High Speed Implementation of the Deformable Shape Tracking Face Alignment Algorithm

Nikos Petrellis
Dept. of Electrical and Computer Engineering
University of Peloponnese
Patra, Greece
npetrellis@uop.gr

Stavros Zogas
Dept. of Electrical and Computer Engineering
University of Peloponnese
Patra, Greece
s.zogas @esda-lab.gr

Panagiotis Christakos
Dept. of Electrical and Computer Engineer
University of Peloponnese
Patra, Greece
p.christakos @esda-lab.gr

Georgios Keramidas
School of Informatics
Aristotle University of Thessaloniki
Thessaloniki, Greece
gkeramidas@csd.auth.gr

Panagiotis Mousouliotis
School of Informatics
Aristotle University of Thessaloniki
Thessaloniki, Greece
p.mousouliotis@esda-lab.gr

Nikolaos Voros
Dept. of Electrical and Computer Engineering
University of Peloponnese
Patra, Greece
voros@esda-lab.gr

Christos Antonopoulos
Dept. of Electrical and Computer Engineering
University of Peloponnese
Patra, Greece
ch.antonop@esda-lab.gr

***Abstract*— The 2D facial landmark alignment method, implemented in C++ in the open source libraries DLIB and Deformable Shape Tracking (DEST), is used in several applications such as driver drowsiness detection. The most challenging of these applications require fast video frame processing. Therefore, the alignment of the facial landmarks in a single video frame has to be performed with the minimum possible latency without precision loss. In this paper, the DEST implementation of the face alignment method that is based on regression trees is heavily restructured to reduce latency. The resulting face alignment predictor is implemented in C. The elimination of multiple nested routine calls, excessive argument copying, type conversions and integrity checks lead to a software implementation that is 240 times faster than the one provided in the DEST library. Moreover, the structure of the new face alignment predictor is appropriate for hardware implementation on a Field Programmable Gate Array (FPGA) for further acceleration[1].**

I. INTRODUCTION

Regression-based methods are the dominant solutions for face alignment tasks in computer vision. These methods employ a series of mapping functions to iteratively update the face shape hypothesis. A fast 2D facial landmark detection algorithm has been presented by Kazemi and Sullivan in [1] where an Ensemble of Regression Trees (ERT) has been used to estimate the position of the facial landmarks. A subset of image pixel intensities is used to reduce latency. ERT is trained using gradient boosting in order to optimize the sum of square error loss.

The authors in [2] formulate the regression procedure as a sparse coding problem. An occlusion dictionary is used into the face appearance dictionary to recover face shape from partially occluded face appearance. Cascaded regression has been interpreted as a learning-based approach to iterative optimization methods like the Newton's method. In [3], the problem of facial deformable model fitting is addressed using cascaded regression. A method is proposed in [4], that locates the facial landmarks and extracts discriminating features from suitable facial regions. Histogram of Oriented Gradients (HOG) features are extracted from the active facial patches instead of the whole face, which makes the system robust against the scale and pose variations.

The authors of [5] also relied on ERT to evaluate computer graphics rendered datasets. More specifically, they used 500 trees with depth 5 each one. This size is used in our implementation too. Masui et al., employ an ERT-based method [6] to align 68 facial landmarks, in order to avoid facial expression errors, and to measure the reaction of people on advertisements. Finally, the authors of [7] developed an ERT method to estimate head pose for human-machine interaction achieving a latency equal to 1ms per image frame.

The face alignment algorithm initially presented in [1] is used in the C++ DLIB machine learning toolkit and the Deformable Shape Tracking (DEST) library [8]. The DEST library implementation of the facial landmark alignment algorithm presented in [9], is modified in the approach described in this paper. The source code of the DEST implementation has been developed in C++ based on the Eigen template library for linear algebra operations. Eigen library optimizes matrix/vector operations as well as Singular Value Decomposition (SVD), Jacobi matrices, etc. However, the compact description of the algebraic operations comes at a high latency cost caused by an excessive number of nested routine calls. The CPU implementation of the DEST library shows a latency that is more than 100 times slower than 1ms that is advertised in [1]. Moreover, the original code uses Eigen/DEST C++ classes, that are not appropriate for the implementation of the computational intensive operations in reconfigurable hardware (FPGAs).

In this paper, the libraries DEST and Eigen have been ported to Linux Ubuntu and compiled with tools similar to the ones used in state-of-the-art hardware design tools such as Xilinx Vitis (GNU C++ and Vitis compilers). Thus, the porting of the Ubuntu DEST/Eigen package to the Xilinx Vitis environment is straightforward. We have profiled a DEST video processing application that tracks the facial landmarks, in order to extract the computational intensive operations. As a second step, we modified their source code so that they can also be accelerated in hardware. The face landmark prediction routine and the nested calls have been flattened, converted

[1] The source code of the modified DEST face alignment implementation will be available in Github: https://github.com/ncpetrellis/

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. project.

from C++ to C, and their arguments are now pointers to integers or floating point buffers (according to the memory arena programming paradigm). The resulting code is not only appropriate for hardware acceleration using reconfigurable hardware, but it is also 240 times faster since the latency of the face landmark prediction routine is reduced from 116 ms to 479 μs on an Intel Core i5-9500 CPU @3.00GHz, 6 core processor with 16GB RAM.

This paper is structured as follows. A brief review of the face alignment prediction method suggested in [1] is presented in Section II. The implementation of the inference step and the modified structure of the source code that implements face alignment is described in Section III. Experimental results and the reasons of the high latency showed in the original DEST implementation are discussed in Section IV. Finally, the conclusions and future work are presented in Section V.

II. Face Alignment Based on Regression Trees

In [1], focus is given mainly on the training of the regression trees while we emphasize in the efficient implementation of the inference step. Let p be the number of face landmarks in an image *Img*, and x_i be the pair of the i-th landmark coordinates. The shape $S \in R^{2p}$, formed by these p-landmarks is defined as:

$$S = \{x_0, x_1, .., x_p\} \quad (1)$$

The algorithm starts by assuming that the shape S is the mean landmark positions retrieved from the trained model. Then, the position of the actual landmarks in the image under test, is refined by a sequence of cascaded regressors. The current shape estimate in the regressor t is denoted as $\hat{S}^{(t)}$ (t=1,.., T_{cs}). The transition to the next regressor, involves the prediction of a correction factor r_t that depends on the image *Img* and the current estimate $\hat{S}^{(t)}$. This correction factor r_t is added to $\hat{S}^{(t)}$ in order to generate the updated shape in the next regressor t+1: $\hat{S}^{(t+1)}$. The r_t value is estimated based on the intensities of pixels that are indexed relative to the current shape $\hat{S}^{(t)}$. Each regressor r_t is trained using a gradient tree boosting algorithm with a sum of square error loss. This training algorithm exploits the triplet $(I_{\pi_i}, \hat{S}_i^{(t)}, \Delta S_i^{(t)})$ where the training data consists of a set of N images $I_{\pi_i}, 0 \le \pi_i < N$ and $\hat{S}_i^{(t)}$ is the shape of any training image $i \neq \pi_i$. The residual $\Delta S_i^{(t+1)}$ in the regressor r_{t+1} is estimated as [1]:

$$\Delta S_i^{(t+1)} = S_{\pi_i} - \hat{S}_i^{(t+1)} \quad (2)$$

The residuals correspond to the gradient of the loss function and are estimated for all training samples. The shape estimation for the next regressor stage is performed as [1]:

$$\hat{S}^{(t+1)} = \hat{S}^{(t)} + r_t(I_{\pi_i}, \hat{S}_i^{(t)}) \quad (3)$$

Then, for K tests (k=1, …, K) and for N training images (i=1,…, N) in each test the following calculation is repeated:

$$r_{ik} = \Delta S_i^{(t)} - f_{k-1}(I_{\pi_i}, \hat{S}_i^{(t)}) \quad (4)$$

In the k-th step, a regression tree is fitted to r_{ik} using a weak regression function g_k, thus f_k is updated as follows [1]:

$$f_k(I, \hat{S}^{(t)}) = f_{k-1}(I, \hat{S}^{(t)}) + lr \cdot g_k(I, \hat{S}^{(t)}) \quad (5)$$

where $lr<1$ is the learning rate used to avoid overfitting. The final correction factor r_t of the t-regressor is equal to f_K.

The next node that should be visited in a regression tree is determined by comparing the intensity difference of two pixels p_1 and p_2 with a threshold T_h. Different threshold T_h values are defined for each regression tree node in the trained model. The coordinates of p_1, p_2 should be indexed relative to the reference mean shape and for this purpose, the image could be warped to match the mean shape. A global process called Similarity Transform (ST) is used to perform the required warping. If q is an image pixel and its closest landmark has index k_q, their distance δx_q is estimated as $\delta x_q = \left\| q - x_{k_q} \right\|$. The pixel q' in the original image *Img* that corresponds to q in the mean shape is estimated by [1]:

$$q' = x_{i,k_q} + \frac{1}{s_i} R_i^T \delta x_q \quad (6)$$

The parameters s_i and R_i are scale and rotation matrices used in the ST to warp the initial shape. The minimization of the mean square error between the actual q' value and the one estimated using eq. (6), can be used to estimate the optimal values for s_i and R_i. More details can be found in [1].

III. Original DEST and Modified Implementation

The implementation of the method described in [1] with the open source DEST package [8] was modified to allow its acceleration with reconfigurable hardware. The DEST implementation takes advantage of the Eigen template library that simplifies complicated matrix operations. All the supported DEST/Eigen operations are well-defined and protected from overflows, rounding errors, inappropriate type conversions, etc. Eigen operators are overloaded using multiple templates that support various numeric types (e.g., complex numbers). The cost of these facilities is a vast performance degradation. Moreover, the employed Eigen classes cannot be used in hardware synthesis.

In order to overcome these limitations, the source code was reconstructed by simplifying operand types, flattening nested routine calls, copying large data once (e.g., during initialization), etc. The computational intensive operations are described in ANSI C, to reassure that they can be ported to hardware by state of the art tools such as Xilinx Vitis.

A DEST application that aligns landmarks in faces detected in the frames of a video or camera stream is used. Frames from the video stream are retrieved and if face landmark alignment is needed in the specific frame, face detection takes place using the OpenCV library. In the frames where a new face detection is not required, the face is assumed to exist in an extended bounding box around the shape estimated in the previous frame. Landmark alignment is performed in the bounding box returned by OpenCV using the *predict()* function. The ST process has to be applied on the detected face bounding box to adapt its coordinates to the ones used by the mean shape stored in the trained model. This model consists of a number of regression trees in each cascade stage and the tree node values are available from the training.

The most computational intensive operation, i.e., the *predict()* function, is implemented in three nested levels: *Tracker::predict()* calls *Regressor::predict()* which in turn calls *Tree::predict()*. *Tracker::predict()* (top level) accepts as input the image frame and the position of the bounding box of the recognized face and returns the estimated face landmark positions. Within *Tracker::predict()*, the *Regressor::predict()* is called T_{cs}=10 times. The *Regressor::predict()* accepts as input the *Img*, the face bounding box coordinates and the current estimate (corresponds to *f* in eq. (5)) of the landmark shape. The *Regressor::predict()* returns the shape residuals s_r, that are used to update the estimate *f* in the *Tracker::predict()*. The *Regressor::predict()* routine initially performs ST to adapt the current landmark shape to the mean coordinates of the trained model. Then, pixel intensities are read from the sparse *Img* representation and the system locates the closest landmarks in these pixels by trying to fit the regressor trees stored in the trained model. The routine *Tree::predict()* is called for all the stored N_{tr} binary regressor trees (N_{tr}=500). Each *Tree::predict()* call returns a correction factor called mean residuals (m_r). The leaves of each regression tree have different m_r values stored. Each tree is accessed from the root to the leaves within the *Tree::predict()* routine, following the appropriate path. Each tree has depth equal to T_d=5 and $2^{T_d} - 1 = 31$ nodes. In each intermediate tree node, the intensities of a pair of pixels indexed in the trained model are compared. The right or left direction of the binary tree is opted depending on whether the intensity difference is larger than a threshold T_h that is also stored in the trained model.

In the proposed Modified Implementation (MI), the functionality of the three nested *predict()* routines is included in the new *Kernel_predict()* routine that has been developed in ANSI C, in order to be portable to hardware. All the numerous parameters of the trained model that were accessed in the original DEST implementation from *predict()* routines, are now loaded during initialization into contiguous buffers from the new *predict_prepare()* routine called once during initialization. More specifically, the number of the N_{tr} regression trees and their node values are read: the threshold T_h, the mean residual m_r, and the indices of the pixels that their intensity difference has to be compared with T_h. Moreover, the T_{cs} learning rates *lr*, the *LM* coordinates of the mean shape landmarks, the N_c relative pixel coordinates of the sparse image and the closest landmark to these relative pixel coordinates are also read from the trained model.

In the new *Tracker::predict()*, some model values (default mean shape estimate and mean residuals) are initially read into buffers from the trained model for each new frame. Pointers to these buffers along with pointers to the buffers prepared by *predict_prepare()* and frame-specific information (pointer to image buffer, image dimensions, position of the detected face) are passed to the *Kernel_predict()* routine. The final landmark shape is returned by *Kernel_predict()* and its homogeneous coordinates are converted to the absolute coordinates needed in order to display the landmarks on the tested image frame.

IV. Experimental Results-Discussion

To validate whether the MI implementation exhibits any accuracy degradation compared to the original DEST implementation, videos from the dataset of [11] were used. The videos from this dataset last 5-8 sec. and their resolution is 1920×1080. Face alignment was performed once every five frames and the extracted landmarks were 100% identical between the MI and the original DEST version. The performance comparison between the original DEST and the MI implementation showed that the latency of the *Tracker::predict()* routine was reduced from 116 ms to 475 μs on an Ubuntu 18.04, Intel 6-Core i5-9500 CPU @3.00GHz, with 16GB RAM. The frame rate was increased from 1.6 fps to 28 fps. These performance results are also confirmed when a webcam input stream is used.

In order to better understand why the MI implementation runs so fast on the same platform, we will examine two architectural differences between the original DEST and the MI source code. The first one concerns the number of operations needed to define the effective memory address of the model parameters. These parameters are copied once in contiguous buffers in the *predict_prepare()* routine in the MI code. The number of operations needed to estimate the effective address of an individual parameter encounters the additions of the offsets to the base address of the buffer. In most of the cases, the offset additions were reduced by 1 in the MI implementations. The DEST implementation required 1.87 times more operations than the MI source code for the estimation of the effective addresses.

The reduction in the operations needed to estimate effective memory addresses cannot justify the improvement of the *Tracker::predict()* latency by 240 times. Eigen library offers a convenient way to describe algebraic operations. For example, matrix multiplication, complex number handling, SVD and other complicated operations can be described in a compact way with overloaded operators. The cost of these facilities is the increased latency posed by the Eigen library. The MI implementation runs much faster because the functionality of the *Tracker::predict()*, and the nested *Regressor::predict()* and *Tree::predict()* routines are implemented as a C *Kernel_predict()* routine. Eigen types are replaced in this routine, with simple C types, avoiding type conversions. No excessive Eigen constraint checks (e.g., for value overflow) are applied. Moreover, no operator overloading is used and each operation is implemented by custom inline code in the *Kernel_predict()*.

An indicative way to measure how much overhead was posed by the Eigen library, is to use the GNU debugger (GDB) and measure how many "Step Into" commands are needed for a specific operation either in the original DEST or the MI source code. For example, reading the number of trees in each regressor is completed in 2 GDB steps. Accessing a tree node requires 10 GDB steps. Reading two pixel intensities and comparing their difference with the tree node threshold requires 62 GBD steps. Accessing the first singular value from the *svd* structure requires 75 GDB steps. The number of GDB steps needed between the call of the routine *estimateSimilarityTransform()* that implements ST and the execution of its first instruction, is 128. Initializing a 2×2 matrix with zeros, requires 165 GDB steps. Finally, when the *readPixelIntensities()* routine is called to access the pixel intensities of the input image, 162 GDB steps are needed only for the type conversion and integrity checks of the first two arguments. Then, a nested loop with 1080×1920 iterations takes place and in each iteration the routine *kernel.assignCoeffByOuterInner()* is called. In all of the above cases, the corresponding commands in the MI implementation require only a single GDB step. We will focus on the *Eigen::Matrix2f d = Eigen:: Matrix2f::Zero(2, 2)* command to justify the 165 GDB steps required. Table I lists the various functions and the steps GDB spends in each one of them in

order to complete the *d* matrix initialization. Explaining the functionality of each one of these routines and their input/output arguments is out of the scope of this paper although their operation is implied by their names in most of the cases. Fig. 1 shows examples of face alignment achieved by the MI implementation using the dataset of [11].

V. Conclusions

The popular DEST package, Eigen library-based implementation of face alignment algorithm that exploits an ensemble of regression trees was modified to support hardware implementation. The computational intensive routines were flattened and implemented in C language. The replacement of Eigen library classes and types led to an implementation that is 240 times faster than the original DEST implementation.

In our future work we will examine the accuracy penalty of an integer representation of the model parameters that are realized as floating point values in the original source code. An integer representation would lead to a hardware implementation with fewer resources. The face alignment algorithm will also be implemented in FPGA devices for applications that require ultra-high speed face alignment.

References

[1] V. Kazemi and J. Sullivan, "One millisecond face alignment with an ensemble of regression trees," 2014 IEEE Conference on Computer Vision and Pattern Recognition, Columbus, OH, USA, 2014, pp. 1867-1874, doi: 10.1109/CVPR.2014.241

[2] D. J. Tan, F. Tombari, and N. Navab, "A Combined Generalized and Subject-Specific 3D Head Pose Estimation," 2015 International Conference on 3D Vision, Lyon, France, 2015, pp. 500-508, doi: 10.1109/3DV.2015.62.

[3] J. Xing, Z. Niu, J. Huang, W. Hu, X. Zhou, and S. Yan, "Towards Robust and Accurate Multi-View and Partially-Occluded Face Alignment," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 40, no. 4, pp. 987-1001, 1 April 2018, doi: 10.1109/TPAMI.2017.2697958.

[4] Z. Gan, L. Ma, C. Wang and Y. Liang, "Improved CNN-based facial landmarks tracking via ridge regression at 150 Fps on mobile devices," 2017 10th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI), Shanghai, China, 2017, pp. 1-9, doi: 10.1109/CISP-BMEI.2017.8301921.

[5] Y. Dong, M. Lin, J. Yue, and L. Shi, "A low-cost photorealistic CG dataset rendering pipeline for facial landmark localization," Multimedia Tools and Applications, 2019.

[6] K. Masui, G. Okada, and N. Tsumura, "Measurement of advertisement effect based on multimodal emotional responses considering personality," ITE Transactions on Media Technology and Applications, vol. 8, no. 1, pp. 49-59, 2020.

[7] F. Madrigal and F. Lerasle, "Robust head pose estimation based on key frames for human-machine interaction", EURASIP Journal on Image and Video Processing, vol. 2020, doi: https://doi.org/10.1186/s13640-020-0492-x.

[8] Deformable Shape Tracking (DEST). Available online: https://github.com/cheind/dest (accessed on 17 April 2021).

[9] G. G. Chrysos, E. Antonakos, S. Zafeiriou, and P. Snape, "Offline Deformable Face Tracking in Arbitrary Videos," 2015 IEEE International Conference on Computer Vision Workshop (ICCVW), Santiago, Chile, 2015, pp. 954-962, doi: 10.1109/ICCVW.2015.126.

[10] T. Hastie, R. Tibshirani, and J. H. Friedman. "The elements of statistical learning: data mining, inference, and prediction," New York: Springer-Verlag, 2001.

[11] L. Jeni, H. Yang, R. Pillai, Z. Zhang, J. Cohn, and L. Yin, "3D Dense Face Reconstruction from Video (3DFAW-Video) Challenge", 2nd 3DFAW-Video Workshop/Challenge, IEEE International Conference on Computer Vision (ICCV), 2019.

TABLE I. Eigen Functions Called When Executing the Command: Eigen::Matrix2f d = Eigen::Matrix2f::Zero(2, 2);

Function	***GDB steps***	***Function***	***GDB steps***	***Function***	***GDB steps***
zero()	5	DenseStorage()	2	generic_dense_assignment_kernel()	2
scalar_constant_op() //2 versions	3	resize() // 2 versions	4	assignPacket() // 2 versions	5
nullaryExpr()	6	data() // 2 versions	2	assignPacketByOuterInner()	3
CwiseNullaryOp()	4	_check_template_params()	1	rowIndexByOuterInner() // 2 verions	6
rows() // 3 versions	11	resizeLike()	4	colIndexByOuterInner()	2
cols() // 3 versions	8	_set_noalias()	4	resize_if_allowed()	5
functor()	1	EIGEN_EMPTY_STRUCT_CTOR (assign_op)	1	call_dense_assignment_loop()	9
EIGEN_DEFAULT_EMPTY_CONSTRUCTOR _AND_DESTRUCTOR(MatrixBase)	2	derived() // 3 versions	7	actualDst()	1
DenseBase()	1	const_cast_derived()	1	evaluator() // 3 versions	6
variable_if_dynamic()	2	check_for_aliasing()	2	packet()	2
value()	9	pset1<Packet4f>()	2	noncopyable() // 2 versions	8
constant()	1	pstore<float>()	2	coeffRef()	2
innerSize()	2	pstoret()	1	matrix()	3
outerStride() // 2 versions	3	PlainObjectBase()	6	run() // 4 versions	14

Fig. 1. Example face alignment using video frames from [11]. Both original DEST and MI implementations estimate identical landmark positions.