Parallel H.264/AVC Fast Rate-Distortion Optimized Motion Estimation using Graphics Processing Unit and Dedicated Hardware

(Article begins on next page)

16 August 2022

Parallel H.264/AVC Fast Rate-Distortion Optimized Motion Estimation using Graphics Processing Unit and Dedicated Hardware

Muhammad Usman Shahid, Ashfaq Ahmed, Maurizio Martina, Guido Masera,
and Enrico Magli

Abstract—Heterogeneous systems on a single chip composed of CPU, Graphical Processing Unit (GPU), and Field Programmable Gate Array (FPGA) are expected to emerge in near future. In this context, the System on Chip (SoC) can be dynamically adapted to employ different architectures for execution of data-intensive applications. Motion estimation is one such task that can be accelerated using FPGA and GPU for high performance H.264/AVC encoder implementation. In most of works on parallel implementation of motion estimation, the bit rate cost of motion vectors is generally ignored. On the contrary, this paper presents a fast rate-distortion optimized parallel motion estimation algorithm implemented on GPU using OpenCL and FPGA/ASIC using VHDL. The predicted motion vectors are estimated from temporally preceding motion vectors and used for evaluating the bit rate cost of the motion vectors simultaneously. The experimental results show that the proposed scheme achieves significant speedup on GPU and FPGA, and has comparable rate-distortion performance with respect to sequential fast motion estimation algorithm.

Index Terms—H.264/AVC, Parallel Fast Motion Estimation, GPU, OpenCL, FPGA.

I. INTRODUCTION

H.264/AVC [1] is currently the most commonly utilized video coding format because of its high coding efficiency compared to its predecessors. Video coding is achieved by exploiting temporal and spatial redundancies and Motion Estimation (ME) is one of the main tools employed for eliminating temporal redundancies. It is the most critical and time consuming tool in the whole encoder and typically requires 60-80% of the total computational time during the encoding process [2]. Block matching ME algorithms divide a frame into macroblocks and look for the best possible match in the reference frame with minimum Rate Distortion (RD) cost. Practically, the exhaustive full search is too expensive to implement, so fast ME algorithms such as

unsymmetrical multi-hexagon search (UMHexagonalS) [2], simplified UMHexagonalS [3], Diamond search [4] have been proposed to reduce the computation time.

Most fast ME algorithms were proposed in the context of medium resolution video applications, such as VGA and 480p. However, new high resolution applications such as 720p and 1080p recently stimulated the demand for encoders with significantly increased speed. The performance can be increased by either using high performance serial processors or by parallelizing the ME process and using parallel accelerators such as Graphics Processing Units (GPUs) or implementing reconfigurable accelerators on Field Programmable Gate Arrays (FPGAs).

GPU/FPGA accelerators are commonly available as add-in boards, but in near future, Systems on Chip are expected to adopt heterogeneous architectures including CPUs, GPUs and FPGAs on a single platform [5]. This offers an opportunity to utilize different architectures for scheduling jobs based on the available resources at run time. For example, the FPGA can be used in the case when GPU is already engaged and vice versa. Since ME is a computation intensive part of video encoding, different design trade-offs can be foreseen for different scenarios. In this context, as an example, a video encoder implemented on such a system could modify adaptively its throughput, and this capability can be enabled by switching among alternative architectures, based on frame resolution. Processing units can be utilized in an efficient and dynamic way to get better results in terms of processing speed, resource and energy management.

GPUs are fixed-architecture parallel processing devices that offer high memory bandwidth and concurrent programming cores executing programs

in Single Instruction Multiple Data (SIMD) mode. Availability of high level programming frameworks such as OpenCL [6] and CUDA [7] have led to enormous interest in developing general purpose data-intensive applications using GPUs. FPGAs are particularly suited for implementing parallel architectures and can nearly achieve Application Specific Integrated Circuit (ASIC) performance with lower fabrication costs. In the video coding context, GPUs and FPGAs are typically used to implement the most computationally demanding tasks, particularly ME.

Generally, parallel ME implementations overlook the bit rate cost for computing motion vectors (MV) due to the motion vector prediction (MVp) technique. The data dependencies intrinsic to the MVp method restricts macroblock level parallel implementations due to the unavailability of spatially adjacent MVs. To implement block level parallelism, the bit rate cost of the MVs can not be estimated accurately, which leads to degradation in RD performance.

In this work, we present a low complexity fast ME algorithm exploiting block level parallelism. We propose a new approach for removing data dependencies in MVp that will enable full parallelism at block level. To mitigate the data dependency in the calculation of MVp, the proposed approach introduces the concept of Coarse MV (MVc), which are evaluated using MVs from previous frame. MVc are then used as MVp to compute the bit rate cost of MVs. This method of MV computation makes the proposed ME approach highly amenable to parallel implementation, as no further data dependency exists among macroblocks.

In this paper we present a two-fold implementation of the proposed algorithm on the GPU using OpenCL and on the FPGA using VHDL. Our parallel fast ME algorithm exhibits comparable RD performance with respect to sequential fast ME algorithms. The GPU implementation outperforms the state of the art implementations in terms of execution time and achieves higher order of speedup. For FPGA implementation, it is shown that proposed implementation occupies fewer resources in terms of slices and lookup table achieving high frequency compared to state of the art solutions. The proposed design is also synthesized on ASICs and the results show significant improvement in terms of gate count and throughput. Finally, we discuss

target application scenario for future heterogeneous systems.

The remainder of the paper is organized as follows. Section II provides a brief overview of the related work. Details of the proposed ME algorithm and its RD performance are presented in Section III. The GPU and FPGA/ASIC implementations are presented in Section IV, and results are discussed in Section V, which is followed by concluding remarks in Section VI.

II. RELATED WORK

In literature, several GPU and FPGA implementations for fast ME exist and an overview of some of them is provided in this section. Cheung *et al.* present a GPU implementation of simplified UMHexagonalS in [8]. They introduce tiling to solve the data dependency: the frame is divided into tiles, where each tile is executed independently by one GPU thread, and ME is performed sequentially within each tile. The tile size determines the performance of the implementation. Schwalb *et al.* [9] have proposed a macroblock level parallel implementation of Diamond search by completely ignoring MVp. They propose to use predictors generated by Forward Dominant Vector Selection [10] and to use Split and Merge [11] techniques to generate predicted MVs for all blocks. Rodriguez *et al.* [12] have proposed a parallel ME algorithm based on Full search to mitigate the effects of MVp calculation by reusing the corresponding MV from reference frame to adjust their search area for the current frame. Also, they use the co-located temporal MV of each macroblock as MVp in cost calculation to get a better estimate.

In terms of hardware ME acceleration, few of the reported architectures have implemented full search ME [13]–[15], whereas the major contribution is for fast ME algorithm implementation [16]–[25]. The works in [13], [16]–[20] are implemented on FPGA-based platforms, while [14], [15], [21]–[25] are ASIC implementations. In [14], full search is deployed with some speedup techniques, such as pixel bit precision reduction, macroblock subsampling, and adaptive search range adjustment. In [22]–[24], hierarchical search approach is combined with full search. Typically, two [22], [23] or three [24] levels are employed. The architecture in [21] supports Diamond search and Cross search. In [25],

(a) Generic algorithm flow
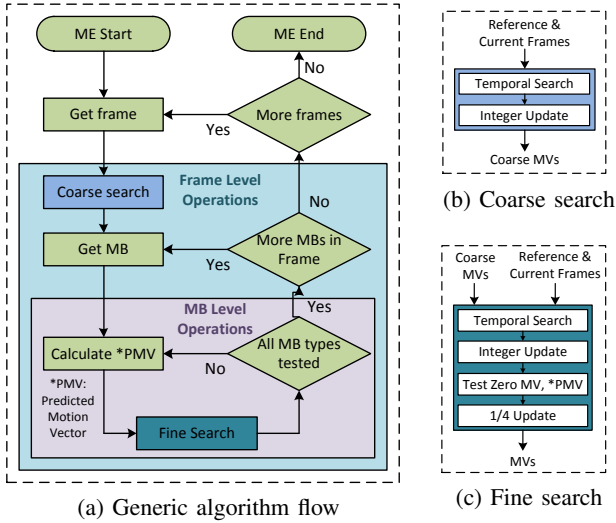
(b) Coarse search

(c) Fine search

Fig. 1. Overall flow diagram representing all the steps of the proposed motion estimation algorithm are shown in (a). Sequential steps required for performing Coarse and Fine search are shown in (b) and (c).

Fig. 2. Temporal search method: for a macroblock located at (x,y) in frame N, 6 temporal predictors referred by corresponding MVs of temporally collocated macroblocks in frame N-1 are tested.

an architecture is presented for adaptive ME using hierarchical and multipath search techniques. The architecture in [15] presents a configurable VLSI architecture for integer ME in H.264. Most of the reported works show the implementations for a particular block size, whereas the proposed architecture is implemented for all block sizes given in H.264/AVC standard.

III. PROPOSED ALGORITHM

An important part of this work is the design of an inherently parallel fast ME algorithm with satisfactory RD performance. In order to fully exploit macroblock level parallelism, and take advantage of MV correlations to reduce the number of operations, we have designed a parallel and low complexity hierarchical-recursive block matching algorithm based on fast ME algorithm [26]. The algorithm is hierarchical, as it consists of two sequential steps, namely Coarse search and Fine search. The Coarse search is performed for each macroblock of each frame in display order and produces a coarse predictor and its corresponding MVc. Fine search further refines the already estimated coarse predictor and attains the best predictor for each block size with respect to each reference frame. Figure 1(a) shows the flow diagram representing the sequence of steps performed for computing MVs based on the proposed algorithm. The algorithm exploits temporal corre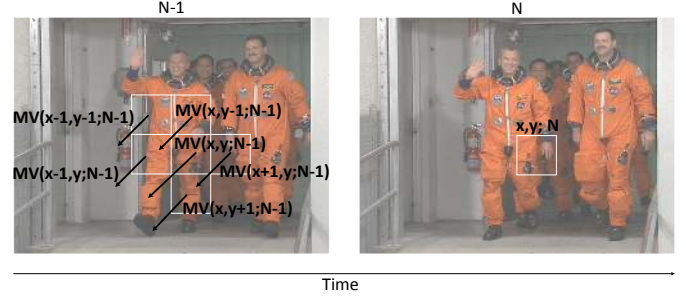lations among frames by using MVs from previous frames to find best temporal prediction for the current frame. The algorithm supports full macroblock level parallelism by eliminating spatial dependencies amid calculating MVp.

A. Coarse Search

The Coarse search obtains an integer pixel MV for each macroblock (16×16 block) of the current image by searching the most similar predictor in the previous frame. Sum of Absolute Difference (SAD) is used as the decision criterion for estimating the best predictor. In order to find this predictor, we perform two consecutive steps (Fig. 1(b)):

1) We examine six MVs belonging to six temporally adjacent macroblocks (Temporal Search).
2) We add to the best of the temporal predictors, a grid of 12 points called Updates.

1) Temporal Search: The Temporal Search is based on the idea that motion typically does not change drastically between two consecutive frames. So in order to find the best predictor for the current macroblock, we check if it has the same MV as the temporally collocated surrounding blocks. For a generic macroblock located at coordinates (x, y) in the frame N, we test six temporal predictors in the frame $N-1$ and select the one with least SAD value. The candidate predictors are pointed to by the MVs $MV(x, y; N-1), MV(x+1, y; N-1), MV(x, y+1; N-1), MV(x-1, y; N-1), MV(x, y-1; N-1), MV(x, y+1; N-1)$ as shown in Fig. 2, where *e.g.* $MV(x, y; N-1)$ is MV of macroblock in frame $N-1$, at coordinates (x, y) pointing to the corresponding predictor macroblock in frame $N-1$.

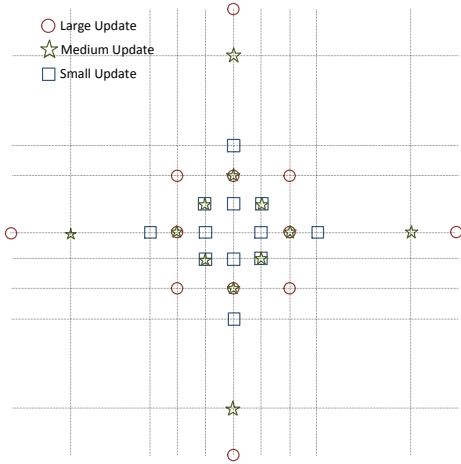*2) Updates:* After the Temporal Search, a grid of 12 points referred to as Updates is added to get

Fig. 3. Integer Update grid of 12 points consisting of three different sets of positions represented by different shapes.

TABLE I
SAD VS. RD OPTIMIZATION COMPARISON (CIF SEQUENCES).

Sequence	$\triangle$PSNR (db)	$\triangle$BR (%)
Football	0.4599	-7.2654
Flower	0.3421	-4.6125
America	0.8287	-25.6052
Silent	0.5082	-9.7733
Stefan	0.4089	-7.0375
Coastgaurd	0.3015	-6.2135
Foreman	0.4966	-11.1326
Mobile	0.3476	-5.4540
Average	**0.50**	**-9.64**

the best integer predictor. The Updates have fixed relative positions, and there are three different sets of updates that can be used, as shown in Fig. 3. One set of Updates (small, medium, large) is applied for each macroblock depending on the value of the SAD of the best predictor from the Temporal Search step. After considering all the grid points, the vector with smallest SAD becomes the MVc for current macroblock. The best predictor with respect to the current macroblock is found by exploring the MV field set up by the Updates.

B. Fine Search

The coarse predictors and MVc are improved in order to find the best prediction for each macroblock during the Fine search, as shown in Fig. 1(c). It must be noted that while Coarse search is performed only on 16×16 pixel sized blocks, Fine search is performed for every possible block size and for each reference frame. RD optimization combined with SAD is employed to select the best possible predictor for the encoding block. The cost for each candidate predictor is evaluated as,

$$J = SAD + \lambda \cdot R \cdot (\text{MV} - \text{MV}_\text{p}) \qquad (1)$$

where J is the cost, λ is the Lagrangian multiplier. The term $R \cdot (\text{MV} - \text{MV}_\text{p})$ represents the bit rate required to encode the difference between the candidate MV and the MVp. Fine search performs Temporal search (Sec. III-A1) and Updates (Sec. III-A2) to find the best integer pixel predictor. Temporal search in this case is triggered by the MVc computed earlier and tests the temporal predictors

surrounding the coarse predictor. Then, MV field generated by the Updates is applied to the output of Temporal search to compute the best integer predictor for each block size. Zero MV is also tested before applying a quarter-pixel Update grid to the best of the integer pixel MV. The predictor with minimum cost and its corresponding quarter-pixel MV is selected as the final output of the algorithm based on (1).

C. Parallel MVp Modification

The proposed ME algorithm exhibits data dependency, specifically the MV_p term in (1) represents a spatial dependency. In the H.264/AVC standard MVp are computed using MVs of neighbouring macroblocks, which restricts fully parallel encoding at block level. We mitigate this problem by exploiting the hierarchical nature of our algorithm. We compute MVc using SAD as decision criterion first, and then perform RD optimization by substituting already available MVc as MVp for the cost function calculation shown in (1). Using this approach, we remove any data dependency between adjacent blocks of the encoding frame and can implement the proposed algorithm exploiting parallelism at different levels best suited to different architectures.

D. Rate-Distortion Performance Evaluation

The proposed algorithm has been developed and tested in a commercial-grade software model compliant with JM 17.2 implementation of the H.264/AVC encoder. For performance evaluation, multiple resolution sequences *i.e.*, CIF, VGA, 576p, 720p and 1080p, are encoded using a Group Of Pictures (GOP) pattern composed of one I frame followed by eleven P frames and the Quantization

TABLE II
RD PERFORMANCE COMPARISON BETWEEN PROPOSED ALGORITHM AND OTHER POPULAR FAST SEARCH ALGORITHMS. FOR EACH
CASE, FULL SEARCH METHOD IS ASSUMED AS THE REFERENCE ALGORITHM IN THE COMPARISONS.

Sequence	Proposed Scheme		UMHexagonal Search		SUMHexagonal Search	
	ΔPSNR(dB)	ΔBR(%)	ΔPSNR(dB)	ΔBR(%)	ΔPSNR(dB)	ΔBR (%)
Football	-0.1248	2.2407	-0.3215	5.4504	-0.0876	1.4638
Miss-america	-0.0228	0.8454	-0.0547	1.9962	-0.0365	1.3467
Silent	-0.0701	1.4418	-0.0868	1.7671	-0.0474	0.968
News	-0.0417	0.6823	-0.0918	1.5098	-0.0647	1.0674
Harbour	-0.0097	0.2288	-0.0661	1.4498	-0.0065	0.1442
Crew	-0.0716	1.9791	-0.1972	5.8054	-0.0573	1.6666
Stockholm	-0.0228	0.5465	-0.0657	1.6645	-0.043	1.0844
Ballroom	-0.0821	1.8821	-0.1073	2.5729	-0.0784	1.9141
Exit	-0.0829	3.2022	-0.1163	5.0131	-0.0813	3.5541
Mobcal	-0.0063	0.1912	-0.046	1.5213	0.0031	-0.102
Shields	-0.0115	0.4123	-0.0505	1.9344	-0.0125	0.495
Ducks take off	-0.0053	0.117	-0.0275	0.4892	-0.0091	0.1607
Parkrun	-0.0021	0.0453	-0.0255	0.5175	-0.0093	0.1895
Blue-sky	-0.0432	0.9641	-0.2917	7.3461	-0.0362	0.8711
Rushhour	-0.0271	0.5696	-0.3195	13.3872	-0.086	3.4167
Station2	-0.0275	0.7151	-0.4883	15.7194	-0.0233	0.6809
Terrace	-0.0345	1.5089	-0.1773	8.378	-0.0522	2.4013
Tennis	-0.0084	1.4963	-0.5426	17.7237	-0.1418	4.3162
Crowd run	-0.0729	1.5983	-0.1942	4.3624	-0.0585	1.3006
Average	**-0.040**	**1.087**	**-0.172**	**5.189**	**-0.048**	**1.417**

Parameter (QP) is varied among 25, 28, 31, 33, and 35. The search range is set to 32, and the number of reference frames is set to 1. Video sequences with different characteristics, like high motion, low motion, camera zooming, camera panning, and so on are used to test the algorithm for different scenarios.

We performed a preliminary experiment using UMHexagonal search algorithm [2] implemented in JM 17.2 software model to report the gain in performance when using RD optimised ME. As discussed in Section I, most parallel ME implementations just use SAD as the deciding metric instead of RD cost for choosing the best MV. Table I presents the gains achieved with RD optimisation enabled using average Peak Signal-to-Noise Ratio (PSNR) degradation and the average increase in bit rate measured by Bjontegaard Delta PSNR, denoted as ΔPSNR and Bjontegaard Delta bit rate, denoted as ΔBR, for different types of sequences [27]. It is shown that enabling RD optimisation greatly improved the coding efficiency with average PSNR improvement of 0.5 dB and significant bit rate reduction of approximately 10%. This shows that the proposed algorithm achieves a significant performance gain with respect to implementations based only on the SAD.

1) Serial MVp performance: Table II reports the performance evaluation of standard MVp method

and the results are compared with some some fast search algorithms. We have used Full search as the reference algorithm, *i.e.*, the performance is measured by comparing Full search implementation with i) our proposed fast search algorithm, ii) UMHexagonal Search [2] and iii) Simplified UMHexagonal search [3]. UMHexagonal search and Simplified UMHexagonal search algorithms are chosen for comparison because they are popular fast search algorithms implemented in H.264/AVC test models and several adaptive hardware as well as GPU implementations of these algorithms are reported in literature.

Table II shows that the proposed algorithm performs remarkably better than UMHexagonal search algorithm, whereas the rate-distortion performance is comparable with Simplified UMHexagonal search. The proposed scheme yields average ΔPSNR decrease of 0.04dB and ΔBR increase of 1.087%, whereas UMHexagonal search produces an average ΔPSNR decrease of 0.172dB and ΔBR increase of 5.19%, and Simplified UMHexagonal search provides an average ΔPSNR decrease of 0.048dB and ΔBR increase of 1.42%. The proposed algorithm uses previously calculated MVs to estimate motion, so it is understandable that coding efficiency decreases for sequences with extremely fast or extremely complex motion. It is observed that for sequences containing low and medium level

TABLE III
RD PERFORMANCE LOSS DUE TO PARALLEL MVP USING
PROPOSED ALGORITHM WITH STANDARD MVP METHOD AS
REFERENCE.

Sequence	Proposed MVp method		Collocated MB as MVp [12]	
	ΔPSNR(dB)	ΔBR(%)	ΔPSNR(dB)	ΔBR(%)
Football	-0.1661	2.8947	-0.3100	5.3117
Mobile	-0.2561	4.5184	-0.1963	3.4654
News	-0.1577	2.5665	-0.1919	3.1041
Harbour	-0.0754	1.7099	-0.1302	2.9293
Crew	-0.1937	5.0545	-0.4405	11.1883
Ballroom	-0.1037	2.3185	-0.1833	4.0403
Exit	-0.1323	4.8015	-0.2606	9.2431
Mobcal	-0.0955	2.9203	-0.0768	2.3162
Shields	-0.0441	1.4615	-0.0616	2.0269
Old town cross	-0.0870	2.7812	-0.1317	4.264
Ducks takeoff	-0.0185	0.3572	-0.0051	0.0983
Blue-sky	-0.0809	1.7665	-0.1471	3.2259
Riverbed	-0.1870	3.3066	-0.2705	4.5369
Rushhour	-0.2516	8.8501	-0.5367	18.6217
Station2	-0.019	-0.5245	-0.1815	4.5437
Average	**-0.092**	**2.71**	**-0.20**	**5.01**

of motion complexity, our algorithm performance is comparable to Full search. For very complex sequences, we observe a performance degradation compared to Full search but still the maximum ΔPSNR degradation is just 0.12dB.

2) Parallel MVp performance: Table III presents RD performance evaluation for two different parallel MVp methods quantified by ΔPSNR and ΔBR using the proposed algorithm with standard MVp method as reference. It can be seen that using our proposed MVp method, the average PSNR loss incurred is just 0.1 dB with average increase of just 2.7% in bit rate. The reported loss is due to the usage of MVc as MVp instead of using the standard method for computing MVp. It is shown that the proposed parallel ME algorithm consistently outperforms [12], which employs MVs of the collocated macroblocks in previously encoded frame as MVp to parallelize the ME process, with a performance loss that is essentially halved.

In [9], a GPU implementation of Diamond search algorithm with comparable RD performance to UMHexagonal search algorithm is proposed. Moreover, [8] reports a maximum PSNR loss of 0.4dB for their parallel ME implementation using Simplified UMHexagonal search algorithm. Our proposed low complexity parallel ME algorithm outperforms UMHexagonal search, and PSNR loss of 0.25dB is observed in the worst case.

IV. PARALLEL IMPLEMENTATION

In this section we provide implementation details of the proposed parallel algorithm for both GPU and FPGA.

A. GPU Implementation

In this section we provide a detailed account of parallel implementation of the proposed algorithm using OpenCL.

1) GPU Architecture: NVIDIA's recent Fermi architecture [28] enables high performance parallel computing applications. The exact architecture of each GPU model is different but, generally, a Fermi based GPU consists of independent processors known as compute cores, where sets of cores are organized to form Streaming Multiprocessors (SMs). The GPU also contains on-chip global memory accessible to all computing cores, a small private memory space accessible to each core individually, and each SM is equipped with local memory space shared by all the cores resident on that SM.

OpenCL [6] is an industry standard programming API for parallel programming of heterogeneous computing platforms. The GPU is invoked by the CPU through a special program called kernel, which is written using a C-like language. Launching a kernel on GPU leads to execution of several concurrent threads known as work-items that execute the same kernel code for different parts of the input data. Work items can be grouped together in independent blocks known as work-groups which are arranged as a grid spanning a multi-dimensional index space. A multi-dimensional execution kernel model for GPU is shown in Fig. 4(a).

The OpenCL API places the programmer in control by granting access to three kinds of GPU memory spaces: global (off-chip), local and private (on-chip) memory, where each of them is tuned for a specific purpose. An overview of the memory types is provided in Fig. 4(b). Global memory constitutes a major chunk of the memory but is the slowest one and is accessible to all the multiprocessors on the device. It provides interface for data transfer between CPU and GPU and can be used to communicate data between different work-groups. Private memory is a limited per work-item memory location with very fast access. The other on-chip memory, local memory is a limited (around 64KB) per work-group chunk of memory and is
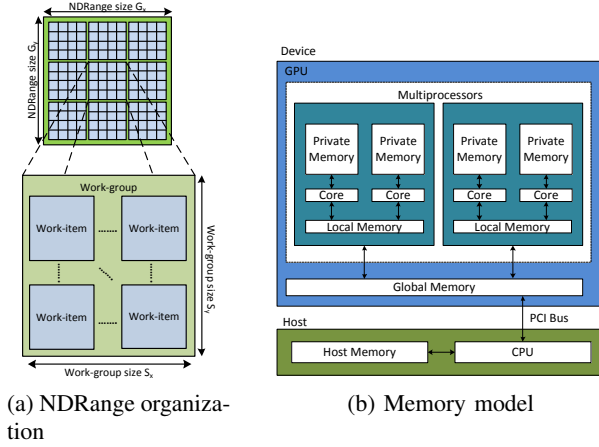
(a) NDRange organiza-
tion

(b) Memory model

Fig. 4. Example of a simplified GPU architecture showing thread organisation and memory model from the perspective of an OpenCL programmer.

Fig. 5. Organisation of work-items and the path followed by each work-item to calculate final MV for each block.

used to communicate data between work-items of a work-group. Maximum performance on GPU can be achieved by maximizing the GPU occupancy and minimising the access conflicts during coalesced memory transfers.

2) Motion Estimation Implementation: The OpenCL GPU implementation of fast ME is based on the highly parallel algorithm described in Section III, which lends itself to be parallelized at different levels. We can exploit frame level parallelism in the case of more than one reference frame. Inside a frame we can exploit pixel level as well as block level parallelism. Using pixel level parallelism leads to a large number of work-items for high video resolutions and each work-item will be performing small amount of work. On the other hand, exploiting block level parallelism requires fewer work-items but each work-item will execute multiple instructions and instruction level parallelism can be exploited in this case. In order to gain maximum performance, we have implemented block level parallelism on GPU employing smaller number of work-items, as suggested in [29]. Our GPU implementation follows the same hierarchical pattern shown by the ME algorithm *i.e.,* we divide our implementation into two modules where one module implements Coarse search and the other one Fine search. Coarse search module is well suited for GPU implementation using block level parallelism, as calculation of SADs for blocks to determine MVc and coarse predictors can be done independently. Fine search module employs RD optimisation using (1) as the cost function to calculate MVs for each block type. SAD, λ,
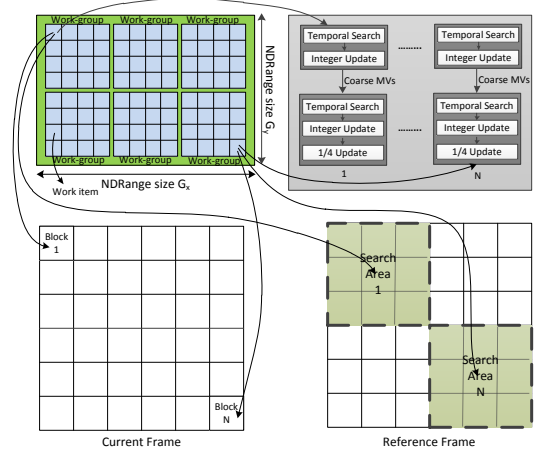
and Rmv_c are hence readily available and already calculated MVc are used as MVp in Fine search in order to compute the bit-rate cost in parallel. Using this approach, we remove any data dependency between adjacent blocks of the encoding frame and can implement the complete proposed algorithm on the GPU fully exploiting block level concurrency. A general overview of the program flow is as follows.

Initially, the CPU transfers the current frame to be encoded and the reference frame to the GPU. Coarse and Fine search steps are performed sequentially but each one is implemented to exploit the GPU parallelism. To implement Coarse search, two different OpenCL computational kernels are employed handling Temporal Search and Updates respectively. Each thread of the GPU processing core is assigned the computations for one macroblock in the encoding frame to take full advantage of the block level parallelism possible with the proposed scheme.

In the second step, Fine search is implemented to refine the MVc. It is composed of three sub-steps, namely Temporal search, Integer Updates, and Quarter pixel refinement Updates. As Fine search is performed for each block type, an OpenCL kernel is implemented to handle each block type and each sub-step. For example, one kernel does Temporal Search for 16×16 sized block, while another kernel is implemented to perform Temporal Search for 16×8 sized block and so on. For each kernel, the total number of OpenCL work-items generated depends on the block size under consideration. For example, for 16×8 block size, each GPU thread handles one block, so that the total number of

threads are twice the number of total macroblocks in the frame. For all kernels, the OpenCL work-items are organized into work-groups, where each work-group contains 32 work-items. Fig. 5 shows the program flow and the organisation of work-items. We prefer to launch a 2-dimensional NDRange kernel so that handling work-item functionality is simplified. The work-group size is hard coded as 32, while the number of work-groups are computed by analysing the video resolution and creating enough work-items to cover the complete frame *e.g.,* for VGA resolution we launch 38 work-groups. For smaller block sizes, the total number of work-items and total number of work-groups are scaled accordingly.

This work-group size is used for maximum GPU occupancy and was optimised by profiling the GPU code and using CUDA occupancy calculator [30]. Also, deploying work-groups on GPU enables the usage of local memory (on-chip memory) to reduce the data transfer time from GPU global memory. Each work-group places current block and potential update region for each block in its local memory to save memory accesses to global GPU memory. In the end, MVs for each block type of every macroblock in the frame are transferred back to the CPU. In this way, the data transfer between CPU and GPU is only performed twice for each frame.

Generally, fast ME algorithms exhibit a lot of branching and termination, which causes performance degradation on GPU. The proposed algorithm is window size independent and always performs the same number of steps to avoid early termination completely. Our implementation keeps branching to a bare minimum by adding dummy values to the border of the frames having no impact on the MV calculations.

B. Hardware Implementation

As discussed in Section III, we have implemented a two step algorithm, where each step comprises different search phases, like temporal search, update search, quarter update search *etc*. Therefore, the total number of candidates required to predict one block type are the following ones. i) For coarse search, we need to predict 6 blocks for temporal prediction and 12 blocks for integer update prediction. ii) For fine search, we need to predict 6 blocks for temporal prediction, 12 blocks for integer update, 1 zero MV, 1 MVp, and 8 quarter update blocks. So

46 candidates (blocks) are required for prediction. Since the fine search is devoted to refine MVc, it always has to be performed after the coarse search. Thus, two main choices for the hardware design can be explored: i) either to implement two separate engines for coarse and fine searches or ii) to go for hardware reuse, *i.e.* a single prediction engine for both coarse and fine search in sequential order. The latter choice is selected for our implementation, as it is observed indeed that coarse search is much faster than the fine search. The reason is that during coarse search the prediction is performed on macroblocks of fixed-size, whereas in the fine search the prediction is calculated for all block sizes. Also the number of candidates during coarse and fine searches is different, indeed 18 candidates are tested during the coarse search and 28 candidates during the fine search. Thus, implementing separate prediction engines would lead to slow down the fine search, resulting in a slow prediction process.

The proposed ME architecture relies on a parallel architecture, where the parallelism degree was selected to get maximum hardware utilization and maximum possible throughput, in terms of frames per second. Taking into account 1080p video sequences, the throughput of the architecture is defined as the ratio between the number of macroblocks to be predicted (P) and the time employed to perform the prediction:

$$TP = \frac{P \times F_{ck}}{M \cdot (N_{cs} + N_{fs})} \qquad (2)$$

where M is the number of macroblocks in one frame, equal to 8160 in case of 1080p, F_{ck} is the target clock frequency, and N_{cs} and N_{fs} are number of clock cycles required in the prediction of one macroblock by coarse and fine search respectively. Since there are seven block types (ranging from 16x16 to 4x4) and each block type requires to test 46 candidate blocks, 46x7=322 blocks have to be predicted, corresponding to $N_{cs} = 72$ and $N_{fs} = 966$ respectively. From (2), with F_{ck} equal to 175MHz and P equal to 3, the throughput is about 60 frames/s. Therefore, the proposed hardware architecture is liable to predict 48 4×4 blocks in parallel, *i.e.* 3 macroblocks. Given a $c \times r$ block, in the following we will refer to the nth $c \times r$ block of a macroblock as $c \times r(n)$. For example, 16×8(2) means the second half of block type 16×8 in a macroblock.
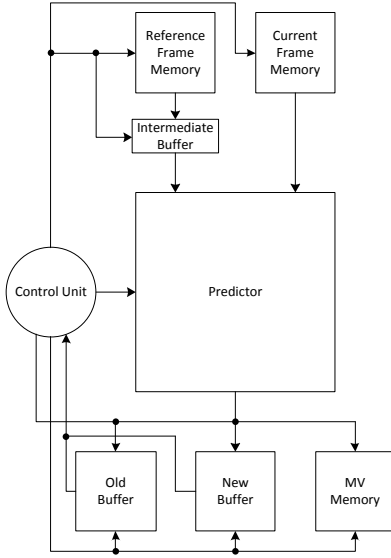
Fig. 6. Proposed architecture block scheme.

Fig. 7. Proposed architecture: frame memory organization.

The top level architecture is shown in Fig. 6, where the major unit is the Predictor, which calculates the cost to find the best match. There is a control unit (CU), where the CU starts the prediction, generates addresses, and provides control signals to the data path and to the memories. Two memories are used to store the current and the reference frames, shown in Fig. 6 as Current Frame Memory and Reference Frame Memory respectively. Similarly, there is a memory to store the final MVs (MV Memory in Fig. 6). These are the final MVs of the prediction for one complete frame. A simple intermediate buffer is deployed between the reference frame memory and the predictor in order to ensure that the required blocks could be accessed in parallel. Two buffers are deployed to store the MVc, shown in Fig. 6 as Old Buffer and New Buffer.

1) Control unit: This unit performs two major tasks: (1) generation of control signals to various blocks in the data path and the memories; (2) address generation and the verification if the block to be used as reference lies within the search window range or not. The CU adds to them proper offsets, depending upon if temporal prediction or update prediction. If the address, after the addition of offsets, lies outside the search range, the data from the frame memory are simply copied to the intermediate buffer for future usage, but no real prediction is done with these blocks.

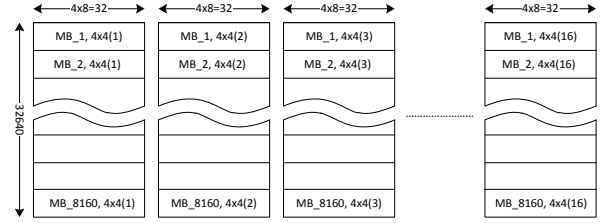*2) Frame memories:* Reference and Current memories are designed for 1080p frames. Fig. 7 shows the memory organization where 16 parallel banks are allocated. Each 4×4 block of a macroblock is stored at a same address in a different bank so that a complete macroblock can be read in single clock cycle.

3) MV memories: Old and New Buffers in Fig. 6 work in a ping-pong order for every two consecutive frames, *i.e.*, either of the buffers stores the MVc for frame N and at the same time the other buffer, which contains the MVc of the frame $N-1$, is read for prediction. The MV Memory contains the final MVs for each macroblock and the sub-blocks. After the complete prediction of a frame, the MVs stored in the MV Memory are transferred back to the CPU.

4) Intermediate Buffer: During the prediction, multiple 4×4 blocks are needed by the predictor. To assure the availability of the data, an intermediate buffer, composed of 48 4×4 memory banks, is placed between the reference frame memory and the predictor. Whenever the data has to be read from the frame memories, it is checked in the intermediate buffer. 48 4×4 blocks are read from the frame memory and written to the intermediate buffer at the same time. The CU passes the block addresses to the frame memory and also to the intermediate buffer, and if the data are available in the buffer, the memory is bypassed. During coarse search, the prediction is performed at macroblock level. So during coarse search all the read macroblocks are stored in the buffer, which ensures that the required blocks would be available for fine search.

5) Predictor: The predictor is the main unit of the top level architecture as it is responsible to calculate SADs and costs, which are later used for making decisions on all blocks. This unit takes the reference blocks and the current blocks from the memories, along with the control signals and performs the prediction. Fig. 9 shows the top level architecture of the predictor. The predictor is designed to predict two macroblocks in a ping-pong order, *i.e.* during the prediction of one current block,
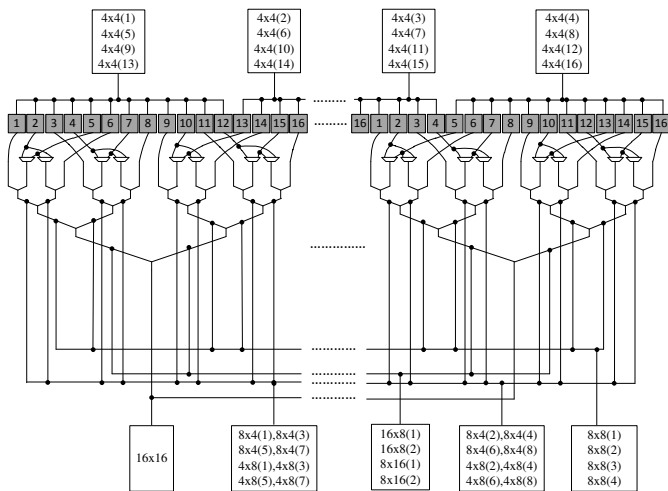
Fig. 8. SAD adder tree architecture.

Fig. 9. Proposed predictor architecture.

the required candidates for the other current block are read from the reference, and vice-versa. The predictor contains two memory banks, shown as MEM_BANK1 and MEM_BANK2, which hold 48 4×4 blocks of the candidate blocks, also referred to as the reference data. Either of the memory banks is in read mode during any 4 clock cycles, while the other one is in write mode.

Two current macroblock banks, shown as CMB_BANK1 and CMB_BANK2, hold the two current macroblocks to be predicted in the ping-pong order. The memory permutation block (MEM_PERM) alternatively selects the data to be used for prediction from one of the memory banks, containing reference blocks. A current macroblock permutation block, shown as CMB_PERM, alternatively selects the data from one of the current macroblock banks. The current macroblock dispatcher block, shown as CMB_DISP, gets in the 16 1×4 blocks from current macroblock banks through CMB_PERM unit and outputs 48 1×4 blocks towards the processing unit (PU) which contains 48 processing elements (PEs), shown in the right bottom part of Fig. 9. For prediction of different block types, the PEs need to get data from different current macroblocks and reference buffers. For example, in case of 16×16 prediction, the first PE requires the data from the $4\times4(1)$ block of the current macroblock, whereas in case of second half of 16×8, referred to as $16\times8(2)$, it would require $4\times4(9)$ of the current macroblock. So the dispatcher provides the correct current macroblock portion to each PE for all types of block predictions. The major unit for prediction to compute SADs is the PU. As
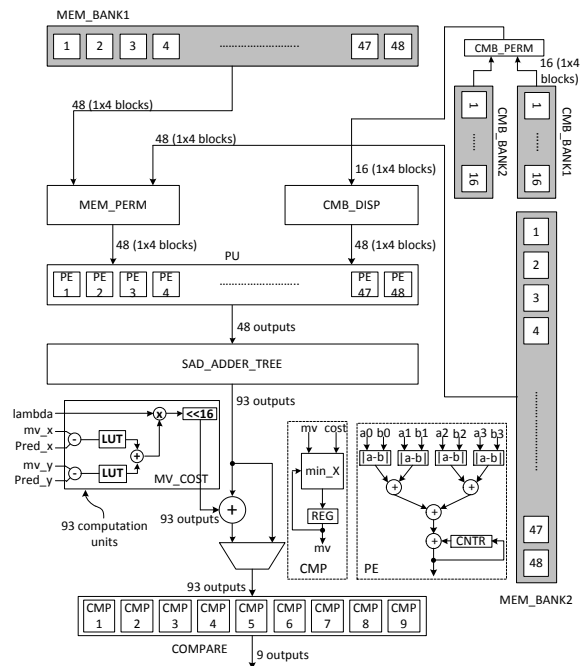
discussed before, the hardware architecture supports to predict 48 4×4 blocks in parallel. The prediction is carried out through the PU (48 PEs). The PU is able to process 3 macroblocks in parallel. Each PE processes one 1×4 block in one clock cycle. So the PU takes 4 clock cycles to find the SAD of one 4×4 block. The PE takes 4 pixels from the CMB and 4 pixels from the memory bank. It then computes the SAD for each row of a 4×4 block in one clock cycle. After 4 clock cycles the SAD of a 4×4 block is completed and it is then passed to the SAD adder tree (SAD_ADDER_TREE), shown in Fig. 8.

The SAD adder tree gets all the 4×4 SADs from the PU and it simply adds them up in different parallel orders to get SADs of all other larger block types. The number of valid SADs are $3\cdot(16\times16)/(r\times c)$ except for 8×4 and 4×8, where the number of valid SADs is 12 and r and c are the number of rows and columns in a block. The SAD adder tree is reused for different block types, such as first and second half of 16×8 and 8×16, referred to as $16\times8(1,2)$, $8\times16(1,2)$. It is worth noting that block type $16\times8(1)$ can be computed adding the terms from also the $4\times4(1,2,3,4)$ blocks, while in case of $8\times16(1)$, $4\times4(1,3,5,7)$ blocks have to be added. Therefore, some multiplexers are placed between the adder tree and the input registers, where the multiplexers alternatively select the inputs for $16\times8(1)$ or $8\times16(1)$.

The coarse search uses SAD as a comparison metric, whereas in fine search MV cost is used for comparisons. Therefore, the MV cost calculation units (MV_COST) implements (1) and calculates the MV cost, taking into account the MV of the current block and the MVp: a small lookup table (LUT) contains the number of bits to represent the MV. Overall, the adder tree generates 93 SADs, given as $3 \cdot (16 \times 16)/(r \times c)$ for 16×16, 16×8 and 8×16 and 12 SADs for all other sub-block sizes, where r and c are the number of rows and columns in a block size.

All the 93 SADs, generated by the SAD_ADDER_TREE, are further added to their corresponding costs. The MV_COST units are used to generate these costs. The compare unit gets all the 93 SADs (or MV costs) and also the corresponding MVs and it finds the block with minimum cost. This unit simply compares all the incoming costs and then stores the minimum cost and the corresponding MV. Finally, the compare unit outputs the MV of the block with minimum SAD. The MV is then passed to the CU and MV memories. Finally, the MV is stored in the MV memories, in case of final steps of coarse or fine search. On the other hand, MV is stored in the Old or New buffer if it is used for further address generations, in case of intermediate steps of coarse or fine search.

V. RESULTS AND DISCUSSION

The performance evaluation experiments are carried out on typical GPU and FPGA commercial products. For GPU experiments three different GPUs were used:

- NVIDIA Quadro 6000 @ 1 GHz, composed of 14 streaming processors and 448 cores with 6 GB memory
- NVIDIA Tesla C2075 @ 1.15 GHz, composed of 14 streaming processors and 448 CUDA cores with 6 GB memory
- NVIDIA Geforce GTX 260 @ 1.24 GHz, composed of 27 streaming processors and 216 cores with 900 MB memory

The CPU used was Core 2 Quad E5607 clocked at 2.27GHz frequency. The GPU code was developed using OpenCL 1.1 API provided by NVIDIA.

The hardware architecture has been described in VHDL, targeting up to 1080p video sequences.

TABLE IV
TIME REDUCTION VALUES FOR DIFFERENT GPU.

Resolution	Time Reduction (%)		
	GeForce GTX 260	Quadro 6000	Tesla C2075
480p	89.69	94.49	94.42
576p	89.26	95.44	95.49
720p	89.11	96.05	96.32
1080p	87.74	95.35	95.31
Average	88.95	95.33	95.38

Then, it has been implemented with the Xilinx integrated software environment (ISE 13.2) on a Virtex-6 Pro xc6vx75t Xilinx FPGA. Since several architectures for ME available in the literature target ASIC implementation, the proposed design has been synthesized with the Synopsys Design Compiler using the TSMC 90 nm standard cell library as well.

A. GPU Execution Time Performance

In order to measure the performance of the proposed GPU implementation in terms of execution speedup, we use two metrics, namely, Time Reduction (TR) and Speedup Factor (SF), given in (3) and (4) respectively.

$$TR(\%) = \frac{T_{CPU} - T_{GPU}}{T_{CPU}} \times 100 \qquad (3)$$

$$SF = \frac{T_{CPU}}{T_{GPU}} \qquad (4)$$

where, T_{CPU} and T_{GPU} are the execution time taken by single-core CPU and multi-core GPU respectively. Note that T_{CPU} and T_{GPU} are dependent on the total number of frames for each sequence, while time reduction and frame rate are independent of that.

Table IV reports the average time reduction achieved for video sequences with different resolutions. It can be seen that for every resolution, the average time reduction attained is almost the same. On Quadro 6000 and Tesla C2075, the performance gains are almost the same as both GPUs are based on Fermi architecture and have the same number of cores but slightly different clock frequencies. The time reduction on GeForce GTX 260 is lower than achieved with the two other platforms, as it is based on the older Tesla architecture and has fewer cores. It can be seen that our GPU implementation can scale successfully to handle different video resolutions and different GPU architectures
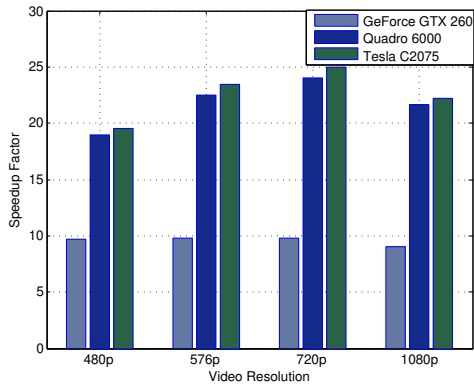
Fig. 10. Speedup factor achieved for different frame resolutions on three different GPU platforms.
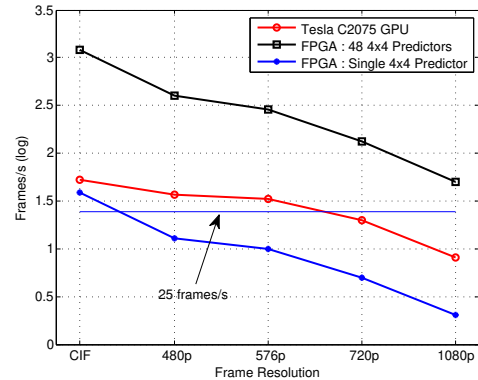

Fig. 11. Throughput (frames/s), shown on log scale, achieved with our implementations on Tesla C2075 GPU and FPGA for different frame resolutions. The straight line depicts 25 frames/s boundary.

efficiently. The results show that an average 88.95% time reduction is achieved using older GeForce GTX 260 GPU card, while on Fermi architecture based GPU cards (Quadro 6000, Tesla C2075) we can attain time reduction of 95.3% on average.

The performance of the algorithm in terms of speedup factor is shown in Fig. 10. The bar chart represents average speedup factors attainable for different resolutions on different GPU platforms. The chart shows that the performance increases with the increase in number of cores for GPUs and reaffirms the trend witnessed in Table IV. It is worthwhile noting that for higher resolutions our implementation provides a gain of 22 times over highly optimised CPU implementation for Fermi based GPUs.

Although it is difficult to draw an execution time comparison with other reported fast ME GPU implementations due to different hardware architectures and programming frameworks used, we try and present a comparison with the most similar GPU implementations. Rodriguez *et al.* [12] employed NVIDIA GTX 480 with 448 CUDA cores and peak performance of 1350 GFLOPS to implement their solution. This GPU is comparable to Tesla C2075 with 448 CUDA cores and peak performance of 1030 GFLOPS, one of the GPUs we use to implement our proposed algorithm. They report on average a speedup factor of 5 for 720p and 1080p frame resolutions, when comparing their parallel GPU implementation with fast ME CPU implementations of UMHexagonal search and Simplified UMHexagonal search. Our implementation provides a speedup of order of magnitude 22 for both 720p and 1080p on average.

NVIDIA GTS 8800 with 96 streaming cores and

peak performance of 345 GFLOPS is used by [8] to report a speedup of factor of 3.5 at best and 2.8 on average for 720p resolution sequences. We have also implemented our solution on older architecture based NVIDIA GTX 260 composed of 216 CUDA cores and having theoretical peak performance of 714 GFLOPS. We attain a speedup factor of 10.3 at best and 9.9 on average for 720p sequences. Although our GPU is 2.25 times more powerful, accordingly scaling the speedup factors indicate that our implementation achieves an average speedup factor of more than double with respect to [8].

B. Hardware Results

Similarly, results of the proposed architecture are compared with other FPGA based architectures proposed in the literature [13], [16]–[20]. In Table V it is shown that the proposed implementation achieves smaller area than the other state of the art reported implementations. The architecture is able to support up to 1080p resolution video sequences with frame rate of 60 frames/s, while 5300 frames/s are reached with QCIF format. These rates are higher than results reported by most of compared architectures. Moreover, many implementations supports the ME for a single or few block sizes, while our implementation supports all block sizes ranging from 16×16 to 4×4. Stepping down our implementation to 16×16 block size only, reduces the required clock cycles from 346 to 70 clock cycles, which leads to around 1000 frames/s for 1080p video sequences.

Table VI presents the comparison of the proposed ASIC architecture with other state of the art ASIC implementations [14], [15], [21]–[25]. The proposed architecture is able to target high resolution videos

TABLE V
FPGA COMPARISONS: THROUGHPUT (TP), FULL SEARCH (FS), FRACTIONAL MOTION ESTIMATION (FME), LOW DENSITY &
ITERATIVE SEARCH(LD&IS), MODIFIED SIMPLIFIED AND UNIFIED MULTI-HEXAGON SEARCH (MSUMH), HARDWARE MODIFIED
DIAMOND SEARCH (HMDS), ADAPTIVE CROSSED QUARTER POLAR PATTERN SEARCH (ACQPPS)

	[13]	[16]	[17]	[18]	[19]	[20]	Proposed
FPGA	Virtex-E	Cyclone-II	Startix-4	Virtex-4	Virtex-2 Pro	Virtex-2 Pro	Virtex-6
Algorithm	FS	FME	LD&IS	ACQPPS	MSUMH	HMDS	Proposed
Resolution	QCIF	1080p	1080p	QCIF	QCIF	QCIF	1080p / QCIF
Search Range	±16	–	±94	±16	±16	±16	Independent
CLB Slices (K)	14.7	–	–	3.9	11.4	7.8	3.2
LUTs (K)	10.0	13.10	18.5	3.2	18.7	10.8	2.8
Block Sizes	16×16	16×16	16×16	16×16-4×4	16×16-4×4	16×16-4×4	16×16-4×4
Freq. (MHz.)	76.1	105.2	254.8	60	145.2	246.5	175
TP (frames/s)	N/A	41	180	120	180	461	60 / 5300

TABLE VI
ASICS COMPARISONS: THROUGHPUT (TP), FULL SEARCH (FS), DIAMOND SEARCH (DS), CROSS SEARCH (CS), COARSE FULL
SEARCH (CFS), MULTI-PATH SEARCH (MPS)

	[14]	[15]	[21]	[22]	[23]	[24]	[25]	Proposed
Tech. (nm)	180	130	180	180	180	180	130	90
Algorithm	FS	FS	DS+CS	2-Step	2-Step	3-Step	CFS+MPS	2-Step
Resolution	720p	1080p	720p	1080p	1080p	1080p	1080p	1080p
Search Range	(±64, ±64)	(±65, ±65)	(±96, ±96)	(±64, ±64)	(±96, ±64)	(±128, ±96)	(±64, ±64)	Independent
Gate Count (K)	707	130	238	556	689	260	140	204
Block Sizes	16×16-4×4	16×16-4×4	16×16-8×8	16×16-4×4	16×16-8×8	16×16-8×8	16×16-8×8	16×16-4×4
Freq. (MHz)	108	300	117	155	200	200	200	300
TP (Frames/s)	30	36	30	30	30	30	30	60

because it is able to operate on many small blocks of 4×4 pixels in parallel, leading to full hardware resources utilization. On the contrary, many of the reported architectures support only 16×16 block prediction, which leads to waste of resources when small block sizes are used.

The hierarchical search strategy is used in all the reported and in the proposed implementations, where all architectures support variable-size blocks. The proposed architecture is independent of the search range, which leads to a reduced complexity in terms of hardware and control logic. It is shown that the proposed implementation occupies less area in terms of gate count among most of reported implementations and offers higher frame rate.

In Fig. 11, a comparison between the proposed GPU and FPGA implementations is shown in terms of throughput (frames/s). The y-axis is displayed in log scale to improve readability. The straight line represents 25 frames/s throughput. Two different scenarios for FPGA implementation, with single 4×4 predictor and 48 4×4 predictors are shown. It is observed that FPGA outperforms the GPU comprehensively in terms of throughput but at the design level the cost incurred for developing FPGA is far more than GPU. Figure 11 also shows that

for attaining 25 frames/s videos, for resolutions lower than 720p, GPU can handle real time video encoding while for higher resolutions like 720p and 1080p the 48 predictors FPGA architecture must be employed. In the context of future heterogeneous SoC, efficient use of resources can be foreseen by efficiently adapting the system based on available resources and constraints.

VI. CONCLUSION

In this paper, we have presented a parallel ME algorithm that aims at minimizing the effect of data dependencies due to MVp, on both the performance and the throughput of a parallel video encoder implementation. The algorithm has been implemented on GPU, on FPGA, and on ASIC, and performance has been investigated on these different platforms. It is observed that for high resolution video sequences, FPGA/ASIC provides higher frame rate compared to the GPU. As the future computing market seems more inclined towards low-power devices, FPGA could be a more suitable choice compared to GPU for ME process. On the other hand, GPU implementation shows a significant time reduction of about 95% with respect to the CPU. On the other side,

GPU can work better in high power devices, like the desktop computers.

ACKNOWLEDGMENT

REFERENCES

[1] JVT, "H.264 : Advanced video coding for generic audiovisual services," *ITU-T Recommendation H.264 and ISO/IEC 14496-10 (MPEG4-AVC)*, March 2010.

[2] Z. Chen, P. Zhou and Y. He, "Fast integer and fractional pel motion estimation for JVT," *JVT-F017, Tsinghua University, China*, December 2002.

[3] X. Yi, J. Zhang, N. Ling and W. Shang, "Improved and simplified fast motion estimation for JM," *JVT-P021, JVT Meeting , Poznan, Poland*, July 2005.

[4] S. Zhu and K.-K. Ma, "A new diamond search algorithm for fast block-matching motion estimation," *Image Processing, IEEE Transactions on*, vol. 9, no. 2, pp. 287–290, 2000.

[5] P. Meng, M. Jacobsen, and R. Kastner, "FPGA-GPU-CPU heterogenous architecture for real-time cardiac physiological optical mapping," in *Field-Programmable Technology (FPT), 2012 International Conference on*, pp. 37–42, 2012.

[6] Kronos Group Inc., "OpenCL-The open standard for parallel programming of heterogeneous systems." http:www.khronos.org/opencl/, 2013. [Online; accessed 19-July-2013].

[7] Nvidia Corporation, "CUDA-Compute Unified Device Architecture for parallel computing platforms ." http://www.nvidia.com/object/cuda_home_new.html, 2013. [Online; accessed 31-October-2013].

[8] N.-M. Cheung, X. Fan, O. Au, and M.-C. Kung, "Video coding on multicore graphics processors," *Signal Processing Magazine, IEEE*, vol. 27, no. 2, pp. 79–89, 2010.

[9] M. Schwalb, R. Ewerth, and B. Freisleben, "Fast motion estimation on graphics hardware for H.264 video encoding," *Multimedia, IEEE Transactions on*, vol. 11, no. 1, pp. 1–10, 2009.

[10] M.-J. Chen, Y.-Y. Chiang, H.-J. Li, and M.-C. Chi, "Efficient multi-frame motion estimation algorithms for MPEG-4 AVC/JVT/H.264," in *Circuits and Systems, 2004. ISCAS '04. Proceedings of the 2004 International Symposium on*, vol. 3, pp. III–737–40 Vol.3, 2004.

[11] Z. Zhou, M.-T. Sun, and Y.-F. Hsu, "Fast variable block-size motion estimation algorithm based on merge and slit procedures for H.264/MPEG-4 AVC," in *Circuits and Systems, 2004. ISCAS '04. Proceedings of the 2004 International Symposium on*, vol. 3, pp. III–725–8 Vol.3, 2004.

[12] R. Rodriguez-Sanchez, J. Martinez, G. Fernandez-Escribano, J. Claver, and J. Sanchez, "A fast GPU-based motion estimation algorithm for H.264/AVC," *Advances in Multimedia Modeling*, vol. 7131, pp. 551–562, 2012.

[13] N. Roma, T. Dias, and L. Sousa, "Customisable core-based architectures for real-time motion estimation on FPGAs," *Field Progammable Logic and Applications*, vol. 2778, pp. 745–754, sep 2003.

[14] T.-C. Chen, S.-Y. Chien, Y.-W. Huang, C.-H. Tsai, C.-Y. Chen, T.-W. Chen, and L.-G. Chen, "Analysis and architecture design of an HDTV720p 30 frames/s H.264/AVC encoder," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 16, no. 6, pp. 673–688, 2006.

[15] N. Yu, W. Jia, M. Gu, D. Wang, G. Xi, and Y. Zheng, "A high-performance configurable VLSI architecture for integer motion estimation in H.264," in *Integrated Circuits (ISIC), 2011 13th International Symposium on*, pp. 55–58, 2011.

[16] E. Castillo, C. Cardenas, and M. Jara, "An efficient hardware architecture of the H.264/AVC Half and Quarter-Pixel Motion Estimation for real-time High-Definition Video streams," in *Circuits and Systems (LASCAS), 2012 IEEE Third Latin American Symposium on*, pp. 1–4, 2012.

[17] G. Sanchez, M. Porto, and L. Agostini, "A fast hardware-friendly motion estimation algorithm and its VLSI design for real time ultra high definition applications," in *Circuits and Systems (LASCAS), 2013 IEEE Fourth Latin American Symposium on*, pp. 1–4, 2013.

[18] Y. Qiu and W. Badawy, "A prototyping virtual socket system-on-platform architecture with a novel ACQPPS motion estimator for H.264 video encoding applications," *EURASIP Journal on Embedded Systems*, pp. 4:1–4:1, jan 2009.

[19] O. Ndili and T. Ogunfunmi., "FPSoC-based architecture for a fast motion estimation algorithm in H.264/AVC," *EURASIP Journal on Embedded Systems*, vol. 2009, pp. 1–16, 2009.

[20] O. Ndili and T. Ogunfunmi, "Algorithm and architecture co-design of hardware-oriented, modified diamond search for fast motion estimation in H.264/AVC," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 21, no. 9, pp. 1214–1227, 2011.

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

[22] S. Warrington, S. Sudharsanan, and W.-Y. Chan, "Architecture for multiple reference frame variable block size motion estimation," *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, pp. 2894–2897, 2007.

[23] Z. Liu, Y. Song, M. Shao, S. Li, L. Li, S. Ishiwata, M. Nakagawa, S. Goto, and T. Ikenaga, "HDTV1080p H.264/AVC encoder chip design and performance analysis," *Solid-State Circuits, IEEE Journal of*, vol. 44, no. 2, pp. 594–608, 2009.

[24] H. Yin, H. Jia, H. Qi, X. Ji, X. Xie, and W. Gao, "A hardware-efficient multi-resolution block matching algorithm and its VLSI architecture for high definition MPEG-like video encoders," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 20, no. 9, pp. 1242–1254, 2010.

[25] G. Pastuszak and M. Jakubowski, "Hardware implementation of adaptive motion estimation and compensation for H.264/AVC," in *Picture Coding Symposium (PCS), 2012*, pp. 369–372, 2012.

[26] F. Rovati, D. Pau, E. Piccinelli, L. Pezzoni, and J.-M. Bard, "An innovative, high quality and search window independent motion estimation algorithm and architecture for MPEG-2 encoding," *Consumer Electronics, IEEE Transactions on*, vol. 46, no. 3, pp. 697–705, 2000.

[27] G. Bjontegaard, "Calculation of average PSNR differences between RD- curves," in *JVT Meeting, Tech. Rep. VCEG-M33*, April 2001.

[28] Nvidia Corporation, "The Nvidia Fermi Compute Architecture whitepaper." http://www.nvidia.com/content/PDF/fermi_white_papers/, 2012. [Online; accessed 19-July-2013.].

[29] V. Volkov, "Better performance at lower occupancy," in *GPU Technology Conference 2010 (GTC 2010)*, 2010.

[30] Nvidia Corporation, "CUDA Occupancy Calculator." http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls.