Learning Local Advantage Functions for Generalizable Graph Optimizations

Yifan Wu[*], **Yanqi Zhou, Phitchaya Mangpo Phothilimthana, Hanxiao Liu, Sudip Roy, Azalia Mirhoseini**
Google Brain
Mountain View, CA 94043
yw4@andrew.cmu.edu, {yanqiz,mangpo,hanxiaol,sudipr,azalia}@google.com

Abstract

Machine learning compilers rely on making optimized decisions in order to generate efficient code for a given computation graph. Many of these decision making processes can be formulated as graph optimization problems. The solution to these graph optimization problems is typically computed based on human designed heuristics. Learning/search-based methods have been recently investigated to improve upon or remove the need of human designed heuristics. However, existing methods that can reliably provide high quality solutions require iterative evaluations on real hardware. The evaluations can be costly especially for large graphs, making these methods infeasible to be deployed in production. To reduce or eliminate the evaluation cost, learning graph optimization strategies that can generalize across graphs is desirable. In this work, we propose learning local advantage functions for generalizable compiler graph optimizations. The learned model can be trained offline with supervised learning on massive amount of training data and then used to guide the search of optimal decisions on previously unseen graphs. We demonstrate the effectiveness of our approach on the operation fusion task and discuss several challenges we encountered in practice.

1 Introduction

Training complex neural networks usually requires hardware accelerators like GPUs and TPUs. The hardware acceleration relies on machine learning compilers [8, 15, 11, 4], to generate machine code that runs efficiently given high level computation graphs. To generate efficient machine code, the machine learning compiler needs to make many optimized decisions in different stages of compilation including graph rewriting [17, 10], device placement [12, 13, 18], operation fusion [1], layout assignment, scheduling, and tiling of tensors [5]. For many of these optimization stages, the decision making process can be formulated as combinatorial graph optimization problems with an objective to minimize the computation run time on the target accelerator. These optimization problems often have an exponentially large search space. Current ML compilers usually fall back to domain-knowledge based heuristic solutions that explore only a small fragment of this search space, and therefore settle for sub-optimal solutions that can found in reasonable search time.

Recent works[12, 13, 7, 18] have used reinforcement learning (RL) and search successfully applied on some of these compiler optimization tasks, and have achieved state-of-the-art results by outperforming heuristic solutions. However, to effectively explore the large search space, these methods require massive number of evaluations. In the absence of a reliable cost model, these evaluations require actual runs on hardware and can be very expensive, often taking few minutes for large computational graphs. The expensive evaluation can limit the RL or search quality as obtaining a single sample can take very long or consume too much computational resources.

[*]Carnegie Mellon University. Work performed while an intern at Google.

NeurIPS 2020 Workshop on Machine Learning for Systems

One of the key ways to overcome this limitation is to ensure that the learned approach generalizes well across different computational graphs and for different optimization tasks. Current approaches at generalization can be broadly categorized into three main categories. First, by using a learned cost model. An accurate cost model can be hard to learn as the run time (label) for large graphs is of significant higher scale than small graphs. Neural network models lack the ability to do extrapolation in such cases. Second, by directly learning a shared policy across graphs. Although the distribution shift in graph size can be addressed by having size-invariant parameterized models, e.g. graph neural networks, the learned policy tends to under-perform on unseen graphs in practice since zero-shot learning can be hard. Finally, by pre-training offline then fine-tuning on specific graphs online. This is a widely applied approach for the purpose of improving sample efficiency on new tasks in both prediction (e.g., supervised learning) and decision making (e.g. RL). Each of the fore-mentioned approaches to generalization does not work well in the presence of distributional shifts, and requires task-specific training, which together make it hard to reuse for newer tasks and environments.

In this paper, we focus on developing methods that can achieve generalization across computational graphs in the presence of substantial distribution shift: we aim at learning from experience on relatively small computation graphs and testing on unseen (and potentially much larger) graphs. Building on the empirical observation that search based strategies often outperform RL approaches on harder tasks such as operation fusion, our approach combines the learned guided exploration strategy with progressive search. Our approach is to **learn single node advantage functions** from existing graphs and use the learned model to **guide exploration** on new graphs for improving sample efficiency. Our experiments on the operation fusion task show that our guided exploration is able to help a progressive search agent find similarly good or better solutions within less than 50% of the number of samples compared to non-guided search.

2 Related Work

RL for Graph Optimization Reinforcement learning has been used for optimizing device placement [12, 7, 13]. Hierarchical Device Placement (HDP) [13], Spotlight [7], and Placeto [2] progressively generate decisions on a per-node basis, which is computationally inefficient on large graphs. NeuRewriter [6] is a generic rewriting system that uses RL to solve combinatorial optimization problems. In terms of generalization across graphs, all of these methods can be used in a pretrain-then-finetune manner or combined with any guided exploration strategies, and thus orthogonal to our work.

ML Compiler Optimization Machine learning has been applied to optimize the execution time of tensor computation graphs [14, 16, 3, 5, 18]. Among these works, GO [18] achieves generalization by jointly training a policy on multiple graphs then finetuning on unseen graphs. In their work, they randomly sample a single hold-out graph from a set of graphs as the unseen graph to test for generalization, which suffers less distribution shift compared to our setting. The generalization strategy of GO follows the pretrain-then-finetune pattern while we explore the direction of using learning to guide exploration. REGAL [14] leverages a learned policy to guide search on new graphs for generalization. The difference between REGAL and our work is that (i) REGAL performs evaluations on a performance model while we consider evaluations on real hardware and (ii) using a learned policy to guide search requires online pretraining (using RL or contextual bandits) on existing graphs, which is resource costly and slow if evaluations are done on real hardware. In our approach, the local advantage function can be learned offline on pre-collected data in a supervised way, thus retraining a model does not requires recollecting data from real hardwares. Also supervised learning is a more stable (e.g., less sensitive to hyperparameter tuning) technique compared to contextual bandits or RL.

3 Method

Given a graph $G = (V, E)$, a graph optimization problem can be expressed as finding the optimal configuration for a set of configurable nodes $V_c \subset V$. Each configuration can be written as a vector $\vec{y} = [y_v \in \mathcal{Y} : v \in V_c]$ where $\mathcal{Y}$ is a discrete decision space, e.g. $\mathcal{Y} = \{1, ..., K\}$. In compiler optimization tasks we also have access to additional features $X = [x_v : v \in V]$, such as operation type, input and output tensor shape, etc. Decisions can be made conditioned on the graph structure as well as these node features. The goal is to minimize some cost function $\texttt{cost}(\vec{y}; G, X)$. The cost

function does not have an analytical form and can only be queried by actually evaluating proposed configurations.

We propose to learn the single node advantage function:

$$f\left(y_v = k; \vec{y}_{v^-}, G, X\right) = \texttt{cost}(y_v = k, \vec{y}_{v^-}; G, X) - \texttt{cost}(y_v = 1, \vec{y}_{v^-}; G, X)\,,$$

for each $k \in \mathcal{Y}$. Here $\vec{y}_{v^-}$ denotes the configuration on other nodes except y_v. The advantage function outputs the difference in cost by flipping the decision at a single node given an existing configuration. This function is (i) easy to train with supervised learning using offline collected data and (ii) usually invariant of graph size in compiler optimization tasks, enabling generalization from smaller graphs to large ones. f can be parameterize as graph neural networks like GraphSAGE [9].

The learned local advantage function can then be applied to expedite search on unseen graphs. Although the potential usage of the local advantage function in search can be quite flexible, in our experiments we use the predictions to prioritize important nodes (where the mutation of node decisions results in large cost differences) for mutation in progressive search, and would leave other possible integration as future work. For example, one may noticed that knowing the single node advantage function may eliminate the need of real evaluations since we can compare any two configurations by build a path between the two configurations by flipping the decision on a single node at each time. However, this required a highly accurate learned advantage function. In our experiments, we found that when generalizing to new graphs, the quality of the learned f if not good enough to totally replace real evaluations. As a result, we first investigate the direction of using a learned f to accelerate search, i.e. reduce the number of evaluations, while leaving eliminating real evaluations as a future direction.

4 Experiments

We experiment on the operation fusion task [1] in the Tensorflow-XLA compiler, where at each node the compiler needs make a binary decision ($\mathcal{Y} = \{1, 2\}$) to determine whether to fuse the operation into its consumer when running on TPUs. In the following we describe how we utilize local advantage learning to achieve generalization on this task and how we addressed several challenges in our experiments.

4.1 Data Collection

The datasets are collected by running Gibbs sampling on a set of training graphs. Gibbs sampling on a graph $G = (V, E)$ starts from a given configuration, and iterates over the indices of nodes $v = 1, ..., |V|$ and sample from $p(y_v|\vec{y}_{v^-})$, where the conditional distribution of the decision y_v on specific node v conditioned on the other nodes $\vec{y}_{v^-}$ is defined by $p(y_v = k|\vec{y}_{v^-}) \propto \exp(-\texttt{cost}(y_v = k, \vec{y}_{v^-})/\tau)$, where τ is a positive temperature parameter. During Gibbs sampling, for each node v to be mutated, all possible decisions $y_v \in \mathcal{Y}$ will be evaluated in order to compute $p(y_v = k|y_{v^-})$. Therefore, the configuration-cost pairs collected during Gibbs sampling is sufficient for training the local advantage function in a supervised way.

4.2 Learning

Although given the collected data described above, learning the advantage function f is simply a supervised regression problem, there are still multiple challenges handling real-world compiler data. We list the major challenges we found with the operation fusion task and discuss how we addressed these problems as follows.

Label scale We found that the label (node advantage) scale can be drastically different across graphs. For example, among the benchmarks we are working on, the label scale in GraphNets is 10 times larger than the label scale in DeepRank. With these labels unchanged, the learned function will only fit the samples in GraphNets as the labels in DeepRank are just like noise in GraphNets. To resolve this, we look into the direction of predicting the log scale of the labels, e.g. predict $\text{sign}(\text{adv}) * \log(1 + |\text{adv}|/\sigma)$, where the σ is a graph-dependent quantity that reflects the scale of single node advantage for each graph. We simply take σ to be the median of the absolute value of the advantages among the samples collected for a single graph. With this label transformation, the new

labels are of the same range across different graphs and the effective labels range much larger than the noise level.

Imbalanced data We also notice that, among all samples collected for a single graph, only less than 10% of the samples come from significant decisions while the remaining ones are non-significant (the flipping of decisions do not significantly affect run time). This affects both training efficiency and generalization. To resolve this, we perform a prioritized sampling when sampling mini-batches: for 90% percent of the time we sample the data with the top 10% absolute value of advantages. This allows us to focus on the important region of the graph when training the advantage function while not over-fitting to the noisy insignificant nodes.

Advantage Direction We also find that the direction (sign) of the advantages are hard to predict, so we choose to predict the absolute value of the advantage, which is still useful for accelerating search as we will show later.

4.3 Guided Search

We propose the following guided search procedure to utilize the learned advantage function. Each guided search step is performed as:

- Get the prediction of absolute advantage values for all nodes.
- Select the top $\alpha\%$ nodes according to their absolute advantages and randomly sample a single node from them.
- Randomly mutate a k-neighborhood of the selected node.
- Keep the better configuration for the next step.

The (none-guided) random search replaces the first two steps with selecting a node uniformly at random, which we found to be able to consistently find better configurations. Our hypothesis is that the learned advantage function contains information about which nodes are important and mutating important nodes (and their adjacent nodes) more often can reduce the amount of evaluations needed to search for a good configuration.

4.4 Evaluation

In our experiments, we first train a local advantage function on a set of 10 computation graphs (with $500 \sim 5000$ configurable nodes) and evaluate our guided search against (non-guided) random search (random search) on graphs that are not in the training set (testing generalization). To evaluate the ability of improving over human heuristics, we compare guided search and random search initialized with the default configuration by the Tensorflow-XLA compiler. Table 1 summarizes the results. The results show the guided search can find a better configuration or a similarly good configuration using less evaluations (samples) compared to random search.

Benchmark	Graph Size	Guided	#Samples	Random	#Samples
GraphNets	1532	18%	**1400**	18%	3500
Unet	2626	5%	**300**	5%	4500
DeepRank	532	5%	**2000**	5%	4500
MNasNet	5713	2%	**2100**	2.8%	3000
SSDModileNet	22176	**2.3%**	400	0.5%	100

Table 1: Comparing guided search v.s. random search when searching from the default configuration. Reported performance numbers are relative improvements over the default configuration. The #Samples column reports the number of evaluations needed to find such improved configurations. The Graph Size column reports the number of configurable nodes for each computation graph.

To evaluate the ability of replacing human heuristics, we also compare guided search and random search initialized with a random configuration. Figure 1 shows that our guided search consistently

achieve a better sample efficiency by a large margin compared to random search when initialing from a random configuration.

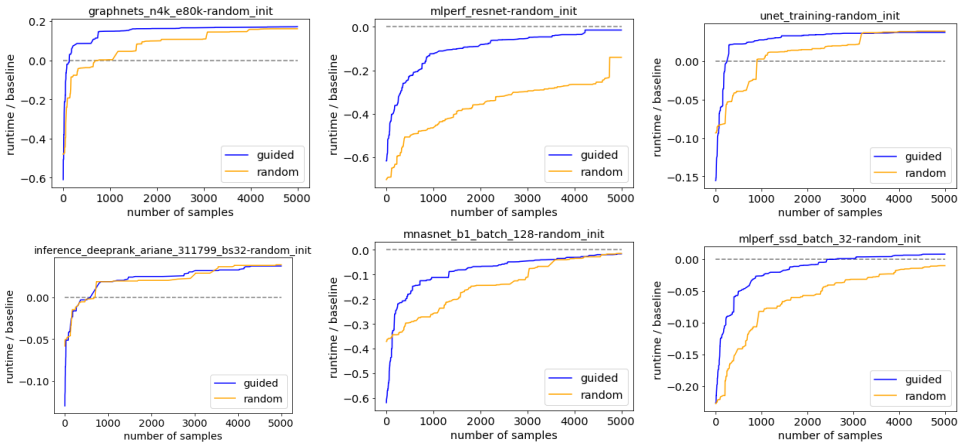

Figure 1: Comparing guided search v.s. random search when searching from a random configuration. Reported performance numbers are relative improvements over the default configuration.

5 Conclusion

We investigate the problem of developing compiler graph optimization strategies that can generalize across graphs in the presence of distribution shift. We propose to learn single-node advantage functions offline from pre-collected data through supervised learning then apply the learned advantage function to guide exploration when searching optimized configurations on new graphs. Our experiments show that, on the operation fusion task, our guided search method is able to optimize over random or default configurations, taking fewer evaluations than non-guided random search.

References

[1] Abdolrashidi, A., Xu, Q., Wang, S., Roy, S., and Zhou, Y. Learning to fuse. *NeurIPS ML for Systems Workshop*, 2019. URL `http://mlforsystems.org/assets/papers/neurips2019/learning_abdolrashidi_2019.pdf`.

[2] Addanki, R., Venkatakrishnan, S. B., Gupta, S., Mao, H., and Alizadeh, M. Placeto: Learning generalizable device placement algorithms for distributed machine learning. *CoRR*, abs/1906.08879, 2019. URL `http://arxiv.org/abs/1906.08879`.

[3] Andrew Adams, Karima Ma, L. A. R. B. T.-M. L. M. G. B. S. S. J. K. F. F. D. J. R.-K. Learning to optimize halide with tree search and random programs. *SIGGRAPH*, 2019.

[4] Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578–594, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-939133-08-3. URL `https://www.usenix.org/conference/osdi18/presentation/chen`.

[5] Chen, T., Zheng, L., Yan, E. Q., Jiang, Z., Moreau, T., Ceze, L., Guestrin, C., and Krishnamurthy, A. Learning to optimize tensor programs. *CoRR*, abs/1805.08166, 2018. URL `http://arxiv.org/abs/1805.08166`.

[6] Chen, X. and Tian, Y. Learning to perform local rewriting for combinatorial optimization. *NeurIPS*, abs/1810.00337, 2019. URL `http://arxiv.org/abs/1810.00337`.

[7] Gao, Y., Chen, L., and Li, B. Spotlight: Optimizing device placement for training deep neural networks. In Dy, J. and Krause, A. (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 1676–1684, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR. URL http://proceedings.mlr.press/v80/gao18a.html.

[8] Google. Xla: Optimizing compiler for tensorflow. 2018. URL https://tensorflow.org/xla.

[9] Hamilton, W., Ying, Z., and Leskovec, J. Inductive representation learning on large graphs. In *Advances in neural information processing systems*, pp. 1024–1034, 2017.

[10] Jia, Z., Thomas, J., Warszawski, T., Gao, M., Zaharia, M., and Aiken, A. Optimizing DNN Computation with Relaxed Graph Substitutions. In *Proceedings of the 2nd SysML Conference*, SysML '19, 2019.

[11] Lattner, C. and Pienaar, J. Mlir primer: A compiler infrastructure for the end of moore's law, 2019.

[12] Mirhoseini, A., Pham, H., Le, Q. V., Steiner, B., Larsen, R., Zhou, Y., Kumar, N., Norouzi, M., Bengio, S., and Dean, J. Device placement optimization with reinforcement learning. *ICML*, 2017. URL http://arxiv.org/abs/1706.04972.

[13] Mirhoseini, A., Goldie, A., Pham, H., Steiner, B., Le, Q. V., and Dean, J. A hierarchical model for device placement. *ICLR*, 2018.

[14] Paliwal, A., Gimeno, F., Nair, V., Li, Y., Lubin, M., Kohli, P., and Vinyals, O. REGAL: transfer learning for fast optimization of computation graphs. *CoRR*, abs/1905.02494, 2019. URL http://arxiv.org/abs/1905.02494.

[15] Rotem, N., Fix, J., Abdulrasool, S., Deng, S., Dzhabarov, R., Hegeman, J., Levenstein, R., Maher, B., Satish, N., Olesen, J., Park, J., Rakhov, A., and Smelyanskiy, M. Glow: Graph lowering compiler techniques for neural networks. *CoRR*, abs/1805.00907, 2018. URL http://arxiv.org/abs/1805.00907.

[16] Yuu Jinnai, Arash Mehrjou, K. C. A. M. A. L. T. E. R. T. S. P. J. A. F. Knossos: Compiling ai with ai, 2019.

[17] Zhihao Jia, Oded Padon, J. T. T. W. M. Z. A. A. Taso: optimizing deep learning computation with automatic generation of graph substitutions. *SOSP*, 2019. URL https://doi.org/10.1145/3341301.3359630.

[18] Zhou, Y., Roy, S., Abdolrashidi, A., Wong, D., Ma, P. C., Xu, Q., Liu, H., Phothilimtha, P., Wang, S., Goldie, A., Mirhoseini, A., and Laudon, J. Transferable graph optimizers for ml compilers. In *Advances in Neural Information Processing Systems*, 2020.