The Pennsylvania State University

The Graduate School

College of Engineering

UTILIZING GRAPHICS PROCESSING UNITS FOR RAPID FACIAL RECOGNITION USING VIDEO INPUT

A Thesis in

Computer Science and Engineering

by

Charles J. Gala

© 2014 Charles J. Gala

Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science

May 2014
The thesis of Charles J. Gala was reviewed and approved* by the following:

Raj Acharya  
Professor of Computer Science and Engineering  
Head of the Department of Computer Science and Engineering  
Thesis Advisor

Bruce T. Einfalt  
Research Engineer, Penn State Applied Research Laboratory  
Special Signatory

Robert Collins  
Professor of Computer Science and Engineering

Vijaykrishnan Narayanan  
Professor of Computer Science and Engineering

*Signatures are on file in the Graduate School.
Abstract

Facial recognition is an active research area that provides a real-time application of pattern recognition techniques. Input can be provided to recognition algorithms using both static images and video data. However, there are significant challenges to working with live streaming data as the recognition method needs to keep up with the frame rate of the video sequence. The main challenge is the speed at which frames from the video are processed. Most of the well-known pattern recognition techniques do not address the issue of processing speed, thus making them ill-suited for working with live video input. What is needed is a means to improve the processing speed of these video-based facial recognition techniques so that they can handle such input.

Graphics Processing Units (GPUs) are an ideal method to accelerate recognition processing. GPU architectures are powerful because they support a large number of cores that can process large amounts of data in parallel. Our goal is then to develop a novel technique for accelerated facial recognition using a GPU. The increased runtime performance allows more frames from a live video stream to be processed, reducing the likelihood of recognition and tracking errors.

We develop a facial recognition method to operate on the central processing unit (CPU), and then accelerate several components of the method to operate on the GPU. By implementing these components on the GPU, we achieve a method that runs up to six times faster than a pure CPU implementation of the method. We evaluate these implementations using live streaming data and find that the GPU implementation achieves a greater accuracy and performance over the CPU implementation.
# TABLE OF CONTENTS

List of Figures .......................................................................................................................... vi
List of Tables .............................................................................................................................. viii
1. INTRODUCTION .................................................................................................................... 1
2. BACKGROUND .......................................................................................................................... 5
   2.1 GPU Acceleration Methodology ....................................................................................... 6
      2.1.1 Kernel Operation ....................................................................................................... 6
      2.1.2 Memory Types .......................................................................................................... 7
      2.1.3 Parallel Reduction ..................................................................................................... 9
   2.2 Related Work ..................................................................................................................... 10
3. DETECTION ............................................................................................................................. 13
   3.1 Variance Filter ................................................................................................................... 14
   3.2 Ensemble Classifier ......................................................................................................... 18
   3.3 Nearest Neighbor Classifier ............................................................................................ 21
   3.4 GPU Acceleration ............................................................................................................ 24
      3.4.1 Variance Filter / Ensemble Classifier GPU Acceleration ......................................... 24
      3.4.2 Nearest-Neighbor Classifier GPU Acceleration ...................................................... 27
4. TRACKING ............................................................................................................................... 30
   4.1 Optical Flow Estimation ..................................................................................................... 30
   4.2 GPU Acceleration ............................................................................................................ 34
5. COMBINATION ......................................................................................................................... 36
   5.2 GPU Acceleration ............................................................................................................ 37
6. RECOGNITION .......................................................................................................................... 38
   6.1 Individual Face Modeling ................................................................................................. 38
   6.2 Assigning Pattern Models ............................................................................................... 40
   6.3 Pattern Model Implementation ......................................................................................... 42
   6.4 GPU Acceleration ............................................................................................................ 42
7. ENVIRONMENT AND EVALUATION ..................................................................................... 43
7.1 Hardware ........................................................................................................... 43
7.2 Software ............................................................................................................. 44
7.3 Evaluation Methodology ..................................................................................... 44
7.4 Datasets .............................................................................................................. 47

8. RESULTS ............................................................................................................. 49
  8.1 Individual Component Evaluations .................................................................. 49
    8.1.1 Integral Image Calculation Evaluations .................................................... 49
    8.1.2 Variance Filter / Ensemble Classifier Evaluations ................................... 51
    8.1.3 Nearest-Neighbor Classifier Evaluations ............................................... 52
    8.1.2 Lucas-Kanade Tracking Evaluations ....................................................... 53
  8.2 Overall Recognition Performance .................................................................... 55

9. CONCLUSIONS AND FUTURE WORK ....................................................... 61

APPENDIX A. Hardware Specifications ............................................................. 63
APPENDIX B. Development of the Pattern Models .......................................... 64
  B.1 Nearest-Neighbor Modeling ......................................................................... 64
  B.2 Conditional Random Field Modeling ............................................................. 65
  B.3 Support Vector Machine Modeling ................................................................. 69
  B.4 NN / CRF Modeling ......................................................................................... 71
  B.5 NN / SVM Modeling ....................................................................................... 71
  B.6 Evaluation ....................................................................................................... 72

Bibliography .......................................................................................................... 75
List of Figures

Figure 1: General Structure of the Recognition Method ................................................................. 2
Figure 2: CPU Processing vs GPU Processing [2] .............................................................................. 5
Figure 3: CUDA Memory Structure on GPU [4] ................................................................................ 8
Figure 4: Binary Tree Representation of Parallel Reduction [5] ......................................................... 10
Figure 5: Detection Cascade for Evaluating Windows ..................................................................... 14
Figure 6: Low Variance vs High Variance Windows (Image from [11]) ........................................... 15
Figure 7: Calculating the Sum of a Region using the Integral Image [12] ........................................ 17
Figure 8: Ensemble Classifier using Randomized Ferns [12] ............................................................ 19
Figure 9: Randomized Fern Classification ....................................................................................... 20
Figure 10: Classification of Window Patch using Nearest Neighbor Examples [1] ......................... 23
Figure 11: Calculating Integral Image in Two Operations ............................................................... 25
Figure 12: Evaluating Nearest Neighbor Examples for All Windows in Parallel ......................... 27
Figure 13: Treating Multiple Parallel Reductions as One Large Reduction [5] ............................... 29
Figure 14: Estimation of Pattern Motion by Tracking Individual Data Points [1] ......................... 30
Figure 15: Image Pyramid Generated for Lucas-Kanade Tracking (Image from [11]) ............... 32
Figure 16: Forward-Backward Error [1] ......................................................................................... 33
Figure 17: Overlapping Windows Selecting the Pattern Instance [11] ........................................... 36
Figure 18: Evaluating the Image Patch against Each Pattern Model (Image from [17]) ........... 39
Figure 19: Applying the Gale-Shapley Algorithm to Label Image Patches .................................... 41
Figure 20: Intel Core 2 Duo Processor E6600 and Tesla C2050 ..................................................... 43
Figure 21: Calculation of the Overlap between Two Bounding Boxes [22] ................................. 45
Figure 22: Scenes from Hannah and Her Sisters Selected for Evaluation [11] ............................... 47
Figure 23: Sample Frames from SPEVI Dataset [17] .............................................................. 48
Figure 24: Processing Time Integral Image Calculations: CPU vs GPU ................................. 50
Figure 25: Processing Time of Nearest-Neighbor Classifier: CPU vs GPU ............................ 52
Figure 26: Processing Time of Lucas-Kanade Tracking: CPU vs GPU ................................... 54
Figure 27: Example Screenshots of Recognition Method Operation ......................................... 55
Figure 28: State Diagram for CRF Transitions ........................................................................ 66
Figure 29: Using Synthetic Negative Training for the CRF Pattern Model (Images from [30]) .. 68
Figure 30: Using Support Vector Machines to Separate Groups of Data [31] ............................ 69
List of Tables

Table 1: Processing Time of Variance Filter and Ensemble Classifier: CPU vs GPU..................51

Table 2: Recognition Method Performance (Offline Mode): CPU vs GPU..........................56

Table 3: Recognition Method Performance (Live Streaming Mode): CPU vs GPU.................58

Table 4: Percentage Loss When Running Recognition Method in Live Streaming Mode........59

Table 5: CPU Specifications.................................................................................................63

Table 6: GPU Specifications.................................................................................................63

Table 7: Pattern Model Evaluations using Hannah Sequences............................................73
1. INTRODUCTION

Facial recognition is an active research area that provides a real-time application of pattern recognition techniques. Research in this field has provided many advances in fields such as biometrics and security applications. Input can be provided to recognition algorithms using both static images and video data. The advantage of using video input over static images is the fact that additional spatio-temporal information is provided which can be used to improve the performance of the recognition technique, such as easing the challenge of detecting/tracking faces and refining models of a face being identified.

However, there are significant challenges to fully utilizing this spatio-temporal information when working with live streaming data. The main challenge is the speed at which frames from the video are processed. During the time that an algorithm is processing a single frame, additional frames can become available as the next frame and then be replaced. While it is not necessary to try and utilize every single frame in the video sequence, it is good to use as many frames as possible since any of the frames can contain useful information for the algorithm. Most of the well-known pattern recognition techniques do not address the issue of processing speed, thus making them ill-suited for working with live video input. What is needed is a means to improve the processing speed of these video-based facial recognition techniques so that they can handle live streaming data.

Graphics Processing Units are an ideal way to accelerate recognition processing. GPU architectures are powerful because they support a large number of cores that can process large amounts of data in parallel. Our goal is then to develop a novel technique for implementing facial recognition using a GPU. For the best results, the amount of memory transfers between
CPU and GPU should be minimized, and the amount of work performed on the GPU should be maximized. Ideally, a GPU implementation should significantly improve both the processing speed and performance of the recognition. The increased runtime performance allows more frames from a live video stream to be processed, reducing the likelihood of recognition and tracking errors.

In this thesis we develop a recognition method to detect faces that appear in the video sequence, and then identify these faces to distinguish them from the other faces detected. Following this we select specific components of the method that would benefit from GPU parallelization and accelerate them on the GPU. Figure 1 presents the general structure of the recognition method developed for this thesis.

![General Structure of the Recognition Method](image)

**Figure 1: General Structure of the Recognition Method**

Our recognition method is broken up into several stages, where each stage performs a different task. The detection stage looks for new instances of some target pattern that has be modeled,
such as a face, in a particular frame of the video sequence. The tracking stage, on the other hand, takes previous known instances of the target pattern and tries to predict the next instances. Both the detection and tracking stages provide bounding boxes that describe the location of pattern instances as output. The combination stage then combines those bounding boxes that describe the same pattern instance. Finally, the recognition stage examines the patterns described by the bounding boxes found and labels them. In terms of facial recognition this means that the stage identifies the different faces that are found. For each frame in the video sequence, all of these stages are run in a large main loop. The loop ends when we reach the end of the video sequence. Note that the tracking stage takes the results of the previous run of the combination stage as input. This allows it to estimate the movement of the bounding boxes in the next frame of the video sequence.

We base the detection and tracking stages of our recognition method on the TLD (Tracking-Learning-Detection) method introduced by Zdenek Kalal et al. [1]. The algorithm was developed to handle issues that are encountered with long term-tracking algorithms, and provide a means for on-the-fly training of classifiers instead of learning an object using offline learning. It is designed to search a scene to detect instances of a pattern modeled by a series of classifiers in each frame of a video sequence. Detected patterns are then tracked for future frames using the Lucas-Kanade tracking method. In terms of the recognition stage of our method, we develop a pattern model for learning a particular face on-the-fly using a minimal amount of initialization data. The recognition stage then finds the best arrangement of pattern models to identify the faces currently detected by the recognition method.

The format for the remaining sections of this thesis is as follows. We introduce GPU computing and related works in Section 2. Sections 3 and 4 introduce the detection and tracking
stages, respectively, used in our recognition method. The combination of results from the two stages is described in Section 5. The recognition stage of the method is discussed in Section 6, and the testing environment and methodology for the recognition method are described in Section 7. The results of the evaluations are presented in Section 8. Concluding remarks and future possible work on the recognition method are then provided in Section 9.
2. BACKGROUND

GPUs are specialized multicore hardware used to perform the heavy workload involved with computer graphics rendering. Thousands of mathematical calculations are necessary for the rendering process, and GPUs are designed to handle this using a powerful parallel processing framework to perform these operations. Because GPUs are specifically designed to perform a large number of computations in parallel across many cores (as shown in Figure 2), GPUs outperform Central Processing Units (CPUs) in parallel processing power. By pushing the rendering computations onto the GPU this not only speeds up the processing time but also frees up processing power for the CPU.

![Figure 2: CPU Processing vs GPU Processing](image)

In recent years it has become possible to apply GPUs to implement parallel processing algorithms using the same framework used for graphics rendering. CUDA [2] (Computer Unified Device Architecture) is one such architecture used to delegate work on the GPU from the CPU. By pushing parallelizable work onto the GPU, we achieve additional processing power and reduce the runtime of the overall program. The architecture is supported in several programming languages; for our purposes we used C++.

GPUs operate for graphics rendering by creating thousands of threads that run in parallel which all perform the same operation on a different pixel. CUDA extends this methodology by
declaring a kernel function that is called by each of the generated threads. Each thread is given a
different index value to distinguish itself from other threads; this allows each thread to perform
the same operation on a different segment memory or perform slightly different behaviors.
Threads are grouped into a grid of thread blocks that can be used to cooperate to perform various
functions. In addition there are some simple barrier constructs provided to help with thread
synchronization.

2.1 GPU Acceleration Methodology

There are several strategies for accelerating code on the GPU, although some methods are
more efficient than others. In this section we will discuss some of the approaches that are used
to fully take advantage of the GPU.

2.1.1 Kernel Operation

In order to push work onto the GPU, memory must be made available on the device for it
to work on. A memory transfer operation must be performed to copy data over to the GPU.
Similarly, after the GPU has completed doing its work the results must be copied over to the
CPU so that it can access them. The overhead to perform this memory transfer between the two
devices is significant enough that it can slow down the processing time of a program. Therefore,
it is desired to minimize the number of the memory transfers between the CPU and the GPU. If
possible, we prefer to recalculate sections of memory rather than transfer them over constantly if
it does not require much processing time. It is also good to avoid multiple memory transfers in
favor of a single memory transfer to cut down on this overhead.

As stated previously, when CUDA is used to generate a set of threads to run, each thread
is associated with a kernel function. Ideally a kernel function should involve fast and
processing-light operations as each thread will be performing the same operations. Over
thousands of threads, heavy operations can create a bottleneck and significantly slow down the processing time of the threads. In other words, try to avoid operations such as multiple for-loops and other heavy computations when writing kernel functions. When a kernel function is used to create a set of threads, the number and arrangement of threads can be specified. We try to take advantage of this by arranging the threads in the best possible way to minimize the necessary number of threads generated.

2.1.2 Memory Types

There are several different GPU memory sources for threads on the GPU. Normally data is uploaded to global memory for each kernel function to have access to. However, this is the slowest of the memory types available since it tends to have long access latencies and limited access bandwidth [3]. There exist alternatives that can provide a faster read time than just using global memory depending on how the memory is used.
Figure 3: CUDA Memory Structure on GPU [4]

Figure 3 shows how the different types of memory are represented on a GPU. If it is not necessary for the data to be modified by the GPU threads, then as an alternative constant memory can be used to store data for the threads. Because this memory is read-only, this means faster access times and more parallel access opportunities for thread kernels than with global memory [3]. Technically constant memory is stored in the same space as global memory but it is cached for efficient access.

If data is going to be used solely for coordination between threads in a particular block, then we can go a step further with shared memory which has even faster access times. Shared memory can be both read and written by threads in the same block. The memory is only available for the lifetime of the threads. It is ideal for data that is heavily accessed during the
execution phase of the kernel function. Unfortunately a tradeoff for the fast access times of shared memory is the size of the memory, which is significantly smaller than that of global memory. That being said, a common strategy to resolve this problem is to partition the data into subsets called *tiles* so that each segment fits into the shared memory for a particular block. This assumes that all the necessary data for kernel execution of a particular block can be fit into a small enough space.

There is also the option of using *texture memory* for storing data. Texture memory is a variety of global memory that can improve access times and reduce memory traffic when memory reads have certain access patterns [4]. The memory works with specially designed texture caches used to access a large amount of data that have significant spatial locality. In other words, this means that texture memory is ideal for data access with addresses that are “near” each other. If kernel functions are generally reading from the same region in memory, then it would be useful to apply texture memory. Normally texture memory is read-only, but on some GPUs there is support for *surface memory*, which allow for writes to occur directly on texture memory [2].

### 2.1.3 Parallel Reduction

Once of the most common applications of GPU computing is the *parallel reduction* method [5]. The technique is mentioned several times in the research, so it is worth describing here. Parallel reduction is a method used to apply the same grouping operation between all elements in very large arrays, such as summation or finding the maximum value. The difficulty in performing this work in parallel on a GPU is that partial results need to be communicated between threads. The solution is to apply a tree-based approach within each thread block (see Figure 4).
The grouping operation is decomposed into many sub problems, where each thread solves a different sub problem. At first, $N/2$ problems are created, where $N$ is the number of elements in the large array. Each problem solves for a segment of the array. Once these problems are solved, a new set of sub problems are created using solely the previous results, reducing the number of problems in half. This process continues until there is one thread left, which calculates the final solution. In order to perform this method, it is necessary to coordinate threads since the next set of sub problems cannot be started until the previous set has completed. This is done by utilizing the barrier constructs used for synchronization in CUDA. The large number of synchronizations necessary to perform this work would normally be a poor design choice, but because each grouping operation before the next synchronization is relatively quick this is hardly an issue. Shared memory in particular is useful here as it can be used for inter-block communication.

2.2 Related Work

There has been some research in terms of implementing video-based facial recognition using GPUs. For the detection stage of recognition, there has been some work performed using GPU to cut down on the amount of work involved for locating faces. Oro et al. [6] presented a
Haar-based detector that exploits both coarse and fine grain parallelism with the GPU. The implementation was developed with the intent to handle HD video sequences, which contain large amounts of data to process. Scans are performed across blocks of threads in the GPU, and images are resized rather than having to resize the filters, which would be slower. Hefenbrock et al. [7] presented a GPU-based implementation of the Viola-Jones face detector that allows for multiple GPUs to be used. Both feature evaluation and window scanning are performed in parallel using the GPU threads.

In terms of actual facial recognition, in [8] the authors develop an implementation of the Neocognitron Neural Network using a GPU. Each neuron in the network is represented as a GPU thread, and groups of neurons (cell plans) are handled as thread blocks. While this is an interesting approach to utilizing GPUs, the algorithm was not designed to take advantage of video input. Ouerhani et al. [9] proposed a facial recognition technique that utilizes a composite filter based on correlation. A dataset of 100 persons is referenced during the recognition, and the method must pick from one of their faces to perform recognition. This study does in fact study real-time data in a video sequence, but its scope is somewhat limited. Only one video sequence is used in evaluations, which has a very short runtime (6 seconds) and small frame size (288 x 352 pixels). In addition, the method does perform any face detection or tracking; the location of the face is known to be in the center of the video sequence.

The closest research related to video-based facial recognition that could be found was a feature tracking and matching algorithm presented by Sinha at al. [10]. The algorithm implements a KLT tracker on a GPU, where the tracker is focused on collecting the eigenvalues from a pattern and predicting where these values will appear next. They also provide a SIFT feature extraction algorithm implementation for GPUs. While this is similar to what is being
proposed, the authors focus primarily on the speed of the algorithms, and provide minimal focus on the actual tracking and detection performance of the techniques. In addition, the KLT tracker and SIFT extraction methods are different from the implementation being proposed in this thesis.
3. DETECTION

The purpose of the detection stage is to find new instances of some target pattern being modeled (e.g. a face). Finding such pattern instances are also important in determining the bounding boxes for the tracking stage to follow for the subsequent frame. To perform detection the TLD method takes advantage of a sliding window approach which scans a particular image frame using windows of various sizes [1]. The number of windows searched depends on the initialization of the model and dimensions of the input video sequence, and can range from 10,000 to 250,000 windows. No prior assumptions are used to limit the number of windows as such assumptions can limit the accuracy of the detection method. Each window is evaluated independently of other window evaluations.

Due to this particularly large number of windows to evaluate, the detection method takes advantage of a cascade of classifiers to identify instances of the target pattern. The detection cascade is comprised of several different classifiers arranged from weak and fast to strong and slow. The weaker classifiers are used to cut down on the number of windows that need to be evaluated by the later classifiers by eliminating windows that are obviously not the target pattern. By having the stronger and slower classifiers evaluate fewer windows, we effectively reduce the processing time to evaluate every window (see Figure 5).
After the cascade has performed its run, any windows that haven’t been rejected are used to determine the output for the detection stage. Because it is possible to have multiple windows around the same pattern, a clustering technique is used to combine any overlapping windows. It is generally assumed that windows with significant overlap are describing the same pattern. The windows calculated using the clustering technique is the final result of the detection stage.

The TLD method uses three different classifiers for the detection cascade. These classifiers are a variance filter, randomized forest/ensemble classifier, and a nearest-neighbor classifier [1]. These classifiers will be described in the following sections.

### 3.1 Variance Filter

The variance filter is the initial stage of the detection cascade which is used to eliminate windows which have very low variance. For our purposes, variance is used to describe the range of pixel values that appear in a particular window. If the variance is too low, then we can expect
there to be very few details in the image bounded by the window as there is little change in the pixel values for the image. Figure 6 demonstrates this idea, showing examples of windows with high/low variance.

![Figure 6: Low Variance vs High Variance Windows (Image from [11])](image)

We assume that windows with low variance are considered unreliable; having low variance suggests that few meaningful features can be extracted from these windows. Whether or not a window has low variance is dependent on if the window’s variance is smaller than a set threshold value selected at the detection cascade initialization.

If we treat the image bounded by a particular window as a one-dimensional array \( x \) of length \( n \), we can define variance using Equation 1 [12].

\[
\sigma^2 = \frac{1}{n} \sum_{i=1}^{n} (x_i - \mu)^2
\]  

Here \( \sigma^2 \) is the variance value and \( \mu \) is defined via Equation 2 [12].
\[
\mu = \frac{1}{n} \sum_{i=1}^{n} x_i
\]

Alternatively Equation 1 can be given as:

\[
\sigma^2 = \frac{1}{n} \sum_{i=1}^{n} x_i^2 - \mu^2
\]

This version of Equation 1 will be useful later on, and its derivation can be found in [12]. Based on the given equations, it is necessary for \( n \) memory lookups for each window to be evaluated in the detection cascade. Considering that this same operation has to be performed for the thousands of windows that need to be evaluated, this many memory lookups is not sufficient for our purposes.

To quickly and efficiently calculate each window’s variance, we take advantage of the integral image for the current frame image. Integral images were introduced by Viola and Jones [13] as a means to calculate the sum of pixel values in a region of the overall image. Once the integral image is calculated, the sum of pixel of values can be calculated in four memory lookups (constant time), regardless of the size of the region. The calculated integral image is the same size as the original image, where each pixel value in the integral image is calculated using Equation 4 [12].

\[
I'(x, y) = I(x, y) + I'(x - 1, y) + I'(x, y - 1) + I'(x - 1, y - 1)
\]

Here \( I \) is the original image and \( I' \) is the integral image. In the case that \( x = 0 \) or \( y = 0 \) we use \( I'(x, y) = 0 \). The values for the integral image ultimately propagate the sum of the original image from the top left of the image to the bottom right. By doing this we can find the sum of a particular region in the image using the four corners of the region. See Figure 7 and Equation 5.
Figure 7: Calculating the Sum of a Region using the Integral Image [12]

In Equation 5, \( x \) and \( y \) describe the top left corner of the selected region (A). The values \( h \) and \( w \) are the height and width of the selected region. Figure 7 shows that by finding these four values (A, B, C, D) we are able to find the sum of the region. This is the same as summing up all the pixel values to D from (0,0) of the original image, subtracting the pixel values up through B and C, and adding the pixel values up to A. Therefore, finding the sum of the region is no longer dependent on the number of pixels in the region and is now a very fast operation.

In order to use the idea of integral images to find the variance of a particular window, we actually need to maintain two integral images per image frame. Based on Equation 3, one integral image is used to find the first term of the equation and another integral image is used for the second term. For the second term \( \mu \) the following notation is used, where \( w \) is the window being evaluated.

\[
\sum_{i=1}^{n} x_i = I'(w)
\]
To find the first term, the formula for finding the corresponding integral image must be slightly modified to account for the fact that the squared value of $x_i$ is used inside the summation.

$$I''(x, y) = \sum_{x' \leq x, y' \leq y} I(x', y')^2$$  \hspace{1cm} (7)

Using the same notation as Equation 6 we get:

$$\sum_{i=1}^{n} x_i^2 = I''(w)$$  \hspace{1cm} (8)

Finally by combining Equations 3, 6, and 8 we end with the following equation.

$$\sigma^2 = \frac{1}{n} I''(w) - \left[ \frac{1}{n} I'(w) \right]^2$$  \hspace{1cm} (9)

If it takes four memory lookups to find the value of each integral image, then we are able to reduce the work to calculate the variance of a window to eight memory lookups and a simple arithmetic operation.

By taking advantage of the idea of integral images, we are able to make the variance filter run very fast for each window to be evaluated. The variance filter has been found to be a very powerful tool for the cascade, not only because of its speed but also by the number of windows it eliminates. By eliminating those windows with low variance, this component of the detection cascade is typically is able to reduce the number of windows to be evaluated by at least 50 percent. This greatly improves the processing time of the overall cascade.

### 3.2 Ensemble Classifier

The second stage of the detection cascade is an ensemble classifier composed of several randomized ferns. This technique is commonly referred to as a randomized forest. The classifier
takes the windows that were not rejected by the variance filter and runs each window against each of the ferns. Each fern assigns a confidence value to the window. If the average of the confidence values is greater than 50 percent then the window is accepted. This arrangement of classifiers is presented below in Figure 8 where three different ferns are used.

![Figure 8: Ensemble Classifier using Randomized Ferns [12]](image)

Each fern in the ensemble classifier evaluates over a set of 2-bit binary features. Each feature is a comparison between two pixel values performed on the region of the image described by the window. If the first pixel value is less than the second pixel value, then the feature value is 1; otherwise the feature value is 0. Equation 10 presents these features used for the ensemble classifier.
The locations of the pixel values compared are randomly generated using a random distribution when the ensemble classifier is initialized. Because we are comparing two values rather than checking for a particular value, the features are invariant against constant brightness variations. Once all of the features are calculated for a particular fern, their values are combined to create a binary number. This binary number is a unique index value that describes the arrangement of features for the given window. This index value is then used to look up the particular confidence score assigned for the arrangement of features (see Figure 9).

\[
   f_i(l) = \begin{cases} 
   1 & \text{if } I(p_1) < I(p_2) \\
   0 & \text{otherwise} 
\end{cases}
\]

Figure 9: Randomized Fern Classification
While this classifier has a longer runtime than with the variance filter, it is still a relatively fast classifier. Only two memory lookups and one comparison operation is necessary for each feature, and a small number of operations are needed to find confidence scores and combine them. For our purposes, we use 13 randomized ferns for the ensemble classifier and 10 features for each fern.

3.3 Nearest Neighbor Classifier

The final stage of the detection cascade is a nearest-neighbor classifier that behaves as a template-matching technique. Decisions are based on a set of normalized image patches stored in memory. Both image patches representing the target pattern and not representing the target pattern are stored and used in the classification process. Out of all the classifiers in the detection cascade this classifier is the strongest and most rigorous of the three classifiers; however it is also the slowest of the classifiers.

The nearest-neighbor classifier stores a set of image patches which act as positive and negative examples for the target pattern. In terms of this research, positive examples are those image patches that describe the target pattern, and the negative examples are those image patches that do not. Using these examples, a confidence values is calculated which determines whether or not the classifier accepts the window or not. The confidence is calculated based on pixel-by-pixel calculations using the Normalized Correlation Coefficient (NCC) between two image patches. The similarity is calculated for two image patches $P_1$ and $P_2$ using Equation 11 [12].

$$NCC(P_1, P_2) = \frac{1}{n - 1} \sum_{x=1}^{n} \frac{(P_1(x) - \mu_1)(P_2(x) - \mu_2)}{\sigma_1 \sigma_2} \quad (11)$$

The values $\mu_1$, $\mu_2$, $\sigma_1$ and $\sigma_2$ represent the mean and standard deviations of the two patches, respectively. The more similar the patches are the closer the NCC score is to 1, and the
calculations are invariant against uniform brightness variations. Since the operation requires the image patches to be the same size, we use a fixed patch size (15 x 15) for the examples stored in memory. In terms of the window images, we extract the images bounded by the windows and resize them to the same dimensions.

For each example in the classifier’s memory we find the NCC between the window patch and the example patch. Because the range of values for NCC is from -1 to 1, the resulting value is converted to a distance measurement using Equation 12 [12].

\[ d(P_1, P_2) = 1 - \frac{1}{2} (NCC(P_1, P_2) + 1) \]  

Equation 12

By converting to a distance measurement we acquire a value within the range of 1 to 0. The closer the value is to 0 the more similar the example patch is to the window patch. The minimum distance values for the positive examples and the negative examples are then found, which are then used to calculate the confidence value (Equations 13 – 15 [12]).

\[ d^+ = \min_{P_i \in P^+} d(P_0, P_i) \]  
\[ d^- = \min_{P_j \in P^-} d(P_0, P_j) \]

Equations 13 and 14

\[ \text{confidence} = \frac{d^-}{d^- + d^+} \]  

Equation 15

In the equations given, the set of positive examples is \( P^+ \) and the set of negative examples is \( P^- \). The distance values for the most similar positive and negative examples are given as \( d^+ \) and \( d^- \), respectively.

In simpler terms, the confidence given to a particular window patch is dependent on the closest positive and negatives patches to the window patch. If there is a positive example that
matches the window patch exactly, then the confidence for that patch is 1.0. This confidence value will decrease as the as distance to the closest positive example increases and the distance to the closest negative example decreases. Figure 10 gives an example of one such arrangement of positive and negative examples to calculate the confidence value.

![Figure 10: Classification of Window Patch using Nearest Neighbor Examples [1]](image)

If the confidence value for the window is above a certain threshold, then the patch is accepted. For our purposes we use a threshold value equal to .65.

Typically this classifier only evaluates 20 to 50 of the 10,000 to 250,000 windows that are reviewed by the detection cascade; the other windows in the set have already been rejected by the previous two components in the cascade. It is important to keep the number of windows evaluated by the nearest-neighbor classifier small. As stated previously, this classifier is the slowest of the classifiers in the detection cascade. This because a large number of calculations are necessary to find the norm-cross correlation of every image patch stored in memory. In addition, the amount of processing involved with the classifier is dependent on the number of image patches in memory, which can be very large. Therefore, it is necessary that the previous
components of the detection cascade reduce the number of windows that reach the nearest-neighbor classifier as much as possible.

3.4 GPU Acceleration

To accelerate the detection stage of the recognition method on the GPU, we divide the detection cascade into two separate segments. The first segment is used to evaluate each window over the variance filter and ensemble classifier components of the cascade. The second segment then evaluates those windows that passed the first segment over the nearest-neighbor classifier.

3.4.1 Variance Filter / Ensemble Classifier GPU Acceleration

Since each window evaluation in the detection cascade is independent of the other window evaluations, it is logical to perform these evaluations in parallel, with each window evaluation handled by a separate GPU thread. Since the amount of work per window is relatively small for the ensemble classifier and variance filter, this makes them ideal to run on a GPU kernel. This is how we evaluate the first two classifiers in the detection cascade.

In order to run these classifiers on the GPU though, it is necessary to copy the necessary information to run the cascade onto the GPU. This includes the window locations, ensemble classifier confidence scores, and the current image frame. Most of the information only needs to be moved once when the detection stage of the recognition method is initialized. However, some pieces of memory need to be updated every time that the detection cascade is run. These elements are the current image frame and the integral images for the variance filter since they change with every frame processed.
We take advantage of the GPU when calculating the integral images to significantly reduce the processing time involved in finding them. Previously it was shown in Equation 4 (shown below) that to find the value of a particular pixel for the integral image, it is necessary to have three other integral image values calculated already.

\[
I'(x, y) = I(x, y) + I'(x - 1, y) + I'(x, y - 1) + I'(x - 1, y - 1)
\]  (4)

This makes it difficult parallelize the operation, based on Equation 4. However, an alternative means for calculating the integral image was introduced by Bilgic et al. in [14]. Rather than propagating the sum of the image from the top left to the bottom right of the image in one single step, the integral image is calculated in two steps. For the first step we propagate the sum over each row of the image from left to right (Equation 16). We then take the results of that step (\(I^{temp}\)) and propagate the sum over each column from top to bottom (Equation 17).

\[
I^{temp}(x, y) = I(x, y) + I^{temp}(x - 1, y)
\]  (16)

\[
I'(x, y) = I^{temp}(x, y) + I'(x, y - 1)
\]  (17)

Same as before, we use \(I^{temp}(x, y) = 0\) for \(x = 0\) in Equation 16 and \(I'(x, y) = 0\) for \(y = 0\) in Equation 17. The result of this method is the same as if we were to use Equation 4 to find the image, as depicted in Figure 11.

\[\sum \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} = \sum \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} + \sum \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} \]

Figure 11: Calculating Integral Image in Two Operations
This new method for finding the integral image is something that can be parallelized on the GPU. Two kernel calls are used to perform the calculations; the first kernel performs the summations over each row and the second kernel performs the summations over each column. For the first kernel call, \textit{height} many threads are spawned, and \textit{width} many threads are spawned for the second kernel call (where \textit{height} and \textit{width} refer to the height and width of the image frame).

With all of the necessary information available on the GPU, we can then run the first two classifiers on the detection cascade on the GPU. A number of threads are created, where each thread handles a different window to be evaluated by the classifiers. Each thread first evaluates their respective window on the variance filter; if the window is accepted, the thread then evaluates the window using the ensemble classifier. If the window passes this classifier, then the window is marked as ‘passed’ using an array of results stored on the GPU that keeps track of each window. If the window fails at either of the two classifiers the window is marked as ‘failed’. This result array is copied back to the CPU, and the windows that have been marked as ‘passed’ are presented to the second segment of the detection cascade.

For this segment of the detection cascade we do not take advantage of any of the faster types of memory on the GPU. Because the first two classifiers try to minimize the number of memory references used, runtime is minimal, and any benefits to mapping the information to texture memory would be outweighed by the overhead involved. In addition the information used to run the classifiers is too large to place in constant memory. Thankfully though there are several opportunities to map data to these memories for the second segment of the detection cascade.
3.4.2 Nearest-Neighbor Classifier GPU Acceleration

Unlike the previous two classifiers, the nearest neighbor classifier is not well suited to entirely run on a GPU kernel. The main issues lie in the amount of work performed for each window evaluation. For each window evaluation hundreds of examples must be compared using NCC. If the nearest-neighbor classifier were to be implemented on the GPU in the same fashion as the previous two classifiers, it would only serve to increase the processing time of the method rather than decrease the time.

Therefore, it is necessary to approach the GPU acceleration of the nearest-neighbor classifier from a different direction. Rather than implementing the classifier to perform window evaluations in parallel, it is much more efficient to compare examples in parallel. This methodology can also be extended to allow us to compare examples for all windows to be evaluated, rather than just for one specific window. By performing all of the example comparisons in parallel at once, this reduces the amount of overhead required to perform the comparisons. Figure 12 better depicts this idea, where each blue block is a separate example evaluation. It is only necessary to make one kernel function call as opposed to having to make multiple calls, a number which depends on the number of windows to be evaluated by nearest neighbor classifier.

Figure 12: Evaluating Nearest Neighbor Examples for All Windows in Parallel
An additional bonus of running the classifier in parallel based on the number of examples is that we can ensure that the processing time per window is relatively the same regardless of the number of examples stored in memory.

Although we are able to evaluate both positive and negative example comparisons in parallel using the above method, there is still the question of finding the minimum positive and negative distances to calculate the confidence score (Equation 15). After we perform the example evaluations we are left with an array of length $n \times m$ on the GPU that contains all of the distance results, where $n$ is the number of examples compared per window and $m$ is the number of windows to be evaluated. There are a couple different ways to find these minimum values. One option is to copy all $n \times m$ distance values back onto the CPU and iterate through all of the distances find the minimum values for each window. Alternatively we can iterate directly on the GPU to find the minimum value, and then need only to copy back these $m$ values back to the CPU to calculate the confidence scores.

We utilize parallel reduction to quickly find these minimum values on the GPU. For each sub problem in the reduction, we apply a minimum operation that returns the smaller of the two inputs. We take this a step further since for each window we have the same number of distances to search over to find the minimum (since each window is compared to the same set of examples). Rather than perform a separate parallel reduction for each window, we perform all of the reductions at the same time and treat them as one large reduction. Figure 13 describes this process. Similar to when finding one minimum value, a many threads are generated to evaluate many sub problems. Once these sub problems are answered, we solve the next set of problems using the results of the first set.
The only difference is then that we stop the reduction process when we have the minimum value for each window, rather than waiting until there is only one thread left.

To accelerate the processing time for the nearest neighbor classifier, we take advantage of some of the different types of memory on the GPU. In order to evaluate a window against each of the stored examples, it is necessary to extract a normalized image patch for each window. We store the image frame in texture memory for this purpose. Texture memory is useful here since every pixel value in a particular window must be accessed to extract an image patch, and some pixel values may be accessed multiple times depending on the arrangement of windows to evaluate.

Constant memory is also used in our implementation of the nearest neighbor classifier. Since the number of windows to evaluate using the classifier is typically small, we can store the extracted patches in constant memory. The maximum number of image patches that can be stored in constant memory is around 70, based on the size of a single image patch stored in memory. If the number of windows to be evaluated is larger than this amount, then we are forced to divide the windows to be evaluated into separate groups of size ≤ 70. Additional kernels calls are then necessary to calculate all of the similarity distances. This is hardly ever necessary though as most runs of the classifier work with a number of windows that was much smaller than 70.
4. TRACKING

Object tracking is important to the recognition method because it is unreasonable to assume that the method will be able properly detect pattern instances in every frame of a live streaming video sequence. We can ease the burden of the detection stage by taking advantage of pattern instances found in the previous frame to estimate where they will appear in the current frame. The goal of the tracking stage is to estimate the new locations of these previously found patterns using the previous pattern location in addition to the previous and current image frame. This is done by generating a set of data points at the previously known location, then estimating their motion. The new locations of the data points are then used to find the new location of the pattern, as depicted in Figure 14.

![Figure 14: Estimation of Pattern Motion by Tracking Individual Data Points [1]](image)

4.1 Optical Flow Estimation

To predict the movement of a particular pattern instance the TLD method [1] applies the popular *Lucas-Kanade tracking technique* (LK) for optical flow estimation. The LK method operates under two assumptions. The first assumption is that the movement of a particular data point between frames should be expected to be within a local neighborhood around the original
location of the data point. The second assumption is the fact that the appearance of the pixel should not be expected to change much between two consecutive frames. We can further ensure this second assumption by using normalized images calculated from the original image frames; by doing so we negate the effect of lighting in the video sequence. Based on these assumptions we search the surrounding neighborhood and estimate the optical flow by minimizing the difference in pixel appearance between the original location of data points in the previous image and the estimated location in the current image. The difference between pixel values is found using the least squares method. In order to minimize this difference the technique needs to iterate several times to gradually improve the trajectory estimation.

For the recognition method we specifically the pyramidal implementation [15] of Lucas-Kanade is utilized for tracking. This implementation improves on the original algorithm by improving the accuracy of the technique while still maintaining its robustness. It was found that by searching a smaller surrounding neighborhood of a data point that the accuracy generally improves; however this makes it more difficult for the method to handle larger optical flows. The pyramidal implementation works around this issue by initially running with a scaled-down versions of the before and after images. This allows the method to get a rough estimate of the flow while still keeping the search neighborhood small. Following this the method is run again using a slightly larger scale and same window size to refine the previous estimate of the flow. This continues until we reach the original scale of the image. By this point we have a very accurate estimate of the optical flow of a data point.
Typically the scaled images of the previous and current image frames are calculated prior to estimating the optical flow of the data points. This collection of images is referred to as the *image pyramid*, where each level of the pyramid refers to a different scale for the two image frames (see Figure 15). The highest level of the image pyramid is the smallest scale and the lowest level is the original scale of the image frame. For this research we use a pyramid with 6 different levels.

Using the above technique, 400+ data points are generated equally spaced within the bounding box found for the previous pattern instance location. We found that this number of data points achieved the most accurate tracking estimate. Each of the generated data points are tracked using Lucas-Kanade and the resulting estimates are used to generate a new bounding box. Only estimates that are considered reliable are used in constructing the new bounding box. To determine the reliability of the optical flow estimates two error measures are used. The first measure is the *Forward-Backward error* estimate as introduced by Kalal et al. in [1]. The idea behind this measure is the fact that an accurate optical flow estimation can be used to predict the...
movement of a data point back to its original location from its estimated destination. In the figure shown below (Figure 16), data point 1 is shown to have an excellent optical flow estimate while data point 2’s optical flow estimation has some significant error.

![Figure 16: Forward-Backward Error](image)

In order to find the Forward-Backward error, it is necessary to run LK a second time using the estimated optical flow as input and swapping the previous and current image frames. The error is then determined by the Euclidean distance between the original data point locations and the locations found from running LK a second time. The closer the error value is to zero the more accurate the estimation is.

The second measure is found using the Normalized Correlation Coefficient (NCC) equation given in Section 3.3. For the sake of convenience, the equation is shown again below.

\[
NCC(P_1, P_2) = \frac{1}{n-1} \sum_{x=1}^{n} \frac{(P_1(x) - \mu_1)(P_2(x) - \mu_2)}{\sigma_1 \sigma_2}
\]

The values \(\mu_1, \mu_2, \sigma_1, \) and \(\sigma_2\) represent the mean and standard deviations of the two patches, respectively. For each data point tracked, we extract small image patches \(P_1\) and \(P_2\) extracted around the data point before and after movement. This is meant to check and see if the pixel values surrounding the data point match up before and after motion. The equation returns a
value within the range of -1 to 1, where 1 means that the two patches are identical. We assume that the tracking results for that data point are reliable in terms of NCC if the two patches are similar.

To determine the reliability of each data point’s optical flow estimation, the median of all Forward-Backward errors and NCC calculations are found ($med_{FB}$ and $med_{NCC}$ respectively). A particular estimation is then considered reliable if the forward-backward error is less than $med_{FB}$ and greater than $med_{NCC}$. Only those points that are considered reliable are used in calculating the new bounding box, which is the final result of the tracking stage.

**4.2 GPU Acceleration**

To accelerate the tracking stage of the TLD method on the GPU, we develop a GPU-implementation that evaluates each data point to be tracked on a separate GPU thread. Both the Lucas-Kanade tracking and the error calculations for a particular data point are independent of the calculations involved for the other data points. We can parallelize this work on the GPU and expect to see a significant increase in the processing time of the recognition method.

In terms of the Lucas-Kanade pyramidal method, the implementation we use is based on a pre-existing implementation of Lucas Kanade developed by Nghia Ho in [16]. That being said, the original method was heavily modified to resemble the OpenCV implementation of Pyramidal LK, as well as to improve the accuracy of the implementation. The image pyramid is calculated and stored on GPU; the same pyramid is used for all optical flow estimations that use the same two image frames. We utilize texture memory to store the image pyramid data because it works very well for our purposes; the data points evaluated are located in the same general region of the image frames. Therefore, it is beneficial to use texture memory here. For each level of the image pyramid a separate kernel call is used to estimate the optical flow of the data points for
that level. Since we use 6 different levels in the image pyramid, this means that 6 kernel calls are used to perform each LK operation. During each kernel call enough GPU threads are created so that each thread tracks a data point. The next pyramid level is then evaluated once all threads have completed their respective optical flow estimations.

Once the optical flow estimates are performed (both forward and backward) we do not immediately copy the resulting estimates back to the CPU. At this point we still have the results of the Forward and Backward runs of LK, as well as the original data point locations. With this information readily available on the GPU, we can calculate the error measures before copying the information back to the CPU. Both error calculations are simple enough that they can run on a GPU kernel. In addition, we continue to take advantage of the image frames stored on texture memory to extract the image patches for the NCC error. When the extracting the image patches for each data point, it tends to be the case that many patches reference the neighboring pixel values if not the same pixel values in the image frame. With the error measures calculated we then copy the resulting information back onto the CPU. The resulting bounding box construction is then performed the same as usual using this data.
5. COMBINATION

After both the detection and tracking stages have run for their duration, the next step in the recognition method is to combine the results of the two stages. Both of these stages return a number of bounding boxes which are believed to be the target pattern modeled by the detection stage. The combination stage is then tasked with determining which bounding boxes describe the same instance of a pattern and which describe separate instances. The end result is a collection of bounding boxes which serve several purposes. These boxes specify which images should be passed to the recognition stage for identification. They are also used as the previous-known-locations of pattern instances for the next run of the tracking stage.

The main means used to determine if two bounding boxes are describing the same pattern is by calculating the overlap between the two boxes. If there is significant overlap between the two boxes, then we generally assume that the boxes describe the same pattern instance. Figure 17 demonstrates such an example where multiple windows can describe the same pattern.

![Figure 17: Overlapping Windows Selecting the Pattern Instance [11]](image)

In the case that there is an overlap, we prefer the use the bounding box associated with the tracking stage over the box from the detection stage.
The recognition method is designed such that there is a hard limit to the number of patterns instances that are tracked. This limit is used to specify the maximum number of the bounding boxes returned by the tracking and detection stages. If after combination there are more boxes than this limit we select the top confident boxes within the limit. The confidence of a particular bounding box is determined by evaluating the box with the nearest-neighbor classifier from the detection stage. This is done because the nearest-neighbor classifier is the strongest classifier to available to determine if image is an instance of the target pattern. For the purpose of the research we set the hard limit to be four bounding boxes, although there would not be any problems with increasing this number.

5.2 GPU Acceleration

No GPU acceleration is used for the combination stage of the recognition method since the stage is relatively short. The main work involved for the stage involves finding the overlap between the bounding boxes and finding the confidence of the boxes in the case that the number of boxes exceeds the hard limit. While we do use the GPU accelerated version of the nearest-neighbor classifier for finding the confidence of each bounding box, the remaining work in the combination stage would not be accelerated if implemented on the GPU.
6. RECOGNITION

The recognition stage is the final stage of the overall recognition method. Given the bounding boxes found from combining the results of the tracking and detection stages, we extract the corresponding normalized image patches for those bounding boxes. These image patches are the input to the recognition stage, and the goal is to label each image patch with a unique identifier. If we assume that each image patch is an instance of some generic pattern (such as a face), then the recognition stage is used to distinguish between different instances of the pattern that appear in a particular video sequence.

For the remainder of this thesis we will assume that the detection stage has been trained to detect face patterns in video sequences, and that the recognition stage will be used to distinguish between the different faces that are found.

6.1 Individual Face Modeling

For the recognition stage we use a one-vs-all strategy when identifying faces, where we build a unique representation for each face that appears in the video sequence. Each representation is a separate pattern model that acts as an expert for that particular face and is given a unique identifier. The pattern model is then used to identify which image patches are instances of its modeled face in question, and which image patches are not. When an image patch is provided as input to the recognition stage, each of the pattern models evaluates the image patch (see Figure 18). The pattern model that provides the largest confidence value labels the image patch with its unique identifier.
Figure 18: Evaluating the Image Patch against Each Pattern Model (Image from [17])

When running the recognition method, we assume no prior knowledge of each face that appears in the video sequence. This leads to a significant learning challenge for the recognition stage. The stage initially starts without any pattern models stored. Whenever a new face appears in the video sequence it is then necessary to create a new pattern. It is up to the recognition stage to determine when an existing pattern model can be used to identify a particular face, and when a new pattern model must be created for the image patch in question. In addition, it is necessary to train and update these pattern models over time. Because we assume no prior knowledge of the faces encountered in the video sequences, we are given very little data to work with to train the new pattern model. In most cases we are only given the current image patch as training data to initial the pattern model. Therefore, we must train the pattern model to the best our ability using the available data, then continue to train the pattern model over time as more image patch data becomes available.

In response to these challenges, we explore several different pattern model implementations to find a model that works well for our purposes. The pattern model used is described in Section 6.3 and in Appendix B.
6.2 Assigning Pattern Models

Given a small set of image patches to identify, the recognition stage needs to quickly and accurately find the best assignment of pattern models to identify these patches. However, we cannot evaluate each image patch individually because there is no way to ensure that two image patches will not be labeled with the same identifier. Since we intend to use this method for facial recognition, we make the assumption that each face only appears at most once in a given image frame. Ultimately this means we need to find the best combination of pattern models to assign to the image patches. Ideally we would like to use a combination that will maximize the confidence of each pattern model to their respective image patch.

We treat this assignment problem as a case of the classical *stable marriage problem*. The stable marriage problem is a logic problem where we are given a set of *n* men and *m* women where each man and woman has a set of rankings for all members of the opposite sex. The goal is to find the best set of pairings between the men and woman so that each person is paired with their highest possible preference. The problem is considered solved if we can ensure that there are no two pairings *A* and *B* such that the man in *A* prefers the woman in *B* and the woman in *A* prefers the man in *B*. Although there are some variations to this problem, this is it in its simplest form. The most common solution to this problem is the *Gale-Shapley Algorithm* [18], which can be applied for the assignment problem encountered in our research. Gale-Shapley has been applied in similar matching problems, such as in Suvonvorn et al. [19]. The algorithm in terms of the assignment problem is presented below in Figure 19. In the algorithm the syntax (*p*, *m*) refers to image patch *p* being labeled using pattern model *m*. 
When generating the list of pattern models for each image patch, we choose not to include pattern models which reject the image patch since they shouldn’t be used to label it. We only create a new pattern model for a particular image patch when there are no more pattern models available to label the patch in the patch’s list.

After running the Gale-Shapely algorithm to determine the best assignment of pattern models to image patches, the image patches are used to train the pattern models. The label assignment is used to determine how the image patches are used in training. Training is performed using positive patches, which are examples of the face we wish to model, and
negative patches, which are examples of which do not represent the face we wish to model. For a particular pattern model, if any image patch was labeled using the model then the patch is considered positive. All other image patches are considered negative to that pattern model. The recognition stage provides all of the patches labeled as training data to the pattern model to improve its accuracy. The means in which that the image patches are actually used for training depend on the pattern model implementation.

6.3 Pattern Model Implementation

Several different pattern model implementations were studied to see if they would useful for identifying faces in a video sequence from scratch. After individually evaluating each of the different pattern models developed, it was determined that a pattern model utilizing nearest-neighbor classification and support vector machines (SVMs) [20] had the desired recognition accuracy. Details on the pattern models that were developed and how they were evaluated can be found in Appendix B.

6.4 GPU Acceleration

The recognition stage does not take advantage of any GPU acceleration for its implementation. The reason for this is because there are not any opportunities for GPU acceleration. The Gale-Shapely algorithm cannot particularly be parallelized since each image patch evaluation must be sequential. In addition, the processing times of the various pattern models explored were relatively small; if the models were pushed onto the GPU the overhead involved would only increase the processing time. Therefore, we elected to have the recognition stage remain on the CPU.
7. ENVIRONMENT AND EVALUATION

We will now talk about the hardware and methodology used to evaluate this new GPU-based technique. In order to demonstrate the gain in processing speed by acceleration on the GPU we evaluate our method using both a CPU-based implementation and a GPU-based implementation. The CPU-based method performs the same work as our proposed method, however all work is left on the CPU. Memory is not copied onto the GPU for this implementation.

7.1 Hardware

Both implementations of the code were run on a Dell XPS 720 desktop computer running Fedora 16 with an Intel Core 2 Duo Processor E6600. The processor contains two cores and runs at a clock speed of 2.4 GHz per core. As for the GPU used for the code, we made use of a NVIDIA Tesla C2050 GPU. This device contains 448 cores and runs at a clock speed of 1.15 GHz per core. Additional specification information on the devices (shown below in Figure 20) can be found in Appendix A at the end of this thesis.

Figure 20: Intel Core 2 Duo Processor E6600 and Tesla C2050
7.2 Software

Both the CPU and GPU-based implementations of the recognition method were programmed in C++. We use a C++ implementation of the TLD method developed by Nebehay in [12] for the base for both implementations. The TLD code was then modified to allow for multiple patterns to be tracked, and the recognition stage was then added. For the GPU-based implementations the tracking and detection stages were replaced with GPU-accelerated implementations.

GPU acceleration of the detection and tracking stages were accomplished using the CUDA architecture developed by NVIDIA. To simplify the code used in the GPU implementation of the recognition method, we took advantage of the Thrust parallel algorithms library [21]. The library is designed to offer classes and functionality that resemble the C++ Standard Template Library (STL) but they are executed on the GPU. In doing so the library provides these functions running at a much higher speed than the original STL. It was selected for our research because it also offers implementations for some common GPU accelerations, such as parallel reduction.

7.3 Evaluation Methodology

We evaluate the two different implementations using several different criteria. The first of these criteria is accuracy of the tracking and detection stages. Our first concern is to determine if the size and locations of the bounding boxes drawn around faces align with those boxes found in the ground truth data for a particular evaluation sequence. This is found by calculating the overlap between the bounding box in question and the bounding box given in the ground truth data. Overlap is defined by the ratio between the intersection and union of the two bounding boxes, as shown in Figure 21.
The areas of the different regions shown are calculated, and then these values are used to find the overlap value between the boxes. If the overlap between the two boxes is at least 25 percent [22] then they two boxes are considered to be overlapping. A correctly placed bounding box is commonly referred to as a *true positive*.

To express the accuracy of the bounding boxes given by the tracking and detection stages, we use the common *precision* and *recall* metrics. Precision is referred to as the number of true positives divided by the number of all detections made by the recognition method. Recall is the number of true positives divided by the number of bounding boxes that should have appeared (based on the ground truth data). The two metrics can be combined into a single score value by calculating the harmonic mean of the two values, shown in Equation 18. We refer to this score value as the *f-measure*.

\[
    f\text{-}measure = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}
\]  

To save space in the tables shown in this thesis, we only present the f-measure score.

We next concern ourselves with the accuracy of the recognition stage. For each face using the detection and tracking stages, the recognition stage is tasked with giving the face a
unique identifier to distinguish one face from any of the other faces that appear in the video sequence. We define the recognition accuracy to be how consistently a particular face is labeled a particular number that distinguishes it from other faces. After running the recognition method on a particular video sequence, the most common label for each of the faces that appear is found. The accuracy is then determined by how often each face is labeled by their most common label. No two faces can share the same most common label; if this is the case then one of the faces is evaluated using their second most common label.

Lastly we will discuss how the two implementations of the recognition method process the video sequences as input. There are two different processing modes available for a given video sequence. When running a video sequence using offline mode the recognition method has access to every frame of the video sequence. The frames are provided to the recognition method in sequential order. On the other hand, live streaming mode does not guarantee that the recognition method will have access to every frame; the availability of frames is dependent on the processing speed of the recognition method. After receiving the current image frame, the recognition method is timed in terms of how long it takes to process that frame. If the processing time is too long, then a number of frames are skipped based on how long the duration was.

Live streaming mode is meant to simulate running the recognition method working with a live feed rather than a pre-recorded video sequence. When working with a live feed the method receives the next current available frame in the feed rather than always the subsequent frame. This causes the processing speed to have a significant influence on the f-measure and recognition accuracy of the recognition method. We simulated the video sequences at 25 frames per second (fps). When running in live streaming mode, the f-measure and recognition accuracy is based on
the frames actually seen by the recognition method, rather than over all frames since it is impossible to detect and identify faces in frames that haven’t been processed.

### 7.4 Datasets

We evaluate the performance of the recognition method using two different datasets. The first of these datasets is the *Hannah Dataset* [23], which is a collection of annotations that describe the location and identity of every face that appears in the film *Hannah and Her Sisters* [11]. The film was directed by Woody Allen and released in 1986. Six different scenes were selected from the film to be used to evaluate the performance. These scenes were selected because they contain several common challenges to face tracking and identification, such as various face rotations and occlusions. The six scenes used from the film are shown below in Figure 22.

![Figure 22: Scenes from Hannah and Her Sisters Selected for Evaluation][1]

The other dataset used for evaluation is the *Surveillance Performance EValuation Initiative* (SPEVI) [17] dataset which is a collection of sequences with 3-4 people moving around in scene. The people that appear in the sequence repeatedly occlude each other while appearing...
and disappearing from the scene (see Figure 23). We use two out of the three sequences from the dataset. These sequences were found to be some of the more challenging sequences used to evaluate the recognition method, especially since some faces are seen for a very limited amount of time before they are occluded by another face. In addition, there are changes in scale as people move closer to and further away from the camera.

![Sample Frames from SPEVI Dataset](image)

**Figure 23: Sample Frames from SPEVI Dataset [17]**

To test the performance of the two implementations on the evaluation sequences, for each sequence we generated a unique face model for the detection stage to detect the faces that appear in the sequence. This was done to maximize the f-measure value for the two implementations, thus allowing us to better observe the drop in f-measure when comparing the results of running the implementations in offline and live streaming mode.
8. RESULTS

We report the results of evaluating the recognition method. As stated previously, we work with two different implementations of the recognition method; a pure CPU-based implementation that does not use any of the GPU accelerations, and a GPU-based implementation that does use the accelerations. We first evaluate the individual stages of the two implementations to show their individual gain in processing speed made possible using the GPU accelerations. Following this, we present the results of the overall implementations when running the evaluation sequences.

8.1 Individual Component Evaluations

In earlier sections of this thesis, several different GPU accelerations were given. These accelerations include the integral image calculations, the variance filter / ensemble classifier implementation, the nearest-neighbor implementation, and the Lucas-Kanade tracking implementation. These accelerations were evaluated in comparison to their CPU equivalents to determine the speedup gained from running on the GPU.

8.1.1 Integral Image Calculation Evaluations

The first set of evaluations that will be discussed will be integral image calculations. To evaluate these calculations, we generate a number of different sized images and time how long it takes to generate the integral image using the CPU implementation and GPU implementation of the calculations. We use a range of images from 200 pixels for the height/width to 1600 pixels for the height/width. For each image size we run the calculations 100 times and take the average processing time. The timing results are presented in Figure 24.
Figure 24: Processing Time Integral Image Calculations: CPU vs GPU

Processing runtimes are given in microseconds (μs). As the image size increases the processing time increases as well, although the CPU implementation takes much longer to calculate the image than the GPU implementation. At the largest image size (1600 by 1600 pixels), we found that the GPU implementation had a speedup of 4.5x over the CPU implementation.

It is worth noting that even though the integral image calculations are performed in parallel for the GPU implementation, the runtime of the method is still significantly dependent on the dimensions of the image. For each thread that propagates the sum down a column or row of the integral image, it still necessary to perform a loop of size height or width. This explains why the runtime of the GPU implementation increases with the image size, rather than staying relatively constant. In addition, although we do see significant processing speed with the integral image calculations, it is unfortunately difficult to see such speedup in practice. The problem lies in the common size of frames in video. For example, the frame size of the Hannah sequences is only 720 by 480 pixels, and the frame size of the SPEVI sequences is 720 by 576 pixels. Typically we don’t see image frames larger than this, other than in HD quality videos. That
being said, while we do not see particularly large speedups, calculating the integral images still has been found to be faster on the GPU than on the CPU, which should help with the runtime of the GPU implementation of the recognition method.

### 8.1.2 Variance Filter / Ensemble Classifier Evaluations

We next present the results of timing the variance filter and ensemble classifier implementations. The two classifiers of the detection cascade are evaluated together as they are both run on the same set of GPU threads in our GPU accelerated method. To evaluate the runtime, we find the average processing time for these classifiers when being running against the evaluation sequences. This is done for both the CPU implementation and the GPU implementation. The main challenge lies in not only the number of windows to be evaluated for a particular video sequence, but also the number of the windows that are accepted by the two classifiers. Depending on the particular image frame, a different number of windows could be accepted by the variance filter to be then evaluated by the ensemble classifier. This number of windows effects the processing time necessary to run all of the windows for the first part of the detection cascade. The runtimes recorded for the two implementations are given in Table 1.

<table>
<thead>
<tr>
<th></th>
<th>CPU (µs)</th>
<th>GPU (µs)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hannah 1</td>
<td>2541</td>
<td>367</td>
<td>6.92x</td>
</tr>
<tr>
<td>Hannah 2</td>
<td>3573</td>
<td>483</td>
<td>7.40x</td>
</tr>
<tr>
<td>Hannah 3</td>
<td>3849</td>
<td>470</td>
<td>8.19x</td>
</tr>
<tr>
<td>Hannah 4</td>
<td>4868</td>
<td>487</td>
<td>10.00x</td>
</tr>
<tr>
<td>Hannah 5</td>
<td>8040</td>
<td>931</td>
<td>8.64x</td>
</tr>
<tr>
<td>Hannah 6</td>
<td>4097</td>
<td>487</td>
<td>8.41x</td>
</tr>
<tr>
<td>SPEVI 1</td>
<td>18608</td>
<td>1943</td>
<td>9.58x</td>
</tr>
<tr>
<td>SPEVI 2</td>
<td>37227</td>
<td>4671</td>
<td>7.97x</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>10350</td>
<td>1230</td>
<td>8.39x</td>
</tr>
</tbody>
</table>
As shown in the table, the time needed to process the image frames for the GPU implementation is significantly smaller than the time for the CPU implementation. The speedup gained in using the GPU acceleration ranges from around 7x to 10x speedup, with an average speedup of approximately 8.39x.

8.1.3 Nearest-Neighbor Classifier Evaluations

To evaluate the nearest-neighbor classifier we had the two different implementations of classifier to evaluate a varying number of windows using an increasing number of examples. The number of positive/negative examples used ranged from 50 of each type of example to 1000 with steps of 50 examples (e.g. 50, 100, 150, etc.). The number of windows evaluated ranged from 10 to 50 at steps of 10 windows. For a specific number of examples and windows we take the average over 50 runs. The results of this test are presented in Figure 25.

![Figure 25: Processing Time of Nearest-Neighbor Classifier: CPU vs GPU](image)

In the figure above blue lines represent the runs of the CPU implementation and red lines represent runs of the GPU implementation. Each line is one of the implementations evaluating a fixed number of windows. As the number of examples and windows to evaluate increases, the
runtime of the classifier increases. However, this increase in runtime is much more significant with the CPU implementation than with the GPU implementation. Although it is difficult to see, there is indeed an increase in the runtime for the GPU implementation, but the increase is much less significant. As a result the lines for the GPU implementation in Figure 25 are very close together if not overlapping. The speedup achieved by using the GPU implementation was found to range from 18x to 33x when using the largest number of positive and negative examples.

### 8.1.2 Lucas-Kanade Tracking Evaluations

Lastly, we investigated the performance of the Lucas Kanade tracking implementations. For both the CPU and GPU implementations we evaluate not only over time taken to track data points forward and backwards between two image frames, but also over the time needed to calculate the tracking errors. When evaluating the tracking implementations, we used a varying number of data points to track over several pairs of image frames. The image frame pairs were selected to vary the duration of time between the two frames; one pair was comprised of sequential frames and others were separated by a gap of 1-5 frames. The reason for this was to add different levels of difficulty in tracking the data points. For each pair of image frames 100 different random bounding boxes were selected inside the frames to track. The average runtime over each of these pairs and bounding boxes was calculated, and the results are presented in Figure 26.
In the figure above the blue line is the CPU implementation and the red lines represent runs of the GPU implementation. As shown in the figure, the processing time generally increases as the number of data points to be tracked increases, although the increase rate is more apparent with the CPU version of LK than with the GPU version. The GPU implementation was found to run at a much faster rate than the CPU implementation, achieving a speedup of 7x when tracking the larger groups of points.

We do not see as much of an expected speedup with the GPU implementation of the LK, as compared to the other GPU accelerated methods, because the CPU implementation used was the OpenCV implementation. This implementation was designed to take advantage of CPU threads, meaning that it was already developed to take advantage of parallel processing and that it was already a very fast method. That being said, we still see a significant speedup with our GPU implementation over the OpenCV implementation. The OpenCV implementation only uses a single core to run the threads, unlike our implementation which uses hundreds of cores.
8.2 Overall Recognition Performance

Having found the runtime of the individual components, we now proceed to present the evaluation results of the overall CPU and GPU implementations of our recognition method. We first evaluate the two implementations in offline mode to find the baseline f-measure and recognition accuracy, as well as the frames-per-second (fps) of the two implementations. Following this we determine the loss in performance when running the two implementations in live streaming mode. For both sets of evaluations the values given are the average over multiple runs to ensure accurate results. Some screenshots of the recognition method operating are presented in Figure 27.

![Example Screenshots of Recognition Method Operation](image-url)

Figure 27: Example Screenshots of Recognition Method Operation
The results for running the two implementations of the recognition method in offline mode are given in Table 2. On top of giving the f-measure, recognition accuracy and frame rate for each evaluation sequence, we also give the average performance of both implementations.

Table 2: Recognition Method Performance (Offline Mode): CPU vs GPU

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th></th>
<th></th>
<th>CPU</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>f-measure</td>
<td>recognition</td>
<td>fps</td>
<td>f-measure</td>
<td>recognition</td>
<td>fps</td>
</tr>
<tr>
<td>Hannah 1</td>
<td>0.970</td>
<td>0.913</td>
<td>16.49</td>
<td>0.970</td>
<td>0.913</td>
<td>41.20</td>
</tr>
<tr>
<td>Hannah 2</td>
<td>0.844</td>
<td>0.724</td>
<td>10.52</td>
<td>0.824</td>
<td>0.832</td>
<td>39.93</td>
</tr>
<tr>
<td>Hannah 3</td>
<td>0.872</td>
<td>0.851</td>
<td>14.37</td>
<td>0.874</td>
<td>0.800</td>
<td>39.56</td>
</tr>
<tr>
<td>Hannah 4</td>
<td>0.751</td>
<td>0.908</td>
<td>10.04</td>
<td>0.814</td>
<td>0.998</td>
<td>36.34</td>
</tr>
<tr>
<td>Hannah 5</td>
<td>0.927</td>
<td>1.000</td>
<td>7.31</td>
<td>0.927</td>
<td>0.764</td>
<td>36.81</td>
</tr>
<tr>
<td>Hannah 6</td>
<td>0.924</td>
<td>0.582</td>
<td>13.10</td>
<td>0.928</td>
<td>0.839</td>
<td>37.95</td>
</tr>
<tr>
<td>SPEVI 1</td>
<td>0.798</td>
<td>0.743</td>
<td>6.17</td>
<td>0.800</td>
<td>0.717</td>
<td>30.08</td>
</tr>
<tr>
<td>SPEVI 2</td>
<td>0.902</td>
<td>0.848</td>
<td>4.15</td>
<td>0.902</td>
<td>0.865</td>
<td>23.90</td>
</tr>
<tr>
<td>Average</td>
<td><strong>0.874</strong></td>
<td><strong>0.821</strong></td>
<td><strong>10.31</strong></td>
<td><strong>0.880</strong></td>
<td><strong>0.841</strong></td>
<td><strong>35.72</strong></td>
</tr>
</tbody>
</table>

The most immediate observation from the data in Table 2 is that the f-measure and recognition for the two implementations differ to some degree. This is to be expected; even though the two different implementations perform the same amount of work and the same calculations (albeit in different methods), this does not ensure that the results will have exactly the same results. The issue lies in the fact that the calculations involved are being performed on different architectures (CPU vs GPU). It is not unusual for different architectures to perform floating point calculations differently. Rounding errors may be handled differently depending on the architecture, among other differences. In most cases this is normally a trivial issue. For example, in the case of the NCC error calculations for the tracking stage, the difference in calculations between the CPU implementation and the GPU implementation was found to be estimated at 0.000002.
However, there are some aspects of the recognition method where this amount of floating point difference is significant. Using the same example as before, recall that the tracking stage only uses the data points whose NCC error is above the median of the calculated error values (\(med_{NCC}\)) to determine the new bounding box. Having the error for a particular data point differ by approximately 0.000002 is sufficient enough to be greater than or less than \(med_{NCC}\) when it originally wasn’t. This causes some of the resulting bounding boxes to be slightly different when comparing the CPU implementation vs the GPU implementation. This difference also propagates to the recognition stage of the method, even though it is solely implemented on the CPU. Since the bounding boxes for the two implementations are slightly different, this means that the pattern models for the two implementations will learn differently. Ultimately this means that both the f-measure and the recognition accuracy values for the two different implementations of the recognition method will be slightly different. To account for this, we find the percentage loss for the CPU implementation and the GPU implementation, and choose to compare the implementations based on loss rather than specific f-measure and recognition accuracy values.

The other observation that can be gathered from Table 2 is the difference in frame rate for the two implementations. Based on data given, we found that the GPU implementation had a speedup over the CPU implementation ranging from 2.5x to 6x, with an average speed up of approximately 3.5x. While this is a significant speedup over the CPU implementation, it can be said that it does not resemble the speedup found for the individual components given previously in this section. This is because there are many components in the recognition method that could not be implemented on the GPU without significantly slowing down the method. Since these components were left on the CPU for both implementations, this causes the overall speedup for
GPU implementation to become closer to 1x. This is why the overall speedup does not resemble the speedup of the GPU accelerated components.

We now move onto the results of running the two implementations of the recognition method using live streaming mode. As a reminder, live streaming mode is where the recognition method retrieves frames from the evaluation sequence based on the amount of time taken to process the previous frames. If the processing time takes too long, then frames can be skipped, leading to the possibility of faces being lost or identified incorrectly. The results of running the implementations in this mode are shown in Table 3. Here we do not include the frame rate of the implementations since the values are the same as before when running using offline mode. However, we do include the average number of frames lost after each iteration of the recognition method.

Table 3: Recognition Method Performance (Live Streaming Mode): CPU vs GPU

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th></th>
<th></th>
<th>GPU</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>f-measure</td>
<td>recognition</td>
<td>frames lost</td>
<td>f-measure</td>
<td>recognition</td>
<td>frames lost</td>
</tr>
<tr>
<td>Hannah 1</td>
<td>0.956</td>
<td>0.930</td>
<td>1.18</td>
<td>0.967</td>
<td>0.914</td>
<td>0.57</td>
</tr>
<tr>
<td>Hannah 2</td>
<td>0.732</td>
<td>0.955</td>
<td>1.54</td>
<td>0.804</td>
<td>0.826</td>
<td>0.61</td>
</tr>
<tr>
<td>Hannah 3</td>
<td>0.768</td>
<td>0.810</td>
<td>1.14</td>
<td>0.824</td>
<td>0.836</td>
<td>0.62</td>
</tr>
<tr>
<td>Hannah 4</td>
<td>0.273</td>
<td>0.713</td>
<td>1.73</td>
<td>0.702</td>
<td>0.988</td>
<td>0.73</td>
</tr>
<tr>
<td>Hannah 5</td>
<td>0.913</td>
<td>0.881</td>
<td>2.33</td>
<td>0.927</td>
<td>0.843</td>
<td>0.69</td>
</tr>
<tr>
<td>Hannah 6</td>
<td>0.843</td>
<td>0.675</td>
<td>1.20</td>
<td>0.822</td>
<td>0.635</td>
<td>0.62</td>
</tr>
<tr>
<td>SPEVI 1</td>
<td>0.574</td>
<td>0.598</td>
<td>4.34</td>
<td>0.750</td>
<td>0.590</td>
<td>1.08</td>
</tr>
<tr>
<td>SPEVI 2</td>
<td>0.671</td>
<td>0.594</td>
<td>4.89</td>
<td>0.845</td>
<td>0.717</td>
<td>1.15</td>
</tr>
<tr>
<td>Average</td>
<td>0.716</td>
<td>0.769</td>
<td>2.29</td>
<td>0.830</td>
<td>0.794</td>
<td>0.75</td>
</tr>
</tbody>
</table>

Based on the results from Table 2 and Table 3, we can find the percentage loss when running the implementations using live streaming mode (Table 4).
### Table 4: Percentage Loss When Running Recognition Method in Live Streaming Mode

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>f-measure</td>
<td>22%</td>
<td>6%</td>
</tr>
<tr>
<td>recognition</td>
<td>7%</td>
<td>6%</td>
</tr>
</tbody>
</table>

Most notably the f-measure for the CPU implementation has a large drop when compared to its performance during offline mode. This tells us that the implementation was more likely to lose track of certain faces due to certain image frames being lost. On the other hand, the GPU implementation was found to have lost a smaller number of frames and achieved a much smaller drop in f-measure. This is due to the GPU implementation having a faster frame rate for processing each image frame.

It’s interesting to note that while the f-measure for the CPU-based implementation took a significant hit when using live streaming mode, the recognition accuracy did not fall as much. We believe that the reason for this is that enough information was collected via the detection and tracking stages that the pattern models were still able to distinguish between the different faces that appear in the evaluation sequences. In a few cases the recognition accuracy actually improved during live streaming mode, achieving recognition accuracy larger than when using offline mode. This is because some of the frames lost during live streaming mode prevented the pattern models from learning certain image patches that would have affect the pattern models’ accuracy. It has been found during the research that learning certain image patches can cause the pattern models to make more mistakes. For example, viewing a person’s face from a certain angle may cause the face to appear similar to another person’s face. Normally this is up to the pattern model to be able to identify the differences in the two faces, but one of the challenges encountered during this research was building a pattern model using a limited amount of training...
data. With a limited amount of data it is more difficult to distinguish these differences. Not only that but by identifying the wrong face as the modeled face, the pattern model may start learning that face since it was previously identified as the modeled face. Therefore, by having a particular image patch skipped for the recognition stage, this can lead to the pattern models avoiding making certain mistakes.
9. CONCLUSIONS AND FUTURE WORK

We developed in this research a novel facial recognition technique that identifies faces that appear in a video sequence. By accelerating components of this method on the GPU we achieve a significant increase in processing speed that allows the method to perform well when processing live streaming data. This aspect is important because many of the well-known pattern recognition methods do not address the issue of processing speed when being developed. These methods tend to have a low frame rate as a result, making them ill-suited for working with live streaming data. As shown in the results section of this thesis, it was found that the CPU implementation suffered a 22% loss in f-measure when working with live streaming data, whereas the GPU implementation only suffered a 6% in f-measure. Considering that there are many applications of pattern recognition using live streaming data, such as in security, the processing speed of a pattern recognition algorithm is a significant aspect that should be considered when developing the algorithm. GPU acceleration is an excellent option to improve the processing runtime of an algorithm, as long as there are opportunities for parallelization in the algorithm.

In regards to our recognition method, there may be additional opportunities to accelerate our method on the GPU. For example, the speedup found for calculating the integral images for the variance filter could possibly be improved. In our calculations of the integral image, we use a loop in each GPU thread to propagate the sum across each row and column. In [14], which is the basis for the way we calculate the integral images, parallel reduction is used to propagate the sum rather than using a loop. We found that the overhead involved in performing the reduction was too large for the frame sizes typically encountered in the evaluation sequences; therefore we
choose not to use parallel reduction. That being said, there may be other ways to utilize parallel reduction differently from [14] to improve the integral image calculations for small frame sizes.

Another option for improving the processing time of our recognition method is the concept of *dynamic parallelism*. Dynamic parallelism refers to the idea that additional GPU threads can be spawned from other GPU threads rather than just from the CPU. This is very appealing since it eliminates the need to wait on the CPU to spawn new GPU threads. One possible application of this concept with our recognition method would be with the two segments of the detection cascade. Rather than having to wait on the first two classifiers to complete in order to run the nearest-neighbor classifier, we could spawn a set of threads straight from the GPU to evaluate the set of examples for a given window. Dynamic parallelism was introduced by NVIDIA this past year for CUDA versions 5.0 and up. Unfortunately, in order to use dynamic parallelism, it is necessary to utilize NVIDIA’s *Kepler* line of GPU cards, which was not used for this research.

There is also some further work that could be done to improve the pattern models used in the recognition stage of our method. Currently the pattern models are allowed to learn after every run of the recognition cascade. However, it was found in the results section of this thesis that not learning every single image frame can sometimes lead to better recognition accuracy (based on evaluating the CPU implementation during live streaming mode). A heuristic could be developed to determine when the pattern models should learn from the image patches. In addition, the runtime of each pattern model was timed to be very small, meaning we could introduce additional classification work to improve the recognition accuracy without greatly affecting the overall runtime. Possible improvements include introducing more powerful features or adding another machine learning model to assist in classification.
APPENDIX A. Hardware Specifications

Here are the specifications for the CPU and GPU used in the research, presented in Tables 5 and 6. Information on the CPU was provided by [24] [25] and information on the GPU was provided by [26] [27].

Table 5: CPU Specifications

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Cores</td>
<td>2</td>
</tr>
<tr>
<td>Clock Speed</td>
<td>2.4 GHz</td>
</tr>
<tr>
<td>L1 Cache Size</td>
<td>128 KB</td>
</tr>
<tr>
<td>L2 Cache Size</td>
<td>4096 KB</td>
</tr>
<tr>
<td>Max TDP (Power Consumption)</td>
<td>65 W</td>
</tr>
<tr>
<td>FSB Speed</td>
<td>1066 MHz</td>
</tr>
<tr>
<td>Lithography</td>
<td>65 nm</td>
</tr>
</tbody>
</table>

Table 6: GPU Specifications

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Cores</td>
<td>448</td>
</tr>
<tr>
<td>Clock Speed</td>
<td>1.15 GHz</td>
</tr>
<tr>
<td>L1 Cache Size</td>
<td>64 KB</td>
</tr>
<tr>
<td>L2 Cache Size</td>
<td>768 KB</td>
</tr>
<tr>
<td>Max TDP (Power Consumption)</td>
<td>247 W</td>
</tr>
<tr>
<td>Double Precision Floating Point Performance (Peak)</td>
<td>515 Gflops</td>
</tr>
<tr>
<td>Single Precision Floating Point Performance (Peak)</td>
<td>1.03 Tflops</td>
</tr>
<tr>
<td>Total Dedicated Memory</td>
<td>3 GB</td>
</tr>
<tr>
<td>Memory Speed</td>
<td>1.5 GHz</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>144 GB/sec</td>
</tr>
<tr>
<td>Constant Memory Size</td>
<td>64 KB</td>
</tr>
<tr>
<td>Shared Memory Size per Block</td>
<td>48 KB</td>
</tr>
<tr>
<td>Number of Registers per Block</td>
<td>32768</td>
</tr>
<tr>
<td>Max Number of Threads per Multiprocessor</td>
<td>1536</td>
</tr>
<tr>
<td>Max Number of Threads per Block</td>
<td>1024</td>
</tr>
</tbody>
</table>
APPENDIX B. Development of the Pattern Models

Several different pattern model implementations were studied to see if they would be useful for identifying faces in a video sequence using a minimal amount of training data. Both temporal and non-temporal based models were explored; temporal-based models calculate the confidence of a particular image patch based both on the current patch and previously identified patches for that model. These types of models are commonly used to learn changes over time, and generally have better accuracy than those models that only use the current image patch. However, temporal based models are more difficult to train as well. Details on each of the different model implementations are given below.

B.1 Nearest-Neighbor Modeling

The first of the pattern models explored is a model based on the nearest-neighbor (NN) classifier used in the detection cascade. The operation of this pattern model is relatively the same as the original classifier, although there is a significant difference. In particular, the model is designed so that if there are not any negative examples stored in memory that a confidence value can still be returned. With the original nearest-neighbor classifier a confidence value was not calculated if there were not any positive or negative examples stored. This is because the confidence calculations ask for both a positive distance and a negative distance. However, there are sometimes instances where there is not any negative data available, such as when a video sequence starts with a single face appearing in the scene. With a single face there are no other faces to be used as negative data. In this case there is only positive data available to train and initialize the pattern model. To bypass this limitation, in the absence of negative data, the pattern model is designed to use the distance to the nearest positive example as the confidence. Once the
The first negative example is introduced the pattern model then uses the same calculations as before for finding the confidence value of an image patch.

The pattern model is trained in the same way that the nearest-neighbor classifier is trained, by introducing image patches to the set of positive and negative examples. In the original TLD method, image patches were collected by a learning stage to act as training data for the classifier. The classifier then determines whether or not each image patch would be useful if added to the set of examples. This is done using a bootstrapping technique developed by Kalal et al. [1] called P-N Learning. The technique maintains two different constraints, a P-constraint and an N-constraint, which determine the desired confidence range for each positive and negative example, respectively. If a provided image patch violates one of these constraints, then the image patch is added to set of stored examples. This prevents the patch and similar patches from the failing the constraints in the future since the patch is now stored in memory.

To train the pattern model using P-N Learning, we use the image patches provided by the recognition stage of our recognition method as input. If there are not negative patches stored for the pattern model, then the first negative patch introduced is automatically accepted.

**B.2 Conditional Random Field Modeling**

The temporal-based model that was explored for the research was the Conditional Random Field (CRF) statistical model [28]. CRFs are considered temporal-based models because they estimate the probability of changes over time based on certain states. In terms of our research, CRFs are used to model the probability that one of the next image patches to be evaluated will or will not be the modeled face, given the results of the past few runs of the recognition stage.
We use two different states in our CRF implementation of the pattern model; a positive state which refers to the current image patch being the modeled face, and a negative state which refers to the current image patch not being the modeled face. CRFs attempt to model the changes in state over time, assuming that the state either remains the same or it changes to another state. There are several different ways to model these changes in state; the most common way to model the transitions is to use the current state and the previous state. This implementation is referred to as a linear-chain CRF, which is what we use for the pattern model. This leaves us with four possible state transitions to model, given a positive and negative state. These transitions are as follows: the positive state remaining in the positive state, the positive state changing to a negative state, the negative state remaining in the negative state, and the negative state changing to the positive state. These transitions are shown in Figure 28 using a two-state diagram.

In order to model these four different state transitions, a different set of feature evaluations are performed for each transition. Technically speaking, for each image patch we only need to evaluate for two different transitions. The CRF model keeps track of the previous state in memory; the model then only needs to evaluate the transitions from the stored previous state to either the positive or the negative state. For each transition that is evaluated, a set of features are extracted from the image patch and then weights are applied to the features. We use
2-bit binary features for our implementation. The confidence of a particular image patch is the sum of the weighted features (as shown in Equation 19).

\[
\text{transition}_j \text{ confidence} = \exp \left( \sum_{i=1}^{n} \lambda_{ij} f_i(P) \right)
\]

Here \( P \) is the image patch, \( f \) is a function that calculates the feature, and \( \lambda \) is the set of weights applied to the features. The value \( j \) refers to the transition we are currently calculating the confidence for. The state transition that has the largest confidence is selected as the predicted transition. If that transition ends in the positive state, then the image patch is considered the modeled face.

While the CRF model is certainly a powerful model, there is some difficulty in training the model. Specifically, not only do we need to train the model using positive and negative examples, but we also need to train for these examples for the four different transitions. This requires a significant amount of training data to model these transitions, and it is absolutely necessary to have both positive and negative examples. CRFs can be trained in several different ways; we choose to use the maximum-likelihood method [29], which uses a set of known labeled transitions and adjusts the feature weights to maximize the probability that the predictions made by the model match the labeling. Several iterations of the training method may be necessary to adjust the feature weights to maximize probability.

In order to provide training data for all transitions, we need to have both positive and negative examples. The training method does not allow us to work without negative examples like with the nearest-neighbor pattern model. To provide the needed negative data to train the CRF model when none is available, we introduce a technique called *synthetic negative training*.
When the recognition stage is initialized, a small dataset of face images are uploaded (~30) and stored in memory. Every time that a CRF model is created, we use the associated positive image patch to select examples from the dataset to act as negative examples, which are then used to train the CRF model. Figure 29 depicts this concept.

![Diagram](image)

**Figure 29: Using Synthetic Negative Training for the CRF Pattern Model (Images from [30])**

Negative examples are selected by finding the Normalized Cross Correlation (NCC) between an example and the positive example. The most similar patches to the positive example but are below some threshold (approximately .75) are used as negative examples. This is done because negative examples should identify faces that should definitely be considered to not be the modeled face. The most common mistake made by any of the pattern models is misclassifying a face as the modeled face because it is similar to the modeled face. By selecting negative examples that are similar to the face but still considered to be negative, we prevent some of these common mistakes from occurring. The small dataset of faces images used for SNT was pulled from the *Chokepoint Dataset* [30]. The dataset is a collection of video sequences that depict people walking through hallways and entryways. We use face images extracted from the sequences to act as the small dataset.

The results of using the SNT technique will be provided later on in Appendix B. Although negative examples selected by the technique have been very useful for identifying,
using these examples do not have the same effect as the negative examples actually found in the evaluation sequences. This is because there is no way to ensure that the dataset loaded will contain negative examples encountered in the sequences. This is why we use the term *synthetic* to describe this training technique; these examples could be considered to not be “true” negative examples. By using machine learning models that generalize well to new data, we negate this issue to some degree. Thankfully the CRF pattern model is such a model that does generalize well, so we can use this technique. However, we cannot use this technique for the NN pattern model since it calculates similarity based on pixel-by-pixel calculations, making it ill-suited for training using SNT.

**B.3 Support Vector Machine Modeling**

Support Vector Machines (SVMs) are a type of machine learning model that organizes data on a hyperplane such that sets of examples are linearly separable. In other words, we wish to build a representation of data that allows us to draw a single line to separate positive and negative examples if they were plotted on a graph.

![Support Vector Machines to Separate Groups of Data](image)

*Figure 30: Using Support Vector Machines to Separate Groups of Data [31]*

Figure 30 gives an example of this sort of representation, where black circles are one set of examples and black triangles are another set. Various transforms and functions are available to
map the data in different spaces to make it easier to create this dividing line. In terms of implementation, a series of features are extracted from each image patch, and a set of weights are then applied to the features to determine which side of the line the image patch should fall on.

$$\text{confidence} = \sum_{i=1}^{n} w_i f_i(P)$$  \hspace{1cm} (20)

Equation 20 is a simple representation of this methodology, where \( P \) is the image patch, \( f \) is a function that calculates the feature, and \( w \) is the set of weights applied to the features. The confidence is found from the sum of these weighted features. If the confidence is above some threshold, then the model labels the image patch as one class; otherwise it is labeled the other class. For our purposes we use 100 randomized 2-bit binary features for the SVM implementation.

The SVM model is trained by introducing positive and negative examples, and then verifying that they fall on the correct side of the separating line. If any examples violate the linearly separable rule, then an iterative method, \textit{sequential minimal optimization} [32], is applied to adjust the separating line by modifying weights. Similar to the previous models, SVMs require that there is training data available for both positive and negative examples. We have found that the SNT technique created for the CRF pattern model is useful for initially training the SVM pattern model since it too generalizes well to new data.

For the SVM implementation used for the pattern model we utilize a modified version of an open source library called LIBSVM [33]. The library is modified to simplify and eliminate some unnecessary operations since it is known exactly how the library will be used.
B.4 NN / CRF Modeling

As an alternative to initially training the CRF pattern model using the SNT technique, a combination pattern model was created using the NN pattern model and the CRF pattern model. When the combination model is first created, the model behaves exactly the same as the NN model. Once enough positive and negative examples are stored in memory, we use these examples to initialize a method that behaves the same as the CRF model. By training the CRF method in this way we eliminate the need for synthetic negative data since we now have available negative data that appears in the evaluation sequence.

With CRF functionality now available, we then calculate the confidence for a particular image patch to be a weighted average of the scores from both the NN and CRF methods. Each method evaluates the image patch independently from the other method. We continue to use the confidence found from the NN part of the combination model to support the decisions made by the CRF part of the combination model. Both parts of the combination model are trained using the same techniques as before. The NN method is trained using P-N Learning and the CRF method is trained using the maximum-likelihood method.

B.5 NN / SVM Modeling

By studying the performance of the NN pattern model and the SVM pattern model, it was found that the two models made different mistakes on the same evaluation datasets. Therefore, a second combination pattern model was designed that gives a confidence value based on the combined scores of the NN and SVM methods. The belief is that by using these methods together that they can correct the mistakes made by each other. Both methods are initialized at model creation and are used to evaluate each image patch. The two evaluations are independent of each other and the resulting confidence score is a weighted average of the two resulting
scores. The two parts of the combination model are trained using the same techniques as before, using P-N Learning and sequential minimal optimization.

**B.6 Evaluation**

Having developed several different pattern models to identify faces, we then evaluated each of the models to determine which would be best suited for the recognition method. This was done using the recognition stage developed for the recognition method. The recognition stage is provided the ground truth information for the current evaluation sequence. We then evaluate the stage using each of the different pattern models. We use the recognition accuracy metric described in Section 7 where the accuracy is determined by how often each face is labeled by their most common label.

For each pattern model we evaluate in two different ways. We first evaluate without any pattern models already created and it is up to the recognition stage to create and train models as necessary to accurately label each face. This evaluation is meant to determine the model’s ability to learn using live streaming data (*live training*). The second evaluation involves training a model for each face that appears in the evaluation sequence, and making these models available to the recognition stage upon initialization. The recognition stage is not permitted to create new pattern models or train existing ones. This evaluation is meant to determine the model’s ability in general to identify a face (*pre-training*).

We use the Hannah video sequences to perform the pattern model evaluations, and the results of these evaluations are presented below in Table 7. To simplify reading we only provide the average recognition accuracy for each pattern model and evaluation.
Table 7: Pattern Model Evaluations using Hannah Sequences

<table>
<thead>
<tr>
<th>Model</th>
<th>Live Training</th>
<th>Pre-Training</th>
</tr>
</thead>
<tbody>
<tr>
<td>NN Model</td>
<td>0.798</td>
<td>0.749</td>
</tr>
<tr>
<td>CRF Model</td>
<td>0.457</td>
<td>0.784</td>
</tr>
<tr>
<td>SVM Model</td>
<td>0.748</td>
<td>0.826</td>
</tr>
<tr>
<td>NN / CRF Model</td>
<td>0.727</td>
<td>0.766</td>
</tr>
<tr>
<td>NN / SVM Model</td>
<td>0.832</td>
<td>0.796</td>
</tr>
</tbody>
</table>

There are several observations that can be made from the evaluations. The first observation is that the CRF pattern model had the most difficulty during the live training evaluation. This is not because the pattern model is a poor model; for the pre-training evaluations the accuracy of the CRF pattern model is comparable to the other pattern models. That being said, several different variations of the CRF pattern model were attempted and the live training accuracy was about the same. In addition, the combination pattern model involving the CRF model and the NN model had a lower live training accuracy than the live training accuracy of the NN model by itself. This tells us that the CRF methodology is not well suited for live training. Since there is a limited amount of data available for training, this would lead to some difficulty in training the CRF method. This is the cause of the low live training accuracy. Worth noting is that some searching was performed to find an existing implementation of Conditional Random Fields that supports live training. Nothing particularly successful was found.

Another observation is that it was found pattern models generally made the most mistakes around the time they were first initialized. With a limited amount of training data available it is more challenging for a pattern model to make accurate classifications. Therefore, it is important to have the pattern model to be as accurate as possible to support the model during
this early stage of operation, at least until more training data becomes available. In this respect, between the two combination pattern models reviewed, the NN / SVM pattern model would be better suited for our purposes since both parts of the model are available for classification at initialization. And this does turn out to be the case, as this combination model had the best accuracy for live training evaluations. It is common knowledge that ensemble-based classification methods general achieve better accuracy, so this outcome makes sense.

The fact that the combination model generally had the best overall accuracy leads to an interesting question: if additional machine learning methods were introduced to the pattern model would the accuracy improve even more? At the time of this thesis there was not a chance to attempt additional methods, although this would be a good area to explore in future work. In any case we choose to use the NN / SVM combination pattern model in our recognition method.
Bibliography


Y. Ouerhani, M. Jridi and C. Brosseau, "Optimized pre-processing input plane GPU implementation of an optical face recognition technique using a segmented phase only composite filter," *Optical Communications*, vol. 289, pp. 33-44, February 2013.


G. Nebehay, "Robust Object Tracking Based on Tracking-Learning-Detection," The Vienna University of Technology, 2012.


E. Maggio, E. Piccardo, C. Regazzoni and A. Cavallaro, "Particle phd filtering for multi-target visual tracking," in *IEEE International Conference on Acoustics, Speed and Signal*


[27] "Tesa GPU Computing, Tesla C2050 Performance Benchmarks," NVIDIA.


