Machine Vision and Applications (2013) 24:899–908
DOI 10.1007/s00138-012-0443-3

ORIGINAL PAPER

An optimized approach to histogram computation on GPU

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

Received: 24 November 2011 / Revised: 8 May 2012 / Accepted: 25 June 2012 / Published online: 20 July 2012
© Springer-Verlag 2012

Abstract A histogram is a compact representation of the distribution of data in an image with a full range of applications in diverse fields. Histogram generation is an inherently sequential operation where every pixel votes in a reduced set of bins. This makes finding efficient parallel implementations very desirable but challenging, because on graphics processing units thousands of threads may be atomically updating a short number of histogram bins. Under these circumstances, collisions among threads will be very frequent and such collisions will serialize thread execution, seriously damaging the performance. In this paper we propose a highly optimized approach to histogram calculation, which tackles such performance bottlenecks. It uses histogram replication for eliminating position conflicts, padding to reduce bank conflicts, and an improved access to input data called interleaved read access. Our so-called $\mathcal{R}$-per-block approach to histogram calculation has been successfully compared to the main state-of-the-art works using four histogram-based image processing kernels and two real image databases. Results show that our proposal is between 1.4 and 15.7 faster than every previous implementation for histograms of up to 4,096 bins.

Keywords Histogram · GPU · CUDA · Replication · Padding

J. Gómez-Luna (✉) · J. I. Benavides
Department of Computer Architecture and Electronics, University of Córdoba, Córdoba, Spain
e-mail: el1goluj@uco.es

J. M. González-Linares · N. Guil
Department of Computer Architecture, University of Málaga, Málaga, Spain

1 Introduction

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 one. Histograms are actively used in many applications, notably in the image processing and pattern recognition fields [2,10]. For example, in content-based image retrieval systems it is very common to compute several features, including histograms, of very large image databases that typically include millions of images. Thus, efficient codes for histogram generation are required to compute as fast as possible all these histograms.

One of the most successful trends in high performance computing is general-purpose computation on graphics processing units (GPGPU), thanks to programming environments such as CUDA [8] and OpenCL [3]. Nevertheless, developing efficient codes for histogram generation constitutes a quite challenging task due to the multithreaded architecture of GPUs. Histograms will be generated by thousands of threads voting in a limited number of bins, while atomicity will be required for each vote. This is generally resolved using atomic additions, but these present a considerable objection: if two or more threads try to update the same memory location at the same time, accesses will be serialized. Such a collision is a *position conflict* and the number of colliding threads is the *conflict degree*. Conflicts come from threads belonging to both the same warp (*intra-warp* conflicts) and different warps (*inter-warp* conflicts). Roughly, serialization will entail a latency penalization that is proportional to the conflict degree. In the case of image processing, where typically neighboring pixels will have similar or equal color

values, conflicts will be very frequent, and performance of histogram calculation will be significantly burdened.

An effective technique to reduce the number of position conflicts consists of replicating the histogram, that is placing private copies, called *sub-histograms*, in order to spread the votes along more memory positions. Once the voting step has finished, sub-histograms are reduced into a final histogram. *Replication* has been used in previous main works in histogram generation on CUDA-capable GPUs [4,11,12]. In these works, one sub-histogram is used per thread or per *warp* (i.e., basic Single-instruction Multiple-data unit in CUDA). However, these per-thread and per-warp approaches present several drawbacks, which limit the benefit of replication.

On the one hand, the per-thread approach by Shams et al. [12] 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. [4] propose a per-thread approach in the scarce on-chip *shared* memory, which presents other drawbacks, such as the limited maximum size of a histogram.

On the other hand, the per-warp approach in [11,12] places one sub-histogram per warp in shared memory. This makes necessary the use of atomic additions, since threads of a warp might incur in 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 [4], but it is based on *uncoalesced* global memory accesses, which are one of the most undesirable bottlenecks for GPU performance.

A conceptually different approach is presented by Shams in [13]. This method is based on counting while sorting the input data. Since sorting is a highly optimized technique on GPU, the achieved performance is high, beating the per-warp and per-thread approaches for histograms of more than 10,000 bins.

In this paper we propose a new approach to histogram calculation which applies replication and *padding* for optimizing the voting process in shared memory. Our replication approach declares a number $\mathcal{R}$, called *replication factor*, of sub-histograms per block of threads in shared memory. Adjacent threads will vote in different sub-histograms in order to minimize the number of position conflicts. However, since the shared memory is divided into memory banks [7], if the size of the histogram is a multiple of the number of banks, position conflicts will turn into *bank conflicts* which serialize memory accesses too. Therefore, we propose the use of padding for reducing the amount of bank conflicts. Moreover, a read access optimization, called *interleaved read access*, reduces inter-warp conflicts.

In addition, unlike the former works that experimented solely with reduced ad hoc data sets, we present evaluation results using a very large amount of real images. Shams et al. only used uniform and degenerate (i.e., all input elements set to the same value) data distributions in [12]. In [13] they present a comparison of their per-thread, per-warp and sort-and-count approaches using two 3D medical images from the Vanderbilt database [14]. Nugteren et al. [4] used uniform and degenerate distributions, and four real images.

Therefore, in this work our main contributions are:

- An optimized approach to histogram generation, called *$\mathcal{R}$-per-block*, which applies replication, padding and interleaved read accesses.
- Guidelines for an efficient kernel configuration: number of blocks, number of threads per block and replication factor $\mathcal{R}$.
- Comparison of our approach to the implementations developed by other authors [4,12,13] for histograms of up to 4,096 bins. Tests with four histogram-based kernels, using two natural image databases (monochrome [1] and color [9]) show significant speedups of our approach.[1]

The rest of the paper is organized as follows. In Sect. 2 GPU hardware and atomic additions are introduced. Section 3 depicts our approach to histogram calculation and presents the set of techniques that it uses. Section 4 shows the experimental evaluation of our approach and a comparison to the existing implementations. Finally, the main conclusions and future work are stated.

2 Atomic additions in shared memory

GPUs consist of a high-capacity off-chip global memory and an array of *streaming multi-processors* (SM) [7], which contain processing cores, registers and an on-chip shared memory. From a programmer's point of view, threads are arranged into blocks of size up to 1,024, but they are executed on the SMs as collections of 32 threads called warps. SMs schedule alternate instructions from active warps, in a way that enables hiding long latencies due to memory accesses or read-after-write register dependencies. With this aim CUDA literature [6] recommends a number of active warps over a minimum. Such a number is conditioned by the availability of registers and shared memory.

Threads can access global and shared memories. Since global memory is a hundred of times slower than shared memory, every calculation that requires repeated accesses to certain memory addresses, such as histogram computation, should be performed in shared memory. In this way, we focus on histogram computation in shared memory.

[1] Source code of R-per-block approach to these histogram-based kernels is available at http://www.ac.uma.es/~vip/downloads.html

The shared memory is a scratchpad memory divided into equally-sized memory 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 32-bit word, A resides in bank $\mathrm{mod}(A, N)$, that is, the remainder of the division of A by N. This permits to achieve 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 the current Fermi architecture [5], the shared memory has 32 banks, which is the warp size too. Thus, the granularity of memory requests is 32 and bank conflicts are only possible among threads belonging to the same warp.

For devices of compute capability 1.2 and above, CUDA offers atomic functions which perform a read-modify-write operation on a word residing in shared memory. Specifically, `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. In this regard, threads compete for locking the access to those addresses which are to be atomically updated. This fact exposes the serialization that threads of a warp suffer when a position conflict occurs (i.e., more than one thread try to update the same address). Moreover, since the thread scheduler of the GPU launches alternately instructions for different warps, some of them may compete if they must update the same locations. Thus, some warp may have to wait until other warp finishes the atomic operation. From the former observations, we distinguish between intra-warp and inter-warp position conflicts.

As it can be seen, a proper approach to histogram computation in shared memory should deal with intra-warp position and bank conflicts and inter-warp position conflicts.

3 An optimized approach to histogram generation

The use of atomic additions in shared memory is necessary for designing a histogram calculation approach independent of histogram size, because per-thread approaches are limited by the availability of shared memory [4] or require voting in the slower global memory [12].

On the other hand, the per-warp approach in [12] is subject to many intra-warp position conflicts when working with real images. In a typical image or video application on GPU, threads belonging to the same warp will read contiguous pixels of an image or frame stored in global memory. Such an access pattern is recommended on GPUs in order to fulfill *coalescing* requirements, which permit a faster access to global memory [7]. Real images typically present high spatial correlation of pixels. Thus, color values of neighboring pixels will be generally in the same range. Furthermore, adjacent pixels will often have the same value. For instance, Fig. 1 shows the luminance values of one Lenna's image window. Threads of the same warp will vote in a reduced range of the histogram, due to the spatial similarity of the input distribution. Since these threads vote in the same sub-histogram, position conflicts will be very frequent.

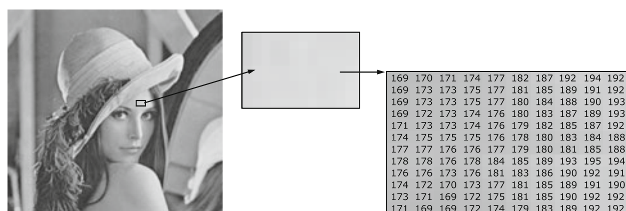

Fig. 1 Detail of a Lenna's grayscale image. Neighboring pixels on her forehead present similar or equal luminance values

In this way, we propose a per-block replication approach that reduces the number of conflicts. Replication is used to turn position conflicts into bank conflicts by making consecutive threads vote in consecutive sub-histograms, as it is explained below. However, bank conflicts entail a latency penalty as well. In this way, padding is necessary to minimize the number of bank conflicts. Finally, we complete our approach proposing an interleaved read access which deals with the access to the input data, and permits to decrease inter-warp conflicts.

Our approach uses a number of blocks whose threads read pixels from global memory and vote in $\mathcal{R}$ sub-histograms in shared memory. It is applicable to histograms up to 4,096 bins on current Fermi GPUs, with bin size equal to 32 bits.

Figure 2 shows the kernel function that implements our approach. It basically consists of three steps: first, threads initialize sub-histograms in shared memory; second, threads read image pixels in an interleaved manner, perform optionally some computation and vote in a number $\mathcal{R}$ of sub-histograms per block, called replication factor; third, the $\mathcal{R}$ sub-histograms per block are reduced, and finally, merged into a final histogram in global memory. This reduction step uses the same code as the per-warp approach [11,12].

3.1 Replication

Replication consists of placing several sub-histograms in shared memory with the aim of reducing or eliminating position conflicts during the voting process.

In this work, we propose a replication approach per block in which consecutive threads belonging to a block will access consecutive sub-histograms in shared memory, as Fig. 3 shows. Thus, if the replication factor is $\mathcal{R}$, thread ThId (such that $0 \leq \mathrm{ThId} \leq \mathrm{block_size}-1$, where block_size is the number of threads within a block) will vote in sub-histogram $\mathrm{mod}(\mathrm{ThId}, \mathcal{R})$. This strategy will mainly permit to reduce the

```
__global__ void Histogram_kernel(int* histo, int* data, int size, int BINS, int R){
   // Declaration of R per-block sub-histograms in shared memory
   // Padding adds 1 to BINS
   __shared__ int Hs[(BINS + 1) * R];

   // Warp index, lane and number of warps per block
   const int warpid = (int)(threadIdx.x / WARP_SIZE);
   const int lane = threadIdx.x % WARP_SIZE; // Operator % expresses mod()
   const int warps_block = blockDim.x / WARP_SIZE;

   // Offset to per-block sub-histogram
   const int off_rep = (BINS + 1) * (threadIdx.x % R);

   // Constants for interleaved read access
   const int begin = (size / warps_block) * warpid + WARP_SIZE * blockIdx.x + lane;
   const int end = (size / warps_block) * (warpid + 1);
   const int step = WARP_SIZE * gridDim.x;

   // Initialization
   for(int pos = threadIdx.x; pos < (BINS + 1) * R; pos += blockDim.x) Hs[pos] = 0;

   __syncthreads(); // Intra-block synchronization

   // Main loop
   for(int i = begin; i < end; i += step){
      int d = data[i]; // Global memory read

      // Optional computation on d can be performed here

      atomicAdd(&Hs[off_rep + d], 1); // Vote in shared memory
   }

   __syncthreads(); // Intra-block synchronization

   // Merge per-block sub-histograms and write to global memory
   for(int pos = threadIdx.x; pos < BINS; pos += blockDim.x){
      int sum = 0;
      for(int base = 0; base < (BINS + 1) * R; base += BINS + 1)
         sum += Hs[base + pos];
      atomicAdd(histo + pos, sum);
   }
}
```

Fig. 2 Kernel code of our $\mathcal{R}$-per-block approach to histogram calculation. `threadIdx.x`, `blockIdx.x`, `blockDim.x` and `gridDim.x` are built-in variables that stand for thread index, block index, block size and number of blocks, respectively. `size` is the image size. Constant `off_rep` indicates in which of the `R` per-block sub-histograms thread `threadIdx.x` will vote. `histo` is the final histogram in global memory

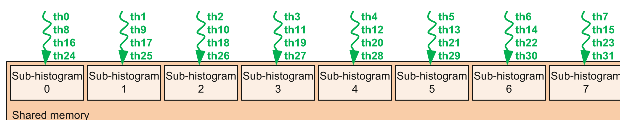

Fig. 3 Replication in shared memory consists of allocating several private copies, called sub-histograms. If the replication factor is 8, thread $ThId$ votes in sub-histogram $mod(ThId, 8)$. Probability of collision among threads of the same warp is reduced by 8

serialization caused by threads of the same warp (i.e., intra-warp conflicts) when updating the same memory location. Moreover, it will also reduce inter-warp conflicts, if the number of sub-histograms is higher than the size of a warp.

The potential benefit of replication can be figured out when observing Fig. 1. Unlike in the per-warp approach, threads in the same warp vote in several different sub-histograms. Hence, the number of position conflicts will significantly decrease.

Such a decrease can be observed in Fig. 4, which presents a quantitative analysis of the number of position conflicts while changing the replication factor. We have studied the pixel distributions of Van Hateren's natural image database [1], in order to count the number of position conflicts while computing 256-bins histograms. We focus on intra-warp position conflicts, so that pixels are analyzed as sets of 32 consecutive ones that will be read by the threads belonging to one warp.

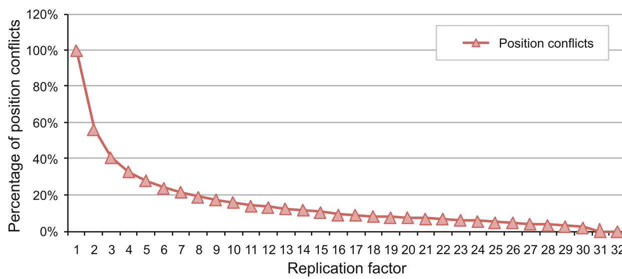

Fig. 4 Percentage of position conflicts when computing 256-bins histograms of Van Hateren's image database, while changing the replication factor between 1 and 32

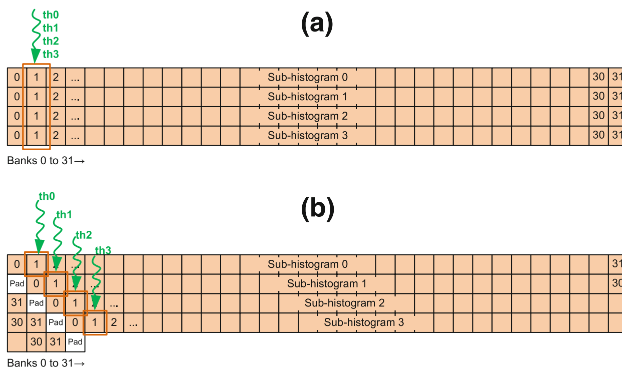

Fig. 5 Degenerate case in a 32-bin histogram in shared memory. The use of replication (**a**) avoids position conflicts but provokes bank conflicts. Therefore, threads 0–3 access bank 1 sequentially. Replication and padding (**b**) make threads vote in different banks in parallel

For each image we computed the number of position conflicts of every degree, and obtained a grand total by weighting these numbers by their degree. Replication factor changes between 1 and 32. As expected, the number of position conflicts diminishes as the replication factor increases. A replication factor equal to 32 ensures a complete removal of the intra-warp position conflicts.

3.2 Padding

As it has been explained, the use of replication in shared memory reduces the number of position collisions. However, position conflicts turn into bank conflicts, that limit the performance as well. This is shown in Fig. 5a. In this regard, the use of padding is recommended to improve the performance. Padding strengthens replication by avoiding bank conflicts when two or more threads of the same warp access the same histogram bin in contiguous sub-histograms in shared memory. Figure 5b explains the use of padding in histogram calculation.

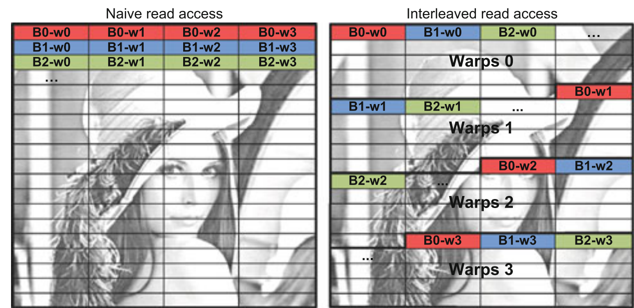

Fig. 6 Naive (*left*) and interleaved (*right*) read accesses. Pixels accessed by warps belonging to different blocks are highlighted in different colors (*red*, *green*, *blue*). $B_i - w_j$ stands for warp j in block i. In the naive access, consecutive warps of a block access consecutive groups of 32 pixels. In the interleaved access, warps w_j only access part j of the image

3.3 Interleaved read access

As it has been seen in Fig. 1, any image is typically composed by many different regions with similar color values. In this way, read access to pixels can have an important influence on how voting is performed, that is, how many position conflicts occur.

When processing an image, read access patterns to global memory typically consist of consecutive threads of a warp reading consecutive pixels in order to take advantage of coalescing. A naive addressing makes also consecutive blocks access consecutive chunks of pixels, as Fig. 6 (left) shows, and consecutive warps access consecutive groups of 32 pixels. In this regard, thread ThId in block B_i will read pixel $B_i \times$ block_size + ThId. Such an access ensures a good performance in most image processing applications, especially if computations are not input dependent. Nevertheless, execution time of histogram generation is dependent on pixel distribution. Thus, since real images are divided into color regions, it is very probable that consecutive warps access pixels with similar or equal color values while using the mentioned naive addressing. In this way, they will incur in many inter-warp conflicts.

For this reason, we propose a read access method that separates warps belonging to the same block as much as possible. This consists of dividing the image in as many parts as warps within a block, so that warp w_i of any block will only access part i of the image. Thread ThId in block B_i will start reading pixel $\frac{\text{image_size}}{\text{warps_per_block}} \times w_i + \text{warp_size} \times B_i + \text{mod}(\text{ThId}, \text{warp_size})$. Figure 6 (right) illustrates the method. This way, probability of inter-warp conflicts will likely decrease. Moreover, this access method ensures coalesced reads to global memory since consecutive threads within a warp read consecutive addresses.

Table 1 Hardware and software features in NVIDIA GeForce GTX 580

Parameter	GeForce GTX 580
Architecture	GF110 (Fermi)
Compute capability	2.0
Multi-processors/GPU	16
Processors/multi-processor	32
Threads/warp	32
Threads/block	Up to 1,024
Threads/multi-processor	Up to 1,536
Blocks/multi-processor	Up to 8
32-Bit registers/multi-processor	32,768
Shared memory/multi-processor	48 Kb
Global memory	1,536 Mb

Table 2 Recommended execution configurations for histogram generation on GeForce GTX 580

	Blocks/SM							
	1	2	3	4	5	6	7	8
Threads	768	384	256	192	192	128	128	128
/Block	1,024	512	384	256	256	192	192	192
		768	512	384		256		

The same number of blocks is used in each SM, in order to ensure load balancing. Moreover, the number of threads per block follows recommendations in CUDA literature [6], that is, a multiple of 64 and a minimum of 768 threads per SM

4 Experimental evaluation

In this section we evaluate our approach to histogram generation, in which kernel code exploits the use of optimization techniques in Sect. 3, and compare it to Shams' and Nugteren's implementations using four histogram-based kernels. As an important novelty in the evaluation of histogram calculation in GPUs, we have used Van Hateren's natural image database [1], which contains 4,164 monochrome images, and McGill's color image data-base [9] with 1,152 images.

Tests in this section use kernel execution configurations (i.e., the number of blocks and the number of threads per block) that are chosen for achieving load balancing across hardware resources following recommendations in CUDA literature [6]. Thus, configuration values are selected to obtain an evenly distributed number of blocks among SMs and a high *occupancy*, which is the ratio between the number of active warps within a SM and the maximum number of active warps. Table 2 collects all the execution configurations that have been used in this work on the device presented in Table 1.

Once determined the execution configuration, a number $\mathcal{R}$ of sub-histograms must be declared per block. In preliminary experiments using Van Hateren's database, we have tested all possible replication factors from 1 to a maximum that does not burden the occupancy, as shown in Sect. 4.1. This maximum is dependent on the size of the histogram, the possible use of padding, the number of blocks per SM and the shared memory size. It is calculated with the following expression:

Table 3 Recommended replication factor $\mathcal{R}$ for histogram generation on GeForce GTX 580

Histogram size	Blocks/SM							
	1	2	3	4	5	6	7	8
32	372	186	124	93	73	62	52	46
64	189	94	63	47	37	31	26	23
128	95	47	31	23	18	15	13	11
256	47	23	15	11	9	7	6	5
512	23	11	7	5	4	3	3	2
768	15	7	5	3	3	2	2	1
1,024	11	5	3	2	2	1	1	1
2,048	5	2	1	1	1			
4,096	2	1						

$\mathcal{R}$ is the maximum replication factor per block that does not burden the occupancy. It is obtained with Eq. 1

$$\mathcal{R} = \frac{\text{ShMem}_{\text{size}}}{B_{\text{SM}} \times (\text{Histogram}_{\text{size}} + 1)} \quad (1)$$

where $\text{ShMem}_{\text{size}}$ is the size of the shared memory in 4-byte words, B_{SM} is the number of blocks per SM and $\text{Histogram}_{\text{size}}$ is the size of the histogram (1 is added if padding is used).

Those preliminary tests led us to use the highest possible replication factor per block which does not reduce the occupancy. This maximum replication factor per block will depend on the number of blocks mapped onto each SM, as presented in Table 3. It should be noticed that the total number of sub-histograms is evenly distributed among the blocks in a SM. For instance, if two blocks are used on each SM, the maximum replication factor will be half the replication factor when only one block is mapped. The highest the total number of sub-histograms, the lowest the probability of conflict. In addition, we have observed that the execution time due to sub-histograms reduction is not significant and does not impact on the overall performance.

In summary, the guidelines for an efficient kernel configuration are:

- The number of blocks is chosen as a multiple of the number of SMs. Thus, the same number of blocks will be mapped on each SM.
- The number of threads per block is a multiple of 64. Moreover, the total number of threads per SM is at least 768.
- The replication factor is given by Eq. 1. Table 3 contains recommended replication factors on GeForce GTX 580.

First, we evaluate the impact of the optimization techniques used in our approach in Sect. 4.1.

Then, we compare our $\mathcal{R}$-per-block approach to Shams' [12,13] and Nugteren's [4] implementations. Shams' and Nugteren's codes are downloadable at the respective authors' sites.[2] Since the name of our approach, $\mathcal{R}$-per-block, remarks the number of sub-histograms used per block, we extend this kind of naming to per-warp and per-thread approaches by calling them *1*-per-warp and *1*-per-thread respectively.

Tests have been performed using the execution configurations in Table 2 and the replication factors in Table 3. It is remarkable that Shams' *1*-per-thread approach works properly only with a power-of-two number of blocks and a power-of-two number of threads. Thus, we have tested 20 execution configurations for $\mathcal{R}$-per-block, Shams' *1*-per-warp and Shams' sort-and-count approaches, while Shams' *1*-per-thread approach has been tested with 30 execution configurations. The number of blocks and the number of threads per block in Nugteren's implementations are fixed, otherwise they do not work correctly. The number of blocks is equal to the image size divided by the number of threads per block. In the case of Nugteren's *1*-per-warp approach, the number of threads per block is fixed to 256. Nugteren's *1*-per-thread implementation uses 32 threads per block.

Histogram-based kernels for monochrome and color images are presented in Sect. 4.2 and 4.3. Results are discussed in Sect. 4.4.

4.1 Impact of the optimization techniques

Figure 7 shows the impact of replication, padding and interleaved read access on GeForce GTX 580. As it can be seen, the use of replication combined with padding makes the execution time diminish as the replication factor increases, following a similar trend to that in Fig. 4. Replication reduces the number of position conflicts, while padding avoids bank conflicts. Moreover, the interleaved read access reduces the execution time due to the reduction of the number of interwarp conflicts. These observations have been ratified for histogram sizes between 32 and 4,096, and for every execution configuration in Table 2.

4.2 Histogram calculation of monochrome images

This kernel calculates a histogram for a monochrome image. We have used all 4,164 1,536×1,024, 12-bit depth, images of Van Hateren's database. This depth permits to experiment with histograms of 32–4,096 bins length. We have measured the number of gigabytes per second processed for every approach and every histogram size. Table 4 presents an average value, obtained with all the execution configurations tested, and the best performance value (in parentheses).

[2] We have updated Shams' per-warp code in order to use hardware atomic additions that replace the original simulated ones. Sort-and-count code has been re-implemented using explanations and code included in [13].

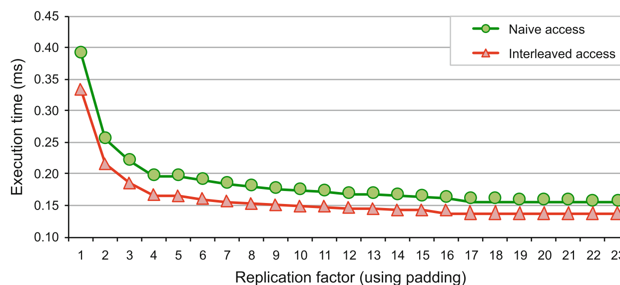

Fig. 7 Average execution time (ms) for 256-bins histogram calculation of images from Van Hateren's database on GeForce GTX 580, while changing the replication factor. The interleaved read access introduces an additional improvement. Results correspond to an execution configuration of 32 blocks of 384 threads, and a maximum $\mathcal{R}$ per block of 23

4.3 Histogram-based kernels for color images

We have implemented three common histogram-based kernels. Our $\mathcal{R}$-per-block approach to these kernels is compared to Shams' and Nugteren's approaches using all 1,152 2,560 × 1,920 RGB images of McGill's database. Table 5 shows average and minimum execution times of all the approaches.

First kernel consists of converting a RGB image to gray scale and then voting in a 256-bin histogram.

Second kernel generates the direct color histogram of a RGB image. The size of the histogram depends on the resolution of the RGB color space. We have considered two resolutions of 8 and 16 levels per color component. These values entail two histogram sizes of 512 and 4,096 bins respectively.

Third kernel calculates three color histograms, one per color component. This is equivalent to computing a histogram of $3 \times 256 = 768$ bins.

4.4 Discussion

Results in Tables 4 and 5 show that our $\mathcal{R}$-per-block approach clearly outperforms the rest of approaches.

In the case of histogram calculation of monochrome images, the best performance of our $\mathcal{R}$-per-block approach obtains a speedup with respect to the best performance of Shams' *1*-per-thread approach, which is the best of the rest of approaches, between 1.6 and 2.8. Moreover, our $\mathcal{R}$-per-block approach is much more stable along execution configurations: the coefficient of variation (i.e., the ratio of the standard deviation to the mean) for every histogram size is between 6 and 25 %, while it is between 71 and 81 % for Shams' *1*-per-thread approach. Thus, our algorithm does not need to

Table 4 Average performance in gigabytes per second for $\mathcal{R}$-per-block, Shams' and Nugteren's approaches to histogram calculation on GeForce GTX 580

Histogram size (Bins)	Performance (GB/s)					
	Our approach	Shams' approaches			Nugteren's approaches	
	$\mathcal{R}$-per-block	1-per-warp	1-per-thread	Sort-and-count	1-per-warp	1-per-thread
32	51.8 (66.5)	19.0 (21.0)	14.6 (41.6)	3.0 (3.8)		
64	58.2 (63.9)	21.7 (24.1)	12.8 (41.3)	3.0 (3.7)		
128	58.1 (64.2)	23.8 (27.7)	11.1 (32.7)	3.0 (3.7)		
256	50.0 (54.5)	21.6 (26.3)	9.2 (27.5)	3.0 (3.7)	15.9 (15.9)	22.4 (22.4)
512	40.8 (43.6)	17.5 (21.1)	7.2 (22.0)	3.0 (3.7)		
1,024	32.1 (39.3)	7.9 (12.1)	5.3 (15.8)	3.0 (3.7)		
2,048	25.6 (36.9)	7.0 (7.5)	3.9 (11.5)	3.0 (3.7)		
4,096	19.7 (21.9)		2.6 (7.7)	2.8 (3.7)		

Best performance values are in parentheses

Table 5 Average and minimum (in parentheses) execution times per image in milliseconds for $\mathcal{R}$-per-block, Shams' and Nugteren's approaches to three histogram-based kernels on GeForce GTX 580

Histogram-based kernals	Execution time (ms)					
	Our approach	Shams' approaches			Nugteren's approaches	
	$\mathcal{R}$-per-block	1-per-warp	1-per-thread	Sort-and-count	1-per-warp	1-per-thread
RGB to grayscale	0.51 (0.48)	0.70 (0.66)	4.55 (2.94)	8.04 (6.05)	1.73 (1.73)	3.15 (3.15)
Direct color						
8 levels/component	0.73 (0.54)	2.74 (2.43)	3.75 (1.09)	8.04 (6.22)		
16 levels/component	1.65 (1.54)		5.28 (2.53)	8.43 (6.28)		
Color histograms	0.88 (0.78)	1.67 (1.43)	13.40 (7.56)	23.16 (17.73)	5.06 (5.06)	5.55 (5.55)

be optimally tuned to obtain a good performance. This compensates for the impossibility of determining in advance the best execution configuration while applying our $\mathcal{R}$-per-block approach, that is caused by the strong variability of pixel distributions within images.

The sort-and-count approach gives a very flat performance which is independent of histogram size and data distribution due to the use of a sorting procedure. It is a specially interesting approach for very big histograms. In fact, it outperforms Shams' 1-per-thread approach for monochrome histogram of 4,096 bins in average.

Shams' 1-per-warp approach reports good performance values in RGB to grayscale conversion and color histograms kernels although it is burdened by intra-warp conflicts. The author reported a good performance with uniform data distributions [12], but this is far from real conditions in image processing as explained with Fig. 1.

Nugteren's implementations work only for 256 bins histograms. Despite that the authors proclaimed performance improvements with respect to previous implementations [4], they did not compare their implementations to the latest ones by Shams. Together with the rigid establishment of the number of blocks and threads, Nugteren's 1-per-warp approach is burdened by the use of two separate kernels: the first one for voting and the second one for reducing the sub-histograms. This corresponds to the original CUDA SDK implementation of 256 bins histogram [11], which was later improved to use one single kernel. Nugteren's 1-per-thread approach performs better but does not improve the best performance of the latest Shams' 1-per-thread implementation. A severe drawback of this method is the fixed block size of 32 threads which makes possible to place 3 blocks per SM. This means only 96 active threads per SM, which is a too low occupancy for Fermi devices.

5 Conclusions and future work

This work has presented a highly optimized approach to histogram calculation on GPU, called $\mathcal{R}$-per-block approach. This approach uses several optimization techniques that overcome the drawbacks of previous per-warp and per-thread implementations. Our approach applies a histogram replication scheme, devised for eliminating position conflicts among

consecutive threads that are typical in histogram calculation of real images. Thus, position conflicts are turned into bank conflicts, and their associated penalties are further reduced using padding. Moreover, an interleaved read access diminishes inter-warp conflicts.

Our approach is completed by giving some recommended execution configurations and replication factors $\mathcal{R}$ that ensure best performance rates.

Finally, we have carried out an exhaustive comparison with the main state-of-the-art implementations using four histogram-based kernels and two natural image databases. Our $\mathcal{R}$-per-block approach is between 1.4 and 15.7 faster than the rest of implementations.

As our $\mathcal{R}$-per-block approach is only applicable to histograms of up to 4,096 bins, due to the limited size of the shared memory, we plan to design a new approach in global memory. Thus, larger histograms could be calculated.

Acknowledgments This study has been supported by the Government of Spain (TIN2010-16144).

References

1. van Hateren, J.H., van der Schaaf, A.: Independent component filters of natural images compared with simple cells in primary visual cortex. Proc. Biol. Sci. **265**(1394), 359–366 (1998)
2. Idris, F., Panchanathan, S.: Review of image and video indexing techniques. J. Vis. Commun. Image Represent. **8**(2), 146–166 (1997). doi:10.1006/jvci.1997.0355. http://www.sciencedirect.com/science/article/B6WMK-45KKSGK-5/2/df25df5374b5ce44616de5550980b9d2
3. Khronos group: OpenCL (2011). http://www.khronos.org/opencl/
4. Nugteren, C., van den Braak, G.J., Corporaal, H., Mesman, B.: High performance predictable histogramming on gpus: exploring and evaluating algorithm trade-offs. In: Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units, GPGPU-4, pp. 1:1–1:8. ACM, New York (2011). http://doi.acm.org/10.1145/1964179.1964181
5. NVIDIA: Fermi compute architecture. White paper (2009). http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf
6. NVIDIA: CUDA C Best Practices Guide 4.0 (2011). http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf
7. NVIDIA: CUDA C Programming Guide 4.0 (2011). http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf
8. NVIDIA: CUDA Zone (2011). http://developer.nvidia.com/category/zone/cuda-zone
9. Olmos, A., Frederick, A.: A biologically inspired algorithm for the recovery of shading and reflectance images. Perception **33**(12), 1463 (2004)
10. Pal, N.R., Pal, S.K.: A review on image segmentation techniques. Pattern Recogn. **26**(9), 1277–1294 (1993). http://www.scopus.com. Cited by (since 1996): 975
11. Podlozhnyuk, V.: Histogram calculation in CUDA. White paper (2007). http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/histogram256/doc/histogram.pdf
12. Shams, R., Kennedy, R.A.: Efficient histogram algorithms for NVIDIA CUDA compatible devices. In: Proc. Int. Conf. on Signal Processing and Communications Systems (ICSPCS), pp. 418–422. Gold Coast, Australia (2007)
13. Shams, R., Sadeghi, P., Kennedy, R.A., Hartley, R.: Parallel computation of mutual information on the GPU with application to real-time registration of 3D medical images. Comput. Methods Programs Biomed. **99**(2), 133–146 (2010)
14. West, J., Fitzpatrick, J.M., Wang, M.Y., Dawant, B.M., Maurer, C.R., Kessler, R.M., Maciunas, R.J., Barillot, C., Lemoine, D., Collignon, A., Maes, F., Sumanaweera, T.S., Harkness, B., Hemler, P.F., Hill, D.L.G., Hawkes, D.J., Studholme, C., Maintz, J.B.A., Viergever, M.A., Mal, G., Pennec, X., Noz, M.E., Maguire, G.Q., Pollack, M., Pelizzari, C.A., Robb, R.A., Hanson, D., Woods, R.P.: Comparison and evaluation of retrospective intermodality brain image registration techniques. J. Comput. Assist. Tomogr. **21**, 554–566 (1997)

Author Biographies

Juan Gómez-Luna received his B.S. degree in telecommunication engineering from the University of Sevilla, Spain, in 2001. He obtained his 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 on GPU.

José María González-Linares received his B.S. degree in telecommunication engineering from the University of Málaga, Spain, in 1995, and 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 Ignacio Benavides received his bachelor's degree in physics from the University of Granada, Spain, in 1980 and Ph.D. 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 received his B.S. in physics from the University of Sevilla, Spain, in 1986 and Ph.D. degree 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.