Predicting Model Runtimes from Tensor Graphs

Aayush Karan
akaran1@g.harvard.edu
Harvard University
Cambridge, USA

Michael Hla
michaelhla@college.harvard.edu
Harvard University
Cambridge, USA

Victor Goncalves
victorgoncalves@college.harvard.edu
Harvard University
Cambridge, USA

ABSTRACT

A learned cost model for the runtime of compiled tensor computation graphs can offer powerful performance boosts to deep neural network (DNN) training, streamlining the search over exponentially many implementation configurations in hardware. Such performance boosts in time/cost reduction are most pronounced in large scale training and inference, improving deliverable efficiency for commercial products that rely on the synergy of many different integrated DNNs. Naturally, for our cost model, we appeal to the capacity for graph neural networks (GNN) to generate rich feature embeddings of graph structures that can then be fed into downstream classification or prediction tasks. However, due to the massive size of tensor computation graphs for DNNs, individual training samples cannot fit into local GPU memory, imposing a litany of complications when training on a single device. In this project, we propose a learned cost model for tensor computation configurations specifically on the TPUGraphs dataset [3] and explore techniques for training on a single device. To rectify the memory capacity deficiencies, we propose interpretable feature concatenation in the form of *violation ratios* to supplement the GNN. In addition, we examine tradeoffs between feature compression and expanded graph access. We observe up to 1.25x boost in validation OPA score of our learned cost model on TPUGraphs compared to the baseline, and we note this arises not from expanded graph access but rather from both data-driven and architectural improvements in GNN feature extraction.

KEYWORDS

Learned Cost Model, Runtime Optimization, Compiled Tensor Computation Graphs, Attention, Graph Neural Networks (GNN), Feature Embeddings, Downstream Tasks, TPUGraphs Dataset, Memory Capacity, Interpretable Feature Concatenation, XLA, NLP, Tiling, Feature Compression

1 INTRODUCTION

Compilers often predict performance for task-inherent optimization problems, as collecting performance measurements on hardware over a vast number of configurations can be expensive and computationally infeasible. This is especially true for machine learning compilers, where the search space for model training is dense but the performance benefits for a training epoch from optimal configurations are immense. Therefore, we aim to build a model that, given neural network characteristics and a tensor computational graph of the network for a single training or inference step predicts the overall runtime of the model. To this end, Google released the TPUGraphs dataset [3], which is a labeled dataset of tensor computational graphs, associated tile graphs, and total runtimes of their executions. The tensor graphs (otherwise known as layout graphs) represent the tensor operations to be executed, the sequence in which they will be executed, the shapes of the tensor being operated on, the sequence of execution, and the sequence of memory retrieval for each dimension of the tensor (referred to as the configuration). This graph can be further condensed into a tile configuration, where the tensor operations are combined into a fused kernel, representing all of the operations that are executed together as a single unit. These configurations (such as the sequence of memory retrieval and the condensation of tensor operations) influence model performance greatly, and a low cost model to search this space is needed.

The baseline approach for this problem includes a graph neural network with concatenated node feature embeddings used to generate a graph embedding, which is then concatenated with graph config features and then passed into a standard FFNN. The key challenge with improving this baseline learned cost model on a single GPU device pertains to limitations on memory storage – entire dataset graphs cannot fit into local GPU memory, so what quality of feature embeddings can we ensure from the resulting GNN? Accompanying challenges are determining the architectural modifications and manually extracted, data-driven insight features to compensate for reduced graph visibility by the GNN. At a high level, we introduce graph attention to extract more information from the nodes that are visible to the GNN, and we manually extract an interpretable feature called the *violation ratio* that reflects configurational impact on execution runtime. We find that these changes allow for up to a 1.25x boost in validation OPA score of our model compared to the baseline. Furthermore, experiments on the Harvard research cluster demonstrate that even if memory issues are somewhat rectified, inclusion of more nodes of the dataset graphs does not lead to a noticeable performance boost but rather serves to reduce training volatility, emphasizing the power of data-driven insights in our learned cost model.

2 RELATED WORK

2.1 Tile Performance Model

Previous research in machine learning has successfully leveraged machine learning to develop a learned performance model aimed at predicting TPU runtime execution, as highlighted in the study [2]. This model outperforms heavily optimized analytical models in key tasks such as tile-size selection and operator fusion. Furthermore, the model eliminates the need for manual feature engineering, thereby simplifying and streamlining the modeling process. The practical applications are further evidenced through its successful integration into an XLA autotuner, demonstrating utility in identifying and implementing faster program configurations.

Central to this approach is a graph-based neural network (GNN) coupled with a sequence model, designed to effectively evaluate various program configurations. The model's architecture facilitates the creation of embeddings for nodes and operator codes, and is subsequently applied in the TPUGraphs dataset. Our research builds upon this work by focusing on the more complex challenge of predicting runtimes on layout graphs, instead of the tiles. Notably, the referenced study predates the TPUGraphs dataset and therefore does not include results pertaining to training on this dataset. This methodology forms a significant basis for the TPUGraphs dataset and baseline models.

2.2 TPUGraphs

Building off the model described in [2], Google's release of the TPUGraphs dataset marks a significant advancement for machine learning model TPU performance prediction. As detailed in [3], this dataset encompasses an extensive collection of both tile and layout graphs, complete with configurations for each graph and corresponding runtime labels. The dataset boasts an increase of 25 times in the number of graphs compared to the largest existing graph performance prediction datasets. Moreover, the average graph size is approximately 770 times larger than those in current datasets designed for machine learning model performance prediction. This substantial expansion in dataset scale opens up possibilities for developing more accurate models to predict runtime performance on TPUs. Such advancements could lead to significant conservation of compiler compute resources by facilitating the selection of optimal configurations, thereby accelerating production workloads.

However, the introduction of this dataset poses several research questions:

- Training a neural network model capable of making graph-level predictions is challenging when the memory required to train the model on a single graph may exceed the capacity of a single computing device.
- Ensuring that a model generalizes effectively to unseen graphs, especially when the dataset contains diverse graphs and potential data imbalances (ex: more ResNet), is non-trivial.

The study further introduces two baseline models: the tile model and the layout model. The layout model, in particular, encounters specific challenges to the extensive number of nodes in its graphs, averaging between 5,000 to 14,000 nodes. This scale can significantly strain memory resources during training, especially when trying to load entire graphs into memory at once. To address this, the layout model utilizes a technique of random node dropout. Specifically, it randomly selects 1,000 nodes, along with the edges connected to these nodes, and masks the remaining nodes and edges. This method helps manage memory usage during training. In contrast, the tile model proved to be much simpler to train, requiring only a few hours at most with less memory usage and better performance.

3 TENSOR COMPILATION CONFIGURATIONS

Compilers for neural networks are tasked with representing a given tensor computation graph in a format executable by target hardware. A tensor computation graph consists of *operation nodes* and the edges between them, where a directed edge between nodes reflects that the output tensor of node on the tail endpoint is the input tensor for the node on the head endpoint (see fig).

One prominent example of a DNN compiler is XLA [4], which can represent computation graphs on TPU hardware. XLA compiles computation graphs by first grouping connected subgraphs into *kernels* based on subgroups of operations that can potentially be fused and common layout in hardware for a given tensor that flows through the kernel. These kernels are then individually transformed into hardware instructions.

Configurability arises in two locations: once in *layout* at the granularity of individual operator nodes, and once in *tiling* at the granularity of compiled kernels.

3.1 Layout Configurations

Each operator node interacts with memory (whether cached a register, or stored in SRAM/DRAM) as an operation reads in a tensor and outputs a transformed version. In a DNN, a tensor is specified by entries along each of at most six dimensions that are ordered abstractly based first on the batch index and then by a consistent set of dimensions reflecting DNN transformations. However, this ordering can be altered when placed into memory for more efficient read-in/out, especially since the dimensions of the storage almost always will not match those of the tensor. This motivates the need for a *major-to-minor* dimension ordering, where the ordering sets how to interpret a flattened tensor in hardware as a higher-dimensional object.

As a simple example, consider mapping tensor $\mathbf{A} = [[1, 2], [3, 4]]$ to a one dimensional memory register. The 1D hardware representation $[1, 2, 3, 4]$ corresponds to a major-to-minor ordering of $(1, 0)$ as the entries in the second dimension are consecutive, followed by consecutivity in the first dimension. The storage $[1, 3, 2, 4]$ gives a major-to-minor ordering of $(0, 1)$ as consecutive entries first loop over the first dimension, and then the second.

In particular, for TPU hardware targets, memory registers are in the form of 128×8 grids [1], so a high dimensional tensor must be flattened to a 2D storage space. Hence, the choice of the first two orderings of tensor dimensions in major-to-minor convention, as these align with the side lengths of the TPU memory grid. If a dimension of small size is aligned with the 128-side, extra padding will be included that slows down read-in/out, so it is preferred to order tensor dimensions in a way that minimizes such padding.

The choice of permutation in major-to-minor ordering for how a given operation stores its tensor gives the first configurability option for hardware. However, notice that for operator nodes in a given kernel, different orderings for tensor read-in/out for connected nodes requires an additional **copy** operation that copies a tensor from one layout ordering to another, which adds additional overhead (see fig.) Hence, layout configurability introduces the tradeoff between connected subgraph layout compatibility and individual operator interaction with hardware memory. A compiler like XLA can attempt to search through the landscape of this tradeoff, but a learned cost model that can automatically find optimal layout configurations is much more efficient.

3.2 Tiling

The tiling dataset is a heavily distilled version of the layout dataset, where each tensor operation in the layout graph is condensed into a tile, a fused kernel representing all of the operations that are executed together as a single unit. These condensations are determined upon compilation by the XLA compiler with a coded optimization for cache utilization and data locality. The configurations associated for the tiling prediction task are purely relating to input and output dimension of each convolution. These configurations apply to the entirety of a highly simplified subgraph and thus, are inherently less complex than the layout prediction task detailed above. Only one dataset for tiling is provided (XLA tiling) and therefore, the tiling prediction task is less of a focus for the purposes of this paper.

3.3 TPUGraphs

We now explain how layout and tiling configurations are represented in the TPUGraphs dataset.

Each tensor computation graph is stored as a .npz file. The nodes of the computation graph are defined by tensor operations, and the npz file identifies these nodes by their *operation code*, which details the precise operation (**add**, **conv**, **reshape**, etc.), as well as a set of 140 *node features*, which detail information about the incoming and outgoing tensor shapes, along with any additional specifications needed by the operation. Finally, the .npz stores an *adjacency matrix of edges* describing directed flow of the operation graph.

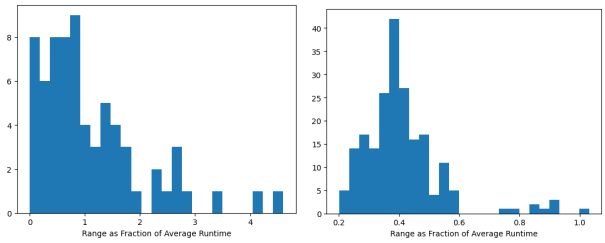
For layout specific information, only a certain number of operation nodes can have the major-to-minor layout configured, so these indices are stored as *configurable nodes*. For each configuration and configurable node, a size 18 *configuration feature vector* details the ordering of the incoming and outgoing tensor dimensions. The tiling dataset follows the same format with key differences being that configuration is at the graph level instead of the node level, and node features are organized into the predetermined condensed tiles.

The computation graphs are partitioned into four main categories, with two sources *XLA* and *NLP* of neural networks and two configuration settings of *random* and *default*.

3.4 Motivation

Given that tiling has been explored in past literature to good success, we focus our cost model efforts on learning layout configuration impact on execution runtime. But to what extent do layout configurations really effect execution runtime?

To answer this question, we ran exploratory statistics on the XLA and NLP training dataset to motivate targeting layout for a learned cost model. For each computation graph, we looked at the distribution of execution runtimes arising from random layout configurations. For each such distribution, we report the range of runtimes (max minus min) as a fraction of the average runtime across configurations. A histogram of these fractions over computation graphs is shown below.



(a) XLA Random **(b) NLP Random**

In a particular example of layout effect, in the XLA Random training set, the **mlperf transformer** runtime can vary by **12.7** minutes per iteration, making an immense difference in training time over the entire computational graph.

4 BASELINE

We adopt the baseline models from the TPUGraphs dataset [3]. They adopt GNNs, tailored to handle the graph-formatted tensor program data.

The foundational structure of these models involves a two-part process in handling node features. Initially, an operation code (opcode) identifier categorizes each tensor operation (like convolution). These opcodes are then converted into embeddings via an embedding table, and subsequently combined with other node attributes. The node matrix is then multiplied by an adjacency

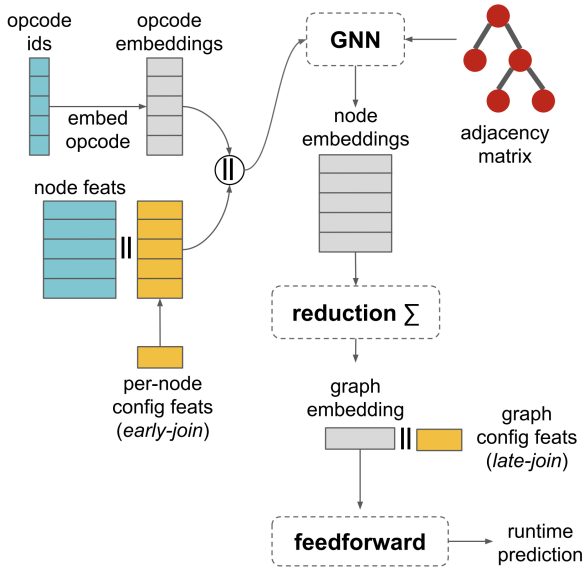


Figure 2: Baseline Model Architecture

mask to only include information on adjacent nodes as input to the GNN. This combined data serves as input for the GNN, which then produces node embeddings based on the contextual information of each node, including its isolated features, specific configuration per input, and yhe contextual information of the other nodes in the graph. These embeddings are aggregated to form a graph-level representation, eventually leading to the final output through a linear transformation process. Baseline models are employed with GCN and GraphSAGE architectures. A visual of the model is shown in Figure 2.

These models are designed with the primary goal of ranking different graph configurations for real-world hardware evaluation. To achieve this, the baselines implement a blend of regression losses, such as Mean Square Error (MSE), and ranking losses, including pairwise hinge loss. These losses are computed within batches and graphs to derive a cumulative loss. Ordered Pair Accuracy (OPA) is used as a validation metric to determine the best-performing model configuration. OPA over all pairs of configurations counts the number of times the configurations are correctly ranked in runtime as a fraction of total number of pairs, as ultimately, the industrail purpose is to extract the configuration with minimal runtime.

The baseline employs two models using TensorFlow-2 and TF-GNN frameworks:

Layout Model: Implemented as a 3-layer GraphSAGE with residual connections, this model integrates both node features and specific configuration features per node. Zero vectors represent non-configurable nodes. The model supports both full graph and graph segment training methodologies.

Tile Size Model: Comprising an MLP model and two GNN variants (GraphSAGE and GCN with residual connections), this model focuses on combining opcode embeddings with node and kernel configuration features. We explore both late and early integration of these features, with the early-join GraphSAGE model paralleling the structure of the original TPU learned cost model.

These models are available for access and exploration in the TPUGraphs GitHub repository[1].

4.1 Metrics

Final scoring based on the Kendall Tau ranking metric assesses the correlation between two ranked lists. It quantifies the similarity by comparing the order of items in the lists, with a higher Kendall Tau value indicating greater concordance and a lower value indicating less agreement between the rankings. Full mathematical description of the scoring can be found on the Kaggle competition site. As a proxy, we use ordered pair accuracy (OPA) to determine model performance.

4.2 Memory Constraints

Initially a challenge identified by Google, we encountered significant challenges related to memory usage when training the GNNs on the TPUGraphs dataset.

A primary bottleneck we observed was the substantial size of the layouts, which posed challenges in loading these large graphs into memory. Additionally, the baseline model pre-cached the entire dataset and loaded it on-demand during the training process. However, this approach led to an initial spike in RAM usage. Specifically, during the initial phase of loading and caching the graphs, we noticed a substantial increase in RAM usage, reaching around 30GB. This high memory demand persisted throughout the training, especially evident when loading and using the graphs, where RAM usage soared to approximately 40-50GB.

As a result, it is necessary to induce a random dropout of nodes. Since the full graphs cannot be included due to memory constraints, the baseline subsamples 1000 nodes from the original tensor computation graph at random. These memory constraints give rise to the main questions our project attempts to examine:

- How can we achieve high quality embedded representations with a GNN if we are subsampling only a small set of nodes of the original computation graph?
- To what extent does restricting input data features to increase the graph nodes kept during dropout actually improve model performance?

5 METHODS

The methods we implement to augment the baseline GNN model seek to generate informative embeddings of the input tensor computation graphs, both architecturally as well as phenomenologically, despite the memory issues mentioned above.

5.1 Attention Layers

Architecturally, we appeal to attention. Motivated by the improvements shown with graph attention in other papers in the field, we decided to implement attention layers in our GNN as the first experiment in model improvement. This was a relatively intuitive implementation, since each contextual node should be weighted differently in the mapping of the node to its embedding. We also want this weighting to be dynamic based on the input node and the relation between contextual nodes, making graph attention a clear choice. This is akin to the application of attention in language models, where the prediction of each token is dependent on a learned weighting of all context words in the sequence prior. For this implementation of attention to be effective, we also reweigh the adjacency mask such that rather than being binary, non-adjacent nodes are weighted with 0.1 instead of 0. This allowed each input to consider all nodes in the sampled graph as input to the GNN, with the intention of allowing the model to produce node embeddings with respect to the sampled process as a whole. Standard multi-head attention blocks were added as an initial layer of both the layout and tile GNN. We also experimented with shifting this layer to different positions in the GNN (ex. attention layer as last layer vs. first) but this showed marginal effect and has thus been omitted from our evaluation section. We have also omitted results on the number of attention heads, as these improvements were marginal.

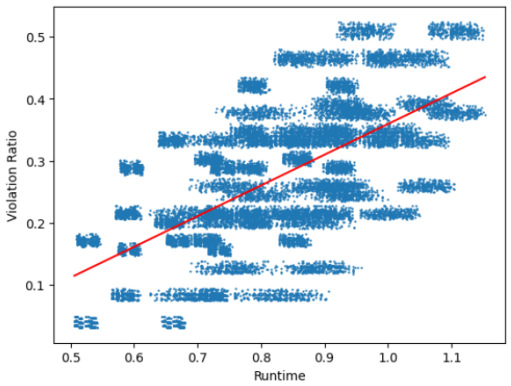


Figure 3: Positive correlation of violation Ratio and runtime for arbitrary computation graph.

5.2 Violation Ratios

Our second method to create a rich embedding of tensor computation graphs is phenomenological, using data-driven insights to manually extract an interpretable feature called the *violation ratio* to augment the GNN.

Recall the intuition from Section 3.1 that TPU memory grids are laid out in 128x8 chunks, so when flattening tensors in major-to-minor memory layout, it is beneficial for the dimension with the largest size to be aligned with the 128 chunk, to avoid unneeded padding. The minimal runtime configuration defines some permutation layout, and the more "swaps" in these layout permutations across all configurable nodes, the more the runtime increases.

Equipped with this intuition, we define the **violation ratio** of a configuration to be the fraction of times over all configurable nodes that the largest dimension size of the tensor is not first in the major-to-minor ordering; i.e., not aligned with the 128-side length of the TPU memory grid. To test the explainability of this feature with regards to execution runtime, over all tensor configurations for a given computation graph, we plot the violation ratio as a function of runtime and observe noticeable positive correlation (see Figure 3).

We are thus motivated to include the violation ratio within our model. However, notice that the violation ratio is a graph configuration-level feature rather than a node or edge-level feature. Moreover, inclusion of this ratio into the learnable weights of the GNN would obfuscate the data-driven interpretability, so we elect to append the violation ratio as an additional feature of

[1]https://github.com/google-research-datasets/tpu_graphs/tree/main

the GNN embedding before passing to the feedforward prediction head.

5.3 Feature Compression

Graph attention and violation ratios attempt to rectify the issue of a lack of rich embeddings due to node dropout from the original computation graph. However, holding these techniques constant, if we restrict the raw npz data features the GNN inputs for the sake of a larger number of nodes kept, does model performance improve?

To test this hypothesis, we select a subset of interpretable operation node features from the original set of 140 that are most closely aligned with layout tensor dimensions and operation runtime (subset of 30 hand selected features from the original provided). With a smaller subset, we can proportionately increase the number of nodes kept during dropout by a factor of 5 to 5000 nodes. We then examine the validation OPA during training.

6 EVALUATION

For brevity, architectural experiments that were performed but did not show any effect have been excluded from the results section of this paper. This includes changing the position of the attention heads, using GraphaSAGE as our base GNN, increasing the hidden dimension size, increasing the batch size, and general hyperparameter tuning. In addition, figures for all four source/setting TPUGraphs datasets are excluded for the sake of avoiding redundancy.

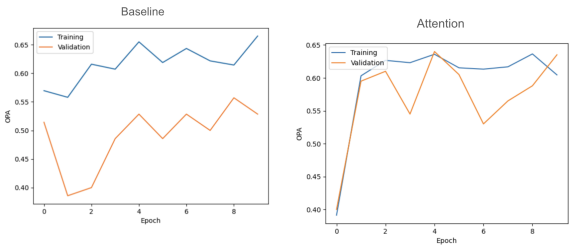
6.1 Experimental Setup

For our model improvement experimentation, we employed the professional Google Colab CPU instance with 50GB of RAM and various T4, V-100, and A-100 Colab GPUs with 16GB of GPU memory as per availability.

Additionally, on the Harvard supercomputer, we executed experiments using both CPU and GPU instances to evaluate various memory-intensive tasks. Our GPU of choice was the NVIDIA A100-SXM4-40GB, complemented by an Intel(R) Xeon(R) Platinum 8358 CPU with 500GB of RAM. The majority of our model improvements, especially with the Graph Neural Network (GNN) architecture and design, were obtained using Google Colab.

6.2 Attention Layers

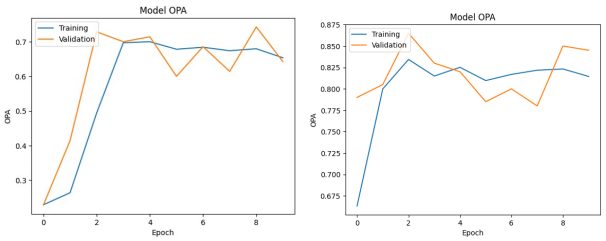
Adding attention layers in the embedding network showed immediate improvement in model performance. For a 16 head attention layer each with hidden dimension size of 32, average model performance improved from a validation OPA of 0.52 to 0.64 after 10 epochs of training. We also note that this result held across all four TPUGraphs datasets, with every implementation of attention performing better or equal to that of the respective baseline.



(a) XLA Random without attention **(b) XLA Random with attention**

6.3 Violation Ratios

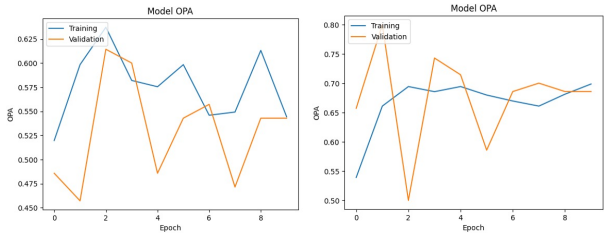
We observe a massive boost in validation OPA scores when training on all four source/setting divisions of the TPU graphs dataset. For example, XLA Random with attention achieves around 0.65 OPA, and adding violation ratios raises performance to 0.75 OPA. On average across all data sets, OPA with attention improved from 0.6 to 0.75 upon including violation ratios, leading to a boost of 1.25x in OPA score. These results confirm the correlation between violation ratios and execution runtime examined during data exploration. The key point of violation ratios is that they embed full graph-level features into a single statistic that is then fed into the feedforward neural network, rectifying not being able to fit the raw tensor computation graphs fully into GNN feature extractions.



(a) XLA Random with violation ratios **(b) NLP Random with violation ratios**

6.4 Feature Compression

Surprisingly, we find that retaining more nodes during dropout by sacrificing apparently unimportant raw features leads to a performance decrease, as shown below. XLA Random OPA drops from 0.7 to 0.625 – a 0.9x multiplier.



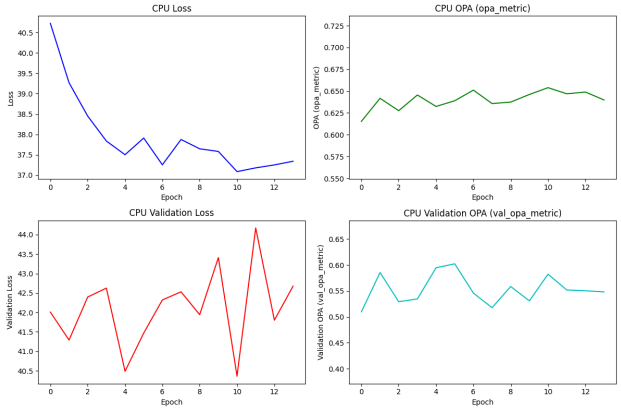
(a) XLA Random with feature compression **(b) XLA Default with feature compression**

These results demonstrate that the raw feature set provided for TPUGraphs is quite important – in a sense, disproportionately more important for model performance than subsampling more nodes from the input graphs. Indeed, this can be justified given the nature of the TPUGraphs dataset: deep neural networks tend to have many repeated operation node patterns, leading to many common subgraphs, so subsampling more of these nodes does not actually enrichen the graph structure presented to the GNN. This gives rise to a method we would like to try in the future, of more informed subsampling of dropout nodes while training to ensure the operator nodes contributing most to the runtime are sampled more frequently.

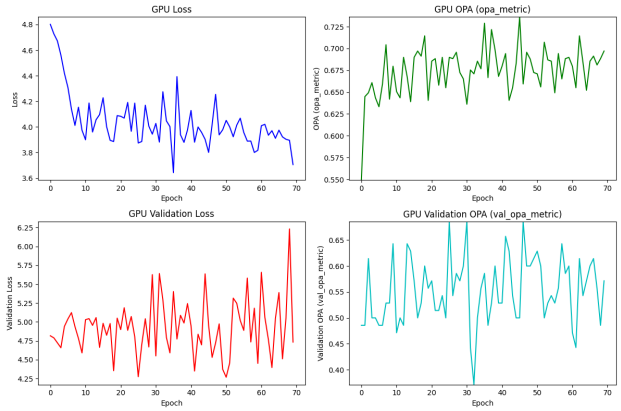
6.5 Harvard Research Cluster

To further explore the extent to which these memory issues constrain our model performance, we turned to the Harvard research cluster to run further comparative experiments.

Our cluster GPU had only half the memory capacity compared to the GPU used in the TPUGraphs baseline. Consequently, we reduced the MAX_KEEP_NODES parameter from 1000 to 500 nodes for random selection, aiming to avoid out-of-memory errors. In a distinct experiment, we increased the batch size from 16 to 32, which refers to the number of configurations loaded into memory simultaneously. Here, we could accommodate a MAX_KEEP_NODES size of up to 10,000, which was well within our memory capacity limits. This adjustment allowed us to load entire graphs into memory for a majority of the TPUGraphs dataset.



(a) CPU Results



(b) GPU Results

Figure 7: Harvard Research Cluster Results

Our experiments revealed a notable increase in the volatility of results across epochs when a smaller portion of the graph was loaded into the GPU memory 7b. In contrast, loading either a majority or the entire graph, as illustrated in Figure 7a, resulted in significantly less volatility between epochs. This observation aligns with expectations, as the model gains more comprehensive insights into the graph structures during training with increased data availability. Notably, in our CPU-based experiments, loading 10,000 MAX_KEEP_NODES with a batch size of 32 resulted in a substantial slowdown, specifically a 5.4x increase in computational time compared to loading 500 MAX_KEEP_NODES with a batch size of 16 on the A100 GPU. We were able to run 70 epochs with the GPU compared to 13 epochs in the same amount of time.

These initial findings highlight the trade-offs involved in loading entire graphs into memory. A balanced approach was found when MAX_KEEP_NODES was set to 1,000 with a batch size of 16. This configuration not only mitigated the issue of increased volatility but also aligned well with the hyperparameters established by the baseline model. We used these hyperparameters for the rest of the experiments to test the performance of our model design. However, our analysis indicated that loading a larger portion of the graph into memory did not substantially enhance the validation OPA, at least in the initial epochs. While we anticipate a marginal improvement in overall OPA over time, the magnitude of this enhancement is expected to be limited, especially when compared to the gains achieved by leveraging key features within the dataset.

6.6 Kaggle Submission

Our final Kaggle submission with violation ratios as a feature and global attention layers placed in the top 20th percentile (107th out of 616 teams) with an average convergence OPA of 0.751. Across the top submissions we notice common themes such as more intelligent node selection algorithms (only sampling configurable nodes and their neighbors), graph attention applied to some degree in the GNN, and ensembling results in some manner.

7 DISCUSSION

We observe from the Kaggle competition key motifs present in all of the top solutions and in future works would aim to implement some form of these. For example, we propose an improved subsampling scheme of the graph by selecting only the configurable nodes and self defined nodes of importance, allowing our model to run with greater batch size and avoid memory constraints while aiming to increase performance. Our proposed definition for "nodes of importance" would be any pair of adjacent nodes where the layout configuration dimension changes from one node to another. We would also like to experiment with ensembling, which could be of high utility due to the reliance on graph sampling during training.

Another potential application during training involves utilizing a graph partitioning algorithm, such as METIS, to segment the graph upon loading, followed by employing Graph Segment Training. This approach enables the distribution of each graph segment across different GPUs for parallel processing. However, it is crucial to note that our experiments did not demonstrate significant improvements when loading a majority of the graph into memory. While METIS and distributed training may enhance training efficiency by leveraging multiple GPUs, we anticipate that these methods may not necessarily lead to increased training accuracies or improvements in validation/test OPA. Instead, our findings suggest that prioritizing key data insights, such as graph compression or selective pruning of graphs by choosing only configurable nodes, is more impactful for optimizing model performance.

We also note the implications of our model for distributed training operations, emphasizing the compatibility of a computational graph-based model with model parallelism and pipelining strategies. By splitting a model into computational subgraphs, we can predict runtimes for each compute node, and even potentially optimize the pairing of hardware with the computational subgraph assigned. While our model will not be able to account for latency and networking constraints, the runtime prediction on each node still has utility for topology design before a massive training job.

While our model provides high performing results compared to the baseline, it is important to note that this explicit model

should not be generalized to non-TPU hardwares without further validation and testing. We may hypothesize, however, that because of the commonalities in the key elements of our runtime prediction, such as the importance of minimized padding between memory layout and the importance of contextual node information in computational graphs, our model may generalize better than anticipated across hardware. We also hypothesize that our model may be robust to any large operation that may be represented as a computational graph, including other large model training paradigms. This is again a result of the common characteristic across computational graph optimization, particularly for machine learning tasks where batching and memory retrieval are highly influential in determining runtime. However, further testing would be needed for both model and hardware robustness.

8 CONCLUSION

This paper has presented an approach for predicting model runtimes from tensor computation graphs, utilizing GNNs on the TPUGraphs dataset. While training on a single GPU device, we were forced by memory constraints to subsample nodes from the original graphs in the datasets and engineer data-driven features and GNN architectures to rectify not being able to fit in entire graphs in memory. Our model leverages attention mechanisms and violation ratios, demonstrating its efficacy in the context of the Google Kaggle competition: Fast or Slow? Predict AI Model Runtime. Our submission ranked 107th out of 616 teams, achieving an average convergence OPA of approximately 0.75.

We further demonstrated using the Harvard cluster that for the TPUGraphs dataset specifically, fitting more nodes into memory does not result in higher model performance upon validation. In fact, the key tradeoff of fitting more vs. less nodes into memory is reducing volatility during training; however, the validation OPA scores of our models were unchanged. We propose to attribute this to the repetitive nature of tensor computation graphs. Rather, our data-driven insights condensing full-graph level attributes into simple statistics easily interpretable by our learned cost model leads to higher performance.

The introduction of TPUGraphs [3] and the demonstrated capability of our model for tensor prediction runtimes indicate a substantial potential for advancements in compiler technology. Our findings contribute to the growing body of knowledge in this field and pave the way for future research endeavors aimed at optimizing compiler efficiency and performance.

REFERENCES

[1] 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 (jun 2020), 67–78. https://doi.org/10.1145/3360307

[2] Sam Kaufman, Phitchaya 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*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 387–400. https://proceedings.mlsys.org/paper_files/paper/2021/file/6bcfac823d40046dca25ef6d6d59cc3f-Paper.pdf

[3] Phitchaya Mangpo Phothilimthana, Sami Abu-El-Haija, Kaidi Cao, Bahare Fatemi, Mike Burrows, Charith Mendis, and Bryan Perozzi. 2023. TpuGraphs: A Performance Prediction Dataset on Large Tensor Computational Graphs. arXiv:2308.13490 [cs.LG]

[4] Amit Sabne. 2020. XLA : Compiling Machine Learning for Peak Performance.