ETO: Accelerating Optimization of DNN Operators by High-Performance Tensor Program Reuse

Jingzhi Fang
HKUST
jfangak@connect.ust.hk

Yanyan Shen
Shanghai Jiao Tong University
shenyy@sjtu.edu.cn

Yue Wang
Shenzhen Institute of Computing Sciences
yuewang@sics.ac.cn

Lei Chen
HKUST
leichen@cse.ust.hk

ABSTRACT

Recently, deep neural networks (DNNs) have achieved great success in various applications, where low inference latency is important. Existing solutions either manually tune the kernel library or utilize search-based compilation to reduce the operator latency. However, manual tuning requires significant engineering effort, and the huge search space makes the search cost of the search-based compilation unaffordable in some situations. In this work, we propose ETO, a framework for speeding up DNN operator optimization based on reusing the information of performant tensor programs. Specifically, ETO defines conditions for the information reuse between two operators. For operators satisfying the conditions, based on the performant tensor program information of one operator, ETO uses a reuse-based tuner to significantly prune the search space of the other one, and keeps optimization effectiveness at the same time. In this way, for a set of operators, ETO first determines the information reuse relationships among them to reduce the total search time needed, and then tunes the operators either by the backend compiler or by the reuse-based tuner accordingly. ETO further increases the reuse opportunities among the operators by injecting extra operators as bridges between two operators which do not satisfy the reuse conditions. Compared with various existing methods, the experiments show that ETO is effective and efficient in optimizing DNN operators.

PVLDB Reference Format:
Jingzhi Fang, Yanyan Shen, Yue Wang, and Lei Chen. ETO: Accelerating Optimization of DNN Operators by High-Performance Tensor Program Reuse. PVLDB, 15(2): 183 - 195, 2022.
doi:10.14778/3489496.3489500

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Experiment-code/ETO.

1 INTRODUCTION

In recent years, various deep learning models have made great success in many applications, where low latency in inference is important (e.g., language translation [7]). To achieve efficient execution of DNNs, people either rely on the manually tuned kernel libraries (e.g., cuDNN [6]) provided by the AI-hardware vendors, or search-based compilation [1, 3–5, 27, 28] which automatically optimizes an operator (or a set of operators connected with each other). Since manually tuning kernel libraries requires significant engineering effort, search-based compilation has attracted much attention recently. In order to find performant *tensor programs* (low-level programs), search-based compilation generally defines a huge search space with comprehensive coverage of useful optimizations for an operator (or a set of operators), which incurs high search cost. For example, to optimize an operator, Ansor [27], a search-based compilation method, needs about 1,000 measurement trials to converge in general, which consumes thousands of seconds.

In practice, long compilation time is unaffordable in some situations and may limit the final optimization effect. For example, some DNNs work on data sets which are updated periodically or continuously, e.g., the graph taken by a graph neural network can have updates to nodes or edges. The operators involved in the models that take different input data shapes need to be optimized accordingly when the data set changes. If the optimization process is not efficient enough, we may not even finish it before the next data set update arrives. Another example is BERT [7] for input with dynamic shapes. Since the sequence length in input can be any value (no greater than 512), the number of possible input data shapes for an operator in BERT can be exponentially large [17]. In this case, optimizing the operator for every input data shape is not practical, and we may only do the tuning for some shapes, making a trade-off between the search time and the optimization effect of an operator in terms of its final inference performance. If the optimization of an operator with a given input data shape can be done more efficiently than the existing time-consuming compilers, we can tune a dynamic operator for more different input data shapes within a fixed time period, and potentially achieve higher inference performance. The above facts lead to the following question: can we accelerate the optimization process of DNN operators?

An idea to speed up the search-based compilation is making use of the good tensor programs of well-optimized operators. An important work is Selective Tuning (ST) [19]. ST is based on an assumption that the transformation steps to get high-performance tensor programs for similar operators can be the same. Therefore, operators which are pairwise similar enough are put in the same cluster, and they directly try the transformation steps in the performant tensor programs of the selected operator in the cluster. To

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 15, No. 2 ISSN 2150-8097.
doi:10.14778/3489496.3489500

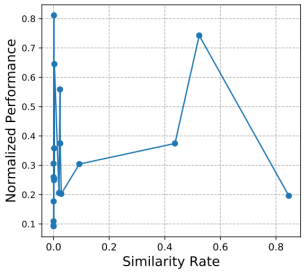


Figure 1: Operator similarity vs. the performance of a conv2d in ResNet-50 adopting transformation steps of another conv2d in ST.

achieve good optimization effect, it is crucial to define a similarity measure that is inversely proportional to the latency of an operator after such transformation step sharing. However, the similarity measure defined in ST, i.e., the overlap ratio of the search space (transformation steps) between two operators, does not satisfy this requirement as shown in Figure 1. In this way, ST fails to effectively make use of the good tensor programs of other operators in optimization. More importantly, transformation steps cannot be shared between two operators that are not similar enough, and this may limit the optimization efficiency of ST, e.g., ST puts the 20 conv2d operators in ResNet-50 [10] in 7 groups in our experiments.

By analysing the good tensor programs of different operators, we find that the their difference in the hardware resource utilization information (e.g., the number of GPU blocks) is related to the corresponding difference in the operator computation loops, and hence we can prune the search space of an operator when given the performant tensor program of another operator. Following this insight, we aim to design a new method to accelerate the optimization of a set of operators, which can make use of the performant tensor programs effectively and reach higher overall optimization efficiency. To reach this goal, there are two challenges. First, we need to search for a good tensor program of an operator efficiently based on the performant tensor program information (about resource utilization) of another operator. The information of the tensor program found in this way can help optimize other operators again, which reduces the number of operators to bootstrap the optimization of all the operators. Second, when optimizing a set of operators, we should estimate the optimization time needed and determine the information reuse relationships among the operators to save the overall optimization time based on our estimation.

To this end, we propose ETO, a framework for accelerating DNN operator optimization. ETO currently focuses on GPU, which is popular for DNN training and inference, and it has two major components: the reuse-based tuner, the reuse pair selector. Two conditions about the operator characteristics are proposed to decide whether one operator can reuse the performant tensor program information of another. For any two operators satisfying the two conditions, the reuse-based tuner uses a hierarchical search method to explore a pruned search space of the operator to be optimized, based on the given performant tensor program information. To optimize multiple operators, the reuse pair selector first determines the information reuse relationships among them to reduce the overall search time needed, based on the estimated search cost in optimization. The operators with no operator to reuse would be tuned by a backend compiler, while the others are optimized by the reuse-based tuner. The reuse relationships form a tree in the end, with operators and a special root (connecting operators sent to the backend compiler) as nodes and reuse relationships as edges. Besides, we propose to add extra operators, namely bridge operators, to further accelerate the overall optimization process when the operator pairs satisfying the conditions are insufficient.

Our framework reaches a trade-off between inference performance and optimization efficiency. We evaluate the performance of ETO in two levels: the reuse-based tuner, and the whole framework (including the reuse pair selector). Experiments are conducted on both popular DNN operators and DNN operator sets, where Ansor [27] and ST [19] are baselines. Results show that, compared with Ansor, the reuse-based tuner gets close or better inference performance ($94.7\% - 257.1\%\times$) but is $1.1 - 12.1\times$ faster; the overall performance of ETO is good as well, achieving $91.4\% - 101.3\%$ inference performance with $3.1 - 16.1\times$ speedup in searching. ETO also achieves a better trade-off than ST.

To summarize, our major contribution in this paper is fourfold. (1) We present a framework to accelerate DNN operator optimization. (2) We propose an effective and efficient search method with reuse conditions to find a good tensor program for an operator by reusing the performant tensor program information of another. (3) We design a method to increase reuse opportunities among operators and then determine reuse relationships among operators to reduce the overall search time. (4) We conduct extensive experiments with various types of operators and operator sets on GPU which shows ETO can achieve relatively good operator performance in a much shorter time than the state-of-the-art methods.

2 BACKGROUND

This section is about the background knowledge of DNN operator optimization: operator and loop nest, GPU tensor program pattern and tensor program search space. Table 1 summarizes the notations.

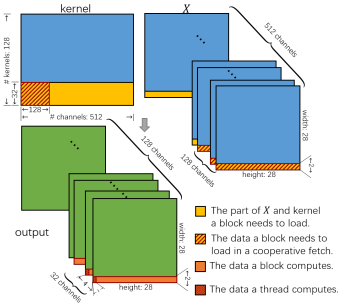
Operator and loop nest. An operator conducts computation specified by its type over the input values, for example, an operator of type conv2d conducts 2D convolution. A loop nest is a set of loops nested together. For an operator, there can be many tensor program variants, i.e., different low-level programs. Each tensor program has its ***latency*** (inference time) and can be written as a sequence of loops nests. Figure 2a shows the ***naïve tensor program*** of a conv2d operator, i.e., the tensor program without any optimization. This naïve tensor program has 2 loop nests: the first one pads the input data, while the second one computes convolution. We call the loop nest in the naïve tensor program of an operator, which does the major computation task, the ***major loop nest***, e.g., the second loop nest in Figure 2a. There are two kinds of loops in a loop nest: ***space loop***, denoted by $s\ell$, and ***reduction loop***, denoted by $r\ell$. $s\ell$s are related to the output tensor after computation, while $r\ell$s are about the reduction in computation. In Figure 2a, the loops in line 6-9 are $s\ell$s and the loops in line 10-12 are $r\ell$s. The ***iteration space*** of k loops $\mathcal{L} = \{\ell_1, ..., \ell_k\}$, denoted by $\mathbb{I}(\mathcal{L})$, is a k-dimensional space where the i-th dimension corresponds to the iteration of ℓ_i. For example, the $r\ell$s in the second loop nest of Figure 2a form a 3D space of shape $[512, 1, 1]$. Each operator is associated with an ***optimized tensor program***, which is the best tensor program found by an optimization method, like autoTVM [5], Ansor [27],

```
for i0 (0,1)
  for i1 (0,512)
    for i2 (0, 28)
      for i3 (0,28)
        pad_temp = ...
for nn (0,1)
  for ff (0, 128)
    for yy (0,28)
      for xx (0,28)
        for rc (0,512)
          for ry (0,1)
            for rx (0,1)
              compute = ...
```

(a) The naïve tensor program.

```
blockIdx.x nn.0@ff.0@yy.0@xx.0@ (0,56)
  threadIdx.x nn.2@ff.2@yy.2@xx.2@ (0,224)
    compute.local auto_unroll: 1024
    for rc.0 (0,4)
      for ax0@ax1@ax2@ax3@.0.0 (0,19)
        threadIdx.x ax0@ax1@ax2@ax3@.0.1 (0,224)
          placeholder.shared = ...
      for ax0@ax1@ax2@ax3@.0.0 (0,32)
        threadIdx.x ax0@ax1@ax2@ax3@.0.1 (0,224)
          pad_temp.shared = ...
      for rc.1 (0,64)
        for ff_c.3 (0,2)
          for rc.2 (0,2)
            for ff_c.4 (0,2)
              for yy_c.4 (0,2)
                compute.local = ...
      for ff.3 (0,4)
        for yy.3 (0,2)
          compute = ...
```

(b) A high-performance tensor program.



(c) Data loading and parallel computing.

Figure 2: A conv2d example (in NCHW layout; input X of shape $[1, 512, 28, 28]$; kernel of shape $[128, 512, 1, 1]$; stride and dilation are 1; padding is 0): the naïve tensor program, a performant tensor program and how it works.

Table 1: Notation Table.

Variable	Description
$\alpha, \beta, \gamma, \ldots$	operator
τ_α	operator type of α
$\ell; s\ell; r\ell$	loop; space loop; reduction loop
$\mathcal{L}_\alpha$	major loop nest of α
$s\mathcal{L}_\alpha; r\mathcal{L}_\alpha$	space loops of $\mathcal{L}_\alpha$; reduction loops of $\mathcal{L}_\alpha$
$\mathbb{I}(\cdot)$	iteration space
$S_\alpha; \mathcal{S}_\alpha$	a tensor program of α; all tensor program of α
S^*_α	optimized tensor program of α
$\mu(S_\alpha)$	latency of S_α
$\mathcal{K}_\alpha; \mathbb{K}_\alpha$	a sketch of α; all sketches of α
$\mathbb{A}_{\alpha,\mathcal{K}_\alpha}$	an annotation combination of α on $\mathcal{K}_\alpha$
$\mathfrak{A}_{\alpha,\mathcal{K}_\alpha}$	all annotation combinations of α on $\mathcal{K}_\alpha$
$f(\alpha, \mathcal{K}_\alpha, \mathbb{A}_{\alpha,\mathcal{K}_\alpha})$	S_α specified by $\alpha, \mathcal{K}_\alpha, \mathbb{A}_{\alpha,\mathcal{K}_\alpha}$

and our reuse-based tuner which would be introduced in Section 5. We define notations related to an operator below.

Definition 1 (operator). *An operator α is of type τ_α and associated with: a major loop nest $\mathcal{L}_\alpha = s\mathcal{L}_\alpha \cup r\mathcal{L}_\alpha$ where $s\mathcal{L}_\alpha$ is $s\ell$s in $\mathcal{L}_\alpha$ and $r\mathcal{L}_\alpha$ is $r\ell$s in $\mathcal{L}_\alpha$; corresponding iteration spaces $\mathbb{I}(\mathcal{L}_\alpha)$, $\mathbb{I}(s\mathcal{L}_\alpha), \mathbb{I}(r\mathcal{L}_\alpha)$; a set of all tensor program variants $\mathcal{S}_\alpha$; an optimized tensor program $S^*_\alpha \in \mathcal{S}_\alpha$; and a function $\mu : \mathcal{S}_\alpha \to \mathbb{R}^+$ mapping each $S_\alpha \in \mathcal{S}_\alpha$ to its latency.*

GPU tensor program pattern. The thread hierarchy of GPU has two levels: *thread block* and *thread* [14]. Programmers often partition the computation task into independent subtasks for thread blocks to solve them in parallel, and further partition each subtask into smaller tasks for threads in the block to solve in parallel. Threads in a block can work cooperatively, by sharing data through shared memory and synchronizing execution for memory accesses coordination. Optimizing tensor programs is essentially transforming loop nests. A *general pattern* of optimizing loop nests on GPU is first tiling them and then binding the iteration variables to GPU thread indices, to make use of the massive parallelism provided by GPU. Based on the general pattern, other optimization methods can also be applied, like loop reordering and unrolling, cooperative fetching (using all threads in a block to cache the input data the block needs in shared memory [4]), memory load vectorization [27].

For example, the multi-level tiling method used in [27] tiles the loop nests in an "SSSRRSRS" structure, where "S" means a tile level of space loops and "R" means a tile level of reduction loops. The iteration variables of the first three space loop tile levels are bound respectively to BlockIdx, virtual threads (for reducing shared memory bank conflicts [4]), and ThreadIdx in GPU. The first "R" corresponds to the number of cooperative fetches. The loop tiles on the last four "RSRS" levels are the workload for a single thread. Using this multi-level tiling method on the conv2d in Figure 2, denoted by α, if we tile $\mathcal{L}_\alpha = \{nn, ff, yy, xx, rc, ry, rx\}$ (listed from the outermost loop to the innermost loop) in the way that, *nn, ff, yy, xx* all have 5 tile levels which are $[1, 1, 1, 1, 1]$, $[4, 1, 8, 2, 2]$, $[14, 1, 1, 1, 2]$, $[1, 1, 28, 1, 1]$ respectively, and *rc, ry, rx* all have 3 tile levels which are $[4, 64, 2]$, $[1, 1, 1]$, $[1, 1, 1]$ respectively, then after reordering the loop tiles according to the "SSSRRSRS" structure, fusing loop tiles on the same level and doing thread binding, we can get the major part of the tensor program in Figure 2b. Figure 2c illustrates data loading and parallel computing in that tensor program.

Tensor program search space. Tensor program search space definition is not our focus in this paper. Instead, we use the search space defined in an existing work [27], which has comprehensive coverage of optimizations on tensor programs and enables flexible optimization combinations. Please note that given other categories of search spaces, like the template-based search space in [5], the reuse-based optimization idea (Section 5) can also be applied.

The search space $\mathcal{S}_\alpha$ we use is hierarchical and has two levels: ***sketch*** and ***annotation***. A sketch $\mathcal{K}_\alpha$ is a high-level tensor program structure derived by applying a sequence of transformations to the naïve tensor program of α ($\mathcal{K}_\alpha$ contains information like loop tile structures, how to fuse loops, cache nodes [27]), and there is a set of possible sketches for α, denoted by $\mathbb{K}_\alpha$. $\mathcal{K}_\alpha$ can also be applied to another operator β of the same type. If a tensor program S_α of α using $\mathcal{K}_\alpha$ has more than one loop nest and each loop nest has an individual high-level structure, $\mathcal{K}_\alpha$ is designated as a ***composite sketch***, and each individual high-level structure in $\mathcal{K}_\alpha$ is designated as a ***sub-sketch*** of $\mathcal{K}_\alpha$. Annotations are concrete low-level choices in a sketch (e.g., tile sizes, loop auto unrolling pragmas) [27]. Given $\mathcal{K}_\alpha$, the search space of annotations for α is determined, denoted by $\mathfrak{A}_{\alpha,\mathcal{K}_\alpha}$. Each $S_\alpha \in \mathcal{S}_\alpha$ is therefore determined by a sketch $\mathcal{K}_\alpha \in \mathbb{K}_\alpha$ and a set of annotations $\mathbb{A}_{\alpha,\mathcal{K}_\alpha} \in \mathfrak{A}_{\alpha,\mathcal{K}_\alpha}$, denoted by $S_\alpha = f(\alpha, \mathcal{K}_\alpha, \mathbb{A}_{\alpha,\mathcal{K}_\alpha})$. The size of $\mathbb{K}_\alpha$ is rather small, which can be enumerated [27], e.g., $|\mathbb{K}_\alpha| = 1$ for the conv2d in Figure 2. By contrast, given α and $\mathcal{K}_\alpha$, the annotation search space size $|\mathfrak{A}_{\alpha,\mathcal{K}_\alpha}|$

can be in the order of billions of possible annotation combinations. Therefore, the major efforts of our reuse-based tuner (Section 5) is spent on searching a good $\mathbb{A}_\alpha \in \mathfrak{A}_{\alpha,\mathcal{K}_\alpha}$ efficiently.

3 REUSE OPPORTUNITY

In this section, we illustrate the opportunity of reusing the performant tensor program information, i.e., learning the hardware preference for particular types of tensor programs from the high-performance ones and using it to optimize other operators. Specifically, according to GPU architecture characteristics, three features can be extracted from a tensor program of an operator, which are: (1) B, the number of thread blocks; (2) T, the total number of threads in all thread blocks; (3) M, the total amount of shared memory required by all thread blocks.

In order to analyse the feature relationship between the performant tensor programs of two operators and find the information reuse opportunity for one operator on the other one, we focus on ordered operator pairs (α, β) satisfying the ***reuse conditions*** that the types and sketch sets of α, β are the same, and $\mathcal{L}_\alpha$ dominates, or is dominated by $\mathcal{L}_\beta$. We call such (α, β) a ***reuse pair***.

Definition 2 (Reuse condition). *Given two operators, α, β, there are two reuse conditions:*

(1) $\tau_\alpha = \tau_\beta$ *and* $\mathbb{K}_\alpha = \mathbb{K}_\beta$;
(2) $\mathcal{L}_\alpha = \{\ell_1, ..., \ell_k\}$ *and* $\mathcal{L}_\beta = \{\ell'_1, ..., \ell'_k\}$ *satisfy* $|\mathbb{I}(\ell_i)| \leq |\mathbb{I}(\ell'_i)|$, *for* $i = 1, ..., k$ *or* $|\mathbb{I}(\ell_i)| \geq |\mathbb{I}(\ell'_i)|$, *for* $i = 1, ..., k$.

We want to show the following feature relationship exists between a performant S_α and a performant S_β:

$$r_B(S_\alpha, S_\beta) \in [r_1, r_2], r_B(S_\alpha, S_\beta) \coloneqq B_{S_\alpha}/B_{S_\beta} \quad (1)$$

$$r_T(S_\alpha, S_\beta) \in [1, r_2], r_T(S_\alpha, S_\beta) \coloneqq T_{S_\alpha}/T_{S_\beta} \quad (2)$$

$$r_M(S_\alpha, S_\beta) \in [1, r_2], r_M(S_\alpha, S_\beta) \coloneqq M_{S_\alpha}/M_{S_\beta} \quad (3)$$

where r_B, r_T, r_M denote three feature ratio between S_α and S_β respectively; r_1, r_2 describes the difference between the iteration spaces of α, β ($\mathbb{I}(\mathcal{L}_\alpha), \mathbb{I}(\mathcal{L}_\beta)$), i.e., $r_1 = |\mathbb{I}(r\mathcal{L}_\beta)|/|\mathbb{I}(r\mathcal{L}_\alpha)|, r_2 = |\mathbb{I}(\mathcal{L}_\alpha)|/|\mathbb{I}(\mathcal{L}_\beta)|$. The reason for using r_1, r_2 to define the above relationship is that different operators of the same type do the same type of computation but in different iteration spaces, and hence the iteration space difference can represent a part of the difference between α and β. If such feature relationship exists, then we can make use of it together with a high-performance S_β (which is S^*_β in our method) to narrow down the feature value ranges for α, and only search S_αs whose feature values are within these ranges to save the time of optimizing α.

To show the existence of feature relationships, we collect the tensor programs measured on an NVIDIA GPU (P100) by an existing deep learning compiler [27] during the process of optimizing a set of conv2d operators from a DNN, SSD ResNet50. There are 106 ordered conv2d pairs satisfying the aforementioned conditions. We compare the tensor program features of the two operators for each pair. Before we present statistics, we first define some notations. $\mathcal{G}_\alpha(t) = \{S_\alpha : \mu(S_\alpha) < \mu(S^*_\alpha)/t, S_\alpha \in \mathcal{S}_\alpha\}$ denotes the set of all the tensor program variants of α with t-performance compared with S^*_α, where t is a constant controlling the quality of S_α. We set t to 90% to get the following analysis results. For each ordered operator pair (α, β) satisfying the aforementioned conditions and each $S_\alpha \in \mathcal{G}_\alpha(90\%)$, we compute three feature ratios, r_B, r_T, r_M, between S_α and S^*_β. To learn the relationship between the feature ratios and the iteration space difference, for each above pair (α, β), we compute 3 sets:

$$\Phi_B(\alpha, \beta) = \{S_\alpha : r_B(S_\alpha, S^*_\beta) \in [r_1, r_2], S_\alpha \in \mathcal{G}_\alpha(90\%)\} \quad (4)$$

$$\Phi_T(\alpha, \beta) = \{S_\alpha : r_T(S_\alpha, S^*_\beta) \in [1, r_2], S_\alpha \in \mathcal{G}_\alpha(90\%)\} \quad (5)$$

$$\Phi_M(\alpha, \beta) = \{S_\alpha : r_M(S_\alpha, S^*_\beta) \in [1, r_2], S_\alpha \in \mathcal{G}_\alpha(90\%)\} \quad (6)$$

where r_1, r_2 are defined as above. $|\Phi_B(\alpha, \beta)|$ is the number of performant S_α in $\mathcal{G}_\alpha(90\%)$ such that the ratio of B between it and S^*_β is within the range of $[r_1, r_2]$. Similarly, $|\Phi_T(\alpha, \beta)|$ is the number of performant S_α in $\mathcal{G}_\alpha(90\%)$ such that the ratio of T between it and S^*_β is within the range of $[1, r_2]$. $|\Phi_M(\alpha, \beta)|$ is the number of performant S_α in $\mathcal{G}_\alpha(90\%)$ such that the ratio of M between it and S^*_β is within the range of $[1, r_2]$. We find that, of the 106 conv2d pairs, 87% satisfy that $|\Phi_B(\alpha, \beta)| > 0$; 92% satisfy that $|\Phi_T(\alpha, \beta)| > 0$; 89% satisfy that $|\Phi_M(\alpha, \beta)| > 0$.

These results show that, even if we limit the search space of tensor programs to those whose three features are within a narrowed range based on Equations (1) to (3), denoted by $\mathcal{S}'_\alpha$, $\mathcal{S}'_\alpha$ is still very likely to contain an S_α with relatively good performance (90% inference performance of S^*_α). Therefore, we conclude that, it is possible to learn the hardware preference for particular types of tensor programs from the performant ones and make use of it to speed up the optimization of other operators.

4 FRAMEWORK OVERVIEW

In this work, given a set of distinct operators $\mathcal{A}$ and the frequency w_α of each $\alpha \in \mathcal{A}$, our goal is to reach a trade-off between the operator optimization effect, i.e., the total latency of the operators, and the optimization efficiency, i.e., the total time in finding a good tensor program for each $\alpha \in \mathcal{A}$.

As illuminated in Section 3, to optimize an operator, our ETO framework can reuse the tensor program information of another optimized operator. Based on such information reuse, we design a component in ETO, the ***reuse-based tuner***, to achieve good optimization effect on any reuse pair with shorter search time than the time-consuming compilers. With the reuse-based tuner, given the operators to be optimized, the next task is to determine the reuse relationships among the operators, i.e., selecting reuse pairs for the operators, and we design a component for this task, the ***reuse pair selector***. There are various ways to select the reuse pairs, and different reuse pairs correspond to different search costs. Since we rely on the reuse-based tuner to obtain good operator performance on any reuse pair, the reuse pair selector is used to select the set of reuse pairs with the minimum total search cost when using the reuse-based tuner in ETO. The reuse pair selector considers extra operators as well to further reduce the overall search time.

Figure 3 shows the architecture of ETO. The input is a set of operators to be optimized. The reuse pair selector first generates a set of reuse pairs and then selects a subset of them such that the total search time is minimized. Then the operators involved with no information to reuse are optimized by the backend compiler, whose optimized tensor programs are stored in a database. The reuse-based tuner optimizes the remaining operators involved. It takes a selected reuse pair and the corresponding tensor program from the

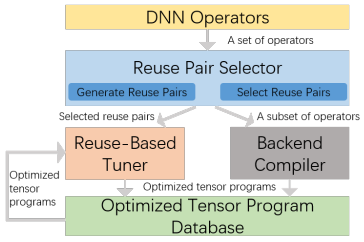


Figure 3: Framework overview of ETO.

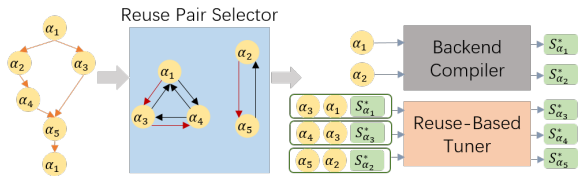


Figure 4: Example workflow of ETO.

database as input each time. The optimized tensor program is also stored in the database for later reuse pairs to reuse its information.

Example 1. *Figure 4 shows how ETO works. The given DNN has 5 different operators: $\alpha_1 - \alpha_5$ (α_1 appears twice). Suppose the reuse pair selector generates 8 reuse pairs from the operators (no extra operators in this example), which correspond to the edges in the second step (e.g., the edge (α_1, α_3) represents the reuse pair (α_3, α_1)), and selects 3 of them (the red edges) to achieve the minimum total search time. Then α_1, α_2 are tuned by the backend compiler, because they have no information to reuse. Their best tensor programs after tuning are $S^*_{\alpha_1}$ and $S^*_{\alpha_2}$, respectively, which are stored in a database. $S^*_{\alpha_1}$ would be used for the two α_1 in the DNN. The reuse-based tuner next takes the reuse pair (α_3, α_1) and $S^*_{\alpha_1}$ as input and generates $S^*_{\alpha_3}$ for α_3, which is also stored. $S^*_{\alpha_3}$ and the reuse pair (α_4, α_3) are then taken by the reuse-based tuner to optimize α_4. α_5 is also tuned by the reuse-based tuner with the reuse pair (α_5, α_2) and $S^*_{\alpha_2}$ as input.*

5 REUSE-BASED TUNER

The reuse-based tuner optimizes an operator by reusing the information of a performant tensor program of another operator. The problem that the reuse-based tuner needs to solve is defined below.

Definition 3 (operator optimization). *Given a reuse pair (α, β) and S^*_β, select a sketch $\mathcal{K}_\alpha \in \mathbb{K}_\alpha$ and a set of annotations $\mathbb{A}_{\alpha,\mathcal{K}_\alpha} \in \mathfrak{A}_{\alpha,\mathcal{K}_\alpha}$, such that $\mu(f(\alpha, \mathcal{K}_\alpha, \mathbb{A}_{\alpha,\mathcal{K}_\alpha}))$ is minimized.*

Given a reuse pair (α, β), since $\mathbb{K}_\alpha = \mathbb{K}_\beta$, whatever $|\mathbb{K}_\alpha|$ is, the reuse-based tuner would directly take the sketch used in S^*_β as the selected $\mathcal{K}_\alpha$. Thus, the left optimization work is to find a good $\mathbb{A}_{\alpha,\mathcal{K}_\alpha}$ for the selected $\mathcal{K}_\alpha$. Specifically, in this section, we first discuss how to find $\mathbb{A}_{\alpha,\mathcal{K}_\alpha}$ for one important type of sketch (Section 5.1), which has a multi-level tile structure. After that, we discuss the optimization work for other types of sketches (Section 5.2).

5.1 Multi-Level Tile Sketch

The multi-level tile sketch is the type of sketch generated by multi-level tiling [27], i.e., the sketch is tiled using the "SSSRRSRS" structure (Section 2). We mainly focus on this type of sketch because (1) the "SSSRRSRS" tile structure can be applied to all the compute-intensive dense operators in deep learning (like conv2d, matrix multiplication) [27], and (2) it enables us to tile each loop flexibly based on the hardware preference understanding and thus results in an extremely large search space of $\mathbb{A}_{\alpha,\mathcal{K}_\alpha}$ (on the order of billions of possible annotation combinations).

From Section 3 we know it is likely that we prune the annotation space $\mathfrak{A}_{\alpha,\mathcal{K}_\alpha}$ according to the feature ratio ranges (i.e., $[r_1, r_2]$, $[1, r_2]$) in Equations (1) to (3) but still find a performant tensor program. However, even the pruned search space $\mathfrak{A}'_{\alpha,\mathcal{K}_\alpha}$ is too large to be enumerated due to the following two reasons: (1) for reuse pair (α, β), the size of the pruned feature space (B, T, M) of α is $O(d_1 \times d_2 \times d_3)$, where d_i is the number of possible values such that the feature ratio is within the specified range in Equations (1) to (3) for each feature, and d_i is related to $|\mathbb{I}(r\mathcal{L}_\beta)|/|\mathbb{I}(r\mathcal{L}_\alpha)|$, $|\mathbb{I}(\mathcal{L}_\alpha)|/|\mathbb{I}(\mathcal{L}_\beta)|$ and $|\mathbb{I}(\ell)|$ of each involved loop ℓ of α; (2) multiple annotation combinations can correspond to the same feature value combination. For the first reason, we devise a hierarchical search method, which decomposes the combination of the 3 features. For the second reason, we use two cost models to guide the tile size selection given feature values.

Before introducing the solution, we first define some concepts related to a tensor program. **BShape**, the block shape, is defined to be the shape of the output tensor a thread block needs to compute. **TShape**, the thread shape, is defined to be the shape of the output tensor a thread needs to compute. The last concept, **LShape**, the load number shape, is defined to be a list of the number of cooperative fetches on each reduction loop. Below is an example.

Example 2. *Suppose a conv2d operator α has an output tensor of shape $[1, 4, 16, 16]$ (the shape dimensions correspond to batch size, kernel number, height, width). Then if the BShape of an S_α is $[1, 4, 8, 8]$, it means in S_α a thread block needs to compute half of the height and the width of each output channel in each batch, and there are 4 thread blocks in total. Given the above BShape, suppose the TShape of S_α is $([1, 2, 1, 1], [1, 1, 4, 4])$. Then the first $[1, 2, 1, 1]$ indicates the second dimension of the BShape is divided into two parts by virtual split [4, 27], and the second $[1, 1, 4, 4]$ is the shape of the output tensor in each subarea after virtual split a thread needs to compute. Therefore a thread computes 32 points in the output tensor, and there are 8 threads in total. Since α has 3 reduction loops (corresponding to input channel number, kernel height, kernel width), suppose LShape of S_α is $[4, 1, 1]$. Then it means a thread block of S_α would do 4 cooperative fetches, and each time only load the data for computing convolution on $1/4$ input channels but the whole kernel height and width.*

We use 2 cost models to guide the selection of BShape and TShape of a tensor program. The ***BShape cost*** is defined as the number of cache line requests by all thread blocks to transfer data they need from global memory to shared memory. The ***TShape cost*** is defined as the number of separate conflict-free shared memory requests by all threads in a block. Based on the cost models, the main idea of the hierarchical search method is as follows. Instead of enumerating all possible combinations of the 3 features B, T, M mentioned in Section 3, for each B, we can compute the best BShape according to the BShape cost. Then we compute a default M (by computing a default LShape) and enumerate T given B, M. For each T, we compute the best TShape according to the TShape cost. We next enumerate M (by enumerating LShape) given B and the best T

for B we get previously. In the above process, other annotations (i.e., auto unrolling and memory load vectorization) are set to default values adaptively, and after it, these annotations are enumerated in order. We repeat these steps for each B and return the best tensor program during the search.

Algorithm 1: Hierarchical Search

Input: A reuse pair (α, β) and S^*_β.
Output: A high-performance S_α.
1: $S^*_\alpha \leftarrow$ None
2: $B_{S^*_\beta}, T_{S^*_\beta}, M_{S^*_\beta} \leftarrow$ analyse S^*_β
3: **for** B in $[B_{S^*_\beta} \times |\mathbb{I}(r\mathcal{L}_\beta)|/|\mathbb{I}(r\mathcal{L}_\alpha)|, B_{S^*_\beta} \times |\mathbb{I}(\mathcal{L}_\alpha)|/|\mathbb{I}(\mathcal{L}_\beta)|]$ **do**
4: $\quad C \leftarrow \emptyset$
5: $\quad \tilde{S}_\alpha \leftarrow$ None
6: $\quad$ BShape$\leftarrow$ GetBShape(B)
7: $\quad$ LShape$\leftarrow$ GetLShape(BShape)
8: $\quad$ **for** T in $[T_{S^*_\beta}, T_{S^*_\beta} \times |\mathbb{I}(\mathcal{L}_\alpha)|/|\mathbb{I}(\mathcal{L}_\beta)|]$ **do**
9: $\quad\quad$ TShape$\leftarrow$ GetTShape(T, BShape, LShape)
10: $\quad\quad C \leftarrow C \cup \{GetS(BShape, TShape, LShape)\}$
11: $\quad \tilde{S}_\alpha \leftarrow$ BestS(C)
12: $\quad C \leftarrow \emptyset$
13: $\quad$ BShape, TShape $\leftarrow$ BShape, TShape of $\tilde{S}_\alpha$
14: $\quad$ **for** LShape s.t. M in $[M_{S^*_\beta}, M_{S^*_\beta} \times |\mathbb{I}(\mathcal{L}_\alpha)|/|\mathbb{I}(\mathcal{L}_\beta)|]$ **do**
15: $\quad\quad C \leftarrow C \cup \{GetS(BShape, TShape, LShape)\}$
16: $\quad \tilde{S}_\alpha \leftarrow$ BestS($\{C \cup \tilde{S}_\alpha\}$)
17: $\quad C \leftarrow$ Tune(different auto unrolling pragmas, $\tilde{S}_\alpha$)
18: $\quad \tilde{S}_\alpha \leftarrow$ BestS($C \cup \{\tilde{S}_\alpha\}$)
19: $\quad$ **for** input X of α **do**
20: $\quad\quad \mathcal{A} \leftarrow$ {vectorization lengths for loading X}
21: $\quad\quad C \leftarrow$ Tune($\mathcal{A}$, $\tilde{S}_\alpha$)
22: $\quad\quad \tilde{S}_\alpha \leftarrow$ BestS($C \cup \{\tilde{S}_\alpha\}$)
23: $\quad S^*_\alpha \leftarrow$ BestS($\{\tilde{S}_\alpha, S^*_\alpha\}$)
24: **return** S^*_α

Algorithm 1 shows the details of the search method. Given a reuse pair (α, β) and S^*_β, S^*_α tracks the best S_α (line 1). We first extract the 3 features from S^*_β: $B_{S^*_\beta}, T_{S^*_\beta}, M_{S^*_\beta}$ (line 2). Then we enumerate all B within the range of $[B_{S^*_\beta} \times |\mathbb{I}(r\mathcal{L}_\beta)|/|\mathbb{I}(r\mathcal{L}_\alpha)|, B_{S^*_\beta} \times |\mathbb{I}(\mathcal{L}_\alpha)|/|\mathbb{I}(\mathcal{L}_\beta)|]$ (line 3). $\tilde{S}_\alpha$ tracks the best S_α with specific B (line 5). For each B, we use GetBShape to get the BShape with the lowest cost (line 6), and GetLShape to get the default LShape (line 7). These two operations are defined as follows.

GetBShape. Given a thread block number, GetBShape returns the min-cost BShape of α, assuming the input a block needs is requested together at once and the cache line request number by each block is the same as that by the first block.

GetLShape. Given a BShape, GetLShape computes an LShape of α, such that the amount of shared memory requested by a thread block is close to that in S^*_β. This is because the maximum amount of shared memory per thread block is limited by the hardware, and such LShape makes the constaint more likely to be satisfied. To simplify the computation, we set only one dimension of the LShape to a value larger than 1. For example, the LShape for a conv2d operator with 3 reduction loops cannot be $[4, 2, 1]$ in our method.

Given the BShape and the LShape, we next enumerate all T within the range of $[T_{S^*_\beta}, T_{S^*_\beta} \times |\mathbb{I}(\mathcal{L}_\alpha)|/|\mathbb{I}(\mathcal{L}_\beta)|]$ (line 8). For each T, we use GetTShape to get the min-cost TShape (line 9), and collect the S_α generated by GetS with the BShape, the LShape and the TShape (line 10). GetTShape and GetS are defined as follows.

GetTShape. Given the thread number of a block, a BShape and an LShape, GetTShape returns the min-cost TShape of α, assuming the data a thread needs would only be loaded once by it from the shared memory.

GetS. Given a BShape, a TShape, and an LShape, GetS outputs an S_α with other annotations set to default values. Specifically, the first 4 tile levels of the "SSSRRSRS" tile structure can be determined directly by the input. For other tile levels, we tile each loop in a way such that, supposing the remaining loop length of it is x, the last tile size d is $\min\{d|x/d \in \mathbb{Z}, d \neq 1\}$, if such d exists; otherwise, we set d to 1. Suppose for a space loop of length 16, the tile sizes on the first 3 levels are 2. The remaining loop length is hence 2, and the final tile sizes related to this loop is $[2, 2, 2, 1, 2]$. For other annotations, we do not use vectorization in cooperative fetching by default, which is controlled by tile sizes on data load loops, so we set the related tile sizes to 1; we set the auto unrolling pragma such that the number of registers needed by a thread, which is approximated by the sum of the number of registers for the thread (1) to store the final results and (2) to store the loaded input data in the innermost loop after unrolling, is close to that in S^*_β. Then we do the first round of measurement, and set $\tilde{S}_\alpha$ to the min-latency S_α in C (line 11). The second round of measurement is on S_αs generated by changing the LShape of $\tilde{S}_\alpha$ such that M of S_α is in the range of $[M_{S^*_\beta}, M_{S^*_\beta} \times |\mathbb{I}(\mathcal{L}_\alpha)|/|\mathbb{I}(\mathcal{L}_\beta)|]$ (line 12-15). We update $\tilde{S}_\alpha$ in line 16. The third round of measurement is about changing the auto unrolling pragma in $\tilde{S}_\alpha$ with other annotations fixed (line 17-18). We use the auto unrolling pragma options in the backend compiler directly. Finally, we change the memory load vectorization length in cooperative fetching of $\tilde{S}_\alpha$ for the inputs of α one by one, and keep on updating $\tilde{S}_\alpha$ (line 19-22). Since GPU global memory instructions only support reading/writing words of 1, 2, 4, 8, or 16 bytes [14], we use 16 bytes to bound the vectorization lengths. For example, if the data type of α is 32-bit float, then we only try 2 and 4 for the vectorization length. When an iteration of B is finished, we update S^*_α using $\tilde{S}_\alpha$ (line 23). In the end, S^*_α is returned.

Example 3. *Let β denote the conv2d in Figure 2 and α denote another conv2d which is the same as β except that its input channel number is 256. Suppose S^*_β is the tensor program in Figure 2b, and we need to optimize α given the reuse pair (α, β). The BShape, TShape, LShape of S^*_β are $[1, 32, 2, 28]$, $([1, 1, 1, 1], [1, 4, 2, 1])$, $[4, 1, 1]$, respectively. We know $B_{S^*_\beta} = 56$, $T_{S^*_\beta} = 56\times224$, $M_{S^*_\beta} = 56\times11264$. The auto unrolling pragma of S^*_β has a value of 1024. $|\mathbb{I}(r\mathcal{L}_\beta)|/|\mathbb{I}(r\mathcal{L}_\alpha)| = 2$. $|\mathbb{I}(\mathcal{L}_\alpha)|/|\mathbb{I}(\mathcal{L}_\beta)| = 1/2$. The range for B is $[28, 112]$, and hence B can be $\{28, 32, 49, 56, 64, 98, 112\}$. We take the iteration of $B = 56$ as an example. When $B = 56$, the best BShape is $[1, 32, 2, 28]$. The default LShape is $[2, 1, 1]$. T can be $\{56\times112, 56\times128, 56\times224\}$. The respective best TShapes are $([1, 1, 1, 1], [1, 4, 2, 2])$, $([1, 1, 1, 1], [1, 2, 1, 7])$, $([1, 1, 1, 1], [1, 4, 2, 1])$. After the first round of measurement, $\tilde{S}_\alpha$ is set to the tensor program with $T = 56 \times 224$ and the TShape being $([1, 1, 1, 1], [1, 4, 2, 1])$, which achieves 3303.31 GFLOPS. In $\tilde{S}_\alpha$, the auto unrolling pragma value is set to 1024 adaptively and the vectorization lengths are set to 1. Then we change the LShape of $\tilde{S}_\alpha$ to*

$[4, 1, 1]$, *because* M *is in the range* $[M_{S^*_\beta}/2, M_{S^*_\beta}]$ *only when LShape is* $[4, 1, 1]$ *or* $[2, 1, 1]$. *After the second round of measurement, we try to update* $\tilde{S}_\alpha$. *We next change the auto unrolling pragma value of* $\tilde{S}_\alpha$ *to* $\{0, 16\}$ *(the full options are* $\{0, 16, 64, 512, 1024\}$, *but only* $\{0, 16\}$ *have different unrolling effect from that of* 1024*). After this, we update* $\tilde{S}_\alpha$ *again according to the results. Finally, we change the vectorization length of loading data to* $\{2, 4\}$ *(the vectorization length of loading the kernel is not tuned because* 2 *or* 4 *cannot divide the number of kernel data points a thread needs to load).* $\tilde{S}_\alpha$ *is updated after this round of measurement. Then we use* $\tilde{S}_\alpha$ *to update* S^*_α.

Analysis. The time complexity of Algorithm 1 is $O(d_1 \times (d_2 + d_3))$, and the space complexity is $O(max(d_2, d_3))$, where d_1, d_2, d_3 are the number of B, T and LShapes (hence M) being enumerated, respectively. The detailed analysis is in the technical report [8].

5.2 Other Sketch Types

Besides the multi-level tile sketch, there are also other types of sketches, e.g., the cross-thread reduction sketch which does cross-thread reduction [27] on an operator naïve tensor program, the two-level tile sketch which fuses space loops and tile the fused loop into 2 levels [27], the unroll-constant-tensor sketch which unrolls loops of constant tensors and tiles the remaining space loops into 2 levels [27]. Sketches on GPU all satisfy the general pattern (Section 2) of partitioning a computation task into subtasks by loop tiling and assigning them to threads. In fact, they can be regarded as special cases of the multi-level tile sketch and the search methods for them are simplified versions of Algorithm 1. We have summarized the common cases in different types of sketches that would simplify the search process, and each sketch may involve more than one case. We provide more details in our technical report [8].

For an operator with a composite sketch, we deal with the sub-sketches one by one. If a sub-sketch is a multi-level tile one, we apply Algorithm 1 on it. Otherwise, the search method is a simplified version of Algorithm 1 as mentioned above. For the search method of each sub-sketch, a new major loop nest is chosen from the related loop nests in the naïve tensor program to decide the feature value ranges (Equations (1) to (3)).

6 REUSE PAIR SELECTOR

With the reuse-based tuner used in ETO, the reuse pair selector solves the following problem.

Definition 4 (reuse pair selection). *Given a set of operators* $\mathcal{A}$, *determine the reuse pairs for* $\mathcal{A}$ *to minimize the total time needed by the backend compiler and the reuse-based tuner to optimize* $\mathcal{A}$.

Given all the possible reuse pairs of the operators to be optimized, the search time of each reuse pair on the reuse-based tuner, and the search time of each operator on the backend compiler, we can build a directed graph G with the operators as nodes and the reuse pairs as edges. We then add a special root to G with an edge from it to each operator, and such an edge represents the operator being tuned by the backend compiler. The edge weights are the corresponding search time. G is designated as an ***optimization choice (OC) graph*** of the related operators. Hence the reuse pair selection problem (Definition 4) can be regarded as a ***min-cost arborescence problem*** on G, which can be solved in polynomial time. Moreover, we can further reduce the total search time of ETO by adding some extra operators. The details would be presented in Section 6.2. Below we first introduce the method of estimating the search time.

6.1 Search Time Estimation

Since we cannot obtain the actual search time before running the optimization method, to estimate the search cost, we use the number of tensor programs which are measured on the hardware. For the backend compiler to tune an operator, the search cost can be set to the maximum number of measurement trials of it (many works [5, 27] require us to limit the number of measurement trials because of the huge search space). However, for the reuse-based tuner to tune an operator, the exact number of measurements trials still cannot be computed, because for example we do not know the ranges in Equations (1) to (3) without S^*_β in Algorithm 1.

On the other hand, given the sizes of BShape, TShape, and LShape in the reused tensor program, the number of measurement trials by Algorithm 1 can be estimated. Note that to approximate the number of auto unrolling pragmas and vectorization lengths (line 17, 20 in Algorithm 1), we assume all the possible choices of them would be tested. Based on all possible combinations of the sizes of BShape, TShape, and LShape in the reused tensor program, we can compute the average estimated measurement trial number by Algorithm 1 for a reuse pair, and we observe that this result is close to the actual number of measurement trials. We have a similar observation for the simplified versions of Algorithm 1 (Section 5.2) as well. To verify that our observation holds for different operator types, we have randomly sampled 40 reuse pairs, and for each reuse pair we plotted the curve of the estimated search cost vs. the assumption about the sizes of BShape, TShape, and LShape in the reused tensor program. Figure 5 provides an example curve for a reuse pair with the multi-level tile sketch where our observation holds. The other 39 curves can be found in the technical report [8], and they are very similar to Figure 5 since the definition of feature value ranges (Equations (1) to (3)) and the search methods are independent of operator types (like Algorithm 1 works for all types of operators with the multi-level tile sketch).

Based on the above observation, we can compute the average estimated measurement trial number to estimate the search cost. For a reuse pair (α, β) where $|\mathbb{K}_\alpha| > 1$, we first use the maximum possible search cost of it in the reuse pair selection. After we observe the selected sketch for an operator with the same type and sketches as α (this can be done after optimizing the first such operator according to the reuse pair selection result), we can reestimate the search cost and then reselect relevant reuse pairs.

One practical issue of this estimation method is that it can be very costly, because of the large sample space (sizes of BShape, TShape, and LShape) and the large number of reuse pairs (after we add extra operators as aforementioned). Hence we only randomly sample a small part of the sample space to approximate the exact average estimated search cost. We tested this sampling method on the above 40 reuse pairs, and the ratio of the average estimated search cost over the actual measurement trial number is 1.04 on average, ranging in $[0.54, 1.62]$. Hence, the empirical results indicate that sampling will not cause extreme estimation error, and the performance of our search cost estimation method is acceptable.

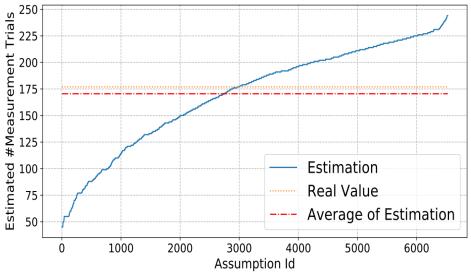
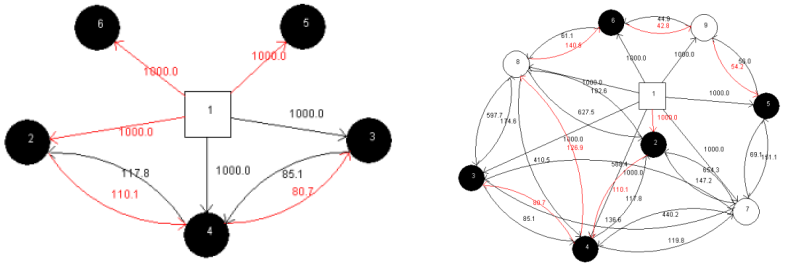


Figure 5: Estimated # Measurement Trials.



(a) OC graph without bridge operators. **(b) OC graph with bridge operators.**

Figure 6: Improve search efficiency with bridge operators.

6.2 Bridge Operators

Only considering the reuse pairs from the operators we need to optimize may be insufficient to improve the overall search efficiency of ETO significantly. For example, for the 5 different batch matmul operators in BERT-Base [7], there are only 4 possible reuse pairs. Figure 6a shows the OC graph G to find the min-cost arborescence on when selecting reuse pairs, where node 1 is a special root and the other nodes correspond to the operators. In G, node 5,6 are only connected to node 1, i.e., there is no reuse pair for them. The edges of the min-cost arborescence are shown in red, and the total cost of them is 3190.8, lower than the total cost without reuse pairs (5000). However, there is still room to reduce the total search cost.

To address the limitation, we consider extra operators, designated as ***bridge operators***. Specifically, if operators α, β cannot directly reuse each other, a bridge operator γ can be added for them such that α, γ satisfy the reuse conditions and so do β, γ. In this way, α and β build an indirect reuse relationship with each other. Note that there can be many choices of γ. For example, α, β are 2 conv2d operators with $\mathbb{I}(\mathcal{L}_\alpha)$ of shape $[1, 64, 14, 14, 64, 3, 3]$ and $\mathbb{I}(\mathcal{L}_\beta)$ of shape $[1, 64, 28, 28, 64, 1, 1]$. The loop nest domination condition (condition 2 in Definition 2) is violated, because the third dimension of $\mathbb{I}(\mathcal{L}_\alpha)$ is smaller than that of $\mathbb{I}(\mathcal{L}_\beta)$ ($14 < 28$), but the last dimension of $\mathbb{I}(\mathcal{L}_\alpha)$ is larger than that of $\mathbb{I}(\mathcal{L}_\beta)$ ($3 > 1$). Suppose $\mathbb{K}_\alpha = \mathbb{K}_\beta$. Then many operators can serve as a bridge between α and β, e.g., a conv2d operator γ_1 with $\mathbb{K}_\alpha = \mathbb{K}_{\gamma_1}$ and $\mathbb{I}(\mathcal{L}_{\gamma_1})$ of shape $[1, 64, 14, 14, 64, 1, 1]$, or a conv2d operator γ_2 with $\mathbb{K}_\alpha = \mathbb{K}_{\gamma_2}$ and $\mathbb{I}(\mathcal{L}_{\gamma_2})$ of shape $[1, 64, 12, 12, 64, 1, 1]$ (or $[1, 64, 28, 28, 64, 3, 3]$). Therefore, it is necessary to only focus on interesting bridge operators to limit the bridge operator search space. Intuitively, the search time of a reuse pair (α, β) will be smaller if the difference between $\mathcal{L}_\alpha$ and $\mathcal{L}_\beta$ is smaller, and an operator with smaller iteration space size of the major loop nest is easier to be optimized (in terms of both smaller search space and less measurement time to get its latency). Hence we would only consider γ_1 above as the bridge operator for α, β. The bridge operator is formally defined below.

Definition 5 (bridge operator). *Given two operators, α with $\mathcal{L}_\alpha = \{\ell_1, ..., \ell_k\}$ and β with $\mathcal{L}_\beta = \{\ell'_1, ..., \ell'_k\}$, where $\tau_\alpha = \tau_\beta$ and $\mathbb{K}_\alpha = \mathbb{K}_\beta$, the bridge operator γ for them is an operator such that:*

- $\mathbb{I}(\mathcal{L}_\gamma)$ *is of shape* $[min(|\mathbb{I}(\ell_1)|, |\mathbb{I}(\ell'_1)|), ..., min(|\mathbb{I}(\ell_k)|, |\mathbb{I}(\ell'_k)|)]$;
- $\tau_\gamma = \tau_\alpha$ *and* $\mathbb{K}_\gamma = \mathbb{K}_\alpha$.

For the parameters of γ which cannot be determined by $\mathcal{L}_\gamma$, we set them to default values, e.g., 1 as the default stride of a conv2d operator.

In the content below, $\mathcal{A}$ denotes an operator set without bridge operators; $\overline{\mathcal{A}}$ and $\mathcal{P}$ denote an operator set expanded from $\mathcal{A}$ with bridge operators and a set of selected reuse pairs, respectively; $\mathcal{A}'$ denotes the operators involved in $\mathcal{P}$. The total search cost in this case is the sum of (1) the search costs for the backend compiler to tune the operators in $\mathcal{A}'$ with no operator to reuse tensor program information of, and (2) the search costs for the reuse-based tuner to tune other operators in $\mathcal{A}'$ according to $\mathcal{P}$. To select reuse pairs, we also build an OC graph $G = (V, E)$ of $\overline{\mathcal{A}}$, but the reuse pair selection problem is now regarded as a ***directed steiner tree problem*** on G, which can be solved by an existing fast $|\mathcal{A}|$-approximation algorithm [23] in $O(|\overline{\mathcal{A}}||E||\mathcal{A}|^2)$ time.

A practical issue is the cost of enumerating all the bridge operators. Given $\mathcal{A}$, we can keep on adding bridge operators until we get the closure of $\mathcal{A}$ under this operation, and the closure size is finite: if we define the k-hop bridge operator set $\mathcal{B}_k$ to be all the bridge operators for $\forall \alpha, \beta \in \mathcal{A} \bigcup_{i=1}^{k-1} \mathcal{B}_i$ and $\mathcal{B}_k \cap (\mathcal{A} \bigcup_{i=1}^{k-1} \mathcal{B}_i) = \emptyset$ ($k \in \mathbb{N}^+$), then for operators whose major loop nests have m loops, the bridge operator can be at most $(m-1)$-hop. However, even if $m = 2$, there can be as many as $O(n^2)$ bridge operators and $O(n^4)$ reuse pairs, where $n = |\mathcal{A}|$. In fact, we found it takes 1.5 hours to compute all the bridge operators for a set of 167 conv2d operators and get all the reuse pairs of the expanded operator set. To control the time for ETO to add bridge operators and generate reuse pairs, given $\mathcal{A}$, we need to find the smallest set of bridge operators such that the total search cost according to the later selected reuse pairs $\mathcal{P}$ is minimized.

The problem of finding such a minimum set of bridge operators is NP-hard. The proof can be done by reducing the directed steiner tree problem to it. It is worth noting that we have no requirement on the search cost c of an operator, except that $c \in \mathbb{R}^+$. Specifically, (1) for reuse pairs $P_1 = (\alpha, \beta), P_2 = (\beta, \alpha)$, the respective search costs $c(P_1)$ and $c(P_2)$ can be different; (2) the search cost is not guaranteed to satisfy the triangle inequality on the OC graph G, i.e., given three edges $e_1 = (v_1, v_2)$, $e_2 = (v_2, v_3)$ and $e_3 = (v_1, v_3)$ on G, and their respective weights c_1, c_2, c_3, $c_1 + c_2 < c_3$ may hold. The proof can be found in [8]. Therefore, ETO heuristically only adds 1-hop bridge operators as they can result in a significant increase to the possible indirect reuse relationships, which can be done in $(|\mathcal{A}|^2)$ time. The example in Figure 6b shows that with 1-hop bridge operators (the white circular nodes), the number of reuse pairs is increased from 4 to 24, and the minimum search cost achieved by ETO decreases from 3190.8 to 1555.2.

7 EXPERIMENTS

ETO is implemented mainly in Python based on Ansor [27]. The algorithm for reuse pair selection with bridge operators is implemented in Java based on [23, 24]. Ansor [27] is used as the backend compiler. We evaluate the effectiveness and efficiency of ETO in optimizing DNN operators. Specifically, compared with the existing methods, we have tested two points: (1) for an operator, whether ETO has close or better optimization effect in a shorter time using the reuse-based tuner; (2) for a set of operators, whether ETO has close or better overall optimization effect in a shorter time. The details about these two tests are presented in Section 7.1 and Section 7.2 respectively. All the experiments are run on a machine with a 12-core Intel Xeon E5-2690 CPU, an NVIDIA GPU (Tesla P100) and 220 GB of RAM. The data type is float32 in all the evaluations.

7.1 Performance on Single Reuse Pair

Workloads. We evaluate ETO on 10 types of deep learning operators, which are: 1D, 2D, and 3D convolution (C1D, C2D, C3D respectively), batch matrix multiplication (BMM), group convolution (GRP), depthwise convolution (DEP) [12], transposed 2D convolution (T2D) [15], capsule 2D convolution (CAP) [11], Winograd 2D convolution (WIN) [13], and frobenius norm (FRB). For each type τ, we collect operators from popular deep learning models or from the experiments of Ansor [27], and set their batch sizes to be 1 and 16 to get a set $\mathcal{A}_\tau$: the CAP and FRB operators are from [27], and operators of the other types are randomly sampled, respectively, from an operator set in Table 2 (the WIN operators are from the operators in ResNet50 which can be computed using the Winograd algorithm) and its 1-hop bridge operators. Then from all the reuse pairs of $\mathcal{A}_\tau$ with estimated search cost lower than the maximum of 1000 and the operator search space size, we randomly sample 40 reuse pairs as the test cases, since we set the maximum measurement trial number to be 1000 for the backend compiler, and a reuse pair with higher search cost would not be selected by ETO. For all types of operators we collect, except FRB and WIN, their possible sketches are multi-level tile sketches (Section 5.1). Each FRB operator has two possible sketches: a cross-thread reduction sketch; a composite one consisting of two two-level tile sub-sketches. The composite sketch of the collected WIN operators has four sub-sketches: two unroll-constant-tensor sub-sketches, a multi-level tile sub-sketch and a two-level tile sub-sketch.

Settings. The baselines are **Ansor** (commit: 3635945) [27] and **TopK** of ST [19]. For each reuse pair (α, β), we use Ansor to tune α, β with up to 1,000 measurement trials respectively (the same as in [27]), and store all the tensor programs of β (S_βs) during the search. TopK directly applies the respective sketches and annotations of K S_βs found by Ansor to α in an iteration, and repeats this process from high-performance S_β to low-performance S_β until it finds an S_α which can run, and outputs the best S_α during the search. In TopK, K is set to 3 (the same as in ST of TVM [19]). ETO optimizes α based on the best S_β found by Ansor.

Metrics. To measure the performance of ETO on a single reuse pair, we adopt two metrics: throughput and search time (to optimize an operator). The throughput of an operator is the reciprocal of its latency. For a reuse pair (α, β), the throughput of it refers to the throughput of α. We report 2 types of the search time for Ansor:

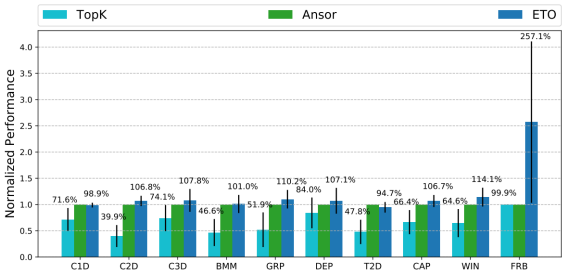


Figure 7: Inference performance of an operator.

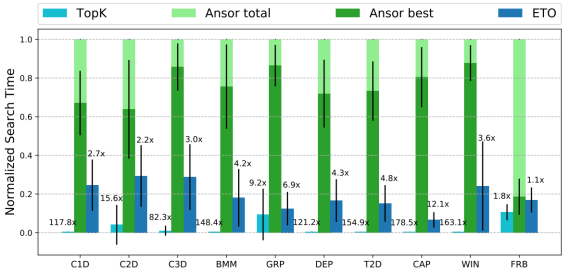


Figure 8: Search time to optimize an operator.

"Ansor total", "Ansor best" are the total search time and the time to get the maximum throughput for Ansor, respectively.

Results Analysis. We normalize the throughput to the throughput optimized by Ansor for each reuse pair, and report the average normalized throughput of 40 reuse pairs in Figure 7, where the value on a bar is the average result for the operator type. The search time is normalized to the search time of "Ansor total" for each reuse pair, and Figure 8 shows the average result of 40 reuse pairs for each operator type, where the value on a bar is the speedup relative to the average normalized search time of "Ansor best". The error bars in Figure 7 and Figure 8 show the standard deviation of the normalized throughput and the normalized search time of 40 reuse pairs for each operator type, respectively. Overall, ETO finds tensor programs having close or better inference performance compared with Ansor, with the normalized throughput ranging from 94.7%, 257.1%, but 1.1 – 12.1× faster compared with "Ansor best". Given that the reuse pairs are randomly sampled and their costs vary from small to large, these results show that the reuse-based tuner performs well in terms of the operator throughput on reuse pairs with different costs, and on the other hand, with the information of another performant tensor program, the reuse-based tuner can indeed speed up the searching for the best performing tensor program of an operator. On FRB, ETO outperforms Ansor by 257.1% because we regard the larger loop nest of it, which computes the sum of squares, as the major loop nest, and therefore make it possible to partition it into more subtasks to run in parallel, while Ansor tiles this loop nest depending on the smaller loop nest after it which computes the square root, resulting in less parallelism. As for the search time, the relative speedup is small on FRB, because Ansor can already finish the searching fast due to the limited search space of FRB. For other test cases, the search spaces of ETO and Ansor are the same. On T2D, ETO achieves 94.7% performance compared with Ansor, but is 4.8× faster than "Ansor best". In fact,

Ansor can only find tensor programs with 59.7% normalized performance using the same search time as ETO on T2D, which means ETO is faster than Ansor in searching performant tensor programs for T2D. For most of the other types of operators, ETO has higher normalized throughput than Ansor (except C1D, where the normalized throughput is 98.9% and very close to 100%). For the WIN operator involving a composite sketch, since the loop nest with the multi-level tile sub-sketch accounts for a large percentage of its latency, the search simplification for other sub-sketches does not degrade the effectiveness of ETO: the average normalized performance is 114.1%. Because the tuning on sub-sketches is conducted individually, ETO is 3.6× faster than "Ansor best", even with the composite sketch.

ETO outperforms TopK for all the operator types in terms of the operator throughput. TopK can sometimes find relatively good tensor programs on average, e.g., on DEP and FRB, but it fails to do this on most of the other types of operators, e.g., the 39.9% average normalized throughput on C2D. For FRB, the search space defined by Ansor is much more smaller and simpler than other types of operators. The best tensor programs of the tested FRB operators in the search space of Ansor are the same in terms of their sketches and annotations: using the cross-thread reduction sketch, the same thread block number (i.e., 1) as well as the same thread number in a block, and not being auto-unrolled due to the large fused reduction loop. As a result, TopK finds exactly the same programs as those by Ansor for the tested FRB operators. The normalized performance is 99.9% instead of 100% because of the noise when measuring operator latency on hardware. We evaluate TopK on another data set for FRB by setting larger batch sizes and smaller reduction loops (the data set description can be found in [8]), where the respective best tensor program in the search space of Ansor are different for the operators, and the average normalized performance of it drops to 78.9%, while that for ETO is 168.2% (although TopK is 1.5× faster than ETO). For the tested DEP operators, although the average normalized performance is 84.0%, the worst and the best normalized performance is 12.8% and 182.4% respectively, the reason may be that in the tested reuse pairs, the operators in a reuse pairs are not guaranteed to be similar for TopK to perform well on (while the normalized performance by ETO is in the range of [87.9%, 192.4%]). This means TopK's performance is not stable and may suffer on some cases. In fact, the standard deviation of the normalized throughput of TopK is larger than that of ETO on all types of operators except FRB (the normalized performance of ETO ranges between 98.6% and 430.3% on FRB). The reason is TopK directly applies the optimization steps in the given performant tensor programs, while the reuse-based tuner of ETO tries to learn the hardware preference for particular types of tensor programs and can find better annotations. As for the search time, since in most cases, TopK can get a valid tensor program which can run successfully from the first 3 measurement trials, its search efficiency is much higher than both Ansor and ETO. The big standard deviation for some operator types (e.g., C2D, GRP) is because it takes TopK hundreds of measurement trials to get a valid tensor program for some reuse pairs. On GRP, TopK even fails to find a valid tensor program for 3 reuse pairs (the corresponding normalized throughput for them is 0).

Table 2: Operator set summary.

DNN	#Operators (with type τ and sketches $\mathbb{K}$)
ResNet50	20 C2D ($\mathbb{K}_1$)
MobileNetV2	20 C2D ($\mathbb{K}_1$), 1 C2D ($\mathbb{K}_2$), 10 DEP ($\mathbb{K}_3$)
ResNeSt50	22 C2D ($\mathbb{K}_1$), 4 C2D ($\mathbb{K}_2$), 7 GRP ($\mathbb{K}_4$)
1D-IR	35 C1D ($\mathbb{K}_5$)
R(2+1D)	25 C3D ($\mathbb{K}_6$)
DCGAN	5 T2D ($\mathbb{K}_7$)
BERT-Base	5 BMM ($\mathbb{K}_8$)
ResNet50(L)	40 C2D ($\mathbb{K}_1$)
BERT(L)	36 BMM ($\mathbb{K}_8$)

7.2 Performance on Operator Set

Workloads. The workloads are divided into 2 groups: single DNN and multiple DNNs. For single DNNs, we prepare: ResNet50 [10], MobileNetV2 [16] and ResNeSt50 [25] for image classification, 1D variant of Inception-ResNet-V2 (1D-IR) [9] for tasks like 1D data classification, R2Plus1D-ResNet18 [21] (R(2+1D)) for action recognition, DCGAN [15] generator for image generation, and BERT-Base [7] (sequence length is 128) for language understanding. We report the results when batch size is 1 on the above DNNs. For multiple DNNs, we prepare 2 DNN sets: ResNet50 with batch size being 1, 16 (ResNet50(L)); BERT-Base and BERT-Large with batch size 1, 16, and sequence length being 64, 128 (BERT(L)). For each test case (a DNN or a set of DNNs above), we get an operator set, where each operator has a frequency, i.e., the number of times it appears in the test case. Table 2 summarizes these operator sets (due to space limitations, the frequency information is not shown).

Settings. We use **Ansor** (commit: 3635945) [27] and **ST** [19] as the baselines (the TopK component of ST is as mentioned in Section 7.1, and the similarity rate threshold of ST is 0.01 as in [19]). We let Ansor run up to 1,000 $\times|\mathcal{A}|$ measurement trials on each operator set $\mathcal{A}$. ETO and ST use Ansor as the backend compiler to tune up to 1,000 measurement trials for an operator with no operator to reuse tensor program information of. The sample size in the search cost estimation for ETO is 10.

Metrics. Two metrics are used: throughput, search time (to optimize a set of operators). The throughput of an operator set is the reciprocal of the weighted latency sum of all operators, where the weight of an operator is its frequency. For the search time of Ansor, we report 3 types of it to better compare Ansor and ETO: "Ansor total", "Ansor best", "Ansor same" are the total search time, the time to get the maximum throughput, and the time to get the same throughput as that by ETO for Ansor, respectively. The search time of ETO does not contain the preprocessing time of reuse pair generation and selection as it is negligible compared with the time of searching performant tensor programs.

Results. We normalize the throughput to that by Ansor, and report the results in Figure 9. Figure 10 shows the search time of compared methods, where the speedup is relative to the search time of "Ansor best". Overall, the throughput by ETO is close to that by Ansor in all cases, with the normalized performance ranging from 91.4% to 101.3%, but ETO is $3.1-16.1\times$ faster compared with "Ansor best". Even compared with "Ansor same", ETO is still $1.8-6.2\times$ faster. On DCGAN and BERT-Base, the throughput by ETO is higher than that by Ansor, so the corresponding results of "Ansor same" is not shown in Figure 10. The reason for this big speedup is that after

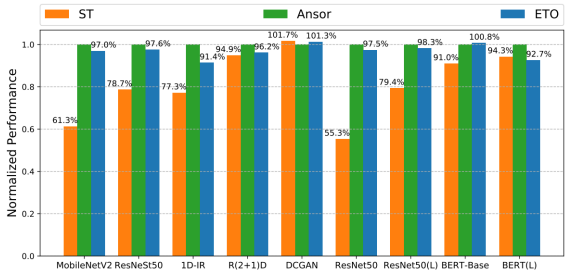


Figure 9: Inference performance of an operator set.

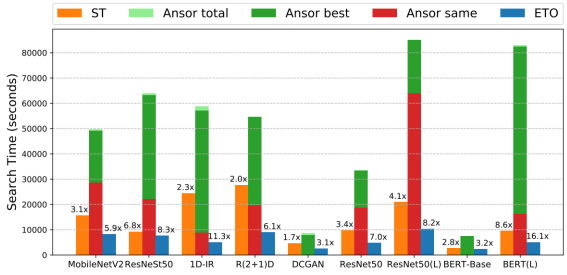


Figure 10: Search time to optimize a set of operators.

the reuse pair selection, only 1 or 2 operators for the operators with the same type and the same sketch set are optimized by the backend compiler in each operator set, and therefore ETO can save a lot of search time with the reuse-based tuner. For the absolute difference of the weighted latency sum between ETO and Ansor, in most cases, it is less than 0.5 ms; on R(2+1)D it is 1.9 ms; on BERT(L) it is 32.9 ms. The reason for the big 32.9 ms difference on BERT(L) is that the frequency of an operator will magnify the performance loss in the final weighted latency sum. In fact, the single operator throughput ratio of ETO over Ansor is 118.7% on average and ranges between 80.6% – 225.1%, but the operator frequency ranges from 12 to 96. By comparing the results on ResNet50(L) with those on ResNet50, and comparing the results on BERT(L) with those on BERT-Base, we can see that the more operators with the same type and the same sketch set to be optimized, the higher the speedup ETO can bring, and there can be no or just small drop of the overall normalized performance (due to the operator frequency).

The normalized performance by ST ranges from 55.3% to 101.7%. Although on DCGAN and BERT(L), ST performs slightly better than ETO with 1.004× and 1.018× throughput respectively, ETO outperforms ST in most cases, with up to 1.8× throughput, because ST's similarity model is not effective enough (Section 1) and the reuse-based tuner has optimization effectiveness superiority over ST's TopK reuse method. For the search time, despite that ST only needs 3 measurements to find a runnable tensor program for an operator using TopK in most cases, ETO is still 1.1 – 4.8× faster than ST on all operator sets. This is because ST finds the maximal cliques to cluster operators (the number of clusters can be large because of the similarity rate threshold) and operators can only reuse the tensor program of the selected representative operator in the same cluster, while in ETO the information of the tensor program found by the reuse-based tuner can be reused by other operators again, and we bridge operators to provide more reuse opportunities, so we can send fewer operators to the time-consuming backend compiler and save more search time.

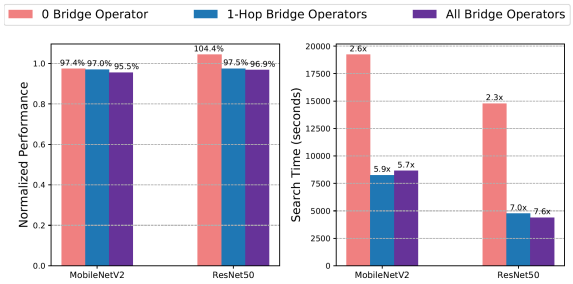


Figure 11: Inference performance and search time

Ablation study. We run three variants of ETO on MobileNetV2 and ResNet50: "0 Bridge Operator" means we do not generate bridge operators in the reuse pair selector; "1 Hop Bridge Operators" means we only generate 1-hop bridge operators (this is the method we adopt); "All Bridge Operators" means we generate all the bridge operators. MobileNetV2 has 81 1-hop bridge operators and 121 bridge operators in total. ResNet50 has 44 1-hop bridge operators and 49 bridge operators in total. Figure 11 shows the results: the left figure shows the throughput normalized to Ansor, and the right figure shows the search time, where the speedup is relative to "Ansor best". The normalized inference performance of the three variants is close. On MobileNetV2, the normalized throughput difference is less than 2%, and the absolute difference of the weighted latency sum is less than 0.01 ms. On ResNet50, the normalized throughput difference between "1 Hop Bridge Operators" and "All Bridge Operators" is only 0.6%, and "0 Bridge Operator" performs about 7% better than the other two variants, because "0 Bridge Operator" uses the backend compiler to optimize 8 operators out of the 20 conv2d in ResNet50 and gets better tensor programs for some operators. The absolute difference of the weighted latency sum among the variants on ResNet50 is also small, i.e., 0.2 ms. As for the search time, "1 Hop Bridge Operators" and "All Bridge Operators" are both faster than "0 Bridge Operator", with 2.2 – 3.3× speedup, thanks to the increased reuse opportunities brought by the bridge operators. The speedup difference between "All Bridge Operators" and "1 Hop Bridge Operators" is not significant. This result shows that 1-hop bridge operators already generate sufficient reuse pairs which are likely to be chosen to save the overall search time on MobileNetV2 and ResNet50. Compared with "1 Hop Bridge Operators", the search time of "All Bridge Operators" is shorter on ResNet50 but longer on MobileNetV2 because we cannot estimate the cost of each reuse pair perfectly.

8 RELATED WORKS

Automatic tensor program generation. There are different search space representation methods for automatic tensor program generation. Halide [1] defines the search space of tensor program variants to be transformation option sequences on the unoptimized tensor program, and uses a backtracking tree search algorithm based on beam search together with a learned cost model to find the best sequence. A recent work [3] improves [1] by partitioning the search space of options into buckets according to the structural

similarity and randomly selecting candidates from each bucket to be evaluated by the cost model. TVM [4, 5] uses manual templates with tunable parameters to define the search space, and searches the best parameters to optimize an operator. FlexTensor [28] automatically generates a search space of an operator by enumerating the transformation steps in a specific order. Ansor [27] defines a hierarchical search space with two levels: sketch and annotation (as we have introduced in Section 2). Tensor Comprehensions (TC) [22] leverages polyhedral compilation and auto-tuning (on specified parameters) to optimize operators. We use the hierarchical search space of Ansor in this paper, because it is large and enables flexible optimization combinations. However, the reuse-based optimization idea (learning hardware preference for particular types of tensor programs from the performant ones) can also be applied to other search spaces, because GPU tensor programs all satisfy the general pattern (mentioned in Section 2).

Search-based compilation acceleration. A common workflow of search-based compilation [1, 3–5, 27, 28] is an iterative process, i.e., searching candidate tensor programs based on a cost model, measuring the programs on hardware, and updating the cost model with the measurement results, so existing works focus on more efficient search algorithms or better cost models.

Halide uses importance sampling [1] and hierarchical sampling [3] to select candidates tensor programs for measurement. Another work, CHAMELEON [2], expedites the convergence of optimization with reinforcement learning and reduces the number of hardware measurements with adaptive sampling. FlexTensor [28] combines a simulated annealing based method with Q-learning in search. An evolutionary search method designed specifically for tensor programs is used by Ansor [27]. However, these methods do not consider reusing the information from a performant tensor program in search. Selective Tuning (ST) [19] reuses the transformation steps of performant tensor programs from other operators directly in optimization, however, their performance may be limited by the design of the operator similarity model and the number of similar operator groups in the given operators. Experiments show that ETO is more effective than it and has higher overall optimization efficiency. We also conducted more experiments on the variants of ST by changing the parameters in ST, and the results can be found in the technical report [8]. TC [20, 22] uses a compilation cache to reuse the tensor programs in a way similar to ST: it can directly apply the transformation steps in an optimized tensor program of another similar operator, or alternatively it can use that set of transformation steps as a starting point of its genetic search algorithm. Due to lack of enough details of the compilation cache usage in TC, we did not include it in the current experiments.

As for better cost models, transfer learning is used by autoTVM [5] to make use of the historical data collected during optimizing previous workloads, so as to speed up the optimization. Halide [1, 3] supports the one-shot mode of searching, i.e., searching a set of tensor programs completely depending on a pretrained cost model without hardware measurement and outputing the tensor program ranked best by the cost model. Using this mode, the search can finish in seconds, but the optimized throughput is worse than searching with hardware measurements, because of the limited accuracy of the cost models. A recent work [18] designs a cost model, which can be pretrained, using LSTM over engineered features to accurately estimate the expected performance of a partial schedule. Given the cost model, it makes a sequence of optimization decisions greedily, and outperforms [1] in effectiveness. This method can be combined with our reuse-based tuner by using the trained cost model to rank the generated tensor programs, instead of hardware measurements. To save the overall search time of a set of optimization tasks, Ansor [27] and DynaTune [26] dynamically allocate time slots to tasks, instead of sequentially optimizing them. Since ETO does not run in iterations, there is no task scheduling issue. Experiments show ETO searches performant tensor programs more efficiently than Ansor despite its task scheduling strategy.

9 CONCLUSION AND FUTURE WORK

Efficient execution of DNNs is important in many applications. In this work, we propose ETO, which (1) reuses the information of performant tensor programs to speed up operator optimization, and (2) determines the information reuse relationships among operators with the minimum total search cost, reaching a trade-off between operator performance and optimization efficiency. Furthermore, we have introduced a concept, bridge operators, to offer indirect reuse opportunities among operators. Compared with the existing works, the experimental results confirm the effectiveness and efficiency of ETO in optimizing operators of DNNs.

One limitation of ETO is that the operator frequency is not considered when optimizing a set of operators. How to change the search method and schedule the tuning time to improve the overall inference performance is an interesting problem. Besides, ETO takes several minutes on average to optimize an operator, so another interesting problem is how to make use of more powerful cost models to further reduce the search time on a single operator. We can also consider combining our search method with existing methods to improve the optimization effectiveness and efficiency, like using the tensor programs in our pruned search space as the starting points of the genetic search algorithm in [22, 27]. Lastly, we can explore the possibility of incorporating ETO into existing systems which support dynamic neural networks.

ACKNOWLEDGMENTS

Yanyan Shen is partially supported by Shanghai Municipal Science and Technology Major Project(2021SHZDZX0102) and SJTU Global Strategic Partnership Fund (2021 SJTU-HKUST). Yue Wang is partially supported by China NSFC (No. 62002235) and Guangdong Basic and Applied Basic Research Foundation (No. 2019A1515110473). Lei Chen's work is partially supported by National Key Research and Development Program of China Grant No. 2018AAA0101100, the Hong Kong RGC GRF Project 16202218, CRF Project C6030-18G, C1031-18G, C5026-18G, RIF Project R6020-19, AOE Project AoE/E-603/18, Theme-based project TRS T41-603/20R, China NSFC No. 61729201, Guangdong Basic and Applied Basic Research Foundation 2019B151530001, Hong Kong ITC ITF grants ITS/044/18FX and ITS/470/18FX, Microsoft Research Asia Collaborative Research Grant, HKUST-NAVER/LINE AI Lab, HKUST-Webank joint research lab grants. The corresponding author of this paper is Yanyan Shen.

REFERENCES

[1] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, et al. 2019. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)* 38, 4 (2019), 1–12.

[2] 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).

[3] Luke Anderson, Andrew Adams, Karima Ma, Tzu-Mao Li, and Jonathan Ragan-Kelley. 2020. Learning to Schedule Halide Pipelines for the GPU. *arXiv preprint arXiv:2012.07145* (2020).

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 578–594.

[5] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to optimize tensor programs. *arXiv preprint arXiv:1805.08166* (2018).

[6] 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).

[7] 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).

[8] Jingzhi Fang, Yanyan Shen, Yue Wang, and Lei Chen. 2021. *ETO: Accelerating Optimization of DNN Operators by High-Performance Tensor Program Reuse.* Technical Report. https://github.com/Experiment-code/ETO

[9] hanxuh hub. 2020. *1D-Inception-ResNet-V2-Model.* hanxuh-hub. Retrieved October 20, 2021 from https://github.com/hanxuh-hub/1D-Inception-ResNet-V2-Model

[10] 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.

[11] Geoffrey E Hinton, Sara Sabour, and Nicholas Frosst. 2018. Matrix capsules with EM routing. In *International conference on learning representations*.

[12] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[13] Andrew Lavin and Scott Gray. 2016. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4013–4021.

[14] NVIDIA. 2021. *CUDA C++ Programming Guide.* NVIDIA. Retrieved March 25, 2021 from https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[15] Alec Radford, Luke Metz, and Soumith Chintala. 2015. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434* (2015).

[16] 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.

[17] Haichen Shen, Jared Roesch, Zhi Chen, Wei Chen, Yong Wu, Mu Li, Vin Sharma, Zachary Tatlock, and Yida Wang. 2021. Nimble: Efficiently compiling dynamic neural networks for model inference. *Proceedings of Machine Learning and Systems* 3 (2021).

[18] Benoit Steiner, Chris Cummins, Horace He, and Hugh Leather. 2021. Value Learning for Throughput Optimization of Deep Learning Workloads. *Proceedings of Machine Learning and Systems* 3 (2021).

[19] TVM Team. 2019. *Selective Tuning.* TVM. Retrieved March 25, 2021 from https://github.com/apache/tvm/issues/4188

[20] Tensor Comprehension Team. 2018. *Variable tensor sizes support for TC.* Tensor Comprehension. Retrieved Oct 4, 2021 from https://github.com/facebookresearch/TensorComprehensions/issues/46

[21] Du Tran, Heng Wang, Lorenzo Torresani, Jamie Ray, Yann LeCun, and Manohar Paluri. 2018. A closer look at spatiotemporal convolutions for action recognition. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*. 6450–6459.

[22] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[23] Dimitri Watel and Marc-Antoine Weisser. 2016. A practical greedy approximation for the directed steiner tree problem. *Journal of Combinatorial Optimization* 32, 4 (2016), 1327–1370.

[24] Dimitri Watel and Marc-Antoine Weisser. 2017. *DSTAlgoEvaluation.* mouton5000. Retrieved October 20, 2021 from https://github.com/mouton5000/DSTAlgoEvaluation

[25] Hang Zhang, Chongruo Wu, Zhongyue Zhang, Yi Zhu, Zhi Zhang, Haibin Lin, Yue Sun, Tong He, Jonas Mueller, R Manmatha, et al. 2020. Resnest: Split-attention networks. *arXiv preprint arXiv:2004.08955* (2020).

[26] Minjia Zhang, Menghao Li, Chi Wang, and Mingqin Li. 2021. DynaTune: Dynamic Tensor Program Optimization in Deep Neural Network Compilation. In *International Conference on Learning Representations*. https://openreview.net/forum?id=GTGb3M_KcUl

[27] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-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} 20)*. 863–879.

[28] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. 2020. 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*. 859–873.