Bansor: Improving Tensor Program Auto-Scheduling with Bandit Based Reinforcement Learning

Chao Gao[1]*, Tong Mo[1], Taylor Zowtuk[1], Tanvir Sajed[1], Laiyuan Gong[2], Hanxuan Chen[2], Shangling Jui[2], Wei Lu[1]
[1]*Huawei Research Institute, Canada* [2]*Huawei Technologies, China*
{chao.gao4, tong.mo, taylor.zowtuk, tanvir.sajed, jui.shangling}@huawei.com
{gonglaiyuan, chenhanxuan, robin.luwei}@hisilicon.com

Abstract—**Efficient execution is crucial to the successful deployment of deep learning models to real-world applications. Considerable recent effort have been devoted to computer systems for automatic discovery of efficient *schedules* for executing tensor programs on given hardware platforms. Built on TVM [1], Ansor [2] is the most recent and state-of-the-art framework for auto-scheduling deep neural net computation pipelines. In this paper, we present *Bansor*, an improved version of Ansor for automatic optimization of tensor programs, using bandit-based reinforcement learning (RL). We reexamine the algorithmic procedures of Ansor, and identify two *selection* sub-problems where RL techniques are readily usable. We then introduce bandit-based algorithms to balance exploration and exploitation during the processes of *sketch* selection and *task* scheduling. We evaluate the resulting algorithm, *Bansor*, on a wide range of tensor programs and two hardware platforms. Experiment results show that *Bansor* yields significant improvement over Ansor. On hard network test cases, Bansor uses an order of magnitude less number of measurement trails to attain Ansor's best schedules, eventually converging to significantly better results given an equal number of measurement trials, despite the fact that Ansor's performance has been superb.**

Index Terms—**Tensor program scheduling, Reinforcement learning, Bandit algorithms**

I. Introduction

Recent advancements in deep learning [3] have prompted significant progresses in a number of areas, resulting in ubiquitous industrial applications. Due to their extensive use of tensor computations, it is known that executing deep neural networks (DNNs) can incur high latency. Indeed, algorithms and frameworks to reduce the execution latency of deep neural network models on heterogeneous hardware platforms have attracted considerable industrial and academic interest in recent years [4]. A number of deep learning frameworks have been developed to facilitate research and deployment (e.g., PyTorch [5]), which translate tensor operators (e.g., conv2d) to vendor-supported kernel libraries (e.g., cuDNN [6]) to attain fast execution. However, developing kernel libraries requires significant human engineering for every specific hardware variant, hindering hardware innovations, especially for AI-specialized accelerators [7].

To overcome this obstacle, Chen et al. [4] developed the TVM framework, which uses AutoTVM [1] to automatically generate efficient and hardware-dependent low-level implementations (i.e., *schedules*) for tensor programs. AutoTVM requires hand-crafted manual templates for each tensor operator, then uses simulated annealing (SA) and a cost model to search for fast schedules. The cost model is used to predict the quality of schedules, and is used to guide the SA. The SA search is repeated many times, where at the end of each round, the cost model is updated from a dataset consisting of all generated and measured schedules. AutoTVM achieved competitive results with hand-tuned libraries on both CPUs and GPUs. The upgraded version of AutoTVM, namely Ansor [2], significantly improved the performance of AutoTVM by removing the burden of *template* writing with an automatic *sketch* generation procedure, and repeatedly using a well-designed genetic algorithm for searching parameters within sketches.

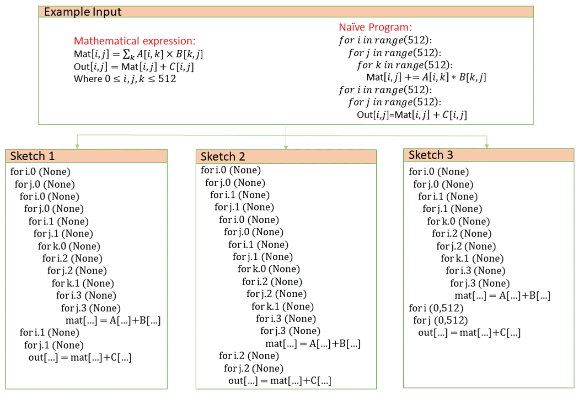
Despite the superior performance, in light of the algorithmic advancements in many AI sub-fields, in this paper, we show that by introducing RL techniques from the *bandit* [8] literature, prominent improvements can be achieved. We thus name our algorithm *Bansor*. Experimental comparisons with Ansor show that Bansor surpasses Ansor in almost all test cases for both operator tuning and whole network optimization scenarios.

The rest of this paper is organized as follows. In Section 2 we review Ansor. In Section 3 we describe our algorithmic improvements over Ansor. Section 4 discusses related work. Experiment results are reported in Section 5. Finally, we provide our conclusion in Section 6.

II. Auto-Scheduling in Ansor

To keep this paper self-contained, we first provide introductory description of two important concepts (i.e., *tasks* and *sketches*) for Ansor, then give a brief summary of its auto-scheduling algorithm.

* corresponding author



Mathematical expression:

$\mathrm{Mat}[i,j] = \sum_k A[i,k] \times B[k,j]$
$\mathrm{Out}[i,j] = \mathrm{Mat}[i,j] + C[i,j]$
Where $0 \le i,j,k \le 512$

Naïve Program:

```
for i in range(512):
  for j in range(512):
    for k in range(512):
      Mat[i,j] += A[i,k] * B[k,j]
for i in range(512):
  for j in range(512):
    Out[i,j]=Mat[i,j] + C[i,j]
```

Sketch 1

```
for i.0 (None)
 for j.0 (None)
  for i.0 (None)
   for j.0 (None)
    for i.1 (None)
     for j.1 (None)
      for k.0 (None)
       for i.2 (None)
        for j.2 (None)
         for k.1 (None)
          for i.3 (None)
           for j.3 (None)
            mat[...] = A[...]+B[...]
  for i.1 (None)
   for j.1 (None)
    out[...] = mat[...]+C[...]
```

Sketch 2

```
for i.0 (None)
 for j.0 (None)
  for i.1 (None)
   for j.1 (None)
    for i.0 (None)
     for j.0 (None)
      for i.1 (None)
       for j.1 (None)
        for k.0 (None)
         for i.2 (None)
          for j.2 (None)
           for k.1 (None)
            for i.3 (None)
             for j.3 (None)
              mat[...] = A[...]+B[...]
    for i.2 (None)
     for j.2 (None)
      out[...] = mat[...]+C[...]
```

Sketch 3

```
for i.0 (None)
 for j.0 (None)
  for i.1 (None)
   for j.1 (None)
    for k.0 (None)
     for i.2 (None)
      for j.2 (None)
       for k.1 (None)
        for i.3 (None)
         for j.3 (None)
          mat[...] = A[...]+B[...]
for i (0,512)
 for j (0,512)
  out[...] = mat[...]+C[...]
```

Fig. 1: Ansor generated sketches for a matrix multiplication task on an Intel CPU.

A. Tasks

A DNN can be topologically expressed as a directed acyclic computation graph (DAG). Ansor optimizes the total latency by partitioning the graph into a set of independent subgraphs. Each subgraph is viewed as a *task*. For example, ResNet-50 [9] contains 29 different subgraphs, where each of them stands for a convolution layer with different shape information (e.g., input dimension, kernel size, stride, padding). A subgraph may appear multiple times in a DNN $\mathcal{E}$. Let w_i be the number of times task $\mathcal{T}_i$ appears and suppose there are m tasks in total, the overall latency can be expressed using weighted sum:

$$\hat{f}(\mathcal{E}) = \sum_{i=1}^{m} w_i \cdot g(\mathcal{T}_i) \tag{1}$$

where $g(\mathcal{T}_i)$ is the latency of a single task $\mathcal{T}_i$, e.g., a 2D convolution layer composed by convolution operation followed by element-wise activation function. We note that a standalone operator (e.g., conv2d operation itself) can also be optimized by Ansor as a single *task*.

B. Sketches

Given a task, sketches represent a high-level structure for the implementation of that computation task. Suppose we are to optimize task $\mathcal{T}_i$, Ansor first derives a set of sketches $\mathcal{S}_{\mathcal{T}_i}$ by taking into account the target hardware properties, and then fine-tunes the parameter choices within a sketch through random annotation. Figure 1 shows an example for a matrix multiplication tensor operator task on CPU. Three sketches with different loop structures are derived.

C. Auto-Scheduling

The auto-scheduling algorithm in Ansor is a repeated procedure that can be summarized as in Algorithm 1. The input to the algorithm contains a tensor program e (e.g., a neural net, a conv2d layer or a matrix-multiplication operator), hardware information h, and parameters for configuring the auto-tuning. Two key parameters are total number of measurements $n\_measures$ and number of measurements to be performed each round $n\_measures\_per\_round$. Note that the genetic algorithm (GA) also contains several parameters — we ignore them in Algorithm 1 as we will keep GA untouched in this paper.

Algorithm 1: Auto-scheduling in Ansor

```
Input : $e$: tensor program expression; $h$: hardware property;
        $n\_measure$: maximum number of measure trials;
        $n\_measure\_per\_round$: measures per round.
Result: An efficient hardware-dependent schedule for $e$
1  Split $e$ into $m$ tasks $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_m$
2  For each task $_i$, using $h$, generate a set of sketches $\mathcal{S}_{\mathcal{T}_i}$
3  $count \leftarrow 0$
4  Initialize cost model $\mathcal{C}$
5  while $count < n\_measures \times m$ do
6      Select a task based on gradient
7      Randomly select a sketch $\kappa$ from $\mathcal{S}_{\mathcal{T}_i}$
8      Sample initial population of schedules $P$
9      GA: repeatedly do mutation, cross-over on $P$, using $\mathcal{C}$
10     Send top $n\_measures\_per\_round$ schedules to
         measure on hardware and write result to dataset $\mathcal{D}$
11     Update cost model $\mathcal{C}$ using data in $\mathcal{D}$
12     $count \leftarrow count + n\_measures\_per\_round$
13 end
14 return best schedules found
```

To tune a DNN $\mathcal{E}$ that consists of m tasks, a straightforward approach is tuning each task $\mathcal{T}_i$ independently in parallel; however, in practice, different tasks have to share a fixed total number of measure trials, i.e., $n\_measures$. Thus, given $\mathbf{t} \in \mathbf{Z}^m$ as a measure trial vector, Ansor optimizes a surrogate function $f(g(t_1), g(t_2), \ldots, g(t_m))$. At each round, in order to achieve largest latency reduction, Ansor selects a task by following the task that has the largest approximated gradient $|\frac{\partial f}{\partial t_i}|$:

$$\begin{aligned}\frac{\partial f}{\partial t_i} \approx \frac{\partial f}{\partial g(t_i)} \cdot (\alpha \frac{g(t_i) - g(t_i - \Delta t)}{\Delta t} + \\ (1-\alpha)(\min(-\frac{g(t_i)}{t_i}, \beta \frac{C_i}{\max_{k \in Nr(i)} V_k} - g(t_i))))\end{aligned} \tag{2}$$

Δt is a small time window, C_i is the number of floating point operation in task i and V_k is the number of floating point operation per second in task k. $Nr(i)$ contains neighbors of i, i.e., those tasks similar to i. See [2] for details. In the beginning, Ansor initializes $\mathbf{t} = \mathbf{1}$ by warming up with a single round of round-robin. In the remaining, Ansor uses ε-greedy [10] to encourage exploration, i.e., it chooses a random tasks i by probability ϵ, otherwise task $i \leftarrow \arg\max_i |\frac{\partial f}{\partial t_i}|$, then sets $t_i \leftarrow t_i + 1$. Note that if Ansor is used to optimize a single task, this task scheduling is ignored.

III. Bansor: Improved Ansor with Bandits

We identify two *selection* subproblems to improve Ansor's auto-scheduling.

A. Sketch Selection using UCB

From Algorithm 1 we see that Ansor selects a sketch uniform randomly in each round to run the evolutionary search on. However, for a given task $\mathcal{T}$ and hardware h, a reasonable argument is that sketches are having disparate *qualities*, i.e., it might be easier for evolutionary search to find good performing schedules from sketch a than b simply because a's loop structure is more suitable to run $\mathcal{T}$ on hardware h. Suppose there are k sketches, the optimal schedule cost for each sketch κ_i is noted as $o_i, \forall 1 \le i \le k$, and at round $N(\mathcal{T})$ the previously obtained schedule cost sequences at sketch i is noted as $\mathbf{X_i}$. We then perform sketch selection as a stochastic bandit that can be optimized using an upper-bound confidence (UCB) [11] formula:

$$UCB(\kappa_i) = r(\hat{X}_i) + C\sqrt{\frac{2\log N(\mathcal{T})}{N(\kappa_i)}}, \tag{3}$$

where $\hat{X}_i \triangleq \min\{\mathbf{X}_{ij} \mid j = 1, \ldots, N(\kappa_i)\}$, $N(\mathcal{T})$ is the number of rounds so far; $N(\mathcal{S}_i)$ is the number of rounds that κ_i has been selected; C is constant; r is a function that transforms the cost sequence into a single score of $[0, 1]$.

$$r(\hat{X}_i) = \frac{\min\{\hat{X}_j \mid j = 1, \ldots, k\}}{\hat{X}_i} \tag{4}$$

That is, we use the minimum latency to define the reward because we care more about whether a sketch can generate the fastest schedule, rather than the average ability to generate fast schedules.

B. Task Selection

As shown in Algorithm 1, when tuning the neural network, Ansor will select the task with the largest gradient value $|\frac{\partial f}{\partial t_i}|$ in each round and adopt an ε-greedy strategy to encourage exploration. This prioritizes a subgraph that has a high initial latency, but it is known that (e.g, for stochastic bandits) ε-greedy strategy may result in a cumulative regret that grows linearly with number of rounds [8]. Thus, similar to sketch selection, we propose to use the UCB to balance exploration and exploitation for task selection:

$$UCB(\mathcal{T}_i) = r(\mathcal{T}_i) + C\sqrt{\frac{2\log N(\mathcal{E})}{N(\mathcal{T}_i)}}, \tag{5}$$

where $N(\mathcal{E})$ is the number of search rounds so far; $N(\mathcal{T}_i)$ is the number of search rounds that task $\mathcal{T}_i$ has been selected; C is constant. Reward for task $\mathcal{T}_i$, i.e., $r(\mathcal{T}_i) \in [0, 1]$, is defined as follows:

$$r(\mathcal{T}_i) = \frac{|\frac{\partial f}{\partial t_i}|}{\max\{|\frac{\partial f}{\partial t_j}| \mid j = 1, \ldots, m\}} \tag{6}$$

The importance of adding exploration to task selection is that the term $|\frac{\partial f}{\partial t_i}|$ is essentially a crude approximation, resembling a stochastic reward in bandit problems. An algorithm that does not effectively explore alternatives may get stuck on certain tasks that repeatedly appear to be promising due to the noise in evaluation. For stochastic bandits, UCB attains a successful exploration with logarithmic cumulative regret, while ϵ-greedy's regret is linear [11].

IV. Related Work

Schedule search framework and algorithms: Based on manually written schedule templates, AutoTVM [1] is the first auto-schedule search algorithm developed upon TVM. It searches for best schedules based on a statistical cost model learned from data obtained by running tensor programs on hardware. The major drawback for AutoTVM is that the performance of schedule search is limited by quality of the manually created templates for each operator and targeting hardware platform. In this regard, FlexTensor [12] improves upon AutoTVM by leveraging a method that uses template for a family of operators sharing similar computation structures. Instead of letting human engineers directly encode hardware properties into operator templates, Ansor [2] proposes to use concise *rule sets* to describe different hardware platforms, then devises an algorithm to automatically create preliminary schedule templates, called sketches. Given an operator or subgraph, Ansor randomly samples a sketch from a list of available sketches and runs an evolutionary search to annotate the parameters of the sketch. Similar to AutoTVM, a cost model, XGBoost [13] by default, is used by the evolutionary search for assessing schedule qualities. Ansor runs iteratively, where at each iteration, top schedules are evaluated by hardware, and the cost model is then retrained on these newly produced data points. Ansor outperforms Halide [14], FlexTensor, AutoTVM and other frameworks on a wide range of benchmarks.

Bandits and reinforcement learning: Multi-armed bandit [8] is often regarded as the simplest problem model for reinforcement learning [10], where there is only one state and multiple actions - each associated with a reward drawn from some unknown probability distribution. A variety of bandit problems have been studied in the literature with different specific assumptions on the reward distributions, where the central quest is to balance *exploration* and *exploitation*. In our application scenario, we adopt *stochastic bandit*, since the cost model training typically converges after certain rounds of search, making the rewards sampled from the genetic algorithm akin to i.i.d. Besides UCB [11], Bayesian approach, such as Thomas Sampling [15], can be used for stochastic bandits, and in practice they may show better empirical results in terms of *cumulative regret* due to more concentrated exploration, whereas in our application, the continual exploration of UCB is arguably more desirable as the ultimate goal is to identify a best schedule, not to optimize cumulative regret.

V. Experiments

We evaluate the performance of programs generated by Bansor on three aspects: single operators, subgraphs, and entire neural networks. To have a fair comparison, both Bansor and Ansor are built from the same code base [1].

The generated tensor programs are benchmarked on two hardware platforms: Intel CPU (Xeon Gold 6140 CPU @

[1]https://github.com/apache/tvm

2.30GHz) and NVIDIA GPU (Tesla V100). All the test cases are exactly the same as those in [2].

Due to expensive evaluation cost, extensively parameter tuning is difficult, we thus tune the exploration constant C from $\{0.1, 0.3, 0.5, 1.0\}$ using a matrix multiplication task on CPU, then set C to 0.3 for both sketch and task selection.

A. *Single Task Optimization*

We first evaluate the performance of Bansor on 10 common deep learning operators, namely one-dimensional convolution (C1D), two-dimensional convolution (C2D), three-dimensional convolution (C3D), matrix multiplication (GMM), group convolution (GRP), dilated convolution (DIL), depth-wise convolution (DEP), transposed 2D convolution (T2D), capsule 2D convolution (CAP), and matrix 2-norm (NRM). We compare Ansor and Bansor on 80 test cases. For each operator, we use 4 shapes and evaluate them on two different batch sizes (1 and 16). These test cases are exactly the same compared to those used in [2].

As in [2], we allow Ansor and Bansor to have 1000 measurement trials for each test case, which is arguably large enough for search to converge. For each test case, we collect the best schedule costs in milliseconds, then plot the geometric mean of the four shapes. The geometric mean is normalized to the best performing framework such that the best algorithm has a normalized performance of 1. As shown in Figure 2, Bansor performs better for all operators and batch sizes - finding schedules that are faster than those found by Ansor by 1.04–1.59×.

We next evaluate Bansor on two common subgraphs in DNNs: the 2D convolution layer (ConvLayer) that consists of a two-dimensional convolution, batch normalization and ReLU operators, and the TBS layer composed of two matrix transposes, one batch matrix multiplication, and a softmax function. Again, we use exactly the same test cases as in [2] (four different shapes and two batch sizes), and run them on both CPU and GPU for 1000 measurement trials. Figure 3 summarizes the result, which indicates that Bansor outperforms Ansor with a speedup of up to 1.40×.

Table I shows the number of sketches generated for each operator or subgraph. We see that most of them have 3 sketches. Subgraph "ConvLayerG" has only 1 sketch, indicating that Bansor would behave exactly the same as Ansor for this case. This explains the almost identical performance between Ansor and Bansor in Figure 3. For the other three cases however, Bansor outperforms Ansor, suggesting that UCB sketch selection indeed brought a beneficial effect.

To further investigate the performance of Ansor and Bansor under different parameter settings, we run them on a convolution operator and report the performance curve. We pick a 1D convolution operator with the shape configuration (batch, length, input channel, output channel, kernel size, stride, padding) equals to (1, 64, 256, 256, 5, 1, 2). Each run of a variant uses 1000 measurement trials. Each curve is the median of 5 runs. As in [2], for each trial i, we collect the lowest median latency achieved for the operator up to trial i.

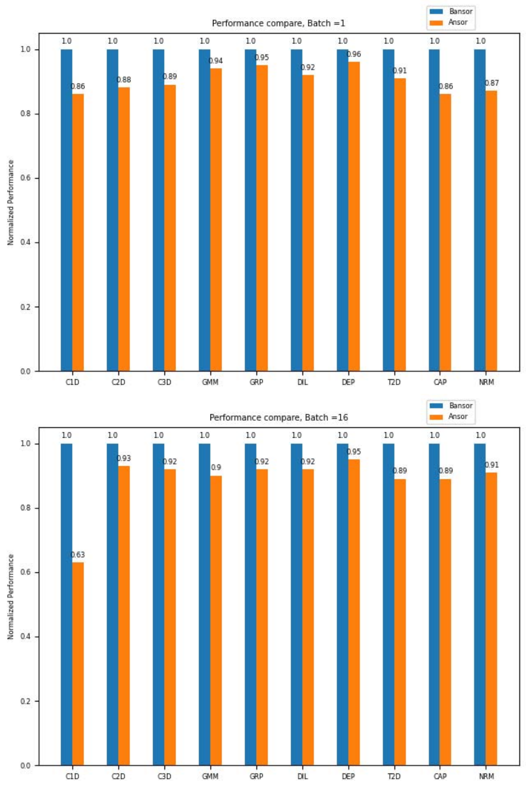


Fig. 2: Single Operator Results on CPU.

TABLE I: Number of sketches for each operator or subgraph. C.L means convolution layer. @C means on CPU. @G means on GPU. The single operators, i.e., C1D, C2D, ..., NRM, were only tested on CPU.

C1D	C2D	C3D	GMM	GRP	DIL	DEP
3	3	3	3	3	3	3
T2D	CAP	NRM	C.L@C	C.L@G	TBS@C	TBS@G
3	3	3	3	1	27	4

We plot this latency normalized to the latency achieved by the best variant. The four variants are named "Ansor_nmpr16", "Bansor_nmpr16", "Ansor_nmpr64" and "Bansor_nmpr64", where *nmpr* represents *number of measurement trials per round*. Figure 4 shows that the two Bansor variants outperform Ansor's throughout nearly the entire auto-tuning process. The variant Bansor_nmpr16 performed better than Bansor_nmpr64 because the nmpr16 version conducted four times more rounds allowing Bansor to spend more rounds on sampling from better sketches.

B. *Multi-Task Whole Network Optimization*

As in [2], we evaluate task scheduling algorithms on the following DNNs: 3D-ResNet-18 [16], DCGAN [17], and BERT [18]. Each network is evaluated at batch sizes of 1 and 16 on CPU and GPU. Besides Ansor, we run two versions

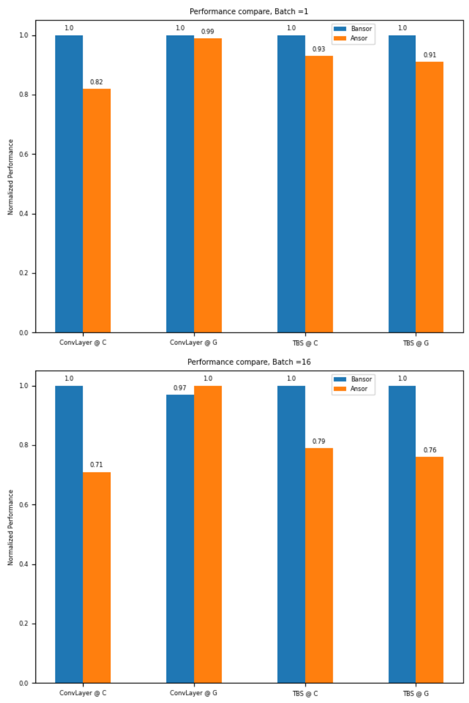


Fig. 3: Subgraph results on CPU and GPU.

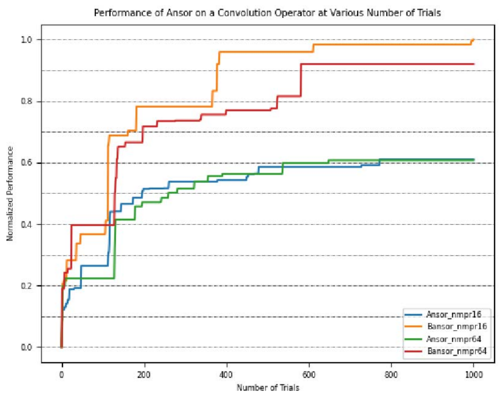


Fig. 4: Search progress of Bansor versus Ansor on 1D convolution operator on CPU.

of Bansor: one with sketch UCB selection (*Bansor1*), and the other with both sketch and task UCB selection (*Bansor2*). As recommended by Ansor, for any given DNN, we run those algorithms for $1000 \times m$ measurement trials, where m is the number of tasks in the network. Table II lists the details of each network.

Table III summarizes the comparative results of Ansor, Bansor1 and Bansor2, which indicate that gradient UCB task selection and UCB sketch selection can indeed lead to better schedules. The largest schedule speed up is $1.38\times$ on CPU 3D-Resnet-18 with batch 16. We also observe that Bansor's strengths are mainly manifested on harder cases (i.e., those with high latency) than easier ones. To see this, we plot the ratio between Bansor's schedule costs against Ansor's in Figure 5. The downward curves suggest that Bansor's speedup relative to Ansor seems to improve linearly w.r.t schedule latency and network complexity. In most cases, Bansor2 yields slightly better results than Bansor1, indicating the task selection by UCB is worthwhile.

TABLE II: The number of tasks and number of trials used for evaluation of each DNN.

Network	Tasks	Trials	#Sketch max	#Sketch min
3D-ResNet-18	23	23000	(9, 4)	(1, 1)
BERT	10	10000	(9, 4)	(3, 1)
DCGAN	5	5000	(3, 1)	(3, 1)

It is worth noting that in Table II, for the DCGAN on GPU, all tasks have only one sketch. This renders Bansor1 equivalent to Ansor, and thus any speedup of Bansor2 should be solely attributed to the task UCB selection, while small discrepancies between Ansor and Bansor1 are due to randomness of the evolutionary search.

While Table III shows the best schedules found by Bansor and Ansor given the same number of measure trials, another way to demonstrate Bansor's strength is to report how much computation is required for Bansor to achieve Ansor's best schedule performance for each case. Tables IV list the measurement trials and search time required respectively for Bansor to match the performance of Ansor on the network benchmarks: In 3 among 12 cases, Bansor can match the performance of Ansor with *an order of magnitude less* measurement trials. The largest trial savings is $26.52\times$. These results further certify the algorithmic superiority of Bansor over Ansor.

VI. Conclusions

As pointed out in [4], the TVM framework can be used as a testbed for new algorithmic innovations from various machine learning directions. In this work, we have presented a simple yet powerful algorithmic improvement for Ansor by borrowing ideas from reinforcement learning, and demonstrated the advantages of the new system on a diverse set of test cases and two hardware platforms. Despite the superior performance of Bansor, we identify a few future directions that are worth exploring for achieving even better results:

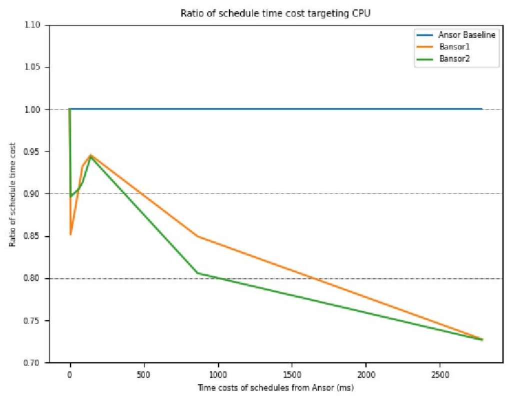
- Using a full implementation of Monte Carlo tree search to replace evolutionary search is an interesting direction for future study.
- Quantifying uncertainty of cost model. In the current system, the cost model is trained in an online fashion and is used to replace hardware execution in search. Thus, given a schedule, it is desirable that we can quantify to what extent the cost model predictions can be trusted, thereby allowing a more preferable use of the cost model during search.

TABLE III: Network results on CPU and GPU. Boldface indicates the schedule cost is more than 10% smaller than Ansor.

	CPU					
	Batch 1			Batch 16		
	Ansor	Bansor1	Bansor2	Ansor	Bansor1	Bansor2
DCGAN	6.74	**5.74**	6.04	86.78	80.91	79.23
BERT	62.83	56.99	56.90	865.45	**734.93**	**697.22**
3D-ResNet	141.76	134.05	133.73	2783.98	**2025.51**	**2023.05**
	GPU					
	Batch 1			Batch 16		
	Ansor	Bansor1	Bansor2	Ansor	Bansor1	Bansor2
DCGAN	0.64	0.65	0.60	5.06	4.99	5.00
BERT	5.81	5.72	5.42	53.38	**47.37**	**46.26**
3D-ResNet	10.24	9.22	**8.69**	135.05	**116.52**	**116.98**

TABLE IV: Number of *measure trials* used by Bansor to match the best performance of Ansor. Among 9 of the 12 test cases, Bansor can achieve a measure trial saving of at least 2×.

	Network	Ansor	Bansor1	Bansor2	Trials_Saving(Bansor1, Bansor2)
CPU	DCGAN_B1	4634	1122	918	(**4.13**×, **5.05**×)
	BERT_B1	9921	3193	2138	(**3.11**×, **4.64**×)
	3D-ResNet_B1	22497	19954	19873	(1.13×, 1.13×)
	DCGAN_B16	4450	817	698	(**5.45**×, **6.38**×)
	BERT_B16	9920	801	374	(**12.38**×, **26.52**×)
	3D-ResNet_B16	21759	9022	13270	(**2.41**×, 1.64×)
GPU	DCGAN_B1	4836	5000	3704	(0.97×, 1.31×)
	BERT_B1	9990	2112	1763	(**4.73**×, **5.67**×)
	3D-ResNet_B1	22769	2024	1382	(**11.25**×, **16.48**×)
	DCGAN_B16	4137	3296	2939	(1.26×, 1.41×)
	BERT_B16	9943	1783	2510	(**5.58**×, **3.96**×)
	3D-ResNet_B16	22977	1672	3346	(**13.74**×, **6.87**×)



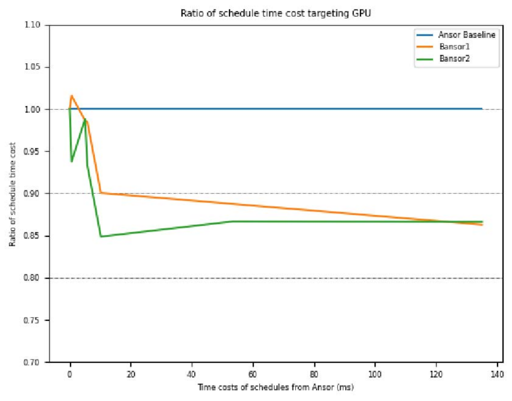


Fig. 5: Relative ratio of schedule time costs on CPU and GPU.

References

[1] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to optimize tensor programs," in *Advances in Neural Information Processing Systems*, 2018, pp. 3389–3400.
[2] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen *et al.*, "Ansor: Generating high-performance tensor programs for deep learning," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 863–879.
[3] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.
[4] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, and L. Ceze, "Tvm: an automated end-to-end optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, p. 578–594.
[5] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in neural information processing systems*, 2019, pp. 8026–8037.
[6] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," 2014, arXiv preprint arXiv:1410.0759.
[7] T. Moreau, T. Chen, L. Vega, J. Roesch, E. Yan, L. Zheng, J. Fromm, Z. Jiang, L. Ceze, C. Guestrin *et al.*, "A hardware–software blueprint for flexible deep learning specialization," *IEEE Micro*, vol. 39, no. 5, pp. 8–16, 2019.
[8] T. Lattimore and C. Szepesvári, *Bandit algorithms*. Cambridge University Press, 2020.
[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
[10] R. Sutton and A. Barto, *Reinforcement learning: an introduction*. MIT press, 2018.
[11] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, no. 2-3, pp. 235–256, 2002.
[12] S. Zheng, Y. Liang, S. Wang, R. Chen, and K. Sheng, "Flextensor: 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*, 2020, p. 859–873.
[13] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.
[14] A. Adams, K. Ma, L. Anderson, R. Baghdadi, T.-M. Li, M. Gharbi, B. Steiner, S. Johnson, K. Fatahalian, F. Durand *et al.*, "Learning to optimize halide with tree search and random programs," *ACM Transactions on Graphics (TOG)*, vol. 38, no. 4, p. 121, 2019.
[15] O. Chapelle and L. Li, "An empirical evaluation of thompson sampling," *Advances in neural information processing systems*, vol. 24, pp. 2249–2257, 2011.
[16] K. Hara, H. Kataoka, and Y. Satoh, "Can spatiotemporal 3d cnns retrace the history of 2d cnns and imagenet?" in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR*, 2018, p. 6546–6555.
[17] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," 2015, arXiv preprint arXiv:1511.06434.
[18] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: pre-training of deep bidirectional transformers for language understanding," 2018, arXiv preprint arXiv:1810.04805.