Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2009, Article ID 893897, 16 pages
doi:10.1155/2009/893897

Research Article

FPSoC-Based Architecture for a Fast Motion Estimation Algorithm in H.264/AVC

Obianuju Ndili and Tokunbo Ogunfunmi

Department of Electrical Engineering, Santa Clara University, Santa Clara, CA 95053, USA

Correspondence should be addressed to Tokunbo Ogunfunmi, togunfunmi@scu.edu

Received 21 March 2009; Revised 18 June 2009; Accepted 27 October 2009

Recommended by Ahmet T. Erdogan

There is an increasing need for high quality video on low power, portable devices. Possible target applications range from entertainment and personal communications to security and health care. While H.264/AVC answers the need for high quality video at lower bit rates, it is significantly more complex than previous coding standards and thus results in greater power consumption in practical implementations. In particular, motion estimation (ME), in H.264/AVC consumes the largest power in an H.264/AVC encoder. It is therefore critical to speed-up integer ME in H.264/AVC via fast motion estimation (FME) algorithms and hardware acceleration. In this paper, we present our hardware oriented modifications to a hybrid FME algorithm, our architecture based on the modified algorithm, and our implementation and prototype on a PowerPC-based Field Programmable System on Chip (FPSoC). Our results show that the modified hybrid FME algorithm on average, outperforms previous state-of-the-art FME algorithms, while its losses when compared with FSME, in terms of PSNR performance and computation time, are insignificant. We show that although our implementation platform is FPGA-based, our implementation results compare favourably with previous architectures implemented on ASICs. Finally we also show an improvement over some existing architectures implemented on FPGAs.

1. Introduction

Motion estimation (ME) is by far the most powerful compression tool in the H.264/AVC standard [1, 2], and it is generally carried out in two stages: integer-pel then fractional pel as a refinement of the integer-pel search. ME in H.264/AVC features variable block sizes, quarter-pixel accuracy for the luma component (one-eighth pixel accuracy for the chroma component), and multiple reference pictures. However the power of ME in H.264/AVC comes at the price of increased encoding time. Experimental results [3, 4] have shown that ME can consume up to 80% of the total encoding time of H.264/AVC, with integer ME consuming a greater proportion. In order to meet real-time and low power constraints, it is desirable to speed up the ME process. Two approaches to ME speed-up include designing fast ME algorithms and accelerating ME in hardware.

Considering the algorithm approach, there are traditional, single search fast algorithms such as new three-step search (NTSS) [5], four-step search (4SS) [6], and diamond search (DS) [7]. However these algorithms were developed for fixed block size and cannot efficiently support variable block size ME (VBSME) for H.264/AVC. In addition, while these algorithms are good for small search range and low resolution video, at higher definition for some high motion sequences such as "Stefan," these algorithms can drop into a local minimum in the early stages of the search process [4]. In order to have more robust fast algorithms, some hybrid fast algorithms that combine earlier single search techniques have been proposed. One of such was proposed by Yi et al. [8, 9]. They proposed a fast ME algorithm known variously as the Simplified Unified Multi-Hexagon (SUMH) search or Simplified Fast Motion Estimation (SFME) algorithm. SUMH is based on UMHexagonS [4], a hybrid fast motion estimation algorithm. Yi et al. show in [8] that with similar or

even better rate-distortion performance, SUMH reduces ME time by about 55% and 94% on average when compared with UMHexagonS and Fast Full Search, respectively. In addition, SUMH yields a bit rate reduction of up to 18% when compared with Full Search in low complexity mode. Both SUMH and UMHexagonS are nonnormative parts of the H.264/AVC standard.

Considering ME speed-up via hardware acceleration, although there has been some previous work on VLSI architectures for VBSME in H.264/AVC, the overwhelming majority of these works have been based on the Full Search Motion Estimation (FSME) algorithm. This is because FSME presents a regular-patterned search window which in turn provides good candidate-level data reuse (DR) with regular searching flows. A good candidate-level DR results in the reduction of data access power. Power consumption for an integer ME module mainly comes from two parts: data access power to read reference pixels from local memories and computational power consumed by the processing elements. For FSME, the data access power is reduced because the reference pixels of neighbouring candidates are considerably overlapped. On the other hand, because of the exhaustive search done in FSME, the computational complexity and thus the power consumed by the processing elements, is large.

Several low-power integer ME architectures with corresponding fast algorithms were designed for standards prior to H.264/AVC [10–13]. However, these architectures do not support H.264/AVC. Additionally, because the irregular searching flows of fast algorithms usually lead to poor intercandidate DR, the power reduction at the algorithm level is usually constrained by the power reduction at the architecture level. There is therefore an urgent need for architectures with hardware oriented fast algorithms for portable systems implementing H.264/AVC [14]. Note also that because the data flow of FME is very similar to that of fractional pel search, some hardware reuse can be achieved [15].

For H.264/AVC, previous works on architectures for fast motion estimation (FME) [14–18] have been based on diverse FME algorithms.

Rahman and Badawy in [16] and Byeon et al. in [17] base their works on UMHexagonS. In [14], Chen et al. propose a parallel, content-adaptive, variable block size, 4SS algorithm, upon which their architecture is based. In [15], Zhang and Gao base their architecture on the following search sequence: Diamond Search (DS), Cross Search (CS) and finally, fractional-pel ME.

In this paper, we base our architecture on SUMH which has been shown in [8] to outperform UMHexagonS. We present hardware oriented modifications to SUMH. We show that the modified SUMH has a better PSNR performance that of the parallel, content-adaptive variable block size 4SS proposed in [14]. In addition, our results (see Section 2) show that for the modified SUMH, the average PSNR loss is 0.004 dB to 0.03 dB when compared with FSME, while when compared to SUMH, most of the sequences show an average improvement of up to 0.02 dB, while two of the sequences show an average loss of 0.002 dB. Thus in general, there is an improvement over SUMH. In terms of percentage computational time savings, while SUMH saves 88.3% to 98.8% when compared with FSME, the modified SUMH saves 60.0% to 91.7% when compared with FSME. Finally, in terms of percentage bit rate increase, when compared with FSME, the modified SUMH shows a bit rate improvement (decrease in bit rate), of 0.02% in the sequence "Coastguard." The worst bit rate increase is in "Foreman" and that is 1.29%. When compared with SUMH, there is a bit rate improvement of 0.03% to 0.34%.

The rest of this paper is organized as follows. In Section 2 we summarize integer-pel motion estimation in SUMH and present the hardware oriented SUMH along with simulation results. In Section 3 we briefly present our proposed architecture based on the modified SUMH. We also present our implementation results as well as comparisons with prior works. In Section 4 we present our prototyping efforts on the XUPV2P development board. This board contains an XC2VP30 Virtex-II Pro FPGA with two hardwired PowerPC 405 processors. Finally our conclusions are presented in Section 5.

2. Motion Estimation Algorithm

2.1. Integer-Pel SUMH Algorithm. H.264/AVC uses block matching for motion vector search. Integer-pel motion estimation uses the sum of absolute differences (SADs), as its matching criterion. The mathematical expression for SAD is given in

$$\mathrm{SAD}(dx,dy) = \sum_{x=0}^{X-1}\sum_{y=0}^{Y-1} \left| a(x,y) - b(x+dx, y+dy) \right|, \tag{1}$$

$$\left(\mathrm{MV}_x, \mathrm{MV}_y\right) = (dx,dy)\,|_{\min \mathrm{SAD}(dx,dy).} \tag{2}$$

In (1), $a(x,y)$ and $b(x,y)$ are the pixels of the current, and candidate blocks, respectively. (dx,dy) is the displacement of the candidate block within the search window. $X \times Y$ is the size of the current block. In (2) $(\mathrm{MV}_x, \mathrm{MV}_y)$ is the motion vector of the best matching candidate block.

H.264/AVC features seven interprediction block sizes which are 16×16, 16×8, 8×16, 8×8, 8×4, 4×8, and 4×4. These are referred to as block modes 1 to 7. An up layer block is a block that contains sub-blocks. For example, mode 5 or 6 is the up layer of mode 7, and mode 4 is the up layer of mode 5 or 6.

SUMH [8] utilizes five key steps for intensive search, integer-pel motion estimation. They are cross search, hexagon search, multi big hexagon search, extended hexagon search, and extended diamond search. For motion vector (MV) prediction, SUMH uses the spatial median and up layer predictors, while for SAD prediction, the up layer predictor is used. In median MV prediction, the median value of the adjacent blocks on the left, top, and top-right (or top-left) of the current block is used to predict the

MV of the current block. The complete flow chart of the integer-pel, motion vector search in SUMH is shown in Figure 1.

The convergence and intensive search conditions are determined by arbitrary thresholds shifted by a blocktype shift factor. The blocktype shift factor specifies the number of bits to shift to the right in order to get the corresponding thresholds for different block sizes. There are 8 blocktype shift factors corresponding to 8 block modes: 1 dummy block mode and the 7 block modes in H.264/AVC. The 8 block modes are 16×16 (dummy), 16×16, 16×8, 8×16, 8×8, 8×4, 4 × 8, and 4 × 4. The array of 8 blocktype shift factors corresponding, respectively, to these 8 block modes is given in

$$\text{blocktype\_shift\_factor} = \{0, 0, 1, 1, 2, 3, 3, 1\}. \tag{3}$$

The convergence search condition is described in pseudocode in

$$(\text{min\_mcost} < (\text{ConvergeThreshold} \gg \text{blocktype\_shift\_factor}[\text{blocktype}])), \tag{4}$$

where min_mcost is the minimum motion vector cost. The intensive search condition is described in pseudo-code in

$$\left\{\begin{array}{c}\left(\begin{array}{c}\text{blocktype} == 1\ \&\& \\ \left(\text{min\_mcost} > (\text{CrossThreshold1} \gg \text{blocktype\_shift\_factor}[\text{blocktype}])\right)\end{array}\right) \\ || \\ \left(\text{min\_mcost} > (\text{CrossThreshold2} \gg \text{blocktype\_shift\_factor}[\text{blocktype}])\right)\end{array}\right\}, \tag{5}$$

where the thresholds are empirically set as follows: ConvergeThreshold = 1000, CrossThreshold1 = 800, and CrossThreshold2 = 7000.

2.2. Hardware Oriented SUMH Algorithm. The goal of our hardware oriented modification is to make SUMH less sequential without incurring performance losses or increases in the computation time.

The sequential nature of SUMH arises from the fact that there are a lot of data dependencies. The most severe data dependency arises during the up layer predictor search step. This dependency forces the algorithm to sequentially and individually conduct the search for the 41 possible SADs in a 16 × 16 macroblock. The sequence begins with the 16 × 16 macroblock then computes the SADs of the subblocks in each quadrant of the 16 × 16 macroblock. Performing the algorithm in this manner consumes a lot of computational time and power, yet its rate-distortion benefits can still be obtained in a parallel implementation. In our modification, we skip this search step.

The decision control structures in SUMH are another feature that makes the algorithm unsuitable for hardware implementation. In a parallel and pipelined implementation, these structures would require that the pipeline be flushed at random times. This is in turn wasteful of clock cycles as well as adds more overhead to the hardware's control circuit. In our modification, we consider the convergence condition not satisfied, and intensive search condition satisfied. This removes the decision control structures that make SUMH unsuitable for parallel processing. Another effect of this modification is that we expect to have a better rate-distortion performance. On the other hand, the expected disadvantage of this modification is an increase in computation time. However, as shown by our complexity analysis and results, this increase is minimal and will also be easily compensated for by hardware acceleration.

Further modifications we make to SUMH are the removal of the small local search steps and the convergence search step.

Our modifications to SUMH allow us to process in parallel, all the candidate macroblocks (MB), for one current macroblock (CMB). We use the so-called HF3V2 2-stitched zigzag scan proposed in [19], in order to satisfy the data dependencies between CMBs. These data dependencies arise because of the side information used to predict the MV of the CMB. Note that if we desire to process several CMBs in parallel, we will need to set the value of the MV predictor to the zero displacement MV, that is, MV = (0, 0). Experiments in [20–22], as well as our own experiments [23], show that when the search window is centered around MV = (0, 0), the average PSNR loss is less than 0.2 dB compared with when the median MV is also used. Figure 2 shows the complete flow chart of the modified integer-pel, SUMH.

2.3. Complexity Analysis of the Motion Estimation Algorithms. We consider a search range s. The number of search points to be examined by FSME algorithm is directly proportional to the square of the search range. There are $(2s + 1)^2$ search points. Thus the algorithm complexity of Full Search is $O(s^2)$.

We obtain the algorithm complexity of the modified SUMH algorithm by considering the algorithm complexity of each of its search steps as follows.

(1) Cross search: there are s search points both horizontally and vertically yielding a total of $2s$ search points. Thus the algorithm complexity of this search step is $O(2s)$.

(2) Hexagon and extended hexagon search: There are 6 search points each in both of these search steps, yielding a total of 12 search points. Thus the algorithm complexity of this search step is constant $O(1)$.

(3) Multi-big hexagon search: there are $(1/4)s$ hexagons with 16 search points per hexagon. This yields a total of $4s$ search points. Thus the algorithm complexity of this search step is $O(4s)$.

(4) Diamond search: there are 4 search points in this search step. Thus the algorithm complexity of this search step is constant $O(1)$.

Therefore in total there are $1 + 2s + 12 + 4 + 4s$ search points in the modified SUMH, and its algorithm complexity is $O(6s)$.

In order to obtain the algorithm complexity of SUMH, we consider its worst case complexity, even though the

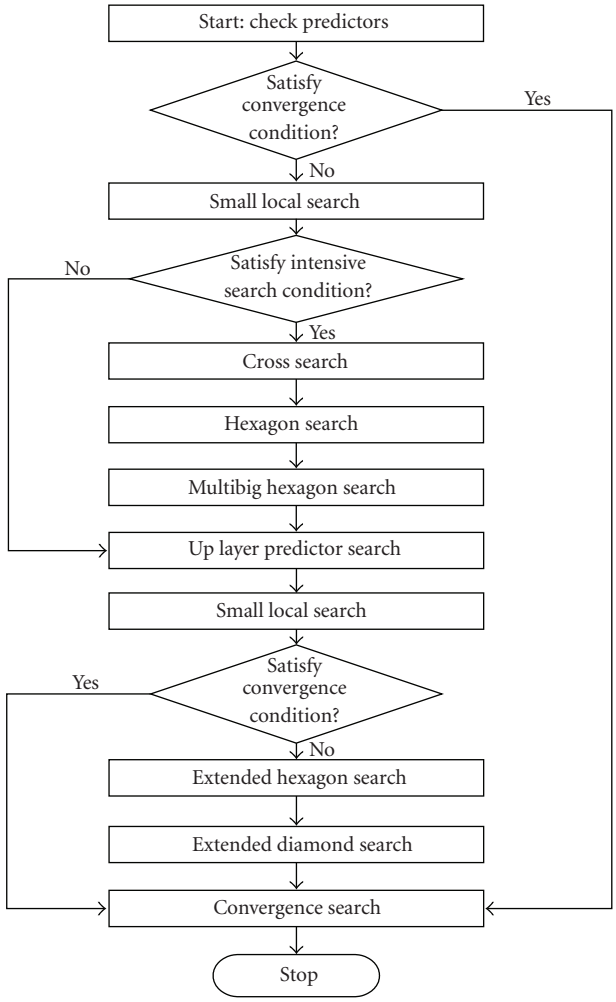

Figure 1: Flow chart of integer-pel search in SUMH.

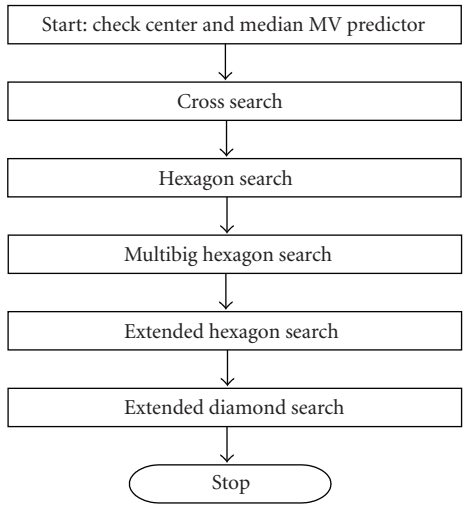

Figure 2: Flow chart of modified integer-pel search.

Table 1: Complexity of algorithms in million operations per second (MOPS).

Algorithm	Number of search points for search range $s = \pm 16$	Number of MOPS for CIF video at 30 Hz
FSME	1089	17103
Best case SUMH	5	78
Worst case SUMH	127	1995
Median case SUMH	66	1037
Modified SUMH	113	1775

algorithm may terminate much earlier. The worst case complexity of SUMH is similar to that of the modified SUMH, except that it adds 14 more search points. This number is obtained by adding 4 search points each for 2 small local searches and 1 convergence search, and 2 search points for the worst case up layer predictor search. Thus for the worst case SUMH, there are in total $14+1+2s+12+4+4s$ search points and its algorithm complexity is $O(6s)$. Note that in the best case, SUMH has only 5 search points: 1 for the initial search candidate and 4 for the convergence search.

Another way to define the complexity of each algorithm is in terms of the number of required operations. We can then express the complexity as Million Operations Per Second (MOPS). To compare the algorithms in terms of MOPS we assume the following.

(1) The macroblock size is 16×16.

(2) The SAD cost function requires $2 \times 16 \times 16$ data loads, $16 \times 16 = 256$ subtraction operations, 256 absolute operations, 256 accumulate operations, 41 compare operations and 1 data store operation. This yields a total of 1322 operations for one SAD computation.

(3) CIF resolution is 352×288 pixels = 396 macroblocks.

(4) The frame rate is 30 frames per second.

(5) The total number of operations required to encode CIF video in real time is $1322 \times 396 \times 30 \times z_a$, where z_a is the number of search points for each algorithm.

Thus there are $15.7z_a$ MOPS per algorithm, where one OP (operation) is the amount of computation it takes to obtain one SAD value.

In Table 1 we compare the computational complexities of the considered algorithms in terms of MOPS. As expected, FSME requires the largest number of MOPS. The number of MOPS required for the modified SUMH is about 10% less than that required for the worst case SUMH and about 40% more than that required for the median case SUMH.

2.4. Performance Results for the Modified SUMH Algorithm. Our experiments are done in JM 13.2 [24]. We use the following standard test sequences: "Stefan" (large motion), "Foreman" and "Coastguard" (large to moderate motion) and "Silent" (small motion). We chose these sequences because we consider them extreme cases in the spectrum of low bit-rate video applications. We also use the following

Table 2: Simulation conditions.

Sequences	Quantization parameter	Search range	Frame size	No. of frames
Foreman	22, 25, 28, 31, 33, 35	32	CIF	100
Mother-daughter	22, 25, 28, 31, 33, 35	32	CIF	150
Stefan	22, 25, 28, 31, 33, 35	16	CIF	90
Flower	22, 25, 28, 31, 33, 35	16	CIF	150
Coastguard	18, 22, 25, 28, 31, 33	32	QCIF	220
Carphone	18, 22, 25, 28, 31, 33	32	QCIF	220
Silent	18, 22, 25, 28, 31, 33	16	QCIF	220

Table 3: Comparison of speed-up ratios with full search.

Quantization Parameter	18		22		25		28		31		33		35	
	SUMH	Modified SUMH	SUMH	Modified SUMH	SUMH	Modified SUMH	SUMH	Modified SUMH	SUMH	Modified SUMH	SUMH	Modified SUMH	SUMH	Modified SUMH
Foreman	N/A	N/A	48.55	8.16	41.55	6.86	32.68	5.66	25.87	4.77	21.68	4.23	19.11	3.74
Stefan	N/A	N/A	15.35	4.62	13.16	4.21	12.20	3.93	10.67	3.50	10.05	3.23	8.96	3.06
Mother-daughter	N/A	N/A	16.63	2.49	19.31	2.72	21.56	3.01	28.63	3.47	35.43	4.20	43.90	5.08
Flower	N/A	N/A	9.73	3.07	10.72	3.29	11.32	3.49	12.94	3.78	13.77	4.02	15.02	4.21
Coastguard	86.34	12.06	70.12	10.31	58.05	9.01	43.62	7.98	36.04	6.80	30.10	6.13	N/A	N/A
Silent	21.86	3.54	16.74	3.18	13.17	2.99	11.90	2.82	9.29	2.66	8.56	2.64	N/A	N/A
Carphone	24.67	4.14	29.44	4.62	37.12	5.38	46.97	6.02	53.97	7.07	64.07	8.82	N/A	N/A

Table 4: Comparison of percentage time savings with full search.

Quantization Parameter	18		22		25		28		31		33		35	
	SUMH	Modified SUMH	SUMH	Modified SUMH	SUMH	Modified SUMH	SUMH	Modified SUMH	SUMH	Modified SUMH	SUMH	Modified SUMH	SUMH	Modified SUMH
Foreman	N/A	N/A	97.94	87.75	97.59	85.43	96.94	82.34	96.13	79.04	95.38	76.36	94.76	73.31
Stefan	N/A	N/A	93.48	78.38	92.40	76.29	91.80	74.61	90.63	71.46	90.05	69.05	88.83	67.35
Mother-daughter	N/A	N/A	93.98	60.00	94.82	63.34	95.36	66.85	96.50	71.22	97.17	76.21	97.72	80.35
Flower	N/A	N/A	89.72	67.45	90.67	69.62	91.16	71.37	92.27	73.56	92.71	75.14	93.34	76.27
Coastguard	98.84	91.71	98.57	90.30	98.27	88.91	97.70	87.47	97.22	85.29	96.67	83.70	N/A	N/A
Silent	95.42	71.77	94.02	68.62	92.40	66.61	91.60	64.56	89.23	62.47	88.32	62.20	N/A	N/A
Carphone	95.94	75.87	96.60	78.36	97.30	81.41	97.87	83.41	98.14	85.87	98.43	88.66	N/A	N/A

sequences: "Mother-daughter" (small motion, talking head and shoulders), "Flower" (large motion with camera panning), and "Carphone" (large motion). The sequences are coded at 30 Hz. The picture sequence is IPPP with I-frame refresh rate set at every 15 frames. We consider 1 reference frame. The rest of our simulation conditions are summarized in Table 2.

Figure 3 shows curves that compare the rate-distortion efficiencies of Full Search ME, SUMH, and the modified SUMH. Figure 4 shows curves that compare the rate-distortion efficiencies of Full Search ME and the single- and multiple-iteration parallel content-adaptive 4SS of [14]. In Tables 3 and 4, we show a comparison of the speed-up ratios of SUMH and the modified SUMH. Table 5 shows the average percentage bit rate increase of the modified SUMH when compared with Full Search ME and SUMH. Finally Table 6 shows the average Y-PSNR loss of the modified SUMH when compared with Full Search ME and SUMH.

From Figures 3 and 4, we see that the modified SUMH has a better rate-distortion performance than the proposed parallel content-adaptive 4SS of [14], even under smaller search ranges. In Section 3 we will show comparisons of our supporting architecture with the supporting architecture

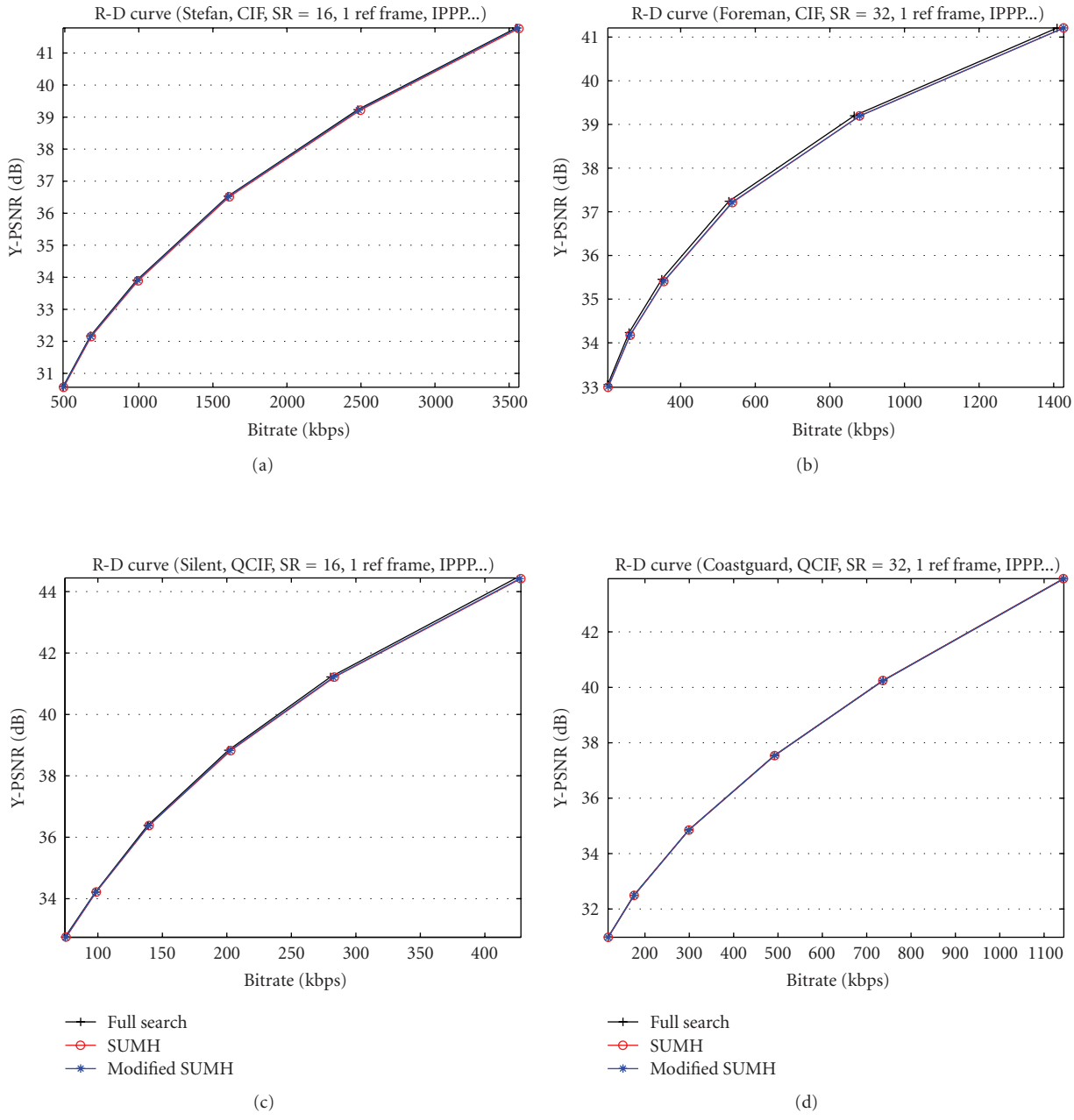

Figure 3: Comparison of rate-distortion efficiencies for the modified SUMH.

proposed in [14]. Note though that the architecture in [14] is implemented on an ASIC (TSMC 0.18-μ 1P6M technology), while our architecture is implemented on an FPGA.

From Figure 3 and Table 6 we also observe that the largest PSNR losses occur in the "Foreman" sequence, while the least PSNR losses occur in "Silent." This is because the "Foreman" sequence has both high local object motion and greater high-frequency content. It therefore performs the worst under a given bit rate constraint. On the other hand, "Silent" is a low motion sequence. It therefore performs much better under the same bit rate constraint.

Given the tested frames from Table 2 for each sequence, we observe additionally from Table 6 that Full Search performs better than the modified SUMH for sequences with larger local object (foreground) motion, but little or no background motion. These sequences include "Foreman," "Carphone," "Mother-daughter," and "Silent." However the rate-distortion performance of the modified SUMH improves for sequences with large foreground and background motions. Such sequences include "Flower," "Stefan," and "Coastguard." We therefore suggest that a yet greater improvement in the rate-distortion performance of

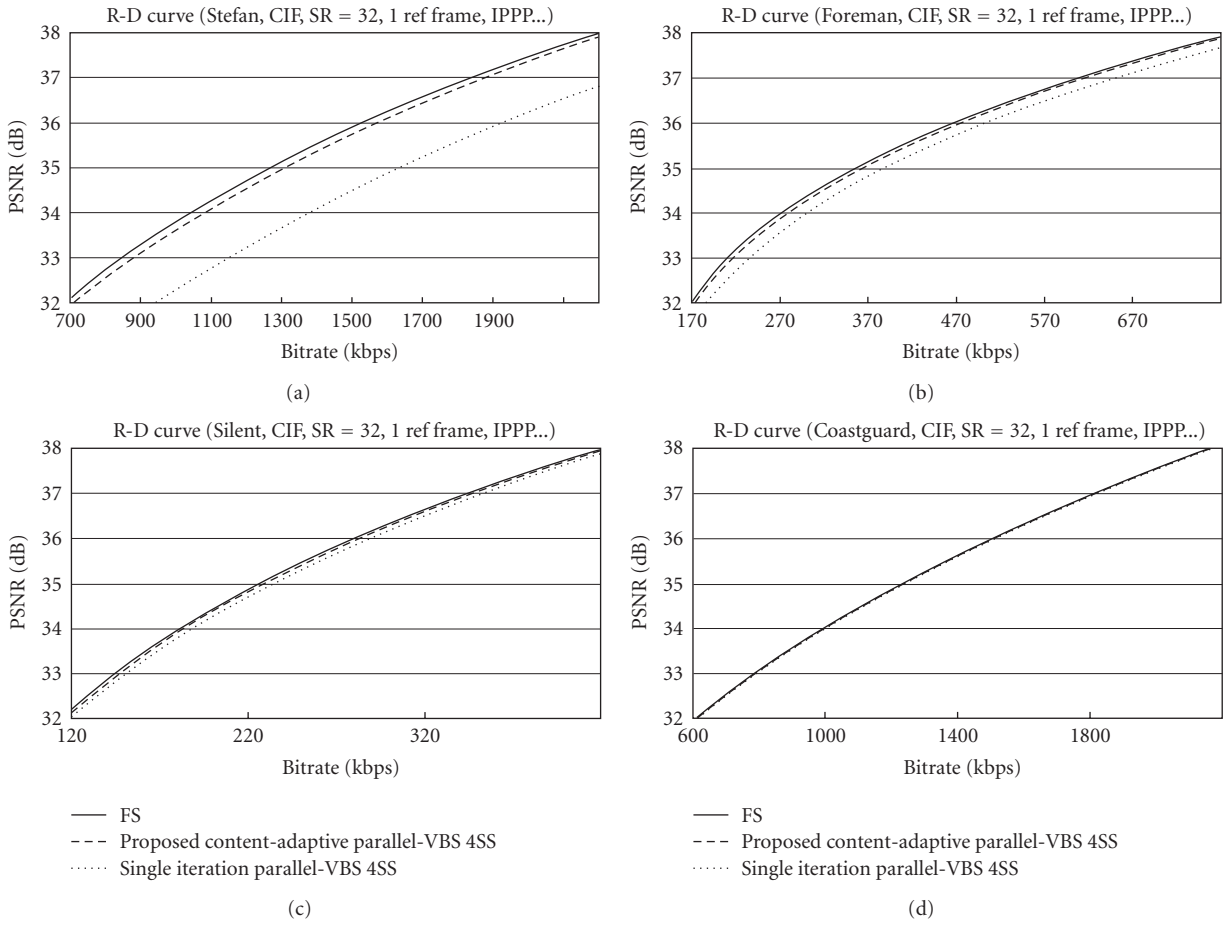

Figure 4: Comparison of rate-distortion efficiencies for parallel content-adaptive 4SS of [25] (Reproduced from [25]).

the modified SUMH algorithm can be achieved by improving its local motion estimation.

For Table 3, we define the speed-up ratio as the ratio of the ME coding time of Full Search to ME coding time of the algorithm under consideration. From Table 3 we see that speed-up ratio increases as quantization parameter (QP) decreases. This is because there are less skip mode macroblocks as QP decreases. From our results in Table 3, we further calculate the percentage time savings t for ME calculation, according to

$$t = \left(1 - \frac{1}{r}\right) \times 100, \tag{6}$$

where r are the data points in Table 3. The percentage time savings obtained are displayed in Table 4. From Table 4, we find that SUMH saves 88.3% to 98.8% in ME computation time compared to Full Search, while the modified SUMH saves 60.0% to 91.7%. Therefore, the modified SUMH does not incur much loss in terms of ME computation time.

In our experiments we set rate-distortion optimization to high complexity mode (i.e., rate-distortion optimization is turned on), in order to ensure that all of the algorithms compared have a fair chance to yield their highest rate-distortion performance. From Table 5 we find that the average percentage bit rate increase of the modified SUMH is very low. When compared with Full Search, there is a bit rate improvement (decrease in bit rate), in "Coastguard" of 0.02%. The worst bit rate increase is in "Foreman" and that is 1.29%. When compared with SUMH, there is a bit rate improvement (decrease in bit rate), going from 0.04% (in "Coastguard") to 0.34% (in "Stefan").

From Table 6 we see that the average PSNR loss for the modified SUMH is very low. When compared to Full Search, the PSNR loss for modified SUMH ranges from 0.006 dB to

Table 5: Average percentage bit rate increase for modified SUMH.

Sequences	Compared with	
	Full search	SUMH
Foreman	1.29	−0.04
Stefan	0.40	−0.34
Mother-daughter	0.15	−0.05
Flower	0.19	−0.17
Coastguard	−0.02	−0.03
Silent	0.56	−0.33
Carphone	0.27	−0.06

0.03 dB. When compared to SUMH, most of the sequences show a PSNR improvement of up to 0.02 dB, while two of the sequences show a PSNR loss of 0.002 dB.

Thus in general, the losses when compared with Full Search are insignificant, while on the other hand there is an improvement when compared with SUMH. We therefore conclude that the modified SUMH can be used without much penalty, instead of Full Search ME, for ME in H.264/AVC.

3. Proposed Supporting Architecture

Our top-level architecture for fast integer VBSME is shown in Figure 5. The architecture is composed of search window (SW) memory, current MB memory, an address generation unit (AGU), a control unit, a block of processing units (PUs), an SAD combination tree, a comparison units and a register for storing the 41 minimum SADs and their associated motion vectors.

While the current and reference frames are stored off-chip in external memory, the current MB (CMB) data and the search window (SW) data are stored in on-chip, dual-port block RAMS (BRAMS). The SW memory has N 16×16 BRAMs that store N candidate MBs, where N is related to the search range s. N can be chosen to be any factor or multiple of $|s|$ so as to achieve a tradeoff between speed and hardware costs. For example, if we consider a search range of $s = \pm 16$, then we can choose N such that $N \in \{\ldots, 32, 16, 8, 4, 2, 1\}$. The AGU generates addresses for blocks being processed.

There are N PUs each containing 16 processing elements (PEs), in a 1D array. A PU shown in Figure 6 calculates 16 4×4 SADs for one candidate MB while a PE shown in Figure 8 calculates the absolute difference between two pixels, one each from the candidate MB and the current MB. From Figure 6, groups of 4 PEs in the PU calculate 1 column of 4×4 SADs. These are stored via demultiplexing, in registers D1–D4 which hold the inputs to the SAD combination tree, one of which is shown in Figure 7. For N PUs there are N SAD combination trees. Each SAD combination tree further combines the 16 4×4 output SADs from one PU, to yield a total of 41 SADs per candidate MB. Figure 7 shows that the 16 4×4 SADs are combined such that registers D6 contain 4×8 SADs, D7 contain 8×8 SADs, D8 contain 8×16 SADs, D9 contain 16×8 SADs, D10 contain 8×4 SADs, and finally, D11 contains the 16×16 SAD. These SADs are compared appropriately in the comparison unit (CU). CU consists of 41 N-input comparing elements (CEs). A CE is shown in Figure 9.

3.1. Address Generation Unit. For each of N MBs being processed simultaneously, the AGU generates the addresses of the top row and the leftmost column of 4×4 sub-blocks. The address of each sub-block is the address of its top left pixel. From the addresses of the top row and leftmost column of 4×4 sub-blocks, we obtain the addresses of all other block partitions in the MB.

The interface of the AGU is fixed and we parameterize it by the address of the current MB, the search type and the search pass. The search type is modified SUMH. However we can expand our architecture to support other types of search, for example, Full Search, and so forth. The search pass depends on the search step and the search range. We show for instance, in Table 7 that there are 15 search passes for the modified SUMH considering a search range $s = \pm 16$. There is a separation of 2 pixels between 2 adjacent search points in the cross search, therefore address generation for search pass 1 to 4 in Table 7 is straightforward. For the remaining search passes5–15, tables of constant offset values are obtained from JM reference software [24]. These offset values are the separation in pixels, between the minimum MV from the previous search pass, and the candidate search point. In general, the affine address equations can be represented by

$$AE_x = iC_x, \qquad AE_y = iC_y, \tag{7}$$

where AE_x and AE_y are the horizontal and vertical addresses of the top left pixel in the MB, i is a multiplier, C_x and C_y are constants obtained from JM reference software.

3.2. Memory. Figures 10 and 11 show CMB and search window (SW) memory organization for $N = 8$ PUs. Both CMB and SW memories are synthesized into BRAMs. Considering a search range of $s = \pm 16$, there are 15 search passes for the modified SUMH search flowchart shown in Figure 2. These search passes are shown in Table 7. In each search pass, 8 MBs are processed in parallel, hence the SW memory organization is shown in Figure 11. SW memory is 128 bytes wide and the required memory size is 2048 bytes. For the same search range $s = \pm 16$, if FSME was used along with levels A and B data reuse, the SW size would be

Table 6: Average Y-PSNR loss for modified SUMH.

Sequences	Compared with	
	Full search	SUMH
Foreman	0. 0290 dB	−0. 0065 dB
Stefan	0. 0058 dB	−0. 0125 dB
Mother-daughter	0. 0187 dB	−0. 0020 dB
Flower	0. 0042 dB	−0. 0002 dB
Coastguard	0. 0078 dB	0. 0018 dB
Silent	0. 0098 dB	0. 0018 dB
Carphone	0. 0205 dB	−0. 0225 dB

Table 7: Search passes for modified SUMH.

Pass	Description
1-2	Horizontal scan of cross search. Candidate MBs seperated by 2 pixels
3-4	Vertical scan of cross search. Candidate MBs seperated by 2 pixels
5	Hexagon search has 6 search points
6–13	Multi-big hexagon search has $(1/4)(\|s\|)$ hexagons, each containing 16 search points
14	Extended hexagon search has 6 search points
15	Diamond search has 4 search points

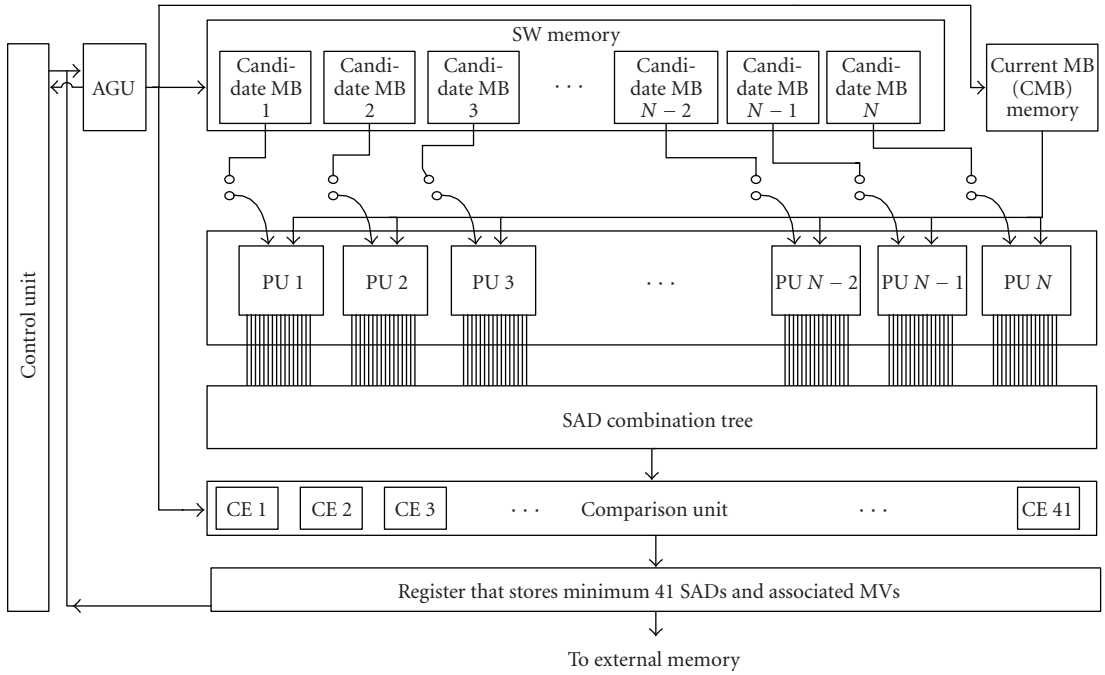

Figure 5: The proposed architecture for fast integer VBSME.

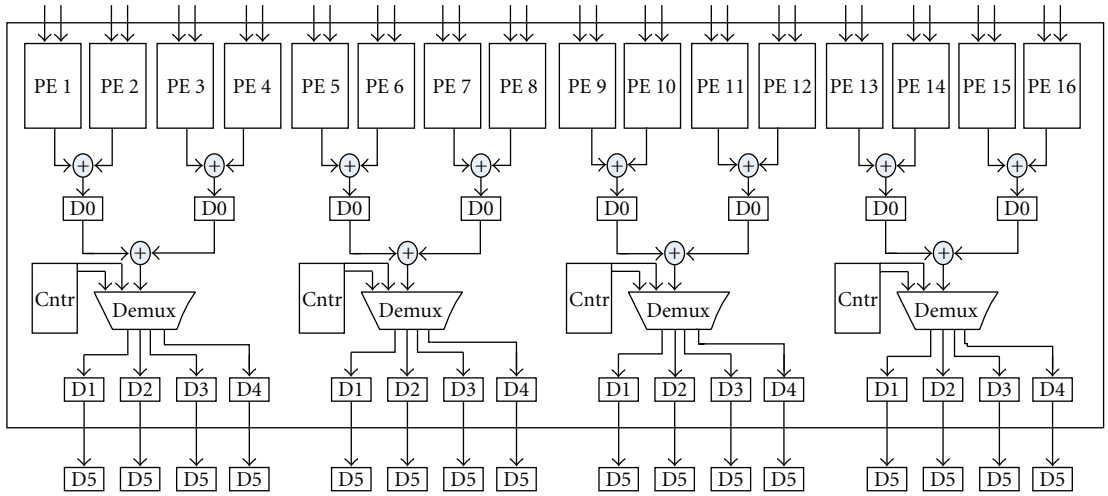

Figure 6: The architecture of a Processing Unit (PU).

48 × 48 pixels, that is 2304 bytes [25]. Thus by using the modified SUMH, we achieve an 11% on-chip memory savings even without a data reuse scheme.

In each clock cycle, we load 64 bits of data. This means that it takes 256 cycles to load data for one search pass and 3840 (256 × 15) cycles to load data for one CMB. Under similar conditions for FSME it would take 288 clock cycles to load data for one CMB. Thus the ratio of the required memory bandwidth for the modified SUMH to the required memory bandwidth for FSME is 13.3. While this ratio is undesirably high, it is well mitigated by the fact that there are only 113 search locations for one CMB in the modified SUMH, compared to 1089 search locations for one CMB in FSME. In other words, the amount of computation for one CMB in the modified SUMH is approximately 0.1 that for FSME. Thus there is an overall power savings in using the modified SUMH instead of FSME.

3.3. Processing Unit. Table 8 shows the pixel data schedule for two search passes of the N PUs. In Table 8 we are considering as an illustrative example the cross search and a search range $s = \pm 16$, hence the given pixel coordinates.

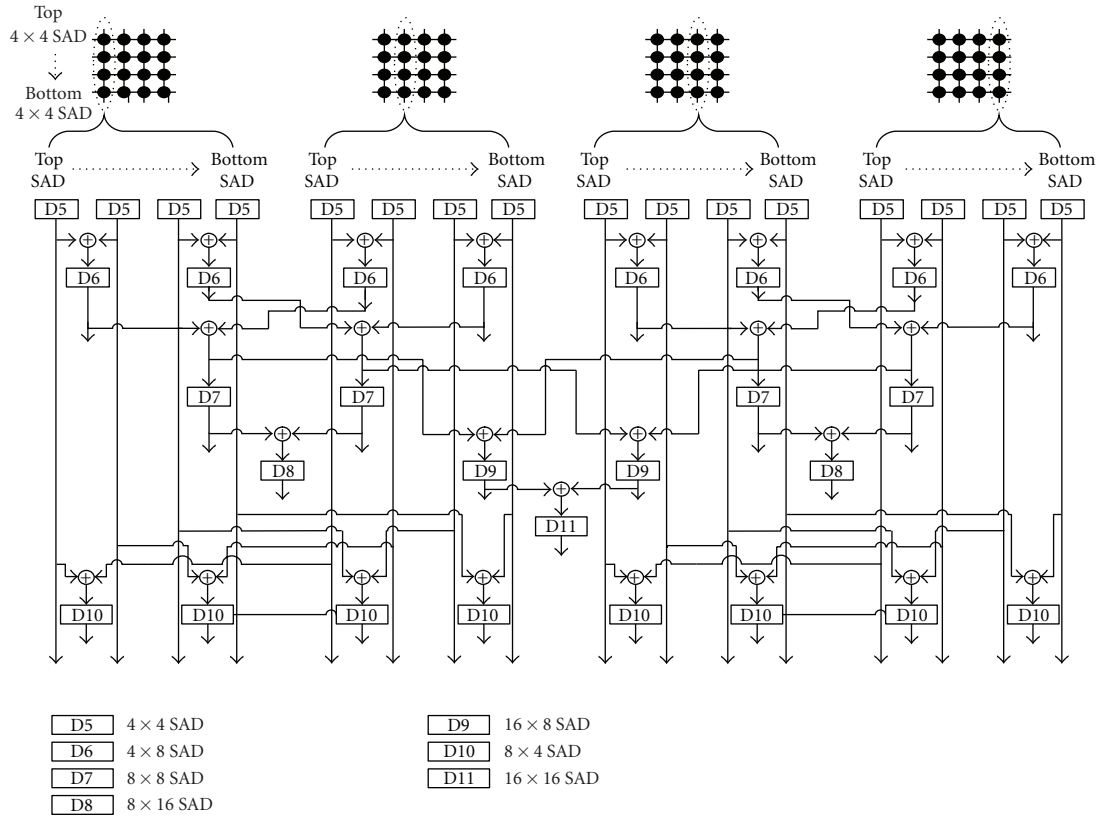

Figure 7: SAD Combination tree.

Table 8: Data schedule for processing unit (PU).

Clock	PU1	···	PU8	Comments
1–16	(−15, 0)–(0,0) ⋮ (−15, −15)–(0, −15)	···	(−1,0)–(14,0) ⋮ (−1, −15)–(14, −15)	Search pass 1: left horizontal scan of cross search
17– 32	(1, 0)–(16,0) ⋮ (1, −15)–(16, −15)	···	(15,0)–(30,0) ⋮ (15, −15)–(30, −15)	Search pass 2: right horizontal scan of cross search
33–48	(0, 15)–(15,15) ⋮ (0, 0)–(15, 0)	···	(0, 1)–(15, 1) ⋮ (0, −14)–(15, −14)	Search pass 3: top vertical scan of cross search
49–64	(0, −1)–(15, −1) ⋮ (0, −16)–(15, −16)	···	(0, −15)–(15, −15) ⋮ (0, −30)–(15, −30)	Search pass 4: bottom vertical scan of cross search
⋮	⋮		⋮	⋮

Table 8 shows that it takes 16 cycles to output the 16 4×4 SADs from each PU.

3.4. SAD Combination Tree. The data schedule for the SAD combination is shown in Table 9. There are N SAD combination (SC) trees, each processing 16 4×4 SADs that are output from each PU. It takes 5 cycles to combine the 16 4×4 SADs and output 41 SADs for the 7 interprediction block sizes in H.264/AVC: 1 16×16 SAD, 2 16×8 SADs, 2 8×16 SADs, 4 8×8 SADs, 8 8×4 SADs, 8 4×8 SADs, and 16 4×4 SADs.

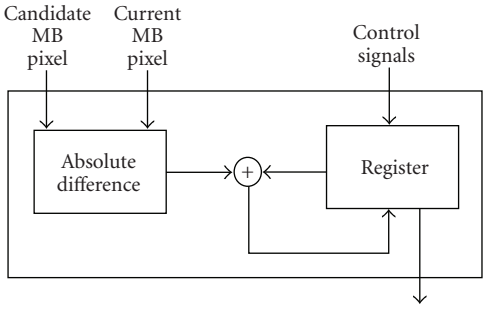

Figure 8: Processing element (PE).

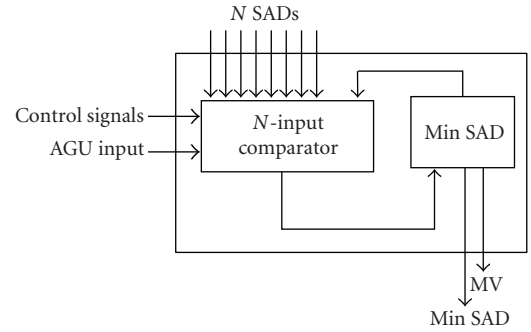

Figure 9: Comparing element (CE).

Table 9: Data schedule for SAD combination (SC) unit.

Clock	SC1	SC2	· · ·	SC8
17	16 4 × 4 SAD	16 4 × 4 SAD		16 4 × 4 SAD
18	8 4 × 8 SAD	8 4 × 8 SAD		8 4 × 8 SAD
	8 8 × 4 SAD	8 8 × 4 SAD		8 8 × 4 SAD
19	4 8 × 8 SAD	4 8 × 8 SAD		4 8 × 8 SAD
20	2 8 × 16 SAD	2 8 × 16 SAD		2 8 × 16 SAD
	2 16 × 8 SAD	2 16 × 8 SAD		2 16 × 8 SAD
21	1 16 × 16 SAD	1 16 × 16 SAD		1 16 × 16 SAD

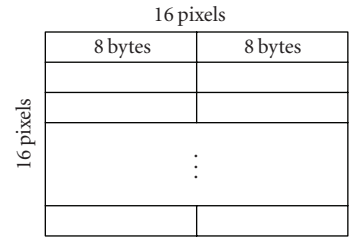

Figure 10: Data arrangement in current macroblock (CMB) memory.

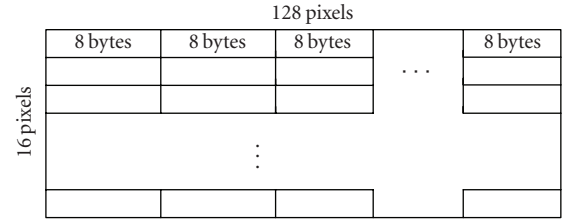

Figure 11: Data arrangement in search window (SW) memory.

3.5. Comparison Unit. The data schedule for the CU is shown in Table 10. The CU consists of 41 CE, each element processing N SADs of the same interprediction block size, from the N PUs. Each CE compares SADs in twos. It therefore takes $\log_2 N + 1$ cycles to output the 41 minimum SADs. Thus given $N = 8$, the CU consumes 4 cycles.

3.6. Summary of Dataflow. The dataflow represented by the data schedules described variously in Tables 8–10 may be summarized by the algorithmic state machine (ASM) chart shown in Figure 12. The ASM chart also represents the mapping of the modified SUMH algorithm in Figure 2, to our proposed architecture in Figure 5.

In our ASM chart, there are 6 states and 2 decision boxes. The states are labeled S1 to S6, while the decision boxes are labeled Q1 and Q2. In each state box, we provide the summary description of the state as well as its output variables in italic font.

From Figure 12 we see that implementation of the modified SUMH on our proposed architecture IP core starts in state S1 when the motion vector (MV) predictors are checked. This is done by the PowerPC processor which is part of our SoC prototyping platform (see Section 4). The MV predictors are stored in external memory and accessed from there by the PowerPC processor. The output from state S1 is the MV predictors. In the next state S2, the minimum MV cost is obtained and mode decision is done to obtain the right blocktype. This is also done by the PowerPC processor and the outputs of this state are the minimum MV, its SAD cost, its blocktype, and its address. The minimum MV cost is obtained by minimizing the cost in

$$J_{\text{motion}}(\vec{m}, \text{REF} \mid \lambda_{\text{motion}}) = \text{SAD}(dx, dy, \text{REF}, \vec{m}) + \lambda_{\text{motion}} \cdot \left(R\left(\vec{m} - \vec{p}\right) + R(\text{REF})\right), \quad (8)$$

where $\vec{m} = (m_x, m_y)^{\text{T}}$ is the current MV being considered, REF denotes the reference picture, λ_{motion} is the Lagrangian multiplier, $\text{SAD}(dx, dy, \text{REF}, \vec{m})$ is the SAD cost obtained as in (1), $\vec{p} = (p_x, p_y)$ is the MV used for the prediction, $R(\vec{m} - \vec{p})$ represents the number of bits used for MV coding, and, $R(\text{REF})$ is the bits for coding REF.

In the state S3, some of the outputs from state S2 are passed into our proposed architecture IP core. In state S4, the AGU computes the addresses of candidate blocks, using the address of the MV predictor as the base address, and the control unit waits for the initialization of search window data in the BRAMs. The output of state S4 is

Table 10: Data schedule for comparison unit (CU).

Clock	CE1–CE16	CE17–CE32	CE33–CE36	CE37–CE40	CE41
22	8 4 × 4 SAD	8 4 × 8 SAD 8 8 × 4 SAD	8 8 × 8 SAD	8 8 × 16 SAD 8 16 × 8 SAD	8 16 × 16 SAD
23	4 4 × 4 SAD	4 4 × 8 SAD 4 8 × 4 SAD	4 8 × 8 SAD	4 8 × 16 SAD 4 16 × 8 SAD	4 16 × 16 SAD
24	2 4 × 4 SAD	2 4 × 8 SAD 2 8 × 4 SAD	2 8 × 8 SAD	2 8 × 16 SAD 2 16 × 8 SAD	2 16 × 16 SAD
25	1 4 × 4 SAD	1 4 × 8 SAD 1 8 × 4 SAD	1 8 × 8 SAD	1 8 × 16 SAD 1 16 × 8 SAD	1 16 × 16 SAD

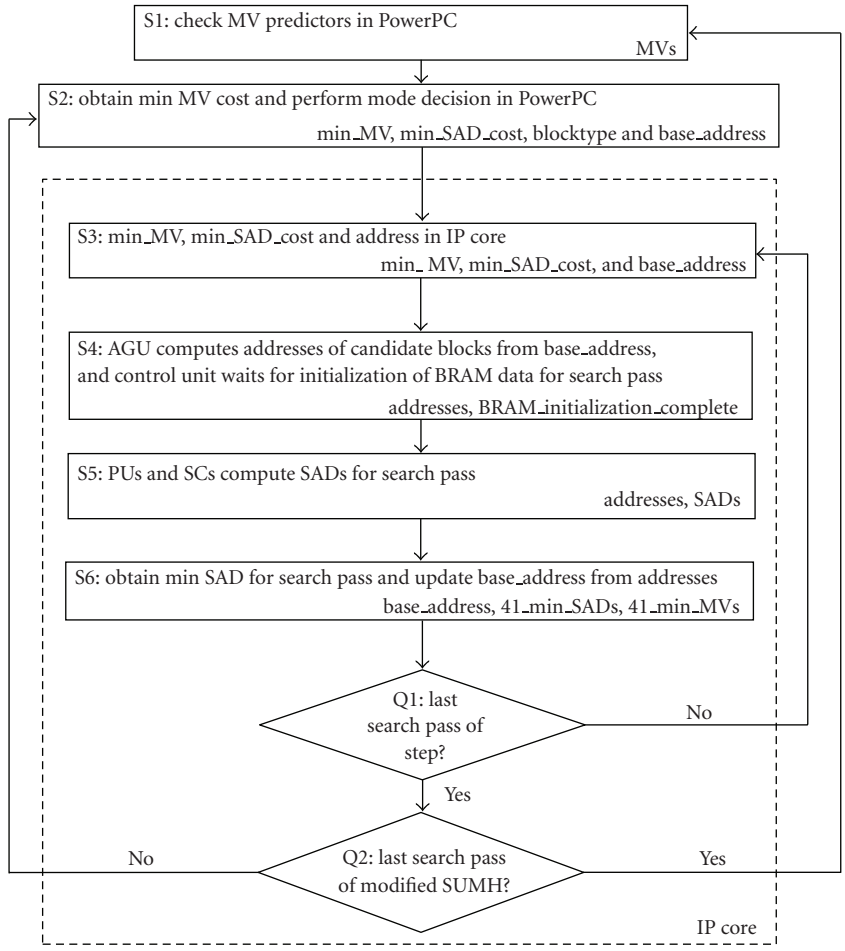

Figure 12: Algorithmic state machine chart for the modified SUMH algorithm.

the addresses of the candidate blocks and a flag indicating that BRAM initialization is complete. In state S5, the processing units and SAD combination trees compute the SADs of the candidate blocks. The output of S5 is the computed SADs and unchanged AGU addresses. In state S6, the CU compares these SADs with previously computed SADs and obtains the 41 minimum SADs. The outputs of S6 are the 41 minimum SADs and their corresponding addresses.

In the decision block Q1, we check if the current search pass is the last search pass of a particular search step, for example, the cross search step. If no, we continue with other passes of that search step. If yes, we go to decision block Q2. In Q2 we check if it is the last search pass of the modified SUMH algorithm. If no, we move onto the next search step, for example, hexagon search. If yes, we check for the MV predictors of the next current macroblock, according to the HF3V2 2-stitched zigzag scan proposed in [19].

Table 11: Synthesis results.

Process (μm)	0.13 (FPGA)
Number of slices	11.4K
Number of slice flip flops	16.4K
Number of 4-input LUTs	18.7K
Total equivalent gate count	388K
Max frequency (MHz)	145.2
Algorithm	Modified SUMH
Video specifications	CIF 30-fps
Search range	± 16
Block size	16×16 to 4×4
Minimum required frequency (MHz)	24.1
Number of 16×8-bit dual-port RAMs	129
Memory utilization (Kb)	398
Voltage (V)	1.5
Power consumed (mW)	25

3.7. Synthesis Results and Analysis. The proposed architecture has been implemented in Verilog HDL. Simulation and functional verification of the architecture was done using the Mentor Graphics ModelSim tool [26]. We then synthesized the architecture using the Xilinx sythesis tool (XST). XST is part of the Xilinx integrated software environment (ISE) [27]. After synthesis, place and routing is done targeting the Virtex-II Pro XC2VP30 Xilinx FPGA on our development board. Finally we obtain power analysis for our design, using the XPower tool which is also part of Xilinx ISE.

Our synthesis results are shown in Table 11. From Table 11 we see that our architecture can achieve a maximum frequency of 145.2 MHz. The FPGA power consumption of our architecture is 25 mW obtained using Xilinx XPower tool. The total equivalent gate count is 388 K.

Our simulations in ModelSim support our dataflow described in Sections 3.1 to 3.6. We find that it takes 27 cycles to obtain the minimum SAD from each search pass, after initialization. The 27 cycles are obtained from 1 cycle for the AGU, 1 cycle to read data from on-chip memory, 16 cycles for the PU, 5 cycles for the SAD combination tree, and 4 cycles for the comparison unit. Therefore, it takes 405 (15×27) cycles to complete the search for 1 CMB, 1 reference frame, and $s = \pm 16$. For a CIF image (396 MBs) at 30 Hz and considering 5 reference frames, a minimum clock frequency of approximately 24.1 ($405 \times 396 \times 30 \times 5$) MHz is required. Thus with a maximum possible clock speed of 145.2 MHz, our architecture can compute in real-time CIF sequences within a search range of ± 16 and using 5 reference frames.

We provide Table 12 which compares our architecture with previous state-of-the-art architectures implemented on ASICs. Note that a direct comparison of our implementation with implementations done on ASIC technology is impossible because of the fact that the platforms are different. ASICs still provide the highest performance in terms of area, power consumed, and maximum frequency. However, we provide Table 12 not for direct comparisons, but to show that our implementation achieves ASIC-like levels of performance. This is desirable because it indicates that an ASIC implementation of our architecture will yield even better performance results. Our Verilog implementation was kept portable in order to simplify FPGA to ASIC migration.

From Table 12 we see that our architecture achieves many desirable results. The most remarkable is that the power consumption is very low despite the fact that our implementation is done on an FPGA which typically consumes more power than an ASIC. Besides the low power consumption of our architecture, other favorable results are that the algorithm we use has better PSNR performance than the algorithms used in the other works. We also note that our architecture achieves the highest maximum frequency. By extension our architecture is the only one that can support high definition (HD) 1080 p sequences at 30 Hz, a search range $s = \pm 16$ and 1 reference frame. This would need a minimum frequency of approximately 85.9 MHz.
In the next section we discuss our prototyping efforts and compare our results with similar works.

4. Architecture Prototype

The top-level prototype design of our architecture is shown in Figure 13. It is based on the prototype design in [25]. In [25], Canals et al. propose an FPSoC-based architecture for Full Search block matching algorithm. Their implementation is done on a Virtex-4 FPGA.

Our prototype is done on the XUPV2P development board available from Digilent Inc. [28]. The board contains a Virtex-II Pro XC2VP30 FPGA with 30,816 Logic Cells, 136 18-bit multipliers, 2,448 Kb of block RAM, and two PowerPC Processors. There are several connectors which include a serial RS-232 port for communication with a host personal computer. The board also features JTAG programming via on-board USB2 port as well as a DDR SDRAM DIMM that can accept up to 2 Gbytes of RAM.

The embedded development tool used to design our prototype is the Xilinx Platform Studio (XPS), in the Xilinx Embedded Development Kit (EDK) [29]. The EDK makes it relatively simple to integrate user Intellectual Property (IP) cores as peripherals in an FPSoC. Hardware/software cosimulation can then be done to test the user IP.

In our prototype design, as shown in Figure 13, we employ a PowerPC hardcore embedded processor, as our controller. The processor sends stimuli to the motion estimation IP core and reads results back for comparison. The processor is connected to the other design modules, via a 64bit processor local bus (PLB).

The boot program memory is a 64 kb BRAM. It contains a bootloop program necessary to keep the processor in a known state after we load the hardware and before we load the software. The PLB connects to the user IP core through an IP interface (IPIF). This interface exposes several programmable interconnects. We use a slave-master FIFO attachment that is 64-bits wide and 512 positions deep. The status and control signals of the FIFO are available to the user logic block. The user logic block contains logic for reading

Table 12: Comparison with other architectures implemented on ASICS.

	Chao's et al. [11]	Miyakoshi's et al. [12]	Lin's [13]	Chen's et al. [14]	This Work
Process (μm)	0.35	0.18	0.18	0.18	0.13 FPGA
Voltage (V)	3.3	1.0	1.8	1.3	1.5
Transistors count	301 K	1000 K	546 K	708 K	388 K
Maximum frequency (MHz)	50	13.5	48.67	66	145.2
Video Spec.	CIF 30-fps	CIF 30-fps	CIF 30-fps	CIF 30-fps	CIF 30-fps
frequency (MHz)	50	13.5	48.67	13.5	24.1
Algorithm	Diamond search	Gradient decent	4SS	Single-Iteration Parallel VBS 4SS w/1-ref.	Hardware oriented SUMH
Block size	16×16 and 8×8	16 × 16 and 8 × 8	16 × 16	16 × 16 to 4 × 4	16 × 16 to 4 × 4
power (mW)	223.6	6.56	8.46	2.13	25
Normalized Power (1.8 V, 0.18 μm)*	17.60	21.25	8.46	4.08	69.02
Architecture	1D tree. No data reuse scheme	1D tree. No data reuse scheme	1D tree. Level A data reuse scheme	2D tree. Level B data reuse scheme	1D tree. No data reuse scheme
Can support HD1920 × 1080 p	No	No	No	No	Yes

* Normalized power = Power × (0.18^2/process^2) × (1.8^2/voltage^2).

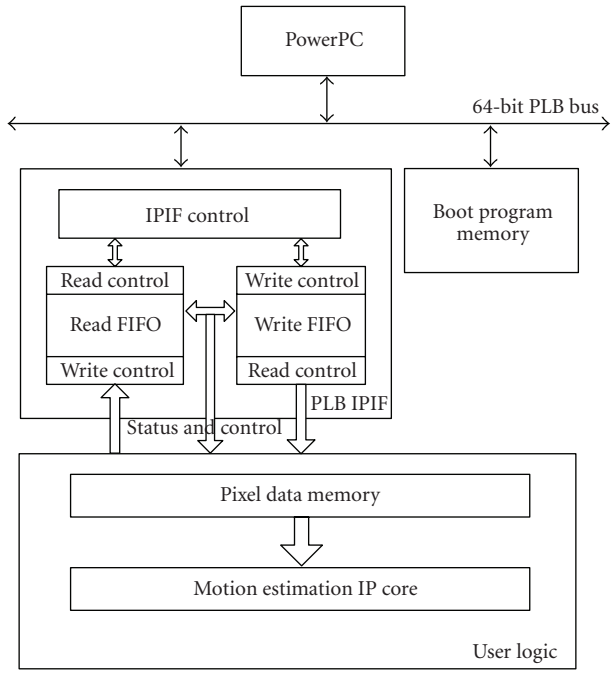

Figure 13: FPSoC prototype design of our architecture.

and writing to the FIFO and the Verilog implementation of our architecture.

During operation, the PowerPC processor writes input stimuli to the FIFO and sets status and control bits. The user logic reads the status and control signals and when appropriate, reads data from the FIFO. The data passes into the IP core and when the ME computation is done, the results are written back on the FIFO. The PowerPC reads the results and does a comparison with expected results to verify accuracy of the IP. Intermediate results during the operation are sent to a terminal on the host personal computer, via the RS-232 serial connection.

Table 13: Comparison with other FPSOC architectures.

	Canals et al. [25]	This work
FPSoC FPGA	Virtex-4	Virtex-II Pro
Algorithm	Full Search	Hardware oriented SUMH
Video format	QCIF	QCIF
Search range	±16	±16
Number of slices	12.5 K	11.4 K
Memory utilization (Kb)	784	398
Clock frequency (MHz)	100	100

We target QCIF video for our prototype, in order to compare our results with the results in [25]. Table 13 shows this comparison. We see from Table 13 that our architecture consumes less FPGA resources and has a lower memory utilization. Again, we note that a direct comparison of both architectures is complicated by the fact that different FPGAs

were used in both prototyping platforms. The work in [25] is based on a Virtex-4 FPGA which uses 90-nm technology, while our work is based on Virtex-II Pro FPGA which uses 130-nm technology.

5. Conclusion

In this paper we have presented our low power, FPSoC-based architecture for a fast ME algorithm in H.264/AVC. We described our adopted fast ME algorithm which is a hardware oriented SUMH algorithm. We showed that the modified SUMH has superior rate-distortion performance compared to some existing state-of-the-art fast ME algorithms. We also described our architecture for the hardware oriented SUMH. We showed that the FPGA-based implementation of our architecture yields ASIC-like levels of performance in terms of speed, area, and power. Our results showed in addition, that our architecture has the potential to support HD 1080 p unlike the other architectures we compared it with. Finally we have discussed our prototyping efforts and compared them with a similar prototyping effort. Our results showed that our implementation uses less FPGA resources.

In summary therefore, the modified SUMH is more attractive than SUMH because it is hardware oriented. It is also more attractive than Full Search because Full Search is hardware oriented, it is much more complex than the modified SUMH and thus will require more hardware area, speed, and power for implementation.

We therefore conclude that for low power handheld devices, the modified SUMH can be used without much penalty, instead of Full Search, for ME in H.264/AVC.

Acknowledgments

The authors acknowledge the support from Xilinx Inc., the Xilinx University Program, the Packard Foundation and the Department of Electrical Engineering, Santa Clara University, California. The authors also thank the editor and Reviewers of this journal for their useful comments.

References

[1] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra, "Overview of the H.264/AVC video coding standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 560–576, 2003.

[2] G. J. Sullivan, P. Topiwala, and A. Luthra, "The H.264/AVC advanced video coding standard: overview and introduction to the fidelity range extensions," in *Proceedings of the 27th Conference on Applications of Digital Image Processing*, vol. 5558 of *Proceedings of SPIE*, pp. 454–474, August 2004.

[3] H.-C. Lin, Y.-J. Wang, K.-T. Cheng, et al., "Algorithms and DSP implementation of H.264/AVC," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '06)*, pp. 742–749, Yokohama, Japan, January 2006.

[4] Z. Chen, P. Zhou, and Y. He, "Fast integer pel and fractional pel motion estimation for JVT," in *Proceedings of the 6th Meeting of the Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCE*, Awaji Island, Japan, December 2002, JVT-F017.

[5] R. Li, B. Zeng, and M. L. Liou, "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, 1994.

[6] L.-M. Po and W.-C. Ma, "A novel four-step search algorithm for fast block motion estimation," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 6, no. 3, pp. 313–317, 1996.

[7] J. Y. Tham, S. Ranganath, M. Ranganath, and A. A. Kassim, "A novel unrestricted center-biased diamond search algorithm for block motion estimation," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 8, no. 4, pp. 369–377, 1998.

[8] X. Yi, J. Zhang, N. Ling, and W. Shang, "Improved and simplified fast motion estimation for JM," in *Proceedings of the 16th Meeting of the Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG*, Posnan, Poland, July 2005, JVT-P021.doc.

[9] X. Yi and N. Ling, "Improved normalized partial distortion search with dual-halfway-stop for rapid block motion estimation," *IEEE Transactions on Multimedia*, vol. 9, no. 5, pp. 995–1003, 2007.

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

[11] W.-M. Chao, C.-W. Hsu, Y.-C. Chang, and L.-G. Chen, "A novel hybrid motion estimator supporting diamond search and fast full search," in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS '02)*, vol. 2, pp. 492–495, Phoenix, Ariz, USA, May 2002.

[12] J. Miyakoshi, Y. Kuroda, M. Miyama, K. Imamura, H. Hashimoto, and M. Yoshimoto, "A sub-mW MPEG-4 motion estimation processor core for mobile video application," in *Proceedings of the Custom Integrated Circuits Conference (ICC '03)*, pp. 181–184, 2003.

[13] S.-S. Lin, *Low-power motion estimation processors for mobile video application*, M.S. thesis, Graduate Institute of Electronic Engineering, National Taiwan University, Taipei, Taiwan, 2004.

[14] T.-C. Chen, Y.-H. Chen, S.-F. Tsai, S.-Y. Chien, and L.-G. Chen, "Fast algorithm and architecture design of low-power integer motion estimation for H.264/AVC," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 17, no. 5, pp. 568–576, 2007.

[15] L. Zhang and W. Gao, "Reusable architecture and complexity-controllable algorithm for the integer/fractional motion estimation of H.264," *IEEE Transactions on Consumer Electronics*, vol. 53, no. 2, pp. 749–756, 2007.

[16] C. A. Rahman and W. Badawy, "UMHexagonS algorithm based motion estimation architecture for H.264/AVC," in *Proceedings of the 5th International Workshop on System-on-Chip for Real-Time Applications (IWSOC '05)*, pp. 207–210, Banff, Alberta, Canada, 2005.

[17] M.-S. Byeon, Y.-M. Shin, and Y.-B. Cho, "Hardware architecture for fast motion estimation in H.264/AVC video coding," in *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E89-A, no. 6, pp. 1744–1745, 2006.

[18] Y.-Y. Wang, Y.-T. Peng, and C.-J. Tsai, "VLSI architecture design of motion estimator and in-loop filter for MPEG-4 AVC/H.264 encoders," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS '04)*, vol. 2, pp. 49–52, Vancouver, Canada, May 2004.

[19] C.-Y. Chen, C.-T. Huang, Y.-H. Chen, and L.-G. Chen, "Level C+ data reuse scheme for motion estimation with corresponding coding orders," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 16, no. 4, pp. 553–558, 2006.

[20] S. Yalcin, H. F. Ates, and I. Hamzaoglu, "A high performance hardware architecture for an SAD reuse based hierarchical motion estimation algorithm for H.264 video coding," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '05)*, pp. 509–514, Tampere, Finland, August 2005.

[21] S.-J. Lee, C.-G. Kim, and S.-D. Kim, "A pipelined hardware architecture for motion estimation of H.264/AVC," in *Proceedings of the 10th Asia-Pacific Conference on Advances in Computer Systems Architecture (ACSAC '05)*, vol. 3740 of *Lecture Notes in Computer Science*, pp. 79–89, Springer, Singapore, October 2005.

[22] C.-M. Ou, C.-F. Le, and W.-J. Hwang, "An efficient VLSI architecture for H.264 variable block size motion estimation," *IEEE Transactions on Consumer Electronics*, vol. 51, no. 4, pp. 1291–1299, 2005.

[23] O. Ndili and T. Ogunfunmi, "A hardware oriented integer pel fast motion estimation algorithm in H.264/AVC," in *Proceedings of the IEEE/ECSI/EURASIP Conference on Design and Architectures for Signal and Image Processing (DASIP '08)*, Bruxelles, Belgium, November 2008.

[24] H.264/AVC Reference Software JM 13.2., 2009, http://iphome.hhi.de/suehring/tml/download.

[25] J. A. Canals, M. A. Martínez, F. J. Ballester, and A. Mora, "New FPSoC-based architecture for efficient FSBM motion estimation processing in video standards," in *Proceedings of the International Society for Optical Engineering*, vol. 6590 of *Proceedings of SPIE*, p. 65901N, 2007.

[26] Mentor Graphics ModelSim SE User's Manual—Software Version 6.2d, 2009, http://www.model.com/support.

[27] Xilinx ISE 9.1 In-Depth Tutorial, 2009, http://download.xilinx.com/direct/ise9_tutorials/ise9tut.pdf.

[28] Xilinx Virtex-II Pro Development System, 2009, http://www.digilentinc.com/Products/Detail.cfm?Prod=XUPV2P.

[29] Xilinx Platform Studio and Embedded Development Kit, 2009, http://www.xilinx.com/ise/embedded/edk_pstudio.htm.