

# Parallelization of the Generalized Hough Transform on GPU

Juan Gómez-Luna<sup>1a</sup>, José María González-Linares<sup>b</sup>, José Ignacio Benavides<sup>a</sup>, Emilio L. Zapata<sup>b</sup> and Nicolás Guil<sup>b</sup>

*Abstract*—Programs developed under the Compute Unified Device Architecture (CUDA) obtain the highest performance rate, when the exploitation of hardware resources on a Graphics Processing Unit (GPU) is maximized. In order to achieve this purpose, load balancing among threads and a high value of processor occupancy, i.e. the ratio of active threads, are indispensable. However, in certain applications, an optimally balanced implementation may limit the occupancy, due to a greater need of registers and shared memory. This is the case of the Fast Generalized Hough Transform (Fast GHT), an image processing technique for localizing an object within an image. In this work, we present two parallelization alternatives for the Fast GHT, one that optimizes the load balancing and another that maximizes the occupancy. We have compared them using a large amount of real images to test their strong and weak points and we have drawn several conclusions about under which conditions it is better to use one or another. We have also tackled several parallelization problems related to sparse data distribution, divergent execution paths and irregular memory access patterns in updating operations by proposing a set of generic techniques as compacting, sorting and memory storage replication.

*Keywords*—GPU, CUDA, occupancy, load balancing, Generalized Hough Transform.

## I. INTRODUCTION

GRAPHICS Processing Units (GPUs) have emerged as general purpose coprocessors in the last years. An extensive variety of applications is nowadays benefiting from their impressive potential, especially after the launch of CUDA [9]. GPUs offer a huge amount of computing threads, arranged in a Single-Program Multiple-Data (SPMD) model. Such extensive resources make them attractive for general-purpose computations. This interest has boosted the development of GPU programming tools, such as CUDA and OpenCL [10]. General programming recommendations are optimizing load balancing and increasing processor occupancy. However, depending on the algorithm structure, both recommendations cannot be applied simultaneously. Then some kind of tradeoff must be undertaken, since an optimally balanced implementation may increase the use of registers and the need for sharing data among threads, what decreases occupancy. Moreover, parallelization becomes even more challenging, if the algorithm presents workload-dependent computations and non-linear memory references. The former provokes

divergence among threads, if the layout is not carefully planned. The latter affects the locality of references, what entails serialized memory accesses.

In this work, we will discuss how performance problems caused by previous considerations can be mitigated using suitable strategies. They will be illustrated by implementing an efficient parallelization of the GHT. We conduct an exhaustive analysis of the previous considerations that leads us to the following results:

- We propose compacting and sorting, in order to reduce accesses to global memory and the number of executed instructions.
- We present two efficient mechanisms for distributing two sorted lists among blocks and threads. One of them optimizes the load balancing, whilst the other maintains the occupancy at the highest possible values.
- We use replicated sub-histograms per block with successful results.

The rest of the paper is organized as follows. Section 2 depicts the characteristics of regular and irregular applications. Section 3 presents the GHT. Section 4 describes our proposals for an efficient implementation of irregular parts. In section 5, we propose the use of replicated sub-histograms per block, in order to improve a voting process. Section 6 presents the execution results. Finally, section 7 concludes the paper.

## II. REGULAR AND IRREGULAR PROBLEMS

Parallelizing any application on any parallel platform requires programmers to apply a certain level of abstraction. The change from a sequential conception to a parallel conception is never trivial. However, some algorithms are regular in the sense that they use linear addressing and apply the same computation on every instance of the input data. This inherent parallelism makes those algorithms to be easier to implement on a SPMD platform, as is a GPU. In this regard, many image and video processing applications exhibit regular computation patterns and regular memory accesses. The CUDA SDK includes some samples of regular image processing.

Following the optimization recommendations when a regular problem is parallelized, ensures a good performance and impressive speedups on CUDA-enabled GPUs. However, achieving an important improvement with the implementation of an irregular problem is always harder. The distribution of work and data in such algorithms cannot be characterized a priori, because these quantities are input-dependent and evolve with the computation itself. Algorithms with these properties yield performance problems for parallel implementations, where equal distribution of work over

---

<sup>1</sup>Corresponding author: elgoluj@uco.es

<sup>a</sup>Computer Architecture and Electronics Department, University of Córdoba, Córdoba, Spain

<sup>b</sup>Computer Architecture Department, University of Málaga, Málaga, Spain

processors and locality of reference are required within each processor. In this way, programmers should carefully analyze the memory access patterns, in order to avoid penalties due to uncoalescing or bank conflicts. Idle threads and warp divergence should be minimized by properly planning of work distribution and load balancing across threads and blocks.

### III. PARALLELIZING A COMPLEX IMAGE AND VIDEO PROCESSING APPLICATION

This work illustrates different strategies to cope with sparse data array access, divergent execution paths and irregular memory access patterns in GPU applications using, as a case study, a version of the Generalized Hough Transform, called Fast GHT. As it is explained next, this technique presents regular as well as irregular components.

#### A. The Generalized Hough Transform

Template matching is a difficult problem with high computational requirements. One of the most popular algorithms for detecting shapes in images is the Hough Transform [8]. Ballard [1] generalized the Hough Transform (GHT) to detect arbitrary shapes, which are represented by a template. In the original formulation, called Classic GHT, a feature space (composed of the template contour points and their vectors to a reference point) is transformed into a four-dimensional Hough space (the rotation, scale and displacement of the template in the image). The maximum value in this Hough space corresponds to the rotation, scale and displacement parameters of the template in the image.

The size of the Hough space and the number of voting operations can be enormous. Thus, the computation time of the Classic GHT is very high, making it inappropriate for real-time applications. A solution to reduce the memory and computational requirements were presented by Guil et al [6]. In that work, the detection process is split into three stages by uncoupling the rotation, scale and displacement calculation using invariant information. Three transforms are applied in this version, called Fast GHT, to obtain the rotation, scale and displacement parameters.

The invariant features selected in that work are pairings of contour points,  $p_i$  and  $p_j$ , whose gradient angles,  $\theta_i$  and  $\theta_j$ , are separated by a given difference angle  $\xi$ . For every pairing a spatial angle,  $\alpha_{ij}$ , a distance value,  $d_{ij}$ , and reference vectors,  $\vec{r}_i$  and  $\vec{r}_j$ , are computed as shown in Figure 1. The feature space (composed of the pairings with their gradient angles, spatial angles, distances and vectors) is transformed in a two-dimensional Hough space (the gradient and spatial angles) with every pairing voting in the bin with the same gradient and spatial angle. The Hough spaces of the template and the image can be compared using a special cross-correlation function whose maximum value is located in the rotation value,  $\beta$ , of the template in the image.

Next, the gradient angles of the pairings in the template are rotated  $\beta$  degrees and a new transform in a one-dimensional Hough space (the scale parameter) is

applied. Every pairing in the template and the image feature space with the same gradient and spatial angles are selected and the quotient of their distances is used to vote in the Hough space. The position of the maximum of the Hough space is the scale parameter. Finally, the reference vectors of the pairings in the template are rotated and scaled using the calculated parameters, and a transform in a two-dimensional Hough space (the displacement coordinates) is computed. Pairings with the same gradient and spatial angles are selected and the vectors superimposed to vote in the Hough space whose maximum corresponds to the position of the template in the image.

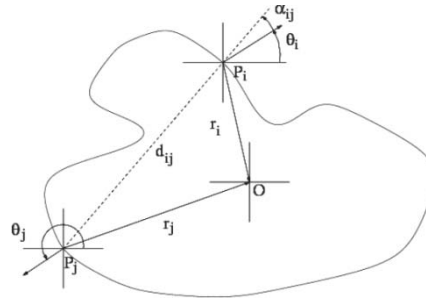


Fig. 1. Variables defined in the GHT

Let  $\mathcal{T}$  be the template,  $\mathcal{I}$  the image,  $(x_i, y_i)$  the coordinates of an edge point  $p_i$ ,  $\xi$  the difference angle,  $\mathcal{O}$ ,  $\mathcal{S}$  and  $\mathcal{D}$  the Hough spaces to compute orientation, scale and displacement respectively, and  $\text{maxi}(M)$  a function that returns the index where the maximum value of  $M$  takes place, the algorithm steps in pseudo-code are

1. Compute contour points  $p_i^{\mathcal{T}} \stackrel{\text{def}}{=} \{x_i, y_i, \theta_i\}$  in  $\mathcal{T}$
2. For each pairing  $\{p_i^{\mathcal{T}}, p_j^{\mathcal{T}}\}$  with  $\theta_i - \theta_j = \xi$ , compute  $p_{ij}^{\mathcal{T}} \stackrel{\text{def}}{=} \{\alpha_{ij}, d_{ij}, \vec{r}_i, \vec{r}_j\}$
3. For each  $p_{ij}^{\mathcal{T}}$  increment  $\mathcal{O}^{\mathcal{T}}(\theta_i, \alpha_{ij})$
4. Repeat steps 1, 2, 3 for  $\mathcal{I}$  to obtain  $p_{ij}^{\mathcal{I}}$ ,  $p_{ij}^{\mathcal{I}}$  and  $\mathcal{O}^{\mathcal{I}}$
5.  $\beta = \text{maxi}(\text{corr}(\mathcal{O}^{\mathcal{I}}, \mathcal{O}^{\mathcal{T}}))$
6. Rotate template contour points  $p_i^{\mathcal{T}} \stackrel{\text{def}}{=} \{x_i, y_i, \theta_i + \beta\}$
7. For each  $\{p_{ij}^{\mathcal{T}}, p_{kl}^{\mathcal{T}}\}$  with  $\theta_i = \theta_k$  and  $\alpha_{ij} - \alpha_{kl}$  increment  $\mathcal{S}(d_{ij}, d_{kl})$
8.  $\zeta = \text{maxi}(\mathcal{S})$
9. Scale vectors in  $p_{ij}^{\mathcal{T}}$  using  $\zeta$
10. For each  $\{p_{ij}^{\mathcal{T}}, p_{kl}^{\mathcal{T}}\}$  with  $\theta_i = \theta_k$  and  $\alpha_{ij} - \alpha_{kl}$ , increment  $\mathcal{D}((x_k, y_k) + \vec{r}_i)$ ,  $\mathcal{D}((x_k, y_k) + \vec{r}_j)$ ,  $\mathcal{D}((x_i, y_i) + \vec{r}_i)$  and  $\mathcal{D}((x_i, y_i) + \vec{r}_j)$
11.  $(\delta_x, \delta_y) = \text{maxi}(\mathcal{D})$

#### B. Regular computation within the GHT

Edge detection and correlation, applied in steps 1 and 5 respectively, exhibit a regular parallelism, since a simple workload distribution guarantees a good load

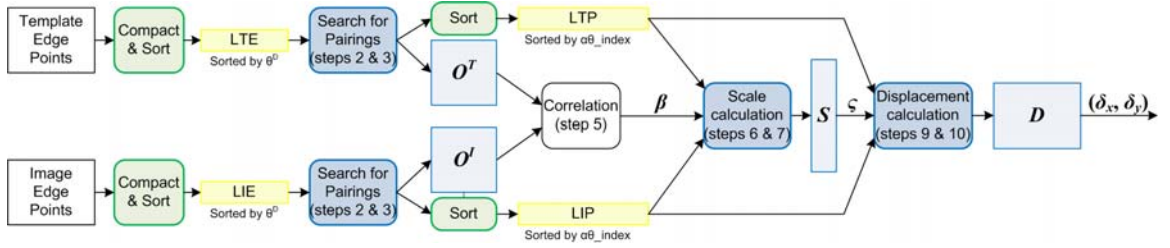


Fig. 2. Our implementation of the GHT: Stages in blue compute the  $\mathcal{O}$ ,  $\mathcal{S}$  and  $\mathcal{D}$  Hough spaces; stages in green compact and/or sort the workloads of the kernels.

balance, coalesced memory accesses and, consequently, good performance values. Edge detection is performed using the widely known Canny algorithm [2]. The version of the Canny edge detector, used in this work, was developed by Gómez-Luna et al. [4].

Correlation between  $\mathcal{O}^T$  and  $\mathcal{O}^I$  is implemented using a separable convolution [12]. Correlation is applied once per each possible rotation angle of the template within the image. Finally, the function  $\text{max}_i(M)$ , used in step 5 to obtain  $\beta$ , is performed by a kernel which follows the optimizing strategies presented in the CUDA parallel reduction [7].

### C. Irregular computation within the GHT

Computation of  $\mathcal{O}$  (steps 2-3),  $\mathcal{S}$  (steps 6-7) and  $\mathcal{D}$  (steps 9-10) Hough spaces is not regular. All these stages have some common features as far as memory accesses and work distribution are concerned:

- Computation of the three Hough spaces requires some kind of features comparison, followed by some computation, among the elements of the corresponding input workload, as stated in steps 2, 7 and 10.
- The features comparisons in steps 2, 7 and 10 need a huge number of memory accesses, which seriously penalizes the performance. By sorting the input workload, the number of memory accesses will be greatly reduced.
- $\mathcal{O}$ ,  $\mathcal{S}$  and  $\mathcal{D}$  Hough spaces are generated during a voting process. This entails the use of atomic additions—in shared or global memory, depending on the size of the voting space—, which serialize the execution. In Section V, we propose the replication of the Hough spaces, in order to decrease the impact of serialization.

Figure 2 shows a scheme of our implementation of the GHT, which is thoroughly described in Sections IV and V. Stages in blue correspond to kernels that perform the computation of the  $\mathcal{O}$  Hough spaces (Search for pairings), the  $\mathcal{S}$  Hough space (Scale calculation) and the  $\mathcal{D}$  Hough space (Displacement calculation). Stages in green represent the primitives applied for regularizing the problem by compacting and/or sorting the workloads of the kernels.

## IV. EFFICIENT MEMORY ACCESS AND WORK DISTRIBUTION

In this section, we detail how to re-organize the workloads, in order to obtain an efficient computation of the Hough spaces on the GPU. Then, we propose two

mechanisms for efficiently distributing the workloads among blocks and threads.

### A. Re-organizing the workload: compacting and sorting

An efficient implementation of the search for pairings requires the compaction of the whole set of contour points into a dense list. Thus, for each contour point in the template or the image, a tuple  $p_i$  composed of its gradient direction  $\theta_i$  and its coordinates  $(x_i, y_i)$  is stored into a *List of Template Edges* (LTE) or a *List of Image Edges* (LIE). The compact primitive returns a List of Edges composed by three output arrays: one for the gradient directions and two for the coordinates. The gradient directions are used to detect pairings and, together with the coordinates, are needed for computing the angle  $\alpha_{ij}$ . The CUDPP library [3] includes a compact primitive based on the prefix sum or scan operation, that we have used.

As it is shown in Figure 2, the List of Edges is the workload of the kernel that performs the search for pairings. It outputs a *List of Template or Image Pairings* (LTP or LIP), whose elements are tuples  $p_{ij}$ , and a template or image  $\mathcal{O}$  Hough space ( $\mathcal{O}^T$  or  $\mathcal{O}^I$ ). LTP and LIP are dense lists used as inputs for the scale and the displacement calculations. Due to implementation convenience, tuples  $p_{ij}$  in a List of Pairings contain the index of the pairing in the corresponding  $\mathcal{O}$  Hough space ( $\alpha\theta\_index = \alpha_{ij} \times 90 + \theta_i$ ), the index of each contour point ( $p_k\_index = y_k \times width + x_k$ , where *width* is the width of the image) and the distance between these paired contour points ( $d_{ij}$ ).

At this point, we propose a previous sorting of the dense lists, in order to minimize global memory accesses. In the search for pairings, the List of Edges can be sorted by the quantized gradient direction. Then, given a certain value of the quantized gradient direction, this value plus the difference angle ( $\xi$ ) determines the part of the List of Edges where the pairing points lie. In the scale and the displacement calculations, the Lists of Pairings are sorted by the  $\alpha\theta\_index$ , that is, pairings are grouped in sub-lists with the same  $\alpha$  and  $\theta$  values.

### B. Work distribution among blocks and threads

In this sub-section we present two mechanisms for working with the created sorted lists. Both can be applied to the search for pairings and to the scale and displacement calculations.

As it is seen in Figure 2, computing stages (in blue) which generate the Hough spaces use sorted dense lists

as inputs. We assume that two lists (*List1* and *List2*) are the inputs to the computing stages. Specifically, LTP and LIP are *List1* and *List2* for scale and displacement calculations and in the case of the search for pairings, a Template or Image List of Edges takes the role of both *List1* and *List2*.

We consider a kernel, whose inputs are two dense lists (*List1* and *List2*), which have been sorted by an index  $I$ . *List1* and *List2* are divided into sub-lists, in which every tuple has the same index. Each list has its own constant array associated (*Pointers* array), in which the  $k^{\text{th}}$  element contains the position of the list where the sub-list with index  $I$  equal to  $k$  starts. Pointers arrays are placed in constant memory or texture memory, depending on their size, since they are read-only data.

Each block takes one chunk of *List1*, belonging to a sub-list with a certain index  $I_1$ , and loads it in shared memory. Each thread loads just one tuple of the chunk, thus the size of the chunk is at most the number of threads in a block. Then, the block performs an iterative process with an outer and an inner loop. The outer loop accesses those chunks of *List2*, which belong to the sub-list with an index  $I_2$  that fulfills a certain condition with respect to the index  $I_1$ . The inner loop distributes the work among the threads, which perform some computation using one tuple from *List1* and another from *List2*. The following code summarizes this process:

```
Load chunk of List1 sub-list with I1 in shared memory;
I2 = function of I1;
// Outer loop:
For (each chunk of sub-list with I2 of List2)
  Load chunk in shared memory or registers;
  // Inner loop:
  For (depending on mechanism)
    // Compare and compute:
    If (Features comparison)
      Computation(tuple List1,tuple List2);
```

Work distribution within the inner loop can be done in two ways that are explained next: the first one achieves an optimal load balancing, while the second one focuses on increasing occupancy of the multi-processors.

#### 1) Load-balancing (LB) mechanism

A load-balancing work distribution must ensure that every thread will perform the same number of features comparisons or, in other words, the same number of iterations of the inner loop. The red chunk in Figure 3 contains  $n$  tuples and the blue chunk contains  $m$ . Values  $n$  and  $m$  are less or equal to the number of threads in the block (*block\_size*). Thus, the number of comparisons is  $n \times m$ . Thread  $N$  performs the  $N^{\text{th}}$  comparison, the  $N^{\text{th}}$  + *block\_size*, and so on. This mechanism requires that every tuple of the chunk of *List2* is available for every thread. Thus, the chunk of *List2* is loaded in shared memory.

#### 2) Save-shared-memory (SSM) mechanism

Although the former mechanism ensures an optimal load balancing, it requires loading two chunks in shared memory. Unfortunately, the occupancy is determined by the amount of shared memory and registers used by each thread block, thus load balancing can affect negatively the efficiency. We propose a new mechanism, *SSM*, which saves shared memory to increase occupancy.

This mechanism, as it can be seen in Figure 4, assigns one tuple of the chunk of *List2* to one thread, so that each thread loads only its tuple in registers. Then, the thread performs the comparisons between its tuple and

all the tuples of the chunk of *List1*. Since usually the number of tuples with the same index  $I$  is not a multiple of the block size, there will be idle threads in the inner loop. Nevertheless, we expect a good performance due to the increase of occupancy.

## V. REPLICATION OF THE VOTING SPACE

Implementations of the three stages have also in common the fact that they perform a voting process, which is the generation of some kind of histogram. Since voting operations entail unpredictable memory accesses, efficient implementations use several copies or sub-histograms, in order to reduce conflicts among threads, while updating the histogram bins.

The CUDA SDK implementations of 64- and 256-bins histograms [11] use, respectively, per-thread and per-warp sub-histograms, which are lied in shared memory. At the end of the process, all the sub-histograms are merged into a single histogram in global memory. However, the use of per-thread sub-histograms is limited to those cases in which the number of bins of the histogram is very small. On the other hand, the drawback of per-warp sub-histograms, with respect to per-thread, is the use of atomic additions in shared memory. Since every thread, belonging to a half-warp, tries to access the same sub-histogram at the same time, serialization is unavoidable.

For both reasons, we propose the use of replicated sub-histograms per block in shared memory. Threads of each block access more than one sub-histogram. If  $M$  sub-histograms per block are declared, thread number  $N$  accesses sub-histogram  $N \% M$ , where  $\%$  stands for the operation modulo. This represents an improvement with respect to per-warp sub-histograms, since consecutive threads, belonging to the same warp, access different sub-histograms, reducing serialization due to atomic additions. There will be an optimal value of  $M$ , which represents a trade-off between reducing serialization, when using atomic additions, and preserving a good value of occupancy.

Replication is also useful in global memory, in order to reduce serialization while using atomic functions. In the case of the displacement calculation, since the  $\mathcal{D}$  Hough space does not fit in shared memory, the whole voting space is replicated in global memory.

## VI. EXPERIMENTAL RESULTS

In this section, a thorough analysis of the improvements achieved by the proposed techniques is carried out. In addition, the impact of these improvements in the final performance of the GHT is evaluated. Thus, we have analyzed the impact of the irregular stages in the total execution times as they are the most time-consuming ones in the GHT. In fact, computation of  $\mathcal{O}$ ,  $\mathcal{S}$  and  $\mathcal{D}$  Hough spaces require more than 90% of the execution time while Canny detection, rotation calculation, compacting and sorting have negligible execution times. Tests have been made on a GeForce GTX 280 GPU.

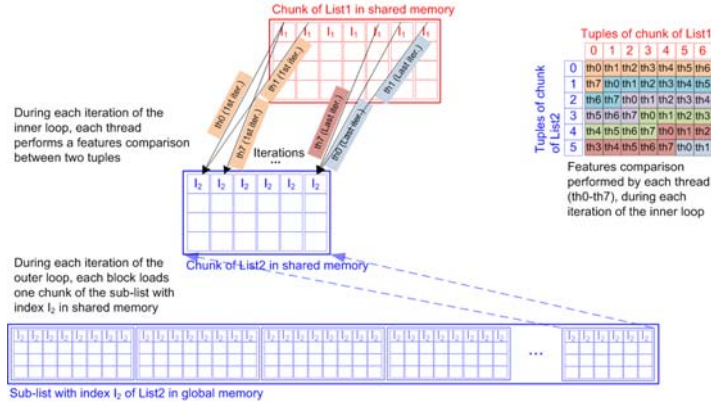


Fig. 3. Load-balancing mechanism: Each thread performs approximately the same number of features comparisons, represented by black arrows. For the sake of clarity, blocks of 8 threads are represented.

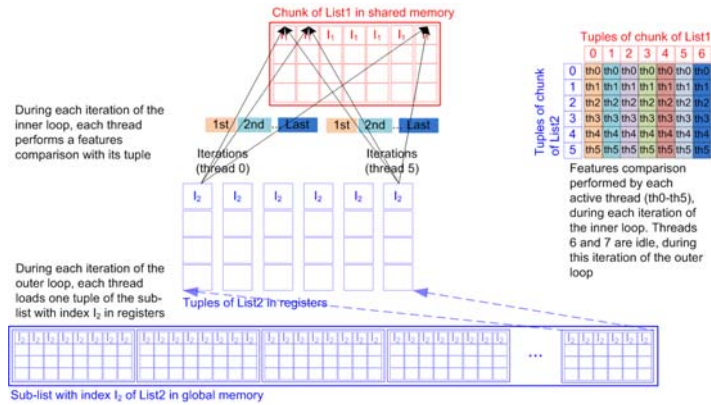


Fig. 4. Save-shared-memory mechanism: Each thread performs the features comparisons of one tuple, as indicated by the black arrows.

Although the GHT was originally designed to detect arbitrary shapes in two-dimensional images, it can be easily applied to video processing [13]. We have selected this real application for the experiments because videos provide an assorted database of images to test our improvements, especially when the chosen videos belong to different genres. In Table I the workloads of the four videos used in the experiments are shown. These videos have been selected from the MPEG-7 Content Set.

TABLE I  
TEST WORKLOADS CHARACTERISTICS. VIDEOS HAVE A RESOLUTION OF  $352 \times 288$  PIXELS. NUMBER OF EDGE POINTS AND PAIRINGS ARE AVERAGE VALUES. EACH VIDEO IS CONSISTING IN 4000 FRAMES

Video	Edge points	Pairings ( $\xi = 90^\circ$ )
Cycling	2778	78770
Movie	1436	13332
Basket	5061	140030
Drama	2684	54921

#### A. An exhaustive comparison between the load-balancing and the save-shared-memory mechanisms

We are not able to assert which of the mechanisms is better, since both have their own strong points. For this reason, we have compared both mechanisms changing the size and data distribution of a sorted list. Without loss of generality, we have used a synthetic sorted list, equally divided among sub-lists with different index values. Each element of the synthetic sorted list emulates a tuple. Since each block works with chunks belonging to a sub-list, we have changed the number of

tuples per sub-list, so that the number of chunks in a sub-list changes between 1 and 6.

In the case of SSM, the saving of shared memory permits 5 blocks of 128 threads per multi-processor, one more than LB. On the other hand, LB guarantees an optimal load balancing, while SSM will have idle threads in the last block assigned to a sub-list. Using blocks of 128 threads, if each sub-list contains  $T$  tuples, this last block have only  $T\%128$  active threads.

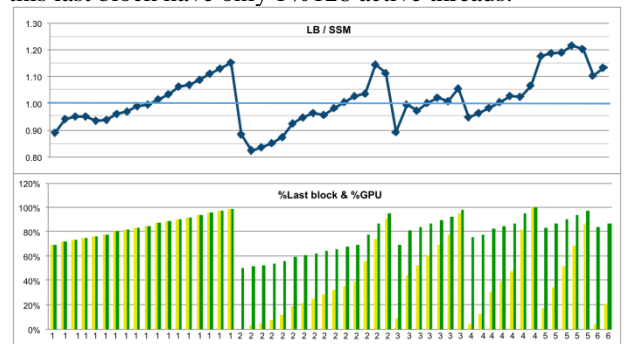


Fig. 5. Comparison between SSM and LB using a synthetic sorted list. The number of blocks assigned to a sub-list has been changed from 1 to 6, as the abscissas shows

We have carried out 55 tests of the SSM and LB mechanisms, changing the number of tuples of the sub-lists. Figure 5 presents the execution results for these tests. Abscissas represent the execution results for these tests. Abscissas represent the number of 128-tuples chunks per sub-list, which is also the number of blocks working with the same sub-list. The graph on the top shows the ratio between the execution times of LB and SSM. Values above 1 mean the SSM mechanism runs

TABLE II.

AVERAGE EXECUTION TIMES (MS) OF THE MAIN PARTS OF THE APPLICATION FOR FOUR VIDEOS. THE NUMBER BESIDE THE NAME OF THE STRATEGY REPRESENTS THE REPLICATION FACTOR.

Video	Canny	Search for pairings		Orientation	Scale		Displacement	
		LB(1)	SSM(2)		LB(8)	SSM(16)	LB(64)	SSM(64)
Cycling	0.45	1.78	1.66	5.68	86.53	53.10	265.02	210.09
Movie	0.46	0.28	0.63	5.66	20.13	18.37	29.93	22.29
Basket	0.46	1.13	1.36	5.68	102.38	83.83	198.09	152.28
Drama	0.46	0.67	0.99	5.66	49.31	42.00	85.17	60.07

faster. The graph on the bottom shows two columns for each test. The left column (yellow), called “%Last block”, represents the percentage of active threads in the last block assigned to a sub-list, in SSM. The right column (green), called “%GPU”, stands for the percentage of active threads in the whole GPU, in SSM. The higher these values the better is the distribution of the workload in SSM. Thus, both columns give a hint of the computational load balance of SSM.

For a number of blocks per sub-list between 1 and 4, there exists a value of “%Last block” which determines that the SSM mechanism outperforms the LB one because the impact of load unbalance is less important than the occupancy value. When the number of blocks per sub-list is 5 or more, the SSM mechanism always performs better due to the higher occupancy, which permits to execute more blocks simultaneously.

### B. Replication for the generation of Hough spaces

In the search for pairings, the case with 2 replicated sub-histograms results in 13% improvement with respect to the per-warp approach.

The size of the  $\mathcal{S}$  Hough space is not critical for occupancy. Such a small size permits to attempt even sub-histograms per thread. The best case is a per block replication of 16. It works 28% faster than the per-warp approach and 109% faster than the per-thread approach.

Displacement calculation replicates the  $\mathcal{D}$  Hough space in global memory. The best approach is with replication of 64, obtaining a speedup of 3.2 with respect to the version without replication.

### C. Comparison among implementations

The execution times of the different implementations of the main parts of the application are shown in Table II. Results reflect that the search for pairings performs better using the LB mechanism in three of the four videos. This makes sense with the conclusions presented in sub-section VI.A because the size of the sub-lists is small. If the number of tuples increases, the percentage of idle threads decreases for SSM. In this way, its load balancing improves and the occupancy becomes more decisive. This explains that SSM outperforms LB for scale and displacement calculations, since the Lists of Pairings are much longer than the Lists of Edges.

## VII. CONCLUSIONS

This work has studied how load balancing and occupancy impact the performance of an application on a GPU using the CUDA environment. This analysis has been carried out through the parallelization of the Fast Generalized Hough Transform (Fast GHT).

We have implemented and compared two general parallelization strategies. One, called Load Balancing (LB), tries to obtain an optimum load balancing. Another one, called Saved Shared Memory (SSM), tries to maximize processor occupancy by reducing data loaded in shared memory. Results show the SSM mechanism outperforms the LB one when there is enough input data for every simultaneous block in the GPU. We have also minimized the impact of divergent execution paths by compacting and sorting the input data.

Memory accesses conflicts have been addressed in two ways depending on if they were read or write accesses. Read accesses to global memory have been minimized by sorting input data. Unpredictable write accesses can seriously reduce the performance due to memory bank conflicts and serialization, but by replicating the write area these conflicts are minimized. We have proposed a new technique that replicates in shared memory data per block and compared it with the common strategy of replicating per warp, obtaining a better performance. Voting spaces are also replicated in global memory with successful results.

## REFERENCES

- [1] Ballard, D.H. (1981). Generalizing the Hough transform to detect arbitrary shapes. *Pattern Recognition* 13 (2): 111-122.
- [2] Canny, J. (1986). A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 8 (6): 679-698.
- [3] CUDPP (2010). CUDA Data Parallel Primitives Library home page. <http://code.google.com/p/cudpp>
- [4] Gómez-Luna, J., González-Linares, J.M., Benavides, J.I. and Guil, N. (2009). Parallelization of a video segmentation algorithm on CUDA-enabled Graphics Processing Units. In proceedings of 15th International Euro-Par Conference (Euro-Par'09), pp. 924-935.
- [5] Green, S. (2007). CUDA particles. [http://developer.nvidia.com/object/cuda\\_sdk\\_samples.html](http://developer.nvidia.com/object/cuda_sdk_samples.html)
- [6] Guil, N., González-Linares, J.M. and Zapata, E.L. (1999). Bidimensional shape detection using an invariant approach. *Pattern Recognition* 32 (6): 1025-1038.
- [7] Harris, M. (2007). Optimizing parallel reduction in CUDA. [http://developer.nvidia.com/object/cuda\\_sdk\\_samples.html](http://developer.nvidia.com/object/cuda_sdk_samples.html)
- [8] Hough, P.V.C. (1962). Method and means for recognizing complex patterns. U.S. Patent 3069654.
- [9] NVIDIA (2007). NVIDIA CUDA home page. <http://www.nvidia.com/cuda>
- [10] OpenCL (2009). OpenCL home page. <http://www.khronos.org/opencl>
- [11] Podlozhnyuk, V. (2007a). Histogram calculation in CUDA. [http://developer.nvidia.com/object/cuda\\_sdk\\_samples.html](http://developer.nvidia.com/object/cuda_sdk_samples.html)
- [12] Podlozhnyuk, V. (2007b). Image convolution with CUDA. [http://developer.nvidia.com/object/cuda\\_sdk\\_samples.html](http://developer.nvidia.com/object/cuda_sdk_samples.html)
- [13] Sáez, E., González-Linares, J.M., Palomares, J.M., Benavides, J.I. and Guil, N. (2003a). New edge-based feature extraction algorithm for video segmentation. In proceedings of Image and Video Communications and Processing (SPIE'03), pp. 861-872.