Faster Upper Body Pose Estimation and Recognition Using CUDA

Dane L. Brown[1], Mehrdad Ghaziasgar[1], James Connan[2]
Department of Computer Science
University of the Western Cape[1], Private Bag X17 Bellville, 7535, South Africa
Tel: + (27) 21 959-3010, Fax: + (27) 21 959-3006
and Department of Computer Science
Rhodes University[2], P O Box 94 Grahamstown, 6140
Tel: + (27) 46 603-8291, Fax: + (27) 46 636-1915
email: {2713985, mghaziasgar}@uwc.ac.za[1]; j.connan@ru.ac.za[2]

Abstract— Image processing techniques can be very time consuming when applied linearly on the Central Processing Unit (CPU). Many applications require processing to take place in real-time. The Upper Body Pose Estimation and Recognition system developed by Achmed and Connan has shown to be 88% accurate, but operates at less than real-time on the CPU. This paper proposes an adapted version of this algorithm, which runs on the Graphics Processing Unit (GPU) to achieve real-time processing speed. The system was found to achieve a slightly improved recognition accuracy of 92.95% while achieving on average a real-time processing speed of no less than 18 Frames Per Second (FPS) and a mean speed of 33.26 FPS.

Index Terms— Central Processing Unit, Frames Per Second, Graphics Processing Unit, Parallel Processing, Pose Estimation and Recognition.

I. Introduction

Upper Body Pose estimation and recognition determines and identifies the gestures performed by the arms, hands and other parts of the upper body of a human being.

The estimation and recognition of upper body poses has many applications such as determining suspicious behaviour in security systems and analysing sports injuries in medical systems. Another application is the focus of this research, namely, sign language recognition.

The South African Sign Language (SASL) research group at the University of the Western Cape is in the process of designing a machine translation system that can automatically translate between SASL and English [1]. A prototype of this system called iSign was created by the group. iSign is a digital phrase book that can translate simple phrases from SASL to English and vice versa using whole gesture recognition [1]. In order to be able to translate complex phrases, gestures need to be broken into their constituent parts which can be achieved by Upper Body Pose estimation and recognition.

Upper Body Pose estimation extracts semantic information about the sign language performed by a sign language speaker. This can be used to determine the meaning of the gestures in English.

Achmed and Connan created the current Upper Body Pose estimation and recognition system [2]. The system uses a web camera and computer vision techniques and is able to determine the motions of the arms of a signer with a high accuracy of 88%. The system is performed on the CPU without focusing on performance.

This paper discusses the use of parallel processing techniques using the GPU with OpenCV and Compute Unified Device Architecture (CUDA) to enhance the processing speed of the existing system.

Modern GPUs contain hundreds of cores and are capable of running millions of threads concurrently, far greater than typical modern CPUs, containing between 4 and 12 cores. An example is the NVIDIA 580GTX GPU which contains 512 cores. The use of parallel computing using the GPU can address the problem of speed in the existing Upper Body Pose estimation and recognition system, as well as in image processing in general.

The popular image processing library OpenCV has recently provided partial support for the CUDA API and image processing on the GPU. However, some functions in this library do not yet support this capability.

The CUDA Application Programming Interface (API) provides access to the GPU [3]. The porting of computer vision algorithms to the GPU using CUDA has shown a significant improvement in performance [4] [5].

The rest of this paper is organised as follows: Section II discusses related work in the field; Section III gives a background to CUDA; Section IV discusses the research methodology; Section V discusses the proposed implementation; Section VI discusses the testing process; Section VII shows the tables of results; Section VIII discusses the test results; the paper is concluded in Section IX.

II. Related Work

This section first discusses the Upper Body Pose estimation and recognition system developed by Achmed and Connan [2], which will be improved on this research. The subsection that follows discusses two studies that compare the performance of image processing techniques on the CPU and GPU.

A. The Current Upper Body Pose Estimation System

The existing system was designed using OpenCV [2]. It uses both an example-based and a learning-based method to recognise upper body poses. In this research, only the learning-based method is considered.

In Figure 1, the layout of the various components used

within the system is shown. The system uses the Viola-Jones algorithm, which uses Haar Classifiers, to determine the position of the face. The centre of the facial frame is usually situated on the nose. It was proposed that a person's nose colour is representative of their skin colour and can be used to perform skin detection to eliminate non-skin pixels in an image. This is combined with the learning-based background subtraction method, Gaussian Mixture Models (GMM), to obtain only moving skin pixels, that is, the arms. Morphological operations are used to enhance the image to prepare it for classification. The user is centred on the frame. It is resized and sent to a Support Vector Machine (SVM) which determines the position of the arms. The position is used to animate a 3D humanoid avatar in Blender.

The system was tested on 6 different signers, each performing 15 different SASL signs once. It was shown to achieve an average recognition accuracy of 88%. The system focused solely on optimising the recognition accuracy. It runs on a single thread. No tests were carried out to determine the processing speed of the system.

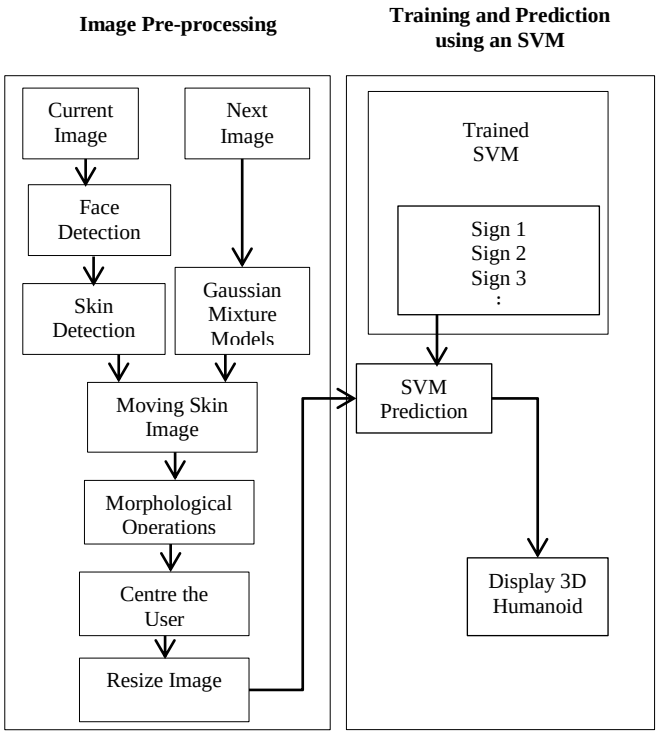

Figure 1: Current System

This algorithm was adapted and modified for use in the research described in this paper.

B. Comparison of Image Processing on the GPU and CPU

This subsection discusses how the GPU module in OpenCV, henceforth referred to as the GPU module, and the CUDA API can increase the run-time performance of image processing applications. It is shown that the performance of Gesture Recognition systems can be improved using both the GPU module and CUDA API. It is also shown that the performance of Gaussian Blur and Colour Conversion algorithms can be improved using only the GPU module.

1. Real-time Motion-based Gesture Recognition

Bayazit *et al.* created a real-time gesture recognition system using a motion-based algorithm to determine the movement of the arms, similar to the system discussed in this research. The CUDA API was used to increase the processing speed of an image classifier to achieve real-time speed [6].

The system is given a video input containing a user performing a gesture. Motion features are extracted from smoothed optical flow estimates. Viola-Jones face detection is used to normalize the frames by centring the user in each frame. Face detection and optical flow are executed on separate CPU threads to reduce the total computation time.

The face detection thread runs at a lower frame rate than the optical flow thread. An assumption is made that the face detection thread does not need to be synchronised to run in real-time with the rest of the system. This causes the system to drop frames. In such cases, the system can make use of a previously detected face, which is effective assuming minimal torso movement.

The resulting frame is resized to 30 x 40 pixels and sent to a classifier. The discriminative classifier, Adaboost, uses a subset of the motion features in the resulting frame to distinguish between gestures.

A test was performed on the classifier to determine the performance difference between running Adaboost on the GPU compared to on the CPU. It consisted of 512 to 8192 weak learners each sampled 30 times. The GPU implementation of Adaboost was found to be four times faster than the CPU implementation when using 8192 weak learners. The factor by which the speed increased was shown to be directly proportional to the number of weak learners.

The entire system was also tested using a dataset consisting of seven gestures, performed by ten different people. The types of gestures performed were: punch-left, punch-right, sway, wave-left, wave-right, waves, and idle. Each gesture was iterated a minimum of five times per recording.

A Canon GL2 camera, recorded these gestures at 29.97 FPS at a resolution of 720 x 480 pixels, to create the initial dataset. However, the processing did not take place at this resolution. The videos were resized to two different resolutions. The first set of tests was performed at a resolution of 320 x 240 pixels, while the second set was performed at a resolution of 160 x 120 pixels.

TABLE I
ACCURACY AND SPEED

Resolution	Accuracy (%)	Mean Speed (FPS)
160 x 120	86.7	38
320 x 240	87.3	20.7

Table 1 summarises the accuracy and average FPS at two pixel resolutions. In table I it is clear that the resolution affects the processing speed considerably, but the accuracy remains almost unaffected.

It was concluded that the CUDA framework could be used to increase the processing speed without having to drop frames in future.

The normalization method is similar to the method used in the proposed system. However, no frames are dropped in the proposed system. This guarantees that the user-centric frame will always be updated. This is necessary for gestures that require more torso movement than this system allows.

2. *Gaussian Blur and Colour Conversion*

Lengyel *et al.* performed a comparison of the processing speed of Gaussian Blur and Colour Conversion as performed on the CPU and using the GPU module [7].

Gaussian Blur is used in image processing for noise reduction and smoothing [8] and is used for post processing in the proposed implementation.

Colour Conversion is the process of converting one colour representation value to another. Examples of colour spaces are: HSV, RGB and greyscale [8].

A test was conducted to determine the difference in processing speed when using the CPU as compared to using the GPU module. The computer hardware used consisted of the NVIDIA 9800GTX GPU and the Intel(R) Core(TM) 2 duo 6400 CPU running at a clock speed of 2.13GHz.

The test ran on a 1280x1024 input image, which was processed with the Colour Conversion algorithm. The Gaussian blur algorithm was applied on the resultant image. This procedure was performed 144 times. Each time a frame was processed, the CPU transferred the frame to the GPU memory for processing [3]. The computation speed was found to be 270% faster on the GPU than on the CPU when excluding the transfer time. However, including the transfer time, the GPU was only 1.6 % faster than the CPU. This observation was used to significantly speed up the proposed system.

III. THE CUDA FRAMEWORK

This section provides a background on the CUDA framework, discusses the software and hardware view, and discusses how it forms part of the rest of the CPU framework in the proposed implementation.

Compute Unified Device Architecture (CUDA) is a parallel programming model and software environment that provides general purpose programming on the GPU. The hardware inside a GPU device consists of a collection of multiprocessors. These multiprocessors operate on the Single Instruction Multiple Data architecture – concurrent execution of the same program instruction on different data. The multiprocessors communicate through the shared device memory, common to every processor core [3].

The software side of the CUDA programming model extends the C/C++ programming language. NVIDIA Performance Primitives is an example of an image processing library for CUDA that extends the C/C++ programming language [9]. From a CUDA programmer's perspective, the computing system consists of host code and device code running on the CPU and GPU respectively. The CPU is called the host and the GPU is called the device. The sections of code that exhibit little or no data parallelism are implemented in host code. When there are sections of code which exhibit a rich amount of data parallelism, they are copied to device memory and implemented in device code.

Device code is structured into kernels on the host to communicate on a low hardware level, illustrated in Figure 1. A kernel call is issued on the host to execute a portion of an application on the device that can: be executed many times, be isolated as a function and work independently on different data. The result is an initially sequential program that gets executed in many data independent parallel threads in device memory. Memory allocations are automatically controlled by the GPU module on the device. The CUDA API requires the programmer to manually specify these memory allocations.

The multiprocessors contain hardware thread contexts, which are executed as a group of threads, called a warp, on the multiprocessors in a lock-step manner.

A warp contains 32 threads and is controlled at a hardware level. However, in image processing and other general purpose programming applications, more threads might be required in device memory. In Figure 2, blocks are used to group as many as 1024 threads. Many warps are assigned to a block and controlled in this lock-step manner by a hardware scheduler in order to hide memory latencies and pipeline stalls by intelligently switching between different warps [3] [9]. The result of the computation is sent back to host memory.

However, this introduces a memory transfer overhead, which is especially noticeable when the transfer occurs within loops of code [9]. This overhead can be minimized by the performance gained when performing a large number of computations on the device, before transferring the result back to the host. In the proposed implementation this overhead is removed by loading all the video frames into device memory before computation during image processing. This solution was used on both the GPU module and the CUDA API.

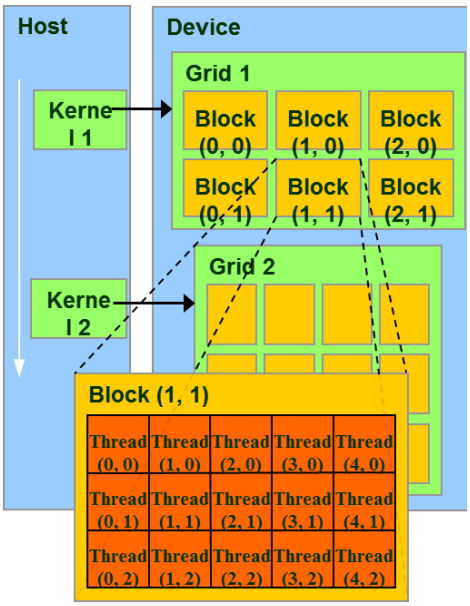

Figure 2: CUDA Framework.

IV. RESEARCH METHODS

1. The algorithm used in the current system was modified to improve the accuracy.
2. The components required in the algorithm were ported to the GPU to increase the overall processing speed and was expected to achieve a real-time performance of 15 FPS.
3. The accuracy was evaluated by comparing each frame in the input image to the corresponding frame of the 3D humanoid.

V. IMPLEMENTATION

This section explains the implementation of the system

designed in this research. Figure 3 provides an overview of the proposed implementation, which is an adaptation and modified version of the algorithm in Figure 1.

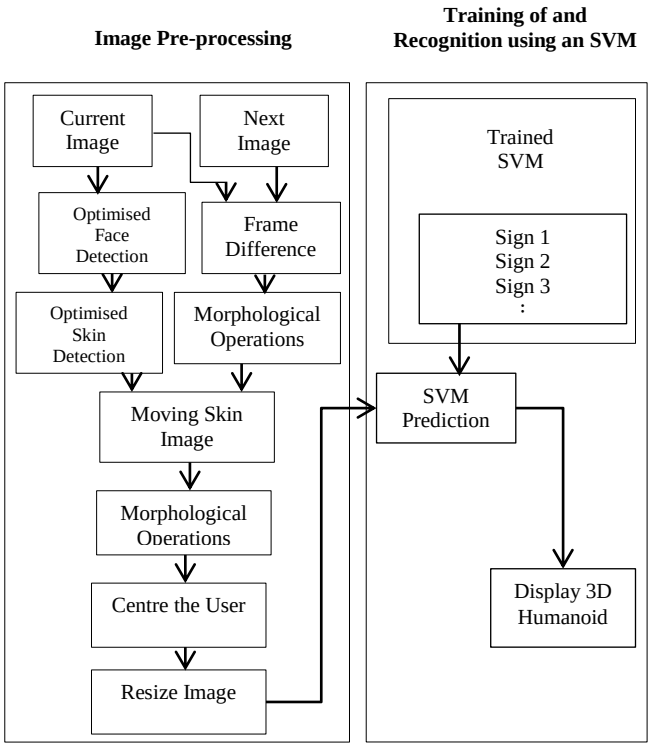

Figure 3: Proposed Implementation.

The following subsection contains an optimised skin detection algorithm that uses the face and nose as a reference.

A. Optimised Skin Detection using the Face and Nose as a Reference

An adaptive skin detection algorithm was used. It first detects and tracks the face. This was done by using the Viola-Jones face detector method from the GPU module. The face detector method was modified to only search for the biggest face and for face sizes in the appropriate range. This is used to reduce the processing overhead and to reduce the chance of finding a false positive. The face area is set as the Region of Interest (ROI). It is observed from Figure 4(a) that the nose area can be obtained by taking the centre of this ROI. The nose area contains the skin colour distribution, later used in this implementation.

To determine the skin colour, the original image is converted to the HSV colour space. The GPU module's enhanced colour space conversion method was used. The Hue contains the skin tone values. The distribution of the skin pixels is determined by computing a histogram of the Hue values.

(a) **(b)**

Figure 4: Face and Skin image.

The GPU module is used to calculate the histogram. Histograms are made up of bins. Each bin represents a particular intensity value range. The histogram is calculated by assigning each pixel to a bin depending on the pixel intensity. The number of bins used can affect the signal-to-noise ratio of the image. In this research the number of bins used in the histogram was changed from 16 to eight.

The final step for skin detection is to calculate the back projected image from the histogram. The resultant image contains the face and arms determined by the skin colour distribution of the user. The back projected image is shown in Figure 4(b). The GPU back projection functionality was added using the CUDA API.

B. Frame Difference and Morphological Operations

In the proposed implementation, GMMs are replaced by frame differencing to remove the background. Frame differencing was found to be more effective than GMMs. It requires no training and is more robust to dynamic lighting conditions when using a low threshold value [10]. However, the low threshold causes the image to lose some detail, which is regained by using the morphological operations - closing and dilating, in the GPU module. Closing fills black pixel areas that are surrounded by neighbouring white pixels with the value of the neighbouring pixel [8]. Dilating enlarges the white pixel areas, which represent the arms, to obtain a clear foreground image [8].

The skin image is mapped on top of the foreground image to obtain an image containing only the arms. This image only contains the skin pixels that are moving, as seen in Figure 5(a).

This process introduces additional noise into the resulting image. A morphological operation called Erosion is used to remove noise [8] denoted by the circle in Figure 5(b). In Figure 5(b), Closing and Dilating is reapplied to further enhance the image before it is centred. The coordinates of the nose, obtained in section A, are used to centre the user in each frame.

The resulting image is resized to 30 x 40 pixels. Achmed and Connan used an external program called Convert from the ImageMagick library to resize the image. This caused a significant overhead and was left out of the proposed system. The proposed system replaced Convert with the GPU module functions: Resize, Pixel Interpolation and Gaussian blur. Pixel Interpolation is applied to minimise the loss of detail due to the resizing of the image. The Gaussian blur algorithm is used to remove the noise from the Interpolation process. The proposed method provides comparable results without the significant overhead of Convert. The resized image is sent to the SVM for classification.

(a) **(b)**

Figure 5: Results of Frame Difference and Morphological Operations.

C. Classification and Displaying the 3D Humanoid

SVMs are a set of supervised learning methods derived from statistical learning theory [11]. In this research the CUDA SVM library is used to increase the processing speed of the system developed by Achmed and Connan. A multi-label SVM is used to map the coordinates to each joint of the 3D humanoid to the pixel coordinates of the SVM data file.

The 3D humanoid [12] in Blender moves its arms according to the pixel coordinate within the 30 x 40 image as seen in Figure 6.

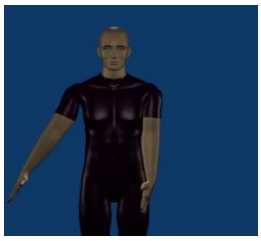

Figure 6: Blender 3D Humanoid [12]**.**

VI. Testing

Testing was performed to evaluate and compare the accuracy and the processing speed of the proposed system.

The accuracy testing was performed to ensure that the proposed algorithm provides a sustained or improved accuracy to the original system. Performance testing was carried out on the modified algorithm to evaluate the processing speed and whether real-time performance was achieved.

In order to perform these tests four systems were produced. Two of which were the algorithm developed by Achmed and Connan and two of which were the proposed algorithm. These systems are described below.

1. The original system using the algorithm of Figure 1, running on the CPU, henceforth referred to as OrigCPU.
2. The original system using the algorithm of Figure 1, running on the GPU, henceforth referred to as OrigGPU.
3. The system using the proposed algorithm of Figure 3, running on the CPU, henceforth referred to as ModCPU.
4. The system using the proposed algorithm of Figure 3, running on the GPU, henceforth referred to as ModGPU.

The machine used to carry out these tests was an Intel 2600k 3.8 GHz quad core CPU with the NVIDIA 580 GTX CUDA-enabled GPU.

Test data was collected from six subjects. Each subject was asked to stand such that their arms and head were within the boundaries of each frame. They performed 14 signs once, which resulted in a total of 84 videos. The 14 signs are listed in table II and table III.

The following subsections describe the procedure used to perform accuracy testing and performance testing.

A. Accuracy Testing

Each video was given as an input into each of the four systems. The input frames of each sign were compared with the resulting output frames displayed by the 3D humanoid in Blender and the average of the matches was recorded. The average was also taken for all the signs per system. The results for OrigCPU, OrigGPU, ModCPU and ModGPU are summarised in table II.

B. Performance Testing

The modified algorithm's performance was tested in order to identify the difference in processing speed between the CPU and GPU. Each video was given as an input into ModCPU and ModGPU. In each case the time was recorded for the system to process one frame. The total number of the frames was divided by the total time to obtain the FPS for each sign. The average FPS of all the signs was recorded. The minimum FPS was also recorded, which is the lowest processing speed registered for each sign. Table III summarises the minimum speed and mean speed results for ModCPU and ModGPU.

VII. Test Results

TABLE II
Accuracy Testing of OrigCPU, OrigGPU, ModCPU and ModGPU

Sign	**OrigCPU**	**OrigGPU**	**ModCPU**	**ModGPU**
Away	90.61	91.27	93.62	94.28
Bye	90.60	89.87	98.24	98.24
Cracker	79.75	79.75	86.88	84.80
Curtains	85.99	85.99	95.75	95.75
Dress	90.91	90.91	93.46	93.46
Eat	91.35	90.42	94.95	94.03
Left	93.31	94.82	92.32	97.29
Light	94.98	94.19	98.51	97.75
Love	92.68	92.68	97.20	94.64
Right	81.84	83.51	95.38	95.38
Run	85.03	85.03	87.88	87.88
We	88.63	87.52	87.15	87.15
Wide	77.94	79.79	89.91	92.80
Why	85.54	83.88	87.80	87.80
Average	**87.80**	**87.83**	**92.79**	**92.95**

TABLE III
Minimum Speed and Mean Speed in FPS for ModCPU and ModGPU

Sign	**Minimum Speed (FPS)**		**Mean Speed (FPS)**	
	ModCPU	**ModGPU**	**ModCPU**	**ModGPU**
Away	6.87	18.34	21.98	40.92
Bye	6.99	19.18	16.09	32.35
Cracker	6.96	18.11	18.01	33.61
Curtains	6.79	18.26	11.47	25.55
Dress	7.03	18.12	17.28	31.93
Eat	7.01	18.74	14.09	27.96
Left	7.12	19.03	15.26	30.52
Light	7.09	18.24	17.19	33.31
Love	6.91	18.78	17.20	34.42
Right	6.99	18.56	13.55	28.66
Run	6.73	18.19	14.26	27.53
We	6.77	18.44	22.09	39.10
Wide	6.98	18.59	20.98	36.80
Why	7.01	18.73	19.11	37.14
Average	**6.95**	**18.52**	**17.04**	**33.26**

VIII. Discussion

A. Accuracy Testing

Table II illustrates that the two systems using the modified algorithm, shown in Figure 3, achieve improved average accuracies than the two systems that use the original algorithm. The modified algorithm attained a 5% higher accuracy than the original algorithm.

Contrary to expectation it is observed that there are minute differences in the accuracies of each algorithm running on the CPU and GPU. The expectation was to achieve the exact same accuracy for ModCPU and ModGPU and also for OrigCPU and OrigGPU. The difference in accuracy is attributed to the lack of floating point precision in the CUDA API [3]. Many of the techniques used in the pre-processing procedure of both algorithms perform mathematical computation, such as Gaussian blur and greyscaling. The loss in precision during these computations leads to slight variations in the pre-processing procedure. The precision of the CUDA API is under active development.

B. Performance Testing

Referring to table III, it is observed that the ModGPU always achieves a frame rate of real-time, with the minimum speed never falling below 18 FPS. This is a substantial improvement on ModCPU which achieves minimum frame rates of less than real-time for every sign. In fact it is observed that the minimum frame rate of ModGPU is comparable to the average frame rate of ModCPU. Overall, ModGPU achieves a frame rate that is approximately three times that of ModCPU. This encourages the use of the GPU to substantially improve the speed of the image processing algorithm.

IX. Conclusion

This paper proposed an improved approach to Upper Body Pose Estimation and Recognition using the CUDA Framework on the GPU. The focus of the approach was on providing real-time speed with a sustained accuracy. A modified algorithm of the current Upper Body Pose Estimation and Recognition System was also proposed. These two approaches were combined to create the proposed system, ModGPU, which achieved an improved accuracy of 92.95% and a processing speed of faster than real-time performance.

References

[1] M. Ghaziasgar and J. Connan, "Investigating the Intelligibility of Synthetic Sign Language Visualization Methods on Mobile Phones," University of the Western Cape, South Africa, 2010.

[2] I. Achmed, "Upper Body Pose Estimation towards the translation of South African Sign Language," University of the Western Cape, South Africa, MSc Thesis, 2010.

[3] D. Kirk and W. Hwu, *Programming Massively Parallel Processors.*: Morgan Kaufmann, 2010, pp. 16-51.

[4] J Fung and S Mann, "Using graphics devices in reverse: GPU-based Image Processing and Computer Vision," in *IEEE International Conference on Multimedia & Expo*, 2008.

[5] D. Hefenbrock, J. Oberg, and N. T. N. Thanh, "Accelerating viola-jones face detection to fpga-level using gpus.," in *IEEE Annual International Symposium on FieldProgrammable Custom Computing Machines*., 2010, pp. 11-18.

[6] M. Bayazit, A. Couture-Beil, and G. Mori, "Real-time Motion-based Gesture Recognition using the GPU," in *IAPR Conference on Machine Vision Applications*, Japan, 2009.

[7] T. K. Lengyel et al., "GPU Vision:Accelerating Computer Vision algorithms with Graphics Processing Units," 2011.

[8] G. Bradski and A. Kaehler, *Computer Vision with the OpenCV Library*, Press Release ed.: O'Reilly Media, 2008.

[9] J Sanders and E Kandrot, *CUDA by Example*, 1st ed.: Addison-Wesley, 2011, pp. 38-56.

[10] Y. Sun et al., "From GMM to HGMM: An Approach in Moving Object Detection," *Computing and Informatics*, vol. 23, pp. 215–237, 2004.

[11] V. Vapnik, *The Nature of Statistical Learning Theory*, 2nd ed.: Springer, 1999.

[12] D. Van Wyk, "Virtual Human Modelling and Animation for Sign Language Visualisation," University of the Western Cape, MSc Thesis, 2008.

Dane Brown is currently an M.Sc student at the University of the Western Cape. He is currently doing research on sign language synthesis and novel communication applications on mobile interfaces for the Deaf and hearing impaired.

Mehrdad Ghaziasgar is the project manager of the South African Sign Language (SASL) research group. His interests are internet programming, mobile computing, computer vision and machine learning.

James Connan heads up the South African Sign Language (SASL) research group. His interests are computer vision and machine learning.