DEEC
DEPARTAMENTO DE ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES
TÉCNICO LISBOA

HIGH-PERFORMANCE COMPUTING ARCHITECTURES

(2023-2024)

Lab assignment #3

Abstract: The third lab consists in the acceleration of a simple program, by relying on a heterogeneous system composed of a CPU and a GPU. For this purpose, students will exploit the CUDA application programming interface and data-level parallelism to offload computationally intensive (regular) tasks to a GPU.

1 INTRODUCTION

For the development of this work, students should use CUDA, a parallel computing platform and application programming interface model created by Nvidia to enable general-purpose computing on their own GPUs. For the purpose of this work, students will use a heterogeneous machine composed of:

- CPU: Intel Core i7-7700K CPU @ 4.20GHz (4 cores, supporting up to 8 concurrent threads)
- GPU: NVIDIA GeForce GTX 1070 (15 SMs, each with 128 CUDA cores – a total of 1920 CUDA cores)
- Memory: Host 16 GB / device 8GB.

To connect to the machine, students should use cuda1 or cuda2 machines, as in the previous lab.

Machine names: `cuda1.scdeec.tecnico.ulisboa.pt`
`cuda2.scdeec.tecnico.ulisboa.pt`
application source folder: `/extra/acedes/interPrediction`
application data folder: `/extra/acedes/video`

To help the development of this work, students are advised to use the information on the lecture slides, but can also use other information on the web, e.g.,

- https://www.tutorialspoint.com/cuda/cuda_introduction.htm
- David B. Kirk, Wen-mei W. Hwu. "Programming Massively Parallel Processors: A Hands-on Approach", Morgan Kauffman.

Nvidia also provides multiple examples of CUDA applications, which are usually distributed with the CUDA drivers and API. On the target machine, the CUDA examples are available at:

```
/extra/cuda-samples-12.3/
```

All applications have been previously compiled and are available at:

```
/extra/cuda-samples-12.3/bin/x86_64/linux/release/
```

One particularly useful application is `deviceQuery`, which allows to query the number of available NVIDIA GPUs, as well as its properties. In this case, by running the command:

```
/extra/cuda-samples-12.3/bin/x86_64/linux/release/deviceQuery
```

We get the output shown in Figure 1.

To develop the programs, students should rely on NVIDIA compiler. For this purpose, students can either: (1) re-write the code using one single file that includes both the host and the device code; or (2) use multiple files, one for the host code, others for the device code. In either case, always remember that device code must be placed inside a file with the .cu extension. If using a single file (option 1), you can compile the code by using the following line:

```
nvcc -O3 -lm <source code.cu> -o <output binary file>
```

```
/extra/cuda-samples-12.3/bin/x86_64/linux/release/deviceQuery Starting...

CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "NVIDIA GeForce GTX 1070"
  CUDA Driver Version / Runtime Version          12.3 / 12.3
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 8114 MBytes (8507949056 bytes)
  (015) Multiprocessors, (128) CUDA Cores/MP:    1920 CUDA Cores
  GPU Max Clock rate:                            1721 MHz (1.72 GHz)
  Memory Clock rate:                             4004 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 2097152 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384,
16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        98304 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 12.3, CUDA Runtime Version = 12.3,
NumDevs = 1
Result = PASS
```

Figure 1 - Output of the example

`/extra/cuda-samples-12.3/bin/x86_64/linux/release/deviceQuery`

2 ASSIGNMENT WORK PLAN

The objective of this assignment is to accelerate part of a video encoding application, available at the host folder `/extra/acedes/interPrediction`. The folder contains the following files:

- `interPrediction.c`: main computational file **←this is the file you need to accelerate**
- `Makefile`: script for compilation and cleaning. Running "`make`" will generate the binary file, which will compile your program to run on host (optimized with flags -O3) and to run on the GPU (notice that the current source files do not include any GPU kernel, hence nothing will be run on the GPU at the moment – this is just a placeholder). Running "`run.sh`" will run the generated binary file, outputing the kernel execution time.
- Folder `video/`: holds the input video sequences.

Start by copying the interPredition source files to your own folder location, e.g.:

```
cp -r /extra/acedes/interPrediction interPrediction
```

Move to the created folder and create a symbolic link to the input video data:

```
cd interPrediction
ln -s /extra/acedes/video
```

Compile the binary:

```
make
```

Run the application with the default input:

```
run.sh
```

Application acceleration:

Students can apply any transformation to the code that may help minimizing the overall execution time. The results (speed-up) should always consider as reference the time of the original source code, when compiled with flag -O3.

To measure the execution time, the `clock_gettime()` function can be used (part of the `<time.h>` library). This allows accurate time measurements with nanosecond precision. The main file already uses this function to measure time.

Grading criteria:

The provided source code implements 3 distinct algorithms to compute the image prediction (see Appendix):

- **Full-Search Block-Matching (FSBM)**: represents the optimal "brute-force" exhaustive search, requiring a long execution time. It is characterized by a very regular execution pattern.
- **Step-Search (SS)**: sub-optimal search approach, where the search pattern adopts a logarithmic search along the central position of the search area. It is characterized by a quasi-regular execution pattern, with some intermediate decisions that imply some reduction of the regularity.
- **Test Zonal Search (TZS)**: state-of-the-art sub-optimal search approach, where the search procedure implements a diamond-search pattern followed by a regular refinement raster pattern. It is characterized by a less-regular execution pattern, as a result of several intermediate decisions.

The students are expected to optimize only one of the provided algorithms. Nevertheless, since these algorithms are characterized by a distinct regularity, the required optimization effort will not be the same. Consequently, the maximum grade that will be given to the students will be the following:

- **Full-Search Block-Matching (FSBM)** - Maximum Grade = 15 points;
- **Step-Search (SS)** - Maximum Grade = 17.5 points;
- **Test Zonal Search (TZS)** - Maximum Grade = 20 points;

Acceleration tips:

1. Start by implementing a basic kernel on the GPU. A simple use case you can try is the addition of two matrices. Make sure you can run a kernel on the GPU and that the result is correct. You can also use the CUDA examples.
2. Identify the most frequently used function of the application (`SAD`) and focus your efforts on optimizing this function, trying to improve the memory accesses and reduce (as much as possible) the amount of instructions that compromise the regularity of its execution (e.g., reduce/eliminate if-clauses).
3. Identify which loops can be parallelized and estimate the total number of iterations in each parallel loop.
4. Ensure the loops you accelerate are the ones with the highest parallelism level and avoid multiple data transfers (if you can transfer data to the GPU only once, the better).
5. It might be useful to unroll some loops, either inside the `SAD` function, or in the parent routine that calls this function. This will provide you with more opportunities for parallelization (and thus acceleration). You can also try to merge some of these functions.
6. Always validate the resulting output by comparing the computed Accumulated Residue with the one that is obtained (with the same algorithm and video sequence) at the CPU.
   NOTE: Since the search procedure of the SS and TZS algorithms depend on the intermediate decisions, the provided result might correspond to a local minimum. This is not problematic provided that the resulting Accumulated Residue value is not too much different than the expected one.
7. Measure your speed-ups with different video sequences.

3 REPORT FORMAT

The project should be completed until March 31st 2024, with the PDF file of the up to 6-page report being submitted online (via Fenix) with the corresponding code (in a .zip file). Do not forget to clean the directory before submitting your work. The report should follow the template for the IEEE Computer society journals as in the previous assignment:

https://www.ieee.org/publications_standards/publications/authors/author_templates.html

The report should be structured as follows:

- **Abstract**: 100-150 words summarizing the problem, solution, and results.
- **1. Introduction**: Explain the difference (in architecture) between CPUs and GPUs and identify how (and in which cases) GPUs can be used to accelerate the execution of an application.
- **2. Methodology:** Explain how (and why) the original source code was modified to improve the performance.
- **3. Results:** Describe the attained results and draw your conclusions regarding what are the used solutions.
- **4. Conclusions:** Summarize the report, including approach and results.

Please notice that while the report can be shorter than 6 pages, no extra page will be considered, i.e., pages 7 and beyond will be ignored for evaluation.

NOTE: The following table should be included in the results section with the obtained values in order to sustain the presented discussion of the attained optimization in the GPU.

Video Sequence	Considered Search algorithm <FSBM|SS|TZS>	CPU		GPU		Prediction Loss $(5)=\frac{(3)}{(1)}$	Speedup $(6)=\frac{(2)}{(4)}$	Normalized Gain $(7)=\frac{(6)}{(5)}$
		Accumulated Residue (1)	Execution Time (2)	Accumulated Residue (3)	Execution Time (4)			
Video 1								
Video 2								
Video 3								
Video 4								

4 DEADLINES

March 27 (class): Validation of the attained speedup (during the lab class).

March 31 (23h59): Report submission.

APPENDIX – SOME MPEG-X AND H.26X VIDEO CODING BASICS

In the last decades there has been a general proliferation of advanced video services and multimedia applications, where several MPEG-x/H.26x video compression standards have been developed to store and broadcast video information in digital form. The main goal of such video compression algorithms is to exploit both the spatial and the temporal redundancy within a sequence of images, in order to achieve the maximum compression as possible. While spatial redundancies can be extensively exploited through the use of transform-based algorithms (such as the Discrete Cosine Transform (DCT)), temporal redundancies are usually exploited through the use of inter-image prediction schemes.

Reducing the Temporal Redundancy

Even at the presence of motion, we can easily expect a high degree of temporal redundancy between the pixels of a given region (A) in one image and the corresponding pixels, moved to region (X), in the following image.

In video coding, such temporal redundancy is usually exploited by only considering the difference between consecutive frames. Hence, if we denote the two contiguous frames in Figure 2 by $I(t_0 - 1)$ and $I(t_0)$, one first approach to implement such prediction method would be to compress the differences frame $\Delta I = I(t_0) - I(t_0 - 1)$.

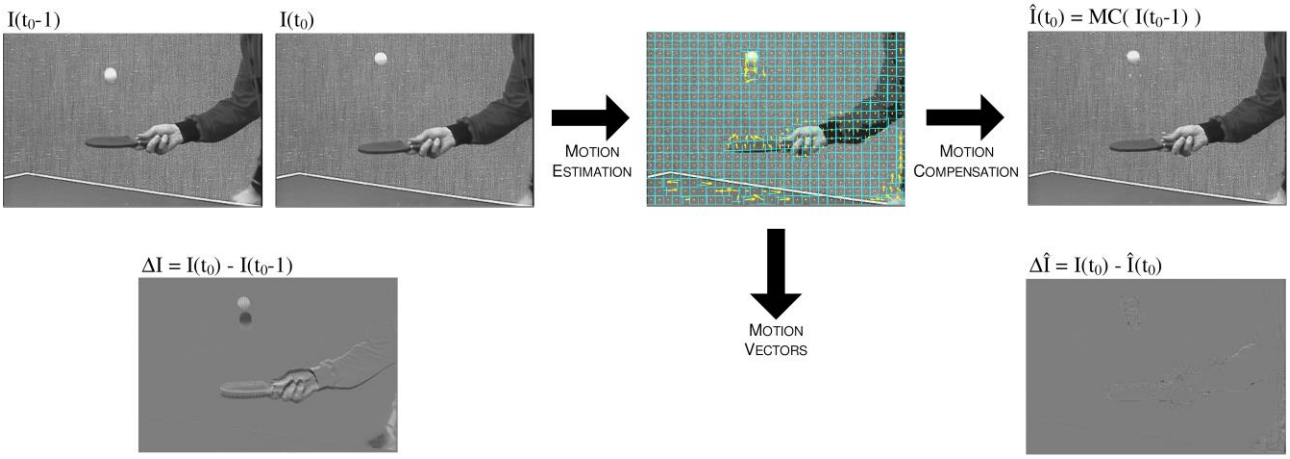


Figure 2 - Reducing the temporal redundancy using the motion compensation algorithm.

However, as it can be seen in Figure 2, such differences frame may still contain a large amount of information. The reason for this fact is mainly due to the presence of motion activity, which introduces a certain displacement to every region of frame $I(t_0 - 1)$ into a new region at frame $I(t_0)$.

One way to reduce the amount of residual information in the differences frame ΔI is to track and compensate such movement using the previously encoded images. The process of computing the changes among frames is usually referred to by **inter-frame prediction** with Motion Compensation (MC). Motion compensation is thus defined as the process of compensating for the displacement of moving objects from one frame to another. In practice, motion compensation is preceded by Motion Estimation (ME), the process of tracking and finding the corresponding pixels among consecutive frames.

Intuitively, one might expect that the ideal procedure for reducing temporal redundancy is the one that tracks every pixel from one frame to the following frame. However, this would be very computationally intensive, and such methods would not provide reliable tracking due to the presence of noise in the frames. Instead of tracking individual pixels from frame to frame, video coding standards only allow tracking of information for $M \times M$ pixel regions, commonly referred to as “blocks”.

Hence, the result of such a prediction scheme will be a synthesized frame $\hat{I}(t_0) = MC(I(t_0 - 1))$, obtained with the blocks of the previous frame $I(t_0 - 1)$, displaced by the corresponding Motion Vectors (MVs). The residue frame obtained by subtracting the current frame $I(t_0)$ with the reconstructed (motion compensated) frame $\hat{I}(t_0)$ will be defined as:

$$\Delta\hat{I}(t_0) = I(t_0) - \hat{I}(t_0) \quad \Leftrightarrow \quad \Delta\hat{I}(x, y, t_0) = I(x, y, t_0) - I(x - u, y - v, t_0 - 1)$$

where $I(x, y, t_0)$ are the pixel values at spatial location (x, y) in the current frame $I(t_0)$, and $I(x-u, y-v, t_0-1)$ are the corresponding pixel values at spatial location $(x-u, y-v)$ in the previous (a.k.a. reference) frame $I(t_0-1)$. The output of the motion estimator, the coordinates (u, v), defines the relative motion of a block from one frame to another and is referred to as the motion vector for that block at (x, y). $I(x-u, y-v, t_0-1)$ is refereed to as the motion-compensated prediction of $I(x, y, t_0)$, and $\Delta\hat{I}(x, y, t_0)$ is the prediction residual for $I(x, y, t_0)$.

As it can be seen in Figure 2, such a residual frame will contain a significantly smaller amount of information. It is this differences or residual frame, as well as the computed motion vectors, that are coded and transmitted to the video decoding system.

Motion Estimation Procedure

Motion estimation is usually the most computation intensive operation of any MPEG-x video encoding system. Figure 3 illustrates the motion estimation problem as it is posed in video coding standards. Given an $N_h \times N_v$ sized reference picture and an $N \times M$ block in a current picture, the objective of motion estimation is to determine the $N \times M$ block in the reference picture that better matches (according to a given criterion) the characteristics of the block in the current picture. As it was referred before, we define the current picture as the frame (or image) at time instant t_0, whereas the reference (previous) picture is defined as the frame (or image) at time instant $t_0 - 1$.

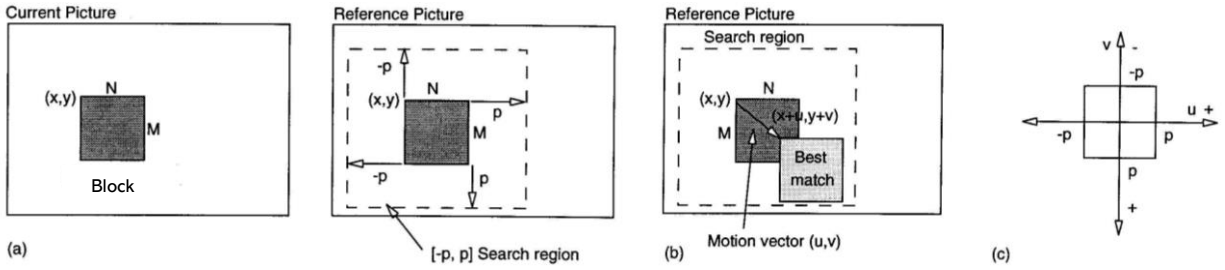


Figure 3 - Motion estimation procedure.

The location of the block region is given by the (x, y) coordinates of their left-top corner. Ideally, we would like to search the whole reference frame for the best match; however, this is impractical. Instead, we restrict the picture search to a $[-p, +p]$ search region around the original location of our block in the current picture.

Let $(x+u, y+v)$ be the location of the best matching block in the reference picture (see Figure 3b). In motion estimation terminology, the vector from (x, y) to $(x+u, y+v)$ is referred to as the motion vector associated with the block at location (x, y). Often, the motion vector is expressed in relative coordinates, that is, we assume that (x, y) is at location $(0,0)$ and thus the motion vector is simply expressed as (u, v).

The coordinate system associated with the motion vector is shown in Figure 3c. In particular, for the search region shown in Figure 3a, $-p \leq u, v \leq +p$.

Matching Criterion

Let the pixels of the block in the current frame be denoted as $C(x, y)$ and the pixels in the reference frame be denoted as $R(x, y)$. We define the Sum of Absolute Differences (SAD) cost function as:

$$SAD(i,j) = \sum_{k=0}^{N-1} \sum_{l=0}^{M-1} |C(x+k, y+l) - R(x+i+k, y+j+l)|$$

with $-p \leq i, j \leq +p$. The best matching block is defined as the block $R(x+i, y+j)$ for which the $SAD(i, j)$ measure is minimized. The coordinates (i, j) will then define the corresponding motion vector.

Since the matching procedure is carried out with rectangular regions, in video coding terminology such processing scheme is referred to as Block-Matching Algorithms (BMA).

Full-Search Block-Matching (FSBM) motion estimation algorithm

The simplest method to find the motion vector for each block is to compute $SAD(i,j)$ at each location of the search space. This is referred to as the (optimal) **Full-Search Block-Matching (FSBM)** algorithm. The implementation of this algorithm to compute all the motion vectors $MV(x,y)$ of an entire $N_h \times N_v$ pixels image is illustrated in **Error! Reference source not found.**.

For each motion vector there are $(2p+1)^2$ search locations. At each search location (i,j) we compare $N \times M$ pixels. Each pixel comparison requires three operations, namely, a subtraction, an absolute-value calculation and one addition. Thus, the total complexity per macroblock is $(2p+1)^2 \times N \times M \times 3$ arithmetic operations. For a picture resolution of $N_h \times N_v$ and a frame rate of F pictures per second, the overall complexity is $\frac{N_h \times N_v \times F}{N \times M} \times (2p+1)^2 \times N \times M \times 3$ operations per second. Hence, for typical setups used in broadcast digital TV (HD format: $N_h = 1920$ pixels, $N_h = 1080$ pixels, $F = 30$ frames/s), $N = M = 32$ and $p = 64$, the full-search block-matching algorithm requires about 788.5 GOPS (Giga operations per second).

```
/* ME algorithm for all blocks of the N_h × N_v image */
for lin = 0 to ⌊N_v/M⌋ do
    for col = 0 to ⌊N_h/M⌋ do
        /* Computation of the MV of the block defined by its upper-left corner (lin × M, col × N) */
        Best_SAD ⇐ ∞ /* Minimum SAD initialization */
        /* Searches all position of the search window defined within the previous image */
        for i = −p to p do
            for j = −p to p do
                /* Processing of all the M × M pixels of the candidate block under processing */
                SAD ⇐ 0 /* SAD measure initialization */
                for k = 0 to (M − 1) do
                    for l = 0 to (N − 1) do
                        SAD += | C (lin × M + k, col × N + l) − R (lin × M + i + k, col × N + j + l) |
                    end for
                end for
                /* Comparison between the obtained SAD and the minimum SAD computed so far */
                if SAD (i, j) < Best_SAD then
                    Best_SAD = SAD
                    MV (lin, col) = (i, j)
                end if
            end for
        end for
    end for
end for
```

Figure 4 - Full-search block-matching motion estimation algorithm.

Step Search (SS) motion estimation algorithm

This sub-optimal algorithm is particularly popular because of its simplicity, robustness, and near optimal performance. It searches for the best motion vectors in a coarse-to-fine search pattern. The algorithm can be described as:

Step 1: An initial step-size is picked. Eight blocks at a distance of step-size pixels from the centre (around the centre block) are picked for comparison.

Step 2: The step-size is halved. The centre is moved to the point with the minimum distortion that was found in the previous iteration.

Steps 1 and 2 are repeated till the step size becomes smaller than 1. An example path for the convergence of this algorithm is shown in **Error! Reference source not found.**. One problem that occurs with this search algorithm is that it uses a uniformly allocated checking point pattern in the first step, which becomes inefficient for small motion estimation.

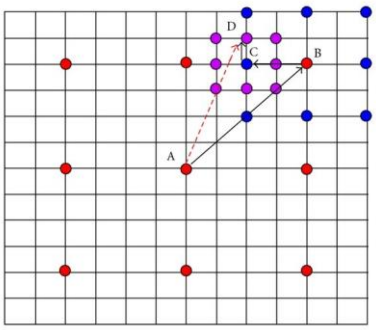
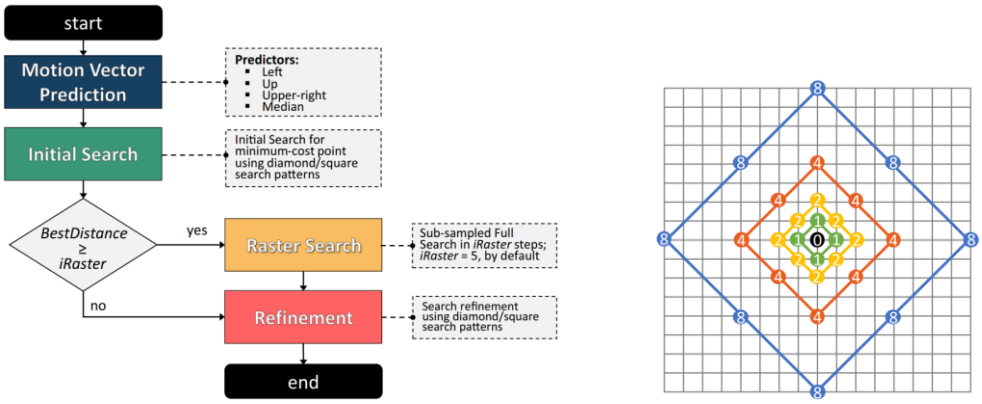


Figure 5 - Example convergence path of the Step Search algorithm.

Test Zone Search (TZS) motion estimation algorithm

This algorithm is one of the current state-of-the-art block-matching algorithms that is often adopted in HEVC and VVC encoder implementations. It is also a sub-optimal fast motion estimation algorithm that presents a coding efficiency close to the achieved by the optimal FSBM, requiring less computational resources.

TZS is a combination of four steps, as shown in Figure 6: (1) Motion Vector Prediction, (2) Initial Search, (3) Raster Search, and (4) Refinement.



(a) (b)

Figure 6 – (a) Flowchart of the Test Zone Search (TZS) algorithm; (b) Diamond-Search pattern.

The GPU optimization that is required in this assignment covers only step-2 (Initial-Search), step-3 (Raster-Search), and step-4 (Refinement). It is assumed that step-2 begins with a start-position corresponding to the centre of the search area $(0,0)$.

The Initial-Search step performs the first search around the Search Area (SA), in which a diamond search pattern is executed, as shown in Figure 6b. This pattern is expanded in powers of 2 until a predefined limit of the SA is reached. The best point is again the one with the lowest SAD value. The distance obtained for the best point is stored in the "$bestResult$" variable, which will be evaluated in further steps.

The Raster-Search step is a sub-sampled full search, performed in a limited SA. This Raster-Search is performed when the difference between the motion vectors obtained from the previous step and the starting position is too large. By default, if the $bestResult$ stored in the previous step is smaller than $iRaster$ (a constant equal to 5 by default), the Raster-Search is skipped. The $iRaster$ parameter also defines the sub-sampling factor performed both horizontally and vertically in the Raster-Search, resulting in a total of n candidate blocks to be compared, with $n = \left\lceil \frac{1+(SR \times 2)}{iRaster} \right\rceil$.

The Refinement is the last step, to provide a greater accuracy for the MV obtained from previous steps. A new search, identical to the one performed in Initial-Search, occurs around the best candidate block found in the previous step.

If needed, see more details about this algorithm in:

P. Goncalves, C. Moraes, M. Porto, G. Correa, "Complexity-Aware TZS Algorithm for Mobile Video Encoders", Journal of Integrated Circuits and Systems, Vol. 14, No. 3 (2019) https://doi.org/10.29292/jics.v14i3.60

P. Goncalves, G. Correa, L. Agostini, et al. "Learning-based bypass zone search algorithm for fast motion estimation", Multimed Tools Appl 82, 3535–3560 (2023). https://doi.org/10.1007/s11042-022-13094-6