Parallelizing General Histogram Application for CUDA Architectures

Ugljesa Milic*†, Isaac Gelado*, Nikola Puzovic*, Alex Ramirez*† and Milo Tomasevic‡
*Barcelona Supercomputing Center
Centro Nacional de Supercomputacion, Barcelona, Spain
Email: {first.last}@bsc.es
†Universitat Politecnica de Catalunya, Barcelona, Spain
‡School of Electrical Engineering, University of Belgrade, Belgrade, Serbia
Email: mvt@etf.rs

***Abstract*—Histogramming is a tool commonly used in data analysis. Although its serial version is simple to implement, providing an efficient and scalable way to parallelize it can be challenging. This especially holds in case of platforms that contain one or several massively parallel devices like CUDA-capable GPUs due to issues with domain decomposition, use of global memory and similar.**

In this paper we compare two approaches for implementing general purpose histogramming on GPUs. The first algorithm is based on private copies of bin counters stored in shared memory for each block of threads. The second one uses the Thrust library to sort the input elements and then to search for upper bounds according to bin widths. For both algorithms we analyze how the speedup over the sequential version depends on the size of input collection, number of bins, and the type and distribution of input elements. We also implement overlapping of data transfers between host CPU and CUDA device with kernel execution.

For both algorithms we analyze the pros and cons in detail. For example, privatization strategy can be up to 2x faster than sort-search with realistic inputs, but can only support a limited number of bins. On the other hand, sort-search strategy has about 50% higher speedup than privatization when we use characters as input and can support unlimited number of bins. Finally, we perform an exploration to determine the optimal algorithm depending on the characteristics and values of input parameters.

I. Introduction

Histogramming is one of the basic statistic tools used in data analysis [1]. Strictly speaking, a histogram is a function that counts the number of observations (elements of the input) that fall into each of disjoint categories (bins). A histogram is commonly represented as an array where each element corresponds to one of the bins and contains the number of input elements that fall into it. Bins are defined by their starting element and widths that are chosen so that different bins do not overlap. When an input element is found to fit into that range, the value of the bin will increment by one. In this way, a histogram approximates the probability density function for a given input set. Histograms are widely used in data mining, image analysis, pattern recognition, data presenting and data analysis in general.

The calculation of a histogram is straightforward when it is done in a sequential way. There are also a wide range of parallel implementations of histograms for multi-core, SMP, and NUMA machines. These implementations have served as the basis for the implementation of histogramming algorithms on GPUs using CUDA (*Compute Unified Device Architecture*) [2]. However, GPUs pose several challenges to achieve an efficient and scalable execution. For instance, the usage of atomic instructions greatly harms the performance in GPUs because it can potentially serialize the execution of all threads in a warp that, otherwise, would be executed in parallel. However, if the histogram is being computed for very sparse data that results in small bin counts, such a conflict on concurrent threads seldom happens and, therefore, the usage of atomic instructions might be quite efficient.

There are two main strategies to produce a parallel implementation of dense histograms with an arbitrary number of bins, bin width, and data type. The first approach is based on decomposing the input set in many domains that are individually computed by each thread block (i.e., privatization). Each of these individual histograms is used to update the output histogram when all threads in the block have finished. This approach requires an extensive use of atomic operations both in shared and global memory to produce the private histogram and to update the final output respectively. Hence, this implementation might be quite inefficient when computing histograms with few bins and/or very sparse input data. We also explore a different implementation where the input data is first sorted. After that, positions of sorted elements are found according to the upper bounds of bin widths. This implementation completely avoids the usage of atomic instructions, but requires a sorting stage that might introduce large overheads. In this paper, we present an experimental analysis of the different trade-offs of each of these implementations.

The main contributions of this paper are:

- Analysis of the state-of-the-art algorithms for implementing histrogramming on GPU architectures.
- Implementation of the general purpose histrogramming algorithms that can operate with any data type or bin widths, and with any given size of the input and output arrays.
- Analysis of trade-offs for tuning the performance of algorithms for GPUs, and design space exploration for determining the best algorithm for a given set of input parameters.

The rest of the paper is organized as follows: Section II introduces sequential implementation of algorithm. In Section III, we present implementation details of both CUDA algorithms. Results are observed in Section IV. Related work is discussed in Section V showing several approaches for implementing histogram on GPUs. We conclude in Section VI and we propose several possible improvements to the algorithms that are presented here.

II. Sequential version

The sequential version of histogramming is straightforward: for each element in the input collection, the algorithm finds the corresponding bin and increments its counter. The algorithm is implemented in C++, and allows for flexibility in the definition of the resulting histogram. The user can completely customize the output bins, specifying the bin widths which do not have to be equal for all bins, or the bin width can be calculated in the initialization phase based on the range of input elements and the required number of bins. The user can also specify the additional filtration of input elements to narrow the range that will be considered when the histogram is created. By using C++ templates, we allow the use of any data type in the input collection, as long as it provides comparison operations for locating the correct bin.

Since the bin width is not fixed, we cannot find the corresponding bin for each element in constant time. Hence, this is performed using binary search with $O(\log(B))$ complexity (B is the number of bins). Consequently, the complexity of the sequential version is $O(N \cdot \log(B))$, where N is the number of elements in the input collection.

```
//N      - input size
//bw     - upper limits(bin widths)
//bc     - bin counters
//B      - number of bins
template<typename T>
void cpu_histogram(T* input, int N, float* bw,
                   uint* bc, int B) {
uint i, b;

reset_counters(bc, B);
for(i = 0; i < N; i++) {
  b = find_bin_binary(input[i], bw, B);
  bc[b]++;
}
```

Listing 1: Pseudo code for the sequential algorithm (T is the data type provided by the user).

Pseudo code of sequential version is given in Listing 1. The input of the algorithm is the collection that contains elements we want to analyze (denoted as *input* in the listing). The user also specifies bin widths (denoted as *bw*). As a result, algorithm calculates the histogram as an array of counters (denoted as *bc*) – one counter for each bin.

In all our analysis, we use the execution times of this serial implementation as a baseline to calculate the speedups of our CUDA codes.

III. CUDA versions

A. An overview of CUDA

Here we provide an overview of the CUDA as well as its basic concepts and terminology.

The code running on the CUDA device, known as the *kernel* code, is executed by large number of *threads* grouped in *blocks* enabling internal synchronization and communication. Different threads and blocks can be distinguished by their unique ID. Blocks make a *grid* which presents a top level abstraction of thread hierarchy. The number of threads per block and number of blocks in a grid can be set in run-time for each kernel launch. The device itself is consisted of streaming multi-processors (SM) so that each block of threads is running on one of them. Threads within a block are grouped into *warps* as a basic unit of scheduling. All threads in a warp execute the same instruction since they share same program counter but operate on different data and are free to branch independently. Communication and data transfers between CPU (host) and CUDA device is done using global memory. It is the largest but also the slowest one in memory hierarchy on CUDA devices. Each thread from any block can both read and write to global memory in coalesced way since it is banked. Beside that, shared memory is smaller but much faster since it is on-chip, close and private for every SM. Threads within a block can communicate and use atomic operations on data that are stored there. Proper and extensive use of shared memory can significantly increase performance of kernels. For more details on the CUDA programming model we refer the reader to CUDA programming guide [3].

In this paper we use two latest NVIDIA's architectures named Fermi and Kepler. Kepler architecture, released last year, is built on the foundation of Fermi GPU architecture. The big jump from Fermi to Kepler is the number of CUDA cores per SM (now called SMX) which has been increased to 192 so now 192 concurrent threads can be executed in parallel on one SMX. When it comes to memory hierarchy everything remained the same concerning our algorithms except reduced

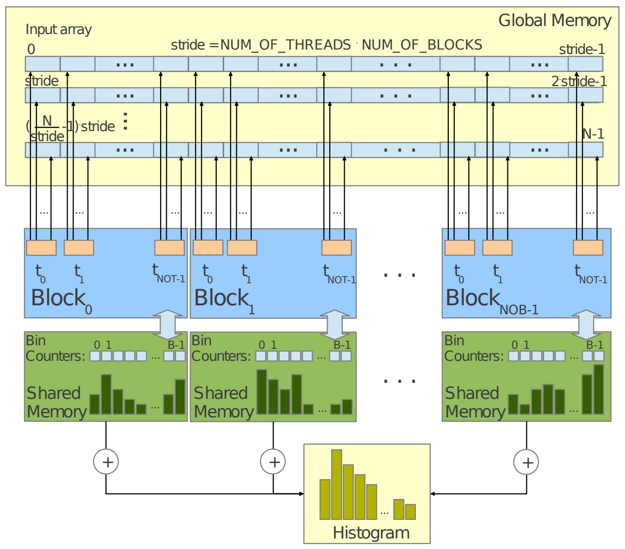


Fig. 1: Graphical representation of privatization algorithm.

shared memory latency and increased memory bandwith. One feature introduced on Kepler that can change the behaviour of our application is GPU Boost which automatically adjusts the clock speed based on the power consumed by the currently running kernel. We will refer to some of these differences in Section IV. More specific details on Fermi and Kepler architectures can be found in [4] [5].

B. Privatization algorithm

The first algorithm uses private copies of bin counters for each thread block (Figure 1 and Listing 2). Before the kernel is launched the input collection is copied from the host to the device's global memory along with information about bins (bin widths and global bin counters). Threads within a block will have their own copy of bin counters which are stored in shared memory. After initializing each counter to zero, threads read input elements based on their global ID in grid. The stride between the elements that are read by a single thread is the product of the number of threads and number of blocks that are launched in kernel. For each element appropriate bin is found using binary search and the corresponding counter is atomically incremented. At the end, global bin counters are atomically updated from each thread block. There are several observations that should be made about this algorithm.

```
__global__ void privatization(T* input, int N,
                  float* bw, uint* bc, int B) {
  __shared__ uint subhist[];
  int tid = threadIdx.x;
  int gid = threadIdx.x +
            blockIdx.x * blockDim.x;
  // Initialization
  while(tid < B) {
    subhist[tid] = 0;
    tid += blockDim.x
  }
  __syncthreads();
  // Calculate private histogram
  int stride = blockDim.x * gridDim.x;
  int b;
  T* current = input + gid;
  while(current < input + N) {
    b = find_bin_binary(*current, B, bw);
    atomicAdd(&subhist[b], 1);
    current += stride;
  }
  __syncthreads();
  // Update global histogram
  tid = threadIdx.x;
  while(tid < B) {
    atomicAdd(&bc[tid], subhist[tid]);
    tid += blockDim.x;
  }
}
```

Listing 2: CUDA kernel of privatization algorithm.

First one is the obvious limitation on the maximum number of bins that this algorithm supports. With the latest CUDA GPU devices, the size of shared memory is 48 KB per block, and when unsigned integers are used as bin counters this leads to a maximum of around 10K bins in the private copy. This number of bins will satisfy most of the today's applications [8] [9]. Still, when the higher number of bins is required, there are several modifications that could be applied [6] [7]. For example, instead of using unsigned integers we could use unsigned chars or other data structure with smaller number of bits per bin counter in the private copy. However, using smaller data type may lead to overflow during the computation of the histogram, and this brings additional overhead in updating corresponding global counter each time when it occurs. Another approach is to launch the kernel several times, each time for different set of bins which also leads to overheads due to multiple launches of the kernel.

Second observation is that the algorithm uses atomic operations in each computational phase. We have tried to implement several modifications in order to reduce the number of atomic instructions. One of them is allocating private copy of bin counters per thread instead of per block. Although this technique completely eliminates atomic instructions, it limits the maximum number of bins to around 50, rendering the algorithm unusable. Instead, we tried another technique where the number of blocks is equal to the number of bins. In this way each block counts only elements that fit into its bin and each thread within this block has private counters only for the bin of block it belongs to. This eliminates the need both for atomic instructions and for binary search. Unfortunately, it brings additional overheads in parallel reduction of per-thread bin counters and increased number of accesses to global memory (since the elements are read with stride equal to the number of threads per block). Due to these overheads this technique does not give us improvement in speedup.

Last comment refer influence of consecutively reading of input elements by threads within block. The basic and most important idea is to have coalesced reading since global memory is banked. This means that neighbouring threads should access adjacent memory banks which can be done accessing memory locations stridden by multiple of number of banks. Special case is when stride is equal to one, which means that threads read from adjacent memory locations. This can be very important because our algorithm is sensitive on distribution of input elements. For those with random values taken from uniform distribution it is not important which stride is used since there is high probability that neighbouring threads will find different bins for their elements thus not waiting for atomic update of counter. But if input elements present pixels of a real image for example it is highly probable that adjacent memory locations will store pixels that belong to same bin. In that case it is better for neighbouring threads to read elements that are distanced with stride larger than one but still from adjacent memory banks.

C. Sort-search algorithm

Our second strategy, called sort-search, takes a different approach to create the histogram. Instead of going through the input collection, finding and updating bin for each element, this approach sorts input elements and then searches for the position of upper bounds among sorted elements according to bin widths (Figure 2). Implementation of this strategy is based on example provided by developers of Thrust library [10] with modifications to support histograms with custom bin widths. Unfortunately, functions from the Thrust library are not compatible with CUDA streams and cannot use asynchronous calls. Hence, this approach cannot overlap data transfers with

```
template<typename T>
void sort_search(T* input, int N, float* bw,
                 uint* bc, int B) {

  //prepare vectors on GPU device
  thrust::device_vector<T> in_d(N);
  thrust::device_vector<float> bw_d(B+1);
  thrust::device_vector<uint> bc_d(B);

  //initiate them with proper data
  thrust::copy(in_d.begin(),in_d.end(),input);
  thrust::copy(bw_d.begin(),bw_d.end(),bw);

  //sort the input elements
  thrust::sort(in_d.begin(), in_d.end());

  //find upper bounds thus creating cumulative
  //histogram and subtract (i-1)th from (i)th
  //counter
  thrust::upper_bound(in_d.begin(), in_d.end(),
        bw_d.begin, bw_d.end(), bc_d.begin());
  thrust::adjecent_difference(bc_d.begin(),
        bc_d.end(), bc_d.begin());

  //copy counters back
  thrust::copy(bc, bc + B, bc_d.begin());
}
```

Listing 3: Implementation of sort-search algorithm.

computation and cannot be used in multi-GPU scenarios. The complete source code for this algorithms is provided in Listing 3.

The problem that could arise with this approach is that the time consumption of the sorting part of algorithm depends on the internal distribution and arrangement of input elements. Whenever it is possible, Thrust uses fast and efficient radix-sort [11] and then input distribution cannot affect the sort performance. Using real numbers as input elements forces Thrust to change the sorting implementation to other methods less efficient than radix-sort.

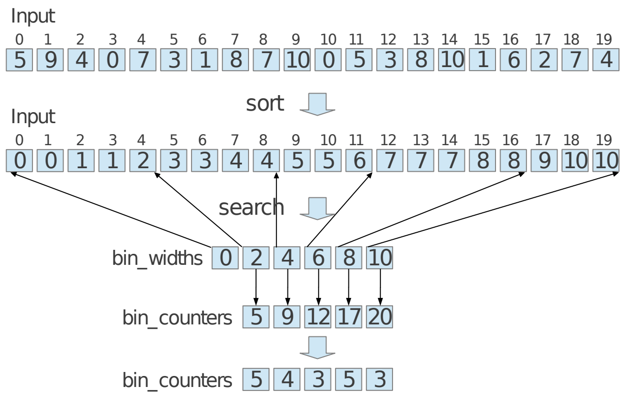


Fig. 2: Graphical representation of sort-search algorithm. For simplicity this example assumes $N = 20$ and $B = 5$.

D. Programming effort

Quantifying the effort required for parallelizing applications is difficult because it is dependent on many factors. Still, with Listings 2 and 3 given above it is obvious that programming effort in implementing histogram application is significantly reduced when we use second approach. This is expected since Thrust is a template library made to implement high-performance applications with minimal programming effort. That is also one of the main reasons why we use it in our implementation.

IV. Results

Here we provide a wide set of results we obtained. First we present hardware platforms used in our measurements as well as methodology. After that, we analyze speedup dependencies on the input size, the number of bins, type and the distribution of input elements. At the end, we give optimal-maps for both Fermi and Kepler architectures.

A. Hardware platforms

Our baseline sequential algorithm is executed on an Intel Core i7-2760QM CPU with 4 superscalar cores each running at 2.8GHz and with 8GB of RAM memory, 6MB of Intel Smart cache memory and 500GB of local disk storage. Program is compiled with the GCC C++ compiler using maximum optimization for performance with auto-vectorization enabled. For CUDA execution time we used NVIDIA Tesla C2070 (Fermi architecture) and Tesla K20c (Kepler architecture) GPUs each with 6GB of GDDR5 memory. These systems are running on 64-bit Linux kernel with Debian distribution.

B. Methodology

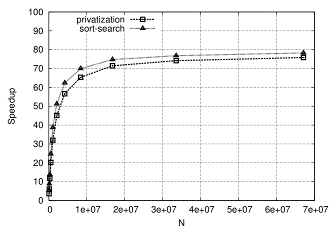
We analyze the speedup with respect to the sequential version, and how it depends on input size, number of bins and on the type and distribution of input elements. The speedup is determined as:

$$Speedup = \frac{T_{CPU}}{T_{GPU}}$$

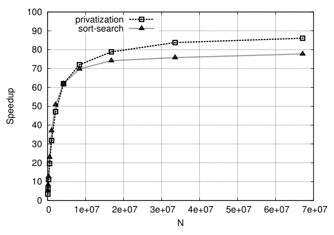
where T_{CPU} is time measured with sequential implementation, and T_{GPU} is the time on the GPU device (including data transfers and kernel execution time).

C. Input size

Figure 3 shows the execution time speedup of each GPU implementation compared to a sequential implementation on a CPU. The speedup of the implementation using privatization and atomic updates on shared and global memory scales logarithmically with the number of input elements until about four million elements, where the speedup starts becoming flat. The speedup increases due to the ability of the GPU hardware of scheduling new thread-blocks as work is being committed, so the GPU resources are fully occupied during most of the execution time. However, the speedup becomes flat when the number of elements reaches about eight million, as a consequence of the usage of atomic operations. The larger the number of elements, the greater the chances of serialization when performing atomic updates on both shared and global memory and, hence, the benefits of parallel execution are voided by the serialization of memory accesses. The overhead due to atomic operations starts harming the overall performance when the number of elements is about thirty million where the probability of serialization during atomic operations is very high. Comparing obtained speedups on Fermi and Kepler architectures, we can observe higher speedup on Kepler. Reason for such a behaviour is improved shared memory latency on Kepler architecture.



(a) on Fermi



(b) on Kepler

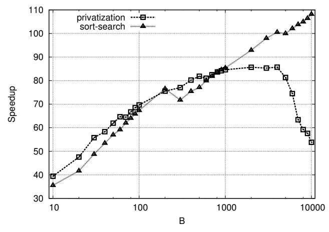
Fig. 3: Dependence of speedup on input size for both privatization and sort-search algorithm. Number of bins is $B = 256$, input distribution is uniform and elements are unsigned integer type.

The implementation based on pre-sorting the input to avoid atomic operations offers a worse performance than the previous approach for input sizes smaller than about six million elements. This is a direct consequence of the computational cost of the sorting stage ($O(N \cdot \log(N))$), and the overhead of executing two GPU kernels instead of one. However, this approach performs better than using privatization when the number of elements is very large, despite its higher algorithmic complexity, because there is no serialization of memory accesses. The speedup of this approach also starts becoming flat when the number of elements becomes thirty-two million. In this case the speedup does not improve because the increasing cost of the sorting stage which depends linearly with the input size, while the parallelism of the second stage does not increase because it depends on the number of bins. Interesentingly, there is no difference between Fermi and Kepler architectures when this implementation is used. Reason for this may be a not updated Thrust library for new Kepler architecure.

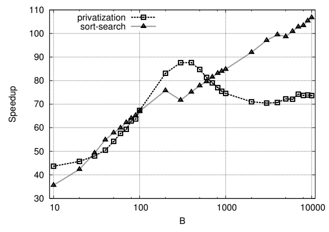
These experimental results show that the optimal histogramming implementation depends on the size of the input data, as well as the distribution and number of bins. A small number of bins and/or a distribution where most input values fall in a narrow range increase the chances of serialization of atomic updates, so the cost pre-sorting the input pays off. However, if the number of bins is large and/or the input values are uniformly distributed, the small number of conflicting atomic operations favors the implementation using privatization and atomic updates of shared and global memory.

D. Number of bins

Fixing the input size and varying number of bins we observe interesting speedup dependencies showed on Figure 4. On Fermi architecture, speedups are almost equal until we reach number of bins of 1K after which speedup starts becoming flat. Reason for that is increased control divergence since each thread is independently searching for appropriate bin of current elements. After we reach the number of bins of 5K speedup has a sudden decrease. It is because of reduced occupancy of CUDA streaming multiprocessors due to shared memory consumption by one block. Since shared memory has a limited size, for histograms with $B > 6K$ one streaming multiprocessor can handle just one block of threads. Due to maximum number of threads per block is 1024 that is also the total number of threads that are executed by one streaming multiprocessor instead of 1536 giving us occupancy reduced to 66%. Here we can observe clear advantage of sort-search approach over privatization. Not just that it's speedup continues to increase with number of bins but it is not limited by that parameter at all.



(a) on Fermi



(b) on Kepler

Fig. 4: Dependence of speedup on number of bins for input size $N = 2^{26}$ integer elements taken from uniform distribution.

With Kepler, dependency line is slightly different. There is decrease of speedup when number of bins is about 200. This is a direct consequence of having more cores on one SM, from 32 in Fermi to 192 in Kepler, so now up to 192 threads can run concurrently. With a simple calculation, concerning that input elements are randomly distributed, we observe that for $B = 300$ for example, the probability of serialization caused by updating same bin counter in Kepler is $\approx 63.8\%$, while in Fermi is $\approx 10.3\%$. That is the reason why execution time increases and thus speedup decreases.

E. Type of input elements

In order to see the impact of input data type, we study how speedup changes with respect to the number of bins for three different data types: unsigned characters, integers and double precision floating point numbers. Results are presented in Figures 5 and 6.

When using 8-bit unsigned characters, sort-search algorithm outperforms privatization. Due to the limited maximum value that unsigned character can have, the number of bins is up to $2^8 - 1$. When it comes to Fermi and using integers as input elements, both algorithms give almost the same speedup for up to 1000 bins, which is the previously explained threshold. Having input with double precision elements gives advantage to privatization algorithm over sort-search. This could be expected since fast and efficient implementation of radix-sort provided by Thrust library can not be applied to double precision numbers and this degrades the overall performance of sort-search. The same situation is with Kepler.

F. Distribution of input elements

As we already mentioned, both privatization and sort-search algorithms depend on the distribution of input elements.

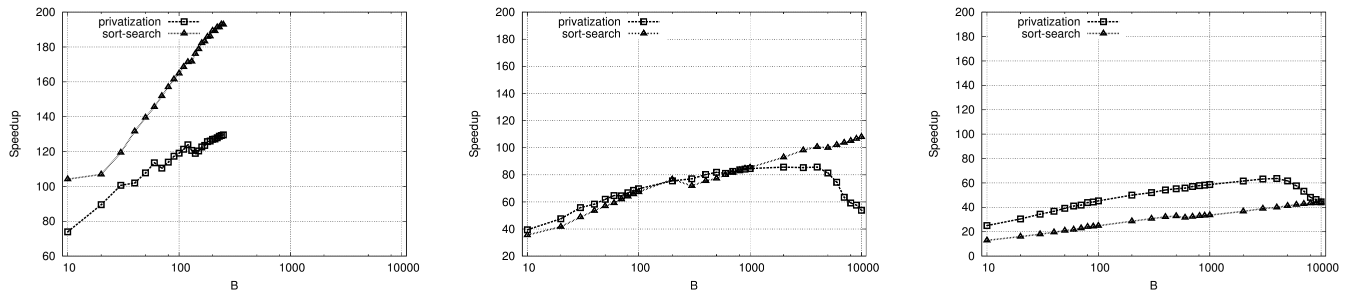

(a) unsigned characters (b) integers (c) doubles

Fig. 5: Speedup dependencies on number of bins for three different types of input elements. Input distribution is uniform and size is $N = 2^{26}$. Results are obtained using Fermi architecture.

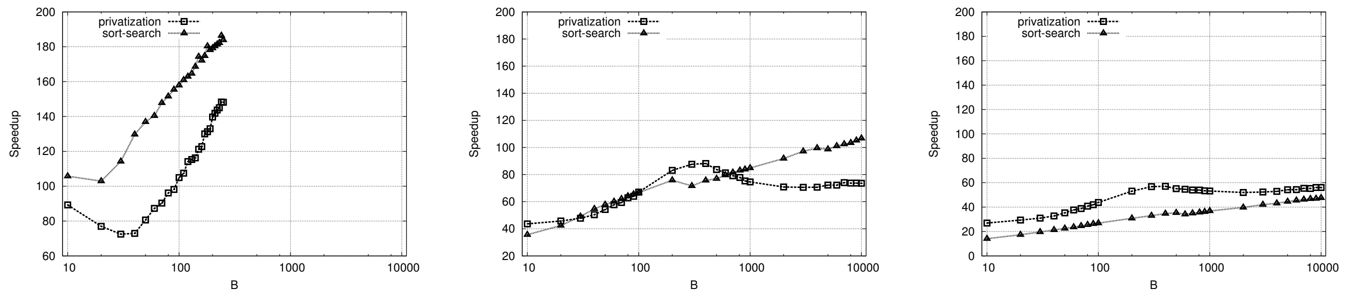

(a) unsigned characters (b) integers (c) doubles

Fig. 6: Speedup dependencies on number of bins for three different types of input elements. Input distribution is uniform and size is $N = 2^{26}$. Results are obtained using Kepler architecture.

In order to try to quantify this dependence, we provide analyze how speedup depends on input data distribution, using three different distributions (histograms are shown on Figure 7).

The first distribution that we use is the uniform distribution, one we used for all results presented up to this point. Then, we include normal distributions with small standard deviation (σ). For the experiments, we take elements with random values from these distributions. Finally, we use again uniform distribution but now with sorted chunks of elements.

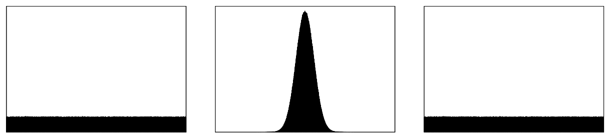

(a) (b) (c)

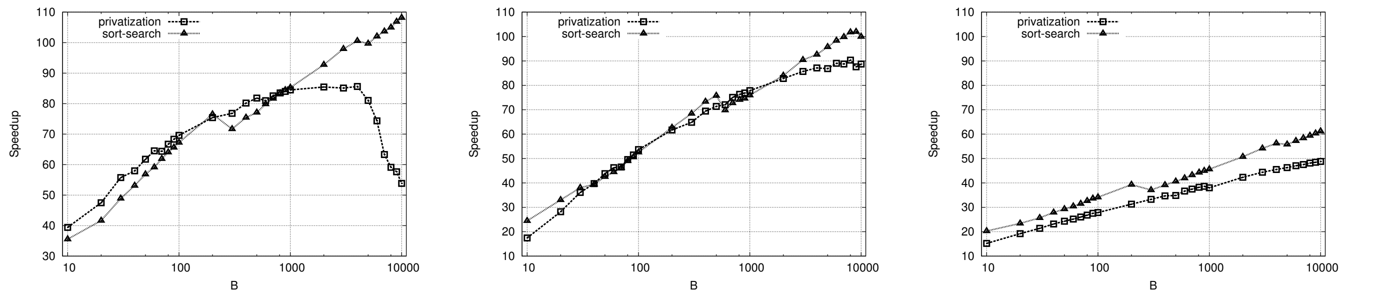
Fig. 7: Histograms of three different input distributions used in this paper: uniform distribution (a), normal distribution with small σ (b) and uniform distribution with sorted elements (c).

In case of sort-search algorithm, all distributions have the same effect. This is expected since we use integers as input elements and radix-sort does not depend on the input distribution. On the other side, privatization algorithm behaves differently for different distributions when we have more than 5K bins in Fermi. In the case of using uniform (Figure 8a), the decrease in speedup is much higher than in the case of using a normal distribution (Figure 8b). This is an example on the big impact that control divergence can have on performance of CUDA kernels. When we have "wide" range of bins (as in Figure 7a), threads within a warp are taking different execution paths in their binary searches, thus increasing execution time. On the other hand, having "narrow" range of bins where input elements can fit in, it is more likely that threads within a warp will take the same execution path searching for a bin of a current element. Extreme example is shown in Figure 8c where input elements are sorted and control divergence almost completely removed. The same influence can be observed in case of using Kepler architecture (Figure 9b). The highest obtained speedup is shifted towards higher number of bins. Impact of serialization is not dominant here as on Figure 9a because it is constantly present (threads are waiting on atomic operation in order to update current bin counter). Figure looks more like one when we are using Fermi architecture, having decrease of speedup after $B = 5K$ because of control divergence.

We also tested both algorithms for other input distributions like normal distribution but with bigger σ, sequence of several normal distributions, image-like distributions etc. In all those cases dependencies of speedup on number of bins look the same as one of the three already given on Figures 8 and 9.

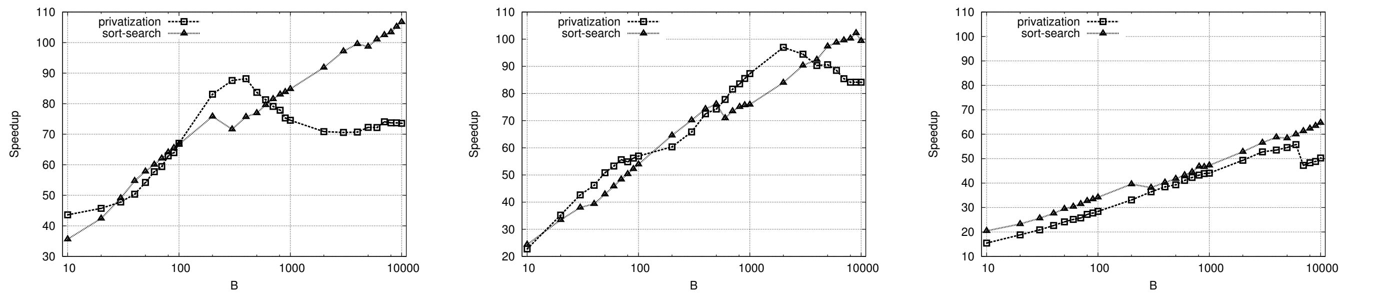
G. Overlapping

In order to further increase speedup over sequential execution we improved the privatization algorithm to take advantage



(a) uniform distribution (b) normal distribution (small σ) (c) uniform distribution (sorted)

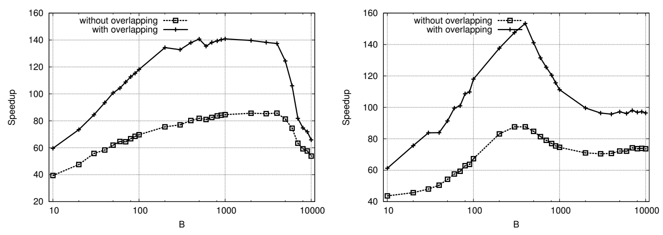
Fig. 8: Speedup dependencies on number of bins for three input distribution shown on Figure 7. Input contains $N = 2^{26}$ integers. Results are obtained using Fermi architecture.



(a) uniform distribution (b) normal distribution (smaller σ) (c) uniform distribution (sorted)

Fig. 9: Speedup dependencies on number of bins for three input distribution shown on Figure 7. Input contains $N = 2^{26}$ integers. Results are obtained using Kepler architecture.

of the possibilities for overlapping data transfers with kernel execution. To do this, we use different streams for transfers and asynchronous function calls. In ideal case, when memory transfer takes almost the same time as kernel execution, we can obtain a 2x increase in performance when overlapping is used. We implement this technique breaking input collection into a number of chunks. As soon as we store one chunk of input data in the CUDA global memory, we can launch kernel that will create subhistogram based just on that chunk. While kernel is executing, next chunk can be transfered thus overlapping memory transfers and kernel executions. Figure 10 shows dependencies of speedup on number of bins in cases with and without overlapping.



(a) on Fermi (b) on Kepler

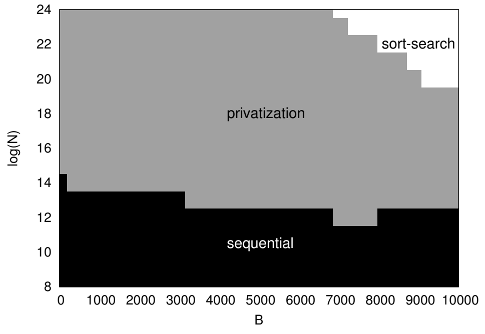
Fig. 10: Dependence of speedup on number of bins with and without overlapping. Input size is $N = 2^{26}$ and elements are integers with values taken from uniform distribution.

H. General overview

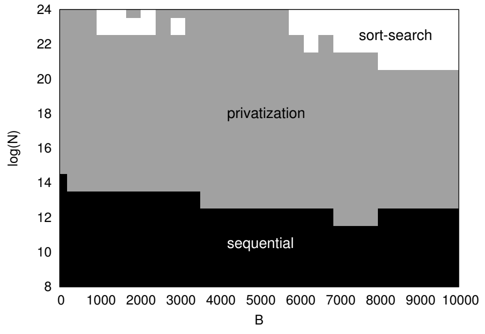
Based on these results, we can see that it is not easy to conclude which algorithm is the best option for achieving the maximum performance. Hence, we have performed an exploration to determine the algorithm that should be used depending on the input parameters (size of the input array and the number of bins). Figure 11 shows this exploration for both the Fermi and the Kepler architecture (for each combination of the two input parameters B and N). This is the optimal-map for integers as input elements which are taken randomly from uniform distribution. As we can see, because of additional overheads of transfering data from the host to the device memory and launching the kernel, for some histograms with small number of bins and small input collection the best solution is to use sequential version executed on general purpose CPU. In case where histogram is generated using input collection with size higher than about 2^{20} and number of bins above 5K the fastest way to generate histogram is by using sort-search implementation. In all other cases best solution is using privatization algorithm on GPU.

V. Related work

So far, several different algorithms for histogramming were suggested. One of them is a part of NVIDIA's SDK



(a) on Fermi



(b) on Kepler

Fig. 11: Map of the input space (B on the x-axis and $\log(N)$ on y-axis) and the selection of the optimal algorithm. Elements are integers with values taken from uniform distribution.

implemented by Podlozhnyuk [12]. Although very optimized, such implementation is not general enough since it is not able to support number of bins bigger than 256 and it assumes that bin widths are equal and discrete. The second assumption also limits another algorithm that is suggested in [7]. Our first algorithm is based on concepts used in those two papers, but is improved to eliminate the limitations of original algorithms. On the other hand, completely different approach is shown in examples provided by developers of Thrust library [10]. Our second strategy relies on that approach.

VI. Conclusions

In this paper we have analyzed the two dense histogramming algorithms in CUDA. We have described an implementation of both algorithms that has no limitations in terms of bin widths, input distribution or data types. We have also performed an experimental evaluation of the performance of each implementation depending on the characteristics of the input data and the kind of histogram to be produced.

Our analysis has presented the trade-offs of each implementation. Using privatization and atomic updates tends to overperform the implementation based on pre-sorting the input when the probability of collision during atomic updates is low. This situation happens, for instance, when the number of input elements is low, the number of bins high, and the data is uniformly distributed. However, if the chances of serialization due to atomic updates is high, pre-sorting the input data to avoid this situation pays off. This shows that there is no silver bullet for a CUDA implementation of histogramming, but heavily depends on the characteristics of the application.

The results presented in this paper show that a simple histogramming library call for CUDA is not likely to be enough to provide the best performance in all applications. Consequently, we have presented an extensive analysis of the speedup depending on the input parameters, that can lead the selection of the optimal algorithm for the task. Based on these experiences, we plan to design a histogramming template library where the optimal strategy is selected based on the template parameters. Moreover, a dynamic version of the histogram call will be also provided for those applications where the characteristics of the input data are not known at compile time.

Acknowledgements

The research leading to these results is supported by Mont-Blanc project (European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement n^o 288777), the grant SEV-2011-0067 of Severo Ochoa Program awarded by the Spanish Government, the European Network of Excellence HIPEAC-2 (ICT-FP7-217068) and NVIDIA CUDA Center of Excellence at BSC/UPC. The authors thank the reviewers for their guidance and comments to improve quality of this paper.

References

[1] N. R. Tague, "Seven basic quality tools," *The Quality Toolbox. Milwaukee, Wisconsin: American Society for Quality*, p. 15, 2004.

[2] NVIDIA, "Compute unified device architecture–reference manual," *NVIDIA Corporation*, 2008.

[3] C. Nvidia, "C programming guide 3.2," *NVIDIA Corporation, Nov*, 2010.

[4] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi gf100 gpu architecture," *Micro, IEEE*, vol. 31, no. 2, pp. 50–59, 2011.

[5] NVIDIA, "Geforce gtx 680: The fastest, most efficient gpu ever built," *NVIDIA Corporation*, 2012.

[6] T. Brosch and R. Tarn, "A self-optimizing histogram algorithm for graphics card accelerated image registration," in *Medical Image Comput ing and Computer Assisted Intervention (MICCAI) Grid Workshop*, 2009, pp. 35–44.

[7] R. Shams and R. Kennedy, "Efficient histogram algorithms for nvidia cuda compatible devices," in *International Conference on Signal Processing and Communication Systems*, vol. 3, 2007.

[8] A. Collignon, F. Maes, D. Delaere, D. Vandermeulen, P. Suetens, and G. Marchal, "Automated multi-modality image registration based on information theory," in *Information processing in medical imaging*, vol. 3, 1995, pp. 264–274.

[9] P. Viola and W. Wells III, "Alignment by maximization of mutual information," *International journal of computer vision*, vol. 24, no. 2, pp. 137–154, 1997.

[10] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for cuda," *GPU Computing Gems*, pp. 359–371, 2011.

[11] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore gpus," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–10.

[12] V. Podlozhnyuk, "Histogram calculation in cuda," *NVIDIA Corporation, White Paper*, 2007.