The Pennsylvania State University

The Graduate School

College of Engineering

OPTIMIZATION AND HARDWARE ACCELERATION OF CONSENSUS-BASED MATCHING AND TRACKING

A Thesis in

Computer Science and Engineering

by

Joshua S. Snyder

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

May 2015

The thesis of Joshua S. Snyder was reviewed and approved* by the following:

Vijaykrishnan Narayanan
Distinguished Professor of Computer Science and Engineering
Thesis Adviser

John Sampson
Professor of Computer Science and Engineering

Lee Coraor
Associate Professor of Computer Science and Engineering
Graduate Program Head for Computer Science and Engineering

Kevin Irick
External Research Adviser

*Signatures are on file in the Graduate School

ABSTRACT

Image and video understanding has become an increasingly valuable capability for many emerging applications such as smart retail, intelligent surveillance, and autonomous robotic systems. The critical barrier to enabling these applications is the high execution latencies of complex vision tasks that make real-time system constraints difficult, or impossible, to achieve. One specific instance of a complex vision task is object tracking, which is the focus of this thesis. Object tracking is a necessary component of grocery shopping assistance applications that track a grocery item and a person's hand and guides the hand to the item to pick it up. Although there are many object tracking algorithms to choose from, this work investigates the performance bottlenecks and optimizations of the Consensus-based Matching and Tracking, CMT, algorithm. To circumvent the limitations of standard optical-flow based trackers, CMT uses a descriptor matching step to redetect an object's key features that would be permanently lost in the standard approach. This allows for an object to be hidden or occluded from view and redetected once it reappears in the view of the camera.

For fully autonomous systems, in which re-initialization of a failed object track may not be possible or prohibitively costly, robustness of the tracker is of critical importance. As such, this work introduces, an enhanced version of the CMT algorithm that exhibits improvements in accuracy and robustness as evaluated against a standardized benchmark. The improvement in accuracy and robustness of the enhanced CMT comes at the cost of a significant increase in computational latency. Accordingly, this work also proposes a hybrid system that integrates high-performance custom hardware accelerators with a traditional processor to alleviate these new performance bottlenecks and to support real-time throughput.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I am very grateful to my thesis advisor, Dr. Vijaykrishnan Narayanan, for allowing me to work in the Microsystems Design Laboratory (MDL) and providing guidance throughout my research and thesis work. I am also greatly indebted to my honors academic advisor, Dr. Lee Coraor, who helped me apply to the Integrated Undergraduate/Graduate program through the Schreyer Honors College and helped successfully guide me through each semester of my five years at Penn State.

I would like to give a special thanks to all of my lab mates who were crucial to my success during my time in the MDL, with special recognition to Dr. Kevin Irick and Dr. Matthew Cotter. I have gained invaluable experience working with these two individuals, both in hardware and in software, and I cannot thank them enough.

Lastly, I would like to thank my family and friends for all of their support and encouragement throughout my time at Penn State and for pushing me to be the best person I can be, whether in the classroom, lab, or in life. Without them, I would not be where I am today.

CHAPTER 1

INTRODUCTION

Processors have become increasingly powerful since their inception. The first microprocessor, invented by Intel in 1971, featured a single in-order pipeline and operated at a maximum clock frequency of 740 kilohertz. In comparison, modern processors feature eight or more cores with each operating in the three to four gigahertz range. Even with significant improvements in processor capabilities and performance, there are many real-time constrained applications whose computational requirements far exceed the capacity of modern processors. In the context of image and video applications, real-time is generally defined as a system that maintains a minimum throughput of 30 frames/second or a maximum processing latency of 33ms per input. Object Tracking, the topic of this thesis, is one such vision task that requires real-time operation and can exhibit higher accuracy if image frames are provided at higher than real-time rates.

1.1 Image Processing and Object Tracking

Most complex image tasks are very computationally expensive. Working with pixels, especially in large resolution images, requires a large amount of data to be processed. A high-definition color image with a size of 1920x1080 contains about 50 million bits, where each pixel is 24 bits. Also, with most cameras recording at 30 frames per second, the image task must process about 1.5 billion bits per second. The amount of data needing to be processed per second is simply too large for most complex image tasks when running on a standard processor.

The algorithm that this thesis focuses on is Consensus-based Matching and Tracking, or CMT [1]. This algorithm is taken to be one of the leading object trackers that is available to the public, but is optimized for processing speed and not for extreme accuracy. This thesis will explore more accurate and robust versions of this algorithm and compare the different versions in order to decide which version is best.

1.2 FPGA Acceleration

FPGAs, or Field Programmable Gate Arrays, allow a programmer to completely customize a datapath within the hardware to perform specific jobs; they simulate a processor that is built to do one job. FPGAs are being used more frequently because of their customizability, high throughput of massively parallel processes, and the ability to simulate a physical chip without having to print one. For the purpose of object tracking, any part of the algorithm that can be processed in parallel can be mapped to the FPGA in order to accelerate the process. All other parts will stay on the host processor, which will communicate with the FPGA. This thesis will focus on several FPGA accelerators and their architectures for specific parts of the object tracking algorithm.

1.3 Organization of Thesis

The remaining chapters of this thesis will be organized in the following fashion: Chapter 2 will discuss the object tracking algorithm CMT, as well as modifications made to CMT to increase accuracy and robustness. Chapter 3 discusses the architecture of several hardware accelerators that will be used in this system. Chapter 4 will discuss expected results of the accelerators in several different configurations of the system. Finally, Chapter 5 will draw conclusions from the rest of the thesis.

CHAPTER 2

OBJECT TRACKING

Previous work has tried to significantly improve object tracking, specifically for model-free versions. Algorithms that are “model-free” are given an initial region of interest (ROI) in the first frame of the video or camera stream and it is the job of the tracking algorithm to determine where the object is located in subsequent frames. The biggest advantage of model-free tracking algorithms is the ability to be used in virtually any situation. There is no need to train the algorithm on different object models for the specific applications before tracking when using a model-free tracker.

Another important property of a tracking algorithm is the ability to redetect a tracked object after it has stopped tracking it for any reason, whether leaving the field of view or an error by the algorithm. If an object is lost while tracking, the tracker must be able to determine where the object returns to view. Tracking algorithms that have this property are considered long-term object trackers. Long-term object trackers are crucial to systems that have little or no user input after initialization, such as an assistance application for visually-impaired individuals.

Although there have been advancements in the model-free object tracking domain, there is still a lack of methods to deal with partial and full occlusions, noise, and appearance changes. Some tracking algorithms use online learning methods to deal with these problems, especially object appearance changes. A few of these methods are discussed in more detail [2] [3] [4], but one tracker in particular that performs online tracking is OpenTLD [5], which is an open-source, fast version of the original TLD algorithm [6]. OpenTLD, where TLD stands for Tracking Learning and Detection, employs a learning method that stores positive and negative patches of an object so that it can more accurately track changes in its appearance. The largest drawback to online learning is the error that it introduces throughout the video sequence. As the ROI, produced by the algorithm, starts to slowly drift away from the object, patches will be classified as being positive templates when they are actually negative, and the algorithm will start learning to track an incorrect region in the frames.

One algorithm that was developed to address these issues was CMT, or Consensus-based Matching and Tracking [1].

2.1 Overview of CMT

There is strong evidence that shows a combination of static and adaptive elements will improve the robustness of a tracking algorithm [6] [7]. CMT employs a solution to decouple the static and adaptive model elements. This algorithm decouples the elements by modeling the appearance of the object and background on only the initial frame and processes appearance changes by using an adaptive tracking method, which tracks BRISK keypoints.

2.1.1 BRISK Keypoints and Descriptors

A keypoint in an image is a location where the region around it is salient, or interesting. In other words, this region stands out from the others around it. Each keypoint contains a set of values that describe the keypoint (and the region around it), called descriptors. There are many different types of keypoint detection and descriptor extraction algorithms, but CMT uses BRISK [8].

BRISK stands for Binary Robust Invariant Scalable Keypoints. This algorithm is based upon the use of scale-space keypoint detection [8]. This method estimates the true scale of each keypoint by using different layers of scales, going from fine to coarse, and finds the maximum FAST score [9] from all of the layers [8]. The keypoint descriptor is then built as a 512-bit binary string (64 descriptors, which are 8 bits each) by concatenating the results from simple pixel brightness tests and using the orientation of each keypoint to achieve rotation invariance [8].

2.1.2 Initialization of CMT

Beginning with the first frame of a video sequence and an initial ROI, CMT initializes the algorithm by first detecting all keypoints and descriptors and categorizing them as either belonging to the object (inside the ROI) or global (all keypoints in the image) models. Each object keypoint is given a unique class identifier, starting with one, while all background keypoints are given a class identifier of zero, which will be used later in keypoint matching.

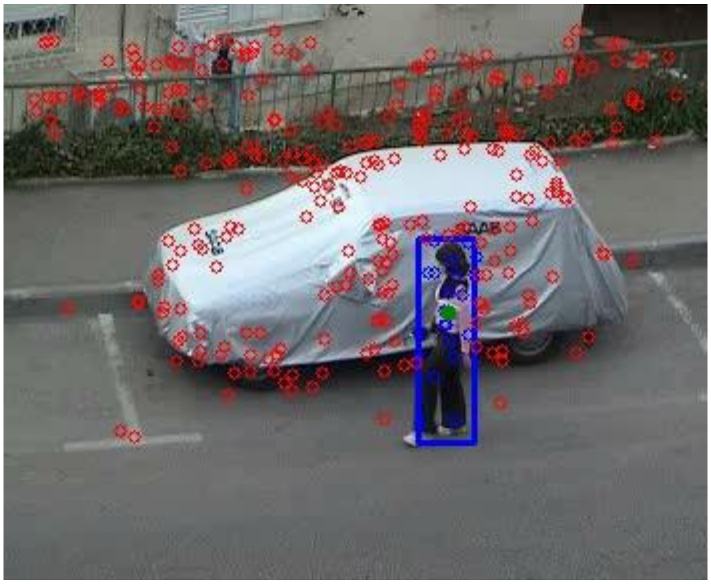


Figure 1. Initialization of CMT using the first frame of the sequence. Blue keypoints (inside the ROI) are object keypoints while red keypoints (outside the ROI) are background keypoints.

The keypoints in the object model will be used to calculate different values. Each keypoint will have its distance to the center of the ROI calculated, which are called springs. The distance from each corner of the ROI to the center is also calculated (for simplification, all initial ROIs are axis aligned rectangles). These distances are called anchors. Finally, the distances between keypoints and the angles between the keypoint descriptors are calculated. The springs, anchors, inter-keypoint distances, and keypoint descriptor angles are used to determine scale and rotation changes.

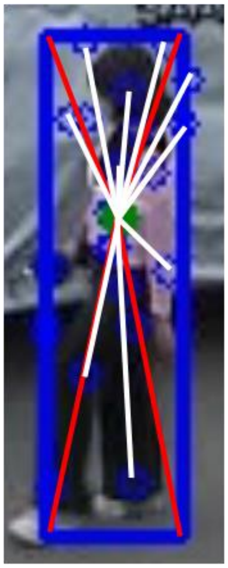


Figure 2. Calculation of anchors and springs. The anchors are red and the springs are white.

2.1.3 Keypoint Tracking

All keypoints in the initial ROI will be classified as active keypoints and will try to be tracked into the next frame. CMT uses a combination Lucas-Kanade (LK) optical-flow [10] and error thresholding to perform keypoint tracking. The LK optical-flow algorithm searches for a location in the current frame where the keypoint from the previous frame is found. CMT then uses the LK optical-flow in a reverse fashion and a forward-backward error method to determine the error of the tracked keypoint [11]. If the backward optical-flow step produces a keypoint that is above a certain distance (error measure) from the original keypoint, then that keypoint is not tracked and is removed from consideration. The following image shows an example of the forward-backward keypoint tracking and error measure.

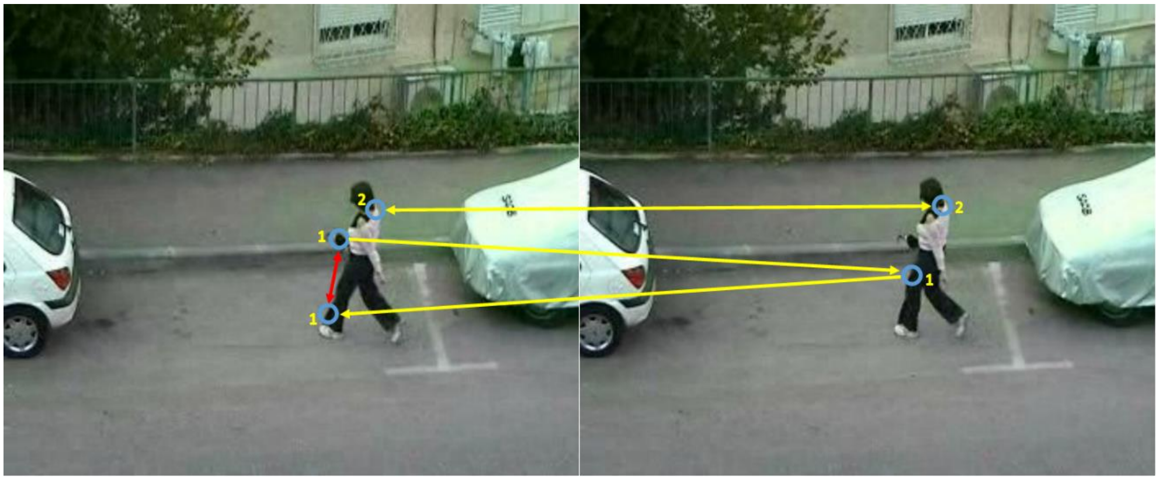


Figure 3. Forward backward tracking example where the yellow lines are the keypoint tracking and the red line is the error distance.

The above example shows that for keypoint 1, the forward-backward tracking produces an error distance between the predicted location and the original location of the keypoint. However, keypoint 2 is tracked back to the exact location and produces no error.

All keypoints that are successfully tracked are set as active keypoints and will be tracked to the next frame (if possible). The tracking step of CMT is the adaptive property of the algorithm because it has the ability to track any keypoint, even if it is not in the object model. Therefore, changes in the object's appearance can be processed by the tracker.

2.1.4 Scale, Rotation, and Center Voting

Following the tracking step, CMT estimates the new center, scale, and rotation of the ROI. The center of the new ROI is estimated by using the spring distances and a voting method, where each tracked keypoint votes for the center of the ROI. The votes are then clustered to form the center point [1]. The algorithm also estimates the scale and rotation of the ROI by using the inter-keypoint distances and descriptor angles [1].

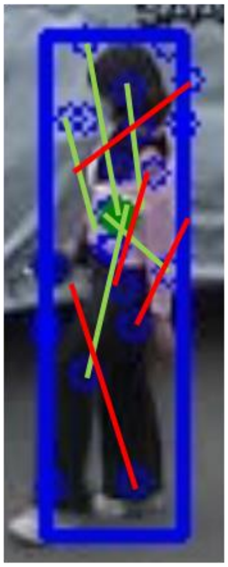


Figure 4. Example of center voting. Green springs are accurate votes for the center while the red springs are votes considered as outliers.

2.1.5 Keypoint Matching

The final step of CMT is the matching of keypoints. This step allows for lost keypoints in the tracking step to be redetected and continue to be tracked. When the object is lost all together, it can also be redetected during this step. CMT first detects all BRISK keypoints in the current frame. Then, the keypoints are used to extract the descriptors. These descriptors are first matched to the global model (all keypoints from the initial frame), and, if there is a valid center estimate, all keypoints are then matched to the object model. All matches that have been matched to a background keypoint in the global model are not considered further. Matches to object keypoints in the model are evaluated for their confidences and only kept if the value is over a certain confidence threshold [1]. The keypoint distances are also calculated and used to check the geometric location of the keypoints in respect to the new center [1]. All unmatched tracked keypoints are also added to the new active keypoint list. As long as there are enough keypoints retained from the tracking and matching steps, a new ROI is formed and the cycle repeats with the next frame.

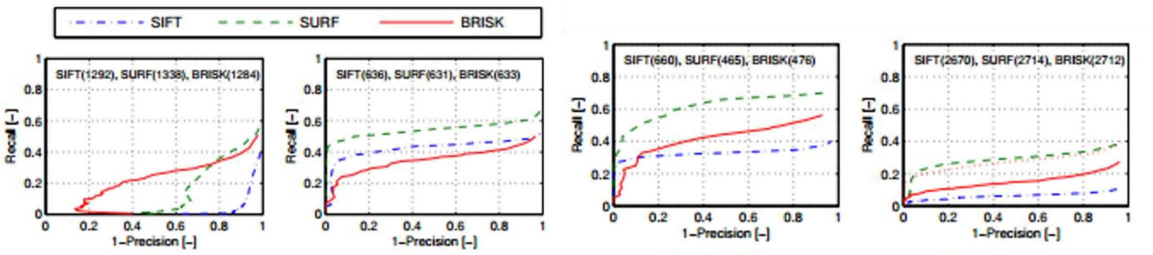
2.2 Improvements to CMT

Although CMT gave good visual results when running it with different video sequences, there were still improvements that could be made to increase the accuracy and robustness of the algorithm. The three major improvements that were made in this work were using SURF keypoints and descriptors instead of BRISK, adaptive background subtraction, and weighted distance confidences from the center of the ROI. These improvements are discussed in the following subsections.

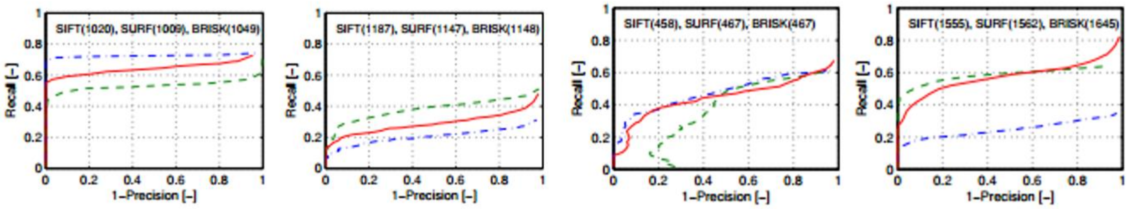
2.2.1 SURF Keypoints and Descriptors

One keypoint detection algorithm that can replace BRISK is SURF, or Speeded Up Robust Features [12]. It was created to be a faster, yet comparable, algorithm than SIFT [13], which is thought of as the most appealing algorithm in terms of accuracy and robustness. SURF is based upon the use of a Hessian matrix and uses the determinant of this matrix to detect and determine the location and scale of keypoints in an image [12]. Although SURF does not find the exact Hessian matrix, it uses integral images and box filters to approximate second-order Gaussian derivatives, which are used to create the matrix [12]. The descriptors for SURF are based upon SIFT, but with a lower complexity. First, SURF fixes a circular, reproducible fixed region around a keypoint [12]. Then a square region is constructed around the created circular region, aligned with the selected orientation, and the descriptor is extracted from different computations inside this region [12].

SURF, although slower computationally in software, tends to produce more robust keypoint and descriptors than BRISK, which is used in the original version of CMT. According to [8], SURF does not outperform BRISK in all situations, but on average its performance is higher than BRISK. The following graphs of precision vs. recall, taken from [8], shows that on average, SURF outperforms BRISK.



(a) Graffiti 1-3 (b) Wall 1-4 (e) Bikes 1-4 (f) Trees 1-4



(c) Image Rotation of 60° on Wall 1. (d) Boat 1-4 (g) Leuven 1-4 (h) Ubc 1-4

Figure 5. Precision-recall graphs from eight different examples [8].

Because part of this work is to improve the performance of CMT, SURF can be used instead of BRISK to make CMT a more robust and accurate object tracking algorithm.

2.2.2 Adaptive Background Subtraction

One of the largest pitfalls of CMT is the adaptability of the models. Because the algorithm uses static global and object models that are initialized in the first frame of the sequence, it becomes increasingly difficult to match these models as the number of frames increase and the scene changes. Although keypoint tracking handles some of the adaptability issues (such as a change in appearance of the object), it does not help when trying to match keypoints. When using descriptors to match keypoints, whether it is BRISK or SURF, the more keypoints that are trying to be matched to a model increases the number of keypoints that are incorrectly matched due to noise. Therefore, the objective of background subtraction is to reduce the number of keypoints being matched to the object model by not considering keypoints that are in the background.

Currently, CMT does two separate matching steps: it matches all of the keypoints in the current frame to the global model and it matches all current keypoints to the object model. The condition on the first match is if a keypoint is matched to a background keypoint in the global model (has a class of zero), then it should not be considered further. However, the second matching step tries to match all of the same keypoints being matched in the first step, which contains all of the noisy background keypoints, to only the object model.

The problem with normal background subtraction is that it is not easy to perform when the background is changing. If the background is guaranteed to be stationary, then a straightforward foreground detection method can be applied. But for the intended application of this work, the camera is not assumed to be in a fixed location. Therefore, a new adaptive background model must be created to try to match and subtract background keypoints from consideration. In this new method, a new adaptive background model is created in each frame when the object is tracked and an ROI is produced.

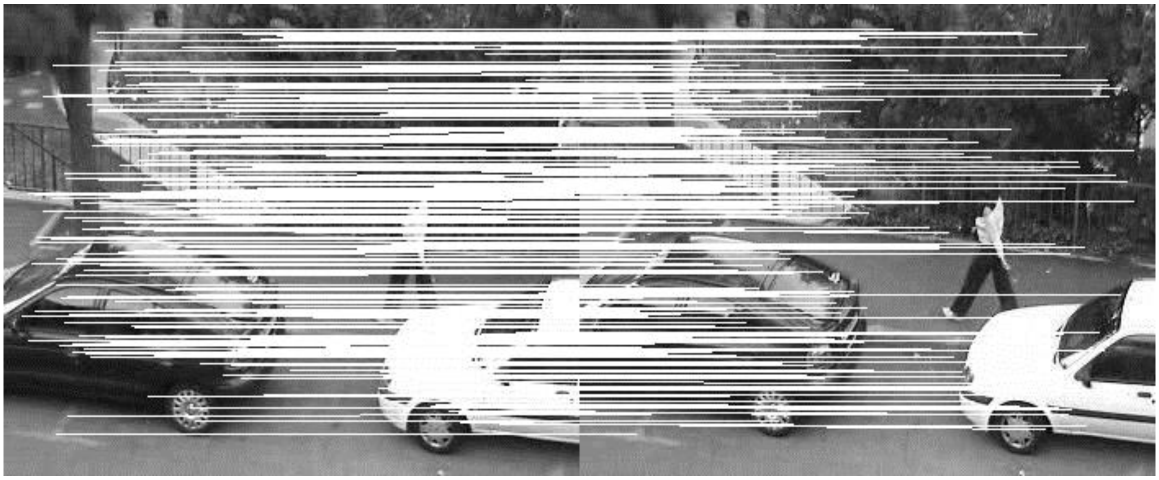


Figure 6. Example of Background Subtraction. The white lines are matches from the previous background (left) to the current background (right). All matched keypoints are removed from consideration for object matching.

An adaptive background model is formed by all keypoints, with descriptors, that are outside of the current ROI, plus a small percentage buffer. The keypoint descriptors are then used as the model for matching all observed keypoints in the next frame. A straight-forward matching step is applied to all observed keypoints with the adaptive background model. If a keypoint matches a model background keypoint with a high confidence value, it is kept as a background keypoint for further processing. Otherwise, the keypoint is kept as a possible object keypoint to later be matched to the object model. All keypoints that are matched to the background are then run through a homography algorithm to get rid of any outliers. The keypoint that are considered as outliers are added to the list of possible object keypoints. This process will reduce the number of keypoints trying to be matched to the object model, which in turn will increase the accuracy of the matches. When an ROI is not produced, a new background model will not be created and the observed keypoints will be matched to the most recent model when the object was tracked.

2.2.3 Weighted Distance Confidences

The last major improvement made to CMT is the way that match confidences are calculated. Along with the normal confidence calculations, there needed to be a way to restrain keypoints from being falsely matched to object keypoints that were a large distance away from the ROI. One problem with CMT is that it considers all keypoints equally when matching them against the object model. This results in many keypoints being falsely matched. In this distance weighted confidence solution, keypoints that are being matched to the object model are each given a weight that represents how far the point is from the center of the ROI. This weight is calculated using a Gaussian distribution, setting the mean value to zero (the center of the ROI) and a variance equal to the largest distance from center to a corner of the ROI. The confidences of the matches are then multiplied by the calculated weight. The following equation shows this step:

$$Weighted\ Confidence = (Normal\ Confidence) * e^{-\frac{x^2}{2\sigma^2}}$$

In this equation, x is the Euclidean distance from the center of the ROI to the point being considered and σ^2 is the variance as defined earlier.

If the weighted confidence is above a certain confidence threshold value (which is set lower than the normal confidence value), then the keypoint is kept as a valid match. This method reduces a large amount of error and only allows the object to move a small distance or change scales slowly in each frame.

2.3 Comparison of Tracking Algorithms

The goal of the enhanced CMT algorithm is to improve both the accuracy and robustness of the object tracking algorithm. With this in mind, there needs to be a way to evaluate the different algorithms in a uniform fashion.

2.3.1 Evaluation Toolkit

For this task, an evaluation toolkit used for the Visual Object Tracking (VOT) 2014 challenge [14] was used to compare the different tracking algorithms. The challenge provides a data set of 25 different video sequences and a MATLAB toolkit in order to measure accuracy and robustness, along with some other values, such as camera motion, of an object tracker. Accuracy is defined as the amount of overlap between the ground truth ROI and the ROI produced by the tracking algorithm [15]. The more that the tracking ROI overlaps the ground truth ROI, the higher the accuracy will be. The robustness measure is simply the failure rate, or the amount of times there is no overlap between the ground truth ROI and the tracker produced ROI [15].

An important note to make on the measuring of the accuracy and robustness is that the toolkit will re-initialize the object ROI after each failure within a sequence. Therefore, a higher number of failures will usually result in a higher accuracy. Re-initialization gives the algorithm a chance to collect new information about the object as the object's appearance is changing. Therefore, a decrease in the number of failures should ultimately result in a higher overall performance of a tracker, even if it does not show in the accuracy results.

The toolkit runs two different tests, a baseline test and a region noise test. The baseline test initializes the tracker with a tight ROI around the object in the first frame, while the region noise test initializes the tracker with an imperfect ROI, which introduces noise into the initial models. Both tests run through all of the video sequences and reinitialize the tracker after every failure [14]. Each sequence is run several times in order to get good results. The toolkit then outputs the different values for each sequence and an overall average of each measured value over all of the sequences.

2.3.2 Results of Evaluation Toolkit

In order to decide which tracking algorithm is better, both the original CMT and the enhanced CMT algorithms were run through the toolkit. Also, OpenTLD [5] was run through the toolkit as a reference to a different type of object tracking algorithm. The following figures and tables are the results of the different tests and show the average values for accuracy and robustness (number of failures).

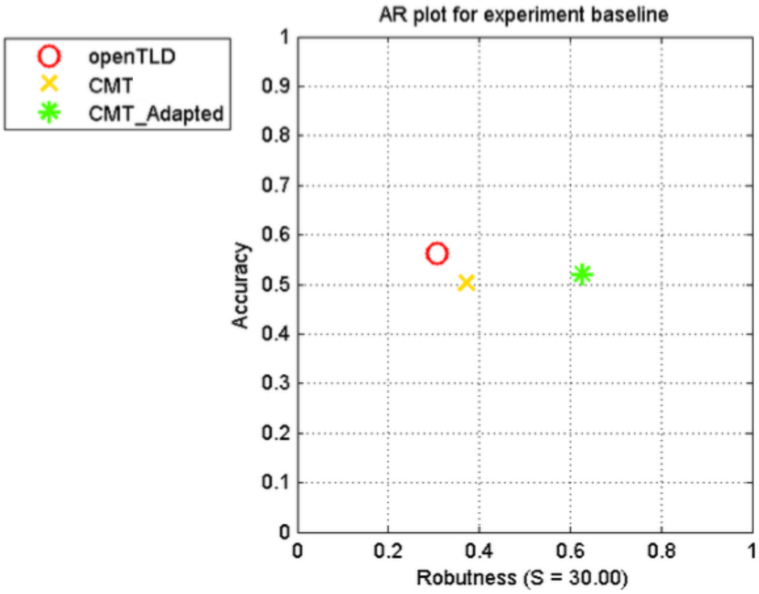


Figure 7. VOT toolkit analysis of OpenTLD, CMT (Original Algorithm), and CMT_Adapted (Enhanced Algorithm) on baseline test.

Algorithm	**Average Accuracy**	**Average No. of Failures**
OpenTLD	0.529	8.96
CMT	0.496	7.32
CMT_Adapted	0.513	3.36

Table 1. Average accuracy and number of failures for the three algorithms on the baseline test.

In Figure 7, the accuracy and robustness is plotted on a graph for the baseline test, with accuracy ranging from 0 to 1 on the y-axis and robustness ranging from 0 to 1 on the x-axis.

Table **1** is a numeric representation of the baseline test results. These values are the averages from all 25 video sequences, where robustness is a function of the number of failures and then normalized between 0 and 1. By visually inspecting the graph, OpenTLD achieves the best accuracy out of the three algorithms, but has the worst robustness. The enhanced CMT algorithm obtains the highest robustness values (by a significant amount) and has the second highest accuracy of the three algorithms. More importantly, the enhanced CMT algorithm is better in both accuracy and robustness than the original CMT algorithm.

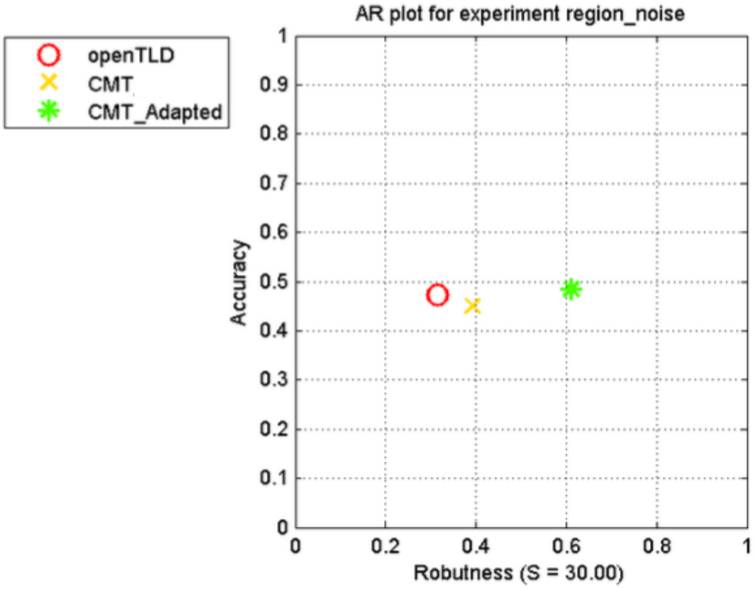


Figure 8. VOT toolkit analysis output of OpenTLD, CMT (Original Algorithm), and CMT_Adapted (Enhanced Algorithm) on region noise test.

Algorithm	**Average Accuracy**	**Average No. of Failures**
OpenTLD	0.467	8.757
CMT	0.462	7.051
CMT_Adapted	0.482	3.539

Table 2. Average accuracy and number of failures for the three algorithms on the region noise test.

Figure 8 and Table 2 show the results from the region noise test performed by the VOT toolkit. As shown, all of the accuracies of the algorithms decrease from the baseline test, but the enhanced CMT algorithm achieves better accuracy and robustness values that the other two algorithms. This improvement over OpenTLD is the effect of using SURF keypoints instead of pixels in the random fern classifier [5]. Keypoints are more robust and less prone to noise than a simple random pixel intensity calculation, as done in OpenTLD. The enhanced CMT algorithm achieves better results that the original CMT algorithm because of the three improvements outlined in Section 2.2.

2.4 Reasons for Choosing Enhanced CMT

There are two main reasons for choosing the enhanced CMT algorithm to continue working on. The first reason is the improvement in accuracy over the original CMT algorithm and the large decrease in the number of failures over both CMT and OpenTLD. As the results of the two tests shown in the previous section, the accuracy for the enhanced CMT increased slightly over the original algorithm, although it is still slightly below the accuracy of OpenTLD. However, the largest improvement of enhanced CMT is the robustness. The average number of failures were cut in over half compared to the original CMT algorithm and OpenTLD. This is important to note because for most applications of object tracking, there will be no user intervention to re-initialize the tracker on the object every time it fails.

The second reason to use the enhanced CMT tracker is because it uses SURF keypoint and descriptors, which are also used in many other vision algorithms, such as image stabilization, object recognition, stereo correspondence, and similar object detection. When using object tracking in a vision pipeline, it is more efficient to use keypoints and descriptors that are already being used, rather than computing completely different ones. For example, when trying to find and pick up an item on a grocery shelf, the first step is to detect the item using an object recognition algorithm. Once the object is found, the ROI produced can initialize the tracker and guide a person’s hand to the item in order to pick it up. Because object recognition already uses SURF keypoints and descriptors, there is no reason to use a different keypoint and waste resources.

2.5 Motivation to Accelerate Enhanced CMT

One goal of CMT is to do accurate object tracking as quickly as possible. At smaller frame sizes, such as 640x480 (the default frame size for the CMT application), the algorithm runs at about 15-20 frames per second on an Intel Core i7 processor being clocked at 1.6 GHz with 8 GB of RAM. However, at larger frame sizes, such as 1920x1080 with high resolution, the average frame rate of the algorithm is only about six frames per second. The problem with processing at low frame rates is if a camera is recording at 30 frames per second and the algorithm can only process six frames per second, the algorithm can only process one in every five frames. This can result in larger distances between objects in consecutive processed frames if the object, or camera, is moving, which will decrease the accuracy of the object tracking. Therefore, the goal of this thesis is to provide a system that can perform object tracking at as close to real-time (30 frames per second) as possible with large, high resolution frames.

Because the enhanced CMT algorithm uses SURF keypoints and descriptors that are more robust than BRISK, the tracking tends to be much slower in software. The average processing time for a 1920x1080 frame is about 1.4 seconds, which is only 0.7 frames per second. The following figure shows the breakdown of the major parts of the enhanced CMT algorithm.

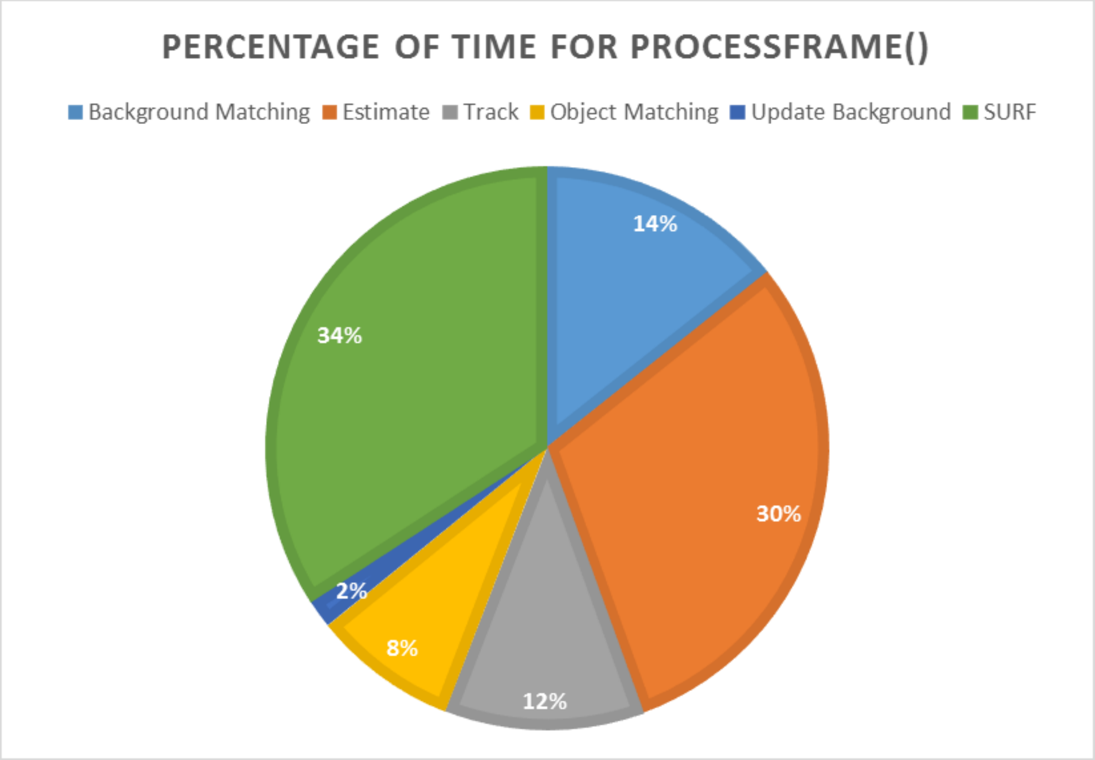


Figure 9. Pie chart of average timing percentages per frame for enhanced CMT algorithm using a frame of size 1920x1080.

As Figure 9 shows, the largest portion of enhanced CMT is spent processing SURF keypoints and descriptors, which is 34% of the total time. Because of this large percentage of time, the enhanced CMT algorithm would greatly benefit from a SURF accelerator that computed both the keypoints and descriptors. The second largest portion of the algorithm is the estimation step, which calculates the estimated center using clustering techniques and also estimates the scale and rotation of the ROI. This step can also be subject to acceleration. Lastly, the background and object matching steps take up the next largest amount of time in this algorithm. Because background matching uses almost all of the same techniques as the object matching step, these two steps can use one configurable matching accelerator. Background matching takes longer than object matching purely because there are more keypoints to be matched. The following figure shows the breakdown off times in the background matching step, and is also representative of the object matching step (minus the homography step).

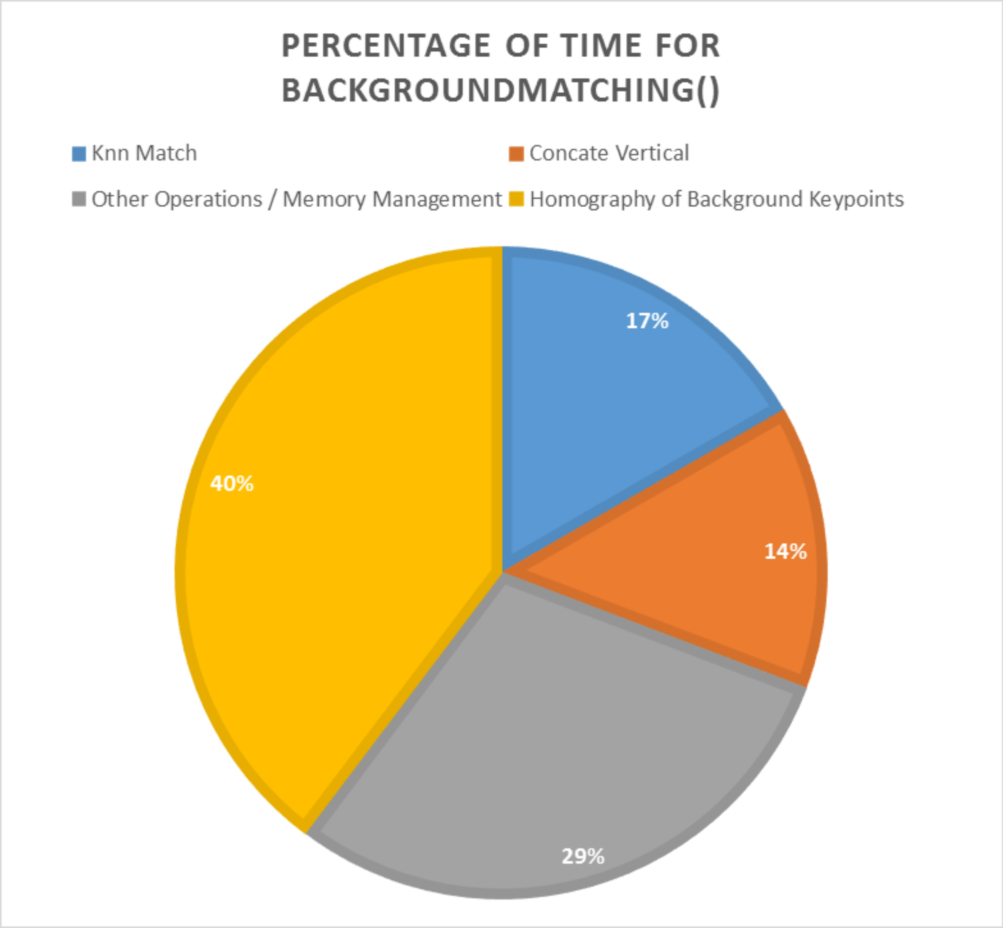


Figure 10. Pie chart of average timing percentages per frame of the background matching step for a frame size of 1920x1080.

Currently, this figure shows four separate steps that are done separately in software. When done separately, the homography algorithm, which detects outliers in a set of keypoints, takes up the most time at 40%. Homography is the last step in background matching, and is not done in object matching at all. Therefore, this can easily be done in software after the matches are computed using the matching accelerator. The key part of the matching steps for both the background and object models is knn-matching (where k equals two). This, along with the “concate vertical” and “other operations/memory management” portions of the figure, make up the core of descriptor matching. Unfortunately, memory management (reads and writes) in software can be expensive. If a matching accelerator can handle both the brute-force knn-matching and the logic to determine whether or not the match is acceptable, then the software does not have to deal with memory management issues and the whole matching step will be sped up significantly. The next chapter will look at architectures for both the descriptor matcher and SURF algorithm.

CHAPTER 3

HARDWARE ARCHITECTURE

The goal of this work is to speed up the enhanced CMT algorithm using FPGA accelerators to offload the more computationally intensive algorithms from software. By doing this, the throughput of the system will greatly increase and, hopefully, will reach real-time processing speeds. Although the whole enhanced CMT algorithm could be put onto an FPGA, this work will only explore two algorithms to accelerate: the descriptor matching algorithm and the SURF algorithm. Working with both software and hardware allows the tracking algorithm to be modular, which means there can be different parts of the enhanced CMT algorithm happening on both hardware and software with the ability to add more hardware modules in the future.

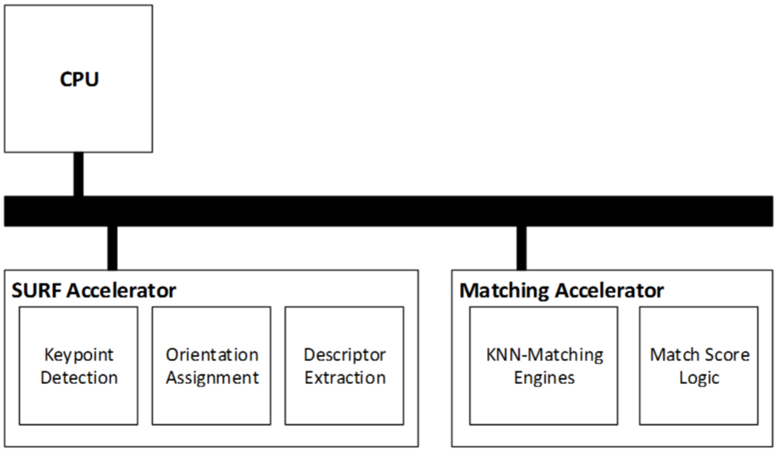


Figure 11. High-level overview of the whole object tracking system.

The above figure shows the overview of the whole tracking system, where the SURF and matching accelerators are done on an FPGA, and are connected to the CPU that will be performing the rest of the algorithm. The following sections will give an overview of the SURF accelerator, currently in development, and discuss, in detail, an architecture of the descriptor matching algorithm.

3.1 SURF

The SURF accelerator takes the current SURF algorithm that is used in OpenCV and speeds it up to compute the keypoints and descriptors faster. There are three main parts of this accelerator: keypoint detection, orientation assignment, and descriptor extraction. Keypoint detection encompasses three sub-steps: integral image generator, fast hessian filters, and keypoint localization. The following figure shows a high-level overview of the different steps in this accelerator.

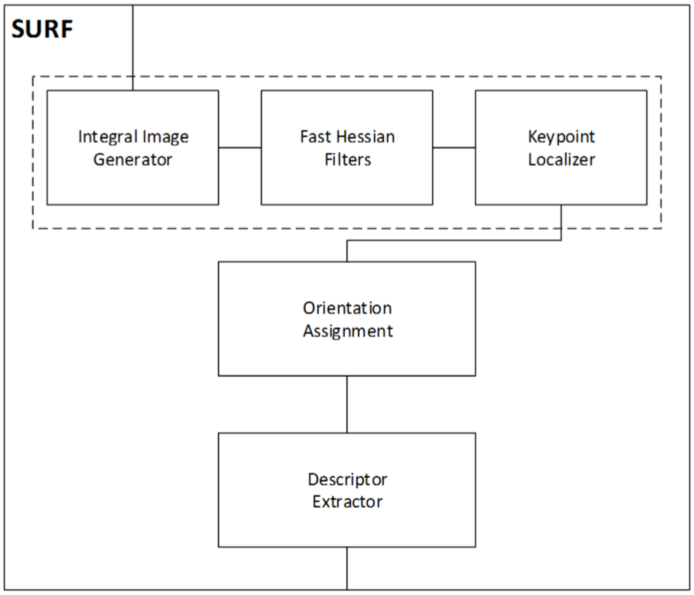


Figure 12. High-level block diagram of the SURF accelerator.

The next three subsections will go into more detail about the architecture and algorithms used SURF accelerator.

3.1.1 Keypoint Detection

The first step in SURF keypoint detection is to calculate the integral image. The integral image is used for the Hessian filters [12]. The calculation for the integral image is explained in detail in [16], but the following figure shows a representation of the calculation.

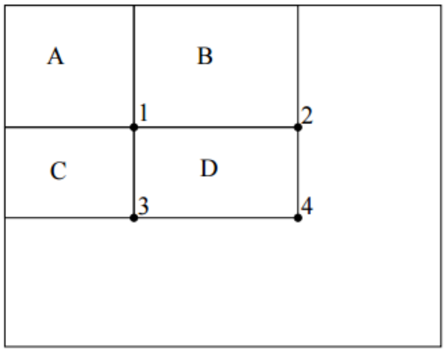


Figure 13. Example of integral image calculation.

Calculating the integral image is simply taking the intensity values of all the pixels within a rectangle, formed by a point and the origin, and summing them up. For example, in Figure 13, the integral image value at Point 1 is the sum of all of the pixel values in Rectangle A. However, if an integral image value is needed for a specific rectangle, such as rectangle D, the value can be computed by taking the pixel value at Point 4, adding it to the integral image value at Point 1, then subtracting both of the integral image values at Points 2 and 3.

The integral image values are very useful to make the Hessian Filter step fast, which is the next step of keypoint detection. The inputs to these filters are Gaussian second order partial derivatives, but because the integral image was calculated, the integral image values can be used as a good approximation of these partial derivatives [12]. The fast Hessian block contains eight different box filters, which the values from the integral image go through before outputting a response to the keypoint localizer. This step is thresholded by a value determined before running the algorithm. A larger threshold value for the Hessian responses will result in fewer keypoints produced by this step. However, too small of a threshold means the algorithm will produce more, less informative keypoints that will add noise for the other steps, such as matching.

The keypoint localizer is the final step in keypoint detection. A non-maximum suppression is applied in a certain neighborhood around each response [12]. Then, the maximum of the determinant of the Hessian responses are interpolated in scale and image space using a method proposed in [17].

3.1.2 Orientation Assignment

Once all of the keypoints are detected, the orientation of each keypoint is assigned. This step is critical in order for the features to be rotation invariant. First, the Haar-wavelet response is calculated in the x and y directions, which occurs is in a circular region around the keypoint with a scaled radius according to the scale at which the keypoint was found and the current scale [12]. After the wavelet responses are calculated, each one is weighted using a Gaussian distribution from the center of the keypoint and are represented by vectors [12]. Finally, the horizontal and vertical responses are summed using a sliding orientation around the center with an angle of π/3. The maximum of the summations reveal the dominant orientation of the keypoint.

3.1.3 Descriptor Extraction

The last step of the SURF algorithm is to extract the descriptors from each of the keypoints. First, a square, scaled region is constructed around the keypoint, aligned in the direction of the keypoint. Then, more Haar-wavelet responses are calculated in smaller sub-regions of the larger square in both the vertical and horizontal directions according to the orientation of the square region [12]. The responses are then summed up and combined to produce a descriptor vector with length of 64.

3.1.4 Performance Metrics

Although the SURF accelerator is currently being implemented, it is not currently to a point where it can be fully evaluated for an exact latency time. Therefore, the following equation is a good estimation of the total latency of the SURF algorithm.

$$SURF\ Latency = \frac{[(\#\ of\ pixels\ in\ an\ image) * (K_L)] * O_S}{Clock\ Frequency}$$

This equation is based on the size of an image (in pixels) because of the keypoint detection step must go through the whole image. The factor of K_L being multiplied to the number of pixels represents the average number of cycles to compute each keypoint and descriptor pair. In this architecture, K_L is equal to 11 clock cycles. O_S is the overhead associated with memory bottlenecks. For the current implementation, the overhead is about five percent of the total value, so O_L is estimated to equal 1.05. This model is for an architecture that is pipelined, but not fully streaming. The ideal case is a fully pipelined and streaming architecture, in which case the following equation can estimate the total latency.

$$SURF\ Latency_{ideal} = \frac{[(\#\ of\ pixels\ in\ an\ image)] * O_{Sideal}}{Clock\ Frequency}$$

The K_L factor is deleted from the equation because of the fully streaming nature of this architecture because each cycle will produce a keypoint. The only increase in this model equation is the overhead value, where O_{Sideal} is now estimated to equal 1.08 (an additional three percent overhead from the first model). This streaming architecture will make the SURF accelerator extremely fast and will be able to perform SURF keypoint detection and descriptor extraction in greater than real-time (30+ frames per second).

3.1.5 Software Integration

The SURF accelerator will be called upon once per frame by the CPU in order to get all keypoints and descriptors in the entire image. When initialization occurs, the hardware accelerator will provide the software with all of the keypoints and descriptors, so that software can set the object and background models to the corresponding keypoints. Every frame after initialization will use the SURF accelerator to get the keypoints and descriptors that will be used to match against the background and object models. Because SURF is the most time consuming process of the enhanced CMT algorithm, this accelerator will produce a large speed up in the overall system and be able to help the tracker reach real-time performance.

3.2 Descriptor Matching

The descriptor matching algorithm tries to match observed descriptors to a set of model descriptors and produces the two best match scores for each observed keypoint. It then determines whether or not the best match is confident enough to be considered a true match to a model descriptor. It does this by using a brute-force distance computation method where each observed keypoint is given a match score to each model keypoint. A match is considered to be a close match when the distance between the two descriptors is close to zero. The match scores can then be used to do other calculations, such as confidence and difference ratio between the two closest matches.

The following subsections will introduce the descriptor matching architecture, from a high-level overview, to more detailed descriptions of the more complex modules.

3.2.1 Overview

The architecture for the descriptor matcher is made up of two main modules: a controller and a datapath. The controller consists of a state machine and takes overall controll of the matcher. The datapath is where all computations are performed. This includes the streaming of data in and out of the module. The following figure shows a high-level design of the datapath module.

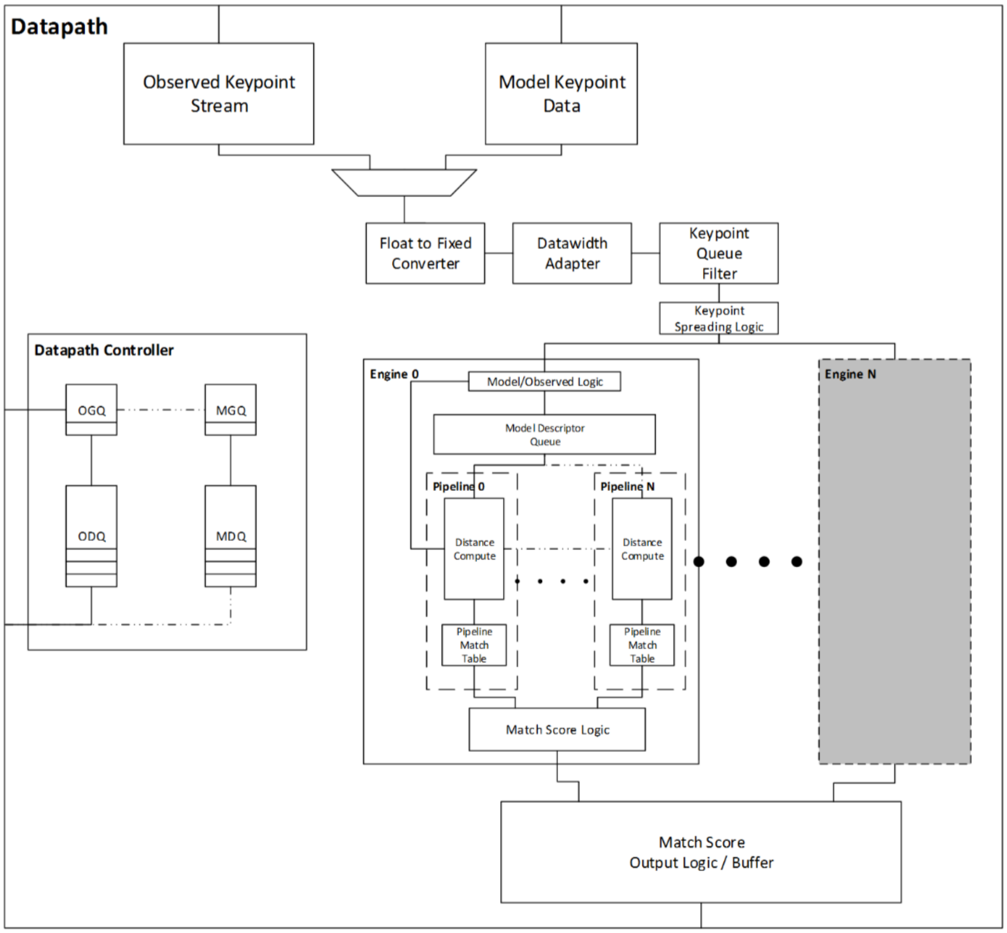


Figure 14. Overview of the descriptor matcher datapath module.

In this overview, there are three main sections: the datapath controller, input data logic, and computational logic. The datapath controller acts like a traffic controller, making requests for data and directing it through the datapath. The input data logic, consisting of the observed keypoint stream, model keypoint data, and the keypoint conversion/spreading logic, takes care of data coming into the datapath (both the observed and model keypoints) and converts them to applicable data types before distributing them to the different engines. Finally, the computational logic consists of the engines where the match scores are calculated (there can be up to eight of these in a datapath), and the match score output logic/buffer that takes the best score from all of the engines in respect to an observed keypoint and buffers it until needed by an external source.

3.2.2 Datapath Controller

The job of the datapath controller is to request the necessary data from memory and direct it to the correct place in the datapath. It is possible to use one controller for each engine, but it is better to let one overall controller be in charge of the incoming data and distribute it to each of the engines. This not only reduces the complexity of the engines, but it also saves resources by not having to duplicate controllers in each of the engines. The following figure is a representation of the datapath controller.

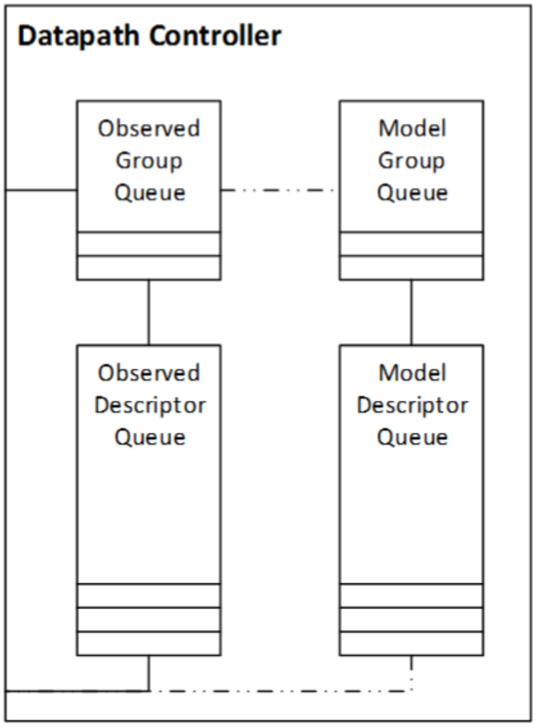


Figure 15. Block-level design of the datapath controller.

Although the controller will mostly be a state machine controlling signals and requests, there are two main sets of queues being used for the controlling of data requests. The first set of queues are the observed and model group queues.

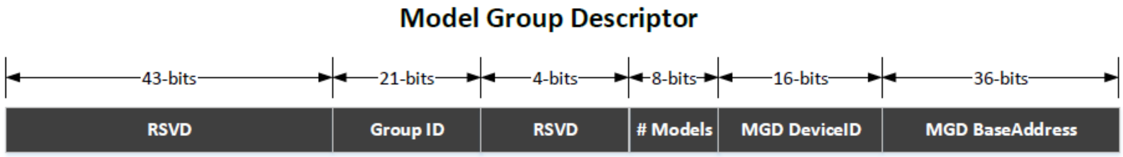


Figure 16. Description of the 128-bit Model (Observed) Group Descriptor stored in the Model (Observed) Group Queue.

Figure 16 shows the 128-bit model group descriptor. This descriptor has the same layout as the observed group descriptor. Each group descriptor points directly to a keypoint descriptor, show in Figure 17. The MGD base address and device ID is a combination of bits that points to the address of the first of up to 255 model descriptors, represented by the “# Models” field. The model (or observed) group queue will store these descriptors in order make requests to memory for the model descriptors when needed.

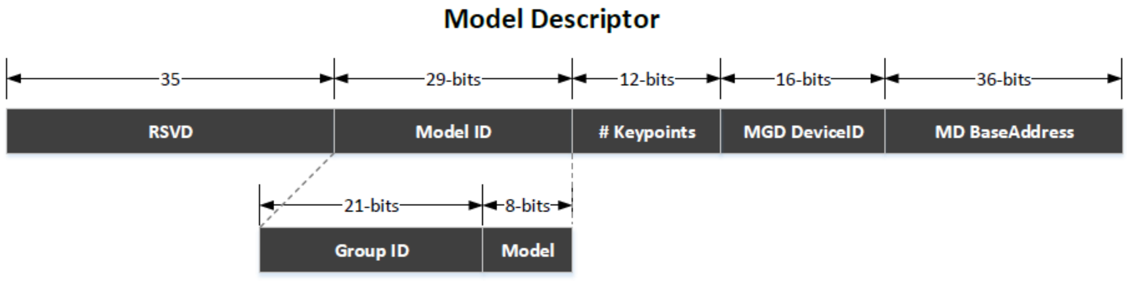


Figure 17. Description of the Model (Observed) Descriptor stored in the Model (Observed) Descriptor Queue.

The above figure is the bit layout for the model (or observed) descriptors. These descriptors are pointed to by the group descriptors, and are set up as a list. There can be up to 255 model descriptors per group descriptor, and each model descriptor can have up to 4,095 keypoints associated with it, which are all stored in a list. The MD base address and device ID fields are used to access the first keypoint's data. These descriptors are used to make requests for individual keypoint's descriptors that will be loaded into the different engines depending on whether they are observed or model keypoints.

3.2.3 Input Data Logic

The input data logic includes the two keypoint data paths as well as some data manipulations before sending the descriptors to the engines. The following figure is a representation of this logic.

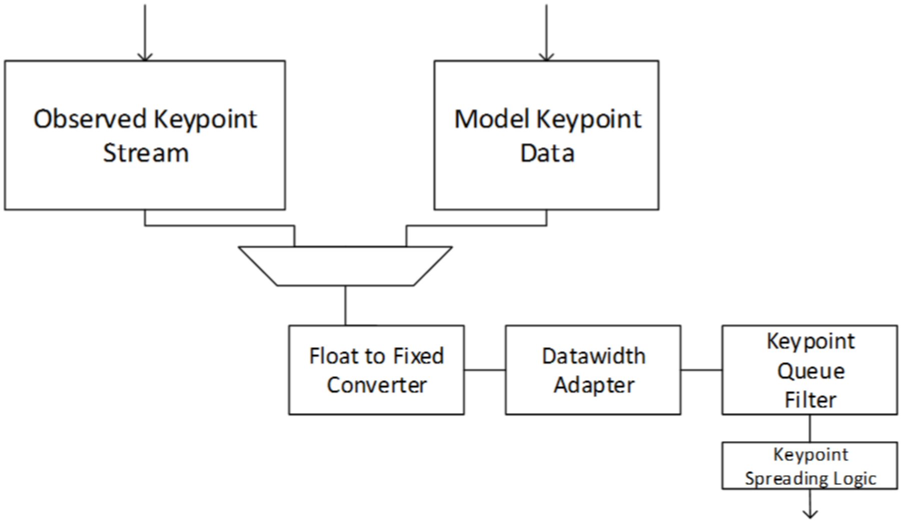


Figure 18. Block diagram of the input logic for the matching accelerator.

The observed keypoint stream is a stream of descriptor data from each of the requested keypoints from the current image, while the model keypoint data is descriptor data coming from memory that was stored from a previous model. The descriptors will never be loaded into the datapath at the same time, which allows the datapath to make resource optimizations. The multiplexor shown in the above figure, controlled by the datapath controller, selects which data is to be sent through to the rest of the input logic.

The first step is a float to fixed data converter. This simply takes the floating point representation of the descriptor (which is used in software) and converts it to a fixed point representation of the same value. The input to the module is 128-bits, which is split up into two 64-bit floating point data structures. The output is a 128-bit representation of four 32-bit fixed point data structures.

The second step of the input logic is a data width adapter. The goal of this step is to buffer the descriptor data for each keypoint, plus 128 bits of information about the keypoint, until all 64 descriptor dimensions at 32 bits each have been buffered. Then, the output is a descriptor vector of 2,176 bits. These data structures are equivalent to one SURF feature in software and will be used in all of the calculations for matching.

Next, the SURF feature will go through a keypoint filtering step where keypoints are filtered out of consideration if not needed. Keypoints do not come into the datapath in a manor where it only gets the keypoints that the datapath needs. Being able to filter unwanted keypoints is a crucial step. For example, if wanting to only get keypoints inside a specific ROI while a large amount of keypoints are coming into the datapath, it is possible for the keypoint filtering step to get rid of any keypoints that are outside the ROI.

Finally, the keypoint descriptors will be spread out to all applicable and available engines to perform the match score calculations. Since there can be up to eight engines in the datapath, each engine can be configured to work on different models. This can be useful for object detection and matching when trying to match an ROI to a known model. Different engines can also work on the same model, such as the background model, when there are a large number of model keypoints. The keypoint spreading logic will determine the engines that have space for descriptors (in the case of model descriptors being loaded into engine queues) and which engines are working on which models. For observed descriptors, each one will need to be spread to all of the engines currently working on model descriptors because each observed keypoint must be matched to all model keypoints.

3.2.4 Computational Logic

The last step of the descriptor matching datapath is the computational logic section, which is made up of the matching engines and match score logic. The following figure is a high-level block diagram of this section. More detailed architectures of some of the specific blocks will be explored in this section.

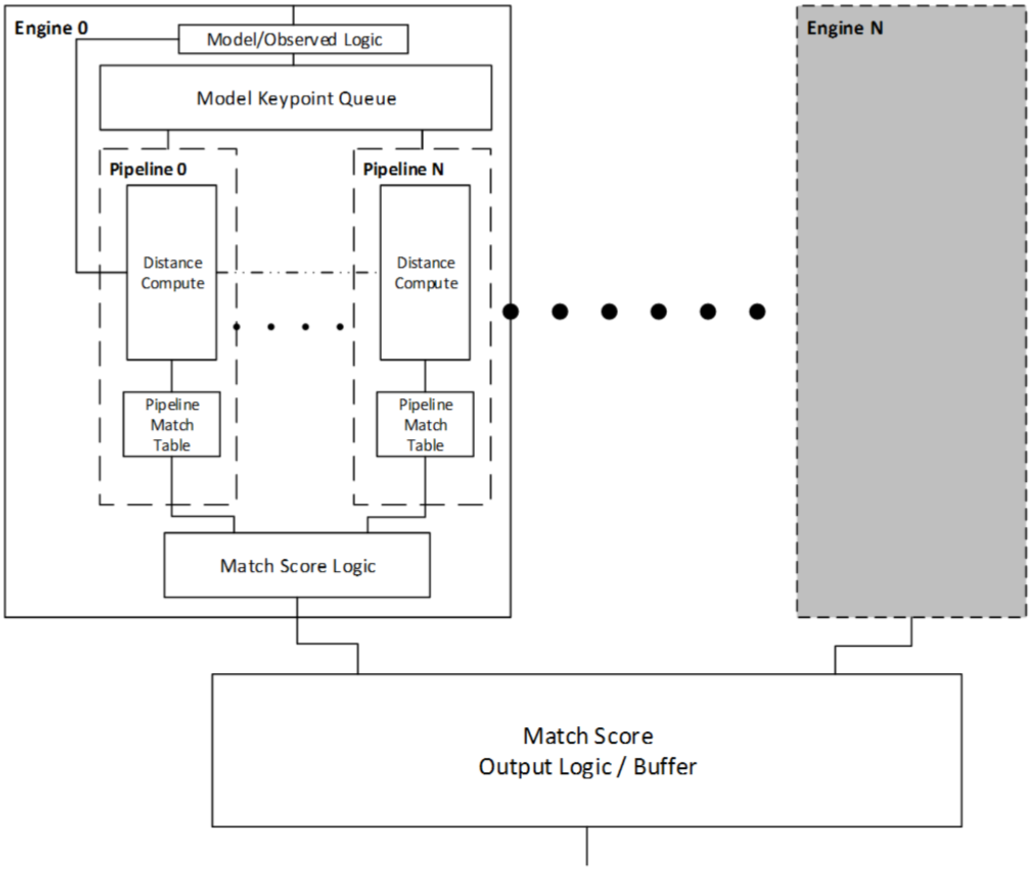


Figure 19. High-level architecture of the overall computational logic section in the descriptor matching datapath.

The computational portion of the data path is composed of matching engines. There can be up to eight of these engines. Each engine is responsible for computing a match score for each observed keypoint descriptor against all model descriptors loaded into the engine's model keypoint queue. The engine will then output the two best match scores to the model keypoint in the engine (using the match score logic block), and pass those to the overall datapath match score output logic/buffer block. This last step will take the overall two best matches from the different engines and compute confidence and ratio scores to determine whether or not the descriptor is a good match to the model descriptor.

There is also a configurability aspect of the datapath score output logic/buffer block. For the background subtraction step in enhanced CMT, the accelerator must keep track of both good and bad matches to the background model. Good matches will be given to the homography algorithm to detect outliers and bad matches will be used for the object matching step. So, for descriptor matching used in background subtraction, the datapath will be configured to output both good and bad matches, but when performing object descriptor matching, it will only be configured to output good matches.

3.2.4.1 Matching Engines

Figure 19 shows a high-level representation of one matching engine. Within the engine there are four main parts: model/observed logic, model keypoint queue, computational pipelines, and match score logic. The first step is to have logic to determine type of data is coming into the engine, either a model or observed descriptor. The type of data will determine where it will be sent in the engine. In the case of model descriptors, the data will be sent to the model keypoint queue. If the data is an observed descriptor, then the descriptor will be sent to each active pipeline where match scores can be calculated with respect to every model descriptor.

Next, the model keypoint queue is a set of BRAMs that will hold all of the model keypoints for that particular engine. Each pipeline will only compute match scores on a subset of this queue, depending on how many pipelines are being used. The model descriptors will be stored in the queue until all observed keypoints in an image have been streamed through, at which point the current descriptors will be deleted from the queue and new ones will be loaded.

Following the model descriptor queue are the computational pipelines. These pipelines take care of all of the distance computations. After the model keypoint queues are populated in the engine, observed descriptors will be loaded into the distance compute block of the pipeline. The following figure represents the architecture for the distance compute block.

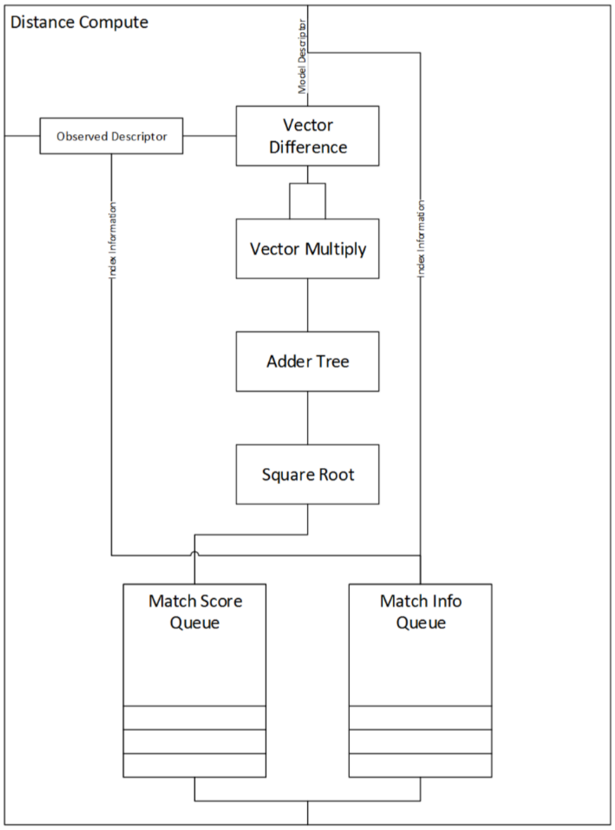


Figure 20. Architecture of the distance compute block for computing match scores.

The distance compute block in the computational pipeline represents the calculation of match scores for each observed and model keypoint pair. Because every observed keypoint has to be matched to every model keypoint, the number of calculations will be $K_O * K_M$, where K_O is the number of observed keypoints and K_M is the number of model keypoints. However, these computations are spread across different pipelines and engines, which will reduce the amount of time to calculate all matches.

The first thing the distance compute block does is load an observed keypoint's descriptor vector. This vector will be used for the distance calculation until all stored model descriptors have been matched. The match score calculation follows the equation for a Euclidean distance measure:

$$Match\ Score = \sqrt{\sum_{i=1}^{64} (OD_i - MD_i)}$$

Where OD_i is the i^{th} dimension of the observed descriptor vector and MD_i is the i^{th} dimension of the model descriptor vector.

To start the pipeline, model descriptors will be fetched from the model keypoint queue and sent to the vector difference block, which takes the difference between the model and observed descriptors. This difference happens at each dimension of the descriptor vector, so there are 64 32-bit subtractions happening simultaneously. As soon as this calculation is done, the result is sent to be squared in the vector multiply block and another model descriptor vector can be sent into the pipeline. The vector multiply block takes two descriptor vectors as inputs and multiplies them together. But in this case, the vectors will be exactly the same, so the result will produce a squaring of the difference between the observed and model descriptors. Next, the adder tree sums up all of the different dimensions of the squared descriptor vector. This result alone will give the L1, or Manhattan, distance for the match score. However, to be more accurate, the square root is taken and the resulting value is the L2, or Euclidean, distance between the two descriptor vector. The smaller this distance is, the better the match.

Once the match score is calculated, it is placed into a match score queue, along with the information associated with the score, such as the indices of the observed and model keypoints being matched. The two outputs of the queues will merge into one data structure and be sent from the distance compute block to the pipeline match table.

The pipeline match table does a quick comparison between the match scores of each observed descriptor and outputs the two best matches from the pipeline. The same logic used in this step will be used in the engine match score logic block. Because the accelerator needs to returns the top two matches for each observed keypoint, each match table logic step must output the top two scores from each observed keypoint. The engine match score logic will output the top two match scores from each observed keypoint across all pipelines in the engine.

3.2.4.2 Datapath Match Score Output Logic/Buffer

Once the two best match scores for an observed keypoint have left each active engine, the datapath must choose the best two scores from across all engines. There also needs to be some post processing on these matches to determine whether or not the match to the model is good or not. The following figure shows this logic.

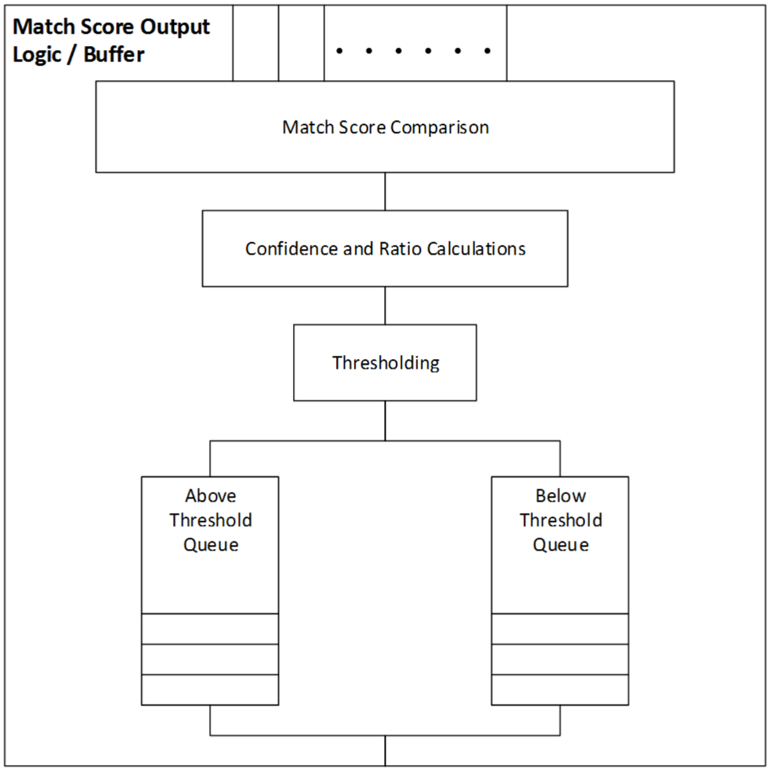


Figure 21. Match Score Output Logic/Buffer for the entire matcher datapath.

The inputs to this block are the two best scores from each of the active engines for a particular observed keypoint. The first step is to take the top two match scores from all inputs for a keypoint. The next step is calculating the confidence for the best match score as well as the ratio between the matches. The confidence and ratio equations are shown below.

$$Confidence = 1 - Best\ Match\ Score$$

$$Ratio = \frac{Best\ Match\ Score}{Second\ Best\ Match\ Score}$$

The confidence value shows how good a match is between the observed and model keypoints. If the confidence is equal to one (it will never be greater than one), then it is considered an exact match. The ratio value is the difference between the two matches. If this value is high, it means that the two matches are very similar and they could be confused with a wrong keypoint. However, if the ratio is low, the two matches are enough different from each other. The matcher wants to produce matches that are very different from the second best match.

Once these values are calculated, they are sent through a thresholding step that determines whether the values produced are good enough to be considered a good match. For the confidence thresholding, the value must be above a certain percent to be considered a good match, while the ratio value must be under a certain value to be a good match. If both of these values pass their thresholding step, then the keypoint is sent to the "Above Threshold Queue." If they do not meet the thresholding requirements, then there are two options. If the matcher is configured for keeping both good and bad matches, such as the case for background subtraction, then the bad match will go to the "Below Threshold Queue." But if it is configured to not care about bad matches, such as object matching, then they will just be thrown away. The matches will wait in the queues until called upon for data.

3.2.5 Performance Metrics

The proposed descriptor matching architecture will definitely speed up the matching process compared to software. Because this architecture is being explored and has not been implemented, the following equation is a good estimation as to how much latency can be expected in the worst case number of keypoints in this architecture.

$$Matching\ Latency = \left(\left(\left\lceil \frac{K_M}{E_N * K_E} \right\rceil * K_O * \left(\frac{K_E}{P_N}\right)\right) + L_{KM}\right) * \left(\frac{1}{Clock\ Freq.}\right)$$

$$L_{KM} = \frac{(D_N * B_D + B_H)}{D_B} * K_M$$

E_N is the number of engines being used in the datapath. P_N is the number of pipelines each engine has. K_M is the number of model keypoints for the current matching model. K_O is the number of observed keypoints in an image. K_E is the maximum number of model keypoints an engine can support. Lastly, L_{KM} is the time it takes to load all of the model keypoints for each engine. This value is equal to the number of descriptors in each keypoint (D_N) times the number of bits per descriptor (B_D), plus a header size for the keypoint in bits (B_H), all divided by the bit width of the data bus (D_B), and finally multiplied by the total number of model keypoints (each keypoint will need to be loaded into an engine).

Although the above equation can be simplified, the expanded version that is written is easier to explain. The first value in the ceiling function, K_M / (E_N * K_E), is the number of iterations that will need to be performed in order to cycle through all model keypoints. Depending on how large each engine's model keypoint queue is and how many engines are available determine the number of iterations the matcher must perform. The iteration value is multiplied by the number of observed keypoints and the worst case number of model keypoints per pipeline, K_E / P_N. Finally, L_{KM} will give a total load time for all model keypoints into the engines. The model keypoints are loaded one at a time into the queues before matching occurs, so all of the active engines have to wait until all model keypoints are loaded.

This architecture is designed in a pipelined, streaming fashion so that engines can compute one match score per cycle. This model equation assumes the worst case where all engine pipelines are full for each iteration. In reality, each engine will load model keypoints into the queue until it is full, then either load another engine or start streaming through observed keypoints. If more than one iteration is needed to match all of the keypoints, the available engines will fill up with model keypoints while the remaining keypoints are left for the next iteration. The other computations for the match table logic blocks are insignificant compared to the distance compute computations and should not be considered as adding a significant amount of latency to the model.

3.2.6 Integration with Software

This descriptor matcher accelerator will be used in conjunction with software. The software will offload all matching tasks to the accelerator and then receive either the good and bad matches (for background subtraction) or only the good matches (for object matching). For background subtraction, all observed keypoints and descriptors will be passed to the accelerator, which are computed by a keypoint and descriptor extraction module (either in hardware or software). This matching accelerator can be used for either SURF or SIFT [13] descriptors due to their nature. For enhanced CMT, the algorithm uses SURF descriptors. The accelerator will output the good and bad matches to the background, and the good matches will then have homography performed on them to make sure there are no outlier matches. The descriptors of the bad matches to the background, as well as the descriptors of the outlier background matches, are then passed to the matching accelerator once more to perform object matching. At this point, only the good matches are returned and set as active keypoints for the next iteration of the enhanced CMT processing cycle.

CHAPTER 4

RESULTS

With the SURF and descriptor matcher accelerators being discussed from an architectural standpoint in the previous chapter, this chapter will discuss the results of these accelerators and their performance increases with respect to time. In order to get good estimates of all possible cases, there will be a discussion of both average cases and best cases with respect to the parameters being chosen for the different performance equations given. Because the current SURF architecture can only support an image size of up to 1024x1024, the example in this section will use an image size less than this at 800x600 to represent an average frame size; however, larger image sizes will be discussed. Finally, several different configurations will be chosen for different target hardware.

4.1 Average Software Times for Enhanced CMT

This section will give the average times for the example case of a frame being of size 800x600 at high resolution. Although algorithm times were discussed in Section 2, these are for the new frame size and also give an overview of the code timings at a high level. The following figure is a breakdown of the average timings per frame.

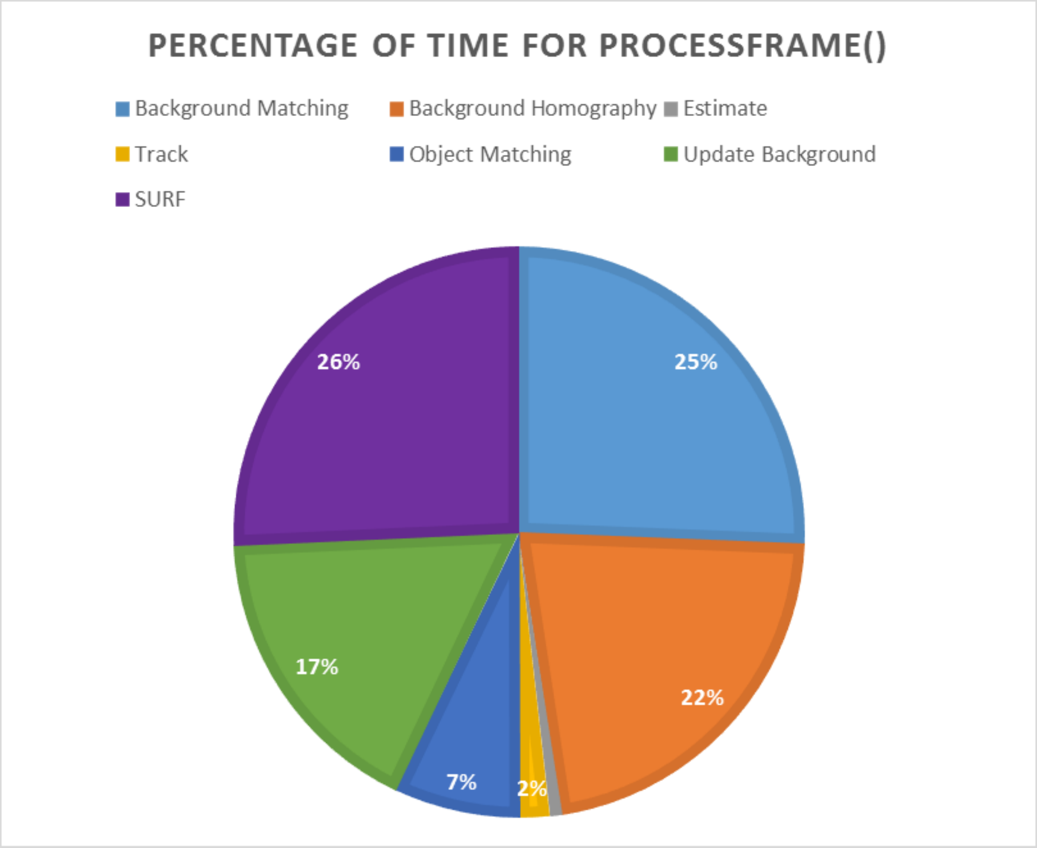


Figure 22. Performance timing percentages for an 800x600 size frame.

The figure above differs in some of the percentage values from the percentages of the 1920x1080 image size in Chapter 2. The main differences in this example video are the size of the object ROI (much smaller in this video) and the number of overall keypoints (much more in this video due to a more textured background). The smaller ROI increases the cost of the Update Background function due to memory management of creating a new descriptor matrix each time a keypoint outside the tracked ROI is added to the updated background model. A larger ROI will normally have less keypoints to add to the background during this step. The estimation process also greatly decreases in time in the new video, which can be attributed to the smaller number of keypoints within the ROI. The number of keypoints, shown in

Table 3 also increases in this video because there are more textured objects in the scene. This also contributes to memory management issues and the increased time it takes to do the homography of the background keypoints.

In order to show true estimates for this frame size, there must be averages for the number of keypoints being generated from each frame from different categories. The table below shows these numbers for the example video.

Total Keypoints	Adapted Background Model	Possible Object Keypoints	Object Model Keypoints
2582	2485	899	77

Table 3. Average number of keypoint in the example video.

The total number of keypoints is the average number of keypoints being extracted in each frame of the video. The adapted background model value represents how many keypoints are being used as the background model for background subtraction. The possible number of object keypoints is the total number of keypoints being matched to the object model (they were not matched to the background model). Finally, the number of object model keypoints, which is constant through the whole video, is the number of keypoints associated to the initial object model and are used as the model for object matching.

Also, in order to estimate the actual time taken in each step and the potential speed up of the accelerators, the following table shows these numbers as actual time in milliseconds. The average overall time to process one frame was 2,086 milliseconds.

Function Name	**Total Time (in milliseconds)**
SURF	536.3
Background Matching	534.8
Background Homography	458.0
Update Background	358.9
Object Matching	148.6
Track	35.4
Estimate	14.1
Total	**2,086.0**

Table 4. Average times per function in milliseconds.

The above times will be used to determine the estimated speed up from the two accelerators in the following sections.

4.2 SURF Algorithm Accelerator

The current SURF accelerator uses the non-streaming, pipelined architecture. This means that the model equation depends on the image size and will be multiplied by a certain factor according to the number of cycles it takes to produce one keypoint and descriptor pair. The following equation can be used to model the non-ideal architecture for an 800x600 frame size.

$$SURF\ Latency = \frac{[(800 * 600) * (11)] * 1.05}{100\ MHz} = 55.4\ ms$$

For the current architecture, the average expected number of cycles to produce one pair is eleven and the average overhead throughout the whole pipeline is an additional five percent. The total estimated time for the SURF accelerator is 55.4 milliseconds, which is almost a 10x speedup from the software version of SURF.

The ideal version of the SURF architecture, streaming and pipelined, provides an even faster estimate of latency, as shown below.

$$SURF\ Latency_{ideal} = \frac{[(800 * 600)] * 1.08}{100\ MHz} = 5.18\ ms$$

If the ideal architecture can be achieved, then for an 800x600 frame size, the SURF accelerator would be able to produce all keypoints and descriptors in 5.18 milliseconds. This is an almost 100x speedup over the software algorithm.

The following graph shows the latencies of the two versions of the architecture depending on the size of the frames.

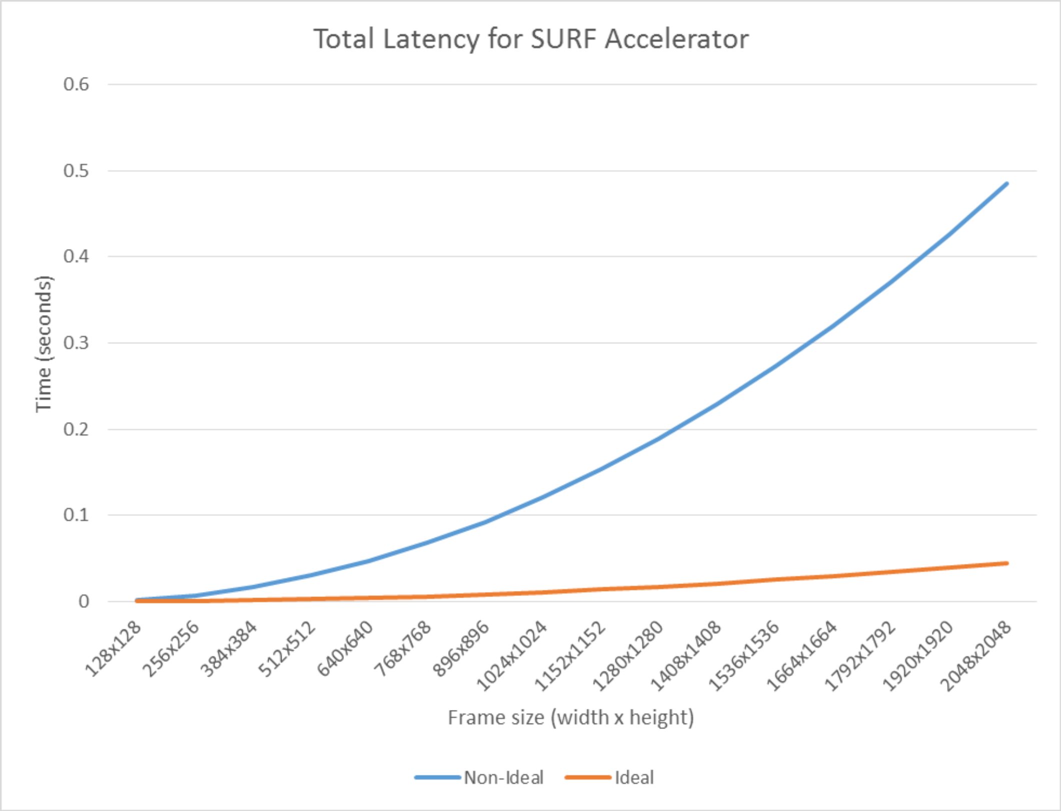


Figure 23. Total latencies of SURF accelerator for varying frame.

Although the current SURF architecture can only support up to a 1024x1024 frame size, increasing the amount of memory within the accelerator will increase this size. As shown in the above figure, the ideal architecture out performs the non-ideal version by a factor of 10. This ideal version is necessary in order to perform real-time tracking.

4.3 Descriptor Matching Accelerator

The improvements with respect to time for the descriptor matching accelerator are two-fold. One is the improvement of the actual match score computations, and two is the memory management time being performed by the hardware instead of software. By far the largest speed up will be due to the memory management improvements, but the matching times will also decrease compared to software.

For the example in this section, the total time for descriptor matching is 534.8 milliseconds per frame for background matching and 148.6 milliseconds per frame for object matching. To estimate the expected speed up, the keypoint values in Table 3 are plugged into the equation for descriptor matching in Section 3.2.5. This will produce an estimated time to complete the background matching and object matching steps. Also, assume that clock frequency is set to 100 MHz for all accelerators.

$$\begin{aligned} &Background\ Matching\ Latency \\ &\qquad = \left(\left(\left\lceil\frac{2,485}{4 * 1,024}\right\rceil * 2,582 * \left(\frac{1,024}{8}\right)\right) + 42,245\right) * \left(\frac{1}{100\ MHz}\right) \\ &\qquad = 3.73\ ms \end{aligned}$$

The previous model uses four engines and the maximum number of pipelines per engine (eight). This model assumes that each engine can hold up to 1,024 model keypoints, a standard depth for BRAM. So, the total worst-case estimated time for the accelerator to perform background matching is only 3.73 milliseconds, which is about a 700x speedup from software.

The same type of estimation can be done for object matching, which is shown below.

$$Object\ Matching\ Latency =$$

$$= \left(\left(\left\lceil \frac{77}{1 \ast 1{,}024} \right\rceil \ast 899 \ast \left(\frac{1{,}024}{8}\right)\right) + 1{,}309\right) \ast \left(\frac{1}{100\ MHz}\right) = 1.16\ ms$$

Due to the small number of object model keypoints, one engine will be plenty for this example, and probably for most object matching steps (unless there are greater than 1,024 keypoints in the model). But, there are still eight pipelines to increase the speed of the matching for an optimal configuration. The total worst-case latency for the object matching step is only 1.16 milliseconds, which is over 130x faster than software.

Although these speedups are very significant, it is because a large amount of hardware is being dedicated to the matching. For the background matching step, there is a possibility of speeding it up by a factor of two by using a total of eight engines with eight pipelines each, but this would not be possible due to the limited amount of hardware resources in most platforms. There will be a slight slowdown of these accelerators due to the matching logic of the match tables and the additional logic to perform the confidence and ratio calculations, but these are not significant.

In order to get an overall estimation for the descriptor matching accelerator, the following graph shows different configurations for the number of engines and pipelines per engine based on the number of keypoints in the image. For simplicity, the number of keypoints on the x-axis is equal to both the number of observed and model keypoints. For background matching, the number of model keypoints is usually only slightly less than the number of observed keypoints.

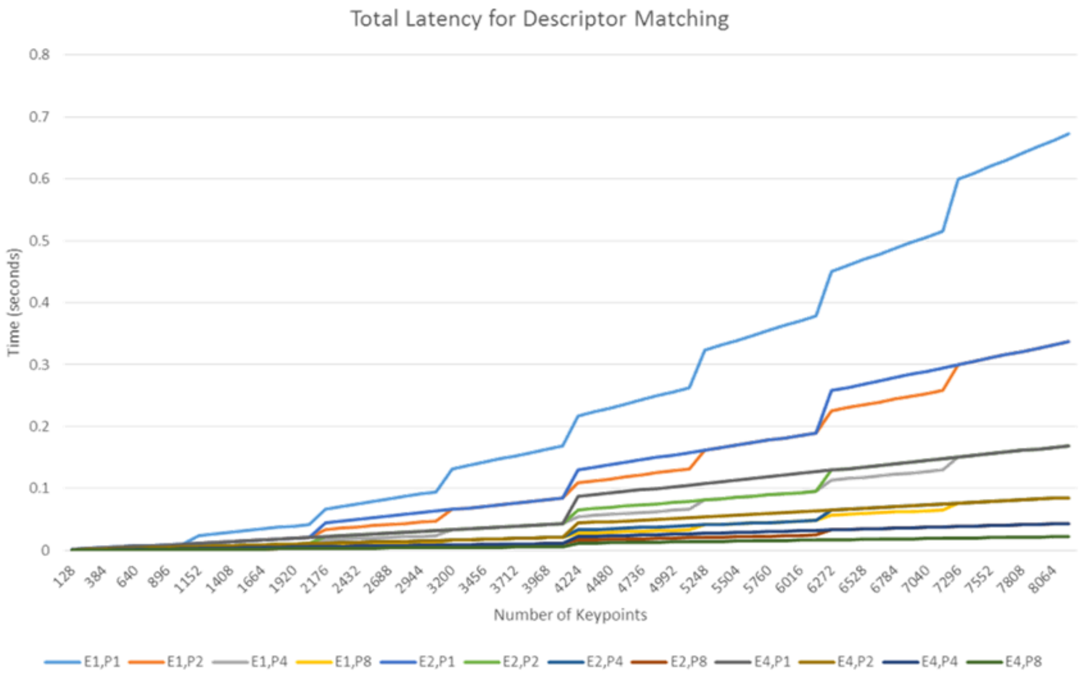


Figure 24. Overall times for descriptor matching based on number of keypoints and the number of engines and pipelines with maximum engine queue of 1,024 keypoints.

Since the accelerator can only support up to 8,192 keypoints in the model queue, this is the upper limit of the graph. The legend indicates the number of engines (E) and the number of pipelines (P) per configuration. All of the engines in these configurations have a maximum queue size of 1,024 keypoints. As the number of pipelines and engines increase, the total processing time decreases. To show the possibilities of engines having up to 2,048 total keypoints in their queues, the following graph gives the times for this configuration.

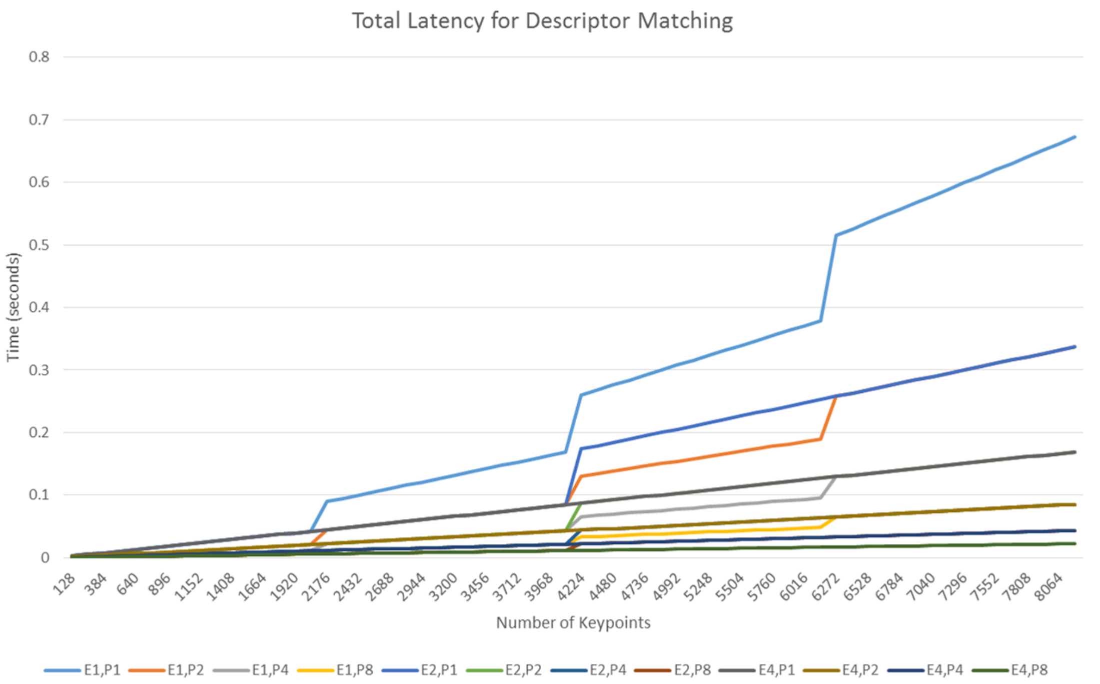


Figure 25. Overall times for descriptor matching based on number of keypoints and the number of engines and pipelines with maximum engine queue of 2,048 keypoints.

As shown above, the total times go down, on average, for all configurations. The differences in time between the configurations also gets smaller do to the decreased amount of iterations needed per configuration. Therefore, as the size of the model queue in the engines increases, and the number of total pipelines increases, the total processing time will decrease.

4.4 Overall Enhanced CMT

With the two accelerators, SURF and matching, added into a hybrid object tracking system (split between hardware and software), the tracking will perform much faster than the pure software version. However, these two accelerators alone will not allow the tracking to be done in real time, even with the ideal SURF architecture. The following table shows the performance of the overall system with the two accelerators added in the case of the 800x600 video example.

Function Name	**Total Time (in milliseconds)**
SURF Accelerator	5.18
Background Matching Accelerator	0.423
Background Homography	458.0
Update Background	358.9
Object Matching Accelerator	0.013
Track	35.4
Estimate	14.1
Total Time	**872.0**

Table 5. Total time for tracking functions with the SURF and matching accelerators.

As shown above, the total time for this example is still 872 milliseconds, which only give the overall algorithm a speed of 1.15 frames per second. This is about 30x slower than needed to perform object tracking in real time. Therefore, other parts of the algorithm must be accelerated.

Ideally, the whole system would be performed in hardware. However, this might not be possible due to the resources available on a given platform. Figure 7 and Figure 22 both show some motivation to accelerate the estimate and homography functions. When the number of background keypoints increases, the background homography and update background functions have increased latency times. Also, when the number of object model keypoints increases, so does the time to perform the estimate function. Because of these two cases, accelerators must be created for both functions in order to get the system to process in real time.

4.5 Target Hardware Configurations

The two target hardware platforms that this work is targeting is a cloud/desktop-based platform (Virtex-7 690T) and a smaller embedded platform (Zynq 7045) that could be considered a wearable device. The cloud/desktop-based system will have more resources to use for acceleration, while the embedded platform will have far less resources to work with.

The number of DSPs and BRAMs will greatly affect which configuration of descriptor matcher that is chosen for each platform. First, to determine the amount of DSPs that are used in each engine pipeline, the architecture in Section 3.2.4.1 shows that there are several main computations for the distance compute, each needing one DSP. There are 64 subtractions, 64 multiplications, and 63 additions, which totals close to 200 DSPs (rounding for overhead purposes). Also, there will be four lookup tables required for each square root function, where each lookup table is one BRAM. Each engine also needs BRAM to store the model keypoints. Each BRAM is 36 bits wide by 1,024 entries deep. So, to store a set of descriptors for one keypoint, there needs to be about 60 BRAMs per engine. This will allow the descriptors (64 descriptors x 32 bits) to be stored in the engine. This also means that the engine can store up to 1,024 keypoints.

Virtex-7 690T	Configuration 2E, 8P, 4096Q	Device Total	Percentage of Total
DSP	2E*8P*200DSP = 3,200	3,600	89%
BRAM	2E*4096/1024*60 = 480	1,470	33%

Table 6. The total DSP and BRAM usage amounts for the Virtex-7 690T.

Zynq 7045	Configuration 1E, 4P, 4096Q	Device Total	Percentage of Total
DSP	1E*4P*200DSP = 800	900	89%
BRAM	1E*4096/1024*60 = 240	545	44%

Table 7. The total DSP and BRAM usage amounts for the Zynq 7045.

The above tables show the total amount of DSPs and BRAMs per platform and the different configurations that can be made with the number of engines and pipelines. If wanting to maximize the hardware for descriptor matching, there can be a maximum of 18 pipelines (900 DSPs and 72 BRAM) for the Virtex-7 690T and only 4 pipelines (800 DSPs and 16 BRAMs) for the Zynq 7045. So, if the configuration for the Virtex-7 platform uses two engines and eight pipelines per engine, with a maximum support of 4,096 model keypoints per engine, there would be a total of 3,200 DSPs and 1,470 BRAMs being utilized. As for the Zynq 7045 platform, a configuration of one engine with four pipelines, and a maximum support of 4,096 model keypoints per engine will use 800 DSPs and 240 BRAMs. The following figure shows the graphs for the performance of the two configurations.

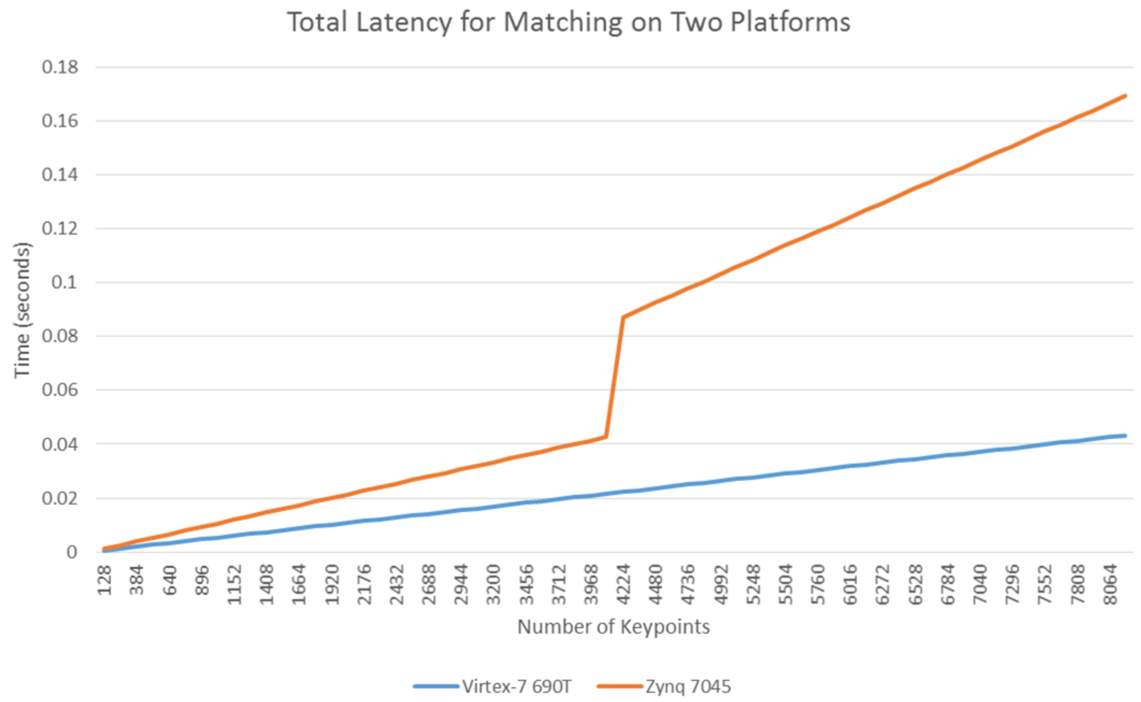


Figure 26. Total latencies of descriptor matching for two different platform configurations.

As shown above, the Virtex-7 690T performs much better than the Zynq 7045 because of the increased amount of resources available. Because the embedded system would not be able to do real-time matching with a large number of keypoints (for the worst case model), it would be easy to either limit the amount of keypoints being matched by raising the SURF threshold value, or by using a camera at lower resolution, the amount of keypoints will be decreased. These modifications can ultimately affect the accuracy and robustness of the enhanced CMT algorithm, but it should not be so significant that an object cannot be tracked. For the cloud/desktop-based platform, a larger amount of resources are available and can do real-time matching with many more keypoints.

CHAPTER 5

CONCLUSIONS

This thesis has introduced the Consensus-based Matching and Tracking (CMT) algorithm and has made improvements to the algorithm in software. It was shown that the enhanced version of CMT outperformed both the original CMT algorithm and OpenTLD (Tracking-Learning-Detection). The robustness of this algorithm is more important than perfect accuracy, and the enhanced CMT version has been proven to provide increased robustness for a system with no user intervention while tracking.

Once the improvements were made and compared, two accelerators were proposed to speed up the algorithm: SURF and descriptor matching. The architectures of these accelerators were discussed and the equation models for their latencies were shown. Lastly, two different platforms, cloud/desktop-based and embedded, were used as real examples of target platforms and their ideal configurations were discussed.

Although these two accelerators will be able to speed up the enhanced CMT algorithm by a significant amount, there are still a few functions left on software that can be costly to the real-time performance of this system. These functions are estimate center and background homography. In order to perform tracking in real time, these accelerators must be designed and implemented. However, until these accelerators are developed, the functions can run in software and the system can still benefit from the use of the SURF and descriptor matching accelerators.

Once the system can track objects in real time, the end goal is to track multiple objects in a video at the same time. Currently, trying to do this is extremely slow in software. It takes almost two times the amount of latency for tracking two objects as opposed to one. Therefore, if able to speed up the processes, the system could realistically track two objects.

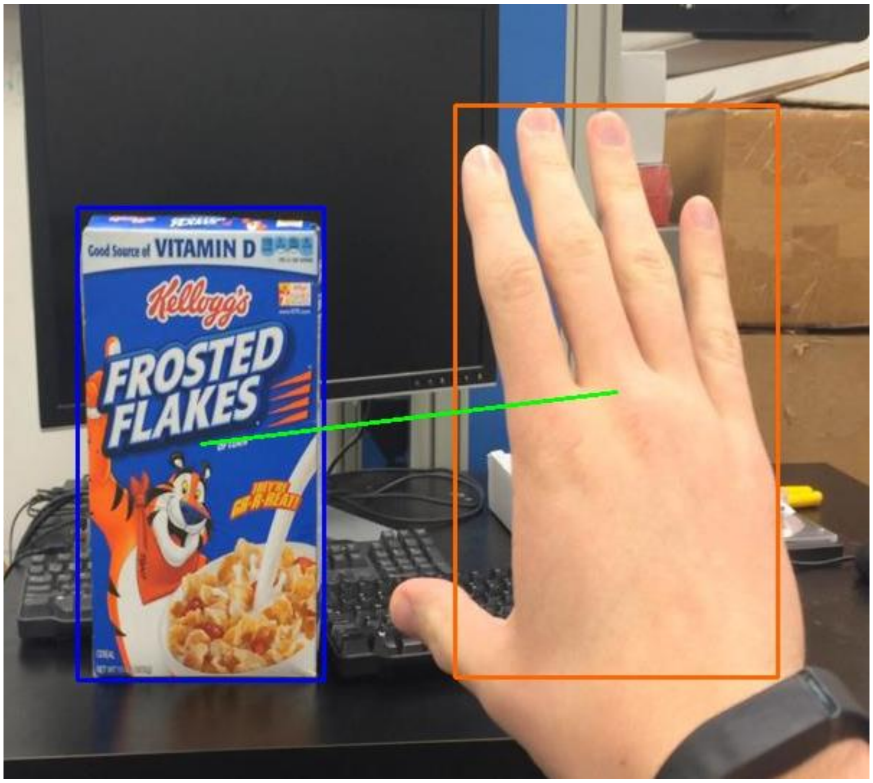


Figure 27. Image representing a video that is tracking two objects simultaneously.

Being able to track two objects in real-time will greatly improve the user experience for a system that assists visually impaired individuals. The enhanced CMT algorithm could track both a grocery item, such as a cereal box, and a person’s hand in order to guide them to the item and pick it up from the shelf. The green line in the image represents a distance and angle calculation between the two objects so that the user interface could give directions on where to move the hand.

This system can be easily created with the configurability of the descriptor matching algorithm. During the background subtraction step, all keypoints not in the two ROIs will be used as a background model and all keypoints will be matched to that model. Then, during object matching, the two separate object models can be run in parallel using separate matching engines. This will greatly reduce the time to perform object matching compared to software and be useful for many complex visual systems.

REFERENCES

[1] G. Nebehay and R. Pflugfelder, "Consensus-based matching and tracking of keypoints for object tracking," in *Applications of Computer Vision (WACV), 2014 IEEE Winter Conference*, 2014.

[2] H. Grabner and H. Bischof, "On-line boosting and vision," in *CVPR*, 2006.

[3] B. Babenko, M. Yang and S. Belongie, "Robust object tracking with online multiple instance tracking," in *TPAMI*, 2011.

[4] A. Saffari, C. Leistner, J. Santner, M. Godec and H. Bischof, "On-line random forests," in *ICCV Workshops*, 2009.

[5] G. Nebehay, *Robust Object Tracking Based on Tracking-Learning-Detection,* Vienna, 2012.

[6] Z. Kalal, K. Mikolajczyk and J. Matas, "Tracking-Learning-Detection," in *TPAMI*, 2012.

[7] J. Santner, C. Leistner, A. Saffari, T. Pock and H. Bischof, "PROST: Parallel robust online simple tracking," in *CVPR*, 2010.

[8] S. Leutenegger, M. Chili and R. Y. Siegwart, "BRISK: Binary robust invariant scalable keypoints," in *ICCV*, 2011.

[9] E. Rosten and T. Drummond, "Machine learning for high-speed corner detection," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2006.

[10] B. Lucas and T. Kanade, "An iterative image registration technique with an application in stereo vision," in *IJCAI*, 1981.

[11] Z. Kalal, K. Mikolajczyk and J. Matas, "Forward-backward error: automatic detection of tracking failures," in *ICPR*, 2010.

[12] H. Bay, A. Ess, T. Tuytelaars and L. Van Gool, "Speeded-up robust features (SURF)," *Computer vision and image understanding,* vol. 110, no. 3, pp. 346-359, 2008.

[13] D. Lowe, "Distinctive image features from scale-invariant keypoints," *IJCV,* vol. 60, no. 2, pp. 91-110, 2004.

[14] M. Kristan, L. Cehovin, T. Vojir and G. Negehay, "Visual Object Tracking Challenge (VOT2014) Evaluation Kit," 2014.

[15] L. Cehovin, M. Kristan and A. Leonardis, "Is my new tracker really better than yours?," in *IEEE Winter Conference on Applications of Computer Vision, WACV 2013*, 2014.

[16] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *CVPR*, 2001.

[17] M. Brown and D. Lowe, "Invariant features from interest point groups," in *BMVC*, 2002.