New Trends and Issues Proceedings on Advances in Pure and Applied Sciences

No 12 (2020) 109-118

www.propaas.eu

Performance analysis and GPU parallelisation of ECO object tracking algorithm

Ugur Taygan*, Department of Computer Engineering, Hacettepe University, 06800Ankara, Turkey
https://orcid.org/0000-0002-2003-2187
Adnan Ozsoy, Department of Computer Engineering, Hacettepe University, 06800 Ankara, Turkey
https://orcid.org/0000-0002-0302-3721

Suggested Citation:
Taygan, U. & Ozsoy, A. (2020). Performance analysis and GPU parallelisation of ECO object tracking algorithm. *New Trends and Issues Proceedings on Advances in Pure and Applied Sciences.* (12), 109–118.

Received date January 15, 2020; revised date February 11, 2020; accepted date April 20, 2020.
Selection and peer review under responsibility of Prof. Dr. Dogan Ibrahim, Near East University, Cyprus.

Abstract

The classification and tracking of objects has gained popularity in recent years due to the variety and importance of their application areas. Although object classification does not necessarily have to be real time, object tracking is often intended to be carried out in real time. While the object tracking algorithm mainly focuses on robustness and accuracy, the speed of the algorithm may degrade significantly. Due to their parallelisable nature, the use of GPUs and other parallel programming tools are increasing in the object tracking applications. In this paper, we run experiments on the Efficient Convolution Operators object tracking algorithm, in order to detect its time-consuming parts, which are the bottlenecks of the algorithm, and investigate the possibility of GPU parallelisation of the bottlenecks to improve the speed of the algorithm. Finally, the candidate methods are implemented and parallelised using the Compute Unified Device Architecture.

Keywords: Object tracking, parallel programming.

* ADDRESS FOR CORRESPONDENCE: **Ugur Taygan,** Department of Computer Engineering, Hacettepe University, 06800 Ankara, Turkey. *E-mail address*: utaygan@aselsan.com.tr

1. Introduction

Detection, classification and tracking of objects are very basic problems in terms of signal processing. Studies on this subject, which have applications in many different fields, such as medical diagnostic systems, autonomous vehicles, astronomy, human–machine interaction and weapon systems, remain up to date [20]. In this regard, object detection, classification and object tracking have always been the focus of attention in order to provide functionality in daily life, as well as in the field of security and military.

Object detection and classification focus on extracting information about objects in images [22], while object tracking focuses on finding the location of related objects in each image [20]. More broadly, object detection is the process of understanding what objects are in the image; object tracking is the process of finding the same objects which have been seen in the previous image. Object tracking is only intended to estimate the trajectory of an object whose initial state has been known in an image. Object tracking has many practical applications, such as autonomous vehicles, video surveillance and target detection.

In recent years, the use of General Purpose Graphics Processing Units (GPGPUs) has become popular in areas like computer vision [7] and big data problems [21]. GPGPUs use the advantage of having many processor units. They are slower in clock speed when compared to the common central processing unit (CPU), but can handle and execute instructions on many threads simultaneously. Nowadays, while a general CPU usually has 8 or 16 cores, a brand new NVidia GPU has more than 3,000 cores. Even the GPU cores are usually slower than CPU cores and the computational power of GPUs are much greater due to the enormous number of cores.

Object detection and tracking algorithms are good candidates to be implemented on GPUs because they usually have a parallelisable nature which can be divided into many threads.

In this paper, an analysis of an object tracking algorithm is carried out on [9], [17], VOT2019 [10], OTB-100 [15], TLP [11] and UAV123 [11] datasets on the Linux operating system with a computer with Intel Core i7-7500U CPU @2.7GHz, 16GB RAM. The suitability of GPU parallelisation of the algorithm is discussed in order to realise real-time processing speeds. Finally, based on the analysis carried out, the parallelisation of a portion of the algorithm is implemented on GeForce 940MX using the Compute Unified Device Architecture (CUDA).

CUDA is, introduced by the NVIDIA Corporation, a parallel programming model which is used to execute programmes written with different programming languages on NVIDIA GPUs. In this work, we used NVIDIA's CUDA C++ extension [3] for analysis and implementations.

In 'Method', the relevant background is introduced about the selected algorithm. The information about the datasets used for analysis is provided in 'Profiling the function calls'. The suitability of GPU parallelisation of the algorithm is discussed in 'GPU parallelisation of element-wise matrix multiplication and results' with the use of a profiler. The parallelisation of a portion of the algorithm is presented with the analysis results in 'Conclusion and future work'. Finally, in 'Section 6', future studies are mentioned.

2. Method

2.1. Discriminative correlation filter

Discriminative correlation filter (DCF) usage has increased in object tracking algorithms in recent years [1], [13], [19]. The first use of the DCF method in object detection dates back to the early 1980s in the work by [8]. It became popular with the MOSSE [2] object tracking algorithm developed by [2]. The DCF method has been observed to significantly improve the tracking performance of the algorithm, but caused a decrease in its speed.

DCFs are usually trained with the first samples taken from image sequences online. It is aimed to increase the reliability of the tracking process by updating the filter with the following image sequences. However, increasing filter updating can take a significant amount of time to calculate. On the other hand, if the successive image sequences have similar characteristics, they can cause an unnecessary waste of time.

In filter-based trackers, the first image is often used to initialise the filter. After the filter is initialised with the first image, object tracking and updating of the filter with new sequence outputs are carried out together. The position of the highest correlation achieved as a result of applying the filter to the image gives the new position of the target in the image.

2.2. Efficient convolution operators

Various competitions and conferences are organised annually on object tracking. One of the most important of these is known as visual object tracking (VOT) challenge. In this section, the operation of the algorithm, Efficient Convolution Operators (ECO) for Tracking [5] which is the best DCF-based algorithm in the VOT-2017 [9]contest held in 2017, will be discussed in general.

The ECO algorithm tries to extract multi-resolution feature maps of images using the continuous convolution operator. As a result of these operations, the filter is constantly updated to ensure that the object contains new features in the changing image. Convolution operators here are trained with feature maps extracted from images. Let D is the number of feature channels and each image sample taken is expressed as *x_j*^1,…,*x_j*^*D*. An interpolation operator as in (1) is defined for the purpose of extracting feature maps. b_d is defined as an interpolation function which uses the shifted samples. In order to transfer each feature map *d* to the continuous spatial domain *t*∈[0,*T*), an interpolation operator *J_d*:*R*^(*N_d*) → *L*^2 (*T*) is introduced [5].

$$J_d\{x^d\}(t) = \sum_{n=0}^{N_d-1} x^d[n] b_d\left(t - \frac{T}{N_d}n\right) \tag{1}$$

The $L^2(T)$space is considered to have complex functions periodic with $T > 0$. The aim is to predict the confidence scores for each layer $S_f\{x\}(t)$ with trained the convolution filters $f = (f_1 \dots f_D)$:

$$S_f\{x\}(t) = f * J\{x\} = \sum_{d=1}^{D} f^d * J_d\{x^d\} \tag{2}$$

In order to avoid over-fitting problem and reduce the number of model parameters, the filter f in formulation (2) is replaced with an alternative filter. The reduction is carried out by modifying the convolution operator given in (2). The obtained factorised convolution operator becomes:

$$S_{Pf}\{x\}(t) = Pf * J\{x\} = \sum_{c,d} p_{d,c} f^c * J_d\{x^d\} = f * P^{\mathrm{T}} J\{x\} \tag{3}$$

P is a coefficient matrix for each of the filters of each feature layer. It is a *D*×*C* matrix where *D* is the number of feature channels and the *C* is the number of filters that have sufficient energy.

The visualisation of the energy scores of the learned filters is shown in Figure 1.

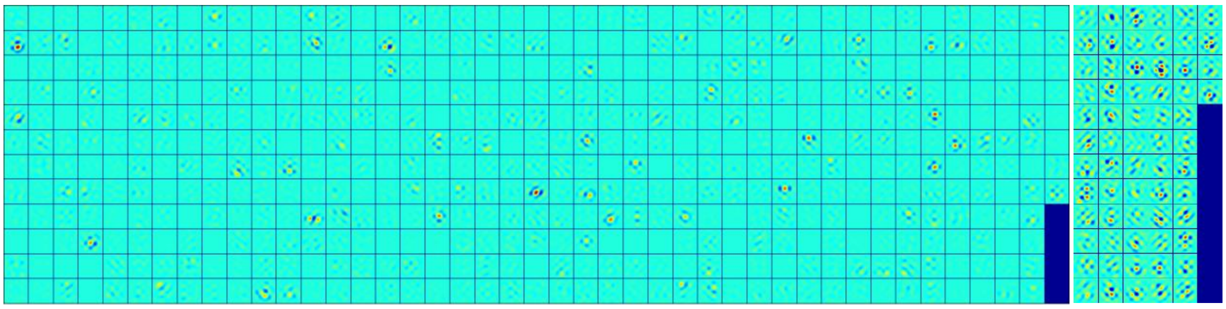


Figure 1. The visualization of energy of the learned filter. (a) All of the channels resulting from Equation (2). (b) The remaining channels resulting from Equation (3) after eliminating the channels that have negligible energy [5]

In Equation (3), *J*{*x*} is multiplied by the matrix *P^T* which results in a C-dimensional vector. This feature map is convolved with the desired filter *f*. After interpolation is carried out, the filter is trained by minimising the following expression:

$$E(f) = \sum_{j=1}^{m} \alpha_j \left\| S_f\{x_j\} - y_j \right\|^2 + \sum_{d=1}^{D} \|\omega f^d\|^2 \tag{4}$$

Figure 2 shows a visualisation of their continuous convolution operator which integrates multi-resolution deep feature maps [6]. Figure 2(a)–(d) shows the feature maps obtained from the sample image frame, shows the convolution filters learned from the feature maps, shows the strength of the convolution filters and shows the predicted location of the target after the filter is applied to the next image frame, respectively.

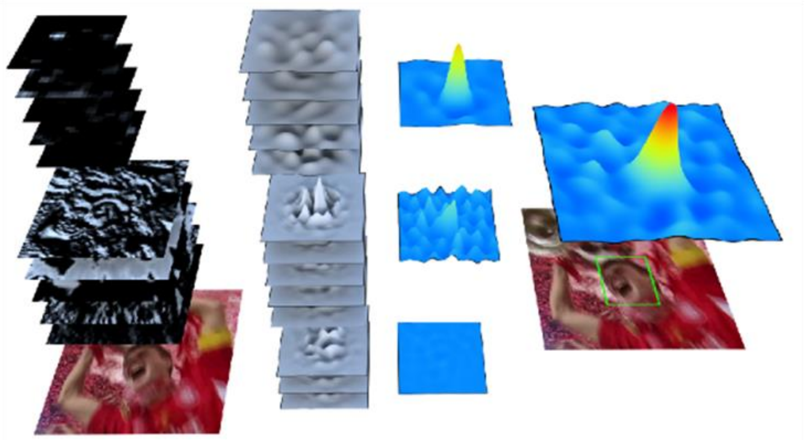


Figure 2. Visualisation of the learning architecture of filters [6]

2.3. Datasets

VOT2017 [17], VOT2019 [18], OTB-100 [16], TLP [11] and UAV123 [11] datasets were used to analyse the algorithm's performance. The datasets consist of a total of 300 sequences and more than 150,000 image frames. VOT-2019 and OTB100 datasets consist of different lengths and standard quality images. The tiny TLP dataset consists of 1,280 × 720 high resolution images. It is an abbreviated version of the original TLP dataset that is suitable for short-term object tracking purposes [11]. The UAV123 dataset contains 1,280 × 720 high resolution images, just like the tiny TLP dataset, all recorded using unmanned aerial vehicles.

2.4. Experimental work

The experiments were run on the computer which had the following specifications: Intel Core i7-7500U CPU @2.7GHz, 16GB RAM and NVIDIA GeForce 940MX with 384 CUDA cores and 2GB memory. The operating system is Ubuntu 16.04 with CUDA 8.0 installed.

The performance is measured with the Intersection over Union (IoU) scores on each dataset. The IoU score is the ratio of the intersection area of the ground-truth bounding box and the resulting bounding box generated from the tracker to the total area covered by the composition of these bounding boxes as shown in Figure 3.

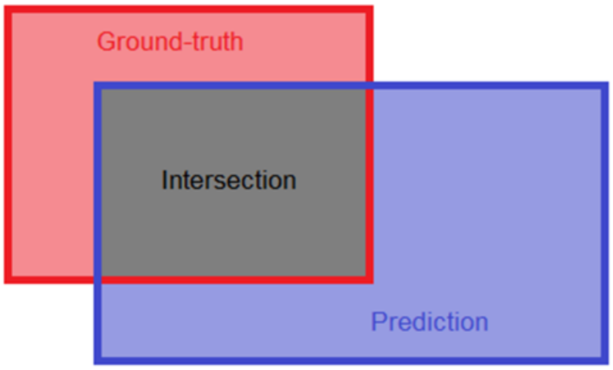


Figure 3. The visualisation of the calculation of *Intersection over Union*

The formulation of IoU can be expressed as follows:

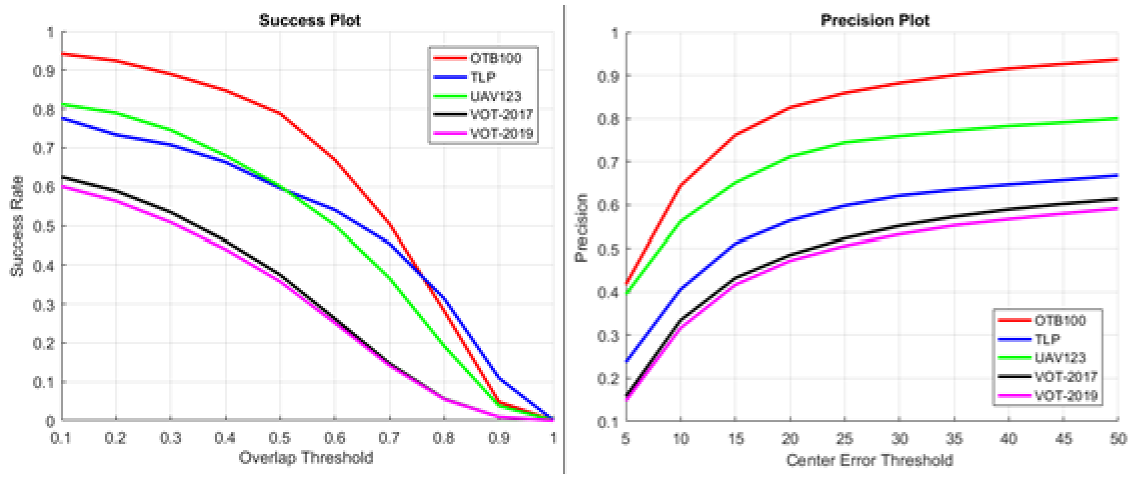
$$IoU = \frac{Intersection\ over\ Overlap}{Intersection\ over\ Union}$$

Success graph is created using IoU. If the IoU value is above a certain threshold value, the prediction is considered to be successful and the success score is increased by one. The success graph created according to the IoU threshold values ranging from 0.1 to 1 is shown in Figure 4(a). The experiments on the ECO algorithm are run without CNN support.

In addition, if the distance between the centre of the real object area and the centre of the estimated object area is above a certain threshold value in pixels, the precision score is obtained and this is shown in Figure 4(b).

When examining these results, it is necessary to consider the characteristics of the data sets. The source of motion in image sequences can occur in two ways: one is the movement of the target and the other is the movement of the image recorder. The OTB100 data set usually contains image sequences where the image recorder is stabilised and the source of the motion is the target. Although both sources are active in the image sequences in the TLP and UAV123 datasets, the motion of the image recorder is less severe. However, in the VOT-2017 and VOT-2019 datasets, the intensity of both the target's and the image recorder's movements is generally very high.

If we first examine the success plot in Figure 4(a), we see that the algorithm's performance is inversely proportional to the motion intensity of the target and the image recorder. In the precision plot in Figure 4(b), it is seen that the algorithm's performance in the TLP dataset approximates its performance in the VOT datasets. Since the resolution of the image frames in the TLP dataset is higher than in other datasets and the target sizes are generally large, the centre error thresholds remain small. This lead to the downward movement of the precision graph.



(a) **(b)**

Figure 4. The Success Plot (a) and Precision Plot (b)

3. Profiling the function calls

As previously described in 'Method', the algorithm includes a large amount of matrix products and convolution processes. An analysis was carried out to understand how much processing power these processes require and affect the speed of operation.

Valgrind's [14] tool was selected for analysis. The Valgrind's tool is a very successful tool in memory management, fault finding and processor profiling. The algorithm was run on the 3,734 image sequence with the Valgrind's analysis tool. In Figures 5 and 6, the visualisation of the analysis outputs is shown.

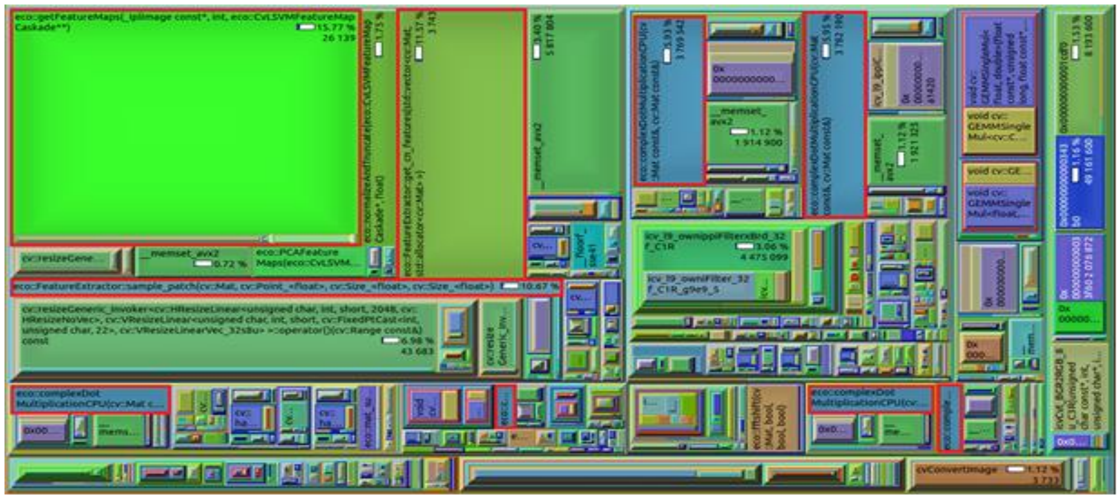

eco::getFeatureMaps(_IplImage const*, int, eco::CvLSVMFeatureMap Caskade**)
15.77 %
eco::FeatureExtractor::sample_patch(cv::Mat, cv::Point_<float>, cv::Size_<float>, cv::Size_<float>)
10.67 %
eco::complexDotMultiplicationCPU(cv::Mat const&, cv::Mat const&)
icv_l9_ownFilter_32f_C1R
cvConvertImage

Figure 5. Indication of how often algorithm components are called and how much processing time they use

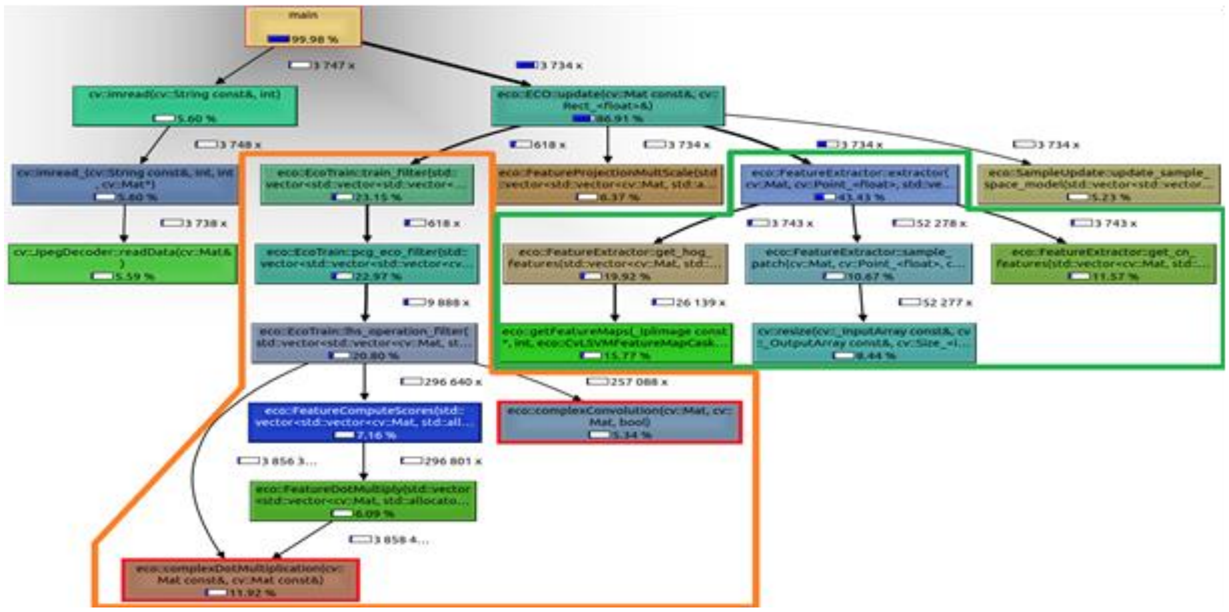

main
99.98 %
3 747 x
3 734 x
cv::imread(cv::String const&, int)
5.60 %
eco::ECO::update(cv::Mat const&, cv::Rect_<float>&)
86.91 %
3 748 x
618 x
3 734 x
3 734 x
3 734 x
cv::imread_(cv::String const&, int, int, cv::Mat*)
5.60 %
eco::EcoTrain::train_filter(std::vector<std::vector<std::vector<...
23.15 %
eco::FeatureProjectionMultScale(std::vector<std::vector<cv::Mat, std::a...
8.37 %
eco::FeatureExtractor::extractor(cv::Mat, cv::Point_<float>, std::ve...
43.43 %
eco::SampleUpdate::update_sample_space_model(std::vector<std::vector...
5.23 %
3 738 x
618 x
3 743 x
52 278 x
3 743 x
cv::JpegDecoder::readData(cv::Mat&)
5.59 %
eco::EcoTrain::pcg_eco_filter(std::vector<std::vector<std::vector<cv...
22.97 %
eco::FeatureExtractor::get_hog_features(std::vector<cv::Mat, std::...
19.92 %
eco::FeatureExtractor::sample_patch(cv::Mat, cv::Point_<float>, c...
10.67 %
eco::FeatureExtractor::get_cn_features(std::vector<cv::Mat, std::...
11.57 %
9 888 x
26 139 x
52 277 x
eco::EcoTrain::lhs_operation_filter(std::vector<std::vector<cv::Mat, st...
20.80 %
eco::getFeatureMaps(_IplImage const*, int, eco::CvLSVMFeatureMapCask...
15.77 %
cv::resize(cv::_InputArray const&, cv::_OutputArray const&, cv::Size_<i...
8.44 %
296 640 x
257 088 x
eco::FeatureComputeScores(std::vector<std::vector<cv::Mat, std::all...
7.16 %
eco::complexConvolution(cv::Mat, cv::Mat, bool)
5.34 %
3 856 3...
296 801 x
eco::FeatureDotMultiply(std::vector<std::vector<cv::Mat, std::allocato...
6.09 %
3 858 4...
eco::complexDotMultiplication(cv::Mat const&, cv::Mat const&)
11.92 %

Figure 6. Call graph showing the dependency of algorithm components on each other

Figure 5 provides a mapping of the algorithm depending on the time spent by each method with respect to the total running time. The feature extraction and the multiplications processes for filter training covers larger areas in the figure, which means these portions of the algorithm take more time than the other methods.

The algorithm proceeds predominantly in two independent branches as can be seen in Figure 6. One is the part about the training of the filter and the other is the part where the feature maps of the image sequences are constructed. Approximately 74% of the time spent in the filter training section, which is enclosed by orange lines in Figure 6, covers matrix multiplications and convolution. It is observed that approximately 30% of the entire running time of the algorithm consists of extracting HOG and Color Names property maps which is enclosed by green lines in Figure 6. FFT and matrix manipulations are frequently used during these operations. As it is known, these processes can be good candidates for GPU parallelisation.

4. GPU parallelisation of element-wise matrix multiplication and results

In this study, we focus on the parallelisation of the methods which have complex matrix element-wise multiplication processes in their basis. The number of calls of this method for each frame changes between 300 and 400 times, depending on the image size and the scaling factor. The matrices have two channels, which means they have complex elements, and the sizes vary between 50 × 50 and 100 × 100.

The initialisation and training of the filter, the extraction of the feature from the images, the convolution of the filter and the features, and the computation of the energy include matrix multiplications and all using a similar method whose algorithm is given in Table 1.

Table 1. Pseudo-code of element-wise multiplication of two *k*×*n* matrices

```
1.    procedure complexDotMultiplication(A, B)
2.            $k = A.rows$
3.            $n = A.cols$
4.    let C be a new k × n matrix
5.    for i = 1 :k
```

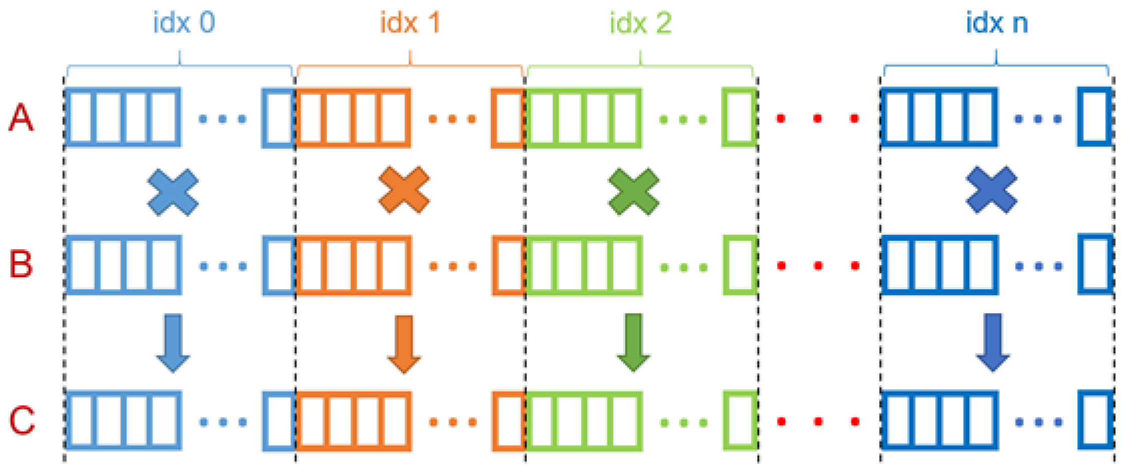
```
6.    for j = 1 :n
7.          $C_{ij} = A_{ij} \cdot B_{ij}$
8.    end for
9.    end for
10.   return C
11.   end procedure
```

These types of arrays can be manipulated in different ways on the GPU. In this study, a straightforward approach is followed. The array elements are distributed evenly between 1,000 threads. The method followed on the distribution of the array elements is shown in Figure 7.

The pseudo-code for the GPU implementation of the methods is given in Table 2. This approach only uses the advantage of a large number of cores on GPUs to shorten the computation time. In order to achieve faster memory transfers between CPU and GPU, the Pinned Host Memory is used for the allocation of memory for the matrices. The host (CPU) memory allocation is pageable by default and it is managed by the operating system. While transferring data on the pageable host memory to GPU memory, the CUDA driver [4] first copies the data to a temporarily allocated pinned memory, and then the transfer to the device memory is carried out. By initially storing the matrices in a pinned memory, the unnecessary memory copy between the pageable and pinned memory is removed.

Table 2. Pseudo-code of element-wise multiplication of two *k*×*n* matrices

```
procedure complex Dot Multiplication (A, B)
$k = A.rows$
$n = A.cols$
let C be a new k x n matrix
for i = 1 :k
for j = 1 :n
$C_{ij} = A_{ij} \cdot B_{ij}$
end for
end for
return C
end procedure
```



The implementation is carried out by using the CUDA C++ with respect to the CUDA Programming Guide [3] Open CV libraries are used for CPU measurements and CUDA's built-in functions are used for GPU measurements. The results of the measurements on the total time spent on the methods are given in Table 3. The GPU calculations resulted in a 57% faster performance than CPUs. Even though the calculations are much faster in GPU, they take a very small portion of time even in CPUs. The main drawback in GPU usage is the data transfers between CPU and GPU. The time spent on memory transactions is minimised with the use of the Pinned Host Memory allocation [3].

Table 3. The total time spent on the element-wise multiplication of the complex matrices

CPU time (in second)	GPU time (in second)	Speed-up
3,205.17	1,371.53	~2.34

5. Conclusion and future work

Thanks to the increasing interest in object tracking in recent years and the increasing use of object tracking in daily applications, the reliability and accuracy of algorithms are being developed every year. However, due to the increased image processing requirements, their speed is decreasing.

In this study, we have obtained the benchmark results of ECO tracking algorithm on five datasets with different characteristics. An analysis was run on the algorithm and an investigation was carried out on the suitability of the algorithm for GPU parallelism with the help of a profiling tool.

After the determination of candidate methods, we implemented a GPU code from a very naive approach. We benefited from NVIDIA’s CUDA C/C++ extension for the implementation.

We have presented a GPU implementation of some methods in the ECO tracking algorithm and have shown that it can be accelerated by 57% with respect to its CPU implementation. In this work, it is proved that the high computing power of the GPUs can be used for large data manipulations like matrix multiplications. This speed-up can be beneficial if the aim is to meet real-time requirements of object tracking algorithms.

In the future, we also expect to parallelise the convolution operators, Fourier transforms and other matrix manipulations in the ECO algorithm, and to test them on NVIDIA Jetson TX2 and Tesla K40 platforms with further memory optimisations.

References

[1] Bhat, G., Johnander, J., Danelljan, M., Shahbaz Khan, F. & Felsberg, M. (2018). *Unveiling the power of deep tracking* (pp. 483–498). Proceedings of the European conference on computer vision (ECCV). doi:10.1007/978-3-030-01216-8_30

[2] Bolme, D. S., Beveridge, J. R., Draper, B. A. & Lui, Y. M. (2010). *Visual object tracking using adaptive correlation filters* (pp. 2544–2550). 2010 IEEE computer society conference on computer vision and pattern recognition. Piscataway, NJ: IEEE. doi:10.1109/CVPR.2010.5539960

[3] CUDA C++ Programming Guide. *NVIDIA corporation*. Retrieved from https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[4] CUDA Toolkit Documentation v8.0. Retrieved from https://docs.nvidia.com/cuda/archive/8.0/index.html

[5] Danelljan, M., Bhat, G., Shahbaz Khan, F. & Felsberg, M. (2017). *Eco: efficient convolution operators for tracking* (pp. 6638–6646). Proceedings of the IEEE conference on computer vision and pattern recognition. doi:10.1109/CVPR.2017.733

[6] Danelljan, M., Robinson, A., Khan, F. S. & Felsberg, M. (2016). *Beyond correlation filters: learning continuous convolution operators for visual tracking* (pp. 472–488). European conference on computer vision. Cham, Switzerland: Springer. doi:10.1007/978-3-319-46454-1_29

[7] Feng, X., Jiang, Y., Yang, X., Du, M. & Li, X. (2019). Computer vision algorithms and hardware implementations: a survey. *Integration, 69*, 309–320. doi:10.1016/j.vlsi.2019.07.005

[8] Hester, C. F. & Casasent, D. (1980). Multivariant technique for multiclass pattern recognition. *Applied Optics, 19*(11), 1758–1761. doi:10.1364/AO.19.001758

[9] Kristan, M., Leonardis, A., Matas, J., Felsberg, M., Pflugfelder, R., CehovinZajc, L., …Fernandez, G. (2017). *The visual object tracking vot2017 challenge results* (pp. 1949–1972). Proceedings of the IEEE international conference on computer vision workshops. doi: 10.1109/ICCVW.2017.230

[10] Kristan, M., Matas, J., Leonardis, A., Felsberg, M., Pflugfelder, R., Kamarainen, J. K., …Eldesokey, A. (2019). *The seventh visual object tracking vot2019 challenge results*. Proceedings of the IEEE international conference on computer vision workshops. doi: 10.1109/ICCVW.2019.00276

[11] Moudgil, A. & Gandhi, V. (2018). *Long-term visual object tracking benchmark* (pp. 629–645). Asian conference on computer vision. Cham, Switzerland: Springer. doi:10.1007/978-3-030-20890-5_40

[12] Mueller, M., Smith, N. & Ghanem, B. (2016). *A benchmark and simulator for uav tracking* (pp. 445–461). European conference on computer vision. Cham, Switzerland: Springer. doi:10.1007/978-3-319-46448-0_27

[13] Sun, C., Wang, D., Lu, H. & Yang, M. H. (2018). *Correlation tracking via joint discrimination and reliability learning* (pp. 489–497). Proceedings of the IEEE conference on computer vision and pattern recognition. doi:10.1109/CVPR.2018.00058

[14] Valgrind. (2020, January). Retrieved from https://valgrind.org/

[15] Visual Tracker Benchmark. Retrieved from http://www.visual-tracking.net

[16] Visual Tracker Benchmark Dataset. Retrieved from http://cvlab.hanyang.ac.kr/tracker_benchmark/datasets.html

[17] VOT2017 Challenge Dataset. Retrieved from https://www.votchallenge.net/vot2017/dataset.html

[18] VOT2019 Challenge Dataset. Retrieved from http://www.votchallenge.net/vot2019/dataset.html

[19] Xu, T., Feng, Z. H., Wu, X. J. & Kittler, J. (2019). Learning adaptive discriminative correlation filters via temporal consistency preserving spatial feature selection for robust visual object tracking. *IEEE Transactions on Image Processing,28*(11), 5596–5609. doi:10.1109/TIP.2019.2919201

[20] Yao, R., Lin, G., Xia, S., Zhao, J. & Zhou, Y. (2019). Video object segmentation and tracking: a survey. doi:10.1145/
3391743

[21] Zhang, Q., Yang, L. T., Chen, Z. & Li, P. (2018). A survey on deep learning for big data. *Information Fusion, 42*, 146–157. doi:10.1016/j.inffus.2017.10.006

[22] Zou, Z., Shi, Z., Guo, Y.& Ye, J. (2019). Object detection in 20 years: asurvey. *ArXiv,* abs/1905.05055.