High Performance Predictable Histogramming on GPUs: Exploring and Evaluating Algorithm Trade-offs

Cedric Nugteren Gert-Jan van den Braak Henk Corporaal Bart Mesman
Eindhoven University of Technology,
The Netherlands
{c.nugteren, g.j.w.v.d.braak, h.corporaal, b.mesman}@tue.nl

ABSTRACT

Graphics Processing Units (GPUs) are suitable for highly data parallel algorithms such as image processing, due to their massive parallel processing power. Many image processing applications use the histogramming algorithm, which fills a set of bins according to the frequency of occurrence of pixel values taken from an input image.

Histogramming has been mapped on a GPU prior to this work. Although significant research effort has been spent in optimizing the mapping, we show that the performance and performance predictability of existing methods can still be improved. In this paper, we present two novel histogramming methods, both achieving a higher performance and predictability than existing methods. We discuss performance limitations for both novel methods by exploring algorithm trade-offs.

Both the novel and the existing histogramming methods are evaluated for performance. The first novel method gives an average performance increase of 33% over existing methods for non-synthetic benchmarks. The second novel method gives an average performance increase of 56% over existing methods and guarantees to be fully data independent. While the second method is specifically designed for newer GPU architectures, the first method is also suitable for older architectures.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures

Keywords

Histogram, CUDA, GPU, Parallel processing, Image processing

1. INTRODUCTION

Many data-mining and image processing applications use histograms for data analysis. Histogramming creates *bins* which are filled according to the frequency of occurrence in the data-set. In the field of image processing, histogramming is used among others for image enhancement, image segmentation and image compression applications [1] [5]. Devices such as cameras, televisions, printers, displays and mobile devices use histogramming for a variety of purposes [5].

Although histograms are trivial to compute on a sequential processor, computation on a parallel processor is not straightforward. Many-core architectures such as the GPU (Graphics Processing Unit) are becoming increasingly important and are nowadays commonly available in personal computers [12]. Compared to a traditional CPU architecture, a GPU dedicates a large amount of its chip area to small processing elements, while using a relatively small amount of chip area for control logic and caches. A GPU contains a number of multiprocessors, each executing instructions in an SIMT (Single Instruction, Multiple Thread) manner and dividing the workload over a number of smaller processing elements.

Since GPUs are popular within the field of image processing [11], the question arises as to how the histogram algorithm should be adjusted to make use of a GPU's hardware efficiently. A number of existing GPU implementations are already available, but perform at best at 13% of the peak performance of the GPU[1] [13]. Additionally, they show a large variance with worst-case execution times a factor of 10 higher than best-case execution times, limiting the suitability for real-time applications.

To illustrate the need to improve the histogram algorithm, we consider processing a video stream with a resolution of 1920x1080. To achieve 60 frames per second, we require all processing to be completed in ≈16ms. Performing histogramming on a high-end GPU gives us a worst case execution time of ≈2ms, already limiting possibilities for further processing. A low-end GPU has a worst-case execution time of ≈25ms, and can thus not meet throughput requirements.

In this paper, we introduce existing mappings and present two novel histogramming methods. We evaluate the performance trade-offs by exploring a number of different mappings. The main contributions of this paper are:

- We present two novel histogramming methods targeted at GPUs. Both methods outperform existing methods and increase performance predictability. While the first method is more suitable for older NVIDIA GPU architectures, the second method performs better on newer architectures[2].

- We explore and discuss algorithmic design choices for both methods, and we identify and evaluate perfor-

[1]Tested using a GeForce GTX470, 8-bit inputs, and 256 bins
[2]Older: G80/G92/GT200. Newer: GF100 (Fermi)

mance limitations.

This paper is organized as follows. First, we discuss related work. Then, we give a brief background on GPU programming in section 3. Section 4 introduce the state-of-the-art histogramming method. Section 5 briefly presents the benchmark set and the used hardware. Following, we present *warp private histogramming* and *thread private histogramming* in sections 6 and 7. Section 8 presents benchmark results and discusses performance limits. Finally, section 9 concludes the paper.

2. RELATED WORK

Existing work describes a number of histogramming implementations using the CUDA GPU programming framework [9]. Podlozhnyuk presents two implementations in [13], one for 64-bin and one for 256-bin histograms. Both implementations are included in the CUDA software development kit. Two other implementations, presented in [17] by Shams and Kennedy, support a range of bin sizes. Their methods target data-mining applications, reading 32-bit values as input in contrast to Podlozhnyuk's 8-bit input values. Another CUDA implementation is found in [19], which is a more straightforward implementation with a lower performance.

Prior to the introduction of general purpose GPU programming languages such as CUDA, the histogram algorithm has been implemented using OpenGL by among others [2], [3], [6] and [16]. The above discussed CUDA implementations perform significantly better, because of the use of a more appropriate programming paradigm.

Other related work maps the histogramming algorithm onto non-GPU multithreaded architectures. For example, Ranger et al. describe a parallel implementation of histogramming using the MapReduce framework in [14], which uses 24 CPU cores and achieves a performance increase of a factor 9 over a single CPU core. Pankratius et al. present another multithreaded implementation in [10], achieving a speed-up of a factor 6 on a 8 core machine. The latter multithreaded application explores different implementations, of which their concepts (e.g. atomic operations and core private histograms) are applicable to a GPU implementation.

3. BACKGROUND

In this paper, the CUDA framework is used as a GPU programming language, since it supports many C++ features, has well maintained tools, and a large user-base. When programming CUDA, the programmer is exposed to a large number of concepts and properties [15]. For good understanding of this paper, we give an architecture overview in figure 1 and introduce several basic CUDA concepts:

- The code running on the GPU, the **kernel** code, is executed by a number of **threads**, each having a unique id. These threads are divided into groups of **threadblocks**, enabling the use of a shared memory and enabling synchronization within a threadblock. Outside threadblocks, no synchronization is possible within a kernel. On execution of the kernel, threads are scheduled by the hardware as **warps** in an SIMD-manner. Warps are groups of threads (typically 32) that share the same instruction counter, but are free to branch independently.
- An on-chip **shared memory** is available, which is shared within threadblocks. Access time of the shared memory is comparable with register access time [9]. The shared memory is banked to maximize throughput.
- Memory accesses to the off-chip memory can be **coalesced** as burst accesses by the hardware, which can lead to a memory bandwidth increase of up to 16x [4].
- To hide memory latency, the GPU does not (entirely) rely on caches, but instead switches to another thread from a pool of **thousands of active threads**. As a programmer, enabling this large amount of simultaneously active threads is important [7], but limited by among others register and shared memory requirements.

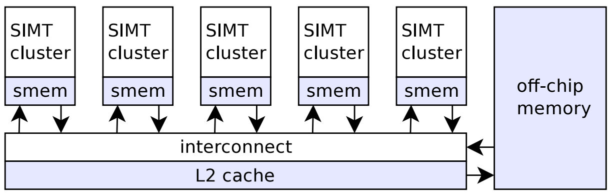


Figure 1: High level abstraction of a GPU architecture, consisting of a number of Single Instruction, Multiple Thread (SIMT) multiprocessors. A multiprocessor consists of multiple *CUDA cores.*

4. REFERENCE METHOD

In this section, we introduce the state-of-the-art of CUDA-based histogramming. Since we only consider 256-bin histogramming implementations, being a common requirement for image processing [5], we discuss Podlozhnyuk's 256-bin method [13]. The method presented by Shams et al. [17] is based on the same techniques. However, since their method focuses on data-mining applications, they require 32-bit inputs and support larger bin sizes. For image processing, inputs are typically 8-bit (either greyscale, luminance or one RGB channel) and require at most 256 output bins. For these reasons, and the fact that the implementation is fundamentally the same as Podlozhnyuk's method, Shams et al.'s method is not further discussed in this paper.

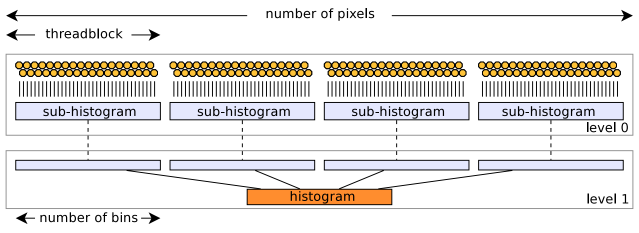


Figure 2: A top level view of the reference histogramming method. Threadblocks calculate sub-histograms, which are summed in a second kernel.

In Podlozhnyuk's 256-bin histogramming method [13], the elements in the input data-set are divided in blocks of equal size. Each threadblock processes one block of work and produces one sub-histogram. Between different threadblocks in a kernel, no synchronization is possible. Thus, to calculate the final histogram, a second kernel is started, summing all the intermediate sub-histograms. An overview is shown in figure 2, in which the input elements, sub-histograms and final histogram reside in the off-chip global GPU memory.

Next, we discuss the first kernel in more detail. Within one threadblock, the fast on-chip shared memory is used to store the sub-histograms. Values in the sub-histogram are either incremented or not depending on the value of the input element. This implies that accesses to the sub-histogram are data dependent, and that multiple accesses can occur to the same element at the same time. This introduces a problem, since operations to the shared memory are not atomic[3]. This causes different threads (with unspecified execution order) to read and write non-atomically to a-priori unknown address spaces in the shared memory. When such a *shared memory collision* occurs, only one memory write will succeed, but it remains unknown which.

We distinguish two types of shared memory collisions: inter-warp (between different warps) and intra-warp (within a warp). To solve inter-warp shared memory collisions, Podlozhnyuk's method introduces a second level of partial results, referred to as *warp-histograms*. As seen in the detailed view in figure 3, each warp calculates its own partial histogram. Then, after a threadblock synchronization barrier, the warp-histograms are summed up as a sub-histogram.

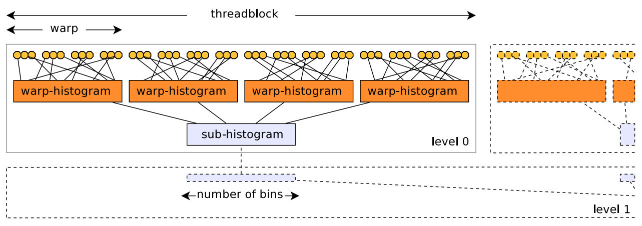


Figure 3: A detailed view of histogramming at a threadblock level.

Although this solves *inter*-warp shared memory collisions, *intra*-warp collisions can still occur. Podlozhnyuk uses the properties of warps to implement a virtual atomic operation and to solve the intra-warp collisions. Figure 4 shows an overview of the virtual atomic operation. First, the to-be-incremented bin is loaded from the shared memory. The value of the bin (27-bits) contains a tag from a previous operation in the upper bits (5-bits for 32 threads). After incrementing the bin, a thread-unique tag is added to the upper 5 bits, and the entire 32-bit word is stored in shared memory. Since other threads within the same warp could have performed the exact same operation[4], the stored value is directly loaded back and compared against the calculated value. If the values are the same, the operation is performed successfully. If not, another thread has won the collision, and the virtual atomic operation is repeated for the losing threads.

Virtual atomic operations are only able to solve intra-warp shared memory collisions, because of a warp's properties: inside a warp, all threads are guaranteed to execute the same instruction. However, outside a warp, a complete virtual atomic operation could be performed in one of the stages of another thread's virtual atomic operation, causing potentially incorrect behaviour.

[3]Newer GPU architectures do support atomic shared memory operations (see section 6)

[4]If multiple threads write to the same location, only one (a-priori unknown) thread will succeed

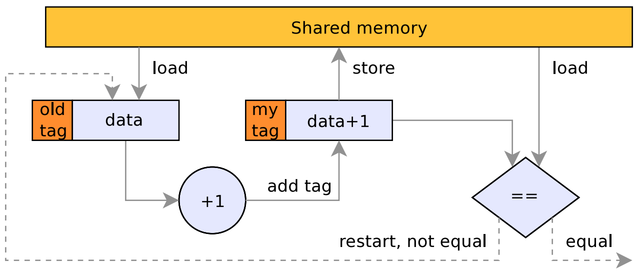


Figure 4: The virtual atomic operation loop.

When all threads within a warp access a different bin, the overhead is one load, one compare and a few arithmetic operations. However, when all threads within a warp access the same bin, the whole virtual atomic operation is performed 32 times (the size of a warp) at most, causing significant overhead. The difference between worst-case and best-case performance can be as much as a factor 8.

5. BENCHMARK SET-UP

A number of benchmarks are presented to measure performance. In this section we introduce the benchmark set, the used hardware, and the applied optimization techniques.

5.1 Input data characteristics

For the benchmark test-set, we selected four greyscale images, each containing 2048x2048 8-bit pixels. The images and their histograms are shown in figures 5(a), 5(b), 5(c) and 5(d). Next to the images, two synthetic inputs are generated. Figure 5(e) show a random data distribution and its histogram, and figure 5(f) show a degenerate data distribution and its histogram.

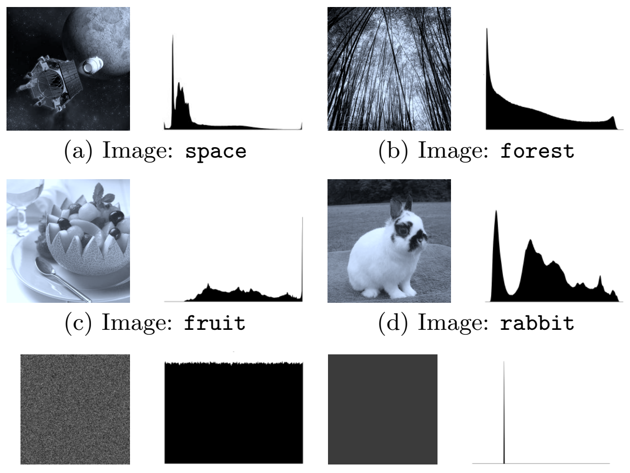


Figure 5: The input data-set and corresponding histograms.

Although the presented test-set provides images with different characteristics and can be used to evaluate design decisions, we provide two larger sets to evaluate average performance. These sets include: (1) 23 high resolution nebulae images are taken from the NOAO image gallery (www.noao.edu/image_gallery), and (2) an equal number of 2048x2048 images from www.shanghaidailyphoto.com.

5.2 Hardware configuration

We use two hardware configurations in this paper, which are shown in table 1. Unless stated otherwise, the GTX470-system is used. All performance figures are averaged over 10 runs and are measured in terms of effective bandwidth. We define this metric as the amount of processed input data divided by the time to execute the algorithm. In this way, a bandwidth close to the practical bandwidth will imply an effective implementation, while a lower bandwidth will be the result of a larger instruction or memory access overhead.

Table 1: Hardware configuration.

	GTX470-system	GTS250-system
Operating System	Fedora 12 64-bit	Fedora 12 64-bit
Host CPU	Core-i7 930	Core2Duo Q8300
GPU type	GeForce GTX470	GeForce GTS250
CUDA cores	448 @ 1.2GHz	128 @ 1.8GHz
GPU practical BW	±100GB/s	±55GB/s

5.3 Optimizations

All presented benchmarks are the result of highly optimized code. We perform code optimizations including recommendations by NVIDIA [8], coalescing off-chip memory accesses where possible, grouping input data as 32-bit elements, inspecting PTX-code (intermediate level assembly), and using pointer arithmetic for address calculations. Additionally, a workload distribution sweep has been performed for both methods, varying the amount of data processed per thread versus the number of blocks, as discussed in [18].

Since we use a 64-bit operating system, GPU kernel code is per default compiled as 64-bit. This results in the use of 64-bit pointers for the GPU's shared memory, causing a significant overhead of 32-bit to 64-bit conversion instructions[5]. Therefore, all benchmarks are performed using the compiler flag `-m32`, compiling 32-bit code.

6. METHOD 1: WARP HISTOGRAM

The reference GPU implementation of histogramming performs at only 5% of the practical peak bandwidth of GPUs, leaving room for improvement [13]. Therefore, three extensions to the existing method of Podlozhnyuk are explored in this section[6], all based on the two-level approach (figure 2), including a private histogram per warp (figure 3). We present the concepts of the three different extensions (the first two are mutually exclusive), followed by a performance comparison, which includes a baseline implementation.

6.1 Extension 1: Data shuffling

Since histogramming does not rely on order nor on coordinates of data elements, input data can be rearranged freely while still guaranteeing correctness. For image data, neighbouring pixels typically have a high correlation, leading to a high probability of incrementing the same bin.

With an additional pre-processing step, input data can be rearranged in such a way that the amount of shared memory collisions is reduced. However, since the overhead of the pre-processing step needs to be minimal, both reading and writing to the off-chip memory must be coalesced. Figure 6 shows the implemented rearrangement pattern, used in `extension1`. First, data is read coalesced from off-chip memory and stored in on-chip shared memory. Following, the two-dimensional shared memory array's x and y-coordinates are swapped, resulting in the rearrangement of the data. Finally, as seen in figure 6, the data is written back coalesced into the off-chip memory.

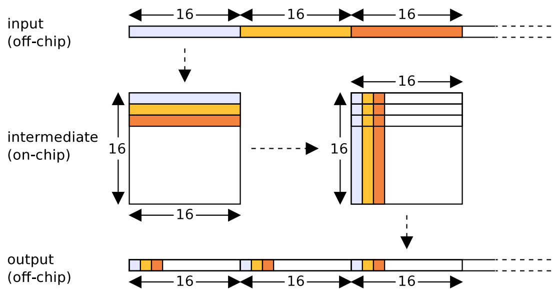
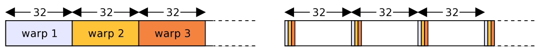


Figure 6: The data rearrangement scheme used for data shuffling in `extension1`.

6.2 Extension 2: Giving up on coalescing

In the baseline implementation, memory accesses are coalesced. In order to coalesce reads, all threads in a warp read sequentially from the input data. A coalesced memory access pattern is shown in figure 7(a). As stated before, this can lead to a high probability of incrementing the same bin for subsequent data elements. On the other hand, pixels further away from each other have a lower correlation, leading to a lower probability of incrementing the same bin. So, although memory accesses are coalesced in the baseline implementation, performance degradation can occur due to shared memory collisions for images with a high local correlation.



(a) Coalesced (b) Uncoalesced

Figure 7: Different memory access patterns.

In `extension2`, access pattern restrictions are relaxed by giving up on coalescing. Pixels can be read uncoalesced within a warp, as shown in figure 7(b). This extension benefits from a lower amount of shared memory collisions and thus less virtual atomic operations, but performance can decrease due to a lower off-chip memory bandwidth.

6.3 Extension 3: Hardware atomics

The virtual atomic operation loop in figure 4 guarantees correctness, but causes a large overhead of loads, stores and arithmetic instructions. At the time when the reference histogramming method was designed, GPUs did not support atomic shared memory operations. Since 2009, architectures with compute capability 1.2 or higher (such as the GeForce GTX470) do support shared memory atomic operations in hardware. `Extension3` uses hardware atomic support instead of the virtual atomic operation loop.

[5]This can be as much as 21% for the presented methods
[6]The concepts are extended, code is written from scratch

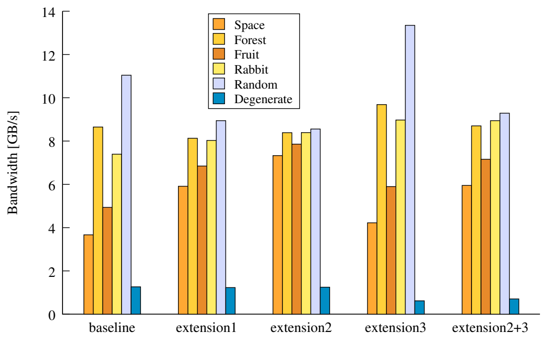


Figure 8: Performance comparison of the baseline implementation, three extensions and a combination of extensions 2 and 3.

Table 2: Results for non-synthetic benchmarks.

Name	Average bandwidth	Relative performance	Standard deviation
`baseline`	6.1 GB/s	1.00	2.3 GB/s
`extension1`	7.2 GB/s	1.18	1.0 GB/s
`extension2`	8.0 GB/s	1.31	0.7 GB/s
`extension3`	7.2 GB/s	1.18	2.6 GB/s
`extension2+3`	7.6 GB/s	1.25	1.4 GB/s

6.4 Extension selection

We measure performance for a baseline implementation in which no extension is applied, the three extensions, and a combination of extensions 2 and 3. Results are shown in figure 8 and table 2. The images `space` and `fruit` show the lowest performance for all methods, which can be explained from the two high peaks as seen in the histograms in figures 5(a) and 5(c). We discuss the results of the three different extensions individually:

- Performance figures of `extension1` as shown in figure 8 include both the pre-processing and the actual histogramming steps. For best-case input data (`random`), the penalty of performing a pre-processing step is significant. However, for non-synthetic benchmarks, pre-processing input data proves to be useful, showing higher performance for most cases. On average for non-synthetic benchmarks, `extension1` shows a performance increase of 18% as compared to `baseline`. Additionally, the standard deviation is much lower, increasing the performance predictability.

- Although the baseline implementation method is faster for the `random` data-set, `extension2` shows a higher bandwidth and a lower standard deviation for non-synthetic benchmarks. On average, `extension2`'s performance is 31% higher compared to `baseline` for non-synthetic benchmarks. Again, the standard deviation is significantly lower compared to `baseline`.

- Using hardware atomic operations, performance increases for data-sets with a low number of shared memory collisions, while performance decreases for data-sets with a high number of collisions, in particular for the extreme case of `degenerate`. On average, adding hardware atomics to `baseline` shows a performance increase of 18% for non-synthetic benchmarks, while performance drops by 5% when added to `extension2`.

`Extension2` shows the best results in terms of average performance and standard deviation. Therefore, from this point on, we refer to `method1` as warp private histogramming extended with uncoalesced accesses.

7. METHOD 2: THREAD HISTOGRAM

Since shared memory collisions can cause a significant overhead, we present a method to circumvent such collisions. Instead of one histogram per warp (seen in figure 2), now, one histogram per thread is available, as shown in figure 9. All threads access a number of elements and process them sequentially. Since no memory locations are shared, no collisions can occur, removing the need for atomic operations.

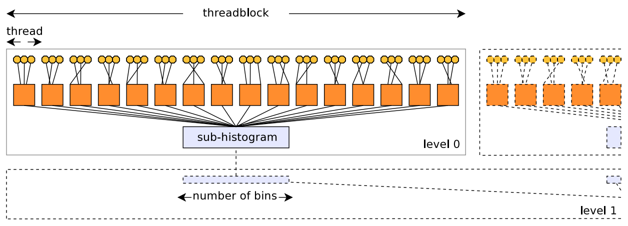


Figure 9: Each thread computes its own partial histogram, which is then summed as a sub-histogram within a threadblock.

However, thread private histogramming introduces a large shared memory cost per thread. When processing 256 or more elements per thread, bins need to be stored as 2-byte integers, resulting in 512 bytes of shared memory usage per thread. With the availability of 48KB shared memory, this limits the number of threads to 96. This number is not sufficient to completely hide pipeline and memory latencies [7], which results in a performance penalty.

The shared memory in the multiprocessors of the GeForce GTX470 architecture consists of 32 banks, each with up to 384 32-bit entries. Addressing is interleaved, e.g. successive 32-bit words are assigned to successive banks [9]. The memory runs at half the clock speed of the two groups of 16 CUDA cores, each scheduled by its own warp scheduler. Each group of 16 CUDA cores executes two half-warps in 2 successive clock cycles. In case of a shared memory access, bank conflicts occur when two or more threads in a warp are accessing a different row in the same bank. With a second scheduler, the constraints tighten, since a total of 64 threads need to access the shared memory conflict free to achieve full performance.

We explore four different mappings of the histograms on the shared memory. The mappings are shown in figures 10 - 13. In these figures, Tx denotes the thread number and different colours refer to different warps. The characteristics of the four memory mappings are summarized in table 3 and are discussed in the following sections.

7.1 Mapping 1: layout 1, 32 threads

In the first layout, histogram bins are mapped onto the shared memory by iterating over all threads, creating thread-wide vectors of 16-bit thread private bins. The first mapping has a total of 32 threads per threadblock, interleaving histogram entries over memory banks (as shown in figure 10). This layout is automatically obtained by requesting a 16-bit array on the shared memory.

Table 3: Shared memory mappings: overview of characteristics.

	CUDA cores active	Bank mismatch	Row mismatch	Inter-warp mismatch	Address calculation	Active threadblocks	Active threads
Layout 1, 32 threads	16 cores	chances	chances	never	2 PTX-instr.	3 threadblocks	96 threads
Layout 1, 64 threads	32 cores	always	chances	never	2 PTX-instr.	1 threadblock	64 threads
Layout 2, 32 threads	16 cores	never	never	never	6 PTX-instr.	3 threadblocks	96 threads
Layout 2, 64 threads	32 cores	never	never	chances	6 PTX-instr.	1 threadblock	64 threads

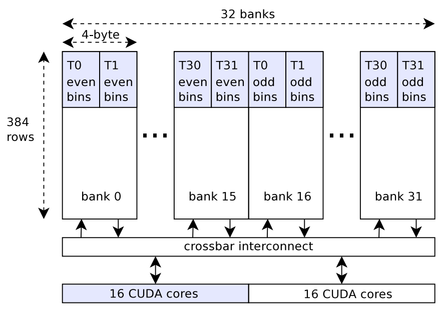


Figure 10: Layout 1 with 32 threads. Two threads have a chance to access the same bank.

Depending on the input data, threads have a chance to access banks that are simultaneously accessed by other threads. However, a bank conflict will only occur whenever a row mismatch occurs, e.g. only in the case that a different histogram bin and thus memory row is accessed within one bank. With only 32 threads per threadblock, only one warp and thus one group of 16 CUDA cores is active. On the other hand, we can now schedule multiple threadblocks on the same multiprocessor. Because there is still shared memory available, three threadblocks can be scheduled simultaneously on one multiprocessor. Note that these other threadblocks will run interleaved on the same 16 CUDA cores.

7.2 Mapping 2: layout 1, 64 threads

The second memory mapping uses the same layout, but has 64 threads per threadblock. This changes the distribution of histograms over the shared memory banks as seen in figure 11. Now, the first 16 banks are occupied by the first warp's histograms, while the next 16 banks are occupied by the second warp. Banks are now always accessed by two threads simultaneously, resulting in a bank conflict if there is a row mismatch. Although both groups of CUDA cores are active, only one threadblock can be active on one multiprocessor at the same time (limited by shared memory size).

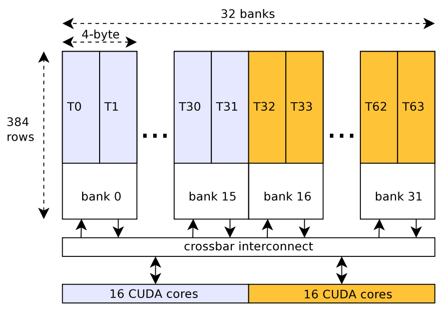


Figure 11: Layout 1 with 64 threads. Two threads share one memory bank.

7.3 Mapping 3: layout 2, 32 threads

We improve the first layout by assigning all 32 threads a private bank in the shared memory. As seen in figure 12, this reduces the chance of bank conflicts to zero. This layout is obtained by implementing complex address calculation, which can lead to performance decrease due to significant instruction overhead.

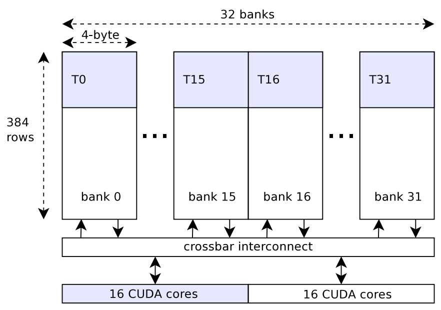


Figure 12: Layout 2 with 32 threads. Each thread has a private memory bank.

7.4 Mapping 4: layout 2, 64 threads

In the final mapping, we enable 64 threads using the same layout (see figure 13). While threads previously had a private shared memory bank, banks are now shared between threads from different warps. Although the two warps are executing independently, the chance exists that all memory banks are accessed twice, causing bank conflicts. In contrast to prior mappings, this chance is not data dependent, but dependent on the behaviour of the warp scheduler.

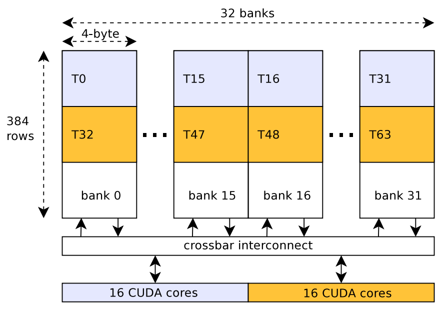


Figure 13: Layout 2 with 64 threads, threads from different warps share a memory bank.

7.5 Mapping selection

We compare the four mappings in terms of performance, with results shown in figure 14. As expected, we observe full data independence for the second layout. The first layout is not data independent, and favors data-sets with a high data correlation, which reduces the chances of a row mismatch.

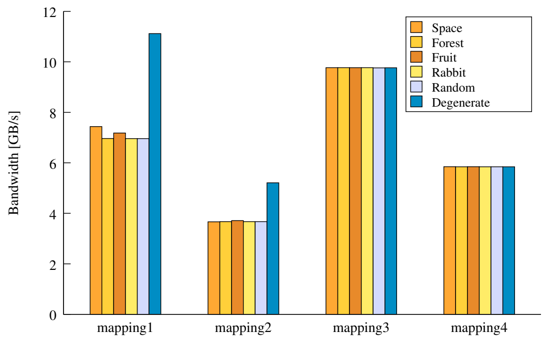


Figure 14: A comparison of the four memory mappings for `method2`.

Although at least 64 threads are needed to occupy all CUDA cores [9], both memory layouts favor 32 threads. Increasing the threadcount from 32 to 64, shared memory pressure increases and bank conflicts occur more often. This can be explained through table 3, as bank mismatching worsens for the first layout. For the second layout, bank mismatching can occur in between different warps, causing bank conflicts. While both 64 thread solutions enable more CUDA cores, they both limit the number of active threadblocks, reducing latency hiding possibilities.

Since `mapping3` shows the best average performance and the lowest variance, we refer to `method2` from this point on as thread private histogramming using `mapping3`.

8. EVALUATION

To evaluate the performance and characteristics of the presented methods, we show benchmark results, evaluate performance limitations, and discuss architecture suitability.

8.1 Benchmark results

In figure 15, we show the benchmarks of Podlozhnyuk's method and the two methods presented in this paper. While `method1` improves Podlozhnyuk's method significantly for most input images, `method2` shows a much higher average performance. Additionally, `method2` is data independent, resulting in a higher performance predictability.

However, we also performed benchmarks for the GTS250-system (instead of the GTX470-system used in all other benchmarks). In these benchmarks, `method2` was not included, since the shared memory size on the older architecture is insufficient. On the GTS250-system, `method1` shows an average performance of 4.6GB/s for non-syntethic benchmarks, which is significantly more than the 2.4GB/s achieved by Podlozhnyuk's method.

8.2 Performance limitations

Although both methods show a higher performance and a lower variance compared to Podlozhnyuk's method, the performance is still not close to the practical bandwidth peak

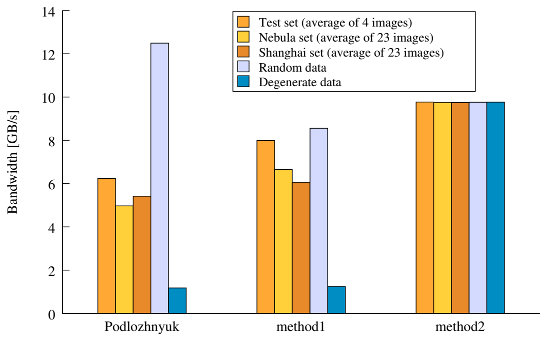


Figure 15: Final benchmark results, comparing the two presented methods against the reference method. Tested using 50 images and 2 synthetic inputs.

of the GPU. Therefore, the performance limitations of the implemented methods are discussed.

Both presented methods are limited by the following aspects: (1) shared memory bank conflicts, and (2) the lack of support to perform ALU operations directly on the shared memory. Additionally, for `method1`, performance is further limited by the atomicity problem: (1) intra-warp shared memory collisions and the need for a virtual atomic operation loop, and (2) inter-warp shared memory collisions and the need for histogram accumulation.

If it is assumed that no inter-warp nor intra-warp shared memory collisions occur, performance increases significantly for `method1`. This is illustrated in figure 16, in which the baseline implementation is tested against two modified versions in which shared memory collisions are assumed not to occur within warps (`no_intra_warp`) or not at all (`no_intra _inter_warp`). Since they do occur, results for the data-sets `space`, `random` and `degenerate` are incorrect. The fourth data-set, `ideal`, consists of a specialized data distribution in which shared memory collisions will not occur.

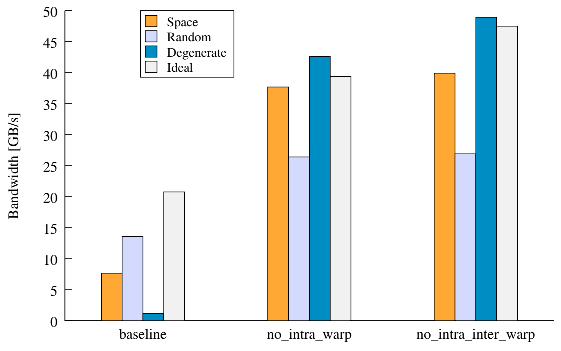


Figure 16: The performance limitations of `method1` are evaluated by assuming that atomicity is handled automatically and at no cost.

From figure 16, it can be concluded that intra-warp shared memory collisions cause the major part of the overhead for `method1`. Especially data-sets with a high correlation benefit greatly, removing the need for multiple iterations of the virtual atomic operation loop. Additionally, they benefit from the fact that bank conflicts do not occur when threads access the same row in a bank. This explains why the best performance is achieved by the `degenerate` data distribution.

8.3 Architecture limitations

In [8], it is stated that the GPUs of compute capability 2.0 (such as the GeForce GTX470) might need 768 threads per multiprocessor to fully hide pipeline latencies. Also, according to [8], it is recommended to have at least 64 threads per threadblock, assuming that multiple threadblocks can be mapped onto one multiprocessor. Although this might be true for most applications, `method2` clearly does not follow these performance guidelines. Even worse, the best performing method has half of all processing elements idling at any point in time, resulting in a huge waste of silicon area.

From these numbers, we can conclude that there is a mismatch between the availability of shared memory (both in terms of size and banks) and the number of CUDA cores per multiprocessor (for GPUs of compute capability 2.0). Although this might not be the bottleneck for most CUDA programs, `method2` does require an increase of the number of shared memory banks to 64 to use all CUDA cores, enabling the use of 64 threads without bank conflict penalties. To hide pipeline and memory latencies, more threads are required. An increase of the memory size to 96KB enables three threadblocks and thus 192 threads in total, improving the performance of histogramming on GPUs even further. A rough calculation estimates a performance gain of 2x when the number of banks is increased and another gain of 3x when the memory size is increased.

9. CONCLUSIONS

In this work, we presented two novel methods to perform histogramming on a GPU. For the first method, three different extensions to an existing method were explored, each resulting in a higher average bandwidth (33% at most) and a lower variance for non-synthetic benchmarks compared to existing methods. Although tested on the latest NVIDIA GPU architecture, the first method is also suitable for older architectures. The second presented method does not share histograms over different warps and benefits heavily from the increased amount of shared memory available on the latest architectures. Different shared memory mappings were explored, showing at most a 56% performance increase over existing methods. Additionally, execution time is completely input data independent, making histogramming suitable for use in real-time applications.

Furthermore, we evaluated both novel methods. For the first method, a performance limitation breakdown shows that atomicity is the main performance bottleneck. For the second method, we observed that the best performing implementation uses half the number of CUDA cores available and only 32 threads. From these numbers, we argued that there is a mismatch between the number of CUDA cores and the number of shared memory banks. Additionally, for a high performance implementation, histogramming requires a higher amount of shared memory.

10. REFERENCES

[1] G. de Haan. *Digital Video Post Processing.* University Press Eindhoven, 2006.

[2] O. Fluck, S. Aharon, D. Cremers, and M. Rousson. GPU histogram computation. In *ACM SIGGRAPH 2006 Research posters*, SIGGRAPH '06. ACM, 2006.

[3] J. Fung and S. Mann. OpenVIDIA: parallel GPU computer vision. In *Proceedings of the 13th annual ACM international conference on Multimedia*, MULTIMEDIA '05, pages 849–852. ACM, 2005.

[4] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov. Parallel Computing Experiences with CUDA. *IEEE Micro*, 28(4):13–27, 2008.

[5] R. Gonzalez and R. Woods. *Digital Image Processing.* Prentice Hall, 2002.

[6] S. Green. Image processing tricks in OpenGL. In *Game Developer's Conference 2005: OpenGL Tutorial Day.* NVIDIA, 2005.

[7] C. Nugteren, B. Mesman, and H. Corporaal. Analyzing CUDA's Compiler through the Visualization of Decoded GPU Binaries. In *ODES-8: Proceedings of the 8th Workshop on Optimizations for DSP and Embedded Systems at CGO '10*, 2010.

[8] NVIDIA. *CUDA C Best Practices Guide 3.2*, 2010.

[9] NVIDIA. *CUDA C Programming Guide 3.2*, 2010.

[10] V. Pankratius, C. Schaefer, A. Jannesari, and W. F. Tichy. Software engineering for multicore systems: an experience report. In *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*, pages 53–60. ACM, 2008.

[11] I. Park, N. Singhal, M. Lee, S. Cho, and C. Kim. Design and Performance Evaluation of Image Processing Algorithms on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 99(99):1, 2010.

[12] D. Patterson. The Top 10 Innovations in the New NVIDIA Fermi Architecture, and the Top 3 Next Challenges. NVIDIA Whitepaper, 2009.

[13] V. Podlozhnyuk. Histogram calculation in CUDA. Technical report, NVIDIA, 2007.

[14] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proc. IEEE 13th Int. Symp. High Performance Computer Architecture HPCA 2007*, pages 13–24, 2007.

[15] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82. ACM, 2008.

[16] T. Scheuermann and J. Hensley. Efficient histogram generation using scattering on GPUs. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, I3D '07, pages 33–37. ACM, 2007.

[17] R. Shams and R. A. Kennedy. Efficient histogram algorithms for NVIDIA CUDA compatible devices. In *ICSPCS: Proc. Int. Conf. on Signal Processing and Communications Systems*, 2007.

[18] V. Volkov. Use registers and multiple outputs per thread on GPU. In *International Workshop on Parallel Matrix Algorithms and Applications (PMAA'10)*, 2010.

[19] Z. Yang, Y. Zhu, and Y. Pu. Parallel Image Processing Based on CUDA. In *Computer Science and Software Engineering, 2008 International Conference on*, volume 3, pages 198 –201, 2008.