OPTIMIZING COMPUTATIONAL GRAPH CONFIGURATIONS FOR TPU COMPILERS WITH GNNS

BACHELOR THESIS

Bessonov A.
M. V. Lomonosov Moscow State University,
the departament of mathematical methods
of prediction , 417 group
beccohov.a@yandex.ru

Djakonov A.
Phd.
Central Tinkoff University, Russia *
djakonov@mail.ru

ABSTRACT

This paper is focused on the challenge of finding optimal compiler configurations for deep learning models, a prominent research area for both engineers and researchers. The importance of hardware performance evaluation models in code optimization is stressed, as it aid compilers in making heuristic decisions and determining the most suitable configuration for various computational programs. Owing to the large-scale nature of computational graphs, consisting of tens of thousands of nodes and edges, conventional analysis methods prove inadequate. An efficient approach is proposed for ranking computational graph configurations, employing graph neural networks, and considering their expected execution time on TPUs. Our method circumvents excessive computational costs and enables training on extensive graphs, making it a highly advantageous contribution to the field.

Keywords TPU XLA · Compiliers · AI optimization

1 Introduction

Achieving remarkable performance in a broad range of deep learning applications often hinges on increasing the size of models Kaplan et al. [2020]. Nevertheless, the escalation of model parameters introduces numerous challenges, particularly in the realm of training efficiency. Modern Tensor Processing Units (TPUs) and Graphics Processing Units (GPUs) accelerators are sensitive to computational graph configurations (Mangpo et al. [2023], Norrie et al. [2020], Patterson [2018]). Factors such as tensor shapes, transformation orders, and various compiler and operation settings affect the efficiency of an identical computational task Kumar et al. [2019].

Deep learning models can be represented as graphs, with nodes signifying tensor operations (e.g., matrix multiplication, convolution, etc.) and edges denoting tensors. The compilation configuration determines how the compiler modifies the graph for a certain optimization pass. In particular, the XLA compiler (Artemev et al. [2022]) can manage two types of configurations/optimizations, including:

- Layout configuration - directing the placement of graph tensors in physical memory by establishing dimension order for each operation node's input and output.
- Tile configuration - controlling the "tile" size of every fused subgraph within the XLA tiling setup.

To trial suggested approach, the TpuGraphs dataset Mangpo et al. [2023] was employed as primary experimental resource. A primary challenge associated with computational graph exploration resides in their unwieldy dimensions, making standard graph models impractical (Fig. 1 shows specifics of data). Additionally, techniques that address ultra-large graphs prove excessive and yield suboptimal outcomes. On average, layout graphs contain approximately 10,000 to 50,000 nodes, rendering state-of-the-art (SOTA) graph model training use of the full context unfeasible (Rampášek et al. [2022], Chen et al. [2022], Zhang et al. [2023]).

*Science advisor

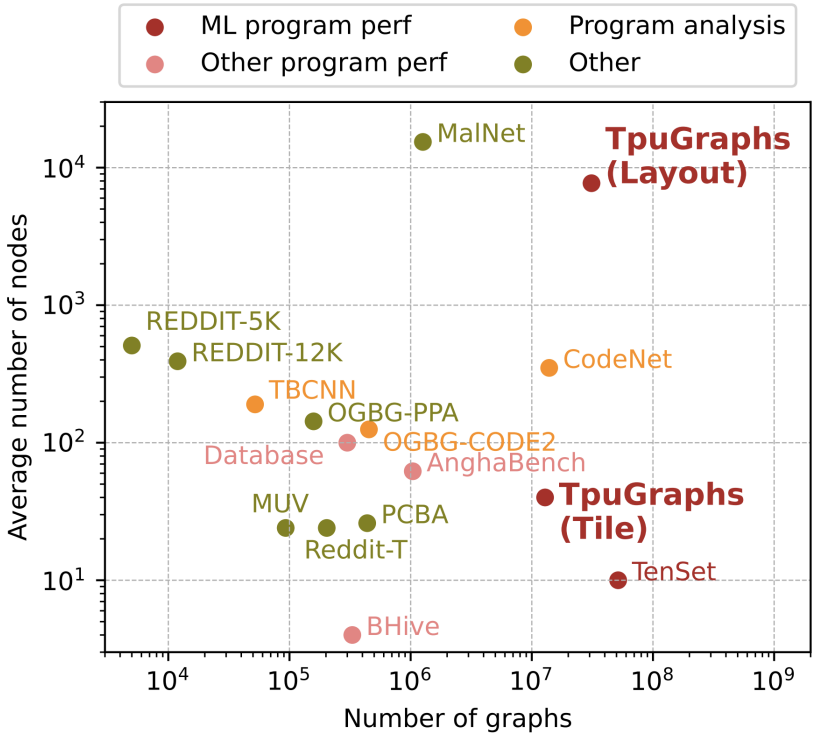

Figure 1: Scale of TPUGraphs compared to other graph property prediction datasets.

To surmount these limitations, the Graph Segment Training (GST) Cao et al. [2023] method is used, enabling the aggregation of large graph properties without necessitating training on the entire graph. This novel technique refrains from error backpropagation across the entire graph during the training process, thus offering a more efficient and effective solution for deep learning applications. It is noticeable that the same problems with extending context are a hot topic in NLP (Dai et al. [2019]) as well as in RL (Bessonov et al. [2023]).

2 Problem statement

The primary task at hand involves obtaining a ranked list of layout configurations for a computational graph's dataset, in accordance with anticipated execution times. To gauge the quality of these rankings, we employ the Kendall rank correlation coefficient as a performance metric, reflecting the extent to which the model-predicted order aligns with the actual order of configurations by execution time.

Popular strategies for seeking optimal configurations, such as genetic algorithms Tağtekin et al. [2021], simulated annealing, and Langevin dynamics Garriga-Alonso and Fortuin [2021], necessitate access to a utility function (i.e., the model's prediction). Consequently, it is essential for the model to effectively maintain the configurations' order, ranging from the fastest to the slowest. This fidelity to ordering can be harnessed when searching for optimal configurations in compilers, substantiating the choice to use the Kendall rank correlation coefficient as the most suitable metric for evaluation (see Eq.1 for reference).

$$\tau = \frac{2}{n(n-1)} \sum_{i<j} sgn(x_i - x_j) sgn(y_i - y_j) \tag{1}$$

Given the distinct characteristics of the TPU dataset, it comprises two types of graphs (refer to Fig 2 for illustration). Each node possesses an operation code, a feature vector, and an optional configuration feature vector specific to layout configurations. The large-scale nature of these graphs demands a training approach that accommodates their enormity. Consequently, we must employ high-quality approximate graph embedding techniques that minimize representation inaccuracies for tractable training, because the majority of large graph models are adopted for extremely large graphs with over many hundreds of thousands nodes Lerer et al. [2019], Xu et al. [2018].

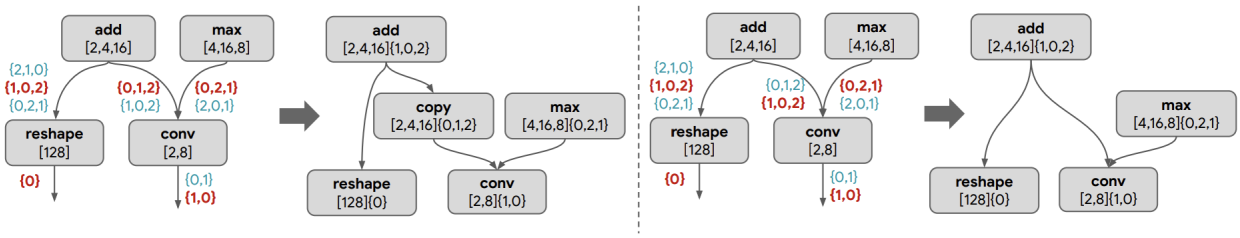

Figure 2: A node represents a tensor operator, annotated with its output tensor shape $[n0, n1, ...]$, where n_i is the size of dimension i. Layout $\{d0, d1, ...\}$ represents minor-to-major ordering in memory. Applied configurations are highlighted in red, and other valid configurations are highlighted in blue. A layout configuration specifies the layouts of inputs and outputs of influential operators (i.e., convolution and reshape). A copy operator is inserted when there is a layout mismatch.

Addressing this problem, the challenges posed by the TPU dataset and emphasizing the necessity for advanced graph embedding methods that account for large graphs while minimizing losses in precision. This concise yet scientific exposition lays the foundation for the exploration and methodology that will follow in subsequent sections.

3 Approach

To tackle the aforementioned challenges, Graph Segment Training (GST) was employed as illustrated in Fig. 3. This method involves dividing each large graph into smaller, controlled-size segments during the preprocessing phase. Throughout the training process, a random subset of segments at each step is selected, updating the model without utilizing the entire graph, i.e.

$$\mathcal{G}^{(i)} = \bigoplus \mathcal{G}_j^{(i)}, \quad j \in \mathcal{J} = \{1, ..., \frac{\mathcal{L}(\mathcal{G}^{(i)})}{\mathcal{K}}\},$$

where $\mathcal{K}$ is maximum allowed subgraph nodes count and $\mathcal{L}(\mathcal{G}^{(i)})$ is a count of nodes in graph $\mathcal{G}^{(i)}$ (see Cao et al. [2023] for more details).

Consequently, intermediate activations must only be maintained for a few segments during backpropagation, while embeddings for the remaining segments are generated without preserving intermediate activations:

$$\bigoplus \mathcal{G}_j^{(i)} = (\bigoplus_{k \in \mathcal{Y}} \mathcal{G}_k^{(i)}) + (\bigoplus_{m \in \mathcal{M}} \mathcal{G}_m^{(i)}), \quad j \in \mathcal{J} = \{1, ..., \frac{\mathcal{L}(\mathcal{G}^{(i)})}{\mathcal{K}}\},$$

$$\mathcal{Y} \cup \mathcal{M} = \mathcal{J}, \quad \mathcal{Y} \cap \mathcal{M} = \varnothing$$

By subsequently merging all segment embeddings, we produce an embedding for the original large graph, facilitating prediction.

The approach ensures that each large graph adheres to an upper bound on memory consumption during training, regardless of its initial size. Consequently, this method enables training on large graphs without encountering out-of-memory (OOM) issues, even when computational resources are limited.

To further enhance the efficiency of the training process, a historical embedding table is incorporated, streamlining the generation of graph segment embeddings that do not necessitate gradients. By leveraging historical embeddings, we bypass superfluous computation on these segments.

Each $\mathcal{G}_j^{(i)}$ embedding is a result of applying three type of encoders, as shown in 3, i.e.:

$$\mathcal{G}_j^{(i)} = \mathcal{M}_1(opcodes) + \mathcal{M}_2(configs) + \mathcal{M}_3(nodes),$$

where $\mathcal{M}_i$ is appropriate embedder (linear layer without bias in this case).

Notably, topological sorting of nodes in the dataset allows for more effective segmentation by taking contiguous subsequences of nodes.

The final prediction is the result of MLP layer applied to $\mathcal{G}_j^{(i)}$. The algorithm is written in List. 1. In the experiments pairwise hinge loss was used.

Algorithm 1 GST TPU prediction algorithm

procedure GST_TPU($\mathcal{G}^{(i)}, b$) ▷ retrieve rank of given graph representation with backward on b parts
  $n \leftarrow \lceil \frac{\mathcal{L}(\mathcal{G}^{(i)})}{\mathcal{K}} \rceil$
  $\mathcal{Y} \leftarrow RandSubset(\{1, ..., n\}, b)$ ▷ Sample min(b, n) items of set
  $E \leftarrow Zeroslike(emb)$ ▷ emb is embedding dimension, this variable keeps final graph embedding
  for $j \in \{1, ..., n\}$ **do** ▷ For each graph part
    $\mathcal{G}_j^{(i)} \leftarrow Slice(\mathcal{G}^{(i)}, (i-1) * \mathcal{K}, i * \mathcal{K})$ ▷ Subgraph is a node and edges slice
    if $j \in \mathcal{Y}$ **then** ▷ $\mathcal{M}$ is a model (GAT)
      $E+ = \mathcal{M}(\mathcal{M}_1(opcodes \leftarrow \mathcal{G}_j^{(i)}) + \mathcal{M}_2(configs \leftarrow \mathcal{G}_j^{(i)}) + \mathcal{M}_3(nodes \leftarrow \mathcal{G}_j^{(i)}))$
    else
      $E+ = StopGrad(\mathcal{M}(\mathcal{M}_1(opcodes \leftarrow \mathcal{G}_j^{(i)}) + \mathcal{M}_2(configs \leftarrow \mathcal{G}_j^{(i)}) + \mathcal{M}_3(nodes \leftarrow \mathcal{G}_j^{(i)})))$
  return E ▷ The resulted embedding which may be used to train model

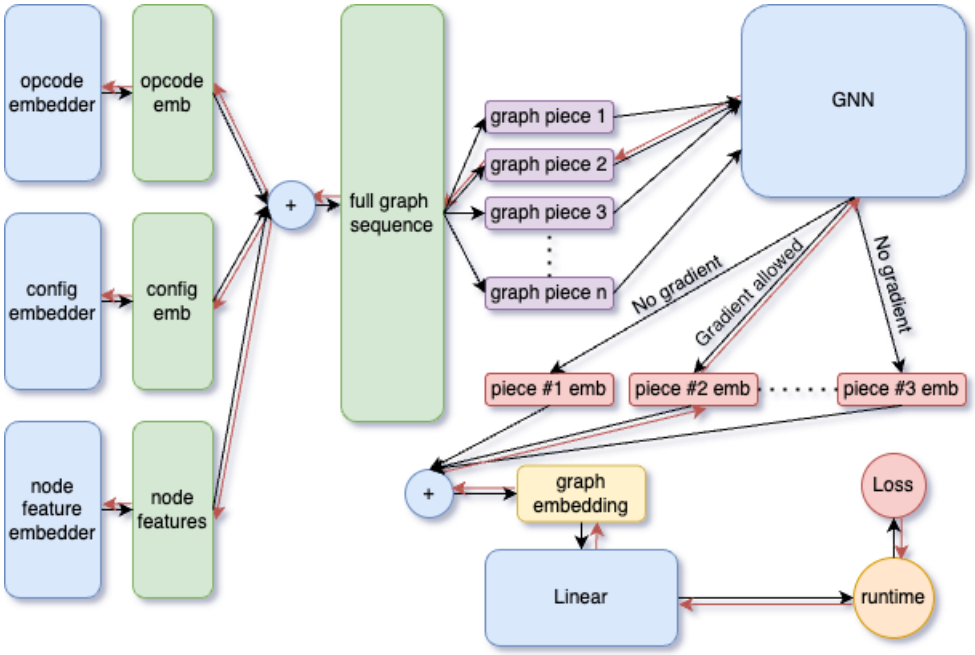

Figure 3: Used GST approach for segment training. Red arrows show the gradient flow during the backward propagation. For each feature type used separate embedder.

4 Experiments

To evaluate Graph Segment Training (GST) approach, two common baseline models were implemented for comparison. The first baseline, referred to as the *Pooling Model*, employs a linear layer over the mean-pooled configuration features and node features across the entire graph.

The second approach, similar to GST, does not include graph feature sequence partitioning. Instead, it selects a random contiguous subsequence of nodes, and prior to backpropagation, halts gradients from all other nodes (see Fig. 4). This model is designated as *Partitioning*. Additionally, we applied the Graph Attention Network (GAT, Veličković et al. [2017]) to this problem for further comparison.

The comparative results can be found in Tab. 1. As evidenced by the experimental data, the GST approach maintains reasonable quality while enabling training on large graphs. Applying GAT directly without the GST method results in memory limitations. Simultaneously, the *Partitioning* training demonstrates reduced performance despite employing the same GAT model, underscoring the significance of the GST method to model performance. Moreover, GST preserves a quality comparable to the GAT model trained on a full graph for moderate graphs, showcasing the strong graph representation learning capability when dealing with large graphs.

Table 1: Comparisson of different methods across the graphs

TPU graph type					
Tile XLA [2]	Layout nlp random	Layout nlp default	Layout xla random	Layout xla default	Method
0.85	0.82	0.42	0.26	0.11	GST
0.45	0.32	0.12	0.11	0.05	Pooling Model
0.67	0.62	0.27	0.21	0.12	Partitioning
0.92	OOM	OOM	OOM	OOM	GAT

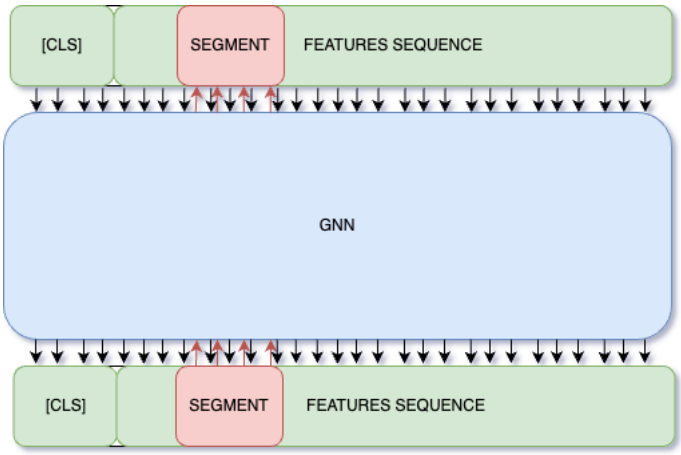

Figure 4: Baseline with graph backward partition. The only subsequence of nodes embeddings chosen to make a backward pass. Forward pass flow shown as black arrows, red arrows show backward pass path.

5 Conclusion

In this study, a solution to alleviate the challenges associated with large graph representation is presented in the context of TPU compiler optimization. Proposed method, Graph Segment Training (GST), enables efficient training on large graphs while consuming significantly less GPU/TPU memory, delivering quality comparable to full graph training. At the same time, this method is not appropriate for common graphs with considerable sizes, because in this case assumption about embeddings aggregation is not reasonable. These limitations show that this is task-specific approach which is, however is well suitable for large scale graphs.

The suggested method allows to build highly accurate TPU compiler's optimizer for large models and effectively utilize resources. For example, given algorithm used for training genetic algorithms to chose optimizations while compiling program for training model.

The suggested method offers a significant advancement in the domain of TPU compiler optimization. Through the integration of genetic algorithms, the optimizer is well-equipped to handle extensive workloads with superior accuracy and efficiency, paving the way for future research and development in this field. And this study presents a novel method aimed at constructing an efficient optimizer for TPU compilers, capable of handling extensive models with high accuracy and optimal resource utilization.

[2]Here used another metric written as $Q = 2 - \frac{min_{i \in K} y_{pred}}{min_{i \in A} y}$, where K = 5 - top predicted configurations actual runtimes, A - real top configurations

References

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Phitchaya Mangpo, Sami Abu-El-Haija, Kaidi Cao, Bahare Fatemi, Charith Mendis, and Bryan Perozzi. Tpugraphs: A performance prediction dataset on large tensor computational graphs. 2023.

Thomas Norrie, Nishant Patil, Doe Hyun Yoon, George Kurian, Sheng Li, James Laudon, Cliff Young, Norman P Jouppi, and David A Patterson. Google's training chips revealed: Tpuv2 and tpuv3. In *Hot Chips Symposium*, pages 1–70, 2020.

David Patterson. 50 years of computer architecture: From the mainframe cpu to the domain-specific tpu and the open risc-v instruction set. In *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 27–31. IEEE, 2018.

Sameer Kumar, Victor Bitorff, Dehao Chen, Chiachen Chou, Blake Hechtman, HyoukJoong Lee, Naveen Kumar, Peter Mattson, Shibo Wang, Tao Wang, et al. Scale mlperf-0.6 models on google tpu-v3 pods. *arXiv preprint arXiv:1909.09756*, 2019.

Artem Artemev, Yuze An, Tilman Roeder, and Mark van der Wilk. Memory safe computations with xla compiler. *Advances in Neural Information Processing Systems*, 35:18970–18982, 2022.

Ladislav Rampášek, Michael Galkin, Vijay Prakash Dwivedi, Anh Tuan Luu, Guy Wolf, and Dominique Beaini. Recipe for a general, powerful, scalable graph transformer. *Advances in Neural Information Processing Systems*, 35: 14501–14515, 2022.

Jinsong Chen, Kaiyuan Gao, Gaichao Li, and Kun He. Nagphormer: A tokenized graph transformer for node classification in large graphs. In *The Eleventh International Conference on Learning Representations*, 2022.

Ziwei Zhang, Haoyang Li, Zeyang Zhang, Yijian Qin, Xin Wang, and Wenwu Zhu. Large graph models: A perspective. *arXiv preprint arXiv:2308.14522*, 2023.

Kaidi Cao, Phitchaya Mangpo Phothilimthana, Sami Abu-El-Haija, Dustin Zelle, Yanqi Zhou, Charith Mendis, Jure Leskovec, and Bryan Perozzi. Learning large graph property prediction via graph segment training. *arXiv preprint arXiv:2305.12322*, 2023.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

Arkadii Bessonov, Alexey Staroverov, Huzhenyu Zhang, Alexey K Kovalev, Dmitry Yudin, and Aleksandr I Panov. Recurrent memory decision transformer. *arXiv preprint arXiv:2306.09459*, 2023.

Burak Tağtekin, Berkan Höke, Mert Kutay Sezer, and Mahiye Uluyağmur Öztürk. Foga: flag optimization with genetic algorithm. In *2021 International Conference on INnovations in Intelligent SysTems and Applications (INISTA)*, pages 1–6. IEEE, 2021.

Adrià Garriga-Alonso and Vincent Fortuin. Exact langevin dynamics with stochastic gradients. *arXiv preprint arXiv:2102.01691*, 2021.

Adam Lerer, Ledell Wu, Jiajun Shen, Timothee Lacroix, Luca Wehrstedt, Abhijit Bose, and Alex Peysakhovich. Pytorch-biggraph: A large scale graph embedding system. *Proceedings of Machine Learning and Systems*, 1: 120–131, 2019.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

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