Towards a Robust, Real-time Face Processing System using CUDA-enabled GPUs

Bharatkumar Sharma, Rahul Thota, Naga Vydyanathan, and Amit Kale
Siemens Corporate Technology
SISL - Bangalore, India
{*bharatkumar.sharma, rahul.thota, nagavijayalakshmi.vydyanathan, kale.amit*}*@siemens.com*

***Abstract*—Processing of human faces finds application in various domains like law enforcement and surveillance, entertainment (interactive video games), information security, smart cards etc. Several of these applications are interactive and require reliable and fast face processing. A generic face processing system may comprise of face detection, recognition, tracking and rendering. In this paper, we develop a GPU accelerated real-time and robust face processing system that does face detection and tracking. Face detection is done by adapting the Viola and Jones algorithm that is based on the Adaboost learning system. For robust tracking of faces across real-life illumination conditions, we leverage the algorithm proposed by Thota and others, that combines the strengths of Adaboost and an image based parametric illumination model. We design and develop optimized parallel implementations of these algorithms on graphics processors using the Compute Unified Device Architecture (CUDA), a C-based programming model from NVIDIA. We evaluate our face processing system using both static image databases as well as using live frames captured from a firewire camera under realistic conditions. Our experimental results indicate that our parallel face detector and tracker achieve much greater detection speeds as compared to existing work, while maintaining accuracy. We also demonstrate that our tracking system is robust to extreme illumination conditions.**

***Index Terms*—Parallel computing, face detection, face tracking, graphics processors, real-time algorithms.**

I. Introduction

Machine perception of human faces is an active research topic in disciplines like image processing, pattern recognition, computer vision and psychology. The interest in facial information acquired from images stems from the fact that it can be extracted non intrusively and provides a valuable cue in several commercial, security and surveillance systems. Furthermore detection and recognition of faces is also increasingly used in interactive video games, virtual reality applications, video-conferencing etc. A common trend in these applications is their interactivity and time critical nature, which makes it important to develop image analysis algorithms that can meet real-time constraints. In addition, the algorithms should also be cognizant of the advent of technologically advanced data capturing devices like smart cameras and the vast amounts of data that they generate.

A typical face image analysis system comprises of face detection, recognition, tracking and rendering. Face detection is used to distinguish faces from the background. Distinct faces can then be tagged and subsequently recognized across disparate locations. Alternatively it would also be useful to track a person in a single camera view to find out to which camera the identity of the face should be handed off. Such information is critical to develop situational awareness in a secure establishment. In order to be useful, these three components - face detection, recognition and tracking, should work at speeds close to frame rates. This is especially critical in a multi-camera scenario where multiple video streams have to be processed simultaneously. Out of the three components, face detection and recognition algorithms are computationally intensive but fortunately are amenable to parallelization.

Recent years have seen the emergence of graphics processing units (GPUs) as sources of massive computing power that can be harnessed for general purpose computations. For example, the NVIDIA GeForce GTX 280 graphics card has a peak computing rate of 933 Gflops. The Compute Unified Device Architecture (CUDA) [1] is a C-based programming model proposed by NVIDIA, that exposes the parallel computing capabilities of the GPU to application developers in a easy to use manner. CUDA facilitates programming general purpose computations on the GPU without requiring developers to remap algorithms to graphics concepts. It also exposes a fast shared memory region that can be accessed by blocks of threads (explained in Section IV), allows for scattered reads and writes, and optimizes data transfers to and from the GPU.

In this paper, we leverage the compute capabilities of modern GPUs for accelerating a face processing system that does face detection and tracking. Face detection is done by adapting the Viola and Jones algorithm [2] based on Adaboost. For robust tracking of faces across real-life illumination conditions, we leverage the algorithm proposed by Thota et al. [3], that combines the strengths of Adaboost and an image based parametric illumination model. We design and develop optimized parallel implementations of these algorithms on graphics processors using CUDA.

We evaluate our face image processing system using both static image databases as well as using live frames captured from a firewire camera under realistic conditions. Our experimental results indicate that our parallel face detector and tracker achieve much greater detection speeds as compared to existing work, while maintaining accuracy. We also demonstrate that our tracking system is robust to changing illumination conditions.

The rest of this paper is organized as follows. The next section gives an overview of the related work. Section III

describes the Adaboost-based face detection and how we adapt it to suit our face processing system. It also described the robust face tracking approach that we use. Section IV gives a brief overview of the graphics processing hardware and CUDA. Section V presents our CUDA accelerated parallel face processing system. Section VI presents our experimental results and Section VII concludes the paper and outlines directions for future research.

II. Related Work

Machine perception and processing of human faces has been a widely researched topic in the domains of image processing and computer vision. Since the late ninetees, researchers have proposed several algorithms for detecting human faces [4], [5], [6], [7], [2]. Out of these, the Viola and Jones [2] face detection algorithm, which is based on the Adaboost learning system has been shown to detect faces faster (15 fps for 320×288 images) than previous approaches, while maintaining the accuracy of detection even at low resolutions.

With the advent of high speed internet access, a number of novel additional features such as awareness of the identity and location of the participants can be included as a part of a video conferencing system, which can even be extended to mobile platforms. Earlier the incorporation of such features which involve face detection and recognition were limited by the available computational resources. However with the proliferation of low cost high performance computational devices, many researchers have begun to explore the usage of these features. In addition, technological advancements that result in data capturing devices that operate at high frame rates, has triggered the need for real-time face processing systems.

Paschalakis and Bober [8] proposed a fast face detection and tracking system for mobile phones which uses skin color information for face detection. Xu and Sugimoto [9] also modeled skin color and used the track to control a PTZ camera to improve the field of view. Their system yielded detection and tracking rates of 5 and 10 fps. A recent work by Lozano and Otsuka [10] proposed a particle filtering based method for 3D tracking of faces using GPUs. However, particle filters rely on a motion model on shape space and in realistic surveillance videos, objects do not always obey the motion model. Further, face detection could become a bottleneck especially while simultaneously processing frames from multiple cameras. Yet another work by Ghorayeb et.al [11] describes a hybrid scheme that does face detection on CPU and GPU using Brooks API which is built on top of OpenGL. They achieve a frame rate of 15 fps for 415×255 sized frames.

Though the above works aim to improve the speed of face detection and tracking, we need much faster implementations to scale to current real-time requirements. Furthermore, most of these approaches rely on the availability of high resolution color face images. In surveillance type scenarios resolution of the face may go as low as 15×15 which makes the usage of a 3D model [10] infeasible. Illumination in such scenarios can also be erratic which can cause methods based on simple skin histograms such as [9] [8] to fail.

With the introduction of the CUDA programming model, it is possible to design better optimized implementations of face detection and tracking on GPUs that achieve much higher detection/tracking rates. This is because, as against OpenGL implementations [11], CUDA has several advantages. CUDA exposes a fast shared memory region that can be accessed by blocks of threads (explained in Section IV), allows for scattered reads and writes, optimizes data transfers to and from the GPU, and makes it easy to program general purpose computations on the GPU. In this work, we design and implement a highly optimized parallel face detection and tracking system using CUDA on GPUs. Our face processing system achieves much higher detection rates than previously proposed solutions. Also to the best of our knowledge, this is the first attempt to address the problem of face tracking in low resolution images across illumination changes using GPU.

III. Face Processing

A. Adaboost-based Face Detection

Viola and Jones [2] pioneered the use of Adaboost [12] for fast and accurate face detection. The algorithm uses Adaboost to select the most discriminative features out of an overcomplete set of Haar features, to separate faces and non-faces.

Four kinds of rectangular haar features shown in the Figure 1 are used in the detection algorithm. In each case the feature value is given by the difference between the sum of pixel intensities in the bright regions and the sum of pixel intensities in the dark regions. In a detection test window of size 24×24 pixels we can overlay more than 10^5 such rectangular features at different locations and sizes. This overcomplete feature set gives us an elaborate description of the image structure in the window considered.

To speed up this feature computation, Viola and Jones proposed the use of the integral image. The integral image at location x, y contains the sum of the pixels above and to the left of x, y

$$ii(x,y) = \sum_{x' \leq x, y' \leq y} i(x', y') \tag{1}$$

Using the integral image any rectangular sum can be computed in four array references. It is easy to see that the two-rectangle features defined above can be computed in six array references, eight in the case of a three-rectangle feature, and nine for four-rectangle feature. A simple way to construct "weak" classifiers from each of these features is by thresholding them

$$h(U) = \begin{cases} 1 & if\ pf(U) < p\theta \\ 0 & otherwise \end{cases} \tag{2}$$

where f is the feature thresholded at θ, p is the polarity indicating the direction of the inequality and U is the 24×24 pixel test sub-window considered. Adaboost training selects the best classifiers $h_t(U)$ (feature/threshold pairs with the parity) which collectively minimize the classification error. A strong classifier output score is the weighted combination of these weak classifiers $H(U) = \sum_{t=1}^{T} \alpha_t h_t(U)$ where the

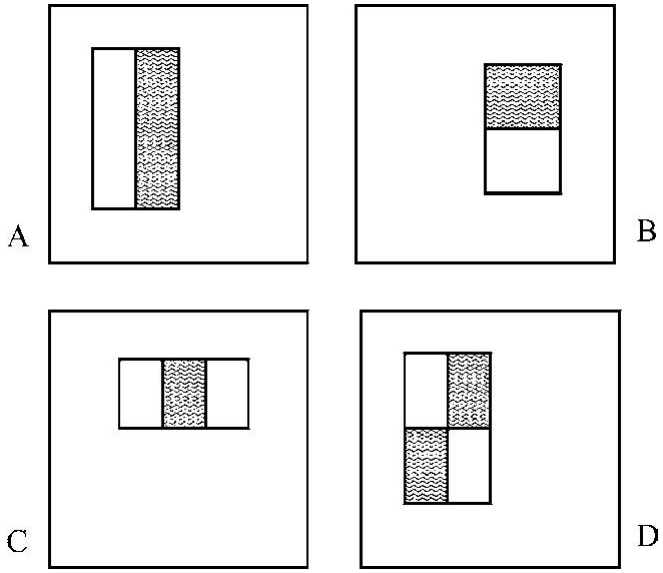


Fig. 1. Example rectangle features shown relative to the enclosing detection window. The sum of the pixels which lie within the white rectangles are subtracted from the sum of pixels in the grey rectangles. Two-rectangle features are shown in (A) and (B). Figure C) shows a three-rectangle feature, and (D) a four-rectangle feature.

normalized weight α_t of the weak classifier $h_t(U)$ is inversely proportional to its classification error. This strong classifier score $H(U)$ gives a direct estimate of the probability of the 24×24 pixel sub-window U containing a face.

Since the base resolution of the strong classifier has been trained for the size 24×24, to detect an arbitrarily sized face in a given image we either need to scale up the features to the size of the sub-windows or scale down the sub-windows to the base resolution. Though scaling the features is a faster approach (used in [2]) it lacks accuracy, so we resort to scaling the sub-windows to 24×24. This can be efficiently done by forming a pyramid of images at different scales and we sample these using 24×24 sub-windows.

Typically a 640×480 image corresponds to a pyramid of images at 29 different scales. If these images at different scales are sampled in x and y directions with step sizes of 4 and 2 pixels respectively we will have to apply the strong classifier on 4.2×10^6 sub-windows of size 24×24. In order to overcome this huge computational burden Viola and Jones proposed a cascade architecture shown in Figure 2where a series of strong classifiers are used instead of just one. While the accuracy of a strong classifier increases with the number of weak-classifiers combined to generate it, this also increases the corresponding computation time. The cascade uses simple strong classifiers with possibly high number of false positives but very low false negatives, followed by progressively more complicated strong classifiers. This allows for quickly rejecting a large number of non-face regions while passing on putative face regions to the next stages of the cascade for further evaluations. This greatly reduces the average number of calculations performed for each subwindow. In our implementation we use a 12 stage cascade with 1800 distinct weak classifiers at an optimal trade-off giving a very good accuracy with acceptable compute time.

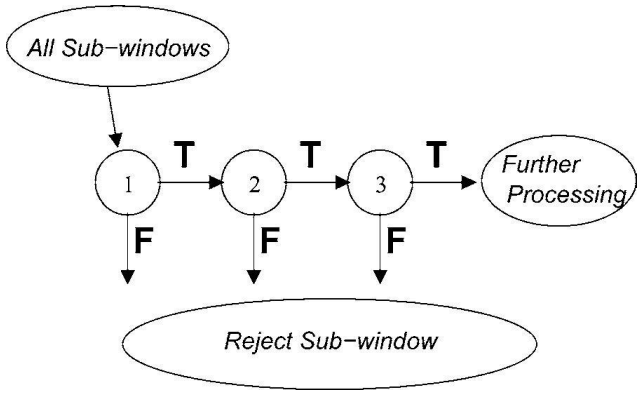


Fig. 2. Schematic depiction of a the detection cascade. A series of classifiers are applied to every sub-window. The initial classifier eliminates a large number of negative examples with very little processing. Subsequent layers eliminate additional negatives but require additional computation. After several stages of processing the number of sub-windows have been reduced radically.

B. Unconstrained Face Tracking

In a streaming video detection of a new face marks the entry of a new person into the field of view (FOV) of the camera. Once the new person is detected he/she should be tracked through the path in the FOV. Appearance change of the object is a big challenge for tracking which may arise due to various reasons like illumination and or pose changes. The standard Adaboost face detector handles uniform illumination changes using variance normalization. However such a normalization cannot handle uneven illumination conditions which often occur in reality. Recently Thota et al [3] proposed a fusion of Kale and Jaynes [13] illumination compensation with the basic Adaboost detection score. We discuss their method briefly here: The image template T_t in the tth frame of the tracking sequence can be expressed as:

$$T_t(x,y) = L_t(x,y)R(x,y) = \tilde{L}_t(x,y)T_0(x,y) \qquad (3)$$

where $L_t(x,y)$ denotes the illumination image in frame t and $R(x,y)$ denotes a fixed reflectance image [14]. In absence of knowledge of R, the problem of estimating the illumination image reduces to estimating $\tilde{L}_t$ w.r.t to the illumination contained in the image template $T_0 = L_0 . * R$. Kale and Jaynes [13] model $\tilde{L}_t$ as a linear combination of a set of N_Λ Legendre basis functions. Denoting $p_k(x)$ as the k th Legendre basis function for $N_\Lambda = 2k+1$, $\Lambda = [\lambda_0, \cdots, \lambda_{N_\Lambda}]^T$, the scaled intensity value at a pixel of the template T_t is computed as:$T_t(x,y) = T_0(x,y) + T_0(x,y)\mathbf{P}(x,y)\Lambda$ where

$$\mathbf{P}(x,y) = \frac{1}{2k+1}[1 \; p_1(x) \cdots p_k(x) \; p_1(y) \cdots p_k(y)] \qquad (4)$$

Rewriting T_0 and T_t as vectors we get $[T_t]_{vec} = [T_0]_{vec} + [T_0]_{vec} \otimes \mathbf{P}\Lambda$ so that when $\Lambda \equiv 0, T_t = T_0$. Operator $\otimes$ refers to multiplying each row of $\mathbf{P}$ by $T_0(x,y)$. Given T_t and T_0,

the Legendre coefficients that relight T_t to resemble T_0 can be computed by solving the least squares problem:

$$A_{T_t}\Lambda_t \approx [T_t - T_0]_{vec} \tag{5}$$

where

$$A_{T_t} = [T_t]_{vec} \otimes \mathbf{P} \tag{6}$$

so that $A_{T_t} \in \mathbf{R}^{N_\Lambda + M}$ Given the ground truth of template locations in successive frames (5) can be used to find the illumination coefficients $\{\Lambda_1, \cdots, \Lambda_N\}$. Although the underlying distribution of these Λ_ts is continuous [13] shows that much of this information can be condensed down to a discrete number of important illumination modes or centroids $\{c_1, \cdots, c_k\}$ via $k-$means clustering, without sacrificing tracking accuracy. For example if a corridor has predominantly three different lighting conditions, we find that the Legendre coefficient vectors also cluster into three groups.

The problem of tracking a face can be broken down to finding the best estimate $\hat{U}_{t+1}$ of the target in the $(t+1)$th frame given the previous estimate U_t. [3] proposes to exhaustively sample subwindows $U_{k,t}$ in a reasonably sized region around the previous location (at multiple scales) and pick the candidate among them which maximizes the following likelihood criterion:

$$L(U_{k,t}, T_0) = exp^{-d^2_{illum}(U_{k,t}, T_0)} I(H(U_{k,t}) > \tau_{min}) \tag{7}$$

where $I(.)$ is an indicator function. It is clear that d_{illum} needs to be evaluated only on a subset of windows for which the Adaboost detection score $H(U_{k,t})$ exceeds a lower bound τ_{min} learnt from the ground truth. d_{illum} is the minimum of the sum of absolute difference (SAD) distances between the stored template of the person T_0 (assumed to be the detected face sample in the first image) and the relighted window $U_t + U_t \otimes \mathbf{P}c_k$ which is illumination compensated using each of the centroids c_k.

$$d_{illum}(U_t, T_0) = \min_{\{c_1, \cdots, c_k\}} d(T_0, U_t + U_t \otimes \mathbf{P}c_k) \tag{8}$$

The algorithm is inherently parallel since the set of windows $U_{k,t}$ around the previous estimate U_t can be processed independently. Firstly, all these subwindows in the ROI at different scales are resized to 24×24 to calculate the detection score on each of them. Only those which have high enough score value are further investigated using illumination compensation. This greatly improves the speed without any compromise over the tracking accuracy.

IV. Overview of GPU Architecture and CUDA Programming Model

The graphics processor with its massively parallel architecture is a storehouse of tremendous computing power. The Compute Unified Device Architecture (CUDA) [1] is a C-based programming model from NVIDIA that exposes the parallel capabilities of the GPU for easy development and deployment of general purpose computations. In this section, we briefly describe the salient features of the GPU architecture and CUDA programming model.

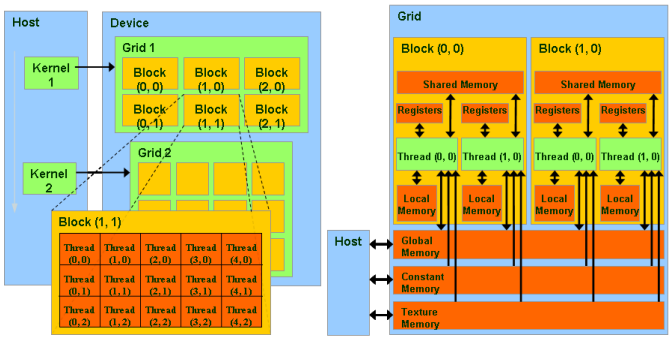


Fig. 3. CUDA Programming and Memory Model. (Courtesy NVIDIA)

The GPU is a data-parallel computing device consisting of a set of multiprocessing units, each of which is a set of SIMD (single instruction multiple data) processing cores. For example, the Quadro FX 5600 GPU has 16 multiple processing units, each having 8 SIMD cores, resulting in a total of 128 cores. Each multiprocessor has a fixed number of registers and a fast on-chip memory that is shared among its SIMD cores. The different multiprocessors share a slower off-chip device memory. Constant memory and texture memory are read-only regions of the device memory and accesses to these regions are cached. Local and global memory refers to read-write regions of the device memory and its accesses are not cached.

Figure 3 shows the memory and programming model of CUDA-enabled GPUs. The programming model of CUDA offers the GPU as data-parallel co-processor to the CPU. In the CUDA context, the GPU is called device, whereas the CPU is called host. Kernel refers to an application that is executed on the GPU device. A CUDA kernel is launched on the GPU device as a grid of thread blocks that are made up of threads. Thread blocks can span in one, two or 3 dimensions and grids can span in one or two dimensions (Figure 3 shows 2-D thread blocks and grids). Threads are uniquely identified based on their thread block and thread indices. A thread block is executed on one of the multiprocessors and multiple thread blocks can be run on the same multiprocessor. Consecutive threads of increasing thread indices in a thread block are grouped into what are known as warps which is the smallest unit in which the threads are scheduled and executed on a multiprocessor.

V. CUDA Accelerated Face Processing

This section presents our optimized parallel implementation of face detection and tracking algorithms using CUDA.

A. CUDA-based Parallel Face Detection

As described in Section III-A, face detection comprises of three main steps: 1) resizing of the original image into a pyramid of images at different scales 2) calculating the integral images for fast feature evaluation, and 3) detecting faces using a cascade of classifiers. Each of these tasks is parallelized and run as kernels on the GPU, as shown in Figure 4. If we have a system with multiple graphics cards, it is possible to pipeline

the frames through these three kernels, with the resizing of the $(n+2)^{th}$ frame being done in parallel with the integral image calculation of the $(n+1)^{th}$ frame and the face detection of the n^{th} frame.

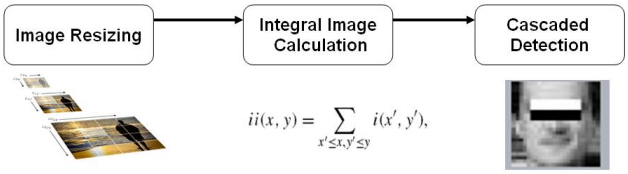


Fig. 4. Pipeline of Face Detection Kernels.

In the next sub-sections, we describe each of these three kernels in detail.

1) Image resizing: In this step, the original image is resized to a pyramid of images at different scales, the bottom of the pyramid being the original image, and the top, a scaled down image at 24×24 resolution, which is the base resolution of the detector. The height of the pyramid, or in other words, the number of resized images depends upon the scaling factor which is 1.1 in our case. This means that starting from the original image at the bottom of the pyramid, each successive image is smaller than the previous by a factor of 1.1.

Computation of the pyramid of images, though straightforward, requires significant time. A simple approach for parallel image resizing is by allowing different CUDA thread blocks to compute images at different scales in parallel. Each thread in a thread block, computes the value of a pixel in an image scale. However, since CUDA thread blocks have fixed dimensions, as the image dimensions progressively decrease, larger number of threads are rendered inactive in this approach.

Alternatively, we can view the pyramid of images as a one-dimensional contiguous block of image data as shown in Figure 5. We also create a one-dimensional grid of 1D thread blocks. Total number of threads spawned will be equal to the number of thread blocks $\times$ the number of threads within a block. Each of these threads computes the value of one or more pixels in the 1D block of resized image data in a cyclic fashion as depicted in Figure 5.

Such a parallel design has several advantages:

- first, it ensures that all threads (except may be at the fag end of the 1D image data) are all active. Hence the processing cores of the GPU are kept busy,
- second, as successive threads read successive memory locations, memory accesses by threads in a half-warp are coalesced into a single memory transaction in CUDA, thereby resulting in efficient usage of global memory bandwidth.

In addition, we store the original image in CUDA textures, which is cached. Hence, when threads of the same warp access nearby pixels in the image data, spatial locality is exploited and data is fetched from the cache, resulting in improved performance. All other control information such as resized image dimensions are pre-calculated by the threads of a thread block in parallel and stored in shared memory for subsequent retrieval. As shared memory is on-chip and fast and multiple threads reading the same data from shared memory results in a broadcast of information rather than multiple reads, performance is further enhanced.

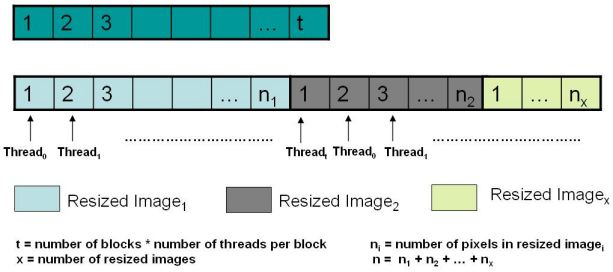


Fig. 5. One dimensional thread and image view.

2) Integral image calculation: The next step after image resizing is the calculation of the integral image for each resized image. The integral image calculation is computationally expensive especially for large images. Computing the integral image of a $N \times N$ image requires $2 \times N^2$ operations. To parallelize this calculation, we break down the integral computation into a horizontal prefix sum followed by a vertical prefix sum on the image data. These two steps are implemented as two kernels. A 1D view of the image and threads is used similar to the one described in Section V-A1. For horizontal prefix sum computation, threads compute the prefix sum of different rows of the resized images in parallel. For example, in Figure 6(A), thread 0 computes prefix sum for row 0 of the first resized image, while thread 1 computes prefix sum for row 1. Though the data is stored as a 1D block, for clarity, we depict the prefix sum computations on 2D representations of images. Each thread adds to the value of the current pixel, the previous pixel value along that row, and moves forward till the last pixel in the row is reached. The pixel values are fetched from textures and hence results in high performance. The number of threads active at a time is the minimum of the total number of threads spawned and the summed height of all the resized images.

The vertical prefix sum computation is done on the output of the horizontal prefix sum computation. It is similar to the horizontal prefix sum, except that threads compute column prefix sums in parallel as shown in Figure 6(B). Each thread adds to the value of the current pixel, the previous pixel value along that column, and moves forward till the last pixel in the column is reached. The number of threads active at a time is the minimum of the total number of threads spawned and the summed width of all the resized images.

3) Cascaded detection: Cascaded detection, as explained in Section III-A, applies a cascade of classifiers to sub-windows (since our base resolution of the detector is 24×24, we use 24×24 sub-windows) at different locations of the image at different scales and computes a score for each sub-window. Sub-windows that pass through all cascades (i.e the scores are above a certain threshold) are classified as faces.

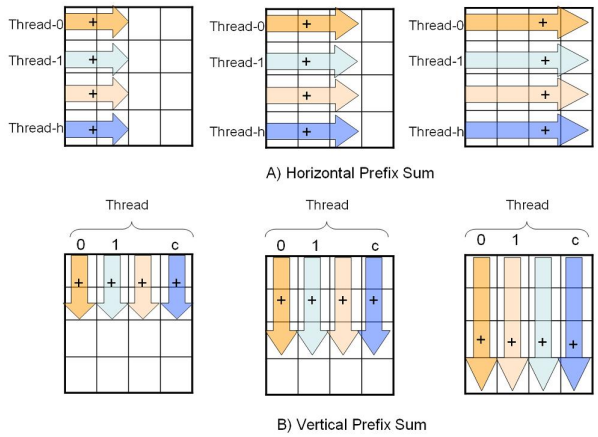


Fig. 6. Integral Image Calculation: horizontal and vertical prefix sum.

We parallelize the cascaded detection process, by allowing the simultaneous computation of the feature values and scores for sub-windows at different locations of the image at different scales in parallel by multiple threads. This is depicted in Figure 7, where two threads are shown, thread 0 and thread 1, which extract sub-windows at different locations and compute the score. The sub-windows are extracted with a step size of 4 pixels along the width and 2 pixels along the height. For fast feature evaluation, the integral images computed previously are used. Both the integral images and the features are stored and retrieved from textures to enhance performance. The cascades are initially stored in textures and transferred to shared memory for faster access.

In the next section, we present our CUDA-based parallel face tracking approach that is robust to real-life illumination conditions.

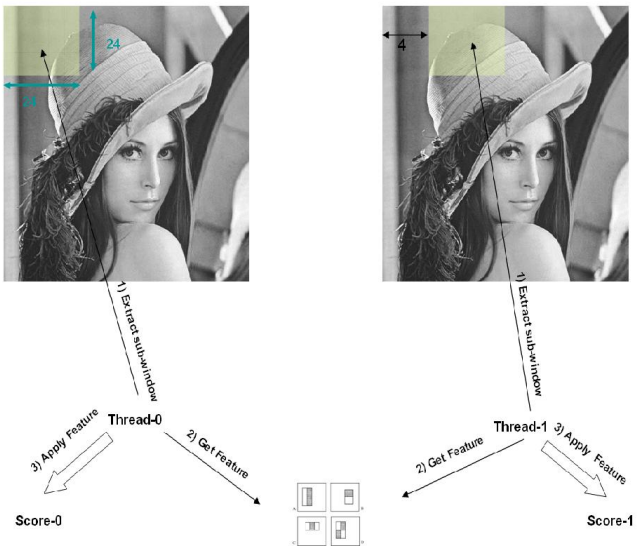


Fig. 7. Cascaded Detection.

B. CUDA-based Unconstrained Face Tracking

Once a new face is detected in a frame of a video sequence, it is tracked through subsequent frames. Face tracking, as explained in Section III-B consists of the following steps:

- extracting sub-windows in the image in the vicinity of the previous location of the detected face,
- resizing the sub-windows to 24×24 resolution,
- calculating the integral images of the resized sub-windows,
- applying Adaboost-based face detection,
- applying illumination compensation to the detected sub-windows, and
- computing the next location of the face using template matching.

Each of the above steps are implemented as CUDA kernels as described below.

1) Extracting sub-windows in the neighborhood of previous face location: Instead of applying features over the whole image, only the region near the previous location of the detected face is searched for the new location, as shown in Figure 8. Sub-windows of different scales are extracted from different pixel locations in the neighborhood of the previous location of the detected face. The scales are determined based on the previously tracked/detected face dimensions, for example sub-windows are extracted at resolutions ranging from 0.9 to 1.2 times the previous face dimensions. Each of these sub-windows is then resized to 24×24 resolution. The parallel extraction of sub-windows at different locations and scales and resizing of these to 24×24 resolution is implemented in a similar manner as the image resizing kernel described in Section V-A1. As explained previously, such a one-dimensional view of both the image data and threads ensures that the GPU cores are kept busy and global memory bandwidth is used efficiently.

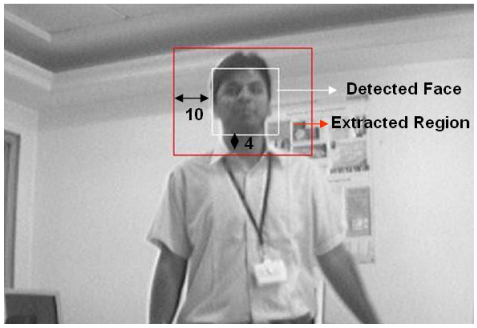


Fig. 8. Extracting sub-windows in the neighborhood of previous face location.

2) Calculating the integral image and Adaboost-based face detection: The calculation of the integral images of each of the resized sub-windows is done similar to the integral image calculation kernel presented in Section V-A2. The Adaboost score for each of the sub-windows is then computed. For tracking, we use a threshold which is much lower as compared to that used in face detection. This threshold is computed

using training data under bad illumination conditions (as explained in [3]). The detected sub-windows are then stored in CUDA textures and accessed by subsequent kernels that do illumination compensation and template matching.

3) Illumination compensation and template matching: In this step, each of the sub-windows that are classified as faces using the Adaboost detector, are re-lighted using the precomputed illumination centroids (described in Section III-B which depend upon the scene lighting conditions. The re-lighting of each pixel in the detected sub-windows is done in parallel by multiple threads. Both the detected sub-windows and the illumination coefficients computed based on the lighting conditions, are accessed from textures to enhance performance by exploiting spatial locality. Once the detected sub-windows are re-lighted, their SAD distance from the template of the detected face is computed in parallel. The re-lighted sub-window with the least SAD distance is output as the next location of the face.

To summarize, in this section, we present an optimized parallel design for a robust and fast face detection and tracking system. Our parallel implementation ensures:

- high processor occupancy,
- efficient usage of global and shared memory bandwidths, and
- intelligent exploitation of spatial locality through use of textures.

In addition, we use instructions that take lesser number of clock cycles such as $\_\_mul24$, the fast 24 bit integer multiplication and $\_\_fdividef$. Also, data such as feature information and classifiers are re-organized and stored in textures using vector integer and floating point data types so as to enhance spatial locality.

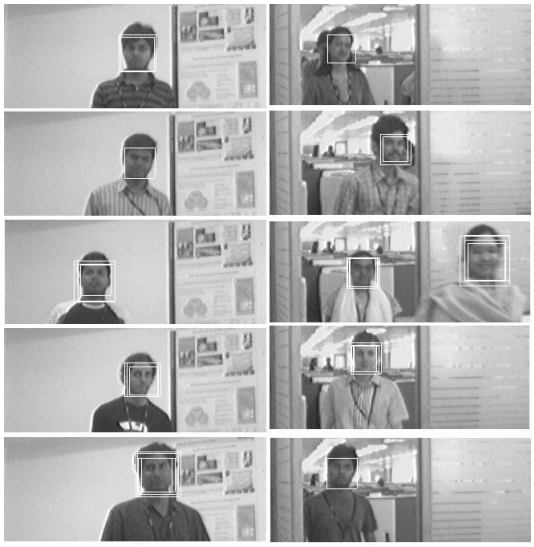
VI. Performance Analysis

The developed face detection and tracking software (a mixture of C++ and CUDA) has been tested on an Intel(R)Xeon(R) CPU, 3.33 GHz host system with 3.25 GB RAM, having a NVIDIA GeForce GTX 285 GPU. This GPU features 30 multiprocessors, 16 KB shared memory per multiprocessor and 1 GB device memory. There can be a maximum of 512 threads per block and 1024 active threads per multiprocessor. For comparison purposes, a CPU-only version of the face detection and tracking algorithm was also developed (single-threaded) for execution on the host CPU.

A. CUDA-based Parallel Face Detection

We evaluated our parallel face detector under two different scenarios: a) using a static image database and b) using live frames captured from a camera. These are described in detail below.

1) CMU Test Set: The CMU test set, collected as part of CMU Face Detection Project, is used for evaluating algorithms that detect frontal and profile views of human faces. 42 frontal face images having 165 frontal faces were chosen and used for assessing the accuracy and speed of our face detector. The results of our detector on CMU dataset are shown in Figure 9.



Results for the live frames	
Frame Size	640×480
Detection Speed	45 frames/sec
Detection Accuracy	100%
Number of False Positives	22

Fig. 10. Performance of our face detector on live frames taken in (A) company lab setup, and (B) company corridor. Both setups do not contain a plain background, but have many variations and other objects, representing a real world scenario.

We can notice that in many of the images, the faces are not well illuminated and some of the faces are not purely frontal. Due to these reasons, 100% detection accuracy is not achieved. Our numbers for the detection accuracy and false positives correlates with that quoted by Viola and Jones [2]. As the CMU test set contains images at different sizes, we report the detection speed in terms of number of pixels processed per second.

2) Live frames: Live frames were captured from a camera and given as a input to our face detector in real time. The camera used was Fire-i Board version monochrome model, with M12x0.5 lens base. The live frames were taken under realistic conditions in a company lab environment (Figure 10(A)), and a company corridor where groups of people walk in continuously(Figure 10(B)). In total, about 370 frames were processed.

Figure 10 shows the performance results of our face detector for the live frames. Our detector achieves 100% detection accuracy and real-time processing rates, even in the presence of complex backgrounds.

Figure 11 shows the performance comparison between CPU only version and our GPU enabled face detector. Images at various scales (320×240, 640×480 and 1280×960) were taken from the CMU dataset and live frames from the camera. The results show the average detection speed(fps) at various scales. We can see that our CUDA based implementation is 12 to 38 times faster than the CPU version and scales much better even at higher resolutions.

CMU Test Set Results	
Detection Speed	9.09 Million Pixels/sec
Detection Accuracy	81%
Number of False Positives	16

Fig. 9. Performance our face detector on images from the CMU test set.

B. Parallel Unconstrained Face Tracking

In order to test the robustness of our CUDA based face tracking system, we set up a camera facing a corridor in our establishment to monitor the flow of people. Input to the face tracker were frames captured continuously from a firewire camera under various illumination conditions(indoor and outdoor). For the indoor scenario, we found that the Legendre coefficients (explained in Section III-B), cluster into three groups, representing three distinct illumination conditions, as can be seen in Figure 12. The results of the face tracker with the calculated three illumination modes/centroids in the indoor scenario is shown in Figure 12. Face tracker results in a outdoor scenario under 2 different illumination modes is shown in Figure 13.

The tracking rate expressed in terms of frames per second(fps) of our CUDA based implementation as compared to CPU and DSP implementations (the CPU and DSP tracking rates are those quoted by Thota et.al [3]) are as shown in the Figure 14.

The CUDA implementation implemented is optimized with negligible loss in precision to obtain a performance of 150 frames per second for the case of one face environment and can be extended to simultaneously track multiple faces without much loss of performance. Such a high frame rate can facilitate tracking of multiple camera feeds in parallel.

The results indicate an important speed boost compared to the CPU-only version of the algorithm, making the face detector and tracker eminently suitable for real-time processing. Our CUDA-based parallel face detector and tracker also outperforms previous implementations discussed in Section II.

VII. Conclusions

This paper presents a GPU-accelerated real-time and robust face detection and tracking system. Our optimized face detection and tracking system is implemented on the GPU using CUDA, a C-based, easy to use, programming model proposed by NVIDIA. Face detection is done by adapting the Viola and Jones algorithm that is based on Adaboost. For robust tracking of faces in real-life illumination conditions, we

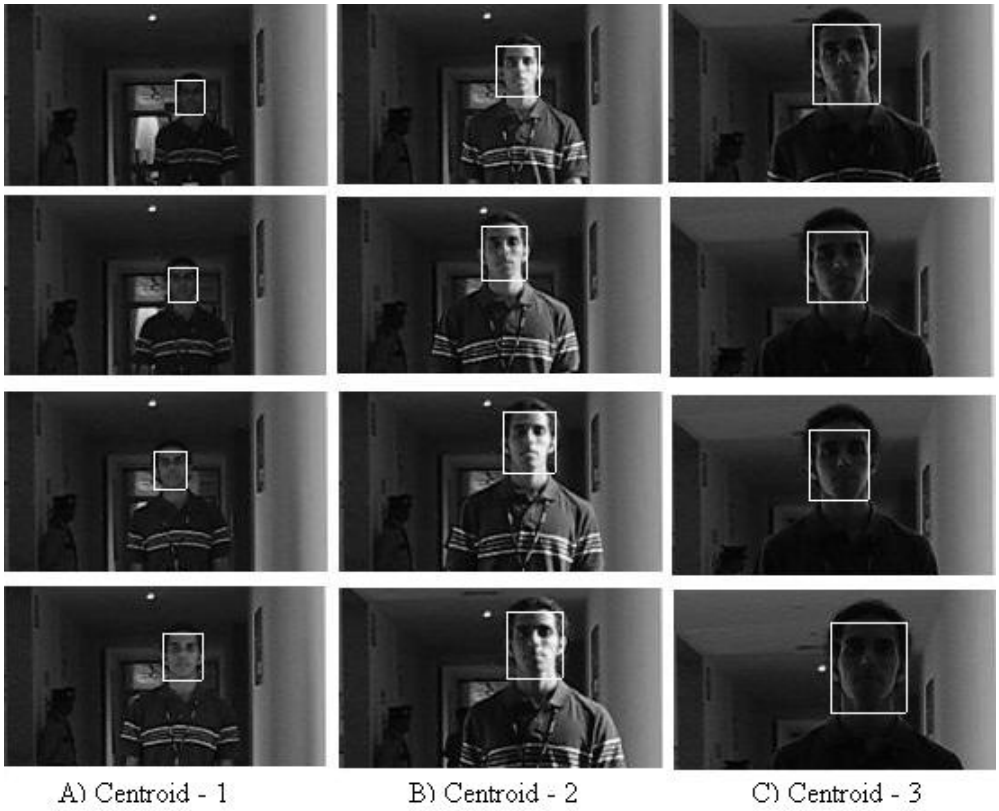
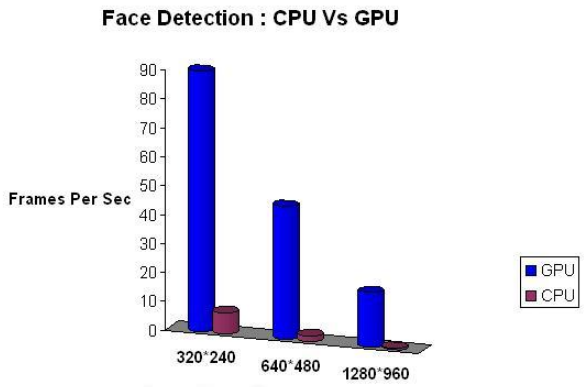


Fig. 12. Face Tracker output under 3 extreme illumination conditions represented by 3 centroids in an indoor environment



	320×240	640×480	1280×960
GPU	90	46	19
CPU	7.4	2.15	0.5

Fig. 11. Figure shows the detection rate in frames per second (fps) of CPU only version(CPU) face detector and our GPU enabled face detector(GPU) for images at various resolutions i.e. 320×240, 640×480 and 1280×960.

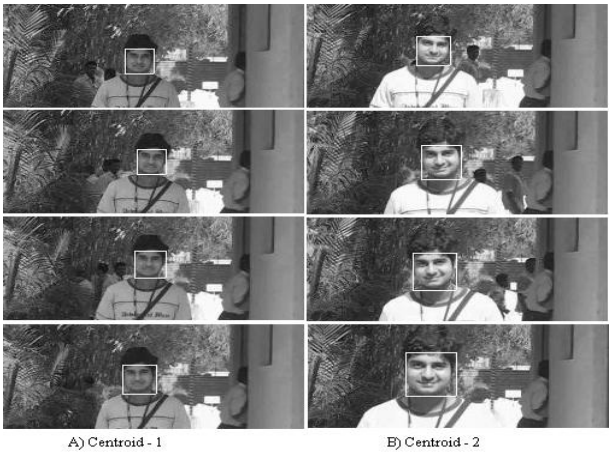
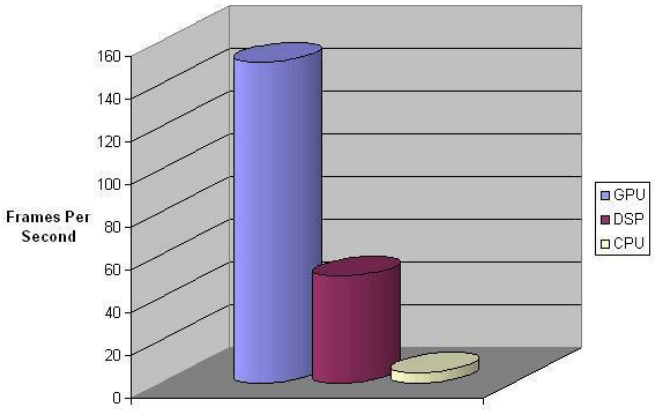


Fig. 13. Face Tracker output under 2 illumination modes/centroids in an outdoor environment. As it can be seen, Centroid-1 corresponds to a shadowed environment while Centroid-2 corresponds to a sun-lit environment.

adopt the algorithm proposed by Thota [3], which combines Adaboost with an image based parametric illumination model. Evaluations using both static image databases and live frames captured from a camera under realistic conditions, indicate that our parallel face detector and tracker achieve much greater detection speeds as compared to existing work, while maintaining accuracy. We also demonstrate that our tracking system is robust to extreme illumination conditions.

In future, we plan to extend the concepts discussed in this paper to face recognition in wide area surveillance networks using GPUs. We are experimenting with application of super resolution to the tracked faces to improve accuracy of recognition in the single camera case. Also we are building a camera



	Tracking Speed(Frames Per Sec)
GPU	150
DSP	50
CPU	5

Fig. 14. Comparison of tracking rates in terms of frames per second for different implementations (CPU Vs DSP Vs GPU).(For details please refer to Section VI-B)

network where in we can use the results of tracking in a single camera to perform handoffs to other cameras in the building in order to build situational awareness of locations to answer questions such as ”who went where?”

REFERENCES

[1] NVIDIA, “Nvidia compute unified device architecture,” http://www.nvidia.com/object/cuda.html, 2008.

[2] P. Viola and M. J. Jones, “Robust real-time face detection,” *Int. J. Comput. Vision*, vol. 57, no. 2, pp. 137–154, 2004.

[3] R. Thota, A. Kalyanasundar, and A. Kale, “Modeling and tracking of faces in real life illumination conditions,” in *Proc. Int. Conf. Acoustics, Speech and Signal Processing*, 2009.

[4] K.-K. Sung and T. Poggio, “Example-based learning for view-based human face detection,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 1, pp. 39–51, 1998.

[5] H. A. Rowley, S. Member, S. Baluja, and T. Kanade, “Neural network-based face detection,” *IEEE Transactions On Pattern Analysis and Machine intelligence*, vol. 20, pp. 23–38, 1998.

[6] H. Schneiderman and T. Kanade, “A statistical method for 3d object detection applied to faces andcars,” *IEEE Conference On Computer Vision and Pattern Recognition*, vol. 1, pp. 746–751, 2000.

[7] M. hsuan Yang, D. Roth, and N. Ahuja, “A snow-based face detector,” in *Advances in Neural Information Processing Systems 12*. MIT Press, 2000, pp. 855–861.

[8] S. Paschalakis and M. Bober, “Real-time face detection and tracking for mobile videoconferencing,” *Real Time Imaging*, vol. 10, no. 2, pp. 81–94, April 2004.

[9] G. Xu and T. Sugimoto, “Rits eye : A software-based system for realtime face detection and tracking using pan-tilt-zoom controllable camera,” in *Proc. Int. Conf. Pattern Recognition*, 1998, pp. 1194–1197.

[10] O. M. Lozano and K. Otsuka, “Simultaneous and fast 3d tracking of multiple faces in video by gpu-based stream processing,” in *Proc. Int. Conf. Acoustics, Speech and Signal Processing*, 2008, pp. 713–716.

[11] H. Ghorayeb, B. Steux, and C. Laurgeau, “Boosted algorithms for visual object detection on graphics processing units,” in *Proc. of the 7th Asian Conference on Computer Vision (ACCV ’06)*, 2006, pp. 254–263.

[12] Y. Freund and R. E. Schapire, “A decision-theoretic generalization of on-line learning and an application to boosting,” in *EuroCOLT ’95: Proceedings of the Second European Conference on Computational Learning Theory*. London, UK: Springer-Verlag, 1995, pp. 23–37.

[13] A. Kale and C. Jaynes, “A joint illumination and shape model for visual tracking,” *Proceedings of IEEE CVPR*, pp. 602–609, 2006.

[14] Y.Weiss, “Deriving intrinsic images from image sequences,” *Proc of ICCV*, 2001.