# High-Performance Computing Architectures Accelerating a Video Encoder through CUDA

*Group 14* - João Rodrigues, Martim Bento and Rodrigo Coimbra
*Under the orientation of:* Prof. Nuno Roma

***Abstract*—Motion estimation plays a crucial role in various video processing tasks, from compression to tracking. However, as it is a computationally heavy process, it can be demanding, time-wise. Leveraging the computational power of Graphics Processing Units (GPUs) can significantly accelerate motion estimation. This paper investigates the efficiency of GPU-accelerated motion estimation techniques on algorithms such as Full Search Block Matching (FSBM) and Step Search (SS). Through careful selection of which blocks to parallelize, which to leave to the CPU, how to minimze the overhead in memory transfers and kernel calls and how to configure the kernel grids and blocks, we validate our solutions, achieving a normalized gain of $10.66\times$ for the FSBM algorithm and of $0.851\times$ for the SS algorithm.**

**Our findings facilitate the development of optimized solutions for video processing applications.**

***Index Terms*—Motion Estimation, Video Processing, Single Instruction Multiple Data (SIMD), GPUs, CUDA Programming**

## I. Introduction

### A. Comparison of CPU and GPU architectures

CPU architectures focus on general-purpose processing, being composed by a moderate number of performant cores which provide the necessary throughput for sequential processing and multitasking. In contrast, GPU architectures are focused on SIMD processing, featuring many streaming multiprocessors (SMs), allowing the launch of single-purpose kernels, where all the launched threads perform the same operations to different blocks of data, in **parallel**.

As such, GPUs are especially useful when dealing with programs that perform the same operations to large amounts of data. However, there is an overhead when data is transferred between the CPU and the GPU, as well as when launching numerous threads. Therefore, it is preferable to use the GPU for functions that also have high arithmetic intensity, so that the overhead is counterbalanced.

### B. Summary of used Motion Estimation Algorithms

*1)* ***Full Search Block Matching****:* Full Search Block Matching (FSBM) involves comparing each block in a frame with blocks in the reference frame to find the best match. Our implemented approach uses the sum of absolute differences (SAD) as the parameter to minimise so to the find the best match between frames. FSBM provides accurate motion reconstructions, however it is computationally intensive due to its exhaustive search, making it a prime target for SIMD parallelization on GPUs to improve efficiency.

*2)* ***Step Search****:* The step-search algorithm is a sub-optimal search approach, which involves iteratively evaluating candidate search area blocks in a quasi-rectangular pattern around its central position, which itself is also updated on each iteration according to the local minima encountered. While computationally simpler than other techniques, step-search may not always find the optimal solution and can be prone to local minima. Nevertheless, it is favoured due to its efficiency, as the search is logarithmic in time.

## II. Methodology

### A. Full Search Block Matching

To paralelize any algorithm using GPUs we have to first identify which block of code can be turned into our main kernel.

Clearly, the current bottleneck of the program is the FSBM algorithm, which is composed of 4 nested for loops. The first two iterate over the determined area of the frame to be searched (search area), and the two inner loops calculate the sum of absolute differences (SAD) between the current image block and the current 32 by 32 block of the search area of the frame. For now, we choose to parallelize the inner loops, as it is the simplest to implement, while still allowing for a reasonable amount of threads to be launched ($1024$). However, the other approach will also be analysed afterwards, as it allows us to launch $128 \times 128 = 2^{14}$ threads ($16\times$ more).

To perform the parallelization, we turn the SAD function into a CUDA kernel and launch it with 32 blocks in the y axis, each block with 32 threads in the x axis. That way, we ensure that there are enough threads so that the absolute differences are computed completely in parallel and that the warps that they are assigned to are all full. Additionally, a reduction is performed on each of the 32 blocks, returning an array of 32 partial reductions, which is then computed on the CPU.

Of course, there is a series of other modifications that can be performed, to further optimise the code. For example, to remove some of the overhead from the memory transfers between host and device, we can:

- Allocate the search area and the 32 by 32 block space only once, and directly through `cudaMalloc()`, to avoid having to perform `cudaMemcpy()`. As a consequence, they are now allocated as arrays, instead of matrices.

- Instead of transferring two frames on each motion estimation and reconstruction iteration, we opt to transfer on the first iteration two frames, and for each following iteration transfer a new frame, using it as the new current frame while we switch the pointer of the reference frame to the previous iteration "current" frame which is still in the GPU memory.
- Following the two previous points, we can turn the `getBlock()` function into a CUDA Kernel, as it is somewhat computationally heavy and easily parallelizable. Likewise for the search area.
- Another way to mitigate such costly memory transfer overhead is through the use of pinned memory, which is always resident in RAM, being locked in place, and thus, more quickly accessible by both the CPU and GPU devices. This, however, will require more caution to work with since the available memory that can be pinned is limited, while also increasing the access time from the CPU or GPU to that block.

While GPUs excel on intense arithmetic operations, they might have their performance harmed with conditional operations such as if statements. So, unless we use conditions in cases where it does not limit the number of useful threads launched, we will want to remove the said conditional operations. The only way to minimise this is by moving such statements to CPU execution. The *if statement* present in the `SAD` function is needed in cases where the search area would cross the border of the frame. As such, a possible solution is to compute the border region in CPU while the middle is computed in the GPU. This, however, requires multiple checks on the search area and image blocks, and thus the reference and current frames as well. Due to how they are allocated, those checks, which require additional memory transfers and accesses will result in a slightly larger overhead. As such, it is wise to minimise the overhead from memory transfers first, with the solutions shown previously.

It is also important to note that, as the calculations in the CPU are much slower, the speedup in performance will ultimately be limited by it. To counteract this, we can try to use assyncronism between the CPU and GPU, through CUDA streams.

Two variations were devised to effectively leverage them. The first involves employing multiple streams to distribute the computation of the array of SAD results and asynchronously copying the data of these results to the CPU. When it finishes processing the portion of data that had already been copied, it synchronises with the next stream, if that one has not already finished. This method was devised due to the size of the array of SAD results ($2^{14} \times 4B \approx 65KB$) and the idle time of the CPU during these computations.

The second variation consists of implementing two additional kernels to calculate the *BestResult*. One kernel performs the reduction along the lines, followed by a second kernel for the final reduction of the values computed in the previous kernel. In this scenario, streams are employed for computing the SAD values and performing the reduction of a specific number of lines. Once all streams have completed their tasks, the last kernel is called, and an asynchronous copy of the *BestResult* is made, allowing the CPU to continue its processing.

One different approach would be to keep the *if statement* and, in cases where the search area would leave the border, place the maximum integer value in the result, to avoid influencing the best result. It would mean that less threads were actually useful, but it would allow for complete computation of the SAD inside the GPU. It would be almost as if we had increased the border with useless values.

Earlier, we decided in parallelizing the inner for loops, which are 32 by 32. However, the outer loops run as 128 by 128. Therefore, if we choose to parallelize the outer loops instead, we will be able to launch $16\times$ more threads, which, without taking the slightly bigger overhead into account, would result in a *relative* speedup of $16\times$. Of course, as each SAD would now be computed in parallel, updating the new best result would now be all done in the CPU.

**Algorithm 1** *Main GPU Kernel - FSBM*

sad $\leftarrow$ `int`; BSz $\leftarrow 32$; SSz $\leftarrow 2 \times 64 + 32$
IBlk: Image Block; SA: Search Area
posX, posY, z: calculated withthread and block IDs

```
1:  sad[z] ← 0
2:  if within borders then
3:      for 0 ≤ i < BSz do
4:          ImgIdx ← i × BSz + j
5:          SAIdx ← (posX + i) × SSz + (j + posY)
6:          sad[z] += |IBlk[ImgIdx] − SA[SAIdx]|
7:      end for
8:      warpReduce(sad, z)
9:      d_results[posY × 2 × 64 + posX] ← sad[0]
10: else
11:     d_results[posY × 2 × 64 + posX] ← 999999
12: end if
```

### B. Step Search

For the step-search optimisation we first analysed which of the previously used methodologies could be reused. In particular, most optimisations related with decreasing the memory transfer overhead between host and device can be preserved. As the number of iterations in the algorithm is much smaller than for the FSBM, it might be a good idea to parallelize the SAD function. This will further studied ahead.

Now, there are some aspects of the algorithm that we can use to our advantage. The step-search is composed of 6-iteration while loop, where, in each of them, the SAD function is called 8 times. We can parallelize this, such that all SAD calls are done in parallel. We launched them with a CUDA block of 3 by 3 by 1. Therefore, the SAD function is launched 9 times, with the ninth call being a dummy-result, a repeat of the center of the previous iteration. As such, it is not harmful to the result and it simplifies the kernel launch.

This strategy by itself barely uses the GPU resources however, and, as such, it will hardly optimise our program. There is not a immediate solution to this, as there are data dependencies between iterations, i.e. the center of the next iteration depends on the result of the previous iteration.

As such, there are two possible approaches that we can take to solve this problem. The first one is to compute all possibilities of the second iteration in parallel with the first iteration. The overhead between memory transfers will quickly increase with this method, but the number of required iterations will decrease from 6 to just 3. Additionally, since we will be analysing more possible cases, it is only natural that the final obtained residue becomes closer to the value obtained from an optimal solution such as FSBM. So, we can configure a CUDA block of 7 by 7 by 1 to perform two iterations in parallel, and we can even go to the extreme and configure a CUDA block of 13 by 13 by 1 to perform three in parallel (two iterations in total). Past this point, it is not worth testing higher granularities, as the overhead will increase immensely just to perform one less iteration of the algorithm, which would then, in practice, be equivalent to the FSBM algorithm. Of course, both the 7 by 7 and 13 by 13 implementations should be compared, as it is possible that the additional overhead does not compensate having to do 2 iterations instead of 3.

The second approach is to, instead, take full advantage of the fact that the grid is defined with three dimensions and launch threads in the z axis to parallelize one of the for loops of the SAD function. Just as with the FSBM algorithm, a shared memory array and a subsequent reduction are both required but we should still see some improvements. For this, we configure a grid of 1 by 3 by 3 blocks and configure each block with 32 threads in the x axis (32 by 1 by 1). Thus, we use the Threads in X for parallelizing the for loop, while using the threads in Y and Z for launching each SAD in parallel. The choice of using threads in X instead of Z comes from the fact that the maximum number of threads we can launch in the z axis is only 64. This should be enough to parallelize one of the for loops, but if we want to parallelize the other loop as well then we rapidly hit a limit. If we use either X or Y however, then, with our configuration, we can launch 256 threads. That way, each thread computes only 4 operations.

Finally, since both approaches are independent, we can merge them into one implementation that benefits from both ideas. That way, we use as many threads as possible, minimise the number of iterations of the loop and get a better residue value than the original implementation. Just as before, we can adjust the granularity to check which provides the best performance.

**Algorithm 2** *Optimized Step-Search Algorithm (13 by 13)*

```
1: grid(1, 13, 13); block(256, 1, 1)
   Distance ← 8
2: while Distance >= 1 do
3:     SAD_GPU <<< grid, block >>>
4:   for 0 ≤ i < 13^2 do
5:       if results[i] < bestResult then
6:           bestResult ← results[i]
7:           bestResult_x_y ← newX_Y[j]
8:       end if
9:   end for
10:    Distance >>= 3; CenterX_Y ← newX_Y[j]
11: end while
```

## III. Results

### A. Full Search Block Matching

The table below shows the performance of the first few strategies that we presented in the previous section, for the **first video**:

TABLE I
First Solutions' Execution Time

| Method | Time [s] | Speedup | Relative Speedup |
|---|---|---|---|
| CPU | 0.983 | - | - |
| Parallel SAD | 97.644 | 0.0101 | - |
| Less Memory Overhead | 82.564 | 0.012 | 1.19 |
| Borders on CPU | 18.894 | 0.052 | 4.33 |
| Parallel FSBM | 2.163 | 0.448 | 8.62 |

The optimisations we performed were built on top of previous ones and the order by which we applied them is the same as the one presented in the table. As such, our first optimisation was creating the cuda kernel for the `SAD` function. It served its purpose, as we got an implementation that makes use of the GPU. However, we can observe that its performance was terrible. There are a number of things to optimise, as shown in the previous section, but our next approach is to try and compute the borders on CPU and only the rest on the GPU, to remove the problematic *if statement*. As noted in the previous section, we first mitigate the memory transfer overhead. This speeds up our execution time by 10 seconds. Of course, it was not the main source of the terrible performance, but it still had a considerable effect.

Now, we implement our strategy of distributing the processing between CPU and GPU. The result, also presented in table I, does not make use of streams or any other asynchronous method. However, we still get a much better performance, as our program now runs $4.33\times$ faster than before. With this we learn two valuable pieces of information. First, this strategy is promising, as it still has a clear room for improvement. Second, the current bottleneck lies in the SAD kernel, with a significant portion of the speedup attributed to the extended duration taken by the SAD operation on the GPU, since part of the blocks are now processed on the CPU, and those have an identical execution time to the original CPU implementation. This is further consolidated through the use of `timespec` variables.

In fact, our original choice of parallelizing the SAD function was flawed. Of course, as stated before, we are launching $16\times$ less threads than what we could, if we chose to parallelize the outer loops instead. However, we did not take into account two potential problems that manifested in our implementation. The first is the fact that parallelizing SAD implies having a reduction at the end of the kernel call, which limits our speedup. The other flaw was the overhead associated with launching a kernel. By parallelizing SAD, we create a kernel that gets launched in each iteration of the FSBM, so $2^{14}$ times in total. As such, not only are we using less resources from the GPU and with a kernel call that translates to one arithmetic operation followed by a reduction, which already hinders our speedup, but we will also have a much larger total overhead.

As we can see, these flaws outweigh whatever speedup we get from the parallelization.

So, the next optimisation step is to transform the whole FSBM algorithm into a CUDA kernel, instead of just the SAD. This means that every thread will perform the SAD function, which implies that the arithmetic intensity is higher and that we don't need a reduction on the GPU anymore. On another hand, we will perform the calculation of the new best result sequentially, after every thread has computed the SAD, and thus, on the CPU.

We can then observe, as the last result presented in table I, that our program is now significantly faster than before, coming very close to a speedup for the first video. Of course, since this video is $416 \times 240$, most image blocks will be borders and thus, calculated on the CPU. However, on the other, larger videos, there is much more room to use the GPU.

Now, we analyse the performance of our better solutions, on the first **3** videos. The speedups presented are with respect to video 3:

TABLE II
BETTER SOLUTIONS' EXECUTION TIME

| Method | Video 1 [s] | Video 2 [s] | Video 3 [s] | Speedup |
|---|---|---|---|---|
| CPU | 0.983 | 21.232 | 318.128 | - |
| Parallel FSBM | 2.163 | 7.765 | 89.902 | 3.539 |
| Streams (SAD) | 1.104 | 13.051 | 188.017 | 1.692 |
| All on GPU | 1.63 | 4.561 | 39.324 | 8.090 |
| Streams (Result) | 1.048 | 10.216 | 144.444 | 2.202 |
| Pinned Memory | 0.3199 | 3.335 | 37.121 | 8.570 |
| Threads in Z | 0.2447 | 2.695 | 29.828 | 10.665 |

First, we take another look at our previous solution of parallelizing the outer loops. As previously expected, since the larger videos feature more room for the use of the GPU, we are able to achieve a speedup on those. However, taking into account that we are launching $2^{14}$ threads for a major portion of the program, it is a bit underwhelming that the speedup is less than, at least, one order of magnitude.

We can improve on our program if we use both the CPU and the GPU asynchronously. However, that implies performing the large reduction, which is *not* the bottleneck, in the GPU, so that the CPU is free to perform the computation for the other blocks. Overall, this strategy ends up being pretty limited.

Upon including the use of streams into our program we observed, surprisingly, that the speedups achieved are below the anticipated levels for videos with bigger resolutions. The performance of both variations was assessed with different numbers of streams. However, only the results of 2 and 4 streams are shown, as they were the best ones.

In the first variation, i.e. distributing the computation of the results array over multiple streams, increasing the number of streams yields marginal improvements in performance, with 4 streams showing a slightly better performance compared to 2 streams. However, the speedup achieved did not meet our expectations. This is likely do to the fact that processing the results array is far from being the bottleneck of the program, which we verified with `timespec` variables. As such, whatever speedup we may obtain is very limited, especially taking into account the computational overhead of launching multiple kernels. However, if the results array was larger, this strategy could become viable, as it would allow us to split the time of copying the data to the CPU, while running computation on the values already received.

Regarding the second variation (computing the SAD values with streams), increasing the number of streams yields similar results. Furthermore, the results were even worse than those of the first variation, indicating that the demanding reductions in the additional kernels, characterised by numerous conditional statements and synchronisation points (syncthreads), outweigh the benefits of parallelization.

Now, we try a different approach. Instead of distributing the workload between CPU and GPU, we perform all computations on the GPU, regardless of the image block being in the border or not. In practice, we will use an if statement inside the CUDA kernel that attributes either the result of the SAD, if the search area is fully inside the image or a really big integer value otherwise. Of course, this means that, in the borders, some search area segments might be discarded when, in the previous implementations they weren't, which might lead to some small errors. Of course, generally this effect is small. However, it still means that we are trading away some precision for performance, which is not ideal.

As we can observe, this trade off works in our favour, as we more than double the speedup.

We can further optimise this approach with pinned memory, as mentioned in the previous section.

Finally, we see that every thread still has much to compute. Up until now, our grid and block configurations consisted of a grid with blocks in the y axis, each with many threads in the x axis, which was determined experimentally as the most effective. Therefore, we are not making any use of threads in the z axis. As such, we will use them to parallelize the inner for loop of the SAD function. A small reduction will be needed, but aside from that and the additional overhead from having to launch more blocks, each thread will have less to compute. Thus, our new grid configuration is 128 by 128 CUDA blocks (in the x and y axis), each with 32 threads in the z axis. We make use of a shared memory array with 32 indices. This additional parallelization pays off, even with the additional overhead and reduction, and we achieve a speedup above 10.

We now switch our focus to the Step Search algorithm. The normalised gains will be analysed afterwards.

### *B. Step Search*

We experimented the proposed solutions. The results both for execution time and the accumulated residue (and potential losses) are presented in tables III and IV respectively. The presented speedups are with respect to video 3 and the presented prediction losses (or gains) are with respect to video 1.

Our first implementation, of parallelizing all SAD launches per iteration, is quite slow, taking almost 3 times as long as the original CPU implementation. That is to be expected, as launching a single block with $3 \times 3$ threads barely makes use of the GPU's resources, making it difficult to get past the overhead.

TABLE III
STEP SEARCH SOLUTIONS' EXECUTION TIME

| Method | Video 1 [s] | Video 2 [s] | Video 3 [s] | Speedup |
|---|---|---|---|---|
| CPU | 0.0464 | 0.818 | 11.468 | - |
| $3 \times 3$ Block | 1.625 | 3.879 | 32.109 | 0.357 |
| $13 \times 13$ Block | 1.550 | 3.263 | 22.462 | 0.511 |
| Threads in Z | 0.0851 | 1.353 | 18.610 | 0.6162 |
| $7 \times 7$ + Z | 0.0746 | 1.100 | 14.023 | 0.8178 |
| $13 \times 13$ + Z | 0.0736 | 1.140 | 14.250 | 0.8048 |

TABLE IV
STEP SEARCH SOLUTIONS' RESIDUE

| Method | Video 1 [s] | Video 2 [s] | Video 3 [s] | Loss |
|---|---|---|---|---|
| CPU | 5927379 | 39866544 | 692617109 | - |
| $3 \times 3$ Block | 5927379 | 39866539 | 692617735 | 1 |
| $13 \times 13$ Block | 5160609 | 33633200 | 681735270 | 0.871 |
| Threads in Z | 5927379 | 39866539 | 692617735 | 1 |
| $7 \times 7$ + Z | 5814731 | 34257434 | 691075972 | 0.981 |
| $13 \times 13$ + Z | 5160609 | 33633200 | 681735270 | 0.871 |

Increasing the granularity gives us some interesting results. First in foremost, the performance slightly increases, although still slower than the original. However, perhaps more important, we can see that the residues are smaller, although still higher than the values obtained for FSBM. This is due to the fact that we are now analysing points that the original algorithm otherwise would not. If we maximised the granularity, i.e. analysed every point, we would have, in practice, the FSBM algorithm.

Now, we make use of pinned memory and also decide to use threads in the z axis to parallelize part of the SAD algorithm. Yet again, the performance increases slightly. The effects of pinned memory are more notable in video 1 as, due to the smaller size, the facilitated memory accesses naturally have more impact. Of course, the overall speedup is still limited by the the additional overhead in the kernel launch and the required reduction.

Finally, we increase the granularity for this new strategy. Obviously, the obtained residue is the same with or without threads in the z axis, as only the granularity has an effect in the algorithm's accuracy. It is interesting to note that a granularity of 2 levels ($7 \times 7$) allows for better performance than a granularity of 3 levels ($13 \times 13$). This shows us that the additional overhead from launching $13^2$ threads in X and Y instead of only $7^2$ is bigger than the speedup obtained in having to perform 1 less `while` loop iteration. Nevertheless, the performance is still similar and the residue is significantly closer to an ideal solution. Therefore, the choice in granularity depends on whether the user prefers performance or accuracy.

Unfortunately, we are not able to achieve a performance better than the original CPU implementation. This means that, despite our strategies to make better use of the available resources, we are not able to produce something that makes up for the overhead.

### C. Normalised Gain

We now perform one final analysis for our implementation of both algorithms, with respect to execution time, accumulated residue and thus our target metric: the normalised gain.

For the FSBM algorithm the choice of implementation is clear. It is the implementation that performs all calculations on the GPU and includes all other mentioned optimization steps, including reduced memory transfer overhead and 3 dimensional grid configuration (with threads in the z axis): `fsbm_threads_in_z.cu`

For the SS algorithm, we choose the implementation with a 3 dimensional grid, but the choice of granularity is not as direct. Since the normalized gain depends on both the performance and the prediction losses (or gains), we choose the implementation with $13 \times 13$ CUDA blocks, as it provides the highest prediction gain, with similar, although slower, performance: `ss_13x13+z.cu`

The results are presented in table V. As we can see, for the FSBM algorithm, the prediction loss ratio is very close to 1. This is because the few segments that get discarded are very rarely the minimum values and, as such, have very little effect on the resulting video. As such, the normalized gain will be equivalent to the obtained speedup. We can see that the obtained speedup is the smallest for video 1 and the highest for video 3. That is to be expected. Video 1 has a smaller size, and less frames to encode. As such the overall weight of the block we parallelize is smaller. Therefore, the speedup we are able to obtain will obviously be smaller. Meanwhile, for larger videos and with more frames to encode, such as video 3, there is much more to parallelize during the program's execution and, consequently, we are able to obtain a higher speedup. Thus, we are able to achieve a normalized gain above $10\times$, for the FSBM algorithm.

On another hand, for the SS algorithm, the results are a bit less promising. Since our implementation has more granularity, the accumulated residue is significantly smaller for all videos, while still higher than the residue for the FSBM. As such, we don't have a prediction loss, but a prediction gain instead. This means that the normalized gain will be higher than our speedup. The prediction gain is highest in video 2 (a loss of 0.844), since in the additional blocks that are analysed happen to be important for the resulting encoded video. For the speedup, the conclusions are the same as the ones presented in the previous section. We are unable to achieve any speedup, although the performance does get close since, with a speedup of 0.805, our implementations is approximately 25% slower. Just as with the previous algorithm, the speedup is higher for larger videos and with more frames. The normalized gain is higher than the speedup, because of the prediction gains. Nevertheless, it is still below 1. However, it is highest for video 2, since it is the one that benefits the most from the higher granularity ($0.851\times$). As such, we can postulate that, for a very large video that benefits a lot from the higher granularity, our implementation might match the performance of the original CPU code, or even attain a speedup, even if very small.

## IV. CONCLUSION

Throughout this lab assignment we explored algorithm optimization for Motion Estimation on an heterogeneous system, specifically for the *Full Search Block Matching* (FSBM) and *Step Search* (SS) algorithms.

TABLE V
FINAL RESULTS

| Video | Algorithm | CPU | | GPU | | Prediction Loss | Speedup | Normalized Gain |
|---|---|---|---|---|---|---|---|---|
| | | Accumulated Residue | Execution Time | Acumulated Residue | Execution Time | | | |
| Video 1 | FSBM | 4216103 | 0.983 | 4216103 | 0.2447 | 1 | 4.02 | 4.02 |
| Video 2 | FSBM | 31547568 | 21.232 | 31547531 | 2.695 | $\approx 1$ | 7.88 | 7.88 |
| Video 3 | FSBM | 634064063 | 318.128 | 634063236 | 29.828 | $\approx 1$ | 10.66 | 10.66 |
| Video 1 | SS | 5927379 | 0.0464 | 5160609 | 0.0736 | 0.871 | 0.630 | 0.724 |
| Video 2 | SS | 39866544 | 0.818 | 33633200 | 1.14 | 0.844 | 0.718 | 0.851 |
| Video 3 | SS | 692617109 | 11.468 | 681735270 | 14.25 | 0.984 | 0.805 | 0.818 |

For FSBM, we focused on parallelizing the SAD function calls, making full use of the CUDA threads in all 3 dimensions and minimizing the data transfer between CPU-GPU channel, which allowed us to achieve a speedup of $10\times$.

On another hand, for the Step-Search algorithm, we were less successful. Although a speedup could not be obtained, mainly related to its significantly lower arithmetic intensity, a break-even in an heterogenous system was almost achieved (normalized gain of $0.851\times$). For that, we combined multiple SAD function calls into one kernel, increased the algorithms granularity and used leftover threads for parallelizing the SAD itself. Our results also hint at the possibility of obtaining small, dependending on the target video.

We decided not to tackle the *Test Zonal Search* algorithm, since it is already highly optimized and thus, with low potential for further acceleration.

Overall our work highlights the importance of code archetype analysis for CUDA execution, specifically the identification of SIMD compatible code with high arithmetic intensity, as well as the development of critical-thinking so to evaluate which algorithms are worth to optimize and run in a CPU-GPU system.

## REFERENCES

[1] *Course Slides on CUDA*.