Performance Modeling of Atomic Additions on GPU Scratchpad Memory

Juan Gómez-Luna, José María González-Linares,
José Ignacio Benavides Benítez, and Nicolás Guil Mata

Abstract—GPU application implementations using scatter approaches will fall into write contention due to atomic updates of output elements, if these result from more than one input element. Colliding threads will be serialized, seriously harming performance. Dealing with these issues requires a proper understanding of the behavior of the scratchpad or shared memory under conflicting accesses caused by concurrent threads. Thus, this paper presents an exhaustive microbenchmark-based analysis of atomic additions in shared memory that quantifies the impact of access conflicts on latency and throughput. This analysis has led us to discover the lock mechanism that enables atomic updates to shared memory and to propose a performance model to estimate the latency penalties due to collisions by position or bank conflicts. Then, we have derived experiments from this model that show us the way to optimize applications using atomic operations. Position and bank conflicts can be diminished by replication and padding, respectively. The benefits of such techniques are illustrated with the optimization of two widely used voting processes: the centroid updating step in k-means clustering, and histogram calculation.

Index Terms—Performance model, atomic operations, shared memory, K-means, histogram, CUDA, GPU

1 Introduction

General-purpose computation on Graphics Processing Units (GPGPU) has become a successful trend in high-performance computing thanks to programming environments such as CUDA [1] and OpenCL [2]. They offer a vast number of threads running logically in parallel and executing on hardware resources in a multithreaded manner. Massively parallel applications are benefiting from them and obtaining striking speedups. Such impressive improvements are easily attainable in regular and workload-independent computations, where an output data instance is generated from an input data instance. In these cases, one thread is assigned to one input element in a *scatter* approach. In other cases, where output elements are affected by more than one input instance, a *gather* approach (i.e., one thread per output element) is more profitable, since write contention can be avoided [3].

Nevertheless, in applications with a limited number of output elements a gather approach would be inappropriate, because the reduced number of threads would be insufficient to exploit the vast GPU resources. Thus, a scatter approach will suffer write contention, since memory locations accessed by threads should be updated without interference from other threads. This typically entails a need to serialize memory updates that is generally resolved by using atomic operations. These consist of a memory read, an arithmetic operation, and a memory write. Roughly speaking, serialization will entail a latency penalty that is proportional to the number of colliding threads.

As serialization leads to immense performance bottlenecks on GPUs, its impact on scatter approaches should be alleviated by effective optimization techniques. To use these, a deep understanding of the underlying hardware is necessary. Although the literature on CUDA [4], [5] provides pertinent programming recommendations on code efficiency, hardware details are scarce. Thus, several research studies have focused on modeling GPU performance from a theoretical point of view [6], [7], [8]. Some other studies have conducted quantitative analyzes through microbenchmarking [9], [10], [11]. A more recent study has specifically developed a stochastic model of the memory hierarchy [12]. However, none of these studies have tackled write contention due to the execution of atomic operations.

This study presents a detailed microbenchmark-based analysis of atomic additions in shared memory (scratchpad memory in CUDA) as an example of atomic operations on GPUs. The target is fast on-chip shared memory because it is specially devised for data reuse, which is typical in write contention scenarios where several threads collide while updating a single memory location. Such a collision is called *position conflict*. Our analysis measures latency and throughput in a variety of scenarios. Latency measurements permit us to characterize the atomic addition execution by a warp (i.e., basic Single-instruction Multiple-data unit in CUDA), that is subject to *intrawarp* position conflicts. Moreover, because the shared memory is divided into memory banks, *bank conflicts* appear when two or more threads in a warp access the same bank, causing serialization in memory read or write instructions. Thus, we propose an intrawarp performance model that accurately estimates the latency

- *J. Gómez-Luna and J.I. Benavides Benítez are with the Department of Computer Architecture and Electronics, University of Córdoba, Edificio Leonardo, Campus de Rabanales, 14071 Cordoba, Spain. E-mail: {el1goluj, el1bebej}@uco.es.*
- *J.M. González-Linares and N. Guil Mata are with the Departamento Arquitectura de Computadores, ETSI Informatica, Universidad de Malaga, Campus de Teatinos, 29071 Malaga, Spain. E-mail: {jgl, nguil}@uma.es.*

Manuscript received 10 Oct. 2011; revised 23 Oct. 2012; accepted 26 Oct. 2012; published online 7 Nov. 2012.
Recommended for acceptance by F. Mueller.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2011-10-0760.
Digital Object Identifier no. 10.1109/TPDS.2012.319.

of atomic additions. The impact of collisions between threads belonging to different warps (*interwarp* conflicts) on throughput is also measured.

This performance model can be used to guide the design of optimized implementations, where atomic additions are needed. In this context, voting processes are paradigmatic: thousands of threads voting in a limited number of memory locations, where each vote is carried out by an atomic addition. For example, histogram calculation and the centroid updating step in k-means clustering.

Thus, the main contributions are as follows:

- We present a microbenchmark-based analysis of atomic additions on shared memory of a NVIDIA GPU with Fermi architecture.
- This analysis has led us to discern the lock mechanism that enables atomic updates in shared memory. Thus, we are able to propose an intrawarp performance model for atomic additions.
- We have quantified the impact of interwarp conflicts on throughput. We note that latency hiding conducted by the multithreaded architecture alleviates the penalties due to interwarp conflicts.
- The model is used to derive some implications that help us to optimize voting processes, such as the centroid updating step in k-means clustering, and histogram calculation.

The rest of the paper is organized as follows: Section 2 presents the microbenchmarking of atomic additions in shared memory. Section 3 deals with an experimental analysis of the intrawarp performance model that leads us to optimize voting processes. Section 4 shows how both the centroid updating step in k-means clustering, and histogram calculation can be optimized. Finally, the main conclusions are stated.

2 Microbenchmark-Based Study of Atomic Additions in Shared Memory

Although some valuable studies have used microbenchmarking for studying the GPU architecture [9], [10], [11], the shared memory and specifically the atomic operations have not been analyzed in detail. Thus, we have quantified the impact of atomic additions on performance by measuring their latency and throughput in the presence of position and bank conflicts.

The shared memory is a scratchpad memory divided into equally sized modules, called banks, which can be accessed simultaneously. Successive 32-bit words are assigned to successive banks. If the number of banks is N and A is the address of a word, A resides in bank $A\%N$, where $\%$ stands for modulo operation. This permits a high bandwidth if threads access addresses that fall in distinct memory banks. However, if two addresses of a memory request fall in the same bank, there is a bank conflict and the access has to be serialized. In Fermi devices [13], the shared memory has 32 banks, which is the warp size too. Thus, the granularity of memory requests is 32. Shared memory size is 48 KB in Fermi.

CUDA offers atomic functions in shared memory for devices of compute capability (c.c.) 1.2 and above. 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. The code of an atomic addition for c.c. 2.0 is in Fig. 1. We note that load and store instructions are augmented with lock acquire (LK) and lock release (UL) suffixes. In this way, the load instruction locks shared memory locations until they are unlocked by the store instruction.

```
/*0210*/ LDSLK P0, R7, [R9]; // Read and lock
/*0218*/ @P0 IADD R10, R7, 0x1; // Increment
/*0220*/ @P0 STSUL [R9], R10; // Write and unlock
/*0228*/ @!P0 BRA 0x210; // Conditional branch
```

Fig. 1. Assembly code for an atomic addition on Fermi instruction set (c.c. 2.0).

The lock mechanism that enables atomic updates to shared memory is implemented by a memory lock unit described in [14]. Memory read and write requests 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 to map request memory addresses to lock bit addresses. The hash function may simply use the low bits of the address.

Read instructions return both the data stored at the indicated address and a flag determining if the lock was successfully acquired. Such a flag is related to a predicate register (P0 in Fig. 1). The lock bits are accessed in parallel with memory read and write accesses.

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. 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.

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 the same address, i.e., a position conflict occurs. Moreover, because the thread scheduler of the GPU will be alternatively launching instructions for different warps, interwarp position conflicts might appear: one warp will have to wait until other warp finishes the atomic operation if threads of both warps access the same locations. Thus, we distinguish between intrawarp and interwarp conflicts. In the following sections, they are studied separately. We analyze how they impact on latency and throughput on a NVIDIA GeForce GTX 580 with Fermi architecture. Moreover, we present a procedure to estimate the latency of an atomic addition executed by one warp, which accesses a set of addresses called *warp access pattern*. The microbenchmark methodology we have followed is explained in detail in the supplemental material file, which can be found on the Computer Society Digital Library

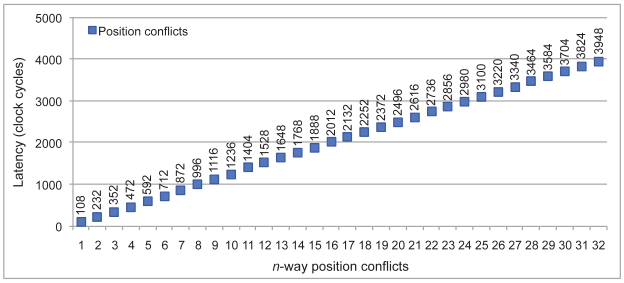


Fig. 2. Latency in clock cycles of an atomic addition with n-way intrawarp position conflicts in GeForce GTX 580. In each test, n threads update the same location.

at http://doi.ieeecomputersociety.org/10.1109/TPDS.2012.319, as well as the warp access patterns used in the experiments. Each pattern contains a certain conflict degree, which is the number of conflicting threads in the intrawarp assessment or the number of conflicting warps in the interwarp assessment.

2.1 Intrawarp Conflicts Assessment

While executing an atomic addition, threads belonging to a warp may suffer a position conflict if they try to access the same address. On the other hand, they may suffer a bank conflict if different accessed addresses belong to the same memory bank. First, we will quantify the impact of position conflicts on latency and throughput, and then we will study how bank conflicts are resolved.

2.1.1 Position Conflicts Microbenchmarking

The impact of intrawarp position conflicts on latency is measured with warp access patterns that result in n-way position conflicts with no bank conflicts. n is given by the number of threads accessing the same address. Fig. 2 presents the latency results. Access without position conflicts ($n = 1$) results in 108 clock cycles. This value is the base latency (t_{base}) for atomic additions. Moreover, it can be observed the gap between two consecutive marks is around 120 clock cycles ($t_{position}$). Thus, the penalty due to an n-way position conflict is $(n-1) \times t_{position}$ clock cycles.

Consistently, the throughput measurement reveals a drastic reduction by 50 percent, when the conflict degree doubles. See the supplemental file, available online, for further details.

We have checked that the former results are independent of the address where conflicts occur. We have also tested many patterns with position conflicts (and no other bank conflicts) in more than one address. Our conclusion is that the exposed latency and throughput are always determined by the address with the highest conflict degree (n).

2.1.2 Bank Conflicts Microbenchmarking

As atomic additions include one shared memory read and one write, we first measure latency penalties on nonatomic read or write accesses due to bank conflicts. We used the access patterns presented in the supplemental file, available online. In both cases, we obtain the penalty increases in steps of t_{bank} (typically 32 clock cycles). This is independent of the stride, which is the distance between addresses accessed by colliding threads.

We then use the same access patterns to estimate the influence of intrawarp bank conflicts on atomic additions. The stride is a multiple of the number of banks between 32 and 1,024. We note that there are two types of bank conflicts:

- If addresses in conflict are at a distance multiple of 1,024 words, the penalty is $t_{bank\text{-}long}$ (typically, 152 clock cycles). We call this *long-latency* bank conflict. For example, if the warp access pattern is $[0, 1{,}024, 2, 3, \ldots, 31]$, the penalty $t_{bank\text{-}long}$ is added to the base latency.
- If addresses in conflict are at a different distance, the latency is increased in $t_{bank\text{-}short}$ (typically, 68 clock cycles). We call it *short-latency* bank conflict. An example is a warp access pattern equal to $[0, 32, 2, 3, \ldots, 31]$: $t_{bank\text{-}short}$ is added to the base latency. This value approximately matches the bank conflict penalty measured in nonatomic read or write: $t_{bank\text{-}short} = 2 \times t_{bank}$.

Moreover, both penalties are increased in steps of t_{extra} (32 clock cycles), whenever a new colliding thread accesses an address at a distance multiple of 1,024 with respect to the addresses being accessed in the two former cases. For instance, a warp access pattern $[0, 1{,}024, 2{,}048, 3, \ldots, 31]$ entails a penalty of $t_{bank\text{-}long} + t_{bank\text{-}long} + t_{extra}$, because thread 2 is accessing an address at distance multiple of 1,024 with respect to addresses 0 and 1,024. If the warp access pattern is $[0, 1{,}024, 2{,}048, 32, 4, \ldots, 31]$, the penalty is $t_{bank\text{-}long} + t_{bank\text{-}long} + t_{extra} + t_{bank\text{-}short}$. The extra penalty appears again with a new colliding thread at distance 1,024 with respect to 32: the warp access pattern $[0, 1{,}024, 2{,}048, 32, 1{,}056, 4, \ldots, 31]$ entails a penalty $t_{bank\text{-}long} + t_{bank\text{-}long} + t_{extra} + t_{bank\text{-}short} + t_{bank\text{-}short} + t_{extra}$.

This behavior is shown in Fig. 3 for two particular cases where the stride takes the values of 32 and 256, respectively.

In the case of a stride equal to 32, the entire range of addresses accessed by the threads of the warp is between address 0 and 1,024 of the shared memory. Therefore, there are no addresses in conflict at distances that are a multiple of 1,024. In this way, the penalty due to an m-way bank conflict is $(m-1) \times t_{bank\text{-}short} = (m-1) \times 2 \times t_{bank}$ clock cycles. These results are shown in Fig. 3(top).

When the stride is 256, the latency function can be approximated by a piecewise linear function whose intervals change at addresses at distances that are a multiple of 1,024 within the same bank. The arrows in Fig. 3(bottom) point to the endpoints of these pieces. Thus, arrow 1 points to the limit between the first and the second pieces and coincides with a new conflict due to two accesses to the same bank with a distance that is a multiple of 1,024. The gap in arrow 2 shows that there is another new conflict in the same bank with a distance that is a multiple of 1,024.

2.1.3 Discussion

As we observe that bank conflicts at distances that are a multiple of 1,024 words are more costly than others on the GeForce GTX 580, we suspect that the lock mechanism in Fermi architecture probably uses 1,024 independent locks. This way, the lock address will be bits 11:2 (see Fig. 2 in supplemental material, available online). Thus, long-latency bank conflicts are due to the limited number of independent

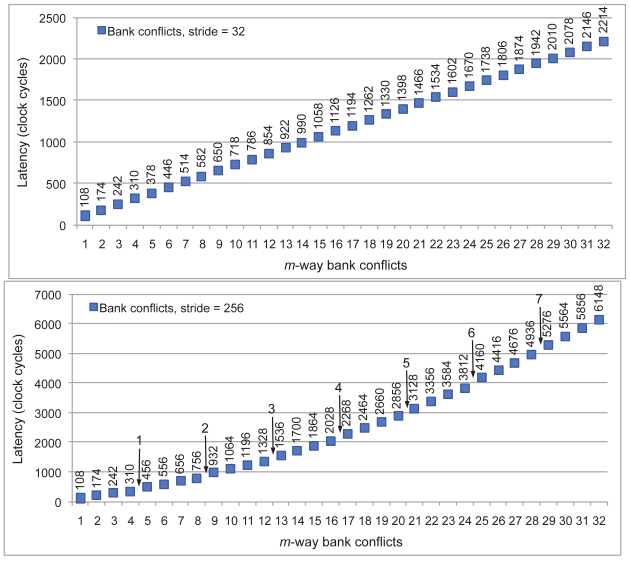


Fig. 3. Latency in clock cycles of an atomic addition with m-way intrawarp bank conflicts in GeForce GTX 580. m is given by the number of threads accessing different addresses in the same bank. The upper part presents results with a stride $= 32$. The gap between two consecutive marks is $t_{bank\text{-}short}$. The lower part shows the results with a stride $= 256$. Gaps are approximately equal between the first four marks ($t_{bank\text{-}short}$). Arrow 1 shows where the gap is significantly higher ($t_{bank\text{-}long}$). Gaps between the second four marks are again approximately equal, but higher than gaps between the first four marks in 32 clock cycles (t_{extra}). Similarly, the gap pointed to by arrow 2 is 32 clock cycles longer than the gap in arrow 1 ($t_{bank\text{-}long} + t_{extra}$).

locks, so that addresses at distances that are a multiple of 1,024 are aliased. Therefore, the program must treat them as if they were position conflicts. In fact, long-latency bank conflicts and position conflicts can be renamed as *lock address conflicts* in different aliased addresses or in the same address, respectively.

Furthermore, since lock bits are accessed in parallel with memory accesses, latency penalties due to bank conflicts will always occur in read access, although the lock is not acquired. For instance, a warp access pattern containing addresses 0 and 1,024 incurs a bank conflict when data is read. As the locks are accessed in parallel and these memory addresses share lock address, only one of the two addresses will finally acquire the lock, but a bank conflict penalty has already been added due to read access. This issue would explain why long-latency bank conflicts are even longer than position conflicts, i.e., $t_{bank\text{-}long} = t_{position} + t_{bank}$ (typically, $152 = 120 + 32$ clock cycles).

The additional penalty t_{extra} can be explained in a similar way. Let us consider a warp access pattern with addresses 0, 1,024, and 2,048. As they are aliased, the code in Fig. 1 will be executed three times. For instance, if the order in which these addresses acquire the shared lock is 0-1,024-2,048, address 1,024 will be read twice and address 2,048 will be read three times. Thus, the penalty $t_{bank\text{-}long}$ due to address 2,048 is increased in t_{bank}. Consequently, $t_{extra} = t_{bank}$. The former issues are summarized in Table 1.

2.1.4 Intrawarp Performance Model

In this section, we present a procedure to determine the latency estimate of atomic additions in shared memory with an arbitrary access pattern. By generalizing the rules detected in the previous sections, we propose Algorithm 1. In each iteration, it calculates the bank conflict degree in the read access, and determines which addresses acquire the locks. Then, it calculates the bank conflict degree in the write access. Finally, it removes those addresses that have been updated from the original set of addresses. A complete example can be seen in the supplemental file, available online.

Algorithm 1. Procedure for determining a latency estimate for a warp access pattern $\mathcal{A}_w$. Algorithm_bank calculates the bank conflict degree of a set of addresses. Algorithm_lock determines the addresses that acquire locks and the maximum number of addresses that share one lock. They are included in the supplemental file, available online.

$lock\_conflict\_degree =$ **Algorithm_lock**$(\mathcal{A}_w)$
$Address[] = \mathcal{A}_w$
for $iteration = 1$ **to** $lock\_conflict\_degree$ **do**
  if $iteration = 1$ **then**
    $Latency = t_{base}$
  else
    $Latency += t_{position}$
  end if
  $bank\_conflict\_degree =$
  Algorithm_bank$(Address[])$
  if $bank\_conflict\_degree > 0$ **then**
    $Latency+ = (bank\_conflict\_degree - 1) \times t_{bank}$
  end if
  $Address\_to\_update[] =$ **Algorithm_lock**$(Address[])$
  $bank\_conflict\_degree =$
  Algorithm_bank$(Address\_to\_update[])$
  if $bank\_conflict\_degree > 0$ **then**
    $Latency += (bank\_conflict\_degree - 1) \times t_{bank}$
  end if
  Remove $Address\_to\_update[]$ **from** $Address[]$
end for
Return $Latency$

In addition, we have evaluated the reliability of the intrawarp performance model with 5,184 different warp access patterns. These tests have successfully shown that

TABLE 1
Summary of Observed Conflicts, Associated Penalties, and Explanation According to the Memory Lock Unit [14]

Observation	Explanation
Position conflict ($t_{position}$)	Lock address conflict in the same address ($t_{position}$)
Short-latency bank conflict ($t_{bank-short}$)	Bank conflict in read and write accesses ($2 \times t_{bank}$)
Long-latency bank conflict ($t_{bank-long}$)	Lock address conflict in aliased addresses ($t_{position} + t_{bank}$)
Extra penalty (t_{extra})	Bank conflict in read access (t_{bank})

```
for(int i=begin; i<data_size; i+=num_threads){
   // Data is read and some computation
   // can be carried out
   int d = data[i];
   int x = function_1(d);
   int y = function_2(d);
   // Vote in shared memory
   atomicAdd(&vote_in_shared[x], y);
}
```

Fig. 4. Code segment of a voting process. Input data `data` are located in global memory. The vote space in shared memory is `vote_in_shared`.

latency estimates match the measured latencies. The median relative error of latency estimates is 1.9 percent.

2.2 Interwarp Conflicts Assessment

Within a Streaming Multiprocessor (SM) the warp scheduler alternates instructions from different warps. While executing atomic operations, one warp may be stalled because of a conflict with another warp. This is what we call an interwarp conflict.

We have carried out two different experiments to measure the throughput in the presence of interwarp conflicts (and the absence of intrawarp conflicts). The first one measures the effect on throughput, if any, of the number of threads in one warp that are colliding with threads in other warps. In the second one, the number of colliding threads in each warp is fixed, and the number of warps with colliding threads is variable. Both experiments reveal that the throughput is independent of the number of colliding threads in each warp (because conflicts in different locations are resolved concurrently), but depends on the number of colliding warps. Moreover, comparing the effect of interwarp and intrawarp conflicts on throughput, we observed that intrawarp conflicts burden much more than inter-warp conflicts. The alternate warp scheduling hides the effect of interwarp conflicts. Extended results can be found in the supplemental file, available online.

3 Experimental Analysis of the Model

The model presented in Section 2.1.4 describes how a warp of threads executes atomic operations on an address set $\mathcal{A}_w$ in shared memory by acquiring lock bits associated with memory positions. The number of lock bits is lower than the number of shared memory addresses; thus, a lock bit is shared by several aliased addresses and a conflict may arise by accessing the same or an aliased memory position. A conflict may also occur by accessing addresses in the same bank of the shared memory.

These three types of conflicts can be summarized as follows: When two or more threads in the same warp access the same bank, a bank conflict occurs. When two or more threads access the same address, a position conflict occurs. Finally, when two or more threads access different aliased addresses, a lock conflict occurs. A position conflict is indeed a particular case of lock conflict that saves some clock cycles thanks to broadcasting in shared memory [4]. As seen in Section 2.2, position and lock conflicts are also possible among threads belonging to different warps. Nevertheless, we focus hereinafter on intra-warp conflicts, which are

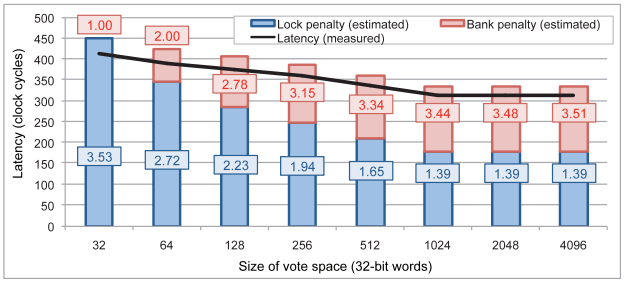


Fig. 5. Average measured latency and estimated latency penalties due to lock and bank conflicts. 1,000,000 random warp access patterns have been used on GeForce GTX 580. The numbers on each column stand for the average of the maximum lock (bottom) and bank (top) conflict degree.

accurately described by the model and cause significantly more impact on throughput than interwarp conflicts.

The number of conflicting threads in a warp is the degree of the conflict. Thus, conflict degree equal to 1 means no conflict. A warp access pattern may have several different conflicts, each with its own conflict degree. Algorithm 1 shows how they occur and the latency penalty they provoke; basically, the highest conflict degree limits the resolution of all the conflicts imposing a latency penalty on the execution time.

In this section, we present three experiments that reveal implications derived from the model that will help us to optimize applications using atomic operations, for example, voting processes such as histogram calculation. In these experiments, one warp of threads carries out atomic additions on a shared memory space of variable size. This shared memory space is composed by consecutive memory positions, corresponding to a vote space in voting processes. Consequently, we refer to this shared memory space as vote space. A general code for a voting process is in Fig. 4.

The first experiment uses random access patterns $\mathcal{A}_w$ to obtain an estimate of the latency penalties caused by bank and lock conflicts. Real access patterns depend on the application using atomic operations, and the real data used, but in a general case we can assume a uniform data distribution over different $\mathcal{A}_w$. Fig. 5 shows the latency and the proportion due to each type of conflict using 1,000,000 warp access patterns. Each column corresponds to a different size of the vote space varying between 32 and 4,096 32-bit words, and the numbers on each column are the averages of the maximum conflict degree. The figure also presents the measured latency, to show the accuracy of the model estimates.

As it can be observed, most of the latency is due to lock (and position) conflicts, even if the bank conflict degree is higher. This is caused by the fact that position and lock conflicts are significantly more costly than bank conflicts. The lock conflict degree between 32 and 1,024 words is entirely due to position conflicts, since there are 1,024 lock bits. Since the probability of position conflict in each $\mathcal{A}_w$ diminishes as the vote space grows, the lock conflict degree decreases. Thus, the latency decreases as well. However, the lock conflict degree is maintained between 1,024 and 4,096, because addresses at distance multiple of 1,024 words (i.e., aliased addresses) appear in $\mathcal{A}_w$.

Hence, a general strategy for reducing the penalty should eliminate or at least reduce the position conflicts. A classic approach is *replication*, which consists of placing $\mathcal{R}$ adjoining copies of the vote space, to spread the accesses over more memory addresses. $\mathcal{R}$ is called replication factor. A final step reduces the copies to calculate the results. Thus, replication is only applicable to associative operations such as addition.

While applying replication, a mapping function is needed to assign to each thread a replicated copy of the vote space, where the thread will perform its atomic operations. Typical mapping functions are *cyclic* and *block*. On the one hand, cyclic mapping makes consecutive threads access consecutive copies: thread $ThId$ will access copy $ThId\%\mathcal{R}$. Each thread will use the offset in (1). On the other hand, block mapping assigns several consecutive threads to the same replicated copy: if N is the thread block size and $\mathcal{R}$ is the replication factor, each $\frac{N}{\mathcal{R}}$ consecutive threads will access the same copy. The offset is given by (2). As indicated above, the experiments in this section use a thread block size $N = 32$, that is, the warp size:

$$Offset\_cyclic(ThId) = Vote\_space\_size \times (ThId\%\mathcal{R}), \quad (1)$$

$$Offset\_block(ThId) = Vote\_space\_size \times \left(\left\lfloor \frac{ThId}{\lfloor \frac{N}{\mathcal{R}} \rfloor} \right\rfloor \%\mathcal{R} \right). \quad (2)$$

When the vote space size is a multiple of the number of shared memory banks, replication makes position conflicts turn into bank conflicts. Anyway, a latency reduction can be expected as position conflicts are more costly than bank conflicts.

The second experiment uses the previous random warp access patterns to study the impact of replication in the reduction of the latency penalty. For the sake of simplicity, we use the maximum possible replication factor for each size of the vote space, because the probability of position conflict is lower in a larger space. The maximum replication factor is limited by the shared memory size. In these tests, we use a replication factor up to 32, which is the number of threads in a warp.

Fig. 6 shows an important decrease in the lock conflict degree in vote spaces under 1,024. In fact, a reduction of the latency is observed in these cases with respect to the results in Fig. 5. Such a reduction is thanks to turning position conflicts into bank conflicts. However, replication has no effect on lock conflict degree in vote spaces equal to or longer than 1,024. This is due to the fact that position conflicts turn into lock conflicts in aliased addresses, because the vote space size is a multiple of the number of locks. Hence, no performance improvement can be expected from replicating beyond a memory space of 1,024 words.

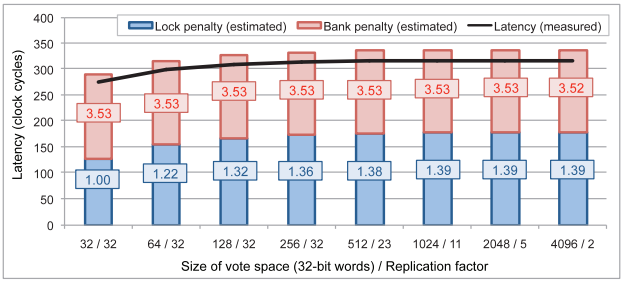


Fig. 6. Average measured latency and estimated latency penalties due to lock and bank conflicts when using the maximum replication factor. 1,000,000 random warp access patterns have been used on GeForce GTX 580. The numbers on each column stand for the average of the maximum lock (bottom) and bank (top) conflict degree.

The former results are identical for both block and cyclic mapping, because of the huge number of random access patterns. Differences in performance of both mapping functions may be noticed with input data that have some type of spatial correlation. To illustrate the impact of spatial correlation, we introduce in the third experiment a sorting stage that sorts every warp access pattern in ascending order before the atomic operation. In this way, possible position conflicts will appear among adjacent threads. The same 1,000,000 random warp access patterns as in previous experiments are used.

Fig. 7 shows the latency for vote spaces between 32 and 4,096 and replication factors between 1 and the corresponding maximum. We test block and cyclic mapping functions with and without the sorting stage. Moreover, we introduce *padding* between consecutive replicated copies of the vote space. Thus, the start address of each copy is shifted. This may strengthen replication by avoiding bank and lock conflicts: if two adjacent threads are to update the same memory position, replication will turn the position conflict into a bank (or a lock) conflict; after introducing padding, the two threads will access different banks (and lock bits). In these tests the pad size is 1, because it is enough to avoid those bank and lock conflicts (derived from replication)

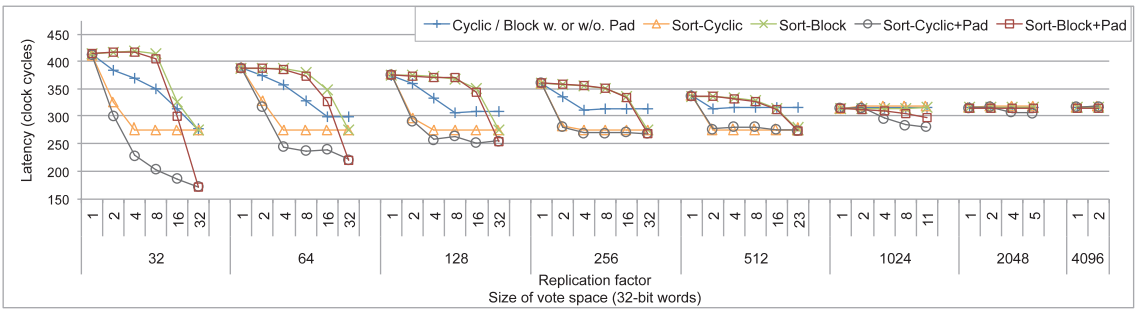


Fig. 7. Average measured latency for 1,000,000 random warp access patterns on GeForce GTX 580. Cyclic and block mapping functions are used. Additionally, a sorting stage and padding may be included. The sorting stage is not included in the latency measurement.

between adjacent threads. Although some new conflicts may arise because of shifted accesses after padding, the probability of conflict is generally much lower, as we have checked using the model.

It can be observed that plain cyclic and block mapping have an identical performance, even if padding is used, because of the random nature of input data. As expected, the lowest latency in these cases is achieved with the highest replication factor that maintains the replicated copies space under 1,024 words.

Cyclic mapping with sorting improves significantly the performance for vote space sizes under 1,024, because replication will turn many position conflicts into bank conflicts. Such an improvement disappears for vote space sizes larger than or equal to 1,024, as lock conflicts appear. The use of padding ensures further improvement, thanks to the avoidance of those bank and lock conflicts caused by replication.

In the case of block mapping with sorting, a low replication factor is not profitable, because neighboring threads will likely access the same replicated copy. Thus, most position conflicts are not avoided. The impact of padding is negligible too, because bank and lock conflicts caused by replication are scarce. Position conflicts between two consecutive threads will only be removed when using a replication factor 32, that is, one copy per thread.

4 Optimizing Voting Processes

Model implications described in the previous section can help us to optimize applications using atomic operations. As the microbenchmarking and the performance model in Section 2 deal with atomic additions, we focus in this section on applications using them and particularly on their voting processes. Representative case studies are the centroid updating step in k-means clustering, and histogram calculation, which are tackled below. Replication approaches with cyclic and block mappings will be applied to both cases. In addition, padding will be used in histogram calculation, to take advantage of the spatial correlation of image pixels.

4.1 K-Means Clustering

K-means clustering [15] is a widely known partition method that classifies a number of input data objects into k clusters. Each cluster is represented by a centroid, i.e., the mean value of all the objects contained in it. The standard algorithm uses an iterative refinement. In each iteration, objects are assigned their nearest centroids based on a similarity function. Once the assignments have been completed, the centroids are recalculated by averaging the object components.

The updating step of the centroids can be considered a voting process. On the one hand, the number of objects assigned to each cluster is counted. On the other hand, each object component is accumulated for the corresponding centroid. Then, the components of the centroid are obtained dividing each accumulated value by the total number of objects within the cluster. Thus, there will be one vote space to count the objects per cluster and as many vote spaces as object components. All these vote spaces will have the same size equal to k.

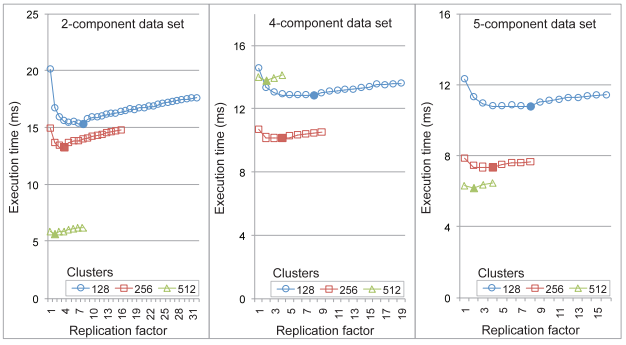


Fig. 8. K-means clustering execution results for three data sets of 2-, 4-, and 5-component objects. The number of clusters is 128, 256 or 512. Abscissas represent the replication factor applied to cluster counter and each centroid component.

4.1.1 GPU Implementation and Evaluation

Since the number of input objects will be usually very large, GPU acceleration will be very desirable for this algorithm. The second step in each iteration, that is, the updating of the centroids, is in the scope of this work, because it needs atomic additions. In this way, we have implemented a scatter approach applying replication in shared memory to the vote spaces (cluster counter and centroid components) and have tested cyclic and block mappings. The replication scheme places the several copies of each vote space consecutively. Thus, the shared memory allocates all copies of the cluster counter, then all copies of the first component, and so on. This scheme is possible thanks to the availability of integer and floating-point atomic additions on Fermi devices [4].

The maximum replication factor $\mathcal{R}$ is dependent on the number of object components in input data set and the number of clusters. For instance, as the shared memory size is 48 KB, the maximum replication factor for 128 clusters and 2-component objects is 32, i.e., 32 copies per vote space. This is indeed the maximum possible replication factor in the following tests.

We have tested cyclic and block mappings with three input data sets of 2-component, 4-component, and 5-component objects, respectively. These data sets are similar to those used in [16], but have been extended with more objects, to leverage fully the enormous computing resources of the GPU. Each data set contains 212,340 objects. The execution configuration is 16 blocks of 1,024 threads, which ensures a minimum of active threads per SM as recommended in CUDA literature [5].

Fig. 8 shows the execution time of the cyclic mapping for 128, 256, and 512 clusters. The best performance is always attained with the replication factor that maintains the memory space used by the copies of each vote space under 1,024 words. In this way, the best replication factor for 128, 256, and 512 clusters is, respectively, 8, 4, and 2. In addition, we have also verified that padding does not improve the performance as input data show a low spatial correlation. These observations entirely agree the implications derived from the model in Section 3.

In the case of block mapping, the number of threads N in (2) is the block size. As we use blocks of 32 warps and the maximum possible replication factor in these tests is 32, each replicated copy is shared by one or more warps. Thus, intrawarp position conflicts are never removed. According to our analysis (see Section 2.2), intrawarp conflicts burden the performance more than interwarp conflicts. Consequently, the performance of the block mapping is lower than the cyclic mapping in all cases (see supplemental material file, available online), as it is derived from our analysis.

4.1.2 The Updating of the Centroids in GPU k-Means

To the best of our knowledge, ours is the first implementation of the centroid updating step on GPU that takes advantage of floating-point atomic additions in shared memory. Most previous works update the centroids on CPU [17], [18], [19], [20], because of the lack of floating-point atomic additions on pre-Fermi devices. The main drawback of these approaches is the overhead dedicated to data transfers from GPU to CPU (after finding the nearest cluster for each object) and from CPU to GPU (after computing the new centroids) in each iteration. In [21], the centroid updating step is also performed on the CPU side, but some acceleration is achieved using asynchronous transfers and CUDA streams.

Other works propose gather approaches on GPU, where one centroid is assigned to one block or one thread, to skip the need for atomic additions. The approach in [22] assigns one centroid to one thread that computes all its new components. Such implementation underutilizes the GPU resources, when the number of centroids can fit in one block. In [23], each block calculates a partial centroid from a subset input objects, and each thread calculates one dimension of the partial centroid. This approach may be burdened by an excessive number of idle threads, if the number of components is low.

The former two works are overcome by the approach in [24]. Although the authors give scarce details about the implementation, they assign input objects to threads. Objects are divided into groups that are distributed to SMs. This way, write conflicts when updating the centroids decrease. Taking into account that this work employs a scatter approach, its design can be seen as a special case of our general proposal.

In the supplemental file, available online, we have compared our GPU implementation with an OpenMP implementation [16]. Our replication scheme attains a minimum speedup 60 compared to the 4-thread OpenMP implementation.

4.2 Histogram Calculation

Histograms are functions that count the number of observations that fall into disjoint categories, known as bins. They permit to estimate the probability distribution of a variable and, in this manner, they are frequently used to obtain the probability density function of the analyzed variable by normalizing the histogram area to 1. Histograms are actively used in many applications, notably in the image processing and pattern recognition fields.

4.2.1 GPU Implementation and Evaluation

GPU implementation of histogram calculation consists of a huge number of threads voting in a limited number of histogram bins, where each vote requires atomicity. Thus, access conflicts will be very frequent. Additionally, in a typical image or video application on GPU, threads belonging to the same warp will read contiguous pixels of an image stored in global memory because such an access pattern fulfills *coalescing* requirements, which permit faster access to global memory [4]. Real images usually present a high spatial correlation of pixels, so that color values of neighboring pixels will be generally assigned to the same histogram bin. Therefore, threads of the same warp will vote in a reduced range of the histogram due to the spatial similarity of the input distribution and position conflicts will be very frequent.

To reduce position conflicts between neighboring pixels, we have applied replication. As previously explained, a number $\mathcal{R}$ of replicated copies per block, called subhistograms, is consecutively allocated in shared memory. Block and cyclic mapping are tested. Moreover, padding is introduced, since it can be very profitable for spatially correlated input data as seen in Section 3. Tests in this section use the Van Hateren's natural image database [25], which contains more than 4,000 monochrome images. Pixels of Van Hateren's images have a resolution of 12 bits. This way, histograms of up to 4,096 bins can be generated.

We have used several execution configurations that follow CUDA literature recommendations to achieve a minimum of active threads [5]. The number of blocks has been changed between 16 and 128, and the number of threads between 128 and 1,024. All of them have demonstrated a similar performance, that is shown in Fig. 9 for 16 blocks of 1,024 threads. It shows the average results for histogram calculation of all Van Hateren's images.

It can be noticed that this figure presents similar trends to Fig. 7, despite that sorting is not applied, but the spatial correlation has a similar effect. In both cyclic and block mappings, the performance is conditioned by the memory space used. These mappings without padding loose their effectiveness when the memory space occupied by the per-block subhistograms is larger than 1,024 32-bit words. As it can be seen for histograms up to 256 bins, this fact is even more harmful for the cyclic mapping, where position conflicts among adjacent threads turn into lock conflicts, which are more costly.

The use of padding is effective in both mappings, particularly in the cyclic mapping under 256 bins. When using low-pixel resolution (5, 6 or 7 bits), the probability of position conflict between consecutive threads will be higher. Most of these conflicts will be intrawarp conflicts. Therefore, the cyclic mapping would turn them to bank or lock conflicts, but padding eliminates them. Intrawarp conflicts are not removed when using block mapping unless the replication factor is more than 32, because thread blocks of 32 warps are used. Moreover, padding in block mapping is only useful when the memory space is longer than 1,024, when some lock conflicts will be removed.

When the pixel resolution is higher, the probability of position conflict between adjoining threads decreases notably. Thus, padding is less effective in cyclic mapping.

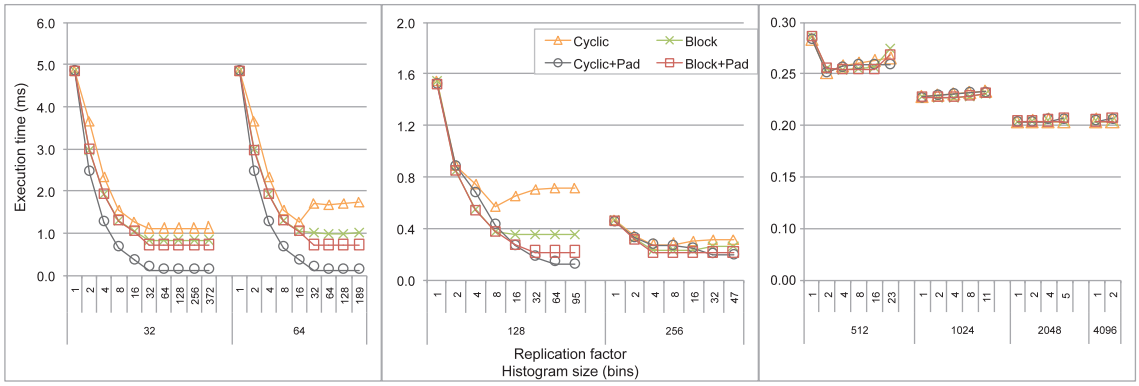


Fig. 9. Histogram calculation execution time for 32 to 4,096 bins. Results for cyclic and block mappings without and with padding are shown. Abscissas represent the per-block replication factor and the histogram size.

Hence, the performance of both mappings is very similar for histograms longer than 256 bins.

4.2.2 GPU Approaches to Histogram Calculation

Several research works have developed implementations of histogram calculation on GPU. Most of these works are based on replication. They assign one replicated copy to one thread or to one warp.

On the one hand, the per-thread approach by Shams and Kennedy [26] declares one subhistogram per thread, which avoids the need for atomic operations, but requires placing a vast number of subhistograms in the high-latency off-chip global memory. Position conflicts are eliminated at the expense of a costly final reduction step. Nugteren et al. [27] propose a per-thread approach using the limited on-chip shared memory, which presents the drawback that the histogram size is limited to 256 bins.

On the other hand, the per-warp approach in [28], [26] places one subhistogram per warp in shared memory. Such approach is indeed a particular case of our replication scheme with block mapping and without padding, using a replication factor equal to the number of warps in each thread block. Our intrawarp performance model predicts threads of a warp might incur many position conflicts due to the typical data distributions in real images. An attempt to overcome this drawback is presented in Nugteren's per-warp approach [27], but it is based on *uncoalesced* global memory accesses that are one of the most undesirable bottlenecks for GPU performance.

Our histogram calculation approach clearly outperforms the former state-of-the-art approaches, as it can be seen in the supplemental material file, available online.

5 Conclusions

This study has presented a microbenchmark-based analysis of atomic additions in GPU shared memory that has permitted us to discern how atomic operations work. There is a lock mechanism that uses a limited number of lock bits, for example, 1,024 in Fermi architecture. This way, lock conflicts occur between addresses at distance multiple of 1,024 words. These are handled as position conflicts with an additional penalty due to the bank conflict.

Therefore, we model the execution of atomic additions in shared memory with a procedure that estimates latency by calculating the number of times that the program repeats the atomic addition code and the bank conflict degree in memory reads and writes. Finally, our analysis shows that interwarp conflicts are less harmful to performance than intrawarp conflicts thanks to the latency hiding in multithreaded architectures.

The performance model leads us to derive experiments that reveal the way to optimize applications using atomic operations. Particularly, we focus on voting processes such as histogram calculation and the updating of centroids in k-means clustering. Replication schemes with cyclic and block mappings are tested. We find that they are profitable when the memory space occupied by the replicated copies is under 1,024 32-bit words. Padding can be successfully used when input data exhibit a spatial correlation, such as in histogram calculation of real images. Moreover, the block mapping performs worse than the cyclic mapping in scenarios with a replication factor under the number of warps, because of the more negative impact of intrawarp conflicts.

Acknowledgments

This study was supported by the Ministry of Education of Spain (TIN2010-16144).

References

[1] NVIDIA, "CUDA Zone," http://developer.nvidia.com/category/zone/cuda-zone, 2011.
[2] Khronos Group, "OpenCL," http://www.khronos.org/opencl/, 2011.
[3] D.B. Kirk and W.-M.W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach,* frist ed. Morgan Kaufmann Publishers Inc., 2010.
[4] NVIDIA, "CUDA C Programming Guide 4.0," http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf, May 2011.
[5] NVIDIA, "CUDA C Best Practices Guide 4.0," http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf, May 2011.
[6] S. Ryoo, C.I. Rodrigues, S.S. Stone, J.A. Stratton, S.-Z. Ueng, S.S. Baghsorkhi, and W.-m. W. Hwu, "Program Optimization Carving for GPU Computing," *J. Parallel Distributed Computing,* vol. 68, no. 10, pp. 1389-1401, 2008.
[7] S.S. Baghsorkhi, M. Delahaye, S.J. Patel, W.D. Gropp, and W.-M.W. Hwu, "An Adaptive Performance Modeling Tool for GPU Architectures," *Proc. 15th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP '10),* pp. 105-114, Feb. 2010.

[8] S. Hong and H. Kim, "An Analytical Model for a gpu Architecture with Memory-Level and Thread-Level Parallelism Awareness," *Proc. 36th Ann. Int'l Symp. Computer Architecture (ISCA '09)*, pp. 152-163, 2009.

[9] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU Microarchitecture through Microbenchmarking," *Proc. IEEE Int'l Symp. Performance Analysis of Systems Software (ISPASS)*, pp. 235-246, Mar. 2010.

[10] Y. Zhang and J.D. Owens, "A Quantitative Performance Analysis Model for GPU Architectures," *Proc. IEEE 17th Int'l Symp. High-Performance Computer Architecture (HPCA 17)*, Feb. 2011.

[11] V. Volkov and J.W. Demmel, "Benchmarking GPUs to Tune Dense Linear Algebra," *Proc. ACM/IEEE Conf. Supercomputing (SC '08)*, pp. 31:1-31:11, 2008.

[12] S.S. Baghsorkhi, I. Gelado, M. Delahaye, and W.-M.W. Hwu, "Efficient Performance Evaluation of Memory Hierarchy for Highly Multithreaded Graphics Processors," *Proc. 17th ACM SIGPLAN Symp. Principles and Practice Parallel Programming (PPoPP '12)*, pp. 23-34, 2012.

[13] NVIDIA, "Fermi Compute Architecture. White Paper," http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2009.

[14] B.W. Coon, J.R. Nickolls, L. Nyland, and P.C. Mills, "Lock Mechanism to Enable Atomic Updates to Shared Memory," Patent, US 8055856, Nov. 2011.

[15] J.B. MacQueen, "Some Methods for Classification and Analysis of Multivariate Observations," *Proc. Fifth Berkeley Symp. Math. Statistics and Probability*, vol. 1, pp. 281-297, 1967.

[16] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary, "Minebench: A Benchmark Suite for Data Mining Workloads," *Proc. IEEE Int'l Symp. Workload Characterization*, pp. 182-188, Oct. 2006.

[17] B. Hong-tao, H. Li-li, O. Dan-tong, L. Zhan-shan, and L. He, "K-Means on Commodity GPUs with CUDA," *Proc. WRI World Congress Computer Science and Information Eng. (CSIE '09)*, vol. 3, pp. 651-655, 2009.

[18] M. Zechner and M. Granitzer, "Accelerating k-Means on the Graphics Processor via CUDA," *Proc. First Int'l Conf. Intensive Applications and Services (INTENSIVE '09)*, pp. 7-15, Apr. 2009.

[19] J. Wu and B. Hong, "An Efficient k-Means Algorithm on CUDA," *Proc. IEEE Int'l Symp. Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW '11)*, pp. 1740-1749, 2011.

[20] J. Sirotkovic, H. Dujmic, and V. Papic, "K-Means Image Segmentation on Massively Parallel GPU Architecture," *Proc. 35th Int'l Convention MIPRO*, pp. 489-494, 2012.

[21] R. Wu, B. Zhang, and M. Hsu, "Clustering Billions of Data Points Using GPUs," *Proc. Combined Workshops UnConventional High Performance Computing Workshop Plus Memory Access Workshop (UCHPC-MAW '09)*, pp. 1-6, 2009.

[22] W. Fang, K.K. Lau, M. Lu, X. Xiao, C.K. Lam, P.Y. Yang, B. He, Q. Luo, P.V. Sander, and K. Yang, "Parallel Data Mining on Graphics Processors," technical report, Hong Kong Univ. of Science and Technology, 2008.

[23] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, and K. Skadron, "A Performance Study of General-Purpose Applications on Graphics Processors Using CUDA," *J. Parallel Distributed Computing*, vol. 68, no. 10, pp. 1370-1380, Oct. 2008.

[24] Y. Li, K. Zhao, X. Chu, and J. Liu, "Speeding up k-Means Algorithm by GPUs," *Proc. IEEE 10th Int'l Conf. Computer and Information Technology (CIT)*, pp. 115-122, 2010.

[25] J.H.v. Hateren and A.v.d. Schaaf, "Independent Component Filters of Natural Images Compared with Simple Cells in Primary Visual Cortex," *Proceedings: Biological Sciences*, vol. 265, no. 1394, pp. 359-366, Mar. 1998.

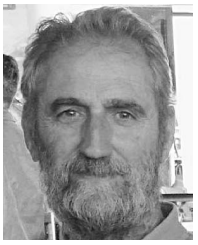
[26] R. Shams and R.A. Kennedy, "Efficient Histogram Algorithms for NVIDIA CUDA Compatible Devices," *Proc. Int'l Conf. Signal Processing and Comm. Systems (ICSPCS)*, pp. 418-422, Dec. 2007.

[27] C. Nugteren, G.-J. van den Braak, H. Corporaal, and B. Mesman, "High Performance Predictable Histogramming on GPUs: Exploring and Evaluating Algorithm Trade-Offs," *Proc. Fourth Workshop General Purpose Processing Graphics Processing Units (GPGPU-4)*, pp. 1:1-1:8, 2011.

[28] V. Podlozhnyuk, "Histogram Calculation in CUDA. White Paper," http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/histogram256/doc/histogram.pdf, 2007.

Juan Gómez-Luna received the BS degree in telecommunication engineering from the University of Sevilla, Spain, in 2001, and the PhD degree in computer science from the University of Córdoba, Spain, in 2012. Since 2005, he has been an assistant professor at the University of Córdoba. His research interests focus on parallelization of image and video processing applications.

José María González-Linares received the BS degree in telecommunication engineering from the University of Málaga, Spain, in 1995, and the PhD degree from the University of Málaga, Spain, in 2000. Since 2002, he has been an associate professor at the University of Málaga. He has published more than 20 papers in international journals and conferences. His research interests are in the areas of parallel computing and video and image processing.



José Ignacio Benavides Benítez received the bachelor's degree in physics from the University of Granada, Spain, in 1980 and the PhD degree in physics from the University of Santiago of Compostela, Spain, in 1990. He joined the University of Córdoba in 1983. He has published more than 50 papers in international journals and conferences.

Nicolás Guil Mata received the BS degree in physics from the University of Sevilla, Spain, in 1986 and the PhD degree in computer science from the University of Málaga in 1995. Currently, he is a full professor with the Department of Computer Architecture at the University of Málaga. He has published more than 50 papers in international journals and conferences. His research interests are in the areas of parallel computing and video and image processing.