Dissertation

Masters in Computer Engineering – Mobile Computing

Optimization of Pattern Matching Algorithms for Multi- and Many-Core Platforms

Pedro Miguel Marques Pereira

Leiria, September 21, 2016

Dissertation

Masters in Computer Engineering – Mobile Computing

Optimization of Pattern Matching Algorithms for Multi- and Many-Core Platforms

Pedro Miguel Marques Pereira

Master's thesis carried out under the guidance of Professor Patrício Rodrigues Domingues, Professor at School of Technology and Management of the Polytechnic Institute of Leiria and co-orientation of Professor Nuno Miguel Morais Rodrigues, Professor at School of Technology and Management of the Polytechnic Institute of Leiria and Professor Sérgio Manuel Maciel Faria, Professor at School of Technology and Management of the Polytechnic Institute of Leiria.

Leiria, September 21, 2016

Acknowledgement

I reserve this page to thank all whom supported me. My thanks go firstly to Professor Patrício Rodrigues Domingues for pulling me to this project and for his solid patience and availability. By extent, I thank adviser Professors Nuno M. M. Rodrigues and Professor Sérgio M. M. Faria for their project insight and patience over my uncertainties. I would like also to thank lab colleague Gilberto Jorge for filling in with his electronics knowledge whenever necessary. The contents of this dissertation would not be possible without any of them.

I also want to thank my family and girlfriend for letting me work after hours whenever it was needed without any concerns or preoccupations and for providing to my every need so that I could concentrate my daily life on this work. I also thank them for their predisposition and encouragement to see the end of this latest academic step.

Finally, I would like to thank the Escola Superior de Tecnologia e Gestão of the Instituto Politécnico de Leiria and the Instituto de Telecomunicações for the environment and working conditions that ultimately enabled this work. This work was supported by project IT/LA/P01131/2011, entitled "OPAC - Optimization of pattern-matching compression algorithms for GPU's", and financed by national Portuguese funds (PIDDAC) through Fundação para a Ciência e a Tecnologia / Ministério da Educação e Ciência (FCT/MEC), contract PEst-OE/EEI/LA008/2013.

Resumo

A compressão de imagem e vídeo desempenha um papel importante no mundo de hoje, permitindo o armazenamento e transmissão de grandes quantidade de conteúdos multimédia. No entanto, o processamento desta informação exige elevados recursos computacionais, pelo que a optimização do desempenho computacional dos algoritmos de compressão tem um papel muito importante.

O Multidimensional Multiscale Parser (MMP) é um algoritmo de compressão de padrões que permite comprimir conteúdos multimédia, nomeadamente imagens, alcançando uma boa taxa de compressão enquanto mantém boa qualidade de imagem Rodrigues et al. [2008]. Porém, este algoritmo demora algum tempo para executar, em comparação com outros algoritmos existentes para o mesmo efeito. Assim sendo, duas implementações paralelas para GPUs foram propostas por Ribeiro [2016] e Silva [2015] em CUDA e OpenCL-GPU, respectivamente. Nesta dissertação, em complemento aos trabalhos referidos, são propostas duas versões paralelas que executam o algoritmo MMP em CPU: uma recorrendo ao OpenMP e outra convertendo a implementação OpenCL-GPU existente para OpenCL-CPU. As soluções propostas conseguem melhorar o desempenho computacional do MMP em 3× e 2.7×, respectivamente.

O High Efficiency Video Coding (HEVC/H.265) é a norma mais recente para codificação de vídeo 2D amplamente utilizado. Sendo por isso alvo de muitas adaptações, nomeadamente para o processamento de imagem/vídeo holoscópico (ou *light field*). Algumas das modificações propostas para codificar os novos conteúdos multimédia baseiam-se em compensações de disparidade baseadas em geometria (SS), desenvolvidas por Conti et al. [2014], e um módulo de Transformações Geométricas (GT), proposto por Monteiro et al. [2015]. Este algoritmo para compressão de imagens holoscópicas baseado no HEVC, apresenta uma implementação específica para pesquisar micro-imagens similares de uma maneira mais eficiente que a efetuada pelo HEVC mas a sua execução é consideravelmente mais lenta que o HEVC. Com o objetivo de possibilitar melhores tempos de execução, escolhemos o uso da API OpenCL como linguagem de programação para GPU de modo a aumentar o desempenho do módulo. Com a configuração mais onerosa, reduzimos o tempo de execução do módulo GT de 6.9 dias para

pouco menos de 4 horas, atingindo efetivamente um aumento de desempenho de 45×.

Palavras-chave: *computação de alto desempenho, Multi-Thread, Multi-CPU, Multi-GPU, OpenCL, OpenMP*

Abstract

Image and video compression play a major role in the world today, allowing the storage and transmission of large multimedia content volumes. However, the processing of this information requires high computational resources, hence the improvement of the computational performance of these compression algorithms is very important.

The Multidimensional Multiscale Parser (MMP) is a pattern-matching-based compression algorithm for multimedia contents, namely images, achieving high compression ratios, maintaining good image quality, Rodrigues et al. [2008]. However, in comparison with other existing algorithms, this algorithm takes some time to execute. Therefore, two parallel implementations for GPUs were proposed by Ribeiro [2016] and Silva [2015] in CUDA and OpenCL-GPU, respectively. In this dissertation, to complement the referred work, we propose two parallel versions that run the MMP algorithm in CPU: one resorting to OpenMP and another that converts the existing OpenCL-GPU into OpenCL-CPU. The proposed solutions are able to improve the computational performance of MMP by 3× and 2.7×, respectively.

The High Efficiency Video Coding (HEVC/H.265) is the most recent standard for compression of image and video. Its impressive compression performance, makes it a target for many adaptations, particularly for holoscopic image/video processing (or light field). Some of the proposed modifications to encode this new multimedia content are based on geometry-based disparity compensations (SS), developed by Conti et al. [2014], and a Geometric Transformations (GT) module, proposed by Monteiro et al. [2015]. These compression algorithms for holoscopic images based on HEVC present an implementation of specific search for similar micro-images that is more efficient than the one performed by HEVC, but its implementation is considerably slower than HEVC. In order to enable better execution times, we choose to use the OpenCL API as the GPU enabling language in order to increase the module performance. With its most costly setting, we are able to reduce the GT module execution time from 6.9 days to less then 4 hours, effectively attaining a speedup of 45×.

Keywords: *high performance computing, manycore, multi-CPU, multi-GPU, OpenCL, OpenMP*

List of Figures

2.1 MMP Test Images 11

2.2 HEVC Test Figure 11

3.1 OpenMP Fork-Join Model 24

3.2 MMP-Sequential encoder partial call-graph 28

3.3 OpenMP Thread-Variation Effect over Jetson 32

3.4 OpenMP Thread-Variation Effect over Laptop 33

3.5 OpenMP Lambda-Variation Effect over Server 3 35

3.6 MMP-OpenMP mean watt oscillation over Jetson 38

3.7 MMP-OpenMP mean watt oscillation over Server 2 39

3.8 Comparison between CPU and GPU Architecture 41

3.9 Work distribution with SIMD instruction sets 44

3.10 MMP-OpenCL mean watt oscillation over Server 2 48

3.11 MMP Rate-Distortion Curves 50

4.1 GT 9 corner points 53

4.2 GT corner point combinations 54

4.3 HEVC+SS+GT sequential encoder partial call-graph 58

4.4 HEVC+SS+GT-OpenCL K2 Visual Explanation 73

4.5 Fast Walsh-Hadamard Transform Calculations 75

List of Tables

2.1 Hardware Equipment Details - CPUs 8

2.2 Hardware Equipment Details - GPUs 9

2.3 Phoronix RamSpeed Test Results 15

2.4 LZMA Benchmark Results 16

2.5 PMBW Benchmark Results 17

2.6 PMBW ScanRead64PtrUnrollLoop Benchmark Results - Access Time . 17

3.1 MMP-Sequential Results 23

3.2 MMP-CUDA Results 23

3.3 MMP-OpenCL-GPU Results 24

3.4 MMP-OpenMP Results 31

3.5 MMP-OpenMP Results with/without memory related problems 33

3.6 MMP-Sequential encoder with improvements 34

3.7 Comparison between improved MMP-Sequential encoder and MMP-OpenMP 34

3.8 OpenMP Energy Consumption Performance over Jetson 38

3.9 OpenMP Energy Consumption Performance over Server 2 39

3.10 Execution Time Evolution from OpenCL-GPU to CPU 45

3.11 Kernel Time Evolution from OpenCL-GPU to CPU for the Laptop . . 47

3.12 OpenCL Energy Consumption Performance over Server 2 48

4.1 HEVC-Sequential Results 60

4.2 HEVC+SS+GT-OpenMP Extremes 62

4.3 HEVC+SS+GT-OpenMP Results 63

4.4 HEVC+SS+GT-OpenCL clEnqueueCopyBuffer Time 69

4.5 HEVC+SS+GT-OpenCL Kernel Copy Time 69

4.6 HEVC+SS+GT-OpenCL N Iteration Simulations for N=2 70

4.7 HEVC+SS+GT-OpenCL Memory Used in GPU 71

4.8 HEVC+SS+GT-OpenCL Kernel Execution Times 80

4.9 HEVC+SS+GT-OpenCL Results 81

List of Abbreviations

Abbreviation [A-I]	Description
ALU	Arithmetic Logic Unit
API	Application Programming Interface
ARM	Advanced RISC Machines
AVC	Advanced Video Coding
AVX	Advanced Vector Extensions
CD	Compute Device
CPU	Central Processing Unit
CSV	Comma-Separated Values
CU	Compute Unit
CUDA	Compute Unified Device Architecture
D2D	Device-to-Device
D2H	Device-to-Host
DDR3	Double Data Rate, type Three
DDR3L	Double Data Rate, type Three, LowVoltage
DFT	Discrete Fourier Transforms
DIMM	Dual Inline Memory Module
DSP	Digital Signal Processor
ESTG	School of Technology and Management (Escola Superior de Tecnologia e Gestão)
FCT	Fundação para a Ciência e a Tecnologia
FFT	Fast Fourier Transform
FP	Floating Point
FPGA	Field-Programmable Gate Array
FWHT	Fast Walsh-Hadamard Transform
GCC	GNU Compiler Collection
GPU	Graphics Processing Unit
H2D	Host-to-Device
HAD	HEVC Walsh-Hadamard Transform
HDMI	High-Definition Multimedia Interface
HEVC	High Efficiency Video Coding
I/O	Input/Output
IEC	International Electrotechnical Commission
ILP	Instruction Level Parallelism
IPL	Instituto Politécnico de Leiria
ISO	International Organization for Standardization

Abbreviation [I-S]	Description
ITU	International Telecommunication Union
ITU-T	ITU Telecommunication Standardization Sector
JCT-VC	Joint Collaborative Team on Video Coding
L1	(cache) Level 1
L2	(cache) Level 2
LLC	Last Level Cache
LP	(Jetson mode) LowPower
MEC	Ministério da Educação e Ciência
MF	(Jetson mode) MaxFrequency
MIPS	Million Instructions Per Second
MMP	Multidimensional Multiscale Parser
MMP-OpenCL	MMP OpenCL implementation
MMP-OpenCL-CPU	MMP OpenCL implementation optimized for CPUs
MMP-OpenCL-GPU	MMP OpenCL implementation optimized for GPUs
MMP-OpenMP	MMP OpenMP implementation
MP	(Jetson mode) MaxPerformance
MPEG	Moving Picture Experts Group
MPI	Message Passing Interface
MSD	Mean Squared Deviation
MSE	Mean Square Error
NVCC	NVidia CUDA Compiler
OpenCL	Open Computing Language
OpenCL-CPU	OpenCL optimized for CPU
OpenCL-GPU	OpenCL optimized for GPU
OpenMP	Open Multi-Processing
OS	Operating System
PBM	Portable BitMap graphics
PE	Processing Element
PGAS	Partitioned Global Address Space
PGM	Portable GrayMap graphics — an alternatice to PBM
PIDDAC	Programa de Investimentos e Despesas de Desenvolvimento da Administração Central
POSIX	Portable Operating System Interface
PSNR	Peak Signal-to-Noise Ratio
RAM	Random-Access Memory
RISC	Reduced Instruction Set Computer
RMS	Root Mean Square
SIMD	Single Instruction, Multiple Data
SMX	Streaming Multiprocessor
SoC	System on a Chip
SO-DIMM	Small Outline Dual Inline Memory Module
SSE	Streaming SIMD Extensions

Abbreviation [T-Z]	Description
TLP	Thread-Level Parallelism
µops	micro-operations
USB	Universal Serial Bus
VCEG	Video Coding Experts Group (also know as Question 6 (Visual coding) of Working Party 3 (Media coding) of Study Group 16 (Multimedia coding, systems and applications) of the ITU-T)
VMX	Virtual Machine eXtensions
WHT	Walsh-Hadamard Transform
WORA	Write Once, Run Anywhere

Symbols	Description
λ	Ponderation factor of the relative weight of the distortion and the number of bits in the cost for a block representation expression
D	Distortion caused by a block representation of the original image vs an element of the dictionary
J	Cost for a block representation
R	Number of necessary bits for a block representation
µ	Micro
$\overline{x}$	Mean

Index

Acknowledgement **III**

Resumo **V**

Abstract **VII**

List of Figures **X**

List of Tables **XII**

List of Abbreviations **XIII**

1 Introduction **1**
- 1.1 Motivation . . . 2
- 1.2 Main Goals and Contributions . . . 3
  - 1.2.1 Publications . . . 4
- 1.3 Outline . . . 5

2 Computational Environment and Methodology **7**
- 2.1 Equipment and Configurations . . . 7
- 2.2 Results Acquisition and Used Images . . . 10
  - 2.2.1 SpeedUp . . . 12
  - 2.2.2 PSNR and Bitrate . . . 12
- 2.3 Energy Measurements Methodology . . . 13
- 2.4 Base Bandwidth Speeds . . . 14

3 MMP – Multi-CPU, Many-Core, Multi-Thread **19**
- 3.1 Multidimensional Multiscale Parser . . . 20
- 3.2 Previous Versions . . . 21
  - 3.2.1 MMP-CUDA and MMP-OpenCL-GPU . . . 21
  - 3.2.2 Base Reference Results . . . 23
- 3.3 OpenMP . . . 24
  - 3.3.1 Profiling MMP-Sequential . . . 26
  - 3.3.2 Implementation . . . 30
  - 3.3.3 Results . . . 31
- 3.4 OpenCL-CPU . . . 40
  - 3.4.1 Migrating from OpenCL-GPU to OpenCL-CPU . . . 42
  - 3.4.2 SIMD-based Optimizations . . . 43
  - 3.4.3 Results . . . 45
- 3.5 Conclusions . . . 49

4 HEVC+SS+GT – Multi-GPU, Many-Core, Multi-Thread **51**
- 4.1 High Efficiency Video Coding . . . 51

4.1.1 HEVC-based holoscopic coding using SS and GT compensations 52
4.2 Parallelization Impediments 55
4.3 Profiling HEVC+SS+GT 57
4.3.1 Base Reference Results 60
4.4 OpenMP 60
4.4.1 Implementation 61
4.4.2 Results 62
4.5 OpenCL-GPU 63
4.5.1 Implementation 64
4.5.2 Results 79
4.6 OpenCL-Multi-GPU 81
4.6.1 Implementation 81
4.6.2 Results 83
4.7 Conclusions 84

5 Conclusions **85**

Bibliography **87**

Chapter 1

Introduction

The research presented in this dissertation relies on the optimization of an image compression algorithm, Multidimensional Multiscale Parser (MMP) [6, 7], that was recently parallelized to GPUs by Ribeiro [2] and Silva [3], resorting to both CUDA and OpenCL languages for GPU programming. To complement their work we further research and develop similar parallel implementations for multicore CPUs. One in OpenCL based on the existing implementation [3] but for CPU, and another resorting to OpenMP – a more CPU directed API – to provide result comparisons.

Afterwards, we decide to exploit our knowledge in the parallelization field to improve a specific algorithm based on the High Efficiency Video Coding (HEVC). This task consisted on the optimization of the recent HEVC-base implementation of Monteiro et al. [5] targeted to compress holoscopic image/video based on geometry-based disparity compensation (SS) and geometric transformations (GT) – henceforth referred as HEVC+SS+GT. This image compression algorithm has been chosen due to the emerging usage of plenoptic (or light field) images in various fields of application. The plenoptic function concept [8], represented by Equation 1.1, was introduced and developed by Adelson and Bergen and describes everything that can be seen (from *plenus*, complete or full, and *optic*). It embodies a way of thinking about light, not only as a series of images formed from 3D space onto 2D planes (whether retinal or cameras), but rather as a three-dimensional field of co-existing rays. This is a formalization of the concept of light-field proposed and coined by Gershun [9], with roots back to concepts introduced by Faraday [10] in the mid 19$^{\text{th}}$ century. Later, McMillan and Bishop [11] discuss the representation of 5D light fields as a set of panoramic images at different 3D locations. The plenoptic function represents the intensity or chromacity of the light observed from every position and direction in 3-dimensional space. In image based modelling, the aim is to reconstruct the plenoptic function from a set of examples images. Once the plenoptic function has been reconstructed it is straightforward to generate images

by indexing the appropriate light rays. If the plenoptic function is only constructed for a single point in space then its dimensionality is reduced from 5 to 2. In the function Equation 1.1, the V_x, V_y and V_z variables are the possible viewing-points, and the x and y variables parametrize the rays coordinates that are entering a given view-point (i.e. spatial coordinates of an imaginary picture plane erected at a unit distance from the eye pupil), for every wavelength λ, at every time t.

$$P = P(x, y, \lambda, t, V_x, V_y, V_z) \tag{1.1}$$

Henceforth, this type of images presents much richer information than common 2D images, as they include information from several view directions, allowing to extract 3D information. However the computational complexity associated with its processing and compression may increase significantly. Therefore, for most applications it is imperative to reduce the algorithm execution time. To this end, we elected the use of the OpenCL API as the GPU enabling language and we opted to initially use the OpenMP API to ease the algorithms learning and provide early implementation comparisons and forecasts.

1.1 Motivation

All market-leading processor vendors have already started to pursue multicore processors as an alternative to high-frequency single-core processors for better energy and power efficiency [12]. With this widespread adoption, scalability is increasingly important. In addition to the rapid decrease in hardware costs, use of multicore CPUs and high end GPUs in commodity machines, mobile phones, laptops, tablets and many more other low cost devices has become a reality.

The transition to multicore processors no longer provides the free performance gain enabled by increased clock frequency for programmers. Parallelization of existing serial/sequential programs has become the most powerful approach to improve application performance. Not surprisingly, parallel programming is still extremely difficult for many programmers mainly because programmers are not taught to think in parallel so far. Furthermore it is not an easy task because several issues like: race condition, data hazards, control hazards and load imbalance between cores, due to highly non-uniform data distribution, are some common issues that might crop up during refactoring or

implementation. This trend places a great responsibility on programmers and software for program optimization. Thus in order to improve the performance, vectorization and thread-level parallelism (TLP) will be increasingly relied upon in place of instruction-level parallelism (ILP) and increased clock frequency.

Despite the existing body of parallelization knowledge in scientific computing, databases and operating systems, we still have many application areas today that are still not attractive for parallelization. Adapted from [13]:

> (...) Many techniques exist that attempt to automatically parallelize loops, although they are most effective on inner loops. In contrast, coarse-grain parallelism is difficult to detect. High level loops may be nested across function and file boundaries, and their significance may not be detectable. Currently, manual techniques remain the predominant method of detecting and introducing high level TLP into a program. (...)

Consequently, the software engineering of such applications also did not receive much attention. However, we are now at an inflection point where this is changing, as ordinary users possess multi-core computers and demand software that exploits the full hardware potential.

In the quest for faster computing, it is also important to consider its energy demand. This is an important topic since the dominant cost of ownership for computing is energy, not only the directly consumed by the devices but also the needs for refrigeration purposes [14]. Moreover, energy efficiency in computing has also become a marketing and ethical trend linked to green computing [15].

1.2 Main Goals and Contributions

With this research we intend to prove that computer CPUs still have a lot of computational power to offer that often goes unexplored. Through software frameworks such as OpenMP and OpenCL, the computational power of multicore processors can be tailored in an almost effortless way by programmers.

The main contributions of this work are: 1) assessing the gain achieved through optimization for multicore CPU instead of GPU; 2) evaluation of both the easiness

and performance gain of using CPU-based vectorization instructions through OpenCL; 3) review and comparison of power consumption over different implementations and hardware equipments; 4) proposal and implementation of CPU-driven MMP implementations that exploit the hardware parallelization possibilities and; 5) proposal and implementation of a parallel HEVC+SS+GT version to support the ongoing research on plenoptic image compression.

Knowing that the MMP algorithm is a computational demanding image encoder/ decoder, with sequential dependencies among individual input blocks, insures some representativeness as an application for parallelization. Likewise, the HEVC+SS+GT algorithm comes with an exponential data and computational growth with large data dependencies between iterations which also insures its representativeness.

1.2.1 Publications

Additionally, this work resulted in the production and submission of tree scientific articles, of which two were already accepted:

- (Accepted) "**Optimized Fast Walsh-Hadamard Transform on OpenCL-GPU and OpenCL-CPU**", in the 6$^{\text{th}}$ *International Conference on Image Processing Theory, Tools and Applications* (IPTA'16), by Pedro M. M. Pereira, Patricio Domingues, Nuno M. M. Rodrigues, Sergio M. M. Faria and Gabriel Falcao.
- (Accepted) "**Optimizing GPU code for CPU execution using OpenCL and vectorization: a case study on image coding**", in the 16$^{\text{th}}$ *International Conference on Algorithms and Architectures for Parallel Processing* (ICA3PP), by Pedro M. M. Pereira, Patricio Domingues, Nuno M. M. Rodrigues, Gabriel Falcao and Sergio M. M. Faria.
- (Submitted) "**Assessing the Performance and Energy Usage of Multi-CPUs, Multi-core and Many-core Systems: the MMP Image Encoder Case Study**", in the *International Journal of Distributed and Parallel Systems* (IJDPS), by Pedro M. M. Pereira, Patricio Domingues, Nuno M. M. Rodrigues, Gabriel Falcao and Sergio M. M. Faria.

1.3 Outline

The outline of this document follows the sequential order of events and implementations. This way, further "down" implementations or discussions may rely in previous written sections. We start by reviewing the hardware in hand for this work in chapter 2 with respective configurations, presenting how we intend to collect results and energy measurements along side with some initial definitions. Afterwards, Chapter 3 solely focuses on the MMP work, while Chapter 4 investigates the HEVC related algorithm. Both chapters include theoretical reviews over the algorithms alongside with existing implementations and results, after which we propose several implementations showing and discussing the achieved results. While the MMP chapter strives to be CPU-driven, the HEVC chapter later changes to GPU and multi-GPU driven implementations. Both chapters comprise a final conclusion section. Finally, the document body ends with global conclusions and summary of the developed work.

Chapter 2

Computational Environment and Methodology

This chapter exposes the hardware configurations and methodologies used for the experimental result presented in this dissertation. Section 2.1 describes the used equipments while Sections 2.2 and 2.3 define the way results are gathered and other details regarding the presented experimental results of this dissertation. Finally, Section 2.4 provides a small comparison between the equipments based on memory speed and computational performance.

2.1 Equipment and Configurations

The main CPU equipments and associated configurations used throughout this dissertation are listed in Table 2.1. It is important to point out that all the OpenCL enabled setups (servers and laptop) have the *"Intel CPU – OpenCL Runtime 15.1 x64 v5.0.0.57"* software installed to provide OpenCL compute capabilities on the CPU. The version 1.1 of OpenCL was selected so that all builds had the same OpenCL language version. This means that all hardware and software support OpenCL version 1.1.

A quick comparison of the different equipments highlights that Server 1 is our best hardware. Server 1 has at least twice as much RAM as the other systems. Moreover, his RAM is also better distributed in 8 modules of 8 GiB. Even so, Server 2 actually possesses a superior RAM data rate (of 1600 MHz). To better understand the RAM influence on the application performance for 2-CPU computers, Server 2 and 3 have different amounts of RAM modules, while still maintaining the same total amount of RAM. While the servers physical setup provides insight about how small changes may

	Server 1	Server 2	Server 3	Laptop	Jetson	Raspberry
CPU Name	Intel(R) Xeon(R) E5-2630 @ 2.30GHz	Intel(R) Xeon(R) E5-2620 v2 @ 2.10GHz	Intel(R) Xeon(R) E5-2620 v2 @ 2.10GHz	Intel(R) Core(TM) i7-2670QM @ 2.20GHz	NVIDIA "4-Plus-1" ARM Cortex-A15 @ 2.32GHz	ARM Cortex-A @ 7900MHz
CPUs	2	2	2	1	1	1
Cores per CPU	6	6	6	4	4	4
Threads per Core	2	2	2	2	1	1
Cores	0-5, 12-17 6-11, 18-23	0-5, 12-17 6-11, 18-23	0-5, 12-17 6-11, 18-23	0-7	0-4	0-4
Cache L1 (Data/Intruction)	32K/32K	32K/32K	32K/32K	32K/32K	32K/32K	32K/32K
Cache L2	256K	256K	256K	256K	256K	512K (shared)
Cache L3	15360K	15360K	15360K	6144K	(none)	(none)
Architecture	x86_64	x86_64	x86_64	x86_64	armv7l	armv7l
Byte Order	Little Endian	Little Endian	Little Endian	Little Endian	Little Endian	Little Endian
RAM	64Gib	32Gib	32Gib	12Gib	2Gib	927MiB
RAM per Slot	8GiB	8GiB	16GiB	4Gib	2Gib	—
RAM Slots	8	4	2	3	1	—
RAM Type	DIMM DDR3	DIMM DDR3	DIMM DDR3	SODIMM DDR3	DDR3L EMC	—
RAM Data Rate	1333 MHz	1600 MHz	1333 MHz	1333 MHz	933MHz	—
RAM Cycle Time	0,8 ns	0,6 ns	0,8 ns	0,8 ns	—	—
Operating System	Ubuntu 14.04 LTS	Ubuntu 14.04 LTS	Ubuntu 14.04 LTS	Ubuntu 14.04 LTS	Ubuntu 14.04 LTS Linux for Tegra R21.1	Raspbian
Kernel Name	3.13.0-36-generic	3.13.0-39-generic	3.13.0-39-generic	3.13.0-32-generic	3.10.40-g8c4516e	3.18.11-v7+
Kernel Version	#63-Ubuntu SMP	#66-Ubuntu SMP	#66-Ubuntu SMP	#57-Ubuntu SMP	#1 SMP PREEMPT	#781 SMP PREEMPT
Hardware Name	EPIC	Epic2	Epic3	EpicPortable	tegra-ubuntu	epicberry
GCC Version	4.8.2-19ubuntu1	4.8.2-19ubuntu1	4.8.2-19ubuntu1	4.8.4-2ubuntu1	4.8.2-19ubuntu1	4.6.3-14+rpi1
GCC Distribution	Ubuntu	Ubuntu	Ubuntu	Ubuntu	Ubuntu/Linaro	Debian
GCC Thread Model	posix	posix	posix	posix	posix	posix
GCC Target	x86_64 linux-gnu	x86_64 linux-gnu	x86_64 linux-gnu	x86_64 linux-gnu	arm linux-gnueabihf	arm linux-gnueabihf
NVCC Version	6.5.12	6.5.12	6.5.12	6.5.12	6.5.12	(none)
NVCC Driver	340.58	340.58	340.58	346.35	340.58	(none)
CUDA Runtime	6.5	6.5	6.5	6.5	6.5	(none)
CUDA Driver	6.5	6.5	6.5	6.5	6.5	(none)

Table 2.1: CPU related Hardware Equipment Details separated in five relevant aspects: CPU, RAM, OS, GCC, NVCC.

or may not affect an algorithm performance, the software side of the setup is maintained identical. In these experiments, only the OS kernel version slightly differs. The setup changes from the servers to the laptop. While we try to maintain the software as close to the same as possible, in this case, both the OS kernel version and the GCC version differ. At the hardware level, the laptop provides a more common setup than the servers by having a single 8-core CPU, while the servers focus on two 12-core CPUs. It is important to highlight that while the Servers 1 and 2–3 may reach turbo frequencies of 2.8 GHz and 2.6 GHz, respectively, the laptop achieves 3.1 GHz.

To further investigate the effects of different hardware, the *Jetson TK1* [16] and *Raspberry Pi 2 model B* [17] are also included in this dissertation. In this case, the setup changes radically to embedded systems. These are the so called System on a Chip (SoC). Although they are quite dissimilar, with Raspberry Pi targeting pedagogical and very low cost markets and Jetson TK1 bringing high performance computing, namely at the GPU level [18], to embedded systems at affordable prices, both systems provide for energy efficient computing. This is an important topic since the dominant computing ownership cost is related to energy, not only directly consumed by the devices but also for the needs of refrigeration [14]. Moreover, energy eficiency in computing has also become a marketing and ethical trend linked to green computing [15].

The Jetson CPU is classified by NVidia as a "4-PLUS-1" to reflect the ability of the system to enable/disable cores as needed for the interest of power conservation [19]. For this purpose, the CPU has 4 working cores and a low performance/low power usage core. This low performance core, identified as the "PLUS-1", drives the system when the computational demand is low. Whenever the computing load increases, the other cores are activated as needed. Conversely, when the load diminishes, the system scales back, shutting down cores as they are no longer required. Other features of the system to balance the computing power versus the power consumption are the ability to reduce/increase the memory operating frequency and to disable/enable support for I/O devices like USB and/or HDMI ports. In terms of hardware specifications, Jetson TK1 development board has a single CUDA multiprocessor (SMX), as further detailed in Table 2.2 alongside with the lists of available GPUs and their main characteristics. From the table, GPU *GTX 570M* refers to the laptop graphics card.

	GTX 750Ti	GTX Titan Black	Jetson TK1	GTX 680	GTX 570M	GTX 480	AMD R9-290x	AMD HD7970
Clock (MHz)	1085	980	72–852	1058	1150	700	1000	925
Memory (GiB)	2	6	2	2	1.5	1.5	4	3
Memory Width (bits)	128	384	64	256	192	384	512	384
Memory Bandwidth (GB/s)	86.4	336.0	14.5	192.2	72.0	177.4	320	264
Power (watts)	60	250	14	195	–	250	290	250
Compute Units	5	15	1	8	7	15	44	32
Cores/Shaders	640	2880	192	1536	336	480	2816	2048
CUDA Compute Capability	5.0 Maxwell	3.5 Kepler	3.2 Kepler	3.0 Kepler	2.1 Fermi	2.0 Fermi	(none)	(none)

Table 2.2: GPU related Hardware Equipment Details.

The Raspberry is a low cost, low power, single board credit-sized computer, developed by Raspberry Pi Foundation [17]. The Raspberry Pi has attracted a lot of attention, with both models of the first version – model A and model B. A major contributor for its popularity is the low price. Another contributing factor for the success of the Raspberry Pi is its ability to run a specially tailored version of Linux and the many applications it bears under the version 6 of the ARM architecture. The model B of version 2 of the Raspberry Pi, which is the one used in this study, was released in 2015. Model B – the high end model of the Raspberry Pi 2 – has a quad-core 32-bit ARM-Cortex A7 CPU operating at 900 MHz, a Broadcom VideoCore IV GPU and 1 GiB of RAM memory shared between the CPU and the GPU. Both the CPU and the GPU, and some control logic, are hosted in the Broadcom BCM 2836 SoC. The Raspberry Pi 2 maintains the low price base. Beside the doubling of the RAM memory, an important upgrade from the original Raspberry version lies in the CPU which has four cores and thus can be used for effective multithreading. Each CPU core has a 32 KiB instruction cache and a 32 KiB data cache, while a 512 KiB L2 cache is shared with all cores. Additionally, the fact that the CPU implements the version 7 of the

ARM architecture means that Linux distributions available for the ARM v7 can be run on the Raspberry Pi 2. The Raspberry Pi 2 maintains the same GPU of the original Raspberry Pi. The GPU is praised for its capability in decoding video, namely the ability to provide resolution of up to 1080 pixels (full HD) supporting the H.264 standard [20]. However, to the best of our knowledge, no standard parallel programming interfaces like OpenCL is available for the GPU of the Raspberry Pi. Another (minor) reported drawback is that the power consumption has increased from 3 watts for the Raspberry Pi 1 (model B) to 4 watts [21] in the Raspberry Pi 2 (model B).

As a final detail, for this dissertation, both Jetson and the Raspberry are referenced with distinguishable modes. By default, both Jetson and the Rasperry boot with their lowest (power saving) settings. We call these default *"Jetson LowPower"* and *"Raspberry Def"*. After booting, both platforms/systems provide means to alter these settings. Only the high performance settings are used for comparison. Raspberry possesses a turbo mode that we identified as *"Raspberry Turbo"* which boosts the CPU from 700 MHz to 1000 Mhz and SDRAM from 400 MHz to 600 MHz, whereas the Jetson lets us tweak several of its components one by one. Because of this, we came up with 2 modes. One, *"Jetson MaxFrequency"*, in which we only maximize the GPU related features, like boosting its clock rate from 72 MHz to 852 Mhz. And another, *"Jetson MaxPerformance"*, where we simply set the maximum allowed values for all configurations plus: enable all four cores of the CPU; disable the CPU scaling by setting the scaling governator to performance mode; disable I/O devices (e.g., HDMI output); and inform the graphics card that no output is needed (by setting it to blank). None of these tweaks/boosts are overclock related, they are all within the hardware capabilities and no overheat was observed during this study.

2.2 Results Acquisition and Used Images

The next two chapters refer to two different algorithms that have different purposes. All execution time results shown in Chapter 3 are calculated through several executions, i.e., they are the mean ($\overline{x}$) of some amount of executions. Typically, execution time results related to Servers 1, 2 and 3 are calculated from the $\overline{x}$ of 30 executions. Execution times from the laptop system are calculated from 10 executions, while Jetson and Raspberry experiments used 5 and 3 execution, respectively. The lower number of execution/runs is to accommodate the huge execution times. Moreover, there is no need for very large mean calculations since all equipments provide results that are always within certain expected ranges. Tests were performed and all execution time results presented, on average, 0.8% standard deviation. Because these deviations are

insignificant no other reference is made to them in this document. When larger deviation were encountered the results were discarded and the test/benchmark redone. Also for Chapter 3, we used both Figures 2.1a and 2.1b as reference images. But since all the results had always the same properties — e.g. the second image achieved always faster results and always by the same proportions — and no image provided more interesting results over the other, results from Figure 2.1b were discarded thus not presented. Chapter 4 follows the same rules. Initially, we ran 5 executions per experiment, but due to the large execution time – several days per run – we scaled back to a simple execution per run. Here the standard deviations also do not have any meaningful weight. Of course, because the executions are so time consuming, small deviations may mean minutes, nevertheless, percentually, they never exceeded a 1.5% deviation. The experiment results in Chapter 4 were performed on the holoscopic image shown in Figure 2.2.

(a) Lena.pgm

(b) Barbara.pgm

Figure 2.1: MMP test images, 512x512 with 8-bits per pixel (greyscale).

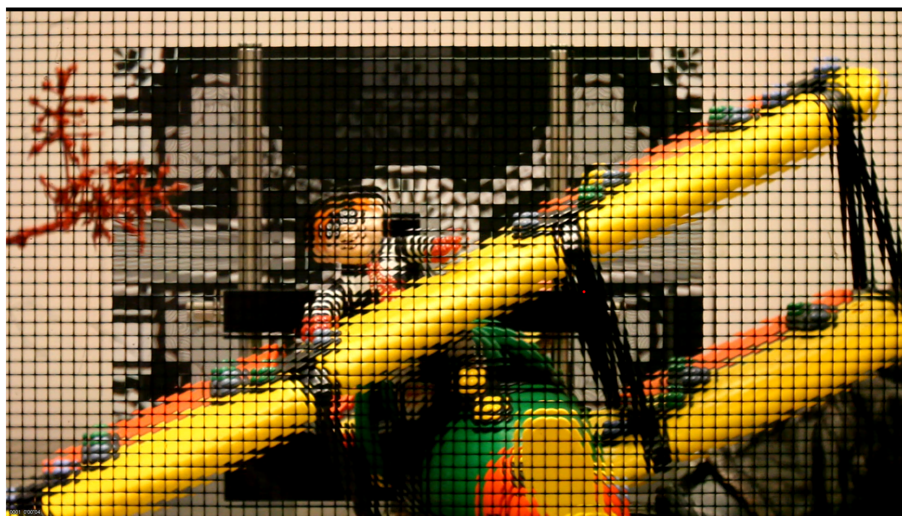


Figure 2.2: HEVC test image, frame 1 of 3D holoscopic test sequence Plane and Toy, size 1920x1088, captured using a 250µm pitch micro-lens array.

For the acquisition of sequential or non-GPU-driven implementation results, no discrete GPU cards were present in the underlying system, except the laptop, the Jetson and Raspberry equipments whose GPU is non-removable.

2.2.1 SpeedUp

As a way to simplify the comparisons made between results of different equipments and/or implementations the *SpeedUp* metric was applied. The *SpeedUp* metric shows the performance gain over a predetermined task between two systems processing the same problem or within the same system, but with different approaches to problem. Here the *SpeedUp* is calculated through:

$$SpeedUp = \frac{Base\ version\ time}{New\ version\ time} \tag{2.1}$$

A speedup of 1 means that the two versions perform the computations in the same amount of time. A speedup in the range $]0;1[$ means that the new version suffered performance degradation. Finally, if the speedup is above 1 the new version is better/faster than the base version.

2.2.2 PSNR and Bitrate

Maintaining the program output when optimizing an algorithm is of utmost importance. In image compression algorithms/software the output image is generally accompanied by a value that tries to describe the quality loss that the image may have suffered. Peak signal-to-noise ratio, abbreviated PSNR, is an engineering term for the ratio between the maximum possible power of a signal and the power of corrupting noise that affects the fidelity of its representation. Because many signals have a very wide dynamic range, PSNR is usually expressed in terms of the logarithmic decibel scale. Basically, PSNR is an approximation to human perception of reconstruction quality – hence a measure of quality. Typical values for the PSNR in lossy image and video compression are between 30 and 50 dB, provided the bit depth is 8 bits, where higher is better. For 16-bit data typical values for the PSNR are between 60 and 80 dB – [22, 23]. PSNR is most easily defined via the mean squared error function. In statistics, the mean squared error (MSE) or mean squared deviation (MSD) of an estimator measures the average of the squares of the errors or deviations, that is, the difference between the estimator and what is estimated. In the absence of noise, two images are identical if the MSE is zero. In this case the PSNR is infinite.

Bitrate is the number of bits that are conveyed or processed per unit of time. The bit rate is quantified using the bits per second unit (symbol: "bit/s"). The non-standard

abbreviation “bps” is often alternatively used – like in the HEVC case.

2.3 Energy Measurements Methodology

Some of the results presented in this dissertation are related with energy consumption measurements. All energy consumptions were recorded with a MCP39F501 Power Monitor Demonstration Board [24] from Microchip. The board is a fully functional single-phase power monitor that does not use any transformers. The MCP39F501 Power Monitor Utility software is used to calibrate and monitor the system, and can be used to create custom calibration setups. The device calculates active power, reactive power, RMS current, RMS voltage, power factor, line frequency and other typical power quantities as defined in the MCP39F501 data sheet [25]. For any technical information, we refer the interested readers to [26].

The record of any of the executions starts with the direct connection of the MCP39F501 board to the laboratory wall outlet and to an USB port of the recording computer. From time to time (months), it is necessary to adjust the board software to correctly provide results over the input current so that accurate values are obtained. As soon as the board is powered and connected to the recording computer the recording loop is started. The recording computer queries the MCP39F501 board at 20 second intervals. The returned values include: current (Amps); voltage (Volts); active power (Watts); reactive power; apparent power; power factor; frequency (Hz); board sensor temperature (bits); and event flags. All the information is stored in a CSV format — plain text tabular data, including the date and time details. For this dissertation, we only considered the active power information. After starting the recording loop, the test equipment is connected to the output end of the MCP39F501 board, booted, and the target algorithm execution is started. In all tests, the equipments were linked to the same keyboard and monitor (with the exception of Jetson in MaxPerformance mode, which lacked the monitor output).

Because large numbers of records were expected, a separate log file was maintained with some relevant events (along side the date/time details). These include the clock time at which a given execution started or when the equipment being tested ended it OS boot.

After the target execution ends, the resulting data files were manually parsed to provide meaningful information. These include average watts consumed in the different

recording stages (connection, boot, OS login and preparation, algorithm execution, etc.). Between executions it was always included the bash "sleep 20s" command to allow the system to return to a stable power consumption and to separate the results in the CSV file.

For more accurate results, tests were performed to assess whether it would be better to execute several times a given target algorithm and work with the mean recorded power of several executions instead of just one. We concluded that it did not provide any improvement, besides a better estimate of the overall energy consumption a given algorithm may require for a given system. Therefore, presented energy results in this dissertation refer to a single execution.

2.4 Base Bandwidth Speeds

In order to quickly understand how our different hardware performs, we initially run several Phoronix benchmarks. The Phoronix Test Suite [27] is a testing and benchmarking platform available for several systems. We targeted the equipments RAM and CPU speeds in both bandwidth and instruction throughput. Table 2.3 provides a first performance comparison between the hardware/systems previously detailed in Table 2.1. Perceivable by the table greyscaled gradient, it is shown that Server 1 provides the best performance throughout several CPU operations and RAM memory accesses. The slightly different RAM architecture between Server 2 and Server 3 – 4x8GiB RAM per CPU for Server 2 vs 2x16GiB for Server 3 – becomes relevant here. In this benchmark, Server 3 has only mostly 3/5 of Server 2 capabilities, while the laptop only achieves 2/5 of Server 2 performance. As expected, SoC systems – Jetson and Raspeberry – have even worse performance with 3/10 and 2/25, respectively.

To complement the memory bandwidth measurements we also included the widely used Lempel-Ziv-Markov (LZMA) chain compression algorithm. The first idea of LZMA implementation was found in [28]. It was first used in the *7z* format of the *7-Zip* archiver [29]. This compression algorithm has proved to be effective in any byte stream compression for reliable lossless data compression. In order to benchmark our systems we resorted to the *p7zip 9.20.1* tool [30] which already provides the benchmarking runtime. The LZMA benchmark shows a rating in MIPS (million instructions per second). The compression speed strongly depends on memory (RAM) latency and Data Cache size/speed. While the decompression speed strongly depends on CPU integer operations. The most important performance factors for the test are: branch

Benchmark	Server 1	Server 2	Server 3	Laptop	Jetson LP	Jetson MP	Raspberry Def	Raspberry Turbo
Int Add	22193.26	18029.11	11316.82	7253.12	4893.18	4948.66	951.30	948.01
Int Copy	22636.24	19444.61	11912.15	7511.04	5689.25	5737.97	1640.12	1645.22
Int Scale	22886.28	19505.67	11899.76	7619.96	6019.64	5987.21	1159.31	1202.76
Int Triad	21967.13	18080.05	11305.06	7279.44	4898.90	4897.00	942.98	989.95
Int Average	22675.05	18774.61	11610.94	7472.66	5395.36	5403.82	1154.93	1208.79
FP Add	19962.39	17773.46	13224.39	8426.04	5660.94	5659.40	1094.99	1157.55
FP Copy	15017.08	14784.75	11443.95	7587.33	6787.17	6780.84	1552.48	1651.85
FP Scale	15111.85	14766.78	11415.36	7510.22	6460.68	6449.32	1207.87	1259.45
FP Triad	19867.79	17710.24	13183.40	8425.50	5649.62	5653.54	892.80	931.77
FP Average	17569.45	16263.09	12310.63	8007.93	6140.20	6140.44	1236.67	1250.75

Table 2.3: RamSpeed Test Results for Integer and Floating Point operations from the Phoronix Test Suite 5.8.1 in MiB/s (more is better).

misprediction penalty (the length of pipeline) and the latencies of 32-bit instructions. The decompression test has very high number of unpredictable branches. Hence, the LZMA benchmark can accurately categorize our different systems.

Table 2.4 shows the resulting MIPS values for each system over the compression 2.4a and 2.4b algorithms. Analysis of the results yields similar conclusions to the one found for the Phoronix Test Suite (Table 2.3). To assess the systems we run the *p7zip* benchmark with a varying number of threads. With this information we can review how each CPU core behaves initially – with 1 thread – and how adding threads affects the results – in terms of MIPS. For the decompression algorithm, both Server 1 and SoC devices seem to take a performance hit.

Additionally, to provide some understanding over the laptop system – that can sometimes be more efficient than other hardware – we further reviewed its memory scheme. The Parallel Memory Bandwidth Benchmark (PMBW) [31] is a suite that measures bandwidth capabilities of a multi-core computer. This is an important test because more cores result in the floating point performance increasing in a linear fashion. However, if the memory bandwidth is not capable of transmitting the data fast enough, processors will stall. Indeed, unlike floating point units, memory bandwidth does not scale with the number of cores running in parallel. The PMBW code was developed in directly assembler language meaning inherent compiler issues such as optimization flags will not occur. The code uses two general synthetic access patterns: sequential scanning and pure random access. The benchmark outputs an enormous amount of data, thus most of it was filtered out [32]. PMBW is already an accepted tool that provides large insight over the expected hardware performance [33, 34].

Table 2.5 shows a summary of the measured values of the PMBW benchmark for the

7zip Threads	Server 1	Server 2	Server 3	Laptop	Jetson Low Power	Jetson Performance	Raspberry Default
1	3460	3433	3379	3737	1602	1611	270
4	10026	9644	9515	8901	4565	4579	744
6	13992	13966	13333	11501	5099	5145	883
12	24086	23713	22826	15706	**5146**	**5157**	**997**
24	38090	37631	33939	15513	5142	5139	985
48	**45646**	**42255**	**36805**	**15770**	5085	5094	949

(a) LZMA Compressing Rating (MIPS)

7zip Threads	Server 1	Server 2	Server 3	Laptop	Jetson Low Power	Jetson Performance	Raspberry Default
1	2928	2797	2795	3206	2013	2030	401
4	11100	9916	9070	9849	7799	**7926**	**1590**
6	15561	13932	13427	12327	7282	7522	1553
12	26152	23510	23247	14614	**7800**	7814	1581
24	**43734**	39006	40761	**14624**	7761	7754	1578
48	42962	**40337**	**40840**	14518	7717	7723	1560

(b) LZMA Decompressing Rating (MIPS)

Table 2.4: LZMA compressing and decompressing benchmark results from the p7zip 9.20.1 in MIPS (more is better) with default dictionary size. The best result for each system is marked in bold.

laptop system and Server 1. The table is divided between *Access Time* and *Bandwidth* results, in nanoseconds and Gib per second, respectively. In terms of bandwidth, the laptop system presents a clear performance advantage in most of the scenarios, while keeping up in the remaining ones – these scenarios can be reviewed in at [31]. For a single thread the access time results confirm that the laptop provides data faster on average. The laptop has also higher reading performance, while Server 1 has higher storing/writing performance. But these results focus only over the utilization of 1 thread. The addition of other working threads change the performances. Table 2.6 shows the results for executions with different amounts of threads in a scenario where the laptop previously had more performance over Server 1 (with only 1 thread). These results show that while the laptop achieves better performance with 1 thread, Server 1 performs faster when multiple threads are used. While the laptop halts with average speeds of 0.7, Server 1 hardware continues to decrease its memory latency down to 0.17ns with 9 threads. The complex mesh of hardware characteristics of each device makes it challenging to accurately rate an equipment over the wide variety of appliances it may be a part of. Because of this, the laptop system may sometimes provide better results, since it sometimes performs better than other systems, making factors like memory access time, block/data sizes, buffers access patterns, etc., very important performance factors.

Benchmark	Access Time		Bandwidth	
	Server 1	Laptop	Server 1	Laptop
ScanWrite256PtrSimpleLoop	4.1268	5.5326	41.18	32.31
ScanWrite256PtrUnrollLoop	4.7866	5.5329	41.18	33.12
ScanWrite128PtrSimpleLoop	2.0342	2.7651	39.50	43.40
ScanWrite128PtrUnrollLoop	2.3999	2.7591	41.31	45.64
ScanWrite64PtrSimpleLoop	1.1481	1.3962	20.44	16.49
ScanWrite64PtrUnrollLoop	1.0492	1.3810	20.73	16.47
ScanWrite32PtrSimpleLoop	0.5516	0.6898	10.31	11.38
ScanWrite32PtrUnrollLoop	0.5609	0.6892	10.38	11.47
ScanRead256PtrSimpleLoop	2.9367	2.9475	75.18	82.98
ScanRead256PtrUnrollLoop	2.9072	2.8877	82.47	89.26
ScanRead128PtrSimpleLoop	1.8558	1.5183	39.47	43.61
ScanRead128PtrUnrollLoop	1.6991	1.5020	80.98	89.40
ScanRead64PtrSimpleLoop	0.8346	0.8075	20.46	21.49
ScanRead64PtrUnrollLoop	0.9308	0.7898	41.13	44.99
ScanRead32PtrSimpleLoop	0.6071	0.4478	10.25	11.29
ScanRead32PtrUnrollLoop	0.5821	0.4434	20.67	22.79
ScanWrite64IndexSimpleLoop	0.9577	1.5099	20.45	22.54
ScanWrite64IndexUnrollLoop	1.1143	1.3824	20.70	22.86
ScanRead64IndexSimpleLoop	0.8700	0.8064	20.46	22.57
ScanRead64IndexUnrollLoop	1.0046	0.8102	41.04	43.70
PermRead64SimpleLoop	95.3199	73.9657	5.19	5.70
PermRead64UnrollLoop	94.2605	80.0693	5.19	4.11

Table 2.5: Access Time (in ns) and Bandwidth Results (in Gib/s) for all Benchmark Modes filtered from PMBW 0.6.2 with 1 thread over Server 1 and the Laptop (more Bandwidth is better and less Access Time is better, highlighted with grey).

Threads	Access Time	
	Server 1	Laptop
1	0.9308	0.7898
2	0.4090	0.6891
3	0.3375	0.6859
4	0.2623	0.6854
5	0.2295	0.6915
6	0.2411	0.6881
7	0.1915	0.6944
8	0.1823	0.6971
9	0.1737	0.7375
10	0.1911	0.6934

Table 2.6: Access Time Results for ScanRead64PtrUnrollLoop Benchmark Mode filtered from PMBW 0.6.2 with 1–10 threads over Server 1 and the Laptop in nanoseconds (less is better, highlighted with grey).

Chapter 3

MMP – Multi-CPU, Many-Core, Multi-Thread

In this chapter we present a small review of the selected algorithm (Section 3.1) and existing implementations (Section 3.2). With these building blocks, we further propose and implement a CPU multi-thread/multi-CPU version resorting to OpenMP (Section 3.3) and migrate an existing GPU implementation to CPU for further comparison (Section 3.4).

OpenMP was selected for this project due to its maturity and relatively high level abstraction. Other strategies were also analysed (e.g. MPI, compared in detail in [35]) but discarded since OpenMP has a lower learning curve [36, 37, 38] and is highly suited for multiprocessors multicore CPUs. A more low level approach, namely with POSIX Threads (*PThreads*), was also considered. However, it soon became apparent that many person-hours would be needed to achieve meaningful performance. Additionally, OpenMP proved to be suited for generating correct and structural parallel code as well as being capable of facilitating fine-grain improvements [39]. POSIX Threads implementations were therefore cast aside.

The migrated GPU implementation is based in the OpenCL API. Because OpenCL provides a vendor-neutral environment, i.e. enables implementations to be run in a large variety of components, OpenCL can thus be regarded as a "write once, run anywhere" (WORA) framework. However this does not implicitly mean that a source code which is optimal in a GPU environment will also deliver optimal performance for other components, namely CPUs. In this work, we demise the process of converting a GPU OpenCL optimized code for CPUs.

Finally, Section 3.5 summarizes this chapter.

3.1 Multidimensional Multiscale Parser

The Multidimensional Multiscale Parser (MMP) is a pattern-matching-based compression algorithm. Although it can compress any type of content [40, 41, 42], it is more appropriate for multimedia images [43, 44], where it lossy compression mode can achieve good compression ratio and maintain good image quality [1]. While compressing, MMP dynamically builds a dictionary of patterns that it uses for approximating the content of the input image [44]. Specifically, the input content is split in blocks, each having 16x16 pixels, and each input block is processed sequentially. For every input block, MMP assesses the patterns that exist in its dictionary in order to find the one that is the closest to the original one [44], that is, the one that yields the lowest overall distortion. For this purpose, MMP needs to compute the Lagrangian cost J between the input block and all blocks existing in the dictionary. The Lagrangian cost is given by the equation $J = D + \lambda * R$ [45], where D measures the distortion between the original block and the candidate block, R represents the number of bits needed to encode the approximation and λ is a numerical configuration parameter, which remains constant along the execution of MMP. The λ parameter steers the algorithm towards more compression quality (low λ, which emphasizes the importance of low distortion over bit rate) or lower bit rate (high λ). MMP also explores patterns that are not yet in the dictionary, computing their Lagrangian cost J. This is done with the goal of finding new patterns that might be more suited for the input being processed, i.e., yield lowest Lagrangian cost. For these possible patterns to be, MMP uses a multiscale approach, which enables the approximation of image blocks with different sizes. Specifically, while the input image is processed in basic blocks of 16x16 pixels, all possible subscale blocks (8x4, 16x2, 1x1, etc.) are tested to find the best possible matches. When the search is exhausted, MMP selects the best block or set of sub-blocks, that is, the one(s) that delivers the lowest overall Lagrangian cost J. This exhaustive search is the main cause for the high computational complexity of MMP.

Having found a set of new blocks/sub-blocks, MMP proceeds to update the dictionary of patterns. However, first it checks whether the newly generated patterns already exist in the dictionary. To this extent, MMP lookups through the dictionary searching for blocks that might be identical or approximate (within a given radius) to the newly proposed blocks. If no close block to the new ones are found, MMP adds the new blocks/sub-blocks to the dictionary. The computing processes – block and sub-blocks search and dictionary update – is repeated for every single block of the input image. To facilitated dictionary searches, the dictionary is composed of sub dictionaries, each one for a subscale block size.

3.2 Previous Versions

Previously to this work, they were already three existing implementations of MMP: a sequential (CPU-driven) and two GPU-diven implementations. Although implemented with different languages/APIs, the two GPU versions are very similar, only differing on their API calls, i.e., their MMP logics are the same.

The denoted sequential implementation refer to the direct implementation of the MMP logic/algorithm with one CPU thread. The MMP algorithm is composed of several subparts/stages. The relevant stages for this dissertation are: (*i*) the initialization, (*ii*) the distortion calculation and tree segmentation, (*iii*) the dictionary actualization and (*iv*) the intra prediction modes. All these stages are presented in detail in [3, 44].

The (*i*) initialization and (*iv*) prediction modes parts become relevant in the later Section 3.3 and the (*ii*) distortion calculation and (*iii*) dictionary update for the GPU-driven implementations above and in Section 3.4.

The next subsections review the GPU-driven implementation (subsection 3.2.1) and present initial performance results obtained by the three existing implementations (subsection 3.2.2).

3.2.1 MMP-CUDA and MMP-OpenCL-GPU

The OpenCL and CUDA based versions [3, 2] are already optimized for GPUs. They consist on the migration of the sequential MMP algorithm in such way that it becomes possible to partially process all the sub blocks of a given larger block at once and later reduce the calculations in the same way that the original algorithm would had done it. The OpenCL-GPU implementation is a migration from CUDA whose conversion is described in [3]. Both versions use 4 kernels. Two major kernels to calculate the Lagrangian cost for each element of the dictionary of each subscale block size of the major block being processed (by the kernel named *optimize_block*) and to reduce the calculated values by selecting the less costing element of each subscale (by the kernel named *j_reducter*). After this process (in each image block), the natural order of things is to compare the findings and update the dictionary but since the GPU (CUDA/OpenCL targeted device) is doing part of the math around the dictionary, there is also the need to maintain its awareness on new selected findings between each

processed image block. As a solution, two minor kernels review the previous selected findings by comparing them to the existing ones on the dictionary (with the kernel *compare_ blocks*) to determine if they are to be copied to the dictionary as new needed elements (with the kernel *update_ dic*, so that the new dictionary is not copied as a whole from the host CPU after he also incorporates the new need elements). This last kernel is the one to actually update the device internal/mirror dictionary.

As in any optimized GPU implementation, important details must be taken care of in order to adieve the highest performance from the underlying hardware. In these implementations, the authors [3, 2] had the need to correctly balance the work between the different GPU workers/threads so that no group of workers ran out of work (since the dictionary is actually divided in several different sized subscale dictionaries). Not correctly balancing the work leads to branch diversion [46] which is a cause of execution delays. This is due to the fact that in-warp threads only re-converge after all divergent execution paths are completed [47]). Through the process of balancing the workload it is often necessary to flatten the access pattern to the memory data [48, 49]. This was also the case for MMP. Given the overall behaviour of the algorithm access patterns, in order to maintain the data locality [50], the dictionary data was laid out in such way that each individual worker did not need to access separated memory regions to obtain all data needed, nor it would generate misaligned access patterns within its group/warp. The goal is to provide a proper coalesced memory access, managing the issues data stride relate to. All of this simplifies other memory related optimizations, like the use of sequential addressing [51] for results reduction. As a last detail, the *optimize_ block* kernel had the need for different local memory sizes between executions which let to less appropriate occupancy levels. To overcome this obstacle, since this particular local buffer directly depended on the number of possible prediction modes available for the given major block being processed, the kernel code was replicated 10 times, in which only the line declaring the local buffer size differed, ranging from $numberOfKernelThreads$ to $numberOfKernelThreads * 10$.

More detail about the CUDA and OpenCL implementations can be found in [3] and [2].

3.2.2 Base Reference Results

As a reference to the rest of this chapter, Tables 3.1 to 3.3 show the execution times registered for the pre-existing implementations on several hardware systems. For simplicity, the results are focused and compared (through speedup) with Server 2. Table 3.1 indicates the executions times encountered for the sequential implementation. The fastest execution times are recorded with the laptop system, followed by the servers. Jetson and Raspberry are without any doubt slower, yielding 2× and 12× times less performance, respectively. Regarding the CUDA implementation (Table 3.2), laptop 570M is still the fastest performer, unlike Jetson that only overtakes the sequential version when exiting Low Power mode. At its maximum performance, Jetson still only achieves around half the performance of the other systems. Table 3.3 shows the OpenCL-GPU results for NVidia and AMD GPUs. It is easily noticeable that NVidia, even when not using its primary language (CUDA), still performs better than AMD. From CUDA to OpenCL-GPU the execution time performance is degraded by around 40%.

	Time	SpeedUp
Server 1	2097.0340	1.03
Server 2	2157.7957	1.00
Server 3	2139.1510	1.01
Laptop	1836.5860	1.17
Jetson LP	5622.4783	0.38
Jetson MP	5558.7413	0.39
Raspberry Def	25734.4300	0.08
Raspberry Turbo	25415.6100	0.08

Table 3.1: Execution time results (in seconds) for the MMP-Sequential encoder for the Lena image with $\lambda = 10$ and dictionary size of 1024.

	Time	SpeedUp
Sequential	2157.7957	1.00
GTX Titan	293.4368	7.35
GTX 680	299.5714	7.20
GTX 750Ti	298.7366	7.22
GTX 570M	287.5720	7.50
GTX 480	294.1414	7.34
Jetson LP	2638.3474	0.82
Jetson MF	1017.8993	2.12
Jetson MP	646.4966	3.34

Table 3.2: Execution time results (in seconds) for the MMP CUDA encoder for the Lena image with $\lambda = 10$ and dictionary size of 1024 for server 2.

	Time	SpeedUp
Sequential	2157.796	1.00
GTX Titan	512.511	4.21
GTX 680	504.957	4.27
R9-290X	991.304	2.18
HD 7970	918.561	2.35

Table 3.3: Execution time results (in seconds) for the MMP OpenCL-GPU encoder for the Lena image with $\lambda = 10$ and dictionary size of 1024 for the servers 1 and 2 with NVidia and AMD GPUs — values from [3].

3.3 OpenMP

Open Multi-Processing is an Application Programming Interface (API) with a fork-join model. At its core level, OpenMP is a set of directives and callable library routines that are used to specify parallel computation in a shared memory style for C, C++, and Fortran [37, 52, 53] that influence run-time behaviour with the addition of environment variables [37]. The API is currently widely implemented by various vendors and open source communities [54]. OpenMP programming model can be used in a non-fragmented manner in contrast to other communication libraries and PGAS (Partitioned Global Address Space) languages [55].

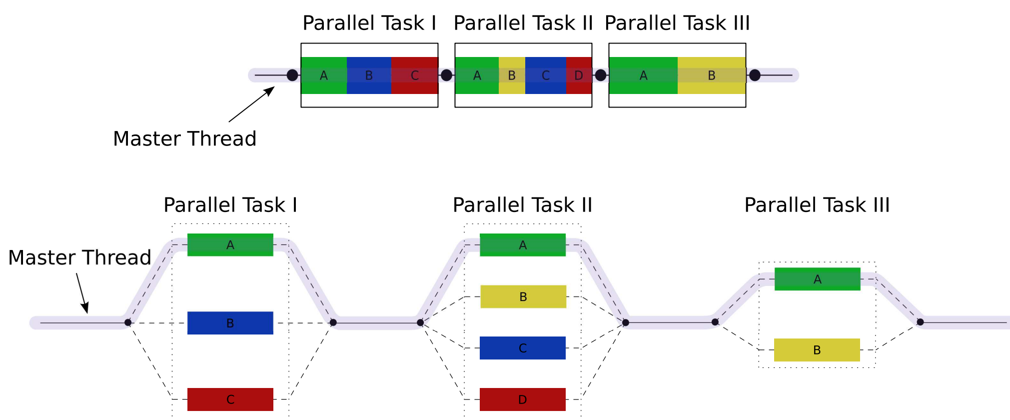


Figure 3.1: An illustration of the fork-join paradigm, in which three regions of the program permit parallel execution of the variously colored blocks. Sequential execution is displayed on the top, while its equivalent fork-join execution is at the bottom. — adapted from [56]

OpenMP is typically used after locating instances of potential parallelism within a sequential program. The section of code that is meant to run in parallel is marked with preprocessor directives. After the identification of loops in which parallel computation should occur, the OpenMP compiler and runtime implement its parallelism using a set of cooperating threads. By default, each thread executes a given parallelized section of code independently. The fork-join execution model (Figure 3.1) makes it easy to get loop-level parallelism out of a sequential program [52], with or without recursive forks, until a certain task granularity is reached. Recursively nested fork-joins can result in a parallel version of the divide and conquer paradigm [57, 58] (with e.g. quicksort [59]) that can reduce the overhead of task creation [60].

In contrast, querying the identity of a thread within a parallel region and taking actions based on that identity is also possible. This enables OpenMP to also be used in a fragmented manner, although it is more often used in a global-view manner, thus letting the compiler and runtime manage the thread-level details [53].

The idea behind OpenMP is to provide a predominantly open method of converting a sequential program into a multi-core, several-processor, application without any major code changes to the original source. All that is needed is to insert a few directives (or pragmas) into the pre-existing source code and enable the OpenMP compiler option [54] (e.g., "-fopenmp" in the GCC compiler). This triggers the compiler into checking for OpenMP directives and use the information they possess to transform the program for multithread execution, making the OpenMP implementation job to implicitly create the code necessary to run on different cores and/or processors. If OpenMP support is not ensured at compilation time, then the directives are ignored and the original single-threaded sequential code is generated. Only sections of code embraced by the directives are potentially translated. This is, if a few rules are observed, e.g. no data dependency between loop iterations. The rest of the program left untouched.

The ability to inject parallelism incrementally into a sequential program by the simple addition of directives is considered the OpenMP greatest strength and productivity gain in addition to being a portable alternative to message passing.

With no explicit configuration, OpenMP generates a maximum number of threads equal to the number of present CPU cores (including hyper-threading [61, 62]). Obviously, limitations to the number of threads generated also include code parallelization limits (e.g. a for-loop with a maximum of 5 cycles will only have 5 threads) and environment context, e.g., if some CPU cores are currently too busy with other computations, there is no point on adding more threads into their schedule. OpenMP

language extensions are divided in 5 constructs: thread creation, workload distribution (or sharing), data-environment management, thread synchronization, and user-level runtime routines and environment variables. Useful OpenMP technical and developer driven information can be found in [53].

To summarize and conclude, OpenMP was designed to enable the creation of, or transformation to, programs that are able to exploit the features of parallel computers where memory is shared. It enables the construct of portable parallel programs by supporting the developer at a high level with an approach that allows the incremental insertion of its constructs. Additionally, it also allows a conceivable low-level programming style, where the programmer explicitly assigns work to individual threads.

3.3.1 Profiling MMP-Sequential

Prior to the adaptation of MMP to multithread, the sequential version of MMP was profiled. Besides the opportunity to better understanding the original code mechanics and flow, profiling provides a means to rapidly spot more critically demanding sections. These sections are generally the most rewording candidates to be rearranged in a parallel fashion simply because they frequently point towards the code that is the more time consuming, making simple optimizations more noticeable [63]. Some experimentation was done with so called 'Automatic Parallelization Tools'. However, none of the tested tools (ROSE[1], PLuTo[2] [64], CETUS[3], iPat/OMP[4] [65]) provided functional code, nor any speedup.

The analysis of a sequential program performance often relies on profilers that provide statistical information about how much time is spent in different regions of code [66, 67]. Most active, or hot, regions are then given special attention. Tools like the open source [68] or the commercial Intel VTune [69] analyze programs at the function level. This may or may not directly include the creation of call-graph profiles that show the structure of the application at the functional level, including parent-child relationships and time spent in each function. To this extent, performance analysis tools intended to analyze sequential code are very mature. But this focus does not translate directly to tasks like extracting thread-level parallelism from applications. Although Intel VTune does lean in that direction since it is able to access Intel special

[1]ROSE: `http://rosecompiler.org/`
[2]PLuTo: `http://pluto-compiler.sourceforge.net/`
[3]CETUS: `http://cetus.ecn.purdue.edu/`
[4]iPat/OMP: `http://ipat.sourceforge.net/`

CPU registers/counters, it is a commercial product, thus out of the scope of this work.

A simplest way of profiling can be done by changing how every function in the program is compiled. This is done in such way that when they are called, it is possible to stash information about where the call came from. From here, is it possible to figure out what function called, count how many times a call happen, etc. This is actually one of the things achieved when the application is compiled with GCC with the "-pg" option does.

When trying to parallelize code, an intuitive first step might be to find which loops are doing most of the work. However, even deciding which loops to target (if any) can be very time consuming. Again, many free tools do not adequately capture the context, from a loop perspective, necessary to provide information about the consumed time. Some however, like LoopProf (introduced in [70]), depend on special instrumentations but then again do not leave guess work to the programmer like a call-graph profile leaves when trying to identify loops that are good candidates for parallelization. A somewhat extensive appreciation over several parallelization tools can be found in [71]. But since the ultimate goal is to use OpenMP, in the end, using this sort of tools does not provide much insight when balanced with the time spent reviewing different tools. Therefore, a more traditional/manual profiling was used for this work.

On a traditional side, it is common to see call-graph charts as a starting point for manual detection and determination of possible optimization-deficient sections. Like most applications, MMP spends a large portion of the execution time in small portions of code, typically inner loops. Tools such as GProf [72], OProfile [73], Intel VTune or DCPI [66] naturally identify such inner loops since they are the most frequently executed regions of code, but they do not focus on identifying loops themselves, thus failing to communicate information about the overall structure of loops in a program.

As stated and explained in [74], GProf quickly became, and still is, the tool of choice for many developers. Being based on the usage of the operating system interrupts, it is only necessary to compile the source code, in the case of GCC, with "-pg" option. Other considerations might also be important for a more loyal output when directly comparing with source code, e.g. not using inline function optimizations or even the "-O" options family (again, in GCC). When executing the application, the sampled data is saved (typically in a file named "gmon.out") just before the program exits. This is the data which is analysed by GProf. Several data files from different execution runs can be combined to a single file. Provided with this data and the source application, a textual, human readable, output is then generated.

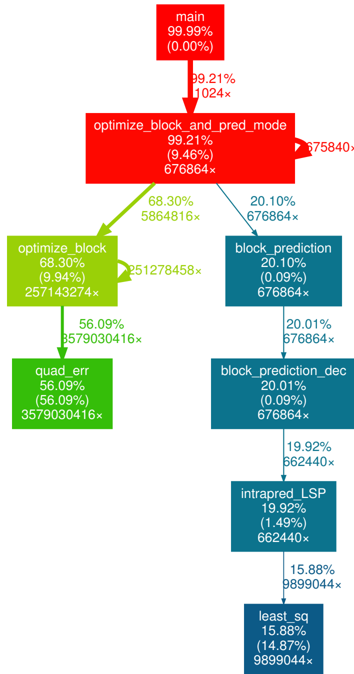


Figure 3.2: Partial call-graph for the execution of the MMP-Sequential encoder on the Lena image with $\lambda = 10$ and dictionary size of 1024.

In conjunction with KProf [75], the reported information can be viewed in an interactive manner. This instantaneously assists on several different visualizations of the same report, including a call-tree representation which relies on Graphviz [76]. However, the direct usage of GraphViz enables more creative visualizations. Figure 3.2 is a visualization from [77] where nodes/functions that occupied less than 5% of the execution time were left out (meaning 98 nodes). The colour scheme is present to aid with hot-spot localization (the gradient is based on the node execution time percentage). Red nodes occupy almost all the execution time, green and later blue nodes gradually consume less execution time percentage.

In parallel with GProf, mostly used as a source code documentation generator tool, Doxygen [78, 79] also provided a substantial understanding over the MMP sequential algorithm through similar grounds. Doxygen configuration with the tags "CALL_GRAPH", "CALLER_GRAPH" and "DOT_MULTI_TARGETS" was particularly relevant. The significant amount of header comments in most MMP functions and pre-generated call-graphs for and from these same functions, all within a web-based GUI, allowed for more efficient navigation and function flow/connectivity visualization throughout the source code files.

From Figure 3.2 it is evident that further analysis over *quad_ err* function is mandatory since it consumes 56% of the whole execution time. But GProf textual report shows

that in fact each individual call to *quad_err* only takes on average 645.5 nanoseconds. This is actually a reasonable amount of time given the amount of calculus this function does when calculating the quadratic error between a given block of the algorithm input image and a block from the dictionary. This concludes that it makes more sense to optimize the way this function is used or the memory mappings in place, making the next best candidate functions the *optimize_block* or the *optimize_block_and_pred_mode*.

On the other side of the spectrum, the *least_sq* function draws almost 15% of the overall execution. This is a small fraction but still representing 612 seconds of the overall execution time, 61.8 microseconds per call, which does not seem much for a least-square solving function. Similarly to the *quad_err* function, the *least_sq* performs a tiny amount of work per call, over millions of calls. Further review of the node functions called between *least_sq* and *optimize_block_and_pred_mode* shows that these nodes do not present any relevance to the parallelization problem in terms of memory usage (since they are very self-contained) or in time consumption. This function reimposes the appropriateness of optimizing the *optimize_block_and_pred_mode* function.

The function *optimize_block_and_pred_mode* is where several independent prediction modes are iterated given an image block provided by its caller, the *main* function. Since computations over each major image block depend on the previously iterated blocks, parallelization over several blocks from the *main* without major reductions and overheads seems impossible. But not over *optimize_block_and_pred_mode*, where several prediction modes loop within each block, making it the perfect starting point for the OpenMP implementation.

Probe sampling can be error prone, although very minimal since it measures statistical approximations. Even so, GProf cannot measure time spent in kernel mode (system calls, CPU stalls, I/O interrupts), only user-space time. Other tools like Perf [80] and OProfile were applied to compensate and look for prime evidence of memory related stalls or instruction-level bottlenecks so that after the OpenMP implementation comparison could be made. This verification actually exposed some unexpected stalls referred in Section 3.3.3.

3.3.2 Implementation

As previously observed, the *optimize_block_and_pred_mode* node from Figure 3.2 provides the best opportunity for a direct parallelization o the code with OpenMP constructs. Some modifications and refactoring were made to the MMP source code in order to make it more suited for OpenMP paralelization. Along the way, other variants more OpenMP friendly were experimented, but failed to deliver the same results of the sequential version and thus were discarded. At the end, the selected implementation was both the simplest and the fastest.

Because each separate thread needs its own set of variables some existing code optimizations for the sequential version needed to be undone. OpenMP constructs allows the programmer to tag which variables need to be 1) replicated and 2) which are to be shared across the different threads. The main problem is that 1) the replication process for large variables repeatedly consume precious execution time and 2) shared variables cannot have memory allocated within the parallel region, since it would generate multiple allocations or imply parallelization/barrier stalls. For these reasons, some variables were manually replicated or extended and others pre-created. For example, case 1), *node_level* variable is used in each one of the prediction modes. Because each loop independently uses it, the variable is cleared of its contents between iterations. This variable is used to avoid redundancy in the tree-like calculations throughout the different block sized partitions. By simply informing OpenMP that this variable is to be replicated (thus made private) to each thread, OpenMP would allocate its space each time for each prediction mode. Going back to Figure 3.2, this would mean that 676.864 allocations (times the number of prediction modes) were additionally made throughout the program execution. To overcome this issue, variables like this one had their allocated space extended by a multiple of the number of prediction modes. By doing so is possible to assign different pointers (to the pre-allocated memory region) to different threads, eliminating OpenMP need to reallocate everything every time. As for case 2), in the same conditions as *node_level*, variables like *block_orig_level* and *pred_orig_level* had their allocations replaced with previously created arrays with the maximum expected sizes already allocated for each partition level. Like in case 1), pointers to different memory regions of the arrays are passed to each individual thread.

After solving the memory related problems, OpenMP-based parallelization is straight forward. Indeed, since the sequential algorithm was already separated into (almost) modular functions, the typing of the source code with OpenMP directives was easy. The directive conjunction "*omp parallel for*" is the most appropriate for this situation

since the prediction modes are iterated with an actual for-loop. This directive is a combination of two separate directives, "*omp parallel*" and "*omp for*". The first directive informs OpenMP of the start of a parallel region. The second lets OpenMP know that we want to parallelize each for-loop iteration. Because the parallelization is over independent iterations which use separate variables / memory spaces, the task scheduling scheme is not particularly relevant [81]. Experimentations were performed with no particular pattern yielding better results than the default behaviour.

3.3.3 Results

Although the parallelization targets the 10 prediction modes of MMP, and thus has a potencial speed gain of 10×, the best achieved speedups are lower. Table 3.4 summarizes the best attained results for each experimental hardware setup. Due to the laptop different hardware layout and environment, code parallelization does not reach the same speedup levels as the remaining computer servers because the sequential version already achieves a superior performance. Since the server computers have more CPU cores than the number of threads actually needed in the implementation, no immediate side effects were observed. Contrarily, the Jetson board only has 4 cores. Figure 3.3 exposes the Jetson performance for different thread numbers/arrangements. OpenMP transparently makes the threads *i*) work over other iterations in the case of thread shortage or *ii*) do nothing if all iterations are already assigned to threads (with minor overheads). From the figure, case *i* correlates to numbers of threads smaller than 10 and case *ii* to numbers larger than 10.

	Sequential	OpenMP	SpeedUp
Server 1	2097.0340	701.3492	2.99
Server 2	2157.7957	719.2870	3.00
Server 3	2139.1510	801.3874	2.67
Laptop	1836.5860	922.2495	1.99
Jetson LP	5622.4783	2539.5830	2.21
Jetson MP	5558.7413	1888.2443	2.94
Raspberry Def	25734.4300	11185.5310	2.30
Raspberry Turbo	25415.6100	11177.9740	2.27

Table 3.4: Execution time results (in seconds) for MMP-OpenMP encoder (Lena image, $\lambda = 10$ and dictionary size of 1024, and achieved speedups).

Findings showed that setting OpenMP with the expected number of actual threads/ iterations improves the performance even if the number of generated threads is larger than the number of cores of the underlying system. Because the Low Power mode of Jetson tries so much to save energy, running OpenMP with 3 threads generates

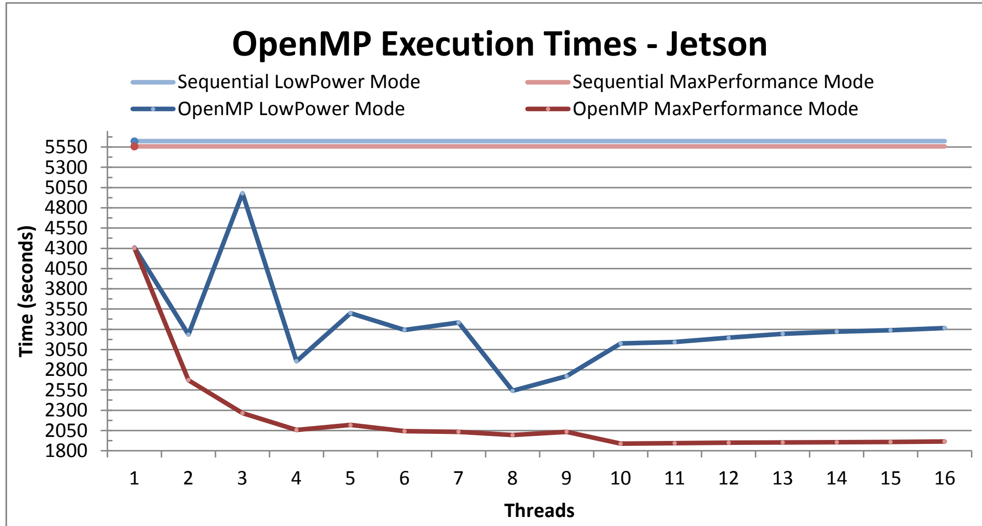


Figure 3.3: OpenMP thread variation execution time results for Jetson (Lena image, $\lambda = 10$ and dictionary size of 1024).

overhead-concurrency within the one available core. Since the 10 prediction modes are not divisible by 3 threads, one of the threads has to execute the last prediction mode, most of the times, not in parallel. Over the hundreds of iterations that Lena requires, performance is degraded. In between all the power related managements (to try and consume as little as possible) the execution of the third thread is also stalled several times for context switches and other computations. Executing with 4 threads actually provides a better balance, because it eventually forces a second core to be always on. When one thread becomes idle due to, for instance, memory related stalls, another one can take its place and make use of the core pipeline. Since 2 threads can interoperate in one given core, a natural balance is set making the core context switches less frequent. This behaviour repeats itself towards the 8th thread. Because there are only 10 prediction modes, this pattern stops at the 10th thread. Further increasing the number of threads only results in OpenMP thread management overheads.

When the number of threads to strictly use is omitted, OpenMP creates a number of threads that is as close to the desired parallelization as possible but limited to a maximum which is equal to the number of available CPU cores. This is not visible on the server environments because they have more than 10 cores. But in the Jetson case, OpenMP would only create 4 threads out of the needed 10 since there are only 4 cores. Again with Figure 3.3, using 8 threads instead of 4 delivers far better performance. More in one performance mode than in the other, building an application without the desired number of threads hint to the compiler would be disadvantageous. Experimentation is important. This is also visible in the laptop environment since the number of available cores is also less than the number of prediction modes, but with different effects. Figure 3.4 depicts the OpenMP performance for the laptop system. In contrast

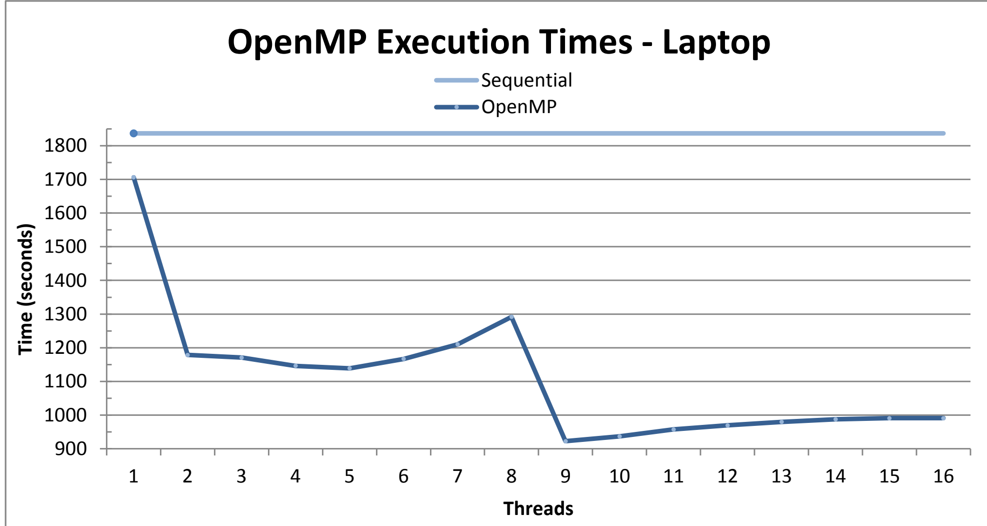


Figure 3.4: OpenMP thread variation execution time results (laptop, Lena image, $\lambda = 10$ and dictionary size of 1024).

with Jetson hardware, the laptop has more powerful components but does not have an installed operating system as simple as the Jetson. Therefore, as the number of threads is increased between executions, the system becomes more and more bottled up with running processes. This means that other minor processes that try to attain CPU time get put aside by the OS scheduler in favour of MMP. Above 5 threads, the execution gets to a point where MMP starts to get halted so frequently that it actually degrades its performance. This reaches a peak at 8 threads, that is, when the number of threads is equal to the number of cores. Like for the Jetson results in Max Performance mode (Figure 3.3) asking OpenMP for a number of threads that exceeds the number of CPU cores results in some OpenMP inner changes, like the threads scheduling constructs. The threads stop being forced to operate in a given core through core-affinity, allowing the other processes to operate more freely.

	Time			SpeedUp		
	Server 1	Server 2	Server 3	Server 1	Server 2	Server 3
Sequential	2097.0340	2157.7957	2139.1510	1.00	1.00	1.00
OpenMP with memory issues	711.1390	743.7252	810.4746	2.95	2.90	2.64
OpenMP without memory issues	701.3492	719.2870	801.3874	2.99	3.00	2.67
OpenMP Performance Gain	1.01	1.03	1.01			

Table 3.5: Execution time results (in seconds) for the MMP-OpenMP encoder (Lena image, $\lambda = 10$ and dictionary size of 1024, with and without memory related issues).

As stated in the previous section, some optimizations were performed in order to maintain the parallelism results and permit better memory management. The effects of these transformations vary depending on the hardware. Because OpenMP expects

	Sequential	With OpenMP Optimizations	SpeedUp
Server 1	2097.0340	1889.4373	110.99%
Server 2	2157.7957	1964.5985	109.83%
Server 3	2139.1510	1967.8613	108.70%
Laptop	1836.5860	1703.9239	107.79%
Jetson LP	5622.4783	4297.0377	130.85%
Jetson MP	5558.7413	4290.3113	129.56%
Raspberry Def	25734.4300	24826.1950	103.66%
Raspberry Turbo	25415.6100	24442.2320	103.98%

Table 3.6: Execution time results (in seconds) for the MMP-Sequential and improved sequential encoder (Lena image, $\lambda = 10$ and dictionary size of 1024).

that the memory space follows a shared architecture, memory mappings that cross other CPUs lead to unexpected stalls that are unaccounted for, since the implementation does not target a specific CPU/RAM for its routines. Table 3.5 presents the OpenMP execution times before and after the solved memory performance problems. As explained earlier in Section 2.1, the different RAM setups between Server 2 and 3 lead to different performance gains. Server 1 RAM DIMMs work at a higher frequency. The improvement of the MMP relation with the RAM bandwidth did not therefore provide as much gain in Server 1 as with Server 2. These optimizations were only a positive side effect of the need to correctly prepare the memory space for OpenMP. To that extent, Table 3.6 provides an insight to the archived results of an improved sequential version.

	Sequential with improvements	OpenMP with only 1 thread	SpeedUp
Server 3	1967.8613	1979.3682	99.42%
Laptop	1703.9239	1705.8587	99.89%
Jetson LP	4297.0377	4308.5267	99.73%
Jetson MP	4290.3113	4302.2790	99.72%
Raspberry Def	24826.1950	24698.1455	100.52%
Raspberry Turbo	24442.2320	24411.2610	100.13%

Table 3.7: Comparison between the improved MMP-Sequential encoder and the MMP-OpenMP encoder (with only 1 thread) execution time results (in seconds) on the Lena image with $\lambda = 10$ and dictionary size of 1024.

The results include the usage of the "-fopenmp" compilation flag. Strangely enough, this flag also provides some unexpected optimizations to the sequential version. The reason for this performance improvement is unknown. Indeed the flag should only add the OpenMP library in the linkage stage. Explicitly tying the execution to a given CPU/core provided some performance increase. Indeed setting CPU affinity to a free core allowed the algorithm to stay more time in the core frontline with less context switching incidences/occurrences, and fewer or none CPU migrations.

All these optimizations yield a 9.9% execution speed improvement to the original base sequential implementation. Because the sequential code utilizes the pre-created memory spaces sequentially, one at the time, the performance gains are larger than the ones of the OpenMP version. In OpenMP multiple threads work in the CPU cores, each accessing several distinct memory areas. Because the CPU cache memory is shared between all of them, each time one thread requires a previously cached memory region, the other 9 threads could have already replaced all the cache content with the one they need. This increased concurrency caused a considerable increase in the reported cache-misses. To avoid misleading results, the OpenMP version was also run with a single thread. The reported cache misses were close to the ones for the sequential version. By running with only 1 thread, it is also possible to sense the overhead due to OpenMP. Table 3.7 shows that OpenMP imposes no significant footprint in the execution times.

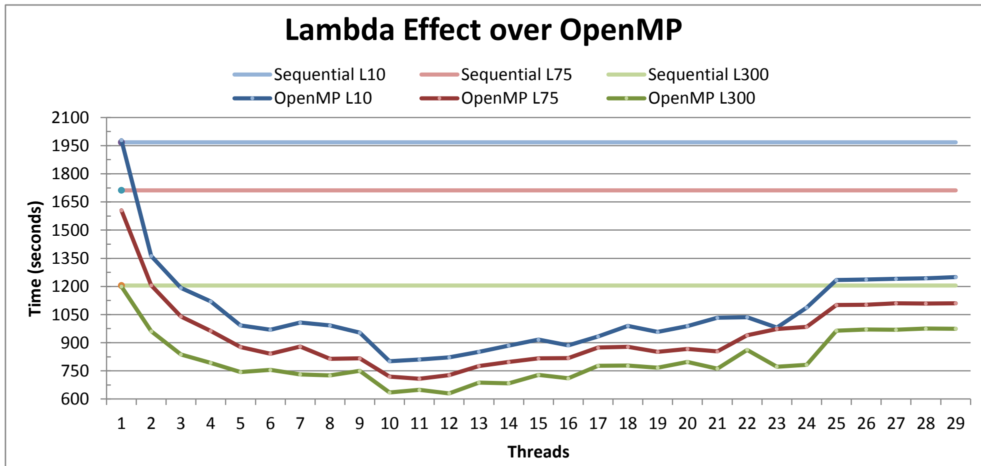


Figure 3.5: Lambda variation effect on OpenMP thread execution times and the sequential MMP encoder (Lena image, $\lambda = \{10; 75; 300\}$ and dictionary size of 1024), on Server 3.

Because the parallelization was made at a high level of the algorithm functions, other input factors such as λ and dictionary size variations also influence the algorithm performance. When λ value varies upwards (less encoded image quality) some of the more intense computations terminate sooner, possibly yielding smaller dictionary sizes, thus making the unfilled size look larger hence causing fewer dictionary revisions. The effects on the execution are shown in Figure 3.5 for Server 3. Note that $\lambda = 10$, $\lambda = 75$ and $\lambda = 300$ are standard values for high, medium and low quality in MMP, respectively. Since there is less time between memory operations and the intensive computations, the sequential algorithm is able to decrease significantly its execution time. Whereas the OpenMP version, which distributes the computations, already achieves maximum bandwidth speeds, and does not exhibit similar significant gains.

Sometimes the best results are not obtained exactly with 10 threads but with one

or two more threads. Processor pipelines are complex, and memory systems have a significant impact on performance. Due to core thread scheduling or other artefacts, sometimes a given thread sent to a CPU core has to wait for other external computations to finish. But other cores might be free (other freed first) and could assume the stalled MMP-OpenMP computations. We believe that this is the reason for the performance gain when using one extra thread. This is why using one extra thread on the 12-core CPU servers sometimes performs better than just with 10 threads. Not all 11 would actually produce work, but if, e.g. thread 2 were to be immediately stalled (before starting) for some external reason, a later thread 11 could pick up the unassigned work for iteration 2, thus partially covering the stalled time. Figure 3.5 clearly exposes this peculiarity for the executions using $\lambda = 75$ and $\lambda = 300$ with 11 and 12 threads, respectively.

Regrading stall related problems, it turns out that cache misses, for both instructions and data, are the dominant source of overhead in both the sequential and OpenMP implementations. As a standard metric, the Perf application reports that the sequential version has on average 1.9 instructions per cycle (IPC) while OpenMP only has 0.55. Generally, an IPC of 1.0 indicates that every instruction is processing in its due time (the optimal processing of work). This means that although the OpenMP version is up to 3× faster than the sequential version, it might still be far from its potential on average. The sequential version IPC is actually greater than 1.0 meaning that more than one instruction is being completed ('retired') per CPU cycle. This is possible because the CPU backend can process multiple micro-operations (µops) in parallel. Frontend and backend metrics refer to CPU pipelines. The frontend processes CPU instructions (fetch, along with branch prediction, and decode). While the µops generated by the frontend decoder are handled by the backend. Because of all the concurrency, the OpenMP version generated almost 8 times more stores to the last level cache (LLC, the largest and slowest), but roughly the same amount of loads as the sequential version. This means that so much cache information was being pushed aside to the last level that when it was needed again it was no longer there. To corroborate, the LLC caches misses also increased by a 7× factor.

Intel processors have many hardware level performance counters [82]. The Perf utility does not map must of them, but does allow their specification through a raw format. Taking the ones referring to resource stall into consideration, it is possible to know that OpenMP produced two times more idle cycles for when the pipeline is full (instruction-level bottleneck) and waiting for instructions to be retired. This roughly translates that around two times more level 2 cache misses are taking place. But since these values are for the overall algorithm, and there are 10 threads running in parallel

in the system, it means that OpenMP would actual be hitting at the very best 5 times more level 2 cache information thus being 5 times faster.

Many parallelization opportunities were not exploited to their full extent or were simply ignored. The original algorithm is greatly optimized for sequential single-core performance. But some improvements are still possible like showed here. In the beginning it seemed promising to profile the sequential code to determine the critical execution path and parallelize along this path. Unfortunately, this approach had its draw backs because the resulting parallelization is strongly bound to the original sequential code. This first approach did not cover some limitations imposed by the sequential implementation, unlike the CUDA/OpenCL GPU implementation in [3], where the algorithm was deeply reorganized to provide the most of modular critical areas. This is particularly relevant when comparing the performance behind CPU-only applications and CPU-GPU applications. No faithful comparison can be made if the way the algorithm logic is implemented is almost completely different. In this case, although the usage of a GPU consumes more energy, the CUDA implementation (Section 3.2.2) only takes about 300 seconds to process our reference case with NVidia GPUs. This is around 40% of the time of OpenMP, i.e., 2.5× faster.

Power Consumption Results

Figure 3.6 shows the ranges of the instantaneous power consumption as well as the overall mean power usage verified over executions with different numbers of threads on the Jetson hardware. From the plot, we can see that running OpenMP with one thread in Low Power is actually slightly more energy demanding than running the sequential version. But relating with Figure 3.3, it becomes evident that the OpenMP code is far more efficient. The execution time speeds-up by 1.30 (23.37% less time) and the overall mean energy needed per second only increased by 0.34 watts (5.63%). With some exceptions, using more threads in the Low Power mode seems to further increased the performance per watt. Unless configured to execute with 2, 4, 8 and 9, the energy demanded per second actually decreases to values below 6 watts. This is particularly relevant because it shows that due to the paralelization more work is performed in less time but also with a decreasing consumption. The best recorded values, in comparison with the sequential version, were archived with 10 threads. That is, an execution time speedup of 1.80 (44.43% less time) with an mean energy demand per second decreasing by 0.56 watts (9.30% less). Table 3.8 summarizes these calculations for all the thread executions, including the sequential version for comparison. From the table, it is possible to see that the optimal number of threads reduces the overall energy consumption to almost half of the originally needed by the sequential version.

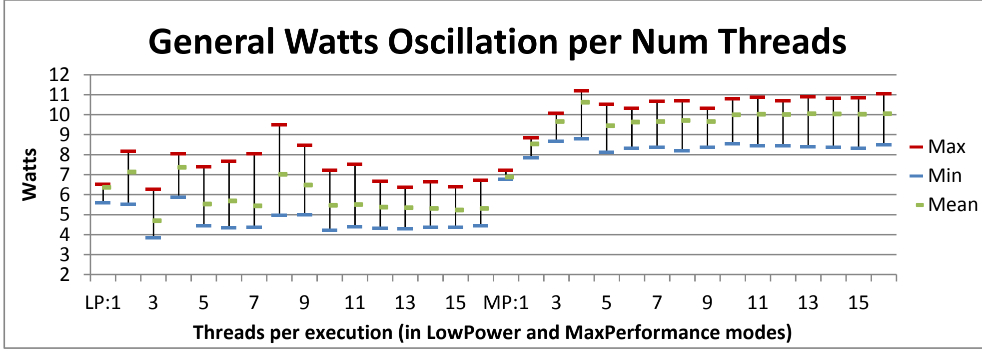


Figure 3.6: Mean (3 executions) watt oscillation of the MMP-OpenMP encoder (Lena image, $\lambda = 10$ and dictionary size of 1024) with different number of threads, on Jetson.

Threads	Low Power Mode		Max Performance Mode	
	Time SpeedUp	Overall Energy Needed ($\bar{x}$ Joules)	Time SpeedUp	Overall Energy Needed ($\bar{x}$ Joules)
Sequential	1.000	33834.18	1.00	33830.69
1	1.305	27386.12	1.29	29560.88
2	1.738	23015.46	2.08	22774.03
3	1.129	23364.13	2.45	21851.43
4	1.935	21370.48	2.70	21823.80
5	1.606	19311.83	2.62	19997.96
6	1.708	18665.94	2.72	19636.65
7	1.662	18361.67	2.73	19617.03
8	2.214	17770.63	2.79	19333.06
9	2.066	17617.55	2.73	19609.62
10	1.800	17054.08	2.94	18850.70
11	1.790	17253.65	2.93	18952.04
12	1.760	17142.95	2.93	18986.48
13	1.734	17311.21	2.92	19103.27
14	1.719	17332.50	2.92	19096.69
15	1.710	17159.50	2.91	19114.05
16	1.695	17579.40	2.90	19222.55

Table 3.8: Jetson energy consumption performance for the MMP-OpenMP encoder (Lena image, $\lambda = 10$ and dictionary size of 1024) with different number of threads for the Low Power and Max Performance modes.

As mentioned earlier, OpenMP behaves differently when the number of requested threads is bigger than the number of available cores. This is also evident in the energy consumption readings. Figure 3.6 show a stabilization of the mean consumptions after the 4$^{\text{th}}$ thread in Max Performance. This is more visible in the servers hardware, since they are equipped with more demanding components. Figure 3.7 shows the watt oscillation ranges verified over executions with different numbers of threads on the Server 2 hardware. Here, since the number of available cores is superior to the number of prediction modes, requesting more threads than the ones actually needed continuously generates overhead up to the point where the number of requested threads is larger than the number of available cores (24). At this point, in conjunction with the system, OpenMP changes the application to a less demanding scheme. There, it

only consumes roughly the same amount of watts as the 2-thread executions but with a run time between the run times achieved with 3 and 4 threads. But this does not mean that it computes in the same time for less energy. Like in the Jetson case, Table 3.9 summarizes the energy consumption calculations for the Server 2 executions (only up to 30 threads). Here it is possible to see that using 25 threads is faster than with 3 but consumes as much as with 4, henceforth not being better for the speed/energy ratio. On the servers, using more threads than the number of available cores in one CPU proved to be disadvantageous. Because there are no more cores available in the first CPU, the second CPU starts receiving threads as well. This leads to some additional memory managements since the second CPU does not have direct access to the memory surrounding the first CPU. But because the parallelization is limited to the 10 prediction modes, there is no actual way to say if MMP would benefit from using 2 CPUs.

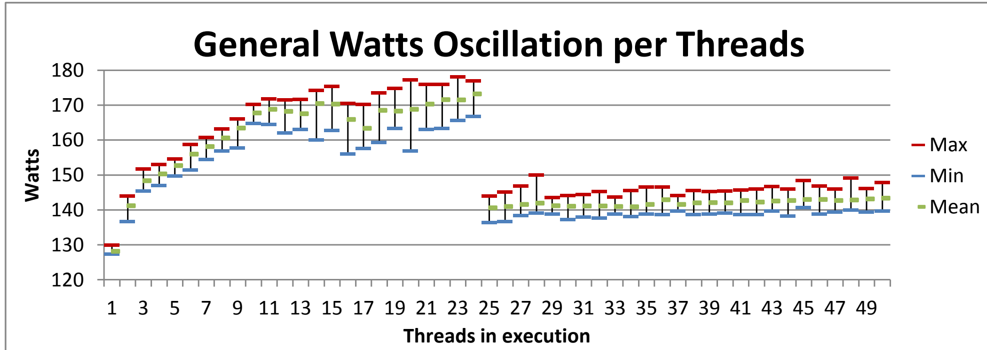


Figure 3.7: Mean (3 executions) watt oscillation of the MMP-OpenMP encoder (Lena image, $\lambda = 10$ and dictionary size of 1024) with different number of threads, on Server2.

Threads	Time SpeedUp	Overall Energy Needed ($\bar{x}$ Joules)	Threads	Time SpeedUp	Overall Energy Needed ($\bar{x}$ Joules)
Sequential	1.000	302502.88	—	—	—
1	1.092	253171.48	16	2.409	148582.78
2	1.645	185148.24	17	2.247	156859.37
3	1.924	166411.94	18	2.330	156080.04
4	2.099	154557.41	19	2.235	162471.43
5	2.397	137450.47	20	2.231	163336.59
6	2.432	138361.06	21	2.238	164191.41
7	2.419	141053.04	22	2.212	167391.30
8	2.434	142405.91	23	2.169	170640.24
9	2.485	141919.70	24	2.022	184827.92
10	3.000	120700.40	25	2.026	149769.53
11	2.924	124592.23	26	2.022	150428.85
12	2.780	130571.83	27	2.025	150810.54
13	2.643	136799.13	28	2.022	151489.17
14	2.647	138952.63	29	2.025	150435.94
15	2.605	141087.91	30	2.017	150946.77

Table 3.9: Energy consumption performance for the MMP-OpenMP encoder (Lena image, $\lambda = 10$ and dictionary size of 1024) with different number of threads, on Server2.

3.4 OpenCL-CPU

Open Computing Language is an open framework for writing programs that execute across heterogeneous platforms. OpenCL views a computing system as consisting of a number of compute devices (CD). CPUs, GPUs, digital signal processors (DSPs), field-programmable gate arrays (FPGAs) are some examples of CDs that generally support, or are supported by, OpenCL. The programming language itself is based on C99. It specifies a language for programming devices and APIs to control the platform and execute programs. The standard interface for parallel computing provided by OpenCL uses task-based and data-based parallelism. Functions executed on an OpenCL device are called kernels [83], which are marked as entry points. Each compute device typically consists of several compute units (CU), which in turn include multiple processing elements (PE). A single kernel execution can run on all or several PEs of a CD in parallel (here CD can also be simply referenced as 'device'). These kernel functions are intended to be compiled at run-time so that the source code becomes portable between the implementations for various 'host' devices [84]. OpenCL C language is extended to facilitate the use of parallelism with vector type operations [85]. Another key factor to this language is its restrictions [85], which include: no function pointers; no bit-fields; no variable length arrays; no recursion; and no standard headers.

How a CD is divided into CUs and PEs is vendor-dependent. A compute unit can be thought of as a 'core' but this abstraction is hard to define transversely to all device types (even between CPUs) [86]. Typically GPUs are roughly composed of several CUs in which several PEs work in an instruction level lockstep. CPUs on the other hand have only some cores/CUs which are their own single PEs. But the number of CUs may not correspond to the number of cores claimed by marketing literature (e.g. it may actually be the sum of the SIMD lanes) [87].

The OpenCL platform can be directly mapped to GPUs since they have hundreds of PE cores. Since CPUs have far less number of cores, mapping the fine-grained OpenCL processing elements can be unnatural. Additionally, applying fine-grained parallelization would restrict CPU cache utilization since one work-item only processes a small proportion of the memory/dataset. CPU cores, in comparison to GPU compute units, are 'overpowered' cores, designed to perform more intense tasks with the support of dedicated hardware, a mismatch for the OpenCL processing elements. Particularly for CPUs, besides scalar types (e.g. *float*, *double*), OpenCL provides fixed-lengh vector types (e.g. *float4* — 4-vector single-precision floats). The available lenghts are: two, three, four, eight and sixteen; for various base types [83]. Instructions using these

vector types are mapped onto SIMD instructions sets (e.g. SSE or VMX) when running OpenCL on CPUs [84].

Memory hierarchy in OpenCL is divided into four levels [84]: *Global*) largest and with highest latency but shared across all PEs; *Read-Only*) smaller with low latency, writable by the host but not by the device; *Local*) shared by a group of PE; and *Private*) fastest, smallest and per-PE. Memory buffers reside in specific levels and pointers are annotated with region qualifiers to address a specific level. But not every device implements each level of this hierarchy in hardware. While GPUs have an on-chip memory region designed to be shared by all workers in a given work group (local memory), which is much faster than the off-chip global memory alternative, CPUs on the other hand do not have it. As a result, all memory objects declared to use such memory are mapped, with extra overhead, into sections of the global memory (RAM). Consistency between the various levels in the hierarchy is relaxed. Because it is implementation dependent, some compilers may choose to optimize local defined memory by temporary copying the elements from the global memory (RAM) into the registers, operate over them in a registers-to-register manner, and then store them back instead of operating them directly over the global memory.

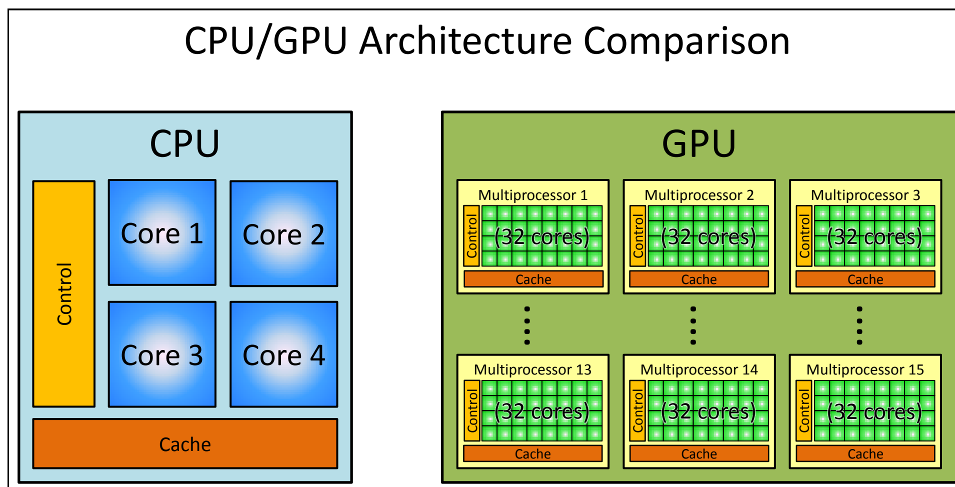


Figure 3.8: Top-layer abstracted comparison between CPU and GPU Architecture. The CPU abstract representation on the left consists of 4 cores/ALUs, a cache system and controlling hardware. The GPU system (right) is composed of 15 microprocessors that contain 32 'light' cores each, forming a total of 480 cores, i.e., 15 workgroups with 32 lock-step workers.

Figure 3.8 serves as an abstract comparison between CPU and GPU architectures. Each design has its own strong points that need to be taken into consideration when implementing for GPU or for CPU. GPUs have high compute density while CPUs have low compute density. High computations per memory access can be made in GPU whereas complex control logic and large caches are available in CPUs. GPUs

are built for parallel operations thus having many execution units (ALUs) and CPUs for serial operations (needing fewer ALUs) but with higher clock speeds. But CPU shallow pipelines have less than half of the stages deep GPU pipelines have [88]. All of this, plus memory architecture, makes CPUs low latency-tolerant in contrast with GPUs which have higher tolerance.

Useful OpenCL beginner-developer driven information and concepts can be found in [85] and [87].

3.4.1 Migrating from OpenCL-GPU to OpenCL-CPU

The first unexpected behaviour we encountered in porting the MMP-OpenCL-GPU to OpenCL for CPUs was that the value for the work group size, obtained through the *clGetKernelWorkGroupInfo()* API call, would be changed to one, thus nullifying parallelism. This occurred whenever existing barriers were removed from a kernel. In fact, the compiler was adding barriers to have cores/threads performing synchronously. This forced communication among cores and thus negatively affected the performance. Furthermore, in one of the kernels, the compiler also merged some work items together in the same thread, swapping the executions to preserve the parallelized execution order and the barriers. Merges also happened when the number of work-items was larger than the number of CPU cores/possible threads.

To deal with the specificities of OpenCL over multicore CPUs, we simplified the MMP kernels in order to have all work-group sized to one. The rationale lies in the fact that *i*) a CPU can only efficiently run a few threads at any given moment and, as stated earlier, *ii*) the OpenCL compiler merges workers together. Moreover, on a CPU, each worker benefits from dealing with a larger workload, promoting better cache utilization.

Memory Mapping. Since we are mapping OpenCL on CPUs, the host and device are the same, thus sharing the same memory space. Explicit host to device (H2D) and device to host (D2H) data transfers may become completely unnecessary. OpenCL imposes some forms of explicit data transfers but does not impose restrictions on memory access patterns. It is up to the device drivers and compiler to make their choices on whether or not to actually replicate the data or just read it from already allocated space. To overcome this irregularity, we then applied the *zero copy* technique. Specifically, by changing the explicit data transfer functions to *clEnqueueMapBuffer* and

clEnqueueUnmapMemObject, we get an OpenCL memory pointer to the same mapped region. These functions perform like a token. By mapping a region, we inform that the host now has the right to read and write. And by unmapping it, we return the control to the device kernel.

Memory Access. As a result of another mismatch between GPU and CPU architecture, due to their different purposes, GPU code with explicit memory coalescing will suffer performance degradation when run on CPUs. This problem occurs when each work-item accesses a column of data elements that are non-adjacent, thus affecting data locality. This happens because memory coalescing for GPUs is orthogonal with cache-beneficial access in CPUs. A GPU optimal code will have a column-major access pattern to enable more effective transactions of global memory loads and stores, in opposition to the required row-major order needed to preserve the cache locality within each CPU thread. Indeed, on the CPU, the OpenCL-GPU coalesced memory optimization was causing heavy cache misses. To this extent, we had to rewind the previous OpenCL-GPU coalesced memory optimizations, and return to a simpler access pattern.

Metadata Access. GPU devices have special purpose registers to hold workers' local and global IDs that speed up access to these metadata. CPUs lack these registers and thus repeated calls such as *get_global_id(0)*, inside a loop, are costly. We also prefetched values such as the number of work groups, local ID and local size to explicitly private-defined variables.

Fostering Loop Unrolling. Originally replicated 10 times in the OpenCL-GPU implementation and later put back together to become a simpler OpenCL-CPU implementation, *optimize_block* kernel had to be replicated once more. This time to eliminate dependencies in some inner-loops, since they directly depended upon the prediction mode at hands (which is known before launching the kernel). This way, the compiler was able to unroll those inner loops.

3.4.2 SIMD-based Optimizations

Through CPU SIMD instructions and vector registers, data elements can be packed into vector data to run simultaneously on SIMD hardware units, hence benefiting from vectorization [89]. As stated earlier, OpenCL simplifies the use of SIMD instructions, sparing developers from lower-level details (e.g., as opposed to Intel Intrinsics [90]).

In fact, some OpenCL compilers perform implicit code vectorization, attempting to transform scalar code into vectorized code. This is done by packing work items together and running them in a SIMD fashion. The number of packed work items is dependent of the width of the underlying SIMD engine. Figure 3.9 serves as a visual representation of the work-load distribution between workers and further inner-optimization with SIMD instruction sets.

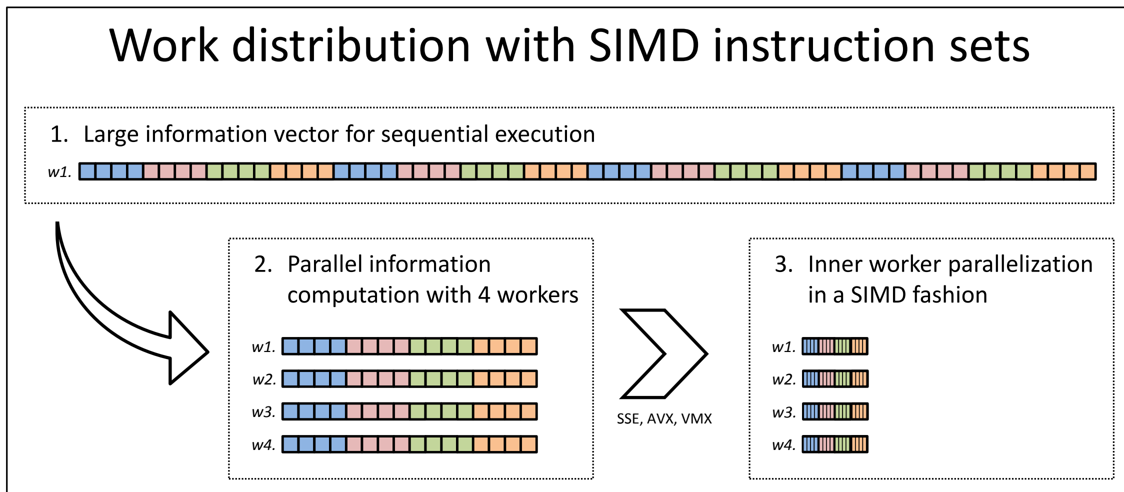


Figure 3.9: Visual representation of the work-load distribution between works and further optimization with SIMD instruction sets. A large information vector on the top is split through several workers in the bottom left running in parallel that further parallelize work by packing information operations with SIMD instruction sets.

Explicit Vectorization. Besides the implicit optimization attempted by compilers, we can explicitly promote the use of vectorization by declaring variables as vector datatypes such as *int4* and *float8* in the kernel code. The compiler attempts to employ vector-based instructions, making use of the explicit knowledge conveyed by the variables declared as vector datatypes. Note that OpenCL specification specifically states that usage of vector datatypes in arguments to a kernel function are assumed to be "appropriately aligned" [83] to the matching data type. For example, a *float4* variable needs to be aligned to a 16-byte boundary.

Since the MMP-OpenCL-CPU implementation almost imposed that no barriers, no memory coalesced transformations nor local managements were to be in place, work item vectorization was fairly simple. The *update_dic* kernel was vectorized by using the *int4* datatype and by padding the aligned data. The vectorization of the kernels *optimize_block* and *j_reducter* required padding the data in an adaptive manner since the MMP block being processed can have one of possible sizes (which impacts calculations). This was dealt with a switch-case to adapt whether the data could fit in 2, 4 or 8 bytes.

Vectorization of Data Reduction. SIMD vectorization was further used to optimize data reduction that forms the core of the *j_reducter*. Specifically, after having completed the usual iterative sum to obtain a vector with 8 elements, the 8-element vector is further reduced to a scalar by performing a vector addition over its high and low parts, that is, adding *vector.hi* + *vector.lo* into an *int4* vector. The procedure is repeated for the *int4* vector, yielding an *int2* vector, and then again for the *int2* vector. This way the reduction sum over an 8-element vector is performed through vectorization, using, in this case, half the instructions needed.

3.4.3 Results

Table 3.10 shows the execution times for the various versions of MMP and the speedup relatively to the corresponding sequential (non-multithreaded) version. The rows of the table correspond, respectively, to the sequential version (*Sequential*), the OpenCL GPU optimized version ran on GPU (*OpenCL-GPU inGPU*), the OpenCL GPU optimized version ran on CPU (*OpenCL-GPU inCPU*), the OpenCL CPU optimized version ran on CPU (*OpenCL-CPU inCPU*) and the OpenCL CPU optimized with explicit SIMD instructions (*OpenCL-CPU inCPU with SIMD*). As referenced in Chapter 2, the Server 2 and 3 correspond to server machines with different hardware RAM memory modules. Here we assess the importance of memory system speed and bandwidth on the execution times of MMP.

	Time			SpeedUp		
	Laptop	Server 2	Server 3	Laptop	Server 2	Server 3
Sequential	1836.5860	2157.7957	2139.1510	1.00	1.00	1.00
OpenCL-GPU inGPU	583.0056	496.0219	506.6877	3.15	4.35	4.22
OpenCL-GPU inCPU	976.3864	1029.9416	1035.9735	1.88	2.10	2.06
OpenCL-CPU inCPU	815.9278	967.7535	1023.3004	2.25	2.23	2.09
OpenCL-CPU inCPU with SIMD	671.7362	901.7323	958.5457	2.73	2.39	2.23

Table 3.10: Execution time results evolution (in seconds) for different iterations over the MMP-OpenCL encoder (Lena image, $\lambda = 10$ and dictionary size of 1024). Server 2 and server 3 are equipped with the Titan GPU.

Surprisingly, the laptop executions (that do not use the GPU) perform around 15% faster than the server ones. This is due to the faster system memory of the laptop. In fact, the importance of the memory system is further highlighted by the execution time differences between the two memory configurations of servers: Server 2 is roughly 5% faster than Server 3. This can be explained by the fact that *i*) the RAM modules of Server 2 are clocked at an higher frequency than the ones of Server 3 configuration (1600 MHz vs. 1333 MHz) and that *ii*) having four memory modules better suits

a four memory channel CPU such as the Xeon 6-Core E5-2620 V2, especially on a multiprocessor setting [91].

Overall, the GPU optimized version of OpenCL yields the fastest execution times, achieving a speedup over the sequential version of 2.93 times for the laptop plus GPU and 3.45 times for Server 2 plus GPU. These results highlight the importance of the GPU, with the more powerful Titan GPU yielding faster results than the 570M.

Comparing the execution times over CPUs of the CPU-optimized OpenCL (*OpenCL-CPU inCPU*) version vs. the GPU-optimized one (*OpenCL-GPU inCPU*), reinforces the importance of the memory system. Indeed, while a 19.7% performance gain is achieved by the laptop, practically no performance is gained for the server with the slowest memory configuration. When the SIMD-based optimization is considered, the performance gain over the GPU-optimized version run on CPU is 1.454 for the laptop and 1.081 for the fastest server configuration. For the laptop, the SIMD optimization adds a 20% speedup over the non-SIMD OpenCL-CPU version, and around 1.07 for the server.

Globally, the SIMD-enabled OpenCL version achieves a 2.54 speedup over the sequential version vs. 2.93 attained by the OpenCL-GPU version ran on the laptop GPU. This means that substantial speedup can still be achieved with OpenCL when no GPU is available. Additionally, enabling explicit usage of SIMD instructions in OpenCL can provide some performance improvement.

Table 3.11 shows the execution times broken down per kernels of each OpenCL version of MMP ran on the laptop machine. For this table it is also included two iterations between the OpenCL-GPU implementation and the OpenCL-CPU. The first shows the effect of the groupsize downgrade to size one (*with GroupSize 1*), directly yielding an 1.27 speedup. And a second one shows the gain with the removal of memory coalescing and the usage of the *zero copy* technique (*with CPU-like Memory Access and Mapping*), further yielding more 0.31 speedup.

The results confirm that the usage of SIMD vectorization instructions through OpenCL meaningfully improve performance, cutting in half the cumulative execution time for all kernels. Comparing the GPU-based version with the SIMD-based version execution times, the GPU provides a much faster execution for the *optimize_block* kernel. However, explicit SIMD-based optimization of the reduction loop gives a significant boost to the *j_reducter* kernel, making it closer, in performance terms, to the GPU-based execution. Finally, the SIMD-based version of the memory intense *compare_blocks* and *update_dic* kernel performs better than the GPU optimized version.

	k1	k2	k3	k4	Kernel-Time SpeedUp	Execution Time SpeedUp
OpenCL-GPU inGPU	33.1733	23.0504	1.3554	0.0900	8.04	1.67
OpenCL-GPU inCPU	279.5924	174.8818	9.3456	0.0259	1.00	1.00
* with GroupSize 1	252.3004	110.1652	3.4965	0.0169	1.27	1.09
* with CPU-like Memory Access and Mapping	230.6028	60.7644	1.6812	0.0120	1.58	1.17
OpenCL-CPU inCPU	213.9147	56.6192	1.3336	0.0114	1.71	1.20
OpenCL-CPU inCPU with SIMD	113.8649	32.4582	0.7279	0.0092	3.15	1.45

* Incremental modifications to the OpenCL-GPU inCPU implementation

Table 3.11: Kernel time results evolution (in seconds) for different iterations over the MMP-OpenCL encoder (Lena image, $\lambda = 10$ and dictionary size of 1024), on laptop. Here *k1–4* respectively mean kernels *optimize_ block*, *j_ reducer*, *compare_ blocks* and *update_ dic*.

Power Consumption Results

Figure 3.10 shows the ranges of the instantaneous power consumption as well as the overall mean power usage verified over executions with different numbers of threads on the Server 2 hardware. From the plot we can see that running OpenCL without explicit SIMD instructions (on the left) is overall less energy consuming than with SIMD usage (on the right). Even with only 1 thread, it is possible to see a 2 watts jump from the execution without SIMD to the one using it. But after the number of threads has passed the number of existing CPU cores (24) the instantaneous power consumption without SIMD usage rapidly increases towards the same level as the one with SIMD instructions. This happens, as stated before, because the OpenCL compiler merges workers together (making use of the SIMD hardware) or, in this particular case, merges work by swapping the executions to preserve the parallelized execution order. Even so, this does not mean – execution wide – that it consumes less energy than with SIMD, nor that it is preferable not to use SIMD instructions from an energy consumption perspective. Table 3.12 summarizes the energy consumption calculations for the Server 2 executions (up to 30 threads) and, within parentheses, the resulting energy speedup relatively to the corresponding sequential (non-multithreaded) version. Here, we can clearly see that the SIMD version obtains higher speedups in both execution time and energy consumption. Further comparison of these results (e.g. dividing the overall energy column by the time speedup) indicates that the SIMD implementation consumes a smaller energy ratio.

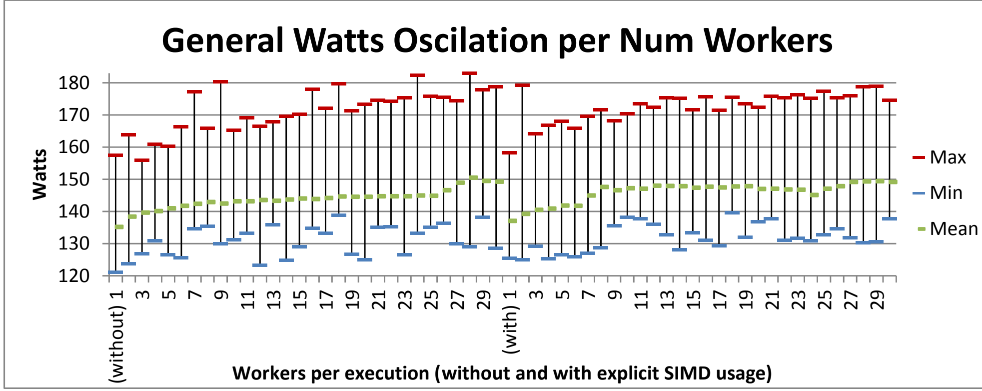


Figure 3.10: Mean (3 executions) watt oscillation of the MMP-OpenCL encoder (Lena image, $\lambda = 10$ and dictionary size of 1024) with different number of threads, on Server 2, with and without explicit SIMD instruction usage.

Threads	OpenCL-CPU inCPU		OpenCL-CPU inCPU with SIMD	
	Time SpeedUp	Overall Energy Needed ($\bar{x}$ Joules)	Time SpeedUp	Overall Energy Needed ($\bar{x}$ Joules)
1	1.152	253190.94 (1.19×)	1.490	198471.93 (1.52×)
2	1.352	220699.69 (1.37×)	1.884	159426.59 (1.90×)
3	1.493	201787.92 (1.50×)	2.041	148491.21 (2.04×)
4	1.574	192086.16 (1.57×)	2.123	143227.56 (2.11×)
5	1.660	183237.30 (1.65×)	2.164	141400.59 (2.14×)
6	1.736	176171.62 (1.72×)	2.171	140854.88 (2.15×)
7	1.798	170843.01 (1.77×)	2.215	141155.52 (2.14×)
8	1.852	166409.68 (1.82×)	2.220	143452.13 (2.11×)
9	1.903	161481.10 (1.87×)	2.245	140839.74 (2.15×)
10	1.946	158728.37 (1.91×)	2.258	140722.29 (2.15×)
11	1.982	155854.70 (1.94×)	2.252	140844.18 (2.15×)
12	2.014	153781.79 (1.97×)	2.284	139818.06 (2.16×)
13	2.052	150699.33 (2.01×)	2.289	139436.83 (2.17×)
14	2.080	149069.94 (2.03×)	2.301	138615.94 (2.18×)
15	2.107	147445.13 (2.05×)	2.328	136566.75 (2.22×)
16	2.128	145872.05 (2.07×)	2.362	134863.99 (2.24×)
17	2.149	144753.48 (2.09×)	2.337	136106.00 (2.22×)
18	2.169	143823.60 (2.10×)	2.332	136718.39 (2.21×)
19	2.181	142973.26 (2.12×)	2.332	136765.29 (2.21×)
20	2.195	142058.61 (2.13×)	2.333	135946.36 (2.23×)
21	2.204	141659.35 (2.14×)	2.328	136269.38 (2.22×)
22	2.211	141221.47 (2.14×)	2.322	136445.19 (2.22×)
23	2.211	141197.99 (2.14×)	2.319	136525.46 (2.22×)
24	2.222	140742.63 (2.15×)	2.272	137842.09 (2.19×)
25	2.224	140499.68 (2.15×)	2.317	136906.51 (2.21×)
26	2.219	142556.33 (2.12×)	2.327	137069.27 (2.21×)
27	2.218	144921.79 (2.09×)	2.322	138583.68 (2.18×)
28	2.215	146564.46 (2.06×)	2.319	138933.01 (2.18×)
29	2.214	145672.62 (2.08×)	2.317	139126.07 (2.17×)
30	2.213	145539.79 (2.08×)	2.317	138891.33 (2.18×)

Table 3.12: Server 2 energy consumption performance for the MMP-OpenCL encoder (Lena image, $\lambda = 10$ and dictionary size of 1024) with different number of threads.

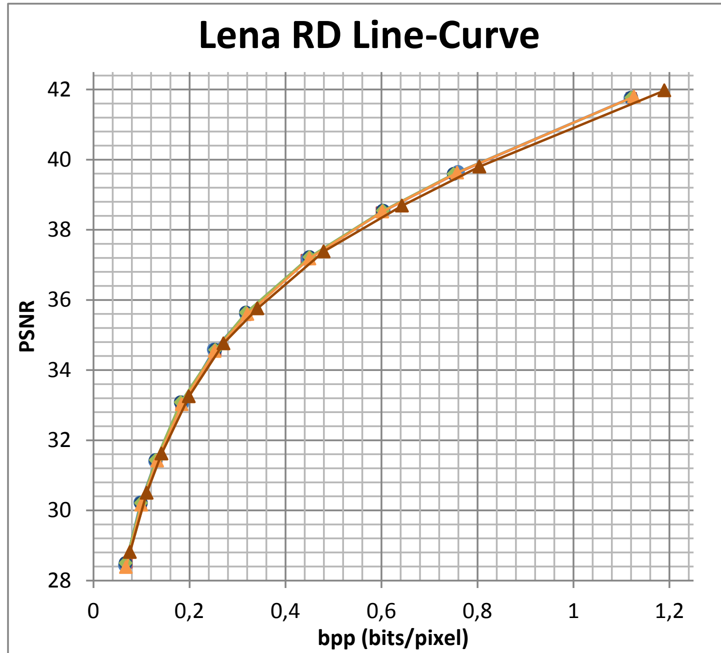
3.5 Conclusions

When talking about green-computing one may opt to embark on the optimizations path. With this chapter we have showed that more optimized and parallel code is actually more energy efficient. And that there is a hardware-OS dependent balance between the number of threads and the attained performances. In this path we have also unravel some OpenMP behaviours with the successful implementation of an MMP-OpenMP version, which achieves speedups of up to 3×, and improved the original sequential version by around 10%.

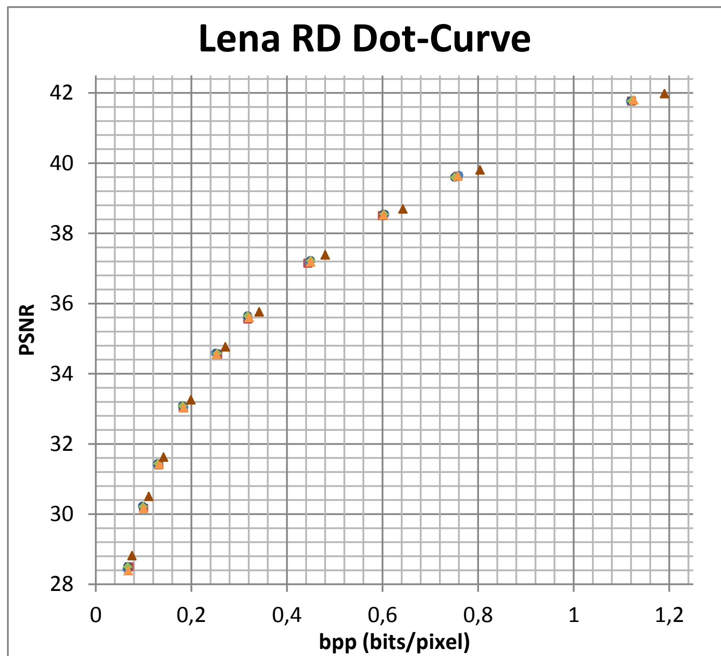
Within this chapter we also successfully documented an OpenCL-CPU case study were we converted the existing OpenCL-GPU implementation to the CPU alternative reviewing several important aspects that differ and ultimately define the implementation direction. We also took advantage of the CPU *Single Instruction Multiple Data* instructions through the OpenCL API abstraction making the CPU-core parallelization one step deeper. Overall, the OpenCL reimplementation for CPU only takes around 2× more time than in a GPU but with around 120 times less cores (24 in server CPU) than a GPU (2880 in Titan Black).

To complete the presented work, since we never referenced the compression results (only the performance speedups), we expose here the Rate-Distortion curves for the algorithms in Figure 3.11. These curves address the problem of determining the minimal number of bits per symbol, as measured by the rate R, that should be communicated over a channel, so that the input signal can be approximately reconstructed without exceeding a given distortion D. That is, the curves present the PSNR and bits-per-pixel relation that the algorithms achieve – image compression quality. Two completely overlapping curves means that both algorithms produce the same output/quality.

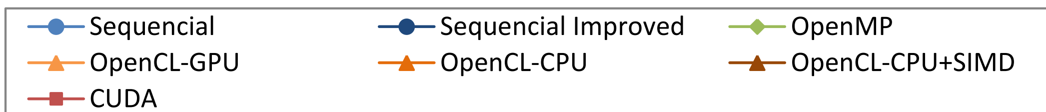
Both Figures 3.11a and 3.11b provide the same information but in different visualizations. While Figure 3.11a provides the traditional visualization with lines (line-graph) hence producing a curve, Figure 3.11b only shows the information dots that compose the information table without the connecting lines so that the results overlap may be better observed if needed. Later Figure 3.11c provides the shape/colour labels for the plot figures. All implementations maintained their expected results ranges – i.e. all results are overlapping.



(a) Traditional Line-graph



(b) Alternative Dot-graph



(c) Labels

Figure 3.11: Rate-Distortion Curves for the existing MMP implementations in the traditional line-graph and alternative dot-graph for better result-overlap visualization, on the Lena image for variable $\{\lambda,$ dictionary size$\}$ values of $\{(200, \{1000, 500, 300, 150, 75\}); (100, \{50, 25\}); (50, \{15, 10, 5\})\}$ – like in [7, 6, 3].

Chapter 4

HEVC+SS+GT – Multi-GPU, Many-Core, Multi-Thread

In this chapter, we focus on the parallelization of what we call HEVC+SS+GT. First we present a small review of the HEVC – current video encoding standard – algorithm, followed by an analysis on the main existing obstacles and impediments to its proper parallelization (Section 4.2). To match with the algorithm theoretical review another, more technical, description is made in Section 4.3. With that knowledge, we further propose and analyse the implementation for a CPU multi-thread/multi-CPU environment resorting to OpenMP (Section 4.4), and introduce a possible GPU implementation within the parallelization limits (Section 4.5) and a multi-GPU extension (in Section 4.6). Finally, Section 4.7 summarizes this chapter.

This chapter relies on knowledge presented previously in this document. Apart from Chapter 2, perusal of the subsections 3.3, 3.3.1, and 3.4 is advised.

4.1 High Efficiency Video Coding

The High Efficiency Video Coding (HEVC) is the video compression standard which succedded to the widely used Advanced Video Coding (AVC). AVC is also know as H.264 [92]. H.264 is a block-oriented motion-compensation-based video compression standard that is currently one of the most commonly used formats. It has a very broad application range that covers all forms of digital compressed video from low bit-rate internet streaming applications to HDTV broadcast and Digital Cinema applications with nearly lossless coding. Currently, HEVC is known as H.265 [93]. HEVC was developed by the Joint Collaborative Team on Video Coding (JCT-VC) – a collaboration between two well known standardization groups ISO/IEC MPEG and ITU-T VCEG.

In comparison to AVC, HEVC doubles the data compression ratio for the same level of video quality, or substantially improves video quality for equivalent bit rate values. The main differences from AVC to HEVC include: the expansion of the pattern comparison and difference-coding areas from 16x16 pixel to sizes up to 64x64, improved variable-block-size segmentation, improved "intra" prediction within the same picture, improved motion vector prediction and motion region merging, improved motion compensation filtering, among others [94].

Complete information about HEVC/H.265 can be found in [95] and comparisons between AVC/H.264 and HEVC/H.265 in [96, 97]. A more detailed review over the HEVC standard can also be found in [93].

4.1.1 HEVC-based holoscopic coding using Self-Similarity compensated prediction and Geometric Transformations for efficient disparity compensation

In this chapter we will be working with an HEVC-based implementation that focuses on holoscopic image/video processing, using geometry-based disparity compensation developed by Conti et al. [98, 4, 99] and further complemented by Monteiro et al. [5, 100] using Geometric Transformations (GT). The concepts behind holoscopic imaging were firstly proposed by Lippmann [101] and referred to as integral photography in 1908. In a nutshell, the holoscopic imaging system comprises a regularly spaced array of small micro-lenses, comparable to a "fly's eye" lens array [102]. Making this holoscopic-HEVC-based algorithm implementation-toned to search and match similar micro-images in a better way than the normal HEVC implementation would match [99, 103].

As a contribution to the previous work, we were asked to optimize part of this existing implementation since it builds up a lot of additional computation (6.9 days) aside of the regular HEVC inner workings (16 minutes). When properly programmed, multicore and manycore platforms such as CPUs and GPUs can yield fast execution, achieving substantial speedups over pure single-threaded versions [104]. The objective of the work described in this chapter was to speedup the GT module, i.e., the part of the code which searches for the GT that provides the best match between a reference area and the block that is being encoded. The next section will briefly describe the GT algorithm before we move on to the description of the parallel implementations.

Geometric Transformations module

The Geometric Transformations search step/module mainly works as follows:

- An rectangular image block with size HxW (H rows by W columns) is given. The block exists in larger memory buffer that contains the surrounding image region or even the whole image. Being rectangular, there are 4 image corners: top-left, top-right, bottom-right, bottom-left – $c0$, $c1$, $c2$, $c3$ respectively.

- Around each image corner there are 9 equally distributed spots/points (8 plus the corner itself) which form the local search region. Figure 4.1 helps to describe the concept. The points around a corner occupy a square region of length $L = \min(H, W)/2$. Therefore, each point is vertically and horizontally distanced by $L/2$.

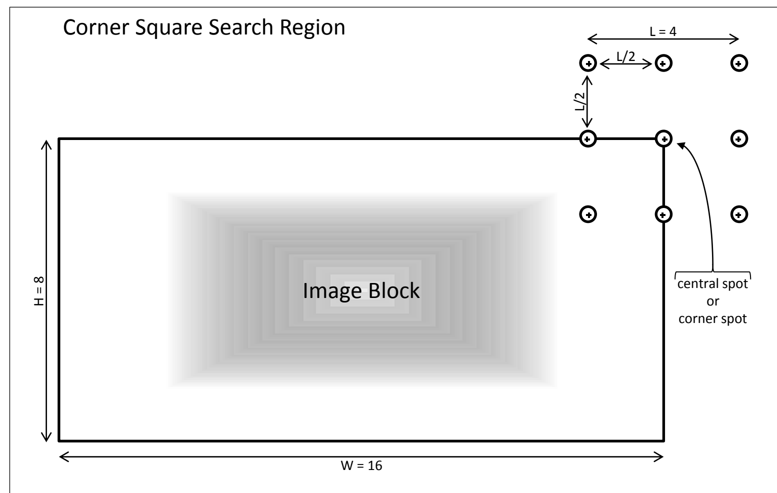


Figure 4.1: Visual representation of the 9 local, equally distributed, spots/points of the top-right corner ($c1$) search region in the Geometric Transformations module.

- Considering 9 different points for each corner, there are $9 * 4$ points. By selecting one point from each corner, there are 9^4 point combinations available. The algorithm moves the image block corner definitions over each point combination and, if the selected points are within the image buffer memory region, verifies the distortion of the new quadrilateral shape after a projective transformation of its geometry. Figure 4.2 provides a visualization of the corner point combinations cycles/iterations. Cycle 1, iteration 1, is the starting point of the algorithm. In a first cycle, only the 9 points of corner $c3$ are evaluated. After all points in $c3$ are evaluated, a point in $c2$ is changed and all combinations with corner $c3$ are re-evaluated (cycle 2). The process continues and passes e.g. by cycle 11 where

corner $c1$ is already processing its second point. The process continues until all possible point combinations are verified.

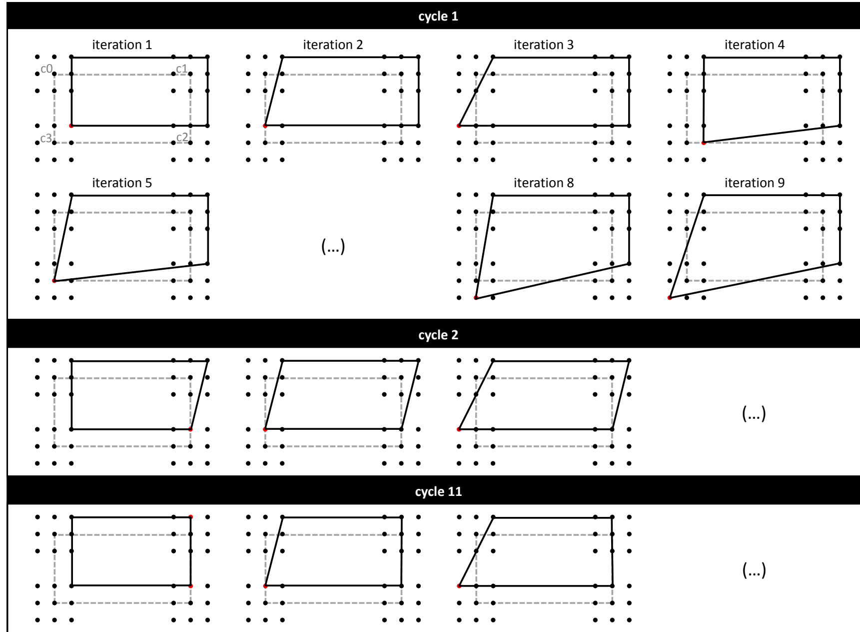


Figure 4.2: Sort visualization of the corner point combinations cyclic behaviour in the Geometric Transformations module.

Apart from the described behaviour, the algorithm also recalculates the L sized regions after each 9 points of a region have been evaluated. It does it by moving the region central point to the point were the best/lowest distortion was found and restarts the local cycle process. This complements the already described behaviour as follows:

- At compilation time a constant integer N is defined within the range $[1 : 5]$ (larger values are technically allowed). The number represented by N defines the maximum number of times a region is verified, while $N-1$ defines the maximum number of translations (also referenced as a 'zoom') that any corner can do.

- After the 9 points of a corner local region have been evaluated, a 'zoom' may occur.

- A 'zoom' can only occur if the count towards N allows it and if $L > 1$. If the conditions allow it, the region central point is moved to the point which previously provided the best/lowest distortion, then:
  - The L size is halved for that corner and the surrounding 8 points are recalculated/moved accordingly.

- All the previous iterations are re-iterated since the corner local region has changed, thus generating new – slightly different – quadrilateral shapes that may provide even lower distortions.
- Depending on the value of N and the position of the block, new 'zooms' may occur again and all 9 points of a given corner are evaluated.

With the new described behaviour, the maximum number of possible combination becomes $(9 * N)^4$ for each block if L allows it.

4.2 Parallelization Impediments

In complex applications and algorithms, there are commonly intrinsic, non-parallelizable issues. These issues can be data and control dependences related or architecture-related. Padua [105, p.1412] very concisely says that:

> (...) the best approach is to rethink the algorithms and program, and restructure the program or use entirely different algorithms. These steps can be very difficult and nonintuitive, unless a developer is specifically driven by the parallel languages (...)

In the HEVC+SS+GT implementation the original developer was driven by the sequentiality of the code, making use of the iteratively produced data in following iterations. This lead to both data and control dependencies.

Several meetings with the authors of the sequential prototype (Monteiro et al.) took place to overcome these issues. The first problem lied in the corner iterations of the GT module. As explained in the previous section, each corner cycle is actually compromised of 9 inner iterations over a corner central point (Figure 4.2). Originally, if one of these inner iterations found a better/smaller distortion value, it would update the global best corners combination values. But this means that if we later find the same distortion value again, but for a different corner combination, that the value will not be saved (since it is not smaller than the current one). Non-extensive tests were made and depending in the order in which the corners and the inner 9 iterations were processed the final encoding results would differ. Differences of 4% in bitrate and 0.07 PSNR drops were registered. This means that in a parallel environment, the concurrent executions must always follow some specific order of events or else, in other words, the

execution results will become inconsistent, displaying different compression rates and image qualities between executions.

In addition, globally, every 10th iteration is dependent on the previous 9 since after every 9 iterations a best corner point can be locally updated – changing how future iterations behave.

Apart from the corner sequential execution restriction, another issue was found regarding the update of a corner central point, as discussed in the next section.

Sequential Code Modifications

In the already exposed GT module description it is referred that each corner will readjust its central point to the previous best point found after its inner 9 iterations. This was not actually happening like the theoretical review stated. Some changes were made to the code in order to make both the theoretical and technical descriptions implementation compliant. These corrections were firstly proposed and later accepted by the original party. This was a two step amendment.

In a first step, we reviewed the implementation and found that between corner iterations the corresponding central points were being reset to their original values. This was only perceptible with a N value bigger than 1, since the central corner points change and later usage is only done for the upcoming iterations to use, hence only when $N : [2, 5]$. I.e., all iterations were using the original central points every time and only changing/halving the radius of the 8 surrounding points, thus never performing the translation to the best found distortion region – unless the central point was coincidently the best distortion region. After a quick correction a second step/problem surfaced.

In a second step, we found that the corners central/focus points only changed when a new global best distortion was found. The problem here is that, while a global best distortion is always applicable to the corner $c0$, it does not implicitly mean that no other less optimal distortions were found and could had been therefore suitable to other corners. For example, lets suppose the global distortion is 100 and that after performing all of the $N = 1$ iterations for corner $c3$, only worser distortions are found, the best one of these was 101, then no corner central points were updated, thus maintaining their location for the $N = 2$ iteration. This only focus the new/same points (for $N = 2$) around the same region again, which makes no immediate sense.

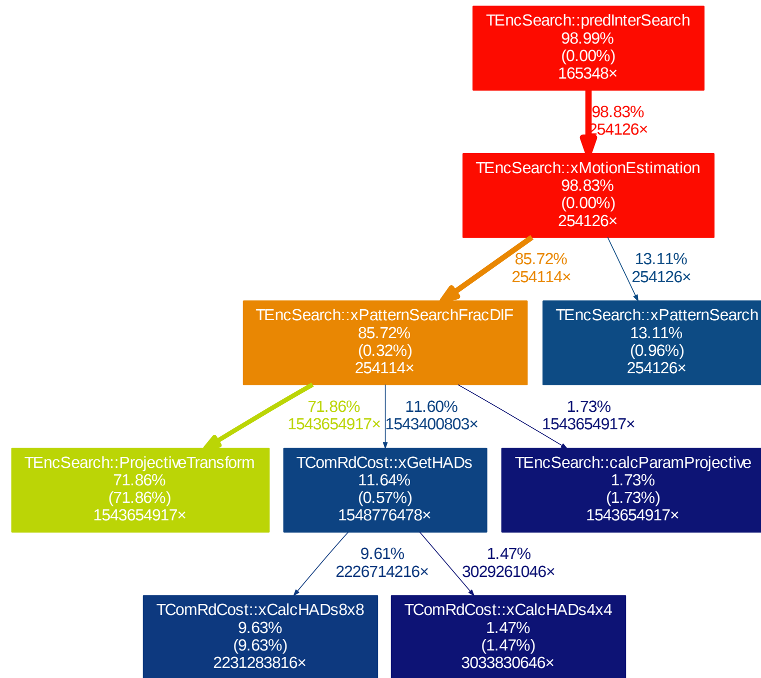
After some collaboration with Monteiro et al., the participating parties concluded that the expected/wanted behaviour was that all of the 4 corners would record their best local distortions (and corresponding central points) independently of the best global distortion. In this way, even if the found distortion was of only 101, the algorithm would reposition the central point and the new 9 generated points for the $N = 2$ iteration would have a higher chance of finding an even better distortion around the 101 region (possibly better than the best 100 in example, say 99).

As a last detail, the function in which the projective parameters are calculated was also altered. Because of its mathematical behaviour, it could sometimes generate a "Division by Zero" (DbZ) exception. As a quick fix, a simple condition was added before the calculation to ensure that the divisor would not result in zero. A later review by Monteiro et al. may be in order. This problem only arose when using the same calculations in the GPU environment. In the CPU environment the calculation results would be masked with a "Not a Number" (NaN) value, in which case further calculations with this value are discarded by the implementation since it yield very large distortions. However, in the GPU environment the kernel execution is actually stopped and the error is generated, not allowing the correct execution flow.

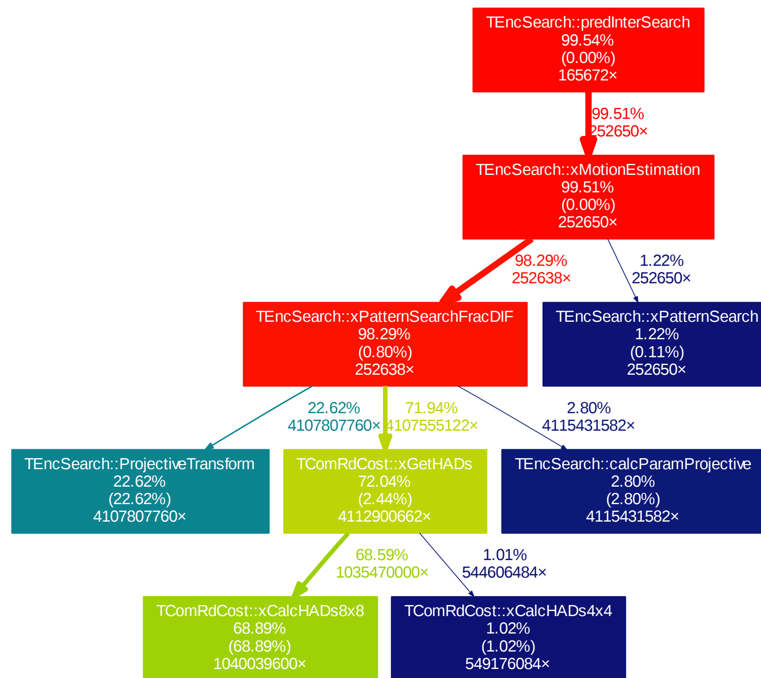
4.3 Profiling HEVC+SS+GT

Using the same methods as in Section 3.3.1, we started by visualysing a call-tree representation of the given sequential implementation to facilitate the absorption of the underlying data and comprehension of the GT module role in the time consumption plane.

Figure 4.3 is a visualization with [77] where nodes/functions that occupied less than 1% of the execution time were left out (almost 700 nodes). It has the same theme as explained previously, based on the node execution time percentage. Red nodes occupy almost all the execution time, orange, green and later blue nodes gradually consume less execution time percentage. Figures 4.3a and 4.3b are given so that the impact of a growing N may be seen in the functions call counter. The call-trees do not start at their root nodes so that they better fit the page. Here, 7 root nodes were cropped prior to the displayed one in each sub-image. In both cases, *xPatternSearchFracDIF* function is called around 254 and 252 thousand times – note how it is called less times for the $N = 5$ case. This function is the entry point of the GT module.



(a) $N = 1$



(b) $N = 5$

Figure 4.3: Partial call-graph for the execution of the HEVC+SS+GT-Sequential encoder on the PlaneAndToy image.

As a part of the inter-prediction motion search algorithm, the GT module is called from the *xMotionEstimation* function. Starting in function *xPatternSearchFracDIF*, the GT module begins looping through all point combination and applying possible zooms. To do this, 12 nested loops are defined, 3 loops are necessary per corner – one loop verifies and applies possible zooms and two inner loops iterate the different x and y point coordinates that exist in a given corner. Within this 12 loops – selecting 4 points, one per corner – the algorithm first calculates the projective transformation parameters in sub function *calcParamProjective* – which is a simple math-function routine – and later computes the transformation with function *ProjectiveTransform*. This consists in the actual transformation of a given block image by the computed parameters – 2 nested loop iterate over the image pixels. After the transformation, the generated/transformed image is compared with a reference block. This comparison is computed by the *xGetHADs* function. Depending on the image block size divisibility, this later function calls *xCalcHADs8x8* or *xCalcHADs4x4* or *xCalcHADs2x2* functions. For image blocks of sizes 8x8 or 16x8 or 16x16, etc., *xGetHADs* selects function *xCalcHADs8x8*. Particularly, for the later image block size of 16x16 the *xCalcHADs8x8* is called 4 times – since 4 8x8 matrices fit into a 16x16 block. But for a 8x4 block it would now call the *xCalcHADs4x4* function since the block is not divisible into 8x8 sub matrices but rather into 4x4. Note that the *xCalcHADs2x2* function is never called, therefore it is not included in Figure 4.3 call-graphs. These *xCalcHADs** functions are part of the HEVC implementation and simply called by the GT module. They calculate the Walsh-Hadamard Transformation for each sub matrix and sum all their elements thus resolving the final distortion value for that given image block.

For the $N = 1$ results, Figure 4.3a, the main GT module function – *ProjectiveTransform* – is called around 1.5 billion ($1.5 * 10^9$) times which implies that the distortion calculation function – *xGetHADs* – is also called around 1.5 billion times. Note how in this scenario the *ProjectiveTransform* function consumes 84% of the root *xPatternSearchFracDIF* function time – where the GT module begins – i.e. 71.86% of the total execution time.

On the other hand, in the $N = 5$ results, Figure 4.3b, both *ProjectiveTransform* and *xGetHADs* functions are now each called around 4.1 billion times. In this case, the *xGetHADs* function is now the most consuming function, occupying 72% of the runtime execution time. For the lower N value the *ProjectiveTransform* function is the main time consuming function and for larger N value the inner *xCalcHADs8x8* gains significance.

4.3.1 Base Reference Results

As a reference to the rest of this chapter, Table 4.1 shows the execution times registered for the pre-existing implementation (after the modifications referred in the previous section) with its variable N. For simplicity, because we already examined the available hardware in the previous chapter, we choose to only use Server 2 in this case study.

The encoded frame (Figure 2.2) of the HD PlaneAndToys sequence takes between 924 and 990 seconds to be processed by the base HEVC implementation. The mean ($\overline{x}$ of 30 executions) value is situated at 988 seconds, thus it was selected as the base result. This means that all the remaining time will be spent on average in the GT module. Immediately, for the selected frame in Table 4.1, the utilization of the GT module with $N = 1$ adds 6775 seconds to the algorithm execution time. And at its maximum, $N = 5$ adds a total of 6.95 days (roughly 167 hours).

Base HEVC Version	N	Execution Time	GT Module Time
without GT	–	988 (0h16m)	– (0h00m)
with GT	1	7763 (2h09m)	6774 (1h52m)
with GT	2	79331 (22h02m)	78342 (21h45m)
with GT	3	275145 (76h25m)	274156 (76h09m)
with GT	4	490876 (136h21m)	489887 (136h04m)
with GT	5	601604 (167h06m)	600615 (166h50m)

Table 4.1: Execution time results for the HEVC+SS+GT-Sequential encoder on the PlaneAndToy image without the GT module and with GT module with $N : [1, 5]$.

4.4 OpenMP

As a starting approach, to further understand the sequential implementation and devise parallelization approaches, we resorted to the previous explored API (Section 3.3) as a way to rapidly implement and test parallelization strategies before the GPU and multiGPU stages (which take longer to formulate). Hence, in this section we explain one parallelization that was relevant for the GPU stage.

4.4.1 Implementation

The sequential implementation consists of 12 loops, nested one after the other in function *xPatternSearchFracDIF*, were the 4 corners iterate over their possible transformation points (x and y coordinates – 2 loops per corner) and eventually zoom to a better region (1 verification loop per corner). Because the algorithm is highly iterative, with 12 nested loops, the parallelization could be implemented through almost any of them, but the data reduction or merge implementation would depend on the case in question. Several strategies were tested to evaluate the time used by parallelization mechanisms. This is relevant because, although the algorithm consumes a large amount of time, the inner-most loop may take almost no actual time but be called in fact a lot of times. This is the case. The last corner c3 loops produce $9 * N$ iterations for each combination of the other corners. In an extreme case, for a large enough block, it produces a maximum of $(9 * N)^4$ iterations per block (4100625 for $N = 5$). If the necessary work per cycle does not take more time on average than the necessary time to prepare and initiate the parallel API, no speedup will be producible.

Like for the previous algorithm (Section 3.3.2), the OpenMP-based parallelization is very straight forward. Since the sequential algorithm is already for-loop based, it was only necessary to apply the OpenMP directives. Although the directive conjunction "*omp parallel for*" would suffice for this situation, a division in "*omp parallel*" to prepare the region and further "*omp for*" to divide the existing work is preferable to better adjust allocated memory usage.

Having a basic parallel implementation, we further improved it by moving its critical section out of the 'for' region. The contents of the critical section represent the update of the calculated distortions (and salvage of the combining points that generated that distortion). To maintain the same algorithm results between executions, and reproduce the sequential implementation, the critical section is composed of a loop that iterates in order by the number of existing threads. Therefore merging each result one at the time, in the sequential order. This provided a small speedup because instead of having a maximum of $(9 * N)^4$ entries to the critical section in the inner-most loop of the 'for' region, there is now only 9 after its very end. In order to do this, we replicated several variables so that each thread would save a local backup of the final results, enabling their usage out of the 'for' region and copy/update over the global/main variables.

Since we are only using a maximum of 9 threads for the parallel region, some CPU cores are still available on the Server 2 machine. Attempts were made to add more

working threads but had no relevant success. Instead of working only over 1 corner, coalescing to 2 corners did not provide further speedups since the implementation complexity became too big. Further thread/work division on subfunctions *GetHADs*, *xCalcHADs8x8* and *xCalcHADs4x4* could not improve the algorithm since they provide too little calculation over the needed variable managements. Parallelization of the *ProjectiveTransform* function was left aside to keep up with the project schedule and objectives, specifically the GPU implementation.

4.4.2 Results

Table 4.2 presents results for two extreme cases: 1) a parallelization over the outer-most loop where a lot of computation exists; and 2) a parallelization over the inner-most loop where the smallest amount of computation per loop exists. Because the usage of an N ranging from $[2, 5]$ would only extend the amount of computation, the results only present the resulting time and speedup achieved with $N = 1$. In these two extreme cases, the parallelization is done over the 9 points of the $c0$ and $c3$ corners, respectively. This makes the maximum number of allowed threads 9. Because each OpenMP thread will start at different point combinations (independently of the case), after the 9 points are evaluated in parallel, it is necessary to merge the different results and update the global and local distortion variables.

OpenMP Case	Threads	Execution Time	SpeedUp
1 (outer)	1	7984	0.97
	2	4921	1.58
	3	3569	2.18
	4	4069	1.91
	5	3588	2.16
	6	3433	2.26
	7	3609	2.15
	8	3462	2.24
	9	2999	2.59

OpenMP Case	Threads	Execution Time	SpeedUp
2 (inner)	1	8288	0.94
	2	5618	1.38
	3	4341	1.79
	4	4556	1.70
	5	4584	1.69
	6	4614	1.68
	7	4687	1.66
	8	4687	1.66
	9	4677	1.66

Table 4.2: Execution time results (in seconds) for two implementation cases of the HEVC+SS+GT-OpenMP encoder on the PlaneAndToy image.

The presented table makes it evident that the OpenMP API has a significant overhead. With 1 thread, case 2 takes more time then case 1, since there are a lot more API calls produced for parallel computations. And since there is less work to do in case 2 (per loop), the achieved speedup upper limit is lower than in the case 1 experiment overall. With larger N values the speedups increases slightly. Alternative cases

between these 2 extreme implementations (in the outer-most and inner loops) only produced results between the present ones.

After assessing that a parallelization over the external/outer loops is preferable, we further improved the suggested case 1 by moving its critical section out of the 'for' region as referred earlier. Table 4.3 shows the attained speedups, including variable $N : [1, 5]$, with 9 threads, for the improved case 1. Both PSNR and bitrate values were maintained equal to the sequential version results.

N	Execution Time	SpeedUp
1	2228 (0h37m)	3.48
2	11001 (3h03m)	7.21
3	34773 (9h39m)	7.91
4	61407 (17h03m)	7.99
5	74676 (20h44m)	8.06

Table 4.3: Execution time results for the final HEVC+SS+GT-OpenMP encoder on the PlaneAndToy image and achieved speedups.

With an increase in the N value, each thread gains more and more workload. This causes the speedups to continuously increase towards the number of existing threads since the API overheads became gradually insignificant. But interestingly, since an increase in N produces an exponential growth in the workload, a speedup jump from 3.48 to 7.21 was registered simply because of the workload increment achieved by $N = 2$. Aiming for the more costly execution, this OpenMP implementation achieves a speedup of 8.06 with $N = 5$.

4.5 OpenCL-GPU

In order to produce a GPU implementation it came to reason that we would resort to the previously explored OpenCL API (Section 3.4). Previous to the acknowledgement of this work attempts were made to resort to the CUDA API but had no success.

The Open Computing Language (OpenCL) is already well reviewed in Section 3.4. Although the section aims for a OpenCL-CPU environment, both OpenCL-GPU and CPU were equally described since 'OpenCL' itself is normally associated with GPUs (thus made sense to base the CPU description on a correlated GPU analysis).

Because initial implementation took a considerable amount of time to run, benchmarking and result acquisition were confined to the GTX Titan Black GPU since it

is the fastest one. Side executions were made in other GPUs for comparison but were not fully benchmarked. Thus the results subsection 4.5.2 will mostly revolve around the GTX Titan Black GPU.

4.5.1 Implementation

In this subsection, we start by enumerating some relevant implementations prior to the final proposal and fully implemented one, so that the reader can comprehend the path and decisions that lead to the final implementation.

From the knowledge already obtained from the previous MMP implementations and the latest HEVC+SS+GT-OpenMP implementation, we started to devise possible algorithm expansions that would better suit a GPU environment, namely the warp and memory mechanisms. As a first attempt, we idealised a direct expansion of the OpenMP implementation, coalescing through all four corners. This would provide 9^4 distributable iterations throughout the GPU threads, but this method was soon discarded since no viable process of dividing the different iterations was found (and it did not take into consideration other values of N apart from $N = 1$). In the process of implementing it, we found out that the conditions which allow, or forbid, the calculation of the geometric transformation produced by a group of points were not always ideal. For a given block, 9 point combinations are always denied since they basically produced a translation and not a transformation – i.e. the combination of the x point of every corner, being $x : [1^{st}, 2^{nd}, .., 9^{th}]$, always produces the same original block image but translated by a small offset. An example of this is present in Figure 4.2, cycle 1, iteration 1. This would mean that 9 GPU threads would always have no actual workload. In addition, further review of the remaining conditions – i.e. if the transformation is within the image and HEVC-allocated memory buffer and if the projective parameters calculation would not result in a division by zero – showed that they occurred more often than it was expected. This made this group of conditions a potential hazard to the GPU kernel execution since it would constantly produce branching problems. We cropped part of the HD image to analyse the resulting blocks and branching decisions. From over 296 different image blocks, 245 more marginal blocks could not have geometric calculations at all, and only 34 actually had all combinations (except the 9 translation ones) computed. The remaining blocks had intermediate results. In a nutshell, this is by far the ideal solution, but it could be resolved by vectoring all actually possible point combinations prior to the actual geometric and distortion calculations.

As a second iteration, we realized that we could maintain the previous attempt by, instead of giving each thread an iteration, giving each warp an iteration. This way an entire warp engages the workload or does not work at all and moves to the next point combination. Now, each iteration would be composed, typically, by 32 threads. Since the inner *ProjectiveTransform* function is composed by 2 more loops, we could then dissolve the existing 32 threads into these loops. However, this approach soon became unimplementable since the complexity of implementing all these workload divisions and rearrangements prior to actual kernel launch consumed a considerable amount of time that limited the actual gain/speedup that one would expect to attain. Additionally, there were a lot of memory related problems still unsolved mostly because the way the threads would access the buffer was not aligned with the way the HEVC itself would, thus buffer replications and rearrangements would also have to be added into factor in the future. Still, this implementation would suffice if we had enough workload which is not the case. Here we still have to launch the kernels numerous times with little work and large overheads.

A new way had to be formulated. It is agreeable that the whole GT module should be integrated into the GPU and only the final point combination that generated the best distortion should be retrieved so that there would be no CPU-side reductions or further calculations required (thus no need for large midway memory transfers). It would also be deemed important to produce a solution that needed not to be pre-prepared for a specific memory arrangement or structure in order to produce an, at least, favourable memory access pattern. As a final relevant aspect, the solution should be ideally capable of overcoming the small workload gaps that eventually emerge when the conditions for the projective transformation are not met. All these prospects align with a modular, self-contained, multi-kernel GPU implementation. We had to rethink all the algorithm implementation like referred by Padua [105].

V0 Implementation

As an initial implementation, we started by simply dividing the theoretical knowledge of the GT module into modular parts (since the sequential implementation was very tightly bound to the existing 12 loop-based code). We selected 4 steps: *i*) marking/listing of valid point combinations; *ii*) calculation of the projective parameters; *iii*) the actual calculation of the projective; and finally, *iv*) the distortion calculations. To implement the marking/listing of valid point combinations we would need to create a dynamic, thread-safe, listing in the GPU side (with all its overheads) or simply allocate a buffer were we could write if a given combination is valid or not. Any of these would

provide the means to initially vectorize the problem at hand. The second method was selected due to its simplicity. By doing this it would also be preferable to introduce a secondary step to reduce this allocated buffer, so that it would only refer the valid point combinations. This way, future utilization of its information would not need to take into consideration the possible existence of non-valid point combinations every time. We then further developed and implemented this idea and realised that further subdivision of the previous steps would be preferred so that each kernel would require less registers and mechanics to correctly replicate the CPU sequential implementation behaviour. For instance, with smaller kernels, we would also need less barriers and be able to empower a more custom control over how certain parts of the algorithm are divided into different warps and threads without adding more complexity inside a given large kernel. In the end, the first compliant and functional implementation (for $N = 1$) was created with the following kernels (steps):

$k0$ Initiate/Clean buffers for the given block size. Some buffers needed to contain standard block-size-dependent information at the beginning. To write the initial data, a simple *clEnqueueWriteBuffer* sufficed.

$k1$ Mark valid point combinations. Each thread checks one combination at the time, and strides throughout the remaining possible combinations.

$k2$ Reduce/Compact results from previous kernel. The 'compact' keyword is used here because the intent is to shift/compact the results in the buffer so that they remain in order and the invalid ones disappear. To achieve this, only 1 thread was applied here.

$k3$ For each combination in the step 1 buffer, verify if valid projection parameters are generated (without the division by zero problem exposed earlier). If invalid parameters are generated we mark the given combination as invalid, otherwise save the generated matrix in a new buffer. Each thread verifies and calculates one combination at the time, and strides throughout the remaining.

$k4$ Reduce/Compact results from previous kernel as performed in step 2 since we may have found point combination that produce the division by zero exception. But this time we divide this step into 2 sub-steps: one compacting the buffer from step 1 again with only 1 thread; and a second step to compact the generated matrices so that the offset in their own buffer is aligned with the reduction made on the buffer from step 1. In this second compaction, we used 9 threads since the matrices have 3x3 elements.

$k5$ For each produced matrix, apply the projective transform function and save the generated transformation image block in a new buffer. To generate the transformation, we copy and adapt the *ProjectiveTransform* function to the GPU side.

Each thread processes one *ProjectiveTransform* loop at the time, and strides throughout the remaining.

$k6$ With all the previous generated buffers, calculate the achieved distortions and save the distortion-combination pairs. For the distortion calculation we simply copy the *GetHADs*, *xCalcHADs8x8* and *xCalcHADs4x4* functions to the GPU side. Each thread computes one distortion at the time, and strides throughout the remaining. Note that *xCalcHADs2x2* was also copied, although it is never called at runtime.

$k7$ Finally, iterate over all the saved distortions and simulate the sequential implementation behaviour, i.e. reduce and store the distortions that each corner would attain thus yielding the distortion that produced the best combination. To simulate the sequential corner local results only 1 thread was used.

In addition to the described kernels/steps, it is relevant to note that:

- When kernel $k1$ marks a point combination as valid, it is actually saving its points in the buffer, otherwise it stores the value zero (a *NODATA* flag). For example, in practice, by indexing a corner point from 0 to 8 (since there are 9 per corner), we can calculate an unique number X that describes the points selected over the 4 corners. This unique number is both used as the index were to save if the combination is valid or not and as the value to store at that index if the combination is valid. We applied the calculus function Equation 4.1 to obtain the unique number, were each i is a point index of a given corner c.

$$X = c0_i * 9^3 + c1_{i'} * 9^2 + c2_{i''} * 9^1 + c3_{i'''} * 9^0 \tag{4.1}$$

Given the function complexity and the knowledge about the problem we can then revert the generated unique number and retrieve the corner index variables that originated it originally by computing:

$$\left| \begin{array}{lcl} temp & = & X \\ c0_i & = & floor(temp/(9^3)) \\ temp & -= & c0_i * 9^3 \\ c1_{i'} & = & floor(temp/(9^2)) \\ temp & -= & c1_{i'} * 9^2 \\ c2_{i''} & = & floor(temp/(9^1)) \\ temp & -= & c2_{i''} * 9^1 \\ c3_{i'''} & = & floor(temp/(9^0)) \end{array} \right. \tag{4.2}$$

- By applying the just explained math, a thread from a future kernel, like $k3$, can retrieve the point combination from a single *int* data type even after a reduction, from e.g. $k2$, since the buffer is not index but value dependent – contrary to all other buffers, which are index dependent, since they need to match the values from this one.

- Because of the array/list like buffer produced by $k1$, all subsequent kernels iterate over it without any complex mechanisms to divide the workload. The same methodology was adopted in kernels $k3$ and $k5$ so that their buffered result indexes would pair with the $k1$ buffer values.

- As stated in the previous description listing, $k0$ was later rewritten to be a kernel instead of the *clEnqueueWriteBuffer* function call. We verified, at least for the available hardware, that the *clEnqueueWriteBuffer* function executions were taking more time to run than kernel $k1$ (which was unexpectable because of the buffer sizes – listed further on). We noticed that custom created kernels could produce the same effect in almost half of the time. Tables 4.4 and 4.5 show the cumulative times attained when running an *clEnqueueWriteBuffer* and when using kernels over the same buffers. These buffers are used to store the calculated distortions and x and y offsets to the points that generated those distortions. Kernel $k6$ saves its results to corner buffer *c3Dist*, which is then propagated by $k7$ to the other corner buffers and the best attained result is saved in buffer *res*. Corner $c3$ buffers do not include a *c3Coords* because the buffer generated by $k1$ (and further maintained by the remaining kernels) already provides the necessary information.

- From Table 4.4, buffers with the "Dist" suffix are cleaned with a MAX_INT value by kernel $k0$ because we are later storing distortions, were the lowest is better. And buffers with "Coords" suffix are initiated with the central points x and y coordinates of each corner. These later buffers allow kernels $k1$, $k3$ and $k5$ to compute where the actual point is located in the image block since $k1$ will only be storing the indexes of valid points. Later kernel $k7$ updates the suffixed "Coords" buffers so that the central points are moved to the best found distortion points.

For this implementation to fully work we still have to consider the exponential computations growth caused when increasing N. With the current description, we can only reproduce the GT module output when $N = 1$ (after executing all kernels once). But this is why we already included into $k7$ the ability to propagate the results from buffer $c3Dist$ to the remaining corner related ones (listed in Table 4.4). Since every kernel receives its own parameters and the execution of all kernels only produces one full-sequential-iteration for $N = 1$, we can then simulate the correct order of events,

clEnqueueCopyBuffer	Corner Buffer	Accumulative Time	Execution Times	Copy Size
	resCoords	0.003237	592	2
	resDist			2
	c0Coords	0.003251	592	9
	c0Dist			9*2
	c1Coords	0.003248	592	9*9
	c1Dist			9*9*2
	c2Coords	0.003212	592	9*9*9
	c2Dist			9*9*9*2
	c3Dist	0.002000	296	9*9*9*9
	Total	0.014948	2664	

Table 4.4: Execution time results (in seconds) for the initialisation of several corner related buffer by resorting to the *clEnqueueCopyBuffer* function that copies the data from other already existing buffers that contain standard data for a given block size.

Copy Kernels	Kernel	Accumulative Time	Execution Times	Kernel Geometry	Writes/Thread
	k0res	0.001300	296	<1.1>	1
	k0c0	0.001373	296	<9.9>	1
	k0c1	0.001447	296	<81.81>	1
	k0c2	0.001447	296	<32.736>	2
	k0c3	0.001528	296	<32.1696>	4
	Total	0.007125	1480		

Table 4.5: Execution time results (in seconds) for the initialisation of several corner related buffers by resorting to custom implemented kernels that copy the data from other already existing buffers that contain standard data for a given block size.

i.e. which buffers provide information and which store results, by swapping the buffers around in order to achieve the superior N results. Let us consider that '1 iteration' represents the execution of all $7+1$ kernels (in order), being the initial execution of $k0$ the '+1'. To achieve $N=1$, we only need 1 iteration, thus the CPU will have to launch the $7+1$ kernels. To achieve $N=2$ we would have to simulate the zooms that the sequential version produces and recompute the results. Because we compute all $N=1$ distortions in a first iteration, we can then rerun the kernels zooming only over corner $c3$, then again for corner $c2$, then $c2$ and $c3$ combined, then only $c1$, etc. to simulate the behaviour. This produces a total of 16 iterations, meaning $7*16+1$ kernel executions. This is possible because kernel $k7$ is responsible for the reduction/propagation of the results in the same way the sequential implementation would. Table 4.6 provides a complete view of the buffer swapping, cleaning/reset and reiteration needed to fulfil a $N=2$ execution. Note that for the objective $N=5$, a maximum of $7*625+1$ iterations per given image block is needed. With the table, we can observe the usage of the corner buffers referred in Table 4.4. To produce the simulation, we sometimes need to reset some of these buffers to their standard/original values (marked in column *Pre-Reset Buffers?*) to comply with the theoretical algorithm. To simplify/optimize the necessary

computations, we replicated all corner buffers so that one of the 2 groups would always retain the original values prepared initially by $k0$ (the '+1'). To differentiate, the group of buffers that retain the original values is named 'orig' and the buffers that cumulate values from each iteration are named 'accu'.

Corner L Sizes				Info about corners (Input)				Where to save new corner info (Output)				Pre-Reset Buffers?			
c0	c1	c2	c3	c0	c1	c2	c3	c0	c1	c2	c3	accu0	accu1	accu2	accu3
32	32	32	32	ori0	orig1	orig2	orig3	accu0	accu1	accu2	accu3				
			16	ori0	orig1	orig2	**accu3**	accu0	accu1	accu2	**TRASH**				
		16	32	ori0	orig1	**accu2**	orig3	accu0	accu1	**TRASH**	accu3				X
			16	orig0	orig1	accu2	**accu3**	accu0	accu1	**TRASH**	**TRASH**				
	16	32	32	orig0	**accu1**	orig2	orig3	accu0	**TRASH**	accu2	accu3			X	X
			16	orig0	accu1	orig2	**accu3**	accu0	**TRASH**	accu2	**TRASH**				
		16	32	orig0	accu1	**accu2**	orig3	accu0	**TRASH**	**TRASH**	accu3				X
			16	orig0	accu1	accu2	**accu3**	accu0	**TRASH**	**TRASH**	**TRASH**				
16	32	32	32	**accu0**	orig1	orig2	orig3	**TRASH**	accu1	accu2	accu3		X	X	X
			16	accu0	orig1	orig2	**accu3**	**TRASH**	accu1	accu2	**TRASH**				
		16	32	accu0	orig1	**accu2**	orig3	**TRASH**	accu1	**TRASH**	accu3				X
			16	accu0	orig1	accu2	**accu3**	**TRASH**	accu1	**TRASH**	**TRASH**				
	16	32	32	accu0	**accu1**	orig2	orig3	**TRASH**	**TRASH**	accu2	accu3			X	X
			16	accu0	accu1	orig2	**accu3**	**TRASH**	**TRASH**	accu2	**TRASH**				
		16	32	accu0	accu1	**accu2**	orig3	**TRASH**	**TRASH**	**TRASH**	accu3				X
			16	accu0	accu1	accu2	**accu3**	**TRASH**	**TRASH**	**TRASH**	**TRASH**				

Table 4.6: Visual representation of the buffer swapping iterations needed to achieve $N = 2$, including the indication of the corner L sized point boxes size and if a given cumulative buffer needs to be reset before the row/iteration happens.

In the Table (4.6), each line represents 1 iteration, all 16 fulfil the $N = 2$ computations. This table was produced for a block image of size 64x64, hence in the first *Corner L sizes* columns row all corner L sizes are 32. As a first iteration, we can read in the first table row that the kernels will receive information from the buffers that retain the original generated values ('orig') and that they will be saving their output (an update over the first set of buffers) to the cumulative buffers ('accu'). To simulate the previous described swapping behaviour, in the second iteration we place the *accu3* buffer of corner $c3$ as an input information buffer so that a zoom is simulated over that corner (thus halving the $c3$ L size). And we can also discard/trash the output, since we will not be zooming again (because we are aiming of the $N = 2$). In the third iteration, we are now zooming over the c2 corner. To do so, we need to set the $c2$ corner buffer *accu2* as an input information buffer. Like before, we can trash the output for corner c2 since we will not zoom a third time. Note that in this particular iteration, we have to reset the cumulative values for corner $c3$ back to their original values because we zoomed into the $c3$ region in the previous iteration. To point these kind of events, column *Pre-Reset Buffers?* marks them as needed. Sequentially, in the fourth row, we are now zooming over both $c2$ and $c3$ corners, i.e. after previously zooming $c2$, we now need to recompute a combined zoom over $c3$ with the resulting $c2$ zoom of the previous iteration 3. This happens with the halving of both corner $c2$ and $c3$. Like before, we will not need the generated outputs. The remaining table rows behave in the already described ways, producing all swapping combinations in an optimal sequence.

Because of the previous explanations, $k0$ kernel is now divided into 5 reset/cleaning sub kernels: 1 to initiate the buffer with all standard values before the actual computations (the '+1' kernel referenced earlier); and 4 to reset each corner buffers ('Dist' and 'Coords') individually – usage marked in Table 4.6, column *Pre-Reset Buffers?*.

At this point, and for the remaining implementation updates, the memory buffers utilized and their respective sizes are describe in Table 4.7. It presents the need for 52.228MiB of global allocated memory space where 0.235MiB retain the standard corner values used by kernels $k0$, and where 98% of the space is used to store the transformed images produce by kernel $k5$ and read by $k6$. Note that the *Pel* data type is implementation/hardware dependent, while in the HEVC+SS+GT it is defined as a *short*, in the OpenCL kernel – at execution time – it may be used as an *int* by the hardware thus doubling its size.

Name	Math	# Elements	Data		Space Used	
			Type	Bits	Bytes	MiB
resCoords_Orig	2	2	INT	32	8	0,000
c0Coords_Orig	9*2	18	INT	32	72	0,000
c1Coords_Orig	9*9*2	162	INT	32	648	0,001
c2Coords_Orig	9*9*9*2	1458	INT	32	5832	0,006
c3Coords_Orig	9*9*9*9*2*4	52488	INT	32	209952	0,200
resCoords	2	2	INT	32	8	0,000
c0Coords (Accu0)	9*2	18	INT	32	72	0,000
c1Coords (Accu1)	9*9*2	162	INT	32	648	0,001
c2Coords (Accu2)	9*9*9*2	1458	INT	32	5832	0,006
c3Coords (Accu3)	9*9*9*9*2*4	52488	INT	32	209952	0,200
resDist_Orig	2	2	UINT	32	8	0,000
c0Dist_Orig	9	9	UINT	32	36	0,000
c1Dist_Orig	9*9	81	UINT	32	324	0,000
c2Dist_Orig	9*9*9	729	UINT	32	2916	0,003
c3Dist_Orig	9*9*9*9	6561	UINT	32	26244	0,025
resDist	2	2	UINT	32	8	0,000
c0Dist	9	9	UINT	32	36	0,000
c1Dist	9*9	81	UINT	32	324	0,000
c2Dist	9*9*9	729	UINT	32	2916	0,003
c3Dist	9*9*9*9	6561	UINT	32	26244	0,025
k1OutBuffer	9*9*9*9+1	6562	INT	32	26248	0,025
k3OutBuffer	9*9*9*9+1	6562	INT	32	26248	0,025
k3ProjectParamBuffer	9*9*9*9*3*3	59049	DOUBLE	64	472392	0,451
k5OutBuffer (PiAux)	9*9*9*9*64*64	26873856	Pel	16	53747712	51,258
Cvalues	18*6+2+15*6+2	202	INT	32	808	0,001
Total		27069251			54765488	52,228

Table 4.7: Listing of buffers created in GPU environment with size and data type descriptions.

For future reference, we label the until now discussed implementation as the "v0" GPU implementation.

V1-MultiThreadReduction and Simplification Implementation

As previously described, kernel $k2$ and both $k4$ sub-steps only operate with 1 thread, and thus are very inefficient. As a first improvement to the proposed V0 implementation, we altered these kernels to take advantage of the GPU environment.

Instead of having only 1 thread going through the entire global buffer, we promoted parallelism by dividing the buffer in equal sized sub buffers and by making each thread to operate over only one sub buffer. To overcome the memory lag – the threads will have non-optimal memory access patterns – and provide some speedup, it would be ideal if the threads could place their sub buffer data immediately as in the expected output – avoiding a final reduction of the data from the different threads. In order to make all threads immediately copy/place their elements into the right final index – i.e. already in the final compacted order – each thread first counts the number of valid elements in their sub buffer and saves that information into a local memory buffer of size equal to the number of threads+1. This way, we can now calculate the offsets each thread must use to store their valid elements. This calculation also provides the exact number of valid elements contained in the end buffer, an information forwarded to the remaining kernels. As a final step, each thread copies the valid data to the calculated local buffer index/ranges over the existing data in the global buffer.

Because the global buffer size is well known (9^4), we only need a number of threads that enables an equally balanced workload. The possible divisors are: 1, 3, 9, 27, 81, 243, 729, 2187 and 6561. Selecting divisor 3 or 9 did not provide enough parallelism to compensate the new needed mechanisms and selecting 81 or bigger removes the possibility to fully take advantage of the local memory mechanisms. Thus, we selected 27 threads because it only uses 1 warp, making each thread have 243 buffer elements to process.

This logic was applied directly to $k2$ and $k4$-step1. Because in $k4$-step2 each element is actually composed of 3x3 matrices, we further parallelized the previously described logic to accommodate data strides inside the elements thus allowing to launch the kernel with $27 * (3 * 3)$ threads instead of only 27. The kernel still has a work group size of 27, but it is launched with 9 groups, one per each matrix element.

During the reimplementation of the referred kernels, as stated before, we came to understand that, because of the $k2$ calculations, the followed kernel launches ($k3$ to $k7$) could know the exact amount of existing data. By knowing this, they no longer strictly

need the reductions provided by $k4$. Therefore, the $k2$ calculus is saved and kernel $k4$ is removed. Additionally, by acquiring this information in the host/CPU side, we can also know if there was any valid data generated and only launch the kernels if needed.

Figure 4.4 provides a visualisation over the just described implementation with 3 threads and a buffer of size 15. As a final note, it is important to remember that the existing data must remain in the same order so that it can be later worked like in the Sequential implementation (by kernel $k7$). In the figure, buffer b is equally divided into 3 sub buffers of size 5. Then each thread counts how may valid elements exist, thread 2 verifies that there is no valid data in its sub buffer (coloured in red). After this, we calculate the offset each thread must apply in the final output buffer. Note that because thread 2 has no data to write, thread 3 also starts at thread 2 offset. Finally, each thread writes their valid data in the buffer. The remaining, untouched, buffer positions (coloured in grey) are no longer zeroed out (with the *NODATA* flag) since the next kernels will know how many valid elements exist and thus know when to stop processing the buffer.

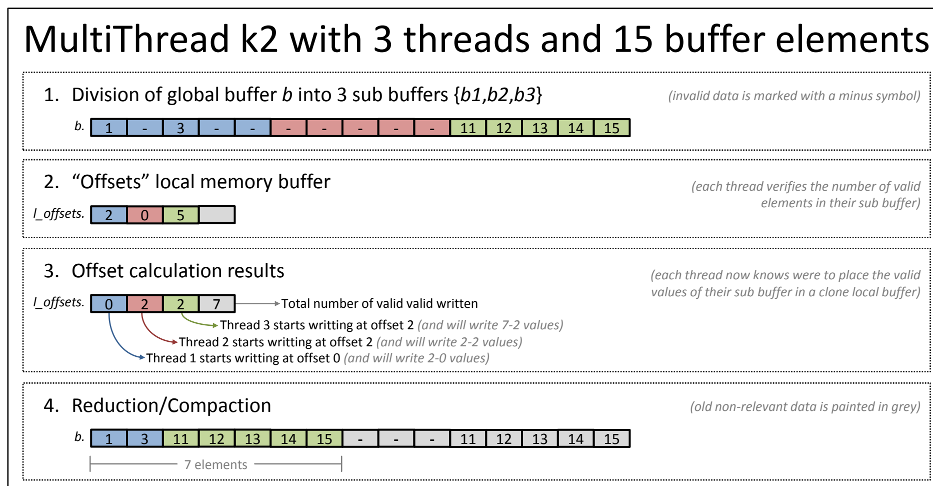


Figure 4.4: Illustration of the improved multi-thread $k2$ kernel working with 3 threads over a buffer of 15 elements.

V2-HAD Implementation

As a second improvement, we reimplemented kernel $k6$ since it was almost just a simple copy from the CPU HEVC+SS+GT Sequential implementation. The importance for a highly optimized Hadamard implementation is well acknowledged in several published works [106, 107, 108, 109, 110, 111], with authors reporting that the transform consumes most of the computing time. In the GT module, the HEVC+SS+GT Sequential implementation only computes Hadamard Transformations for matrices of 2x2, 4x4

and 8x8 elements. If larger matrices appear they are cut into smaller, equally sized, matrices and their distortion results are summed. Referring back to Table 4.6, note that all image block matrices in an iteration have the same size thus they will be all divided into the same equally sized matrices of size 8x8 or 4x4 or 2x2. For example, a image block matrix of size 32x32 would be divided into 16 smaller 8x8 matrices.

Along side with other transforms, the Walsh-Hadamard Transform (WHT) is widely used to perform some operations in computer vision related algorithms. For instance, the WHT has been used for feature selection in character recognition tasks [106]. Due to its complexity over existing resources, Carl and Swartwood [112] designed a hybrid special purpose computer, based on a recursive algorithm, to tackle the discrete WHT computation. The device used feedback to reduce the required summation junctions from $N \log_2 N$ to N. Later Huang and Chung [107] used the Walsh functions for separating similar complex Chinese characters. Each character is imaged 8 times by changing its size, position, and binary threshold value. More recently, with the advent of digital communications, the Hadamard transform has been used for other applications like data encryption [113], and image and video compression, as JPEG XR [92] and MPEG-4 AVC [114].

In the initial $k6$ kernel version, we gathered the information from the previous kernels, called the HEVC+SS+GT Sequential HAD implementation, and saved the results in a proper way so that $k7$ could easily use them. Because we used the Sequential implementation code, a lot of registers were needed since that code is optimized for a more powerful CPU core. Hence this first implementation of the kernel was very slow because of the register spilling and other inherent problems.

To correctly use the GPU manycore paradigm, we looked on how to improve the parallelism of the implementation and reduce the extreme amount of registers in use. For this purpose, we devised an implementation where kernel $k6$ is divided into 4 separate kernels: one to calculate the HAD distortion of matrices divisible into 8x8 sub matrices, another to calculate HAD distortions of matrices divisible into 4x4 but not by 8x8, a third to calculate HAD distortions of matrices divisible into 2x2 but not by 8x8 or 4x4, and a final fourth kernel to save the results in the proper alignment needed by $k7$. At runtime, we only call one of the new HAD kernels (depending on the image block size) and then call the fourth kernel to properly store everything.

At its core, the HEVC Sequential HAD distortion calculation functions are just extended implementations of the Fast Walsh-Hadamard Transformation (FWHT) [115]. The HEVC Sequential HAD distortion calculation functions first compute the difference

between the original image block and a projective transformed block (from $k5$), and only later apply the actual Walsh-Hadamard transformation to the computed difference so that the distortion sum can be calculate afterwards – a form of sum of absolute transform differences. Our new kernels needed to reproduce the same effects in parallel. Depending on the kernel/matrix size, a number of threads is selected to process an entire matrix. The number of used threads is implementation dependent. We first apply a stride over the matrix data to simulate the matrix division into 8x8 or 4x4 or 2x2 sub matrices depending on the case/kernel. Then, the group of threads responsible for that matrix computes the difference between the original image and the projection and calculates the HAD transformation – one sub matrix at the time. At the same time, these threads also sum all matrix elements to attain the sub matrix distortions. This way, the final associated distortion value is saved and the threads move on to the next matrix. Several matrices are processed at the time using multiple GPU thread groups.

To implement our parallel OpenCL-HAD kernels, we started by understanding the FWHT. Figure 4.5 provides an example visualization of the transformation of an 8 element vector. The Fast WHT is a divide and conquer algorithm that recursively breaks down a WHT of size N into two smaller WHTs of size $N/2$. Like in the figure, half of the computations are additions and the other half subtractions. Relevant to the FWHT understanding is the definition of butterflies, also present in the figure.

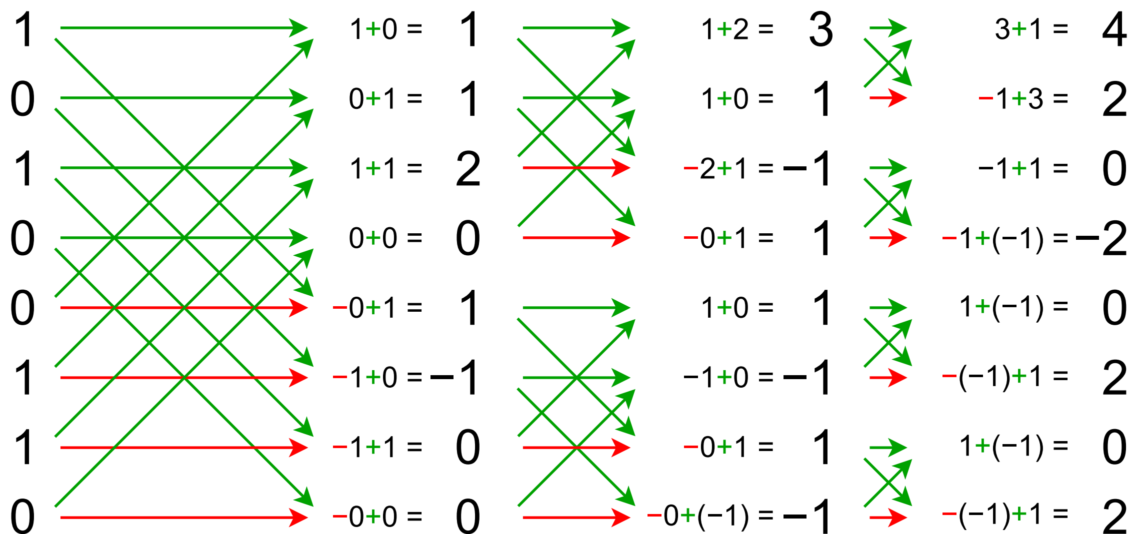


Figure 4.5: An illustration of the Fast Walsh-Hadamard Transform application over an input vector (1,0,1,0,0,1,1,0) of size 8. — adapted from [116]

In the original context of Fast Fourier Transform (FFT) algorithms [117, 118], a "butterfly" is a portion of the computation that combines the results of smaller Discrete Fourier Transforms (DFTs) [119] into a larger DFT. The name "butterfly" comes from the shape of the data-flow diagram in the radix-2 case, as described in [120]. One of the

earliest occurrences of the term can be found a 1969 MIT technical report [121, 122]. More commonly, the term "butterfly" appears in the context of the Cooley-Tukey FFT algorithm [123], which recursively breaks down a DFT of composite size $n = rm$ into r smaller transforms of size m where r is the "radix" of the transform. These smaller DFTs are then combined via size-r butterflies, which themselves are DFTs of size r (performed m times on corresponding outputs of the sub-transforms) pre-multiplied by roots of unity (known as twiddle factors). These notions are important so that we can further explain our implementation.

After the described understanding of the FWHT, we reviewed a sequential FWHT implementation in order to convert it to a parallel FWHT-GPU implementation so we can later tailor it to fit the 8x8, 4x4, and 2x2 kernel demands. That said, note Algorithm 1, a sequential single-core single-thread FWHT-CPU implementation definition. There, the sequential FWHT implementation iterates though strided butterfly stages that cycle through sub vectors of 2∗stride size elements. In turn, these stages iterate a butterfly within the sub vector and calculate the correlated addition and subtraction. This defines the radix-2 FWHT algorithm. In addition, we have included a sum reduction so that we attain what is known as the image block distortion in HEVC. This routine only calculates one transformation per call. Note that the transformations are saved in place, directly in the input buffer.

For the parallel FWHT-GPU general implementation, we devised a procedure/kernel similar to the one provided by the NVidia code samples of the CUDA SDK [124]. The GPU side of the code receives a buffer filled with matrices of a known size that are to be transformed. To provide some Instruction Level Parallelism (ILP), we spread the workload throughout GPU threads by using radix-4 calculations instead of the radix-2 used in Algorithm 1 and do a final radix-2 calculation if necessary. To further promote performance, we first copy a given matrix to the local memory in parallel and compute over it, instead of looping over the slower global memory input buffer. Because we use local memory, there is a limit to the matrix size that the kernel can compute. For large matrices, we can simply remove the local data copy and temporarily work directly over the input buffer (just like in the sequential version). Similarly to the sequential implementation, we apply a parallel reduction of the given matrix at the end and save the sum to an output buffer. In the parallel part of the execution, the two first loops of Algorithm 1 are coalesced, and the workload from the third is distributed throughout the threads of the GPU. Because of the way the workload is distributed, the number of threads needed per matrix is given by: $(A * B)/4$. Inevitably, because large matrices may require more threads than a GPU warp as to offer (typically, a warp has 32 threads), we added local memory barriers so that the reads and concur-

Algorithm 1 Calculate Fast Walsh-Hadamard Transform, radix-2

Input: matrix $\mathbf{D}^{m \times n}$, of data type T, with $m * n = N$ matrix elements, being m and n multiples of 2
Output: data type T value

```
 1: stride = (N)/2
 2: while stride ≥ 1 do                                ▷ butterfly stride statges
 3:     base = 0
 4:     while base < N do                              ▷ subvectors
 5:         f = 0
 6:         while f < stride do                        ▷ butterfly index
 7:             D0 = D[base + f + 0]
 8:             D1 = D[base + f + stride]
 9:             D[base + f + 0] = D0 + D1
10:             D[base + f + stride] = D0 − D1
11:             f += 1
12:         end while
13:         base += 2 ∗ stride
14:     end while
15:     stride /= 2
16: end while
17: sum = 0
18: for each element Di ∈ D do                         ▷ Sum reduction
19:     sum +=ABS(Di)
20: end for
21: return sum
```

rent stores are maintained in the right order. This is necessary because a GPU only ensures instruction level lockstep within each warp independently. In summary, in each transformation-loop each thread is computing the equivalent to 4 sequential loops (radix-4) from Algorithm 1 at the time. For example for a 32x32 matrix, only 5 loops per thread are needed to complete the transform, instead of the 5120 loops required by the sequential version. Note that the described loop-proportion changes slightly for odd sized matrices such as 32x16. Here, each GPU thread loops 4 radix-4 iterations and then 2 extra radix-2 loops are needed, while the sequential version only needs 2304 (half) iterations.

Once the described general FWHT-GPU implementation of radix-4 was done, we tailored it to fit our needs in the $k6$ 3 sub step kernels. We applied it almost directly to the 8x8 and 4x4 kernels and reverted back to a radix-2 implementation for the 2x2 kernel. A special radix-8 implementation for the 8x8 sub kernel was thought but rejected since it made each thread compute more work and enable the possibility of not using all GPU available threads – thus being slower in certain cases, and rarely faster than the radix-4 implementation.

In summary, in the V2 implementation we changed the $k6$ kernel into 4 sub kernels, with 3 sub kernels operating in the same way as the HEVC HAD functions but in parallel, and a later fourth behaving as the original $k6$ without having to do the HAD calculations. These 3 sub kernels for the Walsh-Hadamard Transform are very alike and differ only in the matrix sizes they may compute. They start by computing the difference between a reference image block matrix and a projective matrix from $k5$ into a *diff_matrix* matrix and then apply the Walsh-Hadamard Transform to every sub matrices that the *diff_matrix* may be divided into on the given kernel, that is, 8x8 or 4x4 or 2x2 sub matrices, respectively and in this order of preferred.

V3-MultiThreadK7 Implementation

Our last major improvement is the reimplementation of kernel $k7$. Initially, in order to quickly simulate the Sequential implementation, we implemented a kernel that would go through a buffer with all the results and reduce them to corner buffers that simulate the Sequential implementation run. This however, only worked with 1 thread. Up to kernel $k6$ buffers *c3Dist* and *c3Coords* (from Table 4.7) are built with all resulting information of an iteration of the GT module, i.e. all distortion values and corresponding point coordinates. In order to simulate the zooms (Table 4.6), like in the Sequential implementation, kernel $k7$ has the important task of reviewing all these results and store them in the remaining buffers – *c2Dist* and *c2Coords*, *c1Dist* and *c1Coords*, and *c0Dist* and *c0Coords* – so that we know which point will be the central point were the zoom is applied in the next iteration. Instead of going through all *c3* information only once with only 1 thread that stored the information in the right places all that once, we formulated an implementation which would go through the *c3* information 4 times – one for each corner reduction – but with as many threads as possible. For this purpose, we proceed as follows:

1. Go through *c3* results, 9 points at the time, and select the index of the best/lowest distortion value. Since the buffers are ordered, 9 sequential buffer values correspond to the results of each 9 points a corner has – Figure 4.1.
2. Save each best distortion index of the previous step in a local buffer and update the next corner buffers – *c2* – with the selected best distortions. By doing this, we reduce the data by 1/4.
3. Revisit each 9 saved indexes of the previous local buffer, and select the index with the best/lowest distortion value associated. This simulates the same behaviour as in step 1 but for corner *c2*.
4. Repeat step 2 but store/update over *c1* corner buffers instead.

5. Repeat step 3. This time the local buffer contents will simulate the reduction for corner *c1*.
6. Repeat step 2 but store/update over *c0* corner buffers instead.
7. Repeat step 3. This time the local buffer contents will simulate the reduction for corner *c0*.

With the just described steps, the kernel is now prepared to use multiple threads at once. We selected the maximum allowed sizes (but smaller values are also possible):

- Steps 1 and 2 use $9 * 9 * 9$ threads to process $9 * (9 * 9 * 9)$ distortions.
- Steps 3 and 4 use $9 * 9$ threads to process $9 * (9 * 9)$ distortions.
- Steps 5 and 6 use 9 threads to process $9 * (9)$ distortions.
- Finally, step 7 uses only 1 thread to process 9*(1) distortions.

Other optimizations took place during the implementation process – like dynamic occupancy calculation, and pre-prepared data structures – but are not as essential as the ones described in this document.

4.5.2 Results

Table 4.8 presents the implementation evolution since the first fully functional experiment – V0 – to the final optimized version – V3. Note in the table that some information is coloured grey to facilitate the visualization of the more relevant information in black. In addition to the 4 incremental implementations referred in this section, we also included 2 steps between the V0 and V1 implementations. A first step (*pre K4 removal*) to show the impact of the $k2$ and $k4$ transformations. And a second step (*using K2 count*) to display the small improvement the $k2$ calculations provided in kernels $k3$, $k5$ and $k7$. We selected executions with $N = 5$ for this table to enlarge the impact each implemented version produces. GPU results for $N = 1$ could be confined to as little as 4 minutes thus making implementation differences unnoticeable.

From the table, we can see that implementation V0 already provides a significant improvement over the Sequential implementation (Section 4.3.1) with only 15 hours and 30 minutes (*55822.55*) of GPU time instead of the +150 hours presented in Table 4.1. Note that the GPU time in the V0 implementation is not representative of the whole GT module time. Only later was the CPU/host simplified. In this first complete

implementation, several kernels remained non-optimized since we were first aimed for a functional implementation. With the parallelization of kernels $k2$ and $k4$, version V1 immediately halves the runtime to 6 hours and 28 minutes (*23301.19*). In a first step, we significantly improve $k2$ and $k4$ kernel performances with a speedup of 6.5, and 10.5 and 8.8 respectively. Later, we completely remove the need for a $k4$ reduction kernel since in the process of the $k2$ parallel calculations we attain information that compensates the need for a second reduction. This whole step shrunk 551.5 minutes of GPU computations (*6904.70+11688.35+14500.80* seconds) to only 17.3 minutes (1036.15 seconds). At this point, the most costing kernels are $k5$ – which is already optimized from the beginning – and kernels $k6$ and $k7$ which remain non-GPU-efficient. In our V2 implementation, we finally produce our own GPU distortion calculation functions instead of using the HEVC mono-thread implementations. In doing this, we further increased the algorithm performance by dividing the $k6$ kernel into sub kernels that yield a runtime reduction from around 60.9 minutes (*3654.54* seconds) to 29.7 minutes (*1702.28+30.50+0.00+49.84* seconds). Finally, in implementation V3 we effectively reconstruct kernel $k7$ into 4 parallel reductions that boosted its performance by 58.8 times from version V2 to V3.

Implementation	V0	V1-K2 (pre K4 removal)	V1-K2 (using K2 count)	V1-K2	V2-K6	V3-K7
K0	5.07	3.04	2.82	2.93	2.60	2.52
K1	56.54	57.63	56.87	56.37	53.12	55.70
K2	6904.70	1052.96	1040.49	1036.15	1029.32	1040.52
K3	293.23	285.89	95.25	92.07	88.30	91.64
K4p1	11688.35	1106.84	1106.84	–	–	–
K4p2	14500.80	1640.40	1640.40	–	–	–
K5	11909.52	11498.70	10808.84	12801.91	9643.90	10109.68
K6p1n8	3660.70	3651.69	3654.54	3642.07	1702.28	1848.17
K6p1n4					30.50	28.27
K6p1n2					0.00	0.00
K6p2					49.84	52.72
K7	6803.63	6603.27	5676.44	5669.69	5680.61	101.66
Total	55822.55	25900.41	24082.51	23301.19	18280.48	13330.88

Table 4.8: Execution time results (in seconds) for the HEVC+SS+GT-OpenCL encoder kernels $k0$ to $k7$ over the significant implementation iterations with the PlaneAndToy image and $N = 5$.

Contrarily to the HEVC+SS+GT-OpenMP implementation, where computations remained in the CPU, in this GPU implementation some floating-point operations slightly changed the final results thus producing a slightly different (nevertheless negligible) PSNR value. With Table 4.9 we complement the previous displayed results by adding execution times for $N = 1$. As stated previously, with $N = 1$ the final implementation only needs around 4 minutes to outturn all computations. In Table 4.9, we included both the overall execution time as well as the proportion of the time that the

GPU kernels utilize. Aiming for the more costly execution, our final OpenCL-GPU implementation outruns the Sequential code by 162.9 hours when $N = 5$. This is, we successfully achieved more practical runtimes with speedups of 39.94 when running the GT module to its fullest ($N = 5$). But since we only optimized for the GT module itself, part of the execution time – the HEVC original part – remains the same. Comparing back to Table 4.1, we can calculate that the GT module actually suffered a speedup of 45.05 (when $N = 5$), that is, a reduction of 163.1 hours.

Implementation	N	Execution Time		SpeedUp	
		Overall	GPU	Overall	GPU
V0	1	4356 (1h12m)	2906 (0h48m)	1.78	2.33
V1-K2	1	1842 (0h30m)	737 (0h12m)	4.21	9.19
V2-K6	1	1800 (0h29m)	677 (0h11m)	4.31	10.01
V3-K7	1	1360 (0h22m)	250 (0h04m)	5.71	27.15
V0	5	134838 (37h27m)	55823 (15h30m)	4.46	10.76
V1-K2	5	47827 (13h17m)	23301 (6h28m)	12.58	25.78
V2-K6	5	20023 (5h33m)	18280 (5h04m)	30.05	32.86
V3-K7	5	15064 (4h11m)	13331 (3h42m)	39.94	45.05

Table 4.9: Execution time results (in seconds) for the HEVC+SS+GT-OpenCL encoder on the PlaneAndToy image and achieved speedups when $N = \{1, 5\}$ over the significant implementations.

4.6 OpenCL-Multi-GPU

Throughout the previous single-GPU implementation (Section 4.5), we devised several modular and self-contained kernels that operate in a specific order. Additionally, by swapping around the kernel data buffers, we successfully simulated the original sequential permutations need – refer to as "swapping iterations" (Table 4.6).

In this section we propose a parallelization extension over the previous HEVC+SS+GT-OpenCL implementation (Section 4.5.1) on multiple GPUs.

4.6.1 Implementation

Each time the GT module is called a specific order of events needs to follow. Namely, the simulated iterations order that cannot be rearranged or parallelized since they are entirely dependant on one another. Likewise, internally, the GPU kernels that (mainly) compose each iteration also need to operate on their given order. But the

data within these kernels is totally independent from element to element inside each buffer. Therefore, as a multi-GPU implementation proposal, we decided to divide the buffer data across as many GPUs as possible and replicate the implementation call stack across each one, but for only part of the data.

Depending on the number of detected GPUs, the algorithm first defines the stride size that is to be applied so that each GPU receives an equal amount of data to process. Each 'data' element is a point combination – where 9^4 exist per iteration. In the single-GPU implementation, kernel $k1$ calculates all point combinations that may occur and that are to be processed by the remaining kernels – $k2$, $k3$, $k5$, $k6$ and $k7$. To put it into practice, only kernel $k1$ needs to be accommodated with the data stride. At this point, it is only necessary to replicate the kernel buffers (Table 4.7) and launch logics so that all detected GPUs are used. Additionally, the original buffer sizes may be altered since each GPU only processes part of the data, hence only needing part of the buffers – but not the final buffers for kernel $k7$. Because the data is divided across multiple GPU memories, it is necessary to regroup them back into one GPU before calling $k7$ so that it is possible to process the final results for that iteration. No later kernel adaptations or further alterations need to occur in the GT module core algorithm.

As a last detail, in multi-GPU environments, there are two ways in which a given host binding can support multiple devices: with a single context across all devices and one command queue per device; or one context and command queue per device. The process of utilising multiple devices for our computations is not done automatically by the binding when new devices are detected. Nor is it possible for it do so. Doing this requires active thought from the host programmer. When using a single device one sends all kernel invocations to the command queue associated with that device. In order to use multiple devices we must have one command queue per device either sharing a context or each queue having its own context. Then, we must decide how to distribute our kernel calls across all available queues – which in this case is fairly simple. We want to replicate all invocations but with different data. We initially opted to use a single context because of it simplicity but later changed it to one context per GPU to perform multithread experiments on the hardware.

4.6.2 Results

These experimentations were needed because, although we replicated all buffers and there are no dependencies between different GPUs (except the data synchronization before $k7$), the algorithm ran sequentially, one GPU at the time, never truly in parallel. Overall, the results produced by the multi-GPU implementation were very similar to the single-GPU results – only increasing the total execution time by 1 or 2 minutes and producing the same PSNR and bitrate results. With the exception of kernel $k7$, when using 2 GPUs, all kernel times were reduced to around 53% of the initial single-GPU runtime. Because there is the need to merge the results, kernel $k7$ is ran by only 1 GPU. Additionally, since the multi-GPU execution was not running in parallel, we subdivided each GPU workload into two streams (two command queues) in order to make the GT module perceive that there were 4 GPUs available. With this simulation, we aimed to verify the behaviour of the algorithm with a less common amount of GPUs – because our desktop servers could only take up to 2 GPUs at the time. The output results stayed the same and each stream took only 26.3% of the original single-GPU time. This means that the algorithm would also behave correctly with a larger number of GPUs – namely 4.

We tried expanding the implementation with multiple threads in the host CPU – resorting firstly to OpenMP and then to PThreads – but the execution order stayed sequential. Single thread testing showed, even if we set all kernel parameters beforehand and later called function *clEnqueueNDRangeKernel*, that the kernel launch function only returned after the kernel computations terminated – thus incapacitating the paralellization. Just as a precaution we also duplicated the host side buffers in the event that they were being locked to a GPU.

We also executed both OpenCL and CUDA SDK multi-GPU samples to validate if the servers hardware enabled code paralelization over multiple GPUs. But the observations stayed the same, the host execution launches the kernels sequentially and only after the previous launch terminates its computations. Additionally, using the OpenCL profiling flag, we verified that execution from different GPUs were effectively not overlapping.

In summary, we created a working multi-GPU implementation, maintaining the desired output, but did not actually utilize multiple GPUs in parallel – with overlapping kernel executions – in the available hardware.

4.7 Conclusions

Although very challenging at first, after some modifications to the sequential code, we successfully formulated and implemented a parallel version of the Geometric Transformations module – in both OpenMP and OpenCL. For the targeted extreme case of $N = 5$, initial execution measurements pointed to almost a 7 days execution time, and more gentle times of only 2 hours for $N = 1$. With the non-intrusive OpenMP for CPU we accelerated the module calculations by $8\times$ for the $N = 5$ case, which gradually slowed to around $7.2\times$ when reaching the $N = 2$, and by $3.5\times$ in the $N = 1$ case. This direct parallelization of the existing GT module loop yield an upgrade from 7 days to below 21 hours on the extreme target case.

Within this chapter, we also successfully presented a possible reimplementation of the GT module so that more parallelization is achievable. We separated the complex-12-loop-dependent cycles into self-contained sequential iterations that could be inner parallelized by several steps we implemented through kernels for the GPU environment. After several optimizations, such as the Walsh-Hadamard Transform parallelization that now consumes less than half of its initial runtime, we obtained an OpenCL-GPU implementation that speeds up the original GT module by 45 fold. This is almost a $40\times$ speedup for the whole HEVC algorithm, taking down the original 7 days to about 4 hours, while maintaining the same expected results.

To complement the presented work we also promoted an additional increment by utilizing multiple GPUs. The implementation effectively divided the workload but the hardware/OS did not seem to be able to take advantage of the multi GPU environment. We expect a 2/5 time reduction when using 2 GPUs, but were unable to confirm. It is noted here that we may require future work in this field.

Chapter 5

Conclusions

In this work we reviewed and proposed several implementations for both the MMP and the HEVC+SS+GT performance challenges. As intended, we were able to develop both OpenMP and OpenCL CPU-driven implementations for the MMP case study. We have also restructured the Geometric Transformations module with OpenCL for GPUs, in such way that the algorithm execution takes only a few hours instead of a week. In addition to the proposed case studies we also reviewed the associated energetic consumption of each MMP implementation achieved by the previously discussed hardware. Based on these observations we concluded that not only the decrease in execution time linearly decreases the energetic consumption, but also verified that multicore code yields a more energetic execution efficiency.

We were able to demonstrate that computer CPUs have a lot of computational resources to offer that are usually unexplored and that can be easily tailored by programmers through software frameworks such as OpenMP and OpenCL. In both case studies, the direct usage of the OpenMP API resulted in relevant speedups.

For the current hardware technology the future is multicore, and all GPU implementations showed how they handle it better. Apart from the nuances in both memory schemes and differences between CPU and GPU programming, OpenCL modular/kernel implementations always tend to be more optimal since they utilize multiple cores at ease. However, ot demands for a more careful implementation, requiring a better algorithmic planning.

As future work we suggest reviewing both the possibility of a HEVC+SS+GT multiplatform implementation, using both GPU and CPU instead of multiple GPUs since it is a more common setup, and a MMP-CUDA multi-GPU implementation.

Bibliography

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

[2] T. M. Ribeiro, "CUDA-MMP - Codificação de imagens com sistemas com múltiplos núcleos," Master's thesis, Instituto Politécnico de Leiria, 2016.

[3] J. F. C. Silva, "OpenCL-MMP: codificação de imagens com sistemas com múltiplos núcleos," Master's thesis, Instituto Politécnico de Leiria, 2015.

[4] C. Conti, P. T. Kovács, T. Balogh, P. Nunes, and L. Ducla Soares, "Light-field video coding using geometry-based disparity compensation," in *3DTV-Conference: The True Vision-Capture, Transmission and Display of 3D Video (3DTV-CON), 2014*. IEEE, 2014, pp. 1–4.

[5] R. J. Monteiro, N. M. Rodrigues, and S. M. Faria, "Geometric transforms and reference picture list optimization for efficient disparity compensation," in *Image Processing Theory, Tools and Applications (IPTA), 2015 International Conference on*. IEEE, 2015, pp. 370–375.

[6] M. B. De Carvalho, E. A. Da Silva, and W. A. Finamore, "Multidimensional signal compression using multiscale recurrent patterns," *Signal Processing*, vol. 82, no. 11, pp. 1559–1580, 2002.

[7] N. M. M. Rodrigues, "Multiscale recurrent pattern matching algorithms for image and video coding," Master's thesis, Faculdade de Ciências e Tecnologia da Universidade de Coimbra, 2009.

[8] E. H. Adelson and J. R. Bergen, *The plenoptic function and the elements of early vision*. Vision and Modeling Group, Media Laboratory, Massachusetts Institute of Technology, 1991.

[9] A. Gershun, "The light field," *Journal of Mathematics and Physics*, vol. 18, no. 1, pp. 51–151, 1939.

[10] M. Faraday, "LIV. Thoughts on ray-vibrations," 1846.

[11] L. McMillan and G. Bishop, "Plenoptic modeling: An image-based rendering system," in *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques.* ACM, 1995, pp. 39–46.

[12] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobb's journal*, vol. 30, no. 3, pp. 202–210, 2005.

[13] T. Moseley, D. Grunwald, D. A. Connors, R. Ramanujam, V. Tovinkere, and R. Peri, "Loopprof: Dynamic techniques for loop detection and profiling."

[14] B. Raghavan and J. Ma, "The energy and emergy of the internet," in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks.* ACM, 2011, p. 9.

[15] A. Hooper, "Green computing," *Communication of the ACM*, vol. 51, no. 10, pp. 11–13, 2008.

[16] T. NVIDIA, "K1: A New Era in Mobile Computing," *NVIDIA White Paper*, 2014.

[17] E. Upton and G. Halfacree, *Raspberry Pi user guide.* John Wiley & Sons, 2014.

[18] C. Nvidia, "NVIDIA's next generation CUDA compute architecture: Kepler GK110," Technical report, 2012.[28] http://www. top500. org, Tech. Rep., 2012.

[19] S. Variable, "A multi-core CPU architecture for low power and high performance," *Whitepaper-http://www. nvidia. com*, 2011.

[20] D. Chheda, D. Darde, and S. Chitalia, "Smart Projectors using Remote Controlled Raspberry Pi," *International Journal of Computer Applications*, vol. 82, no. 16, 2013.

[21] F. Kaup, P. Gottschling, and D. Hausheer, "PowerPi: Measuring and modeling the power consumption of the Raspberry Pi," in *Local Computer Networks (LCN), 2014 IEEE 39th Conference on.* IEEE, 2014, pp. 236–243.

[22] D. Saupe, R. Hamzaoui, and H. Hartenstein, *Fractal image compression: an introductory overview.* Univ., Inst. für Informatik, 1997.

[23] S. T. Welstead, *Fractal and wavelet image compression techniques.* SPIE Optical Engineering Press (Bellingham, WA, 1999.

[24] M. T. Inc., "Power Monitoring - Power Monitoring - Utility Metering Solutions," http://www.microchip.com/design-centers/utility-metering-solutions/power-monitoring/overview, 2014, accessed: 18-02-2015.

[25] ——, "MCP39F501 Single-Phase, Power-Monitoring IC with Calculation and Event Detection," http://ww1.microchip.com/downloads/en/DeviceDoc/20005256A.pdf, 2013, accessed: 18-02-2015.

[26] ——, "MCP39F501 Power Monitor Demonstration Board User's Guide," http://ww1.microchip.com/downloads/en/DeviceDoc/50002240A.pdf, 2014, accessed: 18-02-2015.

[27] M. Larabel and M. Tippett, "Phoronix Test Suite - Linux Testing and Benchmarking Platform, Automated Testing, Open-Source Benchmarking," http://www.phoronix-test-suite.com/, accessed: 25-07-2016.

[28] N. Ranganathan and S. Henriques, "High-speed VLSI designs for Lempel-Ziv-based data compression," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 40, no. 2, pp. 96–106, 1993.

[29] I. Pavlov, "7z Format," http://www.7-zip.org/7z.html, accessed: 25-07-2016.

[30] "7-Zip LZMA Benchmark," http://www.7-cpu.com/, accessed: 25-07-2016.

[31] B. T, "Parallel Memory Bandwidth Benchmark / Measurement," https://panthema.net/2013/pmbw/, 2013, accessed: 10-09-2015.

[32] J. W. Smith and A. Hamilton, "Massive affordable computing using ARM processors in high energy physics," in *Journal of Physics: Conference Series*, vol. 608, no. 1. IOP Publishing, 2015, p. 012001.

[33] G. Wrigleya, R. Reed, and B. Mellado, "Memory benchmarking characterisation of ARM-based SoCs," *Computer*, vol. 7, no. 3, pp. 607–617, 2015.

[34] D. Loghin, B. M. Tudor, H. Zhang, B. C. Ooi, and Y. M. Teo, "A performance study of big data on small nodes," *Proceedings of the VLDB Endowment*, vol. 8, no. 7, pp. 762–773, 2015.

[35] S. Gottlieb and S. Tamhankar, "Benchmarking MILC code with OpenMP and MPI," *Nuclear Physics B-Proceedings Supplements*, vol. 94, no. 1, pp. 841–845, 2001.

[36] E. Dedu, S. Vialle, and C. Timsit, "Comparison of OpenMP and classical multi-threading parallelization for regular and irregular algorithms," *Context*, vol. 2, p. 20, 2000.

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

[38] M. Sato, "OpenMP: parallel programming API for shared memory multiprocessors and on-chip multiprocessors," in *Proceedings of the 15th international symposium on System Synthesis*. ACM, 2002, pp. 109–111.

[39] V. Pankratius, A. Jannesari, and W. F. Tichy, "Parallelizing bzip2: A case study in multicore software engineering," *Software, IEEE*, vol. 26, no. 6, pp. 70–77, 2009.

[40] F. da Silva Pinagé, "Codificação de Voz usando Recorrência de Padrões Multiescalas," Ph.D. dissertation, Universidade Federal do Rio de Janeiro, 2011.

[41] N. M. Rodrigues, E. A. Silva, M. B. de Carvalho, S. M. Faria, and V. M. da Silva, "An efficient H.264-based video encoder using multiscale recurrent patterns," in *SPIE Optics+ Photonics*. International Society for Optics and Photonics, 2006, pp. 63 120W–63 120W.

[42] N. M. Rodrigues, E. A. da Silva, M. B. de Carvalho, S. M. de Faria, and V. M. M. da Silva, "Improving H.264/AVC inter compression with multiscale recurrent patterns," in *Image Processing, 2006 IEEE International Conference on*. IEEE, 2006, pp. 1353–1356.

[43] M. B. d. Carvalho, "Compression of multidimensional signals based on recurrent multiscale patterns," Ph.D. dissertation, 2001.

[44] N. C. Francisco, "Estudo da Utilização de Recorrência de Padrões Multiescala na Codificação de Documentos Compostos MEEC/UTAD," Ph.D. dissertation, Universidade de Trás-os-Montes e Alto Douro, 2007.

[45] A. Ortega and K. Ramchandran, "Rate-distortion methods for image and video compression," *Signal Processing Magazine, IEEE*, vol. 15, no. 6, pp. 23–50, 1998.

[46] T. D. Han and T. S. Abdelrahman, "Reducing branch divergence in GPU programs," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2011, p. 3.

[47] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE micro*, no. 2, pp. 39–55, 2008.

[48] B. Jang, D. Schaa, P. Mistry, and D. Kaeli, "Exploiting memory access patterns to improve memory performance in data-parallel architectures," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 1, pp. 105–118, 2011.

[49] S. Che, J. W. Sheaffer, and K. Skadron, "Dymaxion: optimizing memory access patterns for heterogeneous systems," in *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*. ACM, 2011, p. 13.

[50] S.-T. Leung and J. Zahorjan, *Optimizing data locality by array restructuring*. Department of Computer Science and Engineering, University of Washington, 1995.

[51] M. Harris *et al.*, "Optimizing parallel reduction in CUDA," *NVIDIA Developer Technology*, vol. 2, no. 4, 2007.

[52] L. Dagum and R. Enon, "OpenMP: an industry standard API for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[53] R. Chandra, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[54] O. A. R. Board, "OpenMP Compilers," http://openmp.org/wp/openmp-compilers/, 2015, accessed: 23-02-2016.

[55] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the chapel language," *International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[56] Wikimedia, "Illustration of the fork–join model of parallel programming," https://en.wikipedia.org/wiki/File:Fork_join.svg, 2007, accessed: 14-03-2016.

[57] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, M. Spiegel, and J. F. Prins, "OpenMP task scheduling strategies for multicore NUMA systems," *International Journal of High Performance Computing Applications*, vol. 26, no. 2, pp. 110–124, 2012.

[58] H.-W. Loidl and K. Hammond, "On the Granularity of Divide-and-Conquer Parallelism." in *Functional Programming*. Citeseer, 1995, p. 8.

[59] C. A. Hoare, "Quicksort," *The Computer Journal*, vol. 5, no. 1, pp. 10–16, 1962.

[60] E. Mohr, D. A. Kranz, and R. H. Halstead Jr, "Lazy task creation: A technique for increasing the granularity of parallel programs," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 2, no. 3, pp. 264–280, 1991.

[61] W. Magro, P. Petersen, and S. Shah, "Hyper-Threading Technology: Impact on Compute-Intensive Workloads." *Intel Technology Journal*, vol. 6, no. 1, 2002.

[62] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, "Hyper-Threading Technology Architecture and Microarchitecture." *Intel Technology Journal*, vol. 6, no. 1, 2002.

[63] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference.* ACM, 1967, pp. 483–485.

[64] R. B. Shenoy, "An Overview of OpenMP based Automatic Parallelization Tools."

[65] M. Ishihara, H. Honda, T. Yuba, and M. Sato, "Interactive parallelizing assistance tool for OpenMP: iPat/OMP," in *Proc. Fifth European Workshop on OpenMP (EWOMP '03)*, 2003, pp. 21–29.

[66] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl, "Continuous profiling: where have all the cycles gone?" *ACM Transactions on Computer Systems (TOCS)*, vol. 15, no. 4, pp. 357–390, 1997.

[67] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos, "ProfileMe: Hardware support for instruction-level profiling on out-of-order processors," in *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture.* IEEE Computer Society, 1997, pp. 292–302.

[68] J. Fenlason and R. Stallman, "GNU gprof," *GNU binutils.[Online]. Available: http://www. gnu. org/software/binutils*, 1988.

[69] R. K. Malladi, "Using Intel® VTune™ Performance Analyzer Events/Ratios & Optimizing Applications," *http:/software. intel. com*, 2009.

[70] T. Moseley, D. A. Connors, D. Grunwald, and R. Peri, "Identifying potential parallelism via loop-centric profiling," in *Proceedings of the 4th international conference on Computing frontiers.* ACM, 2007, pp. 143–152.

[71] S. Sah and V. G. Vaidya, "A Review of Parallelization Tools and Introduction to Easypar," *International Journal of Computer Applications*, vol. 56, no. 12, 2012.

[72] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," in *ACM Sigplan Notices*, vol. 17, no. 6. ACM, 1982, pp. 120–126.

[73] J. Levon, P. Elie *et al.*, "Oprofile," *A system-wide profiler for Linux systems. Homepage: http://oprofile. sourceforge. net*, 2006.

[74] J. Thiel, "An overview of software performance analysis tools and techniques: From gprof to dtrace," *Washington University in St. Louis, Tech. Rep*, 2006.

[75] F. Pillet, "KProf, Profiling made easy," *URL http://kprof. sourceforge. net/. Interface à gprof (outil de profiling) sous Kde simplifiant son utilisation*, 2002.

[76] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull, "Graphviz and dynagraph—static and dynamic graph drawing tools," in *Graph drawing software*. Springer, 2004, pp. 127–148.

[77] J. Fonseca, "Gprof2dot-jrfonseca-convert profiling output to a dot graph," https://github.com/jrfonseca/gprof2dot, 2013, accessed: 29-04-2015.

[78] D. van Heesch, "Doxygen: Source code documentation generator tool," *Avalibale online: http://www. doxygen. org (accessed on 24 December 2014)*, 2008.

[79] R. S. Laramee, "Bob's Concise Introduction to Doxygen," Technical report, The Visual and Interactive Computing Group, Computer Science Department, Swansea University, Wales, UK, 2007., Tech. Rep., 2011.

[80] V. M. Weaver, "Linux perf_event features and overhead," in *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*, 2013, p. 80.

[81] A. Duran, J. Corbalán, and E. Ayguadé, "Evaluation of OpenMP task scheduling strategies," in *OpenMP in a new era of parallelism*. Springer, 2008, pp. 100–110.

[82] Intel, "Intel 64 and IA-32 Architectures Developer's Manual," vol. Volume 3B: System Programming Guide, Part 2, 2015.

[83] K. O. W. Group *et al.*, "The OpenCL Specification Version 1.1. Khronos Group, 2011," https://www.khronos.org/registry/cl/specs/opencl-1.1.pdf, 2011, accessed: 01-11-2015.

[84] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 1-3, pp. 66–73, 2010.

[85] AMD, "Introduction to OpenCL Programming," pp. 89–90, 2010, archived May 16, 2011, at the Wayback Machine.

[86] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing with OpenCL: Revised OpenCL 1.2 Edition*. Newnes, 2012.

[87] J. Tompson and K. Schlachter, "An introduction to the opencl programming model," *Person Education*, 2012.

[88] V. Srinivasan, D. Brooks, M. Gschwind, P. Bose, V. Zyuban, P. N. Strenski, and P. G. Emma, "Optimizing Pipelines for Power and Performance," in *Microarchitecture, 2002.(MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on.* IEEE, 2002, pp. 333–344.

[89] G. Mitra, B. Johnston, A. P. Rendell, E. McCreath, and J. Zhou, "Use of SIMD vector operations to accelerate application code performance on low-powered ARM and Intel platforms," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International.* IEEE, 2013, pp. 1107–1116.

[90] C. Lomont, "Introduction to Intel Advanced Vector Extensions," *Intel White Paper*, 2011.

[91] Z. Majo and T. R. Gross, "Memory system performance in a NUMA multicore multiprocessor," in *Proceedings of the 4th Annual International Conference on Systems and Storage.* ACM, 2011, p. 12.

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

[93] G. J. Sullivan, J.-R. Ohm, W.-J. Han, and T. Wiegand, "Overview of the high efficiency video coding (HEVC) standard," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 22, no. 12, pp. 1649–1668, 2012.

[94] "HEVC / H.265 Explained," http://x265.org/hevc-h265/, accessed: 14-06-2016.

[95] V. Sze, M. Budagavi, and G. J. Sullivan, "High Efficiency Video Coding (HEVC)," in *Integrated Circuit and Systems, Algorithms and Architectures.* Springer, 2014, pp. 1–375.

[96] D. Grois, D. Marpe, A. Mulayoff, B. Itzhaky, and O. Hadar, "Performance Comparison of H.265/MPEG-HEVC, VP9, and H.264/MPEG-AVC Encoders," in *Picture Coding Symposium (PCS), 2013.* IEEE, 2013, pp. 394–397.

[97] H. Koumaras, M.-A. Kourtis, and D. Martakos, "Benchmarking the Encoding Efficiency of H.265/HEVC and H.264/AVC," in *Future Network & Mobile Summit (FutureNetw), 2012.* IEEE, 2012, pp. 1–7.

[98] C. Conti, P. Nunes, and L. D. Soares, "New HEVC prediction modes for 3D holoscopic video coding," in *Image Processing (ICIP), 2012 19th IEEE International Conference on.* IEEE, 2012, pp. 1325–1328.

[99] C. Conti, L. D. Soares, and P. Nunes, "HEVC-based 3D holoscopic video coding using self-similarity compensated prediction," *Signal Processing: Image Communication*, vol. 42, pp. 59–78, 2016.

[100] R. J. Monteiro, N. M. Rodrigues, and S. M. Faria, "Disparity compensation using geometric transforms," in *3DTV-Conference: The True Vision-Capture, Transmission and Display of 3D Video (3DTV-CON), 2014*. IEEE, 2014, pp. 1–4.

[101] G. Lippmann, "Epreuves reversibles donnant la sensation du relief," *j. phys*, vol. 7, no. 4, pp. 821–825, 1908.

[102] A. Aggoun, E. Tsekleves, M. R. Swash, D. Zarpalas, A. Dimou, P. Daras, P. Nunes, and L. D. Soares, "Immersive 3D holoscopic video system," *MultiMedia, IEEE*, vol. 20, no. 1, pp. 28–37, 2013.

[103] C. Conti, L. Ducla Soares, and P. Nunes, "Influence of self-similarity on 3D holoscopic video coding performance," in *Proceedings of the 18th Brazilian Symposium on Multimedia and the Web*. ACM, 2012, pp. 131–134.

[104] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund *et al.*, "Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 451–460, 2010.

[105] D. Padua, *Encyclopedia of Parallel Computing*, ser. Encyclopedia of Parallel Computing. Springer, 2011, no. vol. 4. [Online]. Available: https://books.google.pt/books?id=Hm6LaufVKFEC

[106] P. P. Wang and R. C. Shiau, "Machine recognition of printed Chinese characters via transformation algorithms," in *Decision and Control, 1972 and 11th Symposium on Adaptive Processes. Proceedings of the 1972 IEEE Conference on*. IEEE, 1972, pp. 532–536.

[107] J. S. Huang and M.-L. Chung, "Separating similar complex Chinese characters by Walsh transform," *Pattern Recognition*, vol. 20, no. 4, pp. 425–428, 1987.

[108] J. Andrade, G. Falcao, and V. Silva, "Optimized Fast Walsh–Hadamard Transform on GPUs for non-binary LDPC decoding," *Parallel Computing*, vol. 40, no. 9, pp. 449–453, 2014.

[109] K. Kasai, Y. Fujisaka, and M. Onsjö, "FFT-based parallel decoder of non-binary LDPC codes on GPU: KFO NBLDPC GPU," *URL: http://www.comm.ss.titech.ac.jp/˜kenta/KFO_NBLDPC_GPU.tar.gz*, 2009.

[110] H. Sarukhanyan, A. Anoyan, S. Agaian, K. Egiazarian, and J. Astola, "Fast Hadamard transforms," in *Proc of. Int. TICSP Workshop on Spectral Methods and Multirate Signal Processing*, 2001, pp. 16–18.

[111] Y. Be'ery and J. Snyders, "Optimal soft decision block decoders based on fast Hadamard transform," *IEEE transactions on information theory*, vol. 32, no. 3, pp. 355–364, 1986.

[112] J. W. Carl and R. V. Swartwood, "A hybrid Walsh transform computer," *IEEE Transactions on Computers*, vol. 22, no. 7, pp. 669–672, 1973.

[113] P. Zheng and J. Huang, "Walsh-Hadamard transform in the homomorphic encrypted domain and its application in image watermarking," in *International Workshop on Information Hiding*. Springer, 2012, pp. 240–254.

[114] "ISO/IEC 29199-2:2010: Information technology – JPEG XR image coding system – Part 2: Image coding specification."

[115] B. J. Fino and V. R. Algazi, "Unified Matrix Treatment of the Fast Walsh-Hadamard Transform," *IEEE Transactions on Computers*, vol. C-25, no. 11, pp. 1142–1146, Nov 1976.

[116] Wikimedia, "Fast Walsh–Hadamard transform of Boolean function 10100110," https://en.wikipedia.org/wiki/File:1010_0110_Walsh_spectrum_(fast_WHT).svg, 2011, accessed: 20-07-2016.

[117] E. O. Brigham and E. O. Brigham, *The fast Fourier transform.* Prentice-Hall Englewood Cliffs, NJ, 1974, vol. 7.

[118] P. Duhamel and M. Vetterli, "Fast Fourier transforms: a tutorial review and a state of the art," *Signal processing*, vol. 19, no. 4, pp. 259–299, 1990.

[119] R. L. Easton, "Discrete Fourier Transforms," *Fourier Methods in Imaging*, pp. 511–572.

[120] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing, 2nd edition, Upper Saddle River, NJ: Prentice Hall, 1989.* Pearson Higher Education.

[121] C. J. Weinstein, "Quantization effects in digital filters," DTIC Document, Tech. Rep., 1969.

[122] A. V. Oppenheim and C. J. Weinstein, "Effects of finite register length in digital filtering and the fast Fourier transform," *Proceedings of the IEEE*, vol. 60, no. 8, pp. 957–976, 1972.

[123] R. Tolimieri, C. Lu, and M. An, "Algorithms for Discrete Fourier Transform and Convolution."

[124] Nvidia, CUDA, "SDK code samples," http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/samples.html, 2016, accessed: 12-07-2016.