HARL: Hierarchical Adaptive Reinforcement Learning Based Auto Scheduler for Neural Networks

Zining Zhang\textsuperscript{1,2} \hspace{1cm} Bingsheng He \hspace{1cm} Zhenjie Zhang
zzn@nus.edu.sg \hspace{1cm} hebs@comp.nus.edu.sg \hspace{1cm} zhenjie.zhang@neuron.sg
\textsuperscript{1}School of Computing, National University of Singapore \hspace{1cm} \textsuperscript{2}NUS Centre for Trusted Internet and Community Singapore

ABSTRACT

To efficiently perform inference with neural networks, the underlying tensor programs require sufficient tuning efforts before being deployed into production environments. Usually, enormous tensor program candidates need to be sufficiently explored to find the one with the best performance. This is necessary to make the neural network products meet the high demand of real-world applications such as natural language processing, auto-driving, etc. Auto-schedulers are being developed to avoid the need for human intervention. However, due to the gigantic search space and lack of intelligent search guidance, current auto-schedulers require hours to days of tuning time to find the best-performing tensor program for the entire neural network.

In this paper, we propose HARL, a reinforcement learning (RL) based auto-scheduler specifically designed for efficient tensor program exploration. HARL uses a hierarchical RL architecture in which learning-based decisions are made at all different levels of search granularity. It also automatically adjusts exploration configurations in real-time for faster performance convergence. As a result, HARL improves the tensor operator performance by 22\% and the search speed by 4.3x compared to the state-of-the-art auto-scheduler. Inference performance and search speed are also significantly improved on end-to-end neural networks.

CCS CONCEPTS

• Software and its engineering → Software performance; • Computer systems organization → Neural networks.

KEYWORDS

neural network optimization, auto tuner, reinforcement learning

ACM Reference Format:

Zining Zhang\textsuperscript{1,2}, Bingsheng He, and Zhenjie Zhang. 2022. HARL: Hierarchical Adaptive Reinforcement Learning Based Auto Scheduler for Neural Networks. In 51st International Conference on Parallel Processing (ICPP ’22), August 29-September 1, 2022, Bordeaux, France. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3545008.3545020

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ICPP ’22, August 29-September 1, 2022, Bordeaux, France © 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9733-9/22/08.
https://doi.org/10.1145/3545008.3545020

1 INTRODUCTION

Deep neural networks (DNNs) with high performance requirements have become ubiquitous in AI applications such as auto-driving [10], real-time speech recognition [15], language translation [11], etc. To achieve the performance requirements under the circumstances of rapidly evolving neural networks and hardware platforms, considerable engineering efforts are frequently required to tune the underlying vendor-provided libraries like oneDNN [3] and cuDNN [9] for neural models. For example, even a single $1024 \times 1024 \times 1024$ matrix multiplication operator contains $\sim 180$ millions of possible implementations for exploration. To liberate engineers and researchers from these labor-intensive tuning tasks, auto-schedulers [5, 8, 28] are developed for automatic tuning of tensor programs. They efficiently search the large search space with the help of statistical optimization methods such as simulated annealing [26] and evolutionary search [17].

However, statistical approaches do not consider the varying performance distributions under different tensor programs, hardware platforms, and parameter sets, leading to suboptimal search efficiency and effectiveness. For example, AutoTVM [8] applies a simulated annealing approach to select the parameter tuning directions throughout the search. Although this method is able to approximate the best-performed tensor program theoretically, simulated annealing assumes heuristic probabilities of the acceptable sets of parameters, which makes the search inefficient.

There are learning-based auto-schedulers proposed to replace the statistical search process. For example, Flextensor [29] and Chameleon [6] utilize the decision power of reinforcement learning (RL). They greatly outperform AutoTVM in both inference performance and search speed. However, a lately developed statistical auto-scheduler Ansor [28] utilizes the more advanced evolutionary search which keep evolving the tensor program candidates in a large search space with a wide coverage on tuning options. Both Flextensor and Chameleon are beaten by Ansor on the search speed and the performance of a wide range of tensor operators, even though Ansor assumes fixed and heuristic probabilities while tuning the parameters. Additionally, since Ansor is capable of optimizing combined tensor operators, it is easy to be applied in end-to-end neural network optimization. We observe that there are following challenges to align RL-based auto-schedulers with Ansor: 1) Lack of support on end-to-end neural networks optimization: reinforcement learning requires grid-like search space which suits the operator-level parameter tuning well, but is unable to solve the complicated combinations of multiple operators. 2) Wasteful
search iterations on poor search paths: reinforcement learning normally requires a termination condition to indicate that the endpoint has been reached. However, in tensor program parameter search, there is no endpoint, i.e. every set of parameters is available for performing the next action. Thus, RL methods tune a whole batch of parameters for a predefined uniform number of steps, without considering the performance potentials case-by-case.

In this paper, we propose HARL, a hierarchical and adaptive RL-based auto-scheduler that eliminates the heuristic and greedy processes in tensor program search. HARL is capable of tuning the whole neural networks by applying a hierarchical RL decision-making process using separate RL models for different levels of search hierarchy: subgraphs, subgraph structures (sketches), and low-level operator parameters. HARL also manages the exploration lengths adaptively by allocating more tuning time slots to potentially well-performing programs in a finer granularity than a whole batch. With the help of both the hierarchical and adaptive learning-based searching schemes, HARL is able to exploit the power of RL algorithms and out-performs Ansor in both the tuning speed and the performance of the final tensor program.

In summary, the contribution of this paper are as follows:

- A learning-based search method for exploring high-performance tensor programs
- A hierarchical search scheme to improve efficiency at all search granularities using RL algorithms
- An adaptive-stopping search process for wisely pruning the less efficient search paths
- A thorough evaluation of the above system, which outperforms prevailing auto-schedulers [5, 8, 28] in both tuning speed and program performance on CPU/GPU platforms.

Experiments are performed on different tensor computations and on different underlying hardware platforms. The results show that for the tensor operators, HARL is able to achieve the same performance up to 4.3x faster than Ansor. Furthermore, the performance of the final tensor program outperforms that of Ansor by 22%. For end-to-end neural networks, the search speed and the inference performance improvement are up to 2.2x and 8%, respectively.

The paper is structured as follows: Section 2 explains the background of current auto-schedulers and our motivations. Section 3 gives an overview of the HARL system. Section 4 and Section 5 elaborate on the details of the main contributions. Section 6 shows the evaluation setup and analysis on different tensor computations and different hardware platforms. Section 7 summarizes this work.

2 BACKGROUND & RELATED WORK

2.1 Problems of Current Auto-Schedulers

Statistical Auto-Schedulers: Statistical auto-schedulers utilize the statistical optimization approaches to explore the tensor programs. Although the search process is heuristic and greedy, they are the prevailing techniques used in research and industry. AutoTVM [8], for example, limits the search space by requiring user-provided templates. It uses simulated annealing [26] to search the optimal parameters under the specified template. However, the predefined templates limit the possibilities of different combinations of multiple operators and thus lead to suboptimal programs. On the other hand, Halide auto-scheduler [5] solves the problem by defining the search space as a sequence of choices of the program construction. To explore this search space, Halide auto-scheduler builds a decision tree that helps sequentially fill the parameters of computational DAG (Directed Acyclic Graph) nodes of the tensor program. The branches of the decision trees are pruned using a cost model that predicts the final performance on the partially constructed programs. This process is applied with general rules and therefore does not need any template. However, the performance prediction on incomplete programs leads to inaccurate predictions. Ansor [28] eliminates the drawbacks of the previous two methods: it is template-free and predicts the schedule performance based on complete programs. It generates high-level program sketches based on the abstract definition of the tensor computation, and then explores and fills the low-level parameters using evolutionary search [17]. The sketches generated by Ansor eliminate the need for templates. The low-level parameters are fully filled during evolutionary search, thus the performance estimations are based on complete programs. With these fine properties and the effectiveness brought by evolutionary search, Ansor is the state-of-the-art auto-scheduler.

RL-based Auto-Schedulers: Flextensor [29] and Chameleon [6] improve the search efficiency using RL algorithms. Flextensor is able to cover the cases of different tensor operators with the help of a general template, but fails to handle the optimization of end-to-end neural networks. Chameleon is another auto-tuner which improves the performance of AutoTVM by replacing the simulated annealing with a reinforcement learning model. It also adaptively manages the initial schedule sampling, which is originally uniform sampling in AutoTVM. However, Chameleon follows the template-based optimization process of AutoTVM, thus the performance of the outcome depends on the quality of the user-provided templates. Note that both Flextensor and Chameleon ignore the importance of the time-allocation for different tuning paths, and thus are outperformed by Ansor. Bananor [13] is an incremental improvement over Ansor which focuses only on modifying the subgraph and sketch selections as bandit-based.

2.2 Problem Formulation & Motivation

For a search-based tensor program scheduler, the search problem can be treated as finding the schedule $S$ with the best inference performance which is measured by the execution time $f(S)$. Since neural networks are mostly modularized, the computational graph of the network is first split into $N$ subgraphs that are sequentially executed. The optimization target is thus approximated by the estimation $f(S) \approx \sum_{n=1}^{N} w_n \times g_n$, where $w_n$ and $g_n$ are the number of appearances and the execution time of the subgraph $n$, respectively. At each tuning step $t$, the auto-scheduler chooses a subgraph $n$ with probability $\pi_t(n)$ for optimization. Under this subgraph selection distribution, the total tuning time-allocation for each subgraph $n$ is denoted as $T^n_t$, and $\sum_{n=1}^{N} T^n_t = T$, where $T$ is the total time-slots for the neural network optimization.

Subgraph $n$ is assumed to contain one or more tensor operators, and thus has different structures (or sketches) on operator combinations. Suppose that there are a total of $U_n$ sketches for the subgraph $n$. Sketch $u$ is selected as the skeleton of the subsequent parameter
We observe that the top 3 subgraphs have no improvement. Only 1% of the total trials are spent on this tiny improvement. Over 35% of the total trials are performed on Flextensor which uses fixed-length search. The histogram shows that most of the search paths find the best parameters within the first 40% of the total search steps. This verifies the search inefficiency mentioned above.

With the above two observations, a search method with adaptive \( \pi_t(n), \pi_t(u), \pi_t^n(u_t|s_t-1), \{T^n_i\}_{1 \leq i \leq I} \) distributions is required for a more efficient neural network optimization process. With this motivation, HARL is developed. As shown in Table 1, HARL applies

| Subgraph Selection | Sketch Selection | Schedule Selection \( \pi_t^n(u_t|s_t|s_{t-1}) \) | Schedule Tracks Time-Allocation \( \{T^n_i\}_{1 \leq i \leq I} \) |
|-------------------|-----------------|---------------------------------|---------------------------------|
| Ansor [28]        | Greedy Selection | Uniform Distribution            | Uniform Distribution            |
| Flextensor [29]   | Not Supported   | Uniform Distribution            | Greedy Allocation               |
| HARL              | MAB RL          | RL Agent                        | Uniform Allocation             |
|                   |                 | RL Actor Network                | Estimation on Future Performance|

**Table 1: System Comparison: Ansor vs. Flextensor vs. HARL.** HARL uses either learning-based or adaptive methods on all the 4 search processes. MAB stands for multi-armed bandit [27].

![Diagram](image-url)  

(a) Time-allocations on top-5 most time-consuming subgraphs of BERT. The sketched bars are the number of trials allocated for last 1% improvement.

(b) Performance improvement distribution. **Improvement Ratio** is the performance improvement from original schedule to a next schedule relative to the performance of the original schedule.

(c) Histogram of search path efficiency on Flextensor. X-axis represents the position of the best-performed schedule on the search path divided by the path length. Y-axis is the frequency of these relative positions collected during auto-tuning.

**Figure 1: Observations made on current auto-schedulers.**

Observation 2: Uniform schedule track allocations \( \{T^n_i\}_{1 \leq i \leq I} \) result in high ratio of useless search steps. When using fixed-length schedule search on a batch of parameters, some search paths may already reach the optimal points at the early steps, thus the following steps do not contribute. In Figure 1(c), various GEMM operations are performed on Flextensor which uses fixed-length search. The histogram shows that most of the search paths find the best parameters within the first 40% of the total search steps. This verifies the search inefficiency mentioned above.

With the above two observations, a search method with adaptive \( \pi_t(n), \pi_t(u), \pi_t^n(u_t|s_t|s_{t-1}), \{T^n_i\}_{1 \leq i \leq I} \) distributions is required for a more efficient neural network optimization process. With this motivation, HARL is developed. As shown in Table 1, HARL applies
learning-based search algorithms that adaptively learns the time-varying sampling distribution of subgraphs and their sketches. It explores the parameter search space with an RL agent. The optimization time slots for subgraphs and schedule tracks are adaptively controlled for better search efficiency. Compared with HARL, Ansor uses greedy or heuristic distributions in these optimization processes. While Flextensor applies RL-based schedule selection process, it does not support optimizing combinations of tensor operators with the help of subgraph and sketch selections, and it simply uses fixed-length uniform allocations on different schedule tracks.

3 SYSTEM OVERVIEW

3.1 Search Hierarchy

HARL adopts a hierarchical search scheme as described in Section 2.2. First, the end-to-end neural network optimization problem is divided into subgraph optimization problems. Each subgraph \( n \) may have multiple computational structures or sketches. The sketches of the subgraph are automatically generated based on the computational definition of the subgraph. The generation rules are the same as those of Ansor [28], and are shown in Table 2. Tiling determines the loop structures (tiling sizes and loop orders) of the tensor programs. Inline configures the locations of the computations. Cache Write adds an additional operation that caches the writes [24] is to apply parallelism on the reduction dimensions. Cache Write adds an additional operation that caches the writes to the output buffer. Applying these rules differently results in different sketches of the optimizing tensor program.

Next, HARL tunes the parameters on a specific sketch \( u \) of the target subgraph \( n \). There are a couple of tuning knobs for the auto-scheduler to modify to transit to the next state. The tuning knobs, which are also known as modification types contain computation-location modification, tile-size modification, parallelism modification, and annotation modification. Section 4.2 describes all the modification types used in HARL in details. Then, the auto-scheduler chooses a specific set of new parameters of the selected modification type as instructed by the search algorithm. These new parameters modified from schedule \( s_{t-1} \) are applied to generate the new schedule \( s_t \) for the next exploration step.

In summary, combined with the problem definition specified in Section 2.2, there are four different levels of search hierarchy that are controlled by the search algorithm: subgraph (a.k.a. task) \( n \) with sampling distribution \( \pi_t(n) \), sketch \( u \) with sampling distribution \( \pi_t^n(u) \), modification type, and parameter tuning transition distribution \( \pi_t^{n,u}(a_t^n|s_t^{n-1}) \).

3.2 HARL Structure

HARL applies RL modeling to all levels of the search hierarchy. Figure 2 shows the overview of the HARL system. It applies separate RL models for different search granularities.

For high-level subgraph and sketch selections, HARL assumes non-stationary distribution of the rewards. As the search goes on, if a subgraph is optimized for most of the time, it is unlikely to keep the same distribution of improvements. This time-varying reward distribution indicates a time-varying subgraph sampling distribution \( \pi_t(n) \). The sampling distribution affects the subgraph time-allocations \( \{T^1, T^2, ..., T^N\} \). For the sketch selection distribution, it also changes over time. In this case, both the subgraph selection and sketch selection phases are modeled as a non-stationary multi-armed bandit problem (MAB) [27]. This is different from Ansor’s approach, where the subgraph sampling distributions \( \pi_t(n) \) is modeled as a greedy process with deterministic selections, and \( \pi_t^u(u) \) is treated as time-independent uniform distribution. More details are explained in Section 4.1.

For the low-level modification types and parameter tuning, the MAB model is no longer applicable, since there are an extraordinary number of parameters for each sketch. In this case, the search space of parameters can be constructed as connected graphs as shown in the RL environment in Figure 2. Each step of parameter tuning is thus modeled as an edge connecting one schedule setting to another. This edge is treated as an action generated by a Markov decision process (MDP) \( \pi_t^{n,u}(s_t^n|s_{t-1}) \), thus the parameter search problem is solvable by RL algorithms like DQN [19], actor-critic method [18], etc. The construction of this search space is explained in more details in Section 4.2.

Compared with the ordinary actor-critic RL algorithm, there are two additional modules in HARL: a cost model and an adaptive stopping module. To avoid a large number of time-consuming measurements, we use a light-weight cost model to predict the actual performance of each schedule. This cost model prunes the schedules with low prediction scores and learns on the fly from the actual measurements. It is also used as the reward function in the RL model. The adaptive stopping module is used to adaptively allocates the times slots \( \{T^1, T^2, ..., T^N\} \) for schedule tracks \( i \in [1, I] \). This track-wise fine-grained length-control module helps to explore more on schedules with better potentials. The details are presented in Section 5.

4 HIERARCHICAL SEARCHING

4.1 Bandit-based RL for High Level Decisions

As described in Section 3, the decision-making process of schedule search goes through four levels of hierarchy: subgraph selection, sketch selection, modification type selection and the modification parameter selection. We observe that high-level decisions usually have a fixed and limited number of choices. For example, in a BERT [11] model, the number of distinct subgraphs is 10, and that of ResNet-50 [16] is 24. For a matrix multiplication subgraph, the number of sketches is 3. On the other hand, the optimization outcome for each
step of these subgraph/sketch selections is with a non-deterministic performance improvement.

Based on these observations, the subgraph selection and the sketch selection process match the MAB problem [27]. Additionally, the two MAB problems are non-stationary as the performance of different subgraphs and sketches continues to improve over the course of optimization. To describe our solution to the MAB problems, the following terms are introduced: In order to trace the course of optimization. To describe our solution to the MAB problems, this gradient estimation formula predicts the benefit from optimizing the subgraph based on both the performance improvement from the previous steps (the first term), as well as the execution-time contribution and the similar subgraphs (the second term). Please refer to the original paper [28] for details. Finally, the $Q_t(\tau, a)$ for subgraph selection is defined as:

$$Q^N_t(\tau, a) = \frac{1}{N_t(\tau, a)} \sum_{t' = t - \tau + 1}^{t} R_{t'}(a) 1_{\{O^N_{t'} = a\}}$$

By utilizing the SW-UCB for solving the non-stationary MAB subgraph/sketch selection problem, the search process starts to explore the search space instead of greedily selecting the action solely based on the previous data or estimations. The exploration importance is controlled by the constant $c$ in Equation 1. Its following term gives more opportunities to the actions that are seldomly explored in the previous rounds.

### 4.2 Parameter Search Space

Before introducing the low-level parameter modification selection process, this section focuses on the construction of its search space. Taking $1024 \times 1024 \times 1024$ GEMM as an example, with 4 tiling levels, there are 286 tiling choices for each dimension\(^1\). In the settings where 2 loop unrolling depths are considered and at most 4 levels of spatial loops are fused for thread-level parallelism, there are $286 \times 286 \times 286 \times 2 \times 4 \approx 180$ millions of configurations. In HARL, these configurations are not fully connected, i.e. the RL agent modifies one configuration (or state in the RL context) to construct a limited set of nearby configurations, in order to make the gradients of the learning process steady. The new configuration construction process is an integration of different modification types, which are also known as action subspaces in the RL context. All of them are described in Table 3, and further explained in the following:

**Tiling Modification:** Computation-intensive operators like matrix multiplications require appropriate tile size for the best cache efficiency. The optimal tile sizes are difficult to be determined by heuristics and are therefore tuned using auto-schedulers. Suppose

\(^1\)since $1024 = 2^{10}$, the loop tiling can be treated as allocating a series of 10 factors 2 to 4 groups, which is equivalent to find all possible 3-border combinations from 11 positions with repetitions, where $\binom{11+3-1}{3} = 286.$
that there are num_iters tiling loops for the considered tensor operator, the actor network consumes the schedule of the current step and outputs an action in the form of a pair of iteration indices (i, j), where i, j ∈ [0, num_iters] and i ≠ j. i represents the source index, where a minimal factor (greater than 1) will be divided from and multiplied to the target index j. An additional dummy action (−, −) is added to the set of directions to represent that no modification is performed on the current tile sizes. This process is able to cover the whole search space of tile sizes.

**Compute-At Modification:** Some operations in the subgraph can be computed inline with one of their data producers to improve the data reuse. However, it is non-trivial to decide which ancestor of all the data producers and which inner loop of the data producer should the operation be computed at. To implement the compute-at modification for the RL agent, HARL extracts all possible compute-at position pairs (stage_id, iter_id), where stage_id means the data producer index of the operator, and iter_id represents the loop index inside the specified data producer. The compute-at pairs are sorted in sequential order into a candidate list. The action a is defined to be in the set {−1, 0, 1}, which means how the compute-at position index should be changed in the candidate list.

**Parallel-Loops Modification:** The number of parallelism of a tiled operation can be modified by changing the number of fused outer loops for parallel executions. In this case, the action a is defined in the set {−1, 0, 1} for indicating how the number of fused loops should be changed in this modification.

**Auto-Unroll Modification:** The depths of the loop auto-unroll can be controlled by the pragma unroll. A list of unroll factors is predefined. The action a ∈ {−1, 0, 1} indicates the change of index to the next unroll factor in this list.

### 4.3 Actor-Critic Based RL for Parameter Modifications

As described in Section 3, after confirming the optimizing subgraph n and its sketch u with the help of the MAB-based RL, actor-critic based RL parameter search is conducted in HARL.

In details, actor-critic method [18] is applied in HARL. As the name suggests, there is an actor network and a critic network. The actor network plays a role as an agent. It takes the current state (i.e. schedule) as an observation and chooses a set of actions (i.e. parameter modifications) to apply to the state for generating the next state as described in Section 4.2. The critic network evaluates the action taken by the actor network and participates in its training process. The optimization of the actor-critic network follows the policy gradient theorem [25]. The gradient can be expressed as:

$$
\nabla_\theta J(\theta) = \mathbb{E}[\Sigma_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(s_t, a_t) A_{\pi_\theta}(s_t, a_t)]
$$

In the formula, t represents the current step and T represents the total number of steps. s_t and a_t are the current state and the chosen action respectively. A_{\pi_\theta} represents the current policy, where the probability of choosing action a_t given state s_t is its output. A_{\pi_\theta} is the advantage function (a.k.a., the temporal difference error).

$$
A_{\pi_\theta} = r(s_t, a_t) + \gamma V_{\pi_\theta}(s_{t+1}) - V_{\pi_\theta}(s_t)
$$

where the function r(·, ·) is the reward function, γ is the discount factor, and V_{\pi_\theta}(·) is the value network.

In addition to the actor-critic network, the search process utilizes a light-weight cost model as the reward function r(·, ·) to save the time cost by the actual measurements. There is also an adaptive stopping module which adaptively controls the search process for better search efficiency. To make the interactions between actor-critic network, adaptive stopping module, cost model and measurer clearer, the formalized algorithm of parameter search is shown in Algorithm 1, and an example of parameter search workflow is demonstrated in Figure 3. The search process is composed of two phases: parameter modification phase and top-k selection phase. Following are the detailed descriptions about the two phases.

**Parameter Modification Phase:** This phase is shown as PHASE 1 at the top of Figure 3. It maps to an 'episode' in the RL language. At the beginning of this phase, a total of I initial schedules or states \{s_{i, 0}^n\}_{i \in I} are first sampled by randomly filling the sketch u of the subgraph n selected from the process described in Section 4.1. Each of these initial states undergoes the parameter modifications independently, so we call the path starting from each initial schedule as a schedule track. In Figure 3, I = 4, there are thus 4 initial schedules (shown as gray parallelograms) leading 4 schedule tracks: T1~T4. The pseudo-code is shown on lines 3-5 in Algorithm 1.

After passing the states to the actor network, the potentially most beneficial parameter modification for each modification type is returned and performed on the current states by the agent. Since each modification type has a dummy action that keeps the input state unchanged, the modification type selection is implicitly performed by the actor network at each step. Next, the newly generated schedules are passed to the critic network for evaluation using the advantage function A_{\pi_\theta} as defined in Equation 6. The advantage values are passed to the adaptive stopping module which

| Table 3: Types of Parameter Modifications |
|-------------------------------------------|
| **Direction** | **Description** |
| Tiling Modification | \{(i, j) | i, j ∈ [0, num_iters], i ≠ j\} | Divide the smallest factor from iteration i and multiply to iteration j. |
| Compute-At Modification | \{-1, 0, 1\} | Move backward, stay or move forward in the list of candidates of the computation location. |
| Parallel-Loops Modification | \{-1, 0, 1\} | Move backward, stay or move forward in the list of auto-unroll depths. |

**Parallel-Loops Modification:** The number of parallelism of a tiled operation can be modified by changing the number of fused outer loops for parallel executions. In this case, the action a is defined in the set {−1, 0, 1} for indicating how the number of fused loops should be changed in this modification.

**Auto-Unroll Modification:** The depths of the loop auto-unroll can be controlled by the pragma unroll. A list of unroll factors is predefined. The action a ∈ {−1, 0, 1} indicates the change of index to the next unroll factor in this list.
determines whether to stop searching on a track-wise granularity. This module controls the time-allocation \( (T_i^n)_{1 \leq i < 1} \) for each schedule track \( i \). The sketched parallelograms shown in Figure 3 represents the adaptively stopped schedule tracks. The details of adaptive stopping are shown in Section 5. Meanwhile, the rewards \( r_s(s^n_{i,t}, s^n_{i,t-1}) = \frac{C(s^n_{i,t}) - C(s^n_{i,t-1})}{C(s^n_{i,t-1})} \) are estimated using the cost model \( C \). The episode goes on as long as there are no less than a minimal number of tracks left. The above procedure is shown on lines 6-19 in Algorithm 1. The states, actions, advantage values and rewards are recorded and used for training the actor and critic network for every specific number of steps.

**Top-K Selection Phase:** This phase is shown as PHASE 2 on the bottom of Figure 3. After an episode of parameter modifications finished, all traversed schedules and their predicted scores are recorded. In the figure, the gathered parallelograms and diamonds are the collections of visited schedules and their corresponding scores predicted by the cost model respectively. Next, in order to save the total tuning time from an excessive number of measurements, only the top-K schedules with the best scores are selected for measuring on the target hardware platform. The measurement results are used to update the cost model \( C \) on the fly. This top-K selection phase is shown on lines 20-22 in Algorithm 1.

5 APAPTIVE-STOPPING SEARCH

Unlike RL scenarios such as game simulations, where an episode ends in specific states, schedule search does not have a well-defined ending state. In other words, each point accessed during one episode of schedule explorations can be treated as an endpoint and should participate in the top-K selection process at the end of the episode. In this case, the length control of the schedule tracks is non-trivial.

**Algorithm 1: Parameter Search Algorithm**

```
function parameterSearch(u)
    Input: Selected Sketch \( u \)
    Output: Chosen Schedules for Measurements \( \hat{S} \), Measured Performance \( P \)

    Load cost model \( C \), actor network \( \pi_\theta \) and value network \( V_\pi_\theta \)
    Initialize number of tracks each round \( p \), minimal number of tracks \( \hat{p} \) RL training interval \( T_{rl} \)
    Initialize step \( t \leftarrow 0 \)
    Initialize schedules of current round \( S \) is sampled from parameter space of sketch \( u \) with size \( p \)

    while \( |S| \geq \hat{p} \) do
        Select the set of modification actions \( M \) for schedules \( S \) using policy \( \pi_\theta(M|S) \)
        Apply each sub-actions in \( M \) to the states \( S \), and generates new states \( S' \)
        Get reward \( R = \frac{C(S') - C(S)}{C(S)} \)
        Calculate \( Y \) using \( A_{\pi_\theta}(S', M, S) \) from Equation 6
        Pass \( Y \) to adaptive-stopping module to test termination for each schedule.
        Record \( (S, M, S', R, Y) \) in to the replay buffer \( B \)
        Record \( S' \) into the heap \( H \) ordered by \( C(S') \).
        if \( t \mod T_{rl} = 0 \) then
            Sample a mini-batch from the replay buffer \( B \)
            Train actor network \( \pi_\theta \) using gradient \( \nabla_\theta J(\theta) \) from Equation 5
            Train critic network \( V_\pi_\theta \) using MSE of the advantage \( Y \).
            \( \hat{S} \leftarrow S' \)
            \( t \leftarrow t + 1 \)
        Select top-K states \( \hat{S} \) from the heap \( H \).
        Measure \( K \) states and record the performance \( (\hat{S}, P) \)
    Train the cost model \( C(\cdot) \) with \( (\hat{S}, P) \)

    return \( \hat{S}, P \)
```

If the length of the schedule track is not enough, the performance of the modified schedules may not be fully exploited. On the other hand, if the length is excessively large, it would affect the total tuning time, since all tracks of the schedules share the same length no matter if they already hit the maximal performance at some early steps. Instead of fixing the lengths of all the schedule tracks, HARL uses an adaptive method to determine the lengths on a track-wise granularity. Meanwhile, it maintains a similar number of candidates as the fixed-length search for the final top-K selection process.

Figure 4 elaborates on the differences between the fixed-length search and the proposed adaptive-stopping search. Each box in the figure represents a visited schedule and the higher the number inside indicates the better performance. There are 6 tracks named \( T_1 \sim T_6 \) in this example. In the fixed-length search, as described on the left side of Figure 4, the algorithm searches each of the schedule
Figure 4: Demo the difference between the fixed-length search and adaptive-stopping search. Each box represents a schedule. Larger number means better performance. Sketched boxes means the eliminated schedules.

6 EVALUATION
6.1 Setup
To evaluate the effectiveness of HARL on tensor program optimization, a set of experiments consisting of tensor operator optimization and neural network end-to-end optimization is conducted. The previous study [28] shows that Ansor outperforms Pytorch [21], AutoTVM [8], Halide auto-scheduler [5], and Flextensor [29] on different kinds of tensor programs and hardware platforms, thus we use Ansor (TVM v0.8.0) [1] as our only baseline. For the actor-critic RL model, we use proximal policy optimization (PPO) [23] algorithm. The PPO implementation is adopted from the online source code [4]. For the cost model we use Xgboost [7] with the same parameters as used in Ansor. More implementation details are shown in Appendix A.1. The optimized tensor programs are evaluated on CPU (Intel Xeon 6226R with 32 cores and 2.9GHz) and GPU (Nvidia GeForce RTX 3090). GPU is also used for the RL-based search process.

In the following, we present workloads, settings, metrics and results of two sets of experiments including tensor operator optimization and neural network optimization.

6.2 Tensor Operator Optimization Comparison

Workloads: For the tensor operator benchmark, we use GEneral Matrix Multiplication (GEMM) [2] with 3 size categories from small to large: GEMM-S, GEMM-M and GEMM-L. We also uses convolutions: 1D convolution (C1D), 2D convolution (C2D), 3D convolution (C3D), as well as transposed 2D convolution (T2D) [12]. The parameters of convolutions are the same as those used in Ansor [28]. Each operator is tested with 4 different sets of parameters. All the parameters of the tested operators are shown in Appendix A.3. These tests are conducted on CPU with 1 and 16 batch sizes.

Search Settings: All tensor operator optimizations are conducted with 1000 measurement trials on both Ansor and HARL for fair comparisons. Ansor and HARL are setup with same number of measurement candidates in each round of exploration for verifying the efficiency of proposed adaptive-stopping search. For adaptive stopping, we set the window size $\lambda = 20$, and the elimination ratio $\rho = 0.5$. More details on the parameter selections are shown in Appendix A.4. For the learning process of actor-critic network, the learning rates are $3\times10^{-4}$ and $1\times10^{-3}$ for actor and critic networks respectively. The discount factor $\gamma$ is set to 0.9. In the critic network, the weight of the MSE loss is 0.5 and that of distribution entropy is 0.01. For the sketch MAB, the constant $c$ used in SW-UCB defined in Equation 1 is set to 0.25 and the window size is 256.

Figure 5: The normalized performance (inverse of execution time) of Ansor and HARL on different tensor operators.

Metrics: To verify the effectiveness of HARL on the search speed of tensor programs and the performance of the output program,
two metrics are used in the experiments. Performance is the inverted execution time of the output programs produced by the auto-schedulers. Search time is the time consumed by the auto-scheduler to find the tensor program with the performance no worse than the final output program of the baseline. To make the comparison between baselines and HARL clearer, the Performance and Search time metrics are normalized to values in the range [0, 1].

Results: As shown in Figure 6, the output schedules of HARL outperform those of Ansor on every tensor operations by 6% ~ 22%. The search time improvement is more significant, as shown in Figure 6. On average, to achieve the performance no worse than the best schedule achieved by Ansor, HARL uses 63% ~ 23% search time. The GEMM-L operation has the most outstanding performance improvement. This is due to their larger data dimensions, thus they have larger tile size configuration spaces. The GEMM-L operations with large search space bring harder problems to auto-schedulers, and HARL is able to make profound performance increase with the help of the adaptive length control and hierarchical learning-based search method.

Ablation Study: Figure 7(a) shows the records when optimizing one GEMM-L operation on Ansor, HARL with fixed-length episodes, and HARL with adaptive-stopping modules. HARL is able to outperform Ansor at early steps. “Hierarchical-RL” in the figure means the HARL implementation with fixed-length search. From the figure, the adaptive-stopping module is able to significantly improve the search efficiency of HARL after the optimization goes for a few hundreds of rounds. This is because adaptive-stopping relies on the success of the agent’s movement only when the critical step and the cost model are sufficiently trained. Without the help of adaptive-stopping, Hierarchical-RL is still able to outperform Ansor. This shows the effectiveness of non-stationary MAB modeling on sketch selection.

To further examine the search efficiency of the adaptive-stopping module, we record the step where the highest performance score is reached in each schedule track during the search process. We call the top-scored step as the critical step. All the steps after this critical step do not find a better schedule, thus they are called wasted steps. The histograms of the critical steps for the two search methods are shown in Figure 7(b). In adaptive-stopping, most of the critical steps appears in the last 10% steps of their schedule tracks, which indicates less than 10% wasted steps. Additionally, 16% of these low-waste schedule tracks are from the longest tracks of adaptive-stopping, which is not shown in the figure. The smaller number of wasted steps in adaptive-stopping leads to a more efficient search process.

6.3 Neural Network Optimization Comparison

Workloads: For end-to-end neural network optimizations, we test Ansor and HARL on ResNet50 [16], MobileNet-V2 [22], and BERT [11]. They are commonly used DNN models utilized in computer vision (CV), and natural language processing (NLP). They are tested on Intel CPU and Nvidia GPU with batch size 1 and 16.

Settings: The number of trials measured is 12,000 for BERT, 22,000 for ResNet50, and 16,000 for MobileNet-V2. For the
actor-critic network and MAB settings, the parameters used in end-to-end benchmarks are the same as described in Section 6.2.

Results: Figure 8 shows the performance comparisons on different neural networks and hardware platforms. Figure 9 shows the comparisons on the search time. HARL improves the performance of the outcome by 8% and 9% for CPU and GPU respectively. The search time reduction is up to 55% and 51%.

Table 4 further dives into the execution time contributions of different subgraphs. In the table, the first column contains the 10 subgraphs of the BERT model. The second column is the percentage of each subgraph’s contribution to the total execution time of HARL’s output program. The third column shows the speedup of HARL relative to Anser on each subgraph. The first 5 subgraphs are with large number of floating-point operations and contribute to 87% of the total execution time. Their speedup matches that of the individual tensor operators as evaluated in Section 6.2. For the remaining subgraphs, they are with smaller execution time contributions. For example, the Batch GEMMs are with magnitudes of smaller number of floating operations (\(~1/3000\)) than that of GEMM operations. Taken together, they take 13% of the total execution time, and HARL has similar performance as Anser on them. Note that, the subgraphs communication overheads affects the end-to-end performance as shown in comparison of Estimated HARL and the Measured HARL in the table. Nonetheless, HARL still achieves the significant performance improvement and search time reduction with the help of hierarchical RL and adaptive-stopping technique.

Table 4: Ablation Study for BERT on CPU

| Subgraph     | Execution Time Contribution | SpeedUp |
|--------------|-----------------------------|---------|
| GEMM-I       | 25.3%                       | 1.15x   |
| GEMM-IV      | 25.1%                       | 1.10x   |
| GEMM-III     | 20.5%                       | 1.06x   |
| GEMM-II      | 11.4%                       | 1.09x   |
| Softmax      | 5.0%                        | 1.14x   |
| Batch_GEMM-II| 4.0%                        | 0.98x   |
| Batch_GEMM-I | 3.8%                        | 1.01x   |
| Element-wise-II | 2.4%                     | 0.95x   |
| Element-wise-I  | 2.2%                      | 1.01x   |
| GEMM+Tanh   | 0.4%                        | 1.02x   |
| Estimated HARL (sum) | 100%                  | 1.10x   |
| Measured HARL | -                          | 1.08x   |
| Measured HARL (w/o subgraph MAB) | -                  | 1.06x   |

To analyze the effectiveness of subgraph MAB selection that is only applicable in the end-to-end neural network optimization, the test result on HARL without subgraph MAB is presented at the bottom of Table 4. It shows that without the subgraph MAB selection process, HARL has a worse performance due to the greedy subgraph time-allocations similar as Anser.

To better analyze this phenomenon, we list the subgraph trial-allocation details \(\{T_n\}_{1 \leq n < N}\) in Figure 10. The experiments are...
performed on HARL and HARL without subgraph MAB modeling. The trial-allocations are split into two parts. The ‘> Ansor’ part means the trials allocated at the time the system reaches the best schedule of Ansor. The ‘> Ansor’ part represents the remaining trials allocated after reaching Ansor’s best performance. When looking at the total allocated trials, all the time-consuming GEMM subgraphs in HARL with MAB (dark-red bars) have fewer allocations than those in HARL without subgraph MAB. Thus, the subgraph MAB algorithm successfully avoids the over-allocation on subgraphs with high early-trials improvements. Instead, the trials allocated for the SOFTMAX subgraph are improved, which leads to a high speed-up (1.14x) as shown in Table 4. On the other hand, when looking at the part of the trials allocated when the systems just reach Ansor’s best performance, i.e., the bars that are not sketched, the similar improvement is observed. When using subgraph MAB method, the trials allocated for GEMMs with lower total improvements GEMM-II and GEMM-III are fewer than those in the greedy subgraph selection approach. Instead, more trials are allocated for GEMM-IV which has more potential for improvement, as indicated in Table 4.

7 CONCLUSION

This work proposes HARL, a hierarchical and adaptive reinforcement learning-based auto-scheduler for tensor programs. It is template-free with improved performance on the generated codes and significantly faster tuning process. The effectiveness of HARL is verified on different kinds of tensor operators and DNNs, under CPU and GPU hardware platforms. We believe that, HARL is an innovation on how to utilize the advanced reinforcement learning based decision process to facilitate the time-consuming tensor code optimization.

ACKNOWLEDGMENTS

This work is in part supported by Centre for Trusted Internet and Community, National University of Singapore.

REFERENCES

1. 2022. Auto-scheduler in TVM v0.8.0. https://github.com/apache/tvm/tree/v0.8.0/python/tvm/auto_scheduler. (Accessed on 03/29/2022).
2. 2022. GEMM - Wolfram Language Documentation. https://reference.wolfram.com/language/LowLevelLinearAlgebra/ref/GEMM.html. (Accessed on 03/29/2022).
3. 2022. onepapi/src/oneDNN: oneAPI Deep Neural Network Library (oneDNN). https://github.com/oneapi-src/oneDNN. (Accessed on 03/29/2022).
4. 2022. PPO-PyTorch. https://github.com/nikhilbarhate99/PPO-PyTorch. (Accessed on 03/29/2022).
5. Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tru-Mao Li, Michael Gharchi, Benoit Steiner, Steven Johnson, Kavyon Fatahalian, Frédéric Du rand, et al. 2019. Learning to optimize halide with tree search and random programs. ACM Transactions on Graphics (TOG) 38, 4 (2019), 1–12.
6. Byung Hoon Ahn, Prannoy Pilligundla, Amir Yazdanbakhsh, and Hadi Esmaeilzadeh. 2020. Chameleon: Adaptive code optimization for expedited deep neural network compilation. arXiv preprint arXiv:2001.08743 (2020).
7. Tianqi Chen and Carlos Guerin. 2016. Xgboost: A scalable tree boosting system. In Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining. 785–794.
8. Tianqi Chen, Lianghui Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guerin, and Arvind Krishnamurthy. 2018. Learning to optimize tensor programs. Advances in Neural Information Processing Systems 31 (2018).
9. Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. arXiv preprint arXiv:1410.0759 (2014).
10. Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. 2016. The cityscapes dataset for semantic urban scene understanding. In Proceedings of the IEEE conference on computer vision and pattern recognition. 3213–3223.
11. Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018).
12. Vincent Dumoulin and Francesco Visin. 2016. A guide to convolution arithmetic for deep learning. arXiv preprint arXiv:1603.07285 (2016).
13. Chao Gao, Tong Mo, Taylor Zowtuk, Tanvir Sajjad, Laiyuan Gong, Hansuan Chen, Shangjing Jui, and Wei Lu. 2021. Bansor: Improving Tensor Program Auto-Scheduling with Bandit Based Reinforcement Learning. In 2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI). IEEE, 273–278.
14. Aurélien Girardier and Eric Moulines. 2008. On upper-confidence bound policies for non-stationary bandit problems. arXiv preprint arXiv:0805.3145 (2008).
15. Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Eric Elsen, Ryan Prenger, Sanjeev Salatheh, Shubho Sengupta, Adam Coates, et al. 2014. Deep speech: Scaling up end-to-end speech recognition. arXiv preprint arXiv:1412.5367 (2014).
16. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition. 770–778.
17. Melanie Mitchell. 1998. An introduction to genetic algorithms. MIT press.
18. Volodymyr Mnih, Adria Fagnanchesch Badía, Hubert Ludi, Alex Graves, Timothy Lillcrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous methods for deep reinforcement learning. In International conference on machine learning. PMLR, 1928–1937.
19. Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602 (2013).
20. Martijn van Otterlo and Marco Wiering. 2012. Reinforcement learning and markov decision processes. In Reinforcement learning. Springer, 3–42.
21. Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems 32 (2019).
22. Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In Proceedings of the IEEE conference on computer vision and pattern recognition. 4510–4520.
23. John Schulman, Filip Wolski, Prafulla Dhariwal, Alex Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347 (2017).
24. Patrizia Siriana, Andrew Adams, and Shoab Kami. 2017. Parallel associative reductions in halide. In 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE, 281–291.
25. Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. 1999. Policy gradient methods for reinforcement learning with function approximation. Advances in neural information processing systems 12 (1999).
26. Peter JM Van Laarhoven and Emile HL Aarts. 2012. Reinforcement learning and markov decision processes. In Markov decision processes. In Advances in neural information processing systems 6. 770–778.
27. Zhao Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Zweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. 2016. Deep speech: Scaling up end-to-end speech recognition. arXiv preprint arXiv:1412.5367 (2014).
28. Zhemin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Hai-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating [High-Performance] Tensor Programs for Deep Learning. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI). 863–879.
29. Sizhe Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiven Sheng. 2020. Flortexer: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. 859–873.

A ARTIFACT DESCRIPTIONS

A.1 Model Implementation Details

The PPO implementation is adopted from the online source code [4]. Modifications are made to the actor network in order to produce actions probabilities for multiple parameter modification subspaces as explained in Section 4.2. The number of actions for each subspace corresponds to the number of directions as shown in Table 3. More specifically, the number of actions is num_iters × num_iters + 1 (1 is for the dummy action) for tile size modification and 3 for the other
### Table 5: Model Parameters

| Default Value | Description                              |
|---------------|------------------------------------------|
| $\lambda$     | 20 adaptive-stopping window size          |
| $\rho$        | 0.5 adaptive-stopping elimination rate    |
| $\hat{p}$     | 64 adaptive-stopping minimum number of remaining tracks |
| $\eta_a$      | 3e-4 learning rate of actor network       |
| $\eta_c$      | 1e-3 learning rate of critic network      |
| $T_{rl}$      | 2 actor-critic network training interval  |
| $\gamma$      | 0.9 discount factor in Equation 6         |
| $w_{MSE}$     | 0.5 critic network MSE loss weight        |
| $w_{entropy}$ | 0.01 entropy loss weight                  |
| $c$           | 0.25 SW-UCB constant in Equation 1        |
| $\tau$        | 256 SW-UCB window size                    |
| $\alpha$      | 0.2 historical gradient importance in Equation 3 |
| $\beta$       | 2 similar subgraph importance in Equation 3 |
| $r_{min}$     | 1 min number of seconds for repeated measurement of a schedule |

### Table 6: Tensor Operator Configurations

| Operator | Description | Configurations                  |
|----------|-------------|---------------------------------|
| GEMM-S   | $(M, K, N)$ | (128,128,128), (128,256,128), (256,256,256), (512,32,512) |
| GEMM-M   | $(M, K, N)$ | (512,512,512), (128,1536,512), (128,512,1536), (256,1024,512) |
| GEMM-L   | $(M, K, N)$ | (1024,1024,1024), (128,3072,768), (128,768,3072), (256,1224,512) |
| C1D      | $(L, C_i, C_o, K, stride, padding)$ | (256,64,128,3,2,1), (128,128,256,1,2,0), (64,256,256,1,2,0), (32,512,512,3,1,1) |
| C2D      | $(H, W, C_i, C_o, K, stride, padding)$ | (224,224,3,64,7,2,3), (56,56,64,64,1,1,0), (14,14,256,256,3,1,1), (7,7,512,512,3,1,1) |
| C3D      | $(D, H, W, C_i, C_o, K, stride, padding)$ | (16,224,224,3,64,7,2,3), (16,56,64,64,1,1,0), (16,14,256,256,3,1,1), (16,7,512,512,3,1,1) |
| T2D      | $(H, W, C_i, C_o, K, stride, padding)$ | (4,4,512,256,4,2,1), (8,8,256,128,4,2,1), (16,16,128,64,4,2,1), (32,32,64,3,4,2,1) |

### Table 7: Sensitivity analysis for adaptive-stopping window size $\lambda$ on 1024x1024x1024 GEMM. Each experiment runs for 1000 trials.

| $\lambda$ | Normalized Performance | Normalized Time/Iteration |
|-----------|------------------------|--------------------------|
| 10        | 0.917                  | 0.491                    |
| 20        | 0.991                  | 0.558                    |
| 40        | 1.0                    | 0.706                    |
| 80        | 0.993                  | 1.0                      |

### Table 8: Sensitivity analysis for adaptive-stopping elimination ratio $\rho$ on 1024x1024x1024 GEMM. Each experiment runs for 1000 trials.

| $\rho$ | Normalized Performance | Normalized Time/Iteration |
|--------|------------------------|--------------------------|
| 0.75   | 0.864                  | 0.748                    |
| 0.5    | 0.990                  | 0.817                    |
| 0.25   | 1.0                    | 1.0                      |

The depths of auto-unroll modification described in Section 4.2 are different for CPU and GPU environment:
- CPU: 0, 16, 64, 512
- GPU: 0, 16, 64, 512, 1024

### A.2 Platform Configuration

The optimized tensor programs are evaluated on both CPU and GPU platforms.
- CPU platform is configured with Intel Xeon 6226R with 32 cores and 2.9GHz. 128GB memory is used. The OS is Ubuntu 20.04. AVX512 instruction set is supported.
- GPU platform is configured with one Nvidia GeForce RTX 3090.
A.3 Shapes of Tensor Operators
The different shapes for the tensor operators used in the evaluation in Section 6.2 are shown in Table 6. The convolution parameters are the same with those used in Ansor [28].

A.4 Hyper Parameters Selection

Adaptive-stopping window size $\lambda$: The window size is set to 20 in the default settings. Experiments on different settings are shown in Table 7. To make the search time metric directly reflect the running speed instead of the search efficiency, we use the normalized time spent for each search iteration as the metric. As shown in the table, smaller $\lambda$ affects the final performance, since the potentially well-performed schedule tracks are not sufficiently explored before being eliminated. However, a larger $\lambda$ does not always result in an efficient search. As $\lambda$ increases, the improvement of the number of potentially good candidates becomes marginal, since they are selected by approximations instead of the actual measurements. However, the search time is largely affected due to increased episode lengths, and it starts to match the time spent on measurements.

Adaptive-stopping elimination rate $\rho$: Similarly to the window size $\lambda$, the elimination rate also faces the trade-off between the performance and the search time. We tried 3 different values 0.25, 0.5, 0.75 on elimination rate $\rho$, and found that $\rho = 0.25$ eliminates less candidates but produces tiny performance improvements on $\rho = 0.5$, while $\rho = 0.75$ has a significant performance drop. For the best search time and performance consideration, we choose $\rho = 0.5$ as the default setting of the experiments.