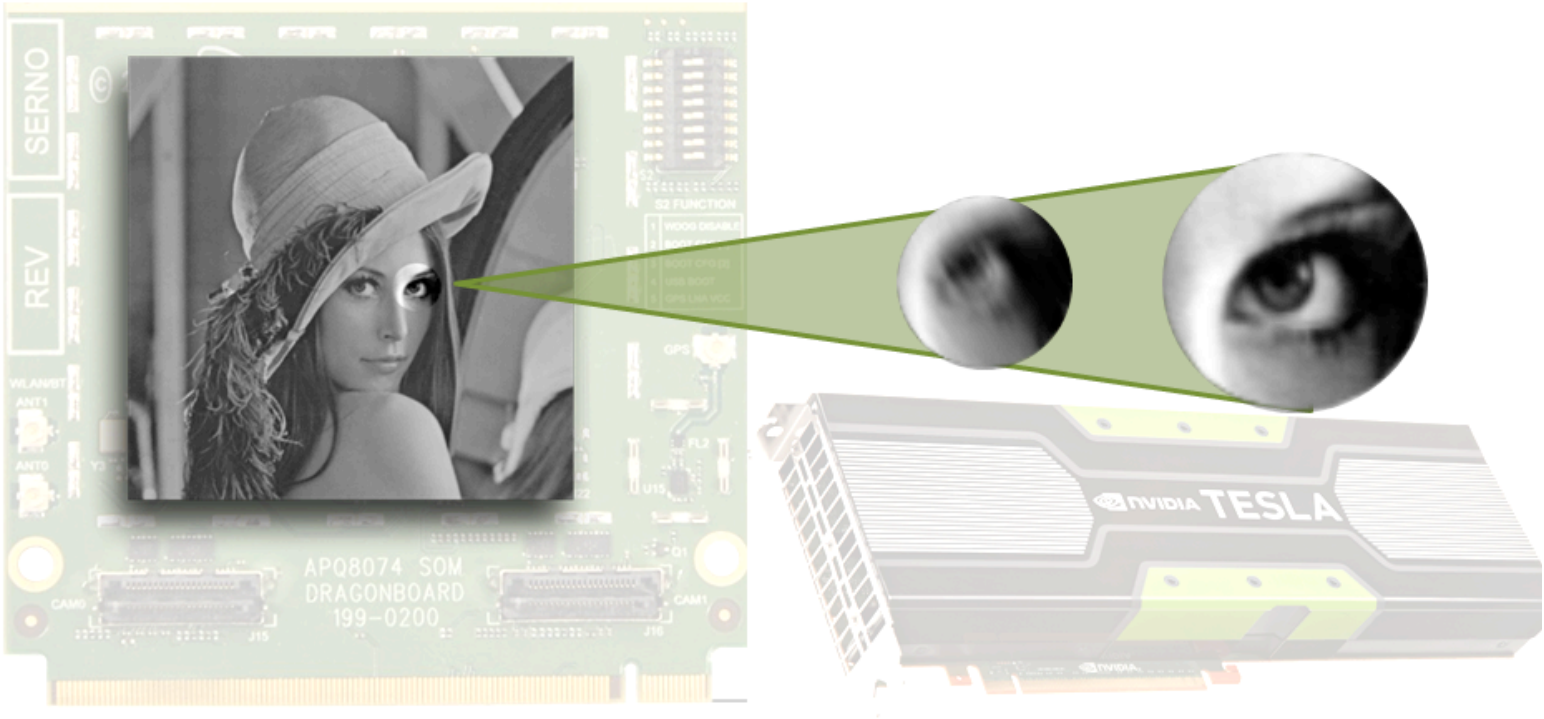


Pedro Nuno Henriques Maia Cordeiro

Efficient and Portable Least Squares Prediction for Image Coding on Manycore Platforms

Dissertação de Mestrado em
Engenharia Eletrotécnica e de Computadores

Setembro de 2015

· U C ·

UNIVERSIDADE DE COIMBRA

Efficient and portable least squares prediction for image coding on manycore platforms

Pedro Nuno Henriques Maia Cordeiro

Dissertação para obtenção do Grau de Mestre em

Engenharia Electrotécnica e de Computadores

Orientador: Doutor Gabriel Falcão Paiva Fernandes

Júri

Presidente:	Doutor Jorge Manuel Moreira de Campos Pereira Batista
Orientador:	Doutor Gabriel Falcão Paiva Fernandes
Vogal:	Doutor Vitor Manuel Mendes da Silva

Setembro de 2015

Agradecimentos

Em primeiro lugar gostaria de agradecer ao meu orientador pela disponibilidade sempre mostrada e pelo conhecimento transmitido. Foi também um elemento importante de encorajamento quando as coisas não estavam a rumar no sentido desejado. Nos bons e nos maus momentos fez juz a esse titulo, orientando-me sempre conforme os resultados que se ia obtendo. Agradecer também aos Professores Patrício Domingues, Sérgio Faria e Nuno Rodrigues, que me deram uma grande ajuda no planeamento inicial e foram continuamente mostrando disponibilidade para me dar feedback quando necessário.

Esta tese teria sido, sem qualquer dúvida, mais difícil sem o apoio e a amizade dos meus colegas. As constantes noitadas no laboratório, os lanches mais longos do que o planeado, entre muitas outras coisas, foram momentos que criaram um ambiente muito benevolente para continuar a trabalhar e ultrapassar os obstáculos.

Por último, e não sem menos importância, agradecer às pessoas mais importantes da minha vida. Os meus pais e os meus irmãos, que sempre estão disponíveis para mim, que me transmitem o que sabem e para com os quais tenho uma dívida eterna. E à minha namorada, pelo apoio, compreensão e dedicação que me faz constantemente crescer como pessoa e como homem.

Queria acabar por dedicar esta dissertação aos meus avós, que estão numa fase terminal das suas vidas. Vocês são um exemplo sempre presente e uma lembrança nunca esquecida.

Abstract

Least squares prediction is a technique used for foreseeing pixel values during image coding by finding the minimum square error of neighboring pixels. It has shown considerable gains in performance, especially for complex images with high variations in pixel intensities (i.e., edges). The drawback of this technique consists of high computational complexity, which makes it difficult to implement in fast, lossy image coders. One challenge of this thesis is therefore to reduce the computational time of this predictor through the use of some parallel techniques, making it more attractive for state-of-the-art Coder-Decoder (CODEC)s. Also, a couple of algorithmic propositions were made, trying to reduce the time spent in exchange for acceptable loss in rate-distortion performance. These propositions are senseful since this predictor is used not only in lossless but also in lossy image coding. Another aim of our work is to analyze energy efficiency among different types of platforms for this signal processing algorithm. We provide comparisons from very powerful to low-cost, General Purpose Graphics Processing Units (GPGPUS).

Keywords

Least squares prediction, Image prediction, Multidimensional Multiscale Parser (MMP), Image coding, Lossy image coding, Parallel processing, Graphics Processing Unit (GPU) processing, Energy efficiency, Manycore platforms

Resumo

A predição pelos mínimos quadrados é uma técnica usada para prever os valores dos píxeis durante o processo de codificação de uma imagem através do cálculo do erro mínimo quadrático dos seus píxeis vizinhos. Esta técnica já mostrou ganhos de qualidade no processo de codificação especialmente para imagens complexas, com grandes variações de intensidades de pixeis (vulgo, arestas). O aspeto negativo desta técnica consiste no seu alto custo computacional, que a torna de difícil implementação em codificadores de imagens rápidos e com perdas. Um dos desafios desta tese é, portanto, o de reduzir o tempo computacional deste preditor através do uso de algumas práticas de computação paralela, que o tornarão mais atrativo a ser incorporado nos Codificadores-Descodificadores (CODECs)s do estado da arte. Par além disso, são aqui propostas algumas melhorias para este preditor em termos algorítmicos, que visam também a reduzir o tempo gasto em troca de alguma qualidade final da codificação. Estas propostas fazem todo o sentido uma vez que este preditor é não só usado em codificação sem perdas, mas também com perdas. Outro objetivo do nosso trabalho consiste em analizar a eficiência energética para várias plataformas de computação para este algoritmo de processamento de sinal. Fazemos comparações desde GPUs de grande potência até GPUs de baixo custo para fins gerais.

Palavras Chave

Predição mínimos quadrados, Predição de Imagem, MMP, Codificação de imagem, Codificação de imagem com perdas, Processamento paralelo, Processamento em GPU, Eficiência energética, Plataformas manycore

Contents

List of Figures

List of Tables

List of Acronyms

GPU Graphics Processing Unit

GPGPUS General Purpose Graphics Processing Units

CPU Central Processing Unit

LSP Least Squares Predictor

MMP Multidimensional Multiscale Parser

CODEC Coder-Decoder

NDK Native Development Kit

OpenCL Open Computing Language

JPEG Joint Photographic Experts Group

FP Flexible Partition

DSP Digital signal processor

RF Radio Frequency

SIMD Single Instruction, Multiple Data

SPMD Single program, Multiple Data

ALU Arithmetic logic unit

API Application Programming Interface

NOP No OPeration

PSNR Peak Signal-to-Noise Ratio

FPGA Field-Programmable Gate Array

MFV Most Frequent Value

1

Introduction

Contents

Prediction is an important process to reduce the amount of information stored in image coding. There are a lot of different prediction modes, each appropriate to a certain set of image characteristics. For smooth images, low complexity predictors may do the job... but for complex, compound images, these predictors may not suffice. For hard images (with sudden variations in pixel intensities, i.e., edges), there is a particular predictor that tends to increase performance: the Least Squares Predictor (LSP). The LSP is especially good at predicting edges, but even for text and compound images, it does not present rate-distortion losses [3]. LSP is a generic predictor thay may be applied to any image coding standard. In this thesis it is applied in the Multidimensional Multiscale Parser (MMP), a context-based, adaptive codec that presents state-of-the-art performance in rate-distortion, but lacks in computational speed.

These predictors are known for having a considerable impact on time consumption and computational complexity and therefore the use in manycore platforms seems completely justified.

Parallel computation is gaining an increased relevance in modern times, as hardware becomes more and more powerful. Signal processing algorithms are generally time constrained and need the acceleration that parallel computation offers. Signal processing images are especially vulnerable to delays. Mobile parallel processing platforms, too, are evolving at a significant rate in terms of the number and power of the cores. They may not have the processing power of modern desktops, but are cheaper and allow achieving higher levels of energy efficiency.

1.1 Motivation

This thesis is all about exploring the possibilities of parallel computing on image codecs and particularly high complexity predictors. This paralelization does not restrict to platform types, as there is a comparison between mobile, desktop and super powerful GPUs. It is a promising area of studies that stands on the idea that large problems can often be divided into smaller ones, which can be solved concurrently, thus reducing the processing time of programs. In order to follow the industry's higher demands in time response we need to explore the hardware capabilities of these devices. An additional motivation for the realization of this thesis is the evolution of energy consumption levels of recent devices... so an analysis on this metric between different platforms is timely and essential for future studies.

The least squares prediction is the most time consumer of the whole MMP algorithm proposed in [3]. This was the main initial motivation for this work.

1.2 Objectives

This thesis aims at exploring the benefits of using parallel computing in signal processing algorithms, specificaly: i) Obtaining time gains for the LSP (and consequently the MMP) and comparing between desktop and mobile processors. Although the LSP produces quality gains for almost all types of images, we need to reduce its time in order to verify whether or not it represents a plausible predictor mode. This was made by the use of several parallelization techniques conveniently illustrated. ii) Analyzing the energy efficiency between different processors. Super-fast GPUs are known for driving very high levels of compute-performance. While this may lead to significant reduction in computation times, it may also lead to little energy efficiency, compared to low-cost, general-purpose processors.

1.3 Main contributions

This thesis shows that the use of parallel computing in signal processing can offer advantages in both time and energy efficiency. The results are thought to be particularly good for complex image codecs (like the MMP), but even for less complex codecs it can be an option to bear in mind. The results achieved with the mobile processor allow us to claim with confidence that mobile platforms can play an important role in the future of signal processing. This work resulted in the article ”Improving least squares prediction for image coding using mobile parallel processing” submitted to the International Conference on Acoustics, Speech, and Signal Processing and is availabe in Appendix A. Moreover, a tutorial on OpenCL development on the Snapdragon processor was documented, to help anyone who wants to enter the world of parallel processing on mobile devices, and is presented in Appendix B.

1.4 Dissertation Outline

This thesis is structured in 6 chapters. It starts with a short introduction followed by the principles of MMP in chapter 2. In chapter 3, we describe a family of mobile platforms in parallel computing and we overview the Open Computing Language (OpenCL) development framework, as well as a guide for development under android devices. In chapter 4 we present some techniques used in the parallelization of LSP and we propose algorithmic optimizations. In chapter 5 we present our results regarding all tested platforms. In the last chapter, a conclusion is made for the results obtained and we also discuss future work, new goals and possibilities.

2

Multidimensional Multiscale Parser

Contents

MMP comes with very interesting ideas of how image compression could be made. Its potential has been taken into consideration and studied in the last years and several approaches have been proposed. For this thesis I use the approach presented in [3] and take the MMP as the basis of my work.

2.1 Block matching

MMP-based encoders make out for their recurrent pattern matching and extensive dictionary search. This dictionary saves the most used elements (codewords) in the coding procedure, and is therefore suitable for recurring input data.
Once a codeword is found in the input data, the corresponding dictionary index is entropy encoded. Therefore, shorter length codes are attributed to frequently occurring codewords. On the opposite side, elements that do not appear in the dictionary are encoded in a less efficient manner. MMP exploits the probability of occurring a match between blocks of different scales (due to the images self-similarity property). This means that parts of the images are compared to others of different scales.

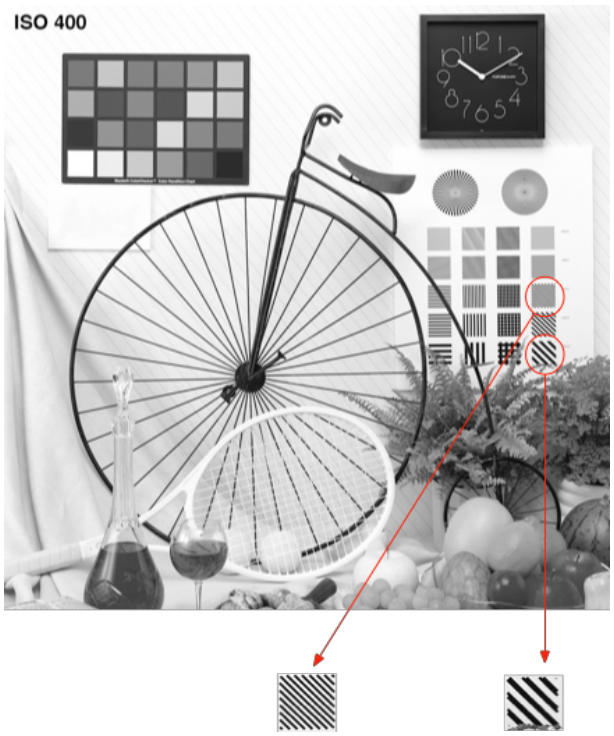


Figure 2.1: Similar parts of the image.

There are two possible outcomes of this. Either a similar match is found, or the block has reached its last possible segmentation. In this case, the value that will be used for the entropy coding is the corresponding pixel.

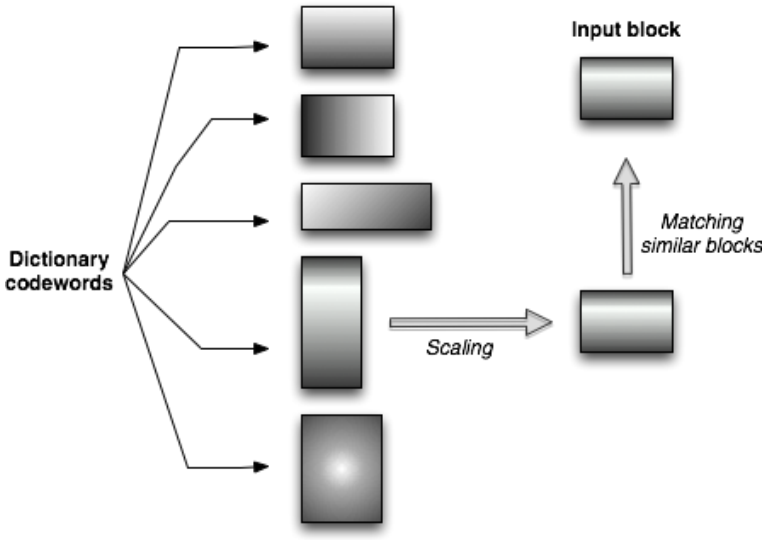


Figure 2.2: Multiscale pattern matching procedure.

The result of this process is a binary segmentation tree. The tree leaves represent the sub-blocks and the tree nodes represent a block segmentation (Figure 2.3).

Figure 2.1 highlights self-similar parts of the bike image and Figure 2.2 illustrates the multiscale pattern matching process. This process is made in order to increase the probability of finding a similar match, since the number of elements tested is greater. Each encoded block is divided if no similar match is found, regarding that particular scale.

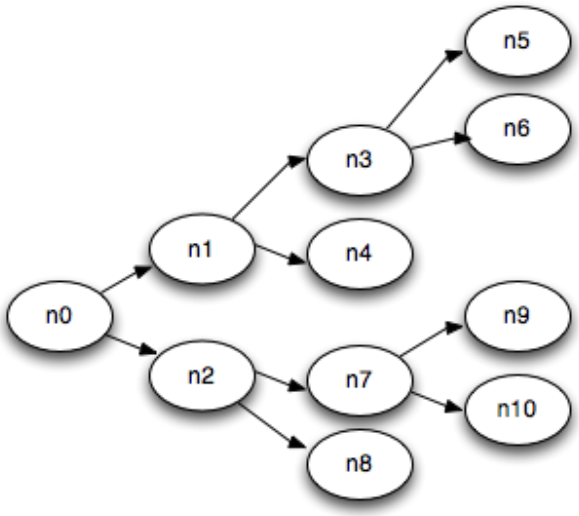


Figure 2.3: Block segmentation.

The resulting bitstream is easy to decode. The segmentation flags sent by the encoder are read by the decoder and a similar tree is constructed with the indexes correspondent

to the dictionary codewords.
Also notice that the values of the output bitstream are encoded along with the block dimension, which enhances the performance of the arithmetic encoder (since the probability of a block to be segmented is different for different sized blocks).

MMP stands out for its ability to be efficiently used in a wide variety of images. In addition to this, MMP showed better results than H.264/AVC Intra mode [4] [5] or Joint Photographic Experts Group (JPEG)2000 [6] for images resulted from computer graphics or even scanned documents. For smooth images, however, it still lacks some robustness.

2.2 Improvements along time (overview)

2.2.1 MMP-Intra

MMP's prediction was based on H.264/AVC standard [4]. For this algorithm, it was applied only for a fixed block size of 16x16. MMP-Intra or MMP-I [7] [8] [9], was the next prediction-based proposal.
Similar to the H.264/AVC standard [10], it used adaptive block size prediction, with multiple block dimensions. There were three possible outcomes to the new block segmentation:

- No segmentation
- Segmentation just on the residue block
- Segmentation on the whole block

The number of prediction modes used by the MMP-I algorithm were also increased (vertical, horizontal, vertical right, vertical left, horizontal up, horizontal down, diagonal down left, diagonal down right and Most Frequent Value (MFV)). The left and upper block neighbors are the most available, due to the scanning order used. For pixels not yet available, the closest available neighbor is then used.
Instead of using the DC mode, MMP-I prefers the use of the MFV among neighboring pixels for the prediction, choosing one that exists in the picture's histogram).

Notice that the prediction step dramatically increases MMP's computational complexity, since the dictionary search has to be made for each mode, in order to determine the residue coding cost.

2.2.2 MMP-II

The following techniques were implemented to create the MMP-II algorithm:

- Scale Restriction: To reduce the number of new elements that are put into the dictionary is the restriction of scales, since scale transformation between too far apart scales may be of no use for the image encoding.
- Norm equalization of scaled blocks: A norm equalization factor is used for scale transforms that increase the block dimension.
- Geometric transforms: Adding blocks generated from geometric transforms, like rotation and symmetry.
- Dictionary Redundancy Control: No need to add new patterns similar to a codeword, since the codeword on the dictionary is already a good enough approximation for the block to be encoded. This process depends on the acceptable distortion level.

These improvements enhance the rate-distortion performance and reduce the computacional cost of the MMP, since the dictionary search process is the main responsible for this.

2.2.3 MMP-FP

The MMP-Flexible Partition (FP) introduces a new idea: sending the segmentation direction to the decoder. Calculations made in [3] show the following results, regarding the MMP computational complexity:

Table 2.1 Number of calls of the search function for the best codeword

Block size	1x1	2x2	4x4	8x8	16x16
MMP	1	3	5	7	9
MMP-FP	1	8	48	224	960
MMP-FP Intra	9	28	500	7604	112340

2.2.4 MMP-FAST

This algorithm focuses on selecting the best prediction mode based on the residue's energy, instead of its coding cost. Since there are less dictionary indexes sent, the bitrate is reduced and block segmentation is avoided.

Altough this process reduces dictionary searches (resulting in a lower computational

complexity), there is a loss in coding performance. With this simple algorithm modification, several dictionary searches are avoided. Results from [3] show a 6.9 times gain in encoding time.

2.3 Least squares prediction

The Least squares prediction method adaptively filters a set of neighbours from each pixel to be predicted. The filter coefficients used are selected based on training over a window containing reconstructed data. This prediction value, Xp, can be calculated as follows:

$$Xp(pos(k)) = \sum_{j=1}^{N} a_j X(pos(k-j)) \tag{2.1}$$

where aj are the weighting factors of the j-th position, N is the number of neighbours, pos(k) is the position of the prediction and pos(k-j) are the neighbours (also called the predictor mask). A causal training window is used to locally optimize the filter coefficients. In [11], a convenient training window corresponds to figure 2.5:

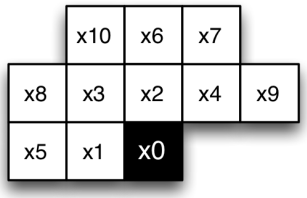


Figure 2.4: Neighborhood of 10 pixels used for prediction of current pixel x0.

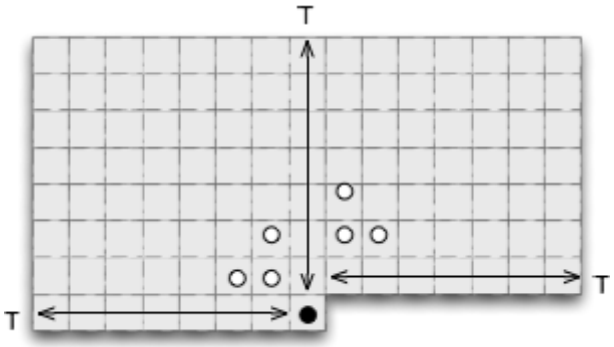


Figure 2.5: Training window with T = 7

The coefficients can be determined by a least squares optimization, minimizing the mean square error, as follows:

$$min||y - Ca||_2 \tag{2.2}$$

And the solution for this might be found in [12]:

$$a = (C^T C)^{-1}(C^T y) \tag{2.3}$$

2.3.1 Predicting arbitrarily oriented edges

If the training window has enough edge pixels (strong variations in pixel intensity), then only one possible solution exists for the predictor's coefficients. If this happens, the prediction will correctly predict the new pixel and the edge direction.
To exemplify this behaviour, we shall consider a second order prediction (i.e., consider just two neighbors: up and left) and a training window of 2 (i.e., for each prediction pixel, there is a double-rectangular window with size of 2 to perform the training, as exemplified in figure 2.6). In our example, the pixel intensities Z and W are very different, i.e.:

$$|W - Z| \gg 0 \tag{2.4}$$

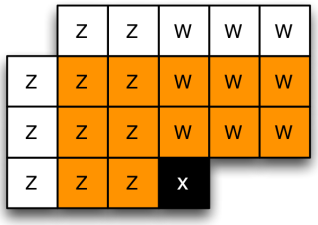


Figure 2.6: The orange pixels correspond to the training window of the black pixel.

The y vector corresponds to the training sequence (orange pixels). For each pixel in the training window, we consider the up and left neighbors, forming the C matrix (the first column corresponds to the left neighbor and the second column to the upper neighbor).

$$y = \begin{bmatrix} Z \\ Z \\ Z \\ Z \\ Z \\ Z \\ W \\ W \\ W \\ W \\ W \\ W \end{bmatrix} \tag{2.5}$$

$$C = \begin{bmatrix} Z & Z \\ Z & Z \\ Z & Z \\ Z & Z \\ Z & Z \\ Z & Z \\ Z & W \\ Z & W \\ W & W \\ W & W \\ W & W \\ W & W \end{bmatrix} \tag{2.6}$$

The optimal solution will therefore be given by:

$$\begin{aligned} a &= (C^T C)^{-1}(C^T y) \\ &= \begin{bmatrix} 8Z^2+4W^2 & 6Z^2+2ZW+4W^2 \\ 6Z^2+2ZW+4W^2 & 6Z^2+6W^2 \end{bmatrix}^{-1} \begin{bmatrix} 6Z^2+2ZW+4W^2 \\ 6Z^2+6W^2 \end{bmatrix} \end{aligned} \tag{2.7}$$

If we choose α , β and γ in order that:

$$\begin{cases} \alpha = 8Z^2+4W^2 \\ \beta = 6Z^2+2ZW+4W^2 \\ \gamma = 6Z^2+6W^2 \end{cases} \tag{2.8}$$

We get:

$$\begin{bmatrix} a(1) \\ a(2) \end{bmatrix} = \begin{bmatrix} \frac{\gamma}{\alpha\gamma-\beta^2} & \frac{-\beta}{\alpha\gamma-\beta^2} \\ \frac{-\beta}{\alpha\gamma-\beta^2} & \frac{\alpha}{\alpha\gamma-\beta^2} \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \end{bmatrix} = \begin{bmatrix} \frac{\gamma\beta-\beta\gamma}{\alpha\gamma-\beta^2} \\ \frac{\alpha\gamma-\beta^2}{\alpha\gamma-\beta^2} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{2.9}$$

This shows that the prediction pixel should be aligned with the upper neighbor (since the first element of the prediction is the left neighbor and the second is the upper one), as we can see from figure 2.6. Experiments in [13] show that LSP predictors can adapt to any edge orientation.

2.3.2 Pixel neighborhood

In [3], a very interesting approach was taken when considering pixel neighborhood. Instead of using only causal neighbors (up and left), which would decrease quality of

prediction in the case of very distant neighbors, the closest ones are being chosen, even if they have not been decoded yet. However, for block-based prediction, some of these pixels may still need to be encoded and therefore for this case the training window needs to become causal, as is illustrated in figure 2.7: Inside a block, there is still a need to use

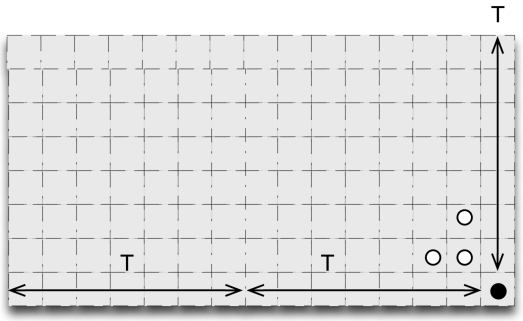


Figure 2.7: Causal training window (T = 7).

prediction values, since there is no information about the decoded pixels yet.

2.4 Summary

MMP currently provides state-of-the-art performance. However, computational complexity and encoding time have not been taken into consideration. Although it has lowered with the introduction of the MMP-FAST, there is still a wide range of techniques that can be used. One of them consists of the use of parallel processing and the power GPUs can provide.

LSP offers a way to adaptively estimate local features. This makes LSP the main prediction mode in MMP, especially for images with high frequency content (edges or 'nearly' edges). However, the computation complexity cannot be ignored. The next chapter explains how this computational burden may be softened, through the use of parallel processing.

3

Mobile parallel computing

Contents

3.1 Overview

The strict single-core era has ended [14] and parallel computing technologies are present everywhere, from mobile devices to supercomputers. Computing economics and market trends benefit low-cost general-purpose processors rather than specialized, performance focused processors. [15] [16]

Mobile Central Processing Unit (CPU)s and Graphics Processing Unit (GPU)s have found great improvements in the last years, and exploring the increasing number of cores and speed of these devices seems very opportune. Currently, 64-bit octa-core CPUs with clock frequencies up to 2,7GHz and GPUs with up to 192 pipelines and up to 950 MHz clock frequencies turn mobile computing into a quite attractive and powerful signal processing approach.

3.2 Snapdragon

All the experiments were performed under the Snapdragon processor (800 and 805). General mobile instructions were always used, which makes the software produced portable to other android devices. Snapdragon is a multiprocessor system that includes components such as a multimode modem, CPU, GPU, Digital signal processor (DSP), location/GPS, power management, Radio Frequency (RF) transceiver, memory and connectivity (Wi-Fi, Bluetooth) units.

Other interesting research studies conducted on using mobile devices for general-purpose computation can be found in [17], [18], [19], [20].

3.2.1 OpenCL architecture

The OpenCL parallel programming framework is supported on the Adreno 300, 400 series GPU, and is fully conformant to the OpenCL standard. [12] Figure 3.1 shows the opencl architecture for the snapdragon processor.

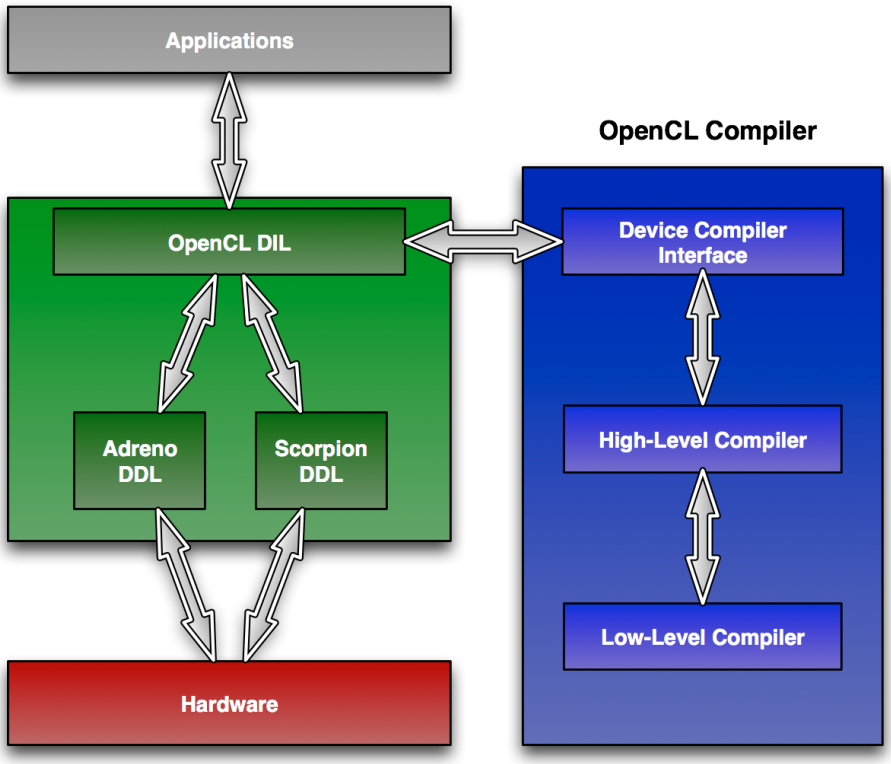


Figure 3.1: OpenCL architecture for the Adreno GPU

3.2.2 Hardware specifications

Table 3.1 Relevant specifications for the Snapdragon 805 processor

CPU	Krait 450
GPU	Adreno 420 with OpenCL 1.2 Full Profile
Cores	4
Core speed	up to 2.7 GHz
Memory	LPDDR3 800MHz Dual-channel 64-bit (25.6GBps)

3.3 OpenCL Development

The Open Computing Language (OpenCL) is a standard for parallel programming of different types of computing platforms and is designed to meet the requirements of devices with General Purpose GPUs (GPGPUs). This standard may be categorized into four models.

- Platform Model - Consists of a host device connected to one or more devices, each containing compute units, which are made of processing elements. Figure 3.2)

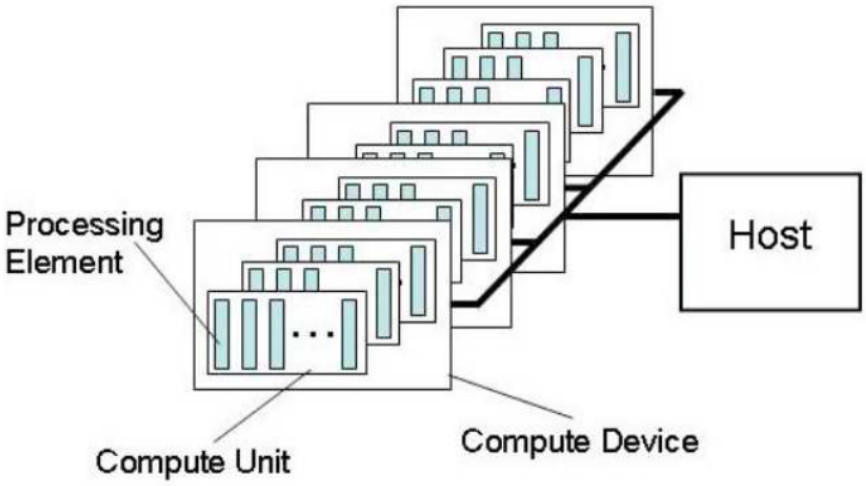


Figure 3.2: Platform model for OpenCL [1]

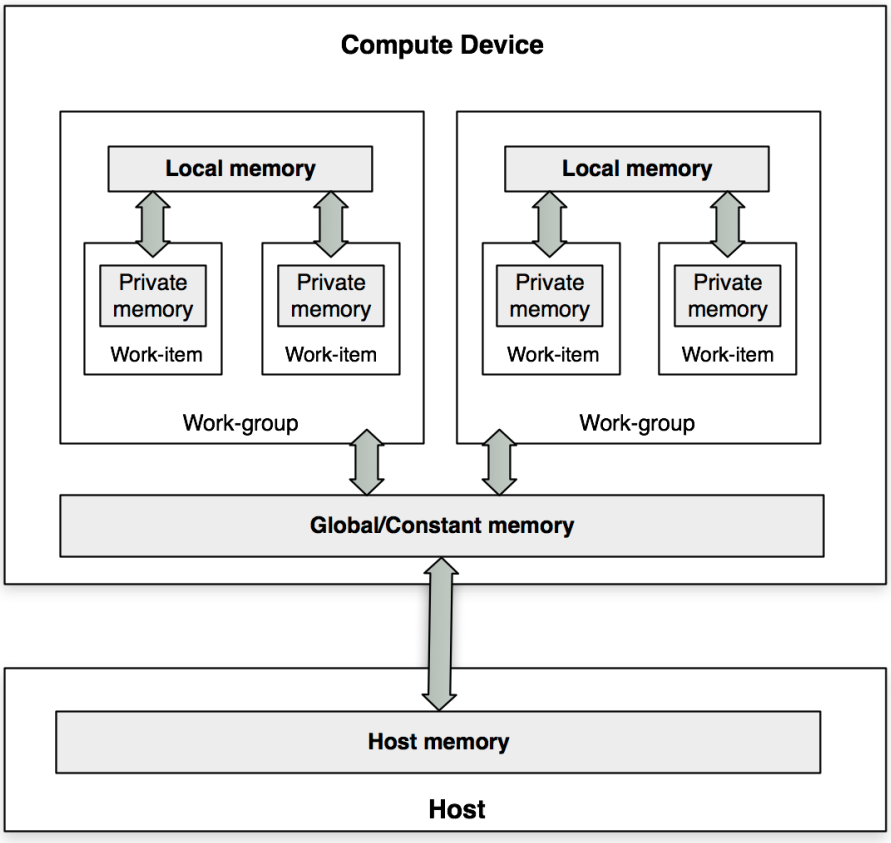


Figure 3.3: Memory regions of OpenCL

shows this. These processing elements are processed in Single Instruction, Multiple Data (SIMD) or Single program, Multiple Data (SPMD).

- Memory Model - There are four different memory regions (see figure 3.3): Global

memory is accessible to all work-items; Local memory: Accessible by all work-items inside a particular work group; Constant memory: Global memory that remains constant during kernel running; Private memory: Accessible by one particular work-item. Local memory is shorter but has greater speeds, as opposed to global memory.

- Execution Model - The execution of a kernel has a global size with 1, 2, or 3 dimensions. This global work space is composed of kernel instances, work-items, that are assigned a unique global ID. Work-items are organized in work-groups (see figure 3.4). Work-groups are also assigned a unique group ID and all work-items inside a work-group execute at the same time.

- Programming model - The execution model supports task parallel programming and data parallel programming models. In task parallel programming model, a single instance of the kernel executes on the device; In data parallel model, each work-item is responsible for one or more elements of memory.

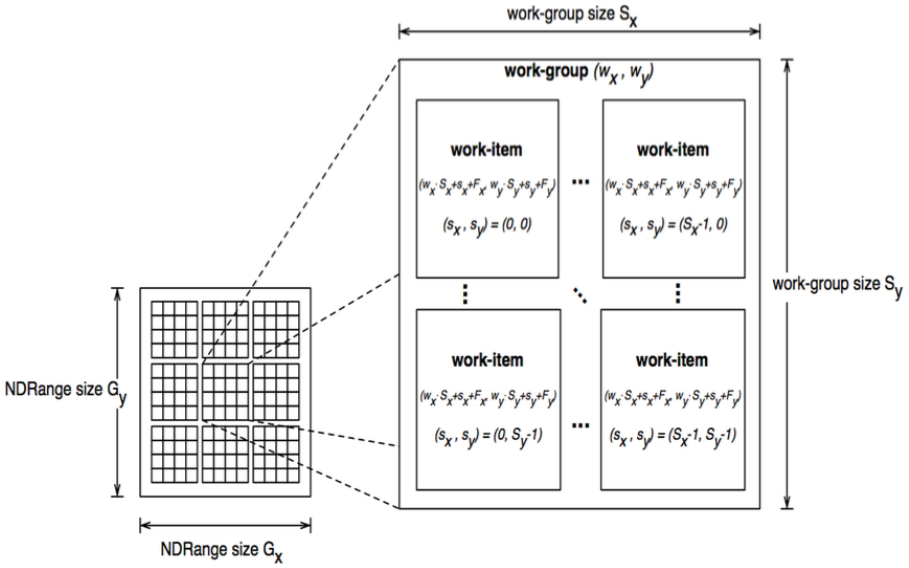


Figure 3.4: Execution model of OpenCL [1]

Programming OpenCL on mobile GPUs is a challenging procedure that has seen little relevance until recently [21] [22]. Since OpenCL is a C-based standard, one has to cross-compile through the Native Development Kit (NDK) [23].

3.3.1 Understanding Android project files

3.3.1.A jni/Application.mk

This file describes the native modules that our app requires, that being a static library, a shared library, or an executable. It defines several variables for compilation such as the platform to run the app, the modules or even the flags to be used. For example:

```
APP_ABI := armeabi-v7a
APP_PLATFORM := android-19
APP_STL := gnustl_static
APP_CPPFLAGS += -fexceptions -frtti
```

3.3.1.B jni/Android.mk

This file describes the sources and shared libraries to the build system and is useful for defining settings for the entire program. It allows us to group our source files into modules and be able to use a single source file in multiple modules, which is useful in the organization since we have source files that belong to different parts of the program. This way, we can create 2 separate modules which is useful for debuging and profiling.

3.3.2 Debugging

3.3.2.A Breakpoints

Different types of breakpoints can be used when attempting to debug a debug-enabled OpenCL driver:

- Program line
- Kernel entry
- Conditional kernel entry
- Events

Program line breakpoints require the program to be loaded into the system and are not pending. Their functionality is self-explanatory. Kernel entry breakpoints stop at the entry of a kernel. They are pending and offer the possibility to indicate which device you want to stop (CPU, GPU or both). Also, they stop on every single possible work-item, one at a time. All work-items are known at a kernel entry breakpoint, including the number of work-items per work-group. Conditional kernel entry breakpoints can receive which work-item the user wants to stop, but loses the ability to specify which device to stop. Event breakpoints act on OpenCL command events, such as CL_SUBMITTED, CL_RUNNING, CL_COMPLETE AND CL_QUEUED.

3.3.2.B GPU details

It is important to note that debugging on the Adreno GPU is very different from the Snapdragon CPU. On the GPU each work-item represents a thread and each thread appears on the debugging process (which is very useful because it enables, for example, thread switching).

Other important differences are:

- GPU kernels restrict to the maximum number of work-items on a work-group;
- All work-items suspend at the breakpoint specified;
- All work-items step or restart, if the user single-steps or continues after restart;
- On the GPU, the local information is obtained by the topmost frame of the stack;
- On the GPU, user-defined types or locals can't be read with correct types;
- On the GPU, registers are impossible to be seen;
- On the GPU, disassembly cannot be viewed;
- On the GPU, local memory values cannot be changed.

3.3.3 Profiling our program

Profiling an application is a very important step needed, for example, during debug. Knowing where the program wastes most of its time allows to make optimizations to ultimately produce better results.
The Adreno Profiler makes profiling OpenCL apps easier through the use of many performance measurement metrics (number of ALU instructions, MOVs, No OPeration (NOP)s, etc) and static analysis of OpenCL kernels.

Some of the important data that the Profiler displays are:

- Suggestions on potential performance problems;
- Information such as the number of Application Programming Interface (API) calls, redundant calls and query calls;
- The number of general purpose registers used and the number of Arithmetic logic unit (ALU) operations executed.

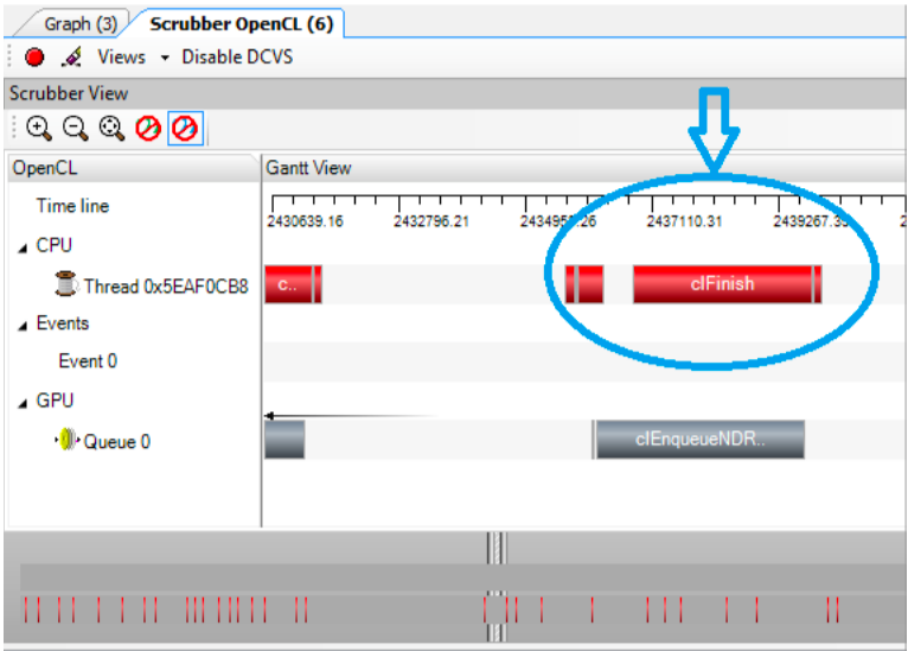


Figure 3.5: Debugging times on Adreno Profiler [2]

3.4 Summary

This chapter presents an overview on the state-of-the-art mobile architectures for parallel computing and proposes one particular processor, the Snapdragon. The architecture was presented for this processor and guidelines for using the OpenCL framework were shown in order to produce acceptable results. The next chapter shows the developed approach to the LSP through the use of these guidelines. The debugging and profiling processes contributed a lot for the results shown. Such good development practises should be followed by a devoted programmer tackling the world of parallel mobile programming.

4

LSP parallelization techniques

Contents

4.1 Overview

Since LSP has the ability to detect local characteristics derived on the fly, it is considered the dominant prediction mode in MMP [1]. However, its computational complexity is considerable, since it performs a matrix inversion for each pixel. In addition to this, the constant communication between host (CPU) and device (GPU) makes the paralellization process more challenging.

In sequential running, the LSP is responsible for 25 to 40% of the total MMP process, depending on the image to be compressed (Tables 5.2, 5.4 and 5.6). It is by far the most time consuming process of the entire MMP compression. New and more efficient approaches, considering execution time (and energy consumption), are possible. In fact, the recent advent of mobile parallel processors has opened new paths to explore these possibilities and run heavy computational programs in a more efficient way. They provide high processing capabilities and also stand for their energy efficiency, when compared to performance-focused high-end processors.

An important decision to make before the paralellization was to choose if the prediction should be made on a per-pixel or per-block basis. If we choose to apply prediction on a per-pixel basis, all the prediction pixel values wait for the updated neighbours. On the other hand, if we make a per-block prediction, all pixels inside that block will make their predictions simultaneously, without waiting for each others. In this case, the original values would be used for calculations.
Since the LSP is mainly used in lossless or almost lossless compression, the per-pixel prediction was chosen, since it will inevitably produce better results.

4.2 A parallel approach

To start the paralellization process, the work to be performed is divided by the number of available GPU threads (see figure 4.1). Since the LSP is block-based and the block size changes continuously, we chose to make the number of GPU threads dependent on the block size, i.e., dynamic. The block sizes vary from 4x4 to 16x16 and we attribute each pixel to a number of threads (work-items) and have their values shared through local memory. The number of threads per pixel is different in each device tested, being 4 work-items/pixel on the Snapdragon processor and MacbookAir and 32 work-items/pixel on the high-end GPU. On the snapdragon processor 4 work-items/pixel is the maximum possible number since each work-group has a limit of 1024 (for 16x16 block size, 4 work-

items/pixel gives 1024). For a given pixel, its assigned threads concurrently execute the prediction, which has a lot of work that can be run independently, like iterations of the loop responsible for each LU decomposition (recall that a matrix inversion is performed for each pixel).

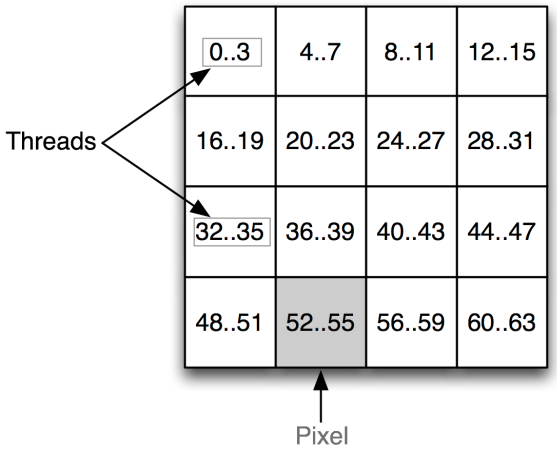


Figure 4.1: Example of a 4x4 block with 4 threads per pixel.

The second procedure consists of optimizing local memory usage, making use of it whenever it is available. Since a lot of this memory was already being used by the other kernels in the MMP, we had to make consistent reductions of local memory usage until the LSP reached the maximum possible storage occupation. Variables allocated in local memory are shared by all work-items inside a work-group and exist only for the lifetime of the work-group executing the kernel. The fact that the LSP needs a lot of dedicated memory makes our work more challenging, since we have no option but to allocate the majority of our buffers in slower, global memory. Local memory has significantly lower access times and was thus exploited for maximum performance.

As we are constantly transfering data, we synchronize the prediction block between host and device (figure 4.2). This maps the prediction block from the device buffer into the host address space and thus everytime data changes in that block, the host updates accordingly. In addition to this, the image also needs to be syncronized from host to device. Imagine we are predicting the upper row of pixels in a block. Now those pixels will need the ones above to correctly calculate the prediction. As we can see, the device running our LSP kernel needs to have access to all the image content, otherwise our approach would impose big losses to the final image. As stated above, we use a per-pixel prediction, which means that there also needs to be a syncronization process between pixels. This is done using a work-group barrier. All the work-items of a work-group (in our case, the block containing the pixels) must execute the barrier before any are allowed to continue execution beyond the barrier.

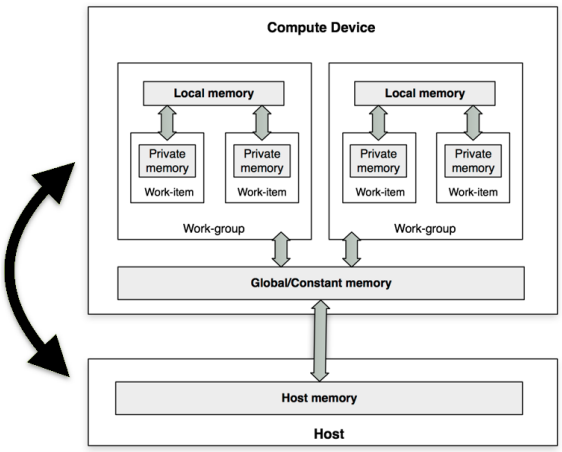


Figure 4.2: Syncing between host and device.

A minor performance improvement but still worth mentioning was obtained when the prediction block and the remaining kernel buffers were transformed from bidimensional into uni-dimensional (figure 4.3). Since these buffers are constantly being accessed, this decision reduced the LSP time even further.

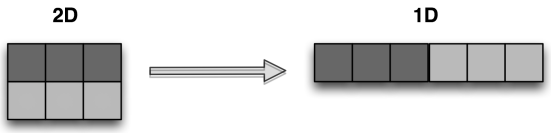


Figure 4.3: Transforming bidimensional buffers into uni-dimensional.

4.2.1 Proposition 1 - Reducing prediction order

In [3] it is claimed that the LSP depends highly on the training process, and studies have shown that the training window has an optimized value of 7. However, results from that source show that reducing the prediction order doesn't have a considerable performance loss on the final result. A proposed change to the suggested parameters is then reducing the prediction order from 10 to 2 in order to reduce computational complexity even further, while keeping the training window with T = 7. In other words, making each prediction pixel to be a weighted average of just 2 neighbouring pixels (usually the left and up), while keeping each of these 2 pixels to be 'trained' by a double-rectangular window of size 7.

With this tradeoff proposal we aim to acheive processing time reductions in exchange of acceptable performance quality.

Figure 4.4: Proposed prediction order.

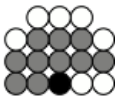


Figure 4.5: Original prediction order.

4.2.2 Proposition 2 - Conditional prediction

Since the LSP is great for areas of high frequency content (i.e., 'edges' or 'near-edges'), a very plausible way of reducing the LSP time consumption without compromising the overall performance of the image compression would be to apply the LSP only for pixels that differ significantly from their neighbors. In this scenario, a second order analysis is performed. For each pixel, we define an offset (basically a difference between two pixel intensity values). If a pixel has intensity equal or greater than the offset, the LSP is applied, otherwise, basic intra-prediction takes place. Left and upper pixels were chosen as neighbourhood since this is the usual convention among image scanning order approaches.

In the following figure we can see the pixels where the LSP is applied, for the Lena image particular case.

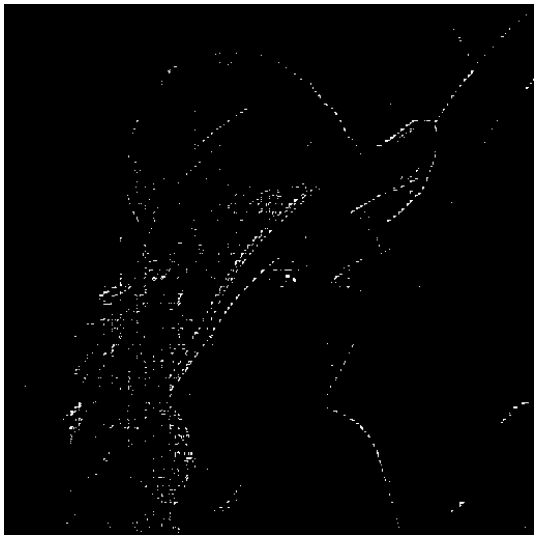


Figure 4.6: The white pixels represent the chosen pixels for LSP prediction. They correspond to edges or near-edges. The offset chosen was 20, i.e., the LSP was applied for pixels that differ in intensity by 20 or more from their 2 neighbors (up and left).

4.3 Summary

In this chapter we address the paralellization techniques applied to the LSP in order to reduce the time burden. Some represent general methods, such as work-items partitioning, while others are platform dependent. In addition to this, we proposed two new approaches to the LSP, aiming to reduce even further the time and energy spent.

5

Experimental results

Contents

5.1 Introduction

5.2 Apparatus

The experimental results were tested under 3 platforms, and the relevant information about each platform is shown below:

Table 5.1 Details of the tested platforms

	Platform A	**Platform B**	**Platform C**
CPU	i7-4790K	i5-4260U	Krait 450
GPU	Tesla K40c	Intel HD Graphics 5000	Adreno 420
OS	Ubuntu 14.04	OSX Yosemite 10.10.4	Android-19
Compiler	gcc 4.8.4	clang-602.0.49	clang
Language	C and OpenCL 1.2		

5.3 Original LSP on multicores

The following experimental results correspond to the Lena image (figure 5.1a), with low distortion. The final decoded image is shown in figure 5.1b. In figures 5.1c and 5.1d we can observe the intra prediction resulting image and the intra prediction error, respectively. The same tests are then shown for smaller images.

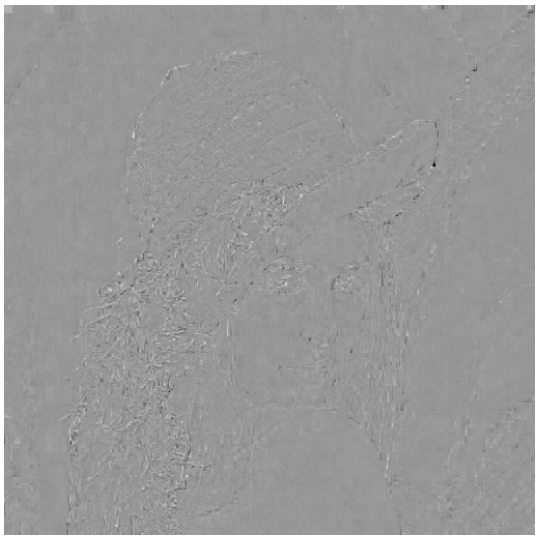
The Peak Signal-to-Noise Ratio (PSNR) of the resulting images and of prediction images are:

- Lena.pgm - PSNR: 39,647672 / PSNR PREDICTION: 30,876516
- Small.pgm - PSNR: 42,361934 / PSNR PREDICTION: 32,052278
- Xsmall.pgm - PSNR: 40,595199 / PSNR PREDICTION: 24,309624

(a) Original image

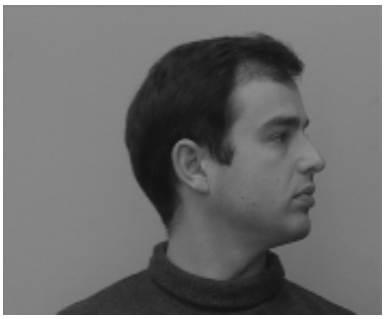
(b) Resulting image

(c) Intra prediction resulting image



(d) Intra prediction error

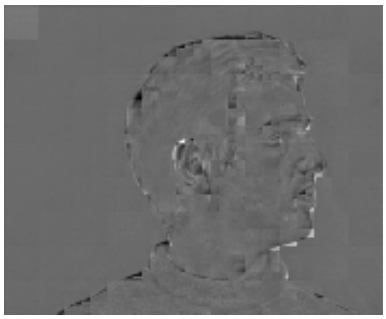
Figure 5.1: Lena.pgm (512 x 512)



(a) Original image

(b) Resulting image

(c) Intra prediction resulting image



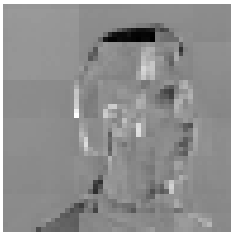
(d) Intra prediction error

Figure 5.2: Small.pgm (176 x 144)

(a) Original image

(b) Resulting image

(c) Intra prediction resulting image



(d) Intra prediction error

Figure 5.3: Xsmall.pgm (48 x 48)

5.3.1 Platform A

Average power consumption: 186 W (for both sequential and parallel implementations).

Table 5.2 Original LSP: Sequential version, platform A

Image	Energy (Joules)	MMP time (seconds)	LSP time (seconds)
Lena	40928	220,0	108,4
Small	2959	15,9	8,6
Xsmall	228	1,2	0,5

Table 5.3 Original LSP: Parallel version, platform A

Image	Energy (Joules)	MMP time (seconds)	LSP time (seconds)
Lena	30962	166,5	15,2
Small	1941	10,4	2,1
Xsmall	177	1,0	0,2

5.3.2 Platform B

Average power consumption: 180 W for the sequential and 190 W for the parallel implementation.

Table 5.4 Original LSP: Sequential version, platform B

Image	Energy (Joules)	MMP time (seconds)	LSP time (seconds)
Lena	124291	690,5	188,9
Small	10670	59,3	15,6
Xsmall	882	4,9	1,0

Table 5.5 Original LSP: Parallel version, platform B

Image	Energy (Joules)	MMP time (seconds)	LSP time (seconds)
Lena	148588	782,0	183,6
Small	12950	68,2	15,5
Xsmall	1113	5,9	1,2

5.3.3 Platform C

Average power consumption: 3,3 W (for both sequential and parallel implementations).

Table 5.6 Original LSP: Sequential version, platform C

Image	Energy (Joules)	MMP time (seconds)	LSP time (seconds)
Lena	5483	1661,6	712,2
Small	361	109,3	11,7
Xsmall	41	12,3	1,2

Table 5.7 Original LSP: Parallel version, platform C

Image	Energy (Joules)	MMP time (seconds)	LSP time (seconds)
Lena	3543	1073,5	168,3
Small	305	92,5	1,3
Xsmall	40	12,0	0,7

We can see that for platform A, we obtain a speedup of up to 1.53 for the MMP and a speedup of up to 7.15 for the LSP. The power consumption of this device didn't change from the sequential to the parallel implementations. Platform B is the only one that performs worse for the parallel verion of the MMP, although the LSP time becomes slightly shortened. For this scenario, the energy consumption is also higher for the parallel version, as opposed to the other platforms. Platform C performs similarly to platform A. Gains of 1,55 for the MMP (Lena image) and 8,19 for the LSP (Small image) were achieved. Comparisons of time and energy for both the sequential and the parallel approaches can be seen in the following figures.

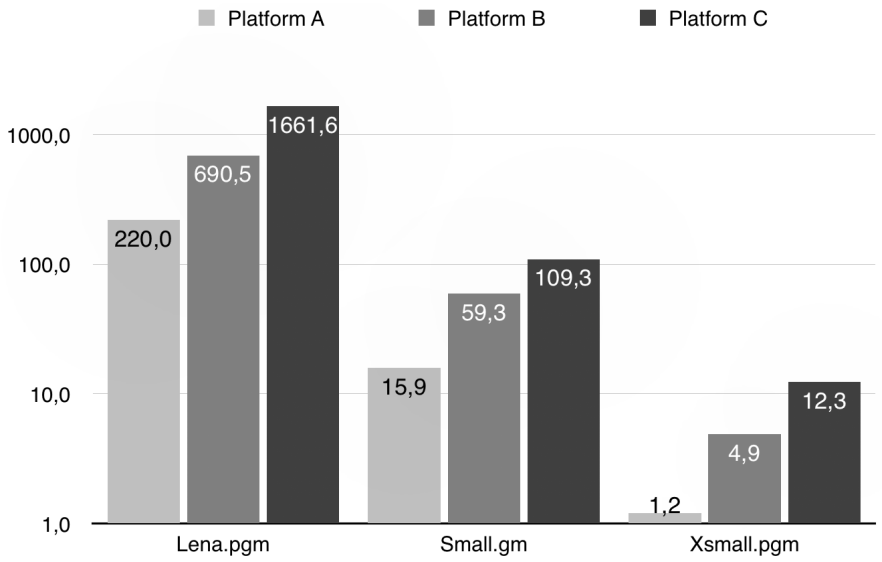


Figure 5.4: Times for the sequential version.

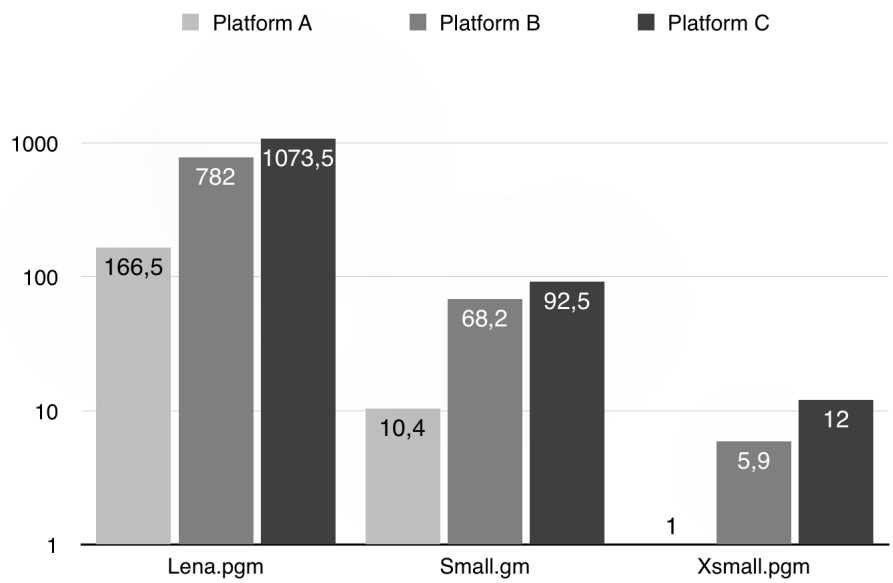


Figure 5.5: Times for the parallel version.

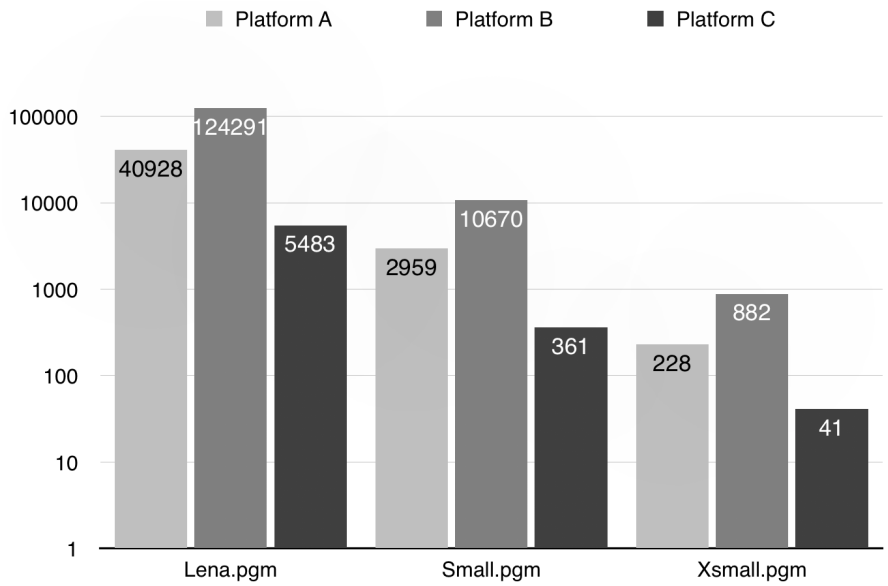


Figure 5.6: Energy for the sequential version.

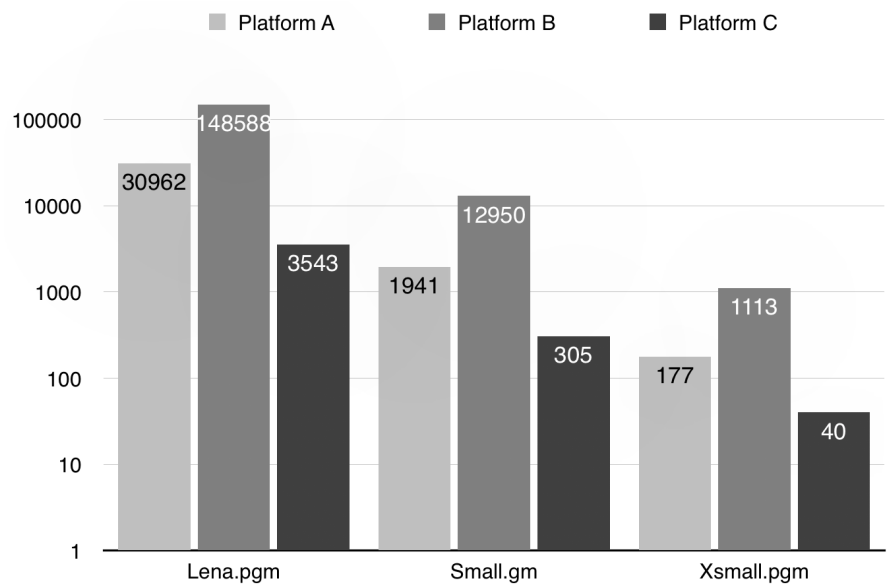


Figure 5.7: Energy for the parallel version.

5.4 Proposed LSP (Reducing prediction order) on multi-cores

The PSNR of the resulting images and of prediction images are:

- Lena.pgm - PSNR: 39,590109 / PSNR PREDICTION: 29,65527
- Small.pgm - PSNR: 42,437122 / PSNR PREDICTION: 31,425513
- Xsmall.pgm - PSNR: 40,771614 / PSNR PREDICTION: 24,324429

5.4.1 Platform A

Table 5.8 Proposed LSP 1: Sequential version, platform A

Image	Energy (Joules)	MMP time (seconds)	LSP time (seconds)
Lena	30921	166,2	21,3
Small	1745	9,4	1,8
Xsmall	176	1,0	0,1

Table 5.9 Proposed LSP 1: Parallel version, platform A

Image	Energy (Joules)	MMP time (seconds)	LSP time (seconds)
Lena	30866	165,9	15,2
Small	1775	9,5	1,3
Xsmall	184	1,0	0,1

5.4.2 Platform B

Table 5.10 Proposed LSP 1: Sequential version, platform B

Image	Energy (Joules)	MMP time (seconds)	LSP time (seconds)
Lena	111121	617,3	74,5
Small	9221	51,2	6,7
Xsmall	805	4,5	0,5

Table 5.11 Proposed LSP 1: Parallel version, platform B

Image	Energy (Joules)	MMP time (seconds)	LSP time (seconds)
Lena	141681	745,7	173,4
Small	12660	66,6	15,6
Xsmall	1107	5,8	1,3

5.4.3 Platform C

Table 5.12 Proposed LSP 1: Sequential version, platform C

Image	Energy (Joules)	MMP time (seconds)	LSP time (seconds)
Lena	3569	1081,4	206,8
Small	311	94,3	9,6
Xsmall	35	10,5	0,3

Table 5.13 Proposed LSP 1: Parallel version, platform C

Image	Energy (Joules)	MMP time (seconds)	LSP time (seconds)
Lena	3552	1076,4	165,1
Small	323	98,0	13,3
Xsmall	41	12,5	1,0

This sequential approach shows gains of 1.54 for MMP and 5.1 for LSP in comparison with the original sequential approach. This change does not alter the power consumption

of any device, thus reducing the energy consumption. Parallelization does not generally bring speedups or energy reductions.

Comparisons of time and energy for both the sequential and the parallel approaches can be seen in the following figures.

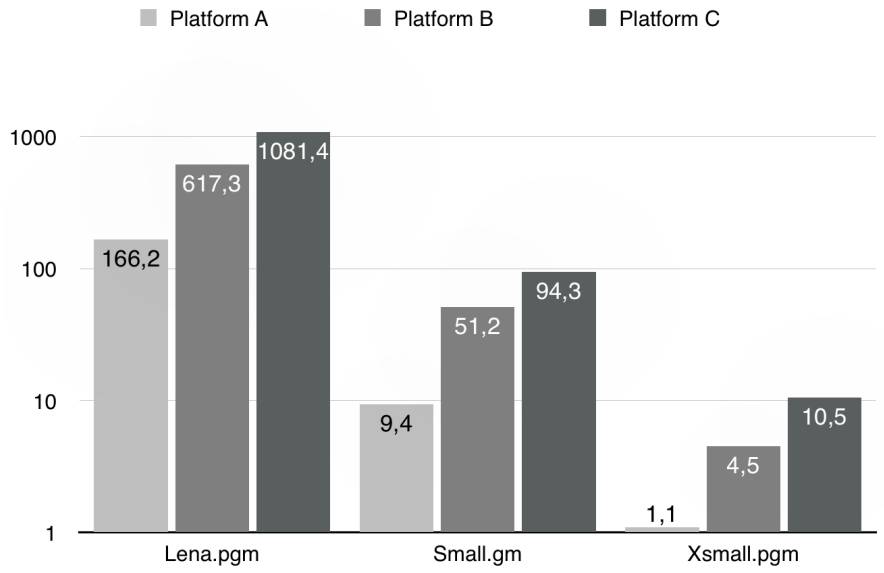


Figure 5.8: Times for the sequential version.

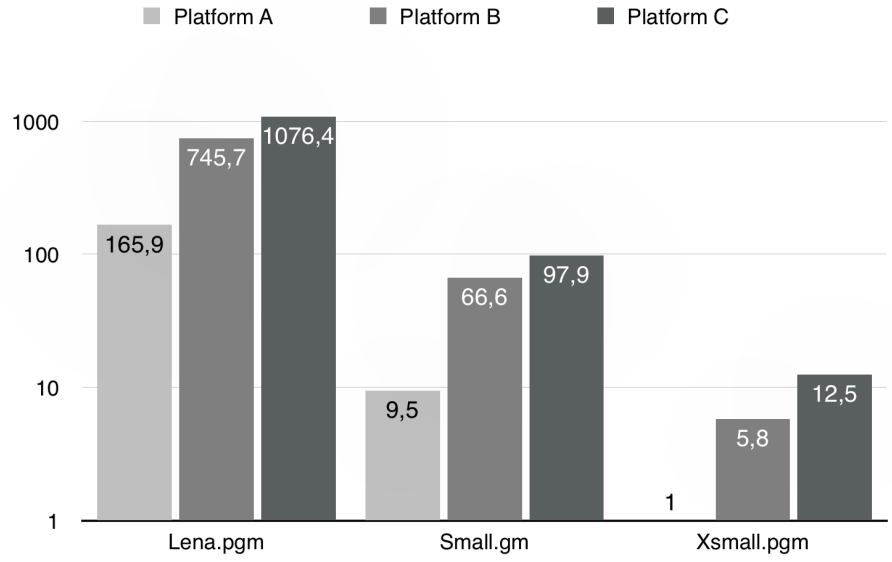


Figure 5.9: Times for the parallel version.

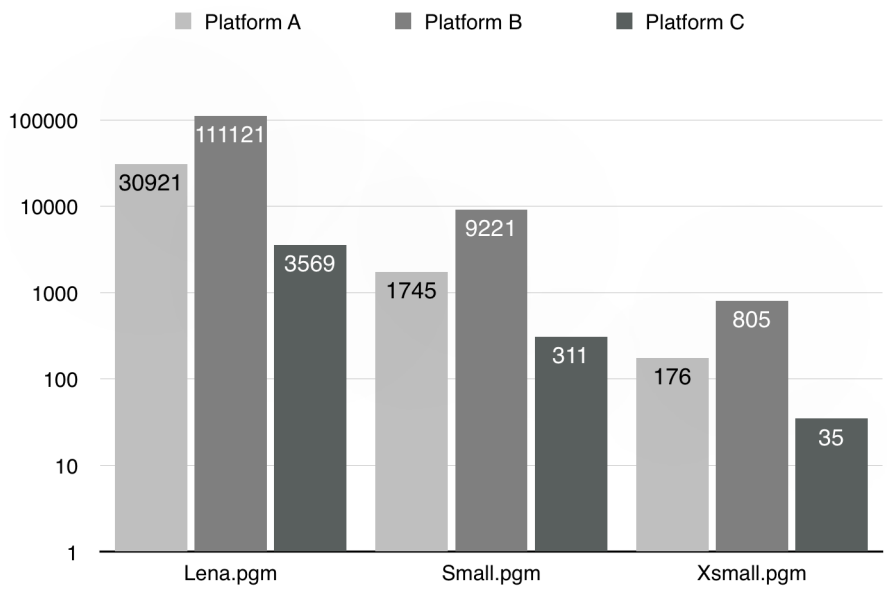


Figure 5.10: Energy for the sequential version.

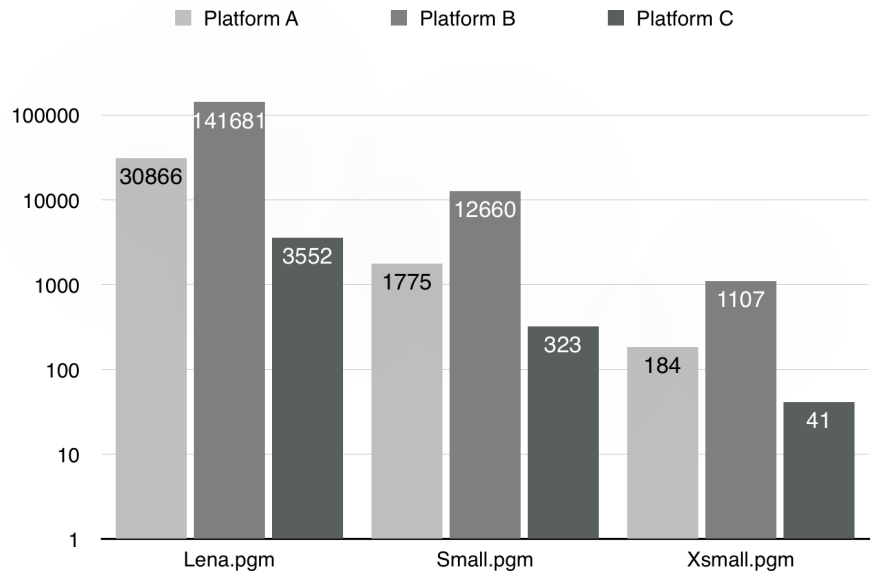


Figure 5.11: Energy for the parallel version.

5.5 Another LSP proposal (Conditional prediction) on multicores

This data resulted from our second LSP proposal for platform A, with a 10th order prediction.

The peak signal-to-noise ratios (PSNR) of the resulting images and of prediction images are:

- Lena.pgm - PSNR: 39,538339 / PSNR PREDICTION: 29,438908

- Small.pgm - PSNR: 42,438508 / PSNR PREDICTION: 31,221927
- Xsmall.pgm - PSNR: 40,771614 / PSNR PREDICTION: 24,324429

Table 5.14 Proposed LSP 2: Sequential version, platform A

Image	Energy (Joules)	MMP time (seconds)	LSP time (seconds)
Lena	27304	146,8	5,7
Small	1467	7,9	0,5
Xsmall	159	0,9	0,1

This approach reveals gains of 2.02 for MMP and 18.88 for LSP in comparison with the original sequential approach. This change does not alter the power consumption of any device. We can see a great decrease in energy consumption as well. In fact, for the Small image, the energy consumed is halved. Parallelization of this approach does not generally bring speedups.

5.6 Discussion

The first thing to point out is the time reduction from the parallel adaptation. Gains of up to 7.15 times were obtained for the LSP, resulting in a 1.32 time reduction of the MMP, with no performance loss. Also the energy dissipated was considerably lower for the parallel processing scenario. Take notice that the power consumption of each device does not represent the tested program alone, but also the idle power consumption needed for that device to function properly. The snapdragon processor, while nearly 7x slower when compared to platform A, is 5 to 8 times more energy efficient. Interesting results regarding the energy efficiency of the snapdragon GPU may also be found in [24].

Figure 5.12 ilustrates this interaction between energy efficiency and time of computation, for platforms A and C (very powerful GPU and mobile GPU, respectively). Platform B, which corresponds to the Intel HD Graphics 5000 GPU (Macbook Air) was left aside since it takes no purpose in running with good performance GPU heavy applications. Experimental results from this Intel GPU show disadvantes in using our parallel solution both in time and energy spent, since the difference in performance from the Macbook Air CPU and GPU is not significant, as opposed to the other platforms.

Figure 5.13 compares prediction orders of the LSP, where we take into consideration computation time of the MMP and LSP, for the Lena image particular case, executed in platform A. The correspondent performance loss of the entire MMP process and of the prediction alone is illustrated in figure 5.14.

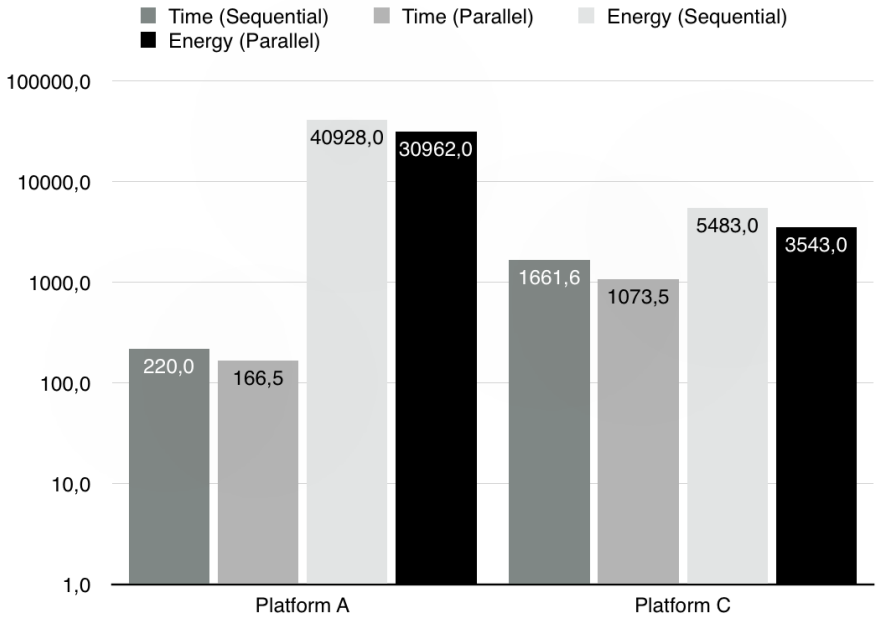


Figure 5.12: Tradeoff between energy and time

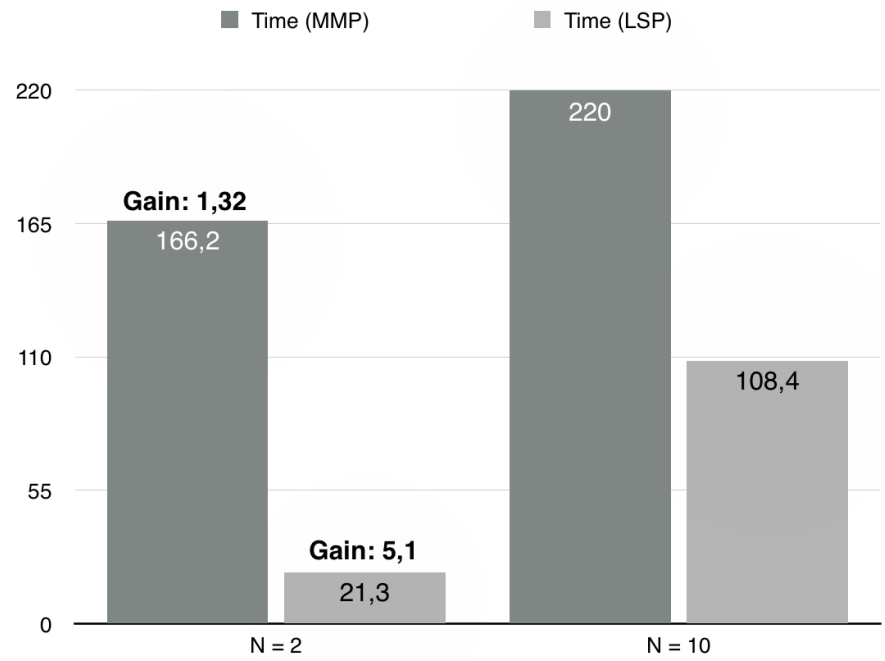


Figure 5.13: Tradeoff between prediction orders.

On a side note, one may notice that for this prediction order reduction, there is no time reduction from the parallel adaptation. This can be explained by the fact that the LSP becomes less computational heavy and thus the use of a GPU kernel is not so appropriate.

Furthermore, making the LSP apply only to areas of the image with high frequency content provides great acceleration, with little performance loss, as figures 5.15 and 5.16

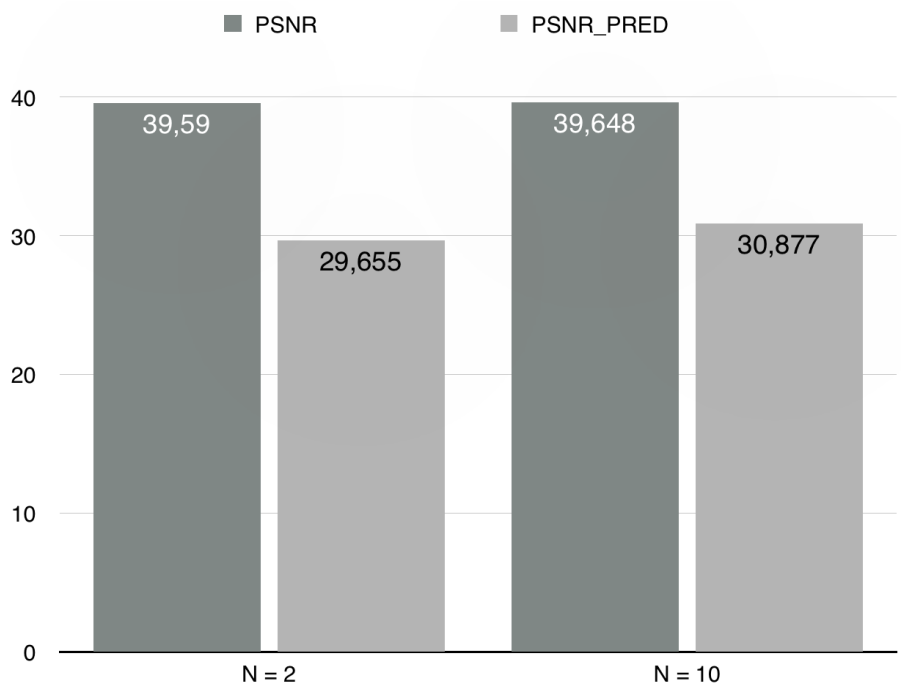


Figure 5.14: Tradeoff between prediction orders.

show, respectively.

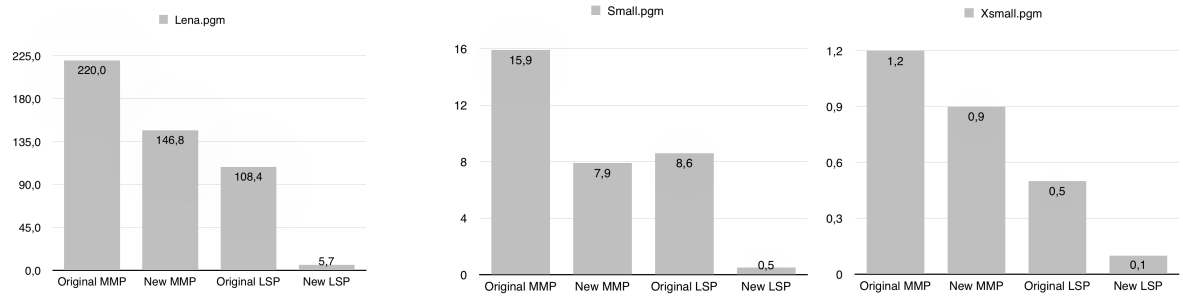


Figure 5.15: Time comparison between the original LSP and the conditional prediction proposition for the three images.

5.7 Summary

In this chapter, we compare time and energy of the MMP codec and the LSP algorithm on different platforms, for the sequential and parallel approaches. Images with different

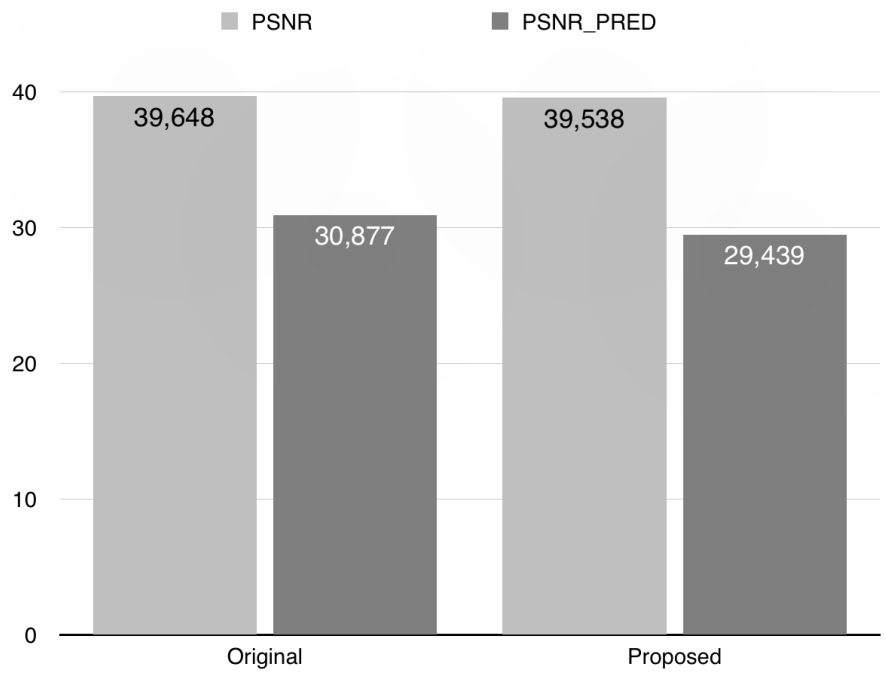


Figure 5.16: Performance loss for the Lena image particular case, executed under platform A for the sequential version.

sizes were used. In addition, we discuss and illustrate the energy-time paradigm. In the next chapter we make conclusions with the results obtained and propose future work.

6

Conclusions

Contents

The use of GPU computing benefits the LSP and, consequently, the MMP. Although there is a lot of communication between CPU and GPU, the LSP is sufficiently complex to benefit from this GPU parallel approach. This is not the case for the algorithmic propositions presented in this thesis, since the computational complexity is significantly reduced for these approaches.

The snapdragon processor, while much slower than the high-end tested GPU, offers great results in energy efficiency, which is very convenient, since this is a mobile processor and mobile energy sustainability is a necessity and currently a very active field of research [25] [26]. Furthermore, it is faster than the desktop GPU for the majority of the experiments using a parallel approach.

The proposed prediction order reduction turns out to be an interesting tradeoff between time reduction and performance loss. Since the LSP is mainly used in lossless or near-lossless image coding, we propose the possibility to adapt this prediction order according to the quality of compression (i.e., use a lower prediction order for higher acceptable distortions).

The conditional prediction proposition, too, presents interesting results. The fact that we choose the offset needed to activate the use of the predictor makes this approach adaptive. And since the MMP is a context-based codec, this proposition seems very opportune. We propose this implementation, or others based on this concept, for this type of predictors.

6.1 Future work

The results obtained are very positive and encourage to continue to investigate this subject. Although good performance has been observed, there are some aspects that can be further studied, namely:

- Improve parallelization techniques to further reduce the LSP computational time, such as creating image objects instead of regular C arrays to store the data transferred between host and device, which is possible using this optimized OpenCL data type;
- Making new algorithmic approaches to both the LSP and MMP, since the computational complexity is much higher than that of competitors;
- Testing the developed kernels in other platforms, such as Field-Programmable Gate Array (FPGA)s, since the energy consuption in low-power processor showed significant results when compared to high-power ones;

Bibliography

[1] *The OpenCL Specification*. Khronos Group.

[2] *Snapdragon OpenCL General Programming and Optimization*. Qualcomm Technologies, August 2014.

[3] D. B. Graziosi, "Contributions to lossy and lossless image compression using multiscale recurrent pattern matching," Ph.D. dissertation, Federal University of Rio de Janeiro, Rio de Janeiro, Brazil, 2011.

[4] "Draft of Version 4 of H.264/AVC (ITU-T Recommendation H.264 and ISO/IEC 14496-10 (MPEG-4 part 10) Advanced Video Coding)," March 2001.

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

[6] D. S. Taubman and M. Marcelin, *JPEG2000: Image Compression Funda- mentals, Standards and Practice*, 2nd ed. Norwell, Massachusetts: Kluwer Academic Publishers, 2001.

[7] N. M. M. Rodrigues, "Multiscale recurrent pattern matching algorithms for image and video coding," Ph.D. dissertation, University of Coimbra, Coimbra, Portugal, 2008.

[8] N. Rodrigues, E. Silva, M. Carvalho, S. Faria, and V. Silva, "On dictionary adaptation for recurrent pattern image coding," *IEEE Transactions on Image Processing*, vol. 17, no. 9, p. 1640–1653, September 2008.

[9] ——, "Universal image coding using multiscale recurrent patterns and prediction," *Proceedings of the IEEE International Conference on Image Processing, ICIP '05*, vol. 2, p. II–245–8, September 2005.

[10] "Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification (ITU-T Rec. H.264 — ISO/IEC 14496-10 AVC))," July 2004.

[11] X. LI, "Edge directed statistical inference and its application to image processing," Ph.D. dissertation, Princeton University Graduate School, Princeton, New Jersey, USA, 2000.

[12] S. L. Campbell and C. D. Meyer, *Generalized Inverses of Linear Transformations*, 1st ed. Philadelphia, PA: Pitman Publishing, 2008.

[13] X. Li and M. T. Orchard, "Edge-Directed Prediction for Lossless Compression of Natural Images," *IEEE Transactions on Image Processing*, vol. 10, no. 6, pp. 813–817, 2001.

[14] K. Olukotun and L. Hammond, "The Future of Microprocessors," vol. 54, no. 5, pp. 26–29, May 2011.

[15] R. Vuduc and K. Czechowski, "What GPU computing means for high- end systems," vol. 31, no. 4, pp. 74–78, 2011.

[16] G. Bell, "Bell's law for the birth and death of computer classes," vol. 51, no. 1, pp. 86–94, january 2008.

[17] I. K. P. N. Singhal and S. Cho, "Implementation and optimization of image processing algorithms on handheld GPU," *Image Processing (ICIP), 2010 17th IEEE International Conference*, p. 4481–4484, September 2010.

[18] K. T. Cheng and Y. C. Wang, "Using mobile GPU for general-purpose computing - a case study of face recognition on smartphones," *VLSI Design, Automation and Test (VLSI-DAT), 2011 International Symposium*, pp. 1–4, April 2011.

[19] M. B. Lopez, H. Nykanen, J. Hannuksela, O. Silven, and M. Vehvilainen, "Accelerating image recognition on mobile devices using GPGPU," p. 78 720R–78 720R–10, 2011.

[20] M. W. B. Rister, G. Wang and J. Cavallaro, "A fast and efficient sift detector using the mobile GPU," *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference*, p. 2674–2678, May 2013.

[21] (2013, january) OpenCL reveals Kindle Fire's true potential. [Online]. Available: http://gearburn.com/2013/01/opencl-reveals-kindle-fires-true-potential/

[22] J. Y. G. Wang, Y. Xiong and J. Cavallaro, "Accelerating computer vision algorithms using OpenCL framework on the mobile GPU - a case study," *Acoustics, Speech and Signal Processing (ICASSP)*, pp. 2629–2633, May 2013.

[23] [Online]. Available: https://developer.android.com/ndk

[24] J. A. Ross, D. A. Richie, S. J. Park, D. R. Shires, and L. L. Pollock, "A Case Study of OpenCL on an Android Mobile GPU," *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*, 2014.

[25] T. Coughlin, "A Moore's Law for Mobile Energy: Improving upon conventional batteries and energy sources for mobile devices," *Consumer Electronics Magazine, IEEE*, pp. 74 – 82, January 2015.

[26] F. H. P. F. P. Perrucci and J. Widmer, "Survey on energy consumption entities on the smartphone platform," *Proc. IEEE 2011 73rd Vehicular Technology Conf. (VTC Spring)*.

Appendix A

IMPROVING LOW-POWER PERFORMANCE OF LEAST SQUARES PREDICTION FOR IMAGE CODING USING MOBILE PARALLEL PROCESSING

Pedro Cordeiro, Gabriel Falcao

Instituto de Telecomunicações
Dept. Electr. and Comp. Eng.
University of Coimbra

Patricio Domingues, Nuno N.M. Rodrigues, Sergio M. M. Faria

Escola Superior de Tecnologia e Gestão de Leiria
Instituto Politécnico de Leiria, Portugal
Instituto de Telecomunicações, Portugal

ABSTRACT

Least squares prediction is a technique used for foreseeing pixel values during image coding by finding the minimum square error of neighboring pixels. It has shown considerable quality gains especially for complex images with high variations in pixel intensities (i.e., edges). The drawback of this technique consists of high computational complexity, which makes it difficult to implement in fast, lossy image coders. One challenge is therefore to reduce the computational time of this predictor through the use of some parallel techniques, making it more attractive for state-of-the-art Coder-Decoders (CODECs). Also, a couple of algorithmic propositions were made, trying to reduce the time spent in exchange for rate-distortion performance. These propositions are senseful since this predictor is used not only in lossless image coding, but in lossy as well. Another aim of this article is to analyze energy efficiency among different types of platforms for this signal processing algorithm. Comparisons are provided on parallel computing processors ranging from very powerful Graphics Computing Units (GPUs) to mobile General-Purpose GPUs.

***Index Terms*—** Least squares prediction, Image prediction, Image processing, Lossy image coding, Parallel processing, Graphics Processing Unit (GPU) processing, Energy efficiency, Manycore platforms

1. INTRODUCTION

Prediction is an important process to reduce the amount of information stored in image coding. There are a lot of different prediction modes, each appropriate to a certain set of image characteristics. For smooth images, low complexity predictors may do the job. But for complex, compound images, these predictors may not suffice. For hard images (with sudden variations in pixel intensities, i.e., edges), there is a particular predictor that tends to increase image quality: the Least Squares Prediction (LSP). The LSP is especially good at predicting edges, but even for text and compound images, it does not present rate-distortion losses. In this article, the LSP implementation conducted in [1] will be adopted as the basis of this work. These predictors are known for having a considerable impact in time consumption and computational complexity and therefore the use in manycore platforms seems completely justified. Parallel computing is gaining an increased relevance, with new processors becoming more and more powerful. Signal processing algorithms are generally time constrained and can benefit from the acceleration that parallel computation offers. Signal processing images are especially vulnerable to delays. Mobile parallel processing platforms, too, are evolving at a significant rate in terms of the number and processing capabilities of the cores. They may not have the processing power of modern desktops, but are cheaper and allow achieving higher levels of energy efficiency, since desktop GPUs still demand high levels of power.

This work has been supported by UID/EEA/50008/2013

1.1. Motivation

The aim is to explore the possibilities of parallel computing on image codecs and particularly high complexity predictors. A comparison between mobile, desktop and super powerful GPUs is provided. An additional motivation is the evolution of energy consumption levels of recent devices. Thus, an analysis on this metric between different platforms is timely and essential for future studies.

2. MOBILE PARALLEL COMPUTING

2.1. Overview

Computing economics and market trends benefit low-cost general-purpose processors rather than specialized, performance focused processors [2] [3]. Mobile CPUs and GPUs have found great improvements in the last years, and exploring the increasing number of cores and speed of these devices seems very opportune.

2.2. Snapdragon

All the experiments were performed under the Snapdragon processor (800 and 805) [4]. Snapdragon is a multiprocessor system that includes components such as a multimode modem, CPU, GPU, Digital signal processor (DSP), location/GPS, power management, Radio Frequency (RF) transceiver, memory and connectivity (Wi-Fi, Bluetooth) units.

2.3. OpenCL Development

2.3.1. Overview

The OpenCL is a standard for parallel programming of different types of computing platforms and is designed to meet the requirements of devices with General Purpose GPUs (GPGPUs), offering a quick way to exploit parallelism in signal processing algorithms [5]. This standard may be categorized into four models:

- Platform Model - Consists of a host device connected to one or more devices, each containing compute units, which are made of processing elements.

- Memory Model - There are four different memory regions: Global memory is accessible to all work-items; Local memory: Accessible by all work-items in a work group; Constant memory: Constant global memory; Private memory: Accessible by one work-item.
- Execution Model - The execution of a kernel has a global size with 1, 2, or 3 dimensions. Work-items are organized in work-groups. All work-items inside a work-group can potentially execute at the same time.
- Programming model - The execution model supports task parallel programming and data parallel programming models.

Programming OpenCL on mobile GPUs is a challenging procedure that has seen little relevance until recently [6] [7]. Since OpenCL is a C-based standard, one has to cross-compile through the Native Development Kit (NDK) [8]. This means the adaptation of one's project settings to reach compatibility, namely the adaptation of the build system and the description of the native modules needed by the program.

2.3.2. Debugging

GPU details

It is important to note that debugging on the Snapdragon GPU (Adreno) is very different from the Snapdragon CPU. On the GPU each work-item represents a thread and each thread appears on the debugging process (which is very useful because it enables, for example, thread switching).

Other important differences are:

- GPU kernels are restricted to the maximum number of work-items on a work-group;
- All work-items suspend at the breakpoint specified;
- All work-items step or restart, if the user single-steps or continues after restart;
- On the GPU, the local information is obtained by the topmost frame of the stack;
- On the GPU, locals of user-defined types cannot be examined with correct type information;
- On the GPU, registers are impossible to be seen;
- On the GPU, disassembly cannot be viewed;
- On the GPU, local memory values cannot be changed.

3. LEAST SQUARES PREDICTION

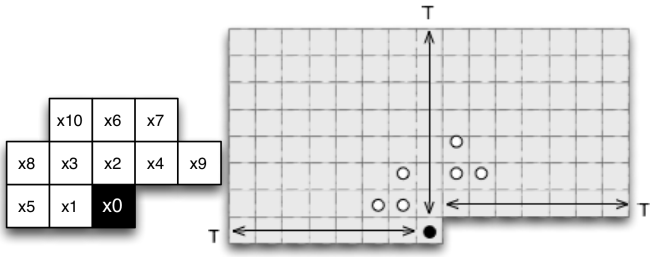
3.1. Overview

The Least squares prediction method adaptively filters a set of neighbours (Fig. 1.a) from each pixel to be predicted. The filter coefficients used are selected based on training over a window (Fig. 1.b) containing reconstructed data. This prediction can be represented by the following formula (1):

$$Xp(p(n)) = \sum_{j=1}^{N} a_j X(p(n-j)) \tag{1}$$

where aj are the weighting factors of the j-th position, N is the number of neighbours, p(n) is the position of the prediction and p(n-j) are the neighbours (also called the predictor mask, figure 1.a).

If the training window has enough edge pixels (strong variations in pixel intensity), then only one possible solution exists for the predictor's coefficients. If this happens, the prediction will correctly predict the new pixel and the edge direction. The coefficients can be determined by minimizing the Mean Square Error (MSE) inside the training area (2):



(a) Predictor mask (b) Training window

Fig. 1: Predictor mask (10th order) and training window (T = 7)

$$\frac{1}{\#(M)} \sum_{X(n-k) \subset M} \left(X(p(n-k)) - \sum_{j=k-1}^{k-N} a_j X(p(n-j)) \right)^2 \tag{2}$$

where M corresponds to the pixels inside the training window and #(M) is the number of elements inside that window.

3.2. Parallel approach

To start the paralellization proccess, the work to be performed is divided by the number of available GPU threads. Since the LSP is block-based and the block size changes continuously, we chose to make the number of GPU threads dependent on the block size, i.e., dynamic. The block sizes vary from 4x4 to 16x16 and we attribute each pixel to a number of threads (work-items) and have their values shared through global memory. The number of threads per pixel is different in each device tested. On the snapdragon processor 4 is the maximum possible number since each work-group has a limit of 1024 (for 16x16 block size, 4 work-items/pixel gives 1024). The second procedure consists of optimizing memory usage. Whenever possible, we make use of the fast local memory available. Since a lot of local memory was already being used by the other kernels in the MMP, we had to make consistent reductions of local memory usage until the LSP reached the maximum possible storage occupation. Local memory has significantly lower access times and needs to be exploited for maximum performance. As we constantly need to transfer data between host and device, the prediction is synchronized. Indeed, the prediction blocks are mapped from the device buffer into the host address space. This maps the prediction block from the device buffer into the host address space and thus everytime data changes in that block, the host updates accordingly. A minor performance improvement but still worth mentioning was obtained when the representation in memory of the prediction block was transformed from 2-dimensional into uni-dimensional.

3.3. Algorithmic propositions

3.3.1. Reducing prediction order

In [1] it is claimed that the LSP depends highly on the training process, and studies have shown that the training window has an optimized value of 7. However, results from that source show that reducing the prediction order does not have a considerable performance loss on the final result. A proposed change to the suggested parameters is then reducing the prediction order from 10 to 2 in order to reduce computational complexity even further, while keeping

the training window. In other words, making each prediction pixel to be a weighted average of just 2 neighbouring pixels (usually the left and up), while keeping each of these 2 pixels to be 'trained' by 7 neighbours.

(a) New (b) Old

Fig. 2: Left: Original prediction order; Right: Proposed prediction order. The gray pixels are the neighbors considered for the prediction of the black pixel

3.3.2. Conditional prediction

Since the LSP is great for areas of high frequency content (i.e., 'edges' or 'near-edges'), a very plausible way of reducing the LSP time consumption without compromising the overall performance of the image compression would be to apply the LSP only for pixels that differ significantly from their neighbors. For each pixel, we define an offset (basically a difference between two pixel intensity values). If a pixel has intensity equal or greater than the offset, the LSP is applied, otherwise, basic intra-prediction takes place. In the following figure we can see the pixels where the LSP is applied, for the Lena image particular case.

4. RESULTS

Experiments are tested on the Lena image, using MMP compression with low distortion. The LSP times and energy measurements correspond to the whole image coding process. Results were tested under three platforms, and the relevant information about each platform is shown in Table 1.

Platform	A	B	C
CPU	i7-4790K	i5-4260U	Krait 450
GPU	Tesla K40c	Intel HD Graphics 5000	Adreno 420
OS	Ubuntu 14.04	OSX Yosemite 10.10.4	Android-19

Table 1: Details of the tested platforms

Power consumption

The average power consumption of the platforms is:

- Platform A: 186 W for both the parallel and sequential approaches
- Platform B: 180 W for the sequential approach and 190 W for the parallel approach
- Platform A: 3.3 W for both the parallel and sequential approaches

4.1. Sequential

Experimental results for this approach can be seen in Table 2. The resulting image can be found in Fig. 3.a. Initial PSNR of the entire coding process and the prediction process:

- PSNR: 39.647672
- PSNR Prediction: 30.876516

Platform	A	B	C
Time	108.4	188.9	712.2
Energy	20154	33999	2350

Table 2: Time (seconds) and energy (joules) of the original sequential LSP version, for platforms A, B and C.

4.1.1. Prediction order reduction

Experimental results for this approach can be seen in Table 3. The resulting image can be found in Fig. 3.b.

Platform	A	B	C
Time	21.3	74.6	206.8
Energy	2953	13866	682

Table 3: Time (seconds) and energy (joules) of the prediction order reduction proposition, for platforms A, B and C.

- PSNR: 39.590109
- PSNR Prediction: 29.65527

4.1.2. Conditional prediction

Experimental results for this approach can be seen in Table 4. The resulting image can be found in Fig. 3.d and the chosen pixels for the LSP are found in Fig. 3.c.

Platform	A	B	C
Time	5.7	60.0	342.4
Energy	1068	10796	1130

Table 4: Time (seconds) and energy (joules) from the conditional prediction proposition for platforms A, B and C.

- PSNR: 39.538339
- PSNR Prediction: 29.438908

4.2. Paralellization

Experimental results for this approach can be seen in Table 5.

- PSNR: 39.647672
- PSNR Prediction: 30.876516

5. RELATED WORK

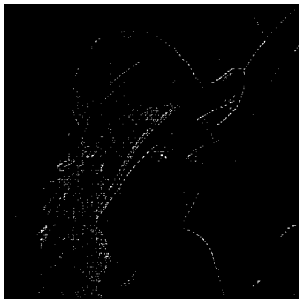
Least squares based approaches for image coding can be found in [9], [10], [11], [12] and many other papers in the literature. However, for the best of our knowledge, none of them considers the algorithm's behaviour from an energy consumption perspective. In this work, we developed a new parallel and optimized version of the

Platform	A	B	C
Time	21.7	183.6	168.3
Energy	2819	34158	555

Table 5: Time (seconds) and energy (joules) of the paralellized optimized approach, for platforms A, B and C.

(a) Original

(b) Prediction order = 2



(c) LSP chosen pixels

(d) Conditional prediction

Fig. 3: (a), (b) and (d) correspond to the resulting images of the image coding process for the original image, the prediction order reduction and the conditional prediction, respectively. (c) corresponds to the chosen pixels for LSP prediction

predictor, along with the entire image coder/decoder, targeted to a low-power mobile processor environment. In addition to this, we compare the energy consumed by the image coder with other platforms.

The first thing to point out is the time reduction from the parallel adaptation. Gains of up to 7.15 times were obtained for the LSP, with no performance loss. Also the energy dissipated was considerably lower for the parallel processing scenario. The snapdragon processor, while nearly 7x slower when compared to platform A, is 5 to 8 times more energy efficient. Interesting results regarding the energy efficiency of the snapdragon GPU may also be found in [13]. Experimental results from the Intel GPU show disadvantes in using this parallel solution both in time and energy spent, since the difference in performance from the Platform B CPU and GPU is not significant, as opposed to the other platforms. Furthermore, making the LSP apply only to areas of the image with high frequency content provides great acceleration, with little performance loss.

6. CONCLUSION

The use of GPU computing clearly benefits Least Squares Prediction (LSP) for image coding. Although communications between CPU and GPU impact performance, the LSP is sufficiently complex to benefit from the proposed GPU parallelization. The mobile snapdragon platform, while slower than the high-end tested GPUs, offers superior results in terms of energy efficiency, which is very convenient, since mobile energy sustainability is a key aspect of modern processing, and thus, currently a very active field of research. Furthermore, it is faster than desktop GPUs for the majority of experiments using a parallel approach. The parallelization process turns the mobile, low-power processor a more viable option for signal processing algorithms, since it reduced both time and energy consumption levels. The proposed sequential algorithmic changes to the LSP turn out to be interesting tradeoffs between energy and time reduction, and an acceptable performance loss. For future work we propose testing the developed LSP kernel in other platforms, such as Field-Programmable Gate Arrays (FPGAs), since energy consumption in costumized low-power processors has shown potential to improve results even further.

7. REFERENCES

[1] Danillo Bracco Graziosi, *Contributions to lossy and lossless image compression using multiscale recurrent pattern matching*, Ph.D. thesis, Federal University of Rio de Janeiro, Rio de Janeiro, Brazil, 2011.

[2] R. Vuduc and K. Czechowski, "What GPU computing means for high- end systems," vol. 31, no. 4, pp. 74–78, 2011.

[3] G. Bell, "Bell's law for the birth and death of computer classes," vol. 51, no. 1, pp. 86–94, January 2008.

[4] Mahendra Pratap Singh and Manoj Kumar Jain, "Evolution of processor architecture in mobile phones," *International Journal of Computer Applications*, vol. 90, no. 4, 2014.

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

[6] "OpenCL reveals Kindle Fire's true potential - http://gearburn.com/2013/01/opencl-reveals-kindle-fires-true-potential," January 2013.

[7] J. Yun G. Wang, Y. Xiong and J. Cavallaro, "Accelerating computer vision algorithms using OpenCL framework on the mobile GPU - a case study," *Acoustics, Speech and Signal Processing (ICASSP)*, pp. 2629–2633, May 2013.

[8] "Ndk guide - https://developer.android.com/ndk," 2015.

[9] Anil Kumar Tiwari and R. V. Raja Kumar, "Least Squares Based Optimal Switched Predictors for Lossless compression of images," *Multimedia and Expo, 2008 IEEE International Conference*, pp. 1129–1132, 2008.

[10] Daisuke Takago Kyoko Kato Shuitsu Matsumura, Takuji Maezawa and Tsuyosi Takebe, "Least-Square-Based Block Adaptive Prediction Approach for Lossless Image Coding," *Circuit Theory and Design, 2007. ECCTD 2007. 18th European Conference*, pp. 188–191, 2007.

[11] Lih-Jen Kau and Yuan-Pei Lin, "Least Squares-Based Lossless Image Coding with Edge-look-ahead," *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium*, p. 4 pp., 2006.

[12] A. Weinlich, J. Rehm, P. Amon, A. Hutter, and A. Kaup, "Massively Parallel Lossless Compression of Medical Images Using

Least-Squares Prediction and Arithmetic Coding," *Image Processing (ICIP), 2013 20th IEEE International Conference*, pp. 1680–1684, 2013.

[13] Song J. Park Dale R. Shires James A. Ross, David A. Richie and Lori L. Pollock, "A Case Study of OpenCL on an Android Mobile GPU," *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*, 2014.

B

Appendix B

OpenCL development on Snapdragon

Pedro Cordeiro

July 29th, 2015

Last edited: 10th September, 2015

Contents

1 Overview

1.1 Snapdragon

Snapdragon is a multiprocessor system that includes components such as a multimode modem, CPU, GPU, DSP, location/GPS, multimedia, power management, RF, optimizations to software and operating systems, memory, connectivity (Wi-Fi, Bluetooth), etc.
For a list of current commercial devices that include Snapdragon processors and to learn more about Snapdragon processors, go to:

http://www.qualcomm.com/snapdragon/devices

1.2 OpenCL on Snapdragon

This tutorial focuses on programming the GPU of the Snapdragon processor for general-purpose data parallel computation using OpenCL.
OpenCL on GPU is supported on the Adreno 300, 400 series GPU, and is fully conformant to the OpenCL standard.

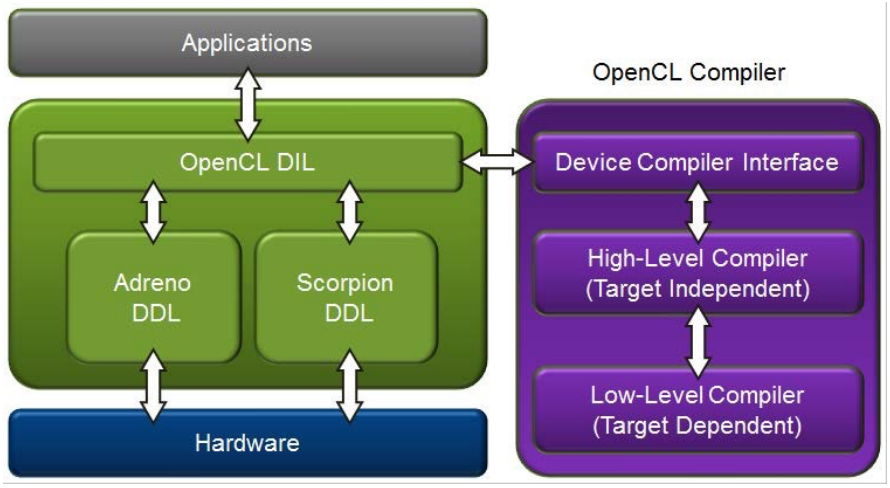


Figure 1: OpenCL architecture [1]

1.2.1 GPU extensions

Table 1 lists available OpenCL extensions on Snapdragon chipsets for the GPU.

Table 1: OpenCL extensions [1]

Extension	Description	Supported
cl_khr_global_int32_extended atomics	Additional atomic operations on 32-bit integers in global memory (min/max/and/or/xor)	Yes
cl_khr_global_int32_extended atomics	Additional atomic operations on 32-bit integers in global memory (min/max/and/or/xor)	Yes
cl_khr_local_int32_base atomics	Atomic operations on 32-bit integers in local memory (add, sub, xchg, inc, dec, cmpxchg)	Yes
cl_khr_local_int32_extended atomics	Additional atomic operations on 32-bit integers in global local (min/max/and/or/xor)	Yes
cl_khr_byte_addressable_store	Allows writing to char and short-based types	Yes
cles_khr_int64	Support for 64-bit integers (long, ulong, longn, ulongn)	No
cl_khr_fp16	Support for fp16 half data types (half, half2, half4, half8, half16)	Yes
cl_khr_gl_sharing	Shares OpenCL image data with texture and buffer objects	Yes
cl_qcom_ion_host_ptr	Augments the functionality provided by clCreateBuffer, clCreateImage2D, clCreateImage3D allowing apps to specify a new flag, CL_MEM_ION_HOST_PTR_QCOM; this new flag maps memory pointed to by Ion host_ptr to the GPU address space; hence, usage of this flag prevents a copy of the host memory to the device and vice versa	Yes
cl_qcom_limited_printf	Introduces the printf built-in function to OpenCL kernels on the earlier release of CL 1.1; this functionality is part of the CL 1.2 specification on the newer release	Yes
Cl_khr_egl_event	Allows creating OpenCL event objects linked to EGL fence sync objects, potentially improving efficiency of sharing images and buffers between the two APIs	Yes
EGL_IMG_image_plane_attribs	Allows creating an EGL image from a single plane of a multiplanar Android native image buffer	Yes
EGL_KHR_cl_event	Allows creating an EGL fence sync object linked to an OpenCL event object, potentially improving efficiency of sharing images between the two APIs	Yes
Cl_qcom_extended_images	Allows app to allocate 2D and 3D images up to a maximum of 8k x 8k size images	Adreno A3x products

2 Installation Guide

2.1 Determine location for installed packages

Packages you need to install:

- Apache Ant
- Android NDK
- Android SDK

First thing is to choose an appropriate installation path for these. Mine are (under MAC OSX):

- APACHE_ANT - /usr/local/bin/apache_ant
- ANDROID_NDK - /Users/pncordeiro/Library/Android/ndk
- ANDROID_SDK - /Users/pncordeiro/Library/Android/sdk

2.2 Configuring paths

Edit ~/.bashrc (or ~/.bash_profile) and add the paths to these packages. I did it this way (under MAC OSX):

```
#-----------------------------------------
# Android Development Setup
#-----------------------------------------
export ANDROID\SDK=/Users/pncordeiro/Library/Android/sdk
export ANDROID\NDK=/Users/pncordeiro/Library/Android/ndk
export APACHE\ANT=/usr/local/bin/apache\ant
# Setup paths for Android Development tools.
export PATH=$ANDROID\SDK/tools:$PATH
export PATH=$ANDROID\SDK/platform-tools:$PATH
export PATH=/usr/local/bin/apache\ant/bin:$PATH
export PATH=$ANDROID\NDK/toolchains/arm-linux-androideabi-4.9/
        prebuilt/darwin-x86/bin:$PATH
export PATH=$ANDROID\NDK:$PATH
```

2.3 Apache ANT

Apache Ant is open source software that is used to package the components of an app into a .apk file.

1. Download the package from http://ant.apache.org/bindownload.cgi
   (a) Select the current release of Ant
   (b) Select the tar.gz archive
2. Copy the downloaded file (apache-ant-1.8.4-bin.tar.gz) to a temp area, e.g., ~/temp/apache_ant.
3. Extract the archive:

```
tar -xvf apache-ant-1.8.4-bin.tar.gz
```

4. Create a directory to store the Ant release:

```
mkdir -p $APACHE_ANT
```

5. Copy the distribution

```
cp -r ~/temp/apache_ant $APACHE_ANT
```

6. Test Apache Ant

```
ant -version
```

2.4 Android SDK

1. Go to the standard Android developer site:

```
http://developer.android.com/sdk/installing/index.html
```

2. Select the Stand-Alone version
3. After unpacking the installer:

```
cd $ANDROID_SDK/tools
```

4. And execute:

```
android sdk.
```

Note: This downloads all the required Android packages. The process may take some time.

2.5 Android NDK

1. Go to the standard Android developer site:

```
http://developer.android.com/ndk/downloads/index.html
```

2. Select the download for your platform

2.5.1 Installing OpenCL library

1. Pull the libOpenCL.so library from the QTI development device:

```
$ adb pull /system/vendor/lib/libOpenCL.so
```

2. Install the library to your NDK lib directory, e.g.,

$ANDROID_NDK/platforms/android-19/arch-arm/usr/lib/libOpenCL.so

3 OpenCL Development

3.1 Download Adreno SDK

1. Download the Adreno SDK for your platform:

   https://developer.qualcomm.com/software/adreno-gpu-sdk/tools

   This SDK will bring you a framework (bunch of functions that are optimized for the Snapdragon processor), some Samples, Documentation, etc

2. Copy a Sample from the SDK folder (and rename it to your app name) under:

   Development > Samples > OpenCL

3. Now open your new folder (with your app name) and you should see an Android project file tree

3.2 Adreno framework

3.2.1 Overview

Platform-independence is achieved through the hardware and Operating System abstraction layers of the Framework.
The Framework code is organized in a file hierarchy which enables all platform-independent to be separated from platform-dependent code. You will see that platform-specific implementations are kept in platform-specific directories. For example, the file FrmStdlib.h declares most of the C standard library functions (with abstracted names) whose implementations can be found in FrmStdlib_Platform.cpp that is present in each supported platform directory. Each SDK samples has a class called CSample that derives from the CFrmApplication class. The derived class is responsible for construction, initialization, handling resize events, updating and rendering the scene, and cleanup.

The following snippets are present in [2].

```
class CSample : public CFrmApplication
{
        ...
        public:

        CSample( const CHAR* strName );

        virtual BOOL Initialize();
        virtual BOOL Resize();
        virtual VOID Update();
        virtual VOID Render();
        virtual VOID Destroy();
};
```

The CSample object is actually created in a global function called FrmCreateApplicationInstance(). All samples using the Framework must supply this function, or else the sample will fail to build.

```
CFrmApplication* FrmCreateApplicationInstance()
{
        return new CSample( "Sample Name" );
}
```

To better understand this framework's features, like File I/O Abstraction or C Standard Library Abstraction, you may read the *Framework.htm* file.

3.2.2 Reference

Inside the framework folder, you can find the functions that guarantee platform-independence. To avoid wasting too much time searching for the correct functions, you should read the *FrameworkReference.htm* file that shows you the main changes you need to make in your code.

3.3 Understanding Android project files

AndroidManifest.xml

The Android manifest is used to pull in all components in the SDK package.
Here you can edit your package name, android version, app permisions, and so on.

jni/Application.mk

The Application.mk file defines the compiler versions and Android platform version, e.g.:

```
APP_ABI := armeabi-v7a
APP_PLATFORM := android-19
APP_STL := gnustl_static
APP_CPPFLAGS += -fexceptions -frtti
```

jni/Android.mk

The Android.mk file is your makefile to build the NDK components that are included in the SDK.
The output should be a library (.so) file.

3.4 Building/Installing the App

3.4.1 Manually

To compile a sample natively, use the ndk-build system (via Cygwin if using Windows).

1. Navigate to the Android/jni directory of the sample and run the following command:

   ```
   $ ndk-build
   ```

2. Go to the Android/ directory for the sample (up one from jni/) and type (where <SampleName>is the name of the sample, e.g., BandwidthTest if building the BandwidthTest sample):

```
$ android update project -p . -n <SampleName> -t android-19
```

Note: Use 'android.bat' if using Windows

3. To install the app assets and build the .apk package:

```
$ ./InstallAssets.sh
$ ant debug
```

4. And to install the sample on the connected device, type:

```
$ adb install -r bin/<SampleName>-debug.apk
```

3.4.2 Automatically

The easiest way to build the Android samples is to run the script build.sh from Samples/OpenCL/Build/Android or build_android.sh under Samples/OpenCL. Running this script with no command line options from the shell builds all of the Adreno SDK samples. The default Android target to build is android-19.

- To build just your sample (in this case DeviceQuery):

```
$ ./build.sh -t DeviceQuery
```

Note: You may need super user privileges

- To build and install the sample on the device, use the following command: e.g., to build the DeviceQuery sample and install it on the connected device (using adb):

```
$ ./build.sh -t DeviceQuery -i
```

- To just install an individual sample to the device:

```
$ ./install.sh -t DeviceQuery
```

3.5 Running the App

3.5.1 Manually

The samples can be run manually from a command line. To run the sample named <SampleName>, type the following:

```
$ adb logcat -c
$ adb shell am start -n
com.qualcomm.<SampleName>/com.qualcomm.common.AdrenoNativeActivity
-e DEVICE gpu -e RUNTESTS true
$ adb logcat | grep OpenCL
```

The following command-line options are supported by the samples:

- -e DEVICE [gpu—all]

- -e RUNTESTS [true—false]

To stop the sample, either close it from the Android user interface or type the following:

```
$ adb shell am force-stop com.qualcomm.<SampleName>
```

3.5.2 Automatically (in test mode)

The easiest way to run Android samples on your device from your computer is to run the script run_tests.sh. To run an individual sample, e.g., VectorAdd, type the following:

```
./run_tests.sh -t VectorAdd
```

3.6 Debugging the App with GDB

'ndk-gdb' is a script that allows you to easily launch a native debugging session for your NDK-generated machine code. It is located at the top-level directory of the NDK, and can be called from your application project directory, or any of its sub-directories.

3.6.1 Requirements

1. Your application must be debuggable

   To do this, you need to set the android:debuggable attribute to 'true' in your AndroidManifest.xml file.

2. You are running your application on Android 2.2 (or higher)

3.6.2 Usage

By default, ndk-gdb will search for an already-running application process, and will dump an error if it doesn't find one. You can however use the –start or –launch=<name>option to automatically start your activity before the debugging session.

When it successfully attaches to your application process, ndk-gdb will give you a normal GDB prompt, after setting up the session to properly look for your source files and symbol/debug versions of your generated native libraries.

To see a list of options, type 'ndk-gdb –help'.
You can find the GDB manual here:

```
https://www.gnu.org/software/gdb/documentation/
```

4 Monitoring/Profiling the App

4.1 Adreno Profiler

The Adreno Profiler tool facilitates with profiling the OpenCL app to identify optimization opportunities through the use of many performance measurement metrics and static analysis of OpenCL kernels.

4.1.1 Installation

In order to run Adreno Profiler the following two dependencies need to be installed:

- Mono Framework 2.10.5 (It is important to get that particular version of the Mono Framework)
  Download and install from:

  http://download.mono-project.com/archive/2.10.5/macos-10-x86/0/

- xQuartz
  Download and install from:

  http://xquartz.macosforge.org/landing/

4.1.2 Usage

First of all:

1. Verify ADB is available in your system PATH
2. Open an xQuartz Terminal window
3. Open AdrenoProfiler

Using Adreno Profiler to examine differences

Grapher

1. Click Grapher on the top tool bar to open the grapher panel.
2. n the Metrics Browser, click Grapher Metrics to display the metric list.
3. Expand EGL and double-click the FPS option to enable logging.
4. Start PostProcessCLGLES on the device and make sure it is running during profiling.
5. Click Connect on the tool bar and select the PostProcessCLGLES app from the popup dialog.
6. Let the app run for a few seconds and click Disconnect.
7. The grapher should plot FPS in real time.

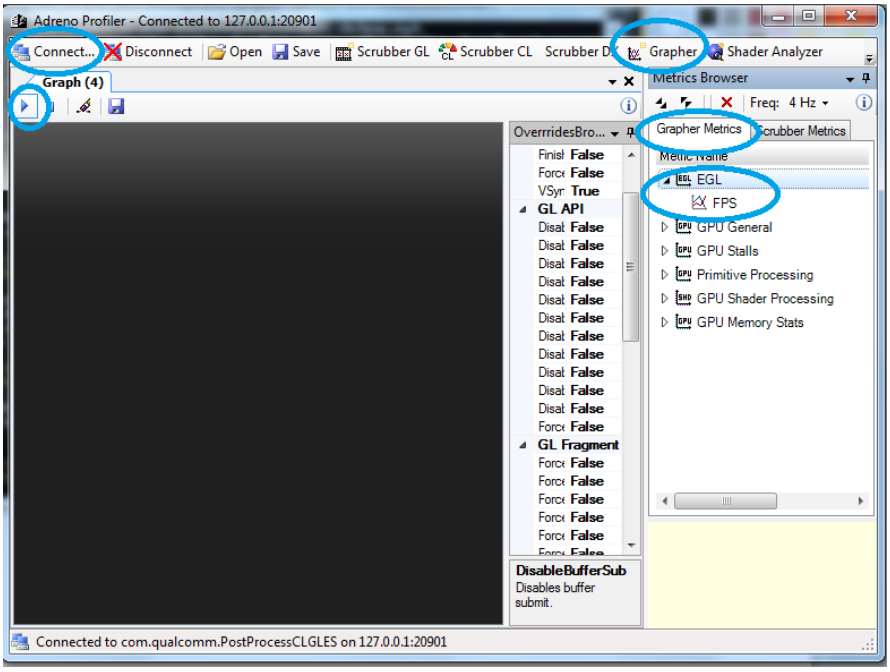


Figure 2: Buttons for the previous steps [1]

Scrubber (captures useful data for a CL app)

1. Open a shell and run:

   ```
   adb shell setprop ADRENO_PROFILER_ENABLE_OPENCL 1
   ```

2. Start up Adreno Profiler.
3. Click Scrubber CL on the top toolbar.
4. Start the PostProcessCLGLES app on the device and make sure it runs during profiling.
5. Click Connect and select PostProcessCLGLES from the popup dialog.
6. Click the red Record button in the scrubber panel.
7. Let the app run for a couple of seconds and click Record again to stop.
8. In the Gantt chart window, continue to zoom in until the API marker bars are visible.

NOTE: No extra API call is needed to copy the buffer. Mouse over the bar and the time spent in the copying buffer API call appears. Copying occurs for every kernel operation that increases the overall time.

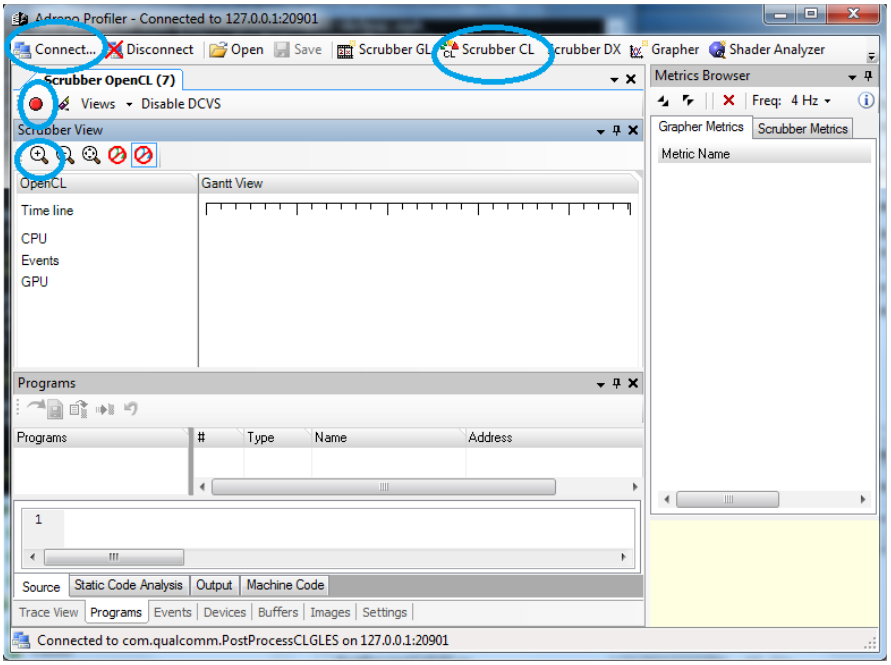


Figure 3: Buttons for the previous steps [1]

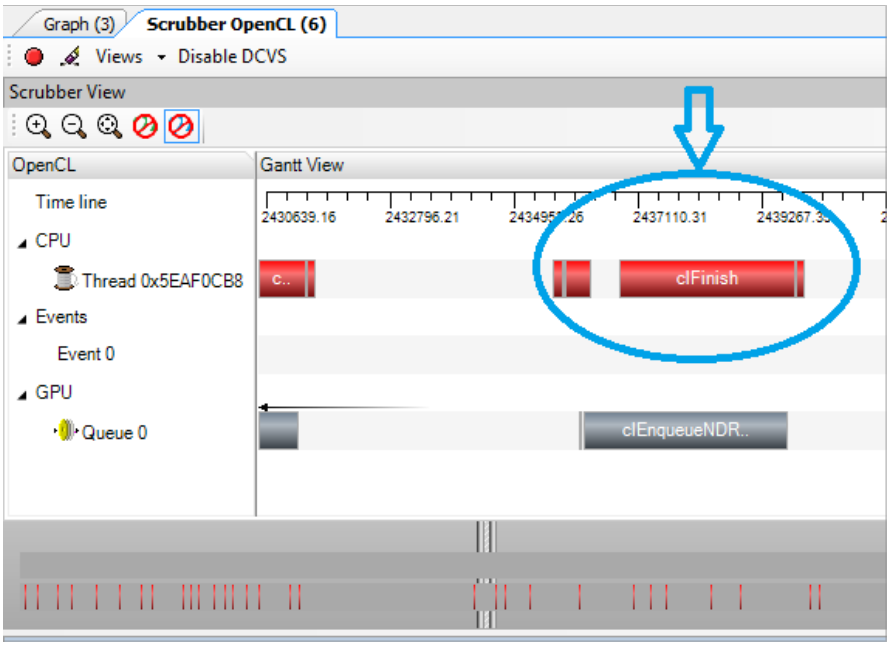


Figure 4: Time spent in the copying buffer API [1]

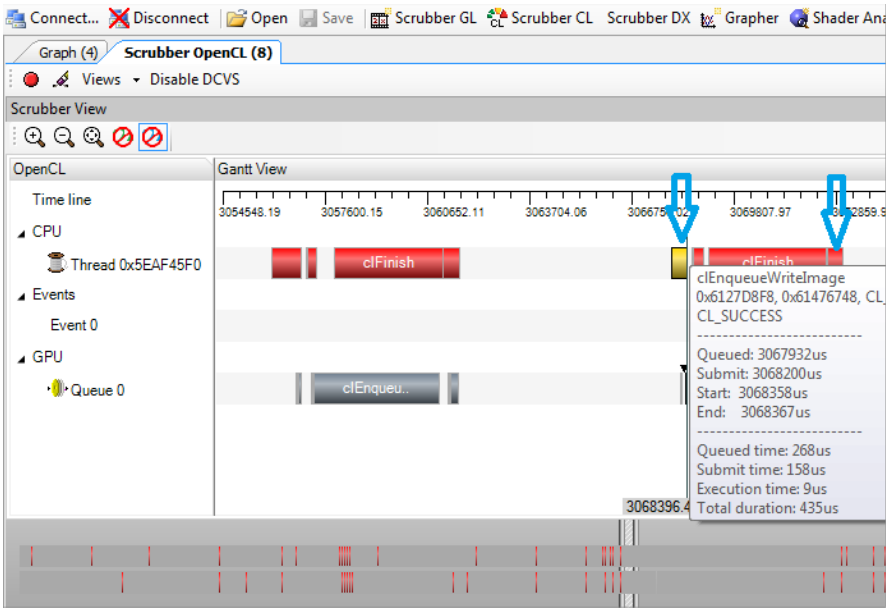


Figure 5: With some detail [1]

4.2 Trepn Profiler

Trepn Profiler is an on-target power and performance profiling application for mobile devices. Although Trepn runs on most Android devices, additional features are available when used with devices featuring Qualcomm Snapdragon processors or development hardware.
With Trepn Profiler, developers can better understand the impact of their programming choices on both power and performance.

4.2.1 Installation

- From Google Play:

  `https://play.google.com/store/apps/details?id=com.quicinc.trepn`

- From QDN:

  `https://developer.qualcomm.com/download/trepn-profiler.zip`

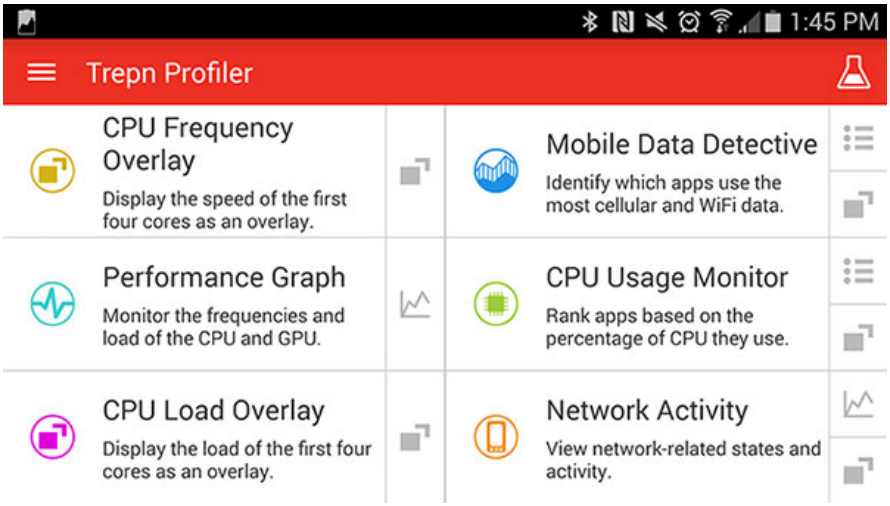


Figure 6: Six fast loading presets allow you to quickly profile your device [3]

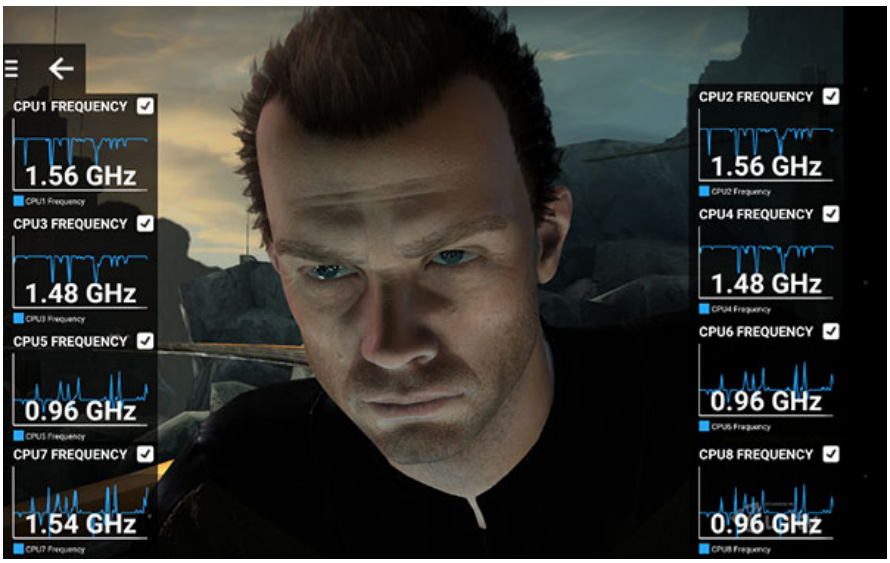


Figure 7: Real-time overlays allow you to see the impact of your actions on any app [3]

Figure 8: Trepn Profiler shows whether an app is CPU or GPU-constrained [3]

References

[1] Qualcomm Technologies, Inc., *Snapdragon OpenCL General Programming and Optimization*, August 2014.

[2] Qualcomm Technologies, Inc., *Adreno SDK, Online DOcumentation*, 2014.

[3] [Online], *https://developer.qualcomm.com/software/trepn-power-profiler*.