Towards a Learning-Based Performance Modeling for Accelerating Deep Neural Networks

Damiano Perri[1], Paolo Sylos Labini[2], Osvaldo Gervasi[1(✉)], Sergio Tasso[1], and Flavio Vella[2]

[1] Department of Mathematics and Computer Science, University of Perugia, Perugia, Italy
osvaldo.gervasi@unipg.it, osvaldo.gervasi@gmail.com
[2] Lab for Advanced Computing and Systems, Free University of Bozen-Bolzano, Bolzano, Italy

Abstract. Emerging applications such as Deep Learning are often data-driven, thus traditional approaches based on auto-tuners are not performance effective across the wide range of inputs used in practice. In the present paper, we start an investigation of predictive models based on machine learning techniques in order to optimize Convolution Neural Networks (CNNs). As a use-case, we focus on the ARM Compute Library which provides three different implementations of the convolution operator at different numeric precision. Starting from a collation of benchmarks, we build and validate models learned by Decision Tree and naive Bayesian classifier. Preliminary experiments on Midgard-based ARM Mali GPU show that our predictive model outperforms all the convolution operators manually selected by the library.

1 Introduction

With the advent of big-data and data-driven applications such as deep learning, convolutional neural network for image classification and graph analytics among the others, the traditional library design looses performance portability mainly due to the unpredictable size and structure of the data. Specific algorithms and implementations are mostly designed by taking into account specific characteristics of the input or the targeting architecture. Autotuners partially mitigate this problem by adapting the implementation to the underline architecture, for example by selecting the best Local Work Size on OpenCL compliant GPUs [14]. Vendors libraries (e.g., Nvidia CuBLAS) still apply manual heuristics in order to select at runtime highly-optimized code for specific inputs. Convolution, the most crucial and computationally expensive part of both the training and inference step in CNNs, represents a notable example where it is quite hard to determine the best implementation for a given input [11]. The choice among direct, Image-to-column, FFT-based or Winograd-based algorithms may vary even in the same CNN, since different layers requires convolution operators to act on different input sizes.

© Springer Nature Switzerland AG 2019
S. Misra et al. (Eds.): ICCSA 2019, LNCS 11619, pp. 665–676, 2019.
https://doi.org/10.1007/978-3-030-24289-3_49

The aim of this work is to study a model-driven approach in order to improve the performance of the ARM Compute library by predicting the best convolution methods for a given convolution layers. Therefore, the model must be able to discriminate the architecture, numerical precision and input size.

The contributions of this preliminary work is twofold:

- we describe a methodology to generate the dataset used to build a predictive model.
- we evaluate a machine-learning based model on a convolutional neural networks on ARM GPUs

The rest of paper is organized as follows. Section 2 provides a brief description of related works. The background is given in Sect. 3. The main contribution of the work is reported on Sect. 4 (Methodology) and Sect. 5 (Experiments). Section 6 concludes the paper by underlining the lesson learned and the possibilities for future works.

2 Related Work

The size of the input matrices and kernels has been carefully analyzed as a function of different systems (server CPU, server GPU, mobile phone) by M. Cho and D. Brand in [2]. The authors carried out a systematic comparison between the main convolution methods (Winograd, image to column and Generic Matrix Multiplication and Fast Fourier Transform). The effects of multiple input channels have been studied by A. Vasudevan et al. in [20], who carried out a systematic analysis of the performances of the Image to Column and GEMM method varying the input channels and kernel sizes (e.g: 3×3 and 5×5), benchmarking the performances on various architectures (Intel Core i5-4570 and ARM Cortex-A57) on different neural networks (VGG-16, GoogleNet, AlexNet). The effects on performances of varying the accuracy has been studied by Vella et al. [12]. As reported, a reduction of the accuracy by 7% increases three times the performances on a Firefly board. Input aware techniques [6] are recently used to address the problem of performance portability on different applications [4,10]. Other seminal works successfully investigated predictive models for the performance modeling [17] for accelerating linear algebra routines [3] or improving the scheduling of processes on hybrid systems [19]. Their results inspired us to adopt a Machine Learning approach to find optimal implementations of the convolution operation and provided several insights for our experimental setup.

3 Background

Convolutional Neural Networks (CNNs) are a class of deep, feed-forward artificial neural networks that are often used to recognize objects in images. CNNs are composed by a set of layers linked by consecutive convolution operations. In image classification tasks, each layer is organized as a multi-channel, two-dimensional collections of neurons. Layers, along with the convolution operators

acting on them, are characterized by several parameters such as width, height, depth, filter dimension, pad and stride. The last layer (output layer) is reduced to a single vector of probability scores, so that a CNN transforms the original pixel values of an input image to the final class scores. The shape of such layers is usually fixed beforehand, while the kernel coefficient of their convolution operators are tuned through training.

Convolution. A great number of standard signal-processing operations are described by linear and time invariant operators. Their action on a function f can be implemented through convolution with a filter (or kernel) k, indicated with the $*$ operator and defined as:

$$(f * k)(t) = \int f(\tau)k(t-\tau)d\tau \tag{1}$$

Often, convolution is applied to discrete, finite signals, such as digital images. For computing a convolution in the notable case of 2-dimensional, single channel digital images, a variation of Eq. (1) is employed:

$$(f * k)(x, y) = \sum_{i=0}^{M} \sum_{j=0}^{N} f(i, j)k(x-i, y-j) \tag{2}$$

Thus, convolution changes the value (color, transparency, etc.) of a pixel to a weighted sum of all other pixels. These weights are the entries of the kernel matrix k, translated so that its center lies on the target pixel. Usually, the kernel matrix is null everywhere but a small region around its center, so that the value of a point after a convolution depends only on its close neighbours. The size of the non-zero part of the kernel may be arbitrary, but a 3×3 matrix if often used in image processing applications.

Convolution is thus a general purpose filter effect for images. Varying the convolution kernel, we may obtain a variety of effects, such as enhancing the edges, increasing the contrast, dilating or eroding the area occupied by the objects in the picture, and so on.

Performing a "direct convolution" and computing directly Eq. (2) can be unnecessarily costly, so other indirect methods are often preferred. The following sections reports a brief overview of some of these methods.

Coppersmith Winograd. In 1969, Strassen developed a matrix multiplication algorithm with complexity $O(n^{2.81})$, outperforming the standard $O(n^3)$ algorithm through the use of a number of intermediate products and additions. Subsequently, in 1986, he developed the "laser method", which further reduced the complexity to $O(n^{2.48})$. The following year, Coppersmith and Winograd developed a faster, now popular algorithm with complexity equal to $O(n^{2.375477})$. Although this algorithm comes with a lower asymptotic cost than its predecessors, a large multiplicative constant in the omega notation makes it truly efficient only when the matrices have particularly large dimensions. Since it is possible to

recast Eq. (2) as a product of matrices, a Winograd-based convolution is possible, and actually very efficient and numerically stable, especially for small 3×3 kernels.

Fast Fourier Transform. Since Fourier functions are the eigenfunctions of the convolution operator, convolution is easily performed in the Fourier domain as a multiplication between the function and the kernel coefficients. The Fast Fourier Transform (FFT) is a computationally fast way of obtaining the Fourier coefficients of a signal. A convolution through FFT can be very efficient when it involves large filters, since the cost of applying the filter in the Fourier domain is small compared to that of transforming the two signals. Unfortunately, as already noted, most modern applications use very small filters that can easily run on highly parallel system, making FFT-based convolution less convenient. Combined with an inherently weak numerical precision, this makes it hard for the FFT procedure to compete with some other methods. For comparison, we report here results extracted from the work of Andrew Lavin and Scott Gray on the comparison between FFT-based and Winograd-based convolution methods. In November 2015, they tested the two algorithms by running them on nVidia Maxwell architecture, specifically using a Titan X graphics card.

Using 32 bit floating point and a 3×3 filter, Winograd performed better: it achieved an error rate of $1.53 * 10^{-5}$, against the $4.01 * 10^{-5}$ of the FFT. Interestingly, when using 16 bit floating point, the two techniques obtained the same level of precision, but in both cases the Winograd speed performances were better by a factor of 2.44.

Image to Column and GEMM. GEneral Matrix to Matrix Multiplication (GEMM) indicates the standard low-level routine for performing matrix-matrix multiplications. Its implementations are usually extremely optimized for speed and can benefit from special floating point hardware. Since images and kernels are represented in memory as five-dimensional arrays (colored RGB), it is necessary to reshape them into 2D matrices in order to perform GEMM. To this end, a color channel is first selected, and then the an *Image to column* procedure is applied. Image-to-column rearranges discrete image blocks into columns, and then dispose the concatenated columns in a new matrix. The order of the columns in the new matrix is determined by traversing the original image in a column-wise manner. This operation has the considerable disadvantage of increasing the occupied memory, since the pixels of the image are replicated for the generation of the new matrix. Once the matrix associated with the image is obtained, the same procedure is applied to the kernel array, and a new matrix is then generated to be used for multiplication. Since GEMM is highly optimized, it can allow better performance than direct convolution. As can be understood, the occupation of memory increases as a result of the generation of several new matrices: with a $n*n$ kernel matrix, for example, a column matrix which is k^2 times larger than the original image is generated.

Yet, in most situations this consistent memory cost comes with a yet more consistent speedup, so that the *image to column and GEMM* approach is employed by a number of Deep Neural Network (DNN) frameworks that target GPUs such as Caffe, Theano and Torch.

3.1 Supervised Classifiers

In the present work we evaluate two of the simplest supervised machine learning methods used for classification and regression, the Decision Tree (DT) [13], and naive Bayesian classifier (nBC) [16]. These straightforward, white-box models greatly simplify this preliminary study and allow us to concentrate on the feasibility of the proposed task.

In the future, we plan to investigate the relation between the characteristics of the input and the parallel implementation of convolution operator provided by the ARM Compute Library through the use of more sophisticated classifiers.

Decision Tree. A *decision tree* is an abstract structure similar to a flow chart graph. In such a tree, nodes identifies decision points, or tests, whit arcs representing outcomes of such test. When the task is classification, leafs are interpreted as class labels, so that traversing the tree from the root to a leaf determines a solution of the decision problem. When creating a decision tree for a particular classification problem, one aims to minimize the overall depth while maximizing accuracy, so that the average classification instance is solved traversing the smallest possible number of decision nodes. We employed standard ML tools like pruning and used metrics such as the Gini index to reduce complexity and limit overfitting in our DTs.

Naive Bayesian Classifier. A Bayesian network is described by a direct acyclic graph, with nodes representing random variables and arcs representing dependencies. Linked nodes shares a direct dependency, while unconnected ones are implied to be conditionally independent. In such networks, nodes have a probability distribution that can be assumed or calculated from their neighbours'. This makes it very easy to ask and answer queries about the probability of a variable given some evidence on the others. In a naive Bayes classifier, all class labels are considered conditionally independent from each other, and depends only on a single input parent node. Despite their extreme simplification, these classifiers have demonstrated exceptionally capable in a variety of real classification tasks. The distributions and the dependency structure of a Bayesian network can be constructed ad-hoc from previous knowledge of the system or learned from a dataset, and a variety of algorithms exists for training naive Bayes classifiers. Our choices in this regard are described in the next section along with the employed methodology.

4 Methodology and Framework Description

The proposed methodology can be logically divided in three steps: the *dataset generation*, where we studied the performances of three convolution implementations on a variety of CNN layer shapes, recording the most successful in each instance; the *training*, where we used the dataset to train our classifiers at coupling a layer shape with the optimal convolution implementation for that layer; and finally the *model validation*, were we investigated the performance of our model-driven convolution against the optimal and standard approaches.

In what follows, we describe these steps and provide some information on the details of our implementation. The code is available git on https://github.com/DamianoP/AdaptiveMethods.

Dataset. In the first part, we evaluated the performance of the direct, Winograd-based and GEMM-based implementations of the convolution operator. We generated the dataset by collecting the outcome of more than 4000 experiments on artificial CNN layer architectures, stored the performances of each implementation on each layer and identified the fastest in view of the training step.

Each convolution layer, as mentioned in Sect. 3, is completely described by five parameters. Three regards the input image: its width W, its height H, its number of channels C_IN. Two characterize the kernel: its side length $KERNEL_SIZE$, and the number of output channels C_OUT. We generate the layers varying each parameter separately. Specifically, the parameter `W` and `H` can take the values 7, 128 or 256. The parameter `C_IN` ranges between 3 and 2048 with a multiplicative factor of 32. 384 and 768 were also added to these values. The parameter `KERNEL_SIZE` ranges between 1 and 11 with an increment factor of 1. The parameter `C_OUT` ranges between 8 and 1024 with a multiplicative factor of 2, 384 and 768 were also added to these values. The stride and padding parameters are set to 1.

Our python script `tool-prepare-dataset` generates a dataset of performances by executing NNTest over several artificial CNN shapes. The dataset is stored in a `.csv` as list of tuples, each containing the feature set of the layer a label with the fastest implementation. An example is reported in Fig. 1.

```
,W,H,C_IN,KERNEL_SIZE,C_OUT (filters count),STRIDE,PAD,,Layer name
,7,7,256,1,256,1,1,,Test 1
,7,7,256,1,512,1,1,,Test 2
,7,7,256,1,1024,1,1,,Test 3
,7,7,256,3,256,1,1,,Test 4
,7,7,256,3,512,1,1,,Test 5
.........................
,X1,X2,X3,X4,X5,X6,X7,,Test N
```

Fig. 1. Example of tensor shapes.

After generation, each input shape was used to evaluate the three implementations provided by the ARM Compute Library and by CK NNTest: direct convolution, winograd, Image to column and GEMM. This step returns two different files as output. The first one is used by existing ML framework (`.arff` file) like Weka [9] and Scikit-learn [15]. The second one represent the dataset. Each row is a pair (tensor, label). Specifically the label is an ordered list of pairs (algorithm name, execution time).

Training. In this phase, we trained the Bayesian and the decision tree classifiers on the dataset. The training, in our case carried through the Scikit-Learn python library, aimed to predict performances of convolution implementation based on the feature set of a layer.

In our code, the pyhton script `modelGenerator.py`, takes an `.arff` dataset file as input and outputs a `.joblib` file that contains the trained model. Based on the information contained in the `.arff` and `.csv` file from the dataset generation phase, we derived the optimal classifier parameter to be used in the next phase for the model evaluation.

Model Evaluation. Finally, we evaluate the quality of the model in terms of accuracy and performance. We test our classifiers on two real-world CNNs: Inception v3 [18], composed by 66 convolution layers, and MobileNets, composed by 15 convolution layers.

In our code, for each network we initialize a new classifier, loading in memory the data of the previously trained `.joblib` model. Our script scans the `.csv` layers list and follows this procedure:

- The classifier predicts the fastest algorithm for the current layer.
- The script retrieves the optimal implementation using the ranking file, for comparison with the classifier's choice.
- The script stores the calculation times of the three implementation, and the classifier prediction.

Finally, a summary is generated, storing the calculation time of the entire network. The results are plotted in a figure such as Fig. 3a, the details of which will be explained in the next section.

5 Preliminary Results

Before discussing preliminary results, we describe the hardware/software infrastructure used for the experiment below.

5.1 Experimental Setup

The hardware setup used for the tests is an ARM Soc with an ARM Mali-T860 equipped with 4 Mali core able to operate at 2 GHz of frequency and 4 GB of DDR3.

We used the ARM Compute Library, an open-source collection of low-level routines optimized for ARM CPU and GPU architectures targeted at image processing, computer vision, and machine learning. It provides basic arithmetic, mathematical, and binary operators and CNN building blocks. As for convolution, it is implemented in three different ways: image to column and GEMM, direct convolution and Winograd. All those methods can be selected at run-time. Depending on the ARM architecture, numeric precision and specific input each methods can exhibit different performance [12,22].

For benchmarks of each convolution implementation and for the generation of the datasets we used the NNTest library [8,12], an open-source library for collaboratively validating, benchmarking and optimizing neural net operators across platforms, frameworks and datasets.

Concerning the model generation frameworks, we use Weka and Scikit-learn. They provide several classification, regression and clustering algorithms. This was used for the training, the tuning and the validation of predictive models.

5.2 Results

In the present section we evaluate the performance of the predictive models trained by a DT and nBC against the implementations of the convolution operator provided by the ARM Compute Library. We analyze the accuracy of the classifiers as well as the execution time of the ARM Compute Library by using predictive models. In Fig. 2b and Fig. 3b, we show the inference phase by using "Inception v3" CNN.

On top of each picture we indicate the classifiers used for training the predictive model, the numerical precision, the convolution neural network and the related number of layers. In each figures, the Y axis represents the execution time needed to perform the convolution operator over of all the layers (microseconds). The columns denote the execution time of each different implementation of convolution:

- image to column and GEMM method;
- direct convolution method;
- Winograd method;
- method predicted by the model;
- the possible best algorithm oracle (the oracle gives information about the best algorithm. The best algorithm is selected by choosing the one with the minimum execution time between Winograd, Image to column and GEMM, Direct Convolution for the layer currently analyzed.)

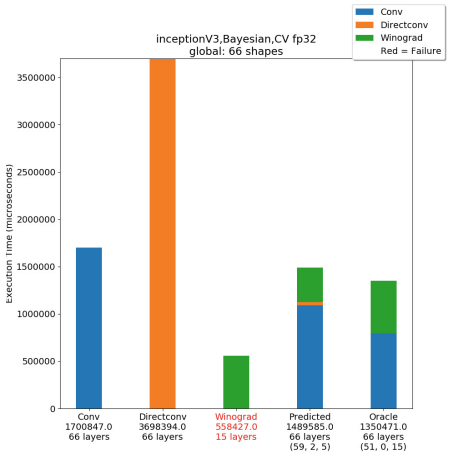
Table 1. Classifier: naive Bayes classifier

	Accuracy	vs IMG to column and GEMM	vs Direct convolution	vs Winograd
MobileNets	93.33%	0.99X	3.02X	winograd failed
Inception v3	81.82%	1.14X	2.48X	winograd failed

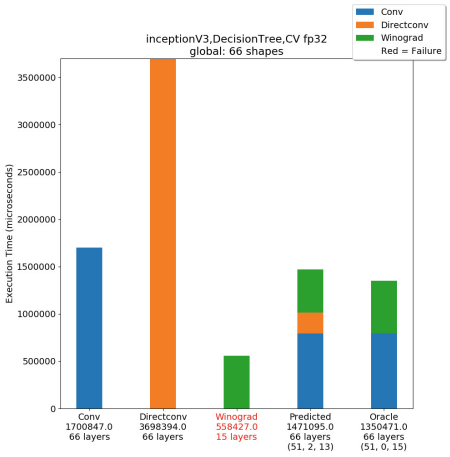
Table 2. Classifier: decision tree

	Accuracy	vs IMG to column and GEMM	vs Direct convolution	vs Winograd
MobileNets	100.00%	1.00X	3.04X	winograd failed
Inception v3	96.97%	1.15X	2.51X	winograd failed

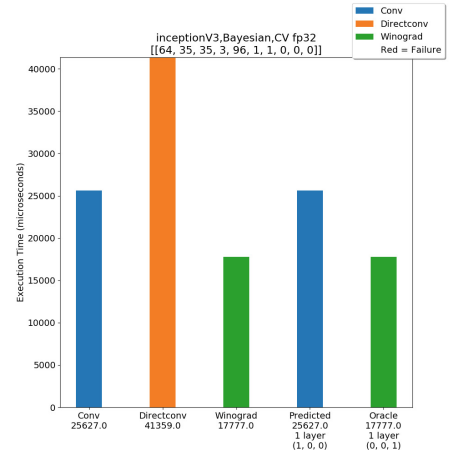
(a) All convolution layers naive Bayes



(b) All convolution layers Decision Tree



(c) Single layer naive Bayes



(d) Single layer Decision Tree

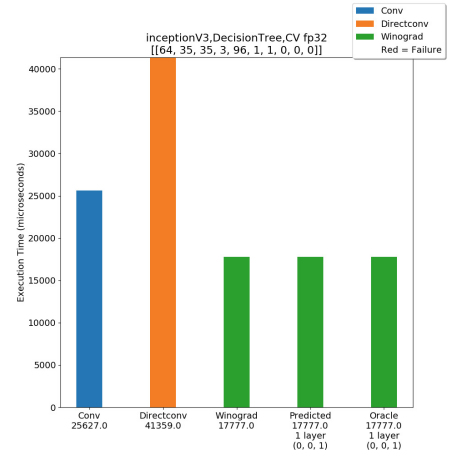


Fig. 2. Inception V3: naive Bayes vs decision tree

Below each column we report the total time in microseconds and the number of layers correctly completed. In addition, the columns related to the predictive model and the oracle report a triple representing the times *image to column and GEMM*, *direct convolution* and *Winograd* have been selected by that method.

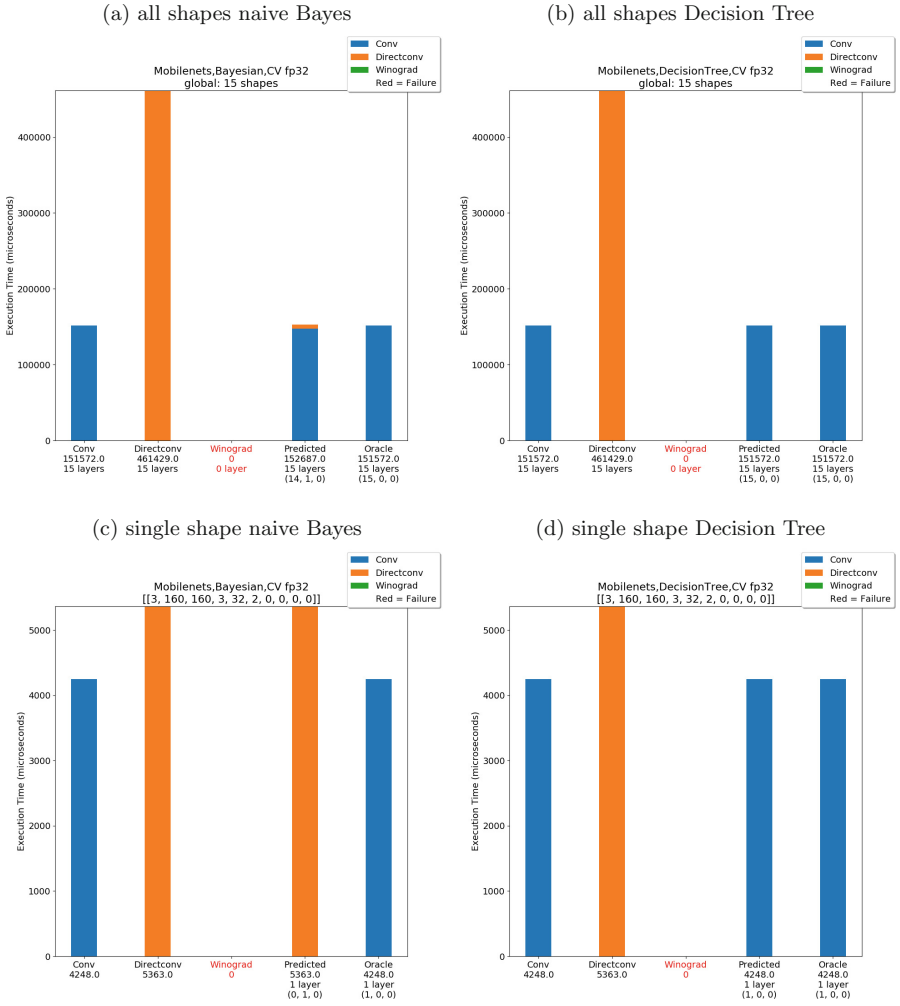


Fig. 3. MobileNets: naive Bayes vs decision tree

Tables 1 and Table 2 summarize the performance of the two models for both networks. In general, the predictive models perform better than each manually selected method except for one case, showing an high accuracy in the selection of the best implementation.

Comparing the predictive models, that one based on the decision tree shows a better accuracy than nBC. However the overall computation times are still comparable.

The results for InceptionV3 are detailed in Fig. 2a. In the two top images the performance over the whole network are shown. The performances of the models learned from both classifiers are slightly worse than the optimal ones. However, the library that uses the model driven approach achieves an improvement over the handed-selected methods.

In the bottom images, a specific layer has been considered. In this example, the model based on the Bayesian approach erroneously selected a GEMM convolution instead of Winograd. Contrarily, the decision tree make the right prediction.

The same analysis is reported for MobileNets in Fig. 3a. For this network, the optimal convolution implementation was always GEMM. In general, the model learned by the decision tree shows better performance than the model based on naive Bayes classifier and the statically selected methods.

6 Conclusions and Future Work

We investigate the opportunity of using learned models to accelerate convolution operator in the case of a library that exhibits multiple implementations. We evaluate our predictive models on two different CNN, InsectionV3 and MobileNet (inference phase) on a low-power consumption ARM GPU.

Our approach outperforms the ARM Compute Library with speed-ups up to $3x$. Future developments are going to focus on the improvement of the predictive models with more sophisticated and tunable classifiers. Also, since the dataset generation is the most expensive part of the proposed methodology, we are going to investigate solutions based on reinforcement learning. We will also explore predictive models for accelerating irregular application like graph analytics [1,21], for selecting the best parallel strategy on GPU [7] or optimizing communication on distributed systems [5].

Acknowledgments. We thank *Dividiti Inc.* for the huge support on CK and NNTest and for providing hardware resources.

References

1. Bernaschi, M., Bisson, M., Mastrostefano, E., Vella, F.: Multilevel parallelism for the exploration of large-scale graphs. IEEE Trans. Multi-scale Comput. Syst. **4**(3), 204–216 (2018)
2. Cho, M., Brand, D.: MEC: memory-efficient convolution for deep neural network. CoRR, abs/1706.06873 (2017)
3. Cianfriglia, M., Vella, F., Nugteren, C., Lokhmotov, A., Fursin, G.: A model-driven approach for a new generation of adaptive libraries. arXiv preprint arXiv:1806.07060 (2018)
4. Cosenza, B., Durillo, J.J., Ermon, S., Juurlink, B.: Autotuning stencil computations with structural ordinal regression learning. In: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 287–296. IEEE (2017)
5. Di Girolamo, S., Vella, F., Hoefler, T.: Transparent caching for RMA systems. In: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 1018–1027. IEEE (2017)
6. Falch, T.L., Elster, A.C.: Machine learning based auto-tuning for enhanced OpenCL performance portability. In: 2015 IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW), pp. 1231–1240. IEEE (2015)

7. Formisano, A., Gentilini, R., Vella, F.: Accelerating energy games solvers on modern architectures. In: Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms, p. 12. ACM (2017)
8. Fursin, G., Temam, O.: Collective optimization: a practical collaborative approach. ACM Trans. Archit. Code Optim. (TACO) **7**(4), 20 (2010)
9. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The weka data mining software: an update. ACM SIGKDD Explor. Newslett. **11**(1), 10–18 (2009)
10. Hou, K., Feng, W.-c., Che, S.: Auto-tuning strategies for parallelizing sparse matrix-vector (SPMV) multiplication on multi-and many-core processors. In: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 713–722. IEEE (2017)
11. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. Adv. Neural Inf. Process. Syst. **25**, 1097–1105 (2012)
12. Lokhmotov, A., Chunosov, N., Vella, F., Fursin, G.: Multi-objective autotuning of mobilenets across the full software/hardware stack. In: Proceedings of the 1st on Reproducible Quality-Efficient Systems Tournament on Co-designing Pareto-efficient Deep Learning, p. 6. ACM (2018)
13. Maron, M.E.: Automatic indexing: an experimental inquiry. J. ACM (JACM) **8**(3), 404–417 (1961)
14. Nugteren, C., Codreanu, V.: CLTune: a generic auto-tuner for OpenCL kernels. In: 2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC), pp. 195–202 (2015)
15. Pedregosa, F., et al.: Scikit-learn: machine learning in python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
16. Safavian, S.R., Landgrebe, D.: A survey of decision tree classifier methodology. IEEE Trans. Syst. Man Cybern. **21**(3), 660–674 (1991)
17. Singer, B., Veloso, M.: Learning to predict performance from formula modeling and training data. In: ICML, pp. 887–894 (2000)
18. Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., Wojna, Z.: Rethinking the inception architecture for computer vision. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2818–2826 (2016)
19. Tasso, S., Gervasi, O., Vella, F., Cuzzocrea, A.: A simulation framework for efficient resource management on hybrid systems. In: 2015 IEEE 18th International Conference on Computational Science and Engineering, pp. 216–223. IEEE (2015)
20. Vasudevan, A., Anderson, A., Gregg, D.: Parallel multi channel convolution using general matrix multiplication. CoRR, abs/1704.04428 (2017)
21. Vella, F., Bernaschi, M., Carbone, G.: Dynamic merging of frontiers for accelerating the evaluation of betweenness centrality. J. Exp. Algorithmics (JEA) **23**, 1–4 (2018)
22. Zheng, L., Chen, T.: Optimizing deep learning workloads on arm GPU with TVM. In: Proceedings of the 1st on Reproducible Quality-Efficient Systems Tournament on Co-Designing Pareto-Efficient Deep Learning, p. 3. ACM (2018)