# On the Anatomy of Predictive Models for Accelerating GPU Convolution Kernels and Beyond

PAOLO SYLOS LABINI*, Free University of Bozen-Bolzano, Italy
MARCO CIANFRIGLIA, National Research Council of Italy, Italy
DAMIANO PERRI and OSVALDO GERVASI, University of Perugia, Italy
GRIGORI FURSIN, ctuning Foundation, France
ANTON LOKHMOTOV, Dividiti, United Kingdom
CEDRIC NUGTEREN, TomTom, Netherlands
BRUNO CARPENTIERI, Free University of Bozen-Bolzano, Italy
FABIANA ZOLLO, Ca' Foscari University of Venice, Italy
FLAVIO VELLA*, Free University of Bozen-Bolzano, Italy

Efficient HPC libraries often expose multiple tunable parameters, algorithmic implementations, or a combination of them, to provide optimized routines. The optimal parameters and algorithmic choices may depend on input properties such as the shapes of the matrices involved in the operation. Traditionally, these parameters are manually tuned or set by auto-tuners. In emerging applications such as deep learning, this approach is not effective across the wide range of inputs and architectures used in practice. In this work, we analyze different machine learning techniques and predictive models to accelerate the convolution operator and GEMM. Moreover, we address the problem of dataset generation, and we study the performance, accuracy, and generalization ability of the models. Our insights allow us to improve the performance of computationally expensive deep learning primitives on high-end GPUs as well as low-power embedded GPU architectures on three different libraries. Experimental results show significant improvement in the target applications from 50% up to 300% compared to auto-tuned and high-optimized vendor-based heuristics by using simple decision tree- and MLP-based models.

*Corresponding authors and main contributors.
This work is partially supported by UNIBZ-RTD-CALL2018-IN2087 Project and INdAM–GNCS Project 2020-NoRMA. Marco Cianfriglia was partially supported by H2020-ICT-2015-687689 Project within HiPEAC industrial internship initiative in dividiti Limited.
Authors' addresses: P. Sylos Labini, B. Carpentieri, and F. Vella, Free University of Bozen-Bolzano - Faculty of Computer Science, piazza Domenicani, 3, 39100, Bozen-Bolzano, Italy; emails: {paolo.syloslabini, bruno.carpentieri, flavio.vella}@unibz.it; M. Cianfriglia, National Research Council of Italy (CNR) - Institute for Applied Computing (IAC) "M. Picone", Via dei Taurini 19, 00185 Rome, Italy; D. Perri and O. Gervasi, Università degli Studi di Perugia - Dipartimento di Matematica e Informatica, Via Vanvitelli, 1, 06123 Perugia, Italy; emails: damiano.perri@unifi.it, osvaldo.gervasi@unipg.it; G. Fursin, Ctuning foundation, 5, rue Camille Desmoulins, 94230 Cachan, France; email: grigori.fursin@ctuning.org; A. Lokhmotov, dividiti Limited, ideaSpace West, 3 Charles Babbage Road, Cambridge, CB3 0GT, United Kingdom; email: anton@dividiti.com; C. Nugteren, TomTom, De Ruijterkade 154, 1011 AC Amsterdam, Netherlands; email: mail@cedricnugteren.nl; F. Zollo, Università Ca' Foscari Venezia, Via Torino 155, 30170, Venezia Mestre, Italy; email: fabiana.zollo@unive.it.
Author current address: D. Perri, Università degli studi di Firenze - Dipartimento di Matematica e Informatica 'Ulisse Dini', Viale Morgagni, 67/a, 50134 Firenze, Italy.


1544-3566/2020/01-ART16
https://doi.org/10.1145/3434402

CCS Concepts: • **General and reference** → **Performance**; • **Computer systems organization** → *Parallel architectures*; • **Computing methodologies** → *Modeling methodologies*; • **Software and its engineering** → **Software libraries and repositories**;

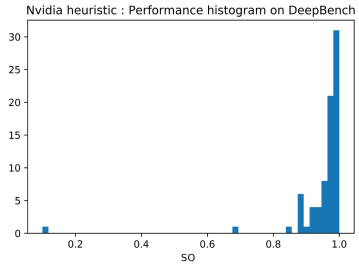
Additional Key Words and Phrases: GPU computing, predictive models, neural networks, performance optimization, supervised classification, tuning

**ACM Reference format:**
Paolo Sylos Labini, Marco Cianfriglia, Damiano Perri, Osvaldo Gervasi, Grigori Fursin, Anton Lokhmotov, Cedric Nugteren, Bruno Carpentieri, Fabiana Zollo, and Flavio Vella. 2020. On the Anatomy of Predictive Models for Accelerating GPU Convolution Kernels and Beyond. *ACM Trans. Archit. Code Optim.* 18, 1, Article 16 (January 2021), 24 pages.
https://doi.org/10.1145/3434402

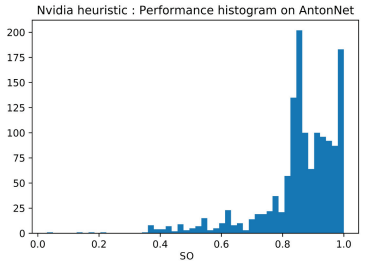
## 1 INTRODUCTION AND CHALLENGES

Scientific High-Performance Computing (HPC) applications are built on top of monolithic parallel routines that are often customized for a specific target architecture. With the advent of data-driven applications such as deep learning and graph analytics, the traditional library design has lost performance portability mainly because of the unpredictable size and structure of the data on the wide range of hardware available. The convolution operator, the core of a Convolutional Neural Network (CNN), is a notable example of how hard it is to determine the optimal method and implementation for a given input or layer [32]. Furthermore, CNN can be implemented in different ways by using direct convolution [75], Generic Matrix Multiplication (GEMM) [66], Winograd minimal filtering algorithm [35], or Fast Fourier Transformation (FFT) [51]. Regardless of the method adopted, the performance of each implementation can vary greatly even among the layers of a single CNN. For example, by using Winograd algorithm the number of multiplications passes from 36 to 16 for $(4 \times 4)$ and $(3 \times 3)$ filters. Winograd-based convolution can reduce the computational time by up to 4× [40]. However, the improvement obtained shows a discrepancy from the theory due to several implementation variables [61]. Another example would be in the context of Automated Machine Learning (AutoML) [70]. Here, the architecture of the Dynamic Neural Network (DNN) is not known *a priori*, and as a consequence from an engineering perspective, the performance is not predictable. Also GEMM, which is a widely used building block in convolution implementations, requires specific optimizations to scale-up over different input dimensions [24]. Several BLAS implementations for GEMM provide fast performance on a target architecture by assuming a fixed data size, layouts and structure (e.g., square matrices) [30, 46, 68]. However, such user-transparent implementations are selected by hand-written heuristics based on simple decision rules that are not able to generalize the wide range of data used in practice. For example, Nvidia heuristics selects the best algorithm (relative speed-up 0.97–1.0) in the 33% of the instances from DeepBench [43], as shown in Figure 1(a). On the contrary, changing the dataset, the performance decreases. The heuristics correctly chooses the best GEMM implementation in the 10% of the matrices only (see Figure 1(b)).

With the variety of parallel architectures available on the market ranging from traditional parallel processors to accelerators (e.g., GPUs) and system on chips (SoCs), several standards have been established to enable portability for heterogeneous architectures (i.e., OpenCL [59] and OpenACC [69]). However, developing generic and performance-portable code has become extremely challenging, especially from an algorithmic point of view. Auto-tuning techniques have partially mitigated the performance portability problem by adapting, for example, the underlying memory hierarchies and loop unrolling to a specific architecture. Within this context, a plethora of hardware-oblivious solutions have been developed [1, 26, 44, 53, 56]. However, auto-tuners seek




(a) DeepBench Dataset [43] (training).




(b) AntonNet Dataset. Collection of matrices from AlexNet [32], GoogLeNet [60] and SqueezeNet [29] (inference).

Fig. 1. Relative performance of Nvidia Heuristics against the possible best Nvidia GEMM algorithm on two different datasets, DeepBench (a) and AntonNet (b). The heuristic selects the best algorithm in the interval $[0.97, 1.0]$. SO (Speed-up over the Oracle) is a metric that considers the possible best performance of an ideal estimator (oracle) against the predictive model (see Section 3.3 for details).

the best configuration by assuming a specific instance as input. Thus, they can hardly achieve the best performance across all possible inputs. For example, GEMM routines are often *per-default* tuned for squared matrices [44, 53].

Recently, auto-code generation techniques have made it possible to write high-performance code on specific architectures automatically [8, 15, 61, 62, 65]. Several aspects might limit such approaches, which often require that the micro-architecture is exposed. Moreover, they generate highly-optimized code for a specific algorithm, so that in the presence of multiple algorithms the problem of selecting the best solution still remains. Finally, architectural changes require to re-encode the problem [15]. Over the last years, several studies have focused on the use of machine learning (ML) techniques to model performance [4, 57, 61] or to prune the search space of auto-tuners [21]. For example, hand-written decision rules can easily be replaced by machine learning techniques such as Decision Tree [16]. Many studies emphasize the performance achieved by these methods as black-box models, however several questions are still open and worth to be investigated. In particular, from a methodology perspective, there is no consensus in the community regarding how to model the problem by using supervised methods (e.g., classification and regression are two valid options) or other machine learning approaches (e.g., semi-supervised techniques). For example, the criteria for generating high-quality datasets have not been investigated yet. This aspect is fundamental for SoC where the generation of a large dataset to train the model may be expensive. Another aspect concerns the assessment of the model. In literature, well-established measures like accuracy usually reflect the quality of a model. However, the relation between accuracy and practical performance has yet to be investigated in depth.

Finally, from an application perspective, sophisticated deep learning compilers already adopt machine learning techniques for generating optimized code [9, 62]; however, they assume that the same implementation of convolution is used for all the convolution layers. Such an assumption does not guarantee the best performance across all the existing networks, or for those generated by AutoML. Figure 2 reports the performance of GEMM-based convolution against an ideal oracle that is able to select the fastest methods for each layer. We may notice that the static method is not efficient in most of the CNNs evaluated.

Motivated by these limitations, this article provides a deep analysis of predictive models to design efficient adaptive and input-aware libraries. The contribution of this article is twofold:

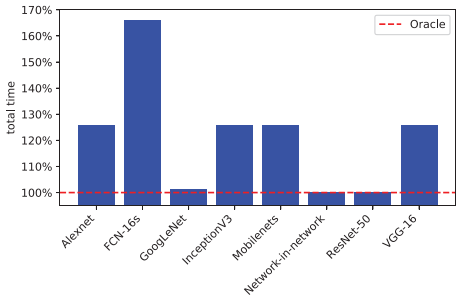



Fig. 2. Comparison of ARMCL GEMM-based convolution method against the performance of the best convolution methods per layer over popular CNNs.

- The study of several supervised classification techniques shows that models learned using Gradient Tree Boosting and Multi Layer Perceptron outperform other models in different settings. Our evaluation considers various aspects such as accuracy-based metrics, learning rate, generalization ability, and performance metrics in relation to dataset generation strategies;
- The implementation of learning-based heuristics to accelerate three libraries and two popular routines, convolution and GEMM, over three different GPU architectures, two high-end Nvidia GPUs (Pascal and Volta) and an embedded low-power ARM GPUs.

Specifically, we accelerate the convolution operator on the ARM Compute Library. Unlike well-established deep learning compilers, which generate optimized code for a specific convolution method (e.g., direct convolution), our framework is able to use a different convolution method for each layer at runtime. Second, by applying the same methodology on Nvidia cuBLAS, our model-based heuristics is two times better than Nvidia heuristics in selecting the best GEMM implementation by achieving an improvement of up to 2.2× over random sizes and up to 2.8× on matrices collected by popular CNNs. In addition, instead of using predictive models to infer parameters from the search space of traditional auto-tuners, we integrate them to select the best algorithm and related parameters when the input changes. We implement this solution by integrating the CLBlast library and several models that have been trained from the information provided by auto-tuners. We show that the model-based version of CLBlast outperforms the traditional auto-tuned version with up to 3× and 2.5× speed-ups on a high-end Nvidia GPU architecture and on an embedded ARM Mali GPU.

The article is organized as follows. Section 2 provides the background on supervised classification and the applications. Sections 3 and 4 describe our methodology, the use-cases, and their characteristics. Section 5 presents an exhaustive experimental evaluation. Section 6 provides a discussion of our results and sketches future works directions, while Section 7 reviews related work. Finally, Section 8 summarizes the contributions of this article and gives some concluding remarks.

## 2 NOTATION, BACKGROUND, CONCEPTS

The most important symbols employed in this article are summarized in Table 1. A classifier is a mapping from a feature space $I \subset \mathbb{R}^d$ to the set of labels $A$. Supervised classifiers learn this mapping from a dataset of known features-class pairings. The training, evaluation and comparison of classifiers are driven by an objective function that the classifiers aim to maximize. Normally, this function is the expected rate of correct classifications over the true distribution of the inputs—the (true) *accuracy*. Supervised methods are evaluated for their ability to recognize specific classes

Table 1. Symbols and Definitions Used in the Article

| | | |
|---|---|---|
| Machine Learning techniques | $DT$ | *Decision Tree* [55]. |
| | $RF$ | *Random Forest* [27] |
| | $GTB$ | *Gradient Tree Boosting* [73]. |
| | $NBC$ | *Naive Bayesian Classifier* [54]. |
| | $LoR$ | *Logistic Regression* [41]. |
| | $KNN$ | *K-Nearest Neighbours* [47]. |
| | $SVN$ | *Support Vector Machines* [13]. |
| | $MLP$ | *Multi Layer Perceptron* [48]. |
| Performance Measures | *Accuracy* | fraction of correctly predicted data points. |
| | *Precision* | fraction of data points correctly assigned to a class, over all points assigned to that class. |
| | *Recall* | fraction of data points correctly assigned to a class, over all points belonging to that class. |
| | *Balanced Accuracy* | the macro-average of recall over classes. |
| | $SO$ | Speed-up over the oracle. |
| | $SB$ | Speed-up over the baseline. |
| Other Symbols | $I$ | is the input space, $i \in I$ is a specific instance. |
| | $A$ | is the set of classes or labels assignable to instances $i$. $a \in A$ is a specific label. |
| | $D$ | is the training dataset, a collection of pairs $(i, a)$. |

using related measures such as *precision* and *recall*. There are various ways to summarize class-specific measures into a single global metric. Accuracy is one such measure; another is *balanced accuracy*, defined as the average of recall among classes and strictly related to *informedness* [52]. When one class is definitively more populated than the others, balanced accuracy provides a fairer and better-generalized estimation of the classifier performance than "simple" accuracy. Accuracy-like measures assume the cost of misclassification to be uniform, however in most practical applications different errors entail different costs. Although many research studies focus on cost-sensitive learning, they mostly concern class-dependent costs, where the cost of misclassification varies among classes but not within elements of the same class. In our setting, the cost of misclassification for each sample depends on the application. Focusing on the accuracy alone may be enough to obtain a good performance in terms of the objective function—the highest score (100% of accuracy) will always maximize the latter. However, we will see that there are other cases, such as GEMM, where almost optimal performance can be achieved despite low accuracy.

### 2.1 Supervised Classification

We will now provide a quick overview of the supervised techniques used in this article. The experienced reader may skip this part, or skim through it, and jump to Section 3. *Decision trees* (DT) [55] are a non-parametric supervised ML method used for classification and regression. DTs generate "white box" models, which are easily interpreted as `if-then-else` statements. However, small data perturbations might result in completely different trees being generated. To reduce variance and over-fitting, *Random Forest* (RF) [27] fits many independent trees on various subsets of the dataset. Another notable tree-based classifier is *Gradient Tree Boosting* (GTB) [73], which uses gradient descent to progressively grow a weighted ensemble of DTs. A *naive Bayesian classifier* (NBC) [54] is the simplest variant of a Bayesian network. A NBC assumes that all features are conditionally independent given the class variable. This assumption, while usually false, has proven to be incredibly effective in practice, producing good, simple models with little training. *Logistic Regression* (LoR) [41] is a simple form of regression analysis that assumes a linear relationship between the independent variables and the log-odds of the classes. *K-Nearest Neighbours* (KNN) [47] searches for the K data points that are nearest to the query feature vector and polls their assigned labels to decide that of the query vector. KNN is a non-generalizing method, which means that, instead of learning the parameters of a model, it just "remembers" the training points and eventually stores them in an appropriate data structure such as a Ball Tree for accelerating the inference.

*Support Vector Machine* (SVM) [13] separates data by drawing hyper-planes in the feature space. All the points that are closest to the hyper-planes are used as a reference to maximize the class separation. The well-known "kernel trick," which amounts to define alternative inner products for the data points, can be used to implicitly embed the problem in higher-dimensional spaces, allowing for complex separating surfaces. A *Multi Layer Perceptron* (MLP) [48] is the simplest feed-forward neural network. It consist of at least three layers of nodes: an input (features) layer, a hidden layer and an output (classes) layer. Except for the input nodes, each node uses a nonlinear activation function.

## 3 METHODOLOGY

In this section, we define the performance maximization problem as a classification task and briefly discuss the methodology for the training and evaluation of predictive models.

### 3.1 Performance as Classification Task

Performance modeling involves the maximization of the objective function over the input domain. One formulation consists in defining the maximization problem as a classification task. More precisely, one wants to select the best classes among multiple possibilities such that the objective function is maximized. Another option would be to predict the performance of each implementation and select either the best among them (ordinal rankings) or the best values of the tunable parameters. This approach would require the use of regression techniques, however classification in this case is advisable. Indeed, classification can handle arbitrarily structured solution spaces $A$, since it treats each solution as a black-box (see the GEMM use-case in Section 4.2). Moreover, regression may infer invalid parameters choice, or wrong values if the features selected for modeling the architecture are not relevant. For these reasons, in this work, we will focus on classification techniques. To this aim, we define the optimization problem as a classification task. Let $a$ be a solution to a particular problem and $A$ a collection of such solutions. Let $f_a : I \to \mathbb{R}$ be an objective function, where $I$ is the multidimensional input domain for $a$. The solution set $A$ may contain algorithms, collections of parameters, or any other implementation description. A classification task is thus defined as follows: for each $i \in I$, we identify the class label $\arg\max_{a \in A} f_a(i)$, i.e., the best solution for that input according to the objective function. A classifier will then consist of a mapping from input descriptions $i$ to solutions $a$.

### 3.2 Dataset Generation

Supervised techniques require datasets that pairs training points in $I$ to optimal solutions in $A$. To build a dataset, entries can be generated by fixing an input $i$ and benchmarking $f_a(i)$ for every solution $a$ on the target architecture. In this process, the size of the solution space (set of possible classes) plays a fundamental role. In the simplest scenario, the solution space includes few algorithm implementations without the need for searching optimal parameters. In this case, the classes are known in advance, and the only challenge is to find enough training points for each class. In other cases, the solution space $A$ is composed by several algorithms, each one having its own configuration parameters. For each algorithm, every possible combination of parameters can potentially be a class on its own. A classifier will only be aware of classes that appear at least once as the optimal solution of some instance in the training set. Unfortunately, this means that for complex problems the class space strongly depends on the choice of training points. Furthermore, classes that appear in the test set and not in the training set will be impossible to predict. Specific policies for dataset generation must thus be identified to address this issue. We will see later that a proper analysis of the learning curves may reveal helpful to determine if and how much the model would benefit, in terms of accuracy, from growing the training set.

### 3.3 Model Evaluation and Selection

In this article, we propose to train standard, cost-insensitive classifiers and then try to discriminate among them in the model selection phase. To assess the quality of the models, we evaluate *(i)* their ability to adapt to previously unseen data (generalization), *(ii)* the model performance w.r.t. the objective function, and *(iii)* the inference cost, which includes feature extraction and class selection.

*3.3.1 Generalization of the Model.* We determine the ability to model unseen data by studying the differences between training and test scores. Specifically, if both are low, the model will be underfitting. Overfitted models will instead present a high training score and a low validation score. When both the training and validation score are high, the model is working well.

*3.3.2 Model Performance.* The choice of the evaluation measure depends on the complexity of the task. For simple classification tasks, where the accuracy of some classifiers is close to 100%, accuracy-like metrics are effective in the prediction of the best model in terms of performance. In more complex tasks, where high accuracy is hard to achieve, near-optimal classes in terms of performance are more important regardless of the accuracy score. In these cases, studying the impact of misclassification is necessary to evaluate the model. We introduce two metrics to assess the quality of the heuristics based on the predictive model:

(1) **SO** (Speed-up over the Oracle), which considers the possible best performance of an ideal estimator (oracle) against the predictive model;
(2) **SB** (Speed-up over the Baseline), which takes into account the baseline performance of the original heuristics/baseline implementation.

In this way, we can measure how far the performance is from the possible best (SO) and the improvement over the baseline (SB).

*3.3.3 Inference Cost.* Finally, we consider the problem of *cost-effectiveness*, i.e., we want the cost of selecting the new routine to be lower than the performance improvement. In other words, we require that $f_{\overline{a}}(i) + c(i) < f_a(i)$, where $c(i)$ is the cost to select a new implementation $\overline{a}$ for the input $i$ by using a predictive model. In this work, we will show how the inference cost may weight on the selection of a specific model. However, for some applications, e.g., Deep Learning workloads, this cost can be amortized over multiple repetitions.

## 4 USE-CASES

Due to their importance in deep learning and scientific computing, we select two use-cases, CNNs and GEMM. Specifically, we formalize three different classification tasks. The first one aims to model the performance of convolution in the ARM Compute Library [3]. As for GEMM, we focus on two different libraries (cuBLAS [46] and CLBlast [44]).

### 4.1 Convolutional Neural Networks

CNNs are a class of deep, feed-forward artificial neural networks widely employed in image recognition. They are composed of a set of layers, each of them applying a set of learned convolution filters over the results (*activations*) of the previous layer. The final layer typically consists of a classifier, associated with a loss function to backpropagate gradients through the network and update the filter values (*weights*). The core of a CNN is the convolution operator, which transforms the input data by applying a set of filters (tipically consists of a classifier, associated with a loss function to backpropagate gradients through the network and update the filter values (*kernels*) to create a feature map for each channel of an image. Formally, given an image $\mathcal{I} \in R^{W \times H}$ and a

Table 2. Convolution: Symbols, Definitions, and Algorithms

| | | |
|---|---|---|
| Convolution description | $W$ | *Width.* |
| | $H$ | *Height.* |
| | $C_{in}$ | *Input channel.* |
| | $C_{out}$ | *Output channel.* |
| | $K$ | is a pair $(k \times k)$, which represents the *filter dimension.* |
| | $P$ | *Pad.* |
| | $S$ | *Stride.* |
| | $NP$ | *Numerical precision.* |
| Conv. Algo. | *directConv* | is the convolution method that implements Equation (1). |
| | *winograd* | is Winograd Filtering Algorithm [35]. |
| | *conv* | is GEMM with image to column convolution implementation [66]. |

kernel $K \in \mathcal{R}^{k \times k}$, the feature map $O \in R^{\hat{H} \times \hat{W}}$ is given by

$$F(\mathcal{I}, K) = \sum_{i=-k/2}^{i=k/2} \sum_{j=-k/2}^{j=k/2} \mathcal{I}(x+i, y+j) \times K(i,j). \tag{1}$$

As a first use-case, we consider the ARM Compute Library (ARMCL), which is an open-source collection of low-level routines for image processing and deep learning, supporting ARM CPU and GPU architectures. It provides basic arithmetic, mathematical and binary operators and CNN building blocks. Convolution is implemented using three different approaches: *(i)* direct convolution (*directConv*), which implements Equation (1); *(ii)* GEMM plus auxiliary algorithms (*conv*) [66]; *(iii)* Winograd filter algorithm (*winograd*) [35]. Depending on the ARM architecture, numeric precision and specific input tensor each method can exhibit different performance [38, 40, 49, 76].

*4.1.1 Dataset Generation.* To define the training dataset, we collected ~ 6,000 instances of synthetic convolution instances. Each $i$ is a tuple $(W, H, C_{in}, C_{out}, K, P, S, NP)$ where the parameters $W \ldots S$ characterize the operation of convolution, while $NP$ denotes the numerical precision:

- $W$ and $H$ can take the values 7, 128, or 256;
- $C_{in}$ ranges between 2 and 2,048 with a multiplicative factor of 2;
- $C_{out}$ ranges between 8 and 1,024 with a multiplicative factor of 2;
- $K$ ranges between 1 and 11 with an increment factor of 1;
- $NP$ can be 32 or 8 bits.

A detailed summary of the tensor description instance is provided in Table 2. To generate a dataset entry from a tuple $i$, we first created a convolution instance, randomly populating the matrix and kernels. Then, we ran the resulting convolution instance with each of the three methods and selected the fastest one as the correct classification label for those features.

*4.1.2 Model Training.* We trained the models using the ML techniques described in Section 2 for the maximization of the accuracy over the synthetic dataset. We used 10-fold cross-validation with balanced accuracy scoring to avoid biasing and overfitting. To prevent the selection of invalid classes, we used the standard convolution method *conv* whenever an inapplicable class is inferred.

### 4.2 GEMM

GEMM is a key component of many traditional scientific applications and extensively employed in deep learning and other ML algorithms. Moreover, GEMM is used in combination with the *image*

Table 3. Generic Matrix Multiplication: Symbols, Definitions, and Algorithms

| | | |
|---|---|---|
| GEMM | *M* | Number of the rows of the first operand and the output. |
| | *N* | Number of columns of the second operand and the output. |
| | *K* | Number of columns of the first operand and number of rows of the second operand. |
| CLBlast | *GEMM* | is a multi-kernel to accelerate larger matrices [42]. |
| | *GEMM Direct* | is a single generic kernel that handles all cases. |
| | *Threshold* | Decision rule based on matrix dimension for switching GEMM Direct to GEMM. |
| | *Default* | Tuned version for a specific matrix dimension. GEMM Direct is tuned for M=N=K=256, GEMM is tuned for M=N=K=1024. |
| | *Oracle* | Tuned version for each matrix dimension. Oracle always selects the best algorithm and related set of tunable parameters. |
| | *Model* | The heuristics selects an algorithm and the parameters based on a predictive model. |
| cuBLAS | *Default* | GEMM implementation selected by Nvidia heuristics. |
| | *CUBLAS_GEMM_ALGO* [0..23] | Nvidia GEMM Algorithms. |
| | *Oracle* | The heuristics always selects the best algorithm. |
| | *Model* | The heuristics selects one of 24 algorithm implementations according to a predictive model. |

*to column* routine in one of the most efficient convolution algorithms, i.e., *conv*. GEMM is usually defined as

$$C \leftarrow \alpha \cdot A \cdot B + \beta \cdot C \quad s.t. \quad A \in \mathbb{C}^{M \text{x} K}, B \in \mathbb{C}^{K \text{x} N}, C \in \mathbb{C}^{M \text{x} N}, \tag{2}$$

where $A$ and $B$ are the input matrices, $C$ is the output, and $\alpha$ and $\beta$ are constants. The operands $A$ and $B$ can be optionally transposed. Without loss of generality, we will set to 1 the values of $\alpha$ and $\beta$. A GEMM instance is then represented by the tuple $(M, N, K)$ describing the sizes of the involved matrices. The complexity is $O(M \cdot N \cdot K)$ [25]. As mentioned at the beginning of this section, we will focus on two different libraries, cuBLAS and CLBlast:

- Nvidia cuBLAS is a CUDA implementation of the standard basic linear algebra subroutines (BLAS), which includes 152 standard BLAS routines. It is designed to leverage Nvidia architecture capability, and supports single, double, complex, and double complex data types as well as succinct data types for deep learning. As for GEMM, cuBLAS provides 24 different kernels that can be explicitly selected. Nvidia heuristics automatically selects one of those implementations that are specifically tuned for sizes used in various neural networks [46].
- CLBlast is a modern, lightweight, fast and tunable OpenCL BLAS library. As for GEMM, CLBlast provides two kernels: a "direct" kernel covering all GEMM use-cases, and an "indirect" kernel making several assumptions about the layout and sizes of the matrices. The "indirect" kernel cannot be used on its own and requires several helper kernels to pad and/or transpose matrices to meet these assumptions (for more details, see References [44, 45]).

A summary of the terminology used for GEMM, cuBLAS, and CLBlast is reported in Table 3.

*4.2.1 Dataset Generation.* To create the dataset, we benchmarked cuBLAS and CLBlast and kept a record of the Floating Point Operation per Second (FLOPS) of GEMM routines over different matrices described by $(M, N, K)$ triples. In particular, we generated three different datasets, one from real-world shapes and two from synthetic shapes:

(1) As real-world dataset, we gathered the shapes of GEMM operands involved in popular Deep Neural Networks. *AntonNet* is a friendly name of the shapes collected from AlexNet [32], GoogLeNet [60], and SqueezeNet [29] by using batch sizes ranging from

Table 4. Description of the Classification Tasks

| Task/Application | Problem | Features | Classes |
|---|---|---|---|
| Convolution | Algorithm selection | 8 | 2–3 |
| cuBLAS GEMM | Algorithm selection | 3 | 24 |
| CLBlast GEMM | Algorithm and parameters selection | 3 | 28–82 |

The number of classes for CLBlast GEMM depends on the benchmark and dataset strategy generation on a specific GPU architectures.

2 to 128 with a step of 2. AntonNet consists of 1,377 different triples, which mostly represent rectangular shapes.

(2) *Grid of two* (*go2*) is composed by $(M, N, K)$ triples where each value ranges from 256 to 8,192 with a step of 256. This dataset involves more than 32,000 different synthetic shapes.

(3) *Power of two* (*po2*) consists of triples where the values are powers of 2 ranging from 64 to 8,192.

In the cuBLAS case, to determine an entry of the dataset we ran all 24 implementations of GEMM and kept the best one in terms of perfomance. For the CLBlast library, we used smaller datasets because of the large search space of tunable parameters. In this case, to define an entry of the dataset, we had to run the tuner of both kernels (GEMM and GEMM direct) for each $(M, N, K)$ tuple. We report all the details in Tables 8 and 9. To implement GEMM, we applied the approach provided by the CLTune tuner [45] for finding the best configuration. Notice that the number of classes appearing in each dataset may vary and strongly depends on the architecture as well as on the dataset generation strategy.

*4.2.2 Models Training.* In this case, the number of classes is larger than the number of features. For example, the number of classes for the Nvidia cuBLAS is 24, the features are only three. Starting from this observation, we applied the following approach:

- **cuBLAS:** We trained the models using the ML techniques described in Section 2 for maximizing the accuracy over the synthetic *go2* dataset.
- **CLBlast:** We selected the simplest ML method, Decision Tree (DT), whose results can be analyzed easily. This choice is also justified by the fact that classifiers based on DT show better performance even in the presence of low accuracy. Specifically, we trained multiple DTs by varying parameters $L$ and $H$, where $L$ is the minimum number of sample required for a class to be a leaf node, and $H$ is the maximum height of the decision tree. Notice that larger values of $L$ will result in smoother decision trees. If $H = +\infty$, then nodes are expanded either until all the leaves are pure (i.e., all the value of the feature in the node comes from a single class) or until they contain less than $2L$ samples. In our analysis, $H \in \mathcal{H} = \{1, 2, 4, 8, +\infty\}$ and $L \in \mathcal{L} = \{1, 2, 4, 0.1, 0.2, 0.4, 0.5\}$, where $L < 1$ indicates a corresponding fraction of the total data points. Our framework auto-generated a C++ source code that implements DT models as an `if-then-else` statement and compiles them into CLBlast.

Table 4 summarises the description of the classification tasks for both convolution and GEMM.

## 5 EXPERIMENTAL RESULTS

To test the effectiveness of our approach, we carried out an exhaustive experimental analysis using three different GPU architectures: an Nvidia Tesla P100, an Nvidia Titan V, and an embedded ARM Mali-T860 based on the Midgard architecture. To collect and automatize the benchmarks, we used the Collective Knowledge framework [22]. For every benchmark, we used from 5 to 10 repetitions

Table 5. Classification Results for Different Models, *Conv*, *Directconv*, and *Winogradconv*, from 10-fold Cross-validation on the Synthetic Dataset

| | accuracy | balanced accuracy | conv | | directconv | | winogradconv | |
|---|---|---|---|---|---|---|---|---|
| | | | precision | recall | precision | recall | precision | recall |
| RF | 0.93 ± 0.005 | 0.94 ± 0.005 | 0.98 ± 0.005 | 0.93 ± 0.01 | 0.85 ± 0.02 | 0.92 ± 0.01 | 0.83 ± 0.03 | 0.99 ± 0.01 |
| MLP | 0.89 ± 0.01 | 0.85 ± 0.05 | 0.92 ± 0.02 | 0.93 ± 0.01 | 0.82 ± 0.03 | 0.80 ± 0.05 | 0.82 ± 0.085 | 0.82 ± 0.1 |
| DT | 0.91 ± 0.01 | 0.93 ± 0.01 | 0.98 ± 0.01 | 0.92 ± 0.01 | 0.83 ± 0.02 | 0.89 ± 0.025 | 0.81 ± 0.07 | 0.98 ± 0.015 |
| LoR | 0.57 ± 0.01 | 0.70 ± 0.01 | 0.90 ± 0.015 | 0.50 ± 0.015 | 0.53 ± 0.025 | 0.64 ± 0.035 | 0.22 ± 0.015 | 0.98 ± 0.02 |
| KNN | 0.72 ± 0.2 | 0.50 ± 0.01 | 0.80 ± 0.01 | 0.84 ± 0.03 | 0.65 ± 0.045 | 0.62 ± 0.025 | 0.05 ± 0.025 | 0.05 ± 0.025 |
| NBC | 0.71 ± 0.15 | 0.74 ± 0.01 | 0.85 ± 0.015 | 0.80 ± 0.015 | 0.59 ± 0.02 | 0.41 ± 0.025 | 0.37 ± 0.03 | 0.99 ± 0.05 |
| SVM | 0.72 ± 0.15 | 0.69 ± 0.02 | 0.90 ± 0.01 | 0.72 ± 0.02 | 0.64 ± 0.03 | 0.76 ± 0.02 | 0.27 ± 0.025 | 0.59 ± 0.055 |
| GTB | 0.94 ± 0.05 | 0.94 ± 0.01 | 0.96 ± 0.005 | 0.97 ± 0.005 | 0.90 ± 0.02 | 0.89 ± 0.01 | 0.93 ± 0.03 | 0.96 ± 0.025 |

and collected the average time. For the ARM Mali GPUs, to avoid DVFS effect, we set an idle time (2 s) between two consecutive runs. For each use-case presented in Section 4—i.e., Convolution, cuBLAS, and CLBlast—we performed the training and evaluation of models in the following way. The synthetic dataset was partitioned in training, validation and test-sets to train, tune hyper-parameters and to perform the evaluation of each model, respectively. Learning curves, and in general, the other evaluation metrics were estimated through 10-fold cross validation. To evaluate the models on test-sets, we removed from the training and validation all entries appearing in the test-sets.

Our analysis will investigate the following aspects:

(1) The quality of the models learned from different machine learning techniques in terms of generalization, accuracy, performance, and inference cost;
(2) The impact of misclassification on performance;
(3) The performance on real use-cases.

### 5.1 Convolution

We compared the performance of several models against the oracle, which always selects the fastest convolution method. The ARMCL does not adopt heuristic for selecting the convolution method layer-by-layer. Thus, we report the performance of the default ARMCL method, the GEMM-based convolution *conv*, for the comparison with our method. We tested our models on hold-out sets from the synthetic dataset and on a collection of 238 convolutional layers (*aggregate* test-set) from several popular CNNs, such as Alexnet, FCN-16s, ResNet-50, VGG-16s, GoogLeNet, and InceptionV3.

*5.1.1 Models Evaluation.* To learn our predictive models, we applied the ML techiniques presented in Section 2 over the synthetic dataset. The learning curves of each model are shown in Figure 3. We may observe that all models converge and provide both good generalization and good balanced accuracy, the only exceptions being KNN, which suffers from overfitting, and SVM, which would require more training points to converge. Decision tree-based estimators showed superior performance overall (Figures 3(a)–3(c)). For each model, Table 5 shows the classification results in terms of accuracy, balanced accuracy, precision and recall. The performance of the models is measured by its balanced accuracy and its total computation time, and is shown in Figure 4. As for performance, tree-based models (**DT**, **RF**, and **GTB**) and **MLP** obtained the best accuracy and balanced accuracy on unseen data; they also achieved the highest *precision* and *recall* scores

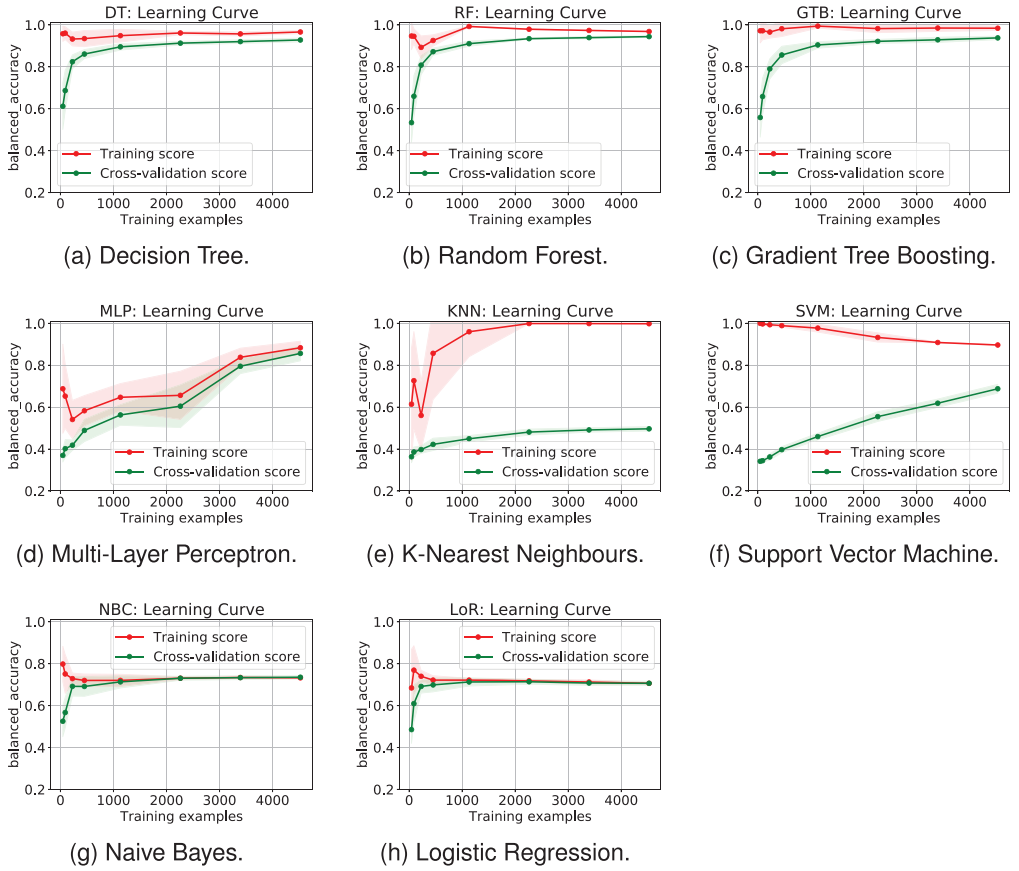



Fig. 3. Learning curves over a training dataset of 5,657 data points.

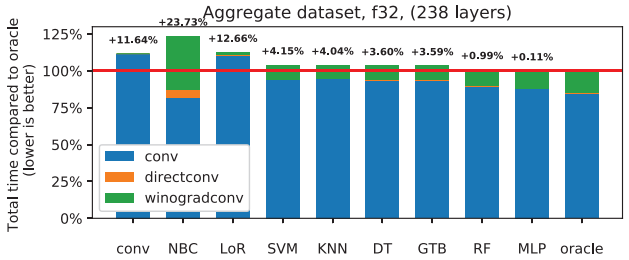



Fig. 4. Performance of the learned models and of the fixed "conv" method against the oracle for the aggregate test-set with f32 precision. Colors denote the fraction of time spent on each method.

on the *directconv* class, contrary to other models (see Table 5). The performance of these models on the aggregate test set is close to the oracle.

**SVM** also showed a good performance improvement against the fixed methods, using only 4% more time than the oracle, against the +11.5% of the fixed *conv* method. **LoR**, **NBC**, and **polynomial-kernel SVM** (not shown) achieved poor performance and accuracy, proving that no simple cut exists in the feature space to adequately separate the classes. **KNN** obtained better performance than static methods, however it achieved a low *balanced accuracy* score due to

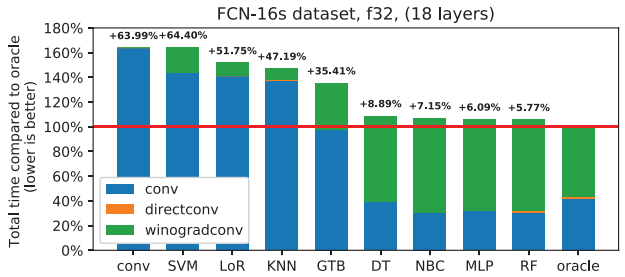



Fig. 5. Performance of the learned models against the static "conv" method and the oracle for the FCN-16s test set (f32 precision).

Table 6. Speed-up Over the Fixed "Conv" Method and Slow Down Over the Oracle for the Best Model in Each CNN

| CNN | SPEED-UP AGAINST 'CONV' | SLOW-DOWN AGAINST ORACLE | BEST MODEL |
|---|---|---|---|
| **Alexnet** | 1.19× | +0.00% | RF |
| **FCN-16s** | 1.55× | +5.77% | RF |
| **GoogLeNet** | 1.01× | +0.05% | MLP |
| **InceptionV3** | 1.25× | +0.00% | DT |
| **Mobilenet** | 1.00× | +0.00% | DT |
| **NiN** | 1.00× | +0.00% | DT |
| **ResNet-50** | 1.04× | +0.00% | DT |
| **VGG-16** | 1.44× | +0.00% | RF |

Ties between models are broken by inference time.

its inability to identify *winogradconv* instances. As for popular CNNs, Figure 5 shows the performance of the models on FCN-16s. The RF model outperforms the others by correctly recognizing the *directconv* instances. Again, the motivation is behind the learning curve: RF performs better than others in data generalization. This demonstrates that this model-driven approach can achieve even higher speed-ups in more balanced test sets. This is also observed by analyzing the results on other CNNs, as reported in Table 6. Since the relative proportion of classes varies considerably among CNNs, the *balanced accuracy* score reported in Table 5 is a better performance indicator than simple accuracy.

*5.1.2 Inference Overhead.* To conclude our analysis, we compared the inference time of the different models, i.e., the time occurring between the feature extraction (the input of the problem in our case) and the selection of the convolution method. We use as a baseline the cost of traversing the `if-then-else` rules of the DTs, which have the lowest inference cost overall. All the models show up to a 7.5× (KNN) of overhead with the exception of RF, for which the overhead cost depends on the number of trees. They provided the best performance (balanced accuracy, generalization, SO) when using around 300 trees, which makes their inference cost significant (261×). This cost can be reduced by cutting the number of trees, possibly down to the still good performance of a single DT. This allows for an easily adjustable trade-off between accuracy on hand, and training and inference costs on the other. The optimal trade-off between precision and inference overhead depends on the circumstances. Our study on inference time for GEMM revealed that the overhead of an efficiently implemented DT amounts to less than 1% of the computation time of the average GEMM multiplication. Taking into account the training cost, inference cost and

Table 7. Relative Importance of Features in the Convolution Classification Task, Extracted from a Trained Random Forest

| **FEATURE** | K | C-IN | C-OUT | W + H | NP | P | S |
|---|---|---|---|---|---|---|---|
| **IMPORTANCE** | 37% | 20% | 17% | 13% | 11% | 2% | 0% |

performance, DTs, RF, and GTB are the best models for this task. Moreover, these models allow for an easy-to-interpret feature analysis. In a DT, the *importance* of a feature is the normalized total reduction of the splitting criterion (in our case, the entropy) due to that feature. Ranking features by their average *importance* in the RF model revealed that the kernel size has the greatest impact on classification, followed by the number of input and output channels, as summarized in Table 7. This information can be useful when generating or expanding a training set to guide the choice and density of training points throughout the feature space.

### 5.2 cuBLAS

The classification task in this case is more complex than the convolution case. Thus, we expect to observe lower performance and accuracy and slower convergence of the learning curves. To measure the quality of our models, we use the *oracle* and the *Nvidia Heuristics* performance. Moreover, the metrics introduced in Section 3.3 allow us to measure how far the performance is from the possible best (SO), and the improvement over the baseline (SB). They are thus especially useful when the model is not accurate.

*5.2.1 Models Evaluation.* The learning curves are shown in Figure 6 and report the training and cross-validation scores over 7,000 data points of the *go2* dataset. The final models have been trained over around 10,000 points. As expected, they do not achieve high accuracy even in the case of larger datasets, because the number of classes is too large in relation to the number of features. Moreover, most of the models converge towards low accuracy on predicting new data, and the distance between training and cross-validation curves is low. The model learned by KNN suffers from underfitting. In this case, the scores increase when the traning set grows, however there is still no convergence. From an accuracy and generalization perspective, GTB (Figure 6(c)) and MLP (Figure 6(d)) are worth considering (0.7 of accuracy). Simple DTs can learn better models by using datasets composed by 10,000 up to 30,000 points. The cost of generating such a dataset require around four days on Nvidia Volta GPU. Therefore, the improvement in terms of accuracy does not justify this cost.

*5.2.2 Benchmark.* The classification results presented so far suggest to avoid the use of predictive models in the case of cuBLAS. Nonetheless, if we look at the performance curves in terms of SO, then the conclusion is different. Figure 7 reports the SO score of the three best models over 8,000 points. In general, all the models beat Nvidia heuristics (green line) with a speed-up from 2% up to 10% on average. This is due to the fact that, even if in nearly 30% of the cases the models do not predict the best algorithm, they are able to select a second or third-best algorithm with comparable performance. More significant average improvements are hard to achieve, because on average the worst implementation is 60% as efficient as the best one. Notice that the performance curves converge faster than learning curves. This means that we can learn a good model by using a smaller dataset. As an example, the analysis of GTB shows that the convergence between the training and cross-validation curves is reached after 2,500 points only, in contrast with the accuracy curve that converges at around 6,000 points. Motivated by the performance analysis carried out in Section 1, we compared the performance of Nvidia heuristics on two different datasets, DeepBench and AntonNet. While the performance on the first was satisfactory, the heuristics designed

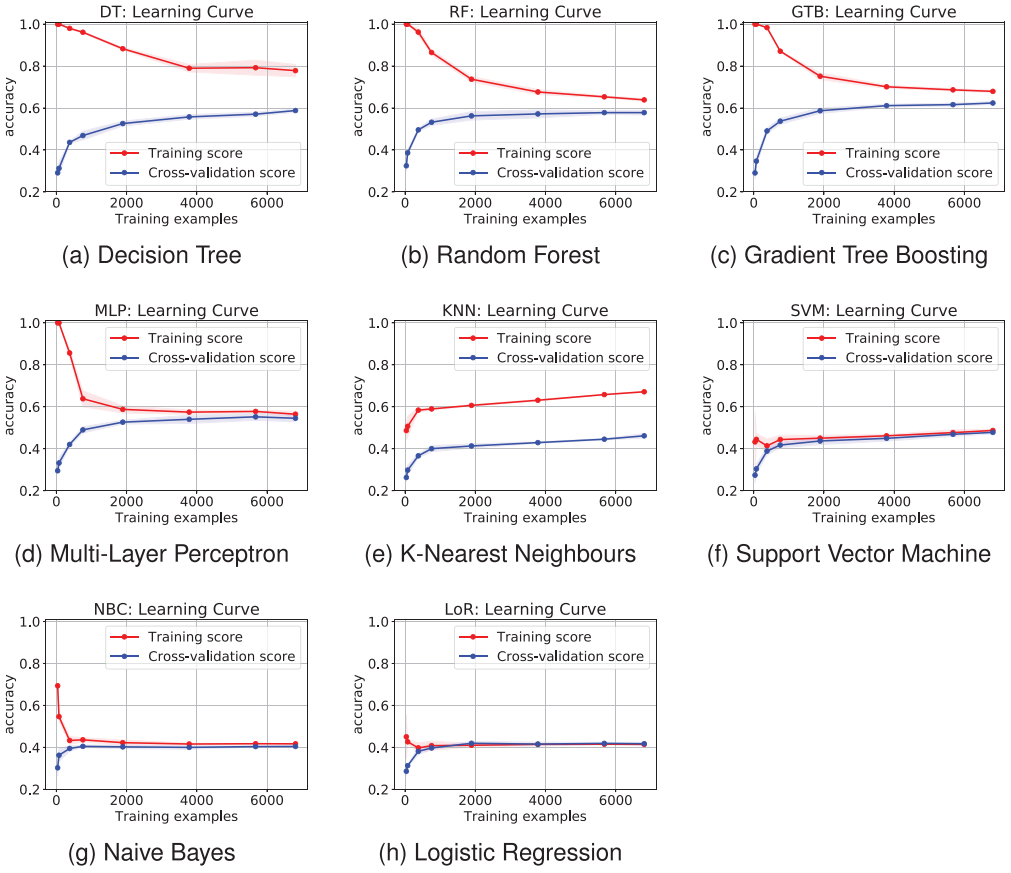



Fig. 6. Learning curves over 7,000 training points in the Nvidia cuBLAS classification task.

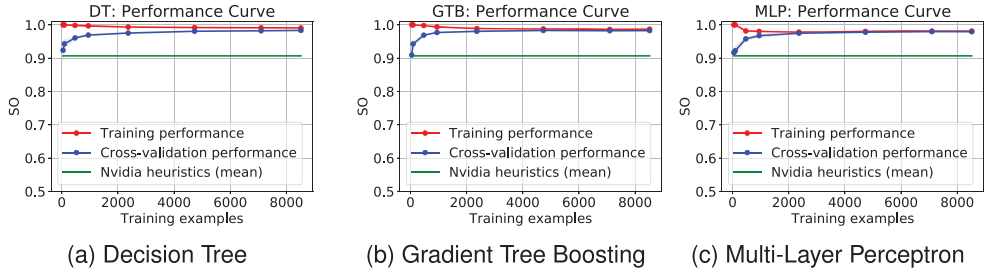
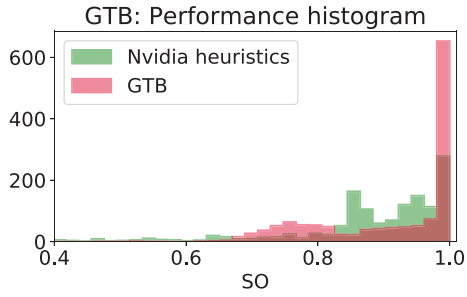
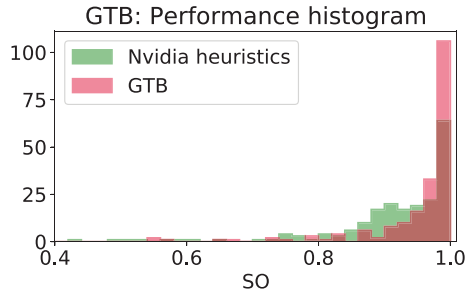



Fig. 7. Performance curves of the best three models over 8,000 points in the Nvidia cuBLAS classification task.

by Nvidia is not able to model the AntonNet dataset properly. The histogram in Figure 8(a) compares GTB-based heuristics against Nvidia heuristics in relation to the oracle. The y-axis shows the number of points where a certain SO (x-axis) is achieved. The red shadows show the overlap between the predictive model and Nvidia heuristics. Our model-based heuristics selects the best implementation (SO 0.9–1.0) using 50% of the points (>2 times better than Nvidia heuristics performance and up to 2.8× on specific instances in terms of FLOPS). A similar improvement can be observed on random instances. Figure 8(b) shows a better performance of the GTB model with




(a) AntonNet Dataset. Nvidia Heuristics SO: 0.87. Model-based heuristics SO: 0.91




(b) Rand300 Dataset. Nvidia Heuristics SO: 0.91. Model-based heuristics SO: 0.95.

Fig. 8. Relative performance of Nvidia heuristics and model-based (GTB) heuristics against the oracle.

Table 8. Datasets Statistics—Nvidia P100

| Dataset Name | Dataset Size | No. Unique Config. Xgemm | No. Unique Config. XgemmDirect | Total No. Classes | Best DT Name | Best DT accuracy | Best DT SO | Best DT SB |
|---|---|---|---|---|---|---|---|---|
| AntonNet | 456 | 1 | 81 | 82 | h4-L1 | 36 | 0.484 | 1.013 |
| PowerOf2(po2) | 216 | 2 | 41 | 43 | hMax-L1 | 21 | 0.431 | 0.931 |
| GridOf2(go2) | 3375 | 6 | 22 | 28 | hMax-L1 | 60 | 0.852 | 1.424 |

"Best DT" indicates the Decision Tree model with the highest SO score.

Table 9. Dataset Statistics—ARM Mali-T860

| Dataset Name | Dataset Size | No. Unique Config. Xgemm | No. Unique Config. XgemmDirect | Total No. Classes | Best DT Name | Best DT accuracy | Best DT SO | Best DT SB |
|---|---|---|---|---|---|---|---|---|
| AntonNet | 456 | 28 | 35 | 63 | h1-L0.1 | 55 | 0.702 | 1.092 |
| PowerOf2(po2) | 216 | 29 | 1 | 30 | h8-L0.1 | 45 | 0.551 | 1.121 |
| GridOf2(go2) | 1000 | 32 | 3 | 35 | h8-L0.4 | 53 | 0.803 | 1.479 |

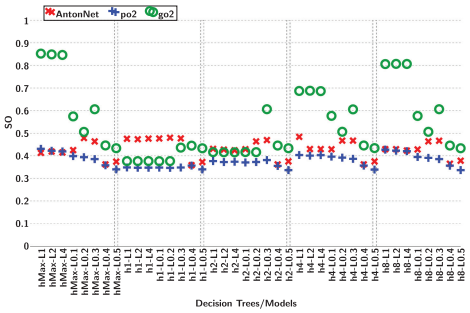
"Best DT" indicates the Decision Tree model with the highest SO score.

respect to Nvidia heuristics (up to 2.2× on single instances) over 300 randomly selected matrices. Thus, DT-based classifiers may allow us to obtain learning models that can reach very good performances despite their accuracy. To achieve a greater speed-up on single instances, the model-based approach requires further highly optimized codes designed for specific instances.

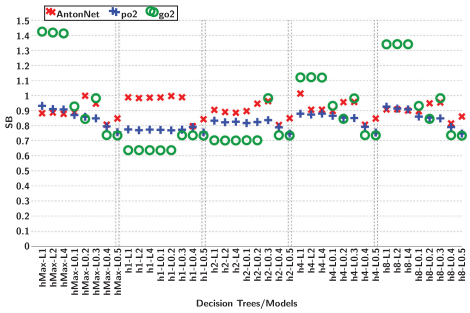
### 5.3 CLBlast GEMM

We have seen so far that DT-based models achieve practical good performance. In this section, we will investigate the impact of hyper-parameter tuning in the CLBlast scenario, where the number of classes is larger than for cuBLAS (up to 82 different classes, see Tables 8 and 9). Moreover, we want to understand the practical implication of misclassification. To this aim, we trained several DTs on the datasets described in Section 4, varying parameters $L$ and $H$. In the following, we refer to the definitions reported in Table 3.

*5.3.1 Models Evaluation.* By varying $L$ and $H$, we observe a huge accuracy variation in the range between 50% and 70% for *go2* on Nvidia 100. More specifically, models trained on the denser *go2* have a much higher accuracy than those trained on *po2* and *AntonNet*. The ARM architec-

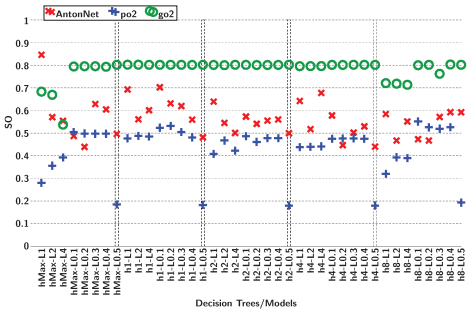



(a) Average performance speed-up between the model-driven and the Oracle (peak of the tuner) of CLBlast (**SO**).

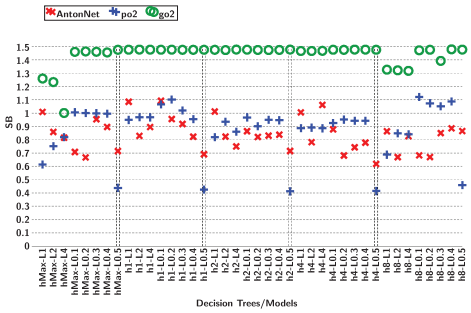



(b) Average performance ratio between the model-driven and the tuned version of CLBlast with CLTune (**SB**).

Fig. 9. Evaluation of the impact of misclassification for models generated by varying $H$ and $L$ parameters on *go2*, *po2* and *AntonNet* on Nvidia P100.



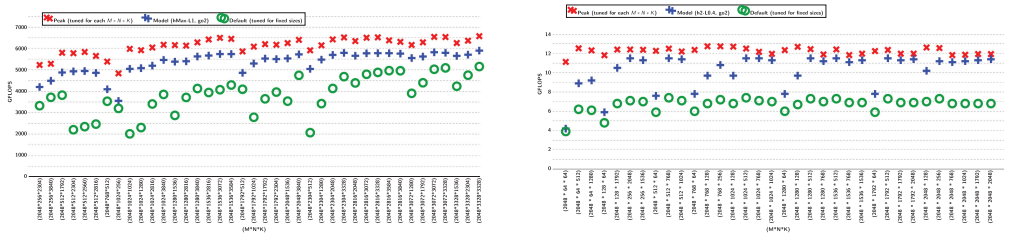

(a) Average performance speed-up between the model-driven and the Oracle (peak of the tuner) of CLBlast (**SO**).




(b) Average performance ratio between the model-driven and the tuned version of CLBlast with CLTune (**SB**).

Fig. 10. Evaluation of the impact of misclassification for models generated by varying $H$ and $L$ parameters on *go2*, *po2* and *AntonNet* on ARM Mali-T860.

ture shows a slightly different trend, with all the models performing similarly, and an accuracy up to 60%. Our results indicate that $H$ does not impact the accuracy significantly, while higher values of $L$ marginally impair it. Experiments on the accuracy show that its value decreases in the presence of an unbalanced distribution of unique configurations among kernels (see Tables 8 and 9). Figures 9 and 10 report SO and SB values for each model on both architectures. These metrics depend more strongly than accuracy on the models parameters, especially on $L$. Figure 9(a) shows that models trained on *go2* achieve the highest scores (SO $\geq$ 0.7). Such models outperform the tuned version of CLBlast (see Figure 9(b)). For the ARM architecture, the scenario is similar (Figure 10). Despite the *go2* models exhibit an accuracy compared to the models trained from *po2* and *AntonNet*, they achieve better performance in terms of SB and SO metrics. Indeed, the best *go2* model is 1.4x faster than the traditional tuned version (SB) on both architectures. Furthermore, the misclassification represents another important issue in this case: for the matrices in *AntonNet* and *po2*, the configurations learned are very specific and different from each other. In the presence of low accuracy, models that have been trained from denser and regular datasets, where the configurations are similar to each other, perform better. To validate that, we analyzed the distance-based



(a) Dataset: *go2*. Model: *hMax-L1*. Architecture: Nvidia P100

(b) Dataset: *go2*. Model: *h8-L0.1*. Architecture: ARM Mali-T860

Fig. 11. Performance on single instances for the best model trained on *go2*.

similarity [37] inside the datasets, defined as the average geometric distance between all tuples of tuning parameters in the dataset. Since each configuration has different parameters, we normalize their numerical domains. For both architectures, *go2* has the lowest average distance between all the pairs. Thus, even in the presence of misclassification, the model is likely to select a configuration that is not too far from the possible best. To get stronger evidence of the correlation between the similarity of the dataset and the performance of the model, we also analyzed the leaves of DTs. We observed that, in some cases, the best configuration is not a leaf of the decision tree even if it belongs to the training set. When this is the case, the misclassification is due to the accuracy of the ML technique used. By analyzing the best model learned from *go2* on Nvidia, we observed that the best parameters configuration is usually closer to the one chosen by the model than to the default one. For such cases, the model-based CLBlast outperforms the approach based on the auto-tuner. When the misclassification introduces a performance penalty, the selected configuration shows different values for either the size of the matrix tile, or the number of elements for the loop unrolling. Such parameters are usually similar for the best model learned from *go2*, while they show a big variation on *AntonNet*-based models. A very small gap in terms of performance has been noticed when the stride or the value of the padding changes.

*5.3.2 Benchmark.* In Figure 11, we report the performance on single GEMM instances for the best performing *go2* model on both architectures. On the Nvidia P100, the model *hMax-L1* learned from *go2* achieves very good performance in most instances, with a maximum speed-up of 3× and an average of 1.42× over the traditional tuned CLBlast (Figure 11(a) and Table 8). The same behaviour can be observed on the ARM architecture, where the model *h8-L0.1* achieves a maximum speed-up of 2× and an average of 1.47× (Figure 11(b) and Table 9). The improvement is greater for those matrix shapes that are far from the default sizes. On the contrary, the best model trained from *po2* does not guarantee satisfactory performance on average. For both the architectures, the models that have been trained from *AntonNet* showed unsatisfactory performance (poor generalization of the model).

*5.3.3 Inference Overhead.* On the Nvidia P100, we analyzed *hMax-L1* model trained on *go2* (which has 1,200 leaves and a depth of 19), measuring its inference times over all the matrices in the test dataset. Traversing the tree of if-else statements introduces less than 2% of overhead on small matrices. The relative overhead definitively decreases as the size of the matrices grows, with an average impact of less than 1% on performance. We observed similar results on the ARM system.

## 6 DISCUSSION AND FUTURE WORK

The adoption of predictive models learned through machine learning techniques is intuitive and may prove to be appealing for different applications, e.g., improving the performance of modeling

tasks. However, determining the right conditions for an appropriate use of such a methodology requires a careful analysis of the application and, in our case, of the target libraries. Along our experiments and the lessons learned thereby, in the following, we discuss the good practices for applying predictive models. We should focus on four main aspects:

(1) *The relation between number of features and number of classes to predict.* Our experiments show that the number of features and the structure of the class space must be taken into account both during the dataset generation and the model selection phase. In simple classification tasks (convolution), where the few possible classes are easily inferred from the features, a high accuracy ensures good performances without the need for further analysis. As the class space becomes more complex (cuBLAS), accuracy and other standard classification metrics are less effective in evaluating the models. High performance may be achieved with relatively low accuracy (60%), and model selection should thus be guided by performance measures.
(2) *Considerations on the dataset generation strategy.* Parametric class spaces (CLBlast) pose the greatest challenge. Here, the dataset generation strategy is of vital importance, as the choice of training points influences the number of classes and their similarity. We explored several strategies for the dataset generation, showing that a regularly spaced dataset is more effective than irregular datasets, even when the latter are collected from real use-cases (AntonNet) and thus better match the test point distribution. Also, an in-depth analysis of models (see Figures 10 and 9) is required for model-selection. Finally, especially in embedded system, dataset generation may be extremely time-consuming. Therefore, it is vital to grow the dataset while monitoring the learning curves. Moreover, in the low-accuracy setting with many similar classes, performance may converge well before accuracy (see Figures 6 and 7), thus allowing for smaller datasets.
(3) *Considerations on the machine learning techniques to be adopted.* According to our experimental results, DTs have short inference time, are easily implemented and interpreted, and are effective across all the application range explored in this article. Ensembles of DTs, such as RF and GTB, would allow for an adjustable increase over the DT accuracy, at the cost of more inference time/implementation complexity. They are thus our chosen models.
(4) *Analysis of performance and possible improvements.* At this point is quite clear that increasing the number of possible classes is a great opportunity for improving the performance as long as there are enough features that allow selecting them. Thus, to obtain more performance it is possible to integrate other strategies (e.g., auto-code generation code methods or specific offline compilers as TVM for Deep learning applications) by extending the class domain. However, it is necessary to monitor learning curves and practical measure such as SO or SB. Adding new classes in some case would require to add new features or, generate larger datasets. Both cases have possible negative consequences. Adding a new feature may imply to pay feature extraction costs which for some applications are not negligible [74] and require a detailed analysis. Similarly, adding new entries in the dataset may be not feasible for SoC.

The approach proposed in this article enables the design of effective predictive models that, contrary to existing methods, are *(i)* general, since they are architecture aware and can be used as a black box; *(ii)* easy to implement and integrate; and *(iii)* efficient, considering that we showed superior performance w.r.t. traditional auto-tuners and hand-written heuristics.

There are a number of directions in which this work can be extended. First, it would be interesting to investigate the design of a new ML technique to generate more effective models by

optimizing multi-objective functions (e.g., performance and energy consumption). Moreover, we will study how to generate smaller but still representative training sets. This aspect is particularly crucial for embedded architectures where generating the training set is expensive—just as an example, seven days were necessary to create *po2* for the Mali GPU. Finally, the study of irregular applications and hence the cost of feature extraction remains an open challenge. As an example, the extraction of the sparsity pattern might be costly when dealing with sparse matrix multiplication, and this aspect is crucial for predicting the best storage format, algorithm [6, 7, 19, 58, 72, 74], and other optimization parameters [17].

## 7 RELATED WORK

The use of machine learning techniques to predict performance has been addressed for the first time by Singer and Veloso [57] 20 years ago. De Mesmay et al. [16] applied DTs for automatically generating heuristics. In their work, they did not provide an analysis of the model but they showed the performance the heuristics trained from different dataset. Traditional auto-tuners focused on the efficient exploration huge search space efficiently [2, 45, 63]. However, all of the existing auto-tuners do not address the problem of performance portability on data-driven applications. The problem of exploring a huge search space of tunable parameters has been partially mitigated by the use of meta-heuristics optimization approaches [45, 64] and machine learning techniques [20, 62]. Again, they specifically generate optimized code by assuming a specific input shape. Recently, auto-tuning and input aware techniques [18, 20] have been used in combination to address the problem of performance portability on specific applications [14, 28, 39]. For example, a comprehensive study on automatic tuning of compilers by using machine learning has been provided by Ashouri et al. [4]. An interesting approach extends such techniques in the presence of multiple algorithmic choice [50]. Their solution is suitable when a specific routine is called multiple times. Specifically to BLAS, several optimized linear algebra and BLAS libraries have been deployed [11, 67, 71]. Others provided auto-tuning and optimization approaches to accelerate only GEMM on different architectures [23, 31, 33, 34, 36, 42]. To limit the cost of auto-tuners, boosted trees models have been used to predict parameters by starting from the exploration of a small search space [5]. Recently, model-driven solutions have been adopted for accelerating sparse linear algebra [12, 74]. However, predicting the best data storage require the extraction of complex features. Existing works do not include this cost, but just provide the speed-up without including the feature extraction. Tillet et al. developed ISAAC, which exploits a MLP to generate highly optimized parametric-code in the training step. At run-time, the library infers the best parameters for the specific input [62]. However, since it generates assembly code, it is not able to run on different architectures like ARM. Regarding deep learning applications, a plethora of compiler-based approaches has arisen [9, 10, 61, 77]. AutoTVM [10] generates the best implementation for a specific DNN by extracting domain-specific features from a given low-level abstract syntax tree. The features include loop structure information (e.g., memory access count). Furthermore, their approach does not consider multiple operators but try to optimize just one. In the context of AutoML workloads this approach is not feasible. On the contrary, our method is (a) implementation-independent, since we select among multiple algorithms/implementations even if the source-code is not exposed, and (b) scaled over different input at runtime. Other approaches adopt polyhedral optimization to generate optimized DNN primitives [61].

## 8 CONCLUSION

In this article, we presented and analyzed several models learned through different machine learning techniques for the design of highly-optimized adaptive input-aware libraries. Our analysis shows that models learned from GTB—and more in general techniques based on Decision

Trees—are more effective than other models because of their ability to enable efficient heuristics even in the presence of few features. To validate these results, we optimized two fundamental kernels used in deep learning and scientific computing (convolution and GEMM) and three different libraries: the ARM Compute Library, the Nvidia cuBLAS, and CLBlast. While determining the best operator for convolution involves few classes and many features, the optimal implementation of GEMM has to be chosen among a great variety of classes based on only three features. In the convolution task, due to the simpler class space, several machine learning techniques were able to achieve near-optimal accuracy, and this translated in near-optimal performance—we obtained a 1.5× speedup. For the GEMM tasks, the models did not achieve high accuracy. However, we observed significant improvements in terms of performance (up to 3×) compared to the traditional, hand-tuned approach and traditional auto-tuners. The impact of mispredictions is mitigated when the model is generated from a dense dataset, even when using just few simple features. Our results suggest to avoid the use of specific datasets (e.g., *AntonNet*) in the presence of poor accuracy and in tasks with very complex class spaces. Regardless of their simplicity, decision trees-based models such as Random Forest and GTB achieve good performance even in the presence of high misclassification. Indeed, their learning curves show a good ability to generalize unseen data even with few training points and few features. Even considering different classification tasks, these models exhibited better performance than traditional auto-tuned heuristics overall.

To conclude, we are able to design effective predictive models that, contrary to existing methods, are general, easy to implement and integrate, and efficient, showing superior performance than traditional auto-tuners and hand-written heuristics.

## REFERENCES


[1] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. 303–316.

[2] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT'14)*. ACM, New York, NY, 303–316. DOI: https://doi.org/10.1145/2628071.2628092

[3] ARM. 2018. A Software Library for Computer Vision and Machine Learning. Retrieved from https://www.arm.com/why-arm/technologies/compute-library.

[4] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2018. A survey on compiler autotuning using machine learning. *ACM Comput. Surv.* 51, 5, Article 96 (Sep. 2018), 42 pages. DOI: https://doi.org/10.1145/3197978

[5] J. Bergstra, N. Pinto, and D. Cox. 2012. Machine learning for predictive auto-tuning with boosted regression trees. In *Proceedings of the Conference on Innovative Parallel Computing (InPar'12)*. 1–9. DOI: https://doi.org/10.1109/InPar.2012.6339587

[6] M. Bernaschi, M. Bisson, E. Mastrostefano, and F. Vella. 2018. Multilevel parallelism for the exploration of large-scale graphs. *IEEE Trans. Multi-Scale Comput. Syst.* 4, 3 (2018), 1–1. DOI: https://doi.org/10.1109/TMSCS.2018.2797195

[7] Massimo Bernaschi, Giancarlo Carbone, and Flavio Vella. 2016. Scalable betweenness centrality on multi-GPU systems. In *Proceedings of the ACM International Conference on Computing Frontiers*. ACM, 29–36.

[8] Somashekaracharya G. Bhaskaracharya, Julien Demouth, and Vinod Grover. 2020. Automatic kernel generation for volta tensor cores. Retrieved from https://arXiv:2006.12645.

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*.

[10] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems*. MIT Press, 3389–3400.

[11] Jaeyoung Choi, Jack J. Dongarra, Roldan Pozo, and David W. Walker. 1992. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the 4th Symposium on the Frontiers of Massively Parallel Computation*. IEEE, 120–127.

[12] Jee W. Choi, Amik Singh, and Richard W. Vuduc. 2010. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'10)*. ACM, New York, NY, 115–126. DOI: https://doi.org/10.1145/1693453.1693471

[13] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Mach. Learn.* 20, 3 (1995), 273–297.

[14] B. Cosenza, J. J. Durillo, S. Ermon, and B. Juurlink. 2017. Autotuning stencil computations with structural ordinal regression learning. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'17)*. 287–296. DOI: https://doi.org/10.1109/IPDPS.2017.102

[15] Simon Garcia De Gonzalo, Sitao Huang, Juan Gómez-Luna, Simon Hammond, Onur Mutlu, and Wen-mei Hwu. 2019. Automatic generation of warp-level primitives and atomic instructions for fast and portable parallel reduction on GPUs. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Press, 73–84.

[16] Frédéric de Mesmay, Yevgen Voronenko, and Markus Püschel. 2010. Offline library adaptation using automatically generated heuristics. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'10)*. 1–10.

[17] S. Di Girolamo, F. Vella, and T. Hoefler. 2017. Transparent caching for RMA systems. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'17)*. 1018–1027. DOI: https://doi.org/10.1109/IPDPS.2017.92

[18] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May O'Reilly, and Saman Amarasinghe. 2015. Autotuning algorithmic choice for input sensitivity. *SIGPLAN Not.* 50, 6 (June 2015), 379–390. DOI: https://doi.org/10.1145/2813885.2737969

[19] Agostino Dovier, Andrea Formisano, and Flavio Vella. 2020. GPU-based parallelism for ASP-solving. In *Declarative Programming and Knowledge Management*, Petra Hofstedt, Salvador Abreu, Ulrich John, Herbert Kuchen, and Dietmar Seipel (Eds.). Springer International Publishing, Cham, 3–23.

[20] Thomas L. Falch and Anne C. Elster. 2015. Machine learning based auto-tuning for enhanced opencl performance portability. In *Proceedings of the International Parallel and Distributed Processing Symposium Workshop (IPDPSW'15)*. IEEE, 1231–1240.

[21] Thomas L. Falch and Anne C. Elster. 2017. Machine learning-based auto-tuning for enhanced performance portability of OpenCL applications. *Concurr. Comput.: Pract. Exp.* 29, 8 (2017), e4029–n/a. DOI: https://doi.org/10.1002/cpe.4029e4029 cpe.4029

[22] G. Fursin, A. Lokhmotov, and E. Plowman. 2016. Collective knowledge: Towards R&D sustainability. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE'16)*. 864–869.

[23] Rahul Garg and Laurie Hendren. 2014. A portable and high-performance general matrix-multiply (GEMM) library for GPUs and single-chip CPU/GPU systems. In *Proceedings of the 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, 672–680.

[24] Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.* 34, 3, Article 12 (May 2008), 25 pages. DOI: https://doi.org/10.1145/1356052.1356053

[25] Ananth Grama. 2003. *Introduction to Parallel Computing*. Pearson Education.

[26] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. 2013. Hardware-oblivious parallelism for in-memory column-stores. *Proc. VLDB Endow.* 6, 9 (2013), 709–720.

[27] Tin Kam Ho. 1995. Random decision forests. In *Proceedings of the 3rd International Conference on Document Analysis and Recognition*, Vol. 1. IEEE, 278–282.

[28] Kaixi Hou, Wu-chun Feng, and Shuai Che. 2017. Auto-tuning strategies for parallelizing sparse matrix-vector (SpMV) multiplication on multi-and many-core processors. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'17)*. IEEE, 713–722.

[29] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50× fewer parameters and <0.5 MB model size. Retrieved from https://arXiv:1602.07360.

[30] Intel. 2018. Intel Math Kernel Library. Reference Manual. Retrieved from https://software.intel.com/sites/default/files/managed/83/0a/mkl-2018-developer-reference-c_0.pdf.

[31] Raehyun Kim, Jaeyoung Choi, and Myungho Lee. 2019. Optimizing parallel GEMM routines using auto-tuning with Intel AVX-512. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*. 101–110.

[32] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*. MIT Press, 1097–1105.

[33] Jakub Kurzak, Stanimire Tomov, and Jack Dongarra. 2012. Autotuning GEMM kernels for the Fermi GPU. *IEEE Trans. Parallel Distrib. Syst.* 23, 11 (2012), 2045–2057.

[34] Junjie Lai and André Seznec. 2013. Performance upper bound analysis and optimization of SGEMM on Fermi and Kepler GPUs. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO'13)*. IEEE, 1–10.

[35] Andrew Lavin and Scott Gray. 2016. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 4013–4021.

[36] Roktaek Lim, Yeongha Lee, Raehyun Kim, Jaeyoung Choi, and Myungho Lee. 2019. Auto-tuning GEMM kernels on the Intel KNL and Intel Skylake-SP processors. *J. Supercomput.* 75, 12 (2019), 7895–7908.

[37] Dekang Lin et al. 1998. An information-theoretic definition of similarity. In *Proceedings of the International Conference on Machine Learning (ICML'98)*, Vol. 98. Citeseer, 296–304.

[38] Anton Lokhmotov, Nikolay Chunosov, Flavio Vella, and Grigori Fursin. 2018. 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 (ReQuEST'18)*. ACM, New York, NY, Article 6. DOI: https://doi.org/10.1145/3229762.3229767

[39] Alberto Magni, Dominik Grewe, and Nick Johnson. 2013. Input-aware auto-tuning for directive-based GPU programming. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units (GPGPU'13)*. ACM, New York, NY, 66–75. DOI: https://doi.org/10.1145/2458523.2458530

[40] Partha Maji, Andrew Mundy, Ganesh Dasika, Jesse Beu, Matthew Mattina, and Robert Mullins. 2019. Efficient Winograd or Cook-Toom convolution kernel implementation on widely used mobile CPUs. Retrieved from https://arXiv:1903.01521.

[41] Robert Malouf. 2002. A comparison of algorithms for maximum entropy parameter estimation. In *Proceedings of the 6th Conference on Natural Language Learning-Volume 20*. Association for Computational Linguistics, 1–7.

[42] Kazuya Matsumoto, Naohito Nakasato, and Stanislav G. Sedukhin. 2012. Performance tuning of matrix multiplication in OpenCL on different GPUs and CPUs. In *Proceedings of the SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE, 396–405.

[43] Sharan Narang. [n.d.]. DeepBench. Retrieved from url https://github.com/baidu-research/DeepBench.

[44] Cedric Nugteren. 2018. CLBlast: A tuned OpenCL BLAS library. In *Proceedings of the International Workshop on OpenCL (IWOCL'18)*. Association for Computing Machinery, New York, NY, Article 5, 10 pages. DOI: https://doi.org/10.1145/3204919.3204924

[45] Cedric Nugteren and Valeriu Codreanu. 2015. CLTune: A generic auto-tuner for OpenCL kernels. In *Proceedings of the IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC'15)*. 195–202. DOI: https://doi.org/doi.ieeecomputersociety.org/10.1109/MCSoC.2015.10

[46] Nvidia. 2020. cuBLAS. Basic Linear Algebra on NVIDIA GPUs. Retrieved from https://developer.nvidia.com/cublas.

[47] Stephen M. Omohundro. 1989. *Five Balltree Construction Algorithms*. International Computer Science Institute Berkeley.

[48] Sankar K. Pal and Sushmita Mitra. 1992. Multilayer perceptron, fuzzy sets, and classification. *IEEE Trans. Neural Netw.* 3, 5 (1992), 683–697.

[49] Damiano Perri, Paolo Sylos Labini, Osvaldo Gervasi, Sergio Tasso, and Flavio Vella. 2019. Towards a learning-based performance modeling for accelerating deep neural networks. In *Proceedings of the International Conference on Computational Science and Its Applications*. Springer, 665–676.

[50] P. Pfaffe, M. Tillmann, S. Walter, and W. F. Tichy. 2017. Online-autotuning in the presence of algorithmic choice. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'17)*. 1379–1388. DOI: https://doi.org/10.1109/IPDPSW.2017.28

[51] Victor Podlozhnyuk. 2007. FFT-based 2D convolution. *NVIDIA White Paper* 32 (2007).

[52] David Martin Powers. 2011. Evaluation: From precision, recall and F-measure to ROC, informedness, markedness and correlation. *J. Mach. Learn. Technol* 2 (2011), 2229–3981.

[53] Ari Rasch and Sergei Gorlatch. 2019. ATF: A generic directive-based auto-tuning framework. *Concurr. Comput.: Pract. Exp.* 31, 5 (2019), e4423.

[54] Irina Rish et al. 2001. An empirical study of the naive Bayes classifier. In *Proceedings of the Workshop on Empirical Methods in Artificial Intelligence (IJCAI'01)*, Vol. 3. 41–46.

[55] S. Rasoul Safavian and David Landgrebe. 1991. A survey of decision tree classifier methodology. *IEEE Trans. Syst. Man Cybernet.* 21, 3 (1991), 660–674.

[56] Johannes Sailer, Christian Frey, and Christian Kühnert. 2019. GPU GEMM-kernel autotuning for scalable machine learners. In *Machine Learning for Cyber Physical Systems*. Springer, 66–76.

[57] Bryan Singer and Manuela Veloso. 2000. Learning to predict performance from formula modeling and training data. In *Proceedings of the International Conference on Machine Learning (ICML'00)*. 887–894.

[58] Edgar Solomonik, Maciej Besta, Flavio Vella, and Torsten Hoefler. 2017. Scaling betweenness centrality using communication-efficient sparse matrix multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'17)*. Association for Computing Machinery, New York, NY, Article 47, 14 pages. DOI: https://doi.org/10.1145/3126908.3126971

[59] John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *Comput. Sci. Eng.* 12, 3 (2010), 66–73.

[60] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 1–9.

[61] Sanket Tavarageri, Alexander Heinecke, Sasikanth Avancha, Gagandeep Goyal, Ramakrishna Upadrasta, and Bharat Kaul. 2020. PolyDL: Polyhedral optimizations for creation of high performance DL primitives. Retrieved from https://arXiv:2006.02230.

[62] Philippe Tillet and David Cox. 2017. Input-aware auto-tuning of compute-bound HPC kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'17)*. ACM, New York, NY, Article 43, 12 pages. DOI: https://doi.org/10.1145/3126908.3126939

[63] Ben van Werkhoven. 2019. Kernel tuner: A search-optimizing GPU code auto-tuner. *Future Gen. Comput. Syst.* 90 (2019), 347–358.

[64] Ben Van Werkhoven, Jason Maassen, Henri E. Bal, and Frank J. Seinstra. 2014. Optimizing convolution operations on GPUs using adaptive tiling. *Future Gener. Comput. Syst.* 30 (Jan. 2014), 14–26. DOI: https://doi.org/10.1016/j.future.2013.09.003

[65] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. Retrieved from https://arXiv:1802.04730.

[66] Aravind Vasudevan, Andrew Anderson, and David Gregg. 2017. Parallel multi channel convolution using general matrix multiplication. In *Proceedings of the IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP'17)*. IEEE, 19–24.

[67] R. Clint Whaley and Jack J. Dongarra. 1998. Automatically tuned linear algebra software. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. IEEE Computer Society, 1–27.

[68] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. 2000. Automated empirical optimization of software and the ATLAS project. *Parallel Comput.* 27 (2000), 2001.

[69] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. 2012. OpenACC—First experiences with real-world applications. In *Proceedings of the European Conference on Parallel Processing*. Springer, 859–870.

[70] Catherine Wong, Neil Houlsby, Yifeng Lu, and Andrea Gesmundo. 2018. Transfer learning with neural AutoML. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS'18)*. Curran Associates Inc., Red Hook, NY, 8366–8375.

[71] Zhang Xianyi, Wang Qian, and Zaheer Chothia. 2014. Openblas. Retrieved from http://xianyi.github.io/OpenBLAS.

[72] Zhen Xie, Guangming Tan, Weifeng Liu, and Ninghui Sun. 2019. IA-SpGEMM: An input-aware auto-tuning framework for parallel sparse matrix-matrix multiplication. In *Proceedings of the ACM International Conference on Supercomputing (ICS'19)*. ACM, New York, NY, 94–105. DOI: https://doi.org/10.1145/3330345.3330354

[73] Jerry Ye, Jyh-Herng Chow, Jiang Chen, and Zhaohui Zheng. 2009. Stochastic gradient boosted distributed decision trees. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management*. ACM, 2061–2064.

[74] Yue Zhao, Jiajia Li, Chunhua Liao, and Xipeng Shen. 2018. Bridging the gap between deep learning and sparse matrix format selection. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'18)*. ACM, New York, NY, 94–108. DOI: https://doi.org/10.1145/3178487.3178495

[75] Yulin Zhao, Donghui Wang, and Leiou Wang. 2019. Convolution accelerator designs using fast algorithms. *Algorithms* 12, 5 (2019), 112.

[76] Lanmin Zheng and Tianqi Chen. 2018. Optimizing deep learning workloads on ARM GPU with TVM. In *Proceedings of the 1st Conference on Reproducible Quality-Efficient Systems Tournament on Co-designing Pareto-efficient Deep Learning*. ACM, 3.

[77] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating high-performance tensor programs for deep learning. Retrieved from https://arXiv:2006.06762.

Received July 2020; revised September 2020; accepted November 2020