Impact of parallelization technologies on an optical flow-based motion interpolation algorithm

Juan Sebastián Rodríguez Castellanos
Sistemas distribuidos
Departamento de Sistemas e Industrial
Universidad Nacional de Colombia
Bogotá - Colombia
juarodriguezc@unal.edu.co
GitHub account

***Abstract*—One of the applications of video processing is the Motion interpolation or the motion-compensated frame interpolation (MCFI), in these algorithms a pair of frames are used to generate an intermediate frame. This paper presents the implementation of a frame interpolation algorithm based on optical flow using different parallelization technologies (CPU, GPU, MPI). The algorithm used is presented in the document. Since the calculation of the Optical Flow is a process that requires a high computational capacity but can be executed in parallel, the impact of these technologies on the execution time will be analyzed. Different tests were performed, resulting in a speedup greater than 3 using CPU and above 34 using GPU with the implementation in CUDA. However, in all cases the execution times were considerably high, allowing only the CUDA implementation to interpolate an average of 3 frames per second for high video resolutions.**

Keywords: Motion interpolation, Optical flow, CPU, GPU, Speedup.

I. Introduction

Motion interpolation is a form of video processing with numerous applications, this type of algorithm is widely used by different technology companies to provide the effect of "smoothing" on different audiovisual content. The goal is to create an intermediate frame or number of frames between existing frames [1]. There are two main types of motion interpolation algorithms, in the first type all frames to be interpolated are known, in the second type only the current and previous frames are known.

There are many uses for this type of algorithm. In most current TVs there is software that produces the effect of "smoothing" on video, to achieve this effect the frame rate is increased to 60 or more Frames, depending on the frequency of the screen. Another of the main uses of video interpolation is in Virtual Reality (VR), being used by companies such as Oculus, Microsoft and Steam for their VR headsets [1].

In this paper an optical flow-based motion interpolation algorithm is presented which can interpolate frames using only the current and previous frames. To generate the intermediate frame, the Optical Flow is calculated, then linear interpolation is applied using the information obtained from the Optical Flow and finally a Gaussian Blur is applied to the areas with the highest motion.

The calculation of the Optical Flow is the most important task for the algorithm developed, the objective of the optical flow is to predict the pixel-level motion between video frames [2]. Classical approaches have been superseded by learning-based methods [2], however, for the development of the algorithm, a brute force approach is used to calculate the motion vectors, called area correlation optical flow [6].

II. Related work

1) Optical Flow:
The optical flow is the pattern of apparent motion of image objects between two consecutive frames caused by the movement of an object or camera [3]. It is a 2D vector where each vector is a displacement vector showing the movement of points from first frame to second [3]. To calculate this vector it is necessary to use the intensity or luminance of each pixel [4]. Consider a pixel $I(x, y, t)$ in the first frame, it moves by distance $(\Delta x, \Delta y)$ in the next frame after Δt time, so:

$$I_1(x, y, t) = I_2(x + \Delta x, y + \Delta y, t + \Delta t) \tag{1}$$

If we apply Taylor Series, we get:

$$I_2(x+\Delta x, y+\Delta y, t+\Delta t) = I_1(x, y, t)+\frac{\partial I}{\partial x}\frac{\Delta x}{\Delta t}+\frac{\partial I}{\partial y}\frac{\Delta y}{\Delta t}+\frac{\partial I}{\partial t} \tag{2}$$

Which results:

$$I_x u + I_y v + I_t = 0 \tag{3}$$

Where u and v represent the velocity of x and y respectively and $I_x = \frac{\partial I}{\partial x}$, $I_y = \frac{\partial I}{\partial y}$, $I_t = \frac{\partial I}{\partial t}$ are partial derivatives of pixel function of coordinates x and y.

An example of the optical flow calculation is shown in Figure 1.

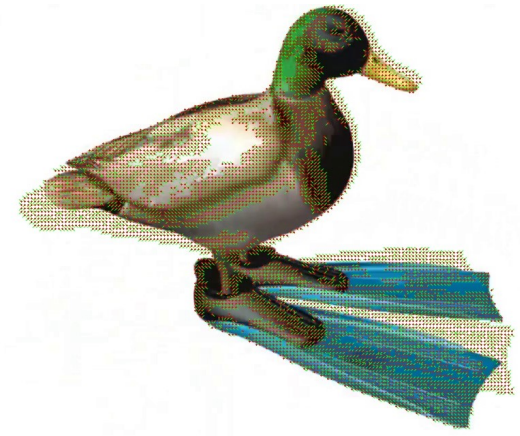

Fig. 1: Example of the Optical Flow

The optical flow algorithm is widely used in different fields and has different applications in the technological world, either for video interpolation, to virtually increase the resolution of images, for object detection in special cameras, among others.

2) Luminance:

To calculate the Optical Flow it is necessary to use the intensity of each pixel. Intensity or luminance is a measure to describe the perceived brightness of a color [5], this value can be calculated using the following formula:

$$L = 0.2987R + 0.5870G + 0.1140B \tag{4}$$

Values for R, G and B must be between 0 and 1.

3) Gaussian Blur:

The calculation of the optical flow can produce some noise, so the interpolated image can have some errors. To fix this, a Gaussian Blur filter is used. Gaussian blur is the application of a mathematical function to an image in order to blur it [7]. This type of filter is widely used by designers and photographers to reduce the noise that an image may have.

In order to apply the Gaussian filter, convolution will be used. The kernel is a group of pixels that moves along with the pixel being worked on by the filter. The width and height of the kernel must be an odd number. The kernel used to apply the filter is the following:

1/16	1/8	1/16
1/8	1/4	1/8
1/16	1/8	1/16

Fig. 2: Matrix used for the filter

Which is a 3 x 3 matrix that, due to its values, allows the application of the Blur filter on the image.

4) Linear Interpolation:

Interpolation is a statistical method by which related known values are used to estimate an unknown value or set of values [1]. For the current project, a linear interpolation modification is used, which is a type of interpolation achieved by geometrically representing a straight line between two adjacent points on a graph or plan.

If the two known points are given by the coordinates x_0, y_0, x_1, y_1, the linear interpolant is the straight line between these points [12].

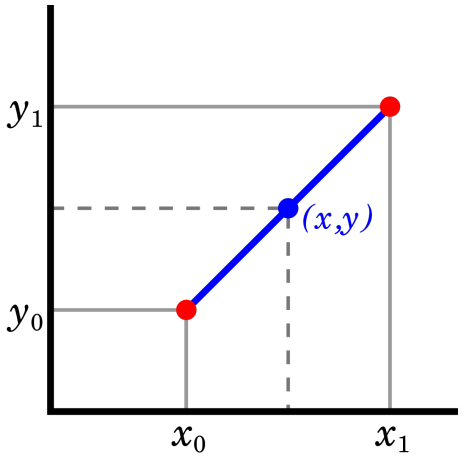

Fig. 3: Example of the Optical Flow

The motion tween algorithm uses two adjacent frames to generate an interpolated intermediate frame. The objective of this algorithm is to provide the smoothing effect, by doubling the number of frames that the video has.

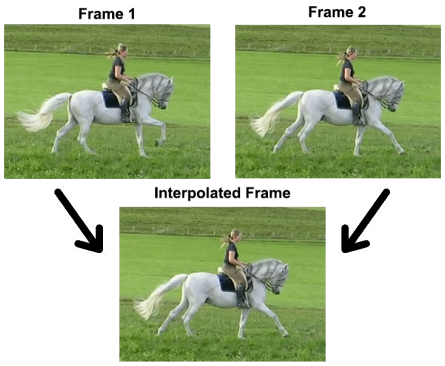

Fig. 4: Example of the Optical Flow

III. Algorithm Designed

The pseudocode of the algorithm designed is presented below:

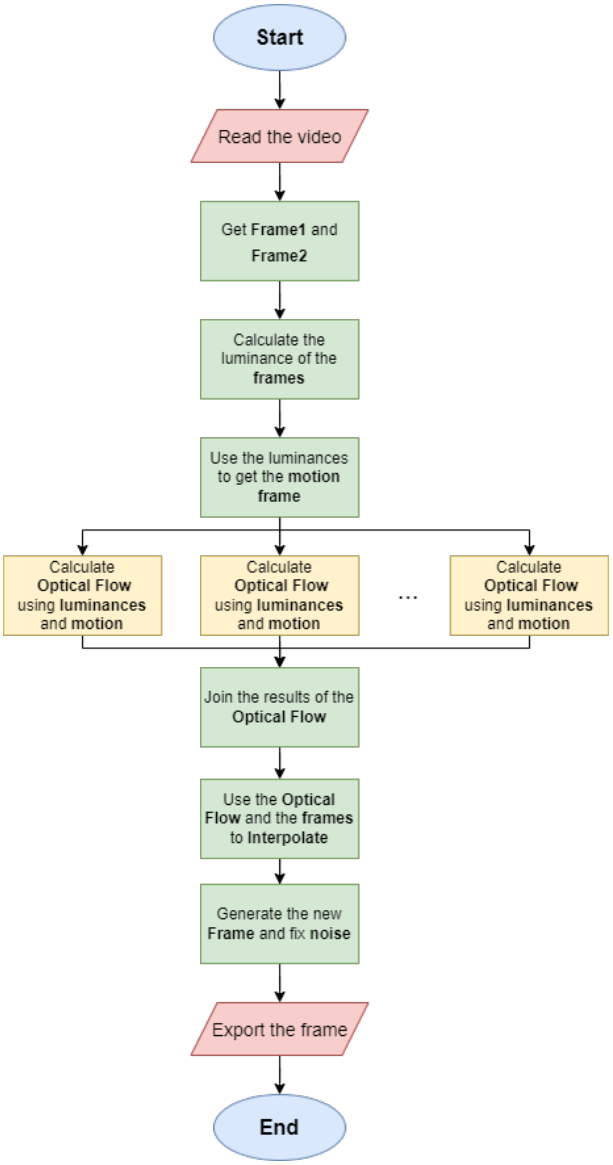

Fig. 5: Pseudocode of the algorithm

The diagram presents seven main processes, which are responsible for performing the motion interpolation using a pair of frames:

Get the frames: In this process, a couple of frames of the loaded video are obtained, for this the openCV library is used. In order to increase compatibility with the different technologies used, the conversion of the "Mat" data type to a flattened matrix of the "Uchar" type was performed.

Calculate the luminance: Each pixel of the loaded frames is calculated for its luminance, to do this equation 4 is used.

Get the motion frame: Using the previously calculated luminances, a motion frame is created. For this, the absolute value of the difference in luminances is calculated.

Get the Optical Flow: With the luminances and the movement frame created, the "Optical Flow" of the frames is calculated. This is the most important process of the designed algorithm, since it is the one that requires the greatest computational capacity. However, this process has the quality of being quite parallelizable.

Interpolate the frames: Using the vectors x and y of each pixel of the optical flow, two frames are generated using information from the original frames. To generate these frames, the midpoint of the vector obtained is used. Finally, the generated frames are joined to create the resulting frame.

Fix noise of the frame: Because the generated frame may have some imperfections, the Gaussian blur filter is used to make some corrections. Once this process is done, the generated frame is exported.

IV. Materials and Methods

The videos and methods used are described below. In addition, the form of parallelization carried out in the methods used is explained.

1) Videos:

The following videos were used to perform the tests of the algorithm:

Video 1

The first video is used to test the performance of the algorithm with high motion situations.

Video resolution: 1280 x 720
Frame rate: 24
Video length: 6 seconds
Total frames: 144

Fig. 6: Video 1 for the experiment

Video 2 The second video presents less movement compared to the first, however, it is expected to evaluate the results of a low framerate video.

Video resolution: 1280 x 720
Frame rate: 15
Video length: 6 seconds
Total frames: 90

Fig. 7: Video 2 for the experiment

2) Parallelization method:

As shown in figure 5, the only process that will be carried out in parallel is the calculation of the Optical Flow. For this reason, it is necessary to explain how it works. In the first instance, the Optical Flow calculation can be performed in parallel, since this calculation is performed bit by bit, without its result depending on the calculation of the other bits. The Optical Flow calculation is performed as shown in the following figure:

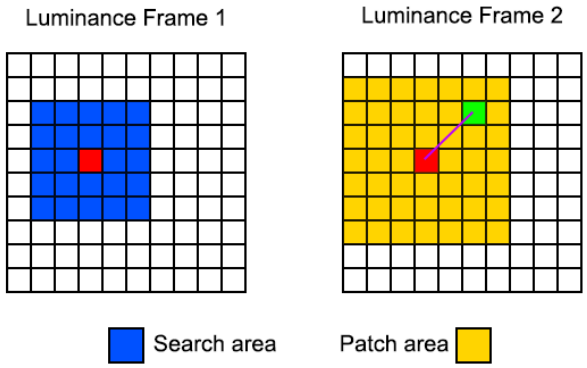

Fig. 8: Optical Flow calculation

The luminances of two adjacent frames are used to iterate over a search area around the pixel to be computed. On each of these search pixels, a comparison of the intensities around a patch is performed. The resulting pixel is chosen as the one with the smallest luminance difference on the patch. This process is carried out for each one of the pixels of the image, therefore, depending on the size of the search area, the execution time will vary, increasing as the area increases.

However, this task can be split to be performed in parallel. This is because it depends only on the luminances of each frame. The implemented parallelization process is shown below:

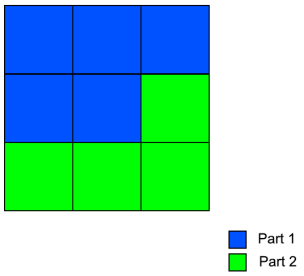

Fig. 9: Method to split a frame

Dividing the image into an amount N of equal or similar size parts, which can be processed independently.

3) Technologies:

OpenCV: (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products [13]. For the development of the project, openCV was mainly used for reading and writing videos.

OpenMP: (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran [14]. Using this API, the first type of parallelization (CPU) will be carried out.

CUDA: CUDA is a parallel computing platform and programming model created by NVIDIA. With more than 20 million downloads to date, CUDA helps developers speed up their applications by harnessing the power of GPU accelerators [15]. This technology is used to perform the parallelization of the algorithm using the GPU.

MPI: Message Passing Interface (MPI) is a subroutine or a library for passing messages between processes in a distributed memory model. MPI is not a programming language. MPI is a programming model that is widely used for parallel programming in a cluster. [16]. This is the last type of parallelization to be performed. However, as part of the experiments, a hybrid MPI + OpenMP will be developed and the existence of improvements in the execution time will be analyzed.

V. Experiments

To evaluate the developed algorithm and the impact of the different parallelization technologies, three criteria will be used:

1) Average number of generated frames per second (FPS)
2) Total execution time environment.
3) Speedup

The two videos shown in the previous section will be used for the experiments, carrying out tests varying the number of threads / processes and analyzing the impact of these on the previous criteria.

1) Computer details:

Each of the technologies requires a specific type of hardware for its operation, for the experiments performed the following characteristics were used.

Sequential, OpenMP:
CPU - Intel Xeon (8) @ 2.2GHz
Memory: 8GB
OS: Ubuntu 20.05.5 LTS

CUDA:
GPU: Tesla T4
CUDA Driver Version: 11.2
Multiprocessors: 40
CUDA Cores / MP: 64
Global memory: 16GB
OS: Ubuntu 18.04

MPI: 6 Nodes with
CPU - Intel Xeon (4) @ 2.2GHz
Memory: 4GB
OS: Ubuntu 20.05.5 LTS

VI. Results

The graphs of the results obtained when performing test # 1 are shown below:

*1) **Sequential**:* When executing the interpolation algorithm on video 1, the graph with the execution times for each frame is the following:

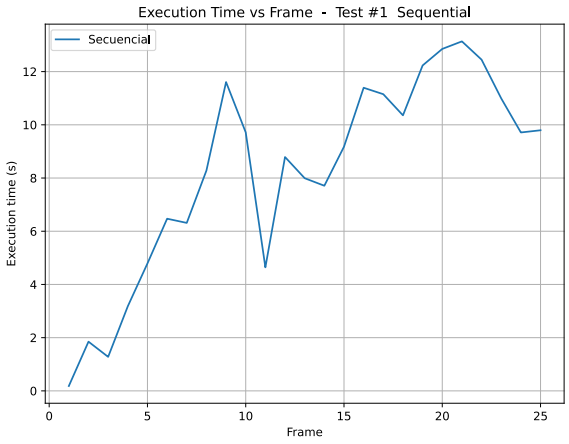

Fig. 10: Execution time by frame - Sequential

The graph shows that between 4 and 12 seconds are required to generate each frame. This is because the pair of frames used to interpolate that frame presents a high degree of movement, making the OpticalFlow calculation take more time. This graph can also represent the number of frames generated per second. These values are listed below:

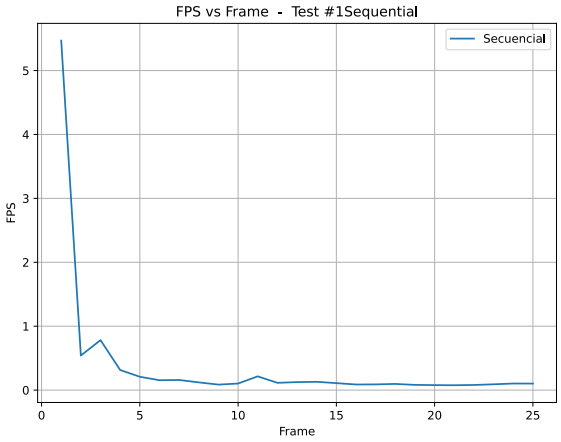

Fig. 11: FPS by frame - Sequential

It is possible to see how the number of frames generated per second ranges between 0.05 and 0.4.

*2) **OpenMP**:* Below are the results obtained by using openMP to parallelize the algorithm. For this, the threads were varied between 1 and 32.
The graph with the execution time for each frame is as follows:

Fig. 12: Execution time by frame - OpenMP

This graph allows us to appreciate how the curve tends to flatten as the number of threads increases. When using 32 threads, the time required to generate a frame varies between 1 and 3 seconds.
Seeing how frames require less time to be generated when

using a larger number of threads, now it is necessary to obtain the number of frames generated per second when more threads are used.

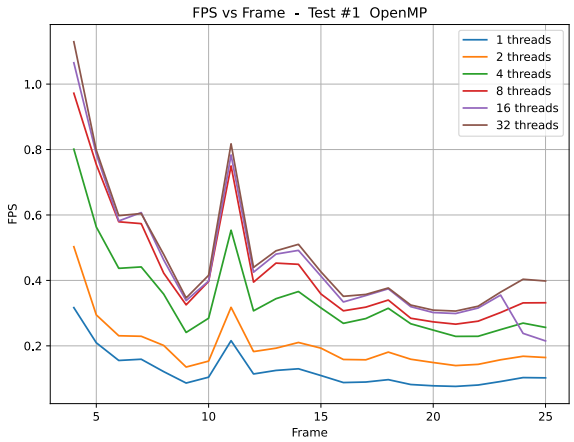

Fig. 13: FPS by frame - OpenMP

It can be seen that when using 32 threads the number of frames per second varies between 0.3 and 0.8. However, this value is not enough to analyze the impact of parallelization on the sequential algorithm. For this, the total execution time of interpolation will be analyzed, whose graph is shown below:

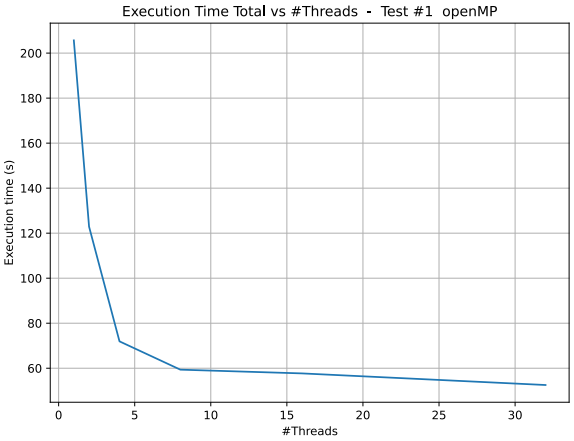

Fig. 14: Execution Time total vs #Threads - OpenMP

Sequentially, the total execution time exceeds 200 seconds, this value is considerably reduced as the number of threads increases, reaching an approximate value of 50 seconds when using 32 threads. Using these values it is possible to calculate the speedup obtained. The graph below shows this value as the number of threads increases.

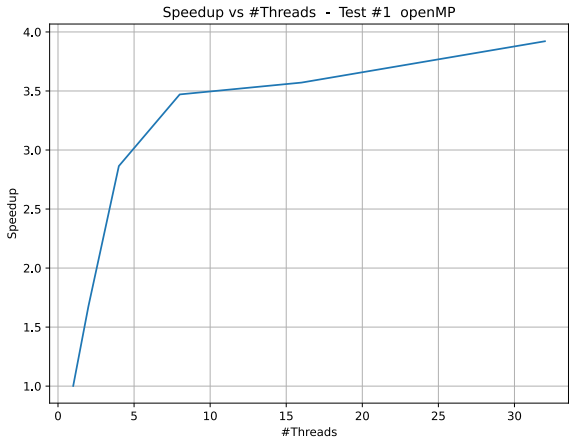

Fig. 15: Speedup vs #Threads - OpenMP

With a speedup of approximately 4 when using 32 threads.

*3) **CUDA**:* For parallelization using CUDA, the number of blocks and threads per block are varied. As mentioned in the Hardware description, there are 40 multiprocessors and 64 cores per block in the used GPU.
For the first test, only one block was used and the number of threads was increased to double its capacity (128 threads).

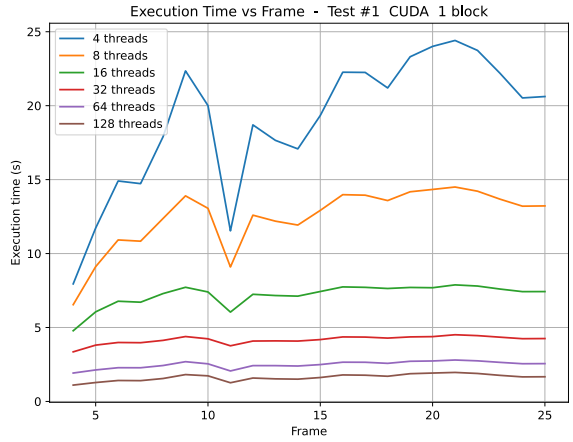

Fig. 16: Execution time by frame 1 block - CUDA

A fairly flattened graph can be seen when using 128 threads. Allowing interpolate frames between 1 and 3 seconds.

The next test performed was to use the same number of blocks as multiprocessors (40). This graph is presented below:

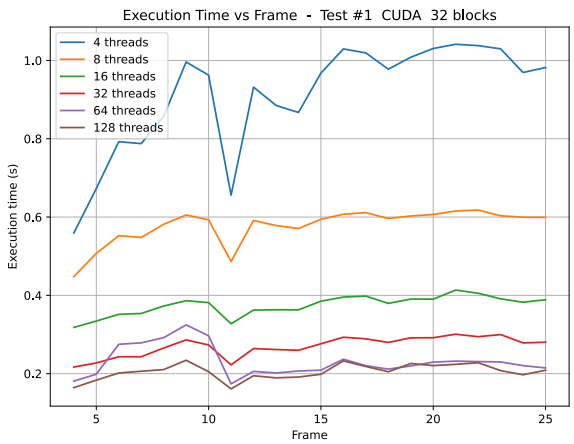

Fig. 17: Execution time by frame N blocks - CUDA

In the same way as in the previous test, the curve flattens as the number of threads increases. In this case, it is possible to generate frames between 0.1 and 0.3 seconds.

As a last test, the configuration recommended by Nvidia will be used, selecting double the number of multiprocessors available as the number of blocks (64 blocks).

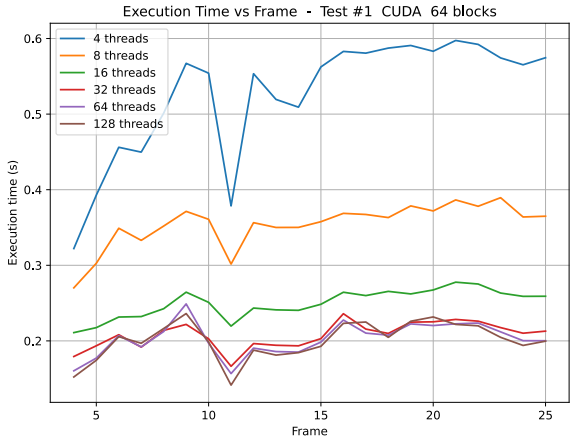

Fig. 18: Execution time by frame 2N blocks - CUDA

Obtaining results very similar to those obtained with 32 blocks.
The following graph shows the generation time of each frame keeping the number of threads at 128 and increasing the number of blocks up to 64.

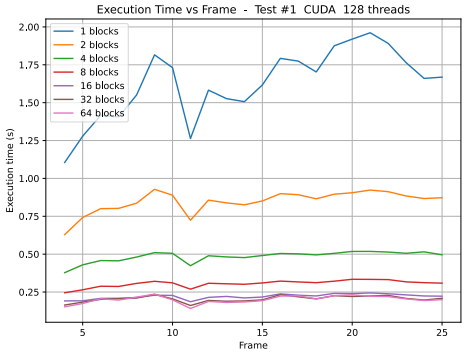

Fig. 19: Execution time by frame 2N threads - CUDA

This graphic allows us to observe how the configuration of 64 blocks and 128 threads allows the generation of frames in a shorter time. However, in order to appreciate this value in a better way, the following graph is presented:

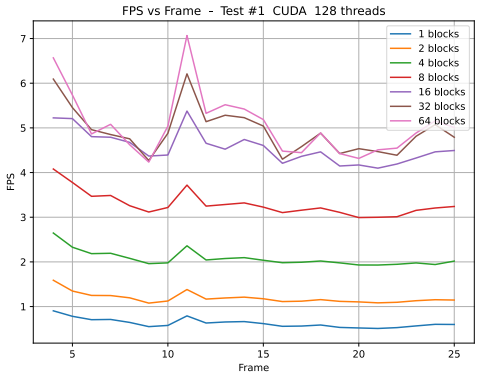

Fig. 20: FPS by frame - CUDA

Generating between 4.5 and 7 frames per second with the best configuration. The total time is shown below:

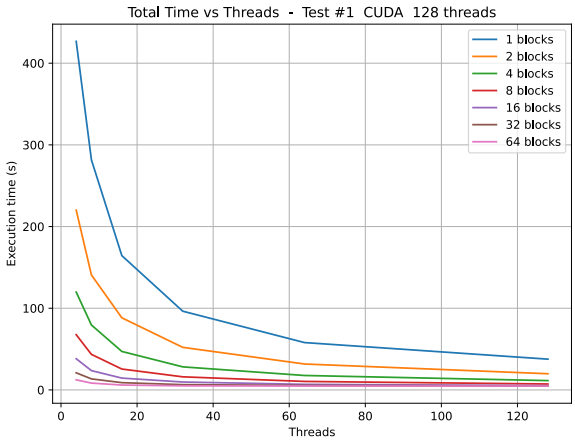

Fig. 21: Execution Time total vs #Threads - CUDA

This graph shows that as the number of blocks increases, the total time tends to 0. In this case, with 64 blocks, the total time is approximately 5 seconds.
The obtained speedup is:

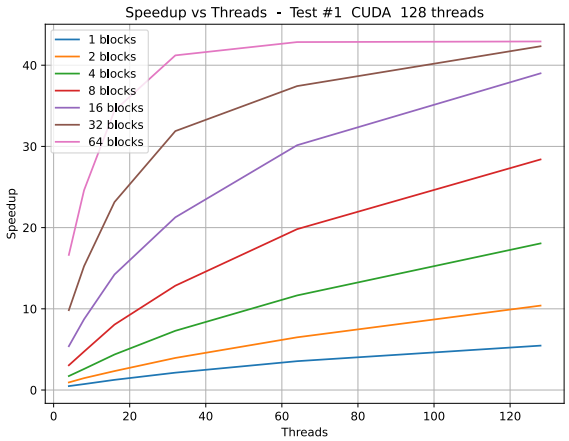

Fig. 22: Speedup vs #Threads - CUDA

Where it is possible to appreciate that the speedup is approximately 27.

4) MPI: For parallelization with MPI, processes are used, which are called from the master node. As previously described, there is a 6-node cluster that can run a total of 16 processes. The graph of the execution time per frame is the following:

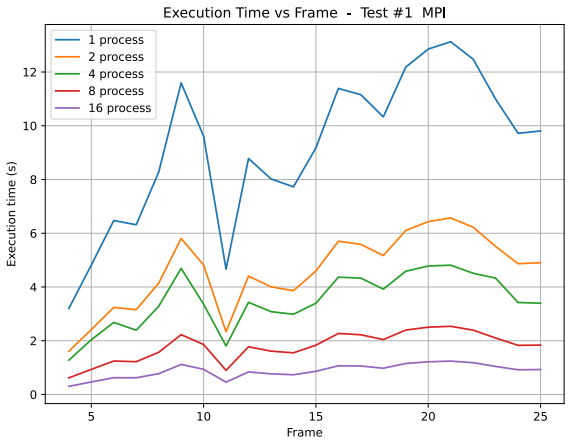

Fig. 23: Execution time by frame - MPI

Similar to the previous tests, as the number of processes increases, the curve with the execution time per frame seems to flatten. Interpolating a frame between 0.4 and 1.5 seconds. The graph with the Frames Generated per second is the following:

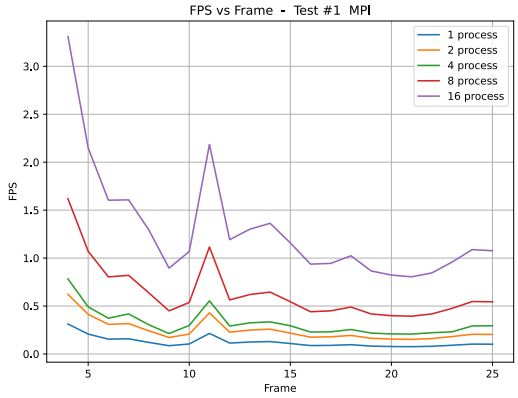

Fig. 24: FPS by frame - MPI

Generating from 0.7 to 2 frames per second when using 16 processes.
The graph with the total execution time is the following:

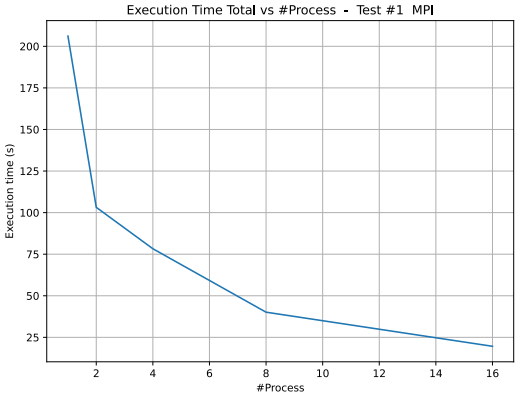

Fig. 25: Execution Time total vs #Process - MPI

Where you can see a considerable reduction in time. Reaching a value less than 25 seconds.

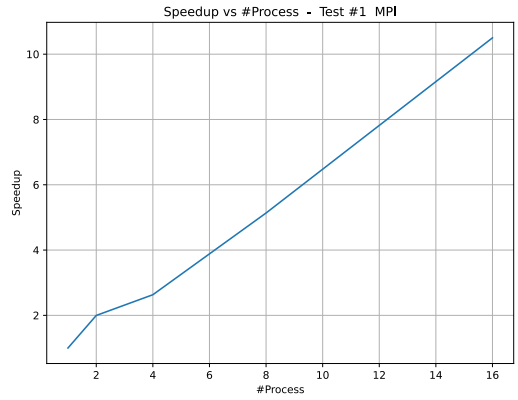

Fig. 26: Speedup vs #Process - MPI

Finally, it can be seen that the speedup achieved exceeds 10 units.

*5) **MPI + OpenMP**:* In this last test, a hybrid implementation between MPI and openMP is carried out, using a smaller number of processes but using thread parallelization to take advantage of the cores of each node.

The graph with the total time is shown below:

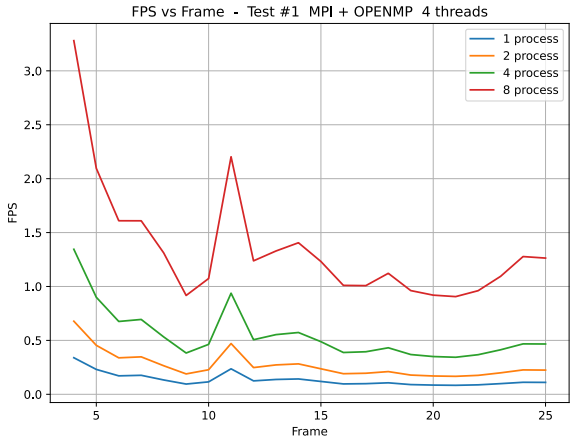

Fig. 27: FPS by frame - MPI + OMP

And the graph with the speedup is as follows:

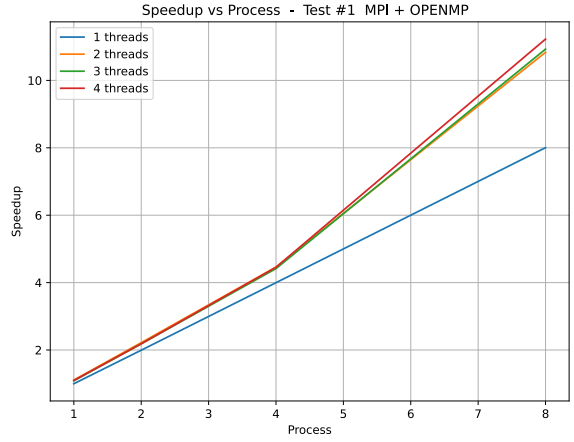

Fig. 28: Speedup vs #Process - MPI

Showing results slightly higher than those obtained only with MPI.

Once all the tests have been carried out, the most important values are gathered in a table, so that the impacts of each of the experiments can be visualized in a summarized way.

*6) **Table of results - Test #1**:* The results obtained in test 1 are:

TABLE I: Results Test 1

	Total time (s)	Average FPS	SpeedUp
Sequential	206.05	0.12	–
OpenMP	52.55	0.48	3.92
CUDA	5.21	4.80	42.92
MPI	19.62	1.27	10.5
MPI + OMP	18.35	1.36	11.22

*7) **Table of results - Test #2**:* The results obtained in test 2 are:

TABLE II: Results Test 2

	Total time (s)	Average FPS	SpeedUp
Sequential	123.9	0.20	–
OpenMP	33.28	0.75	3.72
CUDA	4.53	5.52	27.3
MPI	12.20	2.05	10.15
MPI + OMP	11.20	2.23	11.06

VII. Conclusion

In this work, a video interpolation algorithm was presented, whose objective is to generate an intermediate frame for each pair of frames of a video. This algorithm requires the calculation of the optical flow for its operation, which is a process that requires great computational capacity. When carrying out the tests with different parallelization technologies, the following conclusions were obtained:

1) The calculation of the Optical Flow with the chosen method requires great computational capacity, so it is highly recommended to use parallelization.
2) Using openMP for parallelization allows a certain improvement in the execution of the algorithm (approximately 4 of speedup), however it is not enough to interpolate frames in real time (0.48 Frames per second).
3) CUDA is overall the best parallelization technology for the interpolation algorithm. Having a speedup greater than 40 units.
4) The speedup achieved by using MPI is considerable (approximately 10), however it does not allow real-time interpolation of high-resolution videos (2 Frames per second).
5) The combination of MPI and openMP slightly improves the results, however these results could not be perceived when rendering in real time (2.05 - MPI and 2.23 OMP + MPI).
6) None of the technologies used allows the interpolation of high-resolution videos in real time. However, CUDA has the best results (5.5 FPS).
7) The algorithm could be applied for video calls and streaming if the resolution of the video to be interpolated was reduced.
8) Further research on video interpolation using deep learning is recommended. In this way you get better results.

Github repository:
https://github.com/juarodriguezc/Sistemas-distribuidos

REFERENCES

[1] Lyasheva, S., Rakhmankulov, R., Shleymovich, M. (2020b). Frame interpolation in video stream using optical flow methods. Journal of Physics: Conference Series, 1488(1), 012024. https://doi.org/10.1088/1742-6596/1488/1/012024

[2] Bai, S., Geng, Z. (n.d.). Deep Equilibrium Optical Flow Estimation. Peking University.

[3] OpenCV: Optical Flow. (n.d.-b). https://docs.opencv.org/3.4/d4/dee/tutorial_optical_flow.html

[4] https://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/OWENS/LECT12/node4.html

[5] (2020, April 29). Colour Luminance and Contrast Ratio. 101 Computing. https://www.101computing.net/colour-luminance-and-contrast-ratio/

[6] https://www.southampton.ac.uk/ msn/-book/new_demo/opticalFlow/: :text=Optical

[7] O'Donovan, P. (n.d.). Optical Flow: Techniques and Applications. The University of Saskatchewan.

[8] How to Use the Gaussian Blur Effect — Adobe Photoshop. (n.d.). https://www.adobe.com/creativecloud/photography/discover/gaussian-blur.html

[9] Optical Flow for Industrial Applications — wileyindustrynews.com. (n.d.). https://www.wileyindustrynews.com/en/whitepaper/optical-flow-industrial-applications

[10] Spatial Filters - Gaussian Smoothing. (n.d.). https://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm

[11] What Is Interpolation, and How Do Investors and Analysts Use It? (2022, November 21). Investopedia. https://www.investopedia.com/terms/i/interpolation.asp

[12] Wikipedia contributors. (2022, August 18). Linear interpolation. Wikipedia. https://en.wikipedia.org/wiki/Linear_interpolation

[13] OpenCV. (2020, November 4). About. https://opencv.org/about/

[14] Wikipedia contributors. (2022b, November 27). OpenMP. Wikipedia. https://en.wikipedia.org/wiki/OpenMP

[15] Oh, F. (2022, January 27). What Is CUDA — NVIDIA Official Blog. NVIDIA Blog. https://blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2/

[16] Introduction to MPI — Distributed Computing Fundamentals. (n.d.). http://selkie.macalester.edu/csinparallel/modules/DistributedMemoryProgramming/build/html/introMPI/introMPI.html