ALT: Boosting Deep Learning Performance by Breaking the Wall between Graph and Operator Level Optimizations

Zhiying Xu
zyxu@smail.nju.edu.cn
Nanjing University
China

Jiafan Xu
mf20330099@smail.nju.edu.cn
Nanjing University
China

Hongding Peng
mg20330044@smail.nju.edu.cn
Nanjing University
China

Wei Wang
ww@nju.edu.cn
Nanjing University
China

Xiaoliang Wang
waxili@nju.edu.cn
Nanjing University
China

Haoran Wan
wanhr@smail.nju.edu.cn
Nanjing University
China

Haipeng Dai
haipengdai@nju.edu.cn
Nanjing University
China

Yixu Xu
xuyixu@huawei.com
Huawei Technologies
China

Hao Cheng
chenghao49@hisilicon.com
Huawei Technologies
China

Kun Wang
kun.wang1981@gmail.com
The Hong Kong Polytechnic University
China

Guihai Chen
gchen@nju.edu.cn
Nanjing University
China

ABSTRACT

Deep learning models rely on highly optimized tensor libraries for efficient inference on heterogeneous hardware. Current deep compilers typically predetermine layouts of tensors and then optimize loops of operators. However, such unidirectional and one-off workflow strictly separates graph-level optimization and operator-level optimization into different system layers, missing opportunities for unified tuning.

This paper proposes ALT, a compiler that performs joint graph- and operator-level optimizations for deep models. ALT provides a generic transformation module to manipulate layouts and loops with easy-to-use primitive functions. ALT further integrates an auto-tuning module that jointly optimizes graph-level data layouts and operator-level loops while guaranteeing efficiency. Experimental results show that ALT significantly outperforms state-of-the-art compilers (*e.g.*, Ansor) in terms of both single operator performance (*e.g.*, 1.5× speedup on average) and end-to-end inference performance (*e.g.*, 1.4× speedup on average).

1 INTRODUCTION

Deep learning has become one of the essential building blocks for emerging applications, such as machine translation and autonomous driving systems. To provide ubiquitous services, developers craft high-performance programs supporting various tensor operators (*e.g.*, 2-D convolution and matrix multiplication) on different hardware platforms (*e.g.*, NVIDIA GPU and ARM CPU). However, current vendor libraries (*e.g.*, MKL-DNN [33] and cuDNN [12]) typically demand significant engineering effort on manual optimization. Moreover, the hand-tuning approach can hardly catch up with the fast evolution of deep learning techniques that constantly introduce new tensor operators [32] and new hardware (*e.g.*, neural processing units). Therefore, researchers develop deep compilers [6, 10, 39, 70, 81] to achieve automatic performance optimization by auto-tuning and code generation techniques.

Two key categories of optimizations during compilation are graph-level optimization and operator-level optimization. Graph-level optimization represents operators as nodes and tensors as edges in a computational graph and rewrites nodes or edges to obtain a more efficient graph for inference. For instance, data layout optimization rewrites the tensor storage format to improve memory accessing performance [4, 14, 36, 56, 61, 63, 69]. Constant folding [8, 49, 57] and common subexpression elimination [49, 57] removes redundant nodes. Operator-level optimization, which mainly involves loop optimization, transforms the nested loops in the source code of each operator to schedule the execution of instructions [7, 10, 26, 28, 55]. In this work, we focus on data layout optimization and loop optimization because they yield significant performance improvements and their tuning strongly correlates with operator and hardware characteristics.

Unfortunately, existing deep compilers (*e.g.*, TVM [10], Tensor Comprehension [70], Tiramisu [6], AKG [81]) and auto-tuning techniques (*e.g.*, AutoTVM [11], NeoCPU [44], FlexTensor [89] and Ansor [83]), fail to combine data layout and loop optimizations effectively. These systems first predetermine tensor layouts either manually or via setting a hyper-parameter from a predefined template and then perform loop optimization based on these layouts. There are three major limitations in this unidirectional and one-off workflow. First, manual layout selection implies that only a limited number of layout choices can be explored, hence prone to be suboptimal. Second, altering the tensor layout demands the time-consuming re-implementation of operators that access the tensor. Third, layout optimization and loop optimization are separated into different system layers. Such strict boundary seriously compromises the performance of the generated tensor programs. For instance, we observe that optimizing loops based on the best of three candidate layouts for 2-D convolutional operators can improve the performance by 55.9% on average on the Intel CPU. Moreover, the performance of a specific data layout is sensitive to operator configurations (*e.g.*, tensor shapes) and hardware. Thus it is hard to determine data layouts for each workload without feedback from loop optimization.

This paper proposes ALT, a deep compiler that jointly performs graph-level and operator-level optimizations for deep models. The design of ALT originates from the following insight. Graph-level data layout optimization and operator-level loop optimization could benefit from each other. In the meanwhile, the root cause of the inability to perform cross-layer joint tuning is the coupling between data storage and operator implementation in prior arts, such that altering the data layout requires re-implementing operators. Such high cost for changing layouts further leads to the unidirectional and one-off optimization flow. Therefore, ALT abstracts layout manipulation as easy-to-use primitive functions, such that the task of re-implementing operators can be delegated to a compilation pass without human interference. After reducing the cost, ALT further incorporates layout and loop optimizations into a unified auto-tuning framework, which breaks the boundary between graph- and operator-level optimizations to open new opportunities.

It is not trivial to achieve our goals. We need to strike the following challenges.

Challenge 1: How can we eliminate the overhead of layout transformation? We discover two types of potential overhead when altering the tensor layouts: *layout-conversion* overhead and *fusion-conflict* overhead. First, operators along the data stream may require different tensor layouts to reach their optimal performance. To transform the layouts of tensors produced by other operators at runtime, directly inserting conversion operators will incur extra overhead on data movements. Second, altering the output tensor layout of an operator needs the reconstruction of its loop nest. Such reconstruction may inhibit the operator from being fused with its consumer, which is an important loop optimization technique to improve inter-operator data locality.

Challenge 2: How can we prevent inefficiency due to the search space reconstruction during joint tuning? Changing the output layout of an operator will induce the loop nest reconstruction, which will further lead to the variation of the loop tuning space. For joint tuning, such space variation prohibits a direct iterative exploration. Otherwise, the points we have searched in the last iteration may be invalid in the changing space. This leads to inefficient tuning for most search methods, including genetic and learning-based algorithms, since the accumulated knowledge of the search space structure cannot be further exploited in the newly reconstructed space.

Challenge 3: How can we improve efficiency given the search space explosion with the combination of layout and loop tuning? Along with the joint tuning, the combined search space will be tremendously large, hence inefficient to explore directly. For example, in a typical 2-D convolutional operator, the loop transformation space can contain up to $O(10^7)$ points for its seven nested loops. After combining the layout transformation, the joint search space can be at a scale of $O(10^{19})$ considering three tensors, each of which further involves four dimensions. Moreover, end-to-end optimization is more challenging due to the inter-dependency of many operators and tensors.

To eliminate the two types of overhead brought by layout transformation, we propose a *layout propagation* mechanism. Suppose we have chosen a different layout for the input tensor of an operator. We let the upstream operator, which is the producer of this tensor, directly yield elements based on this new layout, hence no conversion operator is required. To promote operator fusion, we propagate the new layout downstream along the computational graph to let the consumer operator trigger the same loop reconstruction, which helps to align loop nests of multiple operators for fusion. As such, we can safely transform data layouts with minimal overhead, and without sabotaging loop optimization.

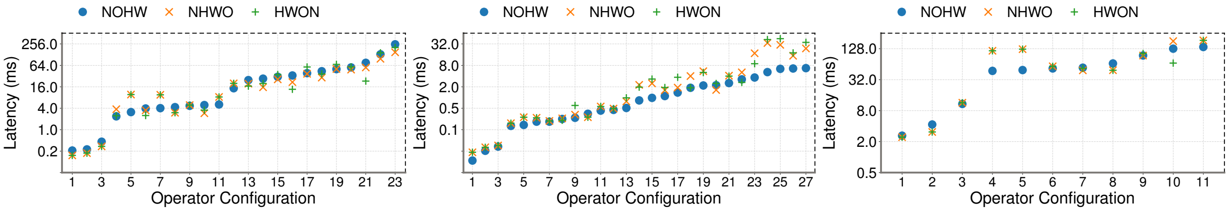
To alleviate the search space reconstruction issue in the co-tuning, our solution is two folds. First, we divide the co-tuning into two stages: *joint* stage and *loop-only* stage. The joint stage searches for optimal tensor layouts, while the loop-only stage only performs loop tuning with the searched layouts remaining unchanged. Second, we propose a *cross-exploration architecture* for the joint stage, rather than the direct exploration. For a new feasible layout, we reconstruct the loop space and then perform multiple rounds of loop optimization to assess the new layout. This design avoids inefficient loop space reconstruction since the loop-only stage keeps layouts unchanged. It also achieves the expected bidirectional and unified tuning flow in the joint stage, because each candidate layout is evaluated based on feedback from loop optimization through our novel tuning architecture.

To avoid the search space explosion due to the combination of layout and loop tuning, we prune the space at two levels. First, we only build layout transformation spaces for tensors accessed by complex operators. In this work, we take convolutions and general matrix multiplication as complex operators, the performance of which are layout sensitive. For other tensors, we further exploit the layout propagation mechanism to propagate the searched layouts onto them without more searching. Second, we identify a promising subspace by tailoring a tuning template for each tensor accessed by complex operators. These templates are constructed based on our analysis of layout optimization considering both operator and hardware characteristics.

By addressing these challenges, ALT achieves joint and efficient graph-level data layout optimization and operator-level loop optimization automatically.

We comprehensively evaluate ALT on Intel CPU, NVIDIA GPU, and ARM CPU. Compared with state-of-the-art vendor libraries (*e.g.*, MKL-DNN [33], cuDNN [12], and XNNPACK [27]) and auto-tuning frameworks (*e.g.*, Ansor [83]), ALT achieves an average of 1.5× speedup in terms of single operator performance, and 1.4× speedup in terms of end-to-end inference performance. Our evaluation also shows that ALT can find data layouts that are not explored in prior arts. Additionally, we have deployed ALT in production environments for four months, boosting a broad spectrum of real workloads (*e.g.*, speech recognition and super resolution).

In summary, we make the following contributions:



(a) C2D on Intel CPU. **(b) C2D on NVIDIA GPU.** **(c) C2D on ARM CPU.**

Figure 1: C2D latency with different data layouts on different hardware platforms.

- We reveal the necessity of joint graph- and operator-level optimizations for deep learning compilation, and that the root cause of the inefficient unidirectional and one-off optimization flow in prior arts lies in the high cost of layout manipulation.
- We design an easy-to-use generic infrastructure that covers a rich layout transformation space. It allows users to manipulate layouts without soiling the hands for re-implementation, and without extra overhead via the layout propagation mechanism during end-to-end optimization.
- We devise a joint layout and loop auto-tuning framework. Via effective space pruning and judicious exploration design, it not only achieves a bidirectional and unified optimization flow but also guarantees tuning efficiency.
- Our extensive evaluation shows that, without human interference, ALT improves performance over state-of-the-art baselines significantly, which also verifies the effectiveness of the proposed techniques.

2 BACKGROUND AND MOTIVATION

A deep compiler typically compiles a neural network with multi-stage lowering and optimization. The compiler takes a model that can be generated by other frameworks (*e.g.*, TensorFlow [1]) as input. It then resolves the model to a computational graph where operators and tensors are represented as nodes and edges, respectively.

Data layout optimization [4, 14, 36, 56, 61, 63, 69] is to rewrite the tensor storage format (*i.e.*, the attributes of an edge) to alleviate memory accessing overhead for operators that access the tensor. Thus, data layout optimization is often classified as graph-level optimization. The storage format refers to the arrangement of tensor dimensions. Take the 2-D convolution (C2D) operator as an example. Popular data layouts for the output tensor of C2D include $NOHW$, $NHWO$, and $HWON$, where N, O, H, W represent the batch size, the number of output channels, the output tensor height, and the output tensor width, respectively. $NOHW$ is widely used on GPU [53], $NHWO$ is the default layout on CPU in TensorFlow [1], and $HWON$ is used in digital signal processing.

After graph-level optimization, the compiler will lower each node in the computational graph to operator-level representation. An operator can typically be represented as deeply nested loops. As the major part of operator-level optimization, loop optimization (*e.g.*, loop tiling, vectorization, etc.) [7, 10, 26, 28, 55] is to transform the loop nest to schedule the execution of statements of each operator.

The motivation for this work is as follows.

Observation 1: It is beneficial to jointly perform data layout optimization and loop optimization. We illustrate the benefits by an experiment that optimizes loops of C2D based on $NOHW$, $NHWO$, and $HWON$ layouts, respectively. Our platforms include 32-core Intel Xeon Silver 4110 CPU@2.1GHz, NVIDIA RTX 2080Ti GPU, and Kirin 990 ARM SoC. We report the performance in Fig. 1, where the latency axis is in log scale, and each hardware involves multiple operator configurations (different number of channels, convolutional strides, etc.) to cover rich workloads. We observe that the best layout could improve the performance of loop optimization by 55.9%, 87.2%, and 48.8% on average on Intel CPU, NVIDIA GPU, and ARM CPU, respectively. On the converse, making a choice among different layouts is not easy when there is no feedback from loop optimization, due to the highly divergent performance with regard to operator configurations and platforms. For example, although $NHWO$ often outperforms $NOHW$ and $HWON$ on CPUs, especially when the number of input channels is small, there is still no clear rule that can fit all configurations.

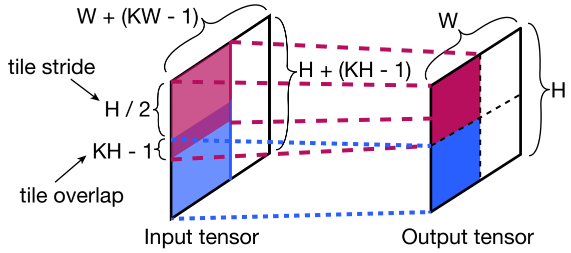


Figure 2: Layout with overlapped tiling.

Observation 2: Existing solutions cannot effectively perform joint tuning due to the high cost of layout manipulation. Existing systems [6, 10, 70] typically couple the tensor storage with the implementation of operators, thus changing layouts requires re-implementation. Such a high cost of layout manipulation limits the number of layout choices that can be explored, and further leads to the unidirectional optimization flow. While there are works using special layouts to improve versatility, *e.g.*, $N\frac{O}{o_t}HWo_t$ where o_t is a tiling parameter that can be changed without re-implementation [44], they still only cover a small layout optimization space. Moreover, switching to another category of layouts still requires re-implementing operators and even rewriting loop-tuning templates.

We use a more versatile layout as a motivating example. This layout is outside the tuning space of $N\frac{O}{o_t}HWo_t$ and is hard to be discovered manually or without joint tuning. It can achieve performance improvement of 32.4% over $N\frac{O}{o_t}HWo_t$. Besides tiling the channel dimension, this layout further tiles the spatial dimensions (the height and the width) of the output tensor into four blocks.

```
for n in range(N):
  for oh, ow in range(2, 2):
    for oo in range(O // o_t):
      for ih, iw in range(H // 2, W // 2):
        for io in range(o_t):
          Conv[n][oh][ow][oo][ih][iw][io] = 0.0
        for i, rh, rw in range(I, KH, KW):
          for io in range(o_t):
            Conv[n][oh][ow][oo][ih][iw][io] += \
                Inp[n][oh][ow][i][ih+rh][iw+rw]\
                * Ker[oo][i][rh][rw][io]
```

Figure 3: Program based on the layout in Fig. 2.

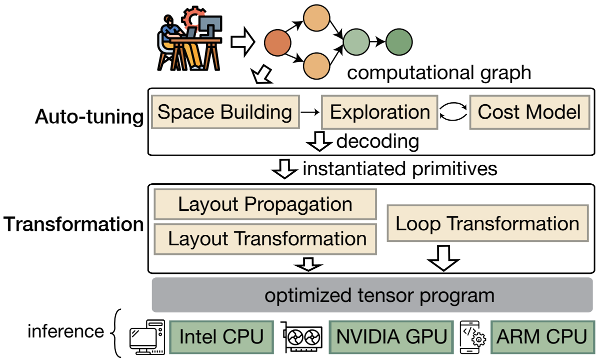


Figure 4: Design overview of ALT.

Each spatial tile of the output tensor has shape $\frac{H}{2} \times \frac{W}{2}$. For a C2D with convolutional stride 1, the height and the width of the input tensor are $H + (KH - 1)$ and $W + (KW - 1)$, where KH and KW are the height and the width of the convolutional window. Due to the sliding-window operation of C2D that has natural overlaps, each output tile requires a $\left(\frac{H}{2} + (KH - 1)\right) \times \left(\frac{W}{2} + (KW - 1)\right)$ tile of the input tensor for convolution. This leads to the layout in Fig. 2, where each colored area denotes a tile, and the overlap between tiles along the input tensor height is exactly $(KH - 1)$. After the layout transformation, the generated loop nest is shown in Fig. 3, where I is the number of input channels, $Conv$, Inp, and Ker are the output tensor, input tensor, and weight tensor, respectively. In Fig. 3, we also tile the output channels by o_t to achieve multi-dimensional layout tiling. Besides, the corresponding loop io is placed as the innermost loop to improve locality, as a showcase for joint layout and loop optimization. The shape of $Conv$ in Fig. 3 is $N \times 2 \times 2 \times \frac{O}{o_t} \times \frac{H}{2} \times \frac{W}{2} \times o_t$. Such multi-dimensional tiling with overlaps promotes data locality and cache utilization. We defer the detailed profiling results on various layouts in Section 7.3.3.

3 SYSTEM OVERVIEW

ALT is a deep compiler that achieves joint graph-level layout optimization and operator-level loop optimization to generate high-performance tensor programs for heterogeneous platforms automatically. The system overview of ALT is depicted in Fig. 4, which incorporates two major modules: *auto-tuning* and *transformation*. The transformation module is a generic infrastructure that achieves low-cost layout and loop manipulation by easy-to-use primitive functions. Based on it, the auto-tuning module performs joint data layout and loop optimization by searching in the parameter spaces of the primitive functions. The workflow of ALT is as follows.

First, the user provides the computational graph of a deep model, which a domain-specific language (*e.g.*, a subset of Python) can express. It can also be constructed from a model file generated by other frameworks (*e.g.*, TensorFlow [1]).

Second, the auto-tuning module builds search space for tensors and operators and explores the space jointly. To reduce the tuning time, it uses a cost model to minimize time-consuming on-device measurements. When the exploration completes, it decodes the best performant point found in the space into a sequence of layout and loop primitives. Then, it delivers these primitives to the transformation module.

Third, the layout propagation submodule propagates layout primitives. Then, the transformation module applies all primitives to perform layout and loop transformation to generate an optimized tensor program. Finally, we deploy the program on different hardware for inference.

4 TRANSFORMATION

We first introduce the transformation module of ALT, which is a generic infrastructure for manipulating data layouts and loops. The transformation module consists of three submodules: layout transformation, layout propagation, and loop transformation.

4.1 Layout Transformation

To achieve low-cost layout manipulation and easy layout tuning, we devise various primitive functions to transform data layouts: *split*, *reorder*, *fuse*, *unfold*, *pad*, and *store_at*. Among them, *split*, *reorder*, and *fuse* are basic primitives and the others are advanced primitives. These primitives lift the data layout transformation from the black-box compiler level to the source level to facilitate leaner control with domain-specific knowledge. We will temporarily cache the operation each time a primitive is applied on a tensor. During program generation, as a compilation pass, we will actually transform the data shapes and alter the corresponding accessing statements in the program. Thus, no human interference is required for re-implementing the operators.

4.1.1 Basic Layout Primitives. The basic primitives perform one-to-one transformations. Given an n dimensional tensor T with original data layout of $N_1 N_2 ... N_n$ and accessing expressions of $i_1, i_2, ..., i_n$, we summarize basic primitives in Table 1, where $1 \le k \le n$ is an index to dimensions, F_k is an integer denoting the splitting factor, p is a permutation vector with $p(k)$ as its k-th element, and $F_{2\to m}$ is an abbreviation for $\prod_{i=2}^{m} F_i$ ($N_{k \to k+m}$ is similar).

For instance, to get the $N \frac{O}{o_t} HW o_t$ layout from $NOHW$, we can apply the following primitive sequence:

```
split(T, dim=2, factors=[O // o_t, o_t])
reorder(T, perm=[1, 2, 4, 5, 3])
```

Alternatively, to pack the layout into spatial blocks, we can transform $NHWO$ through another primitive sequence:

```
fuse(T, dims=[2, 3, 4])
split(T, dim=2, factors=[O // 4, 4, H * W])
reorder(T, perm=[1, 2, 4, 3])
```

Table 1: Basic layout primitives.

Primitive	Parameter	Transformed Shape	Transformed Accessing Expressions
split	$k, F_1, ..., F_m$	$...N_{k-1}F_1...F_mN_{k+1}...$	$..., i_{k-1}, \frac{i_k}{F_{2\to m}}, ..., \frac{i_k}{F_m} \bmod F_{m-1}, i_k \bmod F_m, i_{k+1}, ...$
reorder	permutation vector p	$N_{p(1)}N_{p(2)}...N_{p(k)}$	$i_{p(1)}, i_{p(2)}, ..., i_{p(k)}$
fuse	$k, k+1, ..., k+m$	$...N_{k-1}(N_{k\to k+m})N_{k+m+1}...$	$..., i_{k-1}, (i_kN_{2\to m} + i_{k+1}N_{3\to m} + ... + i_{k+m}), i_{k+m+1}, ...$

During program generation, the first fuse primitive produces shape $N(HWO)$, the second gives $N(\frac{O}{4})4(HW)$, and the final reorder generates $N(\frac{O}{4})(HW)4$, based on Table 1. Assuming the original accessing statement $T[n][h][w][o]$ in the code, it will be transformed as follows:

(1) T[n]$[h(WO) + wO + o]$, and let $e = h(WO) + wO + o$
(2) T[n]$[\frac{e}{HW4}][\frac{e}{(HW)} \bmod 4][e \bmod (HW)]$
(3) T[n]$[\frac{e}{HW4}][e \bmod (HW)][\frac{e}{HW} \bmod 4]$.

4.1.2 Advanced Layout Primitives. The above examples show the versatility of basic primitives. However, there are cases that cannot be covered, such as the overlapped tiling in Fig. 2. To achieve such special transformations, we abstract advanced layout primitives: unfold, pad, and store_at.

unfold: This primitive performs overlapped tiling. It accepts a tile_size parameter, and a stride parameter which is the interval between two tiles:

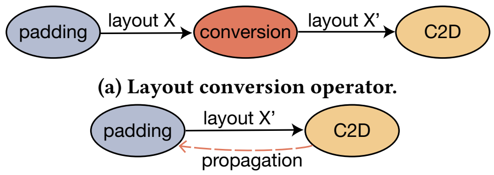
```
unfold(tensor, dimension, tile_size, stride)
```

We denote *tile_size* as B and *stride* as S. If the original size for a dimension is D, this primitive will generate two new dimensions with sizes of $\left(\lceil\frac{D-B}{S}\rceil + 1\right)$ and B. For instance, an array $\{1, 2, 3, 4, 5\}$ can be unfolded to a 2-D array $\{\{1, 2, 3\}, \{3, 4, 5\}\}$ by setting $B = 3$ and $S = 2$. For the input tensor layout in Fig. 2, we can set $B = \frac{H}{2} + (KH - 1)$, $S = \frac{H}{2}$ for the height dimension, and the width is similar.

The unfold primitive is useful for sliding-window computational patterns, *e.g.*, convolutional layers. They have the memory access pattern of $Vi + r$, where V is the constant convolutional stride, i is the window index, and r is a reduction iterator for the offset inside a window. In the following, we use M to denote the window size (*e.g.*, M will be equal to KH and KW for the two patterns $ih + rh$ and $iw + rw$ in Fig. 3, respectively). Then, the original accessing statement $T[Vi + r]$ will be transformed to

$$T\left[\left\lfloor\frac{i}{\lfloor\frac{B-M}{V}\rfloor+1}\right\rfloor\right]\left[Vi+r-S\left\lfloor\frac{i}{\lfloor\frac{B-M}{V}\rfloor+1}\right\rfloor\right] . \qquad (1)$$

Besides unfold, we also propose **pad** and **store_at** primitives. The pad primitive is to append zeros for a selected dimension, which is useful to align data in memory and alleviate bank conflicts on the shared memory of the NVIDIA GPU. The store_at primitive allows fusing two tensors together by attaching one to another to improve inter-tensor data locality. For example, in a fully connected layer, it can attach each element of the bias vector to each column of the weight matrix. Subsequently, the inner product and the bias addition in *general matrix multiplication* (GMM) may be computed together by accessing the weight column and the bias element in the same cache line. Additionally, all three primitives have their inverse counterparts, namely fold, unpad, and decouple_at, to transform layouts back and forth.



(a) Layout conversion operator.

(b) Layout propagation.

Figure 5: Ways to achieve runtime layout conversion.

```
for n in range(N):
  for ht in range(H // 4):
    for w, o in range(W, O):
      for hi in range(4):
        Conv[n][ht][w][o][hi] = 0.0
        for ri, rh, rw in range(I, KH, KW):
          Conv[n][ht][w][o][hi] += Inp[...]*Ker[...]
for n, o, h, w in range(N, O, H, W):
  ReLU[n][o][h][w] = max(Conv[n][h//4][w][o][h%4],0)
```

Figure 6: Loop nests without propagation and fusion.

4.2 Layout Propagation

The layout primitives working at the local tensor level could incur overhead when performing joint or end-to-end optimization on a computational graph. Specifically, we discover two types of such overhead: layout-conversion overhead and fusion-conflict overhead. In this subsection, we will analyze the overhead and propose the layout propagation mechanism to address this issue.

Given a C2D, if it requires a different layout for the weight tensor, we can transform it offline without any runtime overhead because the weight tensor is a constant. Unfortunately, if the C2D requests a different input layout $\mathcal{X}'$, it can only be achieved either by (1) inserting an operator performing runtime layout conversion (Fig. 5a) or (2) letting the producer operator yield each element based on the new layout directly (Fig. 5b). Inserting layout conversion operators will incur extra overhead due to runtime data movements. So, we prefer the second way, which is called layout propagation. After propagation, the padding operator actually performs two tasks at runtime: padding zeros and converting the layout. Similarly, for the output tensor of C2D, we can let its consumer operator access the new layout directly, rather than inserting another conversion operator next to C2D.

Besides the layout-conversion overhead, another delicate issue emerges when incorporating operator fusion. Operator fusion is a loop-tuning technique to promote inter-operator data locality by

```
for n, ht, w, o, hi in range(N, H // 4, W, O, 4):
  Conv[n][ht][w][o][hi] = 0.0
  for ri, rh, rw in range(I, KH, KW):
    Conv[n][ht][w][o][hi] += Inp[...] * Ker[...]
  ReLU[n][ht][w][o][hi] = max(Conv[...], 0)
```

Figure 7: Loop nests with propagation and fusion.

letting the downstream operator consume the intermediate data immediately before spilling out of the cache. Consider two operators: C2D and ReLU, and the original output layouts of them are both $NOHW$. Suppose we transform the output layout of the C2D to $N\frac{H}{4}WO4$ through split and reorder primitives. Then, the generated program is shown in Fig. 6. The loop nest of the C2D is reconstructed accordingly due to the output layout transformation. Different from the original case, we cannot perform loop tiling on the two loop nests with the same tile sizes and then fuse the two nests. Since fusion is an effective technique, reducing the chance of fusion due to the reconstructed loop nest will result in performance loss.

To eliminate such fusion-conflict overhead induced by layout transformation, we extend the layout propagation mechanism such that the same layout can be shared among multiple tensors. Layout propagation can be implemented easily by duplicating the primitive sequence of the source tensor for the target tensor. For instance, we replicate the primitives from tensor *Conv* in Fig. 6, *i.e.*, split and reorder primitives, for tensor *ReLU*. Then ReLU will trigger the same loop nest reconstruction, hence aligned perfectly with that of C2D. Consequently, the fusion-after-tiling in loop tuning will be the same as the normal case, as illustrated in Fig. 7.

Although layout propagation helps to eliminate the overhead incurred by layout transformation, it has three constraints. First, we only propagate primitives along a path with only element-wise operators and among tensors with the same shape. Given an operator $Y[i] = F(X[i])$, there exists an element-wise data mapping between the output tensor Y and the input tensor X. We can propagate the layout of Y to X, or vice versa. This constraint is introduced because the parameters of primitives are shape-dependent. Second, we *will not* propagate a primitive sequence if it contains non-trivial advanced primitives. This is because advanced primitives will induce data expansion. Instead, we will insert conversion operators when they arise, as in Fig. 5a. Third, the layout tuning for each complex operator will be performed independently. This constraint is to eschew the overhead of layout propagation itself, because the optimal layout of a complex operator may lead to inferior performance for another. For example, given two consecutive C2Ds, we will insert a conversion operator between them if needed rather than letting the output tensor of the former C2D and the input tensor of the latter C2D share the same layout. Notably, no conversion operator is necessary when other simple operators exist between the two C2Ds (*e.g.*, we can propagate a layout onto the padding operator like in Fig. 5b and let it perform the actual layout conversion).

4.3 Loop Transformation

We perform loop transformation via reusing the loop primitives of TVM [10]: split, reorder (same names as layout ones, but distinct functions), vectorize, unroll, cache_read/write, parallel, inline, and compute_at. Most loop-tuning techniques, including loop tiling, vectorization, and operator fusion, can be realized by combining these primitives.

5 AUTO-TUNING

Even with the transformation module, optimization is still painful because it requires numerous manual trials. The combination of layout and loop tuning further exacerbates the problem. Thus, in the auto-tuning module, we devise a unified framework to jointly optimize layouts and loops to generate high-performance programs automatically.

Our joint tuning comprehends three steps: 1) we build the layout tuning space for tensors and loop tuning space for operators, each point in the space can be decoded as a primitive sequence; 2) we explore the tuning space to find the best performant point; 3) we decode this point as instantiated primitives and deliver them to the transformation module.

5.1 Space Building

Auto-tuning is to search for the code with the best performance in the tuning space. With our transformation module, we only need to find the best parameters to apply primitives. Thus, the tuning space is equivalent to the parameter space for primitives. For now, we only consider layout split, reorder, and unfold primitives in the layout space. Also, we will omit details on the loop space, which is similar to [83, 89], *e.g.*, space of loop split factors for each operator.

The layout space to be built should be pruned, otherwise, it will be infinitely large because the number of primitives that can be applied is infinite. As in Section 1, we only perform layout tuning for complex operators and propagate their results to reduce the number of tuning tasks. Further, we craft a layout tuning template for each tensor that is accessed by complex operators. Each template only exposes a subset of parameters of primitives as tunable options. The templates are crafted based on the following observations on how data layouts influence performance considering intra-operator data dependency and hardware characteristics.

First, data layout influences the data reuse strategy, [16, 37, 45, 47]. For most architectures, data reuse is vital to reducing the number of memory accesses and improving software pipeline. Consider the C2D as an example, each output element requires $(KH) \cdot (KW) \cdot I$ input elements for reduction. Without data reuse, we need totally $N \cdot H \cdot W \cdot O \cdot (KH) \cdot (KW) \cdot I$ load instructions for the input tensor. Fortunately, an input element is required by at most $(KH) \cdot (KW) \cdot O$ output elements. Thus, we can reuse an input element to accumulate on $KH \times KW$ spatial positions or O channels before spilling it out of the cache. Besides, sequential data accesses can be bundled by SIMD instructions. With these two aspects, we can also explain why $NHWO$ layout often performs better than $NOHW$ layout [83]: 1) an input element can be reused to accumulate on many (at most O) output channels and O is typically large, hence a high reuse rate; 2) output channels can be loaded with SIMD instructions easily since O is the last dimension.

Second, data layout influences cache utilization. Both layout and loop tiling can be exploited to let a data block fit in cache [64]. Besides, we also observe that layout tiling can further prevent cache misses by facilitating hardware prefetching [13, 17, 48]. To verify

Table 2: Profiled L1 data cache misses.

Tile Size	#L1-mis / Pred. (1st F.)	#L1-mis (2nd F.)
512×4	32 / 32	208
512×16	96 / 128	262
512×64	501 / 512	785
512×256	2037 / 2048	2952

this, we conduct an experiment on a Cortex-A76 CPU, which is a big core on Kirin 990 SoC, the L1 data cache line size of which is float32x16 (*i.e.*, 64 bytes). We profile two functions and both of which only load a 2-D data block once from memory with NEON instructions. The data elements for the first function are stored contiguously in memory, *i.e.*, layout tiling case. By contrast, the elements for the second function are stored row by row, *i.e.*, loop tiling case without changing data placements. The profiled L1 cache misses are reported in Table 2, where we also present our predictions based on hardware prefetching in the second column. We observe that the CPU is very likely to fetch four contiguous cache lines when a miss event is triggered. For example, the prediction for tile size 512×4 is calculated as $\frac{512\times4}{16\times4} = 32$. From Table 2, layout tiling is preferable to loop tiling to improve cache utilization via hardware prefetching. Most importantly, the cache performance after layout tiling is always better than in other cases.

The second observation indicates that layout tiling improves cache utilization even though loop tiling has been exploited. Thus, our layout tuning template is a tiling template, with tiling sizes as basic tunable options. For most dimensions, the tiling can be achieved with split primitives. For height and width dimensions of convolutions, it can be achieved with the unfold primitives to enable the overlapped tiling. After splits and unfolds, based on the first observation, we let the tiled channel dimension be the last dimension to promote data reuse and SIMD. Consequently, our data layout tuning template for C2D has the following form:

- output tensor *Conv*: $N\frac{H}{h_t}\frac{W}{w_t}\frac{O}{o_t}h_t w_t o_t$, where h_t, w_t, and o_t are *three tunable* split parameters for tiling H, W, and O.
- input tensor *Inp*: $N\frac{H}{h_t}\frac{W}{w_t}\frac{I}{i_t}(h_t + KH - 1)(w_t + KW - 1)\, i_t$, where $(h_t + KH - 1)$ and $(w_t + KW - 1)$ are the unfolded dimensions, and i_t is the *only tunable* split parameter for tiling I.
- weight tensor *Ker*: $\frac{O}{o'_t}\frac{I}{i'_t}(KH)(KW)i'_t o'_t$, where i'_t and o'_t are *two tunable* split parameters for tiling I and O.

In the above templates, uppercase letters represent the original dimensions, while lowercase letters with a subscript t denote the tiled parameters correspondingly. We do not need to tune the unfolded dimensions for the input tensor, because they are directly related to the tiling of the output tensor. Suppose the tuner splits the H dimension of the output tensor as $\frac{H}{h_t} \times h_t$. It then applies the following unfold primitive on the input tensor directly:

```
unfold(Inp, Inp height, h_t + (KH - 1), h_t)
```

This is the same as the case in Fig. 2 where $h_t = \frac{H}{2}$.

In summary, the pruned layout space for C2D consists of six tunable parameters (*i.e.*, at a scale of $O(10^6)$): h_t, w_t, o_t for tiling H, W, O of the output tensor, i_t for tiling I of the input tensor, i'_t, o'_t for tiling I, O of the weight tensor. For other convolutions (*e.g.*, 3-D case), the template is similar.

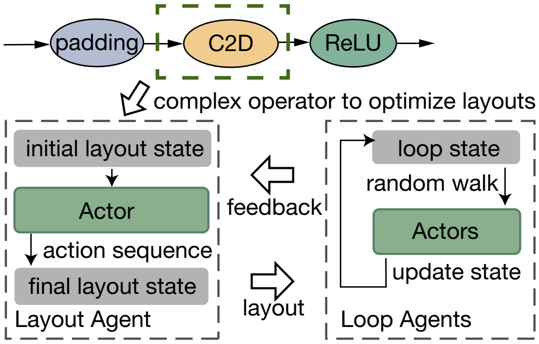


Figure 8: Cross exploration architecture.

For a GMM $C = A \odot B$, where MN, MK, KN are the original layouts of the three matrices, the search space is much smaller due to fewer dimensions. Thus our template consists of split parameters for all dimensions. Then, based on the first observation, the reorder after splits is determined without tuning: $\frac{M}{m_t}\frac{N}{n_t}m_t n_t$ for C, $\frac{M}{m_t}\frac{K}{k_t}m_t k_t$ for A, and $\frac{K}{k_t}\frac{N}{n_t}k_t n_t$ for B. Finally, there are three tunable parameters (*i.e.*, up to $O(10^3)$ points): m_t, k_t, n_t, in the layout space for GMM.

The above templates only perform one-level multi-dimen-sional layout tiling. We can expand them to multi-level cases easily, which can be configured in ALT for scalability. For example, we can use two-level layout tiling templates for ALT, where the template for the output tensor of C2D can be defined as $N\frac{H}{h'_t h_t}\frac{W}{w'_t w_t}\frac{O}{o'_t o_t}h'_t w'_t o'_t h_t w_t o_t$.

Without our template-based pruning, the search space, especially the parameter space for the reorder primitive, will be too large to explore. The only concern after pruning is whether the subspace contains good points. We verify the effectiveness of pruning through experiments.

5.2 Exploration & Cost Model

To explore the search space, we need to: (1) visit points efficiently; (2) evaluate visited points rapidly. We resort to the PPO algorithm [60] from reinforcement learning (RL) to explore the space. Compared with heuristic algorithms (*e.g.*, genetic algorithm) and other RL algorithms, PPO is learning-based and more stable [31], which is introduced in [2] to speed up the tuning space exploration. To speed up the evaluation, we develop a cost model to predict the performance to reduce the number of time-consuming on-device measurements.

In RL, an *agent* will respond (referred to as *action*) to environments based on its *observation*, which is composed of the *state* of the current environment and feedback given by the environment called *reward*. PPO employs two neural networks: *actor* and *critic*. The actor gives actions while the critic judges each action, *i.e.*, fitting the real rewards.

Even with PPO, exploring layout and loop spaces simultaneously is challenging. Consider the C2D as an example, we need to rebuild its loop space every time given a new layout, because the loop nest relies on the output layout, like n, o, h, w in Fig. 6. The reconstructed loop space further leads to that the points searched previously will be invalid in the new space, hence inefficient exploration.

As in Section 1, our solution to this issue involves two aspects. We first divide the performance tuning into two stages: the joint stage and the loop-only stage. We then propose a cross-exploration architecture, as shown in Fig. 8, for the joint stage. The cross-exploration repeats the following process: determining a layout through the layout PPO actor, performing multiple rounds of loop tuning via loop PPO actors, and feeding the best performance back as the reward for the current layout. Consequently, we achieve a bidirectional and unified optimization flow in the joint stage to find better layouts. We also prevent inefficient loop tuning, since the loop reconstruction will not occur in the loop-only stage.

In the following, we will only elaborate on the design of RL action, state, and reward for the joint stage based on the cross-exploration architecture. The loop-only stage can be achieved by removing layout-related searches.

5.2.1 Layout Space Exploration. Since the pruned layout space only involves tunable split parameters, we here develop a generic actor to explore the parameter space of the layout split primitive. Then, the final layout will be determined by a sequence of actions. Take the C2D in Fig. 6 as an example, the action sequence for resolving the output layout of $Conv$ consists of: split H, split W, split O, and reorder them to $N\frac{H}{h_t}\frac{W}{w_t}\frac{O}{o_t}h_t w_t o_t$. The split actor only provides the factors to split H, W, O, while the reorder is determined in the template in Section 5.1. Similarly, replacing the first two splits with unfolds forms the action sequence for the input layout.

Consider the dimension with a size of D in a tensor. To obtain a generic split actor, we map its output action a_s to a contiguous interval $(0, 1)$. Then, the splitting factor F is calculated as follows:

$$F = R(D \cdot a_s)\,. \tag{2}$$

Assume the tensor $Conv$ in Fig. 6 has $O = 32$. The actor gives one action $a_s = 0.5$. Then we derive two split dimensions : $o_t = R(32 * 0.5) = 16$, $\frac{O}{o_t} = R(32/16) = 2$.

The state for the actor is given by the *concatenation* of the current states of all primitives for all tensors of the complex operator (*e.g.*, Inp, Ker, $Conv$ in a C2D). For instance, when unfolding the height of Inp in Fig. 2 into two parts, the current state of the unfold primitive is changed to $[2, \frac{H}{2} + (KH - 1)]$, while the initial state was $[1, H + KH - 1]$. Similarly, the current state for the split primitive is composed of factors, *e.g.*, $[2, 16]$ for $o = 32$ (initial state was $[1, 32]$). Then the final state is the concatenation of all such sub-states.

5.2.2 Loop Space Exploration. The exploration for loop space follows a similar random-walk design as [89]. We first sample a *batch* of points in the loop space and choose the best one as the starting point, then each actor gives a direction for some parameter space. After that, we arrive at the next point by walking along that direction, as shown in Fig. 8.

Including the layout split actor, we have a lot of actors now. To model the interference among subspaces/primitives, we deploy a *global shared critic network* for all actors (not shown in Fig. 8 for simplicity).

The reward r for all RL agents is the same:

$$r = U - l\,, \tag{3}$$

where U is a constant and l is the latency of some point. For layout RL agents, l is chosen as the best latency after several rounds of loop exploration given the current layout.

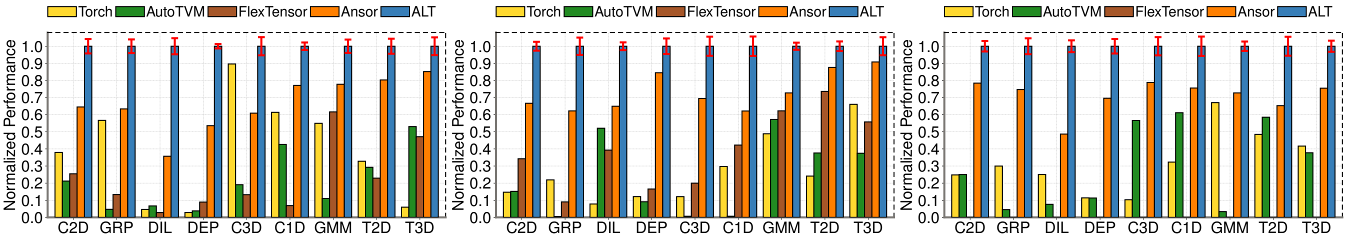
5.2.3 Cost Model. To evaluate points rapidly, we estimate the performance by developing a cost model for each hardware. The cost model is a tree ensemble from XGBoost [9], similar to that of Ansor [83]. For some point, we decode it as primitives and apply them to generate the optimized tensor program. Then we feed the features of the program (*e.g.*, loop structures and accessing expressions) to the cost model to estimate the throughput. During exploration, we only measure the top-k points of a batch or an episode of RL trajectories, which are predicted by the cost model, on the target hardware. These measurements are also used for training the cost model online.

6 IMPLEMENTATION

We implemented ALT based on TVM (v0.8dev1) [10] with 19K LoC of Python and 2K LoC of C++.

To implement the layout transformation, we insert a pass before lowering the tensor expression (TE) of TVM to TVMIR. This pass will rewrite the indices of all tensor accesses in TE when layouts change. With regard to an operator $Y = F(X)$ where the output tensor Y is of shape $N_1N_2..N_m$, in TE this operator has m nested spatial loops, each corresponding to a dimension of Y (one-to-one mapping). We denote the loop variables as $L = \{l_1, l_2, ..., l_m\}$. Assume ALT caches a set of primitive sequences $\mathcal{S}$ either provided manually or by the auto-tuning module automatically. Our pass will first transform accesses for the output tensor Y, and then other tensors. We denote the primitive sequence for Y as $\mathcal{S}(Y)$ (abbreviated as S_Y). Our pass first deducts the final layout of Y by applying each primitive function in S_Y. Assuming the new layout has n dimensions with shape $N'_1N'_2..N'_n$, the loop structure will then be reconstructed by TE as $L' = \{l'_1, l'_2, ..., l'_n\}$. Given the one-to-one mapping between a dimension of the output tensor and a loop variable, we will also have $L' = S_Y(L)$. With this, we can transform accesses for tensor X while ensuring validity. Specifically, the accesses of X must first be remapped with the newly reconstructed loop variables. The remapping is done in two steps: 1) calculating the inverse primitive sequence of S_Y, denoted as S_Y^{-1}; 2) replacing all old loop variables L by $S_Y^{-1}(L')$ in all access indices of X. After this remapping, the tensor accesses of X can be safely transformed to $S_X(S_Y^{-1}(L'))$ by applying $\mathcal{S}(X)$.

To implement the layout propagation, we copy the primitive sequence of the source tensor for the destination tensor. The joint stage of ALT sequentially tunes each complex operator following the topological order and propagates the resulting layouts. A special case is that an operator can have multiple consumers or producers. In the case of multiple consumers, ALT will propagate the layout of the source tensor to all consumers. For the case of multiple producers, consider $Y[i] = F(X_0[i], X_1[i], X_2[j])$, where there are element-wise mappings between X_0 and Y, and between X_1 and Y. When the layouts of X_0 and X_1 are both tuned, ALT will heuristically choose X_0 for propagation onto Y. Conversely, if the layout of Y is tuned first (*i.e.*, there is no complex operator prior X_0 or X_1 that can propagate layouts to them), ALT will propagate the layout of Y to both X_0 and X_1.



(a) Single Operator on Intel CPU.

(b) Single Operator on NVIDIA GPU.

(c) Single Operator on ARM CPU.

Figure 9: Single operator performance.

7 EVALUATION

In this section, we evaluate ALT on various hardware platforms, including 40-core Intel Xeon Gold 6248 CPU@2.5GHz (443GB memory), NVIDIA Tesla V100 (CUDA v11.0), and Kirin 990 SoC (Android v10). We compare ALT with state-of-the-art frameworks and compilers: Torch (v1.7) [53], AutoTVM (v0.8dev1) [11], FlexTensor [89], and Ansor [83]. Torch is a reference point for vendor libraries, which was evaluated by using MKL-DNN library [33] for Intel CPU, cuDNN (v8.0.4) library [12] for NVIDIA GPU, and XNNPACK library [27] for ARM CPU. AutoTVM, FlexTensor, and Ansor are three widely used auto-tuning frameworks. Besides, Ansor outperforms Tensorflow Lite [1] and other hardware-specific compilers [44, 83] such as OpenVINO [34] and TensorRT [52]. Thus, we do not include them as baselines here.

For ALT, if not specified, we use one-level layout tiling templates for layout space building. For loop space exploration, we set the sampling batch size and the episode length to 128, and measure the top-8 points predicted by the cost model on the target hardware. In addition, we take the total number of such on-device measurements as a metric of the search budget for all auto-tuning methods. Thus, a batch or an episode of points in ALT will cost a budget of 8.

7.1 Single Operator Benchmark

We first present the results on single operators. We consider 9 operators, including C2D, Group-wise C2D (GRP), Depth-wise C2D (DEP), Dilated C2D (DIL), 3-D convolution (C3D), 1-D convolution (C1D), GMM, Transposed C2D (T2D), Transposed C3D (T3D). Each operator is evaluated using 10 random configurations with different batch sizes, kernel sizes, etc. For instance, the value of batch size is selected from [1, 16], and the number of input channels is uniformly sampled from [3, 16, 32, 64, 512, 960, 1280]. We generate 90 test cases for each device. The result is normalized based on the geometric mean of speedups over the worst latency of each test case. For C1D, C2D/T2D, and C3D/T3D and their variants, we test NOW/NWO for C1D, $NOHW/NHWO$ for C2D/T2D, and $NODHW/NDHWO$ (D is the depth dimension) for C3D/T3D and report the best for baselines except Torch (it only supports NOW, $NOHW$, $NODHW$). We set the search budget to 1000 for all auto-tuning methods, which is suggested by Ansor. For ALT, the budget for the joint stage and the loop-only stage is 300 and 700 respectively.

As shown in Fig. 9a, on Intel CPU ALT achieves 9.5×, 9.9×, 9.8×, and 1.6× speedups in comparison with Torch, AutoTVM, FlexTensor, and Ansor respectively. Among all operators, DIL and DEP have lower operational intensity (the ratio of the number of computational instructions to the number of memory access instructions), and thus they are more likely to be memory-bound. For DIL and DEP, ALT outperforms other baselines with a large margin because layout tuning can effectively reduce memory accessing overheads. Even for operators that are typically compute-bound, *e.g.*, C2D and C3D, ALT still achieves notable speedups. This is because the operational intensity depends on tensor shapes. ALT can tailor the tensor layouts toward each specific shape and hardware platform.

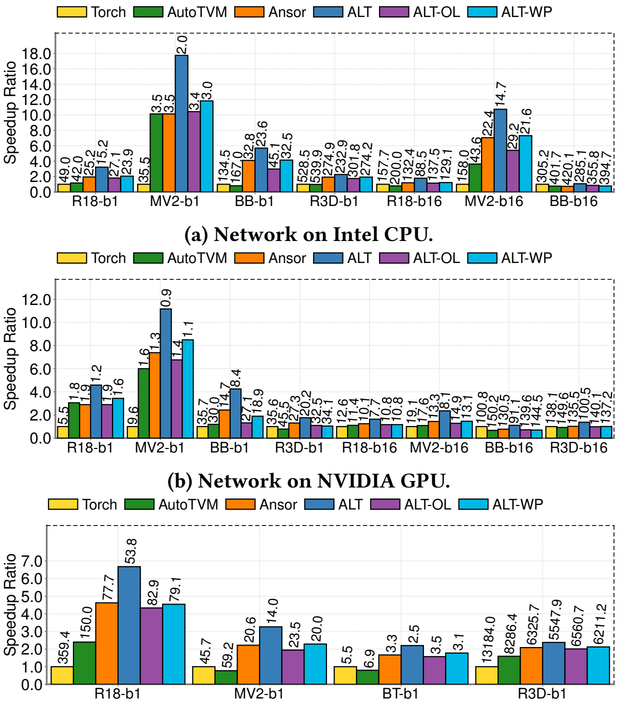
We achieve similar results on NVIDIA GPU and ARM CPU. Compared with Ansor, ALT achieves an average of 1.5× speedup on NVIDIA GPU (Fig. 9b), and 1.4× speedup on ARM CPU (Fig. 9c). We do not include the results of FlexTensor for ARM CPU since it does not support ARM backends. Generally, auto-tuning methods can outperform Torch because non-typical operator configurations are often less optimized in vendor libraries. Further, AutoTVM suffers from small tuning space and FlexTensor has no cost model, thus both demonstrate inferior performance than Ansor and ALT. Additionally, compared with Ansor, ALT can effectively tune data layouts with feedback from operator-level optimization and hence illustrate significant improvements.

7.2 End-to-End Benchmark

We then evaluate the end-to-end performance of ALT with five neural networks, including applications of 1) image processing: ResNet-18 (R18) [30], MobileNet-V2 (MV2) [59], 2) natural language processing: BERT-base (BB) [19], BERT-tiny (BT) [20], and 3) video processing: ResNet3D-18 (R3D) [29]. For Intel CPU and NVIDIA GPU, the benchmarks use batch sizes of 1 and 16. For ARM CPU, we set the batch size to 1 due to the limited resource.

For convolutional networks, the input tensor is of shape $N \times 3 \times 224 \times 224$ (image processing) and $N \times 3 \times 16 \times 112 \times 112$ (video processing), respectively. For BERT, the shape of the input tensor is $N \times 128$. For auto-tuning baselines, we set the search budget as 20,000 (which is suggested by Ansor [83]). We set the budget for the joint stage to 8,000 and the budget for the loop-only stage to 12,000 in ALT. Additionally, Torch uses $NOHW/NODHW$ layouts while AutoTVM and Ansor use $N\frac{O}{o_t}HWo_t/N\frac{O}{o_t}DHWo_t$ after integrating NeoCPU [44].

We illustrate the speedup ratio of all methods over Torch in Fig. 10, where $b1$ denotes batch size 1 and $b16$ denotes batch size 16. The number on top of each bar demonstrates the latency in milliseconds. To verify the effectiveness of the joint tuning and layout propagation, we define two variants of ALT: (1) ALT-OL, which only involves loop optimization without the joint stage based on $NHWO/NDHWO$ layouts; (2) ALT-WP, which only eliminates conversion operators between adjacent operators, as that shown



(a) Network on Intel CPU.

(b) Network on NVIDIA GPU.

(c) Network on ARM CPU.

Figure 10: End-to-end inference performance.

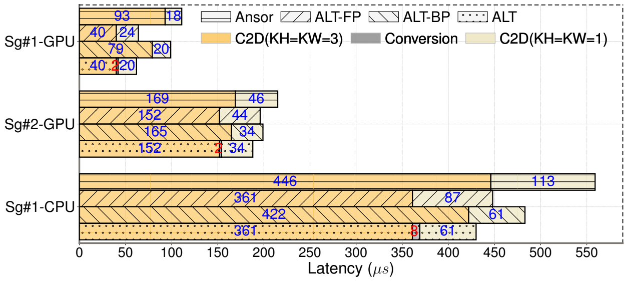


Figure 11: The overhead of layout propagation.

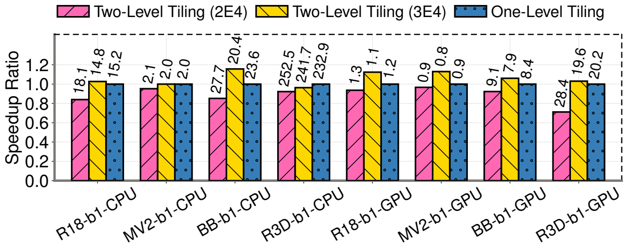


Figure 12: End-to-end performance of different settings.

in Fig. 5b. Compared with Ansor[1], ALT achieves 1.47×, 1.39×, and 1.46× speedups on Intel CPU, NVIDIA GPU, and ARM CPU, respectively. For R3D, most of its operators are compute-bound, thus ALT achieves similar results with Ansor. For MV2, which is a lightweight network with lower operational intensity, ALT outperforms the baselines significantly.

Notice that ALT-OL achieves similar performance as Ansor because both of them mainly involve loop tuning. When incorporating layout tuning and basic layout propagation, ALT-WP shows 1.1× speedup over ALT-OL in general and no improvement in a few cases. ALT achieves 1.3× speedup on average compared with ALT-WP. This is because operator fusion is a critical loop-tuning technique to improve performance, while ALT-WP cannot combine layout tuning and loop tuning effectively.

7.3 Micro Benchmark

We dive into the details to achieve a better understanding of the system design. First, we present the overhead of layout propagation. We then study the parameter sensitivity of ALT in the end-to-end optimization. We will also conduct a case study to help understand the searched layouts and loops. Finally, we present more observations to provide hints for deep compiler optimization. Notably, we do not give more experiments on cost model [83] and the PPO exploration method [2] because they are not our major contributions.

7.3.1 Layout Propagation Overhead: We here study the overhead of layout propagation to show the necessity of the introduced constraints in Section 4.2. We evaluate two subgraphs on 48-core Intel(R) Xeon(R) Gold 5117 CPU @2.0GHz and NVIDIA RTX 3070 GPU. Each subgraph consists of three operators: padding (padding size is 1), C2D ($KH = KW = 3, stride = 1$), C2D ($KH = KW = 1, stride = 1$). The input height/width of subgraph#1 is 7, while it is 14 for subgraph#2. Besides, all the numbers of input/output channels are 512, except that the number of output channels of the latter C2D ($KH = KW = 1$) in subgraph#2 is 2048. We conduct two variants of ALT: ALT-FP and ALT-BP. ALT-FP will first tune C2D ($KH = KW = 3$) and propagate its output layout to the input tensor of the latter C2D ($KH = KW = 1$). While ALT-BP will first tune C2D ($KH = KW = 1$) and propagate its input layout to the output tensor of the former C2D ($KH = KW = 3$). Instead, ALT will tune the two C2Ds separately and insert a layout conversion operator between them according to the third constraint in Section 4.2.

The profiling results are reported in Fig. 11, where we use Ansor as a reference point. We observe that ALT outperforms ALT-FP and ALT-WP. In other words, the best output layout of the C2D ($KH = KW = 3$) is sub-optimal for the second C2D ($KH = KW = 1$), and vice versa. Independent layout tuning for each complex operator brings more benefits while the layout conversion only incurs low overhead (2 microseconds for GPU and 8 microseconds for CPU). Combined with the results of ALT-WP in Fig. 10, the fusion conflicts incur more overhead than layout conversions when performing layout transformation. We alleviate such two kinds of overheads by layout propagation and eschew the overhead of propagation itself by introducing necessary constraints.

7.3.2 Parameter sensitivity: We study the parameter sensitivity by comparing the performance given different budget settings and search space sizes. We include three variants here: 1) two-level tiling templates with 20,000 budget; 2) two-level tiling templates

[1] Ansor performs better than Torch [53] and AutoTVM [11]. We omit the results of Torch and AutoTVM due to the lack of space.

but with 30,000 budget; 3) one-level layout tiling templates with 20,000 budget as the baseline (*i.e.*, same as Section 7.2).

The end-to-end performance in different settings is shown in Fig. 12. The first variant expands the search space size while keeping the budget unchanged. Compared with it, the baseline illustrates 15% performance improvement on average. By contrast, after setting the budget to 30,000, the second variant improves about 6% performance over the baseline. Also, more improvements can be obtained if given a larger budget, since one-level tiling templates constitute a subset of the two-level variant. For the budget of 20,000 in Section 7.2, one-level layout tiling templates yield a more effective trade-off between the final performance and the search space size. The budget of 20,000 to optimize a network typically costs 12-16 hours. But, it is affordable for practitioners as they only need to execute ALT once. Additionally, these results demonstrate the scalability of the tuning space, which is hard to achieve in prior auto-tuning works.

7.3.3 Case study: To understand how the joint tuning improves the loop performance, we perform loop optimization based on $NHWO$, $NOHW$, $N\frac{O}{o_t}HWo_t$, and $N\frac{H}{h_t}\frac{W}{w_t}\frac{O}{o_t}h_tw_to_t$ on Intel CPU. We profiled a small computational graph, which contains several operators: padding (after padding, the tensor will have $N = 1, I = 3, H = W = 230$), C2D ($O = 64, KH = KW = 7$, convolutional stride is 2), bias addition, and ReLU. This small graph is also the first layer of R18-b1. We set $o_t = 16$ for $N\frac{O}{o_t}HWo_t$ ($i_t = 3$ for the input tensor), while the searched layout has $h_t = 4, w_t = 16, o_t = 16$ for $N\frac{H}{h_t}\frac{W}{w_t}\frac{O}{o_t}h_tw_to_t$ ($i_t = 1$ for the input tensor). The platform is 48-core Intel(R) Xeon(R) Gold 5117 CPU @2.0GHz.

Table 3: Profiling results based on several layouts.

Layout (Conv & Ker)	#Inst.	#L1-lds	#L1-mis	#L1-sts	Lat.
$NHWO$ & $rsIO$	509.4	166.4	9.7	103.6	0.34
$NOHW$ & $OIrs$	626.9	206.6	4.5	121.3	0.49
$N\frac{O}{o_t}HWo_t$ & $\frac{O}{o_t}\frac{I}{i_t}rsio$	567.6	193.6	9.9	112.9	0.37
$N\frac{H}{h_t}\frac{W}{w_t}\frac{O}{o_t}h_tw_to_t$ & ...	550.5	174.3	3.9	106.2	0.25

The results are summarized in Table 3, where we abbreviate $(KH)(KW)$ to rs for the weight tensor Ker. The latency (Lat.) is recorded in milliseconds and others are on a scale of 10^6. We observe that for all layouts, except $NOHW$, their optimized loop nests prefer reusing input values by computing multiple output channels once with SIMD, thus reporting fewer instructions and fewer cache loads/stores than $NOHW$. Compared with $N\frac{O}{o_t}HWo_t$, $NHWO$ shows better data locality due to the larger tile size for the output channel. Specifically, $O = 64$ in $NHWO$ yields a higher reuse rate than $o_t = 16$ in $N\frac{O}{o_t}HWo_t$, as analyzed in Section 5.1. Further, $N\frac{H}{h_t}\frac{W}{w_t}\frac{O}{o_t}h_tw_to_t$ achieves more efficient cache utilization (only 2% misses) than $NHWO$, due to the contiguous storage of intra-tile data elements after layout tiling.

7.3.4 Other observations: Besides the profiled results, we observe that the o_t parameter in the templates to tile output channels is often tuned as twice as the number of vector lanes that the platform supports when the spatial dimensions are not tiled. Specifically, we observe that $o_t = 32$ on Intel CPU, $o_t = 8$ on NVIDIA GPU, and $o_t = 8$ on ARM CPU frequently arise, although the number of vector lanes with float32 data types is 16 for AVX-512, 4 for CUDA, and 4 for NEON. This is different from many hand-tuned libraries. However, these results are not applicable to all configurations or platforms. By contrast, the methodology in our micro-benchmarks could help understand the optimized layout, and similar analysis can be conducted for other cases.

8 RELATED WORK

Deep learning compiler. A variety of deep compilers have been developed. Halide [55] and TVM [10] decouple the operator description and schedule to simplify loop optimization. XLA [39], Glow [58], nGraph [18], and Relay [57] develop graph-level representations to support layout selection, constant folding, etc. Rammer [46] supports fine-grained operator fusion. CODE [67] speeds up the ensemble of deep models. Cortex [23], Nimble [62], DietCode [82], and CoRa [24] focus on optimizing recursive/dynamic networks. TASO [35], Tensat [76], PET [71], Unity [68], and Ollie [86] perform subgraph substitutions to obtain a more efficient computational graph. Tensor Comprehension (TC) [70], Tiramisu [6], MLIR [38], and AKG [81] integrate polyhedral techniques. Bolt [75] provides support for tensor core by integrating CUTLASS [51]. SoyBean [72] and Alpa [84] provide auto-tuning support for inter- and intra-operator parallelism in distributed scenarios. UNIT[74], AMOS [88], and TensorIR [25] provide support for tensorization on tensor accelerators. SparTA [87] and SparseTIR [77] introduce representation for sparse tensors. Compared with ALT, the layout auto-tuning, together with the joint data layout and loop optimization, is limited in these works. For instance, TC and Tiramisu require developers to transform data buffers manually. Although Relay and TVM can insert layout conversion operators between C2Ds with different predefined layouts (*e.g.*, $NOHW$, $NHWO$, etc.), each layout combination requires a manual re-implementation of operators. By contrast, ALT supports generic graph-level layout auto-tuning with feedback from operator-level optimization.

Layout and loop tuning. Many systems try to improve the performance with layout transformation [11, 15, 22, 41, 43, 44, 54, 79, 83]. For instance, [22, 79] optimize data layouts for FPGA design. [41, 54] suggests to choose layouts among $NHWO$, $NOHW$, etc. [15, 43] tightly couples it with the sparse computation. Compared with ALT, they lack versatility and are limited to a few tuning options. By contrast, the systems in [11, 83] can typically set the o_t parameter in $N\frac{O}{o_t}HWo_t$ layout after integrating NeoCPU [44]. However, they have limitations: 1) switching to another kind of layout (*e.g.*, a different reorder, or the overlapped tiling in ALT) still requires manually rewriting operators and even the templates of loop tuning, due to the coupling among data storage, operator implementation, and the loop-tuning templates; 2) o_t is typically predetermined, while in Ansor [83] is set via resolving the loop tiling configurations after loop tuning, as a packing technique and only for constant tensors, hence no joint tuning. ALT addresses the two limitations via 1) the generic layout transformation submodule, which requires no re-implementation, and is also independent of the loop transformation to achieve the decoupling; 2) an auto-tuning module at a higher level to orchestrate the cross-layer joint tuning while guaranteeing efficiency. As for recent loop optimization techniques [2, 3, 5, 21, 42, 65, 66, 73, 78, 80, 85, 89–91], such as delicate cost models [3, 5, 42, 73], aggressive operator fusion [21, 40, 46, 50, 80, 90], and micro-kernel construction [91], they are complementary to ALT.

9 CONCLUSION

In this paper, we propose ALT, a compiler that jointly performs graph-level data layout optimization and operator-level loop optimization for deep models. ALT provides a generic transformation module for low-cost layout and loop manipulation. It further integrates an auto-tuning module for bidirectional and unified layout and loop tuning. Experiments show that ALT outperforms state-of-the-art vendor libraries and auto-tuning frameworks.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[2] Byung Hoon Ahn, Prannoy Pilligundla, Amir Yazdanbakhsh, and Hadi Esmaeilzadeh. Chameleon: Adaptive code optimization for expedited deep neural network compilation. *arXiv preprint arXiv:2001.08743*, 2020.

[3] Peter Ahrens, Fredrik Kjolstad, and Saman Amarasinghe. An asymptotic cost model for autoscheduling sparse tensor programs. *arXiv preprint arXiv:2111.14947*, 2021.

[4] David F Bacon, Susan L Graham, and Oliver J Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

[5] Riyadh Baghdadi, Massinissa Merouani, Mohamed-Hicham Leghettas, Kamel Abdous, Taha Arbaoui, Karima Benatchba, et al. A deep learning based cost model for automatic code optimization. *Proceedings of the 3rd Machine Learning and Systems (MLSys)*, 3, 2021.

[6] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Proceedings of IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2019.

[7] Utpal Banerjee. *Loop transformations for restructuring compilers: the foundations*. Springer Science & Business Media, 2007.

[8] Fabian Boemer, Yixing Lao, Rosario Cammarota, and Casimir Wierzynski. nGraph-HE: a graph compiler for deep learning on homomorphically encrypted data. In *Proceedings of the 16th ACM International Conference on Computing Frontiers (CF)*, 2019.

[9] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM International Conference on Knowledge Discovery and Data mining (SIGKDD)*, 2016.

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

[11] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Proceedings of the 32nd Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

[12] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[13] Trishul M Chilimbi, Mark D Hill, and James R Larus. Cache-conscious structure layout. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1999.

[14] Doosan Cho, Sudeep Pasricha, Ilya Issenin, Nikil Dutt, Yunheung Paek, and SunJun Ko. Compiler driven data layout optimization for regular/irregular array access patterns. In *Proceedings of ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems (LCTES)*, 2008.

[15] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Automatic generation of efficient sparse tensor format conversion routines. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2020.

[16] Philippe Clauss and Benoît Meister. Automatic memory layout transformations to optimize spatial locality in parameterized loop nests. *ACM SIGARCH Computer Architecture News*, 28(1):11–19, 2000.

[17] Patrick Cronin, Xing Gao, Haining Wang, and Chase Cotton. An exploration of ARM system-level cache and GPU side channels. In *Annual Computer Security Applications Conference (ACSAC)*, 2021.

[18] Scott Cyphers, Arjun K Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, et al. Intel nGraph: An intermediate representation, compiler, and executor for deep learning. *arXiv preprint arXiv:1801.08058*, 2018.

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[20] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Tiny-BERT: Distilling BERT for natural language understanding. *arXiv preprint arXiv:1909.10351*, 2019.

[21] Yaoyao Ding, Ligeng Zhu, Zhihao Jia, Gennady Pekhimenko, and Song Han. IOS: Inter-operator scheduler for CNN acceleration. In *Proceedings of Machine Learning and Systems (MLSys)*, volume 3, 2021.

[22] Isak Edo Vivancos, Sayeh Sharify, Daniel Ly-Ma, Ameer Abdelhadi, Ciaran Bannon, Milos Nikolic, Mostafa Mahmoud, Alberto Delmas Lascorz, Gennady Pekhimenko, and Andreas Moshovos. Boveda: Building an on-chip deep learning memory hierarchy brick by brick. In *Proceedings of Machine Learning and Systems (MLSys)*, volume 3, 2021.

[23] Pratik Fegade, Tianqi Chen, Phil Gibbons, and Todd Mowry. Cortex: A compiler for recursive deep learning models. *Proceedings of the 3rd Machine Learning and Systems (MLSys)*, 2021.

[24] Pratik Fegade, Tianqi Chen, Phillip Gibbons, and Todd Mowry. The CoRa tensor compiler: Compilation for ragged tensors with minimal padding. In *Proceedings of Machine Learning and Systems (MLSys)*, 2022.

[25] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, et al. Tensorir: An abstraction for automatic tensorized program optimization. *arXiv preprint arXiv:2207.04296*, 2022.

[26] Zhangxiaowen Gong, Zhi Chen, Justin Szaday, David Wong, Zehra Sura, Neftali Watkinson, Saeed Maleki, David Padua, Alexander Veidenbaum, Alexandru Nicolau, et al. An empirical study of the effect of source-level loop transformations on compiler stability. *Proceedings of ACM on Programming Languages*, 2:1–29, 2018.

[27] Google. XNNPACK: Highly optimized library of floating-point neural network inference operators for ARM, WebAssembly, and x86 platforms, 2021.

[28] Mary Hall, Jacqueline Chame, Chun Chen, Jaewook Shin, Gabe Rudy, and Malik Murtaza Khan. Loop transformation recipes for code generation and auto-tuning. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*. Springer, 2009.

[29] Kensho Hara, Hirokatsu Kataoka, and Yutaka Satoh. Learning spatio-temporal features with 3d residual networks for action recognition. In *Proceedings of IEEE International Conference on Computer Vision Workshops (ICCV)*, 2017.

[30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[31] Pablo Hernandez-Leal, Bilal Kartal, and Matthew E Taylor. A survey and critique of multiagent deep reinforcement learning. *Autonomous Agents and Multi-Agent Systems*, 33(6):750–797, 2019.

[32] Jonathan Ho, Tim Salimans, Alexey Gritsenko, William Chan, Mohammad Norouzi, and David J Fleet. Video diffusion models. *arXiv preprint arXiv:2204.03458*, 2022.

[33] Intel. MKL-DNN, 2017. [Online; accessed 15-June-2022].

[34] Intel. OpenVINO Toolkit, 2019. [Online; accessed 15-June-2022].

[35] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: Optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, New York, NY, USA, 2019. Association for Computing Machinery.

[36] Y-J Ju and H Dietz. Reduction of cache coherence overhead by compiler data layout and loop transformation. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*. Springer, 1991.

[37] Mahmut Kandemir, Alok Choudhary, Jagannathan Ramanujam, Nagaraj Shenoy, and Prithviraj Banerjee. Enhancing spatial locality via data layout optimizations. In *European Conference on Parallel Processing (Euro-Par)*. Springer, 1998.

[38] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021.

[39] Chris Leary and Todd Wang. XLA: Tensorflow, compiled. *TensorFlow Dev Summit*, 2017.

[40] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. Automatic horizontal fusion for GPU kernels. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2022.

[41] Chao Li, Yi Yang, Min Feng, Srimat Chakradhar, and Huiyang Zhou. Optimizing memory efficiency for deep convolutional neural networks on GPUs. In *Proceedings of the 16th International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2016.

[42] Rui Li, Yufan Xu, Aravind Sukumaran-Rajam, Atanas Rountev, and P Sadayappan. Analytical characterization and design space exploration for optimization of CNNs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[43] Shaoshan Liu, Bin Ren, Xipeng Shen, and Yanzhi Wang. CoCoPIE: Making mobile ai sweet as pie–compression-compilation co-design goes a long way. *arXiv preprint arXiv:2003.06700*, 2020.

[44] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. Optimizing CNN model inference on CPUs. In *Proceeding of USENIX Annual Technical Conference (ATC)*, 2019.

[45] Qingda Lu, Christophe Alias, Uday Bondhugula, Thomas Henretty, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev, Ponnuswamy Sadayappan, Yongjian Chen, Haibo Lin, et al. Data layout transformation for enhancing data locality on NUCA chip multiprocessors. In *18th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2009.

[46] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[47] Svetozar Miucin and Alexandra Fedorova. Data-driven spatial locality. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)*, 2018.

[48] Mohammad Alaul Haque Monil, Seyong Lee, Jeffrey S Vetter, and Allen D Malony. Understanding the impact of memory access patterns in intel processors. In *IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. IEEE, 2020.

[49] Steven Muchnick et al. *Advanced compiler design implementation*. Morgan kaufmann, 1997.

[50] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. DNNFusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, 2021.

[51] Nvidia. CUTLASS, 2017. [Online; accessed 15-June-2022].

[52] Nvidia. TensorRT, 2017. [Online; accessed 15-June-2022].

[53] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Proceedings of the 33rd Conference on Neural Information Processing Systems (NeurIPS)*, 2019.

[54] Phitchaya Mangpo Phothilimthana, Amit Sabne, Nikhil Sarda, Karthik Srinivasa Murthy, Yanqi Zhou, Christof Angermueller, Mike Burrows, Sudip Roy, Ketan Mandke, Rezsa Farahani, et al. A flexible approach to autotuning multi-pass machine learning compilers. In *30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2021.

[55] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[56] Easwaran Raman, Robert Hundt, and Sandya Mannarswamy. Structure layout optimization for multithreaded programs. In *International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2007.

[57] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. Relay: A new IR for machine learning frameworks. In *Proceedings of the 2nd ACM International Workshop on Machine Learning and Programming Languages (MAPL)*, 2018.

[58] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018.

[59] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[60] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[61] Kamal Sharma, Ian Karlin, Jeff Keasler, James R McGraw, and Vivek Sarkar. Data layout optimization for portable performance. In *European Conference on Parallel Processing (Euro-Par)*. Springer, 2015.

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

[63] Jun Shirako and Vivek Sarkar. Integrating data layout transformations with the polyhedral model. In *Proceedings of International Workshop on Polyhedral Compilation Techniques (IMPACT)*, 2019.

[64] Jun Shirako and Vivek Sarkar. An affine scheduling framework for integrating data layout and loop transformations. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*. Springer, 2020.

[65] Benoit Steiner, Chris Cummins, Horace He, and Hugh Leather. Value learning for throughput optimization of deep learning workloads. *Proceedings of the 3rd Machine Learning and Systems (MLSys)*, 2021.

[66] Benoit Steiner, Chris Cummins, Horace He, and Hugh Leather. Value learning for throughput optimization of deep learning workloads. In *Proceedings of Machine Learning and Systems (MLSys)*, 2021.

[67] Ettore MG Trainiti, Thanapon Noraset, David Demeter, Doug Downey, and Simone Campanoni. CODE: Compiler-based neuron-aware ensemble training. *Proceedings of the 3rd Machine Learning and Systems (MLSys)*, 3, 2021.

[68] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, et al. Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[69] Nicolas Vasilache, Benoit Meister, Muthu Baskaran, and Richard Lethin. Joint scheduling and layout optimization to enable multi-level vectorization. *IMPACT, Paris, France*, 2012.

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

[71] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. PET: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

[72] Minjie Wang, Chien-chin Huang, and Jinyang Li. Unifying data, model and hybrid parallelism in deep learning via tensor tiling. *arXiv preprint arXiv:1805.04170*, 2018.

[73] Yao Wang, Xingyu Zhou, Yanming Wang, Rui Li, Yong Wu, and Vin Sharma. Tuna: A static analysis approach to optimizing deep neural networks. *arXiv preprint arXiv:2104.14641*, 2021.

[74] Jian Weng, Animesh Jain, Jie Wang, Leyuan Wang, Yida Wang, and Tony Nowatzki. UNIT: Unifying tensorized instruction compilation. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021.

[75] Jiarong Xing, Leyuan Wang, Shang Zhang, Jack Chen, Ang Chen, and Yibo Zhu. Bolt: Bridging the gap between auto-tuners and hardware-native performance. In *Proceedings of Machine Learning and Systems (MLSys)*, 2022.

[76] Yichen Yang, Phitchaya Mangpo Phothilimtha, Yisu Remy Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. Equality saturation for tensor graph superoptimization. *Proceedings of the 3rd Machine Learning and Systems (MLSys)*, 2021.

[77] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. SparseTIR: Composable abstractions for sparse compilation in deep learning. *arXiv preprint arXiv:2207.04606*, 2022.

[78] Cody Hao Yu, Xingjian Shi, Haichen Shen, Zhi Chen, Mu Li, and Yida Wang. Lorien: Efficient deep learning workloads delivery. In *Proceedings of ACM Symposium on Cloud Computing (SoCC)*, 2021.

[79] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proceedings of ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2015.

[80] Jie Zhao, Xiong Gao, Ruijie Xia, Zhaochuang Zhang, Deshi Chen, Lei Chen, Renwei Zhang, Zhen Geng, Bin Cheng, and Xuefeng Jin. Apollo: Automatic partition-based operator fusion through layer by layer optimization. In *Proceedings of Machine Learning and Systems (MLSys)*, 2022.

[81] Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, et al. AKG: automatic kernel generation for neural processing units using polyhedral transformations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, 2021.

[82] Bojian Zheng, Ziheng Jiang, Cody Hao Yu, Haichen Shen, Joshua Fromm, Yizhi Liu, Yida Wang, Luis Ceze, Tianqi Chen, and Gennady Pekhimenko. DietCode: Automatic optimization for dynamic tensor programs. In *Proceedings of Machine Learning and Systems (MLSys)*, 2022.

[83] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: generating high-performance tensor programs for deep learning. In *Proceedings of the 14th*

USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2020.

[84] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, July 2022. USENIX Association.

[85] Lianmin Zheng, Ruochen Liu, Junru Shao, Tianqi Chen, Joseph E Gonzalez, Ion Stoica, and Ameer Haj Ali. Tenset: A large-scale program performance dataset for learned tensor compilers. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (NeuIPS)*, 2021.

[86] Liyan Zheng, Haojie Wang, Jidong Zhai, Muyan Hu, Zixuan Ma, Tuowei Wang, Shizhi Tang, Lei Xie, Kezhao Huang, and Zhihao Jia. OLLIE: Derivation-based tensor program optimizer. *arXiv preprint arXiv:2208.02025*, 2022.

[87] Ningxin Zheng, Bin Lin, Quanlu Zhang, Lingxiao Ma, Yuqing Yang, Fan Yang, Yang Wang, Mao Yang, and Lidong Zhou. SparTA: Deep-learning model sparsity via tensor-with-sparsity-attribute. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[88] Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. AMOS: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 874–887, 2022.

[89] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. FlexTensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[90] Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, Feiwen Zhu, Wenyi Zhao, Xiaoyong Liu, Jun Yang, Jidong Zhai, et al. AStitch: enabling a new multi-dimensional optimization space for memory-intensive ml training and inference on modern simt architectures. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[91] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, et al. ROLLER: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.