Efficient Histogram Algorithms for NVIDIA CUDA Compatible Devices

Ramtin Shams* and R. A. Kennedy*
*Research School of Information Sciences and Engineering (RSISE)
The Australian National University
Canberra ACT 0200 Australia

Abstract— **We present two efficient histogram algorithms designed for NVIDIA's** *compute unified device architecture* **(CUDA) compatible** *graphics processor units* **(GPUs). Our algorithm can be used for parallel computation of histograms on large data-sets and for thousands of bins. Traditionally histogram computation has been difficult and inefficient on the GPU. This often means that GPU-based implementation of the algorithms that require histogram calculation as part of their computation, require to transfer data between the GPU and the host memory, which can be a significant bottleneck. Our algorithms remove the need for such costly data transfers by allowing efficient histogram calculation on the GPU. We show that the speed of histogram calculations can be improved by up to 30 times compared to a CPU-based implementation.**

Index Terms— **Histogram, Parallel processing, Compute unified device architecture (CUDA), Graphics processor unit (GPU)**

I. INTRODUCTION

The histogram is a non-parametric density estimator which provides a consistent estimate of the probability density function (pfd) of the data being analyzed [1], [2]. The histogram is a fundamental statistical tool for data analysis which is used as an integral part of many scientific computational algorithms.

Recent advances in *graphics processor units* (GPUs), most notably the introduction of the *compute unified device architecture* (CUDA) by NVIDIA [3], allows implementation of non-graphical and general purpose algorithms on the GPU. GPUs, like NVIDIA 8800 GTX, are massively multi-threaded single instruction multiple data (SIMD) devices and are optimized for floating point calculations. Due to its architecture, the GPU is a natural candidate for implementation of many scientific applications, which require high-precision and efficient processing of large amounts of data.

A. Background and Motivation

Unfortunately, the higher processing power of the GPU compared to the standard *central processor unit* (CPU), comes at the cost of reduced data caching and flow control logic as more transistors have to be devoted to data processing. This imposes certain limitations in terms of how an application may access memory and implement flow control. As a result,

implementation of certain algorithms (even trivial ones) on the GPU may be difficult or may not be computationally justified.

Histogram has been traditionally difficult to compute efficiently on the GPU [4]. Lack of an efficient histogram method on the GPU, often requires the programmer to move the data back from the device (GPU) memory to the host (CPU), resulting in costly data transfers and reduced efficiency. A simple histogram computation can indeed become the bottleneck of an otherwise efficient application.

Currently, there is only one efficient histogram method available for CUDA compatible devices [4]. The histogram is limited to 64 bins, which is too small for many real-life applications. For example, 2D histogram calculations used in estimating the joint pdf of pixel intensities, commonly used in mutual information ([5])-based image registration methods (*e.g.* [6], [7], [8], [9], [10]), typically require in the order of $10,000$ bins.

B. Method and Contribution

We present two efficient histogram methods for CUDA compatible devices which can be used for any number of bins. The histogram methods are designed for NVIDIA 8-series GPUs of *'compute capability'* 1.0[1]. The GPU does not support atomic updates of the device's global memory or shared memory. It also lacks *mutual exclusion* (mutex) and *critical section* thread synchronization primitives which are required for safe access to shared objects by multiple threads. The only synchronization facility offered by the GPU is the *thread join* which only works among the threads of the same *thread block*.

To overcome lack of synchronization primitives, we investigate two strategies for histogram implementation. The first method is based on simulating a mutex by tagging the memory location and continuing to update the memory until the data is successfully written and the tag is preserved. The second method maintains a histogram matrix of $B \times N$ size, where B is the number of bins and N is the number of threads. This provides a collision free structure for memory updates by each thread. A parallel reduction is ultimately performed on the matrix to combine data counters along the rows and produce the final histogram. Various techniques are used to

---

[1]Unless otherwise noted, use of the term *GPU* in the remainder of this paper refers to this class of devices.

minimize access to the GPU's global memory and optimize the *kernel* code for the best performance.

II. CONCEPTS

A. An Overview of CUDA

We provide a quick overview of the terminology, main features, and limitations of CUDA. More information can be found in [3]. A reader who is familiar with CUDA may skip this section.

CUDA can be used to offload data-parallel and compute-intensive tasks to the GPU. The computation is distributed in a *grid* of *thread blocks*. All blocks contain the same number of threads that execute a program on the *device*, known as the *kernel*. Each block is identified by a two-dimensional block ID and each thread within a block can be identified by an up to three-dimensional ID for easy indexing of the data being processed. The block and grid dimensions, which are collectively known as the *execution configuration*, can be set at run-time and are typically based on the size and dimensions of the data to be processed.

It is useful to think of a grid as a logical representation of the GPU itself, a block as a logical representation of a multi-core processor of the GPU and a thread as a logical representation of a processor core in a multi-processor. Blocks are time-sliced onto multi-processors. Each block is always executed by the same multi-processor. Threads within a block are grouped into *warps*. At any one time a multi-processor executes a single warp. All threads of a warp execute the same instruction but operate on different data.

While the threads within a block can co-operate through a cached but small *shared* memory (16 KB), a major limitation is the lack of a similar mechanism for safe co-operation between the blocks. This makes implementation of certain programs such as a histogram difficult and rather inefficient.

The device's DRAM, the *global memory*, is un-cached. Access to global memory has a high latency (in the order of 400-600 clock cycles), which makes reading from and writing to the global memory particulary expensive. However, the latency can be hidden by carefully designing the kernel and the execution configuration. One typically needs a high density of arithmetic instructions per memory access and an execution configuration that allows for hundreds of blocks and several hundred threads per block. This allows the GPU to perform arithmetic operations while certain threads are waiting for the global memory to be accessed. The performance of global memory accesses can be severely reduced unless access to adjacent memory locations is *coalesced*[2] for the threads of a warp (subject to certain alignment requirements).

The data is transferred between the host and the device via the *direct memory access* (DMA), however, transfers within the device memory are much faster. To give reader an idea, device to device transfers on 8800 GTX are around 70 Gb/s[3],

whereas, host to device transfers can be around $2-3$ Gb/s. As a general rule, host to device memory transfers should be minimized where possible. One should also batch several smaller data transfers into a single transfer.

Shared memory is divided into a number of banks that can be read simultaneously. The efficiency of a kernel can be significantly improved by taking advantage of parallel access to shared memory and by avoiding bank conflicts.

A typical CUDA implementation consists of the following stages:

1) Allocate data on the device.
2) Transfer data from the host to the device.
3) Initialize device memory if required.
4) Determine the execution configuration.
5) Execute kernel(s). The result is stored in the device memory.
6) Transfer data from the device to the host.

The efficiency of iterative or multi-phase algorithms can be improved if all the computation can be performed in the GPU, so that step 5 can be run several times without the need to transfer the data between the device and the host.

B. Problem Statement

Histogram calculation is straightforward on a sequential processor as shown in Listing 1.

```
1  for (i = 0; i < data_len; i++)
2  {
3      //'data[]' is normalized between 0.0 and 1.0.
4      bin = data[i] * (bins − 1);
5      //'histogram[]' is already initialized to zero.
6      histogram[bin]++;
7  }
```

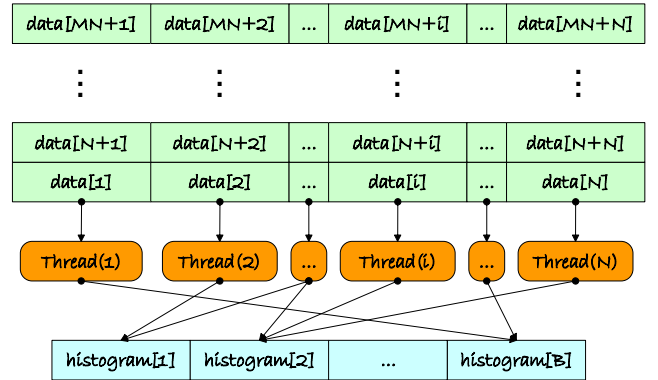Listing 1.   A simple histogram code snippet for a sequential processor.

Fig. 1.   Parallel calculation of a histogram with N threads. Histogram updates conflict and require synchronization of the threads or atomic updates to the histogram memory.

Parallelizing a histogram with B bins over N threads is schematically shown in Fig. 1. The input data is distributed among the threads. Updates to histogram memory is data dependent as such, many threads may attempt to update the same location of the memory resulting in read/write conflicts.

---

[2]Memory accesses are coalesced if for each thread i within the half-warp the memory location being accessed is '$baseAddress[i]$', where '$baseAddress$' complies with the alignment requirements.

[3]Gigabits per second

Since the GPU lacks native mutex synchronization and atomic updates to its memory, we propose the following two methods to avoid the concurrent update problem.

III. METHOD

A. Method 1: Simulating Atomic Updates in Software

The GPU executes the same instruction for all the threads of a warp. This allows simulating an atomic update by tagging the data that is being written to with the thread ID within the warp and repeatedly writing to the location until the tagged value can be read without change as shown in Listing 2.

```
1  //'bin' is defined as 'volatile' to prevent the comp-
2  //iler from optimizing away the comparison in line 13.
3  volatile unsigned int bin;
4  unsigned int tagged;
5  bin = (unsigned int) (data[i] * (bins - 1));
6  do
7  {
8      unsigned int val = histogram[bin] & 0x07FFFFFF;
9      //The lower 5 bits of the thread id (tid) are
10     //used to tag the memory location.
11     tagged = (tid << 27) | (val + 1);
12     histogram[bin] = tagged;
13 } while (histogram[bin] != tagged);
```

Listing 2. Simulating atomic updates to shared memory for threads that belong to the same warp.

For this method to work one needs to ensure that 'histogram[]' array can only be updated by one warp and the frequency of samples does not exceed $32 - \log_2 w$, where w is the warp size. For 8800 GTX, the warp size is 32 and the frequency of samples cannot exceed 2^{27}.

At each iteration, at least one of the threads within the warp is guaranteed to succeed [3], so the worst-case scenario is that the loop has to be executed w times. This happens when all the threads happen to be writing to the same bin.

'histogram[]' has to be allocated on the GPU's shared memory for this method to be efficient. However, given the 4K double word (32-bit) limit for the shared memory, the maximum number of bins that can be supported is 4096[4]. We caution against allocating 'histogram[]' in the global memory to overcome this limit, as repeated updates to the high latency global memory will significantly reduce the execution speed.

We also note that using a single warp under-utilizes the GPU resources. For optimal performance the GPU needs around 4-8 warps. As such, we need to allocate a separate histogram array for each warp. The sub-histogram arrays are then combined to produce the final result. Obviously, increasing the number of warps will further limit the number of bins that can be processed per execution of the algorithm.

To allow calculation of an arbitrary number of bins, we sub-divide the bin ranges into a number of sub-ranges that fit in the shared memory. For a given execution configuration we run the algorithm as many times as required to cover the entire bin range. At each iteration the kernel will only process those data elements which fall in the specified bin range. For example,

---

[4]The actual number is slightly lower, since CUDA uses shared memory to pass arguments and execution information to the kernel.
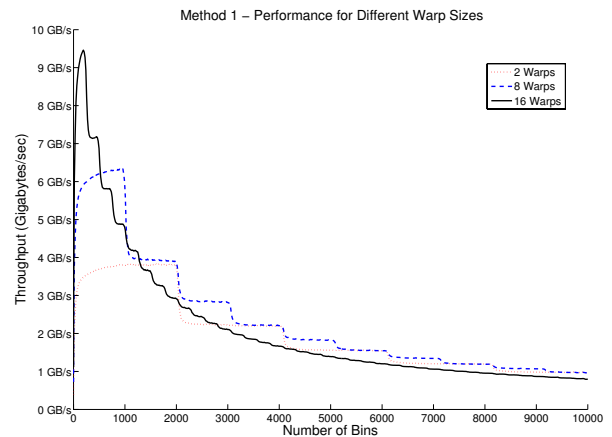

Fig. 2. The performance of the method is higher with more warps but drops more quickly with the number of bins. Lower number of warps perform better for bigger bins.

with 4 warps and a limit of 1024 bins per execution, a $10,000$ bin histogram requires 10 iterations of the algorithm.

Fig. 2 shows histogram throughput in gigabytes per second for 2, 8 and 16 warps. Histogram bins are varied from 1 to $10,000$. The input data is random with a uniform distribution. Higher number of warps result in improved performance for smaller bins but as the number of bins increases, the number of warps has to be reduced in order to achieve the highest throughput. As a result, we choose the optimum number of warps w_{opt} depending on the number of bins and such that

$$w_{\text{opt}} = \underset{w}{\operatorname{argmax}} wB, \ wB \leq s_{\max}, \ 3 \leq w \leq w_{\max}, \quad (1)$$

where w is the number of warps, B is the number of bins, $s_{\max}$ is the maximum number of double words that can be allocated in the shared memory, and $w_{\max}$ is the maximum number of warps supported by the hardware. We also note that the performance of the method with respect to the number of bins is almost piece-wise constant with the exception of small bins where the performance is lower due to increased chance of bin update collisions. The jumps in Fig. 2 correspond with the increasing number of iterations required for calculating the entire bin range.

The disadvantage of this method is that the throughput is dependent on the distribution of the data. In Fig. 3, we have shown the throughput of the method for a random input with a uniform distribution, a random input with a normal distribution, and for a degenerate distribution (*i.e.* all input elements are set to the same value). The random input with uniform distribution is close to the best case scenario, as for large inputs the histogram is going to be uniform and the histogram update collisions are close to minimal. A degenerate distribution results in maximum histogram update collisions, as all the threads try to update the same histogram bin and as such represents the worst case scenario. The performance for a real application is somewhere in between these lower and upper bounds and we have represented this with a random input with a normal distribution with mean 0 and standard
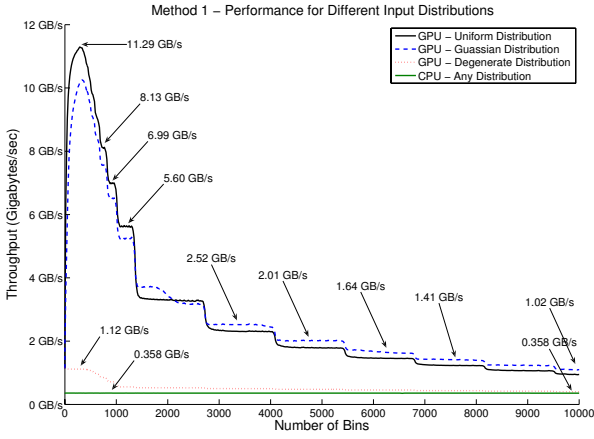
Fig. 3. The performance of method 1 is dependent on the distribution of the input data. Data with a uniform distribution performs (almost) best. The worst performance is observed for a degenerate distribution. The performance for practical applications is somewhere between these upper and lower bounds. Depending on the number of bins, a performance improvement of between 3-30 times is observed compared to the CPU implementation.
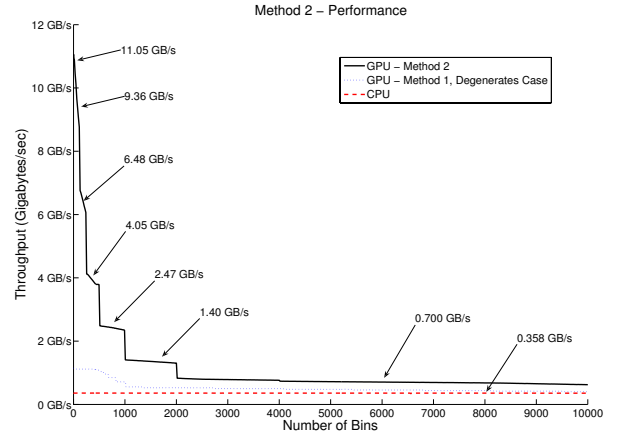

Fig. 4. The performance of method 2 is above the first method's worst case and the CPU-based histogram implementation. The maximum throughput is achieved for smaller bins with an improvement of up to 30 times compared to the CPU implementation.

deviation 1. For comparison, we have shown the throughput of histogram calculation on a high-end CPU (refer to Appendix I for specifications). The performance of the histogram on the CPU is almost constant w.r.t. the number of bins and is not affected by the distribution of the input data. Fig. 3 shows that the GPU implementation has a clear advantage, especially for smaller bins.

B. Method 2: Collision-Free Updates

As discussed, performance of the histogram calculation using the first method depends on the distribution of the input data. This may not be ideal for certain applications. In this section, we present and analyze an alternative method whose performance is not affected by the distribution of data. To this end, we allocate a histogram array per thread in the global memory. Then a partial histogram is calculated per thread and finally the partial histograms are reduced into a single histogram.

The benefit is that, given the size of the global memory, for any practical number of bins the algorithm only requires a single iteration to complete. In addition, there will be no concurrent updates of the same memory location by multiple threads and as such no update synchronization is required, which in turn means that the performance of the method is not data dependent.

However, there are two drawbacks to this method; firstly, a much larger memory for partial histograms needs to be allocated and initialized to zero at the beginning; secondly, histogram updates need to be done on the global memory, this entails non-coalesced read/writes per input data and is inefficient.

The standard memory initialization method that can be called from the host, 'cudaMemset' proved to be inefficient. We implemented a method for initializing floating point arrays

in the kernel with a throughput of around 35 Gb/s, which addressed the first problem.

To address the second problem, and avoid excessive non-coalesced updates of the global memory, we pack multiple bins in a double word in the shared memory and only update the corresponding bin in the global memory when the packed bin overflows. This reduces the updates to the global memory by a factor of 2^b, where b is the number of bits available for storage of a bin in the shared memory. b depends on the number of threads per block and the number of bins and is calculated as

$$b = \frac{s_{\max} \times 32}{B \times N_b}, \qquad (2)$$

where $s_{\max}$ is the maximum number of double words that can be allocated in the shared memory, B is the number of bins and N_b is the number of threads per block.

There is a trade-off between the number of threads and the number of bits per bin. Increasing the number of threads, decreases b, resulting in more global memory updates, while reducing the number of threads can under-utilize GPU resources and affect the performance. We optimized these parameters for the best performance, the result is shown in Fig. 4. As can be seen the second method performs consistently better than the first method's worst case. The performance of the second method for smaller bins is comparable with the first method's best case. However, this method under-performs the first method's best case for the mid and high bin ranges.

IV. DISCUSSION

The histogram implementations, presented in this paper, highlight some of the challenges, trade-offs and rewards in rethinking existing algorithms for a massively multi-threaded architecture, such as CUDA. The performance of the histogram calculation on a GPU can be up to 30 times more, compared to a CPU implementation. While the performance improvement diminishes with increasing number of bins, the improvement is

significant for a wide range of bins and consequently for many practical applications. Even when the performance of CPU and GPU implementations are comparable, the fact that the GPU implementation removes the need for data transfer between the device and the host, will still benefit applications that require histogram calculation as part of their implementation.

We recommend using the first method, where the distribution of the data is known to be *well behaved* such as a uniform or gaussian distribution (with not a too small standard deviation). However, when the distribution is close to a degenerate distribution or very sparse the second method will be more efficient.

In practice, one can choose the appropriate method by experimentation. The source code for the presented algorithms, as well as 2D histogram implementations, can be found online at http://cecs.anu.edu.au/~ramtin/cuda/.

In our opinion, CUDA is a promising technology built upon a solid hardware and software platform that can greatly benefit scientific and applied computing applications. The two major limitations of the current hardware are the small size of the shared memory and the lack of basic synchronization methods. While we recognize that addressing these limitations is no trivial task, we expect that future generations of the platform to provide further improvements and more flexibility over time.

V. Acknowledgement

Appendix I

Hardware Configuration

We used the following host and device configurations in our experiments.

TABLE I

HOST SPECIFICATION

Processor	AM2 Athlon 64×2 6000+ 3.0 GHz
Memory	4 GB, 800 MHz DDR2
Motherboard	ASUS M2N-SLI Deluxe
Graphics card	Leadtek 8800 GTX
Power supply	650 W

TABLE II

DEVICE SPECIFICATION (GPU)

Model	NVIDIA 8800 GTX
# of Multi-processors	16
# of cores per Multi-processor	8
Memory	768 MB
Shared memory per block	16 KB
Max # of threads per block	512
Warp size	32

References

[1] D. W. Scott, "On optimal and data-based hitograms," *Biometrika*, vol. 66, no. 3, pp. 605–610, Dec. 1979.

[2] K. H. Knuth, "Optimal data-based binning for histograms," *ArXiv Physics e-prints*, May 2006.

[3] *Compute Unified Device Architecture (CUDA) Programming Guide*. http://developer.nvidia.com/object/cuda.html: NVIDIA, 2007.

[4] V. Podlozhnyuk, "64-bin histogram," NVIDIA, Tech. Rep., 2007.

[5] C. E. Shannon, "A mathematical theory of communication," *Bell Syst. Tech. J.*, vol. 27, pp. 379–423/623–656, 1948.

[6] A. Collignon, F. Maes, D. Delaere, D. Vandermeulen, P. Suetens, and G. Marchal, "Automated multimodality medical image registration using information theory," in *Proc. Int. Conf. Information Processing in Med. Imaging: Computational Imaging and Vision 3*, Apr. 1995, pp. 263–274.

[7] P. Viola and W. M. Wells III, "Alignment by maximization of mutual information," in *Proc. Int. Conf. Computer Vision (ICCV)*, June 1995, pp. 16–23.

[8] R. Shams, R. A. Kennedy, and P. Sadeghi, "Efficient image registration by decoupled parameter estimation using gradient-based techniques and mutual information," in *Proc. IEEE Region 10 Conf. (TENCON)*, Taipei, Taiwan, Oct. 2007.

[9] R. Shams, P. Sadeghi, and R. A. Kennedy, "Gradient intensity: A new mutual information based registration method," in *Proc. IEEE Computer Vision and Pattern Recognition (CVPR) Workshop on Image Registration and Fusion*, Minneapolis, MN, June 2007.

[10] R. Shams, R. A. Kennedy, P. Sadeghi, and R. Hartley, "Gradient intensity-based registration of multi-modal images of the brain," in *Accepted for publication, Proc. IEEE Int. Conf. on Computer Vision (ICCV)*, Rio de Janeiro, Brazil, Oct. 2007.