Optimization of multimedia applications on embedded multicore processors

Elias Michel Baaklini

Thèse de doctorat

Pour obtenir le grade de Docteur de l'Université de

VALENCIENNES ET DU HAINAUT-CAMBRESIS

et de l'ARAB OPEN UNIVERSITY, LIBAN

Discipline : **Informatique**

Présentée et soutenue par Elias Michel, Baaklini.

Le 12/02/2014, à Valenciennes, France.

Ecole doctorale :
Sciences Pour l'Ingénieur (SPI), Collège Doctoral Lille Nord de France

Equipe de recherche, laboratoire :
Laboratoire d'Automatique, de Mécanique et d'Informatique Industrielles et Humaines (LAMIH)

Optimisation des Applications Multimédia sur des Processeurs Multicoeurs Embarqués

JURY

Président du jury
- Artiba, Abdelhakim, Professeur, Université de Valenciennes, France.

Rapporteurs
- Goossens, Bernard, Professeur, Université de Perpignan, France.
- Diguet, Jean-Philippe, Docteur, Directeur de recherche CNRS, LAB-STICC, Lorient, France.

Examinateurs
- Yurdakul, Arda, Professeur, Bogazici University, Istanbul, Turquie.
- Artiba, Abdelhakim, Professeur, Université de Valenciennes, France.

Directeur de thèse
- Niar, Smail, Professeur, LAMIH, Université de Valenciennes, Valenciennes, France.

Co-encadrant
- Sbeity, Hassan, Assistant Professor, Arab Open University, Beyrouth, Liban.

Optimization of Multimedia Applications on Embedded Multicore Processors

Elias Michel Baaklini

University of Valenciennes

France

A thesis submitted for the degree of

Doctor of Philosophy

February 2014

Acknowledgments

The work presented in this thesis was carried out in the computer science departments at the University of Valenciennes in France and the Arab Open University in Lebanon. I would like to thank all the people who made this research successful.

In the first place, I would like to show my greatest gratitude to my thesis director Smail Niar, professor at University of Valenciennes, France, and to my supervisor Hassan Sbeity, assistant professor at Arab Open University in Beirut, Lebanon, for their continuous guidance, inspiration, and motivation.

Moreover, I would like to deeply thank my reviewers, Bernard Goossens, professor at University of Perpignan, and Jean-Philippe Diguet, research director at LAB-STICC in Lorient, for reviewing, correcting, and enhancing my thesis dissertation with their detailed comments and constructive feedback.

In addition, I would like to thank and to show my appreciation to my examiners, Arda Yurdakul, professor at Bogazici University in Istanbul, Turkey, and Abdelhakim Artiba, professor at University of Valenciennes, France.

Furthermore, I cannot forget all the support of my friends and colleagues who helped me during my long journey. Thank you all for your company and assistance.

Finally, I am very thankful for the valuable support of my family. Special gratitude goes to my mother, Fairouz, for her permanent love and encouragement.

I would like to dedicate this PhD thesis to my loving mother Fairouz and to my late father Michel.

Elias Baaklini

Résumé

L'utilisation de plusieurs cœurs pour l'exécution des applications mobiles sera l'approche dominante dans les systèmes embarqués pour les prochaines années. Cette approche permet en générale d'augmenter les performances du système sans augmenter la vitesse de l'horloge. Grace à cela, la consommation d'énergie reste modérée. Toutefois, la concurrence entre les tâches doit être exploitée afin d'améliorer les performances du système dans les différentes situations ou l'application peut s'exécuter.

Les applications multimédias comme la vidéoconférence ou la vidéo haute définition, ont de nombreuses nouvelles fonctionnalités qui nécessitent des calculs complexes par rapport aux normes précédentes de codage vidéo. Ces applications créent une charge de travail très importante sur les systèmes multiprocesseurs. L'exploitation du parallélisme pour les applications multimédia, comme le codec vidéo H.264/AVC, peut se faire à différents niveaux : au niveau de données ou bien au niveau tâches.

Dans le cadre de cette thèse de doctorat, nous proposons de nouvelles solutions pour une meilleure exploitation du parallélisme dans les applications multimédia sur des systèmes embarqués ayant une architecture parallèle symétrique (ou SMP pour Symmetric Multi-Processor). Des approches innovantes pour le décodeur H.264/AVC qui traitent des composantes de couleur et des blocs de l'image en parallèle sont proposées et expérimentées.

Mots Clés : Multimédia, Standard H.264/AVC, Compression Vidéo, Optimisation, Calcul Parallèle, Systèmes Embarqués, Processeurs Multicoeurs.

Abstract

Parallel computing is currently the dominating architecture in embedded systems. Concurrency improves the performance of the system rather without increasing the clock speed which affects the power consumption of the system. However, concurrency needs to be exploited in order to improve the system performance in different applications environments.

Multimedia applications (real-time conversational services such as video conferencing, video phone, etc.) have many new features that require complex computations compared to previous video coding standards. These applications have a challenging workload for future multiprocessors. Exploiting parallelism in multimedia applications can be done at data and functional levels or using different instruction sets and architectures.

In this research, we design new parallel algorithms and mapping methodologies in order to exploit the natural existence of parallelism in multimedia applications, specifically the H.264/AVC video decoder. We mainly target symmetric shared-memory multiprocessors (SMPs) for embedded devices such as ARM Cortex-A9 multicore chips. We evaluate our novel parallel algorithms of the H.264/AVC video decoder on different levels : memory load, energy consumption, and execution time.

Keywords : Multimedia, H.264/AVC Standard, Video Compression, Optimization, Parallel Computing, Embedded Systems, Multicore Processors.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background and Motivation

Nowadays, mobile devices supporting multimedia applications are pervasive in our modern world. Most hand-held devices are equipped with high resolution screens and fast multicore embedded processors. Dual and quad cores are found in recent smartphones and tablet devices like high end devices offered by Samsung and Apple [5, 53]. ARM Cortex-A9 processors can have up to 4 cores per chip [6]. Cortex-A15 processors can have up to 8 cores per chip (each chip can contain 2 clusters where each cluster can have up to 4 cores) [7]. On the other hand, applications do not benefit automatically from these powerful top-of-the-line processors. Even with new cutting-edge processors, video resolutions are increasing rapidly which require more processing time and consequently more energy consumption. Operating systems simply map independent applications, or multiple threads within an application, on different cores. Therefore, one application alone may not benefit from the additional resources available unless it is designed to execute in parallel. Thus, sequential applications need to be redesigned and recompiled in order to support parallelism. The process of parallelization faces many challenges like dependencies, synchronization, data coherency, etc.

Video players, digital cameras, televisions, and phones support complex video codecs with high resolutions. However, few multimedia applications benefit from the computational potentials that multicore processors offer in these emerging

powerful embedded devices. Video coding standards, like H.264/AVC [26] and HEVC [63], adopted complex algorithms in order to achieve better compression and to lower transmission bitrates. The additional complexity of these algorithms has negative impacts on execution time and energy consumption.

H.264/AVC [26] is currently the most widely used video compression standard for recording, compressing, and distributing high definition (HD) videos. The standard's first draft was released in 2003 and its latest version in 2012 [26]. Most HD video streaming websites like YouTube currently support H.264 as their default video codec [71]. H.264 is a high computational video compression standard that emerged as a result of the joint effort for Moving Picture Experts Group (MPEG) and the Video Coding Experts Group (VCEG). The H.264 standard offers better compression and higher quality compared to other standards like MPEG-2 [65]. This increase in compression results is the cost of high computational blocks like *Deblocking Filters* (DF) and complex *Entropy Decoding* techniques.

In our research, we choose the H.264/AVC video decoder [26] as a high computational multimedia application to be parallelized. We solve the problem of high complexity of the H.264 decoder using parallel execution on multicore embedded processors.

1.2 Problem Statement

H.264/AVC [26] is a high computational video coding standard. The codec achieves a good compression at the expense of a slow performance. Even with new cutting-edge processors, video resolutions are increasing rapidly which require more processing time and consequently more energy consumption. One of the best time and energy optimization strategies is to execute an application on parallel cores. Converting or redesigning an application in order to be executed in parallel present many challenges like dependencies, synchronization, data coherency, shared memory, etc. In our research, we shall use the H.264 video decoder as a complex multimedia application to be paralleled. We solve the problem of high complexity of the H.264 decoder using parallel execution on multicore embedded processors in order to decrease execution time and energy consumption.

1.3 Existing Solutions

Many parallel implementations of the H.264 decoder exist ranging from parallel decoding of macroblocks (fine-grain implementation) till parallel decoding of groups of pictures (coarse-grain implementation). A macroblock is a 16x16 square pixel component of an image in a video sequence. A macroblock can also be divided into sub-blocks of smaller size. Macroblock parallel decoding is highly scalable since many independent macroblocks can be processed in parallel. However, dependencies and huge overheads are created as a result of memory communication and execution synchronization between macroblocks. On the other hand, parallel decoding of groups of pictures require large memory especially for high definition video sequences. In addition, they have a lower scalability than parallel macroblock decoding because of the small number of groups of frames that can be decoded in parallel.

1.4 Contributions

Our approaches to decode H.264 videos in parallel range from single macroblock level until rows of macroblocks. Additional techniques are used to minimize the overhead of the sequential part like the entropy decoder.

At first, we separate the decoding process between color components for each data sample of every macroblock. Then we apply a pipeline in order to minimize the stall time caused by synchronization of parallel cores. The parallel implementation is experimented on dual and quad core embedded processor simulator. In addition to execution time and memory usage statistics, power consumption results are presented using an advanced power estimation tool.

For our second approach, we process rows of independent macroblocks in parallel using a innovative algorithm that minimizes synchronization overhead without adding additional steps to the decoder. The motion compensation stage is processed using a proposed row-based algorithm and the deblocking filter stage using the so-called wavefront algorithm. This level of parallel execution that is based on macroblock rows may be considered between the coarse-grain and the fine-grain parallel approaches offering a balance between large overheads and high

scalability of previous solutions. The proposed parallel algorithm is evaluated on a multicore simulator and on real-board platforms with multicore and graphics processors.

Finally, a detailed study is provided for the impact of parallel algorithms on cache misses in symmetric multiprocessors. Two parallel algorithms for the motion compensation of the H.264 decoder are experimented and analyzed. A prefetching algorithm is proposed in order to minimize the cache misses caused by sharing data between cores.

In the following section, the outline of the thesis with a brief description for every chapter is provided.

1.5 Outline

In chapter 2, we present parallel computing concepts in terms of parallelism, memory architecture, and applications. In chapter 3, an overview of the H.264 standard is presented. The standard's coding process with it features and tools are explained in details. In addition, existing parallel implementations and related work for the H.264 standard are also presented. Our first H.264 parallel implementations that is based on color components parallel decoding is explained and evaluated in chapter 4. Speedup in execution time and energy saving statistics are illustrated. In chapter 5, another parallel algorithm for the H.264 decoder is described and experimented. Groups of macroblocks are processed in parallel on different cores. The algorithm is evaluated on real-board multicore platforms and on graphics processors. Simulation results with high number of parallel cores are also presented and discussed. In chapter 6, a cache optimization technique is proposed that is based on prefetching data of parallel macroblocks. Finally, a conclusion summarizes our contribution in the last chapter.

Chapter 2

Parallel Computing

2.1 Introduction

Parallel computing is a form of computer processing when tasks are executed concurrently at the same time [2]. There are different levels of parallel computing: instruction-level parallelism, data-level parallelism, and thread-level parallelism. Parallelism was mainly used in high-performance computing servers and supercomputers. A decade ago, parallel computing emerged as a solution to frequency scaling due to the physical constraints [48]. As energy consumption by computers has become an important factor in computer systems, parallel computing became the dominant model in computer architecture, mainly in multicore processors. [8]

Several types of parallel computers exist like multicore and multiprocessor computers which have multiple processors in a single machine. Clusters and grids use multiple computers to work on the same task simultaneously. Specialized parallel computer architectures like GPUs are also used with traditional processors in order to accelerate specific tasks like graphics calculations.

Parallel programs are much more difficult to write than sequential programs [21]. Concurrency usually introduces several potential software bugs like race conditions. Communication and synchronization between parallel tasks are typically some of the biggest drawbacks which affect significantly the performance. Theoretically, the maximum possible speedup of a program as a result of parallel processing is known as Amdahl's law. [4]

From the mid-1980s until 2004, frequency scaling was the dominant reason for improvements in computer performance. Each generation of processors offered an increased frequency compared to previous versions while maintaining the remaining components almost the same. The runtime of a program is equal to the number of instructions multiplied by the average time per instruction. So maintaining everything else constant, increasing the clock frequency decreases the average time it takes to execute an instruction [21]. However, a higher frequency increases the amount of power used in a processor. Increasing processor power consumption caused the cancellation of Intel's Tejas and Jayhawk processors in May 2004. This date is generally cited as the end of frequency scaling as the dominant computer architecture paradigm. [18]

Moore's Law states that transistor density in a microprocessor doubles every 18 to 24 months [40]. Moore's law is still in effect despite repeated predictions of its end. With the end of frequency scaling, additional transistors are used to support parallel computing.

In section 2.2, we classify the types of processors and we describe three levels of parallelism: instruction, data, and thread. In section 2.3, the memory architecture for parallel systems is explained. In addition, shared memory communication and cache coherency are also discussed. At last, section 2.4 presents Amdahl's law, challenges for parallel computing, and most common programming languages for parallel development.

2.2 Types of Parallelism

Parallel computers can be roughly classified according to the level at which the hardware supports parallelism. This classification is broadly similar to the distance between main computing nodes. These are not mutually exclusive; for example, clusters of symmetric multiprocessors are relatively common.

2.2.1 Classification of Processors

Michael Flynn created in 1972 one of the earliest and most commonly used classification systems for parallel and sequential computers, the Flynn's taxonomy

Table 2.1: Flynn's classification scheme

	Single Instruction	**Multiple Instructions**
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

[21, 59]. Flynn classifies programs and computers by whether they were operating using a single set or multiple sets of instructions. He also specifies whether or not those instructions were using a single set or multiple sets of data. Table 2.1 lists Flynn's taxonomy in a tabular form.

Single-Instruction-Single-Data (SISD)

SISD classification is equivalent to a sequential program execution. The processor has a single memory and it executes one instruction at a time. Uniprocessors, like Pentium 4, falls in this category.

Single-Instruction-Multiple-Data (SIMD)

SIMD classification is similar to repeating the same operation over a large data set. This is usually found in signal and image processing applications. Another example is matrix multiplication where the same operation is performed on different data. SIMD microprocessors are currently available in most general-purpose processors. The x86 instruction set includes hundreds of SSE instructions that are aimed to improve the performance of multimedia applications.

Multiple-Instruction-Single-Data (MISD)

MISD is a rarely used classification. No machine had been classified in this category mainly because few applications would fit in this class.

Multiple-Instruction-Multiple-Data (MIMD)

MIMD programs are by far the most common type of parallel programs where a set of processors simultaneously execute different instruction sequences on different data sets. MIMD organization is a generalization of the other categories. It has been adopted by most general-purpose processors which allowed the exploitation of thread-level parallelism.

2.2.2 Instruction-Level Parallelism

Basically, a computer program is a stream of instructions that are executed by a processor. These instructions can be re-ordered and assigned into groups which are then executed in parallel without changing the outcome of the program. Since the mid-80s, all processors use pipelining to overlap the execution of instructions and improve performance. This potential overlap of instructions is known as *instruction-level parallelism* (ILP). [21]

Modern processors have multi-stage instruction pipelines. Each stage in the pipeline corresponds to a different action that is performed by the processor. Thus, a processor with an n-stage pipeline can have up to n different instructions at different stages of completion. A simple example of a pipelined processor is a RISC processor, with five stages: instruction fetch, decode, execute, memory access, and write back. [48]

There are two main approaches to exploit ILP. The first approach relies on hardware to help discover and exploit parallelism. The second approach relies on software technology to find parallelism statically at compile time. Processors that use dynamic, hardware-based approach, dominate the servers, desktops, and mobile markets. The recent Intel Core [24] and the ARM Cortex-A9 [6] processors families use this dynamic technology. Compiler-based approaches have not been successful except for a small range of scientific applications. [21]

In addition to instruction-level parallelism from pipelining, some processors can issue more than one instruction at a time. These are known as superscalar processors. Instructions can be grouped together only if there is no data dependency between them. Scoreboarding and Tomasulo algorithms are two of the most common techniques for implementing out-of-order execution and instruction-level parallelism. Also speculative execution and branch prediction are used to avoid stalling between instructions due to data dependencies. [27]

2.2.3 Data-Level Parallelism

For many years, the *single-instruction multiple data* (SIMD) architectures were mainly used for matrix-oriented scientific applications. Nowadays, multimedia applications, like image and sound compression, are being used to exploit

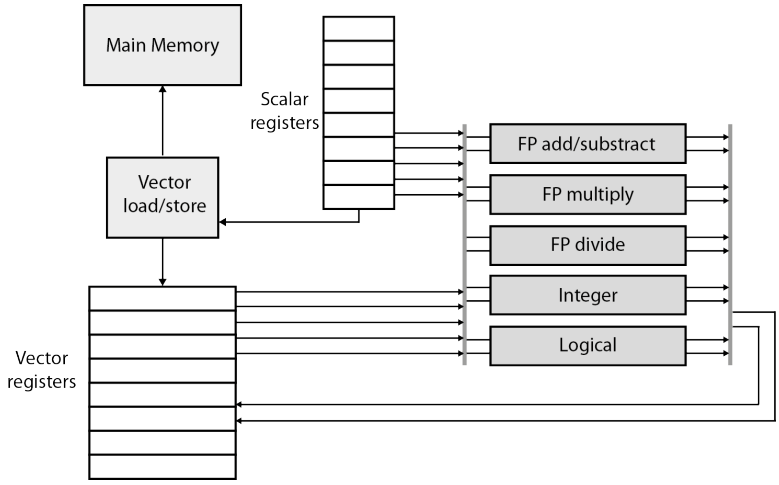

Figure 2.1: Basic structure of a vector architecture (VMIPS). The scalar architecture is similar to MIPS. The processor contains eight 64-element vector registers and all functional units are vector functional units.

data-level parallelism (DLP) on SIMD architectures. In an SIMD processor, a single instruction can be executed on many data operations. Thus, SIMD is potentially more energy-efficient than multiple instructions multiple data (MIMD) architecture, which needs to fetch and execute one instruction per data operation. These two reasons make SIMD attractive for Personal Mobile Devices. There are three main variations of SIMD: vector architectures, multimedia SIMD instruction set extensions, and graphics processing units (GPUs). [21]

Vector Architectures

Vector processors, which are available for more than 30 years ago, support pipelined execution of many data operations. They were very expensive until recently. There are considered a generalized architecture for SIMDs compared to other architectures. They require relatively more transistors and higher DRAM bandwidth with comparison to conventional computers [21]. Figure 2.1 shows the basic architecture of a vector processor with the instruction set architecture VMIPS which is a logical extension of MIPS.

Multimedia SIMD Instruction Set Extensions

SIMD instruction set extensions are currently available in most instruction set architectures that support multimedia applications. These additional

instructions are mainly used to perform simultaneous parallel data operations. For x86 architectures, the SIMD instruction extensions started with the MMX (Multimedia Extensions) in 1996. They were followed by several SSE (Streaming SIMD Extensions) versions, and lately, by AVX (Advanced Vector Extensions) instructions. Programmers need to use these SIMD instructions, especially for floating-point operations, in order to get the most of an x86 computer. [21]

General-Purpose Graphics Processing Units (GPGPU)

Traditional multicore computers today have a graphical processing unit (GPU) hardware. Together with the main processor (CPU) form a heterogeneous architecture that is suitable for multimedia extensive applications [21]. General-purpose computing on *graphics processing units* (GPGPU) is a fairly recent trend in computer engineering research. GPUs are co-processors that have been heavily optimized for computer graphics processing. Computer graphics processing is a field dominated by data parallel operations, particularly by linear algebra matrix operations. In the early days, GPGPU programs used the normal graphics APIs for executing programs. However, several new programming languages and platforms have been built to do general purpose computation on GPUs with both Nvidia and AMD releasing programming environments with CUDA and Stream SDK respectively [3, 43]. The technology consortium Khronos Group has released the OpenCL specification, which is a framework for writing programs that execute across platforms consisting of CPUs and GPUs. AMD, Apple, Intel, Nvidia and others support OpenCL. [30]

2.2.4 Thread-Level Parallelism

In this section, we focus on exploiting *thread-level parallelism* (TLP) through *multiple-instruction-multiple-data* (MIMD) architectures. Thread-level parallelism became relatively recently available in high-end servers, embedded and general-purpose applications. Computers who share the same memory address space and who have a single operating system are called *multiprocessors*. Typically, the number of processors in a multiprocessor system ranges in size from dual processor

to dozens of processors. Multiprocessors which have their shared memory address space on a single chip are called *multicores*. Multiprocessors may also consist of several multicore chips.

In order to benefit from an MIMD multiprocessor with n processors, we must have n threads or processes to run concurrently. These independent threads are typically identified by the programmer. The *granularity*, or *grain size*, of each thread, which is the amount of computation assigned to a thread, usually consists of hundreds of millions of instructions that will be executed in parallel. Threads can also exploit *data-level parallelism* (DLP). However, the overhead of data communication is relatively higher than *single-instruction-multiple-data* (SIMD) architectures. The grain size of parallel threads should be large enough in order to exploit parallelism efficiently.

Shared-memory multiprocessors are divided into two classes depending on the memory organization and the communication protocol. Each memory organization model is suitable for a system with a specific number of processors. For example, 32 processors are likely not to have, at least for now, all the processors and the shared memory on the same chip.

2.2.4.1 Symmetric Shared-Memory Multiprocessors (SMP)

For a small number of processors, typically 16 or fewer, processors may share a single centralized memory to which all the processors have equal access. In other terms, all the processors are symmetric in terms of memory access. This group of multiprocessors is called *symmetric shared-memory multiprocessors* (SMPs). All existing multicore chips are SMPs in a sense that they all have symmetric access, or a uniform latency, to a centralized shared memory. Hence, SMP architectures are also called *uniform memory access* (UMA) multiprocessors. Figure 2.2 shows a basic architecture of an SMP with 4 processors. These processors communicate with each other through shared variables in memory. Access to the shared variables must be coordinated via synchronization primitives, called locks, that prevent multiple access to the same data by different processors at the same time.

SMPs share memory and connect via a bus. However, bus contention prevents bus architectures from scaling. As a result, SMPs generally do not comprise more

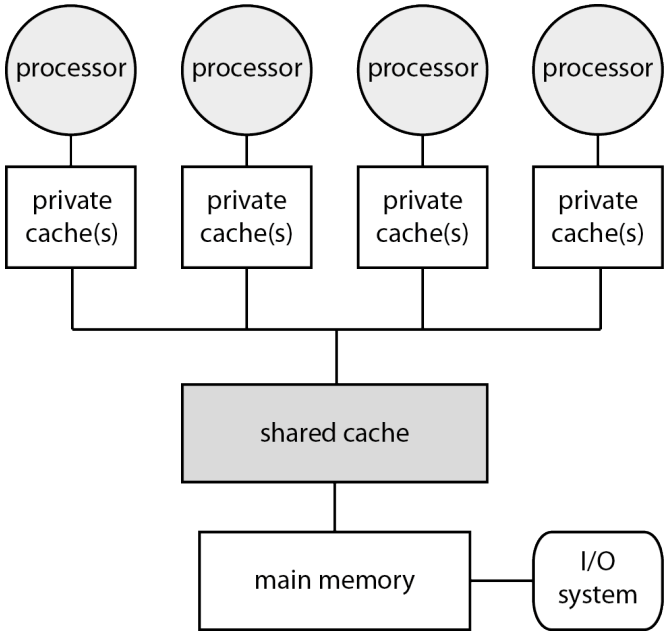

Figure 2.2: Basic structure of a *Symmetric MultiProcessor* (SMP) with a single address space on the same physical memory shared by all processors. This multiprocessor architecture is also called *Uniform Memory Access* (UMA).

than 32 processors. Because of the small size of the processors and the significant reduction in the requirements for bus bandwidth achieved by large caches, symmetric multiprocessors became cost-effective. [21]

Multicore Processors

The most common SMP chips are the multicore processors that are nowadays available in most desktop and portable computer devices. A multicore processor is a processor that includes multiple execution units, called cores, on the same chip. A multicore processor can issue multiple instructions per cycle from multiple instructions streams. Each core in a multicore processor can potentially be superscalar where each core can issue multiple instructions on every cycle from one instruction stream. Communication between the cores is usually maintained by a shared memory access. Multicore processors dominate the consumer market for personal computers with the Intel Core processor family [24] and hand-held devices with the ARM Cortex-A9 processors [6]. Figure 2.3 illustrates the simplified architecture of the ARM

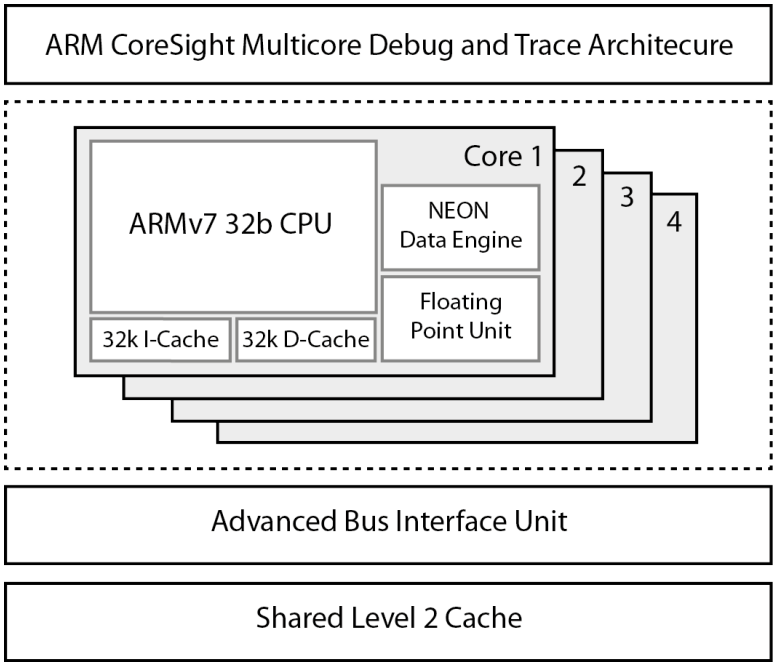

Figure 2.3: Multicore architecture of the ARM Cortex-A9 multiprocessor with an L2 cache shard by 4 cores each having an L1 private cache.

Cortex-A9 multicore processor with four 32-bit cores and a shared level-2 cache.

Reconfigurable Computing with Field-Programmable Gate Arrays

Reconfigurable computing is the use of a *field-programmable gate array* (FPGA) as a co-processor to a general-purpose computer. An FPGA is, in essence, a computer chip that can rewire itself for a given task. FPGAs can be programmed with hardware description languages such as VHDL or Verilog. [45]

Application-Specific Integrated Circuits (ASIC)

Several *application-specific integrated circuit* (ASIC) approaches have been devised for dealing with parallel applications. [1, 37, 56] By definition, an ASIC is specific to a given application for which it can be fully optimized for that application. As a result, for a given application, an ASIC tends to outperform a general-purpose computer. However, ASICs can be extremely expensive which has rendered ASICs unfeasible for most parallel computing

applications.

2.2.4.2 Distributed Shared-Memory Multiprocessors (DSM)

Multiprocessors with physically distributed memory are called *distributed shared-memory* (DSMs). Figure 2.4 illustrates the basic architecture of a DSM with 8 nodes where each node can be a multicore processor. A centralized memory will cause dramatically long access latency in order to support the bandwidth of a large number of processors. Thus, the need arises to have a distributed shared memory in order to connect many processors together. However, distributing the memory among the nodes of a DSM both increases the bandwidth and reduces the latency to local memory. Hence, a DSM multiprocessor is also called a *non-uniform memory access* (NUMA) for the reason that access latency depends on the location of the data being accessed. A major disadvantage for a DSM is the complex communication among processors. Additional effort in the application level should be performed by programmers in order to manipulate the distributed shared data.

The term *shared memory* in both SMP and DSM architectures refers to the fact that both architectures have an address space which is shared. Any processor can access a memory reference to any memory location. In contrast, clusters and warehouse-scale computers are individual computers connected by a network. In these architectures, the memory of one computer cannot be accessed by another without the assistance of message-passing protocols that are used for data communication among processors.

A *distributed shared-memory multiprocessor* (DSM) is a distributed computer system in which the processing elements are connected by a network. Distributed computers are highly scalable. The network communication may have different types of topologies like stars, rings, trees, and meshes. The following architectures are the most common types of distributed systems.

Clusters

A cluster is a group of loosely coupled computers that work together closely so that they can be regarded as a single computer. Clusters are composed of multiple standalone machines connected by a network. While machines in a

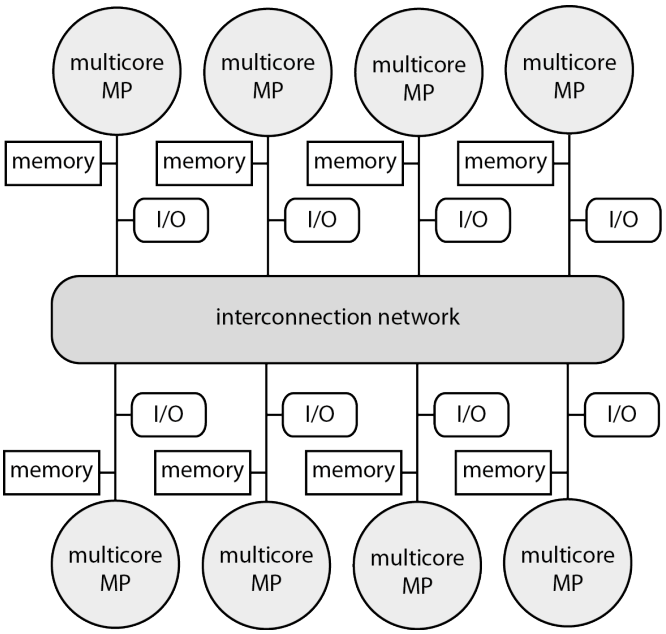

Figure 2.4: Basic structure of a *Distributed MultiProcessor* (DSM) with a single address space composed of several physical memories shared by all processors. This multiprocessor architecture is also called *Non-Uniform Memory Access* (NUMA).

cluster do not have to be symmetric, load balancing is more difficult if they are not. A typical cluster is implemented on multiple identical commercial off-the-shelf computers connected with a local area network.

Massively Parallel Processors

A massively parallel processor (MPP) is a single computer with many connected processors. MPPs have many of the same characteristics as clusters, but MPPs have specialized interconnect networks (whereas clusters use commodity hardware for networking). MPPs also tend to be larger than clusters, typically having more than 100 processors [21]. In a MPP, each CPU contains its own memory and copy of the operating system and application. Each subsystem communicates with other subsystems via a high-speed interconnect.

Grids

Distributed grid computing is the most distributed form of parallel computing. It makes use of computers communicating over the Internet to work on a given problem. Because of the low bandwidth and extremely high latency available on the Internet, grid computing typically deals only with embarrassingly parallel problems. Most grid computing applications use middleware, software that sits between the operating system and the application to manage network resources and standardize the software interface. The most common distributed computing middleware is the Berkeley Open Infrastructure for Network Computing (BOINC).

2.3 Memory Architecture for Parallel Systems

2.3.1 Main Memory

Main memory in a parallel computer is either shared or distributed. When all processing elements have a single address space, then the memory is shared among those elements. When each processing element has its own private memory, a local address space, then the memory is distributed. Accesses to local memory are typically faster than accesses to non-local memory. [48]

2.3.2 Processor Communication

Processor-processor and processor-memory communication can be implemented in hardware in several ways. Networks with different types of topologies can be used like stars, rings, trees, hypercubes, and meshes. Interconnect networks usually have message passing routines for communications. Lower level communications can be done using a shared multiplexed memory, a crossbar switch, and a shared bus. Large multiprocessors machine usually use a hierarchical architectures for communications between processors.

2.3.3 Memory Access

Computer architectures in which each element of main memory can be accessed with equal latency and bandwidth are known as *Uniform Memory Access* (UMA) systems. Typically, that can be achieved only by a shared memory system, in which the memory is not physically distributed. A system that does not have this property is known as *Non-Uniform Memory Access* (NUMA) architecture. Distributed memory systems have non-uniform memory access. Figure 2.4 on page 16 illustrates 8 processors classified into 2 groups. When a processor accesses the memory, its memory latency will depend on its directory which can be relative to its location.

2.3.4 Caches

Most computer systems use caches which are small, fast memories located close to the processor that store temporary copies of memory values. These memory blocks are close to the processor physically and logically. Parallel computer systems have difficulties with caches that may store the same value in more than one location. These inconsistencies may result in the possibility of incorrect program execution. These computers require a cache coherency system, which keeps track of cached values and ensures correct program execution. Designing large, high-performance cache coherence systems is a very difficult problem in computer architecture. For this reason, shared-memory computer architectures do not scale as well as distributed memory systems do. [48]

2.3.5 Cache Coherency

The main purpose of cache coherency is to keep data consistent across multiple cache memories. The two common write policies used in caches are:

Write back - write operations are made only in the cache. Main memory is updated only when the corresponding cache line is flushed from the caches.

Write through - all write operations are made in the cache and in the main memory, ensuring that data in the main memory is always valid.

The write-back approach can result in data inconsistency. If two caches contain the same data, and such data is updated in one cache, the other cache will unknowingly have an invalid value. Subsequently, invalid reads will produce invalid results. Inconsistency can occur even with the write-through policy, unless the other cache monitor the memory traffic or receive some direct notification of the update [21, 48]. The cache coherence protocols that have been proposed to solve these problems have generally been divided into software and hardware approaches. Some implementations adopt a hybrid strategy that involves both software and hardware approaches.

2.3.5.1 Software Solution

Software cache coherence schemes attempt to avoid the need for additional hardware circuitry and logic, by relying on the compiler and operating system to deal with the problem [48]. Software approaches are attractive because the overhead of detecting potential problems is paid during compile time instead of run time, and the design complexity is transferred from hardware to software. On the other hand, compile-time software approaches usually make very conservative decisions, thus frequently leading to inefficient cache utilization. One of the simplest approaches is to prevent any shared data variables from being cached. This is usually too conservative, because a shared data structure may be exclusively used during some periods and may be effectively read-only during other periods. Only during certain periods, when at least one process may update the variable and at least one other process may access the variable, is cache coherence an issue. More efficient approaches analyze the code to determine safe periods for shared variables access. The compiler then inserts specific instructions into the

generated code to enforce cache coherence during the critical periods. [48]

2.3.5.2 Hardware Solution

These solutions provide a runtime recognition of potential inconsistency conditions. Because this approach is on-the-fly, cache coherency is more efficient than the software approach. In addition, the hardware approach is transparent to the programmer and to the compiler, thus reducing the software development responsibilities. Hardware approaches are mainly divided into two categories: directory and snoopy protocols. [21, 48]

Directory protocols

Directory protocols collect and maintain information about where copies of shared data reside in one location, called *directory*. Typically, the directory is managed and manipulated by a centralized controller integrated in the main memory controller. When an individual cache controller makes a request, the directory controller checks and issues the necessary commands for data transfer between memory and caches or between caches themselves. [21, 48]

Snoopy Protocols

In snoopy protocols, the responsibility for maintaining cache coherence is distributed among all cache controllers. A cache must recognize when a memory block is shared with other caches. When an update action is performed on a shared block, it must be announced to all other caches by a broadcast mechanism. Each cache controller is able to snoop on the network to observe these notifications, and react accordingly [21, 48]. Snoopy protocols are ideally suited to a bus-based multiprocessor, since the shared bus provides a simple way of broadcasting and snooping. However, care must be taken so that the increased bus traffic required for broadcasting and snooping does not cancel out the gains from the use of local caches. [21]

2.4 Parallel Applications

Originally, computer applications were written for sequential execution where only one thread handles the entire processing of the program. The central processing unit (CPU) executes the instructions of an algorithm, one instruction at a time. On the other hand, parallel computing solves a problem by using multiple processing elements simultaneously. This is mainly accomplished by breaking the problem into independent parts that can be executed simultaneously. These problems are usually complex algorithms with heavy workload and time consuming. Another type where parallel computing is mainly used is the gaming and multimedia applications like image and video compression. The processing elements that support parallel computing are diverse and they may include several resources such as a single computer with multiple processors, several networked computers, specialized hardware, or any combination of the above. [47]

2.4.1 Amdahl's Law

Theoretically, the runtime of a parallel computer program should be divided by the number of processing elements that are executing the parts of the program concurrently. However, very few parallel algorithms achieve this optimal speedup. Most parallel algorithms can achieve near-linear speedup using small numbers of processing elements. When the number of parallel processors becomes high, the speedup remains constant as the maximum theoretical value is reached.

The potential speedup of an algorithm on a parallel computing platform is given by Amdahl's law [4]. The law states that the overall speedup of a program is limited by the part of the program which cannot be executed in parallel. A program typically consists of several parallel parts and several sequential parts. If n is the number of available processors and p is the proportion that can be executed in parallel, then the speedup, according to Amdahl's law, is calculated using the equation $s(n) = 1 / ((1 - n) + p/n)$. Figure 2.5 illustrates the speedup of a parallel algorithm when 25%, 50%, 75%, 90%, or 95% of the overall program is executed in parallel. A threshold is reached when a certain number of processors is used. For example, when 50% of a program is executed in parallel, the optimal speedup is 2. A maximum speedup of 10 is attained when 90% of the program

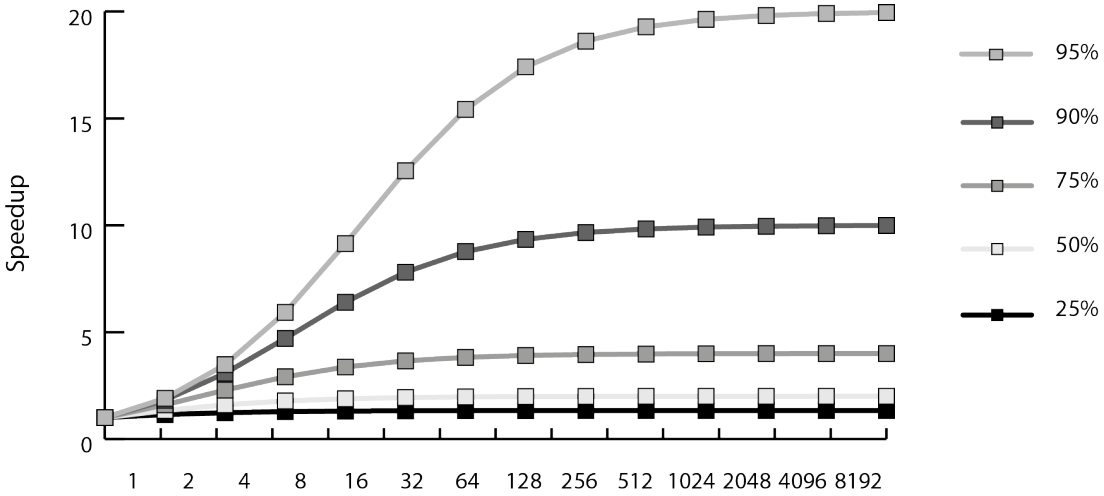

Figure 2.5: Graphical representation of Amdahl's law. The number of threads are displayed horizontally and its corresponding speedup vertically. The values depend on the amount in percentage of the sequential program that can be executed in parallel.

is executed in parallel no matter how many processors are added. In addition to the limitation caused by the serial part in a program, the speedup of parallel algorithms is also affected by several factors like dependencies, scheduling, load balancing, synchronization, and communication overhead.

2.4.2 Challenges of Parallel Processing

A parallel application may have independent threads without communication required to complete the task or it may have dependent threads where communication is essential between the threads to complete the require execution. The speedup of a parallel program, as explained by Amdahl's law [4], mainly faces two important challenges which their difficulty depends on the application and the multiprocessor architecture.

The first challenge is the amount of parallelism available in a program and the second issue deals with the high cost of communication. For example, in order to achieve a speedup of 80 using 100 processors, the sequential part should not be more than 25% of the overall application. Parallel applications with such high parallelism are currently rare, and thus, there are hard to find in typical

algorithms. [21]

The second challenge deal with the large latency of remote access in a parallel processor. Communication of data between different cores in existing shared memory multiprocessor may cost from 35 to 50 clock cycles. On the other hand, communication of data between separate chips in large-scale multiprocessors may cost from 100 to 500 clock cycles. The high memory latency may have a dramatic effect on the overall execution of parallel applications which can be slower than sequential execution of the application.

The two problems described above, insufficient parallelism and long-latency remote communication, deeply affect the usage of multiprocessors. Parallel algorithms should be designed in a way to maximize the size of parallel tasks and to reduce as much as possible the amount of communications. Some techniques include caching shared data, prefetching, reducing remote access frequency, load balancing, etc.

In our research, we explore the possibilities of parallelism of the H.264/AVC [26] decoder as a high computational application. As stated above, we intend to maximize the size of parallel tasks and to decrease the amount of data communications. In addition, we implement parallel algorithms with high scalability with a good load balance in order to improve the overall performance of the video decoder.

In this section, we describe the main characteristics of parallel programs. These features which affect the overall performance of parallel programs provide the efficiency of parallelism applied on parallel applications.

2.4.2.1 Granularity

Applications are often classified according to how often their parallel subtasks need to synchronize or communicate with each other. *Fine-grain* parallelism is identified when parallel threads of a program communicate heavily between each other, many times per second. On the opposite, *coarse-grain* parallelism occurs when parallel threads communicate few times per second. If parallel threads rarely or never have to communicate with each other, these threads are embarrassingly parallel, which is considered the easiest to implement. [48]

2.4.2.2 Synchronization

A computer program is usually called a *process*. A process can create multiple lightweight sub-processes that consider the main process as their parent. These lightweight processes are usually called *threads*. All communications and synchronization between parallel threads are considered an overhead to the overall execution compared to the original serial version of the program. Eventually, an excessive amount of locks for mutual exclusion sections and synchronization barriers will increase the runtime of the program. This increase in execution time is known as parallel slowdown or overhead. [58]

2.4.2.3 Dependencies

Whenever a parallel algorithm is implemented, dependencies among its data should be respected. Some calculations are needed to be completed before subsequent calculations can be performed. This dependency will limit the portion of code that can be processed in parallel. Thus, the parallel segments of code of a program should not have any data dependencies between them in order to obtain correct output as the original sequential program. In addition, the overall execution of the parallel code is limited to the biggest parallel chunk of code. That is some parts may finish before other parts and the program execution is completed only when all the parts of code have been processed. [48]

2.4.2.4 Scalability

In order to achieve a good speedup, parallel programs should scale well with the increase in the number of processors. The speedup should be close to the theoretical speedup of Amdahl's Law [4]. A parallel program is said to have a strong scaling when the size of the program remains almost the same compared to the original serial program. On the other hand, a weak scaling is achieved when the size of the parallel program increases proportionally with the number of processors. [48]

2.4.2.5 Load Balance

In a parallel algorithm, the total execution time of the whole program is limited by the biggest part of code among the parallel parts. Some processors that are executing in parallel parts of a program may finish before or after other processors [48]. If the offset is large between the execution times of parallel codes on different processors, then the speedup will be affected. The calculation of the speedup using Amdahl's law will not apply if load balancing is not respected. So a parallel program with a good load balance has all his parts of codes, which will be executed in parallel on different processors, almost with equal size.

2.4.3 Programming Languages

Concurrent programming languages, libraries, APIs, and parallel programming models have been created for programming parallel computers. These can be generally divided into classes based on the memory architecture: shared memory and distributed memory. Shared memory programming languages communicate by manipulating shared memory variables. POSIX Threads and OpenMP are two of most widely used shared memory APIs [58]. Distributed memory uses message passing. *Message Passing Interface* (MPI) is the most widely used message-passing system API citegropp. CUDA and OpenCL are used to write C-like code for GPGPU kernels [30, 44].

Message Programming Interface (MPI)

MPI is a low-level API which provides a standard communication mechanism and which is implemented on top of high-performance network drivers. The MPI is based on a process fork model. It runs on both distributed and shared memory-systems. MPI scales well to very large numbers of processors. However, the application development in MPI is often rather difficult since each node has to be separately programmed. [52]

Portable Operating System Interface for Unix (POSIX)

In the POSIX standard, parallel programming is exploited by using processes and threads. Processes are defined as independent execution units that contain their own state information, have individual address spaces,

and only interact with each other via interprocess mechanisms managed by the operating system. On the other hand, threads within a process share the same state and memory space, which leads to a lightweight processing flow with a low latency context switching. Communication between threads is usually performed using shared variables. [13]

OpenMP

OpenMP is a standard defining a set of compiler directives for C/C++ and Fortran languages based on the thread-fork model. These directives allow an easy and explicit way to define the code that can be executed in parallel. When OpenMP is used, parallel regions are simply defined using the *#pragma* keyword, reducing the parallelization complexity. Despite of its advantages, the scalability is limited by the number of cores of the certain platform. [13]

CUDA and OpenCL

Recently, *General Purpose Graphic Processor Units* (GPGPU) have emerged as a powerful and an alternative solution in computer graphics manipulation. Their highly parallel structure makes them very suitable to run complex algorithms. In the past, GPUs used to be programmed using programmable shaders found in graphics API (OpenGL, DirectX), which require the adaptation of general purpose code to graphic API. In 2007, NVIDIA released CUDA (Computed Unified Device Architecture), allowing a code written in C language to be executed on a GPU. Meanwhile, the generic framework OpenCL (Open Computing Language) has been also used for GPUs programming. A code written in OpenCL can be executed across heterogeneous platforms, mainly formed by CPUs and GPUs [60]. Nevertheless, despite their efficiency in parallel computation, the usage of CUDA or OpenCL often require a high bus bandwidth between the CPU and the GPU, leading to bus bottleneck. [30]

2.5 Conclusion

In this chapter, we described parallel computing at the hardware and the software levels. At the beginning, we list Flynn's classification scheme of processors. Then, brief explanations of the instruction-level and data-level parallelism are presented. Afterwards, thread-level parallelism with its different hardware parallel devices is explained in details. Moreover, different memory architectures for parallel processors are explained at the main memory and the cache levels. Cache coherency constraints and solutions are also described. In addition, parallel applications characteristics and challenges are also listed. Amdahl's law is also described briefly. Finally, the most common programming languages for parallel software implementations are presented.

Chapter 3

H.264/AVC Standard Overview

3.1 Introduction

The Moving Picture Experts Group (MPEG) and the Video Coding Experts Group (VCEG) developed jointly in 2003 the *Advanced Video Coding* (AVC) standard published as ITU-T Recommendation H.264 and as part 10 of MPEG-4 [26]. Since the first commercial implementations, several multimedia device manufacturers adopted the new video codec. Ten years after the first release of the final draft of the standard, H.264 is currently the mostly used video compression standard in multimedia devices according to many articles and magazines like PCWorld.com [23]. Cameras, smartphones, PDAs, CCTV recorders, blu-ray disc players and many other devices use H.264 for encoding and decoding videos. H.264 achieves better compression and higher quality at the expense of more complex algorithms. Thus, more computation resources are exploited and more energy is consumed when increasing compression ratio of video files. This chapter provides an overview of some of the main features of the standard. The following chapters will use this chapter content as a reference to the concepts and features of the H.264 standard. The proposed parallel algorithms and optimization are based on the H.264 decoding process which is explained in this chapter.

A review on video coding is presented in section 3.2. The development history of the standard is briefly discussed in section 3.3. Next, a high level overview of H.264/AVC is provided in section 3.4. Section 3.5 discusses the functional stages

of the encoder and the decoder. Section 3.6 focuses on some of the specific coding tools available in the video coding layer. Section 3.8 summarizes the H.264/AVC profiles and concludes the chapter.

3.2 Video Coding Review

This section provides essential background information on video coding. A brief description is presented for digital video sampling, color spaces and common picture formats. Then, block based video coding is explained with the different stages for encoding a video sequence. Common video compression standards are also listed with short descriptions.

3.2.1 Digital Video

Digital video is a stream of fixed size images captured at regular time intervals. The images are represented as digitized samples containing visual information (color and light) at each spatial and temporal location.

3.2.1.1 Sampling and Resolution

Figure 3.1 shows the sampling process of digital video relative to time and space. The resolution of the image is determined by the number of horizontal and vertical samples, or pixels. The frequency at which each image is captured (temporal sampling) affects the motion smoothness of the video.

Typical temporal sampling frequencies (frame rates) are 25 Hz and 30 Hz for low resolutions. For high definition resolutions, typical frequencies are 50 Hz and 60 Hz. The frame rate determines the motion smoothness of the video, where motion appears smoother at higher frame rates.

In digital video processing, different spatial resolutions are used depending on the target application. For example, for Blu-ray movies and gaming consoles, the typical resolutions are 1280 x 720 pixels (HD 720) and 1920 x 1080 pixels (HD 1080). On the other hand, in video conferencing and web contents applications, video sequences typically have low resolutions like CIF (Common Intermediate Format) which is composed of 352 x 288 pixels, and VGA (Video Graphics Array)

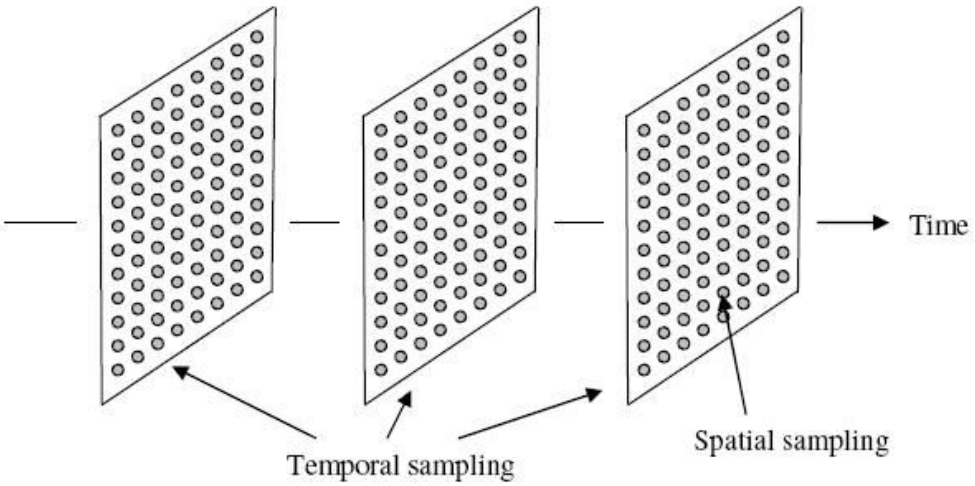

Figure 3.1: Digital Video Sampling.

Table 3.1: Common video formats and resolutions

Format name	Pixel resolution (Horizontal x Vertical)
CIF	352 x 288
VGA	640 x 480
SVGA	800 x 600
XVGA	1024 x 768
HD 720	1280 x 720
HD 1080	1920 x 1080

which is composed of 640 x 480 pixels. Some of the most widely used formats are displayed in Table 3.1.

3.2.1.2 Frames and Fields

A video signal can be sampled in either frames (progressive) or fields (interlaced). In progressive video, a complete frame is sampled at each time instant. In interlaced video, only half of the frame, either odd or even rows of samples, is captured at a particular time instant which are called fields. The field which has the odd rows of samples including the first row is called the top field and the field which has the even rows of samples is called the bottom field. Figure 3.2 illustrates the concept of progressive and interlaced video sampling of frames, in other words, frames and fields.

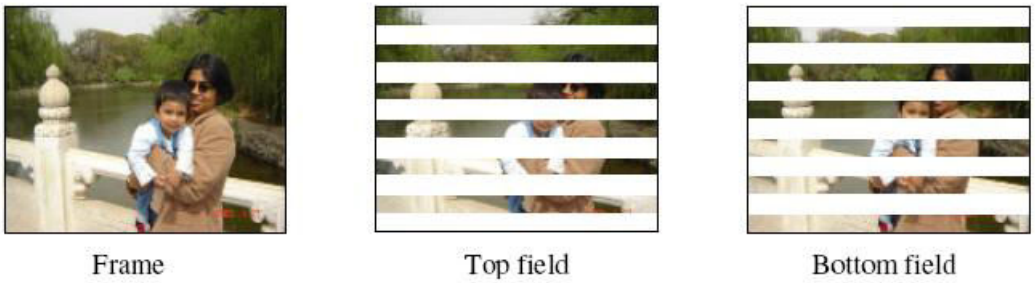

Figure 3.2: Progressive and Interlaced Video.

3.2.1.3 Color Spaces

Visual information at each sample point may be represented by the values of three basic color components Red (R), Green (G) and Blue (B). This color representation is called the RGB color space. Each value is stored in an n-bit number which is also called color depth. For example, an 8-bit color depth can store 256 levels to represent each color component.

The YCrCb color space is widely used to represent digital video. The *luminance* component Y (also called *luma*) is extracted using a weighted average of the three color components R, G and B. The components Cr and Cb are called the *chrominance* (or *chroma*) components are the color differences with Y. Cr is the red chrominance component (Cr = R - Y) and Cb is the blue chrominance component (Cb = B - Y). The computation of YCrCb color space from RGB color space can be found in Richardson's book on Video Codec Design [50]. The human visual system has less sensitivity to color information than luminance (light intensity) information [70]. Therefore, with the separation of luminance information from the color information, it is possible to represent color information with a lower resolution than the luminance information. Figure 3.3 shows the three widely used formats for representing chroma and luma samples: 4:4:4, 4:2:2, and 4:2:0 formats.

In 4:4:4 format, each pixel position has both luma and chroma samples. In 4:2:2 format, chroma components are sub-sampled (every other pixel) in horizontal direction. In 4:2:0 format, chroma samples are sub-sampled in both vertical and horizontal directions. This is the most popular format used in entertainment

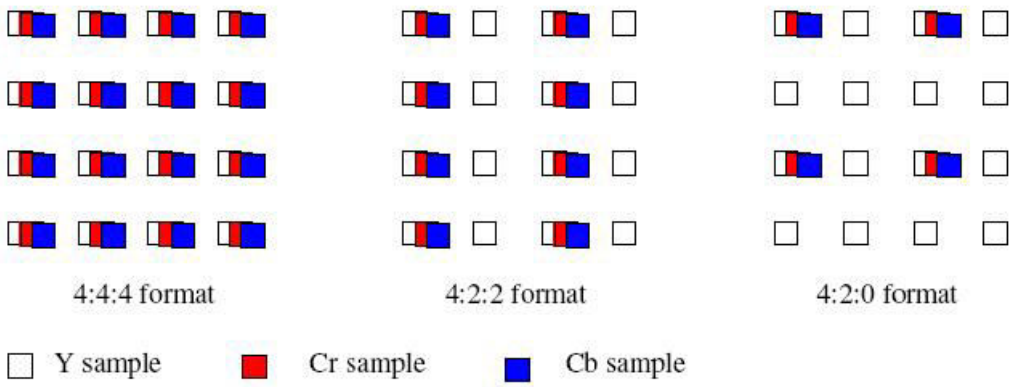

Figure 3.3: Sub-sampling patterns for chrominance components.

quality applications such as DVD video because the human eye does not easily recognize missing color information. All video sequences that are used in our research have the 4:2:0 color sampling format.

3.2.2 Block Based Video Coding

In block based video coding, the basic unit of coding is a block containing n x n array of luma samples and their corresponding chroma samples. The image is divided into a number of fixed size blocks. These blocks are processed in raster scan order, from left to right of each row and top to bottom row by row. Figure 3.4 shows a block diagram of a typical block based video encoder. The encoder has two data flow paths. The forward path represents the encoding process for coding of blocks and the reverse path (grey lines) shows the decoding reconstruction path within the encoder. Major elements of block based encoding are inter and intra prediction processes, transform, quantization and entropy coding.

3.2.2.1 Intra Prediction

Block based video encoders use prediction as a tool for removing redundant information. A prediction signal is obtained from previously coded samples for the coding unit and it is subtracted from the original coding unit to create a residual signal that has much less data than the original coding unit. It is the

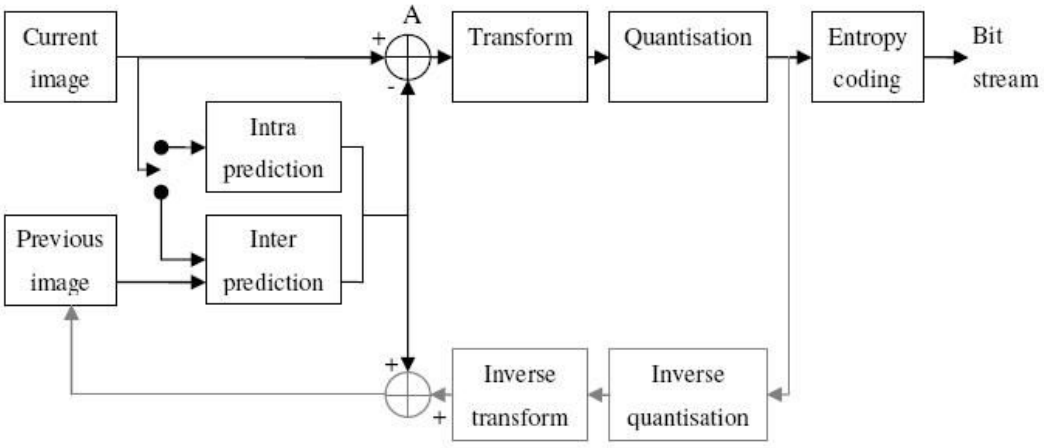

Figure 3.4: Block based encoder diagram.

residual signal that is encoded and transmitted to the decoder (node A in figure 3.4). The decoder obtains the same prediction signal using previously decoded samples, decodes the residual signal and adds them together to reconstruct the coding unit.

In intra prediction each coding unit is predicted using the surrounding pixels which have been already coded in the same image. Intra coding is always used in the first image of a sequence. Intra coding is also very useful in coding uniform regions where surrounding pixels of the block has similar value as the pixels inside the block. Intra prediction is used in relatively new video coding standards such as H.263 [19] and H.264/AVC [26].

3.2.2.2 Inter Prediction

In general, consecutive video images are very similar to each other. Differences in images mostly arise due to the movement of the objects in the video scene. Inter prediction is used to remove this temporal redundancy of video images. The prediction signal of a coding unit is obtained from a previously encoded and reconstructed image. The aim is to find a good match for the current block from the previously coded image. This can be done by following the motion of the object over time between the two images. Usually it is very difficult to find an exact match by precisely following the motion. However, a reasonably accurate

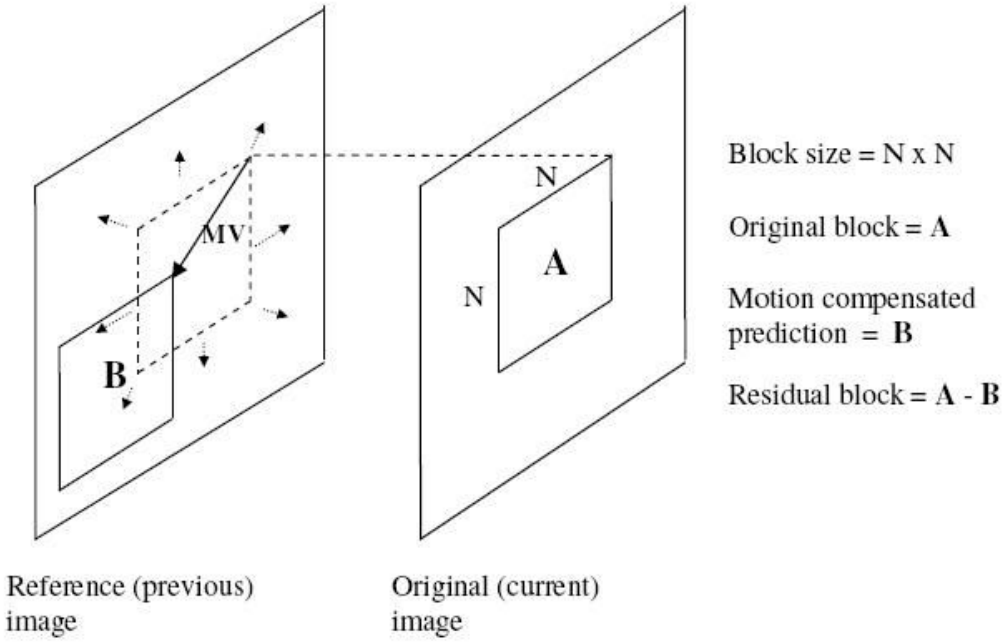

Figure 3.5: Motion estimation and compensation of an n x n block.

match can be found by searching for a similar block within a restricted region of the image. This process is illustrated in figure 3.5.

Common terms related to inter prediction process can now be introduced as follows:

Reference Image is the previously encoded and reconstructed image that is used for the prediction of blocks in the current image.

Motion Estimation is the process of searching and finding the closest matching block (B) from the reference image to the current block (A).

Motion Compensation is selecting the best matching block as the prediction and obtaining the residual by subtracting the prediction from the original block.

Motion Vector (MV) is the vector representing the displacement (horizontal and vertical) of the matching block from the position of the original block.

For inter predicted coding units, the residual signal is encoded and transmitted to the decoder along with the motion vector values. The decoder uses the motion

vector values to find the correct prediction block and the decoded residual is added to reconstruct the coding unit.

3.2.2.3 I, P and B Frames

In I-Frames, all the coding units are predicted using intra prediction only. These are used for the first frames of a sequence and as random access frames for reversing and fast forwarding without the need for decoding all the pictures. P-Frames are inter predicted pictures with the reference as the nearest previously coded picture. They cannot be used for random access, because of the dependency on previously coded pictures. They are also used as reference pictures. B-pictures are bidirectional predicted frames which require two reference frames for inter prediction, one from past and one from future in display order. They typically have high compression efficiency; however, they are not used for reference and cannot be used for random access.

3.2.2.4 Transform Coding

The residual block obtained after prediction stages is transformed from spatial domain into transform domain using a two dimensional block transform process resulting in a block of transform coefficients. These transform coefficients represent the residual image block. The transform needs to be reversible (inverse transform) in order to obtain the image residuals from the transform coefficients. The transform in itself does not achieve any compression. However, it serves for energy compaction where the transform concentrates most of the energy within a small number of large coefficients. The transform also de-correlates the data as its coefficients have minimal inter-dependency between each other. The most widely used block based transform in image and video compression is the Discrete Cosine Transform (DCT) [19].

3.2.2.5 Quantization

Quantization is the process of converting a continuous range of values to a finite range of discrete levels. For example, in digital video an 8-bit color sample is obtained by approximating the signal level of the color component into one

of the finite discrete levels, which is 256 levels in this case. Some of the color information is lost and cannot be recovered due to approximation and therefore more levels are needed to retain more information. In video compression, lossy compression is achieved by quantization. The quantization process consists of two stages. Forward quantization is carried out during encoding and rescaling is carried out during decoding. The two stages are also referred to as quantization and de-quantization. In forward quantization, the original transform coefficient value is typically divided by the quantization stage and rounded to the nearest integer. Information is lost during the rounding process. These integer values are transmitted to the decoder along with the quantization process used. Rescaling is carried out at the decoder, where the received integer is multiplied by the quantization integers in order to obtain the actual quantized transform coefficient. Lower bit rates can be achieved at higher quantization levels at the expense of a large approximation error and therefore higher image distortion.

3.2.2.6 Entropy Coding

The encoder needs to transmit data such as residual quantized transform coefficients, quantization values, motion vectors and other overhead information such as coding parameters to the decoder. Entropy coding is carried out to reduce the statistical redundancy of the transmitted data. This is a lossless compression technique where data with high probability of occurrence is coded with a smaller number of bits and data with lower probability of occurrence is coded with a larger number of bits. Commonly used entropy coding methods are Huffman coding and Arithmetic coding [26].

3.2.2.7 Decoder

The decoding process is identical to the reverse path of the encoder in figure 3.4 on page 33. The bit stream received from the encoder is first entropy decoded and then, inverse quantized and inverse transformed to create the residual. This residual is added to the prediction signal to construct the image. Due to lossy coding (quantization) the reconstructed image is not identical to the original image; however, the reconstructed images of the encoder and decoder are identical

to each other because the decoding process at the decoder and the processing carried out by the reverse path of the encoder are identical.

3.2.3 Video Coding Standards

Standardization of video coding technology has played a major role in the advancement of digital video communication technologies over recent years [54]. Standardization enables interoperability between different manufacturers and is a major requirement for the communications industry. The two international standardization bodies are namely, Video Coding Experts Group (VCEG) of International Telecommunications Union – Telecommunication Standardization Sector (ITU-T) and Motion Picture Experts Group (MPEG) of International Organization for Standardization – International Electrotechnical Commission (ISO/IEC).

The standards released by the ITU-T have been named H.26x series and ISO/IEC has released the MPEG series of standards. The MPEG standards have been mainly aimed at media storage and distribution while the H.26x standards have been aimed at real-time video communication applications [50, 54]. Some of the popular standards are named below:

MPEG-1

The draft MPEG-1 standard was released in 1993. Although this is a generic video coding standard, it was primarily designed for storage on digital media such as CD-ROM supporting bit rates up to 1.5 Mbit/s. The standard employs a block based coding algorithm similar to block based video coding described in the previous section. The standard supports flexible picture types: I-frames, P-frames and B-frames in order to provide good compression efficiency and added functionality such as fast forwarding.

MPEG-2

The MPEG-2 standard, released in 1995, was aimed at broad variety of applications such as media storage, satellite terrestrial TV broadcasting. It builds on MPEG-1 algorithm including new tools for better quality and functionality such as interlaced video and scalable video coding for applications such as digital TV and HDTV. This is the first standard to introduce the concept of profiles and levels as means of implementing compliant de-

coders that support only a subset of syntax with restriction on capability such as maximum supported bit rate.

MPEG-4 Visual

The MPEG-4 Part 2: Visual, released in 1998, supports a wide variety of applications including internet video streaming and digital TV broadcasting as well as applications with combined real world video scenes and computer generated graphics. The standard can support lower bit rates than MPEG-1 and MPEG-2. MPEG-4 Visual supports object-based video coding where a video scene is divided into different video objects that can be coded independently of each other.

H.261

This standard, approved in 1993, was widely used for videophone and video conferencing applications over the Internet. The H.261 standard uses hybrid coding algorithm, similar to MPEG-1, for efficient coding at lower bit rates using relatively a computationally simple algorithm. The H.261 standard only supports QCIF and CIF resolution non-interlaced video.

H.263

This standard was originally aimed at low bit rate video communications. The core algorithm is based on the H.261 standard. However, it supports a broad range of video formats and advanced coding tools such as half pixel precision motion compensation and a variety of coding tools such as unrestricted motion vectors, where the motion vector points to a region outside the picture boundary and advanced prediction, where the macroblock (the basic unit of coding of 16 X 16 pixels) is divided into four blocks. Each block is motion compensated using individual motion vectors, resulting in higher degree of compression efficiency and flexibility. The baseline profile of H.263 and the simple profile of MPEG-4 are functionally identical. [19]

H.264 / MPEG-4 Part 10: Advanced Video Coding

The relatively recent video coding standard commonly known as H.264/AVC was jointly developed by the ITU-T VCEG and the ISO/IEC MPEG. The H.264/AVC is capable of achieving significantly improved compression efficiency and flexibility compared with all previous video coding standards.

The increase in performance is due to the variety of coding tools and options available in the standard which, on the other hand, increases the computational complexity significantly.

H.265 / HEVC: High Efficiency Video Coding

High Efficiency Video Coding (HEVC) is currently the newest video coding standard of the ITU-T Video Coding Experts Group and the ISO/IEC Moving Picture Experts Group. The main goal of the HEVC standardization effort is to significantly improve compression performance relative to existing standards in the range of 50% bit-rate reduction for equal perceptual video quality. HEVC has been designed to address essentially all existing applications of H.264/AVC and to particularly focus on two key issues: increased video resolution and increased use of parallel processing architectures. [63]

This research is aimed at optimizing the performance of the H.264/AVC decoder. Therefore, a good understanding of the H.264/AVC standard is required. The following sections provide an overview of the features and the coding tools available in the H.264/AVC standard. [26]

3.3 Standard Development

In 1998 a call for proposals was issued by ITU-T Video Coding Experts Group (VCEG) for a new video coding standard with the objective of doubling the compression efficiency compared to any video coding standard available at the time. The new proposal was referred to as H.26L. As a result of similar interest by ISO/IEC, the Joint Video Team (JVT), consisting of ITU-T VCEG and ISO/IEC Moving Picture Experts group (MPEG), was formed in 2001 to make the development of the new standard a combined effort. The standard was finalized and the draft was approved in May 2003 [26].

The H.264/AVC standard was originally developed for web quality video where sampling format is limited to 4:2:0 with 8 bit sample accuracy. An amendment was added to the standard in July 2004 called the Fidelity Range Extensions (FRExt) [26, 36, 62] which introduced the so-called 'High Profiles' in order to ad-

dress professional applications and to enhance the compression performance. The high profiles can support up to 4:4:4 sampling format and 12 bit sample accuracy.

The H.264/AVC standard was designed for high compression efficiency, error resilience and flexibility so that it could support a wide variety of applications and different transport environments such as wired and wireless networks. The H.264/AVC standard is intended to support a wide range of applications such as:

- Video conferencing and video telephony services over networks such as LAN, DSL, wireless and mobile networks
- Video on demand and multimedia streaming services
- Digital broadcasting services
- Video storage on media
- Multimedia messaging services

Similarly to previous standards, H.264 only specifies the syntax structure of the bit stream and the decoding process of the syntax. This ensures high flexibility in encoder implementation as long as the generated bit stream conforms to the syntax, while guaranteeing interoperability and correct decoding of content. However, the decoder is also flexible to some extent since the decoder is allowed to decode the syntax in any way as long as the decoding process produces numerically identical results to the process specified in the standard. The flexibility enables the optimization of the encoding process to suit different applications. For example, in a video storage and reproduction application such as DVD, more emphasis can be given to maximize the video quality, whereas in a video telephony application, more emphasis can be given to complexity and implementation costs.

3.4 Features and Tools

The H.264 standard consists of various features and coding tools that contribute to the high compression efficiency, flexibility and robustness. This section describes the structure and some of the high level features of the standard.

3.4.1 Layer Structure

H.264 is designed to be flexible and customizable to handle a variety of applications and transport methods. To achieve the flexibility, the standard was designed to contain two layers.

1. The *Video Coding Layer* (VCL) represents the video encoding process which carries out actual video compression and the data which consists of coded bits.
2. The *Network Abstraction Layer* (NAL) handles the transportation of VCL data and other header information by encapsulating them in NAL units.

The separation of video coding and transportation into two layers ensures that the video coding layer provides an efficient representation of video content. The network abstraction layer transports the coded data and other header information in a flexible manner by adapting to a variety of delivery frameworks.

3.4.2 Profiles and Levels

Profiles and levels are used to specify the tools and capabilities of the decoder that is needed to support different applications and to provide interoperability points between different decoder implementations. Each profile is designed to have particular coding tools to support various coding requirements. The H.264/AVC standard originally specified the following three 'basic' profiles.

1. **Baseline**: low-latency, low-complexity, error resilience, and robustness. Applications: video conferencing.
2. **Main**: high compression efficiency. Applications: video storage and broadcasting
3. **Extended**: Superset of the baseline profile with enhanced error resilience and video stream switching capabilities. Applications: internet video streaming.

The fidelity range extensions introduced a new set of profiles called the 'High' profiles intended for high quality applications (e.g. HD-DVD, HDTV) and professional applications like studio editing.

Levels are defined as performance limits for decoders supporting each profile. Performance limits generally apply to processor load, memory capabilities and the maximum bit rates which in turn affect the frame sizes, frame rates and number of reference frames supported by a compliant decoder.

3.4.3 Picture Format

The source video is coded as a stream of pictures. The color spaces and the sampling formats of the pictures and the process of dividing a picture into coding units comprised of slices and macroblocks are discussed in this section.

3.4.3.1 Color Space and Sampling

The basic profiles support YCrCb 4:2:0 sampling format with 8-bit sample accuracy while the high profiles support 4:2:2 and 4:4:4 with up to 10 and 12 bit sample accuracy. The width and height of the luma sample array of a picture should be a multiple of 16, while the width and height of the chroma sample array is a multiple of 8 or 16 depending on the sampling format, so that the picture includes all the chroma samples associated with the luma samples. Both progressive and interlaced video are supported.

3.4.3.2 Macroblocks

The smallest coding unit in a picture is a *Macroblock* (MB). A macroblock contains data belonging to a region of 16x16 luma samples along with the associated Cr and Cb component samples. A picture should contain an integral number of macroblocks.

3.4.3.3 Slices

A picture consists of one or more slices. Each slice contains an integral number of macroblocks which should be processed in raster scan order. H.264 has the following slice types:

– **I-Slices:** All the macroblocks are coded using intra prediction. Macroblocks are coded using data already coded within the same slice (Intra).

- **P-Slices:** Contains inter coded macroblocks using one reference picture and/or intra coded macroblocks (Predictive).
- **B-Slices:** Contains inter coded macroblocks using two reference pictures as well as macroblock types in P-slices (Bi-predictive).
- **SP and SI-Slices:** Special types of slices, *Switching Predictive* (SP) and *Switching Intra* (SI), for efficient switching between different video streams, random access and error resilience [29].

3.5 Video Coding

The core video compression is carried out by the *Video Coding Layer* (VCL). The VCL compresses the pictures by dividing the pictures into one or more slices which contain an integral number of macroblocks. Macroblocks are the basic coding units of the H.264/AVC encoder. The basic architecture of H.264/AVC is similar to previous coding standards such as MPEG-2 and H.263 where a motion compensated block based transform is used to achieve compression.

Functional block diagrams of H.264 encoder and decoder are shown in figure 3.6 and figure 3.7 respectively. These figures show the high level functional elements which should be present in an encoder and a decoder which complies with H.264/AVC. The operation of the H.264/AVC encoder and decoder is briefly described here.

3.5.1 Encoder

The macroblocks in the current picture are processed as either intra or inter coded macroblocks. The encoder consists of a forward path (represented with thick black arrows in figure 3.6) for the encoding and a reverse path (grey arrows) for decoding and reconstruction of the current picture. A prediction signal for the macroblock is calculated using either intra prediction or inter prediction. In intra prediction, the current macroblock is predicted from the neighboring samples in the current slice which have been already encoded, decoded and reconstructed by the encoder. In inter prediction the prediction signal is obtained through motion estimation and compensation using one or two reference pictures from the refer-

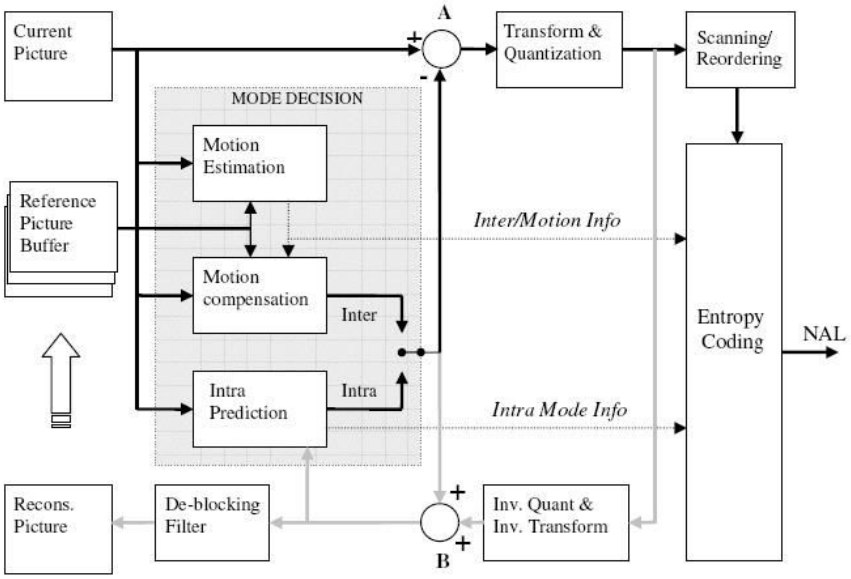

Figure 3.6: H264 Encoder.

ence picture buffer. The reference picture buffer contains previously coded and decoded pictures that can be selected for inter prediction.

The prediction macroblock is then subtracted from the original macroblock to create a residual macroblock at node A. The residual macroblock is transformed, quantized and reordered before entropy coding. Entropy coding is done to remove the statistical redundancy of the data. The entropy coder also processes other information necessary for correct decoding of the residual data such as the quantization parameter, macroblock partition modes, the reference frames used, motion vector information for inter coded macroblocks and intra mode information for intra coded macroblocks. The output of the entropy coder is compressed video bits which are encapsulated in NAL units before transmission or storage.

The objective of the reverse path (marked by thin arrows) is to reconstruct the lossy coded picture exactly the same as the decoder. The reconstructed samples of the neighboring macroblocks in the current slice may be used for intra prediction of macroblocks and the current reconstructed picture may be used for inter prediction of future pictures. The picture is reconstructed after applying a *deblocking filter* (DF) in order to reduce the blocking artifacts appearing due to

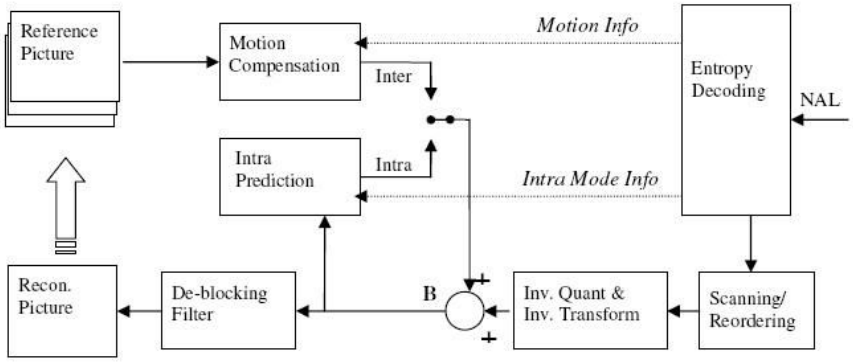

Figure 3.7: H264 Decoder.

quantization of block transforms.

3.5.2 Decoder

The decoder block diagram is shown in figure 3.7. Starting from the right hand side, NAL units are the input of the decoder. The NAL units are first entropy decoded to obtain the quantized coefficients and other information necessary to reconstruct the macroblocks using the quantized coefficients. Inverse quantization and inverse transform are applied to the coefficients to produce the residual macroblock. For inter coded macroblocks, a prediction is obtained by carrying out motion compensation using the decoded information such as macroblock partition modes, reference pictures and motion vectors. Intra coded macroblocks are predicted using the decoded intra mode information and previously decoded pixels of neighboring macroblocks. The macroblock is reconstructed by adding the prediction to the residual at node B. The deblocking filter is applied to reconstruct the current picture. The reconstructed picture is displayed and may also be used as a reference picture for decoding future pictures.

3.6 Coding Tools and Functions

The H.264/AVC standard offers a wide range of coding tools to achieve a high level of compression efficiency. Some of the important coding tools and functions

of the H.264/AVC standard will be discussed in this section.

3.6.1 Intra Prediction

In intra prediction, the prediction signal is produced using the neighboring samples of previously encoded and reconstructed blocks which are located on the left of and above the current block. Therefore, intra prediction exploits spatial correlation of image pixels. The following intra coding modes are supported in all slice types. Note that intra prediction is not carried out across slice boundaries. Therefore, slices can be decoded independently of each other to limit error propagation.

3.6.1.1 Intra 4x4 prediction for luma samples

Intra prediction is carried out for each individual 4x4 block of the macroblock. The small prediction block sizes are particularly useful for areas which have high detail. The pixel values of each 4x4 block are predicted from the neighboring pixel values.

3.6.1.2 Intra 16x16 prediction for luma samples

The samples of the macroblock are predicted without partitioning. This is useful for homogeneous areas that do not contain much detail. A block of 16x16 samples and the corresponding left and above samples are used in the prediction process. There are four intra 16x16 prediction modes which are similar to corresponding modes of intra 4 x 4:

1. **Intra 16x16 Vertical:** pixel values of the macroblocks are predicted from the pixels just above the macroblock.
2. **Intra 16x16 Horizontal:** pixel values are predicted from the pixels to the left of the macroblock.
3. **Intra 16x16 DC:** every pixel of the macroblock is predicted from the mean of upper and left neighboring samples of the macroblock.
4. **Intra 16x16 Plane:** Pixels of the macroblock are predicted using a linear equation that uses both above and left pixels.

3.6.1.3 Intra prediction for chroma samples

The chroma samples are considered to be more homogeneous than luma samples and therefore, chroma intra prediction is always done on macroblocks without partitioning. The same prediction mode is used for both Cr and Cb components. There are four chroma prediction modes which are similar to intra 16x16 modes. However, the exact prediction process is specified for different chroma formats due to the difference in chroma macroblock size. For 4:2:0 sampling format the chroma macroblock size is 8x8.

3.6.1.4 I_PCM

This is a lossless coding mode where the image sample values are transmitted directly without prediction, transform or quantization. Although this is a very inefficient method of coding, this method is useful to represent image regions without any loss.

3.6.2 Inter Prediction

Inter prediction is carried out to exploit the temporal redundancy between pictures. Block based motion estimation and compensation is carried out in order to create the inter prediction signal. The inter prediction tools contribute significantly to the improved compression efficiency of the H.264/AVC standard over previous coding standards. Some of these tools are discussed here.

3.6.2.1 Variable block size motion compensation

Motion compensation is carried out for macroblocks by dividing the macroblocks into partitions and sub-macroblock partitions. Figure 3.8 shows how the luma component of a macroblock can be partitioned for motion compensation. Each macroblock can be partitioned into one 16x16 (whole macroblock), two 8x16, two 16x8 or four 8x8 partitions. Each partition is individually motion compensated using a separate motion vector.

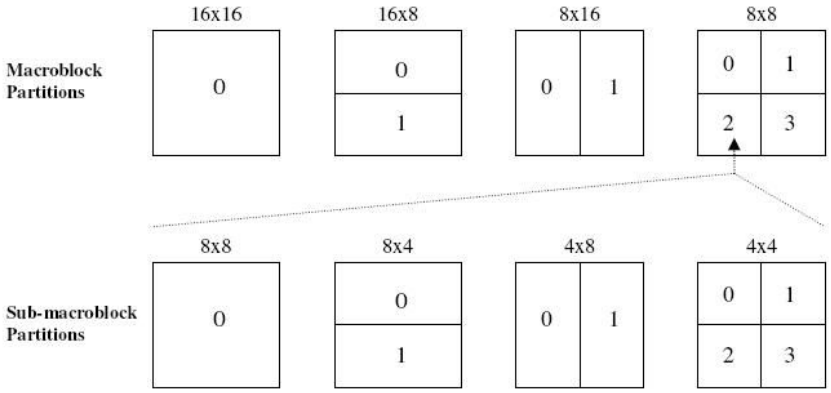

Figure 3.8: Macroblock and sub-macroblock partitions.

3.6.2.2 Quarter pixel accurate motion vectors

Motion estimation and compensation is carried out by generating a prediction signal for each macroblock or sub-macroblock partition from the reference picture. Motion vectors indicate the relative position of the matching area in the reference picture. In H.264/AVC, motion vectors have luma quarter pixel accuracy. Therefore, the reference picture is interpolated to represent sub sample and quarter sample pixel positions. These pixel positions are obtained by linear interpolation between four neighboring samples.

3.6.2.3 Motion vector prediction

H.264/AVC allows motion vectors to point to regions outside the picture boundary. The pixels of the outside region are obtained by extrapolating the pixel values at the picture boundary. This allows for effective motion compensation of objects moving in or out of the picture boundary.

Encoding of motion vectors may result in large number of bits, in particular because there can be a number of motion vectors corresponding to a number of small partitions used for motion estimation. Therefore, motion vector prediction is used to reduce the number of bits needed to transmit the motion vectors.

The motion vector of the current partition is predicted (MVP) from the motion vectors of the neighboring partitions if they are available. Figure 3.9 shows the

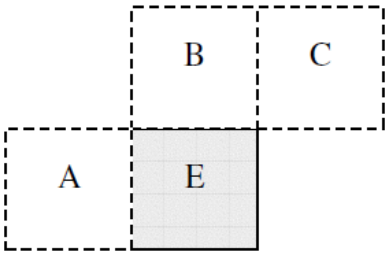

Figure 3.9: Current and neighboring blocks (macroblock partition) used for motion vector prediction.

neighboring blocks that are used for motion vector prediction. The shaded block E is the current partition and the blocks A, B and C are the neighboring partitions.

Only the *Motion Vector Difference* (MVD), which is the difference between the actual motion vector for the current partition and the *Motion Vector Prediction* (MVP), is transmitted. The number of bits needed for the motion vectors can be reduced due to high correlation between the motion vectors of neighboring blocks.

3.6.2.4 P and B slices

Macroblocks in P-slices are inter-predicted using one reference prediction with a reference picture selected from the reference picture 'list0'. Macroblock in B-Slices can have one or two motion vectors. Macroblock partitions can be predicted from a reference picture in 'list0' or in 'list1' where only one motion vector and reference index is used. Macroblock partitions can also be bi-predicted from two reference pictures, one from 'list0' and one from 'list1' and therefore two motion vectors and reference indexes are used. When weighted prediction is not used, the average pixel values of the two reference predictions are used as the bi-prediction signal. If weighted prediction is used, bi-prediction pixel values are obtained as the weighted average of the two reference predictions. There is also a special mode for 16x16 partition size called the direct mode where no motion vectors or reference picture indexes are sent. They are derived from the macroblocks that have already been decoded.

3.6.3 Transform and Quantization

A residual macroblock is generated by subtracting the prediction from the original macroblock. The residual macroblock is transformed to remove the spatial correlation. The Baseline, Main, and Extended profiles only use an 4x4 integer transform [35, 50] which is based on the DCT, to transform the residual data of the macroblock by dividing the macroblock into 4x4 blocks. The integer transforms can be carried out using integer arithmetic and are less complex than the DCT [36, 62]. Since no floating point arithmetic is used, the possible mismatch between the forward and reverse transform is eliminated.

Lossy compression is achieved by quantizing the transformed residual data. The *Quantization Parameter* (QP) specifies the quantization step size. Each macroblock can be encoded using different quantization parameter values. The QP is differentially coded and therefore only the change in QP is transmitted to the decoder.

3.6.4 Skipped Macroblocks

H.264/AVC specifies a special type of macroblocks called skipped macroblocks. For skipped macroblocks, no coded information is sent to the decoder. A syntax element in slice data indicates the skipped macroblocks to the decoder. Skipped macroblocks in P-Slices and in B-Slices are called P-Skip and B-Skip respectively.

The decoder does not receive any motion information or residual data for the skipped macroblock. Since the motion vector differences are zero, the motion vector prediction becomes the actual motion vectors used to obtain the predicted macroblock. Therefore, the prediction macroblock is simply copied as the reconstructed macroblock.

Typically, skipped macroblocks occur in regions with low movements, and therefore, the predicted macroblock is very similar to the original macroblock. The residual data of this type of macroblocks is low resulting in zero coefficients after transform and quantization.

3.6.5 Deblocking Filter

The quantization of block transform coefficients can lead to visible blocks edges in the reconstructed picture. The H.264/AVC standard specifies an in-loop *deblocking filter* (DF) to minimize the blocking artifacts. The deblocking filter is applied in-loop, meaning that the reconstructed and filtered pictures are used as reference pictures for inter prediction. The same filter parameters are used at both the encoder and the decoder to avoid any prediction errors. Typically a filtered picture provides a closer match to the original picture than the unfiltered reconstruction. Therefore a better prediction can be obtained using the filtered reference picture, resulting in higher objective and subjective quality. The filter is applied over 4x4 block boundaries in macroblocks and the filter strength depends on the quantization parameters, prediction modes of neighboring blocks and the actual pixel values across the boundary [34]. In addition, the filter strength can be explicitly changed or the filter can be completely turned off by the encoder.

3.7 Parallel Implementations

Ever since the H.264/AVC standard [26] was published in 2003, researchers started to solve the high complexity issue of the new standard mainly using parallelism. Several modifications were suggested for the H.264 encoders and decoders to improve the performance in terms of execution time and memory usage. Parallel decoding techniques of H.264 starts from the highest level, which is the group of frames or pictures (GOP), coarse-grain level, till the lowest level which is the block inside a macroblock, fine-grain level.

3.7.1 Slice-Level

Gurhanli et al. [20] suggested a parallel approach by decoding independent groups of frames on different cores. The speedup is conditioned with the modification of the encoder in order to omit the start-code scanner process. Any modification to the encoder will require the exclusion of previously encoded video sequences which will need to be re-encoded in order to benefit from the proposed approach. In our parallel implementation, we only modify the decoder which sup-

port all previously encoded video sequences. Nishihara et al. [42] proposed a load balancing mechanism among cores where partitions sizes are adjusted at run-time. The authors also reduced the memory access contention based on execution time prediction. Horowitz et al. [22] compared different H.264 implementations including FFmpeg [17] and the H.264 reference software JM [61]. The authors also analyzed the complexity of the H.264 decoder subsystems.

3.7.2 Macroblock-Level

Kannangara et al. [28] reduced the complexity of the H.264 decoder (19-65%) by predicting the SKIP macroblocks using an estimation based on a Lagrangian rate-distortion cost function. Our experimental results show a better overall speedup (230%) and a better parallel scalability relative to the number of cores in a multicore processor. Zhao et al. [72] proposed a wavefront algorithm for processing independent macroblocks within the same frame and among different frames. This method for parallel processing of macroblocks does not equally distribute workload of different cores as the number of independent macroblocks varies with time. Mesa et al. [39] proposed a similar approach, the 2D-Wave, which decodes independent macroblocks in parallel on different cores. A good scalability is proved for high resolutions. Moreover, an advanced parallel technique that is based on the 2D-wave algorithm, the dynamic 3D-Wave approach, is proposed by Meenderinck et al. [38]. The dynamic 3D-Wave algorithm, which combines spatial and temporal MB-level parallelism, uses a dynamic scheduler that assigns independent macroblocks to parallel threads. The dynamic scheduler minimizes the differences in workload on different threads, and thus, it optimizes the parallel execution of independent macroblocks on parallel threads. Chong et al. [14] added a pre-parsing stage in order to resolve control dependencies for macroblock-level parallelism. Vandertol et al. [67] mapped video sequences data over multiple processors providing better performance over functional parallel algorithms. The authors group macroblocks in a frame with minimal dependency between cores.

3.7.3 Deblocking Filter

Among the literature that already exists for parallel deblocking filter, Sihn et al. [57] proposed a multicore pipeline for the deblocking filter based on the group of pictures data level partitioning. He also suggested software memory throttling and fair load balancing techniques in order to improve multicore processors performance when several cores are used. Wang et al. [68] partitions a slice into independent rectangles with arbitrary granularity. These independent regions are identified by examining the influence of vertical and horizontal lines of pixels. Parallel deblocking of these regions has good scalability, minimal synchronization overhead, and good cache utilization. However, a small number of pixels will have erroneous output without affecting the overall deblocking filter process with what they refer to as the Limited Error Propagation Effect. For an optimized deblocking filter, a speedup of 95% and 224% is achieved on 2 cores and 4 cores respectively. For an H.264 decoder, the overall speedups are 21% on 2 cores and 34% on 4 cores. Pieters et al. [49] proposed a macroblock partitioning algorithm that is based on a parallel version described by Wang et al. [68] with the avoidance of the Limited Error Propagation Effect. The proposed algorithm filters the pixels of macroblocks concurrently. The parallel technique is also tested on GPU platforms. The parallel implementation outperforms both CPU-based and GPU-based implementations by a factor up to 10.2 and 19.5 respectively.

3.7.4 Discussion

Many optimization techniques are proposed in order to increase the efficiency of H.264 video standard. A straight-forward comparison between different literature is not applicable because of many criteria and assumptions adopted in these researches. In addition to software implementation diversities, hardware platforms are rarely similar which makes direct comparisons unreliable. Some assumptions cannot be applied in both our work and related work. In our experimental results in the following chapters, we compare values with related work using almost similar units and hardware platforms.

3.8 Summary and Conclusion

The design of the H.264/AVC standard is targeted at a wide rage of applications from video conferencing to HDTV and professional studio editing applications. Therefore, as mentioned earlier in this chapter, the standard defines a set of 'profiles' that include subsets of available coding tools and features targeted at different application scenarios. Table 3.2 indicates the features and coding tools contained in the baseline, main and extended profiles.

The H.264/AVC video coding standard delivers significantly improved compression efficiency compared with previous standards, supporting higher quality video over lower bit rate channels. Due to improved compression efficiency and increased flexibility of coding and transmission, H.264 has the potential to enable new video services such as mobile video phones and multimedia streaming over mobile networks. The H.264/AVC standard supports a wide range of applications from consumer applications like video conferencing to professional applications like video editing. The H.264/AVC standard has a range of coding tools contributing to its high compression performance, flexibility and robustness. However, the performance improvements come at a cost of significantly high computational complexity. Therefore, the decoder implementations should make use of the available coding tools effectively to achieve the desired compression performance with the available processing resources.

In the following chapters, we describe in detail efficient and scalable parallel implementations of the H.264 decoder in order reduce execution time and energy consumption.

Table 3.2: Features of the Baseline, Extended, Main, and High profiles

Feature	Baseline	Main	High
Bit depth	8	8	8
Chroma formats	4:2:0	4:2:0	4:2:0
Flexible macroblock ordering (FMO)	Yes	No	No
Arbitrary slice ordering (ASO)	Yes	No	No
Redundant slices (RS)	Yes	No	No
Interlaced coding (MBAFF)	No	Yes	Yes
B slices	No	Yes	Yes
CABAC entropy coding	No	Yes	Yes
4:0:0 Monochrome	No	No	Yes
8x8 vs. 4x4 transform			
adaptivity	No	No	Yes
Quantization scaling matrices	No	No	Yes
Separate Cb and Cr QP control	No	No	Yes
Remarks	low complexity high robustness error resilience	high compression efficiency	high compression efficiency
Applications	video conference web videos	digital TV media storage	high resolution HD TV & DVD

Chapter 4

H.264 Color Components Parallel Decoding

4.1 Introduction

Multimedia applications are found in almost every device in our modern technological world. Mobile devices like cell phones and PDAs are essential in our daily life needs. Their growing features require better hardware performance, larger storage, and smoother display resulting in higher power consumption. As systems performance and workload are remarkably increasing, so do network communications. Massive amounts of data are interchanged daily using various wireless networks and infrastructures. Consequently, improved compression algorithms are needed to minimize the size of video files, and thus, benefiting from the growing performance of embedded processors. Enhanced compression will in turn require higher processing power which may affect the overall performance of the embedded device.

Nowadays, most processors for desktop computers and mobile devices are multicore chips. The adoption of multicore processors improved the multi-tasking user experience of all applications. Nevertheless, the majority of existing multimedia applications do not benefit from multicore processors because they are designed to be executed sequentially like the open-sourced H.264 reference software [61] and the FFmpeg codec [17]. Parallelizing the H.264 decoding process offers a huge

credit to existing embedded systems enabling them to decode video sequences with higher resolution more efficiently by benefiting from existing and unused hardware resources like unused cores, private and shared memories, etc.

In this research, we propose our approach that processes each color component (luma and chroma) on a separate core in a multicore processor in order to increase the overall performance of the H.264 decoder. Our novel idea is based on the fact that the H.264 decoder process color components in every frame sequentially; thus, simultaneous processing of the color components is possible due to the reason that luma and chroma processing are independent. In addition, a pipeline version is designed in order to improve load balancing and to hide synchronization overhead. Simulations are conducted using video sequences benchmarks that are simulated on multicore embedded processors. Experiments are conducted on dual and quad core processor simulator in order to collect execution time and energy consumption statistics. [9]

The rest of the chapter is organized as follows. Section 4.2 presents our technique for decoding color components of video frames in parallel. Section 4.3 displays and analyzes experimental results using the H.264 reference software, JM [61]. Section 4.4 shows the results for a parallel H.264 using the FFmpeg codec library [17]. Finally, section 4.5 concludes the chapter.

4.2 Parallel Decoding

In this section, we decompose the H.264 decoder process. Then we explain our algorithm for parallel processing of color components. The H264/AVC [26] video decoder and its features are explained in depth in chapter 3. We also describe our proposed pipeline implementation of the H.264 decoder.

4.2.1 Stages Decomposition

A thorough check of the reference implementation for H.264 codec [61] shows that the decoder can be divided into five main functional parts: *entropy decoding* (ED), *de-quantization* and *inverse transform* (IQT), *motion compensation* (MC) and *intra-prediction* (IP), and *deblocking filter* (DF). Figure 4.1 illustrates a sim-

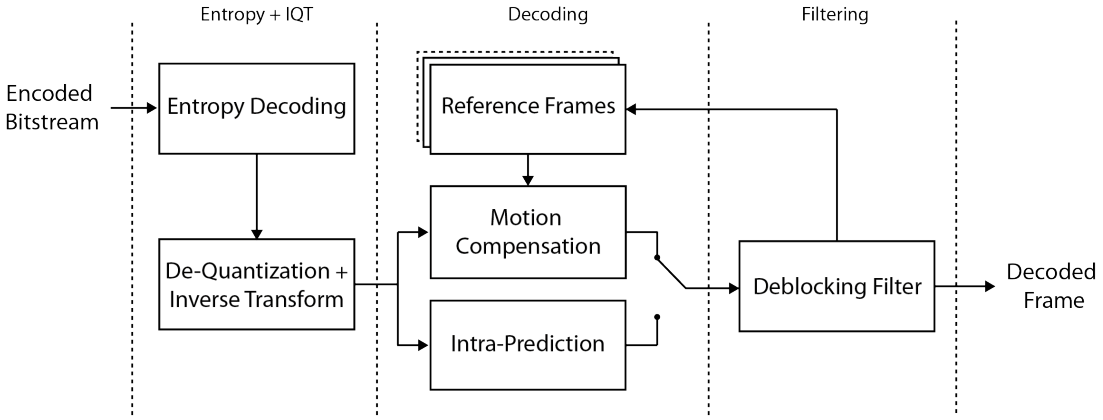

Figure 4.1: Simplified H.264 decoding process

plified representation of the H.264 decoder's stages. Entropy decoding and motion compensation are applied for every macroblock of size 16x16 pixels. Deblocking filter is executed at the end of the decoding process. The average workload of every stage using the baseline profile is shown in figure 4.2. The entropy decoding and the de-quantization and inverse transform stages are merged into one stage in the statistics in figure 4.2. The workload of this stage is 14% on average which is mainly consumed by the *context-adaptive variable length coding* (CAVLC). The CAVLC algorithm is adopted by the baseline profile and it has a lower complexity than the CABAC algorithm which is explained in the previous chapter. The prediction stage which is composed of intra-prediction and motion compensation has a high impact on the overall decoding process which 41% on average. Finally, the deblocking filter stage is also a heavy process that consumes around 45% of the overall process. These workload statistics are profiled using several video benchmarks with low and high resolutions that are listed in the experiments section.

4.2.2 Color Components Processing

The frames in a video sequence are represented as bit streams. Pixels are sampled using three color components: YUV (or YCrCb). Y stands for luminance color sample (luma) which is the light info. UV (or CrCb) stands for the red and blue color samples respectively (chroma). In a 4:2:0 format, each four luma

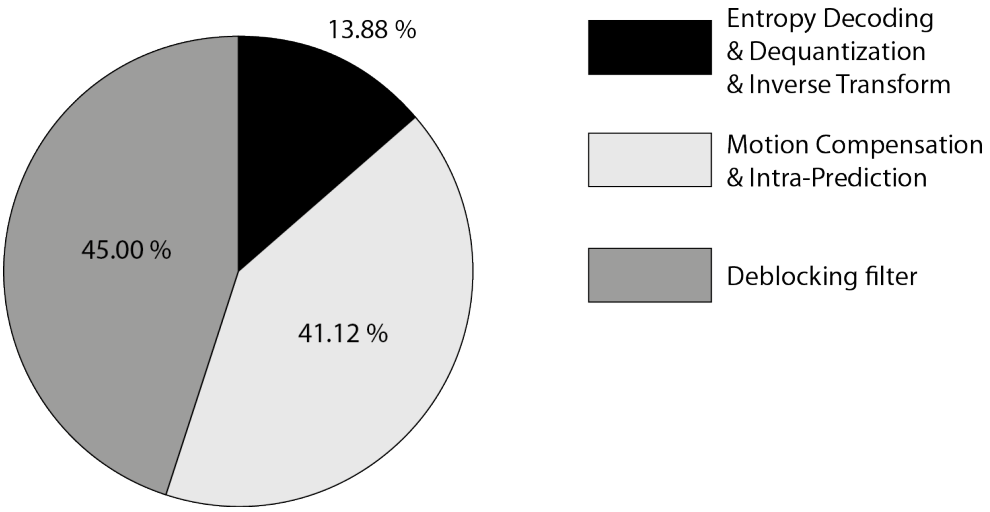

Figure 4.2: H.264 decoding stages workload percentages of the baseline profile.

samples have one red sample and one blue sample as shown in figure 4.3. The 4:2:0 sampling format is the most widely used format. Other formats, 4:2:2 or 4:4:4, where more color samples are available, shows no significant difference to the human vision. The reason is that the vision is more affected by the light than the colors of video sequences.

In our research, we reveal an independent pattern which is found in the decoding process of color components. As we described above, each frame of a video sequence is represented in YCrCb color samples. A pixel is formed by these three color components. The decoder reconstructs each frame picture of each color separately starting with the luma then the chroma colors. The color information in each frame and thus in each macroblock are independent from each other. The H.264 Standard [26] does not show any dependency between the color information data of the decoding algorithm during the motion compensation stage.

4.2.3 Parallel Execution and Synchronization

Parallel execution is considered as a major potential solution for complex algorithms where available resources like parallel cores are being used without any modifications to hardware components. These additional cores are available in most devices nowadays where sequential applications do not effectively benefit

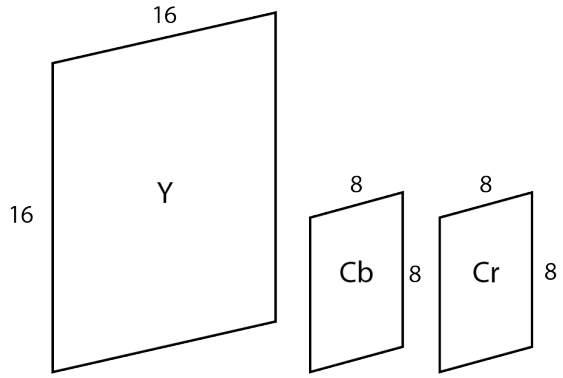

Figure 4.3: YUV 4:2:0 color components sampling format

from multiprocessor systems. Performance optimization using parallel execution can be applied to many extensive processing applications. Even optimized implementations can still take advantage of parallelism using multicore processors.

The H.264 decoding process and all major video decoding algorithms are generally designed to be executed sequentially. The H.264 standard [26] does not support parallelism, and thus, does not benefit from multicore processors that are available in today's market. Several approaches have been studied in order to parallelize the execution of the decoding process. Most of these approaches are based on slices and macroblocks (which are explained in section 3.4.3.3 and 3.4.3.2 in chapter 3) parallel processing [14, 51, 67]. Similar approaches for parallel processing of H.264 are described in details in section 3.7 of chapter 3.

In our research, we modified the H.264 source code in order to decode the luma and chroma components of macroblocks in parallel. The parallelization uses the PThread library with critical sections mutual exclusion and condition variables. One core handles all the stages except the chroma motion compensation and intra-prediction which are executed on the second core. As shown in figure 4.4, the second core handles the motion compensation and intra-prediction for chroma color samples only. The first core executes the luma color samples in addition to all the remaining decoding stages. As stated above, color components are independent of each other. Therefore, decoding different color components in parallel is proved to be correct in theory as well as in experiments. Thus, parallel processing of color components increase the performance of the decoder when it

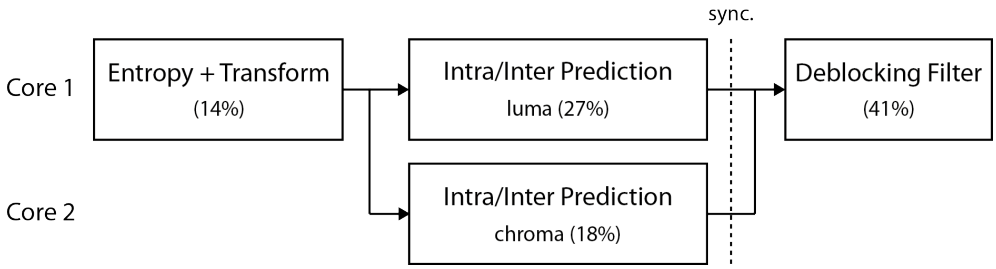

Figure 4.4: H.264 parallel color components decoding on a dual core processor

is executed on a multicore system.

The intra-prediction and motion compensation (inter-prediction) should be completed before applying the deblocking filter. Thus, a synchronization barrier is needed before the deblocking filter stage. With this configuration, the synchronization is performed at the end of the luma and chroma decoding using condition variables. At the end of the entropy decoding stage, the parallel execution step of the decoding process is initiated. Once intra-prediction and motion compensation are completed, all parallel threads will wait at the synchronization barrier before starting the deblocking filter stage.

A complex structure variable, which contains all the information needed to represent a picture frame, is saved in the shared memory. This picture variable can be accessed by all cores. The communication between multiple cores is implemented using a memory that is shared by different cores. When using a dual-core processor, core 2 uses the chroma data in order to decode the color components while core 1 decodes the luma data and executes the remaining sequential algorithms as illustrated in figure 4.4. The average workload of the second core using the Akiyo and Container benchmarks (CIF and QCIF) is 18% which is in turn the average performance gain using dual-core processors.

Figure 4.5 illustrates the workload partitioning over 4 cores. The first core reads the data from NAL units (which are explained in section 3.4 of chapter 3)and it performs the entropy decoding and transformation stages. The second core processes luma data while the third core processes chroma data. They both perform the same work, intra-prediciton and motion compensation, on different color components at the same time. The fourth core executes the remaining part of the deblocking filter and then it outputs the decoded frame picture. The per-

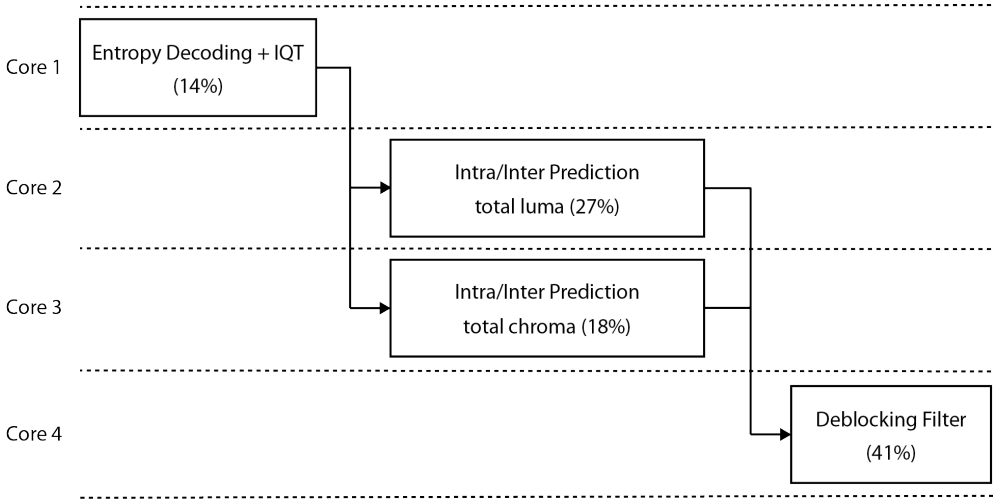

Figure 4.5: H.264 parallel color components decoding on a quad core processor

formance gain using quad core processors is almost the same as the dual core processors due to the sequential characteristics of the H.264 decoder. Thus in order to benefit from quad core architectures, a pipelined execution is proposed and discussed in the following section.

4.2.4 Pipeline Execution

In order to minimize the waiting time between parallel cores, the H.264 decoding stages are executed in pipeline mode over four cores when motion compensation is used. Theoretically, the execution time can be dramatically decreased to the time needed by the core that executes to biggest chunk of code. The processor idle time is reduced leading to higher efficiency by using available resources. The pipeline is illustrated in figure 4.6.

A shared memory storing the blocks of information is used in order to access the consistent data by the four processors. The data variables and their manipulation in the current H.264 implementation went through extensive modification and testing in order to prove the proposed pipeline execution. Figure 4.6 illustrates the best case scenario where the four decoding stages are completely independent, allowing parallel execution using the four cores. This case is applied when the current frame is dependent on a previously decoded frame (P-frames).

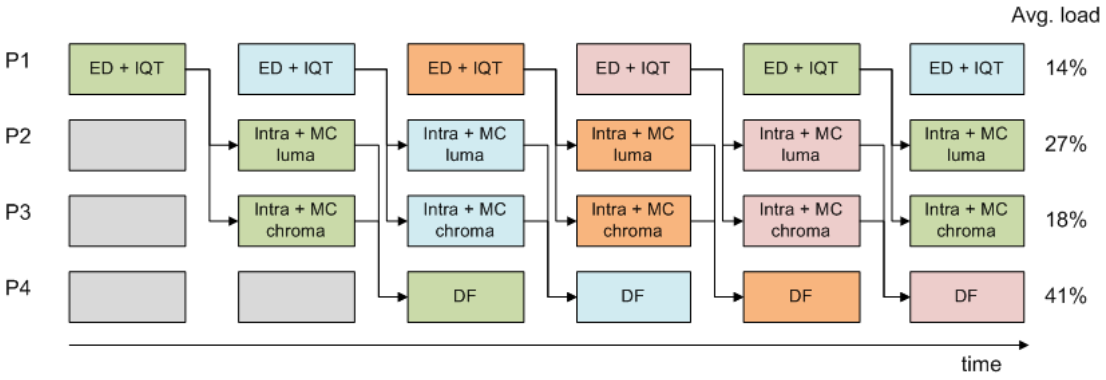

Figure 4.6: H.264 pipeline execution on a 4 cores multiprocessor

P-frames contain intra and inter macroblocks. Intra-prediction frames (I-frames) do not allow pipelined decoding execution because macroblocks depend on other macroblocks in the same frame. I-frames contain only intra macroblocks (I-MBs). Performance gain depends on the number of I-frames in the encoded video frames where the first decoded frame is always an I-frame. On the other hand, subsequent frames encoded using the Baseline or the Main profiles are mostly P-frames considering the fact that there is a motion in the video sequence. I-MBs are useful when adjacent frame are pretty similar and minor motion took place. For the Akiyo and Container benchmarks, only the 1st frame was decoded using intra-prediction among 300 frames. This fact leads us to concur that at least 99% of the decoded frames are P-frames. The maximum load of the H.264 pipelined version over 4 cores is 41% which is executed by the fourth core (P4) in order to perform the deblocking filter process. So the maximum performance speedup is limited by the largest load which is the deblocking filter.

4.3 Experiments with JM H.264 Reference Software

In order to demonstrate the feasibility of our approach, we have performed experiments on our parallel version of the H.264 reference decoder [61] using the MPARM simulator [31]. Runtime statistics of each stage are collected in addition to overall execution results.

Table 4.1: H.264 decoder profiling based on luma and chroma

Benchmark	Total (ms)	Luma (%)	Chroma (%)
Akiyo CIF	754234	29.48	20.06
Akiyo QCIF	379928	24.56	15.93
Container CIF	763781	29.45	20.45
Container QCIF	397664	24.55	15.91
Average		27.01	18.08

4.3.1 MPARM simulator and H.264 porting

MPARM is a multiprocessor cycle-accurate architectural simulator [31]. RTEMS is a real-time operating system for embedded multiprocessor systems [15]. An H.264 ported version of the RTEMS operating system runs on MPARM simulating ARMv6 embedded multicore processor [12]. H.264 reference software [61] is designed to run on desktop systems. Thus, a preliminary step is to port the reference software so that it can be executed by the RTEMS operating system which in turn runs on the MPARM simulator.

4.3.2 Profiling H.264 Stages

The parallel H.264 decoder is executed using the MPARM simulator. Two benchmarks are used, Akiyo (news presenter) and Container (slow moving cargo ship). The encoded video sequences have two formats, QCIF (176 x 144) and CIF (352 x 288). The simulator profiles each stage of the decoder and each luma and chroma components in each phase. Outputs of the simulator include the number for cycles and the execution time for each part of code that we specified for every stage of the H.264 decoding process. Table 4.1 lists the total execution time in milliseconds for each video sequence. In addition, the respective percentages for luma and chroma processing are displayed.

4.3.3 Discussion

The test results in table 4.1 show a difference between the CIF and the QCIF formats. The motion compensation percentages from the total time of execution are quite similar with a 1% difference between video sequences of the same format.

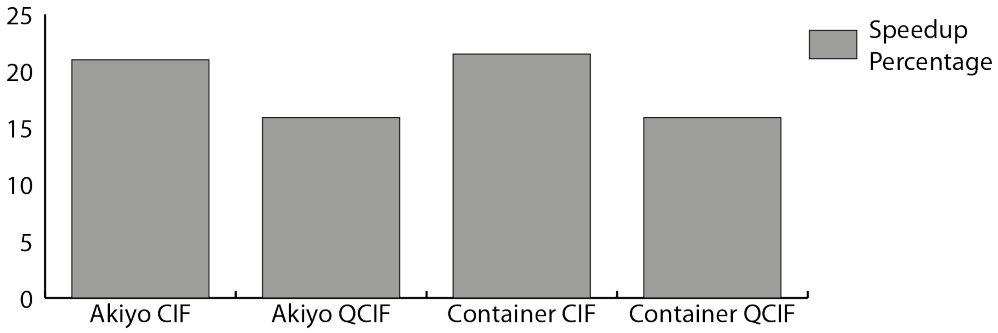

Figure 4.7: Execution speedup per benchmark and resolution

The Akiyo and the Container benchmark have a 5% difference between CIF and QCIF. The deblocking filter, last stage of the decoder, is divided into three parts: strength calculation, deblocking macroblocks, and edge filtering. The strength, which is the amount of filtering, depends on the boundaries differences between macroblocks and on the gradient of image sample across the boundary. The wider the difference is between pixel information across macroblocks boundaries the more complex the strength calculation.

Color component differences between the two video sequences and across all stages are similar as shown in table 4.1. For a 4:2:0 sampling, each 4 luma sample are grouped with 2 chroma samples. Thus, luma processing needs more time than chroma leading to a double time difference at least. By grouping all the luma color information together summed up to 27% on average of the total execution time. However the total chroma execution time is 18% on average of the total execution time.

4.3.4 Speedup using Parallelism

Careful inspection of the source code and the algorithms in the H.264 implementation allows us to conclude that the luminance and chrominance processing and manipulation are completely independent within each slice. Thus, the execution the chroma decoding block of code on a different core simultaneously with the execution of the luma decoding process eliminates 16% of the total execution time for QCIF resolution and 20% for CIF resolution. The total execution part that can be parallelized using color component is limited by the chroma execu-

tion time. This part of the code varies between different video formats. The total performance gain that is achieved using parallel color decoding ranges between 15 - 21% as shown in figure 4.7.

Our proposed parallel H.264 implementation enables embedded devices, which are available in today's market, to benefit from multicore processors in order to increase the video decoding performance and to decrease power consumption. No specific and additional hardware is needed to use our algorithm. The H.264 codec is being widely adopted by most manufacturers for low resolution devices using the baseline profile. High definition resolutions mainly use the Main and the High profiles. Decreasing the decoding process time has many benefits on the user-experience level and hardware performance level. In addition, lowering the execution and processing time extends the battery life of the mobile device. The user enjoys watching higher quality video while the hardware consumes less resources regarding to processing time and power.

4.4 Experiments with FFmpeg H.264 Decoder

The simulator that was used in the previous section is limited to a small number of parallel cores. In this section, we experiment our proposed parallel color components algorithm using the Multi2Sim multicore simulator [66] with the FFmpeg H.264 implementation [17].

4.4.1 Multi2Sim Simulator

Multi2Sim simulator supports multithreading and multicore processors. The cache and memory configurations comply with ARM Cortex9MP processor. The FFmpeg H.264 decoder [17] is used as the main application that is being exploited on multicore processors. The simulator collects several statistic factors including the total number of instructions and cycles, reads and misses, and memory usage. We used 3 video benchmarks with CIF resolution (352x288) and 3 benchmarks with WXVGA (HD) resolution (1280x720). We simulated the H.264 decoding process of 30 frames for each benchmark.

The use of a different is due to the limitation of the MPARM simulator to

simulate more than 4 cores and due to its straightforward execution of C programs. M2S comprehensive pipeline and memory statistics are also easily mapped by the McPAT power estimation tool [33].

4.4.2 FFmpeg H.264 Implementation

FFmpeg [17] is an optimized implementation that supports most common video and audio formats. It is still evolving by open source developers experts. Many researches were made on different H.264 implementations which vary extremely in terms of performance and reliability. FFmpeg is considered as one of the fastest video codec implementations in terms of performance and reliability. The library is open sourced and it is licensed under the GNU Lesser General Public License (LGPL). FFmpeg decode most existing open and proprietary multimedia formats. The source code is implemented with a modular design using C language. It also includes many hardware specific optimizations available for particular processors. Several video codec with similar functionality can access the same code without rewriting or copying the required code. For example, H.264 uses many functions implemented for the MPEG-2 codec. Modifying existing code in general is not easy; however, adding new standards to the library is much easier than rewriting the whole implementation.

4.4.3 Speedup using Parallelism

Comparing the sequential execution on 1 core and the parallel execution on 2 cores shows an increase in performance around 18%. Video sequences with fast moving objects usually have a lower performance increase. This difference in speedup is mainly due to the large number of macroblocks that depends on previous reference frames resulting in more data communication between cores. In addition, macroblocks may be divided into sub-blocks reaching the size of 4x4 pixels. As a consequence for the overall speedup, synchronization overhead is added to the whole execution. An average of 3% of instructions is added to handle synchronizations between cores. Having calculated the overhead, the net performance speedup for parallel execution has an average of 12% as displayed in figure 4.8. An important similarity of speedup is noticed between CIF and HD resolutions. The

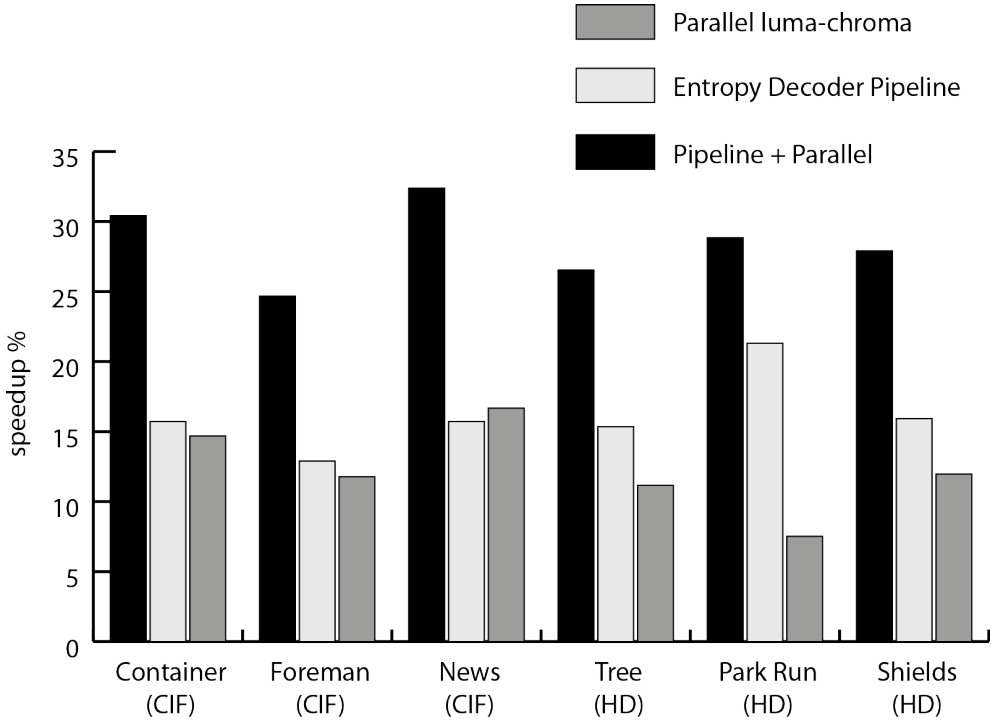

Figure 4.8: Speedup for parallel luma and chroma decoding, pipelined entropy decoder, and combined pipeline and parallel decoding.

parallel implementation seems unaffected by video sequences resolution. However it is mainly affected by the complexity of moving objects in the frames. For example, Shields, Into Tree, and Foreman benchmarks share the similarity of having a moving object in the middle with a slow moving background. Park Run has the lowest performance gain while News achieves the highest speedup.

Applying the pipeline structure illustrated in figure 4.6, we get an important increase in performance by significantly reducing the execution time of entropy decoding. Figure 4.8 displays the speedup gained with the combined structure ranging from 24% to 32% with an average of 29%. Experiments are performed using CIF and HD formats. We also notice that the speedup is indirectly proportional between parallel and pipeline approaches. As the speedup with parallel luma-chroma execution increases, then the speedup with pipelining decreases. This is mainly due to the complexity of the video sequences. When the objects are more complex, the entropy encoding compression efficiency is lower. So the

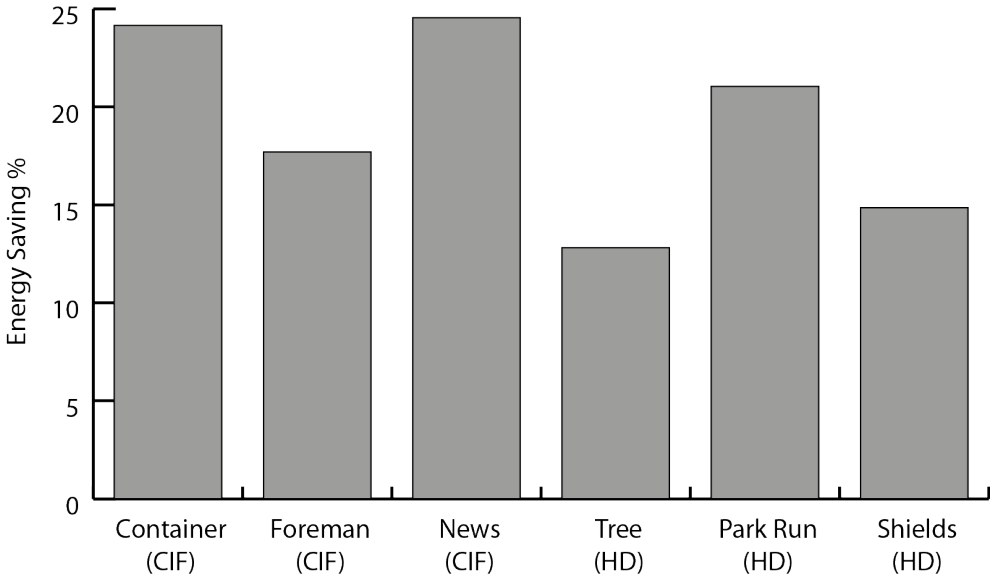

Figure 4.9: Energy consumption decrease with parallel-pipeline decoding.

time needed for entropy decoding becomes higher.

4.4.4 Power Efficiency

One of the most important factors in computer processing nowadays is power efficiency. New chips aim to achieve lower power consumption as display screens become wider and programs require more processing. Multicore processors need more energy at the expense of more processing performance. In our parallel H.264 implementation, overlapping instructions are executed at the same time. The total number of instructions and the numbers of loads and stores are increased. On the other side, the total number of cycles and the total execution time is decreased. Figure 4.9 plots the percentage decrease in energy consumption of the H.264 parallel implementation using 2 cores over the original sequential implementation on 1 core. The average percentage saving is 19%. High definition video sequences have higher power consumption. Power measurements are generated using the McPAT tool [33].

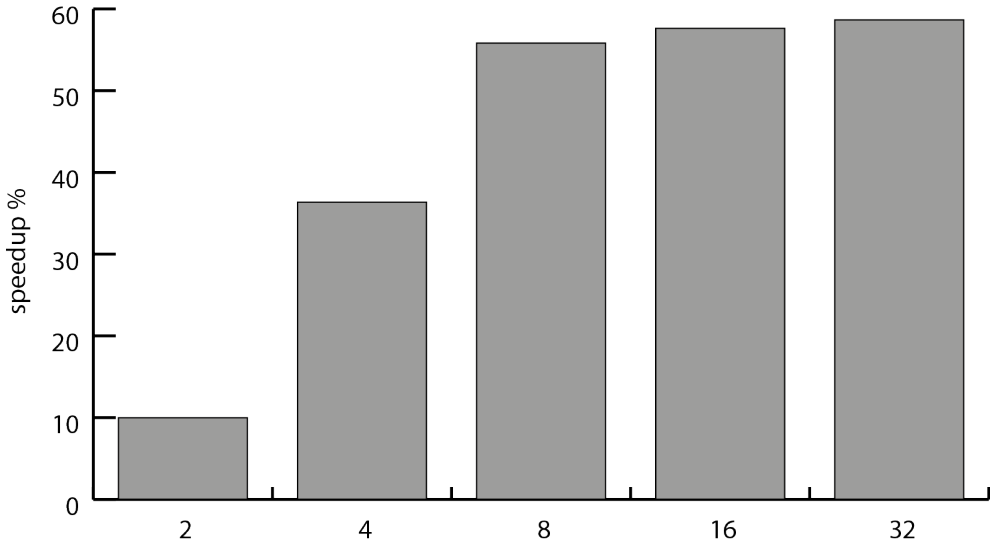

Figure 4.10: Speedup increase for FFmpeg multithread version.

4.4.5 FFmpeg Multi-Threaded Version

In the Google Summer of Code of the year 2008, the multi-threaded decoding branch FFmpeg-mt was created. This version can decode multiple frames in parallel. Recently, work has started to merge the multi-threaded branch to the main FFmpeg source code. In our research, we experiment the multi-threaded FFmpeg version on multiple cores. We further integrate our novel luma-chroma parallel decoding with entropy decoder pipelining into the source code. The new implementation, which decodes frames in parallel, requires doubling the number of cores for color parallel decoding. Synchronization is only required when a frame depends on a reference frame that is being decoded. In order to decode luma and chroma in parallel, we decode each frame using 2 cores. One core executes all the processes for frame decoding except chroma related tasks. The other core executes the motion compensation and intra-prediction of chroma color samples. The entropy decoder pipeline as illustrated in figure 4.6 is applied. Thus our approach is totally integrated in the H.264 parallel source code of FFmpeg [17].

As a result of our combined parallel implementation, we prove that our luma and chroma parallel technique can be applied to existing coarse grain and fine grain methods for parallelism. Coarse grain methods are mainly parallel decod-

ing for a group of frames, frames, and slices. Fine grain methods decode multiple macroblocks or blocks in parallel. Luma and chroma parallel decoding is applied when entropy decoding, inverse transform and de-quantization processes are completed. Depending on the H.264 decoder implementations, these processes can be performed for the whole slice before moving on to inter- or intra-prediction, or, they can be partially executed for each macroblock. FFmpeg uses the latter technique which is based on the macroblock level. Deblocking filter is executed when a row of macroblocks is completely decoded. We execute the decoding process of color components in parallel for each line of macroblocks. Thus one core decodes luma color samples for all macroblocks on the same row in parallel with another core decoding the chroma colors for the same macroblocks on the same row. This level of parallelism may be considered in the middle between coarse grain and fine grain methods.

Executing the H.264 decoder on multicore processors revealed an increase in performance over multi-threaded execution on one core. Adding color components parallelization with entropy decoder pipelining shows a relatively similar performance gain. The performance gain varies between 10% and 60% depending on the number of cores and the number of frames decoded in parallel. The speedup percentages in performance are similar to the numbers shown earlier for both high definition (HD) and low definition (CIF) resolutions. However, 8 cores shows the highest gain offset compared to 4 and 2 cores as displayed in figure 4.10. With 16 cores and above, the gain remains almost the same. Thus, we conclude that a saturation point is reached with 8 cores. The number of reference frames in a video sequence depends mainly on the encoder. In fact, using more cores does not always increase performance due to many factors like data communication, synchronization, frames dependencies in videos sequences, and many others.

4.5 Conclusion

H.264 is being widely adopted in multimedia applications on general-purpose and embedded systems. The high complexity imposed by the H.264 decoder requires enhancement in order to increase the efficiency and to lower power consumption. The proposed H.264 decoding of luma and chroma in parallel provides

high and realistic potentials for video decoding on dual and quad core processors. Execution time speedup of our parallel implementation of the H.264 decoder is around 18%. Moreover, the speedup reaches 32% with our proposed pipeline implementation with an energy saving of 24%.

In the following chapter, an advanced parallel algorithm is proposed to execute motion compensation on large number of parallel cores. The parallel approach is based on processing groups of independent macroblock rows in parallel. The proposed parallel algorithm shows a higher scalability than the color components approach described in this chapter. Thus, good speedup and energy saving are reached on multicore processors with more than 8 cores.

Chapter 5

H.264 Macroblocks Rows Parallel Decoding

5.1 Introduction

Many parallel implementations of the H.264 codec exist ranging from parallel decoding of macroblocks (*fine-grain* implementations) till parallel decoding of groups of pictures (*coarse-grain* implementations). A macroblock is a 16x16 square pixel component of an image in a video sequence. Moreover, a macroblock can also be divided into sub-blocks of smaller size. Macroblock parallel decoding is highly scalable since many independent macroblocks can be processed in parallel. However, dependencies and huge overheads are created as a result of memory communication and execution synchronization between macroblocks. On the other hand, parallel decoding of groups of pictures require large memory especially for high definition video sequences. In addition, they have a lower scalability than parallel macroblock decoding because of the small number of groups of frames that can be decoded in parallel. In our approach, we process rows of independent macroblocks in parallel using a new algorithm that eliminates dependencies between macroblocks and minimizes synchronization overhead. This level of parallel execution may be considered between the coarse-grain and the fine-grain parallel approaches, thus, offering a balance between large overheads and high scalability.

Our main contribution in this research is the design and implementation of

a new algorithm for processing macroblock rows of the H.264 decoder in parallel. In addition, a small footprint data dependency detection algorithm that isolates intra-prediction macroblocks (I-MBs) is implemented and executed on macroblocks of the same slice of a video frame. Experiments are conducted by executing our scalable parallel decoder on a Cuda Development Kit platform [43] with an ARM Cortex-A9 processor including 4 cores [6]. Execution time and energy consumption statistics are collected by running the application on the real-board platform. For HD and Full-HD resolutions, video sequences benchmarks reach their maximum throughput using 4 threads on 4 cores with a speedup of 3.3x for motion compensation and an overall speedup of 2.3x in terms of execution time and with an energy saving percentage of 63%. Moreover, the parallel algorithm has a very high theoretical speedup that is applicable on manycore and vector processors. [10, 11]

In our research, we enhance the H.264 decoder execution time knowing that our approach is also applicable to the H.264 encoder. We focus on improving the efficiency of the H.264 decoder using multicore processors. We decode groups of rows of macroblocks in parallel where each group is mapped to one core. Dependencies between macroblocks are avoided by decoding intra-prediction macroblocks sequentially at the end of the decoding stage. We prove that our approach has a better load balancing on multiple cores in addition to lower synchronization overhead than other approaches. With these advantages, we eventually reach higher theoretical and realistic speedups. We evaluate our approach on a real platform equipped with a quad core processor. Execution time and energy consumption statistics are gathered and analyzed.

The remainder of the chapter is organized as follows. In section 5.2, we briefly describe the H.264 decoding process and the macroblocks that form a slice of picture. In section 5.3, we describe our approach for parallelizing the motion compensation phase and the deblocking filter. In section 5.4, we present our real-board experimental results for execution time and energy consumption. We also discuss and analyze simulated executions and the theoretical scalability of our algorithm. Section 5.5 presents experiments and results of our parallel algorithm on graphics processors. Conclusion and future work are given in section 5.6.

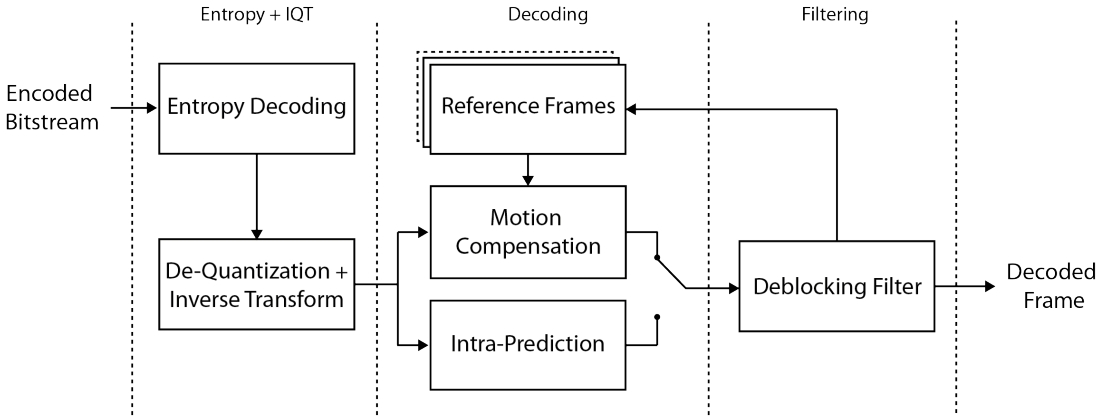

Figure 5.1: H.264 decoding process

5.2 Decoder Decomposition

In this section, we provide brief description of the H.264 video decoding process and the macroblocks in a frame. We also profile the different H.264 decoder stages.

5.2.1 Decoding Stages

The H.264 decoder can be divided into five main functional phases: *Entropy Decoder* (ED), *De-Quantization* and *Inverse Transform* (IQT), *Motion Compensation* (MC) and *Intra-Prediction* (IP), and *Deblocking Filter* (DF). The H.264 decoder stages are illustrated in Figure 5.1. The decoder process starts by entropy decoding the input bitstream. Then, de-quantization and inverse transformation are applied to the resulting data. Afterwards, in every slice of a frame, macroblocks are processed in raster mode. Each macroblock is intra- or inter-predicted (motion compensation) using the reference frames. The deblocking filter is applied at the end in order to make the edges between macroblocks smooth and invisible to human vision. Figure 5.2 illustrates the average execution percentage of each main phase using the baseline and the main profiles. De-quantization and inverse transform phase are grouped with the entropy decoder phase because they have a small footprint on overall execution. Both predictions phases, motion compensation and intra-prediction, are also merged together into one phase. Our parallel

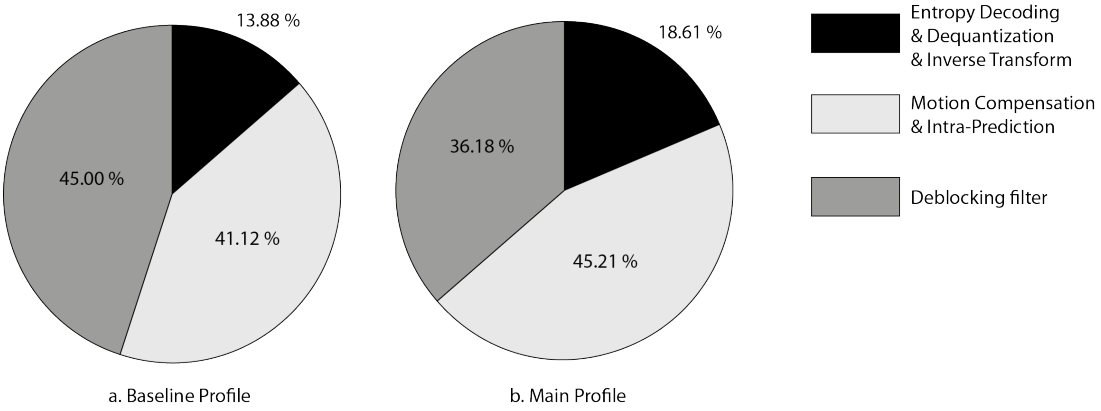

Figure 5.2: H.264 decoding stages workload percentages

algorithm is applied to the prediction phase that ranges from 41% till 45% of the overall decoding process. The entropy decoder with de-quantization and inverse transform is executed sequentially with a percentage ranging from 14% till 19%. We use the wavefront algorithm [72] for the deblocking filter of the H.264 decoder. The deblocking filter has a huge impact on the overall performance of the decoder that is 45% for the baseline profile and 36% for the main profile.

5.2.2 Macroblocks

Each slice of a picture frame is partitioned into square blocks of 16 x 16 pixels called *Macroblock* (MB). The number of horizontal and vertical macroblocks varies with the resolution of the frame. A macroblock can be divided into sub-blocks of 16 x 8, 8 x 8, 8 x 4, and 4 x 4 pixels. The encoder chooses the sub-blocks sizes depending on the amount of details (complexity) for specific parts of an image frame. An image, or part of an image, is considered complex when it contains objects with tiny details. For example, in a video of a flying bird with a consistent blue background, the encoder will divide the macroblocks in the region displaying the bird into sub-blocks smaller than 16x16 and the blue sky macroblocks will remain with the same of size of 16 x 16. The motion compensation stage uses a reference buffer in order to calculate the values of macroblocks in the current frame. The reference buffer contains a list of previously decoded frames. Macroblocks that are inter-predicted and motion compensated from previously

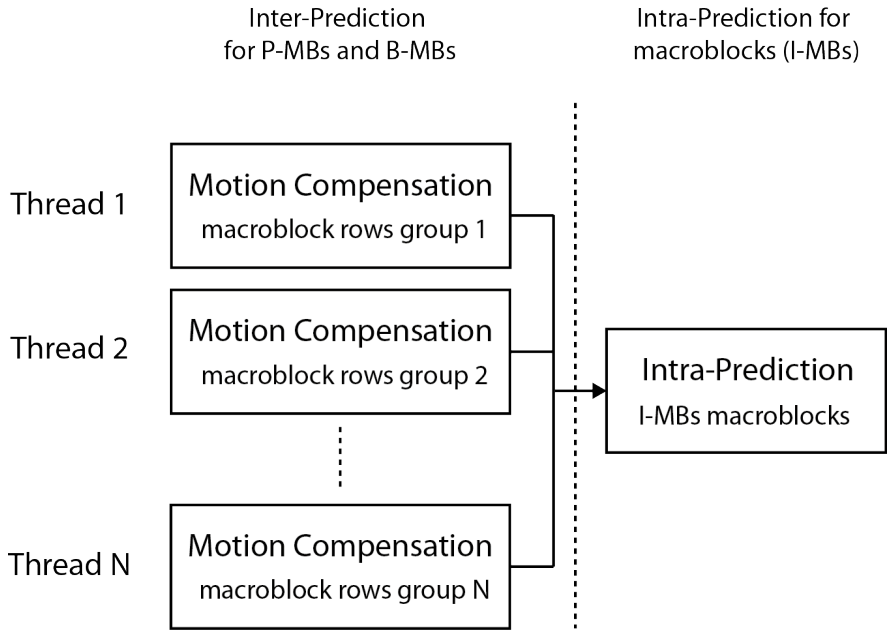

Figure 5.3: Decoding groups of macroblock rows in parallel using N threads

decoded frames are either of type P or B (P-MBs and B-MBs). P-MBs depend on macroblocks in one reference frame. B-MPs are calculated using macroblocks in two reference frames. Macroblocks that depend on other macroblocks in the current frame (called I-MBs) are intra-predicted. Finally, deblocking filtering is applied at the end of the decoding process in order to reduce the edging effect between macroblock borders.

In the following section, we describe in detail our parallel implementation of the H.264 decoder.

5.3 Parallel Implementation

In this Section, we elaborate on our parallel implementation of the H.264 video decoder. We explain how we apply parallelism to the motion compensation and the deblocking filter stages of the decoder. We also discuss macroblocks partitioning and their dependencies.

5.3.1 Parallel Motion Compensation

The H.264 reference implementation, JM [61], is an open source implementation used as a reference implementation for the H.264 standard. In our research, we modified the JM [61] source code of the H.264 decoder in order to decode rows of macroblocks in parallel using the PThread library in C programming language.

A thread is created for every group of macroblock rows. Each thread is mapped to one core. The number of thread is specified by the user or the application. If the number of threads is greater than the number of cores, then the scheduler will assign more than one thread for one core. As shown in figure 5.3, each thread handles the motion compensation stage for a group of macroblocks rows. All threads should complete their task before moving on to the next phase which is intra-predication for I-MBs.

The maximum numbers of parallel decoding blocks is equal to the number of macroblock rows. This level of parallel decoding of macroblock rows may be considered in between coarse-grain and fine-grain approaches. Coarse-grain approaches process multiple slices or frames in parallel. These high level methods, like [20] [28] [42] [57], need high memory usage in order to decode multiple frames in parallel because of the required size to store and to transfer data of several frames. Fine-grain approaches decode macroblocks or blocks inside a macroblock in parallel. These low-level methods, like [14] [67] [72], cause an enormous synchronization overhead affecting deeply the speedup for the reason of large number of macroblocks in every frame. The balance between both approaches is also reflected on synchronization overheads and data communication requirements.

Our approach is aimed to benefit from the balance between both advantages and disadvantages. Macroblock rows require less memory than a frame and more than one macroblock. In fact, our approach is scalable up to the macroblock level. Such granularity will create a huge overhead of parallelism on current multicore architectures. On the other hand, the number of macroblock rows is much less than the total number of macroblocks. For example, in HD resolution (1280 x 720), each frame has 3600 macroblocks, 80 horizontal MBs and 45 vertical MBs. Thus, the number of macroblocks rows is less by a factor of 80 than the total number of macroblocks. As a result, the overhead for synchronization and com-

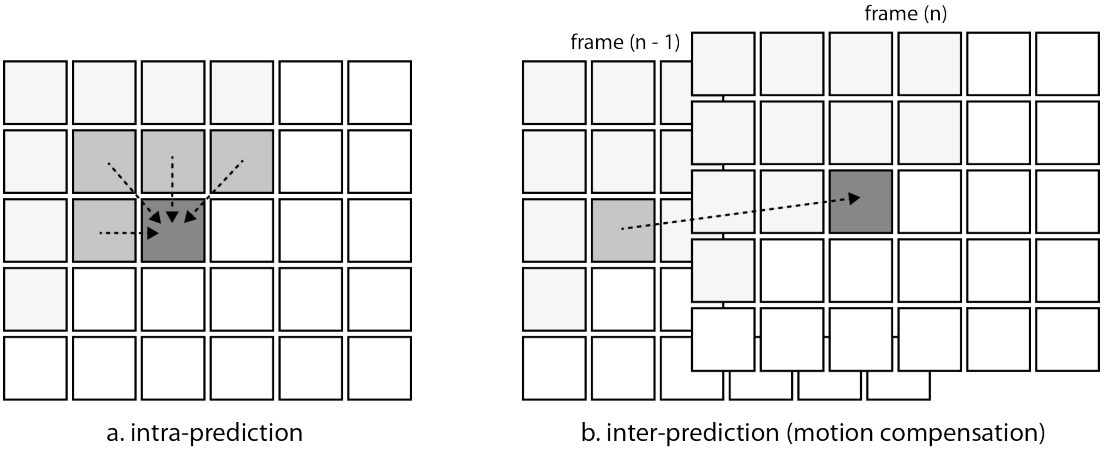

Figure 5.4: Dependencies between macroblocks

munications between cores is also reduced by a factor of 80.

5.3.2 Macroblocks Dependencies

In H.264, there are 4 types of macroblocks: I, P, B, and SKIP. Figure 5.4 illustrates the dependencies between macroblocks of types I and P. I-MBs depend on other macroblocks in the same slice of a frame as shown in figure 5.4-a where the macroblock pointed at by the arrows may be dependent on one or more macroblocks. P-MBs depend on macroblocks from previously decoded frames as shown in figure 5.4-b where the origin of the arrow is a macroblock in a previously decoded frame. Motion vectors info is required for P-MBs in order to reconstruct the coded macroblocks. B-MBs depend on past and future reference frames. They are available in B-Frames and they can have one or two motion vectors. The SKIP macroblock data remains the same when it is compared to another macroblock in a previously decoded frame. So the motion vector differences are zero, and therefore, the prediction macroblock is simply copied as the reconstructed macroblock.

In a frame, all macroblocks can be processed in parallel except I-MBs because they depend on macroblocks which are being decoded in the same slice. So a dependency identification procedure is needed to satisfy intra-prediction dependencies. In order to overcome this constraint, we start by decoding all macroblocks

of type P, B, and SKIP in parallel. During this step, we skip all I-MBs and we save a reference to the skipped macroblocks for future processing. When this stage is completed, the remaining I-MBs macroblocks in the current slice are decoded sequentially as illustrated in figure 5.3. Among the remaining I-MBs, independent macroblocks can be processed in parallel as they depend on macroblocks in the same slice that are already processed. For simplicity and because of their small number in each frame (except I-Frames), we process I-MBs sequentially in our algorithm.

With this ordering mechanism, dependencies between macroblocks in the same slice are satisfied. Table 5.1 lists the percentages of I-MBs, P-MBs and SKIP-MBs in the video sequences that we use in our experiments. The average number of I-MBs for all video sequences is about 2%. I-MBs also exist in P-frames and B-Frames. The number of I-MBs in a P-Frame or a B-Frame depends on objects with high details and on objects rate of movements in the video sequences. P-Frames and B-Frames are mostly composed of P-MBs and SKIP-MBs with a small number of I-MBs. So the small number of I-MBs in P-Frames and B-Frames does not significantly affect the overall speedup for the parallel decoding of macroblocks.

5.3.3 IDR Frame Frequency

An encoded video always starts with an I-Frame (IDR) which is composed completely of I-MBs. This type of frames is available typically every one second in a video sequences in order to overcome communication errors and their propagation when data is lost during transmission. A high number of IDR frames significantly impacts the parallel efficiency and the scalability of our algorithm. The interval between IDR frames is typically equal to the frame rate (as in the default settings of the x264 encoder [46]). For example, an HD video sequence with a frame rate of 60 frames per second (fps) will have an IDR frame every 60 frames (equivalent to 1 second). We can increase or decrease the frequency of IDR frames in the encoder configuration. However, a high frequency of IDR frames, for example one I-frame every 10 frames, decreases the compression efficiency of the encoder and the visual results will not be noticeable by the human vision. The recommended configuration for the IDR period in the x264 [46] and the JM

Table 5.1: Percentages of different types of macroblocks per video sequence

Name	Resol.	Fr.	I	P	SKIP
bus	352x288	150	1.70	79.20	19.10
foreman	352x288	300	1.80	70.95	27.25
waterfall	352x288	260	0.25	70.05	29.70
johnny	854x480	600	0.10	22.35	77.55
basketball	854x480	500	3.40	62.25	34.35
cactus	854x480	500	1.50	42.30	56.20
johnny	1280x720	600	0.15	22.50	77.35
basketball	1280x720	500	3.95	58.50	37.55
cactus	1280x720	500	1.90	42.50	55.60
basketball	1920x1088	500	4.95	55.30	39.75
cactus	1920x1088	500	3.15	44.05	52.80
terrace	1920x1088	600	0.80	56.50	42.70
	Average		**1.97**	**52.20**	**45.83**

[61] H.264 encoders is set to an adaptive decision which basically inserts an IDR whenever a scene changes. We use this feature in our experiments in order to encode the video benchmarks. The numbers of I, P, and B frames are listed in table 5.2 on page 87. The IDR period for low frame rates (25 fps) is around 150 (6 seconds) and for high frame rates (50-60 fps) is 200-250 (3-5 seconds).

5.3.4 Macroblock Dependency Check Algorithm

The macroblock dependency check algorithm is straightforward and simple to implement. Figure 5.5 shows a simple illustration of the algorithm. Given a list containing all the macroblocks in a slice, a loop that iterates over all macroblocks flags all intra-prediction macroblocks (I-MBs) and assigns each remaining macroblock to a group specific for an available core. Then, these groups of macroblocks are decoded in parallel. When all macroblock groups are processed, a loop iterates over all I-MBs that were flagged initially. All the macroblocks in the I-MBs list are decoded sequentially. I-MBs can be processed in parallel if they are not neighbors, meaning they do not have any dependencies between them. The number of I-MBs is not significant in P-Frames and B-Frames as shown in table 5.1. If we assign one macroblock to a different core, the workload is not very important and synchronization overhead will also be added. So we just execute them sequentially

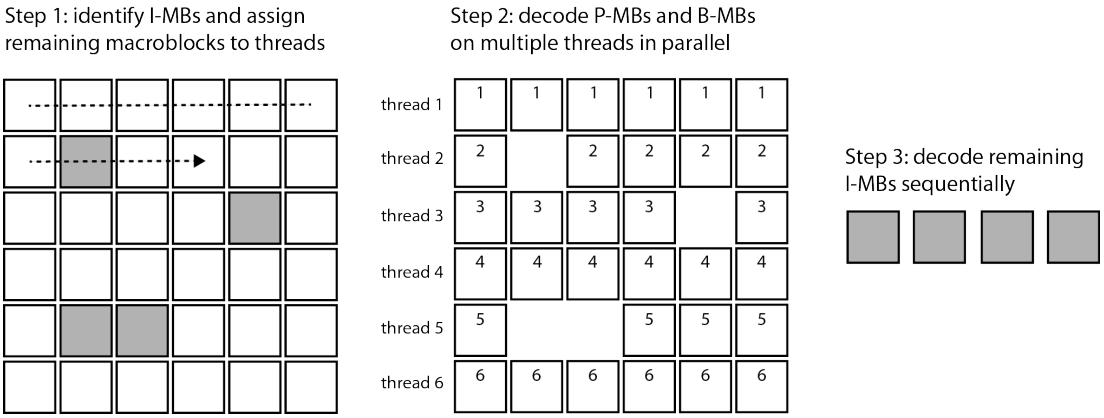

Figure 5.5: Macroblock row-based parallel algorithm. In step 1, all the macroblocks are scanned and I-MBs are identified. In step 2, rows of P-MBs and B-MBs are processed simultaneously. Finally in step 3, the remaining I-MBs are decoded sequentially.

in our experiments for the reasons of simplicity and less communication overhead. With the previously mentioned steps, inter-prediction and intra-prediction stages are completed. The output of this stage complies fully with the H.264 standard [26], which means that the output is exactly the same when sequential execution is performed. Decoded macroblocks are then submitted to the deblocking filter in order to make these edges between macroblocks smooth and nearly invisible.

The asymptotic worst-case complexity of the proposed parallel algorithm remains almost the same as the sequential algorithm. All the macroblocks are processed one time, which is similar to sequential execution. An additional iteration with a constant operation overhead is added before parallel execution. During this process, I-MBs are identified and their pointers are added to a list for further processing in a following step. The runtime cost of this additional loop is linear and is considered negligible among the total complexity of the algorithm. After parallel decoding of the independent macroblocks, the remaining macroblocks which are in the previously described list are processed sequentially and in order. Thus, the worst-case complexity of the algorithm in comparison with the original sequential algorithm remains the same with or without the gain of parallel computing.

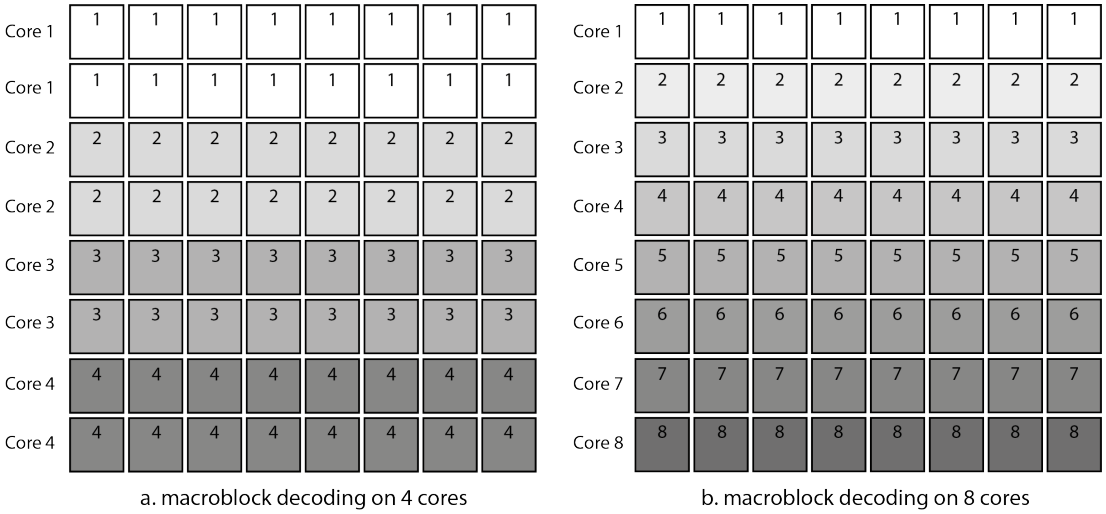

Figure 5.6: Parallel decoding of macroblocks mapped to (a) 4 cores and (b) 8 cores

5.3.5 Macroblocks Partitioning

In the parallel decoding algorithm described above, groups of macroblocks are decoded in parallel. In this part, we explain why we chose groups of macroblocks to be decoded in parallel. As explained above, while iterating over macroblocks in a frame slice, we skip intra-prediction macroblocks (I-MBs) and we decode inter-prediction macroblocks (P-MBs and SKIP-MBs) in parallel on multiple cores. Depending on the number of available cores, we group rows of macroblocks in order to be decoded in parallel. The slice is divided by the number of cores horizontally.

Seitner et al. [55] compare 6 parallel representations in terms of stall time and core usage. Among the presented data partitioning approaches, our partition is similar to the slice-parallel splitting approach that is described in [55]. As shown by the authors, this approach has significant stall time overhead which is caused by synchronization procedures in order to satisfy macroblock dependencies. However, with our approach for satisfying dependencies between macroblocks, the stall time overhead does not apply. We chose this method because of data locality and also due to minimal data transfer overhead. For example, in order to execute a slice of 80 rows of macroblocks on 4 cores processor, each core decode a

chunk of 20 rows of macroblocks. Using this partition method, one frame requires four transfers in order to send data to each core's local memory. This number of transfers is minimal because it is equal to the number of available cores. Communication overhead between caches of different cores is required when I-MBs depend on other macroblocks that are processed by another core. In Figure 5.6, we show an example of a frame of size 8 x 8 MB (64 x 64 pixels) mapped on 4 cores in 5.6-a and on 8 cores in 5.6-b. The numbers inside the squares are the numbers of corresponding cores. Macroblocks in Figure 5.6 are assumed to be all P-MBs or B-MBs. I-MBs are not displayed for illustration purposes.

In a sequential implementation, macroblocks are processed in raster scan mode, starting from top to bottom rows and for each row from left to right macroblock. All independent macroblocks in a slice can be processed at the same time. However, the level of parallelism is limited by the number of available cores. In our parallel implementation, we choose to group macroblocks in rows because it offers a good load balance on different cores. In addition, this level of parallelism has a low synchronization overhead between cores and it can be considered simple to implement and to manage. Moreover, decoding independent macrobocks vertically or diagonally did not show any significant difference with horizontal decoding because all these macroblocks depend on previously decoded macroblocks. Further studies will be performed in order to group macroblocks based on their dependencies to previously decoded macroblocks. In this chapter, we limit our study to the row-based algorithm that is tested on an embedded multicore processor.

5.3.6 Scalability of Parallel Motion Compensation

In our approach, the highest scalability level is the maximum number of independents macroblocks in a frame slice. Once the dependency detection algorithm isolates the I-MBs, all remaining macroblocks can be processed at the same time. However, the level of parallelism is limited by the available cores in a multiprocessor chip. The optimal speedup will occur when all groups of macroblocks are assigned to available parallel cores. This will eliminate the context switching overhead which affects the performance in general. For manycore processors, an

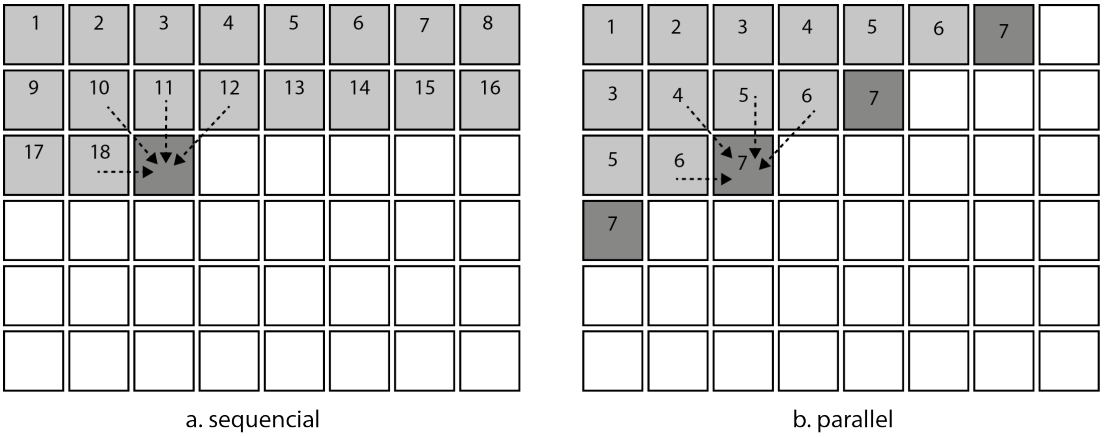

Figure 5.7: Sequential and parallel deblocking filter of macroblocks in the H.264 decoder

important limitation that remains unsolved is the huge data communication overhead between cores. For vector processors or general-purpose graphical processing units (GPGPUs) which offer a very high level of parallelism, great potentials exist that may also benefit from the high scalability of our approach.

5.3.7 Parallel Deblocking Filter

The deblocking filter, last stage of the H.264 decoder, makes the edges between macroblocks smoother. This process decreases the artifacts that appear when a slice is partitioned into macroblocks. This final stage of the decoder that consists of 41% to 45% of the total decoding time as illustrated in figure 5.2 on page 77 is also modified to be executed in parallel on different cores. However, dependencies between macroblocks in this stage are different than the dependencies of motion compensation and intra-prediction. During the deblocking filter stage, each macroblock requires that the top and the left macroblocks are already processed. Figure 5.7 illustrates the sequential (a) and the parallel (b) filtering modes that are applied on macroblocks in a slice. Both scanning modes satisfy the dependencies requirements of the deblocking filter stage. In figure 5.7-a, one marcoblock is filtered at a time. In figure 5.7-b, macroblocks colored in dark gray are processed on different cores in parallel. This method, also known as wavefront scheduling, is considered as a commonly used approach for processing

Table 5.2: Video sequences resolution and frames types info.

Name	Resol.	fps	I	P	B	Total
bus	352x288	25	1	75	74	150
foreman	352x288	25	2	161	137	300
waterfall	352x288	25	2	116	142	260
johnny	854x480	60	3	151	446	600
basketball	854x480	50	2	250	248	500
cactus	854x480	50	2	249	249	500
johnny	1280x720	60	3	151	446	600
basketball	1280x720	50	2	247	251	500
cactus	1280x720	50	2	244	254	500
basketball	1920x1088	50	2	236	262	500
cactus	1920x1088	50	2	181	317	500
terrace	1920x1088	60	3	231	367	600

independent macroblocks. It can be applied at the intra-prediction, the motion compensation and the deblocking filter stages as proposed and explained by Zhao et al. [72].

We implement the wavefront parallel method for the deblocking filter stage only. This method satisfies the dependencies requirements of the deblocking filter process as illustrated in Figure 5.7-b. We implement this parallel processing approach in order to complement our proposed parallel motion compensation algorithm. Both stages process independent macroblocks in parallel. In the following section, experimental results will be provided for the complete parallel implementation of the motion compensation and the deblocking filter stages.

5.4 Experimental Results on Multicore Systems

In this section, we evaluate our H.264 parallel implementation on a multicore embedded processor. We describe the configuration environment for the real-time execution and the tools that were used to collect all execution information. We gather real-board execution time and energy consumption statistics. We also compare our results with similar literature for parallel H.264 implementations. Moreover, we experiment and we collect runtime statistics for our parallel implementation using a multicore simulator and a graphical processor.

5.4.1 Parallel Execution

Parallel execution is considered as a major potential solution for complex applications where sequential execution bounds the performance of these applications. Most processors that are currently available in the market have multiple cores. Applications with high computational complexity may benefit from potential speedup from multiple cores when data or functional parallelism is applicable. Even optimized implementations can still take advantage from parallel techniques. In our research, we choose the H.264/AVC video decoder as our multimedia application benchmark for which we provide an innovative parallel approach. We further gather execution statistics and compare results with other relatively similar implementations. In our H.264 parallel approach, the motion compensation (MC) stage for each row of inter-prediction macroblocks (P-MB) is executed in parallel on different cores. We experiment our parallel implementation using video sequences with CIF (352x288), WVGA (854x280), HD (1280x720), and FHD (1920x1080) resolutions on an embedded multicore processor. Macroblock dependencies in the same picture slice are avoided by decoding intra-prediction macroblocks (I-MBs) when all other macroblocks of the same slice are already decoded. Overheads emerged as a result of shared memory communications and synchronization between cores. We collect execution time and energy consumption statistics using experiments on a real board with an embedded multicore processor. A virtual threshold for the speedup to the number of cores ratio is identified when large numbers of threads are used. Parallel execution is also tested on a multicore and a graphical simulator [66].

5.4.2 Environment and Configurations

Our H.264 parallel implementation described in section 5.3 is executed and tested on a Cuda Development Kit platform [43] with an ARM Cortex-A9 processor with 4 cores [6]. The processor has a memory size of 2 GB and an L2 cache size of 1MB. L1 instruction and data caches both have the size of 32 KB. The maximum frequency is 1.3 GHz when 4 cores are used. This high-end and low-power processor is currently available in many portable devices like smartphones, tablets, notebooks, etc. We execute our parallel H.264 decoder using 2, 4, 6, 8, 12,

and 16 threads. Each thread is mapped automatically by the operating system (Ubuntu in our case) to a different core. When the number of threads is more than 4, context switching is required to run all threads that are created by the application. We gather statistics using 4 different resolutions: CIF, WGVA, HD, and FHD. With each resolution, we use 3 different video sequences with different image complexities in terms of movement speed and number of objects. Table 5.2 on page 87 lists all the video benchmarks that were used in our experiments. The information in table 5.2 include the resolution, the rate of frames per second, the number of I-Frames, the number of P-Frames, the number of B-Frames, and the total number of frames. Real-time execution for all the above video sequences is performed. Execution time is simply calculated by the application and the operating system. Energy statistics are collected by a power measuring instrument, the Agilent LXI digitizer [64]. The digitizer accurately measures the static and dynamic power consumption across the resistors place. The Agilent Technologies L4532A [64] is a high-resolution, standalone LXI digitizer. It offers 2 channels of simultaneous sampling at up to 20 mega samples per second (MSa/s), with 16 bits of resolution. Inputs are isolated and can measure up to 250 volts to handle the most demanding applications. Time and energy results are illustrated and analyzed in the following subsections.

5.4.3 Results for Parallel Motion Compensation

Experiments are preformed on the videos sequences listed in table 5.2. The number of parallel rows of macroblocks increases with the video resolution. Thus, high resolutions scale better than low resolutions with the number of core due to higher number of macroblocks in each frame. Experiments are conducted using 2, 4, 6, 8, 12, and 16 threads on an ARM Cortex-A9 with 4 cores [6]. Figure 5.8 shows the average speedup of the motion compensation stage for every resolution for different number of threads. For the CIF resolution, the maximum speedup of 1.8 is attained using 4 threads. The speedup decreases as the number of threads increases due to large data communication overhead. HD and FHD video sequences have a speedup higher than 3.3 with 4 threads where each thread is mapped to different core. The best speedup to the number of threads ratio is

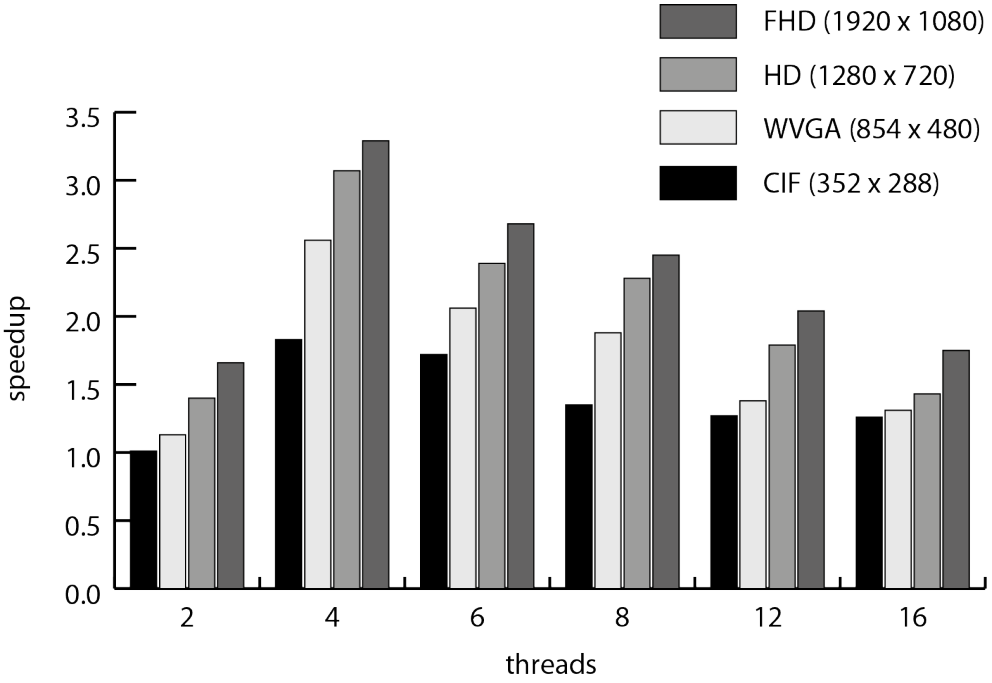

Figure 5.8: Speedup of H.264 parallel execution of the motion compensation stage.

when 4 threads are used. The ratio of speedup to number of threads for high definition resolutions is around 0.8 when 4 threads are used. Doubling the number of threads drops the ratio to 0.6 which cannot be considered as efficient as expected when running a parallel application on a multicore processor. Using a number of threads that is more than the number of cores causes the scheduler to assign more than one thread for one core. Hence, the resulted context switching does not increase the efficiency of the application as shown in our results.

Results for high resolutions in general have better speedups. This is mainly due to greater workload for each core than smaller resolutions. A larger workload reduces the impact of synchronization and data transfer between cores. One of the reasons is less dependencies between macroblocks being processed on different cores. Another reason is the data transfer overhead which is required for sending data to different cores. Synchronization also adds an overhead which is independent of the video resolution. Thus, speedup will be much more efficient for higher resolutions.

Table 5.3: Comparison of macroblock parallelism scalability with Dynamic 3D-Wave in [38].

Resolution	Total MBs	3D-MBs	Par-MBs	Diff.
SD (720 x 576)	1620	1288	1592	+23.6%
HD (1280 x 720)	3600	2886	3528	+22.3%
FHD (920 x 1088)	8160	5819	7917	+36.1%

5.4.4 Comparison with Related Work

For the 2D-Wave approach described in [39], the speedup using 4 cores is 2.6 and the highest speedup is around 9.5 using 24 cores. These results assume minimal data communications and dependencies between cores. Our results have a better ratio between the speedup and the number of cores; however, we can only compare the speedup up to 4 cores. In addition, our approach has a higher theoretical speedup as the number of independent macroblocks that can be processed at the same time is higher. When processing macroblocks simultaneously, workload on different cores is almost equal. On the other side, when applying the wavefront approach in [39] and [72], the number of independent macroblocks reaches its maximum only when almost half of all macroblocks of the current slice are already decoded. Furthermore, the experimental environment is not the same. We are testing our parallel implementation on a real platform, on the other side, most results in other researches like [55] and [39], use simulators. In following sections, we will show simulated results for the overall execution of the parallel H.264 decoder.

Exact comparisons with related work cannot be accurate for several reasons like decoder implementation, processor configurations, video resolutions, and data communication between parallel cores. However, a comparison of the macroblock scalability between our approach and the Dynamic 3D-Wave [38] is shown in table 5.3. The 3D-Wave paper [38] performed a detailed analysis of the parallel scalability of macroblocks. We intend to compare the maximum number of macroblocks that can be processed in parallel between our approach and the Dynamic 3D-Wave approach. Three video resolutions are being compared. SD resolution (720 x 576) is compared to WVGA (854 x 480) because it has the same total number of macroblocks per frame. The remaining resolutions being compared are

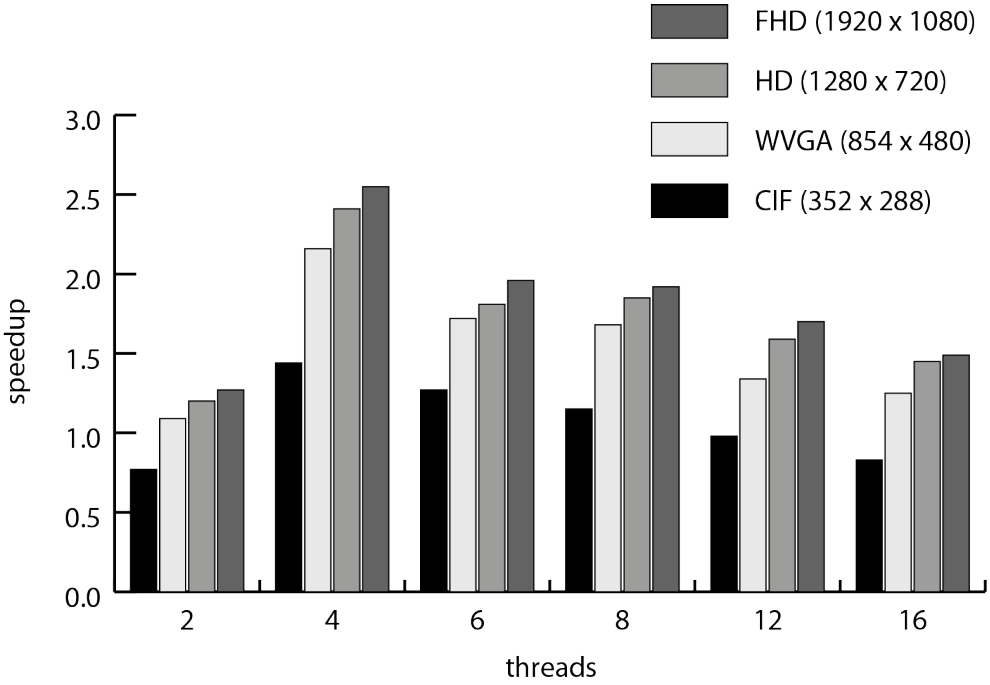

Figure 5.9: Speedup of H.264 parallel execution of the deblocking filter.

HD and FHD. The second column lists the total number of macroblocks per frame for each video resolution. The third column displays the average of the maximum number of parallel macroblocks of the four video benchmarks listed in table 4 in [38]. The fourth column shows the total number of macroblocks per frame that can be processed in parallel using our parallel motion compensation algorithm. Finally, the last column is the difference of the level of parallel macroblock scalability between both approaches. A difference of 22% till 36% is calculated in favor of our approach. In addition, all parallel macroblocks using our approach are in the same frame. Whereas, in the 3D-Wave approach [38], parallel macroblocks are from several frames that are being processed concurrently. We note that the numbers in table 5.3 are maximum values which, in practice, cannot be effectively executed in parallel using today's manycore systems. We choose the group parallel macroblocks in groups of rows depending on the number of available cores in a multicore architecture.

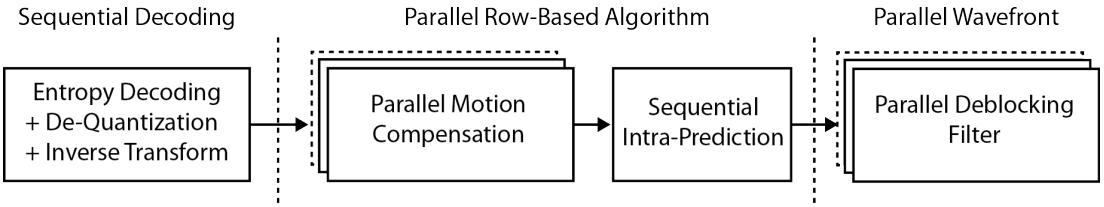

Figure 5.10: Overall H.264 decoding stages with parallel algorithms.

5.4.5 Results for Parallel Deblocking Filter

Similarly to the motion compensation experiments, we gather statistics results of our parallel implementation of the deblocking filter using the wavefront algorithm. For the deblocking filter, the wavefront algorithm is the best known parallel algorithm that satisfies the dependency constraints of this stage. The same videos sequences that are listed in table 5.2 are used. As described previously, the wavefront algorithm reaches the highest number of independent macroblocks that can be filtered in parallel when the diagonal divides the slice into almost two equal partitions. Parallel deblocking achieves a speedup of 1.44 using 4 threads for CIF resolution and a speedup of 2.6 using 4 threads for Full-HD resolution. Figure 5.9 displays the average speedup results for different resolutions and different number of threads. As mentioned earlier, the scalability of the wavefront algorithm is not as high as our parallel decoding algorithm for motion compensation and intra-prediction stages. In addition, the workload for every core using the wavefront algorithm is only one macroblock, whereas, the workload of the motion compensation algorithm is composed of many macroblocks depending on the number if available cores. A smaller workload also adds more synchronization overhead. Thus, the speedups of the parallel deblocking filter are lower than the motion compensation speedups displayed in the previous subsection.

5.4.6 Results for Overall Execution

Our main goal is to optimize all the stages the H.264 decoder. We apply parallel techniques for the motion compensation and the deblocking filter stages. On the other hand, the entropy decoder stage is inherently sequential. Thus, parallel techniques for the entropy decoder are very hard to apply or sometimes

Table 5.4: Overall speedup of video sequences executed with multiple threads on multicore processors.

Seq/Threads	2	4	6	8	12	16
CIF-Bus	0.94	1.40	1.34	1.23	1.12	1.09
CIF-Foreman	0.85	1.30	1.35	1.10	1.13	1.01
CIF-Waterfall	0.82	1.58	1.40	1.27	1.12	1.08
WVGA-John.	1.06	2.15	1.74	1.65	1.26	1.05
WVGA-Bask.	1.13	1.92	1.68	1.62	1.35	1.14
WVGA-Cact.	1.07	1.81	1.62	1.56	1.29	1.10
HD-Johnny	1.27	2.42	1.91	1.93	1.60	1.36
HD-Basket	1.28	2.14	1.81	1.81	1.58	1.39
HD-Cactus	1.26	2.09	1.78	1.78	1.55	1.34
FHD-Basket	1.40	2.26	1.93	1.89	1.68	1.52
FHD-Cactus	1.42	2.28	1.93	1.86	1.68	1.51
FHD-Terrace	1.44	2.33	1.97	1.93	1.72	1.53

impossible due to its specification requirements. Figure 5.10 depicts the stages with parallel algorithms of the H.264 decoder. We collect execution time and energy consumption statistics for the proposed H.264 parallel implementation. The fractions of the different stages vary among different video sequences. As a result, the overall performance is considered as a weighted average of all speedups based on the average percentage of each phase.

Figure 5.11 illustrates the overall speedups attained for the complete execution of the decoder with the described optimization techniques. The total speedups of 1.4, 2.0, 2.2, and 2.3 are reached using 4 threads on 4 cores for the resolutions CIF, WVGA, HD, and FHD respectively. The detailed results for every video sequence are listed in table 5.4. The sequential execution of the entropy decoding stage which is about 14-19% of the overall decoding scales down significantly the overall speedup. This stage may be enhanced by implementing a hardware version of the entropy decoder. FHD resolutions have the highest speedup because of their large frame sizes. All maximum speedups are attained using 4 threads on 4 cores. This is mainly due to the absence of context switching where each thread is mapped to one core. Using more than 4 threads will require the operating

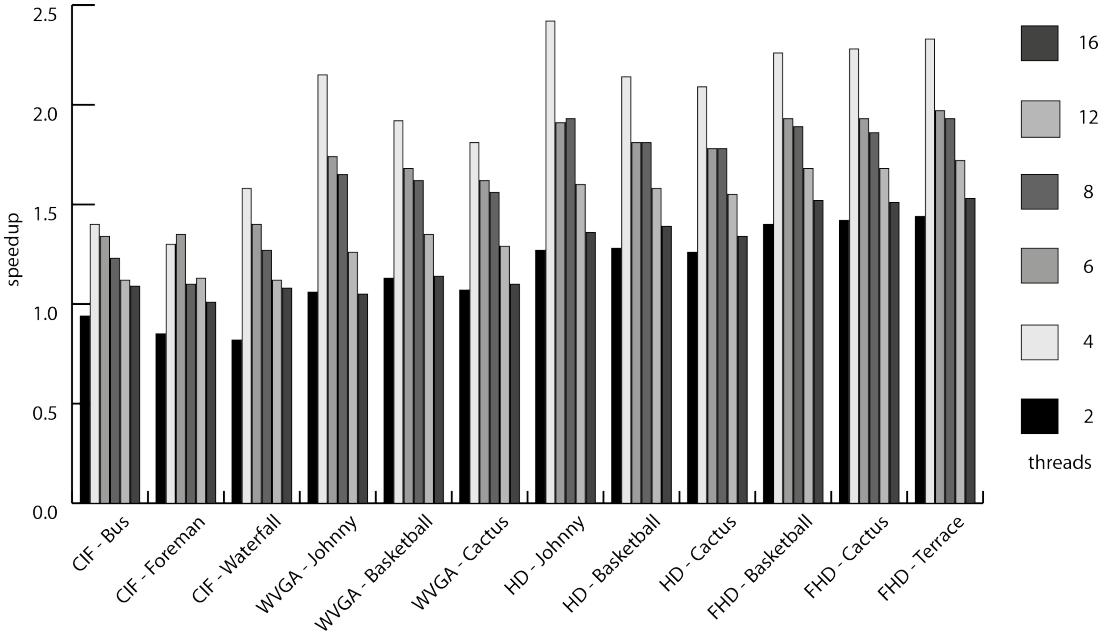

Figure 5.11: Total speedup for the complete decoding process on multicore processor.

system to assign more than one thread to a core causing context switching and, as a result, more overhead and stall time will be added to the overall execution. Only CIF video sequences have speedups less than 2 when 4 threads are mapped onto 4 cores. The ratio of speedup to the number of cores is therefore around 0.6. This leads us to conclude that high resolution benefit more from multicore processors than lower resolutions. So Full-HD resolutions have the best speedup with higher number of cores. 4K resolution appeared recently in high-end TVs and in movies theaters.

Energy measurements for the complete execution are displayed in Figure 5.12. The best energy saving results corresponds to the FHD resolutions using 4 threads which attain 63%. These results are also measured for the complete execution of the optimized decoder. For 12 and 16 threads, energy consumption will increase compared to sequential execution. Thus, we conclude that energy saving does not scale linearly with the number of threads or cores.

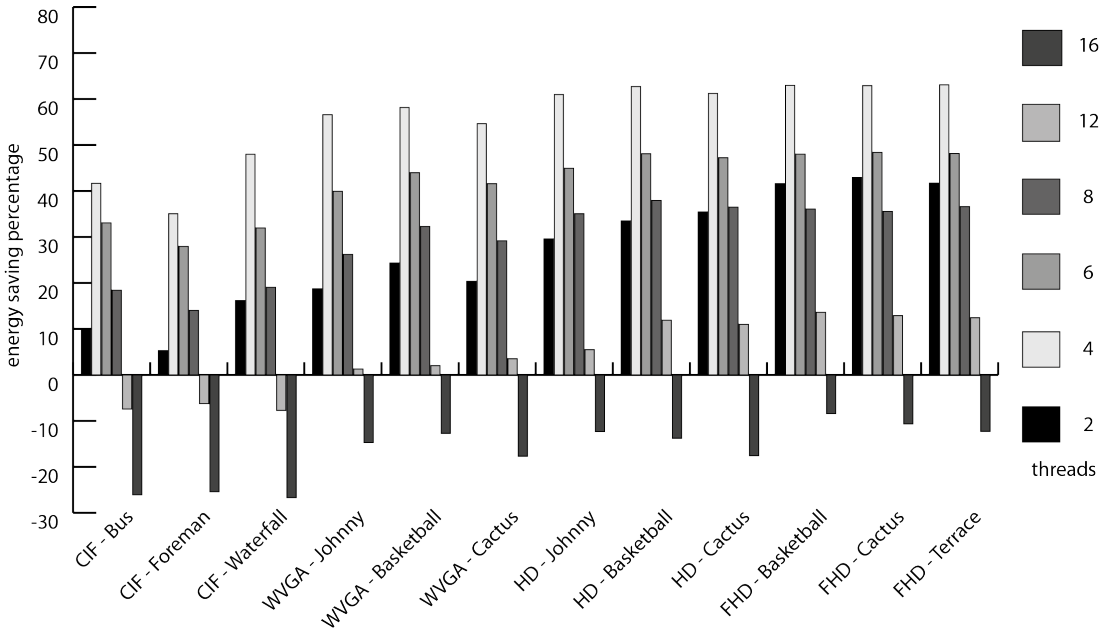

Figure 5.12: Total energy saving for the complete decoding process on multicore processor.

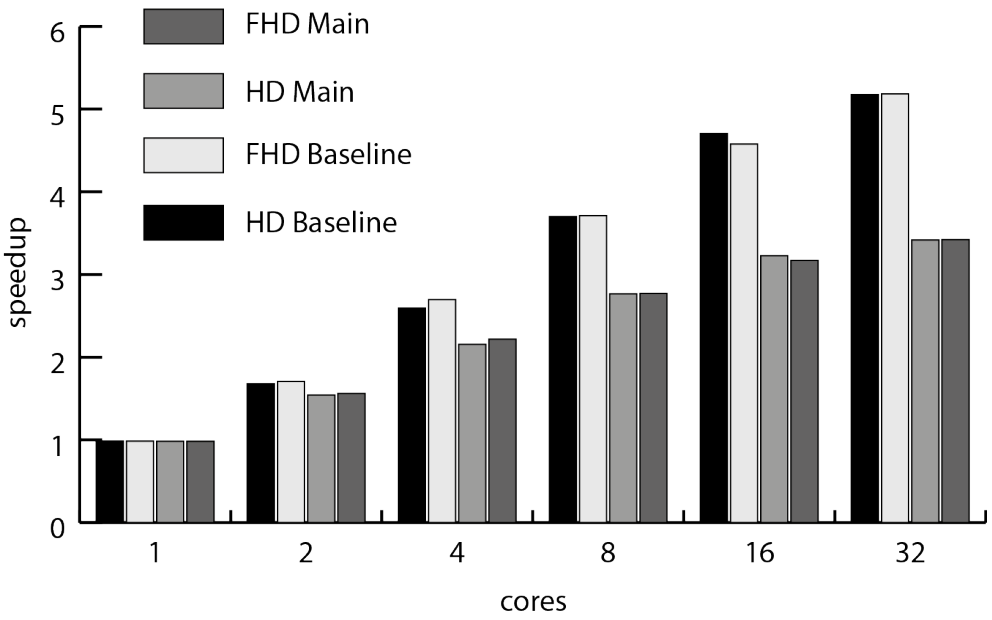

Figure 5.13: Speedup of H.264 parallel execution using the Multi2Sim simulator.

5.4.7 Simulated Execution

As a complementary step to experiment our parallel H.264 algorithm, we execute our implementation on the multicore simulator Multi2Sim [66]. Figure 5.13 shows the speedup of our parallel H.264 implementation on 2, 4, 8, 16, and 32 cores. HD and FHD resolutions are used with the Baseline and the Main profiles. These results display the average of the three video sequences listed in table 5.2. On 2 cores, the speedup for Baseline profile is 1.7 and 1.5 for Main profile. The speedup increases with the number of cores; however, this increase is not linear. Using 32 cores, the speedup reaches 5.2 for the Baseline profile and 3.2 for the Main profile. The difference between both profiles becomes more significant as the number of cores increases. The time needed for motion compensation and deblocking filtering in the Baseline profile is higher than the Main profile. The entropy decoding execution time is less for the Baseline profile compared to the Main profile. This is mainly due to the CABAC algorithm for the entropy decoder which is used in the Main profile. CABAC has a better compression at the expense more complexity. Thus, our parallel method is better exploited with the Baseline profile where the entropy decoder, which is executed sequentially, has less impact on the overall speedup. The parallel scalability of our H.264 decoder is significantly affected by data communication between cores. The results shown in figure 5.13 for 8 cores and more are inefficient compared to theoretical speedup. The ratio of speedup to the number of cores is 0.85 on 2 cores and 0.65 on 4 cores. For higher numbers of cores, the ratio is below 0.5 which is considered inefficient and unworthy of parallel execution. Manycore processors with 16 or more cores should have special memory architecture than dual and quad cores processors. Thus, parallel algorithms, like our H.264 parallel decoder, should be adapted to benefit from manycore processors and to minimize data communication overhead imposed by a large number of parallel cores.

5.4.8 Theoretical Speedup

Figure 5.14 shows the theoretical speedup that can be reached for the overall execution of the H.264 parallel decoder. The differences with the simulated execution results displayed in figure 5.13 are relatively small up to 8 cores. For

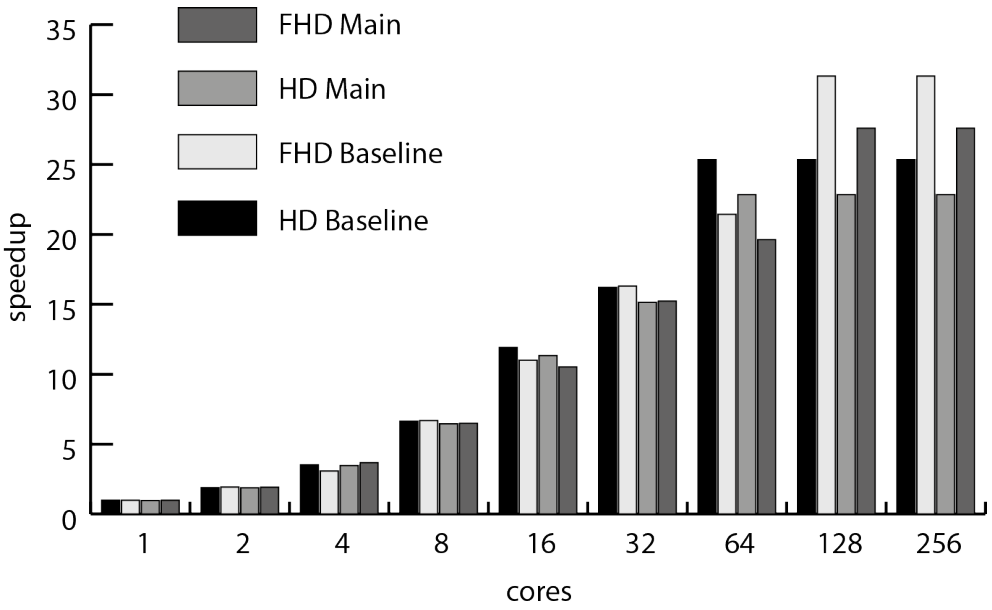

Figure 5.14: Theoretical speedup of H.264 parallel execution.

16 cores, the speedup of our parallel H.264 algorithm should be around 12. The speedups keep increasing until 64 cores for HD resolutions and 128 cores for FHD resolutions. This threshold appears when the number of cores becomes more than the number of macroblock rows. However, using our algorithm for parallel motion compensation, the granularity can become smaller so that we can benefit from additional cores. If the number of cores is close to the number of parallel macroblocks that are listed in table 5.3, then the speedup would become much higher. In real manycore architecture, this speedup comes with a huge memory communication overhead that affects the speedup dramatically. New parallel processing architectures should be used for such high levels of parallelism. This issue is still a major bottleneck in the computing industry. In our research, we also aim to explore and to experiment new parallel architectures in order to show to the full benefits of parallel computing.

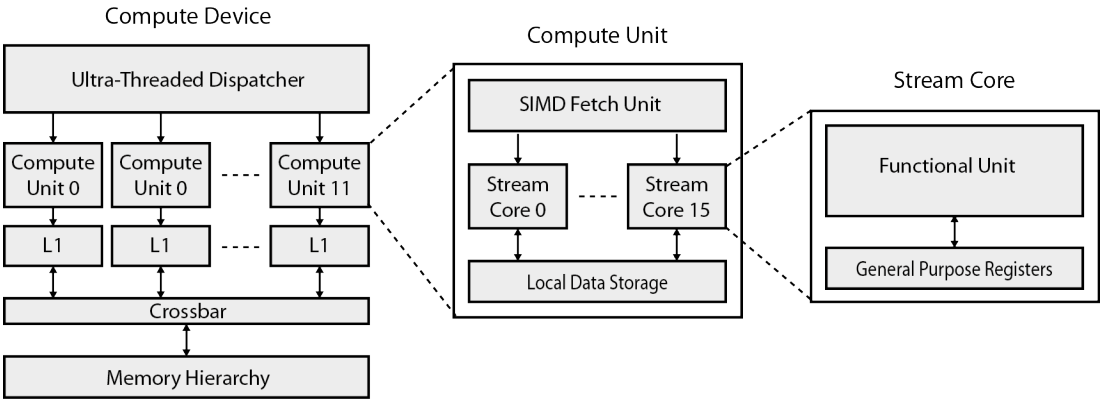

Figure 5.15: Architecture of the graphical processor AMD Radeon HD 6850.

5.5 Parallel Execution on Graphics Processor

In this section, we experiment our parallel algorithm for motion compensation on a general-purpose graphical processor (GPGPU). Brief overviews about GPUs and OpenCL are provided. Experimental results are then listed and analyzed.

5.5.1 General-Purpose Graphical Processing Unit

Originally, a GPU is a specialized hardware unit that is limited for rendering graphics on screen. Modern GPUs are massively parallel processors that are special types of stream computing processors or SIMDs that are explained in chapter 2. These parallel processors can compute large number of values concurrently. Early generations of GPUs had a fixed pipeline with limited programming capabilities. Nowadays, modern GPUs enable general purpose programming through C-like languages such as nVidia CUDA [44] and OpenCL by the Khronos Group [30]. Hence, many high computational algorithms that are not related to graphics processed can now be executed on GPUS which is known as general-purpose computing on graphics processing units (GPGPU). GPGPUs have much less control logic, freeing up more die space for arithmetic logic units (ALUs). This low complexity of the architecture gives GPUs more calculation capabilities at the cost of programming complexity. So in order to reach a good performance, the programmer must explicitly design the application for the target GPU.

Figure 5.15 illustrates the architecture of the GPGPU AMD Radeon HD 6850.

Table 5.5: Specifications for AMD Radeon HD 6850 (Barts PRO).

Description	Value
Engine Speed	775 MHz
Compute Units	12
Stream Cores	192
L1 Cache Size	8 kB
L2 Cache Size	512 kB
Bus Width	256 bits
Frame Buffer	1 GB

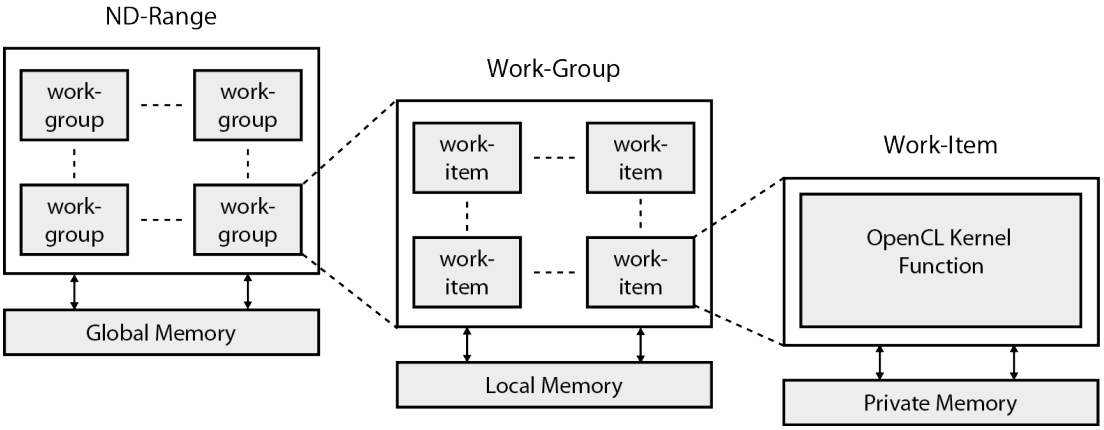

Figure 5.16: OpenCL parallel model.

This particular GPGPU has 12 Compute Units and each compute unit has 16 Stream Cores. Compute units are similar to a core with a private cache in a multicore processor. Stream cores can be represented as hardware light-weight threads that share a local memory. Table 5.5 lists some of the specifications of the AMD GPGPU. We use this hardware device in order to evaluate our parallel algorithm on modern massively parallel processors.

5.5.2 OpenCL C Programming Language

The OpenCL C programming language [30] is used to create programs that describe data-parallel kernels and tasks that can be executed on one or more heterogeneous devices such as CPUs, GPUs, and other processors referred to as

accelerators such as DSPs and the Cell Broadband Engine processor. An OpenCL program is formed of two parts: the host that executes CPU code and the kernel that execute GPU code. Applications cannot call an OpenCL kernel directly but instead queue the execution of the kernel to a command-queue created for a device. The kernel is executed asynchronously with the application code running on the host CPU. OpenCL C is based on the ISO/IEC 9899:1999 C language specification (referred to as C99) [25] with some restrictions and specific extensions to the language for parallelism.

Figure 5.16 illustrates the parallel model from the OpenCL perspective. Work-groups represent the compute units in AMD GPUs. Work items that execute OpenCL kernel functions are the stream cores.

5.5.3 Experimental Results

In order to run our parallel algorithm on graphics processors, we write the code for parallel motion compensation of macroblocks with the OpenCL C Language specifications. The GPU kernel processes independent macroblocks data that are processed in parallel. The frame is divided into 12 groups of macroblocks in order to be executed by compute units. Then, each compute unit processes 16 macroblocks concurrently. The source code is compiled and debugged using the AMD Accelerated Parallel Processing (APP) SDK [3]. Figure 5.17 shows the part of the H.264 decoding process that is executed on the GPU kernel (the shaded rectangle). The remaining stages are executed on the host CPU.

Figure 5.18 shows the speedups attained with the parallel motion compensation on the AMD GPGPU Radeon HD 6850 device. HD resolutions have a speedup of 12.1 and CIF resolutions a speedup of 7.4. These results exclude the data transfer time between the main processor and the graphics processor. In fact, data transfer overhead is still an important limitation in the usage of GPGPUS especially when the level of parallelism is low. In our case, the ratio of the speedup to the number of work-groups is around 0.75 which is considered efficient. Speedup results for CIF and HD resolutions parallel motion compensation on GPU are listed in table 5.6.

Graphics processors have high potential for parallel optimization. The number

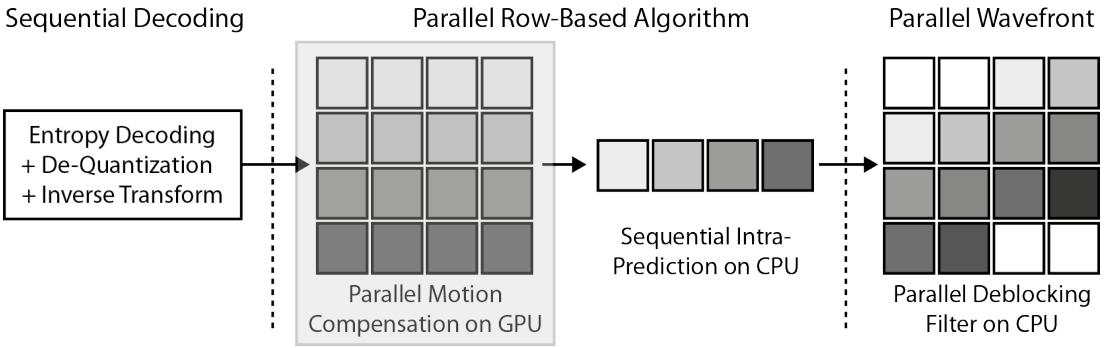

Figure 5.17: Complete H.264 decoding stages with parallel motion compensation on graphics processors (GPU).

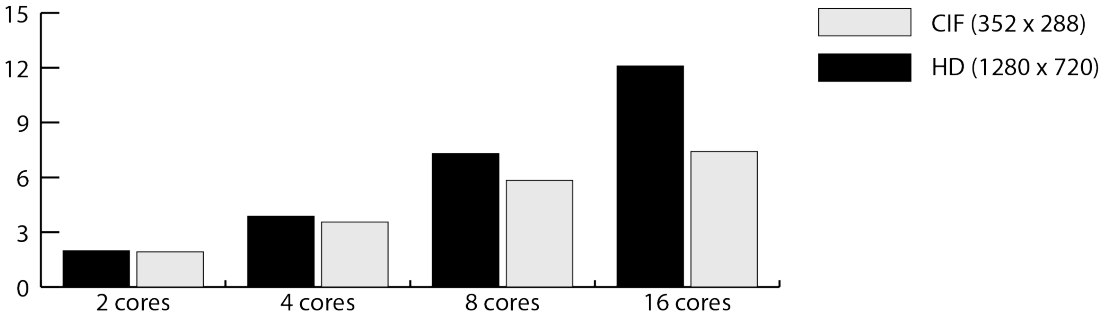

Figure 5.18: Speedup of H.264 parallel execution of motion compensation on graphics processor using CIF and HD video sequences.

Table 5.6: Speedup of H.264 parallel execution of motion compensation on graphics processor.

Resolution	MB Rows	2	4	8	16
HD (1280 x 720)	45	1.983	3.884	7.301	12.095
CIF (352 x 288)	18	1.928	3.560	5.839	7.417

of stream cores is increasing significantly in new devices. These large numbers of parallel cores with recent fast transfer rates with CPUs have huge impact on applications with high parallel data processing like compression algorithms, video games, rendering, etc.

5.6 Conclusion

We have introduced a novel parallel technique for the H.264 video decoder standard. Our approach decodes groups of macroblock rows in parallel with an algorithm that detects dependencies on-the-fly based on isolating intra-prediction macroblocks (I-MBs). Low and high definition video sequences are used in our experiments. The most efficient speedup with the highest ratio to the number of cores of the motion compensation parallel implementation is 3.3 using 4 threads on 4 cores. A parallel macroblock-based implementation of the deblocking filter is also implemented. An overall speedup of 2.3 is attained for the complete H.264 parallel implementation. Our optimized decoder is tested on a real device with an ARM Cortex-A9 processor with 4 cores. The proposed parallel algorithm is tested on a mutlicore simulator in order to explore to scalability of our algorithm on multiprocessors up to 32 cores. Additional experiments are performed on a graphics processor that shows great enhancement with speedups up to 12.1 and high scalability of the proposed parallel motion compensation algorithm.

Chapter 6

Parallel Cache Efficiency

6.1 Introduction

Parallel execution of multi-threaded applications has a great impact on cache memories. We believe that part of cache misses is due to the distribution of the macroblocks data on different cores (data parallelism). Macroblock data signals (YCbCr) are stored in continuous memory area. When decoding takes place on a single core, prefetching of a single cache line (L1, 64 bytes) works very well; however, wrong data are prefetched when different cores decodes different macroblocks that are located in separate memories. Another cause for these misses is due to the processing of the same data at different cores in different decoding phases. These additional data dependencies are not present in sequential implementations because only one core executes the complete sequence of instructions in a program.

In this chapter, we show cache level 1 data bottlenecks of two parallel motion compensation algorithms for the H.264 decoder. Results are compared showing the difference between the row-based and the wavefront parallel processing algorithms. We also present customized software prefetching methodologies for both parallel algorithms. Moreover, we show the results of the impact of data prefetching on speedup of these parallel applications.

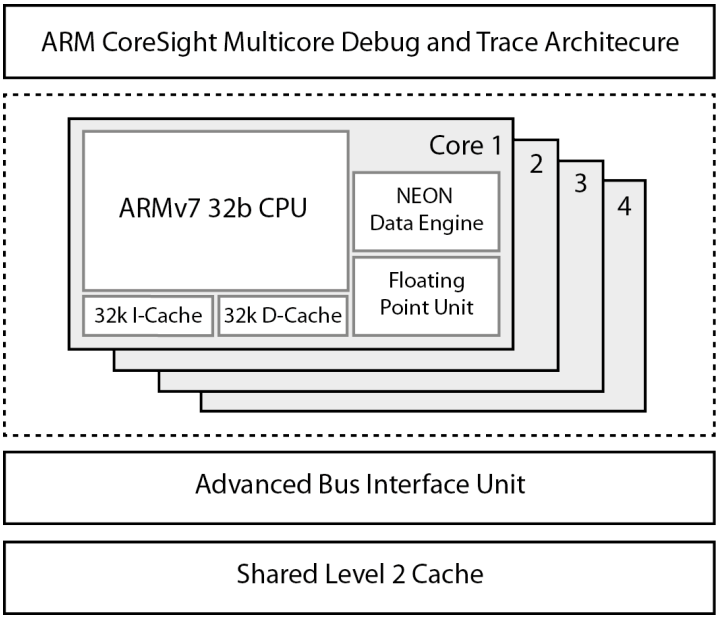

Figure 6.1: Multicore architecture of the ARM Cortex-A9 multiprocessor with an L2 cache shard by 4 cores each having an L1 private cache.

6.2 Parallel Environment

6.2.1 Processor Architecture

In order to collect statistic data among caches in a multicore architecture, we use the Multi2Sim simulator [66]. The processor and the memory configurations are in accordance with ARM Cortex-A9 4-core processor specification [6]. The processor is composed of 4 cores that all have access to a shared memory, the L2 cache, which has 2048 sets with an associativity of 8 and a block size of 64. Each private L1 cache of each core has a geometry that is composed of 128 sets of blocks of size 64 bits each block. The block replacement policy is the Last Recently Used (LRU) algorithm. Figure 6.1 illustrates the basic components of the ARM Cortex-A9 architecture.

6.2.2 Parallel Algorithms

In the experiments that are described and evaluated in this chapter, we use the row-based and the wavefront parallel processing of macroblocks for the motion

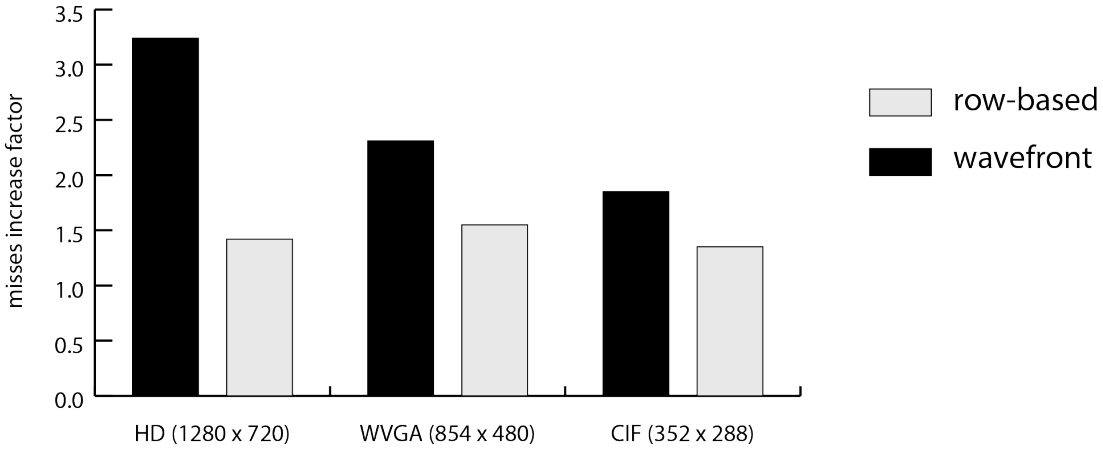

Figure 6.2: Total L1 cache misses for sequential, row-based and wavefront parallel implementations.

compensation stage of the H.264 decoder. For the row-based algorithm, groups of rows of macroblocks are processed in parallel without intra-prediction macroblocks (I-MBs) during the motion compensation stage. These I-MBs are then processed sequentially at the end of the motion compensation stage. For the wavefront algorithm, macroblocks are processed in diagonals where independent macroblocks are processed concurrently. Both the row-based and the wavefront algorithms are explained in more details in chapter 5 section 5.3.

6.3 Multicore Cache Memory

6.3.1 L1 Cache Misses Statistics

Using the configurations described above for the Multi2Sim simulator [66], we experiment the row-based and the wavefront parallel algorithms using three video sequences benchmarks: *waterfall* (352 x 288), *four people* (854 x 480), and *shields* (1280 x 720). We show in figure 6.2 the factor of increase of the total number of L1 cache misses when executing the parallel implementations for each video sequence compared with the sequential execution. The total number of misses for both parallel versions is much higher than the original sequential version. The row-based has about 1.5 times the number of misses of the sequential version. For

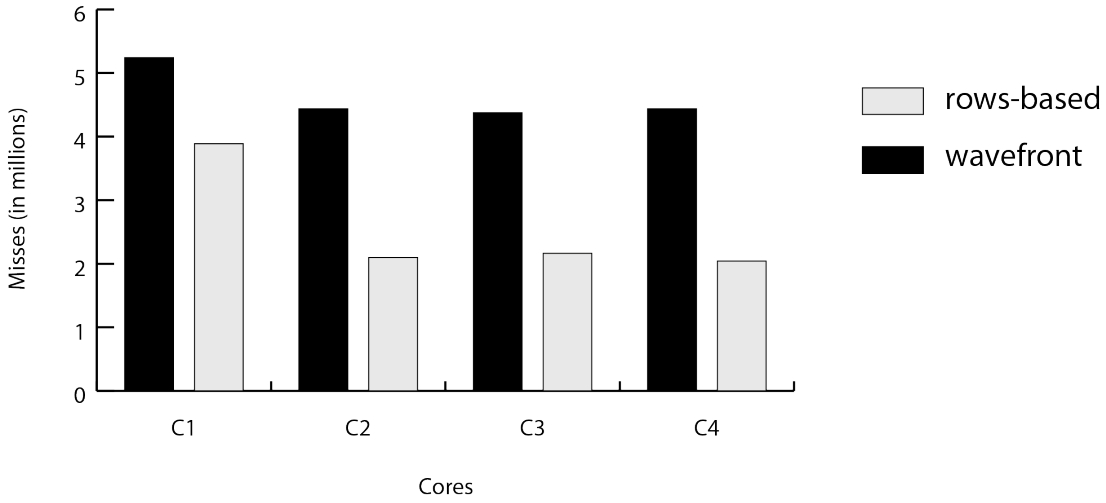

Figure 6.3: L1 cache misses per core for row-based and wavefront parallel implementations of the shields video sequence.

the wavefront implementation, the number of misses reaches 3.2 times compared to the number of misses in sequential execution. The increase factors for the total misses displayed in figure 6.2 are considered high. The sequential implementation is executed on only one core. However, the two parallel versions are executed on a quad-core processor. The total number of misses is increased mainly due to data dependencies between cores during the decoding process. Different cores may require accessing the data of a macroblock that was processed by another core. These dependencies are higher for the wavefront parallel version where nearby macroblocks are processed on different cores. As for the row-based version, several rows of macroblocks are processed on the same core which decreases the data dependencies between cores.

6.3.2 Common L1 Cache Misses among Cores

In this study, we show detailed statistics for L1 cache misses and their distribution among cores of the parallel processor. In figure 6.3, level 1 cache misses per each core of the available 4 cores of the simulated processor are illustrated. Similarly to figure 6.2, the results shown in figure 6.3 show that the number of misses per core for the wavefront parallel version has almost double the number of

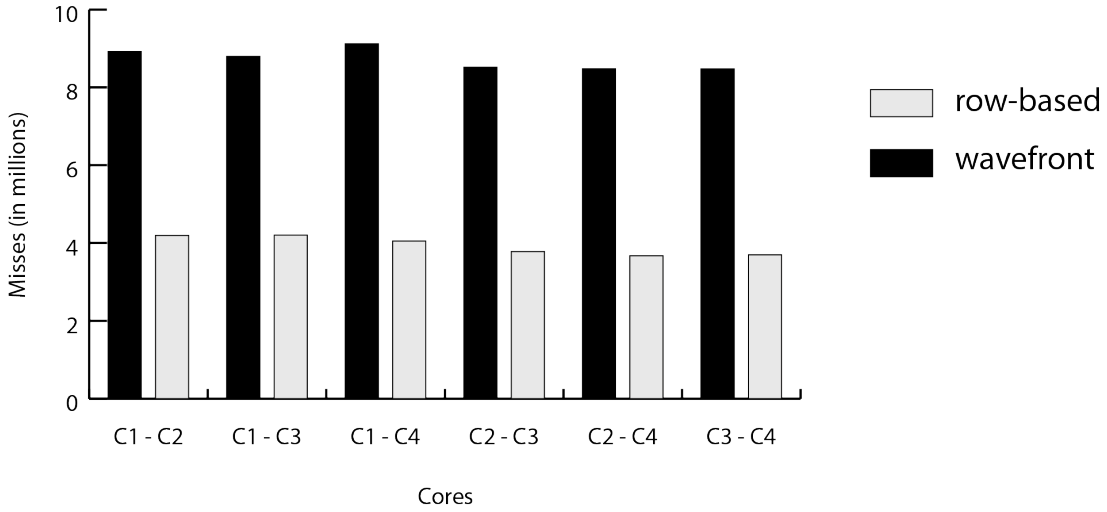

Figure 6.4: Common L1 cache misses between each 2 cores for row-based and wavefront parallel algorithms.

misses for the row-based implementation for HD resolution. One exception is the first core where the difference in the number of misses is only about 20% less of the wavefront algorithm. Moreover, the total number of misses for the first core is higher than other cores. The motion compensation stage always starts with the first core for both parallel versions. Furthermore, in the wavefront parallel version, the first core processes the largest number of macroblocks because at the beginning and at the end of the slice, the number of macroblocks that can be processed in parallel is low. The maximum number of independent macroblocks that can be processed in parallel is when the diagonals divide the frame in its diagonal. As for the row-based version, the rows of macroblocks are divided equally by the number of available cores. However, the remaining rows of macroblocks at the end of the frame are assigned to the first core. For these reasons, the first core has a higher number of misses while the numbers of misses in the remaining cores are almost equal in the same parallel implementation.

Figure 6.4 shows the number of common misses among L1 caches of each core. The misses are gathered when one core access the same data that was already used by another core. These common misses is around 8.4 million for the wavefront version and 4.1 million for the row-based version. The differences between the two versions are almost doubled. These statistics info tells us the number of common

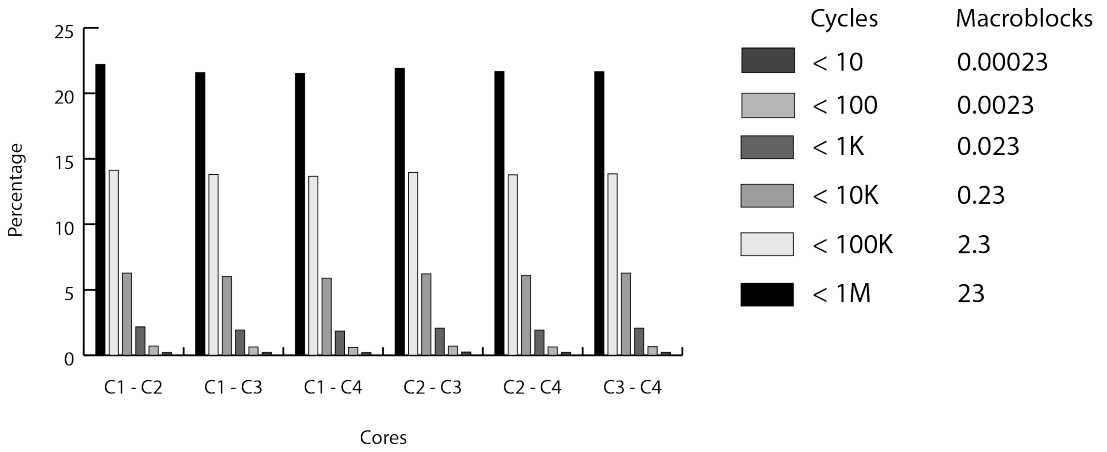

Figure 6.5: Percentage distribution depending on cycles difference of common L1 cache misses between each 2 core for row-based parallel implementation.

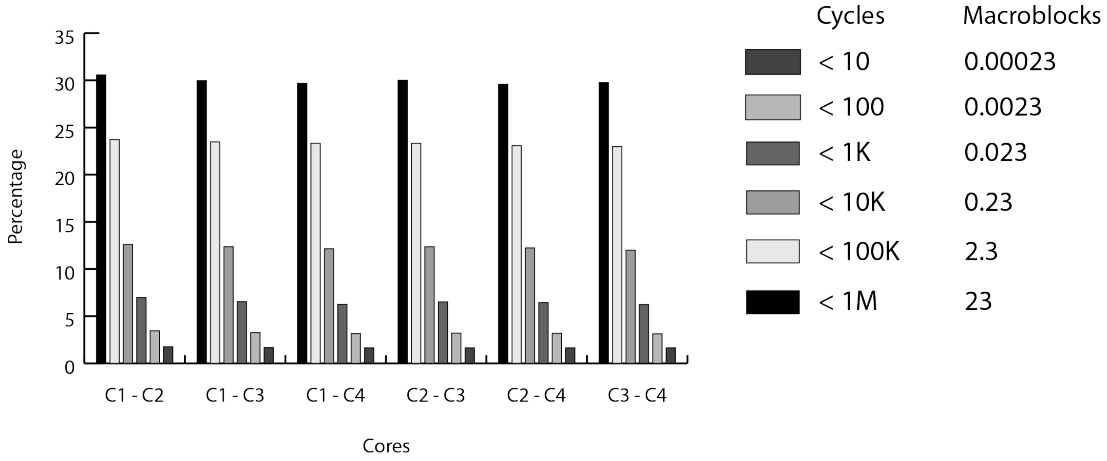

Figure 6.6: Percentage distribution depending on cycles difference of common L1 cache misses between each 2 core for wavefront parallel implementation.

misses between each combination of two cores. These common misses among cores are almost equal for each parallel version. They are almost equal to the sum of the total misses for each combination of two cores. These results mainly show that in both implementations most misses are the same for all cores. However, these misses may not occur in short time differences.

Figures 6.5 and 6.6 show the percentages of the level 1 common misses among cores with the difference of cycles. As we notice, the percentage of misses drops significantly when the difference of cycles is below 100000. These common misses are almost negligible below 100 cycles. In general, the numbers for the wavefront version are much lower than the row-based version which basically means that the wavefront algorithm has higher dependencies between cores as the offset of cycles between the common misses is much higher than the row-based parallel algorithm. The statistics info shown in figures 6.5 and 6.6 reveals the potentials of applying cache optimization techniques like prefetching. Using the statistics from the above figures, prefetching will allow an important decrease in the misses that are common between cores where these misses have a high percentage of the total number of misses.

6.3.3 Parallel Cache Efficiency

The statistics of cache misses that are displayed above reveal several issues related to shared memory architectures in multicore systems.

First, the number of misses that are common among different cores is relatively high. A percentage of 25% or 30% can significantly affect the overall performance of the parallel application. Second, the time difference between common misses for L1 cache memory on parallel cores is large. Most cache misses occur after 100K cycles eliminating time correlation. For this big number of cycles differences, a typical hardware prefetcher will assume that data is no longer needed allowing it to be replaced. Therefore, we cannot apply aggressive hardware prefetching because of cache trashing. Keeping prefetched data for a long time will make the cache inefficient as part of the cache will be blocked with data that will be requested after a long period. Third, any prefetching algorithm that can increase the performance of a parallel algorithm will be suitable only for this specific

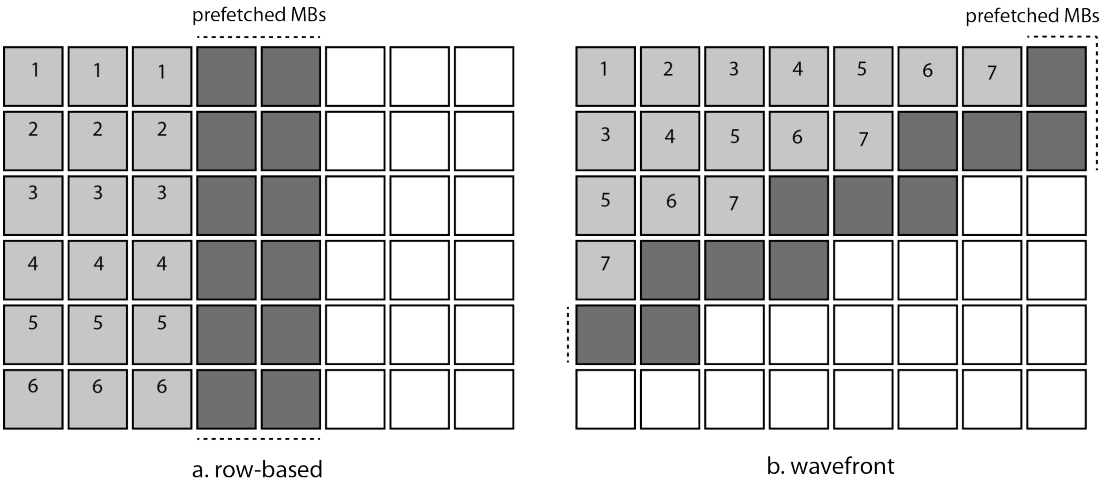

Figure 6.7: Prefetching algorithm for the two parallel motion compensation techniques.

algorithm. In our case, different software prefetching algorithms need to be implemented for each of the row-based and wavefront parallel algorithms. For this reason, hardware optimization is not applicable as it will be only suitable for one specific application. Thus, a software optimization technique is proposed for every algorithm. The software programmer implements hardware dependent source codes which target parallel architectures at compile-time. [21, 32]

In the following section, we will describe our software prefetching algorithms for both the row-based and the wavefront parallel applications. Their performance efficiency will also be tested and analyzed.

6.4 Cache Optimization

6.4.1 Prefetching Algorithm

For any video sequence resolution, the size of each macroblock which is 16 x 16 pixels in the H.264 standard is fixed. So the approximate number of cycles to process each macroblock can be calculated in advance. In our experiments, the average number of cycles that a macroblock required for motion compensation is about 44 thousand cycle. Figures 6.5 and 6.6 on page 109 show the fraction or the number of macroblocks that is equivalent to the number of cycles. Macroblocks

in both parallel algorithms of the H.264 decoder are processed in parallel where their dependencies to other macroblocks are known in advance. A smart software prefetching algorithm loads data of specific macroblocks that will be used during the decoding stages.

In our example, we load the data of 23 macroblocks in order to get the most of data prefetching. So, a software prefetching algorithm is implemented in order to load the data of the macroblocks that will be accessed during motion compensation. As shown in our cache statistics, most of common cache misses are within 1 million cycles of difference. Hence, there is no need to load the data of more than 23 macroblocks. For the row-based and the wavefront algorithms, the following macroblocks that will be processed in parallel are prefetched to the cache. Figure 6.7 shows which macroblocks are prefetched for the row-based and the wavefront algorithms. In this figure, a small frame of a video is depicted. The squares in dark gray are the macroblocks that are being prefetched. The squares in light gray are macroblocks which are already processed or are currently being processed. The example in figure 6.7 shows that data continuity exists only on the same row in the row-based and wavefront algorithms. The segmentation of data that is shown in the memory access pattern through their distribution on different cores misleads the built-in hardware prefetcher. Any enhancement should work on this problem by providing a global view of the data that needs to be prefetched by cache memories on different cores. Therefore, our proposed software prefetching algorithms are only applicable to the above parallel algorithms used in our experiments.

A software prefetching algorithm that is specific for MPEG-4 had been proposed by Cucchiara et al. [16]. On the other hand, Wang et al. [69] implemented private caches with dynamic reconfiguration where the shared cache is partitioned for multicore systems with real-time tasks. In addition, Nesbit et al. [41] proposed a FIFO history buffer which can improve the accuracy of correlation prefetching by eliminating stale data. In the following section, we will show the performance of our proposed prefetching technique.

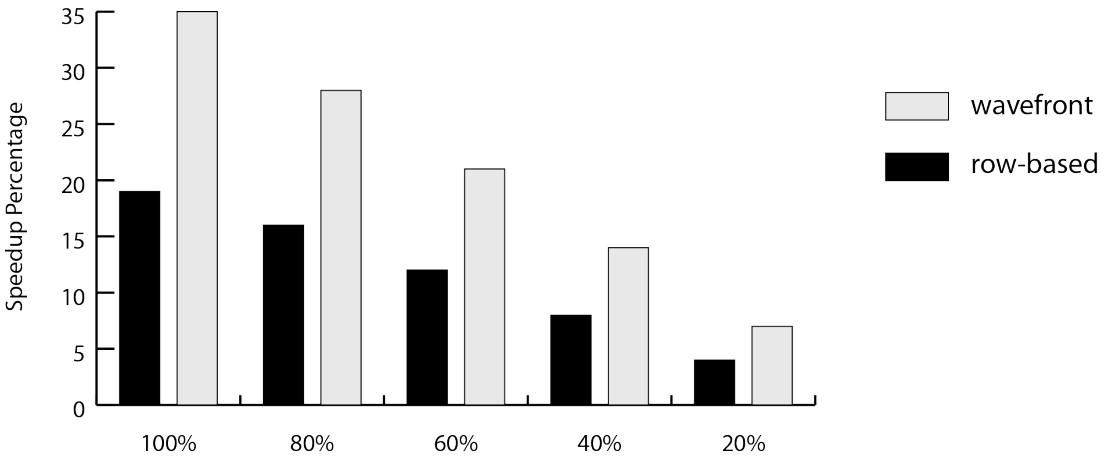

Figure 6.8: Speedup percentage for the row-based and wavefront parallel implementations depending on the successful rate of data prefetching.

6.4.2 Performance Efficiency

Using our proposed data prefetching algorithm for H.264 parallel motion compensation, we minimize data dependencies and memory stalls between local private cache blocks of each core in a multicore processor. This decrease in data dependencies will eventually decrease the number of cache misses leading to a better performance. A bigger buffer has no impact on the performance. However, a smaller buffer affects the performance speedup that is achieved by data prefetching.

We illustrate in figure 6.8 the impact if data prefetching on the row-based and wavefront parallel implementations. The average number of cycles for load instructions in L1 cache is 5 cycles and in L2 cache of 30 cycles. Figure 6.8 displays the projected percentage of the number of cycles that will be eliminated with data prefetching. The results also depends on the rate of success of data prefetching. For example, with a 60% success rate, the total number of cycles in order to complete the program will decrease by 21% for the wavefront parallel implementation and by 13% for the row-based parallel implementation. In the following section, we will discuss the overall performance in terms of cycles and instructions.

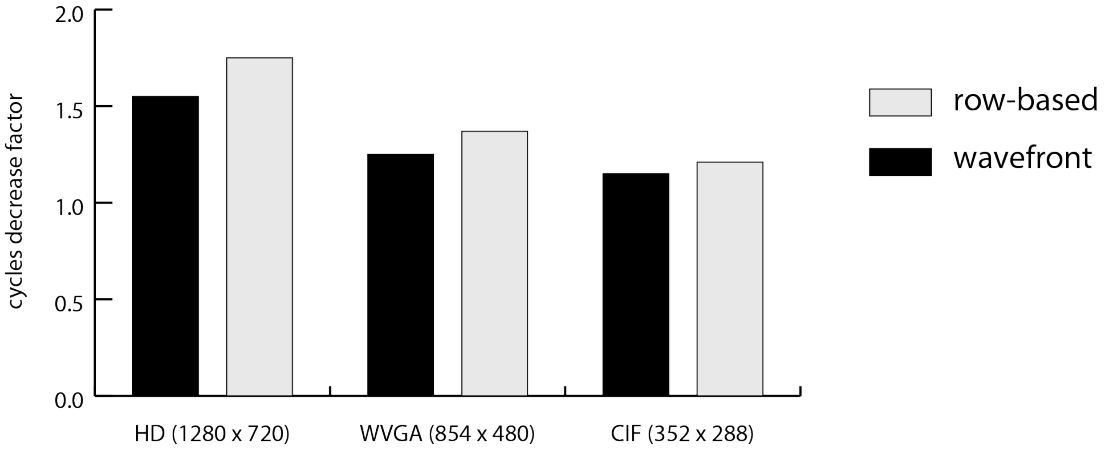

Figure 6.9: Total number of cycles for row-based and wavefront parallel implementations.

6.5 Instructions and Cycles Statistics

Having displayed statistics related to misses, we now focus on the number of cycles and the average instructions per cycle in order to compare overall execution of the parallel implementations. Figure 6.9 illustrates the total number of cycles for the sequential and the parallel versions. We notice that the number of cycles for the row-based and the wavefront parallel implementations is almost 1.7 and 1.5 times less than the sequential execution. In addition, the average number of instructions per cycles is shown in figure 6.10 where the row-based algorithm is almost 3.0, the wavefront version is about 2.3, and the sequential version equal to 1.5. These numbers shows that an average speedup close to 1.6 is attained by the two parallel algorithms in comparison with the sequential implementation. In addition, these speedups results show that the data prefetch algorithm that is discussed above has a successful rate close to 60% of the maximum speedup that can be reached. These overall values include overheads in terms of additional cycles, data transfer, shared memory, etc.

After gathering all the above statistics, we conclude that the row-based parallel algorithm of the H.264 decoder is efficient in terms of parallel execution among cores. The low number of data dependencies between cores which makes the core execution independent of others as much as possible. This low dependency is also very important because it minimizes memory bottlenecks and the need to wait

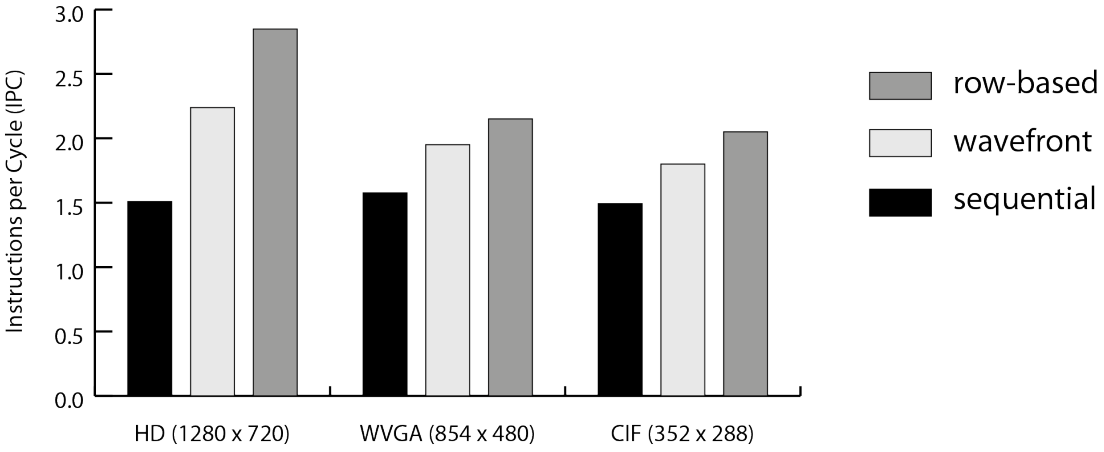

Figure 6.10: Average Instructions per Cycle (IPC) for sequential, row-based and wavefront parallel implementations.

for memory coherency protocols between private caches of multiple cores. On the other hand, the wavefront algorithm require enhancement in order to become more efficient. As shown in our results, minimizing the number of common misses between cores has a huge effect on the overall speedup of the application and on its scalability with the number of parallel cores.

6.6 Conclusion

In this chapter, we presented detailed memory statistics of two parallel algorithms for the H.264 motion compensation stage. The experiments are conducted on a multicore simulator in order to collect and to analyze level 1 cache memory misses. Common misses among cores of the same address are also collected and discussed. The impact of cache memory on the overall execution is also evaluated. Furthermore, a software prefetching algorithm is proposed for the row-based and wavefront parallel algorithms. The impact of prefetching is also calculated for both parallel algorithms.

Chapter 7

Conclusion

H.264 is being widely adopted in multimedia applications on general-purpose and embedded systems. The high complexity imposed by the H.264 decoder requires enhancement in order to increase the efficiency and to lower power consumption.

We proposed and evaluated a parallel algorithm for the H.264 decoder. Luma and chroma color components for the motion compensation stages are processed in parallel providing high and realistic potentials for video decoding on dual and quad core processors. Execution time speedup of our parallel implementation of the H.264 decoder is around 18%. Moreover, the speedup reaches 32% with our proposed pipeline implementation with an energy saving of 24%.

Moreover, an advanced parallel algorithm is also proposed that executes motion compensation on large number of parallel cores. The parallel approach is based on processing groups of independent macroblock rows in parallel. The proposed parallel algorithm shows a higher scalability than the color components approach. Thus, good speedup and energy saving are reached on multicore processors with more than 8 cores. In this approach, groups of macroblock rows are decoded in parallel with an algorithm that detects dependencies on-the-fly based on isolating intra-prediction macroblocks (I-MBs). Low and high definition video sequences are used in our experiments. The most efficient speedup with the highest ratio to the number of cores of the motion compensation parallel implementation is 3.3 using 4 threads on 4 cores. A parallel macroblock-based implementation of the deblocking filter is also implemented. An overall speedup

of 2.3 is attained for the complete H.264 parallel implementation. Our optimized decoder is tested on a real device with an ARM Cortex-A9 processor with 4 cores. The proposed parallel algorithm is tested on a mutlicore simulator in order to explore to scalability of our algorithm on multiprocessors up to 32 cores. Additional experiments are performed on a graphics processor that shows great enhancement with speedups up to 12.1 and high scalability of the proposed parallel motion compensation algorithm.

In addition, we evaluated and discussed the impact of cache misses on the overall performance of the row-based and wavefront parallel algorithms of the H.264 decoder. Detailed memory statistics of two parallel algorithms for the motion compensation stage are presented and analyzed. The experiments are conducted on a multicore simulator collecting level 1 cache memory misses. Common misses among cores of the same address are also collected and discussed. Furthermore, customized software prefetching algorithms are proposed for two parallel algorithms. The impact of prefetching is also calculated for both parallel algorithms.

The work in this research presents solutions to the high complexity of the H.264 decoder using parallel computing. These algorithms can be applied to most block-based video compression standards. Further experiments and enhancements need to be performed in order to apply our parallel algorithms in commercial products. Our intention is to continue our research in order to increase the efficiency and lower the energy consumption of recent video coding standards.

Chapter 8

Résumé en Français

8.1 Introduction

8.1.1 Contexte

Aujourd'hui, les appareils mobiles qui supportent les applications multimédias sont omniprésents dans notre monde moderne. La plupart des appareils portatifs sont équipés d'écrans hauts résolution et des processeurs multicœurs embarqués. Les processeurs aux cœurs doubles et quadruples sont trouvés dans les smartphones et les tablettes comme les appareils offerts par Samsung et Apple [5, 53]. Le processeur ARM Cortex-A9 peut avoir jusqu'à 4 cœurs par puce [6]. Le processeur Cortex-A15 peut avoir jusqu'à 8 cœurs par puce [7]. Néanmoins, les applications ne bénéficient pas automatiquement de ces processeurs puissants haut-de-gamme. Même avec les nouveaux processeurs de pointe, les résolutions vidéo sont en croissance rapide qui nécessite plus de temps de traitement, et par conséquent, plus de consommation d'énergie. Les systèmes d'exploitation affectent tout simplement des applications indépendantes, ou les threads d'une application, sur des différents noyaux. Pour cette raison, une application seule ne peut pas bénéficier des ressources supplémentaires que si elle est conçue pour exécuter en parallèle. Ainsi, les applications séquentielles doivent être modifiées et recompilées afin de soutenir le parallélisme. Le processus de parallélisation confronte à de nombreux défis comme la dépendance, la synchronisation, la cohérence des données, etc.

Les lecteurs vidéo, les appareils photo numériques, les téléviseurs et les téléphones

utilisent des codecs vidéo complexes pour les résolutions élevées. Cependant, quelques applications multimédia bénéficient des potentiels de calcul parallèle offert par des processeurs multicœurs embarqués. Les codecs vidéo récents, comme H.264/AVC [26] et HEVC [63], ont adopté des algorithmes complexes afin d'optimiser la compression et de diminuer les débits de transmission. La complexité supplémentaire de ces algorithmes a des impacts négatifs sur le temps d'exécution et la consommation d'énergie.

8.1.2 Déclaration du Problème

H.264/AVC [26] est un standard de codage vidéo puissant avec des algorithmes complexes. Le codec réalise une bonne compression mais il cause un ralentissement des performances. Même avec les nouveaux processeurs de pointe, les résolutions des vidéos sont en croissance rapide, ce qui nécessite plus de temps de traitement et par conséquent plus de consommation d'énergie. Une des meilleures stratégies d'optimisation de temps et d'énergie est d'exécuter une application sur des noyaux multiple en parallèle. La conversion ou la modification d'une application afin d'être exécuté en parallèle présentent de nombreux défis tels que les dépendances, la synchronisation, la cohérence des données, la mémoire partagée, etc. Dans notre recherche, nous utilisons le décodeur vidéo H.264 comme une application multimédia complexe pour appliquer le parallélisme. Nous résolvons le problème de la grande complexité du décodeur H.264 en utilisant l'exécution en parallèle sur des processeurs multicœurs embarqués afin de réduire le temps d'exécution et la consommation d'énergie.

8.1.3 Solutions Existantes

De nombreuses implémentations parallèles du décodeur H.264 existent allant du décodage en parallèle des macroblocks (grains fins) jusqu'au décodage en parallèle des groupes d'images (gros grains). Un macroblock est un composant de 16x16 pixels en carré d'une image dans une séquence vidéo. Il peut également être divisé en sous-blocs. Le décodage en parallèle des macroblocks est hautement évolutif en raison de nombreux macroblocks indépendants qui peuvent être traitées en parallèle. Toutefois, les dépendances sont créées à la suite de la commu-

nication de la mémoire et de la synchronisation d'exécution entre les macroblocks. En plus, le décodage parallèle des groupes d'images nécessite une grande capacité de mémoire, en particulier pour des séquences vidéo avec une haute définition. En outre, ils ont une extensibilité inférieure à celle des macroblocks à cause du petit nombre des groupes de frames (images) qui peuvent être décodées en parallèle.

8.1.4 Contributions

Nos approches pour décoder des vidéos H.264 en parallèle au niveau des macroblocks sont originales et uniques. D'autres techniques sont utilisées pour réduire la charge de la partie séquentielle comme le décodeur entropique.

Tout d'abord, nous séparons le processus de décodage entre les composantes de couleur pour chaque échantillon de données de chaque macroblock. Ensuite, on applique un pipeline afin de minimiser le temps de blocage provoquée par la synchronisation des taches parallèles. L'implémentation en parallèle est expérimenté sur un simulateur des processeurs embarqués dual et quad. En plus, le temps d'exécution et les statistiques d'utilisation de la mémoire, les résultats de la consommation d'énergie sont présentés à l'aide d'un outil d'estimation puissant.

Pour notre deuxième approche, nous traitons les lignes de macroblocks indépendants en parallèle en utilisant un algorithme innovant qui minimise la synchronisation sans ajouter des mesures supplémentaires pour le décodeur. L'étape de compensation de mouvement (motion compensation) est générée en utilisant un algorithme sur les lignes de macroblocks. L'étape de filtre de déblocage utilise un algorithme appelé wavefront. Ce niveau d'exécution en parallèle qui est basé sur les lignes de macroblocks peut être considéré entre approches parallèles à grain gros et à grain fin offrant un équilibre entre les deux solutions. L'algorithme parallèle proposé est évalué sur un simulateur multicœurs avec des plates-formes multicœurs et des processeurs graphiques.

8.1.5 Plan

Dans la section 8.2, nous présentons les concepts de calcul parallèle en termes des algorithmes, des architectures de mémoire, et des applications. Dans la section 8.3, un aperçu du standard H.264 est présenté. Les implémentations

parallèles existantes et les recherches reliés au standard H.264 sont également présentées. Notre première implémentation du H.264 en parallèle qui est basé sur les composants de couleurs en parallèle est expliquée et évaluée dans la section 8.4. L'accélération du temps d'exécution et les statistiques d'économie d'énergie sont illustrés. Dans la section 8.5, le deuxième algorithme parallèle pour le décodeur H.264 est décrit et expérimenté. Les groupes de macroblocks sont traités en parallèle sur des différents noyaux. L'algorithme est évalué sur des plateformes multicœurs en temps réel. Les résultats de simulation avec un nombre élevé de noyaux parallèles sont également présentés et discutés. Enfin, une conclusion dans la section 8.6 résume notre contribution dans la dernière section.

8.2 Programmation Parallèle

Le calcul parallèle est une forme de traitement de l'ordinateur lorsque les tâches sont exécutées simultanément en même temps [2]. Il existe des différents niveaux de calcul parallèle: parallélisme au niveau des instructions, parallélisme au niveau des données, et parallélisme au niveau des tâches. Le parallélisme a été principalement utilisé dans les serveurs à haute performance et les supercomputers. Il y a dix ans, le calcul parallèle a apparu comme une solution à l'augmentation de la fréquence en raison des contraintes physiques [48]. Comme la consommation d'énergie par les ordinateurs est devenue un facteur important dans les systèmes informatiques, le calcul parallèle est devenu le modèle dominant dans les architectures informatique, principalement pour les processeurs multicœurs. [8]

Plusieurs types d'ordinateurs parallèles existent comme multicœurs et multiprocesseurs équipés de plusieurs processeurs dans une seule machine. Clusters et grids utilisent plusieurs ordinateurs pour travailler sur la même tâche simultanément. Les architectures parallèles spécialisées comme les GPU sont également utilisés avec les processeurs traditionnels afin d'accélérer des tâches spécifiques comme les calculs graphiques.

Les programmes parallèles sont beaucoup plus difficiles à écrire que les programmes séquentiels [21]. La simultanéité présente généralement plusieurs bugs logiciels, tels que les races conditions. La communication et la synchronisation

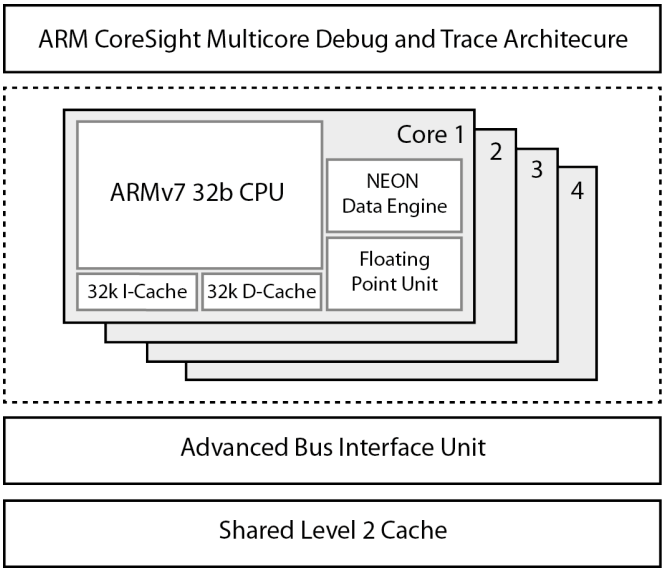

Figure 8.1: Architecture du multiprocesseur ARM Cortex-A9 avec 4 noyaux

entre les tâches parallèles sont généralement les plus grands inconvénients qui affectent considérablement la performance. Théoriquement, l'accélération maximale possible d'un programme à la suite d'un traitement parallèle est connue par la loi d'Amdahl. [4]

8.2.1 Processeurs Multicœurs

Les puces génériques les plus courantes sont les processeurs multicœurs qui sont aujourd'hui disponibles dans la plupart des ordinateurs desktops et portables. Un processeur multicœurs est un processeur qui comprend de multiples unités d'exécution, appelés noyaux (cores), sur la même puce. Un processeur multicœur peut émettre plusieurs instructions par cycle. Chaque noyau dans un processeur multicœur peut potentiellement être superscalaire, où chaque noyau peut émettre plusieurs instructions par cycle pour un flux d'instructions. La communication entre les noyaux est habituellement maintenue par un accès à la mémoire partagée. Les processeurs multicœurs dominent le marché de la consommation pour les ordinateurs personnels avec la famille des processeurs Intel Core [24] et pour les appareils portables avec la famille des processeurs ARM Cortex [6]. Figure 8.1 illustre l'architecture simplifiée du processeur multicœur ARM Cortex-A9 à

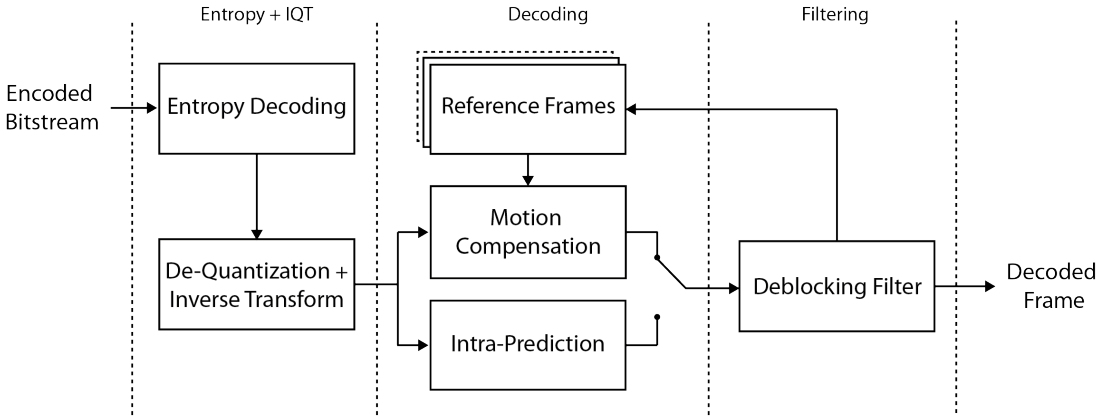

Figure 8.2: Processus du décodeur H.264

quatre noyaux 32 bits et un cache partagé à 2 niveaux.

8.3 Standard H.264

Le groupe Moving Picture Experts (MPEG) et le groupe Video Coding Experts (VCEG) ont élaboré conjointement en 2003 le standard *Advanced Video Coding* (AVC) publié comme la Recommandation ITU-T H.264 et la partie 10 de MPEG-4 [26]. Dès les premières applications commerciales, plusieurs fabricants d'appareils multimédia ont adopté le nouveau codec vidéo. Dix ans après la première publication de la version finale, le standard H.264 est actuellement le plus utilisé pour la compression vidéo dans les appareils multimédia selon de nombreux articles et des revues comme PCWorld.com [23]. Les appareils photo, smartphones, PDA, vidéosurveillance, lecteurs de disques Blu-ray et de nombreux autres dispositifs utilisent H.264 pour l'encodage et le décodage des vidéos. H.264 permet d'obtenir une meilleure compression et une meilleure qualité au détriment des algorithmes plus complexes. Ainsi, plus de ressources de calcul sont exploitées et plus d'énergie est consommée lors de l'augmentation du taux de compression des fichiers vidéo. Cette section donne un aperçu de quelques-unes des principales caractéristiques du standard.

8.3.1 Décodeur H.264

Le processus du décodeur est représenté dans la figure 8.2. Le décodeur peut être divisé en cinq parties fonctionnelles principales: *entropy decoding* (ED), *de-quantization* et *inverse transform* (IQT), *motion compensation* (MC) et *intra-prediction* (IP), et *deblocking filter* (DF).

La figure 8.2 illustre une représentation simplifiée des étapes du décodeur H.264. Le décodage entropique (ED) et la compensation de mouvement (MC) sont appliqués pour chaque macroblock de taille 16x16 pixels. Le déblocage de filtrage (DF) est exécuté à la fin du processus de décodage. La charge de travail moyenne de chaque étape à l'aide du profil de base (baseline) est illustré dans la figure 8.3. Les étapes de décodage entropique et la de-quantification et la transformation inverse (IQT) sont fusionnés en une seule étape dans les statistiques de la figure 8.3. La charge de travail de cette étape est de 14% en moyenne qui est principalement consommée par l'algorithme *context-adaptive variable length coding* (CAVLC). L'algorithme CAVLC est adopté par le profil baseline et il a une complexité inférieure à celle de l'algorithme de CABAC. L'étape de prédiction qui est constitué d'intra-prédiction et de motion compensation a un impact important sur l'ensemble du processus de décodage dont 41% en moyenne. Enfin, l'étape de filtre de déblocage est également un processus lourd qui consomme environ 45% de l'ensemble du processus. Ces statistiques de la charge de travail sont profilées à l'aide de plusieurs benchmarks vidéo avec basse et haute résolutions.

8.4 Décodage des Couleurs en Parallèle

Nous proposons notre approche qui traite chaque composante de couleur (luminance et chrominance) sur un noyau séparé dans un processeur multicœur afin d'augmenter la performance globale du décodeur H.264. Notre nouvelle idée est basée sur le fait que le décodeur H.264 traite en série les composantes de couleurs dans chaque image; ainsi, le traitement simultané des composantes de couleur est possible grâce à l'indépendance des données du luma et du chroma. En outre, une version de pipeline est conçu pour améliorer l'équilibrage de charge et de cacher les effets de la synchronisation. Des simulations sont effectuées en utilisant des

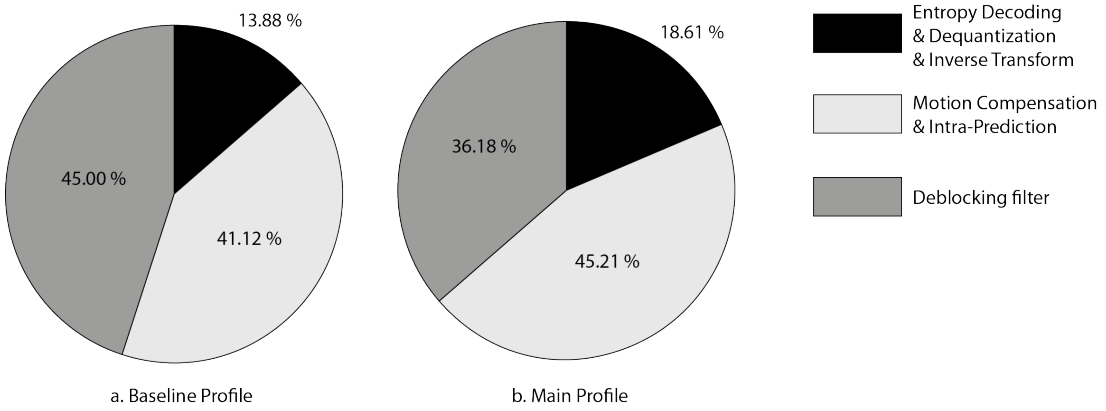

Figure 8.3: Charges moyennes des étapes du décodeur H.264 sur le processeur ARM Cortex-A9

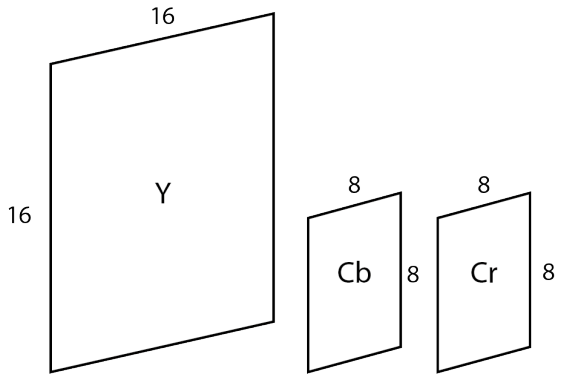

Figure 8.4: Format 4:2:0 des échantillons de couleurs

vidéos benchmarks qui sont simulés sur des processeurs multicœurs embarqués. Des expérimentations sont menées sur le simulateur des processeurs dual et quad cores afin de recueillir le temps d'exécution et les statistiques de consommation d'énergie. [9]

8.4.1 Composants de Couleurs

Les frames (images) d'une séquence vidéo sont représentés comme des streams de bits. Les pixels sont échantillonnés à l'aide de trois composantes de couleur: YUV (ou YCrCb). Y représente un échantillon de couleur de luminance (luma) qui est l'information de la lumière. UV (ou CrCb) représente les échantillons des couleurs rouge et bleue respectivement (chrominance). Dans un format 4:2:0

chaque quatre échantillons de luminance ont un échantillon rouge et un échantillon bleu comme le montre la figure 8.4. Le format d'échantillonnage 4:2:0 est le format le plus utilisé. Des autres formats 4:2:2 ou 4:4:4 où plusieurs échantillons de couleurs sont disponibles, ne montrent pas une différence significative pour la vision humaine. La raison est que la vision est plus affectée par la lumière que par les couleurs des séquences vidéo.

Dans notre recherche, nous révélons un model indépendant qui se trouve dans le processus de décodage des composantes de couleur. Comme nous l'avons décrit plus haut, chaque image d'une séquence vidéo est représentée dans les échantillons de couleurs YCrCb. Un pixel est constitué par ces trois composantes de couleur. Le décodeur reconstruit les données des couleurs dans chaque image séparément à partir des couleurs de luminance et de chrominance. Les informations de couleur dans chaque frame, et donc dans chaque macroblock sont indépendants l'un de l'autre. Le H.264 Standard [26] ne montre aucune dépendance entre les données d'information de couleur de l'algorithme de décodage lors de l'étape de compensation de mouvement.

8.4.2 Exécution en Parallèle and Synchronisation

Le processus de décodage H.264 et tous les principaux algorithmes de décodage vidéo sont généralement conçus pour être exécutées séquentiellement. Le standard H.264 [26] ne prend pas en charge le parallélisme, et donc, ne bénéficie pas des processeurs multicœurs qui sont disponibles dans le marché d'aujourd'hui. Plusieurs approches ont été étudiées afin de paralléliser l'exécution du processus de décodage. La plupart de ces approches sont basées sur les slices (une image est composée d'une ou de quelques slices) et sur les macroblocks (ils sont expliqués dans la section 3.4.3.3 et 3.4.3.2 dans le chapitre 3). Des approches similaires pour le traitement parallèle de H.264 sont décrits en détail dans la section 3.7 du chapitre 3.

Dans notre recherche, nous avons modifié le code source H.264 afin de décoder les composants de luma et de chroma dans chaque macroblock en parallèle. Un noyau gère toutes les étapes sauf la compensation des mouvements de chrominance et l'intra-prédiction qui sont exécutés sur le second noyau comme le montre

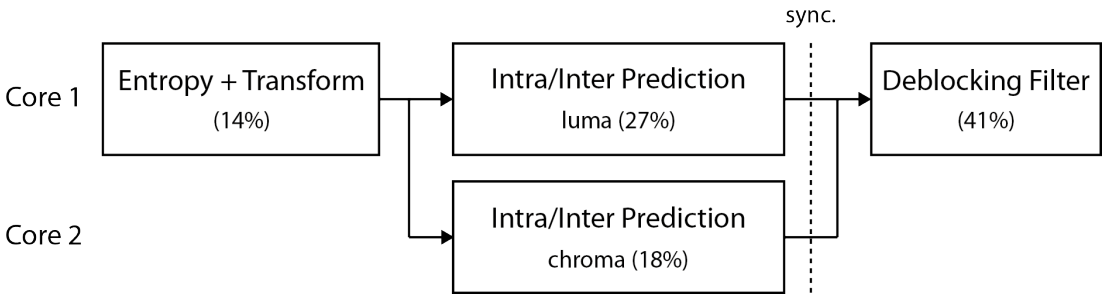

Figure 8.5: Décodage H.264 des composantes des couleurs en parallèle sur un processeur dual-core

la figure 8.5. Le premier noyau exécute les échantillons de couleurs de luminance en plus de tous les étapes de décodage restantes. Comme indiqué ci-dessus, les composantes de couleur sont indépendantes l'une de l'autre. Par conséquent, le décodage des différentes composantes de couleur en parallèle est correct en théorie, ainsi que dans les expérimentations.

L'intra-prédiction et la compensation de mouvement (inter-prédiction) doivent être terminées avant d'appliquer le filtre de déblocage. Ainsi, une barrière de synchronisation est nécessaire avant que l'étape de filtre de déblocage commence. Avec cette configuration, la synchronisation est effectuée à la fin du décodage de la luminance et de la chrominance. A la fin de l'étape de décodage entropique, l'étape de l'exécution en parallèle du processus de décodage est déclenchée. Une fois l'intra-prédiction et la compensation de mouvement sont terminées, toutes les taches parallèles attendent à la barrière de synchronisation avant de commencer l'étape du filtre de déblocage.

Lors de l'utilisation d'un processeur dual-core, le second noyau utilise les données de chrominance afin de décoder les composantes de couleur. Le premier noyau décode les données de luminance et exécute les algorithmes séquentiels restants comme illustré dans la figure 8.5. La charge de travail moyenne du deuxième noyau en utilisant les vidéos Akiyo et Container (CIF et QCIF) est de 18%, ce qui est à son tour le gain de performance moyen en utilisant des processeurs dual-core.

Figure 8.6 illustre la répartition de la charge de travail sur 4 noyaux. Le premier noyau lit les données à partir des unités NAL (qui sont expliqués dans la

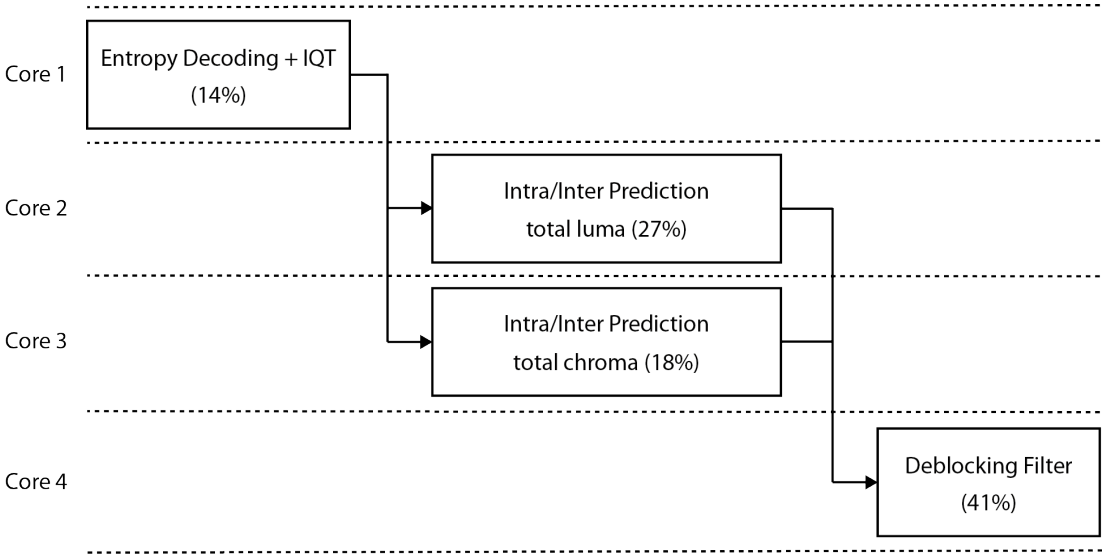

Figure 8.6: Décodage H.264 des composantes des couleurs en parallèle sur un processeur quad-core

section 3.4 du chapitre 3) et effectue le décodage d'entropie et la transformation. Le second noyau traite les données de luminance tandis que le troisième noyau traite les données de chrominance. Ils ont tous deux effectué le même travail, intra-prédiction et compensation de mouvement, sur les différentes composantes de couleur en même temps. Le quatrième noyau exécute la partie restante du filtre de déblocage. Le gain de performance en utilisant des processeurs quad-core est presque le même que les processeurs dual-core en raison des caractéristiques séquentielles du décodeur H.264. Ainsi, afin de bénéficier des architectures quad-core, une exécution en pipeline est proposée et discuté dans la partie suivante.

8.4.3 Exécution en Mode Pipeline

Afin de minimiser le temps d'attente entre les noyaux parallèles, les étapes de décodage H.264 sont exécutées en mode pipeline sur quatre noyaux, lorsque la compensation de mouvement est appliquée. Théoriquement, le temps d'exécution peut être considérablement diminué au temps requis par le noyau qui s'exécute au plus grand morceau de code. Le temps d'inactivité d'un processeur est réduit pour une plus grande efficacité en utilisant les ressources disponibles. Le pipeline

est illustré dans la figure 8.7.

Une mémoire partagée stockant les blocks de données sont utilisées afin d'accéder à des données cohérentes par les quatre processeurs. Les variables de données et leur manipulation dans l'implémentation du H.264 actuel sont passés par des modifications importantes et par des essais pour prouver l'exécution du pipeline proposé. Figure 8.7 illustre le meilleur des cas où les quatre étapes de décodage sont totalement indépendantes, ce qui permet l'exécution parallèle en utilisant les quatre cœurs. Ce cas est appliqué lorsque l'image actuelle est dépendante d'une image précédemment décodée (P-frames). P-frames contiennent des macroblocks intra et inter. Les images d'intra-prédiction (images I) ne permettent pas l'exécution de décodage pipeline parce que les macroblocks dépendent d'autres macroblocks dans la même image. I-Frames ne contiennent que des macroblocks intra (I-MB). Le gain de performance dépend du nombre des I-frames dans les images vidéo codées où la première image décodée est toujours une I-frame. D'autre part, les images suivantes codées en utilisant les profils principaux sont pour la plupart des images des P-Frames. Les I-MBs sont utiles lorsque les images adjacentes sont assez similaires et avec mineur mouvement. Pour les benchmarks de vidéo Akiyo et Container, seule la 1ère image a été décodée à l'aide d'intra-prédiction parmi 300 images. Ce fait nous amène à souscrire au moins 99% des images décodées sont des P-Frames. La charge maximale de la version pipeline H.264 sur 4 noyaux est de 41% qui est exécuté par le quatrième noyau (P4) afin d'effectuer le processus de filtre de déblocage. Donc, l'accélération de la performance maximale est limitée par la plus grande charge qui est le filtre de déblocage.

8.4.4 Résultats

Afin de démontrer la faisabilité de notre approche, nous avons réalisé des expérimentations sur notre version parallèle du référence décodeur H.264 [61] en utilisant le simulateur de MPARM [31]. Les statistiques d'exécution de chaque étape sont collectées en plus des résultats globaux d'exécution.

La différence des composantes de couleur entre les deux séquences vidéo et à travers toutes les étapes sont similaires, comme indiqué dans le tableau 8.1. Pour un échantillonnage 4:2:0, chaque échantillon de 4 luma est regroupé avec 2

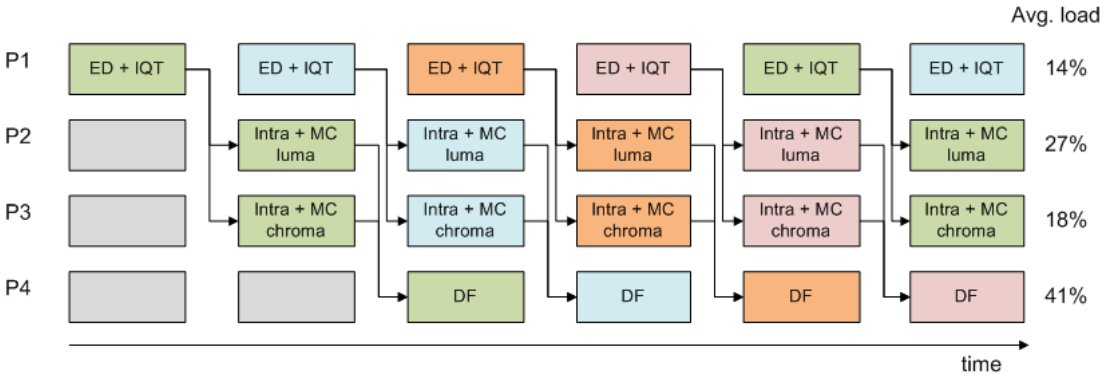

Figure 8.7: Exécution en pipeline H.264 sur un processeur quad-core

Table 8.1: Résultats d'exécution parallèle de luma and de chroma

Benchmark	**Total (ms)**	**Luma (%)**	**Chroma (%)**
Akiyo CIF	754234	29.48	20.06
Akiyo QCIF	379928	24.56	15.93
Container CIF	763781	29.45	20.45
Container QCIF	397664	24.55	15.91
Average		27.01	18.08

échantillons chroma. Ainsi, le traitement de luminance a besoin un temps double que celui de chrominance. Les informations de couleur de luminance ont 27% en moyenne de la durée totale d'exécution. Le temps total d'exécution du chroma est de 18% en moyenne de la durée totale d'exécution.

Dans la section suivante, un algorithme parallèle avancé est proposé pour l'exécution de la compensation de mouvement sur un grand nombre de cœurs parallèles. L'approche parallèle est basée sur le traitement des groupes de lignes de macroblocks indépendants en parallèle. L'algorithme parallèle proposé montre une évolutivité supérieure à l'approche des composantes de couleur décrite dans cette section. Ainsi, une bonne accélération et économie d'énergie sont atteintes sur des processeurs multicœurs avec plus de 8 cœurs.

8.5 Décodage de Macroblocks en Parallèle

De nombreuses implémentations parallèles du codec H.264 existent allant du décodage en parallèle des macroblocks (implémentations *grain fin*) jusqu'au décodage en parallèle des groupes d'images (implémentations *grain gros*). Un macroblock est un composant de 16x16 pixel d'une image dans une séquence vidéo. En outre, un macroblock peut également être divisé en sous-blocks de taille plus petite. Le décodage en parallèle des macroblocks est hautement évolutif pour la raison que de nombreux macroblocks indépendants peuvent être traités en parallèle. Toutefois, des dépendances et des synchronisations sont créées à la suite de la communication de la mémoire et de la synchronisation d'exécution entre les macroblocks. D'autre part, le décodage parallèle des groupes d'images nécessite une grande capacité mémoire en particulier pour la haute définition des séquences vidéo. Ils ont une extensibilité inférieure à celle des macroblocks à cause du petit nombre de groupes de frames qui peuvent être décodées en parallèle. Dans notre approche, nous traitons les lignes de macroblocks indépendants en parallèle en utilisant un nouvel algorithme qui élimine les dépendances entre les macroblocks et qui minimise le surcoût de la synchronisation. Ce niveau d'exécution en parallèle peut être considéré entre les approches parallèles à gros-grain et à grain-fin, ainsi, offrant un équilibre entre la disponibilité et l'extensibilité.

Notre principale contribution de cette recherche est la conception et l'implémentation d'un nouvel algorithme de traitement parallèle des lignes de macroblocks du décodeur H.264. En plus, un algorithme de détection de dépendance de données qui isole les macroblocks d'intra-prédiction (I-MB) est développé. Des expérimentations sont menées par l'exécution de notre décodeur parallèle évolutive sur une plateforme de développement Cuda [43] avec un processeur ARM Cortex-A9 contenant 4 noyaux [6]. Des statistiques du temps d'exécution réel et de la consommation d'énergie sont collectées par l'exécution de l'application sur une plateforme réelle. Pour les résolutions HD et Full-HD, les séquences vidéo de référence ont atteint leur débit maximum en utilisant 4 threads sur 4 cœurs avec une accélération de 3.3x pour la compensation de mouvement et d'une accélération globale de 2,3x en termes de temps d'exécution et avec un pourcentage de 63% d'économie d'énergie. En plus, l'algorithme parallèle a une accélération théorique très important qui est

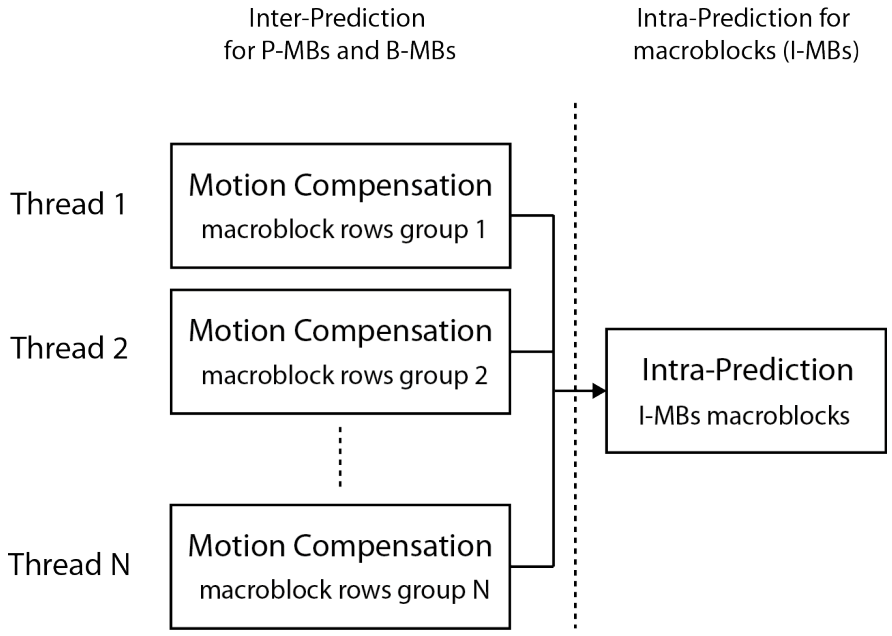

Figure 8.8: Décodage des lignes de macroblocks en parallèle

applicable sur les many-cores et les processeurs vecteurs. [10, 11]

8.5.1 Compensation des Mouvements en Parallèle

Dans notre recherche, nous avons modifié l'implémentation référence de H.264, JM [61] code source du décodeur H.264, afin de décoder les lignes de macroblocks en parallèle à l'aide de la bibliothèque pthread en langage de programmation C.

Un thread est créé pour chaque groupe de lignes de macroblocks. Chaque thread est associé à un noyau. Le nombre de threads est spécifié par l'utilisateur ou par l'application. Si le nombre de threads est plus grand que le nombre de cores, alors le scheduler n'attribue plus qu'un thread pour chaque noyau. Comme le montre la figure 8.8, chaque thread gère l'étape de compensation de mouvement pour un groupe de lignes de macroblocks. Tous les threads doivent remplir leur tâche avant de passer à la phase suivante qui est l'intra-prédiction pour les I-MBs.

Le nombre maximal de blocs de décodage en parallèle est égal au nombre de lignes de macroblocks. Ce niveau de décodage en parallèle peut être considéré entre les approches à grains gros et à grains fins. D'autres approches traitent plusieurs slices ou frames en parallèle. Ces méthodes de haut niveau, comme [20]

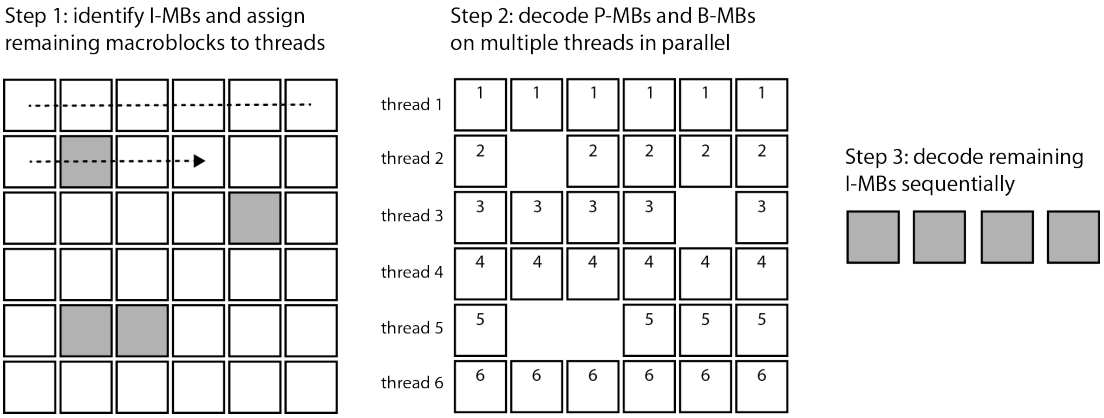

Figure 8.9: Algorithme parallèle de la compensation des mouvements

[28] [42] [57], necessitent une utilisation élevée de la mémoire afin de décoder plusieurs frames en parallèle en raison de la taille nécessaire pour stocker et transférer des données de plusieurs frames. Les approches de décodage à grains fins sont au niveau des macroblocks ou des blocs à l'intérieur d'un macroblock. Ces méthodes de bas niveau, comme [14] [67] [72], provoquent un énorme surcoût de synchronisation affectant profondément l'accélération à cause du grand nombre de macroblocks dans chaque image. L'équilibre entre les deux approches se reflète également sur les surcoûts de synchronisation et les exigences de communication de données.

Notre approche vise à bénéficier de l'équilibre entre avantages et des inconvénients. Les lignes de macroblocks nécessitent moins de mémoire que d'une image et plus de mémoire qu'un macroblock. En fait, notre approche est évolutive au niveau de macroblock. Cette granularité va créer une énorme charge de parallélisme des architectures multicœurs actuels. D'autre part, le nombre de lignes de macroblocks est beaucoup moins que le nombre total de macroblocks. Par exemple, en résolution HD (1280 x 720), chaque image a 3600 macroblocks, 80 MB et 45 MB horizontales verticales. Ainsi, le nombre de rangées de macroblocks est inférieur d'un facteur de 80, le nombre total de macroblocks. En conséquence, les surcoûts pour la synchronisation et la communication entre les noyaux sont également réduites d'un facteur de 80.

8.5.2 Algorithme de Vérification des Dépendances entre les Macroblocks

L'algorithme de vérification de dépendance des macroblocks est relativement simple. Figure 8.9 montre une illustration simple de l'algorithme. Étant donné une liste contenant tous les macroblocks dans une image, une boucle qui parcourt tous les macroblocks signale tous les macroblocks intra-prédiction (I-MB) et attribue à chaque macroblock restant à un groupe spécifique pour un noyau disponible. Ensuite, ces groupes de macroblocks sont décodés en parallèle. Lorsque tous les groupes de macroblocks sont traitées, une boucle parcourt tous les I-MBs qui ont été signalées au départ. Tous les macroblocks dans la liste I-MB sont décodés séquentiellement. Les I-MBs peuvent être traitées en parallèle si elles ne sont pas des voisins. Ce qui signifie qu'ils n'ont pas des dépendances entre elles. Le nombre de I-MBs dans P-Frames et B-Frames est 2% en moyenne. Si nous attribuons un macroblock à un noyau différent, la charge de travail n'est pas très importante et les surcoûts de la synchronisation seront également ajoutés. Alors on exécute les I-MBs d'une manière séquentielle pour des raisons de simplicité et moins de surcoûts de communication. Le résultat de cette étape est entièrement conforme à la norme H.264 [26], ce qui signifie que la sortie est exactement la même lorsque l'exécution séquentielle est effectuée. Les macroblocks décodés sont ensuite soumis au filtre de déblocage afin de rendre les bords entre les macroblocks lisses et presque invisibles.

Le pire cas asymptotique de la complexité de l'algorithme parallèle proposé reste pratiquement le même que l'algorithme séquentiel. Tous les macroblocks sont traités une seule fois, ce qui est similaire à l'exécution séquentielle. Une itération supplémentaire avec une surcharge de fonctionnement constant est ajoutée avant l'exécution parallèle. Au cours de ce processus, les I-MBs sont identifiés et leurs pointeurs sont ajoutés à une liste pour un traitement ultérieur dans une étape suivante. Le coût d'exécution de cette boucle supplémentaire est linéaire et il est considéré comme négligeable pour la complexité totale de l'algorithme. Après le décodage en parallèle des macroblocks indépendants, les macroblocks restants qui sont dans la liste précédemment décrite sont traitées séquentiellement et dans l'ordre. Ainsi, dans le pire des cas la complexité de l'algorithme en comparaison

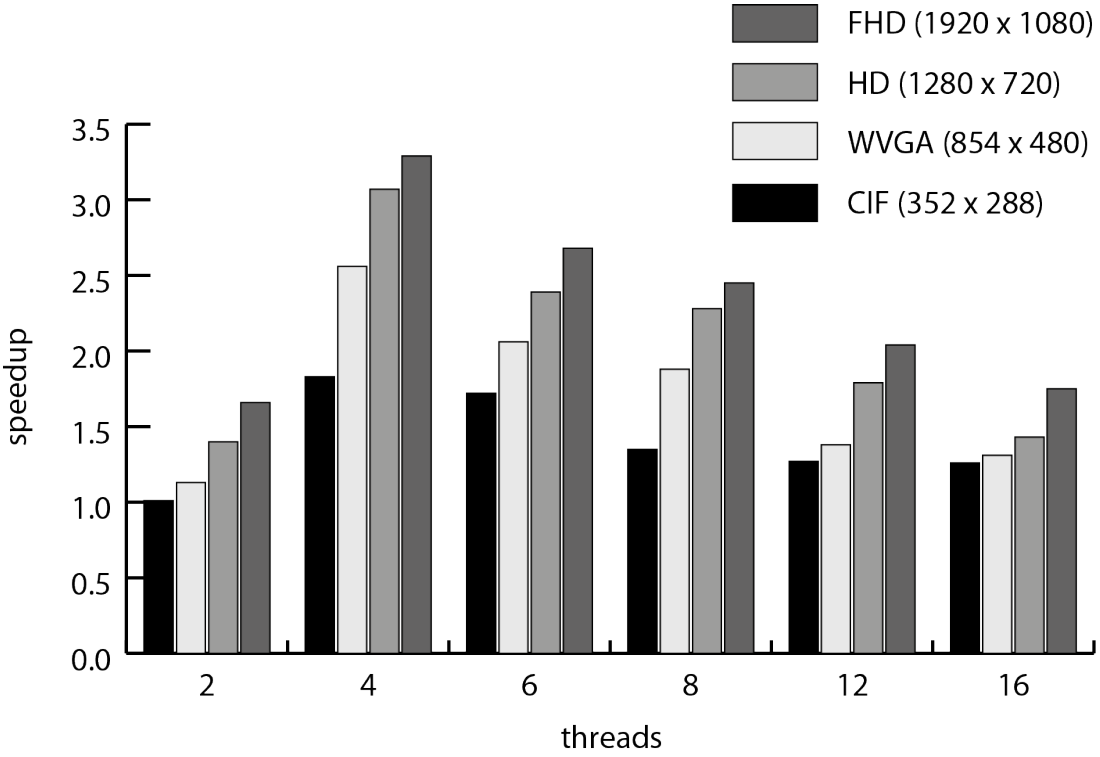

Figure 8.10: Accélération de l'exécution parallèle de l'étape Motion Compensation sur la plateforme ARM Cortex-A9 avec 4 cores

avec l'algorithme séquentiel d'origine reste le même avec ou sans le gain de calcul parallèle.

8.5.3 Résultats de Compensation des Mouvements en Parallèle

Les expérimentations sont préformées sur les séquences vidéo. Le nombre de rangées parallèles de macroblocks augmente avec la résolution. Ainsi, des résolutions élevées s'adaptent mieux que les basses résolutions avec le nombre de noyau en raison du nombre plus élevé de macroblocks dans chaque image. Des expérimentations sont effectuées à l'aide de 2, 4, 6, 8, 12, et 16 threads sur un processeur ARM Cortex-A9 avec 4 cœurs [6]. Figure 8.10 montre l'accélération moyenne de la phase de compensation du mouvement (motion compensation) pour chaque résolution pour un nombre différent de thread. Pour la résolution

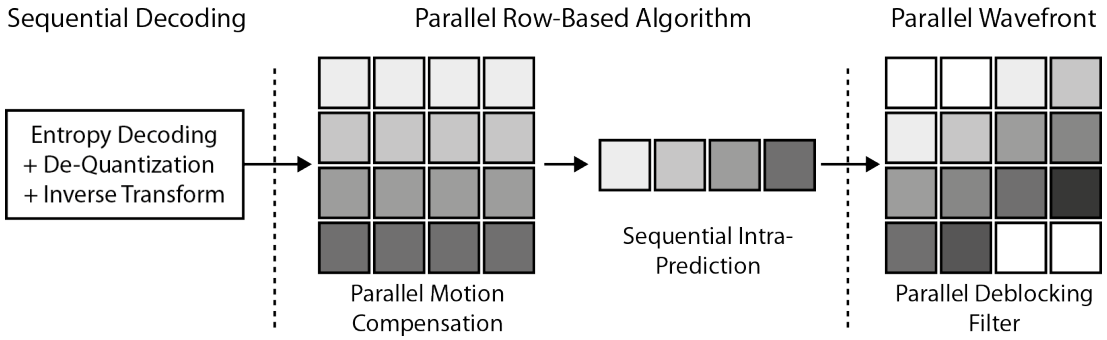

Figure 8.11: Exécution parallèle globale du H.264/AVC

CIF, l'accélération maximale de 1,8 est atteinte à l'aide de 4 threads. L'accélération diminue quand le nombre de threads augmente à cause du gros surcoût de communication de données. Les séquences vidéo HD et FHD ont une accélération supérieure à 3.3 avec 4 threads où chaque thread est attribué à un noyau différent. La meilleure accélération au nombre de rapport de threads est lorsque quatre threads sont utilisés. Le rapport d'accélération de nombre de threads de résolution haute définition est d'environ 0.8 quand quatre threads sont utilisés. Le doublement du nombre de threads baisse le ratio à 0.6 qui ne peut être considéré efficace comme prévu lors de l'exécution d'une application parallèle sur un processeur multicœur. L'utilisation d'un certain nombre de threads qui est plus que le nombre de noyaux provoque le scheduler d'affecter plus qu'un thread pour un noyau. Par conséquent, la commutation de contexte n'augmente pas l'efficacité de l'application, comme indiqué dans les résultats.

Les résultats pour les hautes résolutions ont en général de meilleures accélérations. Ceci est principalement dû à une plus grande charge de travail pour chaque noyau. Un plus grand volume de travail réduit l'impact du transfert de données et la synchronisation entre les noyaux. L'une des raisons est la diminution des dépendances entre les macroblocks en cours de traitement sur les différents noyaux. Une autre raison est les surcoûts du transfert des données qui est nécessaire pour l'envoi des données aux différents noyaux. La synchronisation ajoute également des charges indépendantes de la résolution des vidéos. Ainsi, l'accélération sera beaucoup plus efficace pour des résolutions plus élevées.

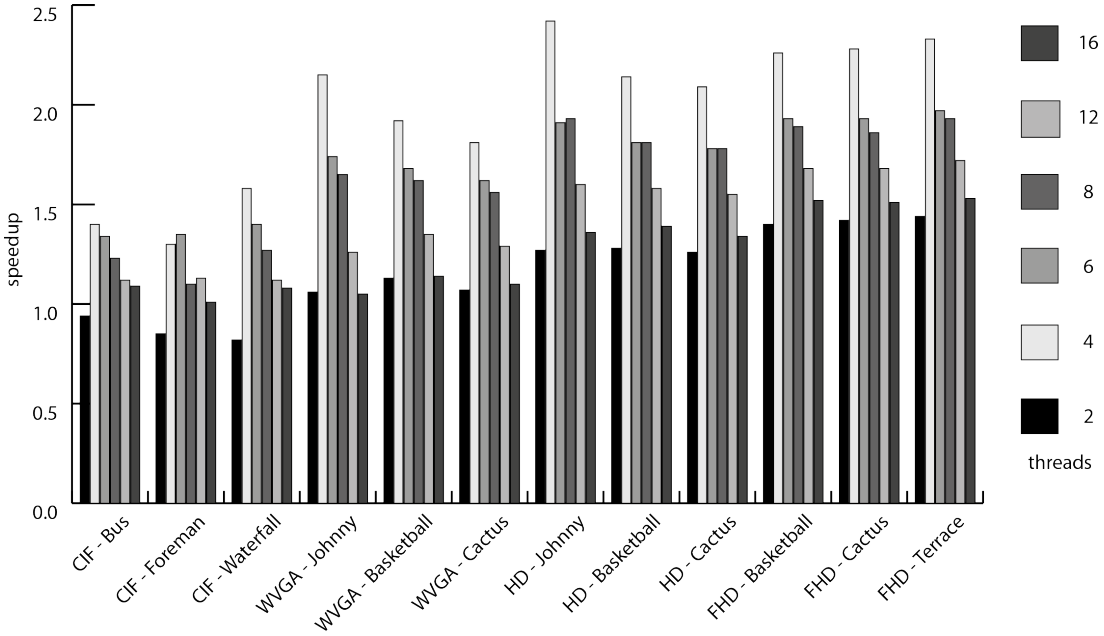

Figure 8.12: Accélération totale du décodeur H.264 sur ARM Cortex-A9 avec 4 cores

8.5.4 Résultats pour l'Exécution Complète

Notre objectif principal est d'optimiser toutes les étapes du décodeur H.264. Nous appliquons des techniques parallèles pour la compensation de mouvement et les étapes de filtre de déblocage. D'autre part, l'étape de décodage d'entropie est principalement séquentielle. Ainsi, les techniques parallèles pour le décodeur entropique sont très difficiles à appliquer ou parfois impossible en raison de ses spécifications. Figure 8.11 dépeint les étapes avec des algorithmes parallèles du décodeur H.264. Nous recueillons le temps d'exécution et les statistiques de consommation d'énergie pour l'implémentation parallèle proposée du H.264. Les fractions des différentes étapes varient entre les différentes séquences vidéo. En conséquence, le rendement global est considéré comme la moyenne totale de toutes les accélérations en fonction de la moyenne de chaque phase.

La figure 8.12 illustre les accélérations globaux obtenus par l'exécution complète du décodeur avec les techniques d'optimisation décrites. Les accélérations totales de 1.4, 2.0, 2.2, et 2.3 sont obtenus en utilisant 4 threads sur 4 cœurs pour les résolutions CIF, WVGA, HD, et FHD respectivement. L'exécution séquentielle

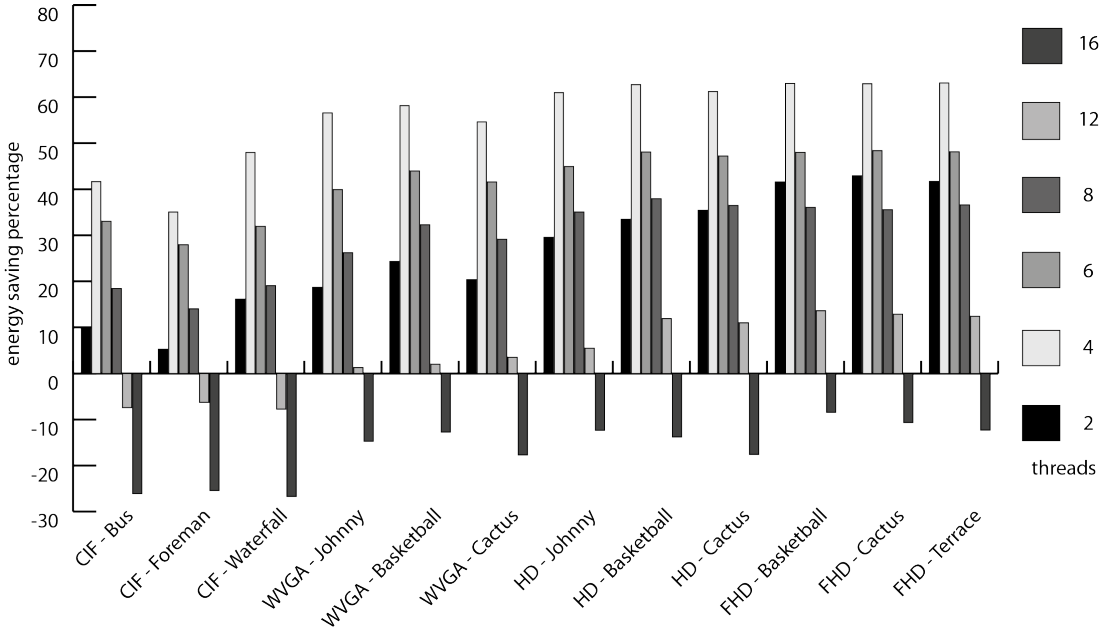

Figure 8.13: Économies Globales de Consommation d'Energie

de l'étape de décodage entropique qui est d'environ 14% à 19% du décodage globale réduit l'échelle de l'accélération globale. Cette étape peut être améliorée par l'implémentation d'une version matérielle du décodeur entropique. Les résolutions FHD ont la plus forte accélération en raison de leurs grandes tailles d'images. Toutes les accélérations maximales sont atteintes en utilisant 4 threads sur 4 cœurs. Ceci est principalement dû à l'absence de changement de contexte où chaque thread est associé à un noyau. Pour plus de 4 threads, le système d'exploitation doit affecter plus d'un thread à un contexte provoquant de base de commutation et, par conséquent, plus les surcoûts et le temps mort sera ajouté à l'exécution globale. Seules des séquences vidéo CIF ont accélérations moins de 2 lorsque 4 threads sont mappés sur 4 cœurs. Le taux d'accélération au nombre de noyaux est donc d'environ 0.6. Cela nous amène à conclure que le bénéfice des hautes résolutions est plus élevé que des processeurs multicœurs en comparaison avec des résolutions inférieures. Donc les résolutions Full-HD ont la meilleure accélération avec un nombre élevé de noyaux.

Les mesures de l'énergie pour l'exécution complète sont affichées dans la figure 8.13. Les meilleurs résultats d'économie d'énergie correspondent aux résolutions

FHD en utilisant 4 threads qui atteignent 63%. Ces résultats sont également évalués pour l'exécution complète du décodeur optimisé. Pour 12 et 16 threads, la consommation d'énergie va augmenter par rapport à l'exécution séquentielle. Ainsi, nous concluons que les économies d'énergie ne s'adaptent pas linéairement avec le nombre de threads ou noyaux.

8.6 Conclusion

H.264 est largement adopté dans les applications multimédias sur les systèmes embarqués. La grande complexité imposée par le décodeur H.264 nécessite à accroître l'efficacité et à réduire la consommation électrique.

Nous avons proposé et évalué un algorithme parallèle pour le décodeur H.264. Les composantes de couleur, luminance et chrominance, pour les étapes de compensation de mouvement sont traitées en parallèle fournissant des hauts potentiels et réalistes pour le décodage vidéo sur les processeurs dual et quad cores. L'exécution parallèle du décodeur H.264 est améliorée d'environ 18%. L'accélération atteint 32% avec un pipeline et une économie d'énergie de 24%.

En outre, nous avons présenté une technique parallèle innovante pour le standard vidéo du décodeur H.264. Notre approche décode des groupes des lignes de macroblocks en parallèle avec un algorithme qui détecte les dépendances sur la volée basé sur l'isolement des macroblocks intra-prédiction (I-MBs). Les séquences vidéo aux définitions hautes et basses sont utilisées dans nos expérimentations. L'accélération la plus efficace avec la meilleur proportion au nombre de noyaux parallèle est de 3.3 en utilisant 4 threads sur 4 cœurs. Une implémentation parallèle à base des macroblocks pour le filtre de déblocage est également implémentée. Une accélération globale de 2.3 est atteinte pour l'optimisation parallèle complète du standard H.264. Notre décodeur optimisé est testé sur un vrai appareil avec un processeur ARM Cortex-A9 avec 4 cœurs. En plus, l'algorithme parallèle proposé est testé sur un simulateur de multicœurs afin d'explorer à l'évolutivité de notre algorithme sur des multiprocesseurs jusqu'à 32 cœurs.

Le travail dans cette recherche présente des solutions à la grande complexité du décodeur H.264 en utilisant le calcul parallèle. Ces algorithmes peuvent être appliqués à la plupart des standards de compression vidéo à base de macroblocks.

D'autres expérimentations et des améliorations doivent être effectuées afin d'appliquer nos algorithmes parallèles dans des produits commerciaux. Notre intention est de continuer notre recherche afin d'augmenter l'efficacité et réduire la consommation d'énergie des standards de codage vidéo récents.

References

[1] *J. VLSI Signal Process. Syst.*, 19(2), 1998. ISSN 0922-5773. 14

[2] G. S. Almasi and A. Gottlieb. *Highly parallel computing*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1989. ISBN 0-8053-0177-1. 6, 121

[3] Advanced Micro Devices (AMD). Accelerated parallel processing (app) sdk. http://developer.amd.com/tools-and-sdks/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/, 2013. 11, 101

[4] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities, reprinted from the afips conference proceedings, vol. 30 (atlantic city, n.j., apr. 18 x2013;20), afips press, reston, va., 1967, pp. 483 x2013;485, when dr. amdahl was at international business machines corporation, sunnyvale, california. *Solid-State Circuits Society Newsletter, IEEE*, 12(3):19–20, 2007. ISSN 1098-4232. doi: 10.1109/N-SSC.2007.4785615. 6, 21, 22, 24, 122

[5] Apple. iphone. http://www.apple.com/iphone/, 2013. 2, 118

[6] ARM-ltd. Cortex-a9 processor. http://www.arm.com/products/processors/cortex-a/cortex-a9.php, 2012. 2, 9, 13, 75, 88, 89, 105, 118, 122, 131, 135

[7] ARM-ltd. Cortex-a15 processor. http://www.arm.com/products/processors/cortex-a/cortex-a15.php, 2013. 2, 118

[8] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen,

John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009. ISSN 0001-0782. doi: 10.1145/1562764.1562783. URL http://doi.acm.org/10.1145/1562764.1562783. 6, 121

[9] Elias Baaklini, Hassan Sbeity, Smail Niar, and Nouhad Amaneddine. H.264 color components video decoding parallelization on multi-core processors. In *Proceedings of the 2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, DSD '10, pages 785–790, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4171-6. doi: 10.1109/DSD.2010.76. URL http://dx.doi.org/10.1109/DSD.2010.76. 57, 125

[10] Elias Baaklini, Hassan Sbeity, and Smail Niar. H.264 macroblock line level parallel video decoding on embedded multicore processors. In *Proceedings of the 2012 15th Euromicro Conference on Digital System Design*, DSD '12, pages 902–906, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4798-5. doi: 10.1109/DSD.2012.67. URL http://dx.doi.org/10.1109/DSD.2012.67. 75, 132

[11] Elias Baaklini, Hassan Sbeity, and Smail Niar. H.264 parallel optimization on graphics processors. In *The Fifth International Conferences on Advances in Multimedia*, MMEDIA 2013, pages 109–114, Delaware, USA, 2013. IARIA. ISBN 978-1-61208-265-3. URL http://www.thinkmind.org/index.php?view=article&articleid=mmedia_2013_5_30_40117. 75, 132

[12] David Brash. The arm architecture version 6 (armv6). Architecture Program Manager, ARM Ltd, 2002. 64

[13] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001. ISBN 1-55860-671-8, 9781558606715. 26

[14] Jike Chong, N. Satish, B. Catanzaro, K. Ravindran, and K. Keutzer. Efficient parallelization of h.264 decoding with macro block level scheduling. In

Multimedia and Expo, 2007 IEEE International Conference on, pages 1874 –1877, july 2007. doi: 10.1109/ICME.2007.4285040. 52, 60, 79, 133

[15] OAR Corporation. Rtems real-time operating system. http://www.rtems.com, 2010. 64

[16] R. Cucchiara, M. Piccardi, and A. Prati. Neighbor cache prefetching for multimedia image and video processing. *Trans. Multi.*, 6(4):539–552, August 2004. ISSN 1520-9210. doi: 10.1109/TMM.2004.830806. URL http://dx.doi.org/10.1109/TMM.2004.830806. 112

[17] FFmpeg. Ffmpeg project. http://www.ffmpeg.org/, 2012. 52, 56, 57, 66, 67, 70

[18] Laurie J. Flynn. Intel halts development of 2 new microprocessors. http://www.nytimes.com/2004/05/08/business/intel-halts-development-of-2-new-microprocessors.html, 2004. 7

[19] T.R. Gardos. H.263+: the new itu-t recommendation for video coding at low bit rates. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, volume 6, pages 3793–3796 vol.6, 1998. doi: 10.1109/ICASSP.1998.679710. 33, 35, 38

[20] A. Gurhanli, C.C.-P. Chen, and Shih-Hao Hung. Gop-level parallelization of the h.264 decoder without a start-code scanner. In *Signal Processing Systems (ICSPS), 2010 2nd International Conference on*, volume 3, pages V3–627–V3–630, 2010. doi: 10.1109/ICSPS.2010.5555416. 51, 79, 132

[21] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011. ISBN 012383872X, 9780123838728. 6, 7, 8, 9, 10, 11, 13, 17, 19, 20, 23, 111, 121

[22] M. Horowitz, A. Joch, F. Kossentini, and A. Hallapuro. H.264/avc baseline profile decoder complexity analysis. *Circuits and Systems for Video Technology, IEEE Transactions on*, 13(7):704 – 716, july 2003. ISSN 1051-8215. doi: 10.1109/TCSVT.2003.814967. 52

[23] IDG-Consumer&SMB. Pcworld magazine. http://www.pcworld.com/, 2012. 28, 123

[24] Intel. Intel core processor family. http://www.intel.com/, 2011. 9, 13, 122

[25] ISO. Iso c standard 1999. Technical report, 1999. URL http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf. ISO/IEC 9899:1999 draft. 101

[26] ITU-T and ISO/IEC(2012). Advanced video coding for generic audiovisual services. *ITU-T Rec. H.264*, January 2012. 3, 23, 28, 33, 36, 39, 51, 57, 59, 60, 83, 119, 123, 126, 134

[27] Mike Johnson. *Superscalar multiprocessor design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991. ISBN 0-13-875634-1. 9

[28] C. S. Kannangara, I. E. G. Richardson, M. Bystrom, J. Solera, Y. Zhao, and A. Maclennan. Complexity reduction of h.264 using lagrange optimization methods. In *IEE VIE 2005, (Glasgow,UK*, 2005. 52, 79, 133

[29] Marta Karczewicz and Ragip Kurceren. The sp- and si-frames design for h.264/avc. *IEEE Trans. Circuits Syst. Video Techn.*, 13(7):637–644, 2003. URL http://dblp.uni-trier.de/db/journals/tcsv/tcsv13.html. 43

[30] Khronos. Opencl: The open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl, 2012. 11, 25, 26, 99, 100

[31] Micrel lab. Mparm simulator. http://www-micrel.deis.unibo.it/sitonew/research/mparm.html, 2005. 63, 64, 129

[32] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. When prefetching works, when it doesn’t, and why. *ACM Trans. Archit. Code Optim.*, 9(1): 2:1–2:29, March 2012. ISSN 1544-3566. doi: 10.1145/2133382.2133384. URL http://doi.acm.org/10.1145/2133382.2133384. 111

[33] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In

Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 42, pages 469–480, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-798-1. doi: 10.1145/1669112.1669172. URL http://doi.acm.org/10.1145/1669112.1669172. 67, 69

[34] Peter List, Anthony Joch, Jani Lainema, Gisle Bjøntegaard, and Marta Karczewicz. Adaptive deblocking filter. *IEEE Trans. Circuits Syst. Video Techn.*, 13(7):614–619, 2003. 51

[35] Henrique S. Malvar, Antti Hallapuro, Marta Karczewicz, and Louis Kerofsky. Low-complexity transform and quantization in h.264/avc. *IEEE Trans. Circuits Syst. Video Technol*, pages 598–603, 2003. 50

[36] Detlev Marpe, Thomas Wiegand, and Stephen Gordon. H.264/mpeg4-avc fidelity range extensions: tools, profiles, performance, and application areas. In *ICIP*, pages 593–596. IEEE, 2005. URL http://dblp.uni-trier.de/db/conf/icip/icip2005-1.html. 39, 50

[37] Oleg Maslennikov. Systematic generation of executing programs for processor elements in parallel asic or fpga-based systems and their transformation into vhdl-descriptions of processor element control units. In *Proceedings of the th International Conference on Parallel Processing and Applied Mathematics-Revised Papers*, PPAM '01, pages 272–279, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-43792-4. URL http://dl.acm.org/citation.cfm?id=645813.668540. 14

[38] Cor Meenderinck, Arnaldo Azevedo, Ben Juurlink, Mauricio Alvarez Mesa, and Alex Ramirez. Parallel scalability of video decoders. *J. Signal Process. Syst.*, 57(2):173–194, November 2009. ISSN 1939-8018. doi: 10.1007/s11265-008-0256-9. 52, 91, 92

[39] Mauricio Alvarez Mesa, Alex Ramirez, Arnaldo Azevedo, Cor Meenderinck, Ben Juurlink, and Mateo Valero. Scalability of macroblock-level parallelism for h.264 decoding. In *Proceedings of the 2009 15th International Conference on Parallel and Distributed Systems*, ICPADS '09, pages 236–243, Washing-

ton, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3900-3. doi: 10.1109/ICPADS.2009.124. 52, 91

[40] Gordon E. Moore. In Mark D. Hill, Norman P. Jouppi, and Gurindar S. Sohi, editors, *Readings in computer architecture*, chapter Cramming more components onto integrated circuits, pages 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000. ISBN 1-55860-539-8. URL http://dl.acm.org/citation.cfm?id=333067.333074. 7

[41] Kyle J. Nesbit and James E. Smith. Data cache prefetching using a global history buffer. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, HPCA '04, pages 96–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2053-7. doi: 10.1109/HPCA.2004.10030. URL http://dx.doi.org/10.1109/HPCA.2004.10030. 112

[42] K. Nishihara, A. Hatabu, and T. Moriyoshi. Parallelization of h.264 video decoder for embedded multicore processor. In *Multimedia and Expo, 2008 IEEE International Conference on*, pages 329 –332, 23 2008-april 26 2008. doi: 10.1109/ICME.2008.4607438. 52, 79, 133

[43] nVIDIA. The cuda development kit from seco. http://www.nvidia.com/object/seco-dev-kit.html, 2012. 11, 75, 88, 131

[44] nVIDIA Developer Zone. Cuda documents. http://docs.nvidia.com/cuda/, 2013. 25, 99

[45] IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002). Ieee standard vhdl language reference manual. pages c1–626, 2009. doi: 10.1109/IEEESTD.2009.4772740. 14

[46] VideoLAN Organization. x264 encoder. http://www.videolan.org/developers/x264.html, 2013. 81

[47] Peter Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011. ISBN 9780123742605. 21

[48] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2008. ISBN 0123744938, 9780123744937. 6, 9, 17, 18, 19, 20, 23, 24, 25, 121

[49] B. Pieters, C.-F.J. Hollemeersch, J. De Cock, P. Lambert, W. De Neve, and R. Van De Walle. Parallel deblocking filtering in mpeg-4 avc/h.264 on massively parallel architectures. *Circuits and Systems for Video Technology, IEEE Transactions on*, 21(1):96–100, 2011. ISSN 1051-8215. doi: 10.1109/TCSVT.2011.2105553. 53

[50] Iain E. Richardson. *Video Codec Design: Developing Image and Video Compression Systems*. John Wiley & Sons, Inc., New York, NY, USA, 2002. ISBN 0471485535. 31, 37, 50

[51] Michael Roitzsch. Slice-balancing h.264 video encoding for improved scalability of multicore decoding. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, EMSOFT '07, pages 269–278, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-825-1. doi: 10.1145/1289927.1289969. URL http://doi.acm.org/10.1145/1289927.1289969. 60

[52] Priscila Tiemi Maeda Saito, Denis Fernando Wolf, Bruno Alexandre Mendonça, Kalinka R. L. J. C. Branco, and Ricardo José Sabatine. A parallel approach for mobile robotic self-localization. In *Proceedings of the 2009 Fourth International Conference on Computer Sciences and Convergence Information Technology*, ICCIT '09, pages 762–767, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3896-9. doi: 10.1109/ICCIT.2009.284. URL http://dx.doi.org/10.1109/ICCIT.2009.284. 25

[53] Samsung. Samsung smartphones. http://www.samsung.com/uk/consumer/mobile-phones/smartphones/, 2013. 2, 118

[54] R. Schafer and T. Sikora. Digital video coding standards and their role in

video communications. *Proceedings of the IEEE*, 83(6):907–924, 1995. ISSN 0018-9219. doi: 10.1109/5.387092. 37

[55] Florian H. Seitner, Michael Bleyer, Margrit Gelautz, and Ralf M. Beuschel. Evaluation of data-parallel h.264 decoding approaches for strongly resource-restricted architectures. *Multimedia Tools Appl.*, 53(2):431–457, June 2011. ISSN 1380-7501. doi: 10.1007/s11042-010-0501-7. 84, 91

[56] Y. Shimokawa, Y. Fuwa, and N. Aramaki. A parallel asic vlsi neurocomputer for a large number of neurons and billion connections per second speed. In *Neural Networks, 1991. 1991 IEEE International Joint Conference on*, pages 2162–2167 vol.3, 1991. doi: 10.1109/IJCNN.1991.170708. 14

[57] Kue-Hwan Sihn, Hyunki Baik, Jong-Tae Kim, Sehyun Bae, and Hyo Jung Song. Novel approaches to parallel h.264 decoder on symmetric multicore systems. In *Proceedings of the 2009 IEEE International Conference on Acoustics, Speech and Signal Processing*, ICASSP '09, pages 2017–2020, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-2353-8. doi: 10.1109/ICASSP.2009.4960009. 53, 79, 133

[58] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 8th edition, 2008. ISBN 0470128720. 24, 25

[59] William Stallings. *Computer Organization and Architecture: Designing for Performance*. Prentice Hall Press, Upper Saddle River, NJ, USA, 8th edition, 2009. ISBN 9780136073734. 8

[60] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73, May 2010. ISSN 0740-7475. doi: 10.1109/MCSE.2010.69. URL http://dx.doi.org/10.1109/MCSE.2010.69. 26

[61] K. Suehring. H.264 reference software. http://bs.hhi.de/ suehring/tml/, 2012. 52, 56, 57, 63, 64, 79, 82, 129, 132

[62] Gary J. Sullivan, Pankaj Topiwala, and Ajay Luthra. The h.264/avc advanced video coding standard: Overview and introduction to the fidelity range extensions. In *SPIE conference on Applications of Digital Image Processing XXVII*, pages 454–474, 2004. 39, 50

[63] G.J. Sullivan, J. Ohm, Woo-Jin Han, T. Wiegand, and T. Wiegand. Overview of the high efficiency video coding (hevc) standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 22(12):1649–1668, 2012. ISSN 1051-8215. doi: 10.1109/TCSVT.2012.2221191. 3, 39, 119

[64] Agilent Technologies. High-resolution lxi digitizers. http://www.home.agilent.com/en/pd-1445167-pn-L4532A/, 2012. 89

[65] P. N. Tudor. Mpeg-2 video compression. *Electronics Communication Engineering Journal*, 7(6):257–264, 1995. ISSN 0954-0695. doi: 10.1049/ecej:19950606. 3

[66] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2sim: a simulation framework for cpu-gpu computing. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 335–344, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1182-3. doi: 10.1145/2370816.2370865. 66, 88, 97, 105, 106

[67] Erik VanDerTol, Egbert Jaspers, and Rob Gelderblom. Mapping of h.264 decoding on a multiprocessor architecture. In *Proc. SPIE Conf. on Image and Video Communications and Processing*, pages 707–718, 2003. 52, 60, 79, 133

[68] Sung-Wen Wang, Shu-Sian Yang, Hong-Ming Chen, Chia-Lin Yang, and Ja-Ling Wu. A multi-core architecture based parallel framework for h.264/avc deblocking filters. *J. Signal Process. Syst.*, 57(2):195–211, November 2009. ISSN 1939-8018. doi: 10.1007/s11265-008-0321-4. 53

[69] Weixun Wang, Prabhat Mishra, and Sanjay Ranka. Dynamic cache reconfiguration and partitioning for energy optimization in real-time multi-core systems. In *Proceedings of the 48th Design Automation Conference*,

DAC '11, pages 948–953, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0636-2. doi: 10.1145/2024724.2024935. URL http://doi.acm.org/10.1145/2024724.2024935. 112

[70] Stephen Westland, Huw Owens, Vien Cheung, and Iain Paterson-Stephens. Model of luminance contrast-sensitivity function for application to image assessment. *Color Research & Application*, 31(4):315–319, 2006. ISSN 1520-6378. doi: 10.1002/col.20230. URL http://dx.doi.org/10.1002/col.20230. 31

[71] YouTube. Youtube advanced encoding settings. https://support.google.com/youtube/answer/1722171, 2013. 3

[72] Zhuo Zhao and Ping Liang. Data partition for wavefront parallelization of h.264 video encoder. In *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on*, pages 4 pp.–2672, 2006. doi: 10.1109/ISCAS.2006.1693173. 52, 77, 79, 87, 91, 133