Nuno Ricardo Regalo Vicente

RaptorQ decoder using heterogeneous computing on a mobile multi-core platform

Dissertação de Mestrado em
Engenharia Electrotécnica e de Computadores

Setembro de 2015

Universidade de Coimbra

RaptorQ decoder using heterogeneous computing on a mobile multi-core platform

Nuno Ricardo Regalo Vicente

Dissertação para obtenção do Grau de Mestre em

Engenharia Electrotécnica e de Computadores

Orientador: Doutor Gabriel Falcão Paiva Fernandes
Co-Orientador: Doutor Vítor Manuel Mendes da Silva

Júri

Presidente: Doutor Nuno Miguel Mendonça da Silva Gonçalves
Orientador: Doutor Gabriel Falcão Paiva Fernandes
Vogal: Doutor Marco Alexandre Cravo Gomes

Setembro de 2015

Agradecimentos

Aos meus orientadores, Doutor Gabriel Falcão e Doutor Vítor Silva, pela inesgotável compreensão e ajuda ao longo de todas as etapas.

Ao João Andrade, pela paciência e gosto em ajudar e ensinar.

Ao Manuel Rodrigues, pelas horas despendidas nas dificuldades do projecto.

Aos colegas do laboratório, pelo bom ambiente e a entreajuda.

Aos meus pais, pelas temporadas ausentes e por todo o esforço diário para me dar o melhor.

À Cláudia, porque seria necessária outra tese para descrever todo o seu apoio e carinho.

Aos meus colegas e amigos, por todos os bons e maus momentos que passamos juntos.

A todos eles, um Muito Obrigado.

Abstract

The Raptor codes, since its invention by 2000/2001, are the most powerful known fountain codes. They have fast encoding and decoding algorithms, offering high reliability at low overheads (i.e. for a small amount of repair information). With the growth of the network and the mobile platforms, these codes attain special interest, both in low processing time and low power consumption. This work presents a study that shows how parallel computing can improve the decoding time of the most advanced Raptor code, the RaptorQ, and how power consumption can be lowered due to the joint CPU/GPU processing time. Even for large block sizes, it is possible to obtain reduction time decoding, with speedups higher than 10x in a mobile device.

Keywords

Raptor codes, mobile platforms, parallel processing, data-parallelism, heterogeneous computing, multi-threading, OpenCL, OpenMP, low power.

Resumo

Os Raptor codes, desde a sua criação em 2000/2001, são os fountain codes conhecidos mais poderosos. Apresentam algoritmos de codificação e descodificação rápidos, oferecendo elevada confiança a um custo baixo (i.e. para um número baixo de símbolos de recuperação). Com o crescimento das redes e dos dispositivos móveis, estes códigos ganham especial interesse, tanto em tempo poupado pelos processadores, como em consumos energéticos. Este trabalho apresenta um estudo que mostra de que forma a computação paralela permite melhorar os tempos de descodificação do mais recente Raptor code, o RaptorQ, bem como a poupança energética face ao tempo gasto pela combinação CPU/GPU durante o processo. Mesmo para grandes blocos, consegue-se obter uma redução do tempo de descodificação superior a 10x num dispositivo móvel.

Palavras-Chave

Raptor codes, plataformas móveis, processamento paralelo, paralelismo de dados, computação heterogénea, multi-threading, OpenCL, OpenMP, baixa potência.

Contents

1 Introduction **1**

1.1 Motivation 2

1.2 Goals 2

1.3 Contributions 2

1.4 Outline 3

2 Principles of non-binary codes and Rapid Tornado codes **5**

2.1 Finite fields 6

2.2 Primitive polynomials 7

2.3 Fountain codes 8

2.4 Rapid Tornado codes 9

2.4.1 Raptor10 encoder 11

2.4.2 RaptorQ encoder 12

2.5 RaptorQ decoder 14

2.6 RaptorQ decoder using matrix dimensionality reduction 15

2.7 Summary 16

3 Parallel programming on mobile devices **19**

3.1 The Snapdragon Architecture 20

3.1.1 Central Processing Unit 21

3.1.2 Graphics Processing Unit 22

3.2 Parallel programming 23

3.2.1 OpenMP API 23

3.2.2 OpenCL framework 24

3.3 Summary 27

4 RaptorQ decoder on Snapdragon CPU/GPU **29**

4.1 Parallelization of matrix multiplication 30

4.2 Parallelization of matrix inversion 31

4.2.1 OpenCL approach for matrix inversion 33

4.2.2 OpenMP approach for matrix inversion 35
4.3 Summary 37

5 Experimental results 39
5.1 Decoding time comparison of the two decoding schemes 40
5.2 Matrix reduction decoder 43
5.3 Matrix inversion 45
5.4 Precode matrix 46
5.5 Summary 47

6 Conclusions 49
6.1 Future work 50

List of Figures

2.1 Fountain representing the receivers asking for symbols from the encoder. 8
2.2 Bipartite graph that relates N_i encoded symbols to $K_{j0}..K_{jl}$ source symbols. 9
2.3 Raptor code scheme. 9
2.4 Block diagram of Raptor10 encoder. 11
2.5 Precode matrix of Raptor10 code. 12
2.6 Block diagram of RaptorQ encoder. 13
2.7 Precode matrix of RaptorQ code. 13
2.8 RaptorQ CODEC. 14
2.9 Precode matrix A' scheme, when one source symbol is missing and one repair symbol is appended. 15
2.10 Block diagram of matrix reduction decoder [11]. 16

3.1 Snapdragon 800 scheme. 20
3.2 Snapdragon SoC scheme. 21
3.3 GPU thread scheme. 22
3.4 OpenMP memory scheme. 23
3.5 The fork-join programming model. 24
3.6 OpenCL platform model representing N devices orchestrated by a host. . 25
3.7 NDRange index space with work-groups and work-items scheme. 26
3.8 OpenCL memory model. 27

4.1 Matrix multiplication scheme. 30
4.2 Scheme of a 4×4 matrix inversion. 32

5.1 Symbol size $T = 4bytes$. 40
5.2 Symbol size $T = 8bytes$. 41
5.3 Symbol size $T = 16bytes$. 41
5.4 Symbol size $T = 32bytes$. 41
5.5 Symbol size $T = 64bytes$. 42
5.6 Symbol size $T = 128bytes$. 42

5.7 Symbol size $T = 256 bytes$. 42
5.8 Symbol size $T = 512 bytes$. 43
5.9 Symbol size $T = 1024 bytes$. 43
5.10 Symbol size $T = 512 bytes$. 44
5.11 Symbol size $T = 1024 bytes$. 44
5.12 Matrix inversion time on Snapdragon. 46
5.13 Pre-code matrix contruction time. 47

List of Tables

2.1 Multiplication table of nonzero elements in $GF(5)$. 7

2.2 Construction of $GF(2^3)$. 7

5.1 Test environment specs. 40

5.2 Snapdragon speedups for a block size $T = 512 bytes$ 44

5.3 Snapdragon speedups for a block size $T = 1024 bytes$ 44

5.4 Snapdragon maximum instant power consumptions for a sequential C approach of the RaptorQ decoder. 45

5.5 Snapdragon maximum instant power consumptions for an OpenCL approach of the RaptorQ decoder. 45

List of Algorithms

1 Matrix multiplication in sequential C . . . 31
2 Parallel matrix multiplication on GPU . . . 31
3 Matrix multiplication kernel in OpenCL . . . 31
4 Row swap in sequential C . . . 33
5 Row swap kernel in OpenCL . . . 33
6 Row division in sequential C . . . 33
7 Row division kernel in OpenCL . . . 34
8 Row sum in sequential C . . . 34
9 Row sum kernel in OpenCL . . . 34
10 Parallel Gauss-Jordan on a GPU . . . 35
11 Row swap with OpenMP . . . 36
12 Row division with OpenMP . . . 36
13 Row sum with OpenMP . . . 36

List of Acronyms

ALU Arithmetic Logic Unit

API Application Programming Interface

ARM Acorn Risc Machine

CPU Central Processing Unit

CU Control Unit

CU Computer Unit

FEC Forward Error Correction

GPGPU General-Purpose Computing on Graphics Processing Units

GPU Graphics Processing Unit

HDPC High Density Parity Check

LDPC Low Density Parity Check

LT Luby Transform

NDRange N Dimensional Range

OpenCL Open Computing Language

OpenMP Open Multi-Processing

PE Processing Element

RAM Random Access Memory

Raptor Rapid Tornado

RISC Reduced Instruction Set Computer

SoC System on Chip

1

Introduction

1.1 Motivation

Over the last years, the number of mobile users surpassed half of the world's population. Nowadays it is rare to see someone who is not surfing on the web or checking the e-mail on a smartphone, or even working on his laptop. This changed the concept of how software applications are developed because of the concern with power consumptions. A common smartphone with an average daily use of 8 hours may run out of charge quickly and in some heavy tasks, the CPU/GPU may overheat, which results in faster hardware wear.

The Raptor codes are a recent set of codes that work on the network application layer. They can be used in digital media broadcast, cellular networks or in satellite communications. They are the most powerful fountain codes available with fast encoding and decoding algorithms and, for the crowded network, this may reduce substantial the bottleneck improving the quality of service. There is a lot of work done to enhance Raptor codes in traditional platforms, however, a few work was presented in the field of mobile devices. The motivation of this work is to show how Raptor codes can be well suited for mobile platforms, using parallel computing.

1.2 Goals

If we analyze the decoding process of a Raptor code, we can identify branches of the algorithm that are optimal for being parallelized. Hereupon, it is inviting to exploit the potential of heterogeneous computing over this codes. The goals of this thesis are:

- To improve RaptorQ decoding time, exploiting matrix multiplication and matrix inversion parallelism.
- To show how a parallel version of RaptorQ may save energy consumption.

1.3 Contributions

Web applications involve encoding and decoding processes for reliable data transmission. Since mobile devices are battery dependent and have low processing power comparing with traditional computing devices, this work presents an optimized solution for the RaptorQ decoder, with decoding speedups varying between 5.3x and 51.8x using OpenCL, comparing with a sequential C implementation. Furthermore, it is proposed an hybrid decoding scheme using OpenMP, when loss rates are high and the recover matrix size is large.

Considering the results of this work and its improvements, an article is being prepared for submission.

1.4 Outline

This thesis is organized in 6 chapters.

The 2nd chapter gives a brief explanation of what's behind a Raptor code, introducing basic concepts of linear algebra and finite fields.

The 3rd chapter introduces the hardware concepts of the heterogeneous computing paradigm and presents the OpenMP API and the OpenCL framework, that take advantage of hardware capabilities to exploit thread-level parallelism.

The 4th chapter explains the techniques developed to parallelize the RaptorQ decoder.

The 5th chapter presents the simulation results of the techniques explained in chapter 4.

Finally, the 6th chapter makes a conclusion of the thesis and presents the proposed future work, based on the results.

2

Principles of non-binary codes and Rapid Tornado codes

Algebra has a wide range of applications, especially in coding theory. Modern codes use complex schemes to ensure data compression and reliable error correction using low computational time.

This chapter provides the basics of what is behind non-binary codes in Forward Error Correction (FEC) schemes starting with a brief introduction to the theory of finite fields and the principles of fountain codes. Thereafter, it addresses the core of this work, the powerful Raptor codes.

2.1 Finite fields

A finite field (also called Galois Field) is a field that contains a finite number of elements [1–3]. Similarly to the set of infinite numbers, a finite field has also well defined mathematical operations and properties:

- Addition
- Subtraction
- Multiplication
- Division (except by 0)
- Commutativity under addition
- Commutativity under multiplication when the additive identity element 0 is removed
- Distributivity
- Identity
- Inverse

Finite fields are usually written as $GF(q)$, where q is the order or cardinality. To satisfy all properties, q must be a prime number or a power of a prime number greater than 1. That is

$$q = p^k, \tag{2.1}$$

where p is a prime number and k is a positive integer. The field $GF(q)$ can be represented as $\{0, 1, ..., q-1\}$ and is constructed under multiplication $modulo - q$.

The elements of $GF(q)$ can be represented by polynomials and their operations are performed over modulo R where R is an irreducible polynomial of degree k.

For example, $GF(5) = \{0,1,2,3,4\}$ is a finite field because the elements $\{1,2,3,4\}$ are a group under $modulo-q$ multiplication.

Table 2.1: Multiplication table of nonzero elements in $GF(5)$.

	1	**2**	**3**	**4**
1	1	2	3	4
2	2	4	1	3
3	3	1	4	2
4	4	3	2	1

2.2 Primitive polynomials

A primitive polynomial is the minimal degree polynomial that defines a field given by $GF(q)$, and generates all its elements. This polynomial is irreducible which clarifies the explanation of section 2.1.

Let $f(x)$ be a polynomial of degree k. Let $GF(q)$ be a finite field where $q = p^k$. The polynomial $f(x)$ is primitive if

$$(x^q + 1) mod f(x) = 0. \quad (2.2)$$

Let's examine one example (which is the field used in subsection 2.4.2). Given a finite field $GF(256)$ where $256 = 2^8$, the polynomial $x^8 + x^4 + x^3 + x^2 + 1$ is a primitive polynomial because

$$(x^{256} + 1) mod (x^8 + x^4 + x^3 + x^2 + 1) = 0$$

.

The following table shows the construction of an extension field in $GF(8)$ with $p = 2$, $k = 3$ and using the primitive polynomial $f(x) = x^3 + x + 1$, [3].

Table 2.2: Construction of $GF(2^3)$.

Elements in $GF(2^3)$	Element expressed as the sum of lower powers of α	Element expressed as 3-tuple vector over $GF(2)$
$\mathbf{0}$	$\mathbf{0}$	000
$\mathbf{1}$	$\mathbf{1}$	001
α	α	010
α^2	α^2	100
α^3	$\alpha + \mathbf{1}$	011
α^4	$\alpha^2 + \alpha$	110
α^5	$\alpha^3 + \alpha^2 = \alpha + \mathbf{1} + \boldsymbol{\alpha^2}$	111
α^6	$\alpha^4 + \alpha^3 = \alpha^2 + \alpha + \alpha + \mathbf{1} = \boldsymbol{\alpha^2} + \mathbf{1}$	101

Representing a field this way is very useful from a computational point of view. When $p = 2$, addition is accomplished by adding the vector representation of the elements $modulo - 2$. This means that sum equals subtraction and can performed by XOR operations. For example, $\alpha^5 + \alpha^6 = (111) + (101) = (010) = \alpha$.

In the case of multiplication and division the operation is more complex and there are several way to perform these. However, a solution that produces a feasible and simple approach consists of creating the exponential and logarithmic tables of the corresponding finite field and then perform sums and subtractions using those values. One practical example is given in [4].

2.3 Fountain codes

The name fountain code [5–7] (also known as rateless erasure code) arises from its nature.

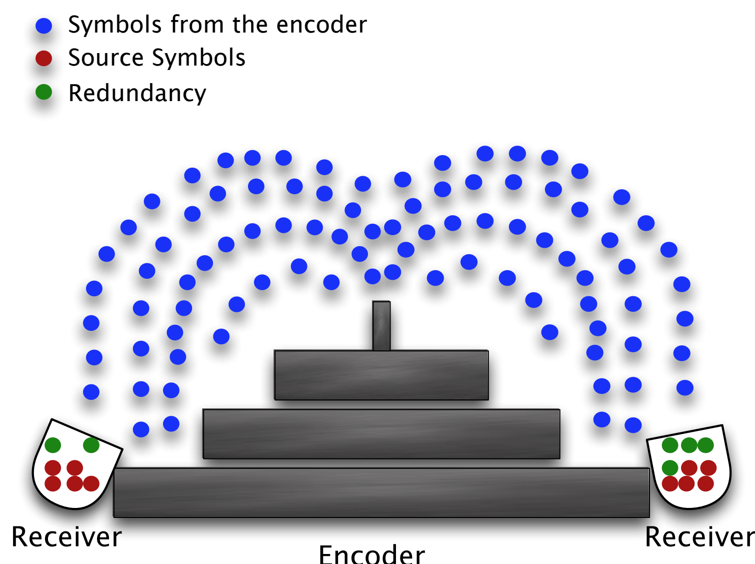

Figure 2.1: Fountain representing the receivers asking for symbols from the encoder.

The encoder can be constantly sending new symbols, which are pseudo-random linear combinations of the source symbols. If the decoding process fails (e.g. because of corrupted or lost packets) the receiver asks for more symbols until the decoding process can be successfully completed, as shown in figure 2.1. Therefore, they don't have a fixed code rate.

Luby Transform (LT) codes were the first applying this concept in practice [8]. The main idea is based on bipartite graphs where K source symbols and N encoded symbols are related to each other [6].

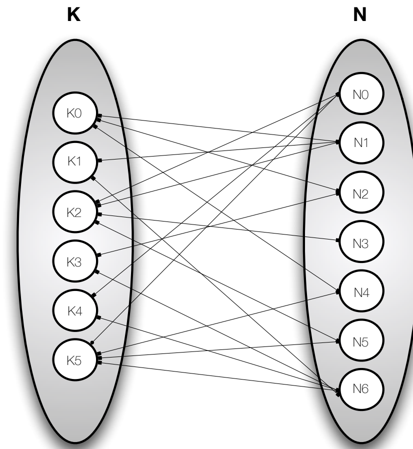

Figure 2.2: Bipartite graph that relates N_i encoded symbols to $K_{j0}..K_{jl}$ source symbols.

2.4 Rapid Tornado codes

Rapid Tornado codes (also known as Raptor codes) are an extension of LT codes explained in section 2.3 with linear time encoding and decoding [6, 9]. Raptor codes are systematic codes, that is, all source symbols are included as part of the encoding symbols of a source block. The main idea of Raptor codes is to precode the source symbols, generating thus a set of intermediate symbols that will be encoded with an LT code.

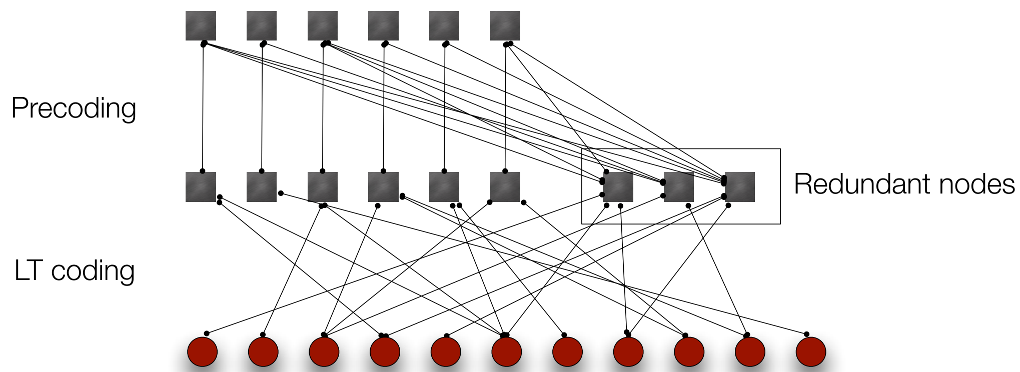

Figure 2.3: Raptor code scheme [9].

The advantage of precoding resides on intermediate symbols, whose redundancy allows the decoder to recover the intermediate symbols if most of them are known. Comparing with LT, this is crucial because if we relate source symbols directly and set the output symbols with a constant average degree, some of them will not contribute to the encoded

symbols and lead us to a situation wherein these source symbols cannot be recovered.

Before moving into the encoders description, some notation must be introduced [4, 10].

- **Symbol:** A unit of data whose size is measured in T bytes.
- **Source symbol:** The smallest unit of data used during the encoding process.
- **Source block:** A block of K source symbols that are considered together for encoding and decoding purposes. All source symbols within a source block have the same size.
- **Padding symbol:** A symbol with all zero bits.
- **Extended source block:** A block of of K' source symbols constructed from a source block K plus padding symbols.
- **Repair Symbols:** The encoding symbols of a source block that are not source symbols. They are generated based on the source symbols of a source block.
- **Encoding symbol:** A symbol that can be sent as part of the encoding of a source block. They consist of the source symbols plus the repair symbols.
- **Intermediate Symbols:** Symbols generated from the source symbols using an inverse encoding process based on precoding relationships. The repair symbols are then generated directly from the intermediate symbols.
- **LT Symbols:** Symbols used to generate part of the contribution to each generated encoding symbol, from the portion of the intermediate symbols.
- **HDPC and LDPC symbols:** A set of intermediate symbols which have a precoding relationship with a small fraction of the other intermediate symbols.
- **Precode matrix:** A matrix that is constituted by sub-matrices, used to generate intermediate symbols.
- **$\mathbf{K}^{+}$:** Number of encoding symbols, that is, source symbols and repair symbols.
- **$\mathbf{K}^{(R)}$:** Number of repair symbols.
- **X:** Value of the Encoding Symbol ID, which uniquely identifies the each encoding symbol associated with a source block.

- **L:** Number of intermediate symbols.
- **S:** Number of LDPC symbols.
- **H:** Number of HDPC symbols.
- **B:** Number of intermediate symbols that are LT symbols, excluding LDPC symbols.

2.4.1 Raptor10 encoder

This encoder is built over $GF(2)$, so its precode matrix is binary as well as the encoded data, and allows to encode between 4 and 8192 source symbols, with symbol size T between 1 and $2^{16}-1$ bytes.

The encoding process consists in generating repair symbols from a source block with K source symbols. First of all, the intermediate symbols are generated. Secondly, the repair symbols are generated from the intermediate symbols. Finally, the encoding symbols are constructed, appending the repair symbols to the source symbols (systematic code).

A block diagram to illustrate the encoder may be as follows:

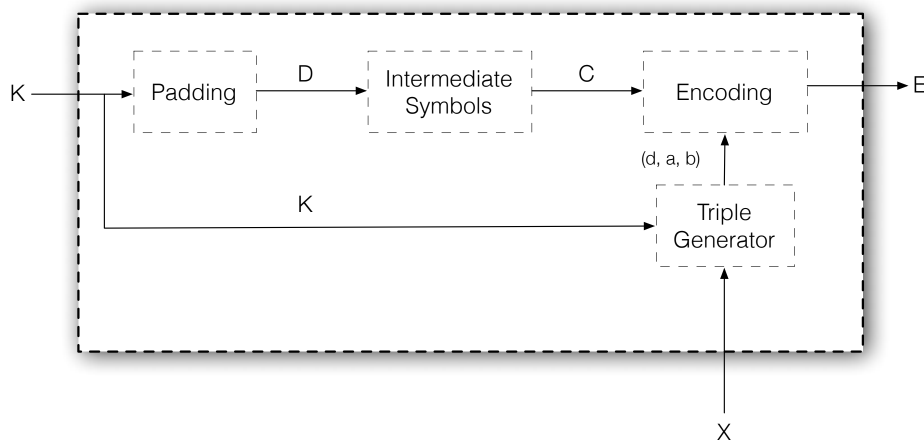

Figure 2.4: Block diagram of Raptor10 encoder [4].

Assuming we want to encode a source block Q with K source symbols of T bytes, the precode matrix $A_{L\times L}$ is generated and is as follows:

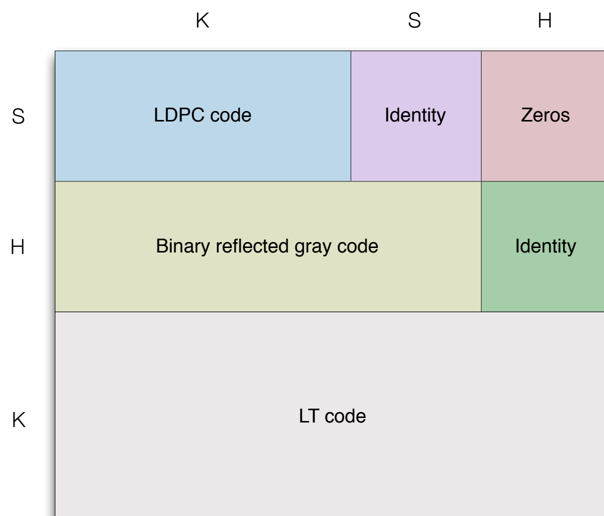

Figure 2.5: Precode matrix of Raptor10 code.

The intermediate symbols C are obtained multiplying the inverse of the precode matrix, A^{-1}, by a matrix D. Matrix D is obtained from the source block Q by adding zeros:

$$D_{(L\times T)} = \begin{bmatrix} 0_{((S+H)\times T)} \\ Q_{(K\times T)} \end{bmatrix}, \tag{2.3}$$

and then

$$C_{(L\times T)} = A^{-1}_{(L\times L)} \times D_{(L\times T)}. \tag{2.4}$$

The encoding symbols E are finally obtained by multiplying an LT code matrix by the intermediate symbols C:

$$E_{(K^{+}\times T)} = (G_{LT})_{(K^{+}\times L)} \times C_{(L\times T)}. \tag{2.5}$$

The encoding symbols E are constituted by the source symbols plus the repair symbols. Since the top of G_{LT} is the same as the bottom of the precode matrix A, the repair symbols are generated as much as more rows are added to G_{LT}, in a pseudo-random manner, using the Triple Generator. The Triple generator uses the encoding symbol ID X to generate the corresponding LT row.

Raptor10 is fully described in [10].

2.4.2 RaptorQ encoder

This encoder is built over $GF(256)$, so the values of its precode matrix are between 0 and 255 as well as the encoded data, and allows to encode between 10 and 56403 source symbols with symbol size T between 1 and $2^{16}-1$. This is an huge upgrade concerning

Raptor10 encoder because it allows a bigger source block. It also supports 16777216 encoded symbols, which admits more repair symbols. In fact, these limitations were imposed due to practical considerations and not due to limitations of the code design [6].

Despite the differences, the encoding process is similar to the Raptor10 encoder. Instead of using a Triple Generator, it uses a Tuple Generator with a different pseudo-random process [4].

A block diagram to illustrate the encoder may be as follows:

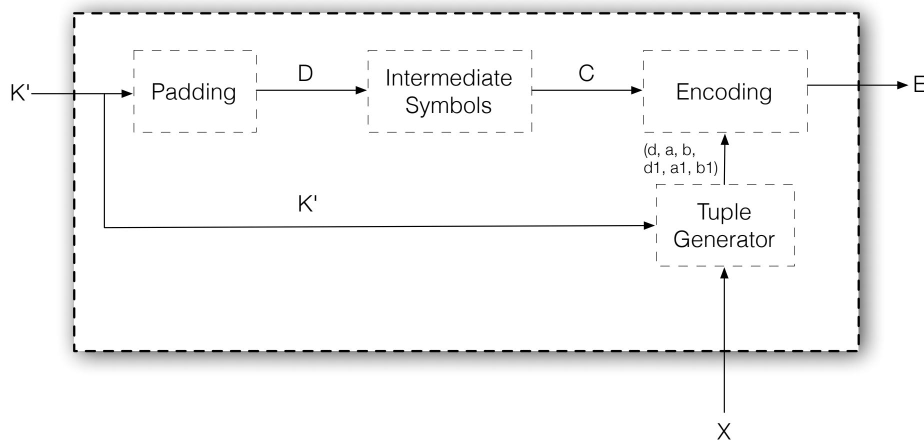

Figure 2.6: Block diagram of RaptorQ encoder [4].

The precode matrix A shows some differences on the sub-matrices. Assuming we want to encode a source block Q with K source symbols of T bytes, the precode matrix A is generated as follows:

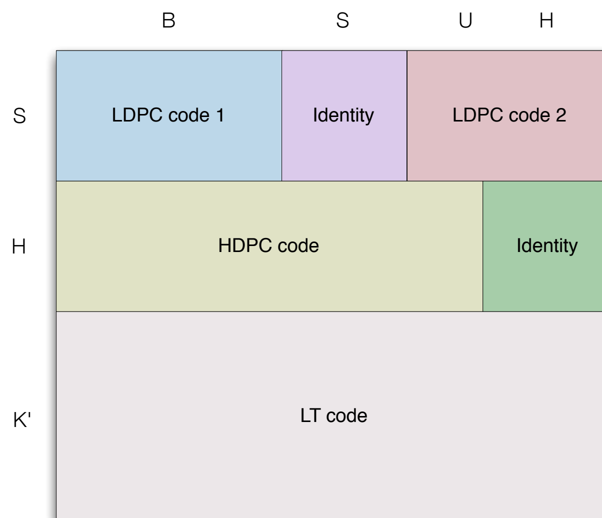

Figure 2.7: Precode matrix of RaptorQ code.

Unlike Raptor10, RaptorQ encoder has fixed values for each source block size. Here K' is the extended source block for which precode A is ready to perform, that is, if $K < K'$, K is padded with padding symbols until it equals K'. RaptorQ is fully described in [4] and these values are defined in table 2 of the standard.

For all these reasons RaptorQ decoder is the case study of this work and two decoding schemes are presented in the following sections.

2.5 RaptorQ decoder

A trivial scheme of the RaptorQ decoder is presented in figure 2.8. It is the inverse operation of the encoder presented in subsection 2.4.2:

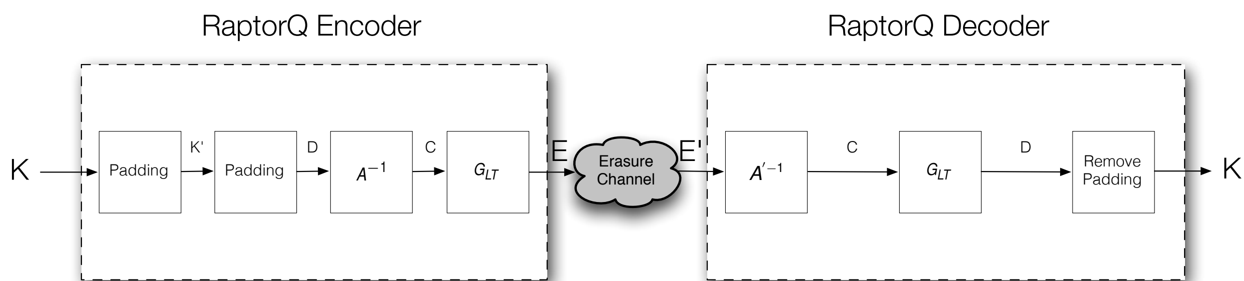

Figure 2.8: RaptorQ CODEC.

The idea of the decoder is to calculate the intermediate symbols C from the received symbols E'. The decoder is able to reconstruct the intermediate symbols when $E' \geq K'$, which means that the decoder must receive enough symbols (source symbols plus repair symbols). If not so, the decoder must ask for more repair symbols to the encoder.

The ideal case occurs when all source symbols are received. Since the code is systematic, the repair symbols are discarded from E and the K original source symbols are correctly delivered. However, if one or more source symbols are lost, some changes in the precode matrix A must be performed. Each row of the sub-matrix LT of A is associated with one encoding symbol. This means that for each missing encoding symbol, the corresponding row of the LT sub-matrix must be discarded. This may lead to a situation where the new precode matrix A' (used for decoding purposes) is rank deficient, that is, it is not invertible. If so, the decoder must ask for more repair symbols until the new precode matrix reaches full rank.

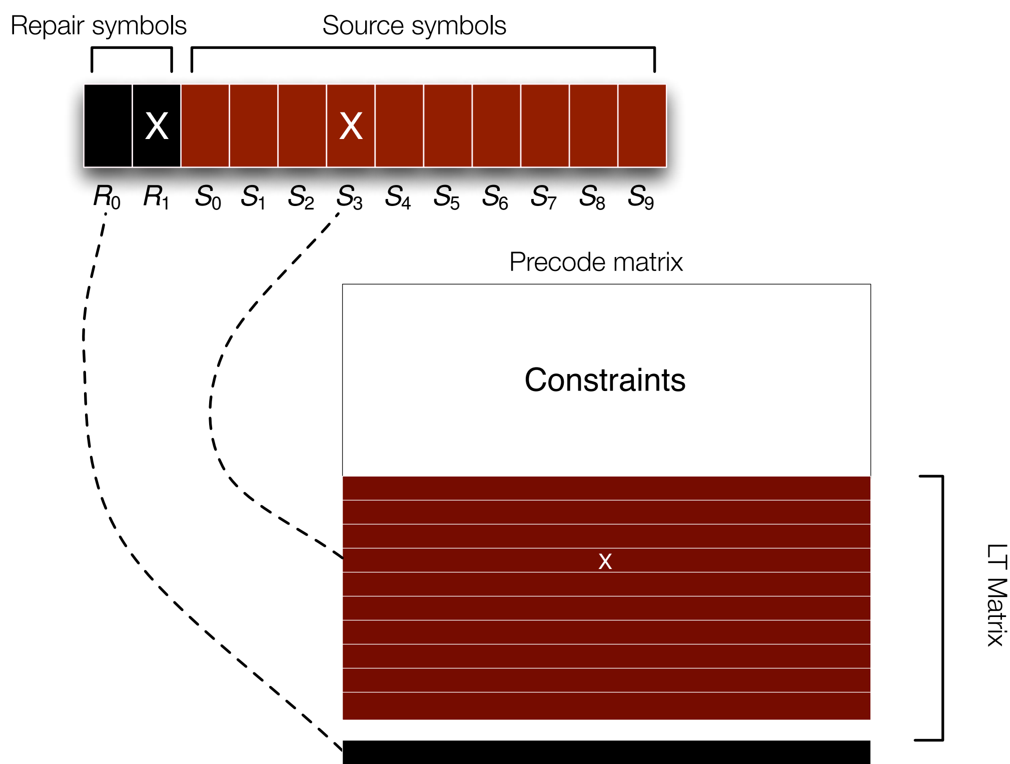

Figure 2.9: Precode matrix A' scheme, when one source symbol is missing and one repair symbol is appended.

This process involves a matrix inversion in $GF(256)$ for each decoding attempt. From a computational point of view, this is not efficient since even when only one source symbol is missing, the entire matrix must be inverted. An alternative decoding scheme is presented in the next section.

2.6 RaptorQ decoder using matrix dimensionality reduction

An alternative decoding scheme is proposed in [11] and reduces substantially the dimensions of the recover matrix. The inverse of the precode A^{-1} is stored once and remains unchanged during the process. Only the repair symbols and the corresponding LT rows are used, which reduces drastically the size of the recover matrix to calculate. In practice, the size of the matrix to invert equals the number of the repair symbols used to recover the source symbols.

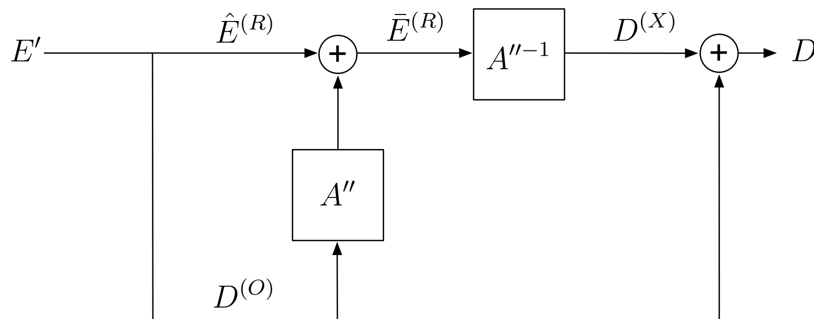

Figure 2.10: Block diagram of matrix reduction decoder [11].

Let D be the matrix obtained from (2.3). If we split D in two matrices,

$$D_{(L\times T)} = D^{(O)}_{(L\times T)} + D^{(X)}_{(L\times T)}, \tag{2.6}$$

where $D^{(O)}$ is the matrix of known source symbols and $D^{(X)}$ the matrix of desired source symbols. Recalling subsections 2.4.1 and 2.4.2 we have:

$$E_{(K^{+}\times T)} = (G_{LT})_{(K^{+}\times L)} \times C_{(L\times T)} = (G_{LT})_{(K^{+}\times L)} \times A^{-1}_{(L\times L)} \times D_{(L\times T)}. \tag{2.7}$$

Replacing D by (2.6) we have:

$$E_{(K^{+}\times T)} = A''_{(K^{+}\times L)} \times D^{(O)}_{(L\times T)} + A''_{(K^{+}\times L)} \times D^{(X)}_{(L\times T)}, \tag{2.8}$$

where $A''_{(K^{+}\times L)} = (G_{LT})_{(K^{+}\times L)} \times A^{-1}_{(L\times L)}$. As we are only interested in missing source symbols, we will only choose the rows of G_{LT} corresponding to the repair symbols resulting in the following equation (recalling the arithmetic of section 2.2):

$$\bar{E}^{(R)}_{(K^{(R)}\times T)} = \hat{E}^{(R)}_{(K^{(R)}\times T)} + A''_{(K^{(R)}\times L)} \times D^{(O)}_{(L\times T)}, \tag{2.9}$$

where $\hat{E}^{(R)}_{(K^{(R)}\times T)}$ is the matrix with the repair symbols and $\bar{E}^{(R)}_{(K^{(R)}\times T)}$ are the desired source symbols.

For instance, if we want to send 60 source symbols and receiver only receives 80% of them, we just need to invert a 12×12 matrix and perform the corresponding multiplications instead of invert a 73×73 precode matrix (see table 2 from [4]).

2.7 Summary

This chapter presented the differences between the two Raptor codes available and two decoding schemes, supported by their mathematical concepts. Raptor10 was the first Raptor code available and is constructed over $GF(2)$. RaptorQ was later developed and

is constructed over $GF(256)$. It allows to encode bigger source blocks with more repair symbols. Regarding the decoding schemes, two methodologies were presented. The first one was the inverse operation of the encoder. Given the received symbols and performing the corresponding substitutions in the precode matrix, the intermediate symbols are calculated inverting the precode and multiplying by the received symbols. The second scheme proposes a matrix dimensionality reduction, which not uses the precode matrix directly. Instead, it uses only the repair symbols and the corresponding LT rows, resulting in a smaller matrix to invert.

The next chapter presents the hardware and software concepts on which RaptorQ decoders will run and introduces the parallel computing paradigm that takes advantage of the threads of CPU and GPU to exploit data parallelism.

3

Parallel programming on mobile devices

3.1 The Snapdragon Architecture

Over the past years, mobile devices have become alternative computing devices. By 2015, the number of mobile devices surpasses the 3.6 billion users, which corresponds to about half of the world's population [12].

Hence, while traditional processing systems are reaching a plateau, not being able to respond to the ever increasing demands of today's developments, engineers tend to search new ways of data processing. Power and energy consumptions have become a significant problem since mobile devices are limited by battery supplies when compared against desktops. Hereupon, a family of instruction set architectures has been developed, called Acorn Risc Machine (ARM) [13, 14]. It uses a Reduced Instruction Set Computer (RISC) architecture whose goal is to keep the design simple to reduce costs, power consumption and heat levels, while keeping programmability.

A powerful processor that meets the requirements is available on market, called Snapdragon [15, 16]. This device is a System on Chip (SoC) product that contains, among other chips, an ARM based processor and a graphics processing unit (GPU), that supports high-levels of data parallelism.

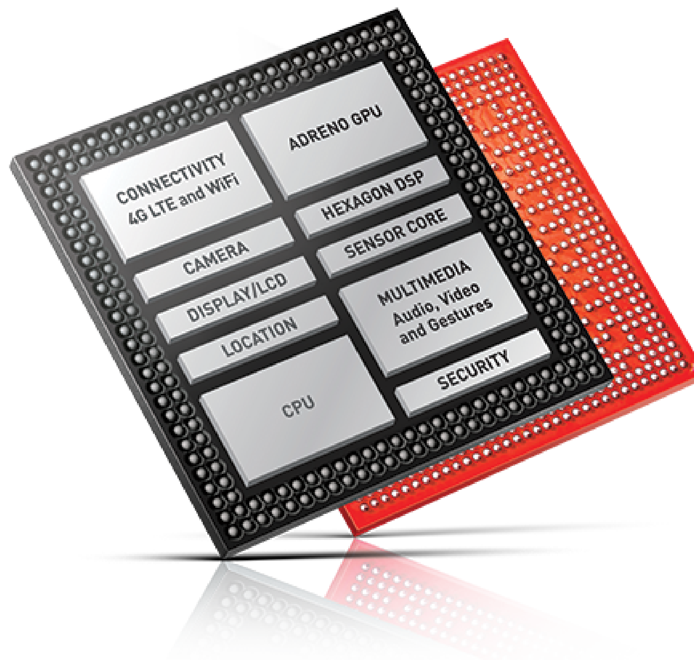

Figure 3.1: Snapdragon 800 scheme.

This chapter shows how parallel programming can be used to take advantage of a Snapdragon processor to deal with the processing of complex algorithms.

3.1.1 Central Processing Unit

Central Processing Unit (CPU) is where computers perform arithmetic and logical operations. This set of operations such as adding, subtracting or comparing numbers is called an instruction set. The instruction set is then processed by Arithmetic Logic Unit (ALU) which is directed by Control Unit (CU) that makes the bridge between all processor's sub-units.

CPUs manufacturers are constantly being challenged as software requires more processing power. As the CPU processing speed is reaching a plateau, mostly by heating, manufacturer's solution was to add more cores to the chip, creating thus a cluster. Current CPUs have more than one core but they are optimized for single thread execution, emphasizing instruction-level parallelism. They are boosted with cache levels to reduce the memory access time and share data among cores. Cache is an hierarchical memory that lies between CPU and Random Access Memory (RAM), with the top level cache being faster (closer to the CPU) and the bottom level slower (closer to the RAM). Figure 3.2 shows a scheme with two cores and two cache levels. L1 cache is private for each core and L2 cache is shared between cores.

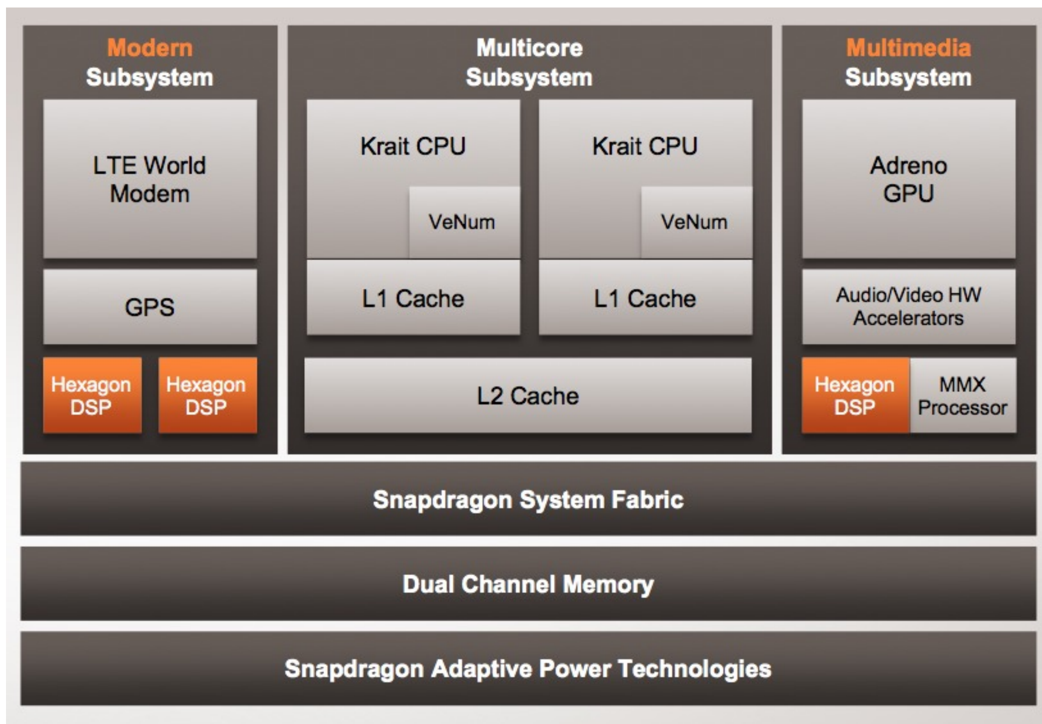

Figure 3.2: Snapdragon SoC scheme showing different cores and cache levels of CPU [17].

Due to these properties, CPU is better suited for serial tasks. In heterogeneous computing [18] CPU is assigned as host and controls the data flow (see 3.2.2).

3.1.2 Graphics Processing Unit

Graphics Processing Unit (GPU) is a device originally designed to render graphics. Hence its memory accesses are optimized to process images quickly, which means it also suits the computation of large matrices. The main stimulus of the GPU market was the games industry. This led to an huge development of the GPUs capabilities, to bring realistic graphics to the users, which means that over the past years the gap between CPUs and GPUs has increased tremendously. GPU now supports large number of threads, which allows exploiting thread-level parallelism and thus accelerating compute-intensive tasks like image processing, among others.

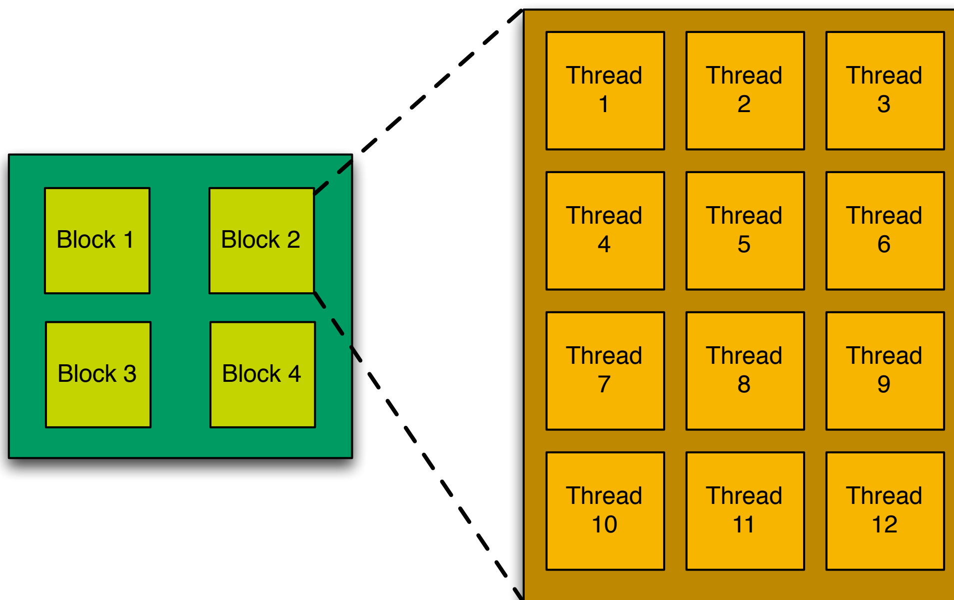

Figure 3.3: GPU thread scheme.

More recently, mobile GPUs have become an important hardware piece, since mobile devices like laptops or smartphones overcame the market. Nowadays, it is possible to obtain powerful graphics with relatively low energy consumption. This also gives the opportunity to exploit thread-level parallelism in mobile devices, just like the dedicated machines.

General-Purpose computation on Graphics Processing Units (GPGPU) [19] is a more recent software concept that takes advantage of these threads to compute traditional CPU tasks on GPUs. A common example is the computation of matrix operations with minimal dependency between data elements, where distinct threads are associated to the processing of different rows and columns. The Open Computing Language (OpenCL) [20] framework exploits this duality and is covered in 3.2.2.

3.2 Parallel programming

As the modern CPUs and GPUs are designed be multi-core devices, software developers start to exploit this feature to compute intensive tasks, distributing the workload among the available threads. Here are presented two approaches that exploit thread-level parallelism.

3.2.1 OpenMP API

Open Multi-Processing (OpenMP) is an Application Programming Interface (API) to create shared-memory parallel programs [21, 22]. It adds notation to sequential C/C++ or Fortran code to specify how some code blocks are shared among processors and/or cores. Recalling subsection 3.1.1, it takes advantage of the CPU shared memory to distribute the workload. The success of this API is its simplicity, attaching #*pragma* directives to the desired code block, letting the compiler handle the details. This gives to the OpenMP the portability to several platforms.

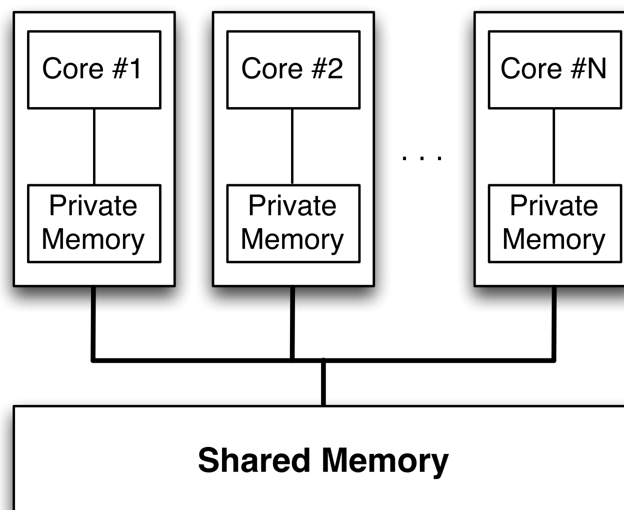

Figure 3.4: OpenMP memory scheme.

The idea of OpenMP is to break a piece of code into multiple instances, and assign a thread to each instance. Each thread runs independently and when the work is done, they assemble data and proceed to sequential code. This is a fork-join programming model [23], as illustrated in figure 3.5.

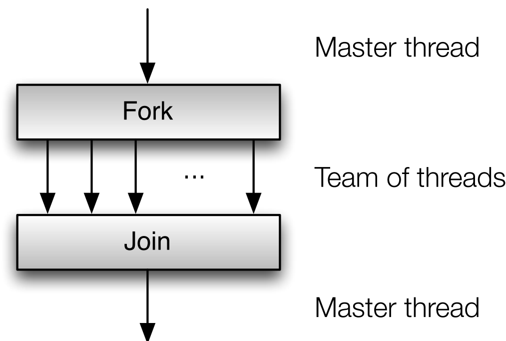

Figure 3.5: The fork-join programming model [22].

The program is running in a single thread fashion, afterwards splitted into a team of threads (fork), depending on how many cores are available and how many threads are explicitly created. After the work is done, the original thread (which is the master of the team) continues its way and all others terminate (join). Each portion of the code between fork and join is called a parallel region.

The OpenMP memory model

Although OpenMP works under shared memory, each thread may need private memory. By default data is shared and visible among threads, but the programmer may declare a variable as private so each thread has a copy of that variable on its own memory region called thread stack. The use of private variables in some contexts may accelerate the parallel region because it saves shared memory accesses which are slower than private memory accesses. The programmer does not need to to concern about architecture details since OpenMP has its own rules and mechanism to rule shared and private objects. To ensure that the thread calling has the same values for shared data objects, OpenMP provides an operation called flush. This guarantees that thread updates are written back to shared memory, providing the updated values to the other threads.

3.2.2 OpenCL framework

OpenCL is a framework managed by the Khronos Group [18, 20, 24] that executes across heterogeneous platforms like CPUs, GPUs and other type of devices supported by computer manufacturers. The computing system consists of an host processor (typically a CPU) that is responsible for logical operations and data flow, and a device (e.g. a GPU) that maps data to its memory system and performs the massive processing of data using high-levels of data parallelism.

The OpenCL programming model can be described in a hierarchy of 3 models [25]:

- Platform Model
- Memory Model
- Execution Model

Platform model

The platform model is where the software is linked to the hardware. It consists of a host connected to one or more OpenCL devices, that are divided into one or more compute units (CU), which are further divided into processing elements (PE). PEs execute kernels, which are the device programs that may be compiled before or during program execution.

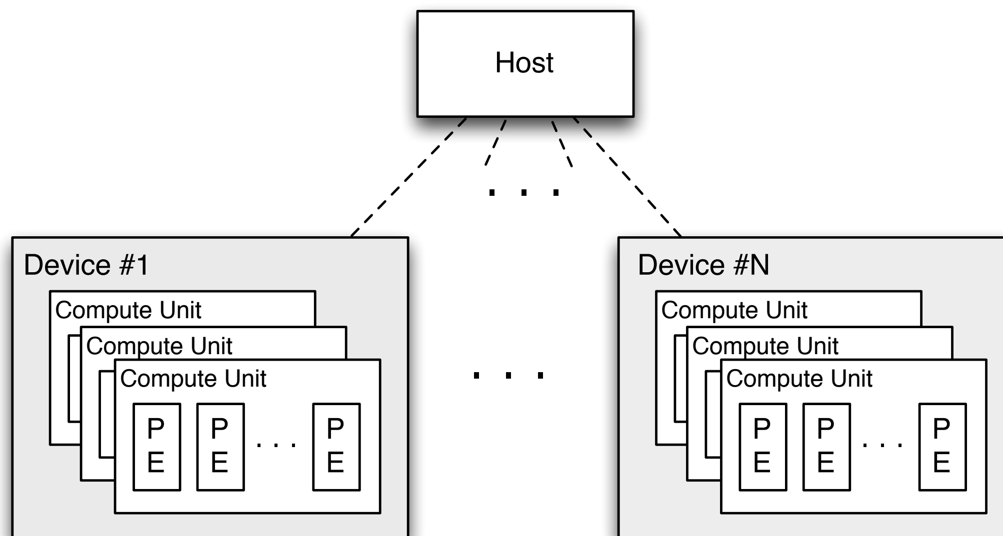

Figure 3.6: OpenCL platform model representing N devices orchestrated by a host.

Execution model

Before the device is able to run OpenCL kernels, a set of instructions must be performed so that data is properly enqueued. First of all a context must be created to coordinate the host-device interaction and memory objects. Basically it keeps track of programs and kernels created for each device. Communication between host and device occurs using a command queue (as the name implies, commands are sent over this queue). The data mapping is done using buffers that are then enqueued to the device. Write operations represent the most significant overhead of an OpenCL execution context, and thus the

programmer must be careful to ensure minimal number of transactions between host and device.

Finally an N-dimensional index space, called NDRange, must be enqueued to inform the device how to partition data over thread blocks called work-groups. A work-group defines a number of work-items that run an instance of the desired kernel.

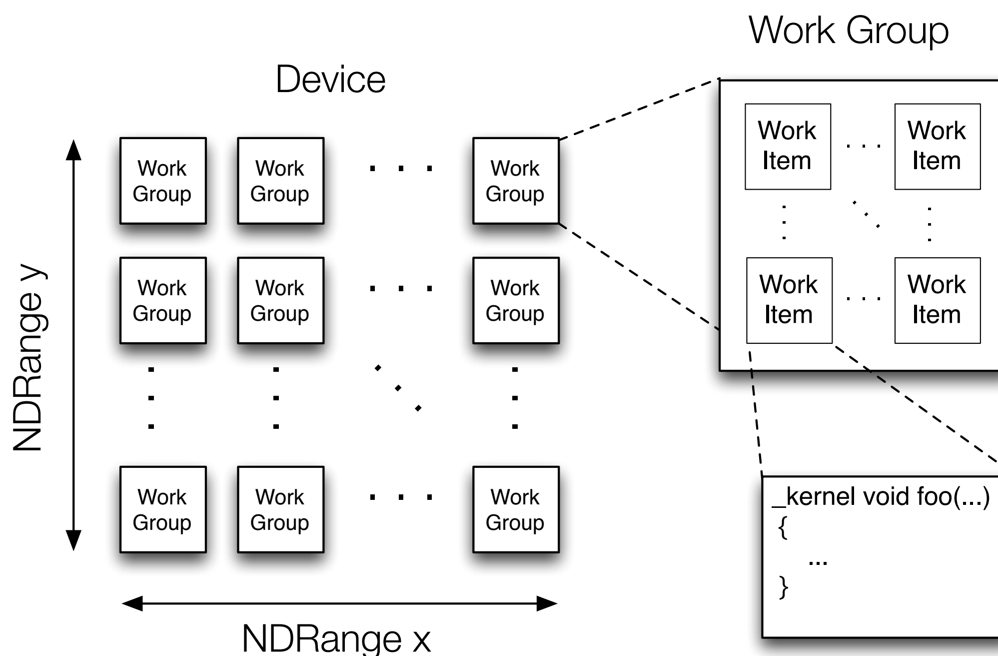

Figure 3.7: NDRange index space with work-groups and work-items scheme.

Memory model

Since OpenCL is a portable framework, memory architecture may vary between computer manufacturers. Due to this fact, the memory model defines an abstraction that programmers can target and manufacturers can map to the real memory system.

Memory model is hierarchically divided in 4 layers:

- **Global memory**: It is the top level memory. It is visible to all work-groups.
- **Constant memory**: Only allows read operations. It is also visible to all work-groups.
- **Local memory**: It is the memory region of work-groups. It can be accessed by all work-items inside a work-group.
- **Private memory**: It is the memory region of work-items. It is private for each work-item.

An accurate memory management performed by the programmer may increase kernel performance since private and local memory are tightly closer to the core and thus faster than global memory.

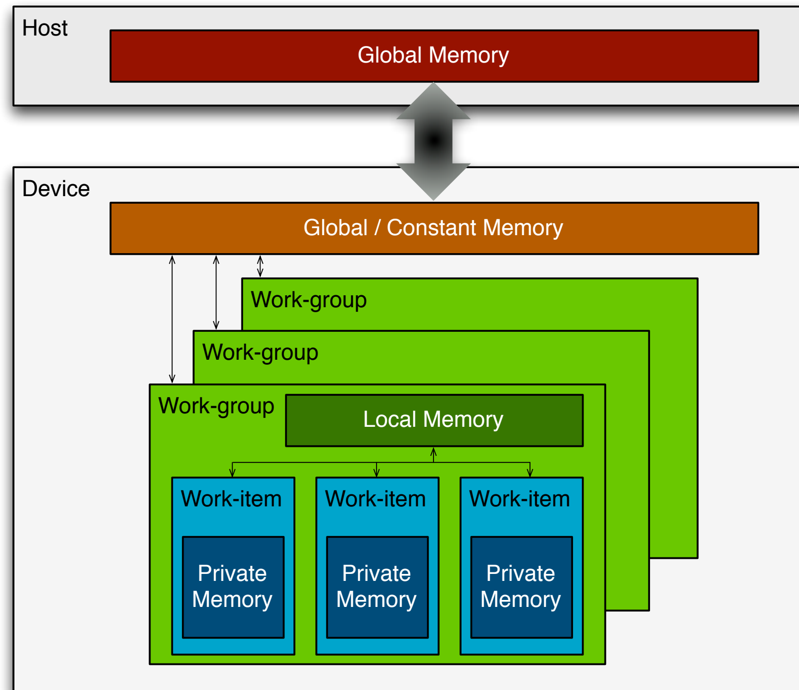

Figure 3.8: OpenCL memory model.

3.3 Summary

This chapter reviews some hardware concepts and introduces the parallel computing paradigm. The OpenMP is an API that supports shared memory parallel programs. It takes advantage of a multi-core platform to share the workload among cores, adding *#pragma* directives on the desired portions of the code. It has a very simple and clean syntax and is supported by many multi-core platforms, which makes the OpenMP a portable API. The OpenCL framework is designed to take advantage of massive thread-level parallelism. Its platform model is composed by a host that perform the logical operations and data flow, and a device that is where the workload is distributed.

Now that all theoretical concepts were introduced, the next chapter explains the techniques used to parallelize the RaptorQ decoder schemes.

4

RaptorQ decoder on Snapdragon CPU/GPU

Recalling sections 2.5 and 2.6, both decoding schemes of RaptorQ have common mathematical operations which are suitable for parallelization. Those are matrix inversions and matrix multiplications and, in a practical manner, they require big loops to be performed. However, these two operations are significantly different in terms of data dependency. While in multiplication each resulting matrix position is totally independent of the others, in inversion they are dependent of each iteration of the Gauss-Jordan elimination method. This means that in heterogeneous computing, inversion inserts much more overhead than multiplication because of the memory operations needed between host and device to perform data flow and keep data coherence. Due to this fact, three parallelization schemes are proposed. For the multiplication we use OpenCL framework to exploit massive thread-level parallelism and for the inversion we use an OpenCL and an OpenMP approach, to exploit parallel regions, since OpenMP introduces low overhead launching threads and works at CPU level.

4.1 Parallelization of matrix multiplication

Given two matrices A and B, the product $C = A \times B$ is given by multiplying A rows by B columns, i.e:

$$C_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj}. \tag{4.1}$$

This operation implies three nested loops since we need to associate all A rows to all B columns (two loops), and for each one we need to perform the sum of the product (4.1) (one loop). Since the two outer loops are independent, the strategy of parallelization is to assign one thread to each (i, j) pair and each thread perform the inner loop, as exemplified in figure 4.1.

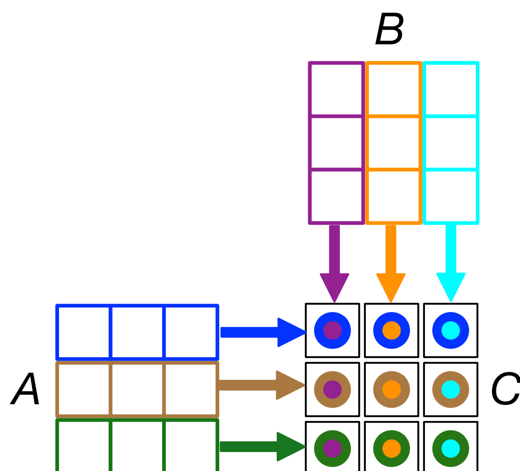

Figure 4.1: Matrix multiplication scheme. Each pair of colors represents one thread.

Algorithm 1 Matrix multiplication in sequential C

```
for(i = 0; i < cols_b; i++)
{
    for(j = 0; j < rows_A; j++)
    {
        C[j][i] = 0;

        for(k = 0; k < cols_A; k++)
            C[j][i] = C[j][i] + (A[j][k] * B[k][i]);
    }
}
```

Algorithm 2 Parallel matrix multiplication on GPU

Require: Matrix $A_{m\times n}$ and matrix $B_{n\times p}$

1: **procedure** gfMatrixMul(A, B, m, n, p)

2: (**Host** $\rightarrow$ **Device**) Copy matrix A, matrix B and matrix C from host to device.

3: (**Kernel**) Perform matrix multiplication $A \times B$ and stores the result on matrix C.

4: (**Device** $\rightarrow$ **Host**) Copy matrix C from device to host

5: **end procedure**

Algorithm 3 Matrix multiplication kernel in OpenCL

```
__kernel void matrixMul(__global unsigned char *a, __global unsigned char *b,
    __global unsigned char *c, int rows_a, int cols_a, int rows_b, int cols_b)
{
    int row = get_global_id(1);
    int col = get_global_id(0);

    unsigned char sum = 0;

    for(int counter = 0; counter < cols_a; counter++)
        sum = sum + (a[row * cols_a + counter] * b[counter * cols_b + col]);

    c[row * cols_b + col] = sum;
}
```

4.2 Parallelization of matrix inversion

Assuming that the matrix to invert is full rank (i.e. is invertible) [26], we will use Gauss-Jordan method to perform the inversion. Gauss-Jordan method is suitable for parallelization comparing with other methods of matrix inversion [27, 28] because each iteration has parallel regions that are only dependent of the values of that iteration. This method consists in a sequence of elementary row operations to nullify the values of the left side, giving rise to the inverse matrix on the right side, as shown in figure 4.2. The diagonal elements are called pivots.

$$\left[\begin{array}{cccc|cccc} x_{11} & x_{12} & \dots & x_{1j} & 1 & 0 & \dots & 0 \\ x_{21} & x_{22} & \dots & x_{2j} & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{i1} & x_{i2} & \dots & x_{ij} & 0 & 0 & \dots & 1 \end{array}\right] \rightarrow \left[\begin{array}{cccc|cccc} 1 & 0 & \dots & 0 & y_{11} & y_{12} & \dots & y_{1j} \\ 0 & 1 & \dots & 0 & y_{21} & y_{22} & \dots & y_{2j} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 & y_{i1} & y_{i2} & \dots & y_{ij} \end{array}\right] \quad (4.2)$$

There are 3 elementary row operations performed in the algorithm:

- **Row swapping**: Swap rows when pivot is zero to keep diagonal without null values.
- **Row division**: Division of row i by its pivot.
- **Row sum**: Sum one row to another to nullify column j .

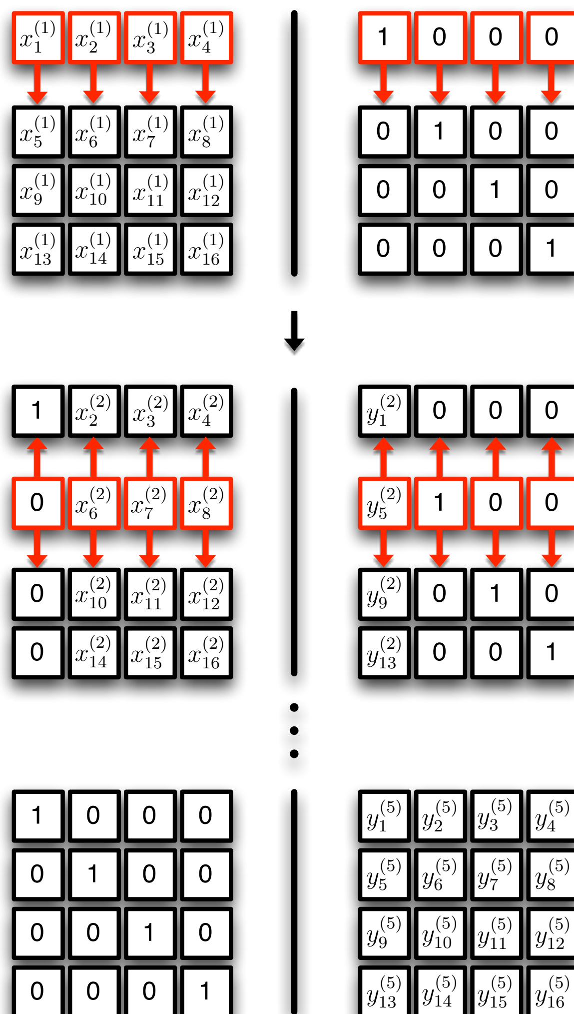

Figure 4.2: Scheme of a 4×4 matrix inversion. Each red row represents one iteration of the algorithm. Row is divided by its pivot, so its value is set to one. Then, that row is summed to the others to nullify the column values.

Since in each iteration of the algorithm, both left and right side of the matrix are updated and in each one of them the pivot position and value must be checked, this cannot be parallelized due to data dependencies. Hence, we exploit parallelism over the three elementary row operations.

4.2.1 OpenCL approach for matrix inversion

The first and second elementary operations are performed by a loop that goes through the row. The strategy of parallelization is to launch one thread for each iteration of the loop.

Algorithm 4 Row swap in sequential C

```
// i -> pivot position / row position

for(k = 0; k < cols; k++)
{
    temp = matrix[j][k];
    matrix[j][k] = matrix[i][k];
    matrix[i][k] = temp;

    temp = inverse[j][k];
    inverse[j][k] = inverse[i][k];
    inverse[i][k] = temp;
}
```

Algorithm 5 Row swap kernel in OpenCL

```
__kernel void rowSwap(__global unsigned char *matrix, int j, int max_cols, int i)
{
    int col_pos = get_global_id(0);

    unsigned char aux;

    aux = matrix[j * max_cols + col_pos];
    matrix[j * max_cols + col_pos] = matrix[i * max_cols + col_pos];
    matrix[i * max_cols + col_pos] = aux;

}
```

Algorithm 6 Row division in sequential C

```
// i -> pivot position / row position

for(k = 0; k < cols; k++)
{
    matrix[i][k] = matrix[i][k] / aux;
    inverse[i][k] = inverse[i][k] / aux;
}
```

Algorithm 7 Row division kernel in OpenCL

```
__kernel void pivotDivision(__global unsigned char *matrix, int i, int cols,
    unsigned char pivot)
{
    int col_pos = get_global_id(0);

    matrix[i * cols + col_pos] = matrix[i * cols + col_pos] / pivot);
}
```

The third operation consists of two nested loops. The outer loop goes through all rows and the inner loop goes though all columns. The strategy is to assign one thread for each sum operation between columns, that is, for each red cell of figure 4.1 pairing with the corresponding black cells.

Algorithm 8 Row sum in sequential C

```
// i -> pivot position / row position

for(k = 0; k < rows; k++)
{
    if (matrix[k][i] != 0  &&  k != i)
    {
        aux = matrix[k][i];

        for(m = 0; m < cols; m++)
        {
            matrix[k][m] = (aux * matrix[i][m]) - matrix[k][m];
            inverse[k][m] = (aux * inverse[i][m]) - inverse[k][m];
        }
    }
}
```

Algorithm 9 Row sum kernel in OpenCL

```
__kernel void linearOperation(__global unsigned char *matrix, int pos, int max_cols
    , __global unsigned char *matrix_aux)
{
   int row_pos = get_global_id(1);
   int col_pos = get_global_id(0);

   if(matrix_aux[row_pos * max_cols + pos] != 0 && row_pos != pos)
   {
      matrix[row_pos * max_cols + col_pos] = (matrix_aux[row_pos * max_cols + pos]
          * matrix[pos * max_cols + col_pos]) - matrix[row_pos * max_cols + col_pos
          ];
   }
}
```

The threads associated to the row of the pivot do not perform any task, since that row does not have any column nullified in that iteration. The full algorithm is shown bellow.

Algorithm 10 Parallel Gauss-Jordan on a GPU

Require: Matrix $A_{m \times n}$, with $m \geq n$
1: **procedure** gfInv(A, m, n)
2: (**Host compute**) Create an extended matrix B with matrix A on the left side and the identity I on the right side;
3: (**Host** $\rightarrow$ **Device**) Write the extended matrix;
4: **for** $i \leftarrow 0$, $(n-1)$ **do**
5: $j \leftarrow i$;
6: (**Host compute**) Check if pivot is different from zero, $j \leftarrow$ pivot position;
7: **if** $j \neq i$
8: (**Kernel 1**) Swap row i with row j;
9: (**Device** $\rightarrow$ **Host**) Read extended matrix;
10: **end if**
11: **if** $E[i][i] \neq 1$ and $E[i][i] \neq 0$
12: (**Host compute**) Saves the pivot value;
13: (**Kernel 2**) Divide row by the pivot;
14: **end if**
15: (**Device compute**) Stores a copy of the extended matrix;
16: (**Kernel 3**) Perform the sum operations;
17: (**Device** $\rightarrow$ **Host**) Read extended matrix;
19: (**end for**)
19: (**Host compute**) Splits the extended matrix
20: **end procedure**

In each iteration, the host must access the values of the extended matrix. This implies one or two read operations from the device to host. To keep data coherence, a copy of the extended matrix is done every iteration, because when kernel 3 is launched it is not guaranteed any thread order execution and that may lead to an access of updated values by later threads.

Since this approach implies multiple communications between host and device during each iteration, a lighter approach is presented bellow.

4.2.2 OpenMP approach for matrix inversion

Recalling section 3.2.1, a parallel region is created by adding #*pragma* directives. Thus, the only overhead introduced by OpenMP is to copy the registers to the shared memory, and when the parallel region terminates, data remains on the CPU. However, OpenMP is limited by the number of cores of the CPU, which are fewer than the GPU.

The strategy of parallelization is similar to the previous subsection, but what OpenMP does in practice is slightly different. It breaks the loop and each thread executes a portion of that loop. So for the first and second elementary operation, a simple #*pragma* directive is added before the loop, and the workload is distributed among the available threads.

Algorithm 11 Row swap with OpenMP

```
// i -> pivot position / row position

#pragma omp parallel for
for(k = 0; k < cols; k++)
{
    temp = matrix[j][k];
    matrix[j][k] = matrix[i][k];
    matrix[i][k] = temp;

    temp = inverse[j][k];
    inverse[j][k] = inverse[i][k];
    inverse[i][k] = temp;
}
```

Algorithm 12 Row division with OpenMP

```
// i -> pivot position  / row position

#pragma omp parallel for
for(k = 0; k < cols; k++)
{
    matrix[i][k] = matrix[i][k] / pivot);
    inverse[i][k] = matrix[i][k] / pivot);
}
```

Similarly to the problem of data coherence in the previous section, some variables must be kept private for each thread to ensure that there isn't any read operation from an updated value by later threads. Thus, the indexes and the values of the row of the pivot (red row of figure 4.1) are private for each thread. The outer loop is broken and each portion is assigned to one thread. The $(k-1)$ times that the inner loop executes is thus distributed among threads.

Algorithm 13 Row sum with OpenMP

```
// i -> pivot position  / row position

#pragma omp parallel for private(m, aux)
for(k = 0; k < rows; k++)
{
    if (matrix[k][i] != 0  &&  k != i)
    {
        aux = matrix[k][i];

        for(m = 0; m < cols; m++)
        {
            matrix[k][m] = (aux * matrix[i][m]) - matrix[k][m];
            inverse[k][m] = (aux * inverse[i][m]) - inverse[k][m];
        }
    }
}
```

4.3 Summary

This chapter presented the techniques developed to exploit data parallelism on the RaptorQ decoder. Two operations were identified to be well suited for parallelization: matrix multiplication and matrix inversion. Although both exhibit parallel characteristics, their algorithms are very different due to data dependency. Matrix multiplication is performed with three nested loops and each iteration of the two outer loop does not depend on the others. So the outer loops are fully parallelized, leaving the inner loop for each thread to perform. Regarding matrix inversion, each iteration depends on the previous. Hence, three portions of the code were identified to be parallelized: row swapping, row division and row sum. The three operations are performed by loops, so the strategy is to launch one thread for each iteration of the loops, taking into account data coherency.

Although the matrix multiplication can fully take advantage of the OpenCL capabilities, the matrix inversion introduces significant overhead. For that reason, two parallel approaches were introduced to understand which one suits better to the problem. The next chapter shows the results of these techniques on the Snapdragon SoC.

5

Experimental results

This chapter shows the results of the techniques presented on the previous chapter. First, both decoding schemes of sections 2.5 and 2.6 are compared in terms of decoding time. It is shown the speedup for the best decoding scheme concerning a parallel approach for higher loss rates. Regarding the matrix inversion, a comparison between OpenCL and OpenMP computing time is made. Finally, it is shown the enhancement of the precode matrix of RaptorQ construction time between a sequential C and a OpenCL approach.

Two platforms were used to set up and test the proposed schemes:

	MacBook Air (13-inch, Early 2014)	**DragonBoard Development Kit with a Snapdragon 800**
CPU	1,4 GHz Intel Core i5	Quad-core Qualcomm® Krait™ 400 CPU at up to 2.3 GHz per core
RAM	4 GB 1600 MHz DDR3	2GB LPDDR3 memory
GPU	Intel HD Graphics 5000 1536 MB	Qualcomm® Adreno™ 330 GPU
OS	Yosemite version 10.10.3	Android 4.3 with kernel version 3.4.0-g0b717d1-00010-gbe2b492
OpenCL Version	1.2	1.1
Compiler	Apple LLVM version 6.1.0	Android NDK, GNU Make 3.81

Table 5.1: Test environment specs.

5.1 Decoding time comparison of the two decoding schemes

First of all, the two decoding schemes are compared to understand which one has better decoding time. It is assumed that 1 source symbol is missing. The source block size K is between 10 and 2005, with symbol size T between 4 and 1024.

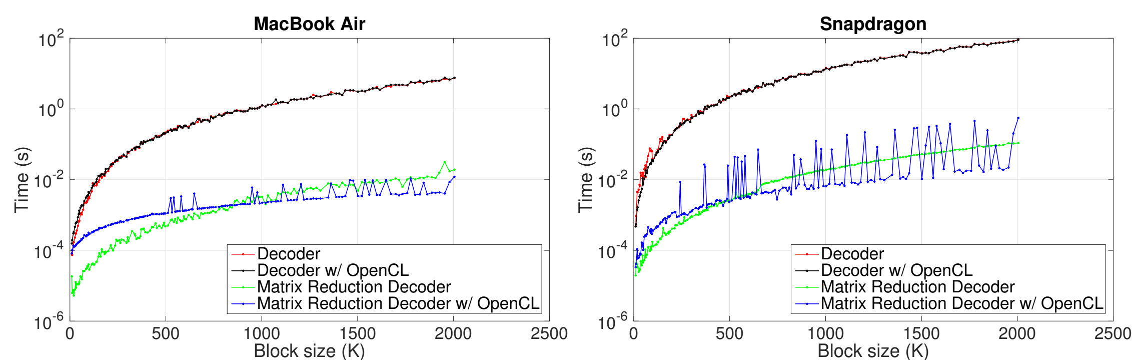

Figure 5.1: Symbol size $T = 4bytes$.

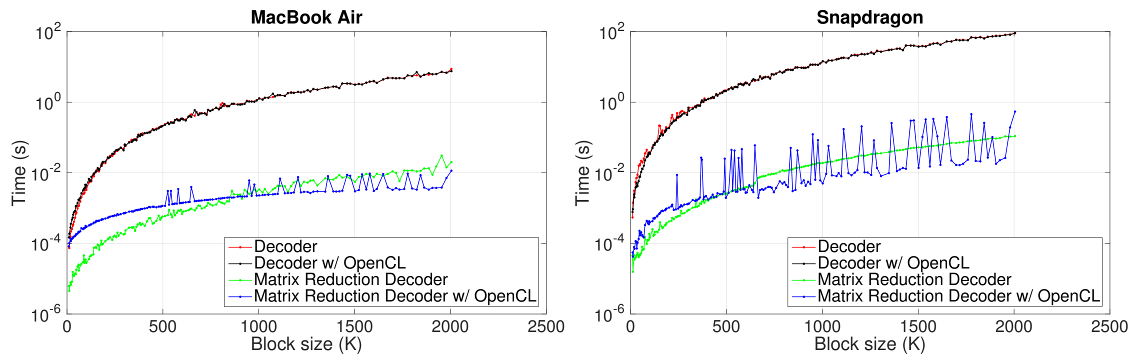

Figure 5.2: Symbol size $T = 8bytes$.

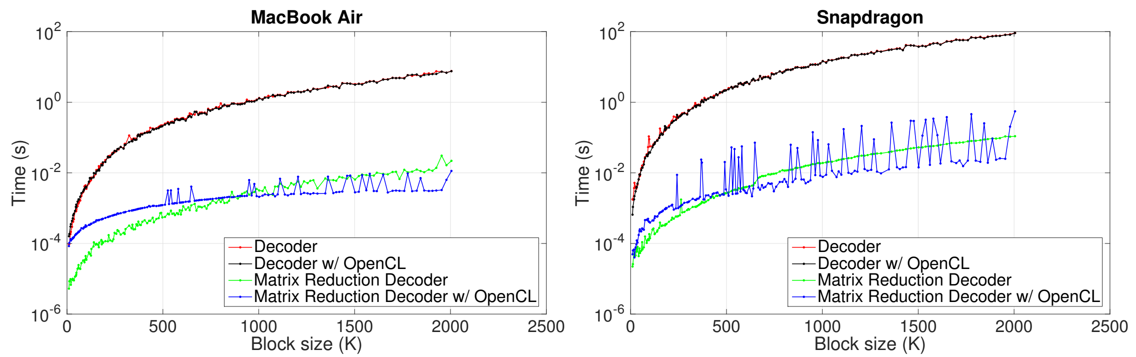

Figure 5.3: Symbol size $T = 16bytes$.

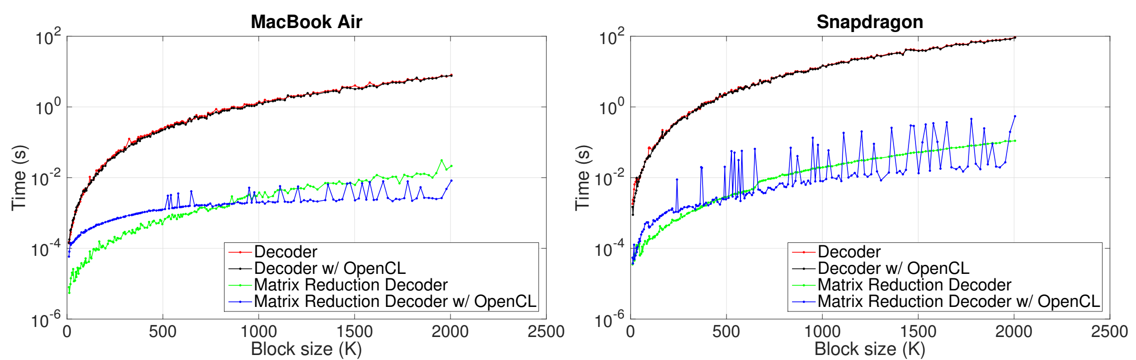

Figure 5.4: Symbol size $T = 32bytes$.

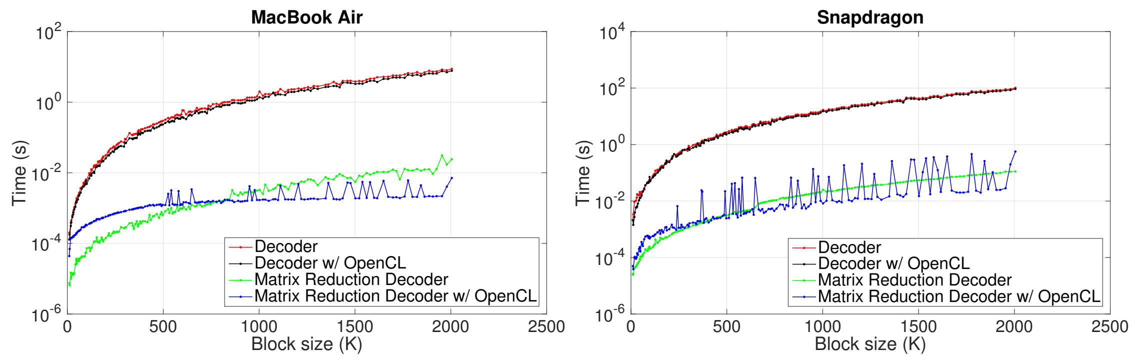

Figure 5.5: Symbol size $T = 64 bytes$.

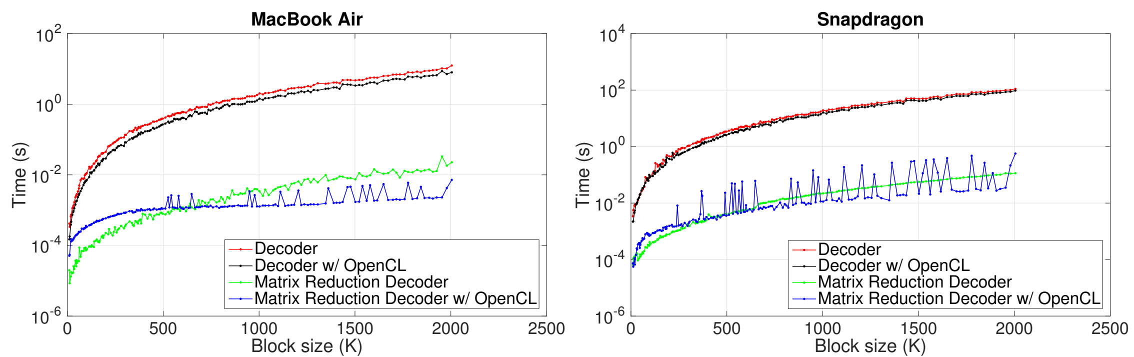

Figure 5.6: Symbol size $T = 128 bytes$.

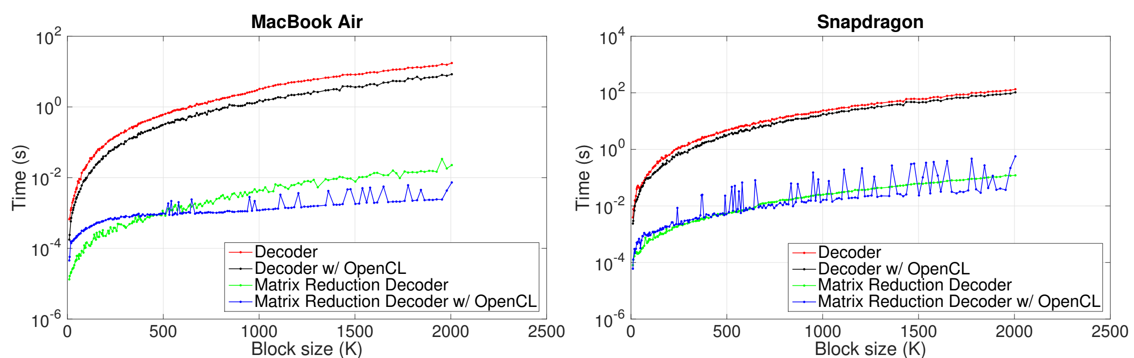

Figure 5.7: Symbol size $T = 256 bytes$.

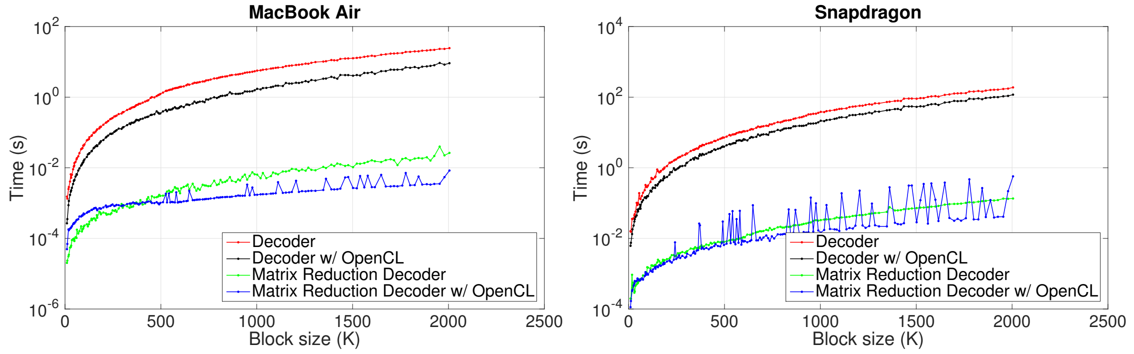

Figure 5.8: Symbol size $T = 512bytes$.

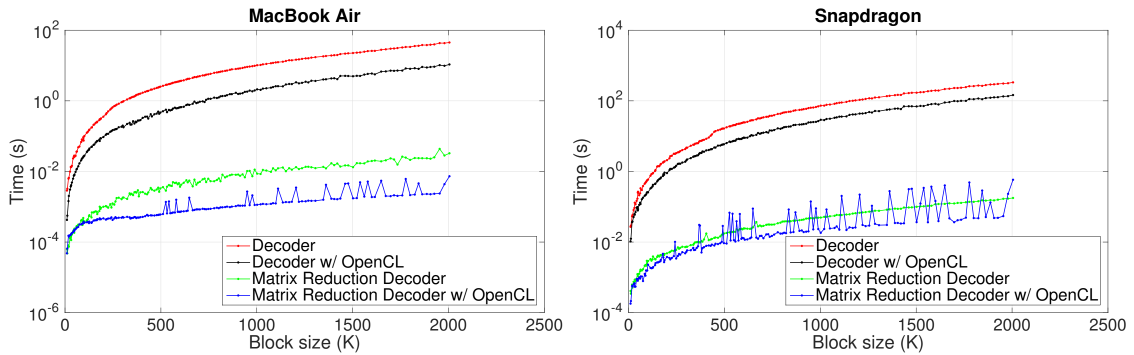

Figure 5.9: Symbol size $T = 1024bytes$.

The matrix reduction decoder has better decoding time, even without OpenCL. The peaks on the blue curves represent the matrices whose sizes do not fit well on the OpenCL execution model (by default). That is, the NDRangeY and NDRangeX are such that the number of work-groups and their size induces a bad thread scheduling. However, choosing proper values, OpenCL exhibits a speedup concerning the sequential approach.

Hereupon, the matrix reduction decoder is the chosen scheme to be tested for higher loss rates.

5.2 Matrix reduction decoder

To set up an approach closer to reality, these simulations measure the decoding time of a corrupted source block under a erasure channel with three loss rates: 1%, 10% and 20% of missing source symbols. The scheme compares a sequential C approach of the decoder with an OpenCL optimized scheme, and is tested for a block size $T = 512bytes$ and $T = 1024bytes$.

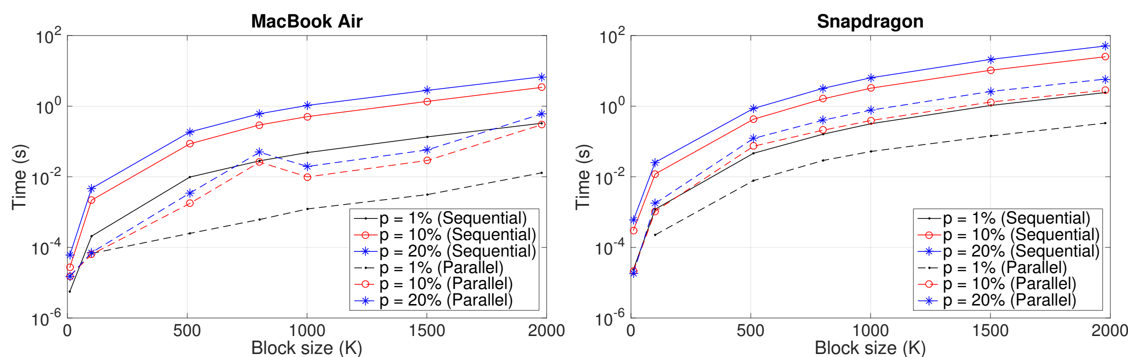

Figure 5.10: Symbol size $T = 512 bytes$.

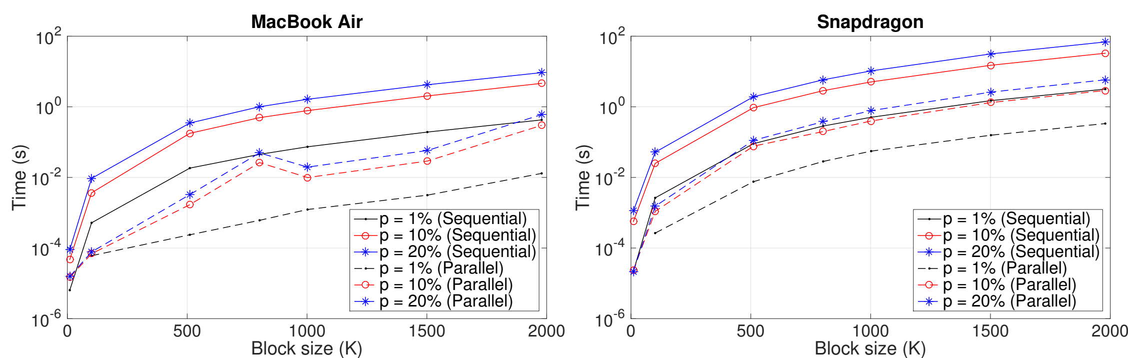

Figure 5.11: Symbol size $T = 1024 bytes$.

OpenCL exhibits an huge improvement on the decoding time for each case, even for small source blocks. The following tables summarize the speedup of the OpenCL approach relatively to the sequential C, on the Snapdragon 800 platform.

Table 5.2: Snapdragon speedups for a block size $T = 512 bytes$

Loss rate (%) \ Block size (K)	**10**	**101**	**511**	**802**	**1002**	**1502**	**1979**
1	-	5.3x	5.9x	5.6x	6.1x	7.3x	7.3x
10	14.2x	11.4x	5.8x	7.7x	8.3x	8.0x	8.9x
20	32.3x	14.6x	7.0x	8.0x	8.2x	8.1x	8.6x

Table 5.3: Snapdragon speedups for a block size $T = 1024 bytes$

Loss rate (%) \ Block size (K)	**10**	**101**	**511**	**802**	**1002**	**1502**	**1979**
1	-	10.0x	11.9x	10.0x	9.1x	9.5x	9.5x
10	24.1x	22.8x	12.3x	14.4x	12.8x	11.4x	11.4x
20	51.8x	34.0x	17.2x	15.4x	13.3x	12.1x	11.8x

Concerning power consumptions, OpenCL exhibits similar maximum instant consumptions, with a slight power decrease.

Table 5.4: Snapdragon maximum instant power consumptions for a sequential C approach of the RaptorQ decoder.

Block size (K) / Loss rate (%)	**10**	**101**	**511**	**802**	**1002**	**1502**	**1979**
1	-	0.5 W	1.4 W	1.5 W	1.5 W	1.5 W	1.5 W
10	-	0.8 W	1.5 W	1.5 W	1.5 W	1.5 W	1.5 W
20	-	0.8 W	1.5 W	1.5 W	1.5 W	1.5 W	1.5 W

Table 5.5: Snapdragon maximum instant power consumptions for an OpenCL approach of the RaptorQ decoder.

Block size (K) / Loss rate (%)	**10**	**101**	**511**	**802**	**1002**	**1502**	**1979**
1	-	0.4 W	1.4 W	1.4 W	1.3 W	1.3 W	1.3 W
10	-	0.2 W	1.4 W	1.4 W	1.3 W	1.3 W	1.3 W
20	-	0.2 W	1.4 W	1.3 W	1.3 W	1.2 W	1.3 W

Despite of they exhibit similar instant power consumptions, the OpenCL approach uses less energy, since the decoder uses less processing time.

5.3 Matrix inversion

Although the matrix $A''_{K^{(R)} \times K^{(R)}}$ is usually small, when the loss rate increases, more repair symbols are needed to recover the source symbols. This corresponds on a increasing of the matrix $A''_{K^{(R)} \times K^{(R)}}$. In such cases, the parallel approach of the matrix inversion may increase even more the speedup shown in the previous section.

Recalling subsections 4.2.1 and 4.2.2, two parallel approaches were presented. The next figure shows the inversion time of a sequential C, an OpenCL and an OpenMP approach.

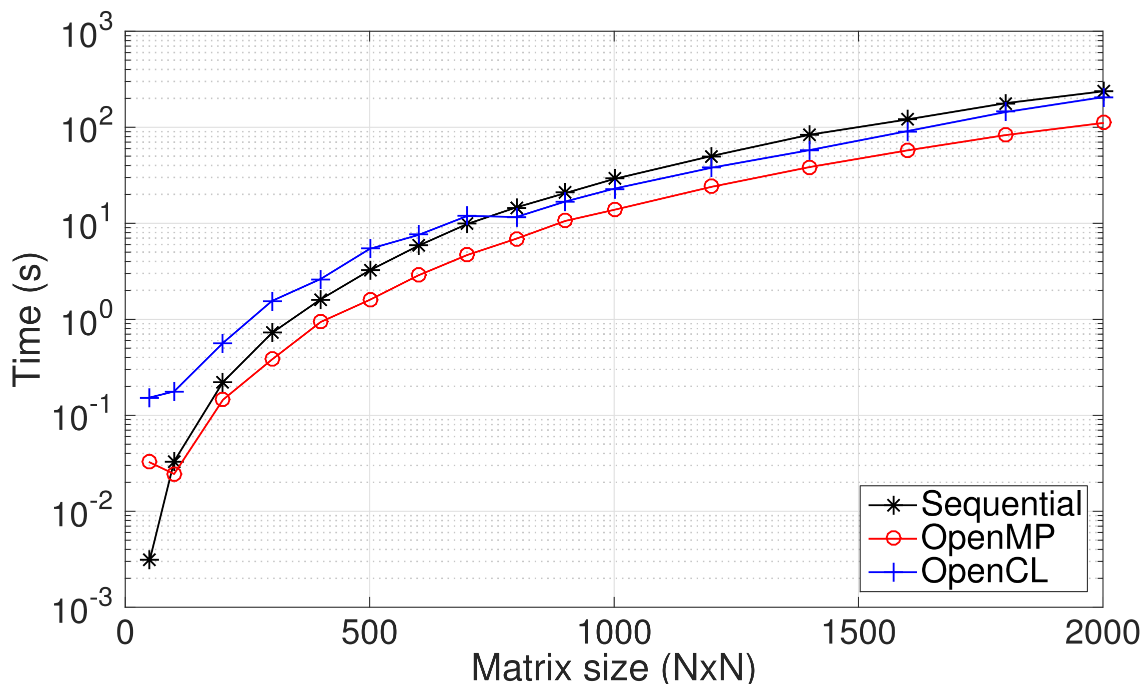

Figure 5.12: Matrix inversion time on Snapdragon.

Figure 5.12 shows that OpenMP exhibits faster inversion time from $N = 100$. Since we are dealing with small matrices, the overhead introduced by OpenCL is hardly recovered by the distributed workload among threads for each iteration. In practice, we are spending more time under memory operations comparing with the speedup of the elementary operations. Instead, OpenMP maps the parallel regions to the shared memory of the CPU and keeps private variables in the private memory, so there is no need of an entire copy between two distant memory regions in each iteration. Thereby, it is possible to improve the speedup of the decoding process using OpenCL for matrix multiplications and triggering OpenMP for matrix inversions when the matrix $A''_{K^{(R)} \times K^{(R)}}$ reaches a predefined threshold.

5.4 Precode matrix

At the beginning of each encoding and decoding process, the precode matrix A must be constructed. Since the algorithm performs a matrix multiplication, it is of interest to see the improvements of the construction time.

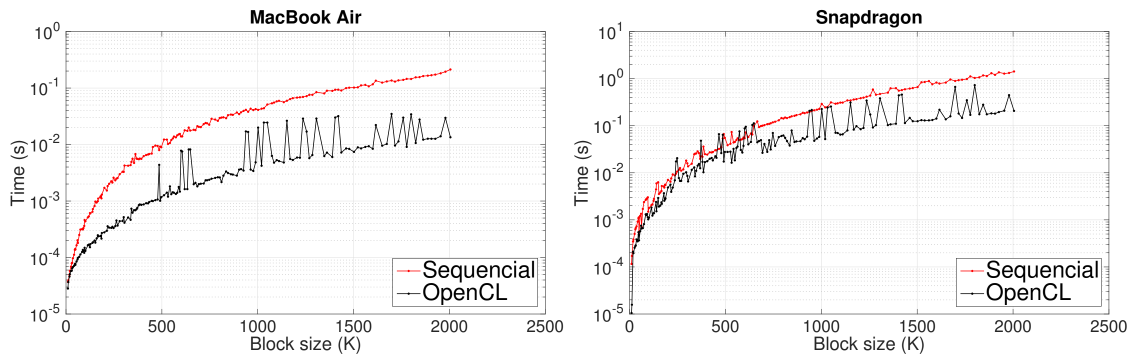

Figure 5.13: Pre-code matrix contruction time.

As expected, an OpenCL approach improves the construction time of the precode. Even without any direct impact on the decoding speedup, since OpenCL runs faster, it has a direct impact on the energy spent by the encoder and the decoder.

5.5 Summary

This chapter presented the results of the techniques proposed on chapter 4. The matrix dimensionality reduction decoder proved to be the best decoding scheme. On all simulations, the parallel approach exhibits significant speedups. However, OpenMP proved to be better than OpenCL for the inversion of small matrices, thus leading to an hybrid parallel approach under a threshold. In terms of energy, the parallel approach allows to lower the consumptions, since the decoder uses less processing time.

6

Conclusions

The goal of this thesis was to accelerate the decoding process of the RaptorQ code. First of all, two decoding schemes were presented. It was shown that the matrix reduction scheme has better decoding time, even without OpenCL. Furthermore, the speedup that is obtained when a matrix multiplication is parallelized can be higher that 10x in a mobile device with similar instant power consumptions. For small matrices, OpenMP shows better inversion time results than the sequential C and OpenCL approaches, which may be used as an hybrid scheme with OpenCL. The precode matrix also shows better construction time when OpenCL is applied, thus accelerating the initiation of both encoding and decoding processes with low power consumptions.

6.1 Future work

Although these techniques exhibit good results, they were not set up to their full capabilities. One barrier of this algorithm is that the size of the matrices are not fixed, and are constantly dependent of the missing symbols. This implies that when OpenCL defines the work-groups for a given NDRange, they may not fit well on the problem. Hereupon, it is proposed to investigate and develop an algorithm to calculate the optimal NDRange and work-group size, based on the source block and the missing symbols, before sharing the workload among threads. Thus, the peaks observed on the figures of the section 5.1 may be smoothed and the speedups may be improved.

It is also proposed to explore loop-unroll techniques with OpenMP, in order to improve the inversion time.

Since all simulations were performed in one development board, it is also recommended to set up these simulations on different mobile architectures to compare the improvements on multiple devices.

Bibliography

[1] N. Jacobson, *Basic algebra I*. Courier Corporation, 2012.

[2] R. Lidl and H. Niederreiter, *Finite fields*. Cambridge university press, 1997, vol. 20.

[3] R. A. Carrasco and M. Johnston, *Non-binary error control coding for wireless communication and data storage*. John Wiley & Sons, 2008.

[4] M. Luby, A. Shokrollahi, M. Watson, T. Stockhammer, and L. Minder, "RaptorQ forward error correction scheme for object delivery (rfc 6330)," *IETF Request For Comments*, 2011.

[5] D. J. MacKay, *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

[6] A. Shokrollahi and M. Muby, *Raptor Codes*. Foundations and Trends in Communications and Information Theory, 2011, vol. 6, no. 3-4.

[7] J. Lopes and N. Neves, "Stopping a rapid tornado with a puff," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 509–523.

[8] M. Luby, "Lt codes," in *null*. IEEE, 2002, p. 271.

[9] A. Shokrollahi, "Raptor codes," *Information Theory, IEEE Transactions on*, vol. 52, no. 6, pp. 2551–2567, 2006.

[10] M. Luby, A. Shokrollahi, M. Watson, and T. Stockhammer, "Raptor forward error correction scheme for object delivery," Tech. Rep., 2007.

[11] X. Guo, G.-X. Zhang, C. Tian, L. Zhang, and W.-D. Zhao, "Fast decoding for raptorQ codes using matrix dimensionality reduction," *Electronics Letters*, vol. 50, no. 16, pp. 1139–1141, 2014.

[12] (2015, August). [Online]. Available: http://wearesocial.net/blog/2015/01/digital-social-mobile-worldwide-2015/

[13] [Online]. Available: http://www.arm.com/

[14] S. B. Furber, *ARM system-on-chip architecture*. pearson Education, 2000.

[15] (2015, August). [Online]. Available: https://www.qualcomm.com/products/snapdragon

[16] Q. Technologies, *SnapdragonTM OpenCL General Programming and Optimization*, Qualcomm Technologies, 5775 Morehouse Drive San Diego, CA 92121 U.S.A., August 2014.

[17] (2015, August). [Online]. Available: http://www.bdti.com/InsideDSP/2012/06/21/Qualcomm

[18] "Heterogeneous computing with OpenCL, author=Gaster, B and Kaeli, DR and Howes, L and Mistry, P, year=2011, publisher=Morgan Kaufmann Pub."

[19] (2015, August). [Online]. Available: http://gpgpu.org/

[20] (2015, August). [Online]. Available: https://www.khronos.org/

[21] (2015, August). [Online]. Available: http://openmp.org

[22] B. Chapman, G. Jost, and R. Van Der Pas, *Using OpenMP: portable shared memory parallel programming*. MIT press, 2008, vol. 10.

[23] J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," *Communications of the ACM*, vol. 9, no. 3, pp. 143–155, 1966.

[24] G. Falcao, V. Silva, L. Sousa, and J. Andrade, "Portable LDPC Decoding on Multicores Using OpenCL," *IEEE Signal Processing Magazine*, vol. 29, no. 4, pp. 81–109, July 2012.

[25] K. O. W. Group *et al.*, "The OpenCL specification, version: 2.0 document revision: 22," *URL http://www. khronos. org/registry/cl/specs/opencl-1.0*, vol. 29, 2014.

[26] D. Lay, *Linear Algebra and Its Applications*. Addison-Wesley, 2012. [Online]. Available: https://books.google.pt/books?id=_4bjtgAACAAJ

[27] G. W. Stewart, *Matrix algorithms volume 2: eigensystems*. Siam, 2001, vol. 2.

[28] D. S. Bernstein, *Matrix mathematics: theory, facts, and formulas*. Princeton University Press, 2009.