Computing Huge Histograms on the GPU

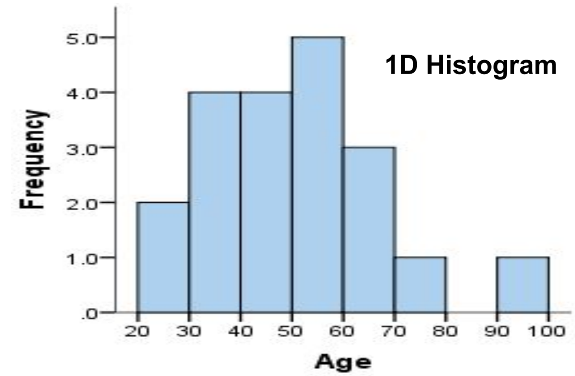
Project in 'Programming Massively Parallel Hardware'

Joachim Tilsted Kristensen

April 12. 2023

In collaboration with:
Christian K. Larsen,
Henrik G. Jensen
Mathias Grymer

What are histograms?

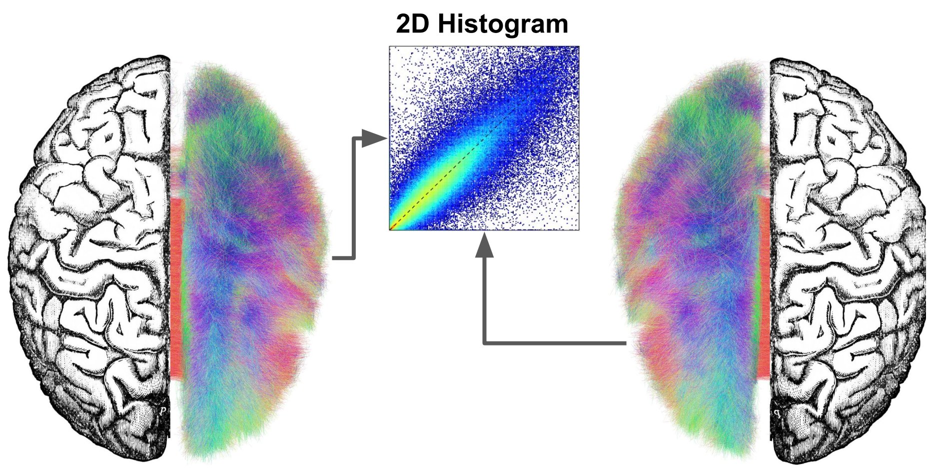


It is a way to **visualize data** frequencies.

A histogram is a **density measure**.

Normalized, it is a discrete **probability distribution over a continuous variable**.

A way to define **numeric similarity between continuous datasets**, e.g. images or more complex data.



The challenge with histograms

Histogram pros:

- ➢ Easy to compute
- ➢ Trivially parallelizable
- ➢ Built over a fully parallel loop

Histogram cons:

- ➢ Array indexing is unpredictable for unsorted data

Histogram pseudocode

```
for(i=0; i<size(data); i++)
    idx = f(data[i])
    // Must be atomic
    hist[idx] += 1
```

So what? Don't we have fast shared memory per block on the GPU?

Well ..

2D histograms quickly become too large for individual blocks. And inter-block communication is bad.

Optimizing For Memory Performance

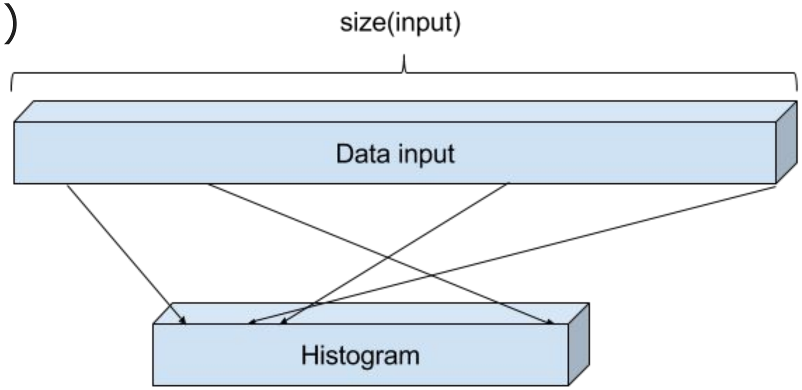
- As the number of bins increase the range of possible memory addresses increase as well

- The increased random accesses to global memory in turn increases the probability of cache misses (expensive)

How do we fix this?

- Partially sort indices in order to make global memory accesses more coalesced to improve cache performance.

- Additionally, minimize the number of writes to global memory by local histograms (working on shared memory)



Eliminating random memory access

- Case : The histogram fits in shared memory (Trivial).
- Case : The histogram does not fit into shared memory.
  - Successive writes to the global histogram are not guaranteed to fall into the same memory block
  - This means, unnecessarily evicting blocks all the time.
  - Partially sort the input data, into *segments*, where each *segment* consists of elements which go to the same sub histogram which is smaller than shared memory.
- Pros :
  - When a block is evicted from cache, it is no longer needed.
  - Cache misses are minimized
- Cons :
  - Introduces significant amount of bookkeeping
  - Sorting is very expensive.

Algorithm for small histograms (the trivial case)

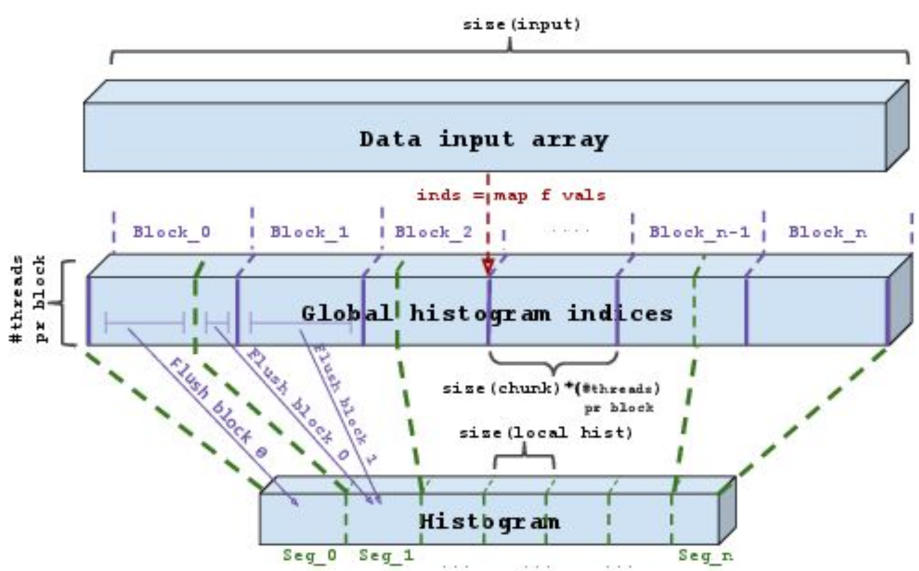
- The histogram fits into shared memory.
  - Shared memory is pr. block, 4096 or 8192 words
  - Each CUDA block works on a local histogram of this size
- To fully utilize the hardware, we distribute the total workload into evenly sized chunks.
- Each CUDA block atomically commits its local histogram to the global histogram.

$$\mathtt{chunk\_size} = \frac{\mathtt{total\_workload}}{\mathtt{hardware\_parallelism}}$$

An algorithm for larger histograms

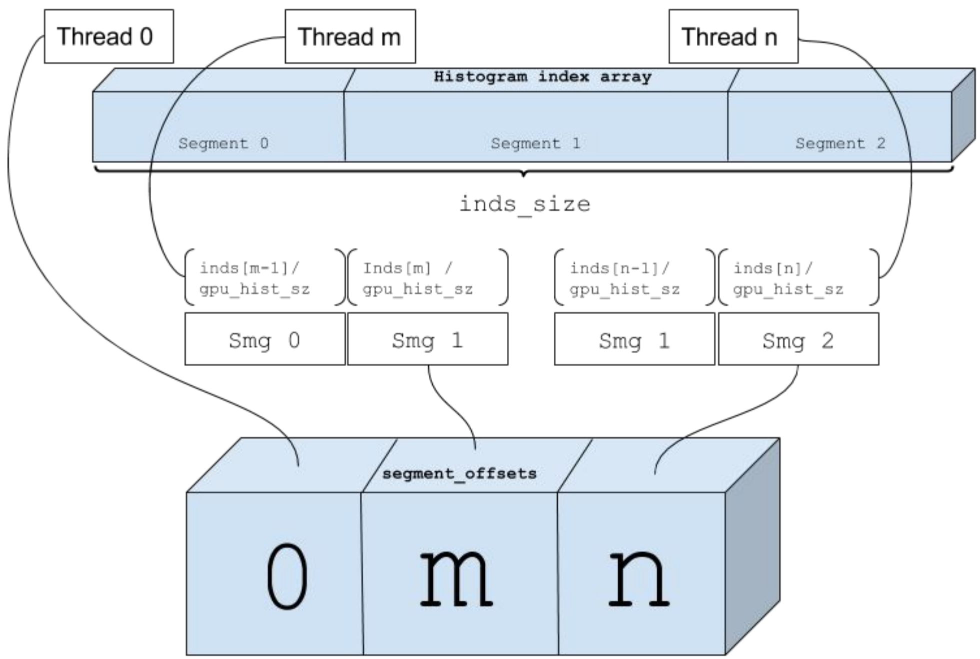
- Prelude :
  - Map f onto the data input array
  - Partially sort the index array
  - Compute information about segments

- Actual algorithm :
  - Distribute the work into evenly sized chunks
  - Each thread jumps with a stride of block_size
  - At a segment split. Commit back the segment related sub-histogram.

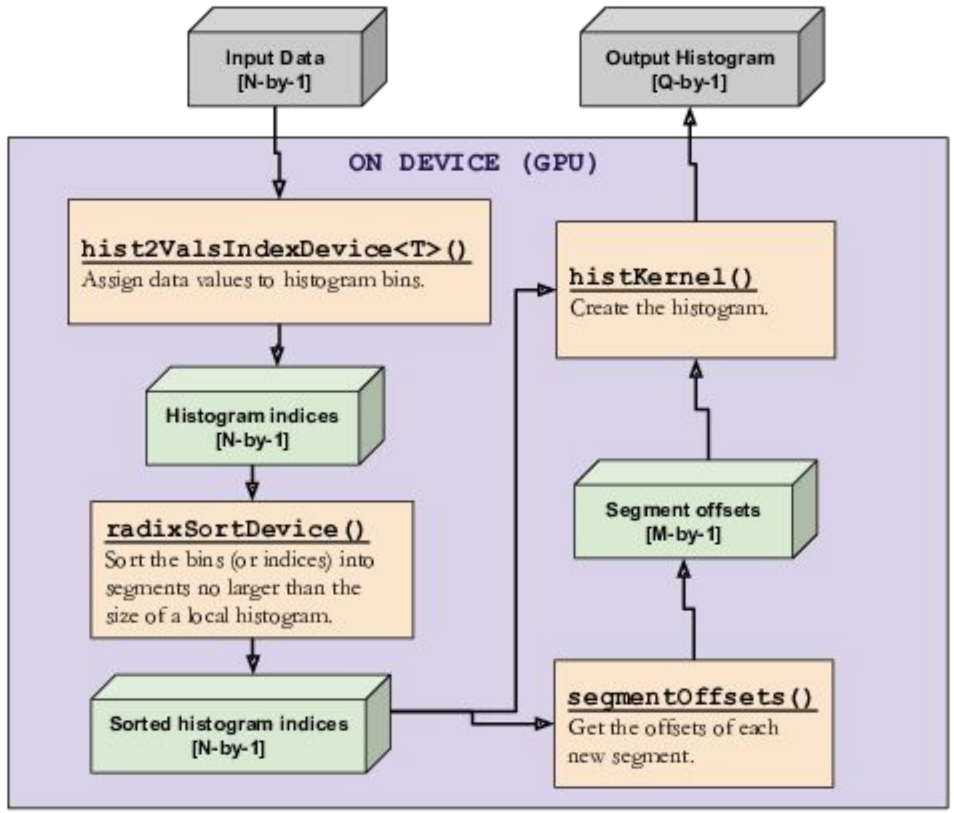


Finding segment offsets

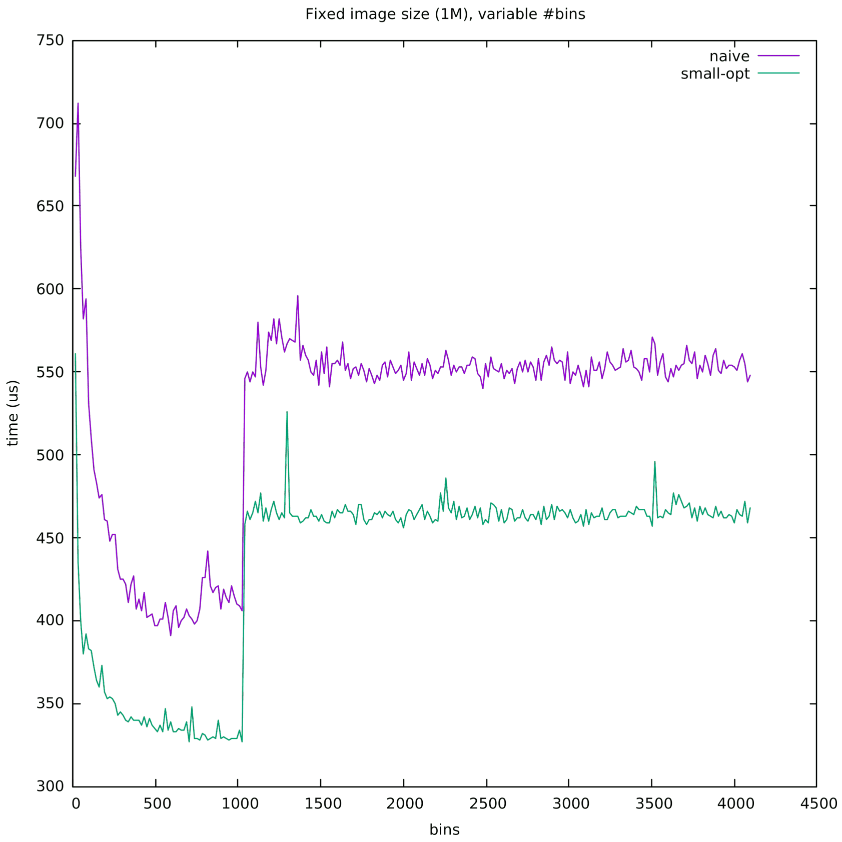
- Pass through the index array.
- For each thread, iff. inds[gid-1] is in a different segment, seg_offsets[this_segment] = gid.

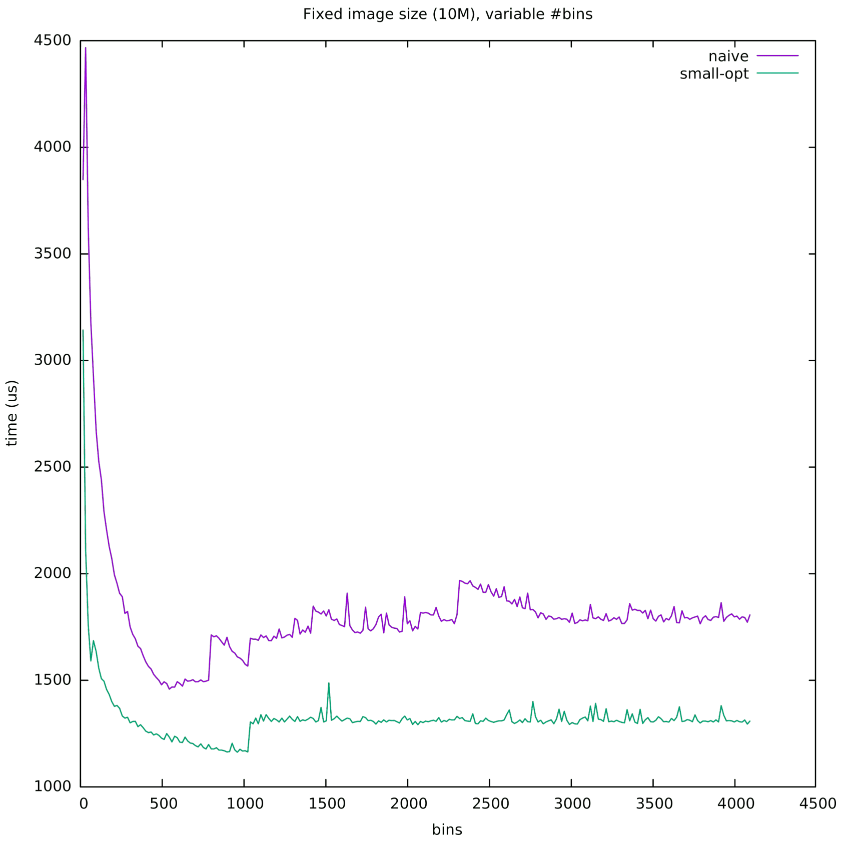


Data flow on the GPU



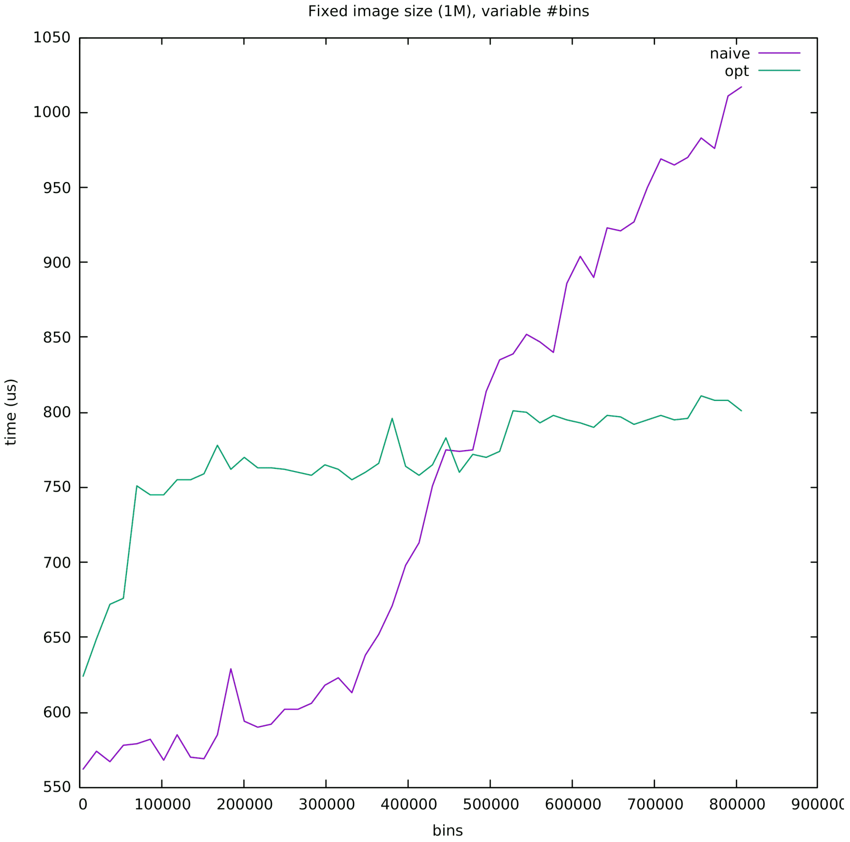
Benchmarks (Small histograms)

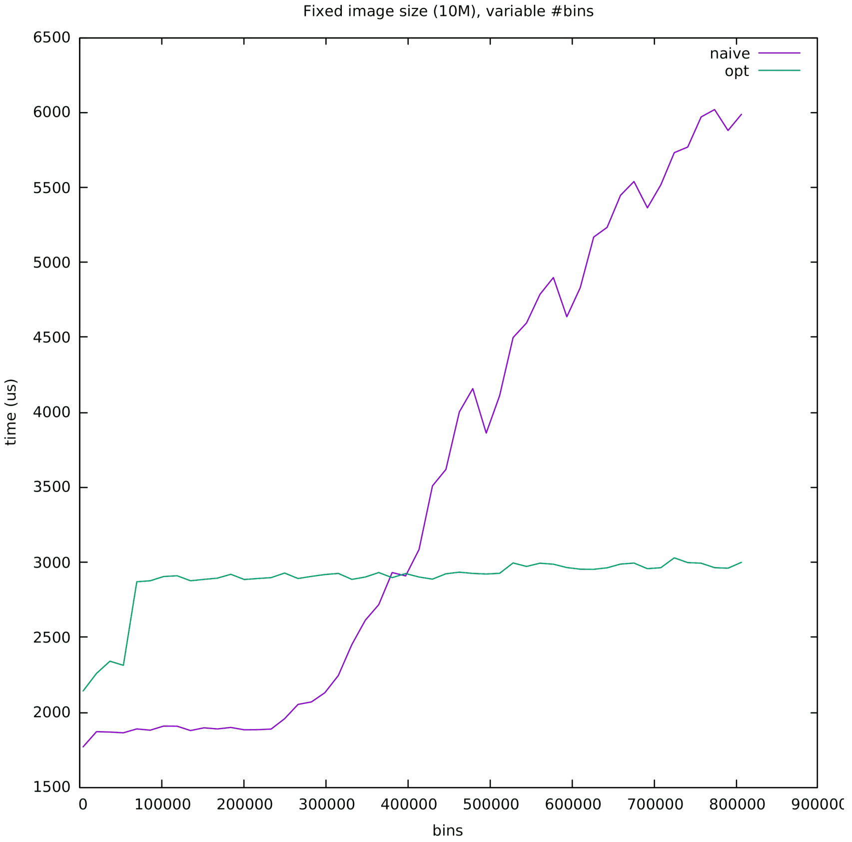




- Remarkably similar characteristics
- The optimised version is a bit faster
- Slow for few bins because of synchronisation

Benchmarks (Larger histograms)





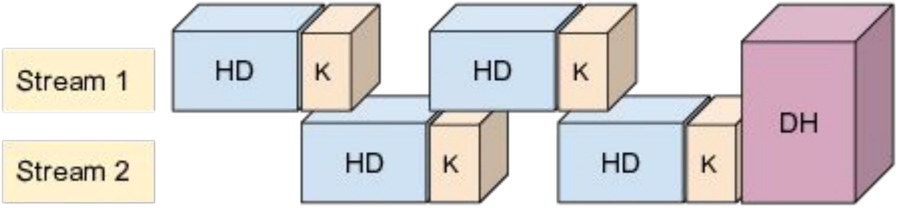
Benchmarks (Larger histograms) cont.

- Our version is better from around 300k to 400k bins
- This is not good
- Profiling of 10M elements 350k bins (`nvprof`)
- Radix sort is expensive
- Our kernel is much faster than the naive one
- The hardware caches efficiently
- Memory copying took 22ms

Kernel	Optimised (μs)	Naïve (μs)
Index and boundary calculation	537	537
Segment offsets	417	-
Radix sort	1228	-
Histogram kernel	446	1433
Total	2628	1970

Streaming

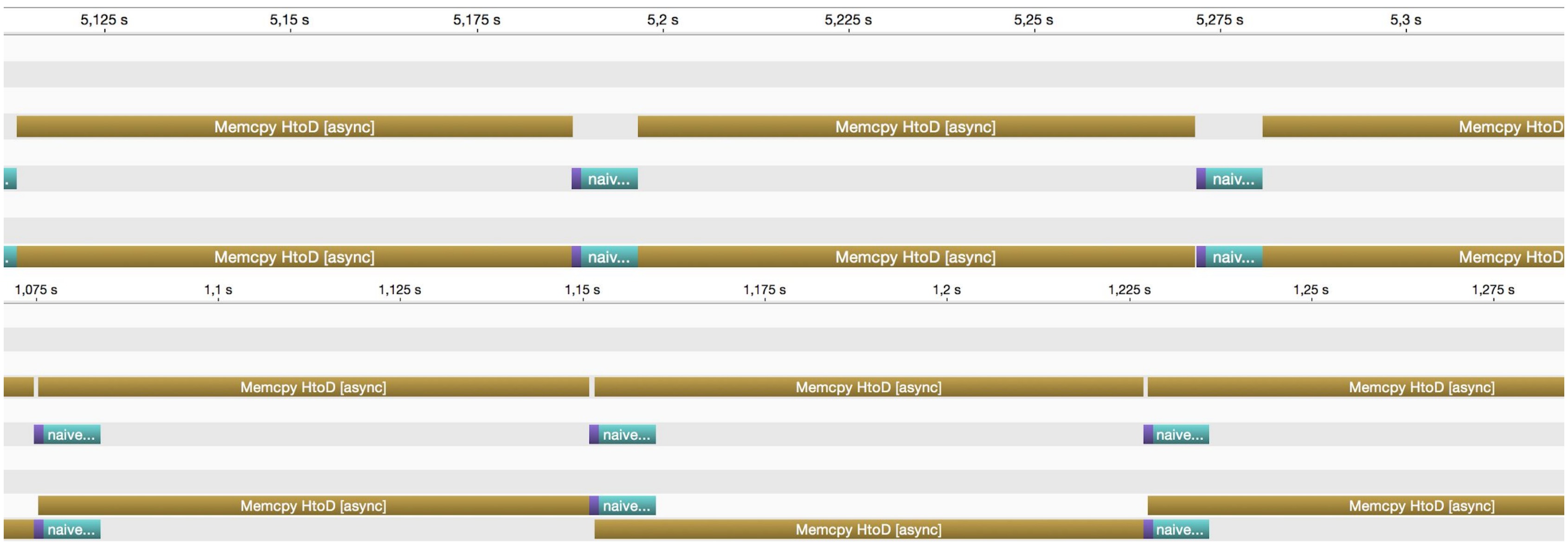
- We would like to compute histograms from data that does not fit on GPU
- Copying is expensive
- We would like to compute histograms while copying
- CUDA streams to the rescue
- Additional pipelined steps can be added




```
cudaStreamCreate(stream);
cudaMemcpyAsync(..., stream);
kernel<<<blocks, blocksz, sh_mem, stream>>>(...);
```

Results (Nvidia Visual Profiler)

- Memory copying is sequential (non-overlapping)
- Pipelining success is limited by the amount of work in the other stream



CPU (without any copying)	9.6s
GPU, 1 stream	4.9s
GPU, 2 streams	4.8s

Conclusion

- Is it possible to make a memory efficient histogram computation, that is also scalable?

- It is possible to implement an algorithm for histograms, that ensure memory efficiency as the number of bins increase.

- The overhead of radix sort and general bookkeeping makes the approach viable later than expected.

- Copying is still the most expensive part.

- Streaming makes it possible to have more computation while copying the data.

Fusing segment offset into kernel

- A fusion will eliminate one passover of the sorted histogram index array
- Segment offsets are simple to compute
- Starting segment for a block workload found easily
- Introduces thread communication and race conditions

From 1D to n-dimensional histograms

- The input data changes from floats to n-dimensional vectors
- Indices (the result of applying f) are then n-dimensional vectors as well
- Convert n-dimensional indices to 1D indices (linear indexing)
- Reuse existing histogram algorithm
- Convert 1D indices to n-dimensional