J Sign Process Syst (2011) 62:301–318
DOI 10.1007/s11265-010-0463-z

# Modeling and Evaluating Non-shared Memory CELL/BE Type Multi-core Architectures for Local Image and Video Processing

**Svetislav Momcilovic · Leonel Sousa**

Received: 4 December 2008 / Revised: 24 February 2010 / Accepted: 24 February 2010 / Published online: 18 March 2010


**Abstract** Local processing, which is a dominant type of processing in image and video applications, requires a huge computational power to be performed in real-time. However, processing locality, in space and/or in time, allows to exploit data parallelism and data reusing. Although it is possible to exploit these properties to achieve high performance image and video processing in multi-core processors, it is necessary to develop suitable models and parallel algorithms, in particular for non-shared memory architectures. This paper proposes an efficient and simple model for local image and video processing on non-shared memory multi-core architectures. This model adopts a single program multiple data approach, where data is distributed, processed and reused in an optimal way, regarding the data size, the number of cores and the local memory capacity. The model was experimentally evaluated by developing video local processing algorithms and programming the Cell Broadband Engine multi-core processor, namely for advanced video motion estimation and in-loop deblocking filtering. Furthermore, based on these experiences it is also addressed the main challenges of vectorization, and the reduction of branch mispredictions and computational load imbalances. The limits and advantages of the regular and adaptive algorithms are also discussed. Experimental results show the adequacy of the proposed model to perform local video processing, and that real-time is achieved even to process the most demanding parts of advanced video coding. Full-pixel motion estimation is performed over high resolution video ($720 \times 576$ pixels) at a rate of 30 frames per second, by considering large search areas and five reference frames.

**Keywords** 2D local processing · Multi-core · Parallel algorithms · H.264/AVC · Motion estimation · Deblocking filtering · Cell/BE

## 1 Introduction

Multimedia and computer games demand high performance computing platforms, namely Local Video and Image Processing (LVIP) components of those applications require a huge number of arithmetic and logic operations [12]. For example, a sustained performance of 7 G FLoating point Operations Per Second (FLOPS) is required to implement in real-time a local operator that typically applies 100 arithmetic operations per pixel over high resolution $1{,}920 \times 1{,}080$ video frames. If pixels of an image are represented as a matrix, and sequences of images as arrays, the local processing of a pixel requires only data located in the sub-matrix, or sub-array, around the pixel. Adjacent sub-matrices, or Local Processing Areas (LPAs), can be defined along the space axes for image processing, and extended to 3-D arrays when the time axis is introduced for video processing. LPAs may overlap each other, but pixels can be processed independently and in parallel when overlapped data is shared or replicated. Therefore, a lot of research has been performed in the last years to develop parallel algorithms and dedicated processors for real-time local processing [10, 16]. Although dedicated

S. Momcilovic ( ) · L. Sousa
INESC-ID IST/TULisbon, Lisboa, Portugal
e-mail: Svetislav.Momcilovic@inesc-id.pt

L. Sousa
e-mail: Leonel.Sousa@inesc-id.pt

hardware processors are more efficient when processing time, energy consumption, and cost are taken all together in consideration, the non-recurring engineering required is quite high.

To sustain the rise of the processing capacity of current processors according to Moore's Law the number of cores available in a single integrated circuit has to constantly increase. Moreover, vector units have been introduced in those cores in order to compute up to 512-bit in parallel by using the Single Instruction Multiple Data (SIMD) approach. However, the achieved performance strongly depends on the applications' parallelism and characteristics of the multi-core processors [14]. Important characteristics are, for example, homogeneity and the ability to share memory: e.g. homogeneous multi-cores with shared memory, such as the Intel and AMD dual-cores and quad-cores, and more recently the Larrabee [19] with up to 32 × 86 cores; heterogeneous chip-level multi-cores without hardware to support shared memory, such as the IBM Cell Broadband Engine (Cell/BE); a graphics processing unit is another typical example of a homogeneous multi-cores engine with shared memory, which can also be used for accelerating general purpose processing but its operation has to be orchestrated by a host processor [6].

In the Cell/BE multi-core processor, the Power Processor Element (PPE) main core controls the operation of eight Synergistic Processor Element (SPE) cores [7]. Each SPE has its own local memory and cannot directly access the main memory. Exploitation of the computational capacity of these heterogeneous multi-core processors without native shared memory is a research challenge. Programming models and parallel algorithms have been recently launched to the Cell/BE processor [1, 4, 5], namely algorithms and techniques to perform video parallel processing on the Cell/BE processor [17, 18]. However, there is no general model suitable to implement in parallel the important class of LVIP methods on this type of architectures. Among the difficulties to develop such model, memory restrictions have to be taken in consideration, as a fundamental aspect for high resolution images. The SPEs are SIMD cores, with 128-bit wide registers and access only to 128-bit aligned words in memory, which means that shuffle and shift instructions have to be used to manage unaligned and scalar data. This restriction makes the vectorization a fundamental task to achieve efficient LVIP on the Cell/BE processor.

This paper proposes a general parallelization model for LVIP on multi-core processors without hardware support for shared memory. This model adopts the Single Program Multiple Data (SPMD) approach, by distributing data to the local memories in an optimal way and processing it almost independently in multiple iterations. The proposed model allows to predict the execution time given the processing requirements and the system's characteristics. It is a flexible model adaptable to the specific characteristics of a given architecture, as it is shown herein for the case of the Cell/BE architecture. Moreover, a new algorithm is proposed for balancing the computational load on data dependent LVIP, as well as a set of practical solutions are discussed for reducing the overheads caused by branch misprediction and data alignment.

The parallelization model is experimentally evaluated by programming the most computationally demanding parts of H.264/AVC video coder on the Cell/BE, namely advanced video motion estimation and in-loop deblocking filtering. Experimental results not only validate the proposed parallelization model but also show that the developed parallel algorithms based on the model are efficient and scale well with the number of cores.

This paper is organized as follows. In Section 2 the main characteristics of the LVIP are presented. In Section 3, the parallelization model for video local parallel processing is proposed and discussed. Section 4 performs the evaluation of the model, and assesses the relative performance of the Cell/BE processor for advanced motion estimation and deblocking filtering. Finally, experimental results are presented in Section 5, and Section 6 concludes the paper.

## 2 Local Video and Image Processing

In local processing, the output signal $\mathbf{Z}$ is obtained by applying a function $\Psi$ over a compact neighborhood of the input signal, represented in Eq. 1 by $\mathbf{X}$, and also of the output signal if recursive computation is considered. In image processing, these signals are represented in space ($\mathbf{s}$) while in video processing they are also dependent on time (t). For the particular case that $\Psi$ is a linear operator, the output signal can be obtained by convolving (*) input data with the system's impulse response (array $\psi$ in Eq. 1). Typical local image processing is image filtering, namely the linear Laplacian of Gaussian for edge detection [20] and the non-linear median rank filter [21].

$$\mathbf{Z}(\mathbf{s},t) = \Psi(\mathbf{X}(\mathbf{s},t),\mathbf{Z}(\mathbf{s},t)) \quad \underline{Linear} \rightarrow$$
$$\mathbf{Z}(\mathbf{s},t) = \psi * (\mathbf{X}(\mathbf{s},t),\mathbf{Z}(\mathbf{s},t)) \tag{1}$$

Equation 1 can be computed in parallel and independently for different output samples over local data

neighborhoods by respecting potential data dependencies. Overlapping between neighborhoods is one of the data dependencies to deal with, which is not a straightforward task for multi-processors that do not support shared memory. Two different local video processing methods adopted in the H.264/AVC [22] encoders have been chosen as LVIP case studies in this paper, namely advanced video motion estimation and in-loop deblocking filtering. They apply different linear and non linear operators and have been chosen also because they are the most computationally intensive components of video encoders.

The block diagram of a H.264/AVC predictive encoder is depicted in Fig. 1. Each Macroblock (MB) can be encoded on *intra* or *inter* modes. In intra mode prediction is performed based on data from the current frame (CF), while in inter mode prediction is performed based on reconstructed samples from previously encoded reference frames (RFs), by applying motion estimation and compensation. A given MB is then differentially encoded by taking the predicted MB as the reference, the residual MB is transformed, and the resulting coefficients are quantized and entropy-encoded. Deblocking filtering is applied to the decoded MBs in the coding loop, in order to reduce the blocking distortion effect.

## 2.1 Advanced Video Motion Estimation

Motion estimation on video sequences is a LVIP method applied to exploit temporal redundancy between consecutive frames. In block matching motion estimation a CF is organized in MBs, and, in general, a search algorithm can be applied independently, in any order, to each MB to find the "best matching" within search areas (SAs) in the RFs. Accordingly, motion estimation can be expressed by Eq. 1, where **X** becomes a 3D sub-array composed by CF and the corresponding RFs, and $\Psi$ represents the Sum of Absolute Differences (SAD) distortion measure:

$$SAD(x, y, \mathbf{d}) = \sum_{i=0,\, j=0}^{m,n} \big|CF(x+i, y+j) - RF(x+i+d_x, y+j+d_y)\big|, \qquad (2)$$

where $x$ and $y$ represent the MB's coordinates in the CF, $\mathbf{d}$ is the motion vector (MV), and $m$ and $n$ are the height and width of the MB, respectively. Up to seven different MB sizes can be considered in the H.264/AVC standard, namely $16 \times 16$, $16 \times 8$, $8 \times 16$, $8 \times 8$, $8 \times 4$, $4 \times 8$ and $4 \times 4$ pixels. These smaller blocks, also called subblocks (SBs), define seven different SB modes.

Two different search algorithms are considered in this paper, the optimal Full-Search Block-Matching with SAD reusing (FSBMr) regular search approach [3] and the Unsymmetrical-cross MultiHexagon-grid Search (UMHS) [9] suboptimal adaptive algorithm. The Full-Search Block-Matching (FSBM) algorithm exhaustively examines all the candidates in a SA by considering the seven different SB modes. With the FSBMr, these different modes can be hierarchical computed by reusing the SAD results of the smaller SBs to compute the larger ones. The UMHS search algorithm dynamically adapts the search procedure to the data, in order to reduce the number of potential candidates that have to be checked. This type of sub-optimal search algorithms is faster but can drop in local minima.

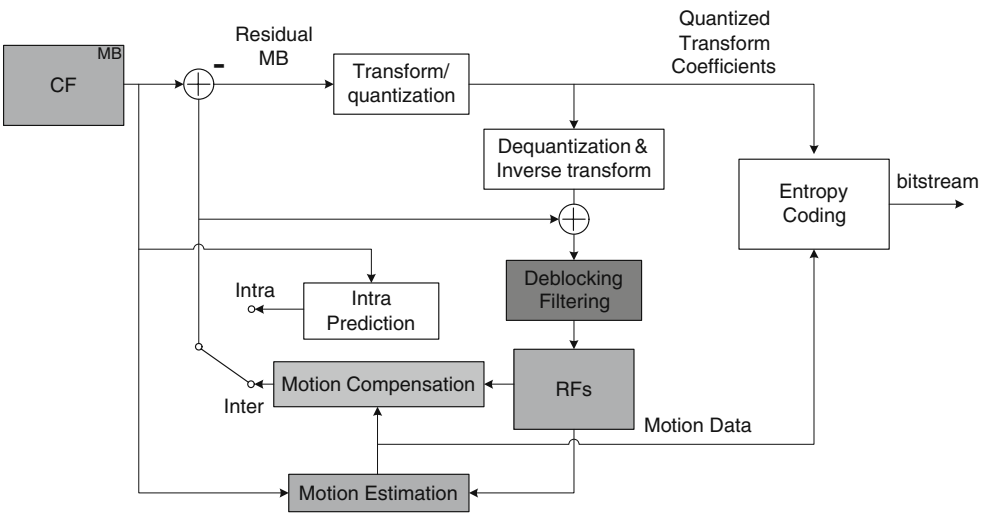


**Figure 1** Block diagram of H.264/AVC; blocks are shaded to emphasize the LVIP considered in this paper.

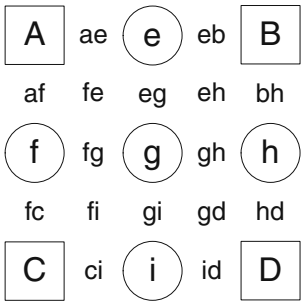


**Figure 2** Quarter-pixel interpolation.

*2.1.1 Sub-pixel Interpolation and Refinement*

H.264/AVC considers not only different SB modes and multiple RFs but also sub-pixel motion estimation, namely quarter-pixel precision. To perform sub-pixel motion estimation firstly RFs have to be interpolated. Initially, the original pixels are interpolated by using a 6-tap filter in order to generate the half-pixels, and then a bilinear filter is applied for achieving quarter-pixel precision. Figure 2 depicts the neighborhood used for quarter-pixel interpolation, where full-pixels are identified by squares and capital letters, half-pixels by circles and small letters, and double letters are used for the quarter-pixels.

The best matching candidate found for full-pixel motion estimation becomes the central point of the sub-pixel SA, and centered at this point the interpolated frames are used instead of the original RFs. The quarter-pixel refinement process performs the regular search in $5 \times 5$ SA (see Fig. 3) .

2.2 In-Loop Deblocking Filtering

Blocking distortion is a known effect on block-based video encoders, which introduces artificial edges, and consequently spurious high frequency coefficients in the transform domain. Within H.264/AVC, deblocking filtering is applied to smooth artificial block edges, in order to improve the appearance of decoded frames. Moreover, deblocking filtering is applied on both luminance and chrominance components of the pixels. Filtering is applied for both horizontal and vertical edges of $4 \times 4$ SBs, by considering three neighbor pixels on both sides of each edge. An example of horizontal and vertical considered edges is shown on Fig. 4, where p2, p1, p0, q0, q1 and q2 are pixels used in the filtering process, while edges are represented by bold lines. The filtering strength, BS, is defined according to a set of parameters, such as type of coding (inter/intra), MV, transformed coefficients and position of the edge within the MB.

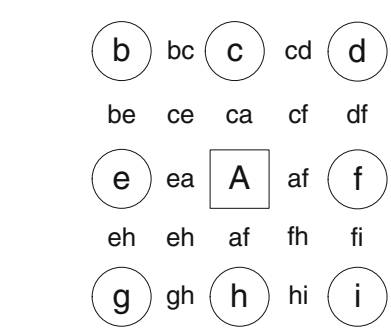


**Figure 3** Quarter-pixel refinement.

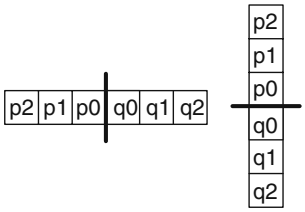


**Figure 4** Horizontal and vertical edges to be filtered.

The filtering only takes place if the gradient of the image across the block edge is greater than the predefined quantization-dependent thresholds $\alpha$ and $\beta$:

- for BS=0 or $|p0 - q1| \geq \alpha$ or $|p1 - p0| \geq \beta$ or $|q1 - q0| \geq \beta$ no filtering is applied;
- for BS$\in$\{1,2,3\}, a 4-tap filter is applied with inputs p1, p0, q0 and q1, producing filtered outputs p'0 and q'0; in addition, if $|p2 - p0| < \beta$, p'1 is produced by a 4-tap filter with inputs p2, p1, p0 and q0; a similar rule is applied for q'1;
- for BS=4: if $|p2 - p0| < \beta$ and $|p0 - q0| < (\alpha/4 + 2)$, p'0 is computed by using a 5-tap filter with p2, p1, p0, q0 and q1; p'1 is produced by a 4-tap filter with p2, p1, p0 and q0, while p'2 is produced by a 5-tap filter with p3, p2, p1, p0 and q0; otherwise, only p'0 is modified by a 3-tap filter with p1, p0 and q0; similar rules are used for filtering of the q-side of the edges.

## 3 Parallelization Model

The proposed general model for LVIP on non-shared memory multi-core processors is organized in two distinct levels. A first basic level common to every non-shared memory multi-core processor, which includes automatic data partition, and a second level that introduces particular processing features and architecture

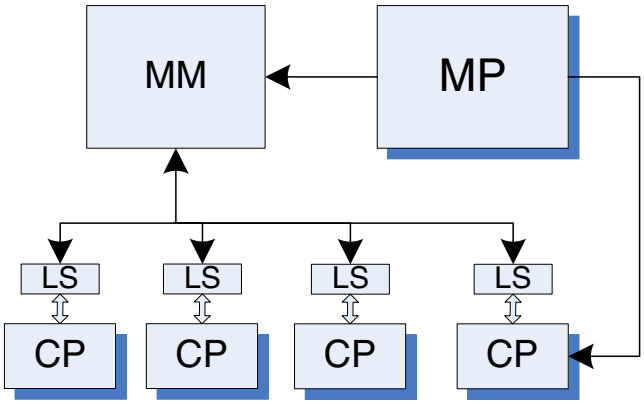


**Figure 5** Heterogenous multi-core processor: *CP* Co-Processor, *MP* main processor, *LS* local store, *MM* main memory.

characteristics to enhance data partition and also to predict the execution time.

Figure 5 illustrates the general architecture of a heterogenous multi-core processor without shared memory. Heterogeneity can exist at a first level between the *Main Processor* and the *Co-Processor* cores, and at a second level between these *Co-Processor* cores. The memory is locally distributed by the *Co-Processors*, each having a *Local Store* with limited size $M_{ls}$. Only the *Main Processor* has direct access to the *Main Memory* and data has to be explicitly exchanged between the *Main Memory* and *Local Stores* usually by Direct Memory Access (DMA).

For such architectures with multiple *Co-Processors*, LVIP can be performed in parallel over local data by applying the SPMD and SIMD approaches. The same instruction, or program, launched on parallel threads is applied to several spatially delimited LPAs, reducing to the minimum the need to share data distributed by the *Local Stores*. Figure 6 shows an LPA, where non-overlapping areas are presented with full lines

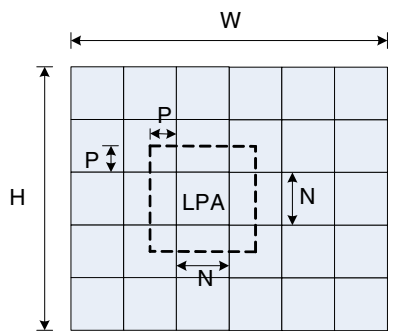


**Figure 6** Local processing area in an image.

and overlapping areas are bounded with dashed lines. To make the presentation of the model simpler, *Co-Processors* are considered to be homogeneous and LPA is assumed to be square, with size $N + 2P$, where $N$ and $P$ are the size of the non-overlapped and overlapped parts, respectively. Multiple LPAs compose the image or frame, with width and height $W$ and $H$, respectively.

### 3.1 Basic Model

The basic model targets: *i*) optimal load balancing between the operating cores available; *ii*) minimum synchronization and communication overheads. *Co-Processors* can process in parallel LPAs resident in *Local Stores*, with reduced requirements of data communication. However, optimal load balancing is not guaranteed for data dependent processing. In this perspective, it is preferable to have regular and data independent processing, even if it requires higher computational load than adaptive but data dependent processing. On the other side, the inexistence of shared memory increases the importance of finding the optimal data partition and of extensively applying data reuse. The basic level of the model provides an analytical solution to compute the optimal data partition and to roughly predict the processing time, and also to analyze the computation and communication components of the whole processing time.

#### *3.1.1 Input Data Partition*

Data parallelism and data reusing techniques can be exploited by splitting the space of a frame into approximately equal areas which are processed in parallel by the *Co-Processors*, while the *Main Processor* handles the control flow (see Fig. 5). In the case that the number of cores (c) is less than or equal to the number of LPAs in a frame row, $c \leq \frac{W}{N}$, the frame can be divided in $c$ vertical data chunks with equal size. Without loosing generality, we assume that $H$ and $W$ are multiples of $N$. Therefore, the width of each data chunk ($w$), taking an LPA as the unit, can be expressed as ($j$ is the chunk index):

$$w(j) = \begin{cases} \left\lceil \frac{W/N}{c} \right\rceil & , j < \frac{W}{N} \bmod c \\ \left\lfloor \frac{W/N}{c} \right\rfloor & , \text{otherwise} \end{cases}, \quad j = 0, 1, .., c-1. \tag{3}$$

It is important to note that although Eq. 3 can also be used for shared memory architectures the only target

of the proposed model are distributed memory multicore architectures. In shared memory architectures data is directly available to all cores, but to efficiently perform LVIP one has to take in consideration the specific characteristics of the memory hierarchy and organization. Each of these chunks will be processed in one or more iterations, depending on the *Local Store* size ($M_{ls}$). Since synchronization is required in each iteration, by minimizing the number of iterations we are also minimizing synchronization overhead. Moreover, data transfer overhead on distributed memory multicores, such as the Cell/BE, decreases when larger data portions are transferred at once [11]. Figure 7 shows an example of a frame division where a data chunk doesn't fit in the *Local Store*. Therefore, chunks are additionally divided in the vertical direction to be sequentially processed in different iterations. In this case, the height of a single partition ($h$), taking an LPA as the unit, can be calculated as:

$$h(i) = \begin{cases} \left\lceil \frac{H/N}{I} \right\rceil & , i < \frac{H}{N} \bmod c \\ \left\lfloor \frac{H/N}{I} \right\rfloor & , \text{otherwise} \end{cases}, \quad i = 0, 1, .., I-1. \tag{4}$$

where $i$ is the iteration index, and $I$ is the total number of iterations. For computing the total number of iterations $I$, the following condition has to be respected:

$$(w_{max}N + 2P)(h_{max}N + 2P) \leq M_{ls}, \tag{5}$$

which is equivalently to:

$$h_{max} \leq \frac{1}{N}\left(\frac{M_{ls}}{w_{max}N + 2P} - 2P\right). \tag{6}$$

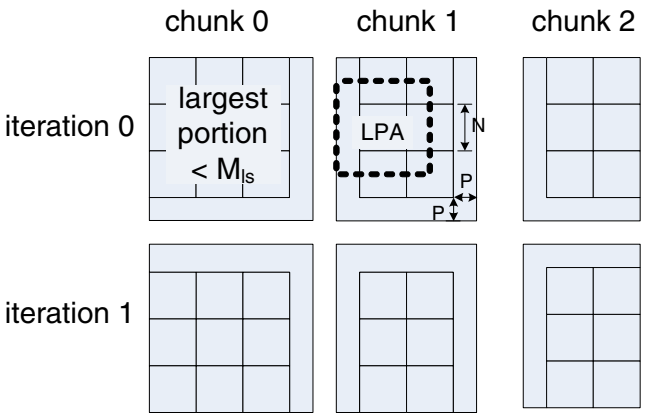


**Figure 7** Frame division in the proposed LVIP model.

By applying Eqs. 3 and 4 in inequation 6:

$$\left\lceil \frac{H/N}{I} \right\rceil \leq \frac{1}{N}\left(\frac{M_{ls}}{\left\lceil \frac{W/N}{c} \right\rceil N + 2P} - 2P\right). \tag{7}$$

The explicit solution for the minimum number of required iterations ($I_{min}$) can be easily found if it is assumed that the result of the right side of the inequation 7 is a natural number. Therefore, the floor function is applied to the right side of inequation 7, instead of using the exact value, in order to simplify the computation of the approximately minimum number of iterations $I_m$, with $I_{min} \leq I_m$ in all cases. Therefore, inequation 7 is reduced into the mathematical problem of finding the minimum for $b$ that satisfies $\lceil \frac{a}{b} \rceil \leq c$, where $a, b, c \in \mathbb{N}$.

**Lemma 1** *The minimum* b *that satisfies* $\lceil \frac{a}{b} \rceil \leq c$ *is* $b_{min} = \lceil \frac{a}{c} \rceil$.

*Proof* Let us suppose, by contradiction, that $b'_{min} = b_{min} - 1 = \lceil \frac{a}{c} \rceil - 1$ also satisfies the condition $\lceil \frac{a}{b} \rceil \leq c$.

Starting from the definition of the ceiling function [13]:

$$\lceil x \rceil = min\{m \epsilon \mathbb{Z} : m \geq x\}, \tag{8}$$

we can say that $x \leq \lceil x \rceil$, or in our case $\frac{a}{b} \leq \lceil \frac{a}{b} \rceil$. Regarding Lemma 1 we can also say that $\frac{a}{b} \leq c$, or, which is equivalent, $b \geq \frac{a}{c}$. Concerning our assumption $b'_{min}$ also satisfies this condition, which means $b'_{min} (= b_{min} - 1) \geq \frac{a}{c}$, it implies that:

$$\left\lceil \frac{a}{c} \right\rceil - 1 \geq \frac{a}{c}, \tag{9}$$

which contradicts Definition 8 if we put $x = \frac{a}{c}$, because a number that is less than $\lceil x \rceil$ also satisfies the condition. This proves Lemma 1. □

By using Lemma 1 in inequation 7 and by rounding:

$$I_m = \left\lceil \frac{H/N}{\left\lfloor \frac{1}{N}\left(\frac{M_{ls}}{\left\lceil \frac{W/N}{c} \right\rceil N + 2P} - 2P\right) \right\rfloor} \right\rceil \tag{10}$$

For the case $c > W/N$, the proposed data partition does not allow to use all the available cores in parallel. In this case, which occur when the number of cores is high regarding the image size, the frame has to be divided into $c$ chunks, in both the horizontal and

vertical directions, approximately proportional to the frame sizes in both dimensions:

$$\frac{c_h}{c_v} \approx \frac{W}{H}, \tag{11}$$

where $c_h$ is the number of chunks in a row, and $c_v$ is the number of chunks in a column. Due to the fact that $c_v c_h \lessapprox c$, the $c_v$ is given by:

$$c_v = \left\lfloor \sqrt{\frac{cH}{W}} \right\rfloor. \tag{12}$$

$c_h$ can be computed as the maximum natural number that satisfies $c_v c_h \leq c$. The number of required iterations can be calculated in a similar way as it was previously done by using inequation 5. Furthermore, in the case that the number of available cores $c$ is higher than the number of LPAs, $c > WH/N^2$, the processing inside of a single LPA can be divided among multiple cores. However, the optimal division is problem dependent and is not considered in this general basic LVIP model.

#### *3.1.2 Processing Time*

Two main different processing components contribute to the total processing time: the serial processing component, executed on the *Main Processor*, and the parallel one executed on the *Co-Processors*. Thus, the total execution time $T$ is represented as:

$$T = T_{serial} + T_{parallel}, \tag{13}$$

where $T_{serial}$ is the serial execution time needed for data preparation and for generating the parallel threads, and $T_{parallel}$ is the time measured since the multiple threads start the execution in the *Co-Processors* till the last produced result is returned back to the *Main Processor*.

According to the parallelization model and the frame partitioning proposed in Subsection 3.1.1, $T_{parallel}$ corresponds to the sum of the individual processing times for the $I$ iterations sequentially executed. The time needed for each iteration ($i$) corresponds to the maximum processing time to execute iteration ($i$) on all *Co-Processors*, and can be found as:

$$T_{CP_{max}}(i) = \max_j (T_{CP}(i, j)). \tag{14}$$

Therefore, Eq. 13 can be rewritten as:

$$T = T_{MP} + \sum_{i=0}^{I_m - 1} T_{CP_{max}}(i) \tag{15}$$

where $T_{MP}$ is the total *Main Processor* time, previously referred as $T_{serial}$. $T_{CP}$ can be further decomposed into the computation time $T_{CP_{comp}}$ and the time spent on DMA data transfers $T_{CP_{DMA}}$.

$$T_{CP}(i, j) = T_{CP_{comp}}(i, j) + T_{CP_{DMA}}(i, j) + T_{CP_{synch}}(i, j),$$
$$i = 0..I_m - 1,\ j = 0..c - 1. \tag{16}$$

$T_{CP_{synch}}$ represents the thread synchronization time, spent on notifying the *Main Processor* that *Co-Processor j* finished computing iteration *i*. This is particulary important when the number of co-processors or iterations (I) is large, since by increasing the number of synchronization requests we decrease the effective bandwidth available for data transfers.

### 3.2 Model Refinement

The basic model can be refined according to the particular features of the LVIP and the characteristics of the architectures.

#### *3.2.1 Extending LPAs in Time*

The basic model and data partition proposed for LVIP have been developed by considering spatial locality in a single frame, but it can be easily extended to consider locality in other dimensions, such as time. Motion estimation is an example of such an LVIP, where an LPA is composed by the current MB and one or more reference search areas located in different frames. Advanced video coding allows to consider search areas in $r$ ($1 \leq r \leq 16$) different RFs.

According to the diagram in Fig. 6, the non-overlapped MBs in the CF have a size of $N \times N$, while each Search Area in a RF overlaps its neighbors by $P$. With these parameters inequation 5 becomes:

$$r(w_{max}N + 2P)(h_{max}N + 2P) + w_{max}h_{max}N^2 \leq M_{ls}. \tag{17}$$

By following the same approach as in Subsection 3.1.1, the minimum number of iterations $I_m$ becomes:

$$I_m = \left\lceil \frac{H/N}{\left\lfloor \frac{M_{ls} - 2Pr(\lceil W/(Nc) \rceil N + 2P)}{N(\lceil W/(Nc) \rceil (r+1)N + 2Pr)} \right\rfloor} \right\rceil. \tag{18}$$

#### *3.2.2 Introducing Data Dependencies*

Although the basic model assumes there are no data dependencies between different LPAs, it is not always

the case. For example in H.264/AVC these dependencies exist between neighboring MB in raster-scan order. They can be found in deblocking filtering (see Fig. 1), and even in motion estimation when the median value of neighbor motion vectors are used. To consider this further dependencies, a latency is introduced in the execution time. Respecting data dependencies, as soon as results from chunk $l$ of the $k$-th row are available, the computation of chunk $l+1$ of the $k$-th row can start. Although with such strategy results have to be communicated in a row basis, which introduces communication overheads, all cores will be processing in parallel after the first $c-1$ rows.

In the case where processing can continue from one frame immediately to the next, the latency will be only initial and does not have a significant impact in the throughput. Moreover, instead of introducing a latency corresponding to the processing time of one row, we can consider the latency in an iteration basis as it is illustrated in Fig. 8. By applying such approach no additional communication is needed. However, by considering different latencies, *Co-Processors* will be in different iterations at a given time. Since the latency of the $j$-th *Co-Processor* is $c-j-1$, the time required for iteration $i$ in Eq. 14 becomes:

$$T_{CP_{max}}(i) = \max_{j}(T_{CP}(i+c-j-1,\, j)). \tag{19}$$

### 3.2.3 Computational Load Balancing

Until now, the frame is divided into approximately equal chunks that are joined to the threads, supposing the computational load balance among them. However, some LVIP methods are data dependent. For example, since the image/video content is not uniform along the frame, the behavior (computational load) of the algorithm on simultaneous running threads can be different. In order to reduce this imbalance, a new balance algorithm is proposed herein.

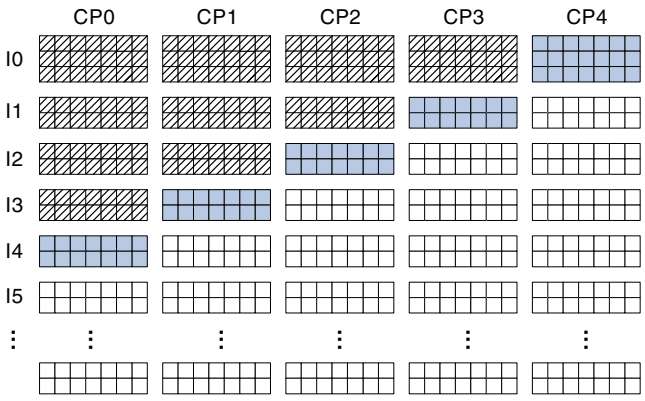


**Figure 8** Introduction of latency in the model, due to the data dependencies.

In this load balancing algorithm, the processing time is predicted for each LPA, based on the time measured in previous frame (video processing) or previous iteration (image processing). Actually, for each thread ($th$) and iteration ($i$), the average processing time per LPA is computed ($t^i_{lpa}$ in Algorithm 1). The balancing algorithm groups the LPAs in chunks, aiming to balance the predicted times ($t_{th}$) as much as possible. Initially, the average predicted time per chunk/thread ($t_{avr}$, alg. 1, line 0.1) is computed. A new LPA becomes a part of a chunk only if adding its predicted time ($t^{i-1}_{lpa}$) the thread's predicted time gets closer to the $t_{avr}$ (Algorithm 1, line 0.8, first condition). Moreover, at least one LPA has to remain to be processed on each of the remaining threads (Algorithm 1, line 0.8, second condition). Finally, the maximum number of the LPAs that can be processed per thread in a single iteration ($w_{max}$) is limited by the LS size. The $w_{max}$ can be found from Eqs. 4 and 5 as:

$$w_{max} \leq \frac{1}{N}\left(\frac{M_{ls}}{\left\lceil \frac{H/N}{I} \right\rceil N + 2P} - 2P\right). \tag{20}$$

As it can be seen in Eq. 20, $w_{max}$ increases with the number of iterations, which makes the balancing algorithm more flexible, since more LPAs that are predicted to be computationally light can be grouped together. On the one hand, the most flexible situation in terms of load balancing occurs when a single LPA line is processed per iteration. However, increasing the number of iterations can imply to slowdown data transfers. Therefore, the right number of iterations should be found as a tradeoff between these two opposite forces. In order to achieve better results, $t_{avr}$ and $T_{th}$ are kept updated (Algorithm 1, lines 0.6 and 0.9).

Finally, the rest of the LPAs are assigned to the last thread (Algorithm 1, line 1.0). However, if the number of the remaining LPAs is larger than $w_{max}$, the number of supernumeraries is divided among the threads, starting from the first one. It is supposed that first part of the frame is computationally light in this case (Algorithm 1, lines 1.2–1.5).

**Algorithm 1** Computational load balancing algorithm for one iteration

```
0.0 lpa=0;t_th=0;
0.1 t_avr=T^{i-1}/#th
0.2 for all threads
0.3   if not last thread
0.4     do
0.5       t_th+=t_lpa
0.6       T^{i-1}-=t^{i-1}_lpa
0.7       w_th++; lpa++;
0.8     while(|t_th + t^{i-1}_lpa − t_avr| < |t_th − t_avr|
          and #lpa − lpa < #th − th)
          and w_th < w_max
0.9     t_avr = T^{i-1}/(#th − th − 1)
1.0   else
1.1     w_th= min(#lpa-lpa, w_max)
1.2     lpa+= w_max; j=0;
1.3     while lpa < #lpa
1.4       if w_j < w_max
1.5           w_j++; lpa++;
1.6       end if
1.7       j= (j+1)mod(#th-1)
1.8     end while
1.9   end if
1.a end for
```

### 3.2.4 Tuning the Proposed Model for the Cell/BE

It is easy to map the adopted general heterogeneous multi-core architecture, depicted in Fig. 5, to the Cell/BE architecture: the *Main Processor* corresponds to the PPE and the *Co-Processors* correspond to the SPEs. Each SPE in the Cell/BE has two independent asymmetric execution pipelines [8]: one of them supports the algorithmic operations, while the other is responsible for loads, stores and branches. The existence of these two independent pipelines allows to overlap processing with memory access by exploiting the double buffering technique. In this case, the data transfer time $T_{CP_{DMA}}$ in Eq. 16 is replaced by the component of this time that does not overlap computation, $\Delta T_{CP_{DMA}}$. However, the penalty of using the double buffering technique is an increase of the memory size required, because independent memory space is needed for processing current data while new data is read. Consequently, to take in consideration this technique in the model we have to reduce the effective size of the *Local Store* to $M_{ls}/2$, and Eq. 10 becomes:

$$I_m = \left\lceil \frac{H/N}{\left\lfloor \frac{1}{N}\left( \frac{M_{ls}/2}{\left\lceil \frac{W/N}{c} \right\rceil N+2P} - 2P \right) \right\rfloor} \right\rceil \tag{21}$$

The lack of a dynamic branch prediction mechanism in the SPEs can also have a significant impact in the execution time. For example, for motion estimation with adaptive search algorithms the flow highly depends on the input data. Moreover SPEs provide a powerful SIMD data-path and vector instructions to process in parallel 128 bits. However, as it is shown in Subsection 4.1.3, data alignment is a key aspect for achieving high performance with its usage, so it should be considered in a parallelization model for the Cell/BE. By considering these two important characteristics of the Cell/BE, the computational time $T_{CP}$ becomes:

$$T_{CP} = T_{CP_{fc}} + T_{CP_{dap}} + T_{CP_{bmp}} + \Delta T_{CP_{DMA}} + T_{CP_{synch}}, \tag{22}$$

where $T_{CP_{dap}}$ represents the time spent on data alignment, $T_{CP_{bmp}}$ the branch misprediction penalty, and $T_{CP_{fc}}$ the rest of the computation time ("flat" computation). It should be noticed that the maximum of the $T_{CP}$ values for the $I_m$ iterations and the $c$ SPEs has to be estimated in order to obtain the $T_{CP_{max}}$(i) in Eq. 15.

## 4 Model Evaluation: Motion Estimation and Filtering on the Cell/BE

In the Cell/BE the PPE is a 64-bit general-purpose PowerPC core and eight SPEs are available, with specialized dual issue 128 bit architecture and 256 KBytes of *Local Store* for program and data. SPEs are specialized for signal processing and vector instructions are provided through the intrinsics library [15]. More than 512 MBytes of *Main Memory* is also available.

The control and the serial part of the H.264 video coder run on the PPE, while LVIP components are off-loaded to the SPEs to be implemented in parallel through multiple threads. CF and RFs are split in vertical slices and mapped into the SPEs according to the parallelization model defined in Section 3. Motion estimation, with sub-pixel accuracy, and deblocking filtering are the chosen LVIP operations for parallel implementation, taking advantage of the SIMD instructions of the Cell/BE multiple cores.

### 4.1 Motion Estimation

The regular FSBMr and the adaptive UMHS search algorithms were implemented by considering 7 different SB sizes (41 different SBs per MB).

Algorithm 2 represents a processing iteration on one SPE. The search algorithm is applied to all MBs in the chunk, over the considered SAs and RFs. In the case of FSBMr, data reusing is extensively used in each SPE. All the 16 $4 \times 4$ SBs that compose a MB are examined at once, and the remaining modes are evaluated based on these results. On the contrary, to implement the UMHS all the 41 SBs of a MB have to be independently examined (see lines 0.4–0.a). The following main sections and difficulties were identified

**Algorithm 2** SPE iteration for ME

```
0.0 main
0.1     initialization
0.2     sub-pixel interpolation on first RF
0.3     for all MBs in the chunk and RFs
0.4       if adaptive search
0.5         for all SBs
0.6           execute Search Algorithm
0.7         end for
0.8       else
0.9         execute Search Algorithm
0.a       end if
0.b       for all SBs
0.c         execute Sub-pixel refinent
0.d       end for
0.e     end for
0.f     send results
0.g end

1.0 procedure Search Algorithm
1.1     load MB pels in al.(igned) vectors
1.2     load SA pels in al. vectors
1.3     for all candidates in pattern
1.4       calculate SAD
1.5       update min SAD
1.6       if adaptive search
1.7         load SA pels in al. vects
1.8       else
1.9         update the oldest SA al. vect
1.a       end if
1.b     end for
1.c end procedure
```

when the parallel motion estimation was programmed on the Cell/BE, namely to efficiently use the 128-bit SIMD extensions in the SPEs: *i*) data alignment (lines 1.1, 1.2, 1.6–1.a), *ii*) parallel SAD computation (line 1.4), *iii*) minimum SAD updating (line 1.5), and *iv*) sub-pixel interpolation and search refinement (lines 0.2 and 0.b–0.d, respectively).

### *4.1.1 SAD Calculation*

Pixels on the current and reference SBs have to be packed in 16-byte vectors in order to compute in parallel the SAD by using the SIMD instructions (Algorithm 2 line 1.4). The /spu_ad a,b/ instruction calculates 16 absolute differences in parallel of byte elements from vectors *a* and *b*, and the /c=spu_sumb a,b/ instruction adds each four consecutive elements of *b* and stores the results in the even elements of vector *c*. The operation is repeated for *a* and the results are stored in the odd elements of *c*.

The scheme to compute the SAD in parallel for the FSBMr is shown in Fig. 9. The SADs for all 16 $4 \times 4$ SBs are computed at once. Initially current and reference MB pixels are stored in *MB[ ]* and *SA[ ]* vectors, respectively, placing each 16-byte line in one vector. Each four successive vectors contain four $4 \times 4$ SBs, which will be called SB rows. There are four SB rows, where either first and third (*k=0*) or second and fourth (*k=1*) are examined in parallel. Absolute differences *AD[ ]* vectors are summed using the /sumb/ instruction, as it is shown in Fig. 9, and accumulated for four entire vectors of an SB row (i=0..3). The result vector contains four SADs of the *k*-th SB row in odd elements and four SADs of the *k+2*-th SB row in even elements. The SADs for $4 \times 4$ SBs are extracted from these vectors

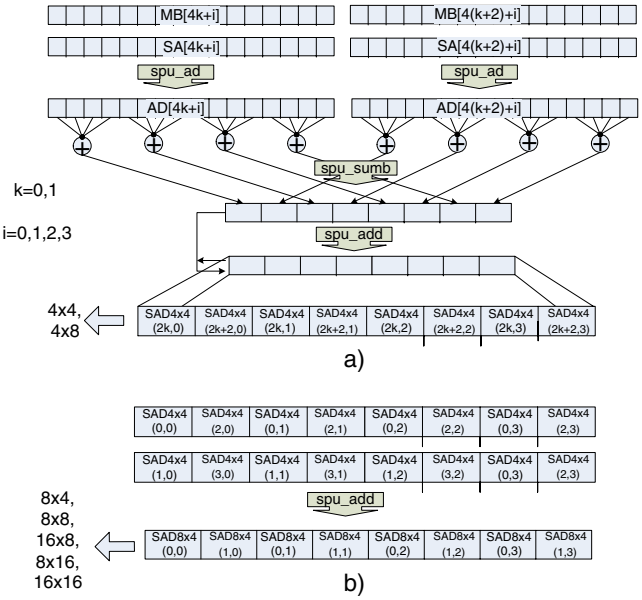


**Figure 9** SAD calculation **a** Calculating SAD of a row of $4 \times 4$ SBs **b** Calculating larger SBs.

while ones for $4 \times 8$ SBs are sequentially calculated by summing pairs of $4 \times 4$ SADs. The calculation for the rest of the SBs is presented in Fig. 9b). The obtained two vectors (for *k=0* and *k=1*) are added and the result represents the vector of the SADs of all $8 \times 4$ SBs ordered column by column. These results are also extracted, and the rest of the SADs are calculated sequentially.

In the case of the adaptive algorithm, the calculation has to be performed independently for each SB. A similar procedure to the one in Fig. 9a is adopted but with the important difference that all operations are performed individually for each SB. At the end of the process, the elements of the accumulation vector are extracted and summed, producing the final SAD.

### *4.1.2 Minimum SAD Updating*

One important characteristic of the SPE microarchitecture is the lack of a dynamic branch prediction mechanism. This aspect is important for example, to achieve high performance motion estimation since the computed SADs for all SB in a SA have to be compared in order to find the minimum value (Algorithm 2 lines 0.3–0.9 and line 1.5). Typically a conditional branch is executed for each comparison, but in this paper we propose to avoid conditional branches by updating in parallel the minimum SAD, using SIMD instructions.

Initially, all minimal distortion values are packed in vectors. These vectors are compared with the current SAD vectors using the instruction /c=spu_cmpgt a,b/. If the element of vector *a* is greater than the corresponding element of vector *b*, all the bits of the related element of vector *c* are set to one, otherwise, they are set to zero. This result, used directly as a pattern in the bitwise /spu_sel a,b,pattern/ instruction, produces a result vector containing the smallest elements found in the two compared vectors. Therefore, the minimum values are computed without using branch instructions.

Once again, this technique is not applicable when fast adaptive search algorithms are adopted, because in these algorithms search paths are usually different. As a result, SADs are not packed in vectors, but saved in independent variables in memory.

### *4.1.3 Data Alignment*

The requirement of 16 bytes data alignment in the SPE is an additional problem since no restrictions are imposed to the location of the input data in memory. Figure 10 illustrates the techniques herein adopted to overcome the data alignment problem (*h* is the height of an SB). Figure 10a illustrates the situation when SB is divided between two successive 16-Byte areas. Initially, the /spu_shuffle a,b,pattern/ instruction is applied *h* times on the two 16-Byte aligned vectors that an SB line belongs to. This intrinsic selects the bytes from *a* and *b* regarding the provided *pattern*, and produces a 16-Byte result. However, the width of most SBs is less than 16 Bytes so only part of the resulting vectors contain useful pixels. Therefore, additional packing is performed by applying the same instruction iteratively in order to fulfill the 16-Byte vectors with useful pixels. This packing is shown in Fig. 10b). In each iteration the number of applied /spu_shuffle/ instructions is reduced to half and the number of useful pixels in a 16-Byte vector doubles. While the current SB is always located inside one 16-Byte area, the reference SB can start from any position inside of the SA and the technique presented in the Fig. 10b has to be applied.

In the case of the FSBMr, all operations are performed on the whole MB and the lines of inner SBs do not need to be aligned. Figure 10c presents the data alignment process for the FSBMr, where only one /spu_shuffle/ instruction needs to be applied for each line of the reference MB. The regularity of the search is further exploited when successive MBs are examined. During the search process, all the candidates inside the SA are checked. In this case checking is performed column by column, instead of row major order, and when the next candidate inside the same column is examined 15 out of 16 vectors can be reused, which means only one additional vector needs to be aligned (see Algorithm 2 line 1.9). This technique decreases almost 16 times the time needed for data alignment in the FSBMr algorithm.

### *4.1.4 Sub-pixel Interpolation and Search Refinement*

Interpolation has to be applied prior to the sub-pixel search refinement. A RF once interpolated can be viewed as a new reference frame, and multiple interpolated RFs can be stored in the *Main Memory*. Therefore, the model proposed in Subsection 3.1 is also directly applied for sub-pixel motion estimation.

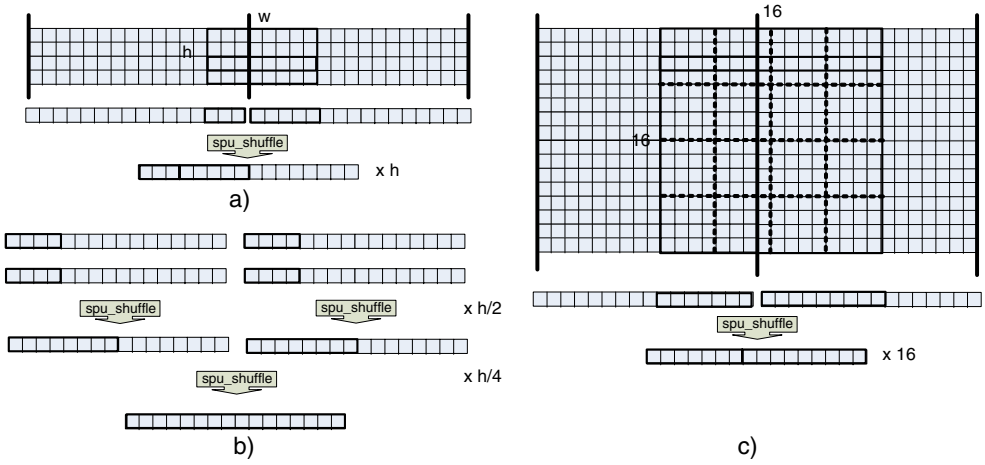


**Figure 10** Data alignment in parallel SAD calculation **a** Fast algorithm: alignment of divided SB lines **b** Fast algorithm: packing of SB lines **c** FSBMr: MB data alignment.

In the case of sub-pixel motion estimation, memory requirements significantly increase relatively to full-pixel motion estimation. Since the interpolated frame is larger, for example four times for half-pixel resolution, it is more probable that one row can not be cached in the available SPEs and therefore it increases the number of required iterations (see Fig. 10).

Finally, the techniques developed for programming full pixel FSBMr can not be applied to sub-pixel motion estimation, since the starting points for searching each SB are data dependent. However, techniques developed for adaptive search algorithms can also be applied to perform search refinement.

### 4.2 Deblocking Filtering

The model proposed in this paper can be directly applied for implementing the in-loop deblocking filtering on the Cell/BE, when latency is introduced as explained in Subsection 3.2.2. Since filtering on the edges between chunks requires three pixels on each side, the size of the overlapping area in the LPA is $P = 3$.

However, the adaptivity of the deblocking filtering makes it harder to be efficiently implemented on the Cell/BE architecture. The main problem are the conditions that have to be tested to decide which filter to apply (see Subsection 2.2). The solution adopted herein is to perform the computation for all the branch outputs, and after that to select the proper results according to the conditions associated to the deblocking filtering. With this approach, branches are skipped and SIMD parallelization is applied. The solution used to select the results with no conditional branches is similar to the one described in Subsection 4.1.2.

An iteration of the algorithm on one SPE is presented in Algorithm 3. Since only the inter frames are processed, the computation of BS can be avoided. However, to keep the generality of the solution this computation is taken into account. The filtering is performed firstly along the vertical direction and then along the horizontal direction. For horizontal filtering, the MB is transposed to make possible the usage of vector instructions, and the results are again transposed after filtering (Algorithm 3 lines 0.3–0.7). The example of computing the $p0$ pixel is presented (the notation is taken from Subsection 2.2; vector variables are bolded). The input value of the algorithm is the **BSeq4** vector that has on each position value '1' if the BS value for the related edge is 4, otherwise this value is '0'. The same algorithm is applied for all edges, while 16 pixels along the same edge are filtered simultaneously. The other conditional vectors are calculated within the procedure (Algorithm 3 lines 1.2 and 1.6). However, the output value $p0'$ is calculated for all the cases (Algorithm 3 lines 1.3–1.5). Finally, the *spu_shuffle* instruction is applied in order to select the value according to deblocking conditions.

**Algorithm 3** SPE iteration for In-loop Deblocking filtering

```
0.0 main
0.1   initialize the algorithm
0.2   for all MBs in the chunk
0.3     calculateBS
0.4     filter_vertical
0.5     transposeMB
0.6     filter_horizontal
0.7     inverse_transpose
0.8   end for
0.9   send results
0.a end

1.0 procedure filter_p0(BSeq4)
1.1   for 4 edges
1.2     calculate need_deblock
1.3     p'0_0=filter_p0(BSeq4==F)
1.4     p'0_1=filter_p0(BSeq4==T,p0_cond==F)
1.5     p'0_2=filter_p0(BSeq4==T,p0_cond==T)
1.6     calculate p0_cond
1.7     p'0_1=shuffle(p'0_1,p'0_2,p0_cond)
1.8     p'0=shuffle(p'0_1,p'0_2,BSeq4)
1.9     p0=shuffle(p'0,p0,need_deblock)
0.a   end for
1.b end procedure
```

## 5 Experimental Results

To experimentally evaluate the proposed parallelization model, we programmed the H.264/AVC coder on the Cell/BE. The JM 14.0 H.264/AVC reference software (JVT Reference Software unofficial version 12.0; http://iphome.hhi.de/suehring/tml/download) was ported to the Cell/BE based systems. The Blade QS20 platform was used with two Cell/BE processors (2 PPEs and 16 SPEs), an operating frequency of 3.2 GHz, two L2 512 KB caches on the PPEs, and 2 × 512 MB of main memory.

Both motion estimation and deblocking filtering were programmed to be computed in parallel in the SPEs according to the proposed parallelization model. Experimental results have been obtained by off-loading this LVIP to the SPEs, while the control and the other software components of the coder are executed on the PPE. The benchmark video sequences *akiyo*, *bus*, *foreman* (Common Intermediate Format (CIF), 352 × 288 pixels), and *blue_sky*, *riverbed* (high definition,

**Table 1** Number of iterations required for the different LVIP algorithms and video formats, 16 SPEs and five RFs.

| Video format | 352 × 288 | 720 × 576 | 1,280 × 720 |
|---|---|---|---|
| $Full-pixelME$ | 1 | 3 | 6 |
| $Sub-pixelME$ | 18 | 36 | 90 |
| $Deblockingfilter$ | 1 | 1 | 1 |

**Table 2** Motion estimation time per frame (milliseconds) for different SA sizes and different precision, and for five RFs.

| Sequence | SA radius (P) | Full-pixel | | Quarter-pixel | | DMA |
|---|---|---|---|---|---|---|
| | | UMHS | FSBMr | UMHS | FSBMr | all |
| Akiyo | 8 | 7 | 3 | 11 | 7 | 3 |
| (352 × 288) | 16 | 27 | 11 | 34 | 17 | 3 |
| Bus | 8 | 31 | 3 | 37 | 7 | 3 |
| (352 × 288) | 16 | 48 | 11 | 55 | 17 | 3 |
| Blue_sky | 8 | 86 | 9 | 26 | 24 | 8 |
| (720 × 576) | 16 | 138 | 33 | 156 | 53 | 8 |
| Riverbed | 8 | 92 | 9 | 107 | 24 | 8 |
| (720 × 576) | 16 | 121 | 33 | 136 | 50 | 8 |
| Blue_sky | 8 | 92 | 20 | 134 | 60 | 16 |
| (1,280 × 720) | 16 | 310 | 76 | 360 | 109 | 16 |
| Riverbed | 8 | 207 | 20 | 156 | 60 | 16 |
| (1,280 × 720) | 16 | 298 | 76 | 390 | 109 | 16 |

720 × 576 and 1,280 × 720 pixels) were used to obtain the experimental results herein presented. These video sequences were chosen because of their different characteristics regarding motion and spatial details. For the considered inter-frame motion estimation, all frames in a sequence, except the first one, are coded as P-frames; search ranges of 32 and 16 pixels (P=16,8), seven different MB shapes, five RFs, and a quantization step QP=28 are considered.

Table 1 shows the number of iterations $I_m$ required for implementing the LVIP algorithms according to the proposed model and Eqs. 18 and 21. For the H.264/AVC main profile and Blade QS20, the following values are adopted for the main parameters of the model: $M_{ls} = 210$ kBytes, $c = 16$, $r = 5$ and $P = 16$.

The processing time of both full-pixel and quarter-pixel precision motion estimation, when considering different resolutions, search algorithms, SA sizes and video sequences are presented in Table 2. Times were averaged for the first 50 frames. The experimental results show the efficiency of the parallel algorithms based on the proposed model, in particular for the FSBMr algorithm, where real-time motion estimation is achieved for 720 × 576 resolution, 32 pixels SA and five RF; about 30 and 20 frames per second for full-pixel and sub-pixel precision, respectively. However, the required time for the FSBMr algorithm grows linearly with the SA size. Although this can be a disadvantage when large SAs are considered, the performance of the FSBMr algorithm on the Cell/BE is always significantly better than the one achieved for the adaptive algorithm.

The advantages of the adaptive approach are mostly visible on the *akiyo* sequence, where the search stops after examining only a few candidates. The importance of regularity and video content independence in FSBMr is shown in sequences with more movement and spatial details, such as the *bus* sequence. In many cases, for the adaptive search, the processing time does not significantly decrease by reducing the SA size; this is mainly due to the fact that computation stops inside a small area, due to the applied thresholds to the SAD values. Finally, the processing time decreases linearly with the number of cores, but for the adaptive algorithm this decreasing is smaller mainly because there is no way of assuring complete load balancing since computation is data dependent.

As expected, motion estimation time increases when quarter-pixel resolution is considered. This increase is due to the additional interpolation and refinement phases, for which computational load is independent of the video content and of the search algorithm. In both

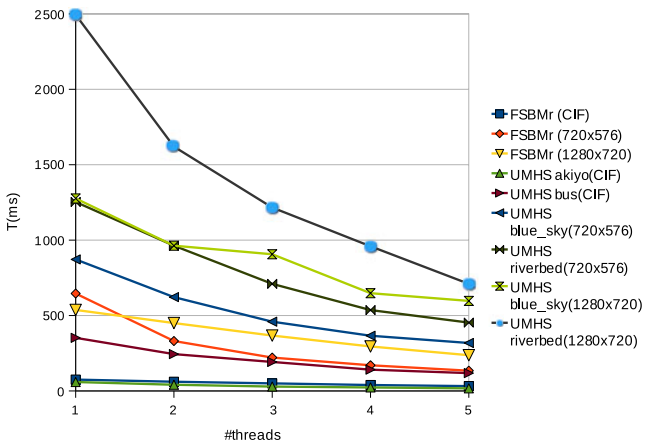


**Figure 11** Motion estimation time per frame (milliseconds), for different number of SPE threads.

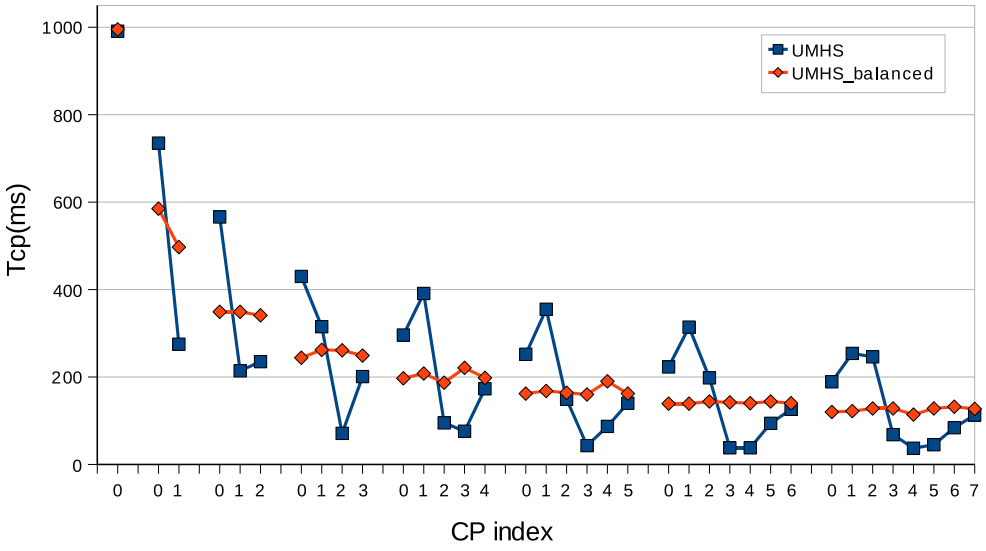


**Figure 12** Motion estimation time per frame (milliseconds), for the *blue_sky* sequence, the UMHS search algorithm, and different numbers of SPEs.

full-pixel and quarter-pixel cases, the DMA transfers are mainly hidden by using the double buffering technique, except in the case of the initial input data and final results.

Figure 11 depicts the time spent performing motion estimation on the Cell/BE, for the FSBMr and the UMHS, with a different number of parallel threads executed on the SPEs, and considering different resolutions. By comparing the time for a different number of threads and resolutions, it can be concluded that the parallel algorithms based on the proposed model are scalable with the number of cores. Moreover, it is shown that significantly lower processing times are obtained for the FSBMr, for any number of SPE threads used. The processing time for the FSBMr and full-pixel resolution decreases mainly linearly with the number of SPE cores. Non-linearity is mainly due to the change in the number of iterations whenever the number of cores is doubled.

Figure 12 presents the full-pixel motion estimation time on individual SPEs for the $720 \times 576$ *blue_sky* video sequence and the UMHS search algorithm. It clearly shows the effect of balancing of the computational load, given the dependency on the video content. In all multi-thread cases, the left half of the frame, that is examined on SPEs with lower indices, is more computational demanding. The difference between

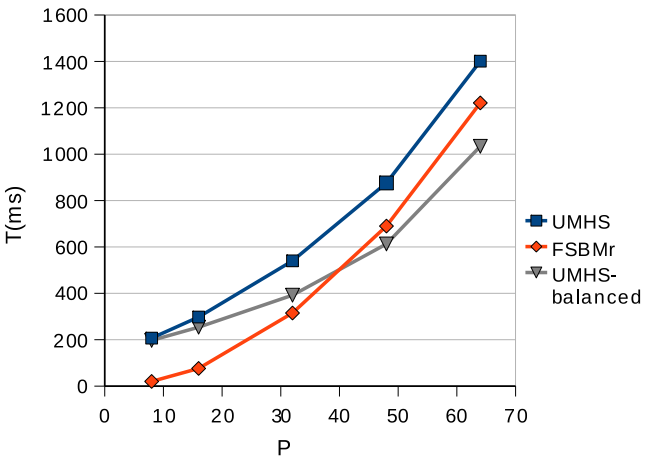


**Figure 13** ME time per frame (milliseconds), for the riverbed $1{,}280 \times 720$ sequence, five RFs and different values of P.

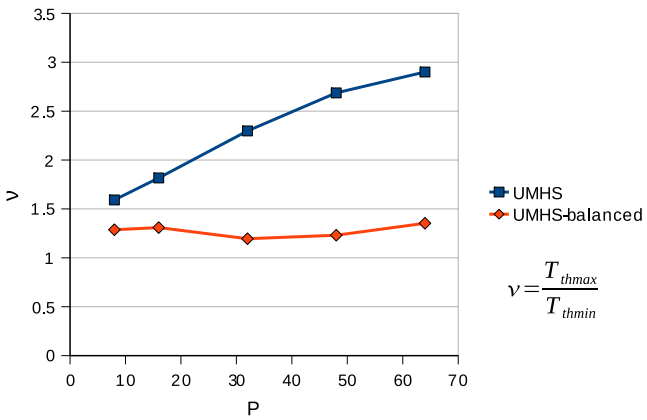


**Figure 14** Ratio between the fastest and the slowest thread for the UMHS and the riverbed $1{,}280 \times 720$ sequence.

individual SPE threads in the case of the unbalanced algorithm is significant. For example, for eight threads, SPE1 spends about seven times more time computing the same amount of data than SPE4. However in the model, as it is expressed in Eq. 15, only the $T_{CP_{max}}$ value is considered for the final performance. On the other hand, the balanced algorithm decreased this difference to less than 10%, and made the processing significantly faster.

In Fig. 13 the execution time per frame is presented for both unbalanced and balanced UMHS, as well as the FSBMr. Although the regular algorithm is significantly more efficient for the small SA, its execution time increases exponentially with P. While the unbalanced UMHS is slower than the FSBM for any values of SA, the balanced version outperforms the regular algorithm for larger SAs; the execution time is similar for P=32 and becomes smaller for larger values of P. Figure 14 depicts the ratio between the slowest and fastest threads for the same experiment, for both implementations of the UMHS. While the ratio for the unbalanced implementation increases with P, from 1.6 to 3, in the case of the balanced algorithm it remains below 1.35.

Figure 15 represents the time spent on data alignment, $T_{dap}$, for both the regular and the adaptive algorithm on 16 SPEs. With the exception of zero-based *akiyo* sequence, where the results are comparable, $T_{dap}$ for the regular algorithm is significantly lower. Although $T_{dap}$ almost linearly increases with the size of SA, it is much lower for the FSBMr than for the UMHS algorithm.

Figure 16 presents the impact of two techniques, namely cached SA alignment and parallel minimal distortion updating, in the motion estimation time.

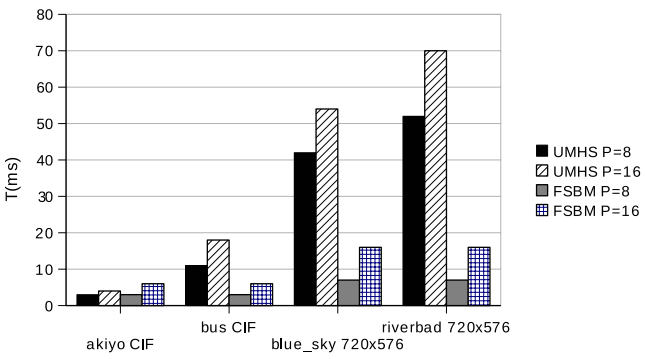


**Figure 15** Data alignment penalty (dap) for different algorithms and video sequences.

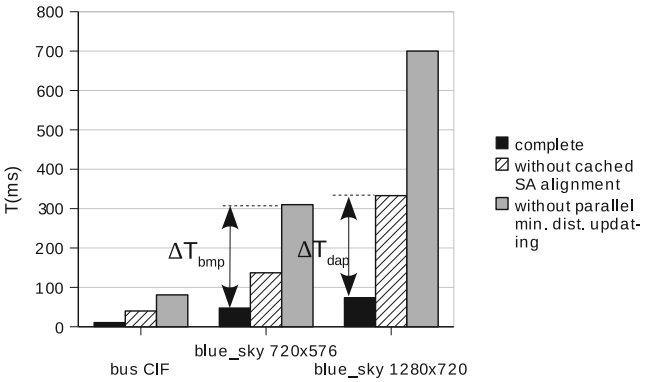


**Figure 16** The influence of applied techniques on motion estimation time per frame.

None of these techniques can be applied when adaptive search algorithms are adopted, as it was explained in Subsections 4.1.3 and 4.1.2 respectively. Parallel minimal distortion updating saves up to 65%, while the cached SA alignment saves up to 85% in the whole processing time. These techniques are especially efficient for high definition video sequences. The presented results are obtained for 16 SPE threads, full-pixel resolution, and the blue_sky $720 \times 576$ video sequence.

Table 3 presents the processing time per frame for the proposed deblocking filtering (vectorized) and straight forward solution (scalar), for different resolutions and number of SPE threads. In all cases the proposed method outperforms the straight forward solution more than one order of magnitude. Although the proposed method executes all the possible processing paths, it allows the usage of vector instructions for computing 16 pixels in parallel. Moreover, in the straight forward solution, apart from the branch misprediction penalty, there is a significant overhead related with scalar processing on the SPE architecture. The processing time decreases with the number of cores, slightly slower than linear, especially for a large number of cores. This slight decrease is mainly caused by the large number of simultaneous DMA requests that go to the PPE bus controller, and by communication and synchronization overheads.

Table 4 presents the times obtained for the different processing components of the deblocking filtering over $1{,}280 \times 720$ frames. The "comp. calc" denotes the sum of the times of these components, while the "comp. exp" represents the total time, which is experimentally obtained. The computational part spends most its time

**Table 3** Processing time (microseconds) per frame for both proposed (vectorized) and straight forward (scalar) deblocking filter for different resolution and number of cores.

| Resolution | Vectorized | | | | | Scalar | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 16 | 8 | 4 | 2 | 1 | 16 | 8 | 4 | 2 | 1 |
| Riverbed 1,280 × 720 | 162 | 316 | 632 | 1,223 | 2,514 | 2,306 | 3,477 | 6,002 | 10,321 | 21,450 |
| Bluesky 1,280 × 720 | 162 | 316 | 632 | 1,223 | 2,514 | 3,018 | 4,210 | 5,785 | 10,942 | 20,750 |
| Riverbed 720 × 576 | 89 | 151 | 280 | 573 | 1,215 | 1,329 | 1,649 | 2,828 | 5,398 | 10,393 |
| Bluesky 720 × 576 | 89 | 151 | 280 | 573 | 1,215 | 1,445 | 2,101 | 3,039 | 5,639 | 10,514 |
| Akiyo 352 × 288 | 30 | 45 | 78 | 140 | 281 | 681 | 957 | 1,405 | 1,967 | 3,665 |
| Bus 352 × 288 | 30 | 45 | 78 | 140 | 281 | 500 | 616 | 990 | 1,628 | 3,165 |

filtering, while the time spent transposing the matrix is smaller. Data read is done in parallel with the computation, so the time reading the data is hidden behind the computation time. However, writing cannot be done in such fashion as the SPE has to wait for the results, the filtered frame. Therefore, given that the time needed for reading the data is smaller than the computation time, only the computation time, the data write time, and the synchronization time contribute to the total time.

In Fig. 17, deblocking filtering times are presented for different numbers of iterations (see Subsection 3.1.1), as well as for the proposed $I_m$ value ($I_m = 1$, see last row of Table 1). The results are presented for different resolutions. In addition to the results obtain for 16 SPEs (solid lines), results obtained for eight SPEs are also provided (dashed lines). Apart from the increasing synchronization overhead explained in Section 3.1, data transfer time is the main reason why the filtering time increases with the number of iterations. Data transfers are performed in portions of 16 KBytes due to the Cell/BE architecture limitation. However, the usual size of data that should be transferred is not a multiple of 16 KBytes. Therefore, the last DMA transfer size in each iteration is less than 16 KBytes. Thus, the more iterations we have, the more data transfers smaller than 16 KBytes have to be done. As it was shown in [11], this increases the communication overhead. The deblocking time increases more slowly when the number of iterations is high, since the total size of data that needs to be transferred in a single iteration is lower than 16 KBytes, but it is still large enough for not decreasing the "total bandwidth". The results that can be experimentally obtained for smaller DMA transfers are not very stable, a lot of experiments have to be performed for such analysis (5,000,000 in [11]), and that is not the main aim of this work. In conclusion, it is experimentally shown how important is the proposed model to select the proper number of iterations.

**Table 4** Processing time (microseconds) per frame for deblocking filtering regarding different parts of both communication and computation.

| # threads | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|
| Chunk width(MB) | 3 | 6 | 12 | 23 | 45 |
| Computation | | | | | |
| filter_v | 48 | 95 | 190 | 365 | 805 |
| fw_transpose | 16 | 32 | 64 | 123 | 270 |
| filter_h | 52 | 103 | 206 | 396 | 899 |
| bw_transpose | 42 | 85 | 172 | 330 | 720 |
| Comp. calc | 158 | 315 | 632 | 1,214 | 2,694 |
| Comp. exp | 153 | 306 | 612 | 1,173 | 2,596 |
| DMA | | | | | |
| Data write | 10 | 18 | 48 | 108 | 220 |

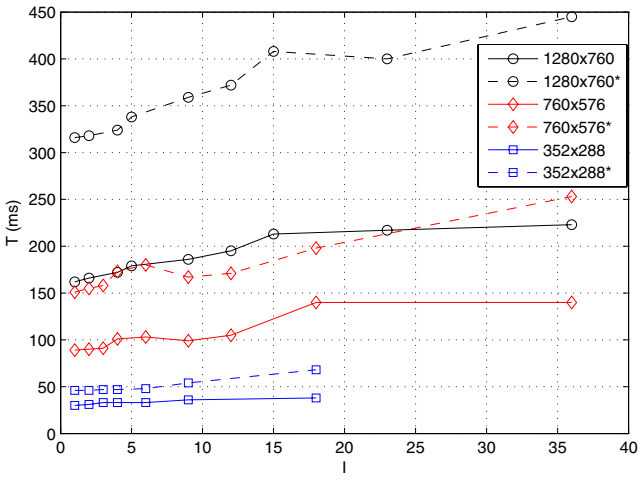


**Figure 17** The influence of increasing the number of iterations on deblocking filtering time with 16 and *8 SPEs.

## 6 Conclusions

This paper proposes a general model for LVIP on multi-core processors without hardware support for shared memory. The model introduces a general framework for the parallelization of LVIP local processing, by considering two different levels: a basic level that can be adopted for any multi-core processor, and a more fine grained level in which the particular details of each processor are taken into consideration. In the basic level, data partition and coarse-grained performance prediction are performed, while at the second level data dependencies and architectures' characteristics are introduced in order to refine the model and to obtain more accurate results.

Motion estimation and deblocking filtering from the H.264/AVC standard were used as case studies and were implemented in the IBM Cell/BE, which is a multi-core processor with non-shared memory. For the example of the in-loop deblocking filter it is shown that the execution of all the possible processing paths for avoiding branches, and to allow vectorization, significantly outperforms adaptive implementations with scalar instructions, which clearly shows the extreme importance of the vectorization on the actual processors. Both adaptive and regular exhaustive search algorithms with data reusing were considered for full-pixel motion estimation. Motion estimation with sub-pixel accuracy was also implemented. This example shows the advantages of applying regular processing, namely for data and intermediate results reusing, and for reducing branch misprediction. The cost of achieving regular processing by exhaustive search increases significantly with the size of the LPAs. Therefore, a new computational load balancing algorithm for adaptive algorithms is proposed in this work, which made the adaptive LVIP more competitive. Experimental results show the adequacy of the model and the efficiency of the proposed algorithms for performing real time LVIP, namely motion estimation is performed over 30 high resolution $720 \times 576$ video frames per second, even when considering large search areas and five reference frames.

**Acknowledgements** This work was partially supported by Portuguese Foundation for Science and Technology (FCT). The authors also acknowledge Georgia Institute of Technology, its Sony-Toshiba-IBM Center of Competence, and the National Science Foundation, for the use of the Cell Broadband Engine resources that have contributed to this research.

## References

1. Aji, A. M., Feng, W. c., Blagojevic, F., & Nikolopoulos, D. S. (2008). Cell-SWat: Modeling and scheduling wavefront computations on the Cell Broadband Engine. In *CF '08: Proceedings of the 5th conference on computing frontiers* (pp. 13–22). New York: ACM.
2. Alvarez, M., Salamí, E., Ramírez, A., & Valero, M. (2007). Performance impact of unaligned memory operations in SIMD extensions for video codec applications. In *ISPASS* (pp. 62–71).
3. Ates, H. F., & Altunbasak, Y. (2005). SAD reuse in hierarchical motion estimation for the H.264 encoder. In *Proceedings of the IEEE international conference on acoustics, speech, and signal processing 2005* (Vol. 2, pp. ii/905–ii/908). (ICASSP'05). Piscataway: IEEE Signal Processing Society.
4. Bader, D. A., Agarwal, V., Madduri, K., & Kang, S. (2007). High performance combinatorial algorithm design on the Cell Broadband Engine processor. *Parallel Computing, 33*(10–11), 720–740.
5. Blagojevic, F., Feng, X., & Cameron, K. W. (2008). Nikolopoulos, D.S.: Modeling multi-grain parallelism on heterogeneous multi-core processors: A case study of the Cell BE. In *Proc. of the 2008 international conference on high-performance embedded architectures and compilers*.
6. Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., et al. (2004). Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics, 23*(3), 777–786.
7. Chen, T., Raghavan, R., Dale, J. N., & Iwata, E. (2007). Cell broadband engine architecture and its first implementation: A performance view. *IBM Journal of Research and Development, 51*(5), 559–572.
8. Chen, T., Raghavan, R., Dale, J. N., & Iwata, E. (2007). Cell broadband engine architecture and its first implementation: A performance view. *IBM Journal of Research and Development, 51*(5), 559–572.
9. Chen, Z., Xu, J., He, Y., & Zheng, J. (2006). Fast integer-pel and fractional-pel motion estimation for H.264/AVC. *Journal of Visual Communication and Image Representation, 17*, 264–290.
10. Ciric, V., & Milentijevic, I. (2007). Area-time tradeoffs in H.264/AVC deblocking filter design for mobile devices. In *Proceedings of the IEEE conference on signal processing and its applications.* Sharjah, UAE.
11. Dou, Y., Deng, L., Xu, J., & Zheng, Y. (2008). DMA performance analysis and multi-core memory optimization for SWIM benchmark on the Cell processor. In *ISPA '08: Proceedings of the 2008 IEEE international symposium on parallel and distributed processing with applications* (pp. 170–179). Washington, DC: IEEE Computer Society. doi:10.1109/ISPA.2008.54.
12. Gonzalez, R. C., & Woods, R. E. (2006). *Digital image processing* (3rd ed.). Upper Saddle River: Prentice-Hall.
13. Graham, R. L., Knuth, D. E., & Patashnik, O. (1994). *Concrete mathematics: A foundation for computer science, chap. integer functions* (pp. 67–101). Boston: Addison-Wesley.
14. Hill, M., & Marty, M. (2008). Amdahl's law in the multicore era. *Computer, 41*(7), 33–38.
15. IBM: C/C++ language extensions for Cell Broadband Engine architecture, version 2.5 (2008). http://cell.scei.co.jp/e_download.html.

16. Sousa, L., Piedade, M. (1993). *Parallel algorithms: For digital image processing, computer vision and neural networks, chap. Low level parallel image processing* pp. 25–52. New York: Wiley.
17. Momcilovic, S., & Sousa, L. (2008). A parallel algorithm for advanced video motion estimation on multicore architectures. In *Complex, intelligent and software intensive systems, 2008. CISIS 2008. International conference on* (pp. 831–836).
18. Park, J., & Ha, S. (2007). Performance analysis of parallel execution of H.264 encoder on the Cell processor. In *ESTIMedia* (pp. 27–32).
19. Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., et al. (2008). Larrabee: A many-core x86 architecture for visual computing. *ACM Transactions on Graphics, 27*(3), 1–15.
20. Sotak, Jr., G. E., & Boyer, K. L. (1989). The Laplacian-of-Gaussian kernel: A formal analysis and design procedure for fast, accurate convolution and full-frame output. *Computer Vision, Graphics, and Image Processing, 48*(2), 147–189.
21. Weiss, B. (2006). Fast median and bilateral filtering. *ACM Transactions on Graphics, 25*(3), 519–526.
22. Wiegand, T., Schwarz, H., Joch, A., Kossentini, F., & Sullivan, G. J. (2003). Rate-constrained coder control and comparison of video coding standards. *Circuits and Systems for Video Technology, IEEE Transactions on, 13*(7), 688–703.

**Svetislav Momcilovic** received the M.Sc. degree in computer engineering from Faculty of Electrical Engineering, University of Nis, Serbia, in 2004. He is currently working toward the Ph.D. degree in the area of parallel video coding on multi-core systems at Instituto Superior Técnico (IST), Universidade Técnica de Lisboa, Lisbon, Portugal. He is also with the Signal Processing Systems Group (SiPS) of Instituto de Engenharia de Sistemas e Computadores R&D (INESC-ID). His research interests include parallel and distributed computing and video coding.

**Leonel Sousa** received the Ph.D. degree in electrical and computer engineering from Instituto Superior Técnico (IST), Universidade Técnica de Lisboa, Lisbon, Portugal, in 1996. He is currently an Associate Professor of the Electrical and Computer Engineering Department at IST and a Senior Researcher at Instituto de Engenharia de Sistemas e Computadores R&D (INESC-ID). His research interests include VLSI architectures, parallel and distributed computing and multimedia systems. He has contributed to more than 150 papers in journals and international conferences. He is currently a member of the HiPEAC European Network, an Associate Editor of the Eurasip Journal on Embedded Systems, and also a Senior Member of both IEEE and ACM.