Multi–GPU Implementation of Machine Learning Algorithm using CUDA and OpenCL

Jan Masek, Radim Burget, Lukas Povoda and Malay Kishore Dutta

***Abstract*—Using modern Graphic Processing Units (GPUs) becomes very useful for computing complex and time consuming processes. GPUs provide high–performance computation capabilities with a good price. This paper deals with a multi–GPU OpenCL and CUDA implementations of *k*–Nearest Neighbor (*k*–NN) algorithm. This work compares performances of OpenCL and CUDA implementations where each of them is suitable for different number of used attributes. The proposed CUDA algorithm achieves acceleration up to 880x in comparison with a single thread CPU version. The common *k*-NN was modified to be faster when the lower number of *k* neighbors is set. The performance of algorithm was verified with two GPUs dual-core NVIDIA GeForce GTX 690 and CPU Intel Core i7 3770 with 4.1 GHz frequency. The results of speed up were measured for one GPU, two GPUs, three and four GPUs. We performed several tests with data sets containing up to 4 million elements with various number of attributes.**

***Keywords*—Artificial intelligence, big data, comparison, CUDA, GPU, high performance computing, *k*-NN, multi–GPU, OpenCL.**

I. Introduction

Parallel computing is a way how to accelerate many algorithms, which are computationally intensive. These algorithms can be found in image, sound and video applications or simulations, data mining, security [1], forecasting systems, etc.

k-NN belongs to the algorithms of artificial intelligence and it is one of the most widely used algorithms in data mining applications. Algorithm can be used for the classification of many various problems from business or science. Sometimes there is a requirement to process large datasets with high dimensional data. These problems can take days to compute. Using parallel computing, these problems can be solved faster than using non–parallel implementation. GPUs have much more cores than CPU, so they can be used as better solution for parallelization. The next advantage to use GPUs is relatively low price due to their high performance. *k*-NN algorithm is a good candidate for GPU parallelization.

Manuscript received on November 18, 2015, revised June 9, 2016.

This research work was supported by the project of the Technology Agency of the Czech Republic - project EPSILON no. TH01010277.

J. Masek is with BurgSys, a.s., Hnevkovskeho 30/65, 617 00 Brno, Czech Republic, phone: +420728947018; e-mail: masek@burgsys.com

R. Burget is with the Brno University of Technology, Faculty of Electrical engineering, Department of Telecommunications, Czech Republic, e-mail: burgetrm@feec.vutbr.cz

L. Povoda is with the Brno University of Technology, Faculty of Electrical engineering, Department of Telecommunications, Czech Republic, phone: 541146962; e-mail: xpovod00@stud.feec.vutbr.cz

M. K. Dutta is with Amity University, Department of Electronics & Communication Engineering, Sector-125, 201 301 Noida, India (phone: 0120-2445252 / 4713600; e-mail: mkdutta@amity.edu

In this paper an OpenCL [2] and CUDA [3] accelerated version of *k*–Nearest Neighbor machine learning algorithm has been introduced. This work is based on our previous work [4]. The algorithm is very computationally intensive mainly when big datasets with high dimensional data have to be processed. The process can take hours or days. To solve this problem, we modified common *k*–NN algorithm to run on multiple GPUs. We used two common gaming dual–core graphic cards NVIDIA GeForce GTX 690 [5] with 2x3072 CUDA cores in total. The theoretical single precision computing performance is 11.24 TFLOPS for both devices. We also used these GPUs to speed up Viola–Jones object detector [6], which was also used in [7] [8] [9].

The main contribution of this paper is the creation of the OpenCL and CUDA versions of *k*–NN algorithm, which can be executed on several GPU cards in parallel. Using this relatively cheap hardware, it is able to speed up computation up to 880 times in comparison with CPU with 4.1 GHz frequency. A newly created algorithms were tested on dataset containing millions of elements with various number of attributes (4, 10, 100 and 1000 attributes) and then algorithms were together compared.

The rest of the paper is organized as follows: Section II describes other GPU implementations of *k*–NN algorithm. Section III describes *k*–NN algorithm. OpenCL and CUDA platforms are introduced in section IV. In section V our GPU implementation is described. Results and discussion are described in section VI. Section VII concludes this paper.

II. Related work

GPU computing has become very popular during last several years. There is also increasing need to process more amount of data with artificial intelligence. The next paragraph describes several articles dealing with CUDA implementations of *k*-NN GPU algorithm and various use cases of the algorithm.

In [10] a new brute force algorithm for building the k-Nearest Neighbor Graph is described. The proposed algorithm has two parts, where the first is for finding distances between the input vectors and the second part is for selection of *k* neighbors for each testing sample. Also new algorithm based on quick sort was implemented for quicker sorting of distance pairs. The algorithm achieves higher speed up, if the *k* variable is increasing.

The paper [11] compares GPU implementation of brute force *k*-NN with several CPU based implementations and the implementation of algorithms from ANN[1] library (A Library

[1]Available from URL: http://www.cs.umd.edu/~mount/ANN/

for Approximate Nearest Neighbor Searching).

In [12] [13] several optimization techniques were applied to maximize the utilization of the GPU.

The work [14] describes MST (minimum spanning tree) problem, which is resolved by the combination of classical Boruvka MST algorithm and the *k*-NN graph structure. Achieved speed-ups were between 30 and 40 in comparison with CPU implementation.

In [15] authors describe how to use GPU *k*-NN algorithm for image processing (texture analysis). Their algorithm is 150 times faster than CPU version during processing synthetic data and up to 75 times faster during processing image data.

In [16], the LSH (Locality Sensitive Hashing) algorithm was used for *k*-NN computation. The results were demonstrated on large image datasets and achieved acceleration was 40 in comparison with CPU version.

Several comparison tests between OpenCL and CUDA frameworks were performed. In [17] authors performed 16 benchmark tests where CUDA achieves for about 30 % better performance than OpenCL. Further they tried OpenCL portability and they did not found differences in performance. In [18] authors compared executive time of CUDA Drive API with OpenCL platform where CUDA was for about 5 % faster than OpenCL.

The implementation of GPU *k*-NN algorithm into RapidMiner[2] data mining platform was described in [19]. The algorithm was created in JAVA programming language with using jcuda[3] library that is responsible for executing CUDA kernel from JAVA. Created algorithm achieves 170x speedup, but it depends mainly on number of used attributes (using more than 128 attributes decreases the speedup).

Our approach differs from using OpenCL platform instead of CUDA and our algorithm has several improvements in comparison with some approaches described in this paragraph. The main improvement is an option to run our algorithm on multiple GPUs in parallel. The created OpenCL kernel was partially vectorized and the algorithm was created without need to have some sorting algorithms. These improvements speed up the algorithm. Our solution was tested on very large data set, where the processing time was minutes against other works, where processing quite small datasets took seconds.

III. *k*-Nearest Neighbor algorithm

k–NN algorithm can be used for classification or regression. The principle of *k*–NN is shown in Fig. 1. The input of algorithm are training examples and testing examples. For each testing example, the distance (Euclidean, Manhattan, etc.) of attributes between testing and training example is computed. The distances are computed for the one testing example and all training examples. Then the distances are sorted according to their values. The training examples with *k* lowest differences are selected as the nearest neighbors. According to their classification classes, the testing example is classified. Usually the lower number of *k* value is set.

[2]Available from URL: `http://rapidminer.com`
[3]Available from URL: `http://www.jcuda.org/`

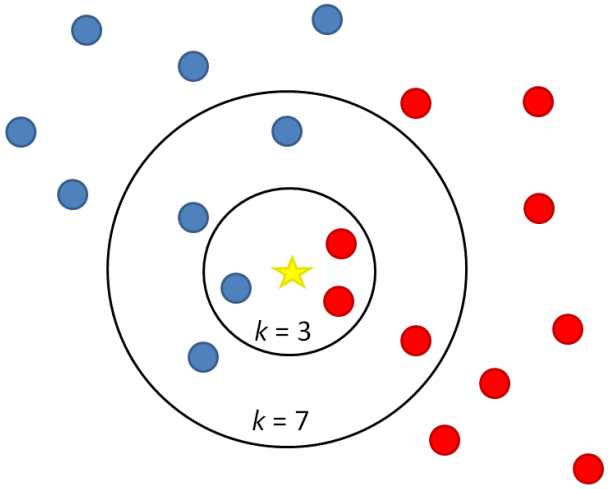


Fig. 1. The principle of *k*–NN algorithm.

IV. OpenCL and CUDA introduction

Nowadays there exist two platforms for GPU computing that are well used by many users. The first developed platform is CUDA [3] and the second is OpenCL [2]. CUDA is being more used but on the other hand CUDA can only be used with NVIDIA GPUs. OpenCL is being used less than CUDA but OpenCL can be performed on many various devices.

When compared these GPU platforms with common CPU solution, GPU hardware is much more specialized for intensive highly parallel computing. It can be seen from Fig. 2 and Fig. 3 where ALU (Arithmetic Logic Unit) elements are used for computing. The GPU hardware can process much more computing units in parallel than CPU.

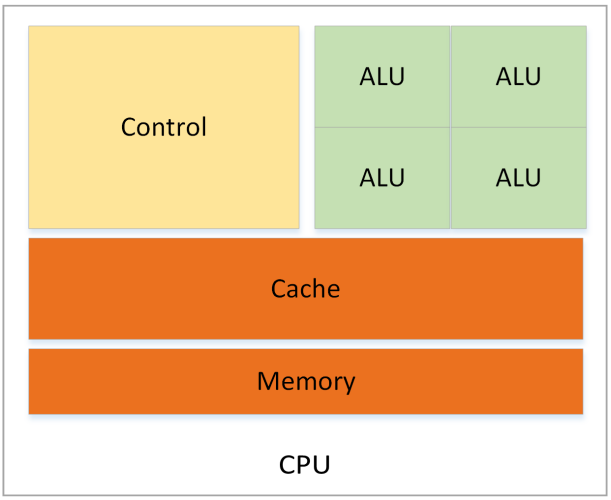


Fig. 2. Scheme of CPU

A. *OpenCL*

OpenCL (Open Computing Language) [2] is an open royalty-free standard determined for parallel programming of suitable devices like CPUs, GPUs and the other devices. OpenCL can solve many problems more efficiently than CPU.

In these days, the GPU computing is very popular and many applications have been developed in OpenCL.

There are two types of OpenCL code. The first type executed on CPU is called host part and the second one that is executed on OpenCL device is called device part. OpenCL kernel is executed in a device. The kernel can contain optimized code with OpenCL functions. OpenCL devices use SIMT (Single Instruction Multiple Threads) architecture. OpenCL device consists of Streaming Multiprocessors (SMs) where each of them contains many simple cores. These cores are able to do only simple operations, so OpenCL programming is more complex. Cores can execute many work-items (threads) in parallel. Work-items are grouped into work-group and they can mutually communicate and use the same (shared) local memory. The number of work-groups and work-items has to be set on the start of the process. OpenCL device contains on–chip and off–chip memories. On–chip (private, local) memories are faster than off-chip (global memory, constant memory, texture cache). However some of these memories can be fast too, because they are cached. [20]

B. CUDA

CUDA [21] is a parallel computing and programming platform and only newer NVIDIA graphic cards are supported by CUDA. Nowadays, there exist many GPU computing applications developed in CUDA for example deep learning algorithm [22] that is used for training neural network for image recognition.

Common CUDA GPU uses same principles that are described in OpenCL section. There are only different names of therms (shown in Table I). In CUDA CPU is marked as a host and GPU is marked as a device.

V. Optimization k-NN for GPUs

Firstly, we tried to process large data sets in RapidMiner, but unfortunately the original CPU version of k-NN was too slow. So we decided to create GPU accelerated algorithm that can be executed from RapidMiner. Our implementation was created in JAVA programming language, because RapidMiner is also programmed in JAVA. The first step was to create OpenCL kernel that was created in C programming language with using OpenCL syntaxes. For mutual cooperation between OpenCL and JAVA, jocl[4] library was used [21]. According to OpenCL kernel we created CUDA version of this kernel using CUDA version 7.5 and jcuda[5] library that was used as JAVA wrapper.

Training and testing data sets have to be transformed into float arrays before they are transmitted to GPU. We used OpenCL vector format called float4 that has a big advantage: it contains four float values that are processed in one step instead of four steps (for common float). So every training and testing example is saved into float4 array. We also optimized kernel with using local memory.

In our implementation, the classical principle of k-NN was a little bit modified. The differences are mutually compared during their computation and the lowest k differences are saved as final nearest neighbors. The algorithm 1 shows the principle of modified algorithm. After this modification, the algorithm can work faster for lower k values. When compared with CPU version, the classification results were the same.

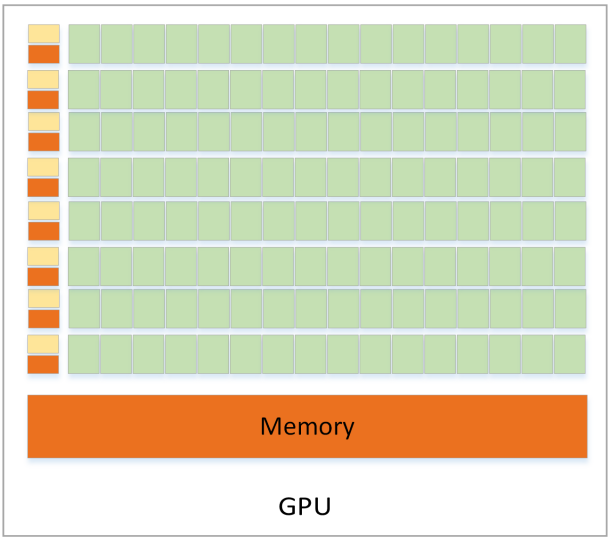


Fig. 3. Scheme of GPU

A. Multi–GPU support

For multi–GPU support we created a JAVA library that is able to utilize all found GPUs. This library is available only for OpenCL. The library can automatically split input and output data and transmit them equally into all devices. It decreases amount of transmitted data. The next advantage is a very easy way, how to write code in JAVA with minimum knowledge of OpenCL. For multi–GPU support for CUDA platform we had to run split data into GPUs and start computing on each GPU in separated JAVA thread in parallel.

In case of k-NN algorithm, the training data vector had to be transmitted into all GPU devices. Testing data vector and vector with final predictions were splitted equally into all device due to lower load of GPU memory. We also tried the version of algorithm, where data were not splitted into GPU devices, but they were copied whole to each device. Differences between computing times of each version were negligible.

[4]Available from URL: `http://www.jocl.org/`
[5]Available from URL: `http://www.jcuda.org/`

TABLE I
CUDA and OpenCL terminology mapping

CUDA	OpenCL
Grid	NDRange
Thread Block	Work–group
Thread	Work item
Thread ID	Global ID
Block index	Block ID
Thread index	Local ID
Shared Memory	Local Memory
Local Memory	Private Memory

Input: training data (float4), testing data (float4)

Output: prediction vector

loading test sample from testing data;

foreach *train sample* **do**
  foreach *attribute* **do**
    compute distance between train and test samples;
    sum distances;
  end
  for $k = 0$ **to** *number of neighbors* **do**
    if *sum of distances* $<$ *distance for k neighbor* **then**
      distance for k neighbor = sum of distances;
      shift distance values for other k distances;
      break;
    end
  end
end
counting number of neighbors for each label;
selecting label with highest number of neighbors;
setting prediction for test sample to prediction vector;

Algorithm 1: k-NN OpenCL algorithm

VI. Results and discussion

We performed several comparison tests to verify the functionality of our accelerated k-NN algorithms. The tests were performed with data sets containing different amount of elements and different numbers of attributes. Since the algorithm has been modified to have a good result for lower k parameter ($k = 5$), we also carried out several tests for higher values of k parameter ($k = 10, k = 20$). For a comparison between CPU and GPU versions, we used RapidMiner platform that consists of many machine learning and data–mining algorithms. First, we generated polynomial data set using one of the RapidMiner algorithms. Then this data set was divided into two parts. The training part contained 25 % of elements and testing part contained 75 % of elements. In the next step, several tests with different number of elements and different number of attributes were performed. We used CPU version of k-NN algorithm integrated in RapidMiner and our GPU versions of k-NN that were also executed in RapidMiner.

TABLE II
OpenCL - comparing for different k neighbors.

	$k = 1$	$k = 10$	$k = 20$
1 million, 10 attributes.	14.1 s	25.2 s	93.7 s
1 million, 4 attributes.	4.1 s	17.8 s	88 s

TABLE III
CUDA -comparing for different k neighbors.

	$k = 1$	$k = 10$	$k = 20$
1 million, 10 attributes.	16 s	21.7 s	88.5 s
1 million, 4 attributes.	3.8 s	16.2 s	78.8 s

Our measurements were performed in computer with CPU Core i7-3770 4.1 GHz (in boosted mode), 32 GB RAM and two dual-core NVIDIA GeForce GTX 690 [5] graphic cards that are very powerful in single precision mode. Every GTX 690 consists of two GPU cores and every GPU core has 8 Streaming Multiprocessors and each SM consists of 192 CUDA cores (1536 CUDA cores in total). The size of a GPU memory is 4096 MB with 6 GHz frequency. GTX 690 has theoretical performance 5.62 TFLOPS in single precision mode. When both GPU cards are used, the performance of system is 11.24 TFLOPS. The measured power consumption of one GPU GTX 690 was 300 watts.

The tests were performed with using one core of CPU, one GPU, 2 GPUs, 3 GPUs and 4 GPUs. The results show how much time each scenario took and they are described in the Table IV for OpenCL implementation and in Table V for CUDA implementation. In this case the measurements were performed for $k = 5$. As we can see from table some CPU computations can take days in comparison with GPU computation, where it takes minutes. Fig. 4 shows speed up of our OpenCL GPU implementation of k–NN algorithm. We can see that increasing amount of attributes can decrease speed up. Speed up can be also increased if higher number of elements is used. Scenarios for CUDA implementation are shown in Table V. The overall speed up of CUDA implementation of k–NN algorithm is shown in Fig. 5. The best speed up was achieved in scenario with 1 million of elements and 4 attributes where achieved speed up was 882 times. The comparison between CUDA and OpenCL implementations is shown in Fig 6. We can see that for scenarios with number of attributes 100 and 1000, CUDA was for about 3 % faster than OpenCL. For scenarios with 10 attributes OpenCL implementation was faster for about 11 %. And for 4 attributes CUDA was for about 18 % faster than OpenCL. These differences in speed up when 4 or 10 attributes are used, can be caused with using float4 data type for storing array of attributes where CUDA can handle much more better with 4 attributes in one float4 array than with 10 attributes in 3 float4 arrays where two elements in array are not used. When computing the average value of all scenarios, CUDA was for about 0.5 % faster.

The table II shows the results for using different values of k (measured for all GPUs). Our OpenCL implementation has been created to work effectively with the number of k neighbors lower than 10. Otherwise, the speed up of algorithm will be radically decreased. In comparison with CUDA implementation (see Table. III) we can see that CUDA is slightly faster than OpenCL implementation.

VII. Conclusion

The main contribution of this work is OpenCL accelerated implementations of k-Nearest Neighbor machine learning algorithm with using OpenCL and CUDA. The algorithm can be executed on multiple GPUs in parallel. We created the modified version of algorithm that achieves very good results for k neighbors lower than 10. We found that with using relatively cheap hardware (2x NVIDIA GeForce GTX 690), it is possible to compute 4 million elements (each has 10 attributes) in 3 minutes in comparison with using one single core CPU

TABLE IV
RESULTS OF COMPUTATION FOR OPENCL IMPLEMENTATION - $k = 5$.

	1 CPU	1 GPU	2 GPUs	3 GPUs	4 GPUs
0.4 million, 1000 attributes	17h 35min	843 s	434 s	295 s	228 s
2 million, 100 attributes	2d 4h 10min	2106 s	1056 s	707 s	532 s
1 million, 10 attributes	1h 55min	40.2 s	21.7 s	15.2 s	11.5 s
4 million, 10 attributes	1d 7h 29min	645 s	332 s	223 s	169 s
1 million, 4 attributes	1h 12min	21 s	11.1 s	7.5 s	5.8 s

CPU - Intel Core i7 3770@4.1GHz, L3 cache - 8192kB
4 GPU - 2x3072 cores, mem. 2x4096MB@6 GHz, GPU - 1019 Mhz

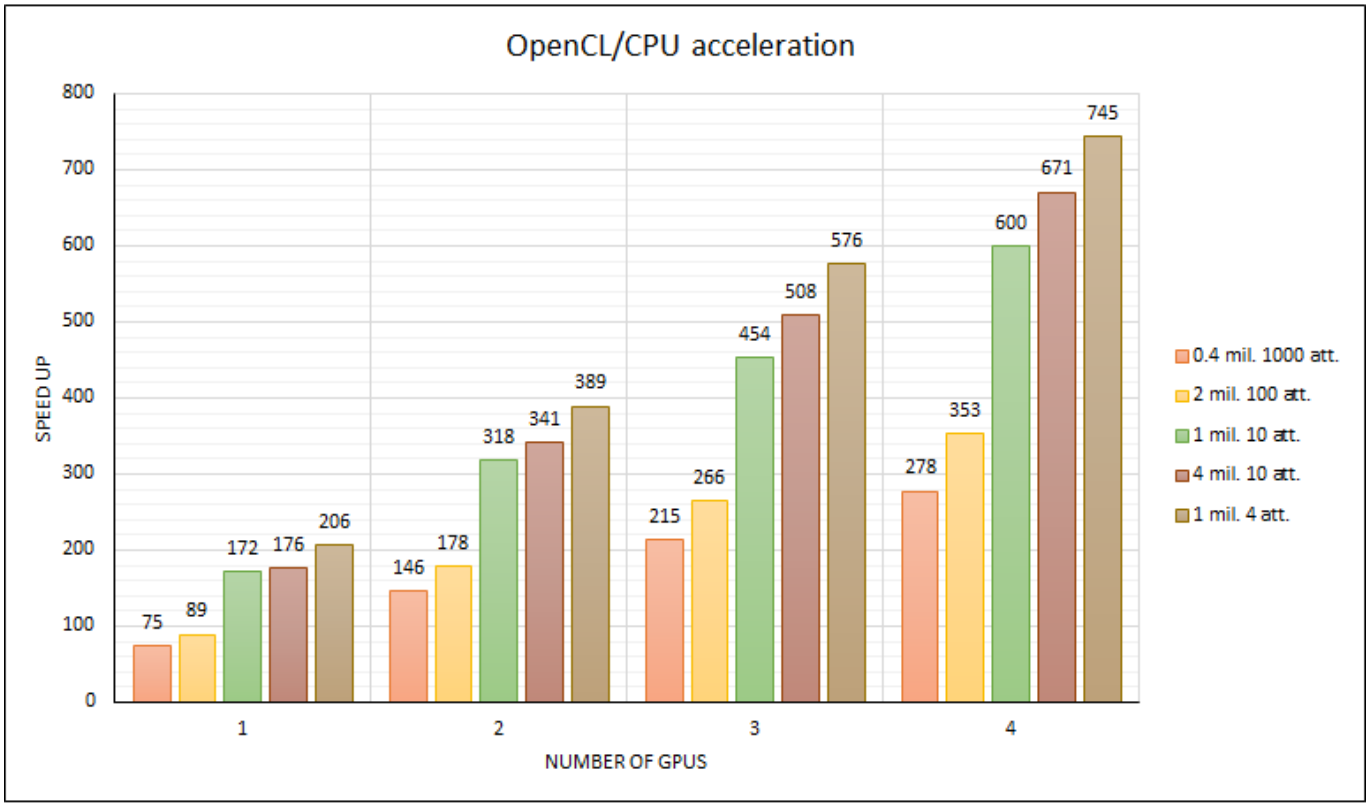


Fig. 4. Acceleration for OpenCL version of k-NN algorithm.

TABLE V
RESULTS OF COMPUTATION FOR CUDA IMPLEMENTATION- $k = 5$.

	1 CPU	1 GPU	2 GPUs	3 GPUs	4 GPUs
0.4 million, 1000 attributes	17h 35min	818 s	419 s	286 s	218 s
2 million, 100 attributes	2d 4h 10min	2038 s	1023 s	683 s	523 s
1 million, 10 attributes	1h 55min	47.3 s	23.9 s	15.9 s	12.6 s
4 million, 10 attributes	1d 7h 29min	757 s	377 s	252 s	196 s
1 million, 4 attributes	1h 12min	18 s	9.3 s	6.4 s	4.9 s

CPU - Intel Core i7 3770@4.1GHz, L3 cache - 8192kB
4 GPU - 2x3072 cores, mem. 2x4096MB@6 GHz, GPU - 1019 Mhz

(Intel Core i7–3770, 4.1GHz), where the computation took over 31 hours. The best achieved acceleration was up to 880x. Our algorithms were created in JAVA programming language and they have been implemented in to the RapidMiner data mining platform. The most time consuming part of algorithm has been created in OpenCL and CUDA. For mutual cooperation between JAVA and OpenCL or CUDA, the jocl and jcuda libraries were used. When compared OpenCL and CUDA implementations, CUDA has better results for data set containing 4 attributes or with 100 or 1000 attributes, but

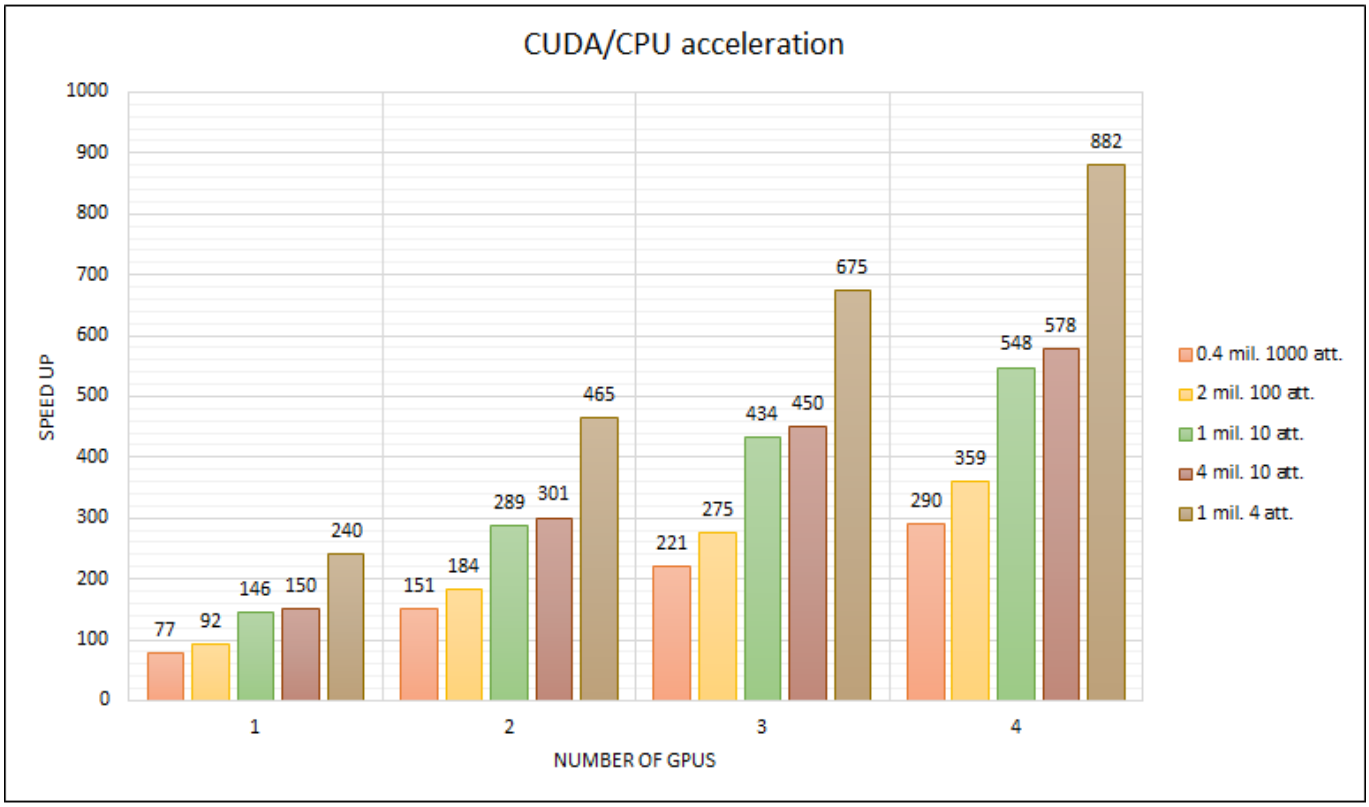


Fig. 5. Acceleration for CUDA version of *k*-NN algorithm.

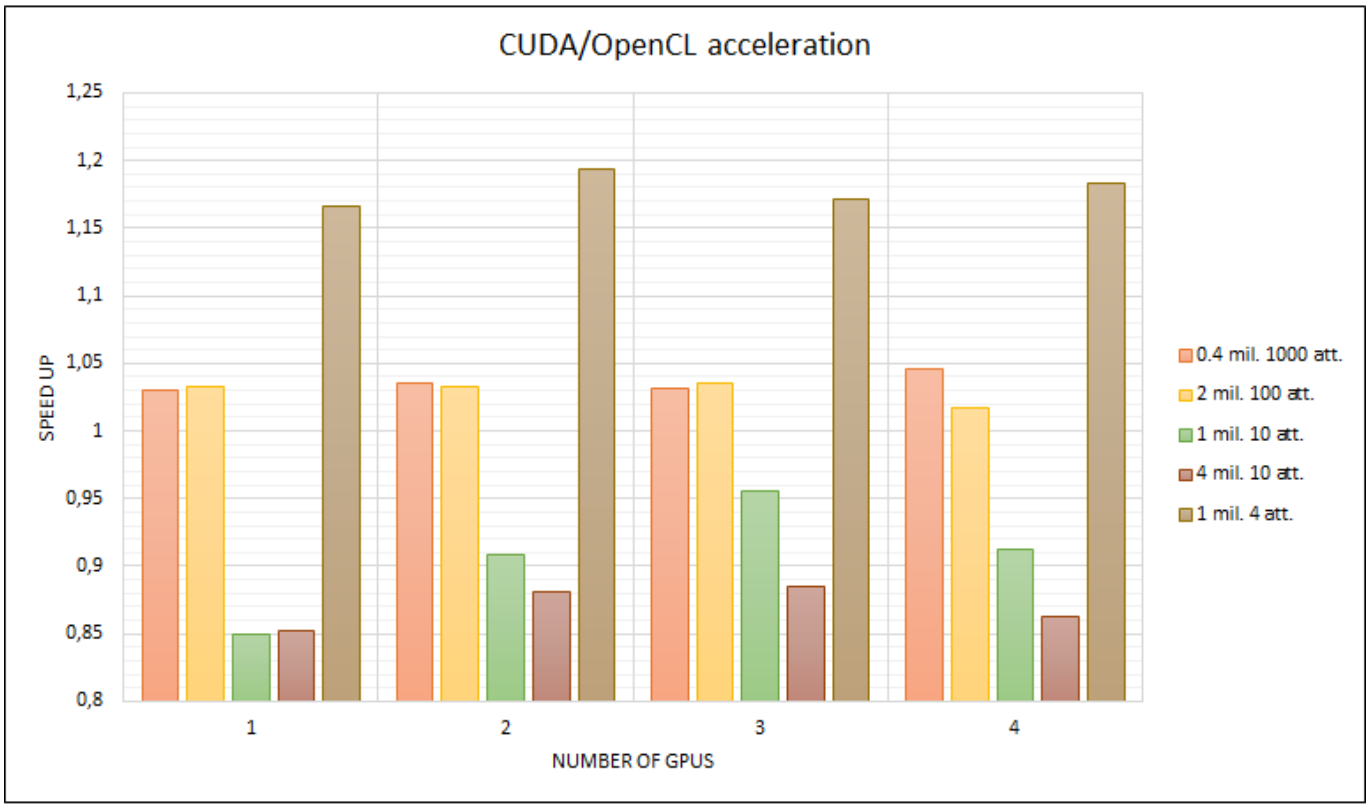


Fig. 6. Speed up comparison between CUDA and OpenCL implementations of *k*-NN algorithm.

on the other hand OpenCL has better result with using data set with 10 attributes. When compared overall results, both OpenCL and CUDA achieves similar speed up.

REFERENCES

[1] J. Minar, K. Riha, H. Tong, "Intruder Detection for Automated Access Control Systems with Kinect Device,"*In 2013 36th International Conference on Telecommunications and Signal Processing (TSP),* 2013, pp. 826-829, ISBN 978-1-4799-0403-7.

[2] Khronos OpenCL Working Group,"The OpenCL Specification - Version: 1.1,"2011, Available: `http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf`

[3] NVIDIA Inc,"CUDA Toolkit 7.5,"2015, Available: `https://developer.nvidia.com/about-cuda`

[4] J. Masek, R. Burget, J. Karasek, V. Uher, M.K. Dutta, "Multi-GPU implementation of k-nearest neighbor algorithm,"*In 2013 36th International Conference on Telecommunications and Signal Processing (TSP),* 2015, pp. 764-767, ISBN 978-1-4799-8497-8.

[5] NVIDIA, 2013, February 5. "GeForce Hardware."Available: `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-690`

[6] J. Masek, R. Burget, V. Uher, S. Guney,"Speeding up Viola–Jones algorithm using multi–Core GPU implementation,"*In 2013 36th International Conference on Telecommunications and Signal Processing (TSP),* 2013, pp. 808-812, ISBN: 978-1-4799-0402-0.

[7] K. Riha, J. Masek, R. Burget, R. Benes, E. Zavodna, "Novel method for localization of common carotid artery transverse section in ultrasound images using modified Viola–Jones detector,"*Ultrasound in Medicine & Biology,* 2013, pp. 1887-1902, ISSN 0301-5629.

[8] R. Burget, P. Cika, M. Zukal, J. Masek,"Automated Localization of Temporomandibular Joint Disc in MRI Images,"*In 2011 34th International Conference on Telecommunications and Signal Processing (TSP),* 2011, pp. 413-416, ISBN: 978-1-4577-1409-2.

[9] J. Masek, R. Burget, J. Karasek, V. Uher, S. Guney,"Evolutionary Improved Object Detector for Ultrasound Images,"*In 2013 36th International Conference on Telecommunications and Signal Processing (TSP),* 2013, pp. 586-590, ISBN: 978-1-4799-0402-0.

[10] I. Komarov, A. Dashti, R. D Souza,"Fast *k*-NNG construction with GPU-based quick multi-select,"2013.

[11] V. Garcia, E. Debreuve, M. Barlaud,"Fast k Nearest Neighbor Search using GPU,"*In IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops,* 2008, pp. 1-6.

[12] S. Liang, Y. Liu, Ch. Wang, L. Jian,"A CUDA-based Parallel Implementation of K-Nearest Neighbor Algorithm,"*In Cyber-Enabled Distributed Computing and Knowledge Discovery,* 2009, pp. 291-296.

[13] Q. Kuang, L. Zhao,"A Practical GPU Based KNN Algorithm,"*In Proceedings of the Second Symposium International Computer Science and Computational Technology (ISCSCT 09),* 2009, pp. 151-155, ISBN: 978-952-5726-07-7.

[14] A. Arefin, C. Riveros, R. Berretta, and P. Moscato,"kNN-Boruvka-GPU: a fast and scalable MST construction from kNN graphs on GPU,"*In ICCSA'12 Proceedings of the 12th international conference on Computational Science and Its Applications - Volume Part I,* 2012, pp. 71-86, ISBN: 978-3-642-31124-6.

[15] V. Garcia, F. Nielsen,"Searching High-Dimensional Neighbours: CPU-Based Tailored Data-Structures Versus GPU-Based Brute-Force Method,"*In Lecture Notes in Computer Science, Springer Berlin Heidelberg* 2009, pp. 425-436, ISBN: 978-3-642-01810-7.

[16] J. Pan D. Manocha,"Fast GPU-based locality sensitive hashing for k-nearest neighbor computation,"*In Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems,* 2011, pp. 211-220, ISBN: 978-1-4503-1031-4.

[17] F. Jianbin, A.L. Varbanescu, H. Sips,"A Comprehensive Performance Comparison of CUDA and OpenCL,"*In Conference of Parallel Processing (ICPP)* 2011, pp.216-225, ISBN: 978-1-4577-1336-1.

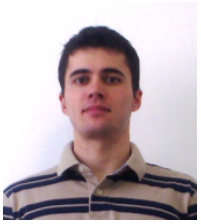
[18] Ch.L. Su, P.Y. Chen, CH.CH Lan,L.S Huang, K.H. Wu,"Overview and comparison of OpenCL and CUDA technology for GPGPU,"*Circuits and Systems (APCCAS), 2012 IEEE Asia Pacific Conference on* 2012, pp. 448-451, ISBN: 978-1-4577-1728-4.

[19] A. Kovacs, Z. Prekopcsak,"Robust GPGPU plugin development for RapidMiner,"*In RapidMiner Community Meeting And Conference - RCOMM 2012,* 2012.

[20] A. Munshi, B. Gaster, T. Mattson, J. Fung, D. Ginsburg,"OpenCL Programming Guide,"2011.

[21] NVIDIA,"OpenCL Programming Guide for the CUDA Architecture,"2010, Available: `http://www.nvidia.com/content/cudazone/download/OpenCL/NVIDIA_OpenCL_ProgrammingGuide.pdf`

[22] NVIDIA Inc,"GPU-Based Deep Learning Inference: A Performance and Power Analysis,"2015, Available: `http://www.nvidia.com/content/tegra/embedded-systems/pdf/jetson_tx1_whitepaper.pdf`.



Jan Masek is Ph.D. student at the Department of Telecommunications, Faculty of Electrical Engineering, Brno University of Technology, Brno, Czech Republic. He obtained his MSc. in 2012 (Communications and Informatics). He is interested in image processing, data mining, parallel systems.

Dr. Radim Burget is associated professor at the Department of Telecommunications, Faculty of Electrical Engineering, Brno University of Technology, Brno, Czech Republic. He obtained his MSc. in 2006 (Information Systems) and his finished his Ph.D. in 2010. He is associated professor since 2014. He is interested in image processing, data mining, genetic programming and optimization.

Lukas Povoda is Ph.D. student at the Department of Telecommunications, Faculty of Electrical Engineering, Brno University of Technology, Brno, Czech Republic. He obtained his MSc. in 2014 (Communications and Informatics). He is interested in image processing, text processing, and genetic programming.

Professor Dr. Malay Kishore Dutta is professor at Amity University, Department of Electronics & Communication Engineering, where he is Additional Director (ADET), Joint Acting Head (ASET), Professor & HOD ECE. He is interested in Digital Watermarking & Encryption techniques of multimedia signals, Image Processing, Computer Vision & Pattern Recognition, Audio Signal processing, Feature extraction from audio signals and its application, Time-frequency analysis of Signals using STFT & Multi-resolution methods like Wavelet Decomposition, Medical Image Segmentation, Fundus Image analysis.