Simulation and Architecture Improvements of Atomic Operations on GPU Scratchpad Memory

Gert-Jan van den Braak*, Juan Gómez-Luna†, Henk Corporaal*, José María González-Linares‡ and Nicolás Guil‡
* Department of Electrical Engineering, Eindhoven University of Technology, The Netherlands
{g.j.w.v.d.braak, h.corporaal}@tue.nl
† Department of Computer Architecture and Electronics, University of Córdoba, Spain
el1goluj@uco.es
‡ Department of Computer Architecture, University of Málaga, Spain
{jgl, nguil}@uma.es

***Abstract*—GPUs are increasingly used as compute accelerators. With a large number of cores executing an even larger number of threads, significant speed-ups can be attained for parallel workloads. Applications that rely on atomic operations, such as histogram and Hough transform, suffer from serialization of threads in case they update the same memory location. Previous work shows that reducing this serialization with software techniques can increase performance by an order of magnitude. We observe, however, that some serialization remains and still slows down these applications. Therefore, this paper proposes to use a hash function in both the addressing of the banks and the locks of the scratchpad memory. To measure the effects of these changes, we first implement a detailed model of atomic operations on scratchpad memory in GPGPU-Sim, and verify its correctness. Second, we test our proposed hardware changes. They result in a speed-up up to 4.9× and 1.8× on implementations utilizing the aforementioned software techniques for histogram and Hough transform applications respectively, with minimum hardware costs.**

I. Introduction

The last 30 years of computer architecture design has been dominated by single core performance growth. This came to a halt in 2004 [1], when processors became limited by power dissipation. Since then, the trend has been towards parallelism and multi-cores to fulfill the ever growing demand for compute performance. At the same time, GPUs (Graphics Processing Units) appeared as an inexpensive accelerator for general-purpose computations. The GPU's many-core architecture is still an unpolished gem though, and minor architecture changes can improve application performance significantly. Over the next 30 years we will see refinements of the GPU's architecture. Not just to increase (peak) performance, but also to effectively use this performance in applications.

In this work we focus on a small, but important, part of a GPU: the scratchpad memory. This on-chip, banked memory supports hardware atomic operations via locks on memory addresses. Applications that rely on these operations, such as histogram and Hough transform, experience a performance loss when threads get serialized in case of bank or lock conflicts. Conflicts by threads updating the same memory address (*position conflicts*) can be reduced by software techniques [2], [3]. Bank conflicts (threads accessing different memory addresses in the same bank) and lock conflicts (threads updating different memory addresses which share a lock) are not resolved by these techniques, but are actually increased. Since these conflicts are less severe for performance than position conflicts, the software techniques still lead to improvements.

To decrease the remaining conflicts and thereby increase performance, this paper proposes to use a hash function in both the addressing of the banks and the locks of the scratchpad memory. The hardware costs of these changes are low, and tests in a simulator show a speed-up up to 4.9× and 1.8× for histogram and Hough transform applications respectively. The simulator used is GPGPU-Sim [4]. As it did not accurately model atomic operations on scratchpad memory, we first implement a detailed performance model [5] of these operations. The main contributions of this work are as follows:

- We improve GPGPU-Sim by implementing a performance model of atomic operations in scratchpad memory. The new simulator is then validated against real measurements on a current GPU.
- We show how to use the simulator to propose architecture modifications that improve the performance of the atomic operations in scratchpad memory. Two hash functions are proposed to reduce the amount of bank and lock conflicts.
- We evaluate the architecture improvements of atomic operations using two widely-used applications (histogramming and Hough transform).

The rest of the paper is organized in the following sections. Section II explains how atomic operations are executed in a current GPU [5]. Such an explanation is utilized to modify GPGPU-Sim in Section III. Section IV explores the use of the simulator to propose architecture improvements. Section V ratifies the positive effect of such improvements on two widely-used voting applications. Section VI presents related works. Finally, the conclusions are stated in Section VII.

II. A model for the execution of atomic operations in shared memory

A proper integration of atomic operations in GPGPU-Sim requires an accurate understanding of their performance in GPU architectures. Thus, this section summarizes how atomic operations are processed on shared memory by one warp [5].

```
1 /*0210*/ LDSLK P0, R7, [R9];
2 /*0218*/ @P0 IADD R10, R7, 0x1;
3 /*0220*/ @P0 STSUL [R9], R10;
4 /*0228*/ @!P0 BRA 0x210;
```

Fig. 1. Assembly code for an atomic addition on Fermi instruction set (c.c. 2.0). LDSLK reads and locks a shared memory location. IADD increments by 1 if the lock has been acquired (predicated by P0). STSUL writes and unlocks the shared memory location. The conditional branch BRA is executed if the lock was not acquired.

The shared memory is a scratchpad memory divided into equally-sized memory modules, called banks, which can be accessed simultaneously. Successive 4-byte words are assigned to successive banks. If the number of banks is N and A is the address of a 4-byte word, A resides in bank A mod N. This permits a high bandwidth if threads access addresses that fall in distinct memory banks. However, if two different addresses of a memory read or write request fall in the same bank, there is a bank conflict and the access has to be serialized. In the Fermi architecture [6], the shared memory has 32 banks, which is the warp size too. This way, the granularity of memory requests is 32 [7], so that shared memory requests for different warps are served in different memory transactions. Thus, bank conflicts are only possible among threads belonging to the same warp. Shared memory size is 48 KB in Fermi architecture.

For devices of compute capability 1.2 and above, CUDA offers atomic functions which perform a read-modify-write operation on a word residing in shared memory. For example, `atomicAdd()` reads a word at some address, adds a number to it, and writes the result back to the same address. It is atomic in the sense that no other threads can access this address until the operation is complete.

A. Lock mechanism for atomic operations in shared memory

The lock mechanism that enables atomic updates to shared memory is implemented by a memory lock unit described in [8]. Memory read and write requests from threads are input to the memory lock unit. A set of lock bits are provided that store the lock status for locations. A lock bit may be shared among several addressable locations. Thus, multiple addresses are aliased to the same lock bit. A hash function may be implemented by the memory lock unit to map request memory addresses to lock bit addresses. The hash function guarantees preferably that consecutive word addresses will map to different lock bits. Otherwise, it may simply use the low bits of the address.

Read instructions return both the data that is stored at the indicated address and a flag that determines if the lock was successfully acquired. Such a flag is related to the content of a predicate register (P0 in Fig. 1). The lock bits are accessed in parallel with memory read and write accesses, so that no additional pipeline stages or clock cycles are needed to acquire and release the lock.

If the lock was successfully acquired, the program may then modify the data, store the new value and release the lock to allow other threads to access the location whose address aliases to the same lock address as the released lock address. If the lock is not successfully acquired, the program should attempt to acquire the lock again. This is why the branch instruction is included. Thus, the number of iterations of the code in Fig. 1

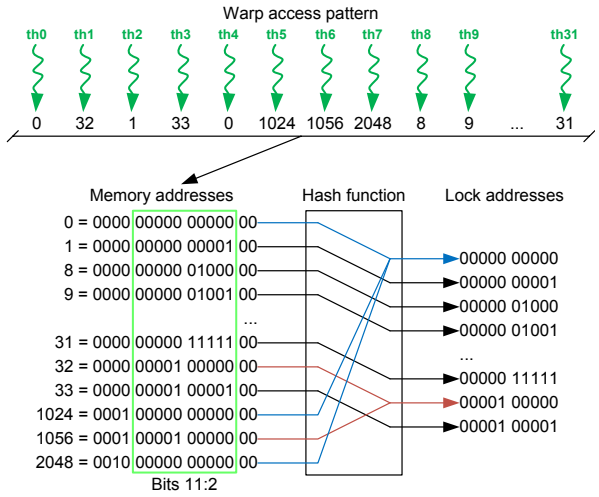


Fig. 2. Implementation of hash function in lock mechanism. Given a memory address, the corresponding lock address is given by bits 11:2. Memory addresses at distance 1024 words are aliased, since they have the same lock address. This way, threads 0, 4, 5, and 7 will be executed sequentially, as well as threads 1 and 6.

that the program carries out is determined by the number of addresses mapping to the same lock bit. The program is also responsible for honoring the lock bits through the predicate register, since the memory lock unit is not configured to track lock ownership.

B. Performance model

It can be seen that threads compete for locking access to those addresses which are to be atomically updated. This fact reveals the serialization that threads of a warp suffer when they try to update addresses sharing the same lock bit, i.e., aliased addresses. With illustrative purposes, let us consider threads of a warp atomically updating addresses [x, y, 2, 3, 4, ..., 31]. Such a set of 32 addresses is called a *warp access pattern*. If x and y are not aliased (and are not aliased to any of addresses 2 to 31), the atomic operation will take a certain minimum latency that we call base latency. However, if x and y share lock address, the latency will be equal to the base latency plus a latency penalty. This can be called a *lock conflict* with lock conflict degree equal to 2. If there is a third address z aliased to x and y (and not aliased to the remaining 29 addresses), the latency will be the base latency plus two times the latency penalty. Thus, the lock conflict degree is 3.

In [5] Gómez-Luna et al. revealed that the lock mechanism in the Fermi architecture uses 1024 independent locks. The lock address is given by bits 11:2, as Fig. 2 illustrates. The access pattern in the figure presents lock conflicts between addresses 0, 1024 and 2048, and between addresses 32 and 1056. We have observed that the highest lock conflict degree (3 in the current example) determines the total latency, since lock conflicts with lower degree are resolved concurrently.

Taking into account the former issues, the shared memory can be understood as composed by a memory lock unit and a storage resource, which is divided into a number of pages containing 1024 4-byte locations. Such an understanding stands for an architecture model of the shared memory according to atomic operation execution, that is illustrated by the schematic of the shared memory in Fig. 3.

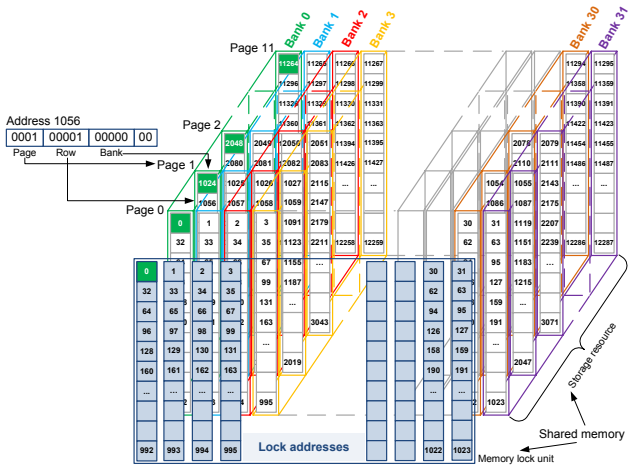


Fig. 3. Scratchpad memory layout on an NVIDIA Fermi GPU. The 48kB of memory is accessed via 4-byte words and is distributed over 32 banks. Each bank has 32 lock bits available for atomic operations.

III. Implementing shared memory atomic operations on GPGPU-Sim

GPGPU-Sim [4] is a detailed simulator of contemporary GPU architectures, such as NVIDIA's GT200 and Fermi architecture. It has detailed models for almost all parts of the GPU, such as the register file and operand collector, caches, interconnect network, instruction scheduler, etc. The operations on the shared memory are only simulated based on their bank conflict degree. Shared memory atomic operations are modeled as general load operations. Although for general load and store operations the bank conflict degree is an accurate model for the latency of the operation, for atomic operations it is far from accurate. As shown in Fig. 4, this provokes an evident divergence between the simulation and the real GPU behavior.

GPGPU-Sim can simulate either NVIDIA's intermediate (GPU independent) instruction set PTX [9] for all CUDA enabled GPUs before Kepler (e.g. G80, GT200, and Fermi), or the GPU's native instruction set SASS for NVIDIA GPUs before Fermi (e.g. G80 and GT200). Since we want to compare the simulator to a Fermi GPU, we have to use the PTX instruction set.

As shown in Fig. 1, an atomic operation in shared memory takes four instructions, which are executed multiple times in case of a lock conflict. The corresponding PTX code consists of only a single instruction. To mimic the behaviour of an atomic operation, we implement the atomic operations in GPGPU-Sim as a finite-state machine (FSM) with four states, *Read*, *Update*, *Write* and *Branch*.

The latency of the *Read* and the *Write* state in the FSM are determined by the level of bank conflicts in the load or store operation respectively. The bank conflict level is determined by how many threads access different addresses in the same bank. This is influenced by how many threads in the warp are active, based on which threads acquired locks. The latency of the add instruction in the *Update* state and the branch instruction in the *Branch* state are fixed values.

All latency values of the states of the FSM are determined by micro-benchmarking using asfermi [10]. The latency of the add instruction in the *Update* state is found to be 18 cycles, the latency of the branch instruction is 32 cycles. The load and store instructions in the *Read* and *Write* state have a latency of 32 and 36 cycles for each bank conflict level respectively.

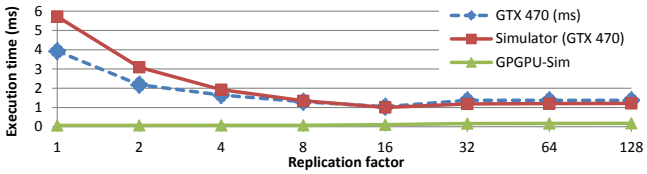


Fig. 4. Comparison between measurements on an NVIDIA GTX 470 (blue line), simulation results on the modified simulator (red line) and simulation results on the original GPGPU-Sim (green line) for a 64-bins histogram computation using a replication factor between 1 and 128 [3].

Validation of the simulator

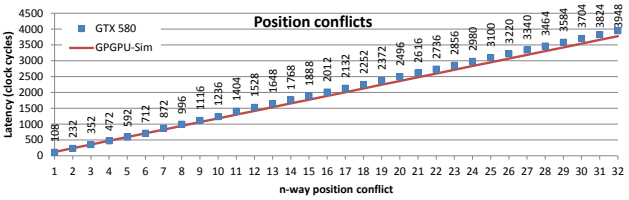
The synthetic benchmarks of Fig. 5 compare the performance of the modified GPGPU-Sim (red line) with experiments on an NVIDIA GTX 580 (blue squares) included in [5]. As can be seen, the simulation is so realistic that the average error is only 2.9%, with a maximum of 8.5% at a conflict degree of 1 (108 vs. 118 cycles). According to [5] there is a linear relation between the number of conflicts and the latency in the first two test cases, but the measurements show a small jitter influencing these error numbers. A single warp (32 threads) is executed in the synthetic benchmarks of Fig. 5. The warp access pattern is calculated according to Eq. 1, where id is the unique identifier of a thread. The $stride$ parameter is equal to 0 for Fig. 5a, 32 for Fig. 5b and 256 for Fig. 5c. In the first test case, Fig. 5a, position conflicts are tested. In the second and third test case, Fig. 5b and Fig. 5c, bank conflicts are validated. In Fig. 5b there are no lock conflicts, in Fig. 5c an extra lock conflict appears every four bank conflicts, creating the step-shaped figure.

$$index(id) = \begin{cases} id \times stride & \text{if } id < conflicts \\ id & \text{otherwise} \end{cases} \quad (1)$$

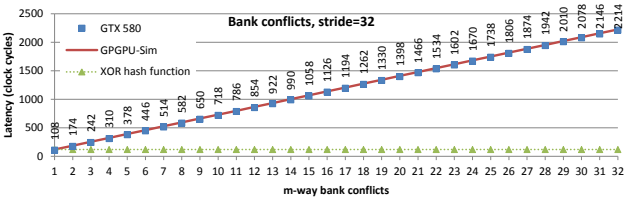
In Fig. 4 a 64-bin histogram is computed. The blue diamonds give the measurements on a NVIDIA GTX 470, the red squares give the simulation results of the modified GPGPU-Sim simulator and the green triangles give the simulation results of the standard GPGPU-Sim. It is evident that the newly implemented model of atomic operations considerably improves the accuracy of GPGPU-Sim for this application. The histogram application has a lower execution time on the GTX 470 for larger replication factors, with an optimum at a replication factor of 16. This is exactly mimicked by the improved GPGPU-Sim simulator, while the original showed a minimum execution time at a replication factor of one. The correlation between the real GPU and the modified GPGPU-Sim is 99%, on par with the IPC correlation of 98.3% for other applications mentioned in the GPGPU-Sim documentation.

IV. Using the simulator to propose hardware improvements

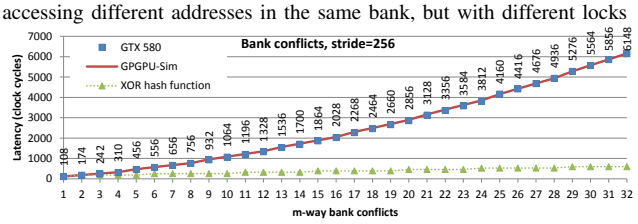
In this section, we explain how the modified simulator can be used to propose and test hardware changes to improve the performance of applications. In some applications, concurrent threads access shared memory addresses with a stride. If the stride is not a relative prime to the number of banks (i.e., an



(a) n-way intra-warp position conflicts, in each test, n threads update the same location



(b) m-way intra-warp bank conflicts, m is given by the number of threads accessing different addresses in the same bank, but with different locks



(c) m-way intra-warp bank conflicts in the simulator, m is given by the number of threads accessing different addresses in the same bank, but an extra lock conflicts appears every four bank conflicts

Fig. 5. Latency in clock cycles of an atomic addition with various intra-warp conflicts. Measurements on a GTX 580 are indicated with blue squares, latency simulated with the modified GPGPU-Sim simulator is indicated with a red line, and latency of the atomic addition with the proposed XOR hash function is denoted with green triangles. No green triangles are shown in (a) since hash functions cannot remove position conflicts.

odd number, given the 32 banks in current architectures), bank conflicts will occur [7]. [1] Moreover, lock conflicts may happen as well, if atomic accesses are used. These conflicts increase the latency of memory accesses, as described in Section II.

Atomic operations are needed in applications where concurrent threads may update the same memory locations. Thus, position conflicts may occur. Illustrative case studies are voting processes, such as histogramming and Hough transform [14]. In these applications, position conflicts are typically avoided by replicating the voting spaces [3], [15], [16], [17], [18]. This way, the number of position conflicts is reduced, but bank and lock conflicts may appear: for those threads that were provoking a position conflict, memory accesses are strided in the replication scheme.

To reduce the number of bank- and lock conflicts in strided memory accesses, a hash function can be applied on the memory address to determine which bank and which lock to use. Traditionally hash functions have been used to increase the bandwidth of interleaved memories and improve the utilization of caches and TLBs [19], [20]. We propose two hash functions called XOR and ADD, as shown in Table I. The former XORs the least and most significant bits (LSB $\oplus$ MSB), and the latter ADDs them (LSB $+$ MSB). The effect of a hash function on the distribution of banks or locks is shown in Fig. 6. For clarity only two bits are used for the LSB and MSB. A memory

[1] A widely-used software technique to avoid bank conflicts is padding [11], [12], [13], that is, keeping some memory locations unused to modify the stride.

TABLE I. BASELINE AND THE PROPOSED HASH FUNCTIONS XOR AND ADD FOR SELECTING WHICH BANK AND LOCK TO USE

	bank	lock
baseline	addr[6:2]	addr[11:7]
XOR hash	addr[6:2] $\oplus$ addr[11:7]	addr[11:7] $\oplus$ addr[15:12]
ADD hash	addr[6:2] $+$ addr[11:7]	addr[11:7] $+$ addr[15:12]

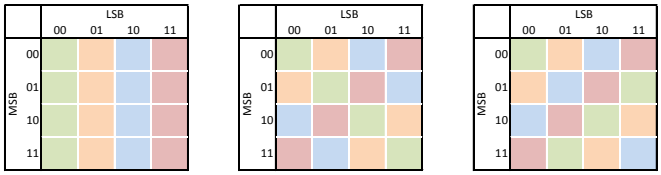
access pattern with a stride of four words will access exactly one column in Fig. 6, and consequently only one of the four memory banks will be used in the case of a baseline hash function, as shown in Fig. 6a. When the XOR or ADD hash function is applied, all four banks will be used, indicated by the four different colors in each column in Fig. 6b and Fig. 6c.

In case a programmer uses padding (of one) to reduce the number of bank conflicts, the stride in the previous example increases from four to five. This results in a memory access pattern as a diagonal from top left to bottom right in Fig. 6. In the baseline hash function (Fig. 6a) this results in all memory banks being accessed, and consequently no bank conflicts. For the XOR hash function (Fig. 6b) however, only one or two banks are used, resulting in a 4- or 2-way bank conflict respectively for this memory with 4 banks. For the ADD hash function (Fig. 6c) half of the banks is used, resulting a 2-way bank conflict. It is interesting to note that the padding which removes the bank conflicts with the baseline hash causes the bank conflicts for the XOR and ADD hash.

Hardware costs

The hardware costs for the XOR hash function is only one XOR gate for every address bit used in the hash function. The costs for the ADD hash function is an adder for each bit. Since the proposed hash functions in the scratchpad memory use five bits and there are two hash functions applied (one for banks and one for locks), the total costs for the hash function is ten XOR gates or ten adders in total for the XOR and ADD hash function respectively. Compared to the 32 cores with floating point fused multiply-add capabilities, 32K register file and 48kB of scratchpad memory, the extra costs for the hash functions in transistor count is negligible.

The latency costs for the XOR hash function is only one (XOR) gate, where the latency for the ADD hash function is determined by how many bits are used in the hash function. If the adder is implemented as a basic ripple carry adder, the added latency for the first adder is one gate and two gates for the others. Therefore the total gate delay for this 5-bit ADD hash function is $1+4\times 2=9$ gates. Other designs than ripple carry adders exists which can reduce the gate delay. Examples can be found in [22].



(a) baseline (b) XOR (c) ADD

Fig. 6. Overview of bank (and lock) distribution for three different hash functions, the different colors represent the four different memory banks. In (a) the baseline hash function is shown in which the LSB determines the bank. In (b) and (c) the XOR and ADD hash functions are shown in which the LSB are XORed / ADDed respectively with the MSB.

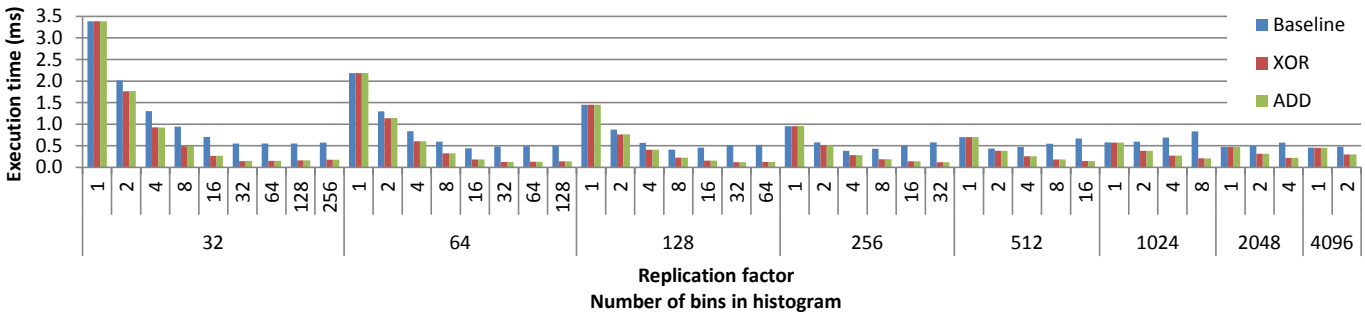


Fig. 7. Histogram execution time, configured for 32 up to 4096 bins. Replication (without padding) is used to improve performance. Results are averaged over 24 12-bit grayscale images [21] of 1536×1024 pixels and are shown for the baseline (blue), XOR (red) and ADD (green) hash function.

V. Evaluation of the hardware improvements

To evaluate the effects of the proposed XOR and ADD hash functions on the number of bank and lock conflicts in strided memory accesses, we first re-evaluate the synthetic benchmarks of Section III. Second the impact on execution time of the proposed hash functions for histogram and Hough transform applications is evaluated.

The simulator used in this section is GPGPU-Sim [4] version 3.2.0. Atomic operations on scratchpad memory have been implemented as described in Section III. The GPU simulated is an NVIDIA GTX 580.

A. Synthetic benchmarks

The simulations of Section III have been repeated with the proposed XOR and ADD hash functions. Both give the same results, due to the access-pattern of these benchmarks and no new conflicts are introduced (see Section IV).

Position conflicts (threads updating the same address) cannot be resolved by changing the addressing of memory banks or locks, therefore the proposed hash functions cannot improve latency in Fig. 5a. The bank conflicts of Fig. 5b can be removed completely by the proposed hash functions, and no new conflicts are introduced.

The proposed hash functions remove all lock conflicts in Fig. 5c, but not all bank conflicts. Some accesses map to the same bank, even if a hash function is applied. For example, in case there is a 2-way bank conflict, the access pattern looks like: 0, 256, 2, 3, 4, ... 31. Without a hash function, thread 0 and thread 1 have a bank conflict at bank 0. With a hash function, address 256 of thread 1 now maps to bank 8, creating a bank conflict with thread 8. So the hash function has not removed the bank conflict, but only moved it from bank 0 to bank 8. In case there is a 3-way conflict, the access pattern looks like: 0, 256, 512, 3, 4, ... 31. Without a hash function, thread 0, 1 and 2 have a bank conflict at bank 0. With a hash function, address 256 gets mapped to bank 8, and address 512 gets mapped to bank 16. Now the conflict degree has been reduced from three to two, as the conflicts between thread 1 vs. thread 8 and thread 2 vs. thread 16 can be resolved concurrently.

B. Histogram

Histogram is a commonly used algorithm in image processing in which a set of bins is filled according to the frequency of occurrence in the input image. The resulting histogram can be used to correct the white balance of the image, for example. The histogram algorithm can also be found in other domains such as finance and statistics.

Pixels next to each other in an image often have the same color, resulting in position conflicts in the histogram algorithm. A software technique to reduce these conflicts is replication [3], [15], [16], [17], in which multiple copies of the histogram are made, reducing the number of concurrent updates on the same memory location. Replication improves performance by reducing the amount of position conflicts, but also creates new bank- and lock conflicts.

The proposed XOR and ADD hash functions can diminish these new conflicts and improve performance further, as is shown in Fig. 7. The histogram application tested is configured to use 32 up to 4096 bins. The replication factor (R) varies from 1 to 256, limited by the total memory requirement, calculated as $4(bytes/word) \times \#bins \times R$. The images used in this evaluation come from the Stanford Center for Image Systems Engineering [21]. They are converted from 24 bit RGB to 12 bit grayscale images with a resolution of 1536×1024 pixels.

Fig. 7 shows a maximum speed-up of the hash functions over the baseline of 4.91× for a 256-bin histogram using a replication factor of 32. For small replication factors the speed-up is low, as most conflicts are position conflicts which cannot be removed by a hash function. For larger replication factors, the memory accesses are spread over a larger part of the scratchpad memory, and bank and lock conflicts can be removed by the hash function.

In case the programmer has applied padding together with replication, baseline performance is better than replication without padding. Still, execution time can be reduced more with the XOR hash, up to 1.79×. Only when the number of bins is 32, equaling the number of banks in the scratchpad memory, the XOR hash worsens execution time. As described in Section IV, the ADD hash does not suffer from this slowdown. In all other cases XOR and ADD hash perform similar.

Bin-stretching [2] is a similar software technique to replication [3], only the ordering of the sub-histogram bins in the scratchpad memory is different, causing fewer bank and lock conflicts than replication does. The hash functions can still reduce the number of remaining bank and lock conflicts, improving performance up to 1.80×.

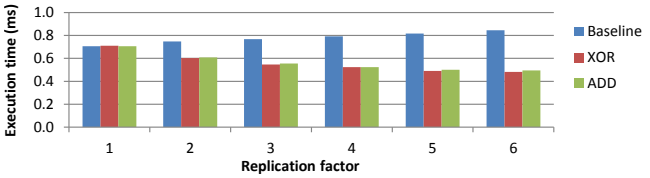


Fig. 8. Hough transform (polar) on a 640×480 image with six different replication factors for the baseline, XOR and ADD hash function.

C. Hough transform

The Hough transform [14] is a commonly used technique to detect lines and other features in images. In a pre-processing step edge detection and thresholding is applied. The coordinates of the remaining pixels are stored in an array [18], which is used in the voting step. The final step in the Hough transform is to find the location of the maximum in the vote space, which indicates the most dominant line in the image. In the voting step the pixel locations in the input array are used to place votes in the polar (instead of Cartesian) 2D-vote space [23] using the following equation: $\rho = x\cos(\theta) + y\sin(\theta)$.

In this example a 640×480 8-bit grayscale image is used as an input. Replication without padding is applied similar to the histogram application. The effect of the hash functions on the execution time of the voting step of the Hough transform is shown in Fig. 8. The maximum speed-up attained is 1.76× for a replication factor of 6. When bin-stretching [2] is used instead of replication [3], no speed-up is attained by the hash functions, but also no slow-down.

VI. Related work

Gou and Gaydadjiev describe the design of the 'elastic pipeline' [24] as a solution to pipeline stalls due to bank conflicts in scratchpad memory. Their focus is on the older NVIDIA G80 / GT200 architecture where memory operations are executed within the cores. In this work the focus is on the more recent Fermi architecture where memory operations are executed in separate load/store units.

Various papers describe techniques to optimize performance for atomic operations, but target only a single application. For example optimizations for a histogram application are discussed in [3], [15], [16], [17] and Hough transform is discussed in [18]. More general (software) techniques, such as replication [5] and bin-stretching [2] are applied to multiple applications. This work describes a modification to the hardware, which leads to increased performance in combination with the aforementioned software techniques.

VII. Conclusions

In this work we have integrated a detailed model of atomic operations in GPGPU-Sim. We have validated the simulator obtaining an absolute error of 2.9% on average for synthetic benchmarks of atomic operations, and a correlation of 99% between a real GPU and GPGPU-Sim for a histogram example. This way, the modified GPGPU-Sim permits a significantly higher accuracy in the simulation of applications using atomic operations. Furthermore, it allows computer architects to propose hardware changes to improve the performance of atomic operations. Following this reasoning, we have presented two hash functions for the GPU's on-chip scratchpad memory's addressing of banks and locks. With negligible hardware costs, a hash function can reduce thread serialization by decreasing bank and lock conflicts, which occur in voting algorithms such as histogram, K-means and (generalized) Hough transform. This results in increased performance, up to 4.9× and 1.8× for histogram and Hough transform applications respectively when the replication software technique is used. When replication with padding, or bin-stretching, is used, the performance increase is smaller, up to 1.8× for histogram and no performance gain or loss for Hough transform. All these test-cases have been executed on the modified GPGPU-Sim simulator.

References

[1] S. Fuller and L. Millett, "Computing Performance: Game Over or Next Level?" *IEEE Computer*, vol. 44, no. 1, pp. 31–38, 2011.

[2] G. J. van den Braak, C. Nugteren, B. Mesman, and H. Corporaal, "GPU-Vote: A Framework for Accelerating Voting Algorithms on GPU," in *Euro-Par*, 2012.

[3] J. Gómez-Luna, J. M. González-Linares, J. I. Benavides, and N. Guil, "An Optimized Approach to Histogram Computation on GPU," *Machine Vision and Applications*, 2012.

[4] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *ISPASS*, 2009.

[5] J. Gómez-Luna, J. M. González-Linares, J. I. Benavides, and N. Guil, "Performance Modeling of Atomic Additions on GPU Scratchpad Memory," *IEEE Trans. on Parallel and Distributed Systems*, 2013.

[6] NVIDIA Corporation, "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," 2009.

[7] ——, "NVIDIA CUDA C Programming Guide 5.0," 2012.

[8] B. Coon, P. Mills, J. Nickolls, and L. Nyland, "Lock mechanism to enable atomic updates to shared memory," *US Patent 8055856*, 2011.

[9] NVIDIA Corporation, "Parallel Thread Execution ISA v3.1," 2012.

[10] "asfermi: An assembler for the NVIDIA Fermi Instruction Set," 2013. [Online]. Available: https://code.google.com/p/asfermi/

[11] Y. Dotsenko, N. K. Govindaraju, P.-P. Sloan, C. Boyd, and J. Manferdelli, "Fast scan algorithms on graphics processors," in *Proceedings of the 22nd annual international conference on Supercomputing*, 2008.

[12] V. Volkov and J. W. Demmel, "Benchmarking GPUs to Tune Dense Linear Algebra," in *ACM/IEEE conference on Supercomputing*, 2008.

[13] Y. Zhang and J. Owens, "A Quantitative Performance Analysis Model for GPU Architectures," in *HPCA*, 2011.

[14] P. Hough, "Method and means for recognising complex patterns," *US Patent 3069654*, 1962.

[15] R. Shams and R. A. Kennedy, "Efficient Histogram Algorithms for NVIDIA CUDA Compatible Devices," in *ICSPCS*, 2007.

[16] V. Podlozhnyuk, *Histogram Calculation in CUDA*, 2007.

[17] C. Nugteren, G. J. van den Braak, H. Corporaal, and B. Mesman, "High Performance Predictable Histogramming on GPUs: Exploring and Evaluating Algorithm Trade-offs," in *GPGPU 4*, 2011.

[18] G. J. van den Braak, C. Nugteren, B. Mesman, and H. Corporaal, "Fast Hough Transform on GPUs: Exploration of Algorithm Trade-Offs," in *Advances Concepts for Intelligent Vision Systems*, 2011.

[19] Z. Zhang, Z. Zhu, and X. Zhang, "A Permutation-based Page Interleaving Scheme to Reduce Row-buffer Conflicts and Exploit Data Locality," in *MICRO 33*, 2000.

[20] H. Vandierendonck and K. De Bosschere, "XOR-Based Hash Functions," *IEEE Transactions on Computers*, vol. 54, no. 7, 2005.

[21] "Stanford Center for Image Systems Engineering (SCIEN)," 2013. [Online]. Available: http://scien.stanford.edu/pages/labsite/scien_test_images_videos.php

[22] N. Burgess, "Fast Ripple-Carry Adders in Standard-Cell CMOS VLSI," in *20th IEEE Symposium on Computer Arithmetic (ARITH)*, 2011.

[23] R. O. Duda and P. E. Hart, "Use of the Hough Transformation to Detect Lines and Curves in Pictures," *Commun. ACM*, vol. 15, January 1972.

[24] C. Gou and G. N. Gaydadjiev, "Elastic Pipeline: Addressing GPU On-chip Shared Memory Bank Conflicts," in *Computing Frontiers*, 2011.