REAL-TIME IMAGE REGISTRATION AND ITS APPLICATION IN MOTION-VISUAL HYBRID CONTROLLER

JUN ZHENG

Real-time Image Registration and Its Application in Motion-Visual Hybrid Controller

by

© Jun Zheng

A thesis submitted to the

School of Graduate Studies

in partial fulfillment of the

requirements for the degree of

Master of Science

Department of Computer Science

Memorial University of Newfoundland

September 2010

St. John's Newfoundland

Abstract

Motion based controllers, such as the Wii Remote, provide users with brand-new gaming experiences and are becoming more popular. However, the accuracy of the motion sensor limits their usages in precision critical games. Instead of the motion based approach, image processing techniques could be used to provide higher accuracy due to their high pixel resolution.

The goal of this thesis is therefore to propose a highly accurate controller that utilizes visual inputs. Users can control cursor in 2D screen by waving the controller toward any place which has textures. The thesis first proposes an image registration algorithm that runs in real-time on graphics hardware, then uses it to build a highly accurate visual based controller through camera focal tracking, and finally further improves the robustness of the controller under fast motion by utilizing both motion and visual information.

Real-time image registration is achieved by implementing the Inverse Compositional Algorithm in parallel using Compute Unified Device Architecture (CUDA). A number of CUDA optimization techniques have been applied and evaluated. The final optimized implementation achieves 150 times speed up over the sequential implementation, more than sufficient for real-time application. To improve the robustness of image registration, the coarse-to-fine processing scheme is also applied and two multi-resolution variants of the image registration algorithm are discussed.

Experiments conducted demonstrate that, using the proposed real-time image registration algorithm, the visual based controller achieves much higher control accuracy than the motion-sensor based approach. The performance under fast motion can be further improved through using the input from the motion sensor as a priori knowledge to assist the image registration process.

Acknowledgements

Foremost, I would like to thank my supervisor sincerely, Dr. Minglun Gong, for his continuous support, guidance and insightful advices.

I wish to thank the School of Graduate Studies of Memorial University, who provides the funding for my research. This thesis would not be possible without the help of the faculty, students, and staff of the Department of Computer Science of Memorial University.

Last but not least, I would like to thank my family: my parents Zhichun Zheng and Yingxuan Wu who support me spiritually throughout my life and my aunt and her husband, Yingshan Wu and Yi Dong who provide finically support for my living cost so that I can completely focus on my research.

Table of Contents

Abstract ii

Acknowledgements iv

List of Tables viii

List of Figures ix

Chapter 1 Introduction 1
1.1 The Evolution of Game Controller 1
1.2 Motivation 3
1.3 Organization of the Thesis 4

Chapter 2 Related Works 6
2.1 Image Registration 6
2.1.1 Feature-Based Approaches 6
2.1.2 Intensity-based Approaches 7
2.1.3 The Inverse Compositional Algorithm 8
2.1.4 Multi-Resolution Image Registration 9
2.1.5 Template Matching 10
2.2 Motion Controllers and Motion Sensing 10
2.3 GPGPU Programming 11
2.4 Optimization of CUDA programming 12

Chapter 3 Image Registration using Inverse Compositional Algorithm and Its Parallel Implementations 14
3.1 The Inverse Compositional Algorithm 15
3.2 Sequential Inverse Compositional Algorithm 17
3.3 Implementing ICA for GPU Processing 19
3.4 Multi-resolution Image Registration 22
3.5 Multi-resolution Image Registration with Select Regions (MRIR-SR) 25
3.5.1 Region Selection 26
The coefficient k is used to convert the coordinate to the fine level. 27
3.5.2 Locating the Corresponding Region in the Input Image 27
3.5.3 Implementation 28
3.6 Discussion 31

Chapter 4 Optimization of the Image Registration Module on CUDA 33
4.1 Introduction of CUDA 33

4.2 Implementation 35
4.3 Optimization Techniques 37
4.3.1 Balancing Workload Distribution 37
4.3.2 Parallel Reduction 37
4.3.3 Sequential-Then-Parallel Processing 39
4.3.4 Memory Access Pattern 43
4.4 Bottleneck Optimization 44
4.4.1 Local Array Buffered Approach 45
4.4.2 Partial Reduction Approach 46
4.4.3 Unrolled Register Buffered Approach 48
4.5 Evaluation 49
4.5.1 Evaluation of the General Optimization Techniques 50
4.5.2 Evaluation of the Optimization Approaches for Bottleneck 51
4.6 Discussion 56

Chapter 5 Motion and Visual Controller 58
5.1 Visual Based Controller 58
5.1.1 Threading image capturing and image registration 59
5.1.2 Mouse control 61
5.1 Motion Based Controller 62
5.1.1 Introduction of Wiimote 62
5.1.2 Motion Estimation 63
5.1.3 Estimating Angle of Roll 64
5.1.4 Implementation 64
5.2 Hybrid Controller 65
5.2.1 Motion Based Pre-Knowledge 66
5.2.2 Floating Window 67
5.2.3 Implementation 69
5.3 Accuracy Evaluation 70
5.4 Discussion 73

Chapter 6 Other Applications and Conclusions 75
6.1 Light-Gun for LCD 75
6.2 Real-Time Image Mosaic 76
6.3 Conclusions 77

Chapter 7 References 80

Appendix A 83
A.1 Bank conflict 83
A.2 Atomic Reduction Without Bank Conflicts on CUDA Device 84
A.3 Optimized Parallel Reduction on CUDA Device 85
A.4 Parallel Reduction for Vectors on CUDA Device 86

A.5 Local Array Cached Parallel Reduction Approach..... 88

List of Tables

Table 4-1 Evaluation of bottleneck optimization approaches 52

List of Figures

Figure 1-1 Prototype of the hybrid controller 4
Figure 3-1 GPU-based implementation of ICA algorithm 20
Figure 3-2 Modules of the GPU-based ICA algorithm 21
Figure 3-3 Multi-resolution image registration 22
Figure 3-4 Comparison between standard IR and MRIR using the same images 24
Figure 3-5 Implementation of MRIR-SR 30
Figure 3-6 Comparison between different image registration approaches 31
Figure 4-1 Hierarchy of CUDA threads 34
Figure 4-2 Structure of the image registration module 36
Figure 4-3 Parallel reduction hierarchy built by the building block which is capable of reducing 512 elements 39
Figure 4-4 Automatically scheduled strategy for aggregating kernels 40
Figure 4-5 Sequential-then-parallel processing 41
Figure 4-6 Local array buffered parallel reduction approach 45
Figure 4-7 Local array buffered approach 46
Figure 4-8 Partial reduction approach 47
Figure 4-9 Unrolled register cached approach 48
Figure 4-10 Evaluation of the general optimization techniques 50
Figure 4-11 Performance of the bottleneck optimization approaches under different blocks per multiprocessor setting 54
Figure 4-12 Performance comparison between the bottleneck optimization approaches 55
Figure 5-1 Tracking focal movement by the registration result 59
Figure 5-2 Implementation of the visual based controller 60
Figure 5-3 Wiimote motion sensing 62
Figure 5-4 Motion estimation 63
Figure 5-5 Block diagram of the motion based controller 64
Figure 5-6 Motion based pre-knowledge 67
Figure 5-7 Floating window 69
Figure 5-8 Implementation of the hybrid controller 70
Figure 5-9 Procedure of the accuracy test 71
Figure 5-10 Testing result of the small motions 72
Figure 5-11 Testing result of the complex motions 73
Figure 6-1 Real time image mosaic 76

Chapter 1 Introduction

Nowadays, the input methods for game console controllers are not limited to game pads with push buttons. New types of game controllers are coming to enhance users' gaming experience. Nintendo introduces Wii remote controller (or Wiimote) for its video game console - Wii. Wiimote has integrated accelerometers and attached gyroscopes to track motion. Users wave the controller to play video games. This creative device brings about the prevalence of motion controlled videos games. However, the acceleration and gyroscope based approach lacks accuracy. In this thesis, we explore an integrated system of visual based approach and motion based approach to track user input. Image registration is used to track the view direction of the camera (the center of the captured image). Users hold a wand containing a combination of video camera/accelerometer/gyroscope. Waving the wand toward any place with textures provides users with accurate control of cursor on a 2D screen.

1.1 The Evolution of Game Controller

Since the inventions of game pads and analog sticks as the controllers for the first generation consoles, input devices for video game consoles never stop evolving. More and more techniques are being introduced to enrich gaming control experiences. In 2003, Sony released the Eye Toy – a digital camera device for the PlayStation 2. For the first time, computer vision techniques, such as gesture recognition were used in consoles. Unfortunately, limited by the performance of the camera and the processing capability of PS2, only several games used the camera.

Three years later, Nintendo showed again her talent in creating brand-new gaming experience. As the main controller of her new console Wii, Wii remote controller (or Wiimote) was the most versatile input device in console history. Integrated with motion and infrared ray sensors, Wiimote is not only a conventional game pad but also a motion tracker and a light-gun. It brought out the prevalence of motion controlling games. Nevertheless, Wiimote is not perfect. While providing novelty experience for casual gamers, enthusiasts and professionals complain about the control accuracy. Until now, conventional gamepads are still the best choice for high competitive console video games.

The competitors of Nintendo are creating new tools to improve the accuracy. For more powerful consoles, new hybrid gaming control systems, Kinect for X-Box 360 and PS Move for Playstation 3 are on the way. Both introduce multiple input techniques, such as motion sensor and voice recognition. Though the hardware specifications of the two relative old consoles limit their performances, HD video games (abbreviate for High Definition video games, an unofficial term for video games with resolution larger than 1280×720) finally have the chance to get rid of the old conventional game pads and bring gamers to the motion controlling world.

Both of the new products can take vantages of image processing techniques. A video camera is native integrated to Kinect for X-Box 360, whereas PS3 has an accessory camera device, called PS Eye, which can be added to PS Move system. Compared to motion sensor solution, image processing has higher accuracy up to pixel level. However, on both Kinect and PS Move systems, the video cameras are fixed near televisions and

facing users. They track user gestures or the signals from the special devices, but won't be able to make use of the available motion information.

1.2 Motivation

This thesis studies how to integrate video cameras with motion controllers so that both visual and motion inputs can be utilized for high precision ego-motion estimation. We are motivated by the following facts: motion information captured by motion sensors is often noisy, which makes it unreliable under small motion due to the low signal-to-noise ratio. Yet, it provides robust information under fast motion. Visual feedback from cameras, on the other hand, is a useful resource for estimating the ego-motion through registering images captured in adjacent frames. Nevertheless, adjacent frames may differ a lot under fast motion, causing image registration to fail due to matching ambiguities. Visual feedback also provides absolute positioning and therefore does not suffer from drift.

Combining visual input with motion input, therefore, seems to be a logical choice for providing robust and highly accurate motion information. To achieve this goal, we build a prototype controller using existing hardware: a Wii motion controller, a high performance video camera, and a PC with programmable graphics card. We also implement image registration algorithm on graphics hardware to achieve real-time processing speed. The prototype is illustrated by **Figure 1-1**.

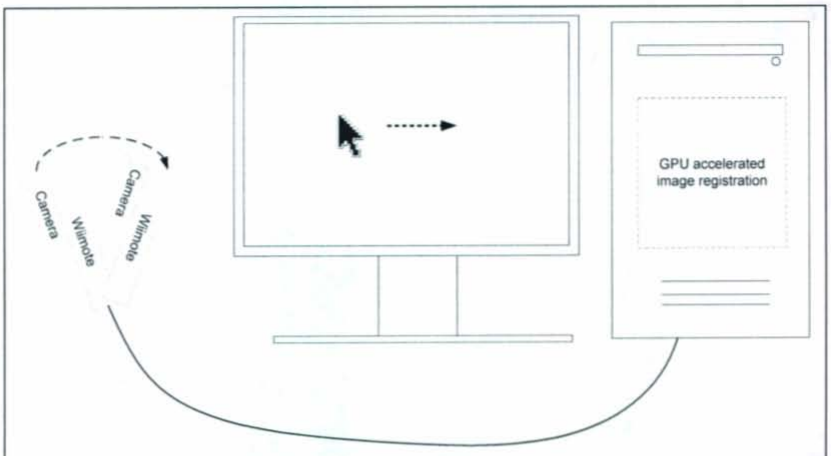


Figure 1-1 Prototype of the hybrid controller

1.3 Organization of the Thesis

The remaining of this thesis is organized as the following: The next chapter discusses previous work related to this thesis. This includes image registration approaches, motion controller and motion sensing, GPU programming and optimization of CUDA code.

Chapter 3 covers the image registration algorithms. First, we introduce the Inverse Compositional Algorithm (ICA). It is an efficient intensity-based image registration algorithm, whose parallel implementation is used as the building block of the proposed registration system. We then propose our techniques to enhance ICA using coarse-to-fine processing strategy. We also propose the Multi-Resolution Image Registration with Select Regions to increase the registering speed and versatility.

Chapter 3 discusses the detailed implementation and the optimization of the parallel ICA on CUDA. The organization of the whole system is first presented and the optimization techniques are then discussed. The techniques include balancing sequential and parallel workload, parallel reduction, sequential-then-parallel processing, and memory access patterns optimized for CUDA devices. More important, we modify the bottleneck of the ICA and use various buffering methods to reduce high-latency memory access. Tests show that the hard-coding enforced register buffering approach yields the best result. Several experiments results are shown at the end of this chapter to demonstrate the efficiencies of the proposed optimization approaches.

In Chapter 5 , we compare three types of controllers that use different information for pointing task. Using a video camera only, we can build a high accuracy visual based pointing controller. If we only have a motion sensor, an efficient motion based air-mouse can be built. By analyzing the pre-knowledge provided by motion sensors, the motion-visual-hybrid controller works well even during high speed movement.

The proposed real-time image registration system not only has application in building controllers, it can also be applied to other real-vision applications. In Chapter 6 , we will discuss how to use it to correct light-gun and generate real-time image mosaic. At last, the thesis is concluded.

Chapter 2 Related Works

2.1 Image Registration

Image registration is the process that transforms different sets of images into one coordinate system. It is widely used in computer vision, medical imaging, and so on. Different image registration algorithms can be classified into two categories – the feature-based approach and the intensity-based approach.

2.1.1 Feature-Based Approaches

The feature-based approaches evaluate the transformation between the two images based on the extraction of salient structures or features in images [1]. The features should be distinct, stable and easily detectable in both images. Various feature detection techniques are proposed to extract features, from the previous Moravec's corner detector [2] to the recent Lowe's Scale-Invariant Feature Transform (SIFT)[3]. The detected features are then used to be matched between images to estimate the transformation model. To make the registration result robust against outliers, Fischler and Bolles' Random Sample Consensus (RANSAC) [4] technique is often used.

The extracted features not only can be used to register images, but also to estimate camera locations. Davison proposed the real-time single-camera-based localization through featured-based mapping [5]. Davison discussed mutual information for the active search to extract useful features [6]. In high frame-rate application, the active search is expected to be more accurate because of the relative small search region. It also requires less processing time.

The feature-based approaches do not compare intensity values and work well when illumination changes or images are sensed by the different devices. On the other hand, they cannot handle scenes that have limited number of distinct features.

2.1.2 Intensity-based Approaches

The intensity-based approaches do not detect features but compare the intensity difference of the whole images or part of the images via certain correlation criterion. The examples are the Normalized Cross Correlation (NCC) [7] and Sequential Similarity Detection Algorithm (SSDA) [8] that sequentially search the optimum to estimate translation between images. More versatile and low computational cost algorithms are also proposed, such as the Lucas-Kanade algorithm [9], which can estimate general projective transformation. The intensity-based approaches do not require salient structures in images. However, because of the pixel-by-pixel calculation, they have high computational complexities; and, because of the intensity comparison they are also sensitive to illumination change.

Intensity-based approach generally works well for registering images captured by a single camera due to the following two reasons: First, images sensed by one camera are consistent in color. Secondly, adjacent frames captured by a camera at full frame rate often share adequate overlapped regions. Although the image traversing is a high cost operation, the uniform calculations for each pixel provide large amount of parallelism which could be speeded up by the widely used Single Instruction Multiple Data (SIMD) based computing devices, such as graphics cards.

Conventionally, intensity-based approaches work on image patches to reduce computational cost. However, the proposed GPU accelerated intensity based approaches are very efficient, even for large images. Therefore, we use the entire images to perform registration rather than using image patches.

2.1.3 The Inverse Compositional Algorithm

For the proposed pointing system, we need an image registration algorithm that can estimate a 2D projective transformation. The Lucas-Kanade algorithm is a potential choice. This algorithm updates the warp parameters by iteratively minimizing the intensity difference between one image and another warped image. In each loop, the algorithm needs to re-evaluate a Steepest Descent Image (SDI) to direct the local parameter updating and then to calculate a Hessian matrix by aggregating SDI. These two steps have high computational cost. Especially calculating Hessian matrix is not suited for a GPU application due to the non-parallel nature.

To address this problem, Baker and Matthews [10] proposes the Inverse Compositional Algorithm (ICA). They refer the Lucas-Kanade algorithm as the forward compositional algorithm since the target image is warped. By contrast, ICA warps the original image back to the target image. Both the SDI and the Hessian matrix are pre-calculated at where all warp parameters are equal to zero and reused during the iteration, rather than re-evaluated at each iteration. Experiment shows that pre-calculating SDI and Hessian matrix takes about 7% execution time when the algorithm iterates 100 times. If we chose Lucas-Kanade algorithm which performs the re-evaluating in each iteration, there would be a huge slow down.

Both the Lucas-Kanade algorithm and the ICA require images to have adequate overlapping areas for a successful registration. For adjacent frames captured at a full frame rate by a slowly waving camera, this requirement is generally met. However, when the speed of movement is too fast, both algorithms may fail.

In this thesis, we choose ICA because it can be efficiently implemented using graphics hardware. We also propose techniques for enhancing the robustness of the registration under fast camera movement.

2.1.4 Multi-Resolution Image Registration

A multi-resolution analysis strategy, which is discussed by Burt [11], is often used to enhance the intensity-based image registration. This coarse-to-fine strategy matches images on a coarse level and then tunes the result on fine levels. Wong and Hall [12] apply this strategy to SSDA method to speed up registration speed. Gaussian and Laplacian pyramids are used in Kumar et al.'s[13] works to register aerial video sequences. Baker and Matthews [10] point out that faster convergence of ICA can be obtained by processing hierarchically on a Gaussian pyramid.

Besides the advantage of reducing computational time, the pyramid can potentially increase the robustness. The coarse images contain large-scale features that tend to yield fast convergence and they also remove the detailed textures that may lead the algorithm to local minimums. In this thesis, we will combine the registration system with the pyramid to increase its robustness. Because of the high efficiency of the GPU acceleration, we perform the pyramid on the whole images instead of image patches.

2.1.5 Template Matching

Template matching searches a sub-image from another template image. It can be obviously used to handle translating registration by extract the template from the image to be transformed. The template matching algorithm usually locates the template position by finding the minimum distortion, or maximum correlation, between the template and the all possible sub-images of the referenced image. Examples for the measuring equation are the Sum of Absolute differences (SAD) and Sum of Squared Differences (SSD), both are easy to implement. For better robustness, Normalized Cross-Correlation is often used [14-16].

Image registration using template matching can be implemented efficiently but it can only handle translation. In contrast, we perform Inverse Compositional algorithm, which can handle all kinds of projections.

2.2 Motion Controllers and Motion Sensing

Wii remote controller, or Wiimote, is the first wide-used motion controller for mainstream gaming consoles. With integrated accelerometers, it can measure accelerations ranged from -3g to + 3g along three perpendicular directions.

Because it is easily programmable, its potential usages are being investigated by many researchers. Wong et al. [17] use it to build an interactive music performance system by analyzing acceleration patterns. Schlomer et al. [18] employ it to perform gesture recognition. Cheong [19] uses it to build a interactive teaching and learning platform.

In 2009, Nintendo releases an add-on to Wiimote, called as MotionPlus, which can measures angular velocities with built-in gyroscopes. Although currently there are very few documents about MotionPlus, the gyroscopes combined with accelerometers have already been researched for motion tracking, such as orientation estimation discussed by Luinge [20] and motion capture discussed by Sakaguchi [21].

Unlike the above research that use motion information only, we here how to combine the motion based feedbacks from the Wiimote with the visual based feedback from a video camera.

2.3 GPGPU Programming

As the advancing of streaming processors, the graphics processing unit (GPU) is being widely used to implement general-purpose parallel applications. As mentioned by Harris [22], GPUs provide much more and faster growing computational power than CPU's. Existing GPU applications involving image processing [23-25], video decoding [26, 27], physics simulation [28, 29], searching [30] and sorting [31] show huge accelerations.

Previous GPUs are not specifically designed for general-purpose computations. Programmers have to use a graphics programming model, such as shading languages, which is designed for graphics rendering. This programming model is not optimized for other usages and lacks efficient local communication mechanisms, which is a common case in general-purpose executions.

Now, more and more general-purpose products are released to make GPU coding more enjoyable. They provide convenient high level programming languages to implement

parallel applications. On-chip storage devices are used to accelerate local communication. On those devices, implementations are more intuitive and efficient. The examples of the GPU programming models are the Compute Unified Device Architecture (short for CUDA) designed for the GPUs manufactured by the integrated circuit supplier – nVidia and the cross-GPU OpenCL. DirectCompute, which is the graphics API that the latest graphic card will support in hardware, would become the standard for PC gaming industry. It will be integrated with Microsoft DirectX 11. It is also highly possible to become the standard programming model for the next console of Microsoft. It is very likely that other console providers would release similar products to take advantage the horsepower of GPU computing.

CUDA is selected in this thesis for implementing the image registration algorithm in parallel. The C-like programming interface is intuitive for programmers to design parallel applications. CUDA complier and linker can generate C++ objects or libraries that can be easily linked to the standard C++ programs. This feature makes integrating GPU accelerating programs very convenient. CUDA also provides features to enable runtime debugging which greatly eases the code debugging and maintenance.

2.4 Optimization of CUDA programming

Previous research has shown that how well the implementation is optimized for graphics hardware can greatly affects the performance of the algorithm. Several papers have discussed how to optimize CUDA code. In [32], Harris discuss step by step about how to optimize the parallel reduction by both algorithmic optimizations and code optimizations. The final result is as 30.04 times fast as the original straightforward version. We modify

Harris' work to perform the inter-thread summation in our CUDA kernels. Harris, Sengupta and Owens discuss how to optimize parallel prefix sums in [33]. Their on-chip memory accessing pattern to resolve accessing conflict provides the inspiration to design the optimized atomic addition proposed in Appendix A.1 and parallel reduction for vector proposed in Appendix A.4 . For more specific application instead of general algorithms, in [25], Yu and Chen propose the techniques to optimize their stereo vision system. Especially, they discuss how to tune the workload distribution to acquire high occupancy of the stream processors which leads to high instruction throughput. In our experiments, we find that high occupancy does not always means high performance, especially for the memory band-width limited applications which benefits more by trading occupancy for memory throughput.

Based on our parallel implementation of the ICA algorithm, we experimented with various optimization techniques and evaluated their effectiveness. In the end, we not only have an optimized implementation that accelerates the processing speed by up to six times, but also obtained valuable insights on how to optimize other image-related CUDA applications.

Chapter 3 Image Registration using Inverse Compositional Algorithm and Its Parallel Implementations

The Inverse Compositional Algorithm (ICA) is an image registration tool to estimate the homographic transformation between two images. The algorithm runs too slow on current PCs for real-time applications, it also has huge parallelism which could be accelerated by a GPU.

In this chapter, we discuss a GPU-based image registration algorithm designed based on ICA. Moreover, we discuss how to improve the registration robustness using multi-resolution processing and to increase the versatility using selected region registration.

Compared to the sequential ICA implementation, the proposed GPU-based image registration (IR) algorithm achieves a speedup of up to 150 times. After integrating the IR algorithm with coarse-to-fine processing scheme, the resulting multi-resolution image registration (MRIR) algorithm is capable of registering images with smaller overlapping regions and thus it improves robustness. To further improve the processing speed and adaptability, we use selected regions during the multi-resolution registration process. That is, after input images are registered at low resolution, we extract appropriate sub-images from the input images and perform local registration only on the sub-images. The corresponding algorithm is referred as multi-resolution image registration with selected regions (MRIR-SR), which uses smaller regions in fine level registration and therefore can achieve higher processing speed, especially when the image resolution is large. In

addition, although the underlining ICA algorithm can handle homographic transformation only, MRIR-SR can be used to align planar regions in arbitrary scene, making it more versatile for motion estimation applications.

3.1 The Inverse Compositional Algorithm

The proposed GPU-based implementation is based on the ICA algorithm. It estimates the transformation matrix between two input images by iteratively minimizing the intensity difference between them.

Any two images of a planar surface or two images of arbitrary scene taken at the same view point are linked by homographic transformation. Given two images to be registered, we refer one of the images as the template Image ($\boldsymbol{T}$), and the second image as the input image ($\boldsymbol{I}$), the coordinate from the template image can be warped to the coordinate system of the input image by a matrix, i.e.,

$$\boldsymbol{x} = W(\boldsymbol{i}) = \boldsymbol{W} \cdot \boldsymbol{i} \tag{3.1}$$

where $\boldsymbol{i} = [i, j, 1]^T$ is the homogeneous coordinates of the pixel in $\boldsymbol{T}$ and $\boldsymbol{x} = [x, y, 1]^T$ is the homogeneous coordinates of the corresponding pixel in $\boldsymbol{I}$. $\boldsymbol{W}$ is a 3×3 matrix with 8 unknown parameters which can be used for modeling all in-plane projective transformations, i.e.,

$$\boldsymbol{W} = \begin{bmatrix} 1 + p_0 & p_1 & p_2 \\ p_3 & 1 + p_4 & p_5 \\ p_6 & p_7 & 1 \end{bmatrix} \tag{3.2}$$

The result of multiplying $\boldsymbol{W} \cdot \boldsymbol{i}$ must be normalized in order to obtain the Cartesian, i.e.,

$$x = \frac{(1+p_0)i + p_1 j + p_2}{p_6 i + p_7 j + 1} \; and \; y = \frac{p_3 i + (1+p_4) j + p_5}{p_6 i + p_7 j + 1} \tag{3.3}$$

Here, we use a vector to refer the eight unknown parameters, $\boldsymbol{p} = \langle p_0, p_1, p_2, p_3, p_4, p_5, p_6, p_7 \rangle$. Let $\boldsymbol{W(i;p)}$ denote the warp with the parameters in vector $\boldsymbol{p}$. The ICA algorithm estimates the vector $\boldsymbol{p}$ through iteratively performing:

$$\Delta \boldsymbol{p} = \boldsymbol{H^{-1}} \sum_{\boldsymbol{i}} \left[\boldsymbol{\nabla T_i} \frac{\boldsymbol{\partial W}}{\boldsymbol{\partial p}} \right]^T [\boldsymbol{I(W(i;p)) - T(i)}] \tag{3.4}$$

and,

$$\boldsymbol{W} \leftarrow \Delta \boldsymbol{W^{-1}} \boldsymbol{W} \tag{3.5}$$

where $\Delta \boldsymbol{W}$ is 3×3 matrix. It is built by replacing all the p_n in Equation 3.2 with Δp_n calculated in Equation 3.4. The algorithm converges when all elements in $\Delta \boldsymbol{p_n}$ are equal to zero or almost zero in a practical implementation. The $\boldsymbol{H}$ above is called Hessian matrix, which is an 8×8 matrix calculated by,

$$\boldsymbol{H} = \sum_{\boldsymbol{i}} \left[\boldsymbol{\nabla T_i} \frac{\boldsymbol{\partial W}}{\boldsymbol{\partial p}} \right]^T [\boldsymbol{\nabla T_i} \frac{\boldsymbol{\partial W}}{\boldsymbol{\partial p}}] \tag{3.6}$$

$\boldsymbol{\nabla T_i} \frac{\partial \boldsymbol{W}}{\partial \boldsymbol{p}}$ is called the steepest descent image (SDI) and it is evaluated where all the parameters for the warp are equal to zero. According to Equation 3.3,

$$\frac{\partial \boldsymbol{W}}{\partial \boldsymbol{p}} = \begin{bmatrix} i & j & 1 & 0 & 0 & 0 & -xi & -xj \\ 0 & 0 & 0 & i & j & 1 & -yi & -yj \end{bmatrix} \tag{3.7}$$

If we refer the gradient of the template image as $\nabla \boldsymbol{T_i} = [\nabla T_i \quad \nabla T_j]^T$ then,

$$\nabla \boldsymbol{T_i} \frac{\partial \boldsymbol{W}}{\partial \boldsymbol{p}}$$

$$= \begin{bmatrix} \nabla T_i i & \nabla T_i j & \nabla T_i & \nabla T_j i & \nabla T_j j & \nabla T_j & (-\nabla T_i i^2 - \nabla T_j ij) & (-\nabla T_i ij - \nabla T_j j^2) \end{bmatrix} \tag{3.8}$$

Since the Hessian matrix and SDI are independent on the warp parameters $\boldsymbol{p}$, they are constant across iterations. We can pre-evaluate them and reuse them during the iterative process.

Notice that, different from the conventional image registration scenario which warps input image to template image, the ICA estimates the warp matrix that warps template image to input image to avoid re-evaluating the Hessian. In the remainder of this thesis, we keep this convention that all the illustrations are based on warping template image to input.

3.2 Sequential Inverse Compositional Algorithm

The sequential implementation of ICA is by simply following **Algorithm 3-1.** We arrange the procedure of the algorithm into 3 steps.

Preprocessing

1 for each pixel $\boldsymbol{i}(i,j)$ in the template image $\boldsymbol{T}$ {

2 calculate ∇T_i

3 calculate $\nabla \boldsymbol{T}_i \frac{\partial \boldsymbol{W}}{\partial \boldsymbol{p}}$

4 $\boldsymbol{H} \mathrel{+}= \left(\nabla \boldsymbol{T}_i \frac{\partial \boldsymbol{W}}{\partial \boldsymbol{p}}\right)^T \cdot \nabla \boldsymbol{T}_i \frac{\partial \boldsymbol{W}}{\partial \boldsymbol{p}}$ }

Local parameters accumulation

5 for each pixel $\boldsymbol{i}(i,j)$ {

6 $diff = I(\boldsymbol{W}(\boldsymbol{i};\boldsymbol{p})) - T(\boldsymbol{i})$

7 $\Delta \boldsymbol{p} \mathrel{+}= diff \times \nabla \boldsymbol{T}_i \frac{\partial \boldsymbol{W}}{\partial \boldsymbol{p}}$ }

Warp updating

8 $\Delta \boldsymbol{W} = ParametersToWarpMatrix(\boldsymbol{H}^{-1}\Delta \boldsymbol{p})$

9 $\boldsymbol{W} = \Delta \boldsymbol{W}^{-1}\boldsymbol{W}$

10 If $CheckConvergeAndStopCondition(\ ) == false$ goto setp 5

Algorithm 3-1 Pseudo-code of the ICA algorithm

- **Preprocessing**

This step (line 1 to 4) traverses the whole template image to pre-calculate the image gradient $\nabla \boldsymbol{T}$, eight SDIs, and the Hessian matrix. For a given registration task, this step only needs to be performed once.

- **Local parameters accumulation**

Using the current warp matrix, this step (line 5 to line 7) computes the intensity difference between the template image and the warped input image. The difference,

scaled by the SDI, is then used to compute the update vector Δp_s of the local warp parameters.

- **Warp updating**

In this step (line 8 to 10), we update the current warp matrix using the update vector Δp_s. We then check whether the stopping condition is met. If it is, the current warp matrix sent to output and the process terminates. Otherwise the process goes back to the local parameter accumulation step.

3.3 Implementing ICA for GPU Processing

GPU programming uses stream processing model, where all the computation are put into kernel functions, which are applied to the data stream.

Figure 3-1 illustrates the whole procedure of the proposed system.

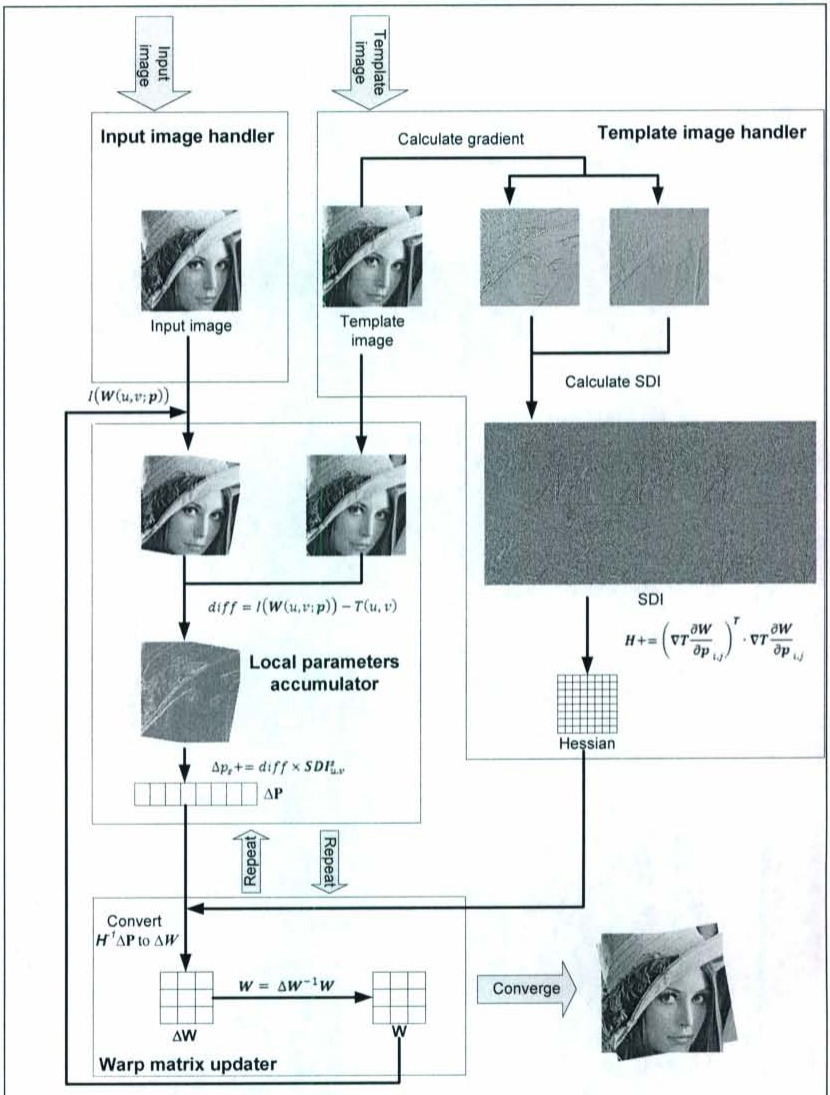


Figure 3-1 GPU-based implementation of ICA algorithm

To simplify code maintenance and maximize code reusability, we arrange the kernels into four modules illustrated by **Figure 3-2**.

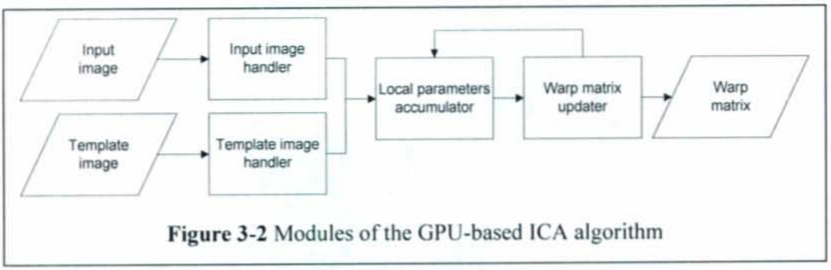


Figure 3-2 Modules of the GPU-based ICA algorithm

- Input image handler

  The input image handler needs to perform two basic tasks. The first one is transferring images from system memory to graphics memory, whereas the second task is converting the transferred images to the layout optimized for GPU addressing. The converting task is paralleled by the conversion kernel which runs the same number of threads as the image size with each thread converting one pixel.

- Template image handler

  Besides the two basic tasks the same as the input image handler, the template image handler is in charge of the preprocessing described in lines 1 to 4, **Algorithm 3-1**. The preprocessing is parallel by both the SDI calculation kernel, which computes SDI for each pixel, and the Hessian accumulation kernel, which computes the Hessian matrix form the SDI.

- Local parameters accumulator

- This module conducts the local parameter accumulation. The local parameter accumulating kernel is introduced to calculate the warped image difference and further accumulate the local parameters of the warp matrix. Warp matrix updater

  This single kernel module updates the warp matrix with local parameters. If the algorithm converges, it output the updated warp matrix as the final result. Otherwise, the updated one is passed back to local parameters accumulator to continue the iteration.

3.4 Multi-resolution Image Registration

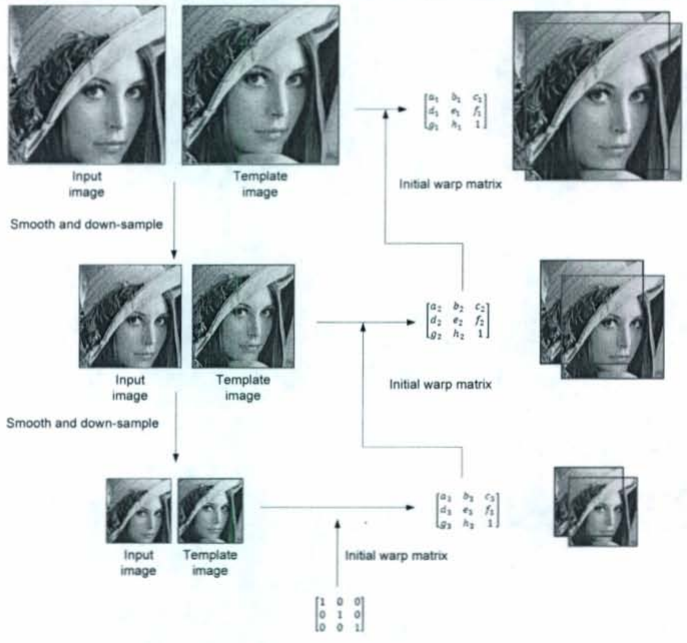


Figure 3-3 Multi-resolution image registration

Intuitively, the ICA algorithm works by comparing the intensity difference between the warped input image and the template image and updating the warp parameters to minimize the total intensity difference. The parameter update is guided by image gradient, which provides information about how each parameter affects intensity difference through the eight SDIs. When the two images contain detailed textures and are initially poorly aligned, the ICA algorithm may fail to register the two images together since the image gradient may guide the parameters toward local minimums.

To overcome this problem and improve the robustness of the registration, we here apply the coarse-to-fine processing scheme. As illustrated by **Figure** 3-3, we first build the Gaussian pyramid by iteratively blurring and down-sampling the high resolution input images to low resolution images. Then the image registration is performed from the lowest resolution level to the highest resolution level. The estimated warp matrix of each level is transformed to the coordinate system of higher resolution image set and is used as the initial solution. For example, assuming the warping matrix between the two images found at a coarse level is $\boldsymbol{W}_n$, the initial solution $\boldsymbol{W}_{n-1}{}'$ at the finer level is calculated using:

$$\boldsymbol{W}'_{n-1} = \begin{bmatrix} 1/k & 0 & 0 \\ 0 & 1/k & 0 \\ 0 & 0 & 1 \end{bmatrix}^{-1} \boldsymbol{W}_n \begin{bmatrix} 1/k & 0 & 0 \\ 0 & 1/k & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where k is the down-sampling ration between the finer level and the coarse level.

The down-sampled process removes the detail and the noise which can potentially mislead the registration process. The lowest resolution images contain only the largest-scaled features that allow robust registration even when the two images are poorly aligned. Using the warping matrix calculated at lower resolution to set the initial warping parameters, the higher resolution registrations only need to tune the warping parameters based on the detail information available.

Figure 3-4 shows the results using registering the same two images using standard IR and MRIR. While the standard approach fails, the MRIR approach still yields good result.

Standard image registraion

Multi-resolution image registration

Figure 3-4 Comparison between standard IR and MRIR using the same images

3.5 Multi-resolution Image Registration with Select Regions (MRIR-SR)

While using the coarse-to-fine scheme helps to improve the robustness of the registration, it also introduces additional computation since the registration needs to be performed at multiple resolution levels. To achieve faster processing speed, here we proposed to use selected regions, instead of the whole image, to perform registration at fine levels. For example, after registering the two images at the coarsest level, where the resolution is $n \times n$, we need to move to the finer level with resolution of $kn \times kn$, where k is the down-sampling ratio between levels. Instead of using all $kn \times kn$ pixels, we chose a region $n \times n$ in size to perform the next registration. Since the size of the regions used for registration is kept as a constant across the coarse-to-fine scale, the computation time can be saved dramatically at the finer levels.

Another benefit of using selected regions for registration is the increase of robustness for outliers. In many real-world situations, only part of the scene can be registered by homographic transformation. For example, a scene may contain both planar and non-planar objects, where only the planar object portion can be registered accurately. Or, a scene contains both far away background and nearby foreground, where the foreground potion is affected by camera movement and hence cannot be registered. Under these cases, selecting the proper regions at finer level registration can successfully register the images.

In the rest of this section, we will explain how to select the proper region for registration and how to adjust the transformation matrix when moving from one level to the next level.

3.5.1 Region Selection

Assume two images are already registered at a coarser level, where image resolution is $n \times n$. Before we perform registration at the finer level, we want to select a region of size ${}^{n}/_{k} \times {}^{n}/_{k}$, which corresponds to an $n \times n$ area at the finer level. The two criteria for selecting the region are:

- The region needs to contain sufficient detail to facilitate registration. That is, the gradient magnitude should be high within the region.
- The existing registration for the region should be successful. That is, the registration error within the region should be low.

To find the best region based on the above two criteria, we first calculate the gradient-to-error ratio for each pixel using the following equation:

$$\boldsymbol{R_i} = \frac{\|\nabla \mathbf{T}(\boldsymbol{i})\|}{\varepsilon + |\boldsymbol{I}(\boldsymbol{W}(\boldsymbol{i})) - \boldsymbol{T}(\boldsymbol{i})|} \tag{3.9}$$

where $\|\nabla \mathbf{T}\|$ is the gradient magnitude of the coarse template image, $\boldsymbol{W}$ is the warp estimated by the coarse images, $\boldsymbol{I}(\boldsymbol{W}(\boldsymbol{i}))$ indicates the intensity value in the coordinate system of image $\boldsymbol{I}$ with coordinate $W(i)$, and $\boldsymbol{I}(\boldsymbol{W}(\boldsymbol{i})) - \boldsymbol{T}(\boldsymbol{i})$ is the different image between the coarse input image transformed by $\boldsymbol{W}$ and the coarse template image. ε is a small number to avoid dividing by zero.

After the per pixel gradient-to-error ratio is calculated, we aggregate the values within local ${}^{n}/_{k} \times {}^{n}/_{k}$ windows using a box filter. That is:

$$\boldsymbol{A}(x,y) = \sum_{j=y}^{y+\frac{n}{k}} \sum_{i=x}^{x+\frac{n}{k}} \boldsymbol{R}(i,j), x \text{ and } y \text{ are integer numbers form } 1 \text{ to } N-n \tag{3.10}$$

Because every element in aggregated ratio $\boldsymbol{A}$ represents the summation of a ${}^{n}/_{k} \times {}^{n}/_{k}$ sub-region in $\boldsymbol{R}$, we can simply find the maximum element in $\boldsymbol{A}$ to locate the region that gives the highest overall gradient-to-error ratio. For example, the top-left (p, q) of the desired region in the fine template image is calculated using:

$$(p,q) = k \times \underset{(x,y)}{arg\max} |\boldsymbol{A}(x,y)| \tag{3.11}$$

The coefficient k is used to convert the coordinate to the fine level.

3.5.2 Locating the Corresponding Region in the Input Image

The region selection process describe above is conducted in the image space of the template image. To find the corresponding region in the input image, we can simply transform the coordinate (a, b) to the input image using the warping matrix obtain at the coarse level. That is:

$$(a', b') = W'(a, b) \tag{3.12}$$

The two sets of coordinates (a', b') and (a, b) are then used to extract two regions for registration at the next level from the input image and the template image, respectively. The initial warp matrix $\boldsymbol{W}'$ between the extracted regions is computed by the warping matrix $\boldsymbol{W}$ previously calculated using the coarse level. Assuming the origins of the coordinate system of the images are the top-left corner and we keep this convention for the rest of the thesis, $\boldsymbol{W}'$ can be calculated by,

$$\boldsymbol{W}' = \left(\begin{bmatrix} 1/k & 0 & 0 \\ 0 & 1/k & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & p' \\ 0 & 1 & q' \\ 0 & 0 & 1 \end{bmatrix} \right)^{-1} \boldsymbol{W} \begin{bmatrix} 1/k & 0 & 0 \\ 0 & 1/k & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & p \\ 0 & 1 & q \\ 0 & 0 & 1 \end{bmatrix} \tag{3.13}$$

where (p', q') is the position of the top-left corner of the extracted input image in the fine input image. To calculate (p', q'), we first locate the center of the selected input region in the coarse input image by,

$$\begin{bmatrix} a \\ b \\ 1 \end{bmatrix} = \boldsymbol{W}_c \begin{bmatrix} (p + \frac{n}{2})/k \\ (q + \frac{n}{2})/k \\ 1 \end{bmatrix} \tag{3.14}$$

then, we compute (p', q') in the fine input image by,

$$\begin{cases} p' = ka - \frac{n}{2} \\ q' = kb - \frac{n}{2} \end{cases} \tag{3.15}$$

3.5.3 Implementation

The implementation of the MRIR-SR is illustrated by **Figure 3-5**. Besides the image registration components explained in Section 3.3 , the system also consists of other four kernels. The registration procedure begins at the down-sampling by a kernel called down-sampler. Then, a region selection kernel calculates the difference image with the warp matrix, the gradient magnitude of the down-sampled template image and further the gradient-to-error ratio. After that, the accumulating kernel aggregates the ratios in parallel and finally the maximum searching kernel locates the maximum point in the aggregation result. After mapping the located point to the original image, two regions are extracted from the input and template images, which are send to the image registration module

again for registration at finer level. The output warp matrix at finer level is transformed back to the coordinate system of the original image as the final result.

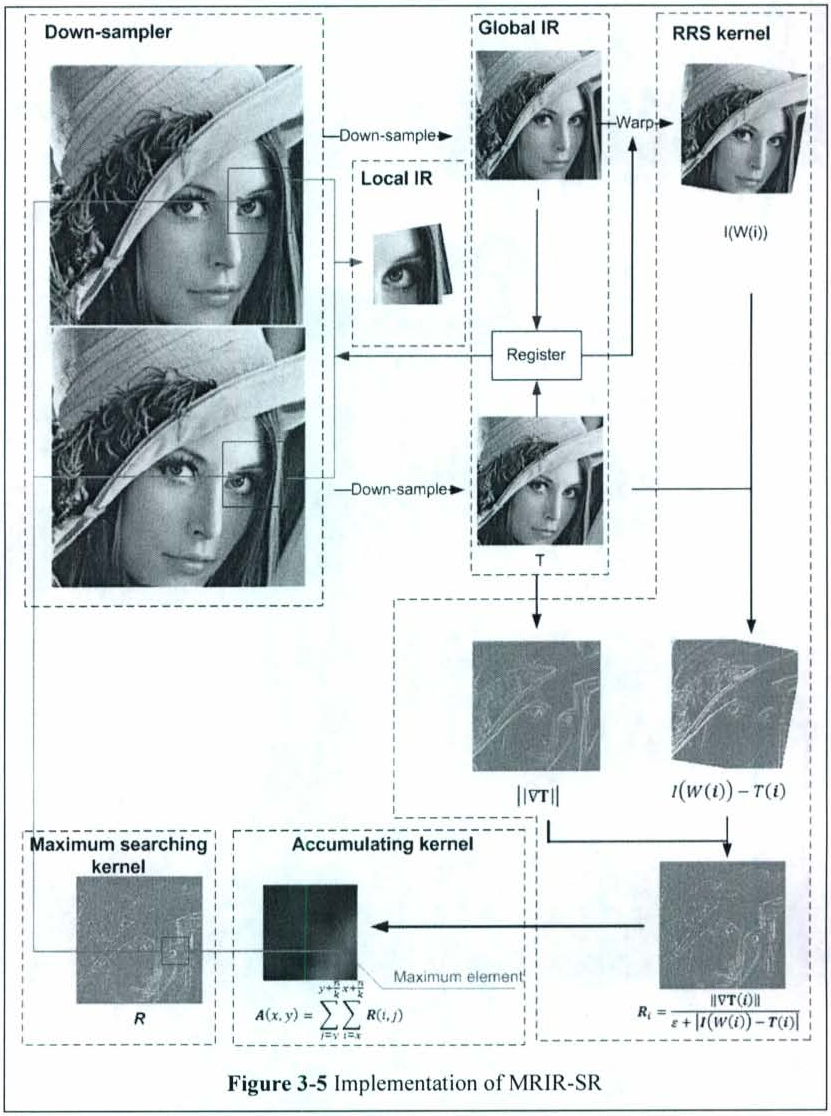


Figure 3-5 Implementation of MRIR-SR

Figure 3-6 illustrates the results using different image registration approaches to register two same images which have relatively small overlapped area. While both the standard IR and the MRIR fail, the MRIR-SR approach still outputs a good result.

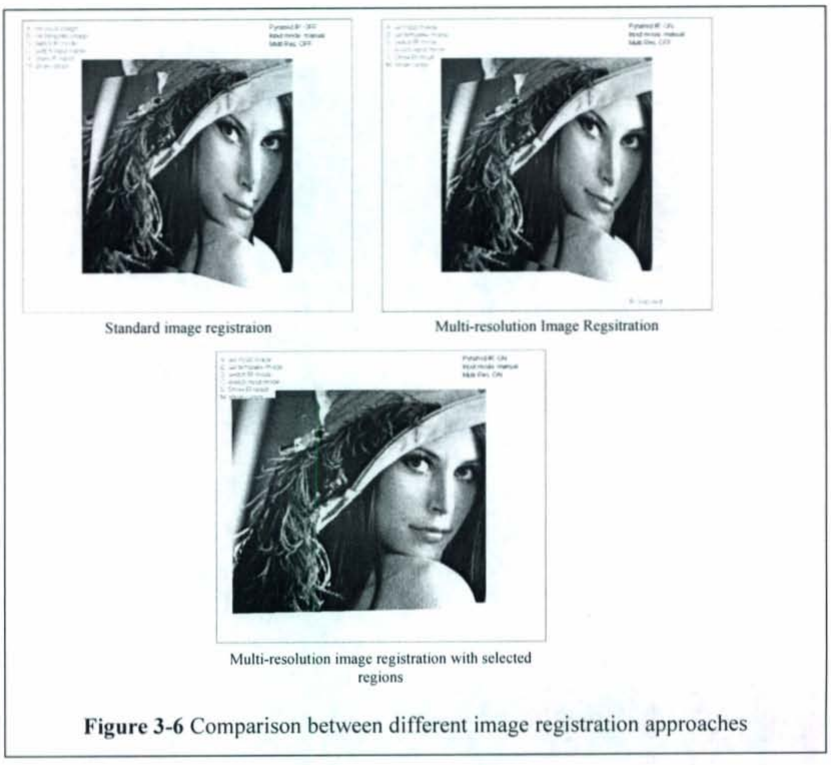


Figure 3-6 Comparison between different image registration approaches

3.6 Discussion

In this Chapter, we discussed a parallel image registration approach and two of its variants. The straightforward IR approach implements the ICA algorithm in parallel using

stream processing model. This allows us to take advantage of the processing power of modern programmable GPUs, which are much faster than the CPUs.

To improve the registration performance, we propose two multi-resolution variants. The MRIR approach helps to improve the robustness of the system through incorporating the coarse-to-fine scheme, but at the expense of additional computational cost. The MRIR-SR approach uses selected region for fine level registration, and hence dramatically reduces the computation cost when the input image resolution is high.

The two approaches have their own advantages under different scenarios. When the whole scene can be registered using homographic transformation and the processing power is sufficient, we recommend using the MRIR approach, which can align the input images accurately using the warping matrix obtained. In contrast, the MRIR-SR approach evaluates the final warping matrix using local region only, which could lead to misalignment along the boundary that is far from the selected region. However, MRIR-SR has the advantage of higher processing speed and being more robust to less informative regions, i.e., regions cannot be registered by homographic transformation.

Chapter 4 Optimization of the Image Registration Module on CUDA

The previous chapter discussed the parallel implementation of image registration and its two multi-resolution variants. Because the first one is the building-block for the two variants, we would like to optimize its performance. In this chapter, we will discuss how to efficiently optimize the image registration module on CUDA. This includes, for example, the benefit of implementing sequential tasks on the GPU in a way to avoid high latency I/O transportation, applying parallel reduction to accelerate inter-thread summation, using the sequential-then-parallel processing to fully utilize the stream processing units, etc. We also discuss the optimized memory accessing pattern for CUDA devices and how to use various buffering approaches to increase memory throughput. The evaluation section shows the effectiveness of the discussed optimization techniques.

4.1 Introduction of CUDA

The CUDA programming model is based on the hardware features of the CUDA architecture. CUDA devices can run thousands of threads in parallel and the threads are organized in a hierarchy, illustrated by **Figure 4-1**.

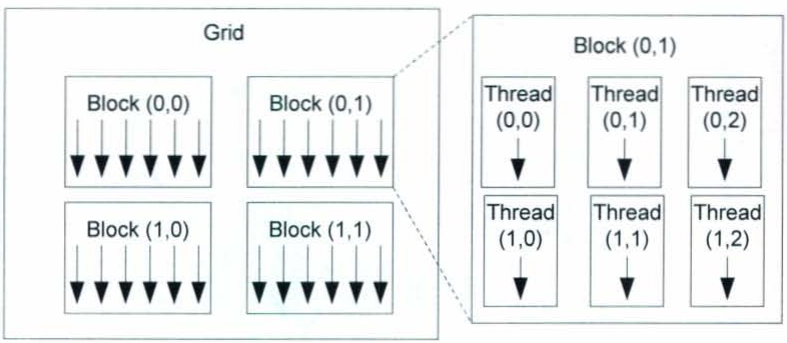


Figure 4-1 Hierarchy of CUDA threads

Threads are first grouped into blocks and blocks are further grouped into a grid. A parallel function, or a kernel in term of GPU programming, has only one grid. The blocks of the grid are distributed to the cluster of the processing units called multiprocessors at run time. The multiprocessors schedules and executes the allocated blocks. The threads in a block can use the fast on-chip shared memory to accelerate inter-threads communications.

Besides the shared memory, CUDA devices also provide other storage devices to speed up memory access or ease programming, such as texture memory for caching special random accessing, constant memory for accelerating global variable fetching, and local memory for holding temporary in-thread data.

The proposed optimization techniques are tuned for two series of CUDA devices. One (including G80 and G90) is the wide-used devices with Compute Capability 1.x (C1.x)

architecture. Another (G100) is the new released devices based on Compute Capability 2.0 (C2.0) architecture. The C2.0 devices are designed more sophisticated for general-purpose computing. They provide larger instruction throughput, faster memory accessing speed and more registers and shared memory per block. The device memory accessing is also accelerated by L1 and L2 caches which are not provided in C1.x devices.

4.2 Implementation

To make full use of CUDA devices and reduce unnecessary high-latency CPU-GPU communications, we implemented most tasks on GPUs. The parallel tasks are assigned to 6 kernels labeled by Kernel 1 to Kernel 6. The rest sequential tasks are encapsulated by the CPU functions.

The functionalities of those kernels and functions and the relationships between them are shown by **Figure 4-2**.

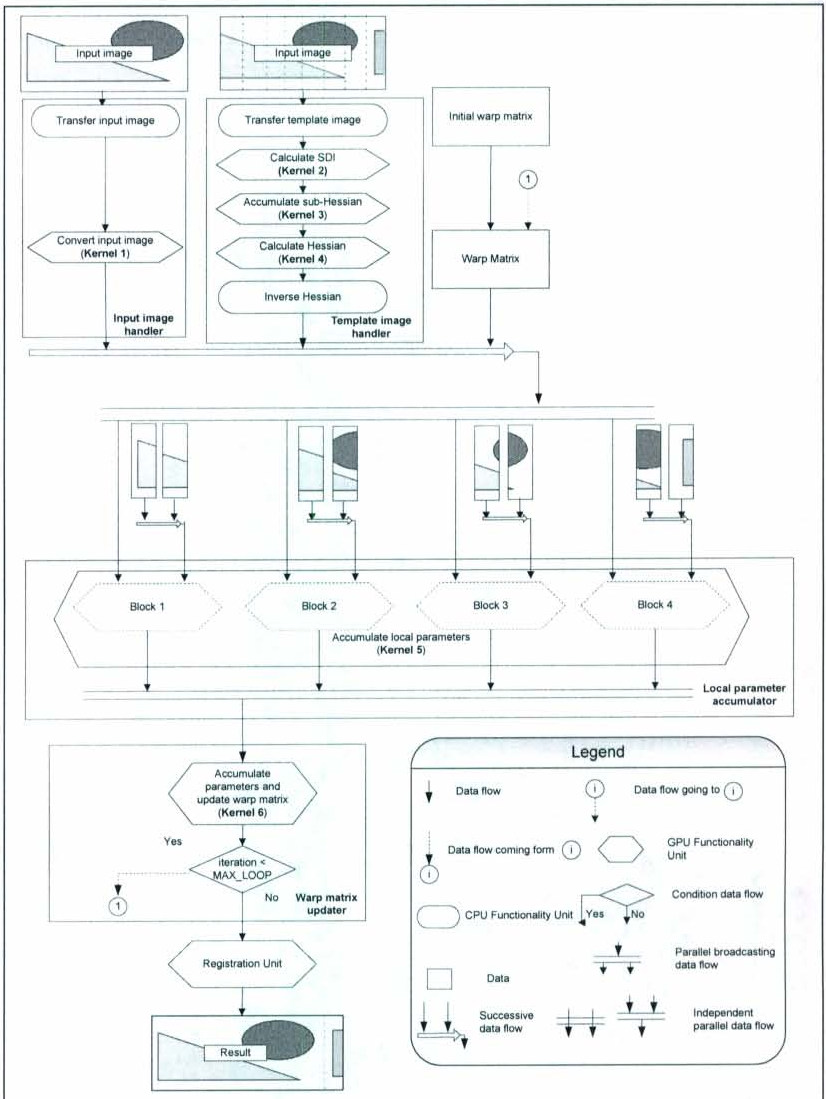


Figure 4-2 Structure of the image registration module

4.3 Optimization Techniques

4.3.1 Balancing Workload Distribution

When handling a single thread, the performance of the CPU is much higher than that of an individual GPU processor. It is therefore beneficial to distribute sequential tasks to the CPU and parallel tasks to GPU for maximum instruction throughput. However, if a sequential task appears in the middle of GPU executions, intermediate results will need to be transferred between CPU and GPU through graphics I/O bus. Those I/O transportations have high latency and dramatically reduce the performance of the pipeline. In this case, we should measure whether the single thread speed advantage of CPU outweighs the high I/Os cost. If the sequential tasks are not complicated, we may run them with several threads in GPU to avoid I/O communications.

For example, the tasks of the warp updater module are sequential-friendly. They are either serial executions or small matrix operations which have little parallelism. The CPU implementation tends to yield better performance. However, the GPU version is twice as fast as the CPU version, since it removes the needs of frequently transporting the local parameters from GPU to CPU and the resulted updated warp matrix back to GPU.

4.3.2 Parallel Reduction

The ICA algorithm requires inter-thread summations (line 4 and line 7, **Algorithm 3-1**, Section 3.3). Most current GPUs support atomic function for this kind of tasks. However, it is not the most efficient approach for large summation. Currently, the speed of atomic function is several times slower than coalesced operations. It will dramatically reduce the instruction throughput when massive used in parallel. Moreover, only current advanced

graphic cards provide atomic operation writing to on-chip storage to cache the intermediate result. If atomic operation is not supported, the results have to be written to off-chip graphics memory which has hundreds of times longer latency. Therefore, we would like to find another method to speed up the summation.

The parallel reduction [32] is an efficient choice for inter-thread summation. It uses the fastest coalesced numeric operations. After loading inputs from graphics memory, it performs using the low latency on-chip storage. Thus, it provides high instruction and memory throughput. Moreover, parallel reduction only uses the standard features supported by almost all CUDA-compatible graphics cards. There is less compatibility issues.

To make full use of the parallel reduction, we should allocate a maximum number of threads per block based on the available resources (the on-chip memory to cache the intermediate results), so that as many elements as possible can be reduced in a single block as possible. Appendix A.3 illustrates a parallel reduction building block modified based on the fully optimized version presented in [32]. It is capable of reducing 512 elements which is the maximum number of threads per block for NVIDIA GPUs with hardware compute capability 1.X. If the number of elements need to be reduced is more than the resource available, we can assign multiple blocks into a two level hierarchy illustrated by **Figure 4-3**. To implement the hierarchy, one kernel uses multiple blocks to perform the first level reduction and output the intermediate result to the graphic memory buffer and another kernel use a single block to reduce the intermediate result.

Our test shows that the discussed parallel reduction is generally 40% faster than the atomic function. Even the atomic function used is the optimized version presented in Appendix A.2 .

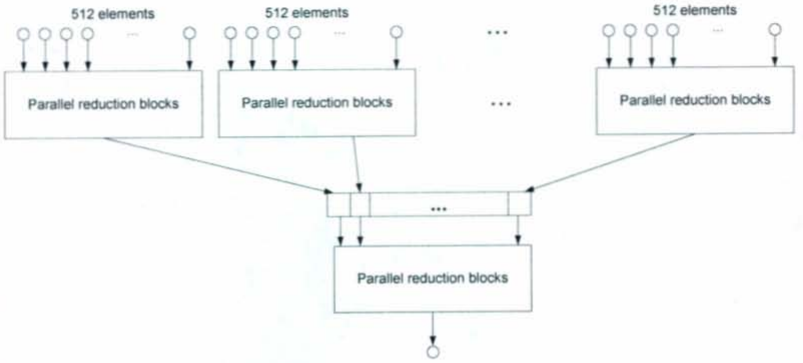


Figure 4-3 Parallel reduction hierarchy built by the building block which is capable of reducing 512 elements

4.3.3 Sequential-Then-Parallel Processing

A common strategy to parallel execution is allocating as many threads as necessary according to the input data and letting GPUs to automatically scheduling their executions. This is an efficient solution when the number of threads needed is not too much larger than available processing units. Otherwise, GPU will schedule a long execution queue, causing extra overhead for the complex context switching. In addition, this strategy is not a good choice for the programs which need to aggregate the outputs of different threads. Because the threads are not active during the whole execution, the active threads cannot cache intermediate results in in-thread memory for the queuing threads. They have to be

written to the graphics memory. Thus, the performance of those programs is highly depended on the graphics memory bandwidth which will limit the instruction throughput.

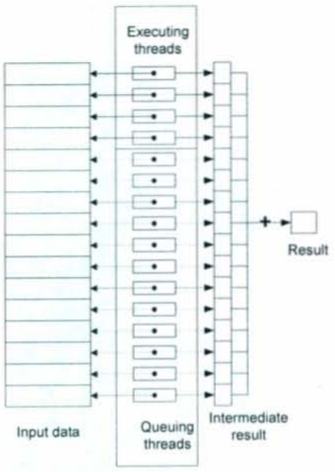


Figure 4-4 Automatically scheduled strategy for aggregating kernels

Figure 4-4 illustrates the automatically queuing case. If a GPU is capable of running four parallel threads and the program requires 16 threads, 12 of them will be queued. The GPU has to load the contexts of the clusters of four threads at least four times. Moreover, it outputs intermediate result 16 times. Those outputs are graphics memory writings whose latency is depended on the bandwidth of the graphics memory, which is usually hundreds of times slower than registers or shared memory.

We would like to apply another strategy to avoid thread scheduling and to enable in-thread caching to make the kernel less depended on the graphic memory bandwidth. The

solution is here referred as sequential-then-parallel processing. To apply the sequential-then-parallel processing, the program runs the maximum number of active threads and each thread sequentially handles multiple tasks by looping. That is, for the same scenario in **Figure 4-4**, the program runs four threads and each thread loops four times to process the whole data. Thus, the GPU only needs to load the context of threads once. If the program needs aggregating result, intermediate result could be cached in-thread using register or shared-memory. In all, it only performs four graphics memory writings after looping.

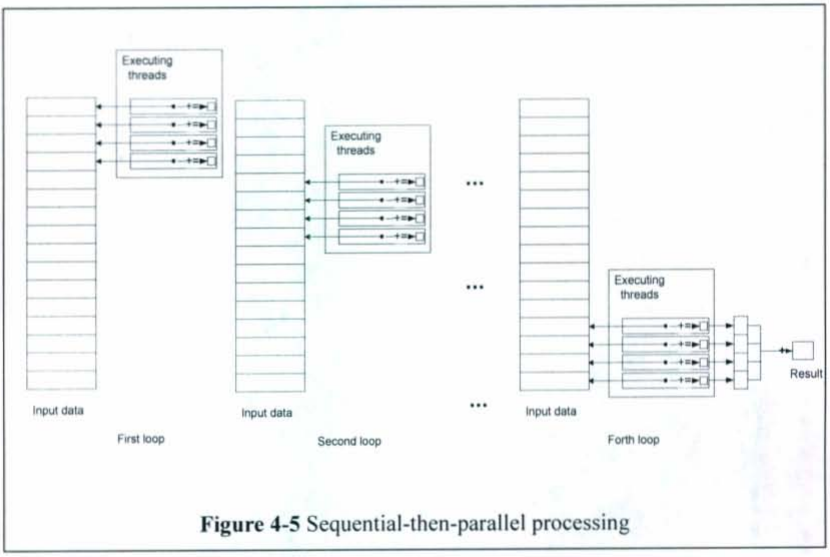


Figure 4-5 Sequential-then-parallel processing

In practice, such as processing large images, applications usually require massive threads. The hybrid processing strategy will dramatically reduce the complexity of thread scheduling and makes aggregating kernels less depend on graphic memory bandwidth.

We apply the hybrid processing to two tasks – the Hessian calculation (Kernel 3 and Kernel 4) and the local parameter accumulator (Kernel 5 and Kernel 6). Kernel 3 and Kernel 5 are in charge of reducing intermediate results. Kernel 4 and Kernel 6 then aggregate the intermediate results. Although the maximum active blocks can be allocated for kernel 3 and kernel 5 are relative small (usually less than 64) in current CUDA devices because of the hardware limitation, the chunk of intermediate results is still quite large because their elements are matrices and vectors. We would like to parallelize the operation of the summation of the elements.

To sum the output of Kernel 3, which is a chunk of 8×8 matrices, we use 64 threads in Kernel 4. Each thread is in charge of sequentially summing together the intermediate result for one element of the matrix.

For kernel 5, whose output is a chunk of vectors with 8 elements, we choose a more sophisticated parallel reduction method. It uses idle threads of the standard parallel reduction to reduce multiple dimensions at the same time. Compared to adding vectors by looping standard parallel reduction for each element, it requires much less adding operations. Appendix A.4 shows its CUDA implementations.

The test shows that there is up to 43% performance gain after applying the hybrid processing.

4.3.3.1Tuning Distribution of Blocks of Sequential-Then-Parallel Kernel

Tuning the sequential-then-parallel processing introduces two steps. The first step is to decide the maximum active blocks per multiprocessor (MAB). Because we use the

maximum threads per block for reduction, the MAB is only decided by the shared memory and register usage. These two factors can be monitored by the compiler output. With the maximum thread per block, shared memory and register usage, we could use CUDA profiler, a tool shipped with CUDA SDK, or reference to the specification of CUDA hardware to calculate the MAB.

The second step is to fully occupy each multiprocessor with the blocks in the number of MAB. Assuming there are total M multiprocessors in the current device, we can simply assign $M \times MAB$ blocks because CUDA evenly distributes the blocks to each multiprocessor.

The experiment shows the peak of performance appears under the discussed workload distribution setting. Compared to less-occupied and over-occupied case, it is generally 10% faster.

4.3.4 Memory Access Pattern

To maximize the memory throughput of the CUDA device, we optimize the data structure and memory accessing pattern for the images, SDI, and Hessian matrix.

4.3.4.1 Coalesced Memory Layout

Because the 32-bit memory layout is the fastest data structure for CUDA to perform coalesced access, we organize the 2D data in this way.

The data includes input image, template image and SDI. Each pixel or element of those data is represented by a single 32-bit float variable and all the pixels are stored in pitched row major order, i.e., rows are stored one after the other in the linear space. The reason

why we use the redundant 32-bit variable is that it is the native length of the Arithmetic Logic Units and Float Point Units of CUDA devices, which yields the fastest arithmetic operation speed.

4.3.4.2 Texture Memory Caching

There could be a massive un-coalesced accessing scenario in the parallel inverse compositional algorithm. It accesses the image with warped coordinates. Because the warp matrix is updated iteratively, it is impossible to organize the template image layout to a coalesced way. Therefore, we use the texture memory to cache image accessing. The speed up is highly depends on the practical condition. In general, it yields up to 5% better result.

4.3.4.3 Constant Memory Broadcasting

The constant memory is ideal for broadcasting global constant for every thread because its content is cached and therefore faster than the un-cached device memory. In the proposed system we use the constant memory for broadcasting the warp matrix to every thread to perform the image transformation. It is 10.5% faster compared with passing warp matrix by device memory.

4.4 Bottleneck Optimization

If we monitor the execution of the parallel ICA implementation, Kernel 5 takes more than 90% execution times. It is obvious the bottleneck we would like to further optimize.

```
__shared__ float s_dp[8];//ouptput buffer
...
//accumulating loop
for(int c = 0; c < M; ++c){
      ...
      for(int s = 0; s < 8; ++s){
            float dp = diff == 0 ? 0 : diff * SDI(i, j, s);
            float sum = reduce512(s_sum, dp);
            if(threadIdx.x  == 0)s_dp[s] += sum;
      }
      ...
} //end of the accumulating loop
...
```

Figure 4-6 Local array buffered parallel reduction approach

The complete implementation of Kernel 5 is illustrated in CUDA-liked pseudo-code in Appendix A.5 . Here we extract the instructions directly related to the optimization and illustrate them in **Figure 4-6**. Notice the bold lines of the algorithm. The number of parallel reductions needed is $\mathbf{8} \times \boldsymbol{M}$. If we can accumulate the local warp parameter (*dp*, labeled in red in the pseudo-code) in a buffer and reduce the accumulated result outside the accumulating loop, then we only need call the parallel reduction **8** times. This will save a significant amount of instruction cycles. Hence, the key to optimize Kernel 5 is to find a way to buffer *dp*.

4.4.1 Local Array Buffered Approach

From the programming perspective, the most straightforward approach to buffer *dp* is by local array. It is illustrated by **Figure 4-7**.

```
__shared__ float s_dp[8];//ouptput buffer
...
float dpBuff[8] = {0};//local array for buffering dp accumulation
//accumulating loop
for(int c = 0; c < M; ++c){
      ...
      for(int s = 0; s < 8; ++s)
      {
            float dp = diff == 0 ? 0 : diff * SDI(i, j, s);
            dpBuff[s] += dp;
      }
      ...
} //end of the accumulating loop
//reduce the buffered result
for(int i = 0; i < 8; ++i){
      float dp = dpBuff[i];
      float sum = reduce512(s_sum, dp);
      if(threadIdx.x == 0) s_dp[i] = sum;
}...
```

Figure 4-7 Local array buffered approach

In this approach, every thread adds *dp* to the local array *dbBuff[]* and add together every elements of the array after the accumulating loop.

4.4.2 Partial Reduction Approach

Using local array to buffer *dp* is not efficient. Stated in Section 5.3.2.2, [34], "the local memory space resides in device memory, so local memory accesses have same high latency and low bandwidth as global memory". For each thread, there will be $8 \times M$ global memory writing in accumulating loop and 8 parallel reductions when reducing the buffered result. The cost of those global memory access may exceeds the saving of reduction instructions and results in performance drop. Actually, there is up to 10% performance drop after using this approach.

We would like to find faster buffering approaches. One possible solution is using shared memory but there is a limitation. In most current CUDA devices (C1.x devices), the available shared memory cannot hold the *dp* for all threads. For a block of 512 threads (the maximum threads per block to fully make use of the parallel reduction) and each thread need 8 float variables for the local parameters vector, the total shared memory usage per block is $512 \times 8 \times 4 = 16384$ bytes. It is the maximum available shared memory per block. Because some shared memory is pre-occupied for parameter passing, the practical usage is slightly smaller than 16KB and thus not enough for the complete buffering.

```
__shared__ float s_dp[8]; //ouptput buffer
__shared__ float s_sum[8][256]; //partial reduction buffer
__shared__ float s_psum[256]; //reductin buffer
...
//accumulating loop
for(int c = 0; c < M; ++c){
      ...
      for(int s = 0; s < 8; ++s){
            float dp = diff == 0 ? 0 : diff * SDI(i, j, s);
//partial reduce one step to decrease the shared memory usage
            s_psum[threadIdx.x] = dp; __syncthreads();
            if(threadIdx.x < 256)
                  {s_sum[s][threadIdx.x] += dp
                        + s_psum[threadIdx.x + 256];
            __syncthreads();}
...} //end of the accumulating loop

//reduce the buffered result
for(int i = 0; i < 8; ++i){
      //reduce 256 elments
      float sum = reduce256(s_sum[i]);
      if(threadIdx.x == 0)s_dp[i] = sum;}
...
```

Figure 4-8 Partial reduction approach

To address the problem, we reduce all the *dp* from 512 elements to 256 elements to decrease the shared memory usage. This approach is illustrated by **Figure 4-8**. The one step reduction shown in bold is referred as partial reduction. It reduces the 512 elements to 256 elements. The total buffer size is decreased to $256 \times 8 \times 4 = 4$KB which is much less than the available shared memory.

The test shows that this approach can accelerate the system up to 41%.

4.4.3 Unrolled Register Buffered Approach

```
__shared__ float s_dp[8]; //ouptput buffer
__shared__ float sum[512]; //reductin buffer
float s0 =0, s1 =0, s2 = 0, s3 = 0, s4 = 0, s5 = 0, s6 = 0, s7 = 0;
...
for(int c = 0; c < M; ++c){
      ...
//unroll the accumulating loop
#define DELTA_P(sdi, s) \
      {(sdi) += diff == 0.0f ? 0.0f : diff* SDI(i, j, s);}

      DELTA_P(s0, 0);DELTA_P(s1, 1);...;DELTA_P(s7, 7);
      ...}
//reduce the buffered result
#define REDUCE_TO(sdi, i) {\
      s_psum[threadIdx.x] = (sdi);\
      __syncthreads();\
      (sdi) = reduce512(sum, (sdi));\
      if(threadIdx.x == 0){ s_dp[i] = (sdi);}__syncthreads();}

REDUCE_TO(s0, 0); REDUCE_TO(s1, 1);...;REDUCE_TO(s7, 7);
...
```

Figure 4-9 Unrolled register cached approach

Another possible caching method is using registers which provides the fastest local accessing speed.

To enforce buffering variables residing in registers, we unroll the accumulating loop with hard coding. **Figure 4-9** illustrates this approach.

Variables *s0* to *s7* are used for buffering the local parameters. Macro *DELTA_P(sdi, s)* unrolls the loop. The CUDA compiler will use the register to reside the variables for the unrolled hard-coding. Macro *REDUCE_TO(sdi, i)* is used for simplifying the coding for reducing the buffered result.

Compared with the partial reduction approach, the unrolled register buffered approach requires no shared memory usage and removes the $8 \times M$ partial reductions. More important, accessing the register buffer should be faster than accessing the shared memory buffer. This version is proved to be the most efficient approach by experiment which can accelerate the system up to 80%.

4.5 Evaluation

The evaluation is to test out the effectiveness of the discussed optimization techniques. There are two sets of images to be evaluated – one with 256×256 resolution and the other with 512×512. They simulate the low workload and the high work load conditions separately.

Because the execution duration of the same kernel differs from times to times, we count the average execution time to compare the performance difference, where the execution time is measured using the number of frames can be processed in one second (FPS).

4.5.1 Evaluation of the General Optimization Techniques

The algorithm independent optimization techniques including simulating sequential execution in GPU, parallel reduction and hybrid processing are evaluated in this section.

Figure 4-10 illustrates the performance of the different versions of the proposed implementation.

GEFORECE GTX 480, Xeon E5440 2.53GHz CPU, 4GB Memory					
Image size	1	2	3	4	5
256 * 256	0.9	31.92	63.7	64.01	91.82
512 * 512	0.34	10.63	21.02	29.81	34.11

1: Sequential ICA

2: Parallel ICA using atomic functions and CPU warp matrix updater

3: Parallel ICA using atomic functions and GPU warp matrix updater

4: Parallel ICA using parallel reduction and GPU warp matrix updater

5: Parallel ICA using parallel reduction with hybrid processing strategy and GPU warp matrix updater

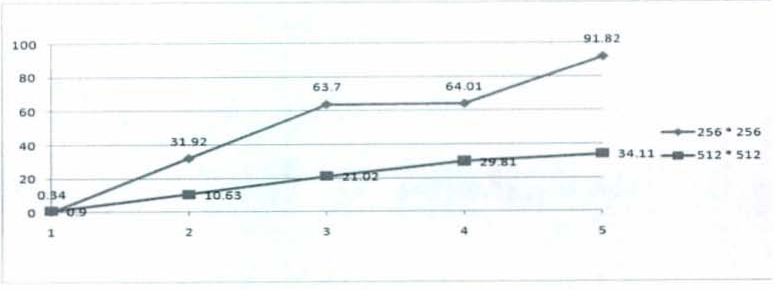


Figure 4-10 Evaluation of the general optimization techniques

Version 1 is the sequential ICA. Both the high workload and the low workload results are less than 1 FPS and hence cannot meet the real-time requirement.

From the comparison between Version 2 and version 3, we can clearly see that the GPU warp updater is much faster than the sequential one, even though this warp updater module itself runs faster on CPU. This is because the updater is called repetitively when updating warp matrix, the I/O cost for transporting intermediates between CPU and GPU overweighs the acceleration of CPU.

The comparison between version 3 and version 4 shows the power of parallel reduction. While the its advantage is not evident under low workload testing because it requires fewer instruction throughputs, the parallel reduction yield 40% (8 FPS) speed up for high workload testing.

The performance is further speeded up after applying hybrid processing strategy to improve Version 4 to Version 5. The low workload testing gains 43% (27.81FPS) speed up while 14% (4.3FPS) for the high workload test.

4.5.2 Evaluation of the Optimization Approaches for Bottleneck

The evaluation of the optimization approaches for bottleneck starts at the previous introduced hybrid processing version as the original approach. Because the memory access methods are different in different CUDA devices, we select two graphic cards as the references. The G90-based Quadra FX4800 represents the currently common used C1.x architecture while the new released G100-based GEFORCE GTX480 represents the more powerful C2.0 device. Notice that the device memory accessing is cached in C2.0 device which may affect the performance of the array buffered approach.

Approaches	Compiling results		Maximum active blocks per multiprocessor (number (occupancy, limitation))	
	Shared memory per block (bytes)	Registers per thread	G90	G100
Original	2212+16	15	2(100%)	3(100%)
Array buffered	2212+16	15	2(100%)	3(100%)
Partial reduction	10304+16	15	1(50%, smem)	3(100%)
Unrolled	2112+16	22	1(50%, reg)	2(67%, reg)
Unrolled(C2.0)	2080+0	20	N/A	3(100%)

smem: shared memory, reg: registers

Occupancies

Quadra FX4800(C1.3, 24 multiprocessors, 1.5GB 384-bit GPU memory), Xeon E5440 2.53GHz CPU, 4GB Memory					
Approach	FPS under different active blocks per multiprocessor (256/512)			Maximum FPS	
	1	2	3	256	512
Original	36.48/12.67	41.30/13.43	39.22/13.57	40.76	14.32
Array buffered	36.70/14.51	31.22/12.53	28.10/12.29	36.7	14.51
Partial reduction	47.84/18.99	42.74/18.23	39.04/17.25	42.84	18.99
Unrolled	59.82/26.94	55.68/25.37	50.33/23.73	59.82	26.94

GEFORCE GTX480(C2.0, 15 multiprocessors, 1.5GB 384-bit GPU memory with 177.4GB/s bandwidth), Xeon E5440 2.53GHz CPU, 4GB Memory						
Approach	FPS under different active blocks per multiprocessor (256/512)				Maximum FPS	
	1	2	3	4	256	512
Original	57.07/17.44	86.72/29.79	91.82/34.11	86.72/29.79	91.82	34.11
Array buffered	77.80/28.54	83.25/32.16	85.46/32.43	72.95/30.04	85.46	32.43
Partial reduction	76.86/28.55	91.24/41.13	93.14/48.12	92.91/38.92	93.14	48.12
Unrolled	90.88/37.56	92.11/56.46	92.14/61.41	91.64/50.70	92.14	61.41
Unrolled(C2.0)	90.97/37.75	91.18/56.85	94.51/61.62	89.88/50.62	94.51	61.62

256: use 256 * 256 images as inputs, 512; use 512*512 images as input, underline: result under MAB, FPS, red/red: number per blocks over-saturated

Performance results

Table 4-1 Evaluation of bottleneck optimization approaches

Moreover, under C2.0 setting we found that the compiling result of the unrolled register buffered approach yields a higher MAB which could provide more parallelism. We list the result under this setting in a separate entry to test whether it helps to improve performance.

The occupancy (occupancy is defined as the ratio of the number of resident warps to the maximum number of resident warps[34] of the stream processor) and performance results of each approach are shown by **Table 4-1**.

4.5.2.1 Workload Distribution of Blocks

To test how the number of blocks per multiprocessors affects the performance, we record the results under different number settings. The results show the same pattern, illustrated by **Figure 4-11**. For all the approaches, the performance increases first from the under-saturated blocks per multiprocessors to the maximum values and then decreases. The maximum values appear under the MAB number setting. This proves setting the block configuration for the local accumulation kernels under MAB is the optimized solution.

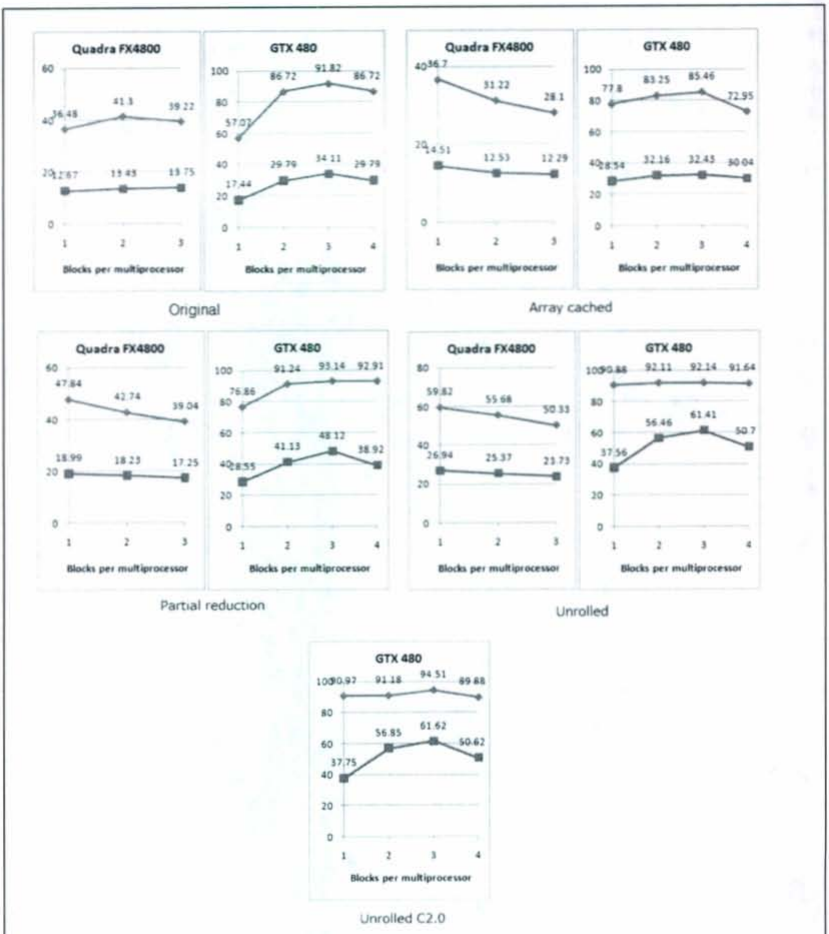
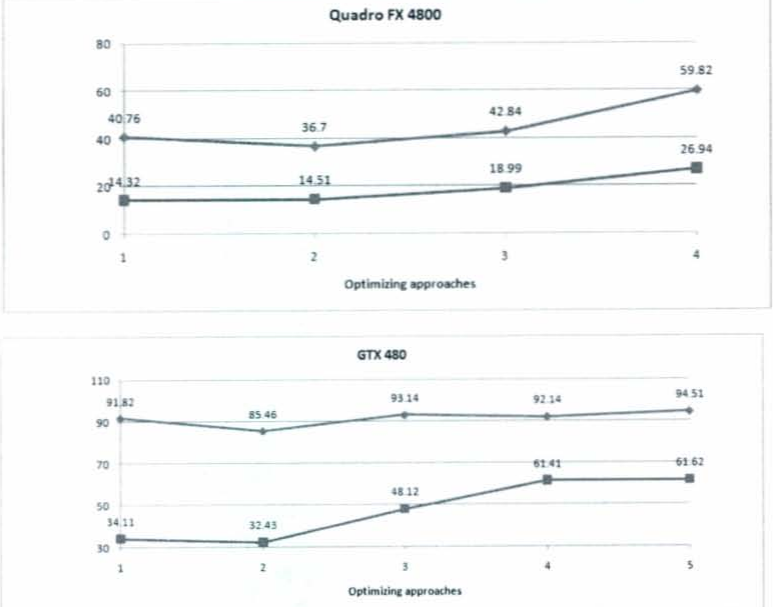


Figure 4-11 Performance of the bottleneck optimization approaches under different blocks per multiprocessor setting

4.5.2.2Performance Comparison



1: Original approach, 2: Local array buffered approach, 3: shared memory buffered approach, 4: Unrolled register buffered approach, 5: Unrolled register buffered approach (complied under C2.0).

Figure 4-12 Performance comparison between the bottleneck optimization approaches

Compared to the original approach, all except the local array buffered approach gain faster speed. This approach frequently accesses the local array and therefore decreases the entire performance. Even running on the C2.0 device which device memory accessing is cached, there is still a performance drop. It suggests that avoiding device memory access is still an important consideration even under this latest platform.

The unrolled register cached approach yields the best result under any conditions (up to 80% increase compared to the origin). Although the occupancy of the unrolled register cached approach is lower than other approaches under C1.x devices (Occupancies, **Table 4-1**) because of the heavy usage of registers, the gain of memory throughput surpasses the drop of instruction throughput and therefore leads to the improvement of the performance. The result suggests that for memory bandwidth-bound applications, we prefer increasing memory throughput to instruction throughput.

Particularly, compared to C1.x compiling version, the C2.0 compiling version doesn't significantly improve the speed although the compiler reports higher occupancy under this setting. It is possible because the C2.0 device could optimize the C1.x compiled kernel in run-time.

4.6 Discussion

This chapter discussed the techniques that optimize the parallel implementation of ICA on CUDA. They are capable of accelerating the parallel system form 10.63 FPS to 61.62 FPS, which accounts for 600% speed up.

Those optimization techniques are also applicable to other applications on CUDA. The low workload sequential execution is highly recommended being simulated in GPU with single thread or several threads to avoid the I/O communication. The I/O cost incurs a large penalty that will dramatically reduce the utilization of GPU.

For massive inter-thread summation, the parallel reduction combined with sequential-then-parallel processing is preferred because of its high efficiency and compatibility. If

atomic functions must be used because of lack of shared memory or other reasons, we recommend using the modified local function proposed in Appendix A.1 .

For in-thread caching, the register is the first choice for performance. If the caching task requires a small array, the array should be replaced by several hard-coding variables to ensure the elements are resided in register instead of the device memory. Even if mass usage of register could potentially reduce the occupancy of the GPU resources, it is still a good trade-off for reducing device memory accessing.

Chapter 5 Motion and Visual Controller

Nowadays, controllers other than the traditional gamepads are already wide-used in gaming consoles. They provide brand new experiences for users. In this chapter, we will propose a series of holding-and-pointing controllers. To control the cursor, the visual based controller uses the real-time image registration (IR) module presented in Chapter 3 , whereas the motion based controller uses accelerometers and gyroscopes. Moreover, we will propose how to build a more accurate hybrid controller by utilizing both visual and motion information.

5.1 Visual Based Controller

The visual based controller is the device that controls the cursor movement in 2-dimensional space using images captured by a video camera. The key to the controller is to use image registration algorithm to track the focal movement of the video camera. For any given two successive captured frames of the video camera, if they can be registered together, then we can use the registration result to update the position of the focus.

Figure 5-1 shows how we use the registration result to estimate the focal movement. The red rectangle indicates the image captured in last frame while the blue rectangle indicates the current frame. The colored dots are the center of the two images respectively. From the figure, we can clearly identify how the focus moves. After updating the new focal position, the current frame is used as the previous frame for the next registration. By

repeating this step, we can track the focal movement and use it to control the cursor in real-time.

Figure 5-1 Tracking focal movement by the registration result

5.1.1 Threading image capturing and image registration

The most important building block for the proposed device is the IR module. It iteratively fetches the image from the video camera, performs the image registration and control the cursor position using the mouse API with the registration result.

Video cameras usually capture the image stream in a constant frequency. If image registration and image capturing are run under the same thread, the final processing speed will be slowed down. For example, if the sampling frequency of the video camera is 30Hz and the IR module is capable of performing 60 registrations per second. Ignoring

the cost of other executions, the total execution time is 1000ms/30 + 1000ms/60 = 50ms, i.e., 20 registrations per second.

To address the problems, we introduce a multithread source fetching approach. Video stream capturing is conducted by an independent thread. This thread keeps updating image to system memory at the frequency of the video camera. If the source is updated, it labels an updating flag with "true" indicating the image is renewed. In another thread, IR module queries whether the image is updated. If yes, it will fetch the image and then set the updating flag "false" and conduct registration. Otherwise, this thread will sleep itself for a while and then do another query.

Resource locking is used for unintended image updating. When the IR module is fetching image, it will lock the image to disable updating from the input handling thread. After fetching, it will unlock the image to enable updating.

Figure 5-2 illustrates the multi-threading approach.

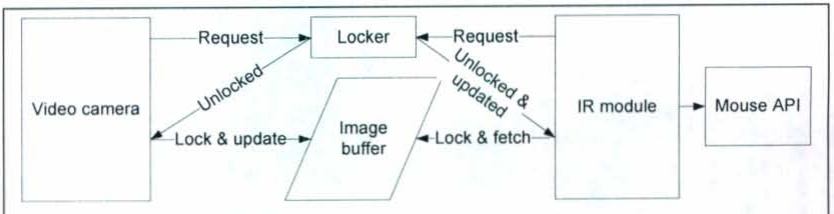


Figure 5-2 Implementation of the visual based controller

In this way, multithreading allows IR module runs faster than image capturing. Moreover, it parallels the image capturing and the image registration process, allowing the CPU to handle image capturing while the GPU performs registration.

5.1.2 Mouse control

After every registration, the result warp matrix is used to control the cursor of the mouse. If a warp matrix is,

$$\boldsymbol{W} = \begin{bmatrix} p_0 & p_1 & p_2 \\ p_3 & p_4 & p_5 \\ p_6 & p_7 & 1 \end{bmatrix}$$

then, the displacement $\boldsymbol{s}$ of the focal movement is

$$\begin{aligned} s_x &= \frac{p_0 f_x + p_1 f_y + p_2}{p_6 f_x + p_7 f_y + 1} - f_x \\ s_y &= \frac{p_3 f_x + p_4 f_y + p_5}{p_6 f_x + p_7 f_y + 1} - f_y \end{aligned} \tag{5.1}$$

where (f_x, f_y) is the coordinate of the focus of the captured image. Usually, it is the center of the image.

There are two ways to control the cursor. The first is passing $s_x \; and \; s_y$ as the displacement of cursor to the mouse API. This method is simple to implementation and allows the proposed device cooperating with other devices to control the same cursor. The second is integrating $s_x \; and \; s_y$ to trace the absolute position of the focus and set it as the absolute coordinate of the cursor. The advantage is that there is no error accumulation due to the conversion of float coordinates to integer coordinates or the losses of mouse control messages. Choosing which way to control the cursor could be depended on the requirements of the application.

5.1 Motion Based Controller

In this section we will present an efficient wireless air-mouse using accelerometers and gyroscopes. When using the controller, the user simply rotates it horizontally to control the cursor movement in horizontal direction and vertically to control the cursor movement in vertical direction.

5.1.1 Introduction of Wiimote

The device to provide accelerometers and gyroscopes is the Wii remote controller (Wiimote). A built-in ADXL330 accelerometer measures accelerations along three perpendicular axes. The accelerations range from -3g to +3g gravitational force. If attached by an add-on device called Wii MotionPlus (short for MotionPlus), Wiimote can sense angular velocities along 3 directions – yaw, pitch and row with the MotionPlus incorporated two-axis tuning fork gyroscope. **Figure 5-3** illustrates the two sets of the motion data.

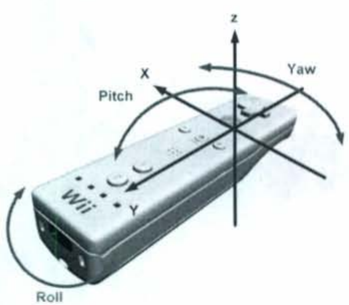


Figure 5-3 Wiimote motion sensing

The Wiimote also provides the necessary buttons and the connection to PC. It can be paired with Bluetooth receiver and programmed by driver APIs to retrieve the motion readings.

5.1.2 Motion Estimation

The movement of the cursor is controlled by the horizontal and vertical angular velocities when waving the Wiimote. If we define the h and v as the magnitudes of the current horizontal and vertical angular velocities, then the displacement of the cursor $\boldsymbol{s}(s_x, s_y)$ can be calculated by

$$s_x = \alpha h, s_y = \alpha v \tag{5.2}$$

where α is the coefficient to decide how fast the cursor move.

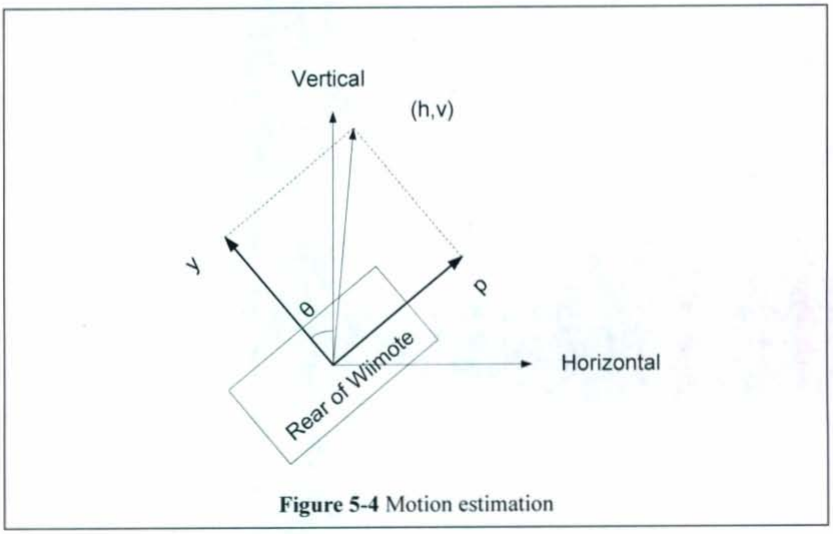


Figure 5-4 Motion estimation

Denote the current readings of yaw and pitch as y and p. If the rotation angle along the Y-axis of the Wiimote (angle of roll) is θ (**Figure 5-4**), then we can calculate h and v by,

$$h = y\cos\theta + p\sin\theta$$
$$v = -y\sin\theta + p\cos\theta \quad (5.3)$$

5.1.3 Estimating Angle of Roll

The only unknown in equation 5.3 is the angle of roll θ. It can be measured by the accelerometers. If we arrange the readings of accelerometers to a vector (x, y, z), when holding still, this vector is actually the gravitational vector. The values of the elements are the projections of gravity on the three perpendicular axes and the length of the vector is equal to 1. When the Wiimote is parallel to ground surface, the value of the vector is (0, 0, 1). If we rest the Wiimote on a slanted surface, the angle of roll is equal to

$$\theta = arctan\frac{x}{z} \quad (5.4)$$

5.1.4 Implementation

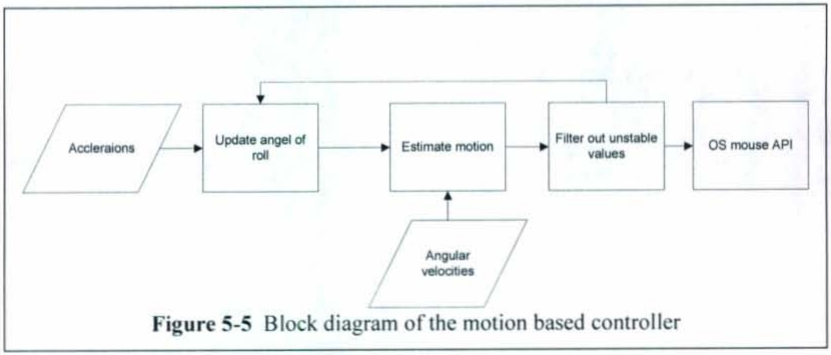


Figure 5-5 Block diagram of the motion based controller

Figure 5-5 shows the implementation of the motion based controller. In each iteration of the mouse controlling, the first step is to update the angle of roll according to Equation 5.4. Notice that for a accurate estimation of the angle of roll, the updating is not performed for each iteration [35]. The system will track the length of the acceleration vector to ensure it is almost equal to one for a small period (e.g., several iterations). If so, the Wiimote is assumed to be almost stationary and therefore the acceleration vector well represents the gravitational vector. At this time, the angle of roll is updated.

The second step is to estimate the motion by simply following Equation 5.3 to compute the horizontal and vertical angular velocities.

To provide stable feedbacks for mouse API, we need to filter out unstable values. Shake of hands and device reading errors result in noises. If we let noises pass through, controlling cursor will be hard. Here, we add a threshold to filter out the noise and amplify the signal-to-noised ratio.

At the last step, we output the motion parameters tuned by Equation 5.2 to mouse API to move the cursor.

5.2 Hybrid Controller

The visual based controller provides high accuracy for slow or small range motion, but the accuracy degenerates under fast motion due to the low sampling frequency of the video camera. Captured at 30 FPS under fast motion, two adjacent frames may have very small overlapped region, which may cause the image registration process fail. Thus the tolerance of focal moving speed is limited to a relative small range. The motion based

controller provides reasonable real-time motion feedbacks, even for the fast movement, but it has a relatively low accuracy because of the error accumulation of the motion measurements and the estimation of the angle of roll. If we combine the merits of the two devices, we could build a more efficient controller.

In this section, we propose the hybrid controller. It uses the motion based estimation to provide pre-knowledge to the visual based image registration and thus yields a high accuracy even under fast motion.

5.2.1 Motion Based Pre-Knowledge

To improve the robustness of the visual based controller under fast motion, we need to provide images with larger overlapped region to the IR module. If we extend the field of view of the video camera and extract an area as the current image to be registered corresponding to the current focal moving direction which is estimated by the motion based approach, the region should have a larger overlapped region with the next frame.

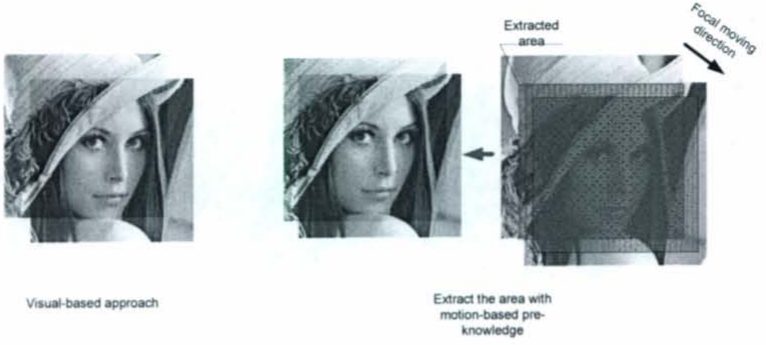


Figure 5-6 Motion based pre-knowledge

Figure 5-6 illustrates this case, where the red rectangle indicates the extracted area in the extended captured image whose position is corresponding to the current focal moving direction. The black rectangle indicates the region to be registered in the next frame which should be the same as the un-extended image. Compared to the two frames used in visual based approach under the same condition (left in **Figure 5-6**), they have larger overlapped region.

5.2.2 Floating Window

Here we use term "floating window" to refer the approach for extracting the informative area using the motion based pre-knowledge. It uses a fixed-size window floating in the extended region to decide which area to be extracted as the input image for the next registration. The position of the window (represented by the coordinate (x, y) of the top-

left corner) is decided by the current focal velocity estimated by the motion based approach.

If the resolution of the camera is extended by $2m$ pixels both in width and height, the coordinate (x, y) is updated according to Equation 5.5

$$x = \begin{cases} m & h = 0 \\ m + \dfrac{h}{A} & h \neq 0, \left|\dfrac{h}{A}\right| < m \\ m + \dfrac{|h|}{h}m & h \neq 0, \left|\dfrac{h}{A}\right| \geq m \end{cases}$$

$$y = \begin{cases} m & v = 0 \\ m + \dfrac{v}{A} & v \neq 0, \left|\dfrac{v}{A}\right| < m \\ m + \dfrac{|v|}{v}m & v \neq 0, \left|\dfrac{v}{A}\right| \geq m \end{cases} \tag{5.5}$$

where h and v is the horizontal and vertical focal velocities and A is a positive integer to reduce the range of the h and v. A should be set according to the screen resolution of the target application.

When holding still, h and v are equal to 0, the floating window is located in the center of the image with (x, y) is equal to (m, m) (**Figure 5-7**, a). If the focus is moving, the floating window will move to the coordinate proportion to the estimated focal velocities, h and v (**Figure 5-7**, b), but it cannot exceed the border of the extended region (**Figure 5-7**, c).

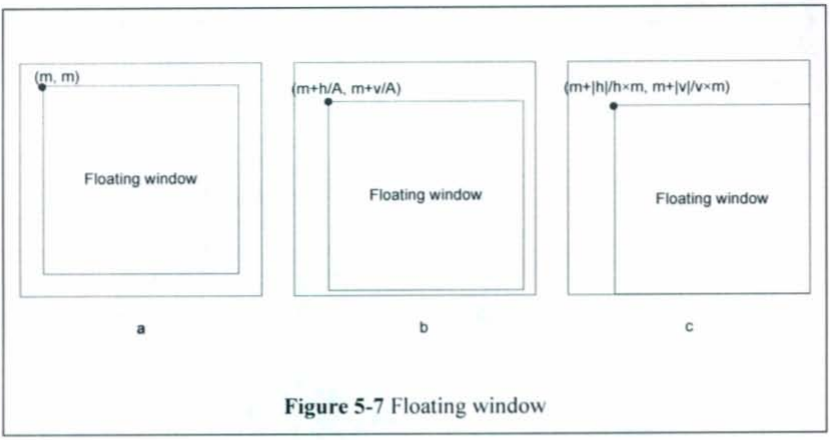


Figure 5-7 Floating window

5.2.3 Implementation

Figure 5-8 shows the flow chart of the hybrid controller. The procedure of each iteration follows the arrow labeled from 1 to 6 in the figure. The iteration begins when the video camera finishes updating the current frame. Next, the center area of the newly captured image is used as the template image and the region extracted from the previous frame is used as input image. Both are sent to the IR module to perform the registration. We use the registration result to control the cursor in the same way as the visual based approach. At last, we use the current focal angular velocities to locate the float window to be used for extracting the input image for the next registration.

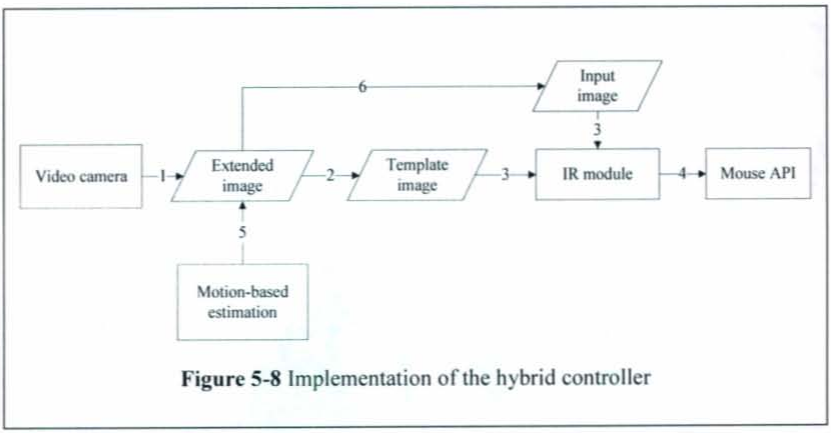


Figure 5-8 Implementation of the hybrid controller

5.3 Accuracy Evaluation

The evaluation is to test the accuracy of the visual based controller, the motion based controller and the hybrid-controller. During the test, the tester holds the controller in an initial state. Then, we let the tester to do a gesture and then go back to the initial state. The cursor will move corresponding to the controller motion. Under ideal situation, the cursor should at the same location before and after the gesture. However, there will be a drift in practice due to either the error accumulation of the controller or the tester failed returning the controller back to the initial position. To reduce error caused by the second reason and measure the error accumulation only, we save the image captured by the camera under the initial position so that the tester can use it to guide the controller back to the initial state after the gesture. The user is asked to perform multiple gestures using the same starting point and we record multiple ending positions after all the gestures and use the distribution of the drifts of the ending positions to measure the accuracy of the tested controller.

The procedure is illustrated by **Figure 5-9**. The tester aims the controller with the image captured video. At first, we let the tester aim at the image center indicated by a dot. Then we reset the cursor to the starting position and let the tester do the gesture. After the gesture, the tester should points the controller back again at the center dot. Then, we record the ending position of the cursor. This procedure is repeated multiple times to draw the distribution of the drifts.

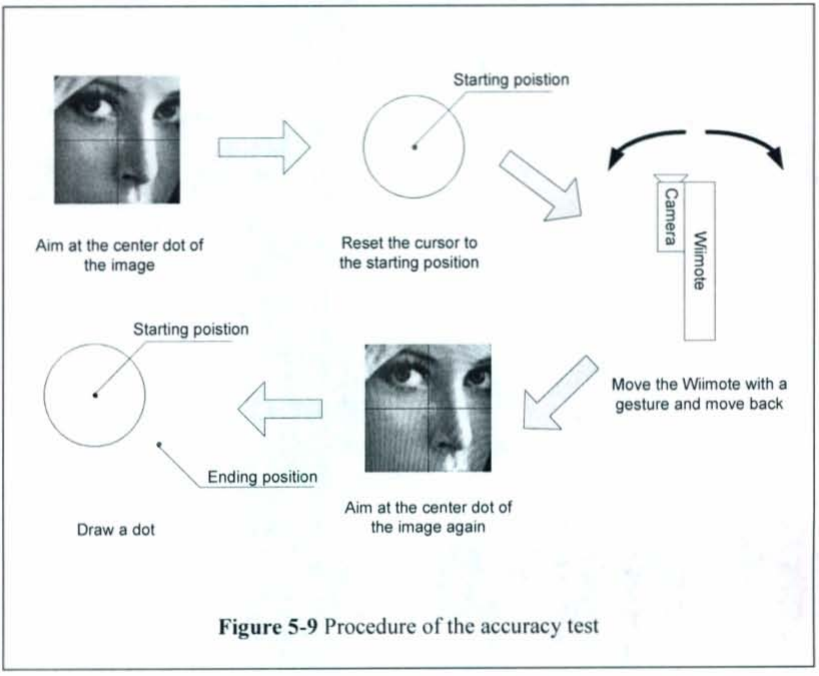


Figure 5-9 Procedure of the accuracy test

The first test uses gestures of small motion. The tester waves the Wiimote towards a random direction and waves back.

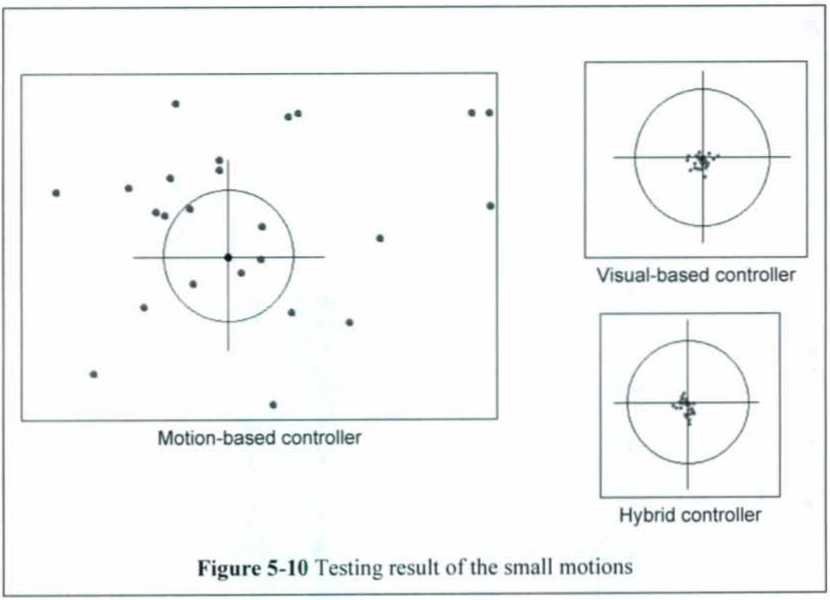


Figure 5-10 Testing result of the small motions

Figure 5-10 shows a set of the testing result in which 25 gestures were performed for each controller. The distribution of drifts of the motion based controller is much sparser and far away from the starting position while the distributions of the other two controllers are both located near the starting position. This shows that the image registration approach provides much higher accuracy than the motion based approach.

In this test, we can hardly tell the difference between the visual based controller and the hybrid controller. The variances of the distribution may be both due to the operation error of the testers. We would like to introduce more complex motions to test the difference. In the second test, the tester shakes the Wiimote very fast for about two seconds before going back to initial state. The testing result is shown in **Figure 5-11**.

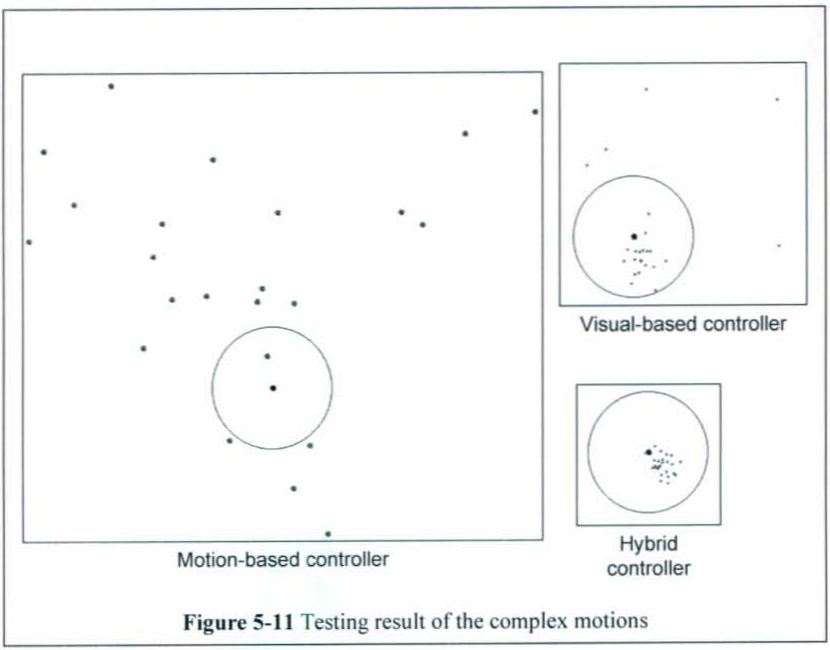


Figure 5-11 Testing result of the complex motions

In this test the distribution of the motion based controller is even worse and we can clearly identify the improvement of the hybrid controller over the visual based controller. The distribution is more dense and closer to the expected destination. Moreover, because of the registration failures, outliers appear in the visual based case but there is none in the hybrid case.

5.4 Discussion

We show in this chapter how to build a controller using the real-time IR module. The evaluation shows that, even using visual information only, the visual based controller can provide more accurate pointing control over the widely-used motion based counterpart. Through combining both visual based and motion based information, the hybrid

controller further improves the robustness of system under fast motion. Compared with the conventional sensor-bar-based Wiimote pointing, the hybrid controller presented here does not restrict users to pointing toward the sensor bar. The holding and pointing controlling method provides users vivid gaming experience, which could make it a more preferred approach over the traditional gamepad or the mouse.

Chapter 6 Other Applications and Conclusions

The proposed image registration system can not only be used to build the proposed hybrid game controllers, but also be applied to other real-time vision applications.

6.1 Light-Gun for LCD

The light gun has been a popular pointing device for shooting games. The traditional light guns only work with CRT monitor since they uses cathode ray timing information. To estimate the pointing position on LCD monitors, Wiimote uses an additional inferred light emitting device, called sensor bar, which is placed near the screen. Nevertheless, this approach requires an additional calibration process and lacks precision. Users may have bad experience in shooting games because, in order to shoot an object on screen, they may have to aim to a different location. Moreover, the resolutions of those devices are much less than the mouse, which makes them not applicable to high competitive games.

To improve the pointing precision, the proposed image registration system can be used. Basically, once we register the image captured by the handheld camera with image displayed on the screen, we will know where the center of the camera points to. To reduce the computational cost and improve the registration robustness, we can also perform the registration using a region extracted around the pointing position provided by the traditional aiming devices, instead of the whole image.

6.2 Real-Time Image Mosaic

Figure 6-1 Real time image mosaic

With the high registering speed, the proposed image registration system could be used to generate an image mosaic from real-time video input. That is, we can use the transformation matrix calculated to warp the current frame and stitch it with previous frames.

To warp the images in real time, we use DirectX 3D (D3D) to accelerate the processing speed. The quadrangle frames are represented by of the D3D geometries. During

registration, the vertices of the geometry are warped by the warp matrix and then pixels are filled in through hardware texturing.

To remove the high cost of I/O transportation between device and host, we design an interface to access image in graphic memory directly. Currently, it is capable of reading images from DirectX 3D textures or OpenGL buffers. The captured images are sent to D3D first and then the image registration system fetches the image from the interface. After registering, the images are again used for real-time warping.

Figure 6-1 shows such a real-time image stitching result. Our current implementation is capable of handling more than 30 images per second in a PC with a mid-range graphic card.

6.3 Conclusions

The thesis proposes to use both visual and motion information for designing hybrid controllers for next-generation game consoles. To achieve this goal, we implement the key building block, the image registration module, in parallel for real-time processing.

Besides implementing the image registration algorithm on GPUs, we also implement two multi-resolution variants, each has its own advantages under difference situations. MRIR is suitable for high accuracy homographic image registration. MRIR-SR has lower computational cost and can be used to register the images that have small overlapped region.

Compared to traditional motion based controller such as Wiimote, the proposed motion-visual-hybrid controller uses image registration result to provide high accuracy ego-

motion information, as well as using input of accelerometer to provide pre-knowledge of the movement. Experimental results show that the hybrid controller performs much better than the controllers that use either motion or visual information only.

In summary, the following techniques/algorithms are discussed in this thesis:

- Real time image registration module

  The proposed image registration module can register 512×512 images at 60 FPS with high accuracy. It can be integrated to other application requiring real-time image alignment.

- Multi-Resolution Image Registration with Selected Regions

  This technique dramatically reduces the computational cost of multi-resolution image registration. Moreover, it increases the adaptability of the homographic image registration to inhomogeneous images.

- Discussed CUDA optimization techniques

  We present a series of optimization techniques to improve the performance of the image registration module. Those techniques are not only applicable to image registration, but also useful for optimizing other CUDA applications. We also discuss how to design optimized data structure for accessing CUDA memory.

- CUDA in-thread buffering

  We propose a method that unrolls small loops and hard codes the small vector with separate variables to enforce using registers to buffer intermediate results (Section

4.3.3). By comparing with the array or shared memory buffering approach, we find that this method is much more efficient. Therefore, we recommend using this method to buffer small vector or array for other applications.

- Motion based pointing device

  We propose how to design a pointing device using Wiimote based on only the motion feedbacks. Compared to the traditional infrared ray solution which needs pointing Wiimote towards a sensor bar, this approach requires no additional device. It also has very low computational cost but lacks of pointing accuracy.

- Hybrid pointing device

  We propose how to integrate the focal tracking based on image registration with motion information to build a high accuracy hybrid pointing device. It provides pixel-level resolution that may extend usage of the motion controller to the high competitive games.

Chapter 7 References

1. Zitova, B.a.F., J, Image registration methods: a survey, in Image and Vision Computing. 2003. p. 977-1000.

2. Moravec, H.P., Rover visual obstacle avoidance, in International Joint Conference on Artificial Intelligence. 1981: Vancouver, Canada. p. 785-790.

3. Lowe, D.G., Distinctive Image Features from Scale-Invariant Keypoints, in International Journal of Computer Vision. 2004.

4. Fischler, M.A. and R.C. Bolles, Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography, in Communications of the ACM. 1981. p. 381-395.

5. Davison, A.J., Real-time simultaneous localisation and mapping with a single camera. 2003.

6. Davison, A.J., Active search for real-time vision, in Tenth IEEE International Conference on Computer Vision. 2005. p. 66-73.

7. Pratt, W.K., Digital image processing (2nd ed.). 1991, New York, NY, USA: John Wiley & Sons, Inc.

8. Barnea, D.I. and H.F. Siverman, A class of algorithms for fast digital image registration, in IEEE Transactions on Computing 21. 1972. p. 179-186.

9. Lucas, B. and T. Kanade. An Iterative Image Registration Technique with an Application to Stereo Vision. in Imaging Understanding Workshop. 1981.

10. Baker, S. and I. Matthews, Lucas-kanade 20 years on: A unifying framework: Part 1: The quantity approximated, the warp update rule, and the gradient descent approximation. International Journal of Computer Vision, 2004. 56(3): p. 221–255.

11. Burt, P.J., Fast filter transforms for image procesing, in Computer Graphics and Image Processing. 1981. p. 20-51.

12. Wong, R.Y. and E.L. Hall, Sequential Hierarchical Scene Matching, in IEEE Transactions on Computers. 1978, IEEE Computer Society p. 359-366.

13. Kumar, R., et al. Registration of video to geo-referenced imagery. in Fourteenth International Conference on Pattern Recognition. 1998.

14. Sarvaiya, J.N., S. Patnaik, and S. Bombaywala, Image Registration by Template Matching Using Normalized Cross-Correlation, in International Conference on Advances in Computing, Control, and Telecommunication Technologies. 2009

15. Stefano, L.D., S. Mattoccia, and M. Mola. An efficient algorithm for exhaustive template matching based on normalized cross correlation. in 12th International Conference on Image Analysis and Processing. 2003.

16. Lewis, J.P., Fast Normalized Cross-Correlation, in Vision Interface. 1995.

17. Wong, E.L., W.Y.F. Yuen, and C.S.T. Choy. Designing Wii Controller As A Powerful Musical Instrument In An Interactive Music Performance System. in The 6th International Conference on Advances in Mobile Computing and Multimedia. 2008.

18. Schlömer, T., et al. Gesture recognition with a Wii controller. in the 2nd international conference on Tangible and embedded interaction. 2008. Bonn, Germany

19. Cheong, S.N., W.J. Yap, and M.L. Chan. Cost-Effective Wiimote-Based Technology-Enhanced Teaching and Learning Platform. in the 10th Pacific Rim Conference on Multimedia: Advances in Multimedia Information Processing. 2009.

20. Luinge, H.J., P.H. Veltink, and C.T.M. Baten. Estimating orientation with gyroscopes and accelerometers. in The First Joint BMES/EMBS Conference. 1999.

21. Sakaguchi, T., et al., in IEEE/SICE/RSJ International Conference on Multisensor Fusion and Integration for Intelligent Systems. 1996. p. 470-475.

22. Harris, M., GPGPU: General-Purpose Computation on GPUs.

23. Rosado, G., Motion Blur as a Post-Processing Effect, in GPU Gem 3. 2007.

24. Fung, J. and S. Mann, Using graphics devices in reverse: GPU-based Image Processing and Computer Vision, in 2008 IEEE International Conference on Multimedia and Expo. 2008. p. 9 - 12.

25. Yu, W. and T. Chen, High Performance Stereo Vision Designed for Massively Data Parallel Platforms, in IEEE Transactions on Circuits and Systems for Video Technology

26. Jargstorff, F. and E. Young, CUDA Video Decoder API. 2008.

27. Shen, G., et al. Accelerating video decoding using GPU. in 2003 IEEE International Conference on Acoustics, Speech, and Signal Processing. 2003.

28. Nyland, L., M. Harris, and J. Prins, Fast N-Body Simulation with CUDA, in GPU Gems 3. 2007.

29. Grand, S.L., Broad-Phase Collision Detection with CUDA, in GPU Gems 3. 2007.

30. Garcia, V., E. Debreuve, and M. Barlaud, Fast k nearest neighbor search using GPU, in IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops. 2008. p. 1-6.

31. Sintorn, E. and U. Assarsson, Fast parallel GPU-sorting using a hybrid algorithm Parallel and Distributed Computing, 2008. 68(10): p. 1381-1388.

32. Harris, M. Optimizing Parallel Reduction in CUDA.

33. Harris, M., S. Sengupta, and J.D. OWens, Parallel Prefix Sum (Scan) with CUDA, in GPU Gems 3. 2007.

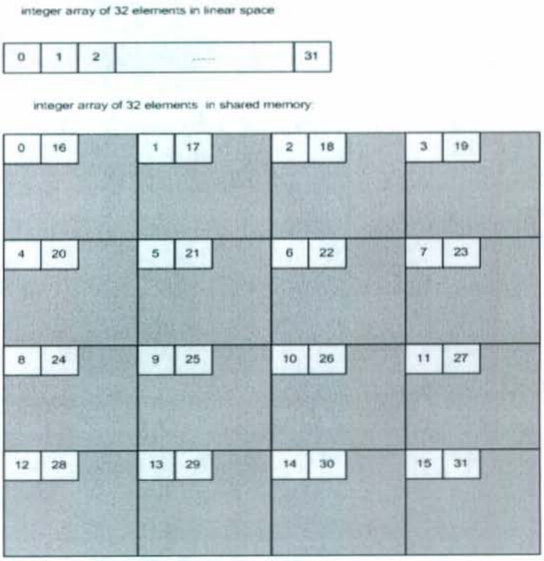
34. NVIDIA. NVIDIA CUDA Programming Guide V3.0.

35. gl.tter. Wiiyourselft! wiiyourself.gl.tter.org

Appendix A

A.1 Bank conflict

Due to the hardware design, simultaneously accessing the shared memory of CUDA devices from multiple processing units may cause bank conflicts. The on-chip shared memory is divided into several banks (16 banks for C1.x devices and 32 banks for C2.0 devices). 32-bit words will be distributed into those banks in repeating sequential order. The following figure shows how the elements of an integer array are distributed to the shared memory.



Data distribution of shared memory under C1.x

The following discussion is based on the C1.x devices. If the memory access in a warp falls into distinct banks, they can be done simultaneously. However, when two or more accesses fall into the same bank, they will be serialized, which causes latency. If two accesses fall into the same bank, the bank conflict is referred as a 2-way bank conflict. Likewise, if the number of accesses is n, it is referred as an n-way bank conflicts. Between certain threads in a warp, there are no conflicts. The shared memory access in a warp is separated into two half, so the first 16 threads in the warp will never conflict with the remained 16 threads in the second half. Thus, the worst case scenario is a 16-way bank conflict where all the threads in a half warp fall in the same bank. The latency is as 16 times as the no conflict case.

A.2 Atomic Reduction Without Bank Conflicts on CUDA Device

To avoid bank conflict (Appendix A.1), we introduce a buffered shared memory access pattern to perform atomic instruction. An integer array of 16 elements is allocated in shared memory to store the intermediate results. The sum of all the elements is calculated by adding together all the intermediate results. The code below shows an example of buffered shared memory access.

```
//Sum elements in one block (up to 512 elements) and write the result
//to d_rst
__shared__ int s_buff[16];
if(threadIdx.x < 16)
      s_buff[threadIdx.x] = 0;
__syncthreads();
atomicAdd(s_buff + (threadIdx.x % 16), elem[threadIdx.x]);
__syncthreads();
&d_rst = Reduce(s_buff);
```

A.3 Optimized Parallel Reduction on CUDA Device

```
//Parameters:
//s_sum: shared memory used as buffer.
//elem: should be passed by a local variable storing the
//          value of the element of the current thread.
//return: result of the parallel reduction
__device__ float reduce512(volatile float* s_sum, float elem)
{
      s_sum[threadIdx.x] = elem; __syncthreads();
      if(threadIdx.x < 256){s_sum[threadIdx.x] = elem = elem +
            s_sum[threadIdx.x + 256]; }__syncthreads();
      if(threadIdx.x < 128){s_sum[threadIdx.x] = elem = elem +
            s_sum[threadIdx.x + 128];} __syncthreads();
      if(threadIdx.x < 64){s_sum[threadIdx.x] = elem = elem +
s_sum[threadIdx.x + 64]; }__syncthreads();
      if(threadIdx.x < 32)
      {
            s_sum[threadIdx.x] = elem = elem +  s_sum[threadIdx.x +
32];
            s_sum[threadIdx.x] = elem = elem +  s_sum[threadIdx.x +
16];
            s_sum[threadIdx.x] = elem = elem +  s_sum[threadIdx.x +
8];
            s_sum[threadIdx.x] = elem = elem +  s_sum[threadIdx.x +
4];
            s_sum[threadIdx.x] = elem = elem +  s_sum[threadIdx.x +
2];
            elem +=  s_sum[threadIdx.x + 1];
      }
      return elem;
}
// in kernel
...
__shared__ float s_sum[512];
float elem = 0;
//load elements from device memory
elem = vector[threadIdx.x]; __syncthreads();
float sum = reduce512(s_sum, elem);
...
```

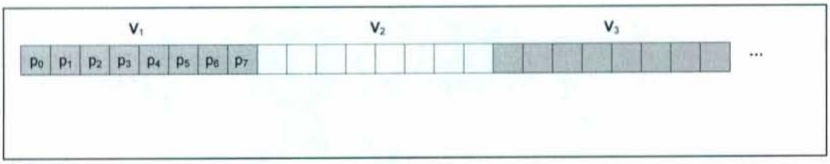
The optimized parallel reduction presented by the above figure is the modified version based on Harris' work [32]. It introduces sequential addressing to avoid bank conflicts and fully unrolls codes to reduce the complexity of flow control. The valuable *elem* is

used for ensuring register is used for storing intermediate result that reduce shared memory accessing. The number of adding operations for each thread is 9. The shared memory usage is $512 \times 4 = 2\text{KB}$.

A.4 Parallel Reduction for Vectors on CUDA Device

The basic idea of parallel reduction for vectors is to use the idle threads to reduce multiple dimensions at one time. If we consider the thread usage of the optimized parallel reduction, we will find a lot of threads idle during the procedure. To load 512 element form global memory, we need 512 threads. The first step of reduction needs 256 threads, then 256 threads idle. The second step needs 128 threads, then 384 threads idle, and so on.

The parallel reduction for vectors needs a special storage pattern to conduct sequential memory accesses so that there are no bank conflicts. Vectors should be stored sequentially in linear space. The following figure illustrates this pattern,



If the vectors are saved in this pattern, we can use the same sequential addressing method as the optimized parallel reduction to sum the vectors but stop the reduction earlier. For example, the following device function can be used for adding 64 8-dimentional vectors.

```
__device__ void reduce64for8(volatile float* s_sum, float elem,
float* s_rst)
{
      s_sum[threadIdx.x] = elem; __syncthreads();
      if(threadIdx.x < 256){s_sum[threadIdx.x] = elem = elem +
s_sum[threadIdx.x + 256]; }__syncthreads();
      if(threadIdx.x < 128){s_sum[threadIdx.x] = elem = elem +
            s_sum[threadIdx.x + 128];} __syncthreads();
      if(threadIdx.x < 64){s_sum[threadIdx.x] = elem = elem +
            s_sum[threadIdx.x + 64]; }__syncthreads();
      if(threadIdx.x < 32)
      {
            s_sum[threadIdx.x] = elem = elem +  s_sum[threadIdx.x +
32];
            s_sum[threadIdx.x] = elem = elem +  s_sum[threadIdx.x +
16];
            elem = elem +  s_sum[threadIdx.x + 8];
      }
      if(threadIdx.x < 8)
            s_rst[threadIdx.x] = elem;
      __syncthreads();
}

//in kernel
__shared__ float s_sum[512];
__shared__ float s_rst[8];
float elem = 0;
//load elements from device memory
elem = vector[threadIdx.x]; __syncthreads();
//add vectors and save result to s_rst[8]
reduce64for8(s_sum, elem, s_rst);
```

The number of addition operation is 6. Compared to 48 operations, it is a significant improvement. If more than 64 vectors of 8 elements are to be added, sequential-then parallel processing can be introduced.

A.5 Local Array Cached Parallel Reduction Approach

To apply the hybrid processing strategy, the whole image is partition into the segments of 512 horizontally adjacent pixels in row major. Each blocks sequentially accumulating M segments.

```
//parallel reduction buffer
__shared__ float s_sum[512];
//output buffer
__shared__ float s_dp[8];
...
//thread index
int tid = blockIdx.x * blockDim.x + threadIdx.x;
//conver thread index, tid,  to pixel, coordinate (i,j)
int i = idx % image_width;int j = idx / image_height;
//sequtetianl accumulation image segments for M times
for(int c = 0; c < M; ++c)
{
      //overflow checking
      if(idx >= image_pixel_size)break;
      float x, y;
      //warp (i,j) to (x,y) by the current warp matrix
      Warp(i, j, &x, &y);
      float diff = 0;
      //intensity difference between template image, T,
      //and input image, I
      if(x < w && y < h && x > 0 && y > 0)
            diff = I(x, y) - T(i, j);
      //accumulate the local parameters of the warp matrix
      for(int s = 0; s < 8; ++s)
      {
            float dp = diff == 0 ? 0 : diff * SDI(i, j, s);
            //parallel reduction
            float sum = reduce512(s_sum, dp);
            //accumulate to output buffer
            if(threadIdx.x  == 0)s_dp[s] += sum;
      }
      //move to the next image segment
      idx += gridDim.x * blockDim.x;
} __syncthreads();
//output the intermediate results
if(threadIdx.x < 8) local_parameter[blockIdx.x * 8 + threadIdx.x] =
s_dp[threadIdx.x];
```