The Journal of Supercomputing
https://doi.org/10.1007/s11227-022-04577-2

BestOf: an online implementation selector for the training and inference of deep neural networks

Sergio Barrachina[1] · Adrián Castelló[2] · Manuel F. Dolz[1] · Andrés E. Tomás[2]

Accepted: 30 April 2022

Abstract

Tuning and optimising the operations executed in deep learning frameworks is a fundamental task in accelerating the processing of deep neural networks (DNNs). However, this optimisation usually requires extensive manual efforts in order to obtain the best performance for each combination of tensor input size, layer type, and hardware platform. In this work, we present BestOf, a novel online auto-tuner that optimises the training and inference phases of DNNs. BestOf automatically selects at run time, and among the provided alternatives, the best performing implementation in each layer according to gathered profiling data. The evaluation of BestOf is performed on multi-core architectures for different DNNs using PyDTNN, a lightweight library for distributed training and inference. The experimental results reveal that the BestOf auto-tuner delivers the same or higher performance than that achieved using a static selection approach.

Keywords Deep neural networks · Auto-tuning · Implementation selector · Python

Abbreviations

BLAS	Basic linear algebra subprograms
DNN	Deep neural network
CNN	Convolutional neural network
Conv2D	2-dimensional convolution

✉ Manuel F. Dolz
dolzm@uji.es

Sergio Barrachina
barrachi@uji.es

Adrián Castelló
adcastel@disca.upv.es

Andrés E. Tomás
antodo@upv.es

[1] Universidad Jaume I, Castellón de la Plana, Spain

[2] Universitat Politècnica de València, València, Spain

Published online: 20 May 2022

ConvGEMM	Convolution via GEMM with implicit IM2COL
DL	Deep learning
GEMM	General matrix multiply
im2col	Image to column transform
im2row	Image to row transform
MLP	Multi-layer perceptron
PyDTNN	Python distributed training of neural networks

1 Introduction

Artificial intelligence, and, in particular, machine learning via deep neural networks (DNNs) have experienced explosive growth due to the appearance of new algorithmic techniques, vast amounts of computer power, and an increased amount of training data [1–5]. This scenario has pushed the industry to design customised architectures for deep learning (DL), e.g. NVIDIA's Tensor Cores or Google's TPUs, as well as to develop frameworks such as Google's TensorFlow or Facebook's PyTorch.

Tuning and optimising DL frameworks on these customised platforms are fundamental to reducing the overall training and inference costs [6]. For instance, the realisation of the forward and backward passes for the training of a convolutional layer may deliver distinct performance results depending on the selected algorithmic variant and the problem (layer) size. Similarly, the configurations to conduct individual tensor operations, such as paddings, shrinks or transpositions, may also affect the overall run time depending on their specific tensor size. A naive approach is to manually optimise the execution of DNN layers by selecting the best implementation according to post mortem profiling data. However, auto-tuners have been demonstrated to provide a better solution in these scenarios by selecting the algorithm for each problem that obtain the best performance [7, 8].

Following this trend, in this work, we present a novel online implementation selector for DL frameworks which automatically selects the best possible implementation at run time. In particular, this work makes the following contributions:

- We present BestOf, an online auto-tuner that selects the best algorithm for each problem according to their previous performance profiles within the same program execution. BestOf has been designed as a Python module and its interface can be easily used to replace actual calls in the original code for making selections. This auto-tuner is able to deal with grouped selections, where all routines in a group must be selected together due to implementation dependencies. Moreover, it can automatically manage and discover nested selections, allowing recursive decisions when inner functions also present alternative implementations.
- We integrate BestOf as a module on PyDTNN, a lightweight framework for distributed training and inference of DNNs [9], and instrument it to permit the selection of (i) algorithms to perform the forward-backward passes in convolutional neural networks (CNNs), via either IM2ROW+GEMM (*lowering* convolution to GEMM or General Matrix Multiplication [10]), CONVGEMM (GEMM with

implicit IM2ROW, see [11]), or variants of the Winograd algorithm [12]; and (ii) implementations to conduct 4D tensor transpositions.
– We evaluate the performance obtained by BestOf for training and inference with VGG16 and inference with ResNet34 using two multi-core nodes equipped with Intel Xeon Skylake processors. This study is completed with a per-layer analysis that assesses the performance gains, as well as the throughput attained along with the training steps.

The rest of the paper is organised as follows. In Sect. 2, we revisit some related work on auto-tuning tools and frameworks and compare them against the approach presented in this work. In Sect. 3, we describe the user interface and the internals of BestOf. In Sect. 4, we briefly introduce PyDTNN and detail how BestOf was integrated to select different implementation alternatives. In Sect. 5, we evaluate the benefits of BestOf by comparing its throughput with native versions. Finally, in Sect. 6, we close the paper with a summary and a collection of concluding remarks.

2 Related work

Current software libraries in general deploy distinct computational kernels depending on the underlying hardware. Typically, once the user selects the processor type (or specification) from within a limited list, the optimum computational kernel is selected [13]. Usually, this approach does not take into account other considerations that could affect the kernels performance, such as the problem dimensions.

Conversely, several automatic selections have been applied for decades in order to extract the maximum computational power of the hardware. The automatic selection of the best implementation for a computational kernel dates back to the ATLAS dense algebra library [14]. This library was probably the first popular BLAS implementation to execute benchmarks during its installation phase in order to select the best algorithm parameters. Among others, the main parameter in ATLAS is the matrix multiplication block size which depends heavily on the memory cache properties. This automatic selection has been extended recently to accelerator platforms, selecting not only the algorithm implementation but also which hardware to use (for example, CPU or GPU) [15, 16]. Nevertheless, as the selection is performed offline, the adopted decision cannot be changed afterwards. The main drawback of offline selectors is the potentially very large search space for all possible input sizes of an algorithm. Typically, libraries with offline selectors use heuristics or some form of optimisation to limit the number of tests performed during the installation process. This is particularly difficult for the convolution in neural networks which have a large number of parameters, exponentially increasing the search space. For instance, the work by Anderson et al. [17] uses partitioned boolean quadratic programming (PBQP) for selecting the optimal configuration after benchmarking all possible combinations of convolution implementations and layer sizes.

In contrast, an online approach traces the execution time during the actual computation, cycling over the different alternatives, to make a decision after sufficient performance data are collected [18–20]. This technique requires the repeated

application of an algorithm with the same parameter set, a condition that is met by the “iterative" nature of DNNs training.

The selection of a proper convolution algorithm has a major impact on DNN training performance as shown in [6] for GPUs. Popular toolkits, such as cuDNN (up to version 7) and OpenVINO, employ heuristics to predict which implementation will be faster given the specific set of parameters of the convolutions at hand. Offline selection methods have appeared in recent literature [7, 8], but even though the majority of neural network toolkits have provisions for benchmarking, as far as we know the latest version of cuDNN is the only one to provide a run-time selector for alternative convolution implementations.

The BestOf online selector presented in this work differs from other state-of-the-art alternatives in the following aspects: (i) it is implemented as a Python module, allowing an easy integration into the PyDTNN DNN training/inference framework, which is developed in the same programming language; (ii) it presents a very simple interface that permits making selections by simply replacing the actual calls in the original source code with calls to BestOf instances; (iii) it supports grouped selections and can automatically manage and discover nested selections for recursive decisions; and (iv) it is open source. Unfortunately, as we have not found comparable online selection tools that could be easily applied to our target application, it was not possible to experimentally compare BestOf with other solutions.

3 BestOf: An Online Implementation Selector

In this section, we present BestOf, an online auto-tuner developed in Python that is able to automatically execute a set of alternative algorithms and eventually select, after a given number of rounds, the best performing option for each problem type. The selections made by BestOf occur at run time according to the execution time data gathered from previous executions of the considered routine/algorithm for each problem size.

Application programming interface The BestOf API is detailed in the example shown in Listing 1, where we have declared a BestOf object for selecting the best implementation for the transposing of a Numpy 4D array. There, the constructor receives the transposition alternatives as a list of pairs, where each pair is formed by a name and a pointer to the function that should be called when that alternative is selected. BestOf requires all the alternatives to receive the same parameters in the same order. However, if this was not the case, it could easily be solved by wrapping the non-conformant functions. In the example, the first two alternatives of the operation are developed in `Cython` and present different loop orderings for transposing the tensor dimensions, while the last invokes the native `transpose` routine from Numpy.

The constructor also requires a pointer to a function that returns the problem size as a hashable object (`get_problem_size` parameter in Line 8). For the case of the transpose, this parameter corresponds to the array shape and is key to enabling BestOf to identify all the transpose calls that share the same problem size. Other parameters of the constructor are the `rounds` value, which specifies the number of

times that all the alternatives have to be executed until a decision is made (see Line 9); and the `pruning_speedup` factor, which aims to accelerate the decision-making by pruning those alternatives that are slower than any other by the specified factor. The pruning is performed only after all the alternatives have been evaluated a minimum given number of rounds, according to the `prune_after_round` parameter (see Lines 10–11).

```
1  best_transpose_1023 = BestOf(
2      name="Transpose 1023 methods",
3      alternatives=[
4          ("ijk_cyt", transpose_1023_ijk_cython_wrapper),
5          ("jik_cyt", transpose_1023_jik_cython_wrapper),
6          ("numpy", transpose_1023_numpy),
7      ],
8      get_problem_size=lambda m: m.shape,
9      rounds=10,
10     pruning_speedup=10.0,
11     prune_after_round=4
12 )
13 x = np.random.rand((10, 4, 32, 32))
14 x_t1023 = best_transpose_1023(x)
```

Listing 1 BestOf for the 4D transposition 1023.

Finally, the example also shows how the created BestOf object can be called on to perform the transpose (see Lines 13 and 14), while it will silently evaluate the alternative used on that occasion (or will call on the selected alternative if a decision has already been made for that problem size).

Internals The BestOf auto-tuner is defined as a Python class implementing the constructor, a series of auxiliary member methods, and the `__call__` method, which permits calling the instantiated object as if it were a function. In fact, the `__call__` method is the function in charge of measuring the execution times and making a selection of the best performing alternative when appropriate. This procedure is repeated until a specific number of rounds is reached. At that point, BestOf selects the alternative that delivers on average the best performance.

Functionality The BestOf auto-tuner is characterised by supporting the following two features: grouping and nesting.

Grouping One of the requirements for using this auto-tuner is that all the alternatives should work interchangeably, that is, they should not present any side effects or dependencies among them. In some cases, however, the alternatives may perform a series of optimisations that assume the state left from a previously called function. A practical example is the use of the IM2ROW transform in the forward and backward propagation methods in a convolution layer. As the same computed IM2ROW transformation is used in both methods, the forward method stores it in a temporary variable, so that the backward method does not need to re-compute it. This optimisation trades memory for execution speed and forces the use of the same algorithm for both the forward and backward phases.

To tackle such dependencies, BestOf can evaluate grouped implementations, which consists of a set of algorithms that have to be executed in conjunction. Listing 2 declares a BestOf object for selecting the best group of alternatives for executing the forward and backward phases of a convolutional layer using either: (i)

IM2ROW+GEMM; (ii) CONVGEMM; or (iii) the Winograd algorithm. To leverage this feature, each of the alternatives in the list has to be defined as a tuple containing the name given to that group and a list with the function pointers that constitute the group. Internally, BestOf keeps track of the execution times for each group and problem size, eventually executing the best performing group after completing the number of rounds specified. Note that, the problem size in the example has to be defined to include the input parameters of all functions in the group. In this case, we use a tuple that combines the shapes of the input and weight tensors, which serves to univocally determine the problem size in the group.

From the user's perspective, calling on a BestOf object that uses the grouping feature requires passing an index for identifying which function of the group has to be executed in the user's code (see Lines 12–13 in Listing 2).

```
1  best_forward_backward = BestOf(
2      name="Conv2D forward backward",
3      alternatives=[
4          ("im2col", [forward_i2c, backward_i2c]),
5          ("convgemm", [forward_cg, backward_cg]),
6          ("winograd", [forward_cw, backward_cw]),
7      ],
8      get_problem_size=lambda *args: args[0].shape    + args[0].weights.shape + \
9                                    (args[0].vpadding,  args[0].hpadding, \
10                                    args[0].vstride,   args[0].hstride, \
11                                    args[0].vdilation, args[0].hdilation) )
12 )
13 FORWARD_PASS, BACKWARD_PASS = 0, 1
14 x  = np.random.rand((10, 3, 32, 32))
15 y  = best_forward_backward(FORWARD_PASS, x)
16 dx = best_forward_backward(BACKWARD_PASS, dy)
```

Listing 2 BestOf for the forward and backward methods in a convolutional layer.

Nesting The second feature of this auto-tuner is the support for making nested selections, i.e. an alternate implementation that internally contains other calls to BestOf objects. In such cases, the selection proceeds by exploring the different branches of a decision tree that is evaluated at run time. To build the decision tree, the auto-tuner uses the `traceback` Python module, which reports the function calls made in the code at a specific point by retrieving the stack. Using the stack frames, BestOf checks whether the current object has been invoked from another instance in a previous frame and, in such cases, registers it as the parent. To select the best-performing branch in the tree, the auto-tuner makes decisions from the leaves to the root nodes. This is because each node is required to know the selection made in all its children prior to measuring the execution time of its alternatives. For this, the implementation of BestOf delays the evaluation of the parents until all their children have determined their best alternative. A practical example is shown in Fig. 1. In this case, the IM2COL forward version is among the forward options being evaluated by a BestOf instance. When the IM2COL forward version is invoked, a 4D transposition must be performed. As there are different possible implementations for this transposition, an additional BestOf instance will evaluate which implementation is faster. While the different transpositions of a given size are being compared, the corresponding BestOf parent will be locked, i.e. it will pause its own time comparisons. Another example is when the Winograd algorithm is among the different forward alternatives being evaluated by BestOf . In this case, as

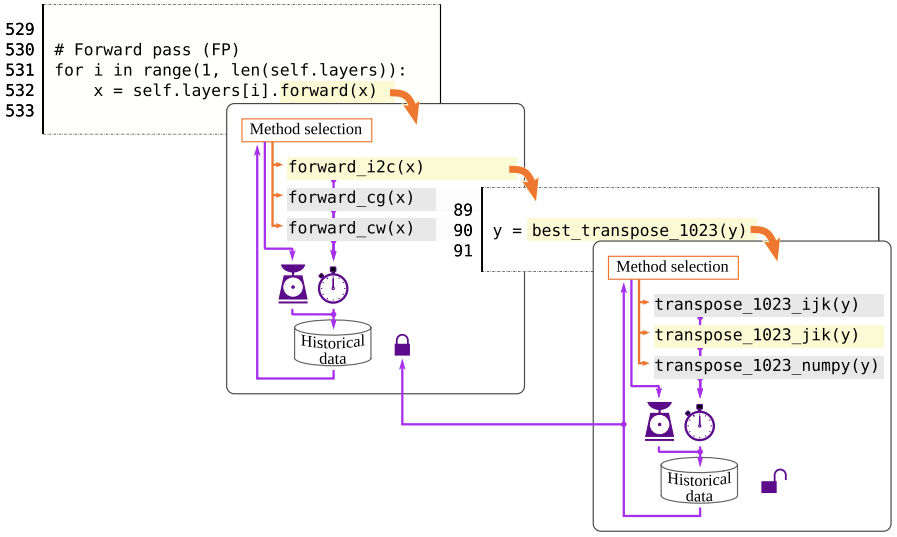

Fig. 1 Illustration of BestOf nesting: one of the forward implementations performs a transposition, which also has different possible implementations

the Winograd algorithm also selects among different variants, BestOf will automatically discover and manage these nested selections.

Apart from the aforementioned functionalities, the auto-tuner also provides, as a result, the collected performance metrics and the associated decision trees, which can be analysed postmortem by users to gain insights into the best performing implementations in different problem sizes.

4 Integration in PyDTNN

In this section, we briefly describe PyDTNN, a framework for distributed training and inference of DNNs, as the BestOf auto-tuner has been integrated into it. Next, we list the operations in PyDTNN that offer different implementation alternatives in order to leverage our implementation selector.

Overview of the DL framework. PyDTNN[1] is a lightweight framework for distributed training of DNNs on clusters of computers that has been designed as a research-oriented tool with a low learning curve. PyDTNN presents the following appealing properties:

– **Flexible** PyDTNN regards extensibility (and, to a certain extent, simplicity) as a first-class citizen to allow users to customise the framework to prototype research ideas.

[1] The PyDTNN framework is available at https://github.com/hpca-uji/PyDTNN/, under a GNU General Public License v3.0.

– **Ample functionality** PyDTNN covers DL training and inference for a significant part of the most common DNN models: multi-layer perceptrons (MLPs), convolutional neural networks (CNNs), and transformers for natural language processing. In practice, PyDTNN provides training and validation accuracies on par with those attained by Google's TensorFlow [9].
– **High performance** PyDTNN exploits *data parallelism* [21], relying on specialised message-passing libraries for efficient communication, and kernels from high performance multi-threaded libraries for the major computational operations in CPUs and GPUs.

Alternate implementations in PyDTNN. The integration of BestOf in PyDTNN consisted of incorporating the auto-tuner as a module and in identifying those operations that rely on alternate implementations. These operations are as follows:

– **Convolution algorithms** The convolutional 2D layers offered in PyDTNN currently support the execution of the forward and backward passes using the: (i) IM2ROW+GEMM; (ii) CONVGEMM; and (iii) Winograd minimal filtering algorithm. The incorporation of BestOf, in this case, leverages the afore-mentioned grouping feature, for selecting the best performing pair of forward-backward alternatives.
– **Winograd variants** When the preceding Winograd algorithm is applied on 3×3 kernel sizes, PyDTNN leverages two different variants using output tiles of size 2 and 4, respectively. The use of BestOf in this case leads to a nested selection, given that the Winograd algorithm, as previously explained, is already an algorithm alternative.
– **Tensor transpositions** While the Numpy `transpose` routine performs well for small tensor sizes, it may behave poorly for larger arrays, as the operation is executed in series, aside from being memory-bound. For this reason, PyDTNN provides two alternate OpenMP-parallel Cython implementations of the transpose which vary the order in which the dimensions are accessed.

While PyDTNN lacks the level of maturity and the complete functionality of production-level frameworks, such as TensorFlow or PyTorch, we believe that PyDTNN offers a more accessible and easier-to-customise solution for the efficient training and inference of DNN models. All these reasons have motivated us to accommodate BestOf within this framework and to evaluate the performance gains that can be obtained in both training and inference stages of DL models while selecting the best alternative for the three previous operations.

Table 1 Computer platforms used for the experiments

Platform	Altec	Volta
CPU model (Intel Xeon Gold)	5120	6126
# Cores	14	12
Frequency (GHz)	2.2	2.6
DDR4 RAM (GiB)	96	32

Table 2 DL framework parameters used for the experiments

PyDTNN parameters	
DNN model	VGG16 (training/inference), ResNet34 (only inference)
Datasets (batch size)	CIFAR-10 (64), ImageNet (32)
Data layout	NHWC
Number format	Floating-point Single-precision (FP32)

5 Experimental Results

In this section, we evaluate the performance of the BestOf auto-tuner within PyDTNN against the static selection of the different algorithms previously described. In particular, we evaluate the training and inference phases of the VGG16 model using different configurations of threads, datasets, and multi-core architectures; and the inference phase of the ResNet34 model.

For that, we measure the overall training and inference throughput of PyDTNN with each statically selected variant and with BestOf using the platforms listed in Table 1. The selected parameters for running the experiments in PyDTNN are shown in Table 2. The per-layer evaluation analyses the time spent by PyDTNN on the convolutional layers appearing in the VGG16 [22] and ResNet34 [23] models for the different convolution algorithms shown in Table 3. Note that, as all filters in the VGG16 and part of the ResNet34 models are of dimension 3×3, each time the Winograd alternative is called on to perform the corresponding convolution, BestOf will also evaluate the two possible Winograd variants that can be applied for this filter size.

5.1 Results on training

Figure 2 reports the throughput obtained by the different convolutional algorithms and the BestOf auto-tuner using 1, 4, 8, and 12 threads on Altec and Volta. The results show that the im2row transform followed by a gemm is consistently the best option for all cases. Even in this scenario, BestOf achieves the same performance when using the CIFAR-10 dataset (see left-hand side plots), and nearly the same performance as the best option in the case of ImageNet. Note that, these results have been obtained by training during a single epoch and 120 steps. Under a more realistic

Table 3 Convolution algorithms used for the experiments

Convolution algorithms	
IM2ROW+GEMM	IM2ROW from Cython v0.29.24 parallelized with OpenMP GEMM from Numpy v1.12.2 linked against BLIS v3.0
CONVGEMM	It uses microkernels provided by BLIS v.3.0
Winograd	In-house OpenMP-parallel library with SSE vector instructions and GEMM from BLIS v3.0

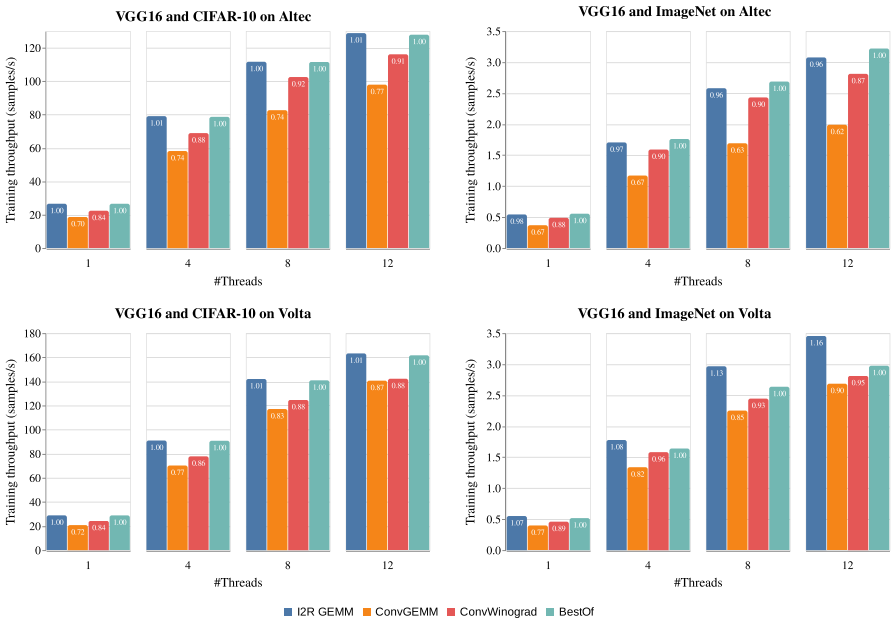

Fig. 2 Training performance of VGG16 on CIFAR-10 (left) and ImageNet (right) using ALTEC (top) and on VOLTA (bottom) using different forward-backward alternatives and BestOf with the same alternatives

scenario, i.e. over 40 epochs with more than 240 steps per epoch, the BestOf overhead due to the evaluation of non-optimal variants would be mostly diluted.

Figure 3 shows, for each VGG16 convolutional layer, the average time of the different forward-backward algorithms. For simplicity, the convolution layers of VGG16 that use the same input and kernel sizes are grouped in the plot. As reported there, the IM2ROW transform followed by a GEMM achieves the best performance in nearly all the layers for any combination of dataset and platform. Nevertheless, it is interesting to note that the relative performance among the different algorithms varies depending on the target architecture.

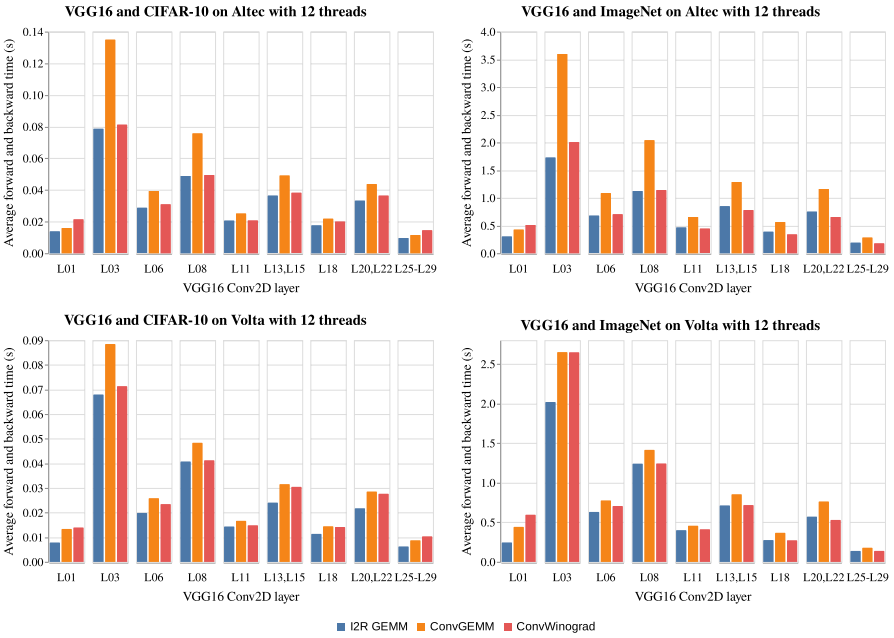

Fig. 3 Average time of the forward-backward alternatives evaluated by BestOf on the convolutional layers of VGG16 during the training on CIFAR-10 (left) and ImageNet (right) using Altec (top) and Volta (bottom) and 12 threads

5.2 Results on inference

Figure 4 depicts the throughput obtained by the different convolutional algorithms and the BestOf alternative when using 1, 4, 8, and 12 threads. The best algorithm for inference depends on the number of threads, the dataset, and the node architecture. This behaviour differs from that observed in the training scenario. For example, the best option for VGG16 and 8 threads on Altec is the Winograd algorithm, while for 12 threads, im2row+gemm is the best option. Likewise, the preferred option for VGG16 and 8 threads on Altec is the Winograd algorithm while for the same scenario on Volta, ConvGEMM is the best alternative. It is worth noting that BestOf not only achieves the performance of the best algorithm in each case, but also outperforms the other algorithms in all scenarios. This is because BestOf does not select the same algorithm for all the VGG16 layers. Figure 5 shows the same information as that in the previous figure, but for the ResNet34 model. As can be observed, the results are similar to those for VGG16, as the best choice on Altec on all the cases but one corresponds to the Winograd algorithm, and the best alternative on Volta in all the cases is the ConvGEMM algorithm.

Figure 6 shows, for each VGG16 convolutional layer, the average time of the distinct forward algorithms with 12 threads. As shown, the best forward algorithm depends on the layer (or problem size), the dataset, and the target node architecture. The same effect can be observed when the ResNet34 model is leveraged (see Fig. 7). Note that, for this

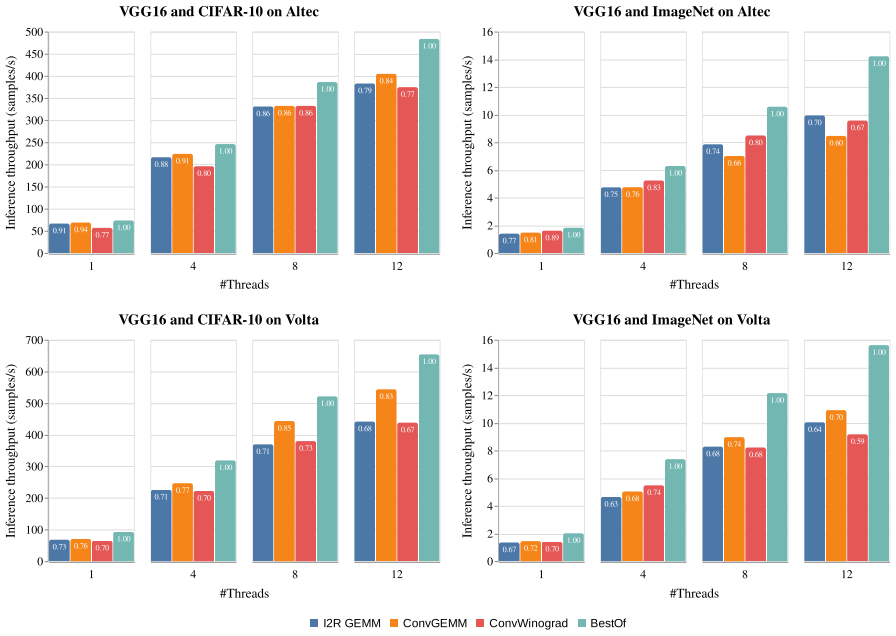

Fig. 4 Inference performance of VGG16 on CIFAR-10 (left) and ImageNet (right) using ALTEC (top) and VOLTA (bottom) using different forward-backward alternatives and BestOf with the same alternatives

model, the Winograd algorithm cannot be used on some of its layers. For these layers, PyDTNN instructs BestOf to resort only to Winograd and CONVGEMM algorithms.

5.3 Evolution of the training and inference performance

To gain insights into the behaviour of BestOf, we have also analysed its throughput over time. Figure 8 depicts the performance evolution over time of the different algorithms for training and inference with VGG16 using 12 threads on ALTEC. As expected, all the PyDTNN variants performing a static selection perform quite uniformly during their entire execution. In contrast, the BestOf variant starts at a given performance that is steadily increased until the best alternative is identified. For the training experiment, the achieved performance is similar to the IM2ROW+GEMM variant, while for the inference scenario, the BestOf selection outperforms all other variants, as it individually selects the best algorithm for each VGG16 layer.

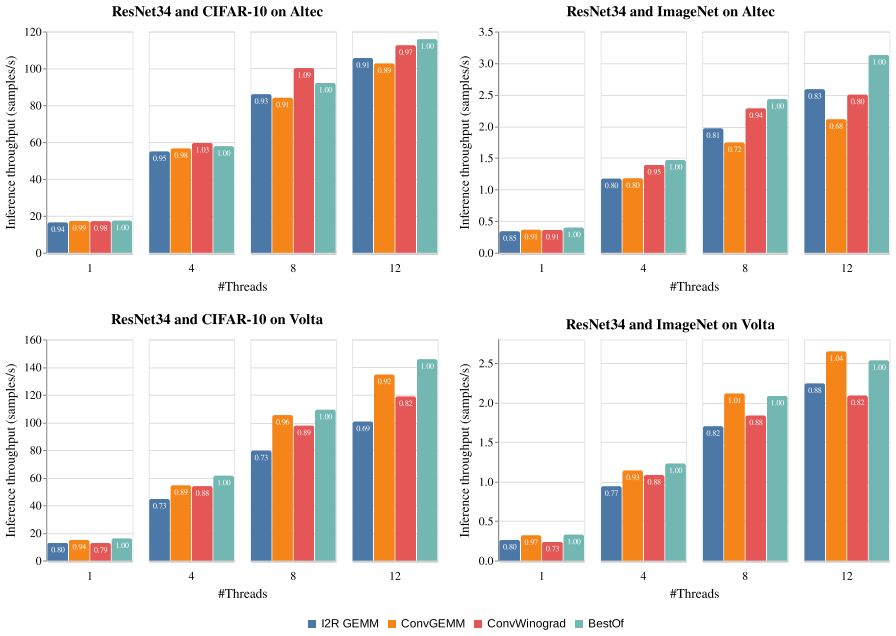

Fig. 5 Inference performance of ResNet34 on CIFAR-10 (left) and ImageNet (right) using ALTEC (top) and VOLTA (bottom) using different forward-backward alternatives and BestOf with the same alternatives

6 Concluding Remarks

In this work, we have presented BestOf, a novel online implementation selector, that is capable of selecting at run time among different alternatives the best performing one. Two important features of BestOf are the ability to evaluate groups of alternatives as a whole as well as making nested decisions.

The experimental results on the VGG16 and ResNet34 model demonstrate that our auto-tuner is able to improve the overall training and inference times when different algorithms are used to process the convolutional layers. We also observed that, when the preferred algorithm depends on the target architecture, BestOf easily identifies the best alternative, avoiding manual efforts to profile each available alternative. With this in mind, we can conclude that the benefits of the BestOf auto-tuner highly compensate for the negligible costs in terms of overheads and lines of code that have to be introduced in the original application.

As part of a future work, we plan to apply the auto-tuner to other fields in order to test its applicability and to find out which additional requirements should be incorporated. As part of this effort, we plan to implement the possibility of retrieving the decisions made in a previous execution so that BestOf could be useful even for short-lived applications.

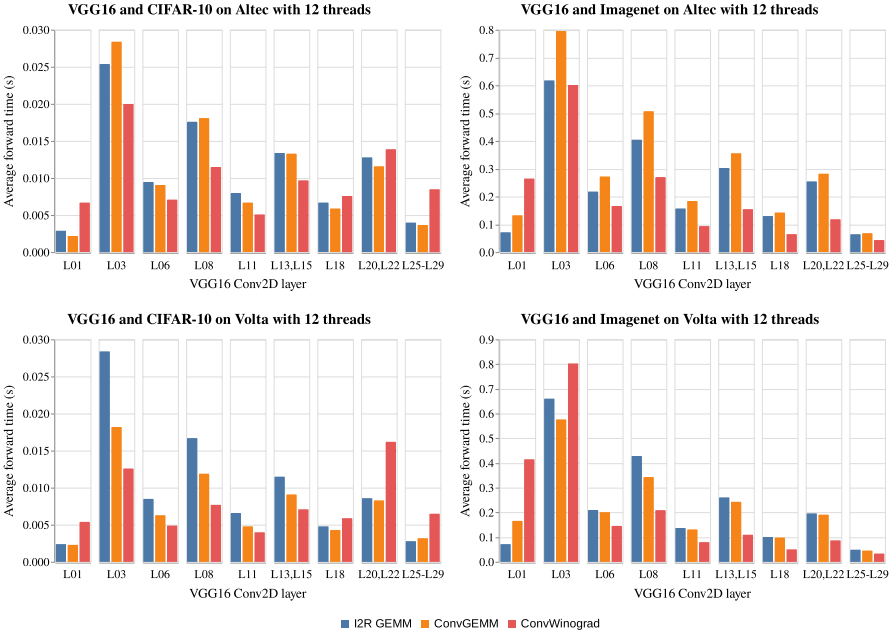

Fig. 6 Average time of the forward alternatives evaluated by BestOf on the convolutional layers of VGG16 during the inference on CIFAR-10 (left) and ImageNet (right) using ALTEC (top) and VOLTA (bottom) and 12 threads

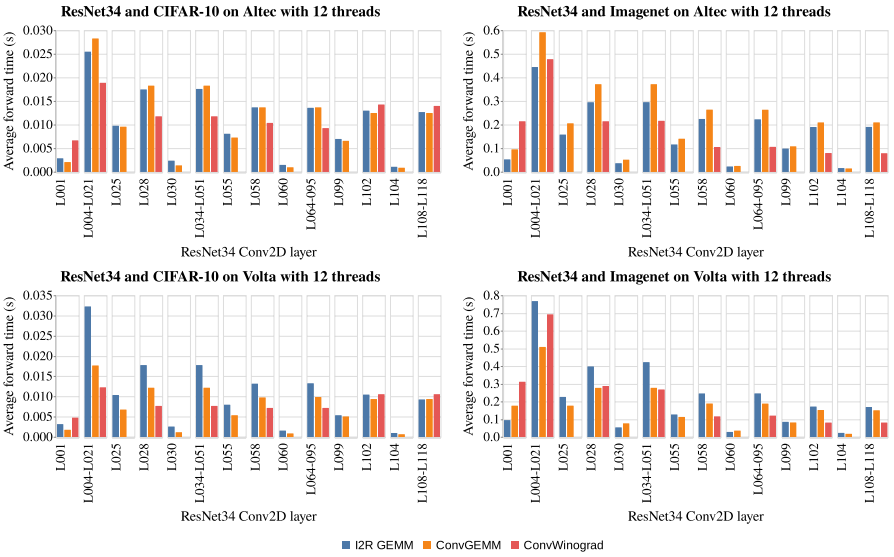

Fig. 7 Average time of the forward alternatives evaluated by BestOf on the convolutional layers of ResNet34 during the inference on CIFAR-10 (left) and ImageNet (right) using ALTEC (top) and VOLTA (bottom) and 12 threads. The Winograd algorithm column is blank on those layers where this algorithm can not be used

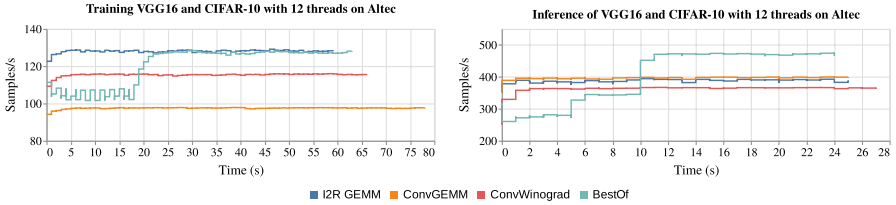

Fig. 8 Evolution of the performance for the training (left) and inference (right) of VGG16 and CIFAR-10 with 12 threads on Altec for the different algorithms evaluated and the BestOf alternatives

Acknowledgements This research was funded by Project PID2020-113656RB-C21/C22 supported by MCIN/AEI/10.13039/501100011033. Manuel F. Dolz was also supported by the Plan Gen–T grant CDEIGENT/2018/014 of the *Generalitat Valenciana*. Adrián Castelló is a FJC2019-039222-I fellow supported by MCIN/AEI/ 10.13039/501100011033.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

Data availibility The CIFAR-10 and ImageNet datasets used for the current study are publicly available from the web. See https://www.cs.toronto.edu/~kriz/cifar.html and https://www.image-net.org/, respectively.

References

1. Sze V, Chen Y-H, Yang T-J, Emer JS (2017) Efficient processing of deep neural networks: a tutorial and survey. Proc IEEE 105(12):2295–2329
2. Pouyanfar S et al (2018) A survey on deep learning: algorithms, techniques, and applications. ACM Comput Surv 51(5):92:1-92:36
3. Hssayni E, Joudar N-E, Ettaouil M (2022) KRR-CNN: kernels redundancy reduction in convolutional neural networks. Neural Comput Appl 34(3):2443–2454
4. Fernandes Junior FE, Yen GG (2019) Particle swarm optimization of deep neural networks architectures for image classification. Swarm Evol Comput 49:62–74
5. Eddine MD, Shen Y (2022) A deep learning based approach for predicting the demand of electric vehicle charge, J Supercomput
6. Jordà M, Valero-Lara P, Peña AJ (2019) Performance evaluation of cuDNN convolution algorithms on NVIDIA Volta GPUs. IEEE Access 7:70461–70473
7. Chen T, Zheng L, Yan E, Jiang Z, Moreau T, Ceze L, Guestrin C, Krishnamurthy A (2018) Learning to optimize tensor programs. In: Proceedings of the 32nd International Conference on Neural Information Processing Systems, Ser. NIPS'18. Curran Associates Inc., Red Hook, NY, USA, pp 3393–3404

8. Zheng L, Jia C, Sun M, Wu Z, Yu C. H, Haj-Ali A, Wang Y, Yang J, Zhuo D, Sen K et al (2020) Ansor: generating high-performance tensor programs for deep learning, In: 14th USENIX symposium on operating systems design and implementation (OSDI 20), pp 863–879
9. Barrachina S, Castelló A, Catalán M, Dolz MF, Mestre JI (2021) Pydtnn: a user-friendly and extensible framework for distributed deep learning. J Supercomput 77(9):9971–9987
10. Chellapilla K, Puri S, Simard P (2006) High performance convolutional neural networks for document processing," In: Tenth international workshop on frontiers in handwriting recognition
11. Juan PS, Castelló A, Dolz MF, Alonso-Jordá P, Quintana-Ortí ES (2020) High performance and portable convolution operators for multicore processors, In: 32nd IEEE international symposium on computer architecture and high performance computing, SBAC-PAD (2020) Porto, Portugal, September 9–11. IEEE 2020:91–98
12. Winograd S (1980) Arithmetic complexity of computations. Society for Industrial and Applied Mathematics
13. Low TM, Igual FD, Smith TM, Quintana-Ortí ES (2016) Analytical modeling is enough for high-performance BLIS. ACM Trans Math Soft (TOMS) 43(2):1–18
14. Whaley RC, Dongarra JJ (1998) Automatically tuned linear algebra software, In: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, Ser. SC '98. IEEE Computer Society, USA, pp 1–27
15. Dastgeer U, Li L, Kessler C (2013) Adaptive implementation selection in the SkePU skeleton programming library. In: Wu C, Cohen A (eds) Adv Parallel Process Technol. Springer, Heidelberg, pp 170–183
16. del Rio Astorga D, Dolz MF, Sánchez LM, Fernández J, García JD (2018) An adaptive offline implementation selector for heterogeneous parallel platforms. Int J High Perform Comput Appl 32(6):854–863
17. Anderson A, Gregg D (2018) Optimal dnn primitive selection with partitioned boolean quadratic programming, In: Proceedings of the 2018 International symposium on code generation and optimization, ser. CGO. New York, NY, USA: Association for Computing Machinery, 2018, pp 340–351. [Online]. Available: https://doi.org/10.1145/3168805
18. Fernández J, Cuadrado AS, del Rio Astorga D, Dolz MF, Daniel García J (2017) Probabilistic-based selection of alternate implementations for heterogeneous platforms, In: *Algorithms and Architectures for Parallel Processing*. Springer International Publishing, pp 749–758
19. Planas J, Badia RM, Ayguadé E, Labarta J (2013) Self-adaptive OmpSs tasks in heterogeneous environments," In: 2013 IEEE 27th international symposium on parallel and distributed processing, pp 138–149
20. Balaprakash P, Dongarra J, Gamblin T, Hall M, Hollingsworth JK, Norris B, Vuduc R (2018) Autotuning in high-performance computing applications. Proc IEEE 106(11):2068–2083
21. Ben-Nun T, Hoefler T (2019) Demystifying parallel and distributed deep learning: an in-depth concurrency analysis. ACM Comput Surv (CSUR) 52(4):1–43
22. Simonyan K, Zisserman A (2015) Very deep convolutional networks for large-scale image recognition, arXiv:1409.1556
23. He K, Zhang X, Ren S, Sun J (2016) Deep residual learning for image recognition, In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp 770–778