
**UNIVERSIDADE DE COIMBRA**

**FACULDADE DE CIÊNCIAS E TECNOLOGIA**

**DEPARTAMENTO DE ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES**


# Parallel Algorithms and Architectures for LDPC Decoding


**Gabriel Falcão Paiva Fernandes**

(Mestre)


Dissertação para a obtenção do grau de Doutor em

**Engenharia Electrotécnica e de Computadores**

Dissertação de Doutoramento na área científica de Engenharia Electrotécnica e de Computadores, especialidade Telecomunicações e Electrónica, orientada pelo Doutor Leonel Augusto Pires Seabra de Sousa do IST/UTL e pelo Doutor Vitor Manuel Mendes da Silva da FCTUC, e apresentada ao Departamento de Engenharia Electrotécnica e de Computadores da Faculdade de Ciências e Tecnologia da Universidade de Coimbra.

**Coimbra**

**Julho de 2010**

*Because we live in a parallel world, my graduate studies and in particular this thesis have seen many other persons participating actively, to whom I would like to thank. To my mother and father for having dedicated full priority to my education and in particular to my father, who has first shown me the importance and power of mathematics. To my brother who, accidentally, a few years ago, has found a newspaper announcing a TA position at the University of Coimbra (UC). I'd also like to thank father-, brother- and sisters-in-law who have been a friendly presence during my periods of absence at work, and in particular to M. Ramos who daily helped with the kids during this turbulent period. And to my wife, Gui, who has guided the ship through the storm with perseverance and enthusiasm during a three-year journey.*

*I thank colleagues M. Gomes for active collaboration over the last years and M. Falcão for having read parts of this text. Working with my students has been highly motivating, namely with (in seniority order) P. Ricardo Jesus, J. Cacheira, J. Gonçalves, V. Ferreira, A. Sengo, L. Silva, N. Marques, J. Marinho, A. Carvalho and J. Andrade. I thank them all. In the SiPS Group at INESC-ID, I would like to express my gratitude to S. Yamagiwa, D. Antão, N. Sebastião, S. Momcilovic and F. Pratas with whom I have worked. I'm also grateful to Professor T. Martinez (INESC Coimbra/UC) and P. Tomás (INESC-ID/IST) for the support with LaTeX. Four other persons had a very important role in my life while I was an undergraduate student at FEUP, and they are: S. Crisóstomo, N. Pinto, L. G. Martins and M. Falcão. I am also grateful to Professor Aníbal Ferreira (FEUP) who has introduced me in the world of signal processing and to Professor F. Perdigão (UC) for many enthusiastic discussions and advices. This work has been possible also thanks to the financial support of the Instituto de Telecomunicações (IT), INESC-ID and the Portuguese Foundation for Science and Technology (FCT) under grant SFRH/BD/37495/2007.*

*Professors Leonel Sousa from INESC-ID and Instituto Superior Técnico (IST) at the Technical University of Lisbon (TUL) and Vítor Silva from IT and Faculty of Sciences and Technology of the University of Coimbra (FCTUC), both at the respective Departments of Electrical and Computer Engineering, have supervised this thesis thoroughly. I thank them guidance and support, and for teaching me the fundamentals of computer science, parallel computing architectures, VLSI, coding theory and also the arts of writing. Their vision was essential to maintain the objectives in clear view.*

*To Guida, Leonor and Francisca*

# Abstract

Low-Density Parity-Check (LDPC) codes have recaptured the attention of the scientific community a few years after Turbo codes were invented in the early nineties. Originally proposed in the 1960s at MIT by R. Gallager, LDPC codes represent powerful error correcting codes that allow working very close to the Shannon limit and achieve excellent Bit Error Rate (BER) due to computationally intensive algorithms on the decoder side of the system. Advances in microelectronics introduced small process technologies that allowed developing complex designs incorporating a high number of transistors in Very Large Scale Integration (VLSI) systems. Recently, these processes have been used to develop architectures able of performing LDPC decoding in real-time and delivering considerably high throughputs. Mainly for these reasons and naturally because the patent has expired, they have been adopted by modern communication standards which triggered their popularity, showing how actual they are.

Due to the increase of transistor density in VLSI systems, and also to the fact that recently processing speed has risen faster than bandwidth, power and memory walls have created a new paradigm in computer architectures: rather than just increasing the frequency of operation supported by smaller process designs, the introduction of multiple cores on a single chip has become the new trend to provide augmented computational power. This thesis proposes new approaches for these computationally intensive algorithms, by performing parallel LDPC decoding based on ubiquitous multi-core architectures and achieves efficient throughputs that compare well with dedicated VLSI systems. We extensively address the challenges faced in the investigation and development of these programmable solutions, with focus mainly given on flexibility and scalability of the proposed algorithms, throughput and BER performance, and general efficiency of the programmable solutions here presented, that also achieve results more than an order of magnitude superior to those obtained with conventional CPUs. Furthermore, the investigation herein described follows a methodology that analyzes in detail the computational complexity of these decoding algorithms in order to propose strategies to accelerate their processing which, if conveniently transposed to other areas of computer science, can demonstrate that in this new multi-core era we may be in the presence of valid al-

ternatives to non-reprogrammable dedicated VLSI hardware that requires non-recurring engineering.

# Keywords

Parallel computing, Computer architectures, LDPC, Error correcting codes, Multi-cores, GPU, CUDA, Caravela, Cell/B.E., HPC, OpenMP, Data-parallelism, ILP, HPC, VLSI, ASIC, FPGA.

# Resumo

Os códigos LDPC despertaram novamente a atenção da comunidade científica poucos anos após a invenção dos Turbo códigos na década de 90. Inventados no MIT por R. Gallager no início da década de 60, os códigos LDPC representam sistemas correctores de erros poderosos que permitem trabalhar muito perto do limite de Shannon e obter taxas de bits errados (BER) excelentes, através da exploração apropriada de algoritmos computacionalmente intensivos no lado do descodificador do sistema. Avanços recentes na área da microelectrónica introduziram tecnologias e processos capazes de suportar o desenvolvimento de sistemas complexos que incorporam um número elevado de transístores em sistemas VLSI. Recentemente, essas tecnologias permitiram o desenvolvimento de arquitecturas capazes de processar a descodificação de códigos LDPC em tempo-real, obtendo taxas de débito de saída consideravelmente elevadas. Principalmente por estes motivos, e também devido ao facto do prazo de validade da patente ter expirado, estes códigos têm sido adoptados por normas de comunicações recentes, o que comprova a sua popularidade e actualidade.

Devido ao aumento da densidade de transístores em sistemas microelectrónicos (VLSI), e uma vez que nos tempos mais recentes a velocidade de processamento tem sofrido uma evolução mais rápida do que a velocidade de acesso à memória, os problemas associados à dissipação de potência e a tempos de latência elevados criaram um novo paradigma em arquitecturas de computadores: ao invés de apenas se privilegiar o aumento da frequência de operação suportada pelo uso de tecnologias que garantem tempos de comutação do transístor cada vez mais reduzidos, a introdução de múltiplas unidades de processamento (cores) num único sistema microelectrónico (chip) tornou-se a nova tendência, mantendo como objectivo principal o contínuo aumento da capacidade de processamento de dados em sistemas de computação. Esta tese propõe novas abordagens para estes algoritmos de computação intensiva, que permitem realizar o processamento paralelo de descodificadores LDPC de forma eficiente baseada em arquitecturas multi-core, e que conduzem à obtenção de taxas de débito de saída elevadas, comparáveis às obtidas em sistemas microelectrónicos dedicados (VLSI). É feita a análise exaustiva dos desafios que se colocam à investigação deste tipo de soluções programáveis, dando-se especial

ênfase à flexibilidade e escalabilidade dos algoritmos propostos, aos níveis da taxa de débito e taxa de erros (BER) alcançados, bem como à eficiência geral das soluções programáveis apresentadas, que alcançam resultados acima de uma ordem de grandeza superiores aos obtidos usando CPUs convencionais. Além do mais, a investigação descrita nesta tese segue uma metodologia que analisa com detalhe a complexidade computacional destes algoritmos de descodificação de modo a propor estratégias de aceleração do processamento que, se adequadamente transpostas para outros domínios das ciências da computação, podem demonstrar que nesta nova era dos sistemas multi-core podemos estar na presença de alternativas viáveis em relação a soluções de hardware dedicado (VLSI) não reprogramáveis, cujo desenvolvimento envolve um consumo significativo de recursos não reutilizáveis.

# Palavras Chave

Computação paralela, Arquitecturas de computadores, LDPC, Códigos correctores de erros, Sistemas multi-core, GPU, CUDA, Caravela, Cell/B.E., HPC, OpenMP, Paralelismo de dados, ILP, HPC, VLSI, ASIC, FPGA.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# List of Acronyms

**AWGN** additive white Gaussian noise

**API** Application Programming Interface

**APP** *a posteriori* probability

**AI** artificial intelligence

**ASIC** Application Specific Integrated Circuits

**BER** Bit Error Rate

**BP** belief propagation

**BPSK** Binary Phase Shift Keying

**CCS** compress column storage

**Cell/B.E.** Cell Broadband Engine

**CPU** Central Processing Unit

**CRS** compress row storage

**CTM** Close to the metal

**CUDA** Compute Unified Device Architecture

**DMA** Direct Memory Access

**DSP** Digital Signal Processor

**DVB-S2** Digital Video Broadcasting – Satellite 2

**FEC** Forward Error Correcting

**FFT** Fast Fourier Transform

**FPGA** Field Programmable Gate Array

**FU** Functional Units

**GLSL** OpenGL Shading Language

**GPGPU** General Purpose Computing on GPUs

**GPU** Graphics Processing Units

**GT** Giga Transfer

**HDL** Hardware Description Language

**HLSL** High Level Shader Language

**IRA** Irregular Repeat Accumulate

**LDPC** Low-Density Parity-Check

**LLR** Log-likelihood ratio

**LS** local storage

**LSPA** Logarithmic Sum-Product Algorithm

**MAN** Metropolitan Area Networks

**MFC** Memory Flow Controller

**MIC** Memory Interface Controller

**MIMD** Multiple Instruction Multiple Data

**MPI** Message Passing Interface

**MSA** Min-Sum Algorithm

**NRE** Non-Recurring Engineering

**NUMA** Non-Uniform Memory Architecture

**PC** Personal Computer

**PCIe** Peripheral Component Interconnect Express

**PS3** PlayStation3

**PPE** PowerPC Processor Element

**QMR** Quick Medical Reference

**RTL** Register Transfer Level

**SDR** Software Defined Radio

**SIMD** Single Instruction Multiple Data

**SIMT** Single Instruction Multiple Thread

**SoC** System-on-Chip

**SNR** Signal-to-Noise Ratio

**SPA** Sum-Product Algorithm

**SPE** Synergistic Processor Element

**SPMD** Single Program Multiple Data

**SWAR** SIMD within-a-register

**SDF** Synchronous Data Flow

**VLDW** Very Long Data Word

**VLSI** Very Large Scale Integration

**VRAM** video RAM

# Mathematical numenclature

## Sets

| | |
|---|---|
| $\mathbb{S}$ | General Set |
| $\mathbb{R}$ | Set of real numbers |
| $\mathbb{N}$ | Set of natural numbers |
| $\mathbb{Z}$ | Set of integer numbers |

## Variables and functions

| | |
|---|---|
| $x$ | Variable |
| $X$ | Constant |
| $\hat{x}$ | Estimated value of variable $x$ |

| | |
|---|---|
| $\text{abs}(x)$ | The absolute value of a number: $\lvert -x \rvert = x$ |
| $\text{sign}(x)$ | Operation that extracts the sign of a number: $\text{sign}(-x) = -$ |
| $\text{xor}\{\cdots\}$ | Logical Xor function between two or more arguments |
| $\log(x)$ | The natural logarithm of $x$: $\log(e^x) = x$ |
| $\log_y(x)$ | The logarithm base $y$ of $x$: $\log_y(x) = \log(y) / \log(x)$ |
| $\min\{\cdots\}$ | Function that returns the smallest value of the arguments; if the argument is a matrix, it returns the smallest of all elements in the matrix |
| $\max\{\cdots\}$ | Function that returns the largest value of the arguments; if the argument is a matrix, it returns the largest of all elements in the matrix |

## Matrix, vectors, and vector functions

| | | |
|---|---|---|
| $\mathbf{x}$ | | Vector, typically a column vector |
| $\mathbf{X}$ | | Matrix |
| $\mathbf{0}$ | / $\mathbf{1}$ | Matrix where all elements are $0/1$ |
| $\mathbf{0}_{[I\times J]}$ | / $\mathbf{1}_{[I\times J]}$ | Matrix with $I$ rows and $J$ columns where all elements are $0/1$ |
| $(\mathbf{x})_i$ | / $x_i$ | Element $i$ of vector $\mathbf{x}$ |
| $(\mathbf{X})_{i,j}$ | / $X_{i,j}$ | Element in row $i$, column $j$ of matrix $\mathbf{X}$ |
| $(\mathbf{X})_{\star,j}$ | | Column vector containing the elements of column $j$ of matrix $\mathbf{X}$ |
| $(\mathbf{X})_{i,\star}$ | | Row vector containing the elements of row $i$ of matrix $\mathbf{X}$ |
| $\mathbf{X}^\mathsf{T}$ | | Transpose of matrix $\mathbf{X}$ |

## Statistical operators

| | | |
|---|---|---|
| $v$,$\mathcal{X}$,$y$,$z$,... | | Random variables; $v \neq v$, $\mathcal{X} \neq x$, $y \neq y$, $z \neq z$, ... |
| $\mathrm{P}(\mathcal{X} = x)$ | / $\mathrm{P}(x)$ | Probability for $\mathcal{X}$ to be equal to $x$; the random variable $\mathcal{X}$ has a finite number of states. |
| $p(\mathcal{X} = x)$ | / $p(x)$ | Probability density function for $\mathcal{X}$ to be equal to $x$; the random variable $\mathcal{X}$ has an infinite number of states. |
| $p\big(\mathcal{X} = x \big\vert y = y\big)$ | / $p(x\vert y)$ | Probability density function for $\mathcal{X}$ to be equal to $x$, given that (conditioned to) $y = y$; the random variable $\mathcal{X}$ has an infinite number of states; however, $y$ may or may not have an infinite number of states. |
| $\mathrm{E}[\mathcal{X}]$ | | Expected value of the random variable $\mathcal{X}$ |
| $\mathrm{VAR}[\mathcal{X}]$ | | Variance of the random variable $\mathcal{X}$ |
| $\mathcal{N}\big(\mu;\sigma^2\big)$ | | Normal distribution with mean $\mu$ and variance $\sigma^2$. |
| $\mathcal{X} \sim \mathcal{N}(\mu,\sigma^2)$ | | $\mathcal{X}$ is distributed according to a normal distribution with mean $\mu$ and variance $\sigma^2$. |

*Há um tempo em que é preciso abandonar as roupas usadas, que já têm a forma do nosso corpo, e esquecer os nossos caminhos, que nos levam sempre aos mesmos lugares. É o tempo da travessia: e, se não ousarmos fazê-la, teremos ficado, para sempre, à margem de nós mesmos.*

*Fernando Pessoa*

# 1

# Introduction

Contents

## 1.1 Motivation

Over the last 15 years we have seen Low-Density Parity-Check (LDPC) codes assuming greater and greater importance in the channel coding arena, namely because they have error correction capability to achieve efficient coding close to the Shannon limit. They were invented by Robert Gallager (MIT) in the early sixties[46] and never been fully exploited due to overwhelming computational requirements by that time. Naturally, the fact that nowadays their patent has expired, has shifted the attention of the scientific community and industry away from Turbo codes[9,10] towards LDPC codes. Mainly for this reason and also because advances in microelectronics allowed the development of hardware solutions for real-time LDPC decoding, they have been adopted by modern communication standards. Important examples of these standards are: the WiMAX IEEE 802.16e used in wireless communication systems in Metropolitan Area Networks (MAN)[64]; the Digital Video Broadcasting – Satellite 2 (DVB-S2) used in long distance wireless communications[29]; the WiFi 802.11n the ITU-T G.hn standard for wired home networking technologies; the 10 Giga-Bit Ethernet IEEE 802.3an; or the 3GPP2 Ultra Mobile Broadband (UMB) under the evolution of the 3G mobile system, where the introduction of LDPC codes in 4G systems has recently been proposed, as opposed to the use of Turbo codes in 3G.

Until very recently, solutions for LDPC decoding were exclusively based on dedicated hardware such as Application Specific Integrated Circuits (ASIC), which represent non-flexible and non-reprogrammable approaches[18,70]. Also, the development of ASIC solutions for LDPC decoding consumes significant resources, with high Non-Recurring Engineering (NRE) penalty, and presents long development periods (penalizing time-to-market). An additional restriction in currently developed ASICs for LDPC decoding is the use of fixed-point arithmetic. This introduces limitations due to quantization effects[101] that impose restrictions on coding gains, error floors and Bit Error Rate (BER). Also, the complexity of Very Large Scale Integration (VLSI) parallel LDPC decoders increases significantly for long length codes, as those used, for example, in the DVB-S2 standard. In this case, the routing complexity assumes proportions that create great difficulties in the design of such architectures.

In recent years, we have seen processors scaling up to hundreds of millions of transistors. Memory and power walls have shifted the paradigm of computer architectures into the multi-core era[59]. The integration of multiple cores into a single chip has become the new trend to increase processor performance. Multi-core architectures[12] have evolved from dual or quad-core to many-core systems, supporting multi-threading, a powerful technique to hide memory latency, while at the same time provide larger Single

Instruction Multiple Data (SIMD) units for vector processing. The number of cores per processor assumes significant proportions and is expected to increase even further in the future[16,58]. This new context motivates the investigation of more flexible approaches based on multi-cores to solve challenging computational problems that require intensive processing, and that typically were only achieved in the past with VLSI dedicated accelerators.

Generally worldwide disseminated, many of the actual parallel computing[12] platforms provide low-cost high-performance massive computation, supported by convenient programming languages, interfaces, tools and libraries[71,89]. The advantages of using software- versus hardware-based approaches are essentially related with programmability, reconfigurability, scalability and adjustable data-precision. The advantage is clear if NRE is used as the figure of merit to compare both approaches. However, developing programs for platforms with multiple cores is not trivial. Exploiting the full potential of multi-core based computers many times involves expertise on parallel computation and specific skills which, at the moment, still compel the software developer to deal with low-level architectural/software details.

In the last decade we have seen a vast set of multi-core architectures emerging[12], allowing processing performances unmatched before. The general-purpose multi-core processors replicate a single core in a homogeneous way, typically with a x86 instruction set, and provide shared memory hardware mechanisms. They support multi-threading and share data at a certain level of the memory hierarchy, and can be programmed at a high level by using different software technologies[71]. The popularity of these architectures has made multi-core processing power generally available everywhere.

Mainly due to demands for visualization technology in the games industry, Graphics Processing Units (GPU) have undergone increasing performances over the last decade. Even in a commodity personal computer, we now have at disposal high quality graphics created by real-time computing, mainly due to the high performance GPUs available in personal computers. With many cores driven by a considerable memory bandwidth and a shared memory architecture, recent GPUs are targeted for compute-intensive, multi-threaded, highly-parallel computation, and researchers in high performance computing fields are exploiting GPUs for General Purpose Computing on GPUs (GPGPU)[20,54,99].

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)[24] [60]. It is characterized by an heterogeneous vectorized SIMD multi-core architecture composed by one main PowerPC Processor that communicates efficiently with several Synergistic Processors. It is based on a distributed memory architecture where fast communications between processors are supported by high bandwidth dedicated buses.

Motivated by the evolution of these multi-core systems, programmable parallel approaches for the computationally intensive processing of LDPC decoding are desirable. Multi-cores can allow a low level of NRE and the flexibility they introduce can be used to exploit other levels of efficiency. However, many challenges have to be successfully addressed fostering these approaches, namely the efficiency of data communications and synchronization between cores, the reduction of latency and minimization of congestion problems that may occur in parallel memory accesses, cache coherency and memory hierarchy issues, and scalability of the algorithms, just to name a few. The investigation of efficient parallel algorithms for LDPC decoding has to consider the characteristics of the different architectures, such as homogeneity or heterogeneity of the parallel architecture, and memory models that can support data sharing or be distributed (models that exploit both types also exist). The diversity of architectures imposes different strategies in order to exploit parallelism conveniently. Another important issue consists of adopting data representations suitable for parallel computing engines. All these aspects have significant impact on the achieved performance.

The development of programs for the considered parallel computing architectures allows assessing the performance of software-based solutions for LDPC decoding. The difficulty of developing accurate models for predicting the computational performance of these architectures is associated with the existence of a large variety of parameters which can be manipulated within each architecture. Advances in this area can help understanding how far parallel algorithms are from their theoretical peak performance. This information can be used to tune an algorithm's execution time targeting a certain multi-core architecture. More importantly, it can be used to help perceiving how compilers and parallel programming tools should evolve towards the massive exploitation of the full potential of parallel computing architectures.

## 1.2 Objectives

Given the many challenges associated with the investigation of novel efficient forms of performing parallel LDPC decoding, the main objectives of the research work herein described consist of: *i*) researching new parallel architectures for VLSI-based LDPC decoders under the context of DVB-S2, pursuing the objective of reducing the routing complexity of the design, occupying small die areas and consume low power, while guaranteeing high throughputs; *ii*) investigating novel parallel approaches, algorithms and data structures for ubiquitous low-cost multi-core processors; *iii*) deriving parallel kernels for a set of predefined multi-core architectures based on the proposed algorithms; and *iv*) assessing their efficiency against state-of-the-art hardware-based VLSI solutions, namely

by comparing throughput and BER performance.

Common general-purpose processors have been used in the research of good LDPC codes (searching low error floors, while guaranteeing sparsity) and other types of offline processing necessary to investigate the coding gains of LDPC codes, but they have never been considered for real-time decoding which, depending on the length of the LDPC code, may produce very intensive workloads. Thus, with these processors it is not possible to achieve the required throughput for real-time LDPC decoders, and hardware-dedicated solutions have to be considered.

One of the main objectives of this thesis consisted of investigating more efficient hardware-dedicated solutions (e. g. based on Field Programmable Gate Array (FPGA) or ASIC) for DVB-S2 LDPC decoders. The length of codes adopted in this standard is so large, that actually they represent the most complex challenge regarding the development of efficient solutions for VLSI-based LDPC decoders. Under this context, performing LDPC decoding with a reduced number of light weight node processors, associated with the possibility of reducing the routing complexity of the interconnection network that connects processor nodes and memory blocks, is a target that aims to improve the design in terms of cost and complexity.

While the development of hardware solutions for LDPC decoding was dependent on VLSI technology, it imposed restrictions at the level of data representation, coding gains, BER, etc. The advent of the multi-core era encourages the development of new programmable, flexible and reconfigurable solutions that can represent data with a programmable number of bits. Under this context, the use of massively disseminated architectures with many cores available to solve these computational challenges can be exploited. In this sense, new parallel approaches and algorithms had to be investigated and developed to achieve these goals, which represented another of the main objectives of the thesis.

This led to the development of new forms of representing data to suit parallel computing and stream-based architectures. Since in parallel systems there are multiple cores performing concurrent accesses to data structures in memory, they may have to obey to special constraints such as dimension and alignment in appropriate memory addresses to minimize the problem of traffic congestion. The investigation of these data structures was motivated by the development of appropriate parallel algorithms.

The variety and heterogeneity of programming models and parallel architectures available represented a challenge in this research work. A set of parallel computing machines, selected based on properties such as computational power, popularity, worldwide dissemination and cost, have been considered to solve the intensive computational problem of LDPC decoding. Special challenges have to be addressed, such as: irregular data

representation of binary sparse matrices; intensive computation, which requires iterative decoding but exhibits different levels of parallelism to target distinct architectures, such as homogeneous and heterogeneous systems, with shared and distributed memory. Also, the inclusion of models to predict the performance of the algorithms developed in these types of parallel computing architectures represents a difficult problem which can be tackled by testing separately (and independently) several case limit situations (e.g. memory accesses, or arithmetic instructions). This thesis also assesses the efficiency of the proposed software-based LDPC decoders and compares it with dedicated hardware (e.g. VLSI).

The research carried out, the problems addressed and the solutions developed for the case of LDPC decoding under the context of this thesis, should preferably be portable and serve a variety of other types of computationally intensive problems.

## 1.3 Main contributions

The main contributions of this Ph.D. thesis are:

*i*) optimization of specific hardware components used in VLSI LDPC decoders for DVB-S2; simplification of the interconnection network; optimization of synthesis results for processing units and RAM memory blocks; development of new semi-parallel, scalable and factorizable architectures; prototyping and testing in FPGA and ASIC devices; this work has been communicated in:

[31] Falcão, G., Gomes, M., Gonçalves, J., Faia, P., and Silva, V. (2006). HDL Library of Processing Units for an Automatic LDPC Decoder Design. In *Proceedings of the IEEE Ph.D. Research in Microelectronics and Electronics (PRIME'06)*, pages 349–352.

[49] Gomes, M., Falcão, G., Gonçalves, J., Faia, P., and Silva, V. (2006a). HDL Library of Processing Units for Generic and DVB-S2 LDPC Decoding. In *Proceedings of the International Conference on Signal Processing and Multimedia Applications (SIGMAP'06)*.

[50] Gomes, M., Falcão, G., Silva, V., Ferreira, V., Sengo, A., and Falcão, M. (2007a). Factorizable modulo M parallel architecture for DVB-S2 LDPC decoding. In *Proceedings of the 6th Conference on Telecommunications (CONFTELE'07)*.

[51] Gomes, M., Falcão, G., Silva, V., Ferreira, V., Sengo, A., and Falcão, M. (2007b). Flexible Parallel Architecture for DVB-S2 LDPC Decoders. In *Proceedings of the IEEE Global Telecommunications Conf. (GLOBECOM'07)*, pages 3265–3269.

[52] Gomes, M., Falcão, G., Silva, V., Ferreira, V., Sengo, A., Silva, L., Marques, N., and Falcão, M. (2008). Scalable and Parallel Codec Architectures for the DVB-S2 FEC System. In *Proceedings of the IEEE Asia Pacific Conf. on Circuits and Systems (APCCAS'08)*, pages 1506–1509.

*ii*) development of efficient data structures to represent the message-passing activity between processor nodes that support the iterative decoding of LDPC codes; demonstration that the amount of data can be substantially reduced introducing several advantages, namely in terms of bandwidth; this work has been communicated in:

[32] Falcão, G., Silva, V., Gomes, M., and Sousa, L. (2008). Edge Stream Oriented LDPC Decoding. In *Proceedings of the 16th Euromicro International Conference on Parallel, Distributed and network-based Processing (PDP'08)*, pages 237–244, Toulouse, France.

*iii*) research of new parallel algorithms supported on the proposed compact data structures, for generic families of programmable GPU multi-core computing architectures:

[40] Falcão, G., Yamagiwa, S., Silva, V., and Sousa, L. (2007). Stream-Based LDPC Decoding on GPUs. In *Proceedings of the First Workshop on General Purpose Processing on Graphics Processing Units – GPGPU'07*, pages 1–7.

[41] Falcão, G., Yamagiwa, S., Silva, V., and Sousa, L. (2009d). Parallel LDPC Decoding on GPUs using a Stream-based Computing Approach. *Journal of Computer Science and Technology*, 24(5):913–924.

*iv*) improvement of parallel algorithms for programmable shared memory architectures, that support high-performance dedicated programming interfaces, which resulted in the following communications:

[37] Falcão, G., Sousa, L., and Silva, V. (2008). Massive Parallel LDPC Decoding on GPU. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP'08)*, pages 83–90, Salt Lake City, Utah, USA. ACM.

[34] Falcão, G., Silva, V., and Sousa, L. (2009b). How GPUs can outperform ASICs for fast LDPC decoding. In *Proceedings of the 23rd ACM International Conference on Supercomputing (ICS'09)*, pages 390–399. ACM.

[35] Falcão, G., Silva, V., and Sousa, L. (2010a). *GPU Computing Gems*, chapter Parallel LDPC Decoding. *ed.* Wen-mei Hwu, vol. 1, NVIDIA, Morgan Kaufmann, Elsevier.

*v*) development and investigation of the behavior of alternative parallel algorithms for programmable distributed memory based architectures; development and analysis of models that allowed to assess the efficiency of the resulting solution; work communicated in:

[36] Falcão, G., Silva, V., Sousa, L., and Marinho, J. (2008). High coded data rate and multicodeword WiMAX LDPC decoding on the Cell/BE. *Electronics Letters*, 44(24):1415–1417.

[38] Falcão, G., Sousa, L., and Silva, V. (2009c). Parallel LDPC Decoding on the Cell/B.E. Processor. In *Proceedings of the 4th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC'09)*, volume 5409 of *Lecture Notes in Computer Science*, pages 389–403. Springer.

[33] Falcão, G., Silva, V., Marinho, J., and Sousa, L. (2009a). *WIMAX, New Developments*, chapter LDPC Decoders for the WiMAX (IEEE 802.16e) based on Multicore Architectures. In-Tech.

*vi*) analysis of distinct software-based solutions; comparison of advantages and disadvantages between them; comparison against hardware-dedicated VLSI solutions; analysis of pros and cons of both novel approaches:

[114] Sousa, L., Momcilovic, S., Silva, V., and Falcão, G. (2009). Multi-core Platforms for Signal Processing: Source and Channel Coding. In *Proceedings of the 2009 IEEE International Conference on Multimedia and Expo (ICME'09)*.

[39] Falcão, G., Sousa, L., and Silva, V. (accepted in February 2010b). Massively LDPC Decoding on Multicore Architectures. *IEEE Transactions on Parallel and Distributed Systems*.

## 1.4 Outline

The remaining parts of the present dissertation are organized into five chapters. Chapter 2 addresses belief propagation, and iterative LDPC decoding algorithms are described with special attention given to quasi-cyclic properties of some of the codes used in popular data communication systems. The comprehension of the simplifications that these properties may introduce in the design of efficient decoder processing mechanisms, either being software- or hardware-based, is essential before we proceed to the following

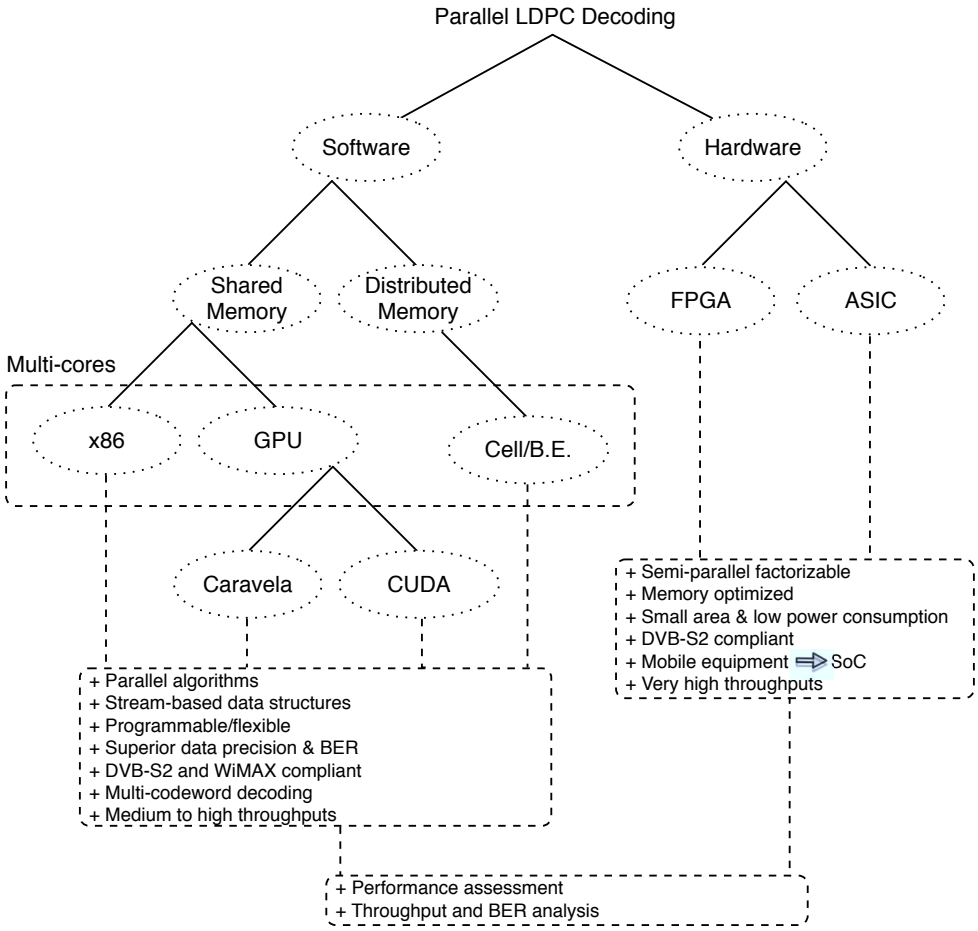


Figure 1.1: Proposed approaches and technologies that support the LDPC decoders presented in this thesis.

chapters. Chapter 3 deeply analyzes in a considerable extent the computational modes and properties of generic parallel programming models and architectures, necessary to perform efficient parallel LDPC decoding. Chapters 4 and 5 describe the most relevant aspects of the developed work. Chapter 4 presents novel scalable VLSI semi-parallel architectures for DVB-S2 that exploit efficient processing units and new configurations of RAM memory blocks that consume significantly less area on a die; and it assesses the different obstacles at the hardware level which had to be overcome to achieve efficient hardware LDPC decoders. The hardware technologies used to develop these solutions are shown on the right side of figure 1.1. On the other hand, in Chapter 5 special attention is given to the compact representation of data structures that suit parallel memory accesses and the corresponding processing on programmable parallel computing architectures; furthermore, it proposes parallel algorithms to accelerate the computation of LDPC decoders to performances that approximate hardware-dedicated solutions; it also proposes computational models to assess the efficiency of these software-based LDPC de-

coders; and compares advantages and disadvantages of the different parallel approaches proposed. The hardware/software that supports these approaches is based on the left subtree in figure 1.1.

The experimental evaluation of the techniques developed in this thesis and the experimental results achieved for the resulting hardware- and software-based solutions are presented, respectively, in chapters 4 and 5. Finally, chapter 6 closes the dissertation by summarizing the main contributions of the thesis and by addressing future research directions.

*"Keep the design as simple as possible, but not simpler."*

*Albert Einstein*

# 2

# Overview of Low-Density Parity-Check codes

## Contents

In channel coding, error correcting codes assume increasing importance in data transmission and storage systems, as the speed of communications, distance of transmission, or amount of data to be transmitted are also increasing. Furthermore, over the last few years, a very important facility has been added to everyday life commodities: mobility. This feature nowadays supports a large variety of products ranging from mobile phones to laptops and, most of the times, it requests the use of error prone data transmission wireless systems. Naturally, data transmissions are not limited to wireless communications and many other systems that support, for example, cable or terrestrial transmissions are also adopting these mechanisms. The capability of detecting (and in some cases even correcting) errors by the recipient of a message, has made this type of systems popular and used in many of today's electronic equipment.

Among the different classes of error correcting codes, we focus this work on the study of binary linear block codes, whose powerful characteristics have been widely addressed, investigated and exploited by the information theory community[93,112]. More specifically, in this thesis we concentrate efforts in the particular case of binary Low-Density Parity-Check (LDPC) codes[106,123,136].

## 2.1 Binary linear block codes

An error correcting code is said to be a block code, if for each message $\mathbf{m}$ of finite length $K$ composed by symbols belonging to the alphabet $\mathbb{X}$, the channel encoder makes corresponding, by means of a transformation $\mathbf{G}$, a codeword $\mathbf{c}$ of length $N$, with $N > K$. The addition of $N - K$ symbols introduces redundancy and is used later to infer the correction of errors on the decoder side. The amount of redundancy introduced usually depends on the channel conditions (e.g. Signal-to-Noise Ratio (SNR)). This amount can be related with the dimension of the code and is given by:

$$rate = K/N, \tag{2.1}$$

where the ratio between the length of the message $K$ and the length of the codeword $N$ defines the *rate* of a code.

The definition of a linear block code, as described in[79], shows that for the complete set of codewords $\mathbb{C}$ that form a linear code, a code is linear if the sum of any number of codewords is still a valid codeword $\in \mathbb{C}$:

$$\forall\ \mathbf{c}_1, \mathbf{c}_2, ..., \mathbf{c}_n\ \in\ \mathbb{C},\ \ \mathbf{c} = \mathbf{c}_1 + \mathbf{c}_2 + ... + \mathbf{c}_n\ \Rightarrow\ \mathbf{c}\ \in\ \mathbb{C}. \tag{2.2}$$

Under the context of this work, we use only binary codes with a corresponding alphabet $\mathbb{X} = \{0, 1\}$. This allows the use of arithmetic modulus 2 operations (such as sum modulus

2 operations represented by $\oplus$). The rule in (2.2) can then be described as:

$$\forall\ \mathbf{c}_1, \mathbf{c}_2, ..., \mathbf{c}_n\ \in\ \mathbb{C},\ \ \mathbf{c} = \mathbf{c}_1 \oplus \mathbf{c}_2 \oplus ... \oplus \mathbf{c}_n\ \Rightarrow\ \mathbf{c}\ \in\ \mathbb{C}. \tag{2.3}$$

Considering the properties of the $\oplus$ operation, we realize that the sum modulus 2 of any codeword $\mathbf{c}$ with itself produces the null vector. This shows another property of linear binary codes: one of the codewords $\mathbf{c} \in \mathbb{C}$ has to be the null vector. The algebraic properties of some linear codes allowed developing efficient encoding and decoding algorithms[94]. This represents an important advantage and many of currently used error correcting codes are sub-classes of linear block codes[79,112].

## 2.2 Code generator matrix G

Considering the message $\mathbf{m}$ of length $K$,

$$\mathbf{m} = [m_0\ m_1\ ...\ m_{K-1}], \tag{2.4}$$

that we wish to transmit over a channel coded as vector $\mathbf{c}$ of length $N$,

$$\mathbf{c} = [c_0\ c_1\ ...\ c_{N-1}]. \tag{2.5}$$

To generate $\mathbf{c}$ from $\mathbf{m}$ we can apply a binary matrix $\mathbf{G}$ of dimension $K \times N$. Then, all codewords can be generated by applying:

$$\mathbf{c} = \mathbf{mG}. \tag{2.6}$$

In order to be able of correctly recovering the initial message on the decoder side of the system, every different message from the set of vector messages $\{\mathbf{m}_0, \mathbf{m}_1, ..., \mathbf{m}_{2^K-1}\}$ should originate a different codeword $\{\mathbf{c}_0, \mathbf{c}_1, ..., \mathbf{c}_{2^K-1}\} \in \mathbb{C}$. The inspection of (2.6) shows that each element of $\mathbb{C} = \{\mathbf{c}_0, \mathbf{c}_1, ..., \mathbf{c}_{2^K-1}\}$ is a linear combination of the rows of $\mathbf{G}$. For this reason, and to guarantee that all codeword vectors $\mathbb{C} = \{\mathbf{c}_0, \mathbf{c}_1, ..., \mathbf{c}_{2^K-1}\}$ are unique, all rows in $\mathbf{G}$ have to be linearly independent (i.e. $\mathbf{G}$ is full rank).

**Systematic codes:** In systematic codes, a different representation of the codeword $\mathbf{c}$ is used that separates message bits (or information bits) from parity-check bits. Then, $\mathbf{c}$ can also be represented by:

$$\mathbf{c} = [c_0\ c_1\ ...\ c_{N-1}] = [b_0\ b_1\ ...\ b_{N-K-1}\ m_0\ m_1\ ...\ m_{K-1}], \tag{2.7}$$

where bits $b_0$ to $b_{N-K-1}$ are parity-check bits obtained from message bits (independent variables) as follows:

$$\mathbf{b} = [b_0\ b_1\ ...\ b_{N-K-1}] = \mathbf{mP}. \tag{2.8}$$

The parity matrix $\mathbf{P}$ has size $K \times (N-K)$ and can be associated to an identity matrix $\mathbf{I_K}$ with dimension $K \times K$, to represent the $\mathbf{G}$ matrix in the systematic form:

$$\mathbf{G} = [\mathbf{P}|\mathbf{I_K}]. \tag{2.9}$$

## 2.3 Parity-check matrix H

If all the $2^K$ codewords of a binary code satisfy $(N-K)$ homogeneous independent linear equations in $c_i$ bits, with $i = 0, 1, ..., N-1$, then the binary code is linear[79,94]. This is equivalent to say that we have a linear system with $N$ binary variables and $(N-K)$ equations. Any linear system of homogeneous equations defines a linear code, and these are called the parity-check equations. They can be represented by the parity-check $\mathbf{H}$ matrix with size $(N-K) \times N$, defined as:

$$\mathbf{H}\mathbf{c}^T = 0_{(N-K)\times 1}, \tag{2.10}$$

where all the $(N-K)$ rows in $\mathbf{H}$ should be linearly independent for the code to be considered a $(N, K)$ code.

A binary linear code defined by a full rank $\mathbf{H}$ matrix can be also represented in the systematic form. The manipulation of (2.6) and (2.10) can be decomposed into:

$$\mathbf{H}\mathbf{c}^T = 0_{(N-K)\times 1} \Leftrightarrow \mathbf{c}\mathbf{H}^T = 0_{1\times(N-K)} \Leftrightarrow \mathbf{m}\mathbf{G}\mathbf{H}^T = 0_{1\times(N-K)} \Rightarrow \mathbf{G}\mathbf{H}^T = 0_{K\times(N-K)}, \tag{2.11}$$

which shows that $\mathbf{H}$ and $\mathbf{G}$ are orthogonal. By applying elementary algebraic operations and according to (2.9) and (2.11), $\mathbf{H}$ can be represented in the systematic form as:

$$\mathbf{H} = [\mathbf{I_{N-K}}|\mathbf{P}^T], \tag{2.12}$$

with $\mathbf{I_{N-K}}$ being an identity matrix of size $(N-K) \times (N-K)$.

## 2.4 Low-Density Parity-Check codes

LDPC are linear block codes with excellent error correcting capabilities, which allow working very close to the Shannon limit capacity[23]. Moreover, as they have been originally invented by Robert Gallager in the early 1960s[46] their patent has expired. In 1996 a very famous paper from Mackay and Neal[84] recaptured the attention from academia/scientific community to their potential and, as a consequence, the industry has recently started to incorporate LDPC codes in modern communication standards. In order to introduce this type of linear codes, a few important properties should be primarily addressed, as they can impact the performance of LDPC codes.

The main properties of an LDPC code are embedded in the parity-check **H** matrix. The **H** matrix is sparse, which means that the number of ones is reduced when compared with the total number of elements. The number of ones per row is defined as the row weight $w_c$ and the number of ones per column as column weight $w_b$. As it will be describe in future sections of the text, the dimensions and weights of a code have great influence in the workload of LDPC decoders. Also, if the weight $w_c$ is the same for all rows and weight $w_b$ is the same for all columns, the code is said to be regular. But if the weight $w_c$ is not constant on every row, or the equivalent happens for columns and $w_b$, then the code is defined as irregular, which introduces additional complexity to control the decoding process of the algorithms.

The dimension of an LDPC code has implications in the coding gain[23,28]. Depending on the application, in order to allow good coding gains, LDPC codes typically are based on $(N, K)$ codes with large dimensions[23] (e.g. $576 \leq N \leq 2304$ bit in the WiMAX standard[64], and $N \leq 16200$ or $N \leq 64800$ bit in the Digital Video Broadcasting – Satellite 2 (DVB-S2) standard[29]). As we increase the length of an LDPC code, we are able to approach the channel limit capacity[23,28].

## 2.5 Codes on graphs

A linear binary block code $(N, K)$ can be described by a binary **H** matrix with dimensions $(N - K) \times N$. Also, it can be elegantly represented by a Tanner graph[118] defined by edges connecting two distinct types of nodes:

- Bit Nodes (BN), also called variable nodes with a BN for each one of the $N$ variables of the linear system of equations, and;
- Check Nodes (CN), also called restriction or test nodes with a CN for each one of the $(N - K)$ homogeneous independent linear system of equations represented by **H**.

Figure 2.1 shows an example for a binary $(8, 4)$ code with $N = 8$ BNs (or variable nodes) and $(N - K) = 4$ CN equations. Each CN connects to all the BNs which have a contribution in that restriction or test equation. For the other type of nodes, each BN corresponding to a $c_i$ bit of the codeword, connects to all the CN equations where bit $c_i$ participates in. The graph edges connect BNs with CNs, but never nodes of the same type, which defines a bipartite graph. Every element in **H** where $\mathbf{H}_{ij} = 1$, represents a connection between $BN_j$ and $CN_i$. In the example of figure 2.1 $BN_0$, $BN_1$ and $BN_2$ are processed by $CN_0$ as indicated in the first row of **H**. From the second until the fourth

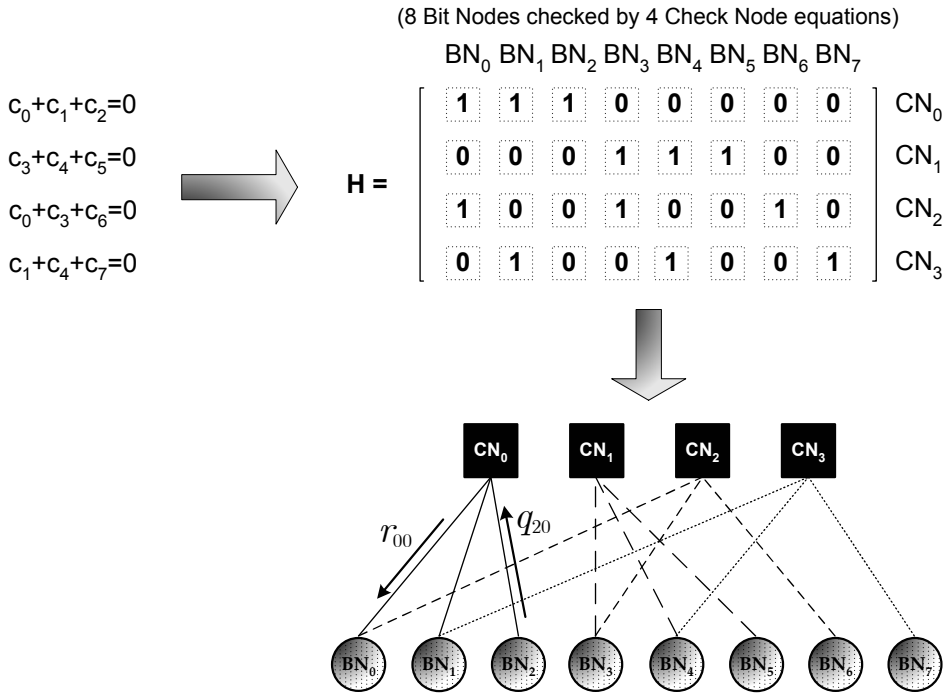


Figure 2.1: A $(8,4)$ linear block code example: parity-check equations, the equivalent **H** matrix and corresponding Tanner graph[118] representation.

row, it can be seen that the subsequent BNs are processed by their neighboring CNs.

### 2.5.1 Belief propagation and iterative decoding

Graphical models, and in particular Tanner graphs as the one addressed in figure 2.1, have often been proposed to perform approximate inference calculations[115,117]. They are based on iterative intensive message-passing algorithms also known as belief propagation (BP) which, under certain circumstances, can become computationally prohibitive. BP, also known as the Sum-Product Algorithm (SPA)[1], is an iterative algorithm[122] for the computation of joint probabilities on graphs commonly used in information theory (e.g. channel coding), artificial intelligence (AI) and computer vision (e.g. stereo vision)[19,117]. It has proved to be efficient and is used in numerous applications including LDPC codes[134], Turbo codes[9,10], stereo vision applied to robotics[117], or in Bayesian networks such as the QMR-DT (a decision-theoretic reformulation of the Quick Medical Reference (QMR)) model described in[77]).

[1] In the literature, the terms BP and SPA are commonly undistinguished. Both are used in this text without differentiation.


Figure 2.2: Illustration of channel coding for a communication system.

**LDPC decoding:** In this text we are interested in the BP algorithm applied to LDPC decoding. In particular, we exploit the Min-Sum Algorithm (MSA), one of the most efficient simplification of the SPA, which are very demanding from a computational perspective. The SPA applied to LDPC decoding operates with probabilities[46], exchanging information and updating messages between neighbors over successive iterations.

Considering a 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[23]. In bipartite graphs, particularly in those with large dimensions (e.g. $N > 1000$ bit), the certainty on a bit can be spread over neighboring bits of a codeword allowing, in certain circumstances, in the presence of noise, to recover the correct bits on the decoder side of the system. 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. The BP mentioned before allows to find the maximum *a posteriori* probability (APP) of vertices in a graph[122].

**Soft decoding:** Propagating probabilities across nodes of the Tanner graph, rather than just flipping bits[46] which is considered hard decoding, is defined as soft decoding. This iterative procedure accumulates evidence imposed by parity-check equations that try to infer the true value for each bit of the received word. Considering in figure 2.2 the original unmodulated codeword $\mathbf{c}$ at the encoder's output and the received word $\mathbf{y}$ at the input of the decoder, we seek the codeword $\hat{\mathbf{c}} \in \mathbb{C}$ that maximizes the probability:

$$p(\hat{\mathbf{c}}|\mathbf{y}, \mathbf{H}\hat{\mathbf{c}}^T = 0). \tag{2.13}$$

However, this represents very intensive computation because all $2^K$ codewords have to be tested. An alternative that makes the decoder perform more localized processing consists of testing only bit $\hat{c}_n$ of all codewords and find a codeword that maximizes the probability:

$$p(\hat{c}_n|\mathbf{y}, \text{all checks involving bit } \hat{c}_n \text{ are satisfied}). \tag{2.14}$$

This represents the APP for that single bit, and only the parity-check equations that participate in bit $\hat{c}_n$ are satisfied. It defines one of two kinds of probabilities used in the

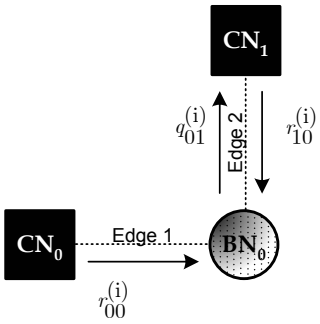


Figure 2.3: Detail of a check node update by the bit node connected to it as defined in the Tanner graph for the example shown in figure 2.5. It represents the calculation of $q_{nm}(x)$ probabilities given by (2.16).

decoding algorithm[94] and can be denoted as:

$$q_n(x) = p(\hat{c}_n = x|\mathbf{y}, \text{all checks involving bit } \hat{c}_n \text{ are satisfied}), \quad x \in \mathbb{X}. \tag{2.15}$$

A variant from (2.15) isolates from this computation the parity-check $m$ that participates in bit $n$:

$$q_{nm}(x) = p(\hat{c}_n = x|\mathbf{y}, \text{all checks, except check } m, \text{ involving bit } \hat{c}_n \text{ are satisfied}), \quad x \in \mathbb{X}, \tag{2.16}$$

and is used to infer decisions on a decoded bit value. The computation of probabilities $q_{nm}(x)$ are illustrated in figure 2.3, which represents the update of a $q_{nm}^{(i)}$ message during iteration $i$.

The second type of probabilities used in the decoding algorithm indicates the probability of parity-check $m$ is satisfied, given only a single bit $\hat{c}_n$ that participates in that parity-check (and all the other observations associated with that parity-check), and it is denoted as:

$$r_{mn}(x) = p(\mathbf{H}_m\hat{\mathbf{c}}^T = 0, \hat{c}_n = x|\mathbf{y}), \quad x \in \mathbb{X}, \tag{2.17}$$

where the notation $\mathbf{H}_m\hat{\mathbf{c}}^T = 0$ represents the linear parity-check constraint $m$ that satisfies the codeword $\hat{\mathbf{c}}$. Probabilities $r_{mn}(x)$ are calculated according to the illustration shown in figure 2.4, which represents the update of a $r_{mn}^{(i)}$ message during iteration $i$. The computation of probabilities $q_{nm}(x)$ and $r_{mn}(x)$, with $x \in \mathbb{X}$, is performed iteratively only for non-null elements of $\mathbf{H}_{mn}$, with $0 < m < N - K - 1$ and $0 < n < N - 1$, i.e. across the Tanner graph edges. The decoder computes probabilities about parity-checks ($r_{mn}(x)$) which are then used to compute information about the bits ($q_{nm}(x)$), on an iterative basis. This propagation of evidence through the tree generated by the Tanner graph, depicted in figure 2.5, allows to infer the correct codeword after all parity-checks are satisfied, or abort after a certain number of iterations is reached.

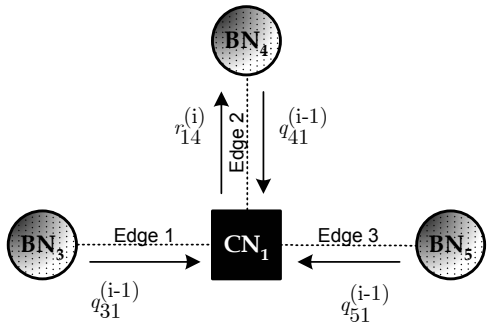


Figure 2.4: Detail of a bit node update by the constraint check node connected to it as defined in the Tanner graph for the example depicted in figure 2.5. It represents the calculation of $r_{mn}(x)$ probabilities given by (2.17).

### 2.5.2 The Sum-Product algorithm

Given a $(N, K)$ binary LDPC code, Binary Phase Shift Keying (BPSK) modulation is assumed, which maps a codeword $\mathbf{c} = (c_0, c_1, c_2, \cdots, c_{n-1})$ into a sequence $\mathbf{x} = (x_0, x_1, x_2, \cdots, x_{n-1})$, according to $x_i = (-1)^{c_i}$. Then, $\mathbf{x}$ is transmitted over an additive white Gaussian noise (AWGN) channel, producing a received sequence $\mathbf{y} = (y_0, y_1, y_2, \cdots, y_{n-1})$ with $y_i = x_i + n_i$, where $n_i$ represents AWGN with zero mean and variance $\sigma^2 = N_0/2$.

The SPA applied to LDPC decoding is illustrated in algorithm 2.1 and is mainly described by two different horizontal and vertical intensive processing blocks defined, respectively, by (2.18) to (2.19) and (2.20) to (2.21)[94]. The equations in (2.18) and (2.19) 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. For each iteration, $r_{mn}^{(i)}$ values are updated according to (2.18) and (2.19), as defined by the Tanner graph[79] illustrated in figure 2.5.

Similarly, the latter pair of equations (2.20) and (2.21) 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, $q_{nm}^{(i)}$ values are updated according to (2.20) and (2.21) and the edges connectivity indicated by the Tanner graph.

Considering the message propagation from nodes $CN_m$ to $BN_n$ and vice-versa, the set of bits that participate in check equation $m$, with bit $n$ excluded, is represented by $\mathcal{N}(m)\backslash n$ and, similarly, the set of check equations in which bit $n$ participates with check $m$ excluded is $\mathcal{M}(n)\backslash m$.

Finally, (2.22) and (2.23) compute the *a posteriori* pseudo-probabilities and in (2.24) 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 ($\mathbf{H}\hat{\mathbf{c}}^T = 0$), or if the maximum number of iterations $I$ is reached.

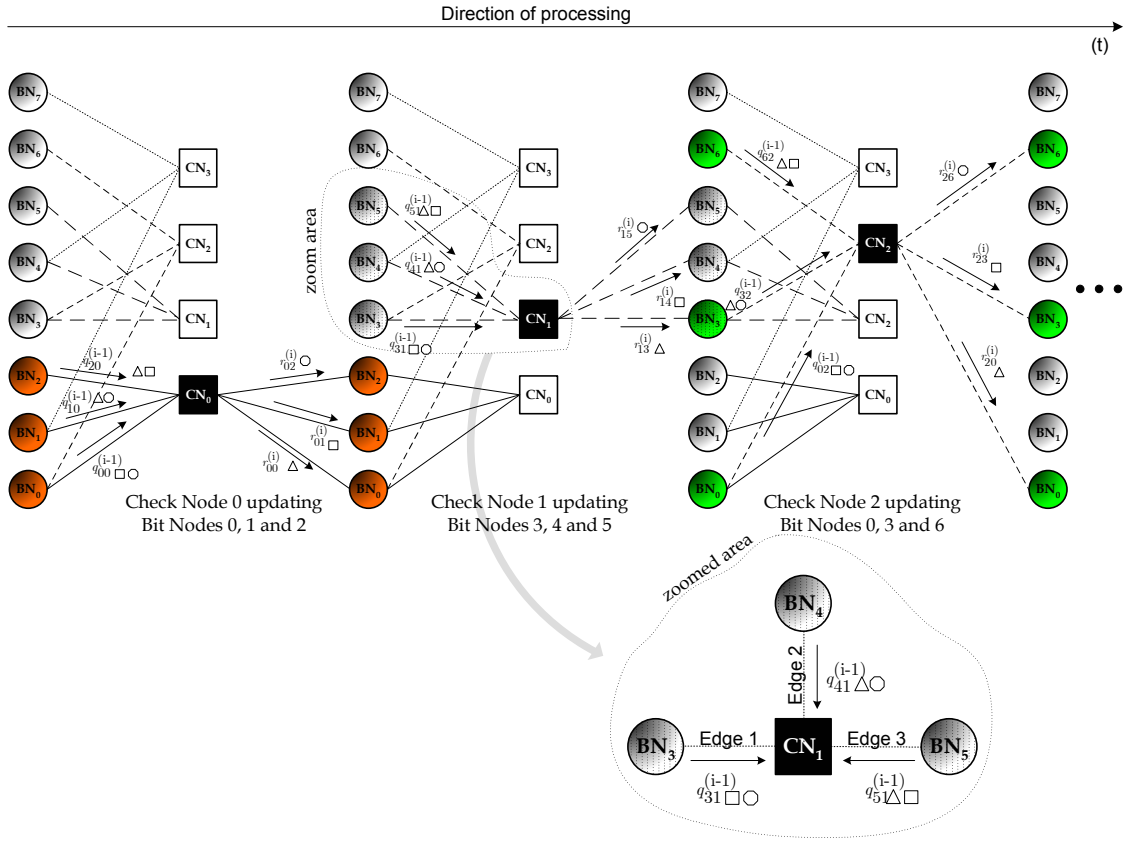


Figure 2.5: Tanner graph[118] expansion showing the iterative decoding procedure for the code shown in figure 2.1.

**Algorithm 2.1** Sum-Product Algorithm – SPA

1: */* Initialization */*

$p_n = \frac{1}{1+e^{\frac{2y_n}{\sigma^2}}}$; $\frac{E_b}{N_0} = \frac{N}{2K\sigma^2}$;

$p_n = p(y_i = 1)$; $q_{mn}^{(0)}(0) = 1 - p_n$; $q_{mn}^{(0)}(1) = p_n$;

2: ***while*** $(\mathbf{H}\hat{\mathbf{c}}^T \neq 0 \wedge \mathrm{i} < \mathrm{I})$ */* c-decoded word; I-Max. no. of iterations. */*
***do***

3: */* For all node pairs $(BN_n, CN_m)$, corresponding to $\mathbf{H_{mn}} = 1$ 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)*

$$r_{mn}^{(i)}(0) = \frac{1}{2} + \frac{1}{2} \prod_{n' \in \mathcal{N}(m) \backslash n} \left(1 - 2q_{n'm}^{(i-1)}(1)\right) \tag{2.18}$$

$$r_{mn}^{(i)}(1) = 1 - r_{mn}^{(i)}(0) \tag{2.19}$$

*/* where $\mathcal{N}(m)\backslash n$ represents BN's connected to $CN_m$ excluding $BN_n$. */*

5: */* Compute message from $BN_n$ to $CN_m$: */*

*(Kernel 2 – Vertical Processing)*

$$q_{nm}^{(i)}(0) = k_{nm}(1 - p_n) \prod_{m' \in \mathcal{M}(n) \backslash m} r_{m'n}^{(i)}(0) \tag{2.20}$$

$$q_{nm}^{(i)}(1) = k_{nm} p_n \prod_{m' \in \mathcal{M}(n) \backslash m} r_{m'n}^{(i)}(1) \tag{2.21}$$

*/* where $k_{nm}$ are chosen to ensure $q_{nm}^{(i)}(0) + q_{nm}^{(i)}(1) = 1$, and $\mathcal{M}(n)\backslash m$ is the set of CN's connected to $BN_n$ excluding $CN_m$. */*

6: */* Compute the a posteriori pseudo-probabilities: */*

$$Q_n^{(i)}(0) = k_n(1 - p_n) \prod_{m \in \mathcal{M}(n)} r_{mn}^{(i)}(0) \tag{2.22}$$

$$Q_n^{(i)}(1) = k_n p_n \prod_{m \in \mathcal{M}(n)} r_{mn}^{(i)}(1) \tag{2.23}$$

*/* where $k_n$ are chosen to guarantee $Q_n^{(i)}(0) + Q_n^{(i)}(1) = 1$. */*

7: */* Perform hard decoding: */*
$\forall n$,

$$\hat{c}_n^{(i)} = \begin{cases} 1 & \Leftarrow \quad Q_n^{(i)}(1) > 0.5 \\ 0 & \Leftarrow \quad Q_n^{(i)}(1) < 0.5 \end{cases} \tag{2.24}$$

8: ***end while***

### 2.5.3 The Min-Sum algorithm

A logarithmic version of the well known SPA [46,83] defined as Logarithmic Sum-Product Algorithm (LSPA) can be derived to achieve a less computationally demanding LDPC

decoding algorithm. Instead of the analysis of two complementary probabilities, this simplification is based on the logarithm of their ratio, defined as the Log-likelihood ratio (LLR). From a computational perspective the complexity of processing decreases as it uses sum operations instead of multiplications, while subtractions replace divisions [63].

**Log-likelihood ratio:** Consider the complementary APP, $p(c_n = 0|\mathbf{y})$ and $p(c_n = 1|\mathbf{y})$ which sum to 1. These two probabilities can be efficiently represented as a unique number by computing their ratio. Its logarithmic representation is given by:

$$LLR = log(\frac{p(c_n = 0|\mathbf{y})}{p(c_n = 1|\mathbf{y})}). \tag{2.25}$$

The MSA [56] consists of a simplification of the LSPA [46,83] and is one of the most efficient algorithms used to perform LDPC decoding [79]. Like the LSPA, the MSA is based on the intensive belief propagation between nodes connected as indicated by the Tanner graph edges (see example in figure 2.5), but only uses comparison and addition operations. Although its workload is lower than the one required by the SPA, it is still quite significant. If the number of nodes is large (in the order of thousands) the MSA can still demand very intensive processing.

The MSA is depicted in algorithm 2.2 [79]. $Lp_n$ designates the *a priori LLR* of $BN_n$, derived from the values received from the channel, and $Lr_{mn}$ is the message that is sent from $CN_m$ to $BN_n$, computed based on all received messages from BNs $\mathcal{N}(m)\backslash n$. Also, $Lq_{nm}$ is the $LLR$ of $BN_n$, which is sent to $CN_m$ and calculated based on all messages received from CNs $\mathcal{M}(n)\backslash m$ and the channel information $Lp_n$. For each node pair ($BN_n$, $CN_m$) we initialize $Lq_{nm}$ with the probabilistic information received from the channel, and then we proceed to the iterative body of the algorithm. Like the SPA, it is mainly described by two horizontal and vertical intensive processing blocks, respectively defined by (2.26) and (2.29). The former calculates the message updating procedure from each $CN_m$ to $BN_n$, considering accesses to $\mathbf{H}$ on a row basis. It indicates the likelihood of $BN_n$ being 0 or 1. The operations $sign(.)$, $min(.)$ (of any two given numbers), and $abs(.)$ described in (2.26), (2.27) and (2.28) represent a simplification of the computation defined in the SPA, and can be performed quite efficiently both in hardware and software. The vertical processing defined in kernel 2 and described in (2.29) computes messages sent from $BN_n$ to $CN_m$, assuming accesses to $\mathbf{H}$ on a column basis.

In (2.30) the *a posteriori* LLR are processed and stored into $LQ_n$, after which we perform the final hard decoding. Differently from the hard decoding decision performed for the SPA in (2.24), here a decision on a bit value can be obtained simply by inspecting the signal of $LQ_n$. The decision on $\hat{\mathbf{c}}$ is described in (2.31) and it can be performed by simple and efficient hardware (and also software) operations, which represents an important

**Algorithm 2.2** Min-Sum Algorithm – MSA

1: */* Initialization */*

$Lp_n = \frac{2y_n}{\sigma^2}$;

$Lq_{nm}^{(0)} = Lp_n$;

2: ***while*** $(\mathbf{H}\hat{\mathbf{c}}^T \neq 0 \wedge \mathrm{i} < \mathrm{I})$ */* c-decoded word; I-Max. no. of iterations. */*
***do***

3: */* For all node pairs $(BN_n, CN_m)$, corresponding to $\mathbf{H_{mn}} = 1$ in the parity check matrix $\mathbf{H}$ of the code* **do**: **/*

4: */* Compute the LLR of messages sent from $CN_m$ to $BN_n$: */*

*(Kernel* 1 *– Horizontal Processing)*

$$Lr_{mn}^{(i)} = \prod_{\mathrm{n}' \in \mathcal{N}(\mathrm{m})\backslash \mathrm{n}} \alpha_{n'm} \min_{\mathrm{n}' \in \mathcal{N}(\mathrm{m})\backslash \mathrm{n}} \beta_{n'm}, \tag{2.26}$$

*with*

$$\alpha_{nm} \stackrel{\Delta}{=} sign\left(Lq_{nm}^{(i-1)}\right), \tag{2.27}$$

*and*

$$\beta_{nm} \stackrel{\Delta}{=} \left|Lq_{nm}^{(i-1)}\right|. \tag{2.28}$$

*/* where $\mathcal{N}(m)\backslash n$ represents BN's connected to $CN_m$ excluding $BN_n$. */*

5: */* Compute the LLR of messages sent from $BN_n$ to $CN_m$: */*

*(Kernel* 2 *– Vertical Processing)*

$$Lq_{nm}^{(i)} = Lp_n + \sum_{m' \in \mathcal{M}(n)\backslash m} Lr_{m'n}^{(i)}. \tag{2.29}$$

*/* where $\mathcal{M}(n)\backslash m$ represents the set of CN's connected to $BN_n$ excluding $CN_m$. */*

*3. Finally, we compute the a posteriori LLRs:*

$$LQ_n{}^{(i)} = Lp_n + \sum_{m' \in \mathcal{M}(n)} Lr_{m'n}^{(i)}. \tag{2.30}$$

6: */* Perform hard decoding: */*

$$\hat{c}_n^{(i)} = -sign(LQ_n^{(i)}) \tag{2.31}$$

7: ***end while***

advantage for any architecture that performs intensive LDPC decoding.

Once again the iterative procedure is stopped if the decoded word $\hat{\mathbf{c}}$ verifies all parity check equations of the code $(\mathbf{H}\hat{\mathbf{c}}^T = 0)$, or a predefined maximum number of iterations is reached, in which case the processing can terminate without obtaining a valid codeword.

## 2.6 Challenging LDPC codes for developing efficient LDPC decoding solutions

Following the recognition of their potential, LDPC codes have been recently adopted by the DVB-S2[29], DVB-C2, DVB-T2, WiMAX (802.16e), Wifi (802.11n), 10Gbit Ethernet (802.3an), and other new standards for communication and storage applications. Such communication standards use their powerful coding gains, obtained at the expense of computational power, to achieve good performances under adverse channel conditions. Some of the LDPC codes adopted in those standards have a periodic nature, which allows exploiting suitable representations of data structures for attenuating the computational requirements. These periodic properties are described next for WiMAX and DVB-S2 codes, in order to allow understanding how architectures can take advantage of them.

### 2.6.1 LDPC codes for WiMAX (IEEE 802.16e)

The WiMAX standard (IEEE 802.16e), which is used in communications for distances typically below the 10 Km range[36], adopts LDPC codes. The Forward Error Correcting (FEC) system of the WiMAX standard is based on a special class of LDPC codes[64] characterized by a sparse binary block parity-check matrix $\mathbf{H}$ which can be partitioned in two block-matrices $\mathbf{H}_1$ and $\mathbf{H}_2$ of the form:

$$\mathbf{H}_{(N-K)\times N} = \left[\begin{array}{c|c} \mathbf{H1}_{(N-K)\times(K)} & \mathbf{H2}_{(N-K)\times(N-K)} \end{array}\right] = \\ = \left[\begin{array}{ccc|ccccccc} \mathbf{P}_{0,0} & \cdots & \mathbf{P}_{0,K/z-1} & \mathbf{Pb}_0 & \mathbf{I} & 0 & \cdots & \cdots & 0 \\ \mathbf{P}_{1,0} & \cdots & \mathbf{P}_{1,K/z-1} & \vdots & \mathbf{I} & \mathbf{I} & & & \vdots \\ \vdots & & \vdots & \vdots & 0 & \mathbf{I} & \ddots & & \vdots \\ \vdots & \ddots & \vdots & \vdots & \ddots & \ddots & \ddots & \mathbf{I} & 0 \\ \mathbf{P}_{N-K/z-2,0} & \cdots & \mathbf{P}_{N-K/z-2,K/z-1} & \vdots & & \ddots & & \mathbf{I} & \mathbf{I} \\ \mathbf{P}_{N-K/z-1,0} & \cdots & \mathbf{P}_{N-K/z-1,K/z-1} & \mathbf{Pb}_{N-K/z-1} & 0 & & \cdots & 0 & \mathbf{I} \end{array}\right], \tag{2.32}$$

where $\mathbf{H}_1$ is sparse and adopts special periodicity constraints in the pseudo-random generation of the matrix[17,64], and $\mathbf{H}_2$ is a sparse lower triangular block matrix with a staircase profile. The periodic nature of these codes defines $\mathbf{H}_1$ based on permutation sub-matrices $\mathbf{P}_{i,j}$, which are: *i*) quasi-random circularly shifted right identity matrices $\mathbf{I}$ (as depicted in (2.32) and in figure 2.6), with dimensions $z \times z$ ranging from $24 \times 24$ to $96 \times 96$ and incremental granularity of 4 (as shown in table 2.1); or *ii*) $z \times z$ null matrices. The periodic nature of such codes allows simplifying the FEC system and storage requirements without code performance loss[64]. Also, the block-matrix $\mathbf{H}_2$ is formed by identity: *i*) $\mathbf{I}$ matrices of dimension $z \times z$; or by *ii*) null matrices of dimension $z \times z$.

The LDPC codes adopted by the WiMAX standard (IEEE 802.16e) support 19 different

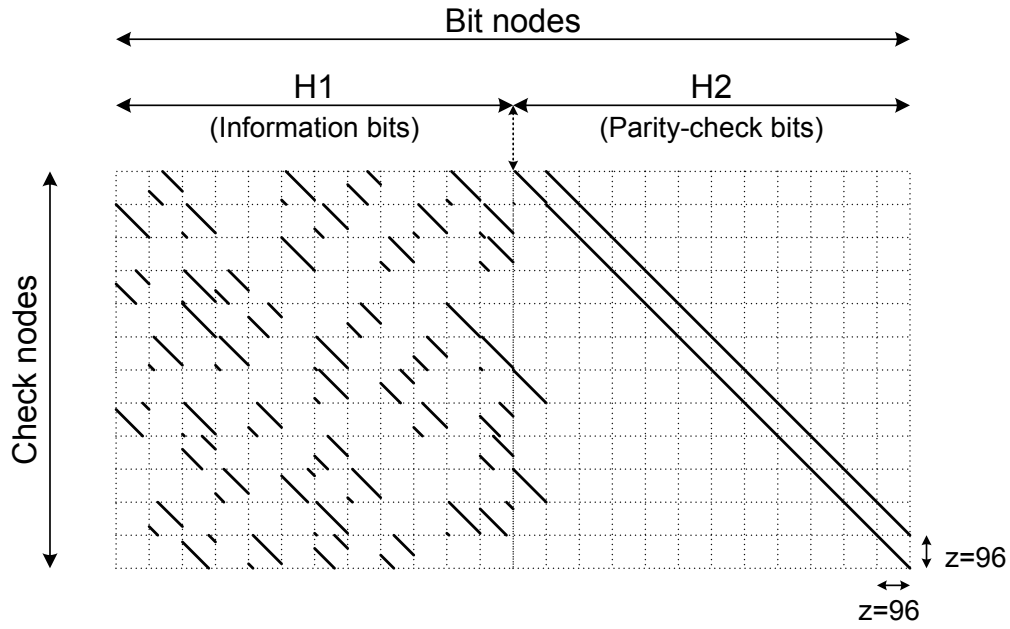


Figure 2.6: Periodicity $z \times z = 96 \times 96$ for an $\mathbf{H}$ matrix with $N = 2304$, $rate = 1/2$ and $\{3, 6\}$ CNs per column.

Table 2.1: Properties of LDPC codes used in the WiMAX IEEE 802.16e standard[64].

| **Code** | **Codeword bits (N)** | **z × z factor** | **Information bits (K)** | | | |
|---|---|---|---|---|---|---|
| | | | (*rate*) | (*rate*) | (*rate*) | (*rate*) |
| | | | 1/2 | 2/3 | 3/4 | 5/6 |
| 1 | 576 | 24 × 24 | 288 | 384 | 432 | 480 |
| 2 | 672 | 28 × 28 | 336 | 448 | 504 | 560 |
| 3 | 768 | 32 × 32 | 384 | 512 | 576 | 640 |
| 4 | 864 | 36 × 36 | 432 | 576 | 648 | 720 |
| 5 | 960 | 40 × 40 | 480 | 640 | 720 | 800 |
| 6 | 1056 | 44 × 44 | 528 | 704 | 792 | 880 |
| 7 | 1152 | 48 × 48 | 576 | 768 | 864 | 960 |
| 8 | 1248 | 52 × 52 | 624 | 832 | 936 | 1040 |
| 9 | 1344 | 56 × 56 | 672 | 896 | 1008 | 1120 |
| 10 | 1440 | 60 × 60 | 720 | 960 | 1080 | 1200 |
| 11 | 1536 | 64 × 64 | 768 | 1024 | 1152 | 1280 |
| 12 | 1632 | 68 × 68 | 816 | 1088 | 1224 | 1360 |
| 13 | 1728 | 72 × 72 | 864 | 1152 | 1296 | 1440 |
| 14 | 1824 | 76 × 76 | 912 | 1216 | 1368 | 1520 |
| 15 | 1920 | 80 × 80 | 960 | 1280 | 1440 | 1600 |
| 16 | 2016 | 84 × 84 | 1008 | 1344 | 1512 | 1680 |
| 17 | 2112 | 88 × 88 | 1056 | 1408 | 1584 | 1760 |
| 18 | 2208 | 92 × 92 | 1104 | 1472 | 1656 | 1840 |
| 19 | 2304 | 96 × 96 | 1152 | 1536 | 1728 | 1920 |

codeword sizes with 4 distinct code *rates* and 6 different class codes (distinct distributions of the number of BNs per column or CNs per row). They are depicted in table 2.1. Class $2/3A$ defines codes having $\{2,3,6\}$ BNs per row and $\{10\}$ CNs per column, while in class $2/3B$ codes have $\{2,3,4\}$ BNs and $\{10,11\}$ CNs. Also, class $3/4A$ has $\{2,3,4\}$ BNs and $\{14,15\}$ CNs, and class $3/4B$ has $\{2,3,6\}$ BNs per row and $\{14,15\}$ CNs per column. To illustrate the periodic nature introduced in the design of $\mathbf{H}$, the matrix structure for $N = 2304$ bit and $rate = 1/2$, is depicted in figure 2.6.

### 2.6.2 LDPC codes for DVB-S2

The FEC system of the recent DVB-S2 standard[29] incorporates a special class of LDPC codes based on Irregular Repeat Accumulate (IRA) codes[30,67]. The parity-check matrix $\mathbf{H}$ is of the form:

$$\mathbf{H}_{(N-K)\times N} = \left[\begin{array}{c|c} \mathbf{A}_{(N-K)\times K} & \mathbf{B}_{(N-K)\times(N-K)} \end{array}\right] = \left[\begin{array}{ccc|cccccc} a_{0,0} & \cdots & a_{0,K-1} & 1 & 0 & \cdots & \cdots & \cdots & 0 \\ a_{1,0} & \cdots & a_{1,K-1} & 1 & 1 & 0 & & & \vdots \\ \vdots & & \vdots & 0 & 1 & 1 & \ddots & & \vdots \\ \vdots & \ddots & \vdots & \vdots & \ddots & \ddots & \ddots & 0 & \vdots \\ a_{N-K-2,0} & \cdots & a_{N-K-2,K-1} & \vdots & & \ddots & 1 & 1 & 0 \\ a_{N-K-1,0} & \cdots & a_{N-K-1,K-1} & 0 & \cdots & \cdots & 0 & 1 & 1 \end{array}\right], \quad (2.33)$$

where $\mathbf{A}$ is sparse and has periodic properties and $\mathbf{B}$ is a staircase lower triangular matrix. The periodicity constraints put on the pseudo-random generation of $\mathbf{A}$ matrices allow a significant reduction on the storage requirements without code performance loss. Assume that the $N$ bits of a codeword are represented in the systematic form, divided in information bits ($IN$) and parity-check bits ($PN$). The construction technique used to generate the $\mathbf{A}$ matrix consists of splitting the $IN$ nodes in disjoint groups of $M = 360$ consecutive 1's. All the $IN$ nodes of a group $l$ should have the same weight $w_l$ and it is only necessary to choose the $CN_m$ nodes that connect to the first $IN$ of the group, in order to specify the $CN_m$ nodes that connect to each one of the remaining $(N-K-1)$ $IN$ nodes. The connection choice for the first elements of group $l$ is pseudo-random but it guarantees that in the resulting LDPC code all the $CN_m$ nodes must connect to the same number of $IN$ nodes. Denoting by $r_1, r_2, ..., r_{w_l}$, the indices of the $CN_m$ nodes that connect to the first $IN$ of group $l$, the indices of the $CN_m$ nodes that connect to $IN_i$, with $0 < i < N-K-1$, of group $l$ can be obtained by:

$$(r_1 + i \times q) mod(N-K), (r_2 + i \times q) mod(N-K), ..., (r_{w_l} + i \times q) mod(N-K), \quad (2.34)$$

with

$$q = (N-K)/M. \quad (2.35)$$

Table 2.2: Properties of LDPC codes[2] used in the DVB-S2 standard [29] for the short frame length.

| *rate* | Codew. bits ($N$) | Inf. bits ($K$) | Col. weight ($w_b$) | Row weight ($w_c$) | # Edges |
|---|---|---|---|---|---|
| 1/4 | 16200 | 4050 | {3, 12} | {4}* | 48600 |
| 1/3 | 16200 | 5400 | {3, 12} | {5} | 54000 |
| 2/5 | 16200 | 6480 | {3, 12} | {6} | 58320 |
| 1/2 | 16200 | 8100 | {3, 8} | {6}* | 48600 |
| 3/5 | 16200 | 9720 | {3, 12} | {11} | 71280 |
| 2/3 | 16200 | 10800 | {3, 13} | {10} | 54000 |
| 3/4 | 16200 | 12150 | {3, 12} | {12}* | 48600 |
| 4/5 | 16200 | 12960 | {3} | {14}* | 45360 |
| 5/6 | 16200 | 13500 | {3, 13} | {19}* | 51300 |
| 8/9 | 16200 | 14400 | {3, 4} | {27} | 48600 |

The factor $M = 360$ is constant for all codes used in the DVB-S2 standard. For each code, **A** has groups of $IN$ nodes with constant weights $w_c > 3$, and also groups with weights $w_c = 3$. Matrix **B** has a lower triangle staircase profile as shown in (2.33). The LDPC codes adopted by the DVB-S2 standard support two different frame lengths, one for short frames ($N = 16200$ bit) and the other for normal frames ($N = 64800$ bit). The short frame mode supports 10 distinct code rates as depicted in table 2.2, while the latter supports 11 rates as shown in table 2.3. The column and row weights are also depicted in tables 2.2 and 2.3 for all rates in the standard. For short frame lengths, all codes have constant weight $w_c$, with the exception of five of them[2] as indicated in table 2.2, while BNs have two types of weights $w_b$, except for code with $rate = 4/5$ which has constant weight $w_b = 3$. For normal frame length codes, all CNs have a constant weight $w_c$, while BNs have two types of weights $w_b$ for each rate, as indicated in table 2.3.

Both tables show the number of edges for each code adopted in the DVB-S2 standard. In each edge circulates a message that is used to update the corresponding node it connects to. A closer inspection, for example, of code with $rate = 3/5$ for normal frames, shows that the total number of edges of the Tanner graph is 285120. If we consider that communications occur in both directions (from CNs to BNs, and then from BNs to CNs), $570240 > 512K$ messages are exchanged per iteration, which imposes huge computational demands (at several levels) for the development of LDPC decoders. In order to perform efficiently these data communications and associated computations, the analysis of LDPC decoding algorithms is performed in the next chapter, paving the way for the analysis of programming models and architectures that suit such types of extremely

[2]The five short frame codes marked with the symbol * don't have a constant weight per row $w_c$, since they have been shortened has defined in the standard [29]. Consequently, they have rates which are an approximation to those mentioned in this table, but not exactly the same.

Table 2.3: Properties of LDPC codes used in the DVB-S2 standard [29] for the normal frame length.

| *rate* | Codew. bits ($N$) | Inf. bits ($K$) | Col. weight ($w_b$) | Row weight ($w_c$) | # Edges |
|---|---|---|---|---|---|
| 1/4 | 64800 | 16200 | {3, 12} | {4} | 194400 |
| 1/3 | 64800 | 21600 | {3, 12} | {5} | 216000 |
| 2/5 | 64800 | 25920 | {3, 12} | {6} | 233280 |
| 1/2 | 64800 | 32400 | {3, 8} | {7} | 226800 |
| 3/5 | 64800 | 38880 | {3, 12} | {11} | 285120 |
| 2/3 | 64800 | 43200 | {3, 13} | {10} | 216000 |
| 3/4 | 64800 | 48600 | {3, 12} | {14} | 226800 |
| 4/5 | 64800 | 51840 | {3, 11} | {18} | 233280 |
| 5/6 | 64800 | 54000 | {3, 13} | {22} | 237600 |
| 8/9 | 64800 | 57600 | {3, 4} | {27} | 194400 |
| 9/10 | 64800 | 58320 | {3, 4} | {30} | 194400 |

demanding processing systems in the following chapters.

## 2.7 Summary

The properties of a particular family of linear block codes – LDPC codes –, based on intensive belief propagation algorithms and their simplified variants, are introduced. Although LDPC codes have been discovered almost half a century ago, due to huge computational requirements only recently they have recaptured the attention of the scientific community.

In this chapter it is given a brief introduction to belief propagation. The iterative decoding representation based on bipartite or Tanner graphs of two main classes of algorithms used in LDPC decoding is described, with particular focus given on the type of processing required, and on the intensive nature of communications. Also, an overview of the periodic nature of LDPC codes adopted in two important and recent standards is introduced. In this overview, the mechanisms adopted for the generation of matrices used in LDPC decoding are equally described.

In the next chapter, the complexity of these algorithms and corresponding computational requirements will be addressed, namely concerning the adoption of computational models and parallelization approaches to accelerate this type of intensive processing.

*"Computers in the future may weigh no more than 1.5 tons."*

*Popular Mechanics, 1949*

# 3

# Computational analysis of LDPC decoding

## Contents

In this chapter we analyze the computational properties of Low-Density Parity-Check (LDPC) code decoders introduced in chapter 2, and propose strategies to achieve efficient processing. To obtain high throughputs as required by communication standards, LDPC decoders can demand very intensive computation. A huge number of arithmetic operations and memory accesses per second is necessary to support high throughputs related with multimedia/storage system requirements. These applications typically impose severe constraints in the processing of video, audio, and data communications to provide services in real-time within a targeted quality. Such demanding for real-time processing imposes the analysis of computational properties of LDPC decoding and the exploitation of parallelism. We study the dependencies that may prevent such achievement, analyze task-parallelism, data-parallelism and also the memory access requirements for performing LDPC decoding in parallel. Finally, we address the importance of selecting a proper message-passing schedule mechanism within the family of LDPC decoding algorithms, when targeting parallel computing architectures. A discussion about the scalability of LDPC decoders in parallel architectures concludes the chapter.

## 3.1 Computational properties of LDPC decoding algorithms

To reduce the high number of arithmetic operations and memory accesses required by the Sum-Product Algorithm (SPA) and Min-Sum Algorithm (MSA) we exploit a simplification based in the Forward-and-Backward optimization[63]. We also use the MSA with $\lambda$-min$=2$[56], which is a simplification of the SPA in order to reduce even further the number of operations. Like the SPA, the MSA is based on the belief propagation between connected nodes, as defined by the Tanner graph edges (and shown in the example of figure 2.5), but only uses comparison and addition operations. The workload produced is still quite intensive, but lower than the one demanded by the SPA.

### 3.1.1 Analysis of memory accesses and computational complexity

The SPA applied to LDPC decoding is illustrated in algorithm 2.1 and is mainly described by two different horizontal and vertical intensive processing blocks defined, respectively, by the pair of equations (2.18), (2.19) and (2.20), (2.21). In (2.18) and (2.19) the message updating from $CN_m$ to $BN_n$ is calculated, considering accesses to $\mathbf{H}$ in a row-major basis – the horizontal processing – obtaining the $r_{mn}^{(i)}$ values that indicate the probability of $BN_n$ being 0 or 1. Similarly, the latter pair of equations (2.20) and (2.21) 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, $q_{nm}^{(i)}$ values are updated according to (2.20) and (2.21), and the edges connectivity indicated by the Tanner graph.

Table 3.1: Number of arithmetic and memory access operations per iteration for the SPA.

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

**Decoding complexity of the SPA:** Assuming a $\mathbf{H}$ matrix with $M = (N - K)$ CNs (rows) and $N$ BNs (columns), a mean row weight $w_c$ and a mean column weight $w_b$, with $w_c$, $w_b \geq 2$, table 3.1 presents 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. The table also presents the number of memory accesses required and it can be seen that for both cases the complexity expressed as a function of $w_c$ and $w_b$ is quadratic. Depending on the size of the codeword and channel conditions (Signal-to-Noise Ratio (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.

The decoding complexity can, however, be reduced by using the efficient Forward-and-Backward algorithm[63] that minimizes the number of memory accesses and operations necessary to update (2.18) to (2.21) in algorithm 2.1. In table 3.2, 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 $w_b$). This represents a reduction in the number of arithmetic operations, and an even more significant decrease in memory access operations, which contributes to increase the arithmetic intensity, here defined as the ratio of arithmetic operations per memory access. This property (a higher arithmetic intensity) suits multi-core architectures conveniently, which often have their performance limited by memory accesses[82]. For a regular code with $rate = \frac{1}{2}$, where $N = 2M$, $w_c = 2w_b$ and $w_b \geq 2$, the arithmetic intensity $\alpha_{SPA}$ for the SPA is:

$$\alpha_{SPA} = \frac{(3w_c^2 - w_c)M + (2w_b^2 + 4w_b + 1)N}{(w_c^2 + w_c)M + (2w_b^2 + 2)N} \approx 2, \tag{3.1}$$

Table 3.2: Number of arithmetic and memory access operations per iteration for the optimized Forward-and-Backward algorithm[63].

| **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 | $(3w_b + 1)N$ |

while the arithmetic intensity $\alpha_{FBA}$ for the optimized Forward-and-Backward algorithm can be obtained by:

$$\alpha_{FBA} = \frac{(9w_c - 8)M + (9w_b - 4)N}{3w_cM + (3w_b + 1)N} \approx 3. \tag{3.2}$$

**Decoding complexity of the MSA:** Table 3.3 presents the number of compare, abs (absolute value calculations), min (the smallest of two given numbers), and xor (for signal calculation) operations per iteration of the MSA algorithm. It also shows the number of memory accesses per iteration necessary to update (2.26) to (2.29). They exhibit linear complexity as a function of $w_c$ and $w_b$. Here, the arithmetic intensity can be described by (3.3) for a regular code with $rate = \frac{1}{2}$, where $N = 2M$, $w_c = 2w_b$ and $w_b \geq 2$.

$$\alpha_{Min-Sum} = \frac{(4w_c - 1)M + 2w_bN}{2w_cM + (2w_b + 1)N} \approx 1.5. \tag{3.3}$$

The value $\alpha_{Min-Sum}$ shows that when applying the $\lambda$-min algorithm, memory accesses should be optimized in order to maximize performance. Moreover, the type of operations is simpler than in the case of the SPA due to the optimized Forward-and-Backward algorithm[63,134,136].

Figure 3.1 shows the number of arithmetic and logic operations (ALO) and memory access operations (MAO) per iteration – the total is indicated in the figure as mega operations (MOP) – performed by the previously mentioned three LDPC decoding algorithms, for two different situations: a) $w_c = 6$ and $w_b = 3$; and b) $w_c = 14$ and $w_b = 6$. LDPC codes A to E have (N, K) equal to $(504, 252)$, $(1024, 512)$, $(4896, 2448)$, $(8000, 4000)$ and $(20000, 10000)$, respectively. It can be seen that for a code of medium to large dimensions

Table 3.3: Number of arithmetic and memory access operations per iteration for the optimized MSA.

| **Min-Sum - Horizontal Processing** | |
|---|---|
| | Number of operations |
| Comparisons | $w_c M$ |
| abs | $w_c M$ |
| min | $(w_c - 1)M$ |
| xor | $w_c M$ |
| Memory Accesses | $2w_c M$ |
| **Min-Sum - Vertical Processing** | |
| | Number of operations |
| Additions/Subtractions | $2w_b N$ |
| Memory Accesses | $(2w_b + 1)N$ |

such as code D, a total of 3 MOP is required for the SPA – arithmetic and logic operations (ALO) and memory accesses (MAO) – per iteration, using $w_c$ and $w_b$ as depicted in 3.1 b). Considering code E and using a typical value of 10 iterations for the same algorithm, we nearly achieve 80 MOP. The workload of intensive LDPC decoding algorithms grows linearly with the number of iterations. The number of iterations typically ranges from 5 to

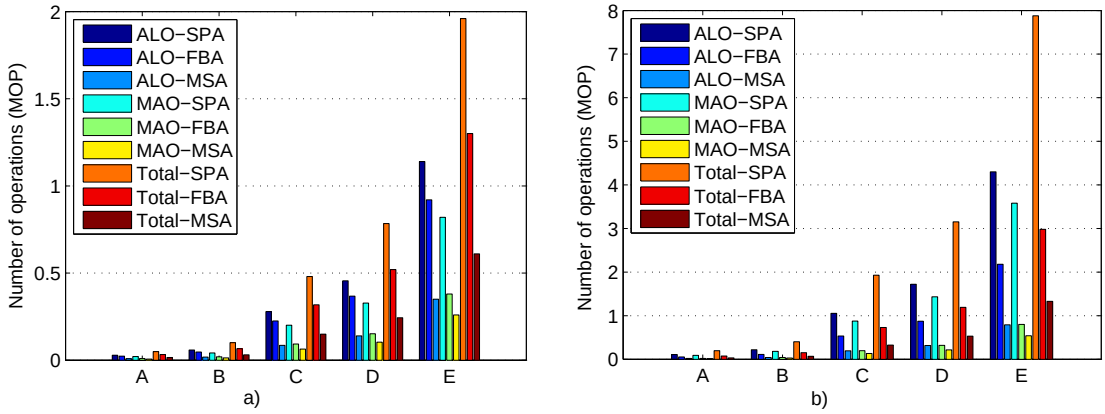


Figure 3.1: Mega operations (MOP) per iteration for 5 distinct **H** matrices (A to E), with a) $w_c = 6$ and $w_b = 3$; and b) $w_c = 14$ and $w_b = 6$.

20 (a pessimistic assumption)[81,88]. The number of operations necessary to decode 1000 codewords is depicted in figure 3.2 for the 3 algorithms mentioned before, each running 15 iterations. In this figure, the total number of operations is indicated as giga operations (GOP).

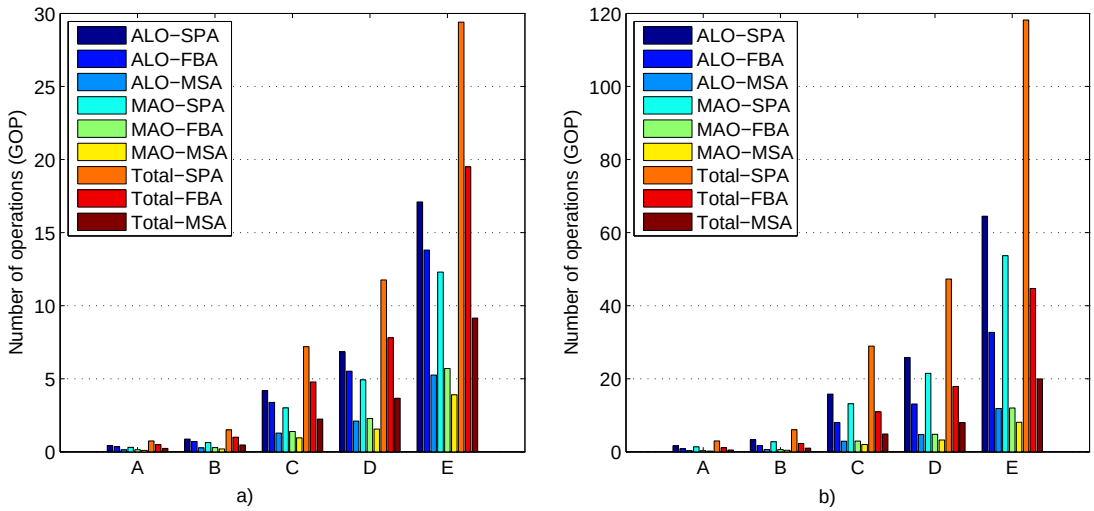


Figure 3.2: Giga operations (GOP) for the parallel decoding of 1000 codewords running 15 iterations each, for the same **H** matrices referred in figure 3.1, with a) $w_c = 6$ and $w_b = 3$; and b) $w_c = 14$ and $w_b = 6$.

### 3.1.2 Analysis of data precision in LDPC decoding

The average number of iterations naturally influences the decoding time. For a low SNR the number of iterations necessary for a codeword to converge is greater than for higher SNRs. Figure 3.3 shows that the average number of iterations in the LDPC decoding process depends on the SNR (represented by $E_b/N_0$) of data received at the input of the decoder. As shown in the figure, if we decode for example $10^8$ bits using the WiMAX LDPC code (576, 288)[64], for a maximum number of iterations equal to 50 and an SNR of 1 dB, we can observe that the average number of iterations nearly reaches its maximum (50 in this case). On the other hand, if the SNR rises to 4 dB, the average number of iterations drops below 4. Depending on the adopted code, LDPC decoders can achieve very good performances (low Bit Error Rate (BER)) even for low SNRs, at the expense of computational power.

Solutions that dedicate more bits to represent data (messages) can compare favorably in terms of the number of iterations needed for the algorithm to converge, when we look at state-of-the-art Application Specific Integrated Circuits (ASIC) LDPC decoders in the literature[17,81,108,111] that use low precision architectures to represent data (e.g. 5 to 6-bits). An additional advantage is observed in the BER curves for typical ASIC solutions. A minimal data precision representation is fundamental in ASIC solutions in order to achieve acceptable die area and power consumption. Figure 3.4 shows the BER curves obtained for two different codes used in the WiMAX standard[64], namely (576,288) and (1248,624) with either 6 or 8-bit precision. A MSA implementation, based on an 8-bit data

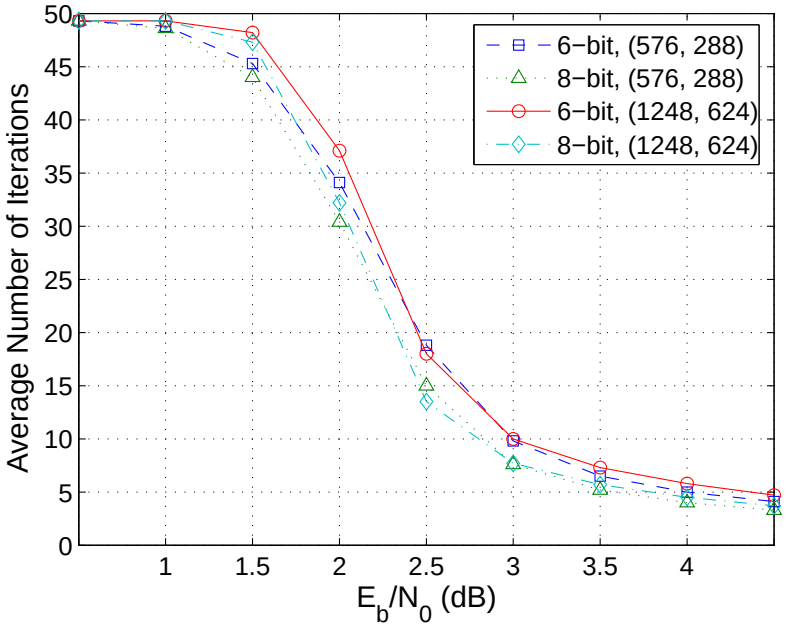


Figure 3.3: Average number of iterations per SNR, comparing 8-bit and 6-bit data precision representations for WiMAX LDPC codes (576,288) and (1248,624)[64] running a maximum of 50 iterations.

precision solution, achieves improved BER compared with ASIC solutions[17,81,108,111].

The programmable solutions proposed in this thesis and described later in future sections of the text, use either 32-bit floating-point single precision, or 8-bit integer precision to represent data, which helps to improve the BER performance.

### 3.1.3 Analysis of data-dependencies and data-locality in LDPC decoding

The $w_c M$ messages involved per iteration in the horizontal step and the $w_b N$ messages required per iteration for the vertical step, are depicted in the two columns of table 3.4, for the Tanner graph example shown in figure 2.5. The $\mathcal{F}_h(.)$ and $\mathcal{F}_v(.)$ functions represent, respectively, the horizontal and vertical kernels described in algorithms 2.1 (for the SPA) and 2.2 (for the MSA), while $m^i_{Z_0 \to W_0}$ denotes a message $m$ passed from node $Z_0$ to node $W_0$ during iteration $i$. In the vertical kernel, $p_0$ represents the channel information associated to $BN_0$ and $p_7$ the channel information for $BN_7$.

**Data-dependencies:** The operations referred before can be parallelized by adopting a suitable message-passing schedule that supports the updating corresponding to different vertices or/and different messages, simultaneously, in distinct parts of the Tanner graph. The main data-dependency constraints arise from the fact that all messages used in the calculation of a new message in any iteration were obtained during the previous iteration.

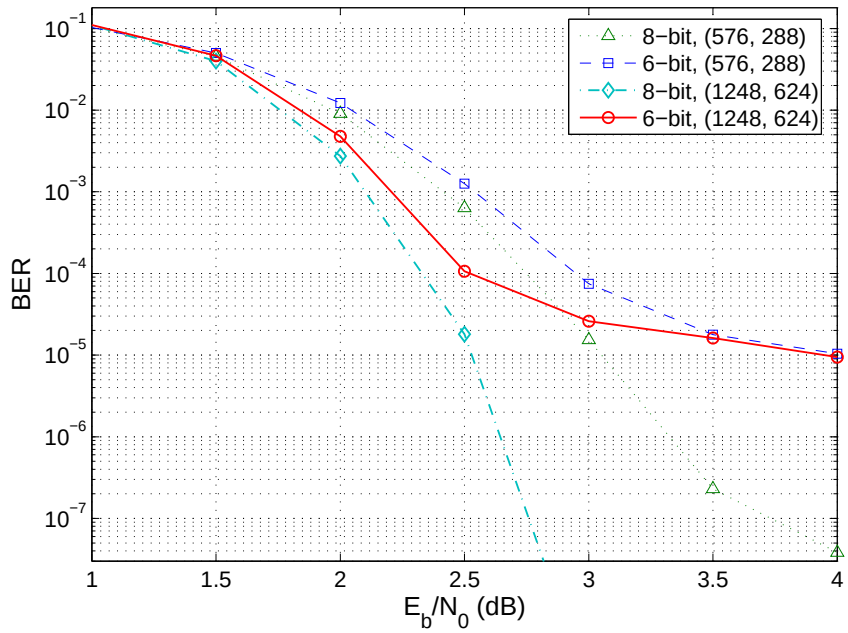


Figure 3.4: BER curves comparing 8-bit and 6-bit data precision for WiMAX LDPC codes (576,288) and (1248,624)[64].

Analyzing the example illustrated in figure 2.5 for the horizontal processing, the update of the 3 BNs associated with the first CN equation (first row of **H** shown in figure 2.1) can be performed in parallel with the update of other CN equations in the same iteration, without any kind of conflict between nodes. The remaining messages on the Tanner graph show that the same principle applies to the other CN equations of the example.

Table 3.4: Message computations for one iteration of the example shown in figures 2.1 and 2.5.

| Horizontal kernel | Vertical kernel |
|---|---|
| $m^i_{CN_0\to BN_0} = \mathcal{F}_h(m^{i-1}_{BN_1\to CN_0}, m^{i-1}_{BN_2\to CN_0})$ | $m^i_{BN_0\to CN_0} = \mathcal{F}_v(p_0, m^{i-1}_{CN_2\to BN_0})$ |
| $m^i_{CN_0\to BN_1} = \mathcal{F}_h(m^{i-1}_{BN_0\to CN_0}, m^{i-1}_{BN_2\to CN_0})$ | $m^i_{BN_0\to CN_2} = \mathcal{F}_v(p_0, m^{i-1}_{CN_0\to BN_0})$ |
| $m^i_{CN_0\to BN_2} = \mathcal{F}_h(m^{i-1}_{BN_0\to CN_0}, m^{i-1}_{BN_1\to CN_0})$ | $m^i_{BN_1\to CN_0} = \mathcal{F}_v(p_1, m^{i-1}_{CN_3\to BN_1})$ |
| $m^i_{CN_1\to BN_3} = \mathcal{F}_h(m^{i-1}_{BN_4\to CN_1}, m^{i-1}_{BN_5\to CN_1})$ | $m^i_{BN_1\to CN_3} = \mathcal{F}_v(p_1, m^{i-1}_{CN_0\to BN_1})$ |
| $m^i_{CN_1\to BN_4} = \mathcal{F}_h(m^{i-1}_{BN_3\to CN_1}, m^{i-1}_{BN_5\to CN_1})$ | $m^i_{BN_2\to CN_0} = \mathcal{F}_v(p_2)$ |
| $m^i_{CN_1\to BN_5} = \mathcal{F}_h(m^{i-1}_{BN_3\to CN_1}, m^{i-1}_{BN_4\to CN_1})$ | $m^i_{BN_3\to CN_1} = \mathcal{F}_v(p_3, m^{i-1}_{CN_2\to BN_3})$ |
| $m^i_{CN_2\to BN_0} = \mathcal{F}_h(m^{i-1}_{BN_3\to CN_2}, m^{i-1}_{BN_6\to CN_2})$ | $m^i_{BN_3\to CN_2} = \mathcal{F}_v(p_3, m^{i-1}_{CN_1\to BN_3})$ |
| $m^i_{CN_2\to BN_3} = \mathcal{F}_h(m^{i-1}_{BN_0\to CN_2}, m^{i-1}_{BN_6\to CN_2})$ | $m^i_{BN_4\to CN_1} = \mathcal{F}_v(p_4, m^{i-1}_{CN_3\to BN_4})$ |
| $m^i_{CN_2\to BN_6} = \mathcal{F}_h(m^{i-1}_{BN_0\to CN_2}, m^{i-1}_{BN_3\to CN_2})$ | $m^i_{BN_4\to CN_3} = \mathcal{F}_v(p_4, m^{i-1}_{CN_1\to BN_4})$ |
| $m^i_{CN_3\to BN_1} = \mathcal{F}_h(m^{i-1}_{BN_4\to CN_3}, m^{i-1}_{BN_7\to CN_3})$ | $m^i_{BN_5\to CN_1} = \mathcal{F}_v(p_5)$ |
| $m^i_{CN_3\to BN_4} = \mathcal{F}_h(m^{i-1}_{BN_1\to CN_3}, m^{i-1}_{BN_7\to CN_3})$ | $m^i_{BN_6\to CN_2} = \mathcal{F}_v(p_6)$ |
| $m^i_{CN_3\to BN_7} = \mathcal{F}_h(m^{i-1}_{BN_1\to CN_3}, m^{i-1}_{BN_4\to CN_3})$ | $m^i_{BN_7\to CN_3} = \mathcal{F}_v(p_7)$ |

A similar conclusion can be drawn when analyzing the vertical processing. In spite

of the irregular memory access pattern, in this case too it is possible to parallelize the processing of the $w_b N$ new messages, as it is the case of $m^i_{BN_0 \to CN_0}$ and $m^i_{BN_0 \to CN_2}$, that exemplify the update of the two messages associated to $BN_0$ (first column of **H** shown in figure 2.1) and that can be performed in parallel with other messages.

**Exploiting spatial locality:** To permit the simultaneous update of messages, code and data structures that allow exposing parallel properties of the algorithm for data read/write operations should be considered. To efficiently exploit spatial locality it is also important to consider the characteristics of the computational systems architecture. If this is not the case, the impact of memory accesses may rise and, depending on the level of disarrangement, have a significant weight in the processing efficiency. For example, the memory hierarchy which typically has larger but slower memories as we move away from the processor, may lead to poor speedups if the locality is not properly exploited (e.g. caches).

In the context of LDPC decoding we group blocks of data associated to CNs and BNs, with sizes that depend on the length of the code and the architecture being used. By varying the granularity of data blocks, different results can be obtained [32,39], and a trade-off between time spent performing communications and data size must be pursued in order to achieve optimal results. Another aspect equally important for exploiting locality consists of placing each one of these groups of data located together in memory. Runtime data realignment techniques were developed [34] to tackle the problem that sometimes arises from such procedure, due to the fact that data cannot be aligned at compile time. They will be described in future sections of the text, for both horizontal and vertical processing steps.

**Exploiting temporal locality:** Another aspect that assumes great importance for achieving efficient LDPC decoding is the proper exploitation of temporal locality. Temporal locality exists when a certain computation accesses the same group of data several times in a short period of time. To exploit temporal locality in parallel systems, the computation that accesses the same data should be assigned to the same processing unit. The main goal here is to organize the processing so that those accesses to the same group of data can be scheduled to execute close in time one to another. To achieve this, the programmer needs to organize the algorithm so that working sets are kept small enough to map well on the sizes of the different levels of memory hierarchy, but not too small to cause inefficiency.

The kernels developed for LDPC decoding in this thesis are structured to process groups of BNs associated to the same CN for the horizontal processing. In the vertical

processing, groups of CNs associated to the same BN can also be processed by the same kernel. The advantage obtained here is clear: inside each kernel, common accesses to data are repeated in very short periods of time for these groups of BNs and CNs, which serves the purpose of special memory structures dedicated to exploit this property (e.g. cache memory).

Either exploiting spatial or temporal locality, the message-passing schedule mechanism assumes high importance regarding to data-dependencies, as it will be mentioned later in section 3.3.2.

## 3.2 Parallel processing

The influence of parallel computing approaches and models in the performance of a parallel application can be analyzed from different perspectives.

### 3.2.1 Parallel computational approaches

Although several sub-categories of parallel computational approaches can be considered, they mainly belong to one of two classes: task parallelism and data parallelism. More detailed sub-classes of parallel computational approaches and models can be identified to target actual parallel systems.

**Task parallelism:** A task defines the smallest unit of concurrency that a program can exploit. In parallel architectures, the granularity adopted when determining the size of tasks (i.e. the number of instructions) should preferably use fine-grained or smaller tasks, in order to stimulate workload balancing. On the other hand, it shouldn't be too small because in that case interprocessor communications and management overheads would relatively rise, introducing penalties that may have a negative impact on the performance.

Some parallel computing machines have a Multiple Instruction Multiple Data (MIMD) stream architecture, where at any given instant, multiple processors can be executing, independently and asynchronously, different tasks or instructions on different blocks of data.

**Data parallelism:** A simple way of exploiting algorithmic intrinsic parallelism consists in using Single Instruction Multiple Data (SIMD), where programming resembles more the serial programming style but the same instruction is applied to a collection of data, all in parallel and at the same time. This model is usually supported on the SIMD stream type of architectures, where several data elements can be kept and operated in parallel,

namely inside special registers (SIMD within-a-register (SWAR), typically ranging from 128-bit to a 512-bit width).

In vector processing approaches, SIMD parallelism is used for processing regular data structures. Stream processing is another example of data parallel programming where kernels are applied in parallel rather than single instructions to a collection of data. The input streaming data is gathered before processing, and results are scattered after finishing it. 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 than the increase of memory bandwidth, the stream programming model suits better modern multi-core architectures.

With Single Program Multiple Data (SPMD) parallelism, the kernels additionally support flow control. With the introduction of branches, the kernels can execute different operations depending on conditions defined in run-time and still be efficient, as long as workload balancing is properly considered. In this model, multiple threads are allowed, however they can be executed as different sequential operations belonging to the same program. For example, modern Graphics Processing Units (GPU), namely those from NVIDIA which are conveniently supported by the recent Compute Unified Device Architecture (CUDA)[97], adopted the SPMD computational model.

**Shared memory versus distributed memory:** In a parallel system, it is fundamental to establish mechanisms that allow processors to communicate in order to cooperatively complete a task (or set of tasks). Mainly two distinct memory models can be adopted in parallel computing models and machines: shared memory and distributed memory[1].

In a shared memory model, memory can be shared by all processors or, at least, distinct groups of processors can access common banks of memory. For example, with hierarchical shared memory, faster memory can be accessed and shared inside subsets of processors, while slower global memory is accessible to all processors. Furthermore, by sharing memory within a limited number of processors (the dimension of the group of processors is architecture dependent), we also decrease the potential for access conflicts. Moreover, modern architectures support automatic mechanisms for efficient data transfers between distinct levels of memory hierarchy (e.g. cache memory, or some compilers ability to automatically place data in fast local/shared memory). However, this model introduces an extra level of complexity necessary to guarantee data integrity.

On the opposite side we have architectures that only support distributed memory models. In this case, each processor only has direct access to its own local memory.

[1] Although distributed shared memory systems exist with Non-Uniform Memory Architecture (NUMA), they will not be addressed under the context of this work.

Different processors communicate with each other over high bandwidth networks using efficient mechanisms such as, for example, Direct Memory Access (DMA) that can overlap data transfers with computation. Architectures based on the distributed memory model are scalable and easier to construct, but harder to program/debug than those based on shared memory models. On a distributed memory model, inter-processor communications have to be efficiently managed in order to allow a task (or set of tasks) to be computed cooperatively by multiple processors.

As we will see in chapter 5, GPUs consist of parallel computing machines based on shared memory models, where groups of processing units share common fast memory blocks. The Cell Broadband Engine (Cell/B.E.) is an example of a distributed memory architecture exploited in this thesis. A few standards already exist and can be used to program parallel machines based on the different memory models. OpenMP[21], or CUDA[97] are usually adopted to program general-purpose multi-core systems and GPUs, which have shared memory architectures. They are based on a set of compiler directives, library functions, or environment variables for developing parallel programs. On the distributed model we can use the Message Passing Interface (MPI). Typically, a set of library functions are provided, compliant with the MPI, that programmers can use to communicate between processors.

In future versions-to-come of current parallel architectures, the number of processors will expectedly rise. Assuming a shared memory architecture, a limit will arrive where the bus occupancy and data collisions will have a significant impact on the global processing time, causing performance to degrade. For this reason, usually this type of architecture has poor scalability when compared with distributed architectures, where each processor can perform non-conflicting independent accesses to its own memory space. However, we should be aware that to obtain short execution times in distributed systems, we must have efficient schedulers and inter-processor communication mechanisms.

### 3.2.2 Algorithm parallelization

The job of translating a sequential algorithm into an efficient parallel version of its own aims at the increase of performance and can be hard depending on the peculiarities of the algorithm and restrictions imposed by the architecture. In many cases, the best sequential algorithm indicated to solve a problem is not the best candidate for parallelization and a significantly different algorithm may be required[25].

Developing a good parallel algorithm involves identifying the amount of workload that can be parallelized, creating appropriate parallel structures to represent data, deciding how to distribute work and data among the several processing elements, as well as managing the fundamental data accesses and data transfers maintaining data coherence,

Table 3.5: Steps in the parallelization process and their goals[25].

| Step | Architecture Dependent? | Major Performance Goals |
|---|---|---|
| **Decomposition** | Mostly no | Expose enough concurrency |
| **Assignment** | Mostly no | Balance workload<br>Reduce communication volume |
| **Orchestration** | Yes | Reduce non inherent communication via data locality<br>Reduce communication and synchronization as seen by the processor<br>Reduce serialization as shared resources<br>Schedule tasks to satisfy dependencies early |
| **Mapping** | Yes | Put related processes on the same processor if necessary<br>Exploit locality in network topology |

by controlling communications and synchronization. Another important aspect we have to deal with is understanding the trade-offs between hardware and software. Not only the architecture influences the performance of the algorithm and programming decisions, but also programming decisions have an impact on the run-time characteristics of the algorithm over an architecture. The following steps can be considered for designing and implementing efficient parallel algorithms:

- decomposition of both computation into tasks and data into structures suitable for parallel accesses;
- assignment of tasks to processes or threads[2];
- orchestration of data (scheduling, transfers, accesses, communications and synchronizations) among processes/threads;
- mapping processes/threads to processing units.

Table 3.5 illustrates the architecture dependencies and how these steps in the parallelization procedure can influence the major performance goals. This procedure is performed both by the programmer and by software tools/compilers/libraries designed to support the architectures.

[2]Typically, processes contain their own state information and represent independent execution units.

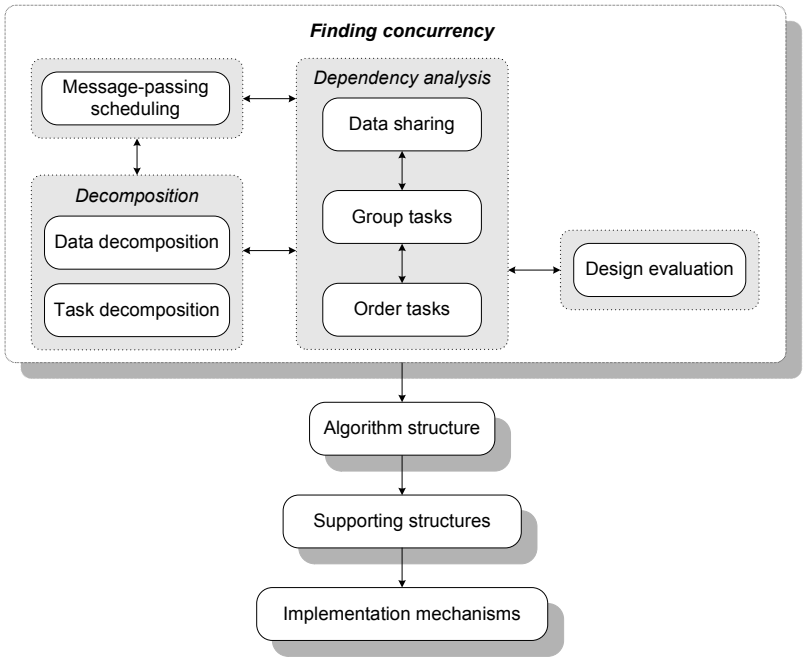


Figure 3.5: Finding concurrency design space adapted from [25] to incorporate scheduling.

**Decomposition:** Decomposition can be understood as splitting the computation into different tasks in order to expose concurrency. The order messages are passed (message-passing scheduling) between adjacent nodes of the graph has a significant impact on the decisions of breaking up the computation into blocks of tasks. Also, data can be split, concatenated or reordered into new collections of data to suit concurrency. Figure 3.5 shows that the analysis of data-dependencies influences (task and data) decomposition.

**Assignment:** Deciding which tasks are distributed over which threads is accomplished on the assignment step. The primary objective of assignment is to balance the workload and minimize inter-process communication. The assignment can be statically predetermined in the beginning of computation or dynamically reallocated during run-time. Together, decomposition and assignment define the partitioning of the algorithm and usually they are the steps that less depend on the architecture.

**Orchestration:** In order to best orchestrate the run-time behavior of the algorithm, we must understand the architecture and programming model in detail. This important step

Each process uses its own program counter, address space, and only interacts with other processes via inter-process communication mechanisms (which are usually managed by the operating system). Processes can be defined as architectural constructs, while threads can be more adequately considered coding constructs. A single process might contain multiple threads, and all threads inside the process can communicate directly with each other because they share common resources (e.g. memory space). Under the context of this text, although their differences remain clear, from now on the term *thread* will be used because it addresses more appropriately the properties (at both algorithmic and architectural levels) of this work.

uses all the available mechanisms to allow tasks inside threads accessing data, transferring it by communicating with other threads, or even synchronizing with one another. Also, the orchestration can define the organization of data structures, the optimal size of block data transfers between threads (enough to avoid starvation on the computation), or the temporal scheduling of a task to minimize the impact of communication time in the overall processing time.

**Mapping:** This last step maps threads into processors and naturally depends on the architecture. The program can control this process, or the operating system may do it dynamically in its turn, not allowing control over the mapping procedure usually to achieve effective aggregate resource sharing. Ideally, related threads should be mapped into the same processor to allow fast data communication and synchronization. Mapping and orchestration are tightly coupled, so many times these parallelization steps are performed together, to improve processing efficiency.

### 3.2.3 Evaluation metrics

The main goal of parallelizing an algorithm aims at an increase in performance, here defined as speedup which compares the performance of sequential and parallel versions of the algorithm. Although the number of cores ($P$) naturally influences speedup, typically the global increase in performance does not vary linearly with the number of cores. The speedup is a metric to evaluate the level of success achieved with the parallelization of an algorithm on an architecture with $P$ processors, by comparing with a sequential version of its own[25]:

$$Speedup_{algorithm}(P) = \frac{Sequential\ Time}{Maximum\ Time\ on\ Any\ Processor} \quad . \tag{3.4}$$

In (3.4) the term *Time* represents time spent performing the task, including computation, memory accesses, and data communications with the system's main memory as well. It gives a more approximate value of the real speedup achieved, by including the limitations of the architecture (bandwidth, concurrency accessing global memory, data collisions, multi-pipeline influences, etc.). Another usual metric for assessing the success of parallelization on $P$ cores is defined by efficiency ($Ef$), which normalizes the speedup by the number of cores:

$$Ef = \frac{Sequential\ Time}{P \times Maximum\ Time\ on\ Any\ Processor} \times 100\% \quad . \tag{3.5}$$

## 3.3 Parallel algorithms for LDPC decoding

In this section we apply the parallel computational models and the parallelization approaches for LDPC decoding. It is also discussed the influence that the different types of message-passing schedules can have in the performance of the parallel algorithms used in LDPC decoding.

### 3.3.1 Exploiting task and data parallelism for LDPC decoding

**Task parallelism in LDPCs:** In the context of LDPC decoding, the smallest task that can be defined should be responsible for updating a single edge of the Tanner graph. This approach would be efficient if the number of processors were large, and if memory accesses did not impose restrictions. Different approaches can be more efficient, for example by completely assigning a node of the Tanner graph to a task. In this case, depending on the message-passing scheduling, redundant memory accesses and extra computation can be saved [134,136], increasing performance.

**Definition 3.1.** *Intra-codeword parallelism denotes the simultaneous processing of different nodes of the Tanner graph within the same codeword, by different tasks or processing units.*

The concept associated to Definition 3.1 is illustrated in figure 3.6. It shows how intra-codeword parallelism can be exploited to achieve LDPC decoding for the example described in figure 2.1. It is based on the fact that the Tanner graph can be partitioned into distinct subsets. Each one of those subsets can then be processed by a different task or processing unit, in parallel, where tasks $t_0$ to $t_3$ are used to describe parallelism applied to the horizontal kernel of algorithms 2.1 or 2.2, and tasks $t_0$ to $t_7$ represent the parallelism applied to the vertical kernel.

**Data parallelism in LDPCs:** Data parallelism can be applied to exploit multi-codeword LDPC decoding, as defined as follows:

**Definition 3.2.** *Multi-codeword parallelism denotes the simultaneous decoding of different codewords using distinct processing units, sharing the same Tanner graph information.*

To exploit data parallelism in the context of LDPC decoding, the SIMD and SPMD approaches can be used. SIMD can be used directly for multi-codeword LDPC decoding [36], where multiple codewords are decoded simultaneously according to Definition 3.2. Multi-codeword decoding exploits the fact that the Tanner graph is common to all codewords under simultaneous decoding. This allows to apply the same instruction to different data.

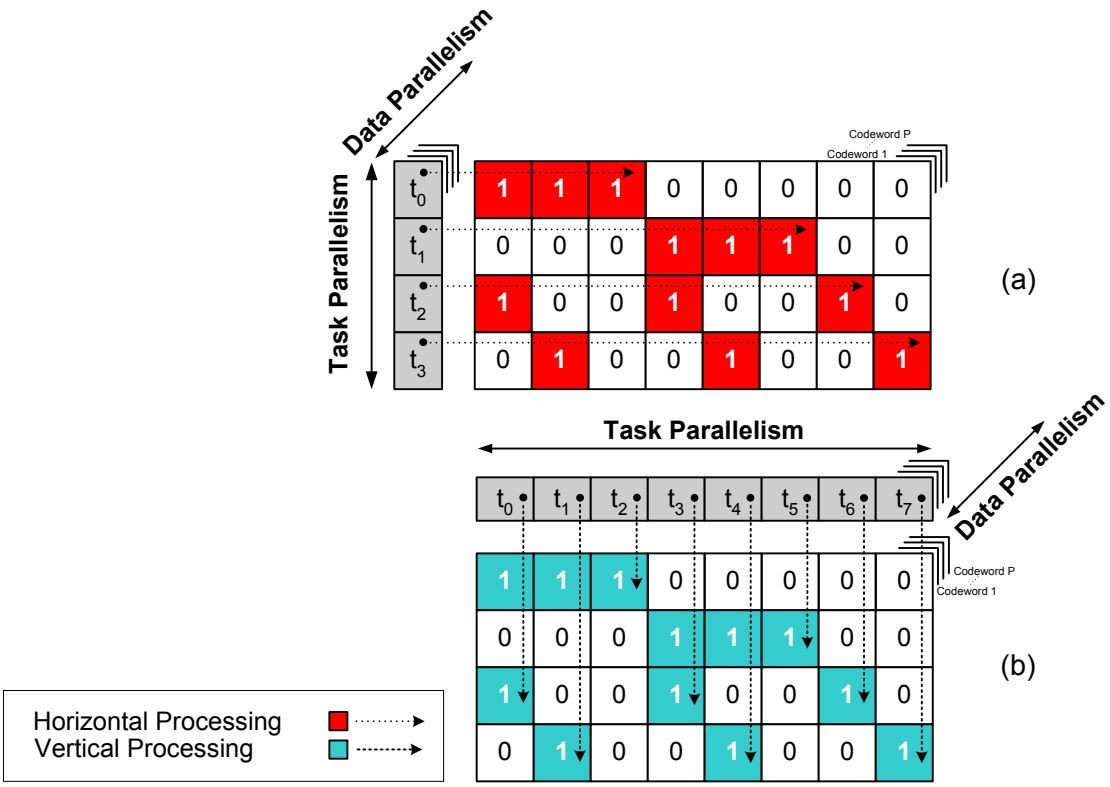


Figure 3.6: Exploiting task (intra-codeword[3]) parallelism and data (multi-codeword) parallelism for the a) horizontal and b) vertical kernels in the example of figure 2.1.

The 3D illustration in figure 3.6 represents this type of parallelism, by showing parallel decoding of distinct codewords.

A clear advantage of data parallelism over the traditional task parallelism model is the required control of the processing, which is unique. Exploiting data-parallelism, several codewords can be packed and processed together in parallel. The kernels 1 and 2 described in algorithm 2.1, were properly developed to support very intensive arithmetic computations and minimize memory accesses, by gathering and scattering data as defined by the stream-based computing model. To show how the adopted SPA can be computed according to this approach, equation (2.18) in algorithm 2.1 can be rewritten as:

$$r_{mn}^{(i)}(0) = \frac{1}{2} + \frac{1}{2} \prod_{n' \in \mathcal{N}(m) \setminus n} \left( q_{n'm}^{(i-1)}(0) - q_{n'm}^{(i-1)}(1) \right). \tag{3.6}$$

The inspection of (3.6) shows less arithmetic calculations than (2.18) at the expense of executing more memory accesses (it introduces the read of element $q_{n'm}^{(i-1)}(0)$), which typically penalizes the performance of the algorithm on multi-core devices [82]. To optimize execution in parallel computing systems, the operations initially performed in (3.6) were

[3] It can also be considered data-parallelism, but in a wider sense and under the context of this work we consider it task-parallelism, because it processes subsets of the Tanner graph simultaneously.

mathematically manipulated and are preferably represented as indicated in (2.18), in order to maximize the use of arithmetic operations regarding the number of memory accesses.

When developing code for an architecture that supports the SPMD computing model, a programmer should concentrate efforts to use the same program and to minimize test conditions. Sometimes, redesigning the algorithm may be necessary to eliminate unnecessary branches, which allows the hardware scheduler (i.e. task scheduler) to adopt a more balanced workload distribution among processors. Although under certain conditions the scheduler's behavior can be very efficient, as it will be discussed later on the text, the nature of LDPC decoders implies that each node establishes communications with other nodes. To minimize the effect of overheads introduced by redundant communications and to allow the efficient Forward-and-Backward recursion[63] to be adopted, the size of a task should not be set too small. For LDPC decoders under this model the most efficient approach experimented under this thesis computes one complete set of nodes per task. Typically, this set is composed by all the BNs associated to a CN for horizontal processing, or by all CNs associated to a BN for vertical processing.

### 3.3.2 Message-passing schedule for parallelizing LDPC decoding

Chapter 2 shows that LDPC decoding algorithms are based on iterative procedures with heavy computation and data communications associated to intensive message passing between adjacent nodes of the Tanner graph. For each type of node and iteration, the update can be performed serially (one node at a time) or in parallel (several nodes simultaneously), that the output messages at the end of an iteration will be the same. The order in which messages are propagated in the graph is relevant due to implications in the efficiency of computation and on speed of convergence, influencing the overall performance and coding gain of the algorithm[109]. The order of propagation can be defined by the message-passing schedule mechanism.

Although in ideal cycle-free codes the message-passing schedule mechanism has no influence on the convergence of the algorithm, in practice cycle-free codes are hard to achieve and high-girth codes are acceptable and used in most applications. Thus, for implementation purposes on parallel architectures, the adopted message-passing schedule procedure is fundamental to achieve efficiency. We adopted fixed message-passing schedule approaches as they allow fair trade-offs between hardware and algorithmic costs.

**Flooding schedule:** In LDPC decoding algorithms, the complete processing of one iteration is divided in two different kernels, being one responsible for updating all the

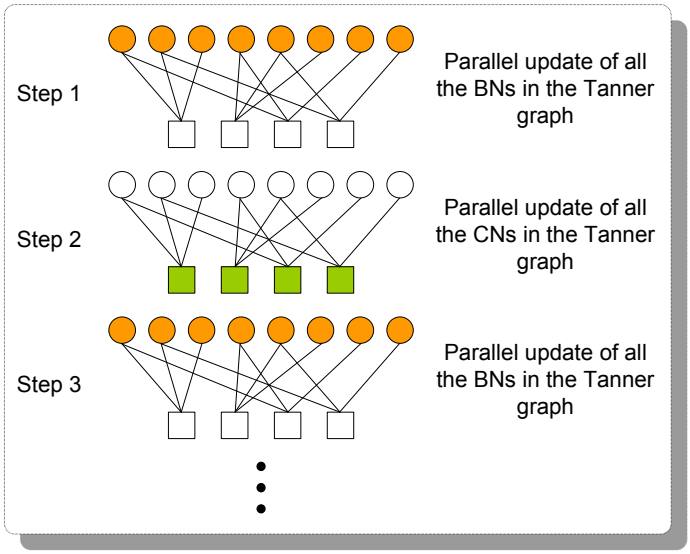


Figure 3.7: Flooding schedule approach.

BNs and the other for updating the CNs in the Tanner graph. Figure 3.7 shows that the flooding approach operates over this structure in parallel, simultaneously updating all the BNs first, and then CNs after. For each type of node, the number of nodes being simultaneously updated does not change the output results or the coding gain of the algorithm.

**Horizontal schedule:** Figure 3.8 depicts an organization of processing that updates each CN at a time. Each $\text{CN}_m$ processor collects data from the BNs connected to it, that were either produced in the previous iteration, or in the present one by $\text{CN}_{m'}$ processors, with $m' < m$, that already updated their corresponding BNs. This latter approach that uses data already updated in the current iteration, accelerates the convergence of the algorithm as proposed in [86,109,131].

**Algorithm 3.1** Algorithm for the horizontal schedule (sequential).

```
1:    /* Initializes data structures by reading a binary M × N matrix H */
2:  for all CN_m (rows in H_mn) : do
3:    CN_m ← BN_n;      /* BN_n updates CN_m... */
4:    for all BN_n (columns in H_mn) : do
5:      if H_mn == 1 then
6:            /* CN_m updates BN_n, for all n ∈ N(m) previously updated by CN_m', m' < m */
7:        BN_n ← CN_m;
8:      end if
9:    end for
10: end for
```

**Horizontal block schedule:** To accelerate the previous type of serial processing, some authors have proposed to simultaneously process blocks of $P$ CN processors in parallel.

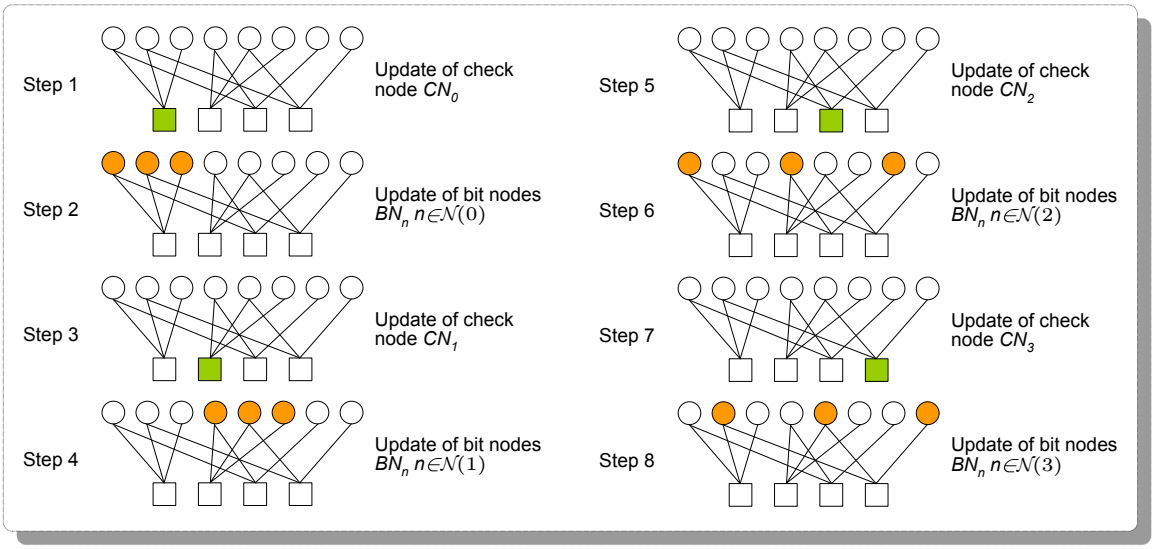


Figure 3.8: Horizontal schedule approach.

The message-passing schedule inside each of these blocks can be considered flooding for the $P$ nodes under parallel processing. Figure 3.9 shows the update of $P = 2$ CNs in parallel, and then the processing of all the BNs connected to them also in parallel. The sequential approach depicted in figure 3.8 is presented in algorithm 3.1. As mentioned before, to perform the updating procedure each $\mathrm{CN}_m$ collects data from all connected $\mathrm{BN}_n$ that might have been already updated in the same iteration, allowing faster convergence of the algorithm. The main disadvantage lies in processing sequentiality imposed by such approach. The horizontal block schedule depicted in figure 3.9 is expanded into algorithm 3.2, where groups of CNs specified by $\Phi\{.\}$ are processed in parallel. It shows the semi-parallel nature of the approach, which allows shorter execution times at the expense of convergence speed. In this case, it is important to eliminate data dependencies inside the block under parallel processing defined by $\Phi\{.\}$ (inside the block the schedule is flooding). Also, bandwidth limitations may create conflicts accessing data in memory,

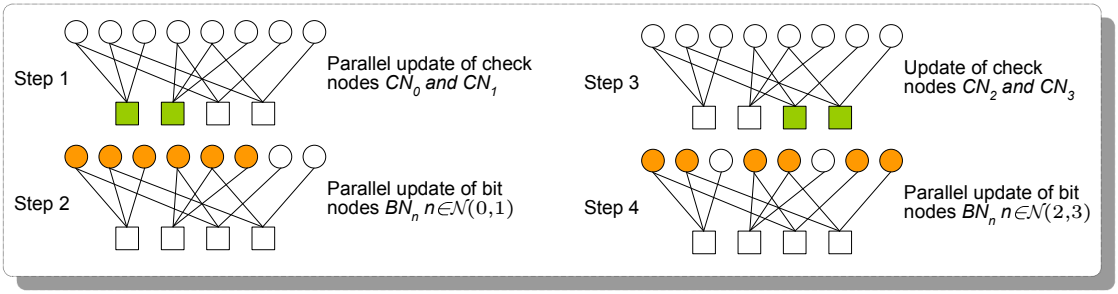


Figure 3.9: Horizontal block schedule approach.

which can impose a decrease of the speedup to values considerably below $P$ according to (3.4). Nevertheless, if the level of parallelism is high enough, as it is expectable, the global processing time necessary to conclude the process can be significantly reduced regarding the one obtained with the sequential approach.

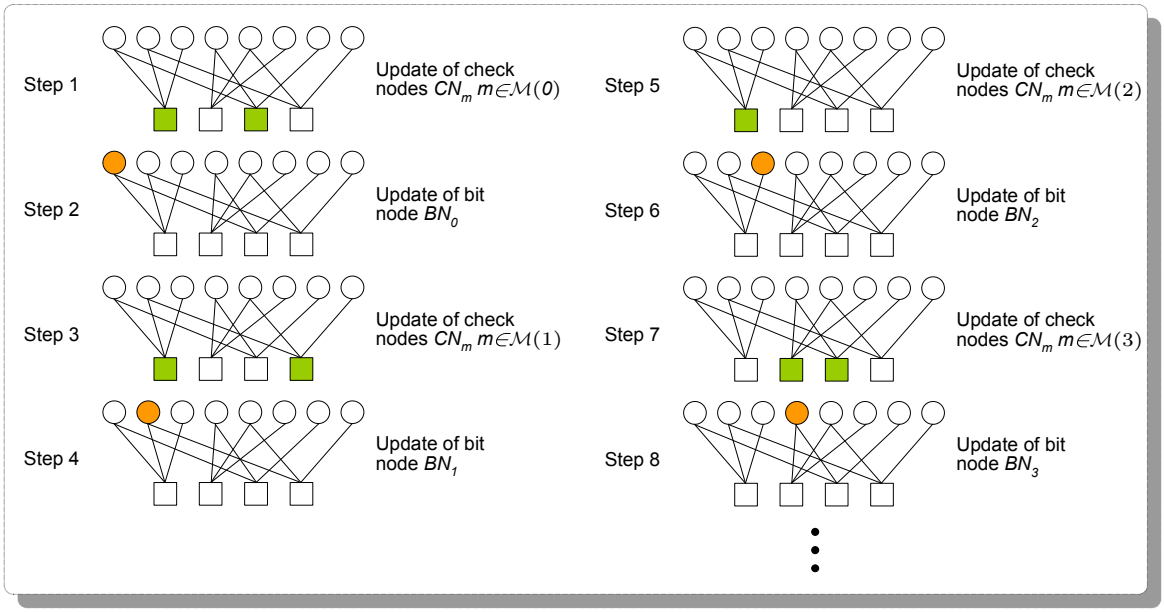


Figure 3.10: Vertical scheduling approach.

**Vertical schedule:** A similar solution has been proposed for the vertical schedule. Having in mind the intent of accelerating convergence when comparing against the traditional Belief Propagation (BP) algorithm, some authors[134–136] propose the serial processing of BN processors, where newly updated information is used as soon as it is computed

**Algorithm 3.2** Algorithm for the horizontal block schedule (semi-parallel). The architecture supports $P$ processing units and the subset defining the CNs to be processed in parallel is specified by $\Phi\{.\}$.

```
1:    /* Initializes data structures by reading a binary M × N matrix H */
2:  for all CN_{Φ{m}} (rows in H_mn) ∈ Φ{m} : do
3:        /* Parallel processing */
4:    CN_m ← BN_n;     /* BN_n performs the semi-parallel update of CN_m... */
5:    for all BN_n (columns in H_mn) : do
6:      if H_mn == 1 then
7:            /* All CN_m inside subset Φ{.} update corresponding BN_n, for all n ∈ N(m_0, m_1, m_2, ...), with
          m_i ∈ Φ{.} */
8:        BN_n ← CN_m;
9:      end if
10:     Synchronize();
11:   end for
12: end for
```

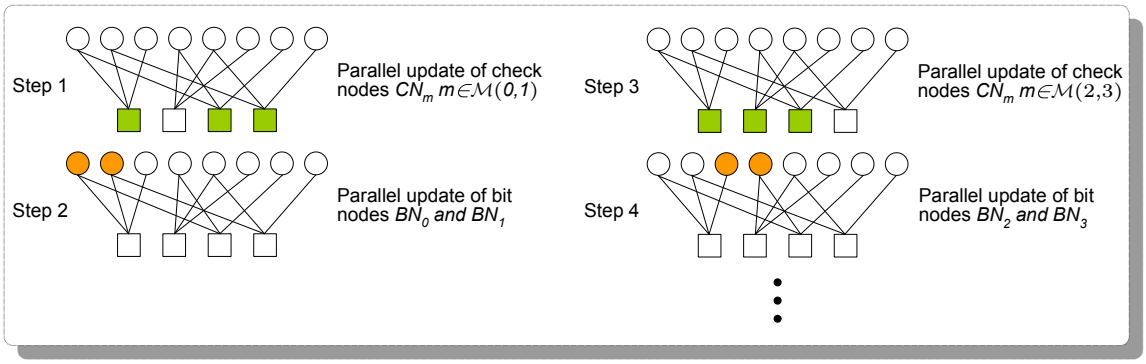


Figure 3.11: Vertical block scheduling approach.

(in the same iteration). Figure 3.10 depicts an example for this type of message-passing schedule: the step 1 shows the CNs associated to $BN_0$ being updated and step 2 updates the corresponding $BN_0$. Then, in step 3 another set of CNs connect to $BN_1$ update it, after which $BN_1$ is also updated in step 4. The process repeats until all the BNs are updated for that iteration.

**Vertical block schedule:** Again, so the decoding speed can be accelerated by grouping blocks of $P$ BNs in parallel. Within this type of message-passing schedule, a good compromise between convergence and decoding time can be achieved. The algorithms for sequential and semi-parallel vertical schedules are similar to those shown in algorithms 3.1 and 3.2 adapted to suit schedules for vertical processing.

The different types of scheduling can impose restrictions in some cases and suit the introduction of parallelism in others. The herein adopted message-passing schedule approaches are described next for each type of parallelism that we were trying to exploit in the context of LDPC decoding. From the previous discussion, where the different types of scheduling for LDPC decoding were analyzed, it can be assumed that three main classes of message-passing scheduling have been distinguished:

- serial-oriented scheduling;
- semi-parallel block scheduling;
- full-parallel flooding schedule.

The most natural approach consists in serial scheduling. Processing one node at a time wastes precious processing time and supplies no efficiency at all, which makes this

kind of scheduling uninteresting for parallel processing targeting high throughput applications.

Semi-parallel block scheduling[85], also called staggered[131,132], layered, or shuffled decoding, or even known as turbo LDPC decoding, exploits some level of parallelism while at the same time uses new information (from nodes already updated) to update data in the same iteration. The main problem in this approach is imposed by data dependencies. It represents, however, a good compromise for parallel computing architectures, and does not neglect the speed of convergence[104]. In the context of this thesis we adopted this scheduling when developing LDPC decoders for Digital Video Broadcasting – Satellite 2 (DVB-S2) targeted to Field Programmable Gate Array (FPGA) or ASIC[51,52] devices. For the CN's update, this particular scheduling allowed to perform the simultaneous processing of nodes associated both with information bits and parity bits[51,52], introducing a good level of parallelism and high throughputs[51,52].

The flooding schedule allows to exploit full parallelism and is therefore the algorithm that supports the maximum number of operations in parallel. In this approach, however, the coding gains are inferior when compared against serial scheduling[109], because data updated during the execution of an iteration will only be used in the next iteration. It does not contribute to update other data in the current iteration. This type of scheduling allowed to exploit two different approaches: *i*) the edge approach, which is ideal for programmable architectures having a large number of cores[16,107]; and *ii*) the node approach, which avoids redundant accesses to memory. In the edge approach, each processing unit is responsible for updating a single message and if we assume an architecture with a significant number of cores (e.g. thousands of cores) and limited memory access conflicts, the level of parallelism obtained with such approach is optimal. In this work, several tentatives were made and interesting results were obtained[32,37,40,41]. The node approach was applied with the Forward-and-Backward strategy[63,134,136], which allowed to identify and eliminate redundant computation to achieve even better results. This approach was followed in this thesis and has shown superior results which can be found in[34,36,38,39].

### 3.3.3 Memory access constraints

To achieve the goal of obtaining efficient parallel algorithms for software LDPC decoders, we have also developed compact data structures to represent the sparse binary **H** matrices that describe the Tanner graph. They adequately allow representing data as streams to suit the parallel processing of the SPA and MSA in stream-based architectures. Also, they represent a different solution from the conventional compress row storage (CRS) and compress column storage (CCS) formats[78,92] used to represent sparse

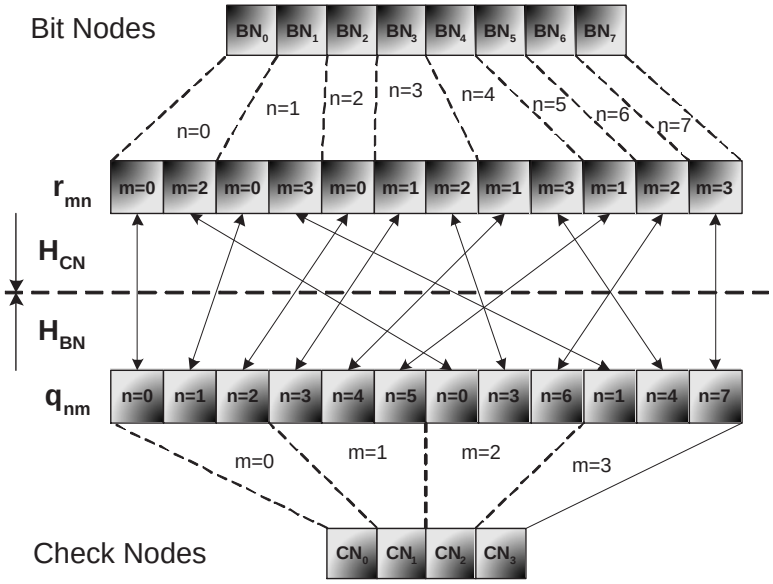


Figure 3.12: Data structures illustrating the Tanner graph's irregular memory access patterns for the example shown in figures 2.1 and 2.5.

matrices in a compact format. In the context of parallel processing based on hardware that supports data streams, the edges of the Tanner graph are represented using a proper addressing mechanism, which facilitates the simultaneous access to different data elements required by the SPA and MSA algorithms. The compact data structures herein proposed also suit parallel architectures, which can impose restrictions associated to the size and alignment of data. Furthermore, the simultaneous decoding of several codewords has been made possible thanks to the new stream-based data structures developed.

**Memory access patterns:** The **H** matrix of an LDPC code defines a Tanner graph, whose edges represent the bidirectional flow of messages being exchanged between BNs and CNs. We propose to represent the **H** matrix in two distinct data structures, namely $\mathbf{H}_{BN}$ and $\mathbf{H}_{CN}$. These two structures are useful for the horizontal and vertical processing steps described previously, respectively in kernel 1 and kernel 2, of algorithms 2.1 and 2.2. $\mathbf{H}_{BN}$ defines the data structure used in the horizontal step. It is generated by scanning the **H** matrix in a row-major order and by sequentially mapping into $\mathbf{H}_{BN}$ only the BN edges represented by non-null elements in **H**, for each CN equation (in the same row). Algorithm 3.3 describes this procedure in detail for a matrix having $M = (N - K)$ rows and $N$ columns. Steps 5 and 6 show that only edges representing non-null elements are collected and stored in consecutive positions inside $\mathbf{H}_{BN}$, for each row associated with a given CN. The vector $\mathbf{w_b}(k)$ holds the number of ones per column $k$, supporting the two distinct types of regular or irregular codes. This data structure is collected from the

**Algorithm 3.3** Generating the compact $\mathbf{H}_{BN}$ structure from the original $\mathbf{H}$ matrix.

1: Reading a binary $M \times N$ matrix $\mathbf{H}$
2: $idx \leftarrow 0$;
3: **for all** $CN_m$ (rows in $\mathbf{H}_{mn}$) : **do**
4: **for all** $BN_n$ (columns in $\mathbf{H}_{mn}$) : **do**
5: **if** $\mathbf{H}_{mn} == 1$ **then**
6: $\mathbf{H}_{BN}[idx++] \leftarrow n + \underbrace{\sum_{k=0}^{m-1} \mathbf{w_b}(k)}_{mw_b\ (regular\ codes)}$;
7: **end if**
8: **end for**
9: **end for**

original $\mathbf{H}$ matrix scanned in a column-major order:

$$\mathbf{w_b}(k) = \sum_{m=0}^{M-1} \mathbf{H}_{mk}, \quad 0 \leq k \leq N-1. \tag{3.7}$$

The $\mathbf{H}_{CN}$ data structure is used in the vertical processing step, and it 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). As presented in algorithm 3.4, $\mathbf{H}_{CN}$ is generated by scanning the $\mathbf{H}$ matrix in a column-major order. Both data structures are depicted in figure 3.12, where the example shown in figures 2.1 and 2.5 is mapped into the corresponding edge connections that form such a Tanner graph. In

**Algorithm 3.4** Generating the compact $\mathbf{H}_{CN}$ structure from the original $\mathbf{H}$ matrix.

1: Reading a binary $M \times N$ matrix $\mathbf{H}$
2: $idx \leftarrow 0$;
3: **for all** $BN_n$ (columns in $\mathbf{H}_{mn}$) : **do**
4: **for all** $CN_m$ (rows in $\mathbf{H}_{mn}$) : **do**
5: **if** $\mathbf{H}_{mn} == 1$ **then**
6: $\mathbf{H}_{CN}[idx++] \leftarrow m + \underbrace{\sum_{k=0}^{n-1} \mathbf{w_c}(k)}_{nw_c\ (regular\ codes)}$;
7: **end if**
8: **end for**
9: **end for**

order to allow generating the $\mathbf{H}_{CN}$ data structure, $\mathbf{w_c}(k)$ should hold the number of ones per row $k$, which again can be constant for regular codes. The vector $\mathbf{w_c}(k)$ is collected by scanning the original $\mathbf{H}$ matrix in a row-major order, to obtain:

$$\mathbf{w_c}(k) = \sum_{n=0}^{N-1} \mathbf{H}_{kn}, \quad 0 \leq k \leq M-1. \tag{3.8}$$

Data structures $\mathbf{H}_{BN}$ and $\mathbf{H}_{CN}$ are also depicted in figure 3.13 to illustrate the generic irregular case. It can be seen that they represent addresses (or indices) of elements to update. $\mathbf{H}_{BN}$ contains row weights in vector $\mathbf{w_c}$ and the corresponding addresses of $r_{mn}$

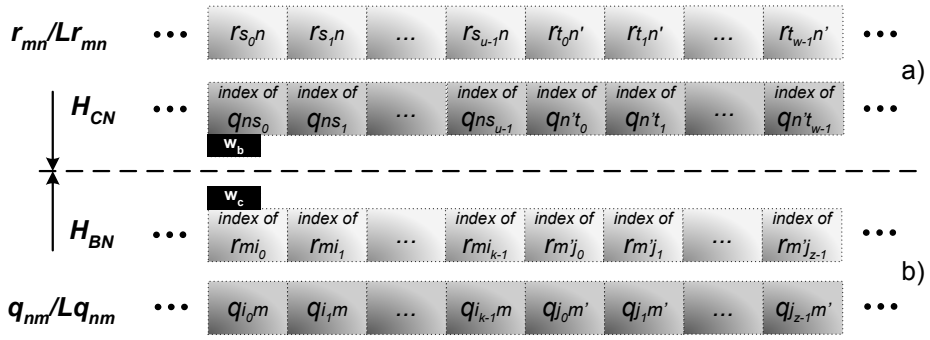


Figure 3.13: a) $\mathbf{H}_{CN}$ and b) $\mathbf{H}_{BN}$ generic data structures developed, illustrating the Tanner graph's irregular memory access patterns.

data elements to be updated by corresponding $q_{nm}$ messages arriving from nodes they connect to. The structure $\mathbf{H}_{CN}$ performs the dual operation: it contains column weights in vector $\mathbf{w_b}$ and the corresponding addresses of $q_{nm}$ elements which have to be updated by incoming $r_{mn}$ messages from connected neighboring nodes.

For regular codes, the summations indicated in algorithms 3.3 and 3.4 can be simplified, respectively into $mw_b$ and $nw_c$. In those cases, the number of ones per row or column is fixed, and all the elements of vectors $\mathbf{w_b}(k)$ and $\mathbf{w_c}(k)$ are equal and can be represented by a constant scalar avalue.

**Parallelizing memory accesses:** Multi-core architectures provide resources that allow to achieve parallel/low latency accesses to memory, typically by masking data transfers with computation[24,36,38], or by using efficient data alignment/resizing strategies, that facilitate parallel accesses to memory[34,39,97]. To better suit these mechanisms, data blocks to be accessed in memory (for example, sub-blocks of $\mathbf{H}_{BN}$ and $\mathbf{H}_{CN}$) should obey to specific size and address requirements, which may not be directly achievable from the data structures in some cases. As it will be described later in the text, we propose runtime data realignment mechanisms[34] to allow data to be accessed more efficiently, since it is not possible to align it at compile time in those cases. Also, as the nature of LDPC codes is irregular and memory access patterns are random (figure 3.14 illustrates this problem), they do not admit the simultaneous use of aligned memory accesses for both read and write operations of the decoding process. To overcome this difficulty we have developed data mapping transformations which allow multiple addresses to be contiguously accessed for one of the mentioned memory access types[34] (we adopted contiguous read accesses and non-contiguous write accesses in this case). These transformations will be discussed in more detail in the next chapters.

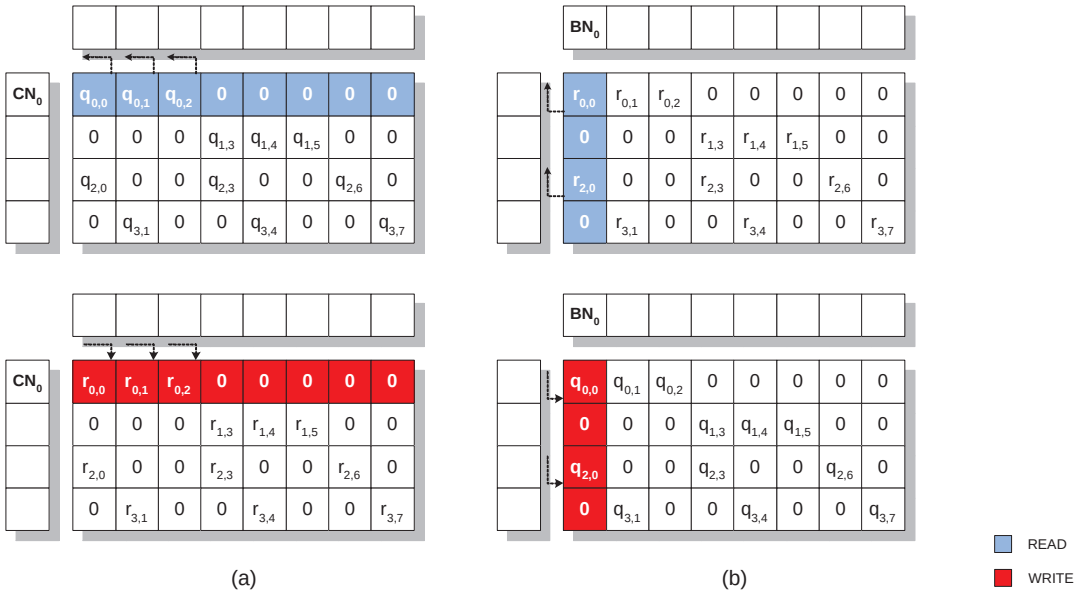


Figure 3.14: Irregular memory access patterns for $BN_0$ and $CN_0$.

## 3.4 Analysis of the main features of parallel LDPC algorithms

Problem-constrained scaling exists when we want to solve the same sized problem on a larger machine. In time-constrained scaling, the time to perform the execution is fixed, and we want to do as much work as possible in that same period of time. It is not the case in LDPC decoding. The target of LDPC decoders is to achieve a certain level of throughput (decoded bits-per-second), which should remain approximately constant independently of the length of the LDPC code (problem size). If larger codes (with a higher number of edges and, consequently, with more data to be processed) have to be decoded, there will be more time available to perform that processing (maintaining throughput constant, and keeping in mind that latency should remain low, under a certain maximum limit set by the corresponding communication's standard using the LDPC decoder). Very often, scaling is mainly constrained in memory, which means that the problem in scaling lies in the accesses to memory. This is the problem that arises in practice when the problem is scaled. The memory wall constraint is expected to be maintained in the next years with processing capacity increasing much faster than the memory technology.

### 3.4.1 Memory-constrained scaling

In order to be able of making adequate choices when selecting the appropriate architecture for an application (and more importantly, to be able of correctly predicting how architectures and application characteristics and demands will evolve in the future to stay ahead of imminent trends), it is fundamental to analyze the evolution of scaling perfor-

mance. One interesting metric for evaluating scalability is to identify the largest possible problem size that will not cause memory accesses on a machine to become the bottleneck of the application. Ideally, we would like to find the optimal size of the problem after which collateral overheads due to data communications will mainly dominate the overall execution time. By doing so, we expect to take advantage of favorable arithmetic intensity (typical in problems with large sizes) and to abundantly exploit concurrency for parallel computation. As discussed in[38,39] multi-core and many-core machines, such as the GPU and the Cell/B.E. can decode large LDPC codes efficiently. In fact, for processors with hundreds of cores, the limit has not yet been reached for codes with size $N = 20000$, or even for DVB-S2 codes which have $N = 64800$, as it will be described later in the text. However, as we increase the size of the codes, memory accesses can represent a key factor for achieving scalability.

### 3.4.2 Scalability of parallel LDPC decoders

For analyzing scalability, both the size of the problem and characteristics of the architecture must be considered. Assuming we have a fixed size problem to be computed on an architecture with multiple cores, if we increase the number of cores the overhead in communications and management may also increase until a point were it will dominate the overall processing time, producing uninterestingly small speedups. Eventually, after a certain number of processors the problem may become too small to properly exploit the potential parallelism allowed by the architecture. On the opposite situation, reverse problems may also occur. If a certain large problem scales well on large machines, if we decrease the number of processors, for example, until the case-limit of having only a single processor, the size of local memory may not be large enough to accommodate the entire problem. Therefore, the overhead of data movements between the different levels of memory hierarchy may significantly decrease the processing efficiency. Consequently, to pursue well-defined scaling models we must adopt problems with varying sizes on machines with varying capacities.

Under the context of LDPC decoding, the scaling approaches can manipulate two distinct features: the length of the LDPC code; and the support of multi-codeword decoding. The length of an LDPC code impacts the granularity level adopted in the partitioning of the algorithm across the processing units. Interestingly, if the number of processors is large enough (i.e. in the order of hundreds[37]), codes with superior lengths can be decoded with performances (throughput) superior to those obtained with smaller length codes, as described in[34,37]. Another interesting property that suits scalability when the size of the problem is not large enough to fully exploit the parallel resources supplied by the architecture, is associated to the new concept of multi-codeword decoding that

we introduce and exploit for parallel architectures[34,36]. In fact, if the number of cores is larger than the size of the problem would ideally request, other than we put different processors decoding different parts of the Tanner graph for the same codeword, we can additionally decode multiple distinct codewords in simultaneous. This introduces different levels of parallelism, all well supported by the convenient SPMD programming model, which allows an efficient adaptation of the size of the problem to the dimension of the architecture. Thus, the number of codewords simultaneously under decoding may expectedly rise with the number of cores in the future.

## 3.5 Summary

This chapter analyzes the computational complexity associated with LDPC decoding and describes basic strategies for finding concurrency. The identification of these strategies is of great importance to design efficient parallel algorithms. Task parallelism can be adopted when the architecture supports multiple cores decoding different parts of the Tanner graph. Data parallelism is mainly exploited by the SIMD and SPMD approaches, where typically the same instruction or kernel is applied to multiple data allowing the simultaneous decoding of distinct codewords.

We also analyze the serious restrictions that result in some parallel architectures when LDPC decoders try to perform parallel accesses to non-contiguous locations of data in memory. Also, message-passing scheduling mechanisms used in LDPC decoding are addressed with the objective of analyzing data-dependencies and the performance of these algorithms when used in parallel computing architectures.

The analysis of scalable LDPC decoding solutions is performed at the end of this chapter, which represents an important aspect for the next generations of multi-core architectures that expectedly will have a higher number of cores.

*"The number of transistors on a chip will double about every two years."*

*Gordon Moore, 1965*

# 4

# LDPC decoding on VLSI architectures

Contents

In the previous chapter we have seen how the use of parallelism can be exploited in the iterative message-passing algorithms used in Low-Density Parity-Check (LDPC) decoding. Some applications can impose challenges which typically have to be addressed by using hardware-dedicated solutions. It is, for example, the case of mobile equipment which demands System-on-Chip (SoC) hardware providing good performance at low-cost, consuming low power and occupying small die areas. In the last years, we have seen LDPC codes considered for adoption in many communication systems, as they became a consistent alternative to their natural competitors – Turbo codes. As a consequence, they have been adopted by important standards, namely those mentioned in chapter 1, where a natural emphasis has been given to solutions addressing the recent Digital Video Broadcasting – Satellite 2 (DVB-S2) standard for satellite communications[29] and the WiMAX standard for local and metropolitan area networks[64]. They have also been incorporated in other types of communication systems, namely in advanced magnetic storage and recording systems[62,113,138] or even in optical communications[120].

LDPC codes with good coding performances demand very long lengths[79,84], as those adopted in the DVB-S2 standard. In order to achieve high-throughput decoding, in this chapter we develop a novel partial parallel Very Large Scale Integration (VLSI) architecture that competes with state-of-the-art hardware solutions for the challenging DVB-S2 LDPC decoders[27,70,95,133]. The solution exploits the periodic properties of the codes addressed in chapter 2 of this thesis. The architecture is supported in either Field Programmable Gate Array (FPGA) or Application Specific Integrated Circuits (ASIC) based devices and synthesis results are presented for both solutions. While FPGAs can offer a significant number of reconfigurable gates at low-cost, ASIC solutions achieve high performance with low power consumption and within a small area context. We also present synthesis results for an optimized ASIC architecture which prove to be competitive with existing state-of-the-art solutions.

## 4.1 Parallel LDPC decoder VLSI architectures

The two types of nodes in the Tanner graph that perform the computation described in algorithms 2.1 and 2.2, impose the need of distinct BN and CN processing. Therefore, processor nodes able of performing these two types of calculations are fundamental components of LDPC decoding architectures. Also, it is necessary to incorporate memory to store messages associated with each processor node, and the decoder requires an efficient switching mechanism to allow each processor node in the graph to access different memory banks. Ideally, this should be performed in parallel to increase throughput. In order to achieve such an efficient solution, a parallel LDPC decoder can be mapped

into a generic architecture[1] like the one depicted in figure 4.1. The message-passing scheduling mechanism controls the overall execution of the algorithm. According to subsection 3.3.2, it defines the order messages are updated (controlling the order memory is addressed by each node processor), this way defining which nodes communicate with each other and in what order. The switching selector in figure 4.1, equally dependent on scheduling, controls the interconnection network that performs this task. It connects $Q = N$ BN and $P = (N - K)$ CN processors to corresponding memory blocks, in order to allow read/write operations (to be performed in parallel) into different subsections of the graph. The interconnection network should be designed in order to allow conflict-free accesses to memory banks. As before, the update of messages is performed iteratively, with BN processing alternating with CN processing. Both types of node processors depicted in figure 4.1 should be prepared to support the processing of BN or CN nodes with different weights $w_b$ and $w_c$, respectively.

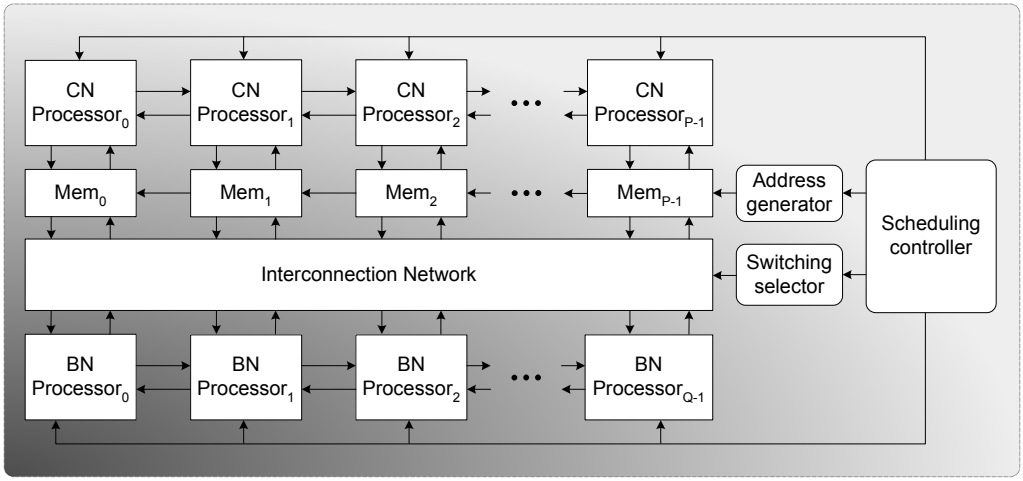


Figure 4.1: Generic parallel LDPC decoder architecture[1] based on node processors with associated memory banks and interconnection network (usually, $P = Q$).

The complex nature of such a VLSI architecture presents several challenges. To perform fast LDPC decoding, it requires a certain level of parallelism, which should have some significance in order to allow achieving the minimal necessary throughput that can be very demanding (90 Mbps in the DVB-S2[29] case). On the other hand, above some level of parallelism the requirements for routing, area and power consumption cannot be achieved for practical reasons. Presently, there are LDPC decoder architectures ranging from serial[102] to fully parallel[13,61], targeting different objectives. Serial decoders trade throughput with architectural simplicity, obtaining reduced die areas. Although limitations at the processing level can make them targeted mainly for low-throughput applications, they can exploit hardware simplifications, such as reducing the storage size of messages[102], and still achieve coding gains without noticeable loss in performance[102].

Parallel approaches offer superior throughput at the expense of architectural complexity. On a fully parallel system, the complexity increases significantly as the code length grows. The dimension and complexity of this type of architectures are only acceptable for short to medium length LDPC codes. One of the first ASIC publications in the area[13,61] proposes a fully parallel architecture for a 1024-bit, $rate = 1/2$ LDPC decoder processing 64 iterations, that obtains a throughput of 1Gbps. The excellent performance achieved is only possible due to the complex routing established to connect all $Q = N$ BN processors with corresponding $P = (N - K)$ CN processors, as depicted for a generic architecture in figure 4.1. Consequently, the complex wiring of the circuit created the extra need of carefully managing the floor planning and routing of the design, which alerted for the prohibitive complexity involved in the design of architectures that support block codes with higher lengths ($N > 1024$-bit).

In recent years, LDPC scalable decoder architectures targeting FPGA devices have also been proposed[11,121,124]. FPGAs combine the advantages of scalable hardware with reprogrammable architectures, increasing therefore flexibility at the expense of area capabilities. However, the memory available in those platforms can impose restrains in the design[124], or because the total number of codes supported within a single architecture is reduced[11,124]. Flexible architectures that support long length codes and that can decode any structured or unstructured LDPC code, have also been proposed for FPGA devices[11,121] at the expense of hardware complexity. Alternative interconnection networks that allow implementing simpler routing have also been designed[121] for hardware-constrained LDPC decoders in FPGAs.

Naturally, the size of the architecture is influenced by the length of the largest code supported and also by the level of parallelism adopted. This has direct implications in the number of node processors necessary to perform computation and, most importantly, in the complexity of the interconnection network of figure 4.1 which allows all processors to establish communications with each other. The complexity of this network decreases by reducing the level of parallelism of the architecture. For a fixed code, permutation laws are known and the indices/addresses that indicate which nodes connect with each other are usually computed offline. In these cases, Benes networks[7] can provide full connectivity requiring reduced complexity of $O(N\,log_2 N)$ when compared to crossbar switching architectures[100].

Flexible interconnection structures for full-parallel or partially parallel LDPC decoders with reduced communication complexity have been suggested[44], although they do not address large LDPC codes used in demanding applications such as DVB-S2. Because parallel hardware-dedicated systems usually achieve very high throughputs, recently partial-parallel architectures[18,27,70,95,133] have been proposed to tackle the complexity

of the interconnection network, and still achieve high throughputs. They reduce the number of node processors required to perform LDPC decoding, by following an efficient time multiplexing strategy that processes alternatively sub-sets of the Tanner graph. Compared with full-parallel decoders, the complexity of an interconnection network in a partial-parallel architecture decreases abruptly, which eliminates important restrictions from the routing process (in the place & route phase of the design).

Efficient processor units to perform computation in this type of architectures have also been developed[27,95,133]. Although they do not occupy the largest part of the circuit area, they also impact power consumption. Hardware solutions based on VLSI technology usually apply the algorithm based on fixed-point arithmetic operations[101]. The level of quantization adopted naturally influences the performance of a code, namely its coding gains, but it also affects the datapath with direct consequences in the area of processor nodes. The Sum-Product Algorithm (SPA) used in LDPC decoding is sensitive to the quantization level and variants to the SPA less sensitive to quantization effects have been proposed[101]. Nevertheless, the development of efficient hardware LDPC decoders most exclusively adopts solutions based on the Min-Sum Algorithm (MSA). They demand simpler processing units with gains in area and power consumption, without a significant performance loss[63].

Over the last years, a diversity of new architectural approaches has been proposed. A joint partially parallel LDPC low-complexity encoder/high-speed decoder design for regular codes has been proposed[137], as well as modifications to the architecture that allow trading hardware complexity for speed on the decoder side. Other partial-parallel architectures for high-throughput LDPC decoding have been proposed[87,88], where the concept of architecture-aware LDPC codes was firstly introduced. Architecture-aware codes decouple the architectural dependence of the decoder from the LDPC code properties. They achieve a faster convergence rate, which allows obtaining higher throughputs[87]. In this case the interconnection network can be efficiently implemented by using programmable multi-stage networks[88]. A comparison between serial, partly-parallel and full-parallel architectures has been performed in[43] for a regular code with $N = 2048$ bits, $w_c = 6$ and $w_b = 3$. However, these architectures are dedicated to particular cases and other solutions were necessary to tackle the general problem, which should be able of supporting other types of more demanding LDPC codes (e.g. irregular and higher block length codes).

Typically, more than 70% of the circuit's area in VLSI LDPC decoders is occupied by memory units[27,95], which are essential to support the iterative message-passing mechanism. They are extensively addressed in the context of this work. Memory requirements for parallel architectures able of supporting long length codes are feasible, but

hard to achieve for practical purposes within a VLSI context. The complexity and amount of memory necessary to store messages exchanged between hundreds or thousands of nodes is extremely high, which imposes restrictions in die area and power consumption. Although the global amount of memory necessary to store messages between connected nodes of the graph remains unchanged for any number of processors, by varying the number of processor nodes in a partial-parallel approach, the height and width of memory blocks change, which can be exploited to achieve more efficient architectures.

## 4.2 A parallel M-kernel LDPC decoder architecture for DVB-S2[§]

The development of LDPC decoders for the DVB-S2 standard can be considered to be among the most demanding applications that support this type of processing, mainly due to the high length of codes adopted [29]. As shown in subsection 2.6.2, LDPC codes used in this standard are Irregular Repeat Accumulate (IRA) codes, where the sparse parity-check **H** matrix has periodic properties that can be exploited to achieve hardware parallelism [30].

The utilization of scalable parallelism to obtain chips with small areas for DVB-S2 has been proposed for technologies ranging from 0.13$\mu$m down to 65nm [18,27,70,95,133] based on the processing of sub-sets of the Tanner graph, which guarantee the minimum necessary throughput of 90Mbps required by the standard [29]. The design of partial-parallel solutions generates systems with lower areas, but also implies lower throughputs, comparing to full-parallel architectures in a SoC. One of the initially proposed architectures [70] is based on figure 4.2 with $M = 360$ processor nodes, also referred in this text as Functional Units (FU). They work in parallel and share control signals, that process both CN nodes (in CN mode) and IN nodes (in BN mode) according to a flooding schedule approach. Attending to the zigzag connectivity between parity nodes (PN) and CNs [70], they are

[§]The results presented in this and the following sections have been partially presented in:

[31] Falcão, G., Gomes, M., Gonçalves, J., Faia, P., and Silva, V. (2006). HDL Library of Processing Units for an Automatic LDPC Decoder Design. In *Proceedings of the IEEE Ph.D. Research in Microelectronics and Electronics (PRIME'06)*, pages 349–352.

[49] Gomes, M., Falcão, G., Gonçalves, J., Faia, P., and Silva, V. (2006a). HDL Library of Processing Units for Generic and DVB-S2 LDPC Decoding. In *Proceedings of the International Conference on Signal Processing and Multimedia Applications (SIGMAP'06)*.

[50] Gomes, M., Falcão, G., Silva, V., Ferreira, V., Sengo, A., and Falcão, M. (2007a). Factorizable modulo M parallel architecture for DVB-S2 LDPC decoding. In *Proceedings of the 6th Conference on Telecommunications (CONFTELE'07)*.

[51] Gomes, M., Falcão, G., Silva, V., Ferreira, V., Sengo, A., and Falcão, M. (2007b). Flexible Parallel Architecture for DVB-S2 LDPC Decoders. In *Proceedings of the IEEE Global Telecommunications Conf. (GLOBECOM'07)*, pages 3265–3269.

[52] Gomes, M., Falcão, G., Silva, V., Ferreira, V., Sengo, A., Silva, L., Marques, N., and Falcão, M. (2008). Scalable and Parallel Codec Architectures for the DVB-S2 FEC System. In *Proceedings of the IEEE Asia Pacific Conf. on Circuits and Systems (APCCAS'08)*, pages 1506–1509.

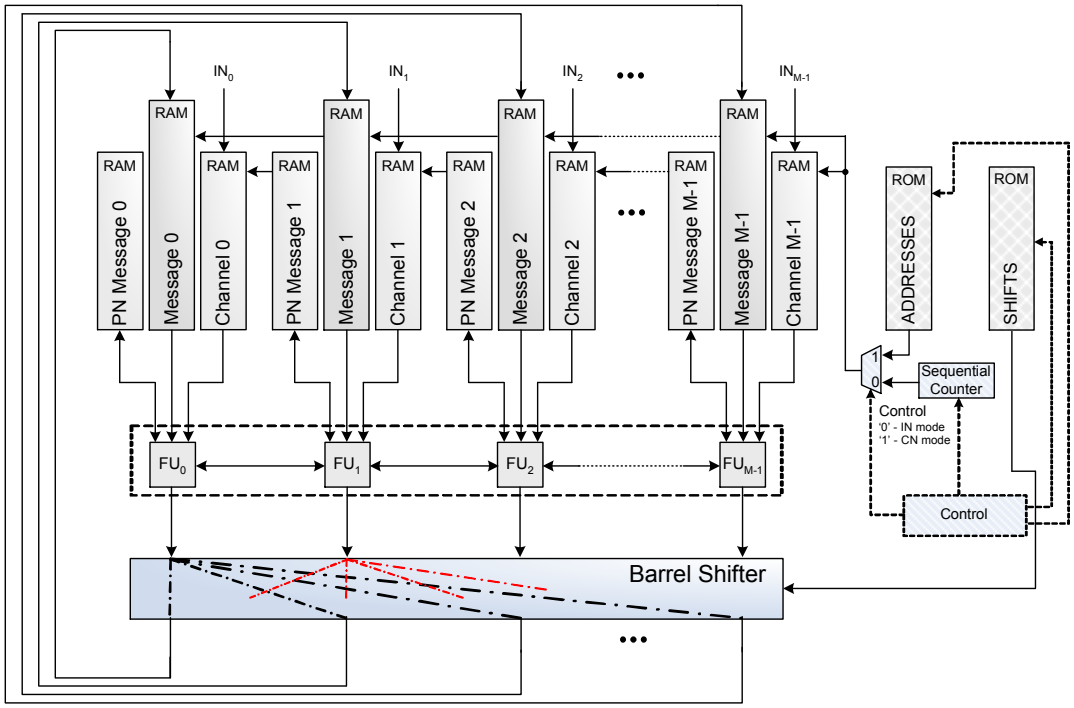


Figure 4.2: Parallel architecture with $M$ functional units (FU)[51,70].

updated jointly in check mode following an horizontal schedule approach[49,51,109] as shown in subsection 3.3.2. All 360 FUs process one message per cycle. In IN mode, all 360 messages are read from the same address (sequentially incremented, for this type of processing) in corresponding memory blocks as depicted in figure 4.3. The new messages resulting from this computation are then stored in the same address cyclicly shifted right (controlled by the values in the *SHIFTS* memory block of figure 4.2) through the interconnection network, which is implemented by a barrel shifter. The periodic properties of LDPC codes used in DVB-S2 allow to replace the complex interconnection network by a common barrel shifter. In CN mode, messages have to be read from specific addresses (present in the *ADDRESSES* memory bank) and stored back in the same addresses cyclicly shifted left to conclude the execution of an iteration. Once again, the access is performed in parallel for all 360 messages. The barrel shifter mechanism and the efficient memory mapping scheme from figure 4.3, constitute the major strengths of the architecture described in[70].

Memory requirements for a partially parallel architecture capable of supporting long length codes used in DVB-S2 are demanding for practical purposes. The appropriate *Control* of *ADDRESSES* and *SHIFTS* memory banks indicated in figure 4.2 guarantees that every time the address of a memory block changes, it changes accordingly and without conflicts for all $M = 360$ processors in parallel. The barrel shifter, which has a switching activity controlled by the *SHIFTS* memory bank, can be properly managed together

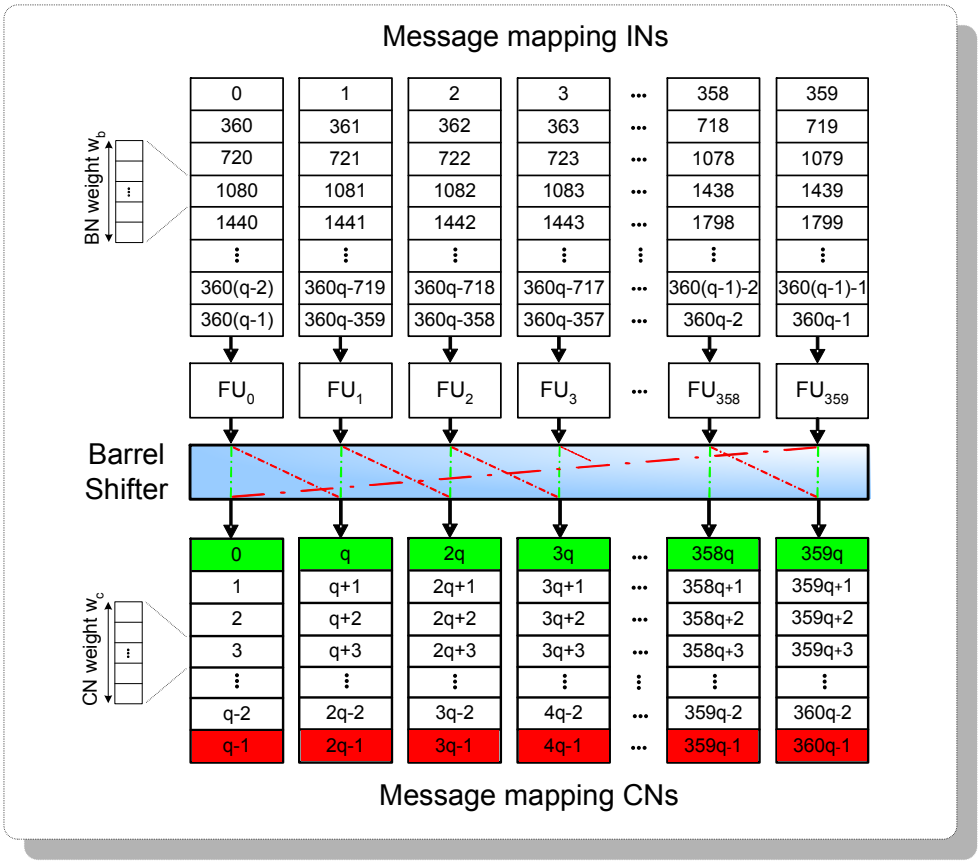


Figure 4.3: Memory organization for the computation of $Lr_{mn}$ and $Lq_{nm}$ messages using $M = 360$ functional units for a DVB-S2 LDPC code with $q = (N - K)/M$ (see table 2.3).

with the *ADDRESSES* memory block that defines memory accesses, to simulate the connectivity of the Tanner graph. Figure 4.3 illustrates the memory mapping mechanisms used in BN processing as a function of $q$ (a parameter defined by the standard) and presented in (2.35). In CN processing, a similar scheme applies. For each different code we use different *SHIFTS* and *ADDRESSES* data values, which can be easily obtained from annexes B and C of the DVB-S2 standard[29]. In figure 4.2 mainly three distinct types of memory are depicted: *i*) *Channel* memory which initially receives data (IN+PN) from the input of the LDPC decoder; *ii*) *Message* memory, where all messages associated with information bits are stored (the FU supports both types of BN and CN processing, which perform alternately during the same iteration); and *iii*) *PN Message* memory that holds parity bits, which are computed in CN mode and have all weight 2 (zigzag connectivity). In *PN Message* memory each FU only needs to store the message which is passed during the backward update of CNs[49].

However, the large number of FUs used still implies a long width of the barrel shifter which requires a significant die area and imposes routing problems: figure 4.2 shows that the architecture will have to support a Very Long Data Word (VLDW) bus in order to

accommodate the simultaneous accesses of all FUs to corresponding messages in memory. Since this architecture is able to provide a throughput far above from the minimum mandatory rate of 90 Mbps, we may reduce the number of necessary FUs even further. In fact, we show in this chapter that this can be done by an integer factor of $M = 360$[51] with the corresponding beneficial reduction of the size of the barrel shifter.

## 4.3 M-factorizable LDPC decoder for DVB-S2

Under this context we propose a novel hardware approach[31,51,52] which is based on a partial-parallel architecture that simplifies the barrel shifter and imposes acceptable complexity and memory requirements. We address the generalization of the well known $M$-kernel parallel hardware structure[70] and propose their partitioning by any integer factor $L$ of $M$[51], without memory addressing/reconfiguration overheads and keeping unchanged the efficient message-passing mechanism. The proposed method[51] provides an efficient way of reducing the number of FUs and the overall complexity of the decoder. This approach does not only surpass some disadvantages of the architecture described in[70], such as die area occupied or routing congestion, as it also adds flexibility

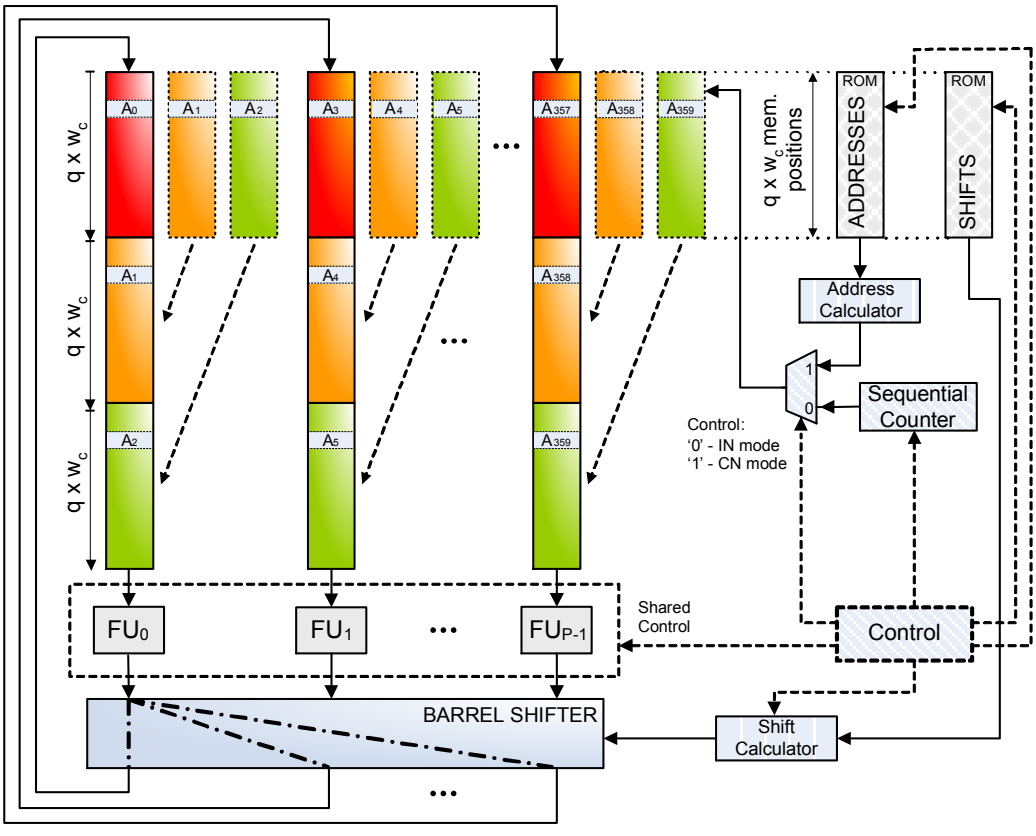


Figure 4.4: $M$-factorizable architecture[51].

and reconfigurability to the system according to the decoder and device constraints. This type of sub-sampling approach[51] preserves the key modulo $M$ properties of the architecture described in figure 4.2[70], with only $P$ processing units as shown in figure 4.4. This strategy[51] allows a linear reduction of the hardware resources occupied by the $P$ FU blocks (which can be obtained from the decomposition of $M = 2^3 \times 3^2 \times 5$), and reduces significantly the complexity ($2 \times O(P\ logP)$) of the interconnection network (or barrel shifter), which simplifies the routing problem. This strategy does not imply an increase by $L$ in the size of ROM memories (that hold *SHIFTS* and *ADDRESSES* values). In fact, as the properties of the first subgroups of CNs/BNs to process are known, the periodic properties of DVB-S2 codes allow to automatically calculate the properties of the remaining subgroups[51]. For the architecture to support only $P$ processing units, memory blocks have to be reorganized according to figure 4.4. We can reshape these memory

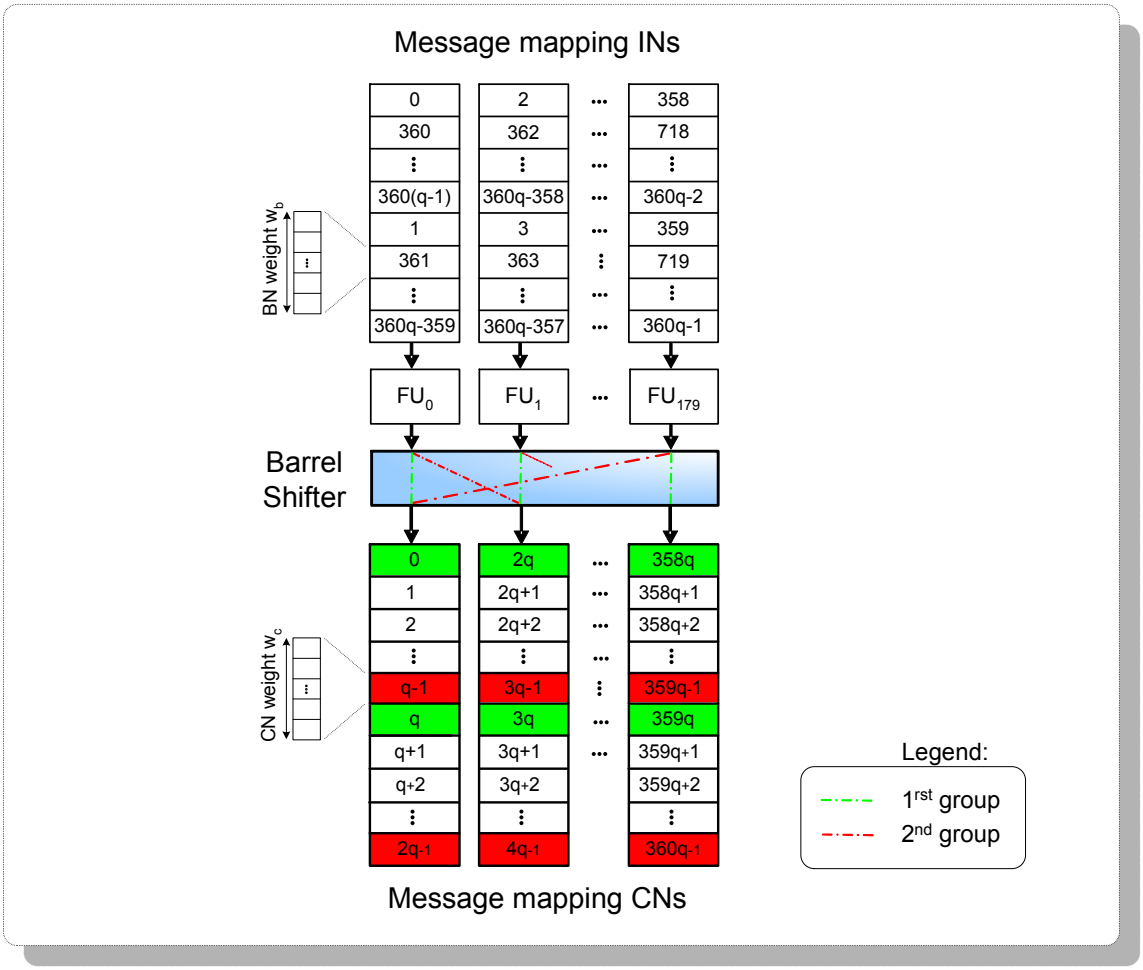


Figure 4.5: Memory organization for the computation of $Lr_{mn}$ and $Lq_{nm}$ messages, automatically adapted for a $L = 2$ factorizable architecture with corresponding $P = 180$ functional units, for the example shown in figure 4.3.

blocks[51] and keep unchanged the size of the system ROM, by computing on the fly the new *SHIFTS* values as a function of those initially stored in the ROM of figure 4.2[51] for a level of parallelism corresponding to $M = 360$. In the configuration[51] shown in figure 4.4, each $FU_i$, with $0 < i < P - 1$, is now responsible for processing information (IN), parity (PN) and check nodes (CN) according to a proper memory mapping and shuffling mechanism conveniently detailed in Appendix A. As we increase $L$, the smaller are the sub-sets of the Tanner graph processed in parallel. Figure 4.5 describes the addressing mechanisms used in the factorizable architecture for $L = 2$, or 180 FUs. The amount of memory is exactly the same as in the previous architecture, but the structure is different. There are less FUs and the corresponding memory word size decreases. But memories have to become higher and thinner in order to hold the same information as before. This new memory configuration will introduce benefits in area occupied by the architecture, as it will be shown later.

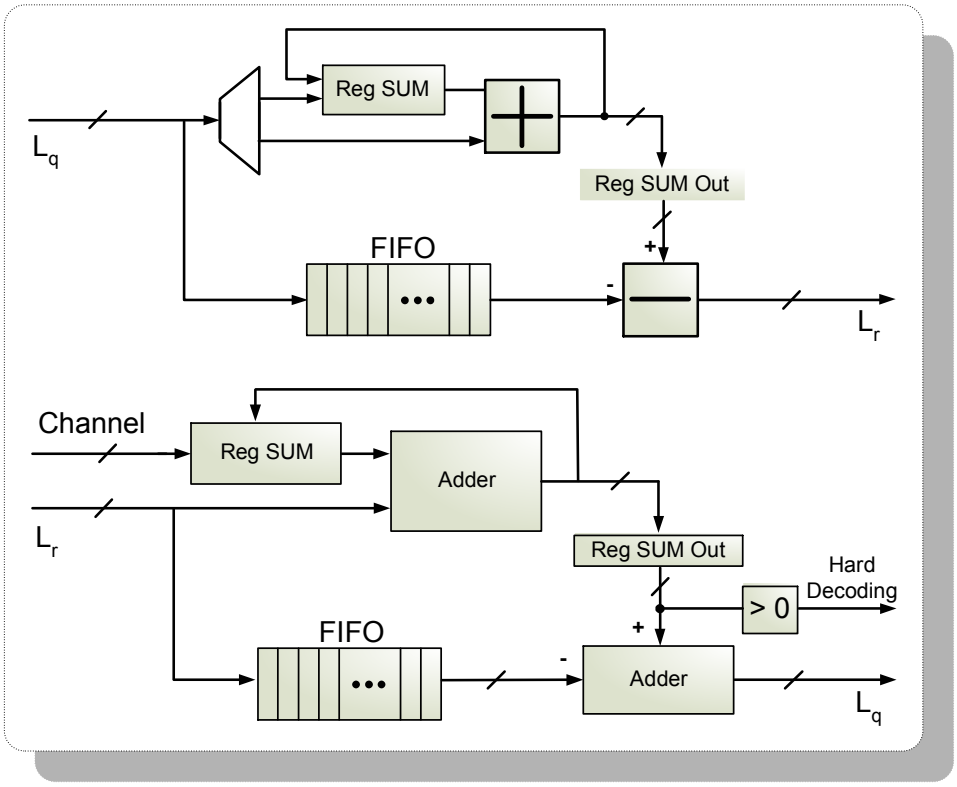


Figure 4.6: Serial architecture for a functional unit that supports both CN and BN processing types[31,49] for the MSA.

### 4.3.1 Functional units

The type of processing performed by each FU is based on the MSA defined in (2.26) to (2.31) of algorithm 2.2. Figure 4.6 shows a FU that supports both BN and CN serial processors and figure 4.7 expands the boxplus and boxminus blocks depicted in figure 4.6. Figure 4.8 depicts a parallel version of the FU. Considering a BN of weight $w_b$, the BN processor can be seen as a black box with $w_b + 1$ inputs, from where it receives the channel information plus $w_b$ CN messages $Lr_{mn}$, sent from the CNs connected to it, and with $w_b + 1$ outputs, through where it communicates the hard decoding decisions and sends the $w_b$ messages $Lq_{nm}$, to the CNs connected to it. The inputs are added on a recursive manner as shown in figure 4.6. The $Reg\ SUM$ register is initialized with the received channel information. The output messages can be obtained in a parallel manner as in figure 4.8, or using a full serial approach as shown in figure 4.6, with a new message being obtained at each clock cycle. This implementation minimizes the hardware com-

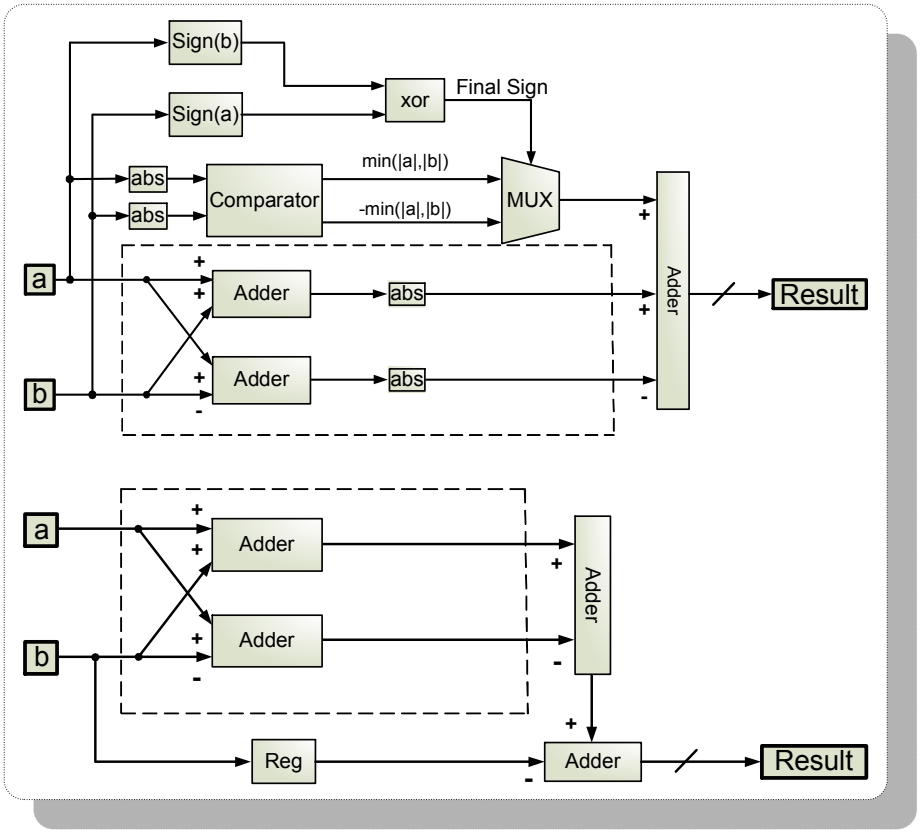


Figure 4.7: Boxplus and boxminus sub-architectures of figure 4.6[31,49].

plexity (measured in terms of number of logic gates) at the cost of a significant increase in processing time (time restrictions could require an increase in the clock frequency). The serial implementation has also the advantage of supporting the processing of a BN of any weight, at the expense of little additional control. In a parallel version, inputs are added

all together, producing the value of the *a posteriori* LLRs $Lq_{nm}$. The message outputs can them be computed simultaneously by just subtracting all entries from the output of the referred adder. This type of implementation (shown in figure 4.8) requires an adder capable of adding $w_b + 1$ inputs, as well as $w_b$ output adders in order to be able of performing the $w_b$ necessary final subtractions. This means that a high number of gates is required to implement just a single processing unit, but presents the advantage of a minimum delay system (able of providing high throughput) that allows lowering the clock frequency, which implies a reduction in power consumption. For the CN serial and parallel processors similar conclusions are achieved and the serial architecture naturally requires less hardware resources than its parallel counterpart[31,49]. The difference is more notorious

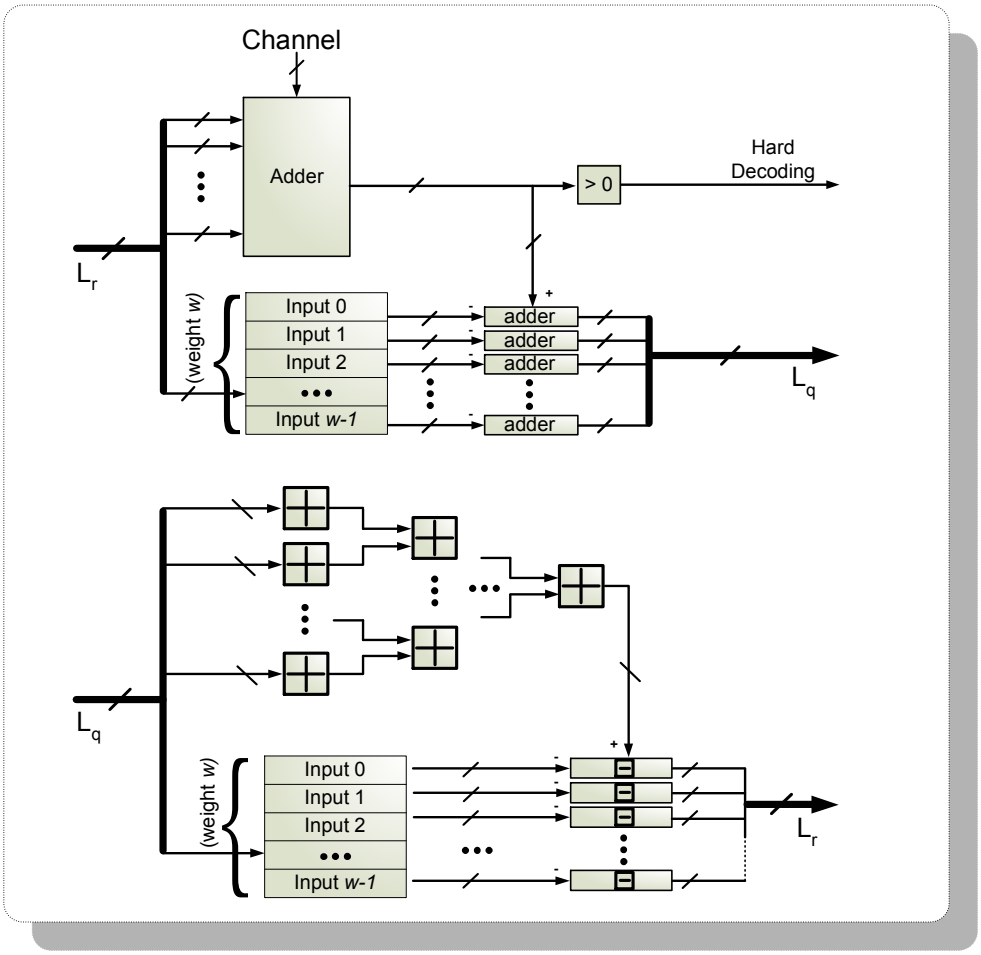


Figure 4.8: Parallel architecture for a functional unit that supports both CN and BN processing types[31,49] for the MSA.

for CN processing units, even for those supporting a small weight $w_c$ as it is the case of this one. Thus, differences in hardware resources required by serial or parallel versions of the FU should be more significant for CNs with higher $w_c$. Boxplus and boxminus operations can both be implemented at the cost of four additions and one comparison, as

shown in figure 4.7. Based on the proposed boxplus and boxminus hardware modules, it is possible to adopt a serial or parallel configuration for the CN processor (similar to the described for BN processor units). Nevertheless, the complexity of the boxplus operation on a parallel implementation requires a boxplus sum chain of all inputs according to the figure 4.8. The advantages of one configuration compared with the other are similar to those mentioned for the BN processor. Nevertheless, it should be noted that the proportion of silicon area occupied by a parallel implementation with respect to a serial implementation, in this case is significantly higher than the one for the BN processor, due to the number of operations involved in the boxplus and boxminus processing. In fact, the number of gates required by the boxplus and boxminus processing units is superior to common add and subtract arithmetic operations.

The fact that only $Lr_{mn}$ messages represented in (2.26) are used in the processing of CNs and, at the same time, only $Lq_{nm}$ messages represented in (2.29) are used in the processing of BNs, allows simplifying the architecture, since memory can be shared by both types of processing (one overlaps the following), as data is discarded at the end of a computation and posterior storing in memory. During the execution of an iteration, the decision of performing BN or CN processing should be performed by the *Control* unit depicted in figures 4.2 and 4.4. The FUs proposed can be efficiently described by using a Hardware Description Language (HDL) library tool developed[31] to automatically generate Register Transfer Level (RTL) code that produces either fully serial or fully parallel synthesizable FU processing blocks[31,49].

The architecture proposed in this section[51] uses functional units based on the full serial approach depicted in figure 4.6, which requires a lower number of gates (and area) at acceptable clock frequency. Also, it supports all different code rates and frame lengths making it fully compliant with the DVB-S2 standard[29]. This solution allows a good compromise between throughput, algorithm complexity and hardware resources. In the next sub-sections we present the obtained experimental results for both FPGA and ASIC devices.

### 4.3.2 Synthesis for FPGA

The architecture of figure 4.4 was synthesized for Virtex-II Pro FPGAs (XC2VPxx) from Xilinx for validation purposes. In all solutions 5 bits are used to represent data (input values and inner messages). Table 4.1 reports the hardware resources used in FPGAs XC2VP30, XC2VP50 and XC2VP100 for 3 different factorization levels, namely $L = \{2, 4, 8\}$, and also the maximum frequency of operation, which can be obtained from:

$$f_{clk\_max} = \frac{1}{(Logic\ Delay + Routing\ Delay)}, \tag{4.1}$$

Table 4.1: Synthesis results[1] for P={45, 90, 180} parallel functional units.

| FPGA | XC2VP30 | | | XC2VP50 | | | XC2VP100 | | |
|---|---|---|---|---|---|---|---|---|---|
| **FUs** | **45** | **90** | **180** | **45** | **90** | **180** | **45** | **90** | **180** |
| L | 8 | 4 | 2 | 8 | 4 | 2 | 8 | 4 | 2 |
| Slices (%) | 87 | 154† | 292† | 50 | 88 | 169† | 27 | 50 | 88 |
| Flip-Flops (%) | 19 | 38 | 75 | 11 | 22 | 43 | 6 | 11 | 23 |
| LUTs (%) | 81 | 142† | 265† | 46 | 80 | 153† | 25 | 46 | 80 |
| BRAM (%) (18 Kbit blocks) | 132† | 144† | 163† | 77 | 84 | 95 | 40 | 44 | 50 |
| Maximum Frequency of Operation (MHz) | 70.8 | – | – | 70.8 | 64.1 | – | 70.8 | 74.0 | 73.2 |
| Logic Delay (ns) | 8.36 | – | – | 8.36 | 8.68 | – | 8.36 | 7.74 | 5.99 |
| Routing Delay (ns) | 5.77 | – | – | 5.77 | 6.93 | – | 5.77 | 5.77 | 7.67 |

where *Logic Delay* and *Routing Delay* indicate maximum obtained values. Synthesis results show that it is necessary to use at least the FPGA XC2VP50 in order to guarantee the minimum memory resources required to implement all code rates and lengths adopted in DVB-S2. However, this particular choice uses nearly only 50% of the FPGA available slices for a 45 FU architecture. From table 4.1 we observe that it is also possible to implement this same architecture on the lower cost FPGA XC2VP30 if we use external memory. For a given frequency of operation $f_{op} < f_{clk\_max}$, the expected throughput is given by:

$$Throughput \geq \frac{frame\_length \times f_{op}}{\left(2 \times W + w_j - 3\right) \times max\_iter \times L}, \tag{4.2}$$

where $W$ represents the number of elements of $\mathbf{H}$ in the compact form as it is represented in annexes B and C of the DVB-S2 standard[29], and $w_j$ is the weight of the IN nodes of the code with weight $j \neq 3$ (see table 2.3 in chapter 2). In the architecture of figure 4.2, a fully serial FU computes a new message per clock cycle and only stops (in bit mode) $w_j - 3$ cycles when switching the processing of IN nodes of weight $w_j$ to nodes of weight $w = 3$. In the architecture of figure 4.4 the number of wasted clock cycles per iteration increases by $L$. The XC2VP100 FPGA allows the implementation of the architecture of figure 4.4 with 180 FUs (a factor $L = 2$), which multiplies the throughput approximately by 4 times when compared to a 45 FU architecture (a factor $L = 8$). Figure 4.9 presents a comparison of the maximum number of decoding iterations for each DVB-S2 LDPC code, that can be performed on a XC2VP100 FPGA, operated at maximum clock frequency $f_{clk\_max}$, and that guarantees the minimum throughput of 90 Mbps for each configuration of the architecture, with $P = \{45, 90, 180\}$ FUs. For example, the normal frame $rate = 1/4$

[1]The symbol † indicates that more than 100% of the device resources are required.

code supports more than 10 iterations for $P = 45$ FUs, nearly 25 iterations for $P = 90$ FUs, and more than 45 iterations for $P = 180$ FUs. By limiting the maximum number of iterations to 15, we can observe in figure 4.10 that the minimum throughput achievable for the 180 FUs architecture is far above the 90 Mbps. This limit is nearly achieved and in some cases even surpassed for DVB-S2 LDPC codes in the architecture based on 90 FUs.

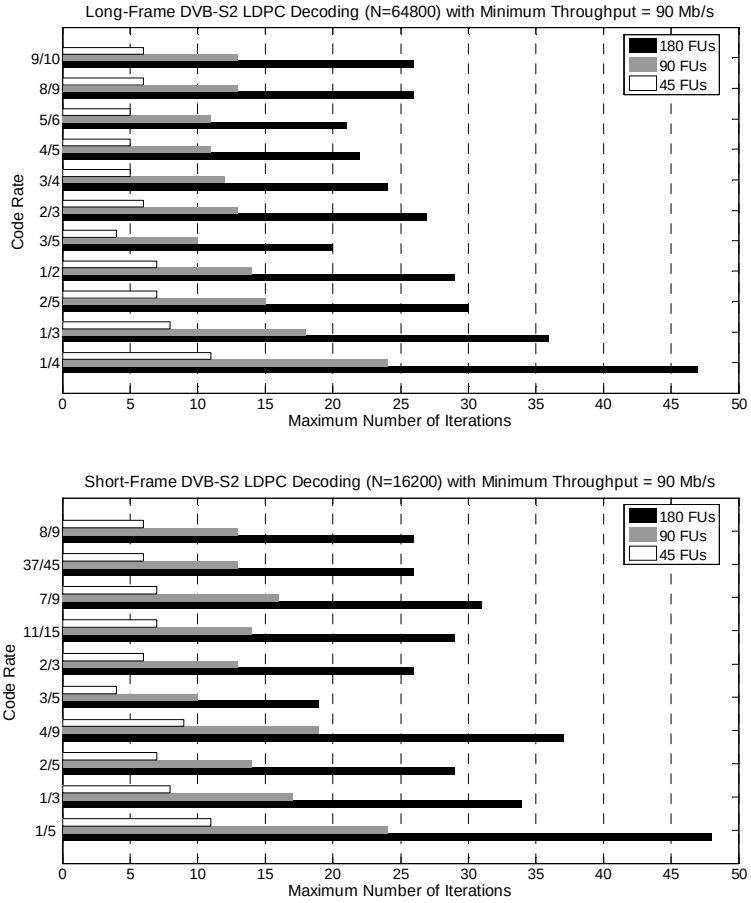


Figure 4.9: Maximum number of iterations[51] that can be performed with $P = \{45, 90, 180\}$ functional units[2], for a throughput of 90 Mbps on a XC2VP100 FPGA.

Comparing throughput for both frame lengths in figure 4.10, we realize that they resemble similar, which on a first analysis may seem incorrect due to the fact that normal frame (64800 bits) codes have a length $4\times$ longer than short frame (16200 bits) ones. The reason for this lies in the fact that throughput represents a relation between the number of decoded bits and the time necessary to perform that operation. As the number of bits is $4\times$ higher, the time spent doing that processing is equally higher because the number of edges to process increases approximately by the same amount. Although processing time is not exactly $4\times$ higher, its value is very close to that number. This is the reason

why both throughputs approximate that much.

In the last couple of years, FPGA technology evolved and the density of new devices is now superior to the one supported by Virtex-II Pro FPGAs. Also, new FPGA technologies support reduced internal delays which allows to achieve higher frequencies of operation. Any of the 3 architectures proposed based on figure 4.4 can be easily supported, for example, by a recent Virtex-6 family device.

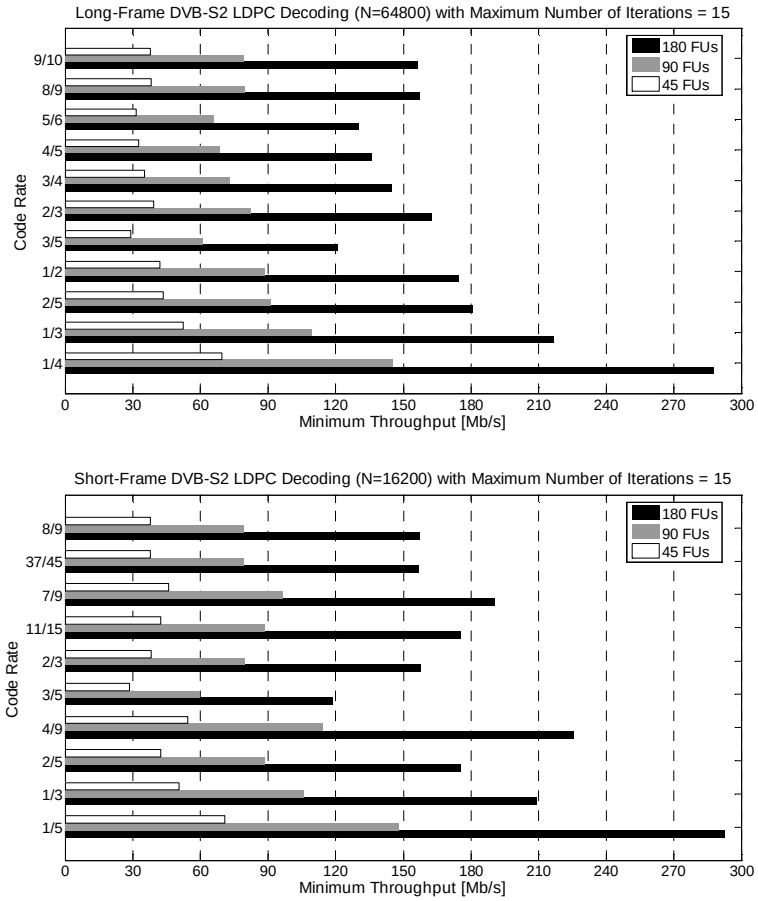


Figure 4.10: Minimum throughput achievable[51] on a XC2VP100 FPGA for $P = \{45, 90, 180\}$ functional units[2]. The maximum number of iterations is 15.

### 4.3.3 Synthesis for ASIC

Synthesizing in ASIC technology the architecture with $P = \{45, 90, 180, 360\}$ FUs aims at finding the one which produces better power consumption and area results, while simultaneously supporting the minimum throughput requirements. Any of the solutions here reported supports short and normal frame lengths, even though the architecture is

[2] As mentioned in 2.6.2 regarding table 2.2, five DVB-S2 short frame codes don't have constant $w_c$ weights. For this reason, five short frame rates in figures 4.9 and 4.10 are approximations of those mentioned in table 2.2 (for example, $rate = 4/9$ in the figures represents code with $rate = 1/2$ in the table).

dimensioned for the worst case. The architecture was synthesized for a 90nm 8 Metal CMOS process. All solutions use 5-bit precision to represent data messages.

The assessment of synthesis results for the proposed architecture presented in figure 4.11 shows that it is extremely competitive, when compared with state-of-the-art solutions[18,27,70,95,133]. Figure 4.11 shows that the total area of the circuit is equal to 21.2mm$^2$ for a 360 FU solution, while the architecture with 45 FUs can be obtained with only 7.7mm$^2$. A 90nm technology is used in[27,133], producing, respectively, an area of 4.1mm$^2$[27] and 9.6mm$^2$[133]. Results for similar architectures but for other technologies have been presented: for example, for a 0.13$\mu$m technology, an area of 22.7mm$^2$ has been achieved[70], while[18] presents 3.9mm$^2$ for a 65nm technology, and[95] achieves 6.03mm$^2$ for the same technology.

Although some of them claim to occupy smaller die areas[18,27], our solution[51] supports both frame lengths, while they only support the normal frame mode. Also, considering that our design is based on a 90nm process, it compares favorably in terms of area against[18,95] which have similar areas but use a 65nm technology. The new architecture here proposed based on 45 FU shows an LDPC decoder circuit with smaller area occupied than those reported in[70] and[133]. For power consumption purposes, perhaps most important is the fact that our architectures work with an inferior maximum frequency of operation (100 and 200MHz) than those just reported above in state-of-the-art solutions[18,70,133] (the maximum frequency of operation is not indicated in[27], only area and throughput are mentioned as described in the previous paragraph). In the new approach here proposed 45% of the circuit works at 200 MHz, while the remaining 55% work at

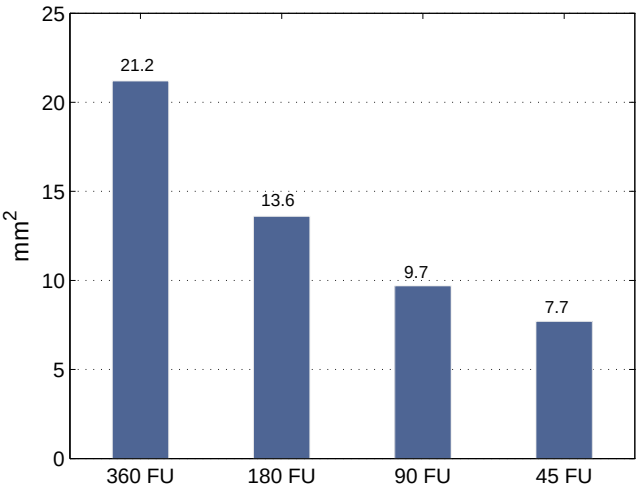


Figure 4.11: Area results for ASIC architectures with $P = \{360, 180, 90, 45\}$ functional units.

100 MHz. Final circuit validation has been performed by place & route using 8 metal layers. No time, physical or routing violations were reported, so in this stage the 45 FUs architecture is physically validated.

At this point and due to the configurable nature of the architecture, we found room to perform some extra optimizations. For practical purposes, and since it presented good results for throughput and area, we adopted the configuration based on the 45 FUs as the basis solution to be optimized. The next section shows where there is still room to perform optimizations and how to achieve that.

## 4.4 Optimizing RAM memory design for ASIC

The presented architecture has been described in RTL, for $P = \{45, 90, 180, 360\}$ FUs. Figure 4.4 shows that all 4 configurations use exactly the same amount of memory, though rearranged with different widths and heights (memory positions). The smaller the number of $P$ functional units adopted, the higher and thicker block memories become. The complete type and amount of memories necessary for such design are[51]:

- *Message* memory holds internal messages calculated during each iteration of the decoding process for all information nodes (IN). *Message* memory width (word length) is given by $w_a = number\ of\ FU \times message\ precision$ which in this case is $w_a = M \times 5/L$ bits. The height $h_{mm}$ can be obtained for the worst case scenario (code with $rate = 3/5$ obtained from table 2.3, where the number of edges is maximum and $(N - K) = 25920$). In this case, $h_{mm} = L \times q \times (w_c - 2)$;

- *Channel* memory stores IN and PN data incoming from the channel. This memory has the same word length as *Message* memory $(w_{ch} = w_a)$, and the height is given by $h_{ch} = L \times 64800/M$, representing the worst case (normal frame);

- *PN message* memory holds messages calculated during each iteration, which are associated only with parity bits. It has the same width as *Channel* memory and the height $h_{PN}$ is obtained for worst case scenario (code with $rate = 1/4$) from table 2.3, which turns $h_{PN} = q = 135$ (only one 5-bit message per each parity bit has to be stored);

- *Hard decoding* memory holds IN and PN data obtained in the hard decoding phase of processing (1-bit per FU), described by (2.30) and (2.31). The width of this memory is given by the number $M$ of FUs of the architecture. The height $h_{HD}$ is obtained from $h_{HD} = L \times 64800/M$.

Table 4.2 summarizes the required width and height (mem. pos.) of all memories used in the 4 synthesized configurations. Unfortunately, sometimes RAM memory libraries of

Table 4.2: Required RAM memory size for each configuration.

| Type of RAM memory | 360 FU (mem. pos. × width) | 180 FU (mem. pos. × width) | 90 FU (mem. pos. × width) | 45 FU (mem. pos. × width) |
|---|---|---|---|---|
| Message | $648 \times 1800$ | $1296 \times 900$ | $2592 \times 450$ | $5184 \times 225$ |
| Channel (IN+PN) | $180 \times 1800$ | $360 \times 900$ | $710 \times 450$ | $1440 \times 225$ |
| PN message | $135 \times 1800$ | $270 \times 900$ | $540 \times 450$ | $1080 \times 225$ |
| Hard decoding (IN+PN) | $180 \times 360$ | $360 \times 180$ | $720 \times 90$ | $1440 \times 45$ |

Table 4.3: Physical (real) RAM memory size for each configuration.

| Type of RAM memory | 360 FU (mem. pos. × width) | 180 FU (mem. pos. × width) | 90 FU (mem. pos. × width) | 45 FU (mem. pos. × width) |
|---|---|---|---|---|
| Message | $2^{10} \times 1800$ | $2^{11} \times 900$ | $2^{12} \times 450$ | $2^{13} \times 225$ |
| Channel (IN+PN) | $2^{8} \times 1800$ | $2^{9} \times 900$ | $2^{10} \times 450$ | $2^{11} \times 225$ |
| PN message | $2^{8} \times 1800$ | $2^{9} \times 900$ | $2^{10} \times 450$ | $2^{11} \times 225$ |
| Hard decoding (IN+PN) | $2^{8} \times 360$ | $2^{9} \times 180$ | $2^{10} \times 90$ | $2^{11} \times 45$ |

standard cells do not exactly support the specified heights (memory positions) requested in table 4.2, but rather standard dimensions which are usually a power of 2, as shown in table 4.3. The area results obtained in the synthesis process and shown in figure 4.11, allowed other interesting conclusions. Memories occupy nearly 97% of the circuits' total area. The remaining part of the circuit is occupied by the barrel shifter, functional units and control logic. The fact that in the 4 configurations the areas are different was also a surprise. Figure 4.11 shows these differences. As mentioned before and depicted from figure 4.4, the total amount of memory is the same for all designs. If we realize that they occupy a significant area of the design, we conclude that their differences should be minimal. To analyze these differences, we first need to understand how memories are generated and the RAM generator limitations. The architecture implemented in RTL uses memories with large width (word length) and height (number of words, or memory positions).

The RAM generator used can create memories that support the requested number of words, but the maximum word width is limited to 64 bits, which is far bellow from architectural needs. To overcome this problem, $B$ RAMs were concatenated until the required memory width has been achieved, as shown in figure 4.12. Each RAM memory has its own internal control logic which can address data in the working clock cycle, with its own antenna diode protection, testing mechanisms, power rings, etc. As represented in figure 4.12, for large words more memories are necessary, which replicates control hard-

ware and increases silicon area. A practical example with real areas can be addressed for the *Message* RAMs for 360 and 45 FU, to give a better perspective of the problem. In the first configuration, the *Message* RAM width is 1800 bits (5 bits per message $\times$ 360 FU) by $2^{10}$ addresses (height). For the second configuration, the RAM width is 225 bits (5 bits per

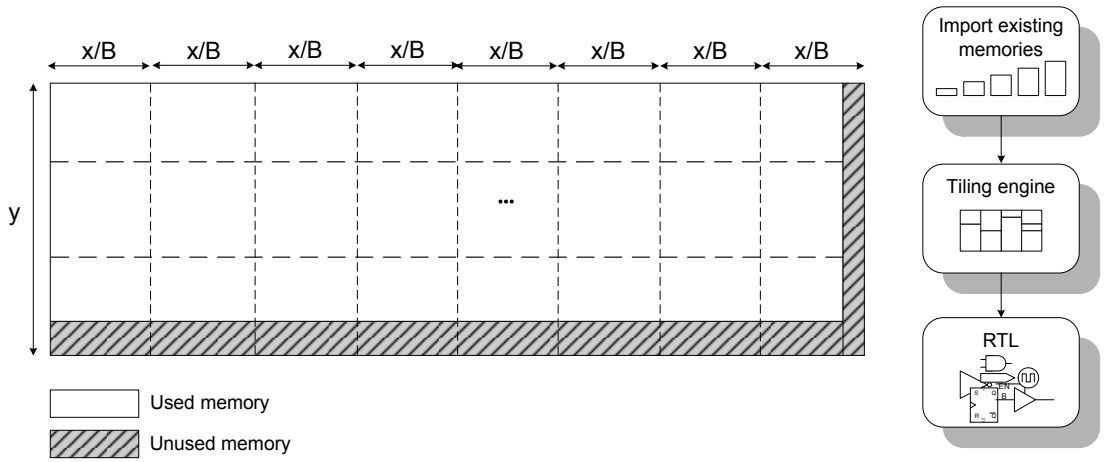


Figure 4.12: Tiling of an $y$ (height) by $x$ (width) RAM memory layout.

message $\times$ 45 FU) by $2^{13}$ addresses. Both *Message* memory blocks have the same capacity ($2^{10} \times$ 1800 bits), however the area used by the wider 360 FU *Message* memory is 6.2mm$^2$, while the thiner 45 FU memory occupies only 3.2mm$^2$. These memories were created by concatenating $B$ banks of 45 bit RAMs, as illustrated in figure 4.12. For 360 FUs, 40 instantiations are necessary ($40 \times 45$ bits $= 1800$ bits) while for 45 FUs only 5 instantiations are needed ($5 \times 45$ bits $= 225$ bits). Whatever the factorization level $L$ adopted, the architecture will still have to support a Very Long Data Word (VLDW) bus. This property is illustrated in detail in figure 4.13, where the VLDW bus implies a complex routing of the wiring associated with it. This introduces additional complexity managing the floor planning and routing process (in the place & route phase of the design). Nevertheless, the architecture with 45 FUs minimizes this problem, moving its complexity to a dimension where it can be more easily tractable.

As described before, the configuration with 45 FUs occupies less area than the other three, but time closure can still become a real problem in this case. To prove that this architecture can be a valid solution, synthesis was performed for 100 MHz without any timing violations.

### 4.4.1 Minimal RAM memory configuration

From all the designs, in figure 4.11 the 45 FU's architecture occupies the smallest area. Comparing tables 4.2 and 4.3, we notice that due to hardware restrictions, there is a portion of the RAM which is never used. For instance, in the 45 FU architecture *Message*

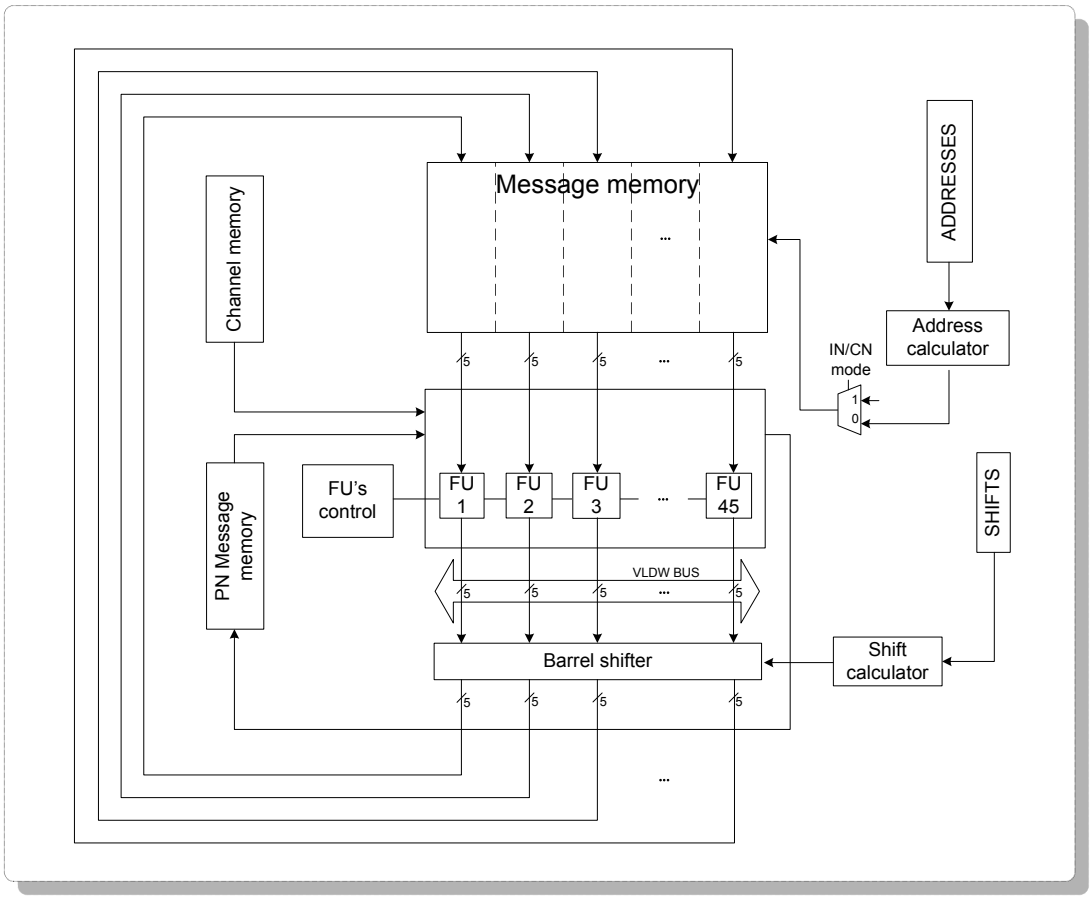


Figure 4.13: Optimized 45 functional units architecture.

RAM needs $225 \times 5184 = 1166400$ bits. But physically, we need $225 \times 2^{13} = 1843200$ bits, resulting in $1843200 - 1166400 = 676800$ unused bits, about 37%. This fact occurs in all memories and for all architectures and it can be minimized using decompositions in powers of 2. From the previous example, we notice that the RAM height can be decomposed into $5184 = 2^{12} + 2^{10} + 2^6$. To minimize the unused area, we can use the concatenation of smaller RAMs, obtaining a RAM depth near the specific architecture needs. Unfortunately, after generating and concatenating the decomposed RAMs, we realize that the area increases nearly 30% (instead of decreasing), which can be justified by the additional number of internal RAM hardware controllers incorporated into the design. This factor is the major responsible for increasing the area. Therefore, decomposing RAMs in powers of 2 is not an interesting solution. However, joining together several blocks of memory that operate in different time slots into one bigger common block of memory can become a more efficient solution. First, we need to understand in more detail the decoder architecture with 45 FUs as depicted in figure 4.13. In *Message* memory of figure 4.13, each 5 bits of the word are directly connected to each FU, and consequently only one main memory block is used. This way, only one (read or write) address is needed to transfer

data to/from all FUs. This memory must be a dual port RAM, in order to allow reading and writing during the same clock cycle. However, due to memory generator limitations in the tools used, this type of memory was not possible to achieve. The solution adopted consists of creating a memory operating at twice the circuit's frequency: in the first cycle it performs read operations; and in the second one it writes data to memory. The LDPC decoder input data comes from an external de-interleaver module that does not work at the same clock frequency as the LDPC decoder. To guarantee that no information is lost, all messages delivered to the decoder must be stored in *Channel* memory. The *PN message* memory is equally stored on a dual port RAM which, as before, due to memory generator limitations was converted into a single RAM working at twice of the initial frequency. In the beginning, this memory is loaded with zeros, and then it will hold data associated with parity bits necessary in each iteration of the decoding process. Shift and address data are loaded from an outside source into *SHIFTS* and *ADDRESSES* memories. The values stored into these memories depend on the code rate and must be previously calculated.

### 4.4.2 ASIC synthesis results for an optimized 45 FUs architecture

The floorplan with placed std cells, RAM memory and pins is presented in figure 4.15 for the optimized 45 FUs design. To translate the mentioned optimizations into area requirements, the new circuit has been synthesized for the same 90nm technology used in the original synthesis. The optimizations reduce the area of the smallest original architecture nearly 20%. Synthesis results for the optimized 45 FUs version and also for the previ-

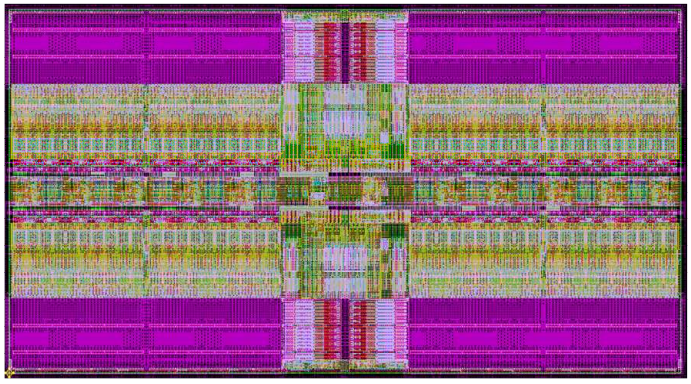

Figure 4.14: Layout of a memory with 10-bit width $\times$10 lines of bus address.

ously mentioned $P = \{360, 180, 90, 45\}$ FUs based architectures, are listed in table 4.4 and the areas are also compared in figure 4.16. They range from 21.2mm$^2$ to 6.2mm$^2$, with corresponding levels of power consumption of, respectively, 290mW down to 85mW. The 45 FU's architecture optimized with an efficient RAM memory reshape presents a total area

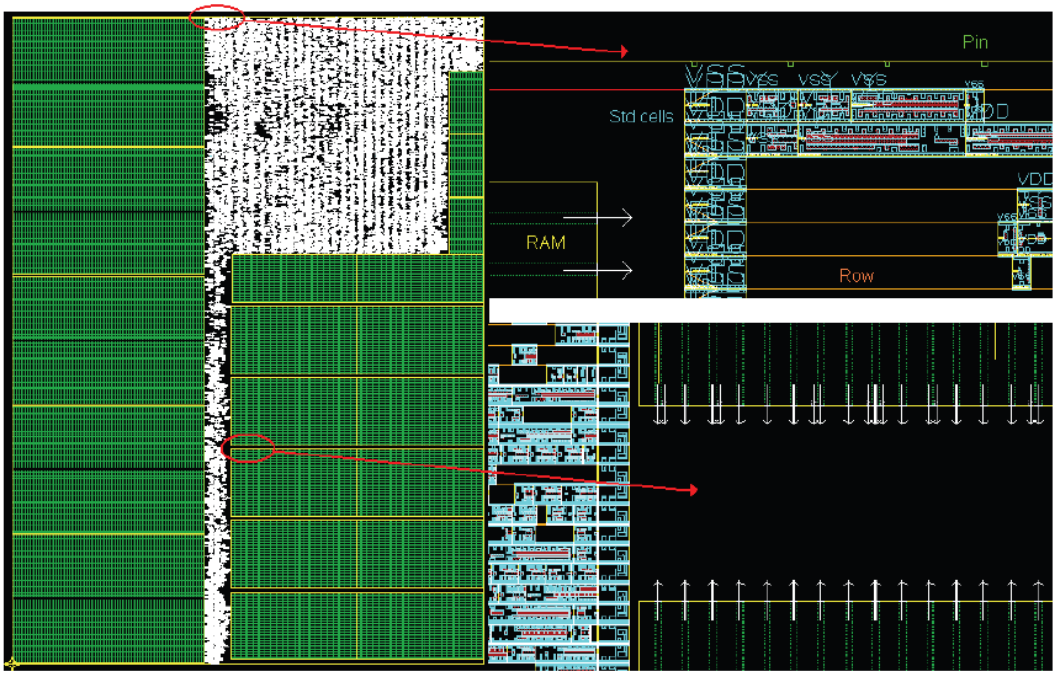


Figure 4.15: Floorplan with placed std cells, RAM and pins for the optimized 45 functional units design.

of 6.2mm$^2$ and 85mW of power consumed at a typical voltage of 1.1V (and maximum 1.32V). The estimation of power consumption was performed by considering typical values for current, voltage and temperature. This approach was followed because extreme working conditions usually do not occur, or if they do, it is during small periods of time which does not negatively affect the design.

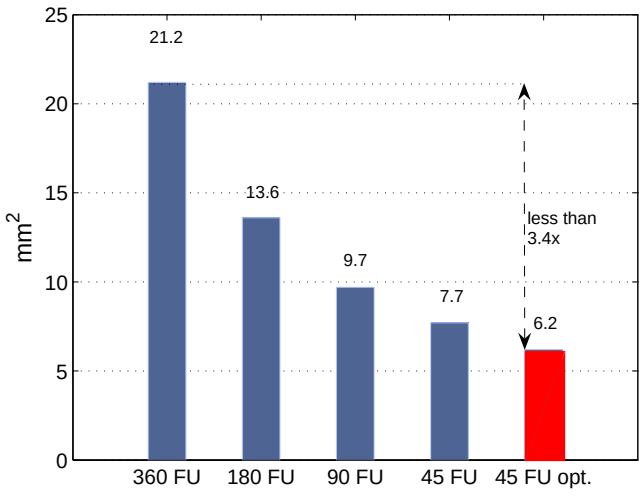


Figure 4.16: Area results comparing the optimized 45 FUs design.

Table 4.4: ASIC synthesis results for $P = \{45, 90, 180, 360\}$ parallel functional units and for an optimized 45 functional units architecture.

| | 360 | 180 | 90 | 45 | 45-optim. |
|---|---|---|---|---|---|
| Technology | 90nm | 90nm | 90nm | 90nm | 90nm |
| Max. voltage | 1.32 v | 1.32 v | 1.32 v | 1.32 v | 1.32 v |
| Typ. voltage | 1.1 v | 1.1 v | 1.1 v | 1.1 v | 1.1 v |
| Min. voltage | 1.08 v | 1.08 v | 1.08 v | 1.08 v | 1.08 v |
| Max. temperature | 125°C | 125°C | 125°C | 125°C | 125°C |
| Typ. temperature | 25°C | 25°C | 25°C | 25°C | 25°C |
| Min. temperature | -40°C | -40°C | -40°C | -40°C | -40°C |
| Max. frequency | 100MHz | 100MHz | 100MHz | 100MHz | 100MHz |
| Power | 290mW | 185mW | 130mW | 105mW | 85mW |
| Current | 260mA | 170mA | 120mA | 95mA | 75mA |
| Gate count | 9.6 Mgates | 6.2 Mgates | 4.4 Mgates | 3.5 Mgates | 2.8 Mgates |
| Area | 21.2mm$^2$ | 13.6mm$^2$ | 9.7mm$^2$ | 7.7mm$^2$ | 6.2mm$^2$ |

Figure 4.17 facilitates assessing the dimension and complexity of this design (designs developed for common applications in the industry of semiconductors are typically much smaller). For example, the size of the circuit for 360 FU is comparable, in terms of the number of gates, with an Intel® Pentium® 4 processor that requires approximately 55 millions of transistors[66] or the equivalent 14 millions of gates[73]. This LDPC decoder design produces a chip with 9.6 millions of gates, as it is also described in table 4.4. Even the optimized architecture with 45 FU requests 2.8 millions of gates, a value which is very

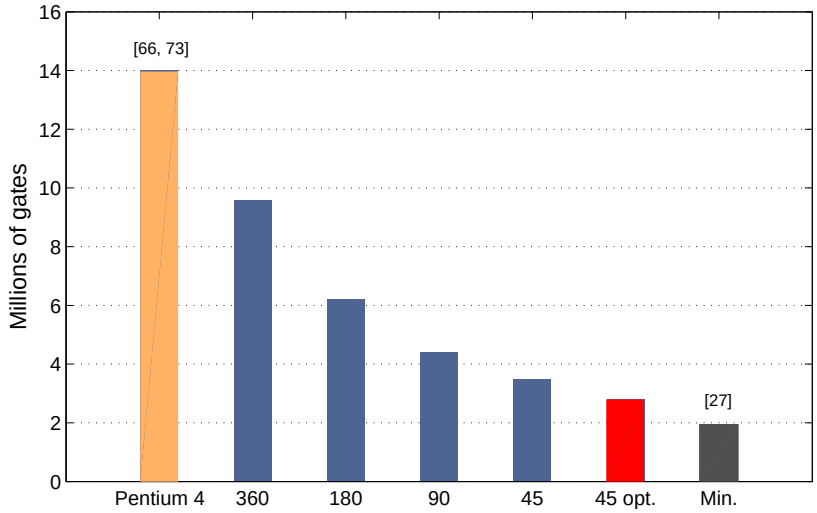


Figure 4.17: Reference designs for gate count comparison of the ASIC based architectures proposed, with 360, 180, 90, 45 and 45-optimized functional units.

inferior to the former but still quite significant, and very close to an architecture described in the literature[27] that presents the minimal number of 2 millions of gates for an LDPC decoder (for the best of our knowledge). This architecture[27] uses a 90nm CMOS technology where each NAND2 gate occupies approximately 2.1$\mu$m$^2$. To estimate the total number of gates of this design, we divided the total area of 4.1mm$^2$ [27] per NAND2 gate area to obtain an approximate value of 1.95 millions of gates. This value is also presented as the rightest most element of figure 4.17.

Again, final circuit validation has been performed by place & route, using 8 metal layers. No time, physical or routing violations were reported, so in this stage the 45 FU optimized architecture is physically validated and ready to be implemented in silicon. To accommodate place & route, it should be noted that, based on previous design experience, we estimated an increase of area equivalent to 20% (after final validation, we realized that it could have been approximately 19%), which corresponds to a global area of 7.4mm$^2$ for the optimized 45 FUs architecture.

## 4.5 Summary

This chapter addresses the generalization of a state-of-the-art $M$-kernel parallel structure for LDPC-IRA DVB-S2 decoding, for any integer factor of $M = 360$. The proposed method adopts a partitioned processing of sub-blocks of the Tanner graph, that keeps unchanged the efficient message memory mapping structure without addressing unnecessary overheads. This architecture proves to be flexible and easily configurable according to the decoder constraints and represents a trade-off between silicon area and decoder throughput. Synthesis results for FPGA devices show that a complete LDPC-IRA decoder for DVB-S2 based on an 180 FUs architecture can be configured on a low-cost XC2VP100 FPGA from Xilinx.

When synthesizing for ASIC technology the limitations in terms of number of gates naturally no longer become important, and larger architectures could be tested. Nevertheless, this approach aims at finding microcircuits with the smallest possible die area and lower power consumption, that still ensure minimum throughput requirements for all DVB-S2 codes. Under this context, 5 configuration designs with different number of processing units have been implemented. They range from 360 to 45 FUs which represents, respectively, an equivalent area occupied of 21.2 and 7.7mm$^2$. Although the process of generating RAM memories imposes constraints, the investigation carried out under this thesis allowed achieving several interesting conclusions, that were applied into the design of the LDPC decoder circuit. Re-dimensioning and rearrangements in the order how memory blocks are grouped together allowed reducing the global area of the circuit

to a value as low as 6.2mm$^2$. This shows the competitiveness of the architecture when compared with state-of-the-art solutions. Moreover, the maximum frequency of operation of the design here proposed is smaller than those reported by competitors, which justifies the low levels of power consumption achieved.

*"If you were plowing a field, which would you rather use? Two strong oxen or 1024 chickens?"*

*Seymour Cray*

# 5

# LDPC decoding on multi- and many-core architectures

## Contents

In chapter 3 we analyzed the decoding complexity of the Sum-Product Algorithm (SPA) and Min-Sum Algorithm (MSA). The required number of operations depicted in figures 3.1 and 3.2 shows that high performance parallel computing should be considered to execute such intensive processing. To achieve real-time computation they can be performed by using dedicated Very Large Scale Integration (VLSI) processors based on Field Programmable Gate Array (FPGA) or Application Specific Integrated Circuits (ASIC), as described in chapter 4. However, hardware represents non-flexible and non-scalable dedicated solutions[137] [27], that require many resources and long development times. More flexible solutions for Low-Density Parity-Check (LDPC) decoding using Digital Signal Processor (DSP)s[53] or Software Defined Radio (SDR) programmable hardware platforms[108] have already been proposed.

In this chapter we exploit the advances in powerful low-cost and widespread parallel computing architectures for achieving high performance LDPC decoding. Although general-purpose x86 multi-cores show modest performances for achieving real-time LDPC decoding for most of the applications, they allowed to show that the solutions here proposed are scalable. From the set of multi-core based architectures analyzed in this thesis, the Cell Broadband Engine (Cell/B.E.) and Graphics Processing Units (GPU) emerge as flexible highly parallel architectures. Because they represent programmable solutions, they offer the possibility of increased productivity at low-cost, namely to impact the performance of demanding algorithms such as the LDPC decoders analyzed in this thesis.

## 5.1 Multi-core architectures and parallel programming technologies

Until a few years ago, general-purpose single-core based processors were unable to handle the computational complexity of LDPC codes in real-time. The throughputs achieved using this type of processors were below the hundreds of Kbps[32], which does not represent a solution powerful enough to be considered as a consistent alternative for LDPC decoding. But the evolution of processor architectures supported on the scaling up of the technology according to Moore's Law has lead to the actual multi-core superscalar processors. Mainly due to frequency of operation, power consumption and heat dissipation constraints, the trade-off speed/power, a fundamental relation of multi-core processors, showed that it would be more efficient to use two or more processing cores on a single processor, rather than to continue increasing the frequency of operation. The integration of multiple cores into a single chip has become the new trend to increase processor performance. These architectures[12] require the exploitation of parallelism to achieve high performance.

Table 5.1: Overview of recent multi-core platforms[1].

| Platform | # cores | Performance | Mem. band. | Homog. |
|---|---|---|---|---|
| **Intel 6-core Dunnington [65]** | 6 | $<$ 51 GFLOPS[2] | 1.07 GT/s[3] | Yes |
| **Intel Xeon Nehalem 2x-Quad** | 8 | – | – | Yes |
| **Tilera 64-core Tile64 [119]** | 64 | 443 GOPS | $4TB/s$ | Yes |
| **STI Cell/B.E. [24]** | 9 | $>$ 200 GFLOPS | 204.8 GB/s | No |
| **NVIDIA 8800 GTX [96]** | 128 | 350 GFLOPS | 86.4 GB/s | Yes |
| **ATI/AMD Radeon HD 2900 XT [5]** | 320 | 475 GFLOPS | 105.6 GB/s | Yes |

Multi-core architectures have evolved from dual or quad-core to tera-scale systems, supporting multi-threading, a powerful technique to hide memory latency, while at the same time provide larger Single Instruction Multiple Data (SIMD) units for vector processing[48]. New parallel programming models and programming tools have been developed for multi-core architectures, namely common programming models that allow the user to program once and execute on multiple platforms, such as RapidMind[105]. Common general-purpose multi-core processors replicate a single core in a homogeneous way, typically with a x86 instruction set, and provide shared memory hardware mechanisms. They support multi-threading and share data at a certain level of the memory hierarchy, and can be programmed at a high level by using different software technologies[71], such as OpenMP[21] which provides an effective and relatively straightforward approach for programming general-purpose multi-cores.

Presently, companies like Intel, Tilera, IBM, NVIDIA or ATI/AMD are manufacturing systems based on multi-core processors with typically from 2 superscalar cores up to a few hundreds of simple cores. Table 5.1 presents some examples of architectures that incorporate multiple cores. The Dunnington processor[65] is a variant of the Intel Xeon family (also known as Penryn processor) with a peak performance that does not allow to achieve real-time LDPC decoding for the most demanding cases (see figure 3.2). The distributed nature of the Tilera 64-cores processor Tile64[119] is supported by multiple x-y orthogonal buses, an efficient mesh that allows each core to communicate in parallel along four different directions thus producing a very interesting aggregate bandwidth of 4 TB/s. Nonetheless, it represents an expensive solution where important parameters, such as the number of sustained floating-point operations per second, remain unknown.

[1] All information indicated in table 5.1 represents peak performance values. The values indicated in GFLOPS (floating-point operations) refer to single-precision operations. The omitted values were unavailable.

[2] The performance value for the Intel 6-core Dunnington processor indicated in table 5.1 represents an upper bound estimate. The peak performance values were not available.

[3] GT/s represents Giga Transfer (GT) per second; the value is provided by the manufacturer.

On the other hand, GPUs from NVIDIA[96] or ATI/AMD[5] have hundreds of cores and a shared memory hierarchy which provide high computational power that can be exploited efficiently to other types of processing not only limited to graphics processing.

Mainly due to demands for visualization technology in the industry of games, the performance of GPUs has undergone increasing performances over the last decade. With many cores driven by a considerable memory bandwidth, recent GPUs are targeted for compute-intensive, multi-threaded, highly-parallel computation. Researchers in high performance computing fields are applying their huge computational power to general purpose applications (General Purpose Computing on GPUs (GPGPU))[15,37,45,54,98], allowing higher levels of performance in Personal Computer (PC) systems[4,72]. Moreover, a cluster-based approach using PCs with high performance GPUs has been already reported[42]. Owens *et al.* summarize the state-of-the-art and describe the latest advances performed by the scientific community in GPGPU[99]. At the compiler level, Buck *et al.* introduced important improvements, such as the extensions to the C language known as Brook for GPUs[20], which allows to perform general purpose computation on programmable graphics hardware. 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 interfaces and tools[89], such as the Compute Unified Device Architecture (CUDA) from NVIDIA[97], the Close to the metal (CTM) interface (replaced by the ATI Stream SDK) from AMD/ATI[6], or the Caravela platform[126] have been developed. CUDA and CTM represent dedicated programming tools for specific GPUs with a strong potential to improve the computational efficiency[47]. CUDA provides a new hardware and software architecture for managing computations on GPUs[22,57,80,130] with the Tesla architecture from NVIDIA. The Caravela platform[125,126] holds the interesting property of being the first generalist and powerful programming tool that operates independently of the operating system and GPU manufacturer. Recently standardized, the OpenCL framework[55] allows to write parallel programs for execution across heterogenous platforms that include CPUs, GPUs and other types of processors.

Also pushed by audiovisual needs in the industry of games, emerged the Sony, Toshiba and IBM (STI) Cell/B.E.[24] listed in table 5.1. It is characterized by a heterogeneous vectorized Single Instruction Multiple Data (SIMD) multi-core architecture composed by one main PowerPC Processor Element (PPE) that communicates very efficiently[2] with eight Synergistic Processor Element (SPE)s that operate without sharing memory. Each SPE holds a small amount of local memory for data and code, which is conveniently exploited by a vectorized SIMD oriented architecture[60]. Data transfers between PPE and SPE are performed efficiently by using Direct Memory Access (DMA) mechanisms. Driven

Table 5.2: Overview of software technologies for multi-cores[71].

| Software Technology | Corporation | Target architecture | Language/ Library |
|---|---|---|---|
| **CUDA** | NVIDIA | NVIDIA GPU | Language (C) |
| **BROOK+** | AMD/ATI | AMD/ATI GPU | Language (C) |
| **Cell SDK** | IBM | Cell/B.E. | Library (C) |
| **OpenMP** | ARB Group | multi-core CPU | Library (C, FORTRAN) |
| **OpenCL** | Stand. Khronos Group | GPU, CPU, FPGA | Language (C) |
| **PCT** | MATHWORKS | multi-core CPU | Language (Matlab) |

by a considerable memory bandwidth, the Cell/B.E. processor powers IBM Cell Blade servers, and supports also the PlayStation3 (PS3) platform. The PS3 can be operated with Linux-based operating systems, thus it can be used beyond gaming[103], providing low-cost high-performance professional computing platforms very attractive for demanding scientific applications[3,76].

Table 5.2 presents a subset of the most important technologies used in the programming of multi-core platforms[71]. It shows some of their main properties, namely the programming language that each parallel platform supports. The C language emerges as a commonly supported language. Other programming languages such as C++, FORTRAN, PYTHON or JAVA are also supported by some multi-core architectures and parallel programming models. At the same time, these architectures support data-parallelism and most of them also support task-parallelism.

From the set of architectures based on multi-core processors presented in table 5.1, the low-cost and worldwide disseminated Cell/B.E. and GPUs supported by CUDA are flexible highly parallel architectures which support data and thread levels of parallelism, and can be efficiently programmed using the conventional C language (see table 5.2) to perform LDPC decoding. Also, common general-purpose x86 multi-core processors, supported by an increasing number of cores, can address scalable implementations of parallel algorithms in future versions of the architectures. For all these reasons, they were selected in this thesis as the processing platforms to develop and implement programmable LDPC decoders.

The GPU is not an autonomous processing system. A very important aspect is related with the need of transferring data from the host (Central Processing Unit (CPU)) to the device (GPU), and in the opposite direction at the end of processing. The computing platforms used in this test are PCs that showed peak data transfers of approximately 1.2 GByte/s, experimentally measured between the host and the device connected to the

workstation through a Peripheral Component Interconnect Express (PCIe) bus.

## 5.2 Parallel programming strategies and platforms

The two types of intra-codeword (Definition 3.1) and multi-codeword (Definition 3.2) parallelism, defined in subsection 3.3, can be applied differently to distinct parallel computing architectures. In this thesis we exploit both concepts and show how they can be efficiently applied, depending on the target architecture. General-purpose x86 multi-cores can make use of them separately, while they can be combined to achieve better performance in other architectures, such as in GPUs and the Cell/B.E., as described in the next subsections.

### 5.2.1 General-purpose x86 multi-cores

The number of cores of general-purpose x86 multi-cores typically ranges from 2 to 16 and includes different levels of hardware-managed cache memory[74]. In these architectures memory cache coherency mechanisms are 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. These processors can be programmed by using POSIX Threads (pthreads) or, at a higher level, the OpenMP programming interface[21].

Applications that exploit the parallel processing of matrices in general-purpose x86 multi-cores using OpenMP can be found in the literature[8,22]. For example, they describe the multiplication of matrices[8] and, in particular, of sparse-matrices[110]. Also, several digital signal processing algorithms such as the Fast Fourier Transform (FFT), or LU matrix decomposition[4] have been parallelized using OpenMP directives[14]. However, research for performing parallel LDPC decoding in this type of architectures has not been reported yet.

### 5.2.2 The Cell/B.E. from Sony/Toshiba/IBM

The Cell/B.E. is an heterogeneous processor[24,60] composed by one main 64-bit PPE that communicates with eight SPEs each having 256 KByte of local storage (LS) memory for data and program, and a 128-bit wide vectorized SIMD within-a-register (SWAR) oriented architecture as depicted in figure 5.1. Data transfers between the PPE (that accesses main memory) and SPE are performed by using efficient DMA that offload

[4] In linear algebra, the LU matrix decomposition[14] represents a matrix as the product of a lower triangular matrix and an upper triangular matrix. This product sometimes involves a permutation matrix as well, and is used in numerical analysis to calculate the determinant or solve systems of linear equations.

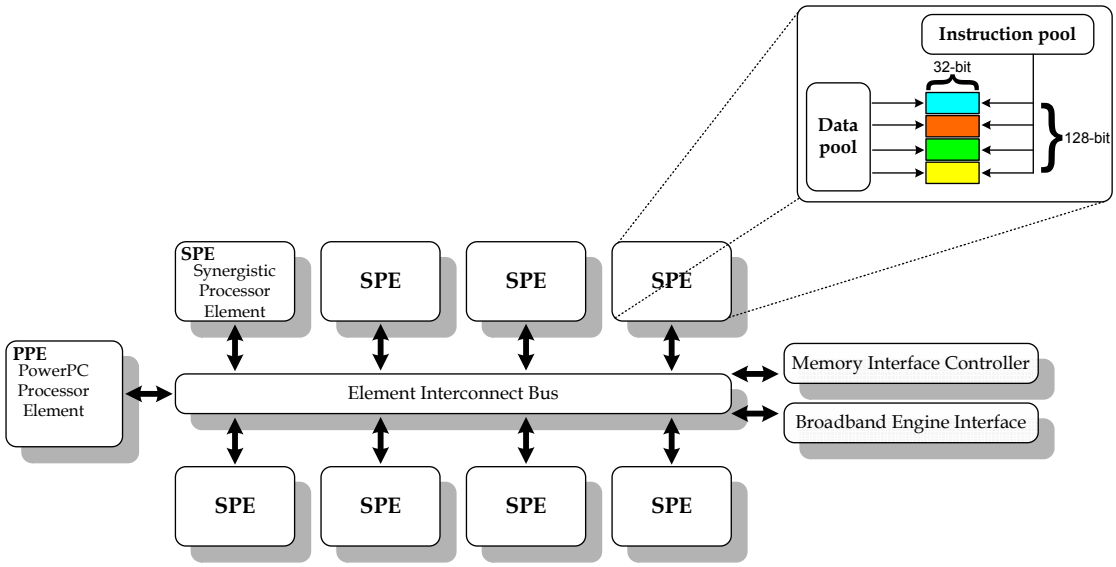


Figure 5.1: The Cell/B.E. architecture.

the processors from the time consuming task of moving data. The Memory Interface Controller (MIC) controls the access to the main memory, with a bandwidth that nearly reaches 4.2 GByte/s[2].

Differently from shared memory based architectures such as x86 multi-cores or GPUs, in the Cell/B.E. data is loaded from the main memory into the LS of a SPE and vice-versa. Data locality has to be individually exploited in each SPE, releasing to the programmer the burden of developing strategies to avoid contention due to memory accesses. An

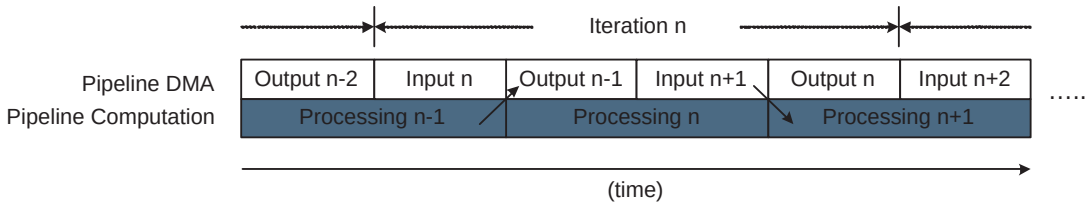


Figure 5.2: Double buffering showing the use of the two pipelines (for computation and DMA).

important restriction in such architectures is related with the limited amount of local memory[24], which in practice may lead to situations where data and code do not fit completely into the LS of a single SPE. In that case, the processing has to be performed by successively partitioning the algorithm, which increases the number of data communications between the PPE and SPEs. The DMA latency can have a critical role in the performance, but its impact can be reduced by using double-buffering techniques as depicted in figure 5.2.

Parallel computing on the Cell/B.E. has been mostly investigated for video applica-

tions, such as motion estimation[91], MPEG decoders[26], or feature extraction for tracking purposes in computer vision[116]. Also, there are approaches to solve systems of linear equations by exploiting single- and double-precision accuracy using appropriate refinement techniques on the Cell/B.E.[75]. In this work we open a novel frontier by developing parallel LDPC decoders in this type of distributed memory architecture.

### 5.2.3 Generic processing on GPUs with Caravela

Among the programming tools and environments developed for GPGPU are the Caravela interface tool[126], a general programming interface, based on a stream-based computing model[68] that can use any GPU as co-processor. Caravela does not directly interact with the GPU hardware, but it has a software layer that operates on the top of the GPU driver, which makes it a generic programming interface tool independent of the operating system and GPU manufacturer. The graphics pipeline of a GPU consists of several stages as depicted in figures 5.3 and 5.4. A graphics runtime such as DirectX or

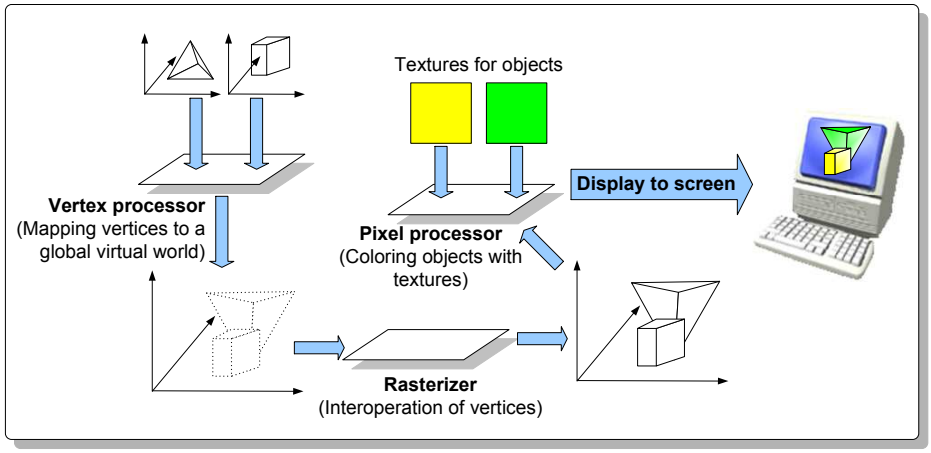


Figure 5.3: Vertex and pixel processing pipeline for GPU computation[126].

OpenGL is used to control GPU hardware. The CPU sends a data stream of vertices of graphics objects to a vertex processor that computes resizing or perspective of the objects calculating rotations and transformations of the coordinates, and generates new vertices. A rasterizer receives the new vertices, interpolates it, and generates data corresponding to the pixels to be displayed. Then, a pixel processor receives this data and computes the color applying texture data received from the CPU side. Finally, an image with graphics objects is ready to be displayed on a video screen. Because in GPUs, namely the old ones, there is no access to output data data of the vertex processor and the rasterizer is not programmable but directly implemented in hardware, general-purpose applications generally only use pixel processors, mapping the I/O data streams to textures.

Caravela has been developed[126] to allow the parallelization of computationally de-

manding kernels based on flow-models and apply them to accelerators such as generic GPUs. Moreover, a meta-pipeline execution mechanism has been developed, which allows to set up a pipeline of flow-models to be executed in parallel[128]. Also, the CaravelaMPI[127], which consists of a message passing interface targeted for GPU cluster computing, has been developed to allow the efficient programming of GPU-based clusters, providing a unified and transparent tool to manage both communications and GPU execution. Some applications such as FIR and IIR filtering[129] have been developed for Caravela and tested successfully reporting real speedups, regarding to the execution time on CPUs.

**Mapping flow-models on the Caravela platform:** On a GPU, the color data is written into the frame buffer, which outputs it to the screen as depicted in figure 5.3. Vertex and pixel processors compute four floating-point values (XYZW for vertex, ARGB for pixel) in parallel. Moreover, the coloring operation in the pixel processor is also parallelized because the output colors are generated independently as data streams, and each element of a stream is also independently processed. GPUs therefore include several pixel processor cores that generate output colors concurrently. These processors perform SIMD computation in four data units, and also concurrent calculations for the resulting output data streams. They can be programmed in standard languages such as the DirectX Assembly Language, the High Level Shader Language (HLSL)[90] and the OpenGL Shading Language (GLSL)[69]. The programs are called *shader* programs.

For GPGPU, programmers need specific knowledge for controlling GPU hardware via a graphics runtime environment. Moreover, there are different runtime environments, depending on the GPU vendor and the programming language. This is an overhead for programmers who have to concentrate their best efforts on implementing efficient paral-

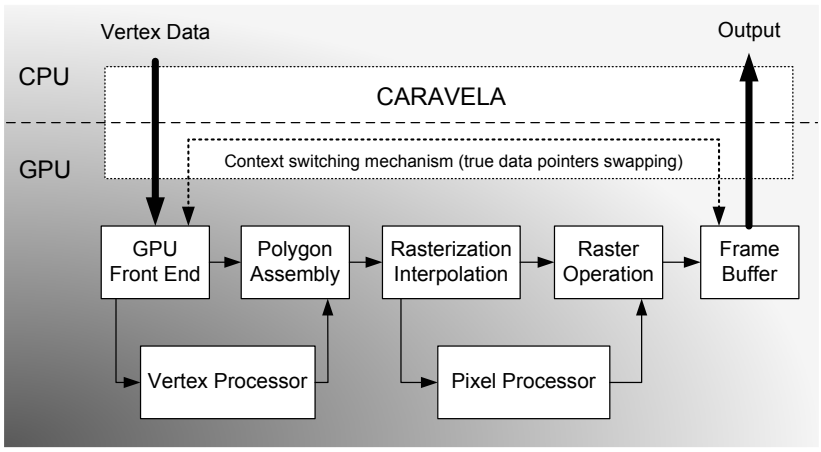


Figure 5.4: A GPU processing pipeline with Caravela.

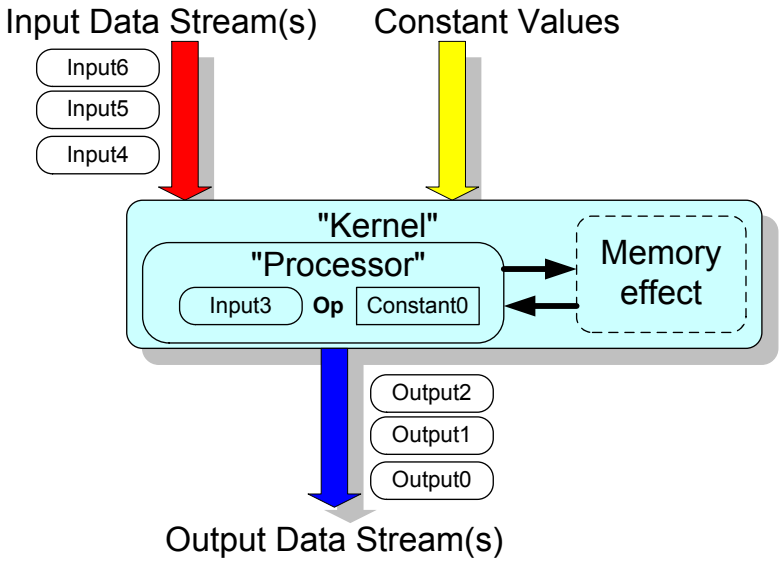


Figure 5.5: Structure of Caravela's flow-model.

lel algorithms in a *shader* program. To solve this disparity in programming GPU-based applications, the Caravela platform[126] has been developed for GPGPU, and is publicly available at the web site[125]. The Caravela platform mainly consists of a library that supports an Application Programming Interface (API) for GPGPU. The execution unit of the Caravela platform is based on the flow-model. As figure 5.5 shows, the flow-model is composed of input/output data streams, constant parameter inputs and a pixel *shader* program or kernel, which fetches the input data streams and processes them to generate the output data streams. The application program in Caravela is executed as stream-based computation. Although input data streams of the flow-model can be accessed randomly because they are based on memory buffers accessible to the program that uses the data, output data streams are sequences of data elements. The designation "pixel" for a unit of the I/O buffer is used because the pixel processor processes input data for every pixel color. A flow-model unit has defined the number of pixels for the I/O data streams, the number of constant parameters, the data type of the I/O data streams, the pixel *shader* program and the requirements for the targeted GPU. The "memory effect" is not available in hardware, which means that we cannot store temporary results during the processing, for example, of successive operations. To overcome this dificulty, data copy or pointer manipulation techniques have been implemented in Caravela[129]. To give portability to the flow-model, these items are packed into an Extensible Markup Language (XML) file. This mechanism allows the use of a flow-model unit located in a remote computer, simply by fetching the XML file.

### 5.2.4 NVIDIA graphics processing units with CUDA

Recent Tesla GPUs from NVIDIA are based on a unified architecture, which represents an evolution from previous streaming architectures, where geometry, pixel and vertex programs share common stream processing resources. The two GPUs from NVIDIA herein adopted have 16 or 30 multiprocessors, each one consisting of 8 cores as depicted in figure 5.6, which makes a total of 128 stream processors available in one case and 240 in the other, with 8192 or 16384 dedicated registers per multiprocessor [97], respectively. These GPUs provide shared memory on each multiprocessor that allows to efficiently exploit data parallelism. Each multiprocessor holds a 16 KByte shared memory block.

Data-parallel processing is exploited by executing a kernel in parallel through multiple threads in the available cores. CUDA [97] defines an API and provides a runtime environment and libraries to easily program these more recent GPUs from NVIDIA [97]. The execution of a kernel on a GPU is distributed across a grid of thread blocks with adjustable dimensions. Each multiprocessor, as depicted in figure 5.6, has several cores that can control more than one block of threads. Each block is composed by a maximum of 512 threads that execute the kernel synchronously with threads organized in groups of warps, and following a Single Instruction Multiple Thread (SIMT) approach: the warp size is 32 threads and each of the multiprocessors time-slices the threads in a warp among its 8 stream processors. However, allowing the existence of inconvenient

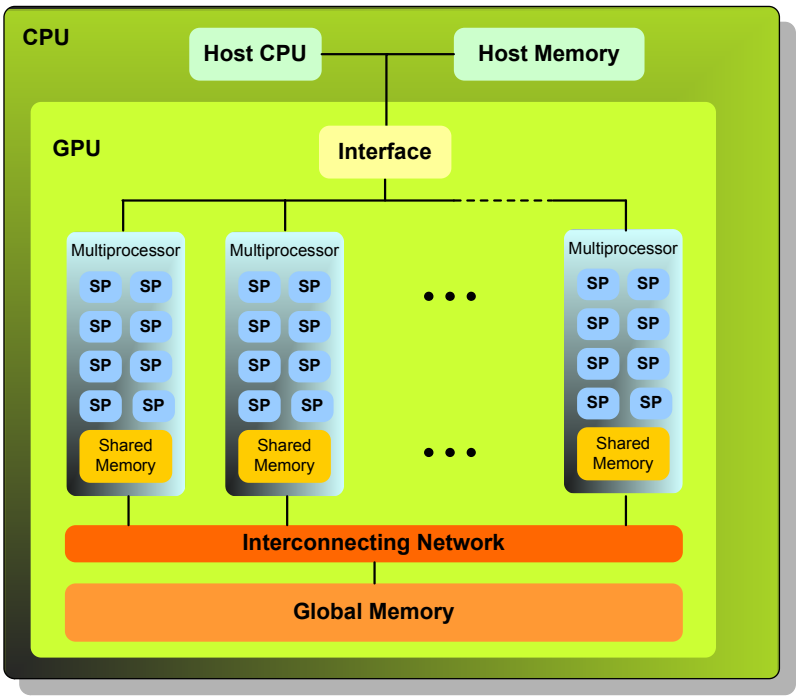


Figure 5.6: A compute unified Tesla GPU architecture with 8 stream processors (SP) per multiprocessor.

divergent threads inside a warp, or using a number of threads smaller than 32, can cause a wasting of resources or making them temporarily idle. Understanding how threads are grouped into warps is fundamental to choose the best block size configuration. Also, 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, in order to guarantee that enough physical registers are allocated to each thread. All threads inside the same block can share data through the fast shared memory block, which minimizes the need for expensive accesses to global memory. They can also synchronize execution at specific synchronization barriers inside the kernel, where threads in a block are suspended until they all reach that point.

CUDA became very popular among the scientific community in the last couple of years, fostering more efficient processing of algorithms based on intensive computation. The literature already presents cases of applications accelerated more than an order of magnitude[4] using this type of architecture[15,45,54,99], as compared to processing times obtained with conventional CPUs or even with CPUs using OpenMP[22]. Regarding the processing related with LDPC decoding, i.e. that involves calculations over matrices, and more specifically considering large data sets which can benefit most from parallelization, we can find parallel algorithms ranging from sparse matrix solvers[15] to belief propagation for stereo vision[19] implemented on GPUs. CUDA has fostered this increase of performance even further[22], allowing new parallel computing experiences[47], where video and image research[91,130] naturally have quickly assumed an important role. Nevertheless, the expected rise of the number of cores may also produce side effects[89], namely the traffic congestion caused by intensive accesses to memory, which represents a limiting factor in achieving performance or the scalability of algorithms.

## 5.3 Stream computing LDPC kernels

The Synchronous Data Flow (SDF) graph in figure 5.7 represents the computation of an LDPC decoder. The complete updating of CNs is performed by kernel 1 and alternates with the updating of BNs in kernel 2. Kernel 1 receives at the input data stream $p_0$ representing the initial probabilities $p_n$ (algorithms 2.1 and 2.2) and another stream representing $\mathbf{H_{BN}}$ data structures (section 3.3.3), to perform the SPA/MSA horizontal processing as indicated in (2.18) to (2.19), or (2.26) to (2.28)). It produces the output stream $r_0$ which is one of the input streams feeding kernel 2, that also receives the $\mathbf{H_{CN}}$ data structure (section 3.3.3) and produces the output stream $q_0$. $\mathbf{H_{CN}}$ is necessary to perform the SPA/MSA vertical processing as indicated in (2.20) to (2.21), or in (2.29). The pair kernel 1 and kernel 2 completes one iteration and is repeated $i$ times for an LDPC decoder

performing *i* iterations. After the final iteration, the last kernel conveys the codeword.

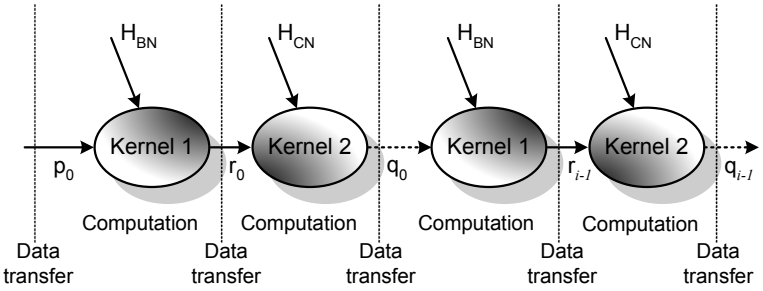


Figure 5.7: SDF graph for a stream-based LDPC decoder: the pair kernel 1 and kernel 2 is repeated *i* times for an LDPC decoder performing *i* iterations.

In each kernel, data elements can be read sequentially but have to be stored in non-contiguous positions (or vice-versa), which defines expensive random memory accesses. These costly write operations demand special efforts from the programmer in order to efficiently accommodate them in parallel architectures with distinct levels of memory hierarchy. Applied to kernel 1 and kernel 2, the stream-based compact data structures shown in figure 3.12 make the SPA/MSA suitable to run on a parallel processing platform. The analysis of figure 5.7 shows a message-passing schedule based on flooding schedule. This approach was detailed in subsection 3.3.2 and is adopted for the implementations described in this chapter.

Table 5.3: Types of parallelism on multi-core platforms[5].

| Platform | SIMD | SPMD | Multi/Intra-codeword |
|---|---|---|---|
| **Intel Xeon Nehalem 2x-Quad** | Y | Y | Y/Y |
| **STI Cell/B.E.** [24] | Y | N | Y/N[6] |
| **NVIDIA 8800 GTX** [96] | Y[7] | Y | Y/Y |

Table 5.3 shows the stream-based parallel approaches exploited for LDPC decoding on the considered platforms. The parallelization approaches for LDPC decoding exploit the properties of each architecture, and are proposed for a heterogeneous Cell/B.E. SIMD vectorized distributed memory architecture, that exploits data locality, and also for shared memory multi-core based GPU architectures using CUDA that support multi-threading, or alternatively use the Caravela interface. Moreover, the general-purpose parallel approaches here proposed can be applied to other multi-core architectures such as, for example, x86 general-purpose CPUs.

[5]These platforms were selected under the scope of this thesis.
[6]The Cell/B.E. exploits intra-codeword parallelism only in the large memory model.
[7]In this case it is called SIMT.

## 5.4 LDPC decoding on general-purpose x86 multi-cores[§]

The general-purpose multi-core processors replicate a single core in a homogeneous way, typically with a x86 instruction set, and provide shared memory hardware mechanisms. They support multi-threading and share data at a certain level of the memory hierarchy, and can be programmed at a high level by using different software technologies[71]. OpenMP[21] provides an effective and relatively straightforward approach for parallel programming general-purpose multi-cores.

Under this context, we developed LDPC decoders based on the computationally intensive SPA to test the viability of this solution based on general-purpose x86 multi-cores and supported by OpenMP.

**Algorithm 5.1** LDPC kernels executing on general-purpose x86 multi-cores using OpenMP

```
1:      /* Initialization. */
2: while (ĉ H^T ≠ 0 ∧ it < I)      /* c-decoded word; I-Max. number of iterations. */
   do
3:          /* Compute all the messages associated to all CNs. */
4:    #pragma omp parallel for
5:    shared(...) private(...)
6:       Processing kernel 1
7:          /* Compute all the messages associated to all BNs. */
8:    #pragma omp parallel for
9:    shared(...) private(...)
10:      Processing kernel 2
11: end while
```

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 straight-forward to parallelize algorithms 2.1 and 2.2, since LDPC decoding uses the intensive loop-based kernels depicted in figure 5.7. 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 distinct cores.

### 5.4.1 OpenMP parallelism on general-purpose CPUs

For programming general-purpose x86 multi-cores, we exploit OpenMP directives. Both horizontal and vertical kernels are based on nested loops. Considering the parallel resources allowed in the OpenMP execution, the selected approach consists of paral-

[§]Some portions of this section appear in:

[39] Falcão, G., Sousa, L., and Silva, V. (accepted in February 2010b). Massively LDPC Decoding on Multicore Architectures. *IEEE Transactions on Parallel and Distributed Systems*.

lelizing the outer-most loop of the costly operations defined in (2.18) to (2.21) or (2.26) to (2.29), thus reducing the parallelization overheads, as depicted in algorithm 5.1. The loop data accesses were analyzed in order to identify shared and private variables. This approach uses the *#pragma omp parallel for* directive to separately parallelize the processing of kernels 1 and 2 in algorithms 2.1 or 2.2.

**Algorithm 5.2** Multi-codeword LDPC decoding on general-purpose x86 multi-cores 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();
```

An alternative approach uses the *#pragma omp parallel section* directive to launch several decoders in parallel, which allows to achieve multi-codeword LDPC decoding, as represented in algorithm 5.2. In this case, the different cores do not need to share data, neither must be synchronized upon completion of the execution of all kernels. Here, each LDPC decoder executes at its own speed and it does not depend on the other kernels.

### 5.4.2 Experimental results with OpenMP

**Experimental setup:** To evaluate the performance of the proposed parallel stream-based LDPC decoder, an 8 core Intel Xeon Nehalem 2x-Quad E5530 multiprocessor running at 2.4 GHz with 8 MByte of L2 cache memory was selected. The general-purpose x86 multi-cores were programmed using OpenMP 3.0 directives and compiled with the Intel C++ Compiler (version 11.0). The experimental setup is depicted in table 5.4 and matrices under test are presented in table 5.5.

Table 5.4: Experimental setup

| | **x86 CPU** |
|---|---|
| Platform | Intel Xeon Nehalem 2x-Quad |
| Language | C + OpenMP 3.0 |
| OS | Linux (Suse) kernel 2.6.25 |
| Number of cores | 8 |
| Clock freq. | 2.4GHz |
| Memory | 8MB (L2 cache) |

**Experimental results with general-purpose x86 multi-cores using OpenMP:** The results presented in table 5.6 were achieved using algorithms 2.1 and 5.1. However, it

Table 5.5: Parity-check matrices **H** under test for the general-purpose x86 multi-cores

| **Matrix** | **Size** ($M \times N$) | **Edges** ($w_c \times M$) | **Edges/row** ($w_c$) | **Edges/col.)** ($w_b$) |
|---|---|---|---|---|
| o1 | $512 \times 1024$ | 3072 | {6} | {3} |
| o2 | $4000 \times 8000$ | 24000 | {6} | {3} |

should be noticed that using algorithm 5.2 for multi-codeword decoding only improves the results in average 20%. From the analysis of the results, it can be concluded that low throughputs are achieved regarding those required for real-time LDPC decoding. As it will be seen later in this chapter, they are also relatively low regarding the obtained for the Cell/B.E. and GPUs. The complex superscalar architecture of the individual cores is not suitable to exploit conveniently the data parallelism presented in the LDPC decoder.

Table 5.6: Decoding throughputs running the SPA with regular LDPC codes on x86 multi-cores programming algorithm 5.1 with OpenMP (Mbps)

| **Number of iterations** | **Matrix o1** | | | | **Matrix o2** | | | |
|---|---|---|---|---|---|---|---|---|
| | **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 |

However, it can be noticed that the proposed parallel approach for implementing LDPC decoders on general-purpose multi-cores with OpenMP, shows to be scalable. For example for Matrix o2 shown in table 5.6, 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 2 cores to a different level of parallelism that uses 4 cores shows a speedup of 1.7. Compared with the same 2 cores execution, the parallelization with 8 cores shows a speedup of approximately 3, but providing throughputs lower than a modest 2.6 Mbps value executing 10 iterations.

## 5.5 LDPC decoding on the Cell/B.E. architecture[§]

The strategy followed to develop an LDPC decoder on the Cell/B.E. consists of defining different tasks for the PPE and SPEs. The PPE controls the main tasks, offloading the intensive processing to the SPEs. The processing is distributed over several threads and each SPE runs independently of the other SPEs by reading and writing data, respectively, from and to the main memory, via DMA units controlled by the Memory Flow Controller (MFC). Synchronization between the PPE and the SPEs of the Cell/B.E. is performed using mailboxes. Data-parallelism and data locality are exploited by performing the partitioning and mapping of the algorithm and data structures over the architecture, while at the same time delays caused by latency and synchronization are minimized.

The proposed parallel LDPC decoder explores SIMD data-parallelism also by applying the same algorithm to different codewords on each SPE. The data structures that define the Tanner graph are also loaded into the LS of the respective SPEs where the processing is performed. Since the Tanner graph is common to all codewords under decoding, these data structures can be shared allowing distributed multi-codeword decoding simultaneously in all SPEs as figure 5.8 depicts (in the figure, each SPE processes 4 codewords in parallel, and each codeword is represented with 32-bit precision). Finally, the parallel algorithm herein presented exploits the double buffering property by overlapping computation and memory accesses, as shown in figure 5.2.

Depending on the LDPC code length, and more specifically if the corresponding data structures plus program fit the LS of the SPE or not, the processing can be performed following two distinct approaches: using *i*) the small memory model; or *ii*) the large memory model. In this context, we proposed to investigate the following scenarios: SPA-based LDPC decoders for regular codes, and MSA-based decoders for irregular LDPC codes, namely those used in the WiMAX standard.

[§]Some portions of this section appear in:

[36] Falcão, G., Silva, V., Sousa, L., and Marinho, J. (2008). High coded data rate and multicodeword WiMAX LDPC decoding on the Cell/BE. *Electronics Letters*, 44(24):1415–1417.

[38] Falcão, G., Sousa, L., and Silva, V. (2009c). Parallel LDPC Decoding on the Cell/B.E. Processor. In *Proceedings of the 4th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC'09)*, volume 5409 of *Lecture Notes in Computer Science*, pages 389–403. Springer.

[114] Sousa, L., Momcilovic, S., Silva, V., and Falcão, G. (2009). Multi-core Platforms for Signal Processing: Source and Channel Coding. In *Proceedings of the 2009 IEEE International Conference on Multimedia and Expo (ICME'09)*.

[33] Falcão, G., Silva, V., Marinho, J., and Sousa, L. (2009a). *WIMAX, New Developments*, chapter LDPC Decoders for the WiMAX (IEEE 802.16e) based on Multicore Architectures. In-Tech.

[39] Falcão, G., Sousa, L., and Silva, V. (accepted in February 2010b). Massively LDPC Decoding on Multicore Architectures. *IEEE Transactions on Parallel and Distributed Systems*.

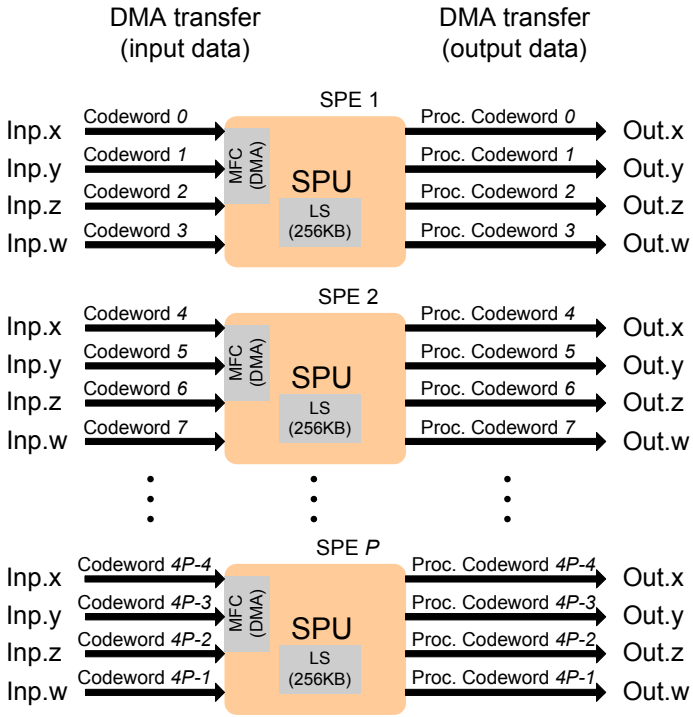


Figure 5.8: Detail of the parallelization model developed for the simultaneous decoding of several codewords using SIMD instructions.

### 5.5.1 The small memory model

Depending on the algorithm and the size of data to be processed, some tasks are small enough to fit into the 256 KByte LS of a single SPE. Concerning LDPC decoding, the single task environment described next and here denominated by "small model" supports workloads capable of decoding LDPC codes with lengths $N < 2000$. In this case, DMA data transfers shown in figure 5.9 will take place only twice during the entire processing: the first transfer is issued at the beginning before the processing starts, and the last one after data processing is completed. The number of communication points between the PPE and the SPEs in the small model is low, and synchronization is performed through mailboxes.

**PPE program:** Algorithm 5.3 is executed on the PPE and it communicates with only one SPE, which is called the MASTER SPE that controls the operations on the remaining SPEs. This is more efficient than putting the PPE controlling all the SPEs in this model.

The PPE receives the $y_n$ information from the channel and calculates $p_n$ (or $Lp_n$) data that is placed in main memory to be sent to the SPEs, after which it sends a *NEW_WORD* message to the MASTER SPE. Then, it waits for the SPEs to download all $p_n$ ($Lp_n$) data, and for the processing to be completed in each one of them (SPEs). Finally, when all the

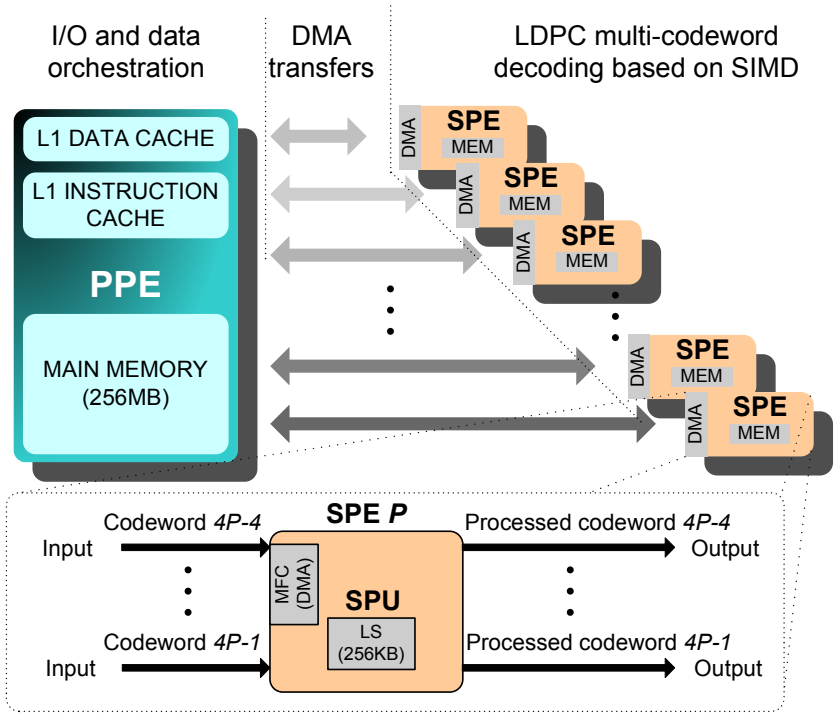


Figure 5.9: Parallel LDPC decoder on the Cell/B.E. architecture running the small memory model.

iterations are completed, the MASTER SPE issues an *END_DECODE* message to the PPE to conclude the current decoding process. The PPE gets ready to start processing a new word.

**Algorithm 5.3** PPE algorithm in the small memory model

1: **for** $th\_ctr = 1$ to $N\_SPEs$: **do**
2: Create $th\_ctr$ thread
3: **end for**
4: **repeat**
5: Receives $y_n$ from the channel and calculates $p_n$ ($Lp_n$)
6: Send msg *NEW_WORD* to MASTER SPE
**Ensure:** Wait until mail is received (SPE[i].mailboxcount $> 0$) from MASTER SPE
7: msg = SPE[i].mailbox (received msg *END_DECODE* from MASTER SPE)
8: **until** true

**SPE program:** The SPEs are used to perform intensive processing executing kernels 1 and 2 (see figure 5.7) on each decoding iteration. Each thread running on a SPE accesses the main memory by using DMA and computes data according to the Tanner graph, as defined by the **H** matrix of the code. The procedure in the MASTER SPE is described in algorithm 5.4, while the SLAVE SPE procedure is described in algorithm 5.5. The *Get* operation is adopted to represent a communication PPE $\rightarrow$ SPE, while the *Put* operation is used for communications in the opposite direction.

**Algorithm 5.4** MASTER SPE algorithm in the small memory model

```
 1: repeat
Ensure: Read mailbox (waiting a NEW_WORD mail from PPE)
 2:   Broadcast msg NEW_WORD to all other SPEs
 3:   Get p_n (Lp_n)
 4:   for i = 1 to N_Iter: do
 5:     Compute r_mn (Lr_mn)
 6:     Compute q_nm (Lq_nm)
 7:   end for
 8:   Put final Q_n (LQ_n) values on the PPE
Ensure: Read mailbox (waiting an END_DECODE mail from all other SPEs)
 9:   Send msg END_DECODE to PPE
10: until true
```

We initialize the process and start an infinite loop, waiting for communications to arrive from the PPE (in the case of the MASTER SPE), or from the MASTER SPE (for all other SPEs). In the MASTER SPE, the only kind of message expected from the PPE is a *NEW_WORD* message. When a *NEW_WORD* message is received, the MASTER SPE broadcasts a *NEW_WORD* message to all other SPEs and loads $p_n$ ($Lp_n$) data associated to itself. After receiving these messages, each one of the other SPEs also gets its own $p_n$ ($Lp_n$) values. The processing terminates when the number of iterations is reached. Then, the results are transferred to the main memory associated to the PPE, and an *END_DECODE* mail is sent by all SPEs to the MASTER SPE, which immediately notifies the PPE with an *END_DECODE* message.

**Algorithm 5.5** SLAVE SPE algorithm in the small memory model

```
 1: repeat
Ensure: Read mailbox (waiting a NEW_WORD mail from MASTER SPE)
 2:   Get p_n (Lp_n)
 3:   for i = 1 to N_Iter: do
 4:     Compute r_mn (Lr_mn)
 5:     Compute q_nm (Lq_nm)
 6:   end for
 7:   Put final Q_n (LQ_n) values on the PPE
 8:   Send msg END_DECODE to MASTER SPE
 9: until true
```

The intensive part of the computation in LDPC decoding on the Cell/B.E. architecture takes advantage of the processing power and SIMD instruction set available on the SPEs, which means that several codewords are decoded simultaneously in each SPE as depicted in figure 5.8.

### 5.5.2 The large memory model

In some situations, however, data and program do not fit completely into the 256 KByte LS of a single SPE. This is the case of LDPC codes with length $N > 2000$, which have to be decoded using a model here denominated as "large model". The approach

followed in this case is described in algorithms 5.6 and 5.7 and consists of having the PPE control process orchestrating the kernels execution on the SPEs and also data transfers. The number of data transfers between the PPE and the SPEs is substantially higher in the large model, because a complete iteration has to be processed sequentially on $K$ partitions of the Tanner graph. Synchronization is performed through mailboxes.

**PPE program:** One of the main tasks of the PPE is to execute a waiting loop, listening for mailbox communications arriving from the SPEs. The Cell/B.E. hardware supports a dual thread execution mode, where two threads perform the following tasks: one listens communications from SPEs 0 to $N\_SPEs/2 - 1$ and the other from $N\_SPEs/2$ to $N\_SPEs - 1$.

Algorithm 5.6 is executed in the PPE. It mainly orchestrates the overall execution mechanism on the SPEs, as well as data sorting of the buffers to be transmitted to each SPE. In the main memory several buffers are dedicated to each SPE. They are used to manage data during the $K$ steps of an iteration of the decoding algorithm. Some of these buffers are represented in figure 5.10 as *Unsorted buffer 1* to *Unsorted buffer N_SPEs*, and *Sorted buffer 1* to *Sorted buffer N_SPEs*. These buffers are used to manage the $y_n$ data initially received from the channel at the input of the LDPC decoder, which is processed to generate $p_n$ ($Lp_n$) data that is sent to each SPE. They also represent the place where the results received from each SPE are accommodated and reorganized. The *Unsorted buffers* hold data arriving from each SPE, which is then sorted and placed into *Sorted buffers* before being sent again to each SPE to continue the computation of an iteration. The sorting procedure is depicted in figure 5.10 and it consists of organizing groups of CNs (or BNs, depending if we are processing, respectively, the horizontal or vertical kernels) in contiguous data blocks of memory to accelerate DMA transfers. When the buffers are created, a number of threads equal to the number of SPEs is 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 and the SPEs use these addresses to perform DMA data transfers.

On every iteration, two different kinds of actions must be completed $K$ times: *i*) a sub-block of the $r_{mn}$ ($Lr_{mn}$) or $q_{nm}$ ($Lq_{nm}$) data is sorted and placed in the buffers to be transmitted to the SPE; and *ii*) the processed data (BNs or CNs alternately updated) is sent from the SPE back to the main memory. The actions are synchronized on both sides of the PPE and SPE. After completing the first action, confirmation mails are sent to the SPEs, while after concluding the last action, confirmation mails are awaited from the SPEs. When data is loaded into the buffers in main memory, confirmation mails inform the SPEs that the DMA transfers can start: for the first sub-block of data to compute, a

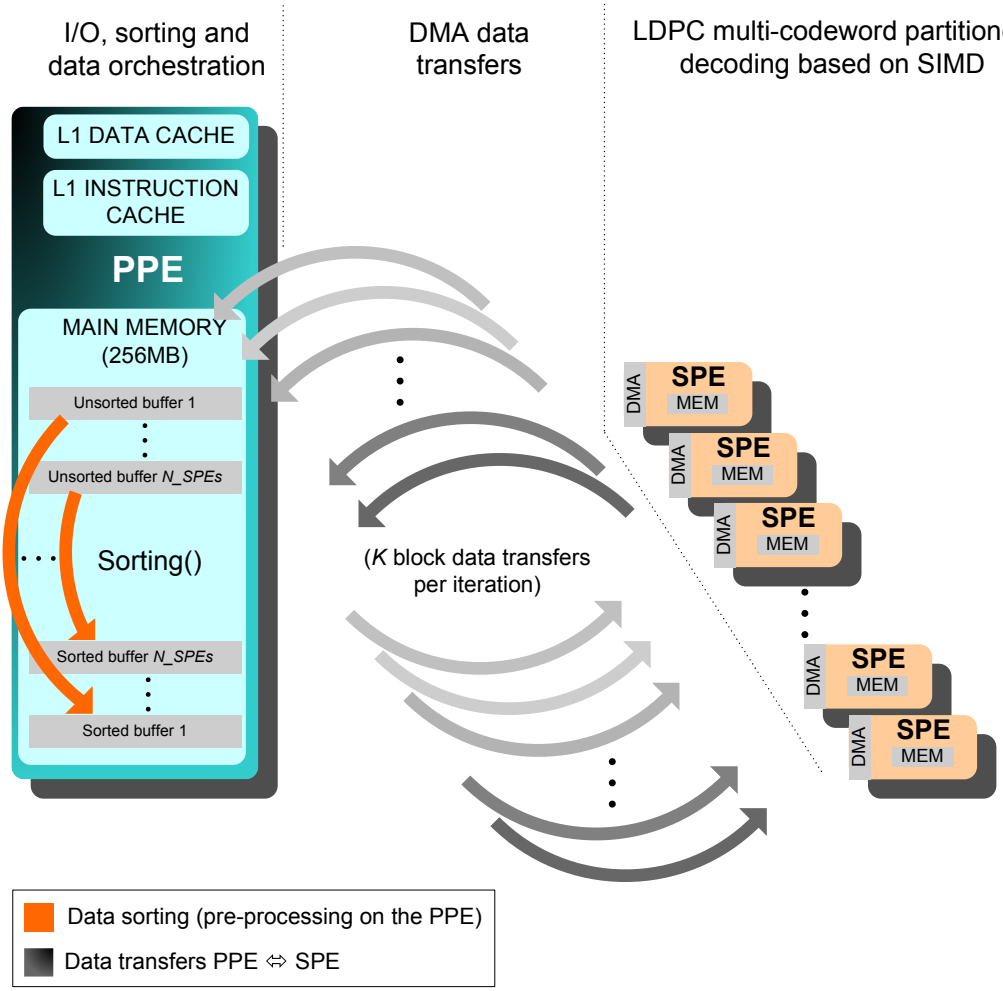


Figure 5.10: Parallel LDPC decoder on the Cell/B.E. architecture running the large memory model.

*NEW_WORD* mail is sent to the SPE to make it start loading data from the buffer in main memory and, when this transfer completes, to initiate the load of the next data buffer.

For the sake of simplicity of representation, the inner loop in algorithm 5.6 omits the synchronization on the PPE side in steps ranging from 12 to 14, for indicating that old information present in the SPE buffers has already been replaced and that the PPE needs to sort a new sub-block with $r_{mn}$ ($Lr_{mn}$) or $q_{nm}$ ($Lq_{nm}$) data. Here, the PPE would expect mails from the SPEs notifying that there is new data available in the *Unsorted buffers* waiting 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 new words.

**Algorithm 5.6** PPE algorithm in the large memory model

```
1: Allocate Sorted buffers and Unsorted buffers
2: for th_ctr = 1 to N_SPEs: do
3:     Create th_ctr thread
4:     Send Sorted buffers and Unsorted buffers addresses to the thread
5: end for
6: repeat
7:     Receives y_n from the channel and calculates p_n (Lp_n)
8:     Send msg NEW_WORD to all SPEs
9:     for i = 1 to N_SPEs: do
10:        for i = 1 to N_Iter: do
11:            for block = 1 to K: do
12:                Get data1 block to SPE[i]
13:                Put data2 block from SPE[i]      /* data1 and data2 will be alternately r_mn (Lr_mn) and q_nm (Lq_nm)
                   */
14:                Sorted buffers ← Unsorted buffers      /* Sort data blocks on the PPE side */
15:            end for
16:        end for
17:    end for
Ensure:    Wait until mail is received (SPE[i].mailboxcount > 0)
18:    msg = SPE[i].mailbox (received msg END_DECODE from all SPEs?)
19: until msg != END_DECODE
```

**SPE program:** In the large model, the SPEs are also dedicated to the computationally intensive updating of all CNs and BNs by executing sub-blocks of kernel 1 and kernel 2 alternately, during each iteration of the LDPC decoder. The SPE procedure is described in algorithm 5.7.

The process is initialized and starts an infinite loop, waiting for communications to arrive from the PPE. At this point, the communication expected from the PPE is a *NEW_WORD* message. At first, when a *NEW_WORD* message is received, the SPE loads the $p_n$ ($Lp_n$) data. After that, the first horizontal sub-block of $q_{nm}$ ($Lq_{nm}$) data is read into the buffer on the SPE. As soon as the DMA transfer completes, a new one is immediately started, so that the next transfer of a vertical sub-block of $r_{mn}$ ($Lr_{mn}$), or an horizontal sub-block of $q_{nm}$ ($Lq_{nm}$), is performed in parallel with the execution of data previously loaded on the SPE. This way, data transfers are masked by calculations using the double buffering technique. When a new mail is received, the SPE starts a new computation to perform either CN or BN updating. When it completes, the SPE updates the buffers with the new processed data and then starts a new transfer. It should be noticed that time spent performing communications should balance the computation time, which is important to achieve efficiency within the architecture.

After completing the processing of a kernel, it waits for the previous DMA transfer to complete and then performs a new DMA transfer to send processed data back to main memory. When the transfer is concluded, a confirmation mail is sent to the PPE indicating that the new data in the *Unsorted buffer* is available and can be sorted by the PPE in order to proceed with the following processing. The computation terminates when the number

of iterations is reached and an *END_DECODE* mail is sent by all SPEs to the PPE.

**Algorithm 5.7** SPE algorithm in the large memory model

```
 1: repeat
Ensure: Read mailbox (waiting a NEW_WORD mail from PPE)
 2:   Get p_n (Lp_n)
 3:   for i = 1 to N_Iter: do
 4:     for block = 1 to K: do
 5:       /* The processing of one iteration is partitioned into K steps */
 6:       Get q_nm (Lq_nm) partial data from PPE
 7:       Compute r_mn(Lr_mn) ← q_nm(Lq_nm)
 8:       Put processed r_mn (Lr_mn) values on the PPE
 9:     end for
10:     for block = 1 to K: do
11:       /* The processing of one iteration is partitioned into K steps */
12:       Get r_mn (Lr_mn) partial data from PPE
13:       Compute q_nm(Lq_nm) ← r_mn(Lr_mn)
14:       Put processed q_nm (Lq_nm) values on the PPE
15:     end for
16:   end for
17:   Put final Q_n (LQ_n) values on the PPE
18:   Send msg END_DECODE to PPE
19: until true
```

### 5.5.3 Experimental results for regular codes with the Cell/B.E.

**Experimental setup:** The parallel processing platform selected at this time to perform LDPC decoding was a Cell/B.E. from STI, where the PPE and each SPE run at 3.2 GHz. The Cell/B.E. has a total of 256 MByte of main memory and each SPE has 256 KByte of fast local memory. The system is included in a PlayStation 3 (PS3) platform, which restricts the number of SPEs available to 6 from a total of 8. The LDPC decoder uses the compact structures defined in figure 3.13 to represent the Tanner graph. Programs were written in C and use single precision floating-point arithmetic operations. A sequential implementation was considered in the present work. It was programmed using only the PPE of the Cell/B.E. architecture with SIMD and dual thread execution. This sequential approach was adopted as the reference algorithm with the purpose of evaluating against

Table 5.7: Experimental setup for the Cell/B.E.

| | **Serial mode** | **Parallel mode** | |
|---|---|---|---|
| Processing mode | PPE | PPE + SPEs | |
| Platform | STI PlayStation3 (PS3) | | |
| Language | C | C | |
| OS | Linux (Fedora) kernel 2.6.16 | | |
| | | PPE | SPE |
| Clock frequency | 3.2GHz | 3.2GHz | 3.2GHz |
| Memory | 256MB | 256MB | 256KB |

Table 5.8: Parity-check matrices **H** under test for the Cell/B.E.

| **Matrix** | **Size** $(M \times N)$ | **Edges** $(w_c \times M)$ | **Edges/row** $(w_c)$ | **Edges/col.)** $(w_b)$ |
|---|---|---|---|---|
| p1 | $128 \times 256$ | 768 | {6} | {3} |
| p2 | $252 \times 504$ | 1512 | {6} | {3} |
| p3 | $512 \times 1024$ | 3072 | {6} | {3} |
| p4 | $2448 \times 4896$ | 14688 | {6} | {3} |
| p5 | $288 \times 576$ | 1824 | {2,3,6} | {6,7} |
| p6 | $480 \times 576$ | – | {2,3,4} | {20} |
| p7 | $560 \times 672$ | – | {2,3,4} | {20} |
| p8 | $480 \times 960$ | – | {2,3,6} | {6,7} |
| p9 | $800 \times 960$ | 3200 | {2,3,4} | {20} |
| p10 | $576 \times 1152$ | – | {2,3,6} | {6,7} |
| p11 | $960 \times 1152$ | – | {2,3,4} | {20} |
| p12 | $624 \times 1248$ | 3952 | {2,3,6} | {6,7} |
| p13 | $1040 \times 1248$ | – | {2,3,4} | {20} |

parallel solutions. The experimental setup is depicted in table 5.7, and matrices under test are presented in table 5.8.

To evaluate the performance of the proposed stream-based LDPC decoder, the Cell/B.E. was programmed using both small and large models. In the small model, a complete matrix fits the SPE LS memory and the processing is efficiently completed using few data transactions between the PPE and the SPEs. In the large model, the decoding of a codeword is performed by decomposing the *Tanner* graph (i.e., the $r_{mn}$ ($Lr_{mn}$), $q_{nm}$ ($Lq_{nm}$) and corresponding data structures) into different sub-blocks and applying the SPE to process them on a sequential basis. Matrices for LDPC codes p1 to p3 and p5 to p7 fit into the LS of a single SPE and were programmed using the small model, while matrix p4 is too large to fit into a single SPE and has to be processed on a sub-block by sub-block basis, using the large model described in subsection 5.5.2. Matrix p3 was also implemented in the large model to allow a comparison between the two computational models. All processing times were obtained for decoders performing a number of iterations ranging from 10 to 50.

**Experimental results with the small model:** Figure 5.11 presents the decoding times for matrices p1 to p3 under the small model running the SPA. They relate the execution time needed for the Cell/B.E. to decode a codeword with the processing time required to decode a codeword on the sequential version. The Cell/B.E. performs concurrent processing on the 6 SPEs, each using SIMD to process 4 codewords simultaneously. The decoding time for matrix p3 shows that the Cell/B.E. takes approximately 353.6$\mu$s to de-

Table 5.9: Decoding throughputs for a Cell/B.E. programming environment in the small model running the SPA with regular LDPC codes (Mbps)

| Number of iterations | Matrix p1 | Matrix p2 | Matrix p3 |
|---|---|---|---|
| 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 |

code $24 \times 1024$-bit codewords on the 6 SPEs (an average value of 14.7$\mu$s per codeword) in 10 iterations, against 244.9$\mu$s per codeword on the serial version that concurrently decodes 8 codewords (dual thread decoding using SIMD) for the same number of iterations. It achieves a throughput of 69.5 Mbps which compares well against VLSI LDPC decoders. Observing table 5.9, we conclude that the decoding capacity per bit is high and approximately constant for all the regular codes under test (matrices p1, p2 and p3). Comparing this throughput with that obtained on the serial approach, the speedup achieved with the Cell/B.E. surpasses 16. In the serial mode, the PPE accesses the main memory that is slower than the LS on the SPEs which, as a consequence, has an influence on the reported speedup [38].

Experimental evaluation also shows that regarding the same algorithm applied only to one SPE, decoding 4 codewords, the full version that uses 6 SPEs decoding 24 codewords, achieves a speedup only 14% below the maximum (6), or equivalently an effi-

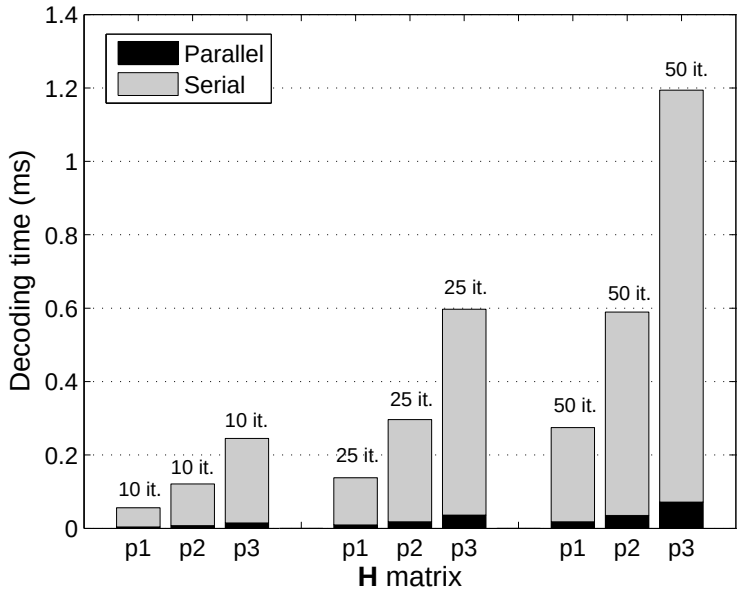


Figure 5.11: LDPC decoding times per codeword for sequential and parallel modes on the Cell/B.E., running the SPA with regular codes in the small model (ms)

ciency $Ef = 86\%$. 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. This allows to conclude that the solution is scalable under the small memory model.

**Experimental results with the large model:** In this case, the 256 KByte of LS memory in the SPE are not enough to fit the complete data structures necessary to decode 4 codewords and also the program. The maximum regular code allowed is limited by the LS size, and for a $rate = 1/2$ code it can be described according to ($w_c$ represents *Edges* per row and $N$ the codeword size):

$$N \times w_b \times 36 + 32 \times N + Program\_Size \leq 256KByte. \quad (5.1)$$

The processing is then performed in successive partitions of the Tanner graph and the number of data transfers and synchronization operations degrades the performance of the decoder. In the large model a single iteration has to be performed using several data transactions. Data transfers between each SPE and the main memory 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 involve concurrent accesses to the slow main memory that can cause some processors to temporarily idle in this model. This is the main reason why the throughput decreases significantly in this model. Matrix p3 was also tested in the large model to allow relatively assessing the two computational models, and the reported throughput running the SPA for 10 iterations, decreases to 9 Mbps which represents a throughput more than 7 times lower than the one obtained using the small model. Matrices with lengths equivalent or superior to p4 are too large to fit into a single SPE and have to be processed on a sub-block by sub-block basis, using the large model. They all achieve throughputs inferior to 9 Mbps, which led to the conclusion that the Cell/B.E. processor is not efficient for LDPC decoding with large **H** matrices.

Increasing the number of SPEs used in the architecture would cause the throughput to rise, but not 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.

**Analyzing the performance on the Cell/B.E.:** The $N\_SPEs$ number of SPEs available and the number of instructions per iteration $N_{op/iter}$ have a major impact in the overall performance of the decoder on the Cell/B.E. This performance also depends on the number of iterations $N_{iter}$ and frequency of operation $f_{op}$. Here, the throughput performance ($T$) can be estimated for the small model according to (5.2). Differently from what

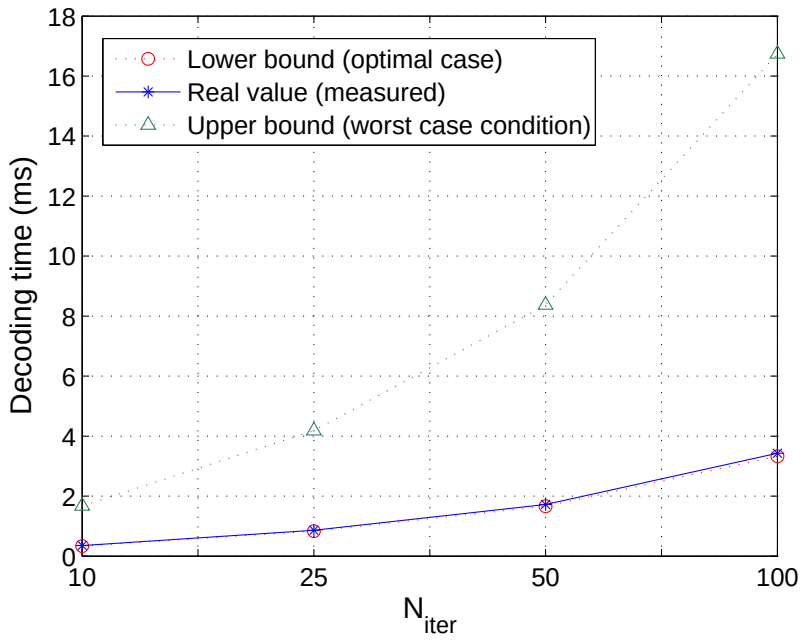


Figure 5.12: Model bounds for execution times obtained on the Cell/B.E.

happens in other parallel architectures, where a task is responsible for decoding only a part of the Tanner graph, in this approach and processing model each SPE performs the complete processing of the Tanner graph as described in algorithms 2.1 and 2.2, independently from the other SPEs. The LDPC decoder supports peak data transfers between the SPE and main memory of approximately 4GByte/s[2] and, consequently, its performance is mainly 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.

$$T = \frac{N\_SPEs \times 4 \times N}{T_{ppe \rightarrow spe} + \frac{N_{op/iter} \times N_{iter}}{f_{op}} + T_{spe \rightarrow ppe}} \quad [bps]. \tag{5.2}$$

The proposed LDPC decoder suits scalability, and it could be easily adopted by other generations of this family of processors having a higher number of SPEs available. In that case, it would be able to decode more codewords simultaneously, increasing even more the speedup and throughput of the decoder. The model proposed in (5.2) makes it dificult to predict the execution time, not because the number of instructions $N_{op/iter}$ generated by the Cell SDK 2.1 compiler is unknown, but rather due to the parallel pipeline that performs data transfers independently from arithmetic operations and whose behavior cannot be predicted due to branch misprediction penalties that force emptying the pipeline. Nevertheless, the model can be used to establish the limits of performance. We decided to adopt upper and lower bounds representing processing times, respectively, for worst and best case working conditions. The worst case condition (upper bound) assumes process-

ing and memory accesses are performed without any level of parallelism and a branch misprediction occurs after every instruction (a very unrealistic condition). The best case condition (lower bound) assumes that no branch mispredictions occur and that there is full parallelism (overlap) between memory accesses and computation. Figure 5.12 shows that the parallelism supported by the dual-pipelined Cell/B.E. architecture exploits the LDPC decoder near the optimal case. The measured experimental results are only 6% worst than those estimated for the best case working conditions, here defined by the lower bound, and the algorithm executes very close to the maximum performance the Cell/B.E. can provide. Measured values and lower bounds almost overlap in figure 5.12.

### 5.5.4 Experimental results for irregular codes with the Cell/B.E.

To increase the efficiency of the LDPC decoder we implemented the MSA on the Cell/B.E.. It requires less computation, based essentially in addition and comparison operations. Additionally, we also adopted the Forward-and-Backward simplification of the algorithm[83] that avoids redundant computation and eliminates repeated accesses to memory. In the MSA, data elements have 8-bit integer precision, which allows packing 16 data elements per 128-bit memory access. This increases the arithmetic intensity of the algorithm, which favors the global performance of the LDPC decoder. The instruction set of the Cell/B.E. architecture supports intrinsic instructions to deal efficiently with these parallel 128-bit data types. Moreover, because there are 6 SPEs available, in this implementation the algorithm supports the simultaneous decoding of 96 codewords in parallel. However, the set of 8-bit integer intrinsic parallel instructions of the Cell/B.E. is more limited than those of the 32-bit floating-point family of instructions. This explains that the speedup obtained when changing from the SPA to the MSA is lower than we would expect.

**Running LDPC WiMAX codes on the Cell/B.E.:** Table 5.10 shows the throughputs obtained decoding a subset of WiMAX IEEE 802.16e standard[64] irregular codes, considering 10 iterations. A more vast set of results can be found in[33]. In some cases they approach quite well, while in others they even surpass the 75 Mbps required by the standard to work in (theoretical) worst case conditions. For the two codes with $N = 576$ represented by matrices p5 and p6, when running 10 iterations we obtain throughputs of 79.8 and 79.3 Mbps. For codes p12 and p13, with $N = 1248$ and for the same number of iterations, they range from 79.6 to 78.4 Mbps. All codes in table 5.10 were tested for the MSA and approach quite well the maximum theoretical limit of 75 Mbps. They all show better performances than those obtained with the SPA. Furthermore, if we consider a lower number of iterations, the decoder's throughputs may rise significantly. For ex-

Table 5.10: Decoding throughputs for a Cell/B.E. programming environment in the small model running the MSA with irregular WiMAX LDPC codes (Mbps)

| Number of iterations | p5 | p6 | p7 | p8 | p9 | p10 | p11 | p12 | p13 |
|---|---|---|---|---|---|---|---|---|---|
| 10 | 79.8 | 79.3 | 78.5 | 79.6 | 78.4 | 79.6 | 78.4 | 79.6 | 78.4 |
| 25 | 32.7 | 32.5 | 32.2 | 32.6 | 32.1 | 32.6 | 32.1 | 32.6 | 32.1 |
| 50 | 16.5 | 16.4 | 16.2 | 16.4 | 16.2 | 16.4 | 16.2 | 16.4 | 16.2 |

ample, for 5 iterations instead of 10, the throughput is approximately the double (above 145 Mbps).

## 5.6 LDPC decoding on GPU-based architectures[§]

Following an alternative parallel approach, the design of software LDPC decoders on GPUs is proposed in the next sections. The algorithms were implemented by using the programming interfaces presented in the previous sections, namely the CUDA from NVIDIA[97] and the Caravela[125], which are nowadays massively disseminated and provide efficient tools to exploit low-cost computational power and achieve flexibility/reprogrammability for LDPC decoding.

The development of flow-models to perform LDPC decoding and apply them to GPUs under the context of Caravela hadn't been tried before. It was the first time that this type of complex and irregular LDPC decoding algorithms were parallelized and accelerated in the scope of this thesis[40,41]. Under this context, Caravela flow-models were developed to perform LDPC decoding based on the SPA to test the viability of the solution with a demanding computational workload. Although it hasn't been tested, applying the MSA (computationally less demanding) to GPUs using the Caravela is also possible using a similar approach. In the CUDA context, we also proposed to investi-

[§]Some portions of this section appear in:

[32] Falcão, G., Silva, V., Gomes, M., and Sousa, L. (2008). Edge Stream Oriented LDPC Decoding. In *Proceedings of the 16th Euromicro International Conference on Parallel, Distributed and network-based Processing (PDP'08)*, pages 237–244, Toulouse, France.

[34] Falcão, G., Silva, V., and Sousa, L. (2009b). How GPUs can outperform ASICs for fast LDPC decoding. In *Proceedings of the 23rd ACM International Conference on Supercomputing (ICS'09)*, pages 390–399. ACM.

[37] Falcão, G., Sousa, L., and Silva, V. (2008). Massive Parallel LDPC Decoding on GPU. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP'08)*, pages 83–90, Salt Lake City, Utah, USA. ACM.

[40] Falcão, G., Yamagiwa, S., Silva, V., and Sousa, L. (2007). Stream-Based LDPC Decoding on GPUs. In *Proceedings of the First Workshop on General Purpose Processing on Graphics Processing Units – GPGPU'07*, pages 1–7.

[41] Falcão, G., Yamagiwa, S., Silva, V., and Sousa, L. (2009d). Parallel LDPC Decoding on GPUs using a Stream-based Computing Approach. *Journal of Computer Science and Technology*, 24(5):913–924.

[39] Falcão, G., Sousa, L., and Silva, V. (accepted in February 2010b). Massively LDPC Decoding on Multicore Architectures. *IEEE Transactions on Parallel and Distributed Systems*.

gate SPA and MSA-based LDPC decoders for regular codes, and MSA-based decoders for irregular LDPC codes, namely those applied to the DVB-S2 standard.

### 5.6.1 LDPC decoding on the Caravela platform

Under the Caravela context, the pixel processors are the only programmable processors and the processing is based on square 2D textures, with dimensions power of 2, as depicted in figure 5.13. As shown in the same figure, the Caravela's computational model is based on a bijective application between pixel processors and pixel textures, which means that there is only one entry point for each pixel processor. This characteristic poses difficult challenges in the development of kernels 1 and 2 of algorithm 2.1. Under this context, the processing of each individual edge is associated to a single pixel processor. For each pixel processor to be able of accessing several data elements from a unique entry point, we developed a circular addressing mechanism dedicated to Caravela that suits well the irregular data accesses imposed by the algorithm, as described next.

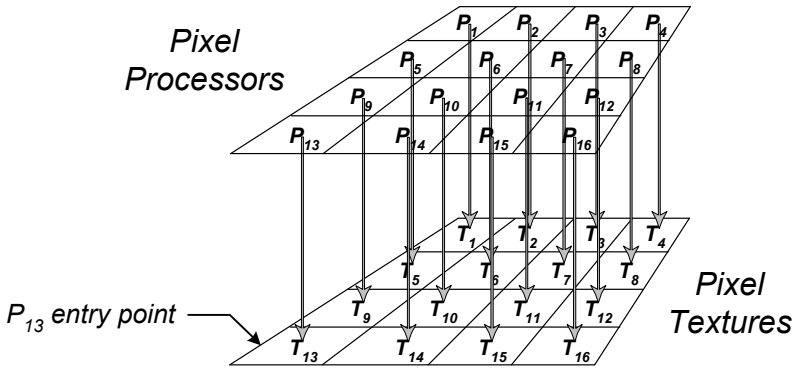


Figure 5.13: 2D textures $T_n$ associated to pixel processors $P_n$ in Caravela.

The tenths of pixel processors available allow exposing parallelism, but impose a type of processing based on textures, where each pixel texture represents an edge of the Tanner graph. This solution imposes redundant computation because it does not allow the reutilization of data. Nevertheless, the level of parallelism achieved is consistent[40,41] and the approach effectively allowed to develop parallel LDPC decoding algorithms based on the simultaneous update of several BNs or CNs processed by distinct pixel processors (following an edge approach, where each pixel processor only updates an edge). In order to implement such an approach, dedicated data structures were developed to support the texture based processing of the SPA (described in algorithm 2.1).

**Stream-based data structures for the Caravela:** To take advantage of the processing performance of GPUs, efficient data structures adapted for 2D textures and for stream

computing were derived from those presented in section 3.3.3. A stream-based LDPC decoder needs different computation and memory access patterns when executing consecutive kernels to update BNs and CNs, respectively. In order to support the execution of kernels 1 and 2 (see figure 5.7) on the GPU based on the Caravela platform, we propose two stream-based data structures adapted from $\mathbf{H}_{BN}$ and $\mathbf{H}_{CN}$ presented in section 3.3.3. These data structures are suitable for stream computing both regular and irregular LDPC codes on the GPU grid based on 2D textures with dimensions that are a power of 2. Algorithm 5.8 details this procedure for the case of $\mathbf{H}_{BN}$, depicted in figure 5.14 for the $\mathbf{H}$ matrix shown in figure 2.1. In step 5, it can be seen that all edges associated with the same CN are collected and stored in consecutive positions inside $\mathbf{H}_{BN}$. The addressing in each row of $\mathbf{H}$ becomes circular. The pixel element corresponding to the last non-null element

**Algorithm 5.8** Generating compact 2D $\mathbf{H}_{BN}$ textures from original $\mathbf{H}$ matrix

1: /* *Reading a binary* $M \times N$ *matrix* $\mathbf{H}$ */
2: ***for all*** $CN_m$ *(rows in* $\mathbf{H}_{mn}$*)* : ***do***
3: ***for all*** $BN_n$ *(columns in* $\mathbf{H}_{mn}$*)* : ***do***
4: ***if*** $\mathbf{H}_{mn} == 1$ ***then***
5: $ptr_{next} = j : \mathbf{H}_{mj} == 1$, *with* $n + 1 \leq j < (n + N) \bmod N$;
/* *Finding circularly the right neighbor on the current row* */
6: $\mathbf{H}_{BN} = ptr_{next}$;
/* *Store* $ptr_{next}$ *into the* $\mathbf{H}_{BN}$ *structure, using a square texture of dimension* $2^D \times 2^D$, *with* $D = \left\lceil \frac{1}{2} \times log_2( \sum_{m=1}^{M} \sum_{n=1}^{N} \mathbf{H}_{mn}) \right\rceil$ */
7: ***end if***
8: ***end for***
9: ***end for***

of each row points to the first element of this row, implementing a circular addressing that is used to update all associated messages in that row. The circular addressing allows to introduce a high level of parallelism. In the limit, for a multi-processor platform, a different pixel processor can be allocated to every single edge, i.e. to the computation of each individual message. Each element of the data structure, here represented by a pixel texture, records the address of the next entry pointer and the corresponding value of $r_{mn}$. Although the pixel elements in figure 5.14 are represented by their row and column addresses, the structures can be easily vectorized by convenient 1D or 2D reshaping according to the target stream-based architecture they apply to. The 3D representation in figure 5.14 shows that the same matrix information can be used to simultaneously decode several codewords (multi-codeword decoding), by applying SIMD processing, for example.

In the upper left corner of figure 5.14, it can be seen that the pixel processor allocated to compute the message $m^i_{CN_0 \to BN_0}$ (identified as message $r_{0,0}$) depends on messages $m^{i-1}_{BN_1 \to CN_0}$ and $m^{i-1}_{BN_2 \to CN_0}$ coming from $BN_1$ and $BN_2$. This is equivalent to saying that to update $BN_0$ (upper left pixel), we have to read the information from $BN_1$ ($BN_0$

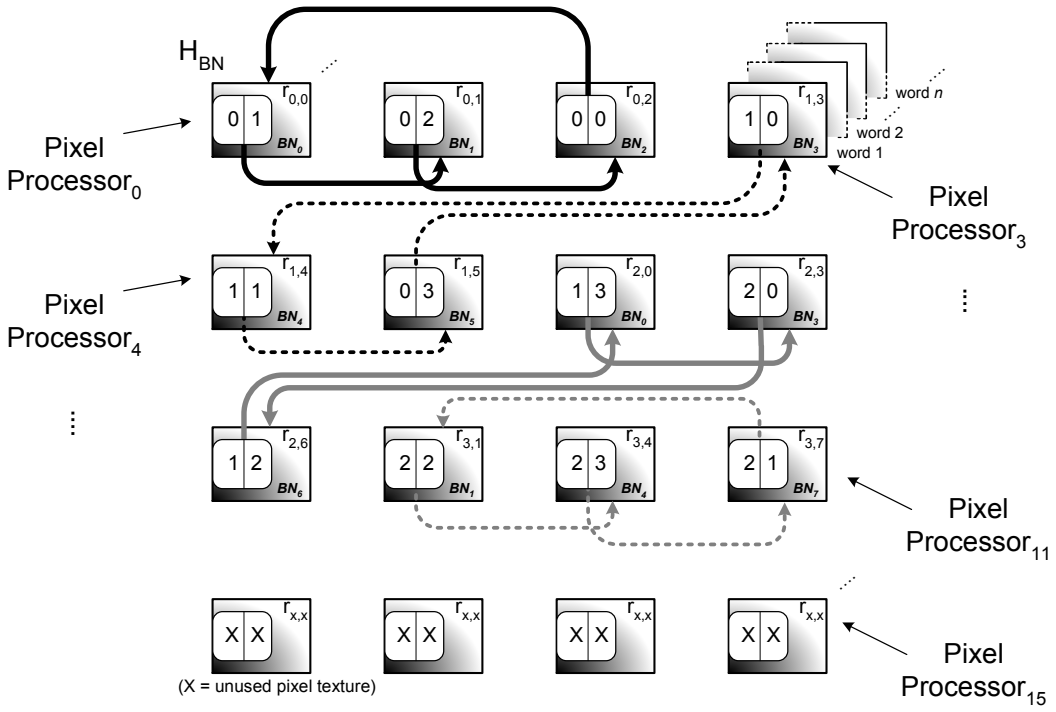


Figure 5.14: $\mathbf{H}_{BN}$ structure. A 2D texture representing Bit Node edges with circular addressing for the example in figure 2.1. Also, the pixel processors entry points are shown.

holds the address of $BN_1$) and $BN_2$ ($BN_1$ holds the address of $BN_2$) circularly, and then update $BN_0$ ($BN_2$ knows the address of $BN_0$). This mechanism is used to update all the other BNs in parallel.

**Algorithm 5.9** Generating compact 2D $\mathbf{H}_{CN}$ textures from original $\mathbf{H}$ matrix and $\mathbf{H}_{BN}$

```
1:      /* Reading a binary M × N matrix H */
2: for all BN_n (columns in H_mn) : do
3:    for all CN_m (rows in H_mn) : do
4:       if H_mn == 1 then
5:          ptr_tmp = i : H_in == 1, with m + 1 ≤ i < (m + M) mod M;
                /* Finding circularly the neighbor below on the current column */
6:          ptr_next =search(H_BN, ptr_tmp, n);
                /* Finding in H_BN the pixel with indices (ptr_tmp, n) */
7:          H_CN = ptr_next;
                /* Store ptr_next into the H_CN structure, with addresses compatible with H_BN, using a square texture
            of dimension 2^D × 2^D, with D = ⌈ 1/2 × log2( Σ_{m=1}^{M} Σ_{n=1}^{N} H_mn ) ⌉ */
8:       end if
9:    end for
10: end for
```

For the vertical processing, $\mathbf{H}_{CN}$ is 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). This data structure is generated by scanning the $\mathbf{H}$ matrix in a column major order. Once again, the access between adjacent elements is circular, as described

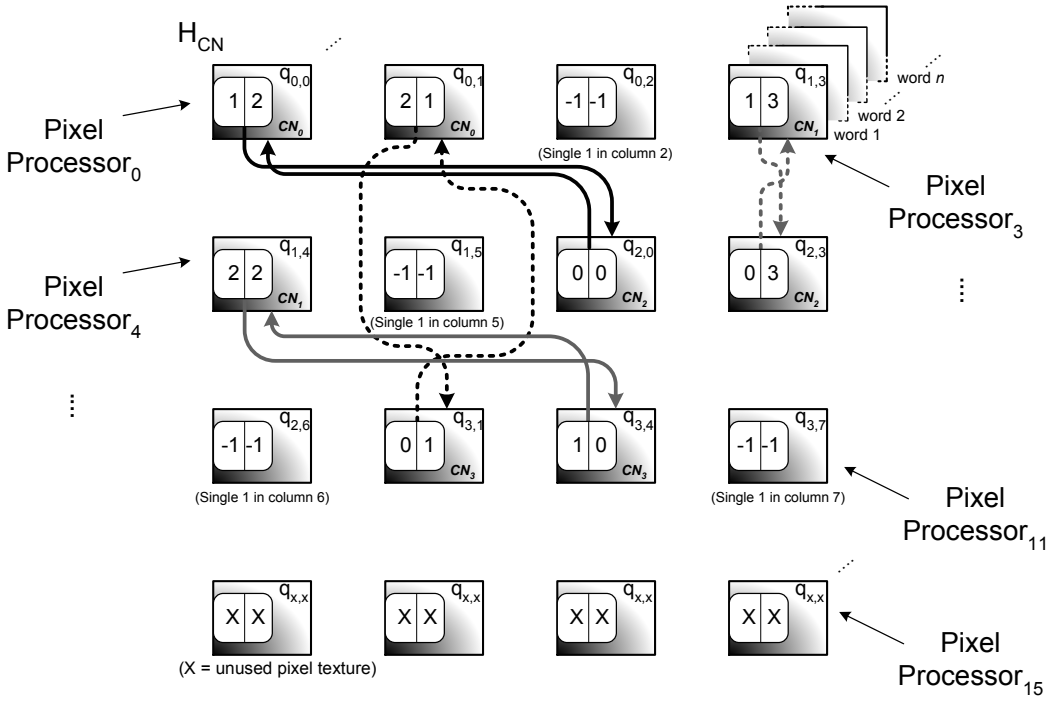


Figure 5.15: $\mathbf{H}_{CN}$ structure. A 2D texture representing Check Node edges with circular addressing for the example in figure 2.1. Also, the pixel processors entry points are shown.

in algorithm 5.9 and illustrated in figure 5.15 for the $\mathbf{H}$ matrix given in figure 2.1. In this case, a careful construction of the 2D addresses in $\mathbf{H}_{CN}$ is required, because every pixel texture representing a graph edge must be in exactly the same position as it is in $\mathbf{H}_{BN}$. This meticulous positioning of the pixel elements in $\mathbf{H}_{CN}$ allows the processing to be performed alternately for both kernels, using the same input textures. Step 6 shows that $ptr_{next}$ is placed in the same pixel texture (or $n, m$ edge) that it occupies in $\mathbf{H}_{BN}$.

Figure 5.15 describes how the $\mathbf{H}_{CN}$ data structure is organized for the same example in figure 2.1, under kernel 2. The message $m^{i}_{BN_0 \rightarrow CN_0}$ (identified as message $q_{0,0}$) is a function of $p_0, m^{i-1}_{CN_2 \rightarrow BN_0}$ and should update the upper left pixel representing $CN_0$, which holds the address of $CN_2$. This is another way of saying that $CN_2$ updates $CN_0$, and vice-versa. This mechanism works in the same way for all the other CNs in the grid.

**LDPC decoders on the Caravela platform:** We developed a flow-model to support the LDPC decoder based on Caravela tools, which uses efficient mechanisms provided for recursive computation [129]. The SDF graph in figure 5.7 represents the stream-based LDPC decoder and figure 5.16 graphically describes the corresponding flow-model unit containing a *shader* program that supports the stream-based computation of kernels 1 and 2, where the input and output data streams are 2D textures. In the first iteration, the input data stream 0 represents data channel probabilities. The first output stream is produced

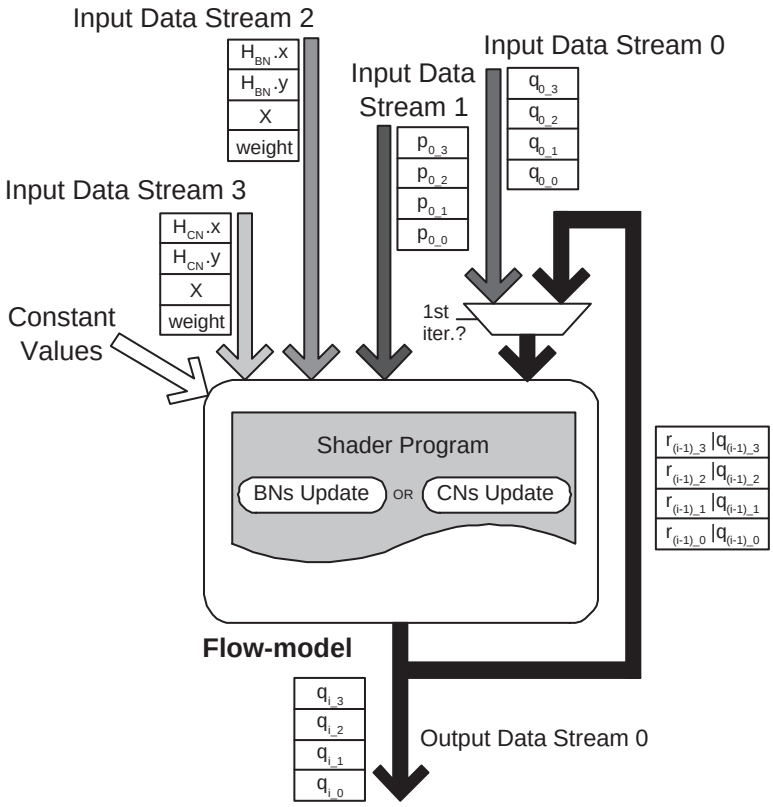


Figure 5.16: Organization of the LDPC decoder flow-model.

by performing kernel 1. After the first execution, this stream directly feeds the input of the next flow-model unit that executes kernel 2. Data streams can be multiplexed through a simple and efficient swapping mechanism[129]. The output data stream can be feedback as an input stream of the next flow-model unit execution, and the process is repeated for each iteration. Also, the input **H** matrix is placed into an input stream and kernel 1 and kernel 2 are processed in pixel processors, according to figures 5.14, 5.15 and 5.16. At the end, the last output stream conveys the decoded codeword.

### 5.6.2 Experimental results with Caravela

**Experimental setup:** The proposed algorithm was programmed in recent GPUs and its performance is analyzed for different workloads (i.e. **H** matrices with distinct characteristics). The experimental setup is presented in table 5.11. It includes an 8800 GTX GPU from NVIDIA, with stream processors (SPs) running at 1.35 GHz. The used GPU supports single precision floating-point arithmetic. The LDPC decoders are programmed on the GPU using version 2.0 of the OpenGL Shading Language and the Caravela library. The experiments were carried out by using five matrices of different sizes and with varying number of edges, matrices c1 to c5 in table 5.12. Their properties were chosen to

Table 5.11: Experimental setup for Caravela

| | **GPU** |
|---|---|
| Platform | NVIDIA 8800 GTX |
| Clock frequency | 1.35 GHz (SP) |
| Memory | 768 MB |
| Language | OpenGL (GLSL) |

Table 5.12: Parity-check matrices **H** under test for Caravela

| **Matrix** | **Size** | **Edges** | **Edges/row** | **Texture Dim.** | **Unused pixel textures**[8] |
|---|---|---|---|---|---|
| c1 | $111 \times 999$ | 2997 | {27} | $64 \times 64$ | 1099 |
| c2 | $408 \times 816$ | 4080 | {10} | $64 \times 64$ | 16 |
| c3 | $212 \times 1908$ | 7632 | {36} | $128 \times 128$ | 8752 |
| c4 | $2448 \times 4896$ | 14688 | {6} | $128 \times 128$ | 1696 |
| c5 | $2000 \times 4000$ | 16000 | {8} | $128 \times 128$ | 384 |

simulate a full set of computational workloads with code sizes typically ranging from small to medium and large (all sizes covered), and approximately similar to those used in recent communication standards. The pixel processors of the GPU were used with data inputted as textures and the output data stream assuming the usual place of the pixel color components' output in a graphical application. However, all programming details are hidden from the programmer by using the Caravela interface tool.

**Experimental results programming the flow-model on the GPU using Caravela:** The experimental results in figure 5.17 show the processing times obtained to decode the LDPC codes from table 5.12. The performance of the GPU increases as the intensity of processing also increases, but not necessarily just depending on it. Comparing matrix c1 with matrix c2, it can be shown that even though the latter has fewer unused pixel textures that represent no edges in the Tanner graph, the former has a better ratio *decoding time*/*Edge* executing 25 iterations. Although matrix c1 has more edges per row than c2 (27 against 10), GPUs perform better for algorithms demanding intensive computation and small amount of data communications. Experimental results in figure 5.17 show that the GPU performs better for large matrices. Although matrix c5 has a number of edges almost four times superior to c2, and even with a similar number of edges per row (8 against 10), the ratio *decoding time*/*Edge* is favorable in matrix c5. It performs more than two times better than the latter ($3\mu$s against $7.3\mu$s).

The GPU-based approach shows better results for the LDPC decoding algorithm with

[8]These are the pixels showing empty coordinates (x, x) in figures 5.14 and 5.15, imposed by the GPU Caravela interface that only supports 2D data textures with square dimensions $D \times D$, where $D$ is a power of 2.

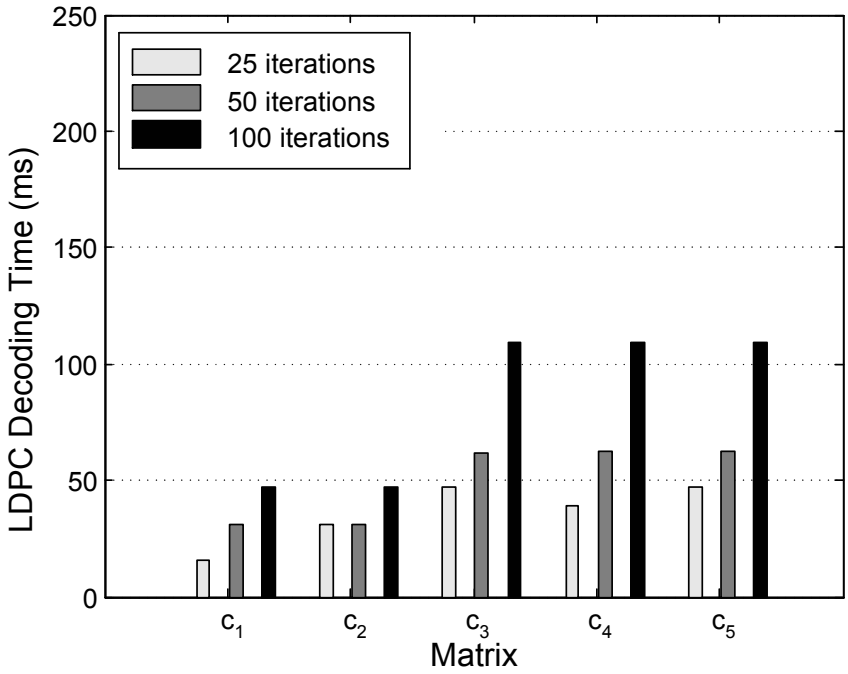


Figure 5.17: Decoding times running the SPA on an 8800 GTX GPU from NVIDIA programmed with Caravela.

intensive computation on huge quantities of data, due to parallelism features and processing power.

### 5.6.3 LDPC decoding on CUDA-based platforms

A different approach consists of exploiting a many-core system with a programming model based on multi-threads, as the one supported by the NVIDIA Tesla series' GPUs using the CUDA [80]. Differently from previous GPU families which could only program pixel processors, the CUDA provides a more efficient exploitation of the computational resources of a GPU. Here, geometry, pixel and vertex programs share common stream processing resources, and the GPUs used in this new approach are based on this unified architecture (CUDA).

In order to achieve LDPC decoding supported by CUDA, algorithms 2.1 and 2.2 were applied to these parallel programmable devices as described next. The initial approach followed on CUDA-based GPUs was similar to the one described previously under the Caravela context in section 5.6.1. The processing adopted was also based on a thread-per-edge approach [32,37]. This solution presents the same penalties as those mentioned in section 5.6.1, because it does not use the efficient forward-and-backward method that allow data reutilization and save memory accesses. Furthermore, the algorithms presented in [32,37] are based on the SPA. Better results are achieved using the more efficient MSA,

that only has to perform add and compare operations. To overcome these difficulties we adopted a different approach based on a thread-per-node approach (thread-per-CN and thread-per-BN based processing)[34].

**Algorithm 5.10** GPU algorithm (host side)

1: Allocate *buffers* in the host and *buffers* on device (cudamalloc)
2: Read $\mathbf{H_{BN}}$ and $\mathbf{H_{CN}}$ data structures (edges' addresses) from file to the host *buffers*
3: Send $\mathbf{H_{BN}}$ and $\mathbf{H_{CN}}$ data structures to *buffers* on the GPU device
4: Configure the grid on device (number of blocks and number of threads per block)
5: **repeat**
6: Receives $y_n$ from the channel and calculate $p_n$ ($Lp_n$)
7: Send $p_n$ ($Lp_n$) data to *buffers* on the GPU device
8: **for** $i = 1$ to $N\_Iter$: **do**
9: Launch kernel 1 (horizontal processing) for the calculation of $r_{mn}$ ($Lr_{mn}$) data on the GPU
10: Launch kernel 2 (vertical processing) for the calculation of $q_{nm}$ ($Lq_{nm}$) data on the GPU
11: **end for**
12: Transfer final $Q_n$ ($LQ_n$) values to the host
13: **until** true

**Multi-thread based programming on the CUDA:** Unlike the Caravela solution, the CUDA-based approach recovers data structures $\mathbf{H_{BN}}$ and $\mathbf{H_{CN}}$ originally presented in subsection 3.3.3. Algorithm 5.10 illustrates how processing is performed in this approach. In the beginning of a new processing, after the necessary data buffers are allocated both on host and device, data to be computed ($\mathbf{H_{BN}}$, $\mathbf{H_{CN}}$, $r_{mn}$ ($Lr_{mn}$) and $q_{nm}$ ($Lq_{nm}$) data structures) is distributed over a predefined number of blocks on the grid of multi-threads. Each block contains $tx \times ty$ elements that represent threads. Threads are grouped in warps and dispatched by one of the 128 stream processors according to the thread-per-CN or thread-per-BN structure of the algorithm and the thread scheduler, as mentioned before. In the horizontal processing step, each thread updates all the BNs associated to a CN (an entire row of $\mathbf{H}$). Rather than moving data from global to fast (but complex) shared memory, an equally efficient processing strategy can be achieved by moving data directly to the registers of the core. The number of registers available per multiprocessor on the GPU is high enough to accommodate the input data and corresponding data structures. Increasing the number of threads per block often allows hiding latency and achieving better performance, as long as we guarantee that there are enough registers per thread on the GPU to compile. For every different application we need to estimate the best fitting number of threads per block. The minimum consists of 64, but 128 or 256 threads per block can be used until a maximum of 512. Figure 5.18 depicts the internal processing inside a block, where the update of every BN message represented by $r_{mn}$ ($Lr_{mn}$) is calculated according to the structure previously identified in $\mathbf{H_{BN}}$.

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

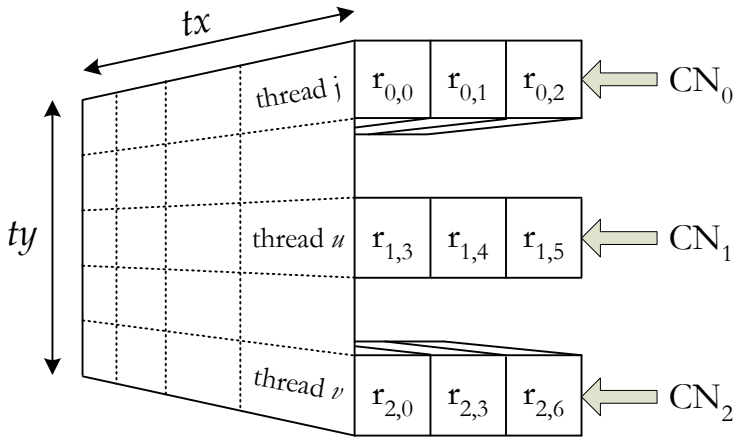


Figure 5.18: Detail of $tx \times ty$ threads executing on the GPU grid inside a block for kernel 1 (the example refers to figure 3.12 and shows BNs associated to $CN_0$, $CN_1$ and $CN_2$ being updated).

**Coalesced memory accesses:** In the 8 series' GPU 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 consists in using 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 the elements have to lie on a contiguous memory block, where the $k^{th}$ thread accesses the $k^{th}$ data element, and data and addresses must obey, respectively, to specific size and alignment requirements[34]. Figure 5.19 shows the activity of half-warps 0 and 1 captured at two different instants, where 16 threads ($t_0$ to $t_{15}$) read data in a single coalesced memory transaction from the GPU's slow global memory. However, $\mathbf{H_{BN}}$, $\mathbf{H_{CN}}$, $r_{mn}$ ($Lr_{mn}$) and $q_{nm}$ ($Lq_{nm}$) data structures 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. This data realignment action is stated in Proposition 1 and depicted in figure 5.19 which shows the simultaneous access of groups of 16 threads to memory.

**Proposition 1.** *For decoding regular LDPC codes using the proposed stream-based data structures shown in figure 3.12, it is possible to compute each kernel of algorithms 2.1 and 2.2 using full coalesced memory accesses for either read or write operations, but not for both.*

Proposition 1 is fulfilled by applying the following transformations to the data structures. Equations (5.3) to (5.6) represent the permutations necessary for kernel 1, where:

$$newAddr = j * p + k \ \text{ div } \ w_c + (k \ \text{ mod } \ w_c) * 16, \tag{5.3}$$

with $p = w_c * 16$, $j = i$ div $p$, $k = i$ mod $p$ and $0 \leq i \leq Edges - 1$. Then,

$$\breve{\mathbf{q}}^c_{nm}(i) = \mathbf{q}_{nm}(newAddr), \tag{5.4}$$

and the new memory addresses become:

$$e = \mathbf{H}_{CN}(newAddr), \tag{5.5}$$

$$\breve{\mathbf{H}}^c_{CN}(i) = j * p + k \text{ div } w_b + (k \text{ mod } w_b) * 16, \tag{5.6}$$

with $p = w_b * 16$, $j = e$ div $p$, and $k = e$ mod $p$. The permutations necessary for kernel 2 are described from (5.7) to (5.10), where:

$$newAddr = j * p + k \text{ div } w_b + (k \text{ mod } w_b) * 16, \tag{5.7}$$

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

$$\breve{\mathbf{r}}^c_{mn}(i) = \mathbf{r}_{mn}(newAddr). \tag{5.8}$$

The new permuted memory addresses become:

$$e = \mathbf{H}_{BN}(newAddr), \tag{5.9}$$

$$\breve{\mathbf{H}}^c_{BN}(i) = j * p + k \text{ div } w_c + (k \text{ mod } w_c) * 16, \tag{5.10}$$

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

Although the number of read and write operations is similar, the solution here proposed adopts coalesced memory read accesses. Unfortunately, due to the pseudo-random nature of LDPC codes, it is not possible to perform coalesced accesses for both operations.

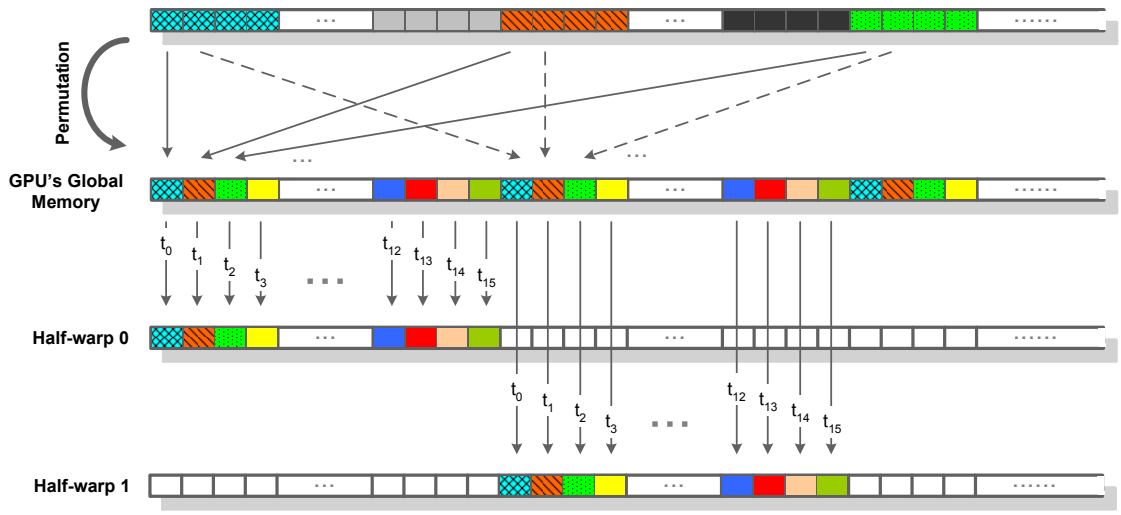


Figure 5.19: Transformations applied to data structures to support coalesced memory read operations performed by the 16 threads of a half-warp on a single memory access.

The recent GT200 GPUs from NVIDIA have less demanding requirements for coalescence than those just discussed for the G80x family. Nevertheless, hand tuned code for coalesced memory accesses on the G80x series family scales well on a GT200 device.

### 5.6.4 Experimental results for regular codes with CUDA

**Experimental setup:** The parallel processing platforms selected to perform LDPC decoding are two NVIDIA Tesla GPUs, respectively with 128 and 240 stream processors (SP), each running at 1.35 GHz and 1.296 GHz and having 768 MByte and 4GByte of video RAM (VRAM) memory. The experimental setups are depicted in table 5.13. Matrices under test are presented in table 5.14 and range from matrix t1 to matrix t6: the regular codes t1 to t4 have $w_c = 6$ edges per row and $w_b = 3$ edges per column, while matrices t5 and t6 represent irregular codes used in Digital Video Broadcasting – Satellite 2 (DVB-S2)[29]. Once again, we considered the compact data structures presented in figures 3.12 and 3.13 to represent the Tanner graph. The program is written in C and uses single precision floating-point arithmetic operations. The GPU under Setup 1 was programmed using the CUDA interface (version 2.0b), while the more recent GPU in Setup 2 already uses CUDA version 2.3.

Table 5.13: Experimental setups for the Tesla GPUs with CUDA

| | **Setup 1** | **Setup 2** |
|---|---|---|
| Platform | NVIDIA 8800 GTX | NVIDIA C1060 |
| Language | C + CUDA | |
| OS | MS Windows XP | Linux (Ubuntu) kernel 2.6.28 |
| # Multiprocessors | 16 | 30 |
| # Stream processors | 128 | 240 |
| # Registers per multiprocessor | 8192 | 16384 |
| Clock speed | 1.35 GHz (SP) | 1.296 GHz (SP) |
| VRAM Memory | 768 MB | 4 GB |

Table 5.14: Parity-check matrices **H** under test for the CUDA

| **Matrix** | **Size** $(M \times N)$ | **Edges** $(w_c \times M)$ | **Edges/row** $(w_c)$ | **Edges/col.)** $(w_b)$ |
|---|---|---|---|---|
| t1 | $512 \times 1024$ | 3072 | {6} | {3} |
| t2 | $2448 \times 4896$ | 14688 | {6} | {3} |
| t3 | $4000 \times 8000$ | 24000 | {6} | {3} |
| t4 | $10000 \times 20000$ | 60000 | {6} | {3} |
| t5 | $48600 \times 64800$ | 194400 | {4} | {3,12} |
| t6 | $32400 \times 64800$ | 226800 | {7} | {3,8} |
| t7 | $25920 \times 64800$ | 285120 | {11} | {3,12} |

**Experimental results on the GPU using CUDA:** All regular matrices under test were run for Setup 1, on blocks with a varying number of threads ranging from 64 to 128. Only the best results achieved are reported in tables 5.15 and 5.16. Matrix t1 was programmed to use 16 blocks of the grid and 64 threads per block. Using more threads per block would have been possible, but it would decrease the number of simultaneous active blocks to a value below 16 which would cause the 16 multiprocessors not to be fully occupied, reducing parallelism and performance. Matrix t2 occupies 128 threads per block and 39 blocks. Matrix t3 uses 63 blocks and 128 threads per block and matrix t4 157 blocks and also 128 threads per block. The higher level of parallelism expectedly supported by the next GPU generations, advises the use of at least 100 blocks in order to exploit full potential and make the solution scalable to future devices with more stream processors. In this case, 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 in the LDPC decoder for Setup 1 are limited to 128 due to the high number of registers used per multiprocessor.

Table 5.15 presents the overall measured decoding throughputs running the SPA 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/B.E.-based one for LDPC codes with dimensions above matrix t1. The GPU-based implementation shows significantly higher throughputs for intensive computation on large quantities of data. A throughput above 18.3 Mbps is achieved for matrix t3 with 24000 edges and executing 10 iterations, while it decreases to 11.3 Mbps for significantly larger codes (60000 edges and above). In the latter, the size of the data structures to represent the matrix is so large that they do not fit into the 64 KByte of constant memory used in smaller codes (t3 and below), which degrades the performance because constant memory on the GPU uses a different bus from global memory. For small codes and with a small number of iterations, better results are achieved with fewer threads per block, although the decoding time differences are minimal if we vary the number of threads per block. However, when decoding larger codes with a higher number of iterations, the reported speedups are higher when a larger number of threads per block

Table 5.15: LDPC decoding throughputs for a CUDA programming environment running the SPA with regular codes (Mbps)

| Number of iterations | Matrix t1 | | Matrix t2 | | Matrix t3 | | Matrix t4 | |
|---|---|---|---|---|---|---|---|---|
| | 32-bit | 8-bit | 32-bit | 8-bit | 32-bit | 8-bit | 32-it | 8-bit |
| 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 |

is used. Figure 5.20 shows the importance of memory accesses in the overall processing

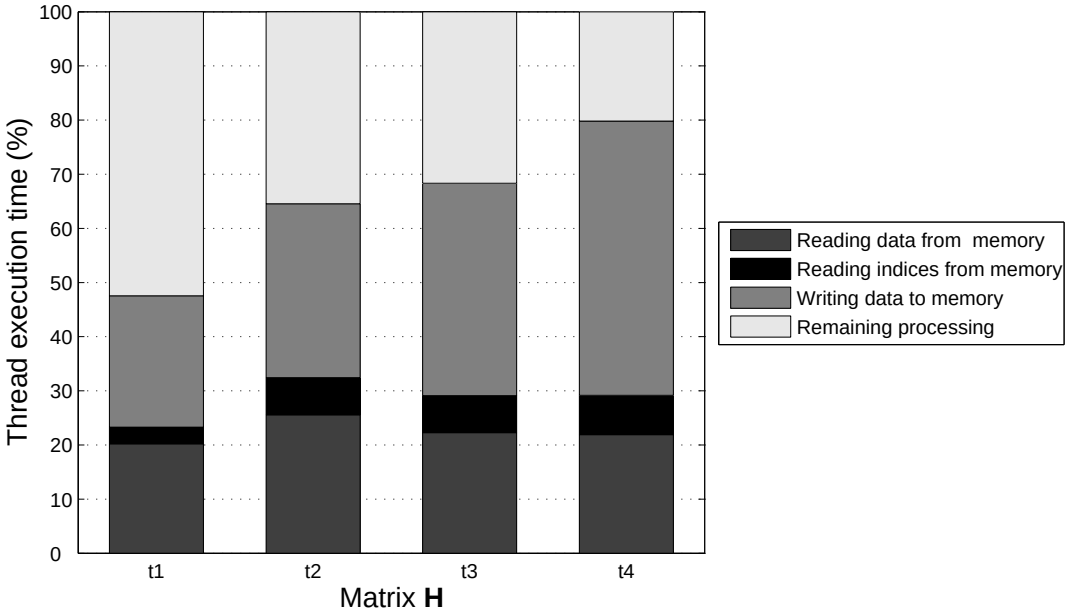


Figure 5.20: Thread execution times for LDPC decoding on the GPU running the SPA.

time as the size of the LDPC code increases. Reports obtained for matrices t1 to t4 running on the GPU show that the time spent in coalesced memory reads is approximately constant (in percentage) for all matrices, while the more expensive and often conflicting non-coalesced memory write operations can represent a bottleneck after a certain dimension of the LDPC code. Figure 5.20 shows that because we use coalescence to read data, if the number of edges changes, the percentage of the overall processing time remains approximately constant at around 20%. For matrix t4 (60000 edges), non-coalesced write accesses occupy nearly 50% of the total processing time, against 39% for matrix t3 (24000 edges), and only 24% in matrix t1 (3072 edges). This shows that using coalescence as described in Proposition 1, is extremely important not only because parallel accesses to memory can be performed by multiple threads concurrently, but also because the number of data collisions decreases.

**Data precision that best fits the arithmetic intensity on the GPU:** We also tested the LDPC decoder for the SPA using data with only 8-bit precision, instead of the original 32-bit precision, which allowed us to pack 16 elements into 128-bit data elements. In this case, using the 8-bit precision solution, the throughputs obtained for 10 iterations in matrices t1 to t4 range from 14.6 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 is unpacked and processed, causing one memory access to

Table 5.16: LDPC decoding times on the GPU (ms) and corresponding throughputs (Mbps) for an 8-bit data precision solution decoding 16 codewords in parallel running the MSA with regular codes.

| | Time (ms) | | | | Throughput (Mbps) | | | |
|---|---|---|---|---|---|---|---|---|
| **Number of iterations** | 10 | 20 | 30 | 50 | 10 | 20 | 30 | 50 |
| Matrix t1 | 0.393 | 0.597 | 0.765 | 1.102 | 41.7 | 27.4 | 21.4 | 14.9 |
| Matrix t3 | 1.331 | 1.589 | 1.812 | 2.261 | 96.2 | 80.6 | 70.6 | 56.6 |
| Matrix t4 | 2.932 | 3.176 | 3.425 | 3.841 | 109.1 | 100.8 | 93.4 | 83.3 |

read/write 16 elements at a time. Even considering the increase in arithmetic operations, the GPU still performs faster. The limitation in performance is imposed mainly by memory accesses. The highest throughputs are achieved for matrices t3 and t4, which provide results superior to 40 Mbps. Due to the use of the efficient forward and backward algorithm[83] that minimizes the number of memory accesses and also the number of add and multiply operations, the experimental results here reported are much superior to the ones obtained in[37].

Another approach based on CUDA exploits the MSA. The results achieved with the MSA are reported in table 5.16 and are superior to those obtained with the SPA. The matrices under test are t1, t3 and t4. As referred before, matrix t1 uses 64 threads per block and a total of 16 blocks. Matrices t3 and t4 use 128 threads per block and 63 and 157 blocks, respectively. The decoding times reported in table 5.16 define global processing times, which include data transfers according to the denominator of (5.12), to simultaneously decode 16 codewords with length depending on the matrix. Data elements are represented and processed using 8-bit precision, causing 16 data elements to be read in each 128-bit memory access. The GPU seems to perform better for larger codes. If we consider throughput, it is interesting to see that the GPU achieves a higher coded data rate for matrix t4, which has 60000 edges to compute. In this case, a very significant throughput of 109.1 Mbps is achieved running 10 iterations. It decodes 16 codewords simultaneously, each of length 20000-bit, in 2.932 ms.

**Analyzing the performance on the GPU with CUDA:** A model to predict the throughput $T$ of the parallel LDPC decoder on the GPU is presented in (5.12), 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 non-divergent 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:

$$T_{proc} = N_{iter} \frac{\frac{Th}{MP} \times (\frac{ND_{op/iter}}{SP} + D_{op/iter}) + Th \times Mop \times L}{f_{op}}. \quad (5.11)$$

Then, the global throughput can be obtained by:

$$T = \frac{P \times N}{T_{host \to gpu} + T_{proc} + T_{gpu \to host}} \quad [bps], \quad (5.12)$$

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

The model defined in (5.11) 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 non-coalesced 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 t4, 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% of divergent threads which allow us to approximate $D_{op/iter} \approx 0$, (5.11) can be rewritten as:

$$T_{proc} = \frac{2N_{iter} \times \frac{Th}{MP} \times \frac{ND_{op/iter}}{SP}}{f_{op}} + \underbrace{\frac{Th \times Mop \times L}{f_{op}}}_{T_{MemAccess}}. \quad (5.13)$$

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

$$T_{MemAccess} = T_{MemAccBNs} + T_{MemAccCNs}. \quad (5.14)$$

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

$$T_{MemAccBNs} = N_{iter} \frac{M \times w_c \times L + M \times 2w_c \times L/16}{f_{op}}, \quad (5.15)$$

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

$$T_{MemAccCNs} = N_{iter} \frac{N \times w_b \times L + N \times (2w_b + 1) \times L/16}{f_{op}}. \quad (5.16)$$

The last partials in (5.15) and (5.16) 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 maintaining the cores occupied and hiding the memory latency. For code t4 our model predicts the processing time quite well, with an error inferior to 13% assuming $L = 4$. Transferring data structures between host and device at a bandwidth of 1.2 GB/s (an experimentally measured value) represents an increase in 8.9% on the global processing time.

### 5.6.5 Experimental results for irregular codes with CUDA

The number of edges of LDPC codes used, for example, in DVB-S2 assumes significant proportions, as depicted from table 2.3. For every iteration, the number of messages to be updated is twice the value shown in field *#Edges*. For example, when running 10 iterations of the decoder for code t7, which has $rate = 3/5$, a total of $2 \times 285120 \times 10 = 5702400 > 5.7M$ messages have to be processed in a short period of time in order to guarantee minimum acceptable throughput. Until very recently, such performances were only possible to achieve using dedicated VLSI, as it is the case of the hardware solution proposed in chapter 4 of this thesis. However, considering the high processing power and bandwidth of GPUs, in this chapter we show that with these programmable devices it is possible to achieve performances similar to those obtained with VLSI hardware.

**Running LDPC DVB-S2 codes on the GPU with CUDA:** Codes t5, t6 and t7 from table 5.14 are a subset of DVB-S2 codes. They represent long length irregular codes with maximal and minimal number of edges (see table 2.3 in chapter 2) used in the standard, and for these reasons were selected to be decoded on a platform based on Setup 2, which has more multiprocessors available than Setup 1. They represent, respectively, B1, B4 and B5 irregular LDPC codes which can be found in annexes B and C of the standard [29]. Although Setup 2 has more registers per multiprocessor than Setup 1, the number of threads allowed per block still has limitations. In this case, the LDPC decoder is limited to 256 threads per block due to the high number of registers that each kernel uses per multiprocessor. The GPU from Setup 2 has 240 stream processors and 30 multiprocessors and the horizontal and vertical kernels demand less than 76 registers per thread. In this case this is acceptable because the C1060 GPU allows to use up until 16384 registers per multiprocessor as shown in table 5.13, and the quantity of threads used per block keep the total number of registers bellow that value. Each block of the GPU was tested with a varying number of threads ranging from 64 to 256, and the best results were achieved with 128 threads per block for codes t5 and t6, while 192 threads per block produce better results for code t7. As for the solutions based on Setup 1, so does the number of blocks used here is imposed by the size of the LDPC code and number of threads per block adopted. Ma-

Table 5.17: LDPC decoding times on the GPU (ms) and corresponding throughputs (Mbps) for an 8-bit data precision solution decoding 16 codewords in parallel running the MSA with irregular DVB-S2 codes.

| Matrix | Iter. | $T_{host \rightarrow gpu} + T_{gpu \rightarrow host}$ (ms) | $T_{proc}$ (ms) | Throughput (Mbps) |
|---|---|---|---|---|
| t5 | 1 | 3.1 | 1.99 | 205.7 |
| | 2 | | 3.97 | 147.6 |
| | 5 | | 9.90 | 80.1 |
| | 10 | | 19.85 | 45.3 |
| | 20 | | 39.66 | 24.3 |
| | 50 | | 99.00 | 10.2 |
| t6 | 1 | 3.4 | 2.30 | 180.7 |
| | 2 | | 4.58 | 129.4 |
| | 5 | | 11.40 | 69.9 |
| | 10 | | 22.85 | 39.4 |
| | 20 | | 45.72 | 21.1 |
| | 50 | | 114.12 | 8.8 |
| t7 | 1 | 4.1 | 3.87 | 129.7 |
| | 2 | | 7.68 | 88.0 |
| | 5 | | 19.20 | 44.5 |
| | 10 | | 38.5 | 24.3 |
| | 20 | | 76.72 | 12.8 |
| | 50 | | 192.09 | 5.3 |

trices t5 and t6 were programmed to use 507 blocks of the grid, while matrix t7 occupies 338 blocks. Table 5.17 presents the decoding times and throughputs obtained running the MSA on a C1060 GPU (described in Setup 2, table 5.13). Experimental results reported in the table show that the GPU-based solution here proposed supports the intensive processing of LDPC codes with very high dimensions within acceptable decoding times. By executing 1 iteration of the decoder for matrix t5 (which has 194400 edges), the solution achieves a processing time of 1.99 ms and a throughput of 205.7 Mbps, which respectively change to 19.85 ms and 45.3 Mbps when executing 10 iterations. The increase in processing time is not linear because for small $T_{proc}$ processing times, $T_{host \rightarrow gpu} + T_{gpu \rightarrow host}$ data transfer times still have a significant weight in the overall processing time. Matrix t6 has a higher number of edges (226800), which allows obtaining 180.7 Mbps executing 1 iteration and 39.9 Mbps for 10 iterations. Matrix t7 represents the DVB-S2 code with the highest number of edges (285120) and in this case throughput decreases to 129.7 Mbps for 1 iteration and to 24.3 Mbps executing 10 iterations of the algorithm.

Additionally, this programmable solution proposed for GPUs also considers different data representations, namely by using 8 or 16-bit precision. Table 5.18 presents throughputs for both cases running code t6, where the 8-bit solution decodes 16 codewords in

Table 5.18: Comparing LDPC decoding throughputs for DVB-S2 code t6 running the MSA with 8- and 16-bit data precision solutions (Mbps)

| Number of iterations | Matrix t6 | |
|---|---|---|
| | Throughput (Mbps) | |
| | 8-bit | 16-bit |
| 1 | 180.7 | 86.5 |
| 2 | 129.4 | 60.6 |
| 3 | 100.7 | 48.7 |
| 4 | 82.6 | 38.2 |
| 5 | 69.9 | 32.2 |
| 10 | 39.4 | 17.9 |
| 20 | 21.1 | 9.6 |
| 50 | 8.8 | 4.0 |

parallel, and the 16-bit solution only decodes 8, producing lower coded data rates. It is interesting to observe that even using the 16-bit solution we still achieve high throughputs for this class of computationally demanding DVB-S2 codes. Recently, platforms that cooperatively allow running multiple GPUs in simultaneous have been introduced. Applying the LDPC decoder kernels developed under the scope of this thesis to such platforms (some solutions incorporate 4 GPUs, supplying a total of 960 cores), most certainly would allow to increase the aggregate throughput even further. The 16-bit solution here reported shows that as GPUs' processing power and bandwidth increase, additional coding gains can be exploited, namely regarding Bit Error Rate (BER), by adopting higher resolution to represent data.

## 5.7 Discussion of architectures and models for parallel LDPC decoding

The different approaches presented in this chapter for several parallel computing architectures reflect the complexity of LDPC decoders associated with restrictions imposed by parallel computing architectures. Some of them proved to be able of competing with hardware-dedicated solutions that typically need allocating high resources to develop this kind of projects and use Non-Recurring Engineering (NRE).

Even by exploiting fast cache memory shared by multiple cores in parallel algorithms specifically developed for computation on general-purpose x86 multi-cores, the throughputs achieved are far from those requested by real-time applications. Nevertheless, the general-purpose approach on x86 multi-cores allowed to conclude that the solutions proposed under the context of this work are scalable.

By exploiting data locality that minimizes memory access conflicts, and by using

Table 5.19: Comparison with state-of-the-art ASIC LDPC decoders using the MSA.

| | Rate (K/N) | Code length (N) | Precision (bits) | Iterations | Throughput (Mbps) |
|---|---|---|---|---|---|
| [108] | 5/6 | 2304 | 8 | 10 | 30.4 |
| [81] | 1/2 | 2304 | 6 | 20 | 63 |
| [111] | 1/2 | 2304 | 8 | 8 | 60 |
| Matrix t3 [GPU proposed solution] | 1/2 | 8000 | 8 | 10 | 96.2 |

several SPE cores that support SIMD and a dual pipelined architecture, the Cell/B.E. achieves throughputs between 70 and 80 Mbps for small to medium LDPC codes. The limitation in performance for larger codes is imposed by the size of the LS memory on the SPEs. The adoption of specific parallelization techniques for the Cell/B.E. platform here described produced throughputs that approach well hardware solutions[81,108]. For the SPA, it nearly achieves 70 Mbps[38] running 10 iterations of the algorithm. Moreover, we also implemented LDPC decoders based on the MSA for the Cell/B.E., using codes adopted in the WiMAX standard (IEEE 802.16e)[64], which have lengths inferior to 2304 bits. We achieved throughputs near 80 Mbps running 10 iterations[33,36] of the decoder, which approaches throughputs that until recently were only achievable using dedicated hardware (e.g. VLSI). Furthermore, this solution can provide a lower BER regarding VLSI-based architectures.

GPUs, which are shared memory based multiprocessors, can also be programmed with Caravela and CUDA. The former is a general, while the latter is an efficient programming framework developed for NVIDIA devices. These GPUs are more efficient than Cell/B.E. for LDPC decoding medium and large length codes. We tested a vast set of codes where we included LDPC codes used in the DVB-S2 standard and report throughputs ranging from dozens of Mbps to over 100 Mbps, depending on the length of the code or number of multiprocessors of the GPU adopted. Here, the constraints are imposed by bandwidth limitations between host and device, and also by conflicting memory accesses to the global memory of the GPU, which can be minimized by using coalesced read/write operations that significantly improve the performance of the algorithm. The adoption of specific parallelization strategies for the GPU platform here described, produced throughputs that approach well hardware-based solutions. Table 5.19 presents some comparisons with recently published work. The throughputs obtained compare well with those reported in recent publications[17,81,108,111] that describe dedicated hardware solutions (ASIC and others) to implement LDPC decoders.

Figure 3.3 shows that the average number of iterations depends on the Signal-to-

Noise Ratio (SNR) of input data, which naturally influences the decoding time. It can be seen that the 8-bit precision solution here developed for the CUDA compares favorably in terms of the number of iterations needed for the algorithm to converge. Again, as in the case of Cell/B.E.-based LDPC decoders, this programmable solution also presents superior performance in terms of BER, when compared with ASIC architectures that use 5 to 6-bit precision to represent data.

## 5.8 Summary

Changes in technical computing now provide massive computational power at low-cost. The advent of high-value graphics computing offers the possibility of increased productivity and flexibility on a variety of intensive algorithms used in real-time demanding applications.

This chapter presents the most relevant results obtained for LDPC decoding on multi- and many-core platforms. These platforms can be divided into three distinct classes: general-purpose multi-cores based on the x86 shared memory architecture; the Cell/B.E. from the STI consortium which is based on a distributed memory model; and GPUs, also based on a shared memory model and here exploited under the Caravela and CUDA contexts. They allow exposing distinct parallel properties of an LDPC code. We developed LDPC decoding algorithms and data structures appropriate for parallel computing. They are used to measure the performance under these parallel computing platforms. The dissimilarities between the architectures and the comparison of performances are presented in this chapter. We conclude that some of them offer performances that approach well hardware-dedicated (e. g. VLSI) solutions. We also present in this chapter models that try to estimate their behavior under controlled parameters. These estimates allow to conclude that both architectures can be programmed to work very close from their peak performance.

*We shall not cease from exploration*
*And the end of all our exploring*
*Will be to arrive where we started*
*And know the place for the first time.*

*T.S. Eliot, in "Four Quartets"*

# 6

# Closure

Contents

Due to the computationally intensive nature of Low-Density Parity-Check (LDPC) decoders, hardware approaches have been developed over the last years. They mainly use Very Large Scale Integration (VLSI) technology, which is able to efficiently handle the complex processing required by LDPC decoding in order to provide high throughputs.

The purposes of the research work developed under the scope of this thesis were threefold. The first consisted of developing and optimizing VLSI parallel LDPC decoder systems with a reduced number of processors. The second objective was to investigate new paradigms, parallel algorithms and suitable stream-based data structures able of performing LDPC decoding efficiently. The third and last purpose of this research was to derive parallel kernels for a set of predefined multi-core architectures and assess their performance against state-of-the-art VLSI solutions.

In this thesis it has been shown that such hardware solutions for LDPC decoding can be implemented with a reduced number of processors supported by a minor reconfiguration of the memory blocks of the system. Using a small number of processors significantly simplifies the routing complexity of the design, allowing savings in terms of area, development costs and complexity. Small process design technologies used in Application Specific Integrated Circuits (ASIC) (90nm and below) allow to operate this architecture at frequencies high enough to guarantee throughputs able of supporting the demanding Digital Video Broadcasting – Satellite 2 (DVB-S2) requirements. The smallest area result achieved in this work is approximately 7 $mm^2$. This research has also been conducted to Field Programmable Gate Array (FPGA) based solutions that are equally able of providing real-time processing for the same requirements, and led to the following publications:

[31] Falcão, G., Gomes, M., Gonçalves, J., Faia, P., and Silva, V. (2006). HDL Library of Processing Units for an Automatic LDPC Decoder Design. In *Proceedings of the IEEE Ph.D. Research in Microelectronics and Electronics (PRIME'06)*, pages 349–352.

[51] Gomes, M., Falcão, G., Silva, V., Ferreira, V., Sengo, A., and Falcão, M. (2007b). Flexible Parallel Architecture for DVB-S2 LDPC Decoders. In *Proceedings of the IEEE Global Telecommunications Conf. (GLOBECOM'07)*, pages 3265–3269.

While dedicated hardware still imposes some restrictions, the advent of the multi-core era encouraged the development of flexible and programmable approaches towards the computation of LDPC decoding. Presently, a set of parallel architectures exist that are worldwide disseminated and generally available at low-cost. The Cell Broadband Engine (Cell/B.E.) architecture suits ideally the decoding of codes with small to medium lengths, while Graphics Processing Units (GPU) can be used more efficiently for medium to large length codes. Parallel algorithms were proposed for performing LDPC decoding by exploiting data parallelism and task parallelism. The results herein reported show that

the former can be used to decode efficiently WiMAX LDPC codes achieving throughputs near 80 Mbps, while the latter can be adopted to decode larger codes, namely those used in the DVB-S2 standard. Both support decoding either regular or irregular codes. This work was presented internationally in journals and conferences:

[36] Falcão, G., Silva, V., Sousa, L., and Marinho, J. (2008). High coded data rate and multicodeword WiMAX LDPC decoding on the Cell/BE. *Electronics Letters*, 44(24):1415–1417.

[38] Falcão, G., Sousa, L., and Silva, V. (2009c). Parallel LDPC Decoding on the Cell/B.E. Processor. In *Proceedings of the 4th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC'09)*, volume 5409 of *Lecture Notes in Computer Science*, pages 389–403. Springer.

GPUs can be used under different contexts and by using different programming tools. They can use the Caravela interface, a generic tool for programming kernels on GPUs, which allows the programmer to use any GPU existing on the market, as reported in the following journal paper:

[41] Falcão, G., Yamagiwa, S., Silva, V., and Sousa, L. (2009d). Parallel LDPC Decoding on GPUs using a Stream-based Computing Approach. *Journal of Computer Science and Technology*, 24(5):913–924.

Or, to achieve superior performance and lower execution times, GPUs from NVIDIA can also be programmed using the Compute Unified Device Architecture (CUDA) interface, which supports kernels running under a multi-threaded environment with multiple levels of efficient memory hierarchy. This part of the work performed with CUDA produced throughputs in the order of hundreds of Mbps and has been presented in some of the best conferences in the area:

[37] Falcão, G., Sousa, L., and Silva, V. (2008). Massive Parallel LDPC Decoding on GPU. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP'08)*, pages 83–90, Salt Lake City, Utah, USA. ACM.

[34] Falcão, G., Silva, V., and Sousa, L. (2009b). How GPUs can outperform ASICs for fast LDPC decoding. In *Proceedings of the 23rd ACM International Conference on Supercomputing (ICS'09)*, pages 390–399. ACM.

Also, power consumption plays an important role in nowadays autonomous systems. The more efficient is the exploitation of the parallel computational resources, the better the ratio Watt-per-bit-decoded can be, which implies that the number of codewords decoded in parallel should be the highest. However, to achieve high aggregate throughput, the execution on GPUs can impose significant latency, which may become unacceptable

for some communications. However, other important areas of research can adopt GPUs as convenient solutions for the intensive problem of LDPC decoding. One example is related with the study and simulation of new LDPC codes and error floors in the range of $10^{-11}$ to $10^{-12}$ (or even further), which typically is not adequate to be achieved by using dedicated hardware (e.g. VLSI), because it demands flexibility and programmability. This type of research is based on the massive simulation of LDPC codes and it can be efficiently achieved on programmable processors such as GPUs, which can accelerate the computation on CPUs. The GPU-based solutions also support floating-point arithmetic, which can introduce advantages in terms of Bit Error Rate (BER) regarding to VLSI approaches:

[39] Falcão, G., Sousa, L., and Silva, V. (accepted in February 2010b). Massively LDPC Decoding on Multicore Architectures. *IEEE Transactions on Parallel and Distributed Systems*.

Furthermore, by developing models that try to estimate the behavior of these architectures, we also show that if programmed according to specific and appropriate strategies targeting high levels of parallelism, LDPC decoding algorithms can perform very close to the theoretical maximum performance of those architectures. The assessment of their performances, namely BER and throughput, shows that they can be competitive with VLSI hardware.

The impact of the research work developed under the scope of this thesis can be measured not only by the publications that arose from it but also from the high number of citations to this very recent work. Recently, a chapter has been accepted for publication at the *GPU Computing Gems* from NVIDIA, and the source code has been made publicly available at the *gpucomputing.net* website:

[35] Falcão, G., Silva, V., and Sousa, L. (2010a). *GPU Computing Gems*, chapter Parallel LDPC Decoding. *ed.* Wen-mei Hwu, vol. 1, NVIDIA, Morgan Kaufmann, Elsevier.

This allows other researchers/engineers to continue this work. By making the source code publicly available, people with practical interests in LDPC decoders can develop their own systems and research. Another purpose of this strategy aims at encouraging the scientific/engineering community to develop other types of parallel algorithms, namely by applying adequate strategies inspired in this work for obtaining high levels of parallelism able of producing more efficient and faster programs.

## 6.1 Future work

Although all optimizations performed in the VLSI architecture proposed in chapter 4 of this thesis, there is still room for improvement. Using smaller technologies in the process design, such as 65, 45, or even 32 nm, can minimize area and power consumption.

Taking advantage of higher frequencies of operation (possible with the use of smaller process technologies) and with a significantly higher effort, deeper changes can be introduced into the architecture by grouping more memories into common blocks of higher and thinner RAMs, using a more complex time multiplexing strategy. As described in chapter 4, this can lead to considerable area improvements. Additional improvements can also be projected to the near future by exploiting the use of memristors, a recent technology of HP that is expected to revolutionize the manufacturing of RAM memories, namely by introducing important changes in area and volatility constraints.

CUDA has been proposed to perform LDPC decoding under the context of this thesis, but another Application Programming Interface (API) has recently emerged with the arrival of OpenCL[55], and consequently it should be exploited. OpenCL allows writing parallel programs once for execution across several heterogeneous platforms that include CPUs, GPUs and other types of processors (in practice, OpenCL code has minor change requirements as we move the execution from platform to platform).

Another area potentially exploitable consists of using GPU clusters for the massive processing of LDPC decoding essentially based on data-parallelism. Here, the main challenges should be imposed by the shared use of the bus for communications with the host system. In fact, preliminary tests showed that when performing such iterative computations in multi-GPU systems, the communication time dominates.

Finally, the utilization of heterogeneous systems that may incorporate, for example, CPUs, GPUs and FPGAs to cooperatively compute LDPC decoding can also be considered. Expectedly, it may allow obtaining interesting results, if we successfully try to overcome the limitations of an architecture with the facilities provided by another.

It should be noticed that, although the work here reported was mainly devoted to the study of computational models and development of a novel concept for LDPC decoding based on newly proposed parallel algorithms, it can be applied in the development of improved and faster algorithms that support other types of computationally demanding applications. Examples are inference calculation algorithms, such as those used in Turbo codes, stereo vision applied to robotics, or in Bayesian networks just to name a few. The methods and solutions proposed in this thesis show that the use of distinct forms of parallelism to deal with the intensive nature of these computational problems appears to be tractable.

# Appendices

# A

# Factorizable LDPC Decoder Architecture for DVB-S2

Contents

The architecture here described surpasses some limitations of [70], and adds flexibility and easy reconfiguration, according with the decoder constraints [51].

The design of Irregular Repeat Accumulate (IRA) Low-Density Parity-Check (LDPC) codes used in the Digital Video Broadcasting – Satellite 2 (DVB-S2) standard has been addresses in chapter 2. The pseudo-random generation of the sparse **A** matrix described in (2.33) allows a significant reduction on the storage requirements without a significant code performance loss. The two types of nodes that form the bipartite Tanner graph are check nodes ($\nu^C$), one per each code constraint, and bit nodes, one per each codeword bit (information and parity, respectively, $\nu^I$ and $\nu^P$). The construction technique used to generate matrix **A** is based on the separation of the $\nu^I$ nodes into disjoint groups of $M$ consecutive ones, with $M = 360$. All the $\nu^I$ nodes of a group $l$ should have the same weight $w_l$, and it is only necessary to choose the $\nu^C$ nodes that connect to the first $\nu^I$ of the group, in order to specify the $\nu^C$ nodes that connect to each one of the remaining $M-1$ $\nu^I$ nodes. The choice of the first element of group $l$ to connect is pseudo-random and has the following restrictions: the resulting LDPC code is cycle-4 free; the number of length 6 cycles is the shortest possible; and all the $\nu^C$ nodes must connect to the same number of $\nu^I$ nodes. Denoting by $r_1, r_2, ..., r_{w_l}$, the indices of the $\nu^C$ nodes that connect to the first $\nu^I$ of group $l$, the indices of the $\nu^C$ nodes that connect to $\nu_i^I$, with $0 \leq i \leq M-1$, of group $l$ can be obtained by (2.34), with $q$ given by (2.35). The factor $M$ is constant for all codes used in the DVB-S2 standard. For each code, **A** has groups of $\nu^I$ nodes with constant weights $w_c > 3$, and also groups with weights $w_c = 3$. Matrix **B** has a lower triangle staircase profile as shown in (2.33).

## A.1 $M$ parallel processing units

This turns possible the simultaneous processing of $\nu^I$ and $\nu^C$ node sets, whose indices are given by:

$$\mathrm{C}^{(c)} = \{c,\ c+1,\ \cdots,\ c+M-1\},\ \text{ with }\ c \bmod M = 0 \tag{A.1}$$

and

$$\mathrm{R}^{(r)} = \{r,\ r+q,\ r+2q, \cdots,\ r+(M-1)q\},\ \text{ with }\ 0 \leq r \leq q-1, \tag{A.2}$$

respectively, (the superscript is the index of the first element of the set and, $r$ and $c$ mean row and column of **H**), which significantly simplifies the decoder control. In fact, according to (2.34), if $\nu_{\tilde{c}}^I$ is connected to $\nu_r^C$, then $\nu_{r+i\times q}^C$, with $0 \leq i \leq M-1$, will be connected to $\nu_{c+(\tilde{c}-c+i) \bmod M}^{\mathrm{I}}$, where $c = M \times (\tilde{c} \text{ div } M)$ represents the index of the first $\nu^{\mathrm{I}}$ of the group $C^{(c)}$ to which $\nu_{\tilde{c}}^I$ belongs.

The architecture shown in figure 4.2 is based on $M = 360$ Functional Units (FU) working in parallel with shared control signals [49], that process both $\nu^{\mathrm{C}}$ (in check mode) and $\nu^{\mathrm{I}}$

nodes (in bit mode) in a flooding schedule manner. $\nu^{\mathrm{C}}$ and $\nu^{\mathrm{P}}$ nodes are updated jointly in check mode following an horizontal schedule approach, according to the zigzag connectivity [49] established between these two different type of nodes.

### A.1.1 Memory mapping and shuffling mechanism

As mentioned before, a single FU unit is shared by a constant number of $\nu^{\mathrm{I}}$, $\nu^{\mathrm{C}}$ and $\nu^{\mathrm{P}}$ nodes (the last two are processed jointly), which depend on the code length and rate. More specifically, for a $(n,k)$ DVB-S2 LDPC-IRA code (due to notation particularities and for a question of style, in chapter 2 variables $n,k$ are represented by $N,K$), the $\mathrm{FU}_i$, with $0 \leq i \leq M-1$, updates sequentially in bit mode the $\nu^{\mathrm{I}}_{\{i,i+M,i+2\times M,\cdots,i+(\alpha-1)\times M\}}$ nodes, with $\alpha = k/M$. In check mode, the same FU updates the $\nu^{\mathrm{C}}_{\{j,j+1,\cdots,j+q-1\}}$ and $\nu^{\mathrm{P}}_{\{j,j+1,\cdots,j+q-1\}}$ nodes, with $j = i \times q$. This guarantees that when processing simultaneously group $\mathrm{C}^{(c)}$, the computed messages have as destination a set $\mathrm{R}^{(r)}$, where each one of them will be processed by a different FU. According to (2.34), the new computed messages only need to be right rotated to be handled by the correct $\nu^{\mathrm{C}}$ nodes. The same happens when processing each $\mathrm{R}^{(r)}$ set, where as considered in (2.34), the right rotation must be reversed in order to the new computed messages have as destination the exact $\nu^{\mathrm{I}}$ nodes. The shuffling network (barrel shifter) is responsible for the correct exchange of messages between $\nu^{\mathrm{I}}$ and $\nu^{\mathrm{C}}$ nodes, emulating the Tanner graph. The SHIFTS values stored in the ROM memory of figure 4.2 can be easily obtained from the annexes B and C of DVB-S2 standard tables [29]. The messages sent along the Tanner graph's edges are stored in RAM memories of figure 4.2. If we adopt a sequential RAM access in bit mode, then the access in check mode must be indexed, or vice-versa. Both options are valid and consequently, without loss of generalization, we assume sequential access in bit mode. Denoting by $\mathbf{r}_i = [r_{i1}\ r_{i2}\ \cdots\ r_{iw_i}]^T$ the vector of $\nu^{\mathrm{C}}$ node indices connected to the $\nu^I_{\mathrm{i}}$ node of weight $w_i$, the message memory mapping can be obtained using the following matrix:

$$\mathbf{R} = \begin{bmatrix} \mathbf{r}_0 & \mathbf{r}_1 & \cdots & \mathbf{r}_{M-1} \\ \mathbf{r}_M & \mathbf{r}_{M+1} & \cdots & \mathbf{r}_{2M-1} \\ \vdots & \vdots & \vdots & \vdots \\ \mathbf{r}_{(\alpha-1)\times M} & \mathbf{r}_{(\alpha-1)\times M+1} & \cdots & \mathbf{r}_{\alpha\times M-1} \end{bmatrix}_{(q\times w_C)\times M}, \tag{A.3}$$

where $w_C$ is a code related parameter ($\nu^{\mathrm{C}}$ weight is $w_C+2$, except for the first row as shown in (2.33)). In order to process each $\mathrm{R}^{(r)}$ set in check mode, the required memory addresses can be obtained by finding the rows in matrix $\mathbf{R}$ where the index $r$ appears.

## A.2 Decoding decomposition by a factor of $M$

The simplicity of the shuffling mechanism and the efficient memory mapping scheme represent the major strengths of the architecture in[70]. However, the high number of FUs and the long width of the barrel shifter require a significant silicon area. Since this architecture can provide a throughput far above the minimum mandatory rate of 90 Mbps for DVB-S2, the number of FUs can be reduced. In fact, this can be accomplished by any factor of $M$.

Let $L, N \in \mathbb{N}$ be factors of $M$ , with $M = L \times N$, and consider a $C^{(c)}$ set as in (A.1). This group can be decomposed by subsampling in $L$ subgroups, according to:

$$\mathrm{C}_0^{(c)} = \{c,\ c+L,\ c+2L,\ \cdots,\ c+(N-1)\times L\} \tag{A.4}$$

and

$$\mathrm{C}_1^{(c)} = \{c+1,\ c+1+L,\ c+1+2L,\ \cdots,\ c+1+(N-1)\times L\}\,, \tag{A.5}$$

until the last subgroup

$$\mathrm{C}_{L-1}^{(c)} = \{c+L-1,\ c+2L-1,\ c+3L-1,\ \cdots,\ c+N\times L-1\}\,. \tag{A.6}$$

Each sub-group $C^{(c)}$, with $0 \leq \gamma \leq L-1$, can be described in terms of the first node of the subgroup $v^{\mathrm{I}}_{c+\gamma}$, according to (2.34). If $v^{\mathrm{C}}_r$ is connected to the first information node of the subgroup $C^{(c)}_\gamma$, then $v^{\mathrm{C}}_{(r+i\times L\times q) \bmod (n-k)}$ is connected to the $i$-th $v^I$ node of the referred subgroup, with $0 \leq i \leq N-1$.

Again, the same subsampling process by $L$ can be performed on each $\mathrm{R}^{(r)}$ group, according to:

$$\mathrm{R}_0^{(r)} = \{r,\ r+L\times q,\ r+2L\times q,\ \cdots,\ r+(N-1)\times L\times q\} \tag{A.7}$$

and

$$\mathrm{R}_1^{(r)} = \{r+q,\ r+(L+1)\times q,\ r+(2L+1)\times q,\ \cdots,\ r+((N-1)\times L+1)\times q\}\,, \tag{A.8}$$

until

$$\mathrm{R}_{L-1}^{(r)} = \{r+(L-1)\times q,\ r+(2L-1)\times q,\ r+(3L-1)\times q,\ \cdots,\ r+(N\times L-1)\times q\}\,. \tag{A.9}$$

Similarly, each subgroup $\mathrm{R}^{(r)}_\beta$, with $0 \leq \beta \leq L-1$, can be described in terms of the first element $v^{\mathrm{C}}_{r+\beta\times q}$. If $v^I_{\tilde{c}}$ is connected to the first node of subset $\mathrm{R}^{(r)}_\beta$, then $v^{\mathrm{I}}_{c+((\tilde{c}-c+i\times L) \bmod M)'}$ with $c = M\times(\tilde{c} \operatorname{div} M)$, is connected to the $i$-th $v^{\mathrm{C}}$ node, with $0 \leq i \leq N-1$, of the considered subgroup.

From the analysis of (A.4) to (A.9), we conclude that the subsampling approach preserves the key modulo $M$ properties and, thus, we can process individually each $C_{\gamma}^{(c)}$ and $R_{\beta}^{(r)}$ subgroup, and the same architecture in [70] can be used with only $N$ processing units as shown in figure 4.4 (in figure 4.4 the variable $N$ is represented by $P$). In fact, when processing simultaneously a $C_{\gamma}^{(c)}$ group, the computed messages have as destination a set $R_{\beta}^{(r)}$, and vice-versa.

### A.2.1 Memory mapping and shuffling mechanism

The subsampling strategy allows a linear reduction (by a factor of $L$) of the hardware resources occupied by the FUs blocks, reduces significantly the complexity of the barrel shifter ($O(N \log_2 N)$) and simplifies the routing problem. Yet, at first glance, it may seem that this strategy implies an increase by $L$ in the size of the system ROM (SHIFTS and ADDRESSES in figures 4.2 and 4.4). Fortunately, if we know the properties of the subgroups $C_0^{(c)}$ and $R_0^{(r)}$, we can automatically find the properties of the remaining subgroups $C_{\gamma}^{(c)}$ and $R_{\beta}^{(r)}$, respectively, with $0 \leq \alpha, \beta \leq N-1$. By using a proper message memory mapping based on a convenient reshape by $L$ of matrix $R$ in (A.3), of the form:

$$\mathbf{R} = \left[\begin{array}{cccc}
\mathbf{r}_0 & \mathbf{r}_L & \cdots & \mathbf{r}_{(N-1)\times L} \\
\mathbf{r}_M & \mathbf{r}_{M+L} & \cdots & \mathbf{r}_{M+(N-1)\times L} \\
\vdots & \vdots & & \vdots \\
\mathbf{r}_{(\alpha-1)\times M} & \mathbf{r}_{(\alpha-1)\times M+L} & \cdots & \mathbf{r}_{(\alpha-1)\times M+(N-1)\times L} \\
\hline
\mathbf{r}_1 & \mathbf{r}_{L+1} & \cdots & \mathbf{r}_{(N-1)\times L+1} \\
\mathbf{r}_{M+1} & \mathbf{r}_{M+L+1} & \cdots & \mathbf{r}_{M+(N-1)\times L+1} \\
\vdots & \vdots & & \vdots \\
\mathbf{r}_{(\alpha-1)\times M+1} & \mathbf{r}_{(\alpha-1)\times M+L+1} & \cdots & \mathbf{r}_{(\alpha-1)\times M+(N-1)\times L+1} \\
\hline
\vdots & \vdots & & \vdots \\
\hline
\mathbf{r}_{L-1} & \mathbf{r}_{2L-1} & \cdots & \mathbf{r}_{M-1} \\
\mathbf{r}_{M+L-1} & \mathbf{r}_{M+2L-1} & \cdots & \mathbf{r}_{2M-1} \\
\vdots & \vdots & & \vdots \\
\mathbf{r}_{(\alpha-1)\times M+L-1} & \mathbf{r}_{(\alpha-1)\times M+2L-1} & \cdots & \mathbf{r}_{\alpha\times M-1}
\end{array}\right]_{(L\times q\times w_C)\times N} \tag{A.10}$$

we can keep unchanged the size of the system ROM and compute on the fly the new SHIFTS values as a function of those previously stored in the ROM memory of figures 4.2 and 4.4 for all $C_0^{(c)}$ groups, as:

$$shift_{\gamma}^{(c)} = (shift_0^{(c)} + \gamma) \text{ div } L, \tag{A.11}$$

and, similarly, the new addresses are given by:

$$address_{\beta}^{(r)} = (address_0^{(r)} + q \times w_C \times \beta) \mod (q \times w_C \times L), \tag{A.12}$$

where $address_0^{(r)}$ represent the ADDRESSES' values stored in ROM for all $R_0^{(r)}$ groups.

The shift addresses stored in ROM are the same for both architectures of figures 4.2 and 4.4. Yet, due to reshaping of matrix **R**, memory addresses contained in $R_0^{(r)}$ change, but they can be easily obtained following the procedure described in the previous subsection, i. e., by finding in matrix **R** the rows where index $r$ appears.

For the configuration shown in figure 4.4, each $FU_i$, with $0 \leq i \leq N-1$, is now responsible for processing $L \times \alpha$ information nodes in the following order:

$$\begin{array}{l} \{i, i+M, i+2M, \cdots, i+(\alpha-1)M\ ; \\ i+1, i+1+M, \cdots, i+1+(\alpha-1)M\ ; \\ \cdots \quad ; \\ i+L-1, i+L-1+M, \cdots, i+L-1+(\alpha-1)M\} \end{array} , \tag{A.13}$$

and $L \times q$ check and parity nodes $\{j, j+1, \cdots, j+L \times q-1\}$, with $j = i \times L \times q$.

# Bibliography


[1] Abbasfar, A. (2007). *Turbo-like Codes – Design for High Speed Decoding*. Springer.

[2] Abellán, J., Fernández, J., and Acacio, M. (2008). CellStats: a Tool to Evaluate the Basic Synchronization and Communication operations of the Cell BE. In *Proceedings of the 16th Euromicro International Conference on Parallel, Distributed and network-based Processing (PDP'08)*, pages 261–268.

[3] Aji, A., Feng, W.-C., Blagojevic, F., and Nikolopoulos, D. (2008). Cell-SWat: Modeling and Scheduling Wavefront Computations on the Cell Broadband Engine. In *Proceedings of the 5th Conference on Computing Frontiers (CF'08)*, pages 13–22. ACM.

[4] Al-Kiswany, S., Gharaibeh, A., Santos-Neto, E., and Ripeanu, M. (2009). On GPU's viability as a middleware accelerator. *Cluster Computing*.

[5] ATI/AMD (2007). HT 2999 XT ATI/AMD homepage. http://ati.amd.com/products/Radeonhd2900/index.html.

[6] ATI/AMD (2009). CTM homepage. http://ati.amd.com/companyinfo/researcher/documents/ati_ctm_guide.pdf.

[7] Benes, V. E. (1964). A Turbo-Decoding Message-Passing Algorithm for Sparse Parity-Check Matrix Codes. *Bell System Technical Journal*, 43:1641–1656.

[8] Bentz, J. L. and Kendall, R. A. (2005). *Shared Memory Parallel Programming with Open MP*. Springer Berlin / Heidelberg.

[9] Berrou, C. and Glavieux, A. (1996). Near optimum error correcting coding and decoding: turbo-codes. *IEEE Transactions on Communications*, 44(10):1261–1271.

[10] Berrou, C., Glavieux, A., and Thitimajshima, P. (1993). Near Shannon limit error-correcting coding and decoding: Turbo-codes (1). In *Proceedings of the IEEE International Conference on Communications (ICC'93)*, pages 1064–1070. IEEE.

[11] Beuschel, C. and Pfleiderer, H.-J. (2008). FPGA implementation of a flexible decoder for long LDPC codes. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL 2008)*, pages 185–190. IEEE.

[12] Blake, G., Dreslinski, R. G., and Mudge, T. (2009). A Survey of Multicore Processors. *IEEE Signal Processing Magazine*, 26(6):26–37.

[13] Blanksby, A. J. and Howland, C. J. (2002). A 690-mW 1-Gb/s 1024-b, Rate-1/2 Low-Density Parity-Check Code Decoder. *Journal of Solid-State Circuits*, 37(3):404–412.

[14] Blume, H., Livonius, J. v., Rotenberg, L., Noll, T. G., Bothe, H., and Brakensiek, J. (2008). OpenMP-based parallelization on an MPCore multiprocessor platform – A performance and power analysis. *Journal of Systems Architecture*, 54(11):1019–1029.

[15] Bolz, J., Farmer, I., Grinspun, E., and Schroder, P. (2003). Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM Transactions on Graphics*, 22(3):917–924.

[16] Borkar, S. (2007). Thousand Core Chips–A Technology Perspective. In *Proceedings of the 44th annual conference on Design automation (DAC'07)*, pages 746–749. ACM.

[17] Brack, T., Alles, M., Kienle, F., and Wehn, N. (2006). A Synthesizable IP Core for WIMAX 802.16E LDPC Code Decoding. In *Proceedings of the IEEE 17th International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC'06)*, pages 1–5. IEEE.

[18] Brack, T., Alles, M., Lehnigk-Emden, T., Kienle, F., Wehn, N., L'Insalata, N. E., Rossi, F., Rovini, M., and Fanucci, L. (2007). Low Complexity LDPC Code Decoders for Next Generation Standards. In *Proceedings of Design, Automation and Test in Europe, 2007 (DATE'07)*, pages 1–6. IEEE.

[19] Brunton, A., Shu, C., and Roth, G. (2006). Belief Propagation on the GPU for Stereo Vision. In *Proceedings of the 3rd Canadian Conference on Computer and Robot Vision (CRV06)*, pages 76–76.

[20] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P. (2004). Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786.

[21] Chapman, B., Jost, G., and Van Der Pass, R. (2008). *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific Computation and Engineering Series)*. The MIT Press.

[22] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., and Skadron, K. (2008). A performance study of general-purpose applications on graphics processors using CUDA. *J. Parallel and Distrib. Comput.*, 68(10):1370–1380.

[23] Chung, S., Forney, G., Richardson, T., and Urbanke, R. (2001). On the Design of Low-Density Parity-Check Codes within 0.0045 dB of the Shannon Limit. *IEEE Communications Letters*, 5(2):58–60.

[24] Corporation, I. B. M. (2006). CELL Broadband Engine Architecture. *IBM Journal of Research and Development*.

[25] Culler, D. E., Singh, J. P., and Gupta, A. (1998). *Parallel Computer Architecture – A Hardware/Software Approach*. Morgan Kaufmann.

[26] Demontes, L., Bonaciu, M., and Amblard, P. (2008). Software for Multi Processor System on Chip: Moving an MPEG4 Decoder from Generic RISC Platforms to CELL. In *Proceedings of the IEEE/IFIP 19th International Symposium on Rapid System Prototyping (RSP'08)*, pages 34–40.

[27] Dielissen, J., Hekstra, A., and Berg, V. (2006). Low cost LDPC decoder for DVB-S2. In *Proceedings of Design, Automation and Test in Europe, 2006 (DATE'06)*, pages 1–6. IEEE.

[28] Elias, P. (1954). Error-free Coding. *IRE Professional Group on Information Theory*, 4(4):29–37.

[29] EN 302 307 V1. 1.1, European Telecommunications Standards Institute (ETSI) (2005). . Digital video broadcasting (DVB); second generation framing structure, channel coding and modulation systems for broadcasting, interactive services, news gathering and other broad-band satellite applications.

[30] Eroz, M., Sun, F. W., and Lee, L. N. (2004). DVB-S2 low density parity check codes with near Shannon limit performance. *International Journal of Satellite Communications and Networking*, 22:269–279.

[31] Falcão, G., Gomes, M., Gonçalves, J., Faia, P., and Silva, V. (2006). HDL Library of Processing Units for an Automatic LDPC Decoder Design. In *Proceedings of the IEEE Ph.D. Research in Microelectronics and Electronics (PRIME'06)*, pages 349–352.

[32] Falcão, G., Silva, V., Gomes, M., and Sousa, L. (2008). Edge Stream Oriented LDPC Decoding. In *Proceedings of the 16th Euromicro International Conference on Parallel, Distributed and network-based Processing (PDP'08)*, pages 237–244, Toulouse, France.

[33] Falcão, G., Silva, V., Marinho, J., and Sousa, L. (2009a). *WIMAX, New Developments*, chapter LDPC Decoders for the WiMAX (IEEE 802.16e) based on Multicore Architectures. In-Tech.

[34] Falcão, G., Silva, V., and Sousa, L. (2009b). How GPUs can outperform ASICs for fast LDPC decoding. In *Proceedings of the 23rd ACM International Conference on Supercomputing (ICS'09)*, pages 390–399. ACM.

[35] Falcão, G., Silva, V., and Sousa, L. (2010a). *GPU Computing Gems*, chapter Parallel LDPC Decoding. *ed.* Wen-mei Hwu, vol. 1, NVIDIA, Morgan Kaufmann, Elsevier.

[36] Falcão, G., Silva, V., Sousa, L., and Marinho, J. (2008). High coded data rate and multicodeword WiMAX LDPC decoding on the Cell/BE. *Electronics Letters*, 44(24):1415–1417.

[37] Falcão, G., Sousa, L., and Silva, V. (2008). Massive Parallel LDPC Decoding on GPU. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP'08)*, pages 83–90, Salt Lake City, Utah, USA. ACM.

[38] Falcão, G., Sousa, L., and Silva, V. (2009c). Parallel LDPC Decoding on the Cell/B.E. Processor. In *Proceedings of the 4th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC'09)*, volume 5409 of *Lecture Notes in Computer Science*, pages 389–403. Springer.

[39] Falcão, G., Sousa, L., and Silva, V. (accepted in February 2010b). Massively LDPC Decoding on Multicore Architectures. *IEEE Transactions on Parallel and Distributed Systems*.

[40] Falcão, G., Yamagiwa, S., Silva, V., and Sousa, L. (2007). Stream-Based LDPC Decoding on GPUs. In *Proceedings of the First Workshop on General Purpose Processing on Graphics Processing Units – GPGPU'07*, pages 1–7.

[41] Falcão, G., Yamagiwa, S., Silva, V., and Sousa, L. (2009d). Parallel LDPC Decoding on GPUs using a Stream-based Computing Approach. *Journal of Computer Science and Technology*, 24(5):913–924.

[42] Fan, Z., Qiu, F., Kaufman, A., and Yoakum-Stover, S. (2004). GPU Cluster for High Performance Computing. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing (SC'04)*, pages 47–47. IEEE Computer Society.

[43] Fanucci, L. and Rossi, F. (2004). A throughput/complexity analysis for the VLSI implementation of LDPC decoder. In *Proceedings of the Fourth IEEE International Symposium on Signal Processing and Information Technology*, pages 409–412. IEEE.

[44] Fanucci, L. and Rossi, F. (2006). Interconnection framework for high-throughput, flexible LDPC decoders. In *Proceedings of the Conference on Design, Automation and Test in Europe, 2006 (DATE'06)*, pages 1–6. IEEE.

[45] Fok, K.-L., Wong, T.-T., and Wong, M.-L. (2007). Evolutionary computing on consumer graphics hardware. *IEEE Intelligent Systems*, 22(2):69–78.

[46] Gallager, R. G. (1962). Low-Density Parity-Check Codes. *IRE Transactions on Information Theory*, 8(1):21–28.

[47] Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y., and Volkov, V. (2008). Parallel Computing Experiences with CUDA. *IEEE Micro*, 28(4):13–27.

[48] Ghuloum, A., Sprangle, E., Fang, J., Wu, G., and Zhou, X. (2007). Ct: A Flexible Parallel Programming Model for Tera-scale Architectures. *Intel*, pages 1–21.

[49] Gomes, M., Falcão, G., Gonçalves, J., Faia, P., and Silva, V. (2006a). HDL Library of Processing Units for Generic and DVB-S2 LDPC Decoding. In *Proceedings of the International Conference on Signal Processing and Multimedia Applications (SIGMAP'06)*.

[50] Gomes, M., Falcão, G., Silva, V., Ferreira, V., Sengo, A., and Falcão, M. (2007a). Factorizable modulo M parallel architecture for DVB-S2 LDPC decoding. In *Proceedings of the 6th Conference on Telecommunications (CONFTELE'07)*.

[51] Gomes, M., Falcão, G., Silva, V., Ferreira, V., Sengo, A., and Falcão, M. (2007b). Flexible Parallel Architecture for DVB-S2 LDPC Decoders. In *Proceedings of the IEEE Global Telecommunications Conf. (GLOBECOM'07)*, pages 3265–3269.

[52] Gomes, M., Falcão, G., Silva, V., Ferreira, V., Sengo, A., Silva, L., Marques, N., and Falcão, M. (2008). Scalable and Parallel Codec Architectures for the DVB-S2 FEC System. In *Proceedings of the IEEE Asia Pacific Conf. on Circuits and Systems (APCCAS'08)*, pages 1506–1509.

[53] Gomes, M., Silva, V., Neves, C., and Marques, R. (2006b). Serial LDPC Decoding on a SIMD DSP using Horizontal Scheduling. In *Proceedings of the European Signal Processing Conf. (EUSIPCO'06)*.

[54] Goodnight, N., Wang, R., and Humphreys, G. (2005). Computation on programmable graphics hardware. *IEEE Computer Graphics and Applications*, 25(5):12–15.

[55] Group, K. (2010). OpenCL – The open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/.

[56] Guilloud, F., Boutillon, E., and Danger, J.-L. (2003). $\lambda$-min decoding algorithm of regular and irregular LDPC codes. In *Proceedings of the 3rd Int. Symp. Turbo Codes Relat. Topics*, pages 1–4.

[57] Halfhill, T. (2008). Parallel Processing with CUDA. *Microprocessor Report*.

[58] Held, J., Bautista, J., and Koehl, S. (2006). From a few cores to many: A tera-scale computing research overview. *Research at Intel white paper*, pages 1–11.

[59] Hennessy, J. and Patterson, D. (2006). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann.

[60] Hofstee, H. (2005). Power Efficient Processor Architecture and the Cell Processor. In *Proceedings of the 11th International Symposium on High-Performance Computer Architectures (HPCA'05)*, pages 258–262. IEEE.

[61] Howland, C. and Blanksby, A. (2001). Parallel decoding architectures for low density parity check codes. In *Proceedings of the 2001 IEEE International Symposium on Circuits and Systems*, pages 742–745. IEEE.

[62] Hu, X. and Kumar, B. V. K. V. (2007). Evaluation of Low-Density Parity-Check Codes on Perpendicular Magnetic Recording Model. *IEEE Transactions on Magnetics*, 43(2):727–732.

[63] Hu, X.-Y., Eleftheriou, E., Arnold, D.-M., and Dholakia, A. (2001). Efficient Implementations of the Sum-Product Algorithm for Decoding LDPC Codes. In *Proceedings of the IEEE Global Telecommunications Conf. (GLOBECOM'01)*, pages 1036–1036E.

[64] IEEE (2005). IEEE 802.16e. IEEE P802.16e/D12: 'Draft IEEE Standard for Local and Metropolitan Area Networks. Part 16: Air Interface for Fixed and Mobile Broadband Wireless Access Systems', October 2005.

[65] Intel (2008). Intel® Xeon® Processor 7000 Sequence homepage. http://www.intel.com/p/en_US/products/server/processor /xeon7000?iid=servproc+body_xeon7400subtitle.

[66] Intel (2009). Intel homepage. http://www.intel.com/pressroom/kits/quickreffam.htm.

[67] Jin, H., Khandekar, A., and McEliece, R. (2000). Irregular repeat-accumulate codes. In *Proceedings of the 2nd International Symposium on Turbo Codes & Related Topics*.

[68] Keckler, S. W., Olukotun, K., and Hofstee, H. P. (2009). *Multicore Processors and Systems*. Springer.

[69] Kessenich, J., Baldwin, D., and Rost, R. (2006). The OpenGL Shading Language. *3Dlabs, Inc. Ltd.*

[70] Kienle, F., Brack, T., and Wehn, N. (2005). A Synthesizable IP Core for DVB-S2 LDPC Code Decoding. In *Proceedings of Design, Automation and Test in Europe, 2005 (DATE'05)*, pages 1–6. IEEE.

[71] Kim, H. and Bond, R. (2009). Multicore Software Technologies. *IEEE Signal Processing Magazine*, 26(6):1–10.

[72] Kondratieva, P., Krüger, J., and Westermann, R. (2005). The Application of GPU Particle Tracing to Diffusion Tensor Field Visualization. In *Proceedings of the IEEE Visualization 2005 (VIS'05)*, pages 73–78. IEEE.

[73] Kumar, R. (2008). *Fabless Semiconductor Implementation*. McGraw-Hill.

[74] Kumar, S., Hughes, C. J., and Nguyen, A. (2007). Architectural Support for Fine-Grained Parallelism on Multi-core Architectures. *Intel Technology Journal*, 11(3):217–226.

[75] Kurzak, J., Buttari, A., and Dongarra, J. (2008a). Solving Systems of Linear Equations on the CELL Processor Using Cholesky Factorization. *IEEE Transactions on Parallel and Distributed Systems*, 19(9):1175–1186.

[76] Kurzak, J., Buttari, A., Luszczek, P., and Dongarra, J. (2008b). The PlayStation 3 for High-Performance Scientific Computing. *Computing in Science & Engineering*, 10(3):84–87.

[77] Lemaire, J. B., Schaefer, J. P., Martin, L. A., Faris, P., Ainslie, M. D., and Hull, R. D. (1999). Effectiveness of the Quick Medical Reference as a diagnostic tool. *Canadian Medical Association Journal*, 161(6):725–728.

[78] Lin, C.-Y., Chung, Y.-C., and Liu, J.-S. (2003). Efficient Data Compression Methods for Multidimensional Sparse Array Operations Based on the EKMR Scheme. *IEEE Transactions on Computers*, 52(12):1640–1646.

[79] Lin, S. and Costello, D. J. (2004). *Error Control Coding*. Prentice Hall.

[80] Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J. (2008). NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55.

[81] Liu, C.-H., Yen, S.-W., Chen, C.-L., Chang, H.-C., Lee, C.-Y., Hsu, Y.-S., and Jou, S.-J. (2008a). An LDPC Decoder Chip Based on Self-Routing Network for IEEE 802.16e Applications. *IEEE Journal of Solid-State Circuits*, 43(3):684–694.

[82] Liu, L., Li, Z., and Sameh, A. H. (2008b). Analyzing Memory Access Intensity in Parallel Programs on Multicore. In *Proceedings of the 22nd ACM International Conference on Supercomputing (ICS'08)*, pages 359–367.

[83] Mackay, D. (1999). Good error-correcting codes based on very sparse matrices. *IEEE Trans. Inform. Theory*, 45:399–431.

[84] Mackay, D. and Neal, R. (1996). Near Shannon Limit Performance of Low Density Parity Check Codes. *IEE Electronics Letters*, 32(18):1645–1646.

[85] Mansour, M. (2006). A Turbo-Decoding Message-Passing Algorithm for Sparse Parity-Check Matrix Codes. *IEEE Transactions on Signal Processing*, 54(11):4376–4392.

[86] Mansour, M. and Shanbhag, N. (2002). Memory-efficient turbo decoder architectures for LDPC codes. In *Proceedings of the IEEE Workshop on Signal Processing Systems (SIPS'02)*, pages 159–164. IEEE.

[87] Mansour, M. and Shanbhag, N. (2003). High-Throughput LDPC Decoders. *IEEE Transactions on Very Large Scale Integration Systems*, 11(6):976–996.

[88] Mansour, M. and Shanbhag, N. (2006). A 640-Mb/s 2048-Bit Programmable LDPC Decoder Chip. *IEEE Journal of Solid-State Circuits*, 41(3):684–698.

[89] McCool, M. (2008). Scalable Programming Models for Massively Multicore Processors. *Proceedings of the IEEE*, 96(5):816–831.

[90] Microsoft (2009). DirectX homepage. http://www.microsoft.com/directx.

[91] Momcilovic, S. and Sousa, L. (2008). Parallel Advanced Video Coding: Motion Estimation on Multi-cores. *Journal of Scalable Computing: Practice and Experience*, 9(3):207–218.

[92] Montagne, E. and Ekambaram, A. (2004). An optimal storage format for sparse matrices. *Information Processing Letters*, 90(2):87–92.

[93] Moon, T. K. (2004). On General Linear Block Code Decoding Using the Sum-Product Iterative Decoder. *IEEE Communications Letters*, 8(6):383–385.

[94] Moon, T. K. (2005). *Error Correction Coding – Mathematical Methods and Algorithms*. John Wiley & Sons, Inc.

[95] Müller, S., Schreger, M., Kabutz, M., Alles, M., Kienle, F., and Wehn, N. (2009). A novel LDPC Decoder for DVB-S2 IP. In *Proceedings of Design, Automation and Test in Europe, 2009 (DATE'09)*, pages 1308–1313. IEEE.

[96] NVIDIA (2007). 8800 GTX NVIDIA homepage. http://www.nvidia.com/page/geforce_8800.html.

[97] NVIDIA (2009). CUDA homepage. http://developer.nvidia.com/object/cuda.html.

[98] Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C. (2008). GPU Computing. *Proceedings of the IEEE*, 96(5):879–899.

[99] Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Kruger, J., Lefohn, A. E., and Purcell, T. J. (2007). A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113.

[100] Papaefstathiou, I., Kornaros, G., and Chrysos, N. (2007). Using Buffered Crossbars for Chip Interconnection. In *Proceedings of the 17th ACM Great Lakes symposium on VLSI (GLSVLSI 2007)*, pages 90–95. ACM.

[101] Ping, L. and Leung, W. K. (2000). Decoding low density parity check codes with finite quantization bits. *IEEE Communications Letters*, 4(2):62–64.

[102] Prabhakar, A. and Narayanan, K. (2005). A memory efficient serial LDPC decoder architecture. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2005)*, pages V–41–V–44.

[103] Rabbah, R. (2007). Beyond gaming: programming the PLAYSTATION®3 cell architecture for cost-effective parallel processing. In *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis (CODES+ISSS'07)*. ACM.

[104] Radosavljevic, P., de Baynast, A., and Cavallaro, J. (2005). Optimized Message Passing Schedules for LDPC Decoding. In *Proceedings of the 39th Asilomar Conference on Signals, Systems and Computers*, pages 591–595. IEEE.

[105] Rapidmind (2009). Rapidmind homepage. http://www.rapidmind.com.

[106] Richardson, T. J. and Urbanke, R. L. (2001). The Capacity of Low-Density Parity-Check Codes Under Message-Passing Decoding. *IEEE Transactions on Information Theory*, 47(2):599–618.

[107] Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Dubey, P., Junkins, S., Lake, A., Cavin, R., Espasa, R., Grochowski, E., Juan, T., Abrash, M., Sugerman, J., and Hanrahan, P. (2009). Larrabee: A Many-Core x86 Architecture for Visual Computing. *IEEE Micro*, 29(1):10–21.

[108] Seo, S., Mudge, T., Zhu, Y., and Chakrabarti, C. (2007). Design and Analysis of LDPC Decoders for Software Defined Radio. In *Proceedings of IEEE Workshop on Signal Processing Systems (SiPS'07)*, pages 210–215. IEEE.

[109] Sharon, E., Litsyn, S., and Goldberger, J. (2004). An efficient message-passing schedule for LDPC decoding. In *Proceedings of the 23rd IEEE Conv. of Electrical and Electronics Engineers in Israel*, pages 223–226. IEEE.

[110] Shengfei, L., Yunquan, Z., Xiangzheng, S., and RongRong, Q. (2009). Performance Evaluation of Multithreaded Sparse Matrix-Vector Multiplication Using OpenMP. In *Proceedings of the 2009 11th IEEE International Conference on High Performance Computing and Communications (HPCC'09)*, pages 659–665.

[111] Shih, X.-Y., Zhan, C.-Z., Lin, C.-H., and Wu, A.-Y. (2008). An 8.29 $mm^2$ 52 mW Multi-Mode LDPC Decoder Design for Mobile WiMAX System in 0.13 $\mu$m CMOS Process. *IEEE Journal of Solid-State Circuits*, 43(3):672–683.

[112] Sklar, B. and Harris, F. J. (2004). The ABCs of Linear Block Codes. *IEEE Signal Processing Magazine*, pages 14–35.

[113] Song, H., Todd, R. M., and R., C. J. (2000). Low Density Parity Check Codes for Magnetic Recording Channels. *IEEE Transactions on Magnetics*, 36(5):2183–2186.

[114] Sousa, L., Momcilovic, S., Silva, V., and Falcão, G. (2009). Multi-core Platforms for Signal Processing: Source and Channel Coding. In *Proceedings of the 2009 IEEE International Conference on Multimedia and Expo (ICME'09)*.

[115] Sudderth, E. B. and Freeman, W. T. (2008). Signal and Image Processing with Belief Propagation. *IEEE Signal processing Magazine*, pages 114–141.

[116] Sugano, H. and Miyamoto, R. (2009). Parallel Implementation of Good Feature Extraction for Tracking on the Cell Processor with OpenCV Interface. In *Proceedings of the Fifth International Conference on Intelligent Information Hiding and Multimedia Signal Processing (IIH-MSP'09)*, pages 1326–1329. IEEE.

[117] Sun, J., Zheng, N.-N., and Shum, H.-Y. (2003). Stereo Matching Using Belief Propagation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(7):787–800.

[118] Tanner, R. (1981). A Recursive Approach to Low Complexity Codes. *IEEE Transactions on Information Theory*, 27(5):533–547.

[119] Tilera (2008). Tilera TILE64™ Processor Family homepage. http://www.tilera.com/products/TILE64.php.

[120] Vasic, B., Djordjevic, I. B., and Kostuk, R. K. (2003). Low-Density Parity Check Codes and Iterative Decoding for Long-Haul Optical Communication Systems. *Journal of Lightwave Technology*, 21(2):438–446.

[121] Verdier, F. and Declercq, D. (2006). A low-cost parallel scalable FPGA architecture for regular and irregular LDPC decoding. *IEEE Transactions on Communications*, 54(7):1215–1223.

[122] Wicker, S. B. and Kim, S. (2003). *Fundamentals of Codes, Graphs, and Iterative Decoding*. Kluwer Academic Publishers.

[123] Wiechman, G. and Sason, I. (2007). Parity-Check Density Versus Performance of Binary Linear Block Codes: New Bounds and Applications. *IEEE Transactions on Information Theory*, 53(2):550–579.

[124] Yadav, M. K. and Parhi, K. K. (2005). Design and Implementation of LDPC Codes for DVB-S2. In *Proceedings of the 39th Asilomar Conference on Signals, Systems and Computers*, pages 723–728. IEEE.

[125] Yamagiwa (2007). Caravela homepage. http://www.caravela-gpu.org.

[126] Yamagiwa, S. and Sousa, L. (2007). Caravela: A Novel Stream-based Distributed Computing Environment. *IEEE Computer*, 40(5):70–77.

[127] Yamagiwa, S. and Sousa, L. (2009a). CaravelaMPI: Message Passing Interface for Parallel GPU-Based Applications. In *Proceedings of the 2009 Eighth International Symposium on Parallel and Distributed Computing (ISPDC'09)*, pages 161–168.

[128] Yamagiwa, S. and Sousa, L. (2009b). Modeling and programming stream-based distributed computing based on the meta-pipeline approach. *International Journal of Parallel, Emergent and Distributed Systems*, 24(4):311–330.

[129] Yamagiwa, S., Sousa, L., and Antão, D. (2007). Data buffering optimization methods toward a uniformed programming interface for GPU-based applications. In *Proceedings of the Computing Frontiers 2007 ACM (CF'07)*. ACM.

[130] Yang, Z., Zhu, Y., and Pu, Y. (2008). Parallel Image Processing Based on CUDA. In *Proceedings of the 2008 International Conference on Computer Science and Software Engineering (CSSE'08)*, pages 198–201. IEEE.

[131] Yeo, E., Pakzad, P., Nikolić, B., and Anantharam, V. (2001a). High Throughput Low-Density Parity-Check Decoder Architectures. In *Proceedings of the IEEE Global Telecommunications Conf. (GLOBECOM'01)*, pages 3019–3024. IEEE.

[132] Yeo, E., Pakzad, P., Nikolić, B., and Anantharam, V. (2001b). VLSI Architectures for Iterative Decoders in Magnetic Recording Channels. *IEEE Transactions on Magnetics*, 37(2):748–755.

[133] Zhang, B., Liu, H., Chen, X., Liu, D., and Yi, X. (2009). Low Complexity DVB-S2 LDPC Decoder. In *Proceedings of the IEEE 69th Vehicular Technology Conference, 2009 (VTC Spring 2009)*, pages 1–5. IEEE.

[134] Zhang, J. and Fossorier, M. (2002). Shuffled belief propagation decoding. In *Proceedings of the 36th Asilomar Conference on Signals, Systems and Computers*, pages 8–15. IEEE.

[135] Zhang, J. and Fossorier, M. (2005a). Corrections to "Shuffled Iterative Decoding". *IEEE Transactions on Communications*, 53(7):1231–1231.

[136] Zhang, J. and Fossorier, M. (2005b). Shuffled Iterative Decoding. *IEEE Transactions on Communications*, 53(2):209–213.

[137] Zhang, T. and Parhi, K. (2004). Joint (3,k)-regular LDPC code and decoder/encoder design. *IEEE Transactions on Signal Processing*, 52(4):1065–1079.

[138] Zhong, H., Zhong, T., and Haratsch, E. F. (2007). Quasi-Cyclic LDPC Codes for the Magnetic Recording Channel: Code Design and VLSI Implementation. *IEEE Transactions on Magnetics*, 43(3):1118–1123.