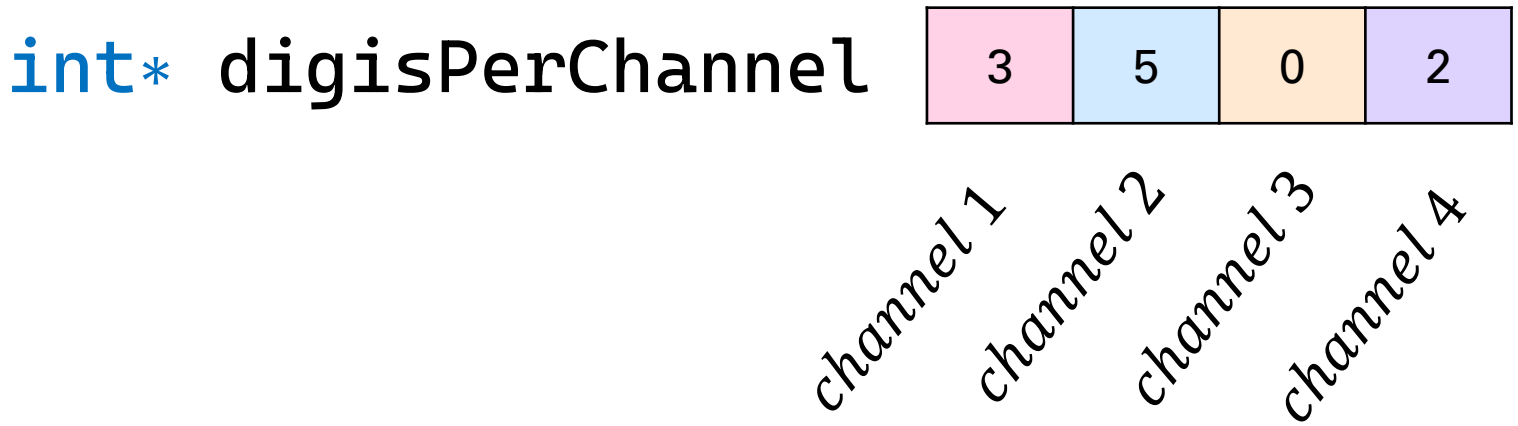
`int* digisPerChannel`

0	0	0	0
0	1	2	3

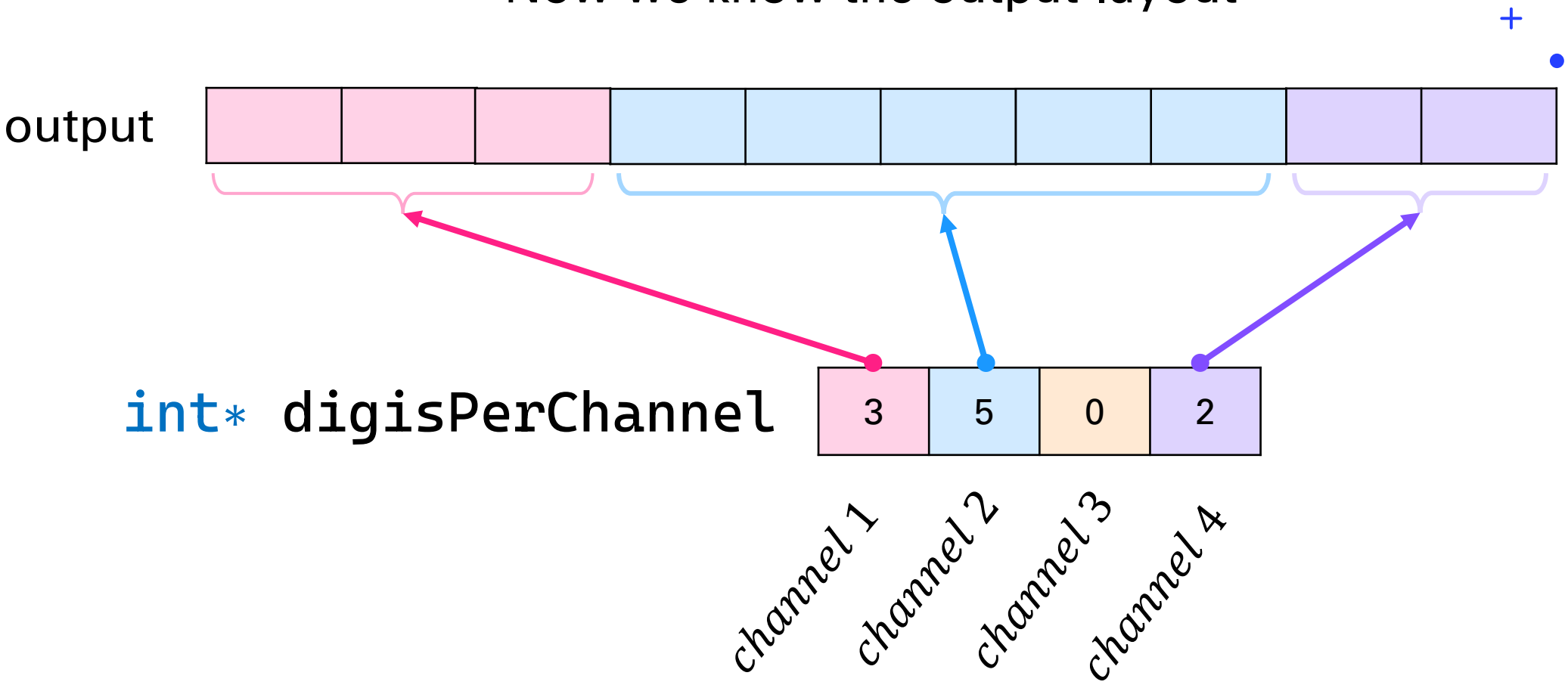



```
__global__ void countKernel(const StsDigit* a, int* digisPerChannel) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    atomicAdd(&digisPerChannel[a[i].channel], 1);
}
```

Only illustrator, n digis counted by n threads



Now we know the output-layout



2. Step

Calculate the prefix sums to get the start index of each sequence

$\theta(n/p)$ `int* digisPerChannel`

3	5	0	2

$\theta(k)$ `int* prefixSum`

0	3	8	8

output

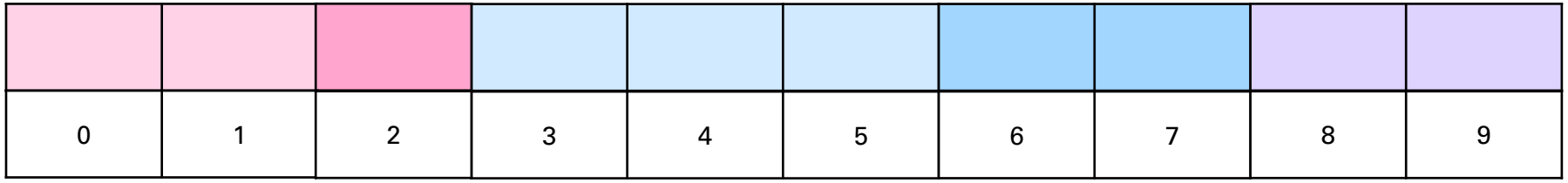
0	1	2	3	4	5	6	7	8	9

3. Step

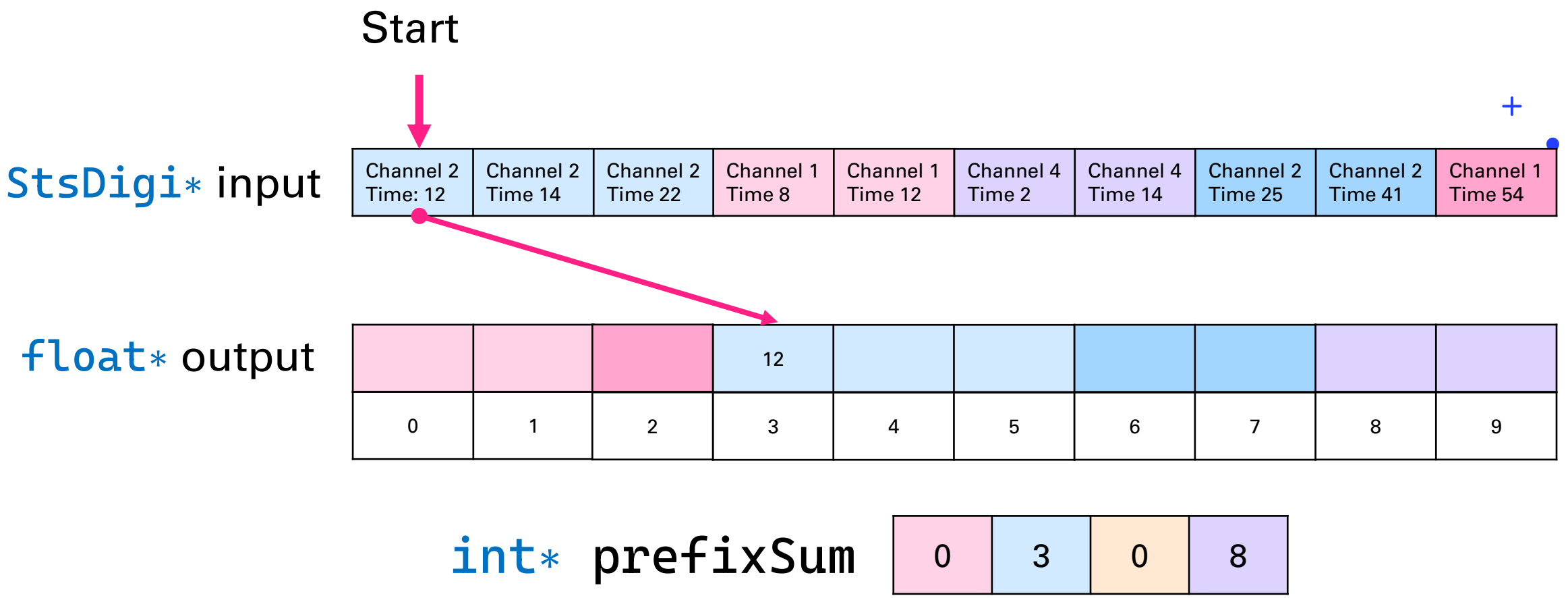
Start

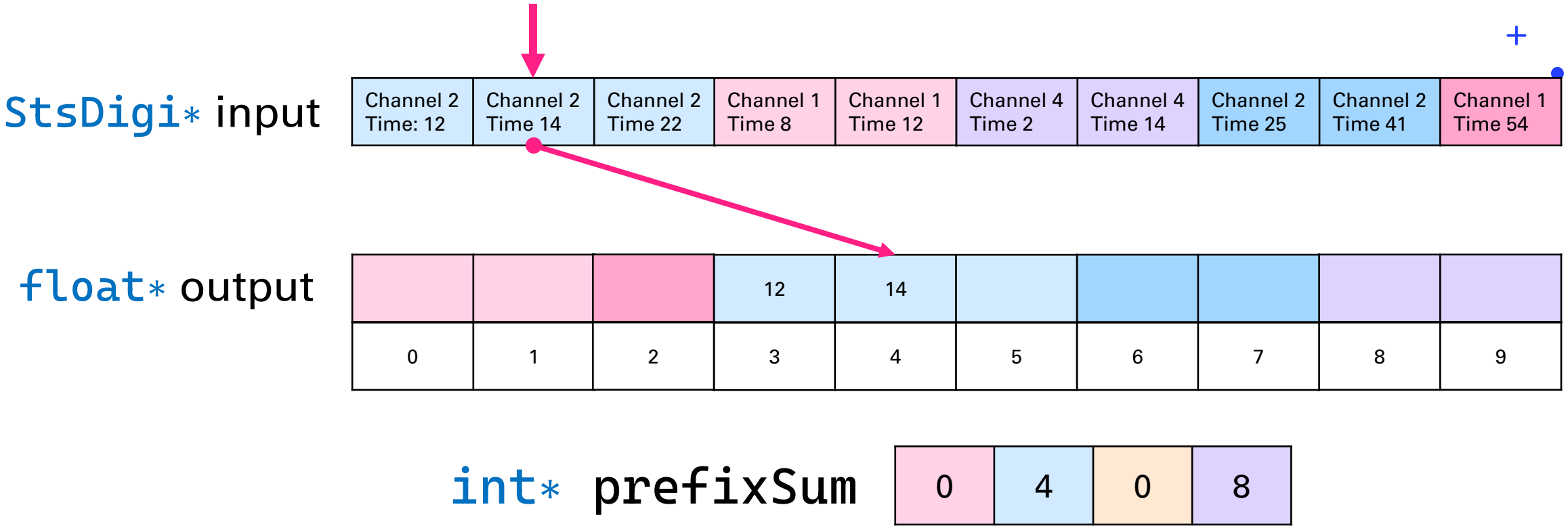
`StsDigi*` input

`float*` output

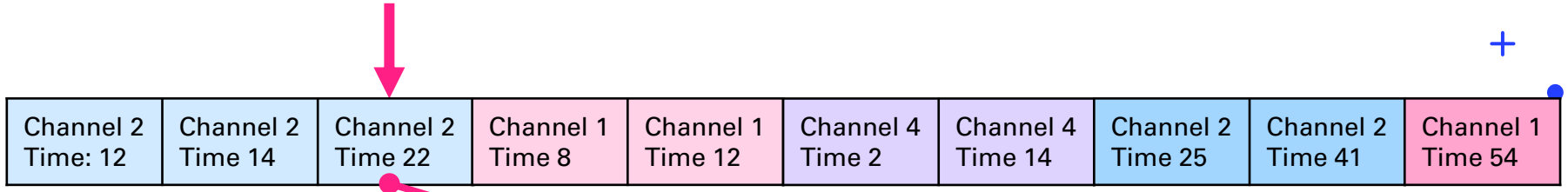


`int* prefixSum`

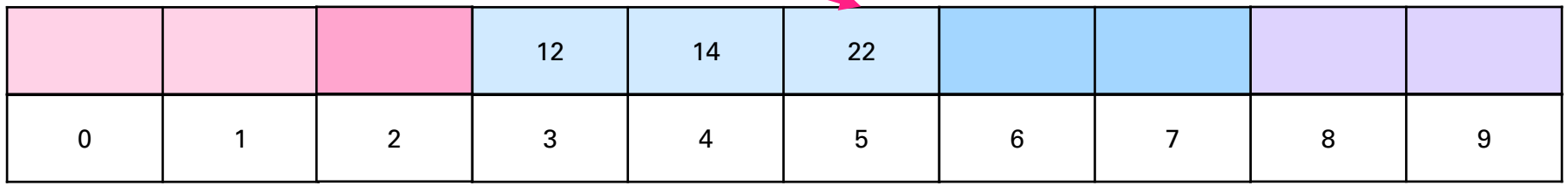

Start
+
StsDigi* input
Channel 2 Time: 12
Channel 2 Time 14
Channel 2 Time 22
Channel 1 Time 8
Channel 1 Time 12
Channel 4 Time 2
Channel 4 Time 14
Channel 2 Time 25
Channel 2 Time 41
Channel 1 Time 54
float* output
12
0
1
2
3
4
5
6
7
8
9
int* prefixSum
0
3
0
8


StsDigi* input
Channel 2 Time: 12
Channel 2 Time 14
Channel 2 Time 22
Channel 1 Time 8
Channel 1 Time 12
Channel 4 Time 2
Channel 4 Time 14
Channel 2 Time 25
Channel 2 Time 41
Channel 1 Time 54
+
float* output
12
14
0
1
2
3
4
5
6
7
8
9
int* prefixSum
0
4
0
8

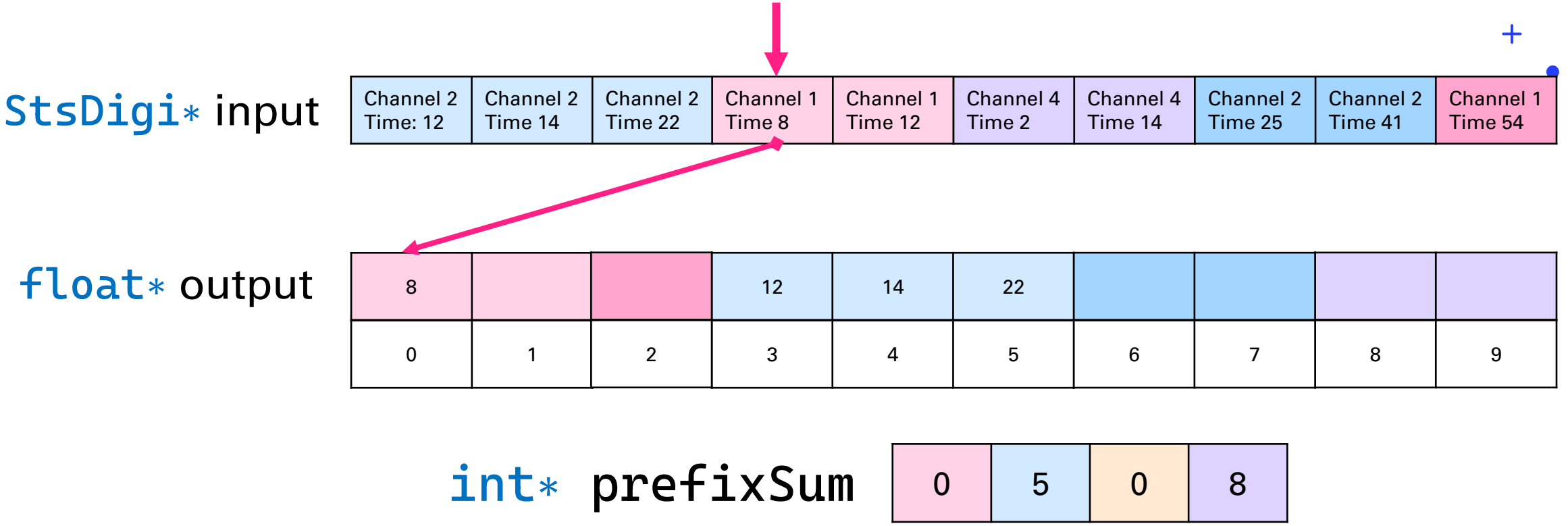
`StsDigi*` input



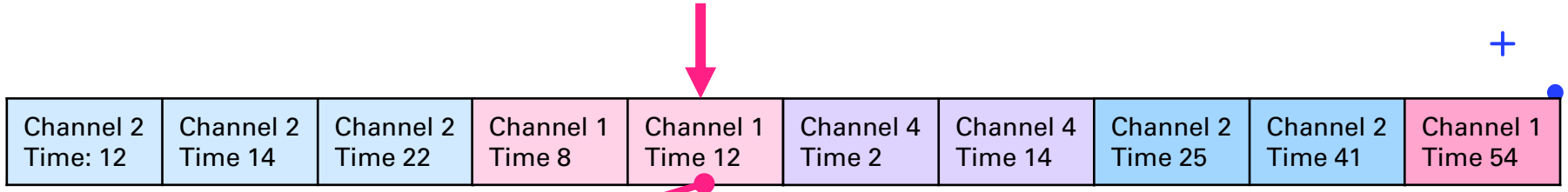
`float*` output



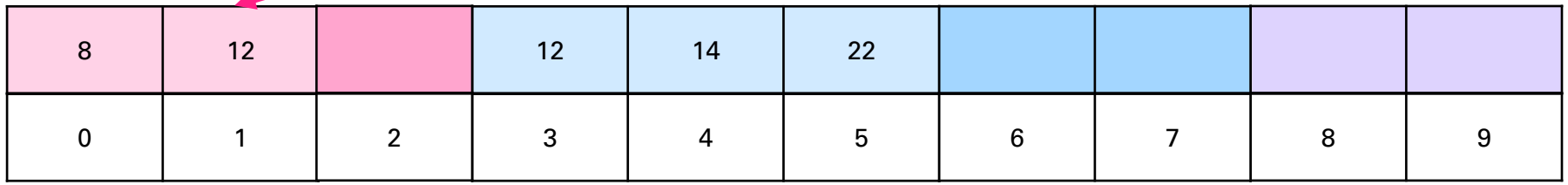
`int* prefixSum`


StsDigi* input
Channel 2 Time: 12
Channel 2 Time 14
Channel 2 Time 22
Channel 1 Time 8
Channel 1 Time 12
Channel 4 Time 2
Channel 4 Time 14
Channel 2 Time 25
Channel 2 Time 41
Channel 1 Time 54
+
float* output
8
12
14
22
0
1
2
3
4
5
6
7
8
9
int* prefixSum
0
5
0
8

`StsDigi*` input

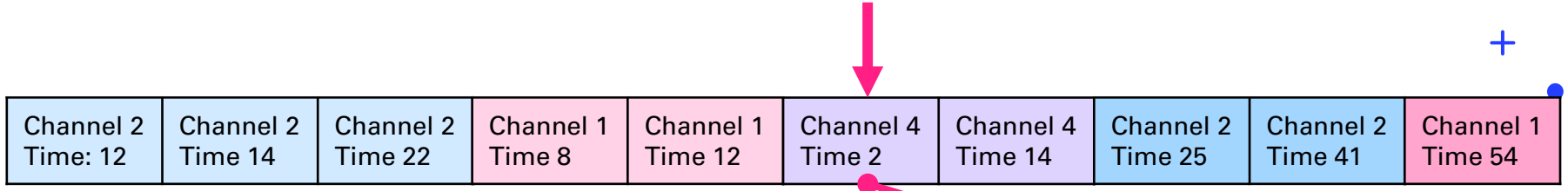


`float*` output

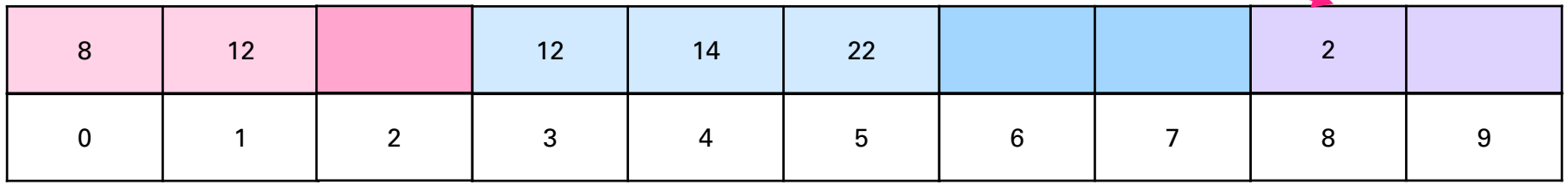


`int* prefixSum`

`StsDigi*` input



`float*` output

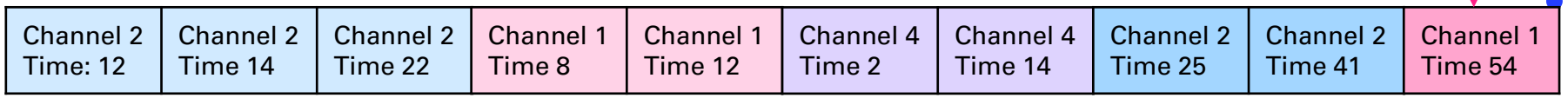


`int* prefixSum`

Done in $\theta(n)$ total in $O(n+k)$

Finished

`StsDigi*` input



`float*` output

8	12	54	12	14	22	25	41	2	14
0	1	2	3	4	5	6	7	8	9

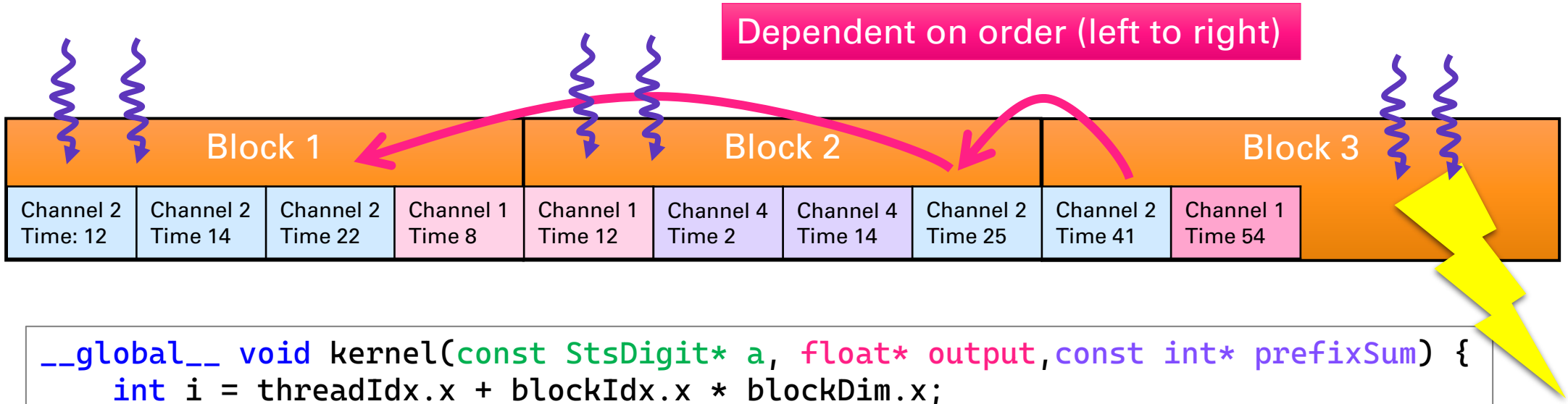
`int* prefixSum`

2	7	0	9

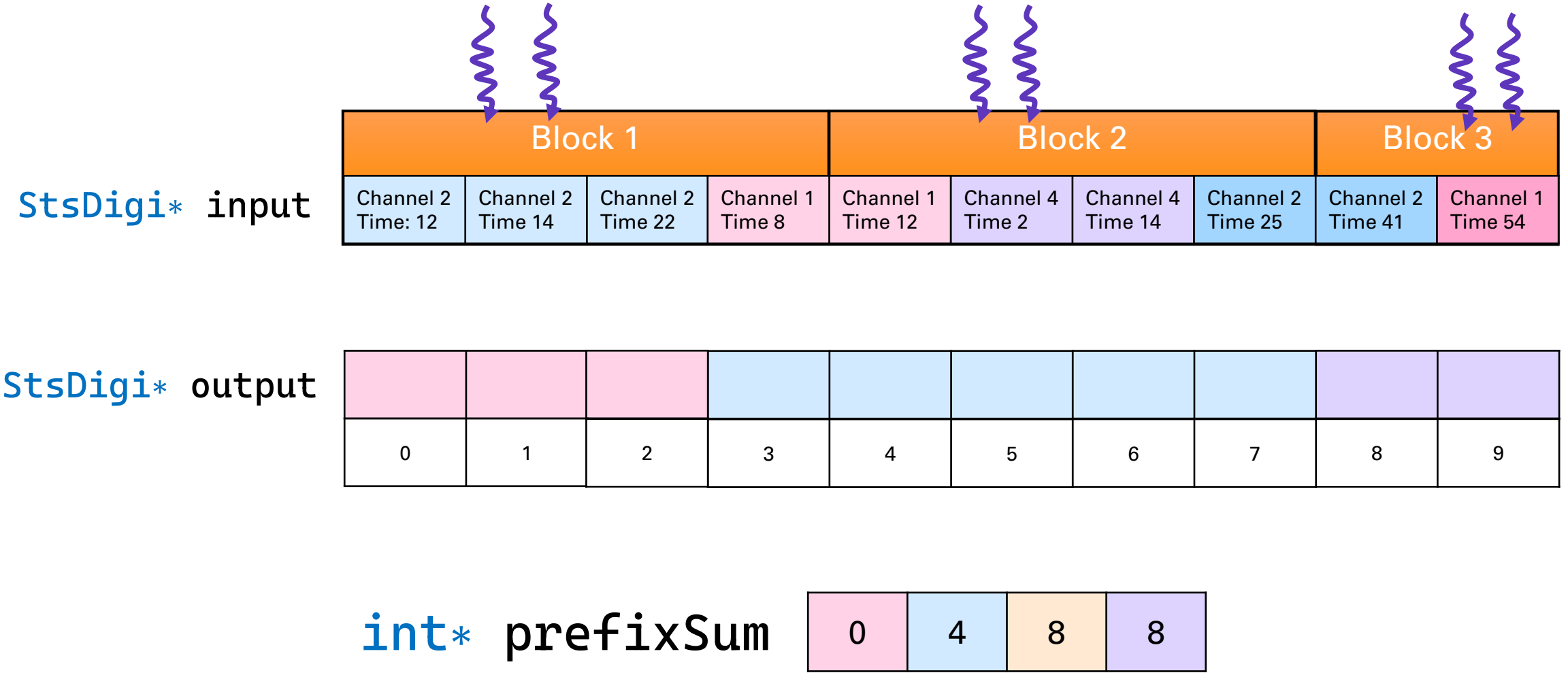
(end index)

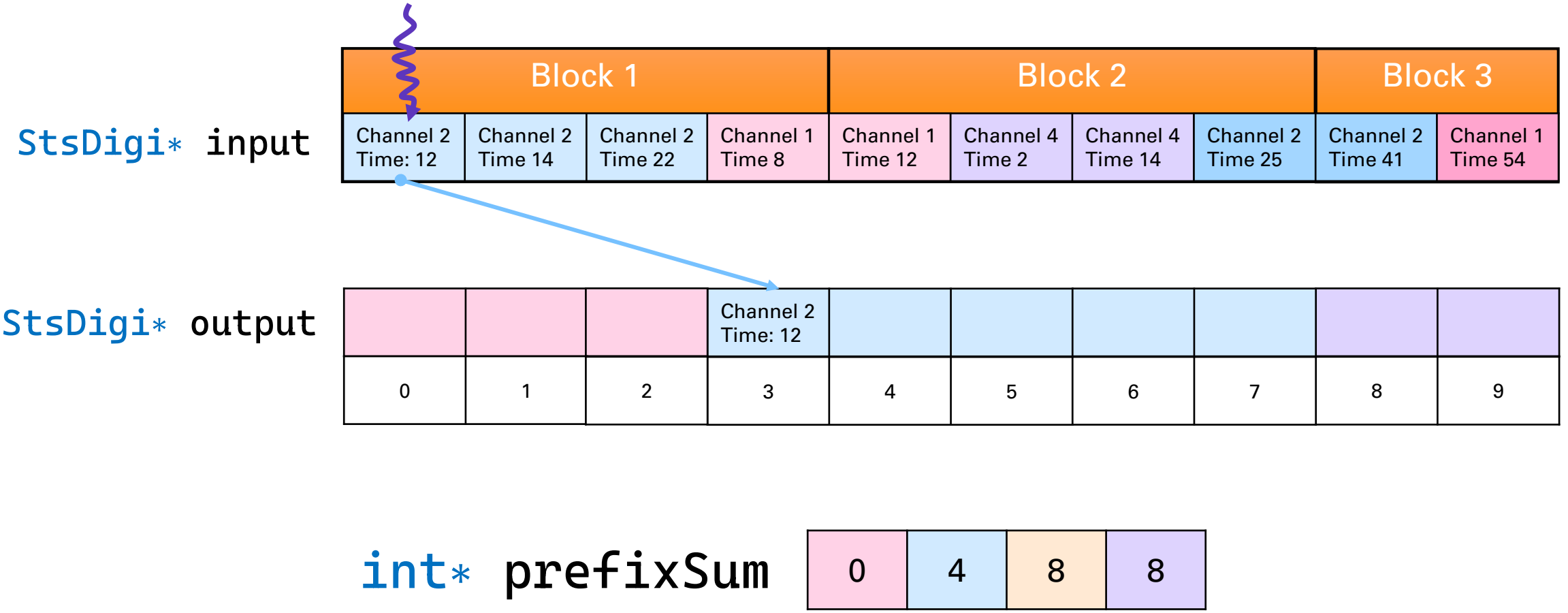
Parallel in $O(n/p + k)$?

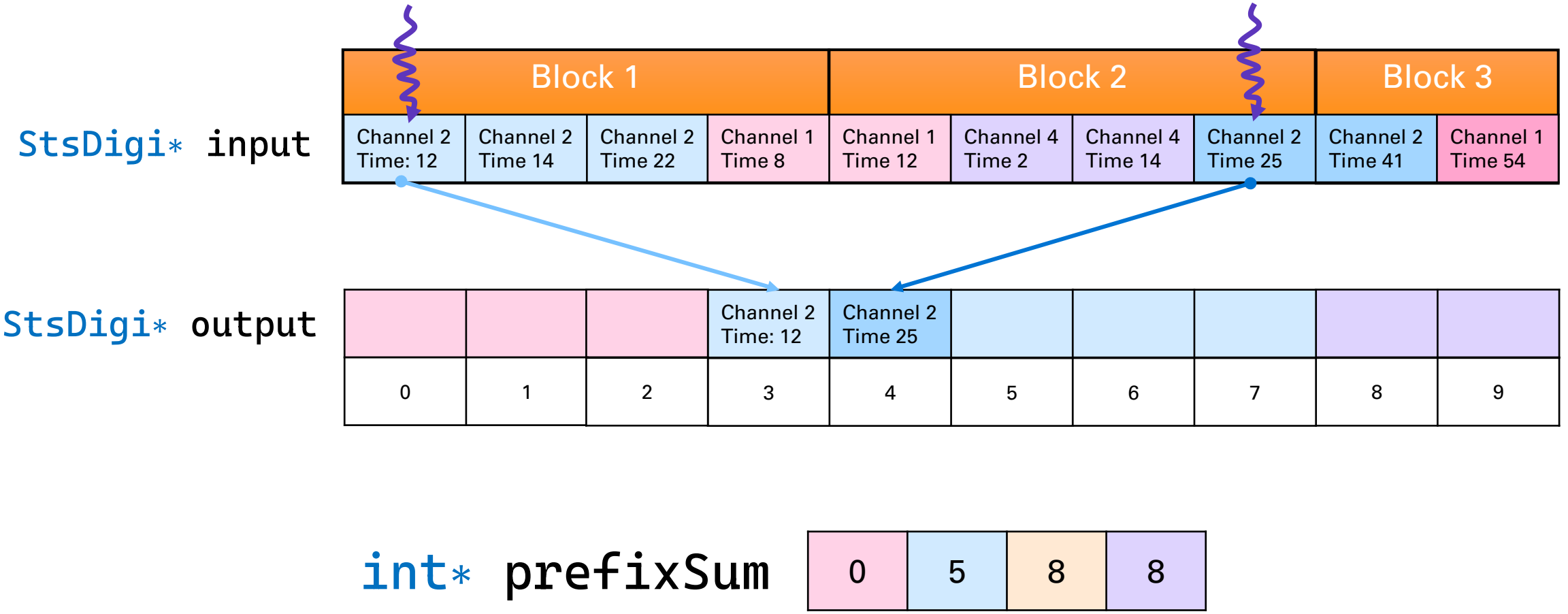
Problem: two or more threads want to insert into the same channel

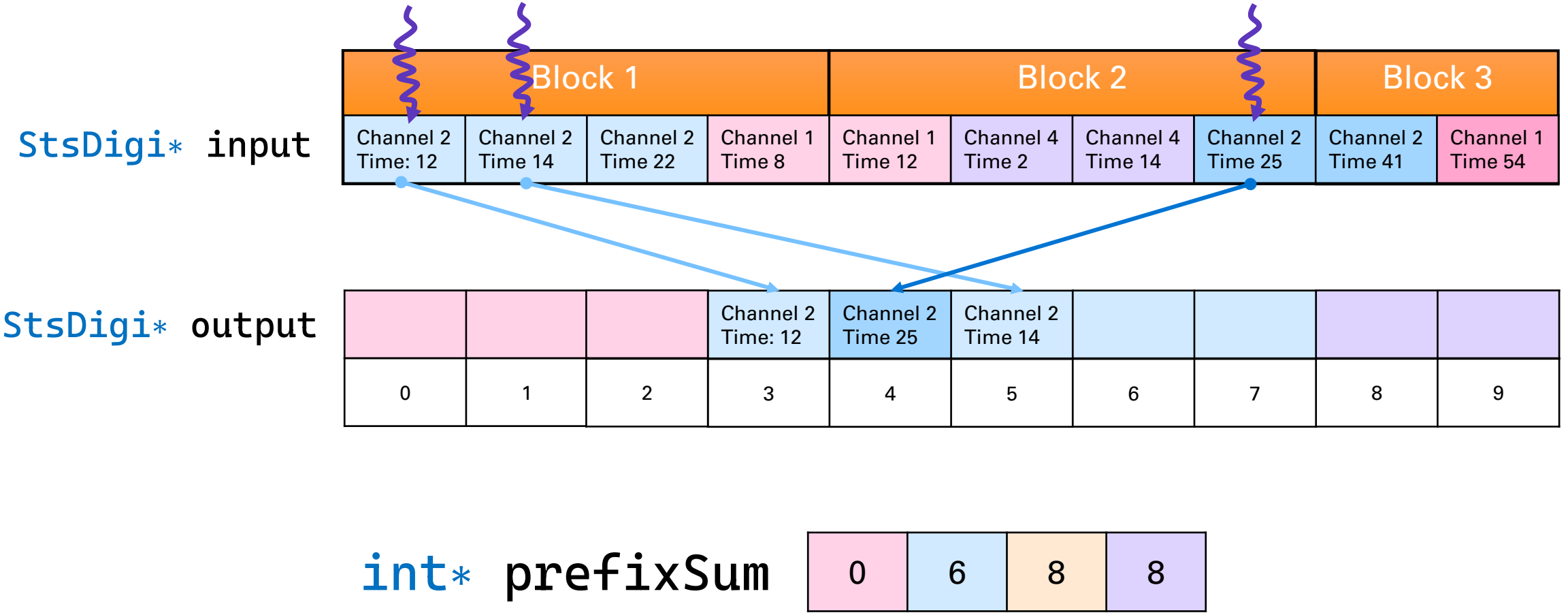



```
__global__ void kernel(const StsDigit* a, float* output,const int* prefixSum) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    output[atomicAdd(prefixSum[a[i].channel], 1)] = a[i].time;
}
```


StsDigi* input
Block 1
Block 2
Block 3
Channel 2 Time: 12
Channel 2 Time 14
Channel 2 Time 22
Channel 1 Time 8
Channel 1 Time 12
Channel 4 Time 2
Channel 4 Time 14
Channel 2 Time 25
Channel 2 Time 41
Channel 1 Time 54
StsDigi* output
0
1
2
3
4
5
6
7
8
9
int* prefixSum
0
4
8
8


StsDigi* input
Block 1
Block 2
Block 3
Channel 2 Time: 12
Channel 2 Time 14
Channel 2 Time 22
Channel 1 Time 8
Channel 1 Time 12
Channel 4 Time 2
Channel 4 Time 14
Channel 2 Time 25
Channel 2 Time 41
Channel 1 Time 54
StsDigi* output
Channel 2 Time: 12
0
1
2
3
4
5
6
7
8
9
int* prefixSum
0
4
8
8


StsDigi* input
Block 1
Block 2
Block 3
Channel 2 Time: 12
Channel 2 Time 14
Channel 2 Time 22
Channel 1 Time 8
Channel 1 Time 12
Channel 4 Time 2
Channel 4 Time 14
Channel 2 Time 25
Channel 2 Time 41
Channel 1 Time 54
StsDigi* output
Channel 2 Time: 12
Channel 2 Time 25
0
1
2
3
4
5
6
7
8
9
int* prefixSum
0
5
8
8


StsDigi* input
Block 1
Block 2
Block 3
Channel 2 Time: 12
Channel 2 Time 14
Channel 2 Time 22
Channel 1 Time 8
Channel 1 Time 12
Channel 4 Time 2
Channel 4 Time 14
Channel 2 Time 25
Channel 2 Time 41
Channel 1 Time 54
StsDigi* output
Channel 2 Time: 12
Channel 2 Time 25
Channel 2 Time 14
0
1
2
3
4
5
6
7
8
9
int* prefixSum
0
6
8
8

Solution?

Copy `prefixSum` to host, one loop, still $O(n + k)$

```
void sort(const StsDigit* a, StsDigi* output, int* prefixSum, const int n) {
    for (int i=0; i < n; i++) {
        output[prefixSum[a[i].channel]++] = a[i];
    }
}
```

Parallel copy?

Start one thread per Channel, each thread manages own channel offset

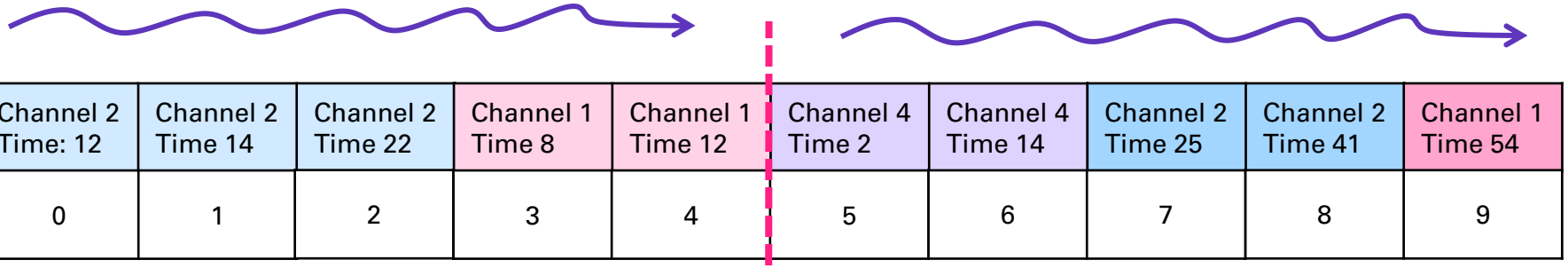
```
__global__ void kernel(const StsDigit* a, StsDigit* output, const int* prefixSum, const int n) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;

    for (int i=0; i < n; i++) {
        if (idx == a[i].channel-1) {
            // No atomicAdd, only this thread will increment prefixSum[a[i].channel] counter!
            output[prefixSum[a[i].channel]++] = a[i];
        }
    }
}
```

Alternative: Each thread does its own sorting?

ThreadIdx=0

$n = 10, p = 2, b = n/p = 5$

ThreadIdx=1



Channel 2 Time: 12	Channel 2 Time 14	Channel 2 Time 22	Channel 1 Time 8	Channel 1 Time 12	Channel 4 Time 2	Channel 4 Time 14	Channel 2 Time 25	Channel 2 Time 41	Channel 1 Time 54
0	1	2	3	4	5	6	7	8	9

```
StsDigi* sorted[n/p];
```

```
int* digisPerChannel
```

2	3	0	0

```
int* prefixSum
```

0	2	5	5

sorted

Channel 1 Time: 8	Channel :1 Time: 12	Channel: 2 Time: 12	Channel: 2 Time: 14	Channel: 2 Time: 22

```
StsDigi* sorted[n/p];
```

```
int* digisPerChannel
```

1	2	0	2

```
int* prefixSum
```

0	1	3	3

sorted

Channel 1 Time: 54	Channel :2 Time: 25	Channel: 2 Time: 41	Channel: 4 Time: 2	Channel: 2 Time: 14

$\theta(n/p + k)$ running time, but
$O(p * k + p * n/p)$ additonal space

$n = 10, p = 2, b = n/p = 5, k = 4$

ThreadIdx=0

ThreadIdx=1

Channel 2 Time: 12	Channel 2 Time 14	Channel 2 Time 22	Channel 1 Time 8	Channel 1 Time 12	Channel 4 Time 2	Channel 4 Time 14	Channel 2 Time 25	Channel 2 Time 41	Channel 1 Time 54
0	1	2	3	4	5	6	7	8	9

```
StsDigi* sorted[n/p];
int* digisPerChannel
int* prefixSum
```

2	3	0	0

0	2	5	5

sorted

Channel 1 Time: 8	Channel :1 Time: 12	Channel: 2 Time: 12	Channel: 2 Time: 14	Channel: 2 Time: 22

```
StsDigi* sorted[n/p];
int* digisPerChannel
int* prefixSum
```

1	2	0	2

0	1	3	3

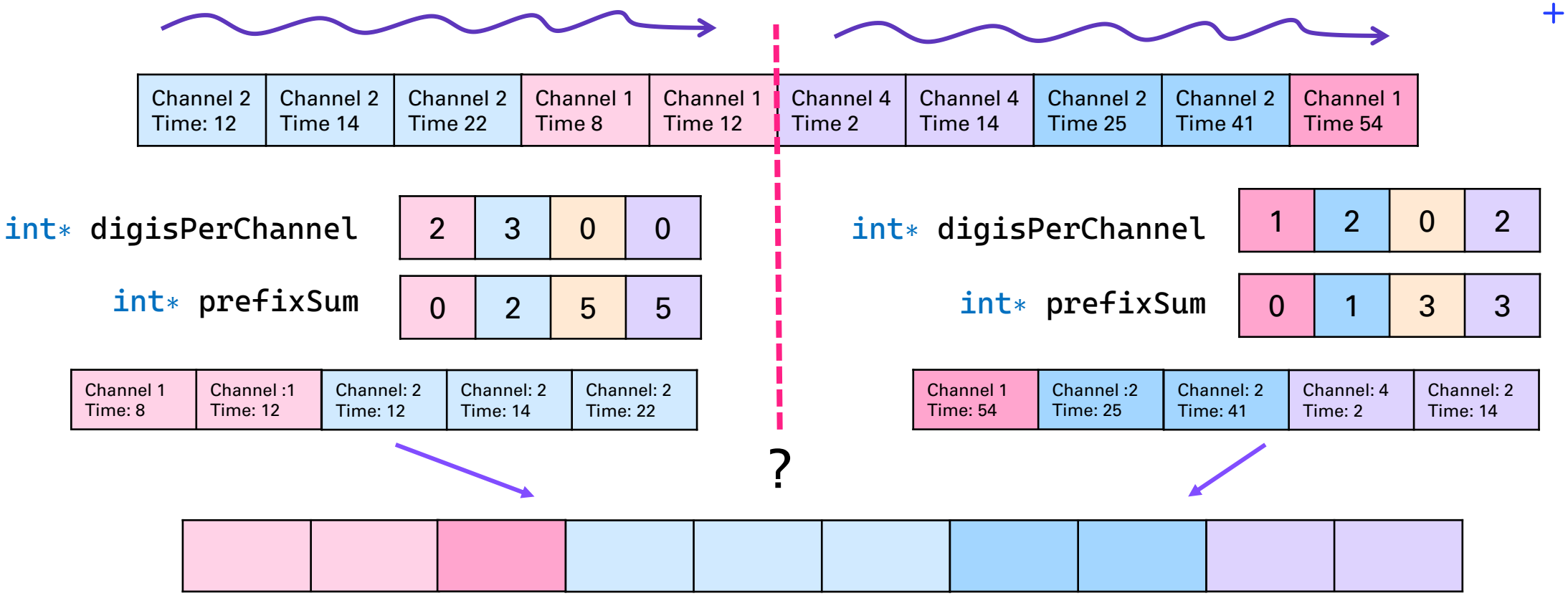
sorted

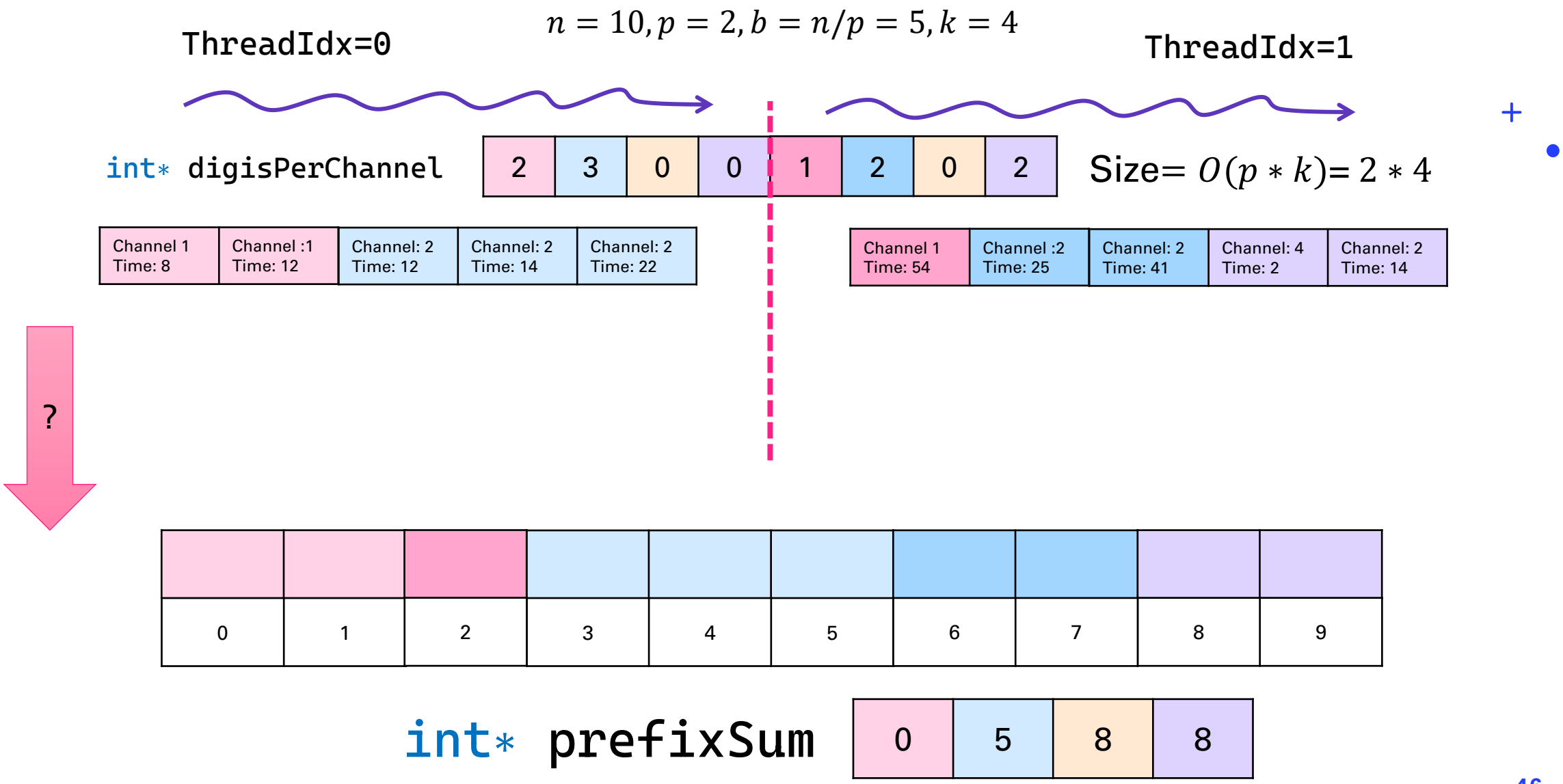
Channel 1 Time: 54	Channel :2 Time: 25	Channel: 2 Time: 41	Channel: 4 Time: 2	Channel: 2 Time: 14

$n = 10, p = 2, b = n/p = 5, k = 4$

ThreadIdx=0

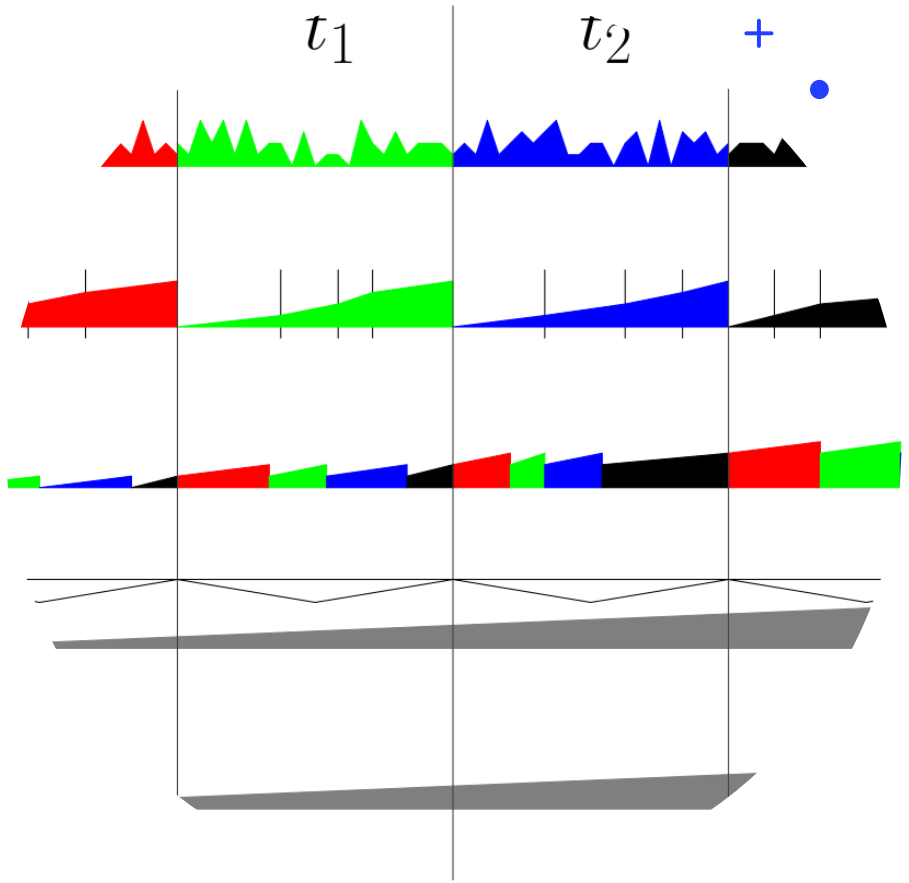
ThreadIdx=1




$n = 10, p = 2, b = n/p = 5, k = 4$
ThreadIdx=0
ThreadIdx=1
int* digisPerChannel
2 3 0 0 1 2 0 2
Size= $O(p * k) = 2 * 4$
Channel 1 Time: 8
Channel :1 Time: 12
Channel: 2 Time: 12
Channel: 2 Time: 14
Channel: 2 Time: 22
Channel 1 Time: 54
Channel :2 Time: 25
Channel: 2 Time: 41
Channel: 4 Time: 2
Channel: 2 Time: 14
?
0 1 2 3 4 5 6 7 8 9
int* prefixSum
0 5 8 8

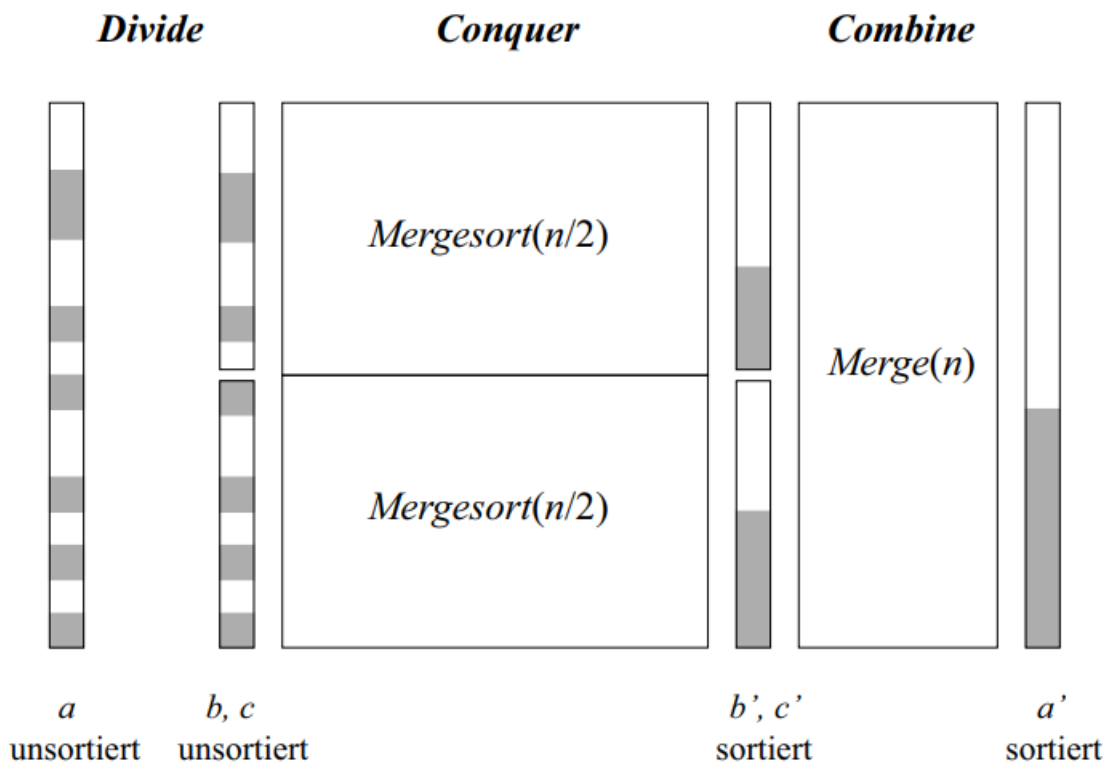
Alternative B

- Adaptive Mergesort
- Speed-up Mergesort
- Divide step is not always at n/2, but all presorted pairs are merged
- Identify the sorted sequences



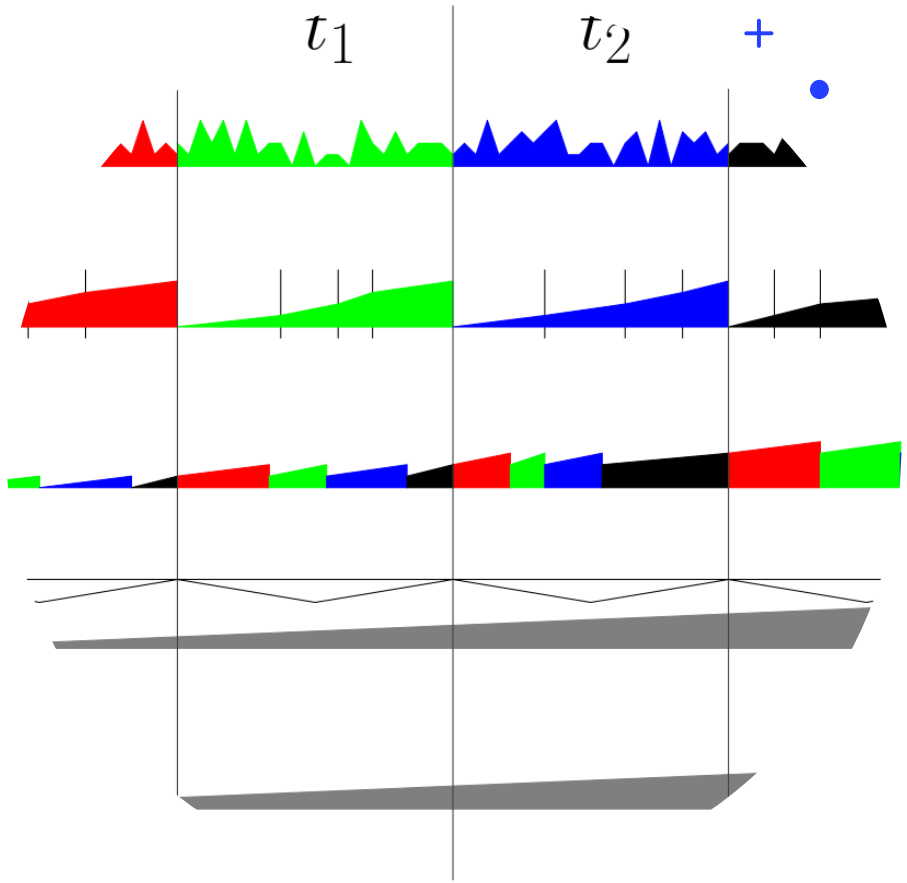
Standard

```
void mergesort(int lo, int hi) {
   if (lo < hi) {
      int m=(lo+hi)/2;
      mergesort(lo, m);
      mergesort(m+1, hi);
      merge(lo, m, hi);
   }
}
```



Natural Mergesort

- Regular Mergesort ignores any presorting and always executes all log(n) iteration-levels
- Best case Natural Mergesort has a time-complexity of Θ(n) (constant number of presorted subsequences)
- Worst case Natural Mergesort has the same running time, as Mergesort Θ(n log(n))

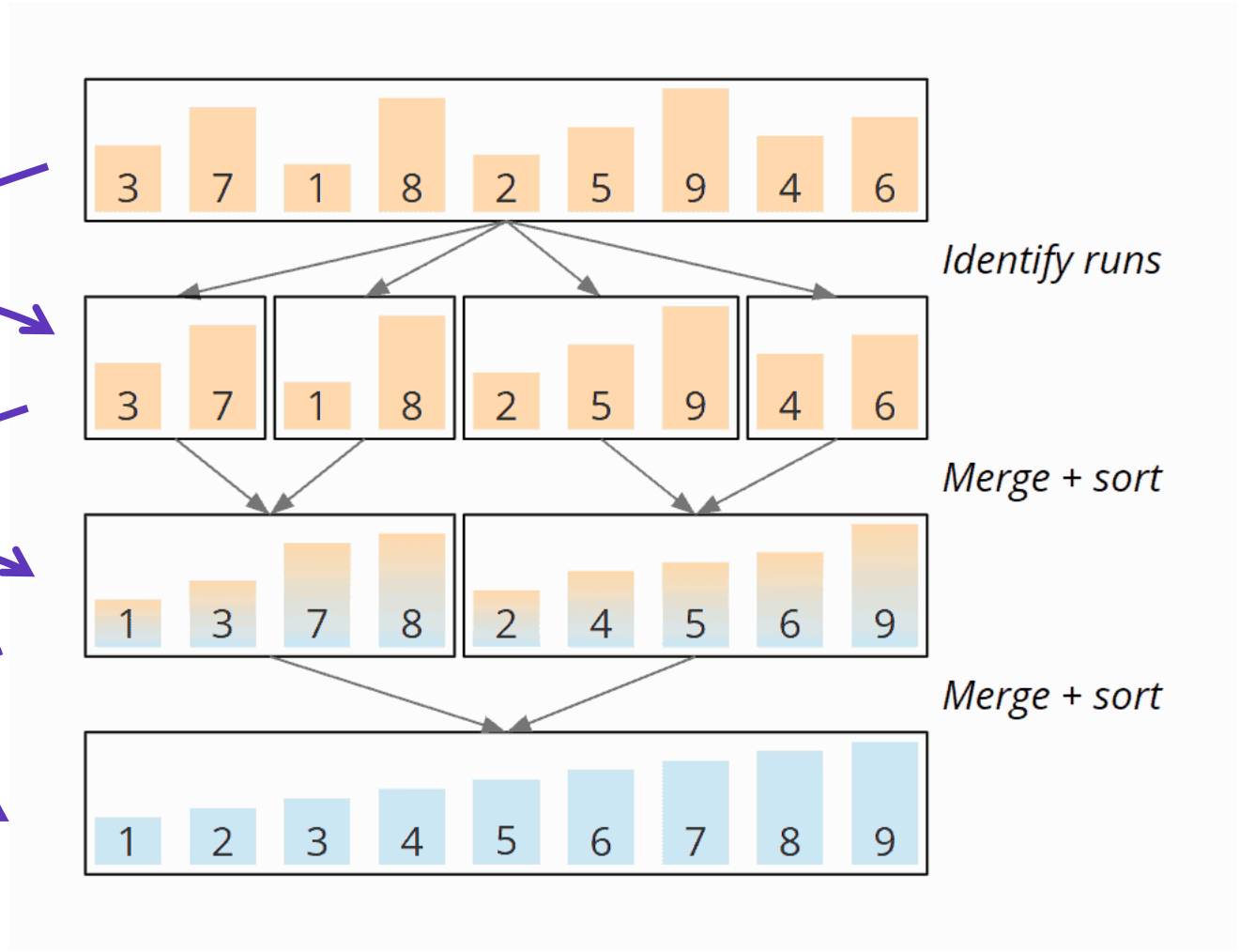


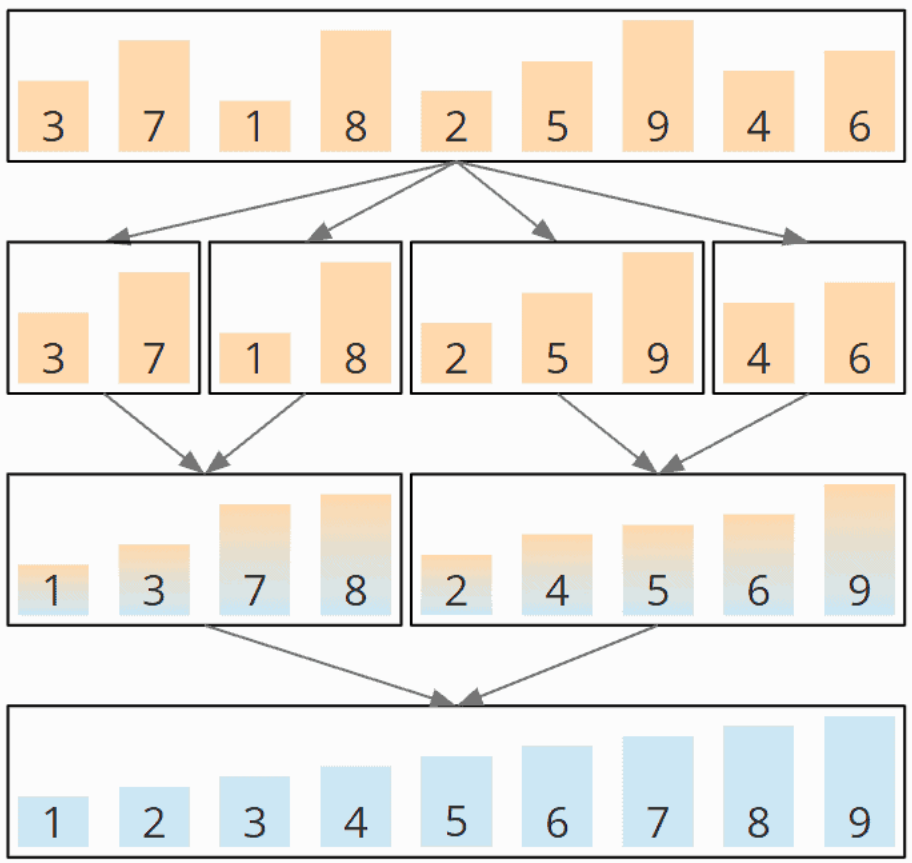
Natural Mergesort

- Monotonic (or bitonic) sub-sequences in the input data are exploited
- **Steps:**
  1. Find monotonic sub-sequences (runs)
  2. Now merge those sequences you found
- After each merge run, the number of runs halfs.
- Starting with r runs then **Θ(log r)** calls are requried. Since still n elements need to be merged the total running time is **Θ(n log r)**.
- Downside of any `merge` procedure is that it requires **Θ(n)** temporary space

Flatter recursion/rounds than $log_2(n)$

θ(n + log r)





Place all runs in queue

Pop 2 runs

Merge + Enqueue 1 run

Pop 2 runs

Merge + Enqueue 1 run

If Queue.size() == 1
→ done, one sequence

Prospect / Next steps

- Improving implementation (basic one already tested)
- Using simulated CBM data and mCBM-Digi measures to benchmark and compare the sorting approaches
- Doing benchmarking with different threading parameters,
  - paramter tuning

THANK YOU!

Saman Sedighi Rad

sasedigh@stud.uni-frankfurt.de