GPU AND CELL PHONE-AIDED MULTIMEDIA PROCESSING

by

Ming-Chao Che
B.A.Sc., Simon Fraser University, 2007

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

In the
School of Engineering Science

© Ming-Chao Che 2010
SIMON FRASER UNIVERSITY
Spring, 2011

All rights reserved. However, in accordance with the *Copyright Act of Canada*, this work may be reproduced, without authorization, under the conditions for *Fair Dealing*. Therefore, limited reproduction of this work for the purposes of private study, research, criticism, review and news reporting is likely to be in accordance with the law, particularly if cited appropriately.

APPROVAL

Name: Ming-Chao Calvin Che

Degree: Masters of Applied Science

Title of Thesis: GPU and Cell Phone-Aided Multimedia Processing

Examining Committee:

Chair: **Dr. Sami (Hakam) Muhaidat**
Assistant Professor, School of Engineering Science

Dr. Jie Liang
Senior Supervisor
Associate Professor, School of Engineering Science

Dr. Faisal Beg
Supervisor
Associate Professor, School of Engineering Science

Dr. Jiangchuan Liu
Internal Examiner
Associate Professor
School of Computing Science, Simon Fraser University

Date Defended/Approved: January 14, 2011

SIMON FRASER UNIVERSITY
LIBRARY

Declaration of Partial Copyright Licence

The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the right to lend this thesis, project or extended essay to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users.

The author has further granted permission to Simon Fraser University to keep or make a digital copy for use in its circulating collection (currently available to the public at the “Institutional Repository” link of the SFU Library website <www.lib.sfu.ca> at: <http://ir.lib.sfu.ca/handle/1892/112>) and, without changing the content, to translate the thesis/project or extended essays, if technically possible, to any medium or format for the purpose of preservation of the digital work.

The author has further agreed that permission for multiple copying of this work for scholarly purposes may be granted by either the author or the Dean of Graduate Studies.

It is understood that copying or publication of this work for financial gain shall not be allowed without the author’s written permission.

Permission for public performance, or limited permission for private scholarly use, of any multimedia materials forming part of this work, may have been granted by the author. This information may be found on the separately catalogued multimedia material and in the signed Partial Copyright Licence.

While licensing SFU to permit the above uses, the author retains copyright in the thesis, project or extended essays, including the right to change the work for subsequent purposes, including editing and publishing the work in whole or in part, and licensing other parties, as the author may desire.

The original Partial Copyright Licence attesting to these terms, and signed by this author, may be found in the original bound copy of this work, retained in the Simon Fraser University Archive.

Simon Fraser University Library
Burnaby, BC, Canada

Last revision: Spring 09

ABSTRACT

Computing technology has been evolving rapidly during the past decades. New ideas and inventions are constantly developed to improve usability and processing power of applications. This thesis develops a multimedia processing system that uses GPU and cell phone to improve speed and user experience. The CUDA framework developed by NVIDIA turns the GPU into a manycore coprocessor of the CPU. We show in this thesis that by taking advantage of GPU computing, algorithms such as image encoding and resolution upconversion can be up to five times as fast. We also develop algorithms to use accelerometer-equipped cell phone as a remote controller to improve user interaction. When user performs various actions, input command is sent to the PC via Bluetooth and identified using motion detection algorithms. Some applications of these tools are presented, including image slideshow, multiview video viewer, as well as cell phone aided Google Map application and web browser.

Keywords: GPU programming; CUDA; resolution upconversion; image coding; JPEG-XR; accelerometer; motion detection; Bluetooth; Silverlight; Google Street View

ACKNOWLEDGEMENTS

I would like to express my gratitude to many people as this thesis would not be possible without their support and guidance.

First, I would like to thank my supervisor Dr. Jie Liang for providing various ideas, tips, and guidance throughout the course of my MASc study. From his source coding and signal processing classes, I also gained valuable knowledge that helps me in writing of this thesis.

I want to thanks Dr. Xiaolin Wu for his image/video resolution upsampling algorithm, which was used for the GPU implementation. I also thank Dr. Mirza Faisal Beg and Dr. Jiangchuan Liu for being my committee member, and Dr. Sami Muhaidat for chairing my defense.

I like to thank Nokia for providing the cell phones used in the research. Without the hardware and their online documentations, this thesis would not be possible.

Finally, I want to thank all my labmates in the multimedia communication lab, as well as all my friends, for making my MASc study enjoyable.

TABLE OF CONTENTS

Approval ... ii
Abstract ... iii
Acknowledgements ... iv
Table of Contents ... v
List of Figures ... vii
List of Tables ... ix

Chapter 1: Introduction ... 1

1.1 Introduction ... 1
1.2 Main Contribution ... 3
1.3 Thesis Outline ... 5

Chapter 2: GPU Implementation of Image Resolution Upsampling ... 7

2.1 Background in GPU Programming ... 7
 2.1.1 GPU Architecture and General-Purpose GPU Programming ... 8
 2.1.2 Fundamentals of CUDA Programming ... 9
 2.1.3 Performance Guideline ... 13
2.2 Directional Image/Video Spatial Resolution Upconversion ... 17
2.3 CUDA Implementation of the Directional Interpolation Algorithm ... 22
 2.3.1 Branch and Loop Replacement ... 24
 2.3.2 Thread Configuration ... 25
 2.3.3 Float Data Type ... 26
 2.3.4 Four Sub-Images ... 26
 2.3.5 Using Shared Memory ... 29
2.4 Experimental Result ... 33
 2.4.1 Interpolation Performance ... 34
 2.4.2 Speed Performance ... 34

Chapter 3: GPU Implementation of JPEG-XR ... 38

3.1 Background of Image/Video Coding and JPEG-XR ... 38
 3.1.1 Overview of JPEG-XR ... 39
3.2 CUDA Implementation of JPEG-XR Encoder ... 41
 3.2.1 Input and Colour Space Conversion ... 42
 3.2.2 Downsampling ... 46
 3.2.3 Transform ... 48
 3.2.4 Quantization ... 52
 3.2.5 DCAC Prediction, Coefficient Scanning, and Entropy Coding ... 53
3.3 Experimental Result ... 54
 3.3.1 Speed Performance for Each Code Segment ... 54
 3.3.2 Speed Performance for Different Output Types ... 56

3.3.3 Speed Performance for Different Image Sizes 57

Chapter 4: Cellphone Aided Multimedia Processing 59

4.1 Introduction 59
4.2 Cellphone as Input Device 60
4.2.1 Rotations 64
4.2.2 Shaking Movements 68
4.2.3 Motion Matching 69
4.3 Communication with PC 73
4.3.1 Bluetooth Communication between Phone and PC Server 74
4.3.2 Communication between Bluetooth Server and Web Content 77
4.4 Implementations and Results 80
4.4.1 TiltShow 81
4.4.2 Veaver 82
4.4.3 Turbo Street View 85
4.4.4 Cellphone Aided Web Browser 92

Chapter 5: Conclusion and Future Work 93

5.1 Conclusion 93
5.2 Future Work 94
5.2.1 GPU Programming Component 95
5.2.2 Motion Detection and Bluetooth Communication Component 97
5.2.3 Google Street View Component 98

Reference List 101

LIST OF FIGURES

Figure 1.1 Illustration of the interactive multimedia manipulation system 4
Figure 2.1 The difference between CPU and GPU architecture 8
Figure 2.2 Heterogeneous programming model 10
Figure 2.3 A kernel launch with threads organized into grids and blocks 11
Figure 2.4 Memory architecture of a CUDA GPU 12
Figure 2.5 Flow chart of memory access for a typical GPU program 16
Figure 2.6 Two-pass interpolation process 18
Figure 2.7 Diagonal cubic interpolation for both 35 and 135 degrees 19
Figure 2.8 Verification for both 45 and 135 degrees interpolators 20
Figure 2.9 Error interpolation window W for the 135 degrees interpolator 21
Figure 2.10 General program flow for GPU implementation of the interpolation algorithm 23
Figure 2.11 Original CPU memory allocation 27
Figure 2.12 Uncoalesced memory access produced with original CPU implementation 27
Figure 2.13 Position of pixels on each sub-image relative to the original pixel 28
Figure 2.14 GPU memory allocation 28
Figure 2.15 Coalesced memory access produced with new GPU implementation 29
Figure 2.16 Size of cached data in shared memory compared to block size 30
Figure 2.17 Global to shared memory transfer mechanism 31
Figure 2.18 Bank conflict occurs when flipping pixels around image boundary 32
Figure 2.19 Conflict-free memory access when flipping pixels around image edge 33
Figure 2.20 Comparison of different interpolation methods 34
Figure 3.1 Block diagram for image and video coding 39
Figure 3.2 Comparison between memory arrangements of CPU and GPU implementations 42
Figure 3.3 Memory access pattern of RGB input stream from global memory 43
Figure 3.4 Memory access pattern of RGB input stream from after memory caching using shared memory 44

Figure 3.5 The downsampling process ... 46
Figure 3.6 The block boundary for PCT and POT ... 49
Figure 3.7 Memory configuration for transform operation ... 51
Figure 3.8 The three kernels for first stage POT on a rectangular image ... 52
Figure 4.1 The x, y, and z axis of the accelerometer relative to the cellphone ... 61
Figure 4.2 Fluctuations in accelerometer value when the phone is resting still ... 62
Figure 4.3 Accelerometer values after filtering to reduce fluctuations ... 63
Figure 4.4 Accelerometer data for a rotate right motion ... 64
Figure 4.5 Change in gravitational acceleration when the phone is tilted ... 65
Figure 4.6 Accelerometer data for a rotate up motion ... 66
Figure 4.7 An example for the detection of rotational motion ... 67
Figure 4.8 The accelerometer reading for a typical shaking motion ... 68
Figure 4.9 The accelerometer reading when flipping the phone upside down ... 70
Figure 4.10 The accelerometer reading when rotating the phone twice horizontally ... 70
Figure 4.11 The accelerometer reading when raising the phone upward ... 71
Figure 4.12 A pre-recorded data for the flipping phone upside down motion ... 71
Figure 4.13 Links between various parts of the cellphone-to-web content communication ... 74
Figure 4.14 Links between server application and web content using windows API, COM, and OLE (Internet Explorer implementation) ... 78
Figure 4.15 Detail for the execution of scripts in embedded browser window ... 78
Figure 4.16 Links between server application and web content using browser plugin (Firefox implementation) ... 79
Figure 4.17 Diagram of the Veaver system ... 83
Figure 4.18 The interface for the Veaver application ... 84
Figure 4.19 The interface for the Turbo Street View application ... 86
Figure 4.20 Illustration of navigation links of a Street View bubble ... 87

LIST OF TABLES

Table 2.1 Comparison of memory types 12

Table 2.2 Maximum available resource on device 13

Table 2.3 Comparison of different thread block dimension 25

Table 2.4 Execution time measured by shell for a 256x256 Lena image........ 35

Table 2.5 Execution time measured by CUDA for a 256x256 Lena image........ 36

Table 2.6 Execution time measured by CUDA for larger image 36

Table 3.1 Evaluation of the trade-offs between register and local memory use........ 48

Table 3.2 Options to assign pixels per thread during transform operations........ 50

Table 3.3 Comparison between CPU and GPU processing time for various blocks for the JPEG XR encoder 55

Table 3.4 Comparison between CPU and GPU processing time for various types of images – to quantization only 56

Table 3.5 Comparison between CPU and GPU processing time for various types of images – full encoder 57

Table 3.6 Comparison between CPU and GPU processing time for various sizes of images – to quantization only........ 58

Table 3.7 Comparison between CPU and GPU processing time for various sizes of images – full encoder........ 58

Table 4.1 Default key configuration for each command 63

Table 4.2 Default threshold values used for rotational motions........ 68

Table 4.3 The content of the 10-bytes packets used for the Bluetooth link 75

Table 4.4 The command codes 76

Table 4.5 List of commands and the corresponding JavaScript function........ 80

Table 4.6 List of commands implemented in the TiltShow application........ 81

Table 4.7 List of commands implemented in the Veaver application........ 85

Table 4.8 List of commands implemented in the Turbo Street View application........ 88

Table 4.9 Comparison of Street View panorama loading time with or without prefetching........ 90

Table 4.10 Comparison of Street View panorama loading time with different test environments........ 91

Table 4.11 List of commands implemented in the cellphone aided web browser........ 92

Table 5.1 Maximum available resource on device 96
Table 5.2 Maximum available memory resource per thread..................................... 96
Table 5.3 Proposed prefetch order based on the previous command 100

CHAPTER 1: INTRODUCTION

1.1 Introduction

Technology has changed our life dramatically ever since the late 20th century. Computer, network, and multimedia entertainment system now play a major role in our daily life as they govern how we work, play, and communicate with each other. As technology continues to evolve, new ideas and inventions are constantly developed to increase user interaction and improve user experience. Inputs are no longer limited to keyboard and buttons. Instead, new devices can now capture voices, movements, and gestures. Users are no longer required to sit beside the system or a device attached to the system, as the input can now be sent wirelessly using various technologies such as Bluetooth and Infrared. Inputs from mice and pointing devices are no longer confined to a 2D plane, as new devices with 3D tracking capability now available. A prime example in the recent years would the Wii console system developed by Nintendo. Wii remote, replacing the game controllers, is now able to track movements in a 3D space using its built-in accelerometer and send the data back to the console using Bluetooth technology [1]. Instead of just pushing buttons, users can enjoy the game as if they are physically in the game world.

With the help of this new idea and technology, gaming experience increased tremendously.

The improving technology also reduces the cost of multimedia production and increases the demand of higher quality content. As the cost of camera and other capturing device reduces, building a multi-camera system becomes more feasible. As a result, there has been an increasing interest in studying multi-camera and multi-view image and video systems in the recent years. There are many new applications emerging, including 3DTV, where multiple cameras are used to capture multiple views of a scene to offer 3D depth impression [2, 3]. Another application of this system is free viewpoint video, where the user is able to freely switching between different viewpoints captured by different cameras, as if they are navigating in a 3D world [3]. An implementation of this system includes Microsoft PhotoSynth, where many photos of the same object at different viewpoints are stitched together to create a 3D scene where user can freely navigate [4]. Similarly, Google Street View captures panoramic imagery of streets using camera arrays mounted at the roof of a car to create an environment where user can navigate as if they are walking on the street [5].

With the increase of viewpoints, the amount of data such multimedia system required to process also increases exponentially. As bandwidth and storage space are at a premium, image/video compression and downsampling are often used to reduce the bitrate and lower the cost of such system. This greatly increases the need of processing power as multiple images or videos are processed simultaneously.

In the recent years, computer processors are moving away from single core single threaded design into a system that contains multiple cores and is able to process multiple threads at once. The GPU (Graphics Processing Unit) was traditionally designed solely to offload graphics processing off the central processing unit (CPU). It is powerful as its Single Instruction Multiple Data (SIMD) architecture can process an array of data in parallel by executing a single instruction. Originally, the GPU does not support integer and scattered memory operations, which makes it difficult for adaptation in general computing. With the recent introduction of the CUDA (Compute Unified Device Architecture) framework developed by NVIDIA, the GPU has evolved into a powerful highly parallel, multithreaded, and manycore co-processor. The CUDA framework greatly simplifies GPU programming, and enables the GPU to solve complex computing problems with large set of data to be processed in parallel. This property makes it suitable for image and video processing.

1.2 Main Contribution

The works in this thesis are parts of an interactive multimedia manipulation system. The system involves the display of multi-view images and/or videos simultaneously on screen that allows user to navigate freely, similar to Microsoft PhotoSynth and Google Street View. One main drawback of both PhotoSynth and Street View systems is that the navigation involves a large amount of mouse clicking or dragging, which is often tedious, time consuming,

and greatly degrade the user experience. We solve this issue by implementing a remote controller using cell phone and communicate with the computer using Bluetooth, similar to a Wii remote.

Typical multimedia applications involving images and videos require hardware with high computational power. Multiview system and the ease of navigation introduced by the remote controller further increase this burden, due to the higher data rate and more frequent command inputs. To ensure that the system is able to keep up with commands and capable of decoding the multiview image and/or video in real-time, GPU-based image and video processing is used as it provides higher image and video processing capabilities than CPU. This complements the cell phone remote controller in making the system faster, more interactive, and more responsive to user commands. A diagram illustrate the components in our system is shown in Figure 1.1.

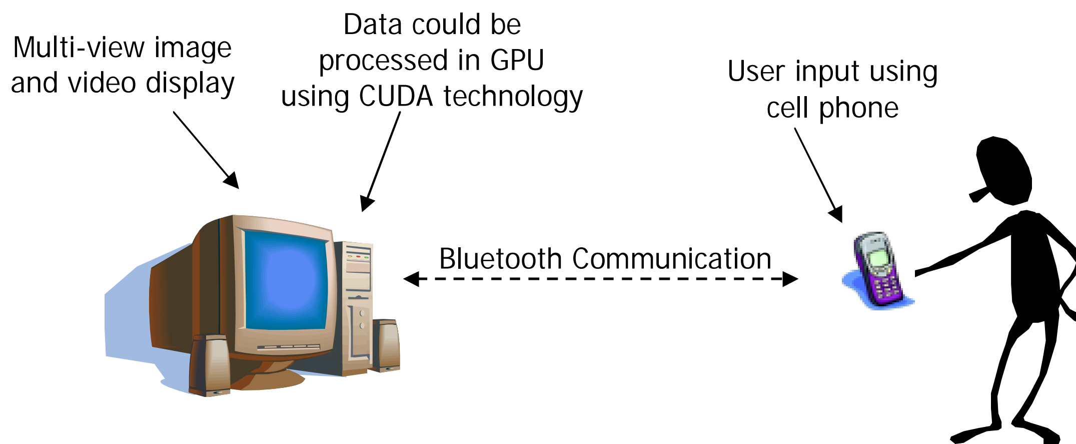

Figure 1.1 Illustration of the interactive multimedia manipulation system

1.3 Thesis Outline

This thesis contains detailed descriptions for three of the components in the multimedia manipulation system. Chapter 2 describes the fundamental of CUDA and GPU programming and discusses about the implementation of an image/video upsampling algorithm using CUDA. This implementation allows processing of video in real-time, which can be useful in the system when user choose to zoom in on an image or video without the need for additional bandwidth for the larger image or video content. The result for this part of the thesis has been published in [6].

Chapter 3 discusses about speeding up image compression, and describes an implementation of a JPEG-XR image encoder using CUDA. The implementation focuses on the first few stages of the process – colour space conversion, downsampling, transform, and quantization. With the similarities between image and video encoding and decoding, the benefit of image encoding using GPU can also be applied to video decoding. This component can be used in the multimedia system to ensure that the system is capable of decoding multiple videos in real-time when the user switches view. The result for this part of the thesis has been published in [7].

Chapter 4 presents the controller and interactive component of the system. The first part of the chapter discusses about the use of accelerometer on the cell phone to capture movements and the algorithm to correctly identify the motion. The second part describes the use of Bluetooth to establish a communication link between the phone and the application on the computer, so

the cellphone input command can be received and processed. The final part of the chapter describes the implementation of four different applications developed using this system.

CHAPTER 2: GPU IMPLEMENTATION OF IMAGE RESOLUTION UPSAMPLING

2.1 Background in GPU Programming

Graphic processing unit (GPU) is a dedicated device for manipulating and displaying computer graphics. Originally developed solely for graphic processing, it has now evolved into a highly parallel, multithreaded co-processor of the CPU. GPU programming is mostly software based, which gives an advantage over some hardware-based implementations, such as FPGA. GPU implementation is much faster to develop and implement as no VHDL and hardware knowledge is required. Given the same cost, it also provides better capabilities compared to hardware implementations. Compared to multi-core CPUs, GPU implementation is also more desirable for highly data parallel application as the GPU scales much faster. In the past few years, NVIDIA GPU had improved from 8 to 512 cores while CPU cores only increased from 1 to 8 [7]. The latest GPU now has close to 10 times as much computation power and memory bandwidth as compared to CPU [7].

This section of the report will describe the fundamentals of programming with CUDA, and discuss the requirement to achieve high speed and efficiency.

2.1.1 GPU Architecture and General-Purpose GPU Programming

The computation power of GPU comes from its Single Instruction Multiple Data (SIMD) architecture, where an array of data can be processed with a single instruction. Compare to the CPU architecture where each instruction can only process a single byte of data, the SIMD architecture is much more efficient in applications where there is a large set of data, such as image and video processing. Figure 2.1 shows the different between CPU and GPU architecture, where the GPU devotes more transistors to data processing (ALU units) rather than flow control (control and cache).

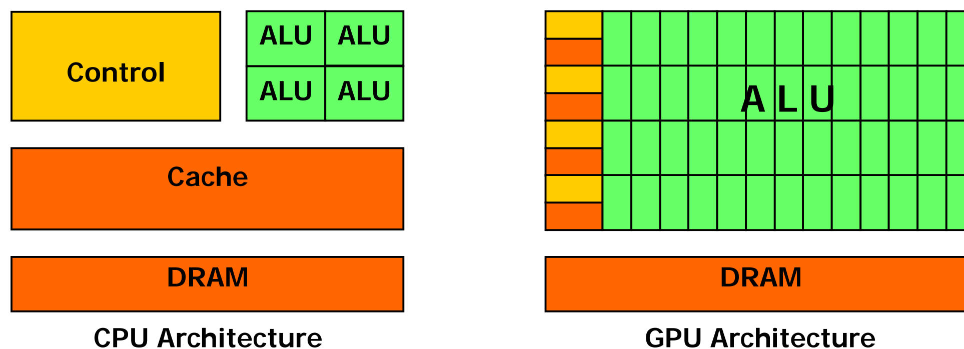

Figure 2.1 The difference between CPU and GPU architecture

To use the computation power of GPU in non-graphics applications, General-Purpose computation on Graphics Processing Units (GPGPU) was developed in 2002, which was a mechanism that used GPU to speed up the critical path of the application [9]. This approach, however, utilizes graphics API that does not support integer operations and scatter memory writes, which make implementation quite difficult [10]. In 2007, NVIDIA released Compute Unified

Device Architecture (CUDA) – a GPGPU technology programmable in C language that effectively turns GPU into a co-processor of the CPU. The C language extension along with the new support for integer operations and scatter writes makes it much easier to develop general-purpose, non-graphical GPU applications that can exploit the SIMD architecture of GPU to its full potential.

2.1.2 Fundamentals of CUDA Programming

In the CUDA framework, each GPU consists of an array of streaming multiprocessor (SM) and each multiprocessor consists of many scalar processors (SP). Each multiprocessor can process multiple blocks of threads simultaneously – up to 768 threads for older model and 1024 for the newer ones. Their power comes from the SIMD architecture, where the light-weighted threads executes the same instruction in parallel on an array of data, with each element of the array handled by a different thread. CUDA uses heterogeneous programming model, 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. Figure 2.2 shows an example of the heterogeneous programming model, where the program alternates between single-threaded host functions on CPU and multi-threaded kernel functions on GPU.

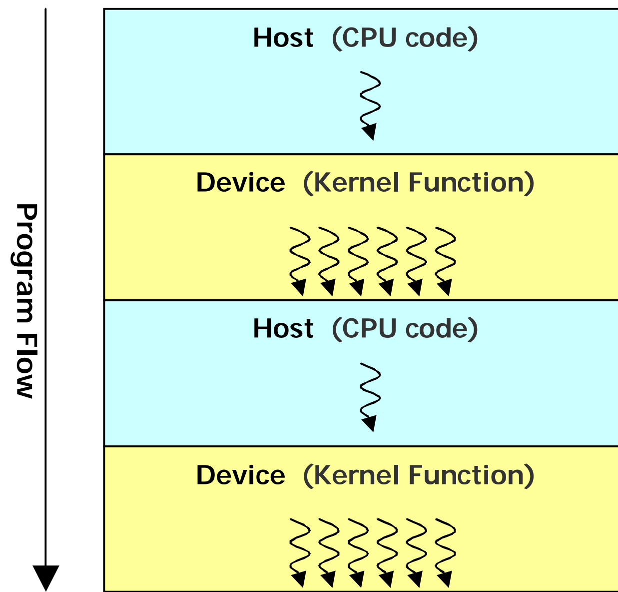

Figure 2.2 Heterogeneous programming model

Threads in the kernel function are ordered into blocks, and blocks are ordered into a grid, as shown in Figure 2.3. A 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. In each thread block, a group of 32 threads forms a warp, and each half-warp, consists of 16 threads, is executed together under the same instruction.

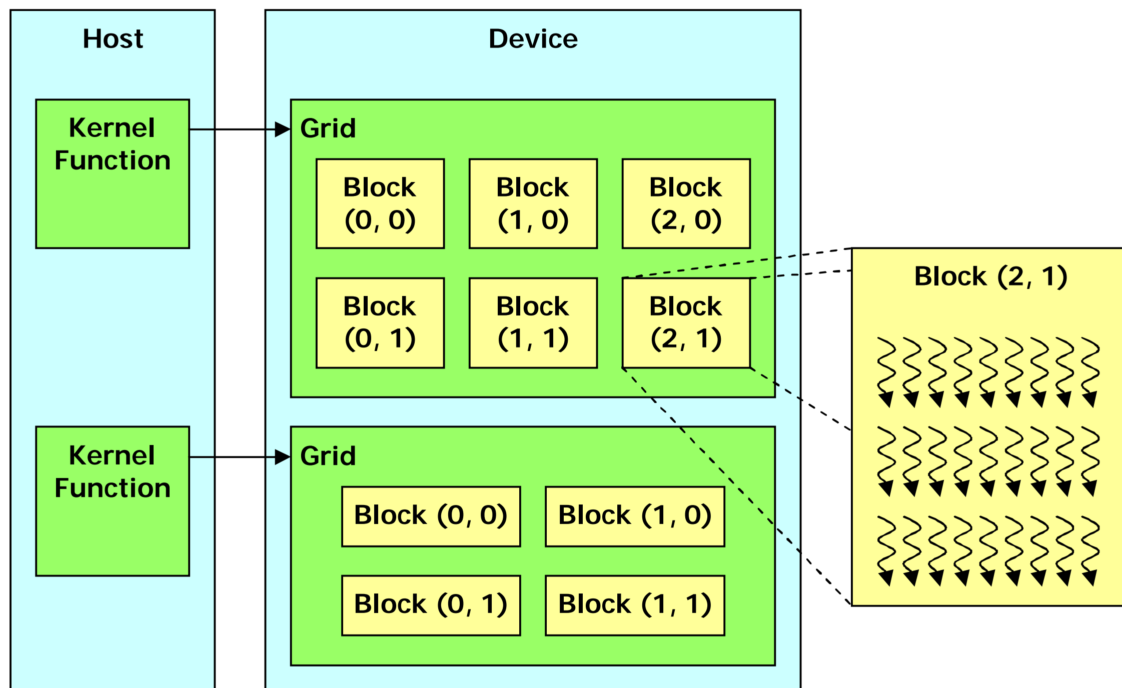

Figure 2.3 A kernel launch with threads organized into grids and blocks

There are six different types of memory available on GPU: global, constant, local, shared, texture, and registers. Most data are stored in global memory as it has the lifetime of the application and can be read and write by both host and kernel functions. However, the memory is off-chip, which means long access time from the kernel function. Both shared memory and registers are on-chip with fast access time but only has the lifetime of the kernel function. Thus they are used mostly for manipulations and calculations within a thread or a block of threads. Both constant and texture memories are off-chip read-only memory. However, fast memory access can be obtained as part of their content is cached onto an on-chip memory upon thread initialization. Figure 2.4 shows the hardware memory model for streaming multiprocessors [7]. The device memory

on the figure consists of global, local, constant, and texture memories. Table 2.1 compares and summarizes the features and constraints of the different memory types.

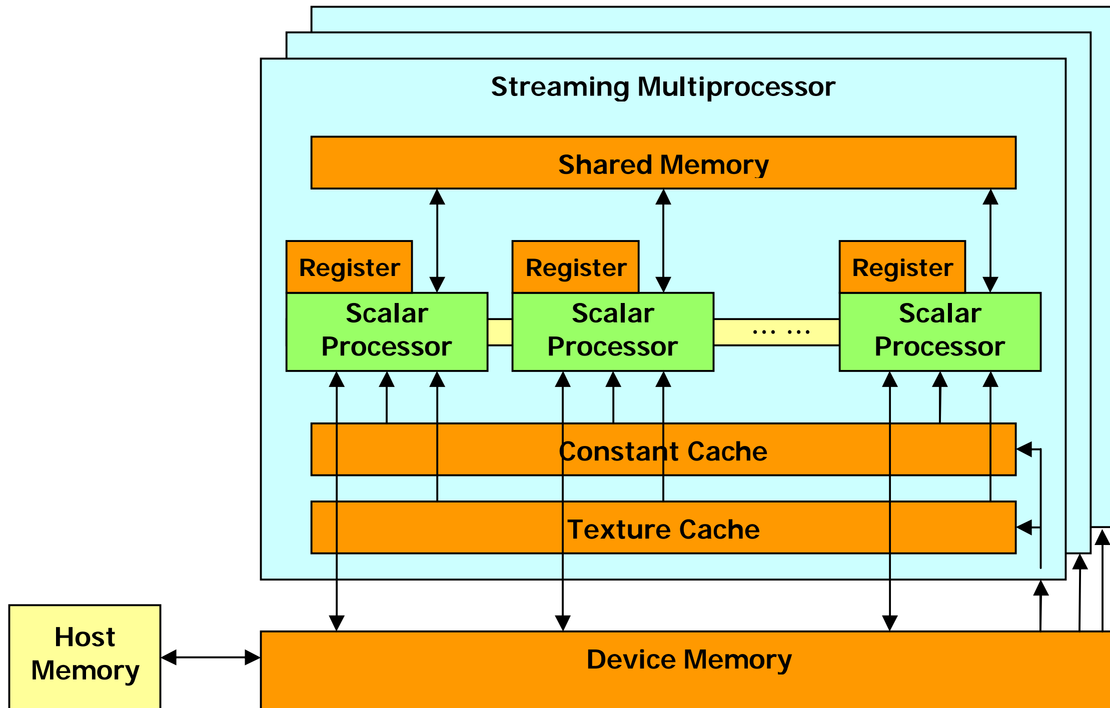

Figure 2.4 Memory architecture of a CUDA GPU

Table 2.1 Comparison of memory types

Type	Scope	Lifetime	On-Chip	Cached	Access	Size	Host Write	Device Write
Global	Grid	App.			Slow	-	√	√
Constant	Grid	App.		√	Fast	8 KB	√	
Local	Thread	Thread			Slow	-		√
Shared	Block	Thread	√		Fast	16 KB		√
Texture	Grid	App.		√	Fast	6~8 KB	√	
Register	Thread	Thread	√		Fast	8192		√

The number of threads that can be executed parallel by a multiprocessor is limited by both hardware resource and memory usage. Table 2.2 shows the maximum resource on a multiprocessor for different generations of GPU [7].

Table 2.2 Maximum available resource on device

Item	**Maximum for Compute Capability**	
	1.0, 1.1	**1.2, 1.3**
Threads per multiprocessor	768	1024
Threads per block	512	512
Block dimension (*x*, *y*, *z*)	512, 512, 64	512, 512, 64
Grid dimension (*x*, *y*)	65535, 65535	65535, 65535
Active blocks per multiprocessor	8	8
Active warp per multiprocessor	24	32
Register per multiprocessor	8192	16384
Shared memory per multiprocessor	16 KB	16 KB
Local memory per thread	16 KB	16 KB
Constant memory	64 KB	64 KB

2.1.3 Performance Guideline

For a GPU application to effectively use all available resources and achieve maximum performance, we must make sure that it can [7]:

1. Maximize parallel execution.

2. Optimize instruction usage to achieve maximum instruction throughput.

3. Optimize memory usage to achieve maximum memory bandwidth.

To **maximize parallel execution**, dimension of block and grid should be chosen so that all available threads on the streaming multiprocessor are put to use. Since a warp of 32 threads is executed together on the streaming processor, the block size should be set to a multiple of 32 to prevent under-populated warp. However, larger block sizes are used to ensure the scalability of the program on future GPU devices with possibly larger number of threads supported per processor. Generally, block size of 192 or 256 yields the best result [7].

Memory use poses another constraint on numbers of parallel executions. In order to achieve full occupancy and obtain maximum performance, each thread within a block can use at most 10 registers and 16 bytes of shared memory space. If the function requires more than 10 registers, local memory may be used to ensure full thread occupancy.

To achieve **higher instruction throughput**, functions that takes large amount of clock cycles to execute should be avoided. Examples of such functions are integer division and modulo. Also, CUDA and NVIDIA GPU currently only have native support for `int`, `long`, and `float` data types, other data types such as char and double should not be used [7].

Under SIMD architecture, each thread under the same warp or half-warp must execute the same instruction. Therefore, full efficiency can only be achieved if all threads within the warp agree on their execution path. If the warp diverges in a conditional branch, then both branches must be executed with some threads disabled in each run. To reduce the possibility of diverging warp, branch and loop functions such as `if`, `switch`, `for`, `do`, and `while` should be avoided as much as possible unless it is sure that all threads under the warp will follow the same path [7].

To **optimize memory access** for higher bandwidth, faster memory such as texture, shared, constant memory and registers should be used more often while access to global and local memory should be minimized. Also, memory transfer between host and device should be limited to start and end of the program only. With newer version of CUDA, memory transfer can be performed asynchronously. Asynchronous memory transfer runs in parallel with kernel function, which saves execution time [7].

Figure 2.5 shows a flowchart of memory access for a typical GPU application. Note that after each memory access inside the kernel function, thread synchronization is used to make sure that all memory operations are completed before the next access and prevent read after write and write after read hazards. However, unnecessary memory synchronization will also slow down program execution.

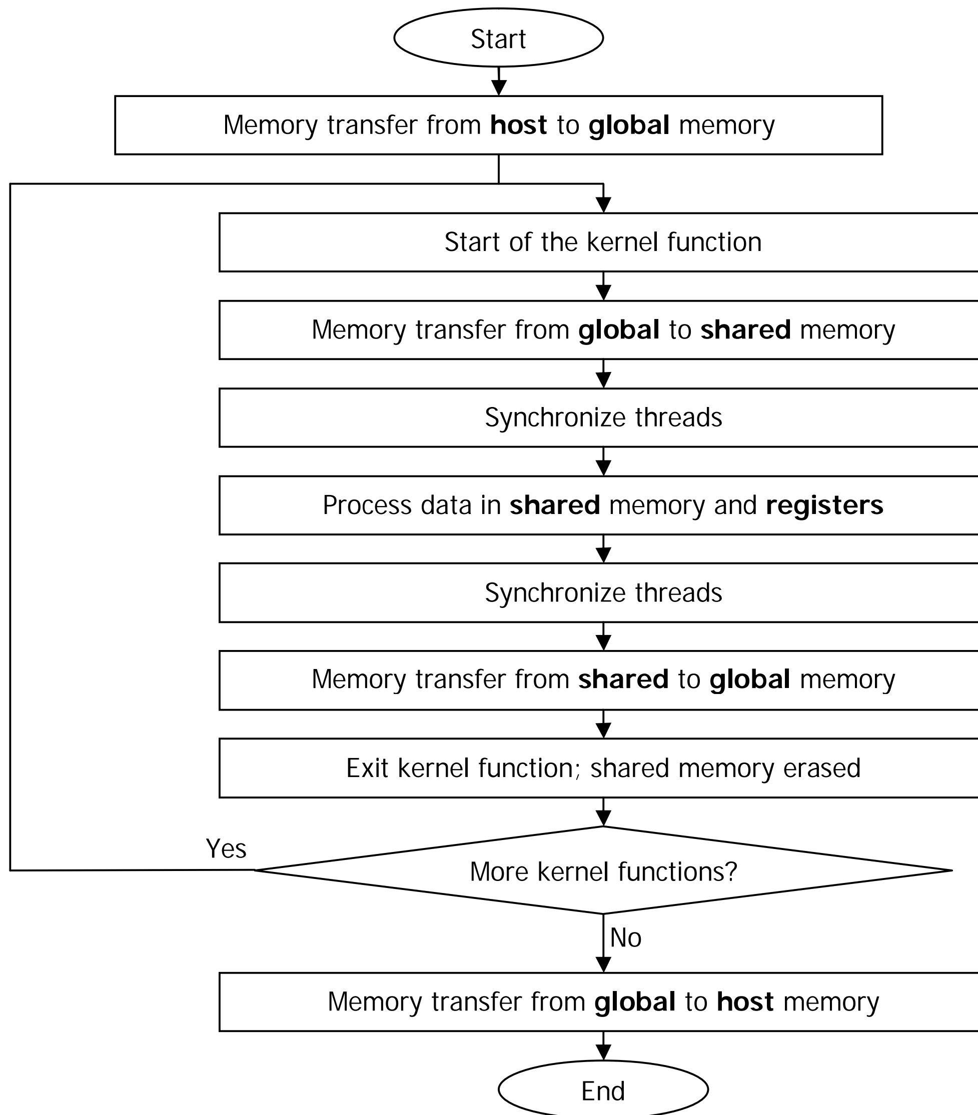

Figure 2.5 Flow chart of memory access for a typical GPU program

In order to reduce memory access time, all global (device) memory access needs to be coalesced. This means that only one memory transaction is required for loading data into all 16 threads inside a half warp. Coalesced memory access can be achieved when each thread reads and writes 32, 64, or 128 bit words that lie in the same continuous memory segment [7]. If this

requirement is not met, the memory access will become serialized and the computation speed will likely be 16 times slower. Similar situation applies to shared memory access, where the memory divides into 16 banks numbered sequentially for each 32 bytes of memory address. In order to avoid conflict and achieve high memory throughput, each of the 16 threads in a half-warp should read from or write to a different bank during a memory access [7]. Conflict-free memory access can also achieved when all threads read from the same memory location. More detail about memory access issue will be discussed in the implementation section of both Chapter 2 and 3.

2.2 Directional Image/Video Spatial Resolution Upconversion

The challenge of image/video spatial resolution upconversion is to preserve and reconstruct fine and sharp spatial details in the enlarged image/video while maintaining low computational complexity. Existing interpolation methods such as linear, cubic spline, and cubic convolution has low computational complexity, but cannot reserve edge well [11], [12]. This problem was solved by edge-guided interpolation techniques developed in the recent years [13], [14], [15], but they involve complex interpolation methods and are not suitable for real-time use.

In this chapter, we present a highly parallelized version of the algorithm in [14,15] using CUDA-based GPU computing. Our version is suitable for real-time resolution upconversion.

We first briefly explain the algorithm in [14, 15]. The image interpolation is carried out in two steps as shown in Figure 2.6. The left hand side of the figure shows the first pass, where the algorithm generates a quincunx image by interpolating the missing pixels (as marked by gray circles) with four available diagonal neighbours (black circles). The right hand side shows the second pass, where the missing pixels in the quincunx image (white and dark grey circles) are then interpolated using the horizontal and vertical neighbours.

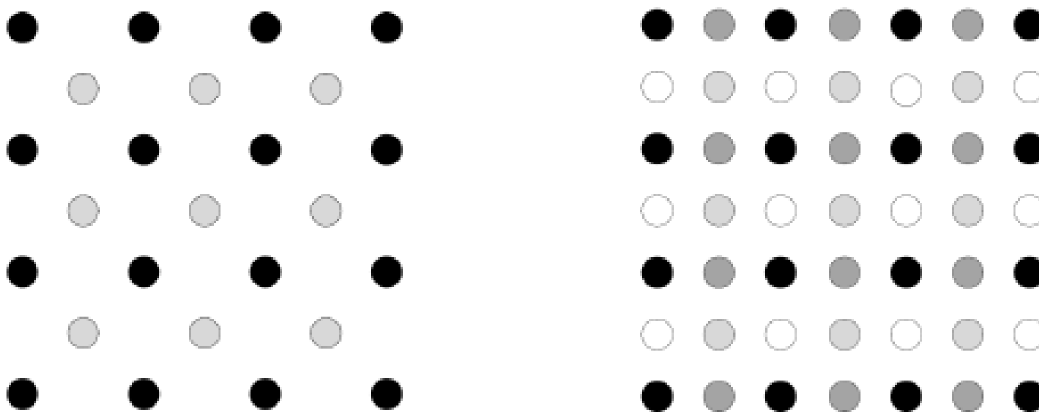

Figure 2.6 Two-pass interpolation process

Each pass of the algorithm involves a three steps process for generating the value of a missing pixel:

1. Multiple interpolation of each missing pixel using different interpolators.

2. Validity verification of each interpolator to determine its accuracy.

3. Select winning interpolator or fusing multiple estimates.

For **interpolating the missing pixels**, multiple interpolators are used to estimate an unknown pixel *Y.* For this implementation of the algorithm, cubic interpolations from two diagonals in the 45 and 135 degrees directions are used, as illustrated in Figure 2.7.

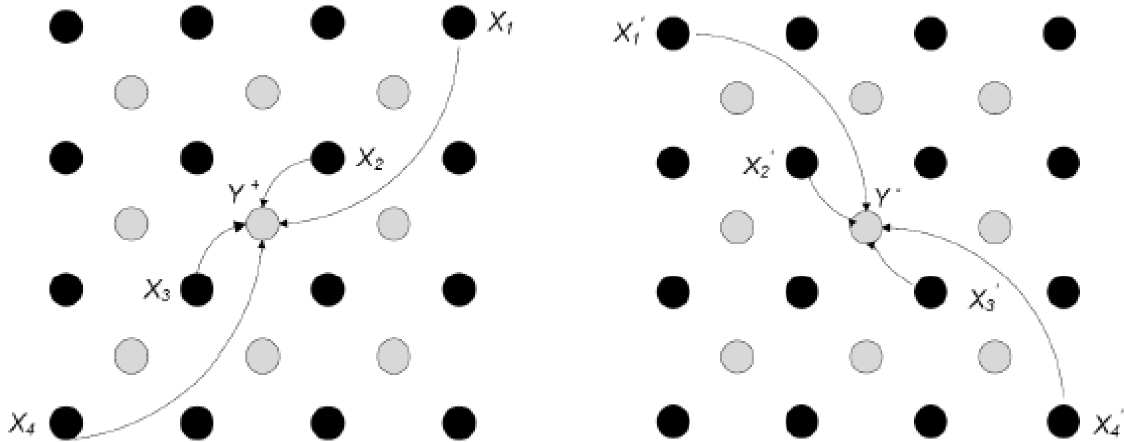

Figure 2.7 Diagonal cubic interpolation for both 35 and 135 degrees

The equation for the 45-degrees interpolation, Y^{+}, from known pixels, *X*, is

$$Y^{+} = -\frac{1}{16}X_1 + \frac{9}{16}X_2 + \frac{9}{16}X_3 - \frac{1}{16}X_4. \tag{2.1}$$

Similarly, the equation for the 135-degrees interpolation, Y^{-}, from the known pixels, *X'*, is

$$Y^{-} = -\frac{1}{16}X'_1 + \frac{9}{16}X'_2 + \frac{9}{16}X'_3 - \frac{1}{16}X'_4. \tag{2.2}$$

The coefficients of the 4-tap filter used in the interpolation are given by [8].

To **determine the accuracy of the interpolation**, the same equations are applied. However, the known pixels *X,* are now being estimated by using the interpolated pixels *Y*,

$$\hat{X}^{+} = -\frac{1}{16}Y_1^{+} + \frac{9}{16}Y_2^{+} + \frac{9}{16}Y_3^{+} - \frac{1}{16}Y_4^{+}, \tag{2.3}$$

$$\hat{X}^{-} = -\frac{1}{16}Y_1^{-} + \frac{9}{16}Y_2^{-} + \frac{9}{16}Y_3^{-} - \frac{1}{16}Y_4^{-}. \tag{2.4}$$

The process is illustrated in Figure 2.8.

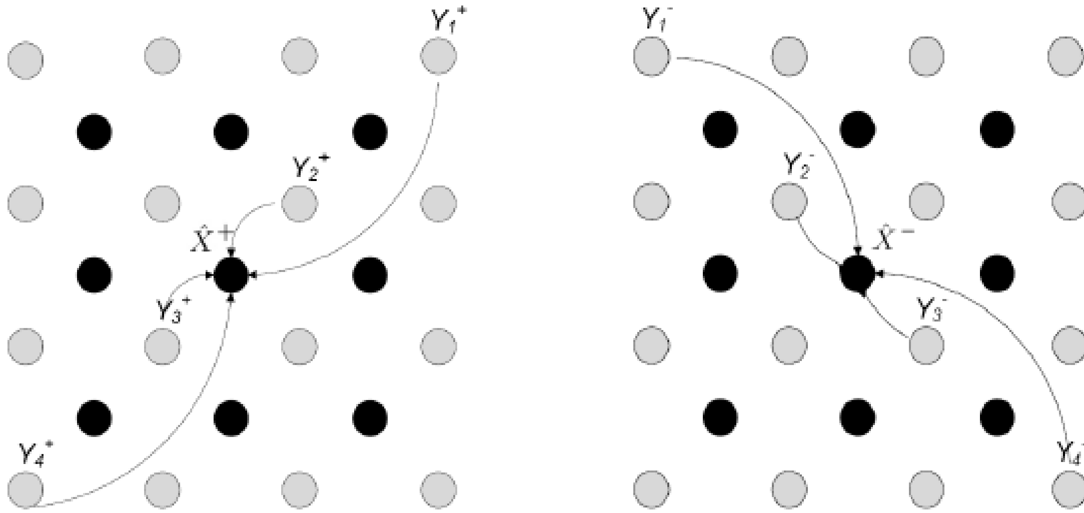

Figure 2.8 Verification for both 45 and 135 degrees interpolators

The error $e_k(i, j)$ of interpolation can then be determined by the total mean square error of nearby reconstructed pixels

$$e_k(i,\ j) = \sum_{(m,n)\in W(i,j)} \left(X_k(m,n) - \hat{X}_k(m,n)\right)^2, \quad 1 \le k \le 2, \tag{2.5}$$

where $X(m,n)$ are the original pixel values and $\hat{X}(m,n)$ are the interpolated pixel values, and *W(i,j)* is the window around (*m,n*) where the mean square error is to be evaluated, as shown in Figure 2.9.

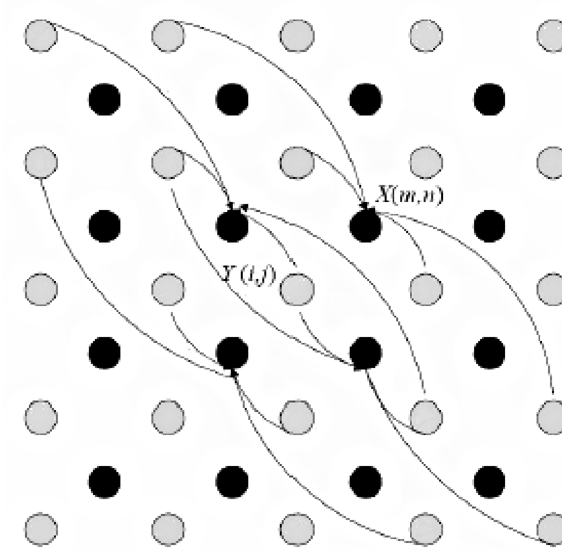

Figure 2.9 Error interpolation window W for the 135 degrees interpolator

The final step of the algorithm is to **select the best interpolator**, based on the errors $e_1(i,j)$ and $e_2(i,j)$ determined from the previous step. The final value Y of the interpolation is selected by [14],

$$Y(i,j)=\begin{cases} Y^{+}(i,j), & e_2(i,j)>e_1(i,j)+\tau, \\ Y^{-}(i,j), & e_1(i,j)>e_2(i,j)+\tau, \\ Y^{+}(i,j), & e_1(i,j)=e_2(i,j)=0, \\ Y_{fuse}(i,j), & otherwise. \end{cases} \tag{2.6}$$

where τ is a predefined threshold and Y_{fuse} is a combination of both interpolations, given by the equation

$$Y_{fuse}(i,j) = \frac{e_2(i,j)}{e_1(i,j)+e_2(i,j)} Y^{+}(i,j) + \frac{e_1(i,j)}{e_1(i,j)+e_2(i,j)} Y^{-}(i,j). \quad (2.7)$$

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 neighbours are collected in the verification step. The illustration of this process is the same as rotating Figure 2.7, Figure 2.8, and Figure 2.9 by 45 degrees.

Despite the simplicity of this algorithm, the performance is significantly better than the cubic method in term of preservation and reconstruction of image details. Furthermore, this algorithm also using local image information, which is suitable for parallel computing and implementation, which will be described in the next section.

2.3 CUDA Implementation of the Directional Interpolation Algorithm

Since GPU is massively threaded parallel processor, to achieve fastest processing speed, all calculations and interpolations are to be done on the GPU. Each GPU thread is designed to process only a single pixel to effectively use all available resources. In this way, up to 768 pixels (or 1024 pixels for the newer generations) can be processed simultaneously. Due to the dependencies of the pixel values generated by each stage of the algorithm, each kernel function only

performs a single stage of the calculation, as the value generated would be required for the next kernel function. The general flow diagram of the GPU implementation is shown in Figure 2.10.

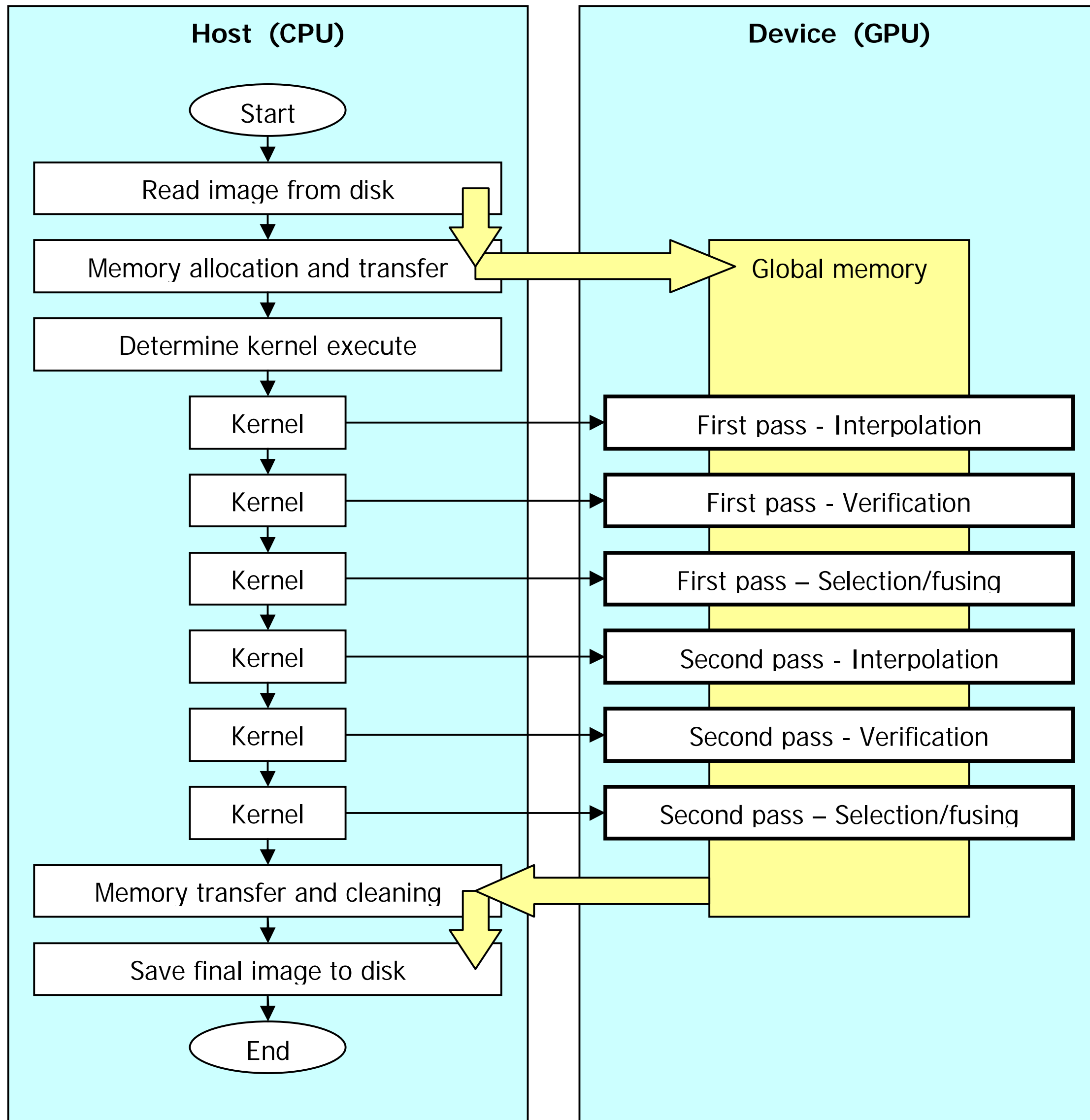

Figure 2.10 General program flow for GPU implementation of the interpolation algorithm

In the figure, bold box indicates functions to be executed on the GPU, and the yellow arrows and box indicate the location of interpolation data, including two memory transfers between host and device – from CPU to GPU and from GPU to CPU. Each of the subsection below describes the changes made to the original algorithm for GPU implementation.

2.3.1 Branch and Loop Replacement

Section 2.1.3 stated that all branches and loops should be best avoided in GPU coding to prevent divergent among the threads within the same warp. One of the possible replacements is the branch for clipping and clamping a value within a certain range, which can be done by using min and max functions. For instance, the clamping of a pixel value x within 0 and 255 becomes

$$min(max(x, 0), 255), \tag{2.8}$$

and the flipping of image pixels for boundary condition can be

$$min(max(x, -x), 2w - x - 2), \tag{2.9}$$

where w is the width of the image.

The other branch in the algorithm occurs during the selection of the interpolator. However, as the branch conditions are generated from mean square error calculation using Equations 2.6 and 2.7, it is independent on thread configuration so divergent warp cannot be avoided.

The main loops for processing each pixel are replaced indirectly by threads and blocks. The loops for accessing the known pixels of each

interpolator, on the other hand, are unrolled and the calculation would be done linearly. There is no other loop inside the kernel function.

2.3.2 Thread Configuration

Another way to avoid diverging warp is to align the thread dimensions to the size of the warp, so for a branch with condition set by the *x* and *y* location of the pixel, all threads within the warp will most likely branch in the same way. Therefore, the size of 32 is chosen for the *x* dimension of the thread block. The most logical choices for the *y* dimension of the block are 8, 12, or 16, and these choices are compared in Table 2.

Table 2.3 Comparison of different thread block dimension

Block Size	Threads	Blocks per SM (old)	Blocks per SM (new)	Relative Speed
32 x 8	256	3	4	100.0%
32 x 12	384	2	2.66	91.8%
32 x 16	512	1.5	2	95.7%

The test results show that using *y* dimension of 8 and a total of 256 threads per block yields the fastest running time by about 5~10%. Also, using this value ensures that all available threads on the multiprocessor gets utilized regardless of older (768 threads) or newer (1024 threads) GPU generations. This choice is also consistent with the goal of increasing the number of blocks for scalability on future devices mentioned in section 2.1.3 of this report as this is one of their recommended values.

2.3.3 Float Data Type

Each memory access from the GPU will load 32 bits of data; therefore, to increase efficiency and produce coalesced memory access, it is better to change the data format from unsigned `char` (8 bits) to `float` (32 bits) in order to align memory with the boundary. Using floating point operation also mean that computation may be faster as there is no longer the need for rounding and the data type is natively supported by the GPU. Moreover, the `cutLoadPGMf` function from the `cutil` library can simplify the image loading process. More discussion about improving memory efficiency can be found in the next two sections.

2.3.4 Four Sub-Images

The CPU implementation of the algorithm allocates and generates a blank high resolution output image right at the very first step. All computations are performed on this allocated memory area, as shown in Figure 2.11. Although this implementation is more memory conserving, it is not possible to obtain coalesced memory access. Figure 2.12 shows the first 10 threads of a memory access, where each thread must skip over a slot in memory; therefore, the memory access is not continuous and is uncoalesced.

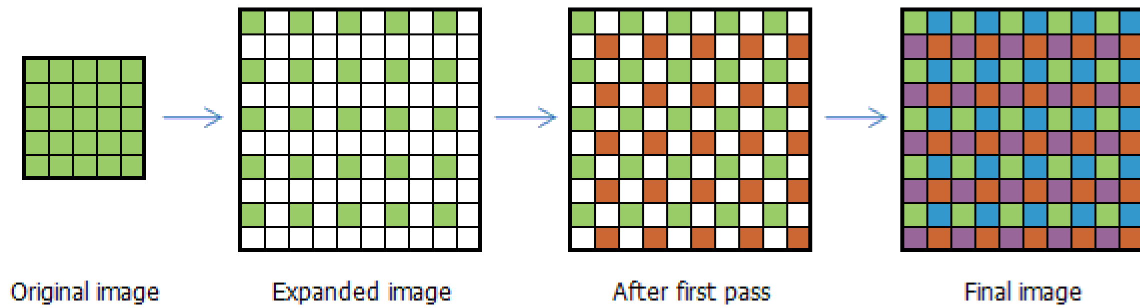

Figure 2.11 Original CPU memory allocation

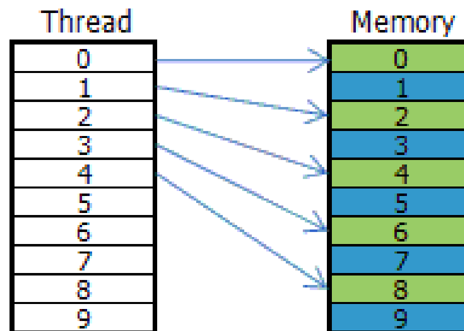

Figure 2.12 Uncoalesced memory access produced with original CPU implementation

In order to achieve coalesced memory access, a new implementation with four sub-images is used. All computations are performed on the sub-images, which are labelled from `im1` to `im4`, with `im1` representing the original low-resolution image. After the first pass, `im2` is generated from `im1`, where the two images together represent the quincunx image from the original implementation. After second pass, `im3` and `im4` are generated, and then the four images merge to form the high-resolution output. Figure 2.13 shows a single pixel split into

pixels in four different sub-images, and how they are labelled relative to each other. Figure 2.14 gives an illustration of the implementation under GPU.

Figure 2.13 Position of pixels on each sub-image relative to the original pixel

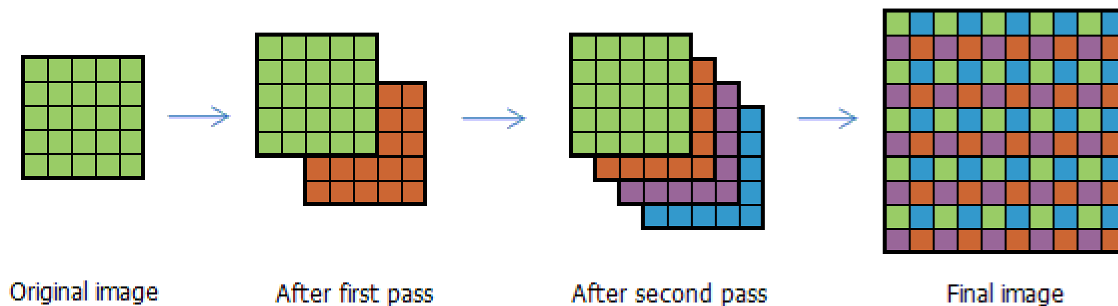

Figure 2.14 GPU memory allocation

With this new method, memory access is now continuous and coalesced for each computation performed on each of the sub-image, as shown in Figure 2.15. This method, however, does not solve all problems with uncoalesced memory access. At the last step of the process where four sub-images merge to form a high-resolution image, serialized write will still be in place because memory access on the output image is still not continuous.

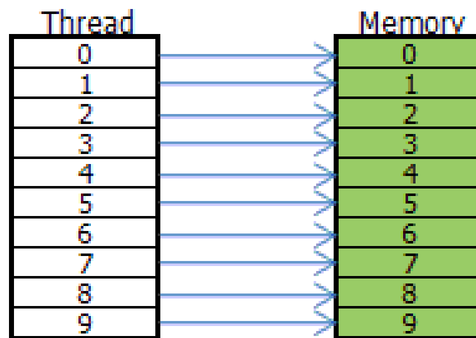

Figure 2.15 Coalesced memory access produced with new GPU implementation

2.3.5 Using Shared Memory

The analysis in Table 2.1 shows that access to global memory is slow because it is located off-chip. There is a need to cache the data into a faster memory before each computation to reduce memory access time. Both texture and shared memory are considered to be a good candidate because of their fast access speed. Since each kernel may require up to two cached images and each multiprocessor can process 768 pixels at once, the minimum space requirement is calculated to be

$$768 \text{ pixels} \times 4 \text{ bytes/pixel} \times 2 \text{ (images)} = 6 \text{ kilobytes.} \qquad (2.10)$$

Additional memory space is required for boundary pixels lying outside the cached image, which is not included in the calculation. Since there are only 6 to 8 kilobytes of texture cache per multiprocessor available [7], it might not be large enough for this application. A further complication is that texture memory is read-

only inside the kernel function, so interpolation output would still have to be stored in shared memory. As a result, texture memory is not used.

Shared memory access is less restrictive than global memory – as stated in Section 2.1.3, memory access will be coalesced as long as each thread in the warp access the same or different bank. However, to transfer the data from global to shared memory still requires sequential memory read and write. Because the algorithm require reading the value of up to 2 pixels outside of the block in both *x* (column) and *y* (row) direction for the boundary situations, one to one mapping between global and shared memory cannot be used. In order to cache all the pixels needed for processing, a caching mechanism is developed that will read and cache a few pixels beyond block boundary. Figure 2.16 shows the size of the cached data compared to block size of 32 x 8. An extra 16 pixels on each side in the *x*-direction and 2 pixels on each size in the *y*-direction are stored in the memory, resulting in a cached block size of 64 x 12.

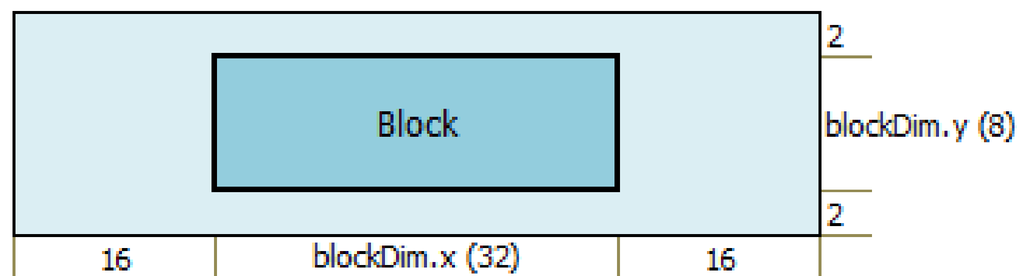

Figure 2.16 Size of cached data in shared memory compared to block size

The mechanism to cache the pixels in *x*-direction requires each thread to cache 2 pixels – 16 pixels (the size of a half-warp) to the left and right of current

position, as illustrated in Figure 2.17. Although this method caches more data than what is required, it can be done without slowing the entire process, as there is no net increase of memory access from the minimum of two required, and all memory access are coalesced.

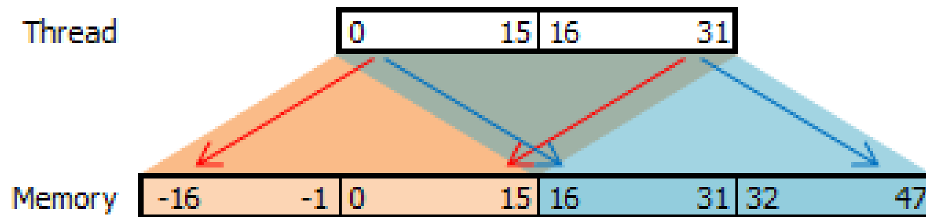

Figure 2.17 Global to shared memory transfer mechanism

Caching in *y*-direction outside of the block is done by letting the first and last row of the block to process three rows each – the row itself and the two rows above or below. This method does not negatively affect the performance, as there is no branching within the warp. After the entire caching process is done, the pixel in the shared memory can now be accessed by

$$(x, y) = (threadIdx.x + 16, threadIdx.y + 2). \qquad (2.11)$$

This caching method does not work well near the edge of the image. Because the edge of the block is also the edge of the image, the algorithm may try to access pixels outside the allocated image and cause the driver to crash. In the CPU implementation, the pixels are flipped around the edge to form a mirror. However, applying this method directly in the *x*-direction during caching will

cause bank conflict on shared memory access. This scenario is illustrated in Figure 2.18, where bank conflict (shown with red arrow) occurs when reading and writing in bank #1 and #2. Flipping the data around the edge in y-direction while loading the pixels into shared memory will not create any problem as each pixel is processed in different warp.

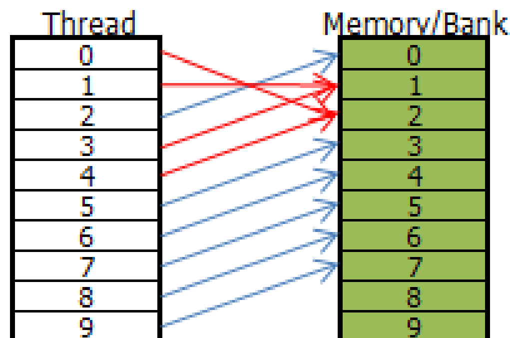

Figure 2.18 Bank conflict occurs when flipping pixels around image boundary

In order to avoid bank conflict when flipping data around the edge of the image in *x*-direction, we developed a two-pass system. The first pass loads the pixels from global memory into shared memory, while skipping all inaccessible pixels using a conditional branch. This does not add extra memory access to the caching function. The second pass is performed for edge blocks only, where it flips the 16 pixels closest to the edge around the vertical edge of the image within the shared memory. This process is shown in Figure 17, which proves that this is conflict-free because there is only one memory read in each of the 16 banks of shared memory.

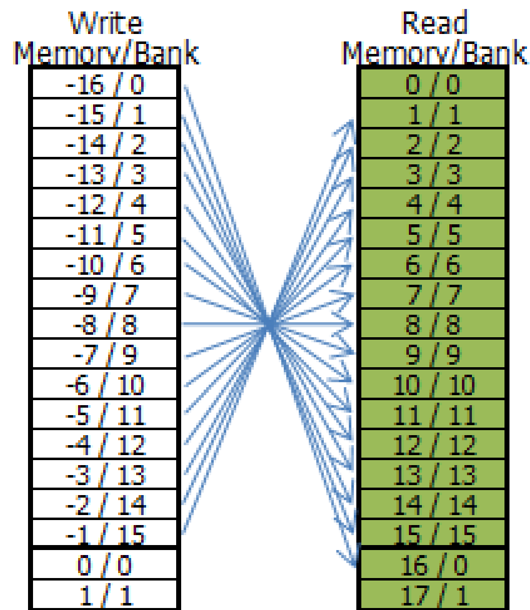

Figure 2.19 Conflict-free memory access when flipping pixels around image edge

2.4 Experimental Result

To evaluate the interpolation algorithm on CUDA, we setup a system with the following environment:

- GPU System: NVIDIA Quadro FX 1700 GPU (512MB SDRAM, 4 SMs)
- CPU System: Intel Core 2 Duo E8400 CPU at 3.0 GHz
- Operating System: Microsoft Windows XP SP3
- Compiler: Microsoft Visual Studio 2005, CUDA Toolkit and SDK v2.0
- NVIDIA Driver for Microsoft Windows XP with CUDA support (v178.24)

2.4.1 Interpolation Performance

In Figure 2.20, we compare the interpolation performance of the popular bicubic interpolation (left) in [12] and the GPU-aided directional upconversion algorithm (right). 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.

Figure 2.20 Comparison of different interpolation methods

2.4.2 Speed Performance

The execution time for processing a single image is shown in Table 2.4. This measurement is done externally using a shell command, which includes the time between the initialization of the executable to the point where the program exits and returns to the shell.

Table 2.4 Execution time measured by shell for a 256x256 Lena image

Code Version	Running Time	Improvement
Original	80 ms	-
Original compiled with CUDA	280 ms	-
CUDA optimized code	210 ms	+33%

At the first glance, it does not seem like CUDA is providing any performance improvement. However, notice that after compiling and linking the program with CUDA library, there is a 200 millisecond overhead added to the execution time. This overhead may be attributed by the initialization of the CUDA runtime dynamic link library and/or the graphic card driver, which was not required if the program was compiled with CPU-only code. Despite the fact that the final version still require longer execution time compared to the original, it can be seen that CUDA optimization results in a 33% improvement of total running time. Note that the time increase from loading CUDA is insignificant when processing video files, which the loss in speed is offset by the large gain from faster computation for each frame.

A better measurement method is to use the CUDA timer function to determine only the time spent on image processing. This ignores the program loading/initialization and image file reading/saving parts as they are more or less limited by the speed of the hard drive and operating system. In a way, this measurement is treated as if image file reading, video decoding, and/or output file saving are done on a different CPU thread. The result in Table 2.5 shows that 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 on memory allocation on the device, the speed of the GPU-aided algorithm is almost five times of the original speed.

Table 2.5 Execution time measured by CUDA for a 256x256 Lena image

Code Version	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

Table 2.6 shows the result with larger, video-sized images. Without the cost of memory allocation, the CUDA optimized code is close to 5.2 times the speed of original implementation for each test case.

Table 2.6 Execution time measured by CUDA for larger image

Video size	640 x 480		720 x 480		1280 x 960	
	Time	Speed	Time	Speed	Time	Speed
Original	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

The first case of the result shows a regular 4:3 DVD-sized video frame input. The processing time of 37 milliseconds per frame corresponds to around 27 frames per seconds, assuming the video decoding is done on a co-processor such as the CPU or another graphics card. For the widescreen DVD video in the

second case, the system can process at around 24 frames per seconds. This result shows that real-time processing of DVD video input is quite possible if the code can be slightly optimized or the hardware can be upgraded. This represents a significant improvement over the original CPU implementation, which it can barely handle even 6 frames per seconds.

Note that the graphics card used for the testing only contains 4 streaming multiprocessors supporting 768 threads, whereas the newer generation of NVIDIA display card offers up to 30 processors that can handle 1024 threads each. Significant speedup can be further obtained using these newer graphic cards.

CHAPTER 3: GPU IMPLEMENTATION OF JPEG-XR

3.1 Background of Image/Video Coding and JPEG-XR

In the field of multimedia, image and video are important mean of communication. As bandwidth and storage space are at a premium, image and video compression technology is essential for multimedia applications and consumer electronics to reduce cost and hardware requirement. The aim is to produce higher image and video quality at a lower bitrates with better bitstream flexibility; however, this usually involves in a trade-off with computational complexity as better compression ratio usually requires additional computation resources.

Figure 3.1 shows a block diagram of a typical video encoder. For image encoder, only transform, quantization, and entropy coding are required, represented within the dashed rectangular box in the figure.

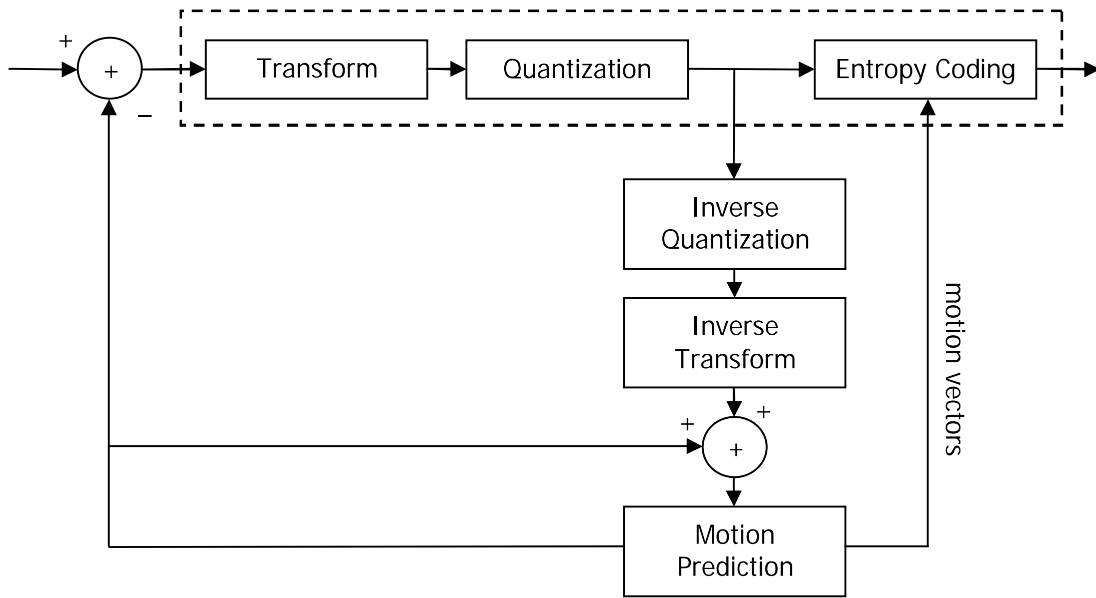

Figure 3.1 Block diagram for image and video coding

3.1.1 Overview of JPEG-XR

In the recent years, image compression technology has been developing at a rapid pace. The JPEG2000 provides better compression performance and higher code stream flexibilities than the JPEG standard. However, the JPEG2000 is high in computational complexity, whose compression speed is more than four times slower than JPEG [16]. This drawback limits its applications in many scenarios.

JPEG XR, originally proposed to the JPEG committee in 2007 by Microsoft based on its Windows Media Photo or HD Photo format, is the latest image-coding standard. It was approved in 2009 as a new international standard, ISO/IEC 29199-2. The JPEG-XR standard provides competitive

compression efficiency comparable to JPEG2000, but the implementation of both encoder and the decoder involves lower computational complexity. The basic building block of the encoder consists of [17]:

1. Pre-scaling
2. Colour conversion
3. Downsampling
4. Transform
    a. Outer Photo Overlap Transform (POT)
    b. Outer Photo Core Transform (PCT)
    c. Inner Photo Overlap Transform (POT)
    d. Inner Photo Core Transform (PCT)
5. Quantization
6. Coefficient Scanning
7. Entropy Coding
8. Interleaving

The key feature of the codec includes the use of lapped biorthogonal transform (LBT) and advanced coefficient coding [17], [18]. The LBT, consists of the four POT and PCT sub-steps, is a high-performance reversible integer transform with low complexity, which also enables lossless coding. Advanced

coefficient coding enables the codec to achieve high compression ratio and encoding flexibility. These features include [17], [18]:

- Independent coding of three levels of transform bands
- Flexible quantization
- Inter-block prediction
- Layered coding
- Adaptive coefficient scanning
- Context adaptive entropy coding with VLC table switching

Details about some of these features are discussed in the implementation section of this chapter.

3.2 CUDA Implementation of JPEG-XR Encoder

The JPEG-XR codec consists of building blocks that are similar to traditional image codec, as described in section 3.1.1. This section will describe the modifications to the Microsoft's CPU implementation of JPEG-XR encoder from [19] into a CUDA-optimized version.

In the original CPU implementation, these processes are iterated by rows of macroblock, with one row of compressed stream written to disk before loading the next row from the input stream. This method minimized the memory

requirement for compression since only two subimages with a height of 16 are present in memory at all time. However, GPU implementation requires parallel processing of the entire image. As a result, the entire image must be loaded into the memory at once, which greatly increases the memory requirement. In order for both implementations to be compatible, the image planes loaded in the GPU memory are partitioned into rows of macroblock with pointers pointing to the correct partitions, as shown in Figure 3.2. This memory arrangement ensures the seamless transfer of processed data from GPU back to CPU at later stage of the codec.

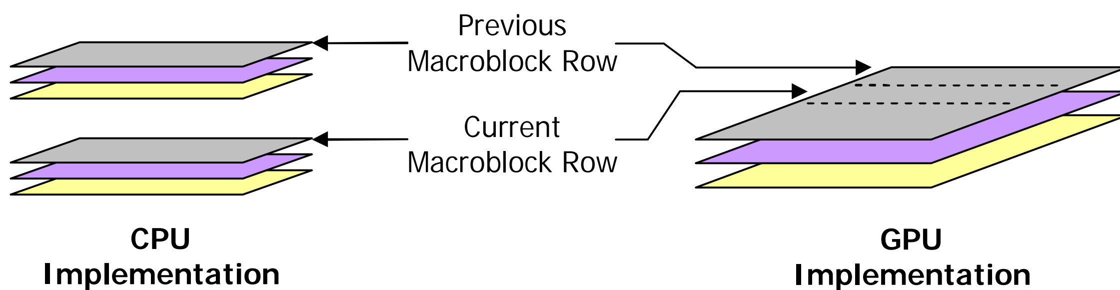

Figure 3.2 Comparison between memory arrangements of CPU and GPU implementations

The implementation of each building block of the codec is described in each section below.

3.2.1 Input and Colour Space Conversion

JPEG XR uses YUV as the internal colour format for data processing; therefore, all other external formats must be converted accordingly [20]. For this step, each thread is assigned to process a single image pixel, and blocks of

16x16 threads are used, with each representing a macroblock defined by the JPEG XR/HD Photo specification [20]. The dimension of the kernel block grid is thus the same as the grid of macroblock in the image. According to Table 2.2, this will have virtually no limit to the dimension of the input image.

For each pixel, multiple bytes of data are read from the input stream, with the size depends on the input format. The data are processed and then write back to memory as three separate image planes – one each for Y, U, and V components. The requirement of reading multiple bytes per thread poses a problem for GPU memory access, as the threads in a warp does not access continuous memory space, resulting in a serialized memory read operation. This issue is illustrated in Figure 3.3 for the RGB input format, with the memory read of the red pixel in a stride of three.

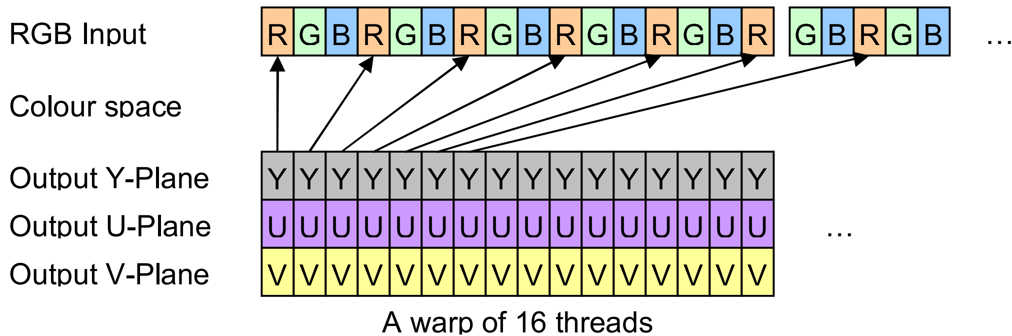

Figure 3.3 Memory access pattern of RGB input stream from global memory

One way to solve this issue is to cache the input stream into shared memory, with each thread reads and writes one byte of input data. Figure 3.4 shows that for RGB input, each colour now automatically lies in a different

memory bank, which enables parallel memory access without bank conflict. The drawback for this mechanism is that different algorithms are required for different pixel strides. For instance, RGBA input with pixel stride of four will require a padding of an empty memory spot every 16 bytes.

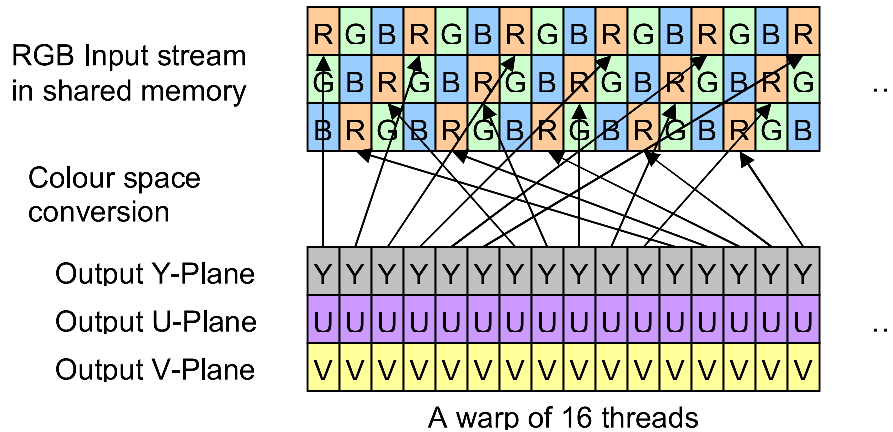

Figure 3.4 Memory access pattern of RGB input stream from after memory caching using shared memory

Since there is no need to write data back to the input stream, using texture memory is another approach for solving the memory access issue, which allows scattered memory read with relatively little cost compared to global memory. Experimental results show that the speed of loading using texture memory is approximately the same as the shared memory approach, but the implementation is much simpler and the same code can be use regardless of the pixel stride. Therefore, this approach is selected and implemented.

In order to increase the efficiency of the transform code block, a special mapping is used during encoding to rearrange the order of pixels when writing

them onto the Y, U, V image planes. In the Microsoft implementation of HD Photo codec, a mapping table is used for this purpose [19]. It is possible to load the table into constant memory and perform mapping on the GPU as well. However, since each thread will read from a different constant memory space, memory access will be serialized. This issue can be solved by converting the mapping table

$$idx = \{0, 1, 5, 4, 2, 3, 7, 6, 10, 11, 15, 14, 8, 9, 13, 12\} \tag{3.1}$$

into the equation

$$\text{idx(x)} = 8x_3 + 4x_1 + 2(x_3 \oplus x_2) + (x_1 \oplus x_0), \tag{3.2}$$

where x_3, x_2, x_1, and x_0 are the most, second, third, and least significant bits of the value x, respectively. The extra complexity of introducing the additional computation is easily offset by the faster memory access time.

After colour space conversion, pixel data cannot be written back directly to global memory with coalesced memory access, as the pixels are no longer in order. This issue is resolved by caching the data first into shared memory, as the swapping does not create any bank conflict.

This implementation of colour space conversion only requires eight registers and 4 bytes of shared memory per thread. Therefore, 100% thread occupancy can be achieved on both older and newer version of the GPUs.

3.2.2 Downsampling

Downsampling by calculating the weighted average of near-by pixels are performed in the chroma (U and V) planes only when the output format is specified to be YUV422 and YUV420. In order to maximize the performance, each thread is mapped to each output pixel, and each block consists of 16x16 threads. This means that for YUV422, where only horizontal downsampling is required, a thread block processes 1x2 macroblocks; for YUV420, where both horizontal and vertical downsampling are required, a thread block processes 2x2 macroblocks.

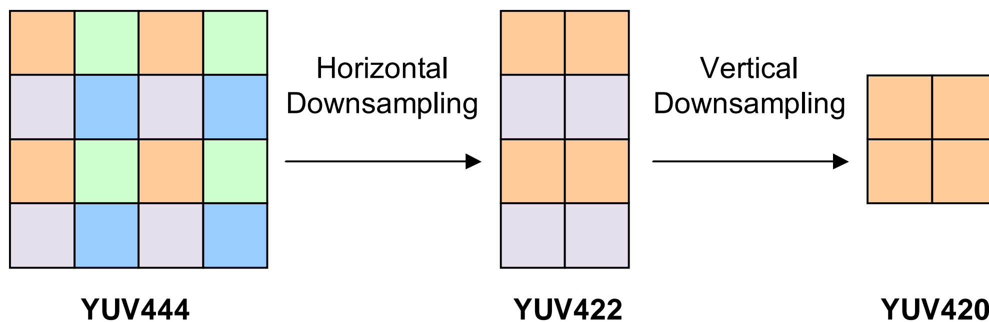

Figure 3.5 The downsampling process

Due to the mapping performed in the previous step, memory access for downsampling becomes quite complex as the pixels are out of order. It is not possible to have a coalesced memory access when reading from and writing into the global memory directly. Similar to the colour conversion block, texture memory is used for scattered read. Because the texture memory is read-only inside a kernel, the processed data cannot be written back to the original memory

position. This is not an issue for the processing of YUV422 format, as it involves only horizontal downsampling. For YUV420, additional temporary storage space and memory transfer are required between the horizontal and vertical downsampling steps.

The downsampling kernel requires a total of 19 and 20 registers for the horizontal and vertical cases respectively. This is much more than the ideal case of 10, as described in section 2. As shown in Table 3.1, using 20 registers for each thread will reduce the number of active thread per multiprocessor in older device to 256, or 1 thread block, which yields only 33% occupancy rate. It is possible to reduce the number of threads per block from 16x16 to 8x16 or 16x8 so that up to 384 threads can be executed concurrently on older devices. But in order to have the configuration of 8x16, the half-warps will be broken up, which results in a higher chance of bank conflicts and diverging branches. Similarly, the configuration of 16x8 separates a macroblock into two different thread blocks in the horizontal downsampling case, which would result in a much higher coding complexity.

Fortunately, the CUDA compiler provides a –`maxrregcount` option that allows the user to predefine the maximum number of registers to use in a kernel [21]. When the limit is reach, local memory is used in place of registers. Although local memory is slow and seems to be undesirable, sometimes it is beneficial to unload some of the register onto local memory to increase occupancy of the multiprocessor. Three different options are tested in this

project, as shown in Table 3.1. It is determined that using 16 registers and up to 4x32 bit local memory yield the fastest running time.

Table 3.1 Evaluation of the trade-offs between register and local memory use

Register Usage	Local Memory Usage	Maximum Threads for Compute Capability	
		1.0, 1.1	1.2, 1.3
20	0	256 (1 block)	768 (3 blocks)
16	4	512 (2 blocks)	1024 (4 blocks)
10	10	768 (3 blocks)	1024 (4 blocks)

3.2.3 Transform

Transform for the JPEG-XR is divided into four steps consisting of two stages of Photo Overlap Transform (POT) and Photo Core Transform (PCT). The POT involves the processing of a 4x4 region that consists of a quarter of four neighbouring blocks, while the PCT applies to the 4x4 blocks. The block boundary for both transforms is shown in Figure 3.6. The first stage of the transform processes all pixels within a macroblock, while the second stage only deals with the 16 DC coefficients in each macroblock. These together forms a two-stage lapped biorthogonal transform (LBT); however, each of the POT stages are free to be turned on or off [17], [18], [20].

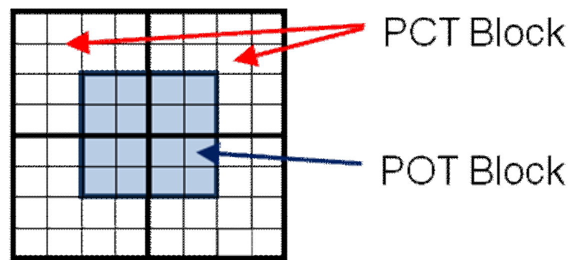

Figure 3.6 The block boundary for PCT and POT

One of the challenges to parallelize transform operation is to determine how much data is to be processed by a single GPU thread. Table 3.2 shows the various options we considered. Since the pixels are dependent upon each other during transform, assigning one pixel per thread would require memory synchronization after almost every lifting operation, which is highly inefficient. On the other hand, assigning each thread per block would make the implementation much easier, but there is not enough shared memory cache on the device, capping the maximum thread per multiprocessor to 256, or 25% occupancy. Since each transform operator involves either 2 or 4 pixels, it is possible to assign 4 pixels per thread, with each responsible for a single operator. This method allows the maximum use of GPU resource while having the most macroblocks being processed in parallel.

Table 3.2 Options to assign pixels per thread during transform operations

Pixels per Thread	Threads per Macroblock	Minimum Shared Memory per Thread	Maximum Threads per Multiprocessor	Maximum Macroblocks per Multiprocessor
1	256	4 bytes	4096 → 1024 or 768	4 or 3
4	64	16 bytes	1024 or 768	16 or 12
16	16	64 bytes	256	16

The maximum number of parallel threads can be executed on a multiprocessor is further limited by registers; therefore, the block dimension of 16x4 is chosen, as it is aligned to macroblocks of the image. When loading the pixels into shared memory, *x*-index represents one of the 16 pixels in the block while *y*-index represents a group of 4 blocks. This ensures coalesced and conflict-free memory access during caching since each block is stored in the global memory as a 1x16 strip. To compute the transformation, *x*-index of the thread block now represent each block in the macroblock, while *y*-index are used to identify each of the 4 threads assigned to each block. This would eliminate the possibility of divergent warp, because the instructions would have to be the same when processing same pixel in different block. To avoid bank conflict in both dimensions, shared memory is allocated as a 17x16 array [22].

Figure 3.7 shows the detail of this implementation. The image data for the macroblock is located within the thick bounding box of 16x16. Each row within the box represents a single image block, and each colour of the grid represents

different memory bank. This example shows the access of pixel 0, 4, 8, 12 for each of the 16 blocks.

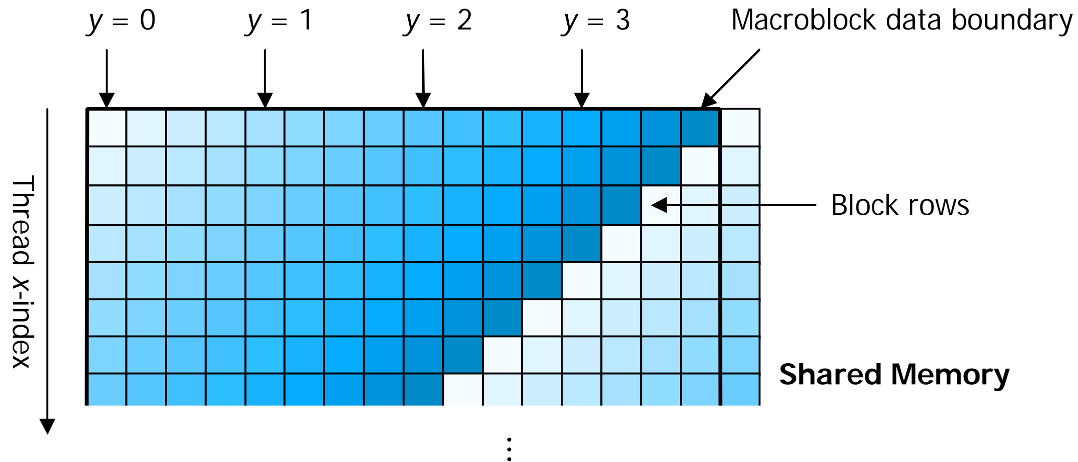

Figure 3.7 Memory configuration for transform operation

The POT consists of 4x4 pre-filter for each block of data, as well as 4-point pre-filter for the boundary pixels. To avoid branching inside a kernel, the POT operation is divided into three kernels, shown in Figure 3.8 – Kernel 1 process the interior blocks, kernel 2 processes the upper and lower boundary, and kernel 3 processes the left and right boundary. The four corners of the image are not processed [17].

POT blocks and macroblocks are offset by two pixels in both x and y direction, so the data for the POT block are scattered onto four different macroblocks. Each block would require 4 memory accesses for caching, and each memory access would cache 4 different blocks. PCT are more straightforward, as the image blocks are aligned to the thread block and there are

no boundary cases to consider. Memory reading and caching is always one-to-one.

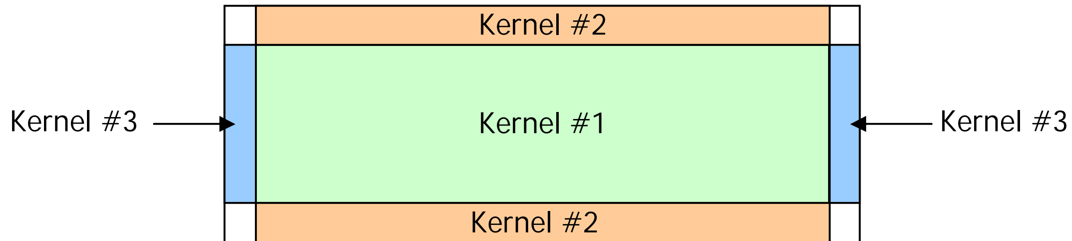

Figure 3.8 The three kernels for first stage POT on a rectangular image

After first stage PCT, a sub-image containing only the DC coefficients is created and is sent back to the POT and PCT kernels for the second stage transform. This method ensures a continuous global memory access instead of having a stride of 4 from selecting only the DC coefficients.

After second stage PCT, each DC coefficient from the sub-image replaces the original value on the full images to complete the transform process.

3.2.4 Quantization

In JPEG XR, the quantization is split into DC (first coefficient of a macroblock), low-pass (first coefficient of other blocks in a macroblock), and high-pass (all other coefficients). Porting the quantization to CUDA is much easier than previous sections as each pixel is independent of each other.

Since the computation can be performed regardless of the state of other pixels, quantization is processed directly after the last step of transform, before

the data being written back to global memory. Thus for high-pass coefficient, quantization is performed after the first stage PCT; for DC and low-pass, it is preformed after the second stage PCT, before the coefficient being merged back into the full image. This way, the number of global memory access can be reduced by two.

3.2.5 DCAC Prediction, Coefficient Scanning, and Entropy Coding

The last three building blocks of the JPEG-XR encoder are challenging for CUDA implementation because the process is adaptive and dependent upon previous results. For the DC and low-pass prediction, the process is dependent on the prediction from previous macroblock; therefore, the process has to be serialized. The high-pass prediction, on the other hand, can be parallelized to some extents, but it would result in only 12 active threads per macroblock. Due to memory constraint, most of the available computing resources on the GPU would be wasted.

Coefficient scanning process in JPEG XR is adaptive, where the order of the scan can change in the course of the tile [17]. If the scanning is done in parallel, it is impossible to predict the location of the order change before the scan, thus the scanning would most likely to be done incorrectly. There are also not enough tiles in a regular-sized to be scanned efficiently if tile is processed by a thread.

Similar problems apply to the entropy coding, where the process is adaptive in the course of tile and there is not enough tile to justify parallel

processing. The entropy coding also deals heavily with disk I/O, which can only be done in CPU. These challenges make CUDA implementation undesirable.

3.3 Experimental Result

In this section, the execution time of our GPU implementation of the JPEG XR encoder is compared to the CPU version using the Microsoft HD Photo source code [19]. The test environment contains:

- GPU System: NVIDIA Quadro FX 1700 GPU (512MB SDRAM, 4 SMs)
- CPU System: Intel Core 2 Duo T8300 CPU at 3.0 GHz
- Operating System: Microsoft Windows Vista SP2
- Compiler: Microsoft Visual Studio 2005
- CUDA Toolkit and SDK v2.3
- NVIDIA Driver for Microsoft Windows XP with CUDA support (v178.24)

3.3.1 Speed Performance for Each Code Segment

Table 3.3 compares the execution time of each GPU code block and/or kernel function with the original CPU implementation. The results for all blocks are taken from a lossless compression test with input image size of 1024x768. An exception is for downsampling, which uses another test case that produce a YUV422 output image.

Table 3.3 Comparison between CPU and GPU processing time for various blocks for the JPEG XR encoder

Blocks	CPU Processing Time	GPU Processing Time	Relative Speed
GPU Memory Allocation	-	2.77 ms	-
Memory Transfer to GPU	-	1.85 ms	-
Color Space Conversion	6.71 ms	3.05 ms	220%
Downsampling	7.21 ms	5.19 ms	139%
First Stage Photo Overlap Transform	19.39 ms	13.73 ms	141%
First Stage Photo Core Transform	7.97 ms	3.70 ms	215%
Second Stage Transforms	2.15 ms	1.02 ms	211%
Quantization	9.63 ms	0.97 ms	993%
Memory Transfer to CPU	-	11.85 ms	-

The results show that in all cases, the GPU implementation yields better performance, with an average relative speed of about 215%. For the kernel that involves complicated memory access and mapping – downsampling and first stage POT – the speed up is less significant. On the other hand, the segment that does not require any memory access, namely quantization, can achieve a speed up of almost 9 times. This observation shows that the main bottleneck of the GPU implementation is memory bandwidth. If quantization were to be implemented as a separate kernel, an additional 3 milliseconds is required for memory access, so the relative speed would fall back to the 215% range.

The use of texture memory instead of global memory speeds up the color space conversion kernel by approximately 10 milliseconds. The use of

computational equation rather than constant mapping table further increase the speed by 4 milliseconds, allowing it to surpass the speed of CPU implementation. Using texture memory for downsampling also increases the speed by 5 milliseconds, whereas the cost of providing a temporary space for data storage between the horizontal and vertical downsampling is less than 1 millisecond.

3.3.2 Speed Performance for Different Output Types

Table 3.4 and Table 3.5 compare the processing time for different output formats. The results show that GPU kernels are close to twice as fast as the CPU counterpart (up to quantization only). However, with the addition of entropy coding and memory allocation/transfer time, there is only a 2~7% benefit.

Note that for the lossless, YUV, and grayscale cases, a 1024x768 input image is tested. For the alpha cases, an 800x800 image is used instead.

Table 3.4 Comparison between CPU and GPU processing time for various types of images – to quantization only

Output Format	CPU Processing Time	GPU Processing Time	Relative Speed
Lossless	41.53 ms	22.54 ms	184%
YUV422	38.45 ms	21.55 ms	178%
YUV420	36.43 ms	21.94 ms	166%
Grayscale	19.76 ms	9.55 ms	207%
Planar Alpha	48.62 ms	26.36 ms	184%
Interleave Alpha	48.71 ms	26.26 ms	185%

Table 3.5 Comparison between CPU and GPU processing time for various types of images – full encoder

Output Format	CPU Processing Time	GPU Processing Time	GPU Memory Transfer	Relative Speed
Lossless	118.60 ms	99.58 ms	16.34 ms	102%
YUV422	101.41 ms	84.52 ms	13.92 ms	103%
YUV420	84.66 ms	70.15 ms	11.69 ms	103%
Grayscale	49.89 ms	39.67 ms	7.03 ms	107%
Planar Alpha	92.18 ms	69.88 ms	20.61 ms	102%
Interleave Alpha	95.51 ms	73.06 ms	17.04 ms	106%

3.3.3 Speed Performance for Different Image Sizes

Table 3.6 and Table 3.7 show the processing time of lossless compression on various sized input images. It indicates that the GPU implementation is more beneficial for larger image than smaller ones. In fact, for small images, it is faster to run the entire encoder on CPU and ignores the GPU completely. Even for very large images, there is only a modest 11% speed up if memory allocation time were to be included.

As mentioned above, the bottleneck for the GPU implementation of the JPEG XR codec is memory bandwidth. It seems that any time saved from computation would be lost to memory transfers. But one may not take this observation for granted as both CPU and GPU hardware used for this test are not the latest offers from Intel and NVIDIA, and they are both 2 to 3 years old. For instance, the newest graphics card offers 5 to 10 times the memory bandwidth and more than 5 times the amount of processors compared to the

Quadro FX 1700 used in this test [23]. This would provide tremendous help in speeding up the codec.

Table 3.6 Comparison between CPU and GPU processing time for various sizes of images – to quantization only

Output Size	CPU Processing Time	GPU Processing Time	Relative Speed
256 x 256	3.53 ms	2.66 ms	133%
512 x 512	13.86 ms	8.07 ms	172%
1024 x 1024	56.41 ms	31.87 ms	177%
2048 x 2048	223.04 ms	115.62 ms	193%
4096 x 4096	917.47 ms	462.92 ms	198%

Table 3.7 Comparison between CPU and GPU processing time for various sizes of images – full encoder

Output Size	CPU Processing Time	GPU Processing Time	GPU Memory Transfer	Relative Speed
256 x 256	7.37 ms	6.50 ms	1.85 ms	88%
512 x 512	27.50 ms	21.71 ms	6.56 ms	97%
1024 x 1024	109.96 ms	85.42 ms	20.60 ms	104%
2048 x 2048	418.60 ms	311.18 ms	76.60 ms	108%
4096 x 4096	1631.03 ms	1176.48 ms	294.73 ms	111%

CHAPTER 4: CELLPHONE AIDED MULTIMEDIA PROCESSING

4.1 Introduction

As technology improves, new ideas are constantly developed to improve user experience and interaction. This chapter describes a system that used cellphone as an input device, with signal sent wirelessly to the computer using Bluetooth technology. The phone allows user to send commands remotely without the need to be physically beside their computer. This section represents the right-hand-side of the system illustrated in Figure 1.1.

Remote input device is not a new idea, however. Beside the Nintendo Wii system described in Section 1.1, numerous other systems such as [24] and [25] use cellphone as an input device. Most of these systems use raw data from the cell phone, where the user can position mouse pointer based on the movement, position, and orientation of the phone. Mouse clicks and key presses can be emulated using buttons on the cell phone. As a result, the cellphone now acts as an input device that replaces mouse, keyboard, and even joystick on a computer. It allows the user to obtain full control of the system.

In contrast, our system is command-based, which can reduce the amount of data transmitted between the cell phone and the computer. The user performs

a specific movement using the cellphone, and the movement is matched in the phone against a predefined command list to identify the action and execute the corresponding command. The command list includes, but not limited to:

- Play and pause toggle
- Stop
- Mute and unmute toggle
- Pan or rotate in up, down, left, or right directions
- Snap to the next view in top, down, left, or right directions
- Zoom in or zoom out

The functions of these commands are slightly different and are defined separately by the applications. This will be described in detail in the implementation in Section 4.4.

4.2 Cellphone as Input Device

For the system, we use Nokia's N-series cellphone as an input device, which runs on Symbian S60 platform. The current implementation supports the third edition, feature pack 2 (Symbian OS v9.3) version. There are currently two different types of inputs supported in our system:

- Key press – by capturing the code of the input key

- Phone movement

The phone movement can be further divided into rotation, shaking, and other more complex motion matching. The phone motion is captured by the built-in accelerometer in Nokia cellphones using its Sensor API and R&D Plugin [26]. The sensor provides readings in all *x*-, *y*-, and *z*-axis, in the way illustrated in Figure 4.1. The rectangular box in the figure represents the cellphone with front facing upward and top going into the page [27].

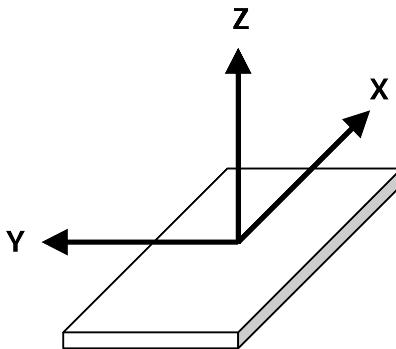

Figure 4.1 The x, y, and z axis of the accelerometer relative to the cellphone

When the cellphone is resting on the table, reading from the accelerometer is approximately (0, 0, -350). This value is not zero due to the normal force acting on the z-direction. This turns out to be a nice property that makes motion tracking much easier, which will be discussed in detail in the following sections.

The accelerometer in the Nokia phone is quite sensitive. Even when the phone lying still on a table, the reading can still fluctuate up to 10 values, as shown in Figure 4.2 with 300 samples of *x* and *y* values. The noise is

undesirable for motion detection as it may cause some algorithms to fail. To rectify this issue, a smoothing filter is used on the data. Instead using the values directly, up to 16 samples are saved, and the output now becomes the rolling average of the last 16 samples. Figure 4.3 shows the accelerometer values after filtering and smoothing, where the fluctuations reduced dramatically and is now limited to 3 or 4 values.

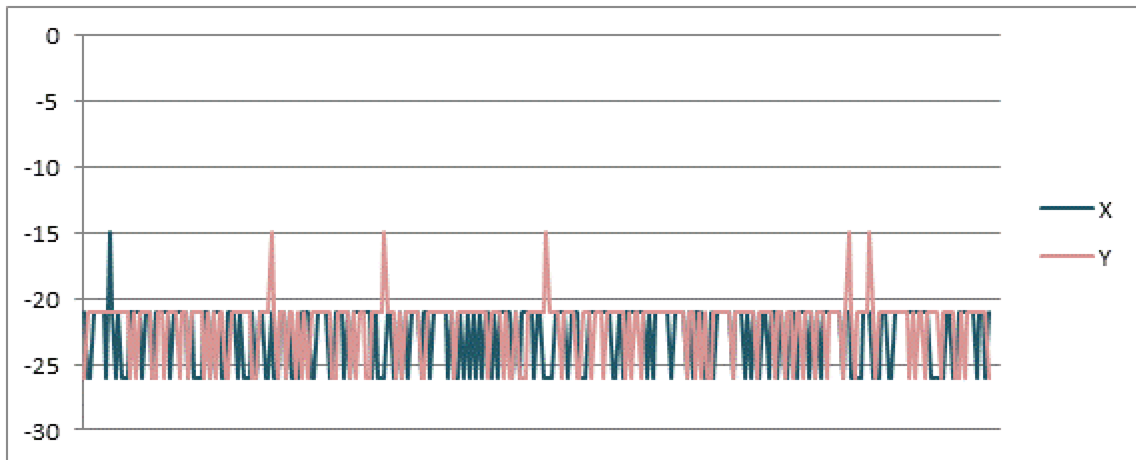

Figure 4.2 Fluctuations in accelerometer value when the phone is resting still

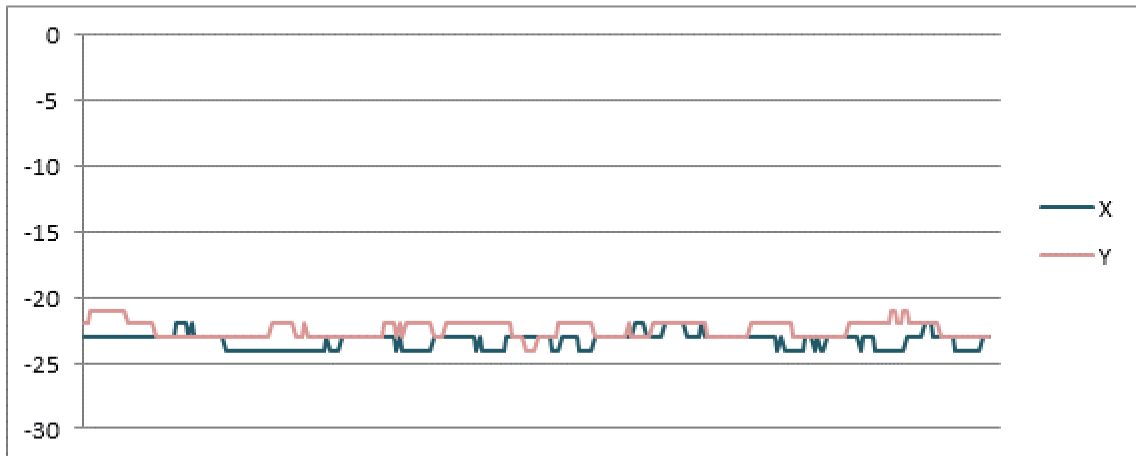

Figure 4.3 Accelerometer values after filtering to reduce fluctuations

Table 4.1 shows the default detection configuration for each of the command listed in Section 4.1. The user is free to modify this configuration to use any other detection algorithms for any of the command using the PC server application. Details about these detection algorithms are described in upcoming sub-sections.

Table 4.1 Default key configuration for each command

Command	Type	Value
Play/Pause Toggle	Key press	Right key
Stop	Key press	Left key
Mute/Unmute Toggle	Not defined	–
Pan/Turn Up	Rotate	Up movement
Pan/Turn Down	Rotate	Down movement
Pan/Turn Left	Rotate	Left movement
Pan/Turn Right	Rotate	Right movement
Snap Upward	Rotate + Key	Up movement
Snap Downward	Rotate + Key	Down movement
Snap to the Left	Rotate + Key	Left movement
Snap to the Right	Rotate + Key	Right movement
Zoom In	Key press	Up key
Zoom Out	Key press	Down key

4.2.1 Rotations

Rotation movement is the easiest action to perform and understand by the users. Therefore, it is the most commonly used action in our system and are the defaults for all directional commands, as shown in Table 4.1. A reliable detection algorithm is essential for this kind of movement.

Figure 4.4 shows the change in the acceleration data values when the phone is rotated to the right. The graph shows that the values in *y*-direction decreased and the values in *z*-direction increase dramatically, while the values for *x* remain relatively unchanged. Rotating the phone to the left produces similar result, where the values for *y* and *z* both increase while the values for *x* stay unchanged.

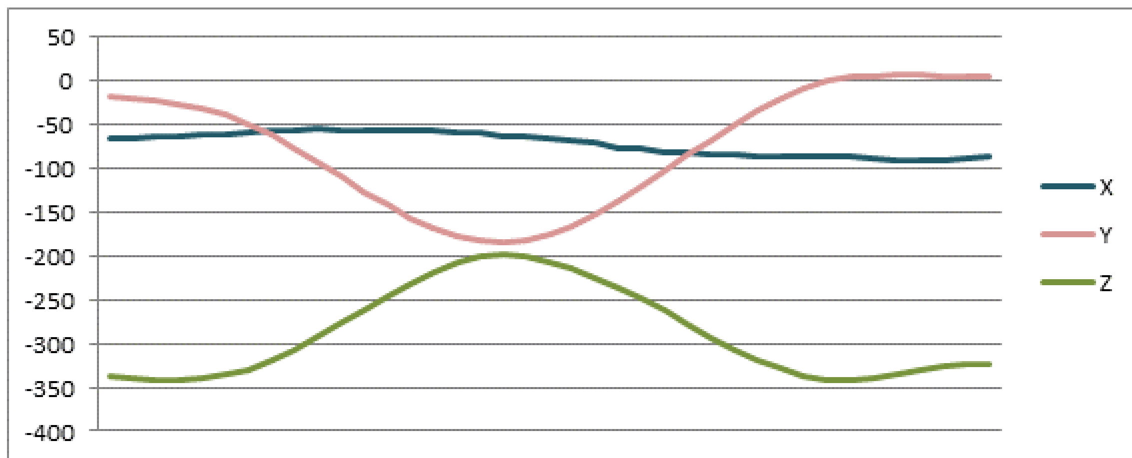

Figure 4.4 Accelerometer data for a rotate right motion

This observation can be explained by the normal force countering the force of gravity. As described in Section 4.2, the phone experienced acceleration

in *z*-direction while lying in a level position due to gravity. When the phone is tilted at an angle, the acceleration is shared between both *y* and *z* directions, as shown in Figure 4.5 (right). Therefore, acceleration in *z*-direction decreases while a change is experienced in *y*-direction.

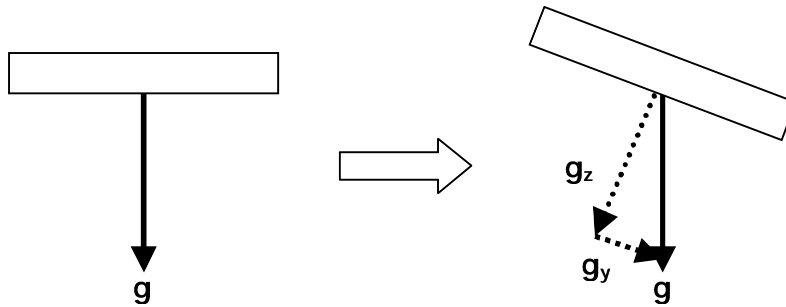

Figure 4.5 Change in gravitational acceleration when the phone is tilted

Similarly, when the phone is rotated upward or downward, the normal force is now shared between *x* and *z*. Figure 4.6 shows the accelerometer value when the phone is rotated upward, where the values in *x* decrease, *z* increase, and *y* remain relatively unchanged. When the phone is rotated downward, values for both *x* and *z* increased.

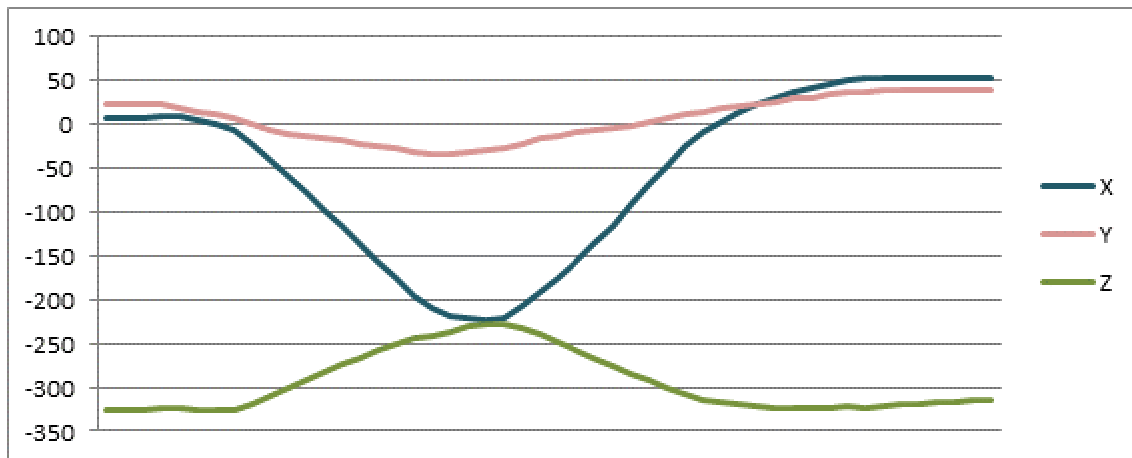

Figure 4.6 Accelerometer data for a rotate up motion

From the observations mentioned above, the accelerometer reading for rotational motion can be described with the following properties:

- The values are almost the same when the phone is in a level position
- The amount of change in value is proportional to the angle of the tilt
- The change in values for *x* and *y* is different for each direction of tilt

From these properties, it can be concluded that a threshold can be used to accurately identify the motion. When the user tilts the phone past a specific angle that is large enough for the accelerometer reading to pass the threshold, a rotation action is triggered. The algorithm ignores a specific amount of samples after the triggered action; however, if the user does not return the phone back into a level position during this time and still tilting the phone at an angle, another action will be triggered.

Figure 4.7 shows an example of the detection of rotational motion. The example represents a tilting of the phone to the right for an extended amount of time. For this action, the threshold is set to $y < -110$, represented with the dashed line labelled “T” in the figure. Once the user tilt the phone past the threshold, a “rotate right” action is triggered, and some samples after it are ignored. The user, however, does not return the phone back into a level position before the end of the ignored frame. As a result, two more “rotate right” actions are triggered.

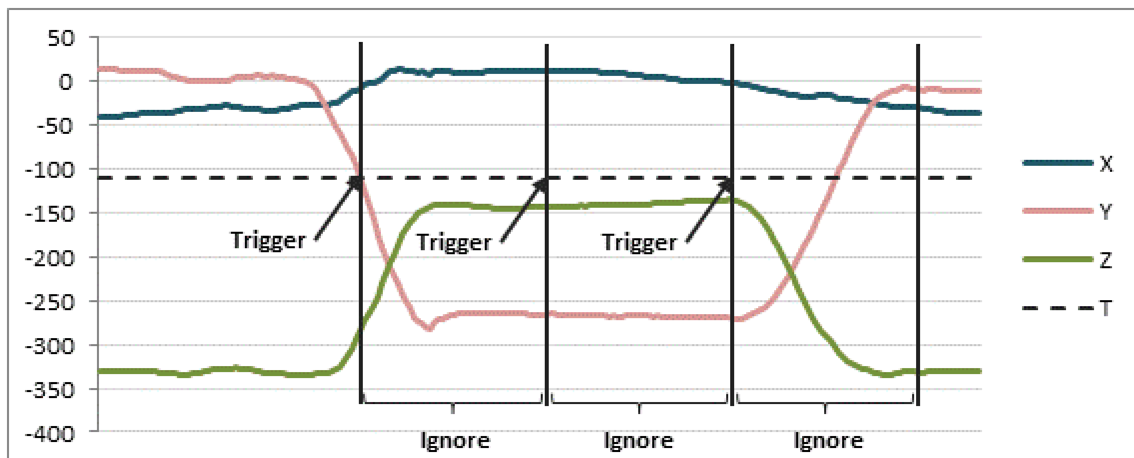

Figure 4.7 An example for the detection of rotational motion

Table 4.2 shows the default threshold value for each of the rotational motion. These values are not fixed, as the user is free to adjust any of them with the PC server application. The difference in the value for up and down motions is due to the assumption that users would normally hold the phone at a slight angle upward. Therefore, upward motion requires a larger tilt angle compared to the downward motion.

Table 4.2 Default threshold values used for rotational motions

Direction	X	Y	Z
Up	< -180	Don’t care	Don’t care
Down	> 25	Don’t care	Don’t care
Left	Don’t care	> 110	Don’t care
Right	Don’t care	< -110	Don’t care

4.2.2 Shaking Movements

Another type of simple motion that can be easily performed by the user is shaking. Figure 4.8 shows an example of the accelerometer reading when shaking the phone in all directions. It shows the values are oscillating heavily in the direction of the motion.

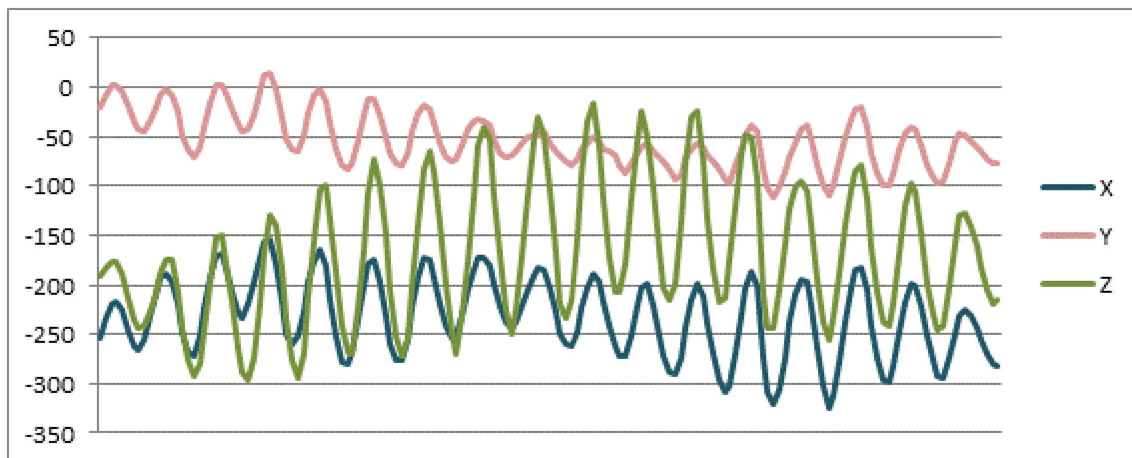

Figure 4.8 The accelerometer reading for a typical shaking motion

A simple way to identify this type of motion is to set a threshold on the range of the reading, using the equation

$$\max(y) - \min(y) > \text{threshold}, \tag{4.1}$$

where y is an array of values over a fixed amount of samples (the default value is set to 50), whose range would be higher for harder motions. Equation 4.1 detects the shaking motion in the y directions, or left and right movements. To detect up/down shaking motion, calculation is performed in the z-direction. Similarly, x-direction is used for shaking motion toward and away from the user, as illustrated in Figure 4.1.

In order to have proper detection of this action, it should be performed before or independent of the rotational movement detection described in Section 4.2.1. Moreover, detections of harder shaking motions should be performed before the lighter ones.

4.2.3 Motion Matching

Although the rotation and shaking motion detection algorithms described in the previous section are very reliable, they only cover a very limited amount of motions that user can perform with their phone. Figure 4.9, Figure 4.10, and Figure 4.11 shows the accelerometer readings for three motions that cannot be accurately characterized by the previous detection algorithms. A different algorithm is required for these types of motions.

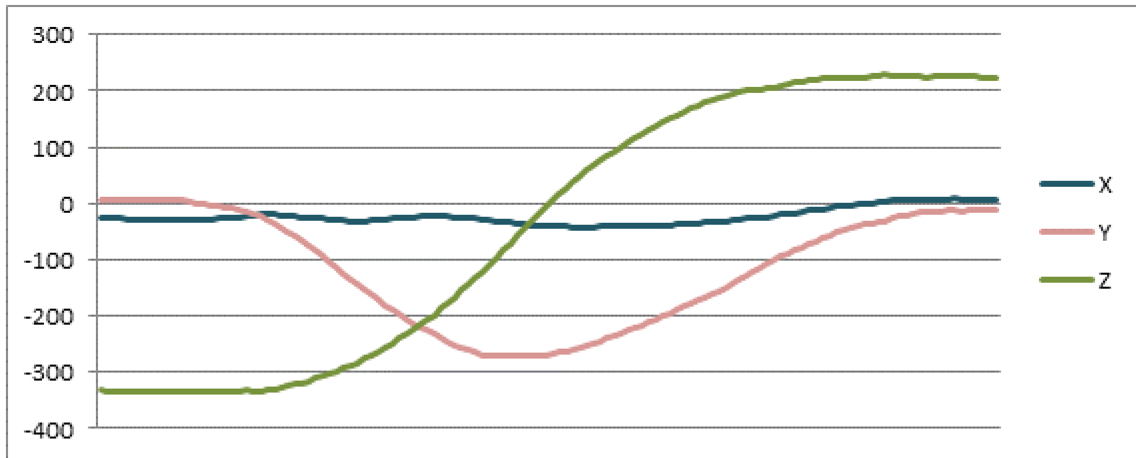

Figure 4.9 The accelerometer reading when flipping the phone upside down

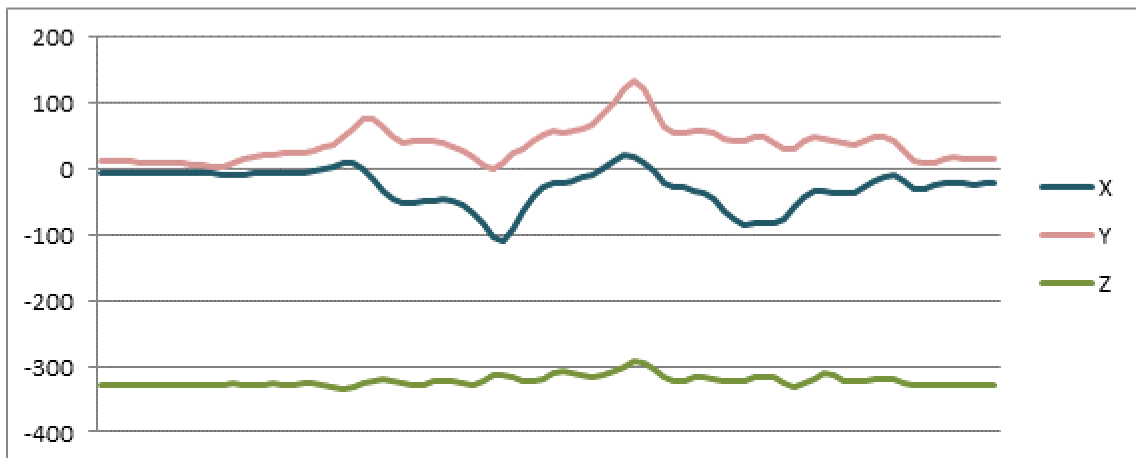

Figure 4.10 The accelerometer reading when rotating the phone twice horizontally

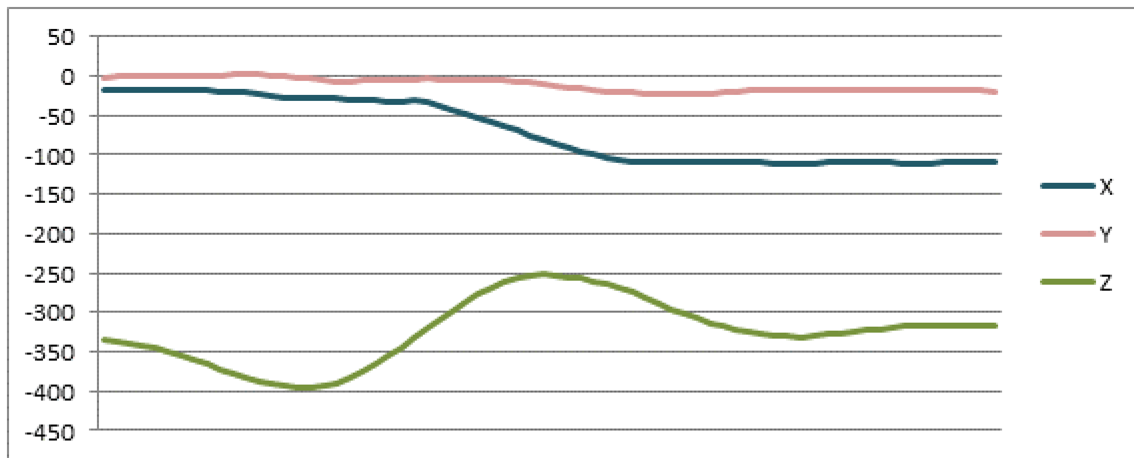

Figure 4.11 The accelerometer reading when raising the phone upward

A simple algorithm for the detection of these motions is to pre-record the readings for the motions and compare them, in real-time, with live data obtained from the accelerometer. For instance, input values from Figure 4.9 would be matched against the pre-recorded data shown in Figure 4.12.

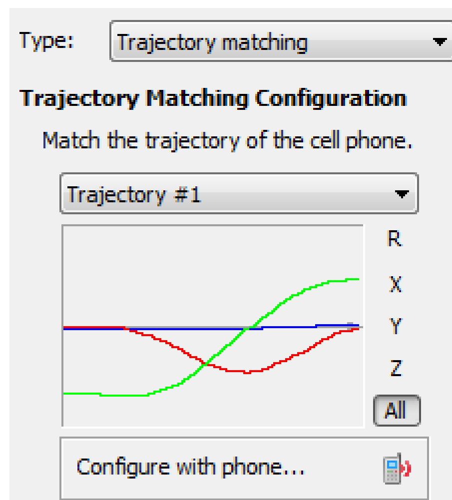

Figure 4.12 A pre-recorded data for the flipping phone upside down motion

The comparison is done by computing the mean square error (MSE) between the two values. We use MSE instead of some simpler mechanisms such as absolute error or MAD because it has higher penalty for values with larger errors, which is essential in this case to obtain higher accuracy. MSE is also simple enough to enable the comparison to be done in real-time. The algorithm uses only the derivation from the initial value, that is,

$$X'_k = X_k - X_1, \quad 2 \le k \le K, \tag{4.2}$$

where X_k are the accelerometer readings in the *x*-direction, *X'* represents the adjusted values, and *K* represent the numbers of samples to be compared. The number of sample used in the matching process depends on the recorded motion, with the maximum value of 50 and minimum value of 2. The adjusted values for the *y* and *z* readings are computed similarly. The final equations for computing the MSE becomes

$$\frac{1}{K-1}\sum_k \; (X'_k - \hat{X}'_k)^2 + (Y'_k - \hat{Y}_k)^2 + (Z'_k - \hat{Z}'_k)^2 < \text{Threshold}, \quad 2 \le k \le K. \tag{4.3}$$

In order to simplify the calculation, the mean is not computed; instead, the threshold value increases with the amount of sample. The threshold is set to be 1000 per axis per sample, that is,

$$\text{Threshold} = 1000 \times 3 \times K. \tag{4.4}$$

For a typical motion with 50 samples, this gives a threshold of 150,000.

4.3 Communication with PC

The second component is to establish a link between the cell phone and the application that displays the multimedia content. For our system, we choose to link the PC with phone using Bluetooth communication, and display the content as a web-based application. However, since none of HTML, JavaScript, Flash, and Silverlight has built-in Bluetooth support, we need to establish the link on our own. This is done by writing a stand-alone Windows Bluetooth server application, which effectively breaks this communication link into two independent parts:

- Link between the remote device and Windows server application via Bluetooth protocol.
- Link between the windows server application, web browser and the web-based application via Windows or browser APIs, JavaScript, etc.

Figure 4.13 shows a diagram of the links between the different parts of the system. The Bluetooth link between the phone and server application will be described in detail in Section 4.3.1, while the other parts will be discussed in Section 4.3.2.

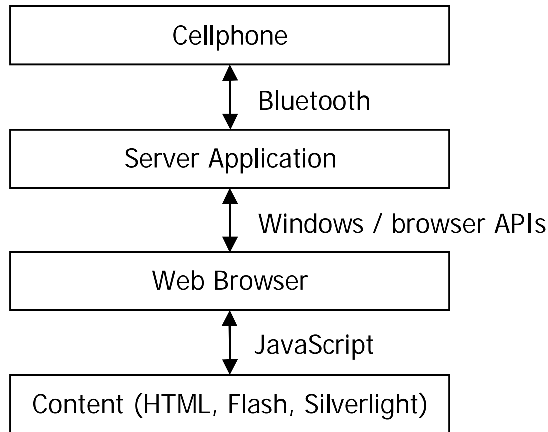

Figure 4.13 Links between various parts of the cellphone-to-web content communication

4.3.1 Bluetooth Communication between Phone and PC Server

Bluetooth is a short-ranged wireless communication technology that is intended to replace cable connection between devices. It is capable of transmitting data within the range of 10 metres and up to a data rate of 1~3 Mbps [28]. The network relationship is service based, where a service discovery protocol (SDP) is used. Using the protocol, the client device broadcasts service requests to all Bluetooth server devices in range [29]. The connection is established if any matching service, identify by a unique service ID, found in one or more server [29].

For our application, the PC acts as a Bluetooth server while the cellphone acts as a client that tries to establish the communication. As per Bluetooth protocol, it first discovers all available devices, checks for available services, and then tries to connect to the first service found. This link uses socket

implementation of Bluetooth, which utilized the Winsock library on the Windows server and Symbian Bluetooth API on the phone client [30].

Once the link is established, the cellphone and PC communicate with 10-bytes data packets using the Bluetooth RFCOMM protocol. The content of the packet is shown in Table 4.3. This is a one-way communication, as the PC server does not send any packet back to the cellphone.

Table 4.3 The content of the 10-bytes packets used for the Bluetooth link

Byte	Item
1	Packet ID
2	Cellphone ID
3	Command ID
4 ~ 5	Value 1
6 ~ 7	Value 2
8 ~ 9	Value 3
10	Error checking code

The **Packet ID** is a value between 0 and 255 that increments with every packet sent by the phone (and reset back to zero once 255 is reached). This identifies each packet and check if there is any packet loss during the communication.

Cellphone ID is a one-byte value that identifies the type of cellphone used as the remote device. It is essential for the PC server to know which cellphone it tries to communicate with since the accelerometer readings and key code might be different across platforms, which requires different values in the detection

algorithm. The current implementation consist of only one type of platform – Symbian S60 3rd FP1, which is identified by the byte 0x01.

Command ID is a one-byte value that identifies the type of data the packet contains. Table 4.4 shows a list of currently implemented commands. Although codes higher than 0x0100 are used internally in the Windows server application to identify detection algorithm such as motion matching, they are not used in the Bluetooth link.

Table 4.4 The command codes

Code	Type
0x00	System commands
0x01	Key press
0x02	Accelerometer reading
0x03	Accelerometer reading, when OK key is pressed

The system command is currently only used right after the communication link is established, for the correct identification of the type of phone.

The three **values** field represents the data payload of the packet. The data differs with different command:

- For phone type-identifying system command, all value fields are zero.
- For key press command, value 1 contains the key code of the pressed key, and the rest are zeros.
- For both accelerometer readings commands, the values represent the readings for the *x*, *y*, and *z* direction respectively.

The **error checking code** is a one-byte field that ensures the validity of the received packets. This is performed by computing the XOR of the nine proceeding bits. If the value computed by the server application does not match the value specified in the packet, the packet would be marked invalid and dropped.

4.3.2 Communication between Bluetooth Server and Web Content

Once the Windows server application received the packet, it runs them through the detection algorithms described in Section 4.2. Once an input command is identified, the corresponding JavaScript function will be executed on the webpage. This link is implemented using two different methods:

- Embedded Browser (with Internet Explorer)
- Browser Plugin (with Firefox)

The **embedded browser** implementation uses Component Object Model (COM) and Object Linking and Embedding (OLE) technology to place the Internet Explorer browser inside the server application [31], where scripts on the webpage can be executed directly using the APIs. Figure 4.14 shows a diagram of this implementation and Figure 4.15 presents the detail for the execution of script inside embedded browser window.

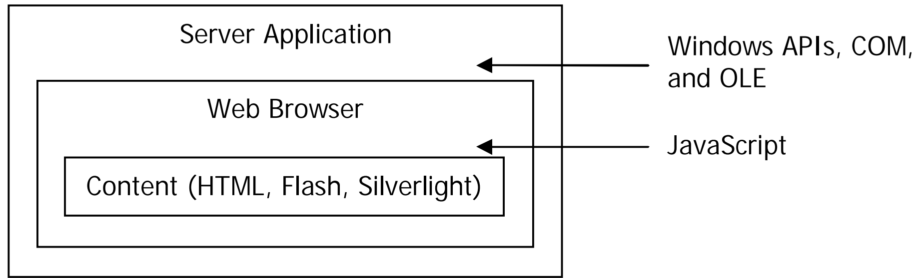

Figure 4.14 Links between server application and web content using windows API, COM, and OLE (Internet Explorer implementation)

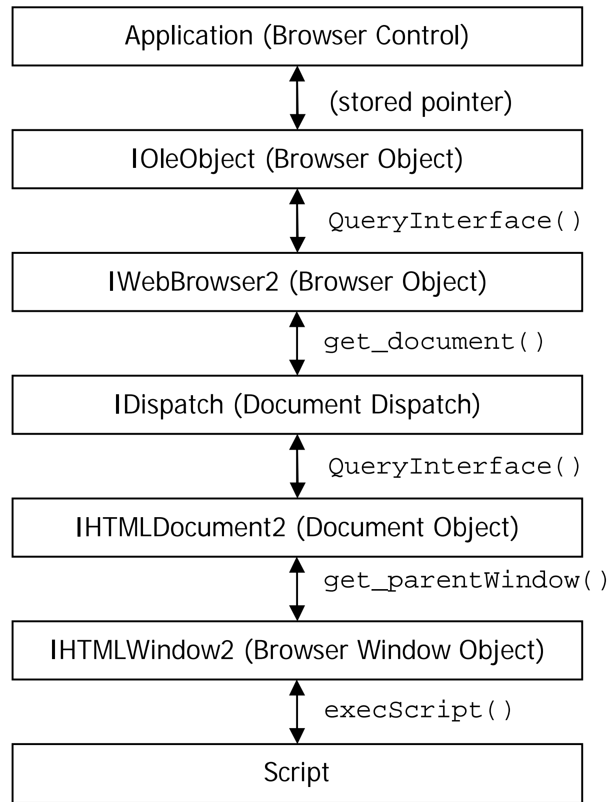

Figure 4.15 Detail for the execution of scripts in embedded browser window

The second method is to implement a **browser plugin** so that the system can be used directly inside a browser. We built this system as a Firefox extension, which is illustrated in Figure 4.16.

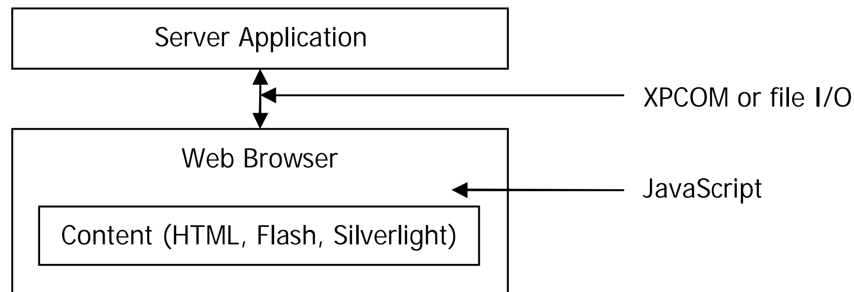

Figure 4.16 Links between server application and web content using browser plugin (Firefox implementation)

The figure shows two independent applications, where the Firefox browser launches the Bluetooth server application in background when the extension is executed. The two separate applications communicate with each other through either Cross Platform Component Object Model (XPCOM) architecture [33] or simple file I/O. Although XPCOM is the proper method, file I/O is used for our prototype due to the faster implementation. The input commands are written to file after they are identified, and the browser extension reads the file and execute the script on the webpage.

Table 4.5 shows a list of commands and the corresponding JavaScript functions that will be executed when the command is identified. If the target webpage does not implement the function, the system will execute the default command function, which will be described in more detail in Section 4.4.4.

Table 4.5 List of commands and the corresponding JavaScript function

Command	Function
Play/Pause Toggle	`Play_Pause()`
Stop	`Stop()`
Mute/Unmute Toggle	`Mute()`
Pan/Turn Up	`Turn_Up()`
Pan/Turn Down	`Turn_Down()`
Pan/Turn Left	`Turn_Left()`
Pan/Turn Right	`Turn_Right()`
Snap Upward	`Snap_Up()`
Snap Downward	`Snap_Down()`
Snap to the Left	`Snap_Left()`
Snap to the Right	`Snap_Right()`
Zoom In	`Zoom_In()`
Zoom Out	`Zoom_Out()`

4.4 Implementations and Results

As a part of this thesis, the cellphone-based remote control module developed in this chapter has been integrated into four separate applications, which consist of parts or all of the system described in Sections 4.2 and 4.3:

- TileShow – a cellphone image viewer that allows user to browse the images based on actions
- Veaver – a cellphone-aided multiview video viewer that uses the phone to control the viewpoint

- Turbo Street View – an improvement on the Google Street View using cellphone as input device
- Cellphone-aided web browser – default application for all webpages without the JavaScript interface implemented

4.4.1 TiltShow

TileShow is a motion-controlled image slideshow application we developed on the cellphone. With a user-defined list of images on the phone, the application changes the image display when the user rotates the phone to the left or right. There is no PC component required for this application; hence, no Bluetooth communication is involved. Table 4.6 shows a list of implemented commands.

Table 4.6 List of commands implemented in the TiltShow application

Command	Function
Pan/Turn Left	Load and display the previous image
Pan/Turn Right	Load and display the next image

Although this application is simple, the idea can be expanded into a generic mobile image viewer that automatically find and load all images within a directory. This system potentially creates a faster and easier way for browsing images without the need of clicking buttons; thus improves the user experience tremendously.

4.4.2 Veaver

Veaver is a 3D modeling system that captures multiview videos of a scene and allows the user to explore the 3D scene formed by these videos. The system is developed by another student in our group. It is similar to the system developed by Carnegie Mellon University in 2001, which was a real-time multiview video system with 30 cameras for capturing the Super Bowl game [34]. However, this system is expensive to construct and maintain, thus it is difficult to expand it into a wider range of applications. In contrast, Veaver is a low-cost user-driven system that utilizes user-uploaded videos captured by cellphone or consumer cameras. Figure 4.17 shows an illustration of this system during a football game, in which many users can capture and upload videos using their cellphones [35]. Since each user has a different seat and thus different viewpoint, an ad-hoc multiview system is formed.

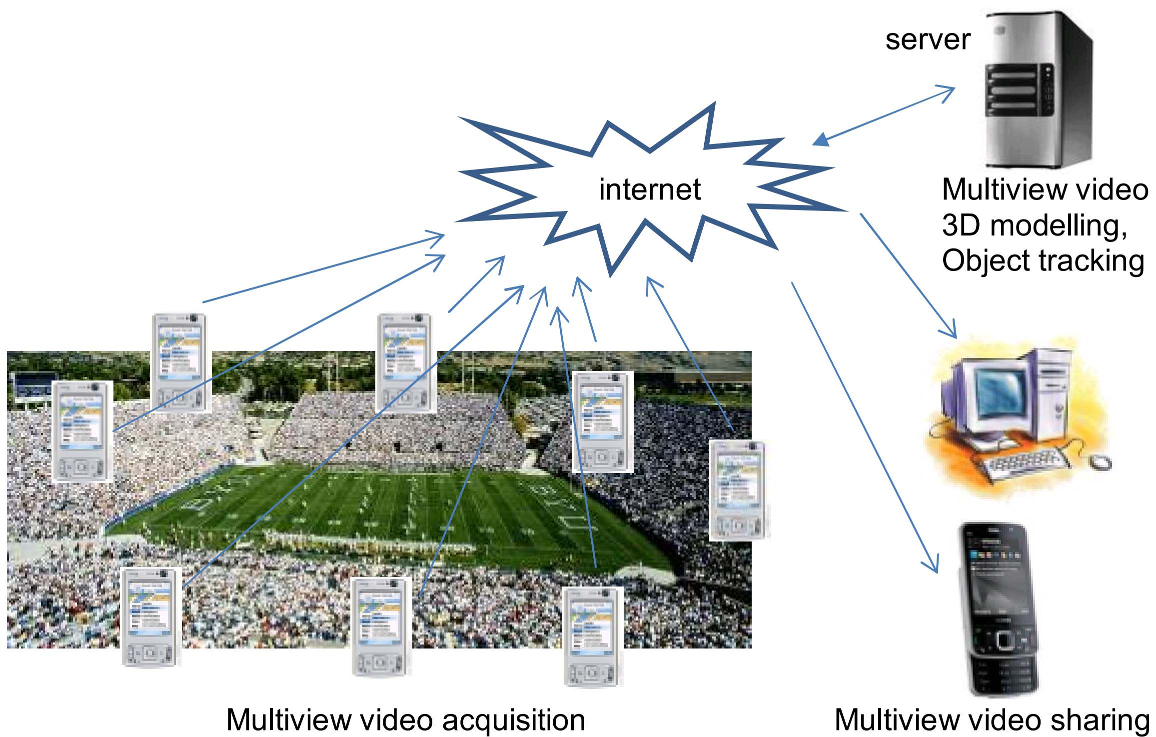

Figure 4.17 Diagram of the Veaver system

This system is similar to Microsoft PhotoSynth [4] with video replaces image as the media content. As shown in Figure 4.17, it consists of a client cellphone application that allows users to capture videos and upload them to a central server. The upload should also contain location and time information to identify which scene it belongs. The server groups the videos into scenes and analyzes them to determine the 3D position and point of view for each video within the scene. From there, a client application is used to allow the user to browse the 3D scene generated by the multiview videos. Figure 4.18 shows a screen capture of this user interface.

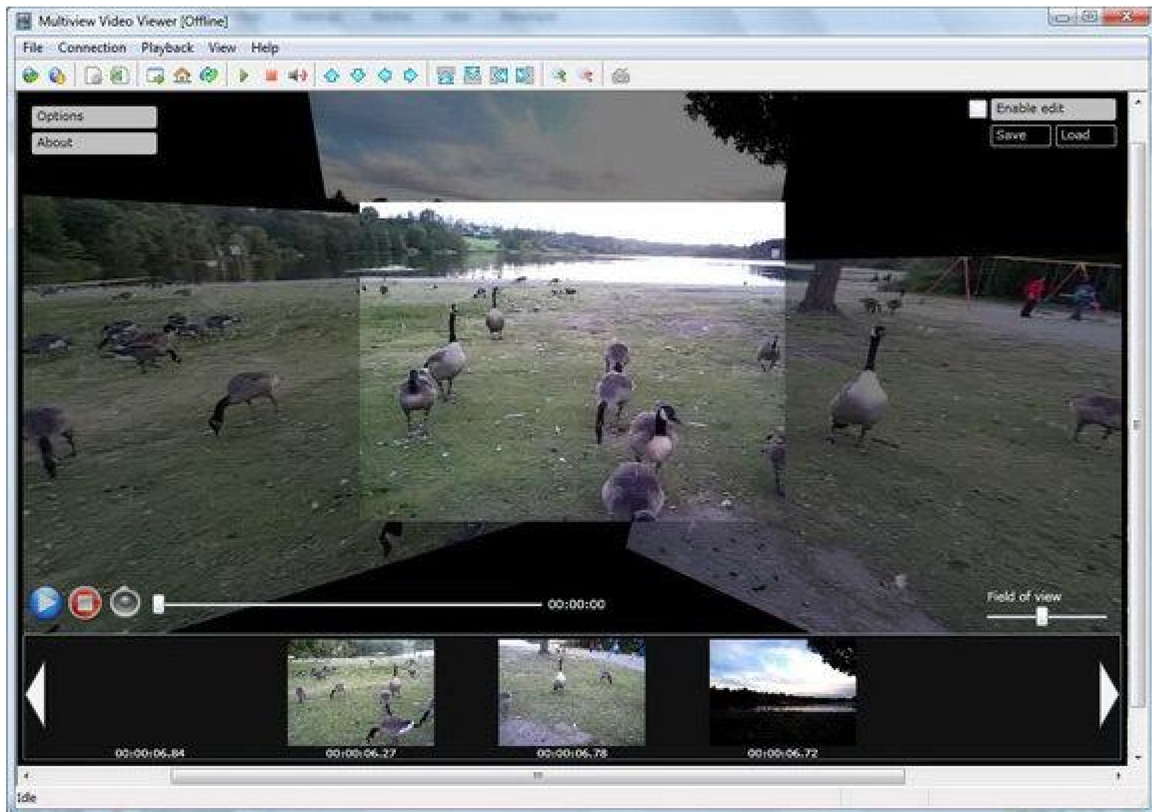

Figure 4.18 The interface for the Veaver application

The interface is developed using Microsoft Silverlight, which was chosen due to its cross-platform compatibility and the easiness of video display and transformations. User can navigate and explore the 3D video scene through the Silverlight-based interface. The cellphone remote control developed in this chapter is also integrated in this system. The remote provides a better user experience in which the user controls the point of view by performing similar motions. Table 4.7 shows a list of commands implemented for this application.

Table 4.7 List of commands implemented in the Veaver application

Command	Function
Play/Pause Toggle	Play or pause the focused video
Stop	Stop the current playing video
Mute/Unmute Toggle	Enable or disable the sound of current playing video
Pan/Turn Up	Rotate the current point of view slightly upward
Pan/Turn Down	Rotate the current point of view slightly downward
Pan/Turn Left	Rotate the current point of view slightly to the left
Pan/Turn Right	Rotate the current point of view slightly to the right
Snap Upward	Focus and rotate to the next view above
Snap Downward	Focus and rotate to the next view below
Snap to the Left	Focus and rotate to the next view to the left
Snap to the Right	Focus and rotate to the next view to the right
Zoom In	Decrease the field of view
Zoom Out	Increase the field of view

4.4.3 Turbo Street View

Turbo Street View is an application we developed that aims at improving the user experience of Google Street View. It is both faster and easier to use. The application is developed using version 3 of the Google Map's JavaScript API [36]. Figure 4.19 shows the user interface of this application running within the embedded web browser.

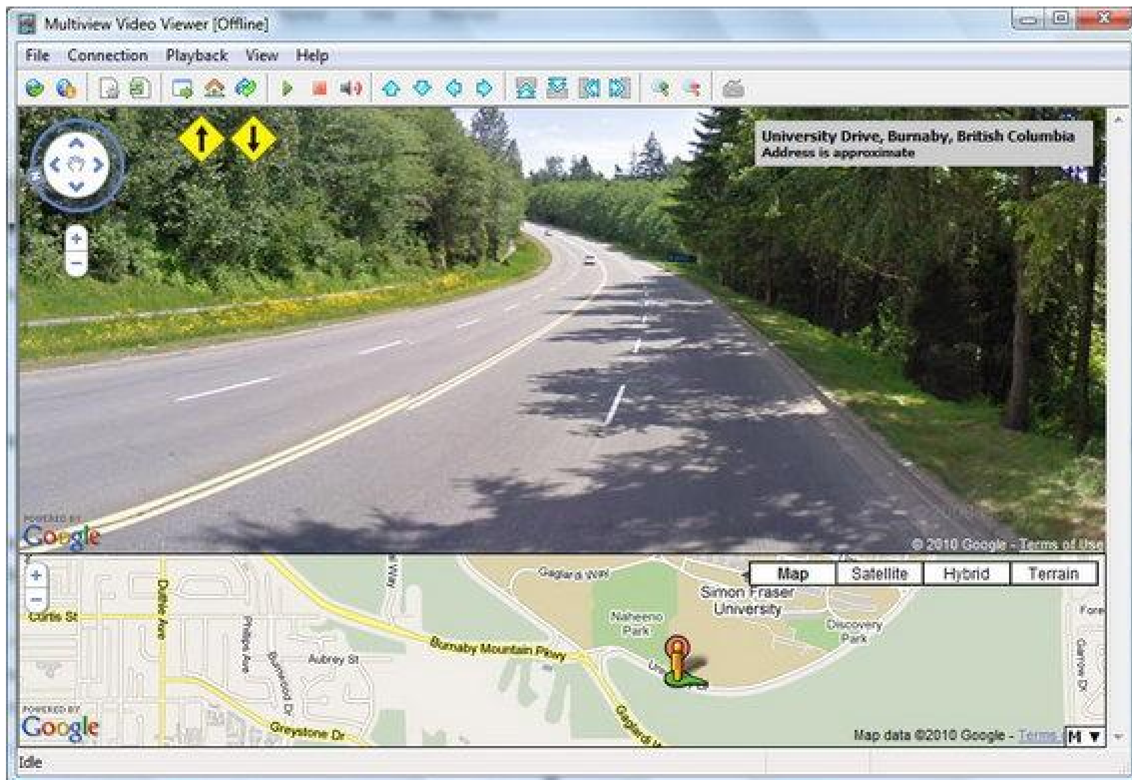

Figure 4.19 The interface for the Turbo Street View application

In Google Street View, user can only navigate using a series of mouse clicks or key presses and have to wait for the transitions and the image loading time – both are often slow and tedious. Turbo Street View solves these issues by proposing two improvements:

- Uses cellphone as remote input device for easier navigation.
- Pre-fetch images to provide faster transition.

In Google Map API, each scene or location that contains the Street View panorama is called a *bubble*. Each bubble contains information required for

locating the next closest bubble in different directions. This information is called *navigation links*, and it contains the location and heading towards the next bubble. Figure 4.20 shows an example of the navigation links of a bubble with forward, backward, and right links available. Note that the road is at a slight curve, so the heading toward the next bubble forward differs slightly from the heading (point of view) of the current bubble.

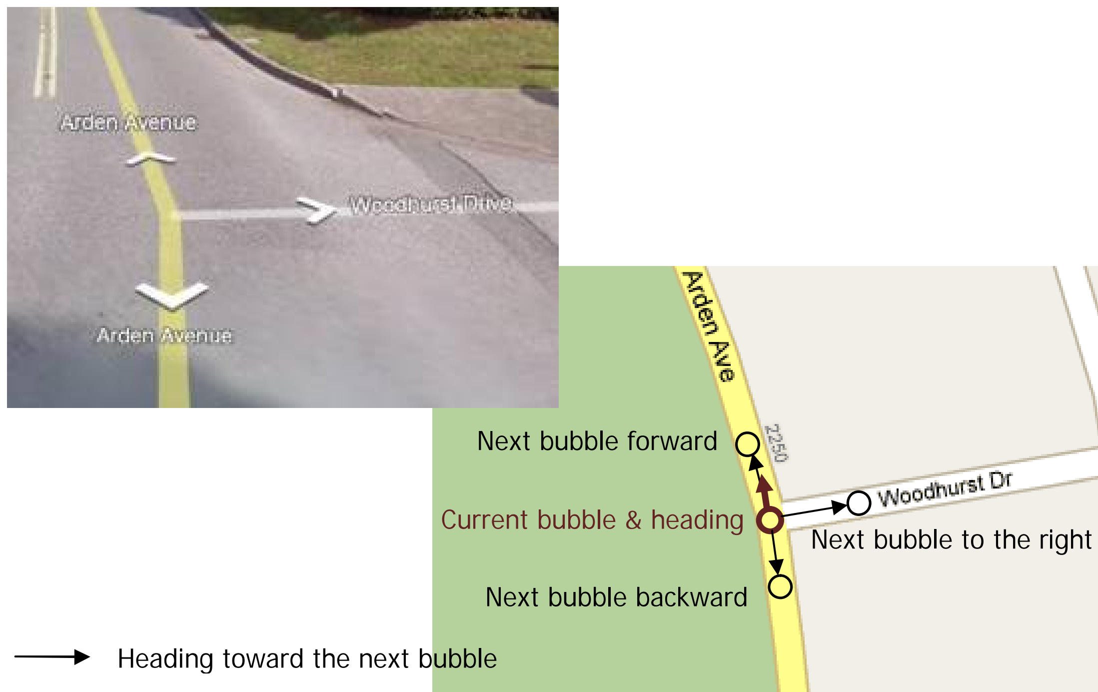

Figure 4.20 Illustration of navigation links of a Street View bubble

Using cellphone as remote input device allows user to move between bubbles or explore the panorama of current bubble by performing similar motions. The user can simulate car driving using the remote control. Table 4.8 shows a list of command implemented in this application. Play, Stop, and Mute commands are not used. If a link is unavailable in the target direction, the

snapping commands will be ignored. There is currently no command for making a U-turn in this application, as the move backward command does not turn the point of view 180 degrees toward the back.

Table 4.8 List of commands implemented in the Turbo Street View application

Command	Function
Pan/Turn Up	Rotate the current POV 1 degree upward
Pan/Turn Down	Rotate the current POV 1 degree downward
Pan/Turn Left	Rotate the current POV 1 degree to the left
Pan/Turn Right	Rotate the current POV 1 degree to the right
Snap Upward	Move forward (if link is available): Jump to next bubble with less than 45˚ turn, and rotate the POV to match link heading
Snap Downward	Move backward (if link is available): Jump to next bubble whose heading is 135˚ ~ 225˚ away, and rotate the POV to the opposite of link heading
Snap to the Left	Turn left (if link is available): Jump to next bubble that is 45˚ to 135˚ to the left, and rotate the POV to match link heading
Snap to the Right	Turn right (if link is available): Jump to next bubble that is 45˚ to 135˚ to the right, and rotate the POV to match link heading
Zoom In	Increase the zoom level by one, to a maximum of 5.
Zoom Out	Decrease the zoom level by one, to a minimum of 0.

For easier navigation, it is desirable to disable upward and downward turning commands to prevent current POV from going into the ground or toward the sky. It is also desirable to assign an easy action to the upward snapping command, as it is most common in this application. For instance, it can be set to

let the application trigger the move forward command when holding the phone in the upright position.

Cellphone remote provides faster and easier control, which increases the demand for faster transition to improve the overall navigation speed. To decrease the transition time, we developed a **prefetching algorithm**. As users are most likely to move forward while exploring a street, the algorithm preloads images in the next bubble forward that has the same POV and zoom level as the current ones. Preloading is triggered by rotations, turnings, and forward movements, and it is performed right after the application has finished loading the current view.

Table 4.9 compares the loading time of street view panorama with or without prefetching algorithm, when performing a forward movement. The test environment consists of:

- Embedded browser application (using Internet Explorer 8)
- Intel Core 2 Duo T8300 running Windows Vista SP2
- 2 Mbps ADSL internet connection
- 800x600 resolution for Street View images with zoom level of 1
- 10 bubbles per location
- 1 to 2 seconds delay between bubbles

Table 4.9 Comparison of Street View panorama loading time with or without prefetching

Location	Images	Average Load Time (ms)			Max Load Time (ms)		
		Without Prefetch	With Prefetch	%	Without Prefetch	With Prefetch	%
Straight #1	60	1092	344	32%	1535	460	30%
Straight #2	60	1067	331	31%	1526	472	31%
Small Curve	60	1026	327	32%	1570	445	28%
Large Curve	108	1036	366	35%	1544	1100	71%
Hairpins	120	1695	703	41%	3105	2398	77%

Locations:
Straight #1 – Gaglardi Way going down SFU (49.267˚N, 122.916˚W)
Straight #2 – East Hastings at Sperling Avenue (49.280˚N, 122.968˚W)
Small Curve – Burnaby Mountain Parkway East (49.272˚N, 122.914˚W)
Large Curve – Clarke Road going up Snake Hill (49.274˚N, 122.871˚W)
Hairpins – Dewdney Trunk Road E. of Stave Lake Dam (49.235˚N, 122.354˚W)

Measurement is done by capturing the `onload` DOM event for each image using JavaScript, and record the time in each call-back function. Result shows that with prefetching enabled, load time of the panorama decreased considerably. Transition is approximately three times faster when the images are pre-loaded. Further speed improvement can be observed as the Google API does not provide the transition animation that is available in Google Street View.

The improvement is most noticeable on a straight or slightly curved road, where the algorithm is able to prefetch all required images for the next bubble. If the road contains large curves, the algorithm may miss some images. This can be observed from the increased amount of images required in Table 4.9. In this

case, the loading time would increases; however, since the algorithm still managed to preload some bubbles partially or even completely, it still provides faster performance.

As a nature of webpage-based applications, performance and experience can vary drastically from user to user due to factors such as network connection and the type of browser used. Table 4.10 shows the performance of Turbo Street View running the "Straight #1" scenario, tested under different environments:

- Faster network (100Mbps instead of 2Mbps)
- Higher screen resolution (1280x800 instead of 800x600)
- Different browser (Firefox 3.6 extension instead of embedded IE8).

Table 4.10 Comparison of Street View panorama loading time with different test environments

Configuration	Average Load Time (ms)			Max Load Time (ms)		
	Without Prefetch	With Prefetch	%	Without Prefetch	With Prefetch	%
Default Configuration	1092	344	32%	1535	460	30%
Faster Network	532	313	59%	682	426	62%
Higher Resolution	1794	724	40%	3106	1281	41%
Firefox Extension	525	130	25%	733	180	25%

Turbo Street View achieves better performance in all scenarios, although the improvement is more noticeable with slower network. Higher screen resolution, where there are 15 images per scene instead of six, also slightly

reduces the speed improvement gained from prefetching. Note that the faster image load time under Firefox may due to the difference in browser architecture and/or the slightly different implementation of JavaScript timer for each browser.

4.4.4 Cellphone Aided Web Browser

For webpages that does not implement the command functions listed in Table 4.5, a default set of functions is used. The use of these functions not only aids users in web browsing, but also prevents JavaScript errors for undefined functions. Table 4.11 shows a list of these functions.

Table 4.11 List of commands implemented in the cellphone aided web browser

Command	Function
Play/Pause Toggle	*Do nothing, only to prevent JavaScript error*
Stop	*Do nothing, only to prevent JavaScript error*
Mute/Unmute Toggle	*Do nothing, only to prevent JavaScript error*
Pan/Turn Up	Scroll page up by 40 pixels
Pan/Turn Down	Scroll page down by 40 pixels
Pan/Turn Left	Scroll page to the left by 40 pixels
Pan/Turn Right	Scroll page to the right by 40 pixels
Snap Upward	Jump to the upper-left corner of the webpage
Snap Downward	Jump to the lower-left corner of the webpage
Snap to the Left	Jump to the upper-left corner of the webpage
Snap to the Right	Jump to the upper-right corner of the webpage
Zoom In	Increase the page font size by 1
Zoom Out	Decrease the page font size by 1

CHAPTER 5: CONCLUSION AND FUTURE WORK

5.1 Conclusion

In this thesis, we proposed a GPU and cellphone-aided multimedia processing system that provides both faster processing speed and better user experience. Three components of the system were discussed: a GPU-based image interpolation algorithm, a GPU-based image encoder, and a system that uses cellphone as remote controller.

GPU is a powerful parallel processor; its SIMD architecture can process a large array of data simultaneously under the same instruction. This architecture benefits computation intensive tasks such as multimedia applications, where each thread inside a streaming processor is responsible for one pixel, thus the thousands of thread in the processor can process thousands of pixels at once. The release of CUDA greatly simplifies GPU programming and unveils the power of GPU to general computing.

The implementation of image resolution upconversion and restoration algorithm on GPU takes the full advantage of this highly parallel nature of the processor, and the result is a 5.2 times speed increase compared to the original CPU implementation. With GPU implementation, real time processing of DVD-sized video may soon to become a reality as a speed of up to 27 fps was

obtained from the initial testing. GPU implementation of JPEG-XR encoder using CUDA enjoys similar speed boost. Experimental results show that the GPU-based codec is almost twice as fast compared to the original CPU-based version. The entire codec, including memory transfer and entropy coding components that are not ported to the GPU, also experiences a modest speed improvement.

Cell phone-aided multimedia processing involves the use of the phone as a remote controller. User motions are captured by the phone's internal accelerometer, and the data values are transmitted to PC via Bluetooth protocol. The reading are then analyzed and matched against a list of predefined actions to determine the user's input command.

Applications developed with the phone remote control system, such as Veaver and Turbo Street View, execute commands that correspond and resemble to the input motion. This poses a fast learning curve, greatly simplifies an otherwise slow and tedious process of exploring multiview images/video, and greatly improves user experience. The system can be further developed into an interactive application that uses motion input to simulate an environment.

5.2 Future Work

This thesis presents three different components of an interactive multimedia manipulation system. These are the preliminary works toward this project; further improvements and system integrations are required. The newest version of Microsoft Silverlight introduces COM support, which makes it possible

to process Silverlight media content using CUDA technology [40]. System that focuses on faster, easier control and improved user experience will mostly require much more processing power. As a result, it is essential to offload some tasks such as video decoding and zooming to the GPU in order to minimize delay and ensure the system is capable of operating in real-time.

Beside system integration, many improvements can be made to various component of the system to further improve usability and performance. Some of which will be described in this section.

5.2.1 GPU Programming Component

The CUDA codes discussed in Chapter 2 and 3 are written for the GPUs that were released before 2010. GPU and CUDA technology has improved rapidly during the past two years, and the code has yet to be updated and optimized for the latest releases. Some concerns described in Section 2.1, 2.3, and 3.2 are no longer an issue with the newest release [8], [22]. For instance, the slow transfer between host and device memories can be partially solved by creating page-locked host memory, where the transfer can be run concurrently as a kernel function. The resource available on the GPU has also been improved greatly with the newest card. Table 5.1 compares the three generations of graphic cards and Table 5.2 shows the memory resource available per active thread [8]. The applications described in this thesis are optimized for compute capability 1.2 and 1.3 devices.

Table 5.1 Maximum available resource on device

Item	Compute Capability		
	1.0, 1.1	1.2, 1.3	2.0
Threads per multiprocessor	768	1024	1536
Threads per block	512	512	1024
Active warp per multiprocessor	24	32	48
Register per multiprocessor	8,192	16,384	32,768
Shared memory per multiprocessor	16 KB	16 KB	48 KB
Shared memory banks	16	16	32
Local memory per thread	16 KB	16 KB	512 KB
Constant memory	64 KB	64 KB	64 KB

Table 5.2 Maximum available memory resource per thread

Type	Compute Capability			Improvement 1.3 to 2.0
	1.0, 1.1	1.2, 1.3	2.0	
Register	10	16	21	1.3x
Shared	21 bytes	16 bytes	32 bytes	2.0x
Local	64 KB	64 KB	512 KB	8.0x

Table 5.2 shows the much higher memory resource available per thread in the newer device. Some changes to the code can be made to fully exploit these additional resources and improve performance. For instance, additional register helps in the downsample kernel described in Section 3.2.2, as register usage is no longer required to be confined to 16. Another example is the doubling of shared memory, which simplifies the caching mechanism in various kernel functions and reduces the length of code and the possibility of branching. With the possibility of even faster performance achievable with newer CUDA driver

and GPU devices, it might be worthwhile to take a closer look at the DCAC prediction, coefficient scanning, and entropy coding components of the JPEG-XR codec to see if GPU implementations are now feasible in any of these areas.

5.2.2 Motion Detection and Bluetooth Communication Component

The current prototype is implemented using Nokia's Sensor R&D plugin, which supports only three types of phones – N82, N93i, and N95 [26]. It is desirable to rewrite the cell phone client application using sensor framework to extend the support to all new Symbian smartphones. It is also desirable to port the application to different platforms such as Microsoft's Windows Mobile, Apples' iPhone and Google's Android.

The Bluetooth communication component also has limited hardware support. The current implementation of the server application uses Windows Socket API, which only supports the hardware devices that uses Microsoft Windows Bluetooth protocol stack. Due to incompatibility, the application will not work on the device that uses Broadcom's Widcomm stack without modifying the driver, which is both dangerous and difficult for average user [38]. It is desirable to implement Broadcom's API given in [39] to extend the support to most Bluetooth devices.

The motion detection algorithm also contains many rooms for improvements. This includes the modification of rotation detection algorithm discussed in Section 4.2.1 so it reset when the reading drops below a certain value instead of ignoring the readings for a fixed interval. This way, the algorithm

can be easily applied to more cases other than rotations. The motion matching algorithm can also be modified so it becomes independent of time and speed of the action. More reliable algorithms can also be developed to improve the accuracy of the detection.

As mentioned in Section 4.4, the orientation of the phone will affect the accelerometer reading, which would also affect the reliability of the motion detection algorithm. Having to hold the phone in a predefined orientation creates a major inconvenience to the user and also make the system less reliable. This issue can be solved by developing a calibration mechanism that either automatically or manually determines the starting position and orientation of the phone. After the initial values are identified, the detection algorithms can be updated accordingly to ensure a correct motion detection result.

5.2.3 Google Street View Component

The current prefetching algorithm used in Turbo Street View works well when the user is moving forward on a straight road. However, as Table 4.9 shows, the performance degrades severely in curvy mountainous roads. The algorithm also fails when the user choose to turn left or right instead of moving forward. This is due to the fact that prefetching applies only to the few images in next bubble forward that has the same point of view as the current one. Even when the user stays in the same bubble for prolonged period of time, the system remains idle and no more images are downloaded.

Miss rate of prefetching can be reduced dramatically with the trade-off of bandwidth. The algorithm can be modified to download as many images as time allows before the next command is received. The images can be grouped into six categories based on possible actions, as listed below:

1. **Forward movement** – if forward link is available, all images in next bubble forward that match the forward link heading (instead of the current point of view)
2. **Turning (left/right) action** – if left or right links are available, all images that has the same heading as the left and right link in the current
3. **Panning action** – all images that surrounds the images that are currently displayed for the current viewpoint
4. **Zooming (in/out) action** – if available, all images in the next (zoom in) and previous (zoom out) level for the current viewpoint
5. The rest of the images in the panorama

The processing order can be determined by either studying the user behaviour while exploring street view or using a training mechanism. It is intuitive to assume that the user is more likely to repeat the previous command, more likely to pan than zoom, and also more likely to zoom in than out at the default zoom level. With these assumptions, the desired prefetch order after different commands is listed in Table 5.3. The implementation of this algorithm is

not straightforward, however, as there is no API available to obtain the street view images address directly, and Google also provides no documentation for manual retrieval.

Table 5.3 Proposed prefetch order based on the previous command

Previous Command	Prefetch Order	Note
Move forward	1 ➔ 2 ➔ 3 ➔ 4 ➔ 5	
Panning	4 ➔ 1 ➔ 2 ➔ 3 ➔ 5	
Turn left or right	1 ➔ 2 ➔ 3 ➔ 4 ➔ 5	1
Move backward	2 ➔ 3 ➔ 4 ➔ 5	2
Zoom in or out	4 ➔ 3 ➔ 1 ➔ 2 ➔ 5	3

Notes:

1: Assume the user is more likely to move forward than turning twice in a row, which either equals to a U-turn or going back to original heading (already cached)

2: For Turbo Street View only; forward images in the next bubble are already cached

3: Image required for zooming in should be processed first, unless the current view is at the lowest zoom level

Recently, Microsoft has released Bing Streetside to compete with Google Street View [40]. However, only a few cities are currently supported as it is still in the Beta phase. In the future, the Turbo Street View can be modified to support Bing Map as its coverage expands and the API becomes available.

REFERENCE LIST

[1] Nintendo Europe, "Technical Details," http://www.nintendo.co.uk/NOE/en_GB/systems/technical_details_1072.html

[2] A. Smolic, K. Müller, P. Merkle, C. Fehn, P. Kauff, P. Eisert, and T. Wiegand, "3D Video and Free Viewpoint Video – Technologies, Applications and MPEG Standards," *IEEE International Conference on Multimedia and Expo (ICME)*, Toronto, Canada, pp.2161-2164, July 2006.

[3] R. Hartley and A. Zisserman, "Multiple View Geometry in Computer Vision," 2nd edition, Cambridge University Press, 2004.

[4] Microsoft PhotoSynth. http://photosynth.net/

[5] D. Anguelov, C. Dulong, D. Filip, C. Frueh, S. Lafon, R. Lyon, A. Ogale, L. Vincent, J. Weaver, "Google Street View: Capturing the World at Street Level," *Computer*, vol.43, no.6, pp. 32-38, June 2010.

[6] J. Cao, M.-C. Che, X. Wu, and J. Liang, "GPU-aided Directional Image/Video Interpolation for Real Time Resolution Upconversion," *IEEE International Workshop on Multimedia Signal Processing (MMSP)*, Rio de Janeiro, Brazil, Oct. 2009.

[7] M.-C. Che, and Jie Liang, "GPU Implementation of JPEG XR," *Proc. 2010 SPIE Visual Information Processing and Communication Conference*, San Jose, CA, USA, vol.7543, no.1, Jan. 2010.

[8] NVIDIA CUDA C Programming Guide, 2010. http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf

[9] General-Purpose Computation on Graphics Hardware, 2009. http://www.gpgpu.org/

[10] D. Kirk, W. Hwu. Programming Massively Parallel Processors, Spring 2010. http://courses.engr.illinois.edu/ece498/al/Syllabus.html

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] Santa-Cruz, D., Ebrahimi, T., Askelof, J., Larsson, M., Christopoulos, C., “JPEG 2000 still image coding versus other standards,” *Proc. SPIE 4115*, 446-454 (2000).

[17] Srinivasan, S., Tu, C., Regunathan, S., Sullivan, G., “HD Photo: A new image coding technology for digital photography,” *Proc. SPIE 6696* (2007).

[18] Srinivasan, S., Tu, C., Zhou, Z., Ray, D., Regunathan, S., Sullivan, G., “An Introduction to the HD Photo Technical Design,” (2007).

[19] HD Photo Device Porting Kit 1.0. http://www.microsoft.com/whdc/xps/hdphotodpk.mspx

[20] Microsoft Corporation, HD Photo Bitstream Specification, Version 1.0, November 2006.

[21] The CUDA Compiler Driver NVCC (2009).

[22] NVIDIA CUDA C Programming Best Practice Guide (2009). http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide_2.3.pdf

[23] NVIDIA Quadro – Product Comparison. http://www.nvidia.com/object/IO_11761.html

[24] NeeMee – Nokia2MovingExperience by Asier Arranz (A2Jsoft). http://www.niime.com/index.htm

[25] WeeWheel. http://www.weewheel.com/

[26] Forum Nokia, “Nokia Sensor APIs.” http://wiki.forum.nokia.com/index.php/Nokia_Sensor_APIs

[27] STMicroelectronics, “MEMS motion sensor. 3-axis - ± 2g/± 8g smart digital output piccolo accelerometer.” http://www.st.com/stonline/products/literature/ds/12726/lis302dl.pdf

[28] Bluetooth SIG, “Bluetooth High Speed – V3.0 + HS Data Sheet.” http://www.bluetooth.com/SiteCollectionDocuments/HS_Doc_Web.pdf

[29] Bluetooth SIG, “Core Specification v2.0 + EDR.” http://www.bluetooth.com/Specification%20Documents/Core_v210_EDR.zip

[30] Microsoft Developer Network (MSDN), “Bluetooth Programming with Windows Socket.” http://msdn.microsoft.com/en-us/library/aa362928%28v=VS.85%29.aspx

[31] Microsoft Developer Network (MSDN), “Component Object Model (COM).” http://msdn.microsoft.com/en-us/library/ms680573%28v=VS.85%29.aspx

[32] Microsoft Support, “OLE Concepts and Requirements Overview.” http://support.microsoft.com/kb/86008

[33] Mozilla Developer Centre, “XPCOM.” https://developer.mozilla.org/en/XPCOM

[34] Carnegie Mellon goes to the Super Bowl. http://www.ri.cmu.edu/events/sb35/tksuperbowl.html

[35] J. Liang, “VSynTube: A Multiview Video Acquisition, Sharing and 3D Modelling System Using Nokia Phones.”

[36] Google Map JavaScript API V3. http://code.google.com/apis/maps/documentation/javascript/

[37] Planet Marshall, “Silverlight and CUDA interop.” http://www.planetmarshall.co.uk/2010/01/silverlight-and-cuda-interop/

[38] Using the Microsoft Bluetooth Stack (instead of WIDCOMM) on Windows XP with SP2. http://www.shootingsoftware.com/Widcomm.htm

[39] Broadcom, “Development Kit Download.” http://www.broadcom.com/support/bluetooth/sdk.php

[40] Bing Map Blog, “Bing Maps Adds Streetside, Enhanced Bird’s Eye, Photosynth and More.” http://www.bing.com/community/Site_Blogs/b/maps/archive/2009/12/02/bing-maps-adds-streetside-enhanced-bird-s-eye-photosynth-and-more.aspx