Real-time GPU Implementations of Image/Video Spatial Resolution Upconversion and Video Deinterlacing

REAL-TIME GPU IMPLEMENTATIONS OF IMAGE/VIDEO SPATIAL RESOLUTION UPCONVERSION AND VIDEO DEINTERLACING

BY

JIE CAO, B.Sc.

A THESIS

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

Master of Applied Science (2010) McMaster University

(Electrical & Computer Engineering) Hamilton, Ontario, Canada

TITLE:	Real-time GPU Implementations of Image/Video Spatial Resolution Upconversion and Video Deinterlacing
AUTHOR:	Jie Cao B.Sc., (Electronic Engineering) Fudan University, Shanghai, China
SUPERVISOR:	Dr. Xiaolin Wu
NUMBER OF PAGES:	xii, 56

To my beloved family

Fukang Cao and Jianhua Du

Abstract

In this thesis, we reexamine the classical problems of image/video spatial resolution upconversion and video deinterlacing with an aim to develop real-time, adaptive solutions. The research of this thesis is important because most video applications require real time throughput. We study the use of GPU (Graphics Processing Unit) technology for high throughput video interpolation and deinterlacing. The main technical challenge is how to fully utilize the processing power and parallel architecture of GPU to maximize the throughput of upconversion and deinterlacing without compromising the visual quality of the resulting videos. To achieve the goal we develop a GPU-friendly two-pass directional image/video resolution upconversion algorithm and present a GPU implementation of the method, using the NVIDIA CUDA (Compute Unified Device Architecture) technology. We also devise a GPU-motivated motion-adaptive deinterlacing algorithm and develop a CUDA-based implementation of the algorithm. To strike a balance between performance and complexity, we discuss the techniques of adapting the computations in motion detection and adaptive directional interpolation to the GPU architecture for maximum video throughput possible. Experimental results demonstrate that using a mid-range GPU card, our CUDA-based implementations offer real-time solutions for image/video spatial resolution upconversion and video deinterlacing.

Acknowledgements

I would like to take this opportunity to thank all those who have made the completion of this thesis possible.

First and foremost, I would like to express my sincerest appreciation to my supervisor, Dr. Xiaolin Wu. It is an honor and a pleasure to work with him. His guidance, encouragement and keen insights are highly appreciated and will always be remembered. The knowledge acquired from Dr. Wu is indispensible and will be tremendously beneficial in my future career.

I would like to thank to my examiners, Dr. Jian-Kang Zhang and Dr. Shahram Shirani for their time reviewing my thesis and providing valuable feedbacks. My special thanks to Cheryl, Cosmin, Terry for their friendly assistance and expert technical support.

Furthermore, I wish to thank my colleagues and friends Xiaohan, Xiangjun, Mingkai, Ying, Heng, Yong, Xiao, Jiayi, Yinhan, Reza. Their help and friendship have made this an experience to remember and cherish.

Last but not least, I would like to express my grateful thanks to my father and my mother. Thank you for your unconditional love and support.

Notation and abbreviations

GPU	Graphics Processing Unit
2D	Two-dimension
AR	Autoregressive
CCD	Charge-Coupled Device
CMOS	Complementary Metal Oxide Semiconductor
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DPI	Dots Per Inch
DVD	Digital Video Disc
FPS	Frames Per Second
HDTV	High-definition Television
HR	High Resolution
IPTV	Internet Protocol Television
LCD	Liquid Crystal Display
LR	Low Resolution
MDC	Multiple Description Coding
MMSE	Minimum Mean Square Error
PAR	Piecewise Autoregressive

PC	Personal Computer
PPI	Pixels Per Inch
SAD	Sum of Absolute Differences
SAI	Soft-decision Adaptive Interpolation
SDTV	Standard-definition television
SM	Streaming Multiprocessors
SNR	Signal-to-Noise Ratio
SP	Scalar Processor

Contents

List of Figures

Chapter 1

Introduction

1.1 Overview

Digital video is becoming arguably the most popular and important form of visual communication and presentation in our information technology era. Different from digital still images, a digital video consists of moving pictures in time and it is generated either by a digital video camera or by digitizing a motion picture film. Due to the inclusion of time dimension in the pictorial data, digital videos offer much richer information contents than digital images, and hence have a much wider range of applications, including entertainment, consumer electronics, engineering, sciences, medicine, security, defense and etc.

Video processing is a field of visual signal processing that encompasses a number of technical topics: acquisition, communication, motion estimation, scene analysis, restoration and enhancement. In this thesis we are primarily concerned with video

restoration and enhancement. Specifically, we investigate the problem of video resolution upconversion in both spatial and temporal domains. The focus of our investigation is on real-time solutions of the upconversion problem. The research of this thesis is important because video applications, such as video phones, teleconference, digital television, IPTV, and etc., commonly demand real time throughput. Real-time video processing is challenging because video data are inherently voluminous, incurring heavy computational burdens.

To meet the demands for special purpose of real time applications in computer graphics and video processing, the GPU (Graphics Processing Unit) technology was developed in early nineties. GPU uses a massive parallel processing architecture to expedite large scale visual information processing. In the past decade GPU has evolved into a manycore and multithreaded computation engine of great prowess. It is now used not only for speeding up graphics rendering computations but also for accelerating more general-purpose applications.

In order to provide a convenient, high-level software development environment for general-purpose GPU programming, NVIDIA developed the CUDA (Compute Unified Device Architecture) platform. As GPU becomes a standard hardware component in most PCs and severs and the GPU software development platform improves, developing and implementing the GPU-aided video techniques has become more cost-effective.

In this thesis, we reexamine the well-known problems of spatial resolution upconversion and video deinterlacing with the objective of developing GPU-based real-time solutions. The main technical challenge is how to fully utilize the processing power and massive parallelism of GPU to maximize the throughput of upconversion and

(a) (b)

Figure 1.1: Images with different resolutions (a) Low resolution; (b) High resolution.

deinterlacing without compromising the visual quality of the resulting videos.

1.2 Image/Video Spatial Resolution Upconversion

One of the most important quality metrics of digital video is and will continue to be the spatial resolution. Spatial resolution is defined as the number of pixels per unit length. It refers to the pixel density of a digital image or a video frame. Spatial resolution is commonly measured in Dots Per Inch(DPI), or Pixels Per Inch(PPI). In general, given a scene, the higher the spatial resolution, the more and finer details an image/video contains. High spatial resolution of video has paramount importance in computerized video analysis applications in medical, scientific, space, military and security fields. For consumer applications higher spatial resolution directly translates to superior visual quality.

In an ideal world, one can always increase the sensor resolution of image acquisition devices to obtain a desired spatial resolution. There exist hard physical limits on how high a spatial resolution that a video acquisition device can achieve. Firstly, most digital images are acquired by an array of semiconductor sensors such as Charge-Coupled Device (CCD) and Complementary Metal Oxide Semiconductor (CMOS). As the image resolution gets higher and higher, smaller and smaller becomes the pixel. Consequently, the amount of light intercepted by each pixel diminishes and the signal strength reduces. This reduces the signal-to-noise ratio (SNR) of the acquired image. To make the matter worse, densely packed pixels are prone to electronic inference between neighboring sensors. Smaller pixels mean more severe a problem of inference. Therefore, given an SNR requirement, either the size of the sensor or the distance between neighboring sensors cannot be below a hard threshold. Secondly, in some applications the imaging process itself incurs a penalty to the imaged object, which limits the number of pixels to be acquired. For example, for certain medical imaging technologies, high spatial resolution is associated with high dosage of radiation that is harmful to the patient.

Due to the aforementioned limits of the digital video technologies and systems, it is unlikely that newer imaging devices in the future, by themselves, can completely meet the spatial resolution requirements of many scientific, medical and military applications at present and in the future. As such, image/video resolution upconversion is and will remain an important technology to overcome the resolution limit of imaging hardware devices.

Image/video spatial resolution upconversion, or image/video interpolation, is a

process of obtaining a high resolution (HR) image or video frame from a lower resolution (LR) version. Image/video resolution upconversion can be beneficial in many applications ranging from consumer electronics to visual arts and to cutting edge medical and scientific research and development. Image/video spatial resolution upconversion technology is required whenever a user needs an image/video representation of higher resolution than the original source. For instance, image/video resolution upconversion is indispensable for digital multimedia and television industries, as user frequently display videos and image of low resolution(due to compression or old source format) on high-definition television panels and computer monitors.

Furthermore, in many telecommunication applications such as wireless multimedia streaming, the communication bandwidth is at a premium. Video signals have to be compressed for transmission and storage. One of the effective video compression techniques, particularly at very low bit rates, is to down sample video at the encoder. The decoder can then employ a spatial resolution upconversion algorithm to reconstruct the decompressed video back to the original resolution. This approach can have less artifacts than direct compression of original video in many cases [2].

Spatial resolution upconversion can also be used as a technique for multiple description coding (MDC). The MDC is an effective way for multimedia communications over unreliable diversity channels. One way to achieve multiple description image/video coding is to spatially partition an image or a frame into multiple downsampled subimages by a spatial multiplexer [3]. In this case, each subimage can be regarded as a LR version of the original image. When some descriptions (subimages) are lost, the reconstruction of the original image from the received descriptions (subimages) is essentially a spatial resolution upconversion problem.

1.3 Video Deinterlacing

Video consists of a series of images played in rapid succession. Each image refers to a video frame. All mainstream analog and many digital television systems arrange the scan lines of each frame into two consecutive fields, one consisting of all even lines, another of the odd lines. The fields are then displayed in succession at a rate twice that of the nominal frame rate. For instance, PAL and SECAM systems have a rate of 25 frames/s or 50 fields/s, while the NTSC system delivers 29.97 frames/s or 59.94 fields/s. This process of dividing frames into half-resolution fields at double the refresh rate is known as interlacing. Fig. 1.2 shows the sampling scheme for the interlaced video sequences.

Interlaced scan (or interlacing) was invented in order to improve the visual quality of a video signal without consuming extra bandwidth. The popularity and wide use of interlaced videos were largely motivated by the desire of reducing the cost and complexity of video systems. Indeed, high bandwidth increases the costs of all components of a video system: cameras, storage devices (e.g., tape recorders or hard disks), transmission (video compressor and decompressor), and display devices (e.g., television sets, PC monitors). Interlaced video reduces the signal bandwidth by a factor of two, for a given line count and refresh rate. For instance, 1920x1080 pixel resolution interlaced HDTV with a 60 Hz field rate (known as 1080i60) has a similar bandwidth to 1280x720 pixel progressive scan HDTV with a 60 Hz frame rate (720p60), but approximately twice the spatial resolution. In other words, a given bandwidth can be used to provide an interlaced video signal with twice the display refresh rate for a given line count (versus progressive scan video). This helps to reduce flickering artifacts by taking advantage of the persistence (memory) of human vision,

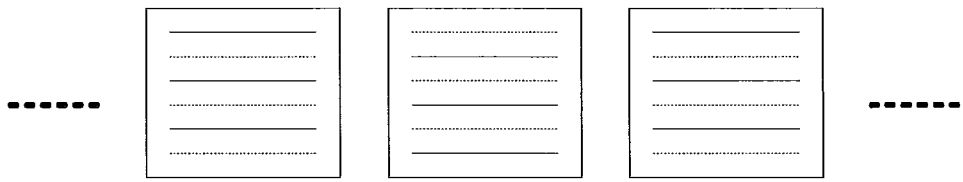


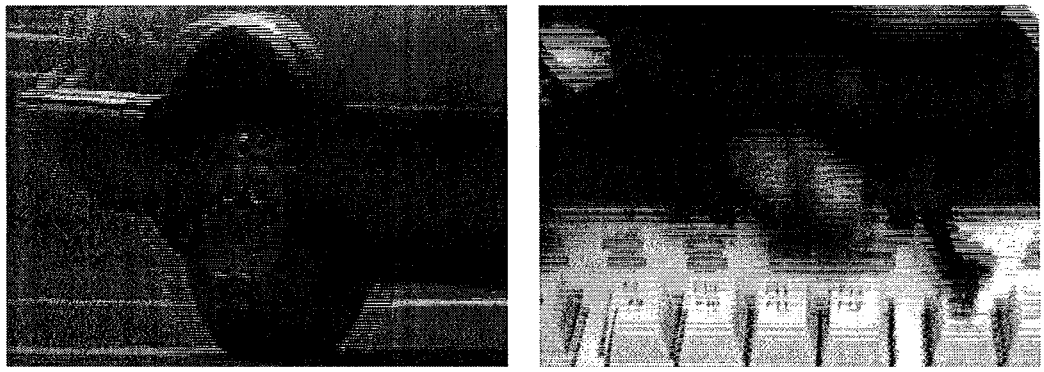
Figure 1.2: An example of the interlaced video sequence

and achieves a visual quality as though the frame rate was doubled.

Interlacing is cost effective and has until recently been considered adequate in offering most users satisfactory viewing experience. But this is no longer the case for several reasons. Firstly, modern video output devices (e.g., LCD television sets, monitors, projectors, etc.) are almost exclusively progressive, because progressive scan offers superior visual quality than the interlaced counterpart. Secondly, interlacing artifacts that were hardly visible in the past can become visually objectionable as screens have grown larger, brighter, and are of higher contrasts. Interlacing artifacts can be quite annoying to human viewers.

Interline twittering effect shows up when the subject being captured contains fine striped patterns that approaches the vertical resolution of the video format. For instance, a person on television wearing a shirt with fine dark and light stripes may appear on a monitor as if the stripes on the shirt are "twittering". Moreover, since each frame of interlaced video is composed of two fields that are captured at different moments in time, interlaced video sequences will exhibit motion artifacts known as "line crawl", or "serration", as shown in Fig. 1.3.

However, most modern broadcast television systems still adopt interlaced video formats. Even with the emergence of high-definition television (HDTV) and the



(a) (b)

Figure 1.3: Interlacing artifacts (a) Line Crawl; (b) Serration.

known superiority in visual quality of progressive video over interlaced video, interlacing persists as one of the formats used for HDTV in the US and Japan (1080i, 1080×1920 resolution with only 540 lines scanned in each field). As most progressive display devices, such as LCD monitors/projectors and LCD/plasma television sets are dominating the market, and as devices used for television and video are integrating with computers, there is and will be an increasing need for conversion from interlaced to progressive formats. This process is known as video deinterlacing.

Converting interlaced video to progressive video doubles the spatio-temporal sampling density. It requires interpolating a set of missing lines in each field. In other words, progressive to interlaced conversion is a form of spatio-temporal subsampling whereas interlaced to progressive conversion - deinterlacing - is a spatio-temporal resolution upconversion process.

1.4 Contributions

This thesis is concerned with hardware expedition of the spatial resolution upconversion and video deinterlacing algorithms. In particular, we study the use of GPU (Graphics Processing Unit) technology for real-time video spatial resolution upconversion and video deinterlacing. Design decisions were made to take full advantage of the GPU architecture and the properties of the CUDA framework. The contributions of this thesis are summarized as follows:

- We first develop a two-pass directional image/video interpolation algorithm for real time resolution upconversion [4]. The first pass of the algorithm generates a quincunx image by interpolating the missing pixels with four available diagonal neighbors. The missing pixels in the quincunx image are then interpolated in the second pass. Each pass of the algorithm employs an estimation-verification procedure to determine the value of a missing pixel. We then propose a GPU implementation of the method, in the NVIDIA CUDA (Compute Unified Device Architecture) platform. Design considerations to speed up the algorithm are discussed. Experimental results show that the GPU-based implementation can be five times as fast as the C implementation using a mid-range GPU card.

- We devise a GPU-motivated motion-adaptive deinterlacing algorithm and develop a CUDA-based implementation of the algorithm [5]. To strike a balance between performance and complexity, we discuss the techniques of adapting the computations in motion detection and adaptive directional interpolation to the GPU architecture for maximum video throughput possible. The proposed video

deinterlacing algorithm operates in two modes: interfield and intrafield, depending on whether significant motions are detected or not. If the current pixel is in a static region of the video scene, then temporal(interfield) deinterlacing is performed that merges the associated even and odd fields to benefit from the correlations between consecutive fields. In the presence of motions, we develop a highly parallelized directional interpolation algorithm for real time intrafield deinterlacing. The interpolation is carried out in three steps:(1) generate five interpolation candidates for the missing pixel using five different directional interpolators; (2) verify the accuracy of each interpolator; (3) select and fuse two winning interpolators to get the final estimate. The design takes full advantage of the CUDA technology and the parallel nature of the proposed algorithm. Experimental results show that the GPU-aided implementation offers real-time solutions even for large video formats, using a mid-range GPU card.

1.5 Organization

The remainder of the thesis is organized as follows. Chapter 2 reviews existing works on image/video interpolation and video deinterlacing. In Chapter 3, we introduce the CUDA and GPU concepts and constructs that are important to the implementation of our image/video interpolation and video deinterlacing algorithms. Chapter 4 presents the proposed directional interpolation algorithm and discusses the detailed design considerations for CUDA adaptation of the upconversion algorithm. In Chapter 5, we discuss the proposed GPU-friendly motion adaptive deinterlacing algorithm in detail. The thesis closes with conclusions and suggested future works in Chapter 6.

Chapter 2

Review of Existing Works

2.1 Image/Video Spatial Resolution Upconversion

In signal processing, resolution upconversion is to reconstruct a continuous signal at higher resolution from a set of observed (measured) low-resolution samples. In this view, the interpolation of an acquired digital image can be interpreted as re-sampling of the original continuous two-dimensional image signal at a higher spatial sampling frequency. According to the Nyquist-Shannon sampling theorem, those signal components that have frequency lower than the Nyquist frequency can be exactly reconstructed. This indicates that, low-frequency components of image signals, such as smooth shades and large-scale edges/textures, can be reconstructed with ease. The real challenge of image/video resolution upconversion is the reconstruction of the high-frequency components of an image or video frame, such as sharp, fine-scale edges and textures, which exceed the Nyquist limit in the frequency domain.

Over the past three decades significant amount of research has been devoted to

image/video resolution upconversion technologies. The existing image/video resolution upconversion methods fall into two categories: 1) simple signal-independent interpolation filtering and 2) adaptive directional filtering. The former methods are simple, inexpensive to implement, but they produce objectionable artifacts in the areas of edges and textures. The latter methods in general produce better visual quality than the former methods, but they incur significantly higher computational complexity. This is why consumer video products, both in software and hardware, adopt the former methods, because they need real-time low-cost solutions.

2.1.1 Non-adaptive Image/Video Interpolation

The simplest non-adaptive image interpolation method is the nearest neighbor technique. To interpolate a missing pixel x, the nearest neighbor algorithm copies the value of the existing pixel that is closest to x. The performance of nearest neighbor interpolation is poor because it ignores the values of other neighboring existing pixels that contain information of the 2D image waveform. Nearest neighbor interpolation produces objectionable checkboard artifacts, particularly for large scaling factors;

The popular bilinear image interpolation method uses up to four neighboring pixel values to interpolate a missing pixel. Bilinear interpolation is simple and can be executed in real time even by software, but its performance leaves much to be desired. It tends to severely blur edges and textures.

The bicubic algorithm is also widely used for scaling images and video sequences. It determines the value of a missing pixel from the weighted average of up to sixteen closest existing pixels. Bicubic interpolation preserves fine detail better than the bilinear and nearest neighbor algorithms and is often chosen over the previous two

methods in image/video resampling.

2.1.2 Adaptive Image/Video Interpolation

As mentioned before, edges details play an important role in human visual system. Therefore, the reproduction of these high-frequency components namely, edge details and fine textures, is crucial to the visual quality of acquired images or video frames. The aforementioned non-adaptive interpolation methods tend to blur edges and/or introduce artifacts in edge areas due to their isotropic interpolation kernels. To maintain the edge sharpness and improve visual quality, a number of edge-guided image interpolation techniques have been proposed in recent years [6; 7; 8; 9]. Carrato and Tenze used some predetermined edge patterns to improve the parameters in the interpolation operator [6]. In [7], Li and Orchard proposed an new edge-directed interpolation algorithm for image spatial resolution upconversion. The algorithm first estimates local covariance coefficients from the input low-resolution image and then uses these covariance estimates to conduct directional interpolation. Zhang and Wu proposed a soft-decision adaptive interpolation (SAI) technique that estimates missing pixels in groups [8]. The SAI technique learns and adapts to varying scene structures using a 2D piecewise autoregressive (PAR) model. The model parameters are estimated in a moving window in the input low-resolution image. The pixel structure dictated by the learnt model is enforced by the soft-decision estimation process onto a block of pixels. Another edge-based interpolation method [9] first interpolates a missing pixel from two mutually orthogonal directions. The two interpolation results are then adaptively fused by the minimum mean square error (MMSE) estimation.

2.2 Video Deinterlacing

In order to bridge the mismatch between interlaced video contents and progressive displays, many video deinterlacing algorithms were proposed [10; 11; 12; 13; 14; 15]. They fall into three categories: intrafield algorithms, interfield algorithms and hybrid algorithms, with different trade-offs between computational complexity and visual quality. The success of video deinterlacing relies on thorough exploitation of both intra- and inter-field correlations.

2.2.1 Intrafield Video Deinterlacing

Intrafield deinterlacing methods interpolate the missing lines by using the pixels of the sampled lines within the current field. Among them line doubling and line averaging are well known and widely used. The line doubling deinterlacing method duplicates lines of the field to fill in the missing lines; the line averaging algorithm interpolates every missing line by averaging two adjacent lines in the same field. The strength of these linear intrafield methods is their low implementation cost. But they often create an annoying artifact known as jagged edges. To alleviate the problem of jagged edges various directional interpolation methods were proposed. Such methods estimate the local edge direction and conduct interpolation along this direction [16; 17]. These algorithms work well if the edge directions are estimated correctly but, if not, they introduce errors and degrade the visual quality. Intrafield methods ignore the temporal correlation between successive video fields, and consequently their performance is suboptimal. These methods are prone to severe video artifacts in regions of high vertical frequencies [10].

2.2.2 Interfield Video Deinterlacing

In interfield approach of deinterlacing, the missing lines are interpolated by utilizing the correlations between current field and previous and/or future fields. Field insertion and bilinear field averaging are two popular interfield deinterlacing techniques [10].

The field insertion method, also called "weaving" in the computer community, fills in the missing lines with neighboring lines in time. The bilinear field averaging algorithm averages the before and after temporal neighboring lines of each missing line. Pure interfield methods work well in absence of motions, but they introduce serration, blurring or flickering artifacts if there are motions between the fields.

2.2.3 Hybrid Deinterlacing Methods

The hybrid deinterlacing methods aim to improve the visual quality by taking advantage of both temporal and spatial corrections of an interlaced video sequence. The widely-used vertical temporal deinterlacing is a linear combination of line averaging and field averaging algorithms. Many variations of vertical temporal filters have been described by Thomas in [18].

The motion adaptive deinterlacing algorithms are among the most popular hybrid methods. They switch between different interpolation strategies (inter-filed or intra-filed), depending on the presence or absence of motions in the current field. In [19], Skarabot *et al.* proposed to first perform per pixel motion detection and then choose different deinterlacing schemes according to different conditions of the motion: a) field averaging when no motion is detected; b) spatio-temporal median filtering when the estimated motion is slow; c) line averaging when fast motion is

detected. A thresholding technique is used to classify the motion into the above three categories. Another approach is to make a smooth weighted transition between field averaging and line averaging interpolation schemes instead of a hard switching between the two schemes Kovacevic *et al* [11]. The weights are determined by sum of absolute differences (SAD). Generally, the hybrid deinterlacing approach obtains better visual quality, but they have higher computational complexity than the previous two approaches.

Chapter 3

Fundamentals of CUDA

3.1 Background

In this chapter, we present the basic terminologies of GPU and CUDA technologies that are needed to describe the GPU-aided image/video interpolation and video deinterlacing algorithms.

A graphics processing unit or GPU is a specialized processor that offloads graphics rendering computations from the CPU. It is used in embedded systems, mobile phones, personal computers, workstations, and game consoles. Modern GPUs use most of their transistors to perform calculations related to 3D computer graphics [1], as schematically illustrated by Fig. 3.1. Their highly parallel structure makes them more effective than general-purpose CPUs for a range of complex algorithms. More specifically, the architecture of GPUs is well suited to data-parallel computations, in which many data elements are processed concurrently in the same program. The CUDA programming model is very well suited to expose the parallel capabilities of GPUs [1]. At its core are three key abstractions–a hierarchy of thread groups, shared

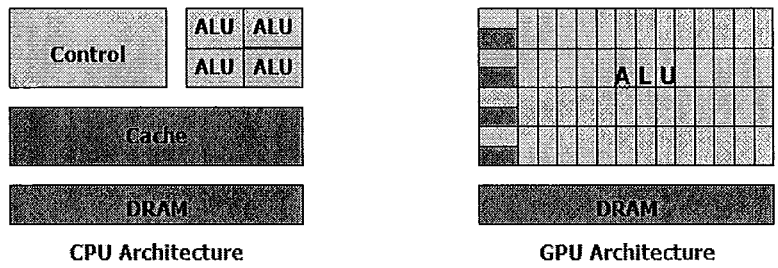


Figure 3.1: The GPU devotes more transistors to data processing [1].

memories and barrier synchronization. These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. They allow the programmers to partition the problem into coarse sub-problems that can be solved independently in parallel, and refine each of these subproblems into yet smaller pieces that can be solved in parallel with shared data.

3.2 Hardware Architecture

The GPU can be viewed as an array of Streaming Multiprocessors (SMs), each containing eight Scalar Processors (SPs). Each SM contains four types of on-chip memory: registers, constant cache, texture cache and shared memory. Each GPU contains three types of off-chip memories: the constant memory and texture memory are read-only memory for SPs, whereas the global memory is both readable and writable by SPs. Each SM also contains four types of on-chip memories that enable faster access: registers, constant cache, texture cache and shared memory. The data in the constant and texture memory are cached from the off-chip memory upon memory access, and is read-only within the SM. The 16-banked shared memory is readable and writable by all SPs within a single SM, which allows SPs to communicate with each other [1].

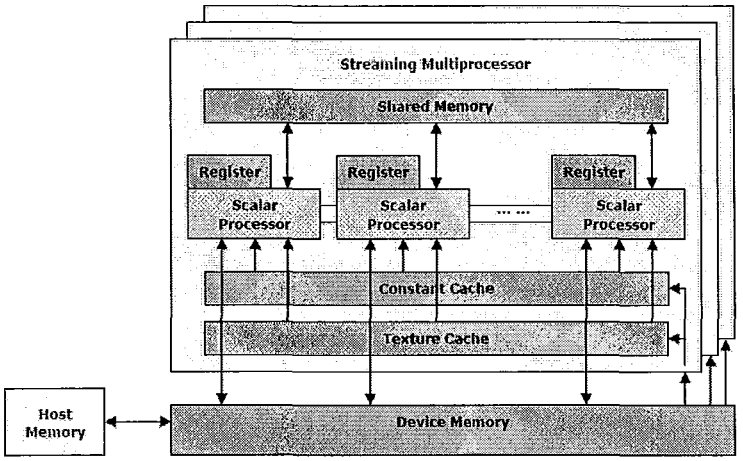


Figure 3.2: Memory architecture for CUDA GPU [1].

Fig. 3.2 shows the memory architecture for a streaming multiprocessor.

3.3 Programming Model

The CUDA programming platform is the first to provide the programmers with C-like programming environment. The CUDA uses heterogeneous programming method, where only the parallel code segments are executed on the device (GPU) while the rest of the program are executed on the host (CPU) in serial. The code segments to be executed on the device, known as kernel functions, are called and spawn from the host function. Fig. 3.3 shows an example of heterogeneous programming model, where the program alternates between single-threaded host function on the CPU and multi-threaded kernel functions on the GPU [1].

The kernel function is executed as a grid of thread blocks, and the total number of threads in the function is determined by the dimension of blocks and grid, as shown in Fig. 3.4. A thread block is a batch of threads that can cooperate with each

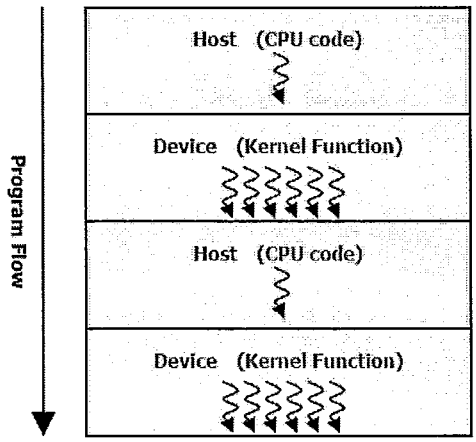


Figure 3.3: Heterogeneous programming model [1].

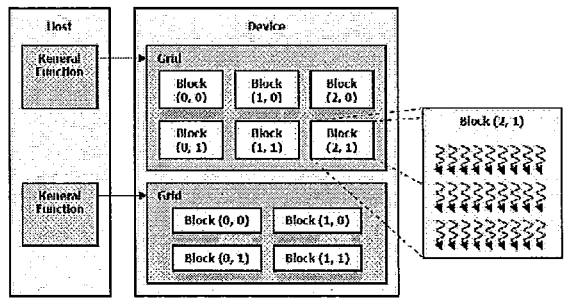


Figure 3.4: Thread, block, and grid arrangement inside kernel functions [1].

other through the on-chip shared memory and synchronize their execution [1]. Each SM may execute one or more thread blocks concurrently depending on the shared memory and register usage; however, threads in different blocks cannot share on-chip memory as they may be executed on a different SM. The SM maps each thread to one scalar processor core, and each thread executes independently with its own instruction address and register state. Each thread block is split into groups of 32 threads called "warps". Full efficiency will be achieved when all 32 threads of a warp agree on their execution paths [1].

3.4 GPU Acceleration of Image/Video Processing

Real-time video processing is challenging because video data are inherently voluminous, incurring heavy computational burdens. In many cases when running video applications on PCs or servers, CPU can be so overwhelmed by heavy computation loads that itself alone cannot meet the real time requirements. For example, currently CPUs in most household PCs alone are not powerful enough to decode and playback 1080p high definition (HD) video in real-time. On the other hand, most modern PCs are equipped with GPU. A question comes along naturally: can we leverage the power of GPU and off-load some or most of video processing operations from the CPU to GPU? The answer is positive. Many video applications that process large data sets can indeed benefit from data-parallel programming models and be sped up drastically by GPU. In 3D rendering large sets of pixels and vertices are mapped to parallel threads. Similarly, in video processing applications such as post-processing, video encoding and decoding, resolution upconversion and deinterlacing, one can easily map blocks of pixels to parallel processing threads. GPU-aided real-time video processing has recently evolved into an active research area [20; 21; 22; 23; 24].

Chapter 4

GPU-aided Directional Image/Video Interpolation for Real Time Resolution Upconversion

The technical challenge of image/video spatial resolution upconversion is how to preserve and reconstruct fine and sharp spatial details in the enlarged image/video while keeping the computational complexity low enough for real time applications. The widely used linear interpolation, cubic spline interpolation and cubic convolution interpolation cannot preserve the edges very well, although they have relatively low computational complexity [25; 26]. Some edge-guided interpolation techniques have been proposed in recent years [7; 8; 9], but they involve complicated computations to maintain the edge sharpness and visual quality, and are therefore not suitable for real time applications. In particular, the method in [9] first interpolates a missing pixel from two mutually orthogonal directions. The two interpolation results are then adaptively fused using the statistics of a local window. The method achieves very

good performance, but it needs relatively complex computation to obtain the local statistics.

In this chapter, by modifying the scheme in [9], we develop a highly parallelized two-pass directional image/video interpolation algorithm for real time resolution upconversion. A novel scheme of estimation and verification is introduced to reduce the computational complexity without significant degradation of the performance.

The rest of the chapter is structured as follows. Section 4.1 presents the proposed directional interpolation algorithm. Section 4.2 discusses the detailed design considerations for CUDA adaptation of the upconversion algorithm. Experimental results are reported and discussed in Section 4.3. Section 4.4 concludes.

4.1 Adaptive Directional Image/Video Interpolation

For clarity and without loss of generality, we limit our description of the algorithm to resolution upconversion by a factor of two.

The image interpolation is carried out in two passes, as shown in Fig. 4.1. The first pass generates a quincunx image by interpolating the missing pixels with four available diagonal neighbors, as marked by gray circles in Fig. 4.1 (a). The missing pixels in the quincunx image are then interpolated in the second pass, as shown in Fig. 4.1 (b).

Each pass of the algorithm employs a three-step procedure to determine the value of a missing pixel: (1) generate two interpolation candidates for the missing pixel using two different interpolators; (2) verify the accuracy of each interpolator; (3) select the

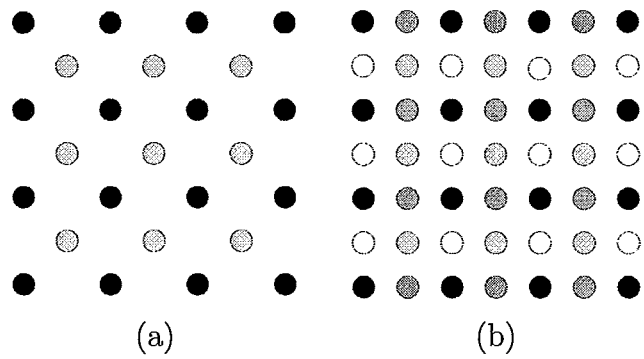


Figure 4.1: Two pass interpolation. (a) The first pass; (b) The second pass.

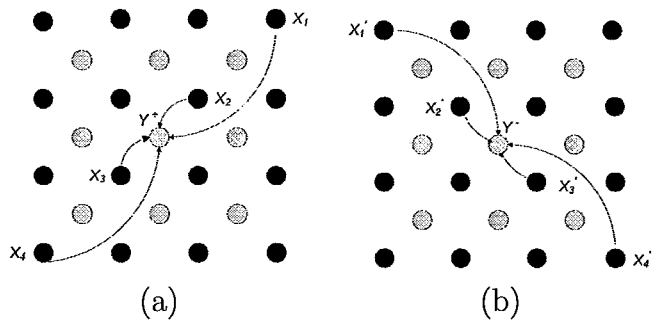


Figure 4.2: Diagonal cubic interpolation. (a) 45 ° cubic interpolation; (b) 135 ° cubic interpolation.

winning interpolator or fuse the two interpolators to get the final estimation.

In the first step, we use the cubic convolution interpolation to estimate an unknown pixel from two diagonal directions, denoted by Y^+ and Y^-, as shown in Fig. 4.2. The estimation is given by [26]

$$Y^+ = -\frac{1}{16}X_1 + \frac{9}{16}X_2 + \frac{9}{16}X_3 - \frac{1}{16}X_4 \tag{4.1}$$

$$Y^- = -\frac{1}{16}X_1' + \frac{9}{16}X_2' + \frac{9}{16}X_3' - \frac{1}{16}X_4'. \tag{4.2}$$

In the second step, we evaluate the accuracy of the two directional cubic convolution interpolations. This is achieved by applying the same formula to estimate the known

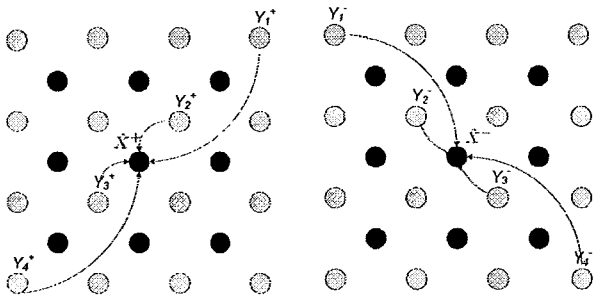


Figure 4.3: Diagonal cubic verification. (a) 45 ° verification; (b) 135 ° verification.

pixels using the interpolated pixels, as shown in Fig. 4.3, *i.e.*,

$$\hat{X}^+ = -\frac{1}{16}Y_1^+ + \frac{9}{16}Y_2^+ + \frac{9}{16}Y_3^+ - \frac{1}{16}Y_4^+ \tag{4.3}$$

$$\hat{X}^- = -\frac{1}{16}Y_1^- + \frac{9}{16}Y_2^- + \frac{9}{16}Y_3^- - \frac{1}{16}Y_4^-. \tag{4.4}$$

The errors of the two directional interpolators in a neighborhood around $Y(i,j)$ can thus be obtained as

$$e_k(i,j) = \sum_{(m,n)\in W(i,j)} (X(m,n) - \hat{X}_k(m,n))^2, k = 1,2 \tag{4.5}$$

where $e_1(i,j)$ ($e_2(i,j)$) represents the error in the 45° (135°) direction, $X(m,n)$'s are the known pixel values of the diagonal neighbors around $Y(i,j)$ in a window $W(i,j)$, $\hat{X}(m,n)$'s are the diagonal estimations of $X(i,j)$. For the 135° cubic verification, the verification window $W(i,j)$ is shown in Fig. 4.4, where the errors of four known neighbors of $Y(i,j)$ are evaluated in Eq. 4.5. The 45° cubic verification process is obtained similarly.

In the third step, the final value of the missing pixel $Y(i,j)$ is determined based

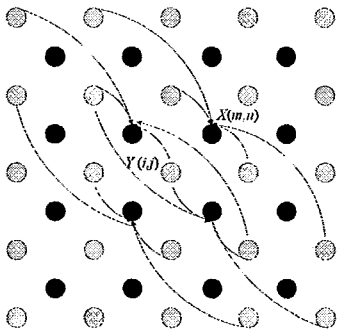


Figure 4.4: The verification of the 135° cubic interpolator in a local window of four known pixels surrounding the missing pixel $Y(i,j)$.

on $e_1(i,j)$ and $e_2(i,j)$ [9]:

$$Y(i,j) = \begin{cases} Y^+(i,j), & e_2(i,j) > e_1(i,j) + \tau, \\ Y^-(i,j), & e_1(i,j) > e_2(i,j) + \tau, \\ Y^+(i,j), & e_1(i,j) = e_2(i,j) = 0, \\ Y_{fuse}(i,j), & \text{otherwise}, \end{cases} \tag{4.6}$$

where τ is a prespecified threshold, and

$$Y_{fuse}(i,j) = \frac{e_2(i,j)}{e_1(i,j) + e_2(i,j)} Y^+(i,j) + \frac{e_1(i,j)}{e_1(i,j) + e_2(i,j)} Y^-(i,j). \tag{4.7}$$

Given the results of the first pass, the second pass of the interpolation algorithm is carried out with two interpolators applied horizontally and vertically. Accordingly, the estimation errors of the horizontal and vertical neighbors are collected in the verification step. These operations can be understood by rotating Fig. 4.2 to Fig. 4.4 by 45°. The final upsampled image is produced after the second pass.

Despite its simplicity, the performance of this algorithm is significantly better than the cubic methods, as shown in Sec. 4.3. In addition, the algorithm only uses local image information and is therefore suitable for GPU-based parallel computing.

4.2 CUDA Implementation of Adaptive Directional Image/Video Interpolation

4.2.1 General Program Flow and Thread Configuration

In this section, we describe how to efficiently implement the spatial upconversion algorithm in Sec. 4.1 using GPU. For a GPU program to effectively use all available resources and achieve maximum performance, we must maximize parallel execution, optimize instruction usage to achieve the maximum instruction throughput, and optimize the memory usage for maximum memory bandwidth.

Since the GPU is a massively threaded parallel processor, we choose to perform all calculations and interpolations on the GPU, and each thread only processes a single pixel. In this way, many pixels can be processed simultaneously. Due to the dependencies of the pixel values generated by each pass of the interpolation algorithm, each kernel function should only perform a single pass of calculation as the value generated would be required for the next kernel function. The general flow diagram of the GPU implementation of the proposed two-pass method is shown in Fig. 4.5. The wide yellow arrow shows the memory transfers: from host memory, to device shared memory, and back to host memory.

It is recommended to have 192 or 256 threads per block in order to get optimal utilization of the available computing resources [1]. In addition, the thread dimensions should be aligned to the size of the warp to avoid diverging warp. So for a branch with condition determined by the x and y coordinates of the pixel, all threads within the warp will most likely branch in the same way. Therefore, we choose 32 as the x dimension of the thread block, and 32×8 as the dimension of the thread block.

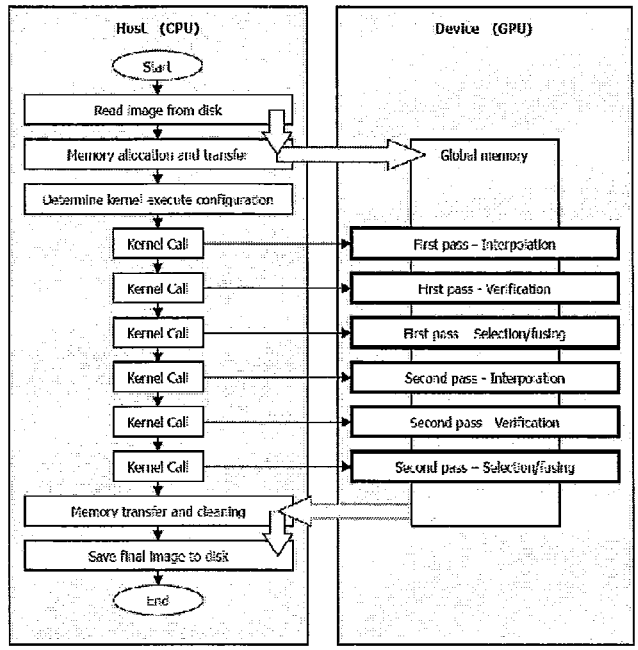


Figure 4.5: General program flow of GPU implementation of the algorithm.

4.2.2 Four-subimage-based Memory Allocation

In the CUDA platform, the global memory spaces are the readable and writable regions of device memory for all SPs. Due to the lack of caching, the access to global memory is relatively slow. Therefore it is desired to improve the efficiency of the global memory access. This can be achieved if the simultaneous memory accesses by threads in a half-warp can be coalesced into a single memory transaction [1].

The straightforward implementation of the directional interpolation algorithm would allocate a memory space that equals to the size of the upconverted image, and then perform all the two-pass computations in the allocated memory space. Although this method requires less memory space, it precludes coalesced memory access when implemented in CUDA. This can be seen from Fig. 4.6 (a), where each thread must skip over a slot in memory to access the next desired pixel. Therefore, the

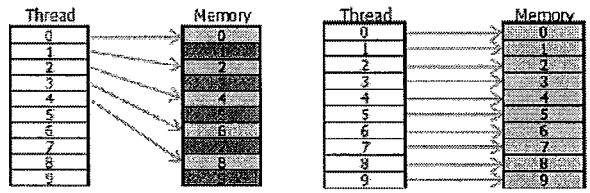


Figure 4.6: (a) Uncoalesced Memory Access; (b) Coalesced Memory Access.

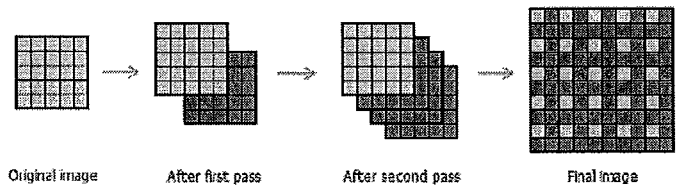


Figure 4.7: Subimage-based memory allocation. The high resolution image is generated at the last stage.

memory access is not contiguous.

In order to achieve coalesced memory access, we group the pixels in each subset in Fig. 4.1 (b) into a subimage and assign a contiguous memory space. As a result, four memory spaces are allocated. After the first pass, the second subimage as shown by gray pixels in Fig. 4.1 (a) is generated. The two subimages together represent the quincunx image from the original algorithm. After the second pass, the other two subimages are generated. Finally, the four subimages are merged together to form the final high resolution output. Fig. 4.7 shows the process of this approach. With this method, the memory access is contiguous and can be coalesced for the computations performed on each subimage, as shown in Fig. 4.6 (b).

4.2.3 Caching Mechanism

Since access to global memory is slow, it is necessary to cache the data into a faster memory before each computation. The shared memory is chosen in our current implementation, because it has more spaces than the texture memory. Furthermore, for all threads of a warp, accessing the shared memory is as fast as accessing a register, as long as there is no bank conflict between the threads. Bank conflict occurs when more than one address of a memory request fall in the same memory bank. In this case, the access has to be serialized, which leads to significant performance degradation. Therefore, in order to avoid bank conflict, each of the 16 threads in a half-warp should access a different bank.

In addition, since the algorithm needs to read the values of up to 2 pixels out of each image block in both the x and y directions when processing the boundary pixels, one to one mapping between the global and shared memory does not work well. A simple tradeoff is to cache two extra rows or columns of pixels around the target block, as shown in Fig. 4.8. This caching mechanism does not completely eliminate blank conflict, as bank conflict is still possible when threads in a half-warp access the memory addresses that fall in the same shared memory bank. However, such a probability is quite low and does not have significant impact to the performance.

On the other hand, it is possible to completely avoid bank conflict by using texture memory. However, this requires smaller thread blocks and might affect the throughput. In the future, we plan to implement this approach and compare the overall performance with the current scheme.

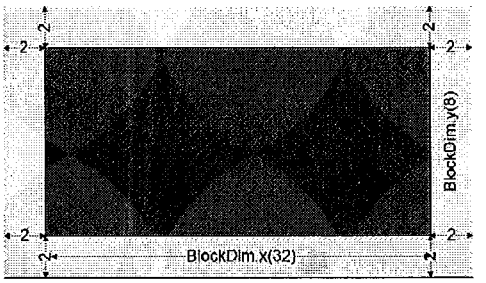


Figure 4.8: The size of cached data compared to block size.

4.2.4 Branch and Loop Replacement

Branches and loops should be avoided in GPU computing to prevent divergence among the threads within the same warp. For example, to clamp a pixel value x within 0 and 255, the branches can be replaced by the *min* and *max* functions:

$$min(max(x, 0), 255). \tag{4.8}$$

Similarly, the flipping of image pixels at boundary can be implemented as

$$min(max(x, -x), 2w - x - 2), \tag{4.9}$$

where w is the width of the image, and x is the target position.

Branches also occur during the selection of the interpolators in Eq. 4.6. However, as the branch condition is generated from the interpolation errors calculated using Eq. 4.5, it is independent of thread configuration, thus divergent warp cannot be avoided in this case.

In our implementation, the main loop for processing each pixel is replaced indirectly by threads and blocks. There is no other loop inside the kernel function.

Table 4.1: Processing time for a 256 × 256 Lena image

	Running Time	**Relative Speed**
Original	41.6 ms	1.00x
CUDA Optimized	11.2 ms	3.71x
Optimized without malloc/free	8.5 ms	4.89x

4.3 Experimental Results and Discussion

To evaluate the performance of the proposed interpolation algorithm on CUDA, the following experimental environment is used: (1) NVIDIA Quadro FX 1700 GPU card with 512MB SDRAM memory and 4 streaming multiprocessors, (2) 3GHz Intel Core 2 Duo E8400 CPU, (3) Microsoft Windows XP sp3, (4) Microsoft Visual Studio 2005, (5) CUDA Toolkit and SDK 2.0, and (6) NVIDIA Driver for Microsoft Windows XP with CUDA Support (178.24).

We first compare in Fig. 4.9, 4.10 and 4.11 the interpolation performance of the popular bicubic interpolation in [26] and the GPU-aided directional upconversion algorithm. It can be seen that the result of the proposed method is visually more pleasing than the bicubic interpolation, with edges faithfully reconstructed without any jaggy.

We next compare the executing time of the image processing part for the 256×256 lena image, by ignoring the program initialization and reading/saving of image file. As shown in Table 4.1, the CUDA optimized multi-threaded computation is close to 4 times as fast as the original implementation. For video processing, memory on the device can be reused for each frame. Therefore, memory allocation is only needed for the first frame. If we ignore the overhead of memory allocation, the speed of the GPU-aided algorithm is almost 5 times of the original method.

Table 4.2 shows the results for some video-sized images. When the cost of memory

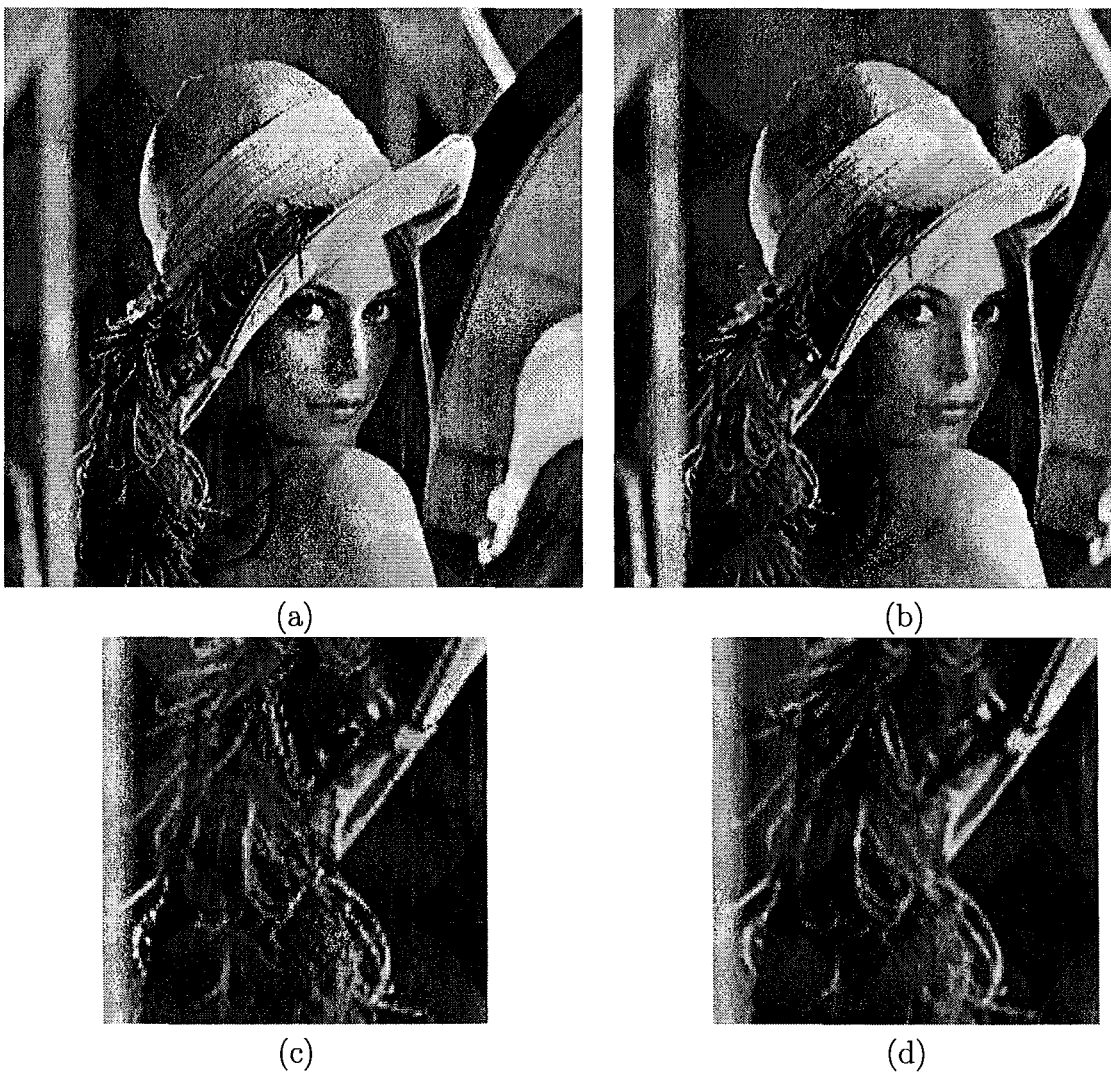


Figure 4.9: Comparison of different methods. (a) (c) Bicubic interpolation; (b) (d) The GPU-aided directional interpolation.

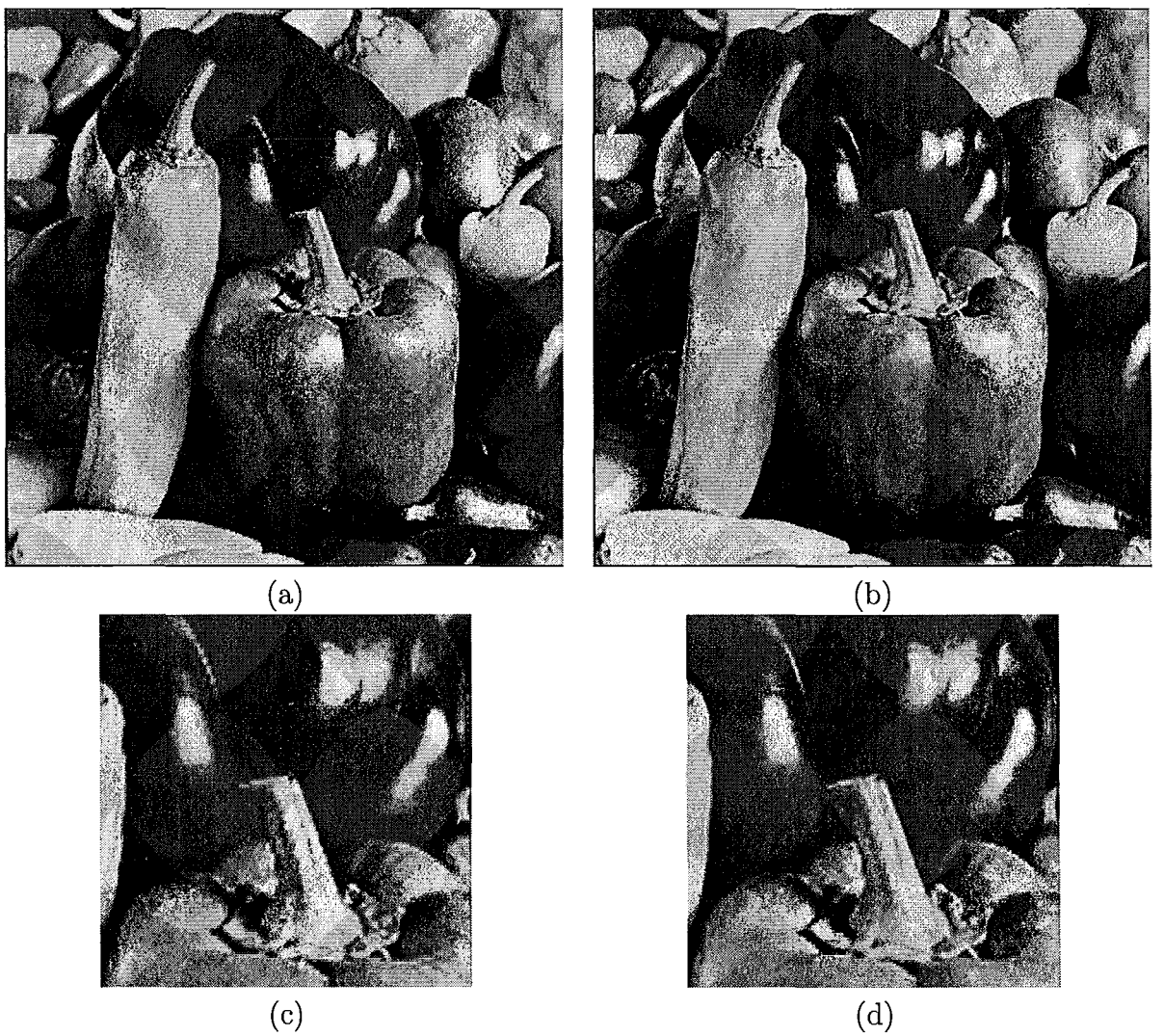


Figure 4.10: Comparison of different methods. (a) (c) Bicubic interpolation; (b) (d) The GPU-aided directional interpolation.

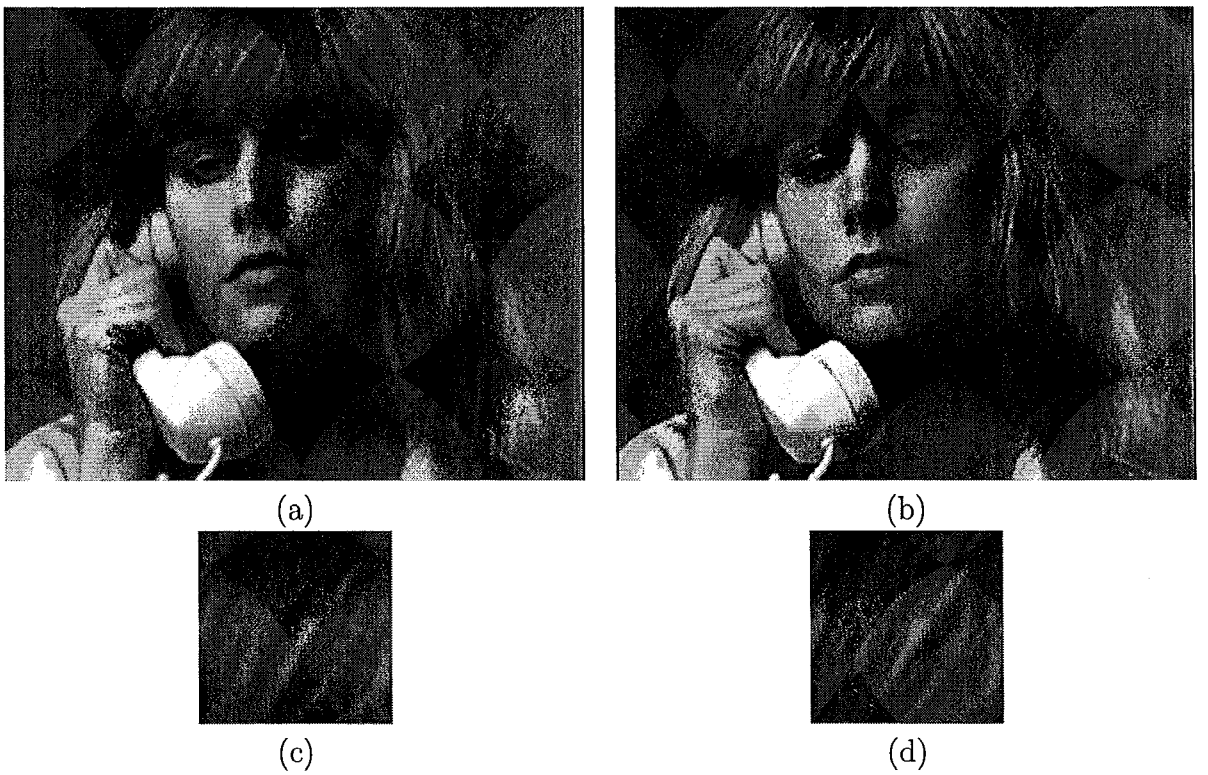


Figure 4.11: Comparison of different methods. (a)(c) Bicubic interpolation; (b)(d) The GPU-aided directional interpolation.

Table 4.2: Processing time for video-sized image

Video size	**640×480**		**720×480**		**1280×960**	
	Time	**Speed**	**Time**	**Speed**	**Time**	**Speed**
Original CPU-based	187 ms	1.00x	213 ms	1.00x	763 ms	1.00x
CUDA-Optimized	40 ms	4.68x	44 ms	4.84x	152 ms	5.02x
Optimized without malloc/free	37 ms	5.05x	41 ms	5.20x	148 ms	5.16x

allocation is ignored, the CUDA optimized code is also about 5 times as fast as the original implementation. The first case of Table 4.2 has a regular 4:3 DVD-sized video frame input. The processing time of 37 milliseconds per frame corresponds to around 27 frames per seconds(FPS), assuming the video decoding is handled by a co-processor such as the CPU or another graphics card. Therefore real time upconversion of DVD video input using GPU is possible. This represents a significant improvement over the original CPU implementation, which can barely handle 6 frames per seconds.

Note that the graphics card used for the testing only contains 4 streaming multiprocessors, and each of them supports 768 threads, whereas the latest NVIDIA GPU card can have up to 30 multiprocessors, and each can handle 1024 threads. Therefore, significant speedup can be further obtained using these latest graphic cards.

4.4 Conclusion

In this chapter, we present an efficient interpolation algorithm for image/video resolution upconversion and its GPU implementation, by taking full advantage of the CUDA technology and the properties of the interpolation algorithm. Experimental results show that the GPU-aided algorithm can be 5 times as fast as the original

version, using a mid-range GPU card.

Chapter 5

GPU-aided Motion Adaptive Video Deinterlacing

In this chapter we develop a motion adaptive video deinterlacing algorithm that has a high degree of data parallelism so that it can be implemented on GPU. The remainder of the paper is organized as follows. Section 5.1 presents the proposed motion adaptive video deinterlacing algorithm. Section 5.2 discusses in detail how to adapt CUDA to video deinterlacing. Experimental results are reported and discussed in Section 5.3. Section 5.4 concludes.

5.1 Motion Adaptive Video Deinterlacing With Adaptive Directional Interpolation

Our motion adaptive video deinterlacing algorithm operates in two modes: interfield and intrafield, depending on whether significant motions are detected or not. If

the current pixel is in a static region of the video scene, then temporal (interfield) deinterlacing is performed that merges the associated even and odd fields to benefit from the correlations between consecutive fields. In the presence of motions, merging even and odd fields becomes error prone, the algorithm resorts to adaptive directional interpolation within the current field.

The key issue in intrafield deinterlacing is how to preserve and reconstruct fine and sharp spatial details in the deinterlaced video while keeping the computational complexity sufficiently low for real time applications. The methods of line repetition and vertical line averaging tend to blur edges too much, although they have low computational complexity. Some edge dependent algorithms have been developed to preserve details and maintain edge sharpness [10; 11; 12; 13; 14; 15], but they are complicated and hence not suitable for real time applications.

To strike a balance between performance and complexity, we develop a highly parallelized directional interpolation algorithm for real time intrafield deinterlacing. The algorithm is a three-step procedure to estimate the value of a missing pixel in current field: (1) generate five interpolation candidates for the missing pixel using five different directional interpolators; (2) verify the accuracy of each interpolator; (3) select and fuse two winning interpolators to get the final estimate.

In the first step, five directional interpolators are used to compute five tentative values of a missing pixel, denoted by $Y_i (i = 1, 2, ..., 5)$. In the first three directions as shown in Fig. 5.1, cubic interpolator is used. Simple averaging method is used for the remaining two cases.

$$Y_1 = -\frac{1}{16}X_{1,1} + \frac{9}{16}X_{1,2} + \frac{9}{16}X_{1,3} - \frac{1}{16}X_{1,4} \tag{5.1}$$

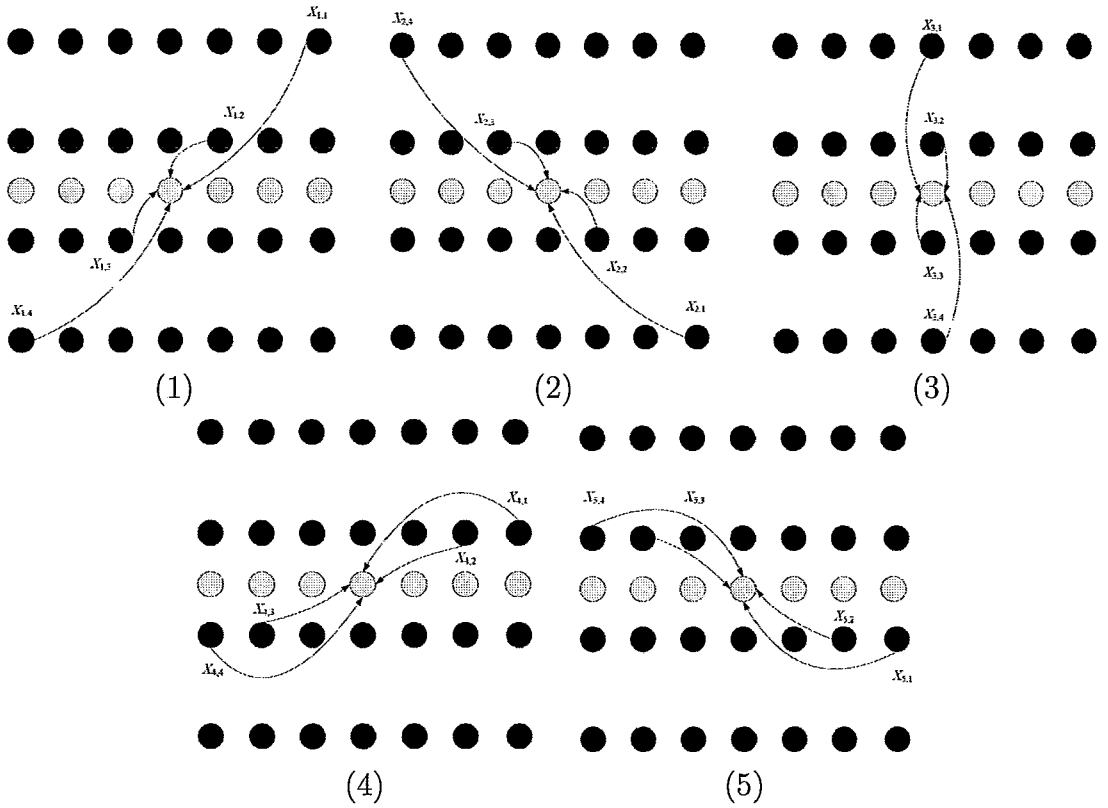


Figure 5.1: Five directional interpolators.

$$Y_2 = -\frac{1}{16}X_{2,1} + \frac{9}{16}X_{2,2} + \frac{9}{16}X_{2,3} - \frac{1}{16}X_{2,4} \tag{5.2}$$

$$Y_3 = -\frac{1}{16}X_{3,1} + \frac{9}{16}X_{3,2} + \frac{9}{16}X_{3,3} - \frac{1}{16}X_{3,4} \tag{5.3}$$

$$Y_4 = \frac{1}{4}X_{4,1} + \frac{1}{4}X_{4,2} + \frac{1}{4}X_{4,3} + \frac{1}{4}X_{4,4} \tag{5.4}$$

$$Y_5 = \frac{1}{4}X_{5,1} + \frac{1}{4}X_{5,2} + \frac{1}{4}X_{5,3} + \frac{1}{4}X_{5,4} \tag{5.5}$$

In the second step, we evaluate the accuracy of these five directional interpolators. This is achieved by applying the same formula to estimate the known pixels using the interpolated pixels, as shown in Fig. 5.2, *i.e.*,

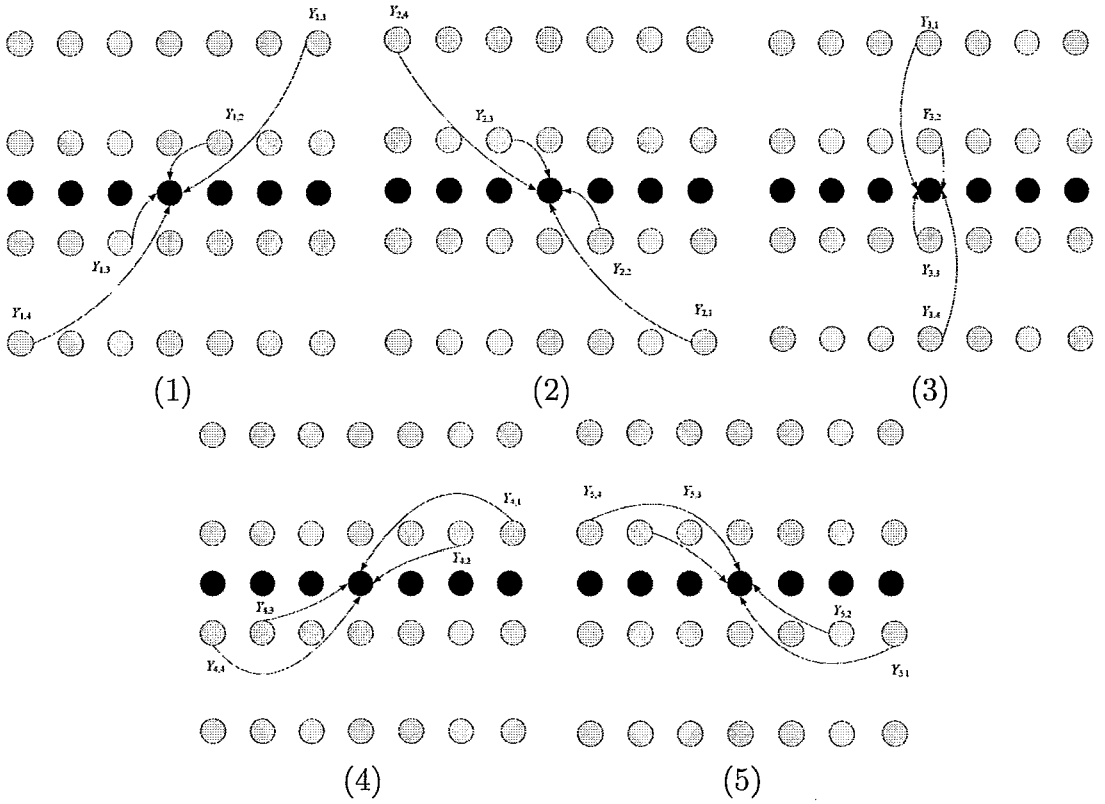


Figure 5.2: Verification process

$$\hat{X}_1 = -\frac{1}{16}Y_{1,1} + \frac{9}{16}Y_{1,2} + \frac{9}{16}Y_{1,3} - \frac{1}{16}Y_{1,4} \tag{5.6}$$

$$\hat{X}_2 = -\frac{1}{16}Y_{2,1} + \frac{9}{16}Y_{2,2} + \frac{9}{16}Y_{2,3} - \frac{1}{16}Y_{2,4} \tag{5.7}$$

$$\hat{X}_3 = -\frac{1}{16}Y_{3,1} + \frac{9}{16}Y_{3,2} + \frac{9}{16}Y_{3,3} - \frac{1}{16}Y_{3,4} \tag{5.8}$$

$$\hat{X}_4 = \frac{1}{4}Y_{4,1} + \frac{1}{4}Y_{4,2} + \frac{1}{4}Y_{4,3} + \frac{1}{4}Y_{4,4} \tag{5.9}$$

$$\hat{X}_5 = \frac{1}{4}Y_{5,1} + \frac{1}{4}Y_{5,2} + \frac{1}{4}Y_{5,3} + \frac{1}{4}Y_{5,4} \tag{5.10}$$

The errors of these directional interpolators in a neighborhood around $Y(i,j)$ can

thus be obtained as

$$e_k(i,j) = \sum_{(m,n)\in W(i,j)} (X(m,n) - \hat{X}_k(m,n))^2, k = 1,2,...,5 \tag{5.11}$$

where $e_k(i,j)$ represents the error in the corresponding direction, $X(m,n)$s are the known pixel values of the neighbors around $Y(i,j)$ in a window $W(i,j)$, $\hat{X}_k(m,n)$'s are the different directional estimates of $X(m,n)$, which are interpolated through Eq. 5.6 to Eq. 5.10.

We select the best two directional estimates, denoted by $Y_{1st}(i,j)$, $Y_{2nd}(i,j)$. Let the estimation errors for $Y_{1st}(i,j)$ and $Y_{2nd}(i,j)$ be $e_{1st}(i,j)$ and $e_{2nd}(i,j)$.

In the third step, the final estimate of the missing pixel $Y(i,j)$ is computed by fusing $Y_{1st}(i,j)$ and $Y_{2nd}(i,j)$ based on $e_{1st}(i,j)$ and $e_{2nd}(i,j)$:

$$Y_{intra}(i,j) = \begin{cases} Y_{1st}(i,j), & e_{2nd}(i,j) > e_{1st}(i,j) + \tau, \\ Y_{2nd}(i,j), & e_{1st}(i,j) > e_{2nd}(i,j) + \tau, \\ Y_{1st}(i,j), & e_{1st}(i,j) = e_{2nd}(i,j) = 0, \\ Y_{fuse}(i,j), & \text{otherwise}, \end{cases} \tag{5.12}$$

where τ is a prespecified threshold, and

$$Y_{fuse}(i,j) = \frac{e_{2nd}(i,j)}{e_{1st}(i,j) + e_{2nd}(i,j)} Y_{1st}(i,j) + \frac{e_{1st}(i,j)}{e_{1st}(i,j) + e_{2nd}(i,j)} Y_{2nd}(i,j). \tag{5.13}$$

5.2 CUDA Implementation of Motion Adaptive Video Deinterlacing

5.2.1 General Program Flow

Having introduced the basics of GPU we are now ready to describe efficient implementation of the motion adaptive deinterlacing algorithm proposed in section 5.1 on

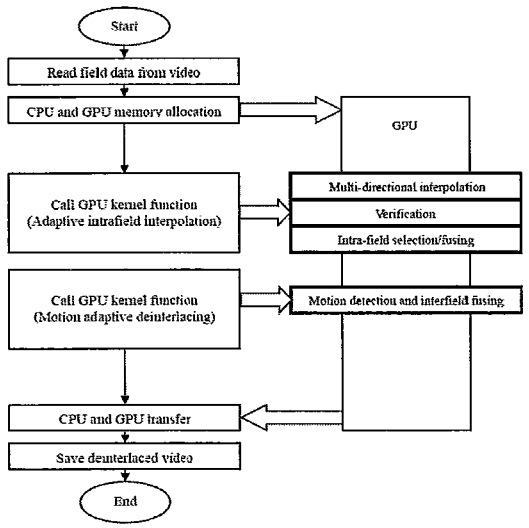


Figure 5.3: General program flow of GPU-aid implementation

GPU. For a GPU program to effectively use all available resources and achieve maximum performance, we need to maximize the degree of parallelism, optimize the code to achieve the highest possible instruction throughput. In addition, we optimize the memory usage for maximum memory bandwidth.

Since the GPU is a massively threaded parallel processor, we can assign the processing of each pixel to a thread. High throughput is gained by deinterlacing a large number of pixels simultaneously. This is also the reason that we developed our deinterlacing algorithm to have identical operations on each pixel. Each kernel function should only perform a single step of the deinterlacing algorithm as the current results are required by the next kernel function. Fig. 5.3 demonstrates the general flow diagram of the GPU implementation of the proposed deinterlacing algorithm.

5.2.2 GPU Memory Allocation

In the CUDA platform, the global memory spaces are the readable and writable regions of device memory for all SPs. Due to the lack of caching, the access to global memory is relatively slow. To make the use of memory bandwidth as efficient as possible, the simultaneous memory accesses by threads in a half-warp should coalesced into a single memory transaction [1].

The straightforward implementation of the motion adaptive deinterlacing algorithm would allocate a memory space that equals to the size of the deinterlaced frame, and then perform all the computations in the allocated memory space. Although requiring less memory space, this method prevents coalesced memory access when implemented in CUDA. This point is made by Fig. 5.5 (a), where each thread must skip over a slot in memory to access the next desired line of pixels. Therefore, the memory access is not contiguous.

In order to achieve coalesced memory access, we allocate a contiguous memory space to store the pixel values in each field as shown in Fig. 5.4. To support the execution of the proposed deinterlacing algorithm, four contiguous memory spaces are created.

The first memory space contains the pixel values of the current field. The second memory space stores the results of the intrafield deinterlacing process, namely the estimates of the missing pixels. The pixel values in these two memory spaces together represent the deinterlaced frame by the adaptive intrafield directional interpolation. The remaining two memory spaces contain pixel values of the previous and next fields. They are needed when merging the even and odd fields in the absence of motion. Fig. 5.4 shows the described memory organization. In this design the memory access

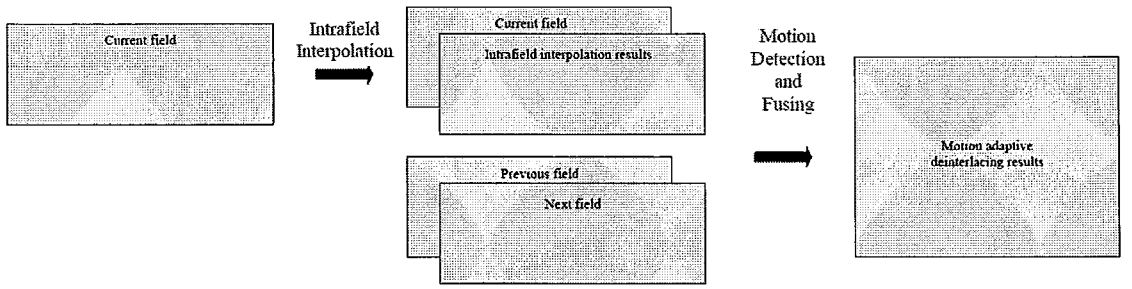


Figure 5.4: Continuous and coalesced memory allocation

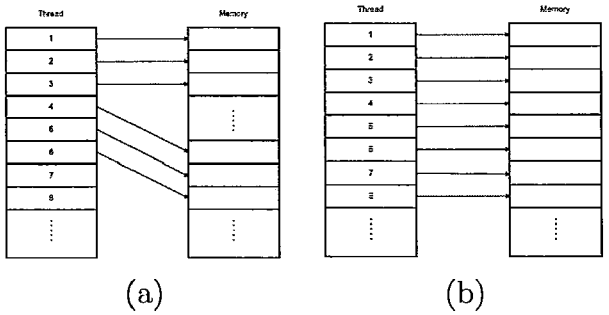


Figure 5.5: (a) Uncoalesced Memory Access; (b) Coalesced Memory Access.

is made contiguous and coalesced as shown in Fig. 5.5 (b).

5.2.3 Thread Configuration and Overlapped Caching Mechanism

It is recommended to have 192 or 256 threads per block in order to get optimal utilization of the available computing resources [1]. In addition, the thread dimensions should be aligned to the size of the warp to avoid diverging warp. Therefore, the size of 32×8 is chosen for the dimension of the thread block.

Since access to global memory is slow, it is necessary to cache the data into a faster memory before each computation. The shared memory is chosen in our current

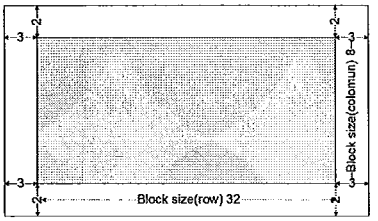


Figure 5.6: Overlapped data caching mechanism compared to block size

implementation, because it has more spaces than the texture memory. Furthermore, for all threads of a warp, accessing the shared memory is as fast as accessing a register, as long as there is no bank conflict between the threads. Bank conflict occurs when more than one address of a memory request fall in the same memory bank. In this case, the access has to be serialized, which leads to significant performance degradation. Therefore, in order to avoid bank conflict, each of the 16 threads in a half-warp should access a different bank.

Another issue in memory management is the handling of boundary pixels for interpolation filters. The proposed algorithm needs to read the data up to three pixels outside of each thread block in the x direction, and two pixels outside of the block in the y direction. When processing the boundary pixels, one to one mapping between the global and shared memory does not work well. In order to cache all the pixels needed for processing, a boundary-overlapped memory allocation is required. It caches two extra rows and three extra columns of pixels around the block, as shown in Fig. 5.6.

5.3 Experimental Results and Discussion

The performance of the proposed GPU-aided video deinterlacing algorithm is evaluated in the following hardware and software setting: (1) NVIDIA GeForce 9800 GTX+ GPU with 512MB memory (2) 2.83GHz Intel Xeon CPU E5440, (3) Microsoft Windows XP sp3, (4) Microsoft Visual Studio 2005, (5) CUDA Toolkit and SDK 2.0, and (6) NVIDIA Driver with CUDA Support (178.24).

We first compare in Fig. 5.7 the performances of our intrafield adaptive directional interpolation and the widely used bicubic interpolation. It can be seen that the result of the former is visually more pleasing than the latter, with edges faithfully reconstructed without any jaggy.

We present the results of the proposed motion adaptive deinterlacing method in Fig. 5.8 and Fig. 5.9. Visual performance are compared between the bilinear field averaging method and the proposed motion adaptive algorithm. It can be found that the proposed motion adaptive method can adapt itself to motion level and achieves good performance both in presence and absence of the motion in the current field.

Finally, we demonstrate that the proposed GPU implementation of the motion adaptive deinterlacing algorithm has sufficiently high throughput for real-time video applications. The execution times of the CPU and GPU implementations are compared in Table 5.1. In the comparison we ignore the time for program initialization and reading/saving of video files to evaluate the speed up factor for deinterlacing operations only. For video processing, memory on the device can be reused for each frame or field. Therefore, we ignore the overhead of memory allocation.

As shown in Table 5.1 the GPU-aided implementation is almost eight times as fast as the CPU version. The throughput of the GPU-aided implementation is around

Table 5.1: Processing time for interlaced video sequence

Field size	**640×240**		**720×240**		**1280×480**	
	fields/second	**speed**	**fields/second**	**speed**	**fields/second**	**speed**
Serial C program	15.2	1.00x	11.8	1.00x	3.1	1.00x
CUDA-Implementation	110.3	7.26x	96.2	8.15x	30.7	9.90x

100 fields per second for DVD-sized video sequences, excluding the time for video decoding. Even for high definition interlaced video sequences, the processing speed can still satisfy real-time requirement. In contrast, the CPU implementation can barely process around 10 frames per seconds at DVD size.

5.4 Conclusion

In this chapter, we designed an efficient motion adaptive video deinterlacing algorithm that is particularly suitable for GPU implementation. The design takes full advantage of the CUDA technology and the parallel nature of the problem. Experimental results show that the GPU-aided implementation offers real-time solutions even for large video formats, using a mid-range GPU card.

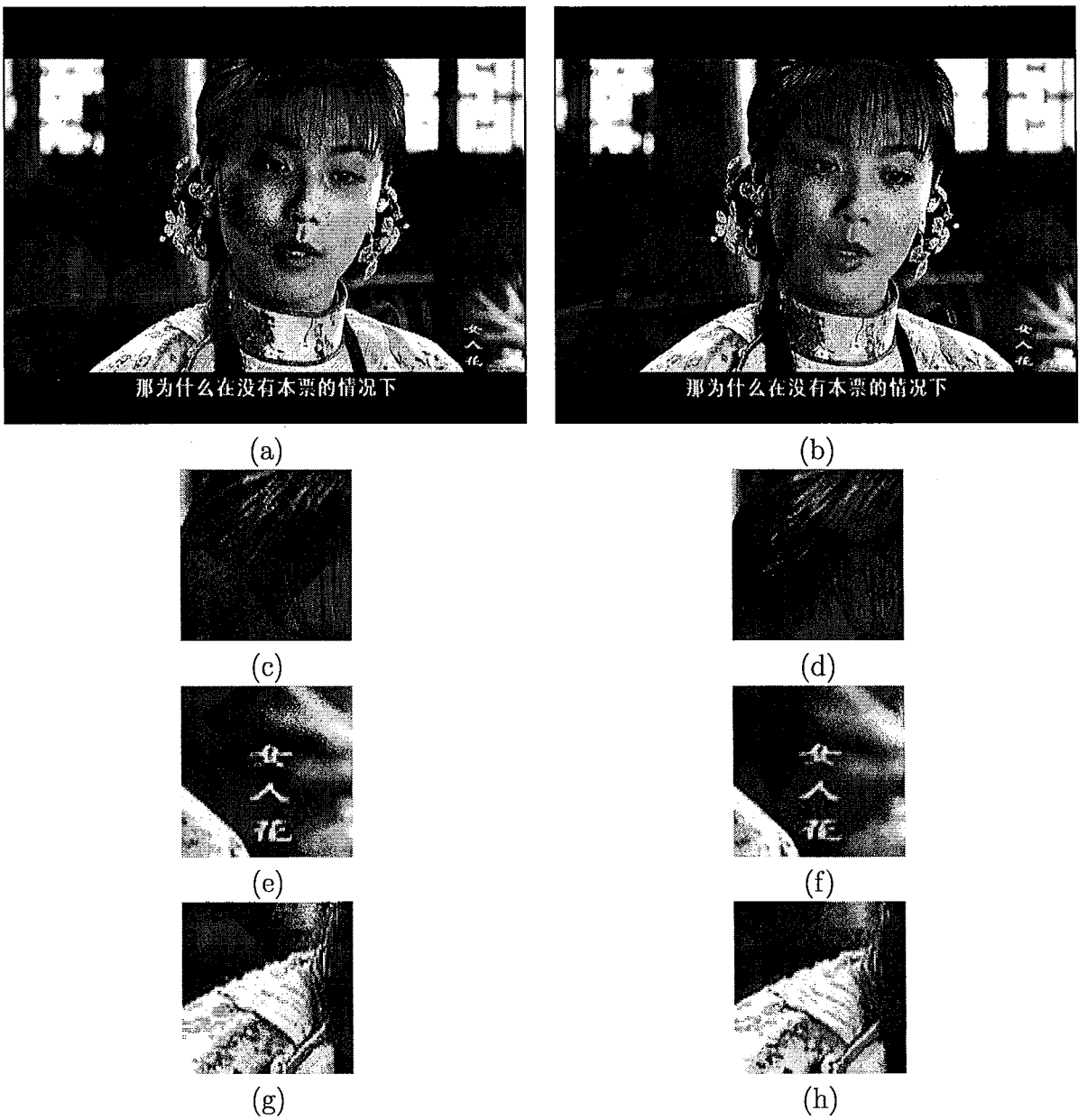
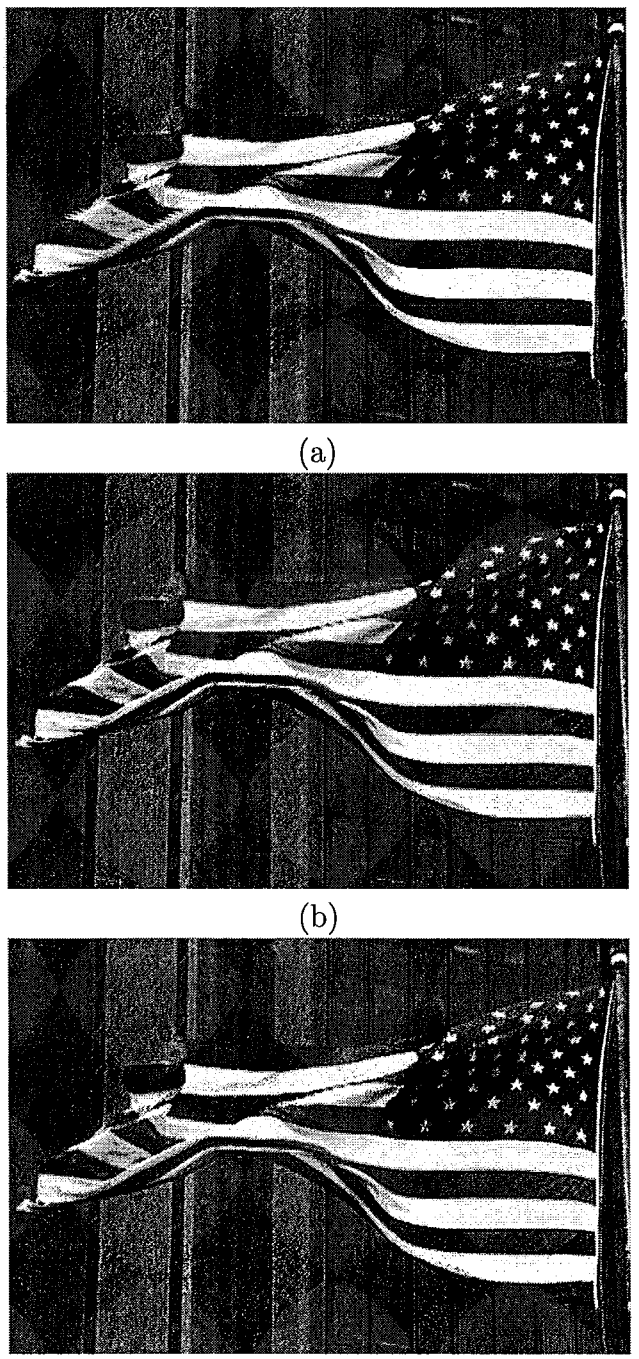


Figure 5.7: Comparison of different methods. (a) (c) (e) (g) Bicubic interpolation (b) (d) (f) (h) Proposed intrafield interpolation

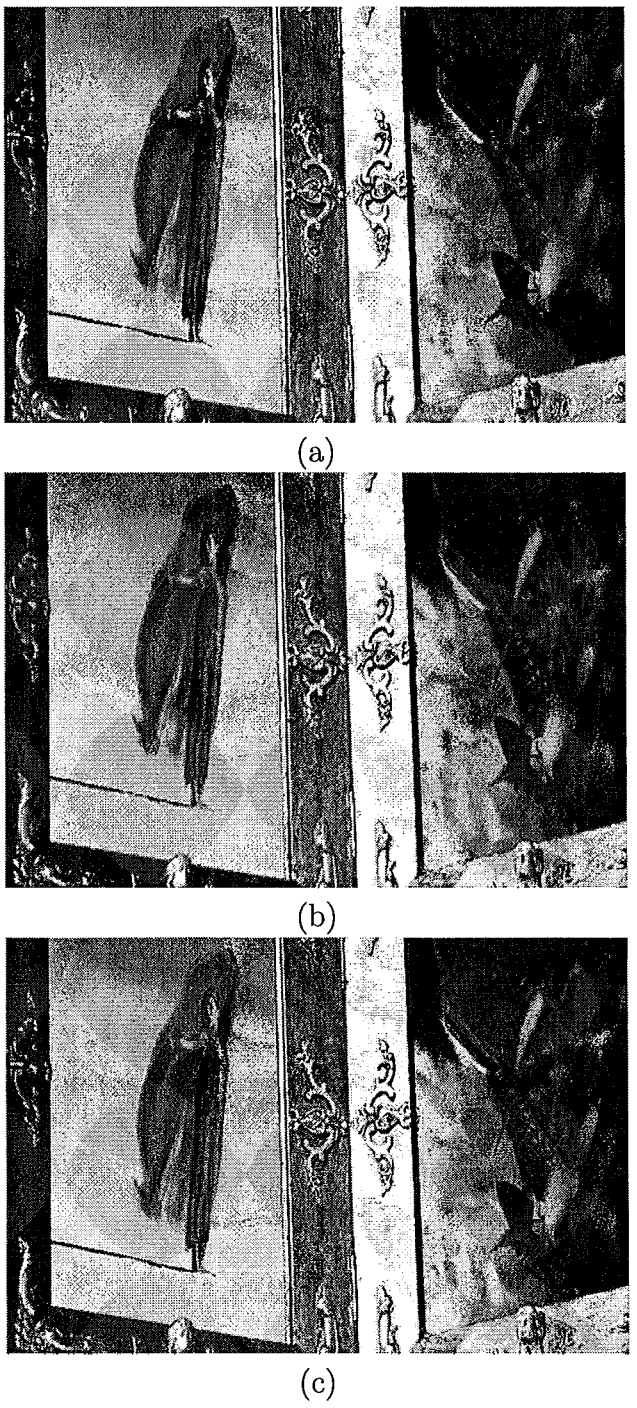


(a)

(b)

(c)

Figure 5.8: Comparison of different methods. (a)Bilinear field averaging (b)Proposed intrafield interpolation (c)Proposed motion adaptive deinterlacing



(a)

(b)

(c)

Figure 5.9: Comparison of different methods. (a)Bilinear field averaging (b)Proposed intrafield interpolation (c)Proposed motion adaptive deinterlacing

Chapter 6

Conclusions and Future Work

In this thesis, we reexamined the problems of image/video spatial resolution upconversion and video deinterlacing, aiming at real-time, adaptive solutions of these problems. In particular, we investigate the use of GPU technology to massively parallelize the computations involved in adaptive video scaling and deinterlacing. This is realized by novel directional interpolation algorithms for both scaling and deinterlacing and their GPU implementations. The future work includes further improvement of the visual quality of the proposed video deinterlacing and resolution upconversion algorithms, and a CUDA-based implementation for real-time SDTV to HDTV upconversion.

Bibliography

[1] NVIDIA, *The NVIDIA CUDA Compute Unified Device Architecture Programming Guide 2.0*, 2008.

[2] X. Wu, X. Zhang, and X. Wang, "Low bit-rate image compression via adaptive down-sampling and constrained least squares upconversion," *IEEE Trans. Image Processing*, vol. 18, no. 3, pp. 552–561, Mar. 2009.

[3] X. Zhang and X. Wu, "Standard-compliant multiple description image coding by spatial multiplexing and constrained least-squares restoration," in *Proc. IEEE 10th Workshop on Multimedia Signal Processing*, Oct. 2008, pp. 349–354.

[4] J. Cao, M. C. Che, X. Wu, and J. Liang, "GPU-aided directional image/video interpolation for real time resolution upconversion," in *Multimedia Signal Processing, 2009. MMSP '09. IEEE International Workshop on*, Rio De Janeiro, 2009, pp. 1–6.

[5] X. Wu and J. Cao, "GPU-aided motion adaptive video deinterlacing," in *SPIE Visual Information Processing and Communication Conference*, San Jose, California, USA, 2010, vol. 7543.

[6] S. Carrato and L. Tenze, "A high quality 2x image interpolator," *IEEE Signal Processing Letters*, vol. 7, no. 6, pp. 132–134, 2000.

[7] X. Li and M. T. Orchard, "New edge-directed interpolation," *IEEE Trans. Image Processing*, vol. 10, no. 10, pp. 1521–1527, Oct. 2001.

[8] X. Zhang and X. Wu, "Image interpolation by adaptive 2-D autoregressive modeling and soft-decision estimation," *IEEE Trans. Image Processing*, vol. 17, no. 6, pp. 887–896, Jun. 2008.

[9] L. Zhang and X. Wu, "An edge-guided image interpolation algorithm via directional filtering and data fusion," *IEEE Trans. Image Processing*, vol. 15, no. 8, pp. 2226–2238, Aug. 2006.

[10] G. De Haan and E. B. Bellers, "Deinterlacing-an overview," *Proceedings of the IEEE*, vol. 86, no. 9, pp. 1839–1857, Sept. 1998.

[11] J. Kovacevic, R. J. Safranek, and E. M. Yeh, "Deinterlacing by successive approximation," *IEEE Transactions on Image Processing*, vol. 6, no. 2, pp. 339–344, Feb. 1997.

[12] G. De Haan and E. B. Bellers, "De-interlacing of video data," *IEEE Transactions on Consumer Electronics*, vol. 43, no. 3, pp. 819–825, Aug. 1997.

[13] S. H. Keller, F. Lauze, and M. Nielsen, "Deinterlacing using variational methods," *IEEE Transactions on Image Processing*, vol. 17, no. 11, pp. 2015–2028, Nov. 2008.

[14] S. Yang, D. Kim, and J. Jeong, "Fine edge-preserving deinterlacing algorithm

for progressive display," *IEEE Transactions on Consumer Electronics*, vol. 55, no. 3, pp. 1654–1662, 2009.

[15] S. J. Park, G. Jeon, and J. Jeong, "Deinterlacing algorithm using edge direction from analysis of the DCT coefficient distribution," *IEEE Transactions on Consumer Electronics*, vol. 55, no. 3, pp. 1674–1681, 2009.

[16] M. H. Lee, J. H. Kim, J. S. Lee, K. K. Ryu, and D. I. Song, "A new algorithm for interlaced to progressive scan conversion based on directional correlations and its IC design," *IEEE Transactions on Consumer Electronics*, vol. 40, no. 2, pp. 119–129, May 1994.

[17] C. J. Kuo, C. Liao, and C. C. Lin, "Adaptive interpolation technique for scanning rate conversion," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 6, no. 3, pp. 317–321, 1996.

[18] G. Thomas, "A comparison of motion-compensated interlace-to-progressive conversion methods," *Signal Processing, Image Communication*, vol. 12, no. 3, pp. 209–229, 1998.

[19] A. Skarabot, G. Ramponi, and D. Toffoli, "Image sequence processing for videowall visualization," in *Proc. SPIE*, 2000, vol. 3961, pp. 138–147.

[20] A. Brunton and J. Zhao, "Real-time video watermarking on programmable graphics hardware," in *Electrical and Computer Engineering, 2005. Canadian Conference on*, Saskatoon, Sask., pp. 1312–1315.

[21] G. Shen, G. P. Gao, S. Li, H. Y. Shum, and Y. Q. Zhang, "Accelerate video

decoding with generic GPU," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 15, no. 5, pp. 685–693.

[22] W. N. Chen and H. M. Hang, "H.264/avc motion estimation implmentation on compute unified device architecture (cuda)," in *Proc. IEEE International Conference on Multimedia and Expo*, June 23 2008–April 26 2008, pp. 697–700.

[23] S. T. Yang, T. K. Lin, and S. Y. Chien, "Real-time motion estimation for 1080p videos on graphics processing units with shared memory optimization," in *Signal Processing Systems, 2009. SiPS 2009. IEEE Workshop on*, Tampere, pp. 297–302.

[24] S. Shimizu, H. Kimata, and Y. Ohtani, "Real-time free-viewpoint viewer from multiview video plus depth representation coded by h.264/AVC MVC extension," in *3DTV Conference: The True Vision - Capture, Transmission and Display of 3D Video, 2009*, Potsdam, pp. 1–4.

[25] H. Hou and H. Andrews, "Cubic splines for image interpolation and digital filtering," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 26, no. 6, pp. 508–517, Dec. 1978.

[26] R. Keys, "Cubic convolution interpolation for digital image processing," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 29, no. 6, pp. 1153–1160, Dec. 1981.