A Real-Time Parallel Image Processing Approach on Regular PCs with Multi-Core CPUs

Huseyin Atasoy[1], Esen Yildirim[2], Serdar Yildirim[3], Yakup Kutlu[1]
[1]*Department of Computer Engineering, Iskenderun Technical University, Hatay, Turkey*
[2]*Department of Electrical and Electronic Engineering, Adana Science and Technology University, Adana, Turkey*
[3]*Department of Computer Engineering, Adana Science and Technology University, Adana, Turkey*
huseyin.atasoy@iste.edu.tr

***Abstract*—In this paper a parallel image processing and frame rate stabilization approach is proposed. This approach works on a regular PC with a multi-core CPU. It is implemented under .NET Framework and tested on Microsoft Windows 7 operating system, performing several experiments. It is also applied to a face recognition application to increase its image processing performance successfully. Results show that, handled workload when 4 physical cores are used is approximately 5.25 times the workload handled with one core. It is also shown that the approach successfully distributes the workload on CPU cores and produces output at a stable frame rate under both steady and unsteady workloads. This approach can be used for various signal processing or multimedia applications to parallelize their tasks to increase the performance on multi-core CPUs.**

***Index Terms*—Multicore processing; image processing; software performance.**

I. INTRODUCTION

High demand for multimedia content has increased enormously for the past decade with the excessive use of social media. However, real-time image and/or video processing are computationally intensive. In order to address this challenge, graphics processing units (GPUs), which are processors, specialized for processing graphics, and technologies like NVIDIA's Compute Unified Device Architecture (CUDA) have become popular among image processing community. GPUs are specialized for graphics rendering, but their processing capabilities have also been used for non-graphics applications. GPUs have cores that can process parallel workloads. There are many studies on methods to accelerate or implement various algorithms in many fields [1]–[7], especially in image processing [8]–[12], mostly using GPUs. Most of the existing approaches have focused on parallelizing a specific algorithm by partitioning it into parallelizable steps [13]–[16]. For instance, images have been partitioned into blocks and filters have been applied on these blocks with some more operations, to make the filtering processes parallelizable in [16]. There are other approaches that allow processing images in parallel using process pipelines. In [17], a framework that consists of a data simplification and a parallel pipeline processing method has been proposed.

Manuscript received 2 February, 2017; accepted 22 July, 2017.

In this paper, we focus on order dependence and frame rate stabilization for parallel image processing on multi-core CPUs. The rest of this paper is structured as follows: First, order dependence of images and frame rate problems are explained. Then the proposed parallelization approach is presented. In the following sections the implementation, experiments and results are discussed. Finally, the paper is concluded in Section V.

A. Order Dependence and Unstable Frame Rates

A multi-core CPU is capable of executing independent processes on its cores. Processes that have to run in an order sequentially cannot directly be executed in parallel, while independent processes can be executed in parallel on the CPU cores. However, managing the order dependence of the output streams might be a challenging problem, even though input processes are independent and run in parallel. In video processing, for instance, frame order is important and processed frames should be presented in an order. Therefore, k^{th} frame has to wait for $(k-1)^{th}$ frame to be ready. In parallel processing on independent CPU cores, neither the information about frame numbers and the cores processing these frames nor the time periods of execution times are available. Therefore, processing N images in parallel will be equal to the longest time spent for each image on each core. This situation is shown in Fig. 1 for two processes and two cores. In order to display each image in its order, the program has to wait for a time t2 which is the time that the longest process takes. Managing the order of the images gets more complicated as the number of cores increased.

Another challenge in parallel processing is to maintain a stable frame rate. Even though the correct order of the output is ensured, images have to be ready in a determined time period to sustain a stable frame rate. Otherwise,

noticeable pauses might occur between frames. This problem might result in unacceptable performances especially in interactive systems.

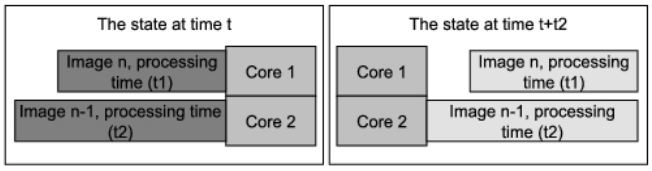

Fig. 1. The time required to process two order-dependent images on two cores in parallel.

This paper presents an approach to maintain order dependence and a stable frame rate in real-time or near real-time applications on regular multi-core CPUs.

II. METHODOLOGY

Proposed approach is implemented using circular buffers, which are well-known structures, with added special functionalities.

A. Numbered Circular Buffers

A circular buffer is a fixed-size memory region, which is considered to be connected end-to-end virtually. In the presented method, circular buffers, with additional functionalities, are used to provide available data in case of fluctuations in time taken by CPU cores while processing images.

In real-time image processing applications, showing processed images in their order is important. Therefore, each element in the buffer has to have a number to represent its order. A numbered circular buffer (NCB) is derived using two nested circular buffers, as shown in Fig. 2, in order to achieve this task.

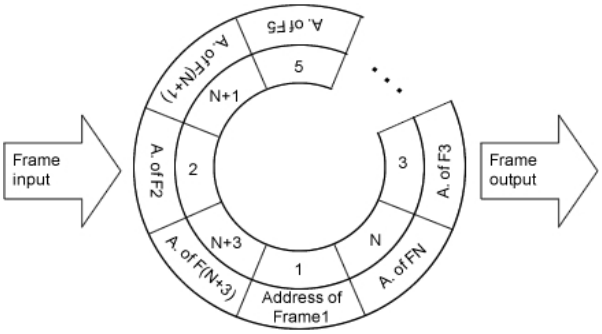

Fig. 2. A circular buffer that has N cells and sequence numbers.

Sequence numbers are written to the interior cells of the buffer without enforcing them to be consecutive. Inputs and outputs are controlled by two pointers; read pointer and write pointer. In the example shown in Fig. 2, location of each frame written to RAM is copied to the external cell whose location is stored in the write pointer. Then sequence number of this frame is written to the corresponding interior cell in the buffer.

The frame which is subsequent to the last queried frame is searched in the interior cells when a thread queries a frame from NCB. The exterior cell which corresponds to the interior cell that contains the searched number is returned as the result of the query. Hence, each frame can be read in its original order even though they do not have to be written to the buffer in their original order.

NCB is used to ensure the frame flow in the approach presented in this paper. However, some problems may arise due to instant changes in reading/writing speeds. For instance, NCB may become full if writing speed is higher than the reading speed. Similarly, NCB becomes empty if reading speed exceeds the writing speed. In order to prevent frame loss, writing operation should result in failure if NCB is full and reading operation should result in failure if NCB is empty correspondingly. In an attempt to solve this problem, interior cells are used as flags. Value of the interior cell is set to -1 when corresponding exterior cell contains no data. Therefore, exterior cells that correspond to interior cells which contain -1 are write-only, and the others are read-only. Besides, NCB has to be locked when writing or reading operation is started and unlocked after the operation is finished to prevent multiple accesses to NCB at the same time.

Two NCBs, namely, input NCB and output NCB, are required to implement the approach. The input NCB is used to keep unprocessed frames that are read from the source and processed frames and/or related data are stored in the output NCB.

B. CPU Cores and Work Sharing

In the proposed method, one of the cores of the CPU is used as the controller core and the others are worker cores. The controller core runs the following 4 threads.

Graphical User Interface (GUI) thread: It is the main thread of the application and is responsible for user interaction.

Input-writer thread: It reads frames from the source periodically at x fps (period (T) = 1000/x ms) and writes them into the input NCB. If the writing operation is resulted in a failure, it keeps trying periodically until it writes the frame into the NCB successfully.

Output-reader thread: It reads frames and/or related data from the output NCB at x fps and writes them to the target (for example to the GUI).

Evaluation thread: It evaluates measurements taken by the other threads that run on the other cores to show or save information about their current state if needed.

The worker cores are responsible for ensuring the following jobs:

– Reading the next frame with its sequence number from the input NCB;

– Processing the frame and cleaning the data that will no longer be used from the memory;

– Writing the processed frame and/or extracted data to the output NCB.

Flow of the frames on the NCBs and cores is shown in Fig. 3.

C. Frame Rate Stabilization

Every thread working on worker cores sends the image it processed to output NCB and reads the next image on input NCB and starts processing. Therefore, it is not possible to forecast the order of the frames written to output NCB and which worker core processed the frame. This might cause output NCB, which is expected to provide processed images

continuously, not to be fed by the worker cores periodically. Although buffering provides some processed frames for the output-reader thread in the case that frame processing on the cores delays, it cannot be guaranteed that there will be frames ready to be read or the buffer will not be full. Since the buffer is fed by more than one worker core and sequence numbers of the frames are disregarded while they are written to the output NCB, a deadlock case might be encountered as shown in Fig. 4.

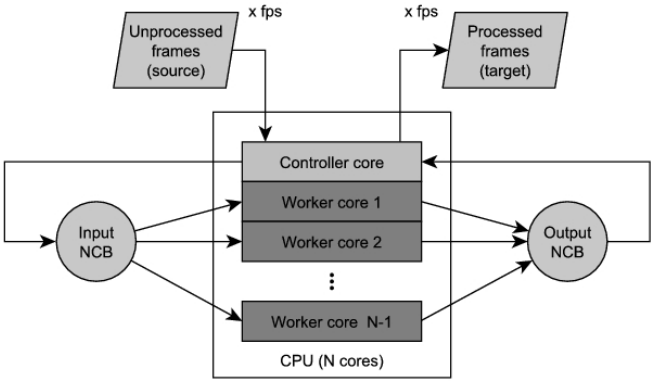

Fig. 3. Frame flow on the NCBs and cores.

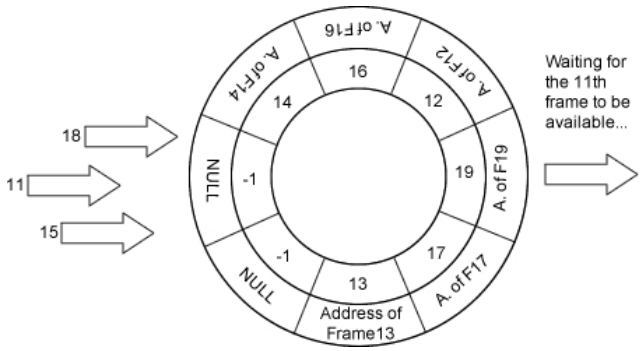

Fig. 4. A deadlock state caused by unbalanced input/output to the NCB. The reader thread waits forever for frame 11, since frame 11 will never be able to be written to the NCB.

In Fig. 4, the output reader thread is waiting for frame 11 to be available at the output NCB. When frame 18, 15 and 11 are made ready by the worker cores in the order shown in Fig. 4, there are only two empty cells in the NCB and they will be filled with frame 15 and frame 18. The thread that made frame 11 ready waits for a cell to be read and empty. This prevents the reader thread that is waiting for frame 11 to continue. This deadlock causes the other threads that process the other frames, to wait for a cell to be empty and frame flow is cut off entirely.

Because of these reasons, the difference between locations of the read pointer and the write pointer has to be kept in a certain range without breaking the frame flow or the frame rate. The problem and the solution can be expressed more effectively using a simple metaphor. Let W(rite pointer) and R(ead pointer) be two vehicles on a circular track (Fig. 5). The speed of the vehicle R is constant while the speed of the vehicle W is not. In order to prevent a possible crash, the safest distance between the vehicles is equal to the half of the circular track length. Since R has no control over the vehicle W, R has to keep the distance safe by speeding up or slowing down. However, acceleration of R should be kept low in order to prevent speed of R to be unstable.

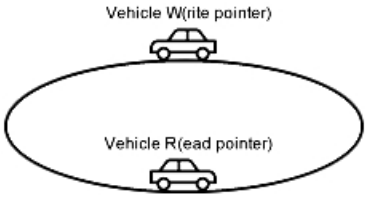

Fig. 5. The safest distance between two vehicles on a circular track.

Since the write pointer moves when a core finishes processing its frame, the speed of the write pointer depends on speeds of the cores. It can be changed only by delaying writing operations of the cores. However, the speed of the reading operations from the output NCB can be changed without reducing the image processing speed. Speed of the read pointer has to be changed interfering the reading period (t_w, (1)), because the read pointer moves periodically

$$t_w = \max\left(0, \left(T_0 - t_p\right) - T_0\lambda\right), \tag{1}$$

where t_w is the time for which the thread has to wait before it reads the next frame, T_0 is the frame reading period, t_p is the time spent on the reading operation. λ (hereinafter referred to as waving factor) is the coefficient that will change the period to keep the distance between the frame numbers which read and write pointers point to, safe while stabilizing the frame rate.

D. Linear Waving Factor

Waving factor is used to keep the distance (X in (2)) between the numbers of the frames which the read and write pointers point to, equal to half of the capacity of the NCB which is the safest value. Thus, the system aims to keep the difference between X and $N/2$ (H in (3)) equal to 0 by speeding up or slowing down the reading speed by means of the waving factor:

$$X = X_w - X_r, \tag{2}$$

$$H = X - \frac{N}{2}, \tag{3}$$

where X_r and X_w are not physical addresses of the frames, they are numbers of the frames which the read and write pointers point to, respectively. Waving factor is computed as

$$\lambda_l = \frac{H}{\frac{N}{2}} = \frac{2\left(X_w - X_r\right)}{N} - 1. \tag{4}$$

The range of waving factor is between [-1, 1] since maximum and minimum values of X are N and 0.

E. Exponential Waving Factor

The linear waving factor λ_l changes the period when H is not equal to 0. This may cause sharp changes in period. In order to make changes softer, exponential waving factor (λ_e) is derived. For a predefined positive constant $A > 1$, $A \in Z^+$, (5) shows the exponential waving factor.

A bends linear waving factor line toward $+x$, $-y$ and $-x$, $+y$ (Fig. 6). Therefore, the exponential waving factor softens the change of the period when the difference between optimal

distance and distance between the numbers which the pointers point to is close to zero. The value of the exponential waving factor increases exponentially. In critical situations (when the NCB is about to be full or empty) the waving factor can extend or shorten the period enough to maintain the optimal distance. Note that, it is possible to bend the curve more, using higher A values. But a very high A value may cause the change in the period to be insufficient even in critical situations

$$\lambda_e = \left(A^{|\lambda_l|-1} - A^{-1}\right)\mathrm{sgn}(\lambda_l). \quad (5)$$

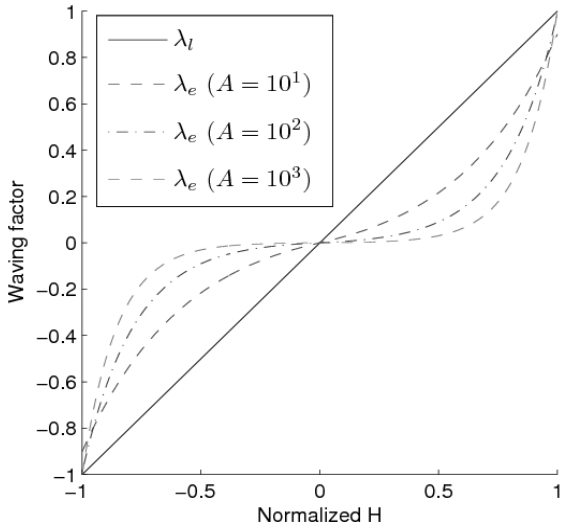

Fig. 6. Normalized H (equation (3)) and waving factor curves.

III. IMPLEMENTATION

The proposed approach was implemented under Microsoft .NET Framework as a .NET library. Since Windows is not a real-time operating system, latencies between when timers have to expire and when they are expired were measured to make sure that the system is implementable in practice. Those measurements and the results are discussed in the following subsection.

A. Timer Resolution

Timer resolution determines how frequently the operating system checks if a timer has expired. Sensitivity of time measurements depends directly on timer resolution of the system. Low timer resolutions cause problems for operations that have to be run periodically on the system since it may take more than one period for the system to realize that one period has elapsed. This might cause problems in real-time applications. For instance, an application that processes 25 frames per seconds (fps) and takes 35 milliseconds (ms) to process each frame has to wait 5 ms after processing. If this application is running on Microsoft Windows 7, sleeping for 5 ms approximately takes 15 ms since the default value of the timer resolution is 15.6 ms. This causes the period to increase to 50.6 ms and the speed to reduce to 20 fps.

Although Microsoft Windows is not a real-time operating system, it is possible to change timer resolution of the system to make it closer to real-time. In this study, various timer resolutions are tested. Timer resolution was set to 15.6 ms, 10 ms, 5 ms, 1 ms and 0.5 ms and a thread, that requests to sleep for 1 ms 20 times, is executed at each trial. Times elapsed between the time that thread is put to sleep and is woken up were measured. The results are shown in Fig. 7.

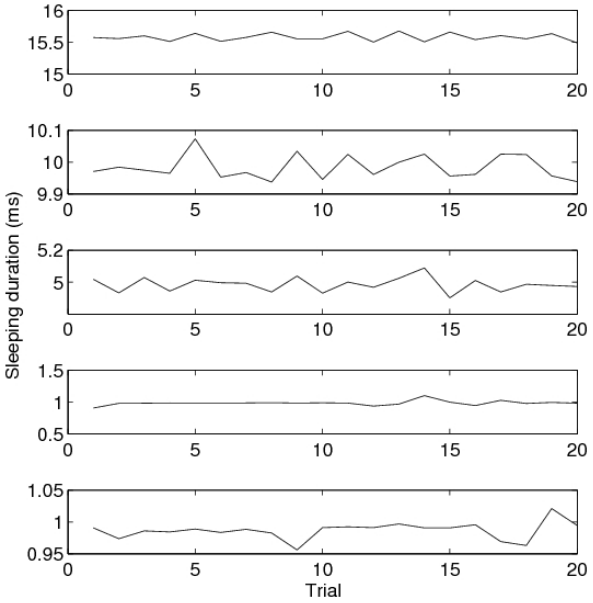

Fig. 7. Sleeping durations of a thread that requests to sleep for 1 ms at timer resolutions of 15.6 ms, 10 ms, 5 ms, 1 ms and 0.5, respectively.

The deviation at the lowest and also the default resolution (15.6 ms) is approximately 14.5 ms. This is very high for an image processing application that is intended to run near real-time. But at the highest resolutions (1 ms and 0.5 ms) the deviation did not exceed the values 0.1 ms and 0.05 ms respectively.

Since the approach was implemented under .NET Framework and the minimum value that *Sleep()* function accepts is 1 ms, the resolution was set to 1 ms in the experiments.

IV. EXPERIMENTS AND RESULTS

Two sets of experiments were performed to test the proposed algorithm. First, the approach was tested on a dummy image processing tasks to evaluate performance enhancement under steady workloads (E1-E7). Then it was applied to a real-time face recognition application and tested using parameters that require more processing capability than one core has, under unsteady workloads (E8-E13).

All the experiments were performed on a regular desktop computer with an Intel i7-3770 3.40 GHz CPU (4 physical cores and 2 logical cores per each) and 8 GB of RAM. Average percentage of usage of each core and instant frame rates were recorded by the evaluation thread during the executions.

A. Experiments on a Dummy Image Processing Task

In these experiments, a video containing 750 frames (1280 × 720 pixels) was used. The implementation was tested with a dummy image processing task which consists of the following steps:

– Convert image to gray-scale and make two copy of it;
– Equalize histogram of the first copy;

– Convolve a 2 × 2 averaging filter with the first copy, repeat this step M times;

– Draw the second copy to the first copy from 20 pixels inside the edges.

These steps were performed using Open Source Computer Vision (OpenCV) libraries. *M* was considered as the workload unit. Its values were selected so that the program can produce output at 25 fps using CPU cores around 85 %–95 % on average. For example when the task was being executed on a single core, for $M = 8$ the average core usage is about 85.45 % and for $M = 9$, is about 100 %. Therefore in this example $M = 8$ is used to achieve a stable frame rate. Also, the capacities of the input and output NCBs are set to 5 and 25, respectively. Note that the safest distance between frame numbers which the pointers point on the output NCB is 12.5 since the frame rate is 25.

In the first experiment (E1), all reading, processing, writing operations and the GUI thread were executed on the same core. One core was able to handle workload *M* from 1 to 8. Hence, $M = 8$ is considered as the baseline. On the next 3 experiments, the dummy image processing task was executed in parallel using the presented approach.

In experiment 2 (E2), proposed algorithm is used to distribute the workload onto 4 logical cores (on 2 physical cores). Although the number of cores is quadrupled, the workload value was less than 4 times of the baseline value since one core is used as controller and the others as workers in the presented method. Besides, usage of two logical cores on the same physical core degrades the performance.

One logical core on each of 4 physical cores is used in experiment 3 (E3). Although 4 logical cores are used in both experiments E2 and E3, handled workload increases to 30 in E3 because of the fact that each logical core is on a separate physical core.

The tasks were executed using all the physical cores, including two of the logical cores in each physical core, in the last experiment (E4). As can be seen from the Table I, maximum performance, with a handled workload of M = 42, is obtained in E4, where all the logical cores were used.

All those 4 experiments were performed using the linear waving factor. The waving factor does not affect percentage of usage of the cores, because both of the linear and exponential waving factors can be applied consuming ignorable amount of CPU time. The fourth experiment was repeated using the linear and exponential waving factors with different *A* values (E5, E6 and E7) in order to see the effect of waving factor parameter on the proposed algorithm. Instant frame rates are shown in Fig. 8, Fig. 9, Fig. 10 and Fig. 11 for E4, E5, E6 and E7. Statistical values are given in Table II.

TABLE II. MAXIMUM / MINIMUM VALUES, MEANS AND STANDARD DEVIATIONS OF THE OBTAINED INSTANT FRAME RATES WHEN LINEAR AND EXPONENTIAL WAVING FACTORS WERE USED.

Experiment	Waving factor	Instant frame rates			
		max	min	mean	std. dev.
E4	λ_l	34.52	21.68	25.57	2.99
E5	λ_e ($A = 10^1$)	28.58	22.46	25.28	0.86
E6	λ_e ($A = 10^2$)	26.33	22.45	25.13	0.48
E7	λ_e ($A = 10^3$)	26.32	22.62	25.08	0.44

X value (1) is zero at the beginning and it takes some time to bring the distance to its safest value, as it is explained in Section II.C. Besides, when the output NCB is about to be empty, the algorithm tries to extend the period to keep the distance safe. Therefore, instant frame rates are unstable at the beginning and the end of the process for a short period of time. For this reason, the first and the last 4 seconds were ignored while calculating the results given in Table II.

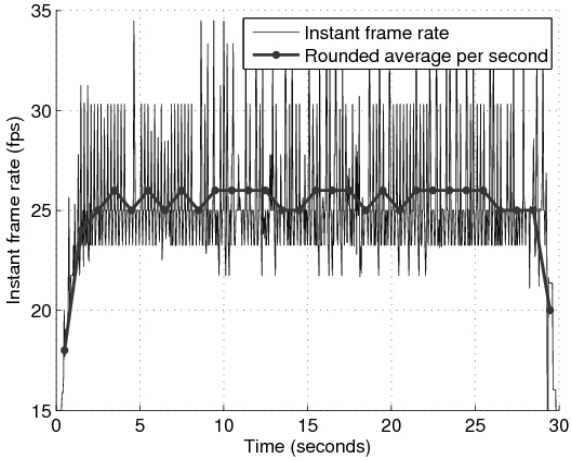

Fig. 8. Obtained frame rates when linear waving factor was used (E4).

Table II shows that mean frame rates for linear and exponential waving factors are stable. However, standard deviation of the frame rates when linear waving factor is used is high resulting in a highly varying frame rates as seen in Fig. 8. Standard deviation is much lower when exponential waving factor is used. Standard deviations are observed as 0.86, 0.48, 0.44 for $A = 10^1$, $A = 10^2$ and $A = 10^3$ respectively.

Figure 12 shows core usages during the best experiment (E7 for $A = 10^3$). Since the first core is the controller core, it is used only for reading images from the source, displaying them, collecting data about the frame rates and core usages and responding to the user continuously. Worker cores use their processing capacities only to process images they take from input NCB.

TABLE I. WORKLOADS AND CORE USAGES DURING THE FIRST 4 EXPERIMENTS. C1, C2, ..., C8 ARE THE LOGICAL CORES AND EACH COUPLE REPRESENTS ONE PHYSICAL CORE (C1-C2, C3-C4, C5-C6, C7-C8). (P: PHYSICAL, L: LOGICAL).

Experiment	Number of used cores	Workload	Core usages (average %)							
			C1	C2	C3	C4	C5	C6	C7	C8
E1	1L(1P)	8	85.45	-	-	-	-	-	-	-
E2	4L(2P)	20	20.18	88.09	88.64	87.82	-	-	-	-
E3	4L(4P)	30	17.61	5.46	95.88	0.36	94.76	0.28	95.01	0.21
E4	8L(4P)	42	21.53	86.63	90.3	92.52	90.33	91.46	89.37	90.32

TABLE III. MAXIMUM / MINIMUM VALUES, MEANS AND STANDARD DEVIATIONS OF THE OBTAINED INSTANT FRAME RATES WHEN LINEAR AND EXPONENTIAL WAVING FACTORS WERE USED ON THE REAL-TIME FACE RECOGNITION APPLICATION.

Experiment	Target frame rate (fps)	Waving factor	Number of used cores	Instant frame rates (fps)			
				max	min	mean	std. dev.
E8	25	λ_l	8L(4P)	25.60	22.20	25.14	0.45
E9	25	λ_e ($A = 10^1$)	8L(4P)	25.00	24.40	24.80	0.28
E10	25	λ_e ($A = 10^2$)	8L(4P)	25.60	24.40	24.95	0.36
E11	25	λ_e ($A = 10^3$)	8L(4P)	25.00	23.80	24.63	0.30
E12	60	λ_e ($A = 10^2$)	8L(4P)	66.70	52.60	59.56	2.40
E13	60	λ_e ($A = 10^2$)	4L(2P)	66.70	47.60	58.37	3.55

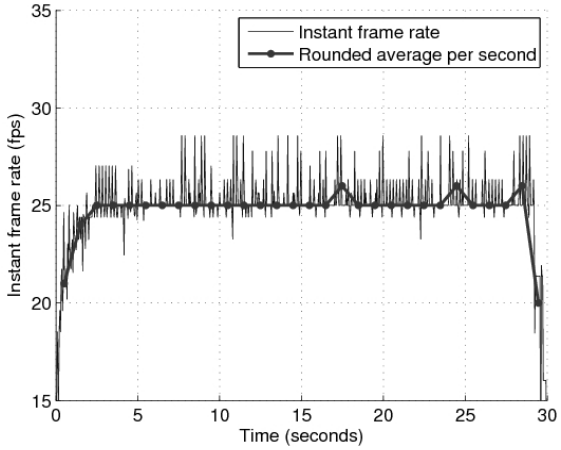

Fig. 9. Obtained frame rates when exponential waving factor was used with A = 10^1 (E5).

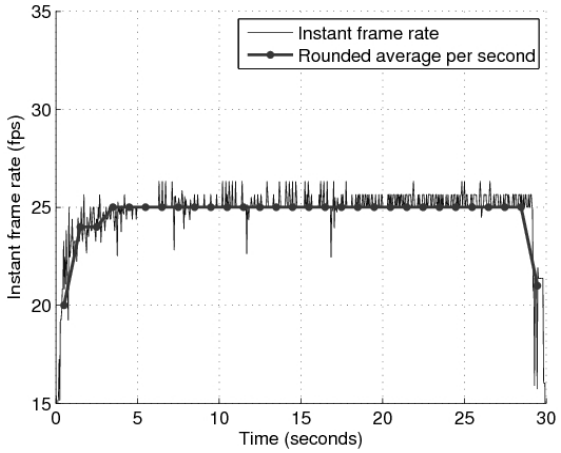

Fig. 10. Obtained frame rates when exponential waving factor was used with $A = 10^2$ (E6).

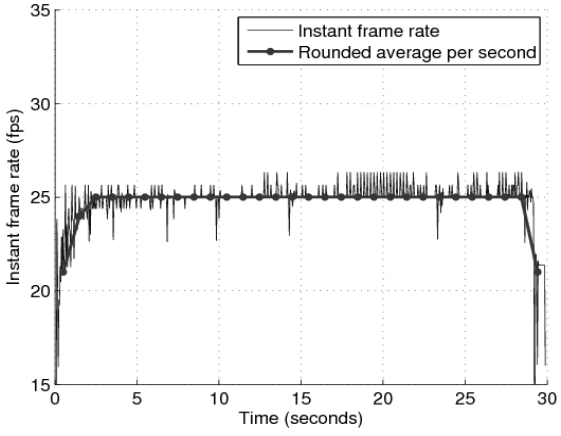

Fig. 11. Obtained frame rates when exponential waving factor was used with $A = 10^3$ (E7).

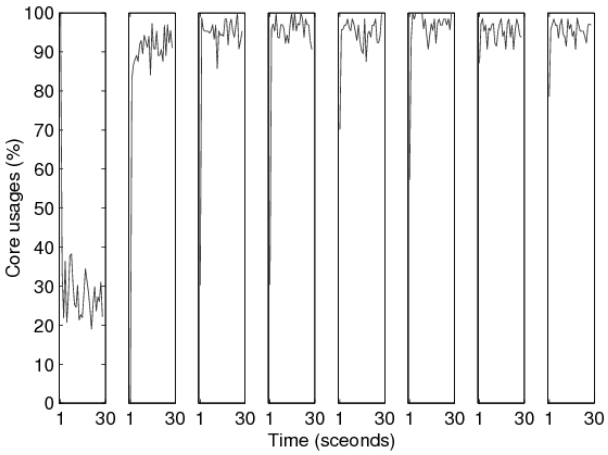

Fig. 12. Usages of cores during the execution in E7. The first core is the controller core and the others are the worker cores.

B. Experiments on a Real-time Face Recognition Application

A real-time face detection and recognition application that was implemented under Microsoft .NET Framework is parallelized using the presented approach to test the approach on a real image processing application both on a single core and on multiple-cores.

In the application, faces are detected using Viola/Jones face detection algorithm [18] and recognized using principal component analysis [19]. The approach is applied to the application to increase image processing speed of the application, not to increase its face recognition rate.

Workloads that were used in the previous experiments are constant. However, face recognition process requires detection of faces on images and workload caused by this process may vary from image to image. Therefore, these experiments are important to show that the approach can provide desired frame rates under unsteady workloads.

6 experiments were performed on the face recognition application using two video files (resolution: 640 × 360). In the first 4 experiments (E8-E11), linear and exponential waving factors were used on the first video file to obtain 25 fps. On the last two experiments (E12, E13), the second video file was processed at 60 fps using the exponential waving factor with A value that provide the closest average frame rate to the desired frame rate in previous 4 experiments. The results are shown in Table III.

In the first 5 experiments (E8-E12) the workloads were at levels that the cores can handle easily. The best result was obtained in E10 considering the differences between desired

frame rates and obtained mean frame rates. However, in E13 only 2 physical cores were used. Besides, the workload was increased decreasing minimum face size to be searched on images to generate a workload that cannot be handled by the cores.

Instant frame rates obtained in E8 and E13 are shown in Fig. 13 and Fig 15. In Fig. 13, peaks at after about 9th second show that the waving factor shortened the period to prevent NCB from being full. In Fig. 14, the frame rate started to drop at about 7th second because of the excessive workload on two physical cores. This time the waving factor tried to extend the period to prevent the NCB from being empty.

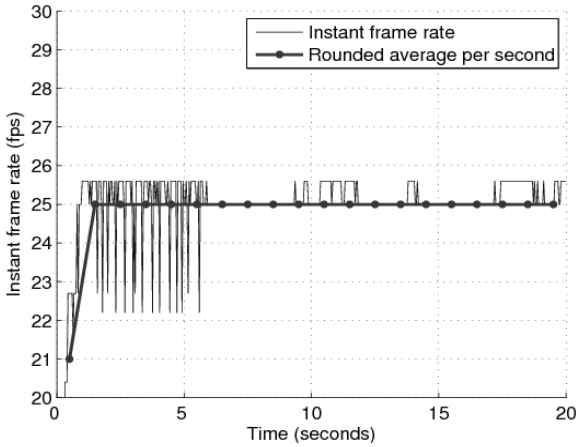

Fig. 13. Obtained frame rates on the face recognition application in E8.

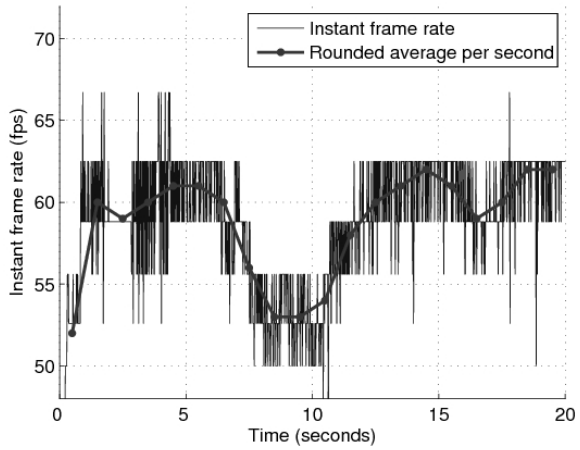

Fig. 14. Obtained frame rates on the face recognition application in E13.

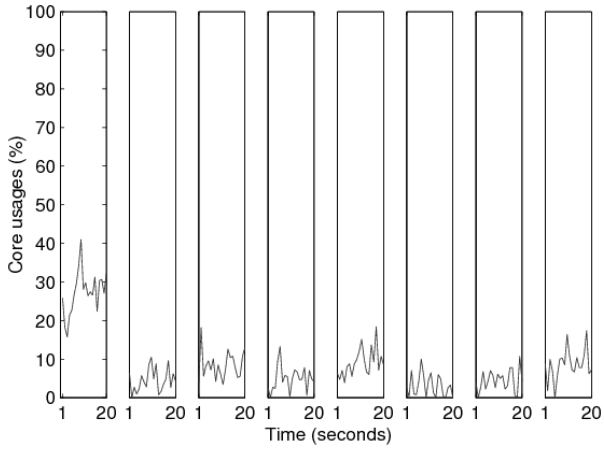

Fig. 15. Usages cores during the execution of the face recognition application in E11.

Figure 15 and Fig. 16 show core usages during E11 and E13. The workload was successfully distributed to the cores in E11 as can be seen in Fig. 15. On the other hand, processing capacities of the first two physical cores were not sufficient to handle the workload although their capacities were being used fully, in E13 as shown in Fig. 16. Nevertheless the approach could still provide frame rates close to desired values in that experiment.

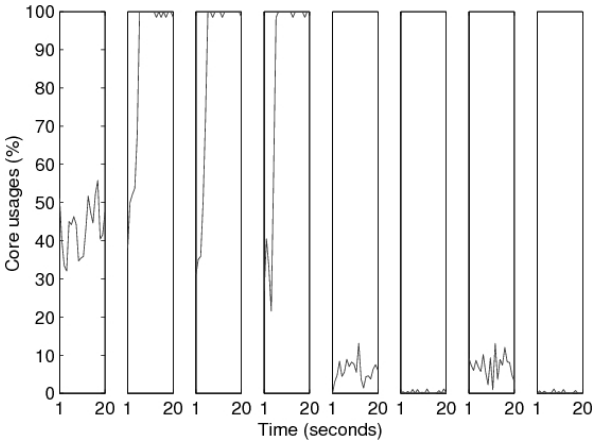

Fig. 16. Usages of cores during the execution of the face recognition application in E13.

V. CONCLUSIONS

In this paper, a real-time image processing and frame rate stabilization approach, which works on regular multi-core CPUs and does not use an explicit hardware such as GPU or CUDA, was proposed. It was implemented as a Microsoft .NET library and tested using various conditions; 2 physical (4 logical) cores, 4 physical (4 logical) cores and 4 physical (8 logical) cores. Results were compared to single core results.

Maximum workload handled by a single core is $M = 8$. The best performance is achieved using 8 logical cores. A workload of $M = 42$, which is 5.25 times the workload handled by a single core, is handled. Results show that the workload was distributed to all worker cores equally, such that the percentage of the core usages was about 90 %. Maximum workload achieved by 4 logical (4 physical) and 4 logical (2 physical) cores are about 3.75 and 2.5 times the maximum workload handled by a single core respectively. Besides, the results show that proposed approach can provide stabilized frame rates successfully.

The library was also tested under unsteady workloads on a real-time face recognition application. Workloads were successfully distributed to the cores again and desired frame rates were obtained. The approach provided acceptable frame rates even under excessive workloads.

The presented approach can also be applied to parallelize various multimedia or real-time signal processing applications which take order-dependent data packets as input as the face recognition application does. For instance, it can be used to recognize speech, to process EEG data, to encode / decode media formats etc., in parallel, in real-time.

REFERENCES

[1] J. Kruger, R. Westermann, “Linear algebra operators for gpu

implementation of numerical algorithms", *ACM Trans. Graphics (TOG)*, vol. 22, pp. 908–916, 2003. Online. [Available]: http://dx.doi.org/10.1145/882262.882363

[2] K. S. Oh, K. Jung, "Gpu implementation of neural networks", *Pattern Recognition*, vol. 37, no. 6, pp. 1311–1314, 2004. Online. [Available]: http://dx.doi.org/10.1016/j.patcog.2004.01.013

[3] D. Cederman, P. Tsigas, "GPU-quicksort: A practical quicksort algorithm for graphics processors", *Journal of Experimental Algorithmics*, vol. 14, no. 4, pp. 1–22, 2009. Online. [Available]: http://dx.doi.org/10.1145/1498698.1564500

[4] M. Andrecut, "Parallel GPU implementation of iterative pca algorithms", *Journal of Computational Biology*, vol. 16, no. 11, pp. 1593–1599, 2009. Online. [Available]: http://dx.doi.org/10.1089/cmb.2008.0221

[5] C. Obrecht, F. Kuznik, B. Tourancheau, J.-J. Roux, "Multi-gpu implementation of the lattice Boltzmann method", *Computers & Mathematics with Applications*, vol. 65, no. 2, pp. 252–261, 2013. Online. [Available]: http://dx.doi.org/10.1016/j.camwa.2011.02.020

[6] C. Jermain, G. Rowlands, R. Buhrman, D. Ralph, "GPU accelerated micromagnetic simulations using cloud computing", *Journal of Magnetism and Magnetic Materials*, vol. 401, pp. 320–322, 2016. Online. [Available]: http://dx.doi.org/10.1016/j.jmmm.2015.10.054

[7] O. Schutt, P. Messmer, J. Hutter, J. VandeVondele, "GPU accelerated sparse matrix multiplication for linear scaling density functional theory", *Electronic Structure Calculations on Graphics Processing Units: From Quantum Chemistry to Condensed Matter Physics*, pp. 173–190, 2016. Online. [Available]: https://doi.org/10.1002/9781118670712.ch8

[8] S. N. Sinha, J.-M. Frahm, M. Pollefeys, Y. Genc, "GPU-based video feature tracking and matching", *Workshop on Edge Computing Using New Commodity Architectures (EDGE)*, vol. 278, pp. 4321–4335, 2006.

[9] J. Chen, S. Paris, F. Durand, "Real-time edge-aware image processing with the bilateral grid", *ACM Trans. Graphics (TOG)*, vol. 26, no. 3, 2007. Online. [Available]: http://dx.doi.org/10.1145/1276377.1276506

[10] D. Castano-Diez, D. Moser, A. Schoenegger, S. Pruggnaller, A. S. Frangakis, "Performance evaluation of image processing algorithms on the GPU", *Journal of Structural Biology*, vol. 164, no. 1, pp. 153–160, 2008. Online. [Available]: http://dx.doi.org/10.1016/ j.jsb.2008.07.006

[11] Vu Pham, Phong Vo, Vu Thanh Hung, Le Hoai Bac, "Gpu implementation of extended gaussian mixture model for background subtraction", in *IEEE RIVF Int. Conf. Computing and Communication Technologies, Research, Innovation, and Vision for the Future (RIVF 2010)*, Hanoi, Vietnam, 2010. Online. [Available]: http://dx.doi.org/10.1109/RIVF.2010. 5634007

[12] P. Karas, V. Morard, J. Bartovsky, T. Grandpierre, E. Dokladalova, P. Matula, P. Dokladal, "GPU implementation of linear morphological openings with arbitrary angle", *Journal of Real-Time Image Processing*, vol. 10, no. 1, pp. 27–41, 2015. Online. [Available]: http://dx.doi.org/10.1007/s11554-012-0248-7

[13] R. K. Satzoda, S. Suchitra, T. Srikanthan, "Parallelizing the hough transform computation", *IEEE Signal Processing Letters*, vol. 15, pp. 297–300, 2008. Online. [Available]: http://dx.doi.org/10.1109/LSP.2008.917804

[14] A. Buttari, J. Langou, J. Kurzak, J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures", *Parallel Computing*, vol. 35, no. 1, pp. 38–53, 2009. Online. [Available]: http://dx.doi.org/10.1016/j.parco.2008.10.002

[15] Y. Lu, H. Zhou, L. Shang, X. Zeng, "Multicore parallel min-cost flow algorithm for cad applications", in *Proc. 46th Annual Design Automation Conf., (ACM 2009)*, 2009, pp. 832–837. Online. [Available]: http://dx.doi.org/10.1145/1629911.1630124

[16] D. Akgun, "A practical parallel implementation for tdlms image filter on multi-core processor", *Journal of Real-Time Image Processing*, pp. 1–12, 2014. Online. [Available]: http://dx.doi.org/10.1007/s11554-014-0397-y

[17] M. B. Fattepur, J. B. Huttanagoudar, "Processing videos using parallel computing: A novel approach", *International Journal of Innovative Technology and Research*, pp. 214–219, 2015.

[18] P. Viola, M. J. Jones, "Robust real-time face detection", *International journal of computer vision*, vol. 57, no. 2, pp. 137–154, 2004. Online. [Available]: http://dx.doi.org/10.1023/B:VISI.0000013087.49260.fb

[19] M. A. Turk, A. P. Pentland, "Face recognition using eigenfaces", in *IEEE Computer Vision and Pattern Recognition*, pp. 586–591, 1991. Online. [Available]: http://dx.doi.org/10.1109/CVPR.1991.139758