Learned TPU Cost Model for XLA Tensor Programs

Samuel J. Kaufman[*]
Paul G. Allen School of Computer Science & Engineering
University of Washington
kaufmans@cs.washington.edu

Phitchaya Mangpo Phothilimthana
Google Brain
mangpo@google.com

Mike Burrows
Google Brain
m3b@google.com

Abstract

At Google, we would like to develop a cost model that can accurately estimate the execution time of a machine learning model running on a Tensor Processing Unit (TPU). This cost model can be used by a compiler to make heuristic decisions, by an autotuner to find an optimal configuration of a specific program, and by Neural Architecture Search to co-optimize accuracy and inference time. However, building an accurate analytical cost model is challenging because of the complexity of modern processors.

We propose to learn a cost model using a neural network. Our cost model uses a feedforward neural network to predict execution time from a graph embedding based on GraphSAGE. Our model's mean predictions are within 13% of the actual execution time.

1 Introduction

Cost models are useful for developing optimizations in compilers, as well as manually optimizing programs and library kernels. For example, LLVM's loop vectorizer uses a cost model to compute the optimal vectorization factor and unroll factor [13], and GCC's auto-vectorizer uses a cost model to decide when to apply loop-peeling, loop-versioning, outer-loop vectorization, and intra-iteration vectorization [8]. A more accurate cost model generally leads to better optimization decisions. In addition, a cost model can be used in a compiler's autotuner as a faster alternative to generating and running code on real hardware. Similarly, Neural Architecture Search [6, 7, 11] can use a cost model to find the fastest model that meets a target accuracy, and vice versa.

At Google, we have developed an autotuner for the XLA compiler [15] that searches for the fastest fusion configurations of an XLA program when running on TPUs v2 and v3 [9]. The autotuner has found up to 15% speedup on some production deep learning models. However, the autotuner evaluates each configuration on real hardware, and search time is dominated by time spent compiling and executing XLA programs. As a result, when tuning large XLA programs, the autotuner is unable to explore a sufficient number of candidates in the search space within a reasonable time limit.

To reduce the search time, we propose to use a cost model to estimate the execution time of an XLA program on a TPU. Building an accurate cost model to predict execution time on a modern processor is notoriously difficult and time-consuming because of the complexity of the processor [3], especially in cases where relevant details of the hardware design are not available. Thus, we are developing a learned cost model using a neural network, which requires far less effort than developing it manually.

[*]Work done during internship at Google Brain in Mountain View, CA.

In particular, we train a neural network over XLA kernel graphs to predict the execution time of a given kernel.

2 XLA Program and Compiler

XLA is a compiler for TensorFlow [1] for various hardware targets, including CPUs, GPUs, TPUs, and other custom accelerators. A TensorFlow graph program can be translated into an XLA graph program, which is an input to the XLA compiler. An XLA program consists of basic blocks, called *computations*. Each computation is represented by a directed acyclic graph called a *computation graph*. A node in a computation graph represents a tensor operation, processing one or more input tensors into an output tensor. An edge connects an output tensor from one node to an input tensor of another node.

The XLA compiler performs optimizations such as tensor layout assignment, operation fusion, and operation scheduling. Before optimizations, a node in a computation graph is a single primitive tensor operation. After optimizations, a node is either a single primitive operation or a fused operation (a fusion of multiple primitive operations). In this paper, we call a node in an optimized computation graph a *kernel*. A TPU executes one kernel at a time, reading from and writing to main memory at start and termination respectively; there is no overlap in execution between two kernels. Therefore, the execution time of a computation graph is, with some minor exceptions, the summation of the execution time of all kernels.

3 Design of Learned Cost Model

The goal of this work is to build a cost model to predict the execution time of each individual TPU kernel. Recall that a kernel is a node in an optimized XLA computation graph, which can be expanded to a graph of primitive operations. For the purpose of predicting cost, we represent a kernel as an undirected graph with nodes corresponding to XLA primitive operations. Inputs to a kernel are expressed by nodes corresponding to XLA *parameter* operations, and outputs are expressed via an extra feature associated with the output nodes.

Figure 1 depicts the architecture of our cost model to predict the execution time (y') of a kernel. We predict y' from the opcodes (x^o), the features of the operations ($\mathbf{X}^f$) and an adjacency matrix ($\mathbf{A}$), capturing the connections of operations in the kernel. A row of $\mathbf{X}^f$ includes, among other things, information about an output tensor shape, tensor layout, striding, padding, and where applicable, convolution filter size.

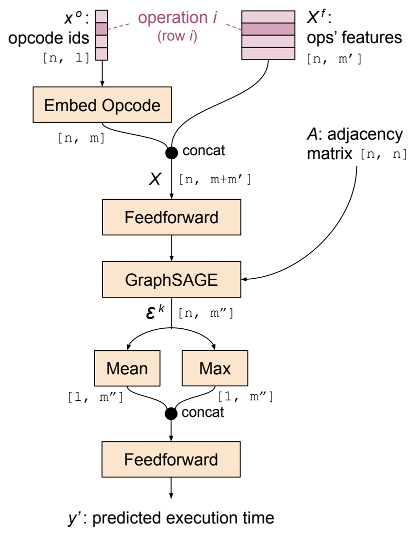

Figure 1: The proposed cost model that predicts the execution time of a kernel.

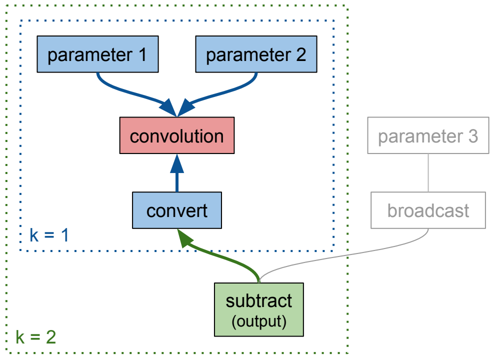

Figure 2: Illustration on how to compute the embedding of the red node using GraphSAGE. An entire graph represents a kernel. Blue and green highlight one-hop and two-hop neighbors of the red node, respectively.

We use graph embeddings to allow the neural network to more easily reason about both the consumers and producers of instructions. We anticipate that it is useful to know whether or not a node is a direct consumer of another node. However, our model does not explicitly encode the directions of edges because such information can be easily determined from properties of neighboring nodes. For example, we can infer the direction of an edge between a parameter node and a non-parameter node because a parameter node consumes no tensors. In future work, we will evaluate the effect of encoding the directions of edges.

Individual operation encoding. The opcode (x_i^o) of an operation i is embedded into a vector of floats via a simple embedding lookup table. An operation's features occupy a fixed region of the $\mathbf{X}_i^f$ vector. For example, the tensor shape feature occupies the first d entries, where d is the maximum number of tensor dimensions the compiler supports. Assume operation i has an output tensor shape of 32×32. The first d entries of $\mathbf{X}_i^f$ will be $32, 32, 0, ...$ If a feature is inapplicable for an operation, we assign zeroes to its corresponding vector region.

Neighborhood-aware node embedding. We then use a single feedforward layer f_1 followed by GraphSAGE [10] to combine information from the opcode, the operation's features, and the graph structure to generate the node's embedding. The embedding of node i considering k-hop neighbors can be computed as follows:

$$\varepsilon_i^k = l_2\Bigg(f_3^k\Bigg(concat\Big(\varepsilon_i^{k-1}, \sum_{j \in neighbors(i)} f_2^k(\varepsilon_j^{k-1})\Big)\Bigg)\Bigg) \quad \text{when } k > 0$$

$$\varepsilon_i^0 = f_1(\mathbf{X}_i)$$

$f_{2...3}^k$ denote feedforward layers with ReLU activation specific to depth k. l_2 denotes L2 normalization of a vector: $l_2(v) = v/||v||_2$. $neighbors(i)$ is a set of immediate neighbors of node i. $\sum$ is a reduction chosen during hyperparameter search (i.e., summation, mean, or element-wise max).

Execution time prediction. Once we have node embeddings ε^k, we create the embedding of the kernel by computing mean and max over rows of ε^k. Then, we pass the concatenation of the mean and max vectors into the final feedforward layer (without activation) to produce the estimated execution time (y') of the kernel.

Optimization & loss function. Kernel graphs vary widely in size, posing a problem for efficient batching of training examples. Our training procedure buckets graphs according to their sizes in memory to produce batches that consume a roughly fixed amount of memory despite containing varying numbers of examples. This optimization increases training throughput significantly.

To avoid overweighting larger kernel examples bucketed into smaller batches, training minimizes the sum of errors rather than the more common mean of errors. This is equivalent to optimizing the mean of errors with weights proportional to the number of examples in a batch. We consider both squared error $(y_i' - y_i)^2$ and absolute percentage error $|y_i' - y_i|/y_i$ as error calculation methods, selected by hyperparameter search.

In addition, since the compute cycles of the kernels in our setting have a wide range and are heavily skewed, we take the natural logarithm of measured runtime as the target; y is the log of the observed compute cycles. To recover compute cycles, we compute $\texttt{exp}(y_i')$. We observe that this improves training and generalization errors over a linear scale.

The model is trained end-to-end with Adam [12].

4 Evaluation

Dataset. We collected our dataset by running the existing XLA fusion autotuner over 107 TensorFlow programs, both used in production and research. Recall that one optimized XLA computation graph consists of multiple kernels; a kernel corresponds to one data point. Since the autotuner tries many fusion configurations via simulated annealing — generating many different optimized graphs —

Target Runtime	**MSE Loss (%)**	**MAPE Loss (%)**	**Baseline (%)**	**Kernel Count**
$<$ **5k**	5.5	6.4	29.9	3,140,858
$\geq$ **5k**	33.6	30.1	25.1	1,142,518
All	13.0	12.7	27.7	4,283,376

Table 1: Mean absolute percentage error (MAPE) of model predictions, grouped by whether the actual runtime is below or equal-to-or-above 5,000 cycles.

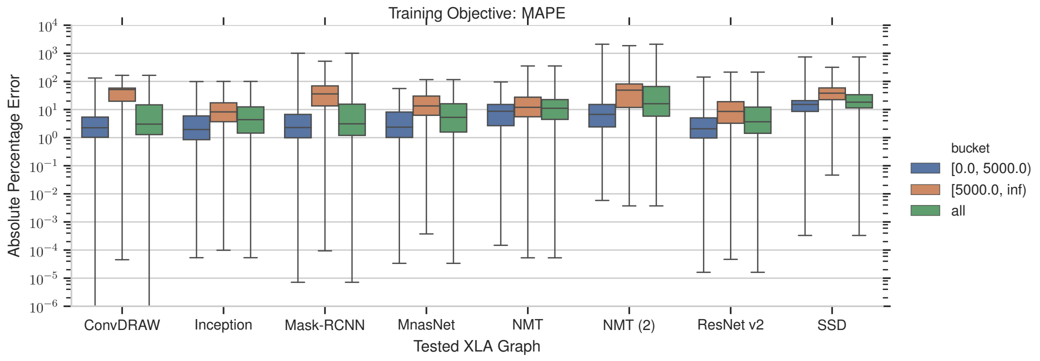

Figure 3: The distribution of absolute percentage error for each kernel prediction from the MAPE-optimizing model, categorized by tested XLA graphs and true runtime ($< 5k$, $\geq 5k$, and combined). Boxes range from the 25th to 75th quartile, and whiskers extend between maximum and minimum runtimes. ConvDRAW's whisker extends below the plot range because its minimum absolute percentage error is zero.

autotuning 107 XLA programs yielded approximately 128 million total examples. Since our goal is to learn a cost model that can generalize to unseen XLA programs, we held out eight TensorFlow programs for the test set — amounting to approximately 8.6 million example kernels — and used the rest as training and validation data.

Targets in our dataset are heavily skewed. Approximately half of our examples have observed cycle counts below 5,000, but these kernels contribute very little to the overall runtime of an XLA program and account for about 1% of the sum of all cycle counts in our dataset. Therefore, we are more interested in the remaining 99% of all kernels, with cycle counts of 5,000 or more.

Baseline. We use the XLA compiler's existing, hand-written cost model as our baseline. This model is used for comparing different tile sizes of the same kernel. Since the cost model is not meant to be used for predicting the runtime of an entire XLA computation graph, estimated costs of different types of kernels (e.g., convolution fusion kernel vs. non-convolution fusion kernel) are in different scales. Therefore, we multiply a predicted cost by a scaling factor according to a kernel's type in order to obtain an estimated runtime in cycles. We calibrate scaling factors for each program in the test set independently. The calibration process involves running all kernels, resulted from compiling an XLA program with the default fusion configuration, on a TPU. A scaling factor of a kernel type is then the division of the sum of kernels' actual runtime over the sum of kernels' estimated costs, considering only kernels of such type. Note that the baseline does not support all kernel types, so the scores reported in this section consider the 99% of kernels that the baseline cost model supports.

Result. Table 1 reports mean absolute percentage error (MAPE) of different cost models for small (below 5,000 cycles), large (5,000 cycles or more), and all kernels in the test dataset. The MSE- and MAPE-optimizing models have similar performance, and overall, substantially outperform the baseline in terms of MAPE: 13% compared to 28%. Additionally, our models' median absolute percentage errors are 4.7% and 3.4% for the MSE- and MAPE-optimizing models respectively.

On large kernels (accounting for 99% of overall runtime), our learned model with MSE loss achieves a MAPE of 33.6%, which is similar to the MAPE-optimizing model (30.1%) and slightly worse than baseline (25.1%). On small kernels, our model outperforms the baseline by a wide margin; the MSE-optimizing model achieves a test MAPE of 5.5% to the baseline's 29.9%. The model trained with MAPE, as opposed to the model trained with MSE, trades worse performance on small kernels (6.4% instead of 5.5%) for improved performance on larger kernels (30.1% instead of 33.6%).

We observe a substantial amount of variation between applications: median APE varies between a low of 3.0% on ConvDRAW and a high of 18.3% on SSD. See Figure 3 for details.

5 Related Work

Many research papers apply machine learning to code optimization. We will focus on ones that use machine learning to build cost models to predict program's execution time.

Ithemal uses a hierarchical recurrent neural network to estimate the throughput of x86-64 basic blocks [14]. Ithemal can estimate throughput with an average error of 9%. While Ithemal focuses on small loop-free programs — which are represented sequentially as sequences of instructions — running on highly complex processors, our work addresses larger machine learning programs with implicit nested loops — which are represented most naturally as graphs — targeting a comparatively predictable accelerator.

The code-feature-based performance model [5] and Halide's cost model [2] use very simple neural nets to predict runtime from manually-engineered features produced by a static analyzer given an optimized program (or a program and a schedule in Halide). Since extracting these features from an XLA graph is non-trivial, we train a more complex neural network using features that can be extracted directly from the XLA graph and sufficient capacity to recover similarly powerful representations.

AutoTVM uses a different flavor of a cost model to guide its search to find a fast configuration for a machine learning program [4]. AutoTVM ranks candidates instead of estimating runtime directly. In addition, AutoTVM models show a limited ability to generalize between kernels and are trained for per-kernel search over a kernel-specific set of parameters. In contrast, we are interested in estimating the runtime of an entire XLA graph as a summation of arbitrarily fused kernels such that our model can be easily extended to reason across kernel boundaries.

6 Future Work

While early results are promising, this is a work in progress. For the next steps, we would like to try the following ideas. First, train and evaluate on a broader dataset of XLA graphs to improve accuracy and better understand the model's ability to generalize. Second, experiment with modifications to the model that encourage generalization between kernels with the same computation graph but different tensor shapes (similar to the approach used in Halide to learn coefficients for hand-engineered performance counters [2]). Third, extend the model to evaluate an XLA graph that varies along other axes, such as kernel's tile size and layout assignment. Finally, we would like to improve performance on large kernels in particular.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[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] Hugues Berry, Daniel Gracia Pérez, and Olivier Temam. Chaos in Computer Performance. *Chaos (Woodbury, N.Y.)*, 16:013110, 2006.

[4] 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.

[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, Apr 2018.

[7] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural Architecture Search: A Survey. *arXiv e-prints*, page arXiv:1808.05377, Aug 2018.

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

[9] Google Cloud. TPU types and zones. https://cloud.google.com/tpu/docs/types-zones. [Online; accessed 21-September-2019].

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

[11] Chi-Hung Hsu, Shu-Huan Chang, Jhao-Hong Liang, Hsin-Ping Chou, Chun-Hao Liu, Shih-Chieh Chang, Jia-Yu Pan, Yu-Ting Chen, Wei Wei, and Da-Cheng Juan. MONAS: Multi-Objective Neural Architecture Search using Reinforcement Learning. *arXiv e-prints*, page arXiv:1806.10332, Jun 2018.

[12] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *CoRR*, 2014.

[13] LLVM. Auto-Vectorization in LLVM. https://bcain-llvm.readthedocs.io/projects/llvm/en/latest/Vectorizers. [Online; accessed 19-May-2016].

[14] 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.

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