Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

GNNs for TPU Graphs

Semester Thesis

Gabriel Gramm
ggramm@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Florian Grötschla, Joël Mathys
Prof. Dr. Roger Wattenhofer

January 11, 2024

Acknowledgements

I would like to thank my supervisors, Florian Grötschla and Joël Mathys, for their guidance, insightful feedback, and support throughout my semester thesis. Special thanks to Professor Prof. Dr. Roger Wattenhofer for the opportunity to work with the distributed computing group and contribute to their research.
I also appreciate the distributed computing group for providing me access to computational resources crucial to my research.

Abstract

In this thesis we developed a learned cost model, predicting the runtime of machine learning models dependent on the compilation by the compiler. Our model is trained with the TPUGraphs dataset, containing over 13 million pairs of graphs and compilation configurations. To extract meaningful information from the graph-structured data, we use different Graph Neural Networks (GNN) architectures as the backbone of the pipeline, followed by a reduction and feed-forward step. An important takeaway is that a transformation of the raw, one-hot-encoded features into a more abstract representation, is essential for the network to extract meaningful node embeddings. In our work, these embeddings are produced using a Multi-Layer Perceptron (MLP) trained concurrently to the full pipeline. Among the different models explored in our work, the Graph Attention Network (GAT) achieved the best results, by weighting the importance of specific neighbors of nodes in the graph structure.

Symbols

t	time step, also used for rounds in a network
$G(V, E)$	graph with edge-set E and node-set V
v	node
$e_{v,u}$	edge from v to u
$N(v)$	neighbourhood of node v in the graph structure
h_v^t	node state vector of node v at time step t
Θ	weigh-matrix of an affine transformation

Contents

Chapter 1

Introduction

A compiler is a software that translates source code written in a high-level programming language into machine code. The process performed by a compiler is known as compilation, where it not only ensures code readability for computers but also concurrently performs code optimizations to enhance efficiency using techniques such as elimination redundancy, rearranging instructions, or optimizing the data traffic between random access memory, cache, and processor. The interplay of the compiler with the underlying processor architecture is complex and in certain scenarios extremely time-consuming to model analytically. Also collecting performance measurements from real hardware can be costly or even infeasible for large programs, and because of that compilers often use performance models to attain as good as possible performance. A compiler can use such a performance model to evaluate candidate configuration out of a search space, leading to overall time savings. Many recent approaches utilize machine learning (ML) to learn performance prediction models supporting the compiler.

The concept for this work was introduced in the Kaggle challenge "Google - Fast or Slow? Predict AI Model Runtime". The goal of the challenge is to develop an ML performance prediction model using the provided runtime data for different configurations to predict the runtime of unseen ML models and architectures.

The task poses several research challenges:

- Which prepossessing, feature concatenation, and feed-forward work best?
- 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)?

These problems are addressed in our developed pipeline, based around a GNN serving as a backbone and multiple MLPs for feature embedding and final runtime prediction.

Chapter 2

Background

2.1 Compiling a Machine Learning Model

The tensor computation graph is a representation of a machine learning model, where each node within this graph corresponds to a tensor operation. These operations involve tasks such as matrix multiplication, convolution, or element-wise addition. As stated before, ML compilers address various optimization challenges to compile human-written code into machine language. For this work, the program to be compiled is depicted as a tensor computation graph, which the compiler then transforms into an efficiently executable form for a specific hardware target.
The optimizations within ML compilers can be divided into graph-level and kernel-level optimizations, visualized in section 2.1. Graph-level optimizations try to involve the entire graph's context to make optimal decisions, efficient adjustments made by the compiler may need the entire graph information. On the other hand, Kernel-level optimization focuses on transforming independent kernels at a time. A kernel is a fused subgraph, which is a group of interconnected tensor operations, as shown in fig. 2.3. (Mangpo Phothilimthana, 2023)

Recent studies have illustrated that search-based autotuning techniques can compile code with performance close to optimality (aut, 2019; Chen et al., 2018). The drawback of autotuning techniques is that they demand a relatively large amount of resources to find high-quality compilation candidates. Consequently, many approaches focus on developing a learned cost model to make the autotuning process less resource-consuming (Cao et al., 2023; Kaufman et al., 2021).
Two of the significant factors in building prediction models are tile configuration as explained in fig. 2.3 and layout configurations, depicted in fig. 2.2. Tuning these aspects promises the highest average performance gain compared to tuning and optimizing other tools the compiler can use. (Phothilimthana et al., 2023)

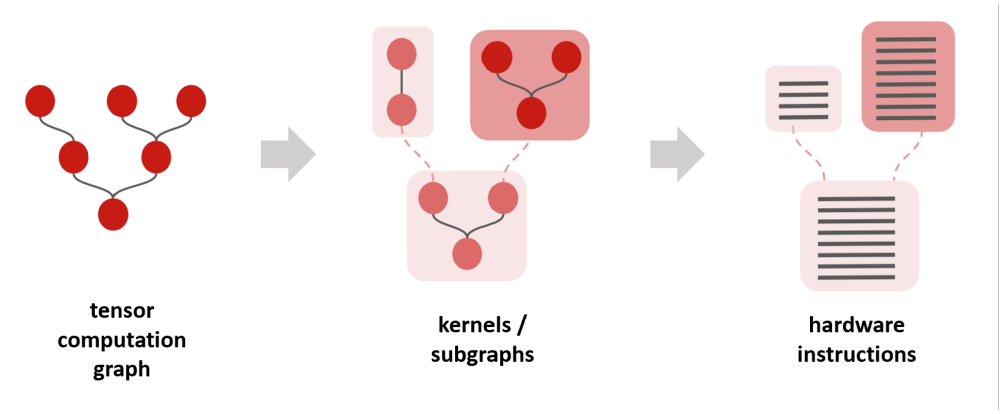

Figure 2.1: **Left:** High-level point of view on how a compiler divides the program into suitable subgraphs or kernels. **Right:** Afterwards the compiler compiles every subgraph individually into machine language. (Phothilimthana et al., 2023)

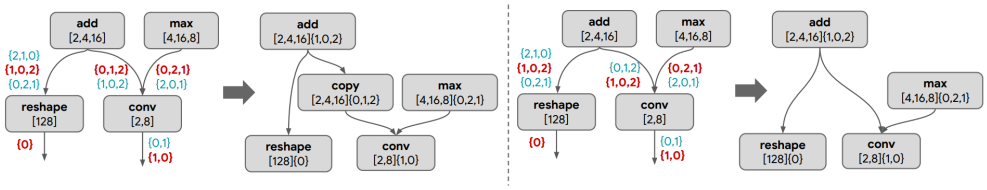

Figure 2.2: The shown nodes each represent a tensor operator, and edges induce the data flow. Every node's possible output tensor shape is displayed inside the node as $[n0, n1, ...]$, where n_i is the size of dimension i. The various layout variations $\{d0, d1, ...\}$ are represented as minor-to-major ordering in memory of the output tensor. The applied configuration is highlighted in red. **Left:** The applied physical tensor output layout of the add operation, forces the compiler to include an additional copy operation, reshaping the tensor to the required input for the following convolution operation. **Right:** The selected physical tensor output layout of the add operation, matches with the required input for the convolution operation, eliminating the need for a copy operation. (Mangpo Phothilimthana, 2023)

2.1.1 XLA and Autotuner

The Accelerated Linear Algebra (XLA) compiler (Abadi et al., 2015) is a production-grade heuristics-based compiler for ML programs that supports tasks such as tuning layout assignment, fusion, memory space assignment passes and compiler flags that control multiple optimization passes. The XLA compiler in combination with autotuning was even able to achieve speedups ranging from 10-20% against state-of-the-art models (Phothilimthana et al., 2023). Although the autotuning reduces runtime of various ML tasks, it often takes a lot of time until the compiler can compute compilation candidates for even a single graph. The substitution of autotuning with a learned cost model could significantly reduce the search time. (Mangpo Phothilimthana, 2023; Phothilimthana et al., 2023)

2.1.2 Kernel-Level Hardware Lowering

In a tensor operational graph, multiple tensors fused together form a subgraph. For example, Convolution and Batch Normalization are two tensor operations that appear frequently in convolutional neural networks (CNN) and are often fused by the compiler into the same subgraph.
At Kernel-level a compiler can achieve improvement of the runtime performance with loop tiling, ordering of computations, unrolling, parallelization, and many more. However, specifically for compiling ML models the critical optimization lies in the selection of tile sizes. Which is choosing the shape of the output tensor of a tile, that simultaneously serves as the input for the subsequent sub-graph in the tensor computation graph. The compiler has to find the tile size, see fig. 2.3, so that, the required input and output data, as well as intermediate data, fit into the local scratchpad memory or cache. The choice of the tile size can be dependent on the underlying hardware. (Phothilimthana et al., 2023)

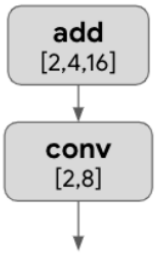

Figure 2.3: Example of a fused subgraph, or kernel consisting of two computational operations. The configurable features per kernel are its output and kernel tile size multipliers. (Mangpo Phothilimthana, 2023)

2.2 Graph Neural Networks

In machine learning an MLP is a framework to learn functions out of data to perform tasks like pattern recognition, classification, regression, and more. While MLPs can solve human-like tasks and have the ability to learn complex relationships within data, they usually require a fixed size of the input data and provide a fixed-sized output.
Today a lot of contemporary and interesting data is represented as a mathematical graph, such as social networks, biological and chemical networks, or recommendation systems. Regular MLPs cannot adapt to the dynamic size of the input within a specific task and so can not be used efficiently to learn algorithms or patterns for these types of data.
Here GNNs provide the necessary ability to handle graph data and their variable input size. GNNs use a neighborhood aggregation approach, where the node's representation tensor is updated through the iterative aggregation and transformation of the states of its adjacent nodes in the graph structure. The algorithm is usually split into two consecutive sections of aggregation of the information across its neighbors and an update of its feature vector. In this way, the node's state update uses the information of the aggregation step from its neighbors and its previous feature vector. These two steps constitute a single round or layer of the algorithm, and multiple rounds together form a GNN. Note that up to round t, a node can only gather information from its $t-hop$ neighborhood and consequently, its feature vector is exclusively dependent on that local environment.

$$a_v^t = \text{aggregate}\left(\{\{h_u^{t-1} \mid u \in N(v)\}\}\right) \tag{2.1}$$

$$h_v^t = \text{update}(h_v^{t-1}, a_v^t) \tag{2.2}$$

Where h_{t-1}^v is the node's state in the previous time step, and $u \in N(v)$ are the node's neighbors with their states h_{t-1}^u in a previous time step. The notation using double brackets represents a multiset, indicating that the network can distinguish between identical neighboring states. (Wattenhofer, 2016; Li et al., 2017)

Chapter 3

Related Work

3.1 Graph Isomorphism Network

While GNNs have revolutionized graph representation learning, there is still a limited understanding of their representational properties. More concretely simple GNNs can struggle to learn to distinguish certain simple graph structures. In this context, Graph Isomorphism Network (GIN) introduces a theoretical framework designed to analyze the expressive capabilities of GNNs in capturing diverse graph structures. The developed architecture stated as powerful as the Weisfeiler-Lehman graph isomorphism test. (Xu et al., 2019)
The graph isomorphism operator:

$$h_v^t = \mathrm{MLP}^t\Big((1+\epsilon^t)\cdot h_v^{t-1} + \sum_{u\in N(v)} h_u^{t-1}\Big) \tag{3.1}$$

Where MLP^t denote a separate neural network for every round t.
Comparing the GIN operator to the in chapter 2.2, one can notice that the difference to the standard aggregation and update procedure is small. The GIN pip-line uses an MLP to learn the update function of the basic in section 2.2 introduced GNN algorithm.

3.2 Graph Sample and Aggregation

The objective of the paper of (Hamilton et al., 2018), is to generate low-dimensional embeddings for nodes in large graphs. The underlying concept of this node embedding approach is to perform dimensionality reduction techniques to compress high-dimensional information of a single node's neighborhood into a dense vector embedding. The paper focuses on extending the extraction of node embeddings, in a way that the algorithm generalizes meaningful to entirely new or unseen graphs. The embedding generation of the Graph Sample and Aggregate (SAGE) algorithm follows closely to a GNN approach but is sampling a fixed-size set of neighbors, instead of using full neighborhood sets. The algorithms workflow

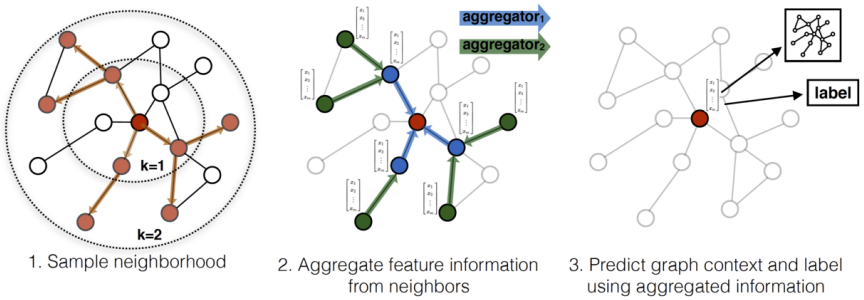

Figure 3.1: The sampling mechanism of the SAGE architecture (Hamilton et al., 2018).

can be seen in fig. 3.1 The results show that this leads to better performance in generalizing to entirely new graphs not encountered during training. (Hamilton et al., 2018)

$$h_{N(v)}^{t} = \text{aggregate}_t\left(\{h_u^{t-1} \mid u \in N(v)\}\right) \tag{3.2}$$

$$h_v^t = \sigma\left(\Theta^t \cdot \text{CONCAT}\left(h_v^{t-1}, h_{N(v)}^t\right)\right) \tag{3.3}$$

Note that $N(v)$ here is not the full neighborhood of node v, but the fixed-size samples neighborhood. The paper defines $N(v)$ as uniform drawn from the set $\{u \in V : (u, v) \in \epsilon\}$, for every round t.
The matrices Θ^t are used to propagate information between different layers of the model.

3.3 Graph Convolution Networks

The paper of (Kipf and Welling, 2017) introduces Graph Convolutional Networks (GCN), a convolution-based method for learning on graph-structured data. The presented architecture is stated as an efficient variant of a CNN which operates directly on graph data. The introduced GNN layers create a first-order approximation of localized spectral filters on graphs. (Kipf and Welling, 2017)

$$h_v^t = \Theta^T \cdot \left(\sum_{u \in N(v) \cup \{v\}} \frac{e_{v,u}}{\sqrt{d_u d_v}} h_u^{t-1}\right) \tag{3.4}$$

with $d_v = 1 + \sum_{u \in N(v)} e_{v,u}$ where $e_{v,u}$ denotes the edge weight from source node u to target node v.

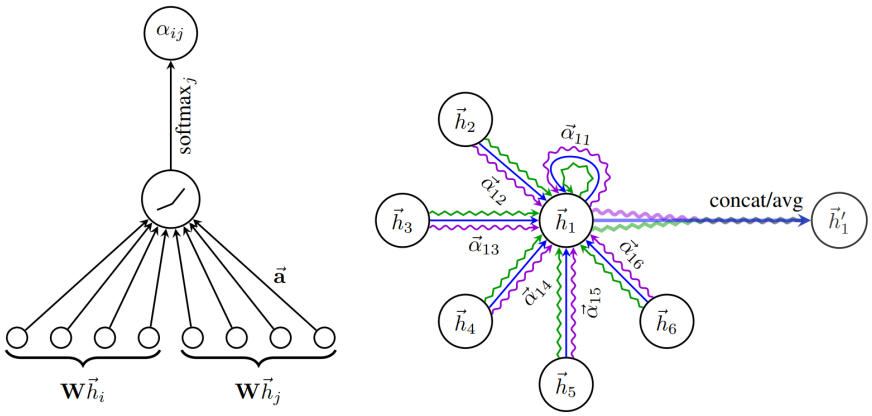

Figure 3.2: **Left:** The attention mechanism $\alpha_{i,j}$, which is parameterized by a weight vector $\vec{a} \in \mathbb{R}^{2F'}$ and fed through a LeakyReLU activation. **Right:** Multi-head attention with 3 heads, the different arrow denote independent attention computations. To obtain the new state of the node, the aggregated features from every attention head are concatenated or averaged. (Veličković et al., 2018)

3.4 Graph Attention Networks

The methods previously discussed for extracting information from graph-structured data have all followed a process of generating node features by treating the features of the immediate neighborhood all the same. To improve that circumstance the GAT tries to focus on neighboring nodes with the most important influence. This is done using an attention mechanism, which is capable of focusing on the most relevant neighbors to extract information. Attention coefficients $\alpha_{v,u}$ are computed for neighbor u determining the significance of its output to node v. It is also possible to calculate multiple attention coefficients for every neighbor, each of these heads uses its own trainable parameters, this is called multi-head attention. In fig. 3.2 an example of the attention mechanism with 3 attention heads is depicted. (Veličković et al., 2018)

$$h_v^t = \alpha_{v,v}\Theta_x h_v^{t-1} + \sum_{u \in N(v)} \alpha_{v,u}\Theta_y h_u^{t-1} \tag{3.5}$$

$$\alpha_{v,u} = \frac{\exp(\text{LeakyReLU}\left(\vec{a}_x^{\intercal}\Theta_x h_v + \vec{a}_y^{\intercal}\Theta_y h_u\right)}{\sum_{k \in N(v)\cup\{v\}} \exp(\text{LeakyReLU}\left(\vec{a}_x^{\intercal}\Theta_x h_v + \vec{a}_y^{\intercal}\Theta_y h_k\right)} \tag{3.6}$$

The attention mechanism a is a single-layer feed-forward neural network, parameterized by a weight vector $\vec{a} \in \mathbb{R}^{2F'}$, and applying the LeakyReLU nonlinearity.

3.5 Graph Segment Training

For very large ML models the tensor computational graphs can become too big to train in on a single machine, but to obtain optimal predictions the information of the entire graph is necessary. The goal of Graph Segment Training is to tackle the problem of large graph data and be able to make meaningful predictions with a constant memory requirement. The proposed approach is to divide the graph in a clever way that the feature extractor is not working on the full graph, which would require all nodes and edges to compute gradients. Despite not working with gradients over the full graph the feature extractor should still get as much updated information as possible. The graph is divided into equally sized smaller segments, and during a gradient update step, only the weights of one of these segments will be adjusted. The embeddings from the remaining segments are still produced in a feed-forward path, but their intermediate activations are not stored. For the prediction, all segments are then combined to generate an embedding for the complete full graph.
While training on one segment the remaining temporally not tuned segments will produce outdated embeddings, lowering the effectiveness of the training step. To address this problem the paper introduces a historical embedding table to produce meaningful embeddings for the segments that are not updated. A high-level overview of the systematics of the graph segment training can be seen in fig. 3.3. (Cao et al., 2023)

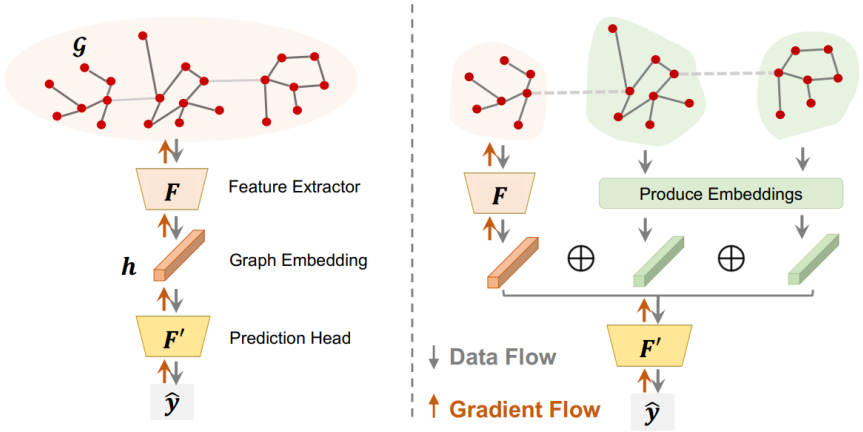

Figure 3.3: **Left:** Training an ML model with classical gradient descent uses all graph information at once, which requires storing all activations of nodes and edges for the entire graph. This might be infeasible for large ML models. **Right:** Graph Segment Training is partitioning the graph into smaller segments. During training, only one segment's weights are updated and thus have to be stored. Meanwhile, the remaining segments are only used in feed-forward, producing embeddings, which are combined to an approximation embedding for the full graph. (Cao et al., 2023)

Chapter 4

Dataset

4.1 Graph-Level Information

In contrast to previous datasets the TpuGraphs dataset is a performance prediction dataset where the data is fully based on tensor programs. The underlying ML program is represented as a graph with nodes and edges, where nodes correspond to one or multiple tensor operations like convolutions, multiplications, or others. The edges induce the data flow between the tensor operations, and consequently, the whole graph represents the primary computations within an ML program like a training or a prediction step. The dataset includes open-source ML programs of popular models (e.g., ResNet, EfficientNet, Mask R-CNN, and a large variety of Transformer), which solve a broad spectrum of tasks like vision, NLP, speech, audio, recommendation, and generative AI. The full dataset averages over 7700 nodes per graph. Since the graph represents the operations and workflow of an ML model, the individual nodes are not as densely connected. On average, the number of edges is around twice the number of nodes per graph. For a single TPU graph, there are numerous configurations, with each configuration constituting a sample. Like that the TPUGraphs dataset results in 13 million pairs of graphs and configurations.

4.2 Node-Level Characteristics

There are many components each sample consists of, the information about the computational graph structure and the node features form the base information and are the same for a batch of samples generated for the same underlying ML model. The node features are split into two tensors, one consisting of the type of the node's operation (e.g., convolution, add, reshape, scatter, or, copy). The other feature vector includes information such as, window size, window dilation, convolution dimensions, slice dimensions, the physical layout used to pack the tensor shape, and more, collectively forming a vector with 140 entries per node. Note that some of these features are relevant only to specific operational types of the nodes. In addition to the graph structure and node features, the dataset

includes configuration features that represent how a sample can be compiled by the compiler. The configuration features include information about kernel output size and the physical layout of the kernel tensor, as stated in chapter 2 these two promise the highest average performance gain. For one ML program, the graph structure and node features remain constant, while each configuration feature serves as an individual sample. The sample compiled with the given configuration is mapped to its corresponding runtime on a Tensor Processing Unit (TPU). (Phothilimthana et al., 2023; Mangpo Phothilimthana, 2023)

- *node feat*: contains `float32` matrix with shape $(n, 140)$. The u'th row contains the feature vector for node $u < n$. Nodes are ordered topologically.
- *node opcode*: contains `int32` vector with shape $(n,)$. The u'th entry stores the op-code for node u.
- *edge index*: contains `int32` matrix with shape $(m, 2)$. If entry i is $= [u, v]$ $(where 0 <= u, v < n)$, then there is a directed edge from node $v \rightarrow u$, where u is a tensor operation consuming the output tensor of v.
- *config feat*: contains `float32` matrix with shape $(c, 24)$ with row j containing the configuration feature vector.
- *config runtime* and *config runtime normalizers*: both are `int64` vectors of length c. Entry j stores the runtime (in nanoseconds) of the given graph compiled with configuration j and a default configuration, respectively. Samples from the same graph may have slightly different *config runtime normalizers* because they are measured from different runs on multiple machines.

The runtime for the final labels is calculated by dividing the configuration runtime by the configuration runtime normalizers. (Phothilimthana et al., 2023)

4.2.1 Data Generation

The graphs are collected from open-source models, where the high-level operations, node features, and graph structure are extracted. Subsequently, the data samples are generated using the XLA autotuner, which complies the the static information with specific configurations and produces the corresponding runtime.

Chapter 5

Method

Our model is based on the architecture from the paper by (Phothilimthana et al., 2023) and designed to extract node features, out of the underlying graph-structured data form the TPUGraphs dataset. The resulting pipeline, illustrated graphically in fig. 5.1, is developed around a GNN that extracts node features based on prepossessed graph-structured data. After the node features are extracted they get compressed into embeddings that include information about that full graph in a as compact form as possible. In the last step of the pipeline, an MLP is used to make a runtime prediction for a particular configuration using the full graph information.

5.1 Prepossessing

Taking a closer look at the dataset and its node features and graph features, it becomes evident that the features are comprehensive and include a broad variety of information presented in a one-hot-encoded format. Consequently, we decided to not make any hand-made feature engineering. The node feature vector is a concatenation of multiple one-hot encodings of a variety of features. The node operation type is encoded by integers indicating the specific operation. To align it to the node feature representation, where each category is transformed into a distinct binary code.
We tried to feed the GNN layers directly with the one-hot encoded feature vectors but later added an MLP embedding step. Initially, the consideration involved utilizing an individual MLP for the one-hot encoding of each feature. However, we later decided that the embedding MLP could extract important information about combinations of features. As a result, the node features and node operations are concatenated first, as depicted in the upper-left corner of fig. 5.1, resulting in an embedding tensor of 258 entries per node. Afterward, the final feature embedding representation is produced by feeding all features into one embedding MLP. The feature embedding MLP consists of one hidden layer, which enlarges the representation of the data to a dimension of 300.

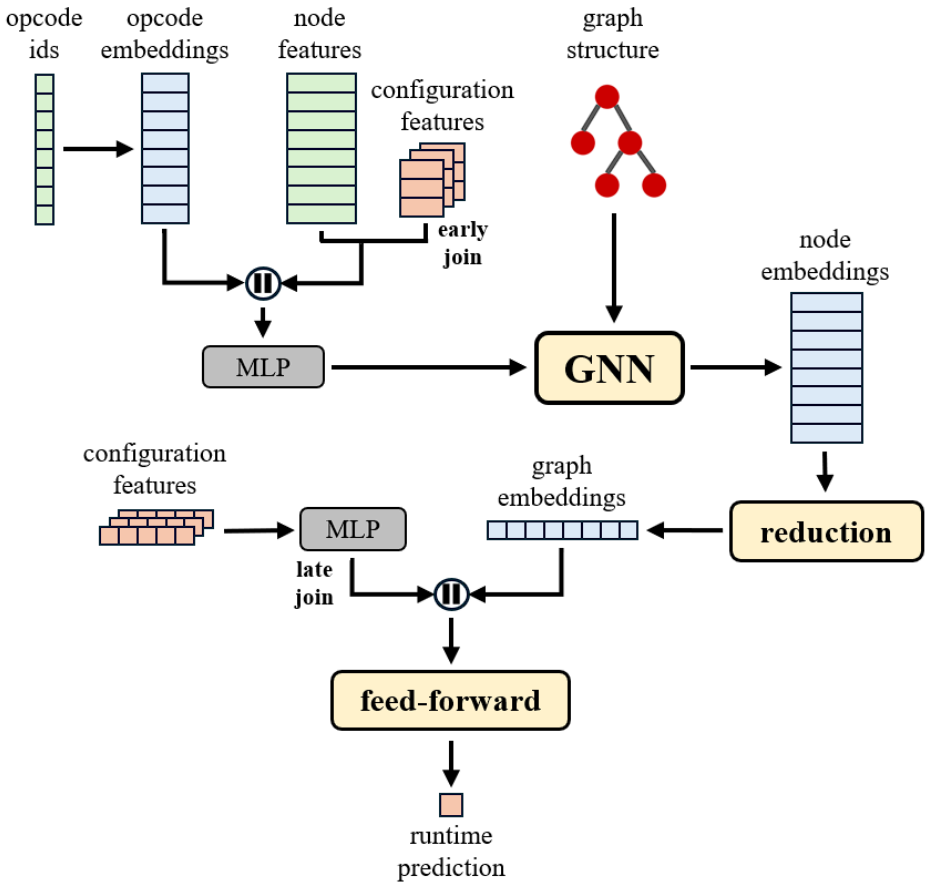

Figure 5.1: Here, our developed pipeline is presented. In green are the node features that remain constant for an entire batch of configurable features, which are depicted in red. Also, the graph structure remains the same throughout the entire batch. The one-hidden-layer MLPs, depicted in grey, are involved in producing representative embeddings of the raw input data. The configurable features can be combined with the per-batch static node features through either an **early join** or a **late join**. The node embeddings produced by the backbone network undergo a reduction step to transform them into embeddings involving information about the whole graph. As the last step in the pipeline, an MLP predicts a single runtime for every configuration feature.

5.2 Backbone Network

To extract meaningful information from the node features in combination with the underlying graph structure, we use several different graph convolutional layers, each designed to perform under specific circumstances. The backbone network is shown as the GNN block in the model overview, of fig. 5.1. In general, we construct the backbone networks with dept 3 to 5, where each layer did a small dimensional reduction. We started by applying a GCN (Kipf and Welling, 2017), which would serve us as a baseline. GCNs extract information from their neighborhood in a way similar to classical convolutions as in CNNs and produce meaningful node representations using spatial filters. The second architecture we tried was the GIN (Xu et al., 2019), which promises to excel in the task of graph isomorphism and thus should distinguish different graphs accordingly. Further, we inserted a SAGE (Hamilton et al., 2018), specifically designed to adapt to new and unseen graphs, particularly in the context of large graph settings. And the last architecture we experimented with is the GAT (Veličković et al., 2018). Similar to the others, the GAT extracts information in its local neighborhood but uses one attention head to decide which neighboring nodes are crucial.

5.2.1 Training and Hyper Parameters

Generally, the training extends over approximately 100 epochs until a diminishing improvement in performance is observed. Across the different model architectures, we set the learning rate to 0.001, momentum of 0.99, and used the Adam optimizer. Since it is a regression task, we choose the loss function to be the standard mean squared error (MSE). Further, to avoid overfitting we use dropout of 0.1 and weight decay of 0.001 for all networks (GNN layers and embedding MLPs). All parameters are also listed in table A.1.

5.3 Reduction and Feed-Forward

The output of the GNN layers is a 64-entry-long embedding vector for each node, based on the local neighborhood of that node. In the next step of our pipeline, we generate a single graph embedding vector from all the per-node features. We use the standard pooling methods mean and max pooling, ending up in a 128-dimensional graph feature vector. At this stage, we concatenate the processed data with configurable graph features as a late join to our pipeline. As with the node features, the configurable graph features are fed through an MLP to create more representative embeddings resulting in a dimension of 50. The last step of our pipeline is mapping the full graph embeddings to a single runtime. This is done by an MLP of the input size corresponding to the graph embeddings

(178), and with 3 hidden layers $(128, 64, 16)$, each reducing the representation dimension and resulting in one single output.

5.4 Evaluation Metric

The evaluation is done over a batch of samples. One Batch includes one set of node features and the underlying graph structure. For all configurations, the model produced the number of configuration runtime predictions. Then out of the $top-k$ predictions, in our case $k = 5$, the evaluation metric considers the best ground truth runtime of the $top-k$ predicted configurations and divides it with the overall ground truth best runtime. This reflects how much slower the $top-k$ configurations predicted by the model are from the actual fastest configuration. (Mangpo Phothilimthana, 2023)

$$1 - \left(\frac{\text{The best runtime of the top-k predictions}}{\text{The best runtime of all configurations}} - 1 \right) = 2 - \frac{min_{i \in K} y_i}{min_{i \in A} y_i} \quad (5.1)$$

Chapter 6

Results

All experiments, except for the difference in the network architectures in section 6.4, were done using the identical setup, with the only modification being the specific factor mentioned for each individual experiment. The whole pipeline used for the experiments is discussed in chapter 5 and in particular in fig. 5.1. The SAGE architecture, as introduced in section 3.2, serves as the backbone for the initial three experiments.

6.1 MLP Embeddings

The results of Table 6.1 point out the importance of generating feature embeddings with an MLP. The embedding step increases the model's performance by e good margin and makes it more stable in score and validation loss, as visible in fig. 6.1. The best performance was achieved by feeding the entire feature vector to a single network, likely because it allows the information of the features to merge before the relatively shallow backbone networks (3-5 layers), which may struggle to create useful embeddings in these few rounds.

MLP Embeddings			
Network Architecture	Score	Training Loss	Validation Loss
Without MLP Embeddings	0.602	0.527	0.541
MLP Embedding of Individual Features	0.684	0.331	0.345
MLP Embedding of full Feature Vector	0.711	0.277	0.281

Table 6.1: Shows the comparison of different input feature vectors for the backbone, by using the raw, feature-wise one-hot-encoded TPUGraphs features versus generating feature embeddings with a one-hidden-layer MLP. We experimented with feeding each raw feature individually to its own MLP and concatenating all the resulting embeddings, and also with feeding the entire concatenated feature vector to a single MLP.

Figure 6.1: The plot illustrates the training curve of the comparison of using MLP-generated embeddings versus feeding the raw one-hot-encoded data to the SAGE backbone network.

6.2 Merging of Configurable Features

As visible in table 6.2 and table 6.3, neither the location of joining the configurable features and node features nor the depth of the backbone network results in a significant difference in the model's performance.
Regarding the joining of features, one plausible explanation is that the model may struggle to extract valuable information from the configurable features, because of its inability to preserve information across GNN layers or through a potential information loss during the reduction step. Regarding the small benefit of an early join, as depicted in table 6.2, it is overall more efficient to join the configurable features after the backbone in the pipeline. This also leads to reduced computation and memory usage for the entire model.

Concatenation of Configurable Features before and after the GNN aggregation			
Network Architecture	Score	Training Loss	Validation Loss
Early Join	0.647	0.287	0.291
Late Join	0.659	0.275	0.279

Table 6.2: Comparing the location in the pipeline of joining the configurable features and node features, first as an early join before the aggregation through the backbone, and second as a late join right after the backbone network.

6.3 Number of GNN-Layers

As observed in table 6.3, a deeper network architecture is facing challenges in generating consistent and reliable predictions. This may arise from difficulties of the backbone in maintaining the data in a form that the following steps of the pipeline can use for accurate decisions and predictions.

Network Architectures			
Number of Layers	Score	Training Loss	Validation Loss
3 Layers	0.684	0.257	0.260
5 Layers	0.689	0.242	0.257
7 Layers	0.617	0.334	0.382

Table 6.3: Comparison of the dept of the backbone network of the pipeline.

6.4 Backbone Network Architectures

Among the various backbone architectures, the GIN performed the worst, as seen in table 6.4. Its architecture is primarily focused on distinguishing entire graphs based on their structure. The diversity of the underlying graph structures in the dataset may be too big for the GIN to predict accurate runtimes.
Both Graph SAGE and GCN accomplish similar performance. The SAGE architecture is well-suited for large graph structures. However, the runtime can be highly dependent on specific single operations, the SAGE's sampling technique, which does not consider all neighboring nodes when calculating the next state, may occasionally miss crucial information for accurate predictions.
The GCN, Graph SAGE, and GAT architectures exhibit similar performance in generalizing to new, unseen data, as evidenced by the variation in the validation loss and score, shown in table 6.4.
The GAT architecture achieves the best performance in our experiments. Its attention mechanism allows the GNN layers to focus on the neighbors that are most relevant for solving the task of runtime prediction. This result could be supported by the observation of Google's paper on the TPUGraphs dataset (Phothilimthana et al., 2023), that within the workflow of machine learning models, specific operations, e.g. convolutions, often have a more significant impact on the runtime than other operations. Therefore, it is advantageous to pay more attention to them.

Network Architectures			
Network Architecture	Score	Training Loss	Validation Loss
Graph Convolutional Network	0.695	0.241	0.236
Graph Sample and Aggregation	0.686	0.238	0.239
Graph Isomorphism Network	0.611	0.321	0.374
Graph Attention Network	0.779	0.198	0.210

Table 6.4: Evaluating various architectures as the backbone for the pipeline, we explore different well-known structures, each built to excel in a specific task compared to the others.

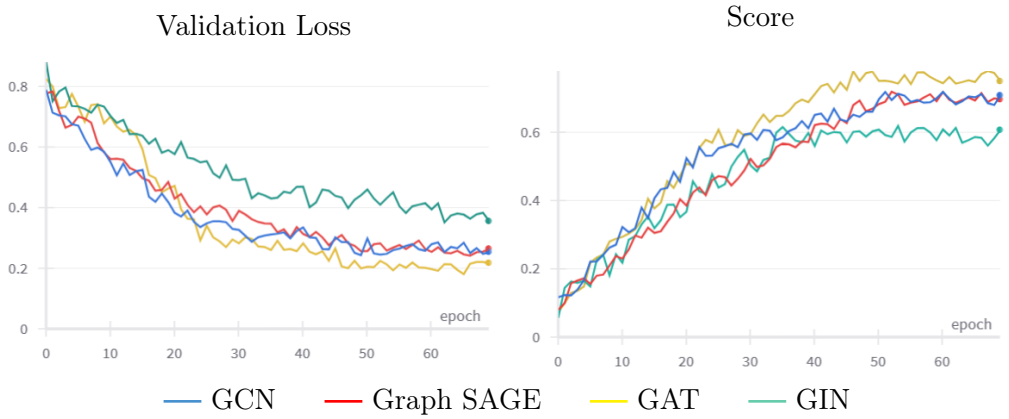

Figure 6.2: Training curve of the validation loss and score for the experiments with various backbone architectures.

Comparing our results with those from the Kaggle Competition reveals that we can match the performance of the top competitors, surpassing the baseline by a significant margin. (Mangpo Phothilimthana, 2023)

Chapter 7

Discussion and Future Work

The goal of this work is to extract meaningful information from sparse and large graph-structured data of the TPUGraphs dataset. Modifications in the data representation and backbone architecture achived the most substantial performance improvement in our experiments. We did not do any additional feature extraction on the provided data nor did we generate more TPU data on our own.

7.1 Data representation

In the early stage of our work, the backbone network was not able to process the data and create meaningful embeddings. The crucial coming through was to use a one-hidden-layer MLP to generate node embeddings, enable the GNN layers to extract information from the data more efficiently, and increase the model's performance significantly, as depicted in table 6.1.
In our experiments deeper backbone architectures did not perform better, since the pipeline overall includes a lot of trainable building blocks, an increase in network dept may make the model too complex. This complexity could lead to the model not being able to extract useful information from the configurable features joined as an early join before the GNN layers. This could lead to the information not being preserved across the backbone network or experiencing partial loss during the reduction step. Similarly, for the number of network layers, a deeper network may encounter difficulties in maintaining the data in a form so that the following parts of the pipeline can be used for accurate predictions.

7.2 Backbone Network

In this work, we encounter the challenge of training a stable prediction pipeline around a GNN backbone. As the validation data may significantly differ from the training data, the model needs to generalize to new, unseen graph structures. The prepossessing step allows the network to concentrate on the aggregation of the information within the graph structure, leading to increased stability and decreasing fluctuation in score, training, and validation losses also for new unseen graph structures.

This thesis highlights that in even large graphs, the architecture of the backbone must be able to identify crucial neighboring nodes, as these may contain much more vital information than others. In this particular task, architectures that sample from the neighborhood directly or average/sum over all neighboring states may fail to focus on crucial information or even entirely miss it. While the GCN and SAGE architectures have nearly the same performance, the GAT architecture is best suited for addressing this concern focusing the model's attention to the important operations in the TPU graph.

7.3 Hierarchical Graph Pooling

The reduction step in our pipeline is done by averaging and maximizing over all node embeddings generated by the backbone network. This is reducing the data from a $\mathbb{R}^{n\times 64}$ dimensional tensor to a 128 feature-long graph embedding vector. This step is entirely decoupled from the backbone architecture, but it might be beneficial to include it directly in the backbone network to avoid losing too much information in this non-learnable averaging step. This could be done with hierarchical graph pooling layers, that are included in the backbone either, as the last layers or interspersed in between, step by step reducing the size of the data.

Bibliography

(2019). Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics*, 38(4).

Abadi, M., Agarwal, A., and and, P. B. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Cao, K., Phothilimthana, P. M., Abu-El-Haija, S., Zelle, D., Zhou, Y., Mendis, C., Leskovec, J., and Perozzi, B. (2023). Learning large graph property prediction via graph segment training.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E. Q., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. (2018). TVM: an automated end-to-end optimizing compiler for deep learning. In Arpaci-Dusseau, A. C. and Voelker, G., editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 578–594. USENIX Association.

Hamilton, W. L., Ying, R., and Leskovec, J. (2018). Inductive representation learning on large graphs.

Kaufman, S. J., Phothilimthana, P. M., Zhou, Y., Mendis, C., Roy, S., Sabne, A., and Burrows, M. (2021). A learned performance model for tensor processing units.

Kipf, T. N. and Welling, M. (2017). Semi-supervised classification with graph convolutional networks.

Li, Y., Tarlow, D., Brockschmidt, M., and Zemel, R. (2017). Gated graph sequence neural networks.

Mangpo Phothilimthana, Sami Abu-El-Haija, B. P. W. R. A. C. (2023). Google - fast or slow? predict ai model runtime.

Phothilimthana, P. M., Abu-El-Haija, S., Cao, K., Fatemi, B., Burrows, M., Mendis, C., and Perozzi, B. (2023). Tpugraphs: A performance prediction dataset on large tensor computational graphs.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., and Bengio, Y. (2018). Graph attention networks.

Wattenhofer, D. R. (2016). Pricipals of dirsributed computing.

Xu, K., Hu, W., Leskovec, J., and Jegelka, S. (2019). How powerful are graph neural networks?

Appendix A

A.1 Hyper Parameters

Hyper Parameters	
Learning Rate	0.001
Momentum	0.99
Optimizer	Adam
Loss Function	Mean Squared Error
dropout	0.1
Weight Decay	0.001

Table A.1: List of static parameters used for training in the various experiments.

A.2 Graph Level Configuration Features

Index	Tile Size
0	kernel_bounds_0
1	kernel_bounds_1
2	kernel_bounds_2
3	kernel_bounds_3
4	kernel_bounds_4
5	kernel_bounds_5
6	kernel_bounds_sum
7	kernel_bounds_product
8	output_bounds_0
9	output_bounds_1
10	output_bounds_2
11	output_bounds_3
12	output_bounds_4
13	output_bounds_5
14	output_bounds_sum
15	output_bounds_product
16	input_bounds_0
17	input_bounds_1
18	input_bounds_2
19	input_bounds_3
20	input_bounds_4
21	input_bounds_5
22	input_bounds_sum
23	input_bounds_product

Table A.2: Input and output tile sizes.

A.3 Node Level Configuration Features

Index	Physical Layout
0	output_layout_0
1	output_layout_1
2	output_layout_2
3	output_layout_3
4	output_layout_4
5	output_layout_5
6	input_layout_0
7	input_layout_1
8	input_layout_2
9	input_layout_3
10	input_layout_4
11	input_layout_5
12	kernel_layout_0
13	kernel_layout_1
14	kernel_layout_2
15	kernel_layout_3
16	kernel_layout_4
17	kernel_layout_5

Table A.3: Physical layout of the output and input tensor of the node. For convolutions also the layout of the kernel tensor.