Software Acceleration of the Deformable Shape Tracking Application

How to eliminate the Eigen Library Overhead

NIKOS PETRELLIS*
Electrical and Computer Engineering Department, University of Peloponnese, Patra, Greece

STAVROS ZOGAS
Electrical and Computer Engineering Department, University of Peloponnese, Patra, Greece

PANAGIOTIS CHRISTAKOS
Electrical and Computer Engineering Department, University of Peloponnese, Patra, Greece

PANAGIOTIS MOUSOULIOTIS
School of Informatics, Aristotle University of Thessaloniki, Thessaloniki, Greece

GEORGIOS KERAMIDAS
School of Informatics, Aristotle University of Thessaloniki, Thessaloniki, Greece

NIKOLAOS VOROS
Electrical and Computer Engineering Department, University of Peloponnese, Patra, Greece

CHRISTOS ANTONOPOULOS
Electrical and Computer Engineering Department, University of Peloponnese, Patra, Greece

Shape tracking is based on landmark detection and alignment. Open-source code and pre-trained models are available for an implementation that is based on an ensemble of regression trees. The C++ Deformable Shape Tracking (DEST) implementation of face alignment that is using Eigen template library for algebraic operations is employed in this work. The overhead of the C++ Eigen library calls is measured and selected computational intensive operations are ported from Eigen implementation to custom C code achieving a remarkable acceleration in the shape tracking application. An important achievement of this work is the fact that the restructured code can be directly implemented with reconfigurable hardware for further speed improvement. Driver drowsiness and distraction detection applications are exploiting shape tracking by measuring landmark distances in order to detect eye blinking, yawning, etc. Fast video processing and accuracy is mandatory in these safety critical applications. The modified software implementation of the original DEST face alignment method presented in this paper, is almost 250 times faster due to the custom implementation of computational intensive vector/matrix operations and rotations. Eigen library is still used in non-time critical parts of the code for compact description and higher readability. Flattening of nested routines and inline implementation is also used to eliminate excessive argument copies and data type checking and conversions.

CCS CONCEPTS • Computing methodologies • Machine learning • Machine learning approaches • Classification and regression trees

Additional Keywords and Phrases: Face Alignment, Deformable Shape Tracking, Eigen, Acceleration, Hardware Implementation

* Contact person, email: npetrellis@uop.gr.

ACM Reference Format:

Nikos Petrellis, Stavros Zogas, Panagiotis Christakos, Panagiotis Mousouliotis, Georgios Keramidas, Nikolaos Voros, Christos Antonopoulos. 2021. Software Acceleration of the Deformable Shape Tracking Face Alignment Application Compensation of the Eigen Library Overhead. In 2nd Symposium on Pattern Recognition and Applications, Nov 06-08, 2021, Larissa, Greece. ACM, New York, NY, USA, 10 pages.

Figure 1: Face alignment based on our application with 68 landmarks. Video frame source from [1].

1 Introduction

A number of landmarks can be used to perform face alignment. Different landmarks determine the shape of the face or more specifically the chin, the mouth, the eyes, the eyebrows, etc., as shown in Fig. 1. The number of landmarks typically used can range from less than 10, if only a specific part of the face has to be detected or aligned (e.g., mouth, eyes), to more than 160, if the detailed face shape has to be drawn. As expected, the use of a large number of landmarks requires a bigger training set and longer training period to achieve a reasonable accuracy. The distance of specific landmarks is used in metrics such as Eye Aspect Ratio (EAR) [2] or Percentage of Eye Closure (PERCLOS) [3] to detect eye blinking, yawning, facial expressions, orientation of the face etc. Face alignment is exploited by applications that detect driver drowsiness, user reaction to specific stimulus, emotional reactions, etc.

Face alignment can be efficiently performed using an Ensemble of Regression Trees (ERT) as described in the fast 2D facial landmark detection algorithm presented by Kazemi and Sullivan in [4]. This method focuses on a small number of predetermined pixel positions from the input image. The gray scale intensities of these pixels is examined and ERT training is performed based on gradient boosting, optimizing the sum of square error loss. The regression procedure is expressed as a sparse coding problem in [5] and additional dictionaries have been used e.g., for the reconstruction of face shapes partially occluded. The authors of [6] use an ERT of 500 binary trees of 31 nodes and depth=5. This size of ERT is also used in our case. In [7], a 2D landmark alignment method detector is presented, based on ERT balancing accuracy and speed. An ERT is also employed in [8] for the estimation of the head pose with a latency of 1ms per image frame. This is the speed advertised in [4], too. Implementations of the face alignment method described in [4] are available in the C++ DLIB library [9] and the DEST repository [10]. Five experiments with different combinations of face detection failure, model free tracking etc, have been conducted in [11] comparing a large number of face tracking approaches. Several important issues are also examined in [12] including acceleration techniques, jitter in face tracking, and appropriate evaluation metrics.

The source code of the DEST repository [10] has been transformed in the framework of this paper, to develop a high performance shape tracking application that can also be implemented in hardware with Field Programmable Gate Arrays (FPGAs). The original DEST source code has been developed in C++ with calls to the Eigen template

library [13] for matrix/vector operations, Jacobi rotations, Singular Value Decomposition (SVD), etc. The use of Eigen library and well defined C++ classes ensures the integrity of the data values in any application and allows the operations to be described in a compact, still portable, way. Nevertheless, the excessive integrity checks and data type conversions are responsible for a high latency overhead. Moreover, the Eigen and C++ classes and data types used in DEST do not allow a reconfigurable hardware synthesis by state-of-the-art tools such as Xilinx Vitis. For instance, every data structure in the original DEST implementation is dynamically allocated while the size of the matrices used in hardware kernels has to be defined as a constant. Although the original DEST application supports vectorization for Graphic Processing Units (GPUs) implementations, the frame processing time needed on a Central Processing Unit (CPU) is more than 116 ms, which is much longer than the 1 ms advertised in [4]. Frame processing execution times on various platforms (Intel, AMD, ARM processors) can also be found in [12].

The original source code of the DEST repository had been tested in Windows and Mac OS X operating systems. The DEST video tracking application has been ported and tested on Ubuntu 18.04 operating system as a part of this work. Common GNU C++ compilers have been employed that are also used in hardware design tools (Xilinx Vitis). In this way, the hardware synthesis of computational intensive operations and consequently the porting to hardware platforms is feasible. Profiling of the original DEST video tracking application revealed that the most computational intensive parts is the Tracker::predict() routine that estimates the position of the landmarks in the current warped video frame (described in detailed in the next sections). Therefore, the code of the Tracker::predict() routine and the functions called in a nested manner from this top level routine are flattened and converted from C++ to C. As a result, the latency of the modified predict() routine has been improved by a factor of 240. More specifically, this latency was reduced from 116ms to 479us measured on the same CPU.

The paper is organized as follows. The concept of the face alignment method presented in [4] and the landmark distance dependent parameters used in driver drowsiness applications are presented in Section 2. The original and modified source code structure are described in Section 3. The experimental results are presented in Section 4 while the conclusions and future work are discussed in Section 5.

2 Face alignment method and applications

Let *LM* be the number of face landmarks. In the DEST implementation, LM is equal to 68. The shape $S \in R^{2*LM}$, is defined as a set of *LM* landmarks: $S = \{x_0, x_1, .., x_{LM}\}$. Each member x_i of the S set is a pair of coordinates. The initial estimated shape $\hat{S}^{(0)}$ is the mean shape retrieved from the trained model. In this model, a set P_r consisting of N_c reference pixels associated with their closest landmark have also been defined. In the current image frame, the algorithm presented in [4], attempts to locate the pixels that correspond to the ones defined in P_r, based on their intensity in gray scale. The cardinality N_c of P_r is much smaller than the number of pixels in the original image frame, thus the processing time is significantly reduced since only a small subset of pixels is examined. In the trained model, T_{cs} regressors are defined, implemented as T_{cs} cascade stages. The estimated shape at the cascade stage t is denoted as $\hat{S}^{(t)}$ (t=1,.., T_{cs}). The estimated shape at stage *t+1* is the one of stage t corrected by a factor r_t that also depends on the current image I_{π_i}:

$$\hat{S}^{(t+1)} = \hat{S}^{(t)} + r_t(I_{\pi_i}, \hat{S}_i^{(t)}) \quad (1)$$

The r_t correction factor is based on the intensities of the P_r pixels and each one of these pixels is associated with the closest landmark. A gradient tree boosting algorithm is used to train a regressor with a sum of square error loss [12]. The training set consists of N images $I_{\pi_i}, 0 \leq \pi_i < N$ and $\hat{S}_i^{(t)}$ is the shape estimated by all training images I_i with $i \neq \pi_i$. The residual $\Delta S_i^{(t)}$ in the regressor r_t is estimated as the difference between the shape of the specific I_{π_i} image (S_{π_i}) and the mean shape estimated from the rest of the training images ($\hat{S}_i^{(t+1)}$) : $\Delta S_i^{(t+1)} = S_{\pi_i} - \hat{S}_i^{(t+1)}$. In each regressor t, K binary trees are visited. Each binary tree has 31 nodes and depth equal to 5. This means that 5 of the 31 tree nodes are visited, following a path from the tree root to a leaf. In each node of the k-th tree (k=1, ..., K), the intensity of a predefined pair of pixels that belong to P_r, is compared in order to decide

the direction in the tree that has to be followed (left or right). In this way, the reference P_r pixels are mapped to the corresponding pixels in the image under test. It can be stated that each one of the K binary trees is used to map a small subset of P_r pixels to the corresponding ones in the current image. This mapping can be viewed as a warping of the current image to fit the mean shape and the P_r pixels associated with the landmarks of the mean shape [12]. For each node of the tree, the following information is retrieved from the trained model: a) the indices of two pixels $p_1, p_2 \in P_r$, b) the threshold T_h that determines the next node that will be visited if the intensity difference between p_1 and p_2 is higher or lower than T_h and c) a weak regression function g_k.

The regressor r_{ik} for the k-th tree and the i-th training image (i=1,..., N) is updated as a function of the strong regressor f_k as follows:

$$r_{ik} = \Delta S_i^{(t)} - f_{k-1}(I_{\pi_i}, \hat{S}_i^{(t)}) \quad (2)$$

The initial strong regressor f_0 is initialized as:

$$f_0(I, \hat{S}^{(t)}) = \underset{\gamma}{\operatorname{argmin}} \sum_{i=1}^{N} \left\| \Delta S_i^{(t)} - \gamma \right\|^2 \quad (3)$$

and it is updated in the k-th tree as follows:

$$f_k(I, \hat{S}^{(t)}) = f_{k-1}(I, \hat{S}^{(t)}) + lr \cdot g_k(I, \hat{S}^{(t)}) \quad (4)$$

The shrinkage factor $lr<1$ is used to avoid overfitting. The weak regressor g_k is retrieved from the leaf of the k-th binary tree. Eventually, r_t is set to f_K. Warping is performed in the DEST implementation by a process called Similarity Transform (ST). If q is a P_r pixel and its closest landmark has index k_q, their distance δx_q is estimated as $\delta x_q = \left\| q - x_{k_q} \right\|$. If s_i and R_i are the scale and rotation matrices used in the ST to warp the initial shape of the current image, the pixel q' in the current image I that corresponds to q is estimated by:

$$q' = x_{i,k_q} + \frac{1}{s_o} R_i^T \delta x_q \quad (5)$$

The minimization of the mean square error in the estimation of the q' value, can be used to select the optimal s_i and R_i values.

The output of the DEST shape tracking algorithm is $\hat{S}^{(T_{cs})}$ i.e., the final landmark position coordinates. This information can be exploited by higher level applications. For example, a driver drowsiness detector can monitor e.g., whether the eyes of the driver are closed or if he is yawning. Two parameters are defined for monitoring the eye but they can also be extended to monitor the mouth [2][3]: the EAR and PERCLOS. The EAR parameter can be used to detect if the eye is open or closed.

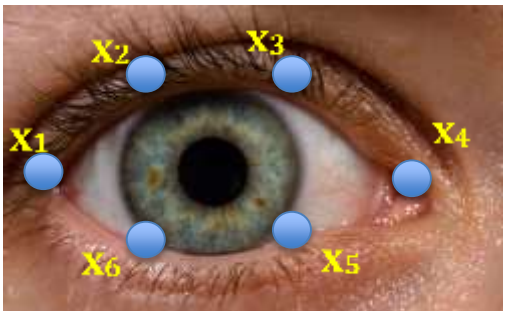

Figure 2: Eye landmarks.

If the landmarks x_1-x_6, determine the shape of an eye as shown in Fig. 2, then EAR is defined as:

$$EAR = \frac{\|x_2 - x_6\| + \|x_3 - x_5\|}{2\|x_1 - x_4\|} \quad (6)$$

EAR values higher than a threshold mean that the distance between x_2, x_6 and x_3, x_5 is relatively high compared to the distance between x_1, x_4 thus, the eye is open, otherwise the eye can be considered closed. Similarly, an open or closed mouth can be detected by measuring the distance between the corresponding mouth landmarks. In both cases, PERCLOS can be used to determine the percentage of time the eyes or the mouth of the driver are closed or open to detect if his eyes are sleepy or if he is yawning. The PERCLOS that defines the percentage of closed eyes time is defined as:

$$PERCLOS = \frac{closed\ eyes\ time}{closed\ and\ open\ eyes\ time} \times 100 \quad (7)$$

3 ORIGINAL AND MODIFIED Face alignment application

As noted, the top level application performs face alignment through 68 landmarks (LM). Its operation is described in Algorithm 1. Initially, a stream is opened from a stored video or a camera. Face alignment can be performed every A_r frames i.e., when F_c % A_r=0, where the symbol '%' denotes the integer division modulo and F_c is the frame counter. A landmark alignment can be performed provided that a face has been detected in the specific frame using the OpenCV library face detector. If a new face alignment has to be performed in the current frame, a shape transform (ST) has to be performed to warp the current image in order to match the reference shape. ST requires scaling and rotation operations as described in equation (5). More details for this operation can be found in [4]. Then, landmark prediction is performed by the Tracker::predict() routine in the original DEST application (step 3.b.2 in Algorithm 1). Profiling showed that the Tracker::predict() routine introduces the highest latency: 94.25% of the time needed to process a single frame is spent in Tracker::predict() or the nested functions called within this routine. In the course of this paper, the functionality of Tracker::predict(), shown in Fig. 3, was implemented as a C kernel called predict_kernel(). The current form of the predict_kernel() function and the data types and structures used in this routine allow its implementation in reconfigurable hardware if further reduction in the execution is desired.

ALGORITHM 1: Video Tracking Application

```
1. open video stream
2. frame counter Fc←0
3. while not the end of the video stream, do
 3.a. read next frame from video stream
        3.b. if Fc % Ar =0 and a face is detected in the frame then
                3.b.1. warp frame with ST to match the mean shape
                3.b.2. predict shape S as a set of landmarks
            else
                3.b.3. if face is detected in the frame then
                        3.b.3.1. warp with ST to match the mean shape
                        3.b.3.2. apply the available shape S to the current frame
                end
        end
        3.c. Fc ← Fc +1
end
4. close video stream
end
```

As shown in Fig. 3, the mean shape is initially loaded from the trained model in the original DEST Tracker::predict() implementation. Then, the T_{cs} cascaded regressors are iteratively called: the routine Regressor::predict() is called T_{cs} times within Tracker::predict(). In the trained model used by the original and the

modified DEST application T_{cs} is equal to 10, as a trade-off between speed and accuracy. The first operation performed within each regressor i.e., within the Regressor::predict() routine is the image warping (ST). The gray scale intensities of selected P_r pixels from the warped image are read next. Then, the initialization of the mean residual s_r and f_0 (equation (3)) takes place and the routine Tree::predict() is called K times, to fit the K trees of the regressor. In the k-th, Tree::predict(), a single binary tree is fitted to get the weak regressor g_k. The strong regressor f_k is updated using equation (4). The mean residual s_r is updated with the last f_K value at the end of the Regressor::predict() routine. The shape estimation S is updated with the mean residual s_r in the top level function Tracker::predict(). Concerning the tree fitting in the Tree::predict() routine, each binary tree has 2^{T_d}-1=31 nodes and thus, a depth (T_d) equal to 5. Different g_k values are stored in the leaves of each regression tree. The root of each tree is accessed first and then a path is followed towards a leaf. This path is decided based on the intensities of predetermined pixels in the warped image (mapped from the P_r set). The right or left direction in the binary tree is opted depending on whether the intensity difference is larger than a threshold T_h.

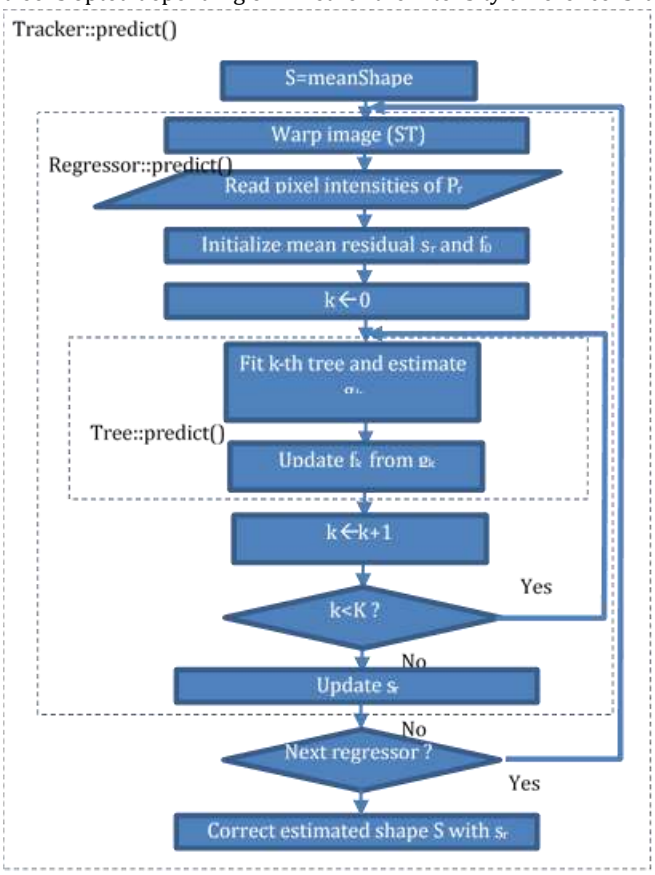

Figure 3: The functionality of the predict() routines. The functionality of Tracker::predict(), Regressor::predict() and Tree::predict() routines is implemented in the single predict_kernel() routine that is hardware synthesizable.

The functionality of the predict() routines shown in Fig. 3 has been flattened in the proposed, new predict_kernel() routine. The Eigen-library calls have been replaced by custom ANSI C implementations of matrix operations. The predict_kernel() routine is portable to reconfigurable hardware since simple C data types

consistent with hardware synthesis tools have been employed. The speed of the resulting predict_kernel() routine is two order of magnitude higher than the original Tracker::predict() routine due to the elimination of unnecessary data type checks and type conversions performed by the Eigen library.
Another issue that slowed down the original DEST implementation was the repeated access of the same parameters of the pre-trained model from scattered memory locations each time a Tracker::predict(), Regressor::predict() and Tree::predict() routines were called. All of these numerous parameters of the trained model are now loaded once during initialization into contiguous buffers from the new predict_prepare() routine that is invoked between steps 2 and 3 of Algorithm 1. The model parameters that are loaded from this initialization routine are listed in Table 1.

Table 1: Model parameters, size and number of accesses. Loaded once in the predict_prepare() routine

Parameters	Buffer size	Description	Number of Accesses
Tree sizes	$T_{cs}\times K\times$sizeof(int)	Each regressor could have a different number of trees. In this implementation each regress has T_{cs} trees	$T_{cs}\times K$
Learning Rates	$T_{cs}\times K\times$sizeof(float)	An identical learning rate is used in this implementation although different learning rates could have been used for each tree	$T_{cs}\times K$
Mean Shape	$2\times LM\times$sizeof(float)	Consists of LM coordinates	$2\times LM$
Relative Pixel Coordinates	$2\times Nc\times$sizeof(int)	The coordinates of the pixels in the P_r set	$2\times N_c$
Closest Shape Landmarks	$N_c\times$sizeof(int)	Each pixels in the P_r set is associated with the closest landmark	N_c
Tree node split index1	$T_{cs}\times K\times 2T_d\times$sizeof(int)	Refer to the index of the next tree node if the left direction is followed	$T_{cs}\times K\times T_d$
Tree node split index2	$T_{cs}\times K\times 2T_d\times$sizeof(int)	Refer to the index of the next tree node if the right direction is followed	$T_{cs}\times K\times T_d$
Tree node threshold	$T_{cs}\times K\times 2T_d\times$sizeof(float)	Tree node threshold T_h	$T_{cs}\times K\times T_d$
Tree node means	$T_{cs}\times K\times 2T_d\times$sizeof(float)	Tree node means (g_k). We are interested in the values stored in the leaves	$T_{cs}\times K$

The scattered values of each one of the parameters listed in Table 1 are stored in contiguous buffers within predict_kernel() making faster their access. These buffers can also be transferred once in the common memory accessed both by the processors and the programmable logic of an FPGA. The size of these buffers are listed in the 2nd column of Table 1. The number of accesses needed to these parameters is listed in the 4th column of Table 1. There are some cases in which there is no need to access the whole buffers as in the case of tree node information. For example, although each tree has 2^{Td} nodes, only T_d nodes are accessed in the tree fitting process of Tree::predict(). Similarly to Table 1, the parameters that need to be initialized from the trained model, at the beginning of each Tracker::predict() i.e., when a new frame is processed, are listed in Table 2. The initial shape estimate is read once and updated at the end. The mean residuals are initialized by the trained model and updated in each regressor. Therefore, the $2\times N_c$ mean residuals need $2\times N_c\times T_{cs}$ accesses.
In the modified application developed in the framework of this paper, the buffers allocated from the predict_prepare() routine (listed in Table 1) are released at the end of the application after step 4 of Algorithm 1. The buffers allocated at the beginning of Tracker::predict() i.e., the ones listed in Table 2, are released at the end of this routine.

Table 2: Model parameters, size and number of accesses. Accessed every time a new frame is processed.

Parameters	Buffer size	Description	Number of Accesses
Shape estimate	$2\times LM\times sizeof(float)$	LM landmark coordinates. Initialized from the mean shape and updated at the end with the mean residuals	$2\times LM$
Mean residuals	$2\times N_c\times sizeof(float)$	Initialized from the trained model and then, updated in each regressor	$2\times N_c\times T_{cs}$

4 EXPERIMENTAL RESULTS

The experimental results are listed in Table 3. The most important achievement of this work is the reduction of the Tracker::predict() latency from 117ms to less than 0.5ms i.e., an acceleration of more than 240 times is achieved. If the frame rate of the input video is 30 frames per second (fps) and a new face alignment is performed every 5 input frames, the frame rate of the application is improved from 1.6fps to 28ftp. This rate is quite close to the one of the input video and can be further improved in the predict_kernel() is implemented in hardware. The measurements listed in Table 3, were performed on an Intel 6-Core i5-9500 CPU @3.00GHz, with 16GB RAM, running Ubuntu 18.04. A 3 sec test video has been used as input both to the original DEST and the modified application. There was no accuracy degradation i.e., the position of the landmarks in this reference video are exactly the same in the original DEST and the modified application. However, if the floating point values in the predict_kernel() are represented as fixed point values or integers (through scaling) an accuracy degradation is expected. This could be useful in hardware implementations for lower resource allocation. Finding data types that require less hardware resources without significant accuracy loss is part of our on-going work.

Table 3: Experimental results

	Original DEST application	Our modified application
Latency of Tracker::Predict()	117ms	0.48ms
Frame rate (A_r=5)	1.6fps	28fps
GDB steps needed in 10 indicative cases	2 to 165	1 to 5

An indicative metric of the overhead posed by Eigen calls is the number of steps needed in a GNU Debugger (GDB): a) to call a function (from the time the function is called until the control reaches its first command), or b) to execute an initialization command, or c) to access a class member method. As shown at the last row of Table 3, ten indicative cases were examined. The fastest call concerned the access of a class member function to get the tree sizes (number of tree nodes): data.trees.size(). This operation needed 2 GDB steps in both the original DEST and the modified application. The slowest case concerned the initialization of a 2×2 matrix with zeros: Eigen::Matrix2f::Zero(2, 2). In the original DEST application this initialization took 165 GDB steps to complete while in our modified application it took only 5 GDB steps. Another interesting example was the copy of a singular value to a matrix. This command executed in a single GDB step in the modified application while 75 GDB steps were required by the corresponding command in the original DEST application: d(0, 0) = svd.singularValues()(0).

These indicative examples explain why the original DEST shape tracking application had a latency more than 240 times higher than the modified one. Exploiting advanced hardware acceleration techniques is also feasible in the current form of the application for further speed improvement. One of the limitations of the proposed approach is the reduced code readability since several lines of code are needed to describe operations that were expressed in a compact way in Eigen. The portability is also reduced and extensions to cover similar problems with different model dimensions are also harder to implement in ANSI C, since type conversions and integrity checks supported by the Eigen library were removed in computationally intensive operations.

5 Conclusions

The acceleration of a popular face shape tracking application offered in the DEST package was examined in this paper. This application exploits an ensemble of regression trees and uses Eigen template C++ library to implement matrix operations in a compact way. However, Eigen library poses excessive overhead with a large number of integrity checks and type conversions that guarantee that the algebraic operations will be correctly performed in any application but most of them were not necessary in our shape tracking application. The source code of the initial application was restructured and the most computational intensive of each part was described in ANSI C eliminating the calls to Eigen library. Of course, Eigen library calls remained in non-time critical parts of the code, due to the convenient way it offers in implementing algebraic operations. The modified computational intensive routines are now approximately 250 times faster and their structure is amenable for a hardware implementation if further acceleration is required.

Future work will focus on the implementation of the computational intensive operations in FPGAs. The penalty in the accuracy will also be studied when the floating point operations are implemented as fixed point or integer operations with appropriate scaling. Hardware acceleration techniques such as pipelined loop optimization, burst argument copy, multithread execution, etc. will also be employed in FPGA implementations.

ACKNOWLEDGMENTS

This work has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement No 871738 - CPSoSaware: Cross-layer cognitive optimization tools & methods for the lifecycle support of dependable CPSoS.

REFERENCES

<bib id="bib1"><number>[1]</number> Jesus Nuevo, Luis M. Bergasa, and Pedro Jimenez. 2010. RSMAT: Robust simultaneous modeling and tracking. Pattern Recognition Letters, ACM 31,16 (Dec. 2010), 2455-2463. https://doi.org/10.1016/j.patrec.2010.07.016.</bib>
<bib id="bib2"><number>[2]</number> Tereza Soukupová and Jan Cech. Eye-Blink Detection Using Facial Landmarks.2016. In Proceedings of the 21st Computer Vision Winter WorkshopRimske Toplice, Slovenia. </bib>
<bib id="bib3"><number>[3]</number> Dini Adni Navastara, Widhera Yoza Mahana Putra, and Chastine Fatichah. 2020. Drowsiness Detection Based on Facial Landmark and Uniform Local Binary Pattern. J. Phys., 1529:052015. http://dx.doi.org/10.1088/1742-6596/1529/5/052015.</bib>
<bib id="bib4"><number>[4]</number> Vahid Kazemi and Josephine Sullivan. 2014. One millisecond face alignment with an ensemble of regression trees. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 1867–1874. 10.1109/CVPR.2014.241.</bib>
<bib id="bib5"><number>[5]</number> Davidjoseph J. Tan, Federico Tombari, and Nassir Navab. 2015. A Combined Generalized and Subject-Specific 3D Head Pose Estimation. In Proceedings of International Conference on 3D Vision (3DV). http://dx.doi.org/10.1109/3DV.2015.62. </bib>
<bib id="bib6"><number>[6]</number> Yanchao Dong, Minjing Lin, Jiguang Yue, and Liang Shi.2019. A low-cost photorealistic CG dataset rendering pipeline for facial landmark localization. Multimedia Tools and Applications, Springer 78(6). https://link.springer.com/article/10.1007/s11042-019-7516-5.</bib>
<bib id="bib7"><number>[7]</number> Jon Goenetxea, Luis Unzueta, Fadi Dornaika, and Oihana Otaegui. 2020. Efficient deformable 3D face model tracking with limited hardware resources. Multimedia Tools and Applications, Springer 79(4). https://link.springer.com/article/10.1007/s11042-019-08515-y. </bib>
<bib id="bib8"><number>[8]</number> Francisco Madrigal and Frederic Lerasle. 2020. Robust head pose estimation based on key frames for human-machine interaction. EURASIP Journal on Image and Video Processing, Springer 2020(1). https://jivp-eurasipjournals.springeropen.com/articles/10.1186/s13640-020-0492-x. </bib>
<bib id="bib9"><number>[9]</number> Dlib C++ library. Retrieved May, 19, 2021 from http://dlib.net/</bib>
<bib id="bib10"><number>[10]</number> Deformable Shape Tracking (DEST). Retrieved May, 19, 2021 from https://github.com/cheind/dest </bib>
<bib id="bib11"><number>[11]</number> Grigorios Chrysos, Epameinondas Antonakos, Patrick Snape, Akshay Asthana, and Stefanos Zafeiriou. 2018. A Comprehensive Performance Evaluation of Deformable Face Tracking "In-the-Wild". Int. J. Comput. Vis.126(2),198-232.doi:10.1007/s11263-017-0999-5.</bib>
<bib id="bib12"><number>[12]</number> Constantino Álvarez Casado and Miguel Bordallo López. 2021. Real-time face alignment: evaluation methods, training strategies and implementation optimization. Springer J. Real-Time Image Processing. https://doi.org/10.1007/s11554-021-01107-w.</bib>
<bib id="bib13"><number>[13]</number> Eigen 3.3.9. Retrieved May, 19, 2021 from https://eigen.tuxfamily.org/</bib>
<bib id="bib14"><number>[14]</number> Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman. 2001. The elements of statistical learning: data mining, inference, and prediction. New York: Springer-Verlag.</bib>