Wright State University

CORE Scholar

[Browse all Theses and Dissertations](#)

[Theses and Dissertations](#)

2016

GPU-Accelerated Feature Tracking

Alex Graves

Wright State University

Follow this and additional works at: https://corescholar.libraries.wright.edu/etd_all

Part of the [Computer Sciences Commons](#)

Repository Citation

Graves, Alex, "GPU-Accelerated Feature Tracking" (2016). *Browse all Theses and Dissertations*. 1477.
https://corescholar.libraries.wright.edu/etd_all/1477

This Thesis is brought to you for free and open access by the Theses and Dissertations at CORE Scholar. It has been accepted for inclusion in Browse all Theses and Dissertations by an authorized administrator of CORE Scholar. For more information, please contact library-corescholar@wright.edu.

GPU-Accelerated Feature Tracking

A thesis submitted in partial fulfillment of the
requirements for the degree of
Master of Science

By

Alex Graves
B.S., Wright State University, 2009

2016
Wright State University

WRIGHT STATE UNIVERSITY

GRADUATE SCHOOL

April 20, 2016

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Alex Graves
ENTITLED GPU-Accelerated Feature Tracking BE ACCEPTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF Master of Science.

Thomas Wischgoll, Ph. D
Thesis Director

Mateen Rizki, Ph. D
Chair, Department of Computer Science
and Engineering

Committee on
Final Examination

Thomas Wischgoll, Ph. D

Arthur Goshtasby, Ph. D

Krishnaprasad Thirunarayan, Ph. D

Robert E. W. Fyffe, Ph. D
Vice President for Research and
Dean of the Graduate School

Abstract

Graves, Alex. M.S., Department of Computer Science and Engineering, Wright State University, 2016. *GPU-Accelerated Feature Tracking*.

The motivation of this research is to prove that GPUs can provide significant speedup of long-executing image processing algorithms by way of parallelization and massive data throughput. This thesis accelerates the well-known KLT feature tracking algorithm using OpenCL and an NVidia GeForce GTX 780 GPU. KLT is a fast, efficient and accurate feature tracker but can easily suffer from low frame rates when tracking many features in an HD video sequence. This research explains how KLT could benefit from GPGPU programming and provides the corresponding OpenCL implementation. Additionally, various optimization techniques are emphasized to further boost GPU performance. The experiments conducted prove that when tracking over 500 features in an HD dataset, GPU-based KLT provides a 92% reduction in total runtime compared to a CPU-based implementation. Furthermore, the experiments demonstrate that these features are tracked while maintaining similar accuracy to the CPU results.

Table of Contents

Chapter 1: Introduction	1
1.1 The Problem Statement	1
1.2 Context	2
1.3 Methodology of Research	3
1.4 Thesis Organization	4
Chapter 2: Background	5
2.1 KLT Overview	5
2.2 KLT: Selecting Features	7
2.2.1 Eigenvalues	8
2.3 Tracking Features	11
2.3.1 Gaussian Image Pyramids	13
2.4 GPGPU Programming	14
2.5 Related Work	15
Chapter 3: OpenCL Architecture and GPGPU Programming	18
3.1 Platform Model	18
3.2 Execution Model	19
3.3 Memory Model	21
3.3.1 Image Data in OpenCL Memory Space	22
3.4 Programming Model	23
Chapter 4: Implementation and Discoveries	27
4.1 Image Convolution	27
4.1.1 Tiled Image Convolution	29
4.1.2 Data Dependencies	31
4.2 Hessian Calculation	32

4.2.1 Bank Conflicts.....	32
4.2.2 Control Flow Optimizations	33
4.2.3 Memory Coalescence.....	34
4.3 Tracking Features.....	35
4.3.1 Reducing Data Transfers	36
4.3.2 Interpolation	36
4.3.3 NDRange Optimizations.....	38
Chapter 5: Results & Analysis	41
5.1 Performance	42
5.2 Accuracy.....	46
5.3 Profiling.....	48
Chapter 6: Future Work & Conclusions	51
6.1 Future Work.....	51
6.2 Conclusion.....	53
References	55

List of Figures

Figure 2.1. Good Features to Track.....	9
Figure 2.2 Summary of the KLT Algorithm.....	12
Figure 2.3. Gaussian Image Pyramid.....	13
Figure 3.1. The OpenCL Platform Model [22]	19
Figure 3.2 NDRange [19]	20
Figure 3.3. The OpenCL Memory Model [22]	22
Figure 4.1. Image Convolution	28
Figure 4.2. Tiled Image Convolution	30
Figure 4.3. Warp Divergence.....	34
Figure 4.4. Coalesced Memory Access vs Non-Coalesced Memory Access.....	34
Figure 5.1. KLT Runtime, CPU vs GPU	42
Figure 5.2. KLT Functions as a Percentage of Total Runtime	43
Figure 5.3. Average Function Runtimes	44
Figure 5.4. Total Runtime with Varying Dataset Sizes, HD	46
Figure 5.5. Root-Mean-Squared Error, SD	47
Figure 5.6. Root-Mean-Squared Error, HD.....	48
Figure 5.7. The NVidia Visual Profiler.....	49

List of Tables

Table 5.1. Lost Feature Count and Average Frames per Second, HD Dataset	45
Table 5.2. Average Frames per Second Based on Image Resolution	46
Table 5.3. Total Program Throughput.....	49

Acknowledgment

I would like to thank Dr. Wischgoll, Dr. Prasad, Dr. Goshtasby and Dr. Liu for their support, guidance and also challenging me throughout the graduate program. In addition, I'd like to thank Dr. Vasquez at AFRL for his motivation, support and providing me the opportunity to grow in my collegiate and professional careers.

Chapter 1 Introduction

1.1 The Problem Statement

Modern Central Processing Unit (CPU) performance and speed have begun to plateau over recent years due to “the power wall,” thus prompting more research into multi-core and many-core systems [30]. General Purpose Graphics Processing Unit (GPGPU) programming is a programming paradigm which employs Graphics Processing Units (GPUs) to run code typically executed on the CPU in order to provide performance gain by way of parallelism and data throughput. In addition to the power wall problem, the need for GPGPU programming has increased as the size of image datasets has continued to increase. New methods and hardware must be discovered to keep pace with the processing power and small form-factor required to handle high definition video in near real-time. Algorithms used in medical image analysis [21], remote sensing applications, autonomous vehicle technology and facial recognition are all computationally demanding real-world applications that can benefit from parallelism provided by GPUs for real-time results.

In image processing, high definition (HD)(1920x1080) image resolution is now a standard, with ultra-high definition (UHD) (3840x2160) 4K resolution on the horizon. Most image processing algorithms perform per-pixel operations, meaning each pixel must be processed at least once initially. With HD resolution having 5 times more pixels than standard definition, it is no surprise that today's CPUs cannot provide the processing speed required for handling HD imagery, especially at typical HD frame rates

of 50 or 60 frames per second (fps). However, when programming GPUs and other parallel programming devices such as Field Programmable Gate Arrays (FPGAs) and Digital Signal Processors (DSPs), it is easy to lose performance gain due to lack of optimization. GPGPU programming presents many potential bottlenecks and to achieve the highest performance gain possible, all optimization techniques must be exhausted. This thesis utilizes a well-known feature tracker to showcase the resulting optimizations.

1.2 Context

Feature tracking is the process of following, or "tracking", unique features in an image sequence. A feature can hold many properties, such as textured areas, lines or contours, sharp contrasts, edges or corners. Accurately tracking features is a computationally intense and time-consuming process that often doesn't meet real-time application needs. Feature tracking is relevant in many image processing algorithms such as image registration and object tracking. Additionally, it is used in medical imaging analysis for 3-D reconstructions of X-ray, CT, MRI and PET images and also in feature matching techniques which are pertinent to facial recognition algorithms.

Many image processing algorithms, including feature tracking, are well-suited for parallelization via GPGPU programming. These algorithms require per-pixel operations in which some operation is performed on each pixel independently. With no co-dependence on adjacent pixel operations, each operation can be performed in parallel and in any order. Additionally, there is often much overlap in data access within these methods. Parallelization using Open Computing Language (OpenCL) can take advantage of this overlap to dramatically reduce computation time. OpenCL is a framework that allows programmers to specify code to execute on devices such as GPUs

using C-like syntax and data structures. This thesis uses a common feature tracker to demonstrate OpenCL acceleration and further focuses on optimization techniques for GPGPU programming.

Several issues must be considered when writing a GPGPU program. First, parallelism in the algorithm under test must be exposed. Some algorithms, like many image processing algorithms, are naturally parallelizable, but others must be slightly altered in order to achieve the necessary concurrency. Secondly, data transfers between the host (CPU) memory space and device (GPU/FPGA) memory space must be minimized. Data transfers are commonly the largest bottleneck in GPGPU programs. For example, data rates over a PCIe x16 bus approach 8 GB/s whereas global memory access in a GTX 280 is 141 GB/s, according to NVidia [4]. This means that accessing data that is already on the device is 17x faster than copying the data to the device. Lastly, data throughput must be maximized in order to fully optimize a GPGPU program. GPUs ability to process massive amounts of data simultaneously gives them an edge over their CPU counterparts that excel at instruction throughput, or processing many instructions in a short period of time.

1.3 Methodology of Research

This research utilized profiling tools to identify bottlenecks and long execution paths of existing code. The identified bottlenecks were then ported to OpenCL to expose parallelism and therefore speed up their computation time. Once the basic or naive versions of these OpenCL functions were implemented, they were further examined to optimize and expose additional performance gain. With each layer of implementation, the accuracy compared to the original code was maintained. Over 500

features are tracked in each frame in a video sequence with various statistical measures of the pixel locations used as a benchmark for accuracy. The implementation was executed on several datasets with varying resolution to diversify the results and show how the hardware affects the performance increase of the algorithm.

1.4 Thesis Organization

Chapter 2 dives into the background of the Kanade-Lucas-Tomasi (KLT) feature tracking algorithm, which includes selecting and tracking features and attempts to shed light on how such an algorithm would benefit from GPGPU programming. In Chapter 3, OpenCL is introduced and the OpenCL programming model is discussed. Chapter 4 combines OpenCL with the KLT feature tracker, showing which parts of the algorithm were suitable for OpenCL parallelization and mentions key discoveries made during the research. The results and future work of the OpenCL acceleration are exhibited in Chapter 5, verifying that GPGPU programming can add enormous performance gain to image processing algorithms, easily reducing the runtime of processing HD video by 92% while maintaining similar accuracy when compared to a CPU-based implementation.

This research aims to show the magnitude of performance increase realized by GPGPU programming. Furthermore, it presents the importance of optimizing GPU-accelerated code and the impact that optimization has on the overall execution of the algorithm. Finally, this research demonstrates the major impact that GPGPU programming has on single-threaded applications with modern CPU performance and small form-factor succumbing to the needs of big data.

Chapter 2 Background

In this chapter, the focus is on the KLT algorithm and how it could benefit from parallelization. Then, KLT is further simplified into two distinct processes: feature selection and feature tracking. Next Amdahl's Law is introduced to determine if KLT or any algorithm in general is suitable for GPGPU acceleration. Finally, any related work that has helped influence this GPU implementation of KLT is reviewed.

2.1 KLT Overview

KLT feature tracking sets the standard for today's image registration algorithms [3][7][17][31]. It is fast, efficient and accurate. The fundamental idea behind KLT is to minimize a Sum of Squared Differences (SSD) objective function between two image frames through a set of parameters to register or align the images [15]. SSD is advantageous because it typically converges quickly, the gradient is easy to derive and it is easy to implement [7]. Using an SSD measurement, KLT can be summarized by the equation below [2]:

$$\sum_x [I(W(x;p)) - T(x)]^2. \quad (1)$$

The two image frames are referred to as $I(x)$ and $T(x)$, where $I(x)$ is the current image and $T(x)$ is the template, or, in a sequence of images, $T(x)$ is the image at time $t=0$ and $I(x)$ is the next image at time $t=1$. $W(x;p)$ refers to a warping function, which can vary

depending on the various transformations in the image, such as affine transformations handling rotation, scaling, shearing, or simple translations in the x, y domain. For this research, the focus was on simple translations, so the warping function becomes [2]

$$W(x; p) = \begin{bmatrix} 1 & 0 & b_1 \\ 0 & 1 & b_2 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \quad (2)$$

In addition to SSD, KLT uses a Gauss-Newton algorithm to estimate the warp function parameters between the two frames. Gauss-Newton, at its core, is an iterative method to approximate the solution to a non-linear least squares problem using an initial guess, a function and that function's first derivative. In KLT, the warp parameters between the two images are computed by minimizing the SSD function as [2]:

$$\Delta p = H^{-1} \sum_x \left[\nabla I \frac{\delta W}{\delta p} \right]^T [T(x) - I(W(x; p))]. \quad (3)$$

In Equation (3), H is the Hessian matrix [2]:

$$H = \left[\nabla I \frac{\delta W}{\delta p} \right]^T \left[\nabla I \frac{\delta W}{\delta p} \right], \quad (4)$$

∇I is the gradient of the image I , and $\frac{\delta W}{\delta p}$ is the Jacobian. The Jacobian is the first derivative of the vector-valued warping function $W(x; p)$ and can be reduced to a constant when considering simple translations. This reduction simplifies the Hessian to [2]:

$$H = \begin{bmatrix} g_x^2 & g_x g_y \\ g_x g_y & g_y^2 \end{bmatrix}. \quad (5)$$

Δp is the parameter update that must converge below a user-defined threshold, and if it does so within a short number of iterations, then this feature can be successfully tracked. Δp is then added to the warp parameters p :

$$p = p + \Delta p, \quad (6)$$

and Equations (3) and (6) are repeated until the solution is minimized.

With this summary of KLT in mind, it may not be easy to see where feature tracking fails to maintain real-time processing in an HD video sequence. It is necessary to dissect KLT further to expose where the long execution paths exist and how a feature tracker such as KLT can benefit from GPGPU programming. To do this, KLT is divided into two main functions: feature selection and feature tracking. In feature selection, the entire image is processed to produce the most prominent features which will end up being the best features to track. Feature tracking is the iterative process that tracks those selected features by examining a small neighborhood around them and attempting to find those same feature neighborhoods in the subsequent image using Equations (3) and (6) above.

2.2 KLT: Selecting Features

KLT focuses on finding the best features to track in a given image. Shi, Jianbo and Tomasi examine the quality of a feature, and the authors define a “good” feature as one that can be tracked well, which in turn will optimize the tracking process [26]. Consequently, a feature that can be tracked well is one in which the minimum eigenvalue of the 2x2 Hessian matrix H is above some predefined threshold. The Hessian is a second-derivative operation that results in a 2x2 matrix in the x and y

directions via Equation 5. If an Eigen decomposition is performed on H , the two resulting eigenvalues λ_1 and λ_2 will describe the tracking quality of that pixel and its surrounding neighborhood. If both eigenvalues are sufficiently large, they are considered good features to track. The entire image is divided into small, overlapping pixel windows within which the Hessian can be derived and the Eigen decomposition performed. This process is performed each time features need to be selected or replaced when tracking. Calculating the Hessian is usually one of the most time-consuming operations since it evaluates the entire image, but it is also inherently parallel. Thus, the Hessian computation is typically a good candidate for GPU acceleration.

2.2.1 Eigenvalues

Eigenvectors are a mathematical concept governed by the equation

$$Ax = \lambda x, \tag{7}$$

Where A is a linear transformation represented by the matrix A , x is a non-zero vector and λ is a scalar value. Any value for λ which yields a solution to Equation (7) is called an eigenvalue. The non-zero vector x which corresponds to this value is called the eigenvector. Eigenvalues and eigenvectors have a myriad of applications in math and engineering, and in this particular case, an eigenvalue represents the noise level or the magnitude of change in intensity within a small image patch or image window.

Eigenvalues can define a local maximum, local minimum, or an edge in the image window [28]. Two large eigenvalues imply a high contrast in intensity values between the window center and its surrounding environment – a local maximum, while only one

large eigenvalue typically describes an edge. Features along edges are not good features to track because of the well-known aperture problem [20]. Two small eigenvalues are indicative of a window with fairly constant intensity throughout and is also not worth tracking because this image window would be difficult to identify in the next frame. In addition, the greater eigenvalue cannot be arbitrarily larger than the smaller eigenvalue as this case describes too large of an invariance in intensity in the window. Figure 2.1 shows what to expect from looking at the eigenvalues.

Figure 2.1. Good Features to Track

When examining the eigenvalues, only cases in which both values are sufficiently large are considered. Therefore, looking at the minimum or smaller eigenvalue can accurately describe the pixel window. To find the minimum eigenvalue of a 2x2 matrix, the eigenvalue problem in (7) can be rewritten as

$$(A - \lambda I) * x = 0, \quad (8)$$

and if x is non-zero, the solution can be determined by taking the determinant of $(A - \lambda I)$, yielding the characteristic equation

$$\det \begin{bmatrix} a - \lambda & b \\ c & d - \lambda \end{bmatrix}, \quad (9)$$

where a, b, c, d make up the four values in the A matrix. Finally, solving with the quadratic formula, the eigenvalues are:

$$\lambda = \frac{(a + d) \pm \sqrt{(a + d)^2 - 4\det(A)}}{2}. \quad (10)$$

Once the entire image is processed, the resulting list of minimum eigenvalues is sorted, yielding the top n values as the features to be tracked in the image sequence. KLT can optionally implement a minimum distance function between feature points. Briefly, the strongest feature points are selected such that there are a minimum number of pixels separating adjacent features. This ensures that features are evenly distributed throughout the image and further increases registration accuracy by avoiding large clusters of features.

Sorting the list of eigenvalues is another process that could benefit from GPU acceleration even though sorting isn't a naturally parallel operation. Still, sorting functions such as radix sort have been implemented for GPUs [24]. Sorting on the CPU space is certainly one of the longer-running functions since the eigenvalue list can return several hundred-thousand values depending on the image size and the pixel window size. In the end, the Hessian calculation provides several potential areas of parallelization via GPGPU programming.

2.3 Tracking Features

Now that the strongest features in an image have been selected, they can be tracked through a sequence of subsequent images. The first step in tracking features is to compute the *image difference* and *gradient sum* in a small pixel window around each feature. The *image difference* is a difference in pixel values of the current image $I(x)$ and reference image $T(x)$, while the *gradient sum* is a separate accumulation in the x- and y- directions of $I(x)$ and $T(x)$. The images are warped with the warping function $W(x;p)$ prior to performing the arithmetic operations. The result is three separate image windows for the difference, x and y summations.

Next, the Hessian is computed in the same manner as in the feature selection process, resulting in a 2x2 matrix. Then the 2x1 error vector is computed which is simply the sum of multiplying the image difference values by the gradient values. The resulting values are then multiplied by a step factor [2]; however research shows that using a value of one resulted in more accurate tracking [3].

Given the 2x1 error vector and the 2x2 Hessian matrix, a linear equation can be formed and solved providing a difference value in the x- and y-directions [3]:

$$\begin{bmatrix} g_x^2 & g_{xy} \\ g_{yx} & g_y^2 \end{bmatrix} \begin{bmatrix} d_x \\ d_y \end{bmatrix} = \begin{bmatrix} e_x \\ e_y \end{bmatrix}, \quad (11)$$

where the vector d is the unknown delta that eventually makes up Δp from Equation (3) and the vector e is the 2x1 error vector. Because KLT is a Gauss-Newton style algorithm, the coefficients of vector d are compared against a small user-defined threshold and the whole process is iterated upon until both values are below the threshold or a maximum

number of iterations have been reached. If the maximum number of iterations is reached before converging, the feature is considered lost and the feature selection process must be called again to replace the lost feature. Figure 2.2 illustrates KLT in 9 iterative steps [2].

Figure 2.2 Summary of the KLT Algorithm

This feature tracking algorithm is a computationally intense process that consumes roughly half of the processing time in KLT. When considering 500 or more features, single-threaded CPU code can only track one feature at a time. GPGPU programming can immensely improve this algorithm since features are not dependent on each other. That is, the output of tracking one feature does not impact the input of tracking another feature. Therefore, 500+ features can be tracked concurrently on a GPU in less time than it takes the CPU to track one feature. This is especially true if each

feature must be tracked several times per image pair if utilizing an image pyramidal approach.

2.3.1 Gaussian Image Pyramids

Image pyramids in image processing aide in image compression, image analysis and graphics processing [1]. KLT and many other tracking algorithms commonly utilize a coarse-to-fine method of tracking features using image pyramids. The idea is that the two images used for tracking are scaled down by a factor of n prior to the tracking process. Then, the images are scaled up to a more fine-grained resolution and submitted to the tracking process again. This is repeated until features are tracked using the original, native resolution. Figure 2.3 illustrates the creation of image pyramids. The number of pyramid levels can vary based upon the algorithm, but in the implementation used in this research, the number of pyramid levels is two. This means each image is subsampled by half-resolution to arrive at pyramid level 0, then scaled back to the original resolution for pyramid level 1.

Figure 2.3. Gaussian Image Pyramid [1]

Image pyramids for tracking features is especially useful in KLT. By design, KLT often fails during or after large-scale translations or rotations. By implementing tracking via an image pyramid, these large translations can appear much smaller in the coarser-grained resolutions, facilitating the tracking of larger displacements [9]. Therefore, it is a way to increase tracking accuracy and persistence at the cost of a few extra iterations of the tracking process.

In sequential code, it's been shown that the tracking process can easily become one of the more lengthy processes, especially if the number of features tracked is large. However, since tracking a feature is not dependent on any other feature, the process can be easily parallelized. Additionally, the number of features tracked in each image can be greatly increased when the process is executed on a GPU which can result in more accurate registration.

2.4 GPGPU Programming

This chapter shows that several aspects of KLT can benefit greatly from GPU acceleration. Ideally, the first step in writing a GPGPU program is to evaluate an algorithm for any potential parallelization. Without parallelization, GPUs will provide very little performance increase, if any. This process of exposing parallelism within an algorithm is an important step and sometimes the most difficult. However, before jumping in right away, software engineers can estimate the performance gain of parallelization by way of Amdahl's Law which is used to estimate the maximum speedup gained by executing a fraction of a program in parallel through the use of additional processors (or cores in many cases) [13].

Amdahl's Law can be explained by Equation (12) below:

$$Speedup = \frac{1}{(1 - p) + \frac{p}{n}} \quad (12)$$

where p is the percentage of the program that is parallelizable and n is the number of additional processors, or in this case, cores that can execute the parallel code. To simplify Amdahl's Law, it is common to consider n as a very large number to effectively reduce the fraction p/n to 0, simplifying Equation (12) to

$$Speedup = \frac{1}{(1 - p)}. \quad (13)$$

Now a value for p can be inserted to estimate the theoretical maximum possible speedup of a program. This is an important equation because if an algorithm is 50% parallelizable, and the parallel code is fully optimized using infinitely many cores, it shows that the total algorithm will only experience at most a 2x speedup.

2.5 Related Work

Using GPUs for general purpose programming is almost as old as the GPU itself. Programmers have realized the computing power of GPUs over CPUs for decades. An early analysis of general purpose GPU computing by Thompson, et al. displayed "startling performance improvements" when comparing GPU and CPU performance [27]. Prior to NVidia's release of CUDA in 2007, programmers were forced to "sneak" operations into the GPU pipeline using pixel shaders and frame buffer objects. This method of utilizing graphics libraries such as OpenGL and Cg was difficult to code and not always the most efficient approach since programmers had to align their algorithms to fit the graphics pipeline. Although the concept of parallelization was similar, the execution was completely different. CUDAs C-like syntax and use of familiar data types facilitated GPU programming for software engineers lacking knowledge in graphics.

The natural data parallelism that occurs in many image processing algorithms makes them a great fit for GPU programming. Specifically, KLT has been implemented using CUDA for numerous applications including HD feature tracking and tracking through affine transformations in illumination changes. Fassold, et al. achieve more than 25 fps when tracking several thousand feature points using GPUs [9], meanwhile Zach, et al. use KLT and GPUs to track features in real-time through varying camera exposures [33].

This research implements a KLT feature tracker using OpenCL to track over 500 features in HD video at an increased frame rate. Similar to Fassold [9], the implementation detailed in this thesis ports the entire feature selection process to the GPU except for the minimum distance calculation which is difficult to parallelize. However, Fassold's KLT implementation iterates until the solution converges instead of setting a fixed number of iterations. Conversely, this implementation uses a pyramidal approach and works to maintain a close similarity to the original work by Kanade and Lucas similar to the work of Farias, et al [8].

Optimization is also an important research topic for GPGPU programming. Although GPUs provide a massive performance gain over CPUs, certain factors can limit performance gain. Hestness, et al. show that minimizing data transfers on discrete GPU systems can have huge benefits to performance [12]. Equivalently this implementation has discovered that one of the highest latency operations in GPGPU programming is data transfer. In the same way that this research emphasizes maintaining high GPU thread occupancy, Ryoo, et al. stress Thread Level Parallelism (TLP) to hide latency [23]. However, in this implementation, some optimizations were left up to the compiler, like

loop unrolling, instead of explicitly forcing them. Likewise, previous research also focuses on creating a balanced workload for threads, since some threads in certain applications will finish faster than other threads [6]. This is typically not an issue for a GPU-based KLT implementation and so was not considered.

The remaining research aims to explain additional GPU optimizations and results which validate the claims made. The OpenCL architecture is analyzed to exploit its strengths and apply them to the GPU-accelerated KLT implementation. Next, profiling results of the algorithm are studied to determine the best areas for parallelization. Finally, any further work that was considered but not pursued will be addressed.

Chapter 3 OpenCL Architecture and GPGPU Programming

OpenCL provides an Application Programming Interface (API) for writing scalable code for heterogeneous systems. It is designed to provide a common solution to writing code for devices of all types and sizes - from embedded systems to many-core devices like GPUs [29]. Understanding the OpenCL Architecture [22] is critical to maximizing performance on a GPU. The four models - Platform, Execution, Memory and Programming - provide useful information to write code that will perform well. GPUs and CPUs differ greatly in terms of hardware design and implementation, so programming on a GPU cannot be done in the same way as traditional single-core CPUs. This chapter presents the four OpenCL Architecture models and attempts to highlight various optimizations that can be made to code written in OpenCL within the context of these models.

3.1 Platform Model

The OpenCL Platform Model [32] considers a Compute Device to be any device capable of running OpenCL and defines a host to be a machine connected to one or more Compute Devices. These Compute Devices make up the Computational Resources in OpenCL. Each Compute Device is made up of Compute Units, or cores, which are further divided into Processing Elements, or kernels, as seen in Figure 3.1. The most common Compute Devices are GPUs or FPGAs. An OpenCL application is code that runs on the host and sends commands to the Processing Elements within a Compute Device.

One major advantage of OpenCL is that one or several different devices connected to the host can be considered as a Compute Device and each one can be programmatically selected to run different parts of the code. Specifying more than one

device provides another dimension of parallelism; however the code becomes more difficult to write at this point. One way KLT could benefit from using multiple Compute Devices is in calculating image gradients in the x- and y-direction. Each compute device could focus on one direction and the resulting gradients could be derived in parallel.

Figure 3.1. The OpenCL Platform Model [22]

3.2 Execution Model

All OpenCL operations are specified in a C-style function called a kernel. All work in the kernel is performed by work-items, or in NVidia terminology, threads. This thesis uses the terms work-item and thread interchangeably to have the same meaning. Multiple work-items are combined to make up a work-group. Both work-items and work-groups are assigned a unique global ID, and further, work-items within a specific work-group are assigned a local ID. That is, work-items can either be identified globally amongst all other work-items, or locally within their respective work-group. This is important to note because uniquely identifying different work-items can facilitate sequential memory accesses, or memory coalescence. Coalesced memory access is another optimization technique that can greatly improve performance. This method is explained in further detail in Chapter 4.

The index space in which work-items in OpenCL operate is called the *NDRange*. An *NDRange* is an N-Dimensional index space where N ranges from one to three. This index range defines the total number of work-items that will execute a specific kernel. Figure 3.2 visualizes the relationship between an *NDRange*, work-groups and work-items. OpenCL also defines an optional local index space where the programmer can specify the size of each work-group. If this local index space is omitted, the OpenCL implementation will attempt to define the best work-group size on its own. This research has determined that it is usually acceptable to let OpenCL determine the local index space, however depending on the application and the data on which is being operated, there can be a slight performance increase by ensuring an optimal work-group size.

Figure 3.2 *NDRange* [19]

There are several OpenCL constructs that are required for any OpenCL application. A *cl_context* must be created which specifies on which device to run and also manages the resources on the device. All OpenCL work is performed within this

context. Next, a *cl_program* handles the C-style OpenCL kernel code by reading it in as a string. The program is then compiled at runtime by an OpenCL API call, passing in the device ID and program string. After the program is built the *cl_command_queue*, which is responsible for queuing tasks for the device, is instantiated. Once the command queue is created a *cl_kernel* is created to eventually run the OpenCL code. The OpenCL kernel *cl_kernel* is not to be confused with kernel definitions mentioned earlier which are C-style functions executed on the GPU in parallel by a large number of work-items in a Single Instruction Multiple Data (SIMD) fashion.

SIMD refers to a method of execution in which a number of threads execute the same instruction at the same time, but on different parts of data. On NVidia devices, a group of 32 threads is called a *warp* and each warp attempts to follow a SIMD execution flow. Problems can arise if there are many branches or conditional statements inside the kernel. This can “break” the SIMD execution if some threads in a warp take one path of a branch, and other threads of the same warp follow a different path. This is referred to as warp divergence. Because GPUs follow this SIMD execution model, GPUs are equipped with little or no branch prediction hardware like most CPUs [14]. As a result, it is crucial to write GPU kernels with as little branching as possible to maximize performance.

3.3 Memory Model

In OpenCL, each Compute Device has its own memory space which is broken down into global memory, local memory, constant memory and private memory (Figure 3.3). Global memory has the highest latency but can be accessed by every thread. When optimizing kernels, the number of global memory accesses should be minimized

because of its high latency. Local memory, or shared memory in CUDA, is faster than global memory but is only allocated and shared amongst work-items of the same work-group and is a much more scarce resource. Additionally, extra care must be taken when using local memory so that data dependencies are kept intact. Chapter 4 elaborates further on the potential pitfalls of using local memory in accordance with data dependencies. Constant memory is also extremely low latency and can be accessed by all work-items in similar fashion to global memory; however it is read-only. Lastly, private memory is scoped only to a single work-item. This is the fastest memory construct, but comes in a limited number. Each work-item is allocated, at most, the total number of registers available on the device divided by the number of threads launched. If the work-items' register usage exceeds the amount of private memory available, the values will spill over into global memory storage with slower access times.

Figure 3.3. The OpenCL Memory Model [22]

3.3.1 Image Data in OpenCL Memory Space

Texture memory is also available in OpenCL but is not pictured in Figure 3.3.

When specifying an image data type in OpenCL, it resides in its own memory space

called texture memory. Texture memory is optimized for storing image data on the device and, as a result, is much faster than global memory. The only setback is that, as of OpenCL 1.2, an image can only be specified as either read-only or write-only in the same kernel; it cannot be read-from and written-to. Still, using texture memory has its advantages.

First, when using texture memory for reading images, an *image_sampler* must be defined. The sampler is just a set of constants that tells the hardware whether or not to normalize coordinates, how to handle addressing an image beyond its bounds and to specify a filtering mode. In short, a sampler defines how to sample the image and return the desired color. Secondly, using texture memory is nearly as fast as using shared memory and could potentially be faster depending on how the image data is accessed. Lastly, by means of the *image_sampler*, values can be interpolated as they are read from texture memory, and boundary cases can be defined and handled without explicitly checking for these conditions.

The OpenCL implementation of KLT can take full advantage of the different memory spaces that are offered. For example, many of the overlapping pixel operations can first be stored into local memory for faster access instead of reading directly from global memory. In addition, image data can be stored in the optimized texture memory space to take advantage of interpolated values. Finally, specific read-only data structures can be directed to low-latency constant memory for faster access.

3.4 Programming Model

The OpenCL Programming Model supports two programming models: data parallel and task parallel, as well as a hybrid of the two. However, the major focus of

most GPGPU programming models as well as this research is that of the data parallel model.

In a Data Parallel Programming Model, each work-item is mapped to a data element in a one-to-one ratio. Work-items execute a sequence of instructions in parallel on different parts of the data in SIMD fashion as described in Section 3.2. A strict one-to-one ratio of work-items to data elements is not required with OpenCL since the programmer can specify total number of work-items as well as the size of a work-group. However, it is normally ideal to provide this one-to-one ratio to help ensure aligned memory accesses.

A Task Parallel Programming Model defines a model in which a kernel is launched and is void of any index space. In other words, a single compute unit creates one work-group with a single work-item executing the kernel code. In this model, it is difficult to express parallelism and take advantage of the massive execution capabilities of the device. However, this model exists for reasons such as queuing multiple kernels at once or using vector data types that are implemented and used efficiently by the device.

KLT is similar to many other image processing algorithms in that the Data Parallel Programming Model would likely be implemented instead of the Task Parallel Programming Model. Each pixel in an image can easily be mapped to a unique thread. This method of processing images using OpenCL is easy to implement, is usually extremely effective and is one way in which a programmer can fully utilize the resources that the Compute Device has to offer.

To recap, GPUs can easily launch tens of thousands of threads to execute in parallel with the help of APIs such as NVidia's CUDA or OpenCL, managed by the Khronos Group. These APIs are extensions to existing languages such as C and C++ and provide the programmer with the means to specify which code to run on the GPU, or device, and which code to run on the CPU, or host by means of programming a kernel. When the programmer calls the kernel, they must also specify the number of work-items, or threads, to launch as well as to provide the size of each work-group, or thread block. This tells the GPU to execute the kernel code in parallel with each work-item. In addition, any data structures to be used on the GPU must be allocated space in GPU memory and explicitly copied over to the GPU. Finally, once the kernel execution is finished, any resulting data that is required by the host for further processing must be explicitly copied back to host memory space. This overhead of copying data back and forth from host to GPU is extremely costly and is often the main bottleneck of GPGPU programs. As a result, copying data back and forth must be minimized in order to maximize efficiency. Often, it is better to execute a function on the GPU that may not be a good fit for GPU programming if it eliminates a data copy back to the host.

In this thesis, OpenCL was used to program portions of the KLT algorithm for acceleration. As aforementioned, not everything is ideal for GPU programming. Operations must be parallelizable and fairly time consuming on the host. This is because there is a certain overhead associated with creating and launching tens of thousands of threads and copying data from the host-side memory space to the device-side memory space and back. This overhead is unavoidable so any calculations to be performed on the device must be computationally intensive to achieve any performance gain.

So far, KLT has been discussed to expose its most computationally intense operations and show how these operations could be improved via data parallelization with the help of OpenCL. In the remaining chapters, the three most time-consuming operations in KLT are implemented using OpenCL and some interesting discoveries are explained. Once all GPGPU optimizations are exhausted, the runtime results of the new GPU-based KLT implementation are analyzed and any further improvements that could be made are considered.

Chapter 4 Implementation and Discoveries

The process for finding the biggest bottlenecks and long execution paths in code can typically be discovered using profiling tools. There are a number of profiling tools that can serve different purposes, however this thesis used Microsoft's profiling tool that accompanies their Visual Studio 2013 IDE and NVidia's Visual Profiler for examining GPU code. After profiling the code and studying the results, three functions consistently appeared in the analysis. These functions were image convolution, which produces the image gradients used in KLT, the Hessian calculation for selecting features and the function that implements Equations (3) and (6) in tracking the features. Since these methods are computationally intense and can be easily parallelized, they are good candidates for OpenCL implementation.

4.1 Image Convolution

Image convolution is a fundamental step in image processing. In general, it is applying a sliding window filter to an image, such that each pixel value is modified by a weighted sum of its neighboring pixel values [5] as in Figure 4.1. Many different filters can be applied to an image to produce different results, such as smoothing (or blurring) filters, sharpening filters or enhancing filters. In the case of KLT, the general consensus is to use image convolution to smooth the image using a Gaussian filter so that the salient features in the image can stand out and be easily recognized.

Figure 4.1. Image Convolution

Normally, convolution is performed bi-directionally in the x- and y-directions to produce the final result. The horizontal and vertical passes can be combined in a single nested for-loop using a two equivalent one-dimensional filters. However the computational complexity of this method is $O(N^2)$, where N is the number of pixels in the image. This can be reduced to $O(2N)$ by separating the horizontal and vertical passes into separate operations. This produces the same result since Gaussian convolution is associative, as long as the output of the first pass is used as the input in the next pass. Also, depending on the application, the boundary conditions for convolution can vary. When the sliding window filter gets to the edge of the image, the filter can extend beyond the image. The accepted protocol in this situation is to either truncate the out-of-bounds values to 0s or simply stop the convolution around the borders, such that the filter never reaches any out-of-bounds indices. In the KLT algorithm, the border values are simply zeroed out.

Two versions of the separable convolution method were implemented in OpenCL in the form of a naïve implementation and an optimized tiled implementation.

Both implementations launch one work-item per output pixel so that the entire image can be convolved in parallel, and both implementations store the convolution filter in constant memory. Since constant memory is read-only and shared among all threads, it is ideal for storing the filter.

The naïve version is a very basic and crude implementation that doesn't try to take advantage of the natural overlap in memory access that occurs when performing the operations in parallel. In short, it simply launches one work-item per pixel and convolves that pixel with its neighbors in the horizontal or vertical direction. The problem is that there is so much overlap between neighboring work-items when reading the pixel values. Each pixel value is being read numerous times by multiple work-items and this redundancy is inefficient. As a result, a tiled approach to reading each pixel value was introduced to reduce the number of reads being performed.

4.1.1 Tiled Image Convolution

Recall that in OpenCL, local memory refers to low latency memory that is shared among all work-items in a work-group. Therefore, consider if the image is separated into tiles that are the same size as a work-group. Each work-group can then operate on a separate tile from the input image where each work-item would read only one pixel value. This value can be stored in an array in local memory where each work-item has access to its neighbor's pixel value by reading from this high-speed local memory array [18].

Using the tiled approach creates an interesting scenario for boundary conditions. Here, boundary refers not to the image bounds, but rather the tile bounds. The OpenCL kernel becomes slightly more complex in order to handle the boundary

conditions in this tiled approach rather than in the naïve implementation. Since local memory is only accessible to work-items in the same work-group, some work-items must read values from neighboring tiles when the filter extends beyond its tile. These work-items must read an additional pixel value and are commonly referred to as *halo values*. Figure 4.2 illustrates tiled convolution for one tile. The green dashed box represents one tile and the purple rectangle is the convolution filter. Notice that the first threads in the tile must read the values from the previous tile, and subsequently the last threads in the row read from the next tile. These are the *halo values* that must be handled in the boundary conditions. Even with these additional *halo* reads from texture memory, the total number of texture memory reads is 5.75x fewer than the naïve implementation.

Figure 4.2. Tiled Image Convolution

Porting the naïve convolution filtering to OpenCL decreased runtime for convolution by 99% over the CPU-based implementation and optimizing the naïve version by executing the tiled version further reduced the runtime. The tiled approach was an important optimization because the convolution function is called three times

per image that is processed: once to smooth the original image and twice more to produce the x- and y-gradients.

4.1.2 Data Dependencies

Much was learned by implementing the tiled image convolution approach. The tiled approach took advantage of local memory that is shared amongst the work-items in a work-group and also exhibited data dependency handling. When utilizing local memory in the kernel functions, programmers must be cautious of data dependencies. A data dependency is when a read and write operation occur on the same data, often within a loop. There are three types of data dependencies which can cause problems in any programming language, not just OpenCL. They are true-data dependencies, anti-data dependencies and output-data dependencies.

True-data dependencies describe a read-after-write (RAW) dependency, where some variable is read after it has just been written to. Anti-data dependencies describe a write-after-read (WAR) dependency and output-data dependencies are for write-after-write (WAW) operations. In many-core programming environments where thousands of threads are executing at once, extra care must be taken to ensure these data dependencies are not violated. OpenCL provides an API call to place *barriers* in the code which act as stall points to synchronize execution across work-items. Once a thread has reached a barrier it cannot proceed until all threads have reached that point. This is necessary for the true-data dependencies that occur in the image convolution kernel by ensuring a shared value has finished being written prior to the next read operation.

4.2 Hessian Calculation

By far the longest executing operation in this KLT implementation is the Hessian calculation. The Hessian involves processing image gradients in the x- and y-directions, performing an Eigen decomposition on thousands of pixel windows and performing a sort which will ultimately decide on the most prominent features in the image. But with the help of OpenCL the runtime of this function can be reduced drastically.

Even with this reduction in execution time the Hessian is still a long-executing operation. One alternative to executing the computationally expensive Eigen decomposition is to use an approximate corner response known as Harris corner detection [11]. The Harris corner detector works by taking the determinant of the 2x2 matrix from Equation 5. This method avoids the full Eigen decomposition and is also a much better fit for GPGPU implementation as it is easier to parallelize. Conversely, a full Eigen decomposition will be fairly accurate when finding corners, whereas the Harris detector will tend to find more edges. Often times with GPGPU implementations, there is a tradeoff between speed and accuracy. Depending on the application, using the Harris detector to approximate the Hessian may be more acceptable if speed is a priority over accuracy.

4.2.1 Bank Conflicts

One reason why the Hessian computation is still slow even in OpenCL is that it suffers from memory bank conflicts. A bank conflict occurs when using local memory and two or more threads attempt to read or write from the same bank. This forces the data access to become serialized because each memory bank can only address data sequentially. Depending on the graphics card used, there are 16 or 32 memory banks

and each one is responsible for 32 bits. For example, bank 0 handles bytes 0-3; bank 1 handles bytes 4-7 and so on. Knowing this, bank conflicts can be avoided on most occasions. The Hessian is not one of these occasions and the bank conflicts cause the Hessian to run 60% slower when utilizing local memory on the GPU. An attempt was made to modify the OpenCL kernel to reduce bank conflicts, but this introduced a larger number of divergent statements using if/else conditionals. At this point, the control flow within the OpenCL kernel must be considered.

4.2.2 Control Flow Optimizations

Control flow divergence is when threads of the same warp take different execution paths. Having a divergent control flow can drastically reduce the performance gain of an OpenCL-enabled device. Conditional statements such as “if” and “switch” often suggest there will be some warp divergence and sometimes it can be unavoidable. However if the condition is based on the global or local thread ID, there exists some cases where divergence can be eliminated. For example, if the condition applies to every 32 thread IDs, there will be no warp divergence because an entire warp will take one of the varying execution paths. However, if for instance, the condition applies to even number thread IDs as shown in Figure 4.3, the warp will diverge with all even numbered threads executing some path, and the rest executing a different path. When this happens the execution within the warp becomes serial, breaking the SIMD execution model mentioned in Section 3.2 until the warp converges again after the conditional statement.

Figure 4.3. Warp Divergence

4.2.3 Memory Coalescence

Another cause of the slow Hessian calculation in OpenCL was due to uncoalesced memory accesses. Memory coalescence is when memory is accessed in contiguous locations by sequential threads in a warp as shown on the left diagram in Figure 4.4. When this happens, the hardware is able to optimize the memory throughput resulting in fewer loads from memory. Satish et al. explain how a warp of 32 threads reading from non-coalesced addresses will be forced to issue 32 requests from memory, while a warp of 32 threads reading 32 consecutive words will only issue 2 requests [24].

Figure 4.4. Coalesced Memory Access vs Non-Coalesced Memory Access

When calculating the Hessian, it accesses image data from neighboring rows. This access pattern requires neighboring work-items to access data with large gaps in-

between. This is a huge inefficiency due to constant cache misses and contributes to the Hessian's slow runtime.

4.3 Tracking Features

The feature tracking kernel implementation was one of the more lengthy OpenCL operations. This is due to the sheer amount of data that must be processed when tracking a large number of features between two images. In addition to the two images, their gradients in the x- and y-directions are also considered. Even further, since this KLT version implements Gaussian pyramids to assist in tracking, each of these images and gradients are down sampled by half resolution and also processed by the tracking kernel.

So far, a typical OpenCL implementation involves some singular datatype (usually an image or large array) that is processed in parallel by a large number of threads. This methodology does not hold in the feature tracking OpenCL kernel because operating on one image at a time would augment the bottleneck associated with data transfer from CPU-space to GPU-space. It was realized that a change was necessary to handle tracking over 500 features in a sequence of images. Instead, the feature tracking kernel launches one thread per feature and each thread works in parallel to perform the computation in Equations (3) and (6) iteratively until convergence below the user-defined threshold is satisfied. In this manner, data transfers are greatly reduced since all the data is transferred and read once initially and kept in the device memory space throughout the tracking process. Additionally, the complexity of the feature tracking kernel imposes additional restrictions on the number of work-items that can track

features concurrently. This section addresses these limitations to provide maximum performance on the GPU.

4.3.1 Reducing Data Transfers

As aforementioned in this research, data transfers from CPU-to-GPU and vice versa are by far the largest bottleneck in GPGPU programming because there is so much overhead involved in performing the copy. Also, typical PCIe speeds cannot compare favorably to device global memory access speeds. It is imperative to reduce these transfers as much as possible to achieve the highest performance gain. One optimization that is easy to implement and can make drastic improvements is to group many small, separate transfers into a larger, single transfer. Doing this increased the data throughput by 52% in the entire program. Recall data throughput is the number of bytes/second transferred from host-to-device, device-to-host or device-to-device. Simply put, the higher the throughput, the better the performance.

4.3.2 Interpolation

One of the first steps in tracking features is to compute an image difference between the reference and template images. Since the image warp is a non-linear operation, interpolation must be performed to get an accurate measure of a pixel's color value. OpenCL offers a "free" implementation of bilinear interpolation by specifying a certain flag when declaring an *image sampler*. In OpenCL, an *image sampler* is a structure that handles querying an image for a specific pixel value and it can additionally handle potential errors such as querying outside the image bounds. *Image samplers* were discussed earlier in Chapter 3.3.1. Per the OpenCL 1.x spec, bilinear interpolation selects elements by way of the following equations:

$$\begin{aligned}
i0 &= \text{addressMode}((\text{int})\text{floor}(u - 0.5)), \\
j0 &= \text{addressMode}((\text{int})\text{floor}(v - 0.5)), \\
i1 &= \text{addressMode}((\text{int})\text{floor}(u - 0.5) + 1), \\
j1 &= \text{addressMode}((\text{int})\text{floor}(v - 0.5) + 1), \\
a &= \text{frac}(u - 0.5), \\
b &= \text{frac}(v - 0.5),
\end{aligned}$$

where u and v are the pixel locations and $\text{frac}(x)$ defines the fractional part of x and equals $x - \text{floor}(x)$. Then, the image element's value is determined by bilinearly interpolating using:

$$\begin{aligned}
T &= (1 - a) * (1 - b) * T_{i0j0} + a * (1 - b) * T_{i1j0} + (1 - a) * b \\
&\quad * T_{i0j1} + a * b * T_{i1j1},
\end{aligned} \tag{14}$$

where T_{ij} is the image element at location (i, j) . The above methods subtract 0.5 from the u and v values because in OpenGL and OpenCL, pixels are located at the “center” of the area they occupy, whereas in other APIs, pixels are located in the upper-left corner. The 0.5 offset is there to bridge the two, thereby getting the $\text{floor}()$ and/or $\text{ceil}()$ of a value.

While studying the *image sampler's* interpolation, it was discovered that there could be potential inaccuracies with this method. Paraphrasing Section 8.2 of the OpenCL 1.x Specification [22], interpolating via the *image sampler* is not guaranteed to produce the precise pixel value represented by the given image coordinates. However, using an *image sampler* to simply query the image without interpolating will produce the precise pixel value and is the recommended usage of image sampling. With this

knowledge, a custom interpolation method was written in OpenCL and the results compared against the *image sampler's* interpolation. The results are explained further in Chapter 5, but in short, the *image sampler's* interpolation was faster and less accurate than the custom interpolation, although the inaccuracies observed did not significantly inhibit the feature tracker while testing several different datasets.

4.3.3 NDRange Optimizations

Chapter 3.2 defines an *NDRange* to be the N-Dimensional index space in which work-items are specified and launched when executing an OpenCL kernel. Specifying an *NDRange* to maximize performance can be a challenge given the OpenCL kernel implementation and the GPU hardware specifications. There are limiting factors with each that contribute to the performance gain of the kernel. This section goes into further detail and aims to show that both must be considered when optimizing GPU code.

The complexity of the feature tracking kernel can inhibit performance gain due to insufficient resources available. Recall that registers are the lowest latency memory type but are also a scarce resource on the GPU. If a kernel requires the use of n registers and there are m work-items created, the multiprocessor should have a minimum of $n*m$ registers. Exceeding this will force the GPU to spill variable storage into higher latency global memory, reducing performance. For example, a device with OpenCL compute capability 1.1 has 8,192 32-bit registers allocated per multiprocessor and a maximum of 768 simultaneous work-items per multiprocessor. This leaves each thread only 10 registers to use to maintain 100% thread occupancy [4]. If just one more register is used by each thread, occupancy is reduced by 33%.

GPU occupancy is a term for the ratio of the number of active warps per multiprocessor to the maximum number of allowed active warps [4]. This ratio should be large enough so that each multiprocessor has at least one work-group to execute, and ideally there should be multiple work-groups per multiprocessor so that in the case of any stalls within a work-group, a multiprocessor can remain busy by swapping execution to a different work-group. The occupancy can be estimated by specifying the work-group size which will give the GPU the number of work-items per work-group. There is a caveat, however, in that higher occupancy does not always imply higher performance. But it can be assumed that low occupancy always returns suboptimal performance with the inability to hide latency.

In addition to having enough work-groups to occupy the multiprocessors, the size of each work-group is a critical factor in *NDRange* optimization. A GPU can execute multiple work-groups concurrently on a single multiprocessor. This means that simply specifying a larger work-group size may not be optimal. For example, for a given device with a maximum number of work-items per multiprocessor of 768, the ideal work-group size is 256, not 512. With 512 work-items per work-group, there can only be one work-group per multiprocessor, effectively occupying only 66% of the multiprocessor. In contrast the work-group size of 256 will result in 100% occupancy by creating three work-groups per multiprocessor.

Keeping this limitation in mind, the track feature kernel's *NDRange* was specified by means of trial and error. Different ranges were specified and their runtimes logged, proving that sometimes the best practice is to experiment with various parameters to find the optimal results. This is the case with specifying *NDRanges* more

often than not. Also, running the code on different GPUs will almost certainly force different *NDRanges* as various GPUs will have different amounts of shared memory, max work-group sizes and max work-items per multiprocessor. Fortunately, NVidia provides an “Occupancy Calculator” with their SDK which attempts to help find the optimal *NDRange* given certain inputs, like register usage and shared memory usage. This Occupancy Calculator was also used to help determine the optimal *NDRanges* in this research.

Chapter 5 Results & Analysis

All results presented in this chapter were gathered on a Dell XPS tower with an Intel Core i7-4770 CPU clocked at 3.4 GHz and 16 GB of RAM running Windows 7 Ultimate 64-bit operating system. The GPU on the system was an NVidia GeForce GTX 780 which has 12 Streaming Multiprocessors, 2304 CUDA cores and 3 GB of memory. The OpenCL 1.2 SDK was used to compile and run the OpenCL kernels. The experiments were conducted against two video datasets, one in standard definition resolution (SD, 720x480) and the other in high definition resolution (HD, 1920x1080). The content and size of the SD and HD datasets were different, with the SD dataset being composed of 2000 frames and the HD dataset being composed of 600 frames.

The following data presented compares the original CPU-only code with the GPU-accelerated code. The GPU-accelerated code contains the image convolution, Hessian and feature tracking operations implemented using OpenCL. The timing of these functions and overall runtime are observed as well as the accuracy between the CPU and GPU versions. The accuracy was measured by calculating the Root Mean Squared Error (RMSE) value for each feature coordinate location in each frame. Figure 5.1 shows the overall speedup of the GPU version over the CPU version for an SD and HD dataset. Also note that three variations of the GPU implementation are shown: one using interpolation provided by the *image_sampler* (denoted “GPU Sampler Interp”), another was a custom interpolation method which required manual invocation (denoted “GPU Custom Interp”), and lastly one implementation using the custom interpolation method but employing the Harris detector (denoted “GPU Hessian Approx”). Recall from Chapter 4 that the *image_sampler* interpolation had potential for inaccuracies when querying for a pixel value. The custom method was guaranteed to be

accurate when querying an image for a pixel value. Each of the GPU implementations provided a 92% reduction in runtime for the HD sequence and an 82% reduction in runtime for the SD sequence, with the Custom Interpolation method slightly outperforming the Sampler Interpolation method in terms of speed.

Figure 5.1. KLT Runtime, CPU vs GPU

5.1 Performance

Recall Amdahl’s Law from Chapter 2, which attempts to estimate the theoretical speedup of an algorithm given the percentage of parallelizable code. Figure 5.2 gives a representation of the percentage of runtime occupied by the three optimized functions. With 90% of the algorithm parallelized, there is a 10x theoretical speedup that can be realized. The 92% speedup in the OpenCL implementation is very close to Amdahl’s Law’s prediction but shows there is still some room for further improvement within the OpenCL kernels.

Figure 5.2. KLT Functions as a Percentage of Total Runtime

Figure 5.3 shows the average time taken for a single instance of each of the three functions from KLT that experienced GPU acceleration. The corresponding CPU times are also available. Note that the y-axis is logarithmic since the GPU times were so small compared to the CPU times. The convolution and Hessian operations experienced a 99% reduction in runtime while the track features function realized a 98% reduction.

Figure 5.3. Average Function Runtimes

From the “Hessian” column in Figure 5.3, the purple bar indicates the Hessian approximation which is simply taking the determinant of the 2x2 Hessian matrix while the other bars perform the full Eigen decomposition. Also, looking at the “Track” column in Figure 5.3 above, it is interesting that the GPU Custom Interpolation method takes longer to execute on average than the other GPU method which implements interpolation via the *image sampler*, but in Figure 5.1 the Custom Interpolation method is faster overall. Both the Hessian approximation and the *image sampler* are less accurate, implying that more features will fail the tracking step and become flagged as lost. When features are lost, they must be replaced, invoking the Hessian operation more often which is the longest executing operation. In this case, accuracy does impact speed. Table 5.1 confirms this suspicion by verifying that on average, the Hessian approximation and *image sampler* interpolation methods lose more features over a period of time and are actually slower overall on the HD dataset. Additionally, the

custom interpolation method tracks as many features on average per frame as the CPU-only code. The maximum number of features tracked per frame is 512.

Implementation	Lost Feature Count	Avg Features Tracked/Frame	Avg FPS
CPU	4106	505	0.81
GPU Sampler Interpolation	4269	504	9.95
GPU Custom Interpolation	4179	505	10.02
GPU Hessian Approximation	9645	495	9.78

Table 5.1. Lost Feature Count and Average Frames per Second, HD Dataset

Another interesting note from Table 5.1 could be made regarding the average FPS from the CPU implementation. The average rate is below one frame per second implying that as time goes on, or as the size of the dataset increases, the overall execution time will become greater, further increasing the gap between the GPU and CPU versions. This point is illustrated in Figure 5.4.

Figure 5.4. Total Runtime with Varying Dataset Sizes, HD

Table 5.2 shows that the average frames per second are correlated to the overall runtime performance. This indicates that the performance of the algorithm and therefore the optimized functions is fairly consistent and does not waver over time.

Implementation	Dataset	# Frames	Overall Runtime (s)	Average FPS
CPU	SD	2000	368.26	5.45
GPU Sampler Interp	SD	2000	66.68	30.37
GPU Custom Interp	SD	2000	66.69	30.37
CPU	HD	600	761.64	0.81
GPU Sampler Interp	HD	600	60.14	9.95
GPU Custom Interp	HD	600	59.39	10.02

Table 5.2. Average Frames per Second Based on Image Resolution

5.2 Accuracy

To measure accuracy in the results, Root Mean Square Error (RMSE) was calculated for each feature position for each frame in the video sequence. The three

GPU implementations – custom interpolation, sampler interpolation and Hessian approximation – were measured against the CPU-based results since the GPU optimization that took place in this study was meant to keep the accuracy of tracking features as close to the original code as possible. Results were gathered for both standard definition and high definition video sequences. Figure 5.5 lists the RMSE values for each feature over the first 250 frames of the SD video sequence and Figure 5.6 shows the RMSE values during the first 250 frames of the HD video sequence. For the GPU values, the closer they align to the CPU blue line, the better the accuracy. However, for majority of the frames both GPU interpolation methods are indistinguishable. The Hessian approximation is noticeably different and deviates beyond any acceptable variance in terms of accuracy. This showcases the speed versus accuracy tradeoff since the Hessian approximation is much faster than the other GPU implementations.

Figure 5.5. Root-Mean-Squared Error, SD

Figure 5.6. Root-Mean-Squared Error, HD

5.3 Profiling

OpenCL provides functions and data structures to programmatically profile kernels, data transfers and object read/writes. These functions return timing results that can then be written to a file or used elsewhere in a program. Conversely, if using an NVidia device for OpenCL, a tool called the Visual Profiler is included in their SDK which provides a visual timeline of CUDA (or OpenCL) kernel and API calls as well as additional information regarding work-items, grid size and memory transfer and throughput.

Profiling is an important step in optimizing code. It can reveal latent functions, time-consuming functions and bottlenecks, memory misses and expensive execution paths. In this research, the NVidia Visual Profiler was used to determine kernel execution times, optimal work-group and grid sizes, data transfer times and thread

occupancy ratios. These statistics were compared to the statistics produced by programmatically profiling the kernel functions using the OpenCL API calls, and they were found to be the same, so the Visual Profiler was used throughout for simplicity.

Figure 5.7 shows an example of the Visual Profiler in action. In this particular case, the profiler helped increase throughput by visualizing several small data transfers, highlighted by the red bounding box. Recall from Chapter 4 that grouping many small transfers into a single larger transfer can increase throughput by reducing the overhead associated with data transfers. Table 5.3 showcases the increase in throughput after combining the smaller transfers into a single larger transfer.

Figure 5.7. The NVidia Visual Profiler

Throughput Before Combining Transfers	Throughput After Combining Transfers	Percent Increase
8.025 GB/s	12.230 GB/s	52.4%

Table 5.3. Total Program Throughput

The important note to take away from using the Visual Profiler is that since there is no real means to debug OpenCL code, the information provided by such tools is critical to evaluating kernel code. Even *printf* statements are not possible within OpenCL kernels so optimizing code and ensuring correct results are that much more difficult. Finally, the Visual Profiler also provides valuable feedback when experimenting with various work-group sizes. This information helped give a sense of GPU occupancy rates and where more (or fewer) work-items could be spared.

Chapter 6 Future Work & Conclusions

This thesis has shown that OpenCL and GPUs can provide vast increases in speed and performance when tracking many features on an HD dataset using KLT. Still, there are numerous improvements that could further increase speed to achieve even greater performance. Additional functions from KLT could be moved to the GPU for processing, OpenCL kernels could be rewritten to minimize global memory accesses, and it's possible to realize even greater improvements by executing the current implementation on better hardware. Lastly, some enhancements to OpenCL could make programming easier and more efficient.

6.1 Future Work

One of the biggest bottlenecks remaining on the host side of the OpenCL KLT implementation is sorting the eigenvalues. Sorting is a difficult process to parallelize efficiently, but if done correctly, could remove this lengthy operation from the CPU. Also, the Gaussian filter used for convolution is calculated on the CPU. Although this isn't a lengthy derivation, it could be calculated on the GPU instead to eliminate an additional costly data transfer and any associated overhead.

Of the three functions implemented in OpenCL, the Hessian calculation is the most computationally intensive among them, as verified in Chapter 5. There is much overlap in memory access associated with the Hessian, and the current Hessian OpenCL kernel could be implemented more efficiently to take advantage of this overlap and utilize low-latency shared memory. Likewise, the feature tracking kernel has some room for improvement, however this is much more difficult to improve since it is a more

complex kernel and uses more resources. As a result, fewer threads can be launched which is more a limitation of the current hardware than the software implementation.

Hardware plays an important role in GPGPU programming. The hardware selected can force limitations when designing and writing the software and can also influence the type of parallelism exploited by OpenCL. The GeForce GTX 780 is already several years old at the time of this research. To further improve runtimes, a faster GPU with more shared memory or room for more concurrently-executing work-items could be tested. Finally, since OpenCL claims their API is somewhat generic and portable to other devices besides just GPUs, this version of KLT could be tested on a device such as an FPGA. FPGAs employ a different style of parallelism than GPUs, but one that could improve speed in new areas and expose other bottlenecks not apparent in GPU implementations.

Despite the successes in this research, the full potential of OpenCL and GPUs was not in the spotlight. Compute Devices can easily handle more data to process, reducing the load of the CPU for other tasks. Also, OpenCL allows for multiple Compute Devices, for which this research only used one, opening the door to yet another level of parallelism. Even more, the portability of OpenCL to run on different devices implies higher scalability of code which is also invaluable. Lastly, state-of-the-art GPU technology would provide an even further increase in performance with larger cache sizes and shared memory, more work-items executing per Compute Unit and higher throughput.

6.2 Conclusion

To conclude, this thesis has demonstrated that OpenCL implementations of computationally intense algorithms executed on a GPU can realize significant increase in performance and reduced runtime. Additionally, GPUs can maintain similar accuracy when performing single precision arithmetic compared to CPUs. With these results, GPUs can easily assist in processing high resolution imagery to produce faster results.

This thesis has evaluated many methods and optimizations of GPGPU programming. Among them, the most important principle to follow when writing a GPGPU program is to find parallelization within sequential code because GPUs mainly provide performance increase by way of parallelization. It is true that some operations are difficult if not impossible to parallelize. However, most algorithms can be re-worked to reveal some parallelism. This is the most challenging problem facing GPGPU programming and can be difficult to implement.

Another important lesson was the reality of a speed versus accuracy tradeoff when using OpenCL. That tradeoff exists and is one which must be determined empirically. Likewise, there is a fine line in determining the number of threads to launch for a given OpenCL kernel. The right number is found by staying within the bounds of the shared resources offered and maximizing GPU occupancy. Over and above determining the work-group size, the most valuable optimization technique to this research was minimizing data transfer from CPU to GPU and vice versa. It appears to be a simple rule to follow but it can be increasingly challenging to implement. Finding ways to reuse data already on the device, or simply re-evaluating some process because it is more expensive to instantiate an additional data transfer is a difficult but necessary

task. The largest performance increase gained when optimizing the OpenCL code was witnessed by reducing data transfers and reusing data already on the device.

In addition to applying these GPGPU optimizations and understanding the OpenCL Architecture, the KLT feature tracking algorithm was studied and analyzed to exploit parallelism that could benefit from GPU acceleration. In this way, a 92% reduction in runtime was achieved when tracking over 500 features in an HD video sequence. With GPU hardware improving each year and researchers experimenting with new devices like integrated CPU-GPU chips, it is an exciting time to study the emerging and dynamic field of GPGPU Programming.

References

- [1] Adelson, E. H., C. H. Anderson, J. R. Bergen, P. J. Burt, and J. M. Ogden. "Pyramid Methods in Image Processing." *RCA Engineer* 29, no. 6 (November 1984): 33-41.
- [2] Baker, Simon, and Iain Matthews. "Lucas-Kanade 20 Years On: A Unifying Framework." *International Journal of Computer Vision* 56, no. 3 (2004): 221-55.
- [3] Birchfield, Stan. "KLT: An Implementation of the Kanade-Lucas-Tomasi Feature Tracker." KLT: Kanade-Lucas-Tomasi Feature Tracker. Accessed July 2015.
<http://www.ces.clemson.edu/~stb/klt>.
- [4] Carnegie Mellon University. "OpenCL Best Practices Guide." Accessed October, 2015. www.cs.cmu.edu/afs/cs/academic/class/15668-s11/www/cuda-doc/OpenCL_Best_Practices_Guide.pdf
- [5] Cornell University. "CS1114 Section 6: Convolution." Lecture. Accessed December, 2015.
http://www.cs.cornell.edu/courses/cs1114/2013sp/sections/s06_convolution.pdf.
- [6] Domínguez, Jose M., Alejandro J.c. Crespo, and Moncho Gómez-Gesteira. "Optimization Strategies for CPU and GPU Implementations of a Smoothed Particle Hydrodynamics Method." *Computer Physics Communications* 184, no. 3 (2013): 617-27.
- [7] Dowson, N., and R. Bowden. "Mutual Information for Lucas-Kanade Tracking (MILK): An Inverse Compositional Formulation." *IEEE Transactions on Pattern Analysis and Machine Intelligence IEEE Trans. Pattern Anal. Mach. Intell.* 30, no. 1 (2008): 180-85.
- [8] Farias, Thiago, João M. Teixeira, Gabriel Almeida, Pedro Leite, Veronica Teichrieb, and Judith Kelner. *A CUDA-enabled KLT Tracker for High Definition Images*. Technical paper. 2009.
- [9] Fassold, Hannes, Jakub Rosner, Peter Schallauer, and Werner Bailer. *Realtime KLT Feature Point Tracking for High Definition Video*. Technical paper. 2009.
- [10] Friedman, Daniel P., Mitchell Wand, and Christopher Thomas. Haynes. *Essentials of Programming Languages*. MIT Press, 2001.
- [11] Harris C., Stephens M. "A Combined Corner and Edge Detector." *Proceedings of the 4th Alvey Vision Conference*, 1988.
- [12] Hestness, Joel, Stephen W. Keckler, and David A. Wood. "GPU Computing Pipeline Inefficiencies and Optimization Opportunities in Heterogeneous CPU-GPU Processors." *2015 IEEE International Symposium on Workload Characterization*, 2015.

- [13] Hill, Mark D., and Michael R. Marty. "Amdahl's Law in the Multicore Era." *Computer* 41, no. 7 (2008): 33-38.
- [14] Jablin, James A., Thomas B. Jablin, Onur Mutlu, and Maurice Herlihy. "Warp-aware Trace Scheduling for GPUs." *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation - PACT '14*, 2014.
- [15] Jackovitz, Kevin. *Integrated Coarse to Fine and Shot Break Detection Approach for Fast and Efficient Registration of Ariel Image Sequences*. Master's thesis, University of Dayton, 2013.
- [16] Jog, Adwait, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. "Owl: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance." *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '13*, 2013.
- [17] Kim, Jun-Sik, Myung Hwangbo, and Takeo Kanade. "Realtime Affine-photometric KLT Feature Tracker on GPU in CUDA Framework." *2009 IEEE 12th International Conference on Computer Vision Workshops, ICCV Workshops*, 2009.
- [18] Kirk, David, and Wen-mei Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Burlington, MA: Morgan Kaufmann Publishers, 2010.
- [19] Kyriazis, George. "Achieving Frictionless Parallel Processing with the Heterogeneous System Architecture." *RTC Magazine*. AMD, May 2014. Accessed March 2016.
- [20] Mcdermott, Josh, Yair Weiss, and Edward H. Adelson. "Beyond Junctions: Nonlocal Form Constraints on Motion Interpretation." *Perception* 30, no. 8 (2001): 905-23.
- [21] Moulik, S., and W. Boonn. "The Role of GPU Computing in Medical Image Analysis and Visualization." *Medical Imaging 2011: Advanced PACS-based Imaging Informatics and Therapeutic Applications*, 2011.
- [22] Munshi, Aaftab. *OpenCL 1.2 Specification*. PDF. Khronos OpenCL Working Group, November 14, 2012. Version 1.2, Document Revision 19
- [23] Ryoo, Shane, Christopher I. Rodrigues, Sam S. Stone, Sara S. Bagsorkhi, Sain-Zee Ueng, John A. Stratton, and Wen-Mei W. Hwu. "Program Optimization Space Pruning for a Multithreaded Gpu." *Proceedings of the Sixth Annual IEEE/ACM International Symposium on Code Generation and Optimization - CGO '08*, 2008.

- [24] Satish, Nadathur, Mark Harris, and Michael Garland. "Designing Efficient Sorting Algorithms for Manycore GPUs." *2009 IEEE International Symposium on Parallel & Distributed Processing*, 2009.
- [25] Scarpino, Matthew. *OpenCL in Action: How to Accelerate Graphics and Computation*. Shelter Island, NY: Manning, 2012.
- [26] Shi, Jianbo, and Tomasi. "Good Features to Track." *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition CVPR-94*, 1994.
- [27] Thompson, C.j., Sahngyun Hahn, and M. Oskin. "Using Modern Graphics Architectures for General-purpose Computing: A Framework and Analysis." *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings*.
- [28] Toennies, Klaus D. *Guide to Medical Image Analysis: Methods and Algorithms*. London: Springer, 2012.
- [29] Tompson, Jonathan, and Kristofer Schlachter. "An Introduction to the OpenCL Programming Model."
- [30] Villa, Oreste, Daniel R. Johnson, Mike Oconnor, Evgeny Bolotin, David Nellans, Justin Luitjens, Nikolai Sakharnykh, Peng Wang, Paulius Micikevicius, Anthony Scudiero, Stephen W. Keckler, and William J. Dally. "Scaling the Power Wall: A Path to Exascale." *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014.
- [31] Wagener, Dirk W., and Ben Herbst. "Face Tracking: An Implementation of the Kanade-Lucas-Tomasi Tracking Algorithm." July 10, 2014. Accessed November 2015.
- [32] Woolley, Cliff. "Introduction to OpenCL." Lecture. Accessed September 2015. www.cc.gatech.edu/~vetter/keeneland/tutorial-2011-04-14/06-intro_to_opengl.pdf.
- [33] Zach, Christopher, David Gallup, and Jan-Michael Frahm. "Fast Gain-adaptive KLT Tracking on the GPU." *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, 2008.