Utah State University
DigitalCommons@USU

All Graduate Theses and Dissertations Graduate Studies

12-2009

Memory Architecture Template for Fast Block Matching Algorithms on Field Programmable Gate Arrays

Shant Chandrakar
Utah State University

Follow this and additional works at: https://digitalcommons.usu.edu/etd

Part of the Data Storage Systems Commons

Recommended Citation

Chandrakar, Shant, "Memory Architecture Template for Fast Block Matching Algorithms on Field Programmable Gate Arrays" (2009). *All Graduate Theses and Dissertations*. 495.
https://digitalcommons.usu.edu/etd/495

This Thesis is brought to you for free and open access by the Graduate Studies at DigitalCommons@USU. It has been accepted for inclusion in All Graduate Theses and Dissertations by an authorized administrator of DigitalCommons@USU. For more information, please contact digitalcommons@usu.edu.

MEMORY ARCHITECTURE TEMPLATE FOR FAST BLOCK MATCHING ALGORITHMS ON FIELD PROGRAMMABLE GATE ARRAYS

by

Shant Chandrakar

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Engineering

Approved:

Dr. Aravind Dasu Major Professor	Dr. Brandon Eames Committee Member
Dr. Jacob Gunther Committee Member	Dr. Byron R. Burnham Dean of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2009

Copyright © Shant Chandrakar 2009

All Rights Reserved

Abstract

Memory Architecture Template for Fast Block Matching Algorithms on Field Programmable Gate Arrays

by

Shant Chandrakar, Master of Science

Utah State University, 2009

Major Professor: Dr. Aravind Dasu
Department: Electrical and Computer Engineering

Fast Block Matching (FBM) algorithms for video compression are well suited for acceleration using parallel data-path architectures on Field Programmable Gate Arrays (FPGAs). However, designing an efficient on-chip memory subsystem to provide the required throughput to this parallel data-path architecture is a complex problem. This thesis presents a memory architecture template that can be parameterized for a given FBM algorithm, number of parallel Processing Elements (PEs), and block size. The template can be parameterized with well known exploration techniques to design efficient on-chip memory subsystems. The memory subsystems are derived for two existing FBM algorithms and are implemented on a Xilinx Virtex 4 family of FPGAs. Results show that the derived memory subsystem in the best case supports up to 27 more parallel PEs than the three existing subsystems and processes integer pixels in a 1080p video sequence up to a rate of 73 frames per second. The speculative execution of an FBM algorithm for the same number of PEs increases the number of frames processed per second by 49%.

(57 pages)

To my family, especially to my late grandmother.

Acknowledgments

I would like to thank Dr. Aravind Dasu for giving me an opportunity to work as a research assistant under his guidance. This task would have never been possible without his encouragement, patience, and expertise in the field of video compression. I would also like to thank my committee members, Dr. Brandon Eames and Dr. Jacob Gunther, for extending their support. I thank Abraham A. Clements for his astonishing ideas and the algorithm for proving the authenticity of this project, Arvind Sudarsanam for providing technical inputs and reviewing my work, Harikrishna Samala for helping me with the documentation, and Jonathan Phillips for mentoring me during the initial days of my research. Last, but not least, I thank my friends Ram, Varun, Jimmy, Manas, Ankur, and my other roommates for always being with me during the toughest times of my graduate studies. I also thank David Sant for providing a wonderful research facility in form of the Sant Innovation building. I thank my parents and my sister for their sacrifices, support, and patience during the entire course of my study.

Shant Chandrakar

Contents

Page

Abstract . . . . **iii**

Acknowledgments . . . . **v**

List of Tables . . . . **vii**

List of Figures . . . . **viii**

Acronyms . . . . **ix**

1 Introduction **1**

2 Background and Related Works **3**
- 2.1 Field Programmable Gate Array 3
- 2.2 Motion Estimation 6
- 2.3 Existing Memory Subsystems 8

3 Architecture Template **11**
- 3.1 Frame Buffer 13
  - 3.1.1 Load Scheme 17
  - 3.1.2 Fetch Scheme 17
- 3.2 Pixel Load Unit 18
- 3.3 Pixel Fetch Unit 21
- 3.4 Pixel Select and Rearrangement Module 22
  - 3.4.1 Reference Frame Selector 23
  - 3.4.2 Current Frame Rearranger 24
  - 3.4.3 PSRM Controller 24

4 Proof of Conflict Free Parallel Access **26**

5 Performance Estimation Model **29**

6 Bounded Set Algorithm **31**

7 Results **36**

8 Conclusion and Future Works **43**
- 8.1 Conclusion 43
- 8.2 Future Works 44

References **46**

List of Tables

Table Page

2.1 Summary of existing memory subsystems. 10

7.1 Comparison of proposed memory subsystem with other memory subsystems for a single PE. 37

7.2 Maximum number of PEs supported by the proposed and existing memory subsystems on a Virtex-4 LX160 FPGA. 40

7.3 Operating frequency of PLU and memory subsystem for different block sizes and FBM algorithms. 41

7.4 Number of frames processed per second (fps) by the proposed memory subsystem through normal and speculative execution of FBM algorithms. . . . 41

List of Figures

Figure Page

2.1 FPGA architecture. 4

2.2 Implementation of a Boolean function using four-input look-up table. 4

2.3 Xilinx FPGA design flow. 6

2.4 Motion estimation. 8

3.1 Overview of the FPGA system featuring the proposed memory architecture template. 13

3.2 Example illustrating the fundamental terms. 14

3.3 Frame buffer. 16

3.4 Addressing scheme for (a) load operation, (b) fetch operation on a RFB, and (c) shows SB[1] being enclosed by the USB of fig. 3.2. 19

3.5 Rearrangement of external pixels and generation of Col_LS signal by pixel load unit for a frame buffer with $B_w \times A_w \geq 8$. 20

3.6 Buffering of external pixels and their shifting by a pixel load unit for loading pixels into the frame with $B_w \times A_w < 8$. 22

3.7 The assignment of column of pixels in D_R as input to the multiplexers of the column selector. 24

3.8 Assignment of columns of D_R to the multiplexers of column selector for $B_w^R = 6$, $A_w^R = 1$, and $p = 2$. 25

7.1 Illustration of speculative execution in UMHexagonS. 39

Acronyms

2DAM	2-dimensional addressable memory
ASIC	application specific integrated circuit
BRAM	block random access memory
BS	bounded set
CFB	current frame buffer
CFR	current frame rearranger
CLB	configurable logic block
CODEC	en*CO*der/*DEC*oder
DDR2	double data rate
FBM	fast block matching
FPGA	field programmable gate arrays
fps	frames per second
I/O	input/output
IP	Intellectual Property
ISE	integrated software environment
K	kilo
LUT	four-input look-up table
MDHC	mixed diamond hexagon and cross
ME	motion estimation
MPEG	moving picture experts group
NGD	native generic data
PE	processing element
PFU	pixel fetch unit
PLU	pixel load unit
PSRM	pixel select and rearrangement module

RAM	random access memory
RFB	reference frame buffer
RFS	reference frame selector
ROM	read only memory
SAD	sum of absolute differences
SB	super block
SDRAM	synchronous dynamic random access memory
SM	switch matrix
TBR	tightest bounding rectangle
USB	union of super blocks
UMHexagonS	umsymmetrical-cross multi-hexagon-grid search
VHDL	very high speed integrated circuits hardware descriptive language

Chapter 1

Introduction

Motion Estimation (ME) is one of the most compute intensive tasks in digital video CODECs such as MPEG-2, MPEG-4, H.264, etc. [1–5], because it operates on a block of pixels and requires significant data throughput. In the past decade, various Fast Block Matching (FBM) algorithms [1,2] have been proposed to reduce the computational demands of ME at the cost of an acceptable degradation in image quality. An FBM algorithm essentially uses a sequence of search steps/patterns to fetch small blocks of pixels from two or more video frames and calculate an error metric/distortion. Distortion computations such as Sum of Absolute Differences (SAD) in a search pattern are independent of each other. Hence, an FBM algorithm can be accelerated by computing multiple SADs in parallel.

The configurable and parallel nature of FPGAs makes them an attractive choice for accelerating various multimedia tasks such as ME [3, 6]. In order to implement an FBM algorithm on an FPGA, two types of modules are required: (i) Processing Element (PE) to compute SAD, and (ii) Memory subsystem to store and supply the pixel data of the video frames. In order to accelerate an FBM algorithm on an FPGA, multiple parallel PEs are needed. Supplying parallel PEs with pixel data to execute an FBM algorithm, requires an efficient way to store pixels on Block RAMs (BRAMs) and an efficient data routing network built using four-input Look-Up Tables (LUTs). The simple approach of replicating data is not efficient because BRAMs are a scare resource on FPGAs.

This thesis presents a memory architecture template that can be used to derive efficient memory subsystems for supporting FBM algorithm acceleration through parallel PEs. Each PE concurrently processes a block of pixels. The objective of the thesis is to derive a memory subsystem from a template with minimal resource requirement. This memory subsystem must support parallel access to multiple blocks of pixels and provide them to their respective

PEs. The blocks of pixels may or may not be contiguous depending on the search pattern of an FBM algorithm. Since there is a limited number of BRAMs present on an FPGA, the memory subsystem should not replicate pixels. This thesis states that for a given search pattern and number of parallel PEs, the memory architecture template can derive a memory subsystem to concurrently access blocks of pixels processed by all the PEs without data replication, provided there are enough available FPGA resources.

The rest of the thesis is organized as follows: Chapter 2 discusses a background about an FPGA device and FBM algorithm. This chapter also reviews the existing memory subsystems to accelerate FBM algorithms. Chapter 3 discusses the proposed memory architecture template followed by a mathematical proof of the architecture's claim in Chapter 4. Chapter 5 presents a performance prediction model for estimating the performance of the derived memory subsystems in terms of frame per second (fps). Chapter 6 discusses the Bounded Set (BS) algorithm used to derive memory subsystems from the proposed memory architecture template. In Chapter 7 the derived memory subsystems are compared with published works and evaluated based on their overall performance in terms of fps. Chapter 8 provide conclusion and discuss about the future works.

Chapter 2

Background and Related Works

2.1 Field Programmable Gate Array

An FPGA is a reconfigurable and user-programmable gate-array device. It consists of various configurable elements and embedded cores optimized for designing high density and high performance systems. As of today, Xilinx and Altera are the two major vendors of FPGAs. Although their FPGAs have similar functionalities, their inherent architectures are different [7, 8]. This chapter only discusses Xilinx FPGA because it is used for implementing the present work. Figure 2.1 shows the internal architecture of a Xilinx FPGA. The Input Output (I/O) blocks provide an efficient interface between the package pins and Configurable Logic Blocks (CLBs) in an FPGA. The global routing matrix provides a configurable interconnection between the CLBs and BRAMs using Switch Matrices (SMs). The configuration of CLBs and SMs is stored in their respective SRAM cells.

The CLBs are the major component of an FPGA and are used for implementing combinational as well as sequential logic. A CLB consists of four slices, each consisting of two LUTs and two storage elements. Each LUT is a function generator, which is capable of implementing any random logic involving four-input Boolean function. The Boolean function is implemented by storing the output column of its truth table as the configuration of the LUT and by selecting the output based on the logical values of the input (as shown in fig. 2.2). Such implementation makes the propagation delay of the LUT independent of the four-input Boolean function. Each storage element in a slice can be configured as an edge-triggered delay flip-flop or level-sensitive latch.

BRAM in an FPGA is a dual-port memory which can store 18Kbits of data [9]. However, it can be configured as single port RAM, simple dual port RAM, true dual port RAM,

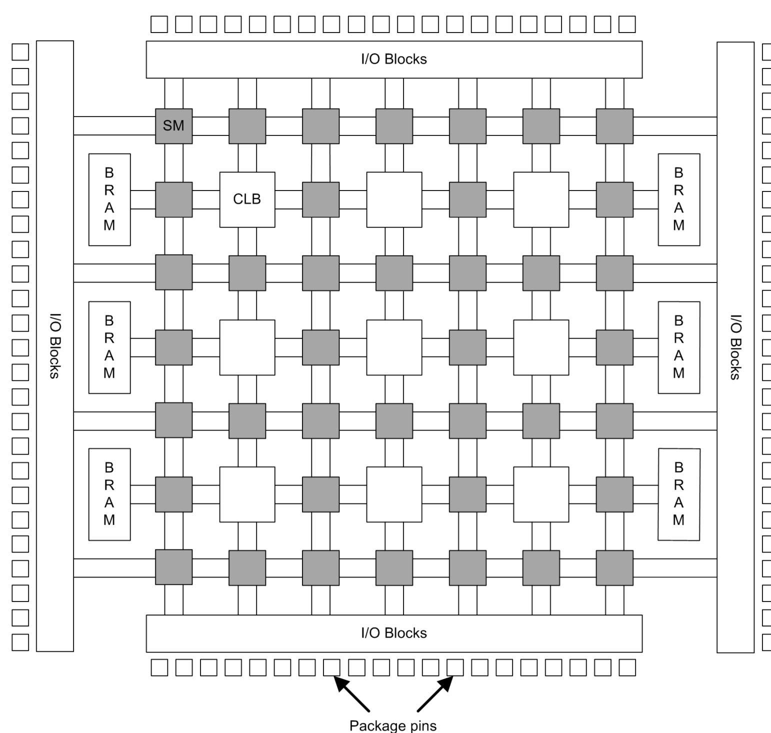
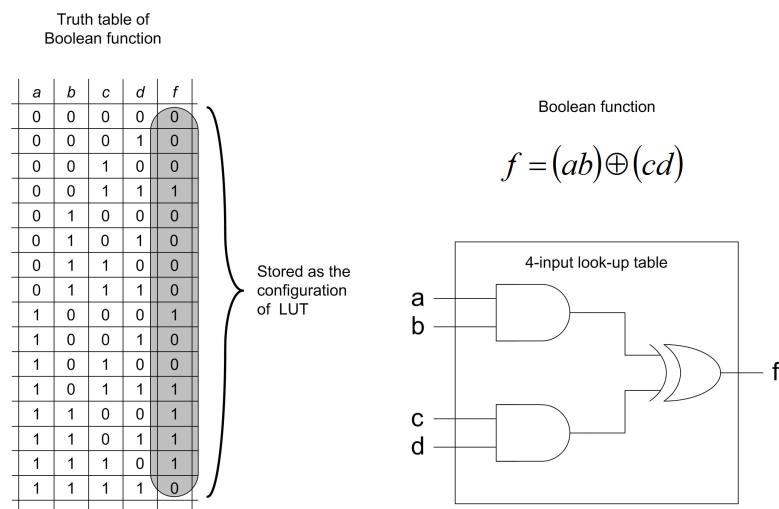


Fig. 2.1: FPGA architecture.



a	*b*	*c*	*d*	*f*
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

Fig. 2.2: Implementation of a Boolean function using four-input look-up table.

single port ROM, or dual port ROM using Block Memory Generator V2.8 in Xilinx ISE 10.1. It has two independent, symmetrical, and interchangeable access ports, A and B. The two ports perform synchronous read and write operation on a shared stored data. Both read and write operations require one clock edge. Each byte in the stored data has one extra parity bit. Each port has its own clock, write address, data in, data out, clock enable, and write enable signals. Each port of the BRAM can be configured at different aspect ratios varying from 16K×1, 8K×2, to 512×36, where the first and second terms are the number of data words and the number of bits per data word, respectively. Since the two ports are independent, they can have different aspect ratios. The addressing in the BRAM ensures that correct data is accessed from the two ports if their aspect ratios are different. If the parity bits are ignored, the aspect ratios of both ports (A and B) are 512×32 and 2K×8 respectively, and port A accesses the content of address 0, then in order to access similar content through port B, address locations 0, 1, 2, and 3 must be accessed. When parity bits are ignored and the ports configured as 16 and 32 bits wide, a byte can be written to a specific byte position of the data word by asserting individual bits of the write enable signal.

Xilinx FPGA Design Flow

The standard flow suggested by Design Flow Overview [10] for implementing designs on a Xilinx FPGA is shown in fig. 2.3. The logic of the design is entered in design entry stage, through the Xilinx supported schematic editor, or through hardware description languages such as Verilog or VHDL. Design hierarchy has an important role to play in this stage, since it partitions the design into hierarchical components allowing easier implementation and debugging. The design synthesis translates the logical information into a Native Generic Data (NGD) file. The NGD file contains the logical description of the design in terms of the hierarchical components, Xilinx primitives, and hard place and route macros. A functional simulation of the synthesized design verifies the logic of the design before it can be implemented on a device. Since the timing information is not available at this stage, the logic is verified using a unit delay. It is recommended to perform functional simulation at

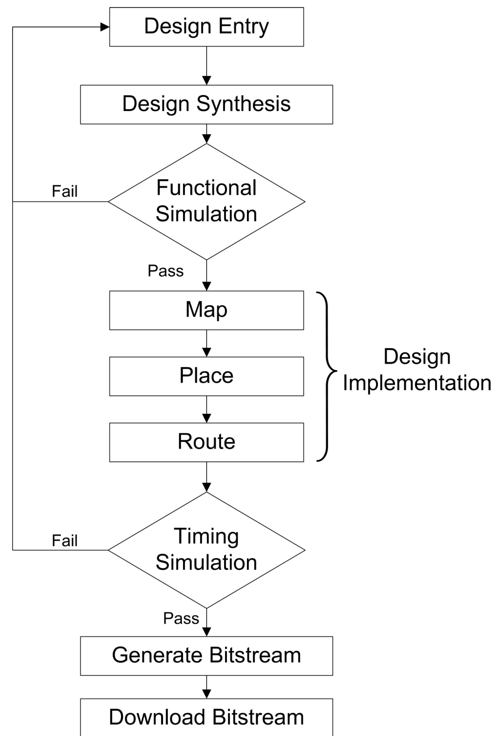


Fig. 2.3: Xilinx FPGA design flow.

an early stage of the design flow because it is faster and easier to correct design errors. In the design implementation stage, the logic defined in the NGD file is mapped into CLBs, BRAMs, and other components of the target Xilinx FPGA. The timing information of the design is obtained by placing and routing these components on the FPGA. Once timing information is available, a timing simulation of the design is performed to verify that the implemented design can run at the desired speed. After a timing verification of the design, a bitstream containing the configuration information is created and downloaded into the FPGA.

2.2 Motion Estimation

The temporal redundancies as stated by Richardson [11], is due to object motion and movement of camera in a video sequence, and can be attributed to the movement of pixels between the frames. If trajectories of every pixel between the frames can be estimated using an optic flow, the flow vector of these pixels can be used to encode a particular frame.

However, calculating the flow vector of every pixel is a compute intensive process and the resulting vectors must be sent to the decoder for reconstructing the source frame. This results in a transfer of more data negating the advantage of finding the trajectories. A less compute intensive approach for encoding a frame is to estimate the movement of blocks of pixels instead of individual pixels. The frame to be encoded (current frame) is divided into blocks of $p \times q$ pixels also known as current frame blocks. For each current frame block, an area in the reference frame (which could be a previous or future frame) is searched to find the best matching block. This area is known as the search area. Block matching can be performed by considering all or selected blocks in the search area (also known as candidate blocks). One of the matching criteria is the SAD of pixel intensities from the current frame block and candidate block. The candidate block with the least sum is considered as the best match for the current frame block. If the top left pixel (anchor pixel) of current frame and candidate block are located at coordinates (i, j) and (i', j') respectively, then the SAD is computed using (2.1), where SAD(i', j') is the SAD value of a current frame block with respect to a candidate block located at (i', j') and (X, Y) represent pixel intensity of current and reference frame, respectively. This process of finding the best matching block is known as ME.

$$SAD(i', j') = \sum_{m=0}^{p-1} \sum_{n=0}^{q-1} \left| X(i+m, j+n) - Y(i'+m, j'+n) \right| \tag{2.1}$$

Figure 2.4(a) shows a current frame partitioned into $p \times q$ blocks of pixels. A search range to find the best match for a current frame block anchored at (i, j) is shown in fig. 2.4(b). The search range is d pixels in both the horizontal and vertical directions. In fig. 2.4(b) the candidate block anchored at (i', j') is found to have the least SAD value and therefore regarded as the best match for the current frame block located at (i, j) position. The motion vector of the current frame block $(mv(i, j))$ in this case is computed using eq. (2.2).

$$mv(i, j) = (i' - i, j' - j) \tag{2.2}$$

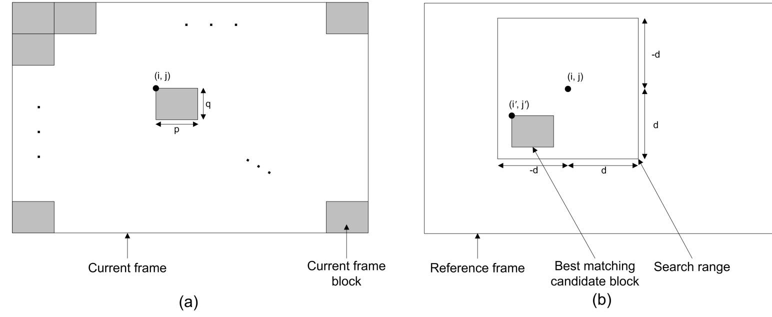


Fig. 2.4: Motion estimation.

One of the simplest approaches to find the best match is to compare every candidate block present in the search area. This approach is known as full search, and it has very high computational demands [2]. Therefore, FBM algorithms such as Mixed Diamond Hexagon and Cross (MDHC) search proposed by Duanmu et al. [2] and UMHexagonS proposed by Chen et al. [1] have been proposed to reduce the computational demands of ME. An FBM algorithm consists of a series of search steps (search patterns) to select specific candidate blocks in the search area for SAD calculations. Each search pattern has a set of anchor points in the search area, where the top-left pixel of the candidate blocks are placed during the block matching process.

2.3 Existing Memory Subsystems

Memory subsystems which can accelerate execution of an FBM algorithm by concurrently providing access to distinct locations of a frame have been the subject of active research. This concurrency is achieved by either (i) storing multiple copies of frame data, or (ii) distributing the frame data in distinct memory modules. In this chapter, we restrict the review to published approaches falling in the second category.

Dutta et al. [12, 13] have designed a systolic array based memory subsystem for full search, three-step search, and conjugate-direction search using scheduling, memory partitioning, and area delay tradeoff of memory and its interconnect network. The memory subsystem consists of multi-bank frame buffer, a pipelined interconnect network, and multiple PEs. Their methodology develops an efficient design for the full search ME algorithm.

However, for other FBMAs, their data flow schedule requires extra number of clock cycles to execute the algorithm because of the memory-access conflicts leading to delayed start of the computation in some PEs.

Vleeschouwer et al. [4] have proposed a directional squared-search algorithm accelerated by a ME kernel with three PEs. The kernel reduces the average memory bandwidth and power consumption through data reuse resulting from the overlapping of adjacent candidate blocks and search areas. The data reuse increases with the proximity of the anchor points in the search pattern, making it dependent on the FBM algorithm. Therefore, their ME kernel is restricted only to the directional squared-search algorithm.

Peng et al. [5] have proposed a parallel memory subsystem with the number of memory modules restricted to powers of two. It is observed that for ME, this memory subsystem can provide a conflict free parallel access to either a column or a row of one candidate block but not both. However, a greater speed up in the execution of an FBM algorithm can be achieved through parallel access of rows and columns of multiple candidate blocks.

In order to access an entire candidate block, Vanne et al. [6] and Kuzmanov et al. [3] have proposed a similar memory subsystem for multimedia applications. The memory subsystems are designed to access an unaligned rectangular block of pixels. The proposed memory subsystem is referred to as M_R by Vanne et al. [6] and 2-D Addressable Memory ($2DAM$) by Kuzmanov et al. [3]. Each memory module in these memory subsystems stores one pixel per address location. The number of memory modules in these memory subsystems is equal to the number of pixels in the required block. Thus, these memory subsystems only fetch the required block of pixels. The fetched block of pixels is then aligned using a shuffle network. These two memory subsystems have separate row and column addressing for a pixel and require only small amount of FPGA resources to realize their addressing scheme due to the dimensions of the required block (which are in the powers of two). Restricting the number of pixels stored per address location to one, often results in a requirement for large numbers of memory modules.

Beric et al. [14, 15] have proposed a two-level ($L1, L0$) cache structure for block-based

video algorithms. The $L0$ cache, stores multiple pixels per address location. This cache also uses folding and data reorganization which reduces clock cycles required to access an arbitrary aligned rectangular block of data. A data reordering unit and shuffler are used as an interface between L0 cache and the PE. The L1 cache reduces the off-chip bandwidth requirements of the L0 cache. Since, this design is targeted towards an Application Specific Integrated Circuit (ASIC), it focuses on minimizing the gate count and the off-chip bandwidth. Their memory subsystem requires duplication of the L0 cache for each PE. Table 2.1 summarizes and compares the existing memory subsystems.

Table 2.1: Summary of existing memory subsystems.

Memory subsytems	Device Utilization of PEs (%)	Restriction to FBMA	Number of Memory Modules	Support to maximum number of PEs without data replication
Dutta et al. [12, 13]	< 100	Diamond search, Conjugate-Direction search	16	Varies from 1 to total number of anchor points in the search pattern
Vleeschouwer et al. [4]	100	Directional squared-search	4	3
Peng et al. [5]	100	independent	maximum of (p, q)	1
Kuzmanov et al. [3] and Vanne et al. [6]	100	independent	$p \times q$	1
Beric et al. [14, 15]	100	independent	$< p \times q$	1

Chapter 3

Architecture Template

The components of an FPGA-based memory architecture template is presented in fig. 3.1. It broadly consists of two parts: (i) a set of parallel PEs to evaluate block matching criterion such as SAD, and (ii) an on-chip memory subsystem that fetches required sub-sets of current and reference frame pixels from an external memory (DDR2 SDRAM storing frame pixels in raster scan order). The on-chip memory subsystem has two variants of Frame Buffer, Current Frame Buffer (CFB), and Reference Frame Buffer (RFB). The CFB and RFB store pixels of current and reference frames, respectively. The Pixel Load Unit (PLU) receives pixels from external memory and rearranges them according to the addressing scheme for loading pixels. The Pixel Fetch Unit (PFU) generates addresses to facilitate transfer of a sub-set of pixels in RFB/CFB to the Pixel Select and Rearrangement Modules (PSRM). The PSRMs (one per PE) select reference frame pixels and rearrange current frame pixels. It is necessary to define the following fundamental terms before explaining the components of the memory subsystem in later sections.

Super Block: A Super Block (SB) is a block of pixels which contains all the candidate blocks that are required for computing block matching criterion such as SAD at a given step of a search pattern. For a search pattern with n anchor points (where the candidate blocks are placed) executed by N_{PE} parallel PEs, the number of candidate blocks in each SB is restricted to N_{PE} and the number of SBs (N_{SB}) is computed using (3.1). Each SB is characterized by its width (SB_w) and height (SB_h), respectively.

$$N_{SB} = \left\lceil \frac{n}{N_{PE}} \right\rceil \tag{3.1}$$

Super Block Offset: A Super Block Offset is the offset of the top-left pixel of an SB from the

center of the search pattern. It assists in accessing the SB which contains the required candidate blocks for a given step of an FBM algorithm. The SB offset (SB_i, SB_j) is computed using (3.2), where (i_k, j_k) is the coordinate of the kth anchor point in a group.

$$SB_i = \min\{i_k\} \, \forall \, 0 \leq k < N_{PE} \qquad SB_j = \min\{j_k\} \, \forall \, 0 \leq k < N_{PE} \tag{3.2}$$

Anchor Point Offset: An Anchor Point Offset is the offset of an anchor point with respect to the top-left pixel of the SB. It assists in selecting out a candidate block from the SB. The anchor point offset $(\Delta i_k, \Delta j_k)$ of the kth anchor point in the SB is computed using (3.3).

$$\Delta i_k = i_k - SB_i \qquad \Delta j_k = j_k - SB_j \tag{3.3}$$

Union of Super Blocks: A Union of Super Blocks (USB) is a rectangular grid of pixels which can provide access to all the SBs. It is the largest Tightest Bounding Rectangle (TBR) which can enclose all SBs and its width $(Grid_w)$ and height $(Grid_h)$ are equal to the width and height of the widest and the tallest SB(s) respectively, computed using (3.4, 3.5), where $SB[k]_w$ and $SB[k]_h$ are width and height of the kth SB. In order to access the candidate blocks contained inside a SB, the top left pixel of the USB is placed at the offset of that SB.

$$Grid_w = \max\{SB[k]_w\} \; \forall \; 0 \leq \; k < N_{SB} \tag{3.4}$$

$$Grid_h = \max\{SB[k]_h\} \; \forall \; 0 \leq \; k < N_{SB} \tag{3.5}$$

Figure 3.2(a) shows a search pattern with seven anchor points, being executed by three parallel PEs. The execution requires three SBs (computed using (3.1)), and each SB is assigned maximum three anchor points. Figures 3.2(b-d) show three SBs (SB[0], SB[1], and SB[2]) enclosing the required candidate blocks through a TBR. It can be inferred from fig. 3.2, that SB_w and SB_h of each SB is determined by the number of parallel PEs, relative

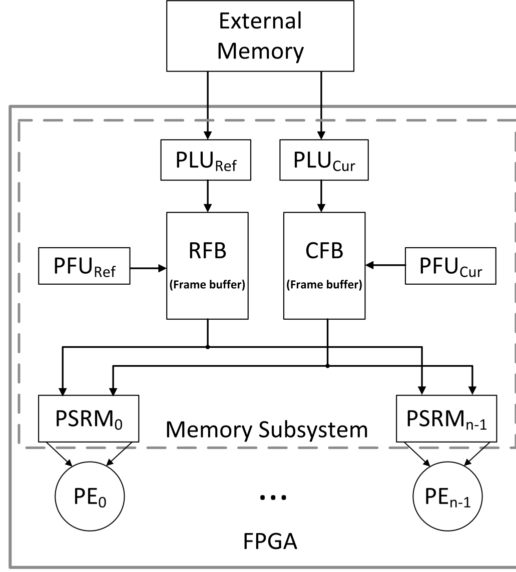


Fig. 3.1: Overview of the FPGA system featuring the proposed memory architecture template.

position of the anchor points in a search pattern, and the size of the candidate block. The SB offsets of SB[0], SB[1], and SB[2] in figs. 3.2(b-d) are (-1, -2), (1, -2), and (-2, 0), respectively. The anchor point offsets of three anchor points in SB[0] (fig. 3.2(b)) are (0, 0), (-1, 2), and (1, 2), respectively. The $Grid_w$ and $Grid_h$ for the SBs shown in figs. 3.2(b-d) are found to be 6 and 6, respectively. It can be observed from (3.4, 3.5) and fig. 3.2 that $Grid_w$ and $Grid_h$ are determined by the allocation of the anchor points of a search pattern to each SB. A large relative distance between the anchor points of each SB will result in larger values of $Grid_w$ and $Grid_h$.

3.1 Frame Buffer

The design of a parameterized frame buffer is shown in fig. 3.3, where the width of the signals (if not specified) is in bits. A frame buffer consists of a 2-dimensional grid of BRAMs configured as dual port RAM. It has B_w columns and B_h rows of BRAMs, respectively. The values of B_w and B_h are calculated separately for RFB and CFB, based on the USB and the block size, respectively. When pixels need to be loaded in to a frame buffer, the BRAMs are identified by the signals PL_Row_S and PL_Col_S, which are computed by the PLU

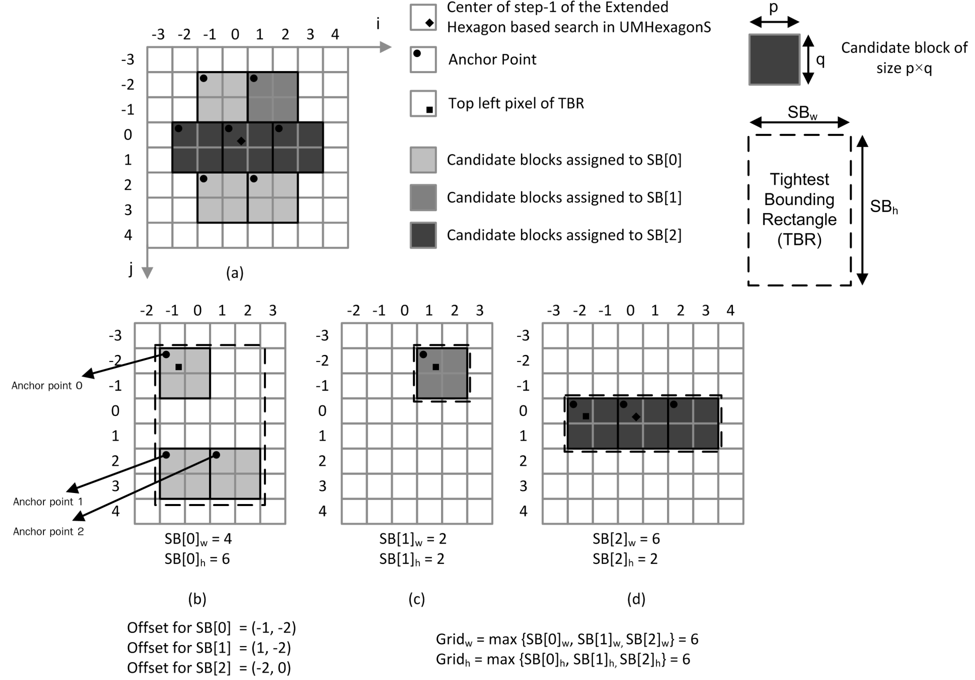


Fig. 3.2: Example illustrating the fundamental terms.

based on the coordinates of pixels to be loaded. When pixels need to be fetched from the frame buffer, the BRAMs are addressed by the PF_K_i, PF_K_j, PF_A_i, and PF_A_j signals, which are computed by the PFU based on the SB offset or the offset of the anchor pixel of the current frame block. The computations performed by PLU and PFU are explained in sections 3.2 and 3.3, respectively. The PLU receives a limited number of pixels from the external memory and loads them into specific BRAMs. But once these pixels are stored in a frame buffer, pixels can be fetched from all the BRAMs. Therefore, the load operation is performed on selected BRAMs, but the fetch operation is performed on all BRAMs. The FBM algorithm processes blocks of adjacent pixels. In order to fetch these blocks from a frame buffer, it is desirable to fetch a block of $A_w \times A_h$ adjacent pixels from each BRAM, where, A_w and A_h are the number of horizontally and vertically adjacent pixels, respectively, that are stored in one address location of a BRAM. The write and read ports of each BRAM are configured as A_w and $A_w \times A_h$ pixels wide. Since the load and fetch operations on a BRAM need to be independent processes, the write and read ports are driven by Clk_L and Clk_F clock signals, respectively. A frame buffer supplies a block

of $(B_w \times A_w) \times (B_h \times A_h)$ pixels ($Data_out$) to the PSRM because there are $B_w \times B_h$ BRAMs and each fetches $A_w \times A_h$ pixels. The external memory supplies a row of pixel to the PLU, and these pixels are loaded to a row of BRAMs in a frame buffer. Therefore, the maximum number of pixels that can be supplied by the PLU to a frame buffer is $B_w \times A_w$ ($Data_in$) pixels. Signals $Load_Addr$ and PL_Cin are used to compute values supplied to the address ports of all the BRAMs during the load operation. Although address is supplied to all BRAMs, the PL_Row_S and PL_Col_S enable a subset of BRAMs to which pixels are loaded. The signals PF_K_i, PF_K_j, PF_A_i, and PF_A_j are used to compute values supplied to the address ports of all BRAMs during the fetch operation.

The RFB and CFB can be designed by customizing the frame buffer template with a unique variant of these four parameters: B_w, B_h, A_w, and A_h. In the Xilinx Virtex 4 family of FPGAs, the port width of BRAMs can be configured to a maximum of 32 bits (ignoring the parity bits). Therefore, the maximum value of $A_w \times A_h$ can be 4, where A_w and A_h can each be 1, 2, or 4. The variant of these parameters to design a RFB and CFB are (B_w^R, B_h^R, A_w^R, A_h^R) and (B_w^C, B_h^C, A_w^C, A_h^C), respectively. The ME process involves aligned access of a single current frame block and misaligned access of multiple candidate blocks of a reference frame. Therefore, the parameters for the CFB are derived from the dimensions of the block using (3.6-3.8), where p and q are, respectively, the width and height of the current frame block.

$$B_w^C = \frac{p}{A_w^C} \qquad B_h^C = \frac{q}{A_h^C} \tag{3.6}$$

$$A_w^C = \begin{cases} 4 \; ; & 4 \leq p \\ 2 \; ; & 2 \leq p < 4 \\ 1 \; ; & otherwise \end{cases} \tag{3.7}$$

$$A_h^C = \begin{cases} 4 \; ; & A_w^C = 1 \; and \; 4 \leq q \\ 2 \; ; & A_w^C = 1 \; and \; 2 \leq q < 4 \\ 1 \; ; & otherwise \end{cases} \tag{3.8}$$

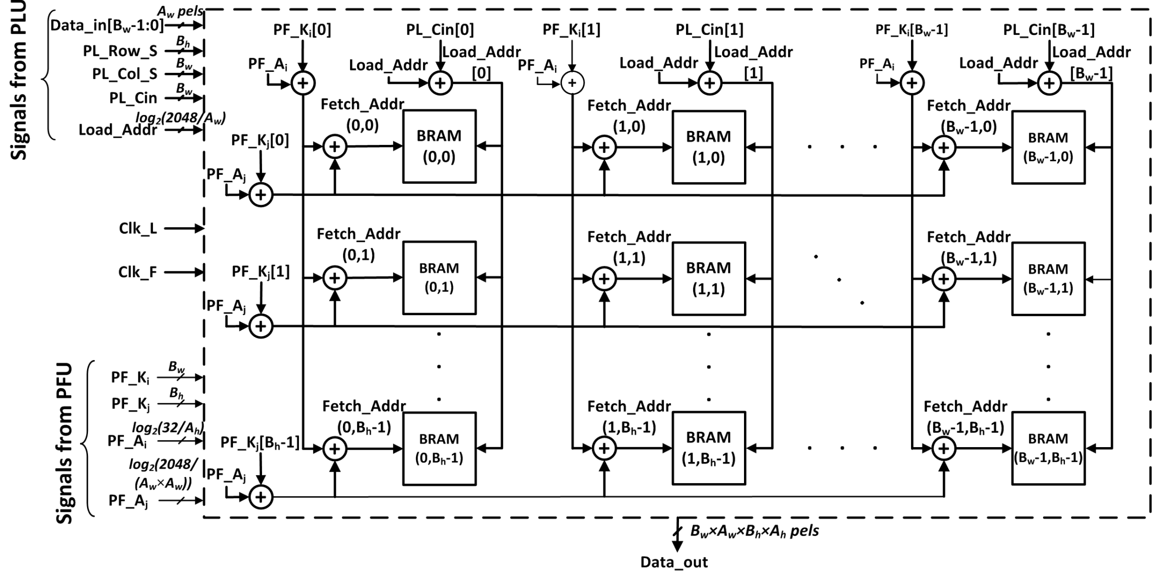


Fig. 3.3: Frame buffer.

As discussed in section 3.1, the candidate blocks can be accessed by accessing the USB. Therefore, the RFB is derived from the USB using (3.9, 3.10).

$$B_w^R \geq \frac{Grid_w + A_w^R - 1}{A_w^R} \tag{3.9}$$

$$B_h^R \geq \frac{Grid_h + A_h^R - 1}{A_h^R} \tag{3.10}$$

Due to aligned access, the current frame block is always present in a single address location and the BRAMs have identical address values ($Fetch_Addr$) at their read ports. Therefore, the signals PF_K_i and PF_K_j from PFU_{Cur} and their respective adders are not required in the CFB.

The on-chip BRAM of a Virtex 4 FPGA can store up to 2048 pixels, assuming eight bits per pixel with parity bits ignored [9]. It is assumed that every BRAM stores 32 rows of 64 pixels. However, these numbers can be modified as required by the application. Since the width of the write and read ports of a BRAM in the frame buffer are different, two different addressing schemes are required to perform load and fetch operations.

3.1.1 Load Scheme

The load scheme identifies a pixel located at (i, j) coordinates of a frame by (m_p, m_q), *Load_Addr*, and *Load_offset*. The values (m_p, m_q) computed using (3.11) correspond to the coordinate of the BRAM in the frame buffer, where the pixel will be loaded to. The address (*Load_Addr*) of that pixel in (m_p, m_q) BRAM is calculated using (3.12-3.14). The offset (*Load_offset*) of the pixel within *Load_Addr* is calculated using (3.15).

$$m_p = \left\lfloor \frac{i}{A_w} \right\rfloor mod\ B_w \quad m_q = \left\lfloor \frac{j}{A_h} \right\rfloor mod\ B_h \tag{3.11}$$

$$PL_A_i = \left(\left\lfloor \frac{i}{B_w \times A_w} \right\rfloor \times A_h \right)\ mod\ \left(\frac{64}{A_w} \times A_h \right) + j\ mod\ A_h \tag{3.12}$$

$$PL_A_j = \left(\left\lfloor \frac{j}{B_h \times A_h} \right\rfloor \times \left(\frac{64}{A_w} \times A_h \right) \right)\ mod\ \frac{2048}{A_w} \tag{3.13}$$

$$Load_Addr = PL_A_i + PL_A_j \tag{3.14}$$

$$Load_offset = i\ mod\ A_w \tag{3.15}$$

A RFB is derived for a USB, which encloses the SBs of figs. 3.2(b-d). The parameter of the RFB which results in least number of BRAMs (i.e., $B_w^R \times B_h^R$) are $B_w^R = 6$, $B_h^R = 3$, $A_w^R = 1$, and $A_h^R = 4$. Figure 3.4(a) shows the value of (m_p, m_q), *Load_Addr*, and *Load_offset* for pixels to be loaded in that RFB.

3.1.2 Fetch Scheme

The fetch scheme assigns (m_p, m_q) using (3.11) but, address (*Fetch_Addr*) and offset (*Fetch_offset*) are different from *Load_Addr* and *Load_offset*, respectively. This is due to the difference in the width of write and read ports of a BRAM. The *Fetch_Addr* is computed using (3.16-3.18), where PF_A_i is a $log_2(64/A_w)$ bit value and PF_A_j has a $log_2(2048/A_w \times$

A_h) bit value with the lower $log_2(64/A_w)$ bits as zeroes. Therefore, the addition operation in (3.18) is performed through concatenation. The *Fetch_offset* is computed using (3.19), where (o_p, o_q) represents the coordinate in the $A_w \times A_h$ block of adjacent pixels stored in an address location of a BRAM.

$$PF_A_i = \left\lfloor \frac{i}{B_w \times A_w} \right\rfloor mod\ \frac{64}{A_w} \tag{3.16}$$

$$PF_A_j = \left(\left\lfloor \frac{j}{B_h \times A_h} \right\rfloor\ mod \frac{32}{A_h} \right) \times \frac{64}{A_w} \tag{3.17}$$

$$Fetch_Addr = PF_A_i + PF_A_j \tag{3.18}$$

$$o_p = i\ mod\ A_w \quad o_q = j\ mod\ A_h \quad Fetch_offset = o_p + A_w \times o_q \tag{3.19}$$

Figure 3.4(b) shows (m_p, m_q), *Fetch_Addr*, and *Fetch_offset* of the pixels fetched from the RFB. Fig. 3.4(c) shows a USB enclosing SB[0] whose offset in fig. 3.4(b) is (0,3).

3.2 Pixel Load Unit

The PLU recieves pixels from an external memory and loads them into the BRAMs of the frame buffer using load scheme. It is assumed that the PLU receives eight adjacent pixels of a frame (PL_{Data}) from an external memory, but it can also be tailored to receive a different number of pixels. The input to the PLU is a coordinate of the left most pixel (i_{load}, j_{load}) of the PL_{Data}. There are two distinct designs of PLU, based on the product of frame buffer's B_w and A_w.

When a PLU is designed to load pixel into the frame buffer with $B_w \times A_w \geq 8$, all eights pixels in the PL_{Data} are loaded in to the frame buffer at every clock cycle. The eight pixels are concatenated with dummy pixels, such that total number of pixels is $B_w \times A_w$ (as shown in fig. 3.5). The concatenation is done so that the barrel shifter can reorder and align pixels in PL_{Data} with appropriate BRAM columns of the frame buffer. The

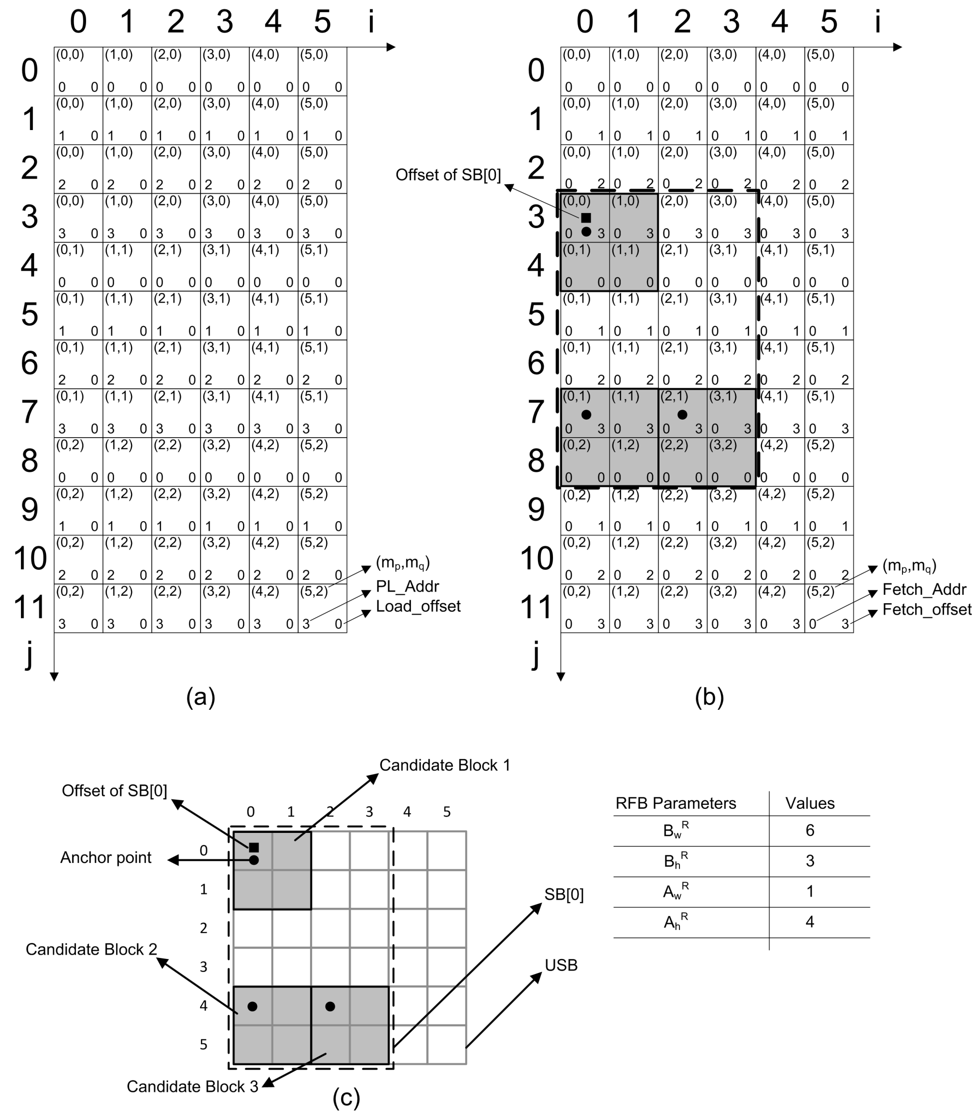


Fig. 3.4: Addressing scheme for (a) load operation, (b) fetch operation on a RFB, and (c) shows SB[1] being enclosed by the USB of fig. 3.2.

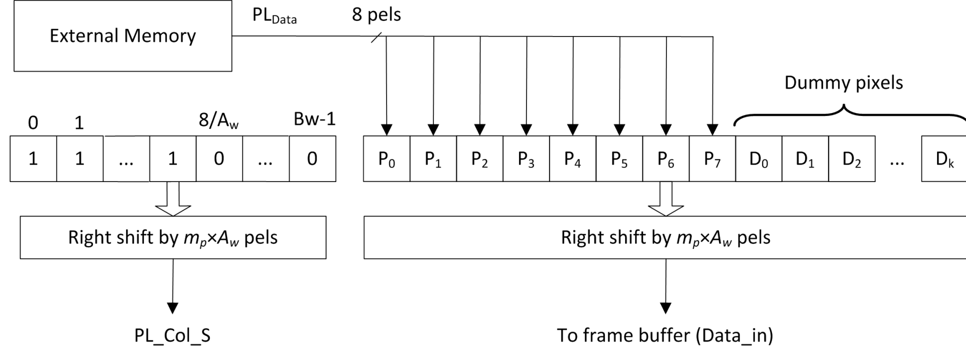


Fig. 3.5: Rearrangement of external pixels and generation of Col_LS signal by pixel load unit for a frame buffer with $B_w \times A_w \geq 8$.

barrel shifter shifts the concatenated pixels by $m_p \times A_w$ pixels in the right direction, where m_p is computed by substituting i_{load} for i in (3.11). The row and column select signals (PL_Row_S and PL_Col_S) are B_w and B_h bits wide, respectively (with the 0th bit as the most significant bit). Since, the PL_{Data} has horizontally adjacent pixels, the m_q values for the BRAMs where these pixels be loaded are same. Therefore, m_q value for all BRAMs can be calculated by substituting j_{load} to j in (3.11) and the PL_Row_S signal is generated by setting its m_q^{th} bit to one. The PL_Col_S signal is computed through a shift operation with wrap around based on the value of m_p (also shown in fig. 3.5).

For a load operation, $Load_Addr$ of the left most pixel is computed by substituting (i_{load}, j_{load}) for (i, j) in (3.12-3.14). A PL_Cin signal of B_w bits is also computed based on m_p. Each bit of this signal is added to $Load_Addr$ to compute the $Load_Addr$ for the remaining pixels. If the value of m_p is not zero, then 0 to m_p - 1 bits in PL_Cin are set to one, or else all bits in the signal are set to zero.

In the second design of the PLU for loading pixels into the frame buffer with $B_w \times A_w <$ 8, only $B_w \times A_w$ pixels are loaded into the frame buffer each clock cycle. The remaining pixels are buffered in a 64-bit register and are referred as residual (as shown in fig. 3.6). The residual at clock cycle t is computed using (3.20), where $r(0) = i_{load}\ mod\ (B_w \times A_w)$.

$$r(t+1) = \begin{cases} 8 + r(t) - B_w \times A_w; & r(t) < B_w \times A_w \\ r(t) - B_w \times A_w; & otherwise \end{cases} \tag{3.20}$$

If the residual at any clock cycle is greater than $B_w \times A_w$, no new pixels are fetched from the external memory and $B_w \times A_w$ pixels of the residual are loaded into the frame buffer the following clock cycle. This is repeated until the residual drops below $B_w \times A_w$ pixels. Once this happens, new pixels are received from the external memory and concatenated to the residual (if any), then $B_w \times A_w$ pixels are loaded into the frame buffer. The left shift operation $l(t)$, for aligning the concatenated pixels to their respective BRAM columns of the frame buffer at clock cycle t is computed using (3.21).

$$l(t) = 8 - r(t) \tag{3.21}$$

Since, $B_w \times A_w$ pixels are loaded into the frame buffer, the $load_Addr$ for every pixel is identical. The $load_Addr$ is computed by substituting (i'_{load}, j_{load}) for (i, j) in (3.12-3.14) and i'_{load} at clock cycle t is computed using (3.22).

$$i'_{load} = \begin{cases} i_{load} + l(t); & r(t) \geq B_w \times A_w \\ i_{load} - r(t); otherwise \end{cases} \tag{3.22}$$

The PL_Row_S signal to select the row of BRAMs where the pixels are to be loaded is computed using the same way as mentioned in the previous design of the PLU. Since every BRAM in the selected row encounters a load operation, all the bits in the PL_Col_S signal are set to high. It is assumed that the inputs to the PLU_{Cur} and PLU_{Ref} are supplied by the external controller, which also controls the flow of the FBM algorithm.

3.3 Pixel Fetch Unit

The PFU facilitates in fetching pixels from the BRAMs of the frame buffer using fetch scheme. The PFU has distinct designs for the CFB and RFB. As discussed in section 3.1, the $Fetch_Addr$ for accessing a current frame block from the CFB is the same for all BRAMs. Therefore, the PFU_{Cur} computes this value based on the coordinate of the top left (or any) pixel in the current frame block, using (3.16-3.18). The parameters for designing the RFB are computed based on the size of USB. Due to the misaligned access of the reference frame,

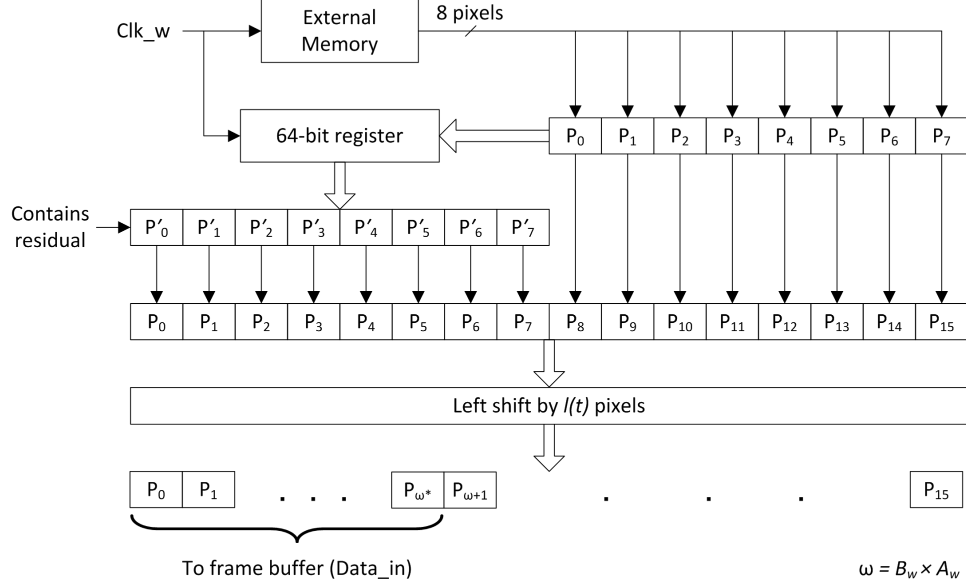


Fig. 3.6: Buffering of external pixels and their shifting by a pixel load unit for loading pixels into the frame with $B_w \times A_w < 8$.

the pixel enclosed by the USB may straddle in more than one address location. Therefore, the PFU_{Ref} only computes (m_p, m_q) and $Fetch_Addr$ for the top left pixel in the USB which also happens to be the offset of SB containing the required candidate blocks. It also computes PF_K_i and PF_K_j, which are later added to PF_A_i and PF_A_j, respectively, (as shown in fig. 3.3) to compute $Fetch_Addr$ for accessing the remaining pixels. PF_A_i and PF_A_j are B_w and B_h bit signals with the 0th bit as the most significant bit. The values of PF_K_i and PF_K_j are calculated based on the values of (m_p, m_q). The columns of BRAMs to the left of m_p have PF_K_i equal to one; and the rows of BRAMs above m_q have PF_K_j equal to one. These computations are performed using a look-up-table, whose inputs and outputs are (m_p, m_q) and (PF_K_i, PF_K_j), respectively. The inputs to PFU_{Cur} and PFU_{Ref} are also assumed to be provided by the controller.

3.4 Pixel Select and Rearrangement Module

The PSRM selects a $p \times q$ candidate block from the fetched contents of the RFB, and rearranges a current frame block fetched from the CFB. It consists of a Reference Frame Selector (RFS), Current Frame Rearranger (CFR), and a PSRM controller.

3.4.1 Reference Frame Selector

The RFS is designed to select any arbitrarily aligned $p \times q$ candidate block from the RFB's output (D_R). The D_R fetched from the RFB is a block of pixels with $B_w^R \times A_w^R$ columns and $B_h^R \times A_h^R$ rows. The selection is done using two selectors: a column selector and a row selector. While the order of selection does not affect the functionality, it can impact the resource utilization. When the RFS is designed with a column selector followed by a row selector, the input to the column selector is $B_w^R \times A_w^R$ columns of $B_h^R \times A_h^R$ pixels and the output is p columns of $B_h^R \times A_h^R$ pixels. This output is later routed to the row selector, which selects q rows from $B_h^R \times A_h^R$ rows resulting in a $p \times q$ block of pixels. When the RFS is designed with a row selector followed by a column selector, the input to the row selector is $B_h^R \times A_h^R$ rows of $B_w^R \times A_w^R$ pixels and the output is q rows of $B_w^R \times A_w^R$ pixels. This output is routed to the column selector, which selects p columns and outputs a $p \times q$ block of pixels.

Figure 3.7 shows columns of pixels in D_R as input to the column selector, when column selection is done first. There are p multiplexers (CM_0 to CM_{p-1}) and the number of columns (N) is equal to $B_w^R \times A_w^R$. Each column of pixels in the D_R can be an input to one or more multiplexers. This assignment of pixel columns to the multiplexer's input enables the column selector to select any adjacent p columns. The adjacent columns can wrap around so that column C_{N-1} and C_0 are adjacent. In fig. 3.7, the CM_α corresponds to the multiplexer where C_{N-1} (last column) of D_R is connected. The value of α is computed using (3.23).

$$\alpha = ((B_w^R \times A_w^R) - 1) \ mod \ p \qquad (3.23)$$

In the previously discussed RFB, where (B_w^R, A_w^R) is equal to (6, 1), to select any arbitrary 2×2 candidate block, the columns of pixels should be assigned to the multiplexers according to the fig. 3.8, when column selection is done first. The value of α for the RFB and candidate block is found to be 1 using (3.23).

The row selector is the row equivalent of the column selector and selects q adjacent

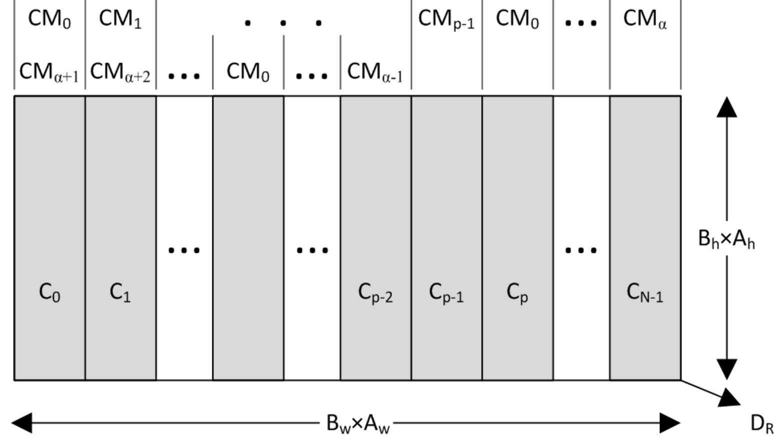


Fig. 3.7: The assignment of column of pixels in D_R as input to the multiplexers of the column selector.

rows of pixels from D_R or the output of the column selector. The row selector consists of q multiplexers and has a similar assignment of rows to multiplexers as described for the column selector.

3.4.2 Current Frame Rearranger

The RFS may select a $p \times q$ candidate block ($RFSO$) with an anchor pixel not positioned at the top left corner of the block. Therefore, the CFR is used to rearrange pixels fetched from the CFB to match the alignment of the pixels in $RFSO$. The rearrangement is performed using two wrap around rotators: a column rotator and a row rotator. The input to the column rotator is in column major order and its output, arranged in row major order is the input to the row rotator. Each rotator is implemented as a barrel shifter with wrap around in right direction. The CFR is similar to the routing network described by Kuzmanov et al. [3].

3.4.3 PSRM Controller

The PSRM controller generates the select line signals for RFS using a look up table. The inputs to the look up table are (m_p, m_q) and (o_p, o_q) computed for each anchor point in the SB. The (m_p^k, m_q^k) and (o_p^k, o_q^k) for the kth anchor point in the SB is computed using

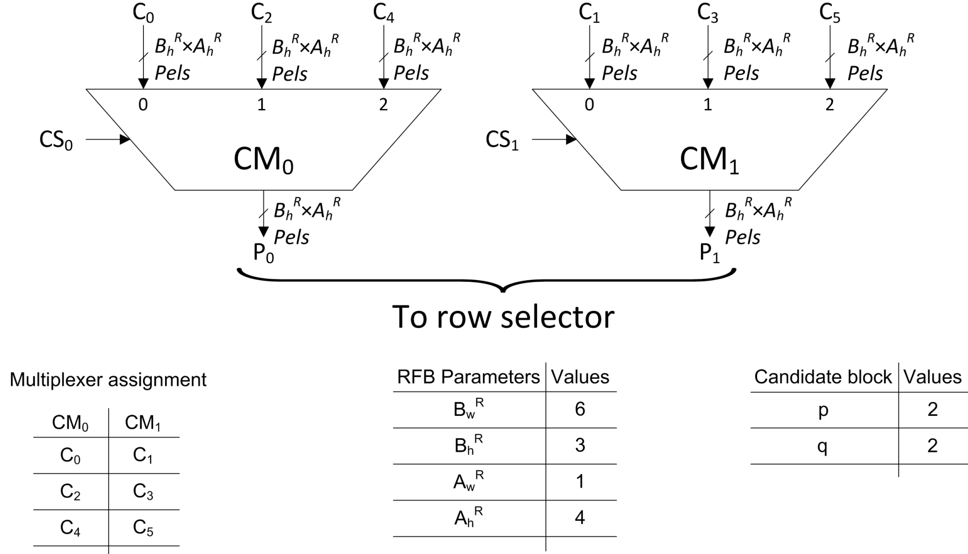


Fig. 3.8: Assignment of columns of D_R to the multiplexers of column selector for $B_w^R = 6$, $A_w^R = 1$, and $p = 2$.

(3.24, 3.25), where $(\Delta i_k, \Delta j_k)$ is the anchor point offset of the kth anchor point in the SB. The shift signals $(Shift_{col}, Shift_{row})$ for the CFR are generated by a look-up table using (3.26).

$$m_p^k = m_p + \left\lfloor \frac{o_p + \Delta i_k}{A_w} \right\rfloor \qquad m_q^k = m_q + \left\lfloor \frac{o_q + \Delta j_k}{A_h} \right\rfloor \tag{3.24}$$

$$o_p^k = (o_p + \Delta i_k) \;\; mod \;\; A_h \qquad o_q^k = (o_q + \Delta j_k) \;\; mod \;\; A_w \tag{3.25}$$

$$Shift_{col} = \left(m_p^k + m_p^k \times A_w \right) \;\; mod \;\; p \qquad Shift_{row} = \left(m_q^k + m_q^k \times A_h \right) \;\; mod \;\; q \tag{3.26}$$

Chapter 4

Proof of Conflict Free Parallel Access

The BRAMs in the RFB have only one read port and so only one value of $Fetch_Addr$ can be accessed at any given time. Therefore, for fetching multiple candidate blocks in a single access (conflict free parallel access), if two pixels in the USB have identical values of (m_p, m_q) and $Fetch_Addr$, then they should have distinct values of *Fetch_offset*. We prove this claim by the following lemma and theorems.

lemma: If x_1 mod y = x_2 mod y, such that $x_1 \leq x_2$ then $x_2 - x_1 = z \times y$ and z is positive integer ($z \geq 0$).

Proof: When x_1 mod y = x_2 mod y, it implies that the remainder of (x_1/y) and (x_2/y) are equal and (4.1) holds true, where $a_2 = x_2/y$ and $a_1 = x_1/y$, respectively. Assuming a_2 - $a_1 = z$, in (4.2), if $x_2 = x_1$ then $z = 0$, and if $x_2 > x_1$ then $z \geq 0$.

$$x_2 - a_2 \times y = x_1 - a_1 \times y \tag{4.1}$$

$$\Rightarrow x_2 - x_1 = (a_2 - a_1) \times y \tag{4.2}$$

Theorem 1: If only USB were to be loaded into the FB, then value of $Fetch_Addr$ to access every pixels of the USB is identical and is equal to zero.

Proof: If a pixel is located at coordinate (i, j) in the USB, then $0 \leq i \leq Grid_w$ - 1 and $0 \leq j \leq Grid_h$ - 1. Rearranging and substituting values of i and j in (3.9, 3.10), we have, $i \leq B_w^R \times A_w^R - A_w^R$ and $j \leq B_h^R \times A_h^R - A_h^R$. This implies that $0 \leq i < B_w^R \times A_w^R$ and

$0 \leq j < B_h^R \times A_h^R$. Even if the maximum possible values of i and j are considered and substituted in (3.16-3.18), we have $PF_A_i = 0$, $PF_A_j = 0$, and $Fetch_Addr = 0$. Since, the $Fetch_Addr$ for a pixel with maximum coordinate is zero, the $Fetch_Addr$ for all other pixels in the USB will also be zero.

Theorem 3: Given that all pixels in the USB have identical (zero) $Fetch_Addr$ (from *Theorem 1*), for conflict free parallel access, no two distinct pixels in the USB have identical values of (m_p, m_q) and (o_p, o_q).

Proof: Let us assume two pixels located at (i_1, j_1) and (i_2, j_2) of the USB, such that $i_1 < i_2$, $j_1 < j_2$, where, $0 \leq \{i_1, i_2\} \leq Grid_w$ - 1, and $0 \leq \{j_1, j_2\} \leq Grid_h$ - 1. Let ($m_{p,1}$, $m_{q,1}$) and ($o_{p,1}$, $o_{q,1}$) correspond to BRAM and offset of a pixel at (i_1, j_1), respectively. Similarly, ($m_{p,2}$, $m_{q,2}$) and ($o_{p,2}$, $o_{q,2}$) correspond to that of (i_2, j_2).

To prove this theorem by contradiction, let us assume that the two pixels have identical values of (m_p, m_q) and (o_p, o_q). If the two pixels have identical values of m_p (i.e., $m_{p,1} = m_{p,2}$), then from (3.11) and *Theorem 1* we get (4.3), where k is a positive integer ($k \geq 0$). Since $0 \leq \{i_1, i_2\} \leq Grid_w$ - 1, we get (4.4). Substituting value of $Grid_w$ - 1 from (4.4) in (3.9), we get (4.5-4.6).

$$\left\lfloor \frac{i_2}{A_w^R} \right\rfloor - \left\lfloor \frac{i_1}{A_w^R} \right\rfloor = k \times B_w^R \tag{4.3}$$

$$i_2 - i_1 \leq Grid_w - 1 \tag{4.4}$$

$$i_2 - i_1 \leq B_w^R \times A_w^R - A_w^R \tag{4.5}$$

$$\Rightarrow \frac{i_2}{A_w^R} - \frac{i_1}{A_w^R} \leq B_w^R - 1 \tag{4.6}$$

Since the address generation involves integer divisions, the floor operation in eq. (4.3) is similar to the division operation in (4.6), and substituting the right-hand side of (4.3) in

(4.6) we get (4.7).

$$k \times B_w^R \leq B_w^R - 1 \tag{4.7}$$

Equation (4.7) holds true only for $k = 0$. If the two pixels have identical values of o_p (i.e. $o_{p,1} = o_{p,2}$), then from (3.19) we get (4.8). Where, k' is a positive integer ($k' \geq 0$). Substituting the value of i_2 computed from (4.8) in (4.3) where $k = 0$, we get (4.9). Equation (4.9) holds true only when $k' = 0$ and substituting this value of k' back in (4.8) we get $i_1 = i_2$. Similarly we can prove that $j_1 = j_2$.

$$i_2 - i_1 = k' \times A_w^R \tag{4.8}$$

$$\left\lfloor k' + \frac{i_1}{A_w^R} \right\rfloor = \left\lfloor \frac{i_1}{A_w^R} \right\rfloor \tag{4.9}$$

The above mathematical identities prove that two pixels at (i_1, j_1) and (i_2, j_2) inside a USB cannot be distinct if they have identical values of (m_p, m_q) and (o_p, o_q). The above theorems prove that when two pixels have identical values of (m_p, m_q) and *Fetch_Addr* during fetch operation, they never have identical values of *Fetch_offset*.

Chapter 5

Performance Estimation Model

This chapter presents an analytical model to estimate the performance of the memory subsystem derived from the proposed memory architecture template for given N_{PE} parallel PEs, FBM algorithm, frame size, and block size. The performance is measured in terms of fps. Test FBM algorithms are MDHC [2] search and UMHexagonS [1]. The video sequence under test consists of integer pixel and has a resolution of 1920 $\times$ 1080 pixels (1080p).

The fps is calculated from the operating frequencies of the memory subsystem. The memory subsystem operates at two different clock frequencies, f_{load} and f_{fetch}. The f_{load} and f_{fetch} correspond to the frequencies at which pixels are loaded to and fetched from the FBs, respectively. In order to keep the operating frequencies high, the load and fetch operations are performed in a pipelined order. The load operation has PLU and frame buffer as pipeline stages, whereas the fetch operation has PFU, frame buffer, and PSRM. It is observed that f_{fetch} will always be less than f_{load} because of the multiplexers and rotators in the PSRM. In order to compute SAD values, PEs are also included in pipeline stages of the fetch operation. This requires partitioning the PE into pipeline stages to avoid slowing down the f_{fetch}. The number of pipeline stages in the PE (PE_{stages}) is computed using (5.1), where PE_{delay} is the critical path delay of the PE and Clk_F_{delay} is the delay of Clk_F signal. The PE_{stages} adds to the total latency of the fetch operation ($F_{latency}$).

$$PE_{stages} = \left\lceil \frac{PE_{delay}}{Clk_F_{delay}} \right\rceil \tag{5.1}$$

The memory subsystem along with the PEs, can concurrently fetch, rearrange, and compute SAD values in different pipeline stages after incurring the initial latency ($F_{latency}$). The number of clock cycles ($search_{cycles}$) required by the memory subsystem to execute a search pattern is computed using (5.2), where n is the number of anchor point in the search

pattern being accelerated by N_{PE} parallel PEs. The inverse of the first term without a floor function in the right-hand side of (5.2) indicates the reduction achieved in the number of clock cycles required to execute a search pattern through parallel PEs.

$$search_{cycles} = \left\lceil \frac{n}{N_{PE}} \right\rceil + F_{latency} - 1 \tag{5.2}$$

For an FBM algorithm with N_s number of search patterns, the time required to compute SAD for all the current frame blocks in a frame ($Frame_{duration}$) is computed using (5.3), where $loop_k$ is iteration count of the kth search pattern and $search_{cycle,k}$ is the number of clock cycles required for executing the kth search pattern.

$$Frame_{duration} = \sum_{k=0}^{N_s-1} (search_{cycles,k} \times loop_k) \times \frac{framesize}{blocksize} \times f_{fetch} \tag{5.3}$$

It should be noted that prior to the initiation of SAD computations, the search area which holds the required candidate blocks must be loaded in the RFB. While these candidate blocks are being fetched and processed at f_{fetch} frequency, due to higher f_{load} frequency the overlapping search area of the adjacent current frame block can be concurrently loaded in the RFB. This process can be continued for the entire duration of the frame processing. If the width and height of the search range is s_w and s_h, respectively, the initial time required to load the search range into RFB ($Load_{duration}$) is computed using (5.4). Equations (5.3) and (5.4) suggests that $Load_{duration} << Frame_{duration}$ and can be ignored while computing fps in (5.5).

$$Load_{duration} = \left\lceil \frac{s_w \times s_h}{8} \right\rceil \times f_{load} \tag{5.4}$$

$$fps = \frac{1}{frame_{duration}} \tag{5.5}$$

Chapter 6

Bounded Set Algorithm

As discussed in Chapter 3, the CFB is derived from the size of the current frame block. The CFB is optimized to this block size because its location is known at the design time. However, the RFB derived from the size of the USB using (3.9, 3.10) can be subjected to optimization through SBs. A SB which provides candidate blocks to N_{PE} PEs is created by forming a group of at most N_{PE} anchor points. If the anchor points in a group are relatively far from each other in the search pattern, the size of the SB will be large. This will result in a larger $Grid_w$ and $Grid_h$ (from (3.4, 3.5)) and ultimately larger number of BRAMs ($B_w^R \times B_h^R$) to realize the RFB. As $B_w^R \times B_h^R$ increases, the size of the pixel block $((B_w^R \times A_w^R) \times (B_h^R \times A_h^R))$ received from the RFB also increases and so does the LUTs required by the multiplexers of the PSRM to select a $(p \times q)$ candidate block. The replication of the PSRM for every PE causes the footprint of the memory subsystem derived for these parameters to increase significantly. Although the number of LUTs utilized by the PFUs and PLUs is also affected by the number of BRAMs in the frame buffers, they have a negligible impact on the overall footprint when compared to the PSRMs. In order to keep the footprint of the derived memory subsystem as small as possible, an efficient grouping of anchor points to create smaller SBs is required. The number of possible groupings of anchor point for a search pattern with n anchor points executed by N_{PE} PEs is computed using (6.1).

$$number\ of\ possible\ groupings = \frac{\prod_{k=0}^{N_{SB}-1} {}^{n-k\times N_{PE}}\mathrm{C}_{N_{PE}}}{N_{SB}!} \tag{6.1}$$

There are 8.811×10^{81} groupings of anchor points possible, when an uneven multi-hexagon-grid search with 88 anchor points of UMHexagons [1] is executed by three parallel PEs. Due to the vast number of possible solutions, an algorithm capable of finding the best

grouping of anchor points resulting in a smaller RFB is very crucial. The BS algorithm (algorithm 6.1) invented by Clements et al. [16] finds a grouping of anchor points from which parameters ($B_w^R,\ B_h^R,\ A_w^R,\ and\ A_h^R$) of the RFB resulting in a smaller footprint can be derived. The resulting RFB assists in reducing the footprint of the memory subsystem derived from the memory architecture template. For the readability of the thesis, the BS algorithm proposed by Clements et al. [16] (under review) is discussed in this chapter.

Bounded Set Algorithm

For a given N_{PE}, set of n anchor points of a search pattern, and USB dimensions ($Grid_w, Grid_h$), the BS algorithm generates a list of group of anchor points (AP_Groups) such that: (i) Each group has nearly N_{PE} anchor points,[1] and (ii) Every anchor point of the search pattern is present in at least one group. The BS algorithm may or may not generate a valid grouping of anchor points for a given ($Grid_w, Grid_h$), therefore a flag ($GroupSuccess$) is set to indicate the success of the algorithm. The BS algorithm is executed for each combination of ($Grid_w, Grid_h$) varying from (p, q) to (s_w, s_h). The BS algorithm is executed for every search pattern of the FBM algorithm and the common solution (in terms of $B_w^R,\ B_h^R,\ A_w^R,\ and\ A_h^R$) which results in the smallest value of $B_w^R \times B_h^R$ are considered as the final parameters of the of RFB. The following definitions are required for the understanding of the BS algorithm:

Bounded Set: The bounded set is defined as the set of anchor points found within a rectangular grid of ($Grid'_w \times Grid'_h$) pixels anchored at the coordinate (x, y) of the reference frame, where $Grid'_w$ and $Grid'_h$ are computed using (6.2). The two properties associated to a BS are *Uniquness* and *Eligiblity*.

$$Grid'_w = Grid_w - p + 1 \qquad Grid'_h = Grid_h - q + 1 \tag{6.2}$$

[1] The goal of the algorithm is to create groups with N_{PE} anchor points. However, if n is not an integral multiple of N_{PE}, at least one group will have fewer than N_{PE} anchor points.

Uniqueness: For a given list of BSs, a unique BS is not a subset of any other BS in the list.

Eligibility: An eligible BS has at least N_{PE} number of anchor points.

The number of unique and eligible BSs that an anchor point is contained in is an indicator of how many possible ways in which that anchor point can be grouped.

Algorithm 6.1 BS

Input: Set (S) of anchor points of a search pattern, N_{PE} number of parallel PEs, size of the rectangular grid ($Grid'_w, Grid'_h$).
Output: List of group of anchor points (AP_Groups) and algorithm success flag ($GroupSuccess$)
1. $GroupSuccess \leftarrow 1$.
2. **while** ($GroupSuccess$ and S is not empty) **do**
    a. Generate an intial list of BSs from S.
    b. Remove the non unique BSs.
    c. **if** a valid group of anchor points can be created from the list of unique BSs **then**
        i. Add this group to AP_Groups.
        ii. Remove the grouped anchor points from S.
    else
        i. $GroupSuccess \leftarrow 0$.
    end if
  end while
3. return AP_Groups and $GroupSuccess$.

In algorithm 6.1, an initial list of BS is generated by placing the top left corner (origin) of $Grid'_w \times Grid'_h$ grid at every coordinate contained inside the search area. The non-unique BSs are removed from the initial list. A group of anchor points is created by selecting its first anchor point (AP_{seed}). The selection of AP_{seed} is a vital step as it affects the selection of other (N_{PE} - 1) anchor points in the same group. The selection process is guided by the following rules in decreasing order of importance:

1. For an anchor point to be a part of the group, it must be contained in at least one BS that also contains all the anchor points in the group.

2. An anchor point must be contained in the fewest number of BSs.

3. An anchor point contained in the BS must have the fewest number of anchor points.

4. Once AP_{seed} is selected for a group, any anchor point with the shortest Euclidean distance from it can be the part of the group.

Rule 1 ensures that the anchor point added to the group fits within the input grid. Rules 2 and 3 ensures that the anchor points which have the least possibilities to be a part of a different group are selected first. Rule 4 ensures the possibility of creating subsequent group of anchor points that fit inside the input grid. The AP_{seed} is selected based on the second rule and may have three possibilities in the following order of priorities:

A. The fewest number of BSs is equal to zero: This possibility arises when some anchor points belong to unique, but not eligible BSs. In this situation, no solution can be found for a given N_{PE} and grid dimension, and the *GroupSuccess* flag is set to zero. However, if n is not an integral multiple of N_{PE}, then there will be one (or more) control steps in the execution of a search pattern, where certain PEs will remain idle. The total number of idle steps ($idlePEs$) allowed to the PEs during an execution of a search pattern is computed using (6.3). If $idlePEs$ is four, then two PEs can be idle during control step 1, and two more PEs can be idle during control step 2. When $idlePEs$ is more than zero, a successively smaller group of anchor points is attempted to be created by selecting AP_{seed} from unique, but not eligible BSs. When a group of $N_{PE} - k$ anchor points is created, the $idlePEs$ is reduced by k. If $idlePEs$ reduces to zero, before all anchor points are grouped, a valid group cannot be formed and the *GroupSuccess* flag is set to zero. If more than one anchor point is contained in unique, but not eligible BSs, then one such point is selected randomly to be AP_{seed}.

$$idlePEs = \left\lceil \frac{n}{N_{PE}} \right\rceil \times N_{PE} - n \qquad (6.3)$$

B. The fewest number of BSs is equal to one: Rule 3 is applied to select AP_{seed}.

C. The fewest number BSs is greater than one: Any random anchor point present in the fewest number of BSs can be selected as AP_{seed}.

Once, AP_{seed} is selected, Rules 1-4 are used to select the remaining $N_{PE} - 1$(or less) anchor points in the group. The BS algorithm continues creating group of anchor points until all anchor points have been grouped or a zero on the *GroupSuccess* flag is encountered. The BS algorithm is attempted for various dimensions of $Grid'_w \times Grid'_h$ and the smallest

dimension which results in the valid group of anchor points is considered the best solution. The Number of BRAMs (B_w^R and B_h^R) in the RFB is computed using (3.9, 3.10, 6.2) for the following combinations of (A_w^R, A_h^R) : (1, 1), (1, 2), (1, 4), (2, 1), (2, 2), and (4, 1). The combination of (A_w^R, A_h^R) that results in the least value of $B_w^R \times B_h^R$ is selected as the parameters of the RFB. The BS algorithm is executed for each search pattern of an FBM algorithm. The least common $(Grid'_w, Grid'_h)$ which result in a successful group of anchor points for all search patterns is considered for deriving the RFB.

Chapter 7

Results

This chapter compares the memory subsystem (hereafter referred to as the proposed memory subsystem) derived from the proposed memory architecture template with that of Vanne et al. [6], Kuzmanov et al. [3], and Beric et al. [14, 15] for accelerating the test FBM algorithms. The memory subsystems are compared based on the maximum number of PEs supported by each memory subsystem for a given set of FPGA resources. The memory subsystems proposed by Vanne et al., Kuzmanov et al., and Beric et al. are first derived to support one PE, and then replicated to support multiple PEs for parallel execution. The replication makes these memory subsystems independent of search pattern, and therefore FBM algorithms. The performance of the proposed memory subsystem is estimated in terms of fps. The BS algorithm explores the proposed architecture template for a given FBM algorithm, N_{PE} PEs, and size of current frame block ($p \times q$). The resulting memory subsystem is implemented in Verilog and synthesized using Xilinx ISE 10.1. The Virtex 4 LX160 FPGA is used as the target architecture. The available resources in this FPGA are, $R_{available}$ = 135168 LUTs and 288 BRAMs.

In the event of accelerating a search pattern using one PE, all the SBs have only one candidate block and so their sizes are identical and equivalent to the size of the candidate block. Since, there is only one candidate block in each SB, their sizes are also not affected by the relative position of the anchor points of a search pattern, making the derived RFB independent of the search pattern and the FBM algorithm. Therefore, a memory subsystem derived for accelerating one FBM algorithm using a single PE can also accelerate another FBM algorithm. Table 7.1 shows BRAMs and LUTs required by the memory subsystems to accelerate test FBM algorithms using one PE. The FBM algorithms are executed for five different block sizes. Only the BRAMs in the RFB and LUTs in the PSRM (without

Table 7.1: Comparison of proposed memory subsystem with other memory subsystems for a single PE.

Memory subystem	component	Block size ($p \times q$)				
		4×4	8×4	8×8	16×8	16×16
Vanne et al. [6], Kuzmanov et al. [3]	RFB BRAMs = $p \times q$	16	32	64	128	256
	PSRM (LUTs)	512	1280	3072	7168	16384
Beric et al. [14, 15]	RFB (BRAMs) $< p \times q$	8	12	24	40	80
	PSRM (LUTs)	640	1792	4096	9592	21232
Proposed	RFB (BRAMs) $< p \times q$	8	12	24	40	80
	PSRM controller (LUTs)	3	10	10	21	21
	PSRM without controller (LUTs)	640	1632	3782	8398	18848
	BRAMs for PFU and PLU	4	4	4	4	4
	Others (LUTs)	140	158	256	339	567
	Total BRAMs	12	16	28	44	84
	Total LUTs	783	1800	4048	8758	19436
Vanne et al. [6], Kuzmanov et al. [3], Beric et al. [14, 15], and proposed memory subsytem	CFB (BRAMs)	4	8	16	32	64

controller) of other memory subsytems are estimated because (i) due to powers of two values of p and q in the memory subsystems proposed by Vanne et al. and Kuzmanov et al., the address generation unit for accessing the RFB and controller for the PSRM require negligible LUTs; and (ii) due to lack of implementation details on AGU and PSRM proposed by Beric et al. The CFB proposed by Vanne et al. is similar to the CFB of the proposed memory subsystems and requires few LUTs for addressing. The memory subsystems proposed by Kuzmanov et al. and Beric et al. do not discuss about the CFB and its addressing; therefore the CFB in those cases has been assumed to be derived using (3.6-3.8).

The memory subsystem proposed by Vanne et al. and Kuzmanov et al. consists of $p \times q$ BRAMs, which are configured to store one pixel per address location. This configuration allows the BRAMs to only fetch the required pixels. However, the fetched pixels require shuffling to compute correct SAD values. The rotators to shuffle the fetched pixels require fewer LUTs than the PSRM of other memory subsystems. Therefore, this memory subsystem requires fewer LUTs than other memory subsystems but due to $p \times q$ BRAMs, it requires the largest number of BRAMs. In comparison to the memory subsystem proposed by Beric et al., the RFB in the proposed memory subsystem requires similar number of BRAMs, but rearranges the required $p \times q$ blocks of pixels using fewer LUTs through

the novel PSRM. However, in case of 4 × 4 blocks the LUT requirements are the same, because the rotator in the reordering logic proposed by Beric et al. and the multiplexers in the proposed PSRM require similar amount of LUTs. The proposed memory subsystem features unrestricted values of B_w and B_h, therefore the modulo and division operations for non-powers of two value of B_w and B_h in eqs. (3.11, 3.12, 3.13, 3.16, and 3.17) are performed using four additional dual port BRAMs. In spite of considering the additional BRAMs, the BRAM requirement of the proposed memory subsystem is always less than proposed by Vanne et al. and Kuzmanov et al. It is also observed that the total LUT requirement of the proposed memory subsystem is always less than the memory subsystem proposed by Beric et al., except for 4 × 4 and 8 × 4 blocks (due to the other components of the proposed memory subsystem).

The maximum number of required N_{PE} to accelerate a normal execution of FBM algorithm is equal to the number of anchor points in the largest search pattern. This value is found to be equal to 13 and 88 for MDHC [2] and UMHexagonS [1], respectively. Thus, there is a limitation to the number of parallel PEs, which can accelerate the normal execution of an FBM algorithm. However, with the memory subsystem supporting sufficient number of PEs, the multiple small cross searches in MDHC [2], and extended hexagon searches in UMHexagonS [1] can be executed speculatively. Speculative execution refers to the execution of the candidate blocks of future iterations of a search pattern along with that of the current iteration. Figure 7.1 shows the speculative execution of two iterations of hexagon search (fig. 7.1(a)) and diamond search (fig. 7.1(b)) belonging to the extended hexagon search of UMHexagonS [1]. It is found that 20 PEs are required for speculative execution of two consecutive small cross searches in MDHC [2], and 18 and 12 for two steps of consecutive extended hexagon searches in UMHexagonS [1].

A comparison of maximum N_{PE} supported by the proposed and other memory subsystems to accelerate test FBM algorithms for different sizes of current frame block is shown in Table 7.2. The resources of all the PEs are combined with the resources for the memory subsystem to compute the total resource utilization (L_{MEM}, B_{MEM}) using (7.1, 7.2),

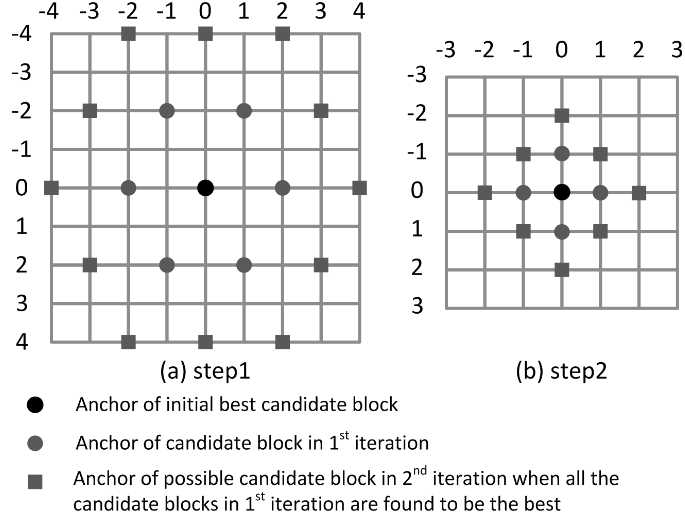


Fig. 7.1: Illustration of speculative execution in UMHexagonS.

where L_z and B_z represent the number of LUTs and BRAMs utilized by component z of the memory subsystem, respectively. A PE computes the SAD using an adder tree proposed by Jehng et al. [17]. In case of the memory subsystems proposed by Vanne et al., Kuzmanov et al., and Beric et al., the maximum N_{PE} is estimated by replicating RFB, PSRM, and PE multiple times until all the resources on the FPGA are exhausted.

$$L_{MEM} = L_{PLU} + L_{PFU} + L_{RFB} + L_{CFB} + N_{PE} \times (L_{PSRM} + L_{PE}) \tag{7.1}$$

$$B_{MEM} = B_{RFB} + B_{CFB} + 4 \tag{7.2}$$

It is observed in Table 7.2 that the memory subsystem proposed by Vanne et al. and Kuzmanov et al. for a given set of FPGA resources supports the least number of PEs for both FBM algorithms, because it requires the largest number of BRAMs to realize its RFB. Lesser the number of PEs, lesser the amount of parallelism, and hence slower will be the execution of an FBM algorithm resulting in lower fps for a given video sequence. In case of UMHexagonS [1], the proposed memory system supports fewer PEs than the memory subsystem proposed by Beric et al. for 4×4 and 8×4 blocks. Since the proposed memory

Table 7.2: Maximum number of PEs supported by the proposed and existing memory subsystems on a Virtex-4 LX160 FPGA.

Block size	Proposed Memory subsystem		Vanne et al. [6],	
	MDHC [2] (N_{PE})	UMHexagonS [1] (N_{PE})	Kuzmanov et al. [3] (N_{PE})	Beric et al. [14,15] (N_{PE})
4×4	44	23	17	35
8×4	27	21	8	23
8×8	16	12	4	11
16×8	8	7	2	6
16×16	4	3	0	2

subsystem features a single RFB and multiple copies of PSRM, in case of 4×4 block the replication of PSRM for more than 23 PEs causes the number of required LUTs to exceed L_{MEM}. The same is the case for 8×4 block.

Table 7.3 shows f_{load} and f_{fetch} of the proposed memory subsystem implemented to support maximum N_{PE} for different block sizes and test FBMAs. The operating frequencies are obtained from the post place and route reports generated using Xilinx ISE 10.1. The performance of the proposed memory subsystem in terms of fps for integer pixels of a 1080p (1920×1080) video sequence is estimated as shown in Table 7.4. The performance of the proposed memory subsystem is estimated using the performance model discussed in the previous chapter. The performance of other memory subsystems is not estimated because (i) the memory subsystems proposed by Vanne et al. and Kuzmanov et al. are targeted towards Altera Stratix and Virtex II FPGA devices, respectively, and the delay of linear access memory proposed by Kuzmanov et al. cannot be estimated for Virtex 4 FPGAs; and (ii) the memory subsystem proposed by Beric et al. is targeted towards ASIC and it is not possible to obtain frequency related information for its FPGA-based implementation. Some assumptions were made to reduce the complexity of the estimations, which are discussed below.

In case of MDHC [2], it is assumed that the search pattern, if iterated, is executed 10 times before satisfying the exit condition. MDHC [2] executes different sequence of search patterns based on the type of motion. In order to estimate performance for the worst case, it is assumed that the motion of each current frame block is fast block motion,

Table 7.3: Operating frequency of PLU and memory subsystem for different block sizes and FBM algorithms.

Operating Frequenciens (MHz)	Block size				
	4×4	8×4	8×8	16×8	16×16
$PLU_{frequency}$	277.23	231.16	173.13	141.64	185.08
	116.33	141.72	99.33	90.08	52.22
$MEM_{frequency}$	161.66	174	174	147.30	163.4
	122.04	101.66	130.04	100.85	100.02

Table 7.4: Number of frames processed per second (fps) by the proposed memory subsystem through normal and speculative execution of FBM algorithms.

FBM algorithms	Block size				
	4×4	8×4	8×8	16×8	16×16
MDHC *Normal Execution* (fps)	8	20	29	52	55
MDHC *Speculative Execution* (fps)	11	27	NA	NA	NA
UMHexagonS *Normal Execution* (fps)	8	13	33	49	73
UMHexagonS *Speculative Execution* (fps)	14	24	38	NA	NA

which requires execution of the largest number of search patterns. In UMHexagonS [1] the extended hexagon search is also assumed to execute for 10 iterations. UMHexagonS [1] executes a different sequence of search patterns for different block sizes. However, to estimate performance for the worst case, a similar sequence of search patterns for different block sizes is assumed. It is also assumed that all the anchor points of the search patterns lie within the search area. The speculative execution is performed when sufficient PEs can be supported by the memory subsystem. The N_{PE} required for speculative execution of two consecutive iteration of small cross search pattern in MDHC [2] is 20, and two steps of extended hexagon search in UMHexagonS [1] is 18 and 12, respectively.

The presence of multiple PEs accelerates the execution of a search pattern by decreasing the number of required clock cycles ($search_{cycles}$). However, the memory subsystem incurs a latency ($F_{latency}$) whenever a new or next iteration of a search pattern is executed, increasing the time to process a frame ($Frame_{duration}$). Since the search patterns in MDHC [2] and UMHexagonS [1] are executed for 21 and 22 times, respectively, the

speculative execution of these FBM algorithms decreases the time to process each frame by overlapping the initial latency incurred for consecutive iterations. The performance in terms of fps was observed to increase by 49% (on an average) through speculative execution of two consecutive iterations of a search pattern. The number of current frame blocks in a frame also affects the performance of the memory subsystem. For a given frame, the memory subsystem must process fewer number of 16×16 current frame blocks than 4×4, and therefore the average performance of the memory subsystem for a 16×16 current block is 8 times better than that for 4×4.

The memory subsystems derived from the proposed architecture template features 100% device utilization unlike Dutta et al. Unlike Dutta et al. and Vleeschouwer et al., the proposed architecture template is applicable to any FBM algorithm for a range of PEs, provided there are sufficient FPGA resources. The template can provide conflict free parallel access to multiple candidate blocks unlike Peng et al. [5]. Unlike Kuzmanov et al., Vanne et al., and Beric et al., the proposed memory subsystem template does not require data replication to support multiple PEs. Like Dutta et al. [12,13] the maximum number of PEs supported by the proposed template depends on the search pattern. The number of PEs can vary from one to the number of anchor points in the largest search pattern of an FBM algorithm.

Chapter 8

Conclusion and Future Works

8.1 Conclusion

A novel memory architecture template to accelerate a given FBM algorithm through multiple parallel PEs is presented in this thesis. Since the proposed memory architecture template is independent of the search pattern, number of processing elements, and block size, it can be used to accelerate various fast block matching algorithms. The memory architecture template can be parameterized for a given FBM algorithm, number of parallel processing elements, and block size to obtain a memory subsystem. The salient feature of the memory architecture template is its capability to provide conflict free parallel accesses to multiple candidate blocks without any need of data replication. The memory subsystem derived from the proposed memory architecture template is unique in its efficient PSRM which accounts for its smaller resource requirements (LUTs and BRAMs) when compared to other memory subsystems. These features make the proposed memory architecture template a suitable choice over the existing memory subsystems and for accelerating FBM algorithms. Due to FPGA-based implementation, the derived memory subsystems also provide higher operations per watt than state-of-the-art PCs. The memory architecture template is explored through a bounded set algorithm. However, it can also be explored by any other exploration algorithm. The performance of the derived memory subsystems is estimated in terms of number of frames processed per second.

Results are provided for two FBM algorithms (Mixed Diamond Hexagon and Cross search (MDHC) and UMHexagonS) in terms of support for maximum number of processing element and performance (fps). It is observed that the proposed memory subsystem can support $1 \sim 27$ more PEs than other published memory subsystems for given FPGA re-

sources. It is also observed that the proposed memory subsystem can process integer pixels of a 1080p video sequence at a varying rate of $8 \sim 73$ fps through normal execution of fast block matching algorithms on Virtex 4 LX160 FPGA device. The performance in terms of fps was observed to increase by 49% (on an average) through speculative execution of two consecutive iterations of a search pattern.

8.2 Future Works

A controller is required to control the memory subsystem. The controller must generate addresses to fetch pixels from an external memory. It must also provide the coordinates of the pixels (fetched from the external memory and loaded into the RFB and CFB) to the PLU_{REF} and PLU_{CUR}, respectively. The sub-module of the controller for generating control signals for PFU_{REF} and PSRMs can be implemented using a finite state machine, where every state is dedicated to one group of anchor points. The BS algorithm derives multiple groups of anchor points for all search patterns of an FBM algorithm. One of the outputs for each state must be the SB offset for the group of anchor points and must be provided to the PFU_{REF}. The other outputs must be anchor point offsets for each anchor points in the group and must be sent to the PSRMs. The controller must also provide coordinate of the top-left pixel of the current frame block to the PFU_{CUR} for fetching a current frame block from the CFB. The controller must disable some PEs in case the number of anchor points in a search pattern is not an integral multiple of N_{PE}. The controller must also disable the PEs in case few anchor points of a search pattern fall outside the search range. The number of BRAMs in the RFB and CFB are not sufficient to store an entire frame; therefore these frame buffers store a portion of frame at any given time. The contents of these BRAMs are therefore constantly overwritten. This requires the controlling of the pixels to be loaded from an external memory to the on-chip frame buffers to avoid overwriting the address location which have candidate blocks required for SAD computations.

Footprint of the PE is also considered while determining the number of PEs supported by the proposed architecture template. Presently a tree-based PE architecture has been

assumed for this experiment, in future different PE architectures can also be assumed. The FBM algorithms are executed for different size of current frame blocks (varying from 4×4 to 16×16), but the derived memory subsystem is limited to one block size. Therefore, the exploration technique must be modified to derive memory subsystems to support variable sizes of current frame blocks. For a given FBM algorithm, support to maximum number of PEs decreases with the increase in the size of the current frame block. Therefore, in order to support variable block sizes and variable number of PEs, a configurable PE architecture is required. If the above concerns are addressed then an Intellectual Property (IP) core for creating FPGA based memory subsystem for accelerating FBM algorithm can be created. The inputs to the IP core generator will be the search patterns of an FBM algorithm, number of parallel PEs, and available FPGA resources.

References

[1] Z. Chen, P. Zhou, and Y. He, "Fast integer pel and fractional pel motion estimation for jvt," in 6th *Meeting on Joint Video Team (JVT) of ISO/IEC MPEG & ITU-TVCEG*, vol. JVT-F017, Dec. 5-13, 2002.

[2] C. Duanmu, X. Chen, Y. Zhang, and S. Zhou, "Mixed diamond, hexagon, and cross search fast motion estimation algorithm for h.264," in *IEEE International Conference on Multimedia and Expo*, pp. 761–764, Apr. 2008.

[3] G. Kuzmanov, G. Gaydadjiev, and S. Vassiliadis, "Multimedia rectangularly addressable memory," *IEEE Transactions on Multimedia*, vol. 8, no. 2, pp. 315–322, Apr. 2006.

[4] C. De Vleeschouwer, T. Nilsson, K. Denolf, and J. Bormans, "Algorithmic and architectural co-design of a motion-estimation engine for low-power video devices," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 12, no. 12, pp. 1093–1105, Dec. 2002.

[5] J. Peng, X. Yan, D. Li, and L. Chen, "A parallel memory architecture for video coding," *Journal of Zhejiang University - Science A*, vol. 9, pp. 1644–1655, Dec. 2008.

[6] J. Vanne, E. Aho, T. Hamalainen, and K. Kuusilinna, "A parallel memory system for variable block-size motion estimation algorithms," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 18, no. 4, pp. 538–543, Apr. 2008.

[7] Xilinx, "Virtex-4 family overview," `[http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf]`, Dec. 28, 2007.

[8] Altera, "Stratix iv device family overview," `[http://www.altera.com/literature/hb/stratix-iv/stx4_siv51001.pdf]`, June, 2009.

[9] Xilinx, "Virtex-4 fpga user guide," `[http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf]`, Dec. 1, 2008.

[10] Xilinx, "Design flow overview," `[http://www.xilinx.com/itp/xilinx7/books/data/docs/dev/dev0013_5.html]`.

[11] I. E. Richardson, *H.264 and MPEG-4 Video Compression - Video Coding for Next-generation Multimedia.* West Sussex, England: John Wiley & Sons, 2003.

[12] S. Dutta, W. Wolf, and A. Wolfe, "A methodology to evaluate memory architecture design tradeoffs for video signal processors," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 8, no. 1, pp. 36–53, Feb. 1998.

[13] S. Dutta and W. Wolf, "A flexible parallel architecture adapted to block-matching motion-estimation algorithms," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 6, no. 1, pp. 74–86, Feb. 1996.

[14] A. Beric, J. van Meerbergen, G. de Haan, and R. Sethuraman, "Memory-centric video processing," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 18, no. 4, pp. 439–452, Apr. 2008.

[15] A. Beric, R. Sethuraman, H. Peters, G. Veldman, J. van Meerbergen, and G. de Haan, "Streaming scratchpad memory organization for video applications," *International Conference on Circuits, Signals and Systems*, pp. 427–433, 2004.

[16] A. A. Clements, S. Chandrakar, A. Sudarsanam, and A. Dasu, "Methodology to design on-chip video buffers for fpga based fast block matching algorithms," *Electronic Letters, IET*, 2009, under review.

[17] Y.-S. Jehng, L.-G. Chen, and T.-D. Chiueh, "An efficient and simple vlsi tree architecture for motion estimation algorithms," *IEEE Transactions on Signal Processing*, vol. 41, no. 2, pp. 889–900, Feb. 1993.