GPGPU Implementation of a Genetic Algorithm for Stereo Refinement

Álvaro Arranz, Manuel Alvar

Zed Worldwide

***Abstract* — During the last decade, the general-purpose computing on graphics processing units Graphics (GPGPU) has turned out to be a useful tool for speeding up many scientific calculations. Computer vision is known to be one of the fields with more penetration of these new techniques. This paper explores the advantages of using GPGPU implementation to speedup a genetic algorithm used for stereo refinement. The main contribution of this paper is analyzing which genetic operators take advantage of a parallel approach and the description of an efficient state- of-the-art implementation for each one. As a result, speed-ups close to x80 can be achieved, demonstrating to be the only way of achieving close to real-time performance.**

***Keywords* — Parallel processing, GPGPU, genetic algorithm, stereo.**

I. Introduction

RECENTLY, custom GPU programming has become one of the most popular tools for increasing the efficiency of parallel algorithms thanks to the computational capacity of the Graphics Processing Unit (GPU) compared to serial CPU programs.

Traditionally, GPUs appeared in the computer market as hardware products specialized on rendering tasks and, more specifically, for improving the gaming experience. Given that most of the rendering pipeline's steps of were parallel, these products rapidly evolved to machines capable of efficiently running highly parallel algorithms. In the last decade, the flexibilization of the GPU hardware and tools has enabled the use of these parallel-processing units for general scientific purposes.

Stereo analysis is a Computer Vision research area that has been widely studied in the literature. However, it remains an unsolved problem and many algorithms are still proposed every year. The aim of the stereo analysis is to obtain depth information from a couple of stereo images, simulating how the human's can perceive the depth using just two eyes. Solving this problem is very computationally demanding, especially when dealing with high-resolution images. GPGPU techniques have been recently used for speeding up these tasks and great results have been reported in the literature.

GPGPU primarily aims to improve the program's performance. It has been demonstrated that using these techniques could result in a speed-up of up to x100, depending on the algorithms' nature. This paper proposes to study the speed-up achieved by GPGPU programming applied to an evolutionary algorithm. A genetic algorithm for stereo refinement is implemented in both CPU and GPU and its performance analyzed and compared.

Improving the accuracy and performance of stereo algorithms is crucial for many real applications. Robotics has been traditionally a research area that has used these techniques, but new fields are arising. The digitalization of the automotive sector is leading to the incorporation of new sensors such as high definition cameras to high-end cars. Fast stereo algorithms are needed to provide accurate information about the car's environment. Other applications of stereo algorithms are biomedicine, virtual reality, automation or the entertainment industry. However, note that any optimization problem solved with evolutionary algorithms might benefit from the work herein proposed.

The paper is structured as follows: Section II is a brief overview of the GPGPU implementations found in the literature, Section III explains the stereo refinement genetic algorithm implemented, Section IV describes the details about the GPGPU implementation, in Section V some results are presented and finally in Section VI some conclusions are drawn.

II. GPGPU Overview

GPGPU has been widely used in the literature by the computer vision community. Its main role has been to enable real-time performance on many demanding algorithms.

First works on stereo GPU processing were proposed in [11]. SSD dissimilarity techniques, a multi-resolution approach and a very primitive GeForce4 were used to obtain performance equivalent to the fastest CPU commercial implementations available. Later, [3] proposed a multi-view plane- sweep-based stereo algorithm for handling correctly slanted surfaces applied to urban environments. Assuming a highly structured scene with buildings, they used a planar prior for estimating disparity maps. The algorithm was successfully implemented in an Nvidia GPU obtaining real-time frame rates.

DOI: 10.9781/ijimai.2015.329

In [6] a high-performance stereo-matching algorithm both fast and accurate is proposed. Using a parallel designed AD-census and scanline optimization implemented in CUDA in an NVIDIA GeForce GTX 480 they achieve near to real- time frame-rates. They report an impressive 140x speed up compared to the CPU implementation. Another GPU stereo matching algorithm using adaptive windows can be found in [13].

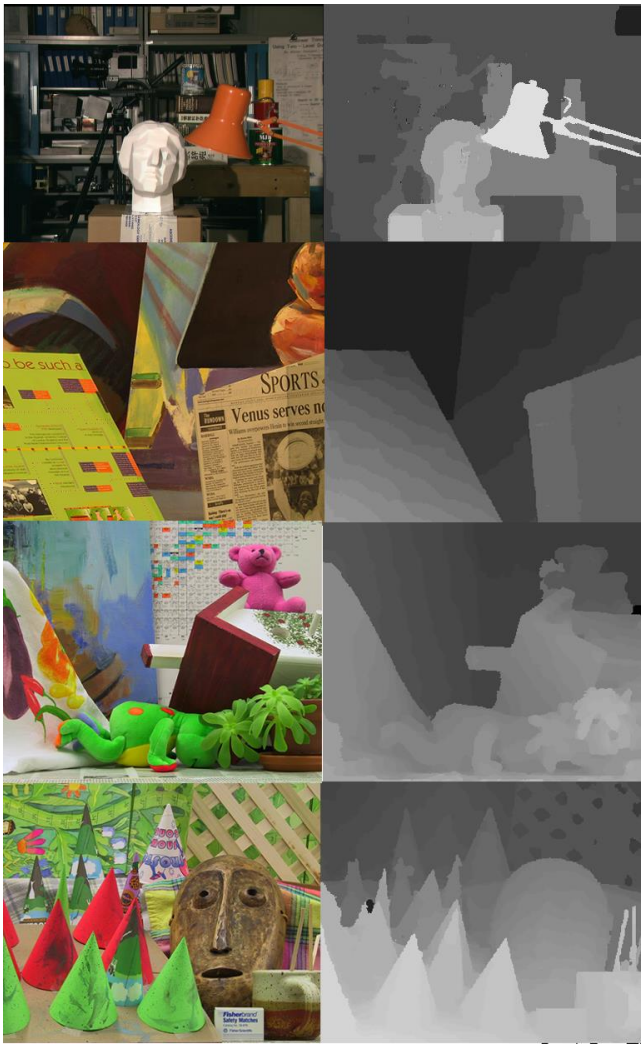

Fig. 1 Disparity map examples

In [4] and [7] a real-time camera tracking and mapping using RGB-D cameras is proposed, obtaining quite impressive results. Their implementation relays heavily on the use of GPGPU, both for tracking and TSDF mapping. Depending on the voxel's resolution, they achieve execution times from 10 to 25ms.

GPGPU has also been applied in other fields, such as in feature detection and tracking, as proposed in [9]. Their KLT GPU implementation achieves real-time 30Hz on 1024768 resolution images, which is a 20x speed-up compared to their CPU implementation. A 10x improve is also reported for the SIFT [5] detector implemented in GPU. A CUDA implementation of the famous graph cuts algorithm [2, 14, 15] is presented in [10], obtaining a 12x performance enhancement.

A similar system to the one herein proposed is presented in [8]. In this work, a genetic algorithm for stereo matching is also implemented in GPU. However, the genetic algorithms have quite different approaches, and their parallel implementation does not seam to provide any performance boost compared to the CPU one. This paper shows that, with the proper GPU implementation, a 50x speed-up can be achieved.

III. Genetic Algorithm for Stereo Refinement

The genetic algorithm for stereo refinement implemented in this paper is based on the work proposed in [1]. The implementation minimizes a fitness function that is related to a Markov Random Field (MRF) and is equivalent to minimizing a global energy function. Due to the flexibility of genetic algorithms, this function is able to include occlusion handling.

This algorithm uses a guided search approach with new crossover and mutation operators adapted to the stereo refinement problem. Each operator will be explained briefly in this section. An example of the results that can be achieved using these techniques is shown in Figure 1.

A. Genome representation

Each individual includes the whole disparity map estimate and the occlusion map for both left and right images.

$$\bar{g}\begin{cases}\bar{f}_L=\{f_{L_1},f_{L_2},\ldots,f_{L_N}\}\\ \bar{f}_R=\{f_{R_1},f_{R_2},\ldots,f_{R_N}\}\\ \bar{O}_L\\ \bar{O}_R\end{cases},\quad f_i\in\Lambda,\Lambda=\{1,2,\ldots,L\}\qquad(1)$$

where g is the genome, gL and gR are the representation of the left and right disparity images respectively, XiL and XiR are the disparities estimated for pixel i on the left and right disparity images, N the total number of pixels in each image and Li the set of different disparity labels.

Occlusion maps are defined as:

$$O(p)=\begin{cases}0 & \textit{if not occluded}\\ 1 & \textit{if occluded}\end{cases}\qquad(2)$$

where O(p) is the occlusion map and p is the pixel.

B. Initialization

For the initialization process two different window-based algorithms with different window sizes, the adaptive support-weight approach [12] with random parameters and the census based with window-cost aggregation have been used. This variation aims to provide a wide range of initial solutions.

$$E(\bar{g}) = E_{data}(\bar{g_L}) + E_{smooth}(\bar{g_L}) \quad (3)$$

$$E_{data}(\bar{g_L}) = \begin{cases} \lambda_d & \text{if } i \text{ is occluded} \\ \sum_{i \in \bar{g_L}} |I_L(x_i, y_i) - I_R(x_i - X_i, y_i))| & \text{otherwise} \end{cases} \quad (4)$$

$$E_{smooth}(\bar{g_L}) = \sum_{\{p,q\} \in N} min\left(\frac{\beta_s}{\varphi_s}|X_p - X_q|, \lambda_{st}\right) \quad (5)$$

$$\beta_s = max(\lambda_s, \gamma_s - |I_L(p) - I_L(q)|) \quad (6)$$

C. Fitness function

An energy function that considers discontinuities and occlusions is used for the fitness function:

where g is a certain individual, gL is the left disparity image, Il and Ir stand for the left and right stereo pair, xi and yi are the image coordinates of pixel i, V{p,q} is a smoothing function and λs, γs and φs are constant parameters for every pixel.

Before any fitness function evaluation, a occlusion management process is triggered for classifying pixels correctly before any energy evaluation.

D. Occlusion management

The process of handling the occluded areas is a two-step operation: occlusion detection followed by an occlusion management.

The following operations are defined for calculating the left occlusion map:

$$O_L(p) = \begin{cases} 0 & \exists i / \begin{pmatrix} x(i) + \bar{g_R}(i) \\ y(i) \end{pmatrix} = \begin{pmatrix} x(p) \\ y(p) \end{pmatrix} \quad p, i \in P \\ 1 & \text{otherwise} \end{cases} \quad (7)$$

being OL the left occlusion map, x(p) and y(p) the x and y coordinates of point p respectively and P the set of disparity image points. A similar expression can be deduced for the right occlusion map. This occlusion map identifies which areas of the image are classified as occluded regions.

For the occlusion management, an iterative process based on neighboring disparities of the occluded pixels is applied. For the left image, each occluded pixel is assigned the disparity value of the most photo-consistent non-occluded neighbor from left to right and afterwards it is marked as non-occluded. If no non-occluded neighbors exist, it maintains its occluded status for the next iteration. Special status have the occluded pixels whose x(p) coordinate is less than the number of disparities analyzed. In this case the iteration is made from right to left and bottom-up. The iteration is finished when no occluded pixels are left on the left occluded map.

For the right image it is similarly done but vice versa (right to left for common pixels and left to right for pixels whose x(p) is at a distance of the number of disparities analyzed from the right image border).

E. Crossover

The crossover is based on comparing parent's blocks of different sizes and assign the best ones to the same son. This operator can be summarized in the following steps:

1) Parents are divided into blocks (random sizes)
2) The fitness function of each block is evaluated
3) Best block is selected to persist in the same child

F. Mutation

Three different mutation operations may occur to each individual. Firstly, one possible mutation operation is to initialize again a group of pixels following the steps explained in Subsection III-B with a probability PMa. Secondly, a bilateral filter operation with a random window size with a probability PMb. Finally, a morphological operation such as erode or dilate may occur with a probability PMc.

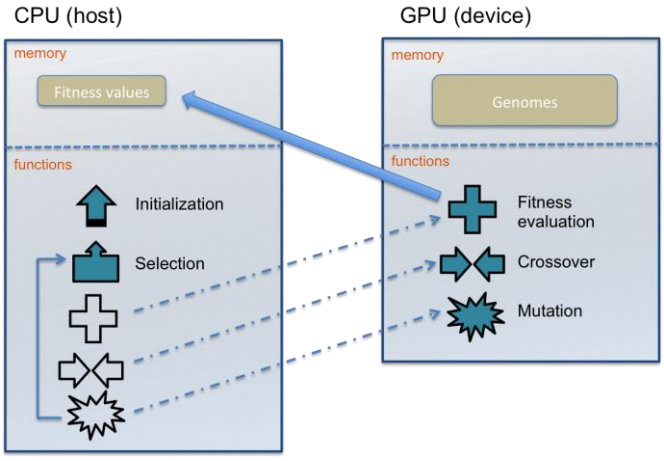

Fig. 2: Assignment of genetic operators to GPU and CPU

IV. Algorithm's GPGPU Implementation

After analyzing the performance of the serial version of the genetic algorithm, it is easy to conclude that the most computationally demanding functions are the genetic operators and not the genetic algorithm itself. This result is straightforward because each genome includes a lot of data and information inside (whole four images: two disparity maps and two occlusion maps). For example, each genome evaluation implies evaluating the energy function for each pixel and neighborhood individually. Besides, each genome operator is naturally parallel, which suggests that implementing these operators in CUDA will have a dramatic impact on the genetic algorithm performance.

In Figure 2 is shown where is computed each genetic operation. The left side of Figure 2 represents data information is stored and which functions are implemented and executed in the CPU. The fitness values are stored in the CPU because they are needed for the selection operator in order to decide which individuals of the actual population will survive to the next one. The right side of the diagram represents which information is stored and which functions are evaluated in the

GPU. All the genomes are stored in the GPU in order to enable fast access to the data from the functions evaluated in the device. The only memory transaction between the CPU and GPU needed is the copy of the fitness value of each individual from device to host and is represented by the big blue arrow from the fitness evaluation function icon to the fitness value memory in the CPU. Remember that this device-host and vice-versa transactions are very costly and must be minimized for achieving the best performance.

The genetic algorithm has been implemented in the CPU using the GAlib library. For the image processing and allocation it has been used the OpenCV library, specifically the GPU module, which facilitates the memory allocation and transaction and has quite a lot processing algorithms built-in in the GPU already. Finally, evaluation, crossover and mutation operators have been implemented in CUDA in several kernels. The next sections describe in detail the strategy used for implementing efficiently each operator in CUDA language.

1) CUDA evaluation kernel

Although the title may suggest that the evaluation of a genome is carried out just by one kernel, the reality is that it is a process composed by three steps. The first two are solved using a single kernel each while the third has to be solved by two kernels. The first two steps could be executed in parallel by two different CUDA streams but the last have to be executed after the firsts have finished. This parallel capability has not been implemented and all four kernels have been

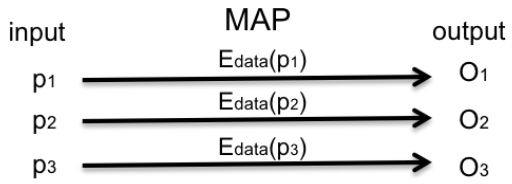

Fig. 3: Parallel MAP operation

programmed to run in the same stream.

The first step in the evaluation process is the data term evaluation of the energy function. The result is one value for each pixel and its calculation is independent from the values of the neighboring pixels. Thus, the relation is one to one and its parallel implementation is very efficient and straightforward. This type of operation is also called MAP, and it has been implemented using one thread per pixel in the disparity image. A simple diagram of MAP is shown in Figure 3.

The data term only depends on the values of the left and right stereo image and on the disparity image evaluated. Left and right stereo images have been allocated in the device as 2D textures, which are very efficient for interpolation. Note that in this case, using shared memory does not make much sense because the number of memory accesses needed per thread would not bee minimized. The result is saved in a floating-point structure of the same size as the original image, and here will be referred as memData.

The second step is the evaluation of the smoothing term. If one thread per pixel is used, it requires to access to its own disparity value and the neighbouring disparities. This operation can be considered a type of Stencil operation, in which many reads are needed as input while only one write is performed. An illustrative example is shown in Figure 4.

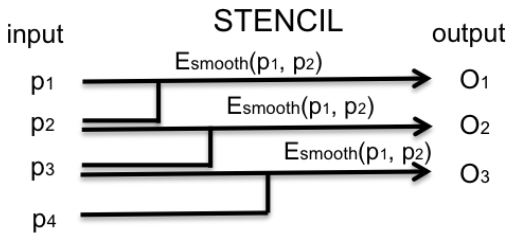

Fig. 4: Parallel STENCIL operation

In order to maximize the performance, shared memory is used for first loading all the disparities in a block and then using that shared memory in all the threads of the same block. Remember that the access to shared memory is much faster than the access to global memory. Each thread is related to each pixel in the disparity image and it is in charge of evaluating the smoothing function that relates itself with the right and bottom neighbors. As happened in the data kernel, the result is saved into a floating point vector with the same size of the disparity image and here will be called memSmooth.

The third and last step is composed of two kernels, one executed after the other. It is in charge of performing the summation over all pixels of the memData and memSmooth structures calculated in the two first steps. This type of summation is an operation also known as Reduce. Although at first glance this operation might seem difficult to parallelize, actually it is fairly simple. Figure 5 shows the two-step reduction implemented. Besides, this step sums memData and memSmooth individually for each pixel and saving it in memTotal in order to facilitate the crossover task explained in subsection IV-2.

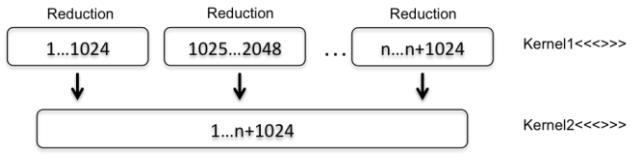

Fig. 5: Reduction operation in two kernels

For enhancing the performance of the Reduce operation, the data has been divided in groups of 1024 addends, each being processed by a CUDA block. All the data in each group is loaded in shared memory to improve its read and write speed. The first kernel performs the summation over each group, obtaining one result per group. Finally, the last kernel performs the last summation over all the results of the previous kernel, and obtains the final value for the fitness function.

Finally, an asynchronous memory copy is performed from device to host to copy the final fitness value calculated for that genome. This operation is recommended to be asynchronous because the memory copy can be performed at the same time

as other kernels are executed in other streams, instead of waiting until the memory copy has finished.

Note that in spite of running all the evaluation kernels in the same stream, different individuals are able to run their evaluation in different streams, which enables copying memory from device to host at the same time as other kernels and a higher level of parallel exploitation.

The evaluation process is performed over the left and right disparity map, but for the right one, it is not necessary to perform the final Reduction and memory copy. This optimization can be achieved because de fitness function of a genome is just the fitness function of its left disparity image.

2) CUDA crossover kernel

As explained in Subsection III-E, the crossover operation consists of three steps:

• Divide the two disparity maps into blocks. In this case, the number of pixel of each block will not be greater than 1024.

• Sum up the memTotal for each pixel inside the blocks, compare them by pairs (one for each parent) and keep the block with the best fitness function.

• Copy the best block to the children.

The limit of 1024 pixels per block is related to the maximum number of threads per CUDA block available by the GPU. All the pixels in a block must be part of the same CUDA block be- cause the summation can be performed using shared memory, which is much more efficient than global memory. Therefore, the CPU realizes the first step, and the following two are done by the GPU, the first one as a Reduction operation very similar to the one in subsection IV-1 and the second one as a very simple copy operation as a MAP.

3) CUDA occlusion handling kernel

Occlusion handling encompasses two different tasks: occlusion estimation and occlusion management. The occlusion estimation is calculated through an image warping, where each pixel of the other disparity image is displaced a number of pixels equal to its disparity level. Pixels left without any assignation are considered to be occluded pixels. Thus, each pixel operation is independent from the rest, but several threads can output their result to the same piece of memory. This operation is also known as Scatter and can be solved using, for example, atomic operations. In our case it is not necessary because the function aims only to output a boolean value, more precisely a zero to indicate that the pixel is not occluded.

The second task is the occlusion management, where the main objective is to re-estimate the disparity value for the pixels that where labelled as occluded. For this parallel implementation the horizontal fast occlusion filling algorithm explained in Subsection III-D was used. Given that a horizontal search for the closest non-occluded pixel has to be performed, the occlusion information was loaded in shared memory, being each block responsible for each independent scan-line. Each thread is in charge of estimating the new depth for each occluded pixel. Figure 8 shows the per-thread operations and the memory accesses incurred.

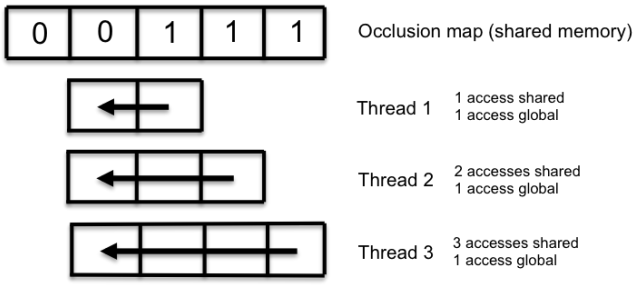

Fig. 6: Horizontal fast occlusion filling implementation example

4) CUDA mutation kernel

The mutation kernel comprises three different operations: bilateral filtering, erosion and dilation. These morphological operations are already efficiently implemented in the OpenCV library using CUDA. A problem that may rise using a third party library is the performance penalty incurred while parsing from the data-types used in your application to the data-types used in the library and vice- versa. However, in the implementation herein proposed, the data types are compatible with those from OpenCV, so this transformation is trivial. Thus, this library has been used for this purpose.

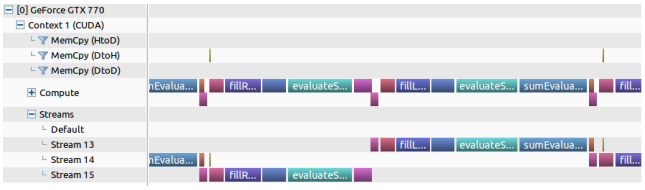

Fig. 7: Nvidia Visual Profiler tool

V. Results

In this section the parallel capabilities of the genetic algorithm are discussed. Both the serial implementation and the parallel one using CUDA are compared. Given the stochastic nature of the algorithm and the various types of mutations that are likely to happen, the algorithm was run for different images during five hundred generations and an average per individual and generations was calculated. The Middlebury dataset will be used for comparison, as it is a standard and well-known test-bed.

For the tests, an Intel i7-2600 at 3.4 GHz CPU and an Nvidia GeForce GTX 770 were used. As operating system, Ubuntu Linux 14.04LTS was used given the CUDA performance improvement compared to Windows. The measuring tool used was the Nvidia Visual Profiler, obtaining valuable data such as timing, occupancy, optimizations, et. A capture of the profiler is shown in Figure 7.

The parameters used in the experiments carried out along this section are shown in Table I and Table II.

A comparison between the performances of the GPGPU versus the CPU implementation for four Middlebury's common test images is shown in Table III.

TABLE I
PARAMETERS FOR THE NEW ENERGY FUNCTION

λ_d	λ_s	γ_s	φ_s	λ_{st}
10.0	50.0	2.0	10.0	$ndisp/2$

TABLE II
PARAMETERS FOR THE GENETIC ALGORITHM

Population	Generations	P_{cross}	P_{Ma}	P_{Mb}	P_{Mc}
50.0	500	0.9	0.01	0.033	0.066

TABLE III
MEAN TIME SPENT FOR EACH INDIVIDUAL AND GENERATION IN CPU AND CUDA IMPLEMENTATION

	CPU (ms)	CUDA(ms)	Speed-up
Tsukuba	20.84	0.359	58.05
Venus	31.6	0.52	60.77
Teddy	45.63	0.579	78.8
Cones	46.77	0.574	81.48

The first column of Table III shows the mean total time spent for the CPU implementation for one genome. Note that not all genetic operations always occur in each genome and, therefore, these results are obtained dividing the total time spent by the algorithm by the number of genomes and generations. The increment in the CPU execution time from Tsukuba to Cones is explained due to the increment in the size of the test images.

The second row shows the same measure, but using now the GPU implementation. It is shown that parallelization of the genetic algorithm provides a great performance improvement compared with the serial one. The speed-up comparison between the two algorithms is shown in the third column. Note the increment in the performance improvement when the images get bigger, suggesting that with more pixels the GPU performs more efficiently. However, both CPU and GPU implementation still depend highly on the number of pixels in the image analyzed.

TABLE IV
MEAN TIME SPENT BY EACH GENETIC OPERATION FOR EACH INDIVIDUAL AND GENERATION IN TSKUBA AND CUDA IMPLEMENTATION

	time (ms)
Evaluation	0.155
Crossover	0.061
Occlusion handling	0.062
mutation	0.033
CPU	0.0467
Total	0.359

In order to study the impact of each genetic operation, Table IV shows in detail how the time is divided for each genome. It shows that evaluating the genome is the most demanding operation. Given that it is an operation that has to be run always in every gnome and that it is quite complex (energy function composed by several complex terms), this result is comprehensible. In comparison, the other operation that is run always and has a lot less impact in the total time is the occlusion handling. The percentage of the impact is shown in Figure 8.

As a result, it can be said that adding the occlusion handling to the algorithm implies a 17% impact on the performance. This result does not account for the impact of the occlusion variable in the evaluation operator, which here will be considered negligible.

TABLE V
MEAN TIME FOR EACH OPERATOR IN TSKUBA AND CUDA IMPLEMENTATION

	time (ms)	times per genome
Evaluation	0.155	1
Crossover	0.0663	0.9018
Occlusion handling	0.062	1
mutation	0.277	0.121

TABLE VI
MUTATION OPERATION EXECUTION TIMES

	time (ms)	times called (%)
Bilateral filter	0.273	36.78
dilate	0.283	32.19
erode	0.275	31.03

Maybe the result that was unexpected was the efficiency of the crossover function. However, although being a demanding operation, a lot of information from the evaluation process could be reused, leading to an efficient implementation. Bear in mind that the crossover it is run with a probability of Pcross, so this fact also has an impact on this measure. The same occurs with the mutation operation, that it is has a low impact due to it is rarely run.

Finally, a CPU entry in this table might seem strange at first. This time is attributed to the tasks of launching the CUDA kernels and managing the genetic algorithm itself, not the operators. As shown in Figure 2, this includes the selection operation, sorting, etc.

The measures presented in Table IV were calculated aggregating the occurrences of all the operations, but they do not occur in the same proportion. Therefore, those metrics do not represent the true performance penalty of each operation. In Table V the performance of each individual operator is shown.

These measures are the mean time spent value for each operation individually. It can be seen that, although the mutation operation has little impact on the total time spent on the algorithm, individually, it is by far the most demanding one. This is explained by the fact that a low mutation probability was set. Incrementing the mutation probability would have a great impact in the algorithm's performance. The second row of Table V shows statistically how many times each operation is called for each genome.

Finally, a more in-depth analysis of the mutation operation

is shown in Table VI.

The three different operations were configured to be triggered with the same probability, and this is represented in the second row of the table. It is shown that the three algorithms perform very similarly.

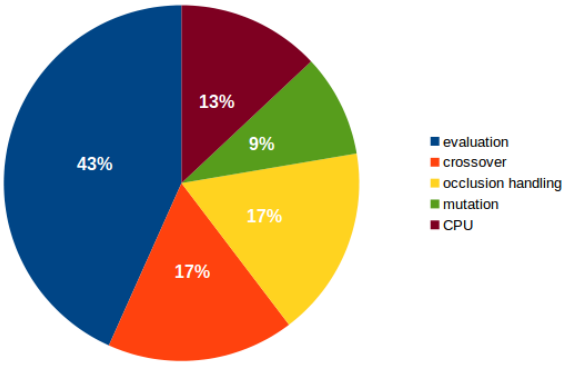

Fig. 8: Portion of time spent by each operation for Tsukuba

As a conclusion, it can be stated that approximately a 80x speedup can be achieved using a parallel implementation of the algorithm used on sufficiently big images.

VI. Conclusions

In this paper, a parallel GPGPU implementation of a genetic algorithm has been proposed. These evolutionary algorithms are very flexible and fit nicely in a parallel architecture given that the operators act independently on each individual genome. This quality suggests that parallelizing the main genetic operators would have a great impact in the algorithm's performance.

The most time-demanding genetic operators considered to be run in GPGPU were the fitness evaluation, the crossover and the mutation. However, the selection and the maintenance of the genetic algorithm itself was decided to be kept in the CPU. The main reason was that these tasks are negligible compared to the other operators; this assumption was supported by the results presented. Each operator was analyzed and a specific parallel implementation was proposed for each one.

A genetic stereo refinement algorithm with occlusion handling was selected for analysis. Using the standard Middlebury's stereo test-set, a comparison between a CPU and a GPGPU implementation was shown. As a conclusion, a great performance improvement can be achieved using GPGPU computation: a x80 speed-up has been achieved for some images. An analysis of the time spent by each operation and the impact of modifying the genetic parameters has been discussed. As a result, the most demanding operation was the fitness evaluation. This is reasonable due to the complexity of the energy function used for testing. However, considering individual function performance, the mutation operations are the most expensive, so an increment in the mutation probability would have a noticeable impact on the performance.

Evolutionary algorithms are generally not designed for real-time applications. Although a great performance improvement has been obtained, real-time performance is still not achievable for these applications. However, the GPGPU implementation improved the algorithm's performance from minutes to seconds order of magnitude.

In order to continue with this line of research, in future works, it would be interesting to try different genetic algorithm's formulations such as migrating or overlapping populations. These approaches might help avoiding local minima during the optimization. In [1] was demonstrated that this algorithm is very sensible to the fitness function. Therefore, trying different and new energy functions is likely to enhance its accuracy. Finally, for improving the algorithm's performance, trying double core GPGPUs and different platforms such as OpenCL is suggested.

References

[1] A. Arranz, M. Alvar, J. Boal, A. S. Miralles, and A. de la Escalera. Genetic algorithm for stereo correspondence with a novel fitness function and occlusion handling. In *VISAPP (2)'13*, pages 294–299, 2013

[2] Y. Boykov, O. Veksler, and R. Zabih. Fast approximate energy minimization via graph cuts. *IEEE Transactions On Pattern Analysis And Machine Intelligence*, 23(11):1222–1239, 2001.

[3] D. Gallup, J. M. Frahm, P. Mordohai, Q. X. Yang, and M. Pollefeys. Real-time plane-sweeping stereo with multiple sweeping directions. *2007 IEEE Conference on Computer Vision and Pattern Recognition, Vols 1-8*, pages 2110–2117, 2007.

[4] S. Izadi, R. A. Newcombe, D. Kim, O. Hilliges, D. Molyneaux, S. Hodges, P. Kohli, J. Shotton, A. J. Davison, and A. Fitzgibbon. KinectFusion: real-time dynamic 3D surface reconstruction and interac- tion. In *ACM SIGGRAPH 2011 Talks*, SIGGRAPH '11, page 23:123:1, New York, NY, USA, 2011. ACM.

[5] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *Int.J.Comput.Vision*, 60(2):91–110, 2004.

[6] X. Mei, X. Sun, M. Zhou, S. Jiao, H. Wang, and X. Zhang. On building an accurate stereo matching system on graphics hardware. In *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Confer- ence on*, pages 467 –474, nov. 2011.

[7] R. A. Newcombe, A. J. Davison, S. Izadi, P. Kohli, O. Hilliges, J. Shotton, D. Molyneaux, S. Hodges, D. Kim, and A. Fitzgibbon. KinectFusion: real-time dense surface mapping and tracking. In *Mixed and Augmented Reality (ISMAR), 2011 10th IEEE International Symposium on*, pages 127–136, 2011.

[8] D.-H. Nie, K.-P. Han, and H.-S. Lee. Stereo matching algorithm using population-based incremental learning on gpu. In *Intelligent Systems and Applications, 2009. ISA 2009. International Workshop on*, pages 1–4, 2009.

[9] S. Sinha, J.-M. Frahm, M. Pollefeys, and Y. Genc. Feature tracking and matching in video using programmable graphics hardware. *Machine Vision and Applications*, 2007.

[10] V. Vineet and P. J. Narayanan. Cuda cuts: Fast graph cuts on the gpu. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on*, pages 1–8, 2008.

[11] R. Yang and M. Pollefeys. Multi-resolution real-time stereo on commodity graphics hardware, 2003.

[12] K. J. Yoon and I. S. Kweon. Adaptive support-weight approach for correspondence search. *Ieee Transactions On Pattern Analysis And Machine Intelligence*, 28(4):650–656, Apr 2006.

[13] Y. Zhao and G. Taubin. Real-time stereo on gpgpu using progressive multi-resolution adaptive windows. pages 420–432, 2011.

[14] Seoane, P., M. Gestal, and J. Dorado, "Approach for solving multimodal problems using Genetic Algorithms with Grouped into Species optimized with Predator-Prey", International Journal of Interactive Multimedia and Artificial Intelligence, vol. 1, no. 5, pp. 6-13, 06/2012

[15] Mey Rodríguez, M., and E. P. Gayoso, "EvoWild: a demosimulator about wild life", International Journal of Interactive Multimedia and Artificial Intelligence, vol. 1, issue Experimental Simulations, no. 1, pp. 25-30, 12/2008

Alvaro Arranz was born in Madrid in 1984. He obtained his Industrial Engineering degree, majoring in electronic engineering, from Universidad Pontificia Comillas-ICAI in 2007. Currently he is finishing his PhD on Computer Vision and working as a researcher and developer at Bitmonlab in Zed Worldwide. His research interests include Computer Vision, Robotics, Intelligent Systems and Human-Computer interaction.

Manuel Alvar was born in Málaga (Spain) in 1984. He obtained his Industrial Engineering degree, from Universidad Pontificia Comillas-ICAI; and the Engineering degree from the École Centrale de Paris (France) in 2007. His areas of interest are: Smartgrids, Mobile Autonomous Robots, Unmanned Helicopters, Computer Vision, Automatic Monitoring, Web Tools Development.