Learning to Optimize Tensor Programs

Tianqi Chen[1] **Lianmin Zheng**[2] **Eddie Yan**[1] **Ziheng Jiang**[1] **Thierry Moreau**[1]
Luis Ceze[1] **Carlos Guestrin**[1] **Arvind Krishnamurthy**[1]
[1]Paul G. Allen School of Computer Science & Engineering, University of Washington
[2]Shanghai Jiao Tong University

Abstract

We introduce a learning-based framework to optimize tensor programs for deep learning workloads. Efficient implementations of tensor operators, such as matrix multiplication and high dimensional convolution, are key enablers of effective deep learning systems. However, current systems rely on manually optimized libraries, e.g., cuDNN, that support only a narrow range of server class GPUs. Such reliance limits the applicability of high-level graph optimizations and incurs significant engineering costs when deploying to new hardware targets. We use learning to remove this engineering burden. We learn domain-specific statistical cost models to guide the search of tensor operator implementations over billions of possible program variants. We further accelerate the search using effective model transfer across workloads. Experimental results show that our framework delivers performance that is competitive with state-of-the-art hand-tuned libraries for low-power CPUs, mobile GPUs, and server-class GPUs.

1 Introduction

Deep learning (DL) has become ubiquitous in our daily lives. DL models can now recognize images [23], understand natural language [38], play games [27], and automate system decisions (e.g., device placement [26] and indexing [21]). Tensor operators, such as matrix multiplication and high dimensional convolution, are basic building blocks of DL models. Scalable learning systems [1, 4, 8, 2] rely on manually optimized, high-performance tensor operation libraries, such as cuDNN, that are optimized for a narrow range of hardware devices. To optimize a tensor operator, programmers must choose from many implementations that are logically equivalent but differ dramatically in performance due to differences in threading, memory reuse, pipelining and other hardware factors. Supporting diverse hardware back-ends therefore requires tremendous engineering effort. Even on currently supported hardware, developing DL frameworks and models is limited by the set of optimized operators in libraries, preventing optimizations (such as operator fusion) that can produce unsupported operators.

This research explores the following question: can we use learning to alleviate this engineering burden and automatically optimize tensor operator programs for a given hardware platform? Our affirmative answer is based on statistical cost models we built that predict program run time using a given low-level program. These cost models, which guide our exploration of the space of possible programs, use transferable representations that generalize across different workloads to accelerate search. We make the following contributions:

- We formalize the new problem of learning to optimize tensor programs and summarize its key characteristics.
- We propose a machine learning framework to solve this problem.
- We further accelerate the optimization by $2\times$ to $10\times$ using transfer learning.

32nd Conference on Neural Information Processing Systems (NeurIPS 2018), Montréal, Canada.

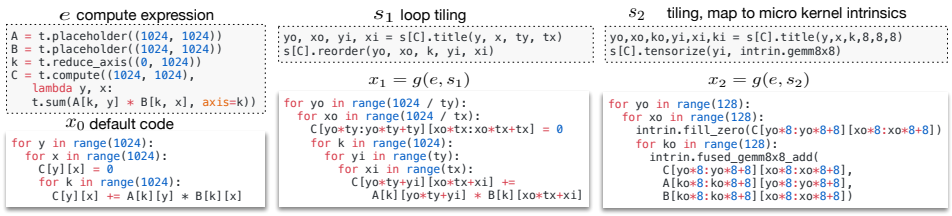


Figure 1: Sample problem. For a given tensor operator specification ($C_{ij} = \sum_k A_{ki} B_{kj}$), there are multiple possible low-level program implementations, each with different choices of loop order, tiling size, and other options. Each choice creates a logically equivalent program with different performance. Our problem is to explore the space of programs to find the fastest implementation.

We provide a detailed empirical analysis of component design choices in this framework. *Experimental results on real-world DL workloads show that our framework yields end-to-end performance improvements ranging from 1.2× to 3.8× over existing frameworks.*

2 Problem Formalization

We begin by walking through the motivating example in Figure 1. To enable automatic code generation, we specify tensor operators using index expressions (e.g., $C_{ij} = \sum_k A_{ki} B_{kj}$). Let $\mathcal{E}$ denote the space of index expressions. The index expression leaves many low-level implementation details, such as loop order, memory scope, and threading unspecified. As a result, we can generate multiple variants of low-level code that are logically equivalent to the expression for a given $e \in \mathcal{E}$. We use $\mathcal{S}_e$ to denote the space of possible transformations (schedules) from e to low-level code. For an $s \in \mathcal{S}_e$, let $x = g(e, s)$ be the generated low-level code. Here, g represents a compiler framework that generates low-level code from e, s. We are interested in minimizing $f(x)$, which is the real run time cost on the hardware. Importantly, we do not know an analytical expression for $f(x)$ but can query it by running experiments on the hardware. For a given tuple of $(g, e, \mathcal{S}_e, f)$, our problem can be formalized as the following objective:

$$\arg\min_{s \in \mathcal{S}_e} f(g(e, s)) \tag{1}$$

This problem formalization is similar to that of traditional hyper-parameter optimization problems [34, 33, 35, 13, 17, 25] but with several key differentiating characteristics:

Relatively Low Experiment Cost. Traditionally, hyper-parameter optimization problems incur a high cost to query f, viz., running experiments could take hours or days. However, the cost of compiling and running a tensor program is a few seconds. This property requires that model training and inference be *fast* ; otherwise, there is no benefit over profiling execution on real hardware. It also means that we can collect more training data during optimization.

Domain-Specific Problem Structure. Most existing hyper-parameter optimization algorithms treat the problem as a black box. As we optimize programs, we can leverage their rich structures to build effective models.

Large Quantity of Similar Operators. An end-to-end DL system must optimize tensor operator programs for different input sizes, shapes, and data layout configurations. These tasks are similar and can offer opportunities for transfer learning.

We describe two key prerequisites for automatic code generation that is competitive with hand-optimized code. (**1**) We need to define an exhaustive search space $\mathcal{S}_e$ that covers all hardware-aware optimizations in hand-tuned libraries. (**2**) We need to efficiently find an optimal schedule in $\mathcal{S}_e$.

There are many domain-specific languages (DSLs) for code generation [32, 36, 15, 37, 20, 30], each with with a different $\mathcal{E}$, $\mathcal{S}_e$ and g. Polyhedral models [5, 42, 41] are a popular choice for $\mathcal{S}_e$; they model the loop domains as integer linear constraints. An alternative approach originating from Halide [32] defines a schedule space using a set of transformation primitives. Improving $\mathcal{S}_e$ is an important research direction that is beyond the scope of this paper; we pick a rich $\mathcal{S}_e$ and focus on schedule optimization in the rest of the paper.

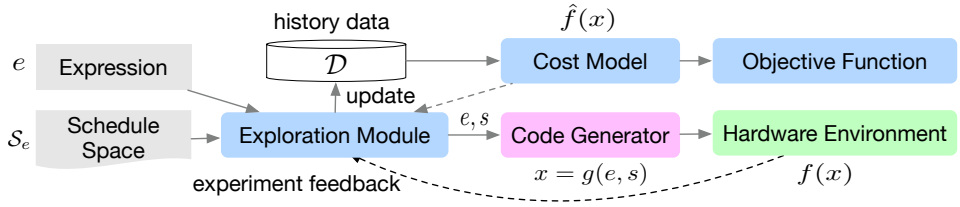


Figure 2: Framework for learning to optimize tensor programs.

We use primitives from an existing code generation framework [9] to form $\mathcal{S}_e$. Our search space includes multi-level tiling on each loop axis, loop ordering, shared memory caching for GPUs, and annotations such as unrolling and vectorization. The search space size $|\mathcal{S}_e|$ can be on the order of billions of possible implementations for a single GPU operator. As we find in section 6 , our choice of $\mathcal{S}_e$ can contain programs competitive with hand-optimized libraries.

3 Learning to Optimize Tensor Programs

We propose a machine learning (ML)-based framework to solve this problem. Figure 2 presents the framework and its modules. We build a statistical cost model $\hat{f}(x)$ to estimate the cost of each low-level program x. An exploration module proposes new schedule configurations to run on the hardware. The run time statistics are collected in a database $\mathcal{D} = \{(e_i, s_i, c_i)\}$, which can in turn be used to update $\hat{f}$. We discuss module-specific design choices in the following subsections.

3.1 Statistical Cost Model

The first statistical model we support is based on gradient boosted trees [11](GBTs). We extract domain-specific features from a given low-level abstract syntax tree (AST) x. The features include loop structure information (e.g., memory access count and data reuse ratio) and generic annotations (e.g., vectorization, unrolling, thread binding). We use XGBoost [7], which has proven to be a strong feature-based model in past problems. Our second model is a TreeGRU[39], which recursively encodes a low-level AST into an embedding vector. We map the embedding vector to a final predicted cost using a linear layer.

GBT and TreeGRU represent two distinct ML approaches to problem resolution. Both are valuable, but they offer different benefits. GBT relies on precise feature extraction and makes fast predictions using CPUs. TreeGRU, the deep learning-based approach, is extensible and requires no feature engineering, but it lags in training and predictive speed. We apply batching to the TreeGRU model and use GPU acceleration to make training and prediction fast enough to be usable in our framework.

3.2 Training Objective Function

We can choose from multiple objective functions to train a statistical cost model for a given collection of data $\mathcal{D} = \{(e_i, s_i, c_i)\}$. A common choice is the regression loss function $\sum_i (\hat{f}(x_i) - c_i)^2$, which encourages the model to predict cost accurately. On the other hand, as we care only about the relative order of program run times rather than their absolute values in the selection process, we can instead use the following rank loss function [6]:

$$\sum_{i,j} \log(1 + e^{-\operatorname{sign}(c_i - c_j)(\hat{f}(x_i) - \hat{f}(x_j))}). \tag{2}$$

We can use the prediction $\hat{f}(x)$ to select the top-performing implementations.

3.3 Exploration Module

The exploration module controls the search loop, which is summarized in Algorithm 1. At each iteration, it must pick a batch of candidate programs based on $\hat{f}(x)$ and query $f(x)$ on real hardware. We cannot simply enumerate the entire space of S_e and pick the top-b candidates due to the size of the search space. Instead, we use simulated annealing [19] with $\hat{f}(x)$ as the energy function.

Algorithm 1: Learning to Optimize Tensor Programs

Input : Transformation space $\mathcal{S}_e$
Output : Selected schedule configuration s^*
$\mathcal{D} \leftarrow \emptyset$
while *n_trials* $<$ *max_n_trials* **do**
  `// Pick the next promising batch`
  $Q \leftarrow$ run parallel simulated annealing to collect candidates in $\mathcal{S}_e$ using energy function $\hat{f}$
  $S \leftarrow$ run greedy submodular optimization to pick $(1-\epsilon)b$-subset from Q by maximizing Equation 3
  $S \leftarrow S \cup \{$ Randomly sample ϵb candidates. $\}$
  `// Run measurement on hardware environment`
  for s ***in*** S **do**
    $c \leftarrow f(g(e, s))$; $\mathcal{D} \leftarrow \mathcal{D} \cup \{(e, s, c)\}$
  end
  `// Update cost model`
  update $\hat{f}$ using $\mathcal{D}$
  n_trials $\leftarrow$ n_trials $+ b$
end
$s^* \leftarrow$ history best schedule configuration

Specifically, we use a batch of parallel Markov chains to improve the prediction throughput of the statistical cost model. We select the top-performing batch of candidates to run on real hardware. The collected performance data is used to update $\hat{f}$. We make the states of the Markov chains persistent across $\hat{f}$ updates. We also apply the ϵ-greedy to select ϵb (e.g. 0.05) candidates randomly to ensure exploration.

Diversity-Aware Exploration. We consider both quality and diversity when selecting b candidates for hardware evaluation. Assume that the schedule configuration s can be decomposed into m components $s = [s_1, s_2, \cdots s_m]$. We maximize the following objective to select candidate set S from the top λb candidates:

$$L(S) = -\sum_{s \in S} \hat{f}(g(e, s)) + \alpha \sum_{j=1}^{m} |\cup_{s \in S} \{s_j\}| \tag{3}$$

The first term encourages us to pick candidates with low run time costs. The second term counts the number of different configuration components that are covered by S. $L(S)$ is a submodular function, and we can apply the greedy algorithm [29, 22] to get an approximate solution.

Uncertainty Estimator. Bayesian optimization methods [34, 33, 35, 17] use acquisition functions other than the mean when an uncertainty estimate of $\hat{f}$ is available. Typical choices include expected improvement (EI) and upper confidence bound (UCB). We can use bootstrapping to get the model's uncertainty estimate and validate the effectiveness of these methods. As we will see in section 6 , considering uncertainty does not improve the search in our problem. However, the choice of acquisition function remains a worthy candidate for further exploration.

4 Accelerating Optimization via Transfer Learning

Thus far, we have focused only on learning to optimize a single tensor operator workload. In practice, we need to optimize for many tensor operators with different input shapes and data types. In a real world setting, the system collects historical data $\mathcal{D}'$ from previously seen workloads. We can apply transfer learning to effectively use $\mathcal{D}'$ to speed up the optimization.

The key to transfer learning is to create a **transferable representation** that is **invariant** to the source and target domains. We can then share the cost model using the common representation across domains. Different choices of representations may have different levels of invariance.

A common practice in Bayesian optimization methods is to directly use configuration s as the model's input. However, the search space specification can change for different workloads or when the user specifies a new search space for the same workload. The configuration representation s is not invariant to changes in the search space.

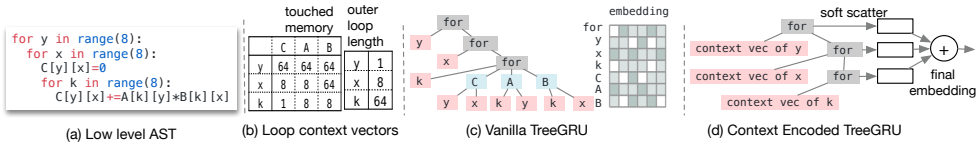


Figure 3: Possible ways to encode the low-level loop AST.

Workload Name	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12
H, W	224,224	56,56	56,56	56,56	56,56	28,28	28,28	28,28	14,14	14,14	14,14	7,7
IC, OC	3,64	64,64	64,64	64,128	64,128	128,128	128,256	128,256	256,256	256,512	256,512	512,512
K, S	7,2	3,1	1,1	3,2	1,2	3,1	3,2	1,2	3,1	3,2	1,2	3,1

Table 1: Configurations of all conv2d operators in a single batch ResNet-18 inference. H,W denotes height and width, IC input channels, OC output channels, K kernel size, and S stride size.

On the other hand, the low-level loop AST x (Figure 3a) is a shared representation of programs that is invariant to the search space. To leverage this invariance, our cost model $\hat{f}(x)$ takes the low-level loop AST x as input. We also need to encode x into a vector space to perform prediction. The specific encoding of x can also result in different levels of invariance.

Context Relation Features for GBT. We define context features at each loop level to represent loop characteristics. A simple representation of context features is a vector (e.g., in Figure 3b, where each loop has a row of features). Context features are informative but, crucially, cannot generalize across different loop nest patterns; we define context relation features to overcome this issue.

To build context relation features, we treat context vectors as a bag of points and extract features that model relations between feature axes. Formally, let Z be the context feature matrix such that Z_{ki} corresponds to the i-th feature of loop k. We define a set of $\log_2$-spaced constant thresholds $\beta = [\beta_1, \beta_2, \cdots \beta_m]$. The relation feature between feature i and j is defined as: $R_t^{(ij)} = \max_{k \in \{k | Z_{kj} < \beta_t\}} Z_{ki}$. This encoding summarizes useful relations, such as loop count vs. touched memory size (related to the memory hierarchy of the access), that affect run time cost.

Context Encoded TreeGRU. The invariant representation also exists for the neural-based model. Figure 3c shows a way to encode the program by learning an embedding vector for each identifier and summarizing the AST using TreeGRU. This model works well for modeling a single workload. However, the set of loop variables can change across different domains, and we do not have embedding for the new loop variables. We instead encode each loop variable using the context vector extracted for GBT to summarize the AST (Figure 3d). We scatter each loop level, embedding h into m vectors using the rule $out_i = \text{softmax}(W^T h)_i h$. Conceptually, the softmax classifies the loop level into a memory slot in out. Then, we sum the scattered vectors of all loop levels to get the final embedding.

Once we have a transferable representation, we can use a simple transfer learning method by combining a global model and an in-domain local model, as follows:

$$\hat{f}(x) = \hat{f}^{(global)}(x) + \hat{f}^{(local)}(x). \tag{4}$$

The global model $\hat{f}^{(global)}(x)$ is trained on $\mathcal{D}'$ using the invariant representation; it helps to make effective initial predictions before we have sufficient data to fit $\hat{f}^{(local)}(x)$.

5 Prior Work

Black box optimization (auto-tuning) is used in high-performance computing libraries such as ATLAS [43] and FFTW [12]. Alternatively, a hardware-dependent cost model can be built to guide the search [28, 5]. Polyhedral methods [5, 42] use integer linear programming to optimize cost. Tensor Comprehensions [41] combine both approaches, using black-box optimization to choose parameters of thread blocks and polyhedral optimization to generate internal loops. Black-box approaches can require many experiment trials to explore a huge $\mathcal{S}_e$. On the other hand, predefined cost models may not be sufficiently accurate to capture the complexity of modern hardware and must be manually redefined for each new hardware target.

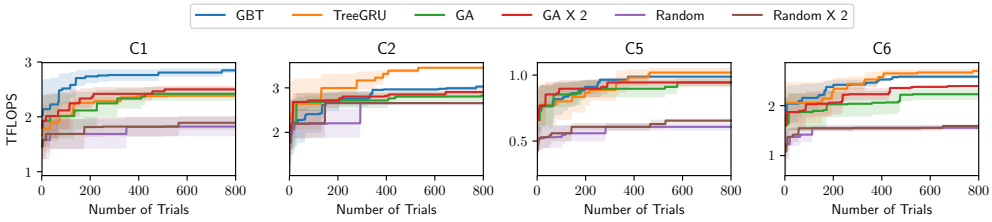


Figure 4: Statistical cost model vs. genetic algorithm (GA) and random search (Random) evaluated on NVIDIA TITAN X. 'Number of trials' corresponds to number of evaluations on the real hardware. We also conducted two hardware evaluations per trial in Random $\times 2$ and GA $\times 2$. Both the GBT- and TreeGRU-based models converged faster and achieved better results than the black-box baselines.

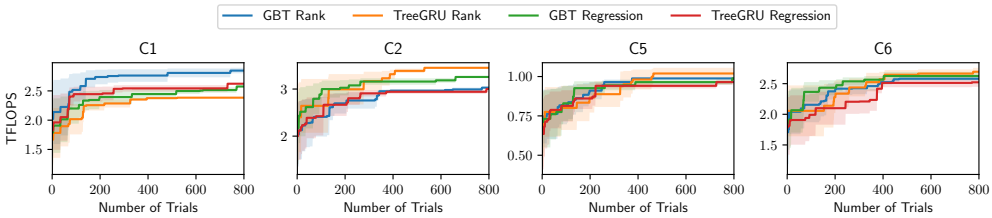


Figure 5: Rank vs. Regression objective function evaluated on NVIDIA TITAN X. The rank-based objective either outperformed or performed the same as the regression-based objective in presented results.

Previously, statistical cost models have been applied to optimize SAT solvers [17, 18]. We apply this idea to our problem and build a domain-specific cost model that enables effective transfer among workloads. A recent trend is to use deep neural networks to perform program analysis [3, 10]. Our new problem setting and experiment environment can serve as a testbed for unexplored research opportunities in related directions.

6 Experiments

6.1 Component Evaluations

We first evaluated each design choice in the framework. Component evaluations were based on convolution workloads in ResNet-18 [14] for ImageNet classification (Table 1). Due to space limitations, we show component evaluation results only on representative workloads; the complete set of results is reported in the supplementary material. All methods compared in this subsection were initialized with no historical data. Section 6.2 evaluates the transfer learning setting.

Importance of Statistical Cost Model. Figure 4 compares the performance of the statistical cost model to black-box methods. Both the GBT and TreeGRU models outperformed the black-box methods and found operators that were $2\times$ faster than those found with random searches. This result is particularly interesting compared to prior results in hyper-parameter tuning [25], where model-based approaches were shown to work only as well as random searching. Our statistical models benefit from domain-specific modeling and help the framework find better configurations.

Choice of Objective Function. We compared the two objective functions in Figure 5 on both types of models. In most cases, we found that using a rank-based objective was slightly better than using a regression-based one: the rank-based objective may have sidestepped the potentially challenging task of modeling absolute cost values. We chose rank as our default objective.

Impact of Diversity-Aware Exploration. We evaluated the impact of the diversity-aware exploration objective in Figure 6. Most of the workloads we evaluated showed no positive or negative impact for diversity-based selection. However, diversity-aware exploration improved C6, which shows some potential usefulness to the approach. We adopted the diversity-aware strategy since it can be helpful, has no meaningful negative impact, and negligibly affects run time.

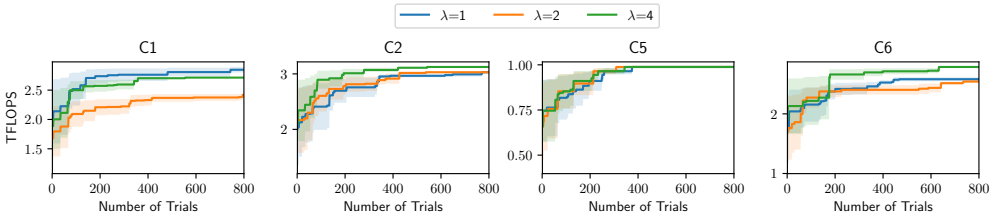


Figure 6: Impact of diversity-aware selection with different choices of λ evaluated on NVIDIA TITAN X. Diversity-aware selection had no positive or negative impact on most of the evaluated workloads.

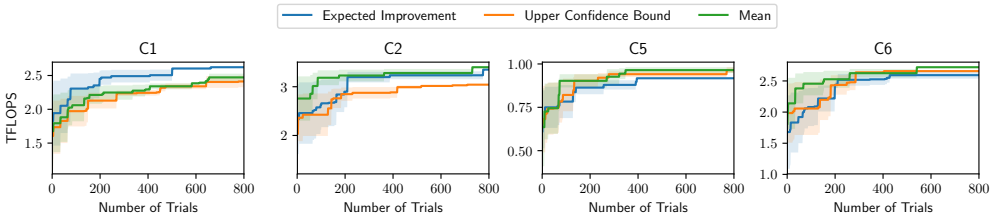


Figure 7: Impact of uncertainty-aware acquisition functions evaluated on NVIDIA TITAN X. Uncertainty-aware acquisition functions yielded no improvements in our evaluations.

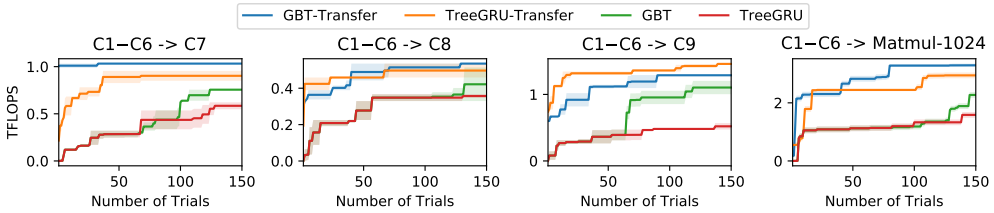


Figure 8: Impact of transfer learning. Transfer-based models quickly found better solutions.

Impact of Uncertainty Estimator. We evaluated the usefulness of uncertainty-aware acquisition functions in Figure 7. The uncertainty measurement was achieved by training five models using bootstrapping. We used the regression objective in this setting—similar to its use in most Bayesian optimization methods. Results show that uncertainty estimation was not as important in our problem, possibly because our models were trained with more training samples than traditional hyper-parameter optimization problems.

6.2 Transfer Learning Evaluations

The evaluations presented so far used no historical data. This subsection evaluates the improvements obtainable with transfer learning.

Improvements by Transfer. We first evaluated general improvements made possible by transfer learning. We randomly picked samples from $\mathcal{D}'$ collected from C1,C2,C3,C4,C5,C6 and used them to form the source domain (30000 samples in the TITAN X experiment and 20000 samples in the ARM GPU and ARM A53 experiments). We then compared the performance of transfer-enabled methods to learning from scratch for target workloads C7,C8,C9. Results are shown in Figure 8. Overall, using transfer learning yielded a $2\times$ to $10\times$ speedup. This approach is especially important for real DL compilation systems, which continuously optimize incoming workloads.

Invariant Representation and Domain Distance. As discussed in Section 4, different representations have different levels of invariance. We used three scenarios to study the relationship between domain distance and the invariance of feature representations: (1) running optimizations on only one target domain; (2) C1–C6→7: C1–C6 as source domain and C7 as target domain (transfer within same operator type); (3) C1–C6→Matmul-1024: C1–C6 as source domain and matrix multiplication as

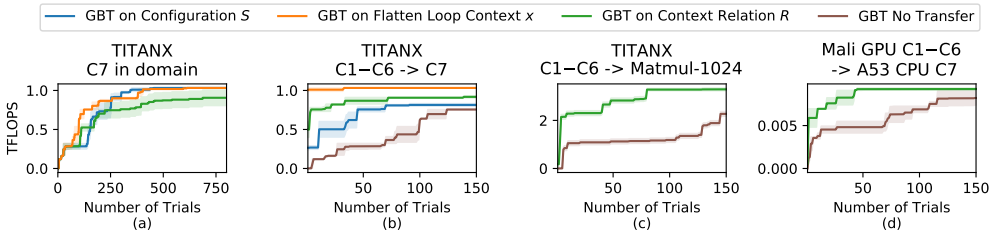
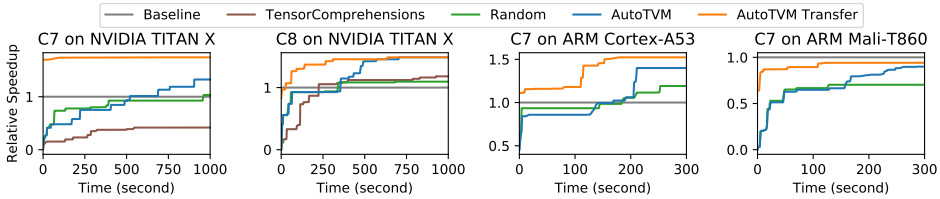
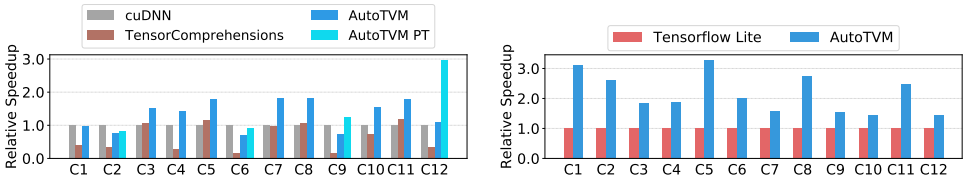


Figure 9: Comparison of different representations in different transfer domain settings. The configuration-based model can be viewed as a typical Bayesian optimization approach (batched version of SMAC [17]). We found that models using configuration space features worked well within a domain but were less useful across domains. The flattened AST features worked well when transferring across convolution workloads but were not useful across operator types. Context relation representation allowed effective transfer across operator types.



(a) Optimization curves in wall clock time. (We set cuDNN v7, Tensorflow Lite and ARM ComputeLibrary v18.03 as the baselines for TITAN X, ARM A53 and ARM Mali-T860, respectively.)



(b) NVIDIA TITAN X Single Op

(c) ARM Cortex-A53 Single Op

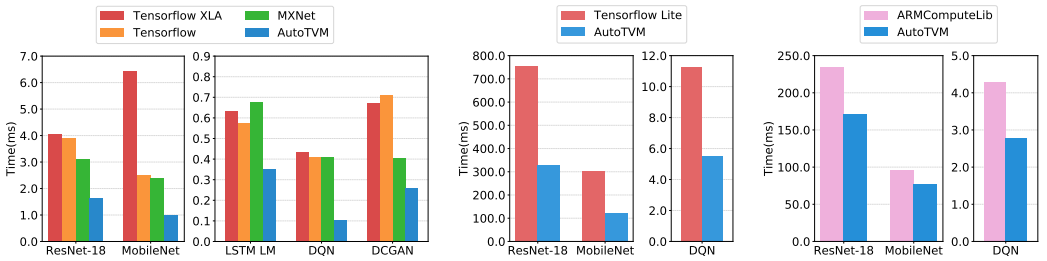
Figure 10: Single operator performance on the TITAN X and ARM CPU. (Additional ARM GPU (Mali) results are provided in the supplementary material.) We also included a weight pre-transformed Winograd kernel [24] for 3×3 conv2d (AutoTVM PT). AutoTVM generated programs that were competitive with hardware-specific libraries.

target domain (transfer across operator types). Results (Figure 9) show the need for more invariance when domains are farther apart. Using our transferable feature representation, our model generalized across different input shapes and operator types. We also ran a preliminary study on transfer from an ARM Mali GPU to an ARM Cortex-A53 (Figure 9d), showing that the proposed representation enabled transfer across devices. Developing an invariant feature representation poses a difficult problem worthy of additional research.

6.3 End-to-End Evaluation

Thus far, our evaluation has focused on specific design choices in our framework. We now segue to the natural follow-up question: can learning to optimize tensor programs improve real-world deep learning systems on diverse hardware targets? We call our framework AutoTVM. We compared our approach to existing DL frameworks backed by highly engineered hardware-specific libraries on diverse hardware back-ends: a server class GPU, an embedded CPU, and a mobile GPU. Note that AutoTVM performs optimization and code generation *with no external operator library*.

We first evaluated single-operator optimization against baselines that used hardware-specific libraries. The baselines were: cuDNN v7 for the NVIDIA GPU, TFLite(commit: 7558b085) for the Cortex-A53, and the ARM Compute Library (v18.03) for the ARM Mali GPU. We also in-



(a) NVIDIA TITAN X End2End (b) ARM Cortex-A53 End2End (c) ARM Mali-T860 End2End

Figure 11: End-to-end performance across back-ends. [2]AutoTVM outperforms the baseline methods.

cluded TensorComprehensions (commit: ef644ba) [41] as an additional baseline for the TITAN X [1] TensorComprehensions used 2 random seeds $\times 25$ generations $\times 200$ population for each operator, and padding was removed (TC does not yet support padding). The results are shown in Figure 10. AutoTVM generated high-performance tensor programs across different hardware back-ends.

Further, we embedded our framework into an existing DL graph compiler stack and performed end-to-end workload evaluation. We evaluated real world end-to-end DL inference workloads, including ResNet [14], MobileNet [16], LSTM Language Model [44], Deep Q Network (DQN) [27], and Deep Convolutional Generative Adversarial Networks (DCGAN) [31]. Our baselines were: MXNet (v1.1), Tensorflow (v1.7) for the GPU, TFLite(commit: 7558b085) for the Cortex A53, and ARM Compute Library (v18.03) for the ARM Mali GPU. Results are summarized in Figure 11. AutoTVM improved end-to-end performance by $1.2\times$ to $3.8\times$. These improvements were due to both tensor program optimization and operator fusion optimizations; the latter would otherwise be impossible if we used libraries with a limited set of operators.

7 Discussion and Conclusion

We presented AutoTVM: a machine learning-based framework that automatically optimizes the implementation of tensor operators in deep learning systems. Our statistical cost model allows effective model sharing between workloads and speeds up the optimization process via model transfer. The positive experimental results of this new approach show promise for DL deployment. Beyond our solution framework, the specific characteristics of this new problem make it an ideal testbed for innovations in related areas, such as neural program modeling, Bayesian optimization, transfer learning, and reinforcement learning. On the systems side, learning to optimize tensor programs can enable more fused operators, data layouts, and data types across diverse hardware back-ends—crucial to improving DL systems. Our framework can be found at https://tvm.ai.

Acknowledgement

We would like to thank members of Sampa, SAMPL and Systems groups at the Allen School for their feedback on the work and manuscript. This work was supported in part by a Google PhD Fellowship for Tianqi Chen, ONR award #N00014-16-1-2795, NSF under grants CCF-1518703, CNS-1614717, and CCF-1723352, and gifts from Intel (under the CAPA program), Oracle, Huawei and anonymous sources.

References

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin

[1] According to personal communication [40], TC is not yet intended for use in compute-bound problems. However, it still provides a good reference baseline for inclusion in the comparison.

[2] DCGAN and LSTM were not reported on A53 and Mali because they are not yet supported by baseline systems.

Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[2] Amit Agarwal, Eldar Akchurin, Chris Basoglu, Guoguo Chen, Scott Cyphers, Jasha Droppo, Adam Eversole, Brian Guenter, Mark Hillebrand, Ryan Hoens, Xuedong Huang, Zhiheng Huang, Vladimir Ivanov, Alexey Kamenev, Philipp Kranen, Oleksii Kuchaiev, Wolfgang Manousek, Avner May, Bhaskar Mitra, Olivier Nano, Gaizka Navarro, Alexey Orlov, Marko Padmilac, Hari Parthasarathi, Baolin Peng, Alexey Reznichenko, Frank Seide, Michael L. Seltzer, Malcolm Slaney, Andreas Stolcke, Yongqiang Wang, Huaming Wang, Kaisheng Yao, Dong Yu, Yu Zhang, and Geoffrey Zweig. An introduction to computational networks and the computational network toolkit. Technical Report MSR-TR-2014-112, August 2014.

[3] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018.

[4] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, and Yoshua Bengio. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[5] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113. ACM, 2008.

[6] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. Learning to rank using gradient descent. In *Proceedings of the 22Nd International Conference on Machine Learning*, ICML '05, pages 89–96, New York, NY, USA, 2005. ACM.

[7] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. ACM.

[8] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, , and Zheng Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *Neural Information Processing Systems, Workshop on Machine Learning Systems (LearningSys'15)*, 2015.

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018.

[10] Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. *CoRR*, abs/1802.03691, 2018.

[11] J.H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of Statistics*, 29(5):1189–1232, 2001.

[12] M. Frigo and S. G. Johnson. Fftw: an adaptive software architecture for the fft. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, volume 3, pages 1381–1384 vol.3, May 1998.

[13] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D. Sculley. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, pages 1487–1495. ACM, 2017.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. *arXiv preprint arXiv:1603.05027*, 2016.

[15] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 556–571, New York, NY, USA, 2017. ACM.

[16] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[17] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proceedings of the 5th International Conference on Learning and Intelligent Optimization*, LION'05, pages 507–523, Berlin, Heidelberg, 2011. Springer-Verlag.

[18] Frank Hutter, Lin Xu, Holger Hoos, and Kevin Leyton-Brown. Algorithm runtime prediction: Methods and evaluation (extended abstract). In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 4197–4201, 2015.

[19] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[20] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, October 2017.

[21] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. *CoRR*, abs/1712.01208, 2017.

[22] Andreas Krause and Daniel Golovin. Submodular function maximization. In *Tractability: Practical Approaches to Hard Problems*. Cambridge University Press, February 2014.

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, pages 1097–1105. 2012.

[24] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 4013–4021, 2016.

[25] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Efficient hyperparameter optimization and infinitely many armed bandits. *CoRR*, abs/1603.06560, 2016.

[26] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, pages 2430–2439, 2017.

[27] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[28] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.*, 35(4):83:1–83:11, July 2016.

[29] George L Nemhauser, Laurence A Wolsey, and Marshall L Fisher. An analysis of approximations for maximizing submodular set functions—i. *Mathematical Programming*, 14(1):265–294, 1978.

[30] Shoumik Palkar, James J. Thomas, Deepak Narayanan, Anil Shanbhag, Rahul Palamuttam, Holger Pirk, Malte Schwarzkopf, Saman P. Amarasinghe, Samuel Madden, and Matei Zaharia. Weld: Rethinking the interface between data-intensive applications. *CoRR*, abs/1709.06416, 2017.

[31] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[32] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[33] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, Jan 2016.

[34] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'12, pages 2951–2959, USA, 2012.

[35] Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Prabhat Prabhat, and Ryan P. Adams. Scalable bayesian optimization using deep neural networks. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pages 2171–2180, 2015.

[36] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Lift: A functional data-parallel ir for high-performance gpu code generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 74–85, Piscataway, NJ, USA, 2017. IEEE Press.

[37] Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Hassan Chafi, Michael Wu, Anand R. Atreya, Kunle Olukotun, Tiark Rompf, and Martin Odersky. Optiml: An implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, pages 609–616, USA, 2011.

[38] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, pages 3104–3112, Cambridge, MA, USA, 2014. MIT Press.

[39] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

[40] Nicolas Vasilache. personal communication.

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

[42] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, January 2013.

[43] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[44] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.

A Supplementary Materials

A.1 Additional Experimental Results

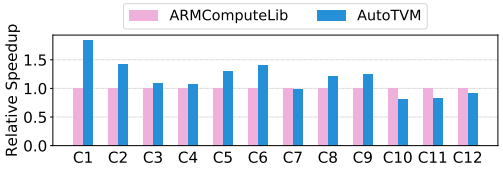


Figure 12: Single Operator Performance on Mali T860MP4

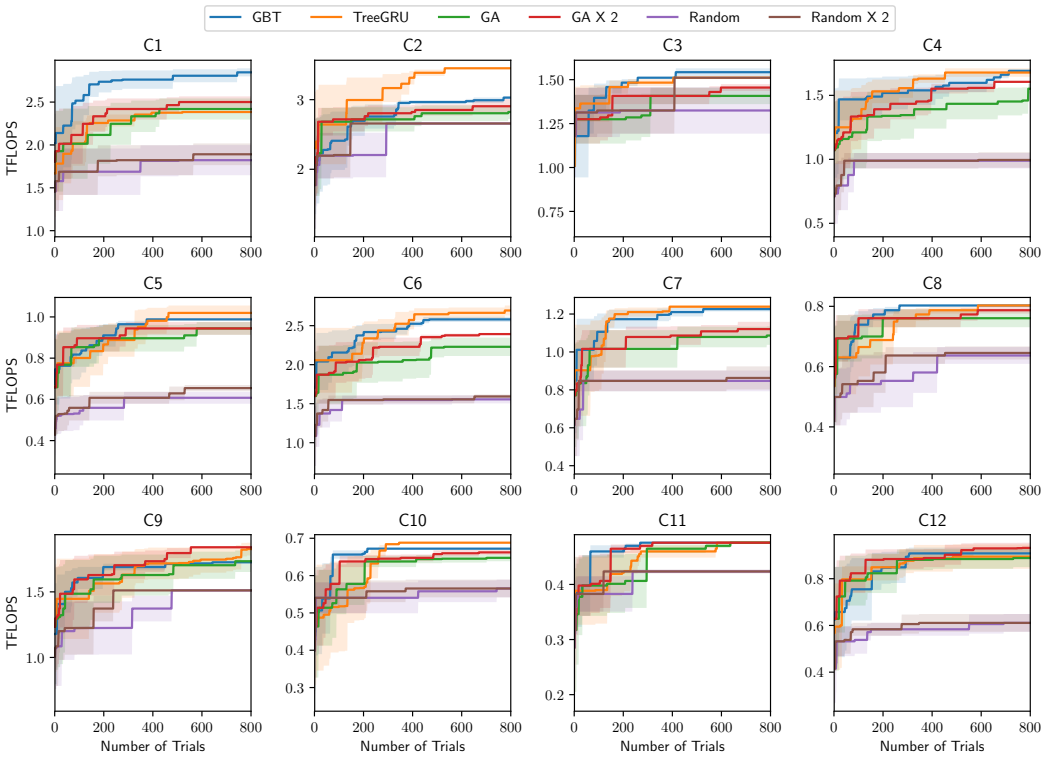


Figure 13: Effectiveness of cost model on all conv2d operators in ResNet-18.

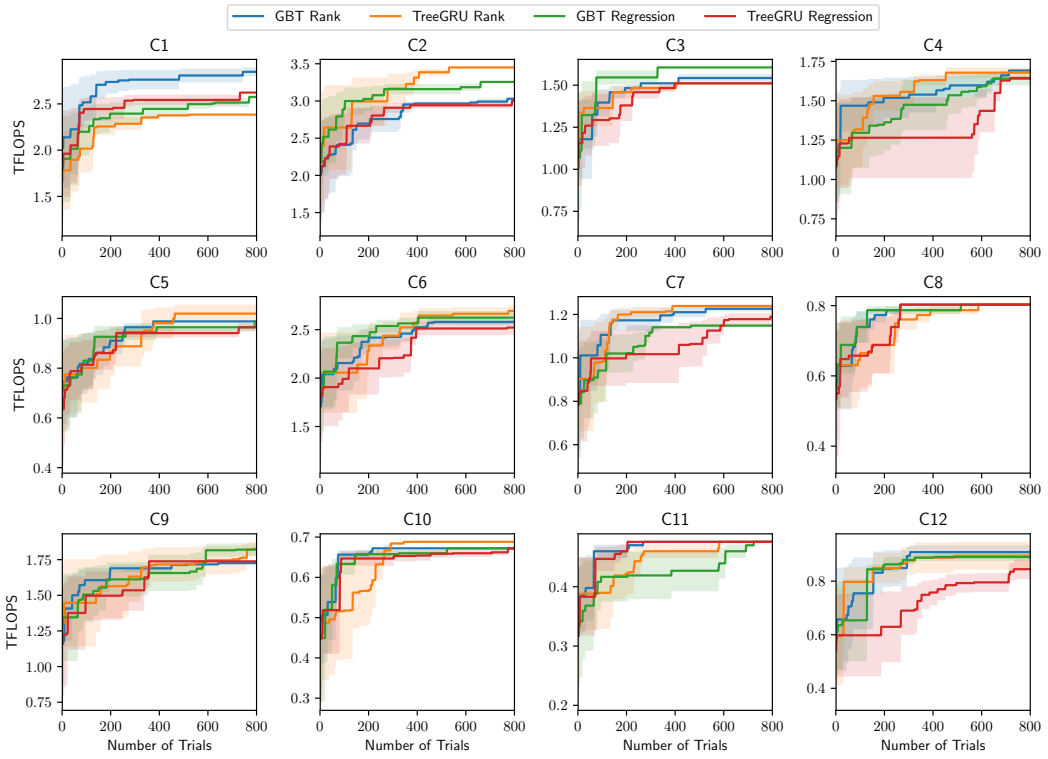


Figure 14: Impact of objective function of cost model on all conv2d operators in ResNet-18.

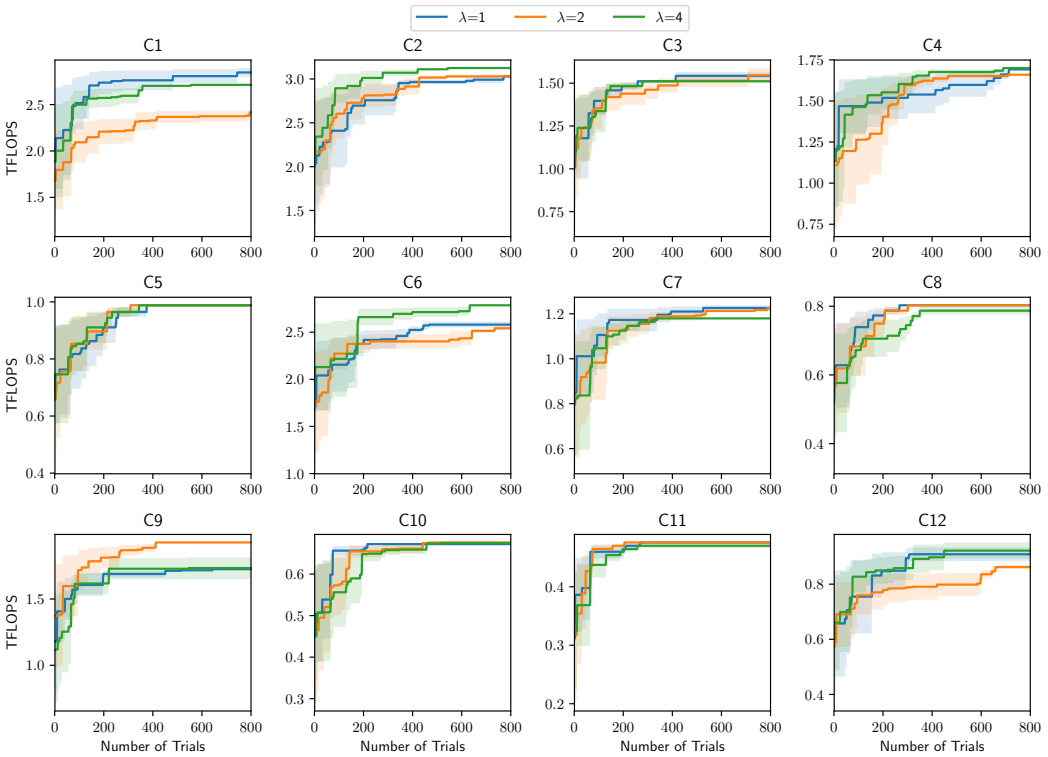


Figure 15: Impact of diversity aware exploration on all conv2d operators in ResNet-18.

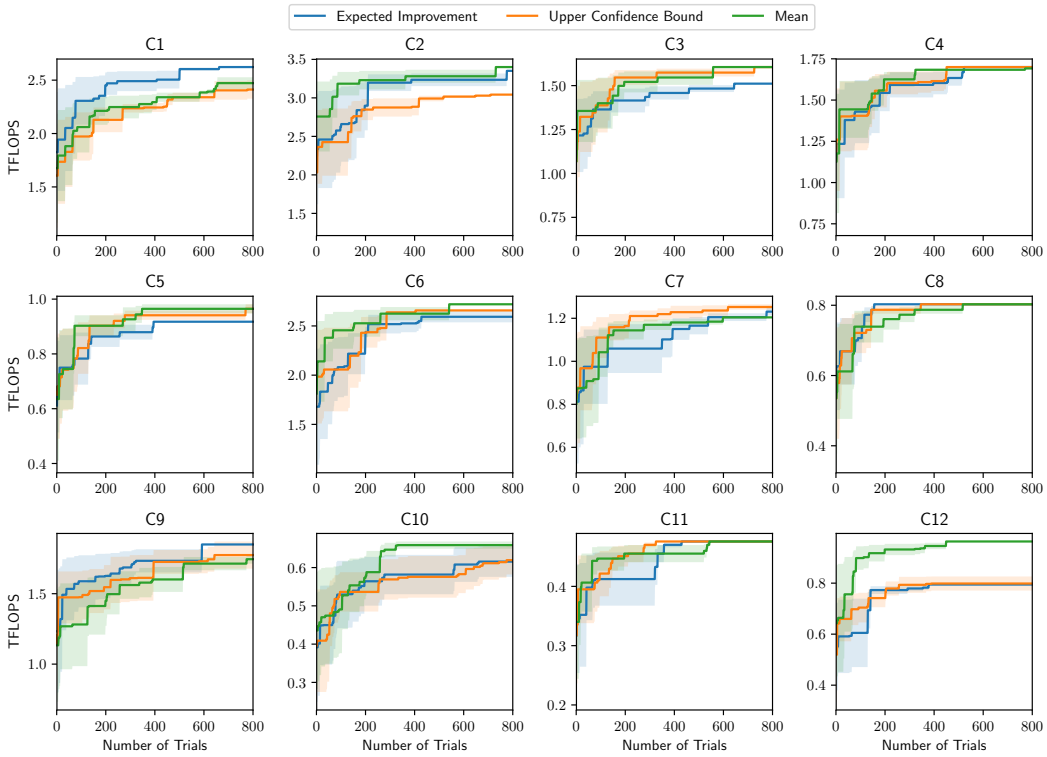


Figure 16: Impact of uncertainty aware acquisition function on all conv2d operators in ResNet-18.

A.2 Summary of Loop Features

A.2.1 Loop Context

We extract loop context for every loop variable. The loop context contains loop attributes and the access patterns for all touched inner buffers.

Feature Name		Description
length		The length of this loop
annotation		One-hot annotation of this loop (can be vectorize, unrolled, paralleled, ...)
top-down		The product of the lengths of outer loops
bottom-up		The product of the lengths of inner loops
access pattern (for every buffer)	touch count	The number of touched elements
	reuse ratio	Reuse ratio of this buffer (= bottom-up / touch count)
	stride	Coefficent of this loop varialbe in the index expression

Table 2: Listing of loop context feature

A.2.2 Relation Feature

First we pick the longest chain from the AST. Then we extract loop context features for the loop variables in this chain. We compute two pairs of relation : touch count vs reuse ratio and touch count vs top-down.

A.3 Experiment Configuration

Hyperparameter	Value	Description
b_{GBT}	64	batch size of planning in GBT
$b_{TreeGRU}$	64	batch size of planning in TreeGRU
emb_dim	128	dimension of loop variable embedding in TreeGRU
$hidden\_size$	128	hidden size of GRU cell in TreeGRU
n_{sa}	128	number of Markov chains in parallel simulated annealing
$step_{sa}$	500	maximum steps of one simulated annealing run