Performance modeling of atomic additions on GPU scratchpad memory

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

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 multi-threaded 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 intra-warp position conflicts. Moreover, since 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 intra-warp performance model that accurately estimates the latency of atomic additions. The impact of collisions between threads belonging to different warps (inter-warp...
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 intra-warp performance model for atomic additions.
- We have quantified the impact of inter-warp conflicts on throughput. We note that latency hiding conducted by the multithreaded architecture alleviates the penalties due to inter-warp 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 intra-warp 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 $A \mod N$, 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 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 Figure 1. We note that load and store instructions are augmented with lock acquire (LK) and lock release (UL) suffixed. In this way, the load instruction locks shared memory locations until they are unlocked by the store instruction.

![Assembly code for an atomic addition on Fermi](image)

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 bit 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 Figure 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, since the thread scheduler of the GPU will be alternatively launching instructions for different warps, inter-warp 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 intra-warp and inter-warp 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, 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 intra-warp assessment or the number of conflicting warps in the inter-warp assessment.

2.1 Intra-warp 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 intra-warp 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. Figure 2 presents the latency results. Access without position conflicts \((n = 1)\) results in 108 clock cycles. This value is the base latency \( t_{\text{base}} \) for atomic additions. Moreover, it can be observed the gap between two consecutive marks is around 120 clock cycles \((t_{\text{position}})\). Thus, the penalty due to an \( n \)-way position conflict is \((n - 1) \times t_{\text{position}}\) clock cycles.

Consistently, the throughput measurement reveals a drastic reduction by 50%, when the conflict degree doubles. See the supplemental file 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 non-atomic read or write accesses due to bank conflicts. We used the access patterns presented in the supplemental file. In both cases, we obtain the penalty increases in steps of \( t_{\text{bank}} \) (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 intra-warp bank conflicts on atomic additions. The stride is a multiple of the number of banks between 32 and 1024. We note that there are two types of bank conflicts:

- If addresses in conflict are at a distance multiple of 1024 words, the penalty is \( t_{\text{bank-long}} \) (typically, 152 clock cycles). We call this \( \text{long} \) latency bank conflict. For example, if the warp access pattern is \([0, 1024, 2, 3, \ldots, 31]\), the penalty \( t_{\text{bank-long}} \) is added to the base latency.

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

Moreover, both penalties are increased in steps of \( t_{\text{extra}} \) (32 clock cycles), whenever a new colliding thread accesses an address at a distance multiple of 1024 with respect to the addresses being accessed in the two former cases. For instance, a warp access pattern \([0, 1024, 2048, 3, \ldots, 31]\) entails a penalty of \( t_{\text{bank-long}} + t_{\text{bank-long}} + t_{\text{extra}} \) because thread 2 is accessing an address at distance multiple of 1024 with respect to addresses 0 and 1024. If the warp access pattern is \([0, 1024, 2048, 32, 4, \ldots, 31]\), the penalty is \( t_{\text{bank-long}} + t_{\text{bank-long}} + t_{\text{extra}} + t_{\text{bank-short}} \). The extra penalty appears again with a new colliding thread at distance 1024 with respect to 32: the warp access pattern \([0, 1024, 2048, 32, 1056, 4, \ldots, 31]\) entails a penalty \( t_{\text{bank-long}} + t_{\text{bank-long}} + t_{\text{extra}} + t_{\text{bank-short}} + t_{\text{bank-short}} + t_{\text{extra}} \).

This behavior is shown in Figure 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 1024 of the shared memory. Therefore, there are no addresses in conflict at distances that are a multiple of 1024. In this way, the penalty due to an \( m \)-way bank conflict is \((m - 1) \times t_{\text{bank-short}} = (m - 1) \times 2 \times t_{\text{bank}} \) clock cycles. These results are shown in Figure 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 1024 within the same bank. The arrows in Figure 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...
When data is read. As the locks are accessed in parallel, conflicts will always occur in read access, although the addresses or in the same address, respectively.

Renamed as long-latency bank conflicts and position conflicts can be explained in a similar way. Let us consider a warp access pattern with addresses 0, 1024, and 2048. As they are aliased, the code in Figure 1 will be executed three times. For instance, if the order in which these addresses acquire the shared lock is 0 - 1024 - 2048, address 1024 will be read twice and address 2048 will be read three times. Thus, the penalty $t_{\text{long}}$ due to address 2048 is increased in $t_{\text{long}}$. Consequently, $t_{\text{extra}} = t_{\text{long}}$. The former issues are summarized in Table 1.

### 2.1.4 Intra-warp 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.

In addition, we have evaluated the reliability of the intra-warp performance model with 5184 different warp access patterns. These tests have successfully shown that latency estimates match the measured latencies. The median relative error of latency estimates is 1.9%.

### 2.2 Inter-warp conflicts assessment

Within an 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 inter-warps conflict.

We have carried out two different experiments to measure the throughput in the presence of inter-warps conflicts (and the absence of intra-warps 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 wars. 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 inter-warps and intra-warps conflicts on throughput, we observed that intra-warps conflicts burden much more than inter-warps conflicts. The alternate warp scheduling hides the effect of inter-warps conflicts. Extended results can be found in the supplemental file.
Algorithm 1 Procedure for determining a latency estimate for a warp access pattern $A_w$. Algorithm $\text{bank}$ calculates the bank conflict degree of a set of addresses. Algorithm $\text{lock}$ determines the addresses that acquire locks and the maximum number of addresses that share one lock. They are included in the supplemental file.

\[
\text{lock\_conflict\_degree} = \text{Algorithm\_lock}(A_w)
\]

\[
\text{Address}[] = A_w
\]

\[
\text{for iteration} = 1 \text{ to lock\_conflict\_degree} \text{ do}
\]

\[
\text{if iteration} = 1 \text{ then}
\]

\[
\text{Latency} = t_{\text{base}}
\]

\[
\text{else}
\]

\[
\text{Latency} += t_{\text{position}}
\]

\[
\text{end if}
\]

\[
\text{bank\_conflict\_degree}
\]

\[
\text{Algorithm\_bank}(\text{Address}[])
\]

\[
\text{if bank\_conflict\_degree} > 0 \text{ then}
\]

\[
\text{Latency} += (\text{bank\_conflict\_degree} - 1) \times t_{\text{bank}}
\]

\[
\text{end if}
\]

\[
\text{Address\_to\_update}[] = \text{Algorithm\_lock}(\text{Address}[])
\]

\[
\text{Algorithm\_bank}(\text{Address\_to\_update}[])
\]

\[
\text{if bank\_conflict\_degree} > 0 \text{ then}
\]

\[
\text{Latency} += (\text{bank\_conflict\_degree} - 1) \times t_{\text{bank}}
\]

\[
\text{end if}
\]

\[
\text{Remove Address\_to\_update}[] \text{ from Address}[]
\]

\[
\text{end for}
\]

\[
\text{Return} \text{ Latency}
\]

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 $A_w$ in shared memory by acquiring lock bits associated to 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 accurately described by the model and cause significantly more impact on throughput than inter-warp 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, e.g., 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 Figure 4.

The first experiment uses random access patterns $A_w$ 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 $A_w$. Figure 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 4096 32-bit words, and the numbers on each column are the averages of the maximum conflict degree. The figure also presents the measured latency,
order 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 1024 words is
entirely due to position conflicts, since there are 1024
lock bits. Since the probability of position conflict in each
\(A_w\) 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
1024 and 4096, because addresses at distance multiple of
1024 words (i.e., aliased addresses) appear in \(A_w\).

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
\(R\) adjoining copies of the vote space, in order to
spread the accesses over more memory addresses. \(R\) 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 \(ThId \% R\). Each thread will use the offset in
Equation 1. On the other hand, block mapping assigns
several consecutive threads to the same replicated copy:
if \(N\) is the thread block size and \(R\) is the replication
factor, each \(N\) consecutive threads will access the same
copy. The offset is given by Equation 2. As indicated
above, the experiments in this section use a thread block
size \(N = 32\), that is, the warp size.

\[
\text{Offset}\_\text{cyclic}(ThId) = \text{Vote}\_\text{space}\_\text{size} \times (ThId\% R) \tag{1}
\]

\[
\text{Offset}\_\text{block}(ThId) = \text{Vote}\_\text{space}\_\text{size} \times \left(\frac{ThId}{N}\right) \% R \tag{2}
\]

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, since 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.

Figure 6 shows an important decrease in the lock
conflict degree in vote spaces under 1024. In fact, a
reduction of the latency is observed in these cases with
respect to the results in Figure 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 1024. 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 1024 words.

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 map-
ning functions may be noticed with input data that have
some type of spatial correlation. In order 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.

Figure 7 shows the latency for vote spaces between 32
and 4096 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) 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 1024 words.

Cyclic mapping with sorting improves significantly the performance for vote space sizes under 1024, since replication will turn many position conflicts into bank conflicts. Such an improvement disappears for vote space sizes larger than or equal to 1024, 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, since 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, in order 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 \( k \).

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 \( R \) 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, in order to leverage fully the enormous computing resources of the GPU. Each data set contains 212,340 objects. The execution configuration is 16 blocks of 1024 threads, which ensures a minimum of active threads per SM as recommended in CUDA literature [5].

Figure 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 1024 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 Equation 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, intra-warp position conflicts are never removed. According to our analysis (see Section 2.2) intra-warp conflicts burden the performance more than inter-warp conflicts. Consequently, the performance of the block mapping is lower than the cyclic mapping in all cases (see supplemental material file), 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, in order 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, 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 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.

In order to reduce position conflicts between neighbor-
boring pixels, we have applied replication. As previously
explained, a number \( R \) of replicated copies per
block, called sub-histograms, 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
4000 monochrome images. Pixels of Van Hateren’s im-
ages have a resolution of 12 bits. This way, histograms
of up to 4096 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 1024. All of them have demon-
strated a similar performance, that is shown in Figure 9
for 16 blocks of 1024 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 Figure 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 oc-
ccupied by the per-block sub-histograms is larger than
1024 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 intra-warp
conflicts. Therefore, the cyclic mapping would turn them
to bank or lock conflicts, but padding eliminates them.
Intra-warp 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 1024, when some lock
conflicts will be removed.

When the pixel resolution is higher, the probability of
position conflict between adjoining threads decreases no-
tably. Thus, padding is less effective in cyclic mapping.
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
et al. [26] declares one sub-histogram per thread, which
avoids the need for atomic operations, but requires
placing a vast number of sub-histograms 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 sub-histogram per warp in shared memory.
Such approach is indeed a particular case of our replica-
tion scheme with block mapping and without padding,
using a replication factor equal to the number of warps
in each thread block. Our intra-warp performance model
predicts threads of a warp might incur many position
conflicts due to the typical data distributions in real im-
ages. An attempt to overcome this drawback is presented
in Nugteren’s per-warp approach [27], but it is based on
uncoalesced global memory accesses which are one of
the most undesirable bottlenecks for GPU performance.

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

5 CONCLUSIONS
This study has presented a microbenchmark-based anal-
ysis 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, e.g., 1024 in Fermi architecture. This way, lock
conflicts occur between addresses at distance multiple of
1024 words. These are handled as position conflicts with
an additional penalty due to the bank conflict.

Therefore, we model the execution of atomic addi-
tions 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 inter-warp conflicts are less harmful
to performance than intra-warp conflicts thanks to the
latency hiding in multithreaded architectures.

The performance model leads us to derive experi-
ments that reveal the way to optimize applications using
atomic operations. Particularly, we focus on voting pro-
cesses such as histogram calculation and the updating
of centroids in k-means clustering. Replica 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 1024 32-bit words. Padding
can be successfully used when input data exhibit a spa-
tial 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 intra-warp conflicts.

ACKNOWLEDGMENTS
This study has been supported by the Ministry of Edu-
cation of Spain (TIN2010-16144).
Fig. 9. Histogram calculation execution time for 32 to 4096 bins. Results for cyclic and block mappings without and with padding are shown. Abscissas represent the per-block replication factor and the histogram size.

REFERENCES

Juan Gómez-Luna received the B.S. degree in Telecommunication Engineering from the University of Sevilla, Spain, in 2001. He has obtained the Ph.D. degree in Computer Science from the University of Córdoba, Spain, in 2012. Since 2005, he is 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 B.S. degree in Telecommunication Engineering from the University of Málaga, Spain, in 1995, and the Ph.D. degree from the University of Málaga, Spain, in 2000. Since 2002, he is 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 I. Benavides received the bachelors 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 Gui received the B.S. in Physics from the University of Sevilla, Spain, in 1986 and the Ph.D. in Computer Science from the University of Málaga in 1995. Currently, he is full professor with the Department of Computer Architecture in 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.