Parallel performance of an image blurring algorithm

Iván Herrera, Nicolás Viveros
Universidad Nacional Bogotá, Colombia
biherrerap@unal.edu.co, nviverosb@unal.edu.co

***Abstract*—Is shown the performance of an image blurring algorithm in its different facets, in a parallel and sequential way. The results are studied and the components of each function are enunciated through which it is possible to carry out the actions. In addition, the load distribution and execution statistics are shown and studied. All the statistics related with response time, speedup and performance are measured with information collected from an Intel core i5-6200u dual physic-core/ quad virtual-core processor.**

I. INTRODUCTION

Through the parallelization of operations in the implementation of algorithms it is possible to achieve a great improvement in the execution performance, the case of an algorithm of image blurring under different parameters is studied. The results obtained through its execution are analyzed to obtain important information about the different factors immersed in the parallel computing procedures. The execution times and other information related with performance is calculated using an Intel Core i5 - 6200u quad-core processor.

II. THE ALGORITHM

The program implements the Gauss function for calculating the transformation to apply to each pixel in the image. In two dimensions, it is the product of two such Gaussian functions, one in each dimension:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2}\sigma^2}$$

Where x is the distance from the origin in the horizontal axis, and y is the distance from the origin in the vertical axis, σ is the standard deviation of the Gaussian distribution. When applied in two dimensions, this formula produces a surface whose contours are concentric with a Gaussian distribution from the center point. Values from this distribution are used to build a convolution matrix which is applied to the original image, this is called $the\ Gaussian\ blur$.

A. applyFilter

This function is responsible for applying the filter to the image, it receives the $image$ matrix and the $filter$ matrix calculated by using the Gauss function.

The new image $newImage$ on which the filter applied to the initial image is seen by the program as a matrix, which is traversed vertically and horizontally by applying the filter to each cell of the matrix. Also, it must be traversed for each RGB component of the matrix, so the blur can be applied properly.

fig 1. View of the matrix $newImage$, $Filter$ and $Image$ in $applyFilter$, each component of $newImage$ is calculated separatedly and doing one operation at the time in each matrix.

III. PARALLELIZATION

The parallelization of the algorithm is done through the use of threads, an amount that can be defined by choice. Each thread will be responsible for executing a part of the $applyFilter$ algorithm and in the same way its own portion of the $NewImage$ matrix performing the respective operations.

A. threadfunction

This function is the parallelization of $applyFilter$, there are some variations to integrate the use of threads in its structure. The variables $from$ and to are calculated thus defining the execution area of each thread within the $NewImage$ matrix.

1) from: The lower bound of the thread operation is calculated taking into account the number of threads and the height of the new image, based on this each thread will start at a point of the height of the new matrix where they can be uniformly distributed in each portion of the matrix many times.

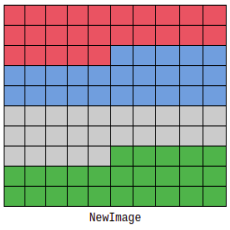

fig 2. View of the matrix $newImage$ after the calculation of the limits for four threads and defined the field of action of these uniformly. Each color represents a thread.

2) to: The upper bound of each thread is calculated taking into account the existence of previous threads and uniformly organizing its scope within the matrix, so that the previously established dimension is added to its starting point. However, it may be the case that the division to find the upper limit is not complete, in this case the last thread will be responsible for reaching the end of the matrix, so the distribution will not be done in a completely uniform manner and the last thread will have additional work.

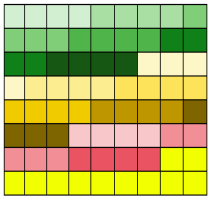

fig 3. View of the matrix $newImage$ after the calculation of the limits for 16 threads represented by a different color and defined their field of action.

Note the last thread in fig 3. , as the matrix has size 9 x 8 and $16/72 = 4,5$, it must cover the additional cells out of the other threads range. Previous technique is applied to the three matrices involved in the process, parallelizing that way the whole procedure in order to improve the program performance.

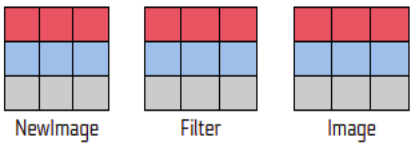

fig 4. Draw of the matrix $newImage$, $Filter$ and $Image$ in $threadfunction$, each thread deals with a portion of the matrices and calculates each component in newImage in parallel form, each color represents a thread.

B. Load balancing

The load balancing implemented is $Blockwise$ type, since the domain of each thread is centered on nx * ny blocks. Since the blocks are organized in an n_1*n_2 (in fact, the matrix $newImage$) it can be seen like an structured grid reaching that way an heterogeneous work load.

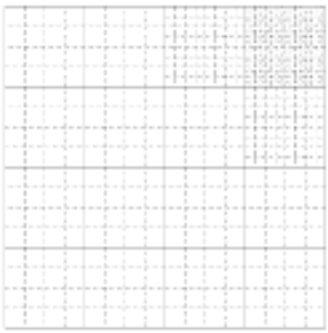

fig 5. The action of threads over $newImage$ in $threadfunction$ can be seen by stablishing defined blocks of operation through the matrix, as in Block-wise load distribution.

IV. EXPERIMENTS AND RESULTS

The program was tested in its sequential and parallel version, for images with 720p, 1080p and 4k quality. In the parallelized case, 2,4,8 and 16 threads with the same image qualities were used. Additionally, the size of the kernel was modified in different scenarios to visualize its behavior. The measurement parameters used were Response Time and Speedup.

A. Sequential version

The sequential version of the program had high execution times in relation to the quality of the images and the algorithm used.

1) Response time vs Quality of the image:

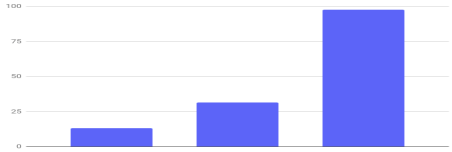

fig 6. The response time for $720p$ = $13,14$ sec, $1080p = 31,476$ sec, and $4k = 97,612$ sec respectively in the sequential version of the program.

B. Parallelized version - Threads

In the parallelized version the execution time of the program varied depending on the quantity of threads and quality of the images used, also the kernel size was modified to analize the performance of the program.

1) Response time vs thread number in 720p:

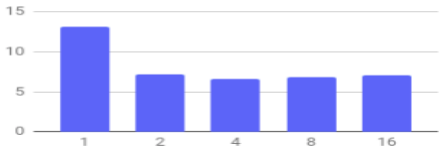

fig 7. The response time for $720p$ with $1, 2, 4, 8$ and 16 threads.

2) Response time vs thread number in 1080p:

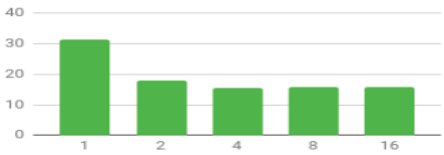

fig 8. The response time for $1080p$ with $1, 2, 4, 8$ and 16 threads.

3) Response time vs thread number in 4k:

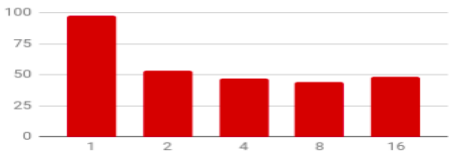

fig 9. The response time for $1080p$ with $1, 2, 4, 8$ and 16 threads.

C. Speedup - Threads

1) SpeedUp vs thread number in 720p:

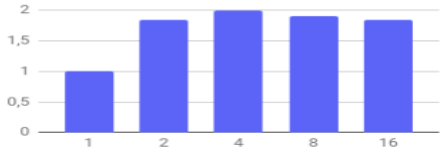

fig 10. The SpeedUp for $720p$ with $1, 2, 4, 8$ and 16 threads.

2) SpeedUp vs thread number in 1080p:

fig 11. The SpeedUp for $1080p$ with $1, 2, 4, 8$ and 16 threads.

3) SpeedUp vs thread number in 4k:

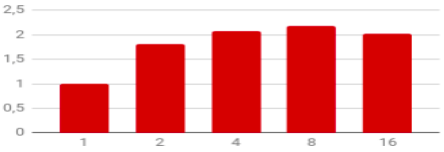

fig 12. The SpeedUp for $4k$ with $1, 2, 4, 8$ and 16 threads.

D. Parallelized version - OpenMP

To delve into the parallelization strategies to improve the program performance, have been implemented and analyzed the times and other data for OpenMP library as is shown in the next graphics.

1) Response time vs thread number in 720p:

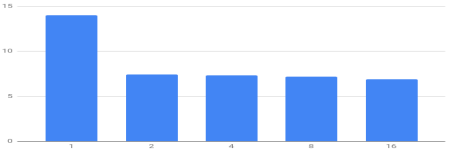

fig 13. Omp response time for $720p$ with $1, 2, 4, 8$ and 16 threads.

2) Response time vs thread number in 1080p:

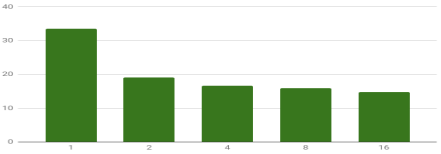

fig 14. Omp response time for $1080p$ with $1, 2, 4, 8$ and 16 threads.

3) Response time vs thread number in 4k:

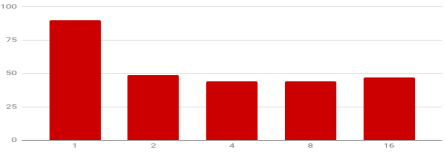

fig 15. Omp response time for $4k$ with $1, 2, 4, 8$ and 16 threads.

E. Speedup - Threads

1) SpeedUp vs thread number in 720p:

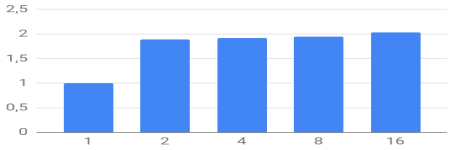

fig 16. The OMP SpeedUp for $720p$ with $1, 2, 4, 8$ and 16 threads.

2) SpeedUp vs thread number in 1080p:

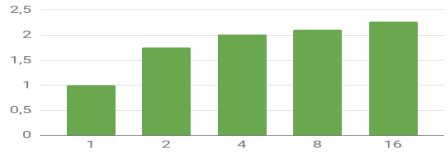

fig 17. The OMP SpeedUp for $1080p$ with $1, 2, 4, 8$ and 16 threads.

3) SpeedUp vs thread number in 4k:

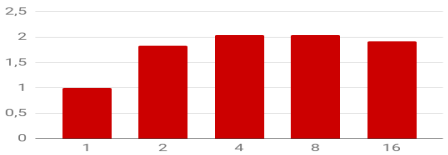

fig 18. The OMP SpeedUp for $4k$ with $1, 2, 4, 8$ and 16 threads.

CONCLUSIONS.

It have been shown that involving parallelization practices in the algorithm represented a big improvement in performance and efficiency of the program.

The best execution times varied depending on the number of threads in the definition of the image, so they were not constant for all, so that for the definitions $720p$ and $1080p$ the best times were recorded with 4 threads ($6,588$ and $15,593$ respectively). In the case of the $4k$ definition, the best response time was recorded with 8 threads and $44,588$ seconds. The SpeedUp reached its maximum peak also at the point of least response time to the corresponding number of threads.

The load balancing corresponding Block-wise type in two dimensions was useful to the extent that the distribution of threads was presented in an orderly and structured way, in order to allow a better performance adapted to the types of data and variables with the which we work.

The Gauss formulas and their respective probability distribution are very useful since it is possible to execute the blurring of the images in an orderly and mathematically viable way, thus helping in the planning and execution of parallelization through the use of them and contributing appearances of interesting images for different uses.

REFERENCES

[1] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *Applied Parallel Computing. New Paradigms for HPC in Industry and Academia*. 5th International Workshop, PARA 2000 Bergen, Norway,2001.

[2] Albert Einstein. *Zur Elektrodynamik bewegter Körper*. (German) [*Parallel Computing: Numerics, Applications, and Trends*]. Trobec, Roman, Vajteric, Marián, Zinterhof, Peter, 2009.

[3] Parallel programming models and parallelization phases
M. Allen, T. Fahringer, M. Gerndt, M. Quinn, B. Wilkinson
`http://www.dps.uibk.ac.at/ tf/lehre/ss11/ps/lectures/part2ParallelProgrammingModels.pdf`