Towards Real-Time Whisker Tracking in Rodents for Studying Sensorimotor Disorders

Yang Ma[‡*], Prajith Ramakrishnan Geethakumari[†], Georgios Smaragdos[‡], Sander Lindeman[‡], Vincenzo Romano[‡], Mario Negrello[‡], Ioannis Sourdis[†], Laurens W.J. Bosman[‡], Chris I. De Zeeuw[‡], Zaid Al-Ars[*] and Christos Strydis[‡]
[*]Computer Engineering Laboratory, Delft University of Technology, The Netherlands
[†]Computer Science Engineering Department, Chalmers University of Technology, Sweden
[‡]Department of Neuroscience, Erasmus MC, The Netherlands
Contact e-mail: c.strydis@erasmusmc.nl

***Abstract*—The rodent whisker system is a prominent experimental subject for the study of sensorimotor integration and active sensing. As a result of improved video-recording technology and progressively better neurophysiological methods, there is now the prospect of precisely analyzing the intact vibrissal sensorimotor system. The vibrissae and snout analyzer (ViSA), a widely used algorithm based on computer vision and image processing, has been proven successful for tracking and quantifying rodent sensorimotor behavior, but at a great cost in processing time. In order to accelerate this offline algorithm and eventually employ it for online whisker tracking (less than 1 ms/frame latency), we have explored various optimizations and acceleration platforms, including OpenMP multithreading, NVidia GPUs and Maxeler Dataflow Engines. Our experimental results indicate that the optimal solution for an offline implementation of ViSA is currently the OpenMP-based CPU execution. By using 16 CPU threads, we achieve more than 4,500x speedup compared to the original Matlab serial version, resulting in an average processing latency of 1.2 ms/frame, which is a solid step towards real-time (and online) tracking. Analysis shows that running the algorithm on a 32-thread-enabled machine can reduce this number to 0.72 ms/frame, thereby enabling real-time performance. This will allow direct interaction with the whisker system during behavioral experiments. In conclusion, our approach shows that a combination of software optimizations and the careful selection of hardware platform yields the best performance increase.**

I. Introduction

Throughout the animal kingdom, multiple systems perform active sensing, such as the fingertips of primates, the antennae of insects and the facial whiskers of rodents. The latter has become a model system for studying the workings of sensorimotor integration, thanks to the very well-defined anatomy of the brain structures involved [1], [2]. Sensor guidance and control are progressively recognized as the key challenges in adaptive systems, whether natural or artificial. Understanding the neural control over the whisker system could inspire new treatments for human diseases involving motor disability and/or defects in sensorimotor integration. Currently, the accurate tracking of the rapid whisker movements is a bottleneck for such research.

Mice sweep their whiskers, or vibrissae, back and forth during active exploration of their environment, enabling them to navigate in the dark. Research on vibrissal tactile sensing of rodents is moving to a critical stage. Emerging data raises the prospect of establishing accurate relationships between vibrissal movements in awake, exploring animals and accompanying brain activity [3]. Accomplishing this goal requires objective, quantitative characterization of the kinematics of the head and whiskers.

High-speed videography is non-invasive and offers great temporal and spatial resolution. Its major disadvantage is the complexity of image processing. High-speed films add up to massive amounts of data. For this reason, software tools have been developed for automated motion analysis, with a most efficient one being the BIOTACT Whisker-Tracking Tool (BWTT) [4]. The main algorithm behind BWTT is the *vibrissae and snout analyzer (ViSA)*[1], which has excellent precision in whisker tracking [5]. It is designed for tracking the head and for extracting whisking parameters, while not requiring whisker manipulation and not requiring to limit mouse movement to unacceptable levels.

Unfortunately, the ViSA processing rate lags far behind the data-generation rate of modern cameras. The acceleration of the whisker-tracking algorithm could speed up behavioral and neurophysiological research considerably. It could also become the cornerstone for supporting online whisker tracking, which shall not only eliminate the need for maintaining large storage to keep raw videos, but shall also allow novel experimental paradigms based upon real-time behavior.

This work focuses on accelerating ViSA through exploring various software and hardware techniques. The contributions of this paper can be summarized as follows:

- Rapid optimization of the original Matlab toolbox using data parallelism and GPU acceleration, resulting in 2x speedup while retaining the same level of user experience.
- Three-orders-of-magnitude acceleration of the ViSA algorithm by porting to C language, optimizing and combining the OpenCV library with various HPC technologies.
- Detailed profiling of the original BWTT toolbox and extensive evaluation of all implemented algorithm versions.

This paper is organized as follows: In Section II, related whisker-tracking techniques are presented. In Section III, ViSA is explained in detail. In Section IV, the implemented optimized and accelerated versions of the BWTT toolbox are

[1]In this paper, the terms BWTT and ViSA shall be used interchangeably.

presented. Evaluation results are given in Section V. Finally, in Section VI, conclusions are presented.

II. Related Work

Over the last decade, three general approaches to whisker-tracking techniques have been proposed: 1) tracking whisker intersection with a sheet of light [6]; 2) using electromyograms (EMGs) on the facial muscles [7]; and 3) high-speed videography. Every method has its merits. The light sheet grants high spatial and temporal resolution but reveals only the 1-D intersection points, requiring restraining the animal so that its whiskers are located in the plane of light. EMG recordings are invasive and measure the activity patterns of the facial muscles, but not the actual whisker movements. In contrast, high-speed (>500 Hz) videography is non-invasive and can record the whisker movements of behaving test subjects.

Nevertheless, high-speed videography has its own methodological challenges. Whiskers usually move in complex patterns and at extremely high speeds [8]. Neighboring whiskers could partake distinct trajectories. Furthermore, whiskers are thin hairs – they attenuate to a thickness of a few micrometers – resulting in limited imaging contrast. To address these challenges, researchers have developed different methods.

In [9] and [10] whiskers of unconstrained animals are traced but require trimming to prevent occlusions and intersections. In [11], markers need to be glued to single whiskers. However, whisker clipping or tagging unavoidably decreases the resemblance of the animal's natural sensorimotor strategy.

Furthermore, there are several approaches using the method of best intensity overlap which establishes the whisker position choosing a spline [10] or segments [12] that overlap the image pixels with a larger intensity sum. However, image noise can compromise the robustness of these approaches. Another approach implements iterative joining of nearby pixels of similar width and orientation in traces [13]. Yet, it fails to pick the best segments based on a global optimization taking into account every combination of the detected intensity nodes.

Finally, the average angle at the base of a set of whiskers can be measured [14]. This approach transfers the input image into a polar-coordinate system, where whiskers are presented as almost straight lines. The transformed image is binarized by applying a Hough-transform voting scheme to detect the orientation of these lines. This is the most sensitive aspect of the procedure since obtaining optimal parameters that are able to select all whiskers is difficult. The ViSA algorithm considered in this work employs and improves on this final approach and is capable of tracking the animal snout and extracting whisking parameters without requiring whisker manipulation or placing any constraints on animal movement. Overall, ViSA is a reliable algorithm worth optimizing and accelerating.

III. ViSA Algorithm

To trace the head and extract relative whisking parameters, ViSA is composed of two modules. The first module tracks snout position, orientation and contour over time. It enables constructing a head-centered coordinate system for describing whisker-relative positions. The second module extracts whisker segments within a user-defined distance from the snout of the test subject. Then, it provides a parameterization of whisker movement over time.

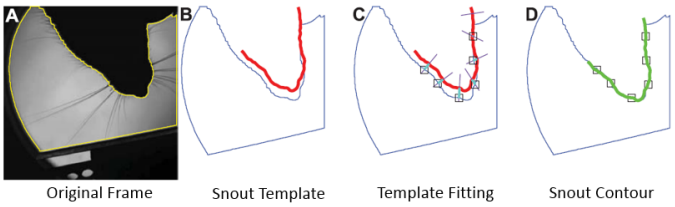

Fig. 1. Snout-template fitting on generic contours to detect snout location and orientation (Images modified from [5])

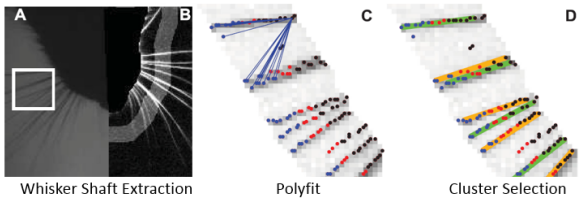

Fig. 2. Whisker angle is computed by approximating whiskers as linear segments in a narrow band around the snout (Images modified from [5])

A. Head detection

1) Generic-contour extraction: Contour detection is required to generically distinguish all contours within the frame. Video frames are converted to binary images after thresholding to separate whiskers and head silhouettes out of the light background. Then, whiskers and also noisy points are filtered out using morphological closing. The pixels of the binary-image boundaries are extracted (yellow outline in Figure 1A).

2) Snout-template fitting: Then, the algorithm traces the head based on the extracted generic outline. It uses a fixed head template (Figure 1B, red outline) to match to the extracted generic contours (Figure 1B, blue outlines). The idea of template fitting is to minimize perpendicular distances between detected contours and the fixed template (Figure 1C, cyan line segments). Therefore, partial occlusions or local asymmetries can be overcome for the reason that the symmetric-points count can easily outweigh them. After the snout template is matched, represented as the green contour in Figure 1D, the local generic-contour substitutes the template to create a fit for the snout edge pixel-by-pixel.

B. Whisker-shaft detection and parameter extraction

1) Background subtraction and masking: Background subtraction and mask multiplication highlight the whiskers in the original image frame (Figure 2A) as shown in Figure 2B. To identify the background, ViSA accumulates brightest values over the course of a video-recording session. Then, it multiplies a binary mask with the resulting image to select pixels within a radius from the snout contour. Finally, standard bicubic interpolation is used to increase image resolution, followed by Gaussian smoothing to decrease recording noise.

2) Whisker-tree creation: This phase first converts the video frame to a head-centered polar system. At a given distance from the head contour, local-maxima are detected on 1-D arrays of pixel intensity values and sorted by the angular coordinate. There are two parameters regulating the intensity-peak selection process: the minimum intensity threshold used to diminish incidences of noise, and the minimum angular distance used to prevent adjacent peaks being perceived inside the same shaft. Figure 2C gives an instance of intensity-peaks detection, with dots indicating their position. The extracted intensity peaks form a tree structure, known as *whisker tree*.

3) Clustering using Polyfit: Segment detection is formulated as a process of clustering, with the peculiarity that the cluster center is a line passing two peaks, termed as *source* and *sink* node. Source and sink nodes are classified based on their distance from the head contour. Extracted peaks positioned near the snout contour are marked as source nodes and those close to the external border of the mask are marked as sink nodes. Source nodes (black dots), sinks nodes (blue dots), and central nodes (red dots) – remaining nodes situated between sources and sinks – are shown in Figure 2C. ViSA utilizes the Matlab *Polyfit* function (i.e. polynomial-curve fitting) to create connections. It constructs a polynomial mathematical function, which has the best fit to a series of input data points.

4) Candidate-segments selection: In the final phase, cluster candidates are ranked based on the count of collinear nodes, which are nodes situated within a predefined distance. The collinear nodes are also marked as candidates for this cluster. Figure 2D shows an example, and colored segments are plotted with a width, which indicates the concept of "covering" regarding a distance range near a line segment. Each time a candidate is selected, whisker-tree nodes within the user-defined distance range are assigned to that cluster and stricken off from other clusters. The process is reiterated until no clusters score above the distance threshold, indicating that there do not exist enough collinear whisker-tree nodes remaining to form a whisker shaft.

IV. Application Profiling

BWTT has been profiled through use of the Matlab Profiler on a representative video chunk of a head-restrained mouse (50,000 frames). The BWTT processing time is approximately 80 hours on a modern CPU. Results are illustrated in Figure 3.

Inefficient functions within the algorithm could be identified and studied for alternatives. As indicated in Figure 3, Polyfit takes up 68.58% of the execution time, which is also consistent with the actual computational complexity of this portion of the algorithm as rationalized next. In the cluster-generating phase, ViSA extracts intensity peaks possibly situated on the whisker shafts stored as a whisker tree for each frame. Starting here, the amount of computation escalates exponentially. For every (proximal) source node, the algorithm initially pairs it with all (distal) sink nodes and creates candidate segments by applying Polyfit. Every candidate segment is sorted according the number of collinear whisker-tree nodes. All possible combinations between source and sink nodes generate a large

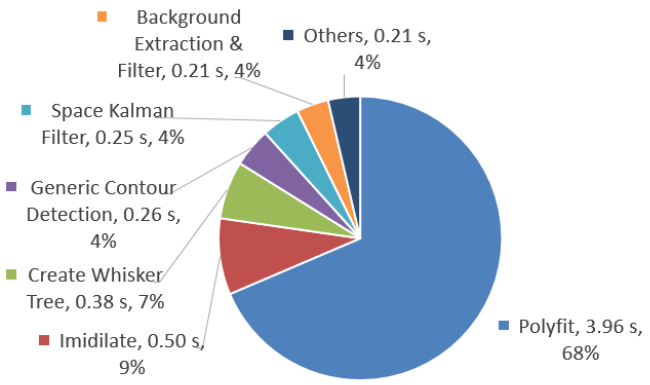

Fig. 3. Profiling of the BWTT toolbox; per-frame execution times shown

candidate space. Then, the method sorts segments according to the number of approximated nodes and runs an iterative selection. At every step, the segment that approximates the maximum number of tree nodes not yet assigned to an already selected segment is selected, leading to optimized partitioning.

Tackling this bottleneck required searching for a simplification of Polyfit or, even, for the substitution of the entire clustering methodology. An approach of lower complexity in this stage of ViSA could improve the overall performance significantly. Secondary and tertiary algorithm bottlenecks could be tackled thereafter. Another fundamental design improvement to make is the porting of ViSA to a low-level programming language, since it could intrinsically perform better than Matlab in which BWTT is written. It would also more easily adapt to a high-performance computing (HPC) system. In the end, C was chosen because it could easily be used as the starting point for porting onto different hardware platforms, especially to Maxeler Dataflow Engines (DFEs) and NVidia GPUs.

V. Implementation

The optimization steps followed in this work are illustrated in Figure 4. To verify the potential for improvement and to establish a reference point for all algorithm optimizations, some initial modifications and optimizations were made to the BWTT toolbox in Matlab while preserving the original GUI.

A. Matlab-tool improvements

ViSA traces the snout movement of the test subject with the assistance of a Kalman space filter, enabling the tracking of freely moving mice. In head-restrained animals, this is not required, thus for the current dataset the Kalman filter has been removed from the processing pipeline of the toolbox. Another change is that a display window originally depicting whisker tracking per frame in the GUI, has been disabled to improve processing speed.

Our test indicates that the task-level parallelism supported by the original toolbox by parallelizing whisker-tracking tasks is not efficient enough to offer scalable performance speedup.

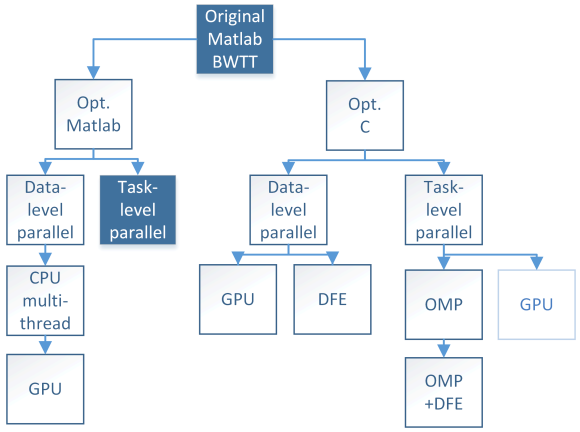

Fig. 4. Design optimization strategy followed: Dark blocks originally supported; light blocks implemented here; ultra-light blocks not implemented

Therefore, possible parallelization of individual functions was explored using data-level parallelism.

1) Data-level parallelism (CPU multithreading): The following functions are accelerated using data-level parallelism:

- *Clustering using Polyfit:* Polyfit is the foremost target to be accelerated (Figure 3). The procedure of clustering includes three loops (Figure 5). There is no data dependence in the first two loops, making this part ideal for parallelization. The third loop analyzing the whisker-tree nodes' fitting status requires the intermediate curve model from previous iterations. Therefore, it is logical to parallelize only the first two loops and append the third loop within the parallelism. As a first acceleration approach, the design was modified to run over *multiple threads*. The multithreaded version first allocates blocks of source- and sink-node pairs. Each thread accepts the node data as input and calculates the curve model using Polyfit. After the curve model is constructed, the process checks all the whisker-tree nodes to obtain intensity peaks located on or extremely near the approximated curve. This approach of multithreading causes reading conflicts while each thread has to iterate through the same set of whisker-tree nodes. Since the whisker-tree data is read-only for the clustering process, any read conflicts do not inflict a performance penalty.
- *Whisker-tree creation:* The next hotspot after Polyfit is the Matlab built-in function for image dilating, *Imdilate*. Multithreading is not adept at improving built-in functions, hence the function of image dilation is left for other acceleration methods as described below. Then, the succeeding hotspot lies in the whisker-tree generation task. The procedure for creating the whisker tree translates to a local-maxima detection problem, by examining neighboring points on the path perpendicular to the whisker-shaft direction, which is the path with the same radial coordinate in the head-centered polar system. Pixels can be grouped by radial coordinate, where local maxima detection for groups with different radial coordinate can be processed in parallel using multithreading.

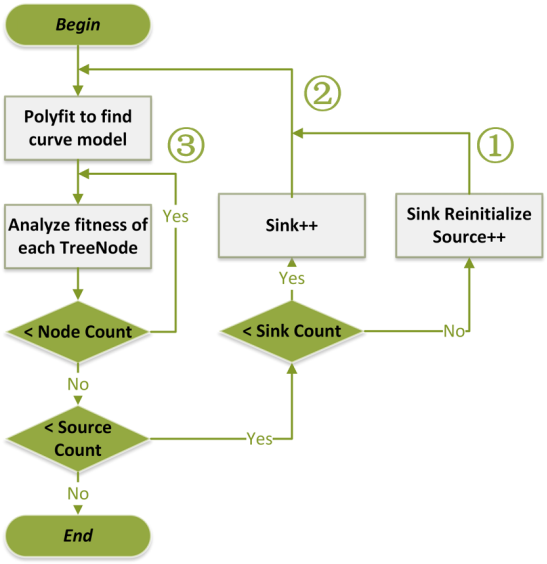

Fig. 5. Control-flow diagram of whisker-shaft clustering invoking Polyfit

2) GPU acceleration: Many Matlab built-in functions are GPU-enabled nowadays, which leads to significant acceleration of Matlab scripts originally executed on the host CPU. Inline CUDA kernels are also supported in Matlab scripts. We have, thus, accelerated further the CPU multithreading data-parallelized version by enabling the GPU enhancements:

- *Polyfit kernel:* CUDA makes usage of a concept called thread block size to allocate available jobs. The block size of the clustering problem has been constructed to be two-dimensional, where the index of source and sink nodes are the horizontal and vertical axes, respectively.
- *Image-processing kernels:* Several succeeding top hotspots of the algorithm, like the detection of generic contour and background extraction, all contain considerable portions of Matlab built-in image-processing functions. Many Matlab built-in functions have been GPU-enabled, including all the image-processing functions mentioned above. These functions support Matlab *gpuArray* objects as input arguments, designating the GPU for executing the desired function.

B. C implementations

Migrating from an interpreted to a compiled language requires changes to the coding style as well as most of the data structures. Since C is not object-oriented, all of the task and function classes had to be replaced with sequences of imperative statements and invocations of sub-functions. The design in C as a procedural language is top-down using as skeleton the control flow of ViSA and with sub-functions implementing details within each procedure.

1) C-GPU-accelerated version: GPU acceleration was also considered for the C implementation. The 2-D thread hierarchy used for the Polyfit kernel in Matlab is visualized in Figure 6. The block size is bounded by the amount of processing threads

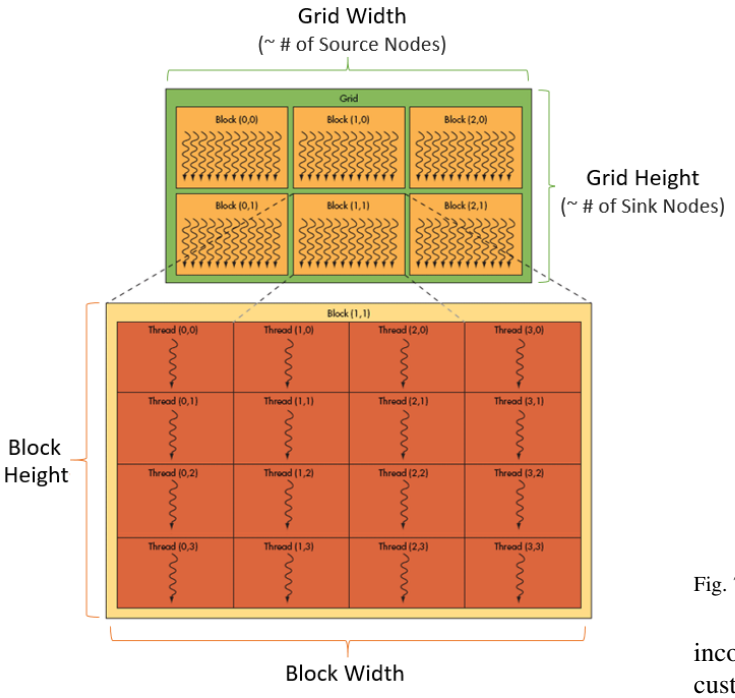

Fig. 6. 2-D CUDA-thread hierarchy for Polyfit kernel in C-GPU version

existing in the target GPU platform. The grid dimensions are determined by the problem size, for organizing enough CUDA blocks for all repetitions. Since the algorithm requires clustering each pair of source and sink nodes, it is sensible to allocate the grid height and width based on the number of source and sink nodes. Therefore, the grid size is determined by the following equations:

$$GridWidth = \left\lceil \frac{Num of Source Nodes}{BlockSize} \right\rceil, GridHeight = \left\lceil \frac{Num of Sink Nodes}{BlockSize} \right\rceil$$

where the ceiling value of the divisions is taken so as not to omit any desired execution. In this design, most of the host code runs in the CPU and the procedure for Polyfit is accelerated on the GPU.

Choosing a proper block size strongly depends on the GPU-device specifications. We used a Maxwell-based GeForce GTX Titan X GPU board, containing 24 multiprocessors, each possessing 128 threads. Therefore, in our design, the maximum amount of threads available for each block is 128. Since in Polyfit other device specifications, like memory size or bandwidth, are not the bottleneck, a suitable 2-D shape utilizing all available threads is the best design choice. Possible combinations are: 16x8, 32x4, 8x16, 4x32, 2x64, and 64x2. Our tests indicated that the amount of source nodes is normally larger than the amount of sink nodes, and usually at a ratio of 2:1, hence a block size of 16x8 was found to be the best choice for diminishing padding at the border regions.

2) C-DFE-accelerated version: Polyfit acceleration was also considered through the use of a Maxeler DFE [15]. Maxeler DFEs are HPC nodes based on FPGA technology and are most suited for stream-based processing. DFE boards also

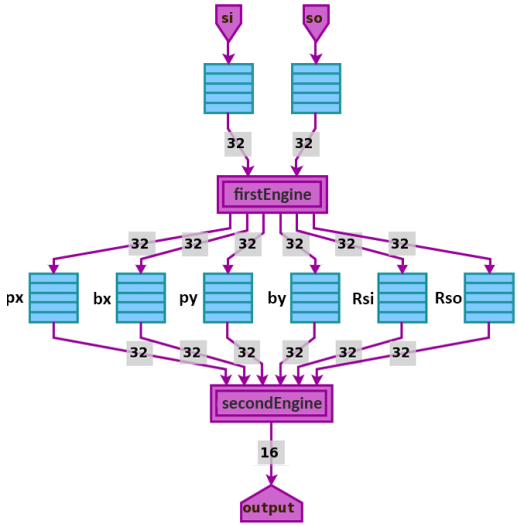

Fig. 7. Dual engine for Polyfit in C-DFE version; blue blocks are data buffers

incorporate a high-bandwidth, multichannel, highly parallel, customizable interface to the on-board DRAM resources (up to 96 GB) making it ideal for image-processing applications.

Our DFE implementation receives input data in streaming fashion and achieves acceleration through aggressive loop unrolling and super-pipelining of the dataflow kernels. For the Polyfit function, there are three nested loops to be considered, where the first two compute the curve model defined by pairs of source and sink nodes, and the third loop calculates the 3-D fitting table by going through all whisker-tree nodes. The C-DFE-accelerated implementation comprises two cascaded DFE engines for deploying this function (Figure 7). The first engine processes the streaming input of source- and sink-node indexes and generates curve models using Polyfit. Then, the model parameters of the corresponding candidate segment curves are passed to the second engine for iterative analysis on the curve fitness in the collection of the whisker tree.

The bottleneck of the streaming design lies in the second engine. Curve-model construction in the first engine executes $O(N^2)$ times, as explained in the previous section, but curve fitting executes $O(N^3)$ times. The second engine is designed to accept streaming input vectors, which groups variables together and allows simultaneous streaming of multiple variables. This approach could intrinsically flatten the iterations at the cost of more DFE resources and diminish the amount of repetition in the second engine. However, there is a restriction on the maximum vector size (in handled variables) allowed in the second engine due to FPGA resource limitations. Additionally, dataflow architectures resulting from different vector sizes may lead to dissimilar timings.

The Maxeler hardware places an additional constraint on the stream size between host and DFE. Its current SLiC (Simple live CPU) interface only supports streaming-data sizes that are multiples of 16 Bytes. Therefore, the source and sink nodes streaming in the first engine are designed to be zero-padded to satisfy the requirement of the DFE interface, albeit at an

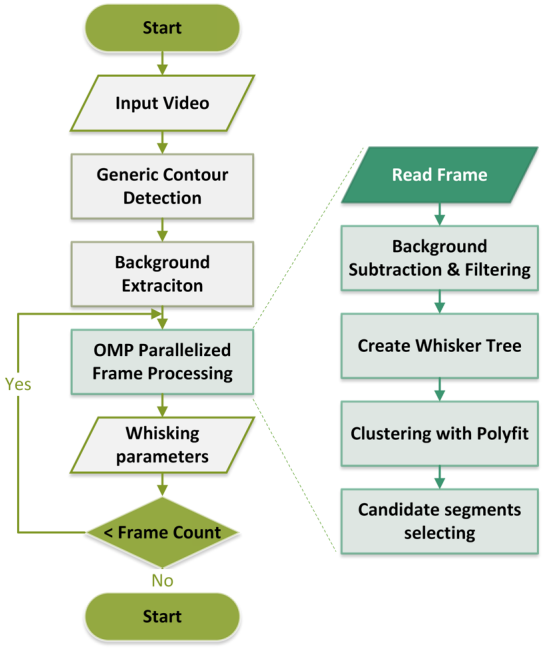

Fig. 8. Control-flow diagram of OMP version

impact on speedup. In contrast, performance is improved by duplicating the whisker tree in the DFE on-chip memory for use during fitness-table generation in the second engine, thus avoiding constant access to the host RAM.

3) OMP-accelerated version: After the removal of the Kalman space filter for snout tracing, there is no data dependency between adjacent frames. Therefore, parallelism among frames is supported, which is innately well-balanced and superior to executing multiple whisker-tracking tasks in parallel, as was originally supported in the toolbox.

To exploit the available parallelism, CPU multi-threading has been considered (task-level parallelism). We employed an Intel Xeon E5-2690 processor @2.9GHz, containing 8 cores, each with 2 threads; it could offer concurrent processing on 16 threads for our tracking problem. To this end, the sequential C version of ViSA has been adapted with Open Multi-Processing (OpenMP, OMP). OpenMP is an application-programming interface (API), which supports multi-platform, shared-memory multiprocessing programming in C. It contains a group of library routines, compiler directives and environment variables that manipulate the run-time behavior of processors. After the preprocessing phases of the algorithm, the OMP-accelerated implementation parallelizes the whisker-tracking functions for each frame, as illustrated in Figure 8.

The whole process executes in batches, where each whisker-tracking batch runs in parallel on multiple CPU threads and its results are sequentially written in an output file. The batch size was adjusted based on the system memory available for storage of intermediate data. However, the batch size has been shown to have a minimal effect on the overall performance due to the high efficiency of multithreading, as will be demonstrated in Section VI. In our modified ViSA

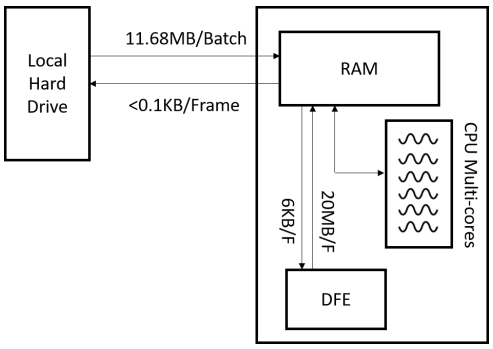

Fig. 9. System architecture of OMP+DFE version (1 Batch = 16 Frames)

algorithm, data dependencies only exist in the construction of the output array holding the whisking parameters, which needs to be stored as the sequence of video frames. The number of extracted whiskers per frame is variable but always <20. Therefore, we preallocate a maximum output array size to be able to maintain a maximum volume of parameters for the extracted whiskers.

4) OMP+DFE-accelerated version: The OMP-accelerated implementation produces satisfactory global parallelism, while the DFE-accelerated implementation has also been proven competent. Therefore, a straight-forward design option would be to combine them and create a hybrid implementation. One approach is to employ multiple dedicated DFEs so that multiple threads have a direct and independent connection each to its own DFE. However, the Maxeler SLiC cannot support DFE initialization from multiple threads for now. An alternative solution is to set an intermediate stage to collect whisker trees and relative data from multiple threads, and then stream them into a DFE kernel for pipelining (Figure 9). However, performance results of the second solution were unsatisfactory, resulting from continuously invoking synchronization barriers after the multithreaded processing of each whisker tree. Data collection in the intermediate stage breaks the parallelism by generating dependencies. Apparently, the achievable DFE acceleration cannot compensate for the barrier penalty. When the Maxeler tools begin supporting multithreaded DFE initialization, the first solution is expected to produce considerable benefits compared to the pure OMP implementation.

VI. Evaluation

A. Experimental Setup

All designs implemented here have been assessed through profiling on a group of benchmark videos. The evaluation set contains 10 randomly selected whisker videos from head-restrained mice. The detection accuracy was proven to be consistent with the output of the original BWTT. Evaluation was conducted on a server hosting an Intel Xeon E5-2690 processor with 16 threads running at a frequency of 2.9GHz and also houses a Maxeler Maia DFE. The GPU used for testing was a Maxwell-based GeForce GTX TITAN X GM200 in a separate computer box. We used the Intel VTune Amplifier software for collecting performance figures.

TABLE I
H/W-ACCELERATOR SPECIFICATIONS

Specification	Maxeler Maia DFE	NVidia Titan X
On-Board DRAM	48 GB	12 GB
RAM bandwidth	76.8 GB/s	336.5 GB/s
On-chip memory	6 MB (FPGA BRAMs)	3 MB (L2 cache)
Number of chip cores	*Not applicable*	3072 CUDA cores
Chip frequency	*Design-dependent*	1 GHz
Power consumption	140 W	250 W
IC process	65 nm	28 nm

The specifications of the Maxeler Maia DFE and NVidia Titan X GPU equipped on the server are listed in Table I. This GPU model contains 24 multiprocessors, each with 128 threads. The DFE usually cannot achieve frequencies as high as GPUs, but the dataflow execution model and the on-chip (BRAM) memory ensure higher efficiency when the workload is suitable. Last but not least, the lag between the DFE and the GPU process technology have to be taken into account when comparing the various performance results.

B. Overall performance comparison

The execution-time comparison is summarized in Figure 10, and the respective performance improvements compared to the original, serial Matlab code are illustrated in Figure 11. Initial attempts at improving the original Matlab version of BWTT by introducing data-level parallelism are proven superior to the original task-level parallelism. The GPU-accelerated, data-level parallelized version has the best processing speed among different Matlab implementations, resulting in a 12.9x speedup compared to the original serial BWTT code.

A significant performance speedup of 416.8x was observed when switching from the original Matlab code to a serial C implementation. The C-GPU and C-DFE-accelerated versions introduced further performance enhancements via porting the most computationally intensive components to the two accelerator nodes – with speedups of 573.4x and 628.4x, respectively.

Besides, by exploiting the underlying task-level parallelism, the OMP-accelerated solution achieved the best processing latency at 1.21 ms/frame, which satisfies real-time processing at more than 500 Hz frame rate. This represents a 4,767x speedup compared to the original, serial Matlab version of BWTT and a 851x speedup compared to the original, task-level parallelized Matlab version.

The final attempt to create a hybrid computing system by combining OMP and multiple DFEs did not lead to additional performance improvements because of the multithreading barriers needed for integrating the two platforms. However, the analysis in the previous section reveals that, with support by future versions of the Maxeler SLiC interface, the OMP+DFE solution could potentially enhance the overall performance further.

C. C-GPU vs. C-DFE kernel

We, next, took a closer look at the C-GPU-and C-DFE-based solutions. For the C-GPU version, a CUDA kernel was

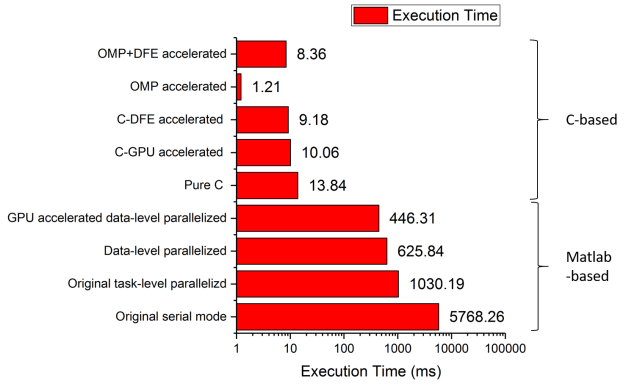

Fig. 10. Average per-frame execution-time comparison of all optimized versions of BWTT (compared to the original serial Matlab version)

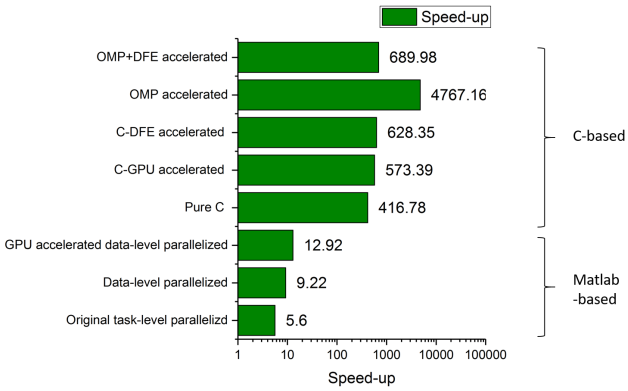

Fig. 11. Average speedup comparison of all optimized versions of BWTT (compared to the original serial Matlab version)

developed to accelerate the Polyfit function. Different 2-D block sizes which fit the specifications of the particular GPU were investigated, and results are summarized in Table II. Tests indicate that the 16x8 block size, that expands all threads per GPU core and fits best the dimension of source and sink nodes, delivers the best performance of the GPU kernel; it is the one included in the overall scores of Figures 10 and 11.

For the C-DFE version, a dual DFE kernel was implemented for accelerating Polyfit. A Maxeler DFE reconfigures its FPGA to construct an arithmetic architecture for programming specified in a particular dataflow language. Therefore, resource usage varies with different implementations, which is worth considering as a measure for performance and efficiency. For the DFE-based version of Polyfit, two cascaded DFE engines were implemented, with the second engine being the performance bottleneck due to involving more computational loops. Since acceleration in the DFE comes in part from unrolling loops, various loop-unrolling versions of the second engine were developed, resulting in different FPGA usage and speed patterns, as documented in Table III. The unrolling factor in the second kernel increases with selecting larger vector sizes, but at the cost of a higher resource usage. A vector size of 32 variables is the maximum achievable size afforded by the Maia DFE. However, the increase in vector

TABLE II
BLOCK SIZE VS. POLYFIT CUDA-KERNEL EXECUTION TIME (EXCLUDING OVERHEADS FOR COMMUNICATION AND GPU-MEMORY OPERATIONS)

Block Size	Kernel Execution Time (ms/frame)
4x4	0.971
8x8	0.359
8x16	0.213
16x8	**0.189**

TABLE III
EFFECT OF DIFFERENT VECTOR SIZES ON PERFORMANCE AND USED RESOURCES FOR THE DFE KERNEL (N/A: FREQUENCY NOT ACHIEVABLE)

VS	Resource Usage (%)				Execution Time at Set Frequency (ms/frame)			
	LUT	FF	BRAM	DSP	100 MHz	150 MHz	200 MHz	250 MHz
1	2.91	1.63	3.27	0.30	6.32	4.46	3.57	2.95
4	14.37	8.41	5.90	1.42	2.17	1.73	1.41	1.32
8	27.71	18.64	7.86	3.18	1.46	1.21	1.03	N/A
16	53.35	27.11	8.91	4.85	1.08	0.95	**0.89**	N/A
32	89.17	52.16	13.17	9.52	0.91	N/A	N/A	N/A

size is detrimental to timing, resulting in a smaller attainable chip frequency. The design choice of 16 variables as vector size achieving a clock frequency of 200 MHz results in the best overall performance.

Table IV presents the Polyfit execution results when run on the GPU and the DFE. Evaluating kernel execution time alone, the GPU achieves 29.21x speedup compared to the original, serial Matlab code. By taking into account GPU book-keeping tasks and and memory-access tasks, overall speedup drops to approximately 3.62x. The DFE kernel is 6.23x faster than the serial Polyfit. The result indicates that the DFE kernel performs slightly better than the GPU kernel (1.71x).

The Maxeler OS cannot report the initialization and data communication time for DFE kernels. However, since data is transfered through PCIe – the same bus channel between GPU and CPU – data-transfer times should be more or less similar. However, during DFE execution all intermediate data are stored in the FPGA registers close to the compute units, thus reducing off-chip memory accesses, resulting in a better performance compared to the GPU, the process-technology mismatch between the two devices notwithstanding.

D. Performance Scalability

All design implementations were tested in terms of their scalability, which could also be referred to as the performance stability with respect to the input video size or frame count processed. When employing task-level parallelism in the original Matlab code, profiling reveals relatively poor scaling properties; see Figure 12. The Matlab implementations in this project based on data-level parallelism do not suffer from the size increase of the input videos. The C implementation also scales linearly with different video sizes.

The OMP-accelerated version running on 16 threads has a processing latency of 1.2 ms/frame. Given that the application does not appear to be approaching a saturation point any time

TABLE IV
EXECUTION-TIME COMPARISON OF POLYFIT KERNELS

	Execution Time (ms/frame)				
C Polyfit	5.55				
GPU Polyfit	Cuda Malloc	Memory Copy-in	Kernel Exec.	Memory Copy-out	Cuda Free
	0.42	0.053	0.19	0.63	0.24
	1.53				
DFE Polyfit	**0.89**				

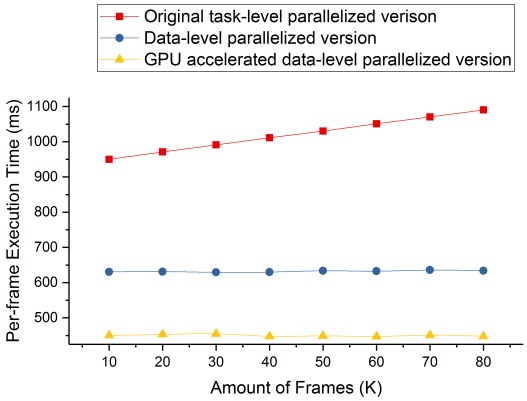

Fig. 12. Performance scaling of Matlab versions with growing input sizes

soon in the multicore CPU, it is interesting to examine its performance scalability, i.e., to determine whether this implementation with frame-level parallelism could gain benefits from using more threads. The processing speed was measured on the OMP-accelerated version using a different number of threads; the result is shown in Figure 13. With an increasing thread count, performance gains appear to be almost linear, and the extrapolation of the curve indicates that using a machine with 32 threads can bring over 19x speedup to the single-thread mode. In more detail, the OMP-accelerated version is sufficient for online processing at 500 Hz rate (or 2ms latency) using 10 threads and the ideal processing rate at 1 KHz (or 1 ms latency) using 21 threads. Additionally, in online mode, input data will arrive in batches, exempting the need for decoding videos and, thus, leading to even lower latencies.

E. Memory usage

We also examined the memory usage of all designs using the Intel VTune profiler [16]. All implementations have a low requirement on memory bandwidth (<2 GB/s) which indicates that it is not a bottleneck. The OMP-accelerated version has the largest amount of memory usage among all designs, because it uses frame-level parallelism implicitly requiring more concurrent memory access than all other versions. The maximum DRAM single-package bandwidth is 42 GB/s for the CPU architecture used, which is sufficient for the application at hand. For the C-GPU-accelerated and the C-DFE-accelerated

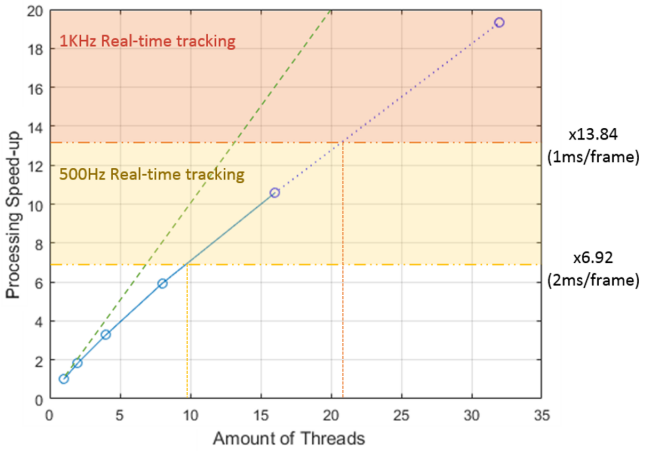

Fig. 13. Speedup scaling of OMP version using increasing numbers of threads as compared to a single-thread execution in C

versions, the memory requirements for Polyfit per each frame are approximately 22 MB/s; thus, the GPU GRAM and DFE RAM supported bandwidths are sufficient.

VII. Conclusions

In this paper, we have considered acceleration alternatives for a specific rodent whisker-tracking algorithm, ViSA, with the purpose of benefiting the neuroscientists in need of fast offline tracking (500 Hz), and ultimately online tracking (1 KHz) of video recordings. We developed six different software and hardware implementations of the algorithm and compared their performance. There are two Matlab-based versions (a data-level parallelized version and a GPU-accelerated, data-level parallelized version), which introduced computing parallelization to the original BWTT toolkit. A C version was developed, too, as a basis for further deployment to high-performance computing systems. And algorithm-level optimization was also engaged along the development process. The change in programming language led to a large performance boost, around 400x speedup compared to the original Matlab toolbox. Then, four C-based versions (a GPU version, a DFE version, an OMP version, and an OMP-DFE hybrid version) were implemented.

Data-level parallelism was first exploited to accelerate individual time-consuming steps of the algorithm. Good speedups have been achieved both in Matlab and in C, however performance bottlenecks arose due to some non-parallelizable processes. Consequently, task-level parallelism was also explored. More specifically, an OMP version of the algorithm utilizing 16 CPU threads was developed, resulting in 1.21 ms/frame processing speed and 4,767x speedup compared to the original Matlab-based algorithm. This rate fulfills the 500 Hz real-time processing requirement, and is shown to be capable of reaching 1 KHz ideal real-time performance when using a CPU with 21 threads or more. Our analysis also indicates that, if the DFE hardware and its SLiC interface supports host-code multithreaded utilization in the future, adding DFE support at the thread level shall lead to even higher performance improvements.

Acknowledgments

This work was supported by the Neuroscience department of the Erasmus MC. We are grateful to Dr. Sebastiaan Koekkoek for his help, to the NVidia Corporation for their donation of the Titan X GPU, and to Maxeler Technologies for their continuous support throughout this research effort.

References

[1] T. A. Woolsey, C. Welker, and R. H. Schwartz, "Comparative anatomical studies of the sml face cortex with special reference to the occurrence of barrels in layer iv," *J. of Comparative Neurology*, vol. 164, no. 1, pp. 79–94, 1975.

[2] L. Bosman, A. Houweling, C. Owens, N. Tanke, O. Shevchouk, N. Rahmati, W. Teunissen, C. Ju, W. Gong, S. Koekkoek, and C. De Zeeuw, "Anatomical pathways involved in generating and sensing rhythmic whisker movements," *Frontiers in Integrative Neuroscience*, vol. 5, p. 53, 2011. [Online]. Available: http://journal.frontiersin.org/article/10.3389/fnint.2011.00053

[3] J. Yu, D. A. Gutnisky, S. A. Hires, and K. Svoboda, "Layer 4 fast-spiking interneurons filter thalamocortical signals during active somatosensation," *Nature neuroscience*, 2016.

[4] "BWTT the biotact whisker tracking tool, an artefact of the eu framework 7 project biotact 215910," http://bwtt.sourceforge.net, vAIB, 2010.

[5] I. Perkon, A. Košir, P. M. Itskov, J. Tasič, and M. E. Diamond, "Unsupervised quantification of whisking and head movement in freely moving rodents," *J. of Neurophys.*, vol. 105, no. 4, pp. 1950–1962, 2011.

[6] P. Gao, R. Bermejo, and H. P. Zeigler, "Whisker deafferentation and rodent whisking patterns: behavioral evidence for a central pattern generator," *J. of Neuroscience*, vol. 21, no. 14, pp. 5374–5380, 2001.

[7] J. Wolfe, D. N. Hill, S. Pahlavan, P. J. Drew, D. Kleinfeld, and D. E. Feldman, "Texture coding in the rat whisker system: slip-stick versus differential resonance," *PLoS Biol*, vol. 6, no. 8, p. e215, 2008.

[8] N. Rahmati, C. B. Owens, L. W. Bosman, J. K. Spanke, S. Lindeman, W. Gong, J.-W. Potters, V. Romano, K. Voges, L. Moscato *et al.*, "Cerebellar potentiation and learning a whisker-based object localization task with a time response window," *J. of Neuroscience*, vol. 34, no. 5, pp. 1949–1962, 2014.

[9] J. Voigts, B. Sakmann, and T. Celikel, "Unsupervised whisker tracking in unrestrained behaving animals," *J. of Neurophysiology*, vol. 100, no. 1, pp. 504–515, 2008.

[10] P. M. Knutsen, D. Derdikman, and E. Ahissar, "Tracking whisker and head movements in unrestrained behaving rodents," *Journal of neurophysiology*, vol. 93, no. 4, pp. 2294–2301, 2005.

[11] L. Liu, K. Xu, H. Wang, P. J. Tan, W. Fan, S. S. Venkatraman, L. Li, and Y.-Y. Yang, "Self-assembled cationic peptide nanoparticles as an efficient antimicrobial agent," *Nature Nanotechnology*, vol. 4, no. 7, pp. 457–463, 2009.

[12] J. T. Ritt, M. L. Andermann, and C. I. Moore, "Embodied information processing: vibrissa mechanics and texture features shape micromotions in actively sensing rats," *Neuron*, vol. 57, no. 4, pp. 599–613, 2008.

[13] D. H. O'Connor, S. P. Peron, D. Huber, and K. Svoboda, "Neural activity in barrel cortex underlying vibrissa-based object localization in mice," *Neuron*, vol. 67, no. 6, pp. 1048–1061, 2010.

[14] T. Treiber, E. M. Mandel, S. Pott, I. Györy, S. Firner, E. T. Liu, and R. Grosschedl, "Early b cell factor 1 regulates b cell gene networks by activation, repression, and transcription-independent poising of chromatin," *Immunity*, vol. 32, no. 5, pp. 714–725, 2010.

[15] O. Pell and V. Averbukh, "Maximum performance computing with dataflow engines," *Computing in Science Engineering*, vol. 14, no. 4, pp. 98–103, July 2012.

[16] "Intel vtune amplifier performance profiler," https://software.intel.com/en-us/intel-vtune-amplifier-xe.