Optimizing Neural Network Compilation via Adaptive Workflow with AutoTVM

Yu-Hsiang Chen[1], Tay-Jyi Lin[2,3], Juin-Ming Lu[3], Tien-Fu Chen[1,3*]

[1]Department of Computer Science, National Yang-Ming Chiao-Tung University, Hsinchu, Taiwan
[2]Department of Computer Science and Information Engineering, National Chung Cheng University, Chia-Yi, Taiwan
[3]Electronic and Optoelectronic System Research Laboratories, Industrial Technology Research Institute, Hsinchu, Taiwan
Email: tjlin@cs.ccu.edu.tw (T.J.L.); jm@itri.org.tw (J.-M.L.); tfchen@cs.nycu.edu.tw (T.-F.C.)
*Corresponding author

Manuscript received January 4, 2024; revised January 20, 2024; accepted February 13, 2024; published September 20, 2024

***Abstract*—With the development of deep neural networks, network compilation plays as an important role for achieving faster execution time. Black-box optimization aims to find an optimal solution by searching the design space. However, it suffers from countless costly hardware measurements, which greatly increase the compilation time. This paper aims to reduce the compilation time by reducing hardware measurements. Our solution includes adaptive early-stop, a self-tuning module that controls tuning workflow according to real-time measurements, a K-means cluster sampling module, a history database that records and organizes measurement results for later usages, a decoupled online tuning service. We extend the work leveraging multiple users of online services, including shared history data and module hyperparameter suggestions. Experiments show our proposed approach achieves 52.39% reduction in hardware measurements for auto-tuning with AutoTVM.**

***Keywords*—TVM, neural network compilation, auto-tuning, auto-optimization**

I. Introduction

As deep neural networks (DNN) become larger and deeper, the computational requirements gradually increase, resulting in the development of accelerated model execution as well as optimizing computations of DNN. In addition to specific AI accelerators, such as Google's TPU [1], neural network compilers also play an important role by model optimizations and implementations. Recently, many DNN compilers are proposed, acting as a bridge between the neural network framework and the hardware. For instance, TensorFlow XLA compiler [2], Glow [3], and ONNC [4]. To surpass the hand-written libraries [5], automated compilation with black-box optimizations was developed. AutoTVM [6], built on top of TVM [7], provides auto-tuning with the genetic algorithm and gradient boosting trees like XGBoost [8]. However, it still takes hours to optimize modern DNN models, such as ResNet50 [9], not to mention deeper models such as Inception or ResNet101.

This paper aims to reduce the hardware measurements and accelerate the time-consuming auto-tuning with adaptive workflow control, configuration sampling and a history database. We also extend the local compilation to the online service and provide history data sharing among different users and hyperparameter suggestions for corresponding tasks. We made the following contributions:

1) An adaptive early-stop module that self-tune maximum trial to reduce hardware measurements without affecting model performance. Our design can prevent missing an optimal configuration or making unnecessary hardware measurements;
2) A sampling module that uses K-means clustering to select a centroid to represent a cluster of configurations;
3) A decoupled online tuning service that provides online DNN tuning with decoupled configurations prediction and history record on a remote server;
4) A shared history database module that records and shares the tuning history for different targets and operations among multiple users, providing initial training data for cost model updating, which prevents training the cost model from scratch;
5) Hyperparameter suggestion that provides hyperparameter for modules by analyzing history data within the shared history database and searches for optimal hyperparameter for different tasks.

We evaluated the performance of our method with modern DNN models (ResNet18, InceptionV2) on an Arm processor (Toybrick RK3399), showing 53.33% reduction in hardware measurements, compared with AutoTVM.

II. Adaptive Optimization for Local Tuning

A. System Architecture

Fig. 1 outlines the overall architecture of our design. The architecture contains both local optimizations and advanced online service, which is combination of AutoTVM workflow and Dr. Opt, an auto-guided hyperparameter tuning system. While auto-tuning a DNN model, tunable operations in a model are first converted into tasks. For each task, a tuner is constructed, taking the task information and corresponding history data as its input. The cost model, which predicts the cost of a given configuration, uses these history data as the initial training data. Rather than measuring the exact cost of hardware, the optimizer uses an estimated cost from the cost model to find the optimal configurations within the search space. At each iteration, the candidate configurations selected by the optimizer are sent to the sampling module. The sampling algorithm selects a centroid to represent the cluster. The candidates are then sent back to a local agent for physical measurements. After measuring, the results are saved in both a local log and server-side database. The adaptive early-stop module then analyzes the measurement results and tunes the maximum number of trials. This parameter controls the maximum configurations to be measured on the physical hardware. The measured configurations and measurement results are reported back to the server and then transformed into features for the cost model fitting. The tuner runs iteratively until the trial count eventually reaches the threshold. After auto-tuning, the measured configurations

with the minimum cost are saved and to be used in the code generation stage.

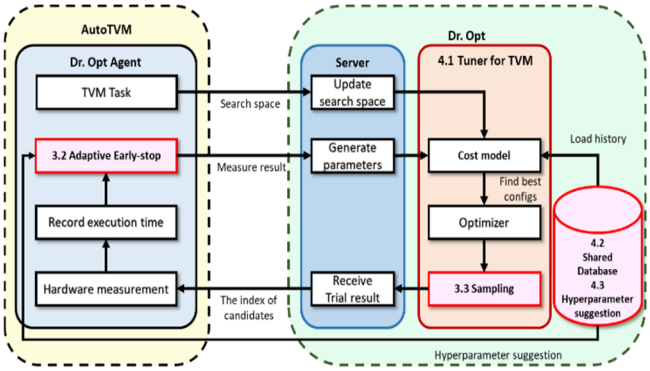

Fig. 1. Adaptive workflow with AutoTVM.

B. Adaptive Early-Stop

The current approach requires the user to assign trial-count and early-stopping for each task before running compilation. Here, the trial-count represents the maximum number of configurations to be measured, and early-stopping represents the threshold to stop tuning in the halfway. For a given early-stopping e, if no better configuration is found in e trials, tuning will be early stopped. Here we face two challenges. First, it is impractical for the user to assign a proper parameter for each task. Setting the early-stop too low leads to incorrectly stopped tuning before an optimal configuration is found, especially when there are not enough initial data for training and thus more iterations are needed. On the other hand, setting the early-stop too high results in unnecessary time-costly hardware measurements. Second, the current design resets the counter of early-stops only when finding better configurations. It does not take the measurement results into consideration. When facing a significant performance drop for a new batch, which often happens after finding an optimal configuration, the tuner will not stop precisely. To overcome the challenges, we propose an adaptive early-stop, a module that adaptively tunes the maximum trials by real-time measurement results. Fig. 2 shows the workflow of the adaptive early-stop. A block contains multiple configurations and corresponding results. When all configurations in a block are measured, results are used to update the cost model. Here, the configuration c represents the configuration with a minimal cost within the current block, configuration c^* represents the configuration with a minimal cost within all previous blocks, and $f(x)$ represents the FLOPS of this configuration. The adaptive early-stop module contains seven parameters:

im: initial maximum trial

ir: increment ratio; controls the increment of maximum trial for cases, where

$$\frac{f(c)}{f(c^*)} > 1$$

dr: decrement ratio; controls the decrement of maximum trial for cases, where

$$\frac{f(c)}{f(c^*)} < 1 - fr$$

fr: FLOPS ratio; defines the size of the interval

ub: upper bound; limits the maximum value of variable Add

lb: lower bound; limits the minimum value of variable Add

Plan-size: number of configurations within a batch; the default parameter in AutoTVM.

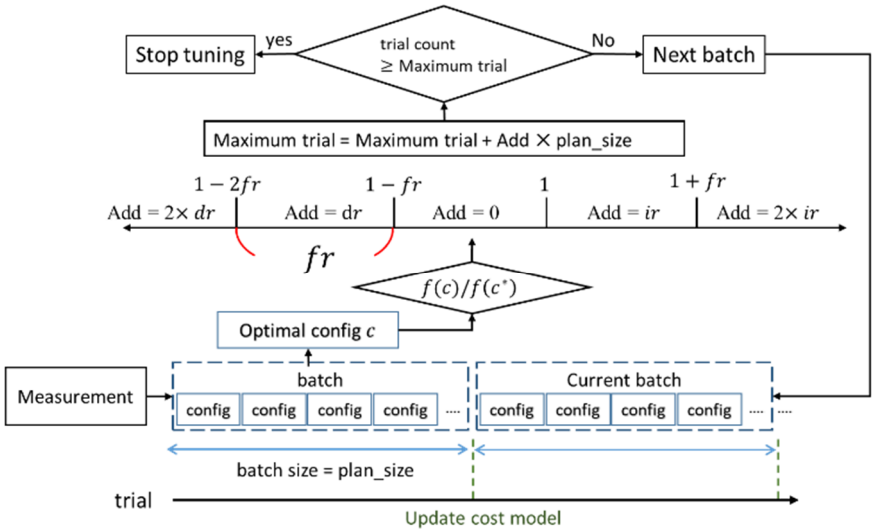

Fig. 2. Workload of the proposed adaptive early-stop.

After a batch of configurations have been measured, we first compare $f(c)$ with $f(c^*)$. If the value of $f(c)/f(c^*)$ falls into the interval between 1 and 1-*fr*, the maximum trial is not changed and we expect a higher probability to find a better configuration in later batches. For cases where $f(c) > f(c^*)$, the maximum trial is increased to encourage further hardware measurements. As cases for $f(c) < f(c^*)\times(1\text{-}fr)$, we decrease the maximum trial. The upper bound and lower bound are chosen to prevent dramatically changes on the maximum trial, so the tuning will not accidently stopped, when a batch of configurations have poor performance. If c performs better than c^*, c^* will be used instead of c in later measurements. The tuner starts a new iteration until the trail count exceeds the maximum trial. Following this policy, the adaptive early-stop module can predict the trend of later measurements and will adaptively change the maximum trial. It stops the tuning

task in advance whenever the performance drops, and extends the tuning workload if better configurations are possible. Also, it handles the cases where the maximum performance slightly increases. The situation happens when the minimum cost of configurations of each batch are very close. While the original approach simply resets the counter of early-stop, the adaptive early-stop increases the maximum trial depending on the improvements of performance.

C. Sampling

While analyzing the measured configuration for each task, we notice that configurations selected by the cost model often fall in some region within the search space, such as the example shown in Fig. 3. Also, it appears that many of the adjacent configurations turn out to have a similar cost. Utilizing the characteristic, we use a sampling module to sample the candidate configurations selected from the cost model. This module uses the K-means clustering algorithm to separate candidates to different clusters. Fig. 3 shows the example of the K-means clustering. The algorithm aims to partition n points into k clusters, in which each point belongs to the cluster with the nearest mean. Iterate through the number of clusters, the algorithm partitions configurations into different clusters. We use the Euclidean distance of each configuration in the search space as loss. For each iteration, we check the total distance until reaching the threshold. The centroid represents points within a corresponding cluster. The number of cluster K represents the trade-offs between more centroids resulting in better performance and fewer centroids for a reduction of measurements. This prevents candidates from the cost model from lying in a small region, which is unfavorable for updating the cost model.

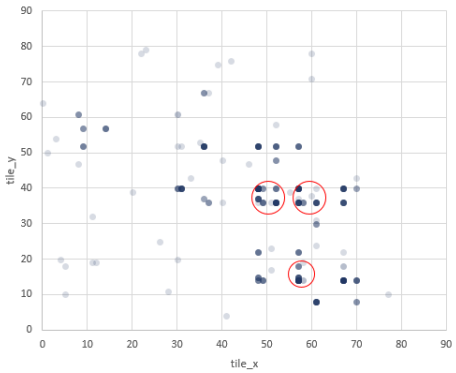

Fig. 3. Example of configurations with similar costs.

III. Transfer Learning and History Database

A. Tuning from History Database

When training the cost model from scratch for a given task, it requires hundreds of hardware measurements for the cost model to have an enough accuracy for prediction. AutoTVM provides transfer learning to accelerate the tuning. Fig. 4 show the framework. Using a log file loading function, the cost model of a new task loads the temporary history data from the previous tasks as the initial training data. However, there are limitations of this approach. First, the first few tasks could not have enough data for training. Second, using data from previous tasks sometimes result in low accuracy, as these tasks have different arguments, including input sizes and kernel sizes. In these cases, the tasks require ever more hardware measurements to explore an optimal configuration, compared with not using any initial training data. To solve this problem, we create a history database implemented in SQLite for recording the history data as shown in Fig. 4 (b). Records are separated according to the task name and task argument. Each row in a table contains measurement inputs and measurement results. Measurement input records the target hardware, task and the corresponding configuration, and measurement result records the cost for this task. All records in same table are measurement records of different configurations for a same task. If the current task is found in the database during tuning, the tuner will load the history data from the corresponding table.

B. Data Organization

After multiple model tuning records are saved into the database, there is often a case that a same configuration is measured for several times, while the measured costs are different. In the circumstances, we provide two maintenance methods. The first is maintained by date. As each record inside the table contains a date column recording the measurement date, this method simply reserves the newest record and removes all other duplicated records. The second is maintained by average. When cleaning duplicated records, for each configuration with multiple measurements, we calculated the average cost of the duplicated records, and replace these records with a new record with the average cost.

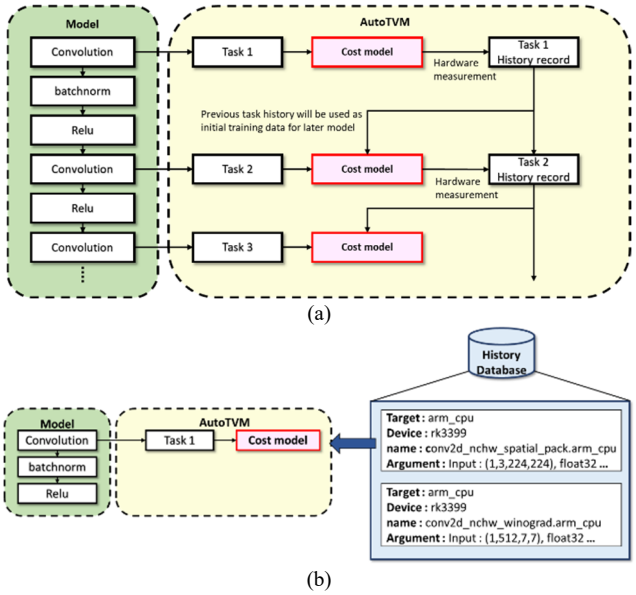

Fig. 4. Framework of transfer learning and history data.

IV. Decoupled Online Tuning Service

Our work is based on the structure of NNI and Dr. Opt. With our specific tuner design for AutoTVM, users can easily tune the DNN models and heavy computations for the cost model and the optimizer, the database interaction and the hyperparameter suggestion will be served on the server. This allows the server maintainer to update different tuners and algorithms without modifying the client-side program. Thus, users can easily try different algorithms and features with simple setting in the local tuner. As shown in Fig. 5, the original tuner is decoupled into two parts, the client-side agent and the server-side tuner. The client-side agent handles

the control flow and physical measurements on the hardware device, while the server-side tuner deals with the cost model, the shared history database and hyperparameter suggestion.

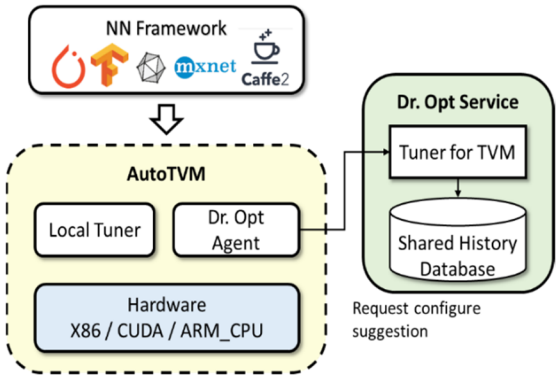

Fig. 5. Decoupled server tuner and local agent.

V. Experimental Results

We first evaluate the performance of each component of our proposed methods, and then eventually measure the whole tuning process with the proposed techniques. The target hardware is the Arm processor (Toybrick RK3399) and the host device is 32-core Intel Xeon E5-2650 v2 at 2.60GHz. We evaluate the three modules on ResNet18 and select the convolution as the tuning operation. After optimization, the convolutions are converted to 15 tasks. The comparison is based on the hardware measurement count in tuning process. Also, we make an end-to-end evaluation on multiple modern DNN models, including ResNet18 and InceptionV2. We compare the total compilation times and the total hardware measurements required in auto-tuning. Fig. 6 shows the comparison between the adaptive early-stop and AutoTVM for inference time. The first configuration (trial count: 1500, early-stop: 800) is a default value in the official document. The second configuration (trial count: 800, early-stop: 400) uses better parameters by the user. Comparing to manually-selected parameters in second configuration, the adaptive early-stop reduce 33.4% hardware measurements of tuning ResNet18. It is worth noticed that in cases where suboptimal configurations perform closely, such as task 12 and task 14, early-stop gets reset with a minor performance increase. The adaptive early-stop gains even more in these cases due to the proposed mechanism. Fig. 7 shows the number of candidates for measurements for tuning ResNet18 with and without applying the sampling module with the inference time. In average, the sampling module removes 16.5% of candidates for hardware measurements. Performance of the sampling module depends on the distribution of configurations within the space. For cases where the candidates from the cost model have similar combinations of knobs, sampling can have better results. Fig. 8 shows the measurement counts for finding an optimal configuration, The first configuration is running auto-tuning without initial data for the cost model. This means the cost model for all tasks are trained from scratch and uses only measurement results of that task as the training data. The second configuration uses a built-in transfer learning mechanism, where a later task uses the measurement results of the previous tasks as the initial training data. The third configuration uses our proposed history database as the initial training data. In average, the transfer learning gains 12.3% reduction in hardware measurements, and our history database reduces 21.4% hardware measurements, compared with the original transfer learning method.

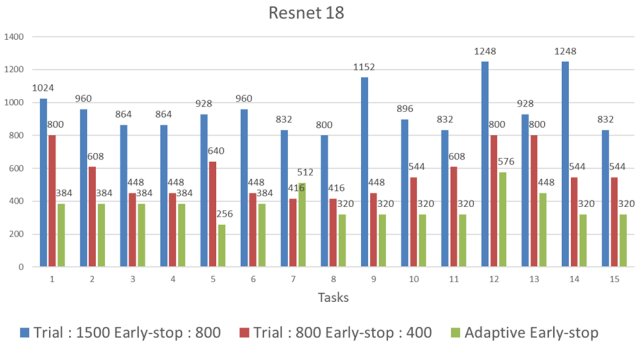

Fig. 6. Evaluation of adaptive early-stop.

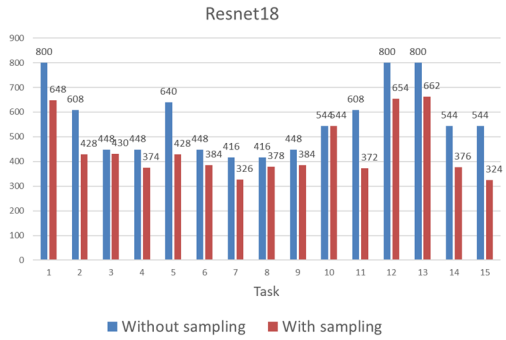

Fig. 7. Evaluation of sampling.

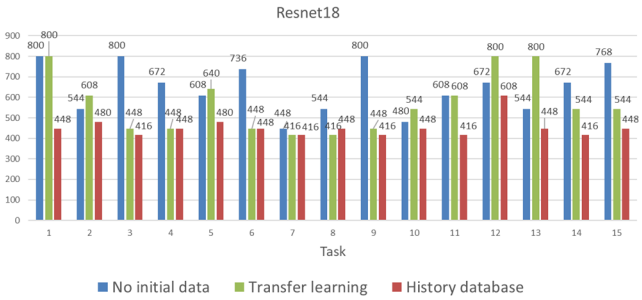

Fig. 8. Evaluation of measurement counts.

VI. Related Works

Frameworks such as PyTorch [10], TensorFlow [11], Caffe2 [12], or MxNet [13] provide a solution for users to design and train neural networks. DNN compilers mainly focus on model optimizations in both graph-level and tensor-level to increase the inference speed. Fig. 1 shows the overall workflow of DNN compilation from a high-level framework to the deployments on physical hardware devices, following the structure of TVM. The first step is to transform the framework model to the intermediate representation by a frontend compiler. Second, target-independent and target-dependent optimizations are applied. In this stage, some compilers provide quantization to reduce computation requirements. Target-independent optimizations, such as operation fusion, constant folding, and layout transformation, do not require information of the target hardware. Target-dependent optimizations, such as the AlterOpLayout pass, act differently depending on the selected target hardware. Further tensor-level optimizations leverage loop transformation or cache locality. Some researchers proposed templates to be used as the search space for operations of different targets

[14, 15]. The black-box optimizations require time-consuming measurements of each configuration within schedule space, which greatly increase the compilation time. Many studies for auto-tuning had proposed different methods for accelerating the compilation. In the framework of Chameleon [16, 17], two approaches were proposed to speed up the compilation. First, a model optimizer using simulated annealing is replaced with a reinforcement learning model, which learns the trend of the configuration cost and finds an optimal configuration. Second, a new sampling module was proposed to reduce similar configurations and predict an optimal configuration by combining frequently-seen knob entities. While the default tuner in AutoTVM uses XGBoost and a genetic algorithm, another research [18] applies Greedy Best-First-Search (G-BFS) and Neighborhood Actor Advantage Critic (N-A2C) to search an optimal configuration for matrix multiplications to gain a better performance.

VII. Conclusion

We proposed multiple optimizations for accelerating the DNN compilation. First, we proposed the adaptive early-stop, sampling and the history database, to reduce hardware measurements in local tuning. Second, we designed an online tuning service, which decoupled history-data-related works to server, and provided data sharing among multiple users and hyperparameter suggestion. Experimental results show that our approach greatly reduces hardware measurements for auto-tuning and effectively accelerates the model compilation. With the updated cost model and database organization, it has potential to gain ever better performance.

Conflict of Interest

The authors declare no conflict of interest.

Author Contributions

Yu-Hsiang Chen and Tien-Fu Chen conducted the research, Yu-Hsiang Chen and Juin-Ming Lu analyzed the data; Tien-Fu Chen and Tay-Jyi wrote the paper; all authors had approved the final version.

Funding

This research was funded by National Science and Technology Council, grant number NSTC 113-2622-8-A49-009.

Acknowledgment

The authors wish to thank Dr. Shih-Chieh Chang of Industrial Technology Research Institute for his valuable comments and supports.

References

[1] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proc. ISCA*, 2017, pp. 1–12
[2] XLA. [Online]. Available: https://www.tensorflow.org/xla
[3] N. Rotem *et al.*, "Glow: graph lowering compiler techniques for neural networks," arXiv abs/1805.00907, 2018.
[4] W. F. Lin *et al.*, "ONNC: A compilation framework connecting ONNX to proprietary deep learning accelerators," in *Proc. AICAS*, 2019.
[5] S. Chetlur *et al*., "cuDNN: efficient primitives for deep learning," arXiv abs/1410.0759, 2014.
[6] T. Chen *et al*., "Learning to optimize tensor programs," in *Proc. NeurIPS*, 2018, pp. 3393–3404.
[7] T. Chen *et al*., "TVM: An automated end-to-end optimizing compiler for deep learning," in *Proc. OSDI*, 2018, pp. 579–594.
[8] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proc. KDD*, 2016, pp. 785–794.
[9] P. Goyal *et al.*, "Accurate, large minibatch SGD: training ImageNet in 1 hour," arXiv abs/1706.02677, 2017.
[10] PyTorch. [Online]. Available: https://pytorch.org/
[11] M. Abadi *et al.*, "Tensorflow: A system for large-scale machine learning," in *Proc. OSDI*, 2016, pp. 265–283.
[12] T. Chen *et al.*, "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," in *Proc. LearningSys*, 2015.
[13] Y. Liu *et al.*, "Optimizing CNN model inference on CPUs," in *Proc. USENIX ATC*, 2019, pp. 1025–1040.
[14] L. Wang *et al*., "A unified optimization approach for CNN model inference on integrated GPUs," in *Proc. ICPP*, 2019, pp. 1–10.
[15] B. H. Ahn, P. Pilligundla, A. Yazdanbakhsh, and H. Esmaeilzadeh, "Chameleon: Adaptive code optimization for expedited deep neural network compilation," in *Proc. ICLR*, 2020.
[16] B. H. Ahn, P. Pilligundla, and H. Esmaeilzadeh, "Reinforcement learning and adaptive sampling for optimized DNN compilation," arXiv abs/1905.12799, 2019.
[17] H. Zhang, X. Cheng, H. Zang, and D. H. Park, "Compiler-level matrix multiplication optimization for deep learning," arXiv: 1909.10616, 2019.
[18] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. NIPS*, 2012.