SIMULATING EXECUTION TIME OF TENSOR PROGRAMS USING GRAPH NEURAL NETWORKS

Jakub M. Tomczak[*], **Romain Lepert**[*] **& Auke Wiggers**[*]
Qualcomm AI Research[†]
Qualcomm Technologies Netherlands B.V.
{jtomczak, romain, auke}@qti.qualcomm.com

ABSTRACT

Optimizing the execution time of tensor program, *e.g.*, a convolution, involves finding its optimal configuration. Searching the configuration space exhaustively is typically infeasible in practice. In line with recent research using TVM, we propose to learn a surrogate model to overcome this issue. The model is trained on an acyclic graph called an abstract syntax tree, and utilizes a graph convolutional network to exploit structure in the graph. We claim that a learnable graph-based data processing is a strong competitor to heuristic-based feature extraction. We present a new dataset of graphs corresponding to configurations and their execution time for various tensor programs. We provide baselines for a runtime prediction task.

1 INTRODUCTION

Current deep learning frameworks, such as TensorFlow, PyTorch, allow to optimize a computational graph representation using, *e.g.*, auto differentiation and memory management (Abadi et al., 2016; Paszke et al., 2017). However, they do not tackle optimization of hardware-specific operator-level transformations, but rely on manually tuned and vendor-specific operator libraries. Thus, there is room to further improve a computational graph by optimizing transformations for specific hardware.

Recently, this gap has been filled by TVM, a compiler framework that allows both graph- and operator-level optimization in an end-to-end manner (Chen et al., 2018a). TVM specifies a *configuration* for an operator, *e.g.*, a specific way of performing a convolution, and compiles the resulting *tensor program* to a target hardware. As a consequence, for each new workload/operator, optimization over a new configuration space must be carried out. This results in a hard optimization problem, *e.g.*, for Nvidia GPU the search space of a single operator consists of more than 10^6 configurations.

Recent efforts overcome this issue by learning how to optimize tensor programs from data (Chen et al., 2018b). Instead of running an exhaustive search over an impractically large search space, a *surrogate model* is trained to predict runtime for a given configuration. This model is in turn used to select the configuration that minimizes the runtime. (Chen et al., 2018b) utilizes XGBoost (Chen & Guestrin, 2016) and TreeGRU (Tai et al., 2015) as surrogate models.

Contribution Similar to (Chen et al., 2018b), we represent a configuration of a tensor operator as an abstract syntax tree (AST) (Allamanis et al., 2017), and extract node features using TVM. We then train a Graph Neural Network (GraphNN) on the resulting graph as the surrogate model. We claim that GraphNNs are a good fit, as, crucially, they preserve the graph structure of the AST and allow propagating information among nodes. The contribution of the paper is threefold:

- We present a new problem for GraphNNs: predicting the execution time of tensor programs from their corresponding AST. For this purpose, we gathered a **new dataset** and we propose to use it as a new application in the GraphNN community.[1] This is the main contribution of the paper.

[*] All authors contributed equally.

[†] Qualcomm AI Research is an initiative of Qualcomm Technologies, Inc.

[1] The dataset will be released soon, please contact romain@qti.qualcomm.com for further details.

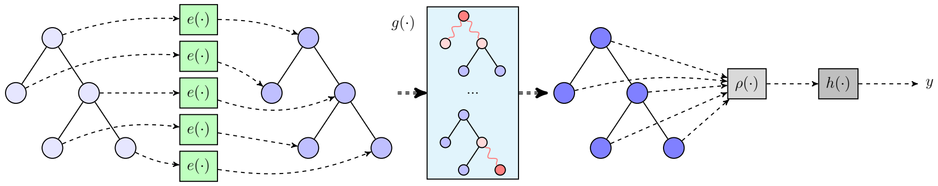

Figure 1: A schematic presentation of our approach: First, each node is transformed by a shared encoder (green rectangle), then a graph convolutional network is used to propagate information between nodes (cyan rectangle). Finally, all nodes are aggregated and a prediction y is made.

- We propose to use a graph neural network as a surrogate model of a compiler. We claim that it is important to use a learnable graph data transformation rather than a fixed feature extractor, *e.g.*, context relation features (Chen et al., 2018b).
- We perform experiments on the newly proposed dataset and provide baseline results for a cross-workload prediction task.

Related work GraphNNs have been proven to be powerful in many applications ranging from chemistry and life sciences (De Cao & Kipf, 2018; Duvenaud et al., 2015; Gonczarek et al., 2018; Zitnik et al., 2018) to social networks Davidson et al. (2018); Kipf & Welling (2017); Hamilton et al. (2017); Veličković et al. (2017), where graph inputs represent chemical compounds or social interactions among users. GraphsNNs find applications in other domains as well, such as geometric modeling (Bronstein et al., 2017), recommendation systems (Berg et al., 2017), relational data and knowledge graphs (Nickel et al., 2016; Schlichtkrull et al., 2018), and regression problems like chemical properties prediction (Duvenaud et al., 2015; Li et al., 2018) and traffic prediction (Yu et al., 2017). An interesting on-going research is on using these networks in generative settings (De Cao & Kipf, 2018; Jin et al., 2018; Simonovsky & Komodakis, 2018).

(Chen et al., 2018b) introduced the idea of learning a surrogate model using the TVM framework. We instead use GraphNNs in this context, and do not focus on runtime minimization yet.

2 Problem formulation

We consider the problem of learning an *execution time simulator*. That is, we aim to predict runtime of a configuration $x \in \mathcal{X}$ on the target hardware, where $\mathcal{X}$ is a discrete configuration space. Importantly, we represent x as a graph corresponding to the abstract syntax tree (AST) of the configuration. Let us define the runtime as a function $y = f(x)$, $y \in \mathbb{R}_+$. The function f can be queried, however, its analytical form is unknown. Our goal is to learn a surrogate model of the function f, denoted by $\hat{f}$, such that we minimize a loss function $\ell(y, \hat{f}(x))$ (e.g., ℓ_1-norm or, equivalently, the logarithm of the Laplacian distribution). Assuming a collection of measurements $\mathcal{D} = \{(x_n, y_n)\}_{n=1}^{N}$, a straightforward way to learn $\hat{f}$ is to minimize the objective $\mathcal{L}(\hat{f}; \mathcal{D}) = \frac{1}{N}\sum_{n=1}^{N} \ell\big(y_n, \hat{f}(x_n)\big)$.

3 Dataset

General information We collect data for the operators defined in a ResNet18 (He et al., 2016). The target hardware is an Intel Xeon CPU E5-1620 v4 processor @ 3.50GHz, with x86-64 instruction set. The ResNet18 architecture defines twelve unique convolution workloads, *i.e.*, there are twelve parameterizations for the convolution operators in the network. We show these in Table 1. For example, for workload 'C1', the convolution has 3 input feature maps of size 224 × 224, and has 64 output channels. We also show the number of configurations (# configs) for each workload for the x86 target hardware. We performed all measurements using TVM v0.5 (Chen et al., 2018a).

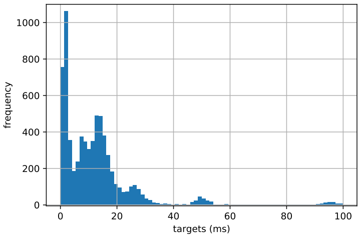

Figure 2: The distribution of targets.

Table 1: Parameters for each unique Conv2D workload in a ResNet18 for x86 CPU target.

Workload	H	W	C$_{in}$	C$_{out}$	kernel	stride	padding	dilation	# configs
C1	224	224	3	64	(7, 7)	(2, 2)	(3, 3)	(1, 1)	252
C2	56	56	64	64	(3, 3)	(1, 1)	(1, 1)	(1, 1)	784
C3	56	56	64	64	(1, 1)	(1, 1)	(1, 1)	(1, 1)	784
C4	56	56	64	128	(2, 2)	(1, 1)	(1, 1)	(1, 1)	672
C5	56	56	64	128	(2, 2)	(1, 1)	(1, 1)	(1, 1)	672
C6	28	28	128	128	(1, 1)	(1, 1)	(1, 1)	(1, 1)	768
C7	28	28	128	256	(2, 2)	(1, 1)	(1, 1)	(1, 1)	576
C8	28	28	128	256	(2, 2)	(1, 1)	(1, 1)	(1, 1)	576
C9	28	28	256	256	(1, 1)	(1, 1)	(1, 1)	(1, 1)	648
C10	28	28	256	512	(2, 2)	(1, 1)	(1, 1)	(1, 1)	360
C11	28	28	256	512	(2, 2)	(1, 1)	(1, 1)	(1, 1)	360
C12	28	28	512	512	(1, 1)	(1, 1)	(1, 1)	(1, 1)	400

Feature representation The dataset contains 6,852 configurations. For each configuration, we extract the corresponding AST that is represented by: an adjacency matrix A, a feature matrix F, the node types (*e.g.*, `for` statement, hardware instructions) G for every node in the graph. We save these matrices and the corresponding measured execution time y as tuples (A, F, G, y). The feature extraction procedure follows the one for loop context features presented in (Chen et al., 2018b).

Distribution of target runtimes The distribution of execution times do not match a normal distribution, but instead resembles a mixture of Gaussians (see Figure 2). This poses two important challenges: (i) the simulator needs to learn a representation for all modes, (ii) generalizing to components of low probability is troublesome if these configurations correspond to unseen workload.

4 Graph neural networks as a simulator

In general, we can build a simulator $\hat{f}$ in a similar manner as it is presented in (Zaheer et al., 2017): $\hat{f}(x) = h\Big(\rho\big[g\big(e(x_1), e(x_2), \dots, e(x_2)\big)\big]\Big)$. The surrogate model $\hat{f}$ consists of four components:

- A function $e(\cdot)$ that *encodes* each node represented by raw features to a vector of a fixed size. Weights of the encoder are shared across nodes.
- A *feature propagation function* $g(\cdot)$ that ensures feature information is propagated across nodes.
- An *aggregation function* $\rho[\cdot]$ that combines information from all nodes into a fixed-size vector. Typically, it is implemented using *sum* or *mean*.
- A *prediction function* $h(\cdot)$ that takes as input the fixed-size aggregated feature vector and predicts a scalar runtime y.

See Figure 1 for a visual representation of the surrogate model (the runtime simulator). Note that while we implement the functions above as neural networks in order to ensure that the surrogate model can be trained end-to-end using backpropagation, the four-stage framework allows us to plug in other functions as well, as long as in- and output-constraints are met. Importantly, the propagation function is implemented using graph convolutions (Duvenaud et al., 2015; Kipf & Welling, 2017).

5 Experiments

5.1 Setup

We investigate two classes of network architectures on the extracted dataset. The first class consists of fully-connected layers and does not propagate information among nodes before aggregation. We refer to this approach as MLP. The second class adds the propagation of the information using a graph convolutional network (GCN). In both classes, the encoding function $e(\cdot)$ and the prediction function $h(\cdot)$ are multi-layer perceptrons with ReLU activation. In all models we choose the aggregation operation $\rho[\cdot]$ to be the average. Additionally, both network types encode the *node type* using a learned embedding of 32-dimensional vectors, and concatenate the resulting embedding vector to

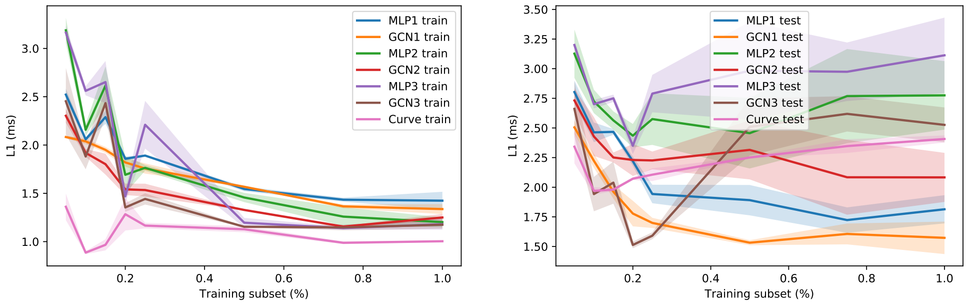

Figure 3: The final evaluation using ℓ_1 loss: (*left*) on training data, (*right*) on test data.

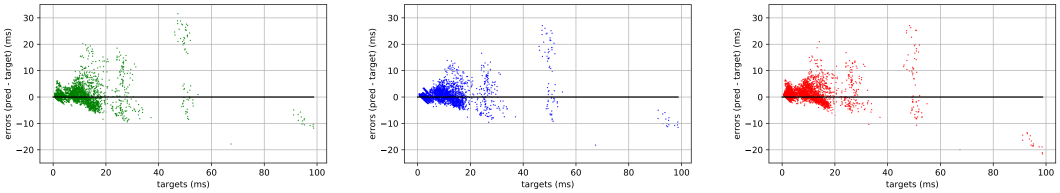

Figure 4: Scatter plots of errors (y-axis) for targets (x-axis) on the test set where only 20% of training data is available: (*left*) MLP, (*center*) GCN, (*right*) Curve.

the node features. The prediction function is composed of two fully-connected layers and the final layer has no activation function. Finally, the prediction function is composed of two fully-connected layers and the final layer has no activation function. For comparison, we also run a fixed feature extractor, namely, *context relation features* (Chen et al., 2018b), with a learnable predictor on top. We refer to this approach as ”Curve”. For more details, please see Table 2 in the supplementary material. During training we use the Huber loss that is a composition of the ℓ_1 and ℓ_2 losses, and ℓ_1 for final evaluation. We use DGL package (Wang et al., 2018) for implementing graph convolutions.

We aim at evaluating the **cross-workload generalization** capability of the surrogate model. Such a scenario corresponds to a real-life situation where we do not have access to all workloads. In the experiment, workloads {C1, C2, C4, C8, C9, C12} are taken as given data and the remaining workloads are used as test data. The rationale is to pick as varied workloads as possible. We split given configurations into 80% of training data and 20% of validation data. Further, we consider a couple of cases where we have a limited access to training data, namely, we learn models on 5%, 10%, 15%, 20%, 25%, 50%, 75% and, for completeness, 100% of the training set. This limitation imitates the real-life situation where we do not have measurements for all configurations.

5.2 Results and discussion

We present a final evaluation on 5 cases with different training datasets in Figure 3. We observe that GCN-based network architectures generalize to the unseen test data better than the models without the graph convolution component. The Curve model performs on par with the best performing GCN for $5\% - 15\%$ of training data, however, it generalizes poorly if more training data is available. To further inspect the performance of the three methods we present differences between predictions and targets in Figure 4. A closer inspection reveals that the GCN makes smaller mistakes in general and, most importantly, for targets between 0 and 20. Note that accurate prediction of target in $[0, 20]$ is critical since the goal of TVM is to identify a configuration with the smallest target execution time. We conjecture that the main reason for this difference is that the GCN is able to better exploit local structure, as the MLP can only share information among nodes after the aggregation function has been applied. Additionally, the results indicate that generalization to unseen workloads by training on similar workloads is possible to some extent, which shows that the found representations likely lend themselves well to transfer learning.

6 Final remarks

In this paper, we have presented a new dataset for learning the runtime of tensor program configurations represented as graphs. We provided three baselines, namely, a neural network with mean as the aggregation function, a GraphNN-based network that propagates information among nodes before the aggregation, and a neural network trained on context relational features representing a whole AST. Additionally, we showed that the GraphNN-based approach allows to obtain competitive results to the fixed feature extractor. We believe that this new dataset and the presented task will attract attention of the GraphNN community to the problem of learning tensor programs.

Acknowledgments

We would like to thank Jinwon Lee for fruitful discussions and helpful remarks.

References

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *Symposium on Operating Systems Design and Implementation)*, pp. 265–283, 2016.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

Rianne van den Berg, Thomas N Kipf, and Max Welling. Graph convolutional matrix completion. *arXiv preprint arXiv:1706.02263*, 2017.

Michael M Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017.

Tianqi Chen and Carlos Guestrin. XGBoost: A Scalable Tree Boosting System. In *KDD*, pp. 785–794, 2016. doi: 10.1145/2939672.2939785. URL `http://doi.acm.org/10.1145/2939672.2939785`.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: end-to-end optimization stack for deep learning. *arXiv preprint arXiv:1802.04799*, pp. 1–15, 2018a.

Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *NeurIPS*, pp. 3393–3404, 2018b.

Tim R Davidson, Luca Falorsi, Nicola De Cao, Thomas Kipf, and Jakub M Tomczak. Hyperspherical variational auto-encoders. *UAI*, 2018.

Nicola De Cao and Thomas Kipf. Molgan: An implicit generative model for small molecular graphs. *arXiv preprint arXiv:1805.11973*, 2018.

David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P Adams. Convolutional networks on graphs for learning molecular fingerprints. In *NIPS*, pp. 2224–2232, 2015.

Adam Gonczarek, Jakub M Tomczak, Szymon Zaręba, Joanna Kaczmar, Piotr Dąbrowski, and Michał J Walczak. Interaction prediction in structure-based virtual screening using deep learning. *Computers in Biology and Medicine*, 100:253–258, 2018.

William L Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584*, 2017.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

Wengong Jin, Regina Barzilay, and Tommi Jaakkola. Junction tree variational autoencoder for molecular graph generation. *ICML*, 2018.

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *ICLR*, 2017.

Ruoyu Li, Sheng Wang, Feiyun Zhu, and Junzhou Huang. Adaptive graph convolutional neural networks. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

Maximilian Nickel, Lorenzo Rosasco, and Tomaso Poggio. Holographic embeddings of knowledge graphs. In *AAAI*, 2016.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In *European Semantic Web Conference*, pp. 593–607. Springer, 2018.

Martin Simonovsky and Nikos Komodakis. GraphVAE: Towards generation of small graphs using variational autoencoders. In *ICANN*, pp. 412–422. Springer, 2018.

Kai Sheng Tai, Richard Socher, and Christopher D. Manning. Improved semantic representations from tree-structured long short-term memory networks. *CoRR*, abs/1503.00075, 2015. URL `http://arxiv.org/abs/1503.00075`.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

Minjie Wang, Lingfan Yu, Quan Gan, Da Zheng, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Junbo Zhao, Haibin Lin, Chao Ma, Damon Deng, Qipeng Guo, Hao Zhang, Jinyang Li, Alexander J Smola, and Zheng Zhang. Deep graph library, 2018. URL `http://dgl.ai3`.

Bing Yu, Haoteng Yin, and Zhanxing Zhu. Spatio-temporal graph convolutional networks: A deep learning framework for traffic forecasting. *arXiv preprint arXiv:1709.04875*, 2017.

Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Ruslan R Salakhutdinov, and Alexander J Smola. Deep sets. In *NIPS*, pp. 3391–3401, 2017.

Marinka Zitnik, Monica Agrawal, and Jure Leskovec. Modeling polypharmacy side effects with graph convolutional networks. *Bioinformatics*, 34(13):i457–i466, 2018.