cuMASM: Realtime Automatic Facial Landmarking using Active Shape Models on Graphics Processor Units

Nicholas Alexander Vandal

CMU-RI-TR-11-16

Submitted in partial fulfillment of the requirements
for the degree of Master of Science

Master's Committee:
Prof. Marios Savvides, PhD (ECE)
Prof. John Dolan, PhD (RI)
Matt McNaughton, MS (RI)

May 2011

Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

Automatic, robust, and accurate landmarking of dense sets of facial features is a key component in face-based biometric identification systems. Among other uses, dense landmarking is used to normalize raw faces for scale perform facial expression analysis, and is an essential component for generating 3D face models from a single 2D image. Active shape models (ASMs), which incorporate constrained statistical models of shape with local texture models of each landmark, have been applied successfully to this problem as well as landmarking tasks in other domains. Recent work has demonstrated that Modified Active Shape Models (MASMs), which utilize improved subspace models of 2D landmark neighborhoods, generalize better to unseen faces and to real-world dynamic environments. This superior performance comes with a significant computational cost, on the order of seconds per image to reach convergence. Compounded with the time required for face detection on high-resolution images, robust facial landmarking on the CPU is decidedly not realtime even for a well-optimized, multithreaded C++ implementation. In this paper, we demonstrate realtime MASM facial landmarking by parallelizing the algorithm on Graphics Processing Units (GPUs) using the CUDA programming platform. Our GPU-based implementation is designed for integration into a larger face recognition routine and is able to accept updated model parameters without recompilation or re-synthesis. Unlike previous GPU-based ASM implementations, which parallelize the original ASM algorithm utilizing 1D profiles, we implement the 2D subspace-modeled profile searching of the more robust MASM technique. We report GPU speedups of 24X over single-threaded CPU implementations of MASM and approximately 12X over a 8-threaded CPU implementation. By leveraging this untapped source of computational power, we are able to achieve realtime frame rates of approximately 20 FPS using a 79-point landmarking scheme. We discuss parallelizing the facial landmarking fitting process, specific GPU implementation details, GPU architecture-specific optimizations required to take advantage of the underlying hardware, and general CUDA programming concepts.

Contents

1 Introduction

1.1 Motivation

Biometrics are physiological characteristics of an individual that allow for the unique identification and/or authentication of a subject. As a rapidly growing inter-disciplinary field of Computer Vision, Pattern Recognition and Signal Processing, Biometrics research is primarily concerned with solving practical security problems in the real world: such as preventing identity theft, the identification of criminals and terrorists, and the authentication of authorized users in physical and virtual spaces. There are numerous advantages to using biometrics over more traditional means of identification and authentication such as passwords or physical objects such as keys or hardware tokens. With biometrics, *you* are your own password and key – there is nothing to lose or forget to bring with you. As we are all imperfect natural objects, the products of random environmental factors and genetics, there is tremendous variability in the human population; biometrics can be highly resistant to forgery and spoofing. There are many well-known biometric modalities including fingerprints, iris patterns, retinal vasculature, gait mechanics, voice prints, DNA, and face recognition. Face-based biometrics have the advantage of being non-intrusive to acquire due to relatively large target area, and omnipresent due to ubiquitous presence of security cameras. Additionally they can convey the emotional state of an individual, which can be useful in predicting future behavior. Some challenges of face recognition are that the face can be relatively easily altered due to disguise, surgery, accidents, the presence of hair, weight gain or even just aging. Even state-of-the-art face detection algorithms are very sensitive to changes in illumination and occlusion. Furthermore, compared to some biometrics such as the iris, where even genetically identical iris patterns (such as in the case of twins or even between the left and right eyes of all people) are radically different, human faces are less discriminative.

1.2 Facial Landmarking

Robust, automatic, facial feature landmarking (labeling prominent features of the face such as the eyes, lips, nose, mouth, and eyebrows) is highly desirable. Such landmarks are not merely interest points, but must be consistently defined across subjects with different craniofacial structure, under various pose configurations and illumination conditions. There are numerous applications requiring accurate spatial representations of a person's face (or any other object for that matter). Robust facial recognition often utilizes various pose detection and correction schemes which rely on accurate localization of prominent features. Expression analysis may reveal useful data about a subject's current state of mind and potential future behavior. Subfeature extraction also relies on accurate landmarks. For example, one can conceive of beard and mustache detectors, and identification schemes restricted to the periocular or nose regions. Additionally, the performance of texture-based face recognition systems can be boosted by the incorporation of accurate spatial data directly into the feature space.

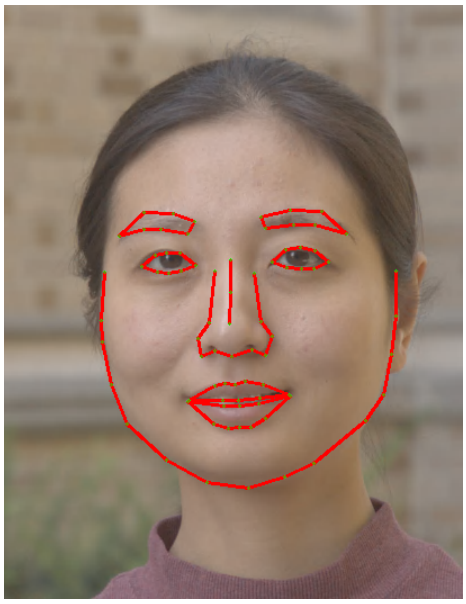


Figure 1: Example of automatically annotated facial image with the dense landmarking scheme employed in this paper.

1.3 Active Shape Models

Active Shape Models (ASMs) have been successfully used since their introduction in [9] to effectively model a variety of shapes including faces. An ASM is a type of deformable model like active contour models (snakes) [7], which seeks to match a set of model points to a test image, but is constrained by a learned statistical model of valid shapes. The ASM fitting algorithm can be thought of as a variant of the Expectation Maximization(EM) method, which iteratively alternates between two steps: (1) searching for the position around each point's current location that best matches the learned texture model expected at that point, and (2) updating and enforcing a global shape model constraint. This process is repeated at each level of an image pyramid, traversing from coarse to fine resolution and using the results from the previous level to initialize the current level. The original ASM implementation by Tim Cootes et al. [8][9][10] has been extended in [12][13][14][15][16][17] for improved accuracy in landmarking facial images.

Given the usefulness of ASM-derived landmarks for biometrics applications, and the fact that ASMs fitting is often but one stage in a much larger integrated tracking or recognition system, run-time performance is critical. Landmarking multiple faces rapidly, or processing video streams in real-time often requires execution times that even modern multi-core CPUs cannot deliver. Although ASM fitting makes use of multi-resolution pyramidal refinement, and there have been run-time optimization attempts such as using sparse covariance matrices in [16], the more robust modified ASM technique introduced in [17] is most decidedly not "real-time". Fortunately, we are able

to exploit the parallelism present in the ASM test stage to dramatically speed up this important task.

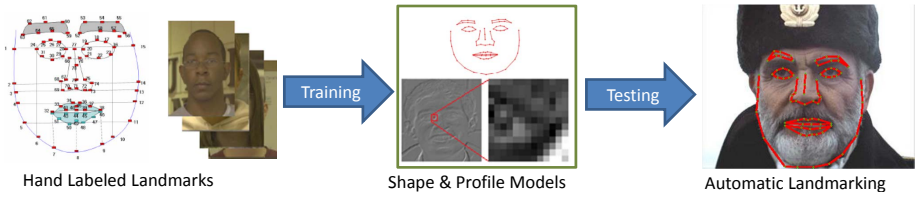


Figure 2: ASM high-level overview.

1.4 Graphics Processing Units

One approach to accelerating compute-intensive tasks is to offload some or all of the computations to dedicated hardware, such as a custom Application Specific Integrated Circuit (ASIC) or Field Programmable Gate Array (FPGA) co-processor. FPGAs are essentially programmable logic devices where virtually any parallel logic function can be synthesized with a few lines of a Hardware Description Language (HDL), such as VHDL or Verilog. While the performance of custom hardware is difficult to beat, FPGAs are often expensive and involve high complexity both in terms of expressing the problem in an inherently parallel HDL and in interfacing the FPGA with the host, peripheral storage, and input devices required for an operational system. Additionally, FPGAs possess reduced flexibility when compared to a pure software implementation.

A lower-cost acceleration alternative, which maintains the high degree of parallelism inherent to FPGAs but avoids many of their pitfalls, is to employ commercial off-the-shelf graphics processing units (GPUs). Driven by the insatiable market demand for realistic 3D games, the GPU has evolved into a highly parallel, multithreaded, many-core processor of tremendous power. In terms of peak FLOPS, modern commodity GPUs overshadow CPUs, as shown in (Fig 3). The growth rate of computational capabilities of GPUs has increased at an average yearly rate (depending on the metric used) of 1.7 (pixels/second) to 2.3 (vertices/second), which is a significant margin over the average yearly rate of roughly 1.4x for CPU performance [3]. The reason for this discrepancy is that while improvements in semiconductor fabrication technology benefit both the CPU and GPU equally, fundamental architectural differences favor the GPU in terms of peak computational throughput. The CPU dedicates a large portion of its die space to cache and flow control hardware, such as branch predication and out-of-order execution – it is optimized for high performance in executing sequential code, whereas the GPU is optimized for executing highly data-parallel rendering code and is able to devote more transistors directly to computation [3]. However, there is a definite trend towards a convergence of the GPU and CPU because CPUs have added more and more cores, and GPUs have evolved from fixed rendering pipelines to today's fully programmable architectures.

Prior to the introduction of Nvidia's Compute Unified Device Architecture (CUDA) API, and ATI's equivalent FireStream in 2006, performing general purpose computa-

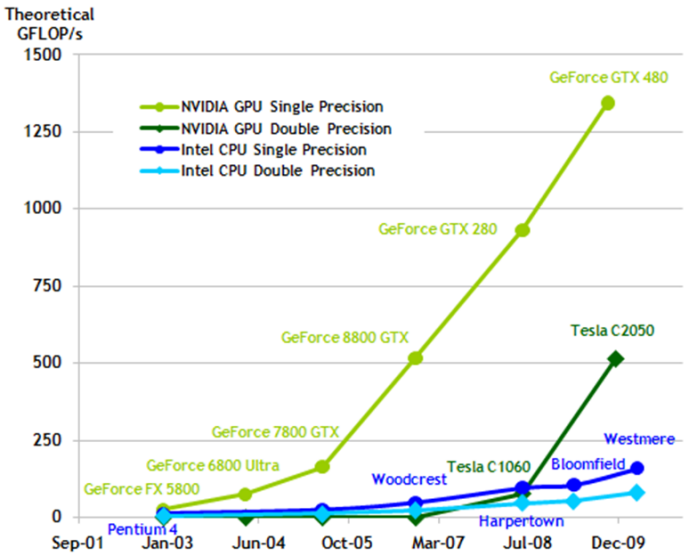


Figure 3: FLOPS capability comparison of GPUs and CPUs clearly demonstrating dominance of the GPU [2].

tion on GPUs was challenging work, requiring programmers to express problems in terms of rendering textures on shader units and other graphics primitives in what is termed General-Purpose Computation on Graphics Hardware (GPGPU). Despite these difficulties, problems such as protein folding, stock options pricing, SQL queries, and MRI reconstruction achieved remarkable performance speedups on the GPU. CUDA is the hardware and software architecture that enables Nvidia GPUs to execute programs written with C, C++, Fortran, OpenCL, DirectCompute, and other high level languages [1]. This abstraction dramatically lowers the barrier to developing GPU-accelerated applications and has led to an explosion in the field of GPU computation.

1.5 Related Prior Work

In this paper we take a well-optimized C++ implementation of the modified ASM fitting algorithm detailed in [17] and parallelize the profile search on GPUs using the CUDA programming model. Our GPU-based implementation is designed for integration into a larger CPU face recognition system and is able to accept updated model parameters without recompilation or re-synthesis. Unlike the previous GPU-based ASM implementations in [19][20], which are parallel implementations of the original ASM algorithm utilizing 1D profiles, we implement the 2D subspace-modeled profiles of the more robust technique found in [17]. Additionally, although [19] reports an impressive speedup of 48X over a single-threaded CPU implementation, their GPU implementation strongly relies on parallelism found in performing ASM fitting on *multiple images simultaneously*. While this scenario works in some cases (e.g. batch processing a number of facial images offline or performing a rolling average of ASM landmarks), for

live video feeds where each frame must be processed independently as it arrives, this use case breaks down if the system needs a real-time response (and can not buffer a huge number of images). While [19] does not cite an execution time or speedup ratio for a single frame computation, they do provide a bar graph which represents a speedup factor of approximately 7X for operating on four images simultaneously. The speedup ratio does not reach 48X until 192 images are operated on simultaneously. Furthermore, while the implementation in [20] is part of a larger tracking system, the paper offers virtually no details on the method used to parallelize the ASM fitting, and fails to list any quantitative results in terms of run-time performance, fitting accuracy, or tracking performance.

2 Active Shape Models

In this section we describe the traditional ASM scheme introduced in [9].

2.1 Training Stage

The training stage of ASM involves learning the two submodels that compose the ASM model (the *profile model* and the *shape model*) at each level of the pyramid. Training is performed using a set of manually labeled training images.

There is one profile model for each landmark point at each pyramid level. The profile models are used to determine the closest matching position for each landmark for the subsequent iteration by template matching. Various template matchers have been proposed, but traditional ASM uses a 1D fixed-length normalized gradient vector sampled along the line perpendicular to the shape boundary at the landmark. For each model point i in each training image j, extract the 1D profile of length $n_p = 2k + 1$ centered at point i.

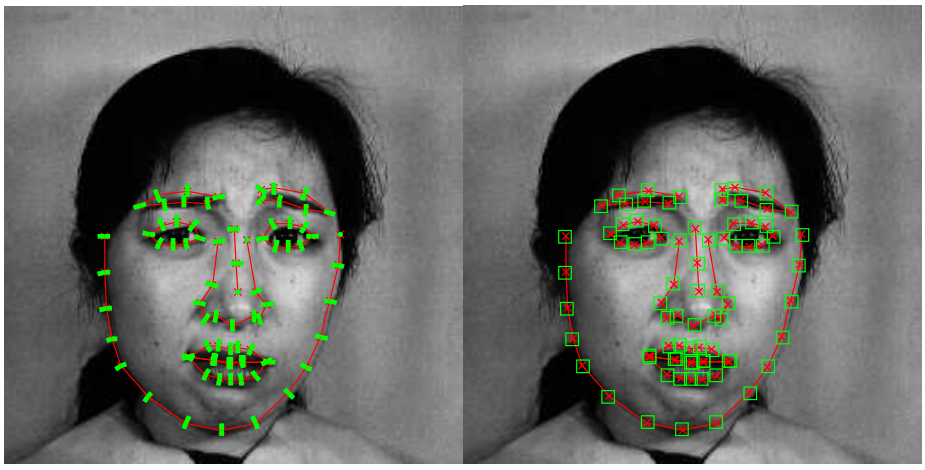


Figure 4: 1D and 2D profile schemes.

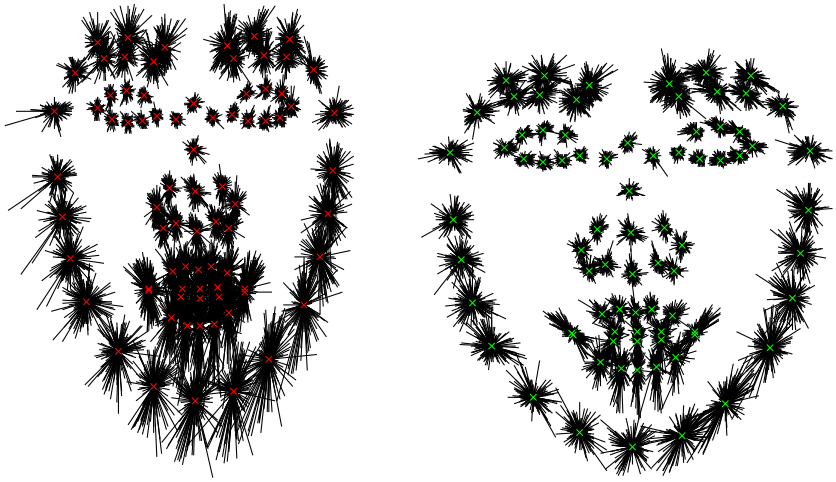
Together, the mean profile vector (averaged over all of the training images) $\bar{\mathbf{g}}^i$ and the covariance matrix ${\mathbf{S}_\mathbf{g}}^i$ form the profile model for the i-th landmark, at a given pyramid level. Analogously, 2D profile vectors can be generated for each landmark by extracting a square $(2k+1) \times (2k+1)$-sized patch of the image gradient around each landmark, vectorizing the resulting matrix, and applying a transform as shown in (Eq. 1) for normalization to each element of the profile g_j where q is a constant.

$$g'_j = g_j / (|g_j| + q) \tag{1}$$

Figure 5: Image, Gradient, and Sobel Edge Intensity pyramids used for multi-resolution search. Individual profile models are extracted to represent the neighborhood surrounding each landmark point at each level of the pyramid.

The shape model specifies allowable landmark arrangements. It is trained by storing the coordinates of all the landmarks for each image in a shape vector $\mathbf{x}$ where $\mathbf{x}_j = (x_1, y_1, ...x_N, y_N)^{\mathrm{T}}$, and x_i and y_i are the x and y coordinates of the i-th landmark, and N is the number of landmarks used. To compare equivalent points from different shapes we must first ensure that they are aligned. We seek to apply a similarity transform (translation, rotation, scaling) T_j to each $\mathbf{x}_j$ so as to minimize the sum of squared errors between equivalent points over our entire training set. We call this transform the *pose* of the shape. This alignment can be performed using Generalized Procrustes Analysis (GPA) [21]. The effect of performing GPA is to reduce variance due to pose differences (resulting from either the actual pose of the subject and/or differences in camera parameters and post-capture cropping). This can be seen when the distance vectors between each point in the training dataset and the unaligned mean shape (Fig. 6(a)), and the aligned shapes and their corresponding mean (Fig. 6(b)) are plotted. It is important to note that the points around the eyes are the most stable, while the points along the neckline demonstrate the greatest amount of variance (especially in the direction parallel to the boundary).

The mean shape $\bar{\mathbf{x}}$ is computed from these aligned shapes and Principal Component Analysis (PCA) is performed to build the basis matrix $\mathbf{P}_\mathbf{s}$.



(a) (b)

Figure 6: Mean shapes of JAFFE dataset before (a) and after (b) Generalized Procrustes Analysis (GPA).

$$\bar{\mathbf{x}} = \frac{1}{M}\sum_{j=1}^{M}\mathbf{x}_j \tag{2}$$

$$\mathbf{S} = \frac{1}{M}\sum_{j=1}^{M}(\mathbf{x}_j - \bar{\mathbf{x}})(\mathbf{x}_j - \bar{\mathbf{x}})^{\mathrm{T}} \tag{3}$$

$\mathbf{P_s}$ is a $(N \times t)$ matrix containing the first t eigenvectors of the covariance matrix $\mathbf{S}$ arranged in columns, corresponding to the t largest eigenvalues that model some fraction (97% in our implementation) of the total training shape variance.

$$\mathbf{S}\mathbf{p}_k = \lambda_k \mathbf{p}_k \tag{4}$$

$$\mathbf{P_s} = (\mathbf{p}_1, \mathbf{p}_2, ...\mathbf{p}_t) \tag{5}$$

Any shape in the training set can be approximated using the mean shape and a linear combination of eigenvectors of our basis, as shown in (Eq. 6). If we were to include all the eigenvectors in our basis, we would be able to exactly represent any shape vector in our training set with 0% reconstruction error.

$$\tilde{\mathbf{x}} = \bar{\mathbf{x}} + \mathbf{P_s}\mathbf{b} \tag{6}$$

2.2 Testing Stage

Like with other EM algorithms, the fitting of an ASM is very sensitive to its initialization values. The testing stage is typically initialized using a fast global face detector such as the Viola-Jones face detector implemented in OpenCV [22]; however, other initialization methods may make use of Kalman filter [18] or Mean-Shift based tracking [20] when using video sequences of moving faces. Once the face is detected, a similarity transform is applied to the mean face to generate a start shape that roughly approximates the test face image. This start shape $\mathbf{x}_{-1}$ initializes the coarsest level of the multi-resolution profile adjustment. For each iteration, candidate locations are assessed for each landmark point by constructing profiles from patches surrounding each candidate location. The candidate location that most closely matches the mean profile for the landmark learned during training is selected as the new location of the landmark for the next iteration. The best new location of each landmark point moved independently, together form a new intermediate shape vector ($\mathbf{x_I}$).

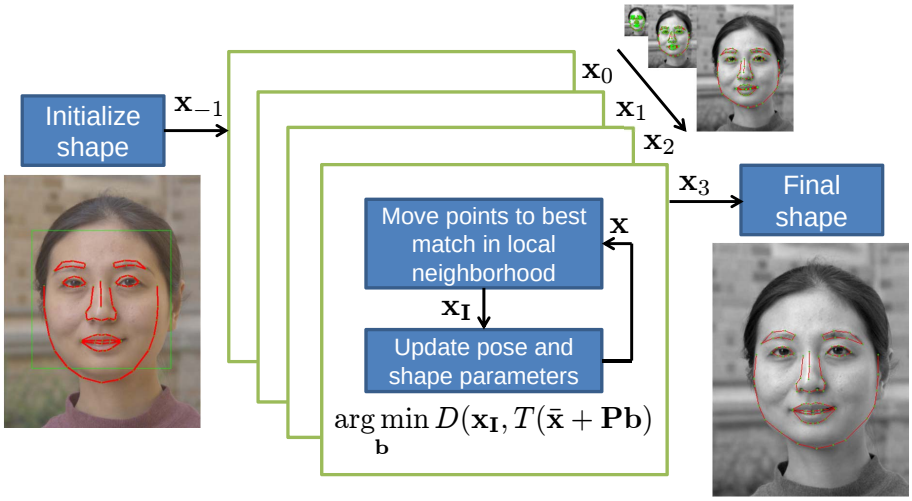


Figure 7: Test stage steps: (a.) Initialization of start shape using Viola-Jones face detection and GPA; (b.) Iterative pyramidal refinement of shape from coarse to fine; (c.) Refitting poorly fitting points to yield final landmarking.

The cost function used in most ASM implementations is the Mahalanobis distance ($D_1(\mathbf{g_s})$) between the candidate profile ($\mathbf{g_s}$) and the mean profile ($\bar{\mathbf{g}}$).

$$D_1(\mathbf{g_s}) = (\mathbf{g_s} - \bar{\mathbf{g}})^T {\mathbf{S_g}}^{-1} (\mathbf{g_s} - \bar{\mathbf{g}}) \tag{7}$$

After moving all landmarks to their optimum new location, the new intermediate shape vector ($\mathbf{x_I}$) must be constrained to a legal shape. Legal shapes are defined by the generative linear shape model equation (Eq. 8). Legal shapes can be generated by varying the projection coefficient vector ($\mathbf{b}$) known as the *shape parameters*.

$$\mathbf{x_L} = \bar{\mathbf{x}} + \mathbf{P_s b} \tag{8}$$

The effect of varying each of the first three shape parameters independently over a range of $\pm 3\sigma$ while the others remain at zero can be seen in (Fig. 8). We seek to find the best approximation of $\mathbf{x_I}$ by minimizing (Eq. 9) with the iterative algorithm given by [9] that produces $\mathbf{b}$ and T, where T is a similarity transform that best maps the model space into the image space (*pose parameters*).

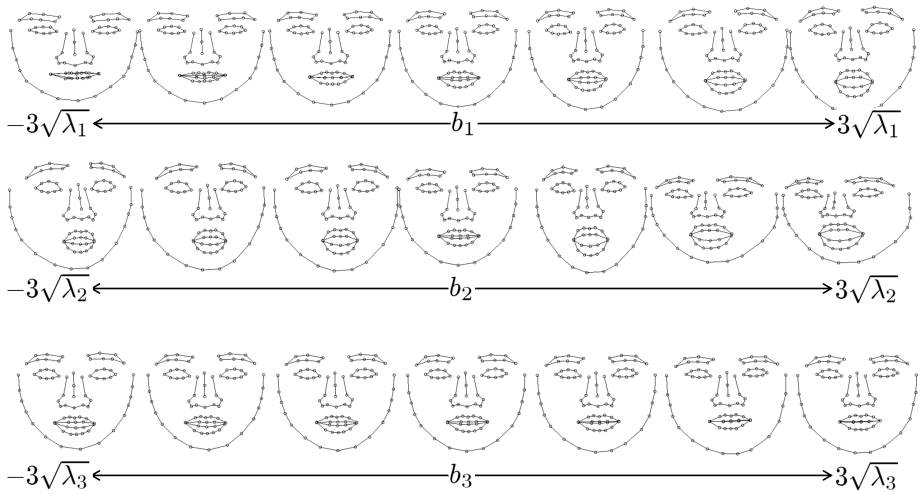


Figure 8: Effect of changing the first three projection coefficients in the shape equation.

$$\arg\min_{\mathbf{b},T} D(\mathbf{x_I}, T(\bar{\mathbf{x}} + \mathbf{P_s}\mathbf{b})$$

Algorithm 1 Constrain to legal face shape

Input: $\mathbf{x_I}$
Output: T and $\mathbf{b}$
  $\mathbf{b} \Leftarrow \mathbf{0}$
  while not converged **do**
    $\mathbf{x_L} \Leftarrow \bar{\mathbf{x}} + \mathbf{P_s}\mathbf{b}$
    $T \Leftarrow \mathrm{GPA}\,(\mathbf{x_L}, \mathbf{x_I})$
    $\mathbf{y} \Leftarrow T^{-1}\mathbf{x_I}$
    $\mathbf{y} \Leftarrow \frac{1}{y^{\mathrm{T}}\bar{\mathbf{x}}}\mathbf{y}$
    $\mathbf{b} \Leftarrow \mathbf{P_s}^{\mathrm{T}}(\mathbf{y} - \bar{\mathbf{x}})$
  end while

Once $\mathbf{b}$ is determined, its elements are clipped to lie between $\pm b_{max}\sqrt{\lambda_k}$ where b_{max} is generally 3 (i.e., 3 standard deviations) and λ_k is the k-th eigenvalue corresponding to the eigenvectors retained in $\mathbf{P_s}$. This constrains the shape to be a feasible one. This process of updating landmark positions is repeated until convergence occurs (no significant movement between subsequent iterations) at each pyramid level, at

which point the landmark positions are scaled and fed into the next level of the pyramid as initialization locations. Once convergence occurs at the highest resolution pyramid level, the algorithm terminates.

3 Nvidia's CUDA

The CUDA hardware model is built on an array of multithreaded Streaming Multiprocessors (SMs), which are each designed to execute hundreds of simultaneous threads. When a program running on a host CPU executes a CUDA "kernel", which is a section of device code that executes N times in parallel by N threads on the GPU, the kernel is launched on the GPU and control immediately returns to the CPU. This asynchronous behavior allows simultaneous work to be performed on both the host and the device in what is known as heterogeneous programming. A basic CUDA kernel is given in (Listing 1), which simply adds two vectors together. As you can see, CUDA syntax is a superset of C, which adds the $<<<>>>$ *execution configuration* operator. Within these blocks, a programmer specifies the device configuration that should be used while executing this block of code on the GPU. Nvidia terms this architecture SIMT (Single-Instruction, Multiple-Thread), which is similar to the more well-known SIMD (Single Instruction, Multiple Data) in that a single instruction operates on multiple processing elements; however, the crucial difference is that SIMD vector operations expose the SIMD width to the software, while SIMT instructions specify the execution and branching behavior of a single thread [2]. SIMT enables programmers to write thread-level parallel code for independent, scalar threads, in addition to data-parallel code for coordinated threads. SIMD architectures require the programmer to coalesce loads into vectors and manage divergence manually [2]. From a correctness perspective, threads are free to branch and execute completely independently, facilitating rapid parallel implementations of many algorithms. However, for maximum performance, programmers must take care to write code that minimizes divergent branching within thread warps (group of 32 threads that execute one common instruction at a time) and properly utilizes the memory hierarchy of the device.

Listing 1: Element-wise vector addition

```
//kernel definition
__global__ void VecAdd(float * A, float * B, float * C)
{
        int i = threadIdx.x;
        C[i] = A[i] + B[i];
}

int main()
{
        ...
        //kernel invocation with N threads
        VecAdd<<<1, N>>>(A, B, C);
}
```

CUDA has three key abstractions that allow for effective and relatively easy development compared to traditional GPGPU techniques: a hierarchy of thread groups,

shared memories, and barrier synchronization.

3.1 Thread Hierarchy

Threads are arranged in a hierarchy of blocks and grids as shown in (Fig. 9). Each thread executes an instance of a kernel function and has a unique thread ID and program counter along with registers and private local memory [2]. A thread block is a set of concurrently executing threads that have shared memory and can synchronize with barrier primitives. Each thread block has a block ID which provides a unique identifier within a grid, which is an array of thread blocks [2]. Grids share global device memory and perform synchronization between successive kernel calls [2]. The threads of a block execute concurrently on a single SM, and multiple thread blocks can execute on one multiprocessor. As thread blocks terminate, new blocks are launched on the vacated multiprocessors [2]. This abstraction allows for automatic scalability of CUDA code, as one typically allocates threads based on problem size (i.e., in order to perform an element-wise addition of two vectors of length N, one launches N threads) instead of targeting a specific hardware implementation. More capable devices, with a greater number of SMs, are able to process more of the threads concurrently, leading to faster execution times without necessitating major code revisions.

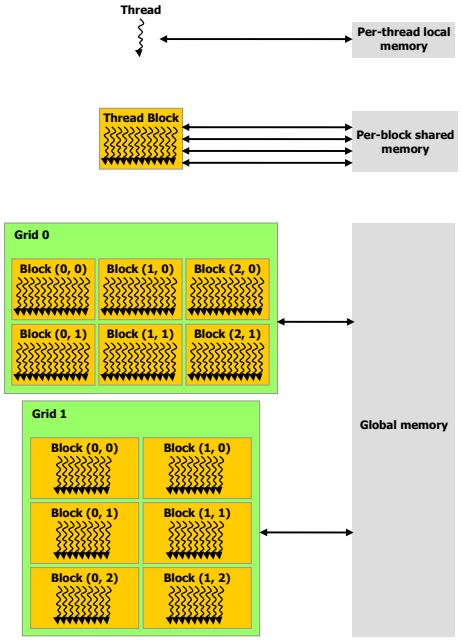


Figure 9: CUDA programming model of threads, blocks, and grids, with corresponding per-thread private, per-block shared, and per-application global memory spaces [2].

3.2 Memory Hierarchy

The CUDA memory hierarchy consists of registers, local memory, shared memory, global memory, the constant cache, and the texture cache. The registers reside on chip and are integrated into the SMs. They are private to each thread and have the fastest access times, but they are an extremely limited resource with 32768 32-bit registers available per SM on current-generation GPUs to be shared amongst all currently executing threads on that SM. Thus, the number of registers a kernel requires limits the number of simultaneously executing threads. If a kernel requires more registers than can be provided or they are artificially limited by the programmer at compile time, *register spillover* to local memory can occur. Despite its name, local memory resides in global memory and is as slow as accessing global memory. Register spillover can have disastrous effects on the performance of CUDA code and should be avoided. Shared memory also resides on chip and is limited to 16KB or 48KB per SM on current-generation devices. It is almost as fast accessing the register file, but like registers and local memory, is non-persistent between kernel launches. Global memory is persistent between kernel launches and provides the only means of communication between threads within different blocks and between the device and the host. It is plentiful, with high-end cards possessing GBs of DRAM, but it is slow (on the order of hundreds of clock cycles) and uncached with all but the latest generation of GPUs. The constant cache is read-only from the device, relatively small (16KB), and most effective when all threads are accessing the same element simultaneously. The texture cache is optimized for 2D spatial layouts and provides special addressing modes, bilinear interpolation, and normalization.

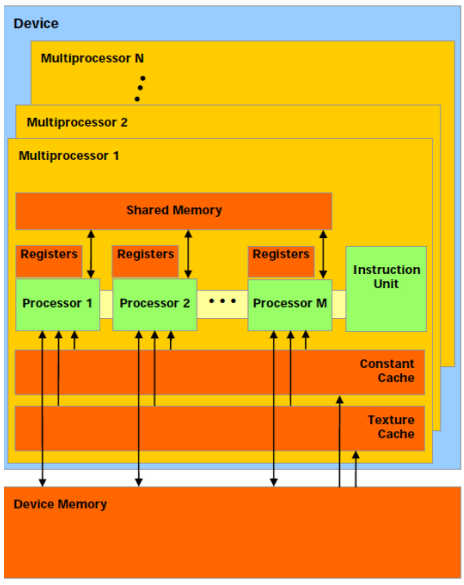


Figure 10: CUDA hardware model with global memory, constant cache, texture cache, registers, and shared memory [2].

4 Modified Active Shape Models

In this paper we implement a parallel version of the modified ASM (MASM) scheme specified by [17], which experimentally achieved better performance compared to the classical Active Shape Model of Cootes et al. and other traditional methods and provides a robust automatic facial landmark annotation despite expressions, slight pose variations and in-plane rotations. Specific details which differentiate MASM from classical ASM are in the next subsections.

4.1 Subspace Profile Models

MASM builds a subspace model of the profile around each landmark point to more effectively model the variations of appearance than could be achieved with a simple mean vector (as done with traditional active shape models introduced by Tim Cootes). This improvement provides superior fitting accuracy [17], particularly when using more training images, as traditional ASM degrades if more training images are added after a particular number (in our experiments we noticed above 500 training images, fitting accuracy degraded for traditional ASMs). This is because the mean landmark becomes blurry and does not have any discriminative information, whereas our approach builds a subspace to model how it changes across a large population of faces. Additionally, the subspace modeling provides a means of determining poorly fit points for additional correction (i.e., the reconstruction error of the best fit landmark patch can be used as a fitting cost function) to provide superior fitting accuracy [17].

During the profile training stage, in addition to obtaining the mean profile vector $\bar{\mathbf{g}}^i$ and covariance matrix ${\mathbf{S_g}}^i$ for each point i, [17] extracts the t_i eigenvectors which account for a sufficiently large fraction of the total variability.

$$\frac{\sum_{k=0}^{t_i} \lambda_k}{\sum_{k=0}^{n_p} \lambda_k} > 97\% \tag{10}$$

Those eigenvectors corresponding to the t_i largest eigenvalues are then stored in the basis matrix (${\mathbf{P_g}}^i$). During testing, each sample profile ($\mathbf{g_s}$) is extracted, and projected onto the set of eigenvectors (its own unique subspace) to obtain a vector of projection coefficients ($\mathbf{g'_s}$) as shown in (Eq. 11).

$$\mathbf{g'_s} = \mathbf{P_g^T}(\mathbf{g_s} - \bar{\mathbf{g}}) \tag{11}$$

These projection coefficients are then used to obtain a reconstructed profile vector ($\mathbf{g^r_s}$) as shown in (Eq. 12).

$$\mathbf{g^r_s} = \bar{\mathbf{g}} + \mathbf{P_g}\mathbf{g'_s} \tag{12}$$

The reconstruction error is then computed using the Mahalanobis distance between the reconstructed profile and original sample profile ($D_2(\mathbf{g_s})$) as shown in (Eq. 13). Better candidate points will have lower reconstruction errors, as they are adequately

represented by the subspace, while poor matches will have high reconstruction errors. MASM uses $D_2(\mathbf{g_s})$ as the template matching criteria instead of the traditional $D_1(\mathbf{g_s})$.

$$D_2(\mathbf{g_s}) = (\mathbf{g_s^r} - \mathbf{g_s})^T {\mathbf{S_g}}^{-1} (\mathbf{g_s^r} - \mathbf{g_s}) \tag{13}$$

4.2 Edge Information

The MASM scheme additionally uses Sobel edge intensity information to more accurately fit points along the facial boundary (points 1-15) using (Eq. 14) described in [13]. This modification is based on the assumption that facial points usually lie along strong edges. In (Eq. 14) I is the Sobel edge intensity and c is a scalar weight constant (our implementation, like [17], sets $c = 2$).

$$D_3(\mathbf{g_s}) = (c - I)(\mathbf{g_s^r} - \mathbf{g_s})^T {\mathbf{S_g}}^{-1} (\mathbf{g_s^r} - \mathbf{g_s}) \tag{14}$$

4.3 Refitting of Poorly Fitted Points

After convergence of ASM fitting on the final (highest-resolution) level of the pyramid, the cost metric of each point is compared to an empirically determined threshold. Those points with an error above the threshold are further subject to additional fitting iterations until their reconstruction error is reduced below the threshold. This boosts accuracy of individual points and leads to an overall improvement in the fitting.

4.4 Miscellaneous Details

Our implementation duplicates the parameters defined in [17]. We make use of 79 landmark points arranged according to the scheme shown in (Fig. 2) and extract 13×13 2D profiles around each point at each pyramid level. We search for candidate points in a 5×5 2D grid centered at every landmark's estimated position from the previous iteration. The improved robustness of MASM over traditional ASMs comes with a greater computational cost, as execution time on a single core of a modern CPU is on the order of seconds to converge. Since we make use of fairly large-dimensional 2D profiles, we have a large (169×169) covariance matrix and resulting subspace basis $(169 \times t_i)$ matrices. Computing the reconstructed profile requires $2(t)(n_p)$ MAD operations and calculating the Mahalanobis distance requires $n_p^2 + n_p$ MAD operations. These matrix operations must be performed for each landmark point over all possible offsets (shifts).

4.5 GPU Implementation

Key to accelerating any algorithm is locating sections where parallelism can be exploited and properly distributing the workload on a given parallel architecture. The computational bottleneck in both the serial CPU implementation and in our parallel GPU implementation, is evaluating each candidate landmark's distance from its subspace model. Even after heavy optimization, this profile-search kernel accounts for

73% of our CUDA implementation's runtime (Fig 11). We exploit the inherent independence of searching for the best matching location for each landmark's updated position to parallelize ASM fitting on the GPU. A naïve work-partitioning scheme would simply distribute the 79 landmark points between all available multiprocessors, as the operations on each landmark point are entirely independent in the profile-search step. This is exactly what is done in our CPU implementation designed for a 4-to-8-core machine; however, this dramatically under-utilizes the capacity of a device designed to have 1000's of threads running.

Additional concurrency is found in the 25 candidate locations that must be tested around each point; although the matrix operations for each offset are independent, determining the best candidate location requires synchronization and communication. This maps very naturally to CUDA's thread-block hierarchy, where problems are coarsely divided into independent sub-problems to be solved by blocks independently, while within a block, threads operate cooperatively. Additionally, by mapping a landmark location to a block, all threads within a block share the same profile model. This allows for reduced memory load requests, as these parameters can be broadcast to all threads in a block.

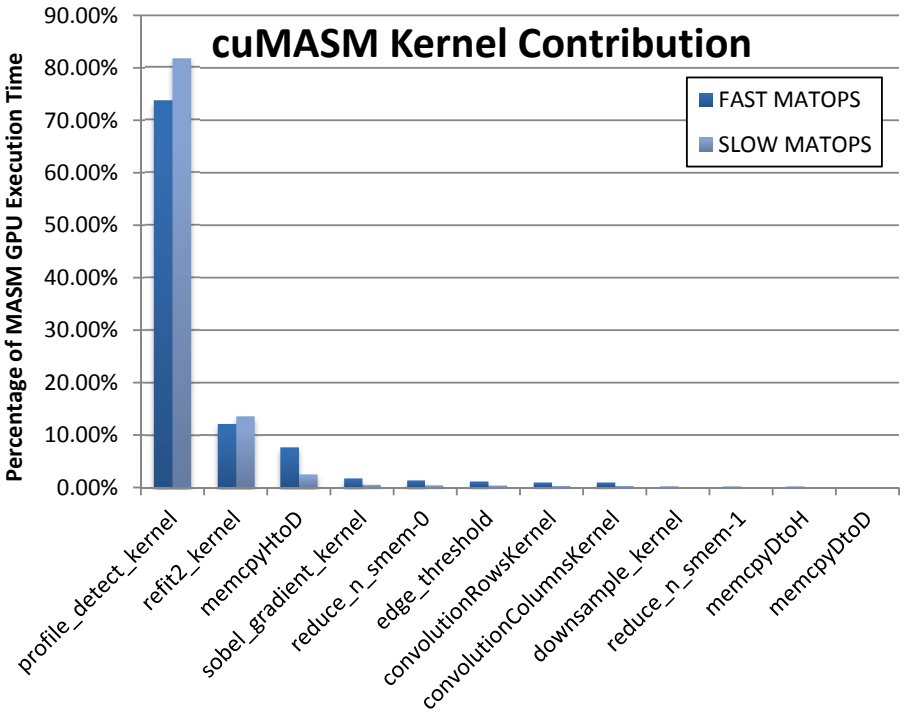


Figure 11: Percentage of total execution time each CUDA kernel or memcpy contributes.

Mapping MASM to 79 blocks of 32 threads (granularity of block size is 32) yielding a total of 2528 threads is still insufficient to hide memory latency and due to high shared memory requirements and register usage (needed to perform our matrix opera-

tions entirely within fast shared memory and registers), the number of threads per block should be $>> 32$. Fortunately, we can still parallelize the matrix operations. Instead of each thread performing its matrix multiplications serially, we assign 32 "worker" threads for each of the 25 offsets.

4.5.1 Execution Configuration

We initialize our routine by loading all overall parameters and model parameters to device global memory once, prior to beginning to process any images. Then we launch our 2D profile search kernel using a grid of thread blocks of size 32×25, where each block is assigned a specific landmark point to update (Fig. 12). Although our GPU implementations of other sections of the algorithm, such as Sobel filtering and gaussian pyramid generation, help to minimize data transfers over the PCI-E bus and are marginally faster than their CPU based counterparts, they contribute very little to the overall speedups achieved. Constraining each fitted face to a legal shape after each iteration and checking for convergence remain on the CPU as there is minimal parallelism to be exploited here.

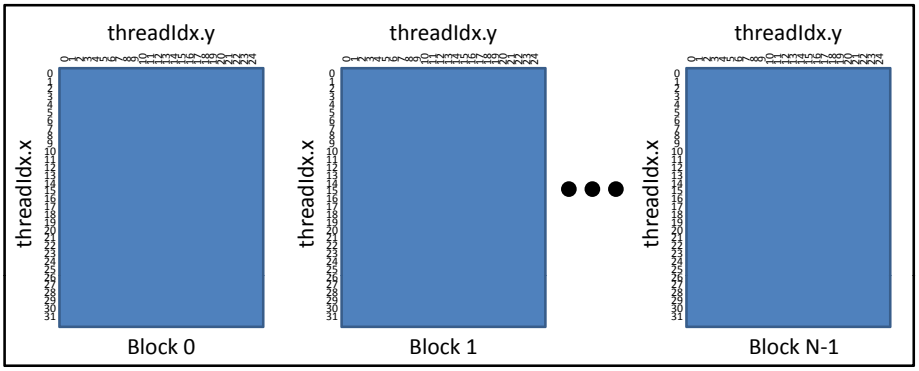


Figure 12: Graphical representation of cuMASM thread block and grid layout.

The *x*-coordinate of the threadIdx corresponds to a "worker ID" and the *y*-coordinate of the threadIdx corresponds to which of the 25 possible 2D shifts our trial profile is centered at. By assigning an entire block to a particular point, we are able to use the fast shared memory to hold intermediate results and model parameters and perform intra-block reductions without performing reads and writes to global memory. This significantly increases the speed at which we are able to perform ASM fitting, because like many applications on the GPU, this is memory-bound as opposed to computation-bound.

4.5.2 Texture Profile Fetching

We take advantage of the spatial locality and significant overlap of the memory access required to perform the initial subpixel level sampling of the shifted 2D profiles surrounding each landmark point's initialization point, by employing the texture cache.

According to the Nvidia profiler, our code achieves a texture cache hit rate of 99.69%. This is congruent with the fact that we found no performance benefit to cooperatively loading the 17×17 overlapping region (Fig. 13) into shared memory prior to filling the offset profile vectors. We could optionally take advantage of the texturing hardware's built-in low-precision bilinear hardware interpolation; however, in practice we found that performing the high-precision software interpolation did not significantly slow down an individual iteration and generally reduced the number of iterations required before convergence was achieved – thereby actually *decreasing overall runtime*. The constant cache is used to amortize the cost of accessing model parameters, especially the large covariance matrices required for computing the Mahalanobis distance at each of the 79 landmark locations, which at 111kB (169×169 matrices of single precision floats) are too large to completely store in shared memory.

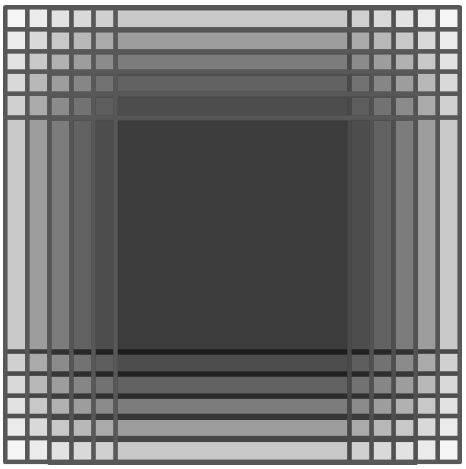
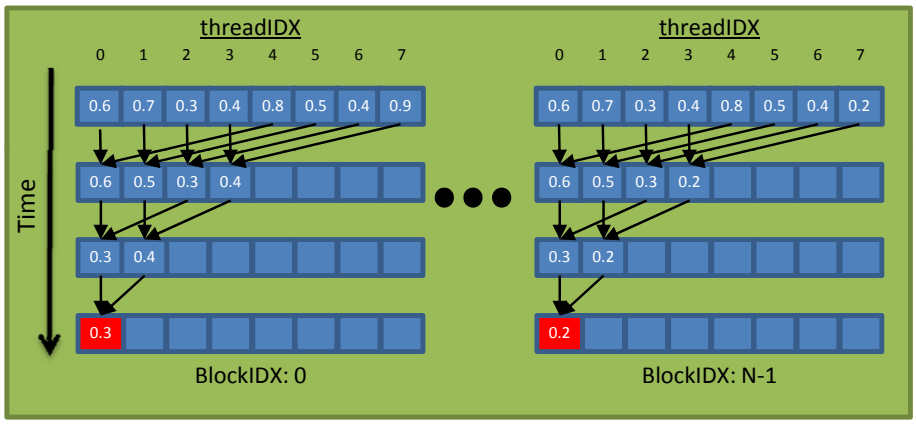


Figure 13: Overlapping 13×13 patches at 5×5 offset locations 1 pixel apart have significant redundant memory accesses.

4.5.3 Reduction Primitive

After the shifted trial profiles have been loaded into shared memory, each block of 800 threads must cooperatively evaluate the reconstruction error of each of the 25 profiles and determine which shift yields the lowest reconstruction error for each landmark point. The three building blocks which are critical for a GPU implementation of MASM are intra-block reduction, matrix-vector multiplication and Mahalanobis distance. Reduction is a common parallel primitive in which a commutative binary operator (ie., sum, product, min, max) is applied to all the elements of a vector to produce a scalar value. This operation is used to determine which shift within a thread-block results in the minimum reconstruction distance, as well as being using within the more complex matrix-vector multiplication and Mahalanobis distance operations. Reduction in CUDA has been well optimized [6], and we utilize this basic skeleton for our indexed minimum distance reduction and summation reductions. See (Fig. 14) for

a graphical representation and a simplified code fragment of non-indexed, minimum value reduction.




```
//each thread puts its local distance into shared memory
s_mem [threadID] = dist;
__syncthreads();

//do reduction in shared mem
#pragma unroll
for(unsigned int s=(blockSize>>1); s>0; s>>=1){
        if (threadID < s)
                s_mem[threadID] = dist = min(dist, s_mem[threadID+s]);
        __syncthreads();
}

//All threads get local copy of the min distance for the
//entire block (which is the point we are operating on)
min_dist2 = s_mem[0];
```

Figure 14: Cooperative reduction performed in block shared memory. Used to find minimal reconstruction distance for a given point (all threads in a block operate on the same point) as well as for several other reductions.

4.5.4 Naïve Intra-block Matrix Operators

Nvidia has released a highly optimized BLAS implementation for CUDA known as cuBLAS; however, it is designed for performing operations on extremely large matrices and only exposes a host interface. Our partitioning scheme makes use of many relatively small matrix operations which take place in the context of a larger kernel. Although we could reshape our MASM implementation to make use of the cuBLAS libraries, it would require wasteful stores to global memory. We require intra-block operators which can be incorporated within our kernels running on the device. Matrix-vector multiplication known by its BLAS function name as GEMV, performs the operation $\mathbf{y} = \alpha\mathbf{A}\mathbf{x} + \beta\mathbf{y}$, where $\mathbf{A}$ is a matrix of size $m \times n$, α and β are both scalars, and $\mathbf{x}$ and $\mathbf{y}$ are both vectors of size $n \times 1$. GEMV is used to project and reconstruct

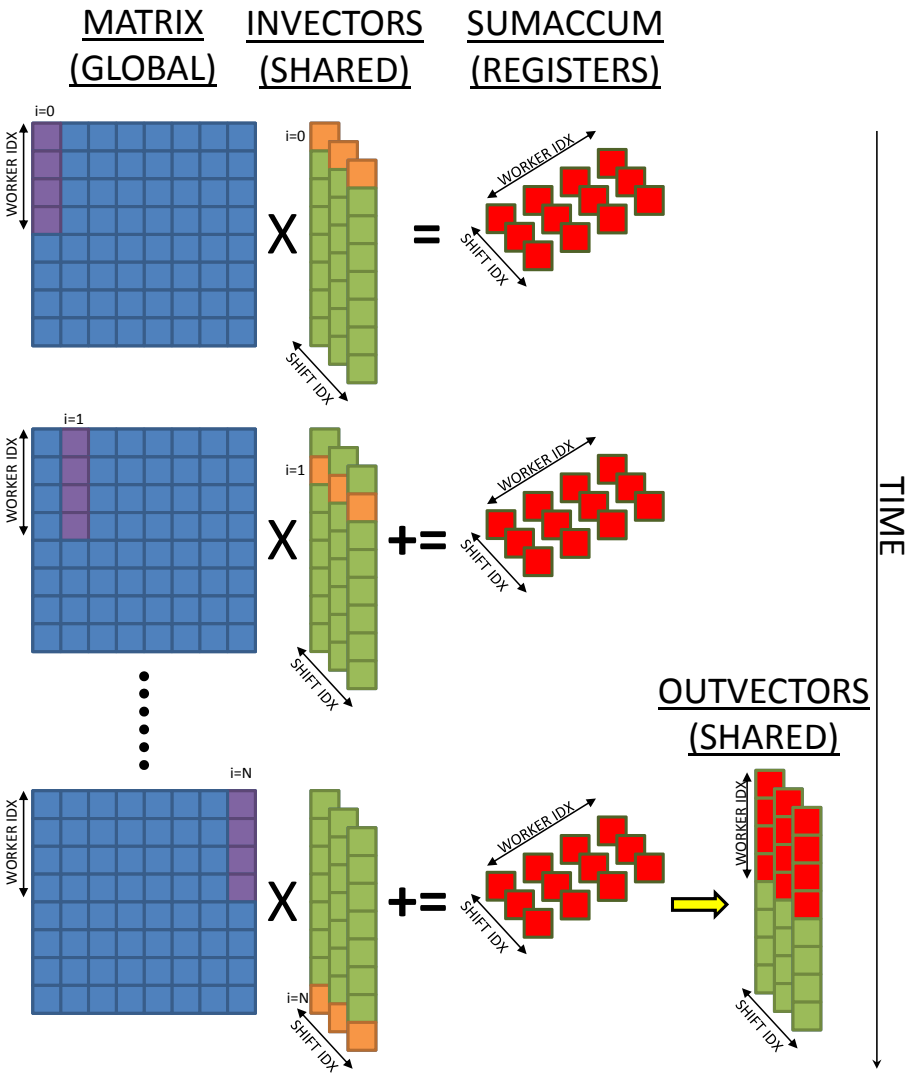


Figure 15: Graphical representation of naïve intra-block matrix-vector multiplication routine.

(Eq. 11 & 12) each profile. Additionally, since this is performed for each of the possible shifts, this can be viewed in terms of general matrix multiplications. A graphical representation of our initial parallel implementation of GEMV is shown in (Fig. 15), in which the rows of matrix A are partitioned by the 32 "worker" threads and iterated across the rows. Each thread maintains a local accumulator, located in the register file, in which the matrix-vector products for a given "worker index" are held before being stored back to shared memory.

Our Mahalanobis distance routine computes $\mathbf{y} = \mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x}$, where $\mathbf{A}$ is a square

matrix of size $m \times m$, and $\mathbf{x}$ and $\mathbf{y}$ are both vectors of size $m \times 1$. A graphical representation of our initial parallel implementation of this primitive is shown in (Fig. 16). This routine proceeds similarly to our matrix multiplication routine, except that instead of storing the accumulated sums from each row into the corresponding row of the output vector residing in shared memory, the sums are multiplied by the value stored in $\mathbf{x}$ at their "worker idx". This product is stored in another local accumulator, until a final summation reduction is performed and a single scalar for each shift remains.

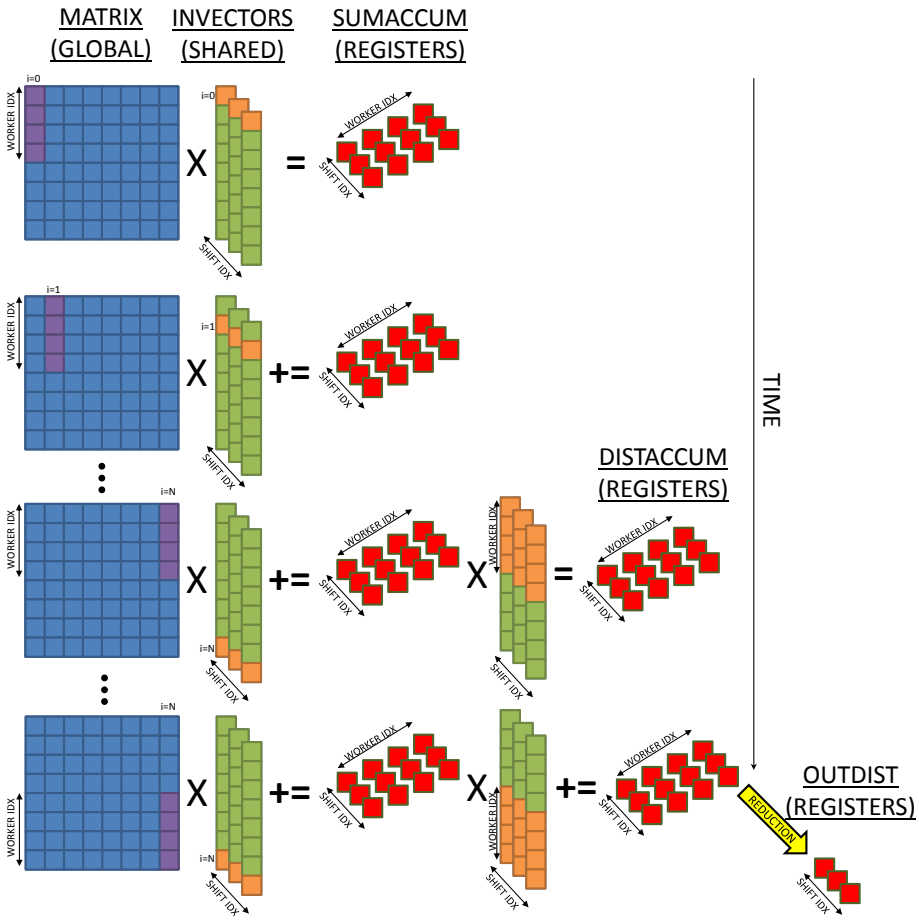


Figure 16: Graphical representation of naïve intra-block Mahanolobis distance routine.

4.5.5 Improved Intra-block Matrix Operators

Although these matrix operations take place in fast shared memory or the register file, and the input vectors also reside in shared memory, the L1/L2 cache can offset the cost of accessing the large matrices stored in global memory only to a limited extent.

Because each of the 79 landmark locations has individual basis eigenvector and covariance matrices, the cache can quickly become thrashed. This results in long latency in our many global memory accesses and significantly lowers throughput of our parallel implementation. Additionally, our access pattern within the matrix is non-optimal, as we are accessing non-contiguous sections of memory (reading columns in a row-major matrix). Clearly we must do better than this naïve implementation of such important primitives. Fortunately, by cooperatively reloading rows of each matrix from global memory into shared memory, we can greatly reduce the number of global memory load requests our kernels make, thereby reducing our memory bandwidth requirements, increasing our cache hit rate, and better hiding global memory access latency.

Listing 2: Cooperative intra-block cooperative load routine used within GEMV and Mahalanobis distance routines

```
template <typename T,int blockSize, bool nIsPow2> __device__ inline __forceinline__
void loadSmem(T * s_out,  const T * g_in, const int tid, const int n)
{
        const int gridSize = blockSize<<1;
        int i = tid;

        while (i < n)
        {
                s_out[i] = g_in[i];

                //Bounds check for non-powers-of 2
                if (nIsPow2 || i + blockSize < n)
                        s_out[i+blockSize] = g_in[i+blockSize];

                i += gridSize;
        }
}
```

We also change our memory access pattern to iterate down the rows of the matrix, and cooperatively partition the columns of the matrix between the ”worker threads”. Graphical representations of our intra-block cooperative GEMV and Mahalanobis distance routines are shown in (Fig. 17 and 18).

The net result of these memory optimizations can be clearly seen in the output of the Nvidia profiler (Fig. 19), which shows that we have reduced our global load requests by approximately 92.1% by relying on shared memory. Additionally, our code makes heavy use of templates and loop unrolling to reduce branching and comparisions as much as possible, while still remaining capable of operating on arbitarily sized images.

4.5.6 Refitting Kernel

The serial implementation of MASM iterates through each of the landmark points sequentially and iteratively refines those points with a reconstruction error above the empirically determined threshold until the error is reduced below this cutoff. Our CUDA implementation of this is identical to our initial profile search kernel executed on the finest level of the gaussian pyramid. We launch a modified kernel with the same execution configuration as before. The kernel has the additional task to checking if the current reconstruction error for the threadblock is below the threshold; if the threadblock

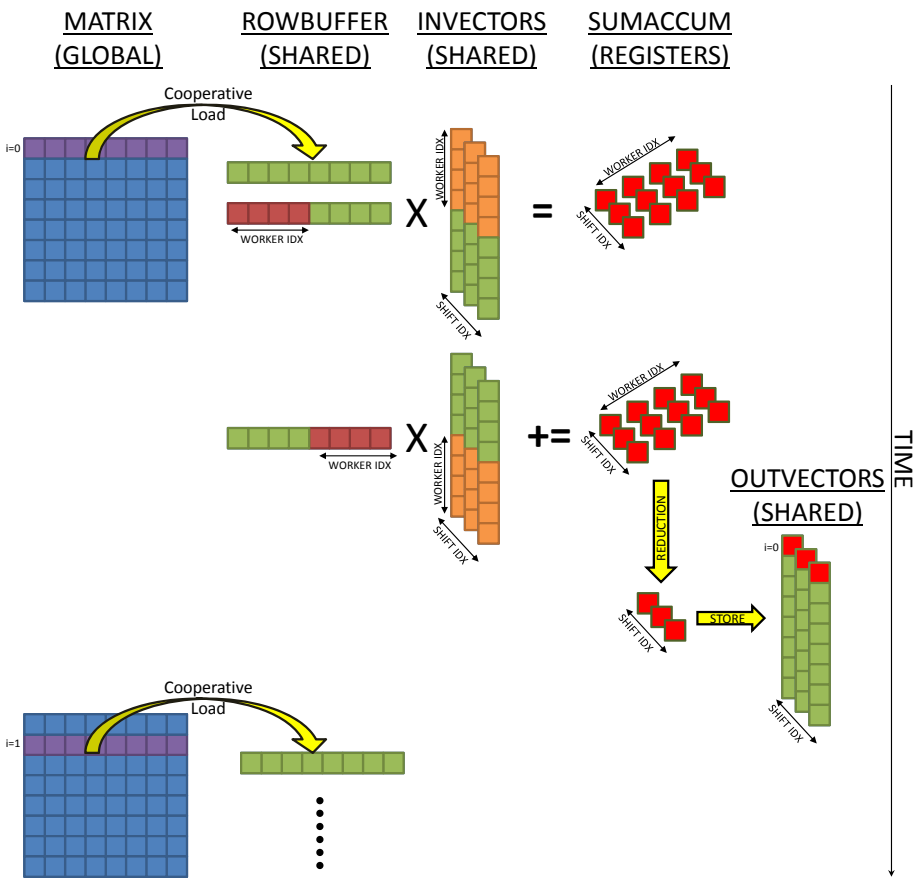


Figure 17: Graphical representation of cooperative intra-block matrix-vector multiplication routine, in which all threads in the block cooperatively load the rows of matrix **A** sequentially into shared memory and then individually perform their inner-products. This greatly reduces global memory bandwidth requirements.

passes this test, the entire block immediately terminates without additional refinement.

5 Experimental Results

The GPU we used was a Nvidia GTX 470 with the CUDA runtime v3.2. The device has a total of 448 cores organized into 14 streaming multi-processors (SM). We target our code specifically for the latest Fermi generation of GPUs from Nvidia (Compute Capability $>= 2.0$) and make use of nearly all of the available 48KB of shared memory in our profile search kernel. This was benchmarked against an Intel Core i7 running at 3.2GHz with 4 physical cores (8 SMT cores). Our CPU implementation was developed in C++ utilizing the OpenCV 2.1 library and multithreaded using OpenMP. The training

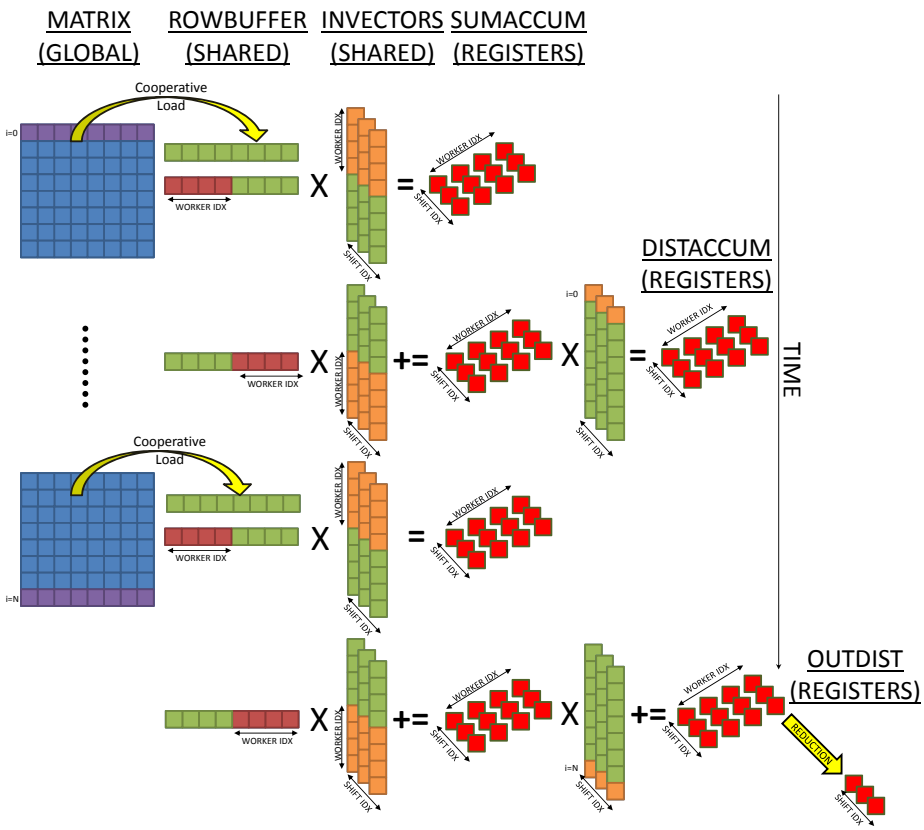


Figure 18: Graphical representation of cooperative intra-block Mahalanobis routine, in which all threads in the block cooperatively load the rows of matrix **A** sequentially into shared memory and then individually perform their inner-products and keep a local running sum. This greatly reduces global memory bandwidth requirements.

stage was performed offline using MATLAB code, as only the testing stage is run-time performance-critical.

5.1 Speed

We evaluated execution time performance by using various face-containing images taken from the Internet of differing size and differing pose/expressions. Additionally, we set up a live video demonstration that performs ASM fitting on a web camera or AVI stream and captured images (of various resolutions) of subjects from our lab performing work at their desks. This unconstrained dataset provides the variety necessary for an effective evaluation of MASM runtimes on the CPU and GPU. Runtime for MASM and ASMs in general is independent of image size; run time is data-dependent because ASM fitting is an iterative technique that runs until convergence. It depends on the closeness of the correct landmark positions to the initial locations provided by the face

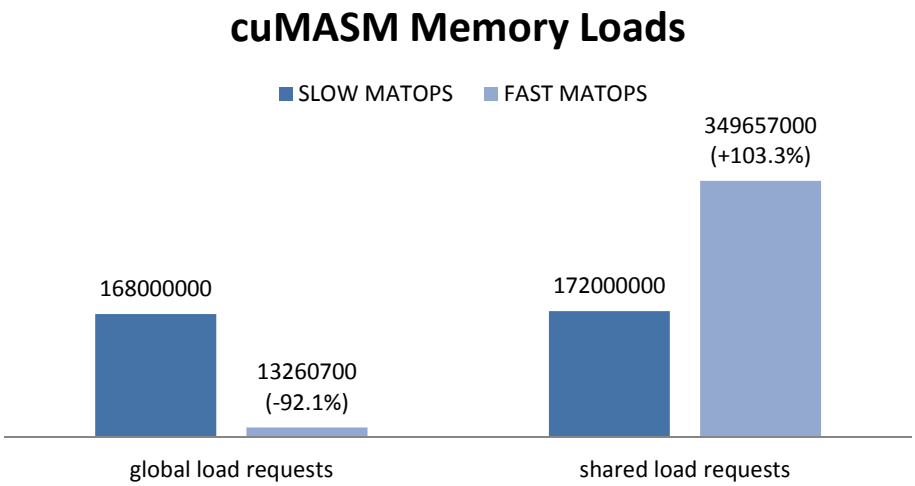


Figure 19: Global and shared memory load requests for naïve and cooperative matrix routines.

detector. However, for certain subsystems of our program (namely conversion from RGB to grayscale, Viola-Jones face-detection, and Sobel edge detection), execution time is a function of image size.

In (Fig. 20), the runtimes of the major constituent components of our cuMASM video stream landmarker are represented in an area chart. This figure clearly demonstrates that performance of cuMASM becomes limited by the performance of the face detector with large image sizes. Our cuMASM implementation maintains almost constant runtime (very weakly linear), while face detection has O(n) performance. For the high-resolution images, which contain the detail preferable for accurate landmarking and a host of other biometrics tasks, OpenCV's haar-cascade classifier implementation simply cannot detect faces in realtime.

In (Fig. 21), we plot the run-times of our GPU cuMASM implementation, and our single-threaded and multi-threaded CPU MASM implementations based on [17] as a function of image size. We show the raw data points demonstrating the variance in the execution time due to specific number of iterations required for each image to converge, as well as the mean execution times. Average execution time for cuMASM ranges from 37.96 ms (26.33 FPS) for VGA-sized images to 50.83 ms (19.67 FPS) for 2MP images. The full range of average execution times (MASM fitting only) can be found in (Table 1).

We are able to achieve GPU speedup factors ($Speedup = \frac{time_{cpu}}{time_{gpu}}$) of approximately 24X over a single-threaded CPU implementation of MASM and speedups of approximately 12X over a 8-threaded CPU implementation on a quadcore CPU. Multithreading MASM fitting on the CPU with OpenMP was only able to achieve a 2X speedup over the single-threaded implementation, despite the 4 physical cores available. The effect of our optimizations on the final runtime of cuMASM is quite dramatic, as shown in (Fig. 23). Reducing global memory access in our matrix operators facilitated a 2.75X speedup over our naïve implementation (reduced our run time ap-

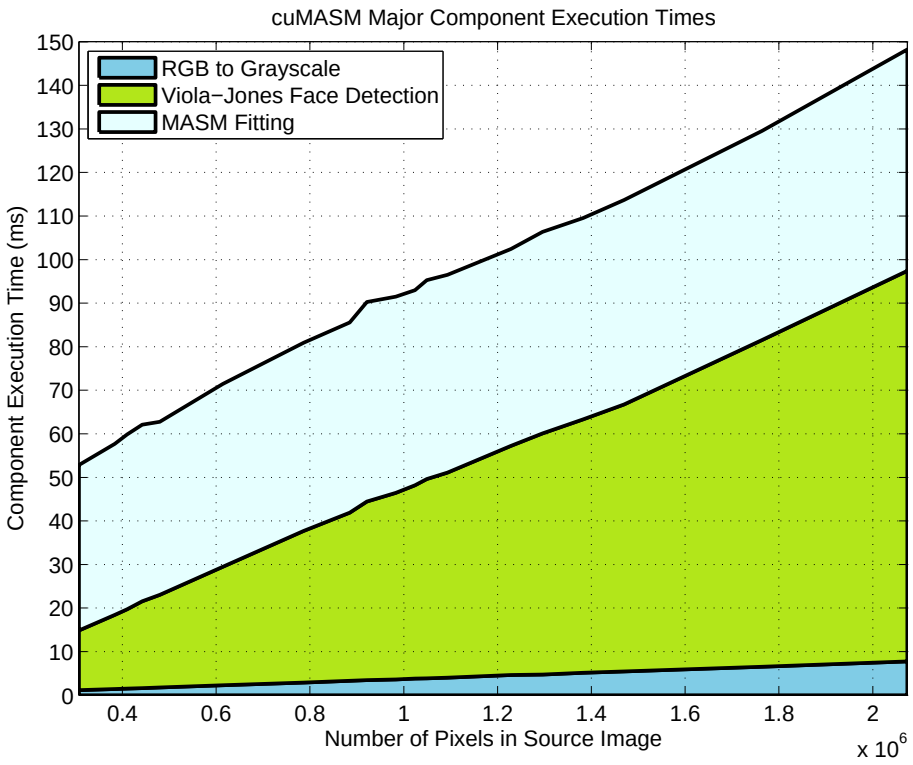


Figure 20: cuMASM landmarking major component contributions to execution times.

proximately by 63%) and was critical to achieving realtime performance.

5.1.1 CUDA Haar Cascade Face Detector

The OpenCV Viola-Jones face detector [22] whose execution time is represented in (Fig. 20) does not even perform a search through the entire scale space of the image; we restrict it to search for a minimum face size of 300×300. This limits the number of levels of the multi-resolution pyramid that must be traversed, and those levels which are eliminated are at the base of the pyramid which contains the most subwindows. While this is a reasonable limitation for our demonstration landmarking system (as our training images are of size 300×300), which assumes there is a single large face per frame, this is not acceptable in other use cases. For example, there may be a need to apply MASM to many faces in an image which are smaller than the training image size (although the lack of detail as one proceeds further down in scale will reduce fitting accuracy significantly). Additionally, there are many uses of haar cascades beyond facial landmarking which require one to traverse the entire scale space of an image, such as counting all the objects present in an image. Although not the focus of this paper, we have implemented a GPU haar cascade classifier in CUDA. Although

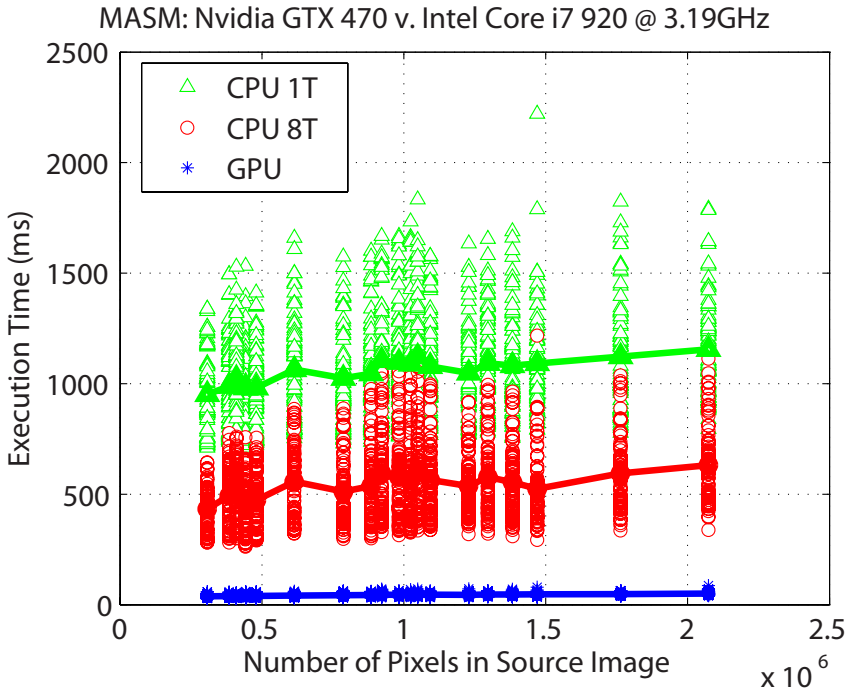


Figure 21: cuMASM execution times as a function of image size.

GPU-accelerated implementations of the often cited Viola-Jones object detector [24] and the de facto OpenCV implementation have been presented before [26, 27, 30, 28], only [29] releases the source code. Additionally, some of the GPU implementations are only partial components of the entire algorithm (notably much work has gone into fast integral image generation on the GPU, despite its relatively small contribution to overall runtime), and none mentions implementing the extended haar feature set proposed in [25] and implemented by OpenCV. Furthermore, the authors of [29] claim a very limited speedup of 2X improvement over OpenCV. However, a subsequent independent survey of vision algorithms on the GPU [31] was unable to duplicate this claimed performance and found that the code performed more slowly than OpenCV in nearly every case. Our implementation is compatible with standard OpenCV-trained haar classifier cascades packaged as XML files and includes support for the tilted features that are used in some OpenCV cascades. We achieve a speedup of approximately 8-12X over the single-threaded, standard (Windows x64 binary distribution) OpenCV 2.1 implementation running on a quadcore CPU. When we recompile the OpenCV 2.1 library with support for the Intel TBB and IPP libraries to enable full multicore utilization, we are still able to achieve a nearly 3X speedup. This is somewhat disappointing, but as our efforts to implement our own haar cascade classifier are still at an early stage of optimization, there is room for improvement. We want to stress that our goal is to provide a complete face detection and facial landmarking pipeline on the GPU so

Table 1: Mean execution times for CPU MASM and GPU cuMASM.

	Execution Time (ms)			GPU Speedup Factor	
Pixels	**MASM (1T)**	**MASM (8T)**	**cuMASM**	**MASM (1T)**	**MASM (8T)**
307,200	951.1	433.4	38.0	25.0	11.4
384,000	988.2	492.9	39.3	25.2	12.6
409,920	1024.0	506.3	40.1	25.5	12.6
442,368	982.0	462.7	40.5	24.2	11.4
480,000	981.3	472.5	39.8	24.7	11.9
614,400	1064.2	557.1	41.9	25.4	13.3
786,432	1024.3	512.3	43.2	23.7	11.9
884,736	1045.4	536.7	43.7	23.9	12.3
921,600	1105.1	594.0	45.8	24.1	13.0
983,040	1097.3	578.5	45.1	24.4	12.8
1,024,000	1089.0	562.4	44.8	24.3	12.6
1,049,088	1122.0	608.8	45.7	24.6	13.3
1,093,120	1078.0	564.5	45.4	23.7	12.4
1,228,800	1048.7	536.6	45.2	23.2	11.9
1,296,000	1091.9	577.9	46.2	23.6	12.5
1,382,400	1081.9	554.3	46.2	23.4	12.0
1,470,000	1091.4	524.7	46.9	23.3	11.2
1,764,000	1122.4	594.7	48.0	23.4	12.4
2,073,600	1155.8	631.8	50.8	22.7	12.4

as to leave CPU resources available for use by higher-level applications that may not be so data-parallel. Even relatively modest speedups relieve the CPU of an enormous computational burden.

Table 2: Haar cascade face detector execution times of OpenCV2.1 and our GPU implementation

	Execution Time (ms)			GPU Speedup Factor	
Pixels	**OpenCV(1T)**	**OpenCV(8T)**	**GPU Haar**	**OpenCV(1T)**	**OpenCV(8T)**
307,200	313.9	69.2	38.4	8.2	1.8
442,368	481.9	107.0	52.1	9.3	2.1
480,000	516.6	115.7	54.4	9.5	2.1
786,432	1007.3	228.9	80.8	12.5	2.8
884,736	985.9	217.1	84.5	11.7	2.6
1,024,000	1215.5	263.0	94.6	12.8	2.8
1,228,800	1464.1	331.5	123.3	11.9	2.7
1,470,000	1744.2	388.6	142.4	12.2	2.7

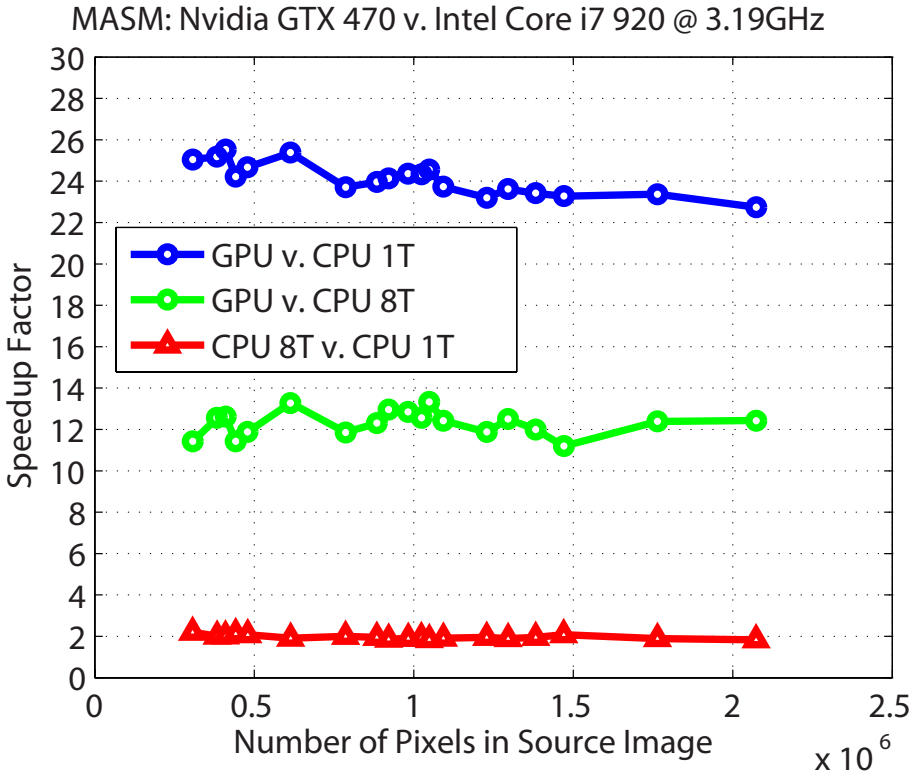


Figure 22: Average speedup factor of cuMASM GPU implementation.

5.2 Correctness

Due to the non-associative nature of floating point arithmetic, there is an expectation of some degree of error whenever a serial algorithm is parallelized. We evaluate the correctness of our GPU implementation by running both the serial CPU implementation and the GPU implementation over the JAFFE (Japanese Female Facial Expression) database, which contains 213 images of 10 subjects displaying 7 facial expressions. This database was previously used in [17] to evaluate performance as an unseen, challenging test set and we have replicated that setup in this paper. We compute the euclidean distance from each implementation's automatically annoted landmarks to hand-labeled ground truth points. The mean fitting errors for each landmark point averaged over the entire database are shown in (Fig. 26). Although the CPU implementation has a marginally lower fitting error for the majority of the landmarks, the root mean squared error (RMSE) over all of the points in the database is actually lower for the GPU implementation ($RMSE_{gpu} = 4.2773$ pixels) and $RMSE_{cpu} = 4.2828$ pixels).

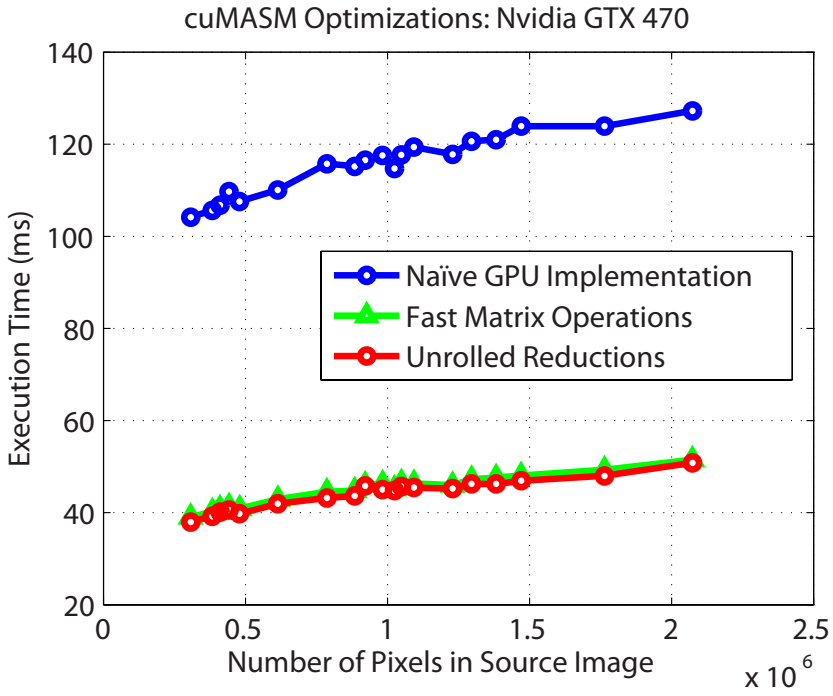


Figure 23: cuMASM runtimes with different levels of optimization applied.

6 Conclusions

Facial landmarking is an important component of many face-based biometrics systems. Robustness to pose, expression, and illumination is a difficult task which often necessitates computationally expensive algorithms such as MASM. By leveraging the massively parallel architecture of GPUs, we are able to convert what once was an offline algorithm to a real-time-capable one. This is an important achievement for biometrics applications, which often involve high stakes scenarios where vast amounts of streaming data must be processed as rapidly as possible. In the past, realtime performance on live video streams and even beyond realtime for rapid batch processing often required large data-centers. With GPU computing, it is possible to put a 10-TFLOP machine in a single ATX form factor PC using commodity components. We achieve frame rates of approximately 20 FPS. Although not discussed here, it is certainly possible to use multiple GPUs on a single machine or even multiple machines to parallelize this algorithm further. The easiest partitioning scheme for a video stream use case would be to simply assign each detected face on incoming frames to the next idle GPU.

We also demonstrate that awareness of the underlying GPU architecture is important for optimizations which can make a significant difference in final runtime performance. By leveraging shared memory, improved memory access patterns, and unrolled loops we improved our GPU performance by 2.75X over our Naïve GPU implemen-

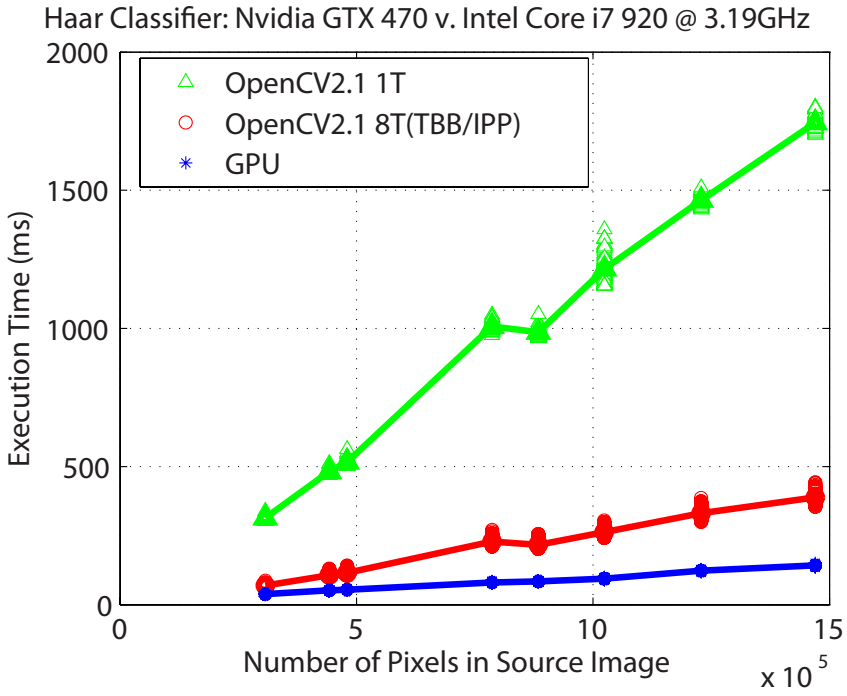


Figure 24: Haar cascade classifier face detector execution times as a function of image size.

tation. We achieve a speedup of approximately 12X over a multithreaded CPU implementation. As the fundamental difference in theoretical FLOP capacity between current CPU and GPU devices is approximately a factor of 20-25, this indicates that our implementation has some room for additional optimization. Like GPU implementations of many algorithms, the speed of our implementation is limited by memory access latency and total available memory bandwidth. Although we make extensive use of shared memory, as well as the constant and texture caches to minimize global memory access operations, we remain memory-bound.

Although not addressed in detail in this paper, at larger image sizes, initialization of MASM becomes problematic. Typically this initialization is performed using a face detector such as the popular Viola-Jones algorithm implemented in OpenCV; however, even with multithreading, the CPU-bound OpenCV implementation of face detection quickly becomes the limiting factor in a real-time landmarking system. Our own implementation of the haar-feature cascade classifier achieves approximately a 12X speed up over the stock binary distribution of OpenCV 2.1 on a quadcore CPU, and nearly 3X over OpenCV 2.1 recompiled from source using the multithreading TBB library, and Intel's proprietary IPP library. Full integration with our GPU-based face detector and/or a temporal tracking and prediction of future landmark initialization points allow cuMASM to operate on live video streams of HD video. An example application of such a system would be a continuous authentication system that is fully integrated on

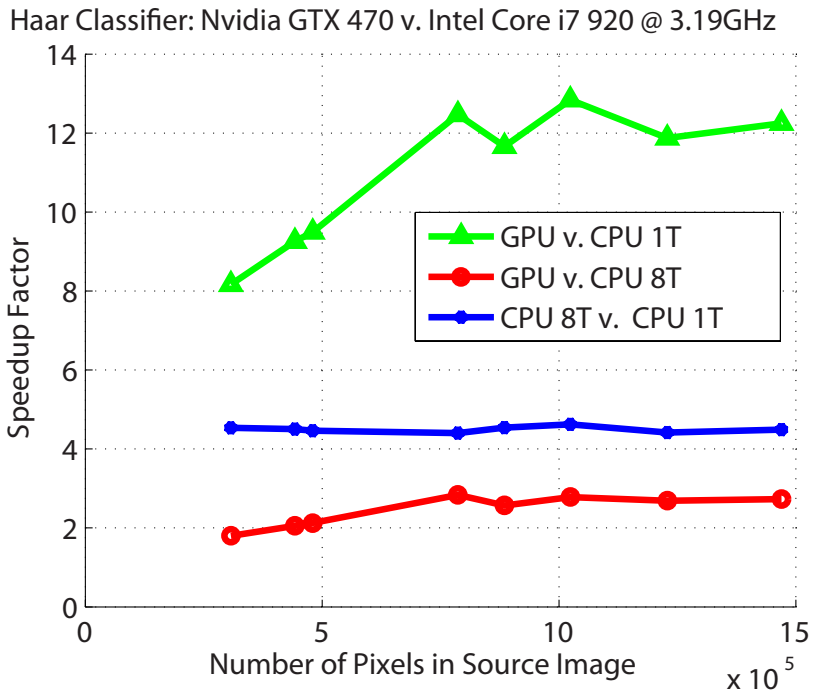


Figure 25: Average speedup factor of Haar cascade classifier face detector GPU implementation.

GPU and does not overtax CPU resources. This would allow continuous secure authentication of user via face in secure facilities without affecting the normal applications being used. This is an ideal application as in many such high-security applications, we can presume the graphics card is not really being utilized and is therefore available for use for such an important application.

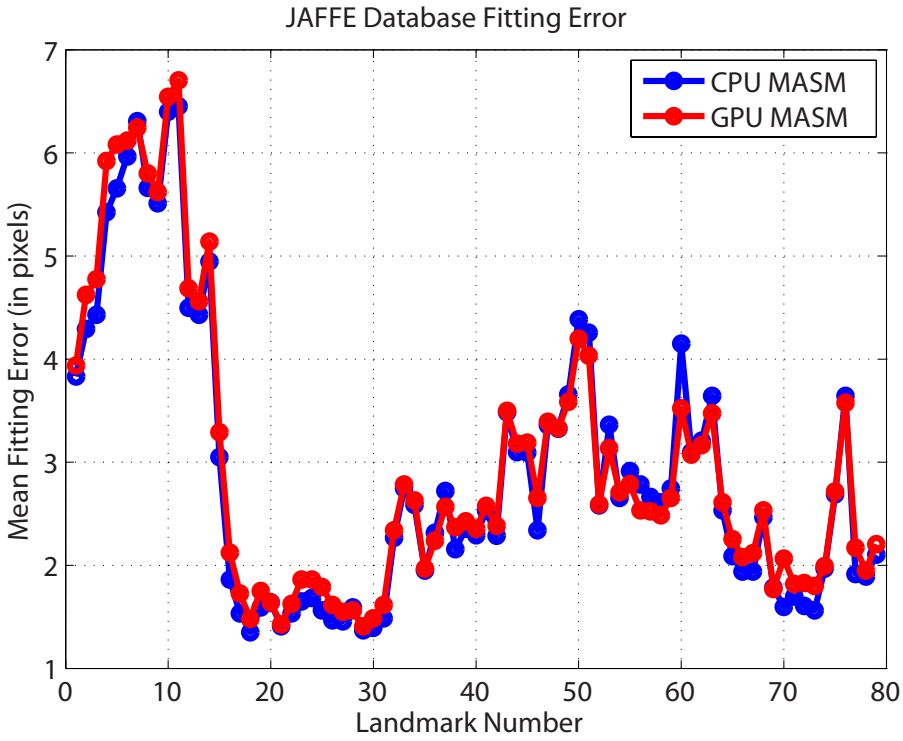


Figure 26: MASM and cuMASM average fitting error over JAFFE database

Figure 27: Sample images which have been automatically labeled with cuMASM.

References

[1] NVIDIA Corporation, "Nvidia's Next Generation CUDA Compute Architecture: Fermi". Whitepaper [Online]. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[2] NVIDIA Corporation, "NVIDIA CUDA Programming Guide", [Online]. Available: http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf.

[3] M. Ekman, F. Warg, and J., "An in-depth look at computer performance growth", *ACM SIGARCH Computer Architecture News 33*, 1, pp. 144-147, Mar 2005.

[4] Intel Corporation, "Intel SSE4 Programming Reference", [Online]. Available: http://softwarecommunity.intel.com/isn/Downloads/Intel%20SSE4%20Programming%20Reference.pdf.

[5] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, Nvidia Corporation, Published by Elsevier Inc. 2010.

[6] M. Harris, "Optimizing Parallel Reduction in CUDA". NVIDIA Corporation. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf.

[7] Michael Kass, Andrew Witkin and Demetri Terzopoulos, "Snakes: Active Contour Models", *International Journal of Computer Vision*, vol. 1, no. 4, pp. 321-331, January 1988.

[8] T. F. Cootes, C. J. Taylor, A. Lanitis, "Active Shape Models : Evaluation of a Multi-Resolution Method for Improving Image Search",*Proceedings of the British Machine Vision Conference*, pp. 327-336, 1994.

[9] T. F. Cootes, C. J. Taylor, D. H. Cooper and J. Graham, "Active Shape Models - Their Training and Application", *Computer Vision and Image Understanding*, vol. 61, no. 1, pp. 38-59, January 1995.

[10] T. F. Cootes and C. J. Taylor, "Statistical Models of Appearance for Computer Vision", Technical Report, Imaging Science and Biomedical Engineering, University of Manchester, March 2004.

[11] T.F. Cootes, G. Edwards and C.J. Taylor, "Comparing Active Shape Models with Active Appearance Models", *Proceedings British Machine Vision Confence*, pp. 173-182, 1999.

[12] Lu Huchuan, Shi Wengang, "Accurate Active Shape Model for Face Alignment", *Proceedings of the 17th IEEE International Conference on Tools with Artificial Intelligence*, pp. 642-646, November 2005.

[13] Wei Wang, Shiguang Shan, Wen Gao, Bo Cao, Baocai Yin, "An Improved Active Shape Model for Face Alignment", *Proceedings of the 4th IEEE International Conference on Multimodal Interfaces*, pp. 523-528, October 2002.

[14] Mohammad H. Mahoor and Mohamed Abdel-Mottaleb, "Facial Features Extraction in Color Images Using Enhanced Active Shape Model", *Proceedings of the 7th IEEE International Conference on Automatic Face and Gesture Recognition*, pp. 144-148, April 2006.

[15] Yan, S.C., Liu, C., Li, S.Z., Zhang, H.J., Shum, H.Y. and Cheng, Q.S., "Face alignment using texture-constrained active shape models", *Image and Vision Computing* v12. 69-75.

[16] Stephen Milborrow and Fred Nicolls, "Locating Facial Features with an Extended Active Shape Model", *Proceedings of the 10th European Conference on Computer Vision: Part IV*, pp. 504-513, October 2008.

[17] Keshav Seshadri and Marios Savvides, "Robust Modified Active Shape Model for Automatic Facial Landmark Annotation of Frontal Faces," *Proceedings of the 3rd IEEE International Conference on Biometrics: Theory, Applications and Systems (BTAS)*, pp. 319-326, Sep. 2009.

[18] Utsav Prabhu, Keshav Seshadri and Marios Savvides, "Automatic Facial Landmark Tracking in Video Sequences using Kalman Filter Assisted Active Shape Models," *Proceedings of the Third Workshop on Human Motion in Conjunction with the European Conference on Computer Vision (ECCV)*, Sep. 2010

[19] Jian Li, Yuqiang Lu, Bo Pu. Accelerating Active Shape Model Using GPU for Facial Extraction in Video, *IEEE ICIS*, 2009.

[20] Bo Pu, Shuang Liang, Yongming Xie, Zhang Yi, Pheng-Ann Heng, "Video Facial Feature Tracking with Enhanced ASM and Predicted Meanshift," *IEEE ICCMS*, 2010.

[21] J. C. Gower, "Generalized Procrustes Analysis," *Psychometrika*, vol. 40, no. 1, pp. 33-51, March 1975.

[22] Intel: Open Source Computer Vision Library, Intel, 2007.

[23] Michael J. Lyons, Shigeru Akamatsu, Miyuki Kamachi, Jiro Gyoba, Coding Facial Expressions with Gabor Wavelets, Proceedings of the Third IEEE International Conference on Automatic Face and Gesture Recognition, pp. 200-205, April 1998.

[24] Paul Viola, Michael Jones, "Rapid Object Detection using a Boosted Cascade of Simple Features," CVPR, vol. 1, pp.511, 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'01)

[25] Lienhart, R. and Maydt, J., "An extended set of Haar-like features for rapid object detection", ICIP02, pp. I: 900-903, 2002

[26] F. Jimin, "Speeding up boosted cascade of object detection using commodity graphics hardware", MS Thesis, National University of Singaport. 2009. [Online:] http://scholarbank.nus.edu.sg/bitstream/handle/10635/15790/Thesis_FENG_Jimin.pdf?sequence=1

[27] Johan A. Huisman. "High-speed parallel processing on CUDA-enabled Graphics Processing Units". MS Thesis, Delft University of Technology, 2010.

[28] Karl Berggre, and Pr Gregersson. "Camera focus controlled by face detection on GPU", MS Thesis. Lund University, 2008.

[29] Chuan-Heng Hsiao and Amy Dai. "Face Detection on CUDA", 2009. [Online:] http://www.cs264.org/2009/projects/web/Dai_Yi/Hsiao_Dai_Website/Project_Write_Up.html

[30] Daniel Hefenbrock, Jason Oberg, Nhat Tan Nguyen Thanh, Ryan Kastner, and Scott B Baden. "Accelerating Viola-Jones Face Detection to FPGA-level using GPUs". *2010 18th IEEE Annual International Symposium on Field Programmable Custom Computing Machines*, pages 1118, 2010. [Online:] http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5474075

[31] Tamas K. Lengyel, James Gedarovich, Antonio Cusano, Thomas J. Peters. "GPU Vision: Accelerating Computer Vision algorithms with Graphics Processing Units". Februray 2011. [Online:] http://c13software.com/downloads/GPUVision_2011.pdf