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. This work presents an inherently parallel low-complexity rate-distortion optimized fast motion estimation algorithm well suited for parallel implementations, eliminating various data dependencies caused by reliance on spatial predictions. This work also provides details of the GPU and FPGA implementations of the parallel algorithm using OpenCL and VHDL respectively, and presents a practical performance comparison between the two implementations. 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.

Ashfaq Ahmed is with the Department of Electrical Engineering, COMSATS Institute of Information Technology, Wah campus, Pakistan (ashfaqahmed@ciitwah.edu.pk) and with the Department of Electronics and Telecommunication, Politecnico di Torino, Torino, Italy (ashfaq.ahmed@polito.it). The other authors are with Politecnico di Torino, Torino, Italy (name.surname@polito.it).

"Copyright (c) 2014 IEEE. Personal use of this material is permitted. However, permission to use this material for any other purposes must be obtained from the IEEE by sending an email to pubs-permissions@ieee.org."
Parallel H.264/AVC Fast Rate-Distortion Optimized Motion Estimation using Graphics Processing Unit and Dedicated Hardware

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 where Motion Estimation (ME) is one of the main tools employed for eliminating temporal redundancies. It is the most critical and time consuming tool of the complete encoder and typically requires 60-80% of the total computational time and about 90% of the energy consumption during the encoding process [2], [3]. 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 (UMH) search [2], simplified UMH (SUMH) search [4], Diamond search [5] etc., 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 enhanced execution performance. The performance improvement can be achieved 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).

GPUs/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 [6]. 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 based 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 [7] and CUDA [8] 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 technique. The data dependencies intrinsic to this prediction 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 paper we provide new contributions tackling the problems outlined above in three different areas, namely algorithm design, GPU implementation, and FPGA implementation. The main contributions of this work are outlined hereafter for the three areas.

- An inherently parallel low-complexity fast motion estimation algorithm has been developed. While the algorithm builds on a previous algorithm [9], it has been significantly modified to allow efficient parallel implementation, particularly breaking spatial dependencies among macroblocks and in the motion vector prediction stage. The proposed parallel algorithm exhibits comparable RD performance with respect to well known serial fast motion estimation algorithms, and is well suited for a GPU/FPGA implementation.
- A GPU based implementation of the proposed algorithm utilizing full RD optimization that supports quarter pixel accuracy is presented. The GPU implementation is developed in OpenCL instead of CUDA, which allows for seamless portability across different platforms and vendors. The GPU implementation is shown to achieve higher order of speedup for high resolution videos compared to state-of-the-art reported implementations.
- A search window independent parallel FPGA implementation, compounded with smaller block size predictors for all block-types is presented. The proposed architecture occupies fewer hardware resources and achieves higher throughput compared to other reported implementations.

Finally, a discussion relating to target application scenarios for heterogeneous systems is presented. To our best knowledge, this is the first paper that performs a practical performance comparison between GPU and FPGA implementations.

The remainder of the paper is organized as follows. Section II provides a brief overview of the related work. A detailed description of the proposed parallel algorithm is provided in Section III. The GPU and FPGA 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 the literature, several GPU and FPGA implementations for fast ME exist. An overview of some of them is provided in this section. Cheung et al. present a GPU implementation of SUMH in [10]. 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. [11] have proposed a parallel implementation of Diamond search by completely ignoring the standard motion vector prediction generation method. They propose to use predictors generated by Forward Dominant Vector Selection [12] and to use Split and Merge [13] techniques to generate predicted MV (MVp) for all blocks. Rodriguez et al. [14] 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 [15]–[18], whereas the major contribution is for fast ME algorithm implementation [19]–[28]. The works in [15], [19]–[23] are implemented on FPGA-based platforms, while [16]–[18], [24]–[28] are ASIC implementations. In [16], full search is deployed with some speedup techniques, such as pixel bit precision reduction, macroblock sub-sampling, and adaptive search range adjustment. In [25]–[27], hierarchical search approach is combined with full search. Typically, two [25], [26] or three [27] levels are employed. The architecture in [24] supports Diamond search and Cross search. In [28], an architecture is presented for adaptive ME using hierarchical and multipath search techniques. The architectures in [17], [18] present configurable VLSI solutions for ME in H.264/AVC. 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 the fast ME algorithm presented in [9]. The algorithm in [9] exhibits strong spatial dependencies, as it relies heavily on reusing MVs of spatially collocated blocks. The algorithm in [9] has been significantly modified, eliminating spatial dependencies, to effectively suit parallel implementations with negligible loss in coding efficiency. The proposed algorithm is hierarchical in nature, as it consists of two sequential steps, namely Coarse search and Fine search. The Coarse search is performed for each macroblock of each frame and generates its corresponding MV called as coarse MV (MVC). Fine search further refines the already estimated MVC and attains the best predictor i.e., a block of pixels values and its corresponding best MV, 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 correlations among frames by using MVs from previous frames to find best temporal prediction for the current frame. The data dependency exhibited in the MVp calculation stage is eliminated by exploiting the hierarchical nature of the proposed algorithm.
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).

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 $MV(x, y; N-1)$, $MV(x+1, y; N-1)$, $MV(x, y+1; N-1)$, $MV(x-1, y; N-1)$, $MV(x, y-1; N-1)$, $MV(x-1, y-1; N-1)$ 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 \( N - 1 \). The six temporal predictors are selected as a trade-off to keep the algorithm complexity low with negligible impact on RD performance compared to [9].

2) **Updates**: After the Temporal Search, a grid of 12 points referred to as Updates is applied to get 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. The threshold used for assigning each macroblock to a set of updates is predefined and the same as in [9]. 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,

\[
J = SAD + \lambda \cdot R \cdot (MV - MV_p)
\]

where \( J \) is the cost, \( \lambda \) is the Lagrangian multiplier. The term \( R \cdot (MV - MV_p) \) 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 Equation (1).

C. Parallel MVp Modification

In the H.264/AVC standard, MVp are computed using MVs of neighbouring macroblocks of the current frame, which restricts fully parallel RD-optimized encoding at block level. The MVp term in Equation (1) manifests this spatial dependency. 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 diverse architectures.
IV. PARALLEL IMPLEMENTATION

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

A. GPU Implementation

In this section we provide a detailed account of parallel implementation of the proposed algorithm using OpenCL. OpenCL is a royalty-free cross-platform industry framework, that provides APIs for parallel programming of heterogeneous computing platforms. We chose OpenCL over CUDA to avoid hardware lockdown and because of the flexibility to seamlessly port developed application to diverse platforms, such as multicore CPUs, GPUs, DSPs, mobile devices and so on.

1) GPU Architecture: NVIDIA’s recent Fermi architecture [29] 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 employs a data-parallel execution pattern and models a heterogeneous parallel computing system as a host (CPU) and an OpenCL device (GPU). 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 2-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 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 [30]. 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 \( \text{MV}_c \) and coarse predictors can be done independently. Fine search module employs RD optimisation using (1) as the cost function to calculate \( \text{MVs} \) for each block type. All the quantities in (1) are hence readily available and already calculated \( \text{MV}_c \) are used as \( \text{MV}_p \) 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 \( \text{MV}_c \). It is composed of three sub-steps, namely

---

**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.

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 $16 \times 16$ sized block, while another kernel is implemented to perform Temporal Search for $16 \times 8$ sized block and so on. For each kernel, the total number of OpenCL work-items depends on the block size under consideration. For example, for $16 \times 8$ 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. The work-group size is hard coded as 32, 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 [31]. 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**

In this section we provide a detailed account of parallel implementation of the proposed ME algorithm targeting FPGA and ASIC platforms.

1) **FPGA/ASIC Architecture:** FPGAs are integrated circuits, consisting of an array of interconnected logic cells that can be configured to realize different digital designs. The configurable logic block (CLB) is the basic array element, which contains customizable logic modules, such as for example look-up tables and flip-flops. Connections among CLBs are configured by means of programmable switch matrices and wires arranged along rows and columns. FPGAs also include hardwired components, such as multipliers, input/output interfaces and memory components. In ASICs, the device is custom built for the particular design. CAD tools are available to map a designed architecture onto ASIC or FPGA technologies; this mapping is driven by a specified set of constraints related to both complexity (evaluated in terms of occupied area or amount of used resources) and clock frequency.

2) **Motion Estimation Implementation:** As discussed in Section III, we have implemented the 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. These characteristics are exploited to tailor the architecture around the proposed parallel ME algorithm. 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 reuse 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 that coarse search is much faster than 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:

\[ TP = \frac{P \times F_{ck}}{M \cdot (N_{cs} + N_{fs})} \]  

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, 46\times7=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\times4 blocks in parallel, i.e. 3 macroblocks. Given a \( c \times r \) block, in the following we will refer to the \( n \)th \( c \times r \) block of a macroblock as \( c \times r(n) \). For example, 16\times8(2) means the second half of block type 16\times8 in a macroblock.

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. The control unit (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.

**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 the address proper offsets, depending upon if temporal prediction or update prediction is performed.
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 a single clock cycle.

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.

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.

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, 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 $1 \times 4$ blocks from current macroblock banks through CMB_PERM unit and outputs 48 $1 \times 4$ 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 $16 \times 16$ prediction, the first PE requires the data from the $4 \times 4(1)$ block of the current macroblock, whereas in case of second half of $16 \times 8$, referred to as $16 \times 8(2)$, it would require $4 \times 4(9)$ 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 $4 \times 4$ 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 $1 \times 4$ block in one clock cycle. So the PU takes 4 clock cycles to compute the SAD of one $4 \times 4$ block, that is then passed to the SAD adder tree (SAD_ADDER_TREE), shown in Fig. 8.

The SAD adder tree simply adds the results produced by the PU in different parallel orders to get SADs of all other larger block types. Thus, the number of SADs produced by the SAD adder tree is $3 \cdot (16 \times 16)/(r \times c)$ except

![Fig. 8. SAD adder tree architecture.](image-url)
Fig. 9. Proposed predictor architecture.

for block size 8×4 and 4×8, where the number of SADs is 12 (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 16×8(1,2), 8×16(1,2). It is worth noting that block type 16×8(1) can be computed adding the terms corresponding to the 4×4(1,2,3,4) blocks, whereas case 8×16(1) requires 4×4(1,3,5,7) blocks. Therefore, some multiplexers are placed between the adder tree and the input registers, where the multiplexers alternatively select the inputs for 16×8(1) or 8×16(1).

As described in Section III, the coarse search uses the 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 \(16 \times 16\), \(16 \times 8\) and \(8 \times 16\) 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 SADadder_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

A. 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 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. Bjontegaard Delta PSNR, denoted as \(\Delta\)PSNR with measure unit (dB) and Bjontegaard Delta bit rate, denoted as \(\Delta\)BR with measure unit (%) is employed to quantify the variations in PSNR and bitrate observed for RD performance evaluations [32].

We performed a preliminary experiment using (UMH) 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. It was found 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% on average.

1) Serial MVp performance: Table I reports the performance evaluation of the proposed algorithm with standard MVp method and the results are compared with other existing fast search algorithms. These results are provided to establish the viability of the proposed algorithm for fast motion estimation. 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) (UMH) Search [2] and iii) (SUMH) search [4]. UMH and SUMH 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 I shows that the proposed algorithm performs remarkably better than UMH search algorithm, whereas its rate-distortion performance is comparable with SUMH search. The proposed scheme yields average PSNR
TABLE I
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.

<table>
<thead>
<tr>
<th>Sequence (Resolution)</th>
<th>Proposed Scheme</th>
<th>UMH Search</th>
<th>SUMH Search</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>∆PSNR</td>
<td>∆BR</td>
<td>∆PSNR</td>
</tr>
<tr>
<td>Football (CIF)</td>
<td>-0.13</td>
<td>2.24</td>
<td>-0.32</td>
</tr>
<tr>
<td>Miss-america (CIF)</td>
<td>-0.02</td>
<td>0.85</td>
<td>-0.05</td>
</tr>
<tr>
<td>News (CIF)</td>
<td>-0.04</td>
<td>0.68</td>
<td>-0.09</td>
</tr>
<tr>
<td>Ballroom (VGA)</td>
<td>-0.08</td>
<td>1.88</td>
<td>-0.11</td>
</tr>
<tr>
<td>Exit (VGA)</td>
<td>-0.08</td>
<td>3.20</td>
<td>-0.12</td>
</tr>
<tr>
<td>Harbour (576p)</td>
<td>-0.01</td>
<td>0.23</td>
<td>-0.07</td>
</tr>
<tr>
<td>Stockholm (576p)</td>
<td>-0.02</td>
<td>0.55</td>
<td>-0.07</td>
</tr>
<tr>
<td>Mobcal (720p)</td>
<td>-0.01</td>
<td>0.19</td>
<td>-0.05</td>
</tr>
<tr>
<td>Ducks take off (720p)</td>
<td>-0.001</td>
<td>0.12</td>
<td>-0.03</td>
</tr>
<tr>
<td>Parkrun (720p)</td>
<td>-0.002</td>
<td>0.045</td>
<td>-0.03</td>
</tr>
<tr>
<td>Blue-sky (1080p)</td>
<td>-0.04</td>
<td>0.96</td>
<td>-0.29</td>
</tr>
<tr>
<td>Rushhour (1080p)</td>
<td>-0.03</td>
<td>0.57</td>
<td>-0.32</td>
</tr>
<tr>
<td>Terrace (1080p)</td>
<td>-0.04</td>
<td>1.51</td>
<td>-0.18</td>
</tr>
<tr>
<td>Tennis (1080p)</td>
<td>-0.008</td>
<td>1.50</td>
<td>-0.54</td>
</tr>
<tr>
<td>Crowd run (1080p)</td>
<td>-0.07</td>
<td>1.60</td>
<td>-0.19</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>-0.03</strong></td>
<td><strong>1.08</strong></td>
<td><strong>-0.16</strong></td>
</tr>
</tbody>
</table>

degradation of 0.03dB and bit rate increase of 1.08%, whereas UMH search produces an average degradation of 0.16dB and bit rate increase of 4.89%, and SUMH search provides an average degradation of 0.05dB and bit rate increase of 1.54%. 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 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 degradation is just 0.13dB.
Table II presents RD performance evaluation for two custom parallel MVp generation methods, namely the proposed one (Section III-C) and the one in [14] that employs MVs of the collocated macroblocks in previously encoded frame as MVp to parallelize the ME process. The two methods have been implemented to compute predicted MVs in the proposed fast parallel ME algorithm in order to assess the effectiveness of the proposed MVp generation method. For RD performance evaluation, the proposed ME algorithm with standard MVp generation method is used as the reference and the performance is quantified by $\Delta$PSNR and $\Delta$BR. It can be seen that using our proposed MVp generation method, the average degradation 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 MVp.
TABLE III
TIME REDUCTION VALUES FOR DIFFERENT GPU.

<table>
<thead>
<tr>
<th>Resolution</th>
<th>GeForce GTX 260</th>
<th>Quadro 6000</th>
<th>Tesla C2075</th>
</tr>
</thead>
<tbody>
<tr>
<td>480p</td>
<td>89.69</td>
<td>94.49</td>
<td>94.42</td>
</tr>
<tr>
<td>576p</td>
<td>89.26</td>
<td>95.44</td>
<td>95.49</td>
</tr>
<tr>
<td>720p</td>
<td>89.11</td>
<td>96.05</td>
<td>96.32</td>
</tr>
<tr>
<td>1080p</td>
<td>87.74</td>
<td>95.35</td>
<td>95.31</td>
</tr>
<tr>
<td>Average</td>
<td>88.95</td>
<td>95.33</td>
<td>95.38</td>
</tr>
</tbody>
</table>

using the standard method for computing MVp. In the framework of the proposed fast algorithm, it is shown that our proposed MVp method consistently outperforms the method proposed in [14] with a performance loss that is essentially halved. The proposed parallel algorithm is then implemented on both GPU and FPGA, where the RD performance remains the same regardless of the implementation.

In [11], a GPU implementation of Diamond search algorithm with comparable RD performance to UMH search algorithm is proposed. Moreover, [10] reports a maximum PSNR loss of 0.4dB for their parallel ME implementation using SUMH search algorithm. Our proposed low complexity parallel ME algorithm outperforms UMH search, and PSNR loss of 0.25dB is observed in the worst case.

B. Execution Performance Evaluation

The execution time 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. 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 Synopsys Design Compiler using a 90 nm standard cell library as well.
Fig. 10. Speedup factor achieved for different frame resolutions on three different GPU platforms.

TABLE IV
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 CROSS QUARTER POLAR PATTERN SEARCH (ACQPPS).

<table>
<thead>
<tr>
<th>FPGA</th>
<th>Resolution</th>
<th>Search Range</th>
<th>CLB Slices (K)</th>
<th>LUTs (K)</th>
<th>Block Sizes</th>
<th>Freq. (MHz.)</th>
<th>TP (frames/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Startix-4</td>
<td>QCIF</td>
<td>±16</td>
<td>14.7</td>
<td>10.0</td>
<td>16×16</td>
<td>76.1</td>
<td>N/A</td>
</tr>
<tr>
<td>Cyclone-2</td>
<td>1080p</td>
<td>±94</td>
<td>13.10</td>
<td>13.10</td>
<td>16×16×16</td>
<td>105.2</td>
<td>105.2</td>
</tr>
<tr>
<td>Virtex-4</td>
<td>QCIF</td>
<td>±16</td>
<td>18.5</td>
<td>18.5</td>
<td>16×16×4×4</td>
<td>254.8</td>
<td>254.8</td>
</tr>
<tr>
<td>Virtex-2 Pro</td>
<td>QCIF</td>
<td>±16</td>
<td>3.9</td>
<td>3.2</td>
<td>16×16-4×4</td>
<td>60</td>
<td>60 / 5300</td>
</tr>
<tr>
<td>Virtex-2 Pro</td>
<td>QCIF</td>
<td>±16</td>
<td>11.4</td>
<td>18.7</td>
<td>16×16-4×4</td>
<td>145.2</td>
<td>145.2</td>
</tr>
<tr>
<td>Virtex-6</td>
<td>QCIF</td>
<td>±16</td>
<td>7.8</td>
<td>10.8</td>
<td>16×16-4×4</td>
<td>246.5</td>
<td>246.5</td>
</tr>
<tr>
<td>Proposed</td>
<td>QCIF</td>
<td>±16</td>
<td>3.2</td>
<td>2.8</td>
<td>16×16-4×4</td>
<td>175</td>
<td>175</td>
</tr>
</tbody>
</table>

1) 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.

\[
TR(\%) = \frac{T_{CPU} - T_{GPU}}{T_{CPU}} \times 100
\]

\[
SF = \frac{T_{CPU}}{T_{GPU}}
\]

where, \(T_{CPU}\) and \(T_{GPU}\) are the times taken for executing the proposed parallel ME algorithm by single-core CPU and multi-core GPU respectively. \(T_{GPU}\) represents the total time taken by the GPU that also includes the time
TABLE V
ASICs comparisons: THROUGHPUT (TP), FULL SEARCH (FS), DIAMOND SEARCH (DS), CROSS SEARCH (CS), COARSE FULL SEARCH (CFS), MULTI-PATH SEARCH (MPS).

<table>
<thead>
<tr>
<th>Tech. (nm)</th>
<th>[16]</th>
<th>[17]</th>
<th>[24]</th>
<th>[25]</th>
<th>[26]</th>
<th>[27]</th>
<th>[28]</th>
<th>Proposed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm</td>
<td>FS</td>
<td>FS</td>
<td>DS+CS</td>
<td>2-Step</td>
<td>2-Step</td>
<td>3-Step</td>
<td>CFS+MPS</td>
<td>2-Step</td>
</tr>
<tr>
<td>Resolution</td>
<td>720p</td>
<td>1080p</td>
<td>720p</td>
<td>1080p</td>
<td>1080p</td>
<td>1080p</td>
<td>1080p</td>
<td>1080p</td>
</tr>
<tr>
<td>Search Range</td>
<td>(±64, ±64)</td>
<td>(±65, ±65)</td>
<td>(±96, ±96)</td>
<td>(±64, ±64)</td>
<td>(±96, ±64)</td>
<td>(±128, ±96)</td>
<td>(±64, ±64)</td>
<td>Independent</td>
</tr>
<tr>
<td>Gate Count (K)</td>
<td>707</td>
<td>130</td>
<td>238</td>
<td>556</td>
<td>689</td>
<td>260</td>
<td>140</td>
<td>204</td>
</tr>
<tr>
<td>Block Sizes</td>
<td>16×16-4×4</td>
<td>16×16-4×4</td>
<td>16×16-8×8</td>
<td>16×16-4×4</td>
<td>16×16-8×8</td>
<td>16×16-8×8</td>
<td>16×16-8×8</td>
<td>16×16-4×4</td>
</tr>
<tr>
<td>Freq. (MHz)</td>
<td>108</td>
<td>300</td>
<td>117</td>
<td>155</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>300</td>
</tr>
<tr>
<td>TP (Frames/s)</td>
<td>30</td>
<td>36</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>60</td>
</tr>
</tbody>
</table>

Table III reports the average ME time reduction achieved for video sequences with different resolutions. It can be observed that for every resolution, the average ME 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 for ME on GeForce GTX 260 is lower than what is 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 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 for ME.

The performance of the ME algorithm in terms of speedup factor is shown in Fig. 10. The bar chart represents average speedup factors attainable for various resolutions on diverse 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 III. 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. While the provided results only affect one segment of the complete encoder implementation namely, ME, it is of interest to calculate the overall encoder speedup that can be achieved. In our profiling experiments ME takes up around 70% of the total execution time. Theoretically an overall speedup factor of 3.33 can be achieved for this case. Our Fermi based GPU implementations achieve a speedup factor of 21 on average for the ME segment, hence an average overall speedup factor of 3.01 is achieved for the complete encoder.

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. [14] 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 UMH search and SUMH search. Our implementation provides a speedup of about 22 for both 720p and 1080p on average.

NVIDIA GTS 8800 with 96 streaming cores and peak performance of 345 GFLOPS is used in [10] 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 that is more than double with respect to [10].

2) Hardware Results: Similarly, results of the proposed architecture are compared with other FPGA based architectures proposed in the literature [15], [19]–[23]. In Table IV 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. This figure scales up to 5300 frames/s if QCIF format is used. 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 $16 \times 16$ to $4 \times 4$. Stepping down our implementation to $16 \times 16$ 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 V presents the comparison of the proposed ASIC architecture with other state of the art ASIC implementations [16], [17], [24]–[28]. The proposed architecture is able to target high resolution videos because it is able to operate on many small blocks of $4 \times 4$ pixels in parallel, leading to full hardware resources utilization. On the contrary, many of the reported architectures support only $16 \times 16$ 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 $4 \times 4$ predictor and 48 $4 \times 4$ 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. Correspondingly, GPU can work better in high power devices, like the desktop computers.

ACKNOWLEDGMENT

The authors would like to thank Dr. Daniele Alfonso from STMicroelectronics for providing valuable support for our work, as well as their H.264/AVC software implementation.

REFERENCES


Muhammad Usman Shahid received his Masters degree with laude in Electronics Engineering from Politecnico di Torino, Italy in November 2010 and the Ph.D from the same institution in 2014. His areas of research interest are Image and Video Compression, High Performance Computing, and Information Security and Privacy.

Ashfaq Ahmed received the M.Sc. and Ph.D. Degree from the Department of Electronics and Telecommunications, Politecnico di Torino, Torino, Italy, in 2010 and 2014, respectively. He is currently an Assistant Professor at the Department of Electrical Engineering, COMSATs Institute of Information Technology, Wah campus, Pakistan. His current research interests include the ASICs and FPGA design and implementation of digital signal processing architectures, especially the areas closely related to digital communications, and video and image coding.

Maurizio Martina (S’98-M’94) was born in Pinerolo, Italy, in 1975. He received the M.Sc. and Ph.D. in electrical engineering from Politecnico di Torino, Italy, in 2000 and 2004 respectively. He is currently an Assistant Professor at the VLSI Lab, Politecnico di Torino. His research activities include VLSI design and implementation of architectures for digital signal processing and communications.
Guido Masera (SM'07) received the Dr. Ing. Degree (summa cum laude) in 1986 and the Ph.D. degree in electronic engineering from the Politecnico di Torino, Torino, Italy, in 1992. Since 1986 to 1988, he was a Researcher with the Centro Studi e Laboratori in Telecomunicazioni (CSELT), Torino, Italy, involved in the standardization activities for the GSM system. Since 1992, he has been an Assistant Professor and then Associate Professor with the Electronic Department, where he is member of the VLSI-Lab group. His research interests include several aspects in the design of digital integrated circuits and systems, with special emphasis on high-performance architecture development (especially for wireless communications and multimedia applications) and on-chip interconnect modeling and optimization. He has co-authored 230 journal and conference papers in the areas of ASIC-SoC development, architectural synthesis, VLSI circuit modeling and optimization. In the frame of competitive National and European research projects, he has been co-designer of several ASIC and FPGA implementations in the fields of Artificial Intelligence, Computer Networks, Digital Signal Processing, Transmission and Coding. He is an associate editor of the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS II.

Enrico Magli received the Ph.D. degree in Electrical Engineering in 2001, from Politecnico di Torino, Italy. He is currently an Associate Professor at the same university, where he leads the Image Processing Lab. His research interests are in the field of image and video compression, multimedia signal processing and networking, compressed sensing, distributed source coding, image and video security. He is an associate editor of the IEEE Transactions on Circuits and Systems for Video Technology, and of the IEEE Transactions on Multimedia. He has been general co-chair of IEEE MMSP 2013, and has been TPC co-chair of ICME 2012, VCIP 2012, MMSP 2011 and IMAP 2007. He has published more than 40 papers in refereed international journals, 3 book chapters, and over 90 conference papers. He is a co-recipient of the IEEE Geoscience and Remote Sensing Society 2011 Transactions Prize Paper Award, and has received a 2010 Best Associate Editor Award of IEEE Transactions on Circuits and Systems for Video Technology.