FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Programming Massively Parallel Hardware

Final Project

Joachim Tilsted Kristensen (qvp401)

Christian Kjær Larsen (wkm839)

Henrik Grønholt Jensen (tdq444)

Mathias Grymer (hpl465)

November 3, 2016

Contents

1 Project Introduction

A histogram is a density estimate over a distribution of data. It is quintessential to quality control, distribution estimation and similarity measures when working with large datasets. As it scales with the size of the input, a parallel solution is preferable when the input is large. However, when it comes to real world unsorted data, histograms are computationally inefficient due to the random memory access caused by the counting of how many values fall into each bin. In this project, we implement a CUDA-based solution for building a 1D histogram in parallel and benchmark the scalability of our solution with the CPU and the naive GPU versions. We also investigate how to efficiently solve the problem when the dataset is larger than the available (device) memory on the GPU using streaming techniques.

2 Design Overview - From CPU to GPU

Building a histogram in parallel is a straightforward map-reduce problem. It can be thought of as simplistic kernel density estimation where a function f maps frequencies of data `vals` over bins `inds` and reduces by counting how many values falls into each bin. The unoptimised pseudo-code for a 1D histogram would look something like:

```
Forall (i = 0; i < size(data); i++)
    Idx = f(data[i])
    hist[Idx]++ // Must be atomic
```

For an efficient parallel implementation on the GPU, the reduce-phase becomes the tricky part. The unknown order of indices from the map-phase prevents us from working with the data in a coalesced manner during the reduce-phase. Our solution to this problem is to partially sort the output indices from the map-phase into *segments* that are guaranteed to fit in the shared memory of a CUDA block (i.e. no indices fall outside the subset of bins we can hold in shared memory).
Each block will use its shared memory to create a local histogram and atomically add this to a global histogram residing on the GPU. However, to fully utilize hardware parallelism we are likely to have each CUDA block working on a large subset of data that will span more than one *segment* (local histogram).
The subset of data (or indices) a block handles is defined as the workload per thread (`chunk size`) times the number of threads in a block (`thread pr block`). This workload depends on the size of the input data vs. total number of threads available, and is independent of the sorted segments. Thus, in each block we need to know when to flush a local histogram to global memory and start a new segment. The problem is illustrated in Figure 1 where e.g. $block_0$ spans $segment_0$ and $segment_1$.

Letting a block handle overlapping segments (i.e. creating more local histograms) is preferable to a more naïve implementation where a block is spawned for each segment. This is to keep the algorithm work efficient. The problem with a more naïve version is that there is no way to guarantee that all blocks have roughly the same workload. One could easily have a case where certain segments have significantly more elements than others, leading to bottlenecks where other blocks will wait for the overworked blocks to finish.

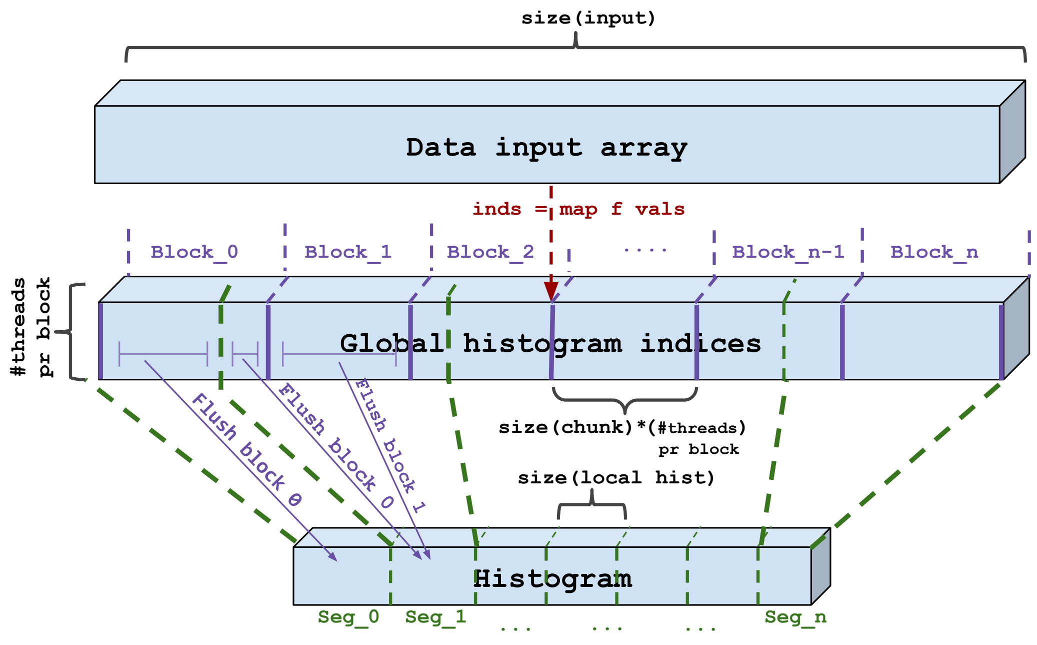

Figure 1: the general idea

3 Overview of Implementation

3.1 Optimised for small histograms

The first step towards the algorithm described previously is an implementation for histograms less than the size of the shared memory for each CUDA block. In our case that is 4096/8192 elements (depending on the configuration). This is just a way of explicitly managing the cache on the CPU, because the entire histogram fits in the cache anyway.

- Reset the shared memory for the block.
- Write the indices (`f(data[i]`) into the shared memory of the block.
- Commit the block level histogram to global memory.

3.2 Optimised for large histograms

This is the version of the previous algorithm with additional bookkeeping to make sure that the segments are written to the correct location in global memory.

- Map f onto the data array
- Partially sort the resulting indices into a number of equivalence classes (segments)
- Calculate the offset for each segment into the partially sorted array of indices. This is needed to find the segment for each block.

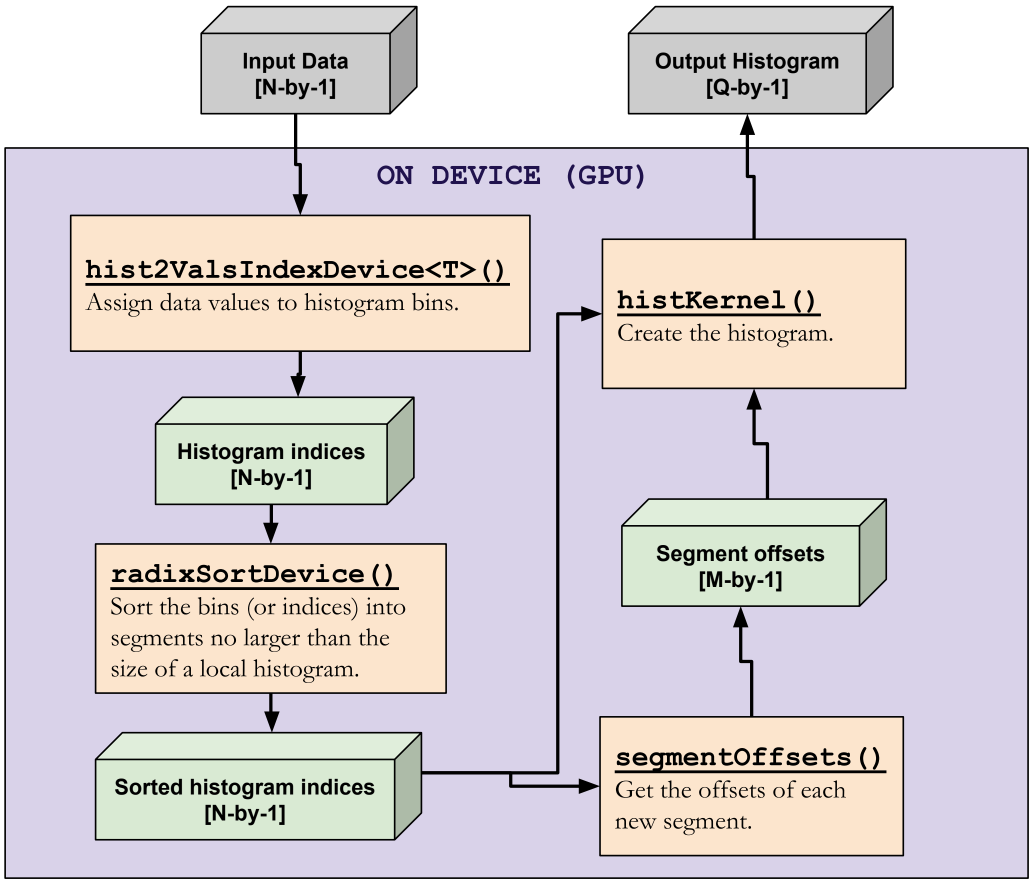

Figure 2: figuretext

- Reset the shared memory for each block
- Write the indices into the shared memory of the block modulo the shared memory size
- If we change segment, we commit the block level histogram to global memory with a certain offset and continue with the next segment in the same manner.

A lot of details are left out, but they are very specific to the hardware, and it is a lot easier just to read the CUDA implementation. This is especially for details regarding the bookkeeping and calculation of segment offsets.

4 Benchmarks

To evaluate our solution, we have benchmarked our implementation with a sufficiently large dataset. We have kept the size of the input data constant, because we are testing whether our implementation is more efficient when the number of bins in the histogram increases.

The benchmark shows that somewhere between 300k and 400k bins, the optimised implementation is faster. This seems like a lot of bins, so we use nvprof to see what kernels dominate the runtime. The following measurements is for 10M elements and 350k bins.

Kernel	**Optimised (us)**	**Naive (us)**
`Index and boundary calculation`	537	537
`Segment offsets`	417	-
`Radix sort`	1228	-
`Histogram kernel`	446	1433
Total	2628	1970

Table 1: Your caption here

It seems like radix sort is very expensive. It takes almost the same time to do as the naive histogram kernel. On a more positive note the optimised histogram kernel is about 3 times as fast as the naive kernel. A possible explanation might be that radix sort from the Cub library sorts on more bits than we need. We also benchmark for a smaller dataset (1M).

The results are similar. We have also benchmarked the kernel optimised for smaller histograms. Again we keep the size of the input array constant, and vary the number of bins.

The histogram kernel optimised for small histogram sizes is around 20% faster than the very simple naive version. Both implementations see a spike around 1k bins. This is probably the sign of some cache effects. This is a pretty good result, and it confirms that it is a good idea to explicitly manage the cache, if you know about the distribution of the input data.

4.1 Asynchronous histogram computation

For the case where a histogram can still fit on the gpu, however we wish to look at the case where the remaining memory does not suffice for data input. In cuda the streaming module can be used for this, which in short diverts from using default stream in order to distribute bandwidth between processes. This allows for asynchronous memory copying, thus granting the GPU time to work on a subpart, whilst receiving new data input. Cuda supports streaming, but happen to be far simpler if a direct kernel call is made. Rather than our case with several calls to wrapper functions, which would require major rework in order to properly work.

5 Discussion

A short introduction, to the things we are going to discuss ?

5.1 The problem with random access to global memory

A read to global memory causes a lookup into the L1 cache, if the data is not present there, it is searched in the L2 cache and so on. At each step out into the memory hierarchy, things become larger but slower and have lower bandwidths due to larger multiplexers and greater wire delays. Additionally, whenever something is read from global memory it is written back through the memory hierarchy, replacing something else. Hence, having threads access global memory at random greatly increases the probability of cache misses, which is not very good for performance.

So in order to avoid evicting blocks from L1 cache, until we are done working with them, we want the cuda kernel to work at small local histograms which fit into shared memory, and then flush back into corresponding parts of the global histogram in sequential order. Effectively, this means partially sorting the input data into segments, such that all values in a segment, contributes to the same small `gpu_hist_size` number of bins. Unfortunately, sorting data and keeping track of segments introduces great deal of extra work, whose expense has to be compared with the cost from being very non cache-effective. A comparison which is done best, simply by measuring performance on different implementations, which is exactly what we did.

About shared memory. Shared memory is divided into equally-sized memory banks. Any memory read or write request made of n addresses that fall in n distinct memory banks can be serviced simultaneously, yielding an overall bandwidth that is n times as high as the bandwidth of a single module. However, if two memory requests fall in the same bank. The hardware resolves the conflict by serializing the requests, obviously decreasing throughput by a factor equal to the number of separate memory requests.

5.2 Atomic addition

(the reason why the parallel version performs better for greater size histograms).

Imagine a histogram of size 1. Even though the thought is silly, it reveals a real bottleneck. Since all of the additions to the histogram must be atomic, the reduction part of the algorithm for such a small histogram (since all of the additions must be atomic) exposes exactly 0 parallelism! But as the size of the global histogram expands, this problem becomes smaller and smaller. –Our focus will not be of the small histogram, and therefore we won't be looking into pitfalls caused by histograms with a size of less than what can fit into shared memory on GPU. –Giver det mening??

5.3 utilizing all available parallelism

Theoretically, the amount of time needed to compute a histogram is proportional with the greatest number of additions to a single bin. In practice however, this is not the case, since the available parallelism is limited by physical device resources. And so the ideal case becomes the one, where the resources of the hardware is fully utilized at all times. At a high level of abstraction, this means dividing the workload into equally sized chunks. The size of a chunk is thus `total_workload / hardware_parallelism`. Where the total workload in this case is defined as the number of index values to be computed per thread, and the available parallelism is the number of threads which can be run at the same time.

6 Conclusion

When computing large histograms, it is often the case, that neither the dataset, nor the global histogram itself fits into shared memory, and so the greatest work overhead in computing huge histograms, are related to memory access patterns. ... and there was much rejoicing ... jææææ...

7 Further improvements

Right now we are making a full pass over the index array to calculate the segment offsets into the index array. We are pretty sure it would be viable to fuse the segment offset kernel into the histogram kernel. Both kernels do a full pass over the index array, and this would reduce some of the overhead introduced by the optimised kerned. If successful, this would reduce reduce the time computing the histogram with a significant factor.