FPGA IMPLEMENTATIONS OF MOTION ESTIMATION ALGORITHMS USING VIVADO HIGH-LEVEL SYNTHESIS

by

Firas Abdul Ghani

Submitted to the Graduate School of Engineering and Natural Sciences

in partial fulfillment of
the requirements for the degree of
Master of Sciences

Sabancı University

August 2017

FPGA IMPLEMENTATIONS OF MOTION ESTIMATION ALGORITHMS
USING VIVADO HIGH LEVEL SYNTHESIS

APPROVED BY:

Assoc. Prof. Dr. İlker Hamzaoğlu ....
(Thesis Supervisor)

Assist. Prof. Dr. Murat Kaya Yapıcı ....

Assist. Prof. Dr. Baykal Sarıoğlu ....

DATE OF APPROVAL: 01/08/2017

To my Mother and Father

ACKNOWLEDGEMENT

Foremost, I would like to express my sincere thanks to my advisor Dr. İlker Hamzaoğlu for his endless support during my MS study. I respect very much his suggestions and comments that improved my design skills more and more. His motivation, and immense knowledge enabled me to finish my thesis with very good results.

In addition, I would like to really thank Ercan Kalalı for his endless support and advices during my MS study. I also want to thank the other members of System-on-Chip Design and Testing Lab, Ahmet Can Mert, and Hasan Azgın for their support during my MS study.

I also want to thank my family and my wife who encouraged me a lot during my thesis.

Finally, I would like to acknowledge Sabancı University and Scientific and Technological Research Council of Turkey (TUBITAK) for supporting me with scholarships throughout my studies. This thesis was supported by TUBITAK under the contract 115E290.

FPGA IMPLEMENTATIONS OF MOTION ESTIMATION ALGORITHMS USING VIVADO HIGH LEVEL SYNTHESIS

Firas Abdul Ghani
Electronics, MS Thesis, 2017

Thesis Supervisor: Assoc. Prof. İlker HAMZAOĞLU

Keywords: HEVC, Fractional Interpolation, Motion Estimation, High-Level Synthesis

ABSTRACT

Joint collaborative team on video coding (JCT-VC) recently developed a new international video compression standard called High Efficiency Video Coding (HEVC). HEVC has 50% better compression efficiency than previous H.264 video compression standard. HEVC achieves this video compression efficiency by significantly increasing the computational complexity. Motion estimation is the most computationally complex part of video encoders. Integer motion estimation and fractional motion estimation account for 70% of the computational complexity of an HEVC video encoder. High-level synthesis (HLS) tools are started to be successfully used for FPGA implementations of digital signal processing algorithms. They significantly decrease design and verification time. Therefore, in this thesis, we proposed the first FPGA implementation of HEVC full search motion estimation using Vivado HLS. Then, we proposed the first FPGA implementations of two fast search (diamond search and TZ search) algorithms using Vivado HLS. Finally, we proposed the first FPGA implementations of HEVC fractional interpolation and motion estimation using Vivado HLS. We used several HLS optimization directives to increase performance and decrease area of these FPGA implementations.

HAREKET TAHMİNİ ALGORİTMALARININ VIVADO YÜKSEK SEVİYE SENTEZLEME İLE FPGA GERÇEKLEMELERİ

Firas Abdul Ghani
Elektronik Müh., Yüksek Lisans Tezi, 2017

Tez Danışmanı: Doç. Dr. İlker HAMZAOĞLU

Anahtar Kelimeler: HEVC, Kesirli Aradeğerleme, Hareket Tahmini, Yüksek Seviye Sentezleme

ÖZET

Joint Collaborative Team on Video Coding (JCT-VC) yüksek verimli video kodlama (HEVC) isminde yeni bir video sıkıştırma standardı geliştirdi. HEVC günümüzde kullanılan H.264 standardına göre 50% daha iyi performans sağlıyor. HEVC bu video sıkıştırma verimini hesaplama karmaşıklığını önemli ölçüde artırarak başarıyor. Hareket tahmini video kodlayıcıların hesaplama karmaşıklığı en fazla olan parçasıdır. Tam sayı hareket tahmini ve kesirli hareket tahmini, HEVC video kodlayıcının hesaplama karmaşıklığının %70'ni oluşturmaktadır. Yüksek seviye sentezleme araçları sayısal işaret işleme algoritmalarının FPGA gerçeklemelerinde başarılı bir şekilde kullanılmaya başladı. Tasarım ve doğrulama zamanını önemli ölçüde azalttılar. Bu nedenle, bu tezde, Vivado HLS kullanarak HEVC tam arama tam sayı hareket tahmininin ilk FPGA gerçeklemesini önerdik. Ardından, Vivado HLS kullanarak, iki hızlı arama algoritmasının (elmas arama ve TZ arama) ilk FPGA gerçeklemelerini önerdik. Son olarak, Vivado HLS kullanarak, HEVC kesirli aradeğerleme ve hareket tahmininin ilk FPGA gerçeklemelerini önerdik. Performansı artırmak ve FPGA gerçeklemelerinin donanım alanını azaltmak için birkaç HLS eniyileme direktifini kullandık.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

BRAM	Block RAM
CABAC	Context Adaptive Binary Arithmetic Coding
DCT	Discrete Cosine Transform
DST	Discrete Sine Transform
DVI	Digital Visual Interface
FPGA	Field Programmable Gate Array
HD	High Definition
HEVC	High Efficiency Video Coding
HM	HEVC Test Model
PSNR	Peak Signal to Noise Ratio
PU	Prediction Unit
SAO	Sample Adaptive Offset
TU	Transform Unit
UART	Universal Asynchronous Receiver/Transmitter
HLS	High Level Synthesis
SAD	Sum Of Absolute Differrence
TZA	TZ Search Algorithm

CHAPTER I

INTRODUCTION

1.1 HEVC Video Compression Standard

Since better coding effiency is required for high resolution videos, Joint Collaborative Team on Video Coding (JCT-VC) recently developed a new video compression standard called High Efficiency Video Coding (HEVC) [1, 2, 3]. HEVC provides 50% better coding efficiency than previous H.264 video compression standard. HEVC also provides 23% bit rate reduction for the intra prediction only case [4]. The video compression efficiency achieved in HEVC standard is not a result of any single feature but rather a combination of a number of encoding tools such as intra prediction, motion estimation, deblocking filter and entropy coder. Motion estimation is the most computationally complex part of video encoders. Integer motion estimation and fractional motion estimation account for 70% of the computational complexity of an HEVC video encoder.

The top-level block diagram of an HEVC encoder and decoder are shown in Figure 1.1 and Figure 1.2, respectively. An HEVC encoder has a forward path and a reconstruction path. The forward path is used to encode a video frame by using intra and inter predictions and to create the bit stream after the transform and quantization process. Reconstruction path in the encoder ensures that both encoder and decoder use identical reference frames for intra and inter prediction because a decoder never gets original images.

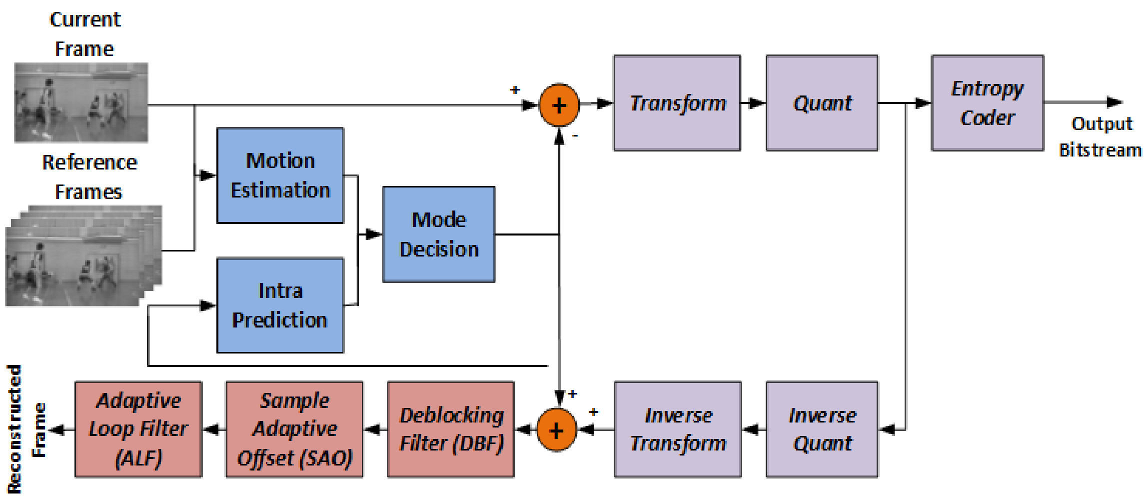


Figure 1.1 HEVC Encoder Block Diagram

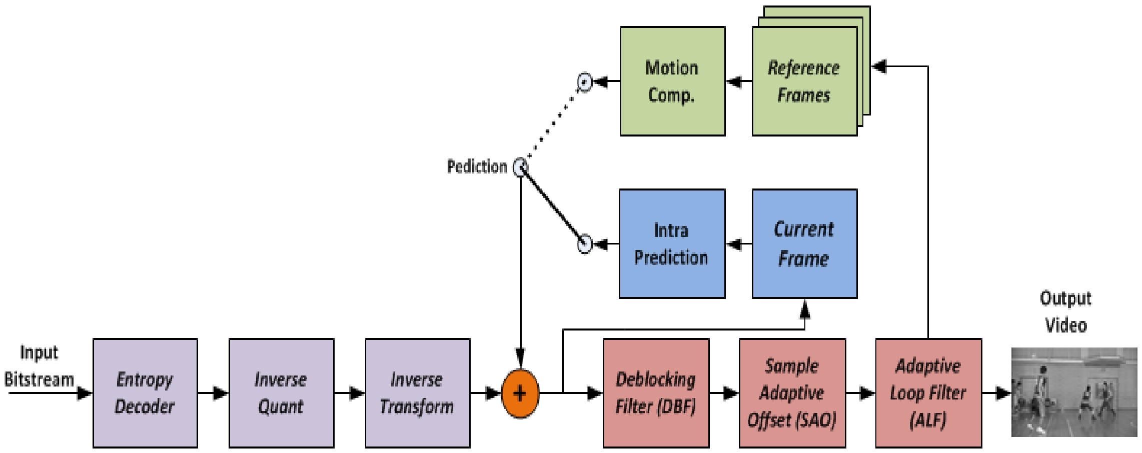


Figure 1.2 HEVC Decoder Block Diagram

In the forward path, frame is divided into coding units (CU) that can be an 8x8, 16x16, 32x32 or 64x64 pixel block. Each CU is encoded in intra or inter mode depending on the mode decision. Intra and inter prediction processes use prediction unit (PU) partitioning inside the CUs. Prediction unit (PU) sizes can be from 4x4 up to 64x64. Mode decision determines whether a PU will be coded intra or inter mode based on video quality and bit-rate. After mode decision determines the prediction mode, predicted block is subtracted from original block, and residual data is generated. Then, residual data transformed by discrete cosine transform (DCT) and quantized. Transform unit (TU) sizes can be from 4x4 up to 32x32. Finally, entropy coder generates the encoded bitstream.

Reconstruction path begins with inverse quantization and inverse transform operations. The quantized transform coefficients are inverse quantized and inverse transformed to generate the reconstructed residual data. Since quantization is a lossy process, inverse quantized and inverse transformed coefficients are not identical to the original residual data. The reconstructed residual data are added to the predicted pixels in order to create the reconstructed frame. DBF is, then, applied to reduce the effects of blocking artifacts in the reconstructed frame.

1.2 High-Level Synthesis

Recently, high-level synthesis (HLS) tools started to generate production quality register transfer level (RTL) implementations from high-level specifications. HLS tools improve productivity of hardware designers by reducing both design and verification time.

In this thesis, Xilinx Vivado HLS tool is used. It is one of the successful commercial HLS tools. It takes C, C++ or SystemC codes as input, and generates Verilog or VHDL codes. Design flow used in this thesis for the FPGA implementations of motion estimation algorithms using Xilinx Vivado HLS is shown in Figure 1.3. First, software models of HEVC video compression algorithms are developed using HEVC reference software video encoder (HM) 15.0 [5]. After the software models are verified with HEVC test sequences, C codes for HLS are developed. Then, the C codes are synthesized to Verilog RTL using Xilinx Vivado HLS tool. Several optimizations offered by Xilinx Vivado HLS tool are also used to increase performance and decrease area of the proposed FPGA implementations. The Verilog RTL codes are synthesized and mapped to a Xilinx Virtex 6 FPGA using Xilinx ISE 14.7. Finally, the FPGA implementations are verified with post place and route simulations.

Xilinx Vivado HLS tool provides C specification testbench to verify the code. This C testbench is used by the tool to verify that the functionality of the synthesized RTL is same as the functionality of the original C code. After verifying the functionality with C testbench, Vivado HLS tool generates hardware (Verilog or VHDL) testbench to verify the hardware. Then, HLS tool compares the output of C testbench and hardware testbench. If they are same, it indicates that the hardware is verified.

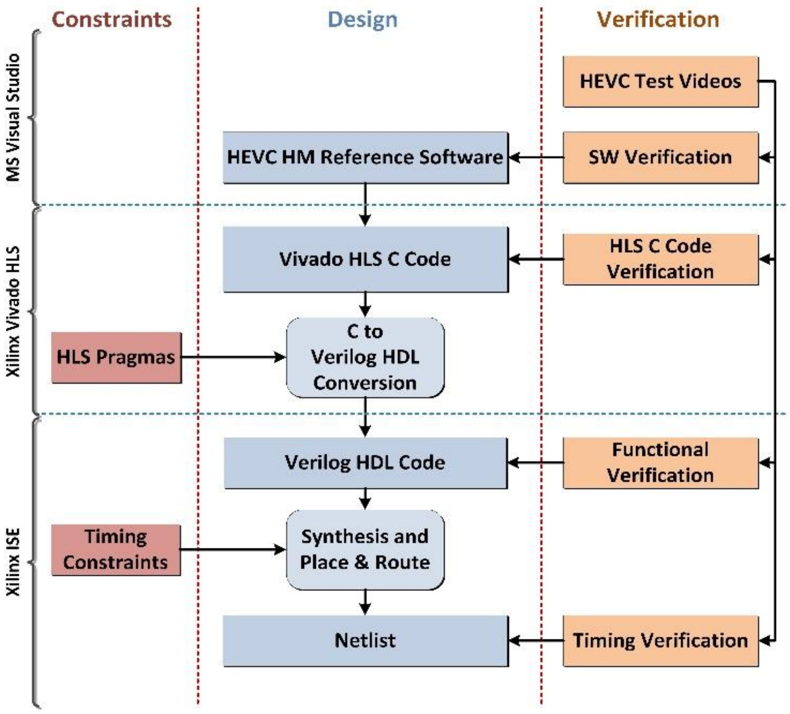


Figure 1.3 Xilinx Vivado HLS design flow

Table 1.1 Xilinx Vivado HLS Optimizations

	Optimizations (Pragmas)
Loop Optimizations	Loop Pipelining Loop Unrolling Loop Merge
Memory Control	Array Map Array Partition Resource
Resources	Allocation Resource

Xilinx Vivado HLS tool performs scheduling of operations, allocation of registers, and binding of operations to functional units. Xilinx Vivado HLS tool provides many optimizations (pragmas) for scheduling, allocation and binding. It also provides bit-accurate or cycle-accurate implementations. It allows adding specific RAM blocks, FIFOs, ROMs or specific DSP blocks. In addition it generates I/O interfaces to connect hardware modules with memories or other peripherals. Xilinx Vivado HLS tool offers these optimizations to increase performance and decrease area of HLS implementations. These optimizations can be grouped as shown in Table 1.1.

Loop Unrolling (LU) directive is used to increase performance using more resources. It creates multiple copies of loop body, and compute them in parallel. In this way, it decreases the loop iterations and increases the performance. However, loop unrolling may cause memory access problems in HLS designs.

Allocation (ALC) directive is used to specify the maximum number of resources that can be used in hardware. It forces the HLS tool to perform resource sharing. It therefore decreases the hardware area. Allocation can be used for addition, subtraction, multiplication, division, shift and comparison operations.

Pipeline (PIPE) directive performs pipelining to increase the performance. Xilinx Vivado HLS tool performs pipelining automatically. However, number of pipeline stages can also be defined for further performance increase.

Resource (RES) directive is used to specify which resource will be used to implement a variable such as an array, arithmetic operation or function argument. DSP elements, specific RAM blocks, FIFOs or ROMs can be used with resource directive.

Array map (AMAP) directive is used to map multiple small arrays into a single large array. The large array can be targeted to a single large memory (RAM or FIFO) resource. It is also used to control how (horizontal or vertical) data is stored in BRAMs.

Array partition (APAR) directive partitions the large arrays into multiple smaller arrays or individual registers for parallel data accesses.

Xilinx Vivado HLS tool also provides a specific library for designing bit-accurate (BIT) models in C codes.

A few HLS implementations for HEVC video compression standard are proposed in the literature [6]-[8]. A few HLS implementations for H.264 video compression standard are proposed in the literature [9]-[12]. There are a few HLS implementations based on MPEG reconfigurable video coding [13]-[14]. There are several HLS implementations for image and video processing algorithms such as sorting in the median filter [15]-[18].

1.3 Thesis Contributions

In this thesis, we proposed the first FPGA implementation of HEVC full search motion algorithm using HLS in the literature. The C codes given as input to Xilinx Vivado HLS tool are developed based on the HEVC reference software video encoder (HM) version 15 [5]. We used several optimizations offered by Vivado HLS to achieve real-time performance. The proposed FPGA implementation of HEVC full search motion estimation algorithm using HLS can process 30 full HD video frames per second for all PU sizes and for fixed search range (64x64). It can process 29 full HD frames per second for variable search ranges.

Fast search motion estimation algorithms are used to reduce computational complexity of motion estimation. Diamond Search (DS) and TZ Search (TZS) are very successful fast search motion estimation algorithms. Therefore, in this thesis, first FPGA implementations of DS and TZS algorithms using HLS in the literature are proposed. The proposed DS and TZS motion estimation FPGA implementations can process 127 full HD (1920x1080) and 46 full HD video frames per second, respectively.

We also proposed the first FPGA implementation of HEVC fractional interpolation and motion estimation using HLS in the literature. We used several optimizations offered by Vivado HLS to achieve real-time performance. The proposed HEVC fractional interpolation and HEVC fractional motion estimation FPGA implementations can process 45 quad full HD (3840x2160) and 46 full HD video frames per second, respectively.

1.4 Thesis Organization

The rest of the thesis is organized as follows.

Chapter II first explains FPGA implementations of HEVC full search motion estimation algorithm using Vivado HLS and presents the experimental results. It, then, explains FPGA implementations of two fast search (Diamon Search and TZ Search) motion estimation algorithms using Vivado HLS and presents the experimental results.

Chapter III explains FPGA implementations of HEVC fractional interpolation and fractional motion estimation algorithms using Vivado HLS and presents the experimental results.

Chapter IV presents conclusions and future work.

CHAPTER II

FPGA IMPLEMENTATIONS OF INTEGER MOTION ESTIMATION ALGORITHMS USING VIVADO HIGH-LEVEL SYNTHESIS

Motion estimation (ME) is used to remove temporal redundancy between current frame and reference frame that has been encoded previously. As shown in Figure 2.1, integer motion estimation (IME) divides the current frame into blocks and finds the motion vector (MV) for each block by determining the reference block in the reference frame that gives the smallest sum of absolute difference (SAD) for this block. Then, it calculates the difference between the current block and the best matching reference block, and encodes this residual and the motion vector.

HEVC standard divides the current frame into blocks called Prediction Units (PUs) for IME. In HEVC standard, 24 different PU sizes are defined. These PU sizes range from 4x8 or 8x4 to 64x64. This allows HEVC standard to do better compression than previous video compression standards.

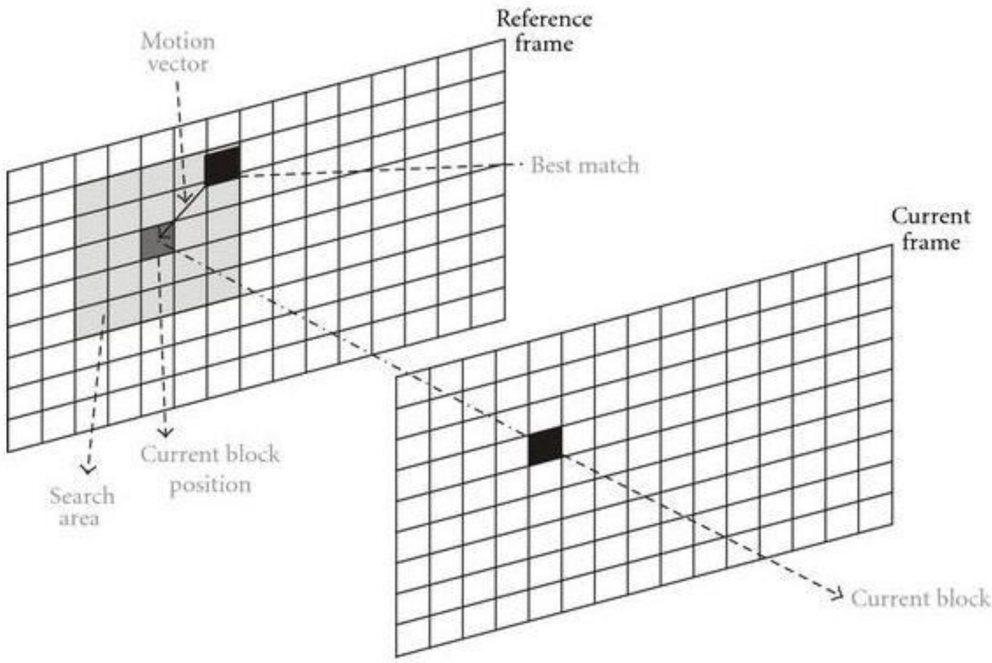


Figure 2.1 Integer Motion Estimation

2.1 FPGA Implementation of HEVC Full Search Motion Estimation Algorithm Using Vivado High-Level Synthesis

2.1.1 Full Search Motion Estimation Algorithm

Full Search (FS) algorithm exhaustively searches all search locations in the defined search window in the reference frame. Therefore, it finds the best MV in the search window. However, it is the most computationally complex motion estimation algorithm.

FS algorithm calculates the SAD value for each search location as shown in Equation 2.1.

$$SAD = \sum_{i=0}^{m} \sum_{j=0}^{n} \left| R_{ij} - C_{ij} \right| \qquad (2.1)$$

R is a pixel in the reference frame. C is a pixel in the current frame. It determines the search location with the minimum SAD value and the MV corresponding to this search location.

2.1.2 FPGA implementation

We, first, designed a full search IME hardware for fixed current block size (8x8) and fixed search range (16x16). In this hardware, 8 parallel absolute difference hardware calculate absolute differences for one column of 8x8 PU. After 8 iterations, SAD value is calculated by adding absolute difference values. 16x16 array stores all SAD values for comparison. Then, comparison unit compares SAD values, and determines the minimum SAD value and the corresponding motion vector.

Verilog RTL codes generated by Xilinx Vivado HLS tool for this HLS implementation are verified with post place and route simulations. The implementation results are shown in Table 2.1.

PIPE, LU, APAR and RES directives are used to increase the performance. Number of frames per second processed by this FPGA implementation is calculated as shown in Equation (2.2).

$$\frac{Frequency(MHz) * 1000000}{\left(\frac{Frame\ Size}{Search\ Range\ Size}\right) * Clock\ Cycles} \tag{2.2}$$

Then, a full search IME hardware implementing the FS IME algorithm in HEVC reference software video encoder (HM) version 15 [5] is designed. It supports all 24 PU sizes defined in HEVC standard. It implements 64x64 fixed search range. The proposed hardware is shown in Fig. 2.2.

In HEVC, 593 SADs and 593 MVs should be calculated for all PU sizes. Numbers of SADs and MVs that should be calculated for each PU size are as follows: 4x8 (128 SADs and 128 MVs) , 8x4 (128 SADs and 128 MVs) , 8x8 (64 SADs and 64 MVs) , 4x16 (64 SADs and 64 MVs) , 8x16 (32 SADs and 32 MVs) , 12x16 (20 SADs and 20 MVs) , 16x4 (64 SADs and 64 MVs) , 16x12 (20 SADs and 20 MVs) , 16x16 (16 SADs and 16 MVs), 8x32 (16 SADs and 16 MVs) , 16x32 (8 SADs and 8 MVs) , 24x32 (4 SADs and 4 MVs) , 32x8 (16 SADs and 16 MVs) , 32x16 (8 SADs and 8 MVs) , 32x24 (4 SADs and 4 MVs) , 32x32 (4 SADs and 4 MVs) , 16x64 (4 SADs and 4 MVs), 32x64 (2 SADs and 2 MVs) , 48x64 (1 SAD and 1 MV) , 64x16 (4 SADs and 4 MVs) , 64x32 (2 SADs and 2 MVs) , 64x48 (1 SAD and 1 MV) and 64x64 (1 SAD and 1 MV).

Table 2.1 Full Search Motion Estimation HLS implementation Results For 8x8 PU Size

Optimizations	Slice	LUT	DFF	BRAM	DSP48	Freq. (MHz)	Clock Cycles (8x8 PU)	Fps
NOOPT	290	644	490	2	0	267	20301	0.912 2560*1440
APAR_RES(BRAM)_PIPE_LU	2132	6247	2346	49	0	200	256	54 2560*1440

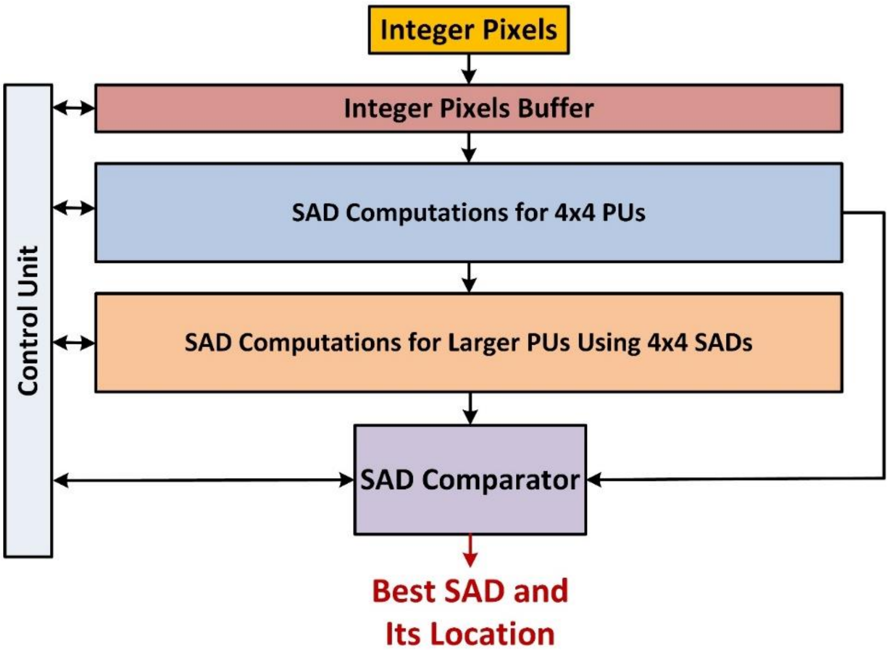


Figure 2.2 HEVC Full Search Motion Estimation HLS Implementation

First, reference and current pixels are stored into integer pixels buffer. 128x128 reference pixels are stored in order to be able to search all search locations in the 64x64 search range. Then, SAD values for 4x4 PUs in the 64x64 CU are calculated. Since there are 16x16 4x4 PUs in the 64x64 CU, a 16x16 array is used to store SAD values of 4x4 PUs. Then, SAD values for the other PU sizes are calculated by adding the SAD values of 4x4 PUs. After that, comparison unit compares the SAD values, determines the 593 minimum SAD values for all PU sizes and their corresponding MVs, and stores them into two different arrays.

APAR is used for the 16x16 array storing SADs for 4x4 PUs. In this way, latency of calculating SAD values of larger PUs is reduced by accessing the SAD values of 4x4 PUs in parallel. Loop unrolling (LU) is used to perform absolute difference calculations in parallel. PIPE is used to increase the performance. Bit-accurate model is used in order to decrease adder bit-width.

Verilog RTL codes generated by Xilinx Vivado HLS tool for this HLS implementation are verified with post place and route simulations. The implementation results are shown in Table 2.2.

Finally, this HLS implementation is parametrized to support 4 different (8x8, 16x16, 32x32 and 64x64) search ranges by only changing the boundaries of nested loops calculating SAD values according to the selected search range.

Verilog RTL codes generated by Xilinx Vivado HLS tool for this HLS implementation are verified with post place and route simulations. The implementation results are shown in Table 2.3.

Table 2.2 HEVC Full Search Motion Estimation HLS Implementation Results

Optimizations	**Slice**	**LUT**	**DFF**	**BRAM**	**DSP48**	**Freq. (MHz)**	**Clock Cycles (64x64 PU)**	**Fps**
NOOPT	6858	17632	18397	6	0	125	1056768	0.23 1920x1080
APAR_RES(BRAM)_PIPE_ LU_ BIT	29875	88286	76271	138	0	86	5705	30 1920x1080

Table 2.3 HEVC Full Search Motion Estimation With Variable Search Range HLS Implementation Results

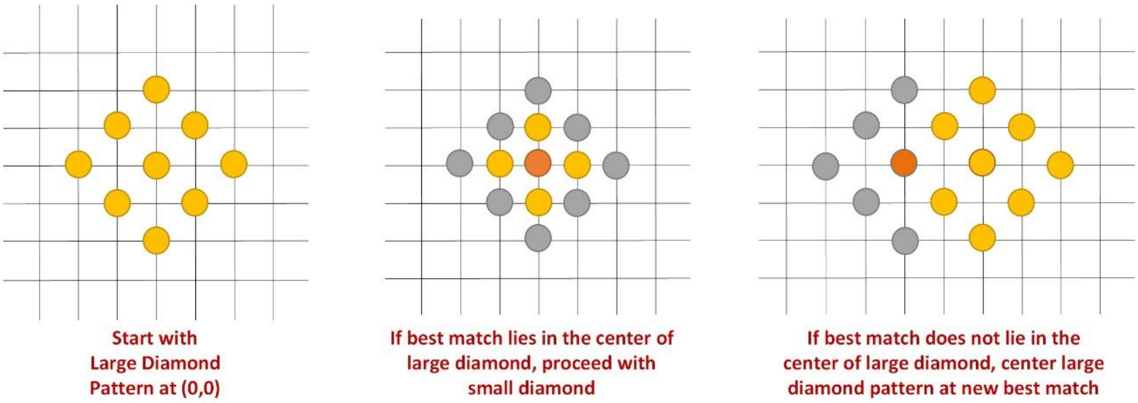
Search Range	**Optimizations**	**Slice**	**LUT**	**DFF**	**BRAM**	**DSP48**	**Freq.**	**Clock Cycles**	**Fps**
8x8	APAR_RES(BRAM)_ PIPE_LU_ BIT	34302	87259	76345	138	0	83	441	372 FHD
16x16	APAR_RES(BRAM)_ PIPE_LU_ BIT							809	202 FHD
32x32	APAR_RES(BRAM)_ PIPE_LU_ BIT							1929	85 FHD
64x64	APAR_RES(BRAM)_ PIPE_LU_ BIT							5705	29 FHD

2.2 FPGA Implementation of Diamond Search Algorithm Using Vivado High-Level Synthesis

Fast search motion estimation algorithms are used to reduce computational complexity of FS algorithm at the expense of slight PSNR loss and bitrate increase.

2.2.1 Diamond Search Algorithm

Diamond search (DS) motion estimation algorithm follows a diamond search pattern. DS algorithm has two steps; large diamond search (LDS) and small diamond search (SDS). LDS calculates SAD values for 9 search locations that form a large diamond shape as shown in Figure 2.3 (a), and determines the search location with minimum SAD. If the search location with minimum SAD is at the center of the diamond shape, SDS is performed. Otherwise, LDS is performed around the search location with minimum SAD as shown in Figure 2.3 (c). SDS calculates SAD values for 4 search locations that form a small diamond shape as shown in Figure 2.3 (b), and determines the search location with minimum SAD and the corresponding motion vector.



(a) (b) (c)

Figure 2.3 Diamond Search Algorithm

2.2.2 FPGA Implementation

The proposed DS HLS implementation for fixed current block size (64x64) and fixed search range size (64x64) is shown in Figure 2.4. First, pixels in the current block in the current frame and necessary pixels in the reference frame are stored into integer pixels buffers. In order to decrease memory area, only 68x68 reference pixels are stored. After the first LDS, if another LDS is performed, only new reference pixels are read and stored into integer pixels buffer. Other reference pixels are shifted.

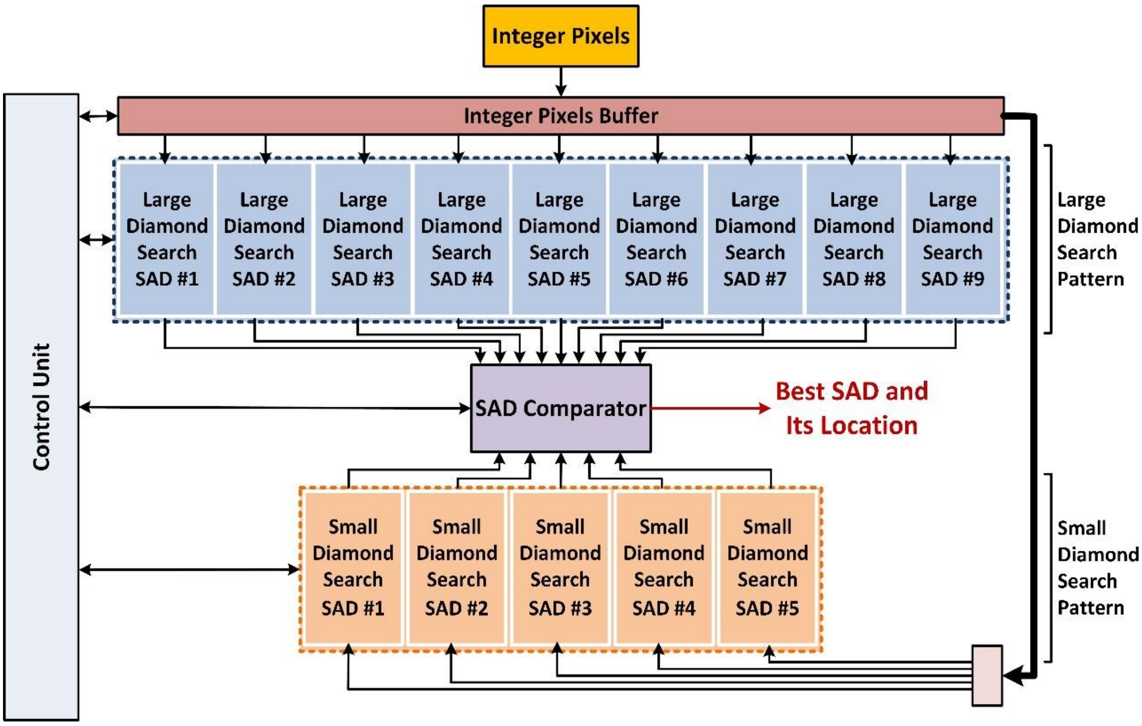


Figure 2.4 Diamond Search HLS Implementation

LDS may never find a search location with minimum SAD that is at the center of the diamond shape. Therefore, a maximum number of LDS allowed should be defined. In the proposed HLS implementation, this maximum number is defined as a parameter which can be between 1 and 10.

In the proposed HLS implementation, 9 SAD values that should be calculated for LDS are calculated in parallel. 64 parallel absolute difference hardware are used for calculating each SAD value. Then, comparison unit determines the search location with minimum SAD.

If the search location with minimum SAD is at the center of the diamond shape, SDS is performed. Otherwise, LDS is performed around the search location with minimum SAD. However, if the maximum number of LDSs allowed are performed, SDS is performed instead of LDS. If another LDS is performed, only new reference pixels are read and stored into integer pixels buffer. Other reference pixels are shifted. We used loop unrolling for shifting.

In the proposed HLS implementation, 4 SAD values that should be calculated for SDS are calculated in parallel. Then, comparison unit determines the search location with minimum SAD. Finally, the minimum SAD values found in LDS and SDS are compared, and the minimum SAD value and the corresponding MV for DS are determined.

Verilog RTL codes generated by Xilinx Vivado HLS tool for this HLS implementation are verified with post place and route simulations. The implementation results are shown in Table 2.4.

APAR, RES, PIPE, LU optimization directives are used in order to increase the performance and decrease the hardware area. Bit-accurate model is also used to decrease hardware area. Number of clock cycles changes with number of steps. These results show that the proposed DS HLS implementation can process 127 full HD frames per second.

Table 2.4 Diamond Search HLS Implementation Results

Optimizations	**Slice**	**LUT**	**DFF**	**BRAM**	**DSP48**	**Freq. (MHz)**	**Clock Cycles (64x64 PU)**		**Fps**
							1 Step	**10 Steps**	
NOOPT	10132	28322	16535	4	0	108	4754	46352	5 1920x1080
APAR_RES(BRAM)_ PIPE_LU_ BIT	12573	37457	20859	67	0	139	334	2152	127 1920x1080

2.3 FPGA Implementation of TZ Search Algorithm Using Vivado High-Level Synthesis

2.3.1 TZ Search Algorithm

TZ search (TZS) is another fast search motion estimation algorithm. It finds better MVs than DS. But, it has higher computational complexity than DS. TZS uses two different search patterns; diamond search pattern and raster search pattern as shown in Figure 2.5 and Figure 2.6, respectively. Raster search is similar to full search, but it searches less number of search locations. It is used as a refinement after the diamond search pattern.

Diamond search pattern starts searching at the (0,0) search location, and it proceeds according to the steps shown in Figure 2.5. It calculates the SAD values and determines the minimum SAD in each step. It has two termination conditions. The first one is reaching the search window boundaries. The second one is not finding minimum SAD in three consecutive steps. For example, if the SAD value of (0,0) search location is smaller than all SAD values calculated in steps 1, 2, and 4, then it is terminated, and the SAD value of (0,0) search location is determined as the minimum SAD.

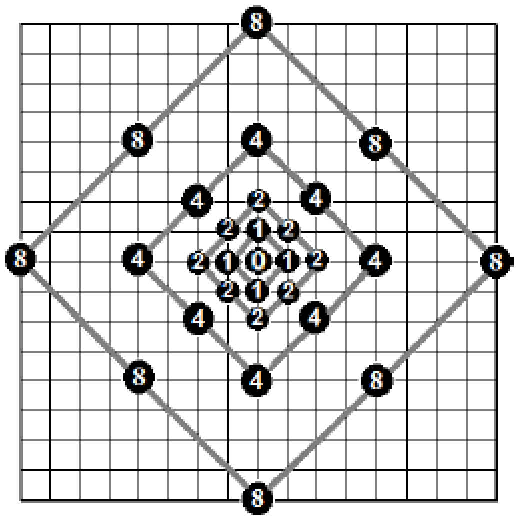


Figure 2.5 TZS Diamond Search Pattern

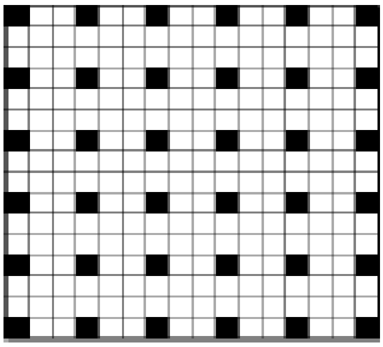


Figure 2.6 TZS Raster Search with Length 3

2.3.2 FPGA implementation

The proposed TZS HLS implementation for fixed current block size (64x64) and fixed search range size (64x64) is shown in Figure 2.7. First, 64x64 current pixels and 128x128 reference pixels are stored into integer pixels buffers. Then, diamond search pattern is performed. Since the search range size is 64x64, maximum number of steps for the diamond search pattern is 6. In each step, SAD values for the search locations are calculated and the search location with minimum SAD is determined.

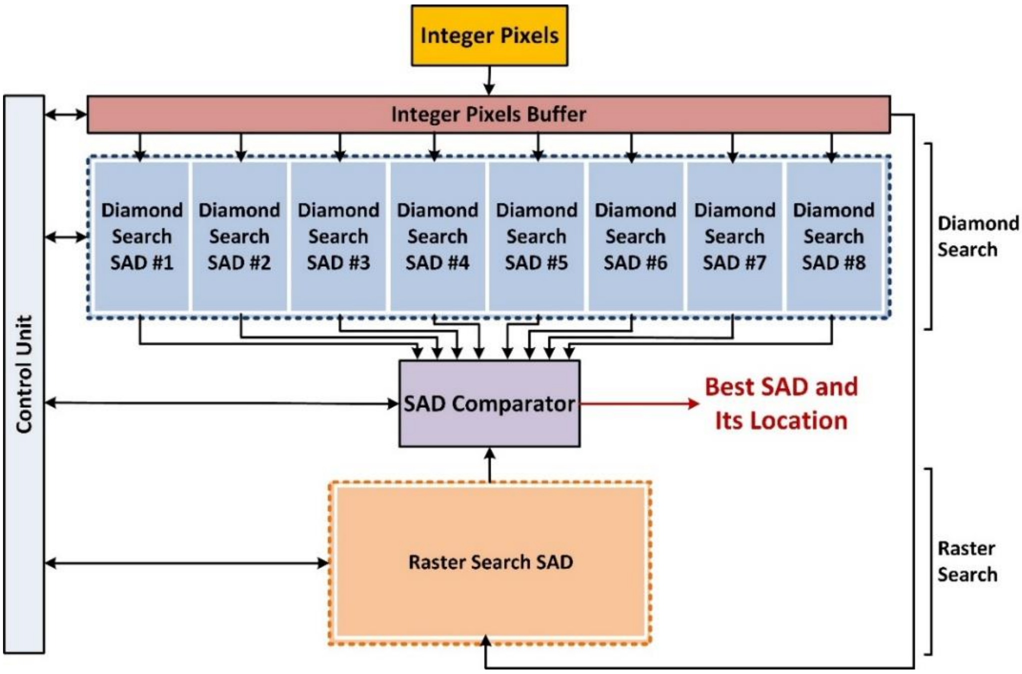


Figure 2.7 TZ Search HLS Implementation

As shown in Figure 2.5, number of search locations for all the steps after step 1 is 8. In order not to repeat the same operations for SAD calculations in steps 2, 3, 4, 5, and 6, control variables are added to HLS code to update memory addresses after each step.

After each step, control unit checks the termination conditions of diamond search pattern. If a termination condition occurs, diamond search pattern is terminated. In that case, if starting condition of raster search pattern occurs, raster search pattern is performed.

As shown in Figure 2.6, raster search pattern is similar to full search. However, it skips some search locations based on raster search length. It searches only the search locations shown as black in the figure. SAD values of these search locations are calculated and the search location with minimum SAD is determined.

Finally, comparison unit compares the minimum SAD found in diamond search pattern and the minimum SAD found in raster search pattern, and determines the minimum SAD and the corresponding MV.

Verilog RTL codes generated by Xilinx Vivado HLS tool for this HLS implementation are verified with post place and route simulations. The implementation results are shown in Table 2.5.

APAR, RES, PIPE, LU optimization directives are used in order to increase the performance and decrease the hardware area. Bit-accurate model is also used to decrease hardware area. These results show that the proposed TZS HLS implementation can process 46 full HD frames per second.

Table 2.5 TZ Search HLS Implementation Results

Optimizations	**Slice**	**LUT**	**DFF**	**BRAM**	**DSP48**	**Freq. (MHz)**	**Clock Cycles (64x64 PU)**	**Fps**
NOOPT	9744	25723	15821	10	0	128	66321	4 1920x1080
APAR_RES(BRAM)_ PIPE_LU_ BIT	39406	114412	15943	128	0	92	3980	46 1920x1080

CHAPTER III

FPGA IMPLEMENTATION OF FRACTIONAL MOTION ESTIMATION USING VIVADO HIGH-LEVEL SYNTHESIS

In order to increase the performance of integer pixel motion estimation, fractional motion estimation (FME), which provides half and quarter pixel accurate motion vector (MV) refinement, is performed. First, fractional interpolation is performed to generate fractional pixels. Then, fractional motion estimation is performed using fractional pixels.

Fractional (half-pixel and quarter-pixel) interpolation is one of the most computationally intensive parts of HEVC video encoder and decoder. On average, one fourth of the HEVC encoder complexity and 50% of the HEVC decoder complexity are caused by fractional interpolation [19]. FME is heavily used in an HEVC encoder. It accounts for up to 49% of total encoding time of HEVC video encoder [20].

HEVC uses FME same as H.264. However, HEVC FME has higher computational complexity than H.264 FME. HEVC standard uses three different 8-tap FIR filters for fractional interpolation and up to 64×64 prediction unit (PU) sizes [21].

3.1 FPGA Implementations of HEVC Fractional Interpolation Using Vivado High-Level Synthesis

Since HEVC fractional interpolation algorithm uses FIR filters, it is suitable for HLS implementation. Therefore, in this thesis, the first FPGA implementation of HEVC fractional interpolation algorithm using Xilinx Vivado HLS tool in the literature is proposed. The proposed HEVC fractional interpolation hardware is implemented on Xilinx FPGAs using Xilinx Vivado HLS tool. The C codes given as input to Xilinx Vivado HLS tool are developed based on the HEVC fractional interpolation software implementation in the HEVC reference software video encoder (HM) version 15 [5].

Three HEVC fractional interpolation HLS implementations are done. In the first one (MM), in the C codes, multiplications with constants are implemented using multiplication operations. In the second one (MAS), multiplications with constants are implemented using addition and shift operations. In the last one (MMCM), addition and shift operations are implemented using Hcub multiplierless constant multiplication algorithm [22].

Some of the optimization options of Xilinx Vivado HLS tool are used in order to increase performances of the FPGA implementations such as pipelining, allocation, resource optimizations, array mapping and array partitioning. Verilog RTL codes generated by Xilinx Vivado HLS tool for the three HEVC fractional interpolation HLS implementations are verified to work in a Xilinx Virtex 6 FPGA.

Using HLS tool significantly reduced the FPGA development time. The implementation results show that the proposed HEVC fractional interpolation FPGA implementation, in the worst case, can process 45 quad full HD (3840x2160) video frames per second with acceptable hardware area.

The HEVC fractional interpolation HLS implementation proposed in this thesis is the first HLS implementation for HEVC fractional interpolation algorithm in the literature. In Section 3.1.2, it is compared with the handwritten HEVC fractional interpolation hardware implementations proposed in the literature [23]-[27].

3.1.1 HEVC Fractional Interpolation Algorithm

In HEVC standard, 3 different 8-tap FIR filters are used for both half-pixel and quarter-pixel interpolations. These 3 FIR filters type A, type B and type C are shown in (3.1), (3.2), and (3.3), respectively. The shift1 value is determined based on bit depth of the pixel.

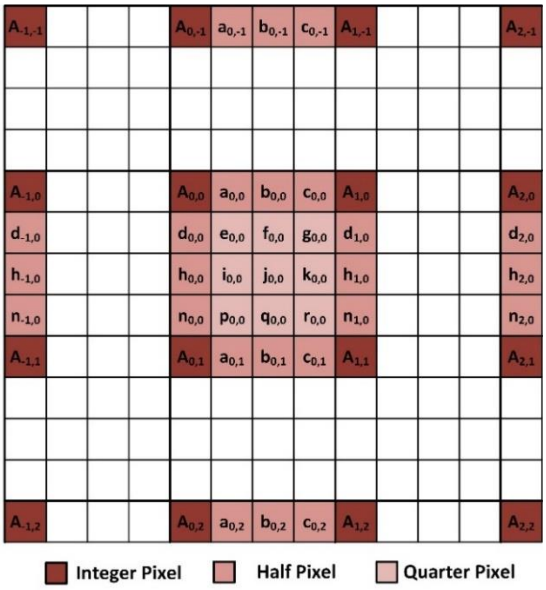


Figure 3.1 Integer, Half and Quarter Pixels

$$a_{0,0} = (-A_{-3,0} + 4 * A_{-2,0} - 10 * A_{-1,0} + 58 * A_{0,0} + 17 * A_{1,,0} - 5 * A_{2,0} + A_{3,0}) \gg shift1 \quad (3.1)$$

$$b_{0,0} = (-A_{-3,0} + 4 * A_{-2,0} - 11 * A_{-1,0} + 40 * A_{0,0} + 40 * A_{1,,0} - 11 * A_{2,0} + 4 * A_{3,0} - A_{4,0}) \gg shift1 \quad (3.2)$$

$$c_{0,0} = (-A_{-2,0} - 5 * A_{-1,0} + 17 * A_{0,0} + 58 * A_{1,0} - 10 * A_{2,,0} + 4 * A_{3,0} - A_{4,0}) \gg shift1 \quad (3.3)$$

Integer pixels ($A_{x,y}$), half pixels ($a_{x,y}$, $b_{x,y}$, $c_{x,y}$, $d_{x,y}$, $h_{x,y}$, $n_{x,y}$) and quarter pixels ($e_{x,y}$, $f_{x,y}$, $g_{x,y}$, $i_{x,y}$, $j_{x,y}$, $k_{x,y}$, $p_{x,y}$, $q_{x,y}$, $r_{x,y}$) in a PU are shown in Figure 3.1. The half pixels a, b, c are interpolated from nearest integer pixels in horizontal direction using type A, type B and type C filters, respectively. The half-pixels d, h, n are interpolated from nearest integer pixels in vertical direction using type A, type B and type C filters, respectively. The quarter pixels e, f, g are interpolated from the nearest a, b, c half pixels respectively in vertical direction using type A filter. The quarter pixels i, j, k are interpolated similarly using type B filter. The quarter pixels p, q, r are interpolated similarly using type C filter.

HEVC fractional interpolation algorithm used in HEVC encoder calculates all the fractional pixels necessary for the fractional motion estimation.

3.1.2 FPGA Implementations

The proposed HLS implementation of HEVC fractional interpolation is shown in Figure 3.2. The proposed HLS implementation is synthesized to Verilog RTL using

Xilinx Vivado HLS tool. The C codes given as input to Xilinx Vivado HLS tool are developed based on the HEVC fractional interpolation software implementation in the HEVC reference software video encoder (HM) version 15 [5].

In the proposed HLS implementation, half pixels and quarter pixels for an 8x8 PU are calculated using 15x15 integer pixels. Half pixels and quarter pixels for larger PU sizes can be calculated by calculating the half pixels and quarter pixels for each 8x8 part of a PU separately. In the C codes, 15 integer pixels are taken as input in each clock cycle. 8 a, 8 b and 8 c half-pixels are interpolated in parallel in each clock cycle. 15x8 a, 15x8 b, and 15x8 c half pixels are interpolated in 15 clock cycles, and they are stored into registers for quarter pixel interpolation. In the same 15 clock cycles, 15x8 integer pixels are also stored into registers for interpolating d, h, n half pixels. Then, 8x8 d, 8x8 h, 8x8 n half pixels are interpolated using 15x8 integer pixels. Finally, all quarter pixels (e, f, g, i, j, k, p, q, r) are interpolated using 15x8 a, 15x8 b, and 15x8 c half pixels.

Three HEVC fractional interpolation HLS implementations are done. In the first one (MM), in the C codes, multiplications with constants are implemented using multiplication operations. In the second one (MAS), multiplications with constants are implemented using addition and shift operations. In the last one (MMCM), addition and shift operations are implemented using Hcub multiplierless constant multiplication algorithm [22].

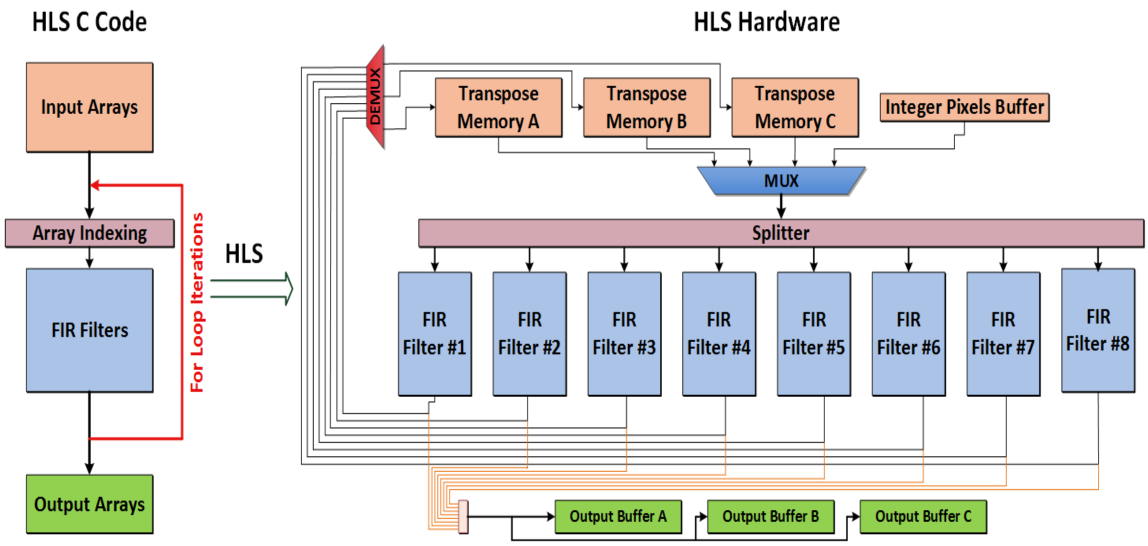


Figure 3.2 HEVC Fractional Interpolation HLS Implementation

Verilog RTL codes generated by Xilinx Vivado HLS tool for these three HLS implementations are verified with RTL simulations. RTL simulation results matched the results of HEVC fractional interpolation software implementation in the HEVC reference software video encoder (HM) version 15 [5]. The Verilog RTL codes are synthesized and mapped to a Xilinx XC6VLX550T FF1760 FPGA with speed grade 2 using Xilinx ISE 14.7. The FPGA implementations are verified with post place and route simulations.

We used several optimizations offered by Xilinx Vivado HLS tool to increase the performance and decrease the area of the proposed HLS implementations [28]. We tried to use loop unrolling directive. However, loop unrolling directive did not work correctly for the proposed HLS implementations. In [29], it is mentioned that loop unrolling may cause memory access problems in HLS designs, and current generation of HLS tools may ignore these problems. As shown in Table 3.1, the performance of the HLS implementation, which implements multiplications with constants using multiplication operations, without loop unrolling is very low. Therefore, we performed manual loop unrolling in the proposed HLS implementations to increase their performances.

In the proposed HLS implementations, ALC is used for subtraction, addition, multiplication, and shifting operations. PIPE directive is used in the proposed HLS implementations. In the proposed HLS implementations, RES directive is used to store input integer pixels into BRAMS. In the proposed HLS implementations, AMAP directive is used to control how data is stored in BRAMS so that the number of BRAMS used in the hardware is reduced as much as possible. In the proposed HLS implementations, APAR directive is used to partition the arrays that store a, b, and c half pixels to increase quarter pixel interpolation performance. In the proposed HLS implementations, bit accurate (BIT) model is used to decrease adder bit widths and therefore hardware area.

The FPGA implementation results for the first HLS implementation (MM) are given in Table 3.2. In this HLS implementation, in the C codes, multiplications with constants are implemented using multiplication operations. These multiplication operations are mapped to DSP48 blocks in RTL synthesis. This decreased the number of LUTs and DFFs used in the hardware. Allocation (ALC), pipeline (PIPE), resource (RES) and array map (AMAP) directives are used in this HLS implementation. In the table, M shows the number of multipliers used in the ALC directive.

The FPGA implementation results for the second HLS implementation (MAS) are given in Table 3.3. In this HLS implementation, multiplications with constants are implemented using addition and shift operations. This HLS implementation does not use any DSP48 blocks, but it uses more LUTs and DFFs than MM. It also has higher performance than MM. Pipeline (PIPE), resource (RES) and array map (AMAP) directives are used in this HLS implementation.

Table 3.1 HLS Implementation without Manual Loop Unrolling with Multipliers Results

Optimizations	**Slice**	**LUT**	**DFF**	**BRAM**	**DSP48**	**Freq. (MHz)**	**Clock Cycles (8x8 PU)**	**Fps**
NOOPT	885	2565	1411	1	15	250	1921	1 3840x2160

Table 3.2 HLS Implementation with Multipliers Results

Optimizations	**Slice**	**LUT**	**DFF**	**BRAM**	**DSP48**	**Freq. (MHz)**	**Clock Cycles (8x8 PU)**	**Fps**
NOOPT	4623	14110	7526	0	113	200	156	10 3840x2160
ALC(M128)	4769	14133	6226	0	135	168	148	9 3840x2160
PIPE	4938	14086	8736	0	113	201	56	28 3840x2160
RES(BRAM)	4723	13883	7395	4	113	201	156	10 3840x2160
ALC(M128)_RES(BRAM)_PIPE	5197	14366	8000	4	147	167	56	23 3840x2160
ALC(M128)_AMAP(4)_ RES(BRAM)_PIPE	4299	12401	7964	2	147	167	56	23 3840x2160
ALC(M20)_AMAP(4)_ RES(BRAM)_PIPE	4299	13100	8037	2	59	168	56	23 3840x2160

Table 3.3 HLS Implementation with Adders and Shifters Results

Optimizations	**Slice**	**LUT**	**DFF**	**BRAM**	**DSP48**	**Freq. (MHz)**	**Clock Cycles (8x8 PU)**	**Fps**
NOOPT	4809	15629	9095	0	0	202	133	12 3840x2160
AMAP(4)_RES(BRAM)_PIPE	4891	15716	9436	2	0	200	55	28 3840x2160

The FPGA implementation results for the last HLS implementation (MMCM) are given in Table 3.4. In this HLS implementation, addition and shift operations are implemented using Hcub multiplierless constant multiplication algorithm [22]. The type A and type B FIR filter equations for 8 a half pixels and 8 b half pixels are shown in Figure 3.3. As shown in Figure 3.3, common sub-expressions are calculated in different equations and same integer pixel is multiplied with different constant coefficients in different equations. Therefore, in this HLS implementation, common sub-expressions in different FIR filter equations are calculated once, and the result is used in all the equations. This HLS implementation also uses Hcub MCM algorithm in order to reduce number and size of the adders, and to minimize the adder tree depth [22]. Hcub algorithm tries to minimize number of adders, their bit size and adder tree depth in a multiplier block, which multiplies a single input with multiple constants. This HLS implementation has the best performance with acceptable hardware area. Allocation (ALC), pipeline (PIPE), array partition (APAR) directives and bit-accurate (BIT) model are used in this HLS implementation. In the table, A and S show the number of adders and subtractors used in the ALC directive, respectively.

Table 3.4 HLS Implementation with MCM Results

Optimizations	Slice	LUT	DFF	BRAM	DSP48	Freq. (MHz)	Clock Cycles (8x8 PU)	Fps
NOOPT	4850	15632	6673	2	0	201	195	8 3840x2160
ALC(A500_S500)_ APAR_PIPE	5288	14619	10118	0	0	168	29	45 3840x2160
ALC(A500_S500)_ APAR_PIPE_BIT	4426	14225	9984	0	0	168	29	45 3840x2160

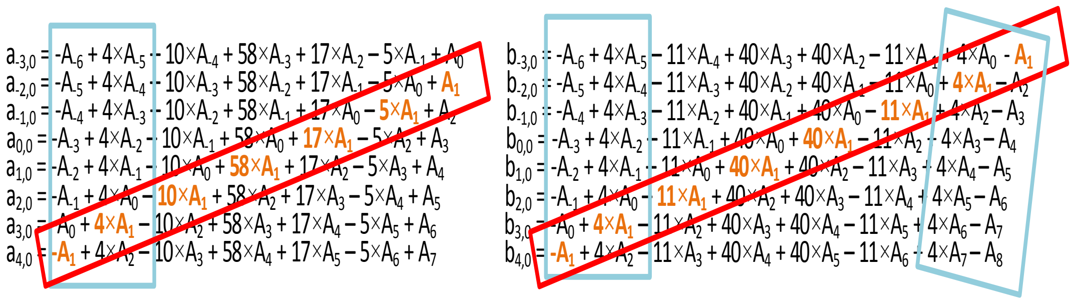


Figure 3.3 Type A and Type B FIR Filters

The best HEVC fractional interpolation HLS implementation proposed in this thesis (MMCM with ALC(A500_S500)_APAR_PIPE_BIT) is compared with the handwritten HEVC fractional interpolation hardware implementations proposed in the literature [23]-[27]. The comparison results are shown in Table 3.5.

The proposed MMCM HLS implementation is similar to the handwritten HEVC fractional interpolation hardware implementation proposed in [23]. In [23], common sub-expressions in different FIR filter equations are calculated once, and the result is used in all the equations. Also, addition and shift operations are implemented using Hcub multiplierless constant multiplication (MCM) algorithm.

In [23], the handwritten Verilog RTL codes are synthesized and mapped to a Xilinx XC6VLX130T FF1156 FPGA with speed grade 3. In this thesis, the handwritten Verilog RTL codes proposed in [23] are synthesized and mapped to a Xilinx XC6VLX550T FF1760 FPGA with speed grade 2 for fair comparison with the proposed MMCM HLS implementation. The proposed MMCM HLS implementation has higher performance than the handwritten HEVC fractional interpolation hardware implementation proposed in [23] at the expense of larger area.

Table 3.5 HEVC Fractional Interpolation Hardware Comparison

	[23]		[24]	[25]	[26]	[27]	Proposed (MMCM)
Technology	Xilinx Virtex 6	90 nm	90 nm	150 nm	90 nm	130 nm	Xilinx Virtex 6
Gate/Slice Count	1597	28.5 K	32.5 K	30.2 K	224 K	126.8 K	4426
Max Speed (MHz)	200	200	171	312	333	208	168
Frames per Second	30 3840x2160	30 3840x2160	60 3840x2160	30 3840x2160	30 1920x1080	86 3840x2160	45 3840x2160
Design	ME + MC	ME + MC	Only MC	ME + MC	ME + MC	ME + MC	ME + MC

Since the handwritten HEVC fractional interpolation hardware implementation proposed in [24] is designed only for motion compensation (MC), it has higher performance and lower area than the proposed MMCM HLS implementation.

The handwritten HEVC fractional interpolation hardware implementation proposed in [25] has lower performance and therefore lower area than the proposed MMCM HLS implementation. In addition, it requires higher clock frequency to achieve real time performance. The handwritten HEVC fractional interpolation hardware implementation proposed in [18] has both lower performance and larger area than the proposed MMCM HLS implementation. The handwritten HEVC fractional interpolation hardware implementation proposed in [27] has higher performance than the proposed MMCM HLS implementation at the expense of larger area.

3.2 FPGA Implementations of HEVC Fractional Motion Estimation Using High-Level Synthesis

3.2.1 HEVC Fractional Motion Estimation Algorithm

After integer pixel motion estimation is performed for a PU, FME is performed for the same PU to obtain fractional-pixel accurate motion vector (MV). In HEVC reference software video encoder (HM) [5], FME is performed in two stages. As shown in Figure 3.4, 8 sub-pixel search locations around the best integer pixel search location are searched in the first stage. 8 sub-pixel search locations around the best sub-pixel search location of the first stage are searched in the second stage.

HEVC FME first interpolates the necessary sub-pixels for sub-pixel search locations using three different 8-tap FIR filters. In Figure 3.4, half-pixels a, b, c and d, h, n are interpolated using the nearest integer pixels in horizontal and vertical directions, respectively. Quarter-pixels e, i, p and f, j, q and g, k, r are interpolated using the nearest a, b and c half-pixels, respectively. HEVC FME then calculates the SAD values, as shown in (3.4) for each sub-pixel search location, and determines the best sub-pixel search location with the minimum SAD value.

$$SAD = \sum_{i=0}^{m}\sum_{j=0}^{n}\left|R_{ij} - C_{ij}\right| \qquad (3.4)$$

$$m = 0\ to\ (PU_{width} - 1), n = 0\ to\ (PU_{height} - 1)$$

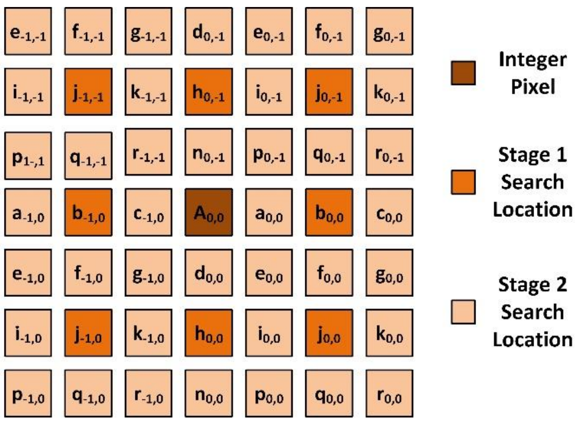


Figure 3.4 Sub-pixel Search Locations

HEVC performs fractional motion estimation for 24 different PU sizes (4x8, 8x4, 8x8, 4x16, 16x4, 8x16, 16x8, 12x16, 16x12, 16x16, 8x32, 32x8, 16x32, 32x16, 24x32, 32x24, 32x32, 16x64, 64x16, 32x64, 64x32, 48x64, 64x48 and 64x64). There are 593 different PUs for these 24 different PU sizes, and 593 different SAD values should be calculated for them.

3.2.2 FPGA Implementations

The proposed HEVC fractional motion estimation HLS implementation for 8x8 PU size is shown in Figure 3.5. Three HEVC fractional motion estimation HLS implementations are done. In the first one (MM), in the C codes, multiplications with constants are implemented using multiplication operations. In the second one (MAS), multiplications with constants are implemented using addition and shift operations. In the last one (MMCM), addition and shift operations are implemented using Hcub multiplierless constant multiplication algorithm [22].

Fractional interpolation is implemented as described in Section 3.1. However, in the proposed FME HLS implementation, 16 integer pixels are taken as input instead of 15 integer pixels for all the necessary SAD calculations. There are 3 9x8 memories for d, h and n half pixels, 3 16x9 memories for a, b, and c half pixels, and 9 9x9 memories for quarter pixels.

In the first stage, 8 fractional pixel search locations around the best integer pixel search location are searched. 8 parallel SAD calculation hardware are used to calculate SAD values of these 8 search locations in parallel. Appropriate current, half and quarter pixels are read from current, half and quarter pixel memories, respectively, for the SAD calculations. 8 parallel absolute difference (AD) hardware calculate AD values of an 8x8 PU in 8 clock cycles. Then, SAD value of this 8x8 PU is calculated using these ADs. After the SAD values are calculated, comparison hardware determines the search location with minimum SAD value.

In the second stage, 8 fractional pixel search locations around the best fractional pixel search location of the first stage are searched. The same hardware used in the first stage is used for SAD calculation. After the SAD values are calculated, comparison hardware determines the search location with minimum SAD value.

Finally, the minimum SAD value found in the FME is compared with the SAD value of the best integer pixel search location, and the search location with minimum SAD value is determined.

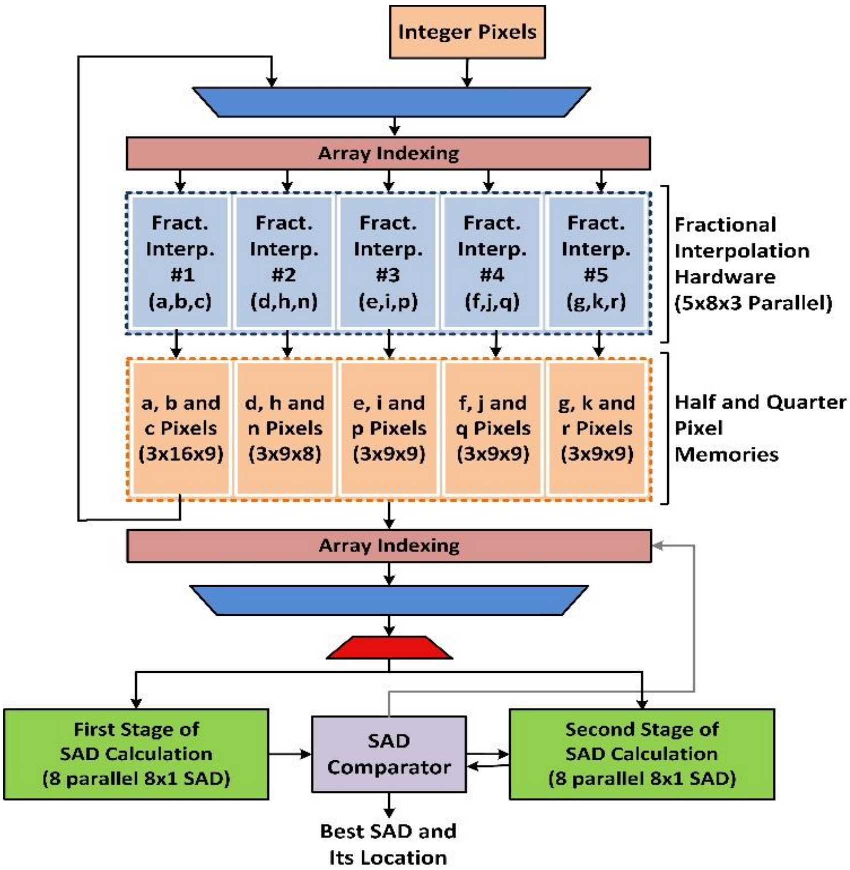


Figure 3.5 HEVC fractional motion estimation HLS implementation

The proposed MM and MAS FME HLS implementations for 8x8 PU size are extended to support almost all (22 out of 24) PU sizes (8x8, 16x8, 8x16, 16x16, 32x8, 8x32, 32x16, 16x32, 32x24, 24x32, 32x32, 16x64, 64x16, 64x32, 32x64, 64x48, and 64x64). For PU sizes larger than 8x8, PUs can be divided into 8x8 pixel blocks. Therefore, the proposed FME HLS implementation for 8x8 PU size is parameterized to support larger PU sizes. All loops are parameterized to satisfy the number of iterations necessary for specific PU size. Because of the asymmetric PU sizes, all loops are designed as nested loops. Also, memory sizes are arranged to support different PU sizes.

ALC directive is used for subtraction, addition, multiplication, and shifting operations to decrease hardware area. Pipeline (PIPE) directive is used between functions, for loop iterations, and computations. PIPE decreases latency and increases frequency of proposed FME HLS implementations. Resource (RES) directive is only used for memories. Some arrays are forced to map to BRAM instead of registers using

RES directive to decrease hardware area. AMAP directive is used to store half pixels in the memory efficiently. APAR directive is used to use registers instead of BRAMs. This increases hardware area. Since APAR provides parallel data accesses, it increases the performance. In addition, bit accurate model is used to decrease adder bit widths and therefore hardware area.

FPGA implementation results for the HLS implementations of HEVC fractional motion estimation algorithm are shown in Table 3.6. As shown in Table 3.6, the proposed FME HLS implementation results are divided into two groups; (i) for only 8x8 PUs, and (ii) for all PU sizes. There are three different FME HLS implementations (MM, MAS and MMCM) for only 8x8 PUs. Allocation (ALC), pipeline (PIPE), resource (RES) and array partition (APAR) directives are used in these HLS implementations. There are two different FME HLS implementations (MM, MAS) for all PU sizes. The best results for these HLS implementations are shown in Table 3.6.

As shown in Table 3.6, allocation and pipeline directives directly affect the performance of the proposed HLS implementations. Allocation limits number of resources used. Therefore, ALC directive decreases the number of DSP48 units for multiplication operations, and LUTs for the addition/subtraction operations. Pipeline directive decreases the number of clock cycles and increases the performance of the proposed HLS implementations.

The effect of the allocation directive for MM HLS implementation is analyzed in Table 3.7. Number of DSP48 blocks, clock cycles and frequency are observed by changing the number of available multipliers. Increasing the number of multipliers after a threshold value do not affect the results because of the data dependencies. Decreasing the number of multipliers increases the complexity of control because of the complex resource sharing mechanism. This reduces the frequency.

Table 3.6 HEVC Fractional Motion Estimation HLS Implementation Results

PU	Design	Optimizations	Slice	LUT	DFF	BRAM	DSP48	Freq.	Clock Cycles	Fps
8x8	MM	NOOPT	8743	24722	10309	9	202	72	1304	1.7 FHD
		ALC(M500)	10088	29707	21741	9	38	125	1501	2.6 FHD
		ALC(M500)_APAR_ PIPE_RES(BRAM)_BIT	12767	36761	26875	44	146	125	201	19.2 FHD
	MAS	NOOPT	11800	33805	24458	9	0	143	1501	2.9 FHD
		ALC(A20_S20)_PIPE	11077	32424	27486	10	0	143	1219	3.6 FHD
		ALC(A20_S20)_APAR _PIPE_BIT	17155	52449	42093	41	0	125	241	16 FHD
	MMCM	NOOPT	10226	29196	22889	6	0	167	713	7.2 FHD
		PIPE	9735	28458	21922	6	0	143	453	9.7 FHD
		ALC(A100_S100)_ APAR_PIPE_BIT	16366	52521	41535	0	0	167	140	36.8 FHD
All Sizes	MM	ALC(M20)_APAR_ PIPE_RES(BRAM)_BIT	13027	41397	21864	69	57	111	9024	24.3 FHD
	MAS	ALC(A20_S20)_APAR _PIPE_BIT	13632	41085	22545	69	10	143	9051	31.2 FHD

Table 3.7 Allocation Analysis for MM HLS Implementations

		M1	M10	M50	M100	M200	M500
Fract. Interp.	DSP48	32	58	104	135	135	---
	C. Cyc.	1133	196	156	148	148	---
	Freq.	165	167	170	170	170	---
FME (8x8)	DSP48	0	2	38	38	38	38
	C. Cyc.	1901	1501	1501	1501	1501	1501
	Freq.	125	130	130	129	129	125

The proposed HEVC FME HLS implementation is compared with the handwritten HEVC FME hardware implementations in the literature [30] - [33]. As shown in Table 3.8, [30] has smaller area and higher performance than the proposed hardware. However, it interpolates SADs instead of pixels. Therefore, it decreases PSNR and increases bit rate. In [31], FME hardware searches all possible 48 sub-pixel search locations. However, it only supports square shaped PU sizes. In [32], FME hardware supports all PU sizes but 8x4, 4x8 and 8x8. It uses bilinear filter for quarter-pixel interpolation. Also, it searches 12 sub-pixel search locations. In [33], FME hardware supports all PU sizes but it uses a scalable search pattern.

Table 3.8 HEVC Fractional Motion Estimation Hardware Comparison

	Tech.	**Gate/Slice Count**	**Freq. (MHz)**	**PU Sizes**	**Fps**
[30]	Xilinx Virtex 6	1814	142	All	19 QFHD
[31]	65 nm	249.1 K	396	Square Shaped	6 QFHD
[32]	65 nm	1183 K	188	All but 8x8, 8x4, 4x8	15 QFHD
[33]	Xilinx Virtex 6	130 K	200	All	32 QFHD
Prop.	Xilinx Virtex 6	13632	143	All but 4x8, 8x4	8 QFHD

CHAPTER IV

CONCLUSIONS AND FUTURE WORK

In this thesis, we proposed the first FPGA implementation of HEVC full search motion estimation using Vivado HLS. Then, we proposed the first FPGA implementations of two fast search (diamond search and TZ search) algorithms using Vivado HLS. Finally, we proposed the first FPGA implementations of HEVC fractional interpolation and motion estimation using Vivado HLS. All FPGA implementations are verified to work correctly at real-time using post place and route simulations.

As future work, FPGA implementations of fast search motion estimation algorithms can be extended for variable block sizes and variable search ranges. FPGA implementations of other fast search algorithms such as hexagon search can be done using Vivado HLS.

BIBLIOGRAPHY

[1] B. Bross, W.J. Han, J.R. Ohm, G.J. Sullivan, Y.K. Wang, and T. Wiegand, “High Efficiency Video Coding (HEVC) Text Specification Draft 10”, *JCTVC-L1003,* Feb. 2013.

[2] G.J.Sullivan, J.R. Ohm, W.J. Han, T. Wiegand, “ Overview of the High Efficiency Video Coding (HEVC) Standard, ” *IEEE Trans. on Circuits and Systems for Video Technology,* vol.22, no.12, pp.1649-1668, Dec. 2012.

[3] F. Bossen, B. Bross, K. Suhring and D. Flynn, “HEVC Complexity and Implementation Analysis”, *IEEE Trans. on Circuits and Systems for Video Technology,* vol.22, no.12, pp.1685-1696, Dec. 2012.

[4] J. Vanne, M. Viitanen, T.D. Hämäläinen and A. Hallapuro, “Comparative Rate-Distortion-Complexity Analysis of HEVC and AVC Video Codecs”, *IEEE Trans. on Circuits and Systems for Video Technology*, vol.22, no.12, pp.1885-1898, Dec. 2012.

[5] K. McCann, B. Bross, W.J. Han, I.K. Kim, K. Sugimoto, G. J. Sullivan, “High Efficiency Video Coding (HEVC) Test Model (HM) 15 Encoder Description”, *JCTVC-Q1002*, June 2014.

[6] E. Kalali, I. Hamzaoglu, “FPGA Implementations of HEVC Inverse DCT Using High-Level Synthesis,” *Conf. on Design and Architectures for Signal and Image Processing (DASIP)*, pp. 1-6, Sept. 2015.

[7] P. Sjovall, J. Virtanen, J. Vanne, T. D. Hamalainen, "High-Level Synthesis Design Flow for HEVC Intra Encoder on Soc-FPGA," *Euromicro Conf. on Digital System Design (DSD)*, pp. 49-56, Aug. 2015.

[8] M. Abid, K. Jerbi, M. Raulet, O. Deforges, M. Abid, "System Level Synthesis of Dataflow Programs: HEVC Decoder Case Study," *Electronic System Level Synthesis Conference*, pp. 1-6, May 2013.

[9] T. Damak, I.Werda, N. Masmoudi, S. Bilavarn, "Fast prototyping H.264 deblocking filter using ESL tools," *8th International Multi-Conf. on System, Signals and Devices*, pp. 1-4, March 2011.

[10] S. Kim, H. Kim, T. Chung, J-G. Kim, "Design of H.264 video encoder with C to RTL design tool," *Int. SoC Design Conference*, pp. 171-174, Nov. 2012.

[11] K. Fleming, C-C. Lin, N. D. Arvind, G. Raghavan, J. Hicks, "H.264 decoder: A case study in multiple design points," *ACM/IEEE Int. Conf. on Formal Methods and Models for Co-Design*, pp. 165-174, June 2008.

[12] P. P. Carballo, O Espino, R. Neris, P. H. Fernandez, T. M. Szydzik, A. Nunez, "Scalable video coding deblocking filter FPGA and ASIC implementation using high-level synthesis methodology," *Euromicro Conf. on Digital System Design (DSD)*, pp. 415-422, Sept. 2013.

[13] S. S. Bhattacharyya, J. Eker, J. W. Janneck, C. Lucarz, M. Mattavelli, M. Raulet, "Overview of the MPEG reconfigurable video," *Springer Journal of Signal Processing Systems*, vol. 63, no. 2, pp. 251-263, May 2011.

[14] J. F. Nezan, N. Siret, M. Wipliez, F. Palumbo, L. Raffo, "Multi-purpose systems: A novel dataflow-based generation and mapping strategy," *IEEE Int. Symposium on Circuits and Systems (ISCAS)*, pp. 3073-3076, May 2012.

[15] H. Ye, L. Lacassagne, D. Etiemble, L. Cabaret, J. Falcou, A. Romero, O. Florent, "Impact of high level transforms on high level synthesis for motion detection algorithm," *Conf. on Design and Architectures for Signal and Image Processing (DASIP)*, pp. 1-8, Oct. 2012.

[16] G. Schewior, C. Zahl, H. Blume, S. Wonneberger, J. Effertz, "HLS-based FPGA implementation of a predictive block-based motion estimation algorithm - A field report," *Conf. on Design and Architectures for Signal and Image Processing (DASIP)*, pp. 1-8, Oct. 2014.

[17] O. A. Abella, G. Ndu, N. Sonmez, M. Ghasempour, A. Armejach, J. Navaridas, W. Song, J. Mawer, A. Cristal, M. Lujan, “An empirical evaluation of high-level synthesis languages and tools for database acceleration,” *Int. Conf. on Field Programmable Logic and Applications (FPL),* Sept. 2014.

[18] M. Schmid, N. Apelt, F. Hanning, J. Teich, “An image processing library for C-based high-level synthesis,” *Int. Conf. on Field Programmable Logic and Applications (FPL)*, Sept. 2014.

[19] J. Vanne, M. Viitanen, T.D. Hämäläinen, A. Hallapuro, “Comparative rate-distortion-complexity analysis of HEVC and AVC video codecs”, *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 22, no. 12, pp.1885-1898, Dec. 2012.

[20] J. Vanne, M. Viitanen, T.D. Hämäläinen, A. Hallapuro, “Comparative Rate-Distortion-Complexity Analysis of HEVC and AVC Video Codecs”, *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 22, no. 12, pp.1885-1898, Dec. 2012.

[21] E. Kalali, Y. Adibelli, I. Hamzaoglu, “A Reconfigurable HEVC Sub-Pixel Interpolation Hardware”, *IEEE Int. Conference on Consumer Electronics - Berlin,* Sept. 2013.

[22] Y. Voronenko, M. Püschel, "Multiplierless Constant Multiple Multiplication", *ACM Trans. on Algorithms*, vol. 3, no. 2, May 2007.

[23] E. Kalali, I. Hamzaoglu, “A low energy HEVC sub-pixel interpolation hardware,” *IEEE Int. Conference on Image Processing (ICIP),* pp. 1218-1222, Oct. 2014.

[24] Z. Guo, D. Zhou, S. Goto, “An Optimized MC Interpolation Architecture for HEVC”, *IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, March 2012.

[25] C. M. Diniz, M. Shafique, S. Bampi, J. Henkel, “High-Throughput Interpolation Hardware Architecture with Coarse-Grained Reconfigurable Datapaths for HEVC”, *IEEE Int. Conference on Image Processing,* Sept. 2013.

[26] G. Pastuszak, M. Trochimiuk, “Architecture Design and Efficiency Evaluation for the High-Throughput Interpolation in the HEVC Encoder”, *16th Euromicro Conference on Digital System Design*, Sept. 2013.

[27] D. Kang, Y. Kang, Y. Hong, "VLSI Implementation of Fractional Motion Estimation Interpolation for High Efficiency Video Coding", *Electronic Letters*, vol. 51, no. 5, pp. 1163-1165, July 2015.

[28] UG902, "Vivado Design Suite User Guide: High-Level Synthesis," May 2014.

[29] P. Coussy, A. Morawiec, "High-Level Synthesis from Algorithm to Digital Circuit", *Springer*, 2008.

[30] A. C. Mert, E. Kalali, I. Hamzaoglu, "Low Complexity HEVC Sub-Pixel Motion Estimation Technique and Its Hardware Implementation", *IEEE Conf. on Consumer Electronics – Berlin (ICCE-Berlin)*, Sept. 2016.

[31] V. Afonso, H. Maich, L. Audibert, B. Zatt, M. Porto, L. Agostini, "Memory-Aware and High-Throughput Hardware Design for the HEVC Fractional Motion Estimation", *Symposium on Integrated Circuits and System Design*, 2013.

[32] G. He, D. Zhou, Y. Li, Z. Chen, T. Zhang, S. Goto, "High-Throughput Power-Efficient VLSI Architecture of Fractional Motion Estimation for Ultra-HD HEVC Video Encoding", *IEEE Trans. VLSI Systems*, vol.23, no.12, pp.3138-3142, March 2015.

[33] D. Ding, X. Ye, S. Wang, "1/2 and 1/4 Pixel Paralleled FME with A Scalable Search Pattern for HEVC Ultra-HD Encoding", *IEEE Int. Conf. on Communication Technology*, pp.278-281, Oct. 2015.