A History-Based Auto-Tuning Framework for Fast and High-Performance DNN Design on GPU

Jiandong Mu[1], Mengdi Wang[2], Lanbo Li[2], Jun Yang[2], Wei Lin[2], Wei Zhang[3]
[1,3]Hong Kong University of Science and Technology (HKUST), China
[2]Alibaba Group, China
[1]jmu@connect.ust.hk, [2]didou.wmd, lanbo.llb, muzhuo.yj, weilin.lw@alibaba-inc.com, [3]wei.zhang@ust.hk

***Abstract*—While Deep Neural Networks (DNNs) are becoming increasingly popular, there is a growing trend to accelerate the DNN applications on hardware platforms like GPUs, FPGAs, *etc.*, to gain higher performance and efficiency. However, it is time-consuming to tune the performance for such platforms due to the large design space and the expensive cost to evaluate each design point. Although many tuning algorithms, such as XGBoost tuner and genetic algorithm (GA) tuner, have been proposed to guide the design space exploring process in the previous work, the timing issue still remains a critical problem. In this work, we propose a novel auto-tuning framework to optimize the DNN operator design on GPU by leveraging the tuning history efficiently in different scenarios. Our experiments show that we can achieve superior performance than the state-of-the-art work, such as auto-tuning framework TVM and the handcraft optimized library cuDNN, while reducing the searching time by 8.96x and 4.58x comparing with XGBoost tuner and GA tuner in TVM.**

***Index Terms*—Hardware Optimization, Auto Tuning, DNN.**

I. Introduction

Nowadays, deep neural networks (DNNs) is gaining traction as they have demonstrated impressive performance in various areas such as image classification, object recognition, *etc.* However, the high performance of the DNN algorithm is achieved at the price of heavy data transferring and high computing complexity. To solve this problem, many hardware devices and optimization methodologies have been proposed to optimize the DNN applications efficiently.

GPU has been widely used in many fields to accelerate the parallelizable algorithms like DNN due to its single instruction multiple threads (SIMT) nature and intensive on-chip computing units. In addition to the emerging powerful devices, optimizations from multiple perspectives have also been proposed to further improve the performance on-board. For example, the algorithm level optimizations include Winograd, FFT, depthwise convolution, *etc.*, and the hardware-aware optimizations include cache hierarchy, vectorization, double buffer, *etc.*

The diversity of the optimizations makes the design space exploration non-trivial. As a result, the auto-tuning process has been proposed to automatically find the optimal schedule, which denotes a specific mapping from an operation to the low-level code running on GPU. To achieve this, different schedule templates are provided to implement the aforementioned optimizations and the knobs in the schedule templates, *e.g.* tiling factor, unroll factor, are used to indicate the parameters to be tuned. During the auto-tuning process, values that maximize the performance on-board will be searched according to the tuning algorithm and the feedback from the device measurements.

However, the current auto-tuning process is time-consuming for several reasons. Firstly, the schedule space is large since there are many tunable knobs. Secondly, existing tuners fail to explore the schedule space efficiently. For example, the widely used XGBoost [1] tuner in TVM [2] builds a boosted decision trees [3] based model to predict the performance, however, the workload for training the model and finding its global minimum for each layer is heavy. Finally, evaluating each design point is time-consuming since running the schedules on-board takes time. As a result, the total design points can be accessed is limited. The cost-expensive nature of auto-tuning has made it prohibitive for emerging applications like auto-pruning and network architecture search, where intensive schedule tuning are conducted to train the learning agent to guide the next pruning policy or network architecture design. For a simple network like ResNet-18 [4], it takes hundreds of epochs (200 in PocketFlow [5], 400 in AMC [6]) for the agent to converge and it takes about 10 hours for the XGBoost tuner to tune schedule in each epoch (measured based on our server which has an Intel Xeon E5 and 8 GeForce GTX-1080Ti), which leads to a running time of more than 3 months.

In this work, we aim to accelerate the auto-tuning process so that the optimal schedule running on-board can be obtained efficiently. To achieve this goal, we propose a novel auto-tuning framework that explores the schedule space efficiently by taking advantage of the previous tuning history in different scenarios. More specifically, a transfer learning (TL) based auto-tuning flow is used to speed up the convergence of the performance model by distilling the history with a novel filtering algorithm. A guided genetic algorithm (GGA) based auto-tuning flow is proposed to find the optimal schedule by exploring the candidate schedules in history database, followed by a guided crossover and mutation based on the elite schedules and human design experience. It requires much fewer measurements on-board than the model-based counterpart since the performance model training process is no longer needed. A match score calculator is used to evaluate the similarity of the input operator and the history database, thereby, determine which tuning flow to apply.

In summary, we made the following contributions:

- We propose a novel filtering algorithm for the TL-based auto-tuning which can improve the tuning performance and converging speed with little computing overhead.
- We propose a novel GGA-based auto-tuning algorithm, which explore the schedule space by leveraging the history records and expertise. Experiments show that our GGA algorithm can achieve superior performance with 8.96x and 4.58x less time than the state-of-the-art XGBoost tuner and GA tuner, respectively.

The remaining paper is organized as follows: Section II states the background and related works of this paper. Section

978-1-7281-1085-1/20/$31.00 ©2020 IEEE

III presents our framework to find the optimal schedule efficiently. Section IV shows the experiments to verify the validity of our algorithms, and section V concludes the paper.

II. Related Works

In this section, we will review the related works from two perspectives: *Handcraft Optimizations* for GPU, and *Auto-Tuning flow* in which the hardware-related features are automatically taken into consideration.

A. Handcraft Optimizations

Many hardware optimizations have been proposed to accelerate DNNs efficiently. In [7], the author provided an open-source end-to-end system that runs a wide range of CNN models on integrated GPUs. A thorough investigation on optimizing the vision-specific and computational intensive operators is done as well; In [8], the author optimized the memory efficiency of CNN applications by revealing the performance impact of different data layout. A light-weight heuristic is proposed to select the optimal layout and a fast multi-dimensional data transformation is proposed to switch the layout between the adjacent layers; [9] improves the memory efficiency of direct convolution on GPUs by introducing a general model to address the mismatch between the SM bank width and computation data width of threads, while in [10], a comprehensive study of multiple algorithms of CNNs on different platforms is provided and possible optimization methods to increase the efficiency of CNN models are given.

B. Auto Tuning Flow

Auto tuning is proposed to automatically find the optimal schedule running on-board considering the various optimizations mentioned before. TVM [2] is one of the representative works. TVM tunes the network by optimizing each DNN operator with the internal templates, which contain many tunable knobs. The appropriate values for the knobs are found by searching for the global minimum of the performance model.

However, TVM fails to meet the requirements in many scenarios for the following reasons: Firstly, the prediction model in TVM targets a specific layer. A new model has to be trained from scratch if a new layer is expected to be tuned. However, in the case of auto pruning and network architecture search, thousands of different layers are expected to be optimized automatically. TVM fails to satisfy this scenario since its too time consuming to construct the performance model and find its minimum value. In [11], the author tries to relieve this problem by using the reinforcement learning agent instead of the simulated annealing algorithm to find the minimum value of the performance model. However, the timing issue still remains a critical problem. Secondly, in TVM, although the tuning record can be shared by directly loading the history of the same template, it fails to utilize the history data efficiently as the massive history data may not benefit the convergence of the performance model, while consuming extra computing power.

In [12], the author provides a framework, which combines the local search with the global search, to auto-tune the inference on CPUs. The local search will find the optimal parameter combinations for each computational intensive layer, while global search will optimize the inference at the graph level. However, this tuning framework also suffers from the time-consuming issue. As its experiment shows, it takes 6 hours to search for the 20 different *conv* layers in ResNet-50 on an 18-core Intel Skylake processor.

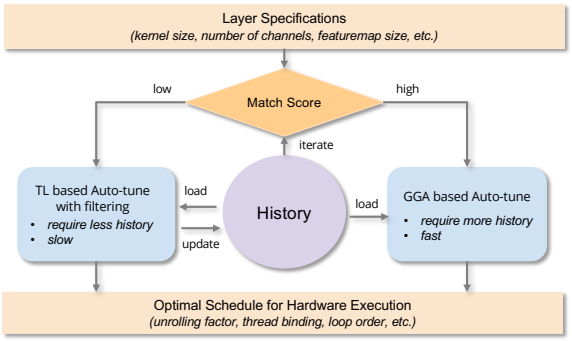

Fig. 1. Illustration of our proposed auto-tuning framework.

In contrast, our framework can filter the history results and leverage the proper history results to speed up the design space searching of the optimal design knobs, and hence significantly improve the time of networking tuning process. We further improve the tuning speed by proposing a novel GGA-based tuning algorithm. The experiments show that our framework can accelerate the tuning process significantly with superior performance.

III. Accelerate Hardware Optimization

The framework of the proposed auto-tuner is illustrated in Fig. 1. Our auto-tuning framework consists of a history database and two phases, which are *TL-based auto-tuning with filtering* and *GGA-based auto-tuning*, respectively, to cope with the schedule optimization problems in different scenarios, so that the history can be efficiently used. When the input network is forwarded to our framework, it is decomposed into layers since our hardware optimization will be conducted in a layer by layer manner. Then, a match score, which represents the similarity of the input layer with the history records, will be evaluated to determine the later control flow for each layer. The match score proposed for the fallback flow in TVM source code is used here and can be illustrated by the following equation.

$$s\left(h,i\right)=\sum_{k\in\{knobs\}}\frac{len\left(factor(h[k])\cap factor(i[k])\right)}{len(factor(h[k]))\times len(knobs)} \quad (1)$$

where $h[k], i[k]$ is the kth knob of the history record and input, respectively. Function $factor$ intends to find the factor list of the given argument.

If the match score is lower than the threshold, which indicates the history records doesn't match the given input settings very well, we will go with the TL-based tuning flow that a performance model will be learned according to the tuning history with an efficient history filter, and the optimal hardware executing schedule will be found by finding the minimum value of the performance model, which is similar with the existing tuning flow in TVM. However, our TL-based auto-tuning differentiates with the existing flow in its ability to filter the history of different layer specifications in the database, therefore, the optimal schedule can be obtained efficiently with little computing overhead. The schedules explored in the TL-based tuning process will also be added to the history database so that the database becomes more and more comprehensive.

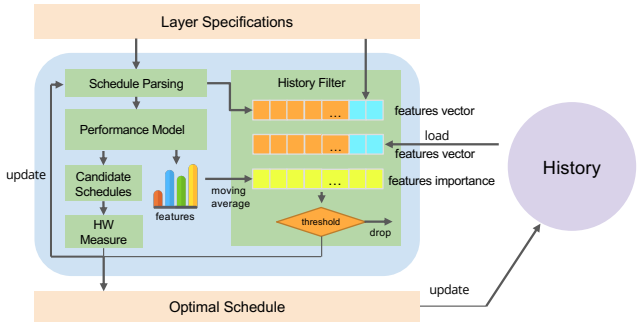

Fig. 2. Illustration of TL-based auto-tuning flow.

On the other hand, if the match score is higher than the threshold, which indicates the history database is comprehensive and similar schedules with the input layer specification can be found in the database, we propose to use a GGA-based auto-tuning flow instead of learning the performance model from scratch so that the optimal schedule for the hardware execution can be found during the guided crossover and mutation process.

The TL-based auto-tuning and the GGA-based auto-tuning are further explained in the following sections.

A. TL-based Auto-tuning with Filtering

The details of the TL-based auto-tuning algorithm are illustrated in Fig. 2. It's non-trivial to reuse the history of different layer specifications for several reasons. Firstly, the tuning history for different input settings may be irrelevant or contradictory. Naively applying the history records may poison the model, thus, leading to a compromised converging speed and performance. This can be illustrated by our experiment in Fig. 4. Secondly, the large amount of the available tuning history may cause a growing training complexity, which diminishes the benefits that transfer learning has brought. To overcome the aforementioned problems, a history filter is proposed to select the history records based on the features' importance and similarity to avoid the irrelevant information and the prohibitive computing overhead.

The history records are firstly loaded from the database for filtering purposes. To be compatible with the state-of-the-art work, our history database adopts the same history format as in [2], and a similar schedule parser is deployed to parse the history records into feature vectors, which contains the statistic information for each knob in the schedule templates and serves as the input of the performance model. In order to take advantage of the history of different layer specifications, we concatenate the feature vector with the input specifications to remove the ambiguity of the feature vectors for different inputs.

Then, the history filtering is conducted based on the similarity of the feature vectors. However, the naive cosine similarity, which measures the similarity of the two feature vector based on the angle in the inner product space, may not work in this case since different features may have distinct impacts on the running latency and throughputs. Indeed, many features have zero impact on the performance model, and filtering history according to such features may misguide the model and lead to compromised performance. Alternatively, we select the history according to the similarities together with the feature importance as shown in Fig. 2. In consideration of the decision-tree-based performance model, which is commonly used in the performance modeling process, the feature importance is obtained by applying a moving average [13] to the statistics of the numbers of times that a feature splits the decision tree. This is based on the fact that features, which split the decision tree more frequently, are more important. The moving average is used to smooth out the short term fluctuations and to highlight the long term trend.

Finally, the history records can be selected according to the similarity with a scaling factor of the feature importance, and the training process of the performance model can be accelerated by reusing the selected history data, thus, reducing the number of the measurements on-board. The most attractive candidate schedules are then predicted by finding the minimum value of the performance model and is used to update the performance model with the measurements on-board as ground truth. Meanwhile, the schedules and the corresponding measurements on-board will be used to enrich the history database. This process will be repeated until the performance converges and the optimal design is found.

B. GGA-based Auto-tuning

As the history database getting more and more comprehensive, the match score of the input layer and the history database may be higher than the threshold, which indicates similar records with the given input specifications can be found in the history database. As a result, it's possible to find the optimal hardware design for the input layer efficiently without the time-consuming model training and global minimum solving process. The flow of our GGA-based auto-tuning algorithm is illustrated in Fig. 3.

Similarly like in TL-based auto-tuning flow, a history filter is also used to prevent the irrelevant history from poisoning the candidates in the GGA-based auto-tuning process. However, noting that we will need to utilize the performance value of the history record in the tuning flow, a new history filtering algorithm is proposed to consider both the history performance and the similarity between the input layer and the history record. The metric we used to select the candidate records is illustrated in the following equation.

$$m(h,i) = Th(h)\gamma(h,i)s(h,i)^{\omega} \tag{2}$$

where $m(h,i)$ is the filtering metric of history h and input i, $Th(h)$ is the throughput of the history schedule, and $s(h,i)$ is the match score as we mentioned before. ω serves as a regulation factor to balance the tradeoff between the match score and the throughput, and we found $\omega = 2$ is a preferable choice in practice. Note the parallelism in the history schedule may not be fully utilized by the given input layer specifications, since the input layer may not provide enough workload for the parallelism. Therefore, $\gamma(h,i)$ is used as a scaling factor to characterize the utilization ratio when the history schedule is applied to the current input specifications. It can be fully expanded into the following form:

$$\begin{aligned} \gamma(h,i) &= \prod_{k\in\{knobs\}} \gamma(h,i,k) \\ \gamma(h,i,k) &= \begin{cases} \frac{prod(i[k])}{\lceil \frac{prod(i[k])}{P(h,k)} \rceil \times P(h,k)} & \text{if } prod(i[k]) > P(h,k) \\ \frac{prod(i[k])}{P(h,k)} & \text{if } prod(i[k]) \le P(h,k) \end{cases} \end{aligned} \tag{3}$$

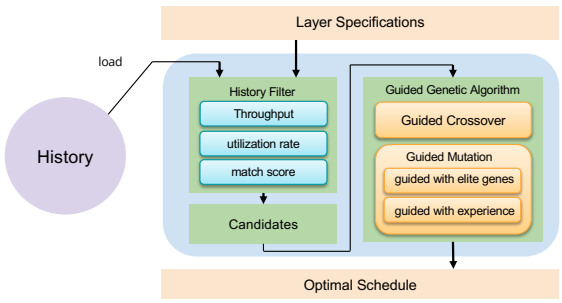

Fig. 3. Illustration of GGA based auto-tuning algorithm.

Algorithm 1: GGA-based Auto-tuning

```
Input: CandSchedules
Output: OptSchedule
init N, P; ▷ N is the number of generations, P is the population size;
init EliteGenes = {};
init Genes = {};
while n ≤ N do
    Rets = HWMeasure(CandSchedules) ;
    EliteGenes = select(Rets, Genes) ;
    if Crossover then
        Probs = calProb(Rets) ;
        while p ≤ P do
            Parents = select(Rets, Probs);
            OffSpring, NewGenes = crossOver(Parents);
            p++ ;
    else
        while p ≤ P do
            if MutationType = ELITE then
                OffSpring, NewGenes =
                  mutateElite(CandScheduless, EliteGenes);
            else
                OffSpring, NewGenes =
                  mutateExperience(CandSchedules);
            p++;
    CandSchedules = select(OffSpring);
    Genes += NewGenes;
    n++;
OptSchedule = top(CandSchedules)
```

where $\gamma(h, i, k)$ stands for the utilization ratio of knob k, P stands for the parallelism, and $prod(i[k])$ is the dimension size of knob k in the input specification.

Given the definition of the filtering metric, the candidate schedules can be selected by traversing the history database. However, directly falling back to the history schedules fails to explore the new design points and may return a sub-optimal design. Existing DSE algorithms, such as genetic algorithm (GA), could be one choice to explore new schedules based on the candidate schedules. However, we found the vanilla genetic algorithm fails to explore the schedule space efficiently for two reasons. Firstly, GA explores the schedule space in a random nature. This is because the chromosome switched in the crossover process and new genes in the mutation process are both randomly produced to imitate the biological evolution process. Therefore, it fails to utilize the experience and previous results efficiently. Secondly, the schedule space is highly non-continuous which leads to significant performance degradation after the random perturbation during the mutation and crossover process.

To solve the problem, a guided GA-based auto-tuning algorithm is proposed to search for the optimal executing schedules more efficiently. The proposed algorithm is illustrated in Alg. 1, where genes refer to the knobs in the schedule. A dictionary, Genes, which maps the individual to the gene, can be used to reference the offspring and its corresponding muted gene without ambiguity since we mutate one gene each time. The mutated genes of the high-performance individuals are considered as elite genes and are stored in the EliteGenes dictionary to guide the later mutations.

In each generation, the candidate schedules are firstly measured on-board to get the hardware performance. Although running on-board is time-consuming, this process is acceptable due to our small population size requirements (P=16 is sufficient for our experiments). Then the elite genes can be selected according to the on-board performance and the Genes dictionary. We sort all individuals and consider the top k (k=8 in our experiments) individuals as elite individuals and the corresponding mutated gene are considered as elite genes.

Then, A guided crossover or mutation is conducted with the user-specified probability.

1) Guided Crossover: In the guided crossover algorithm, the rank of the performance is considered as the fitness function, which indicates the quality of the schedule. The parents are selected proportional to the fitness function so that high-performance schedules are more likely to be selected. Then we switch one gene each time and record the gene in the Genes dictionary in favor of the later elite genes analysis.

2) Guided Mutation: To overcome the inefficiency in vanilla genetic algorithm, random mutation is aborted in our algorithm. Alternatively, we remedy this problem by proposing two kinds of guided mutations, which are elite genes based mutation and expertise based mutation, respectively.

Being aware of our elite gene dictionary, which is obtained by selecting the high-performance schedules and their corresponding mutated genes, we randomly replace the genes in the parents with the elite gene based on the intuition that elite genes are more likely to contribute to a better solution. We realize that the dependency and interference between multiple genes are complex and currently are not considered.

We also mutated the genes according to the expertise for some knobs. For example, the tiling size for different dimensions usually prefers the numbers, which the dimension size can be divided by. This is because the remainder usually indicates a resource wasting, thus, tiling sizes, which are factors of the loop size, are more appreciated. Besides, small factors like 2 or 4 are also added in the mutation space according to the design experience.

Instead of choosing the top P schedules from the offspring, we generate the population of next-generation by selecting the schedules with the probability proportional to their performance to increase the genetic diversity, thus, avoiding the local minimum. To ensure the best schedule can be inherited in this process, the child with the highest performance is inserted into the population manually.

The experiments have verified that our guided-GA can effectively take advantage of the previous measurements and human experience, therefore, leading to a significant improvement in the throughput with much less time.

IV. Experiments

Our framework is running on an Intel(R) Core(TM) Xeon E5-2682 CPU@2.50GHz with a 500 GB DDR memory. TVM 0.6.dev and Tensorflow 1.10 are employed to parse the history

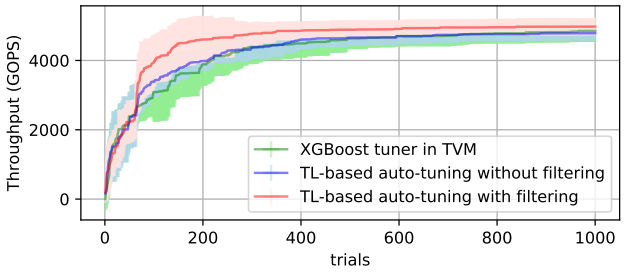

Fig. 4. Comparison of our TL-based auto-tuning and XGBoost tuner in TVM.

record and running the schedules on-board. All the experiments are conducted on Nvidia 1080Ti, which has 2560 cores and a boost frequency of 1733MHz, leading to a peak throughput of 8873 GFLOPS. The Nvidia 1080Ti is connected with the host machine via a PCI-e 3.0 interface which offers a maximum bandwidth of 8GT/s. The widely used CUDA-9.0 is used to program the DNN applications on GPU efficiently.

A. TL-based Auto-tuning

To demonstrate that our TL-based auto-tuning can improve the converging speed and performance, 15 experiments are conducted for XGBoost tuner in TVM and our TL-based auto-tuning, respectively, with the first convolutional layer in the ResNet-18 used as the benchmark. Each experiment is executed for 1000 trials which are recommended in the TVM tutorial. The average of the 15 experiments and the corresponding standard deviation are shown in Fig. 4 to indicate the converging speed and final performance. To validate the effectiveness of our proposed history filtering algorithm, the experiment for TL-based auto-tuning without filtering is also provided for comparison.

It can be observed from the experiments that it takes 800-1000 trials for XGBoost tuner to converge, and the TL-based auto-tuning without filtering has a similar performance although there are some marginal benefits during the initial trials (0-200). This is because not all the records in the history database will be relevant to the current input layer and benefit the corresponding performance model, therefore, leading to less satisfying performance while consuming precious computing resources. However, the red line in Fig. 4 indicates our proposed filtering algorithm can distill the history effectively and our TL-based auto-tuning with history filtering can significantly improve the converging speed and throughput.

B. GGA Based Auto-tuning

We compare the fallback algorithm, which finds the executing schedules by falling back to the schedule with the highest match score in the history database, GA tuner, XGBoost tuner in TVM, and our GGA-based auto-tuning in Fig. 5. We use 1000 trials for the GA tuner and XGBoost tuner for a stable performance. The history database provided by TVM is utilized for the fallback algorithm and our GGA-based auto-tuning algorithm to get a fair comparison.

We provide experiments for different scenarios to justify the superiority of our algorithm. Fig. 5 (1)(2) shows the experiments for all the convolutional layers in ResNet-18 and VGG [14], where *conv x-y/z* in the horizontal axis indicates the yth and zth layer in block x since they share the same

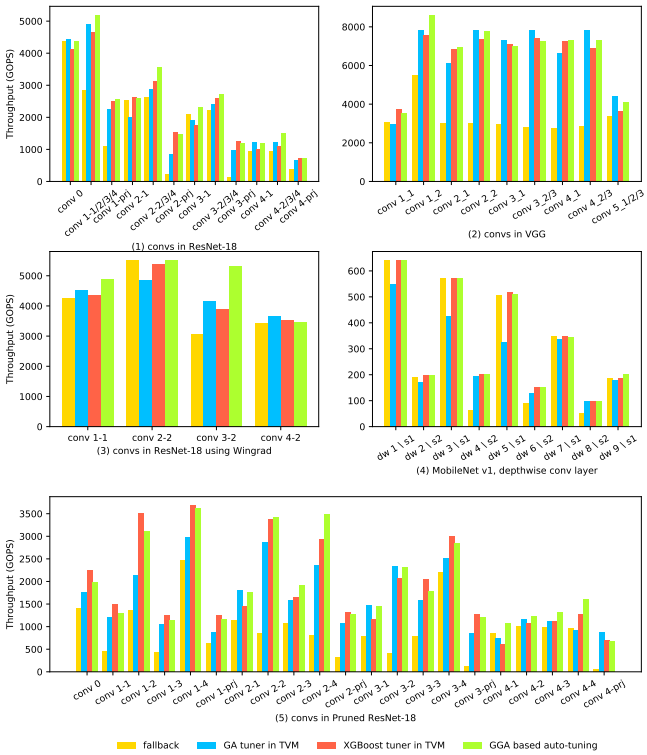

Fig. 5. Performance of GGA based auto-tuning algorithm for different scenarios.

layer specifications. We also include the experiments for the shortcut layers in the ResNet which is referenced by *prj*. The experiments show that our GGA-based auto-tuning algorithm can achieve comparable or better performance for different neural network models than the TVM counterpart while using much less searching time. The fallback algorithm provides an acceptable performance if the input layer is included in the database, however, may suffer from great performance degradation otherwise. In Fig. 5 (3), we show the experiments for Winograd-based convolution. Only convolutions with a kernel size of 3 and stride of 1 are considered here. In Fig. 5 (4), we show our tuning algorithm also works for other layers such as depthwise convolution, where s indicates the stride. All the depthwise convolutional layers in the MobileNet-v1 are included. We also validate the proposed tuning algorithm for the pruned ResNet-18, which is pruned using a reinforcement learning (RL) agent as proposed in [6] and is independent of the open-source history database, in Fig. 5 (5). It is worth noting that the channels after pruning can be any number, which indicates our GGA-based auto-tuning algorithm is effective for any input specifications. Considering the comprehensive experiments shown above, we find that our proposed GGA-based auto-tuning algorithm is able to outperform other algorithms in different scenarios.

Experiments on searching efficiency are provided in Fig. 6. 5 experiments for $conv\ 1-1$ in ResNet-18 are conducted for GA tuner, XGBoost tuner and our GGA-based auto-tuning algorithm, respectively. The best throughput it can achieve and the corresponding searching time in seconds are listed for each algorithm. We also show the time acceleration of our GGA-based auto-tuning flow for all the convolutional layers in ResNet-18 and VGG. The results are shown in Fig. 7.

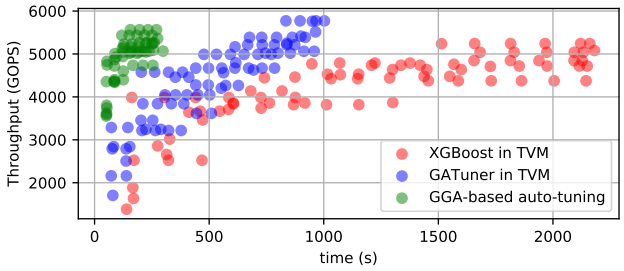

Fig. 6. Search time comparison for the $conv\ 1-1$ in ResNet-18.

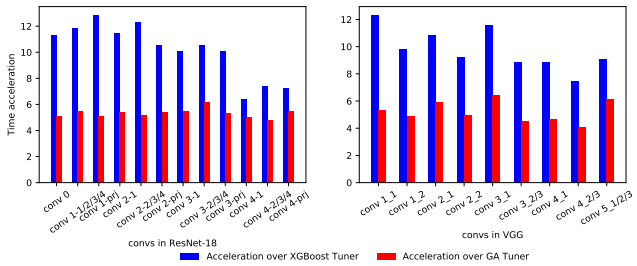

Fig. 7. Time acceleration of GGA based auto-tuning algorithm for ResNet-18 and VGG.

An average of 8.96x and 4.58x acceleration is observed for XGBoost tuner and GA tuner, respectively. Both experiments have shown that our GGA-based auto-tuning algorithm can obtain comparable or better performance efficiently.

The experiments for the whole neural network tuning is illustrated in Table I. XGboost and cuDNN are two baselines for comparison. The result of the state-of-the-art work, RELEASE [11], is also provided. Our proposed GGA-based auto-tuning algorithm can achieve better performance than TVM and cuDNN and comparable performance with RELEASE, while significantly reduce the searching time.

C. Impact of History Database Updating

Notice that previously our proposed tuning algorithm is based on the open-source history, which is provided by TVM, for a fair comparison. However, the open-source history database may become insufficient as more complex models emerge. Our framework solves this problem by updating the history database as mentioned in Fig. 1. In this experiment, we show that the tuning performance can be further improved by updating the history database using the tuning history for the randomly pruned ResNet. Then a new pruned ResNet-18, which is compressed by the RL agent and is independent of the previous history records, is utilized as the benchmark. The experiment is conducted in a layer by layer manner for a detailed comparison and the results are shown in Fig. 8. We observe that the performance is further improved for most of the layers with negligible timing overhead since searching time is mainly depends on the number of schedules running on-board which remains the same here.

We also optimize the ResNet-18 with the updated history database. Latency of 1.46ms can be achieved with a searching time of 45min.

TABLE I
COMPARISON OF DIFFERENT ALGORITHMS FOR RESNET-18

Algorithms	Latency (ms)	Search time
Fallback	1.81	58s
cuDNN 7.3.1 based inference	1.71	None
XGBoost tuner in TVM [2]	1.53	10h 44m
GA tuner in TVM [2]	1.60	6h 52m
RELEASE [11] / TVM	1.04X	4.28X
GGA based auto-tuning	1.48	44min

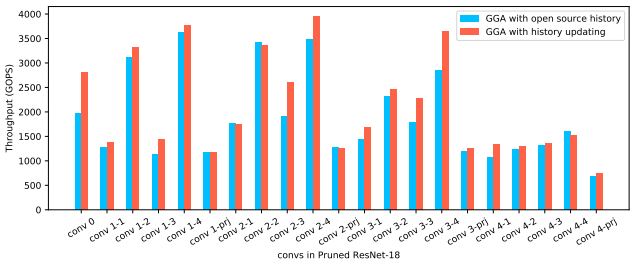

Fig. 8. Performance of GGA-base auto-tuning algorithm with history updating for pruned ResNet-18.

V. CONCLUSION

In this paper, we have proposed a novel framework, which consists of a TL-based auto-tuning algorithm and a GGA-based auto-tuning algorithm, to accelerate the auto tuning process efficiently. The experiments show that our proposed framework can outperform state-of-the-art works with much less searching time. The significant speedup in the auto tuning has potential usage in many areas such as network architecture search and auto pruning since it makes the software-hardware joint optimization possible.

REFERENCES

[1] T. Chen *et al.*, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm SIGKDD*. ACM, 2016.
[2] ——, "TVM: An automated end-to-end optimizing compiler for deep learning," in *13th USENIX Symposium on OSDI 18*, 2018, pp. 578–594.
[3] B. P. Roe *et al.*, "Boosted decision trees as an alternative to artificial neural networks for particle identification," *Nuclear Instruments and Methods in Physics Research Section A*, 2005.
[4] K. He *et al.*, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on CVPR*, 2016.
[5] J. Wu *et al.*, "Pocketflow: An automated framework for compressing and accelerating deep neural networks," 2018.
[6] Y. He *et al.*, "AMC: AutoML for model compression and acceleration on mobile devices," in *Proceedings of the ECCV*, 2018, pp. 784–800.
[7] L. Wang *et al.*, "A unified optimization approach for CNN model inference on integrated GPUs," *arXiv preprint arXiv:1907.02154*, 2019.
[8] C. Li *et al.*, "Optimizing memory efficiency for deep convolutional neural networks on GPUs," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 633–644.
[9] X. Chen *et al.*, "Optimizing memory efficiency for convolution kernels on kepler GPUs," in *2017 54th ACM/EDAC/IEEE DAC*. IEEE.
[10] H. Kim *et al.*, "Performance analysis of CNN frameworks for GPUs," in *2017 ISPASS*. IEEE, 2017, pp. 55–64.
[11] B. H. Ahn *et al.*, "Reinforcement Learning and adaptive sampling for optimized DNN compilation," *arXiv preprint arXiv:1905.12799*, 2019.
[12] Y. Liu *et al.*, "Optimizing CNN model inference on CPUs," 2018.
[13] E. Booth *et al.*, "Hydrologic variability of the cosumnes river floodplain," *San Francisco Estuary and Watershed Science*, 2006.
[14] K. Simonyan *et al.*, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.