Utilizing Graphics Processing Units for Rapid Facial Recognition using Video Input

Charles Gala, Dr. Raj Acharya
Department of Computer Science and Engineering
Pennsylvania State University
State College, Pennsylvania, USA

Bruce Einfalt
Embedded Systems and Technology
Applied Research Laboratory
State College, Pennsylvania, USA

***Abstract*— We propose a facial recognition method that is implemented utilizing a Graphics Processing Unit (GPU) to accelerate the processing time for a single frame of an input video sequence. When working with live streaming data, the processing speed that a particular frame is processed is a significant factor. A fast processing speed is necessary so that a recognition method can keep up with the frame rate of the video sequence. GPU computing is an ideal method to accelerate recognition processing. By implementing components of a facial recognition method on a Tesla C2050 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.**

Keywords*— *facial recognition; GPU; CUDA; parallel computing

I. 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. However, there are significant challenges to utilizing input in the form of with live streaming video. The main challenge is the speed at which frames from the video are processed. In the time that the algorithm is processing a single frame, the next available frame could be replaced multiple times. 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 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. 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 paper we develop a facial 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 the research, which is run for each frame of the input video sequence.

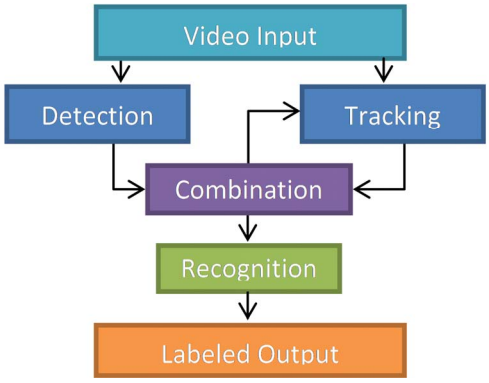


Fig. 1. General Structure of the Recognition Method

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 is modeled, such as a face, in a particular frame of the video sequence. The tracking stage takes previous known instances of the target pattern and tries to predict the next instances. Both the detection and tracking stages are based on the TLD (Tracking-Learning-Detection) method introduced by Kalal [1]. Both 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 faces described by the bounding boxes found and identifies them. We utilize a pattern model developed for learning a particular face on-the-fly using a minimal amount of initialization data.

The remainder of this paper is as follows. Sections 2 and 3 describe the detection and tracking stages, respectively, used in our recognition method. The combination of results from the two stages is described in Section 4. The recognition stage of the method is discussed in Section 5. The evaluation of the facial recognition method is given in Section 6. Concluding remarks are then provided in Section 7.

978-1-4799-6233-4/14/$31.00 ©2014 IEEE

II. DETECTION

The purpose of the detection stage is to find new instances of some target pattern being modeled, such as a face. To perform detection we utilize 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 stage and dimensions of the input video sequence, and can range from 10,000 to 250,000 windows. Each window is evaluated independently of other window evaluations. We take advantage of a cascade of classifiers to quickly detect instances of the target pattern. The 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. This effectively reduces the processing time to evaluate every window. After the cascade has performed its run, any windows that haven't been rejected are used as the result of the detection stage.

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

A. Variance Filter

The variance filter 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 to extract few meaningful features from these windows. The filter rejects those with windows with low variance (smaller than some threshold).

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 [2] 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. Two integral images must be maintained to find the variance; one to find the sum of the region and one to find the squared sum. 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.

B. Ensemble Classifier

The second stage of the detection cascade is an ensemble classifier composed of several randomized ferns [1]. 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 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.

Each fern in the ensemble classifier evaluates over a set of randomized 2-bit binary features. Each feature is a comparison between two pixel values performed on the region of the image described by the window, in which the locations are determined by random distribution. Once all of the features are calculated for a particular fern, their values are used to look up the particular confidence score assigned for the arrangement of features. For our purposes, we use 13 randomized ferns for the ensemble classifier and 10 features for each fern.

C. Nearest-Neighbor Classifier

The final stage of the detection cascade is a nearest-neighbor classifier that behaves as a template-matching technique [1]. Decisions are based on a set of normalized image patches stored in memory. Both image patches representing the target pattern (positive examples) and not representing the target pattern (negative examples) are stored and used in the classification process. Using these examples, a confidence value 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 Cross Correlation (NCC) between each example stored and the normalized image bounded by the window. For each example in the classifier's memory we use NCC to find the distance between the window and example images. The smaller the distance between the image the more similar they are. The minimum distance values to the positive examples and the negative examples are then found and used to calculate the confidence value for a particular window. 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.

D. 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.

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. This works out as the amount of work per window is relatively small for the ensemble classifier and variance filter. A number of threads are created, where each thread evaluates a different window using the two classifiers. We keep track of whether a window passes or fails the two classifiers using an array of results, which are copied back to the CPU when the threads complete. The windows that have been marked as passed are presented to the second segment of the detection cascade.

We take advantage of the GPU as well for calculating the integral images used for the variance filter. New integral images must be calculated for each frame of the video sequence. Originally the integral image is found by propagating the sum from the top left of the image to the bottom right. This method is difficult to parallelize. As an alternative, Bilgic [3] introduced a method where the integral image is found in two operations. The sum is first propagated

over each row from left to right, and then it is propagated over each column from top to bottom. The result of this method is the same as if we were to use the original method to find the image. 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; one for performing summations over each row and the other call over each column.

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 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. It is only necessary then to make one kernel function call as opposed to having to make multiple calls.

Once we find the distances for every window and example combination, there is still the issue of finding the minimal positive and negative distances for each window. This is done by applying a variation of *parallel reduction*. The technique is used to apply some operation, such as summation, to all elements in a very large array by breaking the array up into small sub-problems. Additional sub-problems are created using the result of the previous set until there is one thread left, which calculates the final solution. We apply a variation of parallel reduction where we find the minimum distance value for each set of positive and negative examples for all windows simultaneously. 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. We then stop the 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 frame image in texture memory to accelerate the image extraction process since we are extracting many values from the same region in memory. In addition we store these extracted images in constant memory so they can be quickly accessed for each of the NCC calculations.

III. Tracking

Object tracking is necessary because it is unreasonable to assume that the recognition 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 the motion of each data point.

A. Optical Flow Estimation

We use the popular *Lucas-Kanade* tracking technique [4] for optical flow estimation to estimate the motion of each data point, specifically the pyramidal version of the technique. Scaled down versions of the image frames are used to acquire a rough estimate of the optical flow which is then refined using multiple runs of Lucas-Kanade with larger scaled images. 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 images. A set of 400+ data points are generated equally spaced within the bounding box found for the previous pattern instance location. Each of the generated data points are tracked using Lucas-Kanade and the resulting estimates are used to generate a new bounding box. Only the top confident estimates are used in finding the new bounding box. We use two different error measures to determine the confidence of a data point. The first error is Forward-Backward error introduced by Kalal [1], and the second error is found using NCC. Small image patches are extracted at the locations before and after motion, and NCC determines the similarity between patches.

B. GPU Acceleration

To accelerate the tracking stage of the TLD method on the GPU, we develop a GPU-based 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.

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 since the data points evaluated are located in the same general region of the image frames. 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 each thread tracks a different 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, we calculate the error measures for each data point in parallel, and then copy the results back to the CPU.

IV. 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. The combination stage

finds bounding boxes between the two stage results that have significant overlap. If there is significant overlap between the two boxes, then we generally assume that the boxes describe the same pattern instance. The stage combines all overlapping boxes, and then passes the results to the recognition stage.

No GPU acceleration is used for this stage of the recognition method since the stage is relatively short. The overhead of pushing the work for this stage on the GPU would outweigh any benefits of GPU acceleration.

V. 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. The goal of the recognition stage is to label each image patch with a unique identifier. These identifiers are used to distinguish between the different faces that appear in a particular video sequence.

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. When an image patch is provided as input to the recognition stage, each of the pattern models evaluates the image patch, as shown in Figure 2. The pattern model that provides the largest confidence value labels the image patch with its unique identifier.

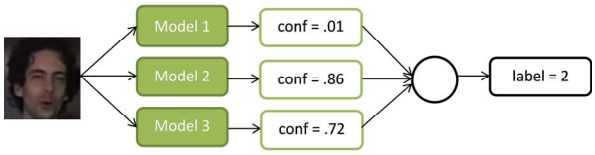


Fig. 2. Identifying a Particular Image Patch (Photo from [5])

However, we cannot evaluate each image patch individually because we cannot ensure that two image patches will not be labeled with the same identifier. Since we intend to use this method for facial recognition, we want to ensure that each face only appears at most once in a given image frame. This means we need to find the best combination of pattern models to assign to the image patches. We treat this assignment problem as a case of the classical *stable marriage* problem, and use the *Gale-Shapley algorithm* [6] to quickly find the best combination of pattern models to image patches.

For the pattern models we selected a pattern model that utilizes both nearest-neighbor classification and support vector machines (SVMs) [7] to identify faces that appear in the video sequences. It was designed to so that new models can be created during runtime to identify new faces that appear in the video sequence, as well as collect data during runtime to train and improve the accuracy of the model.

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 time of the pattern model used is small; if the model was pushed onto the GPU the overhead involved would only increase the processing time.

VI. Evaluation

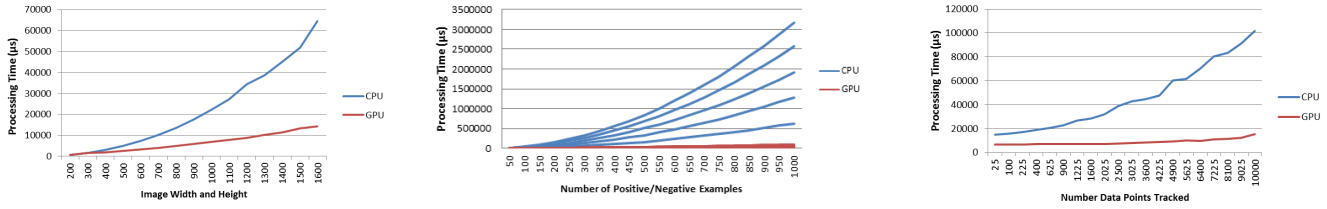
We now report the results of evaluating the recognition method. We first discuss the hardware and methodology used to evaluate this new GPU-based recognition technique. We then give the individual runtimes of the various GPU-accelerations discussed in this paper. Following this, we present the results of the overall recognition method when running the evaluation sequences.

A. Evaluation Methodology

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. 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.

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 [8] as the base for the tracking and detection stages. Both implementations use the OpenCV (Open Source Computer Vision) library during operation, including the hardware accelerations provided. For the tracking stage of the CPU implementation, OpenCV is used as well to implement the Lucas-Kanade tracking technique. GPU acceleration of the detection and tracking stages was accomplished using CUDA (Computer Unified Device Architecture) developed by NVIDIA. Memory is transferred between the CPU and GPU at the beginning and end of both stages. There is also a minimal amount of memory movement during the detection stage. To simplify the code used in the two stages, we took advantage of the *Thrust* parallel algorithms library [9].

We evaluate the two different implementations using several different criteria. To express the accuracy of the bounding boxes given by the tracking and detection stages, we use the common *precision* and *recall* metrics. In order to save space, the two metrics are combined into a single score value by using the harmonic mean of the two values, which we refer to as the *f-measure*. For the recognition stage, 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.



(a) (b) (c)

Fig.3. Individual Component Timings: (a) Integral Image Calculations, (b) Nearest-Neighbor Classifier, (c) Lucas-Kanade Tracking

We evaluate the two implementations of the recognition method using two different processing modes 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. If the processing time is too long, then a number of frames are skipped based on how long the duration was. This causes the processing speed to have a significant influence on the f-measure and recognition accuracy of the recognition method. Live streaming mode is meant to simulate running the recognition method working with a live feed rather than a pre-recorded video sequence. We simulated the video sequences at 25 frames per second (fps).

We evaluate the performance of the recognition method using two different datasets. The first of these datasets is the Hannah Dataset [10], 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]. Six scenes were selected which contain several common challenges to face tracking and identification, such as various face rotations and occlusions. The other dataset used for evaluation is the *Surveillance Performance EValuation Initiative* (SPEVI) [5] 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. We use two out of the three sequences from the dataset.

B. Individual Component Evaluations

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

For the integral image calculations, we timed how long the implementations took to generate the integral image using a number of different sized images. For each image size we run the calculations 100 times and take the average processing time. The timing results are presented in Figure 3a. 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.

We next review the performance of 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. It was found that the time needed to process the image frames for the GPU implementation is significantly smaller than the time for the CPU implementation. The average runtime for the CPU method was found to be 10350 µs, and for the GPU method the average runtime was 1230 µs. The speedup gained in using the GPU acceleration ranged from around 7x to 10x speedup, with an average speedup of approximately 8.39x.

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. For a specific number of examples and windows we take the average over 50 runs. The results of this test are presented in Figure 3b. 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 aparent 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 3b 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.

Last we investigated the performance of the Lucas Kanade tracking implementations. For both the CPU and GPU implementations we evaluate over the time taken to track data points and calculate the tracking error. 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 3c. 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 implementation than with the GPU. 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.

C. 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. For both sets of evaluations the values given are the average over multiple runs to ensure accurate results. The results for running the two implementations of the recognition method in offline mode are given in Table 1. In the interest of space we only give the average results of the two implementations for all video sequences.

TABLE I. RECOGNITION METHOD PERFORMANCE (OFFLINE MODE): CPU VS GPU

	f-measure	recognition	fps
CPU	0.874	0.821	10.31
GPU	0.880	0.841	35.72

We observed 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. It is to be expected that the f-measure and recognition for the two implementations differ to some degree in Table 1. This is due to differences in the architectures used (CPU vs GPU). Although the difference in precision for the two is very small (e.g 0.000002), this difference is significant enough that the performance of the recognition method can vary to some degree. To account for this, we find the percentage loss for the implementations between processing, and choose to compare them based on this loss rather than specific values.

The results of running the implementations in live streaming mode are shown in Table 2. 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 II. RECOGNITION METHOD PERFORMANCE (LIVE STREAMING MODE): CPU VS GPU

	f-measure	recognition	frames lost
CPU	0.716	0.769	2.29
GPU	0.830	0.794	0.75

Based on the results from Table 1 and Table 2, we can find the percentage loss when running the implementations using live streaming mode. For the CPU implementation the f-measure loss is 22% and the recognition loss is 7%. For the GPU implementation the percent loss for both measures is 6%.

We observe that 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 detected in the evaluation sequences.

VII. 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. 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.

With regard to our recognition method, there may be additional opportunities to accelerate our method on a GPU. Additional future work also includes introducing *dynamic parallelism* to the method, which allows for GPU threads to 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.

ACKNOWLEDGMENT

The research was supported by the Applied Research Laboratory at the Pennsylvania State University.

REFERENCES

[1] Z. Kalal, "Tracking Learning Detection," Center for Vision, Speech and Signal Processing, Faculty of Engineering and Physical Sciences, University of Surrey, 2011.

[2] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in Conference on Computer Vision and Pattern Recognition, 2001.

[3] B. Bilgic, B. K. Horn and I. Masaki, "Efficient Integral Image Computation on the GPU," in IEEE Intelligent Vehicle Symposium, 2010.

[4] J.-Y. Bouguet, "Pyramidal Implementation of the Lucas Kanade Feature Tracker- Description of the Algorithm," Microprocessor Research Labs, Intel Corporation, 2000.

[5] 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 Processing, Honolulu, 2007.

[6] D. Gale and L. S. Shapley, "College Admissions and the Stability of Marriage," The American Mathematical Monthly, vol. 69, pp. 9-15, 1962.

[7] C. Cortes and V. Vapnik, "Support-Vector Networks," AT&T Labs-Research, 1995.

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

[9] J. Hoberock and N. Bell, "Thrust Parallel Algorithms Library," [Online]. Available: http://thrust.github.io. [Accessed February 2014].

[10] A. Ozerov, J.-R. Vigouroux, L. Chevallier and P. Pérez, "On evaluating face tracks in movies," in IEEE International Conference on Image Processing, 2013.

[11] W. Allen, Director, Hannah and Her Sisters. [Film]. United States: Robert Greenhut, 1986