Project 3: Image Convolution Parallelized with CUDA and OpenMP

Brenda Rangel Olvera
151108
DSI6161 - Parallel Computing

***Abstract*—For this project we implemented the convolution of an image with a filter in C and parallelized this problem with Openmp and with CUDA. The code was made without any artificial vision library and it can be used for any image size. The filter is in a .txt format and it can be modified in size and values. For the images, we used 3 different sized images to test the time it took for the convolution to be done. As it was expected, the image convolution time dropped as we added processors to the execution with OpenMP and also dropped with the use of CUDA. The test was made in a remote server instead of locally. For the bigger images the time improvement was more notorious than in the smaller image.**

***Index Terms*—Image Convolution, Filter Convolution, Artificial Vision, Parallel Computing, OpenMP, CUDA, Image Enhancement.**

I. Introduction

Image processing has had an important impact in different areas, it can be used to have an improvement in the picture information, for our own eyes' perception, but also it has been a benefit for other digital applications. The implementations of this process are endless, some examples that we can find are medical imaging such as x-ray or scanning, acoustic imaging, Forensic sciences and industrial automation. Also, pictures acquired by satellites now can be used to track or identify earth resources, geographical mapping, urban population, weather forecasting, flood or fire control.

Convolution is a way to obtain, highlight, or eliminate specific details from images. The final result or new image depends directly on the filter used, [1] goes deeper in one of the most known methods/filters in convolution, the Gaussian blur. He explains in a more detailed way some of the main applications of this filter, but also shows different calculations and methods around this. [2], on the other hand, works specifically in one method for Gaussian blur, domain splitting, and proposes it as a faster and accurate way to execute this calculation.

These applications may face different challenges related to the size, amount, or quality of the image and resources, the final result, and the processing time could be impacted by them. [3] presents one of these obstacles in image processing, irregular-shaped images, and a proposal to work with them.

Parallelization consists into divide the process in small tasks. Each task will be completed by a different processor unit (PU) at the same time. Therefore, the process will be done in less time than the serial execution of the problem. Parallelization enables solving more complex and larger problems [4] than serial executions. Parallel computing is an area of computer science that can be used in a wide range of areas inside and outside computation. It is a powerful tool that explodes the whole potential of the processor units. This is the reason why supercomputers work through parallelization. Since the cluster is going to be almost fully used, complex problems are now solvable.

The rest of the paper is structured as follows:
The second section of the document describes the problems to be solved. The convolution of images, the filters and the parallelization through OpenMP and CUDA are explained in the section. The next chapter of the paper describes the characteristics and architecture of the PC used for the implementation of the code. The consequent explains the methodology followed to create the solution (the serial one and the parallel ones). An explanation of the process is described among the pseudo-code of the algorithms. Section 5 contains the results and the discussion of those. There, we present the challenges of the implementation and the findings we had with the projects. In the last section of the paper, we describe some conclusions thrown from the results.

We leave the repository link of the project, so that everything described in the paper,can be consulted in it:
GitHub Repository

II. Description of the Problem

Image processing is widely used in artificial vision since it provides benefits when used properly. [5] Image processing can enhance images, help with noise reduction or even compression of the images with a minimal of information loss. For this project, we convolute an image with a filter that can be chosen by the user. As example, we decided to use a Gaussian filter and a Sobel filter that gives as a result a blurry image and the edges of the objects from the original image, respectively. The images chosen for the testing process are three images that are different in size.

A. Image Convolution

Convolution is the process of transforming one matrix into another one by applying a filter or a mask to the original pattern [6]. In Image Convolution, the picture to be transformed is the first matrix and the use of the filter will highlight, eliminate, or convert specific details (pixels) of the image to obtain a new one. Image convolution is based on the mathematical process of spatial transformation. This process consists of the multiplication of a pixel and its neighbor's color

value by a kernel. Equation (1) gives a detailed process of this. Fig. 1 illustrates this method.

$$g(x,y) = f(x,y) * h(x,y) = \sum_{i=0}^{i=N} \sum_{j=0}^{j=N} f(i,j)h(x-i,y-j) \tag{1}$$

Where g is the resultant matrix, f is the input image, h is the filter and it has an NxN size. This equation implies then that the last value of the kernel will multiply the first neighboring pixel and so on. Fig. 1 illustrates the case of a central pixel (201) and then the neighbor pixels that will be multiplied with the filter. The third image shows the new tone of the pixel value obtained after the convolution.

Fig. 1. Example of a convolution process

A kernel [7] is typically a small matrix used as a filter. Different sizes and different patterns of numbers will produce a different result on the image.

The convolution process needs the neighboring pixels of the one to be transformed. This implies no problem for the central pixels, but for the edge pixels, there are neighbors lacking. [7], [5], [8] propose three solutions to this edge pixel problems. The first one, is to reduce the image, which means to ignore all the edge pixels that don't have complete neighbors and only compute for those with all neighbors. The second solution is to duplicate the edge pixels, so that all the lacking pixels become 1 or -1. The third one is very simple and consists in working with the central pixels and leave the edges as noise, with values that does not correspond entirely to the proper value, but leave it as noise.

[8] mentions some of the applications that this procedure could have and they vary depending on the area. Some of the main ones could be in photography, to correct imperfections of the image, highlight areas, enhance faces or intensify properties. Other application can be found in geography or medicine to highlight specific details of the picture. In pattern recognition, convolution of images may help in the segmentation.

B. RGB to Gray Scale

An RGB image, or color image, is the one with 3 specific values on each pixel. Each of these values represents the red, green, and blue components of the pixel. In other words, an RGB image requires a 3-dimensional representation on each pixel to have all the components and details. A grayscale image, on the other hand, uses a 2-dimensional value based on the light to indicate the components of each pixel. Transforming an RGB image into a gray-scale is a way to compress a three channel matrix into two dimensions (height and width). [9] shows that there are different processes to obtain this new image, but we will explain the three most commonly used.

1) Average: One of the simplest methods for an RGB to Gray Scale process is to obtain the average of the three RGB components, which will represent the new gray value of the pixel, like the following equation expresses:

$$pixel = \frac{(R+G+B)}{3} \tag{2}$$

This transformation is one of the oldest methods to obtain a gray-scale image. The colors are not that naturally reflected but it gives a good representation of the tones and shadows. An example of a transformed image through this method is Fig. 2

Fig. 2. Example of an image (Lena) with average transformation

2) Lightness: This method, like the last one, uses an average operation as well, but it is based in contrasts. With the highest and lowest value of each pixel we can obtain the new gray pixel, as shown in Eq. (3):

$$pixel = \frac{max(R,G,B) + min(R,G,B)}{2} \tag{3}$$

This technique is well used as a fast transformation of dimensions. It brightens the contrasts of the images, like shadows, and is, therefore, useful when we want to separate regions of high contrast.

3) Luminosity: This method is more complex than the previous ones, but it uses a simple weighted sum of the three components. This technique is based on how sensitive our eyes are for each color and it applies a value according to that. The equation used for this operation is the following:

$$pixel = 0.21R + 0.72G + 0.07B \tag{4}$$

Because the human eye perceives green lighting more, the channel with more weight is that one. This transformation obtains a more natural fusion of colors. An example of this transformation is given in Fig. 3 with the Lena Image processed with this technique. This scale transformation is used widely in computer vision since it gives a wider range of gray tones to the image than the other methods mentioned.

Fig. 3. Example of an image with luminosity transformation

C. Filters

As stated earlier in this paper, a convolution requires two matrices to obtain a third one. The filter or mask is the second matrix. This will be applied to the original one and is also known as kernel. This process, in image convolution, will help us to highlight, smooth, eliminate, or modify details that will result in a different image. This update will depend on the kind of filter that is used, some examples are Prewitt, Gradient, Laplacian, sharpening, Low-Pass, High-Pass, Sobel and Gaussian [5].

1) Gaussian Filter: The Gaussian filter is used to remove or eliminate details from the original picture. Using this mask will result in a blurred image [2].

2) Sobel Filter: This filter is particularly used as a way to highlight edges in an image. The result will be an image with the X or Y axis highlighted depending on the type of Sobel filter that was used. If we want to obtain a full Sobel filter, then we have to add the convolution in X and Y to generate the full edge detection in an image. This mask requires the picture to be gray-scaled, else it would not obtain the edges of the objects in the image.

D. Open MP

OpenMP (Open Multi-processing) is an API to help the user write multi-threaded Programs. It contains a set of library routines, environmental variables and compiler directives for parallel implementations. [10] This API supports a multiprocessing programming in C, C++, and Fortran through multi-platform shared-memory. The method used for parallelization is the following: a primary thread forks a number of sub-threads that are specified previously and the system then divides the task among all of them. The sub-threads run concurrently allocating threads to different processors. This results in a faster processing time than the serial execution. The section of the code that runs in parallel (the part that the user wants to run parallel) is marked with a *#pragma* syntax.

E. Parallel Computing

Some applications solve problems that can be executed faster when we add more resources [11]. When the multi-processor can be exploited for a specific problem, then parallelization is highly recommendable. Parallel computing is based on using many threads that work concurrently to reduce the time of execution by exploiting the processors. When used properly, threads improve the performance of complex applications.

1) Amdahl's Law: Many concurrent programs consist of a mixture of serial and parallelizable portions [11]. Amdahl's law describes the proportion of how much a program could be sped up, theoretically, by adding computing resources. This is based on the ratio of parallelizable and serial portions. If P is the fraction of the code executed serially, then, the Amdahl's law states that, with N processors, the speedup that can be achieve obeys the following equation 5:

$$Speedup = \frac{1}{P + \frac{1-P}{N}} \tag{5}$$

When N approaches infinity, then the maximum speedup converges to $1/F$, which means that none code will be entirely parallelizable and there is a limit on the speedup of the process. Amdahl's law also quantifies the efficiency cost of the serialization.

F. CUDA

CUDA is the acronym for Compute Unified Device Architecture [12]. It is a parallel computing platform and also a programming model that helps developers using the GPUs when programming. It is compatible with C, C++, Fortran and more, CUDA is just usable with the extensions in form of a few keywords. These keywords help developers express parallelism with all the potential of the GPUs. It has an easy structure and it helps developers to learn and write codes in an easy way. A basic CUDA code contains 5 important steps for it to work well:

- Set memory in device and host
- Send the data from host to the device
- Execute the kernel
- Copy data from device to host
- Free memories

These steps assure that the algorithm will execute successfully at the environment. It is important to not leave any step behind.

III. Architecture

For this outline, we used the remote server of the CINVESTAV [13] which has the following specifications: An Intel Core Xeon X5675 with 6 cores and 32 GB of RAM, running at a base clock of 3.06 GHz and up to 3.46 GHz with Intel's Turbo Boost. This Core has a 12 MB Intel Smart Cache. The server also has 3 GPUs which are a NVIDIA TESLA 2070, a GTX460 and a Quadro K2000.

IV. Methodology

For the serial implementation of the Equation (1) we developed a code following the general pseudo code in Algorithm 1 . The code in C was implemented to solve the problem. We

made use of the CodeBlocks environment to code and compile the algorithm. After that, we sent the code to the server for testing. For this we used three different images which are in Fig. 4, 5, and 6. The steps followed for the algorithm are explained in the following paragraphs.

Fig. 4. Example of a small sized image. Lena. (512x512)

Fig. 5. Example of a medium sized image. Buildings. (1182x800)

Fig. 6. Example of a large sized image. Landscape.(3840x2160)

These images were chosen based on their size, in order to have different images for the testing part of the code. The images were selected from Google images and transformed into JPEG format to uniform them. These images are open source and we can used them without any problem for the purpose.

Algorithm 1 Serial Code for Convolution

```
Input: Filter, Image
Output: Convoluted gray_image
Initialization of Variables
Image ← length of Image to Convolute
Filter ← content of Filter.txt
for each pixel value in image do
    calculate gray value of pixel with luminosity method into gray_image
    pixel = 0.21R + 0.72G + 0.07B
for each pixel value in gray_image do
    convolute gray pixel with matrix into conv_image
crop border of conv_image
save resultant image
free memories
show convolution time
```

For the C serial code we firstly included the libraries needed for printing, reading, timing, resizing, and use of strings. We needed to implement three libraries that weren't part of the compiler for the images. These were the open source libraries STB_IMAGE_IMPLEMENTATION, STB_WRITE_IMPLEMENTATION and STB_IMAGE_RESIZE_IMPLEMENTATION. Without these, the use of images wouldn't be possible.
Then, like is stated on Algorithm 1, we read the matrix of the filter mask. For this we needed to create a matrix with the size of the filter and we scanned the values from the text file and filled the matrix with them. For the main program we loaded the image (in vector form) and the filter text and then read it (with our function). We obtained their sizes and, after that, we transformed the color picture into gray-scale. This was achieved with the use of the luminosity method described in previous sections. Each color pixel was transformed with Eq. (4). The luminosity Gray Scale highlights more details than the average and lightness methods.
The next step, according to the pseudocode, is the convolution of the gray image and the filter. For this paper, we used three masks to test the code, a Gaussian Matrix of 3x3:

$$\begin{bmatrix} 0.0625 & 0.125 & 0.0625 \\ 0.125 & 0.25 & 0.125 \\ 0.0625 & 0.125 & 0.0625 \end{bmatrix} \tag{6}$$

and Sobel masks in X and Y with the next 2 matrices:

- Sobel in Y

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{7}$$

- Sobel in X

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \tag{8}$$

These three filters were available for the convolution. Since the previous project showed similar timing results between filters, we opted to test with the Gaussian Filter. We convoluted the gray images with Eq. (1) with three for loops: one for the gray image, two for the filter matrix. The convolution is done pixel by pixel. Therefore, a new vector for the convoluted images was created and filled with the resultant pixel at each iteration of the image pixels. It was also timed the convolution process to be able to analyze it afterwards. We cropped the borders of the resultant image, since those were not processed due to the methodology of convolution. For the final steps we saved the new resized picture and freed the memories. All those steps were done for the serial implementation of the code.

For the parallel code, we implemented the flow of the serial pseudocode and added the necessary conditions to parallelize the convolution section of the method. In algorithm 2 there is specified where the changes had to be done to obtain the parallel version of the method.

Algorithm 2 Parallel Section of the Code using OMP

```
Input: Filter, gray_img
Output: conv_img
Initialize omp with its necessary values
#pragma omp parallel for shared(variables) schedule(variables)
for each pixel value in gray_image do
    convolute gray pixel with matrix into conv_image
save resultant image
show convolution time
```

For the parallelization, we are using the API OpenMP. Essentially, there are not so many changes from the serial code. Key here is to know what to change and the reason. We set the number of threads at the beginning of the code and those can be changes by the user. Threads are in charge of convolute a part of the image with the filter. If more threads are involved then the time of execution will decrease. This is valid when it follows Amdahl's Law. After a certain number of threads, the time will stay around the same, even though more threads are used. We use the #pragma syntax characteristic of OMP to indicate where the parallelization will be performed. There we also specify the variables that will be shared or will stay private; by default the counters stay private, because each thread has to work with its own copy of the counters to modify them without affecting the work done by another thread. The shared variables for this problem are the images (gray image and the convoluted image), the filter and its properties. This variables are shared because they either stay constant through the whole process (in this case the filter and its size and the gray image) or are modified to create a whole new product (in this case, the resultant image) because all threads will have to modify this variable to obtain the result. The last consideration that we had was the scheduling of the process. This feature of OMP allows to have control on how loop iterations will be mapped onto the threads. There are different schedulings and for this project we are using static, dynamic, and guided. The iterations are also taken into consideration since that modifies the size of the blocks passed to each thread. Those are the main differences between both implementations, serial and parallel for the convolution of an image with a filter mask.

We also implemented the convolution in CUDA to be able to compare its performance against Open MP. For this, we created a code that followed the steps mentioned before. This can be seen in the Pseudocode 3 where those steps are implemented to the convolution.

Algorithm 3 Parallel Code with CUDA

```
Input: Filter, Image
Output: conv_img
Initialization of variables for host and device
Read image and filter
for each pixel value in gray_img do
    calculate gray value of pixel with luminosity method
copy gray_img and filter from host to device
In kernel: convolute gray_img with matrix into conv_image
Copy conv_img from device to host
save resultant image
show convolution time
free memories
```

The first step is to give memory allocations for the convolution. There we gave space for the image in the host and device and for the filter. We also created the space for the new image that was going to be created. Second step is sending the image and filter data from the host to the device. Then, we call the kernel function to work, which in this case is the convolution function, which follows the equations for this process but instead of doing the model in a for loop, we we separate the task in blocks and each thread deals with one of those blocks. There is the parallelization done. Since the GPU can handle that amount of threads, each work in a serial manner while all threads are working simultaneously.

After this process is completed. The data created or manipulated must be sent back from the device to the host. And, for the last step, memories need to be freed. We also timed the convolution function to be able to compare it with the time obtained from using openMP.

When we wanted to use the image libraries used before to read and save the pictures in the code, we encountered that the version of CUDA that we were using is not compatible with STB_IMAGE and others. Therefore, we used the libraries Image.h and PPM.h to read and write the images. This is one of the reasons why we couldn't reuse the code created before.

We added those headers to the repository to download with the code for the implementation.

V. Results and Analysis

As described in the previous section, the images were preprocessed into Gray-scale and afterwards convoluted with different filters. In Table I the results of the convolution time for are presented. This table contains only the information obtained with the small picture when we change the number of threads and schedule. All values are in miliseconds and an average of 10 executions each. As it can be seen, the convolution time decreases when the number of threads increases. The time decreased around 82% from the serial time of 114 miliseconds.

TABLE I
Convolution time for the small Image (Lena)

	serial	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
Convolution Time dynamic	114ms	68.9ms	37.7ms	28.2ms	31.0ms	26.5ms	24.8ms	22.9ms	19.5ms	19.2ms	17.5ms	19.4ms
Convolution Time static	112ms	56.3ms	46.8ms	35.9ms	22.6ms	19ms	16.1ms	14.2ms	12.6ms	11.7ms	10.3ms	12ms
Convolution Time guided	112ms	56.2ms	37.5ms	28.1ms	22.5ms	18.8ms	16.1ms	14.1ms	12.6ms	11.4ms	10.3ms	11.8ms
Convolution Time dynamic 10	111ms	77.9ms	48.6ms	36.7ms	28.9ms	23.9ms	20.3ms	17.7ms	15.8ms	14.4ms	13.2ms	12.9ms
Convolution Time static 10	111.2ms	65.7ms	53.8ms	46.3ms	38.1ms	32.6ms	28.3ms	64.7ms	22ms	20ms	62ms	72ms
Convolution Time guided 10	111ms	74ms	50.5ms	65ms	28.9ms	58.6ms	20.2ms	17.7ms	44ms	14.4ms	40.5ms	63.5ms
Convolution Time dynamic 100	111ms	75ms	46.8ms	34.4ms	26.8ms	21.9ms	18.5ms	16.1ms	14.1ms	12.5ms	11.4ms	11.5ms
Convolution Time static 100	111ms	83ms	60ms	46.1ms	38.3ms	32.5ms	28.3ms	60.2ms	22.6ms	20.5ms	58.7ms	66.8ms
Convolution Time guided 100	111ms	75.7ms	79.6ms	34.4ms	60.9ms	21.9ms	67.7ms	16.1ms	14.1ms	12.7ms	12.5ms	42.6ms
Convolution Time dynamic 1000	111ms	74ms	46.4ms	33.5ms	26.3ms	21.5ms	18.3ms	15.8ms	13.8ms	12.5ms	11.3ms	12.8ms
Convolution Time static 1000	111ms	71ms	50.5ms	46ms	38.1ms	32.4ms	28.2ms	56.3ms	22.6ms	20.7ms	67.2ms	77.5ms
Convolution Time guided 1000	111ms	75.1ms	46.4ms	55ms	26.1ms	62.5ms	18.2ms	63.8ms	13.9ms	12.7ms	59ms	65.3ms

In table I three different scheduling are reported. When using a dynamic schedule, the time of convolution is slightly greater than using static or guided. This is due to the structure of the scheduling. When dynamic is chosen then each thread works with a block of iterations and when it is done, it works with the next one until the iterations queue is empty. This can cause that the number of iteration blocks gets uneven distributed and results in a slightly slower process than the other two. When static or guided is chosen, then the convolution time is very similar since those scheduling avoid this uneven distribution of the iteration blocks.
In terms of iterations, the timing is faster when few threads are taken in consideration when no chunk size is given to the program. When more threads are involved then the convolution time decreases but at some points it increases in comparison to no chunk size against a predefined chunk. This applies when the scheduling is static or guided. Since this process of convolution is a dynamic problem, the dynamic scheduling is the one that suits best for this issue and the timing results shows it. When dynamic scheduling with a fixed chunk size is chosen, then the resulting convolution timing behaves how we expected. In Fig. 7, 8, and 9 this behaviour is plotted. The time of convolution decreases faster when no chunk is defined but at larger number of threads, the convolution is faster when a defined chunk is chosen. Because of the working definition of the dynamic scheduling, it is best suited for the program and it behaves accordingly to Amdahl's law. There is no significant difference when a larger block size is defined, the important manner is to have a specific size of iterations, so that each thread works on that specifically. As it can be seen in Fig. 7 the convolution time decreases at its best when a chunk is defined.

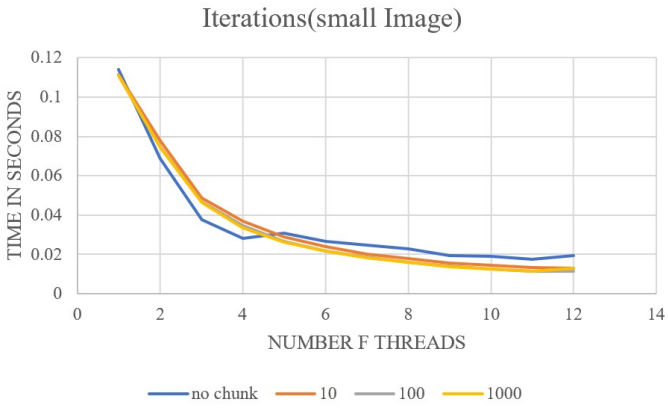

Fig. 7. Graph of the Speedup when large image is processed

This result is due to the size of the "chunk". When guided scheduling is performed, threads take bigger blocks at first and reduce its size until chunk size. If this value is large, then this decrease is not so significant. When static scheduling is performed, then the chunk size is constant. The iterations are evenly distributed in blocks of size chunk between threads. This handling of iterations causes the processing times to be different. A possible reason on why sometimes it takes more time with a bigger number of threads can be that threads
Table II illustrates the convolution times taken from the performance of execution when a medium sized image is processed. This table denotes a similar behaviour than table I. When the number of thread increases, the convolution time decreases. Also, when dynamic scheduling is applied, the convolution is slightly slower than when static or guided is chosen, when no iteration blocks are established.

TABLE II
Convolution time for the medium Image (Buildings)

	serial	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
Convolution Time dynamic	412ms	249ms	169ms	129ms	112ms	95.7ms	89.5ms	82.4ms	70ms	69.1ms	63.2ms	62ms
Convolution Time static	405ms	203ms	135ms	101ms	81ms	67.8ms	57.9ms	51ms	45.1ms	40.7ms	36.9ms	37.8ms
Convolution Time guided	404ms	202ms	135ms	101ms	81ms	67.6ms	57.9ms	50.7ms	45.1ms	41ms	36.9ms	34ms
Convolution Time dynamic 10	401ms	281ms	175ms	132ms	103ms	85.4ms	72.7ms	63.5ms	56.4ms	51.6ms	47.4ms	44.8ms
Convolution Time static 10	401ms	303ms	217ms	172ms	144ms	122ms	106ms	93.1ms	83.3ms	103ms	104ms	154ms
Convolution Time guided 10	401ms	311ms	186ms	133ms	106ms	86.5ms	82.8ms	74.1ms	66.1ms	69.7ms	65.7ms	13.3ms
Convolution Time dynamic 100	401ms	275ms	174ms	128ms	98.3ms	80.1ms	67.3ms	58.4ms	51ms	45.3ms	41ms	37.3ms
Convolution Time static 100	401ms	313ms	223ms	173ms	144ms	122ms	106ms	93.6ms	83.8ms	102ms	97.3ms	134ms
Convolution Time guided 100	402ms	282ms	177ms	125ms	126ms	96ms	84ms	68ms	60.2ms	96.6ms	75ms	166ms
Convolution Time dynamic 1000	400ms	274ms	172ms	124ms	95.6ms	77.9ms	65.5ms	56.5ms	49.8ms	44.4ms	40.3ms	36.5ms
Convolution Time static 1000	401ms	312ms	221ms	171ms	143ms	121ms	104ms	111ms	101ms	114ms	244ms	243ms
Convolution Time guided 1000	400ms	221ms	166ms	132ms	99ms	108ms	94.5ms	73.6ms	49.8ms	44.3ms	40.3ms	71.7ms

When iterations are taken into consideration, a similar behaviour to small image processing was to be seen, as expected, in Fig. 8. When a block size is defined, the dynamic scheduling is the fastest and without anomalies. The results of table II are

based on 10 executions of the program at each schedule, thread and number of iterations. Times are reduced around a 85% of the serial time. The time taken with 12 threads for the medium sized image is around the time taken for a small sized image with two threads of parallelization.

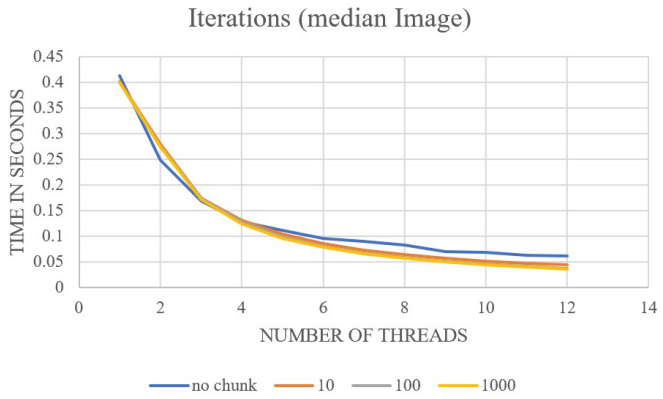

Fig. 8. Graph of the Speedup when small image is processed

Table III the behaviour of the different variables and how those affect the convolution time is shown. These times were taken with the bigger sized image which has a similar behaviour to the previous ones. As expected, the time taken for convolution is larger than the other two but it decreases while more threads are added to the execution. With parallelization (12 threads), the time taken for the convolution is almost the same as the time taken for a medium sized image serial implemented.

TABLE III
CONVOLUTION TIME FOR THE LARGE IMAGE (LANDSCAPE)

	serial	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
Convolution Time dynamic	3.62s	2.18s	1.5s	1.13s	958ms	838ms	780ms	723ms	618ms	605ms	557ms	542ms
Convolution Time static	3.55s	1.78s	1.19s	888ms	711ms	591ms	508ms	443ms	395ms	355ms	323ms	230ms
Convolution Time guided	3.55s	1.77s	1.18s	887ms	710ms	592ms	507ms	444ms	395ms	355ms	323ms	296ms
Convolution Time dynamic 10	3.5s	2.5s	1.5s	1.15s	920ms	767ms	644ms	562ms	498ms	456ms	485ms	421ms
Convolution Time static 10	3.5s	2.6s	1.87s	1.45s	1.2s	1.01s	888ms	787ms	709ms	643ms	589ms	547ms
Convolution Time guided 10	3.5s	1.96s	1.46s	1.19s	1.09s	779ms	710ms	556ms	493ms	453ms	438ms	460ms
Convolution Time dynamic 100	3.5s	2.1s	1.6s	1.22s	885ms	768ms	727ms	563ms	535ms	497ms	357ms	427ms
Convolution Time static 100	3.5s	2.7s	1.94s	1.54s	1.24s	1.06s	923ms	820ms	1.18s	662ms	600ms	555ms
Convolution Time guided 100	3.5s	2.4s	1.5s	1.1s	896ms	700ms	591ms	544ms	1.3s	395ms	370ms	433ms
Convolution Time dynamic 1000	3.5s	1.99s	1.5s	1.08s	838ms	682ms	813ms	791ms	576ms	508ms	560ms	521ms
Convolution Time static 1000	3.5s	2.7s	1.9s	1.5s	1.2s	1.1ms	916ms	810ms	721ms	652ms	594ms	548ms
Convolution Time guided 1000	3.5s	2.4s	1.5s	1.07s	925ms	819ms	662ms	597ms	637ms	409ms	348ms	317ms

Tables I II III illustrate that the behaviour is consistent for any size of image when it comes to parallelization and serial implementation of a process. It is not relevant the size of the image or filter used for the convolution for effects of parallelization. The time will decrease accordingly to the Amdahl's Law while we use more threads. The reduction is similar to an exponential behaviour, which means that after some threads the convolution time does not decrease significantly because of the critical timing of the process, while at first the time decreases significantly.

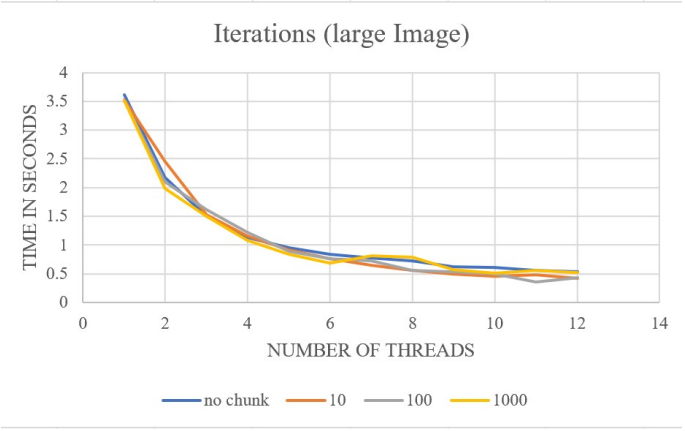

Fig. 9. Graph of the Speedup when small image is processed

The behaviours of convolution timing when serial and when parallelized are resumed in the following graphs. Fig. 10 illustrates the serial time against the size of image. Timing is in miliseconds and it is clear that the landscaping images take more convolution time than the other two, while the small image is the fastest to be processed. All those times were executed in the server and from the last project, when using the local computer, convolution was slower.

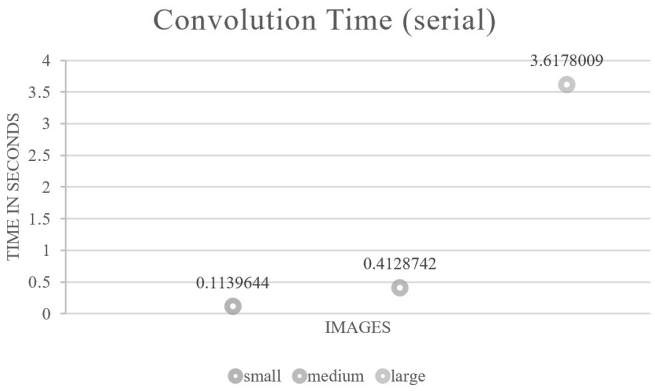

Fig. 10. Graph of the Serial Convolution Time of the different images

The initial convolution time is directly increasing along with the size of the images, as it can be seen in Fig. 10. This could be easily explained by the quantity of pixels on the picture due to the amount of operations that will be executed. Each new pixel will represent an exponential increase in the calculations that the program will perform. This behaviour is both applicable to serial and parallel executions.

In Fig. 11 is illustrated the behaviours described before. The Speedup curve for all three images is exponentially inverse and tend to converge over nine threads. After that the difference in convolution time is minimal, which means it arrived to the maximal speedup proposed in the Amdahl's law. Because the static and guided timings are almost identical, the curves overlap each other at some points of the graph.

This is more clear in table I. The static scheduling takes more time for some number of threads but there is still a decrease of timings between two and five threads. After those numbers, the dynamic scheduling curve stays above the other two.

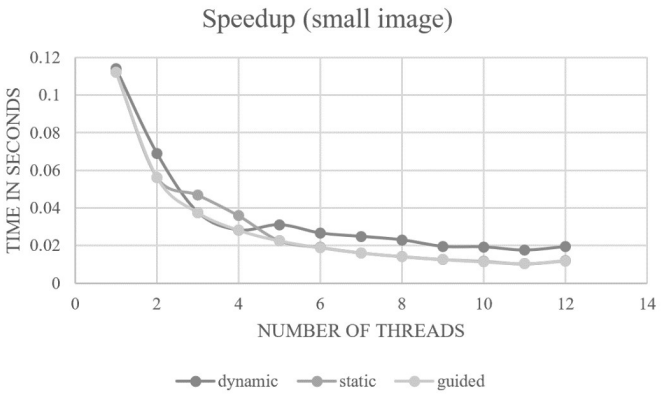

Fig. 11. Graph of the Speedup when a small image is processed

Fig. 12 and Fig. 13 represent the behaviours when a medium and large images are processed, respectively. Both graphs have a very similar shape and behave as described previously. There is visible that the code is working as expected and the convolution time decreases when more threads are used. Because the server has six cores, we decided to use up to 12 threads to really obtain significant results, because after that number, the timing was staying almost at the same as with twelve threads.

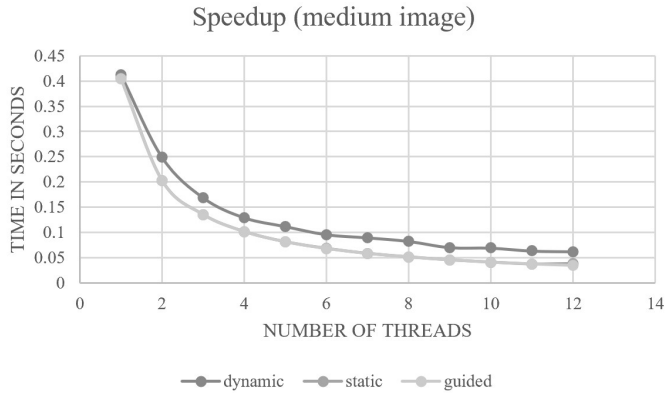

Fig. 12. Graph of the Speedup when a median image is processed

For the code in CUDA, we had some concerns in terms of times of execution. We were having consistency problems on the timing results and found some execution errors when looking at the picture that was created. We had to implement a new way to read and write the images since CUDA is not compatible with the previous libraries. Now the time of reading and writing even though is not timed, it felt slower than with the other libraries. The code for the implementation

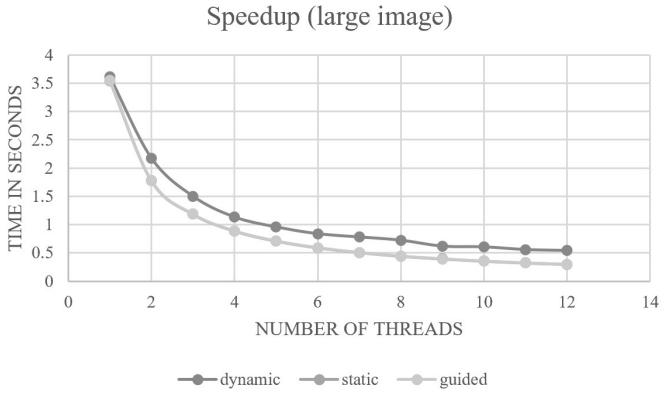

Fig. 13. Graph of the Speedup when the large image is processed

changed and had to be done almost completely again to be able to use it with CUDA, like it was mentioned on the previous section. We were having problems to obtain the convolution times and actually decided to use different timing tools but they all came to same results. This was due to the fact that CUDA's implementation is very fast that the time tracker barely notice the difference. We made also some research about timings and the Hardware times are not so precise when it comes in terms of nanoseconds and it is very difficult to track that. Therefore, even though it is not zero, it appears to be that way because of the very small difference of time that is perceived by the time track. We also did some tests with matrices and multiplications to see if it changed, but when we added matrices of the size of our images, times went back to zero. It needs to have bigger images to really get an impact in timing. At least this was the case for the small and middle sized images (lena and buildings). Also, using the different devices that the server has available, the times didn't differ enough to draw a conclusion about having many differences between them.

Therefore we decided to compact all results, serial, OpenMP and CUDA in one table, which is IV. Here we present a summary of the times we obtained in the different implementations and with all the image's sizes.

TABLE IV
CONVOLUTION TIME FOR THE DIFFERENT IMPLEMENTATIONS AND IMAGES

		Implementation		
		serial	OpenMP	CUDA
Size of Image	small	111 ms	11.5 ms	0.0 s
	median	401 ms	34.6 ms	0.0 s
	large	3.5 s	406 ms	273 ms

As it is described in Table IV, CUDA's implementation is the fastest of all three. For the other codes, we used the averaged results of the previous tables. For OpenMP we used the times when we used 12 threads and a dynamic scheduling for this average. The times in CUDA were tested with different methods and gave the same results over the

time. We can conclude that they are so close to zero and due to hardware imprecise timing the time taken for Lena's and building's images is very low. We are also no expert on CUDA so maybe those timings can be result of a mistake or violation done during the programming step. We also believe that due to the high potential that the GPUs have in terms of programming, those timings are correct, since we are exploding the computational power of the server to obtain faster results. It is also a fact that CUDA is more powerful than other parallelization tools, which made us believe that the convolution is correct. GPU processing is faster than CPU's and it can be seen in the results obtained. The images obtained after the convolution are consistently similar to the ones we had previously.

A. Challenges

As described in Section IV, we needed to implement libraries for the images to be processed in C. There is no library in the C compiler that can load an image and write one. Therefore, for the implementation with OpenMP we decided to use STB_IMAGE_IMPLEMENTATION and STB_WRITE_IMPLEMENTATION, which are both Open Source libraries for C and C++. With these we were able to load and read the input image and then write and save the resultant image. While for the implementation in CUDA, we used the libraries IMAGE and PPM to do this. This was due to an error that was appearing when compiling with STB and CUDA. Therefore, we decided to opt for other libraries.

As it was mentioned in Section II-A, the edge pixels come to the issue that there are missing neighbor pixels for the convolution. It was also mentioned that there are three possible solutions to this problem. We decided to treat these nonexistent pixels as noise and sent them to the outermost rows and columns of the convoluted image. After that we cropped those pixels, reducing the size of the image by the filter size minus one (in case of a 3x3 filter, the convoluted image was reduced by two rows and two columns). For this process we reduced the size of the image by leaving the borders out of the convolution.

For the parallelization we used the API OpenMP and also CUDA. This was the first time for us using this API and we had to learn to use its commands. The theory behind was easy to follow but to put all that in practice was harder. Some concepts weren't clear enough from the presentations and we had to do more research around those topics. Even though in the code it is as simple as just a few lines, those concepts like schedule, reduction, private or shared were hard to decide based on their purpose and the syntax of the parallelization part.

For CUDA it was also the first time we used it and the course we took about it was very helpful. Some methods were more trouble than others and some concepts were hard to understand, but at the end with a little guidance it was easier to understand what was needed to be done. It was interesting to process and solve problems in the compilation and the thought process to change from cycles to a serial/parallel way of seeing and handling the convolution problem.

Another challenge was to use the CINVESTAV's server since we were used to work with windows and in an application like CodeBlocks and not on the batch directly as well as in Linux. Although the operative system seems to not have a role on a programming language, we discovered that many issues were invisible for windows, since while the program was already executing without any flaw on windows script, on Linux at the server we were having many troubles to compile or execute the program. The most common error we had was a "segmentation fault" which can be due to many different factors. This was very hard to solve and we were stuck there for a while since we had to try different approaches to find the problem. Another issue was declaring variables inside a nested loop, which the server never compiled and we had to restructure the code some times to obtain a version that was usable in UNIX. Part of why it was so difficult to find the problems and what was causing them, was the fact that the server doesn't include a debugger at the batch and we have locally a windows operating system.

The challenges mentioned in the last paragraph weren't a problem when working with CUDA, but many others came up: for example, the writing mode for each instruction, since we had to think if that was performed by the device or by the host. We also needed to determine many variables for CUDA like the blocks for the threads that we didn't used on previous codes. We also had to deal with the definition of events and synchronization which is not that intuitive to work. Also the CUDA compiler gives more information than the normal compiler of the prompt. This was very useful to debug and correct some mistakes and logic problems.

VI. Conclusions

Image convolution is a commonly used technique in artificial vision. This method helps in different tasks for image processing and it can improve pictures. This technique, although it is old, it is still valid in many applications. The method consist in a replication of the sum of the multiplication of pixels. Since this process needs to be repeated for every central pixel of the image, this procedure is highly parallelizable. This is the reason why we implemented the convolution with the help of OpenMP and then CUDA. The results in time of the convolution when is parallelized are consistent with the theory of scalability and the whole program is faster than the serial implementation of the process. This is the case for both, Open MP and CUDA.

This project was a continuation of the one we did before but it was still very challenging to deal with the theory of parallel computing since CUDA is nothing alike to OpenMP and both are resources to parallelize. It was interesting to see all the theory executed in a real problem and optimize the times of convolution by using more computational power. CUDA is very powerful and the images used for the convolution are small in comparison to what CUDA is capable to handle. It is very fast because it can create way more threads than OpenMP

and the CINVESTAV has big GPUs. The GPU potential is huge. The use of the GPU for an implementation that was very complex for our local CPU, was very fast to be done at the GPU. The times it takes to compile and do the earlier steps of the program is similar to serial and OpenMP implementation. CUDA is complex to understand but it can give amazing results when used correctly. It is no guaranty that it will give better results than other implementations but it surely is more powerful. To use the CPU to give orders to the GPU was something that we had not done before but it is very good to know it can be done and the results on performance are interesting. Although we got some weird numbers for the convolution time, like 0.0 ms for the smallest image, it was still interesting to test why this was happening. It is a good way to understand what is happening inside the hardware and not just the software implementation.

ACKNOWLEDGMENT

The Author would like to thank Dr. Mireya Paredes López for her guidance in this project. Also, to thank David Limón Cantú for working together in the development of the pseudocode, the serial code as part of an assignment. Also a special thank to Dr. Amilcar Meneses Viveros for the tutorial and introduction course for CUDA.

REFERENCES

[1] J. Flusser, S. Farokhi, C. Hoschl, T. Suk, B. Zitova, and M. Pedone, "Recognition of Images Degraded by Gaussian Blur," *IEEE Transactions on Image Processing*, vol. 25, no. 2, pp. 790–806, Feb. 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7364266/

[2] S. Yoshizawa and H. Yokota, "Fast L1 Gaussian convolution via domain splitting," in *2014 IEEE International Conference on Image Processing (ICIP)*. Paris, France: IEEE, Oct. 2014, pp. 2908–2912. [Online]. Available: http://ieeexplore.ieee.org/document/7025588/

[3] Y. Jiao, C. Qian, and S. Fei, "Mask Convolution for Filtering on Irregular-Shaped Image," in *2018 17th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES)*. Wuxi, China: IEEE, Oct. 2018, pp. 115–118. [Online]. Available: https://ieeexplore.ieee.org/document/8572537/

[4] G. R. Joubert, *Parallel computing: on the road to exascale*, ser. Advances in parallel computing. Washington, DC: IOS Press, 2016, no. v. 27.

[5] R. Gonzalez and R. Woods, *Digital Image Processing*, 2nd ed. New Jersey: Prentice Hall International Edition, 2002.

[6] E. S. Gedraite and M. Hadad, "Investigation on the Effect of a Gaussian Blur in Image Filtering and Segmentation," in *ELMAR*, 2011, p. 5.

[7] J. Ludwig, "Image Convolution," Portland, US, 2004. [Online]. Available: http://web.pdx.edu/~jduh/courses/Archive/geog481w07/Students/Ludwig_ImageConvolution.pdf

[8] F. Giménez-Palomares, J. Monsoriu, and E. Alemany-Martínez, "Application to convolution of matrices to image filterig," *Modelling in Science Education and Learning*, vol. 9, no. 1, p. 12, 2016. [Online]. Available: http://polipapers.upv.es/index.php/IA/article/view/3293

[9] T. Kumar and K. Verma, "A Theory Based on Conversion of RGB image to Gray image," *International Journal of Computer Applications*, vol. 7, no. 2, pp. 5–12, Sep. 2010. [Online]. Available: http://www.ijcaonline.org/volume7/number2/pxc3871493.pdf

[10] T. Mattson and L. Meadows, "A "Hands-on" Introduction to OpenMP," p. 153.

[11] "Amdahls Law | Performance and Scalability." [Online]. Available: https://flylib.com/books/en/2.558.1/amdahls_law.html

[12] "CUDA Zone | NVIDIA Developer." [Online]. Available: https://developer.nvidia.com/cuda-zone

[13] "Cluster Híbrido de Supercómputo - Cinvestav." [Online]. Available: http://clusterhibrido.cinvestav.mx/

[14] A. Asmaidi, D. S. Putra, M. M. Risky, and F. U. R, "Implementation of Sobel Method Based Edge Detection for Flower Image Segmentation," *SinkrOn*, vol. 3, no. 2, p. 161, Mar. 2019. [Online]. Available: https://jurnal.polgan.ac.id/index.php/sinkron/article/view/10050

[15] D. Mouris, "jimouris/parallel-convolution," Jun. 2020, original-date: 2015-10-06T09:03:43Z. [Online]. Available: https://github.com/jimouris/parallel-convolution

[16] A. a. k. a. KZKGˆGaara), "Programando en Bash - parte 2," Oct. 2012. [Online]. Available: https://blog.desdelinux.net/programando-en-bash-parte-2/

[17] "Bash scripting Tutorial - LinuxConfig.org." [Online]. Available: https://linuxconfig.org/bash-scripting-tutorial

[18] A. a. k. a. KZKGˆGaara), "Programando en Bash - parte 3," Oct. 2012. [Online]. Available: https://blog.desdelinux.net/programando-en-bash-parte-3/