Parallelisation of the Block Matching Algorithm Using Three Step Search - Research Assignment

Daniel Bonanno (UserID: dbon0031), *University of Malta*

Question 1.) Problem Description

The aim of this assignment is to implement the block matching algorithm used for motion estimation in video coding. The algorithm aims to maximize compression by exploiting both temporal and spatial redundancies between frames [1-2]. Since successive frames will be separated by a time interval t, it is justifiable to assume that some form of correlation will be present between the frames. Thus, a frame can be used as a predictor for another frame. This exploits temporal correlation. For the sake of simplicity, in this explanation, it is assumed that the x-1th frame will be used as a predictor for the xth frame, that is, successive frames will be used. The amount of temporal correlation present between the frames is dependent on the frame rate: the smaller the time interval between frames (that is, the higher the frame rate), the higher the correlation.

For 2 frames, it is also expected that spatial correlation exists. Firstly, not all objects might move between frames. Consider the case where a camera is stationary: in this case, most of the background will remain constant. Furthermore, spatial correlation exists between moving objects: if a ball has moved between frames, parts of the ball can still be obtained from the previous frame. Consider the example in Figure 1. Most of the blue background has remained constant between frames and, although the red circle has moved, its motion can be easily mapped from the previous frame using a motion vector.

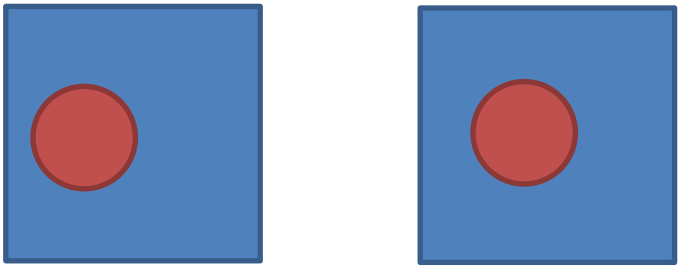


Figure 1: Example of 2 frames in a video; Frame 1 (Left) and Frame 2 (Right) demonstrate the motion between successive frames and how correlation between them exists

In order to exploit such correlations, the block matching algorithm aims to generate this motion vector by splitting the frame that is to be predicted (frame x) into macroblocks of size m x n. For each macroblock, a search area with search parameters $p_{vertical}$ and $p_{horizontal}$ is defined from the predictor frame. The search parameters determine the amount of pixels the search area should span from the macroblock. An illustration of this can be seen in Figure 2. A cost function is used to compare macroblocks in the search area to the macroblock in the actual frame. Examples of this cost function are the Mean Absolute Error and the Mean Square Error. The block in the search area which results in the lowest cost is the block that is closest to the current frame's macroblock. A motion vector is generated, such that the block with the lowest cost is mapped onto the current frame. An example can be observed in Figure 3. It can be noted that this might not result exactly in the frame to be predicted, but the difference between the actual and the predicted frame can be generated and transmitted to obtain the actual xth frame. In this assignment, this is not implemented since this is not considered as a part of the block matching algorithm.

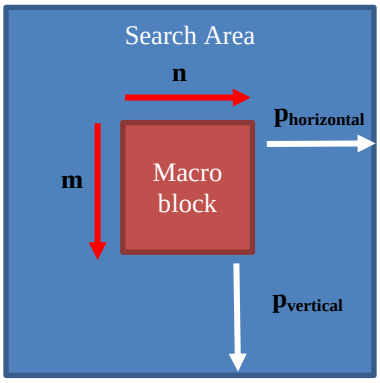


Figure 2: An example of how a search area is defined for a macroblock

Question 2.) Justification of Chosen Problem

Given that this problem is highly utilized in video coding, such as in H.264 video compression [2], the fastest possible implementation of it is desirable, especially for use in real time video coding. The computational complexity of the problem is dependent on different factors. The chosen m and n values determine the macroblock size. The smaller these values are, the more macroblocks there will be and therefore, the more motion vectors need to be generated. For a given pair of m and n values, the resolution of the image also determines the amount of macroblocks on which the

algorithm will need to perform. Assume m and n are both 8: for 1080p (1920x1080), this will result in $\frac{1920}{8} \times \frac{1080}{8} = 32400$ macroblocks whereas for 720p (1280x720), this will result in $\frac{1280}{8} \times \frac{720}{8} = 14400$ macroblocks. Since nowadays higher resolution video is being generated, then speeding up this compression algorithm is necessary.

Similarly, the search area parameters and the search area algorithm chosen will affect the time taken for the block matching algorithm to run [1 - 4]. If the search area parameters are chosen to be significantly large, with interpolation between pixels such that a motion vectors can be made up of non-integer values, the algorithm's computation complexity becomes significantly large [3]. If an exhaustive search is performed (where all possible macroblocks in the search area are considered), the best results are achieved, since the lowest error will be found. However it is very computationally expensive [3-4]. [1] also notes that the cost function used also adds to the computational complexity: if the Absolute Error is used, no multiplications are necessary whereas if the Mean Squared Error is used, the complexity becomes higher.

With these concepts in mind, this problem seems to be an ideal candidate for a parallel implementation.

By parallelizing operations that work on different data, as is the case when calculating the cost for each search block, the solution can be sped up. In fact, in [4] a block matching algorithm with an exhaustive search was developed to work on multiple GPU cards, with the authors noting that their implementation is suitable for processing surveillance video at 720x480 pixel resolution at 30 fps in real time.

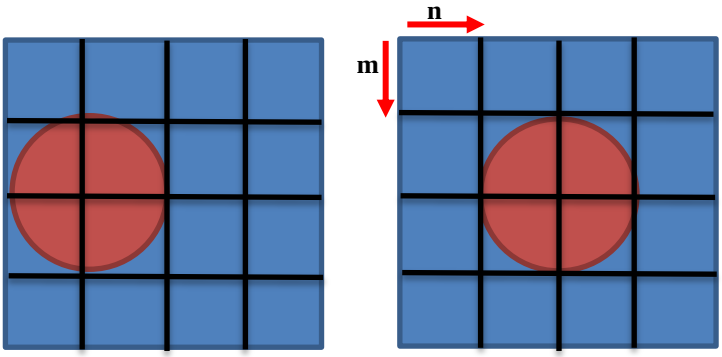
The main difference between this implementation and the ones in [2] and [4] will be the fact that a variant of the Three Step Search will be used instead of an exhaustive search. Whereas an exhaustive search will check every possible macroblock in the search area, the three step search checks only 25. This makes it significantly faster, at the cost of a less accurate result.

The three step search algorithm runs as follows [3]:

- Start at the macroblock location in the search area. Let the top left pixel of this area be known as the centre and be denoted by the coordinates (0,0).
- Calculate the cost function for the 8 surrounding blocks obtained by moving an amount of pixels in the positive and negative directions vertically and horizontally from the centre equal to half the search area parameters. For example, assume that the search parameter is equal to 8 pixels in both directions. This means that the cost function is calculated for search blocks in the search area having the following top left pixel coordinates: (-4,-4), (-4,0), (-4,4), (0, -4), (0,4), (4, -4), (4,0) and (4,4) from the origin define above.
- The block with the least cost is chosen, with its top left pixel denoting the new centre, the search parameters are set to 2 and the previous step is repeated.
- Again, the block with the least cost is chosen as the new centre and the search parameters are set to 1. A final search is performed and the block with the lowest cost is chosen as the best matched block.

It can be noted that this algorithm starts with a coarse search and then ends up with a fine search. In this implementation, 27 searches are performed rather than 25, since the cost of the block which is considered to be at the centre is re-computed. This is done for simplicity. Figure 4, obtained from [3], gives a visual representation of the Three Step Search Algorithm.

Another difference between this study and that in [4] is the fact, whilst in [4] the algorithm was tested with both an integer search grid and a non-integer search grid (therefore making use of interpolation techniques), this study will only focus on an integer search grid, for the sake of simplicity. The way parallelization is done in [4] is also slightly different from what is proposed in this



(0,0)	(0,0)	(0,0)	(0,0)
(-m,0)	(0,-n)	(0,-n)	(0,0)
(m,0)	(0,-n)	(0,-n)	(0,0)
(0,0)	(0,0)	(0,0)	(0,0)

Figure 3: Frame 1 (Left) and Frame 2 (Centre) split into macroblocks. The motion vector (Right) shows the number of pixels that need to be moved by each macroblock in the second frame to obtain the closest matched macroblock in the first frame

study. This will be further discussed in the next section.
The way [2] and [4] propose parallelisation is also different than what is proposed here. This is also discussed in the following section.

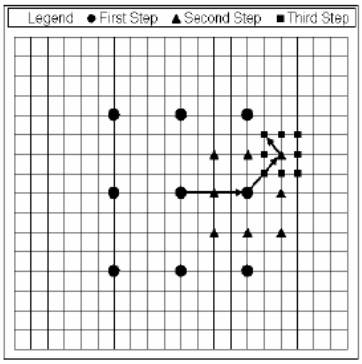


Figure 4: A visual representation of the Three Step Search Algorithm (Source: [3])

QUESTION 3.) PLAN FOR PARALLEL IMPLEMENTATION

A. *Parallel Operations*

The block match algorithm's main operation is the searching operation: this is the operation that takes the most time since the cost function must be calculated for a number of macroblock – search block pairs, each consisting of m x n pixels, with the possibility of each pixel being made up of more than one channel. Thus, it is worthwhile to speed up this operation. This can be done in more than one way.
The approach taken by [4] is to create a thread for each possible search block in the search area. The cost function is then evaluated in each thread, the best search block is chosen and its direction vector is taken as a result. In this case, since as exhaustive search was used there were a considerable number of search blocks, each with its own thread. A similar approach is taken by [2], which also employs an exhaustive search. In [2], a thread works on a 4x4 block, with each possible block in the search area being served by one thread. Threads which belong to the same search area (and therefore the same macroblock) are organised in blocks. All the blocks together for the whole image and are mapped onto the grid. Both [2] and [4] make use of CUDA.
The parallel implementation proposed in this study will be slightly different. Since a three step search will be used, there will only be 9 search blocks per step, with the blocks in the second and third step are dependent on the outcome of the previous step. Thus, in order to make full utilisation of the GPU, a pixel-level thread is being proposed.
With 9 search blocks, m x n pixels per search block and c channels per pixel, a total of 9 x m x n x c threads per step can be created. Each thread of execution will evaluate the cost function of a pixel for a particular channel. In order to obtain the total cost for a search block, two methods are proposed. In the first, the group of threads belonging to a particular macroblock – search block pair (of which there are m x n x c) increment a single variable which holds the total cost for the macroblock – search block pair by the result obtained in the thread. Once all the threads finish, the total cost would be in this variable. This ideology can be observed in Figure 5.
Another proposition is to have all the threads belonging to a particular block's channel write their result in an m x n matrix, rather than updating a single variable. Once all the threads for a channel complete, the matrix can be summed up to obtain the total cost for a particular channel for the macroblock – search block pair. Note that this sum can also be implemented in parallel by splitting up the data. As soon as the cost for all the channels is known, all the channel costs can be summed up to obtain the total cost for this particular pair. This method is depicted in Figure 6 below. Once all the threads complete and all the sums have taken place, the cost for all the 9 blocks will be known. The block with the least cost is chosen and this process is repeated two more times, as per the three step search algorithm described previously.
NB: Both Figure 5 and 6 assume a 2 x 2 search block and 3 channels: R, G and B.

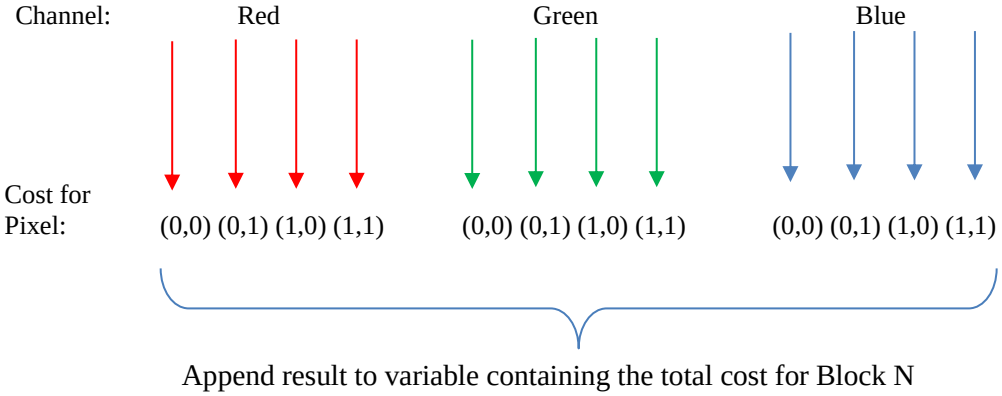


Figure 5: The parallelisation proposed for a single block in a single step. Each arrow represents a thread. The idea is to have this parallelisation mechanism implemented for all 9 blocks in a step. Once the outcome of a step is determined, the same method is repeated for the next step.

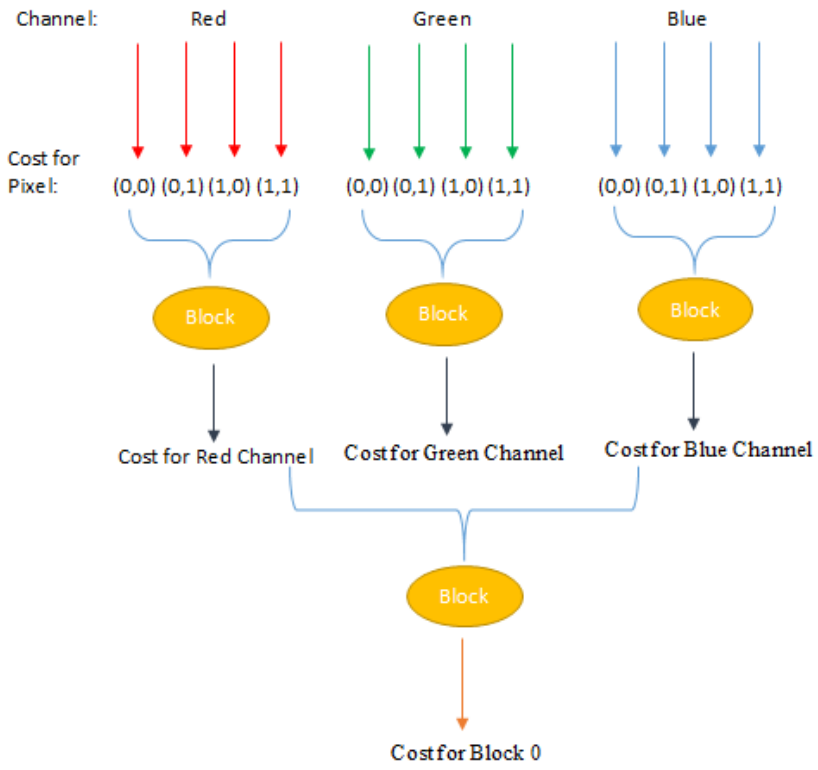


Figure 6: Another approach of how the parallel implementation for one block can be done. Note how more blocks are present, but this does not necessarily make the algorithm slower, since no locking will take place here, but only when summing up. This lock can also be used less frequently than the one proposed in Figure 5.

Another two operations that may be worth parallelising are the actual splitting of the frame into macroblocks and the recombination of the search blocks with the least cost. Assuming that, with the block sizes chosen, L macroblocks will be created from a single frame, then L threads can be initialised to create each macroblock. Similarly, L threads can be created to create the reconstructed frame from the motion vectors. Although these 2 operations are not the bulk of the block matching algorithm, parallelising them might give the algorithm a speed boost.

B. Required Synchronization Points

Following the structure of the block matching algorithm, the first parallel operation which will be looked at is the segmentation of the frame into macroblocks. This operation will read from the frame and write to a data structure which will hold all the macroblocks. Note that each thread will read from the same image without the image changing, therefore there are no risks of race conditions. As for the write operation, each thread will write to a different location in the data structure and therefore, there is no need for locks. The only synchronisation point that might be needed for this operation is a waiting mechanism to block the algorithm from moving forward before all the threads have finished segmenting. However, this might not be needed at all since, once a macroblock is obtained from the segmentation, further processing on it can be done, since all macroblocks are independent.

Once segmentation and creation of the macroblock is complete, the algorithm and starts performing the three step search on it. Again, all the threads are independent, meaning that they will all read 1 pixel from the macroblock and 1 pixel from the search block, with each thread working on a different pixel. This means that no risk of race conditions is present in this part of the algorithm. In the first proposition, however, once a thread finishes executing it will append the cost for that pixel to a variable that will contain the total cost of the macroblock – search block pair. This variable will also be accessible to other threads working out the cost of a pixel in the same macroblock – search block pair. Thus, a locking mechanism is required, such that race conditions are avoided when reading from and writing to this variable. Also, waiting must occur until all the threads that belong to a particular step in the three step search finish, such that the search block with the lowest cost can be chosen and the next step in the three step search can be run.

In the second proposition, all the pixel-level costs will be written to separate locations in a matrix, such that they are only summed once all the pixel-level threads are completed. This would mean that no locking is needed, since data is only being written to this matrix. A locking

mechanism would be needed, however, if the summing of this matrix is implemented in parallel, since the sum of all the different groups of data will be written to one variable which would contain the total cost of a particular channel. Having said that, this lock would be used less frequently, since this would depend on how many threads are employed to sum up the matrix rather than on how many pixels there are in a search block. With all the costs for the different channels for a particular search block calculated, the total cost for a search block can be obtained.

Once all the search block costs are known, the choice of the search block with the lowest cost is performed and this section of code is re-run again twice, until the three step search finishes.

The next step would be the reassembly of the frame from the motion vectors generated. This part of the algorithm has to read each motion vector for each macroblock, read the search block from the reference frame and write the data to the reconstructed frame. Again, no risk of a race condition is present since each thread reads from and writes to different parts of the image and therefore, no lock is required.

C. Mapping Onto the GPU Architecture

1) Device Global Memory

This memory will be accessible to all threads. Thus, this memory should contain the reference frame, the actual frame and the reconstructed frame. These image structures will be used by all the threads in the segmentation, block matching and recombination parts of the algorithm. The data structure containing all the macroblocks should also reside in this part of memory, since it also needs to be accessible by all threads.

2) Blocks and Shared Memory

For the segmentation and recombination parts of the implementation, there does not seem to be a need for the threads to be separated into blocks, since all the threads will only use specific parts of the variables which will be in global memory, as described above.

For the block matching, it is ideal that all the threads that constitute to a macroblock – search block pair reside in a block. In this block, the threads will have access to shared memory. In this shared memory, the variable which will contain the cost for a search block can exist, which for the first implementation proposed would need to be accessed by all the threads when they finish. Similarly for the second implementation, the matrix which would contain all the pixel costs would be stored here. Also, the block ID might help when it comes to identifying which is the search block with the lowest cost and what the motion vector will be.

As regards to the optimal block size, further research needs to be done so as to try and get the best balance between register size per thread and threads per block so as to make optimal use of the GPU. To tackle this problem, the profiler should aid to identify where any bottlenecks are present and what can be optimised.

QUESTION 4.) SERIAL IMPLEMENTATION

A serial implementation of the block match algorithm described above was implemented, roughly based on the following pseudocode:

- `Load the 2 Frames.`
- `Segment the 2nd frame into macroblocks.`
- `For each macroblock, run the three step search. As cost function, use the Mean Square Error.`
- `Match each macroblock with the lowest cost search block and generate the motion vectors.`
- `Reconstruct the 2nd frame using the motion vectors`

To run the implementation, the executable can be called from the terminal, with the parameters passed in the following order: block width, block height, search parameter in the vertical direction, search parameter in the horizontal direction, path were to obtain and store the frames. Note that the frames must be named frame1 and frame2 and must be in ppm format. Note also that the integer parameters passed must be non-zero.

QUESTION 5.) TESTING AND VERIFICATION

In order to verify that the algorithm is working properly, the same 3 channel frame was provided for the block matching algorithm. In this case, since there is no difference between frames, the algorithm should be able to reconstruct another image identical to the one given, since all cost functions should evaluate to 0. The result is as expected.

Another way the algorithm was tested was by supplying 2 consecutive video frames. The algorithm managed to predict quite an accurate representation of the second frame from the first frame, and blocking artefacts typical of such an algorithm were present. This test was run with different block sizes at a resolution of 720p, and the results were as expected every time. For these tests, the times for segmentation, block matching, reconstruction and the total time were recorded. These tests were run three times and the times averaged out. Table 1 contains these averaged results. Note that the segmentation, matching and reconstruction times might not add up to the total time. This may be due to factors such as overhead when performing a function call. As expected, the smaller the block size, the larger the time taken in every stage, since more macroblocks would fit in the

image. It is interesting to note however, that the difference in the reconstruction time is minimal. Other than the change in times, another observation of the different block sizes is the different blocking artefacts produced in the reconstructed images.

The algorithm was tested with the same 2 frames but at different resolutions. Each run was performed with a block size of 8x8 and a search parameter of ±8 in both the vertical and the horizontal directions. For each pair of frames at every resolution, the time to segment the data, perform the matching, reconstruct the image and the total time were taken. These test were performed three times and the averaged out, so that a more accurate representation of the results could be achieved. Table 2 contains these averaged results. As expected, although the block size remained constant, the times increased as the resolution increased, since more macroblocks are being considered if the resolution increases.

Since a three step search is being used, and only 27 comparisons are being taken into consideration, varying the search area will not lead to any differences in the time taken for the algorithm to execute. This was also tested and verified.

NB: These timings were obtained using the release version of the project. Also, the tables only show the average results. The full results can be seen in the file: Results.ods

REFERENCES

[1] A. Bovik, The essential guide to video processing. Amsterdam: Academic Press/Elsevier, 2009.

[2] E. Monteiro, B. Vizzotto, C. Diniz, B. Zatt and S. Bampi, "Applying CUDA Architecture to Accelerate Full Search Block Matching Algorithm for High Performance Motion Estimation in Video Encoding", 2011 23rd International Symposium on Computer Architecture and High Performance Computing, 2011.

[3] A. Barjatya, "Block Matching Algorithms For Motion Estimation", 2004.

[4] F. Massanes, "Compute-unified device architecture implementation of a block-matching algorithm for multiple graphical processing unit cards", J. Electron. Imaging, vol. 20, no. 3, p. 033004, 2011.

Table 1: Results for different block sizes, resolution is kept at a constant 1280x720

Block Size (pixels)	Average Segmentation Time (s)	Average Block Matching Time (s)	Average Reconstruction Time (s)	Average Total Time (s)
8x8	0.503728	4.249920	0.006839	4.785477
10x10	0.419514	3.272330	0.006897	3.754487
16x16	0.191208	1.619347	0.006217	1.827670
20x20	0.144251	1.240710	0.005767	1.405260
40x40	0.075235	0.691209	0.004598	0.778747
80x80	0.048810	0.537627	0.004214	0.594790

Table 2: Results for Different Resolutions, block size is kept at a constant 8x8

Image Resolution (pixels)	Average Segmentation Time (s)	Average Block Matching Time (s)	Average Reconstruction Time (s)	Average Total Time (s)
640x360	0.0952757	0.661972	0.001492	0.765048
848x480	0.1945147	1.716597	0.002756	1.924467
1280x720	0.504950	4.264473	0.006762	4.801567
1920x1080	1.208233	7.136793	0.016563	8.420013