Accelerated Optical Flow Computation using Foveated Vision and Compute Unified Device Architecture

ACCELERATED OPTICAL FLOW COMPUTATION USING FOVEATED VISION AND COMPUTE UNIFIED DEVICE ARCHITECTURE

BY

PETER KUCHNIO, B.Eng.

A THESIS

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

Master of Applied Science (2009) McMaster University
(Electrical & Computer Engineering) Hamilton, Ontario, Canada

TITLE:	Accelerated Optical Flow Computation using Foveated Vision and Compute Unified Device Architecture
AUTHOR:	Peter Kuchnio B.Eng., (Engineering Physics) McMaster University, Hamilton, Ontario, Canada
SUPERVISOR:	Dr. David Capson
NUMBER OF PAGES:	xv, 127

Abstract

Optical flow is a well known technique for the measurement of motion in images. Although it has many applications, calculating the optical flow remains computationally expensive and challenging to use in time-critical tasks. This thesis describes an accelerated approach to optical flow computation using foveation and parallel processing on a Graphics Processing Unit (GPU). Foveation reduces the amount of image data to process by mimicking the variable resolution structure of the human visual system. The resulting image data is processed in parallel on a 240 processor GPU to achieve high frame rates on high resolution images. The newly introduced Compute Unified Device Architecture (CUDA) framework is utilized to create an efficient mapping of optical flow and foveation algorithms to the GPU.

The performance and error of the algorithm is characterized using synthetic and real data. The non-foveated optical flow algorithm is found to perform up to 100× faster than a CPU implementation. Foveated optical flow is found to give an additional performance gain of up to 27× over non-foveated optical flow with a corresponding increase in angular error. The results are shown to match or outperform FPGA and non-CUDA GPU implementations. Finally, the application of the described system to real-time control of a robot arm is demonstrated.

Acknowledgements

I would like to sincerely thank my supervisor, Dr. David Capson, for his support and guidance during the course of this research. I would also like to thank my family for their support, and in particular my wife, Alicja, without whose encouragment and editing assistance I would not have finished this thesis. I also thank my fellow researchers and friends in the Computer Vision laboratory.

Notation and abbreviations

Symbols and Definitions	
x	Scalar
$\vec{x}$	Vector
$I(\vec{x}, t)$	Image frame
J_s	Image Jacobian
$J(\vec{\theta})$	Kinematic Jacobian
ν	Correlation window radius
n	Search window radius
Acronyms	
AAE	Average Angular Error
API	Application Programming Interface
CCD	Charge-Coupled Device
CMOS	Complementary Metal-Oxide Semiconductor
CUDA	Compute Unified Device Architecture
DMA	Direct Memory Access
GPU	Graphics Processing Unit
IBVS	Image-Based Visual Servo
LPT	Log-Polar Transform
SIMD	Single-Instruction Multiple-Data
SIMT	Single-Instruction Miltiple-Thread
SSD	Sum-of-squared Differences
STD	Standard Deviation
RAM	Random Access Memory
VLSI	Very Large-Scale Integration

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Measuring the velocity of objects moving in a sequence of images remains a key problem in computer vision. Calculating the 3-D velocities of moving objects projected onto a 2-D visual sensor is dubbed optical flow, and once computed, can be used for a wide variety of tasks. Optical flow field are used to segment images, find the time-to-collision, estimate motion, compress video, and for autonomous navigation in robotics.

1.1 Problem Definition

Measuring the optical flow is a computationally intensive problem, and it remains challenging to calculate at frame rates suitable for time-critical tasks. The recently introduced Compute Unified Device Architecture (CUDA) is a promising approach to this problem since it allows a standard Graphics Processing Unit (GPU) to be treated as a general purpose parallel computer. The goal of this thesis is to develop a set of scalable, parallel algorithms, mapped to the CUDA platform, to accelerate

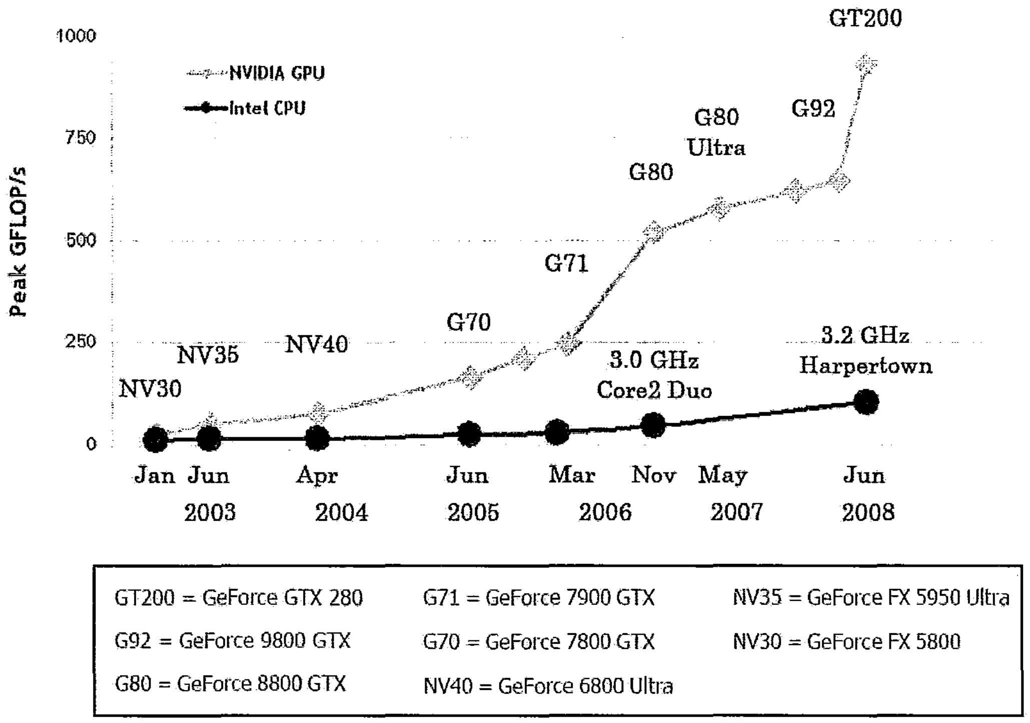


Figure 1.1: Evolution of the computation capability of GPUs versus CPUs. Figure reproduced from NVIDIA [1]

optical flow computation to speeds suitable for time-critical tasks.

1.2 Parallel Programming on the Graphics Processing Unit

Graphics Processing Units (GPUs) have evolved into parallel, programmable, multi-core architectures capable of approaching teraflop performance. The market demand for real-time, high-definition 3-D graphics has driven the continued expansion of GPUs. The number of gigaflops per second that modern GPUs can execute has continued to outpace the capabilities of CPUs [1], as illustrated in Figure 1.1.

However, the traditional graphics processor software model has limited their application to general parallel computation by forcing algorithms to be mapped to graphics application programming interfaces (APIs) [2]. The first generations of GPUs only contained a fixed-function graphics pipeline. As technology advanced, this was replaced with a programmable pipeline, where rendering effects could be programmed using a set of software instructions called *shaders*. Shaders have traditionally lacked features necessary for more general, non-rendering, algorithms. This typically reduces execution speed and limits applications. These restrictions have been lifted by the introduction of a new programming paradigm, the Compute Unified Device Architecture (CUDA) by NVIDIA. CUDA allows GPU hardware, including RAM, cache, processors, and registers to be directly accessed using a set of extensions to the C++ language.

CUDA provides a platform that allows significant performance improvements in image processing applications [3]. Its introduction has led to rapid growth in research. Examples of recent work utilizing this model include biomedical image analysis on clusters of GPUs [4], real-time multiple face tracking [5], and fast CT reconstruction [6]. Furthermore, Allusse et al. have developed an open-source GPU-accelerated framework for image processing and computer vision called GpuCV [7] to facilitate continued development on the CUDA platform. Using the parallelism inherent in the CUDA model, speedups of more than a factor of 100 over CPU implementations have been reported.

1.3 Approaches to Optical Flow

Optical flow methods have played an important role in computer vision research for more than two decades. Approaches to measuring optical flow are grouped into four main categories: variational, correlation-based, energy-based, and phase-based. A comprehensive review of approaches to optical flow is given by Barron et al. [8]. These methods can be summarized as follows:

- Variational: Variational techniques compute the optical flow by minimizing an energy functional. Starting with the original approach by Horn and Schunck [9] and Lucas and Kanade [10], these techniques have expanded to include ever more sophisticated constraints. These include discontinuities in the flow field [11] and coarse-to-fine strategies [12, 13].
- Correlation-based: These techniques use a region-based matching approach. A similarity measure is maximized to find the correct displacement between successive image frames. Correlation-based approaches were first investigated in [14] and expanded by Camus [15] to include multi-frame matching.
- Energy-based: The optical flow in energy-based methods is calculated in frequency space by considering the output of several filters, each tuned to a different spatial scale. The response of these filters will ideally be concentrated about a plane in frequency space [16].
- Phase-based: The generalized use of phase-based techniques was developed by Fleet and Jepson [17], expanding upon earlier work by Waxman et al. [18] and Buxton and Buxton [19]. In their approach, band-pass velocity-tuned filters are

used to decompose the input signal according to scale, speed, and orientation. Derivatives of the phase are then computed and used to estimate the velocity.

Correlation-based optical flow is used in this work since this method is less computationally expensive, is less susceptible to noise, and can generally detect larger velocities than other methods. Block-matching methods similar to correlation-based optical flow are also ubiquitous in video codecs, including MPEG [20], increasing the applicable scope of this work.

While research into improved and more accurate optical flow techniques has continued, a seperate stream of research has focused on accelerating optical flow. On standard CPUs, optical flow algorithms for time-critical applications have typically required high-end workstations [21] or small image sizes and approximation methods [15]. Recent work has explored the use of multi-grid methods to accelerate optical flow computation on standard PCs [22]. Their work was able to achieve frame rates of 18 fps on 316×252 resolution images. However, calculation of optical flow in real-time on PCs remains impractical for large image sizes.

Alternate research paths have focused on introducing custom parallel architecture in FPGA hardware. This includes the work reported by Diaz et al. [23], Wei et al. [24], and Maya-Rueda et al. [25], among others. These approaches have relied on two key ideas: effectively pipelining the image processing system and exploiting the parallel processing architecture of modern FPGAs. The custom architecture approach can have disadvantages, including a lack of scalability, the large amount of engineering time required to create the appropriate logic, difficulty in changing the implementation rapidly, and cost.

To overcome these challenges, several researches have used GPU technology to construct optical flow algorithms. An implementation of optical flow using GPU shader technology has been studied [26], but was limited by the structure of the graphics pipeline. An implementation of variational optical flow on a GPGPU has also been reported using multi-grid methods [27]. Very recently, implementations of variational (Horn and Schunck) optical flow in CUDA has been reported [28], although without foveation and not with correlation-based methods.

1.4 Approaches to Foveation

The computational speed of optical flow can be increased further by considering foveation, or space-variant vision. Foveated vision emulates biological vision systems by maintaining a high resolution region fixed on areas of interest and a lower resolution periphery [29]. Applying foveation reduces bandwidth and processing requirements in image processing algorithms compared to uniformly sampled images. By effectively sub-sampling data, foveation has the potential to further increase performance gain in a parallel implementation of optical flow on the GPU.

A thorough review of foveation methods is given by Yeasin and Sharma in [29]. Approaches to computing synthetic foveated images stem from an original exploration of the human visual system by Schwartz [30]. Since then, research into foveated vision in image processing has split along three paths: computing foveation transforms in software [31], computing foveation transforms in hardware (i.e. FPGA and ASIC) [32, 33], and the development of custom CCD and CMOS image sensors with non-uniform resolution [34, 35]. Due to the difficulty in obtaining or manufacturing custom image sensors, most researchers have adopted the first two methods.

Optical flow techniques with foveation have been explored, including a re-formulation of the classic variational method [36, 37]. Foveation for the purposes of tracking has been studied extensively, including work by Kang and Lee integrating foveation and optical flow to track moving objects in surveillance applications using standard PCs [38]. Kang and Lee also accounted for the deformation that occurs in space-variant coordinate systems [38].

1.5 Summary of Contributions From This Thesis

The work in this thesis builds upon ideas from previous optical flow and foveation systems as described above. A novel parallel mapping of correlation-based optical flow to CUDA architecture is described, together with a parallel foveation system. The effects of the optical flow search parameters on performance are characterized. This is followed by a characterization of the error performance on a number of synthetic and real-world video sequences.

A precise list of contributions of this thesis include:

- A mapping of correlation-based optical flow to the CUDA architecture
- A mapping of a foveation algorithm to the CUDA architecture
- A hybrid GPU/CPU method to move the fovea to points of interest in a video sequence
- Characterization of the performance of the non-foveated optical flow, foveation, and foveated optical flow algorithms for a variety of image resolutions and search parameters

- Characterization of the error performance of the above algorithms on synthetic video sequences with a comparison to the ground truth
- Testing of the above algorithms on a real-world video sequence
- Implementation of a closed-loop controller using the optical flow algorithm to control the position of a robot arm

1.6 Organization of Thesis

This thesis is organized as follows. In Chapter 2, a review of optical flow and foveation theory is given. A description of the CUDA architecture and parallel mappings of the optical flow and foveation algorithms to CUDA are discussed in Chapter 3. Details of the experimental methodology is provided in Chapter 4, followed by presentation and discussion of the results. The thesis is concluded with a summary in Chapter 5. Two sets of appendices discuss further details about control theory and robot mechanics, as well as the implementation of the robotic experiment discussed in Chapter 4.

Chapter 2

Optical Flow and Foveation Techniques

This chapter begins by presenting a brief overview of optical flow techniques. This is followed by a more detailed discussion of the correlation-based optical flow method. Next, foveated vision is discussed and the chapter concludes with a detailed account of the log-polar transform, a common technique for producing foveated images.

2.1 Optical Flow

2.1.1 Overview of Optical Flow

When an object moves in the field of view of a camera, its motion causes a corresponding change in the image. To illustrate this effect, conside the point P, moving with velocity $\vec{v}$, in Figure 2.1. The projection of P onto the image plane, P_i, will

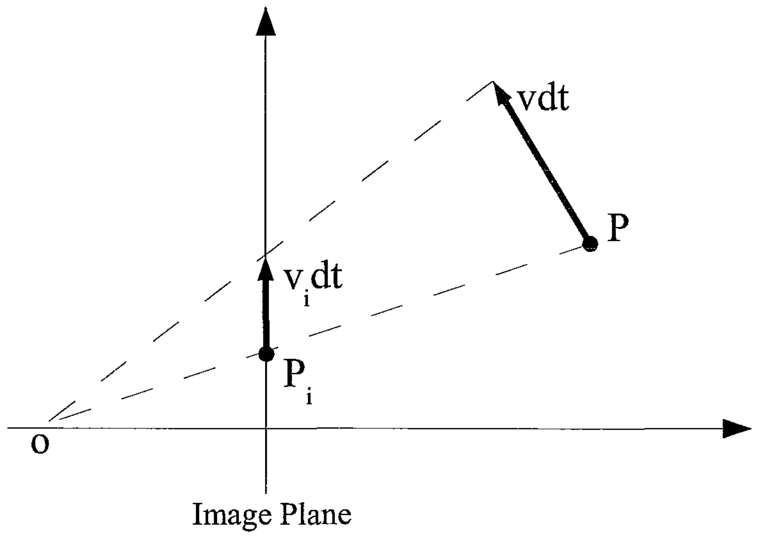


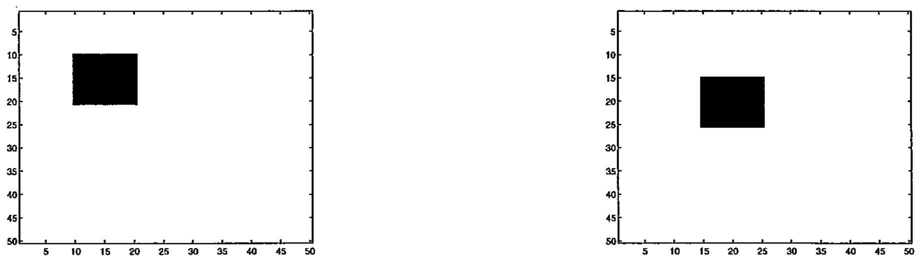
Figure 2.1: Projection of a point and motion-vector to the image plane.

therefore move by an amount given by the vector $\vec{v_i}$. The collection of all such projected vectors in the image forms the motion field. The recovery of the motion field from a set of images is called the *optical flow*. A representative optical flow field, resulting from translating 2-D squares, is shown in Figure 2.2.

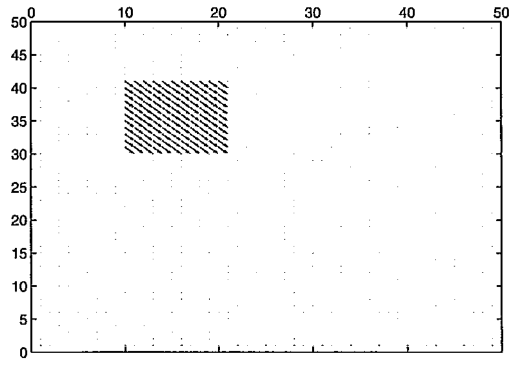
Several assumptions must be made when calculating the optical flow. The first is the *brightness constancy* assumption, which states that the intensity of local regions in the image plane are approximately constant between video frames. This can be expressed more formally as [39]:

$$I(\vec{x}, t) \approx I(\vec{x} + \partial\vec{x}, t + \partial t) \tag{2.1}$$

Where $I(\vec{x}, t)$ is the image intensity function and $\partial\vec{x}$ is the displacement of an image region at time t.



(a) First Frame (b) Second Frame



(c) Optical Flow

Figure 2.2: Optical flow field for a simple translating square

Second, for optical flow to correspond exactly to image motion, objects in the field of view must have Lambertian surface reflectance[1] and their motion must be pure translational motion parallel to the image plane. While these conditions are rarely satisfied in a full scene, they are often locally satisfied. The degree to which these assumptions are met partly determines the accuracy of the resulting optical flow field [39].

Once computed the optical flow field has numerous applications. This includes

[1]The apparent brightness of light scattered by a Lambertian surface, as seen by an observer, is the same regardless of the observer's viewing angle.

motion estimation, motion segmentation, structure from motion, and video compression, among others. Optical flow is also widely used in robotics for collision detection, visual odometry, and navigation [40]. Optical flow for navigation has also recently garnered wide attention for navigation purposes in Unmanned Aerial Vehicles (UAVs) [41]

2.1.2 Variational Optical Flow

Variational methods for calculating optical flow remain popular in the computer vision community and continue to be an active area of research. The original classic algorithm proposed by Horn and Schunck is detailed in this section, followed by a discussion of the computationally simpler correlation based methods and the reasoning for choosing correlation-based methods for CUDA implementation in this work.

2.1.3 Horn and Schunck Method

The Horn and Schunck method is an iterative variational technique. The key assumption of the technique is that the optical flow varies *smoothly.* Therefore, this method is not optimal for finding flow patterns with discontinuities. To find the optical flow vectors, a term defining the smoothness error is formulated and minimized. The first step in the derivation of the method begins by considering the brightness of a point in an image at time t, denoted by $E(x, y, t)$. As the image moves, the brightness of each point is held to be constant. This is known as the brightness constraint. This can be expressed as:

$$\frac{dE}{dt} = 0 \tag{2.2}$$

By applying the chain rule to the above expression [9], an expression relating the derivative of E to changes in x and y is obtained.

$$\frac{\partial E}{\partial x}\frac{dx}{dt} + \frac{\partial E}{\partial y}\frac{dy}{dt} + \frac{\partial E}{\partial t} = 0 \tag{2.3}$$

Allowing $u = dx/dt$ and $v = dy/dt$ and E_x, E_y, and E_t equal the three partial derivatives, the above equation can be rewritten as a linear equation with two unknowns, u and v [9].

$$\epsilon_e = E_x u + E_y v + E_t = 0; \tag{2.4}$$

The next step is to define a term that measures the smoothness of the optical flow. The smoothness can be characterized by the square of the magnitude of the gradient of the optical flow [9]

$$\epsilon_c^2 = \left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2 \tag{2.5}$$

The error to be minimized can then be expressed as [9]:

$$\epsilon^2 = \int\int(\alpha^2\epsilon_c^2 + \epsilon_e)dxdy \tag{2.6}$$

In the above expression, α^2 is a regularization parameter. Applying the calculus of variations, the above expression can be rewritten as [9]:

$$E_x^2 u + E_x E_y v = \alpha^2 \nabla^2 u - E_x E_t \tag{2.7}$$

$$E_x E_y u + E_y^2 v = \alpha^2 \nabla^2 v - E_y E_t \tag{2.8}$$

Horn and Schunck simplify the above equations by introducing an approximation of the laplacian given by [9]

$$\nabla^2 u \approx \kappa(\bar{u}_{i,j,k} - u_{i,j,k}) \tag{2.9}$$

$$\nabla^2 v \approx \kappa(\bar{v}_{i,j,k} - v_{i,j,k}) \tag{2.10}$$

The local averages, $\bar{u}$ and $\bar{v}$ are computed as a weighted sum of the neighbouring points:

$$\bar{u}_{i,j,k} = \frac{1}{6}(u_{i-1,j,k} + u_{i,j+1,k} + u_{i+1,j,k} + u_{i,j-1,k}) + \frac{1}{12}(u_{i-1,j-1,k} + u_{i-1,j+1,k} + u_{i+1,j+1,k} + u_{i+1,j-1,k}) \tag{2.11}$$

$$\bar{v}_{i,j,k} = \frac{1}{6}(v_{i-1,j,k} + v_{i,j+1,k} + v_{i+1,j,k} + v_{i,j-1,k}) + \frac{1}{12}(v_{i-1,j-1,k} + v_{i-1,j+1,k} + v_{i+1,j+1,k} + v_{i+1,j-1,k}) \tag{2.12}$$

Using the Laplacian approximation, the equations to be optimized can be rewritten as:

$$(\alpha^2 + E_x^2)u + E_x E_y v = (\alpha^2 \bar{u} - E_x E_t) \tag{2.13}$$

$$E_x E_y u + (\alpha^2 + E_y^2)v = (\alpha^2 \bar{v} - E_y E_t) \tag{2.14}$$

The above equations can be solved by using the method of Lagrangian multipliers. The full derivation is not shown here, however the interested reader is directed to the original paper by Horn and Schunck [9] for a complete treatment. The final solution then takes the form:

$$(E_x^2 + E_y^2)(u - \bar{u}) = -E_x(E_x\bar{u} + E_y\bar{v} + E_t) \tag{2.15}$$

$$(E_x^2 + E_y^2)(v - \bar{v}) = -E_y(E_x\bar{u} + E_y\bar{v} + E_t) \tag{2.16}$$

This equation is prohibitively expensive to solve directly, therefore iterative methods are used. The above equations can be discretized using the Gauss-Siedel method, which gives the following iterative solution [9]:

$$u^{n+1} = \bar{u}^n - E_x(E_x\bar{u}^n + E_y\bar{v}^n + E_t)/(\alpha^2 + E_x^2 + E_y^2) \tag{2.17}$$

$$v^{n+1} = \bar{v}^n - E_y(E_x\bar{u}^n + E_y\bar{v}^n + E_t)/(\alpha^2 + E_x^2 + E_y^2) \tag{2.18}$$

The Horn and Schunck method in the above form can be used to compute the dense optical flow field of a sequence of images. The next section details a computationally less expensive, but less accurate method: correlation-based optical flow.

2.1.4 Correlation-Based Optical Flow

In correlation-based optical flow, the similarity between a block around each pixel in the current video frame, $I(t)$, and a series of blocks in the next frame, $I(t + dt)$, is maximized to find the correct pixel motion. More precisely, a $(2\nu+1) \times (2\nu+1)$ sized block of pixels in the current frame, $I(t)$, centred at pixel (x, y) is extracted. This pixel block is designated as P_1. In the next frame, $I(t + dt)$, there are $(2n + 1) \times$

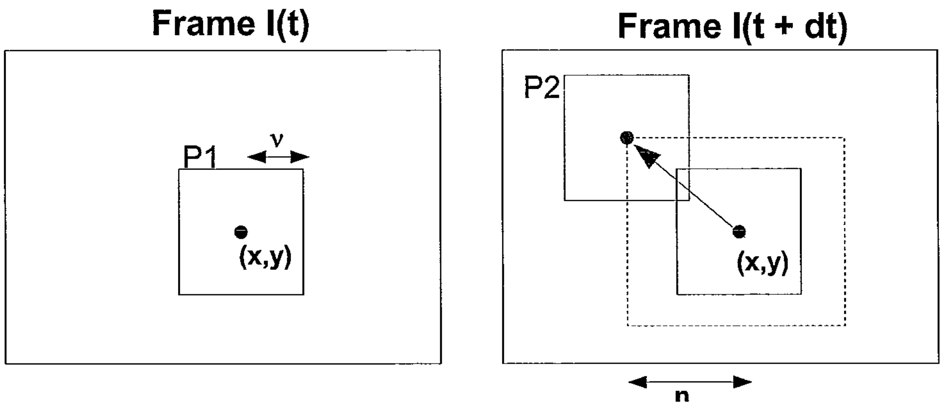


Figure 2.3: Illustration of correlation-based optical flow

$(2n + 1)$ possible displacements that the pixel, (x, y), could have moved. For each displacement, a $(2\nu + 1) \times (2\nu + 1)$ sized pixel block, designated as P_2, is extracted from frame $I(t + dt)$ and compared to P_1.

The pixel patch P_2 that maximizes the similarity measure corresponds to the displacement of the pixel between the two frames. The parameter ν corresponds to the correlation window radius and n is designated as the search window radius. This procedure is illustrated in Figure 2.3.

Normalized cross-correlation and sum-of-squared differences (SSD) are conventional choices for similarity measure. SSD was chosen in this study due to its lower computational cost. The SSD match strength is calculated as:

$$\begin{aligned} M(x, y, dx, dy, t) = \sum_{j=-v}^{v} \sum_{i=-v}^{v} (I(x + i, y + j, t) \\ -I(x + d_x + i, y + d_y + j, t + dt))^2 \end{aligned} \tag{2.19}$$

Where (x, y) is the image coordinate in frame $I(t)$ and (d_x, d_y) are the set of $(2n+1) \times (2n+1)$ possible displacements around (x, y). The pixels displacement is taken to be the set of displacements (d_x, d_y) that minimize the SSD.

2.1.5 Horn and Schunck versus Correlation Methods

Variational techniques, such as the Horn and Schunck method, have unique advantages and disadvantages for optical flow computation compared to correlation based techniques. First is speed of computation. Variational techniques tend to be more computationally expensive than correlation based techniques. The Gauss-Siedel method, especially, can converge slowly. Although faster numerical techniques have been applied to the problem [22], more computation is still required than with correlation-based methods. Since the goal of this work is to develop the fastest possible optical flow implementation, suitable for time-critical tasks, the use of correlation-based methods is an advantage in meeting this goal.

Variational techniques, however, are more accurate than correlation-based techniques. Barron et al. [8] tested the accuracy of a multitude of optical flow techniques on synthetic video sequences. They found that for the standard test sequence 'Yosemite flythrough', the Horn and Schunck method calculated the optical flow vectors with an average angular error (AAE) of approximately 2 degrees, while the correlation-based method had an average angular error of approximately 14 degrees. However, the correlation method can be improved by adding interpolation to the calculation. Accuracy in the determination of the optical flow vectors was not the prime concern of this work, however the structure of the developed optical flow kernel, discussed in Chapter 3, is amenable to replacing the correlation method with a variational technique if more accuracy is desired.

Variational techniques are not well-suited to calculating optical flow vectors with larger velocities, where pixels can move several units between frames. This problem is solved by using a coarse-to-fine image pyramid, however this adds even more computational complexity. Correlation-based methods, on the other hand, are naturally suited to finding larger velocities simply by increasing the search window radius, n. Another important advantage of correlation-based methods is performance with noise. Overall, correlation techniques are less susceptible to noise in the image than variational techniques. This was an important factor in this work since the overall goal of the project was to develop a system capable of working on live video feeds, which can often have significant noise in the input. For the above reasons, correlation-based methods were chosen over variational methods.

2.2 Foveation

2.2.1 Overview of Foveation

The human retina contains two distinct regions in terms of resolving power. The retina has a high-resolution region, the fovea, and a lower resolution periphery. This structure results from the fact that photoreceptors (rods and cones) are non-uniformly distributed in the retina [42, 43]. The density of cones plays an important role in determining the resolving power of the human eye.

When a human observer gazes at a scene, the region around the eye's point of fixation (or foveation point) is projected onto the fovea and sampled with the highest density. The sampling density decreases rapidly with distance from the fovea [42, 43].

As a result, a variable resolution image is transmitted to the high level processing centres of the brain.

The human visual system is therefore space-variant in acquiring and processing information. This structure is common in biological systems and allows organisms to maintain a wide field of view while focusing the highest-resolution part of the eye on regions of interest [30]. This is in stark contrast to traditional digital imaging systems that sample images on a regular, rectangular grid. This approach has typically been taken due to its intuitivness and simplicity of image acquisition, storage, transmission, and computation.

Foveation, or space-variant vision, can be emulated by image processing systems. Foveation reduces the data bandwidth and processing requirements [32] for vision algorithms, making it an attractive approach for computationally intensive applications.

2.2.2 Approaches to Foveated Vision

In the human visual system, foveated vision is part of the structure of the eye [30]. Such a structure is challenging to implement in CCD and CMOS sensors due to the nature of VLSI semiconductor processing, which is suited to rectangular, regular features [32]. Space-variant sensors have been manufactured [34, 35], but remain more expensive and less readily available than commercial, off-the-shelf, digital cameras. For this reason, most foveated vision systems implemented in the literature utilize a camera with uniform resolution and convert the data to a foveated image with software or hardware algorithms [31]. Two types of foveal mappings are commonly used:

- Log-Polar [30]: A cartesian image is mapped into a series of concentric rings, each with decreasing resolution inversely proportional to the distance from the fovea.
- Exponential Cartesian [44]: A cartesian image is mapped into uniformly shaped rexels. The rexels in successive rings are related in size by a power of two.

Log-polar mapping is widely used in the literature, as well as this study, since it has the advantage of being rotation and scale-invariant as well as maintaining similarity the biological retinal model.

2.2.3 The Log-Polar Transform

The log-polar transform (LPT) is a conformal mapping from cartesian space, (x, y), to log-polar space, (u, θ), given by [31]:

$$\rho = \sqrt{(x - x_{fov})^2 + (y - y_{fov})^2} \tag{2.20}$$

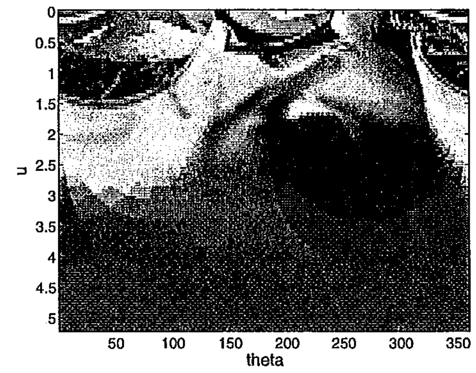
$$u = \ln(\rho/\rho_o) \tag{2.21}$$

$$\theta = \arctan(\frac{y - y_{fov}}{x - x_{fov}}) \tag{2.22}$$

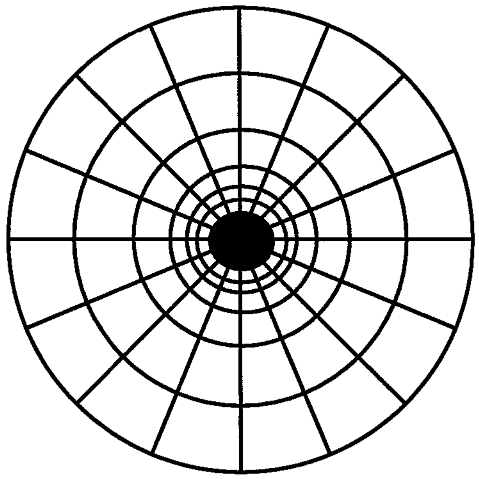
In the above set of equations, ρ is the radius, ρ_o is the radius of the fovea, u is the log scale of the radius, (x, y) are the cartesian image coordinates, and (x_{fov}, y_{fov}) are the coordinates of the center of the fovea. The topology of the log-polar fovea is shown in Figure 2.4(c), as well as a representative example of a log-polar transform image in Figure 2.4(a) and (b), respectively.

The log-polar transform is widely used in part due to its favourable properties. As mentioned earlier, the LPT is scale and rotation invariant. This property is illustrated

(a) Cartesian Image



(b) Log-Polar Transform image



(c) Foveal Topology

Figure 2.4: a) Original cartesian image. b) Log-polar transform of a). c) Illustration of log-polar foveal topology.

in Figure 2.5, which shows the scaling and rotation of a bar [29]. Rotation of an object results in a shift in θ in the transform and a change in object scale results in a shift in u.

To verify these properties, consider scaling an image by a factor, S, which can be written as:

$$\rho \exp^{j\theta} \rightarrow S\rho \exp^{j\theta} \tag{2.23}$$

Applying the LPT yields:

$$\log(S\rho \exp^{j\theta}) = \log S + \log \rho + j\theta \tag{2.24}$$

Similarly, adding a rotation of angle α, which can be written as:

$$\rho \exp^{j\theta} \rightarrow \rho \exp^{j(\theta+\alpha)} \tag{2.25}$$

Applying the LPT yields:

$$\log(\rho \exp^{j(\theta+\alpha)}) = \log \rho + j(\theta + \alpha) \tag{2.26}$$

The scale and rotation invariance properties of the LPT are an advantage in several situations. They simplify the calculation of radial optical flow of approaching objects, which can be exploited to quickly estimate the time to impact. These properties also simplify shape and object recognition since shape and orientation variance problems are eliminated [29].

The primary disadvantage of the log-polar transform is that it complicates image

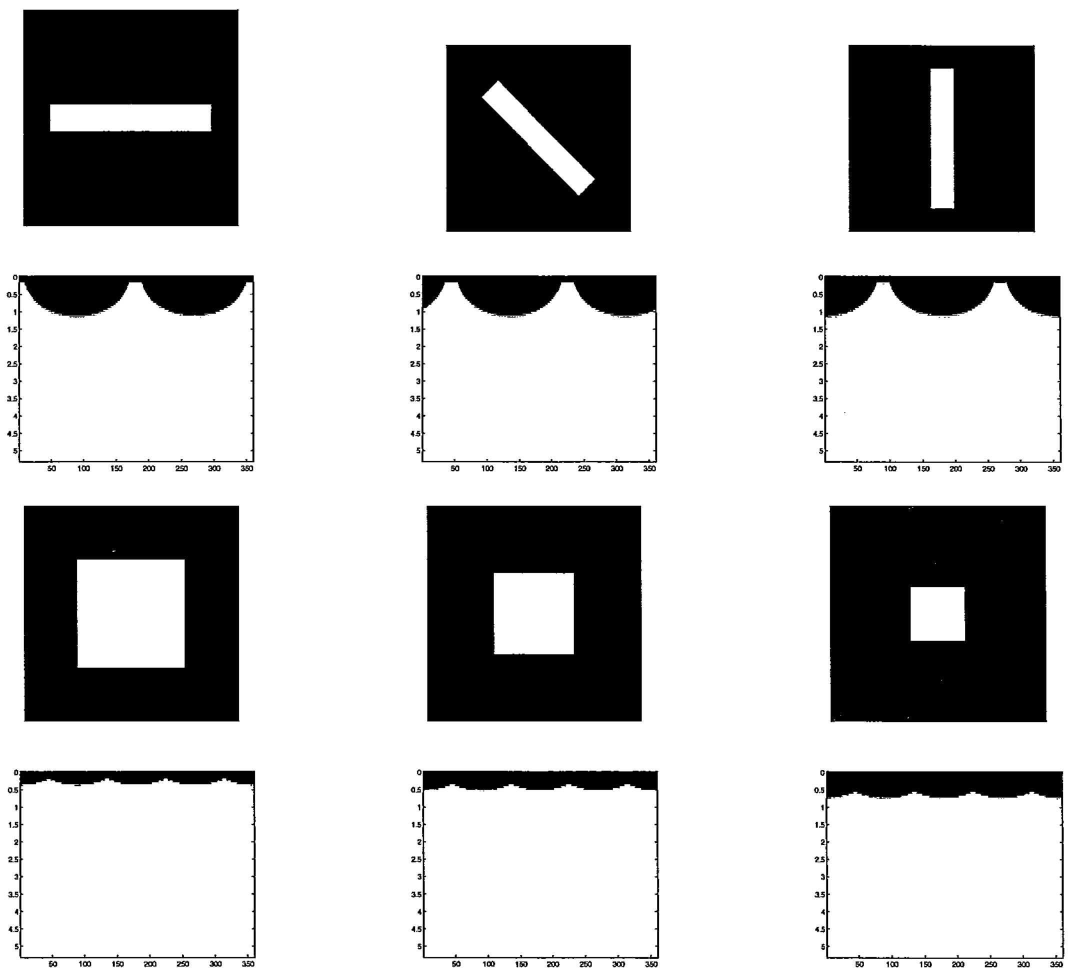


Figure 2.5: Illustration of rotation and scale invariance properties of the log-polar transform

translation. In general, a simple translation in cartesian space results in a deformation in log-polar space [29]. This result breaks the spatial neighbourhood structure required by many traditional image processing techniques. The need to generate new image processing methods has slowed the acceptance of the foveated vision model.

2.3 Optical Flow in the Log-Polar Domain

Due to the translation property of the log-polar transform discussed above, the optical flow is more challenging to compute in log-polar space. One must be aware of the deformation that occurs during translation when analyzing the movement of a region in a space-variant coordinate system. An illustration of log-polar deformation resulting from a cartesian translation is shown in Figure 2.6.

The opposite effect seen in Figure 2.6 occurs when a rectangular search window in log-polar space is used with correlation-based optical flow. Shifting a rectangular window in log-polar space will result in an annular segment shifting along the (r, θ) direction in cartesian space.

A motion vector found using optical flow in log-polar space will have components of $(du, d\theta)$. Vectors in this coordinate system are difficult to use for most applications, such as motion segmentation or comparison to a ground-truth. Following the approach of Kang and Lee [38], this difficulty is solved by transforming each motion vector at each (u, θ) coordinate to cartesian coordinates. The transformation begins by obtaining the radius, r, from the initial, u_o, and displaced, u_f, log radius values:

$$r_o = \rho_o \exp(u_o) \tag{2.27}$$

$$r_f = \rho_o \exp(u_f) \tag{2.28}$$

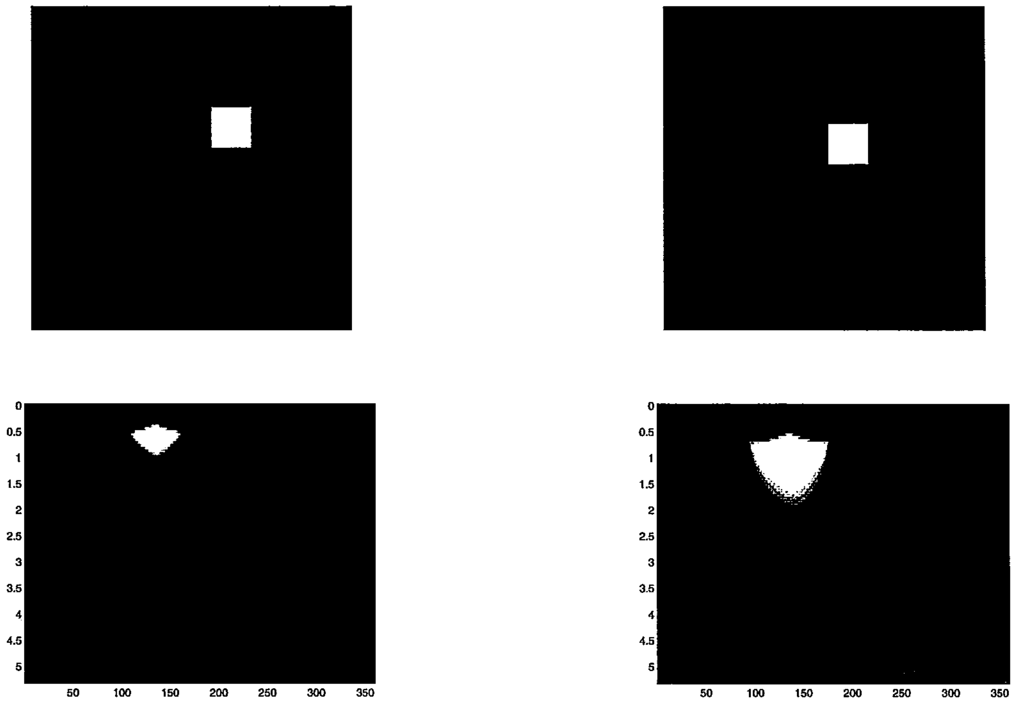


Figure 2.6: Illustration of deformation in the log-polar space (bottom) when translating an object in cartesian space (top).

The initial and final (x, y) components are then calculated as shown below:

$$x_o = r_o \cos(\theta_o) \tag{2.29}$$

$$x_f = r_f \cos(\theta_f) \tag{2.30}$$

$$y_o = r_o \sin(\theta_o) \tag{2.31}$$

$$y_f = r_f \sin(\theta_f) \tag{2.32}$$

Vectors in this form are now ready to be utilized in subsequent processing steps.

Chapter 3

Algorithm Implementation on Compute Unified Device Architecture

In this chapter, an overview of the Compute Unified Device Architecture and the constraints it places on algorithm design are discussed. Details of the mapping of foveation, optical flow, and foveation point selection algorithms to CUDA follow. The chapter concludes with an overview of the overall system design used to conduct the experiments in Chapter 4.

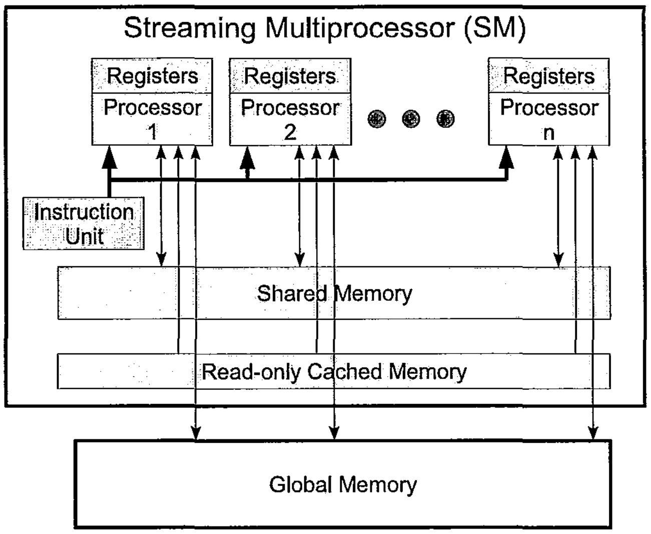


Figure 3.1: Diagram of CUDA architecture organization. Figure courtesy of Michael Kinsner.

3.1 Overview of Compute Unified Device Architecture (CUDA)

The modern GPU is a multi-core, single-instruction multiple thread (SIMT) processor designed for fast floating point calculation. The architecture in NVIDIA Geforce GPUs consists of a multitude of multiprocessors that each encapsulate a set of processors, on-chip shared memory, and an instruction unit, as illustrated in Figure 3.1. Each processor has access to on-board RAM (global memory) and on-chip shared memory. The GPU used in this work contains 20 multiprocessors, each with 12 processor cores, for a total of 240 SIMT processors [1]. SIMT architecture is similar to single-instruction multiple-data (SIMD), with the added ability to execute either single scalar threads or cooperative parallel threads [1].

3.1.1 GPU versus CPU Organization

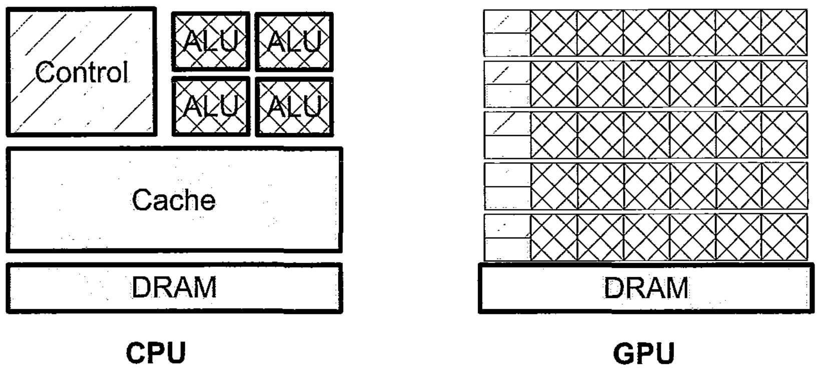


Figure 3.2: GPU versus CPU architecture

The GPU is specially suited to perform computationally-intensive, highly parallel tasks. Its capabilities have evolved from the need to render graphics at high frame rates for the gaming market. The GPU accomplishes this thanks to an architecture that devotes more transistors to data processing rather than data caching or controlling program flow [1]. A schematic of the difference between GPU and CPU architecture is shown in Figure 3.2. GPU architecture makes the device best suited to algorithms that access data elements in parallel with a high ratio of arithmetic to memory operations (arithmetic intensity). This requirement stems from a lack of extensive control logic and cache structures.

3.1.2 GPU Programming Model

Each multiprocessor manages and executes instructions in sets of 32 threads. Each set is called a *warp*. All threads within a warp start at the same program address, but are then free to branch and execute independently of one another. The GPU is designed to execute thousands of lightweight threads with minimal overhead, which allows very finely scaled parallelism to be built into programs, for example, launching a thread for each data element to be processed [1].

Threads launched on the GPU are organized into groups called thread blocks, with a maximum of 512 threads per block. Each warp executes the same common instruction. The most efficient case occurs when all threads in a warp follow the same execution path. When threads in a warp diverge (as occurs during a condition statement), the warp serially executes each possible branch. The threads not active during the branch are disabled. Only when all paths have been completed do all the threads converge and continue on the execution path. Therefore, divergence within a warp slows computation speed [1].

Blocks of threads in general cannot communicate with one another. Only threads in the same block can communicate by using shared memory. The next unit of organization in CUDA is called a grid. A set of thread blocks forms a *grid.* CUDA programs that launch a grid of thread blocks are called *kernels.* To simplify algorithm design, threads can be launched in one-dimensional, two-dimensional, or three-dimensional thread blocks. This simplifies indexing when working with elements of vectors and matrices.

3.1.3 Calling Kernels

An example of a CUDA kernel invocation is shown in the source code listing below. The kernel in this example is called *logpolartrans_gpu.* Calling a kernel is very similar to calling a regular C++ function with the extra addition of two parameters within the <<<>>> brackets. The first parameter, *gridSize* specifies the number of thread blocks to launch while *blockSize* specifies the number of threads in each block.

```
logpolartrans_gpu <<<gridSize, blockSize>>> (arguments);
```

CUDA includes additional types to facilitate the creation of 1D, 2D, and 3D blocks and grids. For example a thread block with 16×16 threads and a grid with 128×128 blocks can be created with the variables:

```
dim3 blockSize (16, 16);
dim3 gridSize (128, 128);
```

A CUDA kernel is declared in a similar manner to a standard C++ function. For example, the log polar transform kernel invoked above would be declared as:

```
// Kernel declaration
__global__ void logpolartrans_gpu(float *d_cart, float *d_log, int w, int padsize, unsigned long
    logSize, float fovea_cx, float fovea_cy, float rho_o, float u_min, float du, int numRings)
{
        (Kernel code)
}
```

A CUDA kernel is therefore specified by using the *__global__* identifier. It should be noted that all global CUDA functions, as shown above, must be declared as void. In order to return values back to the host, the memory must be manually copied from the GPU's RAM to system RAM using a memcpy operation.

Each of the blocks and threads executed by a kernel is given its own block ID and thread ID. These are stored in 3-component vectors stored as built in variables named *blockIdx* and *threadIdx*, respectively. The thread and block IDs are used inside the kernel to determine what data each thread should operate on. As an example, consider a kernel that performs a simple thresholding operation on a 2D image. Each thread should access a single pixel in the image, compare it to a threshold, and write the thresholded pixel back to memory. To solve this problem, a 2-D thread and block structure can be used to simplify addressing. I.e.:

```
__global__ void threshold(float* d_inputIMG[W][H], float* d_outputIMG[W][H], int threshold)
{
        int i = blockDim.x * blockIdx.x + threadIdx.x;
        int j = blockDim.y * blockIdx.y + threadIdx.y;

        if (d_inputIMG[i][j] > threshold)
                d_outputIMG[i][j] = d_inputIMG[i][j];
}

void main(void) {
        dim3 gridSize (40,30); // process a 640x480 image
        dim3 blockSize (16,16);

        // Kernel invocation
        threshold <<<gridSize, blockSize>>> (d_inputIMG, d_outputIMG, threshold);
}
```

In the above example, *blockDim* is a 3 component vector that specifies the size of each thread block. The values of the parameters in the above example were chosen arbitrarily to illustrate the concept of thread indexing.

3.1.4 Algorithm Design Constraints

The GPU architecture imposes unique constraints on the design of algorithms. To maximize use of the available processors and hide memory access latency, CUDA algorithms must use a large number of threads per block (> 192) and a large number of blocks (> 64). Resource usage per thread must be low, due to the limited number of available registers and shared memory (16 KB per multiprocessor). The thread block size should also be a multiple of the warp size (i.e. a multiple of 32) [1].

3.1.5 CUDA Memory Access

CUDA provides several types of memory access methods: global memory, shared memory, texture memory, and constant memory. Global memory refers to the off-chip RAM on the graphics card. Global memory access is not cached and there is a 300 clock-cycle latency between requesting and receiving data. Global memory access can be sped up by coalescing access, which allows a half-warp of threads to read a block of memory simultaneously (i.e. in one transaction) [1].

In order for coalesced memory access to occur, a half-warp of threads must read a continuous block of memory. Furthermore, the start address of the read has to be a multiple of the warp size. For example, a half-warp reading 16 continuous memory locations beginning at memory address 128 would result in coalescing, but if the read began at address 129, coalescing would no longer occur and the memory would be read out in 16 transactions. Coalescing will still occur even if not all threads read a memory location, as long as the alignment requirements are met. Memory coalescing is illustrated in Figure 3.3.

Shared memory is on-chip, and is therefore much faster than global memory. Shared memory can be accessed as quickly as a register, and also allows threads in the same block to co-operate [1]. These properties make shared memory useful for storing commonly accessed data, bypassing the latency associated with global memory. Unfortunately, if shared memory access is structured incorrectly, threads in a warp can encounter memory bank conflicts. This results in serialized threads. Shared memory organization is detailed below.

Shared memory is organized into equally-sized memory modules, designated as memory banks. This organization is used to achieve high-memory bandwidth, since

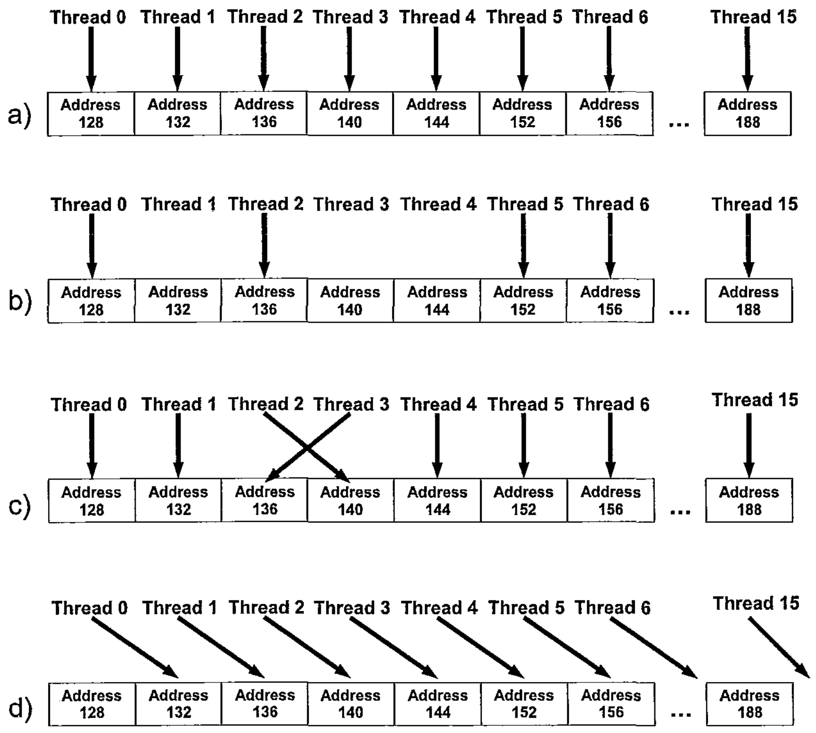


Figure 3.3: Global memory coalescing access patterns. a) and b) result in coalesced access while c) and d) do not.

different memory banks can be accessed simultaneously. A memory-bank conflict occurs when two or more memory requests access the same bank. When this occurs, the control hardware splits the memory request into as many conflict-free requests as necessary, effectively serializing access and reducing memory bandwidth. When data is stored in shared memory, consecutive 32-bit words are assigned to successive memory banks. Therefore, if each thread in a warp accesses a different array element of type float, bank conflicts will not occur. Fortunately, multiple threads can access the same memory bank under certain conditions, thanks to a broadcast mechanism.

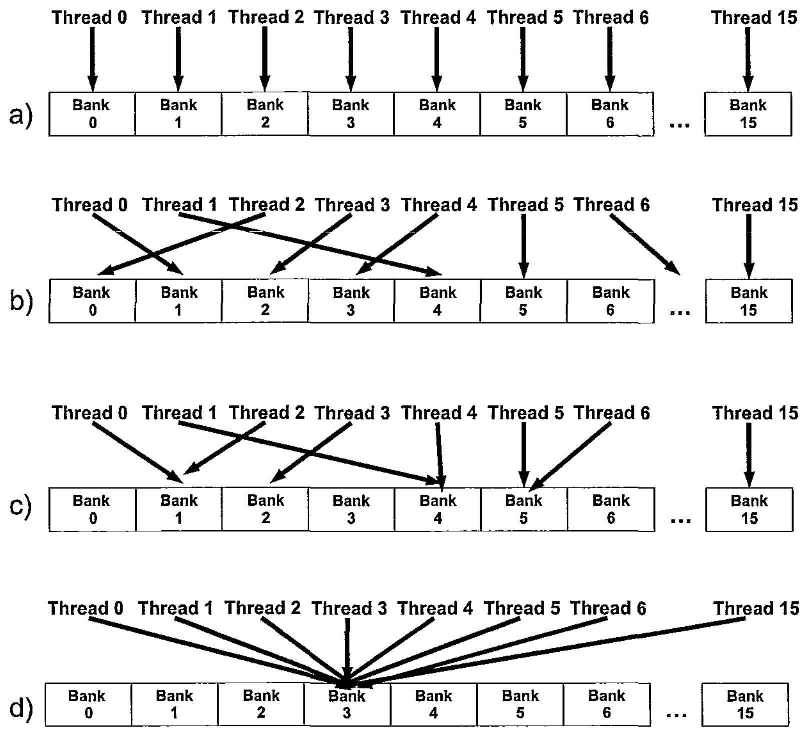


Figure 3.4: Shared memory access without bank conflicts (a and b) and with bank conflicts c). d) is an example of broadcasting, which does not result in a bank conflict

Broadcasting allows a 32-bit word to be read and broadcast to several threads simultaneously with one memory request [1]. Examples of conflict-free and conflicted shared memory access are shown in Figure 3.4.

CUDA provides two additional cached, read-only, methods to access global memory without coalescing: constant memory and texture memory. These methods are useful since structuring data access to meet coalescing requirements is often not trivial. Constant memory is cached, therefore a half-warp of threads reads constant memory as quickly as if it was a register, unless a cache miss occurs. Texture memory is likewise cached, where a read from texture memory only accesses global memory on a cache miss, otherwise the data is read from the texture cache. The texture cache is organized for data that has 2D spatial locality [1], making it particularly suitable for image processing algorithms.

The above advantages and constraints must be kept in mind when mapping algorithms to the CUDA platform. In general, the best performance is achieved when the parallelism of the CUDA platform is exploited for problems with sufficient arithmetic intensity. After decomposing an algorithm to a degree that can be implemented in parallel, the constraints, in terms of memory access, occupancy, register access, and divergence, must be minimized to maintain an efficient algorithm. This procedure was employed in the development of the foveation, optical flow, and foveation point selection algorithms, as detailed in the following sections.

3.1.6 CUDA Memory Programming Model

CUDA kernels are restricted to accessing memory in global and shared memory on the graphics card. In other words, they are not able to access host system memory. Device memory, in turn, must be manually allocated and copied between the host and graphics card. A full discussion of memory managment techniques is provided in the NVIDIA CUDA Programming Guide [1], however a brief introduction is provided here for completeness. The first example is allocation of 1-D linear global memory and is shown in the listing below. It is important to remember that this code will be executed by the host.

```
float* sampleArray;
cudaMalloc((void**)&sampleArray, 256 * sizeof(float)); // allocate a 256 element array
```

Once the CUDA array is allocated, memory can be copied from host to device RAM. The next example illustrates this by transferring the contents of a system array to global memory:

```
float hostArray[256];
// Perform the memory copy
cudaMemcpy(sampleArray, hostArray, 256 * sizeof(float), cudaMemcpyHostToDevice);
```

Once the cudaMemcpy operation has been completed, the data residing in global memory can be accessed by any kernels.

Unlike global memory, shared memory is allocated directly inside a kernel. A shared memory array is allocated in the same manner as a regular array, except that it is prefaced with a __shared__ identifier. An example of shared memory allocation is shown below:

```
__global__ void sharedexample(arguments)
{
        __shared__ sharedArray[256];
}
```

The above source code examples provide a brief introduction to the CUDA memory managment model. The use of each feature will be discussed in more detail in the sections below. Full source code listings for the kernels developed in this work are also provided in Appendix C.

3.1.7 Thread Scheduling

When a CUDA kernel is invoked, the designer specifies the number of threads and thread blocks to launch. However, the mapping of the threads to the available processors is handled by the hardware. In general, a global block scheduler will assign each thread block to a multiprocessor as capacity allows. The number of thread blocks that can execute on a given multiprocessor is determined by the number of threads in a block, required amount of registers per thread, and shared memory requirements. The global block scheduler issues thread blocks in a round-robin fashion [45]. The maximum number of thread blocks that can be executed by a multiprocessor is 8, and the maximum number of threads that can be executed is 1024 on the current generation of hardware.

Each multiprocessor is responsible for scheduling the execution of its assigned thread blocks on an array of FPUs, which the NVIDIA literature refers to as processors. As mentioned previously, threads are issued in groups of 32, designated as *warps*. A warp forms the fine-level parallelism in the CUDA model. Execution continues until all threads in a kernel have been executed.

3.1.8 GPU/CPU Concurrency and Streaming

CUDA programs are written using a set of extensions to the C language. This allows serially executed code, operating on the CPU, to operate concurrently with kernels launched on the GPU. In CUDA parlance, the CPU is termed the *host* and the graphics card the *device*. GPU/CPU concurrency can increase execution speed, since the CPU can operate or prepare data for a subsequent kernel while a kernel executes on the GPU. The exception occurs during memory transfers between the host and device. In this case the kernel cannot start until the memory transfer is complete [1].

CUDA does provide a mechanism to transfer data to memory concurrently with kernel execution through *streams*. A stream allows memory operations to be broken up into a sequence of operations. These operations can then execute out of order or concurrently. Unfortunately, streaming asynchronous execution is still in development and not all memory transfer operations are currently supported. As of the time of this writing, asynchronous memory transfer to texture memory is not yet supported. Since the kernels created to solve the foveated optical flow problem rely heavily on

Algorithm 1 Log-Polar Transform

```
for Pixels in parallel do
    Calculate $(\rho, \theta)$ coordinate for current thread
    $x = \rho * cosTable[\theta] + x_{fov}$ {Perform inverse LPT}
    $y = \rho * sinTable[\theta] + y_{fov}$
    Write pixel from cartesian image to log-polar image in global memory
end for
```

textures, memory streaming was not implement in this thesis.

3.2 Foveation Algorithm

To compute the log-polar transform, a thread is launched for every pixel in the LPT image, which is of size $(N\theta \times NRings)$, as shown in Figure 2.4(c). $N\theta$ corresponds to the number of angular divisions and $NRings$ corresponds to the number of radial divisions. For each (u, θ) coordinate in the LPT image, each thread calculates the corresponding cartesian coordinates, (x, y) according to the equation:

$$\rho = \rho_o \exp(u) \tag{3.1}$$

$$x = \rho \cos(\theta) + x_{fov} \tag{3.2}$$

$$y = \rho \sin(\theta) + y_{fov} \tag{3.3}$$

The thread then loads this data from global memory and stores it in the global memory that corresponds to the LPT image. To increase processing speed, the sine and cosine values are pre-computed at program start and stored in constant memory. This optimization is possible since the range of θ values used in the LPT image is known *a priori*.

The width of the LPT image is padded to be a multiple of the half-warp size

(16 threads) for faster performance of the subsequent optical flow algorithm. This is done since the optical flow kernel uses a 2-D thread indexing structure to access images. Without padding, thread blocks on the edges of the image would have to be inactive. This would lead to warp divergence and break global memory storage coalescing. These issues are more fully described in section 3.3.1.

The accuracy of the LPT image is improved by adding the option to interpolate the cartesian image when sampling image points. Interpolation is achieved by first loading the cartesian image into texture memory. The GPU allows linear interpolation to be performed by dedicated hardware on textures, increasing execution speed.

Pseudo-code for the foveation algorithm is shown in Algorithm 1. The algorithm can be summarized as follows. A thread is launched for every pixel in the LPT image. The transform is therefore computed in parallel and each thread is associated with a (ρ, θ) coordinate. Each thread then performs the inverse LPT to find the (x, y) coordinate that corresponds to its (ρ, θ) coordinate. With the (x, y) coordinate known, the thread samples the cartesian image and writes the resulting pixel to the LPT image in global memory. Full source code for the foveation kernel, along with a more detailed explanation of the code is provided in Appendix C.

3.3 Optical Flow Algorithm

3.3.1 Thread, Block, and Shared Memory Organization

To maximize execution speed of the optical flow algorithm, each image (of size $w \times h$) is divided into a 2D grid, as shown in Figure 3.5(a). Each block in the grid is assigned to a block of threads launched on the GPU. A single thread in the block

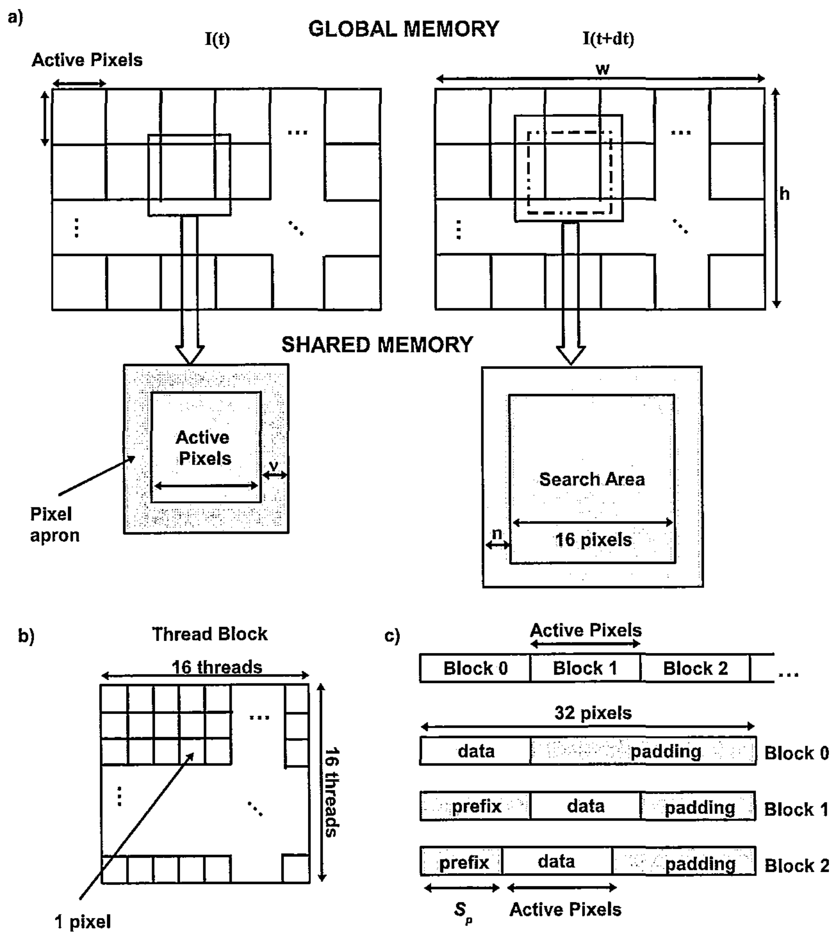


Figure 3.5: a) Diagram of image subdivison in global memory into parallel thread blocks and shared memory loading stage. b) Thread block organization. c) Description of global memory storage coalescing using a shared memory staging area.

Algorithm 2 Foveated Optical Flow

```
Allocate Shared Memory P1 [16][16]
Allocate Shared Memory P2 [16+2n][16+2n]
Allocate Shared Memory dxStorage [16][32]
Allocate Shared Memory dyStorage [16][32]
Load P1 and P2 from Global Memory
for Pixels in parallel do
  if thread corresponds to Active Pixel then
    for each pixel in P2 do
      for each pixel in P1 do
        sum SSD
      end for
      if SSD < previous SSD then
        [1] Calculate origin and end of vector in log polar coordinates
        [2] Transform to cartesian coordinates
      end if
    end for
  end if
  Store displacement in dxStorage and dyStorage
  Synchronize threads
  Write dxStorage and dyStorage to Global Memory
end for
```

computes the motion vector for a single pixel in the image, as shown in Figure 3.5(b). Since the comparison pixel patch, P_1, has a radius of ν, it is necessary to load a pixel apron in each block of size ν, as shown in Figure 3.5(a), since threads belonging to different blocks cannot communicate without resorting to global memory access. Each thread block has a set of threads that compute the motion vector, and a set that correspond to apron pixels. The threads that calculate a motion vector are called Active Pixels. Each thread block has 16×16 threads to maximize occupancy of the processors and hide memory access latency - therefore there will be $16 - 2\nu$ Active Pixels per thread block. To process a $w \times h$ sized image, $(w/ActivePixels) \times (h/ActivePixels)$ thread blocks must be launched.

Each block loads its corresponding active pixels and apron from frame $I(t)$ into shared memory, which functions as a user-defined cache. The thread block also loads the corresponding pixels from frame $I(t+dt)$. This region requires an apron of size $n+\nu$. The shared memory utilization is shown in Figure 3.5(a). It is challenging to load the two pixel sets in a coalesced manner because of their differing sizes. They are also not typically aligned by a multiple of the warp size with the images base address, which is required for coalescing memory access. To overcome this limitation, texture memory, which is cached and optimized when 2D locality is present, is used to store and access incoming frames.

Since the threads associated with apron pixels are not active during computation, there will be a percentage of idle threads, given by:

$$P_{idle} = 1 - \frac{ActivePixels}{(ActivePixels + 2\nu)} \tag{3.4}$$

where in the above equation *Active Pixels* denotes the number of Active Pixels *per row* in a given thread block (i.e., if $\nu = 2$, $ActivePixels = 12$ for a thread block with a width of 16 threads). However, since ν is significantly smaller than the thread block width, the overhead is not significant compared to the savings in memory access by caching pixels in shared memory. The shared memory is accessed in column-major order to avoid memory bank conflicts.

3.3.2 Match Strength Computation and Deformation Correction

Each thread that corresponds to an active pixel (i.e. not including apron pixels) in frame $I(t)$ calculates the displacement for that pixel by implementing Equation 2.19, for all $(2n + 1) \times (2n + 1)$ possible displacements $(du, d\theta)$. For each displacement, the match strength is evaluated. If it is smaller than the match strength of the previous displacement, the new displacement value is stored. The displacement with the smallest sum-of-squared difference is returned.

At this stage, the motion vectors are in log-polar coordinates. Prior to implementing motion-segmentation, the motion vectors must be transformed to cartesian coordinates, by applying equation 3.3. To reduce register usage and further increase execution speed, the intrinsic CUDA function, *expf*, is used instead of exp. As with the log-polar transform kernel, sine and cosine lookup tables in constant memory are utilized.

3.3.3 Output Padding

The displacement vector must be stored in global memory prior to transfer back to the host. Since texture memory is read-only, global memory coalescing must be used. Shared memory is used to coalesce memory access, as shown in Figure 3.5(c). The storage stage width is a multiple of the warp size (i.e. 32) and the height of the number of active pixels in the block. The memory write must start at an address that is a multiple of the half-warp size, 16, to the base address. Therefore, a padding prefix, S_p, is added to the pixels to be written, defined by:

$$S_p = ((bW - \nu)bW) - floor[\frac{(bW - \nu)bX}{bW}]bW \tag{3.5}$$

where bW is the thread block width and bX is the block index in the horizontal direction of $I(t)$. The remainder of the storage stage is padded with zeros to fill out the warp-size multiple requirement. Global memory is written to in two stages, each thread storing two pixels.

Pseudo-code for the foveated optical flow algorithm is provided in Algorithm 2. As described previously, the kernel begins by allocating the appropriate shared memory for each thread block. Shared memory is allocated for the pixel blocks P_1 and P_2 as well as for the x and y components of the resulting motion vectors. Once the memory is allocated, the appropriate pixels are loaded from frame $I(t)$ and $I(t - dt)$ as discussed in section 3.3.1 (i.e. the active and apron pixels are loaded).

The optical flow calculation begins with a thread launched for every pixel in the P_1 pixel patch. A total of 16×16, or 256, threads are launched, although some will be idle in the calculation since they correspond to apron pixels. Each thread then calculates a motion vector using the correlation method discussed in section 2.1.4.

Once the motion vector is found, it is transformed to cartesian coordinates and stored in shared memory. Once all motion vectors in the thread block have been found, they are written to the global memory. As discussed in section 3.3.3, the shared memory arrays dxStorage and dyStorage are used to stage a coalesced write to global memory, with each writing two pixels. Full source code for the optical flow kernel, along with a more detailed explanation of the code is provided in Appendix C.

3.3.4 Foveation Point Selection

The foveation point must be updated each frame to maintain the region of interest in the highest resolution area of the image. Since foveation is used with optical flow in this work, a strategy of maintaining the fovea over areas with the greatest movement magnitude was chosen. This is accomplished by thresholding the optical flow vectors that fall below a certain displacement magnitude and finding the centroid of the origins of the vectors that remain.

The centroid is found with a combination GPU/CPU algorithm as outlined in algorithm 3. A pre-computation step is performed on the GPU. A thread is launched for each pixel in the image, divided into blocks of 256 threads. If a thresholded pixel exists at that location, its coordinate is transformed from log-polar coordinates to cartesian. The (x, y) coordinates are stored in shared memory. At the end of the kernel, one thread from the block sums up all of the (x, y) coordinates and the number of thresholded points. This data is then returned to the CPU, which computes the final centroid. At this step, only the number of points corresponding to the number of thread blocks have to be summed to calculate the centroid.

The algorithm is split up into CPU and GPU portions to reduce the amount of data that must be transferred over the PCI-Express bus and utilize the parallelism of the GPU. To summarize, each thread block is associated with an area of the image for which it computes the local centroid. The local centroids are calculated in parallel and stored in global memory. This data is transferred to host RAM where the CPU sums the local centroids to return the final value. Full source code for the foveation point selection kernel, along with a more detailed explanation of the code is provided in Appendix C.

Algorithm 3 Foveation Point Selection

```
Launch GPU portion of algorithm
Allocate Shared Memory Xcoords[ThreadsPerBlock] {Store centroid components in shared memory}
Allocate Shared Memory Ycoords[TheadsPerBlock]
for Pixels in parallel do
  Convert current thread index to $(u, \theta)$
  Calculate $(x, y)$ from $(u, \theta)$
  Calculate magnitude of optical flow vector accessed by thread
  if magnitude > threshold then
    $magnitude = 1$
  else
    $magnitude = 0$
  end if
  $Xcoords[threadIndex] = x * magnitude$
  $Ycoords[threadIndex] = y * magnitude$
  Synchronize threads
  if threadIndex = 0 then {Only one thread sums centroid components}
    for $i = 0$ to $ThreadsPerBlock$ do
      $centroidx = centroidx + Xcoords[i]$
      $centroidy = centroidy + Ycoords[i]$
      if magnitude > 0 then
        $npts + +$
      end if
    end for
  end if
end for
Return centroidx, centroidy, and npts to CPU
for $i = 0$ to $i = NumBlocks$ do
  $centroidx_{cpu} + = centroidx[i]$
  $centroidy_{cpu} + = centroidy[i]$
  $npts_{cpu} + = npts[i]$
end for
Return centroid
```

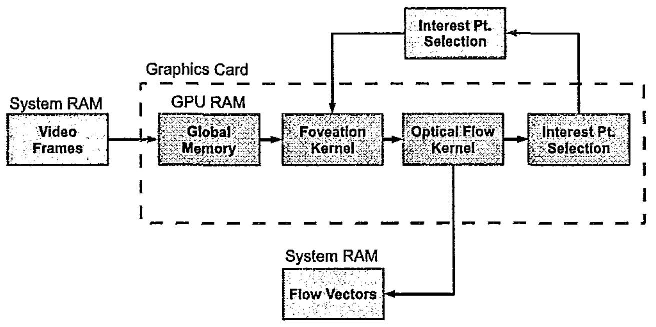


Figure 3.6: Block diagram of video processing algorithm.

3.4 Overall System Implementation

A block diagram of the logical organization of the system is shown in Figure 3.6. A video sequence is loaded into system RAM on the host PC. The video frame is then transferred via direct memory access (DMA) to the GPU global memory via the Peripheral Component Interconnect (PCI) Express bus. The log-polar transform kernel is then performed to foveate the frame, as well as the previous video frame. The LPT has to be recomputed for the previous frame since the foveation point changes between frames. Two consecutive LPT images with the same foveation point are needed for the subsequent optical flow calculation. At this point, the LPT images are transferred into CUDA arrays and bound to texture memory, to increase the processing speed of the optical flow kernel.

The optical flow kernel is then executed, followed by foveation point selection, as described in section 3.3.4. The centroid of the thresholded image is set as the foveation point for the next video frame. The optical flow motion vectors are transferred via the PCI Express bus from global memory to system RAM to complete the process. This process is illustrated in Algorithm 4. The host-side code is also provided in Appendix C.

Algorithm 4 CPU Host Algorithm

```
Load frame $I(t-dt)$ to global memory
for All frames in video sequence do
    Copy frame $I(t)$ to global memory
    Launch LPT transform kernel for frame $I(t-dt)$
    Launch LPT transform kernel for frame $I(t)$
    Swap pointers between frame $I(t)$ and $I(t-dt)$
    Transfer LPT images to texture memory
    Launch optical flow kernel
    Launch foveation point selection kernel
    Transfer optical flow vectors to system RAM
end for
```

3.5 Kernel Efficiency and Resource Utilization

The execution speed of a CUDA kernel is dictated to a large extent by its GPU occupancy, memory access patterns, and divergence. Each multiprocessor is limited to 16 KB of shared memory, hence as the amount of shared memory consumed by a thread block increases, the maximum occupancy (or number of thread blocks executing simultaneously on the GPU) decreases. There is also a maximum number of registers that a thread can use, before variables begin to be stored in *Local Memory*. Local Memory corresponds to registers being stored in the GPU global memory, which negatively impacts execution speed due to the 300 clock cycle access latency between global memory and the GPU. The number of registers available depends on the GPU architecture. The GTX 280 GPU used in this work has a maximum number of 21 registers per thread. All kernels have been structured to avoid local memory use.

Divergent branches and serialized warps (warps that must be executed in sequence, rather than in parallel) also negatively impact performance. Divergent branches and serialized warps are a result of condition statements inside the kernel. Unfortunately, few realistic algorithms can be decomposed to avoid all condition statements, although

Table 3.1: Kernel Resource utilization (Shared mem. values given are per block, others per thread)

Kernel	Registers	Shared Mem.	Local Mem.
Foveation Pt.	7	2112 bytes	0
LPT	8	60 bytes	0
LPT /w interp.	9	56 bytes	0
Optical Flow	20	6780 bytes	0

Table 3.2: Kernel Efficiency (values given are per kernel)

Kernel	Occupancy	Divergent Branches	Serialized Warps
Foveation Pt.	100%	17	792
LPT	100%	0	889
LPT /w interp.	100%	0	877
Optical Flow	50%	721	14130

care must be taken to minimize the number. If a condition affects an entire warp, serialization does not occur.

The kernel resource utilization and efficiency are shown in Tables 3.1 and 3.2, respectively. The optical flow kernel is clearly the most resource-intensive of the four kernels. The optical flow kernel consumes 20 registers per thread and 6780 bytes of shared memory space per block. The shared memory footprint of the kernel reduces its GPU occupancy to 50%. In other words, only half of the maximum number of blocks are executing in parallel as is theoretically possible. All kernels have a low number of divergent branches and serialized warps, especially given the number of threads per kernel.

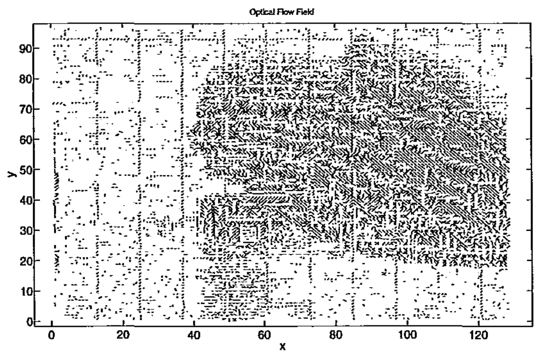
Chapter 4

System Evaluation

Experimental results for the CUDA optical flow system are described in this chapter. A description of the system configuration and performance metrics used to evaluate the system is given first. The performance and accuracy of the algorithms described in Chapter 3 is evaluated using synthetic and real-world video sequences. The final part of the chapter demonstrates the use of the optical flow algorithm in a closed-loop control application.

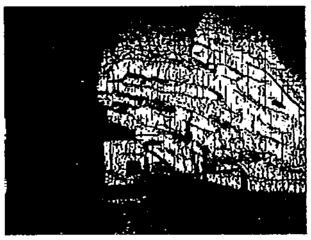
4.1 Experimental Configuration

Experiments were performed on an NVIDIA GeForce GTX 280 graphics card, which includes 240 processors and 1 GB of RAM. The host PC ran on a Pentium Core 2 Duo clocked at 2.6 GHZ, with 4 GB of RAM. Experiments were performed with the foveated optical flow algorithm on four video sequences: a 10 frame sequence of a moving hand (shown in Figure 4.1 and refered to as the hand sequence), two Yosemite sequences, with and without clouds (originally created by Lynn Quam),



(a) Optical Flow Field

(b) Captured video frame



(c) Segmented video frame

Figure 4.1: a) Optical Flow field from live video sequence shown in b). b) One frame of live video sequence. c) Segmented video frame, based on optical flow field magnitude.

and the Ettlinger-Tor sequence [46]. The 10 frame sequence (shown in Figure 4.1) at sizes from 160 $\times$ 120 to 1920 $\times$ 1440 was used to characterize the execution time on the GPU and CPU. The Yosemite sequences, with and without clouds, were used to characterize the error between the results and a ground truth. The standard non-synthetic video sequences, Ettlinger-Tor, was used to test the algorithms performance on real world data.

4.1.1 Measuring Performance Gain

The parallel performance of the algorithm on the GPU was compared to a CPU version using a relative performance gain measure, defined as:

$$P_{rel} = (\text{Execution time on CPU})/(\text{Execution time on GPU}) \tag{4.1}$$

The relative performance measure was proposed in [26]. To characterize the added performance gain resulting from adding foveation to optical flow, a relative performance gain between the foveated and non-foveated GPU algorithms was used. This performance measure is defined by equation 4.2

$$P_{FOV} = (\text{Non-foveated Execution Time})/(\text{Foveated Execution Time}) \tag{4.2}$$

4.1.2 Measuring Optical Flow Accuracy

An angular error measure, as proposed by Barron et al. [8], was used to characterize the accuracy of the optical flow algorithm. The velocity is described by the displacement per unit time: $\vec{v} = (u, v)$ pixels/frame, where u is the horizontal displacement and v the vertical displacement. The error in the velocity vectors can be measured as an angular deviation from the correct orientation. A normalized 3-D direction vector is described by:

$$\vec{v} = \frac{1}{(u^2 + v^2 + 1)^{0.5}}(u, v, 1)^T \tag{4.3}$$

The angular error between the correct velocity, $\vec{v_c}$, and the experimentally calculated velocity, $\vec{v_e}$, is given by:

Table 4.1: Execution time and relative performance gain for hand test sequence with $n = 2$ and $\nu = 2$ for non-foveated optical flow.

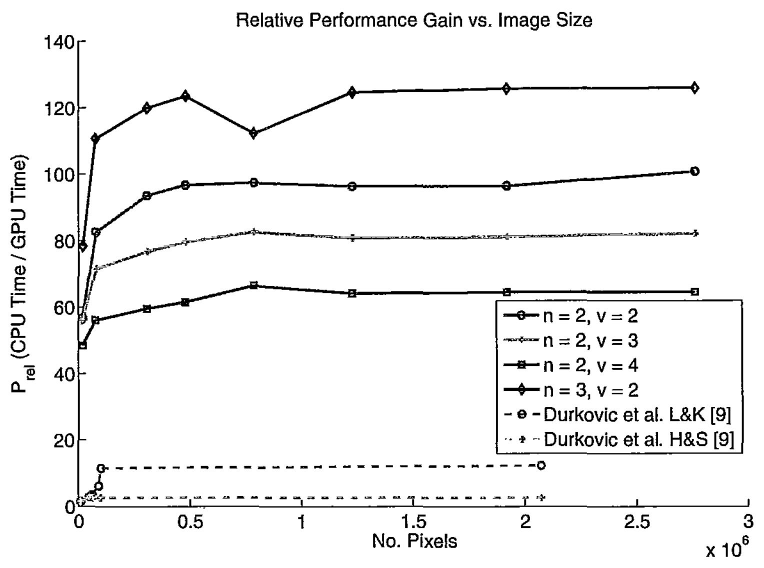
Image Size	GPU (fps)	CPU(fps)	P_{rel}
160×120	659.37	11.62	56.74
320×240	222.12	2.69	82.70
640×480	60.70	0.65	93.64
800×600	39.40	0.41	96.79
1024×768	24.22	0.25	97.50
1280×960	15.64	0.16	96.45
1600×1200	10.00	0.10	96.58
1920×1440	6.94	0.07	100.87

$$\psi_E = \arccos(\vec{v_c} \cdot \vec{v_e}) \tag{4.4}$$

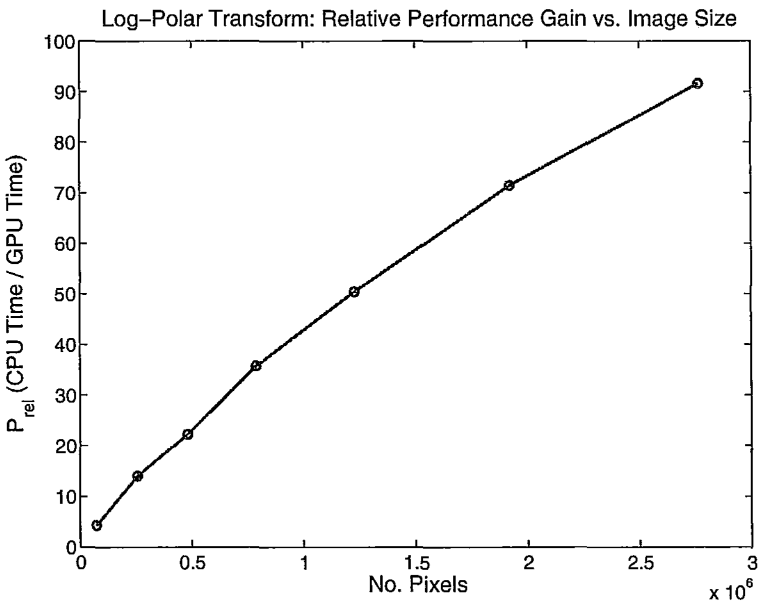
4.2 Performance Evaluation

4.2.1 Non-Foveated Optical Flow

One frame of the speed characterization hand test sequence, along with the resulting optical flow field is shown in Figure 4.1. The performance of the CUDA optical flow algorithm without foveation was tested first, with the results shown in Table 4.1 and illustrated in Figure 4.2(a). Frame rates achieved range from 6.94 fps for a resolution of 1920×1440 to 60.70 fps for 640×480 and 222.12 fps for 320×240. The relative performance gain over the CPU, P_{rel}, ranges from 100.87 to 56.74. The optical flow algorithm was tested for a variety of search window, n, and correlation window, ν, radii.

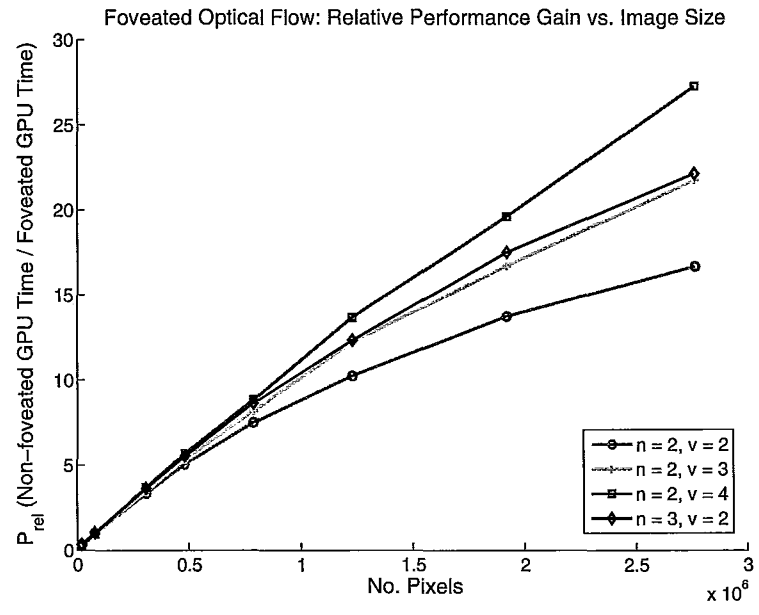


(a) Performance Gain of Non-foveated Optical Flow

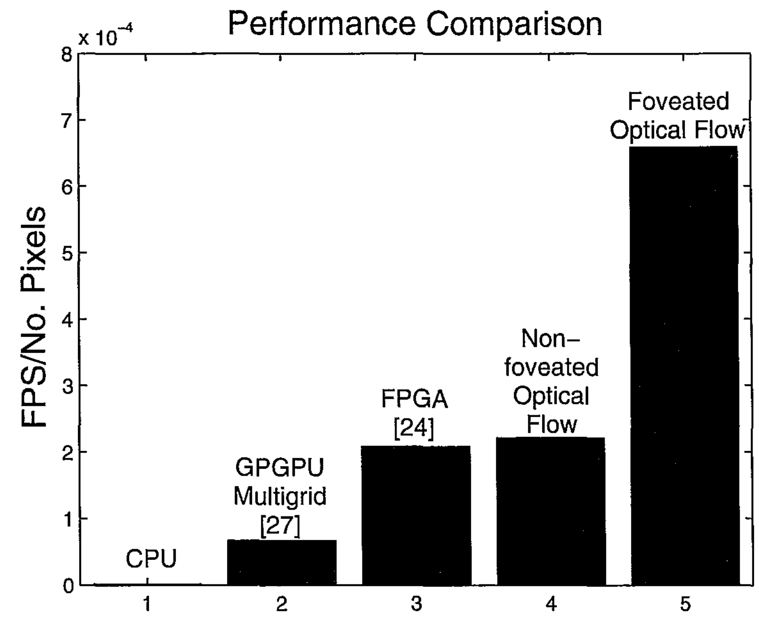


(b) Performance gain of foveation algorithm

Figure 4.2: a) Performance vs. image size for a variety of comparison pixel patch, ν, and search area, n, radii with non-foveated optical flow. b) Performance gain of foveation algorithm



(a) Performance gain of foveated optical flow



(b) Performance Comparison

Figure 4.3: a) Foveated GPU vs. non-foveated GPU performance gain for range of ν and n. b) Performance comparison with literature, normalized by image resolution.

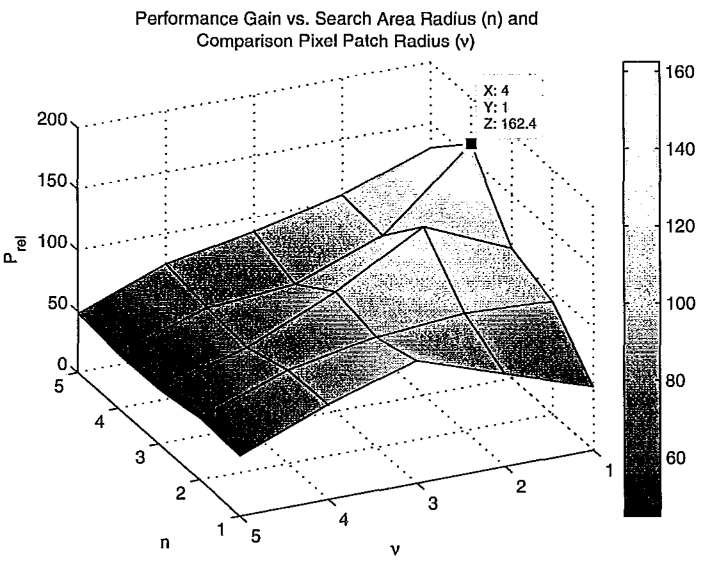


Figure 4.4: Relative performance gain versus ν and n for one resolution (640 × 480)

4.2.2 Foveation Algorithm

The foveation algorithm was applied to the same hand test sequence shown in Figure 4.1, with the relative performance gain illustrated in Figure 4.2(b) and Table 4.2. As illustrated, the performance gain over the CPU is significant, although the algorithm consumes a small amount of GPU time compared to optical flow computation. On the GPU, the algorithm executes at a frame rate of 1037.56 fps for a resolution of 360 × 200 to 628.14 fps for a resolution of 1920 × 1440 when rescaling the cartesian image to a 360 × 200 ($N\theta \times NRings$) LPT image. The performance gain over the CPU shows a nearly linear increase for the range of image sizes tested.

4.2.3 Foveated Optical Flow

A foveated version of the optical flow algorithm, with deformation correction, was applied to the same test sequence. In all cases, the log-polar transform resampled the

Table 4.2: Foveation algorithm execution time and relative performance gain for hand test sequence.

Image Size	GPU (fps)	CPU(fps)	P_{rel}
360×200	1037.56	242.69	4.28
640×400	1027.41	73.14	14.05
800×600	894.77	40.26	22.23
1024×768	877.19	24.53	35.76
1280×960	781.25	15.49	50.44
1600×1200	712.25	9.97	71.47
1920×1440	628.14	6.85	91.64

Table 4.3: Foveated optical flow algorithm execution time and relative performance gain for hand test sequence with $n = 2$ and $\nu = 2$.

Image Size	GPU Foveated(fps)	GPU Non -Foveated(fps)	P_{FOV}
160×120	230.14	659.37	0.35
320×240	224.81	222.12	1.01
640×400	202.41	60.70	3.33
800×600	198.82	39.40	5.05
1024×768	181.87	24.22	7.51
1280×960	160.30	15.64	10.25
1600×1200	137.47	10.00	13.75
1920×1440	115.76	6.94	16.68

input image to 360×200 ($N\theta \times NRings$). The algorithm was tested for a variety of search window and correlation window radii, (n, ν), with the results shown in Figure 4.3(a) and Table 4.3. In this case, the relative performance gain, P_{FOV}, was computed as the non-foveated GPU speed / Foveated GPU speed.

4.2.4 Performance Characteristics

The non-foveated optical flow algorithm exhibits a characteristic exponential rise in performance gain over the CPU at low resolution follow by saturation at higher resolutions, as shown in Figure 4.2(a). Low performance gain at low resolutions occurs because of two factors: overhead in launching the kernel on the GPU and memory transfers between the host and GPU RAM. Launching a kernel on the GPU consumes resources.

As the time spent in the kernel compared to the time spent launching kernels decreases, the performance gain of the algorithm over the CPU also decreases. The same condition holds for memory transfers. As the frequency of transferring image frames increases, extra time is spent setting up memory transfers and binding the foveated images to textures, as well as transferring results back to system RAM. As image resolution increases, the overhead as a fraction of total run time decreases. This causes the rapid rise in performance gain.

When the GPU is functioning at its full capacity, the performance gain levels off, resulting in a saturated region. The saturation region therefore corresponds to a 1 to 1 linear increase in GPU execution time versus CPU execution time with increasing image size.

The values of the comparison pixel patch radius, ν, and search area radius, n, also affect the performance gain. This can clearly be seen in the surface plot presented in Figure 4.4. Certain values of the comparison pixel patch, ν, and search area, n, radii lead to higher performance gain. In particular, there is a peak at low values of ν and medium values of n.

The result is due to different levels of GPU occupancy. The percentage of idle threads increases as ν increases, decreasing the percentage of threads involved in calculating motion vectors. The number of pixels that each thread must access when calculating the SSD also increases with ν, increasing computation time. As n increases, the amount of shared memory used per thread block also increases. Since only 16KB of shared memory is available per multiprocessor, increasing n decreases the number of thread blocks that can execute in parallel. However, as seen by the P_{rel} increase in Figure 4.2(a) with an increase in n, the parallel GPU implementation is more efficient than the CPU with the extra computation.

Introducing foveation to the optical flow algorithm introduces a further performance gain of up to 27 times, depending on resolution and n and ν parameters. A graph of foveated GPU versus non-foveated GPU performance is shown in Figure 4.3(a), with a table of frames per second versus resolution shown in Table 4.3. The foveation algorithm re-samples a higher resolution image (the original cartesian video frames) into a lower resolution image. The optical flow algorithm therefore has to operate on a smaller image. The added complexity of calculating the LPT is more than outweighed by the drastic reduction in pixels, especially with increasing image size. The performance gain is greatest for larger ν and n values. Foveation is therefore best suited to accelerating optical flow at the heaviest computational burdens. However, even a $2\times$ decrease in computation time is significant, given the already accelerated nature of non-foveated optical flow on the GPU.

4.2.5 Comparison with Other Work

The results discussed show an improvement over previous work utilizing FPGA and GPU methods. For non-foveated optical flow, Grossauer et al. [27] report a speed of 17.419 fps using multi-grid methods on the GPGPU at a resolution of 511×511. Wei et al. [24] reported an FPGA implementation of tensor-based optical flow at 64 fps

Table 4.4: Algorithm performance for Yosemite sequence. Experiment corresponds to LPT resolution, $(N\theta \times NRings)$.

Experiment	AAE (deg)	STD (deg)	FPS
Non-Fov	14.81	17.77	137.68
360×200	27.71	37.76	141.04
360×150	32.05	43.52	166.70
360×100	39.94	53.02	269.80

at 640×480 resolution. Mizukami et al. [28] report a CUDA implementation of Horn & Schunck (H&S), but the results are difficult to compare in this case since their implementation did not consider memory transfer overhead. However, they report a P_{rel} of 2.3 at 316×252 resolution.

The work of Durkovic et al. [26] is included in Figure 4.2(a) for comparison. Durkovic et al. implemented the Lucas and Kanada (L&K) and Horn and Schuncke (H&K) algorithms on the GPU using shader technology, prior to the introduction of CUDA. The advantages of an efficient mapping of optical flow to CUDA, compared to shaders, is evident in the graph. Both implementations exhibit similar behaviour, with an initial exponential rise in performance gain followed by saturation.

A comparison of the previously reported work and the current algorithm is shown in Figure 4.3(b). The graph displays the frame rate normalized by the number of pixels in the image. CPU, GPGPU multi-grid [27], FPGA [24], non-foveated optical flow and foveated optical flow results are shown. Both foveated and non-foveated optical flow was calculated with $n = \nu = 2$ and at 320×200 and 640×480 resolution, respectively. Significantly, the non-foveated optical flow performance matches that of a dedicated FPGA implementation reported in [24].

Table 4.5: Algorithm performance for Yosemite sequence with interpolation. Experiment corresponds to LPT resolution, ($N\theta \times NRings$).

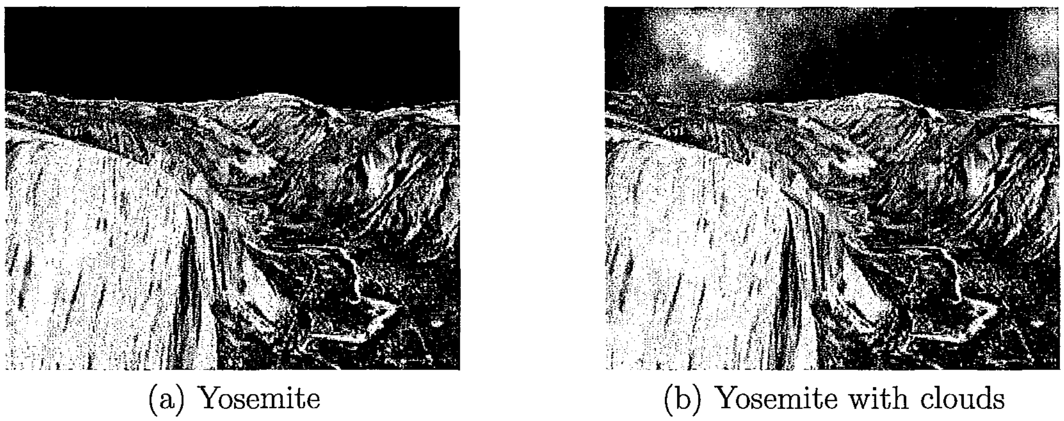
Experiment	AAE (deg)	STD (deg)	FPS
Non-Fov	14.81	17.77	137.68
360 × 200	24.09	32.51	137.02
360 × 150	30.00	41.06	160.35
360 × 100	37.69	50.74	246.72

Table 4.6: Algorithm performance for Yosemite with Clouds sequence. Experiment corresponds to LPT resolution, ($N\theta \times NRings$).

Experiment	AAE (deg)	STD (deg)
Non-Fov	24.12	21.39
360 × 200	34.47	43.88
360 × 150	39.41	49.54
360 × 100	48.09	58.86

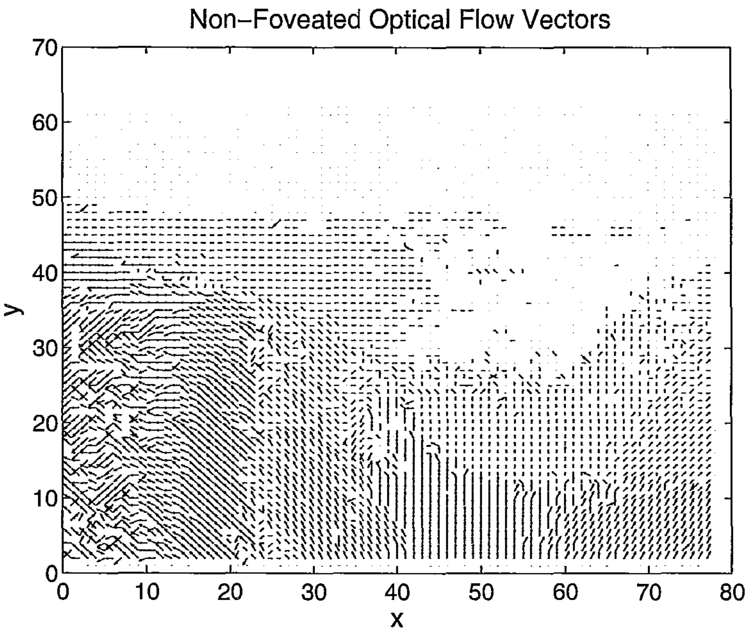
Table 4.7: Algorithm performance for Yosemite with Clouds sequence with interpolation. Experiment corresponds to LPT resolution, ($N\theta \times NRings$).

Experiment	AAE (deg)	STD (deg)
Non-Fov	24.12	21.39
360 × 200	31.06	39.59
360 × 150	37.57	47.56
360 × 100	46.26	57.26

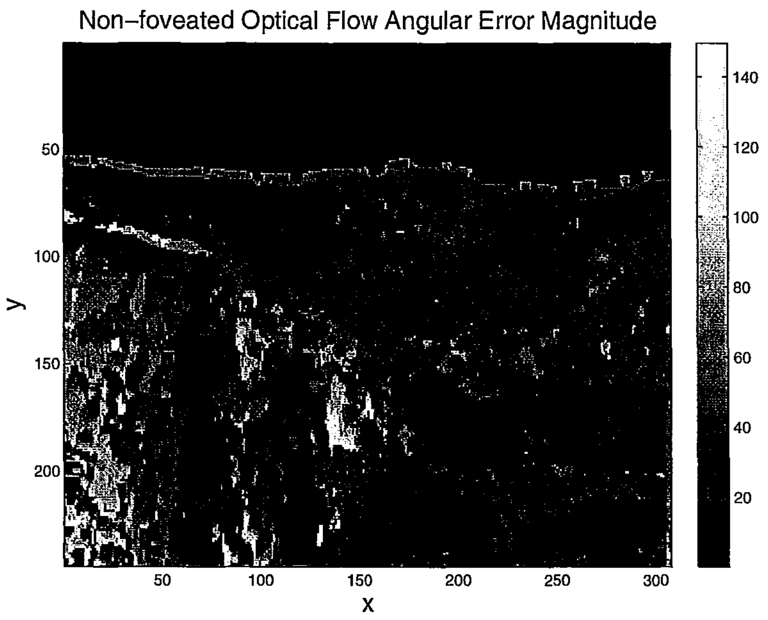


(a) Yosemite (b) Yosemite with clouds

Figure 4.5: a) Frame 10 of Yosemite sequence b) Frame 10 of Yosemite with clouds sequence

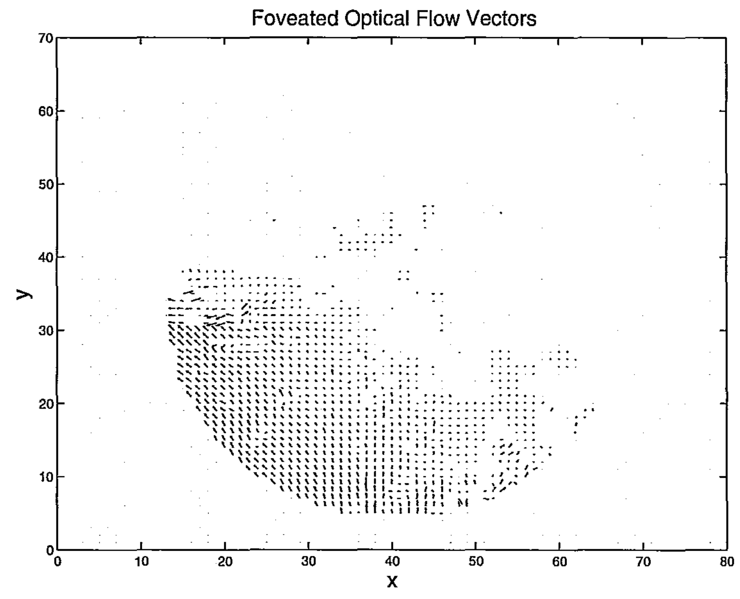


(a) Non-foveated Optical Flow Vectors

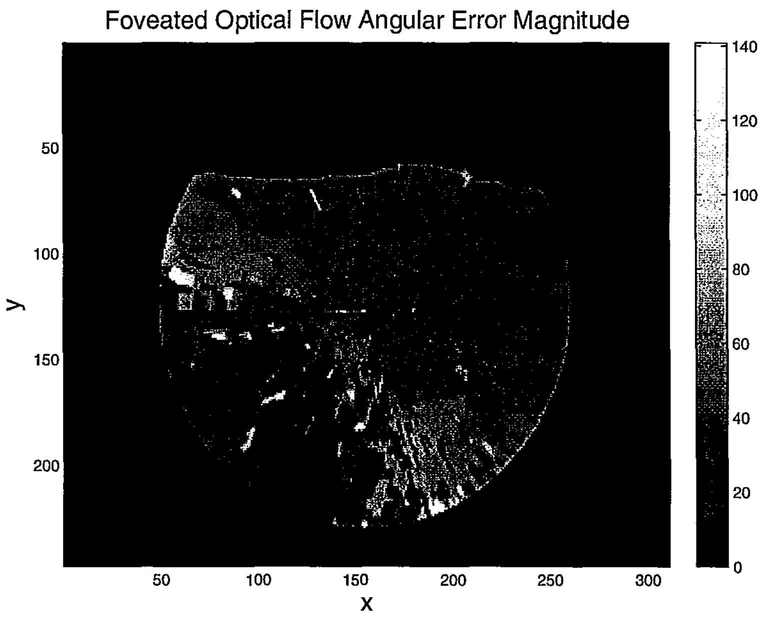


(b) Non-foveated Optical Flow Angular Error Magnitude

Figure 4.6: Optical flow vectors (a) and optical flow angular error magnitude (b) for non-foveated optical flow. Experiments performed with the Yosemite sequence.

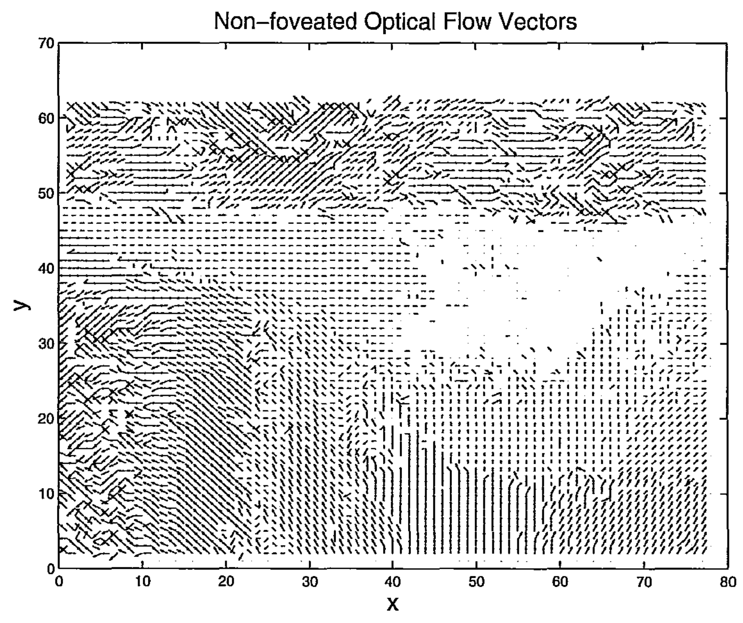


(a) Foveated Optical Flow Vectors

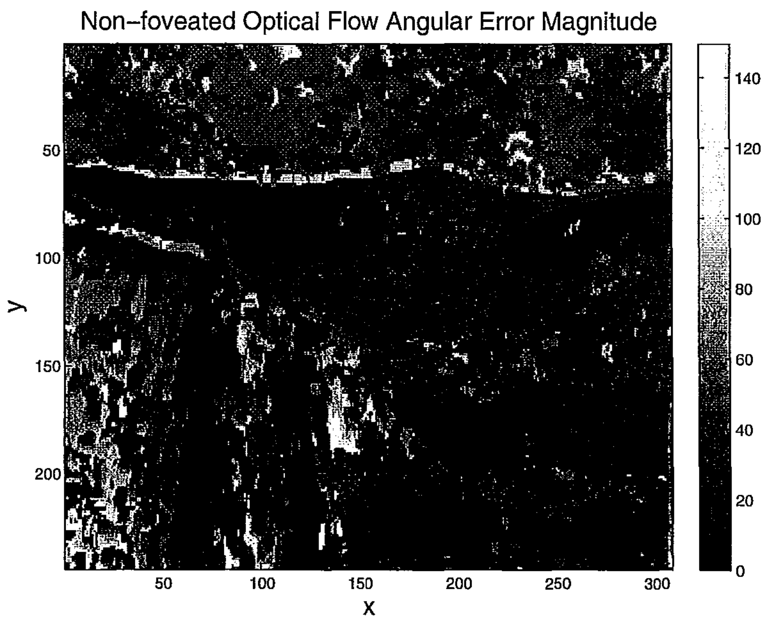


(b) Foveated Optical Flow Angular Error Magnitude

Figure 4.7: Optical flow vectors (a) and optical flow angular error magnitude (b) for foveated optical flow. Experiments performed with the Yosemite sequence.

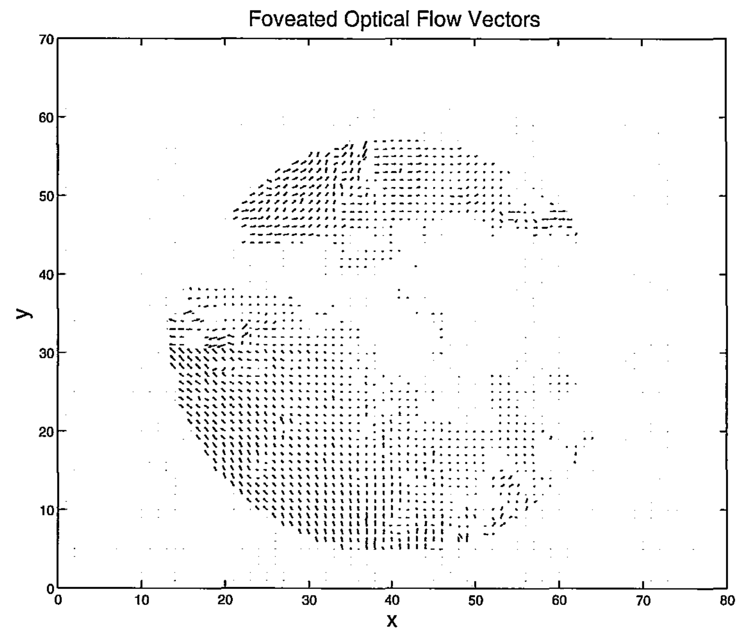


(a) Non-foveated Optical Flow Vectors

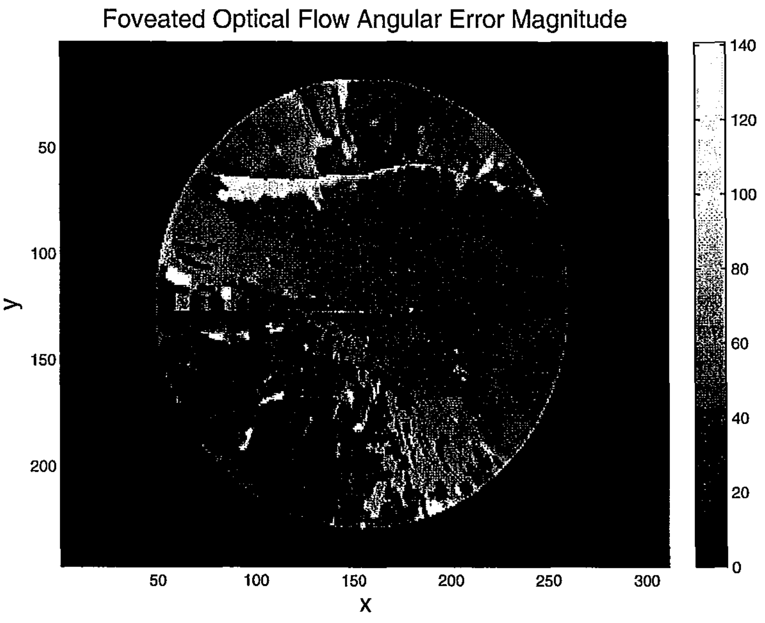


(b) Non-foveated Optical Flow Angular Error Magnitude

Figure 4.8: Optical flow vectors (a) and optical flow angular error magnitude (b) for non-foveated optical flow. Experiments performed with the Yosemite with clouds sequence.



(a) Foveated Optical Flow Vectors



(b) Foveated Optical Flow Angular Error Magnitude

Figure 4.9: Optical flow vectors (a) and optical flow angular error magnitude (b) for foveated optical flow. Experiments performed with the Yosemite with clouds sequence.

4.3 Error Analysis on Synthetic Data

To characterize the accuracy of the optical flow algorithm developed in this work, video sequences are needed with a known motion vector field, or *ground truth*. Synthetic video sequences are useful for this task. The Yosemite sequence, originally created by Lynn Quam of SRI, is a standard test sequence in the computer vision community and is used for ground truth comparison in this work. The Yosemite sequence consists of a fly-through of Yosemite national park texture-mapped onto a 3-D landscape generated from a height-map of the fly-through. Two versions of the sequence are available, one without clouds and one with clouds. Representative frames of the Yosemite sequences are shown in Figure 4.5.

Four test cases are used to quantify the error performance of the algorithm: The Yosemite sequence without clouds, with and without interpolation, and the Yosemite sequence with clouds, with and without interpolation. For each sequence, the AAE and STD were calculated and averaged over the entire 10 frames of the video. For each test case, the algorithm was run without foveation, and with foveation at resolutions of ($N\theta \times NRings$) of (360×200), (360×150), and (360×100).

Results for the proposed algorithm with the Yosemite sequence are shown in Tables 4.4 and 4.5 and Figures 4.6 and 4.7, respectively. The non-foveated correlation based optical flow algorithm returns an average angular error (AAE) of 14.81 degrees, which is consistent with other correlation-based methods [8]. Introducing foveation increases the AAE, with a corresponding drop in processing time. At a foveated resolution of (360×200) the AAE increases by 12.9 degrees as shown in Table 4.4. As the foveated resolution decreases the AAE continues to increase. Decreasing the resolution to 360×150 from 360×200 increases the AAE by 4.34 degrees while increasing the frame rate by 25.66 fps. A further decrease in resolution to 360×100 increases the AAE by 7.89 degrees and increases the frame rate by 103.10 fps.

Introducing linear interpolation to the log-polar transform decreases the overall error in the results as shown in Table 4.5. At a resolution of 360×200, adding interpolation decreases the error by 3.62 degrees and the frame rate by 4.02 fps. At a resolution of 360×150 the AAE and frame rate decrease by 2.05 degrees and 6.35 fps, respectively. Finally, at a resolution of 360×100, the AAE and frame rate decrease by 2.25 degrees and 23.08 fps.

Similar results were obtained for the Yosemite with clouds sequence, as shown in Tables 4.6 and 4.7 and Figures 4.8 and 4.9, respectively. The error for this sequence is higher since the optical flow algorithm is not efficient at calculating the motion vectors of the cloud region. This occurs because the cloud region violates the brightness constancy assumption discussed in Chapter 2. The clouds, in fact, undergo random brownian motion.

A plot of resolution versus AAE for the above results is shown in Figure 4.10. Clearly, the initial resampling to a space-variant LPT image introduces error, which continues to increase with decreasing foveal resolution. Interpolating when sampling reduces the error at all resolutions for a modest decrease in frame rate.

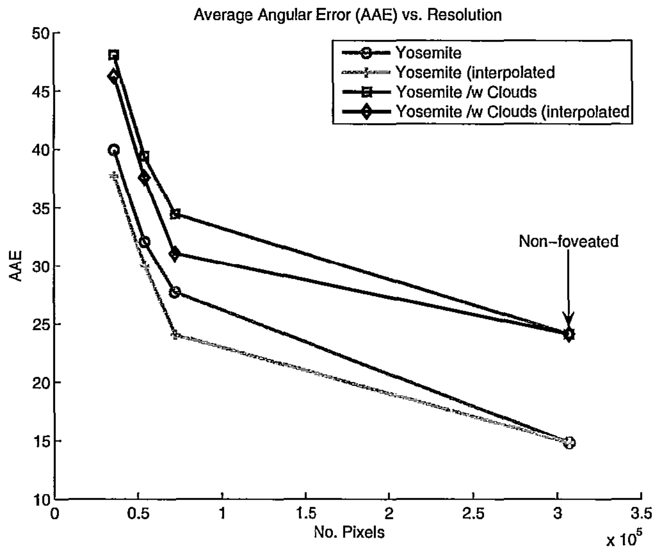
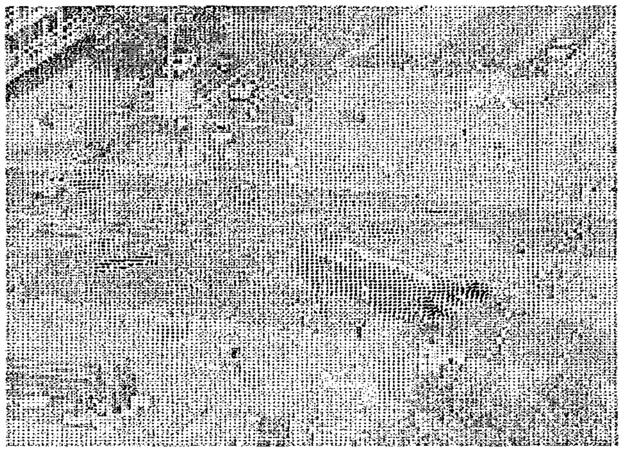
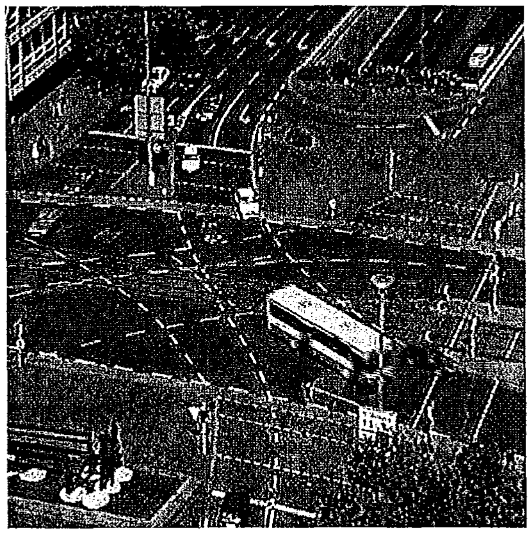


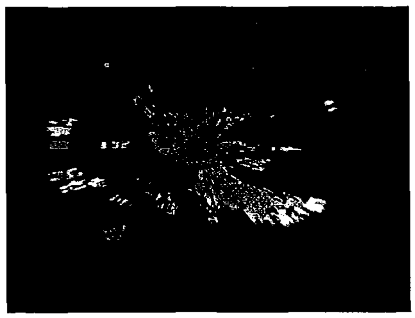
Figure 4.10: Average angular error versus resolution



(a) Optical Flow Field

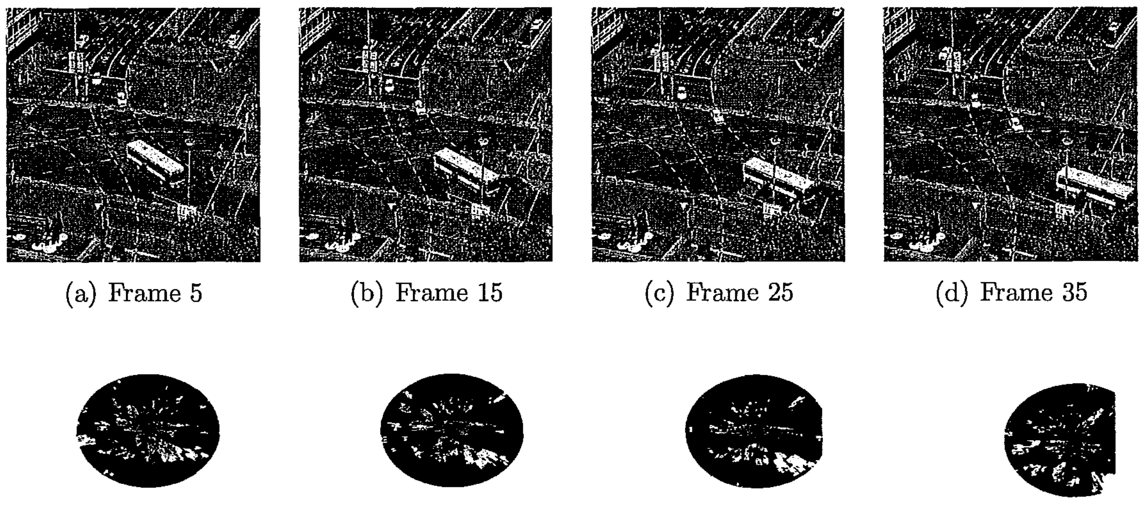


(b) Ettlinger-Tor Frame 10



(c) Optical Flow Magnitude

Figure 4.11: a) Computed optical flow field between frames 9 and 10 of the Ettlinger Tor traffic sequence b) Frame 10 of Ettlinger-Tor traffic sequence c) Magnitude of the optical flow field.



(a) Frame 5 (b) Frame 15 (c) Frame 25 (d) Frame 35

(e) Frame 5 (f) Frame 15 (g) Frame 25 (h) Frame 35

Figure 4.12: Ettlinger-Tor frames 5, 15, 25, and 35 (a)-(d) with corresponding optical flow field magnitude (e)-(h)

4.4 Performance on Real-World Data

The algorithm was tested on real-world data to test its performance in more realistic situations. The Ettlinger-Tor [46] sequences was used for this purpose. The foveation point was allowed to vary to focus the fovea on the moving regions. Results are shown in Figure 4.11. As expected, the results are clearest near the foveation point. The flow boundaries are sharp, and can be used directly for segmentation purposes by applying a thresholding step. A segmented motion magnitude map can be further used to change the foveation point to acquire an accurate flow map at varying points in the image.

A four-frame sequence from Ettlinger-Tor is shown in Figure 4.12(a)-(d). The corresponding optical flow field magnitude, with foveation is shown in Figure 4.12(e)-(h). The fovea is able to stay fixed on the largest moving object, in this case, the bus. The moving objects are also appropriately segmented.

4.5 Application of Foveated Optical Flow to Closed-Loop Control

In the previous sections, the accelerated optical flow algorithm was tested with synthetic and real-world pre-recorded video sequences. However, in many applications, computer vision algorithms must operate in real-time with live video. One example is controlling a robot. A robotic control experiment was devised to verify that the present system is capable of operating under such conditions. The goal of the experiment is to process a video stream in real-time, extract the position of a robot arm, and use that information to position the arm. A robotic control experiment can therefore be thought of as an instructive example of a real-time application of the parallel optical flow system developed in this previous chapters.

4.5.1 Visual Servo Control

The visual servoing technique will be used to implement closed loop position control of a robot arm. The reader is directed to Appendix A for a review of control theory, visual servo, and robot mechanics. A short summary of the method is provided in this section.

The general idea behind visual servo is to minimize the position error between a set of features on an object to be controlled and a set of desired positions for those features. The feature position is measured and tracked directly in images streamed from a video camera. The next step is to relate the movement of the image features to the movement of the object in three-dimensional space. This is done using a mathematical construct known as the *image jacobian*.

In this experiment, the image jacobian related image feature motion to the motion of the end-effector of a robot arm. For control purposes, it is more convenient to express the motion of the image features in terms of the angular motion of the robot's joints. This is done using the robot's *kinematic jacobian*. This forms the joint-space visual servo control law, given by

$$\dot{\vec{\theta}} = -\lambda[J_s V J(\vec{\theta})]^{+}\vec{e} + \mu[J_s V J(\vec{\theta})]^{+}\int \vec{e}dt + \kappa[J_s V J(\vec{\theta})]^{+}\frac{\partial \vec{e}}{\partial t} \tag{4.5}$$

In the above equation, J_s is the image jacobian, $J(\vec{\theta})$ is the kinematic jacobian, V is a transformation matrix from the robot coordinate frame to the camera coordinate frame, $\vec{e}$ is the position error vector, and $\dot{\vec{\theta}}$ is the angular velocity of the robot's joints.

The control law include three terms, a proportional, integral, and derivative term to form a PID controller. λ is the proportional gain, μ is the integral gain, and κ is the derivative gain. A derivation of the above equation is provided in Appendix A. The optical flow algorithm is used to measure the position of the image features while the visual servo system actuates the robot arm.

4.5.2 Experimental Setup

The goal of the experiment is to use the optical flow algorithm to measure the position of a robot's end-effector in real-time. The CUDA system is used as the feedback in a closed-loop visual servo controller in order to position the arm. While the use of

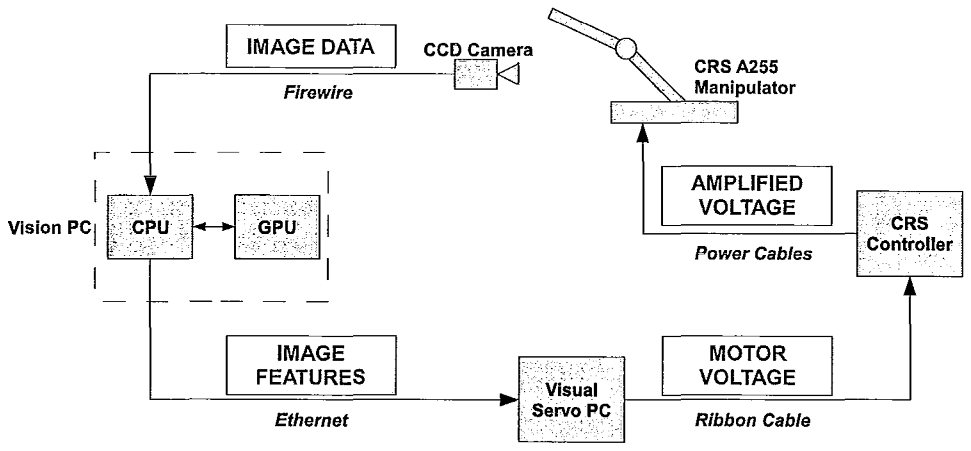


Figure 4.13: System diagram of the visual servo control experiment

visual servo in such an application has already been explored, the experiment serves as a demonstration that the optical flow system can function at a frame rate sufficient for real-time control on a standard PC.

A schematic of the experimental setup is shown in Figure 4.13, as well as a photograph of the actual system in 4.14. The video is acquired by a CCD digital camera and transferred to the vision PC over FireWire. The CUDA algorithm is then executed on the vision PC. The position of the end-effector is extracted and transferred to the control PC via ethernet. The control PC executes the visual servo controller and outputs voltage signals to the robot's power amplifiers, completing the control loop.

A more detailed description of each component is discussed in Appendix B. Briefly, however, the robot arm is a CRS Robotics A255 5 degree of freedom manipulator. The control software is designed in Simulink, and compiled to real-time code using software provided by Quanser Consulting.

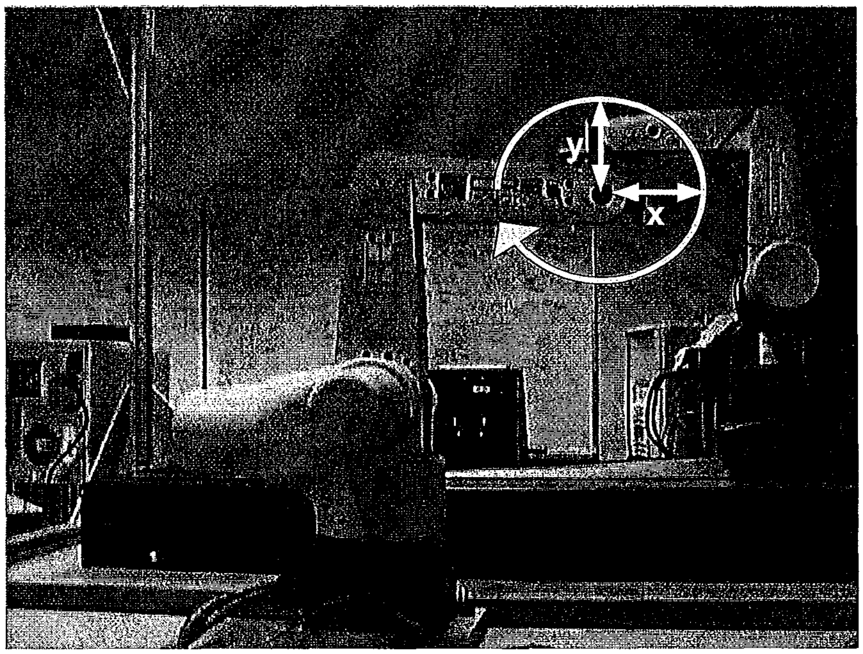


Figure 4.14: Experimental configuration of the CRS A255 robot arm. The robot follows a circular trajectory in a vertical plane

To simplify the experiment, the robot is only controlled in two joints, confining the end-effector to a vertical plane. To aid in feature extraction, a green marker is placed on the end-effector. This allows the end-effector to be located in a motion segmented image (resulting from optical flow computation) using colour segmentation. Finally, the camera is placed parallel to the arm's plane of motion, which sets $V = I$, the identity matrix.

4.5.3 Experiments

Five sets of experiments were performed to characterize the behaviour of the visual servo controller. In all cases, the arm was set to follow a circular trajectory at various sizes and velocities. The configuration of the robot arm and the path it was set to

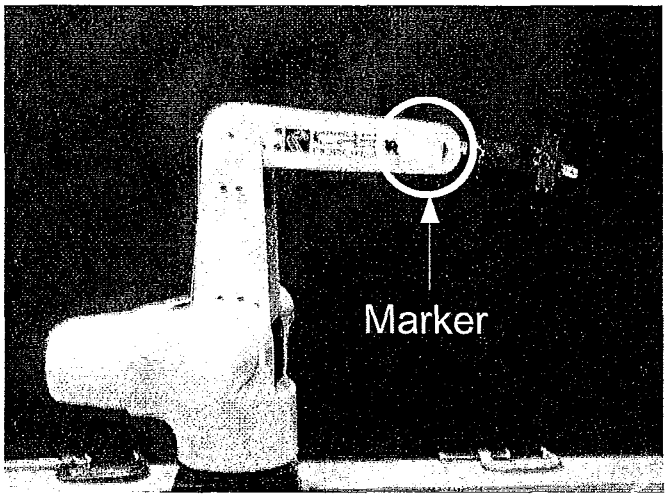


Figure 4.15: Marker used to determine the position of the end effector using image processing

follow is shown in Figure 4.14. The path was specified in the image plane, hence the visual servo controller was responsible for following the trajectory. The optical flow system, meanwhile, was used to measure the position of the image features.

To simplify the experiment, a green square was attached to the end-effector to facilitate tracking its position. This is pictured in Figure 4.15. After the optical flow step, the green square is segmented from the motion segmented image using a simple color threshold. In all cases, the sampling time of the visual servo loop was limited by the maximum speed of the camera: 30 Hz. The camera resolution was set at 640×480.

A summary of the experiments performed is shown in Table 4.8. The amplitude refers to the radius of the circular trajector (in pixels) from the robot's starting position. The starting position is designated as the origin, $(0, 0)$. The first three experiments test the response of the system with changes in the amplitude of the

Table 4.8: Summary of visual servo experiments

Experiment	Amplitude [x y]	Freq. (Hz)
1	[40 40] pixels	1
2	[30 30] pixels	1
3	[20 20] pixels	1
4	[30 30] pixels	2.5
5	[30 30] pixels	5

Table 4.9: Error versus amplitude in visual servo experiments

Experiment	Amplitude [x y]	Error in x [pixels]	Error in y [pixels]
1	[40 40] pixels	7.29	7.02
2	[30 30] pixels	5.85	5.40
3	[20 20] pixels	3.79	3.77

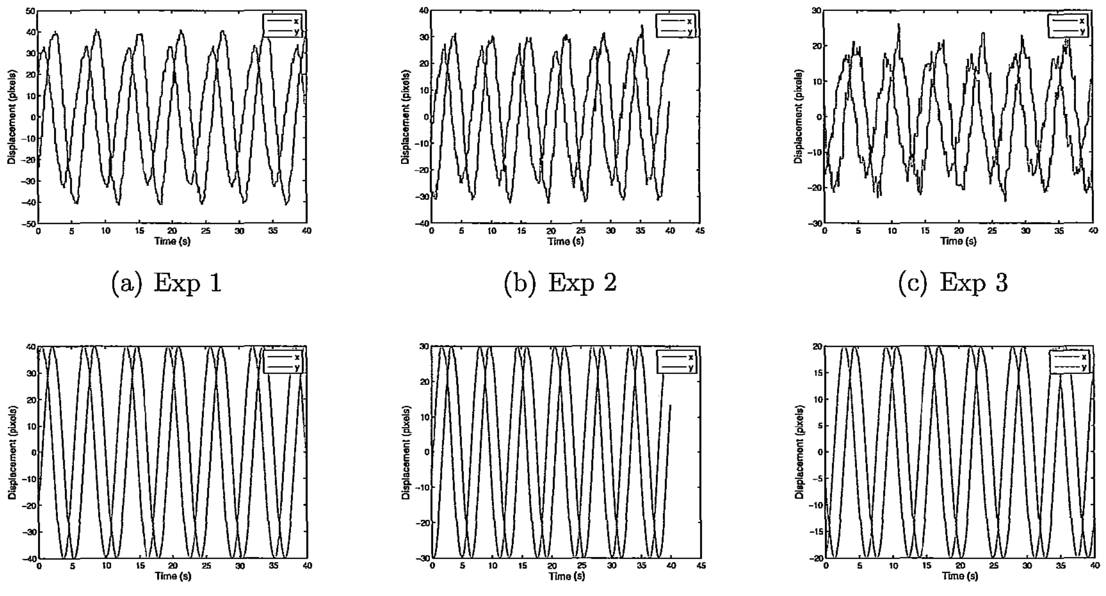
circular trajectory. The experimental results, along with the theoretical trajectory, are shown in Figure 4.16.

Two major effects are visible in the results. The first is the step-like nature of the trajectories. This is caused by the low sampling rate of the camera (30 fps, or 30 Hz). If the sampling rate were increased, the curves would become smoother. The second effect is the noise in the robot's position. The effect becomes especially evident as the size of the trajectory decreases. The noise is caused by the low resolution of the camera (640 $\times$ 480). The resolution limits the precision of the visual position measurement of the end-effector. With increasing resolution, the measurement noise would decrease.

The next set of experiments test the response of the system as the velocity of the desired trajectory is increased. The first set of experiments (experiment 3) tests one revolution per second, the next set of experiments (experiment 3 and 4) increase the rate to 2.5 revolutions per second and 5 revolutions per second. The results are shown

Table 4.10: Error versus frequency in visual servo experiments

Experiment	Frequency	Error in x [pixels]	Error in y [pixels]
3	1.0 Hz	5.85	5.40
4	2.5 Hz	14.29	8.88
5	5.0 Hz	26.67	4.72



(a) Exp 1 (b) Exp 2 (c) Exp 3

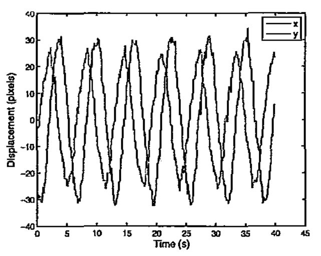
(d) Exp 1 (e) Exp 2 (f) Exp 3

Figure 4.16: Experimental (top) versus theoretical (bottom) trajectories for visual servo experiments 1 - 3

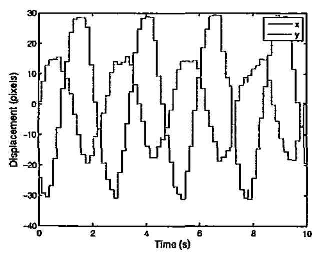
in Figure 4.17.

The discrete nature of the positioning is particularly apparent at higher speeds. As stated previously, this is a result of the low sampling rate of the camera. At 5 Hz, the position has a large deviation from the desired trajectory. This effect would decrease as the sampling rate of the camera increases.

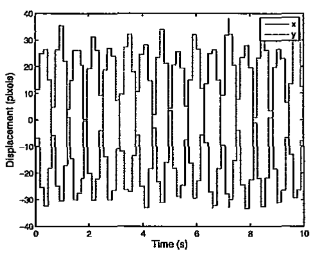
The error between the output (i.e. the position on the image plane of the end-effector) and the input (the desired position) was characterized by calculating the



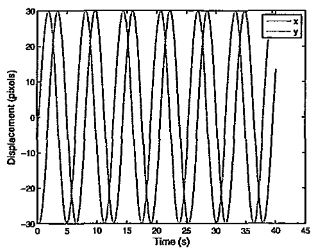
(a) Exp 3



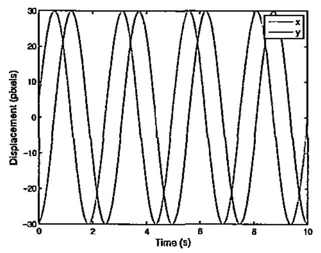
(b) Exp 4



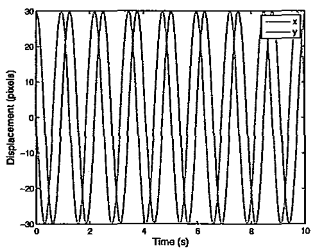
(c) Exp 5



(d) Exp 3



(e) Exp 4



(f) Exp 5

Figure 4.17: Experimental (top) versus theoretical (bottom) trajectories for visual servo experiments 3 - 5

average error between the two. A summary of the results is presented in Table 4.9. The error is largest for the first experiment, with a displacement radius of 40 pixels. The error decreases with decreasing amplitude. As the movement amplitude increases, the frequency of revolutions stays the same (1 revolution per second), hence the speed of the arm must increases. This results in lower accuracy.

This effect is more clearly seen by varying the frequency of revolution while maintaing a constant amplitude, as shown in Table 4.10. As frequency increases, the error increases as well. This is particularly apparent at 5 revolutions per second. It is interesting to note that the error in the y direction does not increase as substantially as in the x direction. This is due to the fact that to move in the y direction the robot must actuate its elbow, while moving in the x direction requires movement in the shoulder. The shoulder is a much heavier joint than the elbow, reducing positioning accuracy. This experiment illustrates that positioning accuracy is a function of both the visual servo system, as well as the mechanical properties of the manipulator.

While the resolution and sampling rate of the camera limited the fidelity of the visual servo controller, the experiments were successful in demonstrating that the foveated optical flow system is capable of functioning in a real-time environment. The application of CUDA to control of robotics was also demonstrated.

Chapter 5

Conclusion

This thesis presents a novel method to compute the optical flow for time-critical tasks by combining GPU hardware acceleration with foveated vision techniques. The performance of the implemented optical flow algorithm was found to achieve frame-rates equaling that of an FPGA implementation [27], even prior to adding the foveation step. These results demonstrate the ability of optimized CUDA algorithms to match the performance of dedicated hardware implementations in certain cases. By adding foveation to the optical flow system, the frame rates achieved were significantly higher than previously reported non-foveated GPU and FPGA implementations of optical flow.

The thesis began with a literature review of the methods used to calculate the optical flow and foveate images, as well as a survey of the computational techniques used to implement the above methods at high frame rates. The variational and correlation-based optical flow techniques were covered in more detail in Chapter 2, along with a justification of the choice of the correlation-based method in this work.

Methods to produce foveated images were discussed, specifically the log-polar transform and the consequences of its use when calculating the optical flow. The CUDA GPU programming model was then discussed in Chapter 3, along with details of the implementation of the foveated optical flow system. Experiments to characterize the execution speed and accuracy of the implemented system were performed in Chapter 4, along with a proof-of-concept study of using the system for real-time closed-loop control.

The performance of the non-foveated optical flow system was characterized on a pre-recorded 10 frame video sequence resampled to resolutions ranging from 160×120 to 1920×1440. With the correlation window, ν, and search window, n, radii set to 2, the optical flow algorithm was found to operate at 659.37 fps a 160×120, 60.70 fps at 640×480, and 6.94 fps at 1920×1440. The relative performance gain, P_{rel} compared to a CPU implementation ranged from 56.74 to 100.87 between resolutions of 160×120 to 1920×1440, respectively. The performance of the algorithm was further characterized as a function of the search window and correlation window radii, (n, ν). It was found that the lowest performance gain occurs with high values of ν, while the highest performance gain occurs for high levels of n. By plotting P_{rel} as a surface plot, it was found that peak performance gain occurs for low values of ν and medium values of n.

The performance of the LPT transform algorithm was characterized next, and found to have a linear performance gain over the CPU over the range of resolutions tested above. The LPT transform was found to be compuationally lightweight compared to the optical flow algorithm. When resampling to a $(\theta, u) = (360, 200)$ LPT image, the foveation algorithm ran at 1037.56 fps for a cartesian image resolution of

360×200 and 628.14 fps for a cartesian image resolution of 1920×1440. The performance gain over the CPU at the given resolution range varied from 4.28 to 91.64, hence executing the LPT transform on the GPU results in a significant computation speed increase.

The foveation and optical flow algorithms were combined and their performance tested on the same video sequence to characterize execution speed. The non-foveated GPU to foveated GPU performance gain, P_{GPU}, was found to be 3.33 at a resolution of 640×480 and 16.68 at a resolution of 1920×1440 with $(n, \nu) = (2, 2)$. Given the already accelerated execution speed of the optical flow algorithm, an additional multiplier of $3.3 - 16.68$ is quite significant and allowed the foveated optical flow system to execute at 202.41 fps at 640×480 resolution and 115.76 fps at 1920×1440 resolution.

With the performance characterization complete, the accuracy of the non-foveated and foveated optical flow methods was tested next. The standard optical flow test sequence, Yosemite fly-through, was used for this task and the computer optical flow vectors were compared to a ground truth.

It was found that the non-foveated correlation based optical flow algorithm returns an AAE of 14.81 degrees, which is consistent with other correlation-based methods [8]. Introducing foveation increases the AAE, with a corresponding drop in processing time. At a foveated resolution of (360×200) the AAE increases by 12.9 degrees. As the foveated resolution decreases the AAE continues to increase. Decreasing the resolution to 360×150 from 360×200 increases the AAE by 4.34 degrees while increasing the frame rate by 25.66 fps. A further decrease in resolution to 360×100 increases the AAE by 7.89 degrees and increases the frame rate by 103.10 fps.

Introducing linear interpolation to the log-polar transform decreases the overall error in the results with a modest increase in computation time. At a resolution of 360×200, adding interpolation decreases the error by 3.62 degrees and the frame rate by 4.02 fps. At a resolution of 360×150 the AAE and frame rate decrease by 2.05 degrees and 6.35 fps, respectively. Finally, at a resolution of 360×100, the AAE and frame rate decrease by 2.25 degrees and 23.08 fps.

Next, the system was tested on a real-world test sequence, the also well-known Ettlinger-Tor video. The foveation point was allowed to vary to focus the fovea on the moving regions. The results were found to be clearest near the foveation point, as expected. The boundaries of the optical flow field were sharp, and hence were eligible to be used directly for segmentation purposes by applying a thresholding step. During the course of the video sequence, the fovea was able to remain fixated on the largest moving object, in this case, the bus in the video. Hence, the implemented system was shown to be able to process a real video sequence with adequate results.

A visual servo experiment with a robotic arm was devised to demonstrate that the foveated optical flow system developed in this thesis is suitable for use in real-time robotics and control applications. The optical flow system was used to segment the moving robot arm from the background and find the position of its end-effector in the image. This data was then used to close a control loop and actuate the robot to follow a circular trajectory. The optical flow system was successfully able to operate at 30 fps in a closed-loop with the robot controller and follow the pre-programmed trajectory. The speed of the image processing in this case was limited by the CCD camera used. The experiment successfuly validated the applicability of the CUDA optical flow system developed to control applications. Besides the optical flow system developed in this work, this experiment also successfuly showed the applicability of CUDA image processing systems to real-time control applications in general.

Several directions are possible for further research based on the work presented in this thesis. The optical flow system could be expanded to consider multiple frames and additional constraints to improve the accuracy of the calculated motion vectors. Further applications of foveated optical flow implemented in CUDA could also be explored, particularly in robotics, surveillance, and video compression applications.

Appendix A

Robot Mechanics and Control

A.1 The General Control Problem

A robot arm, also called a *manipulator*, is a multi-jointed system. Each joint is moved by an *actuator*, typically a DC motor. The position and velocity of each joint can therefore be controlled by some control input (i.e. voltage). The position of each joint is measured by sensors, providing *feedback* to adjust the control inputs so that the robot reaches a desired position (or velocity). These components form the basis of a control system.

In control engineering parlance, the system to be controlled is called the *plant*. The system that provides inputs to the plant is called the *controller*. The input to the controller is the *error* signal, which is formed by subtracting the current plant outputs (i.e. joint position, velocity, etc.) from a desired output (the *set-point*). This arrangement is illustrated in Figure A.1.

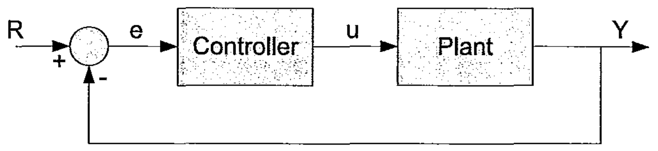


Figure A.1: General control system including set-point (R), error signal (e), plant input (u), and plant output (Y)

A.2 Proportional-Integral-Derivative Control

Proportional-Integral-Derivative (PID) control is a classic controller design that has found widespread use in robotics and process control. A PID controller provides an input to the plant, u, based on three terms. The first is a term proportional to the error. In other words, the greater the error, the greater the response. The second term provides a response based on the integral of the error. The final term provides a response based on the derivative of the error.

The integral term has an important role in reducing the rise-time (time for the output to reach a desired value) and steady-state error of the controller. The derivative term reduces overshoot and reduces the settling time of the system. A block diagram of a PID controller is shown in Figure A.2, with K_p, K_i, and K_d representing the proportional, integral, and derivative gains, respectively.

The PID control law requires a key assumption: that the plant is a linear system. This assumption does not hold for a robot arm, which is a complex, non-linear system, since the joints are *coupled*. Therefore, the acceleration of each joint affects the other joints. Furthermore, the response of the system changes based on the arm position.

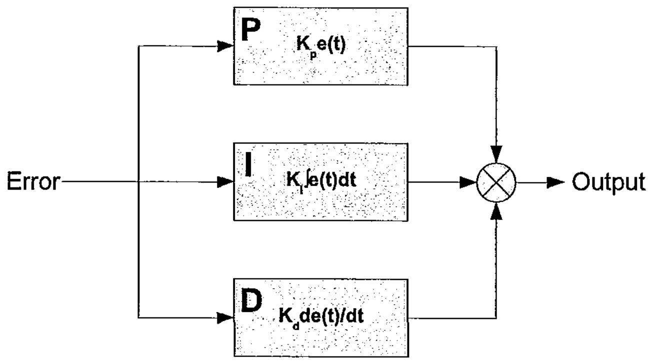


Figure A.2: Block diagram of a PID controller

For example, a fully extended arm will respond differently than one where the joints place the end-effector close to the base [47].

However, the PID controller is standard in most commercial robots. This is possible since the gearboxes attached between each joint and actuator effectively linearize and decouple the joints, provided the gear ratio is high enough. The most common robotic controller designs treat each joint as an independent linear system, with its own PID controller. Coupling effects are treated as disturbances in the system. An in-depth discussion of more sophisticated controllers is provided in [47].

A.3 Robot Kinematics

A robot is a complex mechanism, with multiple degrees of freedom (DOF) and the ability to move in three-dimensional space. A method must be used to keep track of the robot's position and orientation, as well as to compactly apply movements and rotations. This is accomplished through reference frames and transformation matrices.

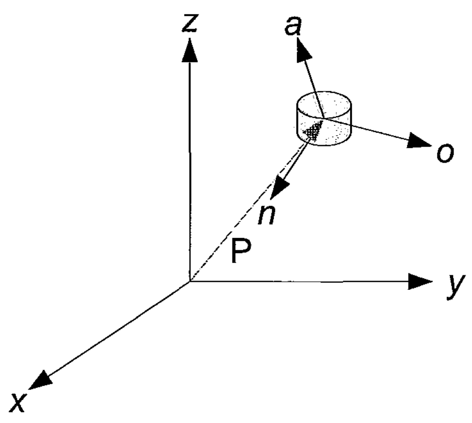


Figure A.3: An object in space represented by a frame

An object can be represented in 3-D space by attaching a frame to it and specifying its position and orientation relative to a reference frame. Therefore, an object can be specified with the matrix [48]:

$$\begin{pmatrix} n_x & o_x & a_x & P_x \\ n_y & o_y & a_y & P_y \\ n_z & o_z & a_z & P_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

where $\vec{n}$, $\vec{o}$, and $\vec{a}$ are the normal, orientation, and approach vectors and $\vec{P}$ is a position vector, as shown in Figure A.3. The frame position and orientation can be transformed by pre-multiplying by translation and rotation matrices, as shown below:

$$T = \begin{pmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$Rot(x,\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$Rot(y,\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$Rot(z,\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The matrix T is a translation matrix, while $Rot(x,\theta)$, $Rot(y,\theta)$, and $Rot(z,\theta)$ rotate the frame about the x, y, and z axes of the reference frame. The set of equations that specify the position of the end-effector as a function of the joint variables are called the *forward kinematics.* For a more complex robot, the forward kinematics are given by a series of matrices that transform a frame attached to the robot's base to its end-effector. This set of transformations fully specifies the position of the end-effector as a function of the joint variables [48].

A.4 Visual Servo Control

A.4.1 The Image Jacobian

In traditional robots, the position and velocity of each joint is measured with optical encoders. Using this data, the control loop can be closed, and the position of the end-effector determined using forward kinematics. While effective, this approach has several limitations. One limitation is that any flexibility and deflection in the robot's links will decrease positioning accuracy. Also, over time, chains, gears, and belts in the robot's drive mechanisms stretch and deform, further increasing positioning error. These limitations can be overcome by measuring the position of the end-effector with a camera and using the resulting data to close the control loop. This form of control is known as *visual servo.*

This work uses the Image-Based Visual Servo (IBVS) approach[1] to control a robot arm. In image-based visual servo, the motion of a set of points in an image are used to directly infer the position of the end effector and close the control loop. This is accomplished through an *interaction matrix*, or interchangeably, *image jacobian*, that relates the velocity of a 3-D point to its velocity in an image. The image jacobian, J_s, will now be derived.

The first step in the derivation is to relate a 3-D coordinate, $\vec{X} = (X, Y, Z)$, defined relative to the camera frame, with its 2-D projection in the camera image,

[1]Several variants of visual servo control have been introduced, with a detailed treatment of each provided in [49]

$\vec{x} = (x, y)$. This can be written as [50]:

$$x = X/Z = (u - c_u)/f\alpha \tag{A.1}$$

$$y = Y/Z = (v - c_v)/f \tag{A.2}$$

where (u, v) are the coordinates of the point in pixels, and c_u, c_v, f, and α are camera parameters[2]. The next step is to take the time derivative of the above equations, yielding [50]:

$$\dot{x} = \dot{X}/Z - X\dot{Z}/Z^2 = (\dot{X} - x\dot{Z})/Z \tag{A.3}$$

$$\dot{y} = \dot{Y}/Z - Y\dot{Z}/Z^2 = (\dot{Y} - y\dot{Z})/Z \tag{A.4}$$

We can now relate the velocity of the 3-D point to its spatial and rotational velocity components using the relation [50]:

$$\dot{\vec{X}} = -\vec{v_c} - \vec{\omega_c} \times \vec{X} \tag{A.5}$$

Or in component form:

$$\dot{X} = -v_x - \omega_y Z + \omega_z Y \tag{A.6}$$

$$\dot{Y} = -v_y - \omega_z Z + \omega_x Y \tag{A.7}$$

$$\dot{Z} = -v_z - \omega_x Z + \omega_y Y \tag{A.8}$$

where $\vec{v_c}$ is the spatial velocity and $\vec{\omega_c}$ is the rotational velocity. Relating the

[2](c_u, c_v) are the coordinates of the principal point, f is the focal length, and α is the ratio of pixel dimensions.

above equations to the velocity of the image point, $(\dot{x}, \dot{y})$, we obtain [50]

$$\dot{x} = -v_x/Z + xv_z/Z + xy\omega_x - (1+x^2)\omega_y + y\omega_z \tag{A.9}$$

$$\dot{y} = -v_y/Z + yv_z/Z + (1+y^2)\omega_x - xy\omega_y - x\omega_z \tag{A.10}$$

or in matrix form

$$\dot{\vec{x}} = J_s\vec{v_c} \tag{A.11}$$

where J_s is the image Jacobian, given by

$$J_s = \begin{pmatrix} -1/Z & 0 & x/Z & xy & -(1+x^2) & y \\ 0 & -1/Z & y/Z & 1+y^2 & -xy & -x \end{pmatrix}$$

The above equations specify the image Jacobian for one image point. However, at least three image points are required to control a 6DOF robot. This is required since the position and orientation of a single point is ambigious when a fully-actuated[3] manipulator is used. Fortunately, the image Jacobian is easily augmented to handle more than one feature points by simply stacking the single-point Jacobian described above. An augmented Jacobian matrix for three image points will be given by [50]

$$J_s = \begin{pmatrix} \vec{J_{s_1}} \\ \vec{J_{s_2}} \\ \vec{J_{s_3}} \end{pmatrix}$$

[3]A robot is considered fully-actuated when its end-effector can reach any arbitrary position and orientation. An arm needs 6DOF to be fully actuated

It is important to note that the image jacobian requires knowledge of the depth of the point, Z, relative to the camera frame. Therefore the value of Z must be estimated or approximated. Therefore, an approximation of J_s, given by $\hat{J}_s$ is used in practice. Fortunately, since the visual servo controller operates in a closed loop, only a rough approximation of Z is required, and the controller is fairly insensitive to changes in Z [49].

A.4.2 The Visual Servo Control Law

With the image jacobian defined in the above section, we are now ready to formulate the visual servo control law. The error to be minimized by the controller is typically represented by a set of image points, $\vec{s}$, and a set of desired values for the points, $\vec{s^*}$

$$\vec{e}(t) = \vec{s}(t) - \vec{s^*} \tag{A.12}$$

From the previous section, the relationship between the velocity of image points and camera velocity is given by

$$\dot{\vec{s}} = J_s \vec{v_c} \tag{A.13}$$

Therefore, the time variation of the error in relation to camera velocity is given by

$$\dot{\vec{e}} = J_s \vec{v_c} \tag{A.14}$$

To obtain an exponential decrease in the error, we require $\dot{\vec{e}} = -\lambda \vec{e}$, where λ is a constant. Solving equation A.14 for $\vec{v_c}$ and substituting the above equation, we obtain the proportional visual servo control law [50]:

$$\vec{v_c} = -\lambda J_s^+ \vec{e} \tag{A.15}$$

where J_s^+ denotes the Moore-Penrose pseudo-inverse. The above control law encapsulates the basic building block of visual servo control.

A.5 Derivation of the Joint-Space Visual Servo Control Law

The above section described a visual servo control law in terms of the velocity of the end effector, $\vec{v_c}$. To control a robot arm, this velocity must be translated into the appropriate angular joint velocities, so that individual joints can be controlled. Therefore, a visual servo control law in the *joint space* of the robot should be formulated [51].

The first step in the derivation is to consider the robot's *kinematic jacobian*, $J(\vec{\theta})$, where $\vec{\theta}$ refers to the joint variables. The kinematic jacobian relates differential motions of the robot's joints to the resulting differential motions of the end-effector [48]. Suppose we can write the position of the end-effector, relative to its base frame, as a function of the joint angles (assuming a 6DOF robot with only revolute[4] joints)

$$\vec{x} = \vec{f}(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6) \tag{A.16}$$

[4]Two types of joints are possible in a manipulator: revolute and prismatic. Prismatic joints move linearly (i.e. a piston) while revolute joints rotate in a circle

then a differential change in the position of the end-effector can be related to differential changes in the joint angles using the relation

$$\begin{cases} \partial x = \frac{\partial f_1}{\partial \theta_1}\partial\theta_1 + \frac{\partial f_1}{\partial \theta_2}\partial\theta_2 + \cdots + \frac{\partial f_1}{\partial \theta_6}\partial\theta_6 \\ \partial y = \frac{\partial f_2}{\partial \theta_1}\partial\theta_1 + \frac{\partial f_2}{\partial \theta_2}\partial\theta_2 + \cdots + \frac{\partial f_2}{\partial \theta_6}\partial\theta_6 \\ \partial z = \frac{\partial f_3}{\partial \theta_1}\partial\theta_1 + \frac{\partial f_3}{\partial \theta_2}\partial\theta_2 + \cdots + \frac{\partial f_3}{\partial \theta_6}\partial\theta_6 \\ \partial \omega_x = \frac{\partial f_4}{\partial \theta_1}\partial\theta_1 + \frac{\partial f_4}{\partial \theta_2}\partial\theta_2 + \cdots + \frac{\partial f_4}{\partial \theta_6}\partial\theta_6 \\ \partial \omega_y = \frac{\partial f_5}{\partial \theta_1}\partial\theta_1 + \frac{\partial f_5}{\partial \theta_2}\partial\theta_2 + \cdots + \frac{\partial f_5}{\partial \theta_6}\partial\theta_6 \\ \partial \omega_z = \frac{\partial f_6}{\partial \theta_1}\partial\theta_1 + \frac{\partial f_6}{\partial \theta_2}\partial\theta_2 + \cdots + \frac{\partial f_6}{\partial \theta_6}\partial\theta_6 \end{cases}$$

or, in matrix form

$$\begin{pmatrix} \partial x \\ \partial y \\ \vdots \\ \\ \partial \omega_z \end{pmatrix} = \begin{pmatrix} \frac{\partial f_1}{\partial \theta_1} & \frac{\partial f_1}{\partial \theta_2} & \cdots & \frac{\partial f_6}{\partial \theta_1} \\ \frac{\partial f_2}{\partial \theta_1} & \ddots & & \\ \vdots & & & \\ \frac{\partial f_6}{\partial \theta_1} & \cdots & & \frac{\partial f_6}{\partial \theta_6} \end{pmatrix} \begin{pmatrix} \partial \theta_1 \\ \partial \theta_2 \\ \vdots \\ \\ \partial \theta_6 \end{pmatrix}$$

Finally, the equation can be written compactly as

$$\vec{v_c} = J(\vec{\theta})\dot{\vec{\theta}} \tag{A.17}$$

We can see that equations A.17 and A.13 share a common term. The only difficulty is that the kinematic jacobian, $J(\vec{\theta})$, is written relative to the robot's base frame, while the image jacobian, J_s, is written relative to the camera reference frame. To fix this problem, $J(\vec{\theta})$ must be multiplied by a transformation matrix, as described in section A.3 to transform it to the camera's reference frame. Since the jacobian deals with velocities, only the rotation of the frame must be transformed. This is accomplished by multiplying the kinematic jacobian by a rotation transformation matrix, V, given by [51]

$$V = \begin{pmatrix} R & 0 \\ 0 & R \end{pmatrix}$$

where R is a rotation transformation matrix from the robot base frame to the camera frame. The resulting equation gives

$$\dot{\vec{s}} = J_s V J(\vec{\theta})\dot{\vec{\theta}} \tag{A.18}$$

The joint space visual servo control now can now be formulated, as in the previous section, however this time in terms of joint angle velocities

$$\dot{\vec{\theta}} = -\lambda[J_s V J(\vec{\theta})]^+ \vec{e} \tag{A.19}$$

A.5.1 A Joint-Space PID Controller

The above control law is a simple proportional controller. The performance of the system can be improved (in terms of rise time, overshoot, and steady-state error) by adding derivative and integral terms. The final PID control law in the robot's joint space is therefore:

$$\dot{\vec{\theta}} = -\lambda[J_s V J(\vec{\theta})]^+ \vec{e} + \mu[J_s V J(\vec{\theta})]^+ \int \vec{e} dt + \kappa[J_s V J(\vec{\theta})]^+ \frac{\partial \vec{e}}{\partial t} \tag{A.20}$$

The above represents the full visual servo control law used to actuate the robot in this project.

Appendix B

Robotic System Design and Implementation

B.1 Video Acquisition

The Firewire (or IEEE 1394) interface is commonly used for live video acquisition due to its simplicity, availability, and high bandwidth. In this work, a Sony (model number) CCD camera at a resolution of 640×480 and acquisition rate of 30 frames per second, connected to a Firewire port, was used to acquire live video. The Digital Camera (DCAM) protocol was used to communicate with the camera and receive its data. DCAM communication was implemented using the freely available Carnegie Mellon University 1394 DCAM driver, available in [?]. The DCAM driver was originally developed by Iwan Ulrich at the CMU Robotics Institute.

B.2 CRS 255 Open Architecture Manipulator

Robotic experiments were performed using a CRS A255 5 degree of freedom robot arm, as pictured in Figure B.2. The robot has 5 joints: a base, shoulder, elbow, and 2 wrist joints (pitch and roll). Since the A255 is missing a degree of freedom in its wrist, the arm cannot achieve an arbitrary position and orientation. However, since the experiments in this work only require the arm to move in a vertical plane, only two degrees of freedom are necessary.

The CRS controller was modified by request by Quanser Consulting to form an *open architecture* robot. The internal PID controller provided by CRS was bypassed, allowing voltage to be sent directly to the motors using a PC interface board (the MultiQ data acquisition board, also provided by Quanser) [52]. This arrangement is shown in Figure B.1. The open architecture configuration allows a visual servo controller to be implemented in software, with commands sent directly to the robot's joints.

B.2.1 Link Coupling

The A255 robot differs from most common robot configurations because it follows a *decoupled* arm configuration [52]. A decoupled configurations means that as a joint rotates, other joints further up the arm maintain a costant angle with the baseplane. For example, if the shoulder joint rotates by 10 degrees, the elbow joint will rotate so that the elbow link maintains a constant angle with the baseplane. Link coupling is illustrated in Figure B.3. This effect is important to keep in mind when calculating the kinematic jacobian that depends on joint angles.

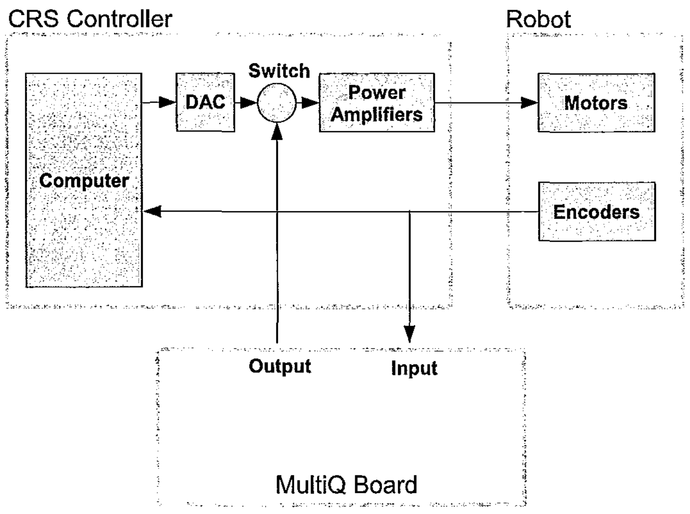


Figure B.1: Open architecture configuration for the CRS A255 robot arm

B.3 Controller Design and Implementation

B.3.1 Real-Time Control using WinCon

The Quanser MultiQ board allows the robot to be controlled in real-time with a controller running on a standard PC. Unfortunately, standard PC operating systems (such as Windows) are not built to run real-time software. In general, this means that processes running on the OS are not guarenteed to meet hard-deadlines, and may be pre-empted by other processes running on the PC. The OS must therefore be extended with a real-time OS extension, such as Ardence RTX in this case. In addition, Quanser Consulting software, called *WinCon*, is used to compile controller code to real-time executable code and to communicate with the robot.

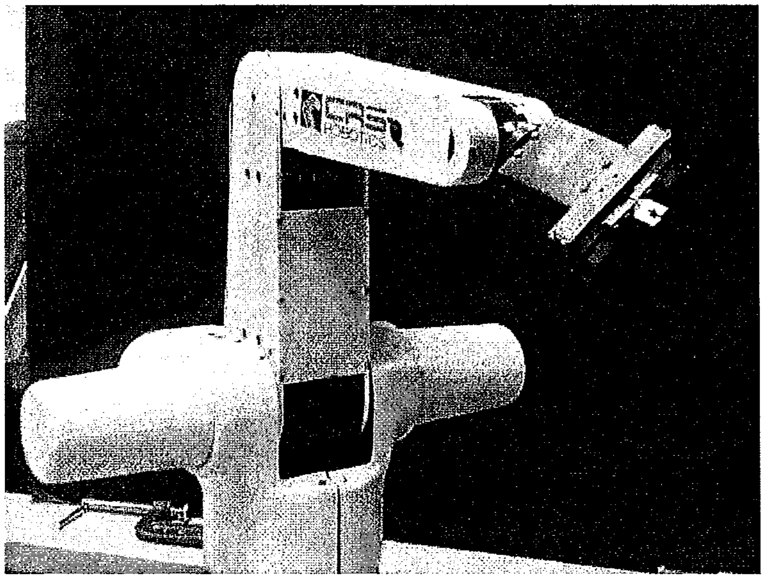


Figure B.2: The CRS A255 5 DOF manipulator

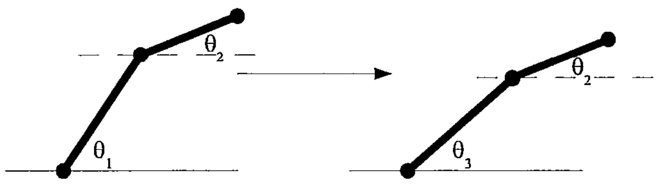


Figure B.3: Illustration of link coupling during manipulator motion. When θ_1 changes (to θ_3), θ_2 remains constant with respect to the base plane.

B.3.2 Controller Implementation with Simulink

The Matlab Simulink environment has become an industry standard for developing, prototyping, and analyzing control systems. WinCon, together with the Real-Time Toolbox, allows Simulink models to be compiled to real-time C code capable of execution under RTX. Designing the visual servo controller in Simulink drastically reduces development and troubleshooting time.

The visual servo controller is built using a set of standard Simulink building blocks provided by Quanser Consulting. Some blocks encapsulate C code that interfaces with the MultiQ board (and hence the robot), while other blocks form a basic Proportional-Derivative (PD) controller for each joint. The most basic building block is a *joint* block, as shown in Figure B.4(a). The joint block performs two functions. It reads the joint encoders and converts the reading to a joint angle ($J1$) and joint velocity ($\dot{J}1$). The block also accepts a voltage input ($V1$) and transmits the voltage level to the MultiQ board, where it will be sent to the power amplifiers and the motors.

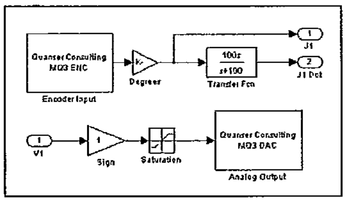
The next building block is a PD controller for each joint, as pictured in Figure B.4(b). The block implements the control law

$$V = K_p(\theta_d - \theta) + K_d\dot{\theta} \tag{B.1}$$

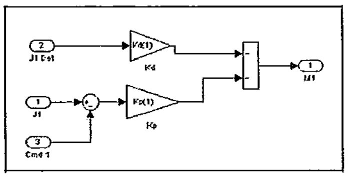
where V is the voltage applied to the motor, θ is the joint angle and θ_d the desired joint angle (both in degrees), K_p is the proportional gain (in V/Deg), and K_d is the derivative gain (in V/(Deg/s)). Finally, the joint and PD blocks are combined to form a closed-loop control block for each joint. This block is pictured in Figure B.4(c).

B.3.3 Visual Servo Controller in Simulink

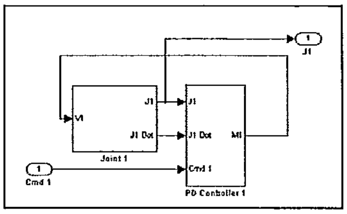
The goal of the visual servo controller is to control the position of the robot in a vertical plane. In other words, two joints of the robot will be directly controlled to move the (x, y) coordinates of the end-effector, as shown in Figure. The visual servo controller uses the joint space control law derived in Appendix A to determine the required joint angles to move toward a set position. These joint angles are then given as desired joint angles to each joint's PD controller. The output of the PD controller is the voltage needed to actuate the joint motors. With this configuration,



(a) Joint Block



(b) PD Block



(c) Closed-loop Joint Controller

Figure B.4: Quanser Simulink building blocks for robotics control

the controller is actually structured as two closed-loop controllers. The joint PD controllers stabilize each joint while the visual servo controller provides closed-loop position feedback. This configuration is termed *dynamic look-and-move* [49].

The dynamic look-and-move method is common in the visual servo literature, and is significantly more stable than using the visual servo controller to directly provide a voltage input to the actuators. This is due to the sampling rate of each loop. Visual servo is limited by the sampling rate of the camera, which is often limited to 30 Hz. The encoder controller loop however, is free to run at a much higher speed since much less data needs to be processed and transmitted. The joint controller operates at 1 kHZ. Using the two controllers in tandem uses the advantages of both: increased positioning accuracy of visual servo and more stable joint control of the encoder stabilized loop.

A Simulink block diagram was developed, taking into account the above concepts. The controller block diagram is shown in Figure B.5. The controller is divided into two

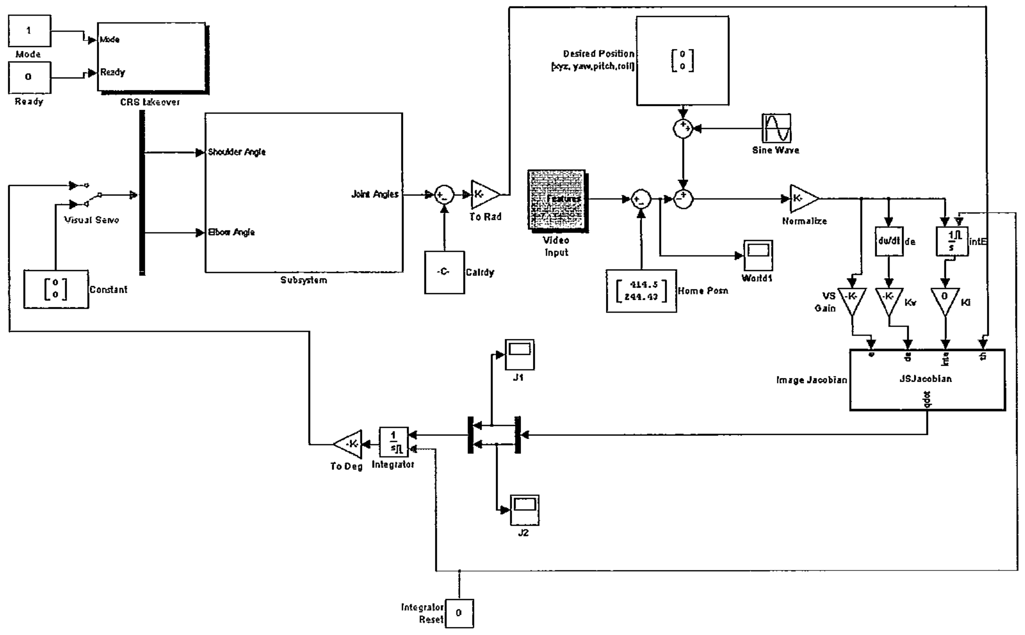


Figure B.5: Simulink block diagram of visual servo controller

sections, with a visual servo loop and an encoder loop for each joint. The visual servo loop begins with the image features block. This block constantly receives the current position, in image space, of the robot's end effector. This position is subtracted from the desired position, producing the error signal. The error signal is then multiplied by the appropriate gains. The error signal, derivative of the error, and integral of the error terms go through the visual servo block, which implements equation A.20. The resulting joint velocities, $\dot{\theta}$ are integrated, producing θ set-points for the joint controller. These joint commands are translated to motor voltages by each joint's PD control blocks, as discussed in the previous section.

B.3.4 Camera and Controller Communication with Ethernet

The robot control and optical flow tasks operate on separate PCs. This occurs since both need specific hardware (i.e. NVIDIA graphics card, MultiQ board) to operate. A dedicated Ethernet link, operating at 100 Mb/s, connects the two computers. The TCP/IP protocol was used to transmit the data in packets. Each feature point coordinate (i.e. (x, y)) is sent as one packet, to ensure that a complete visual servo point is received, even if packets are lost.

Appendix C

CUDA Source Code

This appendix includes listings of the key source code used to implement the foveated optical flow system, with accompanying explanation. Since CUDA source code is distinct from standard C++ code, the inclusion of the source code illustrates the implementation details of the algorithms presented in Chapter 3. The log-polar transform, optical flow, and foveation point selection kernels are detailed in this chapter, along with the host-side code responsible for launching the kernels.

C.1 Log Polar Transform Kernel

The log polar transform kernel accepts a cartesian image as an input (*d_cart*) and returns a log-polar transform image (*d_log*). Source code for the kernel is included in the listing below. It is important to realize that each thread will execute the same instructions, but will access different data. Furthermore, the kernel is executed by the GPU and it is assumed that the images are already in global memory. The first step in the code is to calculate the array index that the thread will access in the LPT image. This is done by multiplying the index of the current thread block (*blockIdx.x*) by the dimension of each block (*blockDim.x*) and adding the index of the current thread (*threadIdx.x*), as shown on line 10. The *IMUL* macro is used to perform an accelerated integer multiplication, since integer multiplication is slower on the GPU than floating point multiplication.

Once the thread has calculated the array index of the LPT image it is accessing, this is transformed to a (u, θ) value (line 20). This information is then used to calculate the correct (x, y) coordinate to sample in the cartesian image using the log-polar transform equations. Padding correction is then applied and the correct pixel is transferred from the cartesian image to the log-polar image. These operations are carried out starting on line 26 of the listing.

Listing C.1: Log Polar Transform kernel

```
 2  // GPU KERNEL (LOG POLAR TRANSFORM)

 4  __global__ void logpolartrans_gpu(float *d_cart, float *d_log, int w, int padsize, unsigned long
        logSize, float fovea_cx, float fovea_cy, float rho_o, float u_min, float du, int numRings)
    {
 6          int uindx, thetaindx, x, y;;
            float rho, u, theta;
 8
            // Generate index of log polar image thread in current block accesses
10          unsigned long idx = IMUL(blockDim.x, blockIdx.x) + threadIdx.x;
            unsigned long idx2;
12
            idx = (idx <= logSize - 1) ? idx : logSize - 1;
14
            // Find index of log polar image in (u, theta) format
16          uindx = floorf(idx / (float)w);
            thetaindx = idx - IMUL(uindx, w);
18
            // Convert uindx and thetaindx to u and theta
20          u = (float)uindx * du + u_min;
            theta = thetaindx * CONVFACTOR;
22
            // Calculate rho
24          rho = rho_o * __expf(u);
```

```
        // Calculate cartesian x and y coordinates
        x = floorf( rho * cosTable[(int)theta] + 0.5f) + fovea_cx;
        y = floorf( rho * sinTable[(int)theta] + 0.5f) + fovea_cy;

        x = ( x >= 0) ? x : 0; //make sure not out of bounds
        x = ( x <= imWidth-1) ? x : imWidth - 1;
        y = ( y >= 0) ? y : 0;
        y = ( y <= imHeight-1) ? y : imHeight - 1;

        // Get index in cartesian map
    unsigned long cartindx = 0;
        cartindx = IMUL(y, imWidth) + x;

        // Apply padding correction
        y = floor(idx / (float)w);
        x = idx - y*w;
        idx2 = y * (w + padsize) + x;

        // Store pixel in LPT image
        d_log[idx2] = d_cart[cartindx];

        __syncthreads();
}
```

C.2 Optical Flow Kernel

The optical flow kernel is the most computationally-intensive kernel in the system. As discussed in Chapter 3, the kernel takes as an input two LPT images, corresponding to frames $I(t)$ and $I(t-dt)$, bound to textures. The use of textures accelerates memory access since texture memory is cached, unlike regular global memory access. The downside is that texture memory is read-only, therefore the resulting optical flow vectors are stored in the arrays *d_x* and *d_y*, both in global memory.

The kernel begins by allocating shared memory for the pixel patches P_1 and P_2, as well as space to stage a coalesced write to global memory (*StoreStage_x* and *StoreStage_y*). It is important to note that the shared memory is allocated one for the entire thread block, using the *__shared__* specifier, as shown on lines 5-9. Once shared memory is allocated, each thread calculates the index of the pixel it is accessing in the LPT image. This data is then loaded into shared memory. The *tex2D* function is used to access texture memory, illustrated on line 37. The *F1PATCH* and *F2PATCH* macros are used to write to the shared memory arrays *f1Patch* and *f2Patch.*

It is important to note that that the procedure for loading data from frame $I(t)$ is different than for frame $I(t-dt)$. Each thread block consists of 16×16 threads. Therefore, each thread can load one pixel from frame $I(t-dt)$, completing the loading operation in one step. Pixel patch P_2, which is loaded from frame $I(t)$, has an additional apron required for the correlation-based search, therefore more pixels exist than threads in the block. Therefore, the load is performed in two steps. One the load has finished, all threads are synchronized and the correlation search algorithm begins. The load into patch P_1 occurs on lines 32-37, while the load into patch P_2 occurs on lines 44-62.

Only threads that correspond to active pixels (not to apron pixels) participate in the search. Each thread calculates one motion vector. This is done by stepping through every possible displacement in frame $I(t)$ and for each displacement calculating the SSD (lines 68-122). Once the minimum SSD has been found, the corresponding motion vector is stored in shared memory. Once all motion vectors in the thread block have been found, the threads are again synchronized and the coalesced store to global memory begins. Since the thread block is 16×16 threads in size, but the storage stage is 32×16 pixels wide, each thread is responsible for writing two pixels. Since there are two storage stages for the x and y components, each thread performs four write operations in total. A width of 32 for the storage stage is required to meet memory alignment requirements. The coalesced store to global memory is organized starting on line 128.

Listing C.2: Optical Flow kernel

```
   // GPU KERNEL (OPTICAL FLOW)
 2 __global__ void opticalflowxy_gpu(float *d_f1, float *d_f2, float *d_x, float *d_y, int w, int h,
       unsigned long imarrsize, float du, float u_min, float rho_o, bool foveated = true)
   {
 4         // Shared memory for pixel patches
           __shared__ float f1Patch[16][16];
 6         __shared__ float f2Patch[16+2*n][16+2*n];
           // Shared memory for coalescing global memory store
 8         __shared__ float StoreStage_x[16][32];  // x component
           __shared__ float StoreStage_y[16][32];  // y component
10
           float sum=0;
12         volatile int ix, iy;
           volatile int ox, oy; // Offset of current thread block, important for deformation
               correction
14
           const float threadIdx_x = IMUL(MODDIMX, blockIdx.x);
16         const float threadIdx_y = IMUL(MODDIMY, blockIdx.y);

18         ox = threadIdx_x; oy = threadIdx_y;

20         // Initialize storestage array
           StoreStage_x[threadIdx.y][threadIdx.x]  = 0.0f;
22         StoreStage_x[threadIdx.y][threadIdx.x+16]= 0.0f;
           StoreStage_y[threadIdx.y][threadIdx.x]  = 0.0f;
24         StoreStage_y[threadIdx.y][threadIdx.x+16]= 0.0f;

26         // ============================================================
           // Calculating variables for the storage stage (coalesced access)
28         int prefix;
           IX = floor((threadIdx_x) / 16.0f) * 16;
30         prefix = (threadIdx_x) - IX;

32         // Index of pixel accessed by current block in f1
           IY = threadIdx_y + threadIdx.y + n;
34         IX = threadIdx_x + threadIdx.x + n;

36         // Load 16 x 16 patch from f1 (of which 12 x 12 pixels are active)
           F1PATCH(threadIdx.y, threadIdx.x) = 255.0f * tex2D(f1, IX / (float)w, IY / (float)h);
38
           // ============================================================
40         // Load 20 x 20 patch from f2
           // making 16x16 (256) threads load 400 pixels
```

```
        // Generate a variable (thread_num) that goes from 0 to 255. Then refactor this number
            into x, y coordinates
        // assuming we're working with a 20x20 square
        volatile int thread_num = threadIdx.y * 16 + threadIdx.x;
        int tx, ty;
        ty = floor(thread_num / (float)(I2Size));
        tx = thread_num - IMUL(ty, I2Size);

        // Batch 1 ->First 256 pixels
        IY = threadIdx_y + ty;
        IX = threadIdx_x + tx;
        F2PATCH(ty, tx) = 255.0f * tex2D(f2, IX / (float)w, IY / (float)h);

        // Batch 2
        thread_num = IMUL(threadIdx.y, 16) + threadIdx.x + 256;
        ty = floor(thread_num / 20.0f);
        tx = thread_num - IMUL(ty, 20);

        if (thread_num <= (I2Size * I2Size)) {
                IY = threadIdx_y + ty;  IX = threadIdx_x + tx;
                F2PATCH(ty, tx) = 255.0f * tex2D(f2, IX / (float)w, IY / (float)h);
        }
        __syncthreads();


        // ================================================================
        // Perform search, each thread takes one pixel in final image (i.e: one of the 12x12
            threads)
        if (threadIdx.x < ActivePixels && threadIdx.y < ActivePixels) {
                float minVal = 10000000.0f;     float threshVal = 0.01f;
                volatile float d_x=0.0f, d_y=0.0f, prod;
                volatile int x, y, i, j, indx1, indx2;
                // Deformation calculation variables
                float u1, u2, r1, r2, x1, x2, y1, y2;

                IX = threadIdx.x + I2Apron;     IY = threadIdx.y + I2Apron;

                for (y = IY - n; y <= IY + n; y++) {
                        for (x = IX - n; x <= IX + n; x++) {
                                // Inner search, evaluating match strength
                                SUM = 0.0f;

                                for (j = y - v; j <= y + v; j++) {
                                        for (i = x - v; i <= x + v; i++) {
                                                indx1 = threadIdx.y + v + (j - y);
                                                indx2 = threadIdx.x + v + (i - x);
                                                prod = F2PATCH(j,i) - F1PATCH(indx1,indx2);
                                                SUM = SUM + prod * prod;
                                        }
                                }

                if (SUM < minVal || (SUM == minVal && y == IY && x == IX)) {
                    minVal = SUM;

                    // ============== DEFORMATION CORRECTION ==============
                    // For our purposes, r = y, theta = x

                    if (foveated) {  // INCLUDE DEFORMATION CORRECTION
                        u1 = du * (oy + IY + n + v) + u_min;
                        u2 = du * (oy + y + n + v) + u_min;
                        r1 = exp(u1)*rho_o;
                        r2 = exp(u2)*rho_o;

                        x1 = r1 * cosTable[ox + IX + n + v];
                        x2 = r2 * cosTable[ox + x + n + v];
                        y1 = r1 * sinTable[ox + IX + n + v];
                        y2 = r2 * sinTable[ox + x + n + v];

                        indx1 = (x2-x1); indx2 = (y2-y1);
                        d_x = indx1; d_y = indx2;
                    }
                    else {          // WITHOUT DEFORMATION CORRECTION
                        indx1 = -(x-IX); indx2 = -(y-IY);
                        d_x = indx1; d_y = indx2;
                    }
                }
            }
        }

        // Store motion vectors in shared memory
        StoreStage_x[threadIdx.y][threadIdx.x+prefix] = d_x;
        StoreStage_y[threadIdx.y][threadIdx.x+prefix] = d_y;
    }

    __syncthreads();

    // ============= COALESCED STORE TO GLOBAL MEMORY ==================
    // Starting point of write operation, making sure its a multiple of 16
    if (threadIdx.y < ActivePixels) {
        IY = threadIdx_y + threadIdx.y;
        IX = floor((threadIdx_x / 16.0f)) * 16;
        volatile unsigned long idx;

        //Write block 1, 0->16
        IX = IX + threadIdx.x; idx = IMUL(IY,w) + IX - (n+v);

        if (threadIdx.x >= prefix && threadIdx.x <= prefix + ActivePixels-1) {
            d_x[idx] = StoreStage_x[threadIdx.y][threadIdx.x];
            d_y[idx] = StoreStage_y[threadIdx.y][threadIdx.x];
        }
```

```
                else {
                        d_x[idx] = d_x[idx];
                        d_y[idx] = d_y[idx];
                }

                //Write block 2, 16->32
                IX = floor(threadIdx_x / 16.0f) * 16;
                IX = IX + threadIdx.x + 16;
                idx = IMUL(IY,w) + IX - (n+v);

                if (threadIdx.x+16 >= prefix && threadIdx.x+16 <= prefix + ActivePixels-1) {
                        d_x[idx] = StoreStage_x[threadIdx.y][threadIdx.x+16];
                        d_y[idx] = StoreStage_y[threadIdx.y][threadIdx.x+16];
                }
                else {
                        d_x[idx] = d_x[idx];
                        d_y[idx] = d_y[idx];
                }
        }
}
```

C.3 Foveation Point Selection Kernel

The foveation point selection kernel finds the next foveation point for the optical flow system. The foveation point is chosen to be the centroid of the region of the image that has experienced the greatest movement. The first step, therefore, is to perform a motion segmentation step, reducing the motion vectors below a threshold to zero. A source code listing for the foveation point selection kernel is shown below.

The kernel begins by determining which array element in the motion vector array is accessed by each thread (line 11). The corresponding (u, θ) coordinate is then calculated and used to find the correct array element to access given the extra padding added to the image (line 25). The magnitude of the motion vector is then found and stored in the shared memory array *xLocal*. The value is then thresholded, with 1 being stored in the array if the motion vector magnitude was above the threshold and

0 being stored otherwise. This is performed on lines 28 and 29.

The (x, y) coordinate corresponding to the current (u, θ) is then calculated. The local centroid is then calculated and stored in the *xLocal* and *yLocal* arrays. In other words, *xLocal* and *yLocal* store the x and y components of the centroid. It should be noted that the *xLocal* array serves two purposes. Before it stores the x component of the centroid, it stores the thresholded motion vector magnitude. This was done to conserve shared memory space. The thresholding operation is performed on lines 41-43.

Once all pixels have been multiplied by the threshold, a single thread in the block completes the algorithm by summing up the centroids and dividing by the number of non-thresholded pixels left. This is shown on lines 50-64. The local centroid is then stored in global memory and returned to the host for further processing.

Listing C.3: Foveation Point Selection Kernel

```
1  // FOVEATION POINT SELECTION
   __global__ void interestcent_gpu(float *d_cx, float *d_cy, int *d_npts, float *d_x, float *d_y,
       int w, float padding, float du, float u_min, float rho_o, float fovea_cx, float fovea_cy,
       unsigned int sizeLog)
3  {
       __shared__ float xLocal[ThreadsPerBlock]; // shared memory for local calculation of
           centroid
5      __shared__ float yLocal[ThreadsPerBlock];

7      //Each block loads its data into shared—mem
       //To conserve shared mem, d_mag values are first stored in xLocal
9      unsigned long idx, idxPAD;

11     idx = IMUL(blockDim.x, blockIdx.x) + threadIdx.x;
       idx = (idx <= sizeLog - 1) ? idx : sizeLog - 1;
13
       // Each thread converts idx value to (u, theta), then (x,y)
15     // ———> idx to (u, theta)
       float uindx, theta, rho, u, x, y;
17
       uindx = floor(idx / (float)w);
19     theta = idx - uindx * w;
       // ———> (u, theta) to (x, y)
```

```
        u = ((float)uindx + n + v) * du + u_min;
        rho = rho_o * __expf(u); // Calculate rho

        // Apply PADDING correction
        idxPAD = uindx * (w + padding) + theta;

        // Thresholding
        xLocal[threadIdx.x] = sqrt(d_x[idxPAD]*d_x[idxPAD] + d_y[idxPAD]*d_y[idxPAD]);
        xLocal[threadIdx.x] = (xLocal[threadIdx.x] > THRESHOLD && xLocal[threadIdx.x] < 20.0f) ?
            1.0f : 0.0f;

        // Calculate cartesian x and y coordinates
        x = floorf( rho * cosTable[(int)theta + n + v] + 0.5f) + fovea_cx;
        y = floorf( rho * sinTable[(int)theta + n + v] + 0.5f) + fovea_cy;

        x = ( x >= 0) ? x : 0;
        x = ( x <= imWidth-1) ? x : imWidth - 1;
        y = ( y >= 0) ? y : 0;
        y = ( y <= imHeight-1) ? y : imHeight - 1;

        // multiply y by d_mag[idx] (stored in xLocal[threadIdx]), store in yLocal
        yLocal[threadIdx.x] = y * xLocal[threadIdx.x];
        // multiply x by d_mag[idx]
        xLocal[threadIdx.x] = x * xLocal[threadIdx.x];

        // One thread calculated centroid, calculates n (# of non-thresholded pixels)
        // Stores in global me
        int i=0, npts=0; float cx=0.0f, cy=0.0f;

        __syncthreads();

        // Calculate centroid
        if ((float)threadIdx.x == 0.0f) {

                for (i = 0; i < ThreadsPerBlock; i++) {
                        cx += xLocal[i]; cy += yLocal[i];

                        if (xLocal[i] > 0)
                                npts++;
                }

                // Write local centroid to global memory
                d_cx[blockIdx.x] = cx;  d_cy[blockIdx.x] = cy;
                d_npts[blockIdx.x] = npts;
        }
        __syncthreads();
}
```

C.4 Host Code

The host code is responsible for allocating and transferring memory between the host and graphics card and launching the CUDA kernels. As such, the host code can be considered the top level code for the system. Several parts of the code have been abbreviated to save space. Likewise, the code that loads and stores video sequences in system RAM is not shown. It is also important to note that the host and GPU code can run concurrently. In other words, the host does not stop executing code after a kernel is launched on the GPU. Several conditions on GPU/CPU concurrency do exist however, and are detailed more fully in section 3.1.8. The source code listing for the host-side code is presented below.

The first line initializes the graphics card, followed by a series of variable declarations and memory allocations which have been omitted to save space. Prior to the start of the video processing loop, the first frame of the video sequence is copied from system RAM to GPU global memory using the *cudaMemcpy* function call. The arrays *d_f1* and *d_f2* store frames $I(t-dt)$ and $I(t)$ in cartesian coordinates while arrays *d_log1* and *d_log2* store frames $I(t-dt)$ and $I(t)$ in log-polar coordinates. All four arrays are stored in GPU global memory so that they can be accessed by the kernels described above.

The video processing loop now begins and steps through all frames in a given video sequence, starting on line 13. The log-polar transform of frame $I(t-dt)$ is computed first, frame $I(t)$ is copied to global from system memory and the log-polar transform of that frame is found. The pointers to the two frames are now swapped so that on the next iteration of the loop frame $I(t)$ is now $I(t-dt)$. These steps are executed on lines 15-24. The next step in the process is texture memory setup. As discussed in chapter

3, the optical flow kernel uses texture memory to accelerate memory access. In order to *bind* an image to texture memory, the data must be transferred to a construct called a *CUDA array*. This is accomplished using the *cudaMemcpyToArray* function, after which the array is bound as a texture using the *cudaBindTextureToArray* call. Texture binding is performed on lines 30-35. The optical flow kernel now executes, with the motion vectors being returned in the arrays *d_x* and *d_y*, as shown on line 42.

The next step in processing is the determination of the next foveation point. The CUDA kernel *interestcent_gpu* calculates the local centroids, as discussed above. This kernel is called on like 45. The results are then transferred to system RAM where the CPU calculates the final centroid. This value is then assigned as the next foveation point. After all frames have been processed, memory on both the graphics card and host is released and the graphics card is released.

Listing C.4: Host source code

```
   // Main Program - HOST
 2 int main(int argc, char** argv)
   {
 4         // Initialize device
           CUT_DEVICE_INIT(argc, argv);
 6
           ( ... Variable declarations and memory allocation ... )
 8
           // Copy first frame of video sequence to device memory
10         cudaMemcpy(d_f1, imFrames[0].h_img, memsizeArr, cudaMemcpyHostToDevice);

12         //---------------------- PROCESS VIDEO FRAMES ----------------------
           for (j = 1; j <= NUMFRAMES; j++) {
14                 // Calculate Log polar transform of frame I(t-dt)
                   logpolartrans_gpu <<<gridSizeTrans, blockSizeTrans>>> (d_f1, d_logl, NTheta,
                        PADDING, sizeLog, fovea_cx, fovea_cy, rho_o, u_min, du, NRings);
16
                   // Copy current frame, I(t), to device memory
18                 cudaMemcpy(d_f2, imFrames[j].h_img, memsizeArr, cudaMemcpyHostToDevice);

20                 // Calculate Log Polar transform of frame I(t)
```

```
logpolartrans_gpu <<<gridSizeTrans, blockSizeTrans>>> (d_f2, d_log2, NTheta,
    PADDING, sizeLog, fovea_cx, fovea_cy, rho_o, u_min, du, NRings);

// Swap frame I(t) and I(t-dt) by flipping pointers
swapPtr(&d_f1, &d_f2);

//--------------------------------------------------------------
//----> Optical Flow
//--------------------------------------------------------------
// Transfer LPT transformed images to CUDA arrays so they can be bound to textures
    for optical flow step
cudaMemcpyToArray(f1_array, 0, 0, d_log1, memsizePad, cudaMemcpyDeviceToDevice);
cudaMemcpyToArray(f2_array, 0, 0, d_log2, memsizePad, cudaMemcpyDeviceToDevice);

// Bind the array to the texture
cudaBindTextureToArray( f1, f1_array, channelDesc);
cudaBindTextureToArray( f2, f2_array, channelDesc);

// Initialize kernel size
dim3 blockSize (16, 16);
dim3 gridSize (gridWidth, gridHeight);

// Execute Optical Flow Kernel
opticalflowxy_gpu <<<gridSize, blockSize>>> (d_log1, d_log2, d_x, d_y, logWidth,
    logHeight, sizePad, du, u_min, rho_o, true);

// ===========>> INTEREST PT SELECTION <<============
interestcent_gpu <<<gridSizeTrans, blockSizeTrans>>>(d_cx, d_cy, d_npts, d_x, d_y,
    logWidth, PADDING, du, u_min, rho_o, fovea_cx, fovea_cy, sizeLog);

// move results back to host
cudaMemcpy(h_cx, d_cx, sizeof(float)*calcGridSize, cudaMemcpyDeviceToHost);
cudaMemcpy(h_cy, d_cy, sizeof(float)*calcGridSize, cudaMemcpyDeviceToHost);
cudaMemcpy(h_npts, d_npts, sizeof(int)*calcGridSize, cudaMemcpyDeviceToHost);

// Sum up local centroids returned from GPU and return the final
// centroid: (cx, cy)
CalculateCentroidOnCPU();

// Assign new foveation pt
fovea_cx = cx;
fovea_cy = cy;

// Synchronize threads
cudaThreadSynchronize();

// Move result back to host
cudaMemcpy(h_dx, d_x, memsizePad, cudaMemcpyDeviceToHost);
cudaMemcpy(h_dy, d_y, memsizePad, cudaMemcpyDeviceToHost);
```

```
        }

        ( ... Release Memory ... )

        // Release device
        CUT_EXIT(argc, argv);
}
```

Bibliography

[1] NVIDIA Corporation, *CUDA Programming Guide*, 2007.

[2] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A.E. Lefohn, and T. Pucell, "A survey of General-Purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[3] J. Fung and S. Mann, "Using graphics devices in reverse: Gpu-based image processing and computer vision," in *Proc. ICME2008*, 2008, pp. 9–12.

[4] T.D.R. Hartley, U. Catalyurek, A. Ruiz, F. Igual, and M. Ujaldon, "Biomedical image analysis on a cooperative cluster of gpus and multicores," in *Proc. of ACM ICS*, 2008, pp. 15–25.

[5] O.M. Lozano and K. Otsuka, "Simultaneous and fast 3d tracking of multiple faces in video by gpu-based stream processing," in *Proc. ICASSP2008*, 2008, pp. 713–716.

[6] H. Scherl, B. Keck, M. Kowarschik, and J. Hornegger, "Fast gpu-based ct reconstruction using the common unified device architecture (cuda)," in *Proc. NSS2007*, 2007, pp. 4464–4466.

[7] Y. Allusse, P. Horain, A. Agarwal, and C. Saipriyadarshan, "GpuCV: an Open-Source GPU-Accelerated framework for image processing and computer vision," Vancouver, 2008, pp. 1089–1092.

[8] J.L Barron, D.J Fleet, and S.S Beauchemin, "Performance of optical flow techniques," *International Journal of Computer Vision*, vol. 12, no. 1, pp. 43–77, 1994.

[9] B. Horn and B. Schunck, "Determining optical flow," *Artificial Intelligence*, vol. 17, pp. 185–203, 1981.

[10] B. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision," Vancouver, 1981, pp. 674–679.

[11] H.-H. Nagel and W. Enkelmann, "An investigation of smoothness constraints for the estimation of displacement vector fields from image sequences," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 8, no. 5, pp. 565–593, 1986.

[12] P. Anandan, "A computational framework and an algorithm for the measurement of visual motion," *International Journal of Computer Vision*, vol. 2, no. 3, pp. 283–310, 1989.

[13] M.J. Black and P. Anandan, "The robust estimation of multiple motions: parametric and piecewise smooth flow fields," *Computer Vision and Image Understanding*, vol. 63, no. 1, pp. 75–104, 1996.

[14] F. Glazer, G. Reynolds, and P. Anandan, "Scene matching through hierarchical correlation," in *Proc. CVPR*, Washington, 1983, pp. 432–441.

[15] T. Camus, "Real-time quantized optical flow," *Real-Time Imaging*, vol. 3, no. 2, pp. 71–86, 1997.

[16] D.J. Heeger, "Optical flow using spatiotemporal filters," *International Journal of Computer Vision*, vol. 1, no. 4, pp. 279–302, 1988.

[17] D.J. Fleet and A.D. Jepson, "Computation of component image velocity from local phase information," *International Journal of Computer Vision*, vol. 5, no. 1, pp. 77–104, 1990.

[18] A.M. Waxman and K. Wohn, "Contour evolution, neighbourhood deformation and global image flow: Planar surfaces in motion," *Intern. J. Robotics Res.*, vol. 4, no. 3, pp. 95–108, 1985.

[19] B. Buxton and H. Buxton, "Computation of optic flow from the motion of edge features in image sequences," *Image Vis. Comput.*, vol. 2, no. 1, pp. 59–74, 1984.

[20] P. Symes, *Video Compression: Fundamental Compression Techniques and an Overview of the JPEG and MPEG compression systems*, McGraw-Hill, New York, 1998.

[21] J. Weber and J. Malik, "Robust computation of optical flow in a multi-scale differential framework," in *Proc. ICCV1993*, 1993, pp. 12–20.

[22] A. Bruhn, J. Weickert, C. Feddern, T. Kohlberger, and C. Schnorr, "Variational optical flow computation in real time," *IEEE Transactions on Image Processing*, vol. 14, no. 5, pp. 608–615, May 2005.

[23] Z. Wei, D-J. Lee, and B.E Nelson, "Fpga-based real-time optical-flow algorithm design and implementation," *J. Multimedia*, vol. 2, no. 5, pp. 38–45, 2007.

[24] Z. Wei, D-J. Lee, and B. Nelson, "FPGA-Based real-time optical flow algorithm design and implementation," *Journal of Multimedia*, vol. 2, no. 5, pp. 38–45, Sept. 2007.

[25] S. Maya-Rueda and M. Arias-Estrada, *FPGA Processor for Real-Time Optical Flow Computation*, vol. 2778 of *Lecture Notes in Computer Science*, pp. 1103–1106, Springer, 2003.

[26] M. Durkovic, M. Zwick, F. Obermeier, and K. Diepold, "Performance of optical flow techniques on graphics hardware," in *Proc. ICME2006*, 2006, pp. 241–244.

[27] H. Grossauer and P. Thoman, *GPU-Based Multigrid: Real-Time Performance in High Resolution Nonlinear Image Processing*, vol. 5008 of *Lecture Notes in Computer Science*, pp. 141–150, Springer, 2008.

[28] Y. Mizukami and K. Tadamura, "Optical flow computation on compute unified device architecture," in *Proc. ICIAP2007*, 2007, pp. 179–184.

[29] M. Yeasin and R. Sharma, *Foveated Vision Sensor and Image Processing - A Review*, vol. 7 of *Studies in Computational Intelligence*, pp. 57–98, 2005.

[30] E.L. Schwartz, "Spatial mapping in the primate sensory projection: Analytic structure and relevance to perception," *Biol. Cybern.*, vol. 25, no. 4, pp. 181–194, 1977.

[31] F.L. Lim, G.A.W. West, and S. Venkatesh, "Use of log polar space for foveation and feature recognition," *IEE Proc.-Vis. Image Signal Process.*, vol. 144, no. 6, pp. 323–331, Dec. 1997.

[32] F.J. Coslado, M. Gonzalez, P. Camacho, and F. Sandoval, "Hardware platform for regions extraction in foveal images," *Proceedings of the SPIE - Visual Communications and Image Processing*, vol. 5150, no. 1, pp. 1730–1740, 2003.

[33] P. Cobos and F. Monasterio, "FPGA implementation of a log-polar algorithm for real time applications," in *DCIS*, 1999, pp. 63–68.

[34] F. Ferrari and J. Nielsen, "Space variant imaging," *Sensor Review*, vol. 15, no. 2, pp. 17–20, 1995.

[35] R. Woodnicki, G.W. Roberts, and M. Levine, "Design and evaluation of log-polar image sensor fabricated using standard 1.2 \mu m ASIC and CMOS process," *IEEE Trans. On Solid State Circuits*, vol. 32, no. 8, pp. 1274–1277, 1997.

[36] M. Yeasin, "Optical flow in Log-Mapped image plane - a new approach," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, no. 1, pp. 125–131, 2002.

[37] K. Daniilidis, C. Krauss, M. Hansen, and G. Sommer, "Real-Time tracking of moving objects with an actice camera," *Real-Time Imaging*, vol. 4, no. 1, pp. 3–20, 1998.

[38] S. Kang and S-W. Lee, *Multiple Object Tracking in Multiresolution Image Sequences*, vol. 1811 of *Lecture Notes in Computer Science*, pp. 240–255, Springer, 2000.

[39] S.S. Beauchemin and J.L. Barron, "The computation of optical flow," *ACM Computing Surveys*, vol. 27, no. 3, pp. 433–467, 1995.

[40] R. Mahony, P. Corke, and T. Hamel, "Dynamic Image-Based visual servo control using centroid and optic flow features," *Journal of Dynamic Systems, Measurement, and Control*, vol. 130, pp. 011005, 2008.

[41] S. Hrabar, S. Sukhatme, P. Corke, K. Usher, and J. Roberts, "Combined Optic-Flow and Stereo-Based navigation of urban canyons for a UAV," in *Proc. IRS*, 2005, pp. 3309 – 3316.

[42] B.A. Wandell, *Foundations of Vision*, Sinauer Associates Inc, Sunderland, MA, 1995.

[43] L.K. Cormack, *Computational models of early human vision*, Academic Press, New, 2000.

[44] C. Bandera and P. Scott, "Foveal machine vision systems," in *Proc. SMC1989*, 1989, pp. 596–599.

[45] D. Kanter, "Nvidia's gt200: Inside a parallel processor," Real World Technologies.

[46] H. Kollnig and H.-H. Nagel, "3D pose estimation by directly matching polyhedral models to gray value gradients," *International Journal of Computer Vision*, vol. 23, no. 3, pp. 283–302, 1997.

[47] L. Sciavicco and B. Siciliano, *Modelling and Control of Robot Manipulators*, Springer, New York, NY, 2000.

[48] S.B. Niku, *Introduction to Robotics: Analysis, Systems, Applications*, Pearson, Upper Saddle River, NJ, 2001.

[49] S. Hutchinson, G.D. Hager, and P.I. Corke, "A tutorial on visual servo control," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 5, pp. 651–670, 1996.

[50] F. Chaumette and S. Hutchinson, "Visual servo control part i: Basic approaches," *IEEE Robotics and Automation Magazine*, vol. 13, no. 4, pp. 82–90, 2006.

[51] F. Chaumette and S. Hutchinson, "Visual servo control part ii: Advanced approaches," *IEEE Robotics and Automation Magazine*, vol. 14, no. 1, pp. 109–118, 2007.

[52] Quanser Consulting Inc., "Quanser / crs model 255 open architecture robot," Tech. Rep., Quanser Consulting Inc., 2007.