TpuGraphs: A Performance Prediction Dataset on Large Tensor Computational Graphs

Phitchaya Mangpo Phothilimthana
Google DeepMind
USA
mangpo@google.com

Sami Abu-El-Haija
Google Research
USA
haija@google.com

Kaidi Cao
Stanford
USA
kaidicao@cs.stanford.edu

Bahare Fatemi
Google Research
USA
baharef@google.com

Charith Mendis
UIUC
USA
charithm@illinois.edu

Bryan Perozzi
Google Research
USA
bperozzi@google.com

ABSTRACT

Precise hardware performance models play a crucial role in code optimizations. They can assist compilers in making heuristic decisions or aid autotuners in identifying the optimal configuration for a given program. For example, the autotuner for XLA, a machine learning compiler, discovered 10–20% speedup on state-of-the-art models serving substantial production traffic at Google. Although there exist a few datasets for program performance prediction, they target small sub-programs such as basic blocks or kernels. This paper introduces TpuGraphs, a performance prediction dataset on full tensor programs, represented as computational graphs, running on Tensor Processing Units (TPUs). Each graph in the dataset represents the main computation of a machine learning workload, e.g., a training epoch or an inference step. Each data sample contains a computational graph, a compilation configuration, and the execution time of the graph when compiled with the configuration. The graphs in the dataset are collected from open-source machine learning programs, featuring popular model architectures (e.g., ResNet, EfficientNet, Mask R-CNN, and Transformer). TpuGraphs provides 25x more graphs than the largest graph property prediction dataset (with comparable graph sizes), and 770x larger graphs on average compared to existing performance prediction datasets on machine learning programs. This graph-level prediction task on large graphs introduces new challenges in learning, ranging from scalability, training efficiency, to model quality.

KEYWORDS

datasets, graph neural networks, compilers, execution time

1 INTRODUCTION

Compilers often use performance models to solve optimization problems [21, 36], as collecting performance measurements from real hardware can be expensive, limited, or infeasible. A performance model can also be used by a compiler autotuner to evaluate candidate configurations in a search space [1, 11, 29, 41, 42]. However, developing an accurate analytical model of program performance on a modern processor is challenging and time-consuming because the underlying processor architecture, the compiler, and their interactions are complex and difficult to model analytically.

Many recent methods [1, 2, 4, 11, 18, 29, 32, 35, 39, 49, 50, 58] apply machine learning (ML) to learn performance prediction models. However, there exist only a few datasets for program performance prediction, and they all target small sub-programs. BHive [12] targets small basic blocks of assembly instructions. TenSet [60] targets ML kernels consisting of a small number of tensor operations. The database query dataset [24] contains larger query programs, but they are still relatively small, most with fewer than 100 nodes.

Unlike prior datasets, TpuGraphs is a performance prediction dataset on full tensor programs, represented as computational graphs. Each graph represents the main computation of an ML program, which is usually one or many training steps or one inference step. The graphs in the dataset are collected from open-source ML programs, featuring popular models (e.g., ResNet, EfficientNet, Mask R-CNN, and a large variety of Transformer) for a wide range of tasks (e.g., vision, NLP, speech, audio, recommendation, and generative AI). Each data sample contains a computational graph, a compilation configuration, and the execution time when executing the graph with the given configuration on a Tensor Processing Unit (TPU) v3 [30], an accelerator for ML workloads. A compilation configuration controls how the XLA compiler [52] transforms the graph for a specific optimization pass. In particular, the TpuGraphs dataset consists of two collections: (i) *layout* and (ii) *tile*. *Layout* configurations control how tensors are laid out in the physical memory, by specifying the dimension order of each input and output of an operation node. A *tile* configuration controls the tile size of each fused subgraph. We primarily focus on layout and tile configurations because tuning them offers the highest performance gain on average, compared to tuning other compiler optimizations.

The layout collection contains 31 million pairs of graphs and configurations, averaging over 7,700 nodes per graph. The tile collection contains 13 millions pairs of kernels and configurations, averaging 40 nodes per kernel subgraph. The layout collection is unique among existing graph datasets, in that it provides data for graph-level predictions on very large graphs. In contrast, most of

GLB '23, August 6, 2023, Long Beach, CA

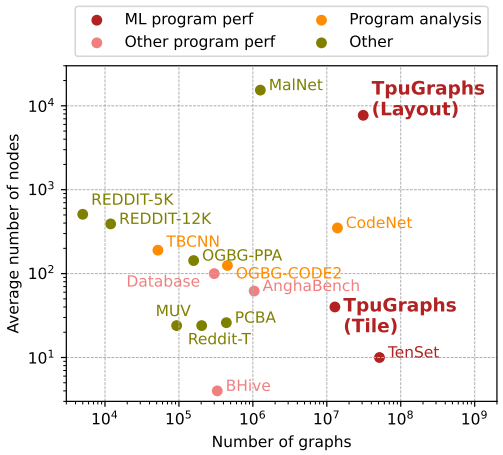

Figure 1: Scale of TpuGraphs compared to other graph property prediction datasets.

the existing graph datasets fall into two categories: graph-level prediction on small graphs [3, 9, 11, 26, 26, 51, 53, 60, 61], and node-level or edge-level prediction on large graphs [7, 10, 22, 27, 37, 57, 62]. TpuGraphs provides 25x more graphs than MalNet [20] — the largest graph property prediction dataset with comparable graph sizes — and 770x larger graphs on average compared to TenSet [59] — the only existing large-sclae ML program performance dataset — as depicted in Figure 1. The scale of TpuGraphs poses several new research challenges:

- How to train a neural network model that can perform graph-level predictions when the memory required to train the model on a single graph may not fit on a single device?
- How to make a model generalize well to unseen graphs when they are diverse, and the training data may be imbalanced?
- How to improve the efficiency of a training pipeline when multiple data points contain a large amount of redundant data (same core graph but different graph configurations)?

We provide baseline models based on a Graph Neural Network (GNN), following the techniques from the most recent works on TPU learned cost models [8, 32]. We verify that the previously proposed techniques — such as training on graph segments and using pairwise ranking loss — are also effective on our dataset. The dataset and accompanying code can be found at https://github.com/google-research-datasets/tpu_graphs.

2 THE TPUGRAPHS DATASET

The TpuGraphs dataset contains execution time data points, where each data point contains an HLO graph, its configuration, and its execution time on a single core of TPU v3. The HLO graph in each data point is a partially optimized graph before being fed into the corresponding optimization pass. For example, in the *layout* collection, an HLO graph is the input graph to the layout assignment pass. The layout configuration of a graph is a collection of per-node layout decisions on configurable nodes (i.e., convolution and reshape). For the *tile* collection, an HLO graph in each data point is a fused subgraph representing a kernel. The tile configuration of a subgraph is a configuration for the entire subgraph, not specific to any particular node.

2.1 Data Generation

Within our dataset, there are multiple collections of data, differing in terms of (1) the compiler optimization (i.e., layout and tile), (2) the source of graphs, and (3) the search strategy.

Graphs Collection. We collect HLO graphs from two sources. The first source, called *XLA*, is the combination of the XLA regression benchmark — from where we collect all open-source models — and the MLPerf benchmark [28, 38]. The *XLA* graphs span diverse types of popular ML training and inference models, such as vision, NLP, speech, audio, and recommendation. The second source, called *NLP*, contains a variety of BERT for training and inference, with varying number of layers, attention heads, and hidden sizes. For each model, we run the program — written in TensorFlow, PyTorch, or JAX — and collect the largest HLO graph compiled by XLA, which represents the model's main computation. Note that the TpuGraphs dataset is similar to the internal datasets used for prior TPU learned cost models [8, 32], but it exclusively contains graphs from open source-programs, while the internal datasets also include production models that we cannot release publicly.

Configurations Generation. Once we have the graphs, we use the XLA autotuner to generate data samples. The set of configurations being generated depends on how the autotuner explores the search space. For the layout collections, we ran the autotuner in two modes. The first mode explores the search space using a genetic algorithm starting from the default configuration, chosen by the compiler's heuristic. Data collected from this mode is labeled *default*. The second mode explores the search space by picking random candidates. Data collected from this mode is labeled *random*. We keep data collected in different modes in separate collections; the default collection tends to contain configurations that are not too different from the default, and have similar execution times, while the random collection includes very different configurations with very different execution times.

For the tile size tuning, the autotuner first invokes the compiler to run the graph-level optimizations and obtain fused subgraphs (kernels). For each subgraph, the autotuner enumerates all possible tile sizes for the kernel in a random order, limited by a timeout. Note that the tile size search space is much smaller than the layout search space, so we can enumerate all possible tile sizes. Therefore, there is one data collection for tile sizes.

Appendix C describes how we measure the execution time of a given graph and configuration.

2.2 Dataset Statistics

Table 1 summarizes the details of the different data collections, where the collection name follows the pattern *optimization:source:search*.

2.3 Dataset Split

We split the data using 80-10-10 ratio by graphs in each collection. Splitting data by graphs ensures that graphs in the validation and test sets do not appear in the training set to evaluate the generalization of the model on unseen graphs. The validate and test graphs

Table 1: Statistics of TpuGraphs collections. The collection name follows the pattern *optimization:source:search*. The search may explore the same configuration multiple times, so the same pair of graph and configuration may appear multiple times with slightly different execution time from multiple measurements. The total number of samples is thus higher than the number of unique pairs.

Collection	Core (Sub) Graphs	Avg. Nodes	Configs per Graph	Total Graphs + Configs	Samples
Layout:XLA:Default	78	14,105 (372–43,615)	10,147 (681–71,574)	**771,496**	1,272,538
Layout:XLA:Random			11,648 (109–99,783)	**908,561**	1,115,709
Layout:NLP:Default	244	5,659 (876–21,919)	56,534 (9032–90,985)	**13,285,415**	15,479,038
Layout:NLP:Random			66,089 (8,843–100,001)	**16,125,781**	16,135,731
Tile:XLA	6,988	40	1,842	**12,870,077**	12,870,077

stay the same across different XLA collections; the same applies to NLP collections. We deliberately holdout the target labels of samples in the test set for competition purposes.

3 LEARNED PERFORMANCE PREDICTION MODEL

The goal of a learned cost model is to rank the performance of different configurations of a given graph. This section explains the baseline models we provide and how we train them, primarily based on the TPU learned cost model papers [8, 32].

3.1 Feature Extraction

TpuGraphs provides data in two formats: raw protobuf format and numpy arrays similar to OGBG format [25]. The autotuner produces output results in protobuf format. A data pre-processing script converts data from the protobuf format to the numpy format. The main function of the data pre-processor is feature extraction. Node features describe the node's property, such as output tensor shape, tensor layout, striding, padding, and operation-specific parameters. Our feature extraction is minimal. To extract a node feature vector, we either copy values from various fields in an HLO instruction (a node in an HLO graph) as they are, or convert categorical values using one-hot encoding. To convert an unbounded list of numbers (e.g. tensor shape) to a fixed-size vector, we truncate the list to six elements and include the summation and/or product of all elements in the list (e.g., the product of dimension sizes represents the volume of the tensor). A per-node layout configuration and tile size can be represented as a nested list with some unbounded dimensions. Similarly, we truncate these unbounded dimensions to six elements.

We provide code for training a variety of models over the numpy format. Nonetheless, the raw format can allow researchers to experiment with different feature extractions and measure impacts on the quality of a learned model.

3.2 Model Architecture

Figure 2 shows the model architecture we use for our baseline models. Our baseline models are based on a GNN since the input program is represented as a graph. Node features consist of two parts as shown in Figure 2. The first part is an opcode id, i.e., type of tensor operation (such as matrix multiplication). Our baseline 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

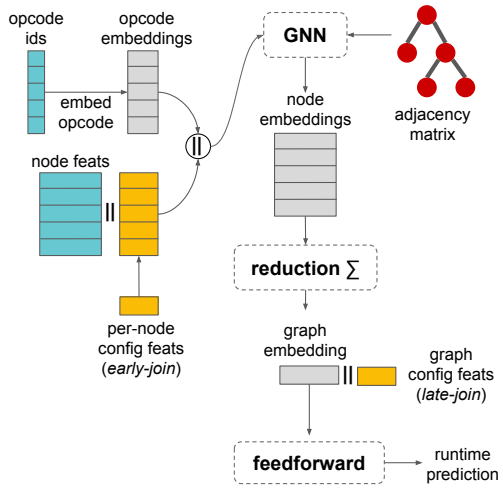

Figure 2: Model architecture.

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 [32] has studied alternative models, including LSTM and Transform, and shown that GNNs offer the best performance. We provide baseline models with GCN [33] and GraphSAGE [23].

3.3 Loss Functions and Evaluation Metrics

The primary use case of the model is to rank configurations within a given graph and select top candidates to evaluate on real hardware. Thus, we can train the model using regression losses (e.g., Mean Square Error (MSE)) or ranking losses (e.g., ListMLE [54]). A ranking loss is computed among sample pairs within the same graph in the same batch, and the losses from different graphs in the batch are reduced to get the total loss. We use Ordered Pair Accuracy (OPA) 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:

$$\frac{\text{Best runtime of the top-k predictions}}{\text{Best runtime of all configurations}} - 1 = \frac{\min_{i \in K} y_i}{\min_{i \in A} y_i} - 1 \quad (1)$$

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

3.4 Implementation

Layout model. Our default baseline model is a 3-layer GraphSAGE. We concatenate node features and per-node configuration features as inputs to the GNN. If a node is non-configurable (having no layout configuration), we use a zero vector as configuration features. To address the memory issue when training on the full program graphs in the layout dataset, we apply the Graph Segment Training (GST) method [8]. GST divides large graphs into smaller segments. During training, a random segment is chosen for model updates at each step. This approach allows us to store intermediate activations for only one segment during backpropagation. The embeddings of all segments are merged to generate an embedding for the original large graph, which is used for prediction. As a result, the memory usage during training is bounded for each large graph, irrespective of its size. By default, we use the maximum segment size of 1,000 nodes and the keep probability $p = 0.5$ for the stale embedding dropout. We consider MSE and pairwise hinge loss as a loss function. Model training takes approximately 2–3 days on a single NVIDIA A100 GPU with 80GB HBM2e Memory. The baseline models are implemented using the GraphGPS framework [44] on top of PyTorch 1.10.

Tile size model. For the tile collection, we implement three baselines using TensorFlow-2 and TF-GNN: an MLP model and two GNNs (GraphSAGE and GCN with residual connections). The MLP model embeds all opcodes, concatenates with node features, sums across all nodes, then concatenates with kernel configuration features, feeding into 3-layer MLP. The GNN variants follow Figure 2. We experiment with two options to combine the graph-level features with the node-level information (orange in the figure): either *late-join* or *early-join*. The first runs the GNN only on node features, reduces the node embeddings, and then concatenates with the graph (configuration) features. As such, multiple configurations over the same graph share the forward and backward pass. The second replicates the graph features onto every node. Here, we group configurations per graph and therefore execute sparse-ops only once per graph (on cube-tensors rather than matrices). We consider MSE and ListMLE as a loss function. These models take a few minutes (MLP), to less than an hour (late-join GNN), to a couple of hours (early-join GNN), to train on Intel Xeon CPUs.

4 BASELINE EXPERIMENTS

Table 2 reports the average top-K error of the best model on across programs all the dataset collections. The Layout:XLA:Random and Layout:NLP:Random are by far the most difficult collections. If we use the learned cost model to select the top configuration, we will be on average 24–25% slower than the known optimal configuration. Even if we consider top 10 candidates, we will still be on average 8–10% off. This is likely because the configurations from the random search are very different from each other. Even when the NLP collection contains only graphs with the transformer architecture, the best learned model still has relatively low accuracy on the random search collections.

Table 2: Prediction errors (Eq. 1) of our best baseline model on different dataset collections. The values of (K_1, K_2, K_3) are (1, 10, 100) for the layout collections, and (1, 5, 10) for the tile collection.

Collection	Top-K_1 Error %		Top-K_2 Error %		Top-K_3 Error %	
	Val	Test	Val	Test	Val	Test
Layout:XLA:Random	24.3	25.3	6.4	10.4	0.4	1.2
Layout:XLA:Default	1.7	1.6	0.5	0.5	0.1	0.1
Layout:NLP:Random	23.7	24.5	7.4	8.0	1.8	2.7
Layout:NLP:Default	2.3	3.0	0.2	0.2	0.1	0.1
Tile:XLA	10.5	10.8	3.9	3.4	2.7	2.1

Table 3: Prediction errors (%) of different model variants and training methods on the Layout:XLA:Random collection.

Model	Top-1 E		Top-10 E		Top-100 E	
	Val	Test	Val	Test	Val	Test
Best	24.3	25.3	6.4	10.4	0.4	1.2
Full Graph	34.3	39.6	11.5	14.9	0.7	2.6
Small Segment	37.9	47.3	13.3	17.9	1.4	3.5
Topo Partition	27.5	27.1	6.5	10.1	0.6	1.5
Fewer Layers	26.9	28.2	7.9	12.5	0.7	1.7
MSE loss	42.7	53.1	12.6	18.8	1.6	3.8
Random	58.1	90.5	15.7	20.6	1.8	3.6

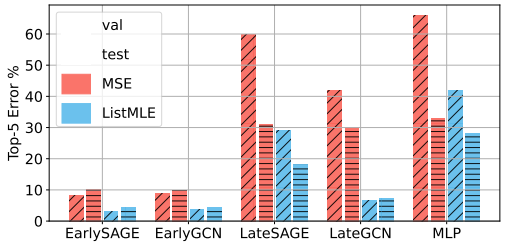

Figure 3: Prediction errors (%) of different model variants on the Tile:XLA collection. *Early* and *Late* refer to *early-join* and *late-join* options.

In contrast, Layout:XLA:Default and Layout:NLP:Default are much easier than the random collections. This is expected because configurations generated from a genetic algorithm starting from the default configuration should have relatively similar performance to the default configuration's. However, the learned model is not always accurate on all graphs. If we look at the model's accuracy per each graph (program), the top-1 error ranges between 0–9%.

On the Tile:XLA collection, the average top-1 error of the best model is very similar to the original TPU learned cost model paper's [32]. Note that our dataset is not exactly the same as the internal dataset used in the original paper, but they share a large number of overlapping graphs.

REFERENCES

[1] 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. 2019. Learning to Optimize Halide with Tree Search and Random Programs. *ACM Trans. Graph.* 38, 4, Article 121 (July 2019), 12 pages. https://doi.org/10.1145/3306346.3322967

[2] Byung Hoon Ahn, Prannoy Pilligundla, Amir Yazdanbakhsh, and Hadi Esmaeilzadeh. 2020. Chameleon: Adaptive Code Optimization for Expedited Deep Neural Network Compilation. In *International Conference on Learning Representations*.

[3] Miltiadis Allamanis. 2019. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 143–153.

[4] Riyadh Baghdadi, Massinissa Merouani, Mohamed-Hicham Leghettas, Kamel Abdous, Taha Arbaoui, Karima Benatchba, and Saman Amarasinghe. 2021. A Deep Learning Based Cost Model for Automatic Code Optimization. In *Proceedings of MLSys*.

[5] Scott Beamer, Krste Asanović, and David Patterson. 2017. The GAP Benchmark Suite. arXiv:1508.03619 [cs.DC]

[6] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural Code Comprehension: A Learnable Representation of Code Semantics. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 3589–3601.

[7] Aleksandar Bojchevski, Johannes Klicpera, Bryan Perozzi, Amol Kapoor, Martin Blais, Benedek Rózemberczki, Michal Lukasik, and Stephan Günnemann. 2020. Scaling graph neural networks with approximate pagerank. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2464–2473.

[8] Kaidi Cao, Phitchaya Mangpo Phothilimthana, Sami Abu-El-Haija, Dustin Zelle, Yanqi Zhou, Charith Mendis, Jure Leskovec, and Bryan Perozzi. 2023. Learning Large Graph Property Prediction via Graph Segment Training. arXiv:2305.12322 [cs.LG]

[9] Ines Chami, Sami Abu-El-Haija, Bryan Perozzi, Christopher Ré, and Kevin Murphy. 2022. Machine learning on graphs: A model and comprehensive taxonomy. *Journal of Machine Learning Research* 23, 89 (2022), 1–64.

[10] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In *International Conference on Learning Representations*.

[11] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to Optimize Tensor Programs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NeurIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 3393–3404.

[12] Y. Chen, A. Brahmakshatriya, C. Mendis, A. Renda, E. Atkinson, O. Sykora, S. Amarasinghe, and M. Carbin. 2019. BHive: A Benchmark Suite and Measurement Framework for Validating x86-64 Basic Block Performance Models. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE Computer Society, Los Alamitos, CA, USA, 167–177. https://doi.org/10.1109/IISWC47752.2019.9042166

[13] Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, and Hugh Leather. 2020. ProGraML: Graph-based Deep Learning for Program Optimization and Analysis. arXiv:2003.10536 [cs.LG]

[14] Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, Yuandong Tian, and Hugh Leather. 2022. CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research. In *CGO*.

[15] Anderson Faustino da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimarães, and Fernando Magno Quinão Pereira. 2021. ANGHABENCH: A Suite with One Million Compilable C Benchmarks for Code-Size Reduction. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 378–390. https://doi.org/10.1109/CGO51591.2021.9370322

[16] Laxman Dhulipala, Jessica Shi, Tom Tseng, Guy E. Blelloch, and Julian Shun. 2020. The Graph Based Benchmark Suite (GBBS). In *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences amp; Systems (GRADES) and Network Data Analytics (NDA)* (Portland, OR, USA) *(GRADES-NDA'20)*. Association for Computing Machinery, New York, NY, USA, Article 11, 8 pages. https://doi.org/10.1145/3398682.3399168

[17] Paul D. Dobson and Andrew J. Doig. 2003. Distinguishing Enzyme Structures from Non-enzymes Without Alignments. *Journal of Molecular Biology* 330, 4 (2003), 771–783. https://doi.org/10.1016/S0022-2836(03)00628-4

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

[19] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[20] Scott Freitas, Yuxiao Dong, Joshua Neil, and Duen Horng Chau. 2021. A Large-Scale Database for Graph Representation Learning. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

[21] GCC. 2019. Auto-Vectorization in GCC. https://www.gnu.org/software/gcc/projects/tree-ssa/vectorization.html. [Online; last modified 18-August-2019].

[22] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[23] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems*.

[24] Benjamin Hilprecht and Carsten Binnig. 2022. Zero-Shot Cost Models for out-of-the-Box Learned Cost Prediction. 15, 11 (jul 2022), 2361–2374. https://doi.org/10.14778/3551793.3551799

[25] Weihua Hu, Matthias Fey, Hongyu Ren, Maho Nakata, Yuxiao Dong, and Jure Leskovec. 2021. OGB-LSC: A Large-Scale Challenge for Machine Learning on Graphs. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.

[26] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems* 33 (2020), 22118–22133.

[27] Wenbing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. 2018. Adaptive sampling towards fast graph representation learning. *Advances in neural information processing systems* 31 (2018).

[28] Vijay Janapa Reddi, David Kanter, Peter Mattson, Jared Duke, Thai Nguyen, Ramesh Chukka, Ken Shiring, Koan-Sin Tan, Mark Charlebois, William Chou, Mostafa El-Khamy, Jungwook Hong, Tom St John, Cindy Trinh, Michael Buch, Mark Mazumder, Relja Markovic, Thomas Atta, Fatih Cakir, Masoud Charkhabi, Xiaodong Chen, Cheng-Ming Chiang, Dave Dexter, Terry Heo, Guenther Schmuelling, Maryam Shabani, and Dylan Zika. 2022. MLPerf Mobile Inference Benchmark: An Industry-Standard Open-Source Machine Learning Benchmark for On-Device AI. In *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu (Eds.), Vol. 4. 352–369. https://proceedings.mlsys.org/paper_files/paper/2022/file/7eabe3a1649ffa2b3ff8c02ebfd5659f-Paper.pdf

[29] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with Roc. In *Proceedings of MLSys Conference*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), Vol. 2. 187–198.

[30] Norman P. Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. 2020. A Domain-Specific Supercomputer for Training Deep Neural Networks. *Commun. ACM* 63, 7 (June 2020), 67–78. https://doi.org/10.1145/3360307

[31] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and Evaluating Contextual Embedding of Source Code. In *Proceedings of the 37th International Conference on Machine Learning (ICML'20)*. JMLR.org, Article 474, 12 pages.

[32] Samuel J. Kaufman, Phitchaya Mangpo Phothilimthana, Yanqi Zhou, Charith Mendis, Sudip Roy, Amit Sabne, and Mike Burrows. 2021. A Learned Performance Model for Tensor Processing Units. In *Proceedings of Machine Learning and Systems*. arXiv:2008.01040 [cs.PF]

[33] Thomas N Kipf and Max Welling. 2016. Semi-Supervised Classification with Graph Convolutional Networks. (2016).

[34] Scott P. Kolodziej, Mohsen Aznaveh, Matthew Bullock, Jarrett David, Timothy A. Davis, Matthew Henderson, Yifan Hu, and Read Sandstrom. 2019. The SuiteSparse Matrix Collection Website Interface. *Journal of Open Source Software* 4, 35 (2019), 1244. https://doi.org/10.21105/joss.01244

[35] Menghao Li, Minjia Zhang, Chi Wang, and Mingqin Li. 2020. AdaTune: Adaptive Tensor Program Compilation Made Efficient. In *34th Conference on Neural Information Processing Systems (NeurIPS'20)*.

[36] LLVM. [n. d.]. Auto-Vectorization in LLVM. https://bcain-llvm.readthedocs.io/projects/llvm/en/latest/Vectorizers. [Online; accessed 03-Feb-2020].

[37] Elan Sopher Markowitz, Keshav Balasubramanian, Mehrnoosh Mirtaheri, Sami Abu-El-Haija, Bryan Perozzi, Greg Ver Steeg, and Aram Galstyan. 2021. Graph Traversal with Tensor Functionals: A Meta-Algorithm for Scalable Learning. In *International Conference on Learning Representations*.

[38] Peter Mattson, Christine Cheng, Gregory Diamos, Cody Coleman, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debo Dutta, Udit Gupta, Kim Hazelwood, Andy Hock, Xinyuan Huang, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao,

Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St John, Carole-Jean Wu, Lingjie Xu, Cliff Young, and Matei Zaharia. 2020. MLPerf Training Benchmark. In *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), Vol. 2. 336–349. https://proceedings.mlsys.org/paper_files/paper/2020/file/02522a2b2726fb0a03bb19f2d8d9524d-Paper.pdf

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

[40] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing *(AAAI'16)*. AAAI Press, 1287–1293.

[41] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3341301.3359646

[42] Phitchaya Mangpo Phothilimthana, Amit Sabne, Nikhil Sarda, Karthik Srinivasa Murthy, Yanqi Zhou, Christof Angermueller, Mike Burrows, Sudip Roy, Ketan Mandke, Rezsa Farahani, Yu Emma Wang, Berkin Ilbeyi, Blake Hechtman, Bjarke Roune, Shen Wang, Yuanzhong Xu, and Samuel J. Kaufman. 2021. A Flexible Approach to Autotuning Multi-Pass Machine Learning Compilers. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 1–16. https://doi.org/10.1109/PACT52795.2021.00008

[43] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. arXiv:2105.12655 [cs.SE]

[44] Ladislav Rampášek, Mikhail Galkin, Vijay Prakash Dwivedi, Anh Tuan Luu, Guy Wolf, and Dominique Beaini. 2022. Recipe for a General, Powerful, Scalable Graph Transformer. *arXiv preprint arXiv:2205.12454* (2022).

[45] Bharath Ramsundar, Steven Kearnes, Patrick Riley, Dale Webster, David Konerding, and Vijay Pande. 2015. Massively Multitask Networks for Drug Discovery. *arXiv:1502.02072* (02 2015).

[46] Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic Model for Code with Decision Trees. *SIGPLAN Not.* 51, 10 (oct 2016), 731–747. https://doi.org/10.1145/3022671.2984041

[47] Sebastian G. Rohrer and Knut Baumann. 2009. Maximum Unbiased Validation (MUV) Data Sets for Virtual Screening Based on PubChem Bioactivity Data. *Journal of Chemical Information and Modeling* 49, 2 (2009), 169–184. https://doi.org/10.1021/ci8002649 PMID: 19161251.

[48] Benedek Rozemberczki, Oliver Kiss, and Rik Sarkar. 2020. Karate Club: An API Oriented Open-Source Python Framework for Unsupervised Learning on Graphs *(CIKM '20)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3340531.3412757

[49] Benoit Steiner, Chris Cummins, Horace He, and Hugh Leather. 2021. Value Learning for Throughput Optimization of Deep Learning Workloads. In *Proceedings of MLSys Conference*.

[50] O. Sykora, P. Phothilimthana, C. Mendis, and A. Yazdanbakhsh. 2022. GRANITE: A Graph Neural Network Model for Basic Block Throughput Estimation. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE Computer Society, Los Alamitos, CA, USA, 14–26. https://doi.org/10.1109/IISWC55918.2022.00012

[51] Damian Szklarczyk, Annika L Gable, David Lyon, Alexander Junge, Stefan Wyder, Jaime Huerta-Cepas, Milan Simonovic, Nadezhda T Doncheva, John H Morris, Peer Bork, et al. 2019. STRING v11: protein–protein association networks with increased coverage, supporting functional discovery in genome-wide experimental datasets. *Nucleic acids research* 47, D1 (2019), D607–D613.

[52] TensorFlow. [n. d.]. XLA: Optimizing Compiler for TensorFlow. https://www.tensorflow.org/xla. [Online; accessed 19-September-2019].

[53] Zhenqin Wu, Bharath Ramsundar, Evan N Feinberg, Joseph Gomes, Caleb Geniesse, Aneesh S Pappu, Karl Leswing, and Vijay Pande. 2018. MoleculeNet: a benchmark for molecular machine learning. *Chemical science* 9, 2 (2018), 513–530.

[54] Fen Xia, Tie-Yan Liu, Jue Wang, Wensheng Zhang, and Hang Li. 2008. Listwise Approach to Learning to Rank: Theory and Algorithm. In *Proceedings of the 25th International Conference on Machine Learning* (Helsinki, Finland) *(ICML '08)*. Association for Computing Machinery, New York, NY, USA, 1192–1199. https://doi.org/10.1145/1390156.1390306

[55] Pinar Yanardag and S.V.N. Vishwanathan. 2015. Deep Graph Kernels *(KDD '15)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2783258.2783417

[56] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

[57] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2019. GraphSAINT: Graph Sampling Based Inductive Learning Method. In *International Conference on Learning Representations*.

[58] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*. USENIX Association, 863–879.

[59] Lianmin Zheng, Ruochen Liu, Junru Shao, Tianqi Chen, Joseph Gonzalez, Ion Stoica, and Ameer Haj-Ali. 2021. TenSet: A Large-scale Program Performance Dataset for Learned Tensor Compilers. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, J. Vanschoren and S. Yeung (Eds.), Vol. 1. Curran. https://datasets-benchmarks-proceedings.neurips.cc/paper_files/paper/2021/file/a684eceee76fc522773286a895bc8436-Paper-round1.pdf

[60] Lianmin Zheng, Ruochen Liu, Junru Shao, Tianqi Chen, Joseph E. Gonzalez, Ion Stoica, and Ameer Haj Ali. 2021. TenSet: A Large-scale Program Performance Dataset for Learned Tensor Compilers. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*. https://openreview.net/forum?id=aIfp8kLuvc9

[61] Marinka Zitnik, Rok Sosič, Marcus W Feldman, and Jure Leskovec. 2019. Evolution of resilience in protein interactomes across the tree of life. *Proceedings of the National Academy of Sciences* 116, 10 (2019), 4426–4433.

[62] Difan Zou, Ziniu Hu, Yewen Wang, Song Jiang, Yizhou Sun, and Quanquan Gu. 2019. Layer-dependent importance sampling for training deep and large graph convolutional networks. *Advances in neural information processing systems* 32 (2019).

A RELATED DATASETS

Table 4 compares related datasets.

ML Program Performance. The TpuGraphs layout collections provide more than 770x larger graphs on average compared to TenSet [59], the only existing large-scale dataset on ML program performance. Our tile size collection is similar to TenSet as the configuration controls the optimization at the kernel (fused subgraph) level. However, it compliments TenSet nicely as it provides data points on different hardware. Halide Auto-scheduler [1] releases their evaluation dataset of Halide programs mainly consisting of image processing benchmarks with a few ML benchmarks.

Other Program Performance. Beyond ML programs, the performance prediction dataset with largest graphs is on database queries [24], whose graphs are still more than a few orders of magnitudes smaller than ours. Another popular performance prediction dataset is BHive [12], consisting of x86 basic blocks sourced from multiple open source programs, with runtime measurements on different Intel hardware platforms. However, the basic blocks are quite small, including four instructions on average. CompilerGym [14] releases a collection of LLVM IR code datasets that can be evaluated in their environment. The largest datasets in their collection includes AnghaBench [15] and CSmith [56]. AnghaBench provides a large number of relatively small real-world programs. CSmith programs are large (comparable to ours), but they are randomly generated programs. Additionally, CompilerGym's datasets do not come with performance measurements, so one would have to execute the programs and configurations in the CompilerGym's environment themselves to obtain program execution time.

Program Analysis. Other closely related datasets are on programming tasks. CodeNet [43] is a large dataset to teach AI to code, in which each code sample is a solution to one of the coding problems. OBGB-CODE2 [25] is for code summarization, containing Abstract Syntax Trees obtained from Python functions. TBCNN [40] releases its dataset on program classification from a pedagogical programming open judge system. CuBERT [31] uses Python files extracted from the ETH Py150 dataset [46] for fine-tuning and uses `github_repos` dataset under BigQuery's public-data project for pre-training. CodeBERT [19] releases its multi-programming-lingual dataset used for pre-training. Works such as inst2vec [6] and ProGraML [13] uses datasets of code in LLVM compiler intermediate representation to learn generic code representation for various program analyses and optimizations.

Other. Apart from code datasets, there are many other graph datasets. Open Graph Benchmark [25] suite presents graphs that are used for machine learning tasks such as GNN inference and training. GAP [5] and Graph Based Benchmark Suite (GBBS) [16] provide large-scale curated sets of graphs, primarily for evaluating traditional graph problems. SuiteSparse [34] consists of a wide variety of sparse matrices, which can be viewed as graphs. Most of these datasets are for node-level or edge-level prediction tasks. TpuGraphs is by far one of the largest graph property prediction datasets. TpuGraphs' average graph size is comparable to that of MalNet [20] — the largest scale graph property prediction dataset to date — while offering 25x more combinations of graphs and configurations. Other popular graph property prediction datasets include small molecule [45, 47], bioinformatic [17, 25], and social network datasets [48, 55].

B EVALUATION RESULTS

B.1 Ablation Study on Layout Collection

Table 3 compares alternative choices in terms of the model architecture and the training method on the Layout:XLA:Random collection. Our results agree with the results from the prior paper [8] that the quality of the model improves significantly with the Graph Segment Training method (Best) over a typical full graph training (Full Graph), as GST potentially introduces a better hierarchical graph pooling mechanism that leads to better generalization. Note that since an A100 GPU has plenty of memory, training on full graphs does not run out of memory, but one would encounter the problem when running on a smaller device. Similar to the prior paper, we also show that the quality of the model drops when the graph segment size is too small. Here, we compare segment sizes of 100 (Smaller Segment) and 1,000 (Best). Using fewer GNN layers (2 layers), however, does not affect the accuracy significantly compared to the best model (3 layers). The choice of a partition algorithm also does not matter so much, where we compare METIS (Best) and random cutting from topological sort (Topo Partition) because our graphs are sparely connected. Another crucial factor is the loss function, where using pairwise hinge loss (Best) is significantly better than using Mean Squared Error (MSE), similar to the finding in the original TPU learned cost model paper [32]. Additionally, we compare our baseline models with a random model that selects K candidates at random, and show that all of the learned cost models are better than random.

B.2 Ablation Study on Tile Size Collection

Figure 3 compares alternative choices on the tile size collection. Similar to the prior work [32], our results show that combining configuration features with node features early (*early-join*) is superior than combining configuration features with a reduced graph embedding later (*late-join*). Similar to the layout collection, using a ranking loss (ListMLE) is much more effective than using MSE. Additionally, we compare the choice of a GNN between GraphSAGE and GCN, and find that GraphSAGE is slightly better than GCN on this dataset collection. We also provide an MLP baseline without a GNN, and confirm that a GNN is essential to achieve good accuracy.

C EXECUTION TIME MEASUREMENT

We measure the execution of a compiled binary on a *single TPU chip* using *random input data*. Note that some of the graphs in the layout collection must be run on multiple TPU chips. However, doing so is not economically viable for autotuning a large number of models and generating the dataset. Therefore, the autotuner modifies the final optimized graph (after all graph-level optimizations) to make it runnable on a single TPU chip in two ways. First, we replace each collective communication operation such as all-reduce and all-gather with a no-op that simply allocates the right amount of output buffer (with undefined values). This means the measured execution time ignores the time taken by collective operations. We think this is reasonable because layout decisions rarely affect the execution

Table 4: Comparison of TpuGraphs properties with other large-scale graph property prediction datasets. * provide only programs, but one may use them in CompilerGym [14] environment to obtain performance measurements when compiling with specific configurations. † provides randomly generated programs.

Application	Dataset	Graphs (+ Configs)	Avg. Nodes
ML Program Perf	**TpuGraphs (Layout)**	31,091,253	7,705
	TpuGraphs (Tile)	12,870,077	40
	TenSet [59]	51,577,248	5–10
Other Program Perf	Database [24]	300,000	< 100
	BHive [12]	330,018	4
	AnghaBench* [15]	1,041,333	62
	CSmith*† [56]	530,000	5,845
Program Analysis	CodeNet [19]	13,916,868	200–500
	OGBG-CODE2 [25]	452,741	125
	TBCNN [40]	52,000	190
Cybersecurity	MalNet [20]	1,262,024	15,378
Molecule	PCBA [45]	437,929	26
	MUV [47]	93,087	24
Bioinfomatic	DD [17]	1,178	284
	OGBG-PPA [25]	158,100	243
Social Network	Reddit-T [48]	203,088	24
	REDDIT-12K [55]	11,929	391
	REDDIT-5K [55]	4,999	509

time of collective operations. The use of random or undefined data does not affect the execution time of a compute operation, such as convolution, because the timing does not depend on the input data. The second modification we perform is to replace dynamic loop bounds with a fixed loop bounds. Without such replacement, a dynamic loop bound may depend on random input data, resulting in an extremely large loop bound, making the program run unrealistically slowly. Because of these modifications, our absolute execution time measurement may be inaccurate in some cases, but it has been used in production to tune graph-level optimizations and deliver large speedups on many important models. Therefore, we believe it is reasonable to use the execution time measured by the approach outlined here as a prediction target.