# A Scalable Architecture for H.264/AVC Variable Block Size Motion Estimation on FPGAs

THEEPAN MOORTHY, PHOEBE PING CHEN, ANDY YE
Department of Electrical and Computer Engineering
Ryerson University
350 Victoria Street, Toronto, Ontario M5B 2K3
CANADA
tmoorthy@ece.ubc.ca, pepe_chen@hotmail.com, aye@ee.ryerson.ca

*Abstract:* - In this paper, we investigate the use of Field-Programmable Gate Arrays (FPGAs) in the design of a highly scalable Variable Block Size Motion Estimation architecture for the H.264/AVC video encoding standard. The scalability of the architecture allows one to incorporate the system into low cost single FPGA solutions for low-resolution video encoding applications as well as into high performance multi-FPGA solutions targeting high-resolution applications. To overcome the performance gap between FPGAs and Application Specific Integrated Circuits, our design minimizes the increase in memory bandwidth as the design scales. The core computing unit of the architecture is implemented on FPGAs and its performance is reported. It is shown that the computing unit is able to achieve 58 frames per second (fps) performance for 640x480 resolution VGA video while incurring only 4.5% LUT and 6.3% DFF utilization on a Xilinx XC5VLX330 FPGA. With 8 computing units at 38% LUT and 55% DFF utilization, the architecture is able to achieve 50 fps performance for encoding full 1920x1088 progressive HDTV video.

*Key-Words:* - Variable Block Size Motion Estimation, H.264/AVC, Field-Programmable Gate Arrays

## 1 Introduction

The Variable Block Size Motion Estimation (VBSME) algorithm is an essential part of the H.264/AVC video-encoding standard. Relative to Fixed Block Size Motion Estimation algorithms, VBSME provides much higher compression ratios and picture quality. VBSME algorithms, however, are much more computationally expensive. In particular, the H.264/AVC standard calls for up to 41 motion vectors for each macroblock and its corresponding subblocks. Due to this high computing demand, many hardware architectures have been proposed to accelerate the computation of VBSME motion vectors for H.264/AVC [1]–[8]. Most of the architectures, however, have been implemented in Application Specific Integrated Circuit (ASIC) technology. Except for limited commercial implementations [9]–[11], little information exists on how these algorithms would perform on reconfigurable technologies such as Field-Programmable Gate Arrays (FPGAs). In particular, the FPGA implementation presented in [12] specifically targets portable multimedia devices with CIF-level resolution and cannot be easily scaled. The FPGA implementation presented in [13], on the other hand, only reaches VGA-level resolution and 27 fps performance. It too cannot be scaled. In this work, we propose a scalable hardware VBSME architecture based on the Propagate Partial SAD architecture [8] and measure its performance on FPGAs as the design scales.

The use of FPGAs encourages design reuse and can greatly enhance the upgradability of digital systems. The programmability of FPGAs is particularly useful for highly flexible encoding systems that can accommodate a multitude of existing standards as well as the emergence of new standards. In particular, our design can be incorporated into single FPGA solutions targeting low cost low-resolution applications as well as into multiple FPGA designs for high performance high-resolution applications.

Comparing to other programmable options such as Digital Signal Processors (DSPs) including Texas Instrument DaVinchi [14] and Analog Device Blackfin [15], FPGAs can offer considerably higher performances for implementing highly parallel data streaming applications such as motion estimation. In particular, while real time implementations of motion estimation algorithms on DSPs are limited to fast-search algorithms, FPGAs can efficiently implement full-search motion estimation algorithms in real time.

Unlike full-search algorithms, fast-search algorithms [16]–[18] trade quality for performance by searching through only a subset of an entire search space while following a set of pre-defined search patterns. During the search process, the algorithms constantly check the quality of the motion estimation results and terminate the motion estimation process as soon as the quality of the results reaches an acceptable level.

Full-search motion estimation algorithms [1]–[8], on the other hand, always find the best motion estimation solution by exhaustively searching through the entire search space. Comparing to fast-search algorithms, full-search algorithms offer better video quality and higher compression ratios since exhaustive search guarantees optimal results that are independent of the actual situation of motion in videos.

The benefits of full-search algorithms do come at a much higher computing cost. The full-search algorithms, however, are particularly well suited for hardware acceleration due to their highly parallel datapath, simple control logic and regular memory access patterns.

Our proposed architecture is based on one of the three widely used full-search VBSME architectures — the Propagate Partial SAD [1] [8], SAD Tree [7], and the Parallel Sub-Tree [6]. The Propagate Partial SAD architecture was selected due to its unique blend of efficiency and scalability. While the SAD Tree architecture has the highest performance amongst the three [7], it requires the support of a complex array of shifting registers that must have the capability of shifting in both horizontal and vertical directions. This array, while efficient to implement in ASICs, consumes a large amount of FPGA resources. The Parallel Sub-Tree architecture, on the other hand, is the most compact design amongst the three. The architecture, however, inherently does not scale well for high performance applications [6].

As proposed in [1] and [8], the Propagate Partial SAD architecture processes a single group of 16 reference blocks at a time. Our design enhances the original design by allowing it to be scaled to process several groups of 16 reference blocks simultaneously. These groups share a large amount of their reference pixels. This sharing minimizes the increase in memory bandwidth as the design scales and makes high performance FPGA-based design feasible. The performance of the scalable design is further enhanced by a new row adder tree design, which reorganizes the addition process in order to achieve more efficient pipelining. The reorganization eliminates retiming registers and further increases circuit throughput and performance.

An early version of this paper appeared in [19]. This work augments that work with an extended investigation on the effect of pipelining on the pixel processing unit – a critical component of the VBSME architecture. It is found that a two-stage-pipelined design of row adder trees can significantly increase performance while the three-stage-pipelined design only results in limited performance benefits.

The remainder of this paper is organized as follows: Section 2 introduces the general Motion Estimation algorithm and the Propagate Partial SAD architecture, Section 3 presents the scalable VBSME architecture, Section 4 describes the detailed design of the pipelined pixel processing unit, Section 5 describes the corresponding memory architecture, Section 6 evaluates the system performance, and Section 7 concludes.

## 2 Hardware Motion Estimation

Video encoding algorithms typically process one 16x16 block of pixels (called a macroblock) at a time. The frame that contains the macroblocks currently being processed is referred to as the current frame. During the encoding process, the goal of Motion Estimation (ME) is to find the best match for a macroblock from a set of reference pixels (where the set is called a search window, and the frame that contains the search window is called a reference frame). To this end all ME algorithms accomplish this goal through three distinct stages of computation. First the macroblock is mapped onto a 16x16 block of pixels (called a reference block) in the search window, and the absolute difference values between the macroblock pixels and the corresponding reference block pixels are calculated. Second, the Sum of the Absolute Differences (SAD) is calculated for the reference block by summing the absolute difference values over the entire block. This process repeats until a SAD value is calculated for each of the reference blocks in the search window. Thirdly, the minimum of all the SAD values in the search window is computed and the corresponding reference block is used by the encoder to calculate the best-match Motion Vector (MV) for the macroblock currently being processed.

Equation 1 and 2 show the arithmetic for calculating the SAD value of a reference block such that pixel (x, y) in the macroblock is mapped to pixel (rx + x, ry + y) in the search window.

$$SAD(rx, ry) = \sum_{x=0}^{W-1}\sum_{y=0}^{H-1}\left|R(rx+x, ry+y) - C(x,y)\right| \quad (1)$$

$$rx \in [0, RW-16]\ ,\ ry \in [0, RH-16] \quad (2)$$

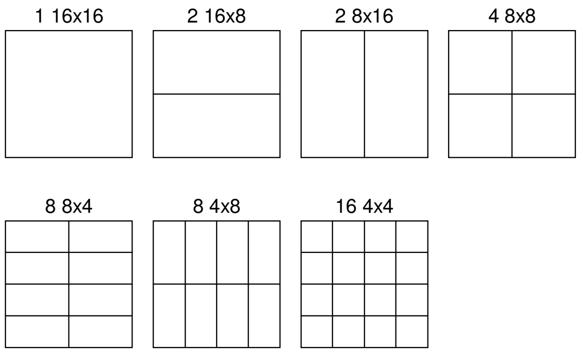

**Fig. 1** Macroblock and Subblocks in VBSME

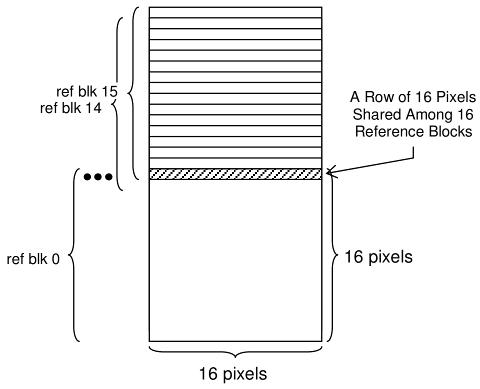

**Fig. 2** A Group of 16 Reference Blocks

Here, W and H represent the width and height of the macroblock. RW and RH represent the width and height of the search window. C(x, y) represents the value of pixel (x, y) in the macroblock while R(rx + x, ry + y) represents the value of pixel (rx + x, ry + y) in the search window. Note that this paper uses a horizontal search range of [-24, +23] pixels and a vertical search range of [-16, +16] pixels for each macroblock. These search range values translate to an RW value of 63 and RH value of 48.

Instead of just calculating one SAD per macroblock/reference-block pair, VBSME algorithms subdivide a 16x16 macroblock into a set of subblocks. Correspondingly, the reference block is also divided into subblocks and SAD values are then calculated for each of the subblocks in addition to the macroblock. In particular, as shown in Figure 1, the H.264/AVC standard subdivides a macroblock into 40 subblocks of size 16x8, 8x16, 8x8, 8x4, 4x8, and 4x4. Consequently, for a macroblock, 41 SAD values are needed per reference block.

The Propagate Partial SAD architecture speeds up VBSME algorithms by simultaneously calculating SAD values for 16 reference blocks at a time. In particular, the architecture takes advantage of the fact that, in a search window, every vertical group of 16 reference blocks share a common row of 16-pixels (as shown in Figure 2). In the Propagate Partial SAD architecture, this common row is then used to simultaneously calculate 16 absolute difference values for each of the 16 reference blocks. A specialized pipeline structure is then used to accumulate these absolute difference values to produce the 41 SAD values per reference block at every clock cycle.

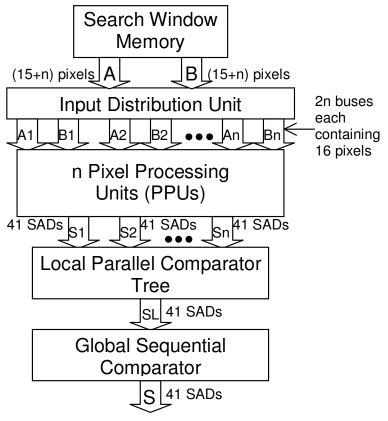

**Fig. 3** The Scalable VBSME Architecture

## 3 System Architecture

The overall structure of the scalable VBSME architecture is shown in Figure 3. It consists of a bank of memory that stores the search window, an input distribution unit, n Pixel Processing Units (PPUs), and two sets of comparators. As in [8], the memory storing the search window is divided into two partitions. Each partition contains an output of 15+n pixels. These outputs are expanded into 2n buses by the input distribution unit, where each bus contains 16 pixels. The 2n buses are then fed into n PPUs, which have been initialized with a macroblock's pixel values.

The PPUs are used to produce n x 41 SAD values at each clock cycle. These n x 41 SAD values are then used to compute the minimum SAD values of the search window in two steps. First, the n x 41 SADs are fed into the local parallel comparator tree. This tree computes 41 minimum SAD values from its n x 41 inputs. The local minimum SAD values are forwarded to the global sequential comparator, which determines the 41 minimum SAD values for the entire search window. Note that the global comparator is of a conventional less-than comparator design [8] and the scaling of the VBSME architecture does not affect its complexity.

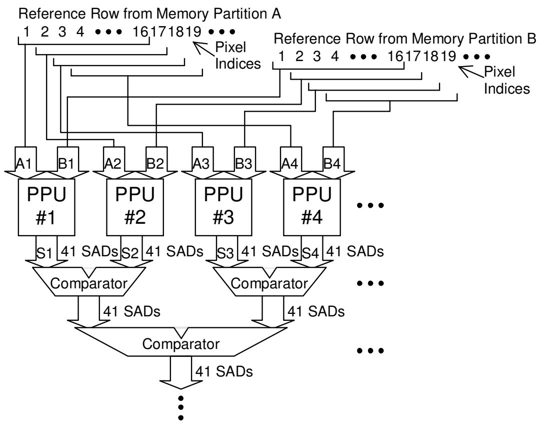

**Fig. 4** Input Distribution Unit, PPUs and Local Comparators

The detailed design of the input distribution unit, the PPUs, and the local parallel comparator tree is shown in Figure 4. As shown, the core of the scalable VBSME architecture is the PPUs, which are based on the Propagate Partial SAD architecture. As discussed in Section 2, each PPU produces 41 SAD values (corresponding to an entire set of SADs for a single reference block) at every clock cycle. The number of PPUs utilized in the scalable architecture, therefore, corresponds directly to the number of reference blocks that can be processed in a clock cycle and the overall performance of the system. However, as the number of PPUs increases, the output bandwidth required for the search window memory increases as well. In particular, in order to keep a PPU fully utilized during motion estimation, one would require two rows of 16-pixels to be forwarded from the search window memory to the PPU at every clock cycle (one row from each of the search window memory partitions) [8]. Typically, a byte is used to encode a pixel, therefore one needs to transport 32 bytes from the search window to a PPU in every clock cycle.

A naive approach would be to simply increase the output of the search window memory by 32 bytes for every additional PPU. However, this can quickly exhaust the internal memory bandwidth of an FPGA (if the search window is stored on the same chip as the PPUs) or the IO pin limit of even the largest modern FPGAs (if the search window is stored off chip). For example, the Xilinx XC5VLX330 is the largest device that Xilinx currently offers. It contains 1200 available IO pins. Assume that the search window is stored off chip. Implementing a single PPU on the XC5VLX330 would require 256 input pins. Implementing four PPU copies would require 1024 pins (over 85% of the available IOs on the XC5VLX330) – leaving an insufficient number of IOs for output and control signals.

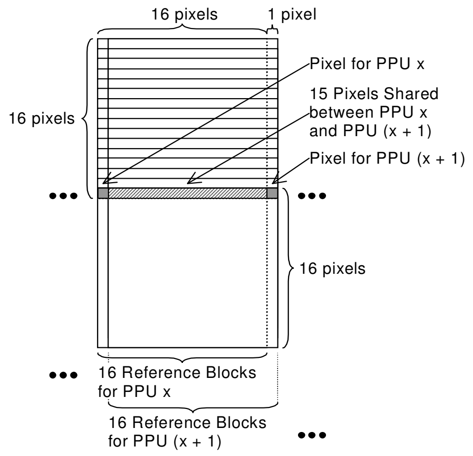
16 pixels
1 pixel
Pixel for PPU x
15 Pixels Shared between PPU x and PPU (x + 1)
Pixel for PPU (x + 1)
16 pixels
16 pixels
16 Reference Blocks for PPU x
16 Reference Blocks for PPU (x + 1)

**Fig. 5** Sharing of Pixels among PPUs

More importantly, the above approach does not take into account the large number of pixels that are shared among the reference blocks. For example, Figure 5 shows 32 reference blocks in a search window. These blocks are divided into two groups where each group contains 16 reference blocks. Within a group, the reference blocks are organized as in Figure 2, where all blocks are contained within a single 16-pixel wide column and one block is offset from the next by a single row of pixels.

As in Figure 2, 16 reference blocks from the same group share a row of 16 common pixels. Furthermore, since one group is offset from another by a single column of pixels, all 32 blocks in Figure 5 share 15 common pixels.

To increase performance, these two groups can be simultaneously processed by two PPUs (shown as PPU x and PPU (x+1) in the figure). Since 15 pixels are shared between the groups, one would require 17 pixels (instead of 32) to be read from the search window (per single bus) at a time. In particular, if pixels (a, y), (a+1, y), …, (a+15, y) of the search window are being processed by PPU x, pixels (a+1, y), (a+2, y), …, (a+16, y) should be simultaneously processed by PPU (x+1).

In general, to fully utilize n PPUs, one would require (15 + n) pixels to be read from each partition of the search window memory for every clock cycle. These signals should then be distributed using the topology shown in Figure 4.

At its output, each PPU shown in Figure 4 produces 41 SAD values at every clock cycle. These SAD values amount to 573 bits of data. To keep the output width constant as the number of PPUs increases, the local parallel comparator tree can be implemented on the same FPGA as the PPUs. Note that the number of comparator tree stages is equal to $\lceil \log_2(n) \rceil$ where n is the number of PPUs that the architecture contains. We observe that by registering the values produced at each stage of the comparator tree one can ensure that the comparator tree does not become the critical path of the system. Consequently, the overall system performance does not degrade significantly when an increasing number of PPUs are used. (Note that, as shown in Table 2 there is a variation in clock frequency as the number of PPUs is increased from 1 to 12 units. This variation is due to increases in routing delay as the size of the design increases and is not due to increases in the logic delay of the comparator tree.)

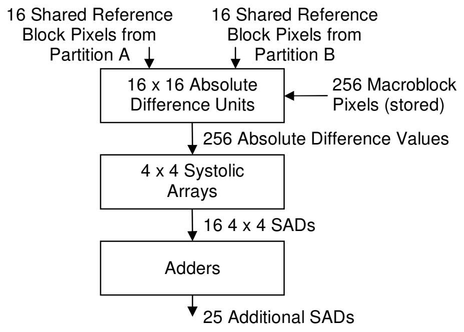
16 Shared Reference Block Pixels from Partition A
16 Shared Reference Block Pixels from Partition B
16 x 16 Absolute Difference Units
256 Macroblock Pixels (stored)
256 Absolute Difference Values
4 x 4 Systolic Arrays
16 4 x 4 SADs
Adders
25 Additional SADs

**Fig. 6** PPU Structure

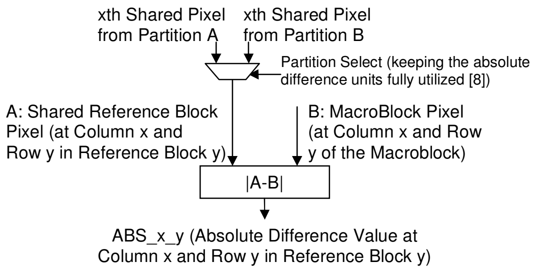
xth Shared Pixel from Partition A
xth Shared Pixel from Partition B
Partition Select (keeping the absolute difference units fully utilized [8])
A: Shared Reference Block Pixel (at Column x and Row y in Reference Block y)
B: MacroBlock Pixel (at Column x and Row y of the Macroblock)
|A-B|
ABS_x_y (Absolute Difference Value at Column x and Row y in Reference Block y)

**Fig. 7** Absolute Difference Unit at Column *x* and Row *y*

## 4 Pipelining the Pixel Processing Unit

To simultaneously compute SAD values for the 16 reference blocks as shown in Figure 2, the work in [8] employs a three stage PPU design as shown in Figure 6. The first stage consists of 256 absolute difference units arranged in a 16 x 16 array. Each unit as shown in Figure 7 computes the absolute difference value between one of the 16 shared reference pixels and its corresponding macroblock pixel. In particular, column *x* row *y* of the array computes the absolute difference value for the pixel located at column *x* and row *y* in the *y*th reference block as shown in Figure 2. (Note that within each reference block the rows are labeled from top to bottom and the columns are labeled from left to right.) The absolute difference values are accumulated into 16 4 x 4 SADs through 16 systolic arrays in stage 2. These 4 x 4 SADs are then used to calculate the remaining 25 SADs in stage 3.

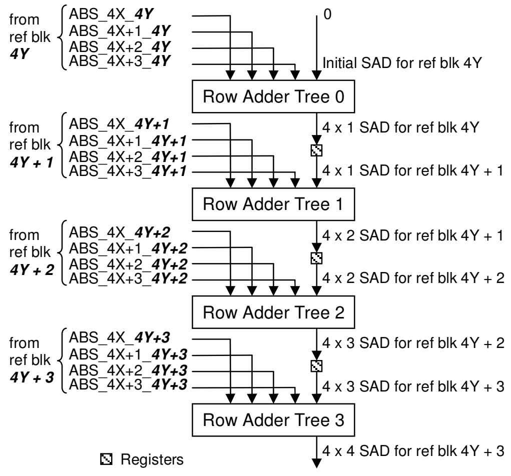

**Fig. 8** A Systolic Array at Column *X* and Row *Y*

As shown in Figure 6, at stage 2 the systolic arrays are arranged in four columns and four rows. As shown in Figure 8, each array contains four row adder trees and each tree generates a SAD output by adding four absolute difference value inputs to a SAD input. In particular, for the array located at column *X* and row *Y*, the *k*th row adder tree outputs a SAD that is accumulated from columns 4*X* to 4*X* + 3 and rows 4*Y* to 4*Y* + *k* in reference block 4*Y* + *k*. The SAD is then registered and fed to the next row adder tree in the array, which processes reference block 4*Y* + *k* + 1. Note that in the figure the highlighted y coordinate of each ABS value is equal to the block number, also highlighted, of its reference block and this equality is required to maintain the 16-pixel-per-clock-cycle memory bandwidth shown in Figure 2.

Figure 9 shows the four implementations of row adder trees that are investigated in this work. Figure 9(a) shows the original design as described in [8] where the five inputs are summed through three Carry Save Addition (CSA) adders and a ripple carry adder. While this design is highly efficient for VLSI implementations, its performance can be improved on FPGAs through pipelining.

Figure 9(b) shows a directly pipelined version of the original design. Note that the additional pipeline stage creates one clock cycle delay between the input SAD and the output SAD. Due to the pipeline delay, the systolic array employing the design must include 24 retiming registers as shown in Figure 10 in order to produce one 4 x 4 SAD for every clock cycle while still maintaining the equality between the y coordinates of the ABS values and their reference block numbers (and consequently the 16-pixel-per-clock-cycle memory bandwidth shown in Figure 2) – without the retiming registers, the 16 pixel bandwidth would limit the systolic array to produce only one 4 x 4 SAD for every two clock cycles or one would need to quadruple the output bandwidth of the search window memory. (Note that with the pipelined design the PPU simultaneously processes 20, instead of 16, reference blocks as shown in Figure 11.)

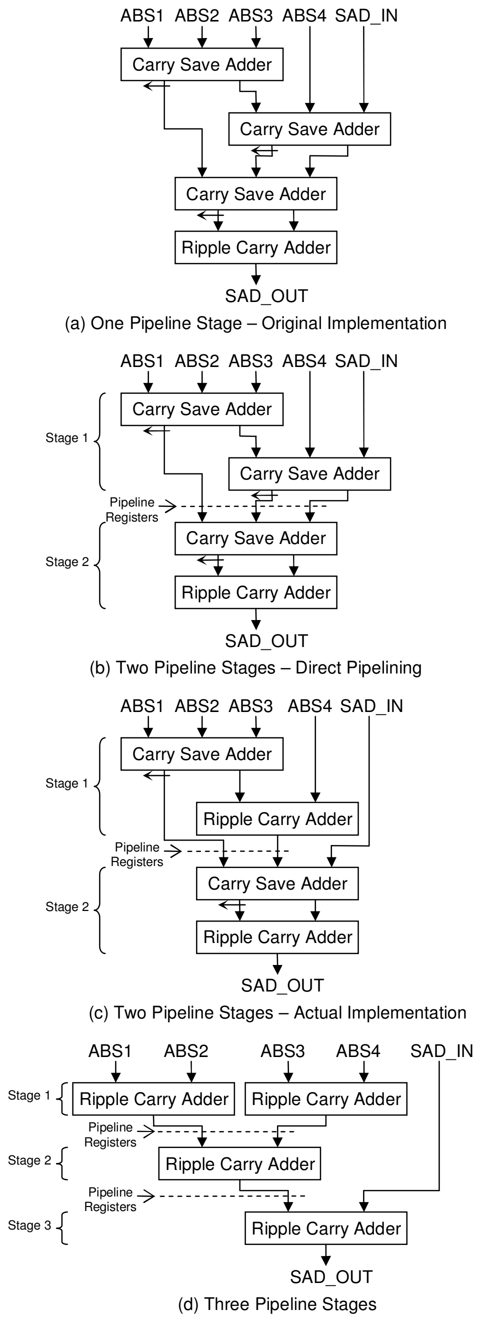
ABS1 ABS2 ABS3 ABS4 SAD_IN
Carry Save Adder
Ripple Carry Adder
SAD_OUT
(a) One Pipeline Stage – Original Implementation
Stage 1
Stage 2
Pipeline Registers
(b) Two Pipeline Stages – Direct Pipelining
(c) Two Pipeline Stages – Actual Implementation
Stage 3
(d) Three Pipeline Stages

**Fig. 9** Pipelining the Row Adder Tree

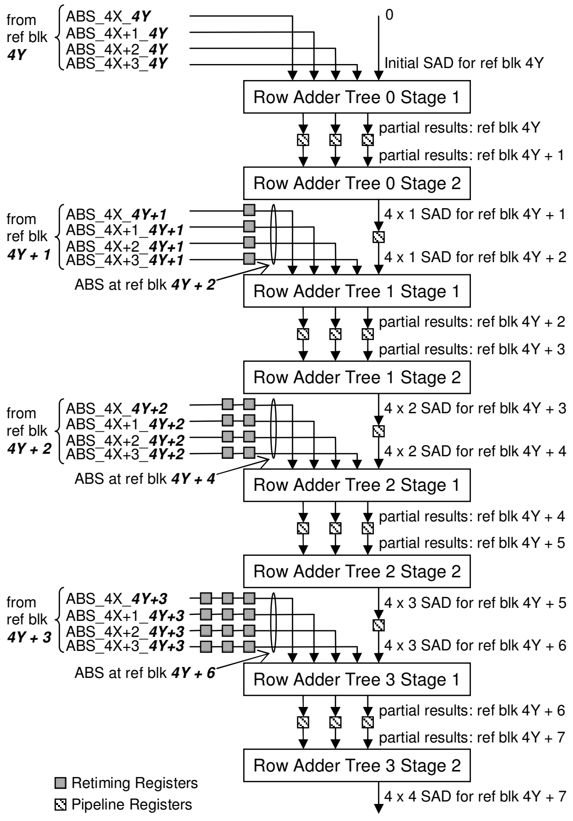
from ref blk 4Y
ABS_4X_4Y
ABS_4X+1_4Y
ABS_4X+2_4Y
ABS_4X+3_4Y
0
Initial SAD for ref blk 4Y
Row Adder Tree 0 Stage 1
partial results: ref blk 4Y
partial results: ref blk 4Y + 1
Row Adder Tree 0 Stage 2
from ref blk 4Y + 1
4 x 1 SAD for ref blk 4Y + 1
4 x 1 SAD for ref blk 4Y + 2
ABS at ref blk 4Y + 2
Row Adder Tree 1 Stage 1
partial results: ref blk 4Y + 2
partial results: ref blk 4Y + 3
Row Adder Tree 1 Stage 2
from ref blk 4Y + 2
4 x 2 SAD for ref blk 4Y + 3
4 x 2 SAD for ref blk 4Y + 4
ABS at ref blk 4Y + 4
Row Adder Tree 2 Stage 1
partial results: ref blk 4Y + 4
partial results: ref blk 4Y + 5
Row Adder Tree 2 Stage 2
from ref blk 4Y + 3
4 x 3 SAD for ref blk 4Y + 5
4 x 3 SAD for ref blk 4Y + 6
ABS at ref blk 4Y + 6
Row Adder Tree 3 Stage 1
partial results: ref blk 4Y + 6
partial results: ref blk 4Y + 7
Row Adder Tree 3 Stage 2
Retiming Registers
Pipeline Registers
4 x 4 SAD for ref blk 4Y + 7

**Fig. 10** Retiming Registeres Required for the Pipelined Row Adder Tree Design As Shown in Figure 9(b)

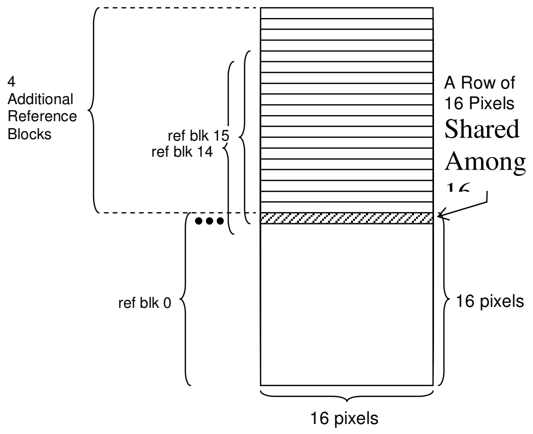
4 Additional Reference Blocks
ref blk 15
ref blk 14
ref blk 0
A Row of 16 Pixels Shared Among 16
16 pixels
16 pixels

**Fig. 11** A Group of 20 Reference Blocks

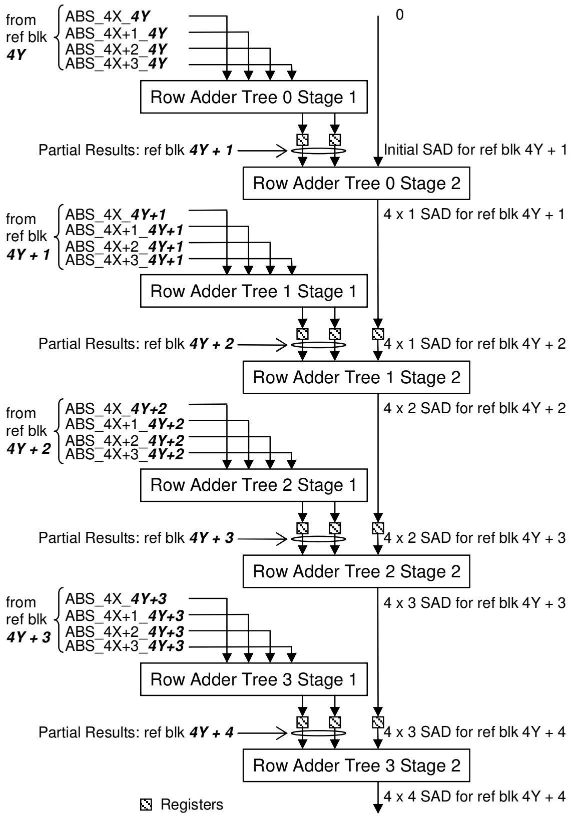
from ref blk 4Y
ABS_4X_4Y
ABS_4X+1_4Y
ABS_4X+2_4Y
ABS_4X+3_4Y
0
Row Adder Tree 0 Stage 1
Partial Results: ref blk 4Y + 1
Initial SAD for ref blk 4Y + 1
Row Adder Tree 0 Stage 2
4 x 1 SAD for ref blk 4Y + 1
from ref blk 4Y + 1
Row Adder Tree 1 Stage 1
Partial Results: ref blk 4Y + 2
4 x 1 SAD for ref blk 4Y + 2
Row Adder Tree 1 Stage 2
4 x 2 SAD for ref blk 4Y + 2
from ref blk 4Y + 2
Row Adder Tree 2 Stage 1
Partial Results: ref blk 4Y + 3
4 x 2 SAD for ref blk 4Y + 3
Row Adder Tree 2 Stage 2
4 x 3 SAD for ref blk 4Y + 3
from ref blk 4Y + 3
Row Adder Tree 3 Stage 1
Partial Results: ref blk 4Y + 4
4 x 3 SAD for ref blk 4Y + 4
Row Adder Tree 3 Stage 2
4 x 4 SAD for ref blk 4Y + 4
Registers

**Fig. 12** Systolic Array Design Based on Figure 9(c)

To eliminate the retiming registers, the designs shown in Figure 9(c) and Figure 9(d) reorganize the addition of the row adder tree inputs. In Figure 9(c), a CSA adder is substituted by a ripple carry adder. The substitution allows the SAD input to bypass the first pipeline stage and be directly connected to the second pipeline stage. As shown in Figure 12, a systolic array employing the design can produce one 4 x 4 SAD for every clock cycle while requiring no retiming registers, and the number of simultaneously processed reference blocks is reduced from 20 to 17.

Similarly, a three stage pipelined design is shown in Figure 9(d). Here all three CSA adders are substituted by ripple carry adders. The SAD input is directly fed into the last pipeline stage. Again, without the retiming registers, the three stage pipelined design allows each systolic array to produce one 4 x 4 SAD every clock cycle while the PPU simultaneously processes 18 reference blocks.

## 5 On-Chip Memory Organization

When targeting an FPGA with a moderate number of user-available IO pins, the scalable system shown in Figure 4 may still become an IO bottleneck. Consider the case of a system scaled to 16 PPUs. With each pixel encoded using a single byte, the input pixels will amount to 62 bytes (16 bytes from each partition of the search window memory for the initial PPU followed by 1 extra byte from each partition for the 15 additional PPUs). The output will consist of 41 SAD values (independent of the number of PPUs used) and would consume 72 bytes of IO. When control signals are considered, the total IO requirement of the circuit shown in Figure 4 becomes 135 bytes or 1080 IO pins (bits).

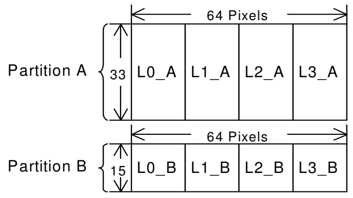

**Fig. 13** Search Window Partition (16 PPUs)

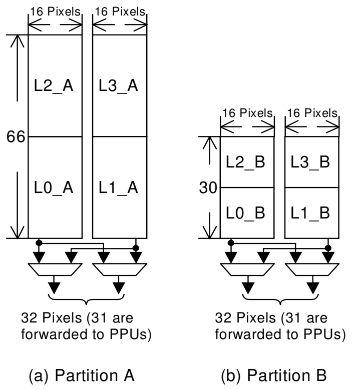

**Fig. 14** Physical Layout of a Buffer (16 PPUs)

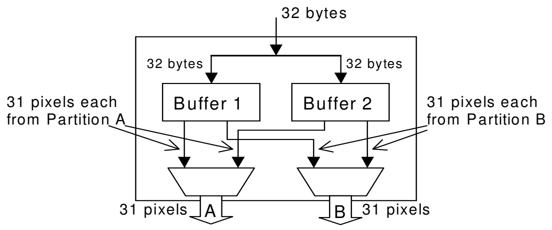
32 bytes
32 bytes
32 bytes
Buffer 1
Buffer 2
31 pixels each from Partition A
31 pixels each from Partition B
31 pixels
A
B
31 pixels

**Fig. 15** Overall Structure of the Search Window Memory (16 PPUs)

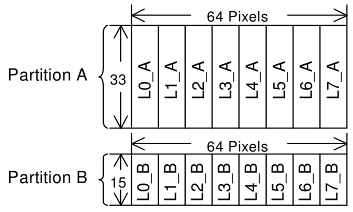

**Fig. 16** Search Window Partition (8 PPUs)

On devices where such a number of IOs is not available, the on-chip RAM blocks available on most modern FPGAs can be utilized to buffer the search window. For a search window of 63 x 48 pixels, this translates to 3024 bytes of data per search window. Using double buffering, while the current search window is being processed, another 3024 bytes of on-chip memory can be utilized to receive the next search window, hence reducing the number of required IOs.

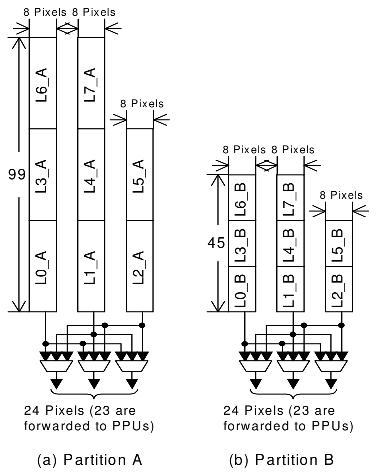

**Fig. 17** Physical Layout of a Buffer (8 PPUs)

**Table 1: Lower Bound on Input Bandwidth**

| # of PPUs | Search Window | | | | Macroblock |
|---|---|---|---|---|---|
| | Without On-Chip Memory | | With On-Chip Memory | | |
| | Bytes | Input Pins (bits) | Bytes | Input Pins (bits) | Input Pins (bits) |
| 1 | 32 | 256 | 1.94 | 15.6 | 0.33 |
| 2 | 34 | 272 | 3.88 | 31.1 | 0.65 |
| 4 | 38 | 304 | 7.76 | 62.1 | 1.30 |
| 8 | 46 | 368 | 15.6 | 124.2 | 2.59 |
| 12 | 54 | 432 | 23.3 | 186.2 | 3.88 |
| 16 | 62 | 496 | 31.1 | 248.3 | 5.18 |

The memory system design for a 16 PPU system is shown in Figures 13 to 15. As shown, each buffer is used to store a search window. To simplify the design process, the search window is extended by one column of pixels to 64 x 48. As in [8], the search window is divided into two partitions, where Partition A contains row 0 to 32 and Partition B contains row 33 to 47.

Each partition is then logically subdivided into four sub-partitions. In particular, Partition A is divided into L0_A, L1_A, L2_A, and L3_A. L0_A is then physically grouped with L2_A and L1_A is physically grouped with L3_A. As shown in Figure 14(a), this results in two banks of memory each containing 66 x 16 pixels. This memory organization allows the PPUs to access all pixels contained in columns 0 to 30 (from sub-partitions L0_A and L1_A), 16 to 46 (from sub-partitions L1_A and L2_A), and 32 to 62 (from sub-partitions L2_A and L3_A) of Partition A in 99 cycles [6]. Partition B is similarly organized, as shown in Figure 14(b), and can be processed in 45 cycles. These 45 cycles overlap the 99 cycles of Partition A [8]. This results in an overall process time of 99 cycles per buffer.

To ensure full utilization of the PPUs, each buffer must be filled within the 99-cycle processing time. This imposes a lower limit on the input bandwidth of the buffers. In particular, an input-width of 32 bytes allows the 64 x 48 byte search window to be updated in 96 cycles to fully utilize a 16 PPU system. Figure 15 shows the overall structure of the double-buffered memory organization for a 16-PPU system.

Similarly, the 16 x 16 pixel macroblock is also double-buffered. Each buffer is updated after every four search windows are processed (each window being from a unique reference frame). Consequently, the buffers need to be updated within every 396 cycles for the 16-PPU system. A one-byte-wide input can be used to update each macroblock buffer in 256 cycles (being well within the 396 cycle limit).

Note that the required input bandwidth decreases as the number of PPUs is reduced. Figures 16 and 17 show the memory organization for an eight PPU based buffer. As shown, each partition is logically subdivided into eight sub-partitions. Within each partition, the sub-partitions are organized physically into three banks. The organization allows the PPUs to access all pixels in columns 0 to 22, 8 to 30, 16 to 38, 24 to 46, 32 to 54, and 40 to 62 in 198 clock cycles. This results in a minimum input bandwidth of 15.6 bytes for the search window and 2.59 bits for the macroblock. Table 1 summarizes the input bandwidth requirements for systems containing 1, 2, 4, 8, 12, and 16 PPUs.

## 6 Experimental Results

To evaluate the performance and area efficiency of the scalable VBSME architecture, we implemented five variations of the design shown in Figure 4 on a Xilinx Virtex 5 XC5VLX330 FPGA. Each variation of the design contains 1, 2, 4, 8, or 12 PPUs and is implemented in three versions based on the original, the two-stage-pipelined, and the three-stage-pipelined row adder tree designs shown in Figure 9(a), Figure 9(c), and Figure 9(d), respectively. As the design scales, the target resolution is scaled from VGA (640x480) to High-Definition (HD) Video (1920x1088) and pipelining is used in conjunction with ISE Equivalent Register Removal to improve the performance of high fanout signals.

Each version of the motion estimation unit are implemented in Verilog and mapped onto FPGAs using the Xilinx Integrated Software Environment (ISE). The synthesis and mapping constraints are set to maximize performance. All designs meet the IO constraint of XC5VLX330 with 80%, 82%, 85%, 90%, and 95% IO utilization, respectively. The performance and area of each implementation are summarized in detail in Table 2, Table 3, and Table 4.

**Table 2: Area and Performance – Original Design**

| # of PPUs | Area |  |  |  | Performance |  |  |
|---|---|---|---|---|---|---|---|
|  | Slice LUTs |  | Slice DFFs |  | Target Resolution | Freq. (MHz) | Fps |
|  | # (K) | % | # (K) | % |  |  |  |
| 1 | 8.69 | 4.19 | 6.76 | 3.26 | 640x480 (VGA) | 193.7 | 25 |
| 2 | 18.6 | 8.97 | 15.6 | 7.54 | 800x608 (SVGA) | 199.9 | 33 |
| 4 | 37.9 | 18.3 | 33.1 | 16.0 | 1024x768 (XVGA) | 195.3 | 40 |
| 8 | 76.5 | 36.9 | 68.4 | 33.0 | 1920x1088 (HD Video) | 191.5 | 29 |
| 12 | 115.1 | 55.5 | 103.5 | 49.9 | 1920x1088 (HD Video) | 149.6 | 34 |

**Table 3: Area and Performance – Two-Stage-Pipelined**

| # of PPUs | Area |  |  |  | Performance |  |  |
|---|---|---|---|---|---|---|---|
|  | Slice LUTs |  | Slice DFFs |  | Target Resolution | Freq. (MHz) | Fps |
|  | # (K) | % | # (K) | % |  |  |  |
| 1 | 9.00 | 4.34 | 12.4 | 5.97 | 640x480 (VGA) | 421.1 | 55 |
| 2 | 19.2 | 9.26 | 26.9 | 13.0 | 800x608 (SVGA) | 398.1 | 66 |
| 4 | 39.1 | 18.9 | 55.6 | 26.8 | 1024x768 (XVGA) | 365.9 | 74 |
| 8 | 79.0 | 38.1 | 113.4 | 54.7 | 1920x1088 (HD Video) | 327.5 | 50 |
| 12 | 99.0 | 47.7 | 113.8 | 54.9 | 1920x1088 (HD Video) | 250.8 | 56 |

**Table 4: Area and Performance – Three-Stage-Pipelined**

| # of PPUs | Area |  |  |  | Performance |  |  |
|---|---|---|---|---|---|---|---|
|  | Slice LUTs |  | Slice DFFs |  | Target Resolution | Freq. (MHz) | Fps |
|  | # (K) | % | # (K) | % |  |  |  |
| 1 | 9.33 | 4.50 | 13.1 | 6.31 | 640x480 (VGA) | 439.2 | 58 |
| 2 | 19.9 | 9.59 | 28.3 | 13.6 | 800x608 (SVGA) | 393.2 | 65 |
| 4 | 40.5 | 19.5 | 58.4 | 28.2 | 1024x768 (XVGA) | 360.2 | 73 |
| 8 | 81.6 | 39.4 | 119.0 | 57.4 | 1920x1088 (HD Video) | 297.8 | 45 |
| 12 | 103.0 | 49.7 | 122.2 | 59.0 | 1920x1088 (HD Video) | 223.3 | 50 |

Table 2 employs the original row adder tree design as shown in Figure 9(a). Column 1 of the table lists the number of PPUs in the design. Columns 2 and 3 lists the number of Slice LUTs required for the design and the number of Slice LUTs required as a percentage of the total number of Slice LUTs in the FPGA, respectively. The same values are summarized in column 4 and 5 for Slice DFFs. Finally column 6 lists the target resolution of each design. The maximum operating frequencies of the circuits are shown in column 7 and their corresponding frame-per-second performances are shown in column 8 (using 4 reference frames per macroblock). Similarly, Table 3 shows the area and performance results for the two-stage-pipelined row adder tree design and Table 4 shows the area and performance results for the three-stage-pipelined row adder tree design.

As shown the original row adder tree implementation achieves 25 (1 PPU at VGA resolution) to 40 (4 PPUs at XVGA resolution) frame-per-second performance as the target resolution is scaled from VGA to HD Video. In particular, the 12-PPU design is able to achieve 34 frame-per-second performance for HD Video while utilizing 115.1K LUTs and 103.5K DFFs.

As shown in Table 3, the use of the two-stage-pipelined row adder trees significantly improves circuit performance. In particular, the maximum clock frequency for the 1 PPU design is increased from 193.7 MHz to 421.1 MHz, which corresponds to a frame rate increase of 120%. For the 12 PPU design the clock frequency is increased from 149.6 MHz to 250.8 MHz, which corresponds to a frame rate increase of 65%. In contrast, increasing the number of pipeline stages from 2 to 3 only slightly increases the performance of the 1 PPU design and decreases the performance of the 2, 4, 8, and 12 PPU designs due to an increase in LUT and DFF count and the subsequent increase in routing delay.

The data demonstrate that as the number of pipeline stages is increased from 2 to 3, the row adder trees are no longer the critical path of the FPGA implementation. Consequently, techniques to further increase the number of pipeline stages in row adder trees beyond 2 stages will not further increase performance.

We also implemented the 12-PPU system with on-chip double buffering using the two stage pipelined row adder tree. The implementation consumes 28 18Kbit Block RAM, 102.0K Slice LUTs and 115.4K Slice DFFs. The system performance is lowered from 250.8 MHz to 224.3 MHz due to an increase in routing delay resulting from the addition of memory. The performance corresponds to a frame-per-second performance of 51 fps.

The circuit performance of the designs employing the two-stage-pipelined and three-stage-pipelined row adder trees remains consistently over 30 frames per second as the design scales from 1 to 12 PPUs and the target resolution scales from VGA to HD Video for FPGA-based H.264/AVC motion estimation. The results show that real time motion estimation performances can be achieved with 1, 2, 4, and 8 PPUs for the resolutions of VGA, SVGA, XVGA, and HD Video, respectively. It also shows that 8 or more PPUs can achieve real time motion estimation for resolutions that are beyond HD Video.

## 7 Conclusions

It is shown that the proposed architecture is able to perform real time (50 - 56fps) H.264/AVC Motion Estimation on 1920x1088 progressive HD video and is capable of being scaled for higher resolutions. The performance is measured with four reference frames and a search window size of 63 x 48 pixels. When scaled for HD-level performance, the architecture utilizes 79K LUTs and 113K DFFs (with 8 processing units), and has a maximum clock frequency of 328 MHz when implemented on a Xilinx XC5VLX330 (Virtex-5) FPGA. Furthermore, the scalability of the architecture makes it suitable for FPGA-based applications where the upgradeability and flexibility of the video encoder are essential requirements.

*References:*

[1] Yu-Wen Huang, Tu-Chih Wang, Bing-Yu Hsieh, Liang-Gee Chen, "Hardware Architecture Design for Variable Block Size Motion Estimation in MPEG-4 AVC/JVT/ITU-T H.264," *Proceedings of the 2003 International Symposium on Circuits and Systems,* Vol. 2, pp. 25-28, May 2003.

[2] S. Yap and J. V. McCanny, "A VLSI Architecture for Variable Block Size Video Motion Estimation," *IEEE Transactions on Circuits and Systems II,* Vol. 51, No. 7, pp. 384-389, July 2004.

[3] M. Kim, I. Hwang, and S. Chae, "A Fast VLSI Architecture for Full-Search Variable Block Size Motion Estimation in MPEG-4 AVC/H.264," *Proceedings of the 2005 conference on Asia South Pacific design automation,* 2005, pp. 631-634.

[4] Yang Song, Zhenyu Liu, Satoshi Goto, Takeshi Ikenaga, "Scalable VLSI Architecture for Variable Block Size Integer Motion Estimation in H.264/AVC," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences,* Vol. E89-A, No. 4, pp. 979-988, April 2006.

[5] Yang Song, Zhenyu Liu, Takeshi Ikenaga, Satoshi Goto, "VLSI Architecture for Variable Block Size Motion Estimation in H.264/AVC with Low Cost Memory Organization," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences,* Vol. E89-A, No. 12, pp. 3594-3601, December 2006.

[6] Zhenyu Liu, Yang Song, Takeshi Ikenaga, Satoshi Goto, "A Fine-Grain Scalable and Low Memory Cost Variable Block Size Motion Estimation Architecture for H.264/AVC," *IEICE Transactions on Electronics,* Vol. E89-C, No. 12, pp. 1928-1936, December 2006.

[7] Tung-Chien Chen, Shao-Yi Chien, Yu-Wen Huang, Chen-Han Tsai, Ching-Yeh Chen, To-Wei Chen, Liang-Gee Chen, "Analysis and Architecture Design of an HDTV720p 30 Frames/s H.264/AVC Encoder," IEEE Transactions on Circuits and Systems for Video Technology, Vol. 16, No. 6, pp. 673-688, June 2006.

[8] Zhenyu Liu, Yiqing Huang, Yang Song, Satoshi Goto, Takeshi Ikenaga, "Hardware-Efficient Propagate Partial SAD Architecture for Variable Block Size Motion Estimation in H.264/AVC," *Proceedings of the 17th Great Lakes Symposium on VLSI,* 2007, pp. 160-163.

[9] W. Chung, "Implementing the H.264/AVC Video Coding Standards on FPGAs," *Xilinx Broadcast Solution Guide,* September 2005, pp. 18-21.

[10] "Faraday H.264 Baseline Video Encoder & Decoder IPs: FTMCP210/FTMCP220," *Faraday Technology Corporation Product Documentation,* 2005.

[11] "H.264 Motion Estimation Engine (DO-DI-H264-ME)," *Xilinx Corporation Product Documentation,* October 2007.

[12] S. Lopez, F. Tobajas, A. Villar, V. de Armas, J.F. Lopez, R. Sarmiento, "Low Cost Efficient Architecture for H.264 Motion Estimation," *IEEE International Symposium on Circuits and Systems*, Vol. 1, pp. 412-415, May 2005.

[13] S. Yalcin, H.F. Ates, I. Hamzaoglu, "A High Performance Hardware Architecture for an SAD Reuse Based Hierarchical Motion Estimation Algorithm for H.264 Video Coding," *Proceedings of the 2005 International Conference on Field Programmable Logic and Applications,* 2005, pp. 509-514.

[14] "DaVinci Technology Overview," Texas Instruments Inc., Dallas, TX, 2010 [Online]. Available: http://www.ti.com

[15] "Blackfin Processor Architecture Overview," Analog Devices Inc., Norwood, MA, 2010 [Online]. Available: http://www.analog.com

[16] R. Li, B. Zeng, and M. Liou, "A New Three-Step Search Algorithm for Block Motion Estimation," *IEEE Transactions on Circuits and Systems for Video Technology,* Vol. 4, No. 4, pp.438-442, August 1994.

[17] C. Zhu, X. Lin, and L. Chau, "Hexagon-Based Search Pattern for Fast Block Motion Estimation," *IEEE Transactions on Circuits and Systems for Video Technology,* Vol. 12, No. 5, pp.349-355, May 2002.

[18] L. Yang, K. Yu, J. Li, and S. Li, "An Effective Variable Block-Size Early Termination Algorithm for H.264 Video Coding," *IEEE Transactions on Circuits and Systems for Video Technology,* Vol. 15, No. 6, pp.784-788, June 2005.

[19] Theepan Moorthy and Andy Ye, "A Scalable Computing and Memory Architecture for Variable Block Size Motion Estimation on Field-Programmable Gate Arrays," *Proceedings of the 2008 International Conference on Field Programmable Logic and Applications,* 2008, pp. 83-88.