7-2014

Accelerated Object Tracking with Local Binary Features

Breton Lawrence Minnehan

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation

This Thesis is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
Accelerated Object Tracking with Local Binary Features

by

Breton Lawrence Minnehan

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Engineering

Supervised by

Dr. Andreas Savakis
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, NY
July 2014

Approved By:

Dr. Andreas Savakis
Primary Advisor – R.I.T. Dept. of Computer Engineering

Dr. Raymond Ptucha
Secondary Advisor – R.I.T. Dept. of Computer Engineering

Dr. Roy Melton
Secondary Advisor – R.I.T. Dept. of Computer Engineering
Dedication

I would like to dedicate this thesis to Robert Woolner, who first introduced me to wonders of computers. A very special thanks to my parents, Peter and Paula Minnehan, and sister, Kaitlin Minnehan, I am forever grateful for your unwavering support.
Acknowledgements

I would like to thank my advisor Dr. Andreas Savakis for his assistance and mentorship in the process of developing this thesis. I would also like to thank my committee members Dr. Raymond Ptucha and Dr. Roy Melton. Additionally I want to thank my colleagues in the Computer Vision Lab, Henry Spang, Ethan Peters, and Riti Sharma, who were an invaluable resource throughout this process.
Abstract

Multi-object tracking is a problem with wide application in modern computing. Object tracking is leveraged in areas such as human computer interaction, autonomous vehicle navigation, and panorama generation, as well as countless other robotic applications. Several trackers have demonstrated favorable results for tracking of single objects. However, modern object trackers must make significant tradeoffs in order to accommodate multiple objects while maintaining real-time performance. These tradeoffs include sacrifices in robustness and accuracy that adversely affect the results.

This thesis details the design and multiple implementations of an object tracker that is focused on computational efficiency. The computational efficiency of the tracker is achieved through use of local binary descriptors in a template matching approach. Candidate templates are matched to a dictionary composed of both static and dynamic templates to allow for variation in the appearance of the object while minimizing the potential for drift in the tracker. Locality constraints have been used to reduce tracking jitter. Due to the significant promise for parallelization, the tracking algorithm was implemented on the Graphics Processing Unit (GPU) using the CUDA API. The tracker's efficiency also led to its implementation on a mobile platform as one of the mobile trackers that can accurately track at faster than realtime speed. Benchmarks were performed to compare the proposed tracker to state of the art trackers on a wide range of standard test videos. The tracker implemented in this work has demonstrated up to double the accuracy of other trackers the while operating several orders of magnitude faster.
# Table of Contents

Chapter 1  Introduction .............................................................................. 1

Chapter 2  Background ............................................................................. 3

2.1.  Object Tracking ............................................................................ 3

2.1.1  KLT Tracker ............................................................................. 4

2.1.2  CAMSHIFT Tracker ................................................................. 6

2.1.3  Particle Filter Tracking ............................................................ 8

2.1.4  Tracking by Classification and Detection .................................. 8

2.1.5  Template Trackers ................................................................. 11

2.2.  GPU Acceleration ........................................................................ 16

2.2.1  General Purpose GPU Computing ......................................... 16

2.2.2  GPU accelerated Object Trackers ......................................... 21

2.3.  Descriptors .................................................................................. 27

2.3.1  SIFT ..................................................................................... 27

2.3.2  SURF .................................................................................... 29

2.3.3  LBP ...................................................................................... 29

2.3.4  BRIEF and ORB ................................................................. 30

2.3.5  BRISK ................................................................................... 31

2.3.6  FREAK .................................................................................. 32

Chapter 3  Tracking with Local Binary Descriptors .............................. 33
# 3. Algorithm Design

## 3.1. Single Object Tracking

## 3.2. Multi-Part Object Tracking

## 3.3. Selecting Local Binary Descriptors

### 3.3.1. Simplified BRISK descriptor

## 3.4. GPU Implementation

## 3.5. Mobile Implementation

# 4. Experimental Results

## 4.1. Experimental Setup

### 4.1.1. Tracking Source

### 4.1.2. Initialization

### 4.1.3. Datasets used

## 4.2. Parameter Selection

## 4.3. Accuracy Tests

### 4.3.1. Accuracy Comparisons between Trackers

### 4.3.2. Examples of well tracked videos

### 4.3.3. Examples of poorly tracked videos

## 4.4. Speed tests

### 4.4.1. Speed Test Configuration

### 4.4.2. Single Object Tracking Speed Tests
4.4.3 Multiple Object Tracking Speed Tests ........................................ 83
4.4.4 Mobile Tracking Speed Tests .................................................... 93
4.5. Parts Tracking Tests ............................................................... 95

Chapter 5 Conclusions and Future Work ....................................... 97
List of Figures

Figure 1: DML candidate search pattern .......................................................... 15
Figure 2: LBP Descriptor Example................................................................. 30
Figure 3: The BRISK Descriptor Pattern Visualized on a Test Video Frame .......... 32
Figure 4: Single object algorithm flow .......................................................... 34
Figure 5: Dense Candidate Grid .................................................................. 38
Figure 6: Fine to Course Candidate Grid ....................................................... 38
Figure 7: Locality Weighting Function ........................................................... 40
Figure 8: Example Frames from a Video of a Blink ........................................ 43
Figure 9: Plot of the Percent Difference of Descriptors over Blinking Sequence. A Blink Occurs from Frame 8 to Frame 12 ......................................................... 43
Figure 10: Object and Parts Algorithm Flow .................................................. 44
Figure 11: BRIEF32 Local Region Distances in Percent Difference ............... 46
Figure 12: BRIEF64 Local Region Distances in Percent Difference ............... 46
Figure 13: BRISK Local Region Distances in Percent Difference .................. 46
Figure 14: SBRISK Local Region Distances in Percent Difference ............... 46
Figure 15: GPU Kernel Execution Pipeline .................................................... 49
Figure 16: BRIEF32 Descriptor With Radius of 20 ........................................ 58
Figure 17: BRIEF32 Descriptor With Radius of 40 ........................................ 58
Figure 18: Tracking Comparison Example Frames. The Yellow Circles are the BRIEF32 Results. The Green Circles are the BRIEF64 Results. The Blue Circles are the BRISK
Results. The Pink Circles are the SBRISK Results. The Cyan Circles are the MIL Results. The White Circles are the Ground Truth Locations. ........................................... 66

Figure 19: Plot of the Euclidian Distance of the Trackers to the Ground Truth for the Cliffbar Video .................................................................................................................. 67

Figure 20: Plot of the Euclidian Distance of the Trackers to the Ground Truth for the David Video .................................................................................................................. 68

Figure 21: Plot of the Euclidian Distance of the Trackers to the Ground Truth for the Dollar Video .................................................................................................................. 69

Figure 22: Plot of the Euclidian Distance of the Trackers to the Ground Truth for the Faceocc Video .................................................................................................................. 70

Figure 23: Well Tracked Example Frames. The Yellow Circles are the BRIEF32 Results. The Green Circles are the BRIEF64 Results. The Blue Circles are the BRISK Results. The Pink Circles are the SBRISK Results. The White Circles are the Ground Truth Locations .......................................................................................................................... 71

Figure 24: Plot of the Euclidian Distance of the Trackers to the Ground Truth for the Animal Video .................................................................................................................. 72

Figure 25: Plot of the Euclidian Distance of the Trackers to the Ground Truth for the Football Video .................................................................................................................. 73

Figure 26: Plot of the Euclidian Distance of the Trackers to the Ground Truth for the Singer1 Video .................................................................................................................. 74

Figure 27: Plot of the Euclidian Distance of the Trackers to the Ground Truth for the Diving Video .................................................................................................................. 75
Figure 28: Poorly Tracked Example Frames. The Yellow Circles are the BRIEF32 Results. The Green Circles are the BRIEF64 Results. The Blue Circles are the BRISK Results. The Pink Circles are the SBRISK Results. The White Circles are the Ground Truth Locations.

Figure 29: Plot of the Euclidian Distance of the Trackers to the Ground Truth for the Basketball Video.

Figure 30: Plot of the Euclidian Distance of the Trackers to the Ground Truth for the Gymnastics Video.

Figure 31: Plot of the Euclidian Distance of the Trackers to the Ground Truth for the Shaking Video.

Figure 32: Plot of the Euclidian Distance of the Trackers to the Ground Truth for the Skating 2 Video.

Figure 33: Execution Time of Each Descriptor with the Tracker.

Figure 34: GPU Speedup on the NVIDIA Tesla K20 GPU.

Figure 35: GPU Speedup on the NVIDIA GTX 480 GPU.

Figure 36: Execution Times of the Sections of Tracker using the BRIEF32 descriptor on the CPU.

Figure 37: Execution Times of the Sections of Tracker using the BRIEF64 descriptor on the CPU.

Figure 38: Execution Times of the Sections of Tracker using the SBRISK descriptor on the CPU.

Figure 39: Execution Times of the Sections of Tracker using the BRIEF32 descriptor on the K20 GPU.
Figure 40: Execution Times of the Sections of Tracker using the BRIEF64 descriptor on the K20 GPU ......................................................... 90

Figure 41: Execution Times of the Sections of Tracker using the SBRISK descriptor on the K20 GPU ......................................................... 91

Figure 42: Parts Tracking Demonstration Frames. Rows 1 and 3 are the Results from Individually Tracking the Eyes. Row 2 and 4 are the Results from Parts Tracking. Red Circles Represent Updates Where the Parts are Not Found because they Fail to Meet the Matching Threshold ......................................................... 96
List of Tables

Table 1: Optimal Parameters for Each Video .......................................................... 61
Table 2: General Parameter Set for Descriptors ...................................................... 62
Table 3: Comparison of Average Distance to the Ground Truth of the Optimal and General Parameters ........................................................................................................... 63
Table 4: Comparison of Average Distance to Ground Truth in Pixels of Trackers ........ 65
Table 5: Results of Well Tracked Videos in terms of Average Distance from Ground Truth in Pixels ........................................................................................................................................ 71
Table 6: Accuracy Results of Poorly Tracked Videos ................................................ 76
Table 7: Single Object Tracking Speed on CPU ......................................................... 82
Table 8: Single Object Tracking Speedup of GPU .................................................... 83
Table 9: Execution times for each descriptor ............................................................. 84
Table 10: Maximum Number of Objects that can be tracked in Realtime ................. 85
Table 11: Comparison of Mobile Execution Time in ms .......................................... 95
## Glossary

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td><strong>GPU</strong></td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td><strong>GPGPU</strong></td>
<td>General-purpose Computing on Graphics Processing Units</td>
</tr>
<tr>
<td><strong>BRIEF</strong></td>
<td>Binary Robust Independent Elementary Feature</td>
</tr>
<tr>
<td><strong>BRISK</strong></td>
<td>Binary Robust Invariant Scalable Keypoints</td>
</tr>
<tr>
<td><strong>SBRISK</strong></td>
<td>Simplified BRISK</td>
</tr>
<tr>
<td><strong>SIFT</strong></td>
<td>Scale-Invariant Feature Transform</td>
</tr>
<tr>
<td><strong>SURF</strong></td>
<td>Speeded Up Robust Features</td>
</tr>
<tr>
<td><strong>CAMSHIFT</strong></td>
<td>Continuously Adaptive Mean Shift</td>
</tr>
<tr>
<td><strong>KLT</strong></td>
<td>Kanade–Lucas–Tomasi</td>
</tr>
<tr>
<td><strong>MIL</strong></td>
<td>Multiple Instance Learning</td>
</tr>
<tr>
<td><strong>TLD</strong></td>
<td>Tracker Learned Detection</td>
</tr>
<tr>
<td><strong>DML</strong></td>
<td>Distance Metric Learning</td>
</tr>
</tbody>
</table>
Chapter 1  Introduction

Object tracking is an important area in computer vision due to the wide variety of potential applications ranging from perceptual user interfaces, to autonomous vehicle navigation and automated panorama generation. These applications can benefit from tracking multiple objects simultaneously and in real time. For example, tracking multiple objects in real time is critical for intuitive and reactive user interfaces which track all the points of interest to the user, or for an autonomous drone which can track a large number of objects that are common in real-world environments. However, a common compromise, even with state of the art trackers, is to sacrifice either speed or accuracy in order to achieve a desired performance goal. Thus, there is a significant need for a tracker that is both computationally efficient and highly accurate to aid in such applications.

Object tracking is a field with foundations largely based on the work by Lucas-Kanade [1]. Later work proposed many other tracking techniques, as in [2, 3, 4, 5, 6, 22]. As these works were developed, a new architecture was introduced by NVIDIA to leverage the Graphics Processing Unit (GPU) for general purpose computing problems in the Compute Unified Device Architecture (CUDA) [7]. The potential of the CUDA architecture in application to the object tracking domain was explored in [8, 9, 10, 11, 12]. Another area of significance to the proposed tracker was the area of local features or keypoint descriptors including SIFT [13], SURF [14], LBP [15], BRIEF [16], ORB [17], BRISK [18], and FREAK [19]. These descriptors were originally developed for image registration, but they have shown promise in application to the object tracking domain [20, 6].
This thesis provides the following contributions to the object tracking field. The first contribution is an object tracker that can track multiple objects simultaneously and has demonstrated a high degree of accuracy while being computationally efficient. The second contribution is an accelerated implementation of the proposed tracker on the GPU using the CUDA architecture. The third contribution is a simplified BRISK descriptor that has demonstrated comparable results to more complex descriptors while being much faster to compute. The fourth contribution of this thesis is an implementation of a parts model for the proposed tracker which leverages accuracy of an object’s motion as a whole to provide a higher quality tracking of individual parts of an object. Finally a mobile implementation has been developed for the Apple iOS platform which has demonstrated promising performance results.

This document is organized as follows: Chapter 2 discusses the prior work done in relation to this thesis and how it helped influenced the design of the proposed work. Chapter 3 discusses the proposed tracking algorithm and its various implementations. Chapter 4 details the experiments performed to benchmark the proposed system as well as their results. Chapter 5 provides a conclusion as well as potential areas for future work for the system.
Chapter 2  

Background

Object tracking is a field that leverages advances made in a many different fields such as object detection and image registration. This chapter outlines the supporting work in these fields and how the techniques can be combined to produce a computationally efficient tracking algorithm. Previous work in the object tracking field is discussed in Section 2.1. Section 2.2 discusses leveraging the GPU for accelerating object tracking algorithms. Finally Section 2.3 details the work done and some of the recent advances in the field of feature descriptors.

2.1. Object Tracking

Object tracking is a field that has progressed rapidly in recent years. Originally object tracking techniques were based on point tracking, such as Kanade-Lucas-Tomasi (KLT) tracking [21], and color based methods, such as the CAMSHIFT tracker [2]. More recently, advanced trackers were designed based on object detection and classification schemes, such as the Multiple Instance Learning (MIL) tracker [22], and Tracking Learning and Detection (TLD) [3], [4]. Methods based on dictionaries of templates are used to match the appearance of the object [5], as was done in the Distance Metric Learning (DML) tracker [6]. Local feature point descriptors, such as the SIFT [13] and SURF [14] floating-point descriptors, were developed for image registration and were adapted for tracking. One such example is the SURF tracker [20]. Important trackers for the work in this thesis are overviewed next.
2.1.1 KLT Tracker

One of the earliest algorithms proposed for object tracking was the KLT tracking method [21]. The KLT approach is described in a series of papers starting with the initial work of Lucas and Kanade [1]. An iterative Newton-Raphson method was proposed in [1] as a means to solve for the transformation that minimizes the dissimilarity between two patches. The equation for the transformation is written as:

$$\delta = Dx + d$$  \hspace{1cm} (2.1)

where $\delta$ is the transformation, $x$ is the image coordinate vector, $d$ is the translation vector and $D$ is the deformation matrix. For a simple translation model, the translation matrix is a zero matrix, while for affine transformations the D matrix is defined as:

$$D = \begin{bmatrix} d_{xx} & d_{xy} \\ d_{yx} & d_{yy} \end{bmatrix}$$  \hspace{1cm} (2.2)

where $d_{xx}$ and $d_{xy}$ are the deformations of $x$ resulting from its $x$ and $y$ positions respectively, and $d_{yx}$ and $d_{yy}$ are the deformations of $y$ resulting from its $x$ and $y$ positions respectively.

The dissimilarity measure is the weighted sum of squared differences (SSD) between the pixel values in the two patches. This leads to the error function between two images $J$ and $I$:

$$\epsilon = \int \int _W [J(Ax + d) - I(x)]^2 \omega(x) dx$$  \hspace{1cm} (2.3)

where $A$ is the distortion matrix, the identity matrix in the case of a pure translation model, and $\omega(x)$ is a weighting function that can be used to favor the central pixels of the patches.
This approach provided the basis for the second paper written by Tomasi and Kanade [23], which proposed a method for detection of optimal points to track. The second paper uses the same Newton-Raphson method to find the displacement of the similarly sized image patches that minimized the error between the image patches in different windows. The full KLT tracker was presented in a third paper written by Shi and Tomasi [21], where they developed a method for determining if there was error in tracking any of the points. A two-part algorithm was proposed, where the first part of the algorithm tracks the image patches from frame to frame using the same algorithm as in the first two papers, and minimization of the error is based on a pure translation motion model. The second part was a method to evaluate the tracked patch as compared to the original patch using a similar Newton-Raphson method to find the optimal affine transformation between the original patch and the current patch. An affine transformation model was used for the comparison to the original, but not the inter-frame update because solving for the affine transformation requires solving for a greater number of parameters, thus increasing the potential for the introduction of local minima in the search area and therefore a greater chance for errors. Additionally, a simple translation model is often sufficient for the motion of objects between frames because the motion is usually small. However, for comparisons between the original frame and the current frame an affine transformation is required because the motion can be much larger due to the accumulation of small affine transformation from frame to frame. If the pure translation model is used, points that are properly tracked might have a large error when compared to the original, which leads to a greater number of false negatives and more discarded points. The affine model accounts for the accumulation of small affine transformations
allowing for a higher quality comparison between the patch and the original. A fourth paper authored by Baker and Matthews [24] proposed an optimization which greatly reduced the complexity of solving the translation optimization calculation.

2.1.2 CAMSHIFT Tracker

A region-based tracking algorithm that has gained widespread popularity is the CAMSHIFT algorithm [2]. CAMSHIFT is based on the MEANSHIFT algorithm [25], which is an iterative process that follows the gradient of the probability density to find the peak distribution, or mode, of a set of data samples. The MEANSHIFT algorithm is adapted for use in tracking by building a target histogram of the object to be tracked. The target histogram is traditionally built using the hue channel of the Hue Saturation Value (HSV) color space. The hue channel is used because it best represents the color of each pixel, while remaining invariant to changes in illumination. Once the target histogram is created, it is used to determine the probability of each pixel in a search window belonging to the object to be tracked. The probability image is used by the MEANSHIFT algorithm to move the center location of the search window to the center of mass of the pixel probabilities in the search window. This is done based on the zeroth ($M_{00}$) and first moments ($M_{10}$,$M_{01}$) of the probability density image $I(x,y)$.

$$M_{00} = \sum_x \sum_y I(x,y)$$  \hspace{1cm} (2.4)

$$M_{10} = \sum_x \sum_y xI(x,y)$$  \hspace{1cm} (2.5)
\[ M_{01} = \sum_x \sum_y yI(x, y) \]  

(2.6)

The x and y coordinates for the center of the distribution is determined as follows.

\[ x_c = \frac{M_{10}}{M_{00}} \]  

(2.7)

\[ y_c = \frac{M_{01}}{M_{00}} \]  

(2.8)

The gradient descent process is repeated until the change in the search window’s position between iterations falls below a predetermined threshold. Once the object’s location estimate meets the threshold, or a maximum number of iterations is reached, the resulting search window is set as the object's updated location for that frame. The MEANSHIFT process is then performed on the next frame, using the updated location as the initial guess for the object's location. The CAMSHIFT algorithm expands upon the MEANSHIFT algorithm by adapting the size of the window so that the tracker can account for changes to the perceived size of the object, due to changes in distance between the object and the camera.

CAMSHIFT follows the same procedure as the MEANSHIFT algorithm with the addition of the step of changing the search window size during each iteration. The window size is changed as a function of the zeroth-moment of the pixel probabilities, also known as the integral of the probabilities. This method is widely used for tracking simple objects with uniform appearance; however, it tends to fail when objects have more complex appearances that vary over time.
2.1.3 Particle Filter Tracking

Particle filtering was first introduced by Doucet et al. [26] as a method to estimate the state of dynamic systems which cannot easily be modeled by a linear Gaussian model. The theory behind particle filtering is that a weighted estimation of the object state, size and location, based on a series of guesses of a large number of separate particles. The weight associated with each particle is based on the accuracy of the particle in estimating the location of the object which is learned as the tracker progresses. Many different methods can be used to model the particles, such as the color-based method in Perez et al. [27] or more complicated adaptive appearance models proposed in Zhang et al. [28] and Kwolek [29].

2.1.4 Tracking by Classification and Detection

Following the early tracking algorithms, more advanced algorithms, such as MIL and TLD, were developed based on concepts borrowed from the object detection and classification field. These algorithms are feasible in modern computing due to the increase in computing power as well as the improved efficiency of modern classification algorithms.

2.1.4.1 TLD Tracker

A popular tracker which is based on the tracking by detection paradigm is the Tracking Learning and Detection method by Kalal et al. [3, 4]. This algorithm leverages the structural constraints present in object detection with techniques such as bootstrapping to train the classifiers better for the object detector. The first step of the
algorithm is to generate a classifier based on a label image patch as well as 300 additional examples that were generated by applying affine transformations to the labeled patch. Once the detector is initialized, it is used in a sliding window manner to detect the object in the next frame. After the location of the object is updated to the location with the highest probability from all of the tested window locations, the TLD tracker uses an online learning process called Positive Negative Learning (P-N Learning). In this learning technique, the patches which have a high probability based on the previous detector, yet have a lower probability than the winning solution, are used as negative examples to retrain the classifiers in a bootstrapped manner.

2.1.4.2 MIL Tracker

The Multiple Instance Learning tracker (MILTrack) [22] is one of the most popular tracking algorithms. MILTrack is commonly used as the baseline for other trackers in terms of both accuracy and performance. As the name suggests, MILTrack is based on the Multiple Instance Learning (MIL) paradigm [30] in which the classifier is trained based on sets of example data, instead of the traditional approach of training based on individual labeled examples. The MILTrack algorithm aims to train a classifier that can differentiate between positive patches (foreground) and negative patches (background). Viola et al. suggest in [31] that the traditional method of object detection lacked the ability to account for the ambiguities that are present in all detections. Therefore, MIL is recommended for use in object detection because it can accommodate for such ambiguities. In MIL, sets are generated based on locality to the labeled point and labeled based on whether the set contains a positive example. If a positive example is
contained in the set, the set is labeled as positive; otherwise it is labeled as a negative set. These labeled sets are then used to determine the set of feature descriptors that can be combined to classify the sets with highest accuracy. The process of using a large number of weak descriptors to form a strong descriptor is known as boosting. This approach was used with MIL in the MILBoost algorithm [32]. However, a disadvantage of using the MILBoost algorithm is that it requires all of the labels for the training sets prior to running, so it can train in an offline manner. Because of this, the MILTrack algorithm leverages additional techniques from the AdaBoost algorithm [33] to allow for online training of the classifier.

The program flow of the MILTrack algorithm is relatively simple compared to the learning portion of the algorithm. Each time a frame is received, the MILTrack algorithm uses the previous location of the object and the set of classifiers that were trained in the previous round to determine the probability of the object for different locations. Once the object location is updated to the location with the highest probability, two sets of windows are cropped to be used for updating the appearance model of the object and updating the classifiers. The positive set of examples is generated by the set of windows within a given radius of the updated location. The negative examples are generated by randomly selecting patches outside of the radius but still within a predetermined distance from the updated location. The two example sets are then used to determine the best classifiers and their weights for the object’s updated appearance. This approach has shown a high success rate in the field of tracking.
2.1.5 Template Trackers

Even though much of the focus in recent years has been on building classification based trackers, such as TLD and MIL, research has continued on trackers which match image patches or other representations of regions of the image that are stored as templates of the object; this approach is known as template matching. Examples of trackers designed based on template matching are the tracker proposed by Matthews et al. in [5], the SURF Tracking [20], and the DMLTracker [6].

2.1.5.1 Matthews et al.

An important aspect of template matching algorithms is how they can adapt to changes in appearance of the object being tracked. If the template is not updated, the object being tracked may be easily lost; thus, it is essential to update the template of the object. Matthews et al. [5] caution that a naive approach to updating the templates may allow the tracker to drift and propose a method to determine when a template is of sufficient quality to be updated. They suggest realigning each matched patch to the original template and then calculating the difference between the aligned patches. If the discrepancy is within a predetermined threshold, the aligned image is stored as the template; otherwise it is discarded. This allows for a margin of acceptable change in appearance of the object without allowing the tracker to drift.

2.1.5.2 SURF Tracker

Another tracker of interest is the SURF Tracker presented in [20]. In this tracker, an object is described by a set of SURF feature points that are initialized at the start of
tracking. The stored feature descriptors are matched to the features that are extracted around them in the next frame. The features are matched by solving an energy minimization problem using a graph-matching approach, as proposed by Torresani et al. [34]. The location of the object is then updated based on a weighted approximation of the movement of the set of points. After the object’s location is updated, the set of features which were lost (i.e. fell below a threshold of similarity or moved in a manner that was inconsistent with the motion of the object) are discarded. New features detected in the updated position of the object are then added to the object’s list of features. This process is repeated for the duration of the video sequence.

2.1.5.3 Distance Metric Learning Tracker

A tracker of key importance to this thesis is a tracker based on Distance Metric Learning (DML) or DMLTracker [6]. Unlike prior classification trackers [22] and [4], which adapt to the appearance of an object by determining the detectors that best distinguish the object from the background, the DMLTracker does not change the method it uses to describe patches. Instead it alters the distance metric used to determine the similarity of two templates. This tracker is based on the template tracking design; however there are many key design components in the tracker, which includes using a library of templates, using a fixed grid of SIFT descriptors for templates, using random projections for dimensionality reduction, and leveraging Distance Metric Learning for tracking.

The first key contribution of the DMLTracker is the use of a library of templates consisting of static and dynamic templates. This allows for multiple appearances of the
object to be stored and used to search for it in each frame. It is important for such a
technique to determine when a new object appearance should be added to the library. As Mathews et al. [5] showed, not every tracked template should be added to the library. Sometimes templates should be discarded (e.g. when there is partial occlusion). In order to make the decision to add the template, there is a two-step check that is performed. First, it must be determined if the object has been found in the new frame. This is accomplished by determining if the candidate image patch that is the closest match for the object is within a given threshold distance to the current elements in the library. If the candidate is within the threshold, the object is determined to have been found; otherwise it is deemed occluded or lost. If the new appearance is a closer match than the previous frame, the new appearance is used to update the template library. This update strategy is used as a means of maximizing the diversity of appearances in the library while minimizing the potential of drift while tracking.

The approach used to describe image patches in the DMLTracker is to use a 3x3 regular grid of SIFT descriptors that are concatenated together, producing a vector that is of dimension 1152, 9 descriptors of 128 dimensions each. This descriptor has optimal differentiation between image patches; however its dimensionality is too high to be used for a real time application. Thus, a dimensionality reduction technique known as Random Projections (RPs) is used. The theory behind Random Projections is based on the Johnson-Lindenstrauss lemma [35], which states that elements in a high dimensional space can be projected into a lower dimensional space by multiplying by a random matrix which adheres to a set of constraints without imposing a significant distortion to the
distances between the elements. Therefore, the high dimensional templates can be reduced by multiplying by a matrix that adheres to the distribution:

$$\begin{cases} 
1, \text{ with probability } \frac{1}{6} \\
0, \text{ with probability } \frac{2}{3} \\
-1, \text{ with probability } \frac{1}{6} 
\end{cases}$$

(2.9)

This is significant because the descriptors can be reduced from 1152 dimensions to 300, a 2/3 reduction in dimensionality.

In order to determine the similarity between elements in the dictionary and those belonging to the background, a distance metric must be used. The distance metric that is used most often is the Euclidian distance (i.e., the L2-Norm). Distance metric learning is a process of determining a transformation of the input data that minimizes the distance between elements of the same class and maximizes the distance to elements of other classes. The DML process is an iterative process which takes an initial estimate of the distance matrix $G_0$ and a set of labeled data and attempts to find a new distance matrix $G$ that maximizes the distances between classes and is similar to the original matrix $G_0$. The distance matrix $G$ is a mapping of the descriptors in a space which maximizes the distance between the classes of the descriptors. The distance matrix $G$ is continually adapted such that the distance is maximized with the introduction of each data point. A benefit of DML is that it can be done both offline, using Information Theoretic Metric Learning (ITML) [36], and online, using Logdet Exact Gradient Online (LEGO) [37]. Both of these methods are used in the DMLTracker algorithm. ITML is used to initialize the distance metric and thresholds by initially running the tracker for a set number of frames with the distance matrix $G$ set to the identity matrix; in [6] they used 4 frames.
Once the foreground and background appearances are collected in the first frames, they are used to train the distance metric $G$ and determine the thresholds. Once the metric is initialized, the LEGO algorithm is utilized to update the distance metric for each iteration that the object template meets the threshold criteria in an online manner.

Another design aspect of the DMLTracker is the use of a local gradient descent method to search for the best match in each frame. The search method consists of finding the best match for a set of patches in a specified pattern centered on the previous location of the object. Once the best match is found, the pattern is moved to the location of the best match. If the best match is at the center of the pattern, the search procedure is finished. This greatly reduces the complexity of the search process by limiting the number of comparisons necessary to only those within a reasonable distance from the last location of the object. This method is based on the assumption of smooth motion, a common assumption for trackers. The pattern used in the algorithm can be found in Figure 1; a course to fine search pattern is used so as to maximize the ability of the tracker to track fast motion while maintaining the ability of the tracker to have accurate localization.

![Figure 1: DML candidate search pattern](image)
2.2. **GPU Acceleration**

2.2.1 **General Purpose GPU Computing**

One of the major advances in computational efficiency today is to do General Purpose Computing on the GPU (GPGPU). This was originally proposed as a means to leverage the computing power of the GPU to process Single Instruction Multiple Data (SIMD) in parallel. At first, the GPGPU algorithms had to map the scientific calculations on the graphics processing domain of the GPU libraries, such as OpenGL [38]. This approach was cumbersome and did not fully unlock the power of the GPU due to the several additional steps that were needed to translate the calculations between domains. In 2007, NVIDIA recognized the potential for GPGPU implementations on their hardware and released the first GPGPU API, the Compute Unified Device Architecture (CUDA) [7]. This API offered a large increase in flexibility to leverage the hardware for GPGPU development, and included features such as fine-grained thread configuration control, access to several different memory architectures, and detailed information on how best to utilize the hardware in GPGPU applications. However, along with the power of the GPU some potential drawbacks emerged, including slow CPU to GPU memory transfer times and requirements on adhering to the SIMD design pattern.

When designing algorithms for the CUDA architecture, the underlying hardware must be well understood in order to achieve maximum performance. Modern NVIDIA GPUs consist of a large number of small, less powerful processing cores. These smaller cores are known as CUDA cores, and some of the most powerful cards have as many as 2496 single precision CUDA cores. Although these cores are less powerful than the cores
found in modern CPUs, they can greatly increase the speed of an algorithm by performing thousands of computations simultaneously. These cores are grouped together to form what are known as Stream Multiprocessors (SMs). The number of cores per SM varies based on the architecture of the GPU. For example, on one of the high end cards, the NVIDIA Tesla k20, there are 13 SMs, each with 192 CUDA cores per SM. The number and size of the SMs are important in the design of the threading configuration, because they play a role in the maximum number of threads that can be allocated to each SM. Each SM schedules the execution of the threads assigned to it in groups known as warps. The size of the warps varies based on the architecture of the GPU; however, generally there are 32 threads per warp.

One of the benefits of the CUDA architecture is that it allows for fine grained control over the configuration of the threads. Threads are configured in a “grid” of “blocks” to be run. Blocks are groups of threads that are run on the same SM and share certain memory structures. The dimension of the block can be specified for the x, y, and z axis. However there are limitations on the size of blocks; thus it is often required to start multiple blocks of threads. The configuration of the number of blocks of threads is known as the grid. The grid can also be configured in the x, y, and z axis. The ability to set the dimensions of both the grid and block in the three separate axes is important, because often the thread’s position in the block and grid is used to determine the data that it uses. Several factors must be taken into account when configuring the threads to be run. The first major factor is the limit on the number of threads per SM. Several blocks can be allocated to one SM, as long as they can be entirely fit under the limit per SM. If blocks are inappropriately configured, there is potential for a large waste of hardware on each
SM. Another architectural specification that plays a role in the thread configuration is the maximum number of threads per block. This number is usually lower than the number of threads per SM; thus blocks are usually designed such that multiple blocks can fit on one SM. An additional architectural specification that plays a major role in the threading configuration is the number of threads per warp for the given GPU. If the number of threads per block is not a multiple of the number of thread per warp, the hardware will be underutilized, which will affect the performance of the algorithm.

The best strategy to use when designing the thread configuration in CUDA is to limit the number of threads per block to a number that is lower than the maximum number of threads per block, yet evenly divides the number of threads per SM. Additionally it is important to ensure that the number of threads per block is a multiple of the number of threads per warp. Once the limit is determined for the number of threads per block, the three dimensions are free to be set in a way that best suits the data and algorithm.

Another benefit of the CUDA architecture is the flexibility that it allows with respect to the memory architecture used to store data on the GPU. There are six different types of memory: registers, local memory, shared memory, global memory, constant memory, and texture memory. Each of these six memory types has it’s own benefits and drawbacks.

Registers are the fastest type of memory on the GPU. Registers are local to each thread and cannot be accesses from another thread. The number of registers is limited, and thus it is important to limit the number of variables used in each kernel so that as many as possible can be stored in registers. If there are too many variables in each thread
to be stored in registers, they will overflow to local memory. Registers and local memory are allocated through traditional variable declaration in CUDA. Local memory is stored in the main memory of the GPU; thus accessing local memory adds to the bandwidth overhead in the GPU’s main memory. The bandwidth of the GPU’s main memory is often a bottleneck in CUDA; thus it should be avoided as much as possible. In order to alleviate some of the bandwidth overhead main memory is cached, thus limiting the number of accesses to main memory.

Shared memory is a special type of memory that is not found in traditional programming for the CPU. Shared memory is local to each block, and each thread in a block can access it; however threads outside the block cannot. Shared memory has many benefits including that accesses to shared memory are on the SM; thus they are not adding overhead to the main memory bandwidth. Shared memory is often used when a block operates on data with high locality multiple times. There is high potential to speed up many algorithms through use of shared memory. The allocation size of the shared memory must be known at compile time.

Global memory can be accessed (read and written) by any thread on the GPU. Global memory is often where results are stored as well as other data that must be communicated between threads. Global memory is stored in the main GPU memory; thus it also requires a hit to the memory bandwidth when it is accessed. As is the case with local memory, global memory is cached.

Constant memory is a special memory in CUDA that is read only by the GPU. Constant memory is a very limited type of memory that is stored in device memory. However, it is highly cached, such that accesses to constant memory can be made much
faster than with global memory. Constant memory is good for data that do not change throughout the run of the kernel, such as lookup tables. As with shared memory, the allocation size of constant memory must be known at compile time.

Another specialized memory type that is unique to CUDA is texture memory. Texture memory is stored on the device memory; however it is cached two dimensionally, and cache misses do not negatively impact the memory bandwidth. Another feature of texture memory is that it is designed to have consistent access times for both cache hits and misses. This allows for better scheduling around texture memory accesses by the GPU. Texture memory is good for data that have high 2D spatial access locality.

It is important to mention that with all of the benefits of the CUDA architecture there are additional design constraints that must be adhered to in order to maximize the performance of the computation. The two areas that are most important in terms of negatively affecting performance are inconsistent program execution and poor memory access patterns.

Program execution and control of the program flow are vital to all programing applications. However, due to the requirement of SIMD for CUDA, all of the threads in each warp must be executing the same instructions. Therefore, if there are conditionals in a kernel that separate threads in a given warp, there will be added overhead for the scheduling of the separate threads. This is why it is important to limit the number of conditionals used in a CUDA kernel. A common area where conditionals are used in kernels is for boundary conditions of a problem, such as the boundary of an image. Instead of using conditionals for such cases, a better strategy is to allow for computations
to be performed on the invalid locations, and then discard the data in the locations where the invalid threads stored their results. Such an approach increases the total number of computations; however, it greatly reduces the execution time of the computations by removing the conditional.

Another common area where performance can get degraded in CUDA is in the memory access patterns. The CUDA architecture attempts to mitigate the impact of global memory accesses by allowing for faster fetches of large chunks of consecutive memory accesses by multiple threads. Such accesses are known as coalesced memory accesses because they result in a single access instead of multiple ones. Coalesced memory accesses provide a large potential increase in speed for many algorithms; thus it is recommended to design such accesses into kernels whenever possible.

### 2.2.2 GPU accelerated Object Trackers

Due to the recent adoption of Graphics Processing Units (GPUs) for general purpose computing, there is a large amount of research on how to best utilize their power for important computer vision problems. Recently there have been attempts to use GPUs for some basic object tracking algorithms. Rymut et al. [10] have used the CUDA API to implement a Particle Filtering approach to tracking, and have demonstrated a speedup of over 30 times the performance of the same algorithm on the CPU. Additionally, Exner et al. [11] have demonstrated that the CAMSHIFT algorithm, as described in Section 2.1.1.1, can be implemented on the GPU with speedup between 2.2 and 6.0 times the performance of the CPU implementation.
### 2.2.2.1 CAMSHIFT on the GPU

The first tracking algorithms to be implemented on the GPU were based on the MEANSHIFT and CAMSHIFT algorithms. There have been several GPU implementations of the MEANSHIFT and CAMSHIFT algorithms; two of the most significant implementations were done in [12] and [11]. In [12], Li et al. implemented the mean shift algorithm on the GPU using CUDA with the addition of k-means clustering of colorspace in order to minimize the number of bins necessary for the histogram. Li et al. processed an algorithm that consisted of eight separate kernels for the different steps of the tracking process. The first kernel computes the portion of the histogram allocated to each block of threads by leveraging shared memory. The result from the first kernel is written to global memory to be used in the second kernel, which calculates the non-normalized histograms for the m different candidates. The third kernel is used to normalize each candidate histogram through the use of a parallel summation reduction algorithm. Kernels four through seven are used to determine the moments of the candidate windows. The last kernel is used to determine the updated position of the objects based on the moments calculated in the previous steps using a parallel summation reduction algorithm. The approach by Li et al. did not add much to the robustness of the tracking and improved the tracking speed by only roughly 2-3 times that of the CPU implementation.

Exner et al. [11] expanded on the CAMSHIFT algorithm by accumulating histograms of multiple views of an object to track objects better that have complex appearances. One drawback of this method is that it requires a-priori knowledge of the object being tracked, so that it can generate the histograms for the multiple appearances.
off-line. It may be possible to generate such histograms in cases where the object is known, such as face tracking, but when the object is not known, this is not possible. A method for stable tracking of multiple objects after occlusion was also proposed, where the search window is matched against all of the histograms of the object by finding the histogram in the set of histograms for the object that maximizes:

\[ d(h, g) = \sum_{b=0}^{B-1} \min(h(b), g(b)) \]  

(2.10)

where \( B \) is the number of bins in the histograms \( t \) is the target histogram. If this distance falls below a predetermined threshold, the object is determined to be lost. Additionally, Exner et al. added a method for redirecting a lost object through use of a hierarchical quad-tree redirection strategy.

The GPU implementation of the extended CAMSHIFT algorithm proposed by Exner et al. was done using the OpenGL API, because it was better established than the CUDA API at the time. Therefore, the exact implementation of the algorithm is substantially different than the GPU implementation proposed in this thesis. However, the techniques used in the separation of the components of the problem based on the inherent parallel nature of each step are still pertinent. The first part of the algorithm which has high potential for parallelization on the GPU is the generation of the multidimensional histograms. This problem is parallelized though use of texture memory for the object histograms, and uses vertex shaders to determine the bin where each pixel belongs. The next step is to parallelize the Back-Projection of the Probabilities, which is done by storing the histogram in texture memory and using the pixel colors as the coordinates to access the texture memory. After this, the moments of the image are
calculated. The moments are calculated using vertex shaders for each pixel in the search window using iterative texture look-ups. The five resulting values are stored in texture memory to be transferred off the GPU. The histogram matching is distributed on the GPU by assigning threads on a per-bin basis that then accumulates the results to be offloaded to the CPU. The entire program flow for their implementation is managed by the CPU; thus there is a high coupling between the GPU and CPU. Their GPU implementation of the traditional CAMSHIFT algorithm is 8.8 times faster than the CPU implementation. Additionally the extended CAMSHIFT algorithm is 2 to 6 times faster than the CPU implementation.

2.2.2.2 Particle filtering on the GPU

Particle filtering has demonstrated high potential for parallelization due to the high number of operations that are independent and thus could be computed simultaneously. Rymut et al. [10] propose an implementation of a Particle Filter object tracker which utilizes adaptive appearance models. Their implementation is separated into five kernels that maximize the potential parallel operations for each step of the process. Their experimental results have demonstrated a high speedup on the GPU of 30 times the CPU implementation when tracking 256 particles.

2.2.2.3 KLT on the GPU

One of the most promising tracking approaches for parallelization of a large number of objects is the KLT tracker, due to the complete independence of the tracking process for each object. Simple implementations were initially proposed that could track
over 100 objects in real time by Hedborg et al. [39] and Sinha et al. [40]. Zach et al. [41] proposed an implementation of KLT on the GPU that could adapt to changes in illumination using a gain parameter. However these approaches have two major drawbacks. The first is that neither implementation leverages the flexibility and control over the calculations on the GPU offered by the CUDA framework. The second drawback is that both implementations rely on a simple translation warping model for template matching. The translation model may be acceptable for some applications with only basic motion. However for more complex scenes, such as the view from an UAV, the translation model is not sufficient and a more complex model is needed, such as the affine photometric model proposed by Jin et al. [42]. The downside to implementing a more complex warping model is that the computational complexity increases on the order of $O(n^3)$, where $n$ is the number of parameters in the motion model. The simple translation model has two parameters leading to a complexity of 8, where the more complex affine photometric model has 8 parameters leading to a complexity of 512 (64 times that of the simple model). For this reason the affine model was avoided in prior implementations.

With the increasing capability of the GPU to perform these operations in parallel, such an increase no longer has such a large impact. A robust implementation of the KLT tracker was presented in Kim et al. [8] where an affine photometric warping model was used for template matching. In order to compensate for the potential error introduced by the high complexity model (higher potential for local minima) Kim et al. relied on additional information gathered from an Inertial Measurement Unit (IMU). The IMU provided a better estimation for the initial guess, and thus the tracker was less likely to
find local minima. Although this approach is helpful, it is not always feasible to have an
IMU on the camera used for tracking.

Kim et al. leverage optimizations that were proposed by Baker et al. [24], as
discussed above. Additionally they utilize many of the key concepts in designing a
GPGPU algorithm by taking into consideration CUDA specific recommended design
practices such as: memory types, memory access patterns, program divergence, and
transfers to and from the GPU. These factors play a major role in all algorithms
implemented on the GPU. Based on their algorithm design, there were two types of
operations that were to be implemented on the GPU: operations on each pixel and
operations for each feature. Thus they had two separate thread design schemes.
Operations that had to be performed on each pixel consisted of calculating gradients and
resampling the image to create an image pyramid. The operations that are performed on a
feature level consisted of generating the table of features, computing the error for the
given warp, and updating the motion parameters of each feature. Due to the random
nature of the memory accesses of the threads for each feature and the sparseness of the
features, the images (input, pyramids, and gradients) are all stored in texture memory,
which is optimized for random accesses. The parameters for the features such as template
images, motion parameters, and Hessian matrices are stored in global memory because of
their sequential memory access patterns. Constant memory is used to store the IMU data
for each iteration on all of the threads to access.

The GPU is designed as a SIMD processor, and thus there are often pieces of
algorithms that are not optimal for implementation on the GPU. This fact often points
toward a hybrid GPU-CPU implementation, such as was the case in Kim et al. In their
implementation, it was clear that operations on sorting the cornerness measures and the calculating the inverse of the Hessian matrix were better suited for the CPU, even with the added overhead of the memory transfers. Their approach has demonstrated the high potential for using GPUs to accelerate object tracking, even with a simple tracking algorithm.

2.3. Descriptors

Another key area of interest for this thesis is local feature descriptors. Local feature descriptors have shown rapid progress in recent years as a means of generating a compact encoding of image patches that is both unique and robust. Common local feature descriptors are SIFT [13], SURF [14], LBP [15], BRIEF [16], ORB [17], BRISK [18], and FREAK [19]. The work in local feature descriptors largely revolves around their use for image registration. Furthermore, it was shown in recent work that descriptors have potential utility in the field of object tracking.

2.3.1 SIFT

The Scale Invariant Feature Transform (SIFT) feature descriptor sparked much of the innovation in the field of descriptors. The SIFT descriptor was introduced by David Lowe in 2004 [13]. In his work, Lowe demonstrated that it was possible to generate a descriptor for a patch of an image which was invariant to changes in scale, rotation and illumination. Thus, descriptors between two images could quickly be compared using the L2-Norm. Comparing two SIFT descriptors requires a simple process of calculating the Euclidian distance between 128 floating-point values in the descriptors for the two
patches. The SIFT descriptor achieves invariance through a two-step approach. In the first step a feature detector determines the location and scale of feature points. The detector identifies key points over different scales by calculating the Difference of Gaussians (DoG) over multiple octaves of the image. The DoGs allow for the detector to determine both the location of “blobs,” based on the location of local maxima, in the individual levels of the DoG, as well as the scale of the blob depending on the level where the local maximum is found. The location of the key points is further refined using Taylor Series expansion in the scale space of the image. Once the scale and location are known for the key point, the orientation of the key point is determined using the gradient magnitude and direction of a variably sized patch surrounding the key point and an orientation histogram around the patch. The gradient magnitude is calculated as follows.

\[
m(x, y) = \sqrt{(L(x + 1, y) - L(x - 1, y))^2 + (L(x, y + 1) - L(x, y - 1))^2}
\]  

(2.11)

The orientation of the patch is computed based on the pixel values in the smoothed image using:

\[
\theta(x, y) = \tan^{-1}\left(\frac{L(x, y + 1) - L(x, y - 1)}{L(x + 1, y) - L(x - 1, y)}\right)
\]

(2.12)

where \(L(x, y)\) is the intensity value as location \((x, y)\) in the Gaussian smoothed image.

Once the orientation of the patch is determined, the patch is split into 16 blocks. An 8-bin orientation histogram is calculated for each of the 16 blocks, and the bins are concatenated together to form the whole descriptor.
2.3.2 SURF

After the success of the SIFT algorithm, much research was done to develop other descriptors. One such feature descriptor, which aims at reducing computational complexity, is Speeded Up Robust Features (SURF) developed by Bay et al. [14]. In this work, the Hessian Matrix of an image is leveraged to extract the same information as the DoGs used in SIFT. The creation of the Hessian matrix is approximated through the use of the integral image; this greatly accelerated the computation. The orientation of the feature is calculated in an efficient manner using the integral image to determine the wavelet response in the x and y directions. The responses are then used to determine the orientation of the descriptor through a summation of the responses along a sliding orientation window. The descriptor is then constructed by orienting a square region along the calculated orientation by breaking the region into 16 subregions, a 4×4 regular grid. Following this step, the responses to wavelets in the major and minor axis, with relation to the feature orientation, are then sampled every 5 pixels. A weighting function with a Gaussian distribution is then applied to the sampled responses. The sum of the values and sum of the magnitude of the values are then calculated along each axis for sub regions. The SURF description process generates a descriptor that consists of 64 floating-point values, 4×16, that is both simpler to compute and often more robust than the original SIFT descriptor.

2.3.3 LBP

One of the first descriptors to move away from floating-point representations and toward binary string representations was Local Binary Patterns (LBPs) [15]. LBPs do not
aim to be as robust a feature descriptor as the floating-point counterparts, because their initial aim was for use in texture classification. The LBP feature descriptor is generated through a series of comparisons of intensity values of a center pixel and pixels located on a ring that is a set distance from the center. A pixel on the ring that has a higher intensity than the center pixel is a “0” in the string. A pixel that has lower intensity than the center is a “1.” The ring is traversed in a clockwise or counterclockwise manner. Similarity calculations between descriptors could be computed much faster with binary strings, because it could be done using the Hamming distance between the two binary strings.

An example of an LBP feature can be found in Figure 2. Here the descriptor describes the ring with a radius of 1 and starts at the top left of the ring. This is a common configuration for LBPs because the descriptor fits entirely in a single byte. Here the center pixel’s intensity value is greater than the first two pixels and the fifth pixels, and smaller than the rest. Thus the resulting string is “11001000.”

![Figure 2: LBP Descriptor Example](image)

**2.3.4 BRIEF and ORB**

In their work [16], Calonder et al. demonstrate the effectiveness of a descriptor generated based on simple randomly distributed binary comparisons between intensity values in the key point region. The descriptor proposed in Calonder et al. is known as Binary Independent Elementary Features (BRIEF) descriptor. The BRIEF descriptor consists of a string of binary values that are generated based on random comparisons.
made in the keypoint region. The BRIEF descriptor approach allows for a much faster method of generating a feature descriptor. Even though the BRIEF descriptor made no attempt to be robust with regard to rotation the results demonstrated that such robustness may not always be necessary.

Eventually robustness to rotation was introduced through an extension to the descriptor known as the Oriented FAST and Rotated BRIEF (ORB) descriptor, developed by Rublee et al. in [17]. The ORB descriptor added robustness to rotation by rotating the comparison points such that they were aligned with the orientation of the gradient of the patch.

### 2.3.5 BRISK

The positive results from the initial local binary features generated interest in developing two more local binary feature descriptors. The first was the Binary Robust Invariant Scalable Keypoints (BRISK) descriptor developed by Leutenegger et al. in [18]. Unlike the random comparisons made in both ORB and BRIEF, the BRISK descriptor uses comparisons on a set of 60 regions in concentric rings around the center point, as shown in Figure 3. The comparisons between the points that were further way from each other were used to determine the gradient orientation of the descriptor. The comparison between points that were closer, were used to generate the binary descriptor. The BRISK descriptor demonstrated better matching behavior than the descriptors before while being simpler to implement.
A recent local binary feature descriptor that has been demonstrating promising results is the Fast Retina Keypoint (FREAK) descriptor, which was developed by Alahi et al. in [19]. The FREAK descriptor uses regions that are based on the anatomy of the human eye in order to quickly and efficiently match objects it sees.
Chapter 3  Tracking with Local Binary Descriptors

This chapter describes an approach to object tracking based on local binary descriptors. The design of the tracking algorithm is discussed in Section 3.1. In Section 3.2 an approach to tracking parts of an object based on a constrained motion model is presented. Section 3.3 outlines the reasoning for the binary feature descriptors that were selected for implementation in this thesis. A GPU accelerated implementation is discussed in detail in Section 3.4. Lastly this chapter details the implementation of the proposed tracking algorithm on mobile devices in Section 3.5.

3.1.  Algorithm Design

3.1.1 Single Object Tracking

The object tracker consists of four main components: candidate descriptor generation, template dictionaries, candidate scoring, and candidate selection. The approaches taken in each of these components play an integral role in the success of the tracker as a whole. Each component of the tracker algorithm is explained in more detail in this section. The program flow for the tracker can be seen in Figure 4.
The algorithm starts with the introduction of a new frame to the tracker. The description of the object in each frame is stored in the dictionary of templates, which consists of a dynamic part and a static part. Dynamic templates are refreshed with every new frame and capture transient variations in object appearance. Static templates are few and represent high confidence instances of the object, for example when an object detector is used. The dynamic templates are capable of tracking object variations due to pose, illumination, and partial occlusions, while the static templates are included to prevent drift.

For every new frame, a search grid of candidate locations is generated based on a sampling configuration that is centered at the previous location of the tracked object. If the static dictionary is not initialized, a descriptor is generated based on the initial location of the object and is stored in both the static and dynamic dictionaries. The algorithm repeats the process of adding templates to the static and dynamic dictionaries until all of the entries in the static dictionary are filled.
The first step in the tracking process is to generate descriptors for the candidates in the search pattern around the location of the object in the previous frame. Then each of the candidate descriptors are scored against the template descriptors in the static and dynamic dictionaries. A locality constraint penalty is incorporated in the scoring to stabilize tracking and prevent drift. The resulting score is used to compare the candidates and select the best match to the templates in the dictionaries. The score of the best match is compared against a threshold value to determine if the candidate is of sufficient quality to be deemed a true match. If the score of the best match is within this threshold, the candidate’s descriptor is added to the dynamic dictionary, and the location of the object is updated; otherwise, the object is declared lost or occluded in the frame, and nothing is updated. The tracker repeats the same process with the next frame until there are no more frames to be processed.

### 3.1.1.1 Template Dictionary

The proposed algorithm relies on two sets of templates, known as static and dynamic dictionaries, in order to find the object in each frame. The dictionaries hold descriptors of the object that are used for comparison with each candidate location. The dictionaries are made up of two parts: the static dictionary and the dynamic dictionary, such that each plays a separate role in the tracking process. The static dictionary maintains information on the object’s original appearance, while the dynamic adapts to changes in the appearance of the object.

The static dictionary is populated by descriptors generated on the initial “labeled” object. It does not matter how the labels are generated, as the labeled object could be
generated based on a ground truth, user selection or an automated detector. Once the static dictionary is filled, it does not change throughout the execution of the algorithm. This is because in tracking mode labeled data are unavailable after initialization, and thus the static dictionary remains unchanged. The static dictionary could potentially be updated if additional detection was run during the operation of the tracker. In general, the detection process slows down tracking, and interleaving detection with tracking was not implemented.

The dynamic dictionary holds the descriptors for winning candidates found in previous frames. The aim of the dynamic dictionary is to adapt to changes in the appearance of an object over time. This adaptation is done through use of a First In First Out (FIFO) queue structure for the dynamic dictionary. The FIFO queue allows for the winning descriptors from the most recent frames to be compared against the candidates in the next frame. As tracking progresses, the dynamic dictionary adapts to the descriptors of the object as it changes. This allows for large overall changes in the object’s appearance although the changes in consecutive frames might have been subtle. Relying solely on the static dictionary would not allow for such adaptation. If only the static dictionary were used, the object’s appearance could change such that it would no longer match the static dictionary, even though it still was a valid track of the object. The dynamic dictionary would normally need to be larger than the static dictionary in order to track the changes of the object over time. One potential negative impact of the dynamic dictionary is the potential to introduce drift to the tracker; this is overcome by applying a bias when comparing candidates against the dynamic dictionary.
3.1.1.2 Search Area Design

An important part of the tracking algorithm is the selection of the potential candidate locations when searching for the object. A first approach to selecting candidates was using existing methods to find points of interest, such as Harris corners [43], GoodFeaturesToTrack [21], FAST points [44], and SIFT points [13]. However, the points generated from these approaches had two major limitations. The first limitation was that the number and location of points generated by the detectors varied greatly from frame to frame depending on the orientation of the object and changes in illumination. Additionally, the descriptor generated from the candidate location had to be centered on the object, due to the geometry of the descriptors. This caused an issue when objects did not have points of interest near the center of the object. These two factors favored a grid approach in the candidate search portion of the algorithm.

A dense grid of candidates that was centered on the previous location of the object was used to determine candidate locations. The grid dimensions were determined by two parameters: the size and spacing of the candidates, which depend on desired accuracy of the tracker and the anticipated magnitude of the inter-frame motion. The grid is a square with length and width of $2r + 1$, where $r$ is the radius parameter. An example grid with a radius of 20 and spacing between candidates of one pixel can be seen in Figure 5. Although this method was simpler than using an interest point detector, it generated much better results. This approach relies on an assumption of smooth motion between consecutive frames. The smooth motion assumption is common for trackers, as the frame
rate of most modern cameras, 30 FPS and up, reduces the potential for large object movement between frames.

A downside of such an approach is the large number of candidate locations. As the search radius increases, the number of candidates increases on the order of \((2r + 1)^2\). The initial approach to mitigate the problem was to increase the spacing between the candidates in the grid. However, this approach negatively affected the accuracy of the tracker and increased the amount of “jitter” introduced. Thus, an alteration to the candidate grid was put forth by Henry Spang in his work [45], which suggested using a fine-to-course grid pattern. In this pattern candidates are selected close together if they are within half of the radius of the grid. Once the points are outside half of the radius, the spacing increases to twice the inner spacing. An example of such a grid is in Figure 6; this grid had a radius of 13 and a spacing of 1 and 2 for fine and course spacing, respectively. This allowed for fine grained measurements of position for small motions while maintaining the ability to capture large motions. This method greatly reduced the number of candidates required to cover the same area from 1681 in the example to 729, without sacrificing much accuracy. This approach has greatly improved the performance of the tracker and decreased the total number of computations required.
### 3.1.1.3 Locality Constraints

The last stage of the algorithm is the selection of the best candidate. Initially candidate selection is done strictly based on the candidate with the lowest Hamming distance. However, the results from such an approach exhibit two undesirable behaviors. The first was a tendency of the tracker to “jitter”. This artifact is traditionally mitigated through use of a motion model such as in [46]. A motion model such as a predictive Kalman filtering [47] introduced both a higher complexity to the algorithm as well as additional parameters to tune; thus it was not selected for use in the proposed algorithm. Instead, a simpler approach was used that relied on the fact that most inter-frame motion would be relatively small and would have a probability density function resembling a Gaussian function. Thus, a locality constraint weighting function was implemented that was based on an inverted Gaussian distribution, penalizing candidates further away from the previous location. The weight for a giving point at the location \((x,y)\) can be found using:

\[
W_I(x,y) = M \times \left(0.5 - \exp \left\{- \left(\frac{(x-x_0)^2 + (y-y_0)^2}{2\sigma^2}\right)\right\}\right)
\]  

(3.1)

where the center of the region is \((x_0,y_0)\), \(M\) the magnitude of the locality weighting function, and \(\sigma\) is the sigma of the distribution. A plot of an example distribution can be seen in Figure 7. The weighting function significantly reduced the potential for the tracker to “jitter” thus increasing the overall accuracy of the tracker.
3.1.1.4 Candidate Scoring and Selection

A significant portion of the decreased computation complexity of the proposed solution is found in the candidate scoring section. The scoring of candidates is the part of the algorithm that requires the most computations. The proposed algorithm minimizes the complexity of these computations by leveraging an advantage of binary features. Tracking algorithms that use traditional floating-point feature descriptors, such as SURF Tracker [20] and DML [6], rely on computationally intense distance metrics between descriptors such as the Euclidian distance. In contrast, the proposed solution uses binary features, which can be compared using the Hamming distance. The Hamming distance can be computed by

\[
d(f, g) = \sum_{i=0}^{n-1} [f(i) \neq g(i)]
\]

where \(f\) and \(g\) are the binary strings to be compared and \(n\) is the number of bits in the string. The Hamming distance can be easily computed by first applying the exclusive OR
function between two strings, and then counting the number of bits in the resulting string that are set to 1.

After the Hamming distance is computed between each candidate descriptor and every dictionary template, a bias is added to the Hamming distances, scores, of all comparisons to the dynamic dictionary templates. This is done so that the tracker will favor matches to the static dictionary, reducing the chance for drift. This static bias term is represented in the following equation

\[ B_s^{(i)} = \begin{cases} b_s & i \in D \\ 0 & i \in S \end{cases} \]  

(3.3)

where \( i \) is the dictionary template that is used in the comparison, \( D \) is the set of dynamic dictionary entries, \( S \) is the set of static dictionary entries, and \( b_s \) is the static bias value.

The overall score for each candidate is then determined by finding the dictionary template that is the closest match while taking the static bias, \( B_s^{(i)} \), into account. The procedure for finding the candidates score is as follows

\[ S_j = \argmin_{t_i \in \{S \cup D\}} \{ d(c_j, t_i) + B_s^{(i)} \} \]  

(3.4)

where \( S_j \) is the score for candidate \( j \), \( d(c_j, t_i) \) is the Hamming distance as described above, and \( B_s^{(i)} \) is the static bias as described above.

Once the overall scores of the candidates are computed, the candidate with the lowest total score is selected as the winner. The equation for the selection process is the following

\[ S_w = \argmin_{S_j \in C} \{ S_j + W_t(j) \} \]  

(3.5)
where \( S_w \) is the winning score for all of the candidates, \( S_j \) is candidate score as described above, and \( W_l(j) \) is the locality constraint weight as described in Section 3.1.1.3.

The winning score is compared against the threshold value that determines if the candidate is a valid match to the object. If the score is above the threshold, the results from the frame are discarded, and the object is classified as lost. The next frame then repeats the process with the previous location of the object. Otherwise the template is added to the dynamic dictionary, and the object location is updated.

3.2. **Multi-Part Object Tracking**

One of the contributions of this thesis is the ability of the tracker to track individual parts of an object in coordination with the entire object. One of the primary applications of the parts tracker is to track parts of a face, such as eye nose and mouth. However, it was determined that these parts can experience a wide range of appearance changes to which the tracker could not adapt if parts are tracked individually. A common example of such a drastic change is a blink of the eye. Blinking is a natural behavior that any facial tracker must be able to accommodate. Example frames from a blink sequence can be found in Figure 8. However, as Figure 9 demonstrates, the change in the descriptor can be drastic for a simple blink. If parts of an object were to be tracked as separate objects, then performance would be suboptimal. The proposed algorithm leverages the information acquired from the overall object motion to facilitate the accuracy of tracking its parts. This approach relies on the motion of the parts of object remaining relatively rigid between frames, as is commonly the case for video sequences. Figure 10 depicts the program flow for the proposed parts tracker. The flow for tracking the object remains
unchanged with the exception that the updated location of the object is used to move the search areas of the individual parts. The procedure for tracking each part is the same as that for tracking objects, except for the location update.

Figure 8: Example Frames from a Video of a Blink

Figure 9: Plot of the Percent Difference of Descriptors over Blinking Sequence. A Blink Occurs from Frame 8 to Frame 12.
3.3. Selecting Local Binary Descriptors

Local binary features have demonstrated robustness and large increase in computation speed. Additionally, local binary features have the added benefit of a very small memory footprint. Binary descriptors often need only 32 to 64 bytes of memory per descriptor, as opposed to the floating-point descriptors such as SIFT [13] and SURF [14] which require 128 or 512 bytes. Thus, local binary features were selected as the descriptors to be used in this tracking algorithm. Henry Spang investigated the uniqueness properties of descriptors that lead to optimal tracking behavior in [45]. Spang
suggests that the BRISK and FREAK descriptors generated optimal tracking behavior, while BRIEF and ORB also generated promising results. This information was used to select the BRIEF and BRISK descriptors used in this tracker, based on their simplicity and accuracy in experiments. An additional factor in selecting these descriptors was that they both could be implemented easily on a GPU. The FREAK descriptor was not used in this implementation, even though it shows high potential for tracking, because of its higher complexity. This tracking algorithm is designed to work independently of the specific descriptor selected, and any binary descriptor may be used, as long as its GPU implementation is available. In order to demonstrate the uniqueness and justify the selection of the descriptors, a test was performed where the descriptors for a local region are compared to the descriptor of the center point. The discrepancy between the descriptors is then plotted in terms of the Hamming distance. The ideal behavior for a descriptor in the test is to achieve clear discrepancy between the center point and its neighbors, which would be demonstrated by a narrow spike in the matching error surface. The ideal descriptor should have a narrow spike at the center, but also show some degree of similarity between the neighbors based on distance from the center. The plots showing matching error surfaces for various descriptors can be found in Figure 11 to Figure 14.
The BRIEF descriptor is the simplest descriptor implemented in this work. Even though the BRIEF descriptor was not the best in terms of accuracy, its simplicity allows for much faster execution of the tracker. The BRIEF descriptor has the option of a 32-byte implementation or a 64-byte implementation, the 32-byte descriptor allows for an even greater increase in speed of the tracker. The result of the local neighborhood tests for the 32-byte and 64-byte BRIEF descriptors can be seen in Figure 11 and Figure 12, respectively. These plots demonstrate a high degree of uniqueness, which is demonstrated by a high spike at the origin.

A descriptor that has shown high promise is the BRISK descriptor. This descriptor is designed to be more invariant to scale and rotation changes. Figure 13
demonstrates that this invariance reduces the uniqueness of the descriptors, as the width of the spike around the center is much wider. In an effort to prevent over generalization and create a more computationally efficient descriptor, an attempt has been made to remove the invariance behavior of the BRISK descriptor. The goal of invariance to rotation and scale largely stems from the image registration environment for which local descriptors were developed. However, in object tracking the uniqueness of a descriptor is more important than invariance. The changes in scale and size between frames in a video tend to be small enough that a tracker can easily adapt. A more generalized descriptor results in a higher likelihood that the tracker might experience a problem with drifting from the object.

### 3.3.1 Simplified BRISK descriptor

The new descriptor that is proposed in this thesis is a simplified version of the BRISK descriptor (SBRISK). The SBRISK descriptor removes the rotate and scale invariance properties from the BRISK descriptor in order to improve its uniqueness and computational efficiency. Additionally, the original BRISK descriptor uses a more accurate Gaussian smoothing technique for the averaging of the local region for the descriptor’s comparisons. The Gaussian smoothing was removed in favor of a pure averaging approach, because Gaussian smoothing was more complex and there was no significant added benefit that could be determined. The plot in Figure 14 shows that these changes significantly increase the uniqueness of the descriptor without removing the relative similarity to the descriptor’s neighbors. This makes the SBRISK descriptor a
good compromise between the overly simple BRIEF descriptor and the more general BRISK descriptor.

The SBRISK implementation leverages Integral Images to accelerate some of its calculation. Integral images were first presented by Viola and Jones in [48], where they proposed integral images as a way of accelerating the process of averaging the intensity values in a given region. The integral image is generated by summing the rows and columns of an image from the top left to bottom right. Each value in the integral image represents the sum of all of the values to its left and above its position. This representation can greatly accelerate operations where a large patch of pixels is to be averaged. Instead of having to sum all of the pixels in the area, only the values at the four corners of the image are necessary, which greatly reduces the number of memory accesses that are necessary.

3.4. **GPU Implementation**

The GPU implementation of the tracking algorithm was split into five kernels in order to maximize the potential parallelism of each section and increase the potential modularity of the tracker as a whole. The five kernels were Integral Image Kernel, Description Kernel, Scoring Kernel, Best Dictionary Kernel, and the Best Candidate Kernel. Figure 15 depicts how the GPU implementation progresses from transferring the image onto the GPU to the Integral Image generation Kernel. The Integral Image Kernel is called only for SBRISK descriptors that leverage integral images to accelerate their calculations when performing averaging operations. Descriptors such as BRIEF do not need to run the Integral Image kernel. After the Description Kernel generates the
The first kernel in the GPU pipeline of operations is typically the Integral Image Kernel. This kernel is only run for SBRISK, the only descriptor implemented that uses integral images. The Integral Image Kernel has the lowest potential for parallelization because of the sequential nature of the summation process. However, the kernel is designed so that the maximum number of threads can operate in parallel. The number of threads that operate on the integral image is based on the largest dimension of the image. Each thread in the kernel first accumulates the totals for each of the columns of the image and stores them in a new image. Next, each thread accumulates the sum of each row of the new image. This leverages as much parallelism as possible in the integral image generation step.

The next kernel is the descriptor generation kernel. The exact kernel that is run is determined by the descriptor being used, as separate descriptors require separate generation processes. In order to maintain consistency within the design of the threading structure, all of the descriptor kernels use the same approach for configuration of their
threads. The dimension of each block is dependent on the optimal number of threads per block for the given card (denoted by $t$) and the size of the descriptor in number of bytes (denoted by $b$): the $X$ dimension is $t/b$, and the $Z$ dimension is $b$. The grid dimensions are dependent on the size of the block in the $X$ dimension (Block.$x$), number of objects (denoted by $o$) and the number of candidates (denoted by $c$); the $X$ dimension is the $c$/Block.$x$, and the $Y$ dimension is $o$. The concept behind such a threading strategy is to maximize the number of operations that can be done independently. Although there is potential for each of the 256 or 512 comparisons to be made by separate threads, each of the threads has to contend for memory access to write its results. This presents a major bottleneck for the system. To overcome this limitation, the kernels are designed to operate on separate bytes, with eight comparisons associated with each byte. Thus, there is no contention for writing results to memory. This configuration allows for a large number of threads, $O(ocb)$, which can operate without interference.

Kernels are implemented to generate both BRIEF and SBRISK descriptors. The designs of the kernels were relatively similar, thanks to the threading configuration. However, the BRIEF kernel is much simpler than the SBRISK kernel. The BRIEF descriptor requires a set of 256 or 512 predetermined comparisons, for BRIEF32 and BRIEF64 respectively. A lookup table in constant memory was used to determine the locations of the pixels to be compared. Each thread would make the eight comparisons for the byte assigned to it, and then store the result in the candidate’s descriptor.

The SBRISK descriptor was much more complicated to implement because it relied on a set of comparisons between 60 different areas around the center of the descriptor. In order to accelerate the generation of the descriptor, the values of each of the
60 areas were calculated and stored in shared memory for access by all of the threads in the block. Once the values for the 60 regions were obtained, each thread determined the comparisons to make based on a predetermined set stored in constant memory. Each thread made the eight comparisons for the byte assigned to it and stored the result in the candidate’s descriptors.

After the descriptors for each of the candidates are generated, the candidates are compared to each of the entries in the template dictionaries using Hamming distance to determine the score of each comparison. This portion of the algorithm presents the highest potential parallelism of all of the kernels because of the number of independent comparisons that need to be made. The threading configuration was designed such that each thread would compare one byte from a single candidate descriptor to the corresponding byte in a single template in the dictionaries. Each thread computes the Hamming distance between two bytes by a bit-wise exclusive OR operation using the results to index into a lookup table with pre-calculated distances. The results were then accumulated in global memory. This approach presents the possibility of some contention; however the high parallelism overcame such overhead. This approach generated \(O(ocbd)\) threads, where \(o\) is the number of objects, \(c\) is the number of candidates for each object, \(d\) is the number of dictionaries, and \(b\) is the number of bytes in each dictionary.

The next kernel calculates the best matching entry in both the static and dynamic dictionaries for each of the candidates. This kernel experiences a reduction in parallelism, as the number of potential matches gets reduced to the final result of one best match. The goal of this kernel is to select the static and dynamic dictionary template that is the
closest match to each of the candidates. This operation is done by using a parallel reduction algorithm that leverages shared memory. The scores for the descriptors are loaded into shared memory, and at each iteration, the threads compare two elements in shared memory and determine the maximum. As the algorithm progresses, the number of active threads decreases by half, as half of the data get discarded with each iteration. This reduction algorithm requires the number of dictionaries to be a power of two for optimal operation. The number of threads that are active at the beginning of this kernel is $O(ocd)$, where $o$ is the number of objects, $c$ is the number of candidates for each object, and $d$ is the number of dictionaries.

The last kernel selects the candidate that is the closest match to one of the templates in the dictionary and updates the dictionary. Due to the varying number of candidates and the impossibility of requiring the number of candidates to be a power of 2, a two-part approach is taken instead of the approach used in the dictionary selection kernel. In the first part, the candidates are broken into equal parts for each of the threads, and the local minimum is found. The kernel takes into account the weighting bias when selecting the minimum. Once each section’s local minimum is determined, a single thread selects the global minimum of all of the candidates; this is the best match. After the best match is selected, the score of the match is compared to the threshold value to determine if the candidate is to be considered a match. If the candidate is a match, the dynamic dictionary is updated with the new template, and the new location is stored.
3.5. Mobile Implementation

Mobile devices represent one of the fastest growing fields in computer vision due to their increasing processing power and the increasing quality of their cameras. Thus, they present an attractive platform for an efficient tracking method used in applications such as augmented vision. The proposed tracking method has demonstrated both accuracy and efficiency such that it has high potential for positive results on mobile devices. An implementation of the algorithm on the Android platform was proposed by Quraishi [49]. However due to the requirements of the Android environment the implementation required the use of the Java Native Interface (JNI) to act as the intermediary between C/C++ portions of the code and Java portions. This requirement introduces a large overhead to a processor that is already fully utilized. The iOS environment did not have such drawbacks, because all applications developed for iOS are written in Objective-C, which is an extension of C/C++. The iOS implementation presented the following advantages. The first advantage was that the proposed tracker could be directly ported to the iOS environment with only minor alterations made, which significantly reduced development time. Secondly, because the application was entirely written in Objective-C, there was no bottleneck introduced by communications between multiple programming languages. Motivated by these advantages, an application was written to demonstrate the potential performance of the tracker in the iOS environment.
Chapter 4  Experimental Results

In order to obtain optimal results from the proposed tracking algorithm there are multiple parameters that must be tuned. The details of this process are discussed in Section 4.2 In this chapter the performance of the tracker was evaluated in terms of both the accuracy of the tracker as well as the speed at which the tracker can track. A very accurate tracker that cannot operate close to real time is not useful for modern applications, such as Human Computer Interaction. Conversely, it does not matter how fast a tracker is, if it cannot accurately track objects. Thus, tracker experiments were run to compare both the tracker’s accuracy and speed in comparison to other methods. The accuracy experiments and their results are discussed in Section 4.3, and the experimental results of the speed tests can be found in Section 4.4. Additionally experiments on the proposed parts tracking model are discussed in Section 4.5.

4.1.  Experimental Setup

4.1.1  Tracking Source

The tracker was designed to operate on either a live feed from a camera or pre-recorded videos. This was done to accommodate both prerecorded experiments and live video. The tracker uses the OpenCV API [50] for the video input; thus, it can handle any video format supported by OpenCV. The tracker was tested on videos ranging in resolution from 160x120 to 1920x1200 (1080p).
4.1.2 Initialization

The algorithm is designed to be as modular as possible for easy initialization. Due to the wide range of potential applications for the algorithm, the goal of the design of the initialization step was to allow for any number of different inputs for initializing the tracker. The tracker needs only the location and size of the object for the first frames until the static dictionary is filled. The tracker handles the rest of the initialization procedure internally. This design choice accommodates the different sources that are often used in conjunction with object trackers: ground truth/initialization files, user selection, and object detection. Ground truth or initialization files are often used on selected videos as a means to test the tracker’s accuracy. User selection is a useful method of initialization, because it allows the user to determine the exact object that is to be tracked; thus the tracker does not need to have prior knowledge of the object’s appearance. The downside of such an initialization strategy is that it requires manual user interaction which may get tedious as the number of objects grows large. The third initialization strategy was to use a pre-trained detector for the desired object to be tracked. This restricts the general applicability of the tracker to known objects; however it automates the initialization and increases the utility of the tracker. The automatic detection approach described by Savakis et al. in [51] has been paired with this tracker and yielded positive results.

4.1.3 Datasets used

The datasets used in the experiments were supplied by MIL [22], Wu et al. [53], and VTD [52]. The results used for comparison between the proposed tracker and other trackers were supplied by [22] and [53]. Each of these datasets provided a wide range of
videos that could test the tracker’s robustness to changes in object appearance as well as occlusions. The MIL dataset provided low definition videos that were designed to test object tracking; thus most of the videos are from laboratory environments with controlled conditions. It is important to note that the ground truth locations are supplied for every fifth frame for the MIL dataset; thus some of the error plots are more sparse than those of other datasets. The videos from the VTD dataset were mostly from television broadcasts, thus they are of a higher resolution and have real world environmental conditions. The VTD dataset is a much harder dataset to track, requiring trackers to maintain robustness to changes in appearance while not drifting, which is a difficult balancing act.

4.2. Parameter Selection

The first step in benchmarking the tracker was to select values for its parameters based on optimal accuracy. It was determined that there are six key parameters that impact the behavior of the tracker: Descriptor Type, Search Radius, Locality Constraint Distribution, Locality Constraint Magnitude, Matching Threshold Value, and the Static Dictionary Bias. Due to the nature of the tracker, these parameters cannot be tuned independently, because of the dependency of certain parameters on each other. Additionally, the performance of the tracker did not show a consistent trend to a single optimal point in parameter space, and thus a gradient descent method could not be used to automate the parameter selection. Instead, the tracker was run through a series of automated tests over a full parameter sweep for the six parameters. The parameter sweeps were run over a set of 25 test videos, each with unique objects to track.
The first parameter tested was the type of descriptor used to describe the object. The descriptor selected for tracking plays a large role in the behavior of the tracker based on its inherent qualities. Some descriptors are more invariant to rotation or scale while others are faster. Four descriptors were selected for testing: 32-Byte BRIEF (BRIEF32), 64-Byte BRIEF (BRIEF64), BRISK, and SBRISK. BRIEF32 and BRIEF64 were selected in order to determine the impact that the BRIEF descriptor size had on the accuracy of the tracking. Additionally BRIEF is a simple descriptor, and thus it could lead to faster tracking performance. BRISK was selected because of its relative simplicity as compared to other descriptors such as FREAK. Yet, the accuracy observed with BRISK was on par with the initial results from FREAK. The SBRISK descriptor removed the invariance to rotation and scale of the BRISK descriptor in order to simplify the descriptor and increase distinctness. These four descriptors were tested over all of the parameter sweeps, and the best parameter set was selected for each one.

The Search Radius was the second parameter in the parameter set. The search radius plays an important role in the tracker’s behavior because it determines the area over which the search for the object takes place. If the search area is too small, fast moving objects are lost because the motion of the object may be larger than the search radius. Conversely, if the search radius is too large, the tracker slows down and is more likely to find false positives in the background. Figures 16 and 17 demonstrate how the descriptor does not always demonstrate the ideal uniqueness as the search area increases beyond the local region. These figures depict the difference between the distances of the candidates to the dictionary, as the search radius changes from 20 in Figure 16 to 40 in Figure 17. Additionally, a larger search radius increases the number of computations that
must be performed for each frame; thus a smaller radius is better with respect to the number of computations necessary. Six different search radii, 10, 15, 20, 25, and 30, were tested in the parameter sweep. All of the search windows had a candidate spacing of one pixel for the dense search region and two pixels for the sparse search region.

A parameter that is closely tied to the search radius is the Locality Constraint Distribution, because it is a function of the search radius. Thus, as the search radius is altered, the variance of the Gaussian Locality Constraint function changes. Increasing and decreasing the sigma parameter widens and narrows the Gaussian distribution of the locality constraint. If the sigma factor is too small, the distribution is too narrow, so the tracker may not keep up with the motion of the object and the object may be lost. If the sigma factor is too large, the tracker might demonstrate a large amount of jitter and drift that may lead to losing the object. Thus, it is important to select a suitable sigma factor. The sigma factors that were tested in the parameter sweep were: 0.25, 0.5, 1.0, 1.5, 2.0, and 3.0.

Another parameter that is related to the search radius is the Magnitude of the Locality Constraint weighting function, which determines how much the locality constraint will affect the scores of the candidates. If the magnitude is small, the locality
constraint will not have much impact on the selection process, leading to a higher potential for the tracker to jitter and drift. If the magnitude of the locality constraint is too large, the tracker will favor not moving from the previous location and tracking of the object may be lost. Thus, it is important to tune the magnitude to reach to optimal tradeoff between the two extremes. The locality constraints magnitudes that were tested in the parameter sweep were: 20, 35, 50, 60, 80, and 100.

The next parameter that was considered was the Matching Threshold Value. This parameter determines what score is deemed a match. This factor was crucial for the tracker to mitigate full or partial occlusions. If the threshold is too low, the tracker is more likely to misclassify a correctly tracked object as a lost object, resulting in false negatives. However, if the threshold is too high, the tracker is more likely to classify incorrectly tracked objects as tracked, resulting in false positives. As the threshold value increases, so does the tracker’s ability to adapt to new appearances of the object. The threshold values that were tested were: 50, 80, 100, 120, 150, and 200.

The last parameter that was tested was the Static Dictionary Bias. This parameter influences the tracker’s preference for matches with the static dictionary. It was used as a means to control the drift of the tracker by favoring candidates that are closer to matches with templates in the static dictionary. However, this bias should not be too large because it would hamper the tracker’s ability to adapt to new appearances of the object. Thus, it is important to find the balance of a bias towards the static templates, but not so large as to prevent adaptation. The static dictionary bias values that were tested were: 10, 20, 30, 50, 80, and 100.
The test was automated using a windows batch script that outputs the results from each test to a series of Comma Separated Values (CSV) files. Each test would output the Euclidian Distance of the tracked object center from the supplied ground truth file. The CSV files were parsed using a python script which aggregated the results from all of the tests into a single CSV file which was imported into Microsoft Excel for plotting and analysis.

The first step in the accuracy analysis of the proposed tracker was to determine the optimal results that could be achieved with the descriptors for each of the videos in the test set. This determination was made by finding the parameter set in the entire sweep that had the minimum error for each of the videos. A table containing the exact parameter set for each video can be seen in Table 1. In this table R represents the search radius, M represents the magnitude of the locality constraint, S represents the sigma value for the locality constraints, T represents the threshold value, and B represents the bias towards the static dictionary. The results from these parameters are shown in the columns of Table 3 labeled Best. The results for the optimal parameter sets demonstrate very high quality tracking which significantly outperforms all other trackers on every video except the Tiger 1 video. These results are important because they demonstrate the maximum potential accuracy of the tracker if it is tuned for a given video type. This could be done in cases where the environment is set and there is a priori knowledge about the type of object and its motion.
After the optimal parameter set was selected, a single unified parameter set had to be selected for all of the videos. This is because often there is no a priori knowledge about the object to be tracked, and a general parameter set must be used. This results in worse performance for each of the videos than the optimal set; however, it is important to have a tracker that can track any potential object. Multiple approaches were taken to determine the best general parameter set. The first approach was to select the parameter set with the minimum average error for all of the videos. This approach led to mediocre tracking for all of the videos, because some of the poorly performing videos skewed the
results. After the mean approach failed, a new approach was adopted that found the best parameter set across the best performing videos. This approach generated much better results because the videos that could not be tracked well by the tracker did not overly skew the results. The best general parameter set for each of the descriptors can be found in Table 2. The average distance from the ground truth with these parameter sets can be found in the General columns of Table 3.

Table 2: General Parameter Set for Descriptors

<table>
<thead>
<tr>
<th>DESCRIPTOR</th>
<th>SEARCH RADIUS</th>
<th>LOCALITY MAGNITUDE</th>
<th>LOCALITY SIGMA</th>
<th>THRESHOLD VALUE</th>
<th>STATIC BIAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRIEF32</td>
<td>25</td>
<td>20</td>
<td>0.5</td>
<td>80</td>
<td>20</td>
</tr>
<tr>
<td>BRIEF64</td>
<td>20</td>
<td>100</td>
<td>2</td>
<td>120</td>
<td>20</td>
</tr>
<tr>
<td>BRISK</td>
<td>25</td>
<td>20</td>
<td>0.25</td>
<td>150</td>
<td>20</td>
</tr>
<tr>
<td>SBRISK</td>
<td>20</td>
<td>50</td>
<td>0.5</td>
<td>150</td>
<td>30</td>
</tr>
</tbody>
</table>

The last row of Table 3 shows the average error over all of the videos for each descriptor. It is clear from the average errors that the descriptor which performs the worst, out of the three fully implemented descriptors with a unified parameter set, is the 32-byte BRIEF descriptor. This is to be expected, because BRIEF32 is the simplest descriptor with the smallest descriptor, leading a less unique description of the image patches. The second best descriptor for the unified parameter set is the 64-byte version of the BRIEF descriptor. Again, this is expected because of the simplicity of the descriptor. The best performing descriptor out of the three, for both the unified parameter set and the optimal sets for each video, was the SBRISK descriptor. The SBRISK descriptor preformed significantly better both for the optimal parameter sets and for the unified parameter set. This significant improvement demonstrates the potential for the SBRISK descriptor. All of the results in Table 3 demonstrate the high potential of the tracker.
They show that if a priori knowledge exists for the object to be tracked, extremely accurate tracking can be achieved. If, however, there is no prior knowledge for the object to be tracked, Table 3 demonstrates that highly accurate tracking can still be achieved.

Table 3: Comparison of Average Distance to the Ground Truth of the Optimal and General Parameters

<table>
<thead>
<tr>
<th>VIDEO</th>
<th>BEST</th>
<th>GENERAL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BRIEF32</td>
<td>BRIEF64</td>
</tr>
<tr>
<td>Animal</td>
<td>8.81</td>
<td>8.76</td>
</tr>
<tr>
<td>Basketball</td>
<td>16.57</td>
<td>14.55</td>
</tr>
<tr>
<td>Cliffbar</td>
<td>3.46</td>
<td>3.39</td>
</tr>
<tr>
<td>Coke11</td>
<td>15.24</td>
<td>15.37</td>
</tr>
<tr>
<td>David</td>
<td>3.87</td>
<td>4.03</td>
</tr>
<tr>
<td>Dollar</td>
<td>3.05</td>
<td>2.80</td>
</tr>
<tr>
<td>Football</td>
<td>8.36</td>
<td>8.48</td>
</tr>
<tr>
<td>Girl</td>
<td>15.57</td>
<td>20.89</td>
</tr>
<tr>
<td>Gymnastics</td>
<td>10.92</td>
<td>10.16</td>
</tr>
<tr>
<td>High Jump</td>
<td>40.00</td>
<td>56.55</td>
</tr>
<tr>
<td>Occluded Face 1</td>
<td>6.02</td>
<td>6.60</td>
</tr>
<tr>
<td>Occluded Face 2</td>
<td>7.38</td>
<td>5.23</td>
</tr>
<tr>
<td>Shaking</td>
<td>9.82</td>
<td>9.06</td>
</tr>
<tr>
<td>Singer1</td>
<td>7.09</td>
<td>8.38</td>
</tr>
<tr>
<td>Singer2</td>
<td>10.23</td>
<td>13.20</td>
</tr>
<tr>
<td>Skating1 LFR</td>
<td>9.96</td>
<td>9.55</td>
</tr>
<tr>
<td>Skating2</td>
<td>25.74</td>
<td>25.80</td>
</tr>
<tr>
<td>Surfer</td>
<td>3.77</td>
<td>5.44</td>
</tr>
<tr>
<td>Sylvester</td>
<td>5.13</td>
<td>4.72</td>
</tr>
<tr>
<td>Tiger1</td>
<td>13.25</td>
<td>16.16</td>
</tr>
<tr>
<td>Tiger2</td>
<td>11.73</td>
<td>11.94</td>
</tr>
<tr>
<td>Transformer</td>
<td>18.51</td>
<td>20.02</td>
</tr>
<tr>
<td>Average</td>
<td>12.17</td>
<td>13.226</td>
</tr>
</tbody>
</table>
4.3. **Accuracy Tests**

4.3.1 **Accuracy Comparisons between Trackers**

In order to assess the performance of the proposed tracker, its accuracy was benchmarked with respect to other trackers on a set of standard videos. The accuracy results reported in Table 4 show the average distance of the tracker’s result in comparison to the ground truth. The trackers that are used for comparison are the TLD tracker [4] and the MIL tracker [22]. The results for the MIL and TLD trackers are based on datasets that were provided from Wu et al. [53] where they tested a wide range of trackers on a large number of standard videos and made available the resulting location from each tracker for every frame. These results were used instead of implementing the trackers and testing the videos directly because of difficulties in setting up both MIL and TLD trackers. The results in Table 4 are based on tests run with the parameter set for each of the descriptors that was selected in the parameter sweep. These results are consistent with a real world application, where the tracker cannot be tuned for each video sequence. These videos consist of a wide variety of objects being tracked under many different circumstances, and thus, they are representative of most applications for the tracker. The accuracy of the tracker is well represented in this table; **bold** numbers signify best tracking results by the proposed tracker compared to the other trackers. The BRIEF32 descriptor performs better than MIL and TLD for 8 of the 12 videos, the BRIEF64 descriptor performs better for 6 of the 12 videos, the BRISK descriptor performs better for 7 of the 12 videos, and the SBRISK descriptor performing better for 9 of the 12 videos. Some video sequences have been selected which exemplify challenging circumstances where other trackers fail, yet
the proposed tracker performs well. Example frames from these four sequences can be found in Figure 18. A description of each sequence, as well as the associated challenges, is described in this section.

### Table 4: Comparison of Average Distance to Ground Truth in Pixels of Trackers

<table>
<thead>
<tr>
<th>VIDEO</th>
<th>MIL</th>
<th>TLD</th>
<th>BRIEF32</th>
<th>BRIEF64</th>
<th>BRISK</th>
<th>SBRISK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bolt</td>
<td>453.02</td>
<td>393.54</td>
<td>139.51</td>
<td>207.75</td>
<td>69.72</td>
<td>28.82</td>
</tr>
<tr>
<td>Car4</td>
<td>15.48</td>
<td>50.78</td>
<td>62.98</td>
<td>45.21</td>
<td>4.94</td>
<td>21.64</td>
</tr>
<tr>
<td>CarDark</td>
<td>27.47</td>
<td>43.48</td>
<td>31.99</td>
<td>31.06</td>
<td>16.57</td>
<td>4.95</td>
</tr>
<tr>
<td>Cliffbar</td>
<td>11</td>
<td>34.62</td>
<td>5.60</td>
<td>5.21</td>
<td>7.92</td>
<td>4.84</td>
</tr>
<tr>
<td>Coke11</td>
<td>20.13</td>
<td>16.74</td>
<td>30.11</td>
<td>47.52</td>
<td>55.06</td>
<td>32.86</td>
</tr>
<tr>
<td>David</td>
<td>23.12</td>
<td>11.27</td>
<td>11.43</td>
<td>11.06</td>
<td>7.17</td>
<td>5.88</td>
</tr>
<tr>
<td>David2</td>
<td>4.98</td>
<td>10.93</td>
<td>2.41</td>
<td>3.62</td>
<td>3.40</td>
<td>4.32</td>
</tr>
<tr>
<td>Deer</td>
<td>86.75</td>
<td>100.54</td>
<td>9.28</td>
<td>10.41</td>
<td>9.87</td>
<td>9.91</td>
</tr>
<tr>
<td>Dog1</td>
<td>4.19</td>
<td>7.83</td>
<td>7.71</td>
<td>9.16</td>
<td>4.37</td>
<td>6.83</td>
</tr>
<tr>
<td>Dollar</td>
<td>14.74</td>
<td>68.06</td>
<td>3.51</td>
<td>2.88</td>
<td>4.25</td>
<td>11.47</td>
</tr>
<tr>
<td>Fish</td>
<td>9.38</td>
<td>24.14</td>
<td>5.13</td>
<td>4.65</td>
<td>4.86</td>
<td>6.64</td>
</tr>
<tr>
<td>Football</td>
<td>15.89</td>
<td>13.41</td>
<td>15.51</td>
<td>15.15</td>
<td>16.19</td>
<td>15.97</td>
</tr>
<tr>
<td>Freeman3</td>
<td>29.73</td>
<td>87.56</td>
<td>10.45</td>
<td>7.11</td>
<td>5.78</td>
<td>6.84</td>
</tr>
<tr>
<td>Jumping</td>
<td>5.94</td>
<td>9.99</td>
<td>73.55</td>
<td>14.69</td>
<td>67.53</td>
<td>21.83</td>
</tr>
<tr>
<td>Mhyang</td>
<td>9.51</td>
<td>20.40</td>
<td>4.42</td>
<td>5.16</td>
<td>5.65</td>
<td>4.08</td>
</tr>
<tr>
<td>Mountain Bike</td>
<td>213.62</td>
<td>73.02</td>
<td>6.82</td>
<td>5.16</td>
<td>7.20</td>
<td>6.66</td>
</tr>
<tr>
<td>Occluded Face 1</td>
<td>27.23</td>
<td>23.47</td>
<td>22.42</td>
<td>30.52</td>
<td>16.38</td>
<td>11.14</td>
</tr>
<tr>
<td>Occluded Face 2</td>
<td>20.19</td>
<td>19.91</td>
<td>17.20</td>
<td>9.65</td>
<td>7.70</td>
<td>13.11</td>
</tr>
<tr>
<td>Girl</td>
<td>52.21</td>
<td>19.77</td>
<td>74.34</td>
<td>104.65</td>
<td>114.05</td>
<td>135.56</td>
</tr>
<tr>
<td>Subway</td>
<td>159.35</td>
<td>7.60</td>
<td>146.33</td>
<td>148.26</td>
<td>135.98</td>
<td>138.96</td>
</tr>
<tr>
<td>Surfer</td>
<td>11</td>
<td>9.947</td>
<td>9.46</td>
<td>7.90</td>
<td>7.48</td>
<td>7.67</td>
</tr>
<tr>
<td>Sylvester</td>
<td>10.82</td>
<td>8.56</td>
<td>7.40</td>
<td>6.56</td>
<td>23.81</td>
<td>5.74</td>
</tr>
<tr>
<td>Tiger1</td>
<td>16</td>
<td>11.07</td>
<td>63.45</td>
<td>35.74</td>
<td>51.42</td>
<td>33.98</td>
</tr>
<tr>
<td>Tiger2</td>
<td>17.85</td>
<td>20.68</td>
<td>16.83</td>
<td>24.94</td>
<td>20.12</td>
<td>17.33</td>
</tr>
<tr>
<td>Trellis</td>
<td>31.60</td>
<td>71.47</td>
<td>29.69</td>
<td>31.34</td>
<td>46.21</td>
<td>14.30</td>
</tr>
<tr>
<td>Twinings</td>
<td>15</td>
<td>61.12</td>
<td>29.21</td>
<td>26.86</td>
<td>14.81</td>
<td>10.80</td>
</tr>
</tbody>
</table>
A common challenge for trackers is tracking an object through rotation and scale changes, because the object’s appearance varies widely through rotation and scale changes. A video which tests the tracker’s ability to adapt to such changes is the Cliffbar video; its example frames can be found in the top row of Figure 18. This video demonstrates rotation and scale changes and partial occlusion. A plot of the distances of the tracker’s location to the ground truth location can be seen in Figure 19. The plot shows that the SBRISK descriptor tends to outperform all other descriptors for this video.
Another of the challenging videos in the dataset is a video named David, with example frames shown in the second row of Figure 18. This video sequence follows a man as he moves from room to room with multiple lighting changes, out of plane rotations, and scale changes, as well as a change in appearance when he removes and puts on his glasses. Even with all of these challenges, the plot in Figure 20 shows that the tracker preforms very well. The MIL tracker has several spikes, whereas the tracker remains relatively low the entire time. The BRIEF and BRISK descriptors do drift to the top of his head toward the end of the video, but the SBRISK descriptor remains within 10 pixels during almost all of the video.
A challenge for some of the trackers occurs when there are multiple objects that look similar in a video. A video to test such behavior is the Dollar video, with example frames shown in the third row of Figure 18. In this video, the object being tracked changes appearance, then there is another object introduced that is an exact copy of the original object. Figure 21 demonstrates that the proposed tracker is not thrown off by such actions. The proposed tracker adapts to the new appearance of the object then continues to track it for the entirety of the video, unlike the MIL tracker which has an error that spikes at frame 45 when the objects appearance is changing. This video demonstrates how well the tracker can adapt and track an object in the presence of multiple objects identical to the original.
Another challenge for object trackers is to deal with partial occlusion of objects without drifting. A video sequence that tests the tracker’s ability to deal with partial occlusion is the Occluded Face 1 video, with example frames found in the fourth row of Figure 18. This video tracks a face while an object is placed in front of the face in many different positions. A plot of the distance between the tracking results and the ground truth can be found in Figure 22. This is a challenging video for the tracker, as it sometimes follows the object that is introduced, such as around frame 240 and 600. Once the face becomes visible, the tracker snaps back to the correct position. This video demonstrates the ability of the tracker to recover from drifting when the object reappears. There are several spikes for the tracker; however on the whole the tracker performs better than the MIL tracker for this video.
There are additional examples of video sequences where the tracker performs poorly in Table 4. The video sequences for which the tracker does not perform well are the Coke sequence, the Girl sequence, and the Tiger 1 Sequence. The Coke and Tiger 1 sequences both perform poorly because the object is very small and there is a large amount of occlusion in the videos. These factors often cause trackers to drift and lose the object. The girl video does poorly for all of the proposed methods. The Girl video is of low quality, and thus there is a significant amount of distortion and noise. There is full out-of-plane rotation of the person being tracked, and there is a significant motion of the Person face when she is turned around. These factors all lead to the poor performance of the tracker for the Girl video sequence.
4.3.2 Examples of well tracked videos

As a further demonstration of the accuracy of the tracker, additional tests were performed on more videos. The additional dataset had many more challenging video sequences in real world environments. This section discusses some of the videos for which the tracker performed well. Table 5 contains the average distance, in pixels, of each tracker for all of the videos in the set. Example frames are shown in Figure 23 for four of the video sequences that demonstrate potential challenges for all object trackers.

Table 5: Results of Well Tracked Videos in terms of Average Distance from Ground Truth in Pixels

<table>
<thead>
<tr>
<th>VIDEO</th>
<th>BRIEF32</th>
<th>BRIEF64</th>
<th>BRISK</th>
<th>SBRISK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Animal</td>
<td>9.28</td>
<td>10.41</td>
<td>9.87</td>
<td>9.91</td>
</tr>
<tr>
<td>Diving</td>
<td>87.35</td>
<td>97.92</td>
<td>15.54</td>
<td>14.71</td>
</tr>
<tr>
<td>Football</td>
<td>15.51</td>
<td>15.15</td>
<td>16.19</td>
<td>15.97</td>
</tr>
<tr>
<td>Singer 1</td>
<td>10.57</td>
<td>13.56</td>
<td>13.66</td>
<td>12.30</td>
</tr>
<tr>
<td>Singer 2</td>
<td>22.64</td>
<td>80.21</td>
<td>18.13</td>
<td>25.29</td>
</tr>
<tr>
<td>Skating 1</td>
<td>29.98</td>
<td>87.27</td>
<td>21.70</td>
<td>50.81</td>
</tr>
</tbody>
</table>

Figure 23: Well Tracked Example Frames. The Yellow Circles are the BRIEF32 Results. The Green Circles are the BRIEF64 Results. The Blue Circles are the BRISK Results. The Pink Circles are the SBRISK Results. The White Circles are the Ground Truth Locations.
Multiple objects, or a single object moving in a cluttered environment, often present a challenge for many state of the art trackers. A video sequence that would pose such a challenge for classification trackers is a video of deer crossing a river; its example frames can be found in the first row of Figure 23. This video contains multiple identical looking deer as well as fast motions. Even with these challenging circumstances, the plot of the distances from the ground truth for the video, found in Figure 24, shows that the trackers do a good job of tracking the deer for the entirety of the video, with some small error spikes that are quickly recovered.

![Animal Error Plot](image)

**Figure 24: Plot of the Euclidian Distance of the Trackers to the Ground Truth for the Animal Video**

A common application for object tracking is tracking athletes during a game. This presents multiple problems for object trackers because there are many players whose uniforms are almost identical, there are large numbers of object and occlusions, and there are usually many fast movements of the objects. An example of such a situation is a
video of a football game; its example frames can be found in the second row of Figure 23, where a player is tracked through a large number of other players on the field. This sequence is a very challenging sequence, yet, as the plot in Figure 25 shows, all of the trackers have shown a high level of accuracy until around frame 285, the frame when the player is completely occluded by an identical helmet. Except for the last section, this video shows the high accuracy that can be expected from the proposed tracker for tracking of athletes.

Another challenge for most trackers is the ability to adapt to changes in illumination of the object without losing it. An example of a large variation in illumination is shown in Singer 1 video. Example frames from the sequence can be found on the third row of Figure 23. In this sequence there is a significant lens flare around frame 120, which causes the image patch that is to be tracked to have a significant change...
in illumination. A plot of the distance between the tracking results and the ground truth can be found in Figure 26. As the plot shows, the trackers are affected by the change, but they still continue to track the object relatively accurately. This is because there is still some detail left in the image patch, so the descriptors adapt to the change in illumination. The benefit to the use of local binary descriptors is that their values are based on comparisons between two patches. If the illumination of the entire image changes, it does not affect the descriptors as long as there is enough definition to provide meaningful description.

![Singer 1 Error Plot](image)

**Figure 26: Plot of the Euclidian Distance of the Trackers to the Ground Truth for the Singer1 Video**

A video sequence that demonstrates the advantages of the more advanced BRISK and SBRISK descriptors is the video of a diver preforming a dive off a tall diving board. Example frames from this video can be found in the fourth row of Figure 23. This is a challenging video because there is a large amount of motion and change in appearance as
the diver performs multiple flips. Figure 27 shows how the two BRIEF descriptors drift away from the diver as they get stuck to the background. BRISK and SBRISK descriptors on the other hand track the diver for the entire video sequence. It is worth noting that the ground truth location in the video is based on the centroid of the diver and is not tracking the center point of the body. Thus, visual inspection demonstrates that the BRISK and SBRISK descriptors do a better job than what is represented in the plot, because the tracker more closely follows the driver thought the video than the centroid of the supplied ground truth.

![Diving Error Plot](image)

Figure 27: Plot of the Euclidian Distance of the Trackers to the Ground Truth for the Diving Video

### 4.3.3 Examples of poorly tracked videos

Although the tracker has demonstrated positive results for a large number of difficult challenges, there are still some circumstances that present major obstacles to the tracker. The tracker does a good job tracking objects through mild to moderate changes in
appearance, illumination, and scale; however when these changes become too large, the tracker fails. Once the object is lost due to such conditions, it is difficult for the tracker to recover if the object moves away from the search region. The average error for a set of videos in which the tracker performs poorly can be found in Error! Reference source not found.. Additionally, example frames from videos where the tracker fails can be found in Figure 28.

Table 6: Accuracy Results of Poorly Tracked Videos

<table>
<thead>
<tr>
<th>VIDEO</th>
<th>BRIEF32</th>
<th>BRIEF64</th>
<th>BRISK</th>
<th>SBRISK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basketball</td>
<td>190.91</td>
<td>173.23</td>
<td>192.33</td>
<td>247.42</td>
</tr>
<tr>
<td>Gymnastics</td>
<td>40.43</td>
<td>34.61</td>
<td>124.99</td>
<td>42.55</td>
</tr>
<tr>
<td>High jump</td>
<td>91.90</td>
<td>69.15</td>
<td>66.11</td>
<td>69.42</td>
</tr>
<tr>
<td>Shaking</td>
<td>23.45</td>
<td>42.74</td>
<td>63.25</td>
<td>83.71</td>
</tr>
<tr>
<td>Skating1 LFR</td>
<td>20.83</td>
<td>73.94</td>
<td>100.83</td>
<td>32.73</td>
</tr>
<tr>
<td>Skating2</td>
<td>205.82</td>
<td>196.19</td>
<td>211.71</td>
<td>214.13</td>
</tr>
<tr>
<td>Soccer</td>
<td>90.42</td>
<td>19.29</td>
<td>80.79</td>
<td>59.45</td>
</tr>
<tr>
<td>Transformer</td>
<td>45.63</td>
<td>42.62</td>
<td>33.53</td>
<td>37.60</td>
</tr>
</tbody>
</table>

Figure 28: Poorly Tracked Example Frames. The Yellow Circles are the BRIEF32 Results. The Green Circles are the BRIEF64 Results. The Blue Circles are the BRISK Results. The Pink Circles are the SBRISK Results. The White Circles are the Ground Truth Locations.
One of the videos for which the tracker does not perform well is a video of a basketball game in which a player is intended to be tracked. The video is named Basketball, and example frames can be found in the first row of Figure 28. A plot of the distance between the tracking results and the supplied ground truth can be found in Figure 29. This video is an example of two problems for the tracker. The first problem is when a region to be tracked is poorly defined (i.e., there is not much variance in the patch for the descriptor to leverage). This is the case for the first frame where there is not much detail in the region, as it is largely the same color and intensity values. Thus, the resulting descriptor is not very unique, and it is easy for the tracker to lose the player. Adding to the problem with the poor quality of the descriptor is that the object gets totally occluded by another player around frame 15. After this point, some of the trackers lock onto the new player and track him, while others get lost entirely. This problem could be mitigated by having the tracked object in higher resolution so that more details may provide a better description.
Another challenging video sequence was a video of a gymnast performing a floor routine, found in the second row of Figure 28. Its tracking error is plotted in Figure 30. The major reason this video was challenging was that the object to be tracked had a tall slender shape. The issue the tracker has with tracking objects that are not square is due to the descriptor design. The descriptors used regions that were all of a circular design, thus they work best when the region to be tracked is circular or square. In order to mitigate the problem, a different descriptor design which better accommodates to the object’s shape must be used. It is important to note that, unlike the other descriptors, the SBRISK descriptor tracks well until frame 385 at which point it gets lost as the gymnast performs an out of plane rotation.
Figure 30: Plot of the Euclidian Distance of the Trackers to the Ground Truth for the Gymnastics Video

The third row of frames in Figure 28 is from a video sequence that challenges the tracker through poor illumination and partial occlusion. The video is of a poorly illuminated guitarist playing at a concert. One of the issues with this sequence is the patch of the object that is to be tracked, the face, is so poorly illuminated that the descriptors generated from the patch have very little information encoded. The plot of the error of the trackers, found in Figure 31, shows that around frame 30 the tracker locks onto the guitar that partially occluded the face and tracks that for the rest of the video because it has more variation and was more unique.
A major problem for the tracker is when there is a large motion of the tracked object while it is occluded. Such a case is found in the fourth row of example frames in Figure 28. This video sequence is of a figure skating pair performance. A plot of the error from this video, found in Figure 32, shows that after the full occlusion of the skater, around frame 115, the tracker never recovers. This is because there is a large motion of the object while it is occluded, and, when it reappears, it is outside of the tracker’s search region. This problem could be mitigated by having a different search procedure that searches a larger area when objects are lost, thus increasing the chances of finding the object when it reappears.
4.4. Speed tests

4.4.1 Speed Test Configuration

It is important to compare the tracker’s speed to other trackers as well as more recent GPU based trackers. The speed of a tracker is often measured in Frames per Second (FPS), where 30 FPS is considered real-time performance. Most modern tracking applications require multiple objects to be tracked simultaneously in real time. Experiments were performed in order to determine the speed of the tracker relative to the number of objects being tracked, which ranged from 1 to 500. The descriptors tested for the timing experiments were BRIEF32, BRIEF64, SBRISK and the OpenCV implementation of BRISK on the CPU, and the GPU implementations of BRIEF32, BRIEF64, and SBRISK descriptors. All of the timing experiments were performed with a
consistent parameter set for all of the descriptors, so that the relative speeds can be fairly compared. The search radius was set to 20 because it showed consistent results for all of the descriptors, even though it was not the best radius for all descriptors. The experiments were performed on a Windows 7, 64-bit machine with an Intel Core i5-2400 3.10-GHz CPU, and 8 Gigabytes of RAM. The GPUs used for testing were a NVIDIA GTX 480 and a NVIDIA Tesla K20, both using the CUDA 5.0 driver. All of the timing results are averages that are strictly based on the execution time of the algorithm and data transfers on and off the GPU. The video decoding and display was determined to be an artificial bottleneck for timing results, and thus the timing results do not include time spent decoding or displaying the video, as they were outside of the scope of this thesis.

4.4.2 Single Object Tracking Speed Tests

The first experiment on the speed of the tracker was a comparison with other trackers while tracking a single object. Table 7 contains the timing results of the CPU implementations of two popular trackers, the MIL tracker and TLD tracker, based on the experiments in [52], as well as the CPU results from [10] and [11]. The timing results from the proposed tracker include the results from each of the descriptors implemented in the tracker.

<table>
<thead>
<tr>
<th>Tracker</th>
<th>Frames per second on the CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIL</td>
<td>38.1</td>
</tr>
<tr>
<td>TLD</td>
<td>28.1</td>
</tr>
<tr>
<td>Particle Filtering</td>
<td>4.1 to 60.5</td>
</tr>
<tr>
<td>Extended CAMSHIFT</td>
<td>58.8 to 68.7</td>
</tr>
<tr>
<td>Proposed BRIEF32</td>
<td>260.9</td>
</tr>
<tr>
<td>Proposed BRIEF64</td>
<td>128.1</td>
</tr>
<tr>
<td>Proposed BRISK</td>
<td>90.4</td>
</tr>
<tr>
<td>Proposed SBRISK</td>
<td>228.2</td>
</tr>
</tbody>
</table>
The comparison of the CPU and GPU implementations of the trackers that were implemented on the GPU can be found in Table 8. The timing results supplied for GPU implementations are presented as ranges of values because the tracking speed depends on the size and number of particles used to track the object. The results for the BRISK descriptor include only the OpenCV CPU implementation because the full BRISK descriptor was not implemented on the GPU. These results demonstrate the high performance of the tracker, and a significant speedup is shown for the GPU implementations even for a single object. These speedups increase as the number of objects increases until all of the GPU resources are saturated.

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>GPU</th>
<th>GPU acceleration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Particle Filtering</td>
<td>4.1 to 60.5</td>
<td>30.9 to 78.1</td>
<td>1.2 to 7.5</td>
</tr>
<tr>
<td>Extended CAMSHIFT</td>
<td>58.8 to 68.7</td>
<td>292.4 to 463</td>
<td>5 to 6.7</td>
</tr>
<tr>
<td>Proposed BRIEF32</td>
<td>260.9</td>
<td>2405.8</td>
<td>9.2</td>
</tr>
<tr>
<td>Proposed BRIEF64</td>
<td>128.1</td>
<td>1492.3</td>
<td>11.7</td>
</tr>
<tr>
<td>Proposed SBRISK</td>
<td>228.2</td>
<td>801</td>
<td>3.5</td>
</tr>
</tbody>
</table>

4.4.3 Multiple Object Tracking Speed Tests

For operation with a standard video camera, there is little benefit for a tracker to perform above real-time, because there are often other bottlenecks, such as video acquisition and decoding, which restrict operation at that speed. Most modern video cameras operate at 30 FPS, 33.33 ms per frame, and thus this is considered real-time operation in this work. Although the frame rate for real-time operation is assumed to be 30 FPS, Figure 33 confirms that the tracker can operate on video from cameras that can capture video at higher frame rates. The utility in the increased performance of a tracker
comes into effect when the additional capability for speed is leveraged to track multiple objects. The proposed tracker design was optimized for multiple object tracking. A comparison of the execution times of the CPU and GPU implementations of the tracker using the set of descriptors can be found in Table 9 and Figure 33. This graph plots the average time spent to run the tracker for all of the objects in each frame as the number of objects increases. The tracker was run on a 320×240 video with randomly selected points for the tracker to track, to minimize the potential advantage that could be gained from caching. The processing times of the CPU implementations are clearly separated from their GPU counterparts. This separation demonstrates how implementing the algorithm on the GPU provides a distinct advantage for the proposed algorithm.

Table 9: Execution times for each descriptor

<table>
<thead>
<tr>
<th>Descriptor</th>
<th>Number of Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>BRIEF32 CPU</td>
<td>3.83</td>
</tr>
<tr>
<td>BRIEF32 GPU</td>
<td>0.42</td>
</tr>
<tr>
<td>BRIEF64 CPU</td>
<td>7.81</td>
</tr>
<tr>
<td>BRIEF64 GPU</td>
<td>0.67</td>
</tr>
<tr>
<td>BRISK CPU</td>
<td>11.06</td>
</tr>
<tr>
<td>SBRISK CPU</td>
<td>4.38</td>
</tr>
<tr>
<td>SBRISK GPU</td>
<td>1.25</td>
</tr>
</tbody>
</table>
An important goal for this experiment was to determine how many objects the tracker could track while maintaining real-time performance. The gray line in Figure 33 represents the cutoff for real-time execution. The number of objects that can be tracked in real-time, 30 FPS, using each descriptor can be found in Table 10. This table makes the discrepancy between the CPU and GPU implementations very evident. The CPU implementations can track only 3 to 8 objects in real-time. The GPU implementations, in comparison, can track 70-140 objects in real time. Thus, the GPU implementation provides an order of magnitude increase in the number of objects that can be tracked in real time, which is a significant increase in performance.

Table 10: Maximum Number of Objects that can be tracked in Realtime

<table>
<thead>
<tr>
<th>DESCRIPTORS</th>
<th>REALTIME NUMBER ON CPU</th>
<th>REALTIME NUMBER ON GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRIEF32 CPU</td>
<td>8</td>
<td>140</td>
</tr>
<tr>
<td>BRIEF64 CPU</td>
<td>4</td>
<td>70</td>
</tr>
<tr>
<td>BRISK CPU</td>
<td>3</td>
<td>N.A.</td>
</tr>
<tr>
<td>SBRISK CPU</td>
<td>7</td>
<td>76</td>
</tr>
</tbody>
</table>
One important behavior of the proposed tracker is that the GPU implementation does not reach the maximum speedup until there is a significant number of objects to track. This happens because the GPU is not fully saturated when tracking a small number of objects, and thus there are unused resources available on the GPU. As the number of objects increases, so does the speedup until the GPU is fully saturated. This behavior is demonstrated in Figure 34 where the speedup of the tracker run on the K20 GPU versus the CPU implementation is graphed. The point at which the GPU begins to saturate is around 10 objects for the BRIEF descriptors and 15 objects for the SBRISK descriptor.

![GPU Speedup (K20)](image)

**Figure 34: GPU Speedup on the NVIDIA Tesla K20 GPU**

A similar plot to Figure 34 is found in Figure 35; however, this plot is for the speedup of the GTX 480 GPU. This plot shows how the less powerful card behaves differently from the higher performance K20. The first observation is that the saturation point occurs at a much lower number of objects and is much less pronounced than it was
on the K20. The saturation point for the BRIEF32 and BRISK64 descriptors occurred when tracking four objects, whereas the saturation point for the BRISK descriptor occurred when tracking seven objects. The lower saturation point and its lower impact is largely due to the fewer resources available; thus there were fewer wasted resources for the lower number of objects.

![GPU Speedup (480)](image)

**Figure 35: GPU Speedup on the NVIDIA GTX 480 GPU**

In order to fully understand the behavior of the GPU implementation and which areas of the algorithm benefit the most from GPU implementation, the execution time of each of the part of the algorithm must be investigated. The algorithm is broken into three major sections: image transfer time, time spent generating the descriptors for the candidates, and time spent scoring the candidates and selecting the winner. The plot of these three elements of the CPU implementation of the BRIEF32, BRIEF64, and SBRISK descriptors can be found in Figure 37 to 38. The CPU implementation does not
require any transfer of the image, and thus transfer time is always zero on the CPU. It is important to note that the description time and the scoring and selection time on the CPU show a direct correlation to the number of objects.

Figure 36: Execution Times of the Sections of Tracker using the BRIEF 32-byte descriptor on the CPU

Figure 37: Execution Times of the Sections of Tracker using the BRIEF 64-byte descriptor on the CPU
Figure 38: Execution Times of the Sections of Tracker using the SBRISK descriptor on the CPU

The plot of the three elements of the GPU implementation, run on the NVIDIA Tesla K20, can be found in Figures 39 to 41. It is clear that the behavior of the GPU implementation is much different than it is on the CPU. The first key difference between the two implementations is that the candidate description time is the fastest on the GPU and does not appear to be largely affected by the increase in the number of objects. This is in stark contrast to the CPU implementation in which the description was the longest part, and is greatly affected by the number of objects. Another key difference is that the impact of the number of objects on the execution time of the scoring and selection part of the algorithm is significantly less. These factors lead to the significant speedup of the GPU implementations.
Figure 39: Execution Times of the Sections of Tracker using the BRIEF32 descriptor on the K20 GPU

Figure 40: Execution Times of the Sections of Tracker using the BRIEF64 descriptor on the K20 GPU
There are two interesting artifacts that are present in the timing results for the three elements. The first is the correlation between the increase in number of objects tracked and the time spent transferring the image to the GPU. The image transfer time should remain constant no matter the number of objects because the size of the image remains the same, but transfer times vary with the number of objects. One potential explanation for this behavior is that the increased scheduling overhead happens to be manifesting itself in the image transfer times. This is a likely explanation because of the increased load on the scheduling hardware on the GPU as the number of objects increases. Another interesting behavior of the tracker is the large discrepancy between the scoring and selection time of the SBRISK descriptor compared to both the BRISK and BRIEF32 descriptors on the CPU. The scoring and selection time should be identical because the number of bytes is the same in all of the descriptors. However the step of scoring and selecting the winners for the BRISK and BRIEF64 descriptors was 2 to 3
times faster than the SBRISK descriptors. Interestingly, the scoring and selection of the SBRISK descriptor is slower than BRIEF64 for a smaller number of objects on the GPU, but for a larger number of objects it is faster. One potential explanation for such behavior is that caching is playing a role for both CPU and the smaller number of objects on the GPU.

It is important to examine the graphs in Figure 33, Figure 34, and Figure 35 with respect to the individual descriptors, in addition to the information they present on the relationship between the CPU and GPU implementations. The graph in Figure 33 provides information on which descriptor provides the lowest execution time for the tracker. This information is important when a descriptor is to be selected when speed is the primary concern. It is clear from the graph that BRIEF32 has the lowest execution time, which is to be expected because of the fewer number of computations that are needed on the smaller descriptor. The BRIEF64 and the SBRISK descriptors demonstrate an interesting relationship when comparing their relative speed on both the CPU and GPU. On the CPU the SBRISK descriptor is only slightly slower than the BRIEF32 descriptor, and much faster than the BRIEF64 descriptor in all cases. On the GPU, the BRIEF64 descriptor is faster than the SBRISK descriptor for small numbers of objects, but SBRISK is faster for larger numbers of objects. This behavior is likely due to the fact that generating the integral image plays a much smaller role in the total computation time on the CPU, because the generation of each of the descriptors is done in series. However, on the GPU, when all of the descriptors are generated in parallel, the integral image generation plays a much larger role. Thus, a large number of objects is required to hide the overhead that is caused by the integral image generation for the SBRISK descriptor.
The data in Figure 34 and Figure 35 provide information on the relative speedups of each of the descriptors as well as information on which descriptors are most optimized for the GPU. These figures show that the BRIEF32 descriptor clearly benefits the most from GPU implementation, with a speedup of 9 to 16 compared to the CPU implementation. However the SBRISK descriptor demonstrates the lowest speedup of only 3 to 10 in relation to the CPU. These two results are likely due to the simplicity of the BRIEF32 generation process, whereas the SBRISK descriptor generation is much more complex and has an additional step of calculating the integral image.

One of the pivotal justifications for developing the SBRISK descriptor was that the descriptor did not need to be as robust to variations in rotation and size; therefore, the descriptor generation process could be accelerated through removing some of the superfluous computations in the BRISK descriptor. The acceleration is clearly evident in Figure 33 and Table 7, where SBRISK is 2.25 to 2.5 times faster than the original BRISK implementation provided by OpenCV. This significant speedup provides justification for a simplified version of the BRISK descriptor in order to increase the speed of the tracker and decrease the complexity of the descriptor generation on the GPU.

4.4.4 Mobile Tracking Speed Tests

The benchmarking of the mobile implementation focused on the speed of the tracker because the mobile implementation of the tracker was identical to the CPU implementation and the accuracy results were identical. There was, however, one alteration that was made to the search pattern of the tracker for the mobile implantation benchmarks. Due to the processing and memory constraints of mobile devices, the tests
done in this section have reduced the search radius to 10 and increased the spacing to twice that of the CPU and GPU tests. This reduced the number of candidates in the search region from 1681 to 441 without altering the size of the area searched. Tests were performed with the CPU implementation using the altered search pattern on the test video sequences. The tests confirmed that this change did not significantly impact the accuracy of the tracker. Because of the performance constraints of the mobile device, the implementation supports only a single object, yet due to the flexibility of the tracker this constraint could easily be removed in the future when mobile devices become more powerful. The performance of the three, fully implemented, descriptors in this thesis was tested with the same configuration on the CPU and GPU as well as on two iOS devices. The iOS devices used in this benchmark were an iPhone 5 with 16 GB of storage, an Apple A6 Processor, and 1 GB of RAM, and an iPad Air with 16 GB of storage, an Apple A7 Processor, and 1 GB of RAM. The results of the benchmark can be found in Table 11. This table shows the time spent for description and scoring sections of the algorithm as well as the total execution time, in milliseconds, and Frames per Second of each implementation. The aim of the mobile implementation was to achieve realtime performance, 30 FPS.
Table 11: Comparison of Mobile Execution Time in ms

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>GPU</th>
<th>iPhone 5</th>
<th>iPad Air</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BRIEF32</td>
<td>BRIEF64</td>
<td>BRIEF32</td>
<td>BRIEF64</td>
</tr>
<tr>
<td>Description Time</td>
<td>0.84</td>
<td>1.56</td>
<td>0.70</td>
<td>0.01</td>
</tr>
<tr>
<td>Scoring Time</td>
<td>0.24</td>
<td>0.42</td>
<td>0.55</td>
<td>0.18</td>
</tr>
<tr>
<td>Total Time</td>
<td>1.08</td>
<td>1.97</td>
<td>1.25</td>
<td>0.21</td>
</tr>
<tr>
<td>FPS</td>
<td>927.15</td>
<td>506.70</td>
<td>798.54</td>
<td>4727.81</td>
</tr>
</tbody>
</table>

Both the iPhone and iPad reach realtime performance for the BRIEF32 and SBRISK descriptors. It is important to note the impact of the random memory accesses on the BRIEF descriptor’s generation time. The BRIEF descriptors take 2 to 4 times as long to generate as the SBRISK descriptor. This should not be the case because the descriptors are much simpler than the SBRISK descriptor. One likely cause of the increased execution time is the smaller cache available on mobile devices, and thus the random accesses result in a larger number of cache misses. Of most interest in these results is the impact of one generation change in the processors of the devices. Both devices have the same amount of ram and storage, and the only difference between them is their processor, yet the iPad tracks objects at more than twice the speed of the iPhone. This demonstrates promising results for the future of Object Tracking on mobile devices.

4.5. Parts Tracking Tests

The parts tracking method was tested against the individual object tracking method for several videos. Unfortunately, there were no publicly available ground truth files for any datasets, and thus the observations for these experiments were primarily qualitative. Since the main motivation for tracking of parts is for facial feature tracking, the parts tracking algorithm was compared using several recordings of faces. These
recordings are from the Stanford database [54]. The videos tested the tracker on many
difficult motions including scale changes and out of plane rotation. The results from two
example video sequences are shown in Figure 42. The two videos demonstrate one of the
hardest motions for the parts tracker to mitigate, out of plane rotation. Rows 1 and 3 show
how the individually tracked objects get lost as the eyes disappear. Rows 2 and 4,
however, demonstrate how the tracking the eyes in coordination with the face maintains
the tracking of the eyes, even when the eyes are occluded and their matching falls below
the tracking threshold. These results demonstrate the potential for a parts tracking model
that leverages the information from the whole object’s motion.

Figure 42: Parts Tracking Demonstration Frames. Rows 1 and 3 are the Results from Individually Tracking the Eyes. Row 2 and 4 are the Results from Parts Tracking. Red Circles Represent Updates Where the Parts are Not Found because they Fail to Meet the Matching Threshold.
Chapter 5  Conclusions and Future Work

The aim of this thesis was to develop an object tracker that leverages local binary descriptors to track multiple objects efficiently and parts of objects with high accuracy. The following contributions to the field of object tracking were presented in this work:

- Accurate multiple object tracking algorithm that is computationally efficient
- Accelerated tracking algorithm using the CUDA architecture for the GPU
- Simplified BRISK Descriptor
- Parts constrained motion model
- Mobile implementation on iOS devices

The proposed tracker has demonstrated that it outperforms state of the art trackers in terms of both speed and accuracy. The tracker has demonstrated that optimal results are achieved when the parameters are tuned for a specific object type and motion, but high quality results can still be achieved with a standardized parameter set. A GPU implementation of the tracker demonstrates the potential of the tracker for use in environments where a large number of objects must be tracked. This work has also demonstrated the benefits of the proposed Simplified BRISK descriptor in terms of both uniqueness and simplicity. Additionally, this work has shown that an object-informed motion model of parts can increase the accuracy for tracking parts of objects which tend to be more difficult to track. Lastly, the mobile implementation shows that the proposed tracker has sufficient computational efficiency that it can perform at real-time speeds on common mobile devices. There are some areas that show potential to extract even better
performance from the tracker, including changing the descriptors used, handling non-
square regions, and performing an improved search if the object is lost.

One potential alteration to the tracker that could improve its performance is to use
a more advanced descriptor instead of the relatively simple BRIEF and SBRISK
descriptors. The benefit of the tracker design is that the tracker operation is independent
of the descriptor used, and thus changing the descriptor would be a matter of a different
function call, provided an implementation of the descriptor is available. An example of a
descriptor that is more advanced than the BRIEF and BRISK descriptors is the FREAK
descriptor. The FREAK descriptor has shown better recall rates under certain conditions
[45]. FREAK was not implemented in this work due to its higher complexity for a GPU
implementation; however the OpenCV implementation of the FREAK descriptor on the
CPU has been verified to work with the tracker. Given that the FREAK descriptor is
relatively new, there is opportunity to refine and further improve binary descriptors that
would benefit this tracker.

A problem that plagues the proposed tracker, even with an advanced descriptor, is
the fact that the descriptors are all based on a circular or square design. This factor leads
to complications when the object to be tracked is not square. If the descriptor includes too
much of the background, it is easy for it to get stuck at a background location. Conversely, if the descriptor is too small, the image patch to be described may not
contain enough variation to provide for a reliable descriptor. These two factors
necessitate the shape of descriptors to change, which has the potential to improve the
performance of the tracker for non-square objects. The potential solution is to develop a
new descriptor that conforms to the shape of the object to be tracked. Such a descriptor
could be as simple as combining multiple preexisting descriptor regions, arranged such that the entire object is described. A more advanced solution would be to alter the arrangements of the comparisons made in the descriptors themselves, such that they themselves can conform to the object’s shape.

The tracker also experienced difficulty redetecting objects which undergo substantial movement while being occluded. The tracker performed well if objects remained stationary or moved slowly while another object occluded them. However, due to the local search pattern of the tracker, the tracker would get lost when the tracked object moved significantly while it was occluded. One potential method for mitigating such a problem is to introduce a separate search procedure for lost objects. In such a solution a search for the object could be conducted over the whole frame, if an object is determined to be occluded. This allows for the tracker to recover from situations where the object is lost and the tracker fails.

In conclusion, this work has demonstrated that a high performance object tracker which uses binary descriptors and locality constraints can be both highly accurate and computationally efficient. The experimental results in this work have shown that the proposed tracker has potential to positively impact a wide range of applications.
Bibliography


