2009-03-12

An Optical Flow Implementation Comparison Study

John M. Bodily
Brigham Young University - Provo

AN OPTICAL FLOW IMPLEMENTATION COMPARISON STUDY

by

John M. Bodily

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Electrical and Computer Engineering

Brigham Young University

April 2009

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

John M. Bodily

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	Brent E. Nelson, Chair
Date	Brad L. Hutchings
Date	Dah-Jye Lee

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of John M. Bodily in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

Brent E. Nelson
Chair, Graduate Committee

Accepted for the Department

Michael J. Wirthlin
Graduate Coordinator

Accepted for the College

Alan R. Parkinson
Dean, Ira A. Fulton College of
Engineering and Technology

ABSTRACT

AN OPTICAL FLOW IMPLEMENTATION COMPARISON STUDY

John M. Bodily

Department of Electrical and Computer Engineering

Master of Science

Optical flow is the apparent motion of brightness patterns within an image scene. Algorithms used to calculate the optical flow for a sequence of images are useful in a variety of applications, including motion detection and obstacle avoidance. Typical optical flow algorithms are computationally intense and run slowly when implemented in software, which is problematic since many potential applications of the algorithm require real-time calculation in order to be useful. To increase performance of the calculation, optical flow has recently been implemented on FPGA and GPU platforms. These devices are able to process optical flow in real-time, but are generally less accurate than software solutions.

For this thesis, two different optical flow algorithms have been implemented to run on a GPU using NVIDIA's CUDA SDK. Previous FPGA implementations of the algorithms exist and are used to make a comparison between the FPGA and GPU devices for the optical flow calculation. The first algorithm calculates optical flow using 3D gradient tensors and is able to process 640x480 images at about 238 frames per second with an average angular error of 12.1° when run on a GeForce 8800 GTX GPU. The second algorithm uses increased smoothing and a ridge regression

calculation to produce a more accurate result. It reduces the average angular error by about 2.3×, but the additional computational complexity of the algorithm also reduces the frame rate by about 1.5×. Overall, the GPU outperforms the FPGA in frame rate and accuracy, but requires much more power and is not as flexible. The most significant advantage of the GPU is the reduced design time and effort needed to implement the algorithms, with the FPGA designs requiring 10× to 12× the effort.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Nelson, and my committee members, Dr. Hutchings and Dr. Lee. A couple of fellow students also contributed to this work and deserve my thanks: Zhaoyi Wei (Roger) designed the optical flow algorithms, and Jeff Chase made improvements to my original naive GPU implementation. Finally, thanks to my loving wife, Brittany, and daughter, Abigail, for their patience, love, and support.

Table of Contents

List of Tables

List of Figures

Chapter 1

Motivation and Related Work

Computer vision is a field of research concerned with obtaining information from digital image data and constructing a system that can use that information for a variety of purposes. Applications of computer vision are useful in many different sectors of the industry. Computer vision systems may be used to control industrial robots in an assembly line at a manufacturing plant, for example, or to guide an autonomous military vehicle. Other applications of computer vision technology include visual surveillance systems and computer-human interaction.

A computer vision system typically provides several important functions. Image information may take the form of a video sequence or of images captured from one or multiple cameras. It must first be acquired by the system and preprocessed, which may include tasks such as the normalization of the data or the removal of image distortion. Once images have been preprocessed, important features of the data are determined and extracted. A detection step determines whether the extracted features contain specific conditions, then high-level processing is done on the data.

The field of computer vision research is wide and includes many interesting and varied sub-domains. Object recognition, machine learning, event detection, scene reconstruction, and motion estimation are all examples of computer vision sub-domains. A computational technique used in many of these areas of research, called the optical flow calculation, is the focus of the research presented in this thesis.

1.1 Optical Flow and Motion Fields

Optical flow can be defined as the apparent motion of patterns of brightness within an image scene [1]. It is generally caused by relative motion between objects

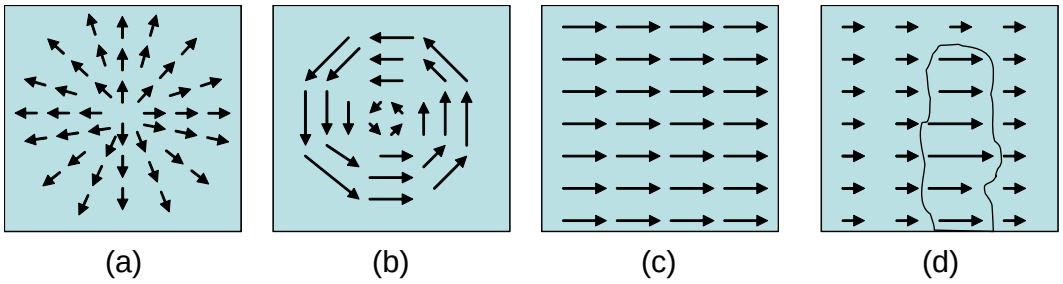

Figure 1.1: Example motion fields: (a) forward motion towards the center of the image (b) rotation (c) horizontal translation (d) an object within an image, outlined for emphasis. Objects that are closer to the camera will appear to move faster and will thus have motion vectors with larger magnitudes.

in the scene and the viewer of the scene, and can be calculated using a number of different techniques. Once the optical flow has been measured, it is frequently represented with velocity vector fields. Figure 1.1, modeled after a figure found in [2], shows some simple motion fields for common motion patterns such as forward motion, rotation, and horizontal translation.

The correctness of most optical flow algorithms is based on an assumption called the brightness constancy constraint, which maintains that changes in image brightness are due only to motion. The brightness constancy constraint is formally defined as

$$\frac{\partial E}{\partial x}\frac{dx}{dt} + \frac{\partial E}{\partial y}\frac{dy}{dt} + \frac{\partial E}{\partial t} = 0 \tag{1.1}$$

where $E(x, y, t)$ denotes the brightness of an image at point (x, y) and time t [1], and each term of the equation is a partial derivative of the image brightness with respect to x, y, and t, respectively. Effects other than motion that might cause changes in brightness, such as a change in the lighting conditions of the scene captured in the image, can be ignored if the interval between image frames is small enough. Because of this, a fast frame rate (about 15 frames per second or greater, depending on the camera used and the velocity of any moving objects) is essential for the real-time use of the calculation.

1.1.1 Calculating Optical Flow

There are many different methods for calculating an optical flow field. [3] discusses several of these methods, and suggests that most methods of calculating optical flow can be broken into three general stages of processing. First, the image is prefiltered to extract signals of interest and to reduce or smooth image noise. Second, basic image structures such as image derivatives are extracted from the image data. Finally, these structures are integrated using regularization, correlation, or a least squares computation [3].

The research presented in this thesis focuses on differential methods of computing optical flow. Differential-based algorithms estimate optical flow using the spatial and temporal derivatives of an image. The algorithms calculate the flow for every pixel in the image, which can be computationally expensive. For a 640x480 image, for example, a single 1D convolution would require at least $640 \times 480 \times (2r + 1)$ multiplication operations, where r is the radius of the convolution operator used. For an image with n pixels, this translates to $n \times (2r + 1)$ operations. A 2D convolution on the same image would require around $n \times (2r + 1)^2$ operations. Typically, differential algorithms require multiple stages of these image convolutions on multiple images within a sequence.

In 1981, two differential-based optical flow algorithms were proposed that are now considered classics in the field: one by Horn and Schunck [1], and the other by Lucas and Kanade [4]. The Horn-Schunck method uses an iterative approach to calculate the velocity field of the image sequence. A pair of equations is derived for each point in the image using the image brightness equation, $E(x, y, t)$, and the value of the flow velocity, (u, v). These equations relate the change in image brightness to the motion of the brightness pattern. While the equations can be solved simultaneously using a method such as Gauss-Jordan elimination, such an approach is quite costly. Instead, the equations are solved iteratively using a method such as the Gauss-Seidel technique. The accuracy of the Horn-Schunck method is largely dependent on the number of iterations it passes through.

The Lucas-Kanade approach, on the other hand, is non-iterative, and uses a two-frame differential method to estimate the optical flow velocity field. By assuming that the optical flow is constant within a small spatiotemporal window of an image sequence, an equation relating brightness and flow can be found for each pixel in the window. The set of equations for all the pixels within the window is then solved using the least squares method to estimate the optical flow velocity field.

Many different optical flow algorithms have been developed since these classics were introduced in 1981, including extensions and modifications of the Horn-Schunck and Lucas-Kanade approaches. One technique, the use of 3D tensors, has proven superior in producing dense and accurate optical flow fields [5][6][7][8][9]. On a basic level, 3D tensors are a powerful and compact way to store information about the orientation of structures within an image sequence. In [6], for example, the 3D tensor takes the form of a 3×3 matrix from which image velocity information can be estimated. The algorithms implemented for this thesis and discussed in following chapters use 3D tensor techniques to obtain image velocity.

Ideally, an optical flow algorithm would be able to project a perfect representation of the 3D velocity field seen by the camera into a two-dimensional image frame. In practice, this is very difficult to achieve, for several reasons. In [1], Horn and Schunck give the example of an unmarked sphere rotating in place. If the lighting in the scene is constant, the rotation of the sphere does not cause any changes in the intensity of the image. In other words, motion does not always result in a change in image brightness.

Optical flow calculations are also limited by the well-known aperture problem. The basic idea of the aperture problem is that different physical motions are sometimes indistinguishable to a motion sensor since its view of the world is finite and limited. Consider the example illustrated in Figure 1.2. If the lines in each rectangle are moving in the directions indicated by the arrows, there is no way to distinguish between the two movement patterns based only on what can be seen within the circle. Because of this, different patterns of motion can result in the exact same changes in image intensity.

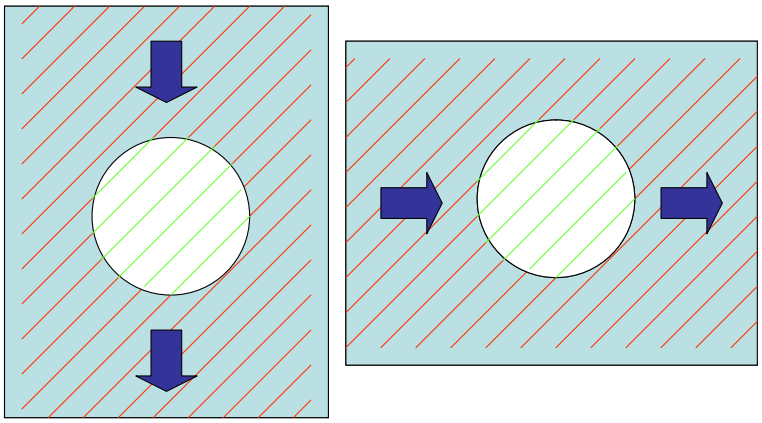

Figure 1.2: An illustration of the aperture problem. The lines in each image are moving in the direction indicated by the arrows. From the point of view of the circle (or aperture) there is no distinguishable difference between these movement patterns.

1.1.2 Uses of Optical Flow

Despite the limitations of the optical flow calculation, it is quite useful in a number of applications. For example, optical flow can give important information about the arrangement of objects within a scene, and this information can be used to reconstruct the three dimensional environment of an image. In addition, since changes in the motion of an object are reflected in the optical flow, it can be used in motion detection algorithms. Optical flow can also be used in object segmentation to break an image up into multiple segments, thus facilitating the extraction of image features such as curves and edges.

One use of optical flow that has recently gained in importance is obstacle avoidance. In 2001 the U.S. Congress mandated that by 2015 one third of military vehicles should be unmanned, and two thirds should be unmanned by 2025. This focus on unmanned vehicles has increased the need for algorithms that can help such vehicles avoid potential obstacles, especially in the case where these vehicles are both unmanned and autonomous. Such vehicles can use optical flow to determine the position and movement of potential obstacles.

Figure 1.3: A few images from the Yosemite sequence, a popular synthetic sequence used to test optical flow performance. Motion in the sequence is towards the horizon.

1.1.3 Measuring Optical Flow

In order to determine the performance and accuracy of an optical flow algorithm, a handful of synthetic image sequences have been created. Perhaps the most well-known of the synthetic image sequences is the Yosemite sequence, first described in [10]. Figure 1.3 shows a few images from this sequence.

Synthetic image sequences are useful because they have a known correct motion vector for each pixel. The output of an algorithm can be compared to this known velocity to determine how well the algorithm performs. One common comparison technique is to measure the angle between each calculated velocity vector and the corresponding ground truth vector and average the differences to find an average angular error. Formally, this angular error can be defined as

$$\psi_E = arccos(\vec{v}_c \cdot \vec{v}_e) \tag{1.2}$$

where $\vec{v}_c$ is the correct velocity and $\vec{v}_e$ is the estimate [11].

The performance of an optical flow algorithm is commonly reported in frames per second. A large number of papers report performance numbers using the Yosemite sequence, allowing for direct comparison of the performance of various techniques. The optical flow algorithms described in this thesis have also been tested using the Yosemite image sequence and report accuracy discrepancies in average angular error.

1.2 Related Work

There is a large body of research on various optical flow algorithms and implementations, including some very recent publications. While many software implementations of optical flow exist, the high computational costs of the optical flow calculation make it difficult for software versions to meet timing requirements needed in embedded and other real-time applications. A number of FPGA-based optical flow solutions have been proposed in the past several years, and GPU implementations have also recently become more common. In this section, a handful of publications related to the research presented in this thesis are discussed.

1.2.1 FPGA Implementations of Optical Flow

Recent FPGA-based optical flow solutions achieve a higher processing speed than software-based solutions, but also produce less accurate results. When processing the Yosemite image sequence, a software-based solution described in [5] produces a motion field with an average angular error of about 1°, but takes around a minute to do so for a single image frame. FPGA designs proposed in [12] and [13], on the other hand, produce results with average angular errors of 18.30° and 12.9°, respectively, but run in real-time: the first processes 320x240 images at 30 frames per second, and the second processes 640x480 images at 64 fps. Average angular errors of over 10° are typical for FPGA-based designs.

Currently, the most accurate FPGA implementation is proposed in [14], a paper published at the International Conference on Pattern Recognition in December 2008. The solution described in the paper achieves an average angular accuracy of about 6.8°, and runs at 60 frames per second for a 320×240 sequence of images.

The FPGA implementations described in both [13] and [14] were developed at Brigham Young University. The solutions proposed in these papers are the basis for the GPU implementations produced for this thesis, and are discussed in more detail in the following chapters.

1.2.2 GPU Implementations of Optical Flow

Recent research has shown that graphics processors (GPUs), like FPGAs, are a good platform for the optical flow calculation. The high computational demands of optical flow and the high computational performance of GPUs fit together well and allow for a faster performance of the algorithm than can be achieved with software alone.

Since GPUs are highly parallel SIMD devices, optical flow algorithms with high levels of parallelism can achieve better performance on graphics hardware than iterative algorithms. The authors of [15] made a comparison between the performance of the classic Horn-Schunck and Lucas-Kanade optical flow algorithms on graphics hardware. They found that the Lucas-Kanade algorithm fit much better into the GPU hardware than the Horn-Schunck approach: "Lucas and Kanade's algorithm allows a highly parallel processing using the [GPU]'s parallel pixel shaders, while Horn and Schunck's algorithm is an iterative one and therefore not a good candidate for parallelizing."

Despite these findings, available literature on GPU implementations of optical flow suggests that the Horn-Schunck algorithm is currently a popular approach. In [16], a solution is proposed that is based on the Horn and Schunck algorithm, with some modifications made to the algorithm to make it more robust against noise and illumination changes in the image sequence. The algorithm is implemented on an NVIDIA GeForce 7800 GS and a GeForce Go 7900 GTX GPUs using the Cg language, a C-based language developed by NVIDIA especially for GPU programming. It is able to process the Yosemite image sequence at about 27.2 fps with an angular error of about 3°. In the future work section of the paper, the authors indicate a plan to implement the same algorithm using the CUDA SDK. (CUDA, an extension to the C programming language, is discussed in more detail in the next chapter.)

Another Horn-Schunck implementation is described in [17]. This time, the algorithm is implemented using CUDA and a GeForce 8800 GTX GPU. The GPU performance results are compared favorably to a CPU implementation of the same algorithm. The error of the implementation is computed using a root mean square

method and is reported in pixels rather than in degrees of angular error. The paper also reports a time of 2.806 seconds for 3000 iterations of the algorithm, but it is unclear from the paper how many of these iterations are necessary for the correct performance of the implementation.

It is also likely worth mentioning that a small number of additional GPU implementations of optical flow can currently be found on NVIDIA's web page at `http://www.NVIDIA.com/object/cuda_home.html`, but these implementations are not accompanied by any published research.

1.2.3 Previous Publication of This Research

Research related to this thesis has previously been published in the *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines* [18] and a forthcoming ACM Transactions paper.

1.3 Contributions of This Research

While optical flow has previously been implemented on both FPGA and GPU platforms, none of the papers discussed in the previous section makes a comparison between the two architectures. In this thesis, the first GPU implementations of two different optical flow algorithms are described and then compared against FPGA implementations of the same algorithms.

In the future, it is likely that parallel programming techniques will only grow in importance with the onset of multi- and many-core technology. Some of the programming ideas and practices discussed in this thesis will very likely be useful and relevant to a large number of applications.

Chapter 2

GPU Programming

For this thesis, a pair of optical flow algorithms previously implemented on FPGA platforms have been implemented to run on a GPU. This chapter discusses the use of GPUs for general purpose applications, like optical flow, as well as the programming environment and specifications of the GPU devices used for the GPU implementations of optical flow.

2.1 GPGPU

In recent years, the performance of uniprocessor computers has plateaued. Clock speeds are no longer increasing, and available instruction-level parallelism has already largely been exploited. Now the trend in the industry is towards creating multi-core and many-core technologies that take advantage of data- and thread-level parallelism within applications to improve performance.

Graphics processing units (GPUs) are an example of many-core machines. Although GPUs were originally developed for the gaming industry to enable real-time, high-definition 3D graphics, they have proven to be effective machines for a variety of non-graphics applications as well. Using GPUs for non-graphics applications is sometimes called general purpose programming on a GPU or GPGPU.

Currently, nearly the entire market of high-end GPUs is controlled by just two companies: NVIDIA and ATI. (Intel has plans to release a GPU chip codenamed Larrabee in the near future, and so the market will likely soon change.) Both NVIDIA and ATI have made efforts in the last couple of years to facilitate general purpose programming on their hardware, but NVIDIA, with its C-based CUDA software API, has been much more successful. The research done for this thesis used NVIDIA GPUs

and the CUDA API. As such, the discussion in this chapter focuses on NVIDIA's technology.

2.1.1 CUDA: A Software Environment for Parallel Programming

Historically, programming on a GPU required the use of a graphics API such as OpenGL. In general, graphics languages are difficult to learn and require a high degree of skill to program effectively. Using such languages for non-graphics applications requires a thorough understanding of the graphics pipeline and can be especially challenging. In 2007, NVIDIA released a new line of GPUs with a more generalized computing architecture and a new API that simplifies GPGPU programming. The new API, called *CUDA*, an acronym for Compute Unified Device Architecture, extends the C programming language by allowing a programmer to define C functions that are executed by a large number of threads in parallel on the compatible GPU hardware.

While CUDA does not magically eliminate the complexity of GPGPU or parallel programming in general, it does present a more approachable interface to the programmer. It is based on a language with which most programmers are familiar and allows the GPU to be seen as a set of parallel processing elements. A summary of CUDA and a few of the basics of CUDA programming are included here as a foundation to explain some of the research that follows. Further information about CUDA programming can be found in NVIDIA's CUDA Programming Guide 2.0 [19].

2.1.2 Kernels, Blocks, and Threads

A CUDA program is broken up into sections that run on the host machine and sections that run on the GPU. The sections that run on the GPU are called *kernels*. A kernel's computation is subdivided into a parameterizable number of blocks, each of which executes entirely within a single multiprocessor and independently from other blocks in the kernel. No communication or synchronization between blocks is possible, since blocks from the same kernel may be running on different multiprocessors. A group of blocks is called a *grid*.

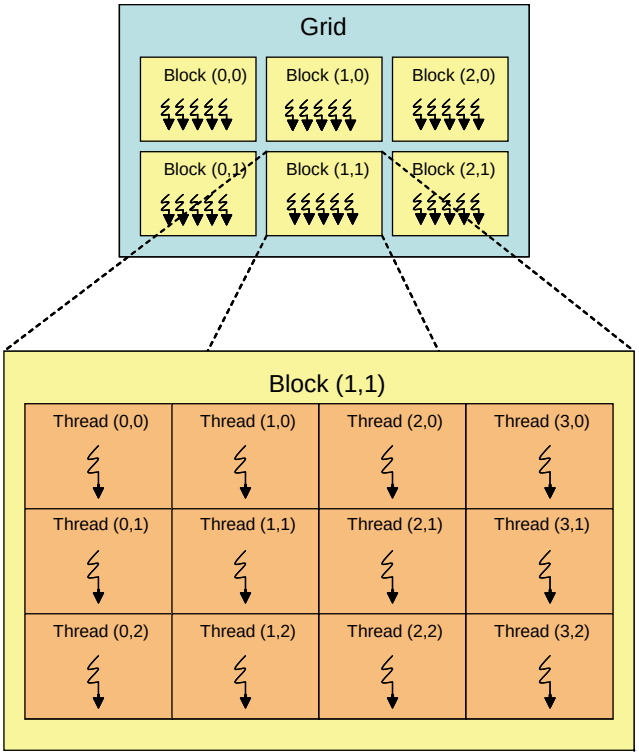

Figure 2.1: Threads are grouped into blocks and blocks are grouped into grids.

Blocks are further subdivided into lightweight computational threads. Groups of these threads, called *warps*, are executed in a multiprocessor's SIMD elements. Figure 2.1, modeled after a figure in the CUDA Programming Guide, illustrates the relationship between threads, blocks, and grids. The choice of grid and block arrangement used in a CUDA program is left to the programmer. Each thread can determine, at run time, its block and thread index and use this information to determine which subset of the data to operate on. Threads within a single block communicate with each other via shared memory (described below) and with thread synchronization calls.

2.1.3 CUDA Memory Heirarchy

CUDA programs have multiple memory spaces from which threads can access data during execution. Figure 2.2, modeled after a figure in the CUDA Programming Guide, shows the different types of memory available to a thread. The GPU platform's

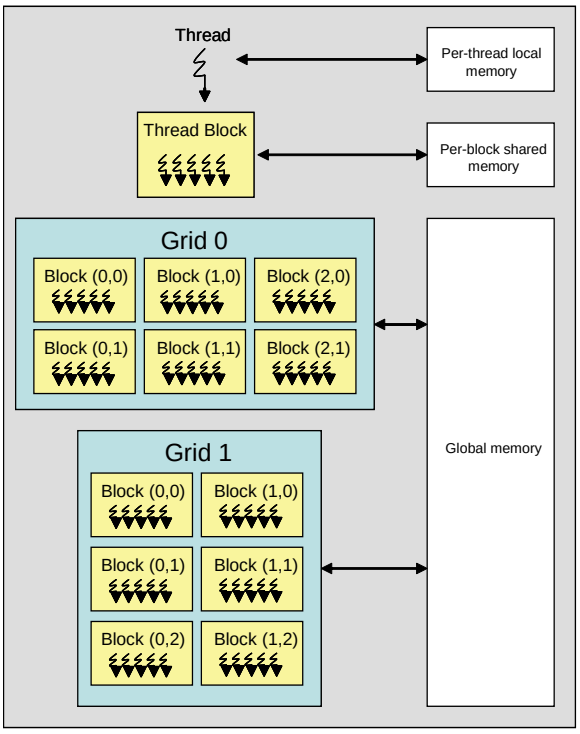

Figure 2.2: The GPU memory hierarchy is made up of multiple memory spaces.

global memory (called device memory) is DRAM, and is not cached. It is large (100's of MB for current GPUs) but also very slow, with access times measured in multiple 100's of clock cycles. Device memory is optimized for block transfers; when concurrent sequentially-addressed memory accesses to global memory are *coalesced* by the hardware into larger block transfers, performance can be greatly improved.

Each multiprocessor also contains its own on-chip local store, called shared memory. Shared memory is small (10's of KB on current GPUs) but can provide single-cycle access. On current GPUs, shared memory is optimized for concurrent memory accesses, and can satisfy up to 16 concurrent accesses per cycle provided no bank conflicts occur.

The GPU also provides two types of read-only memory not pictured in Figure 2.2 called constant and texture memory. These memory spaces are similar to global memory in that they can be accessed by all threads, but they differ in that

they are optimized for different types of memory usage. Both constant and texture memory are cached and can result in low-latency memory access if used properly.

In general, management of the GPU memory hierarchy is manual and left to the programmer. Coding an application properly to enable aligned memory accesses and device memory coalescing and to avoid shared memory bank conflicts is a critical part of achieving good performance on a GPU. Different generations of CUDA-compatible GPUs require slightly different techniques to ensure memory coalescing, but all benefit from the technique.

2.2 NVIDIA GPUs

NVIDIA has a wide variety of GPUs currently on the market. Three different CUDA-compatible NVIDIA GPUs were used for the research done in this study: the GeForce 8800 GTX, the GeForce GTX 280, and the GeForce 8400 GS. Table 2.1 summarizes the specifications of these GPUs. These specifications are available on NVIDIA's website[1].

The GPU designs discussed in the following chapters were first implemented to run on an NVIDIA GeForce 8800 GTX platform. The host for this GPU was an Intel Xeon 1.86 GHz machine with 1 GB RAM. The 8800 GTX contains 128 processing elements, arranged into 16 multiprocessors with 8 SIMD elements each. It has 768 MB of device memory and a memory bandwidth of around 86 GB/sec. The 8800's core clock runs at 575 MHz, and its stream processors run at a little more than twice this frequency. It was built using a 90 nm process.

The GeForce GTX 280, first released in June of 2008, is built on a 65 nm process. The number of processing elements on the GTX 280 has been increased to 240, arranged into 30 multiprocessors. The host machine for this GPU was an Intel 2.66 GHz quad-core processor with an EVGA nForce 790i SLI FTW motherboard and DDR3 SDRAM. The GTX 280 has 1024 MB of device memory and a maximum memory bandwidth of 141.7 GB/sec.

[1] `http://www.NVIDIA.com/object/geforce_family.html`

Table 2.1: A summary of the specifications for the NVIDIA GeForce GPUs used in this study.

	8400 GS	**8800 GTX**	**GTX 280**
Core Clock Rate (MHz)	450	575	602
Stream Processor Clock Rate (MHz)	900	1350	1296
Stream Processors	16	128	240
Device Memory (MB)	256	768	1024
Memory Bandwidth (GB/s)	6.4	86.4	141.7

The 8400 GS was released in mid-2007 and is targeted towards lighter-weight graphics applications. It has only two multiprocessors with a total of 16 processing elements. It uses 256 MB of DDR2 device memory and has a maximum memory bandwidth of 6.4 GB. The processors in the 8400 GS run at 900 MHz, and the core clock frequency is 450 MHz. The 8400 GS was hosted on an HP dc7800 3.0 GHz Core 2 Duo machine.

2.3 Keys to Obtaining Good Performance on the GPU

While CUDA has simplified the GPU programming approach, obtaining good performance using the API can still be challenging. In this section, several keys to obtaining good performance on a GPU are discussed. The summary of GPU programming techniques provided here draws on ideas from [20], as well as from personal experience with the hardware.

2.3.1 Choose the Right Algorithm

Since GPUs are highly parallel machines, they achieve the best possible performance on algorithms that can be highly parallelized. An algorithm that can be split up into thousands of parallel computations will likely achieve a much better speedup using CUDA than an algorithm with several sequential steps. In [18], for example, an implementation of a digital MIMO communication system is discussed. One part of the MIMO system required the evaluation of an error surface and locating a minimum point on that surface. In the original FPGA implementation of the system, the

minimum point on the surface was found using a sequential search algorithm. When the system was ported to run on a GPU, the sequential nature of the search algorithm limited the performance of the system on the GPU. A different, highly parallel search algorithm was used on the GPU in order to meet performance requirements.

2.3.2 Achieve High Processor Occupancy

A GPU kernel is executed most efficiently when all of the multiprocessors are kept occupied. To keep each multiprocessor occupied, the total number of blocks should be a multiple of the number of multiprocessors so that each processor does the same amount of work. In CUDA, obtaining a high occupancy requires that a programmer keep the usage of resources shared between threads, such as local registers or shared memory, to a minimum.

2.3.3 Reduce Number of Divergent Threads

All threads within a GPU thread warp should execute the same code whenever possible. When threads within the same warp are scheduled to execute different portions of the code, the entire warp must execute all code paths sequentially until the threads converge again. A GPU programmer should seek to structure code to ensure that threads diverge as little as possible.

2.3.4 Use Memory Efficiently

Since accesses to the GPU's global device memory have such a long latency, programs that have a high number of numeric computations per memory access perform better on GPUs than programs that frequently access memory. To improve the use of memory in the GPU, and thus the compute-I/O ratio, it is important to ensure that memory accesses are *coalesced*: when a group of threads accesses concurrent sequentially-addressed memory, the hardware transfers the memory in one large, coherent block. Similarly, placing data that will be accessed by multiple threads into the GPU's shared memory can reduce the number of accesses to global memory and improve the performance of a program.

Chapter 3

3D Gradient Tensor Algorithm

A fast and accurate 3D tensor-based optical flow algorithm was developed at Brigham Young University by Zhaoyi Wei. The algorithm was first implemented by Wei on a low-power FPGA suitable for use on small, unmanned vehicles, as described in [13]. As part of the research done for this thesis, the tensor algorithm was then modified to run on NVIDIA's CUDA-compatible GPUs. In this chapter, the details of the algorithm and its GPU implementation are discussed.

3.1 Tensor Algorithm Details

The tensor-based optical flow algorithm treats a sequence of images as a volume of data, with x and y representing the spatial components of the data and t the temporal component. Movement of an object within this spatial-temporal domain causes changes in the brightness pattern of the image sequence. The translation of a single point within the image sequence, for example, forms a line within the volume. The orientation of this line directly corresponds to the velocity of the moving point [6].

Information about the orientation of the brightness pattern within the volume of data can be efficiently stored in 3D orientation tensors. There are several very different methods for deriving these tensors. Quadrature filters are used in [21], for example, and [22] discusses a tensor based on a polynomial expansion model. The algorithm discussed in this chapter uses a tensor derived from the image gradient. The gradient tensor approach was chosen by Wei for the original FPGA implementation because it is similar to the original Lucas-Kanade algorithm and is a good fit for pipelined hardware. Additionally, for simple signals such as those that occur in

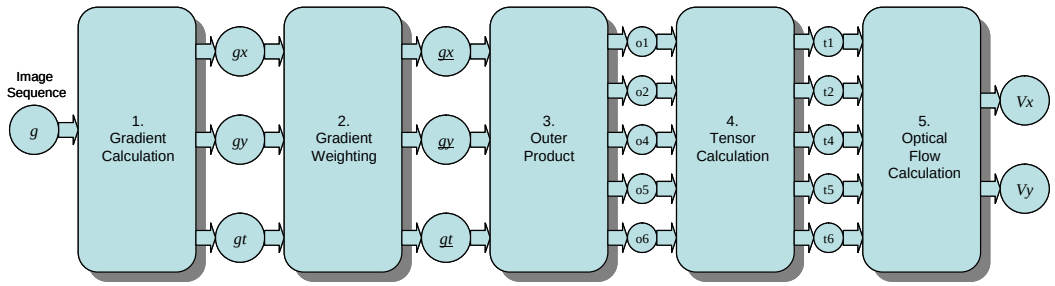

Figure 3.1: The gradient tensor algorithm takes a sequence of images as input, processes it in five computational steps, and outputs two velocity vector fields.

optical flow, the gradient tensor algorithm gives velocity estimates similar to the more complex polynomial tensor [23].

The gradient tensor-based optical flow algorithm is a five-stage computation as shown in Figure 3.1. The first stage performs one-dimensional convolutions in the x, y, and t domains using five-element derivative operators. In order to reduce errors in this first stage of the calculation, a first-order derivative operator with a large radius is desired. An operator with a large radius allows for increased smoothing and better derivative estimates. After considering various performance tradeoffs, the operator $D_1 = \frac{1}{12}(1\ -8\ 0\ 8\ -1)$ was chosen. Convolving the image data with this operator results in gradient images gx, gy, and gt.

The second stage (Gradient Weighting in Figure 3.1) performs seven-element row and column convolutions in succession on each of gx, gy, and gt to produce weighted gradient images $\underline{gx}$, $\underline{gy}$, and $\underline{gt}$. The convolution operator used in this stage of the calculation is $D_2 = (3\ 5\ 7\ 8\ 7\ 5\ 3)$. The weights used in both this and the fourth stage of the computation were selected based on the Gaussian function to give the best tradeoff of performance and efficiency. Generally, weights determined using Gaussian functions are preferred since they emphasize the center pixel in the convolution window.

The third processing stage (Outer Product in Figure 3.1) produces five new matrices $(o1, o2, o4, o5, o6)$ by multiplying various combinations of the weighted gradient matrices together element-wise ($o1 = \underline{gx} \times \underline{gx}$, $o2 = \underline{gy} \times \underline{gy}$, $o4 = \underline{gx} \times \underline{gy}$,

$o5 = \underline{gx} \times \underline{gt}$, and $o6 = \underline{gy} \times \underline{gt}$). A sixth unique matrix, $o3$, could be produced by multiplying $\underline{gt} \times \underline{gt}$, but is unnecessary since it would not contribute to the final velocity calculation.

The fourth stage of the algorithm (Tensor Calculation in Figure 3.1) is another weighting process. It performs a three-element row convolution followed by a three-element column convolution using the operator $D_3 = (1\ 1\ 1)$ on each of the outer product matrices. This calculation produces five tensor matrices $(t1, t2, t4, t5, t6)$.

The final stage (Optical Flow Calculation in Figure 3.1) computes the x and y velocity flow fields V_x and V_y using the equations:

$$V_x = \frac{(t_6 t_4 - t_5 t_2)}{(t_1 t_2 - {t_4}^2)} \tag{3.1}$$

and

$$V_y = \frac{(t_5 t_4 - t_6 t_1)}{(t_1 t_2 - {t_4}^2)}. \tag{3.2}$$

3.2 Two GPU Implementations of the Tensor Algorithm

Two different GPU implementations of the tensor-based optical flow calculation were created. The first is an example of a naive approach to programming on the GPU. The second is modeled, in part, after an NVIDIA separable convolution kernel available as a sample program in the CUDA SDK, and is more optimized for the GPU hardware.

3.2.1 Naive GPU Implementation

The first implementation was created without any previous experience with the hardware or the optical flow algorithm. It is a good example of how a programmer new to GPU hardware and parallel programming might approach CUDA programming.

As discussed in the first section of this chapter, the tensor-based optical flow algorithm has five stages: gradient calculation, gradient weighting, outer product, tensor calculation, and optical flow calculation. Originally, the naive implementation decomposed the calculation in what is perhaps the most intuitive method of splitting

it up: by creating a GPU kernel for each stage of the optical flow algorithm. The second and fifth steps of the calculation were later split into two kernels each, for a total of seven kernels. Details of each of these GPU kernels are discussed in this section.

Gradient Calculation Kernel

The first stage of the algorithm calculates the gradient of an image in three dimensions: x (horizontal), y (vertical), and t (over time). A sequence of five images is necessary for the gradient calculation in the time dimension. For both GPU implementations of the tensor-based algorithm, these images are read into GPU memory and the gradient is calculated for the third (middle) image. The naive implementation reads the original image sequence in as raw image data with one byte of data per pixel. At two points in the computation the size of the pixel data is increased to ensure accuracy of the results; once from one byte to two bytes, and once from two bytes to four bytes. The final velocity vector is output as a vector of floats, with four bytes of data per pixel.

Each of the three gradients is calculated in a single GPU kernel by breaking the total number of blocks running on the GPU into three parts.The first portion of blocks calculates the gradient in the x direction, the next portion in the y direction, and the final portion over time. Since the gradient is being calculated in three directions, the kernel outputs three different gradient arrays (gx, gy, and gt).

If the block number is smaller than the height of the image, then the threads in the block calculate the gradient in the x direction. One entire row of the image is read into shared memory. For the Yosemite sequence, this results in 316 bytes per pixel, since each pixel is a single byte of data. There are enough active threads in the block for each thread to compute the gradient for a single pixel. Each thread multiplies its pixel and four surrounding pixels against the mask (1 -8 0 8 -1), and then sums the five products together. (Since the mask contains a zero and a one, some computation is saved by only doing the multiplication for three of the pixels.)

After the sum is calculated, one result pixel is written back to global memory in the GPU for each thread. Two bytes of data are alloted for each result pixel.

A second chunk of blocks works on the gradient in the y direction, reading in a column of pixels into memory instead of a row. Each thread once again multiplies five of the pixels against the mask and stores the result to global memory. When processing the Yosemite image sequence, fewer threads are needed for this stage, since the height of the images (252 pixels) is less than the width.

The third chunk of blocks does a temporal gradient using pixels from five different images in the sequence. An entire row from each of the five images is read into shared memory, and there are enough active threads for each thread to process the gradient for a single pixel. For the Yosemite sequence, this results in 1580 bytes being read from global memory. Each thread multiplies the mask against a single pixel in each image. Similar to the x and y gradients, one pixel is written back to global memory for each thread in the block.

There are some problems with this implementation of the gradient calculation. For example, decomposing the gradient calculation into three chunks within a single kernel reduces the efficiency of the kernel. The number of threads in each block cannot be changed mid-kernel, so each block launches with the maximum number of threads it will need at any point in the calculation. For the gradient kernel, this means that each block launches with a number of threads equal to the width of the image if the image width is greater than the image height, and a number of threads equal to the height of the image if the opposite is true. In the case of the Yosemite sequence, each block contains 316 threads. In the calculation of the y gradient, only 252 of these threads are needed. The remaining threads are idle, and potential processing power is left unused.

Gradient Weighting Kernel

The gradient weighting kernel step of the algorithm smooths the results from the first step. It does this by once again multiplying a pixel and its neighbors with a

mask, which for this kernel is (3 5 7 8 7 5 3). The results of this multiplication are summed together and averaged.

Each of the three gradient arrays from the first stage are smoothed in both the x and the y direction. Instead of doing both directions at once, the convolution can be decomposed into two cascaded one-dimensional convolutions without changing the results. The input image is first convolved in the x direction, then the output of that computation is convolved in the y direction. Decomposing the convolution into two parts is known as *separable convolution*, and is allowable because the weights in the mask are given by the Gaussian function. For the implementation of the gradient weighting kernel on the GPU, this means that only a row or column of data needs to be available in shared memory, rather than a 2D block that would be required with a 2D convolution. It also means that the computation can be split up into two separate GPU kernels. The first kernel convolves each of the three input images in the x direction. The results of this kernel are then passed to the second weighting kernel, which applies a y convolution to the data.

Since the x and y convolutions are now in separate kernels, each kernel is more efficient in terms of processor occupancy than the gradient calculation kernel. The kernel that does the x convolution contains enough threads for each thread to calculate a single pixel in the width of the image, and the kernel that does the y convolution contains enough threads for each pixel in the height of the image.

Because the smoothing in the x direction works along the rows of the gradients and the smoothing in the y direction works along the columns, the number of calculations done and the amount of shared memory used in each is different. For the x convolution, each block uses enough shared memory to read in a row of data. For the Yosemite sequence, this requires $316 \times 2 = 632$ bytes of shared memory per block, since the input pixels are each two bytes. The y convolution requires $252 \times 2 = 504$ bytes of shared memory per block. Results of the weighting calculation are stored as floating point data, with four bytes of data per pixel.

One problem with this implementation of the gradient weighting calculation is that it computes a convolution for each of the three input images within the same

GPU kernel. Because of this, the data that is stored in shared memory has to be changed for each input image, and each time it is changed the GPU threads must be synchronized. Synchronization slows down the computation because it requires the threads to stop and wait at a point in the program until all threads have reached that point.

Outer Product Kernel

The output of the second stage is three smoothed gradient arrays ($\underline{gx}$, $\underline{gy}$, and $\underline{gt}$). The third stage of the algorithm takes these three arrays and calculates the outer product to produce five new matrices ($o1, o2, o4, o5, o6$). On the GPU, this step requires a single computational kernel, called with a number of blocks equal to the image height and a number of threads in each block equal to the image width. Each thread in the kernel reads in one pixel from each of the smoothed gradient inputs, then calculates a single pixel for each of the five different outputs ($o1 = \underline{gx} \times \underline{gx}$, $o2 = \underline{gy} \times \underline{gy}$, $o4 = \underline{gx} \times \underline{gy}$, $o5 = \underline{gx} \times \underline{gt}$, and $o6 = \underline{gy} \times \underline{gt}$).

Since each pixel of the input data is only used a single time in the computation of the outer product, the GPU kernel does not read the input data into shared memory. In cases such as this, where input data is not reused, each memory access is to the slower global memory.

One potential weakness of this naive implementation of the outer product kernel is that the computation to I/O ratio for each thread is not very high. Since accesses to global memory come with such long latencies on the GPU hardware, kernels with more computation per thread tend to achieve better performance.

Tensor Calculation Kernel

The fourth step of the optical flow algorithm is another smoothing step. The gradient tensor can be calculated by weighting the outer product matrices computed in the previous step.

On the GPU, the tensor is calculated by taking a pixel and summing it with its two nearest neighbors. Similar to the gradient weighting stage, the tensor calculation

is broken up into two different kernels on the GPU. The first kernel does smoothing in the x direction for each of the five outer product matrices, summing pixels along the rows of each matrix. The second kernel does the smoothing on these matrices in the y direction, summing pixels along the columns.

As with the gradient weighting kernels, calculation of the tensor in the x direction uses a GPU thread for each pixel in the width of the input data, and the y direction calculation has a thread for each pixel in the height of the input data. Each thread does less work in this kernel compared to threads in the gradient weighting kernel, since the mask used, (1 1 1), has a smaller radius and does not require any multiplication. Additionally, the sum of the three pixels is not averaged in this step as it is in the weighting calculation. The x tensor calculation reads a row of the outer product into shared memory ($316 \times 4 = 1264$ bytes per block for the Yosemite sequence), and the y tensor calculation reads in a column of the outer product data (1008 bytes per block for the Yosemite sequence).

The naive implementation of the tensor calculation computes the tensor for each of the five outer product matrices in the same kernel. This requires that the data stored in shared memory be replaced for each input, and that GPU threads be synchronized after each replacement of data.

Optical Flow Calculation Kernel

The final stage of the algorithm calculates each pixel's velocity in the x and y directions using the tensor matrices produced in the fourth step. As a reminder, the velocity can be found as follows:

$$v_x = \frac{(t_6 t_4 - t_5 t_2)}{(t_1 t_2 - {t_4}^2)} \tag{3.3}$$

and

$$v_y = \frac{(t_5 t_4 - t_6 t_1)}{(t_1 t_2 - {t_4}^2)}. \tag{3.4}$$

This is a fairly straightforward calculation, and is accomplished with a single kernel on the GPU. Like the outer product kernel, input data is not reused and the

kernel does not require shared memory. The kernel is called with the number of blocks equal to the image height, and with the number of threads in each block equal to the image width. Each thread reads in five pixel values, one from each tensor matrix, then calculates a single pixel of Vx and Vy as indicated in Equations 3.3 and 3.4. The calculation requires six multiplications, three subtractions, and one division. The GPU kernel must check the divisor $(t_1 t_2 - {t_4}^2)$ with a conditional statement to ensure that it is not zero. This conditional statement can result in sequential execution of a thread if one thread in a warp has a divisor equal to zero and the rest do not.

Weaknesses of the Naive Implementation

Besides the specific problems with each kernel discussed above, such as processor occupancy and low compute to I/O ratio, the naive GPU implementation has a few general weaknesses that apply to all of the kernels. First of all, the implementation is not very scalable. The GPU hardware limits the maximum number of threads per computational kernel. Since each of the kernels discussed in this section runs with one thread for every pixel in either the width or height of the image, the program will not work if either the width or height is greater than the maximum number of threads. For the 8800 GTX, the maximum allowable number of threads in each block is 512.

In addition to scalability problems, this implementation ignores several important properties of the GPU hardware. First, threads are run together in groups called warps. The number of threads in a warp is the minimum number of threads that should ever be executing in parallel. If the total number of threads used is not a multiple of the warp size, then at least one of the thread warps will have inactive threads. NVIDIA refers to this as the *occupancy* of a kernel, and has a calculator available to determine the portion of the hardware that is actively used. The naive GPU implementation ignores this property by setting the number of threads in a block equal to either the height or width of an image. If these values are not multiples of the warp size, the GPU hardware will not be utilized as efficiently as is possible.

Second, the GPU hardware is optimized to run computations on floating point data. The naive GPU code reads the input sequence in as character data and uses shorts to store intermediate data. Reading the image sequence in as floating point data and maintaining it that way throughout the computation can increase the performance of the algorithm.

Third, performance of a GPU kernel is highly dependent on the memory access pattern, since accesses to global memory have such a long latency. If memory accesses to global GPU memory are structured correctly in a program, large chunks of data can be transferred simultaneously. This is referred to as *coalesced* memory access. The naive GPU code described in this section does not structure memory accesses to coalesce, especially when accessing one pixel at a time from a column of the image data.

3.2.2 Tile-Based GPU Implementation

In this section, an optimized version of the GPU tensor optical flow algorithm is discussed. This version of the code overcame some of the weaknesses of the naive implementation by using tile-based convolution in the gradient calculation, gradient weighting, and tensor calculation kernels. The tile-based convolution is based on a separable convolution program released in the CUDA SDK[1]. The implementation also breaks up the outer product and velocity calculations into tiled versions to improve scalability and efficiency of computation and memory accesses required in each kernel.

Figure 3.2 is modeled after a figure in an NVIDIA white paper on separable convolution [24], and illustrates the basic idea behind how data can be broken up into tiles. Rather than read the entire image, or an entire row or column of the image, into the shared memory on the GPU, the image is read into shared memory in smaller chunks or tiles. Each block in the GPU convolution kernel processes a tile. The border seen in the figure around each tile is referred to as the *apron* of the tile, and must be as wide as the radius of the convolution operator. Breaking up the image

[1]The SDK code for the separable convolution and accompanying white paper are available at `www.NVIDIA.com/content/cudazone/cuda_sdk/CUDA_Basic_Topics.html`.

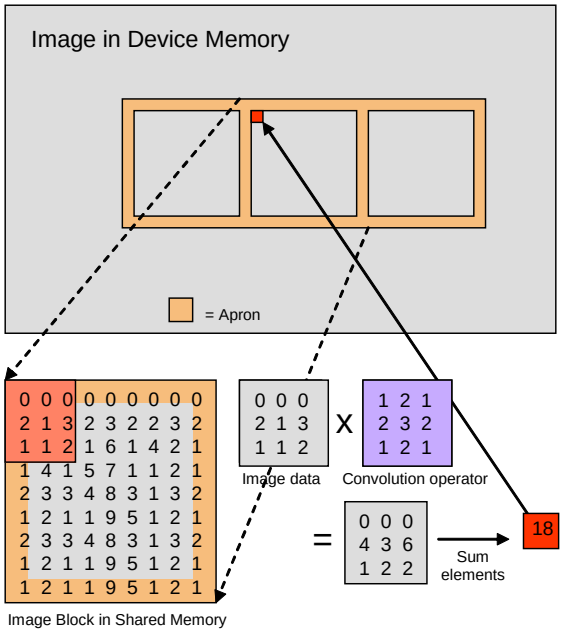

Figure 3.2: Image data can be broken up into tiles that are each processed individually on the hardware to improve performance and scalability.

into tiles allows the code to be more scalable. For larger images the number of tiles increases, but the size of each tile does not.

There are some differences between the SDK code and the convolutions in the optical flow program. The column convolution in the SDK has each thread process multiple pixels, for example, but the column convolutions in the optical flow code were modified so that each thread computes the result for a single pixel. Modifying the convolution in this manner allowed for more easily adjustable tile sizes.

Choosing appropriate tile sizes for each of the GPU kernels proved to be somewhat challenging. Having tiles that are too large limits the number of tiles that can be processed by a single multiprocessor, due to limits on the amount of shared memory and number of threads that each multiprocessor can handle. Having tiles that are too small increases the number of thread blocks required as well as the overhead needed to handle the blocks. Ensuring that the width of each tile is a multiple of the warp or half warp size ensures coalesced memory accesses.

Table 3.1: Optimal tile sizes for the 3D gradient tensor algorithm when processing 640x480 images on an 8800 GTX GPU.

Kernel	Best Tile Width	Best Tile Height
Gradient X	320	1
Gradient Y	16	20
Gradient Z	16	4
Weight X	320	1
Weight Y	16	18
Outer Product	32	2
Tensor X	320	1
Tensor Y	16	20
Flow	32	4

BYU student Jeff Chase created a program that iterated through all possible tile sizes to find the optimal configuration for each kernel in the optical flow program. The best tile sizes for each kernel of the tile-based gradient tensor implementation running on an 8800 GTX can be seen in Table 3.1.

In addition to the decomposition of input data into tiles for each of the five computational stages in the gradient tensor algorithm, the tile-based implementation differs from the naive implementation in a few other important ways. First, the gradient calculation kernel is now split into three separate kernels — one each for convolution in the x and y directions and for the convolution over time. The gradient weighting calculation still consists of two separate kernels, but instead of smoothing each of the three inputs in a single kernel, the kernel is called three separate times, once for each gradient image. Similarly, the two kernels that make up the tensor calculation are each called five separate times, once for each outer product matrix. In total, the number of GPU kernel calls increases from seven in the naive implementation to twenty-one in the tile-based version.

Another change is how data is stored throughout the calculation. In the tiled implementation, image data is stored in the floating point format, four bytes per pixel, throughout the computation. This increases the time it takes to copy image data to the GPU, but improves performance overall.

The tile-based implementation also checks the dimensions of the input image sequence to ensure that the beginning of each row is properly aligned in memory. If the width of an image is not a multiple of 64 bytes, each row in the image is padded in order to align memory accesses and reduce access latency. The Yosemite sequence, for example, has an image width of 252 pixels. Padding the images in the Yosemite sequence to 256 pixels improves performance by nearly 1.5×.

Finally, the tile-based implementation takes advantage of a function in the CUDA SDK that allocates image data in non-pageable memory. Doing so reduces memory copy times between the host and the GPU and increases processing speed another 1.6×. In all, the changes made in the tile-based GPU implementation improve performance by about 3× over the naive implementation for 320x240 images.

Chapter 4

Ridge Regression Algorithm

The gradient tensor-based algorithm results in an average angular error of about 12.9° for the FPGA implementation discussed in [13] and a similar error when implemented on the GPU. While this error compares well with other contemporary FPGA implementations, when optical flow is implemented in software it can achieve a much higher level of accuracy — an average angular error of 1° is not uncommon for software-based optical flow [5]. With a goal to improve on the accuracy of hardware-based optical flow, a modified version of the tensor-based algorithm was developed and implemented on an FPGA. This new version and the results of its implementation are presented in [14]. The new version is similar to the original tensor-based algorithm, but uses additional smoothing and a ridge regression calculation to improve the average angular error of the results. It will be referred to hereafter as the "ridge regression algorithm." A GPU version of the ridge regression algorithm was created for the purposes of this thesis and is discussed in this chapter.

4.1 Ridge Regression Algorithm Details

Optical flow algorithms generally require a large amount of smoothing to reduce the effects of image noise and to obtain accurate information about the motion within an image sequence. The ridge regression algorithm incorporates additional temporal smoothing to increase the accuracy of the solution. Ridge regression [25] is also used in the algorithm to solve the collinear problem that arises in the traditional least squares method of calculating optical flow. This collinear problem occurs when two vectors used in the calculation are nearly linearly dependent, and causes that even small levels of noise can lead to large and inaccurate motion vectors. Ridge

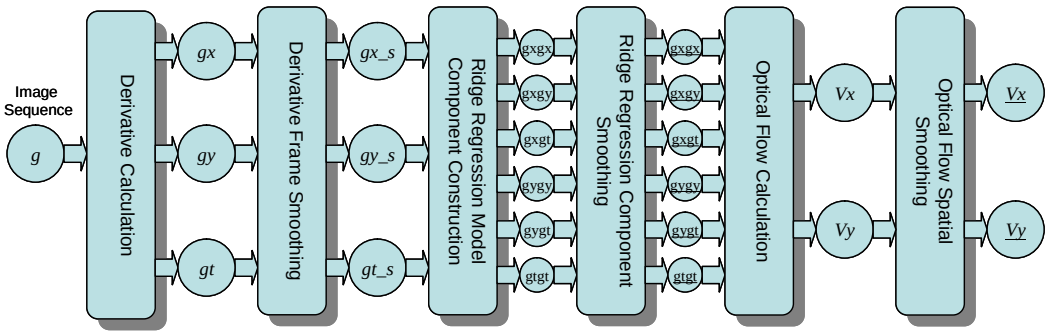

Figure 4.1: Dataflow of the ridge regression algorithm. The algorithm is more computationally intense than the tensor algorithm, but produces more accurate results.

regression reduces the collinear problem by adding small positive quantities to the diagonals of the normal equations in the calculation of the velocity.

The ridge regression algorithm is similar in many ways to the gradient tensor algorithm presented in the previous chapter. Like that algorithm, it is made up of multiple stages of processing. A diagram of the dataflow of the algorithm can be seen in Figure 4.1. The first stage is an elaborate derivative calculation in which three derivative frames (gx, gy, and gt) are calculated from raw input images using 1D convolutions. The convolution operator used in this step, (1 -8 0 8 -1), is the same as that used in the first stage of the tensor algorithm. Since the calculation of each gt frame requires two past and two future images from the sequence, this stage requires a total of five input images for each derivative calculation.

In the second stage of the computation, the derivative frames are smoothed using a 3D smoothing convolution. Three sets of derivative frames are buffered up in order to accomplish this 3D convolution. The three gx frames are smoothed together temporally using a three-element operator, (2 4 2), to create a single smoothed frame, gx_s . Likewise the three gy frames and three gt frames are temporally smoothed using the same operator to produce a smoothed gy and a smoothed gt. Five-element row and column convolutions are then applied to the three temporally-smoothed frames in a spatial smoothing step. In this step, each frame is first convolved in the

x direction using (3 3 4 3 3). Results from the x convolution are then convolved in the y direction using $(3\ 3\ 4\ 3\ 3)^T$.

In the next stage of the algorithm, the temporally and spatially smoothed derivatives are combined to build ridge regression model components. The calculation required in this stage is similar to the outer product stage of the tensor algorithm, and produces six new matrices ($gxgx$, $gygy$, $gtgt$, $gxgy$, $gxgt$, and $gygt$). Another spatial smoothing is then performed on these matrices by way of three-element row and column convolutions and the smoothing mask (5 6 5). As with the tensor algorithm, values used in each smoothing mask of this algorithm were chosen to take the shape of a 2D Gaussian function, and were carefully evaluated to achieve the best possible tradeoff between accuracy and efficiency.

The smoothed ridge regression model components are fed into a ridge regression scalar estimator (detailed in [5] and [25]) to calculate a scalar value, k. k is then used together with the smoothed model components to calculate the optical flow fields as in [26]. The result is smoothed one last time using seven-element spatial convolution with the smoothing operator $\frac{1}{8}(1\ 1\ 1\ 2\ 1\ 1\ 1)$ to produce the final optical flow vectors, V_x and V_y.

4.2 GPU Implementations of the Ridge Regression Algorithm

As with the tensor algorithm, multiple versions of the ridge regression algorithm were implemented to run on a GPU. The original version used separable convolution for each convolution in the calculation. Once that was complete, two versions of the algorithm using non-separable convolution were created to explore the differences in performance between separable and non-separable convolution on the GPU. In this section, each of the GPU implementations is described in detail.

4.2.1 Using Separable Convolution

There are six main steps to the ridge regression algorithm: derivative calculation, derivative frame smoothing, construction of ridge regression model components, component smoothing, calculation of the optical flow, and smoothing of the optical

flow. The GPU implementation of the algorithm has a total of 12 computational kernels spread over these six steps that are called a total of 36 times. Like the tensor-based algorithm, the majority of the kernels perform convolution operations on the image data.

Derivative Calculation

The calculation of image derivatives in the first step of the ridge regression algorithm is accomplished using three separate GPU kernels, one each for derivative calculation in the x and y directions, and one for the calculation over time. Since the second stage of the calculation performs a 3D convolution and needs three sets of derivative frames, each of the derivative kernels is called three separate times on three different images, resulting in a total of nine kernel calls in the derivative calculation for each optical flow frame that is produced by the algorithm.

All three derivative kernels are implemented using a tiled approach based on the previously mentioned CUDA SDK separable convolution program [24], and are similar to the convolution kernels in the tensor algorithm implementation. The calculation of the x derivative uses 1D tiles, while both the y and t derivatives require 2D blocks of image data to run efficiently.

Derivative Frame Smoothing

The second stage of the GPU implementation smooths the results of the derivative calculation both temporally and spatially. The temporal smoothing kernel requires three derivative frames as input. It is called once each for the gx, gy, and gt values calculated in the first stage of the implementation.

Construction of Ridge Regression Model Components

Once the derivative frames have been temporally and spatially smoothed, components of the ridge regression model are calculated using a kernel similar to the outer product kernel of the tensor-based implementation. This kernel does not make use of the GPU's shared memory since none of the input data is reused. Part of the kernel

consists of the calculation of a unique index into global memory for each thread. A single pixel from each of the three smoothed derivatives (gx_s, gy_s, and gt_s) is then read in from this global memory location by each thread. Ridge regression components are calculated by multiplying these pixels together in different combinations to form six matrices: $gxgx$, $gxgy$, $gxgt$, $gygy$, $gygt$, and $gtgt$. The computation needed to construct the ridge regression model components is completed with a single GPU kernel call.

Component Smoothing

In this stage, the ridge regression model components go through another smoothing step that requires two GPU convolution kernels. Each kernel is called six times in order to process each of the six component matrices, for a total of twelve kernel invocations. Processing is done first in the x direction, then the y direction; the output of the x convolution is the input for the y convolution.

Optical Flow Calculation

Optical flow is calculated in the ridge regression algorithm in a manner similar to how it is calculated in the tensor algorithm — with one important difference. If the divisor is found to be less than a certain threshold, a scalar value k is calculated and added to values in the $gxgx$ and $gygy$ matrices. The velocity is then calculated using the new $gxgx$ and $gygy$ values.

In the FPGA version discussed in [14], the calculation of k is dependent on the value of the velocity of the pixel to the left of the pixel for which k is being calculated. For the GPU version, this was modified so that the velocity of the current pixel was first calculated using $k = 0$. k was then re-computed based on the resulting velocity instead of the velocity of the neighboring pixel. In this way, dependencies between pixels were removed and GPU parallelism was increased, since each pixel of the result could be calculated with a unique GPU thread. The accuracy of the final optical flow field result was not adversely affected by this change.

Smoothing of Optical Flow Results

One final smoothing step is applied to the velocity vector results produced in the optical flow calculation. This requires two more GPU kernels, one each for smoothing in the x and y directions. This stage produces the final velocity vector fields that are the output of the ridge regression algorithm.

4.2.2 Non-Separable Convolution

Each of the convolution kernels in the ridge regression algorithm uses separable convolution. In other words, the convolution is first computed in the x direction, then the results of that convolution are convolved in the y direction. This process requires two separate kernels to be used efficiently on the GPU. An alternative to separable convolution is to perform the same convolution in a single kernel using a 2D convolution operator. If the operator used in the separable convolution is a 3×1 matrix, for example, then the operator used in the non-separated convolution will be a 3×3 matrix.

Two additional versions of the ridge regression algorithm were created to test the effects of using non-separable convolution on the performance of the algorithm. In each case, only a single spatial convolution of the original implementation was modified. The convolution used in the smoothing of the ridge regression model components was chosen as the convolution to be replaced since the separable version used an operator with only three elements (5 6 5), and a smaller convolution operator required less shared memory in the non-separable version. In both modified versions of the algorithm, the two GPU kernels required to smooth the ridge regression components in the separable version (one for the x direction and one for the y) were replaced with a single 2D convolution kernel.

Row-Based

The first modified version used a 2D convolution operator based on a previous GPU implementation of a Sobel image filter. This implementation was simpler to code but not as robust as the second version. It was called with the number of

threads per block fixed at 320, and the number of blocks equal to the height of the input image so that each block processed one row of the image. For an image with a width greater than 320, each thread processed the convolution for more than one pixel. Pixels from the first 320 columns of the image were processed first, then the next 320 columns, and so on. One downside of this approach is that if the width of the input image was not a multiple of 320, threads would be idle in at least part of the computation. An image with width 321, for example, would have all threads active on the first pass over the data, but only a single active thread on the next pass.

Tile-Based

The second modified version decomposed the input image into smaller 2D tiles. This tile-based version was more difficult to program than the row-based implementation, and required nearly 60% more code (73 lines of source code vs. 43 for the row-based version). Most of the difficulty came about in ensuring that memory was accessed optimally. To accomplish this, the GPU kernel passed through two steps. In the first step, each thread read a pixel from global device memory into shared memory. The threads were synchronized, then in the second step each thread calculated the convolution for a completely different pixel than that which it read in from memory. Reading in and operating on different pixels ensured that accesses to global memory were properly aligned.

To achieve properly aligned and coalesced memory accesses, the first thread in each warp needed to access a memory location that was a multiple of 16 (the size of a half-warp of threads). Since the convolution operator required a border around the image tile in order to process pixels along the edge of the tile, simply accessing the first pixel in the tile with the first thread was not sufficient. Instead, an *apron* of pixels was placed around the tile so that the actual first pixel in the tile was accessed by a thread with a thread number that was a multiple of 16. In some cases this left threads idle during the second step of the processing, since more pixels were read in than were processed, but this was an acceptable tradeoff due to the long latency of the GPU's device memory. Once data was read into shared memory, the convolution

for the first pixel in the tile could be efficiently computed by the first thread in the block, the second pixel by the second thread, and so on.

Separable vs. Non-Separable Performance

The original separable convolution implementation out-performed both non-separable versions. Running on the 8800 GTX GPU and processing a 320x240 image, the separable version was about 4% faster than the row-based convolution and 19% faster than the tile-based version. As expected, for images with widths that were not multiples of 320 pixels, the separable kernel outperformed the row-based kernel by an even larger margine. On the Yosemite sequence, for example, which has a width of 252 pixels, the original kernel was nearly 21% faster than the row-based non-separable convolution.

The primary reason that the non-separable implementations run slower than the separable convolution is that they use memory less efficiently. A 2D convolution requires that border pixels be read in on all four sides of a tile. The 1D convolutions done with the separable kernel only require borders on half of the tile. 2D convolutions also require a larger amount of shared memory per block, thus reducing the number of blocks that can run concurrently on the GPU.

Chapter 5

Performance Results

In this chapter, results for GPU implementations of both optical flow algorithms are presented and discussed. For the sake of comparison, results from the FPGA designs of the same algorithms, discussed in [13] and [14], are also included. Implementations on both hardware platforms were tested with the Yosemite image sequence for accuracy and with randomly-filled image data for speed.

The FPGA implementation of the gradient tensor optical flow algorithm was created on a Xilinx XUP V2P board with a Virtex-II Pro XC2VP30 FPGA using VHDL and Xilinx EDK tools. The FPGA used has 13,969 slices and access to 256 MB of off-chip DDR memory.

The FPGA version of the ridge regression optical flow design was mapped to the Helios FPGA platform [27], which consists of a Virtex-4 FX60 FPGA with two PowerPC processors. The Helios board is specially designed for embedded vision applications. Xilinx's Base System Builder was used for the initial design of the system. The IP cores used limited the clock speed of the system to 100 MHz.

5.1 Performance

The performance of each implementation was measured in frames per second. For the GPU implementations, the frame rate was measured using CUDA timer functions. Table 5.1 summarizes the performance and accuracy results for both the tensor and ridge regression algorithms for the three GPUs tested for this study as well as the two FPGA implementations.

The gradient tensor algorithm achieved a higher frame rate than the ridge regression algorithm on both the FPGA and GPU implementations. In addition,

Table 5.1: Summary of the performance of the tensor and ridge regression optical flow algorithms on GPU and FPGA platforms.

Algorithm	Platform	320x240 (fps)	640x480 (fps)	Angular Error (degrees)
Tensor	XUP FPGA	258	64	12.9
Tensor	8400 GS	77	19	12.1
Tensor	8800 GTX	847	238	12.1
Tensor	GTX 280	1552	462	12.1
Ridge	Helios FPGA	60	15	6.8
Ridge	8400 GS	58	14	5.3
Ridge	8800 GTX	590	158	5.3
Ridge	GTX 280	1104	337	5.3

other than the 8400 GS, which has just two multiprocessors, the GPUs outperformed the FPGA designs for both algorithms.

For the tensor algorithm and a 640x480 image size, the 8800 GTX and the GTX 280 GPUs outperformed the FPGA implementation by 3.7× and 7.2×, respectively. One of the biggest factors that contributed to the poor performance of the FPGA on the gradient tensor algorithm was a limited memory bandwidth. It is likely that improvements to the FPGA system, such as using a custom memory interface module in the design, would reduce this performance discrepancy.

The FPGA implementation of the ridge regression algorithm only processes 640x480 images at 15 frames per second despite running on a newer FPGA. Interestingly enough, even the low-power GPU achieves nearly the same frame rate. The additional smoothing required in the algorithm significantly increases the memory bandwidth needed in the FPGA system and reduces the performance of the design. A larger SRAM on the Helios board could likely provide an additional 2× performance increase to about 30 frames per second for 640x480 images.

The GPU implementation of the ridge regression optical flow does not suffer from the same memory bandwidth problems, but still runs slower than the GPU tensor implementation. Using the 8800 GTX, the ridge regression implementation processes 640x480 images at 158 frames per second, compared to 238 frames per

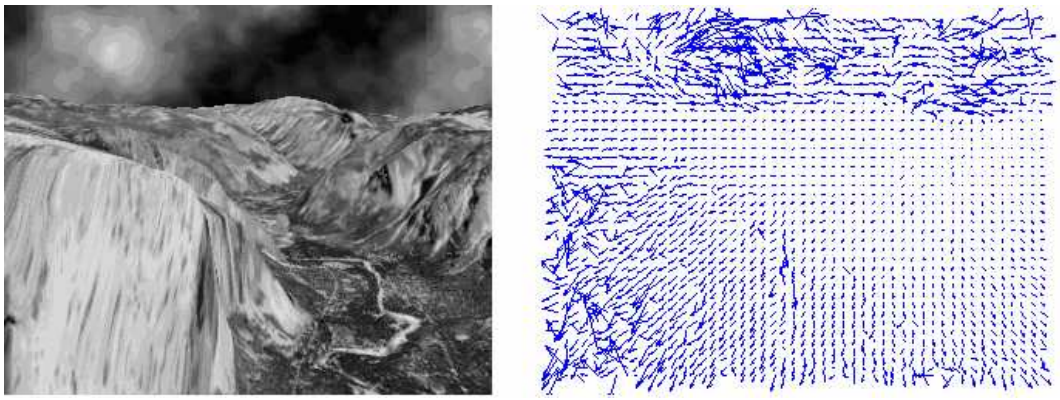

Figure 5.1: The Yosemite sequence and the measured optical flow field: 8th frame of Yosemite sequence (left) and its optical flow field (right)

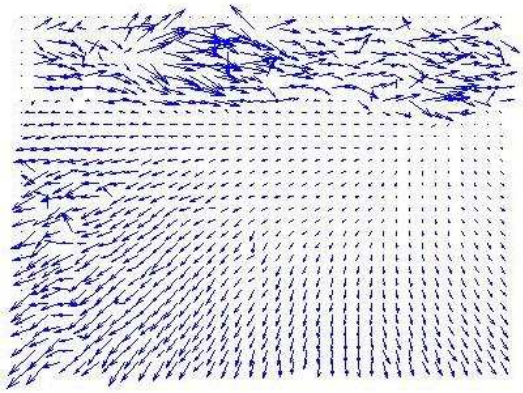

Figure 5.2: Measured optical flow field for the 8th frame of the Yosemite sequence using the ridge regression algorithm.

second for the tensor-based algorithm. The slowdown between the two GPU versions is due to the increased computation required in the ridge regression algorithm, which uses 36 kernel calls for each image compared to 21 in the tensor algorithm.

5.2 Accuracy

Figure 5.1 shows one frame of the Yosemite image sequence and the resulting optical flow field computed by the tensor-based FPGA implementation. Figure 5.2 shows the more accurate output resulting from the ridge regression implementation. The additional smoothing performed in the ridge regression algorithm allowed it to produce results that were around twice as accurate as those of the tensor algorithm.

The noise seen in the upper portion of both figures is due to the sky portion of the Yosemite sequence having very little contrast or image texture. The sky portion of the Yosemite image sequence is problematic for many optical flow algorithms, and in some cases is masked off entirely before the optical flow is calculated.

A comparison of the accuracy between the FPGA and GPU implementations shows that the GPU has a slightly better accuracy for both optical flow algorithms. On the tensor algorithm, the FPGA produced results with an average angular error of 12.9 degrees, while the GPU results had an average angular error of 12.1 degrees. For the ridge regression algorithm, the angular error improved to 6.8 degrees for the FPGA version and 5.3 degrees for the GPU version. The improved accuracy of the GPU implementations is due to the fact that both FPGA implementations use fixed point data and fixed bit widths, while the GPU implementations use single-precision floating point data.

5.3 Cost

The purchase cost of the XUP FPGA platform used to implement the tensor algorithm was around $1,600. (An academic version of the system can be purchased at a discount for about $300.) The fact that the FPGA platform was a prototype board and had some unused I/O interfaces contributed to this cost, and it is possible that a more customized and dedicated platform would have been less expensive. A significant amount of additional money could have also been spent upgrading the platform used in the implementation from a Virtex-II Pro to technology comparable to the 8800 GTX GPU, such as a Virtex-4 or 5. In comparison, the Helios FPGA platform used to implement the ridge regression algorithm contained a more modern Virtex-4 FPGA and cost about $2,000 per board when purchased at low quantities. When purchased in bulk, the Helios board could cost as low as $1,000.

At $1,600 and $2,000, both FPGA systems were relatively inexpensive, especially when compared to leading-edge FPGAs that often cost $1,000s and corresponding commercial platforms that can cost $10,000s. At the same time, the GPUs used in this study were even less expensive. The 8800 GTX and GTX 280 were purchased

about a year and a half apart. Both cost approximately $500 at the time of purchase. The downside of the GPUs is that a host PC is required to operate them, while the FPGA platforms can be stand-alone.

5.4 Power

The 8800 GTX GPU and its host together consume 200-300W of power. The GTX 280 consumes a similar amount, between 150-250W, when in full processing mode, but has a more dynamic power management architecture that allows it to idle at a much lower power consumption (25-30W)[1]. Neither the 8800 GTX nor the GTX 280 is suitable for most embedded applications. In contrast, the FPGA implementations consume much less power — the XUP FPGA used approximately 10W and the Helios system about 2W. This significantly lower power is typical of stand-alone FPGA solutions and allows them to be readily embedded into many applications that are constrained by size, weight, and power.

NVIDIA makes GPUs with fewer multiprocessors that consume less power than the 8800 GTX or GTX 280. To test the performance of a lower power GPU, both optical flow algorithms were also run on a system with an NVIDIA GeForce 8400 GS video card. The 8400 GS has two multiprocessors (compared to the 16 in the 8800 GTX). When running the tensor algorithm, the 8400 GS achieved a performance of 19 fps for the 640x480 image size (77 fps for the 320x240 size) with a power consumption of approximately 25% that of the 8800 GTX.

5.5 Flexibility

One of the strengths of an FPGA system is its flexibility. FPGA memory interfaces have much lower latency than device memory on NVIDIA's GPUs and can provide greater algorithmic flexibility than is available with a GPU implementation. An FPGA platform also allows for greater flexibility when interfacing with the rest of an embedded system, since all required I/O interfaces and data processing can be built into the design using various organizations and form factors.

[1] See `http://tiny.cc/benchmarkreviews`

Table 5.2: Source lines of code for each implementation. The FPGA implementations required about 5.7× and 8.7× the number of lines of source code, respectively.

Implementation	SLOC
Tensor FPGA	6451
Tensor GPU	1131
Ridge FPGA	17118
Ridge GPU	1973

5.6 Productivity

Part of the price paid for the flexibility of an FPGA design is in design productivity. The gradient tensor FPGA design took two graduate students an entire summer to complete, and the ridge regression design was completed by one of the same students in about six months. In contrast, neither GPU design took longer than two or three weeks for a single student. Although it is difficult to directly compare the skill level and experience of the students involved, in general the FPGA implementations of the algorithms discussed in this thesis required about 10× to 12× the design time of the GPU implementations.

An alternative measure of the complexity of each design is the number of source lines of code required for each implementation. Table 5.2 summarizes the source lines of code needed for each version of the algorithm, obtained using the word count function in Linux. The FPGA implementations required about 5.7× and 8.7× the number of lines of source code, respectively.

It is likely that one of the main factors that contributed to the large difference in design time between the two technologies was the relative difficulty of performing design and debug iterations. GPU code is written in an extension of the C language and can be compiled and tested almost instantaneously. This quick turn-around time makes it possible to iterate much more quickly and efficiently through modifications of the solution or to investigate various different algorithms for an application. For example, searching for optimal block and thread size partitions for the various GPU

kernels in the tensor-based implementation was done with simple parameter changes and was accomplished in a few hours.

The FPGA optical flow designs were not nearly as parameterizable, and alternative implementations were much more difficult and time-consuming to test or simulate. Further, the use of logic analyzers and similar tools for hardware debug was significantly more cumbersome than the software debug model supported by the GPU platform. Even as long as the FPGA design took to implement, it is likely that development time would have been much longer had an entirely original design been created in place of a design that used IP cores.

5.7 Summary of Results

To summarize, the 8800 GTX and GTX 280 GPUs significantly outperformed the FPGAs used on both optical flow algorithms, although enhancements to the FPGA designs would likely decrease the performance differential. The GPU implementations were also slightly more accurate and had a 10× to 12× advantage in design time and effort. The only advantages of the FPGA designs were their flexibility and low power consumption.

Chapter 6

Conclusions and Future Work

As the results of this study show, GPUs are a viable alternative to FPGAs for implementations of optical flow and similarly structured algorithms. The GPUs tested offer significant advantages in both productivity and performance over comparable FPGA implementations. At the same time, the power consumption of the GPUs limits their usefulness in some areas of industry. To be truly competitive in the embedded world, for example, the power consumption of GPUs needs to decrease significantly — especially since many uses of the optical flow calculation come from low-power embedded systems such as those in small, autonomous robot vehicles.

Currently, GPUs are improving at a very fast rate, with each new GPU generation being released every one to two years. It may not be long before a low-power high-performance GPU is available. With the upcoming release of Intel's Larrabee chip, competition for the GPU market will likely increase. Hopefully this competition will be a catalyst for new innovation and increased GPU performance. Whatever the case, GPU programming is only likely to grow in importance in the near future.

To extend the work done in this thesis in the future, the optical flow designs that have been implemented on the GPU could be integrated into a larger computer vision system. The frame rates that NVIDIA's GPUs are able to achieve on the tensor and ridge regression algorithms are much faster than is necessary for a real-time application (for which 30 to 60 fps is likely sufficient). Running the optical flow at a slower frame rate would leave room for other processing to be done on the GPU. The motion fields produced by the optical flow calculation could be input to an obstacle avoidance application running on the same GPU, for example, to take advantage of this additional processing power.

Bibliography

[1] B. K. P. Horn and B. G. Schunk, "Determining optical flow," *Artificial Intelligence*, vol. 17, pp. 185–203, 1981. 1, 2, 3, 4

[2] L. Zelnik-Manor, "The optical flow field," PowerPoint presentation, October 2004. [Online]. Available: http://tiny.cc/flow_examples 2

[3] S. S. Beauchemin and J. L. Barron, "The computation of optical flow," *ACM Comput. Surv.*, vol. 27, no. 3, pp. 433–466, 1995. 3

[4] B. D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision," in *IJCAI81*, 1981, pp. 674–679. [Online]. Available: http://citeseer.ist.psu.edu/180224.html 3

[5] G. Farneback, "Very high accuracy velocity estimation using orientation tensors, parametric motion, and simultaneous segmentation of the motion field," in *Proc. ICCV*, vol. 1, 2001, pp. 77–80. 4, 7, 33, 35

[6] ——, "Fast and accurate motion estimation using orientation tensors and parametric motion models," in *Proc. ICPR*, vol. 1, 2000, pp. 135–139. 4, 19

[7] B. Jahne, H. Haussecker, H. Scharr, H. Spies, D. Schmundt, and U. Schur, "Study of dynamical processes with tensor-based spatiotemporal image processing techniques," in *Proc. ECCV*, vol. 2, 2000, pp. 322–336. 4

[8] L. Haiying, C. Rama, and R. Azriel, "Accurate dense optical flow estimation using adaptive structure tensors and a parametric model," *IEEE Trans. Image Processing*, vol. 12, pp. 1170–1180, Oct 2003. 4

[9] H. Wang and K. Ma, "Structure tensor-based motion field classification and optical flow estimation," in *Information, Communications and Signal Processing 2003*, vol. 1, Dec 2003, pp. 66–70. 4

[10] D. J. Heeger, "Model for the extraction of image flow," *Journal of the Optical Society of America A*, vol. 4, pp. 1455–1471, Aug. 1987. 6

[11] J. L. Barron, D. J. Fleet, S. S. Beauchemin, and T. A. Burkitt, "Performance of optical flow techniques," *International Journal of Computer Vision*, vol. 12, no. 1, pp. 43–77, 1994. [Online]. Available: http://citeseer.ist.psu.edu/barron92performance.html 6

[12] J. Diaz, E. Ros, F. Pelayo, E. M. Ortigosa, and S. Mota, "FPGA-based real-time optical-flow system," *IEEE Trans. Circuits and Systems for Video Technology*, vol. 16, no. 2, pp. 274–279, 2006. 7

[13] Z. Wei, D. J. Lee, B. Nelson, and M. Martineau, "A fast and accurate tensor-based optical flow algorithm implemented in FPGA," in *IEEE WACV 07*, 2007, p. 18. 7, 19, 33, 41

[14] Z. Wei, D. Lee, B. Nelson, , and J. Archibald, "Real-time accurate optical flow-based motion sensor," in *International Conference on Pattern Recognition (ICPR)*, December 2008. 7, 33, 37, 41

[15] M. Durkovic, M. Zwick, F. Obermeier, and K. Diepold, "Performance of optical flow techniques on graphics hardware," *Multimedia and Expo, IEEE International Conference on*, vol. 0, pp. 241–244, 2006. 8

[16] C. Zach, T. Pock, and H. Bischof, "A duality based approach for realtime tv-l1 optical flow," in *Pattern Recognition (Proc. DAGM)*, Heidelberg, Germany, 2007, pp. 214–223. 8

[17] Y. Mizukami and K. Tadamura, "Optical flow computation on compute unified device architecture," in *14th International Conference on Image Analysis and Processing (ICIAP 2007)*, September 2007, pp. 179–184. 8

[18] J. Chase, B. Nelson, J. Bodily, W. Z., and L. D.J., "Real-time optical flow calculations on FPGA and GPU architectures: A comparison study," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '08)*, IEEE Computer Society. IEEE Computer Society Press, April 2008. 9, 16

[19] NVIDIA, "NVIDIA CUDA programming guide 2.0," 2008. 12

[20] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008. [Online]. Available: http://dx.doi.org/10.1109/JPROC.2008.917757 16

[21] G. H. Granlund and H. Knutsson, *Signal Processing for Computer Vision*. Kluwer Academic Publishers, 1995, iSBN 0-7923-9530-1. 19

[22] G. Farneback, "Orientation estimation based on weighted projection onto quadratic polynomials," in *Proc. Vision, Modeling, and Visualization 2000*, Nov 2000, pp. 89–96. 19

[23] B. Johansson and G. Farneback, "A theoretical comparison of different orientation tensors," in *Proceedings SSAB02 Symposium on Image Analysis*, Mar 2002, pp. 69–73. 20

[24] V. Podlozhnyuk, "Image convolution with CUDA," Online, pdf, June 2007. [Online]. Available: http://tiny.cc/separable 28, 36

[25] A. Hoerl and R. Kennard, "Ridge regression: Biased estimation for nonorthogonal problems," *Technometrics*, vol. 12, no. 1, pp. 55–67, 1970. 33, 35

[26] J. Grob, "Linear regression," *Lecture Notes in Statistics*, 2003. 35

[27] Http://www.ece.byu.edu/roboticvision/helios/. 41