An Efficient Implementation of H.264/AVC Integer Motion Estimation Algorithm on Coarse-grained Reconfigurable Computing System

Kiem-Hung Nguyen
National ASIC system Engineering Research Center, Southeast University, Nanjing, China
Email: hungnvnu@gmail.com

Peng Cao and Xue-Xiang Wang
National ASIC system Engineering Research Center, Southeast University, Nanjing, China
Email: {caopeng, wxx}@seu.edu.cn

***Abstract*—Variable block size integer motion estimation (VBS-IME) is one of several tools which contribute to H.264/AVC's excellent coding efficiency. However, its high computational complexity and huge memory access bandwidth make it difficult to implement. Therefore, a hardware accelerator is indispensable for full-search VBS-IME in real-time video encoding applications. To overcome some of the limitations of conventional microprocessors and fine-grained reconfigurable devices in the field of multimedia and communication baseband processing, we have proposed a coarse-grained dynamically reconfigurable computing system, called REMUS. The paper presents architecture and compiling flow proposed for REMUS system, and shows that it is possible to implement a high complexity application as H.264/AVC full-search VBS-IME algorithm with competitive performance on platform of REMUS system. Experimental results have proven that the REMUS system operating at 200 MHz can perform VBS-IME at real-time speed for CIF/SDTV@30fps video sequences with two reference frames and maximum search range of [-16,15]/[-8,7]. The implementation, therefore, can apply for H.264/AVC encoder in mobile multimedia applications. REMUS system is designed and synthesized by using TSMC 65nm low power technology. The die size of REMUS is 23.7 mm^2. REMUS consumes about 194mW while working at 200MHz.**

***Index Terms*—Reconfigurable Computing, REMUS, Coarse-grained Dynamically Reconfigurable Architecture, H264/AVC, Variable Block Size Integer Motion Estimation.**

I. INTRODUCTION

Many multimedia systems are hand-held systems, such as MP3 players, PDAs and 3G phones, which require capability of handling real-time functions such as communication, camera, video, audio, touch screen, TV, and GPS in time. On the other hand, because of aiming at mobile multimedia applications, they also have constraints of performance, cost, power, and size. Beside ASIC, DSP, or ASIP approaches that were used for implementing such systems in the past, a very promising solution is to map these kernel functions onto Coarse-Grained Reconfigurable Architectures (CGRAs) which can achieve high performance approximately ASIC while maintaining a degree of flexibility close to DSP processors. By dynamically reconfiguring hardware, CGRAs allow many hardware tasks to be mapped onto the same hardware platform in time-multiplexed fashion, therefore also result in reduction in area and power consumption of the design. In the last decade, several CGRAs have been proposed, such as RaPiD [7], MorphoSys [10], PACT XPP-III [11], ADRES [12], and so on.

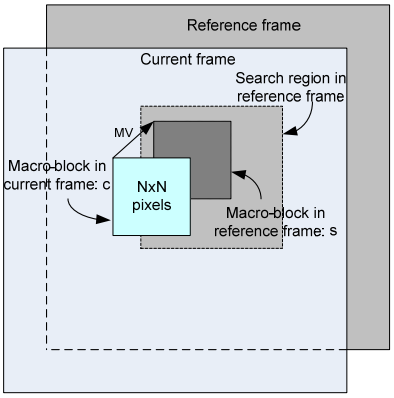


Figure 1. Block matching based ME algorithm.

H.264/AVC is the latest video coding standard developed jointly by ITU-T Video Coding Experts Group and ISO/IEC Moving Picture Experts Group [1]. It is designed to replace all the past video standards in almost all kinds of applications. So, it defines different profiles and tools to meet the various requirements in different applications.

Motion Estimation (ME) in video coding exploits temporal redundancy of a video sequence by finding the best matching candidate block of a current macro-block (MB) from a search window in reference frames (Fig. 1). The H.264 standard supports variable block size motion estimation (VBS-ME), it means that with each current macro-block there are 41 partitions or sub-partitions of 7 different sizes (4x4, 4x8, 8x4, 8x8, 8x16, 16x8, and

doi:10.4304/jcp.8.3.594-604

16x16). The H.264 standard also supports quarter-pixel accurate ME, so VBS-ME is partitioned into integer ME (IME) and fractional ME (FME). If full-search VBS-ME is chosen and search range is [-p, p-1] in both *x* - and *y* - directions, the size of the search window is given by *[N+2p-1, N+2p-1]*, therefore, each of 41 partitions or sub-partitions needs to be matched with $4p^2$ candidates in the search window by evaluating the cost function RDO (Rate-Distortion Optimization):

$$J = SAD + \lambda \times MV\_bit\_rate, \quad (1)$$

$$SAD(m,n) = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} | c(i,j) - s(i+m, j+n) | . \quad (2)$$

Here, SAD(m,n) is sum of absolute differences of the candidate block and the candidate block at search position (m, n), MV_bit_rate is estimated by using a lookup table defined in the reference software by JVT, and λ is Lagranggian parameter that is derived from quantization parameter to make trade-off between distortion and bit-rate. $\{c(x,y) \mid 0 \le x,y \le N-1\}$ is current block pixel; $\{s(x,y) \mid -p \le x,y \le N+p-2\}$ is search area pixel; $MV = \{(m,n) \mid -p \le m,n \le p-1\}$, Motion Vector, is offset between position of the current macro-block and position of a candidate block in search window.

VBS-IME is one of several tools which contribute to H.264/AVC's excellent coding efficiency. However, its high computational complexity and huge memory access bandwidth make it difficult to design. The VBS-IME with 1 reference frames, [-16, 15] search range, consumes more than 50% of the total encoding [6]. Therefore, a hardware accelerator is indispensable for full-search VBS-IME in real-time video encoding applications. In 2003, Huang et al. [2] publicized the first VLSI architecture supporting full-search VBS-IME of H.264. After that, there are a lot of VLSI architectures have been proposed for VBS-IME (e.g. [3-6]). Some works also have tried to mapping the IME algorithms onto CGRAs [7-9], but they are very simple, and mainly aim at validating the proposed architecture. To increase the total throughput and solve high computational complexity, the proposed VLSI designs usually adopt parallel architecture that normally includes 256 PEs (Processing Elements) in order to compute concurrently 256 absolute differences in (2), and an adder-tree to sum up 41 SADs. Whereas, the existing CGRA architectures often do not offer enough computational resources for exploiting complete inherent parallelisms of the algorithm, that makes them impossible to satisfy real-time processing requirement of applications. Therefore, until now all of the VBS-IME hardware designs for real-time H.264 video encoding applications are based on ASIC approach.

In this paper, we first introduce our architecture of coarse-grained reconfigurable computing system, so called REMUS, and then we will present optimizing and mapping of VBS-IME algorithm, which is aimed at mobile multimedia applications, onto our REMUS system. The experimental results showed that the implementation meets requirements for high performance, flexibility, and energy efficiency.

The rest of the paper is organized as follows. In section 2, we present overview of the REMUS architecture. An overall compiling flow that is possible to apply for mapping an arbitrary algorithm onto the platform of REMUS system is proposed in section 3. The section 4 gives implementation of VBS-IME algorithm on REMUS system in detail. The experimental results and conclusion are given in the section 5 and 6, respectively.

II. ARCHITECTURE OF REMUS

A. Overview of REMUS Architecture

REMUS, stands for **RE**configurable **MU**ltimedia **S**ystem, is a coarse-grained dynamically reconfigurable computing system for multimedia and communication baseband processing, the overall architecture of the REMUS system is shown in Fig. 2. The REMUS system consists of an ARM7TDMI, two Reconfigurable Processing Units (RPUs), and several assistant function modules, including an Interrupt Controller (IntCtrl), a Direct Memory Access (DMA) unit, an External Memory Interface (EMI) and a Micro Processor Unit (μPU), etc. All modules are connected with each other by ARM AMBA bus. The ARM processor functions as a host processor which is used for controlling application, and scheduling tasks. The RPU is a powerful dynamically reconfigurable system, which consists of four Reconfigurable Computing Arrays (RCAs). Each of RCAs is an array of 8x8 RCs (Reconfigurable Cells), and can run independently to accelerate computing performance.

In comparison with the previous version [13] of REMUS, this version has been improved by adding the μPU and optimizing Data-flow. The μPU is an array of 8 RISC micro-processors (μPEA). The micro-processors (μPs) of the μPU communicate with each other and the ARM processor by a simple mailbox mechanism. When one mail arriving, an interrupt will inform the corresponding μP to check the mail and handle it. The mailbox mechanism is very similar to mailbox communication in a common Operation System (OS). The μP can operate as a supervisor associated with a RPU or as an independent processing element. As a supervisor, the μP is responsible for generating or/and selecting configuration words at run-time. As an independent processing element, the μP is in charge of executing control-intensive tasks and the other tasks that are not suited for RPUs, such as float point operations, bit-level operations. With the extended ability of μPU, Hardware/Software (HW/SW) partition could become more flexible. By mapping computation-intensive kernel loops onto the RPU, control-intensive tasks or bit-level, float-point operations onto the μPU the REMUS system can achieve high performance approximately ASIC while maintaining a degree of flexibility close to DSP processors. On the other hand, to satisfy high data bandwidth requirement of video encoding applications, data flow of REMUS have been optimized to supports a three-level hierarchical memory, including off-chip memory (DDR SDRAM), on-chip memory (SRAM), and

RPU internal memory (RIM). In order to accelerate the data flow, a Block Buffer is designed to cache data for each RPU, and an Exchange Memory (EM) for swapping data between two RPUs.

The operation of REMUS is reconfigured dynamically at run-time according to the required hardware tasks. Configuration of REMUS includes two aspects: functional configuration of RCA and configuration of data flow in each of RPU. To support such configuration, REMUS is equipped with a Configuration Interface (CI) for each RPU. The CI is responsible for receiving and buffering configuration words, or context, which are sent from the μPU through FIFO write channel. Function of one RCA can be reconfigured as a whole by the CI. Contexts can be dynamically pre-fetched and cached by a hierarchical configuration memory (CM) which includes off-chip CM, on-chip CM, and context register files.

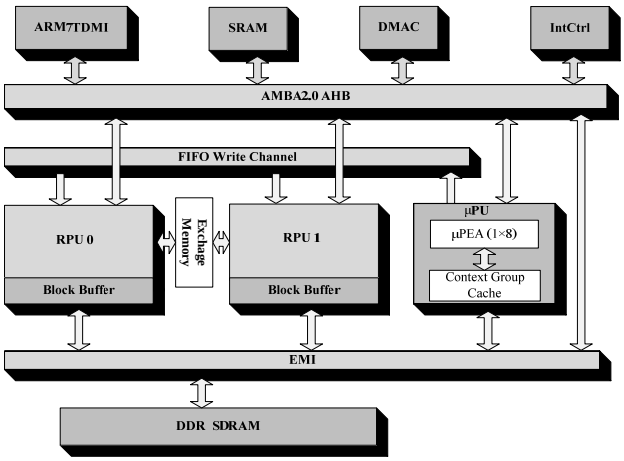


Figure 2. The overall architecture of REMUS.

According to the hierarchical architecture of the REMUS's CGRA, configuration information for the REMUS is organized in three levels: RPU-level context (CL0), RCA-level context (CL1), and Core-level context (CL2). The CL0 context consists of the information about which CL1 context will be loaded into the RPU, as well as how to communicate data between internal and external components of the RPU, etc. The CL0 is dynamically generated by the host ARM or the μP which is specified as a supervisor of the RPU. The CL1 context defines data communication of a certain RCA, and address of CL2 contexts which need to be loaded for an execution session. The CL2 context specifies particular operation of the RCA core (operation of each RC, interconnection between them, inputs, outputs, etc) as well as control parameters which control operation of the RCA core.

B. Architecture of RCA Core

The 8x8 RCA core is composed of an array of 8x8 RCs, an Input FIFO (INPUT_FIFO), an Output FIFO (OUTPUT_FIFO), two Constan_REG registers, RIM memory and a controller， etc (Fig. 3(a)). The input and output FIFO is the I/O buffer between external data flow and RCA. Each RC can get data from Input FIFO or/and Constant_REG, and store data back to Output FIFO. Interconnection among two neighboring rows of RCs is implemented by router. Through router, a RC can get results that come from an arbitrary RC in immediately previous row. The Controller generates control signals which maintain execution of RCA accurately and automatically according to configuration information in the Context Registers.

RC is basic processing unit of RCA. Each RC includes a data-path which has capability to execute signed/unsigned fixed-point 8/16-bit operations with two/three source operands, such as arithmetic and logical operations, multiplier, and multimedia application-specific operations (e.g. barrel shift, shift and round, absolute differences, etc.). Each RC also includes a register, called TEMP_REG. The register can be used either to adjust operating cycles of pipeline when a loop is mapped onto the RCA, or to store coefficients during executing a RCA core loop.

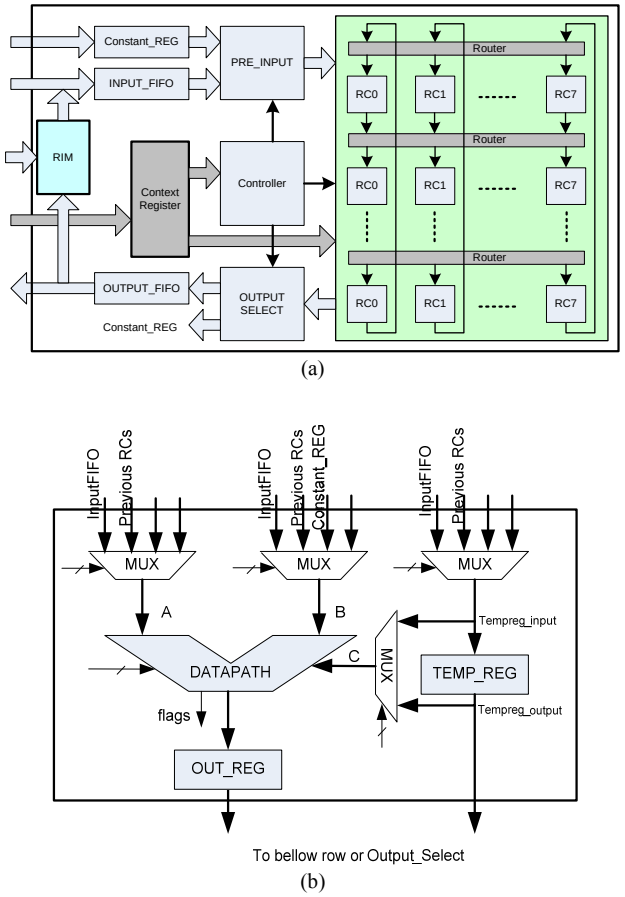


Figure 3. The REMUS's CGRA architecture: (a) RCA architecture, and (b) the structure of one RC.

C. Execution Model of RCA Core

It is a well know rule of thumb that 90% of the execution time of a program is spent in 10% of the code of LOOP constructs [15]. The architecture of RCA core is basically loop-oriented one. Executing model of the RCA core is pipelined multi-instruction-multi-data (MIMD) model. In this model, each RC can be configured separately to process its own instructions, and each row

of RCs corresponds to a stage of pipeline, multiple iterations of a loop are possible to execute simultaneously in the pipeline.

For purpose of mapping, a computation-intensive loop is first analyzed and represented by data-flow graphs (DFGs) [14]. After that, DFGs are mapped onto RCA by generating configuration information which relates to assigning operation nodes to RCs and edges to interconnection. Finally, these DFGs are scheduled to execute automatically on RCA. Once configured for a certain loop, RCA is operating as the hardware dedicated for this loop. When all iterations of loop have completed, the loop is removed from the RCA, and other loops are mapped onto the RCA.

III. COMPILING FLOW

To utilize the abundant amount of resources found in the REMUS system for exploiting inherent multilevel parallelism in applications, a compiling flow started from a high-level description of algorithm keeps an important role. Compiling flow for REMUS system must deal with many aspects of parallel computing and all its associated compiling techniques, such as computation and data partition, inter-process synchronization, and code generation, etc.

In this paper, we propose an overall compiling flow, which is based on traditional SoC (System-on-Chip) design flow, for mapping an arbitrary algorithm onto our platform of REMUS system as shown in Fig. 4. The REMUS system is a heterogeneous Systems-on-Chip (SoC) with reconfigurable hardware resources. The presence of reconfigurable resources in the system leads to the need for modifying and extending to conventional design flows with focus on the higher abstraction levels, where almost important design decisions are given. Since structure of the reconfigurable hardware has been defined, the main characteristics of the RPU have to be taken into account during HW/SW co-design to identify parts of the applications that are candidates for mapping on reconfigurable hardware. Afterwards, the compiling flow synthesizes those parts to generate configuration information (or context) instead of the phase of architecture design as in traditional SoC design flow. At this aspect, the compiling flow is quite similar to the compiling flow for software programs running on a processor. On the other hand, run-time reconfigurability of reconfigurable hardware also brings new problems to the partitioning and mapping of hardware tasks. For partition among hardware tasks instead of just considering spatial partition as it happens in traditional SoC hardware design the temporal partition and scheduling problem must be addressed. Furthermore, it also needs a mechanism, which is similar to scheduling in multi-task operating system, to handle context multiplexing, as well as inter-context data communication. This causes increase in complexity in the compiling flow, since only mapping of hardware tasks also includes the problems of hardware and software co-design.

Input of compiling flow is a description of application/algorithm in high-level language (such as C language). The compiling flow is started by a loop of two phases, *code-level transformation* and *profile*. The *profile* phase identifies the kernel loops by using the ARM profiler, whereas, *code-level transformation* modifies the code and data representations in order to expose inherent parallelism and data locality of the algorithm that are then exploited to maximize the computation performance on the target architecture, while reducing/minimizing hardware resources as well as inter-process data communication. Based on the results from previous phases, *HW/SW (Hardware/Software) partition* phase will partition the overall computation and data of the algorithm between the RPUs and the μPU. For REMUS system, we classify the input application/algorithm into three types of task: (a) the kernel loops for mapping on the RPUs, so called reconfigurable hardware tasks; (b) the control-intensive tasks and the computation task which is not suitable to implement on the RPUs, so called software tasks; and (c) the synchronization and communication tasks, simply communication tasks, which are in charge of data communication and cooperation synchronization between hardware tasks together as well as between hardware tasks and software tasks. The hardware tasks may communicate data with each other through EM memory, on-chip SRAM, and off-chip SDRAM, whereas data communication between the hardware tasks and the software is carried out via on-chip SRAM and off-chip SDRAM.

On the *low-level mapping* branch, the hardware tasks need to be further analyzed and optimized to effectively map onto the RPUs. For this purpose, each the kernel loop will be independently analyzed and optimized by *computation and data partition* phase, this phase is implemented with limitation of the available resource of only one RPU. If a loop requires amount of computing resources that is larger than the computation resource of a RPU, it must be partitioned into some sub-tasks to fit to a RPU. In contrast, some loops that require smaller amount of computing resources than the ones of RPU, are possible to joint together to map concurrently onto a RPU. The result of partition phase is a DFG (Data-flow Graph) of the task or several DFGs and DDGs (Data-Dependency Graph) of sub-tasks. This phase is implemented with support of SUIF/MatcSUIF tools [19]. The *Mappers* (i.e. Core-level and RCA-level Mapper) then generate corresponding contexts for each task/sub-task of the loop and convert them to bit-stream configuration files.

On the *high-level mapping* branch, based on computation and communication partition (i.e. data dependence between tasks), as well as configuration information for each hardware task and the required performance of application, the *Code Generation* phase determines allocation of processing resources to tasks and the order in which the tasks will be performed on RPUs

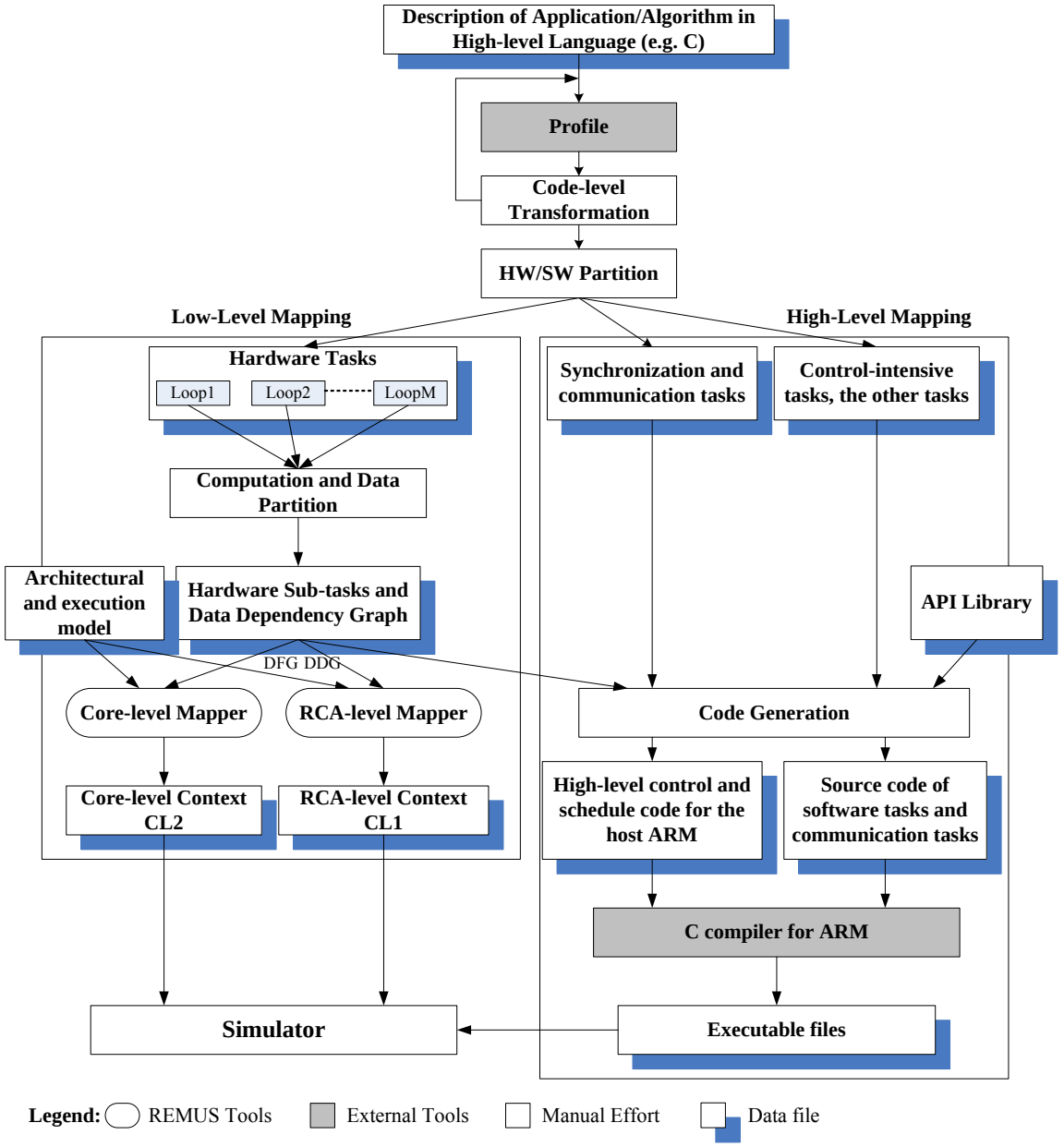


Figure 4. Compiling flow proposed for REMUS.

and/or µPU to exploit as much concurrency as possible. Besides, some communication refinements are also implemented to exploit high-level architectural features, such as the existence of hierarchical memory architecture of REMUS to increase the availability of data, or which µP is used for supervising for each hardware task and how they can communicate with others, etc. The information is then used to generate high-level control and schedule code which is compiled onto the host ARM. For purpose of synchronization and communication, the *Code Generation* also inserts communication codes into source code of software tasks and communication tasks. In the context of REMUS system, the inter-process communication is implemented by "*send*" and "*receive*" routines. Since *receive* routines stall a processor until the mail arrives, mails also serve as synchronization. Besides, for purpose of supervision, some other codes which are used for generating RPU-level configuration contexts are inserted into communication tasks. Finally, the software tasks and communication tasks will be compiled onto the µPU thereafter by ARM compiler.

We have been working in an effort to automate this compiling flow. Currently, some phases have been assisted by automatic tools (e.g. Core-level Mapper, RCA-level Mapper), whereas the others either have been developing (e.g. partitioning of a large DFG into several sub-DFGs) or still need manual effort (Code-level transformation, HW/SW Partition, and Code Generation).

IV. IMPLEMENTATION

The main challenge of full-search VBS-IME implementation is high computational complexity and huge data bandwidth. In this section, we will present application of the above proposed methodology for mapping VBS-IME algorithm, which is aimed at mobile

```
Loop1 (all current MBs)
 Loop2 (reference frames)
   Loop3 (7 size modes)
     Loop4 (partitions)
   ➔  Loop5(all search positions)
        MVCost = λ*MVbit;
        Loop6 (size of a partition)
         Compute SAD;
        endloop6
        J = MVCost + SAD;
        Compare and update min_Js, IMVs;
      endloop5
    endloop4
   endloop3
    return 41 IMVs;
  endloop2
endloop1
```

(a)

```
Loop1 (all current MBs)
 Loop2 (reference frames)
  Loop3 (all search positions)
    Compute SAD[16] of 4x4-partitions ;
loop3a Compute SAD[20] of 4x8-, 8x4-, 8x8-partitions ;
    Compute SAD[5] of 16x8-, 8x16-, 16x16-partitions ;
    Compute MVCost;
loop3b Compute J = MVCost + SAD;
    Compare and update min_Js, IMVs;
  end loop3
   return 41 IMVs;
 endloop2
endloop1
```

(b)

Figure 5. Pseudo-code for implementing full-search VBS-IME: (a) Original IME procedure, and (b) Modified IME procedure aims at efficiently mapping onto RPU and μPU of the REMUS system.

multimedia applications, onto REMUS system in detail. We focus on analyzing loops of VBS-IME, some loop transformations are used to parallelize computation-intensive parts while reducing inter-process data communication, and then mapping them onto RPU to increase total computing throughput and solve high computational complexity. Besides, some data-reuse schemes are also used to increase data-reuse ratio, hence reduce required data bandwidth. The REMUS system still has not automatic tools which aid designers in the process of HW/SW Partition and Code Generation. Therefore, to map VBS-IME algorithm onto the REMUS systems, these phases have been done manually based on heuristic method which requires experience and knowledge on the target architecture as well as algorithm of application. Several solutions are first proposed, and then validated and evaluated through simulation, and finally optimized to receive the final implementation that meets the required performance.

A. Hardware/Software Partition

The VBS-IME procedure of JM reference software [16] is presented by six loops as shown in Fig. 5(a). In order to meet the real-time constraint, these loops are transformed and some computation-intensive loops must be mapped onto reconfigurable hardware for parallel processing. Firstly, loops are reordered so that the loop5 covers loop3/4/6. After that, loop3/4/6 are unrolled and merged so that SADs of partitions of seven size modes are computed according to tree-based hierarchical architecture, i.e. SAD of sixteen 4x4-partitions are computed first, then reused to compute SADs of larger size partitions. By rearranging, we can reduce about 85% total amount of SAD computations by eliminating redundant computations. The modified procedure is shown in Fig. 5(b). Finally, HW/SW partition decides to map control of loop1 and loop 2 onto the ARM7 processor, meanwhile loop3 gets further transformation by distributing to two parts: loop3a covers all computations of SADs of 41 partitions; loop3b covers computation of MVcosts, cost function J, and decision on the minimum Js (min_Js). Because computation of MVCost includes bit-level operations, whereas, decision on the minimum Js includes many *if*-statements, so it is more efficient to map loop3b onto the μPU (denoted as T3 in Fig. 7(a)). Loop3a is mapped onto the reconfigurable hardware array as shown in the next sub-section.

B. Partition and Mapping of Hardware Tasks onto RCAs

To implement computation in the body of loop3a, the proposed VLSI designs usually compute concurrently 256 ADs (Absolute Differences in (2)) first, and then an adder-tree is used to sum up 41 SADs of partitions/sub-partitions. This process makes a continuous data-flow, and it does not require memory for intermediate data. In contrast, the REMUS just has 256 RCs, it is enough to compute 256 ADs concurrently, but for completing computation in the body of loop3a it must be implemented by some contexts. In other words, the loop3a need to be partitioned into several sub-tasks which fit on available resources of the RPU. However, the presence of multi-context gives rise to new problem about inter-context data communication. When partitioning, therefore, we have to consider two criterions: performance and amount of data exchanged among tasks, with constraint of the available (computing and storage) resources of only one RPU. Number of contexts is also taken into account during partition because large number of contexts do not only increase pressure on capacity of memory for intermediate data and configuration contexts, but also increase configuration overhead when switching among contexts. In our implementation, the body of loop3a is divided into two tasks as shown in Fig 7(a): the task T1 first computes SADs of 1x4-pixel columns (SAD_1x4) at all search positions, and then T2 is responsible for computing SADs of all partitions and sub-partitions of 7 types of block size (SAD_4x4, SAD_4x8, SAD_8x4, SAD_8x8, SAD_16x8, SAD_8x16, SAD_16x16). Actually, there are some ways to allocate function to T1 task, three ones of them is shown in table 1. In order to reuse SADs of 4x4 sub-blocks for computing larger size blocks, SAD_4x4 is the largest computation granularity that is possible to allocate to T1 task. As shown in table 1, the way that computes SAD of a 1x4-pixel column achieves the best trade-off between number

of operations and amount of data transferred to the next task. Although SAD_1x4 is selected for T1 task, but also please note that number of operations is still much larger than available computing resources (only 256 RCs) of the RPU. A full parallel, direct implementation for computation of a SAD_1x4 as shown in Fig. 6(a) is unfeasible due to too resources required. To solve this problem, we proposed a systolic-type [20] architecture that utilizes RC's capability of executing three-operand operations to compute SAD_1x4 with only four RCs as shown in Fig. 6(b). This solution does not only allow 64 SAD_1x4s of a candidate to be mapped completely onto the RPU (with only one context), but also achieve high performance (5 + (n-1) cycles for computing n candidates compare with 4+ (n-1) cycles of the solution in Fig. 6(a)). Moreover, it also fully exploits overlapping data between two successive iterations (i.e. two successive candidates in vertical direction) to reduce inner bandwidth of RPU.

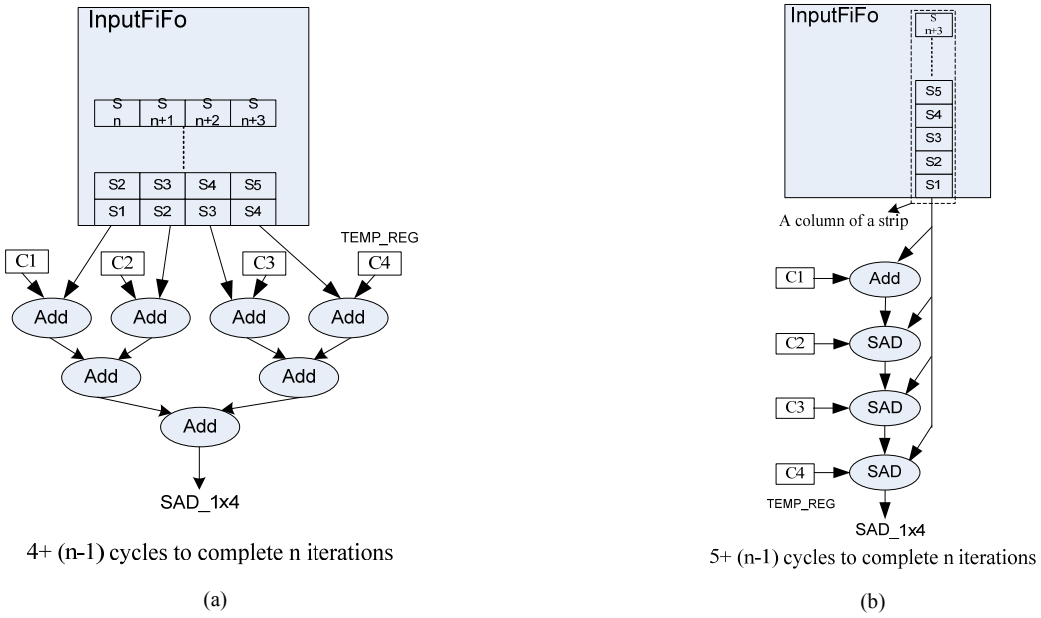


Figure 6. (a) Parallel implementation, and (b) Systolic implementation for computing SAD_1x4 of a candidate

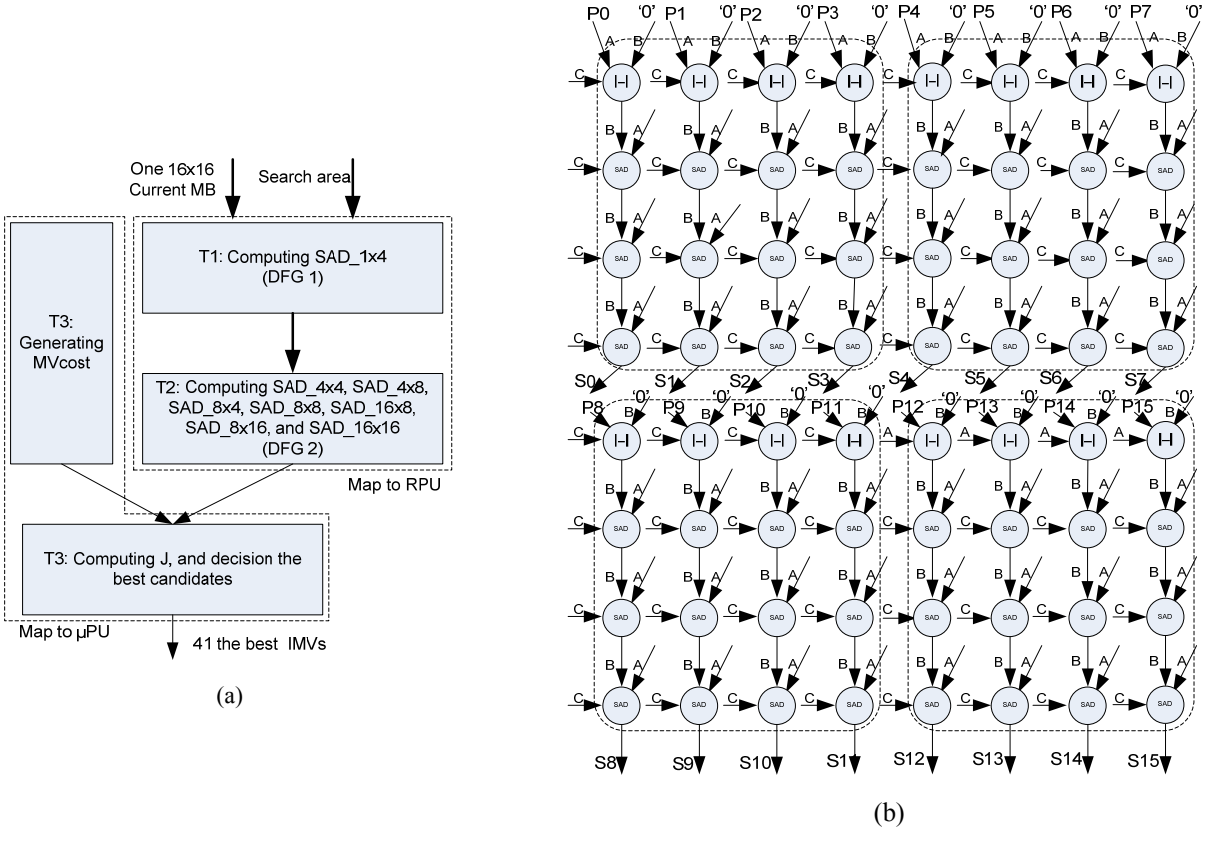


Figure 7. (a) Data dependency among tasks, and (b) DFG of the task T1 (DFG1) (for computing SAD_1x4s of one 8x8- block).

TABLE I.
VARIOUS WAYS OF T1 TASK (FOR COMPUTING A CANDIDATE)

Granularity of T1	AD	SAD_1x4	SAD_4x4
Intermediate results (16-bit word)	256	64	16
Operations	256	448	496
Required Contexts	1	1 or 2	2

Finally, because in the architecture of the REMUS each RPU is separated into four 8x8-RCAs, each 16x16-macroblock is also divided into four 8x8-blocks first, and then the corresponding computation is mapped currently onto four 8x8-RCAs for parallel processing. Fig. 7(b) shows DFG of T1 (DFG1) for computing one 8x8-block as a demonstrative example. Each DFG1 is in charge of concurrently computing sixteen values of SAD_1x4, and four such DFG1s are possible to be mapped simultaneously onto the RPU for parallel processing a complete 16x16-macroblock. The configuration and schedule of tasks are shown in Fig. 9.

C. Partitioning and Mapping of Data

Partition and mapping of data need to solve huge data bandwidth of VBS-IME algorithm by exploiting some data-reuse schemes to increase data-reuse ratio, hence reduce required data bandwidth. At the beginning of operation, four 8x8-TEMP_REG arrays are loaded 16x16 pixels of current MB, and then reconfigured to retain their content during the RCA core is running. Next, pixels of reference area are pre-fetched into the block buffer of the RPU. On the other hand, the distributed RIM modules are also exploited to increase the availability of reference pixels for operation of each RCA. Because each RCA is in charge of computing SADs for one 8x8 partition, the reference area as shown in Fig. 8 need to be read from external memory and stored in the RIM memory of each RCA. Size of the reference area, [8+2p-1, 8+2p-1] pixels, may be much larger than size of the RIM memory, so the reference area is divided into 2p strips of 8x(8+2p-1) pixels in size, i.e. each of strips correspond to one displacement step in horizontal direction, and include all displacement steps in vertical direction. At the particular period of time, only one strip is loaded to RIM.

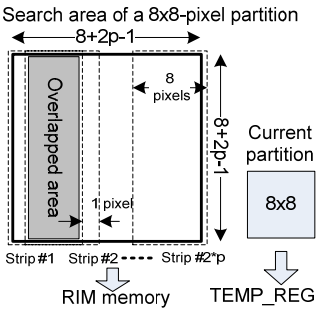


Figure 8. Input data partition for computing SAD of one 8x8-partition.

When ME process is changed from one strip to another strip, pixels of the overlapped area can be reused, and only a (8+2p-1)-pixel column of strip must be loaded. By such partition, we can reduce pressure on capacity of the RIM memory as well as amount of access to external memory. Moreover, in combination with read-while-write functionality of the RIM memory, the time spent for updating the RIM memory is hid under computing time of the RPU.

D. Schedule of Tasks

Fig. 9 shows the simplified scheme of mapping and scheduling of tasks onto the hardware resources of the REMUS platform. Execution of tasks is scheduled to implement in a three-stage pipeline: execution of task T1, execution of task T2, and execution of J computation and decision on the best candidates.

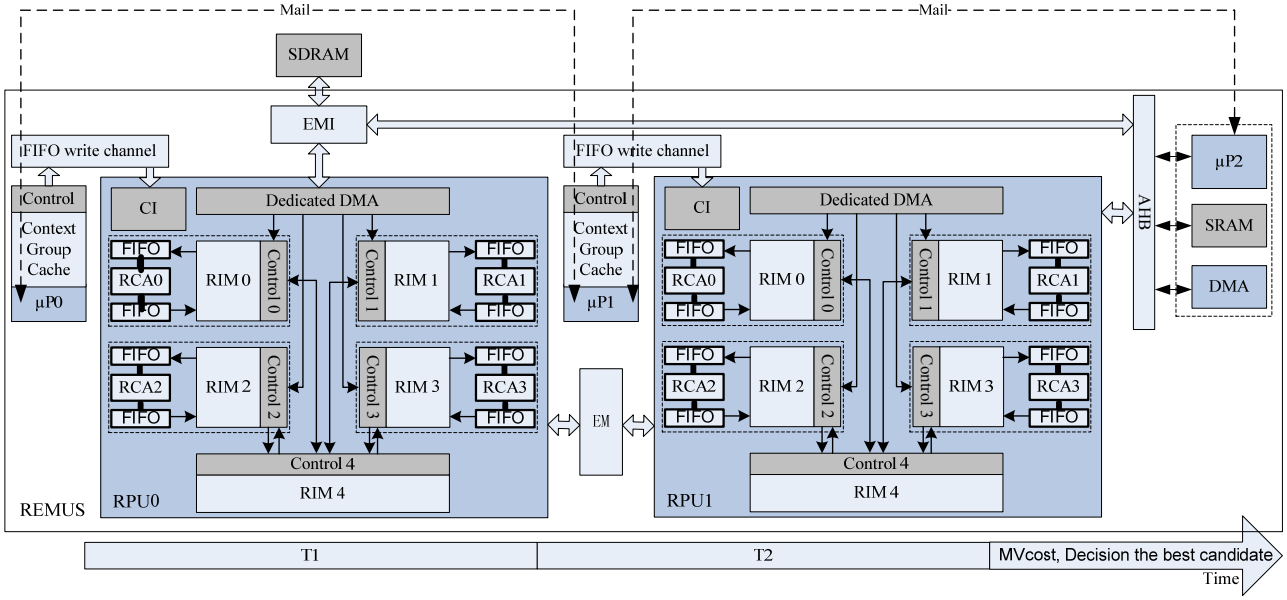


Figure 9. Mapping and Schedule of tasks on the REMUS

The first task, T1, is mapped onto {RCA0, RCA1, RCA2, RCA3} of the RPU0, whereas the task T2 is mapped onto {RCA0, RCA1, RCA2, RCA3} of the RPU1. Execution and Data-flow of RPU0 and RPU1 are reconfigured dynamically under controlling of μP0, μP1 of the μPU, respectively. Other tasks (denoted as T3 in Fig 7(a)) including MVCost generation, computation of the cost function J, and decision on the best candidates are assigned to the μP2 of the μPU. However, because MVCost generation does not depend on computation of SAD, so MVCost generation and computation of SAD was scheduled to parallelize computation. Data communication between T1 and T2 is implemented by the EM memory. Meanwhile, the on-chip SRAM are used for communicating data between T2 and T3.

Cooperation among RPU0, RPU1, and μP2 is synchronized by mailbox mechanism whose operation based on sending and receiving mails. Since receiving stalls a processor until the mail arrives, mails also serve as synchronization. We would describe handshake protocol between T1 and T2 as a demonstrative example. When T1 in progressing and a data block has been stored in EM, a mail containing address and size of data block will be sent from μP0 to the μP1. To prevent new output data from overwriting the old ones, μP0 also needs to keep the storage area occupied by this data block as "write-protected area" in EM. After receiving mail from μP0, μP1 reconfigures RPU1 to compute on the data block which is specified by information in the mail. When the whole data block in EM have been copied into RIM modules of the RPU1, even if it have not been processed by RCA cores, μP1 will sent back a mail to μP0 in order to inform μP0 that the storage area occupied by the data block in EM may be unlocked, and that it is ready to receive next data block.

V. Experimental Results

To evaluate the performance of REMUS, a functional RTL model has been designed in Verilog, and then synthesized by Synopsys Design Compiler using TSMC 65nm low power technology. The die size of REMUS is 23.7 mm^2. REMUS consumes about 194mW while working at 200MHz (the maximum frequency is 400MHz).

Let Max_N_{cycles} be the maximum number of clock cycles allowed to identify the best IMVs of 41 partitions in the VBS-IME. Define R is the frame rate, is also the rate at which video frames is processed per second.

Relation between R and Max_N_{cycles} is given by expression:

$$Max\_N_{cycles} = F_{clk}/(R*N_{MBs}), \quad (3)$$

where, F_{clk} is clock frequency of hardware system, N_{MBs} is the number of macro-blocks per frame.

Also, define the experimental N_{cycles}, or simply N_{cycles}, is number of clock cycles required to identify the best IMVs of 41 partitions by experimentation. Since the used VBS-IME algorithm is full-search one, the number of search points is constant for every MB, thereby N_{cycles} only depends on method for implementing VBS-IME and the supported search range p, but not depend on characteristics of video sequence.

To validate the functionality of our implementation, we also compare the results of our implementation with the JM reference software [16]. A co-simulator model as shown in Fig. 10 is used for this purpose. Experiment shows that our implementation completely matches with the result that is computed by the JM reference software. Because of target at mobile multimedia applications, some performance evaluations are implemented on QCIF, CIF, 4CIF, STDV video sequences at frame rate 30fps by RTL simulation. The experimental results are shown in table 2. As shown in table 2, N_{cycles} required to complete VBS-IME computation for a macro-block is 320 cycles and 1152 cycles with search range p = 8 and p = 16, respectively, it completely satisfies for encoding up to SDTV video sequences with both p = 8 and p =16.

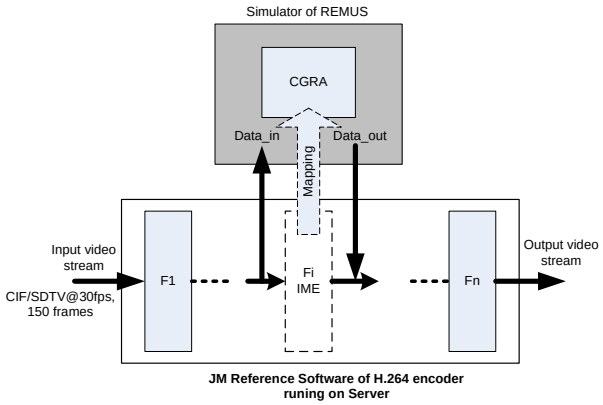


Figure 10. Co-simulation modeling for evaluating functionality

TABLE II.
Performance Evaluation (F_{CLK} = 200MHz, R = 30fps)

Frame size	Max_N_{cycles}	Experimental N_{cycles}	
		P=8	*P=16*
QCIF	67340	320	1152
CIF	16835		
4CIF	4208		
SDTV	4938		

Table 3 shows performance comparison between our implementation and the experimental results of Yap [3], Ou [4], LU [6], and Ebeling [7]. Computation performance achieved by our implementation on the REMUS system is better than that achieved by ASIC architecture in [3, 6] and CGRA architecture in [7]. However, it is not as good as the result achieved by ASIC architectures in [4]. The reason is that: firstly, the authors in [4] just use SAD criterion (as in (2)) for selecting the best motion vector instead of Rate-Distortion Optimized (RDO) criterion (as in (1)) that is used in JM software as well as in our implementation; and secondly, the architecture in [4] are ASIC-based architecture which designed specially in order to dedicate to VBS-IME algorithm, achievement of high performance is advantage

of ASIC approach. Although the performance is not as high as that in [4], our implementation is dominant at flexibility due to the reconfigurability and programmability of the REMUS system. In other words, the goal of our implementation is trade-off between flexibility and performance. It supports a lot of different mapping scheme depending on requirements of each application. In case higher performance is required, mapping schemes with fast VBS-IME algorithms (e.g. successive elimination algorithm (SEA) [17], diamond, UMHexagon [18], etc.) may be used.

TABLE III.
PERFORMANCE COMPARISON (P=8)

Architecture	ASIC			CGRA	
	[3]	*[4]*	*[6]*	*[7]*	***REMUS***
Supported Block Sizes	all	all	all	16x16	all
Evaluating Criterion	SAD	SAD	SAD	SAD	RDO
Experimental N_{cycles}	4096	256	528	5144	320
F_{clk} (MHz)	294	200	345	100	200

VI. CONCLUSION

VBS-IME is one of some tools which have the highest computational complexity and the largest memory access bandwidth in H.264/AVC encoder. An efficient implementation of IME module has an important role in successful implementation of whole H.264 encoder. The paper has presented methodology for mapping a high complex algorithm as full-search VBS-IME algorithm in detail after having proposed an overall methodology which generally may be applied for mapping any algorithm onto the platform of REMUS system. After mapping, we have simulated to validate both functionality and timing of whole implementation. Our mapping methodology exploits data reuse and computation parallelism of the algorithm not only to increase total computing throughput and solve high computational complexity, but also to reduce number of configuration switches and inter-process data communication between tasks together. Experiments in mapping full-search VBS-IME algorithm onto the REMUS system demonstrate that complex applications can be mapped with competitive performance on the REMUS platform. Performance evaluation shows that the REMUS system can perform VBS-IME at real-time speed for CIF/SDTV@30fps video sequences with two reference frames and a maximum search range of [-16, 15]/[-8, 7], respectively. The implementation, therefore, can apply for H.264/AVC encoder in mobile multimedia applications such as video recorder, video telephone, video conference, etc.

To my knowledge, this is first time partitioning and mapping of the full-search VBS-IME algorithm is reported in the literature. The previous works mainly focused on proposing ASIC architectures that implement directly SAD computation as described by (2). These works do not require a lot of effort to partition HW/SW. In contrast, because the REMUS system is designed with computing resources for a range of applications, so it takes a lot of effort to partition task of SAD computation into several sub-tasks which fit on available computing resources of the REMUS system, and then give them a proper scheduling scheme to synchronize their cooperation.

In the future, to achieve higher performance in implementation of VBS-IME algorithm on the REMUS system, some aspects such as task partition, DFG of each sub-task, scheduling scheme, etc., will continue to be optimized. The result of implementation then will to be used to build a complete H.264 encoder for mobile multimedia applications. Experience achieved from mapping algorithm also will to be applied for developing automated compiling tools of the REMUS system.

ACKNOWLEDGMENT

This work was supported by the National High Technology Research and Development Program of China (863 Program) (grant no.2009AA011700).

The authors would like to thank to M.Zhu, C.MEI, B.LIU, JJ.YANG, J.XIAO, and GG.GAO for their helpful discussions and technical support.

REFERENCES

[1] Iain E. Richardson: *The H.264 advanced video compression standard*, second edition, 2010, John Wiley & Sons, Ltd.

[2] Yu-Wen Huang; Tu-Chih Wang; Bing-Yu Hsieh; Liang-Gee Chen: *Hardware architecture design for variable block size motion estimation in MPEG-4 AVC/JVT/ITU-T H.264,* Proceedings of the International Symposium on Circuits and Systems, 2003. IEEE.

[3] S.Y. Yap, J.V. McCanny, *A VLSI architecture for variable block size video motion estimation*, IEEE transactions on circuits and systems-II: express briefs, VOL.51, No.7, July 2004.

[4] C. M. Ou, C. F. Lee, and W. J. Hwang, "An Efficient VLSI Architecture for H.264 Variable Block Size Motion Estimation," IEEE Transactions on Consumer Electronics, vol. 51, no. 4, pp. 1291–1299, Nov. 2005.

[5] Ruchika Verma, Ali Akoglu: *A Coarse grained and hybrid reconfigurable architecture with flexible NOC router for variable block size motion estimation*, 2008, IEEE.

[6] L. LU, J.V. MCCANNY, S. Sezer.: "Reconfigurable system-on-a-chip motion estimation architecture for multi-standard video coding", IET Comput. Digit. Tech., 2010, Vol. 4, Iss. 5, pp. 349–364.

[7] Carl Ebeling, Darren C. Cronquist, Paul Franklin, Jason Secosky, and Stefan G. Berg: "Mapping Applications to the RaPiD Configurable Architecture, FPGAs for Custom Computing Machines", 1997, IEEE.

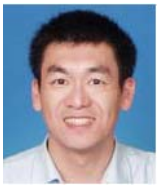
[8] Sudang Yu, Leiba Liu, Shaajun Wei: "Automatic Contexts Switch in Loop Pipeline for Embedded Coarse-grained Reconfigurable Processor", International Conference on Communications, Circuits and Systems, 2008, IEEE.

[9] Francisco-Javier Veredas, Michael Scheppler, Will Moffat, Bingfeng Mei: *Custom implementation of the coarse-grained reconfigurable ADRES architecture for multimedia purposes*, IEEE, 2005.

[10] H. Singh, M. H. Lee, G. Lu, et al.: "MorphoSys: an integrated reconfigurable system for data-parallel and

computation-intensive applications," Computers, IEEE Transactions on, vol. 49, pp. 465-481, 2000.
[11] X. Technologies, "XPP-III Processor Overview", White Paper, July 13 2006.
[12] B. Mei, S. Vernalde, D. Verkest, et al., "ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix," 2003, pp. 61-70.
[13] M. Zhu, L. Liu, S. Yin, et al.: "A Cycle-Accurate Simulator for a Reconfigurable Multi-Media System," Ieice Transactions on Information and Systems, vol. 93, pp. 3202-3210, 2010.
[14] Gajski, D., Dutt, N., Wu, A., Lin, S.: *High-Level Synthesis, Introduction to Chip and System Design*, Kluwer Academic Pub. (1992).
[15] Kiran Bondalapati, Viktor K. Prasanna: *Mapping loops onto reconfigurable architectures*, International workshop on field programmable logic, September, 1998.
[16] JM reference software, http://iphome.hhi.de/suering/tml/.
[17] W. Li and E. Salari, "Successive Elimination Algorithm for Motion Estimation," *IEEE Trans. Image Processing*, vol. 4, no. 1, 1995, pp. 105–107.
[18] Chen, Z., Zhou, P., & He, Y.: *Fast integer pel and fractional pel motion estimation for JVT*, JVT-F017, 6th JVT Meeting, Awaji, December, 2002.
[19] http://suif.stanford.edu/.
[20] H. T. Kung and P. L. Lehman: "Systolic (VLSI) Arrays for Relational Database Operations", *Proc. ACM-Sigmod 1980 Int'l Conf. Management of Data*, p.105, 1980.

Kiem-Hung Nguyen received the B.S. and M.S. degrees in electronic engineering from Vietnam National University, Hanoi, Vietnam, in 2003 and 2005, respectively. He is currently working to get the Ph.D. degree in electronic engineering at Southeast University, Nanjing, China. His research interests mainly include multimedia processing, reconfigurable computing and SoC designs.



Peng Cao received the B.S., M.S. and Ph.D degrees in Information Engineering and Electrical Engineering from Southeast University in 2002, 2005 and 2010 respectively. His research interests mainly include digital signal and image processing, image/video compression, reconfigurable computing and related VLSI designs.

Xue-Xiang Wang was born in Xinjiang Province, China, in 1972. She received the M.S. degree in electronics in 2001 from the Southeast University of China. She is currently pursuing the Ph.D. degree with the Southeast University. Her research interests include SoC design, memory subsystem.