Speed Up of Image Filtering using Parallel Programming Models

José Agustín Barrachina
IEEE Student Member
École Polytechnique
Université Paris-Saclay
Route de Saclay, 91128 Palaiseau, France
Email: joseagustin.barra@gmail.com

François Bidet
École Polytechnique
Université Paris-Saclay
Route de Saclay, 91128 Palaiseau, France
Email: francois.bidet@polytechnique.edu

***Abstract*—The aim of this project was to speed-up a target application through multiple parallelism models (MPI, OpenMP and CUDA) for various architectures (CPU and GPU) and conclude which approach is the most suitable for the specific application.**

1. Introduction

A base C/C++ code that implements an image filtering was given. The aim of this work is to optimize the operation time of the code by using parallel programming methods such as MPI, OpenMP and CUDA. The base C/C++ code covers a large spectrum of algorithms in HPC and Big Data. The Image Filtering can be seen as a direct application of Big Data programs. This application uses a stencil-based scheme to apply a filter to an existing image or set of images. It means the main process of this code is a traversal of each pixel of the image and the application of a 2D stencil [1].

We will work with GIF image because this format has the following advantages:

- Easy to manipulate
- Widespread format (social networks)
- Allow animation (multiple images). which can be paralized as well.

1.1. Project Organization

This project was done using github version control repository [2]. Each different strategy of parallelizing was made in a different branch, having as master branch the original serial program. So that in the end a merge was done with the strategies that proved to work better.

The different branches are as follows:

- **Master** Contains the file with the sequential code.
- **Result** Contains the final realization with a combination of CUDA, MPI and OpenMP. Section: 3.2.2
- **MPI_over_images_of_gif** Applies MPI to treat each image of a gif separatelly. Section: 3.1.1
- **Bidouille** Applies MPI to divide the work on each image. Section: 3.1.2
- **OpenMP_divide_image** Applies OpenMP to divide the work on each image. Section: 3.2
- **CUDA_filtering** Applies CUDA to divide the work on each image. Section: 3.3

1.2. Testing Work

To test the efficiency of the program, many files were created. A simple C program was done to make a summary of all the results obtained. This summary is only an addition of the time taken to apply the filter to each gif image.

Another program was done to compute a coefficient which represents the ratio of time taken by our program compared to the time taken with the sequential program version, which is $c = T_t/T_p$ where c is the given coefficient, T_t is the time taken by the sequential program and T_p is the time taken by the implementation method. By doing the average of this coefficient over a variated amount of gif files we can obtain an estimation of the expected efficiency improvement of a gif image without knowing anything of the file in question.

Also, a shell file was created to run many gif images one after the other. Various variations of this shell were made to run the test only on gif files that share some property as for example, containing only one image or on the contrary containing more than one image.

With the help of ImageMagik [3]. Two shell files were created to compare the original (serial) program output with the new parallel optimized program in order to check that it works perfectly. The first bash file creates a output image that, if both images are exactly the same, should create a complete black image. Because this method can be difficult to be sure the images are exactly the same, a second bash file was created that compares pixel by pixel each image and output 0 if the image is completely identical. In that case, the first file is used to check if the images are identical and if they are not, the second bash can be used to see what the mistake is.

2. Image Filtering Program

This topic focuses on a specific image filter that is useful to detect objects and edges inside various images. This filter is called Sobel and it applies on a greyscale image. Therefore, the first step is to transform the image from a color one to a set of gray pixels.

Then, a blur filter is applied to a small part of the images to exclude the borders. Thus, the main goal is to parallelize an application that apply multiple filters (grayscale, blur and Sobel) to a GIF Image (either a single image or an animated gif). [1]

2.1. Structure

The algorithm can be divided into three main parts:

1· GIF import (section 2.1.1)
2· SOBEL filter (section 2.1.2)
3· Export image (section 2.1.3)

A main function already measures and output the time taken by each of the three listed parts of the project.

2.1.1. GIF import. This part just load the gif file into a pixel array.

2.1.2. SOBEL filter. The sobel filter or Sobel-Feldman operator is used in image processing and computer vision particularly within edge detection where it creates an image emphasizing the edges. Named after Irwin Sobel and Gary Feldman.

This part of the code applies actually three different filters:

1· **Gray Filter** Convert the pixels into a grayscale.
2· **Blur Filter** Apply blur filter with convergence value as seen on Figure 1. This part take about 70% of the total processing time.

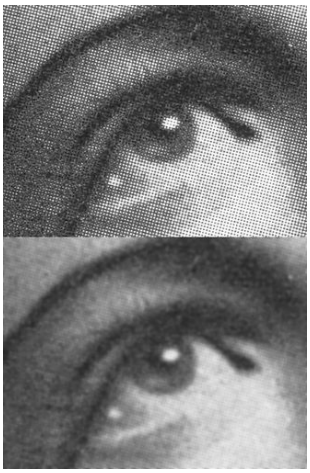

Figure 1. Blur Filter

3· **Sobel Filter** Apply Sobel filter on pixels as seen on Figure 2.

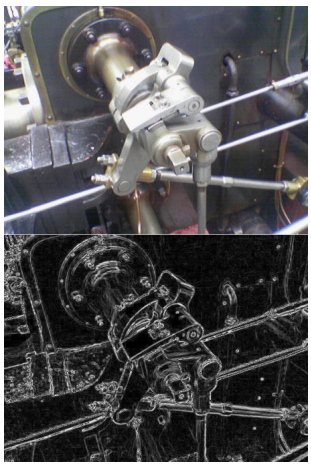

Figure 2. Sobel Filter

2.1.3. Export image. The inverse of the first part (section 2.1.1). Converts the array of pixels and creates a gif file which is then stored into a folder.

3. Parallel Optimization Languages

3.1. MPI

MPI stands for Message Passing Interface, is a standardized and portable message-passing system to function on a wide variety of parallel computing architectures. MPI takes each worker as a process and standardizes the message passing operation between them.

3.1.1. Treating each gif image separately. Each gif image can have several images saved together. The core of this optimization method was to make each worker to treat each image separately in order to make the process faster. Two different ways of organizing the work distribution can be applied:

1) Static Work Distribution
2) Dynamic Work Distribution

In the first case, the Static Work Distribution just gives each worker a different but precomputed image. For example, should there be ”n” images and ”P” processes, we can make a rule that follows the following equation:

$$task \equiv n \mod P \tag{1}$$

Where task will be the number of the process that must work on the image number *n*. In this case, each process will work in their precomputed images and they will be always the same in each run of the code no matter how much time each process takes with each image.

In the case of the dynamic distribution, an initial work (in this case image) is given to each thread, the first one to finish will be given the next work (next image). It is important to notice that in order to manage this distribution, one thread must be used and so it will not be able to do the

computation, this is called overhead. For example, if there are three workers and three images, one worker will give the first two images to the remaining two workers, and the first of them to finish will be given the next image to work on.

Both strategies may be useful depending on some variables. For example, if the different images inside the gif file are very different from each other, it may be possible that using dynamic work distribution will make the most of the resources. If the nodes or threads have very different computation power, for example, they are run in different computers with very different processors and specifications, then dynamic work distribution may also be more effective in this case. Both cases named before are based on that the time each threads takes to work is very different from each other, the more difference there is with each thread execution time, the more effective it will be to use dynamic work distribution. If, on the other hand, the computational power of the threads are similar and all the images of a gif image are similar so that is not expected that each thread will take a great difference of time to make the computation, then it in necessary to look for other variables that may make a difference between each method.

The negative impact of the dynamic work case will be dependent on the amount of tasks used for the computation. With three MPI threads, the loss of one task will make a great negative impact as there will be only two tasks to work on each image instead of three. To go to extremes, with only two tasks, applying dynamic distribution will result in a serial work and no optimization will be done. The more MPI tasks are used, the less impact the overhead will have over the code. As a general rule, the less MPI tasks, more convenient it will be to use static work distribution, whereas the more MPI task, better for dynamic distribution as long as there are no more tasks than images of the gif. Should there be more (at least one) MPI tasks than images of the gif file, then the loss of one task to manage the work distribution will not affect the general performance of the code and there will be no relevant difference between both work distributions as each task will manage one image while others will not be used.

The final decision will be a trade off between the difference of time each task takes to do the job and the amount of available tasks, taking also into account the number of images in the gif file to be filtered. This variables however cannot be known for sure beforehand as the intention is to optimize the work for any gif image in general.

It was chosen to make a Static no only because it was an easier implementation, but also because gif images are supposed to have all images similar to each other. Also, each node is supposed to be run at least for our case in similar hardware, for that reason, the time taken by each process to filter each image will be almost the same, and using a dynamic distribution won't be of much help. Also, all the remaining MPI tasks can be used to speed up the computation time treating each image as explained on section 3.1.2 so that the more MPI tasks I have the better. Of course, this approach will not be so should each image of the gif file be very different to each other.

In this part, each thread loads only the image on which is going to work on, an leaves the other images (allocated) but without initializing the values. Because the work distribution is static, each thread will always work in the same image on each filter and stage of the program. Just before storing the gif image into memory, a root thread gathers all the information from the other threads. It makes it in no particular order to be able to be a little bit faster as no thread must wait for another to send the information before it can send it.

3.1.2. Treating subparts of each image. A master task splits the input images in subpart and sends each subpart to a slave task, which compute the blur filter. After the work, the master task merges each part of the original image and finishes the blur filter iteration.

This work can be done as long as the difference between the input image and the blured image is bigger than a predefined threshold.

The biggest problem of this solution is to split the input image in subparts with a overlap area. To do that, we extract the two areas to blur, the top and the bottom, and we try to split each area with rectangles "as squares as possible" : we assume we have just one task for this region, then if we have enough tasks we add one task in each row if the width is greater than the height, else we add one task in each column. Then we adjust the width and the height of each tile to cover the entire area to blur. Each task wrap its part of the image in a pixel array which contain the overlaps area in each direction.

The blur algorithm needs now to update the overlaps area of each task's neighbors. To do that, we apply two exchange between to tasks in each direction : top $\leftrightarrow$ bottom, left $\leftrightarrow$ right, top left $\leftrightarrow$ bottom right, top right $\leftrightarrow$ bottom left.

3.1.3. Results. When using MPI to treat each image of the gif by each task, there was no increase on performance when the gif has only one image as expected.

The test was run over seven different gif files that contain more than one image each. The average time to apply for example the blur filter with the original sequential program was 8.1 seconds. This time could be taken down to a minimum of 1.1 seconds. Values were between that range depending on how many MPI tasks were chosen.

Running this code in a gif with 20 high resolution images, the blur filter time was taken down from 4 seconds to 0.4 seconds, which means that the MPI code takes 10% of the time the sequential filter uses. Using more than 20 tasks takes almost the same time as 20 which is expected as the tasks 21 and higher will not work at all and will only consume communication time. The loading of the image however, was not so much improved, taken the time from 0.36 to 0.3 seconds.

When using MPI to treat a splitting part of images by each task, we obtain performances which depends of the size of images in input, the number of tasks MPI and the number of nodes used.

Running this code on the test set given with the sequential version of the program, we obtain the best performance with 8 tasks MPI on 1 node, with an average of coefficient on images between 0.25 and 0.3. But when running it on 2 nodes, we can obtain an average of coefficient on images bigger than 1, which means that is worse than the sequential version. This is because for small images, communications take more time than the computation. To reduce this bad effect, we decide to set a minimum size of each tile of the splitting at the size of the overlaps area.

We can however notice the computation time for the blur filter of all this set is smaller than the sequential version, from 35 seconds to about 5 seconds.

3.2. OpenMP

OpenMP stands for Open Multi-Processing. It is an application programming interface (API) that supports milti-platform shared memory multiprocessing programming in C, C++ and Fortran on most platforms, processor architectures and operating systems. OpenMP provides a portable, scalable model for developers of shared memory parallel applications. It consist of a set of three primary API components:

1· Compiler Directives
2· Runtime Library Routines
3· Environmental Variables

OpenMP comes included in gcc compiler so that only a flag is needed for gcc to compile the Parallel code made with OpenMP.

3.2.1. Treating subparts of each image. As in 3.1.2, the goal of this part was to implement a program that divides the image into sections and each thread treats each section. The approach was different than on the MPI implementation. Here, it was not the main which was treated but every specific filter function.

A static schedule was implemented so that similar loops (for example going throught each pixel of the image) was done by the same core and so it was possible to use as less barriers as possible.

3.2.2. Results. The results were quite effective. Running the algorithm through many different gif images, the Blur Filter Time was reduced by around 75% . The Sobel Filter Time was reduced around 60% and the Gray Filter was reduced a 30%. This results are taken by adding all the different times together and comparing both results of the original program and the OpenMP implementation.

3.3. CUDA

CUDA is a parallel computing platform and API created by Nvidia. It uses the Graphics Processing Unit (GPU) for general purpose processing (GPGPU). The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computation elements, for the execution of kernels.

CUDA is designed to work with C, C++ and Fortran. For more information on how to implement it with C please refer to [4].

Throughout history, GPU's evolved to into a highly parallel multi-core systems allowing very efficient manipulation of large blocks of data, which normally makes computation faster than a CPU when large processing of data can be done in parallel.

3.3.1. CUDA vs OpenMP. The difference between the CPU and the GPU's computation power is that, as seen in figure 3, the CPU has less cores than the GPU, so that it can do less operations in parallel. However, the CPU cores are faster to do complex operations as branches and loops.

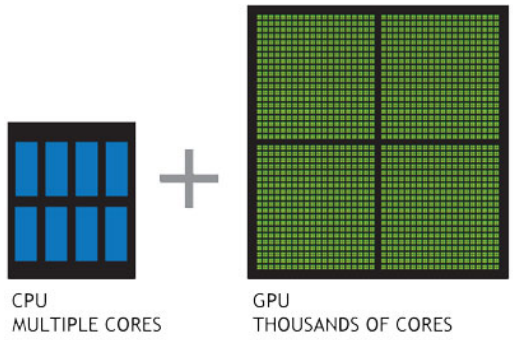

Figure 3. CPU and GPU comparisson

It was difficult to implement both CUDA and OpenMP at the same time. The loops could be divided between both so that, for example the GPU works in one half of the image while the CPU on the other half. We could go even further and check the amount of time each one takes and if, for example GPU does the computations twice as fast, then process the 2/3 of the image with the GPU and 1/3 with the CPU. Of course if that is to be implemented it will depend a lot on the GPU and CPU that is used to run the program so it will be difficult to implement it for a general case.

MPI can also be used to run two tasks on one node and use OpenMP for one thread and CUDA for the other node but this case will only make the programming easier whereas the real performance will be the same.

Running both CUDA and OpenMP was too difficult to implement and because of limited amount of time it was decided to implement only one of them at each time.

3.3.2. Results. It was expected, as image processing is normally very efficient with GPU processing, to obtain better results with the CUDA implementation. However, this is not what happened. OpenMP obtained almost 4 times better performance, taking less than 6 seconds to do something CUDA's implementation did between 19 and 20 seconds. This was probably do to a very complex operations for

the cores of the GPU, less computation should have been applied to each of the processors.

4. Final program

Figure 4 shows the sequential work of the project representing the creation and merging of each branch discussed previously on section 1.1. The left part of the graph represent an early period in time while the right part is the completion of the actual project code.

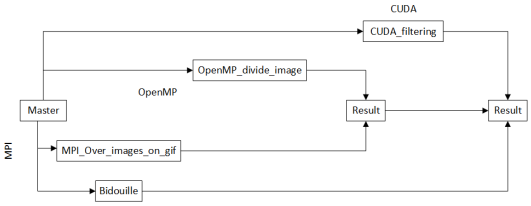

Figure 4. Git Project Flow Diagram

As it can be seen in the figure, the first approach was to implement the MPI code with both strategies discussed on 3.1. The MPI part over each image separately took less time because of less complexity and as soon as it was ready, the OpenMP (3.2) code was implemented. As soon as this OpenMP was ready, it was merged with the MPI part. In this merge, MPI treated each image in each different node while OpenMP divides the work for that image within each core. This method had two inconveniences, first of all, should the gif file have less images that the MPI threads, some of those threads will be unused. Also, the GPU is not used at all to help in the optimization process.

When running the sequential code, the blur part takes around 70% of the time. For that reason, the blur filter was taken as a priority and the optimization methods were focused on this part alone. The following explanation of the code was only done for the blur filter, which means that outside the blur filter, the final code works as recently explained.

Before continuing with the explanation of the final code, it will be important to pay attention to the hardware itself. There are a number of computers (nodes), each with it's own CPU and GPU. Only MPI can be used to take advantage of each node so it was used to do exactly that. MPI was used to take advantage of any number of nodes or computers possible. Because it was the aim of the project to try to apply as many languages as possible, MPI was not used or intended to be used to run on the different CPU cores, OpenMP was used for this purpose, which actually obtained better results than MPI, probably because the communication between threads is done in a more efficient way automatically by the compiler. This will be important to understand so that MPI is not run with many processes in the same node. Should that happen, the threads made by MPI within one node will "fight" for the resources inside the computer and will result in a loss of efficiency, although the code should work anyway.

The MPI application was made by a combination of both sections 3.1.1 and 3.1.2. First, MPI uses each thread to work on each image of the gif. Should the threads be more than needed, sub-groups will be created to treat each image with the technique explained in section 3.1.2. If a gif image has for example 2 images, and three nodes are used, then node 0 and 2 will be used to treat the first image while node 1 will work with the second image alone.

This application of both techniques of MPI, however, was only applied to the blur filter alone for the reason stated before. On the blur filter, another step was added. The use of the GPU to make the processing.

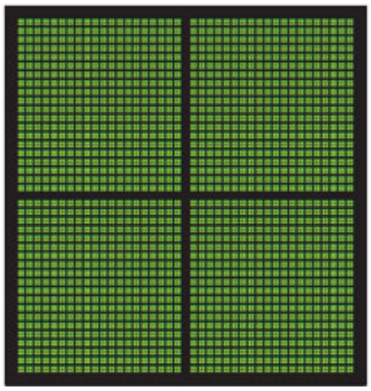

Figure 5. GPU Architecture

The CUDA kernel was done with a three dimensional block, a two dimensional block for the image itself to make it easy to compute the row and column of the treated image and a third dimension was added to treat each RGB (Red Green Blue) color. In order to make use of the multiprocessors, a two dimensional grid was made as shown in 5. In such figure one color architecture is shown. A 2x2 grid is represented with a two dimensional size blocks which corresponds to one of the three colors. Each small green square of the image is a thread or (in the best case scenatio) a GPU core. The row and column of the image to be filtered is a combination of the grid and block index. Whereas the color is given by the z index of each block.

The blur filter makes use of the CUDA kernel to process each image as explained on 3.3. It is important to remark that as it was said before, running many MPI threads in one node will make one thread have to wait till the other thread finish using the GPU and will affect the final performance.

5. Conclusion

Many tools for parallel programming were implemented and tested. It was possible to apply three different API's and compare the strengths and weaknesses of each one.

A sequential code of image filtering was parallelized combining MPI, OpenMP and CUDA to reduce the computation time nearly three times. Even more can be achieved by applying the strategy used for the blur filter in the rest of

the code and by improving the performance on the CUDA code which was less than expected.

References

[1] P. C. . F. Trahay. (2016) Inf560 evaluation. [Online]. Available: https://www.enseignement.polytechnique.fr/profs/informatique/Francois.Trahay/www/T projects-0.html

[2] NEGU93. (2017, mar) Parallel image filtering. [Online]. Available: https://github.com/NEGU93/Parallel-Image-FIltering

[3] I. S. LLC. (1999) Convert, edit, or compose bitmap images. image magick. [Online]. Available: https://www.imagemagick.org/script/index.php

[4] *CUDA C Programming Guide*.