Expedited Tensor Program Compilation Based on LightGBM

Gonghan Liu1*, Yue Li1 and Xiaoling Wang1
1 Beijing Aerospace Automatic Control Research Institute, Beijing, 100854, China
liugh2@12.calt.casc *Corresponding author’s e-mail: lghan_97@163.com

Abstract. If the traditional deep learning framework needs to support a new operator, it usually needs to be highly optimized by experts or hardware vendors to be usable in practice, which is inefficient. The deep learning compiler has proved to be an effective solution to this problem, but it still suffers from unbearably long overall optimization time. In this paper, aiming at the XGBoost cost model in Ansor, we train a cost model based on LightGBM algorithm, which accelerates the optimization time without compromising the accuracy. Experimentation with real hardware shows that our algorithm provides 1.8× speed up in optimization over XGBoost, while also improving inference time of the deep networks by 6.1 %.

1. Introduction
Deep Neural Network (DNN) can be used for image recognition, natural language processing, and beating humans in partly challenging strategy games. In the process of its technological development, the computing power of computer hardware plays an important role. Many of the most popular deep learning (DL) frameworks currently support acceleration on a limited number of server-level GPU devices, and this support is dependent on vendor-specific GPU libraries. In particular, popular DL frameworks such as TensorFlow[1] and PyTorch[2] adopt Nvidia cuDNN[3] or Intel MKL-DNN[4] as back-end. However, given the increasing complexity of tensor operations in DNNs and the volatility of DL algorithms, it leads to a huge workload of operator support for deep learning framework.

In order to imitate or even surpass the performance of manual optimization libraries, some exports have carried out some researches in the field of deep learning compilers and achieved certain results, such as XLA[5], Glow[6] and TVM[7]. Ansor[8] is a tensor generation framework that is integrated into TVM to generate high-performance tensor programs for deep learning applications. The programmer only needs to provide the DNN model, Ansor will automatically search for a set of scheduling policies, and then through the TVM code generation, the optimized code can be obtained. Although Ansor can generate highly optimized program, the process will take a long time to compile. As new neural architectures[9] come out in rapid speed, and with deeper or wider networks[10] on various hardware platforms, we are forced to optimize the networks more frequently. The excessive long optimization time hinders the turnaround time and even puts the practical utility of the current compiler-based solutions into question.

Through research, we find that most current DL compilers that can automatically generate high-performance code use a cost model to predict the time of each program on real hardware, and then select the optimal solution through a search algorithm. For example, Ansor, uses XGBoost[11] as a cost model to predict, so the speed at which XGBoost predicts can have a significant impact on the overall optimization time. We present a new cost model called LightGBM[12] to replace the original XGBoost model. This is an approach designed to achieve similar or better optimization quality but with a shorter optimization time. Extensive experiments show that the proposed approach is superior, allowing us to optimize the DNN model to achieve 1.8× speedup, with similar or even better inference times.
2. Background and Related Work

2.1 DL Compiler

DL compilers like TVM\cite{7} have recently become popular for optimizing DL programs\cite{6}. Like traditional compilers, a typical DL compiler abstracts the DNN model as a variety of programming languages, optimizes it, and produces high-performance code that can be executed at the back end, as shown in Figure 1. The structure of the DL compiler consists mainly of two parts: the compiler front end and the compiler back end. The intermediate representation (IR) traverses the frontend and the backend. IR is an abstraction of the program for the optimization of the program. The DNN model is converted to multilevel IR in the DL compiler, which is divided into high-level IR and low-level IR. Based on high-level IR, the compiler frontend is responsible for hardware-independent transformation and optimization. Based low-level IR, the compiler backend is responsible for hardware-specific optimization, code generation, and compilation.

![Figure 1. DL compilation Pipeline](image)

2.2 Ansor

As shown in Figure 2, code generation for TVM\cite{7} requires the following steps. First, TVM takes the computational graph of the model as input, converts it into its domain specific language (DSL). The DSL in TVM is called Tensor Expressions (the middle yellow region). Programmers need to specify scheduling policies manually or semi-manually (the middle gray region), and then generate code through TVM pipeline. Finally, get a series of nested for loops (at the right of the Figure 2). However, optimization by hand or using templates may fall into local optimality, and programmers need to have a deep understanding of the hardware to write a good scheduling strategy. In order to solve the above problems, TVM used Ansor\cite{8}, so that programmers only need to provide the DNN model. Ansor will automatically select a set of scheduling policies, and then through TVM code generation, the optimized code can be obtained, as shown in Figure 3.
Figure 2. TVM Pipeline

Ansor pipeline is divided into the following steps, as shown in Figure 4:

1) **Schedule Task**: The complete calculation graph is divided into several subgraphs, and the potential subgraphs are optimized.

2) **Sketch**: The high-level features of the operator are extracted, and the coarse-grained optimization of the operator is carried out to determine the basic structure of the code.

3) **Annotation**: Randomly initialize tiling size and some loop strategies to obtain a complete representation of the computational graph.

4) **Evolutionary**: Training cost model evaluates the performance of the code according to the cost model, selects a group of implementations with high scores in the evaluation to obtain its ground truth and actual performance, and selects the implementation with optimal actual performance as the output of Ansor.

In summary, Schedule Task is used to divide the computational subgraphs of DNN model, determine the optimization times of each subgraph. Sketch is used to build the overall framework of the code. Annotation is used to fill in the details, and Evaluation training a model to evaluate and update the generated code, then get the final code generation result.

Figure 3. Ansor in TVM

Figure 4. Ansor Pipeline

### 2.3 LightGBM

XGBoost\[11\] is an efficient implementation of GBDT\[13\]. Its basic algorithm idea is, firstly, pre-sort all features according to the value of features, secondly, find the best segmentation point on features with the cost of \(o(#\text{data})\) when traversal segmentation points, and finally, divide the data into left and right nodes after finding the segmentation points of features. Because XGBoost traverses all the
features each time it builds a node in a tree, and searches all possible split values for each feature to find the best feature with a split point, there is a significant overhead in time. LightGBM\[12\] solves the above-mentioned time-consuming problem from two aspects. Firstly, LightGBM\[12\] adopts GOSS (Gradient-based One-Side Sampling) algorithm to reduce training samples, thus reducing the time of traversal of samples during training. Secondly, EFB (Exclusive Feature Bundling) algorithm is adopted to reduce the features involved in calculation, thus reducing the time of searching for the optimal feature with the optimal segmentation point.

2.3.1 GOSS. The GOSS algorithm preserves the samples with large gradients and carries out random sampling on the samples with small gradients. Suppose that we have a training set \( \{x_1, x_2, \cdots, x_n\} \) with \( n \) groups of data with characteristic dimension \( s \). In each iteration of gradient boosting, the negative gradients of the loss function with respect to the output of model are denoted as \( \{g_1, g_2, \cdots, g_n\} \). The decision tree model splits each node at the most informative feature (with the largest information gain). Let \( P \) be the training dataset on a fixed node of the decision tree. The variance gain of splitting feature \( j \) at point \( d \) for this node is defined as:

\[
V_{j,p}(d) = \frac{1}{n_p} \left( \frac{\left( \sum_{x_i \in P : x_i \leq d} g_i \right)^2}{n_{l,p}^j(d)} + \frac{\left( \sum_{x_i \in P : x_i > d} g_i \right)^2}{n_{r,p}^j(d)} \right)
\]

Where \( n_p = \sum I[x_i \in P] \), \( n_{l,p}^j(d) = \sum I[x_i \in P : x_i \leq d] \), \( n_{r,p}^j(d) = \sum I[x_i \in P : x_i > d] \). \( n_p \) represents the number of training set samples of a fixed leaf node, \( n_{l,p}^j(d) \) is the number of samples with a value less than or equal to \( d \) on the \( j \) feature and \( n_{r,p}^j(d) \) is the number of samples greater than \( d \) at the \( j \) eigenvalue. For feature \( j \), the decision tree algorithm selects \( d^* = \text{argmax}_d V_j(d) \) and calculates the largest gain \( V_j(d^*_j) \). Then, the data are split according feature \( j^* \) at point \( d^*_j \) into the left and right child nodes. The overall process of GOSS algorithm is as follows:

1) All the values of the features to be split are sorted in descending order according to the absolute value, and the top \( a \times 100\% \) is selected as the large gradient sample.

2) It randomly samples \( b \times 100\% \) instances from the rest of the data.

3) Amplify the sampled data with small gradients by a constant \( \frac{1-a}{b} \) when calculating the information gain.

2.3.2 EFB. High-dimensional data is generally sparse, so you can take advantage of this feature to reduce the dimension of the feature. A metrics is used to measure the degree of non-mutual exclusion between features, which is called the “conflict ratio”. When this value is small, two features that are not completely mutually exclusive can be bound without affecting the final accuracy. LightGBM simplifies the graph coloring problem of this problem and uses greedy algorithm to generate reasonable binding, which effectively avoids unnecessary feature calculation and reduces the algorithm complexity from \( o(#\text{data}) \) to \( o(#\text{non_zero_data}) \). The algorithm flow is as follows:

1) All features are regarded as vertices of the graph. For features that are not independent of each other, edges are connected. The weight of edges is the total conflict value of two connected features.
2) Sort the features in descending order according to the degree of nodes

3) For each feature, traverse its existing feature clusters. If it is found that the number of contradictions that the feature adds to the feature cluster does not exceed a certain threshold, the feature is added to the cluster. If the feature cannot be added to any existing cluster, create a new cluster and add the feature.

3. Design of Cost Model

3.1 Data set
As described in Section 2.2, Annotation is used to randomly generate code without considering the performance of the generated code. Therefore, the computation to be optimized may be very complex, and it is unrealistic to traverse all the annotations to obtain the optimal implementation. In order to predict the performance of the code generated by the Annotation through the cost model, we build the feature vector for an innermost non-loop statement by extracting features in the context of a full program. The extracted features include arithmetic features and memory access features, which can be obtained by traversal of the TVM’s TIR. We use one-hot encoding to encode category features. The length of a feature vector including all the listed features for one statement is 164. We use the same set of features for both CPU and GPU. These feature vectors are taken as the input data of the cost model, and part of features extracted are shown in Table 1.

| parameters       | Subclass                   | Feature set                                                                 |
|------------------|----------------------------|----------------------------------------------------------------------------|
| computing class  | total number of operations | float_mad, float_addsub, float_mul, float_divmod, float_cmp, float_match_func, float_other_func, int_mad, int_addsub, int_float_mul, int_divmod, int_cmp, int_match_func, int_other_func, ... |
| keyword          | vec_num, vec_prod, vec_len, vec_type | unroll_num, unroll_prod, unroll_len, unroll_type |
|                  | parallel_num, parallel_prod, parallel_len, parallel_type | ... |
| thread block     | blockidx_x_len, blockidx_y_len, blockidx_z_len | threadidx_x_len, threadidx_y_len, threadidx_z_len |
|                  | threadIdx_x_len, threadIdx_y_len, threadIdx_z_len | threadIdx_len, ... |
| buffer access    | access type                | Acc_type, Bytes, unique_bytes Lines, unique_lines Stride, ... |
| buffer reuse     | reuse_type                 | reuse_dis_iter, reuse_dis_bytes reusedt, ... |
| Accessed bytes   | bytes_d_reuse ct, unique_bytes_reuse ct lines_d_reuse ct unique_lines_d_reuse ct ... |
| computational    | intensity                  | arith_intensity Curve |
| memory application | Space size               | alloc_size, alloc_outer_prod, alloc_inner_prod, alloc_prod |
| loop features    | features of the “for” loop | outer_prod, num_loops, float_auto_unroll_max_step |
3.2. Model Design

The overall design of the model is shown in Figure 5. First of all, the model extracts the feature $\tau$ from the target program $P$ through the Annotation module, and then obtains the ground truth $D_\tau$ of the corresponding sample through the Runtime, forms the initial training set as the input, carries on the specified times of iterative training to improve the prediction accuracy of the model. For the new feature $\tau^*$, the output is predicted to be $D_{\tau^*}$. $\tau^*$ is sent to real hardware for runtime measurements, resulting in real values $D_{\tau^*}$ to update the cost model to enhance exploration of subsequent iterations.

After the specified number of iterations, select $P(\tau)$ with the shortest time as the output of this layer. When optimizing a DNN network, there are typically fewer than 30,000 features for measurement, and training a LightGBM model on such a data set is very fast, so we train a new model every time instead of doing incremental updates.

![Figure 5. Overall design of LightGBM Tree](image)

LightGBM algorithm uses the leaf-wise strategy to grow leaves with depth limitation. Compared with the depth growth algorithm, the leaf-wise strategy has a faster convergence rate. However, if the model parameters are not set properly, the phenomenon of over-fitting of leaf direction growth may occur, leading to reduced prediction accuracy. Some important parameters of LightGBM are shown in Table 2. There are two ways to adjust the super parameters of the model. One is to fine-tune the model according to experience, and the other is to select parameters of different sizes and substitute them into the model to select the parameters with the best performance. A method of fine-tuning is to manually modulate the hyperparameters until a good combination of hyperparameters is found, but this process is lengthy and it is impossible to explore multiple parameter combinations. Another method uses Scikit-Learn's GridSearchCV function to achieve parameter search. In this paper, the latter is adopted for parameter adjustment. Within the specified parameter range, parameters are adjusted in turn according to the step size, all possible parameter combinations are traverses, and the super parameter combination with the smallest error in the verification set is selected and returned. The final result is shown in Table 3.

| parameters       | meaning                                  | use                                           |
|------------------|------------------------------------------|----------------------------------------------|
| max_depth        | maximum depth of the tree                | when the model is over-fitting, reduce this parameter |
| feature_fraction | feature ratio used in each iteration     | speed up training and reduce overfitting     |
| bagging_fraction | the ratio of data used in each iteration | speed up training and reduce overfitting     |
| num_leaves       | control model complexity                 | too large will lead to overfitting           |

Table 2. Typical parameters in LightGBM
min_data_in_leaf: control model complexity
avoid trees that grow too deep
Lambda_l1/lambda_l2: regularization parameters
balance overfitting and underfitting

Table 3. Parameter optimization results

| parameters               | range                | Final value |
|--------------------------|----------------------|-------------|
| max_depth                | [4,10], stride=2     | 6           |
| feature_fraction         | [0.4,0.6], stride=0.01 | 0.41       |
| bagging_fraction         | [0.4,0.6], stride=0.01 | 0.48       |
| num_leaves               | [5,100], stride=5    | 10          |
| min_data_in_leaf         | [5,100], stride=5    | 10          |
| Lambda_l1/lambda_l2      | [0,1], stride=0.001  | 0.001/1.0   |

The Evolutionary Search module of Ansor makes the optimization converge faster and reduces the traversal time. The overall flow of the Evolutionary Search module after adopting LightGBM is shown in Figure 6. It can be seen that the input of feature extraction comes from three sources: 1) random Annotation samples, ensuring the richness of the samples. 2) The excellent Annotation samples obtained in the previous iteration are helpful for the training of LightGBM model with higher accuracy. 3) Variations of excellent samples. According to the evolutionary algorithm, the "mutation" of good samples is often easier to get good samples, so the sample set is expanded by fine-tuning the parameters of good samples. The extracted features will be used to train LightGBM model. Meanwhile, this model can conduct a preliminary screening of the samples generated by annotations, and obtain the ground truth of the samples with high scores through TVM Runtime. Based on the ground truth, codes with better performance can be selected. That's the entire ANSOR output, and that's the end of the algorithm. This output can be seamlessly converted into schedule, saving the trouble of handwritten schedule, and the code can be generated directly through TVM's CodeGen pipeline.

![Figure 6. Overall design of Evolutionary Search](image)

---

**min_data_in_leaf**
- control model complexity
- avoid trees that grow too deep

**Lambda_l1/lambda_l2**
- regularization parameters
- balance overfitting and underfitting

| Parameters     | Range               | Final Value |
|----------------|---------------------|-------------|
| max_depth      | [4,10], stride=2    | 6           |
| feature_fraction | [0.4,0.6], stride=0.01 | 0.41       |
| bagging_fraction | [0.4,0.6], stride=0.01 | 0.48       |
| num_leaves     | [5,100], stride=5   | 10          |
| min_data_in_leaf | [5,100], stride=5    | 10          |
| Lambda_l1/lambda_l2 | [0,1], stride=0.001  | 0.001/1.0   |

The Evolutionary Search module of Ansor makes the optimization converge faster and reduces the traversal time. The overall flow of the Evolutionary Search module after adopting LightGBM is shown in Figure 6. It can be seen that the input of feature extraction comes from three sources: 1) random Annotation samples, ensuring the richness of the samples. 2) The excellent Annotation samples obtained in the previous iteration are helpful for the training of LightGBM model with higher accuracy. 3) Variations of excellent samples. According to the evolutionary algorithm, the "mutation" of good samples is often easier to get good samples, so the sample set is expanded by fine-tuning the parameters of good samples. The extracted features will be used to train LightGBM model. Meanwhile, this model can conduct a preliminary screening of the samples generated by annotations, and obtain the ground truth of the samples with high scores through TVM Runtime. Based on the ground truth, codes with better performance can be selected. That's the entire ANSOR output, and that's the end of the algorithm. This output can be seamlessly converted into schedule, saving the trouble of handwritten schedule, and the code can be generated directly through TVM’s CodeGen pipeline.

![Figure 6. Overall design of Evolutionary Search](image)
4. Evaluation

In this section, we evaluate LightGBM model experimentally, seeking answers to how LightGBM model helps accelerate the optimization process. We implement LightGBM model in Python, and we leverage scikit-learn\cite{14} to implement the optimization. We evaluated from two aspects: optimization time and inference time. In terms of optimization time, we compared the time taken by Ansor with XGBoost Model and LightGBM model, respectively. In terms of inference time, we used the time taken by the PyTorch as a baseline for comparison. In the following contents, we abbreviate the algorithm utilizing the XGBoost model as xgb, and the algorithm utilizing the LightGBM model as lgb.

4.1 Optimization Time

Table 4 shows the optimization time of Ansor under different cost models. We compare the end-to-end optimization time of xgb and lgb on ResNet-18\cite{15}, VGG16\cite{16} and SqueezeNetv1.1\cite{17} on GPU. In Table 4, “num_trials” is the number of measurement trials we can use during the tuning. Appropriate increase of this value is conducive to the full convergence of search. As shown in Table 4, overall, lgb achieves a 1.8× speedup. This is because the LightGBM model reduces the number of features, filters out samples with smaller gradients, and converts the traversal samples into the form of traversal histograms. It greatly reduces the time complexity and reduces a lot of unnecessary calculation. In addition, with the increase of ”num_trials “, the speed increases more obviously. This is because we need to train a new tree model for each iteration. With the increase of the number of iterations and the increase of the data, LightGBM will have a greater advantage in the prediction.

| Network          | num_trials | xgb         | lgb         | speedup |
|------------------|------------|-------------|-------------|---------|
| SqueezeNetv1.1   | 20,000     | 34608.55 (s)| 27910.12 (s)| 1.24×   |
| VGG16            | 25,000     | 37743.53 (s)| 32813.11 (s)| 1.15×   |
| ResNet18         | 30,000     | 48804.33 (s)| 27113.52 (s)| 1.80×   |

4.2 Inference Time

Figure 7 compares the final inference time from ResNet-18, VGG16 and SqueezeNetv1.1 on GPU. We use PyTorch, the most popular deep learning framework today, as a baseline for our measurement. Overall, lgb achieves up to 6.1% faster inference speed over xgb, and up to 90.1% faster speed over PyTorch. On the one hand, LightGBM algorithm uses the leaf-wise strategy to grow leaves. Therefore, compared with level-wise strategy used in XGBoost, in the case of same number of splitting times, leaf-wise can reduce more errors and obtain better accuracy. On the other hand, lgb and xgb achieve much higher speedup on SqueezeNet, presumably because the heuristic-base optimizations in PyTorch are suboptimal.
Figure 7. Inference time comparison

5. Conclusion
Although highly optimized code can be achieved through existing DL compilers, an obvious drawback is their long code optimization time, required to generate many versions of a tensor program and to profile these versions on hardware. In this paper, we propose to integrate the LightGBM model into Ansor. This algorithm allows cut optimization time cost without losing much accuracy. As a result, LightGBM model in Ansor achieves higher speedups in terms of finding a good tensor program on different DNN models, outperforming XGBoost in Ansor, a state-of-the-art approach.

References
[1] Abadi M, Barham P, Chen J, et al. (2016) Tensorflow: A system for large-scale machine learning[C]. proceedings of the In Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI ’16: 265-283.
[2] Paszke A, Gross S, Massa F, et al. (2019) Pytorch: An imperative style, high-performance deep learning library. In Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems, 32: 8024–8035
[3] Chetlur S, Woolley C, Vandermersch P, et al. (2014) cudnn: Efficient primitives for deep learning. arXiv preprint arXiv:1410.0759,
[4] Intel(R) Math Kernel Library for Deep Neural Networks. https://github.com/01org/mkl-dnn.
[5] The Accelerated Linear Algebra Compiler Framework. https://www.tensorflow.org.
[6] Rotem N, Fix J, Abdulrasool S, et al. (2018) Glow: Graph lowering compiler techniques for neural networks. arXiv:cs.PL/1805.00907
[7] Chen T, Moreau T, Jiang Z, et al. (2018) TVM: An automated end-to-end optimizing compiler for deep learning. In Advances in Neural Information Processing Systems, 31: 3389–3400.
[8] Zheng L, Jia C, Sun M, et al. (2020) Ansor: Generating high-performance tensor programs for deep learning. proceedings of the 14th Symposium on Operating Systems Design and Implementation.
[9] Gao S, Cheng M-M, Zhao K, et al. (2019) Res2net: A new multi-scale backbone architecture. CoRR, abs/1904.01169
[10] Wu Z, Shen C, Van Den Hengel A J P R. (2019) Wider or deeper: Revisiting the resnet model for visual recognition. Pattern Recognit,90: 119-133.
[11] Chen T, Guestrin C. (2016) Xgboost: A scalable tree boosting system[C]. proceedings of the Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining, 22:785-794.

[12] Ke G, Meng Q, Finley T, et al. (2017) Lightgbm: A highly efficient gradient boosting decision tree, 30: 3146-3154.

[13] Friedman J H J A o s. (2001) Greedy function approximation: a gradient boosting machine: 1189-1232.

[14] Pedregosa F, Varoquaux G, Gramfort A, et al. (2011) Scikit-learn: Machine learning in Python, 12: 2825-2830.

[15] He K, Zhang X, Ren S, et al. (2016) Deep residual learning for image recognition. proceedings of the Proceedings of the IEEE conference on computer vision and pattern recognition: 770-778.

[16] Simonyan K, Zisserman A J a p a. (2015) Very deep convolutional networks for large-scale image recognition. In 3rd International Conference on Learning Representations,

[17] Iandola F N, Han S, Moskewicz M W, et al. (2016) SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. arXiv preprint arXiv:1602.07360