Evolutionary Algorithm with Non-parametric Surrogate Model for Tensor Program Optimization

1st Ioannis Gatopoulos[12]
University of Amsterdam
Amsterdam, the Netherlands
johngatop@gmail.com

2nd Romain Lepert[1]
Qualcomm AI Research
Qualcomm Technologies Netherlands B.V.
romain@qti.qualcomm.com

3rd Auke Wiggers
Qualcomm AI Research
Qualcomm Technologies Netherlands B.V.
auke@qti.qualcomm.com

4th Giovanni Mariani
Qualcomm AI Research
Qualcomm Technologies Netherlands B.V.
gmariani@qti.qualcomm.com

5th Jakub Tomczak[2]
Vrije Universiteit Amsterdam
Amsterdam, the Netherlands
j.m.tomczak@vu.nl

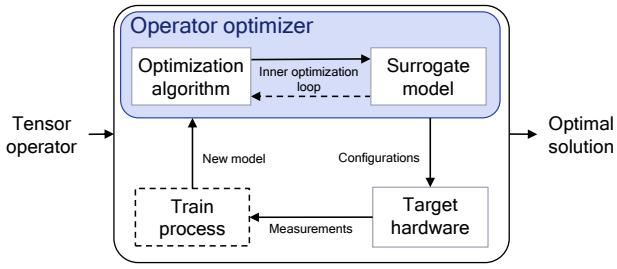
***Abstract*—The efficiency of tensor operators is key to implement fast deep learning models. However, identifying the fastest implementation of a tensor operator for a target hardware is challenging. A wide range of different configurations have to be considered, and the evaluation of a configuration is time consuming as it requires compilation and execution of the operator. A common approach to address these issues is to boost traditional optimization algorithms with a *surrogate model*, *i.e.*, a machine learning model that approximates the objective function and is cheap to query compared to the target hardware. However, as the surrogate model grows in complexity, so does the time needed to train and maintain it. In this work, we propose to use an evolutionary optimizer and augment it with a non-parametric surrogate model (a weighted k-Nearest-Neighbor regression). We evaluate our approach on the convolution layers of a ResNet18, and show a convergence speedup of up to $1.4\times$ when compared to baseline operator tuners.**

***Index Terms*—Evolutionary computing, compilers, neural networks, deep learning**

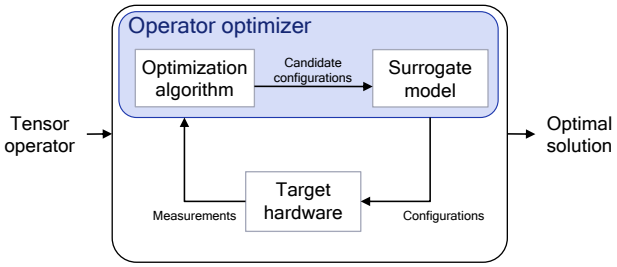
I. Introduction

Modern deep learning (DL) models are computationally intensive to run, as they execute a large number of tensor operations such as matrix multiplications and convolutions. Therefore, to deploy efficient DL models on a target device it is of paramount importance to optimize their execution. Current DL frameworks and packages aim to speed up computation by optimizing the computational graph. Oftentimes, tensor operator implementations are hand-crafted, such as Nvidia cuDNN [1].

The execution time of DL operators can be significantly improved by optimizing them for given hardware at compile time using automatic operator optimization techniques [2], [3]. This optimization process, also known as *autotuning*, alters the execution of a tensor operator, *e.g.*, by using different tiling and loop unrolling. The problem of autotuning is particularly difficult due to the large and discrete search space, which is unique for every tensor operator. Additionally, evaluating a single solution may be computationally expensive as it requires compilation and profiling execution of the operator on-device. Recently, an attempt to address this problem was outlined in TVM [4], a general purpose end-to-end deep learning compiler stack for CPUs and GPUs. TVM is released with a set of autotuners implementing different optimization algorithms. A common approach in state-of-the-art optimization algorithms is to approximate the expensive evaluation on the target hardware by means of a *surrogate model*, *e.g.*, a machine learning model such as XGBoost [5]. Approaches that use these surrogate models (Fig. 1a) are often considered the best



(a) Traditional compiler-autotuning process.



(b) Proposed compiler-autotuning process.

Fig. 1: Traditional (a) and proposed (b) autotuning processes. Dashed lines highlight high-overhead tasks (a): the execution of an *inner optimization loop*, and the *train process* that updates the surrogate model. We propose (b) to leverage a non-parametric surrogate model that does not need training, and we avoid the inner optimization loop by integrating this model in the evolutionary operators of the optimization algorithm.

[1] Equal contribution.
[2] Work done at Qualcomm AI Research. Qualcomm AI Research is an initiative of Qualcomm Technologies, Inc.

978-1-7281-6929-3/20/$31.00 ©2020 IEEE

available option [3], [6], [7], and work by: *a)* executing an inner optimization loop over the surrogate model to select the most promising candidate configurations, *b)* evaluating the selected configurations on the target hardware, *c)* updating the surrogate model with the new observations, *d)* looping back to (a) until a termination criterion is met.

When on-device measurements are sufficiently fast, the inner optimization loop and maintenance of the surrogate model can easily become the bottleneck. Previous approaches aimed at minimizing the total number of configurations evaluated on target hardware [3], [8]. In this work, we specifically investigate the total wall-clock time of the operator optimization process, *i.e.*, the actual time the user should wait for a solution to be returned. We propose a novel methodology that aims to minimize the algorithm execution time overhead while improving the quality of the final results. To achieve this, we employ a non-parametric surrogate model based on the k-Nearest-Neighbor (k-NN) method [9], [10] in order to minimize overhead related to model maintenance. We embed our surrogate model in the operators of an evolutionary algorithm [11]–[13] to quickly estimate the runtime of a candidate configuration. At every iteration (generation), the algorithm proposes a set of candidate configurations. Then, these are evaluated using the surrogate model to select a subset of most promising configurations for on-target evaluation. Since we include the surrogate model as a part of the evolutionary algorithm, there is no inner optimization loop. The full method is visualized in Fig. 1b.

The contributions of this paper are the following:

- We propose an evolutionary algorithm with a non-parametric surrogate model whose goal is to identify the operator implementation with the lowest wall-clock time.
- We provide a full analysis of the search space corresponding to a convolution layer on GPU target hardware. We present a distribution of errors and a break-down on where time is spent. This analysis allows to better understand the autotuning problem, so that practical and efficient tuners can be formulated.
- We evaluate our method on tuning tasks for conv2d operators. We empirically show that the proposed solution surpasses reference methodologies, returning better operator implementations and resulting in up to $1.4\times$ better performance during the first minutes of autotuning.

II. Operator optimization using TVM

We first provide background on the platform we use for operator optimization, TVM [4]. TVM is a compiler stack for CPUs and GPUs that enables optimization of an operator for target hardware by offering fine-grained control over the implementation of nested loops. This has a substantial impact on the data access pattern. While this framework allows for manual optimization of the operator, this is often a time-intensive process that requires domain knowledge of both the operator and hardware.

AutoTVM [3], a sub-package of TVM, allows a user to define an operator using an execution *template* for optimization, which has a fixed number of placeholder values affecting how operators are executed. For example, a convolution template may have a Boolean variable indicating if loop unrolling is used or not. We refer to an instantiation of all placeholder variables as a *configuration*. Note that not every configuration corresponds to a valid kernel. Depending on the operator implementation and the target hardware, some configurations may lead to compilation errors. Additionally, different operators may have different configuration search spaces.

An operator optimization algorithm (or *tuner*) searches for the configuration that minimizes an objective function in the search space defined by the template. An abstract representation of this optimization process, often referred to as *autotuning*, is given in Fig. 1a. Initial on-target measurements are used to train a surrogate model and boost the tuning process. An *optimizer* searches for the configuration that minimizes the cost estimated by the surrogate model. The tuner can run this inner optimization loop for one or more iterations before returning a set of candidate configurations for evaluation. Finally, these candidates are measured on the target device, and their performance is stored in a database, which is then used to fit the surrogate model. This tuning process continues until a stop condition is met, *e.g.*, a threshold performance is met, or a search budget has run out. Finally, the best-performing configuration is returned. Note that performance can be defined in terms of any metric of interest, the most common use case (and the use case we consider) is the minimization of operator execution time.

Ideally, a well-performing solution is returned in the shortest possible walltime. It is therefore important to distinguish between *optimization time*, *i.e.*, the total time the tuner spends searching for the next configuration, *measurement time*, *i.e.*, the time spent to collect measurements, and *runtime*, *i.e.*, the time taken to run the kernel, which is the target metric the tuner is optimizing. If executing actual measurements on-device is cheaper than querying the surrogate model, then it could be beneficial to pick a cheaper model class (or even omit the surrogate entirely). In this work we leverage this idea and propose the integration of a surrogate model that is cheaper than the model that is used in the default tuner provided by AutoTVM. Empirical results demonstrate a clear advantage in terms of the quality of final solution (operator runtime) found in a given wall-clock time.

III. Methodology

A. Problem statement of operator optimization

Let $\mathcal{C}$ be a finite space of configurations $c \in \mathcal{C}$, and let $f : \mathcal{C} \to \mathbb{R}_+$ be an objective function to optimize. In our case, f represents an actual system (a compiler and a hardware) and returns a runtime for a given configuration $c \in \mathcal{C}$ (or a proxy metric thereof). We are dealing with a black-box optimization problem since the objective function can be queried, but its analytic form is unknown [14]. The goal is to find a configuration that minimizes f, namely, $c^* = \operatorname{argmin}_{c \in \mathcal{C}} f(c)$. There are three specific aspects of the given task that make it challenging. First, different configurations have different

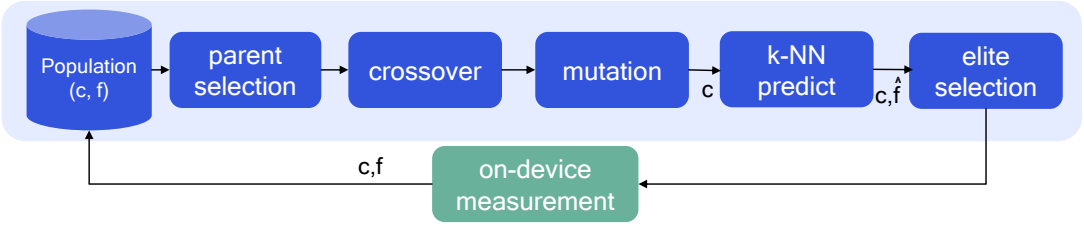


Fig. 2: NoSE pipeline: an evolutionary algorithm is enhanced with a k-Nearest Neighbors based elite selection operator.

measurement times, and as the aim is to achieve convergence to a good solution in the smallest possible total walltime, configurations with long measurement times should be avoided. Second, the search space is discrete, which means the task is a combinatorial optimization problem [15]. Third, the objective function is non-convex and non-smooth.

Combinatorial optimization problems are NP-hard, but approximate and heuristic-based methods can be used to find solutions [16]. Further, since the objective function is non-convex and non-smooth, local search methods cannot be efficiently applied. To cope with this problem there exists a plethora of non-local search algorithms within the evolutionary computing paradigm [17]–[19], which provide convergence guarantees [20] when the following assumptions are met - as is the case in this work: *a)* there is a non-zero probability of generating any individual from the search space, and *b)* the best individual survives unperturbed. Here, we propose to use an evolutionary algorithm to solve the stated problem.

B. Proposed solution

In order to speed up convergence of the evolutionary algorithm, we leverage a non-parametric surrogate model. This model guides the optimization method by updating beliefs about the behavior of the unknown objective function once measurements are available. In particular, as surrogate model we consider a k-Nearest-Neighbors (k-NN) regression model [9], [10] to minimize overhead related to model training and usage. We refer to our approach as Non-parametric Surrogate model for Evolutionary optimizer Tuner, or NoSE for short.

We choose to maximize GFLOPS instead of minimizing runtime, in line with existing work [3], [8]. This does not change the optimal solution or ranking of configurations. For configurations that lead to compilation errors, we set a default measurement of 0 GFLOPS.

1) Surrogate model: the weighted k-NN regressor: The output of the surrogate model $y = \hat{f}(c|k, \mathcal{D})$ is the estimated number of GFLOPS, where $\mathcal{D} = \{(c_n, y_n)\}_{n=1}^{N}$ denotes observed pairs of configurations and measured performance and k is a k-NN hyperparameter. The regressor takes as distance-weighted sum of the neighborhood of the configuration c, using a distance measure $d(\cdot, \cdot)$, as follows:

$$w_{c,i} = \frac{1}{d(c, c_i)}, \tag{1}$$

$$\mathcal{N}_{k,c} = \{i : \text{ indices of } k \text{ nearest neighbors of } c\}, \tag{2}$$

$$\hat{f}(c) = \frac{1}{\sum_{i \in \mathcal{N}_{k,c}} w_{c,i}} \sum_{i \in \mathcal{N}_{k,c}} w_{c,i}\ y_i. \tag{3}$$

Importantly, we need to define a proper distance metric to determine the k nearest neighbors. As the configuration is comprised of ordinal and categorical variables, we use the Canberra distance [21]:

$$d(x, z) = \sum_{l=1}^{L} \frac{|x_d - z_d|}{|x_d| + |z_d|}. \tag{4}$$

The Canberra distance is a weighted version of the Manhattan distance, thus, it is well-suited for ordinal variables [22].

2) Evolutionary algorithm optimizer: We use an evolutionary algorithm (EA) as underlying optimization method to implement NoSE. The traditional approach of running the optimization algorithm on the surrogate model up to convergence, gathering the actual measurement for the so-identified optimum, updating the surrogate model, and looping back by restarting the optimization algorithm on the updated model (Fig. 1a) is not only inefficient but also prone to end up in local optima [6]. Instead, NoSE executes only a single step of optimization (Fig. 1b) by embedding the surrogate model in the evolutionary operators (Fig. 2). The EA starts with a population P_0 of N configurations $c \in P_0$ drawn at random from the configuration space: $P_0 \subset \mathcal{C}$. Each individual c in this population is a vector of discrete values, corresponding to a configuration. During the EA initialization, we evaluate the objective function $f(c)$ for every configuration $c \in P_0$. Then, at each EA iteration t (generation) we proceed as follows:

1) Given the population P_{t-1}, we select M configuration pairs $p_m = (c_i, c_j)$. Each pair represents parents for mating. For each p_m, c_i and c_j are selected with the traditional method of roulette wheel [23], where the fitness function is the real measurement $f(c)$.
2) For each pair p_m, an offspring c_m is generated by applying *single-point crossover*. For this offspring, each gene has a probability p of being replaced by another discrete value sampled at random (*uniform mutation*).
3) We estimate the GFLOPS $\hat{f}(c_m)$ for each offspring c_m using the surrogate model, here, the weighted k-NN.

4) We let the best $E < M$ offspring survive that we refer to as the elite. These elite members are selected with respect to $\hat{f}(c)$, and then evaluated by means of the actual system to gather $f(c)$.
5) The new population P_t is obtained by adding the new E configurations into the old population P_{t-1}, and then shrinking it back to size N by eliminating the worst individuals with respect to their fitness $f(c)$.

The k-NN surrogate model in NoSE does not need to be trained, and needs only a knowledge base $\mathcal{D}$ of observations (as indicated in Equations 1–3). This knowledge base is initialized as $\mathcal{D} = P_0$. At every generation $\mathcal{D}$ is updated by adding the new observations for the E offspring c whose actual objective $f(c)$ has been measured. The efficiency of the surrogate model does decrease as the number of measurements grows, but efficient implementations that leverage k-d trees [24] for fast lookup are readily available [25].

IV. Related work

Computationally expensive deep learning operators such as matrix multiplications and convolutions can be optimized at compile time, *e.g.*, by applying loop transformations such as tiling, unrolling, or interchange [26], [27]. Given the benefits that deep learning systems can achieve when using optimal tensor operators, compile-time optimization techniques for deep learning is an active area of research [3], [4], [8], [28].

Our algorithm fits in the field of optimizing compilers, which concerns itself with using runtime information to further optimize an implementation of a target program [29]. The main challenges are the following: *a)* the huge discrete configuration space, and *b)* the computational cost of measuring the performance of a candidate configuration on the target device. These problems are common in the fields of compiler and computer architecture optimization [6], [30], [31]. Typical approaches for these combinatorial optimization problems rely on specialized algorithms [2], [32], [33], or by taking advantage of machine learning to discover how the configuration parameters shall be tuned for achieving the optimal performance [31], [34], [35]. Combinations of these techniques that simultaneously approximate the objective function with a surrogate model and search that model for the optimal configuration have also been proposed [3], [12], [13], [30].

Evolutionary algorithms are popular solutions to maximize the performance of programs in optimizing compilers. STOKE proposes to identify the program to be used by utilizing genetic algorithms [32]. Tensor Comprehensions suggests to parallelize the execution of a genetic autotuner over multiple GPUs when possible [2]. Genetic algorithms have also been suggested to search for optimal GCC settings [36]. Other works suggest the autotuning of parallel computer programs by selecting evolutionary operators adaptively by using the multi-armed bandits paradigm [37], [38].

The autotuning algorithm we propose combines machine-learning models with evolutionary strategies [39]. Previous work applies those techniques to minimize the total number of configurations evaluated by the optimization algorithm on the actual system, and as a result, the overall tuning wall-clock time (or *walltime*) decreases [3], [4], [8]. In contrast, we suggest that the sole focus of the optimization algorithm should be the overall walltime needed to find the optimal configuration. This wall-clock time includes the time required to compile the different configurations, execute on-device measurements, and the overhead of the optimization algorithm itself. Thus, we propose a light-weight optimization algorithm and we demonstrate that, by applying a simple and computationally-cheap machine-learning technique and integrating it inside an evolutionary algorithm, we can gain significant advantages in terms of total wall-clock time.

V. Experiments

A. Goal

We empirically evaluate the advantages of using the proposed NoSE methodology and compare it with two tuners provided in AutoTVM: a genetic algorithm tuner (GA) [40], and the default TVM tuner, namely, the combination of the simulated annealing optimizer with XGBoost surrogate model [5]. Nowadays, XGBoost is the most commonly used tuner, thus we consider it as main reference methodology. The comparison is carried out with respect to the total wall-clock time (*i.e.* time-to-solution), and quality of the final solution.

We perform two experiments. First, we analyze the complete search space for the C12 tuning task. For this purpose, we measured all possible configurations. The aim of this analysis is to understand the specificity of the problem (*e.g.*, the distribution of valid and invalid configurations, the distribution of target values). To our knowledge, no such analysis has been performed so far. Second, we apply the NoSE tuner on various tuning tasks and compare its performance with baseline tuners.

B. Setup

In this work, we use TVM v0.5 [4] on target hardware Nvidia Geforce GTX 1080Ti. We evaluate the tuning methods on twelve tuning tasks, corresponding to the twelve unique convolution layers of ResNet18 [41]. Each of these layers has a unique parametrization (*e.g.*, input sizes, kernel sizes), leading to a unique search space of configurations. This network architecture is widely used in computer vision applications, such as image classification and object detection, and is used as benchmark architecture in prior work [4], [8].

We use a template *topi.nn.conv2d_nchw*[1] with NCHW layout on the aforementioned target CUDA. This template has 6 ordinal variables that relate to *tiling*, and 2 categorical variables that relate to *loop unrolling*. We show the different tuning tasks and the resulting size of the configuration space in the Table II. For example, for task C1, the convolution layer has 3 input feature maps of size 224×224, and has 64 output channels. For the given template, the associated search space $\mathcal{C}$ consists of 3,763,200 configurations. All measurements are performed on a single GPU.

[1]TVM commit 2005f85, topi/python/topi/nn/conv2d.py, L92.

TABLE I: Configuration types by error category as defined in TVM. Along our experiments, only categories 0, 1, 4, and 6 are encountered.

Error code	Error Type	Details
0	NO_ERROR	No error, valid kernel.
1	INSTANTIATION_ERROR	Actively detected error in instantiating a template with a configuration.
2	COMPILE_HOST	Error when compiling code on host (e.g. tvm.build).
3	COMPILE_DEVICE	Error when compiling code on device (e.g. OpenCL JIT on the device).
4	RUNTIME_DEVICE	Error when run program on device.
5	WRONG_ANSWER	Answer is wrong when compared to a golden output.
6	BUILD_TIMEOUT	Timeout during compilation.
7	RUN_TIMEOUT	Timeout during run.
8	UNKNOWN_ERROR	An error of an unknown nature.

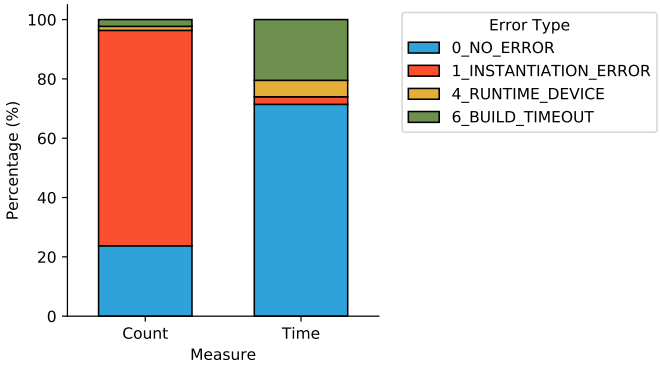


Fig. 3: The distribution of error types and corresponding cumulative measurement time for the C12 configuration space.

C. Experiment 1: Analysis of the C12 search space

The search space of the tuning task is characterized by valid and invalid configurations. To evaluate a configuration, one needs to compile the corresponding kernel, upload it to the device and profile the runtime on device. If this compile-upload-run measurement process is successful, the configuration is valid and gets an associated performance in GFLOPS. If this process fails at some stage, then the configuration is said to be invalid and gets an associated performance 0 GFLOPS. Table I provides a description of all types of configurations considered in the analysis.

We carried out an exhaustive analysis on a single conv2D task to get an in-depth understanding of the search space, the effect of errors, and to find the optimum for the layer. As the exhaustive exploration is expensive in terms of time and compute resources, we carried out this analysis for the C12 task only. Fig. 3 shows the distribution of configurations for each error types, as well as the cumulative walltime spent to evaluate all configurations of each type.

For the chosen template and build configuration, over 75% of all configurations are invalid (`Count` stacked bar in Fig. 3). Additionally, we track the overall *measurement time* for each configuration type (`Time` stacked bar in Figure 3). Interestingly, not all configuration types are expensive to evaluate. The most common error category, *i.e.*, `INSTANTIATION_ERROR`, amounts to less than 5% of the total walltime. This indicates that if a tuner selects a configuration from this category for evaluation, the measurement overhead is negligible because they fail early in the compile-upload-run process. However, there are two rare error categories, namely `RUNTIME_DEVICE` and `BUILD_TIMEOUT`, that have a long measurement time and thus should be avoided by tuners. Finally, valid configurations (`NO_ERROR`) make up 70% of the total walltime and should be where tuners focus their time measuring.

Figure 4 shows the distribution of operator performance for valid configurations in GFLOPS. This resembles the exponential distribution, because many operators have low performance, and few of them have high performance that results in a lighter tail. We expect the majority of walltime in an autotuner to be spent in evaluating valid configurations, as

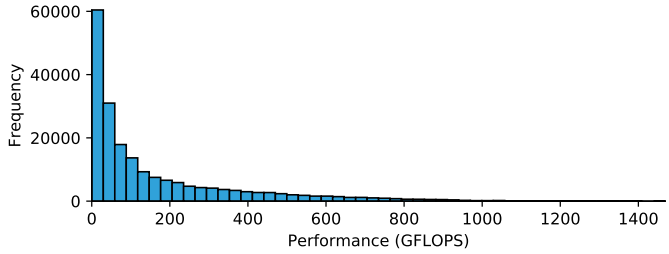


Fig. 4: Distribution of configuration execution performance for the C12 configuration space.

the wall-clock time of valid points (`NO_ERROR`) is significantly larger than the cost of all other error configurations.

These results indicate that the count of configurations evaluated on the actual system by an autotuning framework may be non-representative of the total wall-clock optimization time, *i.e.*, the actual time to solution. In fact, measurement time is variable and depends on the configuration, knowing which configurations are evaluated by the autotuner is more important than knowing how many configurations are evaluated. In particular, evaluating invalid but fast-to-measure configurations is not expensive with respect to evaluating invalid but slow-to-measure configurations. These reasons clarify that tuners should be compared with respect to their total wall-clock time, and that increasing the complexity of the surrogate model (increasing algorithm overhead) in order to avoid non-compilable configurations may be counter-productive. The NoSE tuner is specifically meant to optimize the time-to-solution metric by leveraging a low-overhead surrogate model.

D. Experiment 2: Autotuning conv2D layers

To validate the proposed NoSE approach, we perform autotuning on all 12 unique conv2D layers of ResNet18. The size of the search space and the topology varies among tasks (see number of configurations in Table II). C1, C2, C11 and C12 are used for hyperparameter selection and the remaining 8 tasks are used for testing. We compare NoSE to two state-of-the-art tuners provided in AutoTVM: a Genetic Algorithm based tuner (GA), and a tuner with Simulated Annealing optimizer

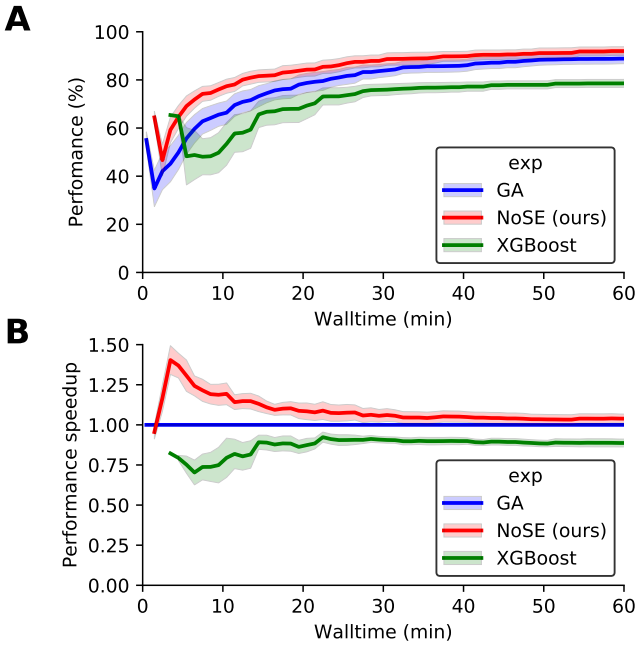


Fig. 5: Results for the experiment 2: (A) The performance of NoSE, GA and XGBoost averaged over all 8 tuning tasks of the test set, relative to the performance of the best found configuration. (B) The averaged performance speedup of NoSE and the XGBoost tuner relative to the GA tuner, averaged over all 8 tuning tasks of the test set. For both plots, the solid line indicates the mean, transparent area indicates the standard error obtained over four runs.

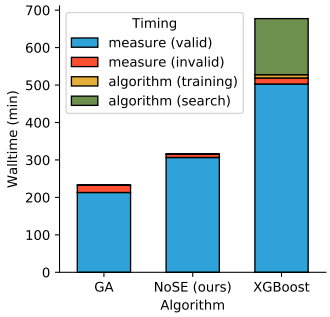


Fig. 6: Optimization time distribution of NoSE, GA and XGBoost after 5000 iterations of tuning, averaged over all 8 tuning tasks of the test set.

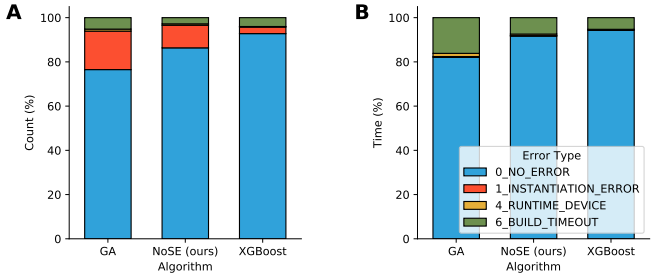


Fig. 7: Error count distribution by type (A) and Error measurement time distribution by type (B), for the NoSE, GA and XGBoost after 5000 iterations of tuning, averaged over all 8 tuning tasks of the test set.

and a XGBoost surrogate model [5] (XGBoost) using default parameters of TVM v0.5.

For the XGBoost tuner, we use the default hyperparameters of AutoTVM including its advanced transfer-learning functionalities[2] [3]. For the GA tuner, we use the default hyperparameters of AutoTVM. For NoSE we set the number of neighbors in k-NN to 9, the population size to $N = 100$, and use mutation with probability $p_m = 0.3$. At every generation, we create $M = 1.5 \times N$ offspring and then select the best $E = 0.3 \times N$ "elite configurations" for actual measurement.

In Fig. 5 we present averaged performance across all tuning tasks. Fig. 5A shows the performance of the best solution found for each method as a function of the walltime. Additionally, Fig. 5B shows the speedup of the best configuration found by NoSE and XGBoost with respect to the best configuration found with the GA tuner along the optimization process. All optimizations are terminated after 60 minutes (wall-clock time). The detailed results for each task are presented in Fig. 8.

In practice, tensor program optimization will be limited by a total tuning time budget, which means that an anytime improvement in performance is valuable. We notice that NoSE outperforms XGBoost by a large margin, for the entire duration of the optimization. NoSE also outperforms GA in the early phase of the optimization, providing a speedup of up to $1.4\times$ within the first 10 minutes. Since evolutionary algorithms generally converge toward the actual optimum [23], both GA and NoSE at convergence return high-quality solutions that are similar in performance.

We further investigate these results by analysing how each tuner spends its time on 5000 iterations of tuning using Fig. 6 and what type of configurations it selected using Fig. 7.

On one hand, compared to GA, NoSE has no noticeable algorithm overhead (Fig. 6). Nevertheless, the NoSE surrogate model search manages to avoid cheap errors of type 1 and expensive errors of type 6 (Fig. 7A) resulting in significantly more measuring time on valid configurations (Fig. 7B). Overall, thanks to a single search step on its k-NN surrogate model, NoSE delivers up to $1.4\times$ anytime performance compared to GA (Fig. 5B).

On the other hand, compared to XGBoost, NoSE has significantly less algorithm overhead (Fig. 6). XGBoost spends about 20% of its time searching for good configurations. While it manages to avoid more cheap errors of type 1 than NoSE, this is not true for expensive errors of type 6 (Fig. 7A), undermining the cost/benefit tradeoff of the search. Overall, the quality of the selected configurations does not compensate for the time-consuming algorithm training and search of the XGBoost tuner, and it underperforms compared to NoSE and its light surrogate model search. These results indicate that the complexity of the inner optimization loop should be taken into account when designing autotuners.

[2]We transfer the surrogate-model learned during the tuning of task C_i forward in the optimization of task C_{i+1}.

TABLE II: Parameters for each unique Conv2D parametrization in a ResNet18, and resulting number of configurations for CUDA target using the scheduling template *topi_nn_conv2d*, layout NCHW. H, W represent spatial dimensions of the input, C_{in} and C_{out} the number of input and output channels.

Task	H	W	C_{in}	C_{out}	kernel	stride	padding	dilation	#configurations
C1	224	224	3	64	(7, 7)	(2, 2)	(3, 3)	(1, 1)	3,763,200
C2	56	56	64	64	(3, 3)	(1, 1)	(1, 1)	(1, 1)	90,316,800
C3	56	56	64	64	(1, 1)	(1, 1)	(1, 1)	(1, 1)	903,168
C4	56	56	64	128	(3, 3)	(2, 2)	(1, 1)	(1, 1)	22,579,200
C5	56	56	64	128	(1, 1)	(2, 2)	(1, 1)	(1, 1)	56,448
C6	28	28	128	128	(3, 3)	(1, 1)	(1, 1)	(1, 1)	36,864,000
C7	28	28	128	256	(3, 3)	(2, 2)	(1, 1)	(1, 1)	5,898,240
C8	28	28	128	256	(1, 1)	(2, 2)	(1, 1)	(1, 1)	1,474,560
C9	14	14	256	256	(3, 3)	(1, 1)	(1, 1)	(1, 1)	9,123,840
C10	14	14	256	512	(3, 3)	(2, 2)	(1, 1)	(1, 1)	570,240
C11	14	14	256	512	(1, 1)	(2, 2)	(1, 1)	(1, 1)	3,564,000
C12	7	7	512	512	(3, 3)	(1, 1)	(1, 1)	(1, 1)	844,800

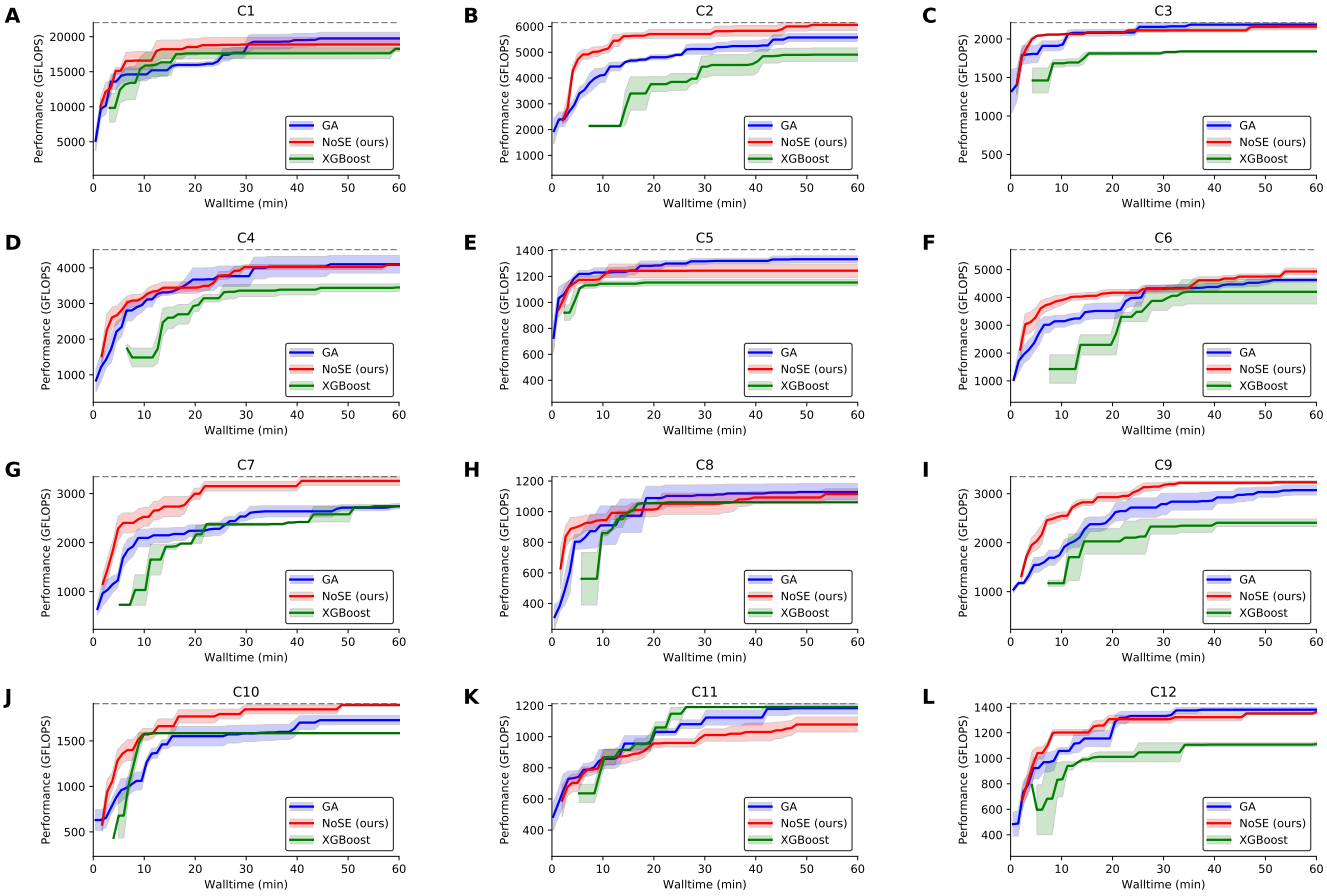


Fig. 8: Performance (in GigaFLOPS) of the best found kernel relative to optimization walltime, for each conv2D tuning task. Solid line indicates the mean, error bars indicate standard error over 4 runs. Dotted line indicates the best kernel found amongst all experiment runs. Higher is better.

VI. Conclusion

In this work, we introduce an algorithm for optimizing operators (or *tuning*) that we refer to as Non-parametric Surrogate Evolutionary search (NoSE). It uses an evolutionary algorithm to search the space of possible operator implementations, and uses a simple yet effective k-Nearest Neighbor regression model to navigate the optimizer towards fast compilation configurations. The application of the surrogate model allows to skip the expensive compilations and on-device performance assessment for poor configuration candidates.

We first present an exhaustive analysis of the search space of a chosen conv2d layer. This analysis allows to get an insight into the autotuning process. We then present results for 8 conv2d tuning tasks collected using TVM, that demonstrate the supremacy of NoSE over an XGBoost-based baseline tuner in terms of quality of the proposed kernel configuration. Moreover, the proposed approach surpasses the GA tuner with almost negligible computation overhead. We believe that our work opens new opportunities for research on automated tensor program optimization.

REFERENCES

[1] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning." *CoRR*, vol. abs/1410.0759, 2014.

[2] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," *arXiv preprint arXiv:1802.04730*, 2018.

[3] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to Optimize Tensor Programs," in *Advances in Neural Information Processing Systems*, 2018, pp. 3389–3400.

[4] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Q. Yan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: end-to-end optimization stack for deep learning," vol. abs/1802.04799, 2018.

[5] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," in *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: ACM, 2016, pp. 785–794.

[6] G. Mariani, G. Palermo, V. Zaccaria, and C. Silvano, "Oscar: An optimization methodology exploiting spatial correlation in multicore design spaces," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 5, pp. 740–753, May 2012.

[7] J. M. Tomczak, R. Lepert, and A. Wiggers, "Simulating Execution Time of Tensor Programs using Graph Neural Networks," *arXiv preprint arXiv:1904.11876*, 2019.

[8] B. H. Ahn, P. Pilligundla, and H. Esmaeilzadeh, "Reinforcement learning and adaptive sampling for optimized dnn compilation," *arXiv preprint arXiv:1905.12799*, 2019.

[9] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992.

[10] T. M. Cover, P. Hart *et al.*, "Nearest neighbor pattern classification," *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 21–27, 1967.

[11] M. Bhattacharya, "Evolutionary Approaches to Expensive Optimisation," *arXiv preprint arXiv:1303.2745*, 2013.

[12] R. Piscitelli and A. D. Pimentel, "Interleaving methods for hybrid system-level mpsoc design space exploration," in *2012 International Conference on Embedded Computer Systems (SAMOS)*, July 2012, pp. 7–14.

[13] G. Mariani, G. Palermo, C. Silvano, and V. Zaccaria, "Meta-model assisted optimization for design space exploration of multi-processor systems-on-chip," in *2009 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools*, Aug 2009, pp. 383–389.

[14] D. R. Jones, M. Schonlau, and W. J. Welch, "Efficient global optimization of expensive black-box functions," *Journal of Global Optimization*, vol. 13, no. 4, pp. 455–492, 1998.

[15] M. Syslo, N. Deo, and J. S. Kowalik, *Discrete Optimization Algorithms with Pascal Programs*. Prentice Hall Professional Technical Reference, 1983.

[16] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM computing surveys (CSUR)*, vol. 35, no. 3, pp. 268–308, 2003.

[17] T. Bäck, *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford University Press, 1996.

[18] A. E. Eiben and J. E. Smith, *Introduction to evolutionary computing*. Springer, 2003, vol. 53.

[19] A. E. Eiben and J. Smith, "From evolutionary computation to the evolution of things," *Nature*, vol. 521, no. 7553, p. 476, 2015.

[20] A. E. Eiben, E. H. L. Aarts, and K. M. Van Hee, "Global convergence of genetic algorithms: A markov chain analysis," in *Parallel Problem Solving from Nature*, H.-P. Schwefel and R. Männer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 3–12.

[21] G. N. Lance and W. T. Williams, "Computer programs for hierarchical polythetic classification (similarity analyses)," *The Computer Journal*, vol. 9, no. 1, pp. 60–64, 1966.

[22] G. Jurman, S. Riccadonna, R. Visintainer, and C. Furlanello, "Canberra distance on ranked lists," in *Proceedings of Advances in Ranking NIPS 09 workshop*. Citeseer, 2009, pp. 22–27.

[23] M. Gen and L. Lin, "Genetic algorithms," *Wiley Encyclopedia of Computer Science and Engineering*, pp. 1–15, 2007.

[24] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[25] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[26] B. Pradelle, B. Meister, M. Baskaran, J. Springer, and R. Lethin, "Polyhedral optimization of tensorflow computation graphs," in *Programming and Performance Visualization Tools*. Springer, 2017, pp. 74–89.

[27] R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe, "Tiramisu: A polyhedral compiler for expressing fast and portable code," in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO 2019. Piscataway, NJ, USA: IEEE Press, 2019, pp. 193–205.

[28] J. Roesch, S. Lyubomirsky, L. Weber, J. Pollock, M. Kirisame, T. Chen, and Z. Tatlock, "Relay: A new ir for machine learning frameworks," in *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, ser. MAPL 2018. New York, NY, USA: ACM, 2018, pp. 58–68.

[29] K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2001.

[30] G. Mariani, G. Palermo, V. Zaccaria, and C. Silvano, "Desperate++: An enhanced design space exploration framework using predictive simulation scheduling," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 2, pp. 293–306, Feb 2015.

[31] A. H. Ashouri, A. Bignoli, G. Palermo, C. Silvano, S. Kulkarni, and J. Cavazos, "Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 3, pp. 29:1–29:28, Sep. 2017.

[32] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic superoptimization," in *ACM SIGPLAN Notices*, vol. 48, no. 4. ACM, 2013, pp. 305–316.

[33] C. Oh, J. Tomczak, E. Gavves, and M. Welling, "Combinatorial Bayesian Optimization using the Graph Cartesian Product," in *Advances in Neural Information Processing Systems*, 2019, pp. 2910–2920.

[34] Z. Wang and M. OBoyle, "Machine learning in compiler optimization," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1879–1901, Nov 2018.

[35] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A Survey on Compiler Autotuning Using Machine Learning," *ACM Comput. Surv.*, vol. 51, no. 5, pp. 96:1–96:42, Sep. 2018.

[36] P. A. Ballal, H. Sarojadevi, and P. Harsha, "Compiler optimization: A genetic algorithm approach," *International Journal of Computer Applications*, vol. 112, no. 10, 2015.

[37] A. Fialho, L. Da Costa, M. Schoenauer, and M. Sebag, "Analyzing bandit-based adaptive operator selection mechanisms," *Annals of Mathematics and Artificial Intelligence*, vol. 60, no. 1-2, pp. 25–64, 2010.

[38] M. Pacula, J. Ansel, S. Amarasinghe, and U.-M. OReilly, "Hyperparameter tuning in bandit-based adaptive operator selection," in *European Conference on the Applications of Evolutionary Computation*. Springer, 2012, pp. 73–82.

[39] Y. Jin, M. Olhofer, and B. Sendhoff, "On evolutionary optimization with approximate fitness functions," in *Proceedings of the 2Nd Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO'00. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, pp. 786–793.

[40] M. H. Yar, V. Rahmati, H. Reza, and D. Oskouei, "A survey on evolutionary computation: Methods and their applications in engineering," *Mod. Appl. Sci*, vol. 10, no. 11, p. 131139, 2016.

[41] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2015.