ALT: Breaking the Wall between Data Layout and Loop Optimizations for Deep Learning Compilation

Zhiying Xu
Jiafan Xu
Hongding Peng
Wei Wang
{zyxu,xujf,hdpeng}@smail.nju.edu.cn
ww@nju.edu.cn
Nanjing University
China

Xiaoliang Wang
Haoran Wan
Haipeng Dai
waxili@nju.edu.cn
wanhr@smail.nju.edu.cn
haipengdai@nju.edu.cn
Nanjing University
China

Yixu Xu
Hao Cheng
xuyixu@huawei.com
chenghao49@hisilicon.com
Huawei Technologies
China

Kun Wang
kun.wang@ieee.org
University of California, Los Angeles
United States

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 deep compiler that performs joint graph-level layout optimization and operator-level loop optimization. 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).

CCS Concepts: • **Software and its engineering → Source code generation**; • **Computing methodologies → Machine learning**.

EuroSys '23, May 8–12, 2023, Rome, Italy
© 2023 Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9487-1/23/05…$15.00
https://doi.org/10.1145/3552326.3587440

Keywords: compiler techniques and optimizations, code generation and synthesis, deep learning systems

ACM Reference Format:
Zhiying Xu, Jiafan Xu, Hongding Peng, Wei Wang, Xiaoliang Wang, Haoran Wan, Haipeng Dai, Yixu Xu, Hao Cheng, Kun Wang, and Guihai Chen. 2023. ALT: Breaking the Wall between Data Layout and Loop Optimizations for Deep Learning Compilation. In *Eighteenth European Conference on Computer Systems (EuroSys '23), May 8–12, 2023, Rome, Italy.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3552326.3587440

1 Introduction

Deep learning is crucial for applications like machine translation and autonomous driving. 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 [32] and cuDNN [11]) 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 operators [31] and new hardware (*e.g.*, neural processing units). Therefore, researchers develop deep compilers [6, 9, 38, 68, 80] to achieve automatic performance optimization by auto-tuning and code generation techniques.

By representing operators as nodes and tensors as edges to compose a computational graph during compilation, two key optimizations are graph-level data layout optimization and operator-level loop optimization. Layout optimization reorganizes tensor storage to improve memory accessing performance [4, 13, 35, 54, 59, 61, 67]. Loop optimization transforms the nested loops in the source code of each operator to schedule the execution of instructions [7, 9, 25, 27, 53].

Unfortunately, existing deep compilers (*e.g.*, TVM [9], Tensor Comprehension [68], Tiramisu [6], AKG [80]) and auto-tuning techniques (*e.g.*, AutoTVM [10], NeoCPU [43], FlexTensor [88] and Ansor [82]), 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 layout among three candidates for 2-D convolutional operators can improve performance by 55.9% on the Intel CPU. Moreover, the performance of a specific layout is sensitive to operator configurations (*e.g.*, tensor shapes) and hardware, making it hard to determine layouts for each workload without feedback from loop optimization.

This paper proposes ALT, a deep compiler that jointly performs graph-level data layout and operator-level loop optimizations for deep models. The design of ALT originates from the following insight. Data layout optimization and loop optimization could benefit from each other. 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 the two optimizations to open new opportunities.

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

Challenge 1: How to eliminate the overhead of layout transformation? Altering tensor layouts can incur two types of overhead: *layout-conversion* overhead and *fusion-conflict* overhead. Operators along the data stream may require different tensor layouts to achieve optimal performance, but introducing conversion operators to transform layouts can cause extra data movements. Additionally, changing the output tensor layout of an operator will reconstruct its loop nest, which may prevent operator fusion with its consumer operator and thus sabotage inter-operator data locality.

Challenge 2: How to 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 to improve efficiency given the search space explosion in joint tuning? The joint search space can be extremely large, hence inefficient to explore directly. For instance, a 2-D convolutional operator can contain up to $O(10^7)$ points in its seven nested loops. Further combined with layout tuning, the joint space can scale up to $O(10^{19})$ points considering that there are three tensors, each of which has four dimensions.

To eliminate the overhead of layout transformation, we propose a *layout propagation* mechanism. Layout propagation lets the upstream or downstream operators access the new layout directly, rather than inserting a new conversion operator. To promote operator fusion, we propagate a new layout across multiple operators, which lets consumer operators trigger the same loop reconstruction, helping to align multiple loop nests for fusion.

To address the search space reconstruction issue, we split the co-tuning into a *joint* stage for searching for optimal tensor layouts and a *loop-only* stage for optimizing loops with the searched layouts unchanged. We then design a *cross-exploration* architecture for the joint stage. For a candidate layout, we reconstruct the loop space and perform multiple rounds of loop tuning to assess the new layout. This design achieves a bidirectional and unified tuning flow by choosing a layout based on feedback from loop optimization. It also avoids inefficient loop space reconstruction since the loop-only stage keeps layouts unchanged.

To avoid the search space explosion in joint tuning, we prune the space at two levels. First, we only create layout transformation spaces for tensors accessed by *complex* operators (convolutions and general matrix multiplication), as their performance is sensitive to data layouts. For other tensors, we utilize layout propagation to transfer the searched layouts without further searching. Second, we use tuning templates, which only expose a few tunable options, to identify a promising subspace. These templates are tailored 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 [32], cuDNN [11], and XNNPACK [26]) and auto-tuning frameworks (*e.g.*, Ansor [82]), 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:

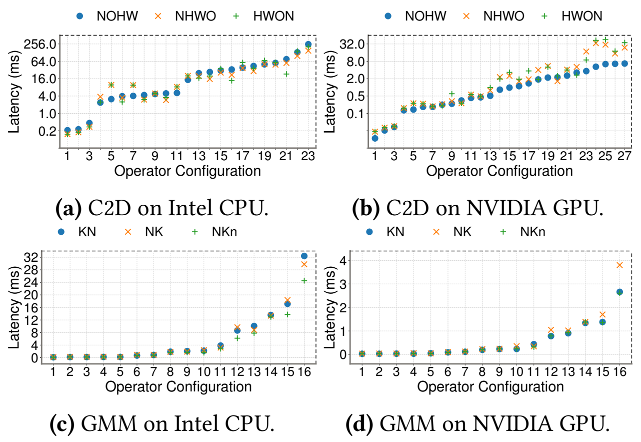
- We reveal the necessity of joint graph-level layout and operator-level loop optimizations for deep learning compilation, and that the root cause of the inefficient unidirectional and one-off optimization flow in prior arts lies in the high cost of layout manipulation.
- We design an easy-to-use generic 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, 13, 35, 54, 59, 61, 67] 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 [51], $NHWO$ 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.)



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

(c) GMM on Intel CPU. **(d)** GMM on NVIDIA GPU.

Figure 1. C2D and GMM latency with different data layouts.

[7, 9, 25, 27, 53] 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 and NVIDIA RTX 2080Ti GPU. We report the performance in Fig. 1a and Fig. 1b, where the latency is in log scale to accommodate the range of values, from several microseconds to tens of milliseconds, and each hardware involves multiple operator configurations (different numbers of channels, convolutional strides, etc.) to cover rich workloads. These workloads are sampled from widely-used settings. We observe that the best layout could improve the performance of loop optimization by 55.9% and 87.2% on average on the Intel CPU and NVIDIA GPU, respectively. We also report the performance of general matrix multiplications (GMM) with different layouts in Fig. 1c and Fig. 1d. Given $C = A \odot B$, we use KN to denote the default layouts (MN, MK, KN for C, A, B). NK denotes an alternative layout [1, 51] that transposes B. NKn represents a custom layout by tiling three matrices with a factor, leading to $\frac{M}{m}\frac{N}{n}mn$, $\frac{M}{m}Km$, and $\frac{N}{n}Kn$ ($m = n = 16$). The results show that the best layout could improve the performance by 20.6% and 24.8% on the Intel CPU and NVIDIA GPU respectively. However, making a choice among different layouts is not easy without feedback from loop optimization, due to the highly divergent performance with regard to operator configurations and platforms. For instance, although $NHWO$ often outperforms $NOHW$ and $HWON$ for C2D on CPUs when the number of input channels is small, and NKn often outperforms KN and NK for GMM, there is still no clear rule that can fit all configurations and platforms.

Observation 2: Existing solutions cannot effectively perform joint tuning due to the high cost of layout

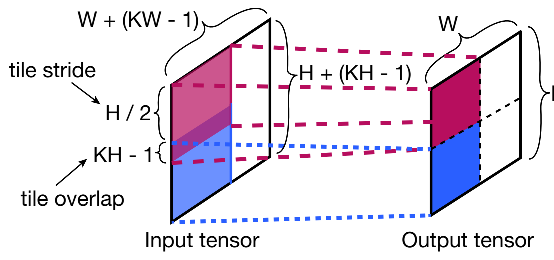


Figure 2. Layout with overlapped tiling.

```
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.

manipulation. Existing systems [6, 9, 68] 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 [43], 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. 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 7.3.4.

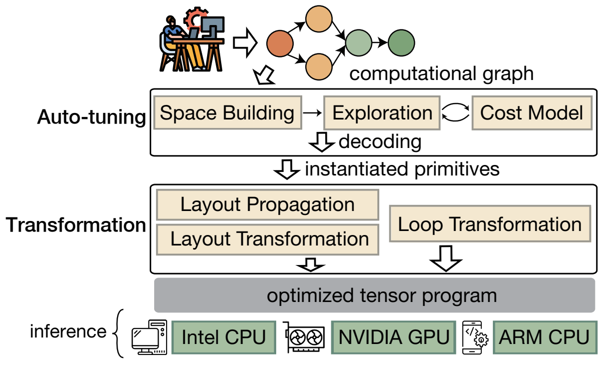


Figure 4. Design overview of ALT.

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, which is a generic infrastructure for manipulating data layouts and loops. It further 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 $N_1N_2...N_n$ 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 $N\frac{O}{o_t}HWo_t$ 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])
```

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][$\frac{e}{HW4}$][$\frac{e}{(HW)}$ mod 4][e mod (HW)]
3. T[n][$\frac{e}{HW4}$][e mod (HW)][$\frac{e}{HW}$ mod 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

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

Besides unfold, we also propose **pad** and **store_at** primitives. The pad primitive is to append zeros for a selected dimension, which helps to align data in memory and alleviate bank conflicts on 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 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.

With these layout primitives, users can develop various layout schedules without re-writing operators. This feature also simplifies the design of our later auto-tuning module.

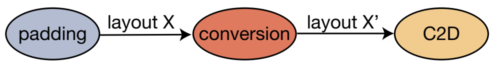
4.2 Layout Propagation

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 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.

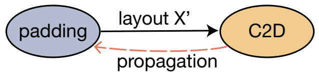
Given a C2D, if it requires a different layout for the weight tensor, we can transform it offline without any runtime overhead because the weight tensor is a constant. Unfortunately, if the C2D requests a different input layout $\mathcal{X}'$, it can only be achieved either by (1) inserting an operator performing

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	$k, k+1, ..., k+m$	$...N_{k-1}(N_{k\to k+m})N_{k+m+1}...$	$..., i_{k-1}, (i_kN_{2\to m} + i_{k+1}N_{3\to m} + ... + i_{k+m}), i_{k+m+1}, ...$



(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.

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 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

```
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.

them. In joint tuning, reducing the chance of fusion due to such loop reconstruction 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 split and reorder primitives from tensor *Conv* in Fig. 6 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.

Algorithm 1: Layout Propagation

```
Function LayoutPropagation($G = (V, E), p$):
    P ⟵ $\{p_T\}$    // P is a set of primitive sequences
    if $p_T = \emptyset$ or $p_T$ contains non-trivial advanced
      primitives or $o_2$ is a complex operator then
        insert conversion op before $o_2$
        return P
    Q ⟵ Queue($\{e^T_{o_1\to o_2}\}$)    // edge $e^T_{o_1\to o_2} \in E$
      connecting operators $o_1$ and $o_2$, denoting tensor T
    while Q $\neq \emptyset$ do
        $e^s_{o_1\to o_2}$ ⟵ Q.pop()
        foreach $e^t_{o_2\to o_3} \in E$ do
            if $o_2$ is elementwise and t.shape() =
              s.shape() and $o_3$ is not complex then
                $p_t$ ⟵ $p_s$.copy() && P.insert($p_t$)
                Q.push($e^t_{o_2\to o_3}$)
    return $P$
```

We present the layout propagation algorithm in Algorithm 1. Given a computational graph $G = (V, E)$ and a primitive sequence p_T for tensor T, we propagate p_T in a topological order until three constraints are unmet. The first constraint (Line 3) checks for non-trivial advanced primitives to minimize data expansion, as advanced primitives can cause extra redundancy. The second constraint (Line 3) eschews the overhead of propagation itself by tuning each complex operator independently. This is because the optimal layout for one operator may be suboptimal for another. The algorithm inserts a conversion operator and terminates if either constraint is unmet (Line 4). Thus, we do not consider layout-conversion overhead in this case. For example, a conversion operator will be inserted between two consecutive C2Ds, rather than sharing the same layout between them. Then we construct a queue Q for further propagation. For each edge $e^s_{o_1 \to o_2}$ in Q, the layout can be propagated across o_2 onto $e^t_{o_2 \to o_3}$ if o_2 is an element-wise operator (with the form $Y[i] = F(X[i])$) and the shape of tensor t is the same as s. This third constraint is introduced because the parameters of primitives are shape-dependent. Additionally, if o_3 is complex (Line 10), the propagation terminates without inserting a conversion operator (unlike Line 4). That is, a simple operator between non-consecutive C2Ds will perform the actual layout conversion like the padding in Fig. 5b.

4.3 Loop Transformation

We perform loop transformation via reusing the loop primitives of TVM [9]: 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. In this section, 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

With our transformation module, we only need to find the best parameters to apply primitives for auto-tuning. Thus, the tuning space is equivalent to the parameter spaces 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 [82, 88], *e.g.*, space of loop split factors for each operator.

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

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 data reuse strategy [15, 36, 44, 46]. For most architectures, data reuse is vital to reducing the number of memory accesses and improving the pipeline. Consider 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 [82]: 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 [62]. Besides, we also observe that layout tiling can further prevent cache misses by facilitating hardware prefetching [12, 16, 47]. To verify, we conduct an experiment on a Cortex-A76 CPU, 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 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 on a miss event. The prediction for 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.

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.

For a GMM $C = A \bigodot B$, 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-dimensional layout tiling. We can expand them to multi-level cases easily, which can be configured in ALT for scalability. For example, the two-level layout tiling template for the output tensor of C2D is $N\frac{H}{h'_t h_t}\frac{W}{w'_t w_t}\frac{O}{o'_t o_t}h'_t w'_t o'_t h_t w_t o_t$.

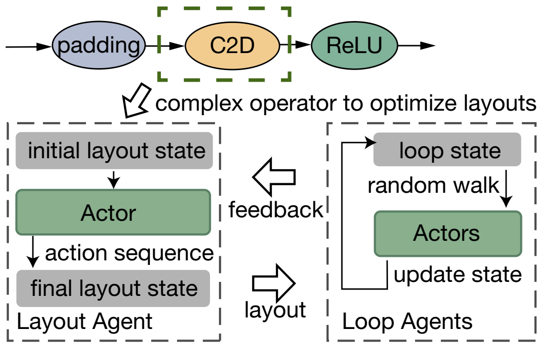


Figure 8. Cross exploration architecture.

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. Besides, our templates are general enough such that users rarely require creating new ones. If the need arises, it is typically straightforward for most hardware architectures. We intend to explore a template-free approach in the future.

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 [58] 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 [30], 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 [88]. 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 a 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 a cost model for each hardware. The cost model is a XGBoost tree ensemble [8], similar to that of Ansor [82]. For a point, we decode it as primitives and apply them to generate the optimized 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.

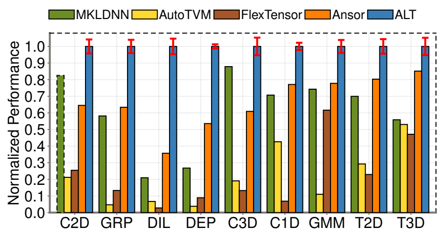
6 Implementation

We implemented ALT based on TVM (v0.8dev1) [9] with 19K LoC of Python and 2K LoC of C++.

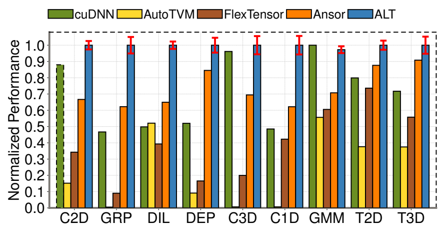
To implement the layout transformation, we insert a pass before lowering the tensor expression (TE) of TVM to TVMIR. This pass will rewrite the indices of all tensor accesses in TE when layouts change. With regard to an operator $Y = F(X)$ where the output tensor Y is of shape $N_1 N_2 .. N_m$, in TE 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 $\mathcal{S}$. We denote the primitive sequence for Y 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, 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. Then the tensor accesses of X can be safely transformed to $S_X(S_Y^{-1}(L'))$.

The joint stage of ALT sequentially tunes each complex operator following the topological order and propagates the resulting layouts. A special case is an operator with 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 prior complex operator that can propagate layouts to X_0 or X_1), ALT will propagate the layout of Y to both X_0 and X_1.

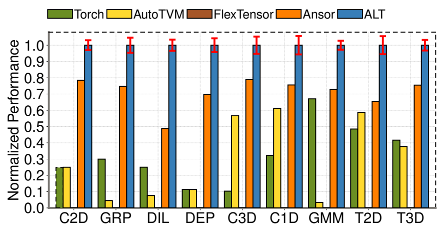
For the RL-based tuning, to improve efficiency, we pretrained our PPO agent by optimizing several workloads (C2D and GMM) with the recommended hyperparameters outlined in [76]. Half a day is sufficient on an NVIDIA V100 GPU.



(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

We evaluate ALT on various platforms, including 40-core Intel Xeon Gold 6248 CPU@2.5GHz (443GB memory), NVIDIA Tesla V100 (CUDA v11.6), and Kirin 990 SoC (Android v10). For the single-operator benchmark, we compare ALT with vendor libraries: MKL-DNN [32] for the Intel CPU, cuDNN (v8.2.4) [11] for the NVIDIA GPU, and Torch [51] with XNNPACK [26] for the ARM SoC, and widely used auto-tuning frameworks: AutoTVM (v0.8dev1) [10], FlexTensor [88], and Ansor [82]. For the end-to-end benchmark, we further compare ALT to hardware-specific compilers: OpenVINO [33] for the Intel CPU and TensorRT [50] for the NVIDIA GPU.

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 nine complex operators that are layout sensitive, 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.

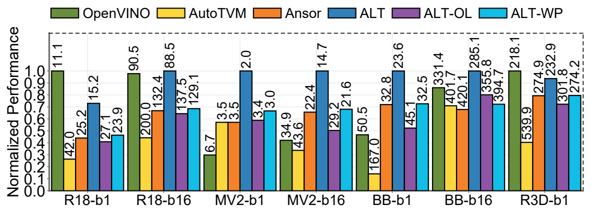
As shown in Fig. 9a, on Intel CPU ALT achieves 2.1×, 9.9×, 9.8×, and 1.6× speedups in comparison with MKL-DNN, AutoTVM, FlexTensor, and Ansor respectively. Among all operators, DIL and DEP have lower operational intensity, and thus they are more likely to be memory-bound. For DIL and DEP, ALT outperforms other baselines with a large margin because layout tuning can effectively reduce memory accessing overheads. Even for operators 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.

Compared with cuDNN on the NVIDIA GPU, ALT achieves averaging 1.4× speedup. Generally, ALT is comparable to cuDNN in typical workloads (*e.g.*, GMM of 2048 × 2048) and better in most non-typical ones which 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. Compared with Ansor, ALT achieves significant speedups owing to joint layout and loop tuning: 1.5× and 1.4× speedups on NVIDIA GPU and ARM CPU respectively.

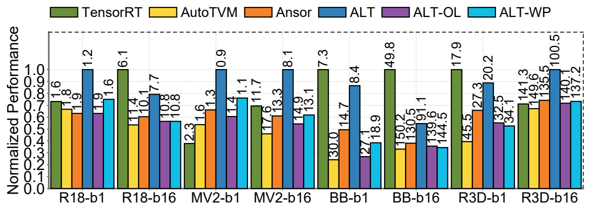
7.2 End-to-End Benchmark

We then evaluate the end-to-end performance of ALT with five neural networks, including 1) image processing: ResNet-18 (R18) [29], MobileNet-V2 (MV2) [57], 2) natural language processing: BERT-base (BB) [18], BERT-tiny (BT) [19] (RNNs are not included due to the lack of space.), and 3) video processing: ResNet3D-18 (R3D) [28]. 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.

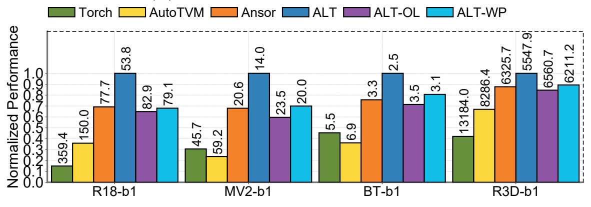
For convolutional networks, the input tensor is of shape $N\times3\times224\times224$ (image processing) and $N\times3\times16\times112\times112$ (video processing), respectively. For BERT, the shape of the input tensor is $N\times128$. For auto-tuning baselines, we set the search budget as 20,000 (which is suggested by Ansor [82]). 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 [43].



(a) Network on Intel CPU.



(b) Network on NVIDIA GPU.



(c) Network on ARM CPU.

Figure 10. End-to-end inference performance.

We illustrate the normalized performance 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 in Fig. 5b. Compared with OpenVINO, TensorRT, and Torch, ALT achieves averaging 1.67× speedup (1.16× in median), 1.24× speedup (1.13× in median), and 3.6× speedup (2.8× in median) on the three platforms respectively. The vendor libraries (MKL-DNN and cuDNN) under OpenVINO and TensorRT put tremendous engineering efforts into optimizing typical workloads (*e.g.*, R18), where ALT can achieve comparable performance. By contrast, for lightweight networks with lower operational intensity (*e.g.*, MV2), ALT achieves significant speedups. Compared with the state-of-the-art auto-tuning system, Ansor, ALT achieves averaging 1.47×, 1.39×, and 1.46× speedups on Intel CPU, NVIDIA GPU, and ARM CPU, respectively.

ALT-OL and Ansor achieve similar performance as both involve loop tuning. With layout tuning and basic layout propagation, ALT-WP shows 1.1× speedup over ALT-OL and no improvement in a few cases. Further, ALT achieves 1.3× speedup over ALT-WP, owing to operator fusion while ALT-WP cannot combine layout and loop tuning effectively.

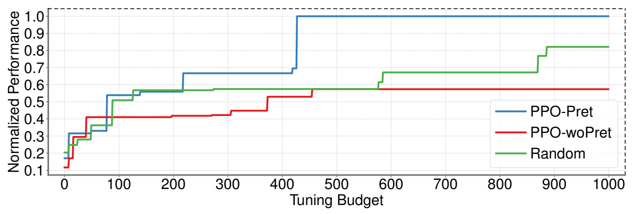


Figure 11. The efficiency of different layout tuning methods.

7.3 Micro Benchmark

We further dive into the details of the system design.

7.3.1 Searching Method: We verify the effectiveness of the pretrained PPO by tuning layouts of a C2D ($N = 1, I = 3, H = W = 230, O = 64, KH = KW = 7$, and the stride is 2), which is also the first C2D in R18, on the Intel CPU used in Fig. 10a. We compare three searching methods: 1) random sampling (Random), 2) PPO without pretraining (PPO-woPret), and 3) PPO with pretraining (PPO-Pret). The results are reported in Fig. 11. PPO-Pret achieves the best performance (1.2× speedup) with 2× less tuning budget compared with the random method. Compared with PPO-woPret, the pretraining can transfer the knowledge from optimizing other operators to improve the online data efficiency.

7.3.2 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 second constraint in Section 4.2.

The profiling results are reported in Fig. 12, 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

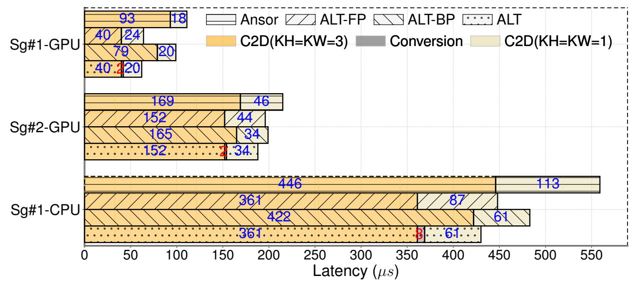


Figure 12. The overhead of layout propagation.

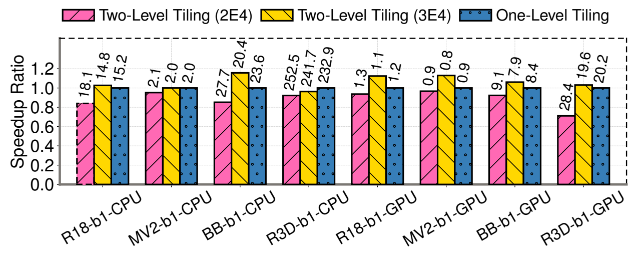


Figure 13. End-to-end performance of different settings.

layout transformation. We alleviate such two kinds of overheads by layout propagation and eschew the overhead of propagation itself by introducing necessary constraints.

7.3.3 Parameter Sensitivity: We study the parameter sensitivity by comparing 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 but with 30,000 budget; 3) one-level layout tiling templates with 20,000 budget as the baseline.

The end-to-end performance in different settings is shown in Fig. 13. 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.

7.3.4 Case Study: To understand how the joint tuning improves 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 (same as Section 7.3.2). We profiled a small computational graph: padding (the padded tensor has $N = 1, I = 3, H = W = 230$), C2D ($O = 64, KH = KW = 7$, stride is 2), bias addition, and ReLU, which 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).

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.

7.3.5 Other Observations: Besides the profiled results, we observe that o_t in C2D in the templates 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. Although not applicable to all configurations or platforms, 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 [53] and TVM [9] decouple the operator description and schedule to simplify loop optimization. XLA [38], Glow [56], nGraph [17], and Relay [55] develop graph-level representations to support layout selection, constant folding, etc. Rammer [45] supports fine-grained operator fusion. CODE [65] speeds up the ensemble of deep models. Cortex [22], Nimble [60], DietCode [81], and CoRa

[23] focus on optimizing recursive/dynamic networks. TASO [34], Tensat [74], PET [69], Unity [66], and Ollie [85] perform subgraph substitutions to obtain a more efficient computational graph. Tensor Comprehension (TC) [68], Tiramisu [6], MLIR [37], and AKG [80] integrate polyhedral techniques. Bolt [73] provides support for tensor core by integrating CUTLASS [49]. SoyBean [70] and Alpa [83] provide auto-tuning support for inter- and intra-operator parallelism in distributed scenarios. UNIT[72], AMOS [87], and TensorIR [24] provide support for tensorization on tensor accelerators. SparTA [86] and SparseTIR [75] 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 [10, 14, 21, 40, 42, 43, 52, 78, 82]. For instance, [21, 78] optimize data layouts for FPGA design. [40, 52] suggests to choose layouts among $NHWO$, $NOHW$, etc. [14, 42] 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 [10, 82] can typically set the o_t parameter in $N\frac{O}{o_t}HWo_t$ layout after integrating NeoCPU [43]. However, o_t is typically predetermined, and hence no joint tuning is involved. Moreover, the operator implementation and data layouts are tightly coupled in NeoCPU and Ansor, such that changing layouts requires inefficient re-implementation. The root cause of such limitations resides in the lack of a versatile infrastructure for layout transformation, which cannot be addressed by incrementally adding more layout candidates for selection. Although Ansor further integrates the auto-packing mechanism, it is only performed on constant tensors after loop tuning by heuristics instead of joint tuning, and the resulting performance is not necessarily better. ALT addresses the 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, 20, 41, 63, 64, 71, 77, 79, 84, 88–90], such as delicate cost models [3, 5, 41, 71], aggressive operator fusion [20, 39, 45, 48, 79, 89], and micro-kernel construction [90], 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.

10 Acknowledgement

We would like to thank Tianxiang Shen, Lei Chen, Yafan Huang, Shihui Song, anonymous reviewers, and our shepherd, Kai Li, for their valuable feedback. This work is supported by Huawei Technologies, the NSFC under Grant Numbers 62272213, 62172204, and 62272223, and Jiangsu Innovation and Entrepreneurship (Shuangchuang) Program.

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] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

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

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

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

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

[47] 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.

[48] 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.

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

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

[51] 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.

[52] 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.

[53] 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.

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

[55] 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.

[56] 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.

[57] 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.

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

[59] 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.

[60] 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.

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

[62] 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.

[63] 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.

[64] 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.

[65] 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.

[66] 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.

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

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] 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.

[73] 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.

[74] 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.

[75] 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.

[76] Chao Yu, Akash Velu, Eugene Vinitsky, Yu Wang, Alexandre Bayen, and Yi Wu. The surprising effectiveness of ppo in cooperative, multi-agent games. *arXiv preprint arXiv:2103.01955*, 2021.

[77] 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.

[78] 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.

[79] 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.

[80] 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.

[81] 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.

[82] 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.

[83] 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.

[84] 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.

[85] 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.

[86] 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.

[87] 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.

[88] 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.

[89] 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.

[90] 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.