Frontiers in Computing and Intelligent Systems
ISSN: 2832-6024 | Vol. 6, No. 1, 2023

Based on TPUGRAPHS Predicting Model Runtimes Using Graph Neural Networks

Jingyu Xu [1, *], Linying Pan [2], Qiang Zeng [3], Wenjian Sun [4], Weixiang Wan [5]

[1] Computer Information Technology, Northern Arizona University, Flagstaff, Arizona, USA
[2] Information Studies, Trine University, Phoenix, Arizona, USA
[3] Computer Technology, Zhejiang University, Hangzhou, Zhejiang, China
[4] Electronic and Information Engineering, Yantai University, Tokyo, Japan
[5] Electronics & Communication Engineering, University of Electronic Science and Technology of China, Chengdu, China

[*] **Corresponding author**: Jingyu Xu (Email: jyxu01@outlook.com)

Abstract: Deep learning frameworks are mainly divided into pytorch in academia and tensorflow in industry, where pytorch is a dynamic graph and tensor flow is a static graph, both of which are essentially directed and loopless computational graphs. In TensorFlow, data input into the model requires a good computational graph structure to be executed, and static graphs have more optimization methods and higher performance. The node of the graph is OP and the edge is tensor. The static diagram is fixed after the compilation is completed, so it is easier to deploy on the server. How to compile a static graph. It is found that in the compilation process of static graphs, the configuration of the compiler (config) affects the way the compiler compiles and optimizes the model, and ultimately affects the running time of the model. We propose a reliable model, which can predict the best compilation configuration of the model according to the compilation configuration and runtime of the machine learning model in the training dataset to minimize the running time.

Keywords: TPU Graphs; Runtime Prediction; Computational Graphs; Compilation Configuration.

1. Introduction

Compilers frequently employ performance models for tackling optimization problems [1]. Building precise cost models to forecast execution times on contemporary processors is exceedingly challenging and time-consuming. This difficulty arises from the intricate nature of processors, particularly when detailed hardware design information is unavailable. The performance model is also utilized by the compiler autotuner to assess candidate configurations within the search space [2]. However, formulating an accurate analytic model for program performance on modern processors is both daunting and time-intensive. This challenge is attributed to the complexity of the underlying processor architecture, compilers, and their intricate interactions.

In this investigation, we utilized the TPUGRAPHS dataset publicly available from Phothi limhana et al. [3]. Various recent methodologies leverage machine learning (ML) to acquire performance prediction models. Nonetheless, datasets for program performance prediction are limited, mainly targeting small sub-programs. ML compilers address multiple optimization challenges in translating an ML program, typically presented as a tensor computation graph, into an efficient executable for a hardware target. Recent studies have demonstrated that search-based autotuning techniques can generate code with performance close to optimal. However, autotuning demands a relatively substantial resource investment compared to traditional heuristics-based compilers. Consequently, several methodologies incorporate a learned cost model to expedite autotuning.

This research strives to contribute to the field of AI model runtime prediction and the application of graph neural networks. By exploring the potential of graph neural network models in comparison to existing methods, we aim to minimize search time and enhance prediction accuracy. This endeavor may pave the way for more efficient model compilation configuration selection and optimization.

2. Related Work

Numerous academic works leverage machine learning techniques for code optimization. Our focus centers on papers employing machine learning to construct cost models for the prediction of program execution times. The ensuing studies illustrate pivotal advancements in this dynamic domain:

It hemal employs a hierarchical recurrent neural network to approximate the throughput of x86-64 basic blocks [4]. With an average error rate of 9%, Ithemal adeptly estimates throughput. While its emphasis lies on small loop-free programs, represented sequentially as instruction sequences executing on intricately designed processors.

The performance model based on code features [5] and Halide's cost model [6] utilize simplistic neural networks to forecast runtime. These networks operate on manually-engineered features generated by a static analyzer from an optimized program (or a program and a schedule in Halide). Given the non-trivial nature of extracting these features from an XLA graph, we train a more intricate neural network using features directly extractable from the XLA graph, possessing adequate capacity to recover similarly potent representations.

AutoTVM adopts a distinct form of a cost model to steer its search for a rapid configuration in machine learning programs [7]. Unlike estimating runtime directly, AutoTVM ranks candidates. Furthermore, AutoTVM models exhibit limited generalization across kernels and are trained for per-kernel search within a kernel-specific parameter set.

Graphs offer a natural representation for real-world data with relational structures, such as social networks, molecular networks, and webpage graphs. Recent endeavors extend deep neural networks (DNNs) to extract high-level features

from graph-structured datasets. These resulting architectures, known as graph neural networks (GNNs), have recently demonstrated state-of-the-art performance in various graph-related tasks, encompassing vertex classification, graph classification, and link prediction.

Noteworthy efforts have been directed towards developing machine learning-based models for both absolute and relative runtime estimation. Huang et al. [8] introduces sparse polynomial regression to predict program execution time using hand-crafted features of high-level programs. Dubach et al. [9] employs neural networks with hand-crafted features to estimate the speedup between two code sequences. Game Time utilizes SMT solvers to generate inputs and employs game-theoretic approaches to predict the runtime distribution of programs.

At any rate, these models necessitate manual feature engineering, and the predictions for runtime occur at a more general granularity, such as the entire program level. Consequently, we are in the process of constructing a learned cost model through a neural network, a significantly less labor-intensive approach compared to manual development. In contrast, our model autonomously acquires the capability to predict the throughput of basic blocks, incorporating minimal architectural knowledge into the model.

3. Methodology

In this section, we provide a comprehensive overview of the approach we have taken in our research focused on model runtime prediction. The main goal is to develop a reliable predictive model for model optimal configuration prediction using TPUGRAPHS data.

The cornerstone of our approach is the use of advanced machine learning techniques, with a special emphasis on the application of graph neural networks (GNNs). This choice is driven by the potential of graph neural networks to discern complex patterns, which in irregular graph data is a critical aspect of accurately predicting the optimal configuration for the model to run.

To the best of our knowledge, there is only one public study by Phothilimthana's team that has been done on the TPUGRAPHS dataset so far [3]. So, to ensure rigorous evaluation, the performance of the GNN model is juxtaposed with the established method for optimal configuration prediction, providing valuable insights into the effectiveness of our approach.

3.1. Model Architectures

The chosen Graph Neural Network (GNN) architecture is meticulously designed to address the intricacies of specific task. The overall architecture of the model is shown in Figure 1.

Node features consist of two parts as shown in Figure 1. The first part is an opcode id, i.e., type of tensor operation (such as matrix multiplication). Our models map an opcode id to an opcode embedding via an embedding lookup table. The opcode embedding is then concatenated with the rest of the node features as inputs to a GNN. We combine the node embeddings produced by the GNN to create the embedding of the graph using a simple pooling reduction. The resulting graph embedding is then linearly transformed into the final scalar output by a feedforward layer. Prior work [10] has studied alternative models, including LSTM and Transformer, and shown that GNNs offer the best performance.

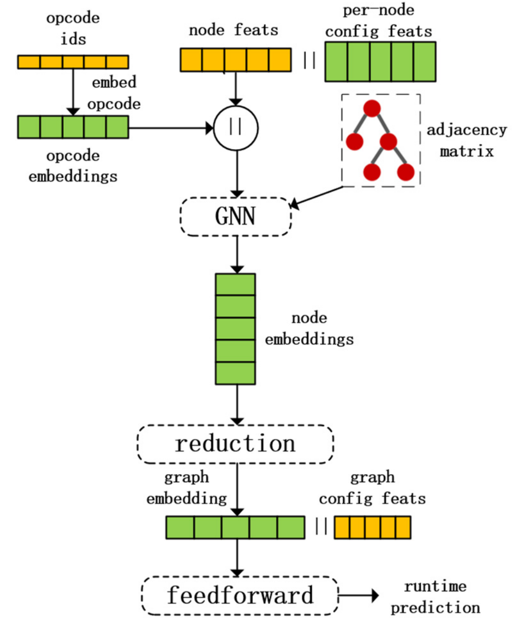


Figure 1. Model architecture.

It is worth mentioning that the graph neural network (GNN) of the graph structure in our model is implemented by means of message passing, and the multi-layer graph network is used to complete the point classification task [11]. In order to intuitively understand graph-structured convolutional neural networks, traditional image convolution superimposes pixels with different weights around a pixel, while graph-structured convolutional superimposes neighbors around a node, stacked together according to different weights.

Calculation formula for graph structure neural network:

$$H^{(l+1)} = \sigma\left(D^{-\frac{1}{2}} A D^{-\frac{1}{2}} H^{(l)} W^{(l)} \right) \tag{1}$$

where A is the adjacency matrix, D is related to the degree matrix, and H is the node representation of each layer.

Message Passing Mechanism: Message delivery consists of two steps, namely, the generation of features from the source node on the edge to the target node, and the aggregation of the received features by the target node [12]. When the neighbor node is adjacent to a large number of nodes, the importance of the neighbor node to the target node should be smaller, so the method of weighting the edge is adopted. The attention-based weighting and feature aggregation formulas are as follows:

The attention calculation method is as follows:

$$\alpha_{ij} = \frac{\exp(LeakyRelu(\vec{a}^{T}[W\vec{h}_i, W\vec{h}_j]))}{\sum_{k\in N_i} \exp(LeakyRelu(\vec{a}^{T}[W\vec{h}_i, W\vec{h}_k]))} \tag{2}$$

where "," represents the connection operation, which LeakyRelu is the activation function.

Feature aggregation calculation method:

$$\vec{h}_i' = \sigma(\sum_{j\in N_i} \alpha_{ij} W \vec{h}_j) \tag{3}$$

where σ is the activation function.

3.2. DataSet Introduction

The data in this dataset is obtained from classical open-

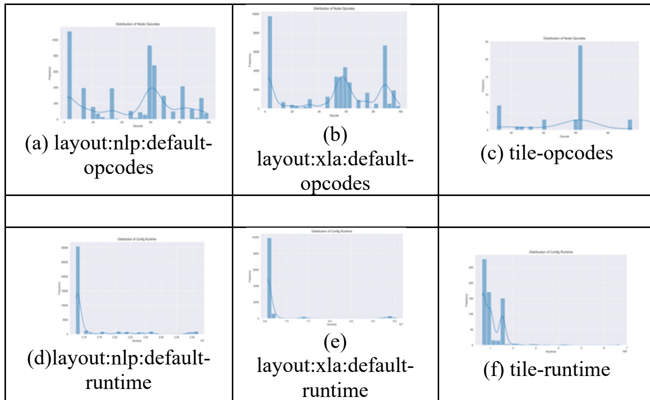
source models such as ResNet, Mask R-CNN and various types of transform models in the process of various tasks, and the dataset is further analyzed, and each data sample contains a computational graph, a compilation configuration, and the time to run on the TPU under the configuration. The TPUGRAPHS dataset can be divided into 5 sections, and the statistics are shown in table 1.

Table 1. Statistics of TPUGRAPHS collections

Collection	Core (Sub)Graphs	Avg.Nodes	Configs per Graph	Total Graphs +Configs	Samples
tiles	6988	40	1,842	12,870,077	12,870,077
Layout:xla:random			11,648	908,561	1,115,709
Layout:xla:default	78	14,105	10,147	771,496	1,272,5381
Layout:nlp:random			66,089	16,125,781	16,135,731
Layout:nlp:default	244	5,659	56,534	13,285,415	15,479,038

3.3. Data Analysis and Preprocessing

We conducted a thorough analysis of runtime data from open-source machine learning programs, focusing on popular model architectures such as ResNet, EfficientNet, Mask R-CNN, and Transformer. This field can be seen as the target of the regression task if you want to predict the runtime directly. Following shows the distribution of config_runtime of the demo example. As can be observed, there seems to exist multiple modals in different regions in default search strategy.



(a) layout:nlp:default-opcodes (b) layout:xla:default-opcodes (c) tile-opcodes

(d)layout:nlp:default-runtime (e) layout:xla:default-runtime (f) tile-runtime

Figure 2. DataSet Exploration

For node opcodes, operation codes (opcodes) span across a wide range from 0 to 100. Regarding config runtime, the runtime values are distributed between 2.15 × 10^7 and 2.31 × 10^7. Although there is no distinct peak, the data exhibits a slight right-skew. This Exploratory Data Analysis (EDA) has unveiled the fundamental characteristics and distributions of each feature and the target variable.

3.4. Loss Function

To guide the optimization process, we can train the model using regression losses (e.g., Mean Square Error (MSE)) or ranking losses (e.g., ListMLE [13]). Calculate the ranking loss between pairs of samples within the same plot in the same batch and reduce the loss of different plots in the batch to get the total loss.

The formula for calculating MSE is as follows:

$$MSE = \frac{1}{n}\sum\left(y - \bar{y}\right)^2 \quad (4)$$

where y represents the true labels, $\bar{y}$ represents the predicted probabilities, and n is the total number of samples.

MSE, as a regression loss, is not a good measure of the difference between the predicted optimal top K configuration of the model and the true optimal top K configuration when doing ordered prediction tasks. Therefore, the layout and tile models use the ListMLE loss function, and the ListMLE is calculated as follows:

$$ListMLE = \left(\{y\},\{s\}\right) = -\log\left(P\left(\pi_y \mid s\right)\right) \quad (5)$$

where $P\left(\pi_y \mid s\right)$ is the Plackett-Luce probability of permutation π_y conditional on fraction s, which randomly breaks the relationship of the number of items y sorted by correlation labels.

3.5. Evaluation Metrics

For both tile and layout data, we consider using a combination of two different evaluation metrics, as shown below.

3.5.1. Tile Evaluation

For tile configuration ranking evaluation, we use Ordered Pair Accuracy (OPA) [14] as a validation metric to select the best model checkpoint, and top-K error as an evaluation metric as they evaluate the quality of ranking. We define a top-K error to reflect how much slower the top-K configurations predicted by the model is from the actual fastest configuration as follows:

where K is the top-K predictions, A is all configurations of the given graph from the dataset collection, and y is the measured execution time.

$$1-\left(\frac{\text{The best runtime of the top-k predictions}}{\text{The best runtime of all configurations}}-1\right)=2-\frac{\min_{i\in K} y_i}{\min_{i\in A} y_i} \quad (6)$$

3.5.2. Layout Evaluation

For the layout configuration ranking evaluation, the Kendall rank correlation coefficient is used, that is, the degree to which the configuration ranking predicted by the model corresponds to the actual ranking at runtime.

$$\tau = \frac{(\text{number of concordant pairs}) - (\text{number of discordant pairs})}{(\text{number of pairs})} \quad (7)$$

4. Experiment Results

Observing the above results, it is found that the model has the best effect for tile training and prediction. For layout data,

the training and prediction effect of random in NLP is significantly greater than that of other data in layout, and the accuracy rate is improved by about 20%~30%. For the data in the third and fourth parts, the accuracy of the model on the validation set is higher than that of the training set.

Table 2. Model results

data	Accuracy	
	train	test
tiles	0.8785	0.8622
Layout:xla:random	0.6841	0.5285
Layout:xla:default	0.5631	0.5887
Layout:nlp:random	0.8197	0.8387
Layout:nlp:default	0.5036	0.4841

These results reinforce the potential of advanced deep learning techniques, especially GNN models, to improve the accuracy of model runtime predictions. GNN's ability to capture complex patterns in high-dimensional data has proven to help achieve such high prediction accuracy.

5. Conclusion

In the pursuit of determining the optimal compilation configuration for a model, this study advocates a model architecture founded on a graph neural network. The achieved test effect on the test set is noteworthy, reaching 0.417. The ability to predict the best compilation configuration enhances the efficiency of AI model execution, thereby mitigating overall time and resource consumption. Additionally, an attention mechanism is incorporated to assign weights to neighbors, thereby enhancing the model's generalization ability and overall performance. Employing models for optimizing compiler configurations serves the dual purpose of enhancing running speed and providing predictive insights into each model's behavior under optimal compiler configurations. This comprehensive approach integrates aspects of performance optimization, machine learning, and graph data processing. The utilization of Graph Neural Networks (GNNs) in performance optimization extends to various domains, including combinatorial optimization, such as the Google Brain team's application of GNNs to optimize power, area, and performance in new hardware chip blocks. The robust learning and representation capabilities of GNNs position them as valuable tools across diverse fields.

The introduction should provide background information (including relevant references) and should indicate the purpose of the manuscript. Cite relevant work by others, including research outside your company. Place your work in perspective by referring to other research papers. Inclusion of statements at the end of the introduction regarding the organization of the manuscript can be helpful to the reader.

In this template, the “Styles” menu should be used to format your text if needed. Highlight the text you want to designate with a certain style, and then select the appropriate name on the Style menu. The style will adjust your fonts and line spacing. Use italics for emphasis; do not underline as shown in Table 1.

References

[1] GCC.AutoVectorizationinGCC.https://www.gnu.org/software /gcc/projects/treessa/vectorization.html, August 2019. [Online; last modified 18-August-2019].

[2] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi,Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. Learning to Optimize Halide with Tree Search and Random Programs. ACM Trans. Graph.,38(4):121:1–121:12, July 2019.

[3] Phitchaya Mangpo Phothilimthana, Sami Abu-El-Haija, Kaidi Cao, Bahare Fatemi, C harith Mendis, Bryan Perozzi: “TpuGraphs: A Performance Prediction Dataset on Large Tensor Computational Graphs”, 2023; [http://arxiv.org/ abs/ 2308. 13490 arXiv:2308.13490].

[4] Charith Mendis, Alex Renda, Saman P. Amarasinghe, and Michael Carbin. Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks. In Proceedings of the 36th International Conference on Machine Learning, ICML, 2019.

[5] Christophe Dubach, John Cavazos, Björn Franke, Grigori Fursin, Michael F.P. O’Boyle, and Olivier Temam. Fast Compiler Optimisation Evaluation Using Code-feature Based Performance Prediction. In Proceedings of the 4th International Conference on Computing Frontiers, CF’07, 2007.

[6] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Efficient Multi-objective Neural Architecture Search via Lamarckian Evolution. arXiv e-prints, page arXiv:1804.09081, Apr2018.

[7] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to Optimize Tensor Programs. In Proceedings of the 32Nd International Conference on Neural Information Processing Systems, NIPS’18, 2018.

[8] Che, C., Hu, H., Zhao, X., Li, S., & Lin, Q. (2023). Advancing Cancer Document Classification with R andom Forest. Academic Journal of Science and Technology, 8(1), 278–280.

[9] Lin, Q., Che, C., Hu, H., Zhao, X., & Li, S. (2023). A Comprehensive Study on Early Alzheimer’s Disease Detection through Advanced Machine Learning Techniques on MRI Data. Academic Journal of Science and Technology, 8(1), 281–285.

[10] Samuel J. Kaufman, Phitchaya Mangpo Phothilimthana, Yanqi Zhou, Charith Mendis, Sudip Roy, Amit Sabne, and Mike Burrows. A learned performance model for tensor processing units. In Proceedings of Machine Learning and Systems, 2021.

[11] Hao Hu, Shulin Li, Jiaxin Huang, Bo Liu, and Change Che. Casting product image data for quality inspection with xception and data augmentation. Journal of Theory and Practice of Engineering Science, 3(10):42–46, 2023.

[12] Hamilton W, Ying Z, Leskovec J. Inductive representation learning on large graphs[C]//Advances in Neural Information Processing Systems. 2017: 1024-1034

[13] Tianbo, Song, Hu Weijun, Cai Jiangfeng, Liu Weijia, Yuan Quan, and He Kun. "Bio-inspired Swarm Intelligence: a Flocking Project With Group Object Recognition." In 2023 3rd International Conference on Consumer Electronics and Computer Engineering (ICCECE), pp. 834-837. IEEE, 2023.

[14] Che C, Liu B, Li S, et al. Deep Learning for Precise Robot Position Prediction in Logistics[J]. Journal of Theory and Practice of Engineering Science, 2023, 3(10): 36-41.