Massively LDPC Decoding on Multicore Architectures

Gabriel Falcao, *Student Member*, *IEEE*, Leonel Sousa, *Senior Member*, *IEEE*, and Vitor Silva

Abstract—Unlike usual VLSI approaches necessary for the computation of intensive Low-Density Parity-Check (LDPC) code decoders, this paper presents flexible software-based LDPC decoders. Algorithms and data structures suitable for parallel computing are proposed in this paper to perform LDPC decoding on multicore architectures. To evaluate the efficiency of the proposed parallel algorithms, LDPC decoders were developed on recent multicores, such as off-the-shelf general-purpose x86 processors, Graphics Processing Units (GPUs), and the CELL Broadband Engine (CELL/B.E.). Challenging restrictions, such as memory access conflicts, latency, coalescence, or unknown behavior of thread and block schedulers, were unraveled and worked out. Experimental results for different code lengths show throughputs in the order of $1 \sim 2$ Mbps on the general-purpose multicores, and ranging from 40 Mbps on the GPU to nearly 70 Mbps on the CELL/B.E. The analysis of the obtained results allows to conclude that the CELL/B.E. performs better for short to medium length codes, while the GPU achieves superior throughputs with larger codes. They achieve throughputs that in some cases approach very well those obtained with VLSI decoders. From the analysis of the results, we can predict a throughput increase with the rise of the number of cores.

Index Terms—LDPC, data-parallel computing, multicore, graphics processing units, GPU, CUDA, CELL, OpenMP.

1 Introduction

Originally proposed by Robert Gallager in 1962 [1] and rediscovered by Mackay and Neal in 1996 [2], Low-Density Parity-Check (LDPC) codes have recently been adopted by new emerging standards for digital communication and storage applications, such as the DVB-S2 standard, WiMAX (802.16e), Wifi (802.11n), 10 Gbit Ethernet (802.3an), and others. LDPCs are linear (N, K) block codes defined by parity-check sparse binary $\mathbf{H}$ matrices of dimension $M \times N$, with $M = N - K$. They are usually represented by bipartite graphs formed by Bit Nodes (BNs) and Check Nodes (CNs) and linked by bidirectional edges, also called Tanner graph [3]. LDPC decoding is based on the belief propagation of messages between connected nodes as indicated by the Tanner graph, which demands very intensive computation running the Sum-Product Algorithm (SPA), or its simplified variants, namely the Logarithmic-SPA (LSPA) and the Min-Sum Algorithm [4].

Therefore, to achieve real-time processing, VLSI hardware processors are used, which usually apply the algorithm based on integer arithmetic operations [5]. In one of the first publications in the area [6], Blanksby and Howland describe a VLSI full-parallel architecture that achieves LDPC decoding with excellent throughput, which compares favorably against LDPC competitors, namely the Turbo code decoders. However, hardware represents nonflexible and nonscalable dedicated solutions [7], [8], which may involve many resources and require long and expensive development. More flexible solutions for LDPC decoding using Digital Signal Processors (DSPs) or Software Defined Radio (SDR) programmable hardware platforms [9] have already been proposed.

The integration of multiple cores into a single chip has become the new trend to increase processor performance. Multicore architectures [10] have evolved from dual- or quad-core to many-core systems, supporting multithreading, a powerful technique to hide memory latency, while at the same time provide larger Single Instruction Multiple Data (SIMD) units for vector processing. The general-purpose multicore processors replicate a single core in a homogeneous way, typically with a x86 instruction set, and provide shared memory hardware mechanisms. They support multithreading and share data at a certain level of the memory hierarchy, and can be programmed at a high level by using different software technologies [11]. OpenMP [12] provides an effective and relatively straightforward approach for programming general-purpose multicores and was selected under the context of this work.

Mainly due to the demands for visualization technology in the games industry, the performance of graphics processing units (GPUs) has undergone increasing performances over the last decade. With many cores driven by a considerable memory bandwidth, recent GPUs are targeted for computationally intensive, multithreaded, highly parallel computation, and researchers in high-performance computing fields are applying GPUs to general-purpose applications (GPGPU) [13], [14], [15], [16]. However, to apply GPUs to general-purpose applications, we need to manage very detailed code to control the GPU's hardware. To hide this complexity from the programmer, several programming interface tools [17], such as the Compute Unified Device Architecture (CUDA)

- *G. Falcao and V. Silva are with the Instituto de Telecomunicações and the Department of Electrical and Computer Engineering, University of Coimbra, 3030-290 Coimbra, Portugal. E-mail: {gff, vitor}@co.it.pt.*
- *L. Sousa is with the Department of Electrical and Computer Engineering, Instituto Superior Técnico, Technical University of Lisbon, and INESC-ID, Rua Alves Redol, 9, 1000-029 Lisboa, Portugal. E-mail: las@inesc-id.pt.*

Manuscript received 26 Nov. 2008; revised 28 Dec. 2009; accepted 15 Feb. 2010; published online 31 Mar. 2010.
Recommended for acceptance by H. Jiang.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2008-11-0466.
Digital Object Identifier no. 10.1109/TPDS.2010.66.

from NVIDIA [18], or the Close to the Metal (CTM) interface (replaced by the ATI Stream SDK) from AMD/ATI [19], or even the Caravela platform [20], have been developed. CUDA provides a new hardware and software architecture for managing computations on NVIDIA Tesla series' GPUs, and was selected as the programming interface in the work reported in this paper.

Also pushed by audiovisual needs in the industry of games emerged the Sony, Toshiba, and IBM (STI) CELL Broadband Engine (CELL/B.E.) Architecture (CBEA) [21], [22]. It is characterized by a heterogeneous vectorized SIMD multicore architecture composed by one main PowerPC Processor Element (PPE) that communicates very efficiently with eight Synergistic Processor Elements (SPEs).

Motivated by the evolution of these multicore architectures, we propose novel approaches for the computationally intensive processing of LDPC decoding. The main objectives of this paper are the investigation of efficient parallel algorithms for LDPC decoding, namely the intensive SPA; the development of compact and regular data structures to represent the Tanner graph, which are suitable for parallel computing; and the assessment of the performance of distinct multicore architectures for real-time LDPC decoding, namely by considering the obtained throughputs.

In this work, the proposed parallel algorithms, based on the multicodeword decoding principle, were programmed in different architectures, such as x86 general-purpose multicores, GPUs, and the CELL/B.E. Experimental results show interesting throughputs, namely for short and medium length codes on the CELL/B.E., which are comparable with VLSI-dedicated solutions [23]. By using the multithreaded data-parallel processing units of GPUs [13], software LDPC decoders can be expected to achieve a performance more than an order of magnitude higher than modern multicore CPUs. Moreover, since general-purpose multicores, GPUs, and the CELL/B.E. are able to perform floating-point arithmetic operations, better accuracy and a lower Bit Error Rate (BER) can be expected regarding to VLSI LDPC decoders [5].

This paper is organized into six sections. Section 2 describes belief propagation and a case study: the SPA used in LDPC decoding. Section 3 describes the parallel features of the architectures used and how to exploit data parallelism for LDPC decoding. Section 4 shows details of the parallelization approaches for the considered architectures and Section 5 reports experimental results and compares their performances. Finally, Section 6 concludes the paper.

2 Belief Propagation

Belief propagation, also known as the SPA, is an iterative algorithm [24] for the computation of joint probabilities on graphs commonly used in information theory (e.g., channel coding), artificial intelligence, and computer vision (e.g., stereo vision). It has proved to be an efficient algorithm for inference calculation and it is used in numerous applications including Low-Density Parity-Check codes, Turbo codes, stereo vision applied to robotics, or in Bayesian networks such as the QMR-DT (a decision-theoretic reformulation of the Quick Medical Reference (QMR) model [25]) depicted in Fig. 1.

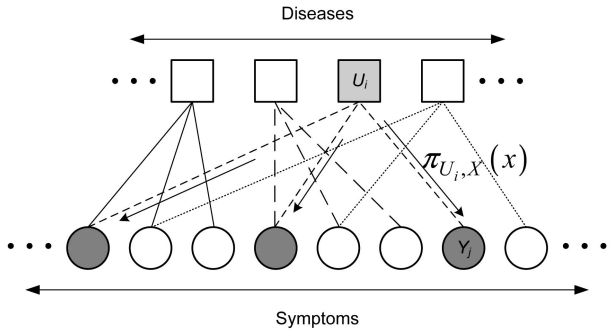

Fig. 1. A QMR-DT example showing the use of inference to obtain the maximum a posteriori probability (MAP) of a disease given all known symptoms [25].

Graphs are often used to denote the interrelationships among elements in a set, and in particular, bipartite graphs, can apply probabilistic techniques to bound the deviation of a random variable from its expected value. Given a probability graph, the belief propagation algorithm is based on the fact that vertices Y_j and U_i need to pass between them the messages $\lambda_{X,Y_j}(x)$ and $\pi_{U_i,X}(x)$, respectively, for vertex X, so that it has enough information to calculate the belief $\beta(x)$ of X. These messages can be recursively calculated from messages Y_j and U_i received from their neighboring nodes, where a neighbor denotes a node connected to another node by an edge. All vertices in the graph make a contribution passing messages to their neighbors, to update the beliefs in the graph, iteration after iteration. This message passing is also called propagation of evidence, a synonym of belief propagation.

2.1 LDPC Decoding

By considering a set of bits, or codeword, that we wish to transmit over a noisy channel, the theory of graphs applied to error correcting codes has fostered codes to performances extremely close of the Shannon limit [26]. In bipartite graphs with a large number of neighbors, the certainty of an information bit can be spread over several bits of a codeword, allowing in certain circumstances, in the presence of noise, to recover the correct value of a bit on the decoder side. In a graph representing a linear block error correcting code, reasoning algorithms exploit probabilistic relationships between nodes imposed by parity-check conditions that allow inferring the most likely transmitted codeword. One such algorithm is the belief propagation or SPA mentioned before, which finds the maximum a posteriori probability (MAP) of vertices in a graph [24].

In this paper, we parallelize the belief propagation algorithm applied to LDPC decoding of regular codes. In particular, the computationally demanding SPA is used for LDPC decoding. A bipartite Tanner graph represents adjacent connections between N BNs and M CNs (see Fig. 2 with $N = 8$ and $M = 4$) and is defined by a parity-check sparse binary $\mathbf{H}$ matrix of dimension $M \times N$. The SPA applied to LDPC decoding operates in the probabilistic domain [1], exchanging and updating messages between neighbors over successive iterations, until stop conditions occur. Given an (N, K) binary LDPC code, we assume BPSK modulation which maps a codeword $\mathbf{c} = (c_0, c_1, c_2, \ldots, c_{n-1})$ into a sequence $\mathbf{x} = (x_0, x_1, x_2, \ldots, x_{n-1})$, according to $x_i = (-1)^{c_i}$. Then, $\mathbf{x}$ is transmitted through an additive white

Gaussian noise (AWGN) channel, producing a received sequence $\mathbf{y} = (y_0, y_1, y_2, \ldots, y_{n-1})$ with $y_i = x_i + n_i$, and where n_i represents AWGN with zero mean and variance σ^2. The SPA applied to LDPC decoding is illustrated in Algorithm 1 and is mainly composed of two different horizontal and vertical intensive processing blocks defined, respectively, by (1), (2), and (3), (4). Equations (1) and (2) calculate the message update from CN_m to BN_n, considering accesses to $\mathbf{H}$ in a row-major basis—the horizontal processing—and indicate the probability of BN_n being 0 or 1. In Fig. 2, BN_0, BN_1, and BN_2 are updated by CN_0 as indicated in the first row of $\mathbf{H}$. From the second until the fourth row of $\mathbf{H}$, it can be seen that the subsequent BNs are updated by their neighboring CNs. For each iteration, $r_{mn}^{(it)}$ values are updated according to (1) and (2), as defined by the Tanner graph [27] illustrated in Fig. 2. Similarly, the latter pair of equations (3) and (4) computes messages sent from BN_n to CN_m, assuming accesses to $\mathbf{H}$ in a column-major basis—the vertical processing. In this case, $q_{nm}^{(it)}$ values are updated according to (3) and (4) and the edges connectivity indicated in the Tanner graph.

Algorithm 1 SPA

1: {Initialization}
$p_n = p(y_i = 1)$; $q_{mn}^{(0)}(0) = 1 - p_n$; $q_{mn}^{(0)}(1) = p_n$;
2: **while** ($\hat{\mathbf{c}}\,\mathbf{H}^T \neq \mathbf{0} \wedge$ it $<$ I) {c-decoded word; I-Max. no. of iterations.}
do
3: {For all node pairs (BN_n, CN_m), corresponding to $\mathbf{H_{mn}} = \mathbf{1}$ in the parity check matrix $\mathbf{H}$ of the code **do**:}
4: {Compute the message sent from CN_m to BN_n, that indicates the probability of BN_n being 0 or 1:}

(Kernel 1 - Horizontal Processing)

$$r_{mn}^{(it)}(0) = \tfrac{1}{2} + \tfrac{1}{2} \prod_{n' \in N(m) \backslash n} \left(1 - 2q_{n'm}^{(it-1)}(1)\right) \quad (1)$$

$$r_{mn}^{(it)}(1) = 1 - r_{mn}^{(it)}(0) \quad (2)$$

{where $N(m)\backslash n$ represents BNs connected to CN_m excluding BN_n.}
5: {Compute message from BN_n to CN_m:}

(Kernel 2 - Vertical Processing)

$$q_{nm}^{(it)}(0) = k_{nm}(1 - p_n) \prod_{m' \in M(n) \backslash m} r_{m'n}^{(it)}(0) \quad (3)$$

$$q_{nm}^{(it)}(1) = k_{nm} p_n \prod_{m' \in M(n) \backslash m} r_{m'n}^{(it)}(1) \quad (4)$$

{where k_{nm} are chosen to ensure $q_{nm}^{(it)}(0) + q_{nm}^{(it)}(1) = 1$, and $M(n)\backslash m$ is the set of CNs connected to BN_n excluding CN_m.}
6: {Compute the *a posteriori* pseudo-probabilities:}

$$Q_n^{(it)}(0) = k_n(1 - p_n) \prod_{m \in M(n)} r_{mn}^{(it)}(0) \quad (5)$$

$$Q_n^{(it)}(1) = k_n p_n \prod_{m \in M(n)} r_{mn}^{(it)}(1) \quad (6)$$

{where k_n are chosen to guarantee $Q_n^{(it)}(0) + Q_n^{(it)}(1) = 1$.}
7: {Perform hard decoding} $\forall n$,

$$\hat{c}_n^{(it)} = \begin{cases} 1 & \Leftarrow \quad Q_n^{(it)}(1) > 0.5 \\ 0 & \Leftarrow \quad Q_n^{(it)}(1) < 0.5 \end{cases} \quad (7)$$

8: **end while**

Finally, (5) and (6) compute the a posteriori pseudo-probabilities, and in (7), the hard decoding is performed at the end of an iteration. The iterative procedure is stopped if the decoded word $\hat{\mathbf{c}}$ verifies all parity-check equations of the code ($\hat{\mathbf{c}}\,\mathbf{H}^T = \mathbf{0}$), or if the maximum number of iterations ($\mathbf{I}$) is reached.

Let us consider an $\mathbf{H}$ matrix with M CNs (rows) and N BNs (columns), a mean row weight w_c and a mean column weight w_b of $\mathbf{H}$, with $w_c \geq 2$ and $w_b \geq 2$ (the weight

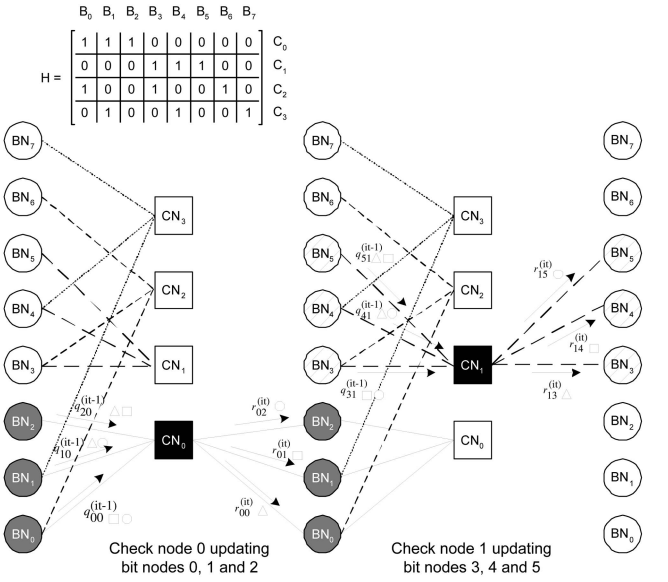

Fig. 2. An example of the Tanner graph and some messages being exchanged between nodes CN_m and BN_n.

represents the number of nonnull elements of $\mathbf{H}$). Depending on the size of the codeword and channel conditions (SNR), the minimal throughput necessary for an LDPC decoder to accommodate an application's requirements can imply a substantial number of (arithmetic and memory access) operations per second, which justifies the investigation of new parallelism strategies for LDPC decoding.

In Table 1, we present the computational complexity in terms of the number of floating-point add, multiply, and division operations required for both the horizontal and vertical steps in the SPA LDPC decoding algorithm. We also present the number of memory accesses required. For both cases, the complexity expressed as a function of w_c and w_b is quadratic.

The decoding complexity can, however, be significantly reduced by using the efficient Forward and Backward recursions [28] that minimize the number of memory accesses and operations necessary to update (1) to (4) in Algorithm 1. In Table 2, the number of operations required by the Forward and Backward algorithm adopted in this work shows linear complexity (as a function of w_c and w_b) with a significant reduction in the number of arithmetic operations. An even more significant decrease in memory access operations is registered, which contributes to increase the ratio of arithmetic operations per memory access, defined as arithmetic intensity. This property suits multicore architectures conveniently, which often have their performance limited by memory accesses. For a regular code with rate $= \frac{1}{2}$, where $N = 2M$, $w_c = 2w_b$, $w_c \geq 2$, and $w_b \geq 2$, the arithmetic intensity α_{SPA} for the SPA is

$$\alpha_{SPA} = \frac{\left(3w_c^2 - w_c\right)M + \left(2w_b^2 + 4w_b + 1\right)N}{\left(w_c^2 + w_c\right)M + \left(2w_b^2 + 2\right)N} \approx 2, \quad (8)$$

while the arithmetic intensity α_{FBA} for the Forward and Backward algorithm can be obtained by

$$\alpha_{FBA} = \frac{(9w_c - 8)M + (9w_b - 4)N}{3w_cM + (3w_b + 1)N} \approx 3. \quad (9)$$

TABLE 1
Number of Arithmetic and Memory Access Operations per Iteration for the SPA

SPA - Horizontal Processing	
	Number of operations
Multiplications	$2(w_c-1)w_cM$
Additions	$(w_c+1)w_cM$
Divisions	0
Memory Accesses	$(w_c-1)w_cM+2w_cM$
SPA - Vertical Processing	
	Number of operations
Multiplications	$2(w_b+1)w_bN$
Additions	$N+w_bN$
Divisions	w_bN
Memory Accesses	$((w_b-1)w_b+1)2N+2w_bN$

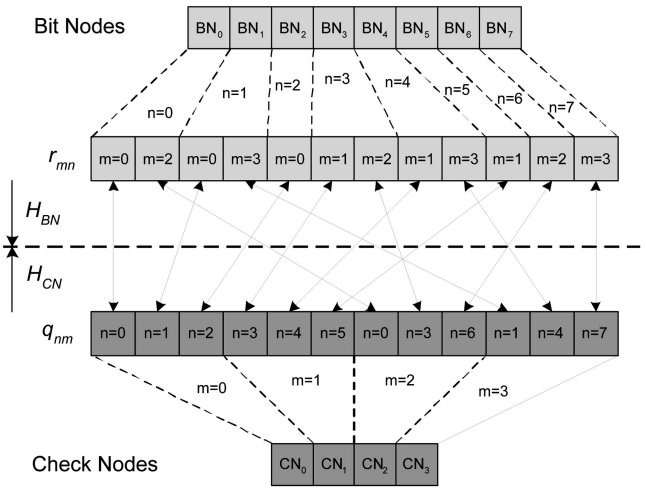

Fig. 3. Data stream structures representing the example in Fig. 2. Due to the random nature of LDPC codes, BN and CN messages associated with the same r_{mn} or q_{nm} equation are read from contiguous blocks in memory and stored in noncontiguous positions.

3 Data Structures and Parallel Computing Models

This section proposes compact data structures to represent the $\mathbf{H}$ matrix that are adequate to design new parallel algorithms suiting the considered data parallelism and the stream processing model. It also addresses the main features of the parallel architectures selected to support this work.

3.1 Data Structures

The $\mathbf{H}$ matrix of an LDPC code defines the Tanner graph, whose edges represent the bidirectional flow of messages exchanged between BNs and CNs. We propose to separately code the information about $\mathbf{H}$ in two independent data streams, $\mathbf{H_{BN}}$ and $\mathbf{H_{CN}}$, suitable for processing kernel 1 (horizontal processing) and kernel 2 (vertical processing), respectively, in the SPA described in Algorithm 1.

The example in Fig. 2 translated into the edge connections shown in Fig. 3 can be used to describe the transformation performed to $\mathbf{H}$ in order to produce the compact stream data structures. $\mathbf{H_{BN}}$ codes information about edge connections used in each parity-check equation (horizontal processing). This data structure is generated by scanning the $\mathbf{H}$ matrix in a row-major order and by sequentially mapping only the BN edges associated with non-null elements in $\mathbf{H}$ used by each CN equation (in the same row). These edges are collected and stored in consecutive memory positions inside $\mathbf{H_{BN}}$. Each element of the data structure records the address of the corresponding value of r_{mn} obtained from (1) and (2). By grouping related data into consecutive memory positions and using the Forward and Backward algorithm, it is possible to efficiently update all the BNs connected to a single CN with a higher level of parallelism.

A similar principle can be applied to code the information necessary for kernel 2 (vertical processing). $\mathbf{H_{CN}}$ can be defined as a sequential representation of the edges associated with non-null elements in $\mathbf{H}$ connecting every BN to all its neighboring CNs (in the same column). Each element of the $\mathbf{H_{CN}}$ data structure records the address of the corresponding q_{nm} value.

TABLE 2
Number of Arithmetic and Memory Access Operations per Iteration for the Optimized Forward and Backward Algorithm

Forward and Backward - Horizontal Processing	
	Number of operations
Multiplications	$(5w_c-6)M$
Additions	$(4w_c-2)M$
Divisions	0
Memory Accesses	$3w_cM$
Forward and Backward - Vertical Processing	
	Number of operations
Multiplications	$(6w_b-4)N$
Additions	$2w_bN$
Divisions	w_bN
Memory Accesses	$(3w_b+1)N$

3.2 Parallel Computational Models

General-purpose x86 microprocessors support the Multiple Program Multiple Data (MPMD) programming model. At hardware level, these systems have a relatively reduced number of cores that are able to exploit the instruction-level parallelism (ILP) for efficient single thread execution. This can be exploited to either parallelize the processing of a single kernel, or in alternative, to simultaneously launch the execution of distinct kernels.

The Single Program Multiple Data (SPMD) and/or SIMD approaches exploit data parallelism to achieve parallel processing. This data-parallel processing can be applied to multicodeword LDPC decoding, where multiple codewords are decoded simultaneously. In fact, this is possible because the Tanner graph structure is common to all data under parallel decoding. Also, intrinsic instructions for operating in the SIMD-within-a-register (SWAR) mode, an extension of SIMD where several data elements are kept and operated in parallel inside special registers (typically in the order of 128-bit), support additional parallelism. Within

a 128-bit register, depending on the resolution adopted, several codewords can be packed and processed together, which increases the arithmetic intensity of the program.

In vector processing, SIMD parallelism is applied to data elements, but the memory bandwidth is exploited in a relatively inefficient way. Stream processing minimizes this inefficiency by increasing the arithmetic intensity. In fact, the stream processing model applies kernels rather than single instructions to a collection of data. A kernel can have many instructions and perform a large number of operations per memory access. As the evolution of processing speed has been much superior to the increase of memory bandwidth, the stream programming model suits better modern architectures, such as GPUs.

Multithreading is the parallel programming paradigm used for multicore systems. The scheduling of multiple threads can be done by software or hardware, but scheduling is not usually under the control of the programmer. Kernel 1, kernel 2, and the data structures that represent the Tanner graph were carefully designed in order to minimize the workload unbalancing. For this reason, we investigated regular codes. Although the stream-based data structures here developed also support irregular codes, their adoption would be possible at the expense of unbalancing the workload.

3.3 Parallel Features of the General-Purpose Multicores

The reduced number of cores of general-purpose multicores typically ranges from 2 to 16 and includes different levels of hardware-managed cache memory [29]. In this memory hierarchy, the last cache level on the chip is in most cases shared by all the cores, and cache coherency is provided to support the shared-memory parallel programming model. Moreover, the internal cache shared by all cores allows efficient data transfers and synchronization between them. For these processors, parallel programming can be achieved using POSIX Threads (pthreads) or, at a higher level, by using the OpenMP programming interface [12].

Parallelizing an application by using OpenMP resources often consists in identifying the most costly loops, and provided that the loop iterations are independent, parallelize them via the *#pragma omp parallel for* directive. Based on a profile analysis, it is relatively straightforward to parallelize a sequential code, since LDPC decoding computes intensive loop-based kernels. Another possibility for OpenMP consists of using the *#pragma omp parallel section* launching the execution of independent kernels into distinct cores. This different approach suits simultaneous multi-codeword decoding in all cores.

3.4 Parallel Features of the GPU

The GPU used in this work is based on a CUDA [18], (a streaming computing platform from NVIDIA, where geometry, pixel, and vertex programs share common stream processors (SPs). The GPU has 16 multiprocessors, each one consisting of eight processors, which makes a total of 128 stream processors available, with 8,192 dedicated registers per multiprocessor [18] as depicted in Fig. 4. Also, the complexity involved in controlling all the GPU hardware resources has been masked by modern programming interfaces [18], [19], [20]. The GPU platform used provides

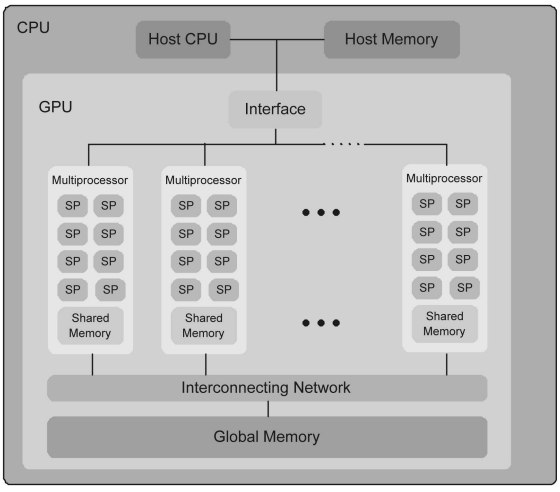

Fig. 4. The compute unified 8800 GTX GPU architecture and its 128 SPs.

shared memory that also allows to efficiently exploit data parallelism. Data-parallel processing is exploited with the CUDA by executing in parallel multiple threads.

The execution of a kernel on CUDA is distributed according to a grid of thread blocks with adjustable dimensions. In this platform, each multiprocessor has several cores as depicted in Fig. 4, and can control more than one block of threads. Each block controls a maximum of 512 threads that execute the kernel synchronously with threads organized in warps: the warp size is 32 threads and each multiprocessor time-slices the threads in a warp among its eight stream processors. The number of threads per block has to be programmed according to the specificities of the algorithm and the number of registers available on the GPU. Moreover, to achieve efficient execution on the GPU, the programmer has to be aware of the memory hierarchy in order to avoid the slow accesses to global memory and to take advantage of the faster coalesced accesses to memory.

A model to predict the throughput T of the parallel LDPC decoder on the GPU is presented in (11), where Th denotes the total number of threads running on the GPU, MP the number of multiprocessors, and SP the corresponding number of stream processors per multiprocessor. Each thread is expected to access memory with latency L per memory access, with a total of Mop memory accesses. $ND_{op/iter}$ represents the number of cycles for nondivergent instructions performed per iteration within a kernel, while $D_{op/iter}$ represents divergent instructions. Finally, N_{iter} defines the number of iterations, N is the size of the codeword, and f_{op} the frequency of operation of a stream processor. The processing T_{proc} time is

$$T_{proc} = N_{iter} \frac{\frac{Th}{MP} \times \left(\frac{ND_{op/iter}}{SP} + D_{op/iter} \right) + Th \times Mop \times L}{f_{op}}. \tag{10}$$

Then, the global throughput can be obtained by

$$T = \frac{P \times N}{T_{host \rightarrow gpu} + T_{proc} + T_{gpu \rightarrow host}} \quad [bps], \tag{11}$$

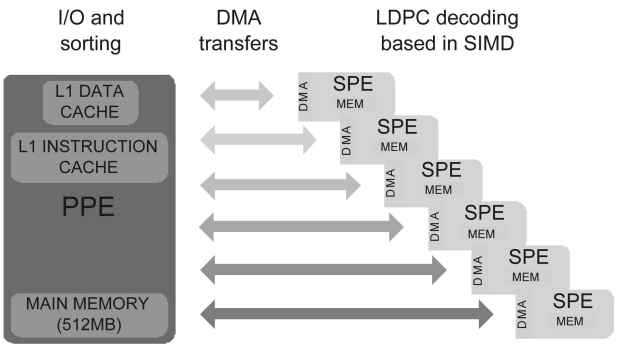

Fig. 5. Parallelization model for an LDPC decoder on the CELL/B.E. architecture.

where P defines the parallelism (number of codewords being simultaneously decoded). $T_{host \rightarrow gpu}$ and $T_{gpu \rightarrow host}$ represent data transfer times between host and device.

3.5 Parallel Features of the CELL/B.E.

The heterogeneous nature of the CELL/B.E. is composed by a 64-bit PPE that communicates with eight SPEs, as depicted in Fig. 5. An SPE has 256 Kbytes of local storage (LS) memory for data and source code, and a 128-bit wide vectorized SWAR-oriented architecture. Data transfers between the PPE (that accesses main memory) and SPE are performed by programming efficient Direct Memory Accesses (DMAs) that relieve the processors of the time-consuming task of moving data. In the CELL/B.E., data are loaded from the main memory into the LS of each SPE, and each SPE must efficiently exploit data locality.

The parallel LDPC decoder explores SPMD and SIMD data-parallel approaches, by also applying the same algorithm to different codewords on each SPE. When data and code environments do not fit completely into the LS of a single SPE, the processing has to be performed in successive partitions of the Tanner graph, which causes the number of data communications between the PPE and SPEs to rise significantly. The DMA latency can have a critical role in the performance, and a possible solution to reduce the impact of this problem in the performance consists of using a double buffering technique. In the CELL/B.E., each SPE has two pipelines executing independently, one for arithmetic and the other for load and store operations. The parallel algorithm for LDPC decoding implemented herein exploits this double buffering property by overlapping processing and data accesses.

The number of SPEs available $NSPEs$ and the number of instructions per iteration $N_{op/iter}$ have a major impact on the overall performance of the decoder on the CELL/B.E., where the throughput performance can be described according to the model defined in (12) for an LDPC code that fits on the LS. Different from what happens in the GPU, where a thread is responsible for decoding only a part of the Tanner graph, in the CELL/B.E. approach, each SPE performs the complete processing of the Tanner graph, as described in Algorithm 1, independently from the other SPEs. Each SPE processes in the SWAR mode 4 codewords in parallel. The LDPC decoder supports peak data transfers between the SPE and main memory of approximately 4 GB/s [30], and consequently, its performance is limited by the number of SPEs that are simultaneously trying to access the main memory. $T_{ppe \rightarrow spe}$ and $T_{spe \rightarrow ppe}$ represent data transfer times between PPE and SPE:

$$T = \frac{NSPEs \times 4 \times N}{T_{ppe \rightarrow spe} + \frac{N_{op/iter} \times N_{iter}}{f_{op}} + T_{spe \rightarrow ppe}} \quad [bps]. \qquad (12)$$

4 Parallelizing the Kernels Execution

The parallelization approach proposed for the LDPC decoder is explained first in a context that suits processing on general-purpose x86 architectures exploiting OpenMP pragmas. Then, it is given the perspective of a multi-threaded-based approach for GPUs using CUDA that hides latency problems, and finally, we address parallelism in the context of a heterogeneous CELL/B.E. architecture that exploits data locality and also SIMD.

Data structures r_{mn} and q_{nm} from Fig. 3 hold the messages exchanged between BNs and CNs. As mentioned before, BNs update CNs and vice versa. Each data structure possesses an array with the corresponding addresses (or indices) of the elements in the other array that it is going to update (here, represented by $\mathbf{H_{BN}}$ and $\mathbf{H_{CN}}$). A detailed view of their common dependencies can be found in Fig. 3. In each kernel, data elements can be read sequentially but have to be stored in nonconsecutive positions, which defines expensive random memory accesses. These costly write operations demand special efforts to the programmer in order to efficiently accommodate them in parallel architectures with distinct levels of memory hierarchy.

4.1 The Multicores Using OpenMP

Operations (1)-(4) in Algorithm 1 represent the most intensive processing in the SPA, which ideally should be performed in parallel. For programming general-purpose x86 multicores, an initial approach consists of exploiting OpenMP directives. Both horizontal and vertical kernels are based on nested loops. Considering the parallel resources allowed in the OpenMP execution, the selected approach consisted of parallelizing the outermost loop of the costly operations defined in (1)-(4), thus, reducing the parallelization overheads, as depicted in Algorithm 2. The loop data accesses were analyzed in order to identify shared and private variables in each iteration. This approach uses the *#pragma omp parallel for* directive to separately parallelize the processing of kernels 1 and 2 from Algorithm 1. An alternative approach uses the *#pragma omp parallel section* directive to launch several decoders in parallel, which allows to achieve multicodeword LDPC decoding, as represented in Algorithm 3. In this case, the different cores do not need to share data, but they must be synchronized upon completion of the execution of all kernels.

Algorithm 2. SPA kernels executing on general-purpose x86 multicores using OpenMP

1: {Initialization.}
2: **while** ($\hat{\mathbf{c}}\,\mathbf{H}^T \neq \mathbf{0} \wedge \mathrm{it} < \mathrm{I}$) {c-decoded word; I-Max. no. of iterations.}
do
3: {Compute all the messages associated to all CNs.}
4: $\#pragma$ omp parallel for
5: $shared(\ldots)private(\ldots)$
6: Processing **kernel 1**
7: {Compute all the messages associated to all BNs.}

8: $\#pragma$ omp parallel for
9: $shared(\ldots)private(\ldots)$
10: Processing **kernel 2**
11: **end while**

Algorithm 3. Multicodeword LDPC decoding on general-purpose x86 multicores using OpenMP

1: {launch decoder #1.}
2: $\#pragma$ omp parallel section
3: LDPC_decoder#1();
4: ...
5: {launch decoder #N.}
6: $\#pragma$ omp parallel section
7: LDPC_decoder#N();

4.2 The GPU Using CUDA

The computationally intensive two kernels in Algorithm 1 justify the usage of a high-performance specific computing engine, preferably a highly parallel programmable device. This approach is followed by considering a many-core system with a programming model based on multithreads, as the one supported by the NVIDIA 8 series' GPUs using CUDA.

4.2.1 Programming the Grid Using a Thread per Node Approach

In the beginning of a new computation, data to be processed ($\mathbf{H_{BN}}, \mathbf{H_{CN}}, \mathbf{r_{mn}}$, and q_{nm} data structures) are distributed over a predefined number of blocks on a grid. Each block contains $tx \times ty$ elements that represent threads, which are independent of the threads in other blocks, as shown in Fig. 6. Threads are grouped in warps and dispatched by blocks in one of the 16 multiprocessors according to the structure of the algorithm and the thread scheduler, as mentioned before. In the horizontal processing step, each thread updates all the BNs associated with a CN (an entire row of $\mathbf{H}$). Rather than moving data from global to fast (but complex) shared memory, an efficient processing strategy can also be obtained by moving data directly to the registers of the core. The number of registers available on the GPU is sufficiently high to accommodate all the input data and corresponding data structures, as long as we keep enough registers per thread to compile. Fig. 6 depicts the internal processing inside a block for a random example. The update of every BN message represented by r_{mn} inside of the block is calculated according to the structure previously identified in $\mathbf{H_{BN}}$.

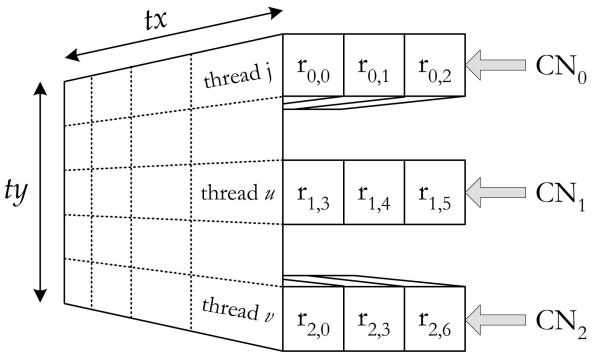

Fig. 6. Detail of $tx \times ty$ threads executing on the GPU grid inside a block for kernel 1 (three examples shown in Fig. 3 update all the BNs associated with CN_0, CN_1, and CN_2).

A similar principle applies to the update of q_{nm} messages, or vertical processing. Here, each thread updates all the CNs associated with a BN (an entire column of $\mathbf{H}$).

4.2.2 Coalesced Memory Accesses

In the 8 series GPUs from NVIDIA, the global memory is not cached. The latency can be up to 400-600 cycles and is likely to become a performance bottleneck. One way to turn around this problem and increase performance significantly entails the use of coalescence. Instead of performing 16 individual memory accesses, all the 16 threads of a half-warp (maximum fine-grain level of parallelism on the G80x family) can access the global memory of the GPU in a single coalesced read or write access, but elements have to lie on a contiguous memory block, where the kth thread accesses the kth data element, and data and addresses have to obey, respectively, to specific size and alignment requirements. Fig. 7 shows the activity of half-warps 0 and 1 captured at two different instants, where 16 threads (t_0-t_{15}) read data on a single coalesced memory transaction from the GPU's slow global memory. However, $\mathbf{H_{BN}}$, $\mathbf{H_{CN}}$, r_{mn}, and q_{nm} data structures shown in Fig. 3 have to be properly disposed in a contiguous mode (from a thread perspective) in order to allow the coalesced accesses to be performed in a single operation as shown in Fig. 7. A permutation has to be previously applied to data elements in order to allow such

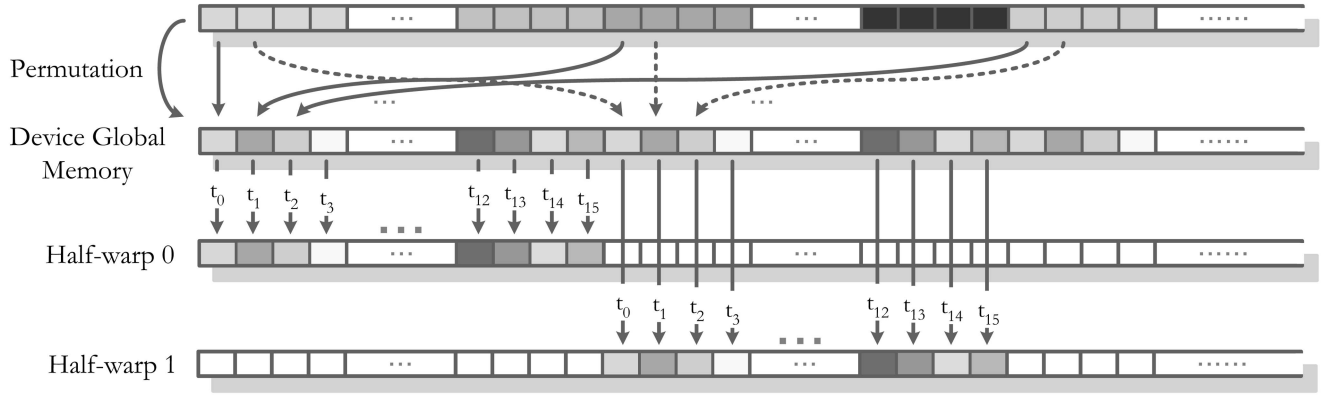

Fig. 7. Transformations applied to data structures to support coalesced memory read operations performed by the 16 threads of a half-warp on a single memory access.

procedure. For regular codes, (13)-(16) represent the necessary transformation for kernel 1, where

$$newAddr = j * p + k \; div \; w_c + (k \mod w_c) * 16, \quad (13)$$

with

$$p = w_c * 16, j = i \; div \; p, k = i \mod p,$$

and $0 \leq i \leq Edges - 1$. Then,

$$\breve{\mathbf{q}}^c_{nm}(i) = \mathbf{q}_{nm}(newAddr), \quad (14)$$

and the new memory addresses become

$$e = \mathbf{H}_{CN}(newAddr), \quad (15)$$

$$\breve{\mathbf{H}}^c_{CN}(i) = j * p + k \; div \; w_b + (k \mod w_b) * 16, \quad (16)$$

with $p = w_b * 16$, $j = e \; div \; p$, and $k = e \mod p$. The permutations necessary for kernel 2 are described from (17) to (20), where

$$newAddr = j * p + k \; div \; w_b + (k \mod w_b) * 16, \quad (17)$$

with $p = w_b * 16$, $j = i \; div \; p$, $k = i \mod p$, and

$$\breve{\mathbf{r}}^c_{mn}(i) = \mathbf{r}_{mn}(newAddr). \quad (18)$$

The new permuted memory addresses become

$$e = \mathbf{H}_{BN}(newAddr), \quad (19)$$

$$\breve{\mathbf{H}}^c_{BN}(i) = j * p + k \; div \; w_c + (k \mod w_c) * 16, \quad (20)$$

where $p = w_c * 16$, $j = e \; div \; p$, and $k = e \mod p$.

4.3 The CELL/B.E.

Following an alternative parallel approach, the design of a software LDPC decoder on the CELL/B.E. processor is also proposed in this work. This approach is based on an asymmetric threaded runtime programming model. The PPE controls the main tasks, offloading the intensive processing to the SPEs. Each SPE runs independently from the other SPEs, executing a predefined task by reading and writing data, respectively, from and to the main memory, through DMA. Synchronization between the PPE and the SPEs of the CELL/B.E. is performed using mailboxes. This approach explores data parallelism and data locality while performing the partitioning and mapping of the algorithm and data structures over the architecture, and at the same time, minimizes delays caused by latency and synchronization.

4.3.1 Small Single-SPE Model

Depending on the algorithm and on the size of data to be processed, some tasks are small enough to fit into a 256-Kbyte LS of a single SPE. This single task environment is sufficient for many dedicated workloads, as it is the case of LDPC decoders processing matrices A, B, and C shown in Table 4. Here, the DMA data transfers will take place only twice during the entire computation: at the beginning, before the processing starts; and after data decoding is concluded. In the single-SPE model, the number of communications between PPE and SPEs is minimum and the PPE is relieved from the costly task of reorganizing data (sorting procedure in Algorithm 4) between data transfers to the SPE.

4.3.2 Large Single-SPE Model

In some situations, however, data and code environments do not fit completely into the LS inside a single SPE. This is the case of the larger matrices under test, also detailed in Table 4. The approach followed in this case is described in Algorithms 4 and 5 and consists of having the PPE's control process managing the kernels execution and data transfers. Communications mainly control DMA transfers, and as in the previous case, are also set up using mailboxes. However, the number of data transfers between the PPE and SPEs is substantially higher in the large single-SPE model because a complete iteration is processed sequentially on partitions of the Tanner graph.

Algorithm 4. CELL/B.E. PPE side of the algorithm

1: Allocate *rsbuff*, *rubuff*, *qsbuff* and *qubuff* buffers
2: **for** $th\_ctr = 1$ to *NSPEs*: **do**
3: Create th_ctr thread
4: Send *rsbuff*, *rubuff*, *qsbuff* and *qubuff* addresses to the thread
5: **end for**
6: **repeat**
7: work = true
8: Receives y_n from the channel and calculates p_n probabilities
9: Send msg *NEW_WORD* to all SPEs
10: Sort p_n to buffer *qsbuff*
11: Send msg *SYNC* to all SPEs
12: **while** work **do**
13: **for** $i = 1$ to *NSPEs*: **do**
Ensure: Wait until mail is received (SPE[i].mailboxcount > 0)
14: msg = SPE[i].mailbox
15: **if** msg = *CHECK_BLOCK* **then**
16: Sort block *rubuff* to *rsbuff*
17: **if** buffer is sorted **then**
18: Send msg *SYNC* to SPE[i]
19: **end if**
20: **end if**
21: **if** msg = *BIT_BLOCK* **then**
22: Sort block *qubuff* to *qsbuff*
23: **if** buffer is sorted **then**
24: Send msg *SYNC* to SPE[i]
25: **end if**
26: **end if**
27: **if** all msg = *END_DECODE* **then**
28: work = false
29: break **for** loop
30: **end if**
31: **end for**
32: **end while**
33: **until** true

Algorithm 5. CELL/B.E. SPE side of the algorithm

1: Allocate *stdbyin*, *readbuf*, *writebuf* and *stdbyout* buffers
2: **repeat**
Ensure: Read mailbox (waiting a *NEW_WORD* mail from PPE)
3: Get p_n probabilities

```
 4:    for i = 1 to N_Iter: do
Ensure:      Read mailbox (waiting a SYNC from PPE)
 5:       Get qblock from qsbuff
Ensure:      Wait until DMA transfer finalizes
 6:       Get qblock from qsbuff
 7:       for j = 1 to N_CheckNodes_BLOCKS: do
 8:          Compute r_mn
 9:          if is last CheckNode of block then
Ensure:           Wait until all pending DMA transfers
                  finalize
10:             Put stdbyout
11:             if Not first block then
12:                Send msg CHECK_BLOCK
13:             end if
14:             if Not last 2 blocks then
15:                Get qsbuff to stdbyin
16:             end if
17:          end if
18:       end for
Ensure:      Wait until last DMA concludes
19:       Send msg CHECK_BLOCK
Ensure:      Read mailbox (waiting a SYNC from PPE)
20:       Get rblock from rsbuff
Ensure:      Wait until DMA transfer finalizes
21:       Get rblock from rsbuff
22:       for j = 1 to N_BitNodes_BLOCKS: do
23:          Compute q_nm
24:          if is last BitNode of block then
Ensure:           Wait until all pending DMA transfers
                  finalize
25:             Put stdbyout
26:             if Not first block then
27:                Send msg BIT_BLOCK
28:             end if
29:             if Not last 2 blocks then
30:                Get rsbuff to stdbyin
31:             end if
32:          end if
33:       end for
Ensure:      Wait until last DMA concludes
34:       Send msg BIT_BLOCK
35:    end for
36:    Send msg END_DECODE
37: until true
```

PPE. In the large model, the part of the algorithm that executes on the PPE side is presented in Algorithm 4. Mainly, it controls the overall execution mechanism on the SPE as well as data sorting on the buffers to be transmitted to the SPE. Four main buffers are used to manage the data flow over the several steps of the algorithm: *rsbuff*, *rubuff*, *qsbuff*, and *qubuff*. These buffers are used to manage the bit streams received at the input of the LDPC decoder in order to be sent to the SPE, and also the results received from each SPE. *rubuff* and *qubuff* denote buffers with unsorted data that, when sorted, are placed in *rsbuff* and *qsbuff* buffers, respectively, before being sent to each SPE. The sorting procedure consists of organizing BNs or CNs associated with the same equation in contiguous data blocks of memory, as shown in Fig. 3, to accelerate DMA data transfers. When the buffers are created, a number of threads equal to the number of SPEs are also created and the addresses of the different locations to be copied are passed to them. All the buffers to be transferred by DMA are aligned in memory on a 128-bit boundary.

For every chunk of data to be processed, two actions must be completed: 1) a subblock of the r_{mn} or q_{nm} data is sorted and loaded into the buffers to be transmitted to the SPE and 2) the processed data, BN or CN alternately updated, are sent from the SPE back to the PPE. The actions are synchronized on both sides of the PPE and SPE. After completing action 1, mails are sent to signal the threads on the SPEs, while after completing, confirmation mails are awaited from the SPEs.

When data are loaded into the buffers, two types of mails are used to start DMA transfers: for the first subblock of data, a *NEW_WORD* mail is sent to the SPE to make the thread start loading the *qsbuff* buffer, and when the transfer is completed, to initiate the load of the next data buffer. A *SYNC* mail informs the SPE to start transferring a new block of data from the *qsbuff* or *rsbuff* buffer.

Algorithm 4 shows the synchronization point on the PPE side after step 13, indicating that all the old information present in the SPE buffers has already been loaded and that it needs to sort a new subblock with r_{mn} or q_{nm} data. Here, the PPE expects mails from the SPEs notifying it that there are new data available in the *rubuff* or *qubuff* buffer to be sorted.

Finally, when all the iterations are completed, the SPEs send *END_DECODE* messages to the PPE to conclude the current decoding process and get ready to start processing a new word.

SPE. The SPEs are used for the intensive processing task of updating all BNs and CNs by executing kernel 1 and kernel 2 alternately, during each iteration of the LDPC decoder. The SPE pseudocode is presented in Algorithm 5. *Get* operations were adopted to represent a communication PPE $\rightarrow$ SPE, while *Put* operations are used to communicate in the opposite direction.

First, buffers *stdbyin*, *readbuf*, *writebuf*, and *stdbyout* are allocated. All the corresponding DMA addresses for these buffers are registered during this initial phase. Then, we initialize the process and start an infinite loop, waiting for communications to arrive from the PPE. The two types of messages expected from the PPE are *NEW_WORD* and *SYNC* mails. At first, when a *NEW_WORD* message is received, the SPE loads the p_n probabilities and waits for a *SYNC* mail to arrive. After that, the first horizontal block of q_{nm} is read into the *readbuf* buffer. After the DMA transfer completes, a new one is immediately started into the *stdbyin* buffer. The next transfer of a vertical block of r_{mn}, or a horizontal block of q_{nm}, will be performed in parallel with the execution of data previously loaded on the SPE. When a new *SYNC* mail is received, the SPE starts a new type of processing, for either BN or CN updating. When it completes, the SPE updates the buffers with the new data and then starts a new transfer.

After completing the processing of a kernel, it waits for the previous DMA transfer from *rsbuff* or *qsbuff* to complete and then performs a new DMA transfer to send data in the results buffer *writebuf* to main memory, placing it in *rubuff*

TABLE 3
Experimental Setup

	x86 CPU	GPU	CBEA	
Platform	Intel Xeon Nehalem 2x-Quad	NVIDIA 8800 GTX	STI CELL/B.E.	
Language	C + OpenMP 3.0	C + CUDA	C + SDK CELL 2.1	
OS	Linux kernel (Suse) 2.6.25	MS Windows XP	Linux kernel (Fedora) 2.6.16	
Number of cores	8	128	1 (PPE)	6 (SPEs)
Clock freq.	2.4GHz	1.35GHz (p/SP)	3.2GHz	3.2GHz
Memory	8MB (L2 cache)	768MB (VRAM)	256MB (main memory)	256KB (LS)

or *qubuff*. At this point, the next processing output can already start being placed in *stdbyout*. When the transfer is concluded, a confirmation mail is sent to the PPE indicating that the new data in the *rubuff* or *qubuff* buffer are available and can be sorted by the PPE in order to proceed with the next processing.

The computation terminates when the number of iterations is reached and an *END_DECODE* mail is sent by all SPEs to the PPE.

5 Experimental Results

The following parallel processing platforms were selected to evaluate the performance of the proposed parallel stream-based LDPC decoders: 1) 8 core Intel Xeon Nehalem 2x-Quad E5530 multiprocessor running at 2.4 GHz with 8 Mbytes of L2 cache memory; 2) NVIDIA 8800 GTX GPU with 128 SPs each running at 1.35 GHz and 768 Mbytes of VRAM memory; and 3) CELL/B.E., where the PPE and each SPE run at 3.2 GHz and have 256 Mbytes and 256 Kbytes of main and fast local memory, respectively. The CELL/B.E. is included in a PlayStation 3 (PS3) platform, which restricts the number of SPEs available to 6 from a total of 8. The experimental setup is depicted in Table 3 and matrices under test are presented in Table 4. The general-purpose x86 multicores were programmed using OpenMP 3.0 directives and compiled with the Intel C++ Compiler (version 11.0). The GPU was programmed using the C language and the CUDA programming interface (version 2.0b). The CELL/B.E. was programmed also in C using the CELL SDK compiler (version 2.1) running on a Fedora Core release, in accordance with both the small and large single-SPE models.

TABLE 4
Parity-Check Matrices under Test

Matrix	Size ($M \times N$)	Edges ($w_c \times M$)	Edges/row (w_c)	Edges/col.) (w_b)
A	128×256	768	6	3
B	252×504	1512	6	3
C	512×1024	3072	6	3
D	2448×4896	14688	6	3
E	2000×4000	16000	8	4
F	4000×8000	24000	6	3
G	10000×20000	60000	6	3

5.1 LDPC Decoding on the General-Purpose x86 Multicores Using OpenMP

The results presented in Table 5 were achieved using Algorithm 2. However, it should be noticed that using Algorithm 3 for multicodeword decoding only improves the results, on average, 20 percent. From the analysis of the results, it can be concluded that relatively low throughputs are achieved regarding those obtained for the CELL/B.E. and the GPU. The complex superscalar architecture of the individual cores is not suitable to exploit conveniently the data parallelism presented in the LPDC decoder.

However, it can be noticed that the proposed parallel approach for implementing LDPC decoders on general-purpose multicores with OpenMP shows to be scalable. For example, for Matrix F shown in Table 5, by varying the number of used cores in the range 2, 4, 6, and 8, we see the speedups raising consistently. Changing the parallel execution of the LDPC decoder in two cores to a different level of parallelism that uses four cores shows a speedup of 1.7. Compared with the same two cores execution, the parallelization with eight cores shows a speedup of approximately 3, but providing throughputs lower than a modest 2.6 Mbps value for 10 iterations.

5.2 LDPC Decoding on the CELL/B.E.

The model proposed in (12) makes it difficult to predict the execution time, not only because the number of instructions $N_{op/iter}$ generated by the CELL SDK 2.1 compiler is unknown, but rather due to the dual pipeline that performs memory accesses independently from arithmetic operations and whose behavior cannot be accurately predicted, due to branch misprediction penalties that force emptying the pipeline. To show that this model is accurate, upper and lower bounded processing times were considered, respectively, for worst- and best case working conditions. The worst

TABLE 5
LDPC Decoding Throughputs (Megabits per Second) for OpenMP and x86 Multicores

Iter.	Matrix C				Matrix F			
	Number of cores used							
	2	4	6	8	2	4	6	8
10	0.69	1.27	1.73	2.08	0.88	1.50	2.15	2.55
25	0.30	0.54	0.73	0.87	0.47	0.77	1.03	1.15
50	0.16	0.29	0.41	0.46	0.26	0.46	0.59	0.61

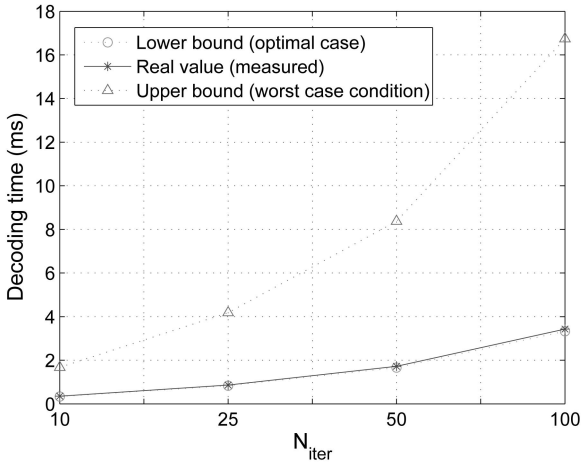

Fig. 8. Model bounds for LDPC decoding execution times obtained on the CELL/B.E. for the small model.

scenario assumes processing and memory accesses being performed without parallelism, and a misprediction branch after every instruction (a very unrealistic condition). The best case scenario (lower bound) assumes that no branch mispredictions exist and that there is full parallelism between memory accesses and computation. As expected, the parallelism supported by the dual-pipeline CELL/B.E. architecture exploited the LDPC decoder near the optimal case, as shown in Fig. 8. In the small model, the measured experimental values are, on average, only 6 percent worse than in the best scenario and the algorithm executes very near the maximum performance the CELL/B.E. can provide. Experimental values and lower bound almost overlap in Fig. 8.

5.2.1 Small Single-SPE Model

Fig. 9 presents the decoding times of a codeword for matrices A-C, under the small single-SPE model described in Section 4.3.1. The sequential LDPC decoder was programmed using only the PPE with SIMD and dual thread execution (it decodes concurrently eight codewords). In the parallel version, the CELL/B.E. performs concurrent processing on the six SPEs, each using SIMD to process four

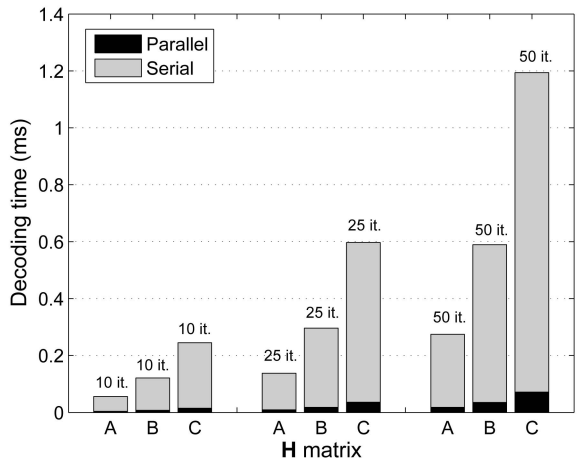

Fig. 9. LDPC decoding times on the sequential and parallel modes on the CELL/B.E. for the small single-SPE model.

TABLE 6
LDPC Decoding Throughputs (Megabits per Second) for a CELL/B.E. Programming Environment in the Small Single-SPE Model

Iter.	Matrix A	Matrix B	Matrix C
10	68.5	69.1	69.5
25	28.0	28.3	28.4
50	14.2	14.2	14.3
Data structures size (KByte)	35.8	70.6	143.4

codewords simultaneously. The CELL/B.E. takes approximately $353.6\ \mu s$ to decode 24 codewords on the six SPEs ($14.7\ \mu s$ per codeword) for matrix C running 10 iterations, against $244.9\ \mu s$ per codeword on the serial version. Observing Table 6, we conclude that the decoding capacity per bit is high and approximately constant for all matrices under test, providing throughputs as high as 69.5 Mbps. Comparing this throughput with the one obtained in the serial approach, the speedup achieved surpasses 16. Experimental evaluation also shows that regarding the same algorithm applied only to one SPE, decoding four codewords, the full version that uses six SPEs, decoding 24 codewords, achieves a speedup only 14 percent below the maximum (6). In fact, the number of data transfers is low in this model, which means that the overhead in communications introduced by using a higher number of SPEs would be marginal.

5.2.2 Large Single-SPE Model

In this case, the 256 Kbytes of LS memory in the SPE is not enough to accommodate the complete data structures and the program. Therefore, the processing is performed in consecutive partitions of the Tanner graph, with the number of data transfers and mailbox synchronizations degrading the performance. Matrix C was also tested in this model (described in Section 4.3.2) to allow relatively assessing the two computational models. Since several data transactions have to be performed during a single iteration, data transfers between each SPE and the main memory suddenly become the most significant part of the global processing time. Also the intensive sorting procedure performed by the PPE, and the large number of data transactions involving the slow main memory, can cause some processors to temporarily idle. This is the main reason why in this case the throughput decreases to 9 Mbps for 10 iterations, which represents a throughput seven times lower than the one obtained using the small single-SPE model.

Matrices D-G are also too large to fit into a single SPE and have to be processed on a subblock by subblock basis, using this model. They all achieve throughputs inferior to 9 Mbps, which led to the conclusion that the CELL processor is not efficient for LDPC decoding with large $\mathbf{H}$ matrices.

Increasing the number of SPEs used in the architecture would cause the throughput to rise, but not exactly by the same percentage as in the small model. This is explained by the fact that the use of more SPEs further increases data communications and the effect of data sorting, which clearly become the bottleneck in the large model.

5.3 LDPC Decoding on the GPU Using CUDA

The model defined in (10) can be further detailed in order to allow the prediction of numerical bounds that can help describing the computational behavior of the LDPC decoder on the CUDA. One iteration executes kernel 1 and kernel 2, while memory accesses can be decomposed into coalesced and noncoalesced read and write operations, each one imposing different workloads. The CUDA Visual Profiler tool was used in order to obtain the number of instructions and memory accesses per kernel. Considering, for example, the code for matrix G, which has a workload large enough to fully occupy 157 blocks of the grid with a maximum of 128 threads per block ($\#Th = 157 \times 128$), an average number of cycles per kernel (excluding load and store operations) similar in both kernels and equal to $ND_{op/iter} = 3867$, and having less than 0.5 percent of divergent threads which allow us to approximate $D_{op/iter} \approx 0$, (10) can be rewritten as

$$T_{proc} = \frac{2N_{iter} \times \frac{Th}{MP} \times \frac{ND_{op/iter}}{SP}}{f_{op}} + \underbrace{\frac{Th \times Mop \times L}{f_{op}}}_{T_{MemAccess}}. \quad (21)$$

$T_{MemAccess}$ defines all memory access operations and can be decomposed in memory accesses to CNs and BNs, respectively:

$$T_{MemAccess} = T_{MemAccBNs} + T_{MemAccCNs}. \quad (22)$$

$T_{MemAccBNs}$ defines time spent doing memory access to update CNs (accessing BNs):

$$T_{MemAccBNs} = N_{iter} \frac{M \times w_c \times L + M \times 2w_c \times L/16}{f_{op}}, \quad (23)$$

and $T_{MemAccCNs}$ represents the time necessary to update BNs (accessing CNs):

$$T_{MemAccCNs} = N_{iter} \frac{N \times w_b \times L + N \times (2w_b + 1) \times L/16}{f_{op}}. \quad (24)$$

The last partials in (23) and (24) show memory read operations, which are divided by 16 to model the parallelism obtained with the introduction of coalesced reading operations. In spite of the latency L can be up to 400-600 cycles, we believe that the thread scheduler operates very efficiently, producing effectively faster memory accesses. Apparently, the scheduler is good enough to maintain the cores occupied hiding the memory latency because our model predicts the processing time quite well, with an error inferior to 12.2 percent for $L < 5$ ($L = 4$ in this case). Transferring data between host and device, and transferring processed data back to the host, represents an increase of 8.9 percent in the global processing time. The accuracy of the proposed model best fits programs with large workloads that completely occupy the grid on the GPU.

5.3.1 Experimental Results on the CUDA

All matrices under test were run on blocks with a varying number of threads ranging from 64 to 128. Only the best results achieved are reported in Table 7. Matrix C was programmed to use 16 blocks and 64 threads per block. Using more threads per block would have been possible, but it would lower the number of simultaneous active blocks below 16, and in that case, the 16 multiprocessors wouldn't be fully occupied, decreasing parallelism and performance. For matrix D, 128 threads per block and 39 blocks are considered, while for matrix F, 63 blocks and 128 threads per block, and for matrix G, 157 blocks and also 128 threads per block are adopted. The number of blocks used is imposed by the size of the LDPC code. At the same time, it also depends on the number of threads per block, which are limited in the LDPC decoder to 128 due to the high number of registers required per multiprocessor. Table 7 presents the overall measured decoding throughputs for execution on an 8800 GTX GPU from NVIDIA, considering 32 and 8-bit precision data elements. Experimental results in the table show that the GPU-based solution can be faster than the CELL-based one for LDPC codes with dimensions above matrix C. For codes above matrix C, the CELL/B.E. presents 9 Mbps or less, against the values in Table 7. The GPU-based implementation shows significantly higher throughputs for intensive computation on large quantities of data. It is, however, likely that beyond a certain dimension of an LDPC code, the bottleneck will lie in the memory bandwidth between the host main memory and the VRAM on the GPU. A throughput above 18.3 Mbps is achieved for matrix F executing 10 iterations, while it decreases to 11.3 Mbps for the larger matrix G. In the latter, the size of the data structures does not fit into the 64 Kbytes of constant memory used in smaller codes (F and below), which degrades the performance. For small codes and small number of iterations, better results are achieved with fewer threads per block, although the decoding time difference is not significant. However, when decoding larger codes with a higher number of iterations, the reported speedups are higher when a larger number of threads per block are used.

TABLE 7
LDPC Decoding Throughputs (Megabits per Second) for a CUDA Programming Environment

Iter.	Matrix C		Matrix D		Matrix F		Matrix G	
	32bit	8bit	32bit	8bit	32bit	8bit	32bit	8bit
10	10.0	14.6	17.9	31.9	18.3	40.4	11.3	40.1
25	5.3	6.5	10.1	14.6	10.0	18.9	5.1	18.1
50	2.4	3.3	5.9	7.7	5.7	10.1	2.7	9.5

Fig. 10 shows the importance of memory accesses in the overall processing time as the size of the LDPC code increases. The random nature of LDPC codes does not allow to simultaneously read and write contiguous data blocks in memory; one of these two operations has to be performed using noncontiguous accesses. We chose to perform noncontiguous write operations. Reports obtained for matrices C-G running on the GPU show that the time spent in coalesced memory reads is approximately constant (in percentage) for all matrices, while more expensive and often conflicting noncoalesced memory write accesses can represent a bottleneck after a certain dimension of the LDPC code. For matrix G, noncoalesced write operations occupy nearly 50 percent of the total decoding time, against only 24 percent in the case of matrix C.

We also tested the LDPC decoder using data with only 8-bit precision, instead of the original 32-bit precision,

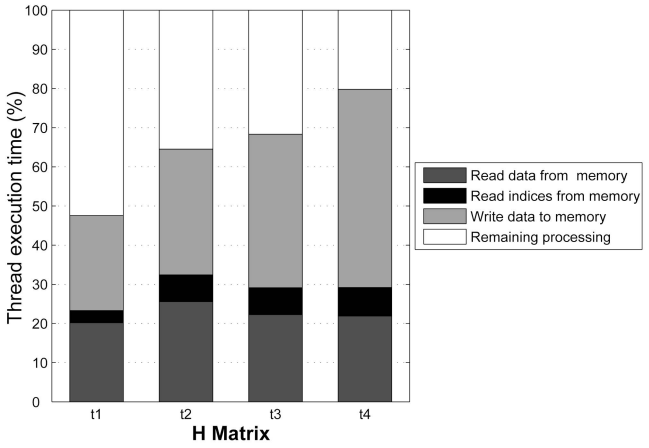

Fig. 10. Thread execution times for LDPC decoding on the GPU.

which allowed us to pack 16 elements into 128-bit data elements. In this case, using an 8-bit precision solution, the throughputs obtained for 10 iterations in matrices C-G range from 14.6 Mbps up to 40.1 Mbps. The reason for this improvement is mainly related with the reduction of memory accesses performed per arithmetic operation. On the GPU side, data are unpacked and processed, causing one memory access to read/write 16 elements at a time. Even considering the increase in arithmetic operations, the GPU still performs faster. Here, the limitation in performance is imposed mainly by memory accesses. The highest throughput is achieved for matrices F and G, which provide throughputs superior to 40 Mbps. Due to the use of the efficient Forward and Backward algorithm [28] that minimizes the number of memory accesses and the number of add and multiply operations, the experimental results here reported are much superior to the ones obtained in [13].

5.4 Discussion

Even by exploiting fast cache memory shared by multiple cores in parallel algorithms specifically developed for computation on general-purpose x86 multicores, we realized that the throughputs achieved are far from those requested by real-time applications. By exploiting data locality to minimize memory access conflicts and by using several SPEs, which support SIMD and a dual pipelined architecture, the CELL/B.E. performs better for small to medium LDPC codes. Here, the limitation in performance is imposed by the size of the LS memory on the SPEs. When decoding medium to large LDPC codes, the GPU solution achieves better results. In this case, the bottleneck is imposed by conflicting memory accesses and it is minimized using coalesced read operations that significantly improve the performance of the algorithm. The adoption of specific parallelization techniques for the last two platforms here described, produced throughputs that approach well hardware solutions such as the recent ones described in [9] and [31].

6 Conclusions

This paper proposes novel parallel algorithms for multicodeword LDPC decoding on multicore architectures. Compact and vectorized data structures were designed to represent the exchanged messages between connected nodes on the Tanner graph of an LDPC decoder. These data structures are suitable for parallel computing and allow a significant reduction of both the memory space and processing time necessary for LDPC decoding. Parallel LDPC decoders were developed on GPUs using CUDA and on the CELL/B.E., and significant throughputs were achieved in both platforms. The CELL/B.E. performed better than the GPU for small to medium LDPC codes, reporting throughputs above 68 Mbps, while the GPU deals better with larger codes, achieving throughputs between 10 and 40 Mbps. LDPC decoders have also been developed on general-purpose x86 multicores using OpenMP, but experimental results show that the achieved throughputs are low regarding the other considered multicore architectures. The solutions proposed in this paper seem to be scalable to the next generations of these architectures, regarding the increasing in the number of cores and the resources in each core, namely the size of local memory. They provide low-cost and flexible software-based multicore alternatives with performances that compare well to the expensive, long time-to-market, and hardware-dedicated typical VLSI LDPC decoder approaches.

References

[1] R.G. Gallager, "Low-Density Parity-Check Codes," *IRE Trans. Information Theory*, vol. 8, no. 1, pp. 21-28, Jan. 1962.
[2] D.J.C. Mackay and R.M. Neal, "Near Shannon Limit Performance of Low Density Parity Check Codes," *IEE Electronics Letters*, vol. 32, no. 18, pp. 1645-1646, Aug. 1996.
[3] R. Tanner, "A Recursive Approach to Low Complexity Codes," *IEEE Trans. Information Theory*, vol. 27, no. 5, pp. 533-547, Sept. 1981.
[4] J. Chen and M.P.C. Fossorier, "Near Optimum Universal Belief Propagation Based Decoding of Low-Density Parity Check Codes," *IEEE Trans. Comm.*, vol. 50, no. 3, pp. 406-414, Mar. 2002.
[5] L. Ping and W.K. Leung, "Decoding Low Density Parity Check Codes with Finite Quantization Bits," *IEEE Comm. Letters*, vol. 4, no. 2, pp. 62-64, Feb. 2000.
[6] A.J. Blanksby and C.J. Howland, "A 690-mW 1-Gb/s 1024-b, Rate-1/2 Low-Density Parity-Check Code Decoder," *IEEE J. Solid-State Circuits*, vol. 37, no. 3, pp. 404-412, Mar. 2002.
[7] T. Zhang and K. Parhi, "Joint (3,k)-Regular LDPC Code and Decoder/Encoder Design," *IEEE Trans. Signal Processing*, vol. 52, no. 4, pp. 1065-1079, Apr. 2004.
[8] J. Dielissen, A. Hekstra, and V. Berg, "Low Cost LDPC Decoder for DVB-S2," *Proc. Conf. Design, Automation and Test in Europe (DATE '06)*, Mar. 2006.
[9] S. Seo, T. Mudge, Y. Zhu, and C. Chakrabarti, "Design and Analysis of LDPC Decoders for Software Defined Radio," *Proc. IEEE Workshop Signal Processing Systems*, pp. 210-215, Oct. 2007.
[10] G. Blake, R.G. Dreslinski, and T. Mudge, "A Survey of Multicore Processors," *IEEE Signal Processing Magazine*, vol. 26, no. 6, pp. 26-37, Nov. 2009.
[11] H. Kim and R. Bond, "Multicore Software Technologies," *IEEE Signal Processing Magazine*, vol. 26, no. 6, pp. 1-10, Nov. 2009.
[12] B. Chapman, G. Jost, and R. Van Der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 2008.
[13] G. Falcao, L. Sousa, and V. Silva, "Massive Parallel LDPC Decoding on GPU," *Proc. 13th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP '08)*, pp. 83-90, Feb. 2008.
[14] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A.E. Lefohn, and T.J. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80-113, 2007.
[15] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream Computing on Graphics Hardware," *ACM Trans. Graphics*, vol. 23, no. 3, pp. 777-786, 2004.
[16] N. Goodnight, R. Wang, and G. Humphreys, "Computation on Programmable Graphics Hardware," *IEEE Computer Graphics and Applications*, vol. 25, no. 5, pp. 12-15, Sept. 2005.

[17] M. McCool, "Scalable Programming Models for Massively Multicore Processors," *Proc. IEEE,* vol. 96, no. 5, pp. 816-831, May 2008.
[18] CUDA Homepage, http://developer.nvidia.com/object/cuda.html, 2010.
[19] CTM Homepage, http://ati.amd.com/companyinfo/researcher, 2010.
[20] S. Yamagiwa and L. Sousa, "Caravela: A Novel Stream-Based Distributed Computing Environment," *Computer,* vol. 40, no. 5, pp. 70-77, May 2007.
[21] Int'l Business Machines Corporation, "CELL Broadband Engine Architecture," 2006.
[22] H. Hofstee, "Power Efficient Processor Architecture and the Cell Processor," *Proc. 11th Int'l Symp. High-Performance Computer Architectures (HPCA),* pp. 258-262, 2005.
[23] G. Falcao, V. Silva, and L. Sousa, "High Coded Data Rate and Multicodeword WiMAX LDPC Decoding on Cell/BE," *IET Electronics Letters,* vol. 44, no. 24, pp. 1415-1417, Nov. 2008.
[24] S.B. Wicker and S. Kim, *Fundamentals of Codes, Graphs, and Iterative Decoding.* Kluwer Academic Publishers, 2003.
[25] J.B. Lemaire, J.P. Schaefer, L.A. Martin, P. Faris, M.D. Ainslie, and R.D. Hull, "Effectiveness of the Quick Medical Reference as a Diagnostic Tool," *Canadian Medical Assoc. J. (CMAJ),* vol. 161, no. 6, pp. 725-728, 1999.
[26] S. Chung, G. Forney, T. Richardson, and R. Urbanke, "On the Design of Low-Density Parity-Check Codes within 0.0045 dB of the Shannon Limit," *IEEE Comm. Letters,* vol. 5, no. 2, pp. 58-60, Feb. 2001.
[27] S. Lin and D.J. Costello, *Error Control Coding,* second ed. Prentice Hall, 2004.
[28] D.J.C. Mackay, "Good Error-Correcting Codes Based on Very Sparse Matrices," *IEEE Trans. Information Theory,* vol. 45, no. 2, pp. 399-431, Mar. 1999.
[29] S. Kumar, C.J. Hughes, and A. Nguyen, "Architectural Support for Fine-Grained Parallelism on Multi-Core Architectures," *Intel Technology J.,* vol. 11, no. 3, pp. 217-226, Aug. 2007.
[30] J. Abellán, J. Fernández, and M. Acacio, "CellStats: A Tool to Evaluate the Basic Synchronization and Communication Operations of the Cell BE," *Proc. 16th Euromicro Int'l Conf. Parallel, Distributed and Network-Based Processing (PDP '08),* Feb. 2008.
[31] C.-H. Liu, S.-W. Yen, C.-L. Chen, H.-C. Chang, C.-Y. Lee, Y.-S. Hsu, and S.-J. Jou, "An LDPC Decoder Chip Based on Self-Routing Network for IEEE 802.16e Applications," *IEEE J. Solid-State Circuits,* vol. 43, no. 3, pp. 684-694, Mar. 2008.

Gabriel Falcao received the graduate degree from the University of Porto (FEUP), Portugal, in 1997, where he also received the MSc degree in the area of digital signal processing in 2002. He is a researcher at the Instituto de Telecomunicações and also performs a collaboration with INESC-ID under his PhD research activities. In 2003, he became a teaching assistant in the Department of Electrical and Computer Engineering at the University of Coimbra (FCTUC), Portugal. His scientific interests include parallel computing, VLSI architectures, LDPC codes, and digital signal processing. He is a student member of the IEEE.

Leonel Sousa received the PhD degree in electrical and computer engineering from the Instituto Superior Técnico (IST), Universidade Técnica de Lisboa, Portugal, in 1996. He is currently an associate professor in the Electrical and Computer Engineering Department at IST and a senior researcher at INESC-ID. His research interests include VLSI architectures and parallel and distributed computing. He has contributed to more than 150 papers in journals and international conferences. He is currently a member of the HiPEAC and an associate editor of the *Eurasip Journal on Embedded Systems*. He is a senior member of the IEEE and a member of the ACM.

Vitor Silva received the graduation diploma and PhD degrees in electrical engineering from the University of Coimbra, Portugal, in 1984 and 1996, respectively. He is currently an auxiliary professor in the Department of Electrical and Computer Engineering, University of Coimbra, where he lectures digital signal processing and information and coding theory. His research activities in signal processing, image and video compression, and coding theory are mainly carried out at the Instituto de Telecomunicações, Coimbra, Portugal. He published more than 70 papers and successfully supervised several postgraduation theses.