GTuner: Tuning DNN Computations on GPU via Graph Attention Network

Qi Sun
CUHK

Xinyun Zhang
CUHK

Hao Geng
ShanghaiTech University

Yuxuan Zhao
CUHK

Yang Bai
CUHK

Haisheng Zheng
SmartMore

Bei Yu
CUHK

Abstract

It is an open problem to compile DNN models on GPU and improve the performance. A novel framework, GTuner, is proposed to jointly learn from the structures of computational graphs and the statistical features of codes to find the optimal code implementations. A Graph ATtention network (GAT) is designed as the performance estimator in GTuner. In GAT, graph neural layers are used to propagate the information in the graph and a multi-head self-attention module is designed to learn the complicated relationships between the features. Under the guidance of GAT, the GPU codes are generated through auto-tuning. Experimental results demonstrate that our method outperforms the previous arts remarkably.

1 Introduction

The great successes of deep learning algorithms in this AI era stimulate the fast development of high-performance computing for deep neural networks. Many techniques have been developed to optimize the on-chip inference performance, tackle heavy workloads, and bridge the gap between hardware designs and algorithm developments. Some representative arts include algorithm compression and pruning [1, 2], hardware/algorithm co-optimization [3], *etc.*

Some compilation frameworks have been proposed to optimize the model inference on GPU. Halide [4] and TVM [5] propose to decouple the model analysis and backend code optimization and use the auto-tuning algorithms to tune the optimal deployment configuration for DNN models. The models are analyzed and partitioned into some small subgraphs. Each subgraph is implemented as a *kernel* on GPU and some parameters in these kernels are tuned to achieve the best performance. Candidate values of these parameters are termed *knobs* [6], or *annotations* [7]. The genetic algorithm (GA) and simulated annealing (SA) are the typical parameter-tuning algorithms.

Based on TVM, many techniques are proposed to help tune the kernel implementations. AutoTVM [8] provides some fixed optimization rules and code templates for the DNN operations on GPUs and encodes the parameters in the templates as fixed-length feature vectors. These features contain the numbers of on-device threads, virtual threads, blocks, *etc.* In the genetic algorithm process, AutoTVM interacts with GPU to collect the on-device performance and trains an XGBoost model as the performance estimator of the feature vectors to guide the search for optimal parameters. The optimization process is slow, and no historical tuning data are utilized. Based on AutoTVM, an advanced active learning method [9] is designed to learn representative parameters in the optimization process. CHAMELEON [6] introduces reinforcement learning to learn the searching strategies from the history tuning data and adapts the searching space during optimization. Guided Genetic Algorithm (GGA) [10] improves AutoTVM by using some heuristic rules to guide the genetic algorithm. The similarities between new deployment tasks and the history tuning data are computed to measure the performance of new knobs. DGP-TL [11] proposes to use deep Gaussian process models to learn the history data and use transfer learning with fine-tuning to guide the new DNN deployment tasks. These methods prove the effectiveness of the learning-based methods. Further, Ansor [7] designs some complicated rules to generate code "*sketches*" for the computational subgraphs in models to break the shackles of fixed templates. The sketches are high-level program structures and leave billions of low-level parameters as "*annotations*". The features representing these codes are *statistical* which are more complicated, including parallelism in multiple programming levels, type of memory access, the number of touched cache lines, the number of floating/integer multiplication-add operations, *etc.* With the advantages of flexible sketches and annotations, Ansor outperforms the AutoTVM-based previous arts significantly.

Despite the advancements, existing frameworks are still unsatisfying. Firstly, the *structural* information of the computational subgraphs is underutilized. Existing techniques rely on the statistical information of codes to train a cost model as the performance estimator [6–8, 10, 11], while the structural information is discarded. The structures reflect the topological relationships and scheduling information of the operations, which greatly influence the inference performance, while the statistical features fail to characterize the structures. Modern DNN models contain different structures. Learning-based techniques which only rely on statistical information are incapable of measuring these diversities. Secondly, the feature items in the statistical feature vectors are treated equally, despite their physical meanings and relationships related to the implementation details. Each statistical feature vector is usually directly passed to the learning models to predict its performance. Therefore, the complicated but implicit relationships between the feature items are not taken into considerations.

To tackle the above problems, a novel method, GTuner, is built based on TVM and Ansor to tune the computations of deep neural networks on GPU. The structural information of the computational graphs and statistical code features are utilized. The complicated

DAC '22, July 10–14, 2022, San Francisco, CA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9142-9/22/07…$15.00
https://doi.org/10.1145/3489517.3530584

relationships between the features are learned automatically. The contributions are summarized as follows:

- A novel method, GTuner, is proposed with a graph attention network (GAT) as the performance estimator. GAT comprises a graph neural network (GNN) module to aggregate structural information and a multi-head self-attention (MHSA) module to mine inter-feature relationships.
- Structural information of the computational subgraphs is extracted from the intermediate representations of the compilation flow with the help of our code parser and analyzer. Then the information is propagated and aggregated via the graph neural layers to learn high-quality features for the graphs.
- The MHSA module is designed to learn the complicated but implicit relationships between the structural and code statistical features via the self-attention mechanism. The drawbacks of losing structural information and long-range dependencies between the features are overcome.
- With the GAT, GTuner optimizes the kernel codes for GPU efficiently. The results demonstrate the remarkable performance of GTuner compared with the baselines.

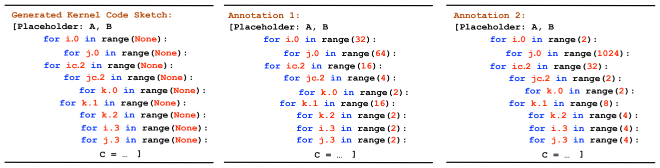
2 Preliminaries

2.1 Computational Graphs

The deep neural network models are usually represented as computational graphs, with layers (operators) as nodes and the edges representing communications in the models and reflecting the topological information. These computational graphs are mapped to the hardware accelerators (*e.g.*, FPGA [12], GPU [13], and ASIC [14]) through some deep learning frameworks and libraries (*e.g.*, cuDNN, oneDNN). The implementation details for the FPGA or ASIC are available for the designers, such as the allocations of the MAC engine, systolic array, cache behavior, and computation patterns [14, 15]. Therefore, accurate models, even analytical formulations, can be built to measure the performance. In contrast, for GPU, the complexities of the hardware and programming model and the lack of implementation details make this problem challenging.

Existing compilation techniques partition the graphs into small subgraphs. Each subgraph contains several neighboring layers, *e.g.*, softmax, pooling, linear layers, and convolutions [5, 7]. Each subgraph is implemented as a kernel on GPU. Many code templates with some parameters to be determined are designed to implement the computations of a kernel. Some parameters are loop boundaries, splits, cache read steps, inlines, *etc.* The final executable kernel codes are generated according to these parameters. As mentioned above, in Ansor, the code templates and parameters are termed *sketches* and *annotations*, respectively. Figure 1 is an example of one sketch and its two annotations. Sketches for the same subgraph may have distinct structures with various numbers and orders of loops and computations. Implementing a unified encoding method to represent the different sketches and annotations is difficult. Therefore, Ansor extracts statistical features for each code annotation via static code analyses, *e.g.*, number of touched cache lines, touched memory in bytes, number of floating multiplication operations, sizes of allocated buffer in bytes, *etc.* Some important concepts are clarified here:

- **Computational graph and subgraph**: the whole DNN model is represented as a computational graph, and the graph is split into some subgraphs by the DNN compilers.



```
Generated Kernel Code Sketch:
[Placeholder: A, B
      for i.0 in range(None):
          for j.0 in range(None):
      for ic.2 in range(None):
          for jc.2 in range(None):
             for k.0 in range(None):
         for k.1 in range(None):
             for k.2 in range(None):
             for i.3 in range(None):
             for j.3 in range(None):
                 C = … ]
```

```
Annotation 1:
[Placeholder: A, B
       for i.0 in range(32):
           for j.0 in range(64):
       for ic.2 in range(16):
           for jc.2 in range(4):
              for k.0 in range(2):
          for k.1 in range(16):
              for k.2 in range(2):
              for i.3 in range(2):
              for j.3 in range(2):
                  C = … ]
```

```
Annotation 2:
[Placeholder: A, B
       for i.0 in range(2):
          for j.0 in range(1024):
       for ic.2 in range(32):
          for jc.2 in range(2):
             for k.0 in range(2):
          for k.1 in range(8):
              for k.2 in range(4):
              for i.3 in range(4):
              for j.3 in range(4):
                  C = … ]
```

Figure 1: A sketch and two annotations of this sketch.

- **Sketch**: the code templates designed by compilers to implement the computations of a subgraph are termed *sketches*. Each subgraph has many templates, *i.e.*, many sketches.
- **Annotation**: the combinations of the parameters to be determined in each sketch are termed *annotations*, *i.e.*, each sketch has many annotations, as shown in Figure 1.
- **Structural feature**: each node in a subgraph has some structural features. The nodes' features are aggregated as the structural features of the subgraph.
- **Statistical feature**: each annotation of the subgraph has a statistical feature vector reported by Ansor via static program analyses.

2.2 Graph Neural Networks and Attention Mechanism

Graph neural networks (GNNs) have been widely used in modeling graph data, achieving impressive results in the prediction, regression, and classification tasks of nodes and graphs [16, 17]. GNNs follow a recursive neighborhood aggregation scheme, where each node aggregates feature vectors from its neighbors. After several iterations of aggregations, each node is represented by a new feature vector. The new feature vectors capture the structural information within the neighborhood [17–19]. The feature representation of the whole graph (*aka.*, graph embedding) is obtained via graph pooling. Mean pooling is a typical and widely used pooling method.

Transformers [20] have achieved great success and become the de facto choice for many natural language processing (NLP) tasks. Further, inspired by NLP successes, many works [21, 22] unveiled the potential of multi-head attention (MHA), which plays a crucial role in transformers for computer vision tasks. The attention operation has three inputs: query, key, and value vectors. Each query vector can attend to all the key vectors and compute the attention scores with respect to these key vectors. The final output is the weighted sum of the value vectors, with the attention scores as the weights. A significant advantage of MHA is that it can learn the long-range dependencies and complicated relationships between the inputs. The MHA stacks the attention modules to achieve outstanding performance. The readers may refer to [20] for more details on attention mechanisms and transformers.

3 Algorithms

3.1 Overall Flow of GTuner

The flow graph is shown in Figure 2. Graph-level optimizations are conducted on the DNN model, *e.g.*, operator fusion, constant-folding, memory planning, data layout transformation, combinations of dense convolutions and linear operations, and operation canonicalizations. Then the optimized model is split into some subgraphs, *i.e.*, subtasks. GPU codes are optimized for each of these subgraphs independently. The final model deployment strategy is achieved by deploying these subgraphs sequentially [5].

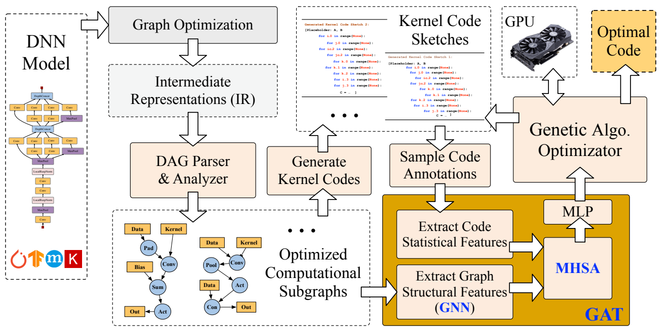


Figure 2: The overall flow of our GTuner.

After the graph optimizations, subgraphs are represented as intermediate representations (IRs) in the compilation tool. We implement a directed acyclic graph (DAG) parser and analyzer (lexical analysis and syntactic analysis) to analyze the IRs and construct the DAG representations for the subgraphs. Nodes in the DAG are computation and data nodes. Structural features are extracted for these nodes, including node types, shapes, operation types, *etc.* The node structural features are passed to a graph neural network to learn a unified embedding for the graph (*i.e.*, graph structural features). This embedding reflects this graph's scheduling and topological information, distinguishes different graphs, and improves the generalization to new tasks with various graphs structures.

Some rule-based kernel code templates (*i.e.*, sketches) are generated for each computational subgraph. The code sketches with parameters (*i.e.*, annotations) comprise the code design space of this subgraph. The statistical features are extracted via Ansor and concatenated with the graph structural features for each sampled code. The concatenated features are then passed to the multi-head self-attention to learn the complicated relationships to predict the inference latency.

The genetic algorithm (GA) is adopted as the optimization method to explore the code design space to find the optimal code. Our GAT model predicts the inference latencies for the codes and guides the exploration of the GA algorithm. In each optimization iteration of the GA algorithm, codes sampled by GA are passed to GAT to predict the performance. Then the codes with the best-predicted performance are compiled and executed on GPU to get the actual performance. The final optimal code is selected according to the actual performance. The number of codes executed on GPU is also termed *measure trials*. For example, 10000 code annotations are sampled by GA and estimated by GAT. GAT selects the best 80 annotations and compiles and deploys them on GPU. Therefore, the measure trial is 80. The final deployment solution is the code with the best actual performance from these 80 codes.

In our framework, the graph level optimization, code generation, and genetic algorithm-based optimization are from TVM [5] and Ansor [7], and the other modules are ours.

Graph Attention Network (GAT) consists of the graph neural network (GNN) module and the multi-head self-attention (MHSA) module, as shown in Figure 3. Structural analysis is conducted on the graph to get the structural features, including the types of operations, dimensions of data, *etc.* These features are the inputs to the graph neural network layers which will be discussed in Section 3.2. The output of the graph neural network is the structural features

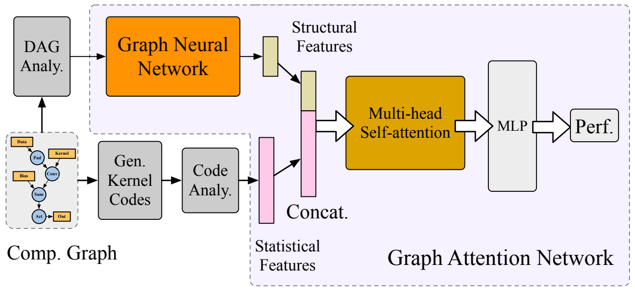


Figure 3: Structure of our graph attention network (GAT).

for the subgraph. The multi-head self-attention module will be discussed in detail in Section 3.3. Then a multi-layer perceptron (MLP) is appended to reduce the dimension and map to the performance.

3.2 GAT: Graph Neural Network Module

DNN models usually have different model structures, *e.g.*, the three structures shown in Figure 4. The distinctive model structures result in different on-chip scheduling, communication, and computation patterns. These differences also help distinguish deployment tasks while their statistical features have no structural information. In other words, it is easy to identify the deployment tasks given structural information while the statistical information is incapable. In this section, the graph neural network is used to learn the representations for the subgraphs. Therefore, this part is independent of the sketches and annotations.

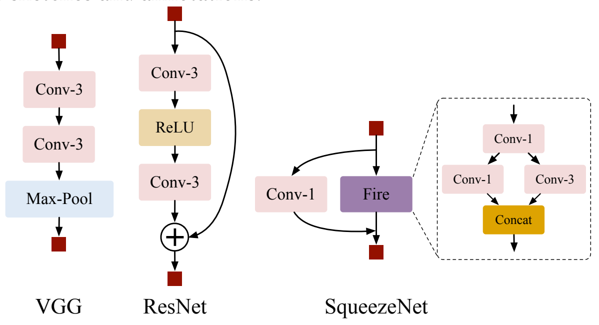


Figure 4: Typical diverse structures in VGG, ResNet (residual block) and SqueezeNet (fire block). Conv-x denotes that the kernel size is x.

Each computational subgraph to be implemented as a kernel is represented as $\mathcal{G}(\mathcal{V}, \mathcal{E})$, with the node set $\mathcal{V}$ and edge set $\mathcal{E}$. $\mathcal{V}$ has n nodes, *i.e.*, $|\mathcal{V}| = n$. The neighborhood set is $\mathcal{N}$, with $\mathcal{N}(v)$ denoting the neighbors of v. The features for the graph are represented as $\boldsymbol{X}$. Each vector $\boldsymbol{x}_i \in \boldsymbol{X}$ with $i \in \{1, \dots, n\}$ denotes the features for node $v_i \in \mathcal{V}$. Denote the feature length as d_x, *i.e.*, $\boldsymbol{x}_i \in \mathbb{R}^{1\times d_x}$. To learn the structural information of the computational subgraph, we update the representations of a node by aggregating information from its neighbors. Let AGGREGATE$(\cdot)$ denotes the aggregation function, and a function COMBINE combines the information from neighbors and the features of the node itself. These functions take the following common forms:

$$\begin{aligned} \boldsymbol{a}_i^k &= \text{AGGREGATE}^{(k)}(\{\boldsymbol{x}_t^{k-1} : v_t \in \mathcal{N}(v_i)\}), \\ \boldsymbol{x}_i^k &= \text{COMBINE}^{(k)}(\boldsymbol{x}_i^{k-1}, \boldsymbol{a}_i^k), \end{aligned} \tag{1}$$

where k represents the k-th iteration of aggregations in the graph neural network. For $k = 0$, $\boldsymbol{x}_i^{k=0}$ is the raw feature of node v_i itself.

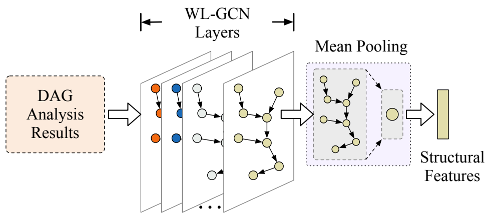


Figure 5: Graph neural network.

Researchers have proposed many aggregation and combination functions for various applications [17–19]. An important target in our problem is to distinguish the different model structures. Therefore, the concept of the Weisfeiler-Leman (WL) test is of vital importance. The WL test is to distinguish the isomorphic graphs via information propagation and can identify the structural similarities between graphs. Further, [23] demonstrated that GNNs could be viewed as an extension of the WL test, which in principle have the same power but are more flexible in their ability to adapt to the learning task at hand and are able to handle complicated node features. In this paper, we choose to use the graph convolutional layer used by [23], denoted as WL-GCN, in which the AGGREGATE and COMBINE steps are integrated. The function of the graph convolutional layer takes the form as follows:

$$x_i^k = W_1^{k-1} x_i^{k-1} + W_2^{k-1} \sum_{v_t \in \mathcal{N}(v_i)} x_t^{k-1}, \tag{2}$$

where W_1^{k-1} and W_2^{k-1} denote the learnable weights. Then, the mean pooling is used to achieve the feature embedding for the computational subgraph, *i.e.*, the feature vectors of the nodes in the graph are summed up and averaged as the feature for the graph:

$$G = \frac{1}{n} \sum_{v_i \in \mathcal{V}} x_i^K, \tag{3}$$

where G denotes the feature embedding for the subgraph, n is the number of nodes, and K is the maximum iteration of information aggregation. The model structure is shown in Figure 5.

3.3 GAT: Multi-head Self-attention Module

The structural features learned by the graph neural network and the statistical features of the annotations should be analyzed to understand the complicated relationships between features, enhance the critical parts and fade out the unimportant parts. Unlike FPGA and TPU, there are no low-level implementation details on GPU. Therefore, determining which features are essential for the tasks should be finished automatically. As discussed above, the statistical features for GPU are weakly related to the computation patterns. In Ansor, the statistical features can be briefly categorized into some groups, *i.e.*, buffer access features, buffer storage features, arithmetic-related features, *etc.* Typical features include the number of touched cache lines, touched memory in bytes, number of floating multiplication operations, number of unrolled iterators, *etc.* They are unstructured data that are directly concatenated, with no information related to the kernel structures. The orders of the features in the vectors have no physical meanings. To build better performance models, it is necessary to learn the implicit relationships between the features while these are the prior knowledge for other types of devices. For simplicity of notations, the statistical and structural features after concatenation are briefly denoted as x.

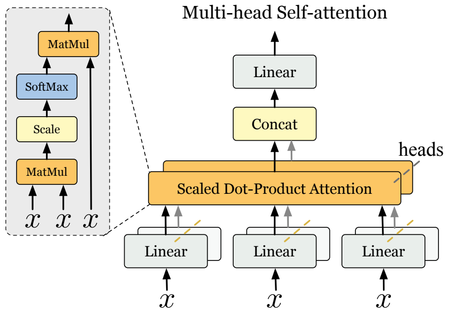


Figure 6: Multi-head self-attention with x as the query (Q), key (K), and value (V) simultaneously.

Inspired by self-attention mechanisms' success in the computer vision field, we propose to use the multi-head self-attention (MHSA) module to learn the features. To formulate MHSA, we first introduce the scaled dot-product attention Attn$(\cdot)$. Given queries $Q \in \mathbb{R}^{n_q \times d_k}$, keys $K \in \mathbb{R}^{n \times d_k}$ and values $V \in \mathbb{R}^{n \times d_v}$, we have [20]:

$$\text{Attn}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V, \tag{4}$$

where n_q, n, d_k, d_v are the query number, key number, key dimension and value dimension, respectively. Then we can define the multi-head attention as:

$$\text{MHA}(Q, K, V) = Concat(H_1, H_2, \cdots, H_h) W^O, \tag{5}$$

where H_i is the output of the i-th attention head, h is the number of heads and W^O is the learnable projection weight matrix. H_i is computed by:

$$H_i = \text{Attn}\left(QW_i^Q, KW_i^K, VW_i^V\right), \tag{6}$$

where $W_i^Q \in \mathbb{R}^{d_k \times d_k}$, $W_i^K \in \mathbb{R}^{d_k \times d_k}$, $W_i^V \in \mathbb{R}^{d_v \times d_v}$ are learnable projection matrices corresponding to the i-th head. Each query vector of Q can attend to all the key vectors of K and compute the attention scores concerning these key vectors. The final output is the weighted sum of the value vectors of V, with attention scores as the weights. The output projection matrix is $W^O \in \mathbb{R}^{hd_v \times d_{out}}$, where d_{out} is the output dimensionality for the multi-head attention function.

To fit the inputs of MHA, we firstly reshape the original vector x into several shorter vectors. Denote the original length of x as l. Then the reshaped x, denoted as x^R, has the shape $h \times \frac{l}{h}$, where h is the number of heads in MHA. The x^R is used as the query (Q), key (K), and value (V) simultaneously. MHA becomes **multi-head self-attention (MHSA)**, which captures global dependencies among the inputs, as shown in Figure 6 and Equation (7).

$$\text{SelfAttn}\left(x^R W_i^Q, x^R W_i^K, x^R W_i^V\right). \tag{7}$$

According to Equation (4), the inner products between vectors in x are computed. These products characterize the similarities between the feature vectors. Further, these similarities reflect the implicit relationships among the features, including the scheduling of the subgraphs, memory patterns, which features are related to each other or have similar influences on the performance, *etc.* The similarity values are then used as the weight scores to sum the values, *i.e.*, vectors in x^R itself. The connections between MHSA and inference latency are learned through model training. And the weights in

MHSA will be updated to enhance the critical parts of the features and fade out the unimportant parts.

4 Experiments

In this section, we conduct experiments on our GTuner to validate the performance. The experimental inference platform is Nvidia GeForce RTX 3090 (Ampere architecture, SM86), with CUDA Driver 11.4, PyTorch 1.10, and TVM 0.8-dev. The CPU is 16 core Intel(R) Xeon(R) Gold 6226R CPU @ 2.90GHz. Some public models defined in TVM Relay library are tested, including ResNet-18, ResNet-34 [24], MobileNet V1 [25], and SqueezeNet V1.1 [26]. The representative DNN layers widely used in both industries and academia are all covered in these models, including conventional convolutional layers, shortcut layers, multi-branch layers, fully connected layers, depth-wise convolutional layers, *etc.* The tuning history of Ansor (with XGBoost as performance model) on Inception-V3 and VGG-11 is collected as the training data to train the models, in total about 170000 annotations. Some baselines are compared in the experiments to prove the effectiveness of our method, including Ansor [7], AutoTVM [8], PyTorch, and PyTorch with JIT optimization [27]. The performance metric is the end-to-end inference latency of the model. We also show the results of Giga floating operations per second (GFLOPS), which reflects the peak computation speed of a task on the device.

4.1 Implementation Details

Our GAT model has two WL-GCN layers [23] to process the graph features, a mean pooling layer, a concatenation layer (to concatenate statistical and graph features), a fully-connected layer (downscale the features to 512), a four-head multi-head self-attention layer (*i.e.*, $h = 4$), and an MLP regression module (with output dimensions: 200-100-20-1). In the benchmarks and Ansor, there are 136 elements representing the types of nodes, inputs, outputs, data shapes, types of operations, *etc.* The graph embedding has 136 elements after graph pooling. The statistical features reported by Ansor have 656 elements and are concatenated with the graph features and downscaled to 512 via the fully-connected layer.

The baseline multi-head self-attention (MHSA) comprises a four-head multi-head self-attention layer (the same as ours) and an MLP regression module (also the same as ours) without graph module and structural features.

We train the models with Adam optimizer for 300 epochs with a learning rate 1e-4 and batch size 512. The loss function is mean square error (MSE). The regression target is inference latency. We denote our method as GTuner (GNN + MHSA) in the result tables for ease of explanation.

Our method's genetic algorithm optimization process and the baselines all follow the default settings of TVM. The different numbers of measure trials are compared in the ablation studies. Note that except for the ablation studies on the measure trials, the measure trials of other results are 80 trials per subgraph. In other words, for a DNN model, the total number of measure trials is the product of the number of subgraphs and 80 trials.

4.2 Ablation Studies on GAT Structure

We compare our method with some widely-used graph convolutional layers, including spectral graph convolution (SpecGCN) [28], masked attention convolution (MaskGAT) [29], and GraphSAGE [30]. SpecGCN propagates information in the graph via a first-order approximation of localized spectral filters on the graphs. Learned filters are used to represent the nodes in the Fourier domain. MaskGAT introduces the attention-based architecture to compute the hidden representations of the nodes by using masks during information aggregation. GraphSAGE learns a function that generates embeddings by sampling and aggregating features from a node's local neighborhood to improve the generalization abilities to unseen nodes.

Table 1: Comparisons between Convolutional Layers

ResNet-18	Ansor	GTuner (WL-GCN)	SpecGCN	MaskGAT	GraphSAGE
Latency (ms)	1.073	0.923	1.016	1.105	1.168

Table 2: Performance without GNN or MHSA

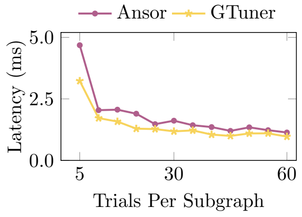
ResNet-18	GTuner (GNN + MHSA)	MHSA	GNN + MLP
Latency (ms)	0.923	0.963	1.121

We replace the WL-GCN layers in GAT with these three convolutional layers. Under the same training and experimental settings, the inference latencies of ResNet-18 are listed in Table 1. The results show that these methods have poor performance in this problem. As mentioned before, graph isomorphism is important in our context. These methods fail to learn this kind of isomorphism and fall into the trap of overfitting to the training set and therefore are incapable of tackling new tasks and the isomorphic graphs with different volumes of computations and communications. SpecGCN outperforms Ansor ($1.016 < 1.073$, *i.e.*, 5.31%). MaskGAT and GraphSAGE force the model to ignore some information to improve the generalization. It degrades the performance significantly since the important topological and scheduling knowledge is discarded. These techniques achieve good results on big data problems, *e.g.*, social networks and recommendation systems with millions of nodes and edges. In our problem, the structures of graphs are limited compared with these applications, and it is unacceptable to forget information.

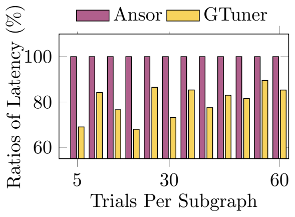
Further, to validate the performance of our proposed MHSA module, we do the ablation study via dropping the MHSA module in GAT. The outputs of the GNN module and the statistical features are concatenated and then directly passed to a fully-connected layer to achieve a feature vector with length 512. Then this feature vector is passed to the same MLP regression module. This experiment is denoted as GNN + MLP in Table 2. For comparison, the baseline MHSA is also listed. Results show that the combination of our GNN and MHSA achieves the best performance.

4.3 Ablation Studies on GPU Measure Trials

The genetic algorithm implemented by TVM is adopted as the default searching algorithm in Ansor and our method. Interacting with GPU helps validate the designs found by the searching algorithms. Therefore, more interactions (measure trials) will find better solutions. Some experiments are conducted on ResNet-18 to reveal the performance of our method under different measure trials, as shown in Figure 7. The numbers of measure trials are 5, 10, 15, . . . , 60. The ratios of latencies with respect to Ansor are shown in Figure 7(b). With the increases in measure trials, both Ansor and our GTuner can find better results, while our performance advantages compared with Ansor are still remarkable (more than 10%).

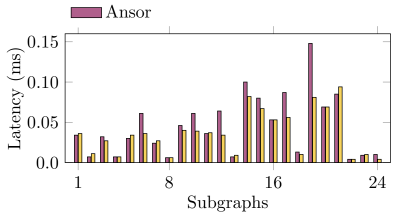


(a)

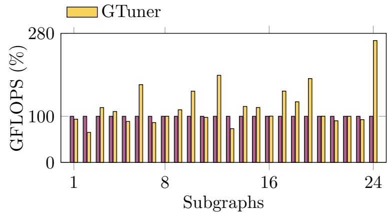


(b)

Figure 7: Results of different measure trials.



(a)



(b)

Figure 8: Detail results of subgraphs in ResNet-18.

Table 3: End-to-end Model Inference Latency (ms)

Model	PyTorch	PyTorch-JIT	AutoTVM	Ansor	MHSA	**GTuner** +
ResNet-18	27.180	4.119	1.056	1.073	0.963	**0.923 (13.98%)**
ResNet-34	48.988	5.929	1.180	0.968	0.907	**0.872 (9.92%)**
SqueezeNet	16.658	3.648	0.311	0.207	0.201	**0.197 (4.83%)**
MobileNet	30.324	6.972	0.513	0.242	0.252	**0.227 (6.20%)**

+ Ratios are performance improvements compared with Ansor.

Table 4: Time Costs (minutes) of the Optimization Processes

Model	Ansor	MHSA	GTuner	AutoTVM
ResNet-18	45.57	45.95	46.94	65.22
ResNet-34	46.66	48.89	50.71	54.86
SqueezeNet	43.53	44.40	45.91	63.90
MobileNet	42.88	43.80	44.20	61.60

4.4 Performance of the Whole framework

Detailed results of subgraphs in ResNet-18 are shown in Figure 8, including the inference latencies and GFLOPS. The GFLOPS values are represented as ratios to Ansor. It is shown that our method reduces the latencies and improves the GFLOPS on most subgraphs. The performance improvements on the difficult subgraphs (with long latencies) are inspiring.

The end-to-end inference latencies of the DNN models are listed in Table 3. The results demonstrate the significant advantages of our method compared with the baselines. The proposed multi-head self-attention module improves the performance based on Ansor. The graph network module in our GTuner further improves the performance. On the four models in Table 3, our method reduces the latencies by **13.98%**, **9.92%**, **4.83%**, and **6.20%**, respectively, compared with Ansor, and more than **30%** on average compared with AutoTVM. The two PyTorch-based methods are rule-based, and their performance is not satisfying. The results also reveal that using the graph structural features helps improve the generalization of the performance model to new tasks that do not exist in the training set. The time costs of the whole optimization processes of TVM-based methods are listed in Table 4, including the searching processes of the genetic algorithm, the compilations of codes, and the interactions with GPU. It is shown that our performance improvements have no extra overheads on the time costs.

5 Conclusion

This paper proposes a novel method GTuner to optimize the computations of DNN models on GPU. The computational graphs are learned via the WL graph isomorphism convolutional layers to extract high-quality features. The structural and statistical features are jointly learned via the multi-head self-attention module to improve the regression quality. Our results outperform the baselines and prove the effectiveness of GTuner.

Acknowledgment

This work is partially supported by SmartMore and ITF Partnership Research Programme (No. PRP/65/20FX).

References

[1] S. Han *et al.*, "Deep Compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," in *Proc. ICLR*, 2016.
[2] T. Chen *et al.*, "An efficient sharing grouped convolution via Bayesian learning," *IEEE TNNLS*, 2021.
[3] X. Zhang *et al.*, "Exploring HW/SW co-design for video analysis on CPU-FPGA heterogeneous systems," *IEEE TCAD*, 2021.
[4] T.-M. Li *et al.*, "Differentiable programming for image processing and deep learning in Halide," *ACM SIGGRAPH*, 2018.
[5] T. Chen *et al.*, "TVM: An automated end-to-end optimizing compiler for deep learning," in *Proc. OSDI*, 2018.
[6] B. H. Ahn *et al.*, "CHAMELEON: Adaptive code optimization for expedited deep neural network compilation," in *Proc. ICLR*, 2020.
[7] L. Zheng *et al.*, "Ansor: Generating high-performance tensor programs for deep learning," in *Proc. OSDI*, 2020.
[8] T. Chen *et al.*, "Learning to optimize tensor programs," in *Proc. NeurIPS*, 2018.
[9] Q. Sun *et al.*, "Deep neural network hardware deployment optimization via advanced active learning," in *Proc. DATE*, 2021.
[10] J. Mu *et al.*, "A history-based auto-tuning framework for fast and high-performance DNN design on GPU," in *Proc. DAC*, 2020.
[11] Q. Sun *et al.*, "Fast and efficient DNN deployment via deep Gaussian transfer learning," in *Proc. ICCV*, 2021.
[12] C. Hao *et al.*, "FPGA/DNN co-design: An efficient design methodology for IoT intelligence on the edge," in *Proc. DAC*, 2019.
[13] Z. Song *et al.*, "GPNPU: enabling efficient hardware-based direct convolution with multi-precision support in GPU tensor cores," in *Proc. DAC*, 2020.
[14] Y.-H. Chen *et al.*, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE JETCAS*, 2019.
[15] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proc. ISCA*, 2017.
[16] Z. Wu *et al.*, "A comprehensive survey on graph neural networks," *IEEE TNNLS*, 2020.
[17] K. Xu *et al.*, "How powerful are graph neural networks?" in *Proc. ICLR*, 2019.
[18] J. Lee *et al.*, "Self-attention graph pooling," in *Proc. ICML*, 2019.
[19] J. Atwood *et al.*, "Diffusion-convolutional neural networks," in *Proc. NeurIPS*, 2016.
[20] A. Vaswani *et al.*, "Attention is all you need," in *Proc. NeurIPS*, 2017.
[21] A. Dosovitskiy *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," in *Proc. ICLR*, 2021.
[22] N. Carion *et al.*, "End-to-end object detection with transformers," in *Proc. ECCV*, 2020.
[23] C. Morris *et al.*, "Weisfeiler and Leman go neural: Higher-order graph neural networks," in *Proc. AAAI*, 2019.
[24] K. He *et al.*, "Deep residual learning for image recognition," in *Proc. CVPR*, 2016.
[25] A. G. Howard *et al.*, "MobileNets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.
[26] F. N. Iandola *et al.*, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and < 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.
[27] A. Paszke *et al.*, "Automatic differentiation in PyTorch," in *NIPS Workshop*, 2017.
[28] T. N. Kipf *et al.*, "Semi-supervised classification with graph convolutional networks," in *Proc. ICLR*, 2017.
[29] P. Veličković *et al.*, "Graph attention networks," in *Proc. ICLR*, 2018.
[30] W. Hamilton *et al.*, "Inductive representation learning on large graphs," in *Proc. NeurIPS*, 2017.