A Dataflow Architecture for Real-Time Full-Search Block Motion Estimation

blind review

No Institute Given

Abstract. Motion estimation is the cornerstone of the main video compression standards, which are based on the reduction of the temporal redundancy between consecutive frames. Although the mechanism is simple, the best method, *Full Search*, uses a brute-force approach which is not suited for real-time applications. This work introduces a high performance architecture for performing on-the-fly full-search block matching estimation in FPGA devices, which has been modeled using C++ programming language and synthesized with Vivado HLS for a Xilinx ZC706 prototyping board. The architecture is based on a dataflow datapath and it is configurable, enabling a fast and easy exploration of the solution space. On-board results achieve a maximum performance of 743fps, 247fps and 110fps for VGA, HD and FHD video resolutions, respectively, for a typical macroblock size of $16x16$ pixels and a search area of ± 16 pixels.

Keywords: FPGA, High-Level Synthesis, Motion Estimation, Full Search Block Matching

1 Introduction

Motion Estimation (ME) explores temporal redundancy of a sequence, which is inherent in consecutive video frames, and it represents a basis for lossy video compression. Thus, ME techniques are present in standard video codecs, such as H.263, H.264, or HEVC, to exploit temporal redundancy in a sequence of frames. However, the relentless growth in digital video applications and their continuous resolution improvement make the ME phase a critical one; the total computing time varies between 60% and 80% depending on the Block Matching (BM) algorithm selected, which also determines the resulting accuracy reached, e.g. the Full-Search Block Matching (FSBM) algorithm gets the most accurate results since it performs all possible comparisons of macroblock, i.e. a group of NxN pixels.

The FSBM algorithm has the aim to get the best match for a block in the reference frame. In this sense, the current block is compared with the candidate blocks of the reference frame. FSBM calculates the Sum-Absolute-Difference (SAD) value at each possible location inside the search window [1]. Unfortunately, the computational costs of the FSBM algorithm is prohibitive on general purpose processors, because it is extremely demanding concerning the computing effort, rendering this technique unsuitable for real-time imaging applications. Contrarily, FPGA (*Field Programmable Gate Array*) architecture is well-suitable for this algorithm because of its parallel nature.

Therefore, increasing the performance of the FSBM algorithm in parallel architecture devices, such as FPGAs, can reduce the computational costs as well as be feasible in real-time solutions. In this work, a high-performance architecture for Full-Search Block-Matching Estimation is introduced, with the following main contributions.

– A configurable hardware accelerator of the FSBM algorithm for different video resolution, i.e. VGA, HD and Full HD.
– Reduce the footprint of the proposed solution as low as possible to enable its deployment on embedded FPGA-based platforms with constraint resource (i.e. edge/fog computing nodes) or power (i.e. energy harvesting or battery powered) budgets.

The rest of this paper is organized as follows. In Section 2, the related works are analyzed. Section 3 describes the architecture and the proposed workflow for the FSBM implementation. Section 4 discusses the strengths and limitations of the proposed solution as well as compares the results with the different configurations implemented. Finally, Section 5 draws the main conclusions of this work.

2 Related work

S. Ghosh et al. in [2] present an FPGA-based solution for the sum-of-absolute-differences (SAD) operator of the FSBM algorithm, in which the main optimization done is the reduced number of basic arithmetic operations performed. This optimization results in a the performance as well as reduces the silicon area. In the same line, H. Loukil et al. describe a VHDL implementation of the SAD operator in [3]. The optimizations of both works get good results for isolated macroblocks, but the scalability of these solutions do not work fine when a whole frame is processed. The arrangement pattern of the pixels would require extra bandwidth to access the memory in the required order.

J. Olivares et al. [4] apply the Online Arithmetic (OLA) that allows to speed up computation by early termination of the SAD calculation when the candidate is bigger than the current reference. The accelerator is optimized for area and is able to process 17.2 VGA frames per second on a Virtex-II with a 425 MHz clock for 16x16 macroblocks.

Furthermore, 1-D and 2-D Systolic array architectures for FSBM implementation have been regarded as an optimal solution due to the efficient use of resources, low power budget and their configurability properties in what concerns the macroblock dimension, the search area and parallelism level [5] [6].

M. Mohammadzadeh et al. present in [7] an array processor written in VHDL and its optimization to achieve the minimum area occupation and maximum operating frequency. It includes control logic blocks to generate memory access addresses, which makes this work one of the few approaching the whole process at frame level, together with the proposal presented by Kasturi et al. in [8]. Implementation results are shown for a size of macroblock of 8x8 and different values for the search area. For the same configuration that the one set in this work ($N = 16$ and $p = 8$), the synthesis results show that the area occupied on a Vritex-II FPGA is about %11 of the chip and the maximum frame rate is 60 for CIF resolution and a 191 Mhz clock. It is not possible to project these results for other video formats.

Nuno et. al [9] validate several optimization strategies to obtain even more efficient systolic array architectures without sacrificing the quality of the result. Some of these techniques (i.e. reduced pixel precision) are of interest and its application to the solution proposed in this work is planned for future improve versions of the FSBM IP.

3 Full-Search Block Matching implementation on FPGA

For the design of the FSBM architecture, a dataflow approach has been followed, avoiding, as much as possible, the use of unnecessary intermediate memory storage. The simplicity of the HLS model and the optimization of the computations adds together to obtain and optimal outcome.

The C++ model defines four stages that operate concurrently (see Figure 1):

- ***Stage 1:*** Accommodation of the input streams. On the one hand, the input reference frames must have their borders extended in order to keep subsequent processing logic simple. On the other hand, the input reference frame must be rearranged in order to deliver the right MB sequence to next stage, at the right time.
- ***Stage 2:*** For each MB of the reference frame, its similarity degree with all possible MBs within the search region is computed. The similarity function implemented in this proposal is the *Sum of Absolute Differences* (SAD).
- ***Stage 3:*** Selection of the MB in the reference image with the minimum value for previously computed SAD costs.
- ***Stage 4:*** Copy of the computed motion vector values to external memory.

Listing 1.1 shows the proposed model implemented in C++ to be synthesized with Vivado HLS tool for a block size of $N = 16$ and a search area of 8 pixels, the simplest model since it is only necessary to cache one line of reference macroblocks, and one macroblock from the current frame has to be compared with. In such code *INTERFACE #pragmas* are not shown, but all input/output ports are mapped to AXI-Stream channels. The model has also been fully parameterized by means of *#define* preprocessor directives, enabling easy adaptation by only changing the values for the video resolution and macroblock size.

Listing 1.1: HLS model for the FSBM dataflow architecture.

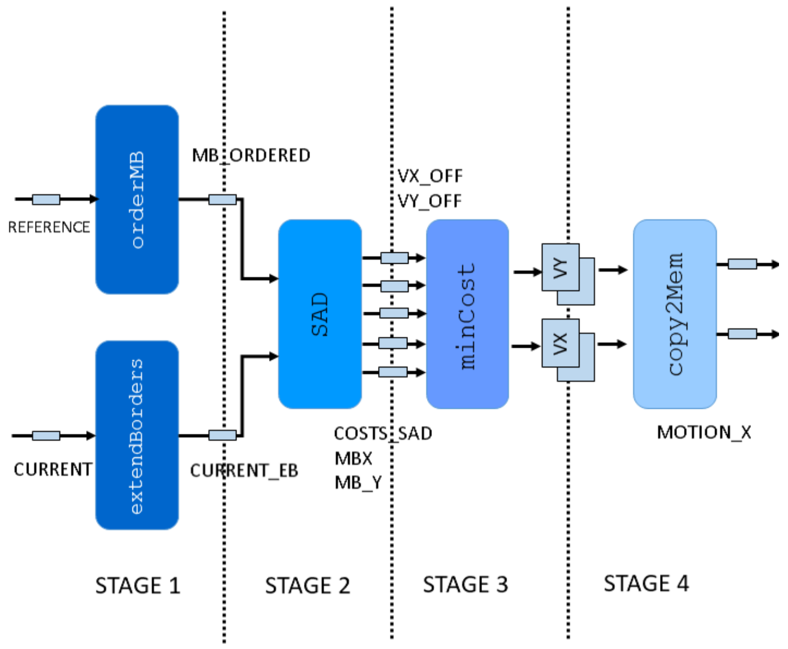

Fig. 1: Proposed dataflow architecture for the FSBM IP.

```
void FSBM(AXI_STREAM& IMG_REF, AXI_STREAM& IMG_CURRENT, MB_OFFSET_T
     MOTION_X[V_MB][H_MB], MB_OFFSET_T MOTION_Y[V_MB][H_MB]){
...
    YUV_IMAGE_T CURRENT;
    MB_OFFSET_T VX[H_MB];
#pragma HLS STREAM variable=VX off
        MB_X_T MB_X[NPIXELS_IMG/PIXELS_WORD];
#pragma HLS STREAM variable=MB_X depth=2 dim=1

#pragma HLS dataflow
        //Stage 0: Xilinx HLS Video data types
        hls::AXIvideo2Mat(IMG_REF, REFERENCE);
        hls::AXIvideo2Mat(IMG_CURRENT, CURRENT);
        //Stage 1
        extendBorders(CURRENT,CURRENT_EB);
        orderMB(REFERENCE,MB_ORDERED);
        //Stage 2
        SAD(CURRENT_EB,MB_ORDERED,COSTS_SAD,MB_X,MB_Y,VX_OFF,VY_OFF);
        //Stage 3
        minCost(COSTS_SAD,VX_OFF,VY_OFF,MB_X,MB_Y,VX,VY);
        //Stage 4
        copy2Mem(VX,VY,MOTION_X,MOTION_Y);
    return;
}
```

All stages communicate by means of streams, implemented as FIFO channels, avoiding the need of large intermediate ping-pong buffers that would require of BRAM resources. The only exception to this rule are the two ping-pong buffer channels between *Stage 3* and *Stage 4* (VX and VY). These memories are used to store the values of the motion vectors computed so far in *Stage 3*. Thus, for the video resolutions and a macroblock size of 16x16 (the one used in this work during the experimental validation of the FSBM core), the dimensions of the motion vector matrices are 40x30 (VGA), 80x45 (HD) and 120x68 (FHD) 4-bit elements. The BRAMs used by Vivado HLS to map the model variables are doubled since the channel implements ping-pong synchronization. In addition, an initialization stage has been included to convert the source data into Xilinx HLS video data type (lines 10-12 of Listing 1.1).

In order to keep a sensible use of FPGA resources, such as BRAMs, the depth of the FIFO channels are set to the smallest possible value due to the balanced design of the stages. The width and depth of some of the channels depend on the width of the AXI-Stream interface to external memory and the resolution of the images.

Concerning the internal demand for resources by the individual modules, the focus is put in avoiding the need for unnecessary intermediate storage. The majority of such demand comes from the `orderMB` and `SAD` functions that implement two line buffers of size N and, for the latter, an additional buffer that holds one, three or five rows of macroblocks, depending on the search area configures (i.e. 8, 16 or 32 pixels).

Next, a complete breakdown of the dataflow architecture is provided so as to give the reader with details about the functioning of each stage, the synchronization mechanism between steps and the strategy to achieve the aimed reduction of resources in the final implementation of the model.

3.1 Stage 1: Input Stream Accommodation

This step is responsible for the preparation of the video frames before the computation of the SAD values. In addition to the use of built-in Xilinx functions (i.e. *hls::AXIVideo2Mat*) so as to be compliant with the HLS utility video data types (lines 11 and 12 of Listing 1.1), two additional actions are carried out.

The general operation of the FSBM algorithm consists in finding the most likely position of a reference macroblock in the current frame within a delimited search area. Macroblocks placed at the borders of a frame are a special case since there are search positions that might not be valid because they are out of the frame area. This situation can be detected by checking the limits before proceeding to compute the cost function.

However, in order to ease the work of the HLS tool, the borders of the input reference frame are extended. This way, the implementation of the SAD function becomes a perfect loop, leading to an optimal pipeline architecture. This is the task of the `extendBorders` function which replicates the pixels placed at the borders of the current frame and fills out the empty search area. To this end, only a buffer of the size of a line is needed to temporally save the pixels of the first and last rows. The output of the `extendBorders` function is a stream (`CURRENT_EB`) of 32-bit words, each one packing four 8-bit luminance components.

Meanwhile, running in parallel to the adaptation of the current frame, the `orderMB` function re-arranges the pixels of the reference frame and feeds the second stage with an ordered flow of MBs. To this end, a line buffering approach is implemented which optimizes the number of BRAMs needed. Figure 2 represents the architecture of the line buffering approach used by the FSBM component, in which the dataflow follows a pull-push model. Firstly, the pixels of the concerned column/s are shifted upwards at each iteration, so as to leave free space. At the same time, those pixel/s are copied to the `lastc` array which holds the values for the current column under processing. Secondly, the new pixel, or group of pixels (depending on the width of the AXI-S interface), are received and stored. Finally, the window is shifted to the right and `lastc` is inserted. The content of the window is processed by the user logic anytime it is necessary, which is usually modelled by a guarded condition that is evaluated in each iteration.

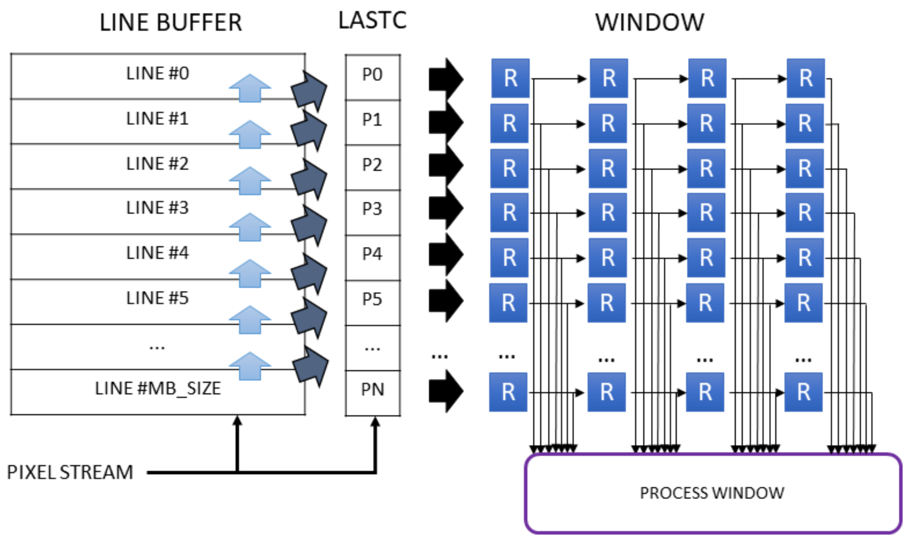

Fig. 2: Line buffering approach used in Stages 1 & 2.

The user logic in `orderMB` packs the 16x16 8-bit pixels of the window in one 2048-bit word. This only happens when `row` and `col` loop indexes are multiple of the macroblock size (lower-right corner). Therefore, the packed word actually represents the content of a macroblock in the reference frame.

3.2 Stage 2: SAD computation

In this stage, each macroblock of the reference frame is compared with all possible adjacent macroblocks within a delimited search area in the current frame. The correspondence of the reference macroblock in the current frame is established by selecting the current macroblock with the minimum cost (similarity) function. In this implementation, the *Sum of Absolute Differences* (SAD) is used.

The challenge is to perform all SAD computations as the pixels of the current (border-extended) frame are received. To this end, the implementation of the `SAD` function uses a similar line buffer structure than the one used in the `orderMB` function in Stage 1. However, it is introduced a new challenge derived from the need of synchronized the stream of the current frame with extended borders and the ordered list of MBs of the reference frame.

Figure 3 sketches the synchronization mechanism and the macroblock consumption pattern. The picture represents a simplified version of the extended current and reference (light blue background) frames which are overlapped. In this example, each cell represents a 8x8 pixel sub-block and this representation assumes a macroblock size of 16x16 pixels and a search area of ±8 pixels. The reference macroblocks are consumed only at the instants represented as black diamonds, filling the `MB` buffer which has a size equals to one row of macroblocks (steps a to d) in Figure 3).

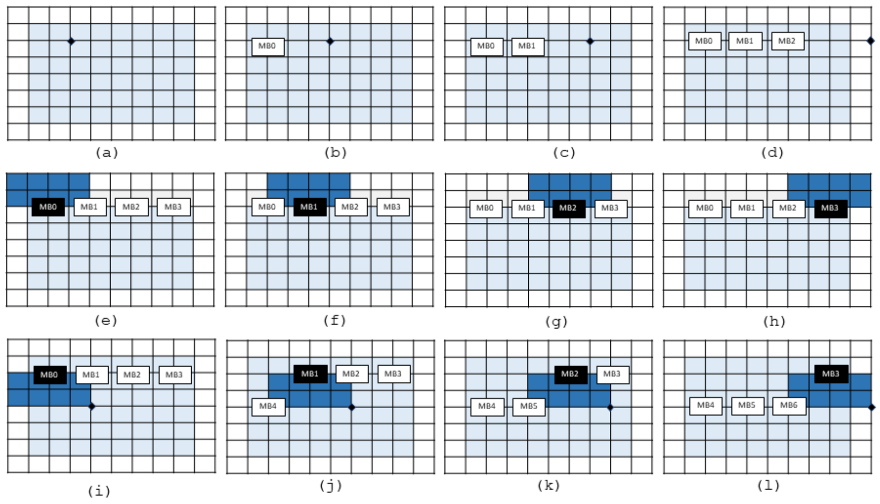

Fig. 3: Reference macroblock consumption pattern and SAD computing areas.

The SAD cost values for each reference macroblock are computed in two phases; upper half (steps e to h) and lower half (steps i to l). After the finalization of the second phase, the macroblock is replaced by a new one since it is not needed anymore. The dark blue cells represent, for the active reference macroblock (highlighted as a black box), the actual search area processed. This mechanism is also followed for values ±16 and ±32 pixels of the seach area. The only difference is that it must be stored three and five lines of reference macroblocks, respectively.

Furthermore, this stage also obtains the SAD cost by the `costSAD` function, which is responsible for the actual computation of the macroblock similarity metric. This function must accept a new window of pixels every clock cycle in order to keep the pace set by the dataflow path. Since `CURRENT_EB` stream has a data width of 32-bits, at each iteration of the pipeline four new luminance components are accepted. This means that, at each iteration, four SAD cost values have to be computed; otherwise, three columns of pixels will be lost due to the shifting of the processing window.

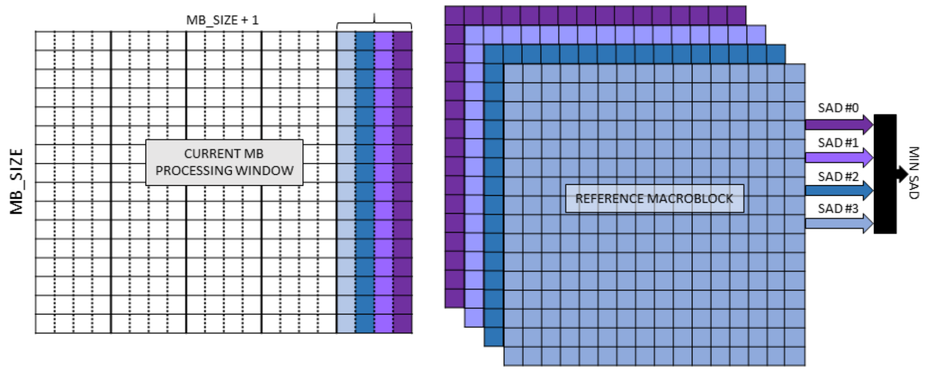

Fig. 4: Parallel computation of SAD cost values for the extended processing window.

Figure 4 represents this functionality. The processing window is extended one word (last coloured column) to avoid the above mentioned loss of pixels. In HLS, the function `costSAD` (see Listing 1.2) implements this task. Since both the processing window and the reference macroblock are mapped to registers, all read operation are completed in one clock cycle. Then, for each SAD cost being computed, the absolute differences for each group of pixels (4 in total) are added (line 17 of Listing 1.2). The automatic expression balancing performed by Vivado HLS produces the minimum latency for the tree of additions.

Listing 1.2: SAD cost computation.

```
 1 typedef ap_uint<2048> MBreg_t; //For 16x16 MB size
 2
 3 ap_uint<2> costSAD(pixel_t MB_curr[W_SIZE], MBreg_t MB_ref, COST_SAD_T *
       cost, MB_OFFSET_T offset){
 4 #pragma HLS INLINE off
 5  ap_uint<2> sad_select0, sad_select1, sad_select;
 6
 7  coste_MAD_i:for(i=0; i<MB_SIZE; i++){
 8   coste_MAD_j: for(j=0; j<MB_SIZE/4; j++) {
 9    p1R = MB_ref((i*MB_SIZE+j*4)*8+7,(i*MB_SIZE+j*4)*8);
10    p2R = MB_ref((i*MB_SIZE+j*4)*8+15,(i*MB_SIZE+j*4)*8+8);
11    //p3R, p4R
12
13     p1C = MB_curr[i*((MB_SIZE/4)+1)+j](7,0);
14     p2C = MB_curr[i*((MB_SIZE/4)+1)+j](15,8);
15     // p3C, p4C, p5C, p6C, p7C
16 
17     SAD_0 += abs(p1R-p1C)+abs(p2-p2C)+abs(p3R-p3C)+abs(p4R-p4C);
18     //SAD_1, SAD_2, SAD_3
19   }
20  }
21 //1st Cycle
22  minSAD0 = SAD_0;
23  sad_select0 = 0;
24 
25  if (SAD_1 < SAD_0)
26   minSAD0 = SAD_1;
27   sad_select0 = 1;
28  } else if (SAD_0 == SAD_1 {
29     if (offset < MB_SIZE/2) {
30      minSAD0 = SAD_1;
31      sad_select0 = 1;
32    }
33  }
34 //Idem for SAD_2 and SAD_3
35 //2nd Cycle
36  *costSAD = minSAD0;
37  sad_select = sad_select0;
38 
39  if (minSAD1 < minSAD0) {
40   *costSAD = minSAD1;
41    sad_select = sad_select1;
42  } else if (minSAD0 == minSAD1){
43   if (offset < MB_SIZE/2) {
44     *costSAD = minSAD1;
45     sad_select = sad_select1;
46   }
47  }
48  return mb_select;
49 }
```

The `offset` parameter provided to `costSAD` function represents the relative position of the reference macroblock within the search area. This parameter is used, in case of equality of the SAD values, to select the macroblock closer to the center (lines 25-47 of Listing 1.2). This design decision has been made based on the high frame rate supported by the FSBM IP and the nature of the video sequences (global motion) that would lead to small macroblock displacements between frames. As a result, the minimum SAD value is selected and its index (`sad_select`) is returned. Back to the parent `SAD` module, the SAD selection variable adjusts the X component of the motion vector (lines 29 and 33 of Listing 1.2).

Depending on the search area parameter, the number of instances of the `costSAD` function takes a value of 1, 9 or 25 in the architecture.

3.3 Stage 3: Selection of the minimum SAD cost

This stage receives the output of the previous SAD computation phase. For a reference macroblock (`MB_X, MB_Y`), a new intermediate cost (`COST_SAD`) needs to be processed and check whether it is the minimum for that macroblock. If so, the current value of the motion vector is updated with the `VX_OFFSET` and `VY_OFFSET` components.

The `minCost` function initializes the `MIN_COSTS` matrix with the maximum possible value for SAD cost. Then, it starts the processing comparing the current minimum with the value took out off the input stream. In case that both SAD costs are equal, the one belonging to the current macroblock closest to the center of the search area is selected (lines 23-31). A ROM is used to stored the pre-computed distance values in order to avoid higher latencies due to extra arithmetic operations.

As in the previous stages, only a well-known subset of the positions in `VX` and `VY` arrays are accessed for a macroblock in the current frame given a specific search area. Therefore, the bottleneck that would represent the `minCost` function when several instances of the `costSAD` functions are created, is solved by maintaining in registers only those positions that could be potentially affected by the arrival of a new minimum.

By the end of this phase, the `VX` and `VY` ping-pong buffer channels have the final values of the motion vectors for the pair of frames under processing. The `copy2Mem` function (Stage 4) only reads these memories and packs up to four components (depending on the specified width of the AXI-Stream interface) in a single memory word.

4 Experimental results

The main design goal of the proposed FPGA-based implementation of the FSBM algorithm is to perform the computation of the motion vectors for a sequence of frames with the maximum throughput, allowing on-the-fly analysis of the video flow directly from the input interface. To this end, a prototype of a video processing platform has been developed on a Xilinx ZC706 board see Figure 5) so the actual performance of the proposed architecture can be measured.

The video stream is captured in YUV 4:2:2 (16 bit/pixel) format by a Digilent FMC-HDMI board and then stored in the DDR memory by a Video DMA component (VDMA0). Two Video DMA cores, VDMA0 and VDMA1, feed the FSBM component with the right pair of frames to be processed at each step, whilst the output of the FSBM, i.e. motion vectors, is stored in the DDR through the second Video DMA component (VDMA1).

VDMA synchronization is achieved by a combination of hardware and software mechanisms. On the hardware side, a *Dynamic* approach is followed [10], with VDMA0 playing the role of the master and VDMA1 the role of slave. VDMA0 singals VDMA1 when a new frame has just been written in memory so VDMA1 does not step on frames that have not been processed yet. On the software side, a circular framebuffer is set, with VDMA0 pointing to the frame labeled as `current` and VDMA1 pointing to the one labeled as `reference`.

The FSBM core has been developed in C++ and then synthesized, packed and deployed using the Vivado 2019.2 toolchain provided by Xilinx. The high-level model is fully parameterized, allowing the possibility of varying the video resolution (VGA, High Definition, and Full HD), and the width of the AXI-Stream interfaces (16 or 32 bits) to access the memory where the frames are

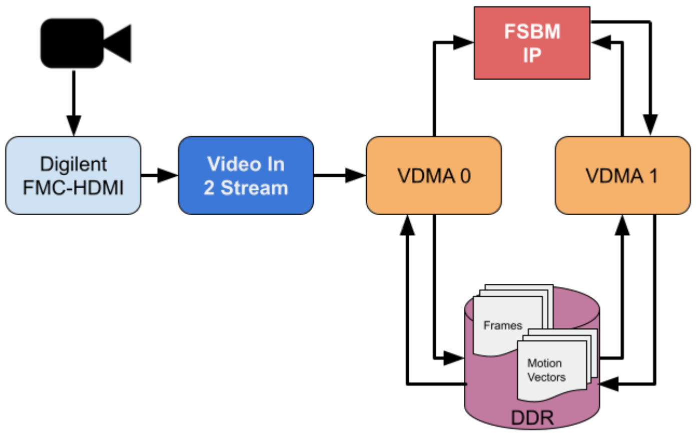

Fig. 5: Video processing platform developed to test the FSBM IP.

stored. Also, the FSBM core can instantiate three versions of the `costSAD` function, which perform 1, 2 or 4 (see Figure 4 in Section 3.2) macroblock comparisons over the same window. The version of the core used in the prototype analyzed in this section has fixed the size of the macroblock to $N = 16$ and the search area parameter to ±16 and ±32.

Table 1: Nominal frame rate (ZC706 board, $\overline{T}$=8.69ns±2.23%). Best configuration: 32-bit AXI-Stream interface and 4-pixel `costSAD` function.

	SEARCH AREA		
INPUT VIDEO	**8**	**16**	**32**
VGA	965	743	371
HD	322	247	124
FHD	143	110	55

Regarding the performance achieved by our design, Table 1 shows the nominal frames per second (fps) for the most performing configurations. That is, the use of a 32-bit AXI-Stream bus and the version of the `costSAD` function that process four new pixels in parallel. Synthesis results prompted a slight variation in the clock period for the proposed FSBM IP as the model parameters where modified. Thus, it is safe to say that the average period (8.69ns $\pm 0.19 ns$) is constantThe nominal working frequency (≃115 Mhz) is enough to perform real-time processing of a 60Hz video stream for VGA and HD resolutions, with a combination of IP parameters that ensures the minimum occupancy of the FPGA.

For Full HD resolutions, the core cannot meet the pixel time (i.e. 148.5Mhz for 60Mhz), so the designer might overcome this limitation by selecting a more aggressive combination of parameters to meet real-time requirements. This approach also applies to contexts where the performance is the principal driving force in the design.

The comparison with other works is not straightforward. Whilst our solution comprises the whole FSBM algorithm, most of the related proposals focus only on the implementation of the SAD operand. A comprehensive approach introduces additional challenges that are not taken into account such as the efficient management of the traffic to/from memory through reutilization of data.

Table 2: Summary of other FPGA-based implementations of the FSBM algorithm and comparison with the proposed architecture.

Work	Platform	Clock (MHz)	MB size	Search Area	Resolution	FPS
[4]	Xilinx Spartan3	366.8	16x16	±16	HD	13.62
[3]	Altera Stratix	103.8	16x16	±16	HD	5.15
[6]	Xilinx XCV3200E	76.1	16x16	±16	HD	20.98
[11]	Altera Flex20KE	197	16x16	±16	HD	4.91
[2]	Xilinx Virtex 4	221.2	16x16	±16	HD	55.33
[7]	Xilinx Virtex 2	191	16x16	±16	HD	2.09
[12]	Xilinx Virtex 5 LX330T	125	8x8,16x6, 32x32,64x64	±64	FHD	26.9
	Virtex 6 LX240T					53.6
[13]	Xilinx Virtex 5	269.3	16x16	±32	FHD	31
						30
Ours	**Xilinx ZCU706 (Kintex-7)**	**115**	**16x16**	±16	**HD**	**247**
					FHD	**110**
				±32	**HD**	**124**
					FHD	**55**

Table 2 lists information on certain implementation results of other FPGA-based implementations of the FSBM algorithm. The use of RTL as the developing language is common in all them whereas in this proposal, HLS technology has proved to be competitive in terms of performance. No developing and testing times have been reported in any of these works, so it is not possible to establish a comparison framework in this regard, which is one the major pluses of HLS tools.

The approach presented in this article surpasses the performance levels reported by those works with the same configuration (i.e. macroblock size, search area and resolution of the video), despite running at a lower clock frequency. Also, the reader should notice that our approach also deals with the burden of moving efficiently the frames from/to the DDR memory.

Only D'huys et al. in [12] faced the implementation of a global solution, where it is also employed a macroblock reordering strategy to optimize the computation of the SAD costs. The comparison with this work is not direct since the solution is intended for a variable-size macroblock problem ranging from $N = 8$ to $N = 64$. Also, the search area is wider so the computational load increases significantly. Although current results look promising, it would be necessary revisiting the architecture to adapt it to this more challenging scenario.

Nevertheless, the comparison with [13] is more direct since the size of the block remains the same.

Table 3: Resource occupation after place & route (XC7Z045-FFG900-2 ZynQ-SoC). Best configuration: 32-bit AXI-Stream interface and 4-pixel `costSAD` function.

Search Area	LUT	FF	18 Kbit BRAM
	VGA		
8	30720 (14,05%)	22745 (5,20%)	48 (4,40%)
16	64304 (29,42%)	52145 (11,93%)	58 (5,32%)
32	131472 (60,14%)	110945 (25,38%)	68 (6,24%)
	HD		
8	30680 (14,03%)	24176 (5,53%)	53 (4,86%)
16	64264 (29,40%)	53576 (12,25%)	71 (6,51%)
32	131432 (60,12%)	112376 (25,7%)	90 (8,26%)
	FHD		
8	31548 (14,43%)	26880 (6,15%)	60 (5,50%)
16	65132 (29,8%)	56280 (12,87%)	89 (8,17%)
32	132300 (60,52%)	115080 (26,32%)	117 (10,73%)

The impact on resource utilization, for different variations of the search area, is summarized in Table 3, where actual synthesis results (after place & poute) are shown. Firstly, the demand for resources remains stable despite the fact of dealing with higher resolutions of the video feed. This is specially true for LUTs and FFs. However, such demand slightly sees a moderate increment for BRAM resources. Secondly, the wider the search area the higher the increment in resource usage, specially in LUTs and BRAMs, due to the need to cache a greater number of macroblocks for concurrent `costSAD` computation.

5 Conclusions

In this work, a high performance implementation of the full search block matching motion estimation algorithm for real-time video processing has been presented. The proposed architecture is based on a dataflow and has been modeled using Vivado HLS. The model is parameterized, enabling the engineer to easily explore the solution space and select the combination of variables (video resolution, width of memory interface, number of parallel SAD computations) that best serve the design requirements.

A video processing platform has been also prototyped and synthesized on a mid-range, cost-optimized Xilinx XC7Z020-CLG484-1 ZynQ-SoC in order to perform actual measurements of the performance of the FSBM core, and verify its functioning. The core runs at 115 Mhz which is enough to cope with VGA and HD 60Hz video timings with the minimum usage of resources. Real-time Full HD video resolutions can also be processed by means of increasing the memory bandwidth and parallel SAD cost computations, reaching a maximum of 110 fps.

Several optimizations and potential improvements are devised for future versions of the component. The reduction of the LUT resources has the highest priority since represents 60% of the total available. Also, alternative versions of the cost function will be explored, which might help to the rationalization of the FPGA resources (e.g. implementing the Mean of Absolute Differences would reduces the number of bits necessary to represent the cost), and variations on the technique to extend the borders of the current frame.

Acknowledgment

References

1. L.C. Manikandan, S.A.H. Nair, K.S., Selvakumar, R.: Efficient and configurable full-search block-matching processors. Cluster Computing **22**(5) (Sep 2019) 11773–11780
2. Ghosh, S., Saha, A.: Speed-area optimized fpga implementation for full search block matching. In: 2007 25th International Conference on Computer Design. (Oct 2007) 13–18
3. Loukil, H., Ghozzi, F., Samet, A., Ben Ayed, M.A., Masmoudi, N.: Hardware implementation of block matching algorithm with fpga technology. In: Proceedings. The 16th International Conference on Microelectronics, 2004. ICM 2004. (Dec 2004) 542–546
4. Olivares, J., Hormigo, J., Villalba, J., Benavides, I., Zapata, E.: Sad computation based on online arithmetic for motion estimation. Microprocessors and Microsystems **30**(5) (2006) 250 – 258
5. Ryszko, A., Wiatr, K.: An assessment of fpga suitability for implementation of real-time motion estimation. In: Proceedings Euromicro Symposium on Digital Systems Design. (Sep. 2001) 364–367
6. Roma, N., Sousa, L.: Efficient and configurable full-search block-matching processors. IEEE Transactions on Circuits and Systems for Video Technology **12**(12) (Dec 2002) 1160–1167
7. Mohammadzadeh, M., Eshghi, M., Azadfar, M.M.: Parameterizable implementation of full search block matching algorithm using fpga for real-time applications. In: Proceedings of the Fifth IEEE International Caracas Conference on Devices, Circuits and Systems, 2004. Volume 1. (Nov 2004) 200–203

8. Rangan, K., Reddy, M., Reddy, V.: A fpga-based architecture for block matching motion estimation algorithm. In: IEEE TENCON 2005. Volume 2007. (11 2005) 1–5
9. Roma, N., Dias, T., Sousa, L.: Customisable core-based architectures for real-time motion estimation on fpgas. In Y. K. Cheung, P., Constantinides, G.A., eds.: Field Programmable Logic and Application, Berlin, Heidelberg, Springer Berlin Heidelberg (2003) 745–754
10. Inc., X.: Xilinx axi video direct memory access v6.2 product guide (pg020). (2016) 39
11. Wong, S., Vassiliadis, S., Cotofana, S.: A sum of absolute differences implementation in fpga hardware. In: Proceedings. 28th Euromicro Conference. (2002) 183–188
12. D'huys, T., Momcilovic, S., Pratas, F., Sousa, L.: Reconfigurable data flow engine for hevc motion estimation. In: 2014 IEEE International Conference on Image Processing (ICIP). (2014) 1223–1227
13. Asano, S., Shun, Z.Z., Maruyama, T.: An fpga implementation of full-search variable block size motion estimation. In: 2010 International Conference on Field-Programmable Technology. (2010) 399–402