GENERAL PURPOSE COMPUTING IN GPU - A WATERMARKING CASE STUDY

Anthony Hanson

Thesis Prepared for the Degree of

MASTER OF SCIENCE

UNIVERSITY OF NORTH TEXAS

August 2014

APPROVED:

Saraju Mohanty, Major Professor
Elias Kougianos, Co-Major Professor
Mahadevan Gomathisankaran, Committee Member
Barret Bryant, Chair of the Department of Computer Science and Engineering
Costas Tsatsoulis, Dean of the College of Engineering
Mark Wardell, Dean of the Toulouse Graduate School

Hanson, Anthony. *General Purpose Computing in GPU - A Watermarking Case Study*. Master of Science (Computer Science), August 2014, 78 pp., 8 tables, 15 figures, references, 42 titles.

The purpose of this project is to explore the GPU for general purpose computing. The GPU is a massively parallel computing device that has a high-throughput, exhibits high arithmetic intensity, has a large market presence, and with the increasing computation power being added to it each year through innovations, the GPU is a perfect candidate to complement the CPU in performing computations. The GPU follows the single instruction multiple data (SIMD) model for applying operations on its data. This model allows the GPU to be very useful for assisting the CPU in performing computations on data that is highly parallel in nature. The compute unified device architecture (CUDA) is a parallel computing and programming platform for NVIDIA GPUs. The main focus of this project is to show the power, speed, and performance of a CUDA-enabled GPU for digital video watermark insertion in the H.264 video compression domain. Digital video watermarking in general is a highly computationally intensive process that is strongly dependent on the video compression format in place. The H.264/MPEG-4 AVC video compression format has high compression efficiency at the expense of having high computational complexity and leaving little room for an imperceptible watermark to be inserted. Employing a human visual model to limit distortion and degradation of visual quality introduced by the watermark is a good choice for designing a video watermarking algorithm though this does introduce more computational complexity to the algorithm. Research is being conducted into how the CPU-GPU execution of the digital watermark application can boost the speed of the applications several times compared to running the application on a standalone CPU using NVIDIA visual profiler to optimize the application.

Copyright 2014

by

Anthony Hanson

ACKNOWLEDGEMENTS

I would like to thank my wife, Yuting, for her support during the time it has took me to finalize this thesis.

I would like to thank my thesis advisor Dr. Saraju Mohanty as well for his support and guidance during my thesis.

I would like to thank Dr. Elias Kougianos and Dr. Mahadevan Gomathisankaran for serving on my committee.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS iii

LIST OF TABLES viii

LIST OF FIGURES ix

Chapters

1. INTRODUCTION 1

1.1. Overview 1

1.2. Heterogeneous CPU-GPU Computing Systems 2

1.2.1. Parallel Programming 3

1.3. Digital Watermark on H.264 Compressed Video 3

1.3.1. Parallel Processing of Digital Watermarking and H.264 Operations 5

1.4. Organization of This Thesis 6

2. RELATED PRIOR RESEARCH 8

2.1. General-purpose Computing on Graphics Processing Units 8

2.2. Digital Watermark Parallelization 10

2.2.1. Hardware Implementation 10

2.2.2. Software Implementation 11

2.2.2.1. Digital Image Watermark 11

2.2.2.2. Digital Video Watermarking 12

2.3. Video Coding on GPU 13

2.4. Novel Contributions of This Thesis 15

3. GENERAL PURPOSE COMPUTING PROBLEM ON GPU 17
3.1. Overview 17
3.2. Modern CPU Architecture 18
3.3. Modern GPU Architecture 20
3.4. Heterogeneous CPU-GPU Architecture 22
3.5. General-Purpose Computing on Graphics Processor Units 23
3.5.1. Overview 23
3.5.2. Concepts and Methods 23
3.5.3. Bottleneck 25
4. PARALLEL COMPUTING AND PROGRAMMING ON A CUDA-ENABLED GPU 27
4.1. Overview 27
4.2. Languages and Libraries 27
4.2.1. CUDA 28
4.2.1.1. Scalability 28
4.2.2. OpenACC 29
4.2.3. C++ AMP 31
4.2.4. OpenCL 31
4.3. Programming Model 32
4.4. Applications 34
5. DIGITAL VIDEO WATERMARK 35
5.1. Introduction 35
5.2. Watermark Framework 36
5.3. Security and Attacks 40

5.4. Video Watermarking 41
5.4.1. Overview..........41
5.4.2. H.264/MPEG-4 AVC..........42
5.4.2.1. Watermark Embedding and Extraction and Detection..........42
5.4.2.2. Security and Attacks..........43
5.5. Software Implementation..........44
5.6. Hardware Implementation 45
5.7. Parallelization 45
5.7.1. Software Implementation on a CUDA-Enabled GPU..........45
5.7.2. Hardware Implementation..........46
6. H.264/MPEG-4 AVC..........47
6.1. Overview..........47
6.2. Concepts of Video Coding..........48
6.2.1. Block Motion Compensation-Based Video Compression..........49
6.2.2. H.264 Syntax and Structure..........49
6.2.3. Profiles and Levels..........51
6.3. Encoding and Decoding..........52
6.3.1. Encoding Process..........52
6.3.2. Decoding Process..........54
6.4. Software Implementation..........55
6.5. Parallelization 57
6.5.1. Software Implementation on a CUDA-Enabled GPU..........57
6.5.2. Hardware Implementation..........57

7. THE PROPOSED VIDEO COMPRESSION AND WATERMARK IN GPU COMPUTING ENVIRONMENT 59

7.1. H.264 Video Compression Algorithm 59

7.2. H.264 Digital Video Watermark Embedding Algorithm 60

8. EXPERIMENTAL RESULTS 61

8.1. Experimental Setup 61

8.2. Experiment 63

8.3. Results 65

8.4. Discussion of Results 68

8.5. Comparison With Related Research 69

9. CONCLUSION AND FUTURE RESEARCH 70

9.1. Summary 70

9.2. Conclusions 71

9.3. Future Research 72

REFERENCE LIST 73

LIST OF TABLES

8.1. Test Hardware Setup 61

8.2. NVIDIA GeForce GT 555M Properties 61

8.3. Test System Software Setup 62

8.4. Test Software Setup 63

8.5. Video Properties 63

8.6. Watermark Properties 64

8.7. Frame Rate and Bitrate Calculated for CPU and GPU 65

8.8. Watermark Execution Time (s) 65

LIST OF FIGURES

3.1. Multi-core chip with inter-core bus 19

3.2. CUDA GPU streaming multiprocessor (SM) 21

3.3. CUDA GPU streaming processor (SP) 21

4.1. 2-Dimensional (2,2,1) grid organization 33

5.1. Human perception 36

5.2. Robustness 37

5.3. Watermark domain 38

5.4. Watermark extraction and detection process 38

5.5. Watermark encoder and decoder 39

6.1. The encoder stages 52

6.2. The decoder stages 54

8.1. A watermark logo used during the watermark insertion 64

8.2. Comparison of the original H.264 and watermarked H.264 frame in the short video 66

8.3. Comparison of the original H.264 and watermarked H.264 frame in the midsize video 67

8.4. Comparison of the original H.264 and watermarked H.264 frame in the long video 68

CHAPTER 1

INTRODUCTION

1.1. Overview

Today's computers are heterogeneous CPU-GPU computing systems. The CPU, at the moment, is a multi-core system designed to work on general computing of data that is serial in nature and maintain the execution speed of sequential programs. The GPU is a massive parallel computing platform supporting many-core (or many-threads) processor comprising of many arithmetic units to work on data that is parallel in nature. The GPU's primary role is to support 2D and 3D graphics, audio-, image-, and video-processing, video games, visual computing, and graphical user interfaces for today's complexly designed operating systems. Recent additions to the GPU such as programming instructions, multiple gigabytes of graphic double data rate (GDDR) DRAM supporting a wider bandwidth for higher throughput, more highly threaded streaming multiprocessors (SMs) and streaming processors (SPs), unified graphics and computing processors, programmable processors, and most importantly, a generic data-parallel computation model for non-graphics applications, have enabled non-graphics programmers to take advantage of the massive parallel computing power of the GPU for developing applications. However, there is one major limitation to the GPU and that is, the data has to be parallel in nature for the GPU to be effective at processing it in a highly efficient manner.

A digital watermark is the insertion of an image in a robust and visible or invisible manner inside an image or video for the purposes of protecting intellectual property (IP) and enforcing copyright protection. Various implementations have been proposed for images and video. Inserting a watermark into a video comes with its own challenges and computational complexities. What complicates this matter even more is the video compression format of the

video the digital watermark is being inserted into, such as the H.264/MPEG-4 AVC (or H.264 for short). Compression is needed for videos that involve storage and/or transmission such as movies and streaming video.

By their very nature, digital watermarking and video compression formats such as H.264 are computationally intensive tasks. Real-time encoding of video streams using a digital watermarking scheme is extremely computationally intensive. Being able to support high video quality and high throughput while minimizing distortion in a reasonable amount of time is a daunting task. Fortunately, the encoding and decoding stage of the H.264 video compression and the inserting of the digital watermarking do contain operations that can be parallelized such as computing DCT coefficients from macroblocks, variable block size motion estimation implementation, inter prediction, and intra prediction to name a few.

1.2. Heterogeneous CPU-GPU Computing Systems

Modern GPUs are massive parallel processors that are designed to support many threads and are throughput oriented. They are designed to complement the CPU, not replace it. This is because the GPUs do not perform well on some tasks that the CPU is designed to perform well on, such as tasks that are serial in nature. For application tasks that are low latency and only have a few threads, the CPU is the best choice. For application tasks that have a long latency, high throughput, and are parallel in nature, the GPU is the best choice. The GPU is also a good choice for handling numerically intensive tasks because of the many arithmetic units the GPU supports. Thus, a CPU-GPU computing system is needed to tackle the many tasks that are presented in the system and execution of an application.

The heterogeneous CPU-GPU system consists of the host, the CPU, and 1 or more GPU devices. To support the heterogeneous CPU-GPU execution of an application, NVIDIA developed CUDA as a generic data-parallel programming computation model for non-graphics applications. This facilitated the ease at which non-graphics programmers can take advantage of the GPU parallel computing capabilities.

1.2.1. Parallel Programming

Many modern applications process huge amounts of data. Much of the data is modeled on physical phenomena. Much of the data in these systems can be evaluated independently thus forming the basis for data parallelism. This data parallelism needs to be exploited by devices such as the GPU and more importantly, by languages such as CUDA that support massive parallelism and heterogeneous computing. CUDA C is a runtime system that is an extension of the C language and is a parallel computing platform for NVIDIA GPUs. This language supports many threads to exploit the data parallelism inherent in physical phenomena. OpenACC, is a standard for parallel computing using compiler directives to facilitate parallel programming of heterogeneous computing systems such as CPU-GPU systems.

1.3. Digital Watermark on H.264 Compressed Video

Digital watermarking is an image inserted into images or videos to protect IP and enforce copyright protection for businesses and industry [1]. Video watermarks tend to be used in video applications such as broadcast monitoring, source tracking, and copyright protection.

Digital watermarking is a form of information hiding related to steganography with the distinction being that digital watermarks are robust to malicious or accidental attacks. The three

aspects of information hiding are security, capacity, and robustness. Security deals with the detection of information, capacity deals with the amount of bits hidden, and robustness deals with resistance to accidental and malicious modifications. With robustness being desirable for copyright protection schemes, video watermarking algorithms normally prefer robustness. Removal of the watermark will introduce severe degradation.

H.264 is an industry standard video compression format (or specification) for reducing video capacity size for storage and/or transmission. It is useful for many applications dealing with recording, compression, and distribution such as Blu-ray disc, broadcast, mobile digital, and digital satellite TV, high definition recording formats, internet streaming video, video surveillance, and many more applications dealing with multimedia. H.264 is a specification only for encoding videos for video compression and decoding videos for decompression for storage and/or transmission. The video codec is the actual software or hardware implementation of a specific video compression format such as the H.264 format and once the video is encoded, it can be bundled with the compressed audio such as the advanced audio coding (AAC) in a multimedia container format such as MP4. The container format can then be used for playback in a software video player.

H.264 encoding involves many states: inter/intra prediction, calculating the residuals, transformation, quantization, and entropy coding. H.264 decoding contains the reverse operations: entropy decoding, rescaling, inverse transform, and reconstruction from the decoded residual macroblock (MB) and prediction MB. Video watermarking for the H.264 format can take place during the H.264 encoding stages and generally happens before or after quantization, which is of course, dependent on the algorithm. Other schemes decode and re-encode the video

for watermark embedding while other schemes embed the watermark in the compressed domain using techniques such as embedding the watermark in the residual of the I-frames.

1.3.1. Parallel Processing of Digital Watermarking and H.264 Operations

Digital watermarking and H.264 operations on video can be very computationally intensive depending on the application, the amount of spatial detail and motion in the video, and the length of the video. Real-time watermarking adds more complexity to this process and thus, it is difficult to achieve high quality video with a real-time watermark in a reasonable amount of time. H.264 encoding also aims to minimize the coded bitrates and maximize the decoded quality of the video and at the same time, the watermark tries to minimize the distortion introduced. This introduces computational complexity to the video encoding and decoding processes.

The CUDA C language enhances the programmability and flexibility of a GPGPU application, and being able to offload the H.264 coding and watermark process on the GPU would greatly increase the speed of the application. To accomplish this, the data itself would have to be parallelizable. Also, the data access to the global memory and memory transfers between the CPU and GPU would have to be reduced, thus increasing the usage of the shared memory in the SMs for the parallel algorithm to be of good use.

The algorithms for the H.264 compression and digital watermark are block-based and the key feature of these blocks is that they can be processed independently. This is a significant observation when considering computational complexities of the motion estimation, the evaluation of the rate-distortion modes, and integer transformations of the H.264 encoding process, to name a few. Also, considering the fact that the watermark has to be embedded during

one or more of these processes depending on the algorithm adds much computational complexity to the process. Being able to offload one or more processes to the GPU would be a significant step towards increasing the application speed and reducing the time to run the application.

1.4. Organization of This Thesis

The major objective of this thesis is to show the high performance that can be obtained on a CUDA-enabled GPU for general computation problems related to non-graphics applications. Two numerically intensive non-graphics applications, H.264 video compression and digital video watermarking, will be parallelized to demonstrate the power of GPU computing.

Chapter 2 covers previous works that are relevant to this thesis. It reviews the history of GPGPU and its present state. It then covers digital watermark parallelization and its hardware and software implementations on the GPU. The parallelization of the video coding process is also covered. Finally, the novel contributions of this thesis are presented.

In chapters 3 and 4, an overview of GPGPU and parallel computing and programming on a CUDA-enabled GPU are presented. Chapter 3 introduces the modern architectures of the CPU, GPU, and the heterogeneous CPU-GPU system. Concepts related to GPGPU are then covered next. Chapter 4 covers the different languages and libraries that can be written and run on a CUDA-enabled GPU. The hardware architecture and programming model of the CUDA-enabled GPU is presented. Finally, a summary of non-graphics applications that can be programmed to run on a CUDA-enabled GPU is presented.

Chapter 5 covers the details related to digital video watermarks. It presents an overview and covers details on video watermarks regarding embedding and detection, classification of

watermarking approaches, security, attacks, and H.264/MPEG-4 AVC watermarks. Finally, sequential and parallel digital video watermark implementation details are discussed.

Chapter 6 introduces the concepts of the H.264/MPEG-4 AVC video compression standard. The encoding and decoding process is discussed in detail. It then covers the sequential software and hardware implementations and the parallel hardware and software implementations.

Chapters 7-9 discuss the proposed algorithm, experimental setup, and conclusion and future research. Chapter 7 discusses the proposed video compression and watermark in GPU computing environment. It covers the details on the H.264 video compression, H.264 digital video watermark, the watermark embedding, and the implementation on the GPU. Chapter 8 covers the experimental setup, the benchmarks, the coding, and the results. Chapter 9 offers some concluding remarks and discusses directions for future research.

CHAPTER 2

RELATED PRIOR RESEARCH

2.1. General-Purpose Computing on Graphics Processing Units

Today's GPUs are characterized as massively parallel processors designed to perform a massive number of floating-point calculations and numerically intensive tasks. They are designed to complement the CPU for numerically intensive tasks and thus form the Heterogeneous CPU-GPU architecture. The GPU is considered the most powerful parallel computing system in the PC market, thus giving the GPU a very large installed base in the marketplace. Researchers and developers have took notice of the power of the GPU and thus, became interested in harnessing the power of the GPU for general purpose computation, otherwise known as general purpose computing on the GPU (GPGPU).

Today's GPUs allow the application developer and researcher to program the fully programmable unified processor using a generic parallel programming model with a hierarchy of parallel threads, barrier synchronization, and atomic operations though this was not the case at the beginning. In 2002, after the release of DirectX 9 by Microsoft, researchers took notice of the raw performance growth of the GPU but to access the performance on the GPU for general-purpose computation, the computation problem had to be mapped to the graphics hardware, or in other words, the computation problem had to be cast to graphics operations. This was a tedious task. Owens et al. [2] provide a good introduction and survey of GPGPU. It is a literature review of GPGPU and reviews many of the algorithms and techniques that were employed to map general computation to GPUs. Ian et al. [3] wrote an extension to the standard ANSI C to incorporate the stream processing paradigm with the focus on data parallelism and arithmetic intensity, operations on data to remove data requests to the host memory and maximize localized

computation. Brooks is also the precursor to CUDA [4]. BrooksGPU [5] is a runtime implementation of the Brooks programming language for the GPU. Mark et al. [6] wrote Cg, a C-like language, to program the programmable floating-point vertex and fragment stream processors. As part of Microsoft research in 2006, the Accelerator [7] was written to provide a general parallel programming model where the programmer can write in a conventional programming language without having to learn the graphics API.

With the advent of the Direct10 x specification, Nvidia [8] took the GPU programming model a step further and came out with a specification for non-graphics applications to support a data-parallel computation model for GPGPU, known as CUDA [9]. The CUDA programming language [9] is part of the CUDA toolkit [10] and is used as an extension to C, an imperative programming language, to enable programmers to construct data parallel applications without having to learn the constructs of the graphics API and to an extent, the graphics hardware. This has accelerated the use of GPGPU to solve general computations to many fields related to cryptography, watermarking, video coding, bioinformatics, computational chemistry, computational fluid dynamics, electronic design automation, and imaging and computer vision to name a few. Kronos Group, Inc. implemented the OpenCL language specification [11], which has similar constructs compared with CUDA. In addition, OpenCL is more general in the sense that it is cross platform, parallel programming language made to across different modern processors and is endorsed by Intel [12], AMD [13], and Nvidia [14].

Many books and articles have been written with regards GPU parallel computing. The CUDA specification [9] covers all the language constructs for CUDA. The OpenCL specification [11] covers all the language constructs for OpenCL. Kirk et al. [15] discuss programming massively parallel processors using programming tools such as CUDA, OpenACC,

C++ AMP, and OpenCL to demonstrate the data-parallel computation model and heterogeneous computing. Farber [16] show the use of the CUDA toolkit for application design and development for massive parallel processors on the GPU. Kim et al. [17] present a high-level overview of GPGPU and parallel programming models and provide detailed performance analysis and guide optimizations for GPGPU programming. The GPGPU [18] website is devoted to GPGPU as a whole and contains many articles written about the current and future trends of GPGPU.

2.2. Digital Watermark Parallelization

There is a plethora of articles dedicated to digital watermarking and video encoding on serial processors such as the CPU though much less has been written regarding GPU and heterogeneous CPU-GPU implementations. The literature that does exist, is mostly dedicated to video coding with very few works in image watermarking and video watermarking being almost non-existent.

Some of the initial watermark implementations for the GPU were implemented using hardware-assisted methods or the graphics API before the advent of CUDA. The two reasons for using hardware assisted methods is based on the fact that digital watermarking involves computationally intensive operations and that results need to be generated instantaneously, especially for digital TV broadcasting and video conferencing.

2.2.1. Hardware Implementation

Mohanty et al. [19] presented an invisible-robust image watermarking hardware solution on the GPU by using a dedicated processor chip as co-processor on the GPU to achieve an

efficient high performance, real-time, and low-cost watermarking-system. Mohanty et al. [20] designed a very-large-scale integration (VLSI) architecture for accelerating watermarking methods that perform invisible robust and fragile watermarking using a field-programmable gate array (FPGA) and custom integrated circuit (IC) implementations. Kougianos et al. [21] conduct a survey of hardware based watermarking systems on GPUs, FPGAs, and digital signal processors (DSP). The caveat for the above hardware assisted watermark implementations is that they require hardware design, thus putting a limit on programming and requiring changes in the hardware architecture.

Maes et al. [22] implemented a hardware real-time watermarking detector implementation for DVD applications on an FPGA board and a TriMedia processor (IC) board. The FPGA board shows better results in terms of implementation costs and relevancy.

The authors in [23] have implemented a spread spectrum-based watermark algorithm that can be used for real-time software or low-cost hardware implementations on low powered devices such as cellular phones and PDAs, thus making the algorithm very versatile. The software and hardware implementation can be combined to reduce the overall necessary gate count, on-chip memory, and system bandwidth of the underlying hardware.

2.2.2. Software Implementation

2.2.2.1. Digital Image Watermark

Lin et al. [24] have proposed a semi-fragile watermarking algorithm for image authentication using CUDA to accelerate the discrete cosine transformation (DCT) and watermark extracting and embedding operations. Compared to the CPU, a single GPU achieves a speed-up of 32x for a 1024×1024 image size while a dual GPU achieves a speed-up of 40x. In

both cases, the goal was to achieve an operational speed-up while maintaining authentication results.

Cano et al. [25] have implemented a blind image watermarking algorithm proposed by Shieh et al. [26] using a CUDA approach to parallelize the DCT and the watermark image insertion and extraction. The benchmarks obtained from the experiments show an improved speed-up from the localized computations in the GPU but modest results when considering the communication overhead between host and GPU. This was probably due to not accessing CUDA's different memories efficiently to reduce global memory traffic between host and CPU. Cano et al. [27] implement a particle swarm optimization algorithm to optimize the blind image watermarking algorithm presented in [26] using CUDA to speed-up the optimization and the watermark algorithm.

In [28] the authors have proposed a CUDA implementation of their image watermarking algorithm that uses Huffman coding for the compression of copyright data and the modified auxiliary carry watermarking method (MACW) for embedding. CUDA is used to improve the operation speed-up of the highly compute-intensive embedding and extraction operations while maintaining coding efficiency and picture quality.

2.2.2.2. Digital Video Watermark

Brunton and Zhao [29] implement an invisible-fragile real-time video watermarking algorithm on a GPU with the watermarking algorithm having to be cast to graphics operations. The real-time implementation on the GPU involves using the vertex shader for the insertion and extraction operations. The fragment shader generates the tiled watermark image from the input

video stream and performs exclusive OR operations for insertion of the results in a video stream and for an output signature image.

2.3. Video Coding on GPU

The H.264 coding process is a compute intensive process with the motion estimation (ME) unit in the encoding process being the most compute intensive operation, taking as much as 80% of the total encoding time. Several papers in the literature have been devoted to parallelizing the H.264 video coding process in the GPU with the majority focusing on the ME.

Youngsub et al. [30] proposed a parallel motion estimation algorithm that takes into account the inter-dependencies in the calculation of motion vector predictions in motion estimation. It uses a similar approach to the x264 encoder diamond search algorithm and hexagon search algorithm in that only a few macroblocks (MBs) are looked at compared to all MBs in a full search algorithm to determine the best candidate for motion estimation. The difference is that the GPU algorithm uses a different approach for calculating the MVPs among the MBs to remove the interdependencies and shows a 20% speedup compared with the x264 encoder diamond search algorithm. Chen and Hang [31] parallelize the variable block size ME with fractional pixel refinement using 5 steps to maximize localized computation and reduce communication overhead between the host and GPU. Rodriguez et al. [32] parallelize the H.264 inter prediction by using CUDA to accelerate the tree structured motion compensation algorithm to generate a memory matrix that can be read in parallel to construct an approximation of the optimal macroblock mode coded partition that is needed for inter prediction. The algorithm does introduce a negligible rate distortion (RD) drop but the improved speedup is one step closer towards an efficient real time video encoder.

Lee and Oh [33] proposed a highly parallel variable block size full search ME algorithm with concurrent parallel reduction (CPR) using CUDA with the aim of maximizing the number of active threads and minimizing the number of synchronization points by using CPR over the conventional PR method. Hierarchical SAD computing is used for data reuse to reduce the computation time and the proposed algorithm efficiently uses on-chip memory of the GPU to minimize long-latency of the accessed data and instructions in memory.

Rodriguez-Sanchez et al. [34] present a CUDA-based ME implementation to reduce the H.264 inter prediction complexity using data structures that can be generated and read in parallel. The algorithm was also adapted for multiple-frame reference-based ME and high video resolution and the testing results show a 99% speedup with negligible coding efficiency penalty.

Wu et al. [35] proposed a multi-level parallel and memory efficient algorithm that maps the H.264 encoder that is based on the x264 reference code to the GPU using CUDA. Almost all the entire framework of the encoder is executed on the GPU to reduce memory transfers, exploit parallelism, and improve optimizations.

Liu and Chen [36] proposed an intra-prediction parallel algorithm for the decoding phase that takes advantage of multiple GPUs to decode 100 to 1000 videos per GPU using four different optimization techniques. The GPU's effectiveness fades as the video load increases and performs nearly the same as the original algorithm for 1000 videos. This could possibly be due to the proposed algorithm not being redesigned to support parallel structures efficiently and there may have been a large communication overhead among the threads.

NVIDIA [37] has proposed a H.264 hardware-accelerated video encoding that supersedes the previous NVIDIA CUDA video encoder and an NVIDIA CUDA video decoder (NVCUVID) API library for decoding video streams MPEG-2, VC-1, and H.264 on Video Processors or

CUDA-enabled GPUs. The former uses dedicated hardware on certain classes of GPUs and can be accessed by NVIDIA encoder (NVENC) API to encode H.264 videos. The latter is part of the GPU computing SDK provided by NVIDIA.

The authors in [38] show the applicability of parallel programming models to video processing for multi-core CPU and many-core GPU technologies. Case studies from three video processing domains are demonstrated to show different optimization strategies for the parallel programming models. The authors in [39] explore efficient implementations of video coding standards on the GPU focusing mostly on the H.264 standard. Datla and Gidijala [40] port the motion JPEG 2000 reference video encoder to a CUDA-enabled GPU to parallelize the video encoder using the CUDA model and show significant speedups of 20.7 compared to the CPU implementation.

Obukhov and Kharlamov [41] propose a discrete cosine transform (DCT) that operates on 8x8 blocks in the GPU and the GPU can process the 8x8 blocks independently. This DCT algorithm can be used in image and video coding.

Xiao and Baas [42] propose a 1080p H.264/AVC baseline residual encoder that exploits fine-grained and task-level parallelization in the integer transform, quantization, and context-based adaptive variable length coding functions and can run on a programmable many-core processor. This algorithm is designed to be independent application specific hardware such as a H.264 encoder being designed to run on a CUDA-enabled GPU.

2.4. Novel Contributions of This Thesis

Most of the literature regarding the parallelization of the H.264 video compression focuses on parallelizing the ME unit in the encoding process because it is the most compute

intensive operation taking as much as 80% of the total encoding time. Many variations of parallelizing the ME unit are proposed. There is also literature regarding the parallelization of the integer transform process, an approximation to the DCT transformation process, for H.264 and generic DCT transformations that would apply to many block-based compression formats including H.264 [41].

The literature regarding parallelization of video watermarking on the GPU is virtually non-existent. There is some literature regarding parallelization of image watermarking on the GPU [24-28].

The contributions to this thesis will be parallelizing the digital video watermark in the H.264 compressed domain, where the literature is non-existent for this type of watermark on a CUDA-enabled GPU. In the H.264 compressed domain, the integer transform, an approximate form of the discrete cosine transform (DCT) during the encoding process, will be parallelized on the CUDA-enabled GPU. The visible watermark will be embedded in the integer transform coefficients during the H.264 integer transformation process and will also be parallelized. The execution time of this program will include the H.264 encoding and decoding process for the MP4 container format though no parallelization was introduced during the decoding process.

So to summarize, the contributions will include:

- Embedding a digital video watermark in the H.264 compressed domain using a CUDA-enabled GPU
- Parallelizing the integer transform process in the H.264 encoding phase
- Parallelizing the visible watermark embedding in the integer transform coefficients

CHAPTER 3

GENERAL PURPOSE COMPUTING PROBLEM ON GPU

3.1. Overview

Currently, the CPU multi-core architecture is a multi-core system designed to work on general computing of data that is serial in nature and maintain the execution speed of sequential programs. Most CPU systems have between two and eight cores, each core supporting two hardware threads to maximize the sequential program's execution speed. The CPU architecture supports out-of-order execution, multiple instruction stream, multiple data stream (MIMD). Though two or more cores allow data-parallelism in the sense that different parts of the program can run on different cores, the cores themselves are primarily serial based. Parallel languages such as OpenMP, MPI, and MCUDA take advantage of the parallelism supported by the CPU cores. The design style of the CPU is latency-oriented design to support the requirements of legacy operating systems, applications, and I/O devices, thus putting the CPU at a disadvantage in terms of parallelism. A software programmer trying to use a parallel programming model strictly on a CPU will be at a performance disadvantage compared to a CPU-GPU execution of an application.

Modern GPUs are many-core processors that are designed to support many threads and support a throughput-oriented style. They are massive parallel processors designed to perform a massive number of floating-point calculations. They are designed to complement the CPU for numerically intensive tasks, not replace it. The GPU uses a large number of parallel threads to execute kernels while hiding long-latency arithmetic operations or memory accesses compared to a CPU, which only executes a few threads at a time. Therefore, if a program does have data portions that can be parallelized and requires a large number of threads, using the GPU is a good

option to achieve higher performance compared to just running the application on a CPU by itself. There are several parallel languages such as CUDA, OpenACC, OpenCL, and C++ AMP.

3.2. Modern CPU Architecture

The multi-core CPU is a Von Neumann model that is designed to support multiple cores with two threads each. The CPU supports a memory hierarchy containing thousands of registers and multiple levels of on-chip caches, usually 3 levels of caches, with non-volatile memory being at the bottom of the hierarchy. The CPU is connected to input/output (I/O) devices such as a mouse, keyboard, monitor display, and printer that allow the user to interact with the CPU. The CPU is a latency-oriented design that supports large cache memories for short-latency cache accesses, low-latency arithmetic units, and sophisticated operand delivery logic. This design comes at the expense of the supporting features taking up more chip area and using more power as opposed to proving more arithmetic units and memory access channels.

The first level on-chip cache is the smallest in size and the fastest with the last level on-chip cache being the largest in size but the slowest. Cache uses SRAM chips that are much faster than DRAM chips. The last-level on-chip cache is designed to store frequently accessed data to reduce the long-latency of the accessed data when it is stored in the DRAM. L1 caches usually are directly attached to a processor core and thus, changes to their content are not reflected in the other processor cores. Thus, a cache coherence mechanism is put in place to ensure that any changes to the contents in one L1 cache, updates the L1 caches of the other processor cores.

The main memory is DRAM, is much larger in size compared to SRAM and is much slower. The memory hierarchy of the CPU is designed to reduce the long-latency data and

instruction accesses in memory for large complex applications. The DRAM is also used to transfer data to and from a GPU device and the bottleneck for parallel computing when a large number of data transfers take place.

Non-volatile memory is memory that retains its information when the CPU is powered down. Typical non-volatile memories include hard-disk drives (HDD), solid-state drives (SSD), optical disks, and flash memory to name a few. Data is transferred to and from the processor cores and non-volatile memories via the memory hierarchy. Non-volatile memories are the largest in size in the memory hierarchy but are also the slowest, thus they are the bottleneck and form the memory wall. Thus, it is important to use faster non-volatile memories such as SSDs to see an increase in speed in the overall system.

I/O devices are needed to allow the user to interact with the CPU. The devices are used to transfer data to and from the CPU using techniques such as polling and I/O interrupts. Figure 1 shows a typical example of a multi-core processor based off the Von Neumann model.

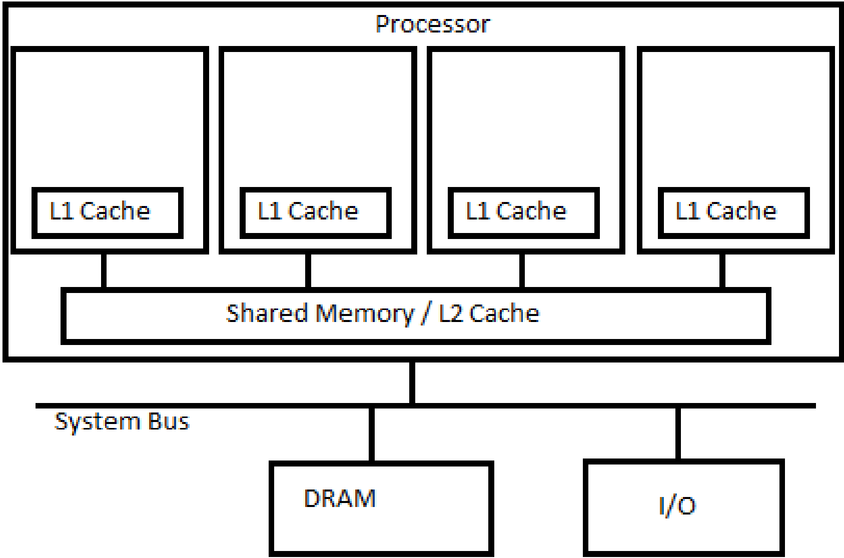


Figure 3.1: Multi-core chip with inter-core bus

3.3. Modern GPU Architecture

The GPU is a massively parallel processor designed to support many processor cores with a large number of threads to achieve a high level of performance for floating-point calculations. The GPU is a throughput-oriented design to maximize the total execution throughput of a large number of threads with an increase in arithmetic units and memory access hardware for higher memory bandwidth. The GPU follows the SIMD model for execution for parallel execution among the threads.

The GPU was redesigned during the DirectX 10-generation graphics era to support graphics and non-graphics applications. Vertex shading, fragment processing, and geometry processing are integrated into a unified fully programmable processor.

The GPU is organized as an array of streaming multiprocessors (SMs) with two SMs forming a building block and each SM containing eight streaming processors (SP) or more. Each SP has its own set of registers, ALU, and FPU. The SPs in the SM share control logic and instruction cache such as shared memory and L1/L2 caches. The SM execution resources include registers, shared memory, thread block slots, and thread slots. Each SM can support a certain number of blocks and a large number of threads. Figure 2 shows an SM example and figure 3 an SP example.

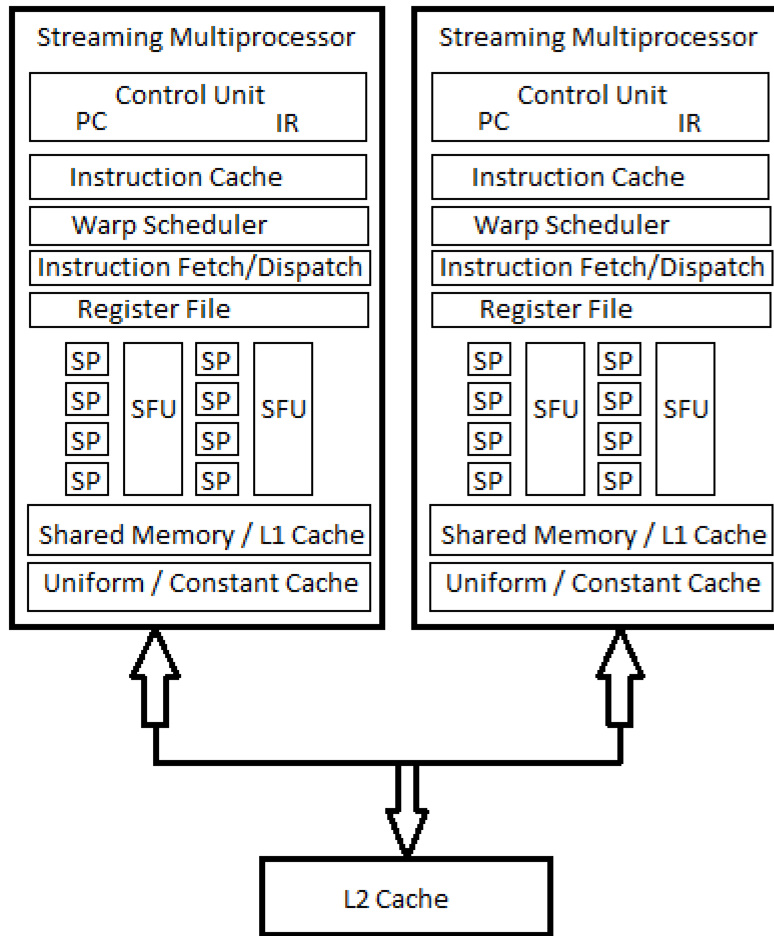


Figure 3.2: CUDA GPU streaming multiprocessor (SM)

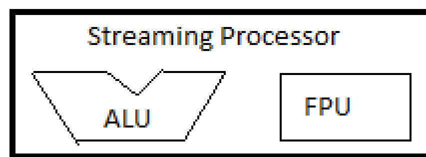


Figure 3.3: CUDA GPU streaming processor (SP)

The CUDA memory architecture includes global memory, registers, shared memory, caches, constant memory, and texture memory. The global memory, known as graphic double data rate (GDDR) DRAM, is designed to support a high bandwidth that hides long-latency instruction and data accesses. The global memory can be read and written by threads and it communicates with the host memory for data transfers. Registers are on-chip memory that is

designed to support short access latency and higher access bandwidth. Each SP has its own local registers for local storage. Shared memory is also on-chip memory that is shared by all SPs in the SM and supports lower latency and higher bandwidth than the global memory. Shared memory is slower than registers due to memory load operations. The GPU cache is on-chip memory designed to reduce the number of accesses to the DDRAM and automatically coalesces data access patterns though the GPU cache does not currently support a cache coherence mechanism. Constant memory is located in the DDRAM though since the memory cannot be modified during kernel execution, the constant memory is cached in the L1 cache and is broadcast to a large number of threads. Though constant memory cannot be modified by a kernel function, it can be modified by the host memory. Texture memory is a special type of device memory that is cached for data locality. Memory coalescing techniques are utilized to move data from the global memory into registers and shared memories more effectively.

3.4. Heterogeneous CPU-GPU Architecture

The GPU is designed to be a co-processor to the CPU, not replace it. This is because the GPUs do not perform well on serial executions that the CPU is designed to perform well on. The CPU is the best choice for application tasks that are low latency and only require a few threads. The GPU is the best choice for long latency, high throughput, and parallel executable application tasks. The GPU contains many arithmetic units for handling numerically intensive tasks and high floating point calculations. Thus, a CPU-GPU computing system is needed to tackle the many tasks that are presented in the system and application execution.

The CPU contains a host interface that is setup to allow communication between the CPU and GPU. CUDA-enabled GPUs communicate with the CPU via a PCI express connection to

the host. The new AMD "Kaveri" accelerated processing unit (APU) takes it a step further and merges the CPU and GPU on the same chip using the heterogeneous systems architecture (HSA) standard. The HSA standard allows different processors including a CPU, GPU, DSPs, and others in a system to work together. It uses a heterogeneous unified memory access (hUMA) to allow the GPU and CPU to access the same memory.

3.5. General-purpose Computing on Graphics Processor Units

3.5.1. Overview

Today's GPUs are characterized as massively parallel processors designed to perform a massive number of floating-point calculations and numerically intensive tasks. They are designed to complement the CPU for numerically intensive tasks and thus form the heterogeneous CPU-GPU architecture. The GPU is considered the most powerful parallel computing system in the PC market, thus giving the GPU a very large installed base in the marketplace. Researchers and developers have took notice of the power of the GPU and thus, became interested in harnessing the power of the GPU for general purpose computation, otherwise known as general purpose computing on the GPU (GPGPU).

3.5.2. Concepts and Methods

GPGPU computing follows the SIMD, single instruction multiple threads (SIMT), and single program multiple data (SPMD) models for parallel computing. In the SIMD model, the processing sends out the same instruction, via the control signal, to many processing units. Each unit is executing the same instruction at the same time to process a different set of data. In the SPMD model, the processing units are executing the same program on different sets of data and

are not required to execute the same instruction at the same time. The SIMT model specifies the execution and branching behavior of a single thread and allows the programmer to write thread-level parallel code and data-parallel code.

Thread-level parallelism (otherwise known as task-level parallelism) decomposes an application into independent tasks. Each thread processes a task using its own sets of instructions and data. Data-level parallelism is where identical instructions are performed on different data.

Decomposition is the division of a computation into smaller and manageable tasks where some or all tasks can be executed in parallel. A task is a programmer defined unit of computation. The granularity of decomposition is the size and number of tasks of a decomposed computational problem. GPGPU supports two types of granularities: coarse-grained data-parallelism and fine-grained data-parallelism. Coarse-grained data-parallelism is a small number of large tasks such as thread blocks that are composed of threads. Fine-grained data-parallelism is a large number of small tasks such as CUDA threads.

Whether using parallel programming languages such as OpenMP, MPI, OpenCL, CUDA, or MCUDA to write applications on the GPU for GPGPU or the CPU, the languages contain a few important fundamental operations that are inherent in data-parallel processing. The methods include scan, map, reduce, stream filtering, scatter, gather, sort, search, and barrier synchronization. Scan, otherwise known as all-prefix-sums, is an exclusive scan where a sum S_k is the sum for all k elements between 0 and p-1. The map operation applies a kernel to a set of data, known as a stream. Stream filtering is a non-uniform reduction in the sense that it uses a set of criteria to remove certain data from a stream. Reduce is the operation where all values are reduced to a single value such as calculating the maximum value. Gather is where a large

number of data items are read from given locations. Scatter is where a large number of data items are written to given locations. The sort operation uses a criterion to sort or rearrange data. The CUDA C++ template library, Thrust, is used to perform data parallel primitives such as sort, scan, and reduction. The search operation searches for one or more items based off some criterion. Barrier synchronization is a coordination mechanism that allows threads in the same block to coordinate their activities to ensure all threads finish executing one phase of a kernel function before moving on to the next phase. This ensures no data is corrupted or overwritten by threads that finish their activities first before others finish. The above operations are done in parallel to decrease the execution time of an application.

3.5.3. Bottleneck

There are many potential performance bottlenecks and resource limitations and this is dependent on the particular use of resources in the application. The most looked at bottleneck is the communication that takes place between the CPU and GPU via the memory. There are other limitations such as registry, thread, and thread-block limitations. For example, declaring two additional automatic variables in a kernel function increases the registry count by two in every thread. If the registry count exceeds the maximum number of registries an SM may support, the CUDA runtime system reduces the number of blocks assigned to each SM and thus, reduces the thread count. This ultimately reduces parallelism and performance in an application. The performance cliff is where an increase in one resource has an unintended decrease in another resource, thus reducing the parallelism and performance in an application as described in the above example. Several tests will need to be performed to determine the application's

bottleneck. Using debugging and profiling tools such as NVIDIA visual profiler can be used to identify the bottlenecks and optimize the application

CHAPTER 4

PARALLEL COMPUTING AND PROGRAMMING ON A CUDA-ENABLED GPU

4.1. Overview

Today's GPUs are fully programmable massively parallel processors designed to perform a massive number of floating-point calculations and achieve higher performance on non-graphics applications using parallel operations. Parallel computing solves a problem much quicker compared to sequential execution, assuming a large portion of data can be parallelized. Various languages including the CUDA language and libraries such as Thrust, have been developed to take advantage of the parallelism that is inherent in the multi-core CPUs and accelerative devices such as the GPU.

4.2. Languages and Libraries

Many parallel processing languages and libraries are designed to improve the performance of parallel computing in non-graphics applications. Programming the GPU just ten years ago required direct knowledge of the GPU architecture and low-level programming techniques. Languages such as CUDA have made it much easier for the non-graphics programmer because they abstract the hardware, alleviate having to learn the graphics API, and allow the programmer to focus on the application at hand. On the other hand, the programmer does have to write their code for parallel execution instead of sequential execution with the exception being OpenACC, a compiler directive language. Also, for the application developer to achieve a high-performance parallel program with an improved speedup, knowledge of the GPU hardware architecture is needed to make better use of the software programming tools such as CUDA.

4.2.1. CUDA

CUDA is a high-level language that is an extension to the C language and is used to program GPU-accelerated applications. It is a parallel computing platform and programming model designed to allow the non-graphics programmer to take advantage of the high performance of the GPU for parallel code execution. CUDA is used in many fields related to cryptography, watermarking, video coding, bioinformatics, computational chemistry, computational fluid dynamics, electronic design automation, and imaging and computer vision to name a few. CUDA is designed to work with GPU-accelerated libraries such as Thrust, a parallel algorithms library similar to the C++ standard template library (STL) for defining type-generic parallel algorithms, and to work with GPU directives such as OpenACC, a programming standard that provides a collection of compiler directives to run parallel programs on accelerator devices such as APUs, GPUs, and many-core processors.

4.2.1.1. Scalability

As opposed to parallel programming for the CPU, parallel programming on the GPU using a parallel processing language is designed to allow scalability as the GPU core count increases. This is due to the design of the GPU parallel programming model.

For example, CUDA follows a GPU parallel programming model that uses fine-grained data parallelism techniques with efficient threading support for graphics and parallel computing. The CUDA threads allow the exploitation of data parallelism inherent in a program. Efficient thread scheduling in particular allows a much larger amount of data parallelism to be exploited than the underlying hardware resources would allow on the machine the software program is running on. This is important for CUDA software programs that run on many different types of

CUDA-enabled GPUs that have different amounts of hardware resources and will have a long software life cycle. Since the CUDA program is based on threads to expose the data parallelism inherent in a software program using fine-grain data-parallelism techniques, the CUDA program just needs to be written once and the software program will be able to take advantage of the hardware resources ranging from a few cores to hundreds of cores to a further increase of cores in the next generations to come in CUDA-enabled GPUs. This provides a form of transparent and portable scalability.

The CUDA parallel programming model is in contrast to the parallel programming models for the CPU that use course-grained data-parallelism to process application tasks. Currently, the CPU cores available in a consumer PC range between four to eight. The CPU-based parallel program is designed to assign application tasks to each core. As the core count increases during each new generation of CPU cores, the application needs to be rewritten to take advantage of the extra hardware resources. Also, if an application is designed to run on four CPU cores and the application will need to run on a CPU architecture with only two CPUs, the application will also need to be rewritten. Thus, the application using the current CPU parallel programming model is not scalable.

4.2.2. OpenACC

The OpenACC application program interface is a programming standard that provides a collection of compiler directives to run parallel programs on accelerator devices such as APUs, GPUs, and many-core processors and simplify parallel programming on heterogeneous CPU-GPU architectures. The standard is designed to support multiple levels of parallelism found in an accelerator such as a GPU. OpenACC is similar to OpenMPI with some of the OpenACC

members serving in the OpenMP standards group. The difference between OpenMPI and OpenACC is that OpenMPI is used to only specify directives for the CPU whereas OpenACC is for accelerative devices.

The simplicity of OpenACC is that programmers do not need to modify their existing code or learn a new syntax and style of programming to support a parallel application. The programmer simply adds compiler directives, known as #pragma directives, to their sequential code where the directives can be ignored if the program is run on a non-OpenACC compiler. The programmer does not need to specify the details of data transfer between host and accelerator memories, thread scheduling, kernel launching, and parallelism mapping since the OpenACC compiler and runtime handle these details.

In the OpenACC execution model, the target machine includes a host and an attached accelerator. There are multiple execution units at the outermost level and within each execution unit there are multiple threads, with each thread executing vector operations. The execution units and the threads within each execution unit run in parallel. Also, OpenACC is designed to work in conjunction with CUDA during program execution.

The OpenACC memory model treats the host memory and accelerator memory as separate with the assumption that each memory is not able to access the other directly. The model uses similar constructs from CUDA with regards to memory allocation, copying, and de-allocation before kernel launches and transferred results from device to host. The difference is that in CUDA, this needs to be specified explicitly by the programmer whereas OpenACC does this automatically.

4.2.3. C++ AMP

The C++ accelerated massive parallelism (C++ AMP) parallel programming model is an open specification developed initially by Microsoft to allow software programmers to design applications to run on data-parallel hardware while hiding the system intricacies of the underlying hardware. This offers portability, better performance, and better productivity for the application as it can be run on different current hardware and future generation hardware from different manufactures without the application having to be rewritten. In certain cases, the application may need to take advantage of the system intricacies and C++ AMP has features to support this.

The C++ AMP model contains a rich subset of the C++ language for data parallel computations that run on the co-processor and the model works in conjunction with the C++ language that runs on the host. This allows for a small learning curve for software developers that are familiar with C++.

4.2.4. OpenCL

The open computing language (OpenCL) parallel programming model is an open specification designed to run on heterogeneous computing systems such as the CPU, GPU, digital signal processors (DSPs), field-programmable gate arrays (FPGAs), and other co-processors as well. OpenCL is maintained by the non-profit Khronos Group and has been adopted by many organizations including NVIDIA, AMD, Intel, and Apple to name a few. The model is very similar to CUDA and the syntax and operations are very similar as well. One can learn CUDA and easily adopt the OpenCL language in a day or two.

4.3. Programming Model

CUDA is a parallel programming model designed to allow non-graphics software developers to write parallel applications for the heterogeneous CPU-GPU architecture. The model supports a hierarchy of threads, thread blocks, barrier synchronization, and atomic operations.

The programming unit in CUDA is the thread and all threads in a grid execute the same kernel function. Each thread is unique and a coordinate system is used to identify each thread so that the appropriate portion of data can be assigned to that particular thread. The thread organization supports a two-level hierarchy. The first level starts with a grid that is organized into thread blocks up to three dimensions. In the second level, each thread block is organized into threads up to three dimensions. The grid and blocks can have different dimensions. Each thread block contains x threads that are multiples of 32 and each thread block subdivided in warps, the unit of thread scheduling in an SM, with each warp containing 32 threads each. The warp uses the SIMD model and thus, all threads in a warp execute the same instruction. The threads within a thread block coordinate their execution using barrier synchronization so all threads finish their current phase before moving to the next phase. The threads can also share data using shared memory. Figure 4 shows an example of a CUDA 2-Dimensional (2,2,1) grid organization where the grid is organized as (x,y,z).

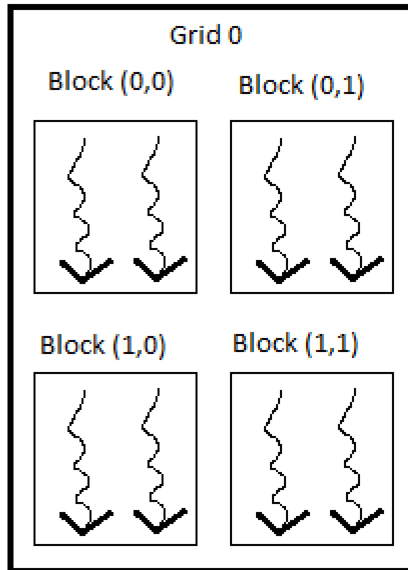


Figure 4.1: 2-Dimensional (2,2,1) grid organization

The kernel configuration is specified with the grid dimensions and block dimensions before execution. To execute, one or more thread blocks are assigned to each SM. Each SM has a limit on the available resources assigned to it. These limitations include thread blocks, threads, registers, and shared memory. Increasing usage on one resource such as the number of threads, will decrease the usage on another resource, such as thread blocks. Each SM is assigned multiple warps, though only a subset can be executed at any given time. The reasoning for this is to hide long latency operations. While a warp is waiting for a result, another warp can be scheduled to execute, thus hiding the long latency operation.

When writing CUDA applications, the programmer needs to be aware of the ratio of floating point calculations performed for each memory access, known as the compute to global memory access (CGMA) ratio. To increase this ratio and thus increase floating-point performance, a programmer needs to optimize the program using algorithmic techniques that make important use of the limited resources at hand.

4.4. Applications

Numerous non-graphics applications have been developed on CUDA-enabled GPUs in the academic and business world. Many of the applications being developed are used to showcase the power of the GPU while other applications being developed are to be used for the specific needs at hand. Fields that are utilizing the GPU for application purposes include cryptography, digital watermarking, video coding, bioinformatics, computational chemistry, computational fluid dynamics, electronic design automation, and imaging and computer vision. The Nvidia CUDA website showcases a number of applications ranging from different fields to show the usefulness and power of the GPU for these types of applications. Many researchers from academia have shown significant speedups across many of these applications when porting the compute intensive areas of these applications to the GPU.

CHAPTER 5

DIGITAL VIDEO WATERMARK

5.1. Introduction

In today's world, there are many digital multimedia applications such as DVDs, TV broadcasting, video-based web applications, and recording to name a few. Today's digital technologies allow content sharing, lossless and high fidelity copying, ease of editing and modifications, and compression. These convenient technologies also introduce the need for digital rights management tools to provide copyright protection, content authentication, and integrity. Many content providers and copyright owners are leery of piracy of their digital products and hence are interested in protecting their IP and illegal copying of their digital products. One solution is the digital watermark, a process of embedding additional data into digital multimedia objects to produce a watermarked multimedia object for the purposes of copyright protection and content authentication. The watermark can later be detected or extracted to prove the content owner of the multimedia object. This technology is provided on top of data encryption.

Watermarking shares its roots in steganography, the art of hiding the existence a secret message to protect against detection. Watermarking has a clear distinction from steganography in that the hidden data is robust against removal techniques. These removal attacks can either be intentional or unintentional.

Various watermarking algorithms have been proposed in the literature for image, audio, video, and text multimedia objects. Since videos are used heavily in the entertainment industry and video surveillance, this thesis focuses on digital video watermarks. Video watermarking algorithms are the most challenging, computationally intensive, and time consuming

watermarking algorithms. Real-time encoding of video streams using a digital watermarking scheme is extremely computationally intensive. What complicates this matter even more is the video compression format of the video the digital watermark is being inserted into, such as the H.264/MPEG-4 AVC (or H2.64 for short). Video compression is needed for videos that involve storage and/or transmission such as movies and streaming video. Being able to support high video quality and high throughput while minimizing distortion with the embedded watermark in a reasonable amount of time is a daunting task. Fortunately, many of the operations of the digital video watermark do show a high degree of parallelism and hence these operations can be computed on a GPU.

5.2. Watermark Framework

Watermark techniques can be classified using the criteria shown in figures 5.1, 5.2, 5.3, 5.4, and 5.5.

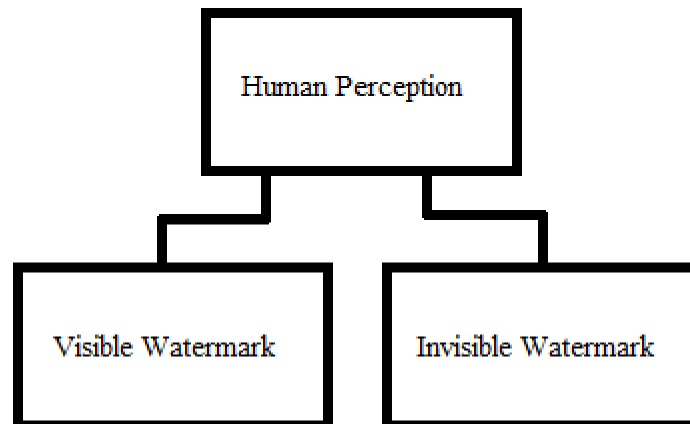


Figure 5.1: Human perception

Regarding figure 5.1, there are two types of watermarks based on human perception: visible and invisible watermarks. A visible watermark is an opaque monochrome or translucent

image embedded as an overlay on an image or video frame. This can be a company logo, copyright notice, or digital photo consisting of text. The more popular invisible watermark embeds a binary image, random or pseudorandom number into the image or video frame and shares the characteristic of being imperceptible to the human eye.

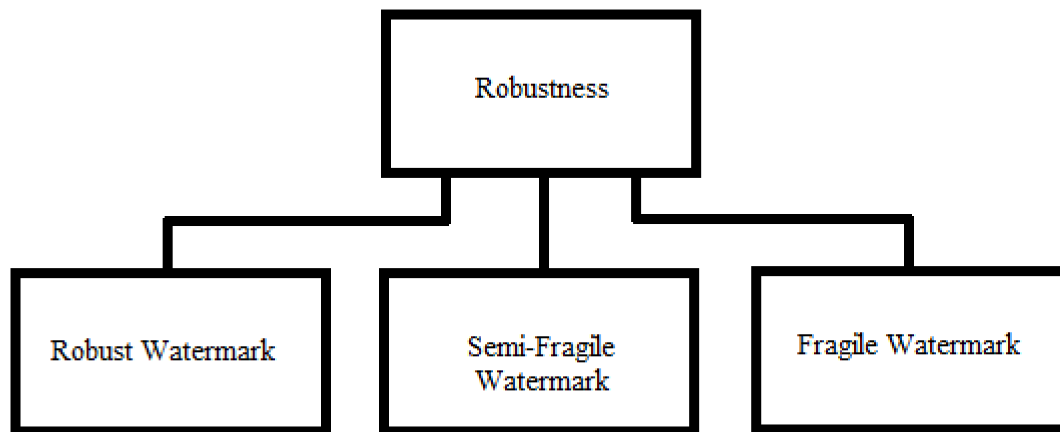


Figure 5.2: Robustness

A robust watermark's performance is determined by being able to withstand intentional and unintentional attacks such as processing techniques and manipulative processes. Robust watermarks are used in applications for copyright protection. A semi-fragile watermark is able to resist simple transformations such as compression that preserve the image or video contents while failing to resist malignant transformations that alter the image or video contents. Its applications involving image authentication are generally used in a court of law to determine if the image or video has been tampered. A fragile watermark fails even the slightest transformation and is used to determine tamper detection in a court of law. A fragile watermark is also another form of image authentication.

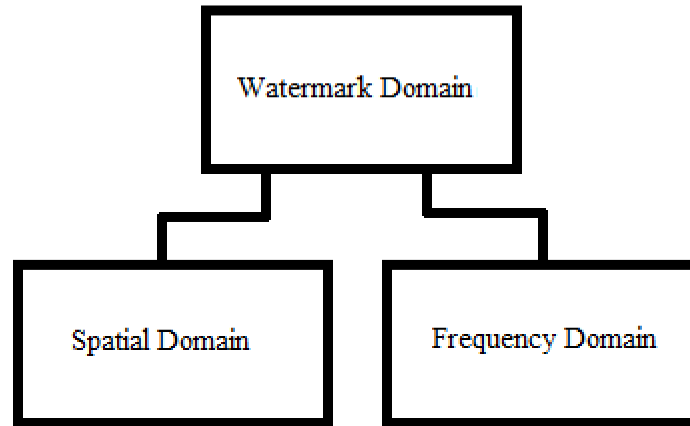


Figure 5.3: Watermark domain

The spatial domain involves slight modifications to the values of the pixels. This is the least computationally intensive of the two and is easier to implement. The caveat is that it is less robust to attacks. In the frequency domain, the multimedia object goes through a transformation, DCT for example, and the watermark is embedded in the transform coefficients by modifying them slightly to represent the watermark.

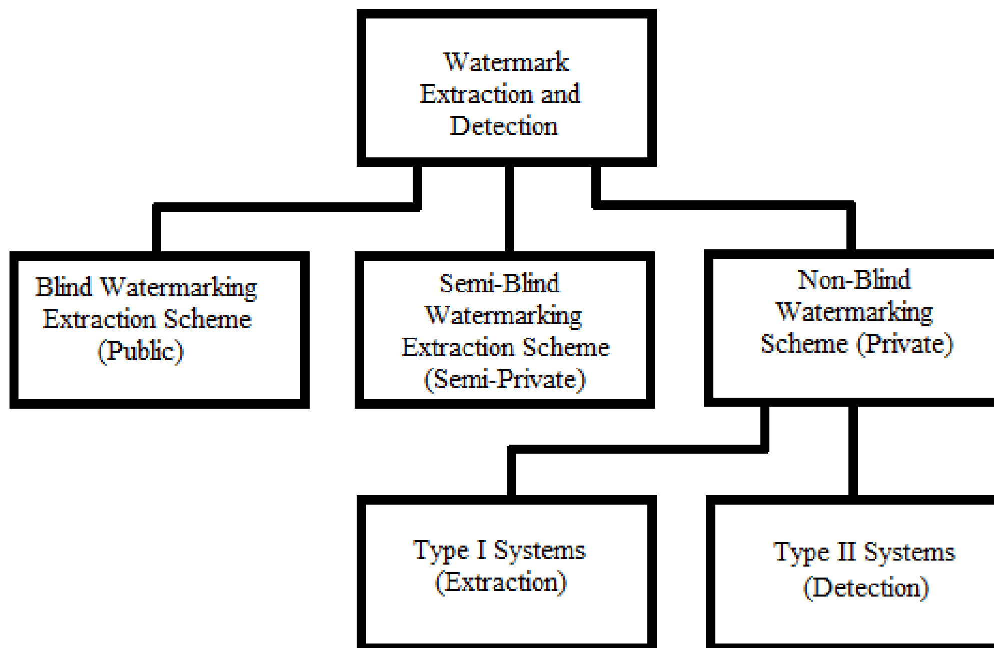
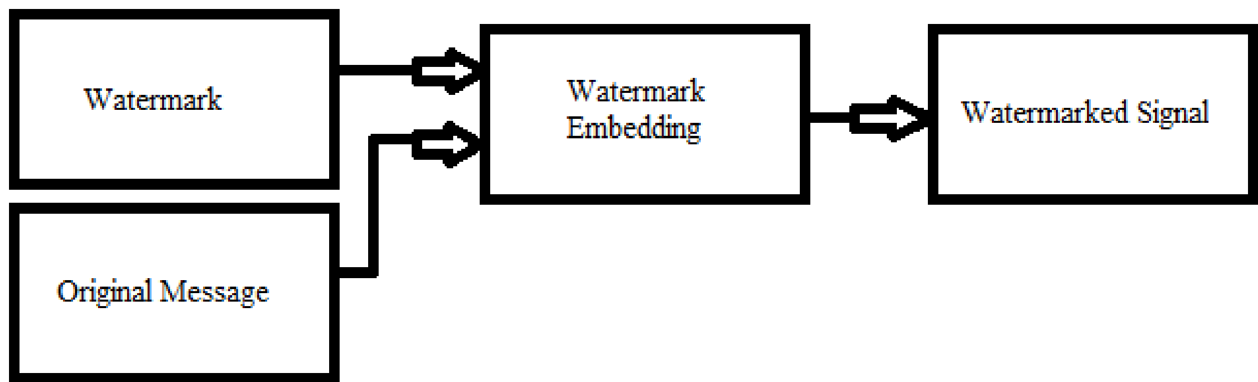


Figure 5.4: Watermark extraction and detection process

A blind (or public) watermarking extraction scheme is the most challenging but most popular type of watermarking scheme. It requires neither the original data nor the original watermark to extract the watermark from the watermarked object. A semi-blind (or semi-private) watermarking detection scheme does not require the original multimedia object but does use the watermarked multimedia object and original watermark for detection of the watermark in the watermarked multimedia object. A non-blind (or private) watermarking scheme is the most robust but has limited application use. These systems require at least the original multimedia object and they consist of two types. Type I systems use the original multimedia object and watermarked multimedia object to determine where the watermark is hidden at so it can be extracted. Type II systems use the original multimedia object, watermarked multimedia object, and the original watermark for detection of the watermark in the watermarked multimedia object. The limited applications for semi-blind and blind schemes include copyright protection in a court of law setting and copy-control.



a) Watermark Encoder: Watermark Embedding

b) Watermark Decoder: Watermark Detection

c) Watermark Decoder: Watermark Extraction

Figure 5.5: Watermark encoder and decoder

The encoder is a function that embeds the watermark data into the host multimedia object. The type and location of the data is dependent on whether the watermark is visible or invisible, whether the host object is audio, text, image, or video, and what compression scheme if any is used for the host object. For example, an invisible-robust watermark for an H.264 compression format may include a copyright message watermark embedded in the luminance DCT coefficients before the quantization phase. The decoder function is the opposite of the encoder and it can used for detection or extraction. An extraction scheme will extract the watermark bits from the watermarked object, such as a robust watermark. A detection scheme will detect if a watermark is present, such as a fragile watermark.

5.3. Security and Attacks

Intentional or unintentional modifications to the watermarked object have the potential to remove the watermark or obscure the watermark data, and thus make it difficult for the watermark to be successfully extracted or detected. Any modification to a watermarked object is known as an attack, whether it is intentional or unintentional. There are many forms of attacks such as geometric, signal processing, specialized attacks, cryptographic attacks, and system-

based attacks. The security of the watermark embedding algorithm depends on the watermark being able to withstand the attack either by making it too computationally difficult to remove or the removal causes too much distortion in the audio, image, or video. In the case of fragile or semi-fragile watermarks, being able to defend against watermark counterfeit attacks is another security measure to be considered.

Geometric attacks include rotation, scaling, translation, cropping, affine transforms, mosaicing, and other related image and video editing operations. These types of attacks are specifically for image and video watermarks. These attacks are meant to cause distortion in the watermarked object and thus, make it very difficult for the watermark to be detected or extracted.

Signal processing attacks include lossy compression, quantization, requantization, dithering, adaptive filtering, denoising, collusion attacks, and stochastic attacks involving watermark estimation and noise addition. These types of attacks either attempt to remove the watermark such as stochastic attacks or attempt to add additional signal noise to the watermarked object such as lossy compression.

A specialized attack requires direct knowledge of the watermark method. De-synchronization, luminance, and chrominance attacks are two such specialized attack methods. Cryptographic attacks use similar attacks in cryptography to breach the security of the watermarked object. A system based attack would be a watermark counterfeit attack.

5.4. Video Watermarking

5.4.1. Overview

Several papers in the literature have proposed many video watermarking methods for various video formats and video compression schemes. Many of the algorithms proposed center

around invisible watermarks since these types of watermarks deal with copyright protection and image authentication. Video watermarks have some similarities with image watermarks in that the watermark bits are embedded in an image. Though in the case of video watermarks, there are multiple images represented as frames that are sampled over different time periods. Thus, this adds additional computational complexity as watermark bits must be computed for different frames or in some cases in block-oriented compression schemes, different blocks. The additional computational complexity also increases the time it takes to embed and extract (or detect) the watermark.

5.4.2. H.264/MPEG-4 AVC

H.264/MPEG-4 AVC is the industry standard for video compression in many multimedia applications and thus, it is essential to provide copyright protection and protect intellectual property (IP) in H.264 compressed videos. Many watermarks in the literature have been proposed for the H.264 video compression. This remains a challenging task because of the high-coding efficiency and the redundancy removal during the encoding process. Robust video watermark data is generally embedded in redundant data that minimizes distortion and is robust against attacks. The efficient removal of the redundant data in the H.264 video compression makes this a challenging task.

5.4.2.1. Watermark Embedding and Extraction and Detection

Digital video watermarking methods for the H.264 compression format is classified into four methods depending on where the watermark is placed during video compression: before compression, before quantization, after quantization, and during entropy encoding. Also, any of

the four groups can take place on a video bit stream, during the decoding and encoding of a compressed stream, or on the compressed stream itself. Embedding the watermark in a video bit stream before compression can make the digital watermark itself sensitive to video compression and thus, cause distortion to the compressed video. Many of H.264 watermarking algorithms presented in the literature take place before quantization. The general structure involves embedding the watermark in the DCT coefficients in the luma and chroma components of the macroblock. This is the structure that will be used in this thesis. Embedding watermark data after quantization entails embedding the watermark data in the quantized transform coefficients. Watermarks can also be embedded during the entropy coding phase in the Context-adaptive variable length coding (CAVLC) mode or context-adaptive binary arithmetic coding (CABAC) mode.

Watermark extraction and detection take place after the video has been decompressed. Watermark detection involves detecting whether the watermark bits are present are not in a secret location using an optional key if one was used in the original embedding algorithm. Watermark extraction requires locating the watermark secret location and extracting the watermarked bits from the watermarked object.

5.4.2.2. Security and Attacks

The H.264 video watermark is designed to be robust against signal processing and hostile attacks. It is also designed to be hidden in a secret location that can only be accessible by authorized parties and it is generally designed to be imperceptible (invisible watermark). There is a large literature detailing various signal processing attacks such as the Gaussian noise, bit-rate reduction, contrast enhancement, frequency filtering, and non-linear filtering attacks that the

H.264 watermark should be robust to. Metrics that are looked at when evaluating the strength of the attack are the distortion caused by the attack and errors in the embedded watermark. An attack that can cause very little distortion is of higher quality. The H.264 watermark is designed with these metrics in mind.

5.5. Software Implementation

A watermark software implementation will need to take into account the video coding standard it is being developed for. The software implementation will need to consider performance trade-offs and capabilities such as robustness. Other features to consider will be execution time of the embedding and extraction (or detection) of the watermark and the amount of distortion that is introduced during the watermark embedding. Adding a watermark without affecting the bit-rate or video quality is a difficult task. Parameters that need to be looked at during a software implementation include the bit-rate and the peak signal-to-noise ratio (PSNR), the ratio that is used to measure the video quality. The watermark implementation must also be optimized to withstand the attacks it is designed for and this would include robust, semi-fragile, and fragile watermarks.

An example is the joint model (JM) reference software in the reference implementation for H.264 developed and maintained by the joint video team (JVT) of ISO/IEC MPEG & ITU-T VCEG (video coding experts group), the same team that is responsible for developing and maintaining H.264. The JM software includes an encoder and decoder. It also features watermark implementations that include robust, semi-fragile, and fragile watermarks. Throughout the literature and academia, the JM reference software is used to test robust watermark implementations and optimize the watermarks to withstand various attacks.

5.6. Hardware Implementation

Video watermarking hardware implementations are designed on custom ICs and FPGA boards for use with commercial applications that include TV broadcast surveillance systems, DVD videos, and video-based web applications. The intended goal of a hardware implementation is to produce an efficient implementation in real-time while maintaining low-power consumption and high-reliability. The implementation also needs to be low-cost as well.

The caveat for the above hardware assisted watermark implementations is that this requires hardware designing, thus putting a limit on programming and requiring changes in the hardware architecture.

5.7. Parallelization

5.7.1. Software Implementation on a CUDA-Enabled GPU

The literature regarding parallelization of video watermarking on the GPU is virtually non-existent. The literature either focuses on parallelizing image watermarks or video compression, such as H.264.

Parallelizing a video watermark on a CUDA-enabled GPU can be rather complex and computationally intensive and this is dependent on the video compression format and the digital watermarking algorithm. The digital watermark algorithm would have to support independent operations such as the watermark embedding after the calculation of the DCT coefficients during the integer transform, where each coefficient and embedded watermark data can done independently and hence, in parallel. Also, once the parallelizable video watermark operations have been offloaded to the GPU, a key feature that needs to be part of the video watermark is to minimize the data transfers between the CPU and GPU. Writing the CUDA code in such a way

where CUDA registers and shared memory are accessed effectively to reduce the number of global memory accesses is a factor to consider when writing CUDA code. A good operation to consider when dealing with very large matrices such as watermark embedding in the integer transform is to divide the matrices into smaller tiles so the matrices can fit into the shared memory, and thus, reduce the number of global memory accesses.

5.7.2. Hardware Implementation

Hardware implementations of digital watermarks is a new area being presented in the literature. Applications that can be of use for the hardware implementations include DVD, Blu-ray disc, cellular phones, and PDAs to name a few. One of the advantages of the hardware implementations is to offer real-time performance at low power. This is very useful for cell phones that are themselves lower-power devices. Many of the new devices coming to the market such as cell phones have multi-core or many-core chipsets. Implementing either a dedicated hardware in the chipset or adding hardware features to take advantage of the cores for digital video watermarking is a viable solution for reducing the overall execution time.

Kougianos et al. [20] presented a survey of recent hardware implementations of the digital video watermarks for FPGA and IC chipsets including GPUs. The authors in [26] discuss an early attempt for a real-time video watermarking on an NVIDIA GPU in 2005 where the watermark had to be mapped onto the GPU using graphics operations. The authors in [41] discuss a viable solution for a low-cost hardware implementation that can be combined with real-time software for digital video watermarks on low-powered devices such as cell phones. Today's cell phones are multi-core architectures and thus, this low-cost hardware implementation can take advantage of the multi-core structure.

CHAPTER 6

H.264/MPEG-4 AVC

6.1. Overview

A digital video is a representation of real world phenomena featuring spatial and temporal locality. The video is sampled at intervals to produce frames and each frame is made up of fields. In today's world, there are many video applications such as DVDs, TV broadcasting, cell phones cameras, video-based web applications, and video calling to name a few. Videos can take up a large amount of space, which can be a problem for storage and/or transmission. One solution is video coding, the compression and decompression of video. Video compression reduces the amount of data of a digital video prior to transmission and/or storage. Video decompression recovers a video from its compressed start back to its normal state.

There are many compression formats for audio, images, and video. This thesis will be focusing on video compression and more specifically, the H.264/MPEG-4 AVC format, otherwise known as H.264. The H.264 format is one of the most popular formats for compression, video recording, and distribution. H.264 is a highly efficient compression format compared with other formats and is used in many applications such as YouTube, Blu-ray discs, and HDTV broadcasts to name a few. H.264 is a data intensive task, especially for real-time video coding and thus, being able to offload the numerically intensive tasks to the GPU would decrease the amount of time for the video coding process.

Digital video watermark algorithmic structures are dependent on the video compression formats. Designing a good watermark algorithm requires sufficient knowledge of the video

compression format being used. H.264 has high complexity and compression efficiency, thus presenting a challenge for embedding the video watermark.

6.2. Concepts of Video Coding

Video coding is the process of compressing and decompressing a video. H.264 video compression or video encoding, is a block-oriented compression that is performed on videos prior to storage and/or transmission. H264 video decompression or video decoding is a block-oriented decompression that is performed after transmission or on video applications such as playing a movie from a DVD. H.264 data compression supports lossy and lossless compression, depending on the application requirements. Lossy compression achieves higher compression, at the expense of the decompressed video not being identical to the source video. Lossless compression produces a decompressed video that is identical to the source video, at the expense of the lower compression and thus, a smaller reduction in bits for the compressed form prior to storage and/or transmission.

H.264 is a specification only for encoding videos for video compression and decoding videos for decompression for storage and/or transmission. The video codec is the actual software or hardware implementation of a specific video compression format such as the H.264 format. A well known video codec for the H.264 format is the x264 video encoder. The encoded video is bundled with an audio stream encoded in an audio compression format such as advanced audio coding (AAC) in a multimedia container format such as MP4, which is the official container format for MPEG-4 audio and video. Multimedia container formats such as MP4 do have the option of containing different video and audio compression format besides H.264 and AAC.

6.2.1. Block Motion Compensation-Based Video Compression

H.264 is a block motion compensation-based video compression format that uses the macroblock as its processing unit for video compression. The macroblock is an MxN block of pixels, usually size 16x16, that represents a small region in a frame. Motion compensation prediction is the prediction of a current video frame using a past and/or future frame as reference with modeling of motion. This algorithmic technique is based on the fact that there is very little difference between one frame and the next so calculating the difference by subtracting the prediction from the current frame produces a residual, which allows for better compression. To produce the residuals of the frame, a candidate region of a past and/or future frame must first be selected. The criterion for selecting the candidate region is finding an MxN block region in the reference frame(s) that best matches an MxN block region in the current frame, this process being called motion estimation. The chosen block then is subtracted from the current block to form an MxN residual block known as block motion compensation. The offset between the chosen block and current block is a motion vector. The residual and motion vector is needed during the encoding and decoding process.

6.2.2. H.264 Syntax and Structure

The structure of the H.264 syntax is setup as a hierarchical organization consisting of units for representing compressed video. The top level is a series of packets, network adaption layer (NAL) units, that include two parameter sets, sequence parameter sets (SPS) and picture parameter sets (PPS), as control parameters needed by the decoder. The NAL units contain coded frames or fields, known as access units, that are represented by one or more slices. Each slice consists of a slice header and data, the data being a coded 16x16 macroblock consisting of

compressed data. The slice also consists of skip macroblocks, which are indicators of macroblock positions containing no data. The coded macroblock itself contains the macroblock type, prediction information, the coded block pattern (CBP), quantization parameters (QP), and residual data.

A frame is made up of luma samples and two corresponding chroma samples. The top and bottom field make up the frame. A luma sample corresponds to luminance, the light intensity of the brightness component of a frame. The chroma sample corresponds to chrominance, the color difference component. There are three color difference components, each one being the difference between red (R), green (G), or blue (B) and the luminance. The RGB color space can produce any chrominance of an image sample using the R, G, and B additive primaries. Based off the human visual model, the human eye is more sensitive to light and hence, the luminance (Y), red chroma (Cr), and blue chroma (Cb), otherwise known as Y:Cr:Cb, is used to reduce the amount of data in the chrominance components for better compression. The Y:Cr:Cb is used in the construction of a macroblocks. For example, a 16x16 macroblock contains a 16x16 region for the luminance component and two 8x8 regions for the Cr and Cb sample.

There are three YCrCb sampling formats: the 4:4:4, 4:2:2, and 4:2:0 sampling formats with the 4:2:0 sampling format being the most popular. In the 4:4:4 sampling format, for every four pixels sampled, there are four Y, Cr, and Cb samples each, thus preserving full resolution for each component. In the 4:2:2 sampling format, for every four pixel samples, there are four Y, but two Cr and Cb components, thus producing the same vertical resolution for the luminance and chrominance components but the chrominance components having half the horizontal resolution as the luminance. In the 4:2:0 sampling format, for every four pixel samples, there are

four Y, but 1/4 the Cr and Cb components, thus producing half the vertical and horizontal resolution of the luminance for the Cr and Cb components. The 4:4:4 sampling format is used in high-end film scanners and cinematic postproduction applications. The 4:2:2 sampling format is used in high-quality color reproduction and high-end digital video format applications. The popular 4:2:0 sampling format is used in consumer applications such as DVD video and storage, Blu-ray disc, and digital television to name a few.

6.2.3. Profiles and Levels

The H.264 format specifies a set of tools and constraints needed for the encoding and decoding process.

H.264 profiles define the set (or subset) of tools to define the features and capabilities of encoding and decoding implementations. For example, a coded video using the high profile, a subset of the main profile, can only be encoded using some or all of the tools in the high profile. The decoder follows suit. The features and capabilities of the main profile would be defined for high-definition digital TV broadcasts and Blu-ray disc storage.

The H.264 level defines a set of constraints that define an upper limit on the amount of data that the decoder can accept. The types of data the constraints apply to include frame size, processing rate, and working memory. For example, a Level 3 decoder can accept bit rates less than 10Mbps and resolutions up to 720×576 at a frame rate of 25 frames per second (fps). Also, the Level 3 decoder can decode any Level decoder that is less than and up to the Level 3 decoder, such as a Level 2.2 decoder for example.

6.3. Encoding and Decoding

The H.264 codec encodes a series of frames or fields using a lossy compression format to produce a compressed H.264 bitstream that can be stored and/or transmitted. The encoding process consists of prediction, transformation, quantization, and entropy encoding. The decoder performs the reverse of the encoder to produce a reconstructed version of the original video sequence that is not identical to the original video sequence due to the lossy compression format. The decoding process consists of entropy decoder, rescaling, inverse transformation, and reconstruction.

6.3.1. Encoding Process

The encoding process consists of a series of steps to produce a compressed H.264 bitsream: prediction, transformation, quantization, and entropy encoding as shown in figure 6.1.

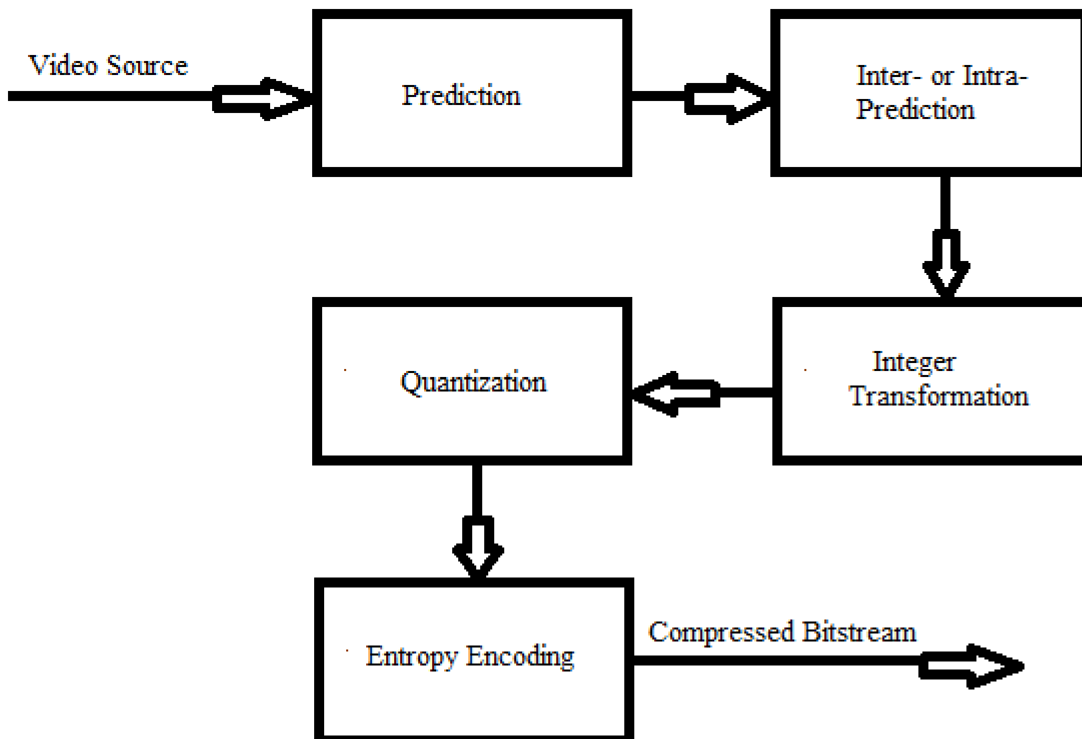


Figure 6.1: The encoder stages

The prediction model consists of two types of predictions: intra prediction and inter prediction. In intra prediction, the macroblock sizes used are 16x16 and 4x4. A predicted macroblock is formed by extrapolating previous values of the neighboring coded-pixels in the current frame to produce a predicted macroblock that is an approximation of the original current macroblock. The predicted macroblock is then subtracted from the original current macroblock to form a residual block that is the same size as the predicted and current macroblock.

In inter-prediction, the macroblock size rages from 16x16 to 4x4. As discussed in section 6.2.1, motion estimation is used to determine the best fit macroblock from previous coded frames or fields that best matches the current macroblock in the current frame or field. The previous coded frames can be 1 or 2 frames from past or future frames. After a best match is found, the current macroblock is predicted from the best fit macroblock. This prediction is subtracted from the current macroblock, known as motion estimation, to form a residual.

The integer transformation step consists of a 4x4 or 8x8 integer transform that is a scaled approximation to the discrete cosine transform (DCT). The integer transform is applied to the residual samples to produce a set of luma and chroma coefficients each in their own MxN block. These coefficients are a set of weighted values for the set of standard basis patterns, a set of 4x4 or 8x8 blocks of cosine functions. When the set of standard basis patters are multiplied by the transform coefficients in the decoder phase, this produces the residual samples.

To reduce the size of the transform block, the transform coefficients are quantized. The coefficient values are divided by a quantization parameter (QP) and rounded to the nearest integer. For insignificant values, this reduces them to zero.

The last phase of encoding is the entropy encoding stage. During this stage, the following values and parameters must be encoded: quantized transform coefficients, the

prediction mode used e.g. intra prediction or inter-prediction and motion vectors, header information, information on the compression tools used, and identifiers and delimiting codes. The values and parameters, represented as symbols, are converted into binary codes. The conversion methods used are variable length coding and/or arithmetic coding to produce short binary codes for commonly used values and longer binary codes for less commonly used values. This produces an encoded bitstream, a compact binary representation of the values and parameters, and can be stored and/or transmitted.

6.3.2. Decoding Process

The decoding process consists of a series of steps to produce a decompressed video bitstream: entropy decoder, rescaling, inverse transform, intra-prediction or inter-prediction, and reconstruction, as shown in figure 6.2.

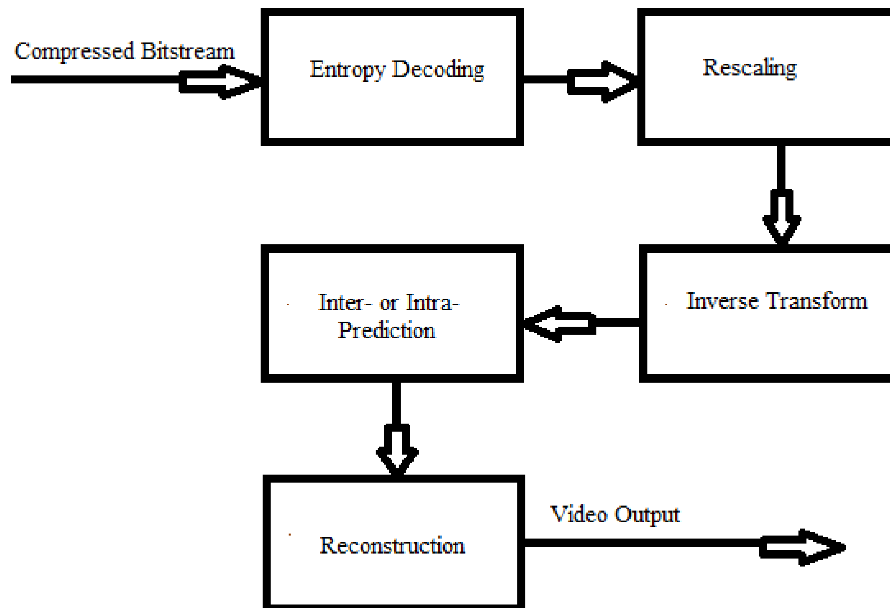


Figure 6.2: The decoder stages

The entropy decoder decodes the encoded bit stream to retrieve the quantized transform coefficients, the prediction mode used, the header information, information on the compression tools used, and identifiers and delimiting codes.

The rescaling process multiples the quantized transform coefficients by a QP to produce coefficients that are similar to the original coefficients produced by the integer transformation in the encoder. The rescaled values are not exact since quantization in the encoder is a non-reversible process due to the floating point values that are rounded to integers and the 0 values.

The inverse transform process multiples the standard basis patterns with the set of weighted values, the rescaled coefficients to produce the MxN residual blocks.

The last phase of decoding is the reconstruction phase. The decoder forms a prediction using either inter-prediction or intra-prediction. In inter-prediction, the predicted MB is formed from previously decoded frame(s). In intra-prediction, the predicted MB is formed from previously decoded MBs in the current frame. In both cases, the prediction is identical to the prediction formed in the encoder. The prediction is then added to the decoded residual to reconstruct a decoded MB.

6.4. Software Implementation

The H.264 compression standard defines the syntax of the compressed video and decoding methods. There is no encoding standard defined and there is no reference software implementation. Encoding standard software implementation and the software implementation of the compression and decoding methods is left to the video codec manufacturers.

The H.264 video codec is the actual software or hardware of the H.264 compressed standard. A well known video codec for the H.264 format is the x264 video encoder. The

encoded video is bundled with an audio stream encoded in an audio compression format such as advanced audio coding (AAC) in a multimedia container format such as MP4, which is the official container format for MPEG-4 audio and video. Multimedia container formats such as MP4 do have the option of containing different video and audio compression formats besides H.264 and AAC. Also, with regards to the implementation of the encoding phase, the video codec manufacturers will generally implement the video encoder that mirrors the decoding steps.

H.264 is an industry standard and thus, there is a large number of H.264 video codecs to choose from. The key features to keep in mind when deciding what codec to use are the trade-offs between compression performance and computational complexity. This is depending on feature implementations in the video codec, the coding performance, coding options, and coding modes. Some implementations, such as the x264, offer a superb coding implementation of the encoding phases but do not implementation the decoder. Fortunately, an x264 encoded video can be decoded by the FFmpeg program decoder. The JM reference software, implemented by the JVT team, the same team responsible for maintaining the H264 standard, implements all the h264 features but is not a practical application for industry use. Whether using free codecs or proprietary codecs for academia or industrial use, different mode selections will offer different compression performance and computational complexity. Also, different applications require different limits on the coded bitrates. Therefore, for a codec implementation to be of practical use, a bitrate control algorithm will need to be implemented that puts an upper and lower limit on bitrate while minimizing distortion in the H.264 compressed video. Such applications would include internet streaming such as YouTube videos, HDTV broadcasts, video conversations on cell phones, and Blu-ray discs.

6.5. Parallelization

6.5.1. Software Implementation on a CUDA-Enabled GPU

The software implementations of parallelizing the H.264 process usually attempt to parallelize the motion estimation since that is the most computationally intensive part of the encoding phase taking as much as 80% of the total execution time depending on how the motion estimation was implemented. The integer transform is also parallelized since it is a matrix of independent operations.

In the literature, there are many studies showing success and results of the parallelization process for H.264 encoders and decoders. Many show at least 20% speedup or more compared to the original H.264 implementation.

Two such codecs that have libraries that can be combined with the CUDA C libraries are the FFmpeg and x264 libraries. When writing the H.264 encoding and decoding programs using the libraries, the codec operations can be parallelized depending on the nature of the data. CUDA can be used to assist with the parallelization.

NVIDIA has released a CUDA-based video decoding library known as NVCUVID, the NVIDIA CUDA Video decoder. The library is part of the GPU computing SDK and is used to decode MPEG-2, VC-1, and H.264 video bit streams on video processors and CUDA-enabled GPUs. This not a full codec since it only decompresses compressed data, not compress data.

6.5.2. Hardware Implementation

NVIDIA has produced a H.264 hardware-accelerated video encoder that uses dedicated hardware on Kepler-class NVIDIA GPUs to assist with H.264 encoding. The dedicated hardware is designed to speed up the encoding and reduce power consumption while at the same

time, freeing up the CUDA and CPU cores from the encoding process so the CUDA and CPU cores can be used for other tasks. The NVIDIA Encoder (NVENC) API allows software developers to access the dedicated hardware.

CHAPTER 7

THE PROPOSED VIDEO COMPRESSION AND WATERMARK IN GPU COMPUTING ENVIRONMENT

7.1. H.264 Video Compression Algorithm

The H.264/MPEG-4 AVC compression format is currently the standard format for video coding and compression. It achieves a much higher compression compared with other formats. It supports many additional options for encoding and decoding such as different modes for inter- and intra-prediction, variable block size, new integer transform design, and quarter-pixel precision for motion compensation to name a few. This multitude of options allows better compression and more flexibility. The H.264 compression format has a large computer base due to it being used in Adobe flash, YouTube, mobile Android devices and other mobile devices, and browser support. H.264 is used across a broad application spectrum. All these features and high compression come at the expense of an increase in computational complexity and execution time.

To decrease the overall computational complexity and execution time of the H.264 compression and decompression, a massively parallel computing device, such as a GPU, can be used to assist with this reduction in time and complexity. This is of course, dependent on data that is parallel in nature since the GPU is optimized for data parallelism, not data that is serial in nature. The CUDA-enabled GPU offered by Nvidia supports a data-parallel programming computation model that can be useful for non-graphics applications such as H.264 video compression. The integer transform uses independent operations to compute its matrix and thus, can be parallelized in the GPU. A kernel function will be created to offload the integer transform computation onto the GPU.

7.2. H.264 Digital Video Watermark Embedding Algorithm

The watermarking algorithm will embed the watermark data into the DCT coefficients during the encoding phase. The subset of coefficients in the H.264 integer transform stage that will be modified will be the coefficients that correspond to the lower left portion of the video stream. Since the H.264 integer transform is being parallelized, this implies the watermark embedding operations will also be parallelized. To optimize the parallelization, the independent matrix operations will need to take advantage of the shared memory, local registers, and constant memory in the GPU to reduce the global memory accesses. In order to accomplish this, since the matrices are too large to be stored in the shared memory, the matrices will need to be divided up into tiles so the matrix operations can fit perfectly into the shared memory and thus, reduce the global memory accesses. Also, the local register use in conjunction will the thread count and block count will be taken into consideration to maximize the amount of threads per block and the blocks per SM that will be in use during the matrix operations.

CHAPTER 8

EXPERIMENTAL RESULTS

8.1. Experimental Setup

The experimental setup regarding the hardware, GPU, system software, and programming software is listed in the below tables. The program was executed using the GCC compiler and NVIDIA's CUDA compiler using the MSYS shell that mimics the Linux bash shell and MinGW-64 runtime environment that provides a runtime environment similar to Linux. The FFmpeg program and libraries was used for the video compression and watermarks in the test videos. The x264 library was used specifically for the video compression in FFmpeg.

The proposed algorithm was implemented using the hardware setup summarized in table 8.1. The specifications of the CUDA-enabled GPU are listed in table 8.2. The system software the program was tested on is listed in table 8.3. The encoder and decoder software and libraries and compilers are listed in table 8.4.

Table 8.1 Test Hardware Setup

System Hardware	Details
Platform:	Dell XPS 17 - L702X Laptop
CPU Processor:	Intel Core i7-2760QM CPU @ 2.40GHz
RAM:	16.0 GB (2x8 GB) DDR3 1600 MHz (PC3 12800) Laptop Memory
GPU processor:	GeForce GT 555M

Table 8.2 NVIDIA GeForce GT 555M Properties

Parameters	Values

GPU processor:	GeForce GT 555M
Driver version:	332.21
CUDA Cores:	144
Core clock:	590 MHz
Shader clock:	1180 MHz
Memory data rate:	1800 MHz
Memory interface:	192-bit
Memory bandwidth:	43.20 GB/s
Total available graphics memory:	10955 MB
Dedicated video memory:	3072 MB DDR3
System video memory:	0 MB
Shared system memory:	7883 MB
Video BIOS version:	70.26.40.00.02
IRQ:	16
Bus:	PCI Express x16 Gen2

Table 8.3 Test System Software Setup

System Software and Drivers	Details
Operating System:	Windows 7 Ultimate, 64-bit (Service Pack 1)
Run Time Environment:	MinGW-w64
Shell	MSYS
Parallel Computing Platform	CUDA 5.5

DirectX version:	11.0
Direct3D API version:	11
Direct3D feature level:	11_0
GPU Driver version:	332.21
Video BIOS version:	70.26.40.00.02

Table 8.4 Test Software Setup

Program Software and Libraries	Details
Multimedia Development Platform	FFMPEG
Encoding Libraries	x264
Compilers	GCC and NVCC
Profilers	NVIDIA Visual Profiler

8.2. Experiments

Videos of different lengths were recorded using a cell phone. Table 8.5 summarizes the video properties.

Table 8.5 Video Properties

Test Video Name	Video Length (s)	Dimensions WxH	Size in MB	File Type	Frames
Short Video	10	1920x1080	21.1 MB	MP4	310
Midsize Video	92	1920x1080	187 MB	MP4	2755

Long Video	301	1920x1080	614 MB	MP4	9025

The videos were watermarked using a .png file created in MS paint. The .png file properties are shown in table 8.6 and the watermark itself is shown in figure 8.1.

Table 8.6 Watermark Properties

Dimensions WxH	Size in KB	File Type
200x200	6.98	PNG

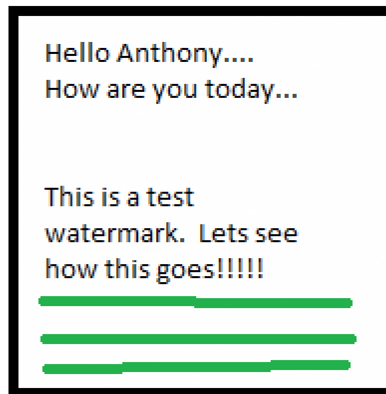


Figure 8.1: A watermark logo used during the watermark insertion

The videos were compressed during the program execution with the watermark inserted before the encoding finished. The videos were than decompressed for video playback. Since this is a visible watermark, no watermark detection or extraction is needed.

8.3. Results

The encoding and decoding results for the frame rate and bit rate on the CPU and GPU are shown in table 8.7. The FFmpeg program was used in the MSYS shell to calculate the data. Table 8.8 displays the watermark execution times on the CPU and GPU and speedups obtained.

Table 8.7 Frame rate and bitrate calculated for CPU and GPU

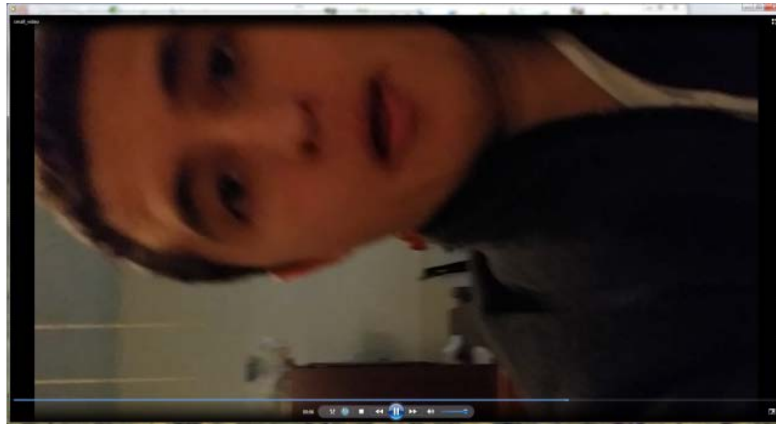
Test Video Name	Frame Rate on input file	Frame Rate on output file	Bitrate on input file in kb/s	Bitrate on output file in kb/s
Short Video	30 fps	19	16536	5245.1
Midsize Video	30 fps	17	16989	5251.7
Long Video	30 fps	16	17098	5253.9

Table 8.8 Watermark Execution Time (s)

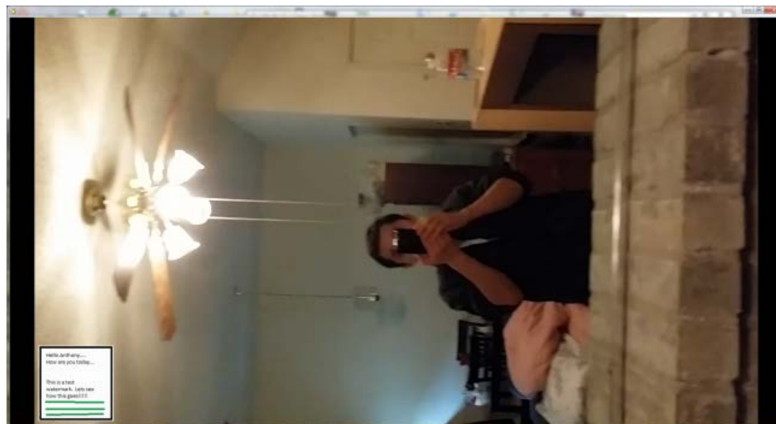
Test Video Name	Video Length	CPU	GPU	Speed up	Speedup Percentage
Short Video	10	10.03	9.06	1.10706401	9.7%
Midsize Video	92	159.71	148.57	1.07498149	7%
Long Video	301	486.09	450.62	1.07871377	7.3%

As can be seen from the table, a small speedup was gained in the GPU. Looking at the raw data from the profiler, the data transfers between the CPU and GPU were the more expensive parts in terms of execution time and this certainly did lower the speedup gained.

Figures 8.2 through 8.4 compare the original H.264 frames and watermarked H.264 frames of short, midsize, and long videos.



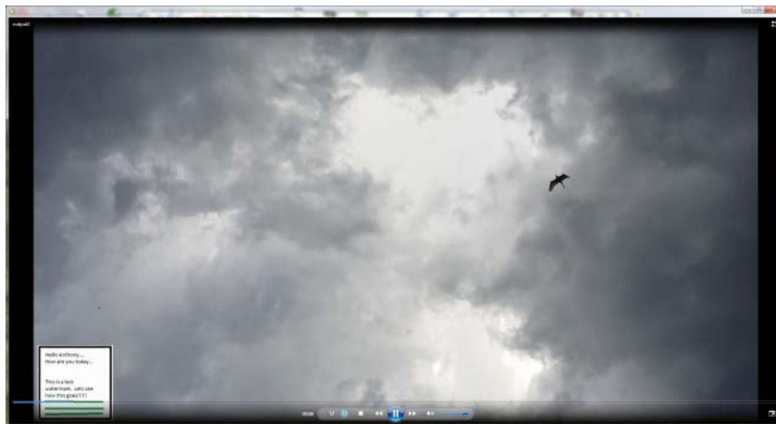
(a) Original H.264 Frame



(b) Watermarked H.264 Frame

Figure 8.2: Comparison of the original H.264 and watermarked H.264 frame in the short video

(a) Original H.264 Frame



(b) Watermarked H.264 Frame

Figure 8.3: Comparison of the original H.264 and watermarked H.264 frame in the midsize video

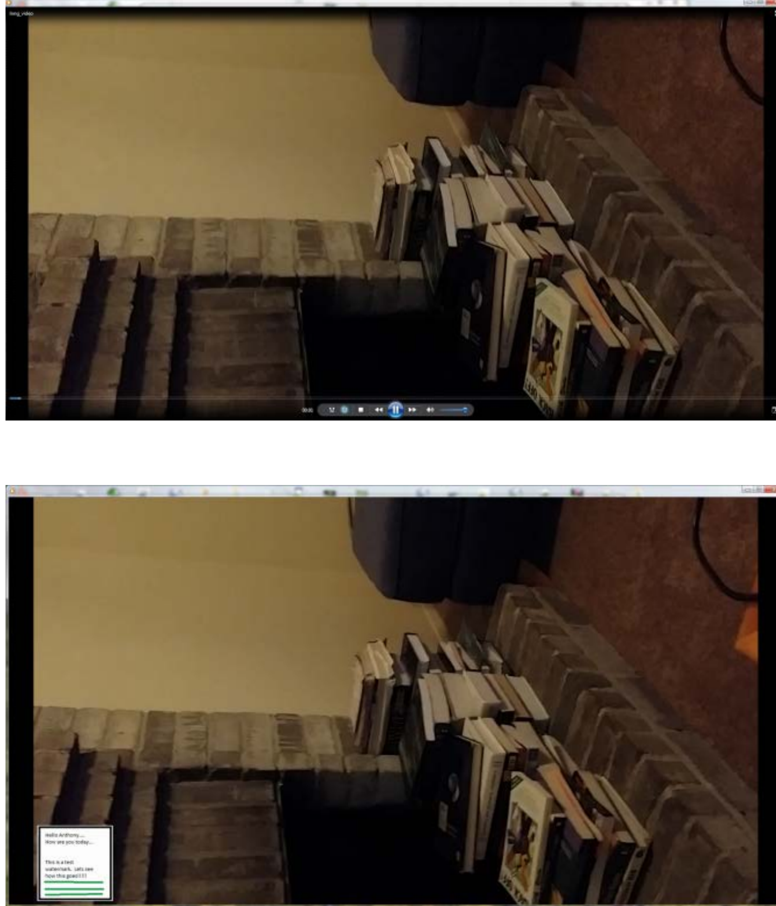


Figure 8.4: Comparison of the original H.264 and watermarked H.264 frame in the long video

8.4. Discussion of Results

The results obtained from the experiment were a bit disappointing considering the speedup gained. One possible reason is the data transfers between the CPU and GPU during the program execution. According to the profiler, this did add some execution time to the program execution. Another reason deals with the fact that the H.264 format is an efficient compression format which implies high computational complexity and difficulty of watermark insertion since there are less insignificant values in the H.264 compressed file. Also, the motion estimation is the most compute intensive part of the video compression and there are a lot of dependencies involved with the motion vector predictions during the motion estimation phase. These issues

present a challenge in obtaining significant speeds on the GPU. Another possible reason for the low speedup is the GPU test setup used. The GPU used was a mobile chipset not meant for heavy duty computations and so it is possible had the benchmarks been performed on a desktop GPU, the results might have been a bit better. Future research will need to be conducted on a desktop GPU.

8.5. Comparison With Related Research

With regards to H.264 video compression on the GPU, most authors obtained a modest speedup at around 20% and that does not include watermark insertion. In [35], the authors obtained a speedup of around 20%.

In [25], the authors implemented a digital image watermark on a GPU and obtained a 6% speedup. There were no digital video watermarks implementations on the GPU in the literature to be used for comparison.

This research combined digital video watermarks and H.264 video compression implementations on the GPU. The literature was non-existent for digital video watermark in the H.264 video compression format. Therefore it is a bit difficult to compare these results with others. The results obtained on the compression and watermark insertion, a 8% speedup on average, were a bit lower compared with other H.264 compression implementations on the GPU but then again, their implementations did not include watermark insertion. In [25], the method for watermark insertion is similar to this work's though it is for image watermarking, not video watermarking. The author did obtain a low speedup, which is similar to these results.

CHAPTER 9

CONCLUSION AND FUTURE RESEARCH

9.1. Summary

Today's GPUs are massively parallel computing devices that can assist the CPUs for general computations in heterogeneous CPU-GPU systems. For the GPUs to be of use, the data that is to be offloaded to the GPU has to be parallel in nature, not serial.

One of the non-graphics applications that is highly computationally intensive is digital video watermarks in H.264 compressed videos. Digital video watermarking is the process of embedding data in an original message, such as the H.264 compressed video stream for protection of IP and enforce copyright protection for businesses and industry. Digital video watermarks are used in many applications related to broadcast monitoring, source tracking, and copyright protection such as HDTV broadcasts, DVD and Blu-ray discs, and internet streaming to name a few. The visible watermark can be used as a company logo to signify the owner of the video content. The invisible watermark can be detected or extracted using different algorithmic techniques. The embedding and extraction process can also be done in real-time as well, which is more computationally demanding as opposed to non-real time applications.

The H.264 process offers efficient compression at the expense of computational complexity and since the digital watermark is highly dependent on the compressed format, the H.264 digital video watermark is a highly computationally intensive task, especially in real-time applications. To assist with the computational complexity and thus, lower the execution time, some of the highly computational tasks in the H.264 digital video watermark can be offloaded to the GPU. There is substantial literature showing success in H.264 video watermarks being implemented on a CUDA-enabled GPU. If the tasks contain dependent data and operations, the

algorithms can be redesigned to implement independent operations as in the case of the motion estimation [30].

In this thesis, a generic video watermark for the H.264 compressed domain was implemented on a CUDA-enabled GPU. The codec libraries used were the FFmpeg for the multimedia development platform and the x264 for the encoding libraries. The operations that were offloaded to the GPU and parallelized were the integer transform coefficient calculations and the placement of the watermark data in the integer transform coefficients. The watermark itself is a visible watermark. The aim of the thesis was to offload the H.264 watermark operations onto the CUDA-enabled GPU to reduce execution time, and thus, the H.264 watermark algorithm was not designed for robustness against attacks or for rate-distortion optimization in the H.264 compressed video.

9.2. Conclusions

The results obtains from the benchmarks were a bit disappointing. A possible reason for this could be that the execution time did include the encoding and decoding phase and the only operations that were parallelized were the DCT coefficient calculations during the encoding phase and the watermark embedding in the DCT coefficients. Motion estimation was not parallelized nor were any operations in the video decoder. Also, the GPU test setup was on GPU mobile chipset that is not meant to handle heavy duty tasks and thus, it is possible that the benchmarks might have shown better results on a desktop GPU setup.

9.3. Future Research

Future research will be conducted on optimizing the algorithm for CUDA by reducing the global memory accesses and increasing the use of shared memory and registers. Mapping the motion estimation of the H.264 video compression to the GPU by removing the motion vector prediction (MVP) dependencies to reduce the computation time and observe the results it may have on the watermark algorithm. It may require the watermark algorithm be redesigned and this redesign may reduce the execution time. Also, test other watermark implementations such as inserting watermark data on H.264 compressed bitstreams. This insertion would have to take place in the I-Frames. Also, inserting watermark data on video streams before compression. All the tests will need to be conducted on a desktop GPU to observe any additional performance gains compared to a mobile GPU.

REFERENCE LIST

[1] Mohanty S. P., “A Secure Digital Camera Architecture for Integrated Real-Time Digital Rights Management”, Journal of Systems Architecture (JSA), Volume 55, Issues 10-12, October-December 2009, pp. 468-480.

[2] J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., and Purcell, T. "A Survey of General-Purpose Computation on Graphics Hardware". Computer Graphics Forum, volume 26, number 1, 2007, pp. 80–113.

[3] I. Buck. "Brook Specification v0.2". October 2003.

[4] NIVIDA. (2014). CUDA Parallel Computing Platform. Retrieved from http://www.nvidia.com/object/cuda_home_new.html.

[5] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P. "Brook for GPUs: Stream Computing on Graphics Hardware". In: ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2004. Volume 23 Issue 3, August 2004, pp. 777-786.

[6] Mark, W. R., Glanville, R. S., Akeley, K., and Kilgard, M. J. "Cg: A System for Programming Graphics Hardware in a C-Like Language". In ACM SIGGRAPH 2003 Papers, pp 896-907, 2003.

[7] Tarditi, D., Puri, S., and Oglesby, J. "Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses". In Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 325.335, 2006.

[8] NVIDIA. NVIDIA CUDA ZONE. 2014. Retrieved electronically February 1, 2014 from https://developer.nvidia.com/category/zone/cuda-zone.

[9] NVIDIA. "CUDA C Programming Guide, Version 6.0". NVIDIA Corporation, Ed. February 2014.

[10] NVIDIA. CUDA Toolkit Documentation v6.0. 2014. Retrieved electronically February 1, 2014 from http://docs.nvidia.com/cuda/index.html.

[11] Munshi, A. "The OpenCL Specification, Version: 1.1". Khronos OpenCL Working Group. June 1, 2011.

[12] Intel. Getting Started With OpenCL™ Applications. 2014. Retrieved electronically February 1, 2014 from http://software.intel.com/en-us/vcsource/tools/opencl.

[13] AMD. OpenCL™ Zone. 2014. Retrieved electronically February 1, 2014 from http://developer.amd.com/resources/heterogeneous-computing/opencl-zone/.

[14] NVIDIA. OpenCL. 2014. Retrieved electronically February 1, 2014 from https://developer.nvidia.com/opencl.

[15] Kirk, D. B., and Hwu, W. W. "Programming Massively Parallel Processors: A Hands-On Approach". 2nd ed. Waltham: Elsevier, 2013. Print.

[16] Farber, R. "Application Design and Development". Waltham: Elsevier, 2011. Print.

[17] Kim, H., Vuduc, R., Baghsorkhi, S., and Choi J., Hwu W. ″Performance Analysis and Tuning for General Purpose Graphics Processing Units (GPGPU)". Synthesis Lectures on Computer Architecture, Vol. 7, No. 2. (2012).

[18] General-Purpose Computation on Graphics Hardware. 2014. Retrieved on February 1, 2014 from http://gpgpu.org/.

[19] Mohanty, S. P., Pati, N., and Kougianos, E. "A Watermarking Co-Processor for New Generation Graphics Processing Units". In: Proceedings of the 25th IEEE international conference on consumer electronics (ICCE); 2007. p. 303–4.

[20] Mohanty, S. P., Kougianos, E., and Ranganathan, N. “VLSI Architecture and Chip for Combined Invisible Robust and Fragile Watermarking”, Computers Digital Techniques, IET, Vol. 1(5), pp. 600–611, 2007.

[21] Kougianos E., Mohanty S. P., and Mahapatra, R. N. "Hardware Assisted Watermarking for Multimedia". In: Proceedings of Computer and Electrical Engineering. Volume 35, Issue 2, March 2009, pp. 339–358.

[22] Maes, M., Kalker, T., Linnartz, J.-P., Talstra, J., Depovere, G., Haitsma, J. "Digital Watermarking for DVD Video Copyright Protection". IEEE Signal Process Mag 2000;17(5):47–57.

[23] Alattar, O.M., and Alattar, A.M. "A Fast Hierarchical Watermark Detector for Real-Time Software or Low-Cost Hardware Implementation". In: Proceedings of the IEEE international conference on image processing (ICIP); 2005. p. 973–6.

[24] Lin, C., Zhao, L., and Yang, J. "A High Performance Image Authentication Algorithm on GPU with CUDA". I.J. Intelligent Systems and Applications, Vol. 3, No. 2, March 2011, pp. 52-59.

[25] García-Cano, C. E., Rabil, B. S., and Sabourin, R. "A Parallel Watermarking Application on a GPU". Congreso Internacional de Investigación en Nuevas Tecnologías Informáticas - CIINTI 2012. September 14, 2012.

[26] Shieh, C.-S., Huang, H.-C., Wang, F.-H., and Pan, J.-S. "Genetic Watermarking Based on Transform-Domain Techniques". Pattern Recognition, Volume 37, Issue 3, March 2004, pp. 555-565.

[27] García-Cano, E., and Rodríguez, K. "A Parallel PSO Algorithm for a Watermarking Application on a GPU". Computación y Sistemas, Vol. 17, No. 3, July-September, 2013, pp. 381-390.

[28] Vihari, P., and Mishra, M. "Image Authentication Algorithm on GPU". 2012 International Conference on Communication Systems and Network , 2012, p. 874 - 878.

[29] Brunton, A., Zhao, J. "Real-Time Video Watermarking on Programmable Graphics Hardware". In: Proceedings of Canadian conference on electrical and computer engineering; 2005. p. 1312–5.

[30] Youngsub, K., Youngmin, Y., and Soonhoi, H. "An Efficient Parallel Motion Estimation Algorithm and X264 Parallelization in CUDA". Design and Architectures for Signal and Image Processing (DASIP), 2011 Conference on , vol., no., pp.1,8, 2-4 Nov. 2011.

[31] Wei-Nien, C., and Hsueh-Ming, H. H. "264/AVC Motion Estimation Implementation on Compute Unified Device Architecture (CUDA)". Multimedia and Expo, 2008 IEEE International Conference on, June 23 2008-April 26 2008, p. 697-700.

[32] Rodriguez, R., Fernandez-Escribano, G., Claver, J.M., and Sanchez, J. L. "Accelerating H.264 Inter Prediction in a GPU by using CUDA", Proc. IEEE Int. Conf. Consum. Electron., pp. 463-464 2010.

[33] Dong-Kyu, L., and Seoung-Jun, O., "Variable Block Size Motion Estimation Implementation on Compute Unified Device Architecture (CUDA)," Consumer Electronics (ICCE), 2013 IEEE International Conference on , vol., no., pp.633,634, 11-14 Jan. 2013.

[34] Rodriguez-Sanchez, R., Martinez, J. L., Fernandez-Escribano, G.; Claver, J. M., and Sanchez, J. L., "Reducing Complexity in H.264/AVC Motion Estimation by using a GPU," Multimedia Signal Processing (MMSP), 2011 IEEE 13th International Workshop on , vol., no., pp.1,6, 17-19 Oct. 2011.

[35] Nan, W., Mei, W., Huayou, S., Ju, R., and Chunyuan, Z. "A Parallel H.264 Encoder with CUDA: Mapping and Evaluation," Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on , vol., no., pp.276,283, 17-19 Dec. 2012.

[36] Bocheng, L., Qingkui, C. "Implementation and Optimization of Intra Prediction in H264 Video Parallel Decoder on CUDA," Advanced Computational Intelligence (ICACI), 2012 IEEE Fifth International Conference on , vol., no., pp.119,122, 18-20 Oct. 2012.

[37] NVIDIA VIDEO CODEC SDK. 2014. Retrieved electronically February 1, 2014 from https://developer.nvidia.com/nvidia-video-codec-sdk.

[38] Lin, D., Xiaohuang, H., Quang, N., Blackburn, J., Rodrigues, C., Huang, T., Do, M.N., Patel, S.J., and Hwu, W.-M.W. "The Parallelization of Video Processing". Signal Processing Magazine, IEEE , vol.26, no. 6, pp. 103- 112, November 2009.

[39] Cheung, N.-M., Fan, X., AU, O.C., and Kung, M.-C. "Video Coding on Multicore Graphics Processors". Signal Processing Magazine, IEEE , vol.27, no.2, pp.79,89, March 2010.

[40] Datla, S., and Gidijala, N. S. “Parallelizing Motion JPEG 2000 with CUDA”. Computer and Electrical Engineering, ICCEE '09. Second International Conference on, pp. 630-634, 2009.

[41] Obukhov, A., and Kharlamov, A. “Discrete Cosine Transform for 8x8 Blocks with CUDA,” NVIDIA white paper, 2008.

[42] Xiao, Z., and Baas, B. M. "A 1080p H.264/AVC Baseline Residual Encoder for a Fine-Grained Many-Core System". IEEE Trans. Circuits Syst. Video Technol., vol. 21, no. 7, pp. 890-902, 2011.