Hindawi Publishing Corporation
VLSI Design
Volume 2012, Article ID 413747, 14 pages
doi:10.1155/2012/413747

Research Article

An Efficient Multi-Core SIMD Implementation for H.264/AVC Encoder

M. Bariani, P. Lambruschini, and M. Raggio

Department of Biophysical and Electronic Engineering, University of Genova, Via Opera Pia 11 A, 16145 Genova, Italy

Correspondence should be addressed to P. Lambruschini, lambruschini@dibe.unige.it

Received 18 November 2011; Revised 20 February 2012; Accepted 3 March 2012

Academic Editor: Muhammad Shafique

The optimization process of a H.264/AVC encoder on three different architectures is presented. The architectures are multi- and singlecore and SIMD instruction sets have different vector registers size. The need of code optimization is fundamental when addressing HD resolutions with real-time constraints. The encoder is subdivided in functional modules in order to better understand where the optimization is a key factor and to evaluate in details the performance improvement. Common issues in both partitioning a video encoder into parallel architectures and SIMD optimization are described, and author solutions are presented for all the architectures. Besides showing efficient video encoder implementations, one of the main purposes of this paper is to discuss how the characteristics of different architectures and different set of SIMD instructions can impact on the target application performance. Results about the achieved speedup are provided in order to compare the different implementations and evaluate the more suitable solutions for present and next generation video-coding algorithms.

1. Introduction

In the last years the video compression algorithms have played an important role in the enjoying of multimedia contents. The passage from analog to digital world in multimedia environment cannot be performed without compression algorithms. DVDs, Blu-Ray, and Digital TV are typical examples. The compression algorithm used in DVDs is MPEG-2, and Blu-Ray supports VC-1 standardized with the name SMPTE 421M [1], in addition to MPEG-2 and H.264. In the digital television, the compression algorithms are used to reduce the transmission throughput. In DVB-T, the picture format for DVD and Standard Definition TV (SDTV) is 720×576, and this resolution is the most used in digital multimedia contents. The most recent standards for digital television as DVB-T2 and DVB-H support H.264/MPEG-4 AVC for coding video.

The H.264/AVC [2] video compression standard can cope with a large range of applications, reaching compression rate and video quality levels never accomplished by previous algorithms. Even if the initial H.264/AVC standard (completed in May 2003) was primarily focused on "entertainment-quality" video, not dealing with the highest video resolutions, the introduction of a new set of extensions in July 2004 covered this lack. These extensions are known as "fidelity range extensions" (FREext) and produced a set of new profiles, collectively called High Profiles. As described in [3], these profiles support all the Main Profile features and introduce additional characteristics such as adaptive transform block-size and perceptual quantization scaling matrices. Experimental results show that, when restricted to intra-only coding, H.264/AVC High Profile outperforms the state-of-the-art in still-image coding represented by JPEG2000 on a set of monochrome test images by 0.5 dB average PSNR [4].

It results that a H.264 encoder addressing high definition (HD) resolutions needs to support High Profiles in order to be part of an effective video application. On the other hand, the already great complexity of the H.264 algorithm is further increased by supporting FREext. In particular, this leads to implement two new modules: the 8×8 intraprediction and the 8×8 transform.

In case of mobile devices, the H.264 complexity issues together with the constraints of limited power consumption and the typical need of real-time operations in video-based applications draw a difficult scenario for video application developers.

The HD resolution involves a large amount of data, and the compression algorithms are high computational demand applications, often used as benchmark to measure the processor performance. In order to support real-time video encoding and decoding, specific architectures are developed. Multicore architectures have the potential to meet the performance levels required by the real-time coding of HD video resolutions. But in order to exploit multicore architectures, several problems have to be faced. The first issue is the subdivision of an encoder application in modules that can be executed in parallel. In this case, the main difficult is the strong data dependency in video encoder algorithms. Parallel architectures can be more easily exploited using other kind of algorithm like computer graphics, rendering technology or cryptography, where the data dependency is not as strong as in video compression. Once a good partitioning is achieved, the optimization of a video encoder should take advantage of the data level parallelism to increase the performance of each encoder module running on the architecture's processing element. A common approach is to use the SIMD instructions to exploit the data level parallelism during the execution; otherwise, ASIC design can be adopted for critical kernel. SIMD architectures are widely used for their flexibility. SIMD ISAs are added at most market spread processor: Intel's MMX, SSE1, SSE2, SSE3, SSE4; Amd 3DNow!; ARM's NEON; Motorola's AltiVec (also known as Apple's Velocity Engine or IBM's VMX).

In this paper, we will show how the data level parallelism is exploited by SIMD and which instructions are more useful in video processing. Different instruction set architectures (ISAs) will be compared in order to show how the optimization can be driven and how different ISA features can lead to different performance. This paper is intended to be a great help to both software programmers that have to choose for the most suitable SIMD ISA for developing a video-based application and for ISA designers that want to create a generic instruction set being able to give good performance on video applications. In that regard, the authors will select a set of generic SIMD instructions that can speed up video codec applications, detailing the modules that will profit from the introduction of each instruction. Besides describing the optimization methods, the paper indicates a few guidelines that should be followed to partition the encoder in separate modules.

Even though the work focuses on H.264/AVC, most of the proposed solutions will also apply to the earlier mentioned standards as well as to more recent video compression algorithms as scalable video coding (SVC) [5]. Moreover, H.264/AVC tools will have a fundamental role in the emerging high efficiency video coding (HEVC) standardization project [6].

This paper is organized as follows. Section 2 gives an overview of the state-of-the-art SIMD-based architectures, giving particular attention to those targeting video-coding applications. A brief description of the three architectures used for the presented project is given in Section 3. Section 4 describes the H.264 optimized encoder, focusing on module partitioning and SIMD-based implementation. The performance results of both the C-pure implementation and the SIMD version are given in Section 5 together with an explanation about what are the key instructions for optimizing a video codec. Finally, the conclusion is drawn in Section 6.

2. Related Works

The basic concept of SIMD instructions is the possibility of fill vector registers with multiple data in order to execute the same operation on several elements. One of the major bottlenecks in the SIMD approach is the overhead due to the data handling needed to feed the vector registers. Typical required operations are extra memory accesses, packing data, element permutation inside vectors, and conversion from vector to scalar results. All these preliminary operations limit the vector dimension and the performance enhancement achievable with SIMD optimization.

In literature, several studies regarding the SIMD optimization of video-coding applications are available [7–11]. The scope driving these studies is the achievement of the maximum performance, adopting measures in order to reduce the known bottlenecks. Since its standardization, SIMD optimizations targeting the H.264 algorithm have been proposed as can be seen in [12, 13]. However, both the works only address the H.264 decoder and present a MMX optimization starting from the H.264 reference code. Besides addressing a more complex application, our aims were also to discuss how the characteristics of different architectures and different set of SIMD instructions can impact on the encoder performance.

In SIMD processors the memory access has an important impact on performance. The unaligned access is not usually possible in SIMD ISA, and when possible it is discouraged due to additional instruction latency. The programmers usually take care of handling the unaligned load adding further overhead to vector data organization. Moreover, the need of unaligned load is always present in video-coding algorithm especially in motion estimation (ME) and motion compensation (MC), where the pixel blocks selected by motion vectors are frequently at misaligned positions even if the start of a frame is memory aligned. Often, the position of a block we need to access cannot be known in advance, and this leads to unpredictable misalignment in data loaded from memory. In Intel's architectures, starting from SSE2 the support to unaligned load has been added, but the performance is strongly reduced either if the load operation crosses the cache boundary or, with SSE3, if the load instruction needs store-to-load forwarding. In AltiVec, it is necessary to load two adjacent positions and shift data in order to achieve one unaligned load, a usually adopted approach to overcome the misalignment access issue. This problem is common in digital signal processor (DSP) as well. Usually, DSP do not support unaligned loads, but due to the large use of DSP in video application several producers have added the support to this kind of operation. For example, Texas Instruments family TMS320C64x supports unaligned load and store operations of 32 and 64 bit element, but with only one of the two memory ports [14].

The MediaBreeze SIMD processor was proposed to reduce the bottlenecks in SIMD implementations [15]. The Breeze SIMD ISA uses a multidimensional vector able to speed up nested loops but at the cost of a very complicated instruction structure requiring a dedicated instruction memory. In [16], a specific SIMD ISA named VS-ISA was proposed in order to improve performance in video coding. The authors adopted specific solutions for sum of absolute difference (SAD), not aligned load applied to ME, interpolation, DCT-IDCT, and quantization dequantization.

Another typical approach to reduce the SIMD overhead is the usage of multibank vector memory where data is stored interleaved. The drawback is the increase of hardware cost for supporting the addresses generation.

An alternative to SIMD implementation on programmable processor architectures is the hardwired processor. Usually, it is only used when performance and low power consumption are essential requirements [7, 14, 17]. In fact, the lack of flexibility typical of hardwired processors reduces their applicability to a narrow segment of the market, where the programmability is either not required or considerably reduced.

3. SIMD ISA Description

In order to optimize the H.264 encoder, we chose three different ISAs. The adopted architectures are ST240, xSTream, and P2012, all developed by STMicroelectronics. The former is a single-processor architecture, and the others are multicore platforms. In the following, the three architectures will be briefly described, giving special attention to the SIMD instruction set.

We chose these architectures for their novelty and for the possibility to have a complete toolchain (code generation, simulation, profiling, etc.) for developing an application in an optimal way. Each toolchain allowed a complete observability of the system. In this way, it was possible to evaluate the effectiveness of every author's solution. Observability is a very important characteristic when developing/optimizing an application. Using a real system it is not always possible to reach the degree of observability you have using a simulator and a suitable toolchain. Moreover, in an architecture under development as P2012 we had the possibility to contribute to the SIMD instruction set and, more important, to evaluate the contribution of each particular SIMD to the performance of the target video codec application. The three instruction sets present suitable characteristics for our research; they are generic instruction set, but ST240 includes a few video-specific instructions; we can analyse the impact of different vector register sizes; even if xSTream and P2012 share many characteristics, only xSTream supports horizontal SIMD (this is a special feature; e.g., other SIMD extensions as Intel SSE and ARM NEON do not have the same support); in P2012 platform, we were able to define and insert new SIMD instructions.

Besides the type of instructions, the SIMD extensions differ in both size and precision. These differences allow analyzing the impact of different architecture solutions on the global performance.

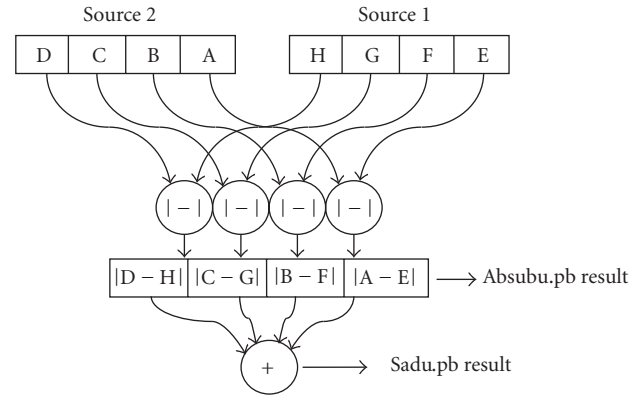

Figure 1: SAD operation.

3.1. ST240. The ST240 is a processor of STMicroelectronics ST200 family based on LX technology jointly developed with Hewlett Packard [18, 19]. The main ST240's features are the following:

(i) 4-issue Very Long Instruction Word (VLIW)

(ii) 64-32-bit general purpose registers

(iii) 32KB D-Cache and 32KB I-Cache

(iv) 450 MHz clock frequency

(v) 8-bit/16-bit arithmetic SIMD.

In the H.264 encoder SIMD optimization, the most significant instructions of the ST240 ISA are the following: the SIMD add.ph and sub.ph which perform, respectively, the packed 16-bit addition or subtraction; the perm.pb instruction which performs byte permutations and the muladdus.pb which multiplies an unsigned byte by a signed byte in each of the byte lanes and then sums across the four lanes to produce a single result. Furthermore, several data manipulation instructions are defined: pack.pb packs 16-bit values to byte elements ignoring the upper half; shuffeve.pb and shuffodd.pb, respectively, perform 8-bit shuffle of even and odd lanes. Two averaging operations (avg4u.pb and avgu.pb) are also defined in the instruction set.

One important operation in video-coding algorithms, the absolute value of the difference, abs (a-b), can be performed with the absubu.pb instruction (Figure 1) which works on each byte lane (treating each byte lane as an unsigned value) and returns the result in the corresponding byte lane of the destination register. The sadu.pb (Figure 1) performs the same operation and then sums the byte lanes value and returns the result.

3.2. xSTream. xSTream is a multiprocessor dataflow architecture for high-performance embedded multimedia streaming applications designed at STMicroelectronics [20, 21].

xSTream is constituted by a parallel distributed and shared memory architecture. It is an array of processing elements connected by a Network on Chip (NoC) with specific hardware for management of communication [22], as depicted in Figure 2.

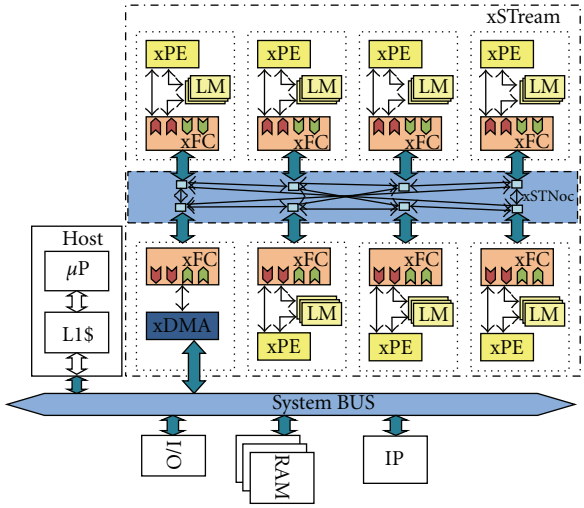

Figure 2: xSTream architecture.

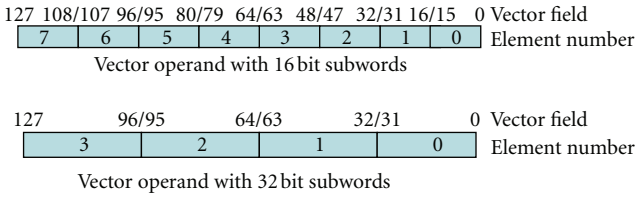

Figure 3: Vector operand.

The main elements in Figure 2 are the general purpose engine, the xSTreaming Processing Engines (XPEs) and the NoC interconnecting all components.

The XPEs are based on ST231 VLIW processors [22] of ST200 STMicroelectronics family [18, 19]. The main features can be resumed as

(i) 2-issue VLIW,

(ii) 128-bit vector registers,

(iii) up to 512 KB local memory cache,

(iv) up to 1 GHz clock frequency, and

(v) 16-bit/32-bit arithmetic SIMD.

In order to achieve excellent performance, the XPE core tries to exploit available parallelism at various levels. It supports a plethora of SIMD instructions to exploit available data-level parallelism. These instructions concurrently execute up to four operations on 32-bit operands or eight operations on 16-bit operands. The core supports wide 128-bit load/store.

The xSTream architecture handles scalar and vector operands.

Vector operands are 128-bit wide and consist of either eight 16-bit half-words or four 32-bit words, as shown in Figure 3.

In the xSTream ISA each SIMD instruction has an additional operand allowing permuting the result's element positions or replicating any element in the other positions.

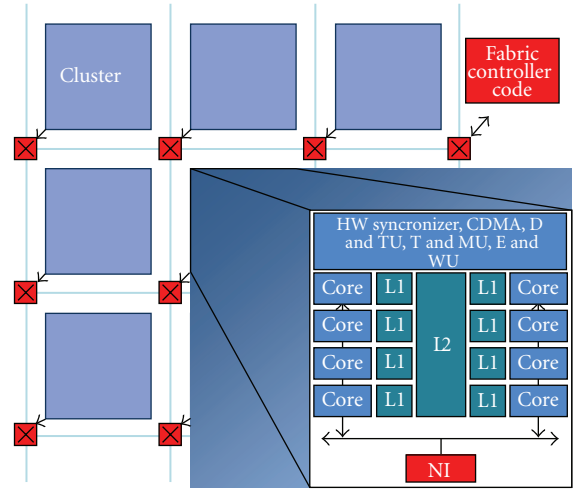

Figure 4: P2012 scheme.

This feature considerably increases the SIMD flexibility because the results have often to be reordered for further elaboration. This is especially true for video-coding algorithm with operations performed on several steps where the input of next step is usually the output of previous one. The permutation operand allows this with the cost of only one additional cycle. This leads to reduced costs to perform all the operation needed for data reordering.

The XPE supports horizontal SIMD as well. This kind of SIMD allows operations among elements in the same vector, and it is a key feature for speeding up execution in several H.264 functional units, as we will see in next sections.

3.3. Platform 2012 (P2012). Platform 2012 is a high-performance programmable architecture for high computational demanding embedded multimedia applications, currently under joint development by STMicroelectronics and Commissariat à l'énergie atomique et aux énergies alternatives (CEA) [23]. The goal of P2012 platform is to be reference architecture for next generation of multimedia product.

The P2012 architecture (Figure 4) is constituted by a large number of decoupled clusters of STxP70 processors interconnected by a Network on Chip (NoC). Each cluster can contain a number of computational elements ranging from 1 to 16. The main features of the STxP70 processor element are as follows:

(i) 32-bit RISC processor (up to 2 instructions per cycle),

(ii) 128-bit vector registers,

(iii) 256 KB of memory shared by all the processors (per cluster),

(iv) 600 MHz clock frequency, and

(v) 16-bit/32-bit arithmetic SIMD.

The P2012 basic modules can be easily replicated to provide scalability [24]. Each module is constituted by a computing cluster with cache memory hierarchy and a communication engine. The STxP70 is dual issue application-specific

instruction-set processor (ASIP) [25] with domain-specific parameterized vector extension named VECx. STxP70 SIMD instructions are used to exploit available data level parallelism [26]. These instructions execute in parallel up to four operations on 32-bit operands or eight operations on 16-bit operands, while 128-bit load/store is supported.

Vector operands are 128-bit wide and consist of either eight 16-bit half-words or four 32-bit words. In order to increase the SIMD flexibility, instructions able to permute data positions inside the vector operands are defined in the instruction set. The support to horizontal SIMD is limited at operation involving only two adjacent elements inside a vector, but its presence is fundamental for typical video-coding operation like sum of absolute difference (SAD).

3.4. SIMD Instruction-Set Evaluation. Whatever platform we choose, we will have a limited number of SIMD instructions because of hardware constraints. For this reason, besides precision and size, one of the key issues while choosing a SIMD extension is generality versus application-specific instructions. The former can show good speedups for a large variety of applications. The latter can reach greater performance, but limited to a particular family of applications. Of course, there are a lot of solutions that lay in the middle.

The vector register size impacts performance, hardware reliability, and costs. The choice of the optimal size and precision of SIMD instructions is a key factor for reaching the desired performance for the target application. The axiom larger SIMD equal to better performance may be valid for applications having no constraints and data dependencies in either spatial or temporal field. It is not the H.264 encoder condition. In general, algorithms with a heavy control flow are very difficult to vectorize, and the SIMD optimization does not always lead to the desired performance enhancement.

The application developers should choose the dimension that best fit their needs, as well as ISA designers should take into account the requirements of the application families they are targeting. As stated in [7], in a processor designed to handle video-coding standards for which the theoretical worst-case video sequence will consist of a large number of 4×4 blocks, four-way SIMD parallelism makes full use of data paths. In this case, increasing the size will lead to little performance improvement. In contrast, if we focus on the H.264's fidelity range extensions, with their 8×8 transform and 8×8 intraprediction, an ISA with eight-way SIMD parallelism will yield to better performance. Next generation video-coding standards like HEVC will use wider ranges of block sizes for both prediction and transformation processes, making the choice of the optimal vector register size even more complicated.

4. H.264 Encoder Implementation

4.1. Software Partitioning. In order to support real-time video encoding addressing HD resolutions, multiprocessor architectures seem to be an optimal solution, as earlier explained. Moreover, we would like to test the multicore architectures with an application of high interest but not so suitable for these kind of architectures in order to stress the architecture design and to evaluate possible issues and finding solutions that could be also useful for other applications.

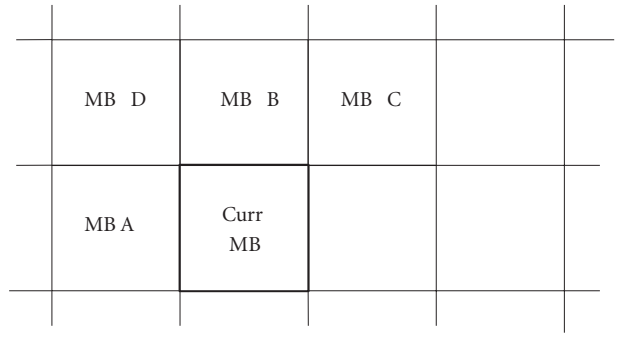

Figure 5: MB neighbours.

The first programmer's task dealing with this type of platforms is the subdivision of the encoder application in modules that can execute in parallel. The H.264 encoder partitioning plays a fundamental role in multicore architectures as xSTream and P2012, where each functional block has to meet the resources of processor elements, and the interconnection system must fulfil the memory bandwidth needed to feed the modules. The designer choice becomes more complex when some modules can run in parallel avoiding stalls in pipeline [26].

Even if a detailed description of the encoder partitioning is beyond the scope of this paper, we can here depict some issues we faced approaching this process and the solutions we adopted.

First of all, it is worth to take into account the data dependency inside the H.264 encoder. Temporal data dependency is implicit in the Motion Estimation mechanism; the coding of the current frame always depends on the previously encoded frame(s) that are used as reference. Thus, there is always a temporal data dependency, except if the current frame is an I picture. Anyway, the encoding process also shows a spatial data dependency between macroblocks, that is, the basic encoding block comprising 16×16 pixel elements. While coding the current macroblock (MB), we need data from the previously encoded MBs belonging to the same frame, or, to be more precise, to the same slice (a sequence of MBs in which the frame can be segmented). Figure 5 shows the current MB together with the already reconstructed neighbours that are needed for its prediction. Specifically, MB A, B, C, and D are required for intraprediction, motion vector prediction, and spatial direct prediction (in the SVC-compatible version). Furthermore, MB A and B are used to check the skip mode in P frames.

Spatial data dependency can even occur inside a MB. The prediction of a 4×4 block may depend on the results of already-predicted neighbouring blocks. e.g., this occurs in Intra 4×4 or in the deblocking filter.

In this scenario, we cannot encode two frames in parallel, because of the temporal data dependency, and we cannot concurrently process different MBs, because of the spatial

Table 1: Encoder data flow.

Module	Input	Output	Input from local buffers
MV prediction	MV A	MV pred	MV B, C, D
Motion estimation	Search window, original MB, MV predictor	MV, cost, best intermode, MB predictor	
Intraprediction	Original MB, reconstructed MB A	Cost, best intramode, MB predictor	Reconstructed MB B, C, D
Residual coding	Original MB, MB predictor	Residual signal, coded MB parameters	
IDCT-DeQuant and reconstruction	Residual signal	Reconstructed MB	
Deblocking filter	Reconstructed MB A	Decoded MB	Reconstructed MB B
Entropy coding	Coded MB parameters	Output stream	

data dependency, unless the MBs belong to different slices. Thus, one opportunity is to concurrently process every slice, but this solution has two drawbacks; it strongly depends on the particular encoder configuration, and it requests to implement the whole encoder on every processor element. Therefore, the only chance to partition the encoder is during the MB processing. This does not mean to separately process 8×8 or 4×4 blocks, but to separately execute the encoder functional units at MB level.

The encoder partitioning should now derive from an evaluation of the functional units that can be concurrently computed, taking into account the amount of data that needs to be exchanged between the different cores.

If we suppose that each module will run on a different core, we must consider both the chunk of data each core needs to exchange with the interconnected cores and the frequency of such communications. Therefore, for an optimal module partitioning, it is important to analyse the encoder data flow. Basically, this analysis should result in a list of selected modules with a set of input and output data for every list's entry, as shown in Table 1. The Figure 5's notation is used to indicate the neighbouring MBs. This table allows identifying the dependencies between modules as well as the data flow, from which we can obtain the requested bandwidth for the communication mechanism between processor elements. This preliminary analysis also produces the partition diagram, shown in Figure 6.

Each module will keep local memory buffers containing the data required to process the current MB. For example, the Intraprediction module needs to store a row of reconstructed MBs plus one MB (the left MB) in order to be able to predict the current MB. The deblocking filter will need to store the same number of reconstructed MBs as well. These local storages are filled by producer modules as soon as they complete the respective tasks. In the previous example, "IDCT-DeQuant & Reconstruction" is the producer for intraprediction; when the MB reconstruction has completed for MB_n, the intraprediction of MB_{n+1} can start. It is worth noting that the intraprediction of MB_{n+1} can be concurrently executed with the motion estimation of MB_{n+1} and the deblocking filter of MB_n.

For the sake of simplicity we did not put into Figure 6 scheme all the project components. The buffer mechanism for passing reference-frame data to the ME and the decoded

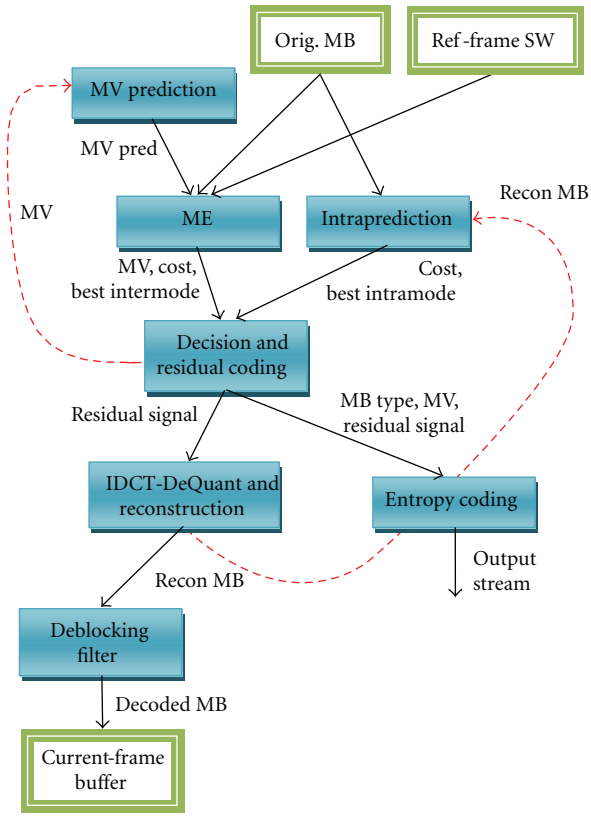

Figure 6: Encoder partition diagram.

picture buffer are not described. We preferred to focus on the encoder data flow in order to highlight the chances for module parallelisation. Moreover, the buffering mechanisms strongly depend on the architecture design implementation.

The here described partitioning seems to both fulfil the data dependency constraints and exploit the few opportunities of parallel execution available in a H.264 encoder. Moreover, the computational weight of the encoder components is quite well distributed among the different cores. The only exception is the ME, which is the most time-consuming module. In our encoder we utilize the SLIMH264 ME algorithm [27]. SLIMH264 is divided into two different stages: the first phase is common to all the partitions and

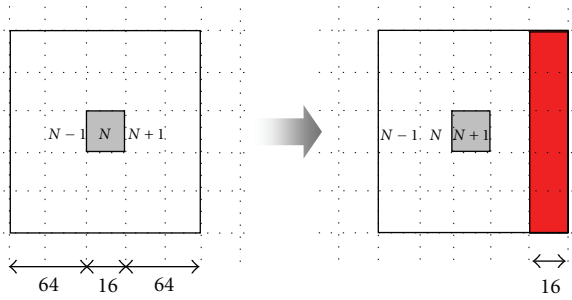

Figure 7: Search window update.

performs a fast search; the second step utilizes the coarse results coming from the first phase to refine the search for every MB partition. The second step can be executed in parallel for every MB partition. This leaves the designer the freedom to subsequently split the module to eventually avoid stalls in pipeline in the likely case the ME requires more cycles than intraprediction.

Among the issues the designer should take into account, there is still the memory bandwidth needed to feed the modules. From Table 1, we can notice that the ME module requests the largest amount of data. Besides coding parameters, the ME should receive data belonging to two frames: current and reference frame. For each MB, the data passed to the ME consists of the original MB luma values and the portion of reference frame enclosed by the search window (SW). Supposing one byte per luma sample and a SW set to 64×64 pixels (a suitable value for HD formats), we will get a width of $(64 + 16 + 64)$ pixels leading to 20736 bytes. Thus, we had 20736 bytes plus the original-image MB 16×16 bytes to send to the ME module for every MB. This leads to a very large memory bandwidth. Anyway, as could be noticed in Figure 7, not all of the SW must be resent every time a new MB is coded. Since MBs are coded in raster-scan order and search window of neighbouring MB overlaps, just a 16-byte-wide column update can be sent after the first complete window, as described in [28, 29]. Figure 7 shows the SW for the MB_N (left side) and the SW for the next MB (right side). The amount of data sent to the ME module for coding MB_{N+1} is shown as a red rectangle. When reaching the end of the row, MB_{N+1} does not need the update because this will be over the image border. Nevertheless, a SW update is written to the array, and it will be part of the SW of the first MB in the next row.

4.2. Modules Optimization. The H.264 encoder modules work on a block basis. Even though the basic block of the coding process is the macroblock, consisting of 16×16 pixel elements, the basic block of each module's computation can vary from 4×4 to 16×16. A number of experiments carried out at STMicroelectronics's Advanced System Technology Laboratories showed that, addressing HD resolutions, it is possible to disable interprediction modes involving the 8×8 blocks subpartitions without significant effects on video-quality and -coding efficiency. The same experiments also showed that fidelity range extensions are needed to improve video quality at high resolutions, as one could expect. For this reason, we choose both to disable ME on partitions 4×8, 8×4, and 4×4 and to add intra 8×8 and transform 8×8. In this scenario, most of the encoder modules work on 8×8 blocks of 8-bit samples. The 4×4 blocks are still used in Intra-4×4 and DCT/Q/IQ/IDCT 4×4. The Intra-16×16 prediction works on the whole MB, whereas the correspondent transformations just iterate the 4×4 procedures.

Usually, inside each module the computations require 16-bit precision for intermediate results. Thus, a typical situation is as follows:

(i) load 8-bit samples from memory;

(ii) switch to 16-bit precision and compute the results;

(iii) store the results to memory as 8-bit samples.

Some of the modules, or at least some parts of them, require a 32-bit precision. Among them, it is worth noting a few computations for pixels interpolation and the Quantization and Inverse-Quantization process.

In order to evaluate the different performance achievable with the three different ISAs, we have inserted the SIMD instructions in an already optimized ANSI C code which is used as reference to evaluate the achieved speedup. For a better understanding of the presented work, the comparison is not only carried out at global level, but for every H.264 functional unit.

In the following, the implementation detail of the sum of absolute difference (SAD) and the Hadamard filter will be shown for all the three addressed ISAs. Among all the several modules implementations, we have chosen to describe these particular operations for different reasons: the SAD is one of the most time-consuming operations in video-compression algorithms; the implementation of Hadamard filter is a good example for describing how an ANSI C implementation can be rewritten to best fit the available SIMD ISA. The access to data stored in memory will be discussed as well because it is a typical issue in optimizing video compression algorithms using SIMD instructions. A complete description of the encoder SIMD implementation on the ST240 processor can be found in [30].

4.2.1. SAD Operation. The sum of absolute differences is a key operation for a large variety of video-coding algorithms. The number of times this operation is executed during a coding process can vary depending on the encoder implementation and it strongly depends on the motion estimation module, that it is not covered by the H.264 standard definition. Anyway, independently of specific implementations, this operation is a key factor for the whole-encoder performance.

Here, we will show three different SAD implementations using SIMD instructions, and we will compare them with an optimized ANSI C code.

Given the essential role the SAD plays in video coding algorithms, some instruction sets include specific instructions to speed up such operation. Here, we will compare SIMD instruction sets having different size and different degree of specializations.

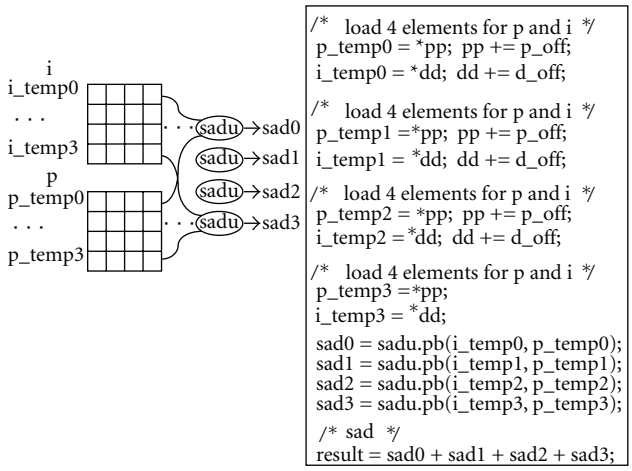

Figure 8: SAD implementation.

Table 2: SAD performance.

	Cycles	Operations	Load	Store
ANSI C version	36	134	8	0
SIMD version	14	30	8	0

Using the ST240 32-bit wide SIMD extension, the optimization of the SAD computation has been quite straightforward thanks to the SIMD instruction *sadu.pb*.

The SAD finds the "distance" between two 4×4 blocks, generally between a prediction block and the original image; given the two blocks in the left side of Figure 8, the pseudocode computing the SAD can be viewed in the right side of the same figure. Besides loading the input data, it basically consists of four calls to the *sadu.pb* instruction.

The achieved speedup is shown in Table 2.

The xSTream and P2012 architectures support 128-bit-wide vector registers, and they can perform 8-bit, 16-bit, or 32-bit arithmetic SIMD operations. Usually, SAD is performed using 8-bit precision, allowing for each SIMD calculation a capability to handle sixteen elements. Using vertical SIMD instructions, it is easy to achieve the absolute difference among several elements stored in two vectors, but the addition of the elements stored in a single vector is onerous because usually it requires several vertical SIMD inefficiently utilized. Both P2012 and xSTream ISAs have horizontal addition of SIMD instructions, but with different capability. In xSTream, it is allowed adding all the elements stored in the same vector, producing a scalar result. In P2012, VECx horizontal addition is limited to only add two adjacent elements inside a vector; in this way, four SIMD instructions must be used in order to achieve the scalar result of the SAD operation. This difference significantly impacts the encoder optimization. For example, when the SAD is calculated to evaluate the predictor cost in Intra 16×16, only two SIMDs are used with xSTream against the six used with P2012. This is schematized in Figure 9.

Even if in P2012 ISA the lacking of a horizontal SIMD for addition partially wastes the obtained great gain, we still complete the SAD operation using six VECx instructions and one scalar instruction, as shown in Figure 9, versus the 48 scalar instructions used in the ANSI C implementation (16 subtractions, 16 absolute values, and 16 additions).

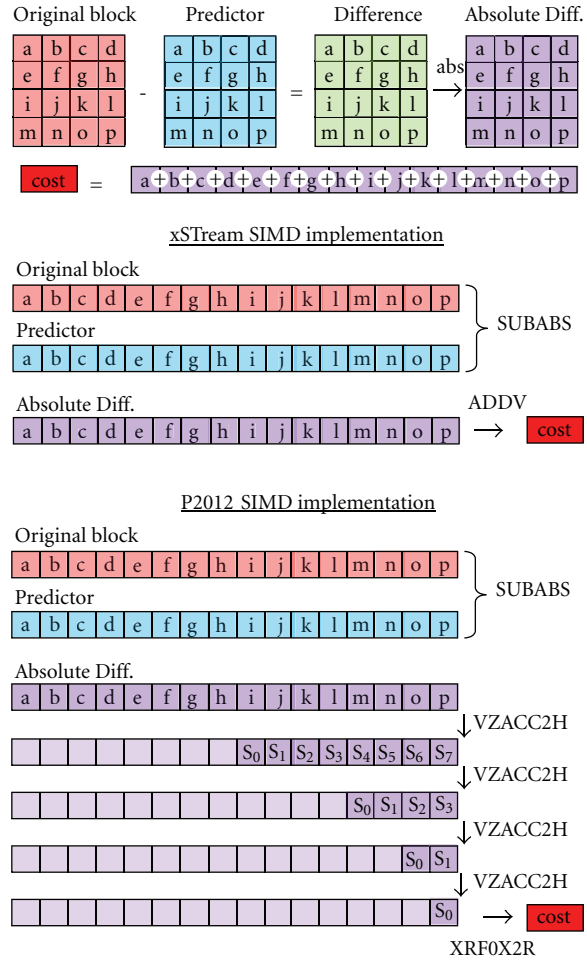

Figure 9: Predictor cost calculation.

4.2.2. Hadamard. We consider very interesting the Hadamard SIMD optimization because it involves a large number of instructions and can be considered a typical case study.

Although the Hadamard transform it is not currently used in the rest of the encoder, the intraprediction module utilizes such transform to find the best 16×16 intraprediction mode. The intramodule divides the predicted MB into sixteen 4×4 blocks. Each block is compared to the correspondent original-image's block, and sixteen differences are calculated. These sixteen values are filtered through the Hadamard transform before computing the SAD of the whole MB.

In the ST240 code, the optimization has started considering that Hadamard can be subdivided into two different phases: horizontal and vertical. The horizontal phase can be subdivided into 4 rows as well as the vertical phase into 4 columns, as shown in the portion of pseudocode in Table 3.

Table 3: Hadamard phases.

Horizontal phase	Vertical phase
/* first row */	/* first column */
$m0 = d0 + d3 + d1 + d2;$	$w0 = m0 + m12 + m4 + m8;$
$m1 = d0 + d3 - d1 - d2;$	$w1 = m0 + m12 - m4 - m8;$
$m2 = d0 - d3 + d1 - d2;$	$w2 = m0 - m12 + m4 - m8;$
$m3 = d0 - d3 - d1 + d2;$	$w3 = m0 - m12 - m4 + m8;$
/* second row */	/* second column */
$m4 = d4 + d7 + d5 + d6;$	$w4 = m2 + m14 + m6 + m10;$
$m5 = d4 + d7 - d5 - d6;$	$w5 = m2 + m14 - m6 - m10;$
$m6 = d4 - d7 + d5 - d6;$	$w6 = m2 - m14 + m6 - m10;$
$m7 = d4 - d7 - d5 + d6;$	$w7 = m2 - m14 - m6 + m10;$
/* third row */	/* third column */
$m8 = d8 + d11 + d9 + d10;$	$w8 = m1 + m13 + m5 + m9;$
$m9 = d8 + d11 - d9 - d10;$	$w9 = m1 + m13 - m5 - m9;$
$m10 = d8 - d11 + d9 - d10;$	$w10 = m1 - m13 + m5 - m9;$
$m11 = d8 - d11 - d9 + d10;$	$w11 = m1 - m13 - m5 + m9;$
/* fourth row */	/* fourth column */
$m12 = d12 + d15 + d13 + d14;$	$w12 = m3 + m15 + m7 + m11;$
$m13 = d12 + d15 - d13 - d14;$	$w13 = m3 + m15 - m7 - m11;$
$m14 = d12 - d15 + d13 - d14;$	$w14 = m3 - m15 + m7 - m11;$
$m15 = d12 - d15 - d13 + d14;$	$w15 = m3 - m15 - m7 + m11;$

In the pseudocode, d_i are the differences and m_i the intermediate values of the transform.

Once we have all the differences contained in packed 16-bit values subdivided into even and odd pairs, we can rewrite the first row of the horizontal Hadamard transform as

$$\begin{aligned} m0 &= (d0 + d1) + (d2 + d3), \\ m1 &= (d0 - d1) - (d2 - d3), \\ m2 &= (d0 + d1) - (d2 + d3), \\ m3 &= (d0 - d1) + (d2 - d3). \end{aligned} \tag{1}$$

In such a way, we can exploit the packed 16-bit addition and subtraction to obtain the high and low halves of the m_i coefficients. As can be noted, the low and high halves of $m0$ and $m2$ are the same, but while the $m0$'s value is achievable by adding its halves, to compute the value of $m2$ we have to subtract its high half from the lower one. Similar considerations can be applied to the odd elements $m1$ and $m3$.

Anyway, since the vertical phase of Hadamard is yet to come, there is no need to compute such values at this point. In fact, we can rewrite the m_i coefficient as functions of their own halves as follows:

$$\begin{aligned} m0 &= m0L + m0H, \\ m1 &= m1L - m1H, \\ m2 &= m2L - m2H, \\ m3 &= m3L + m3H. \end{aligned} \tag{2}$$

and utilize this notation to rewrite the vertical phase of the Hadamard transform as described below

$$\begin{aligned} w0 &= (m0 + m4) + (m8 + m12) \\ &= (m0L + m4L) + (m0H + m4H) \\ &\quad + (m8L + m12L) + (m8H + m12H) \\ &= (m0L + m4L) + (m8L + m12L) \\ &\quad + (m0H + m4H) + (m8H + m12H) \\ &= w0L + w0H. \end{aligned} \tag{3}$$

We can use the low and high halves of the intermediate coefficients to compute the low and high halves of the final coefficients w_i as illustrated in Figure 10.

The Hadamard optimization with ST240 SIMD is quite complex. Due to shortness of SIMD, the standard algorithm has been modified in order to better match the SIMD ISA features.

Using the xSTream and P2012 architectures, we followed a different approach. Our goal was the exploitation of 128-bit-wide SIMD minimizing the data reordering. Considering that the Hadamard transform can be defined as

$$H_n = \begin{bmatrix} H_{n-1} & H_{n-1} \\ H_{n-1} & -H_{n-1} \end{bmatrix} \qquad H_0 = 1, \tag{4}$$

the Hadamard matrices are composed of ± 1 and are a special case of discrete fourier transform (DFT). For this reason, the calculation can exploit the FFT algorithm, usually known as Fast Walsh-Hadamard transform [31].

The only issue is obtaining a good implementation of the FFT butterfly with SIMD, avoiding wasting all the gain achieved using fast algorithm with the data reordering needed to implement the calculation. Our approach consists of a modified butterfly that allows using always the same butterfly structure for every level, even if we have to reorder data between stages (Figure 11).

The output values coming from every butterfly can be calculated for 16 samples at a time using two SIMD instructions, one calculating the additions and one calculating the differences. In this way, we have the advantage of computing the output of each level using a simple SIMD implementation, at the cost of swapping intermediate results between different levels.

Even if xSTream and P2012 share this implementation mechanism, we have measured different performance. In this case, the difference depends on the different types of data manipulation instructions. The xSTream ISA having the third operand allowing the permutation of results inside vectors is more flexible and can implement the above algorithm with a reduced number of instructions respect to VECx P2012 ISA. Algorithm 1 shows the xSTream SIMD implementation.

4.2.3. Memory Access Issues. As previous exposed, a key factor to achieve a good performance improvement with

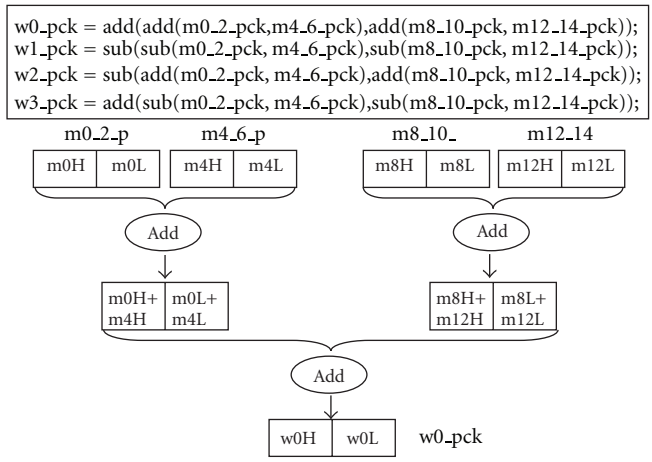

Figure 10: Hadamard vertical phase with ST240 SIMD.

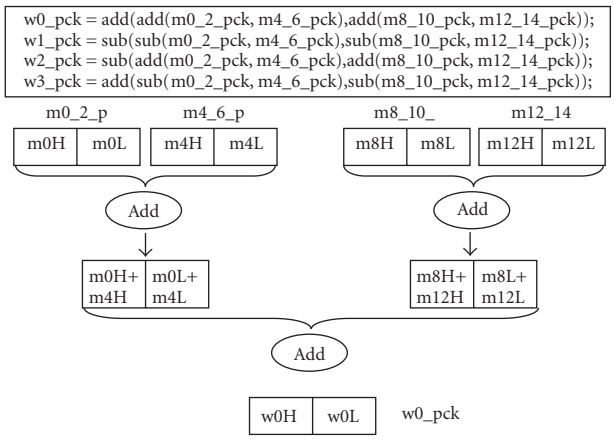

Figure 11: Hadamard modified butterfly.

SIMD optimization is the efficient handling of unaligned load operations. In general, programmers should structure the application data in order to avoid or minimize misaligned memory accesses. In video compression algorithm, the motion compensation is surely a case where it is not possible avoid unaligned memory accesses because it is impossible to predict motion vectors and consequently align data.

None of the three addressed architectures support unaligned load instructions. Therefore, it is important to efficiently use aligned accesses to load misaligned data from memory. The three ISAs support instructions to concatenate two vectors. This allows a solution consisting in two steps: first, we use two aligned load instructions for loading data in two vector registers, and, then, we concatenate and shift their elements in order to extract a single vector containing the needed data, as shown in Figure 12.

```
/*first level: one 16 samples butterfly*/
/*(s0 ÷ s7)+(s8 ÷ s15)*/
vaddh out_low = in_low, in_high
/*(s0 ÷ s7)−(s8 ÷ s15)*/
vsubh out_high = in_low, in_high

/*data reordering*/
/*0 1 2 3 8 9 10 11*/
vmrgbl in_low = out_low, out_high, perm
/*4 5 6 7 12 13 14 15*/
vmrgbu in_high = out_low, out_high, perm

/*second level: two 8 samples butterfly*/
vaddh out_low = in_low, in_high
vsubh out_high = in_low, in_high

/*data reordering*/
/*0 1 8 9 4 5 12 13*/
vmrge in_low = out_low, out_high
/*2 3 10 11 6 7 14 15*/
vmrgo in_high = out_low, out_high

/*third level: four 4 samples butterfly*/
vaddh out_low = in_low, in_high
vsubh out_high = in_low, in_high

/*data reordering*/
/*0 8 2 10 4 12 6 14*/
vmrgeh in_low = out_low, out_high
/*1 9 3 11 5 13 7 15*/
vmrgoh in_high = out_low, out_high

/*fourth level: eight 2 samples butterfly*/
vaddh out_low = in_low, in_high
vsubh out_high = in_low, in_high
```

Algorithm 1: Hadamard transform xSTream SIMD implementation.

```
uint32 AddressAt128;
vector_16b_sw Va, Vb, Vout;

AddressAt128b = ((uint32) (mref_ptr)) & (~0xF);
Offset = ((uint32) (mref_ptr)) & (0xF);
Va = ldq(AddressAt128, 0);
Vb = ldq(AddressAt128, 16);
Vout = wrot(Va, Vb, Offset);
```

Algorithm 2: Unaligned load SIMD implementation with concatenate instruction.

```
ui32_t PackCurr0 = *(orig_line);
ui32_t PackCurr1 = *(orig_line+1);
/* Pack to 128 bits */
TmpVectArray[0] = PackCurr0;
TmpVectArray[1] = PackCurr1;
Pack128In = ldqi(Pack128In, TmpVectArray,0);
/* Reorganize pixels */
Va = vmrgbeh(Va,Pack128In,VZero,permute0);
Vb = vmrgboh(Vb,Pack128In,VZero, permute1);
VPackCurr = vaddh(VPackCurr,Va,Vb,0);
```

Algorithm 3: Unaligned load SIMD implementation without concatenate instruction.

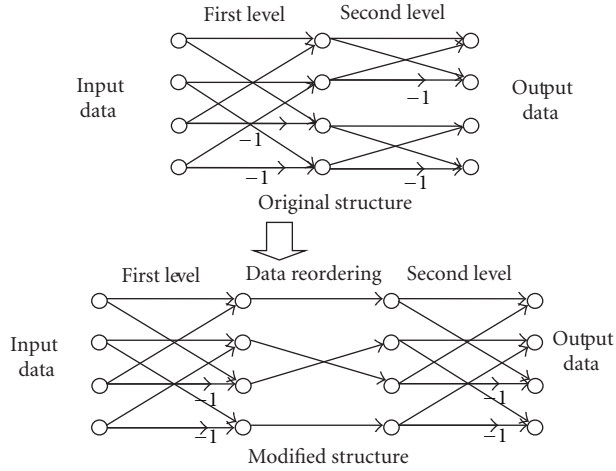

Figure 12: Unaligned load.

If an ISA does not define a SIMD performing this type of concatenate operation, then the unaligned load will be implemented with an extra cost due to the use of additional instructions for merging data between the two vectors.

Algorithm 2 shows the implementation of an unaligned load using xSTream. This solution can be compared to the same operation carried out without a concatenate instruction shown in Algorithm 3, in which we should add three instructions to reorganize the data for composing the required not-aligned vector.

It is very important that these concatenate instructions can take the offset argument not as a constant value but as a variable value; otherwise, modules such as motion compensation would not get any benefit from using them. For example, the Intel SSSE3 "palignr" instruction concatenates two operands and shift right the composite vector by an offset for extracting an aligned results, but the offset must be a compile-time constant value. This is a big issue for a module as motion compensation, in which it is impossible to know in advance the offset of a misaligned address.

5. Results

In the H.264 encoder, the most cycle-demanding modules have been optimized using SIMD instructions: motion estimation and compensation, DCT, Intraprediction, and so forth. The best way to compare different instruction sets in order to judge the effectiveness of both SIMD extensions and code optimizations is to measure the speedup obtained with the SIMD-based implementation versus the ANSI C version of the same source code. In order to separate the effect of SIMD performance improvement from ANSI C optimizations, we have inserted SIMD instructions in previously optimized ANSI C modules.

The results are provided in terms of average cycles spent to process one macroblock. The xSTream and P2012 architectures share the same modules subdivision. For the single-core DSP ST240, the subdivision is less fine, and related modules are joined together. In the reported tests, the presence of the ST240 processor is important because it allows comparing the single-processor elements of the multicore platforms to a single-core architecture. Tests are performed on a set of video sequences addressing different

Table 4: SIMD instructions for video coding.

Instruction description	Affected modules	Notes
Horizontal add: adds all the elements inside a vector register and produces a scalar result	ME, intraprediction	Speeds up SAD
Horizontal permute: rearranges elements inside a vector register	Intraprediction, DCT/Q/IQ/IDCT	Allows zig-zag scan and speeds up intra diagonal modes
Concatenate: concatenates two vector registers into an intermediate composite, shifts the composite to the right by a variable offset	Motion estimation and compensation	Allows software implementation of unaligned load
Promotion/demotion precision: an efficient support for promoting element precision while loading data from memory, and demoting the precision (with saturation) while storing data to memory	All the main modules	It will speed up the load and store operations for several modules
Absolute subtraction: for every element "a" in the first vector and every element "b" in the second vector performs the following operation: $\lvert a - b \rvert$	ME, intraprediction, deblocking filter	Speeds up SAD in conjunction with horizontal add; used in deblocking filter
Shift with round: performs the following operation for every element "a" in the vector operand: $(a + 2^{n-1}) >> n$, where n is a scalar value	IDCT, deblocking filter, motion compensation	Speeds up 1/2 pixel interpolation
Average: for every element "a" in the first vector and every element "b" in the second vector performs the following operation: $(a + b + 1) >> 2$	Intraprediction, deblocking filter, motion compensation	Speeds up 1/4 pixel interpolation

Table 5: Cycles/MB spent in each module for each ISA.

	xSTream			P2012			ST240		
	ANSI C	SIMD	Gain factor	ANSI C	SIMD	Gain factor	ANSI C	SIMD	Gain factor
Luma motion compensation	4788	2257	2.1x	8286	3965	2.1x			
Croma motion compensation	3064	658	4.7x	3626	1282	2.8x	265559	200380	1.3x
Motion estimation	303769	84342	3.6x	603182	114776	5.3x			
Intra 4×4	24366	10076	2.4x	38234	15760	2.4x	32013	19182	1.6x
Intra 8×8	15396	4997	3.0x	26972	9455	2,9x			
DCT/Q/IQ/IDCT 4×4	14994	7616	2.0x	20473	9088	2.3x	32013	19182	1.7x
DCT/Q/IQ/IDCT 8×8	18660	3498	5.3x	24486	11636	2.1x			

resolutions, and average results are resumed in Table 5. The results in Table 5 and Figure 13 show that the ST240, exploiting the instruction level parallelism (ILP) with a 4-issue VLIW architecture, achieves the best performance for the ANSI C implementation. All the SIMD implementations improve performance for every encoder module, but the ST240 with the shortest SIMD size obtains the lowest speedup factor. P2012 and xSTream with their wider SIMD can better exploit the data-level parallelism. In terms of pure number of cycles spent to encode one macroblock, the xSTream ISA achieves the best performance.

It is worth analyzing in detail these results to understand how different instruction sets lead to different performance. The xSTream processor elements take advantage from the "horizontal add" instruction that allow an efficient computation of the SAD operations: it is evident in the ME module, where xSTream spends about 25% fewer cycles than P2012 (84,342 versus 114,776 cycles/MB). The higher speedup obtained by P2012 is mainly due to the less-efficient ANSI C code generated by the P2012 compiler. We already described as the ST240 can exploit a specific instruction for the SAD operation. In fact, its result is not far from the architectures having 128-bit-wide vector registers (the 200, 380 cycles/MB also include motion compensation). From these results, we can state that the support for horizontal SIMD will not only give a great performance improvement for the SAD operation, but it significantly impacts the whole ME module.

As earlier said, data manipulation instructions are a key factor to fully exploit SIMD implementations because operations such as transposing matrices or data reordering become frequent in this type of optimizations. An experimental result confirming this consideration can be seen in the DCT/Q/IQ/IDCT 8×8 module, covering all the toolchain performing the residual coding and decoding. This module involves several data-reordering operations, ranging from matrix transposition to zig-zag reorder. Both ST240 and xSTream instruction sets support the permutation of elements inside a vector in a very efficient way, as described in Sections 3.1 and 3.2. The P2012 SIMD extension includes a series of instructions for interleaving and merging elements between two vector operands. The great speed up the

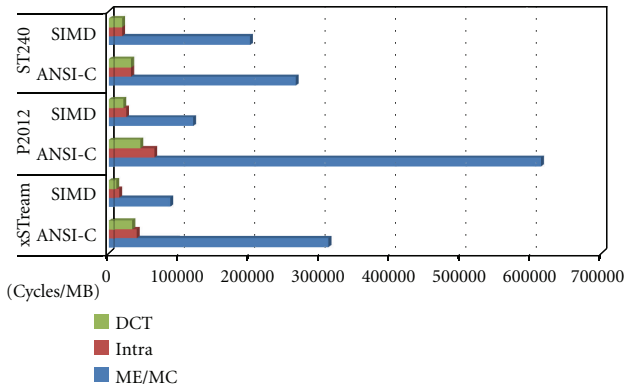

Figure 13: ISA comparison.

xSTream architecture gathers in comparison with P2012 is mainly due to the possibility to permute elements using a single instruction, in a sort of horizontal permute. The effect is emphasized in the 8×8 transform where the data reordering process is stressed more than in the 4×4 case. In our experience, we saw that if such instruction is available, then the zig-zag reordering can be effectively implemented with SIMD instructions; otherwise, we are forced to use the scalar implementation involving look-up tables to perform the reordering.

Intraprediction can exploit the horizontal permute instruction as well; the intraprediction modes involving diagonal directions require the permutation of elements inside the resulting vectors. For similar reasons, ST240 achieves great speedup factors in DCT and intramodules (resp., 1.7 and 1.6), considering that a 32-bit-wide SIMD can only perform two 16-bit-arithmetic operations.

There are other several SIMD instructions that in our opinion are to be considered as key instructions for optimizing video codec applications. Here, we assume an instruction set will already include SIMD for all the common arithmetic operations, compare, select, shift, and memory operations.

In previous sections, we already discussed about the impact of the unaligned memory access to the video codec performance. All the encoder modules are affected by the performance of the unaligned memory operations, but it becomes a keyfactor for motion estimation and compensation. An instruction concatenating two vectors and producing a vector at the desired offset is fundamental to implement an unaligned load instruction. As stated in Section 4.2.3, the capability to support variable offsets is a key factor for the instruction usability because the offset could not be known in advance.

Inside most of the modules, the computations require a 16-bit precision for intermediate results, but the input and output data contained into the noncompressed YUV images are 8-bit values. Thus, a typical operation at the beginning of a module is to load 8-bit input values and extend them to 16-bit precision. At the end, the output data precision is usually demoted down to 8-bit saturating the values before storing the results. Therefore, even if the support to 8-bit operations is not required, it would be very useful that an instruction set will include SIMD instructions for promoting and demoting precision in a fast way. An optimal solution will also combine promotion with load operations and demotion with store instructions.

Usually, the video codec algorithms try to avoid the division operations because of its computational cost. When needed, divisors are power of two, and the division is substituted with a shift right with rounding as follows:

$$\frac{a}{2^n} \Longleftrightarrow (a + 2^{n-1}) \gg n. \tag{5}$$

Therefore, even if most instruction sets already include this type of instruction, it is important to remind its utility. Often, the shift right with rounding is used for averaging two or more values, as in the intraprediction and deblocking filter. In our implementation, one of the reasons the ST240 achieves a good speedup in the intraprediction module is the presence of an average SIMD instruction in the instruction set.

Table 4 summarizes our conclusions based on the presented work. The proposed instructions are described in the first column. For each instruction, the table indicates the H.264 modules that will be mainly affected by the introduction of the instruction, as well as a few notes about specific contributions to basic video coding operations.

6. Conclusions

This paper presents efficient implementations of the H.264/AVC encoder on three different ISAs. The optimization process exploits the SIMD extensions of the three architectures for improving the performance of the most time-consuming encoder modules. For each addressed architecture, experimental results are presented in order to both compare the different implementations and evaluate the speedup versus the optimized ANSI C code.

The paper discusses how SIMD size and different instruction sets can impact the achievable performance. Several issues affecting video-coding SIMD optimization are discussed, and authors' solutions are presented for all the architectures.

Most instruction sets have specific SIMD instructions for video coding. Even though these instructions can lead to great performance improvements, they could be useless for other application families. In this paper, we identify a set of generic SIMD instructions that can significantly improve the performance of video applications.

Besides presenting the SIMD optimization for the most time-demanding modules, the paper describes how a complex application as the H.264/AVC encoder can be partitioned to a multicore architecture.

Acknowledgments

The authors would like to thank STMicroelectronics's Advanced System Technology Laboratories for their support. This work is supported by the European Commission in the context of the FP7 HEAP project (#247615).

References

[1] "VC-1 Compressed Video Bitstream Format and Decoding Process," SMPTE 421M-2006, SMPTE Standard, 2006.

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

[3] G. J. Sullivan, P. Topiwala, and A. Luthra, "The H.264/AVC Advanced Video Coding Standard: Overview and Introduction to the Fidelity Range Extensions," in *Applications of Digital Image Processing XXVII*, Proceedings of SPIE, August, 2004.

[4] D. Marpe, T. Wiegand, and S. Gordon, "H.264/MPEG4-AVC fidelity range extensions: tools, profiles, performance, and application areas," in *IEEE International Conference on Image Processing (ICIP '05)*, pp. 593–596, September 2005.

[5] H. Schwarz, D. Marpe, and T. Wiegand, "Overview of the scalable video coding extension of the H.264/AVC standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 17, no. 9, pp. 1103–1120, 2007.

[6] Joint Collaborative Team on Video Coding (JCT-VC), "WD4: Working Draft 4 of High-Efficiency Video Coding," 6th Meeting, Torino, Italy, July, 2011.

[7] J. Probell, "Architecture considerations for multi-format programmable video processors," *Journal of Signal Processing Systems*, vol. 50, no. 1, pp. 33–39, 2008.

[8] M. Koziri, D. Zacharis, I. Katsavounidis, and N. Bellas, "Implementation of the AVS video decoder on a heterogeneous dual-core SIMD processor," *IEEE Transactions on Consumer Electronics*, vol. 57, no. 2, pp. 673–681, 2011.

[9] M. Sayed, W. Badawy, and G. Jullien, "Towards an H.264/AVC HW/SW integrated solution: an efficient VBSME architecture," *IEEE Transactions on Circuits and Systems II*, vol. 55, no. 9, pp. 912–916, 2008.

[10] T. Rintaluoma and O. Silvén, "SIMD performance in software based mobile video coding," in *10th International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS '10)*, pp. 79–85, July 2010.

[11] H. Lv, L. Ma, and H. Liu, "Analysis and optimization of the UMHexagons algorithm in H.264 based on SIMD," in *2nd International Conference on Communication Systems, Networks and Applications (ICCSNA '10)*, pp. 239–244, July 2010.

[12] X. Zhou, E. Q. Li, and Y.-K. Chen, "Implementation of H.264 decoder on general-purpose processors with media instructions," in *Image and Video Communications and Processing*, Santa Clara, Calif, USA, January 2003.

[13] J. Lee, S. Moon, and W. Sung, "H.264 decoder optimization exploiting SIMD instructions," in *IEEE Asia-Pacific Conference on Circuits and Systems (APCCAS '04)*, pp. 1149–1152, December 2004.

[14] W. Lo, D. Lun, W. Siu, W. Wang, and J. Song, "Improved SIMD architecture for high performance video processors," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 21, no. 12, pp. 1769–1783, 2011.

[15] D. Talla, L. K. John, and D. Burger, "Bottlenecks in multimedia processing with SIMD style extensions and architectural enhancements," *IEEE Transactions on Computers*, vol. 52, no. 8, pp. 1015–1031, 2003.

[16] Z. Shen, H. He, Y. Zhang, and Y. Sun, "A Video Specific Instruction Set Architecture for ASIP design," *VLSI Design*, vol. 2007, Article ID 58431, 7 pages, 2007.

[17] M. Shafique, L. Bauer, and J. Henkel, "Optimizing the H.264/AVC video encoder application structure for reconfigurable and application-specific platforms," *Journal of Signal Processing Systems*, vol. 60, no. 2, pp. 183–210, 2010.

[18] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood, "Lx: a technology platform for customizable VLIW embedded processing," in *27th Annual International Symposium on Computer Architecture (ISCA '00)*, pp. 203–213, June 2000.

[19] J. Fisher, P. Faraboschi, and C. Young, "VLIW processors: from blue sky to best buy," *IEEE Solid-State Circuits Magazine*, vol. 1, no. 2, pp. 10–17, 2009.

[20] N. Coste, H. Garavel, H. Hermanns, F. Lang, R. Mateescu, and W. Serwe, "Ten Years of Performance Evaluation for Concurrent Systems using CADP," in *4th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation ISoLA*, Heraklion, Greece, 2010.

[21] D. Pandini, G. Desoli, and A. Cremonesi, "Computing and design for software and silicon manufacturing," in *IFIP International Conference on Very Large Scale Integration (VLSI '07)*, pp. 122–127, October 2007.

[22] G. Desoli and E. Filippi, "An outlook on the evolution of mobile terminals: from monolithic to modular multi-radio, multi-application platforms," *IEEE Circuits and Systems Magazine*, vol. 6, no. 2, pp. 17–29, 2006.

[23] L. Benini, "P2012: a many-core platform for 10Gops/mm2 multimedia computing," in *21st IEEE International Symposium on Rapid System Prototyping*, Fairfax, Va, USA, June 2010.

[24] C. Silvano, W. Fornaciari, S. Crespi Reghizzi et al., "2PARMA: parallel paradigms and run-time management techniques for many-core architectures," in *IEEE Annual Symposium on VLSI*, pp. 494–499, July 2010.

[25] C. Mucci, L. Vanzolini, I. Mirimin et al., "Implementation of parallel LFSR-based applications on an adaptive DSP featuring a Pipelined Configurable Gate Array," in *Design, Automation and Test in Europe (DATE '08)*, pp. 1444–1449, March 2008.

[26] P. Paulin, "Programming challenges & solutions for multiprocessor SoCs: An industrial perspective," in *Design Automation Conference (DAC '11)*, June 2011.

[27] A. Kumar, D. Alfonso, L. Pezzoni, and G. Olmo, "A complexity scalable H.264/AVC encoder for mobile terminals," in *European Signal Processing Conference (EUSIPCO '08)*, Lausanne, Switzerland, August 2008.

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

[29] B. Zatt, M. Shafique, F. Sampaio, L. Agostini, S. Bampi, and J. Henkel, "Run-Time Adaptive Energy-Aware Motion and Disparity Estimation in Multiview Video Coding," in *48th Design Automation Conference (DAC '11)*, pp. 1026–1031, San Diego, Calif, USA, June 2011.

[30] M. Bariani, I. Barbieri, D. Brizzolara, and M. Raggio, "H.264 implementation on SIMD VLIW cores," STreaming Day 2007, Genova, Italy.

[31] C. S. Lubobya, M. E. Dlodlo, G. de Jager, and K. L. Ferguson, "SIMD implementation of integer DCT and hadamard transforms in H.264/AVC encoder," in *Proceedings of the IEEE AFRICON*, pp. 1–5, September 2011.