Yu F, Zhao JC, Cui HM *et al.* VTensor: Using virtual tensors to build a layout-oblivious AI programming framework. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 38(5): 1074-1097 Sept. 2023. DOI: 10.1007/s11390-022-1457-6

VTensor: Using Virtual Tensors to Build a Layout-Oblivious AI Programming Framework

Feng Yu[1, 2] (俞　峰), Jia-Cheng Zhao[1, 2] (赵家程), *Member, CCF*
Hui-Min Cui[1, 2, *] (崔慧敏), *Member, CCF*, Xiao-Bing Feng[1, 2] (冯晓兵), *Distinguished Member, CCF*
and Jingling Xue[3] (薛京灵), *Fellow, IEEE*

[1] *Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China*
[2] *School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing 100080, China*
[3] *School of Computer Science and Engineering, University of New South Wales, Sydney 1466, Australia*

E-mail: yufeng@ict.ac.cn; zhaojiacheng@ict.ac.cn; cuihm@ict.ac.cn; fxb@ict.ac.cn; jingling@cse.unsw.edu.au

Received March 19, 2021; accepted April 12, 2022.

Abstract Tensors are a popular programming interface for developing artificial intelligence (AI) algorithms. Layout refers to the order of placing tensor data in the memory and will affect performance by affecting data locality; therefore the deep neural network library has a convention on the layout. Since AI applications can use arbitrary layouts, and existing AI systems do not provide programming abstractions to shield the layout conventions of libraries, operator developers need to write a lot of layout-related code, which reduces the efficiency of integrating new libraries or developing new operators. Furthermore, the developer assigns the layout conversion operation to the internal operator to deal with the uncertainty of the input layout, thus losing the opportunity for layout optimization. Based on the idea of polymorphism, we propose a layout-agnostic virtual tensor programming interface, namely the VTensor framework, which enables developers to write new operators without caring about the underlying physical layout of tensors. In addition, the VTensor framework performs global layout inference at runtime to transparently resolve the required layout of virtual tensors, and runtime layout-oriented optimizations to globally minimize the number of layout transformation operations. Experimental results demonstrate that with VTensor, developers can avoid writing layout-dependent code. Compared with TensorFlow, for the 16 operations used in 12 popular networks, VTensor can reduce the lines of code (LOC) of writing a new operation by 47.82% on average, and improve the overall performance by 18.65% on average.

Keywords artificial intelligence (AI) programming, layout-oblivious, tensor processing

1 Introduction

As AI (artificial intelligence) technologies are quickly transforming almost every sphere of our lives, it is imperative to provide an AI programming framework that is easy to use and deploy across a variety of platforms. In the past few years, researchers have proposed a number of such programming frameworks, such as TensorFlow[1], MXNet[2], PyTorch[3], and Caffe[4], which allow users to train and develop neural network models.

However, machine learning systems are stuck in a rut. Paul Barham and Michael Isard, two of the original authors of TensorFlow, came to this conclusion in their recent HotOS paper[5]. They argued that while TensorFlow and similar frameworks have enabled great advances in machine learning, their "current programming abstractions lack expressiveness, maintainability, and modularity, all of which hinder research progress." In their paper[5], they pointed out

Regular Paper

This work was supported by the National Key Research and Development Program of China under Grant No. 2021ZD0110101, the National Natural Science Foundation of China under Grant Nos. 62090024, 61872043, and 61802368, and the Australian Research Council grant under Grant Nos. DP180104069 and DP210102409.

*Corresponding Author

©Institute of Computing Technology, Chinese Academy of Sciences 2023

that layout is one of the factors hindering the development of programming models.

Tensors, also known as NDArray, were proposed to represent multidimensional, fixed-size homogeneous array, which are widely used in AI algorithms for mathematical computations[6]. In the mathematical sense, tensors are a generalization of two-dimensional matrices, one-dimensional vectors, and also scalars[7]. For instance, when considering a representative pooling layer in a deep convolutional neural network, developers can utilize tensors to represent its input and output data. Typically, data are organized into 4-dimensional tensors, representing the number of feature maps (i.e., the batch size), the number of channels, the height, and the width of feature maps. Ideally, with tensors, developers can easily reference the logical dimensions of a data structure without the need to be concerned about the underlying physical layout.

At the application level, AI algorithm developers use the high-level interfaces provided by the AI framework to weave the network. Algorithm developers focus on the semantics of data at the application level. Consequently, people propose named dimensions[①]; in other words, tensor dimensions are associated with textual names to enhance code readability. Named dimensions improve readability by facilitating the determination of how dimensions in the code correspond to the semantic dimensions described in, for example, a research paper.

At the high-performance library level, because the layout affects the performance of library functions by affecting data locality, libraries have conventions for layout. Between high-performance libraries and applications, developers need to write a lot of layout-related code to bridge the two layers. Therefore, the challenge here is how to decouple tensor layouts from an AI programming framework, allowing developers to create layout-agnostic operators that can automatically adapt to different libraries and primitives.

Researchers have noticed that tensor layouts are a performance-critical issue and have proposed numerous approaches to determine the optimal solutions. Li *et al.*[8] analyzed the performance differences caused by different layouts and demonstrated the performance benefits obtainable by tuning the layouts for some individual operations. Anderson and Gregg[9] leveraged a Partitioned Boolean Quadratic Assignment (PBQP) formulation to select the optimal data layouts and optimal primitives. While these approaches are capable of helping developers enhance performance through layout tuning, they still rely on traditional layout-aware programming interfaces. Consequently, maintaining tensor layouts and their corresponding transformations remains to be a substantial burden for developers.

By analyzing the code skeleton in Fig.1, we find that the layout-aware programming paradigm requires writing a significant amount of layout-related code (indicated by the red lines). This extensive layout-related code adversely impacts the framework's maintainability. Furthermore, within the body of these highlighted functions (which are not shown here), there is a scattering of layout checking and transformation code. Moreover, as seen in Fig.1, the layout transformation operation is carried out within the operator. TensorFlow leverages prior knowledge to perform layout optimization. For instance, on GPU platforms, operators employ the NHWC layout due to its superior performance in most cases indicated by cuDNN[10], as compared to NCHW. However, when the neural network library cannot deduce such prior knowledge, the layout conversion operation becomes necessary within the operator, leading to missing opportunities for layout optimization. In summary, TensorFlow's ad-hoc mechanism exhibits two drawbacks.

- *Poor Maintainability.* As shown in Fig.1, when new operators or hardware is introduced, developers have to maintain these layout-dependent code segments scattered throughout the framework.
- *Unoptimized Layout Transformations.* Since the layout used by the application cannot be determined statically, and most neural network libraries do not have a dominant layout, TensorFlow's layout optimizer fails. To make matters worse, a neural network may use multiple neural network libraries at runtime. Therefore, the layout transformation operation can only be performed within the operator, which causes TensorFlow to miss the opportunity for layout optimization.

We observe that the application layer uses the mathematical semantics of the layout, while the neural network library layer employs the physical semantics of the layout. Based on this observation, we adopt the concept of polymorphism to automatically map the mathematical semantics of the layout to the

[①]Tensor considered harmful. http://nlp.seas.harvard.edu/NamedTensor, Sept. 2023.

```
The implementation consists of approximately 500 lines of code.
class MklAvgPoolingOp {
  void Compute(OpKernelContext* ctx) {
    Tensor input = ctx->input(0);
    // Resolve tensor to MklDnnShape and perform a 40-line integrity check.
    ResolveAndCheckIntegrity(input, shape, ctx);

    // Extracts dimension and attribute information, spanning 190 lines.
    PoolParameters pool_params;
    ExtractPoolParameters(ctx, &pool_params, input, shape);

    // Infer output tensor shape, which takes up 30 lines.
    memory::dims output_dims_mkl;
    InferOutShape(pool_params, &output_dims_mkl);

    // Using MKL-DNN structures to store information, occupying 70 lines.
    memory::dims filter_dims, strides, padding;
    PoolParamsToAttributes(&pool_params, &filter_dims, &strides, &padding);
    memory::dims src_dims = shape.IsMklTensor() ? shape.GetSizesAsMklDims()
                          : TFShapeToMklDnnDimsInNCHW(input, tf_format);
    memory::desc input_md = shape.IsMklTensor() ? shape.GetMklLayout()
                          : memory::desc(src_dims, tf_format, …);

    // Prepare library routine parameters, which occupy 10 lines.
    MklPoolingParams fwdParams(src_dims, output_dims_mkl, filter_dims, …);
    MklPoolingFwdPrimitive* pooling_fwd =
                          MklPoolingFwdPrimitiveFactory::Get(fwdParams);

    // Allocate output tensor, which takes up 90 lines.
    AllocateOutputTensor(0, out_tf_shape, output_tensor);
    AllocateMklShapeTensor(1, out_mkl_shape, shape_tensor);

    // Perform data conversion operations, occupying 70 lines.
    if (input_md.format != pooling_fwd->GetSrcMemoryFormt()) {
      CheckReorderToOpMem(input_md, required_layout);
    }
    pooling_fwd->Execute(src_data, output_tensor);
  }
};
```

Fig.1. Layout-aware programming for AvgPool in TensorFlow (with the layout-dependent lines shown in red).

physical semantics. In this paper, we propose VTensor, i.e., Virtual Tensor, a novel AI programming framework. VTensor provides a holistic approach for implementing layout-oblivious tensors. Specifically, the VTensor framework offers a programming interface for virtual tensors to decouple the physical semantics of layout from the programming framework, thereby providing operator developers with a layout-oblivious programming perspective. Furthermore, the VTensor framework incorporates a global runtime layout inference mechanism that transparently resolves the physical semantics of VTensor by analyzing the layout conventions of all operators and underlying library routines. To support efficient execution of VTensor applications, we extend the dataflow graph to explicitly represent tensor layout transformation operations as individual nodes in the graph, and perform layout-oriented graph optimizations to minimize layout transformations.

We implement VTensor on top of TensorFlow. This paper makes the following contributions.

- We propose a layout-oblivious programming model for developers, so that they do not have to be concerned themselves with the physical layouts of tensors and associated tedious layout transformations when developing new operators.
- We propose a tensor layout resolution mechanism. This mechanism explicitly exposes the layout convention for each individual operation or library routine and automatically infers the layout needed for each operation, inserting appropriate layout transformations when necessary.
- We present a global graph optimization enabled by VTensor, i.e., layout-oriented optimization. VTensor defers the timing of layout selection to the runtime phase through partial evaluation, thus creating opportunities for comprehensive optimization of conversion operations. The layout-oriented optimiza-

tions include: eliminating redundant layout transformations based on the graph structure and formulating the selection of the layout for element-wise operators with broadcast semantics as an Integer Linear Programming (ILP) problem.

• We implement VTensor in TensorFlow to showcase its substantial impact on improving the maintainability and extensibility of existing AI programming frameworks. Specifically, when developing a new operation, VTensor can reduce its LOC by 47.82% on average. Furthermore, VTensor outperforms TensorFlow by 18.65% on average for the 12 popular networks evaluated.

The remainder of this paper is structured as follows. Section 2 provides an introduction to the background and motivation. Section 3 introduces the VTensor framework. In Section 4, we present the VTensor programming interface. Section 5 delves into the VTensor runtime. Our experimental validation is detailed in Section 6. Section 7 offers a comprehensive discussion of the VTensor framework in relation to portability, programming efforts, and its connection with AI compilers. Section 8 explores the related work. Finally, Section 9 concludes the paper.

2 Motivation and Background

In this section, we initially introduce the layout-aware programming model from a dataflow graph perspective (Subsection 2.1). Subsequently, we utilize examples in the following two subsections to illustrate the challenges associated with the layout-aware programming model, focusing on maintainability (Subsection 2.2) and layout optimization (Subsection 2.3).

2.1 Design of TensorFlow

In TensorFlow, neural networks are represented as data flow graphs. A data flow graph is a directed acyclic graph in which each node represents a mathematical operation and each edge represents a multidimensional data, known as tensors, upon which these operations operate.

A kernel is an implementation of an operator specific to a particular library. As illustrated in Fig.2(a), TensorFlow employs tensors to traverse various library-based kernels. Since tensors are not decoupled from the layout and tensors are used at both the graph level and the neural network library level, the entire framework is tightly coupled to the layout. Therefore, operator developers need a layout-aware programming model for operator development.

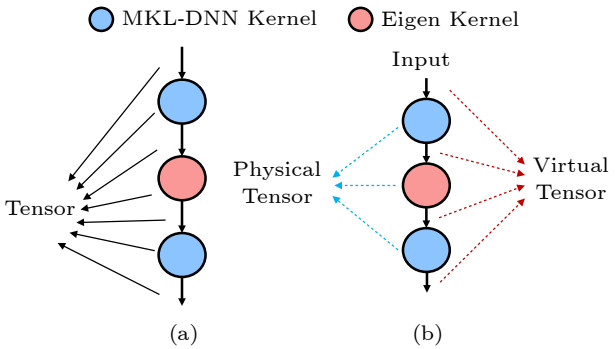

Fig.2. Tensor designed by (a) TensorFlow and (b) VTensor.

In contrast, we abstract the semantics of tensors at both the graph and library levels, representing them as virtual tensors and physical tensors, respectively, thereby decoupling the framework from the layout, as illustrated in Fig.2(b). Thus, the operator developer is decoupled from the layout.

2.2 Poor Maintainability

When writing an operator, the primary concern for the operator developer is to reorganize the input data into a format accepted by the library function and then invoke the library function to obtain the output result. However, the challenge lies in the fact that the operator developer cannot determine the layout of the input data and the optimal layout required by the library function at the compile phase. This leads to the operator developer having to consider all possible mapping relationships. Additionally, each library has its own unique data structure to represent the layout and operator primitives, necessitating operator developers to rewrite each operator for each library. Furthermore, the layout design of the library function involves a comprehensive consideration of algorithms and architectures. As a result, the relationship between the input layout and the library-level layout is not a simple injective and subjective one. Consequently, we argue that operator code developed based on the TensorFlow framework suffers from poor maintainability.

Taking the example of the AvgPool operator written based on MKL-DNN[②] (refer to Fig.1), the operator developer first obtains the shape of the input ten-

[②]MKL-DNN, Intel math kernel library for deep neural networks. https://01.org/onednn, Sept. 2023.

sor for shape inference and legitimacy checks, as highlighted in red in the upper half of Fig.1. Since the input data of the operator may originate from the output of the operator implemented based on the Eigen[③] library, the operator developer needs to insert multiple branch statements and write code to extract the shape from a specific data structure in turn. Next, the operator developer creates primitives based on the shape of the input tensor and other parameters. During this process, MKL-DNN will determine the layout based on the input parameters and the shape of the input tensor. Subsequently, the operator developer allocates the output space and performs data layout conversion. The output memory space includes not only the output data *output_tensor*, but also the *shape_tensor* describing the attribute information of this tensor, corresponding to the red code in the lower half of Fig.1. Finally, the operator developer carries out the pooling operation by invoking the *Execute* function.

Based on the aforementioned observations and analysis, we introduce the concepts of virtual tensors and physical tensors. Similarly, an operator is divided into a virtual operator and multiple physical operators. The virtual tensor circulates among virtual operators, while the physical tensor is exclusively used in physical operators. The mapping of virtual tensors to physical tensors is accomplished by the framework's dynamic layout resolver. The mapping from virtual operators to physical operators is achieved through the automatic generation of a dispatch function based on library priority.

Fig.3 illustrates how to implement the AvgPool operator in the VTensor framework. It demonstrates that by using the virtual tensor APIs (indicated by orange lines), developers can focus solely on the logical computation of the operation (represented by black lines) and access tensor information without needing awareness of the physical layout. Furthermore, the layout checking and the library wrapper code are generated automatically by the VTensor framework (as seen in the blue lines). Developers use the physical tensor APIs to declare the corresponding physical tensors for virtual tensors, and the VTensor framework automatically maintains the layout information through the require/produced attributes.

We employ LOC as a metric to demonstrate the advantages of the layout-oblivious programming paradigm. Fig.1 and Fig.3 depict the number of LOC required by an operator developer to implement an AvgPool operator, excluding comments (represented by green lines) and automatically generated code (indicated by blue lines). The total LOC in Fig.1 is approximately 500, whereas the total LOC in Fig.3 is around 140. Fig.3 achieves a reduction of 360 LOC compared with Fig.1. In particular, the black and orange lines are written by developers (with the orange lines denoting VTensor API calls), while all the blue lines are automatically generated by VTensor. The analysis above highlights that the layout-oblivious programming offered by VTensor significantly reduces the number of LOC required for operator development.

2.3 Unoptimized Layout Transformations

The choice of the layout in TensorFlow Grappler depends on various factors, including the operator's parameters, algorithms, accepted data types, and more. This means that the layout cannot be statically determined. TensorFlow Grappler[④] includes two layout optimizers: one for GPUs and the other for CPUs. The GPU layout optimizer uses predefined rules based on experience to determine the operator layout. For instance, when dealing with a convolution operator that does not trigger Tensor Core, the NCHW layout generally performs better than the NHWC layout. On the other hand, the CPU layout optimizer[11] divides the dataflow graph into multiple subgraphs based on whether the operator is implemented by MKL-DNN library functions. These data conversion operations within the subgraph are performed by the operator itself. The optimizer inserts data conversion operations between these subgraphs. XLA[⑤] (Accelerated Linear Algebra) is a domain-specific compiler in TensorFlow capable of generating low-level IR for networks. When XLA utilizes the library as a backend, the layout assignment optimizer follows a similar design as TensorFlow's optimizer.

In Fig.4(a), we illustrate a scenario where the ResNet network uses MKL-DNN as the backend. Here, the output layout of the Concat operator de-

[③]Jacob B, Guennebaud G, Avery P *et al*. Eigen. https://eigen.tuxfamily.org, Jan. 2019.

[④]Larsen R M, Shpeisman T. TensorFlow graph optimizations. https://storage.googleapis.com/pub-tools-public-publication-data/pdf/baea094c34ab32f24e7883096e212baa4578cbda.pdf, Jan. 2019.

[⑤]Google. XLA: Optimizing compiler for TensorFlow. https://tensorflow.org/xla, Jan. 2019.

```
The total lines of code for the AvgPoolOp and invoker amount to 140.
class AvgPoolOp {
  void Compute(OpKernelContext* ctx) {
    VTensor input = context->input(0);
    // Extracting parameters, approximately 50 lines.
    VPoolParameters param;
    ExtractPoolParametersFromVTensor(ctx, &param, input);
    // Infer output tensor shape, which takes up 20 lines.
    InferOutShape(&param);
    // Allocate output tensor. Omit settings for other parameters.
    VTensorOptions* vto(ctx, 0/*output index*/);
    VTensor* output_tensor = new VTensor({{'N', param.batch_size}, …}, vto);
    Dispatcher(ctx, {input}, {output_tensor},  &param);
  }
  void Dispatcher(OpKernelContext* ctx, vector<VTensor> input,
                              vector<VTensor*> output, Parameters* param) {
    ParametersAndDimensionCheck(ctx, input, output, param);
    auto invoker_dict = getDeviceInvokers(ctx->device());
    for (int priority = highest; priority >= lowest; priority--) {
      Invoker* invoker = invoker_dict.getInvoker(priority);
      if (invoker) { invoker(ctx, input, output, param); return;}
    }
  }
};
REGISTER(CPUDevice, "AvgPool", mkldnnAvgPoolInvoker)
void mkldnnAvgPoolInvoker(OpKernelContext* ctx, vector<VTensor> input,
                            vector<VTensor*> output, Parameters* param) {
  // Utilize VTensor to construct PTensor.
  PTensor* in_p = new VTensor(input[0]);
  LAYOUT p_layout = GetLibGuideLine("MKLDNN", in_p);
  in_p->require(p_layout);
  if (LibTagger("MKLDNN", {in_p}, {out_p}) return;
  MemoryDesc in_desc = in_p->getLibraryDesc("MKLDNN");
  MemoryDesc out_desc = out_p->getLibraryDesc("MKLDNN");
  // Prepare MKL-DNN library routine parameters in 40 lines.
  MklPoolingParams fwdParams;
  ConstructMklPoolParams(param, fwdParams, in_desc, out_desc);

  MklPoolingFwdPrimitive* pooling_fwd
                      = MklPoolingFwdPrimitiveFactory::Get(fwdParams);
  pooling_fwd->Execute(in_p->data(), out_p->data());
}
```

Fig.3. Motivation example: AvgPool operator in VTensor (layout-oblivious programming) (blue lines are auto-generated and orange lines are VTensor API calls).

pends not only on the input layout but also on specific implementation details, such as how to choose an output layout based on different input layouts. The required layout for the MKL-DNN convolution operator is also statically uncertain. Therefore, TensorFlow delegates the layout conversion operation to the operator itself, as exemplified by the *CheckReorderToOpMem* function in Fig.1. Additionally, TensorFlow lacks an interface to describe library-specific layouts, which may introduce unnecessary layout transformations during ad-hoc layout processing.

In contrast, as shown in Fig.4(b), the VTensor framework utilizes partial evaluation passes to collect layout information for each tensor. It then conducts global layout-oriented graph optimizations. Specifically, the VTensor framework creates an individual node for layout transformation and places it immediately after the Concat operation. As a result, only one transformation is required.

3 VTensor Framework Overview

Fig.5 shows the overall VTensor framework, comprising the "VTensor Programming Framework" (the bottom part) and the "VTensor Runtime" (the upper part), serving for the programming interface and runtime support respectively.

The VTensor programming framework provides four categories of programming interfaces to define an operation, describe a library, and illustrate how to invoke a library routine.

• The virtual tensor API (VTensor API) empowers developers to implement operators by writing the *Compute* function to access virtual tensors, such as

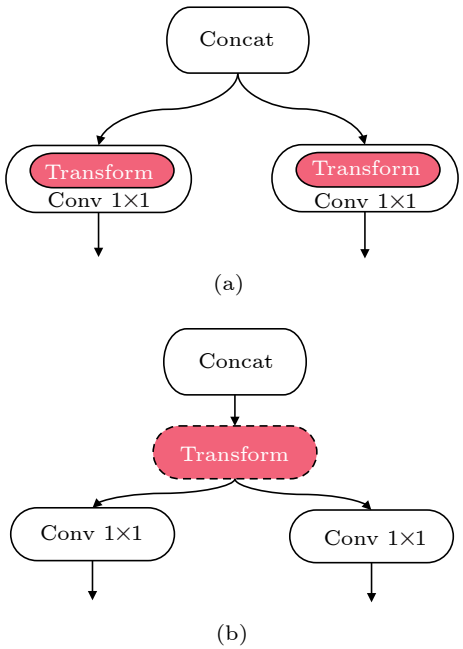

Fig.4. Location where the layout transformation operation takes place. (a) Inside the operator. (b) Outside the operator. Conv: convolutional.

constructing a virtual tensor or accessing a dimension. The corresponding code is illustrated in the green lines in Fig.5. In each operation, the *Dispatcher* API serves as the unified entry point for all libraries and devices. It uses a list of abstract parameters, and its function body is automatically generated by the VTensor Framework.

• The physical tensor (PTensor) API enables developers to declare physical tensors corresponding to virtual tensors, as demonstrated by the brown lines. Typically, the PTensor API is written within the *libraryInvoker* function to invoke a specific library routine. In contrast to virtual tensors, a physical tensor contains its physical layout information, which can be a specific layout (e.g., LAYOUT::NCHW) or without constraints (i.e., LAYOUT::ANY). Additionally, the PTensor API provides a "require" function to declare the physical layout convention for the library routine.

• A library description is essential to facilitate multi-library support. This file is required for each library to describe the mapping of physical layouts used by PTensors to library-specific layout names. For example, it maps from NCHW8c to nChw8c in MKL-DNN. Furthermore, the library description specifies the layout transformation handler within the library and offers guidelines for selecting a layout from multiple alternatives. The corresponding code is depicted by the red lines in Fig.5.

• Framework APIs allow developers to register handlers with the VTensor framework, as indicated by the blue lines in Fig.5. The VTensor framework will invoke these handlers during runtime layout resolution. In particular, for each library routine, a *libraryInvoker* function must be written and registered as the entry point for invoking the routine. This function describes how to create the actual parameters

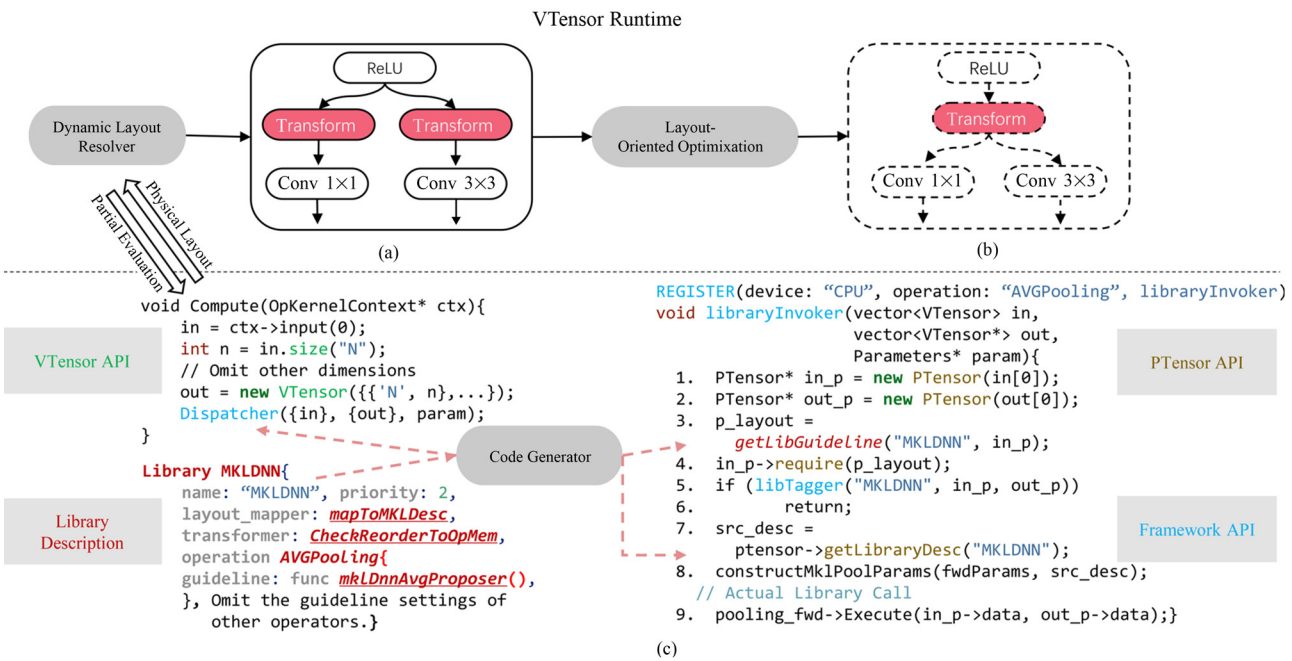

Fig.5. VTensor framework overview. (a) Extended data flow graph. (b) Optimized data flow graph. (c) VTensor programming framework. Conv: Convolution.

from the abstract parameter list. Furthermore, a tagger can be embedded within *libraryInvoker*, serving as a safeguard for partial evaluation during runtime layout resolution. Code following the tagger will not be executed during partial evaluation, but will run during normal graph execution.

To resolve layouts for virtual tensors, we propose a dynamic layout resolver (DLR). DLR partially evaluates the dataflow graph to determine the physical layout for each node in the graph. It identifies the necessary locations for layout transformations and inserts corresponding transformation operations as individual nodes into the dataflow graph (as shown in Fig.5(a)), referred to as the extended dataflow graph. Finally, we apply layout-oriented optimizations (LOOs) to the extended dataflow graph to optimize the layout transformation nodes (as shown in Fig.5(b)). Further details will be discussed in Section 5.

4 VTensor Programming Interface

The VTensor framework provides four categories of programming interfaces: VTensor APIs, PTensor APIs, library description, and framework APIs. Developers can leverage these APIs to implement a tensor operator, i.e., the *Compute* function in Fig.5, in a layout-oblivious manner. To distinguish between the virtual tensor class and the programming model with the same abbreviation, we employ italics to represent the virtual tensor class (*VTensor*) and the regular font for the programming model (VTensor).

4.1 VTensor Class and PTensor Class APIs

Fig.6 demonstrates the APIs of the virtual tensor class and the physical tensor class. The virtual tensor class serves as intermediate data that connects two operators and remains library-agnostic in terms of its layout. When utilizing a virtual tensor instance, developers need not concern themselves with its layout, which can remain virtual until a specific library routine is invoked. Typically, such routines follow predetermined layout conventions for parameters. As a result, we introduce the physical tensor class (*PTensor*) to describe the physical layout of a virtual tensor instance. We categorize these APIs into three groups based on whether they are specific to the *VTensor* class, the *PTensor* class, or shared between them: VTensor-specific APIs, PTensor-specific APIs, and common APIs.

VTensor-Specific APIs. The *VTensor* class has its own specific constructor with three parameters: *scalar_type*, *shape*, and *option*. In particular, *shape* is an array of dimensions, where each dimension consists of two parts: the dimension name and the dimension size.

scalar_type represents the data element type, while *option* is used to provide the necessary context for space allocation and the index number corresponding to the output tensor. An example of using the *VTensor* class constructor is shown as follows: $VTensor\ vt(float,\ \{('N',\ NS),\ ('C',\ CS),\ ('H',\ HS),('W',WS)\},\ \{ctx,\ 0\})$ where *ctx* is an instance of the *OpKernelContext* class in TensorFlow.

PTensor-Specific APIs. The APIs of the *PTensor* class are provided to express specific layout conventions.

- $PTensor(VTensor\ v)$. It constructs a *PTensor* instance from the corresponding *VTensor* instance.
- $void\ require(LAYOUT\ playout)$. It declares the

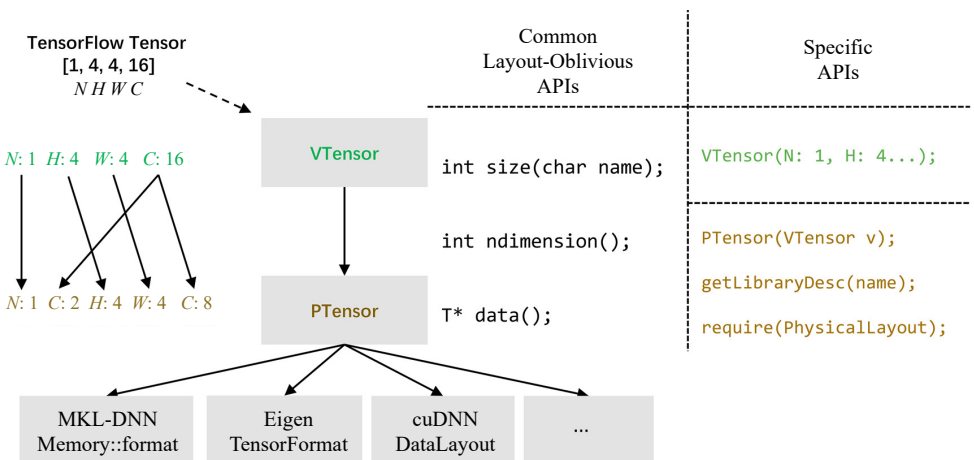

Fig.6. APIs provided by virtual tensor and physical tensor.

physical layout convention of the library routine, which can be called only once in each operation. *LAYOUT* is an enumerated type used to uniformly represent the physical layout of all libraries. A specific LAYOUT::ANY can be provided if the underlying library routine can accept any layout, as commonly seen in element-wise operators (e.g., ReLU). Other values of *LAYOUT* are user-defined. Taking the LAYOUT::HWCN for example, cuDNN uses kYXDepthBatch as the keyword, while MKL-DNN uses hwio, and Eigen uses FORMAT_HWCN.

• *getLibraryDesc*(*string lib_name*). It retrieves the corresponding memory description of the library routine, and the function body is auto-generated. The value of *lib_name* should be consistent with the value of *name* in the library descriptor.

Common APIs. These APIs serve to access a virtual tensor or physical tensor, and they are implemented as member functions of *VTensor* and *PTensor*.

• *int size*(*char dim_name*). This function returns the dimension size corresponding to *dim_name*. We follow the dimension naming conventions of the MKL-DNN library, where *N* represents the batch size, *C* represents the number of channels, *H* represents the image height, *W* represents the image width, and *D* represents the image depth. The value of *dim_name* can be any of the aforementioned symbols.

• *int ndimensions*(). This function returns the number of dimensions.

• *T data*(). This function returns the raw pointer to the actual data.

4.2 Library Description

The library description interface enables developers to integrate a new library as a plug-in. In particular, developers can describe a library from four aspects.

Library Descriptor. Developers are required to specify the *name* and *priority* of the library. *name* is the sole identifier of a library in our framework, while *priority* determines library selection when an operator is implemented using multiple libraries. These properties will be adopted by our code generator to produce some API implementations, e.g., *Dispatcher*.

PTensor Layout Mapping. Within the *PTensor* class, we employ a unified layout representation across all libraries. However, various libraries may employ distinct keywords for the same layout. To bridge this gap, developers are encouraged to employ *layout_mapper* for mapping a *LAYOUT* to library-specific keywords. This involves specifying keyword-*LAYOUT* pairs to harmonize library keywords with the enumeration values.

Layout Transformation Handler. Neural network libraries often include routines for layout transformations, such as *CheckReorderToOpMem* in MKL-DNN and *TransposeUsingEigen* in the Eigen library. These routines enable our VTensor framework to invoke them as needed. Developers can provide these routines as handlers, using the *transformer* keyword (see Fig.5). Handlers must adhere to the following interface definition:

$$void\ TransformHandle(PTensor\ src,$$
$$PTensor\ dst,\ vector < int >\ permutation).$$

Layout Guideline. In state-of-the-art accelerating libraries, the layout convention may not be entirely static. For instance, MKL-DNN offers the *mklDnnAvgProposer* routine to determine the layout convention at runtime. To accommodate such scenarios, we provide the *guideline* keyword for developers. It allows them to specify a function that operates on *PTensor* objects as parameters and dynamically determines the layout convention. The layout guide function must adhere to the following interface definition: $LAYOUT\ LayoutGuideLine(PTensor\ tensor)$.

4.3 Framework APIs

When developers create a new operation, some interactions with the VTensor framework are inevitable. Therefore, we provide a set of framework APIs to facilitate these interactions, as follows.

• *Dispatcher* is the unique entry for invoking libraries inside an operation. It encapsulates the kernel computations from different libraries, and its function body is automatically generated by the VTensor framework.

• *libraryInvoker* is the entry for invoking one library routine, which will be automatically called by *Dispatcher*. Developers need to write the function body and register it to the VTensor framework.

• *libTagger* serves as a safeguard for partial evaluation during runtime layout resolution. Code following the tagger will not execute during partial evaluation, but will run during normal graph execution.

Since different library functions have different function signatures, *libraryInvoker* needs to be writ-

ten by operator developers. Fig.5 shows how to use the PTensor API and the Framework API to write the *libraryInvoker*. To begin, essential *PTensor* instances are created for input *VTensor* instances and output *VTensor* instances (line 1 and line 2, respectively). Next, the required layout of input PTensors is specified using the *require* API (lines 3 and 4). The tagger is then inserted to indicate that the subsequent statement is a library call, ensuring that layout information is passed to the runtime for partial evaluation and layout resolution (lines 5 and 6). Abstract parameters in *Dispatcher* are extracted to generate the actual parameters for invoking the library routine (line 7). Finally, the library is invoked (line 8).

4.4 Automatically Generated Code

As previously discussed, the VTensor framework automatically generates three functions: *Dispatcher*, *getLibGuide*, and *getMemoryDesc*. These functions are designed to insert developer-provided plug-in handlers at appropriate points in the process. Furthermore, the runtime (as discussed in Subsection 5.1) introduces a data transformation operator into the dataflow graph. The VTensor framework then generates the transformation operator's body based on the provided *transformer* specified in each library description.

We leverage template-based code generation here. Specifically, functions like *Dispatcher* are utilized to choose an appropriate function to invoke based on the runtime input parameters or to provide corresponding values according to the runtime input parameters. We represent this process as a code template. Taking the *Dispatcher* function (code highlighted in blue in Fig.3) as an example, it serves to distribute operators to an invoker. In our implementation, *Dispatcher* selects the invoker based on the device type and the library's priority. Each operator has different check conditions for different parameters and dimensions. Consequently, legitimacy check code is generated by the code generator as part of the *ParametersAndDimensionCheck* function.

5 VTensor Runtime

In this section, we present the design and implementation of the VTensor runtime. It comprises two components: a DLR (Subsection 5.1) responsible for inferring the physical layouts of VTensor instances and inserting the necessary data conversion nodes to extend the dataflow graph, and a LOO (Subsection 5.2) tasked with optimizing the extended dataflow graph.

5.1 Dynamic Layout Resolver

The DLR serves to resolve a layout-oblivious *VTensor* instance to a *PTensor* instance with a specific physical layout defined by the library. The Grappler is the default graph optimization system in TensorFlow. The DLR is implemented as part of the TensorFlow Grappler after graph partitioning and device placement.

Data Structure for Layout Resolution. To assist in resolving the layout, two attributes are introduced: *produced_layout* for the *VTensor* class, and *required_layout* for the *PTensor* class. Specifically, *required_layout* indicates the physical layout required by the library routine, specified using the *require* statement in *libraryInvoker*. *produced_layout* indicates the physical layout derived from predecessor nodes. A *VTensor* instance is successfully resolved to a *PTensor* instance when its *produced_layout* matches the corresponding *required_layout* of *PTensor* instance.

Overall Workflow. The DLR works as follows. First, it sets the execution mode to *Partial* and utilizes partial evaluator (PE) to determine *produced_layout* for each *VTensor* instance and *required_layout* for each *PTensor* instance by executing only the code in the graph related to layout determination. This avoids executing time-consuming computations in the operator, making the DLR's cost negligible. After PE processing, the layout of all *VTensor* instances is resolved, as shown in Fig.7(b). Second, the DLR checks the layout attributes for each pair of the *VTensor* instance and *PTensor* instance, inserting a data conversion node (the "transform" node in red) when *produced_layout* does not match *required_layout*, as shown in Fig.7(c).

Partial Evaluation Algorithm. Determining *required_layout* is challenging for three reasons. First, developers may choose different libraries within a single operation based on input parameters (e.g., preferring cuBlas over cuDNN for a "1×1" convolution). Second, operator parameters, algorithms, accepted data types and other factors can affect layout conventions. Third, the application's layout can only be de-

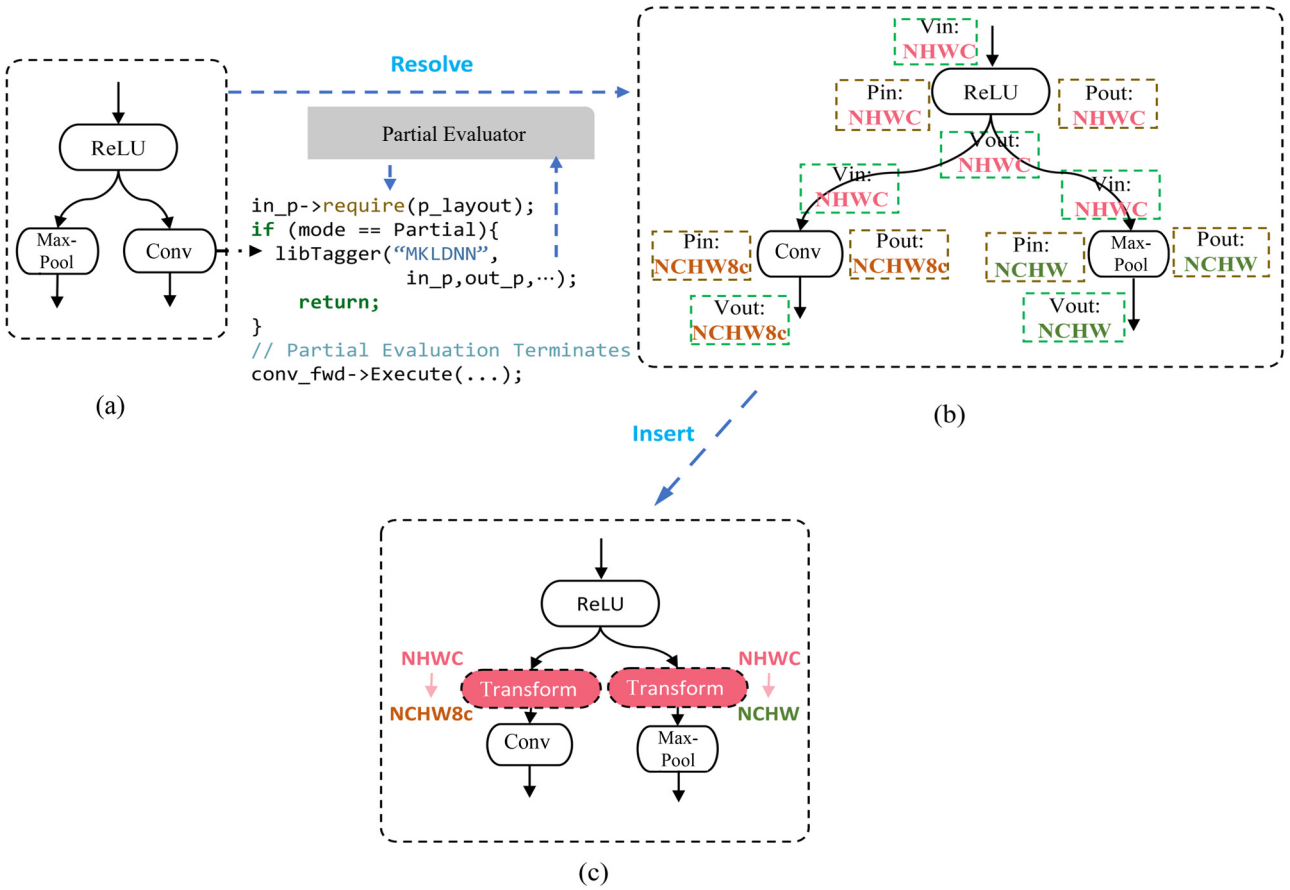

Fig.7. DLR workflow. Vin and Pin refer to *VTensor* and *PTensor* inputs respectively. Vout and Pout are *VTensor* and *PTensor* outputs respectively. (a) Original data graph. (b) Resolved layout. (c) Extended flow graph. Conv: convolution.

termined at runtime.

Hence, as shown in Fig.8, we propose a partial evaluation algorithm to obtain layout information. The algorithm traverses the dataflow graph, processing each node in topological order (lines 2–4, and lines 23–28). Before traversing, for the original data flow graph G's entry node, the *produced_layout* of all its input *VTensor* instances is initialized based on the input format (line 1).

When processing each node, PE partially executes the operation code until encountering the developer-specified *libTagger*. The layout resolving algorithm works as follows. First, we extract the input and output *VTensor* instances of the current node into *vtensor_set* (line 5). Second, for each *VTensor* instance in the set, we create a *PTensor* instance, determining its physical layout per developer-provided library guidelines (lines 8 and 9). If the determined layout is LAYOUT::ANY, it inherits the physical layout of the first input *VTensor* instance (if it is an output) or inherits the *produced_layout* of the *VTensor* instance (if the tensor is an input) (lines 10–15). The determined physical layout is recorded in the *required_layout* of the *PTensor* instance (line 16), indicating the library's layout requirement. Third, for each output *VTensor* instance, its *produced_layout* is set as the *required_layout* of the corresponding *PTensor* instance for propagating the resolved layout to successor nodes (lines 17 and 18). Finally, the developer-specified *libTagger* function is invoked to skip the subsequent library function call and terminate the partial evaluation of the current node (line 21).

5.2 Layout-Oriented Optimization

After explicitly inserting layout transformation operations into the data flow graph, new opportunities for globally optimizing the transformations emerge.

The optimization goal of this paper is to minimize the number of data conversion operation. To minimize the end-to-end latency of the neural network, factors such as layout, primitives, and memory footprint need to be considered together. However,

```
Algorithm 1. Layout Resolving Algorithm
Input: G: original data flow graph
       Entry_Layouts: the input layouts of network
Output: Tensor_Set: the V/PTensors of each node
 1.InitializeEntryNodeLayout(G, Entry_Layouts)
 2.ready_queue = {G.entry_node}
 3.while ready_queue not empty do                    // Topology-based traversal of data flow graphs
 4.  work_node = ready_queue.pop()
 5.  vtensor_set = work_node.vtensor_set()          // Retrieve the input and output VTensor instances
 6.  ptensor_set = {}
 7.  for vtensor in vtensor_set do
 8.    ptensor = PTensor(vtensor)
 9.    physical_layout = work_node.InvokeLibGuideline(ptensor)
                                                  // Determine the physical layout of PTensor instances
10.    if physical_layout == LAYOUT::ANY then
11.      if vtensor.is_out then
12.        physical_layout =
13.          GetFirstInputTensor(work_node).required_layout
14.      else                                     // Inherit the input VTensor instances's required_layout
15.        physical_layout = vtensor.produced_layout
                                                    // Inherit the VTensor instance's produced_layout
16.    ptensor.required_layout = physical_layout
17.    if vtensor.is_out then
18.      vtensor.produced_layout = ptensor.required_layout
                                                  // Distribute the resolved layout to successor nodes
19.    ptensor_set.append(ptensor)
20.  end for
21.  InvokeLibTagger(Tensor_Set, {work_node, ptensor_set})
22.
23.  for succ in work_node.successors() do
24.    PropagateVTensorToSuccessor(voutputs, succ)
25.    if IsReady(succ) then
26.      AppendReadyNode(ready_queue, succ)
27.  end for
28.end while
29.return Tensor_Set
```

Fig.8. Layout resolving algorithm.

these factors are beyond the scope of this paper.

As TensorFlow's layout optimization algorithm is designed without knowing the layout information, the need arises for a new layout optimizer. In the VTensor framework, we introduce two optimization components: layout hoisting optimization (LHO) and element-wise optimization (EWO).

Layout Hoisting Optimization. LHO is proposed to apply the optimization presented in Fig.4 to multi-branch models. We employ a pattern-based graph optimization to hoist the layout transformation nodes. Specifically, we traverse all the branch nodes in the graph from top to bottom. If a branch node satisfies: 1) all of its immediate successors are layout transformation nodes, 2) each immediate successor has a *PTensor* instance generated from a common *VTensor* instance, and 3) all of these PTensors have the same *required_layout*, then the layout transformation nodes from all immediate successors can be hoisted up before the branch node.

Element-Wise Optimization. Element-wise nodes can accept an arbitrary layout of tensors as input in mathematical semantics. However, element-wise operations may have broadcasted semantics, requiring the layouts/shapes of their input tensors to be compatible[12]. However, in TensorFlow, the layout of an input tensor can only be determined at runtime. Consequently, TensorFlow conservatively restricts the layout as NHWC, as shown in Fig.9(a).

Furthermore, the selection of the output layout of a producer operator determines the input layouts of all connected consumer operators. Thus, the layout selection cannot be made in isolation.

It is important to note that operators such as convolution specify the required layout through the *require* statements, and we assume that this specification cannot be altered.

Given an extended data flow graph G, we introduce a *candidate_set* variable for each node, except for constant operations, and initialize the variable according to the following rules.

• *Element-Wise Operators with Broadcast Semantics.* Assuming tensor A is broadcast to tensor B, with the layout of tensor A being LA and the layout of tensor B being LB. If tensor A cannot be represented by LB (for instance, when the channel size of A is not

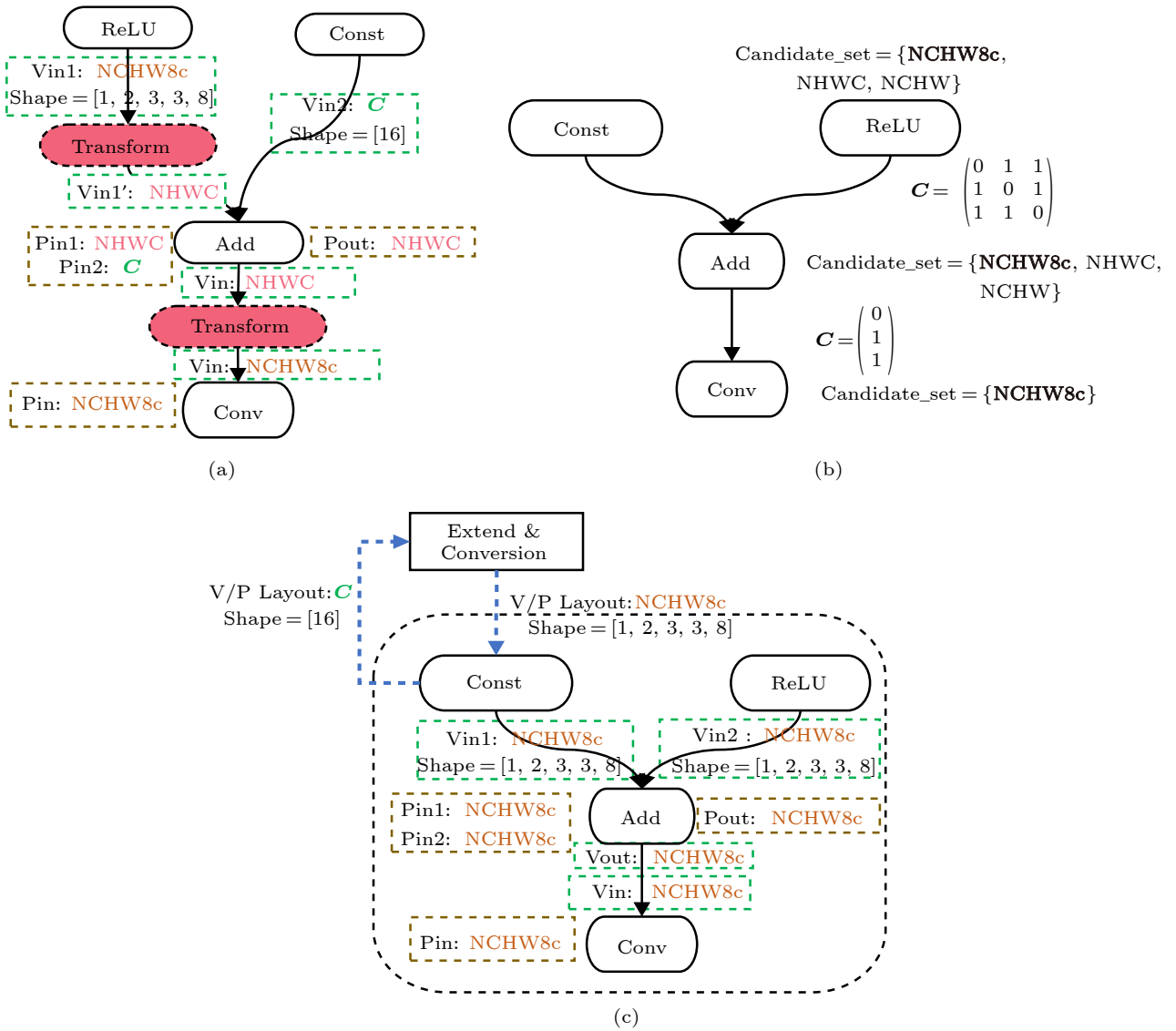

Fig.9. Example for EWO. (a) Extended data flow graph. (b) Data flow graph with candidate_set and constant matrix. (c) Data flow graph after EWO. Const: constant; Conv: convolution.

a multiple of 16, i.e., it cannot be represented by NCHW16c), then *candidate_set* is assigned to {NHWC, NCHW}; otherwise, it is assigned to {LB, NHWC, NCHW}.

- *Normal Element-Wise Operators.* *candidate_set* is assigned to {the layout of inputs, NHWC, NCHW}.
- *Other Operators.* For the other operators, e.g., convolution, *candidate_set* is assigned to attribute *required_layout*.

The cost of each edge is determined by the layout of the two nodes connected by the edge. We initialize the cost matrix as follows:

$$\boldsymbol{C}_{ij}[m][k] = \begin{cases} 1, & \text{if } candidate\_set_i[m] \neq \\ & \quad candidate\_set_j[k], \\ 0, & \text{otherwise,} \end{cases}$$

where i, j represent nodes, and $candidate\_set_i[m]$ represents the m-th candidate layout of node i, as shown in Fig.9(b). Constructed in this manner, we can map the layout selection problem to an ILP problem as follows:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{n}\sum_{j=1}^{n} \boldsymbol{x}_i{}^{\mathrm{T}} \boldsymbol{C}_{ij} \boldsymbol{x}_j \\ \text{subject to} \quad & \boldsymbol{x}_i \in \{0,1\}^{|C_i|}, \qquad i = 1, ..., n, \\ & \boldsymbol{x}_i{}^{\mathrm{T}} \mathbf{1} = 1, \qquad i = 1, ..., n. \end{aligned}$$

Our objective is to minimize data conversions within the graph by selecting the layout for element-wise operators. Each node is associated with a boolean decision vector, denoted as $\boldsymbol{x}$, where the elements are binary, taking values of either 0 or 1. An additional constraint is imposed, ensuring that only a

single element within these vectors is assigned the value 1. Subsequently, we employ an ILP solver to identify the optimal solution. Following this, if we encounter an element-wise operator with broadcast semantics and its input tensor contains a constant tensor, the VTensor framework extends and transforms the constant tensor to match the layout of the non-constant tensor, as illustrated in Fig.9(c).

After applying the EWO and LHO optimizations, we obtain an optimized extended data flow graph. In this graph, each node represents an instance of a computation operation or a layout transformation operation, and each edge represents a multidimensional dataset, specifically tensors, on which the operations are performed.

5.3 Implementation

We plug the implementation of DLR and LOO into the Grappler as a pass to implement the runtime of VTensor framework. Specifically, we begin by extending TensorFlow's *Executor* class to implement the partial evaluator (PE). The PE initiates TensorFlow's runtime to execute the entire network. Subsequently, we modify TensorFlow's runtime module to enable runtime layout information retrieval. This modification involves adapting TensorFlow's *OpKernelContext* class to facilitate the circulation of *VTensor* class instances between operators, and enhancing TensorFlow's profiler to record operators' layout information. The implementation of these changes consumes approximately 1 500 LOC.

Afterwards, we convert the layout selection problem into an integer programming problem, leveraging the collected layout information. This transformation requires approximately 2 000 LOC. To conclude, we introduce the data conversion operation into the diagram, necessitating about 200 LOC for its implementation. Additionally, we extend the *VTensor* and *PTensor* classes based on TensorFlow's *Tensor* class. The implementation of both classes comprises over 600 LOC. Furthermore, within the VTensor framework, we implement 16 operators, resulting in a total of approximately 5 600 lines of code.

6 Evaluation

6.1 Experiment Setup

Hardware and Software Platforms. We evaluate VTensor on both CPU and GPU platforms. The CPU platform comprises a quad-socket server, with each socket housing a Westmere-based Intel 2.0 GHz octa-core Xeon E7-4820 processor. Each processor features a private 32 KB L1 D-cache and 32 KB L1 I-cache, a private 256 KB L2 cache, and a shared 18 MB L3 cache. The GPU platform is a Volta-based NVIDIA TITAN V GPU, featured with 80 SMs and 12 GB global GDDR5 memory. VTensor is implemented on top of TensorFlow 1.14. We utilize Intel MKL-DNN (v0.18) and cuDNN (7.2) as the vendor-provided libraries for CPU and GPU acceleration, respectively. In addition, TensorFlow caches the optimized graph for a compute graph that is repeatedly executed, ensuring that the graph optimization system executes it only once. The evaluation process comprises two stages: the warm-up stage and the test stage. During the warm-up stage, we measure the execution time of DLR modules and layout optimization. In the test stage, we obtain the network execution time by executing one sample at a time and averaging the execution time across 1 000 samples.

VTensor-Powered Operators and Networks. In this paper, we focus on deep learning inference. To accomplish this, we establish a benchmark set comprising 12 DNN models, namely Inception[13–16], ResNet[17], VGG[18], DenseNet[19], MobileNet[20, 21], and NasNet[22]. Detailed information about these models can be found in Table 1. We have extracted 16 operators from the aforementioned benchmark set and subsequently re-implemented them using VTensor. All the code adheres to the Google code style[⑥], which is the default style for TensorFlow developers. We manually count LOC of both TensorFlow operators and VTensor operators.

Performance Baseline. For the CPU and GPU platforms we evaluate, we follow the approach outlined in [23] to configure TensorFlow's tunable parameters. These parameters, which include inter_op_parallelism_threads (inter-op), intra_op_parallelism_threads (intra-op), and KMP_BLOCKTIME, are listed in Table 1 and serve as the performance baseline. The batch size is consistently set to 1 for all networks. Intel's implementation outperforms the Eigen CPU backend by up to 70x and has been seamlessly integrated into the TensorFlow framework[11]. We utilize its layout optimizer as the baseline for the CPU side. Regarding the GPU backend, both XLA and Tensor-

[⑥]Weinberger B, Silverstein C, Eitzmann G *et al.* Google C++ style guide. Section: Line Length. 2013. http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Line%_Length, Jan. 2019.

Table 1. Parameters of the Deep Learning Models in Evaluation

Model	Image Size	KMP_BLOCKTIME	inter_op	intra_op	Number of Layers	Number of Parameters ($\times 10^6$)
GoogleNet	224×224	1	2	16	174	6.80
Inception_Resnet_V2[13–16]	299×299	1	2	16	772	56.00
Inception_V2[13–16]	224×224	1	2	16	212	11.26
Inception_V3[13–16]	299×299	1	2	16	287	23.94
NasNet-Large[22]	331×331	1	2	16	1 142	89.15
ResNet_101[17]	224×224	1	1	32	349	44.76
ResNet_152[17]	224×224	1	1	32	519	60.50
ResNet_50[17]	224×224	1	1	32	179	25.66
VGG19[18]	224×224	1	1	32	27	143.67
DensNet_169[19]	224×224	1	1	32	683	14.47
MobileNet_V1_224[20, 21]	224×224	0	1	32	102	4.27
MobileNet_V2_224[20, 21]	224×224	0	1	32	139	6.13

Flow employ an experience-based layout optimization algorithm, which we adopt as the baseline.

6.2 Overall Results—Maintainability

Since the primary aim of the VTensor framework is to reduce the programming burden on operator developers, our focus is on comparing the maintainability of operators rather than the entire framework. Virtual operators, which can be shared among multiple physical operators, require writing only once for tasks such as shape inference, shape validity checks, and so on. Additionally, tasks like layout conversion and kernel dispatch are handled by the VTensor framework. Furthermore, we provide concise APIs for operator developers to implement virtual and physical operators. Hence, we anticipate a substantial reduction in code volume for operator developers when writing operators, leading to improved maintainability and reduced file dispersion.

As LOC is frequently used as an indirect indicator for assessing maintainability, we compare operator maintainability by examining LOC for operators in different frameworks. The LOC for the operator in Fig.10 is a sum of LOC implemented based on the MKL-DNN, Eigen, and cuDNN libraries. The LOC in Figs.1 and 3 solely represents implementations based on the MKL-DNN library.

Fig.10 illustrates the maintainability provided by VTensor, with the x-axis representing LOC and the y-axis representing 16 operators. For each operator, the green bar represents LOC in TensorFlow, while the colored bar indicates LOC in VTensor. The blue section signifies LOC for operation development, while the yellow/gray/red sections represent LOC for library descriptions of MKL-DNN/cuDNN/Eigen, respectively. Compared with TensorFlow, VTensor has achieved a substantial reduction in code size, ranging from 6.31% to 75.37%, with an average reduction of 47.82%.

Furthermore, in TensorFlow, the layout maintenance code is organized in an extremely decentralized manner. On average, developers need to modify six distinct files when creating a new operator. In contrast, with VTensor, developers only need to modify one .cc file (for CPU code) and one .cu file (for CUDA code), by adding the corresponding *Compute* function and handlers in Section 4. Additionally, developers need to write three library description files (Eigen/cuDNN/MKL-DNN), which are shared by all operators. Occasionally, slight modifications are required for these library description files when new guidelines are introduced, but these modifications require minimal effort, and we consider them negligible.

Moreover, for performance-critical operators that support multiple libraries, such as convolution and pooling, VTensor demonstrates even more significant improvements in maintainability. Taking Conv2DOp as an example, in TensorFlow, developers must explicitly write three different library wrappers (Eigen, cuDNN, and MKL-DNN) for layout selection and transformation, necessitating changes in 15 files, with a total LOC of 1 838. In contrast, VTensor dramatically reduces this burden, as developers only need to modify two files, resulting in a reduced LOC of 714.

6.3 Overall Results—Performance

Fig.11(a) illustrates the overall performance of VTensor and TensorFlow on the CPU platform. The

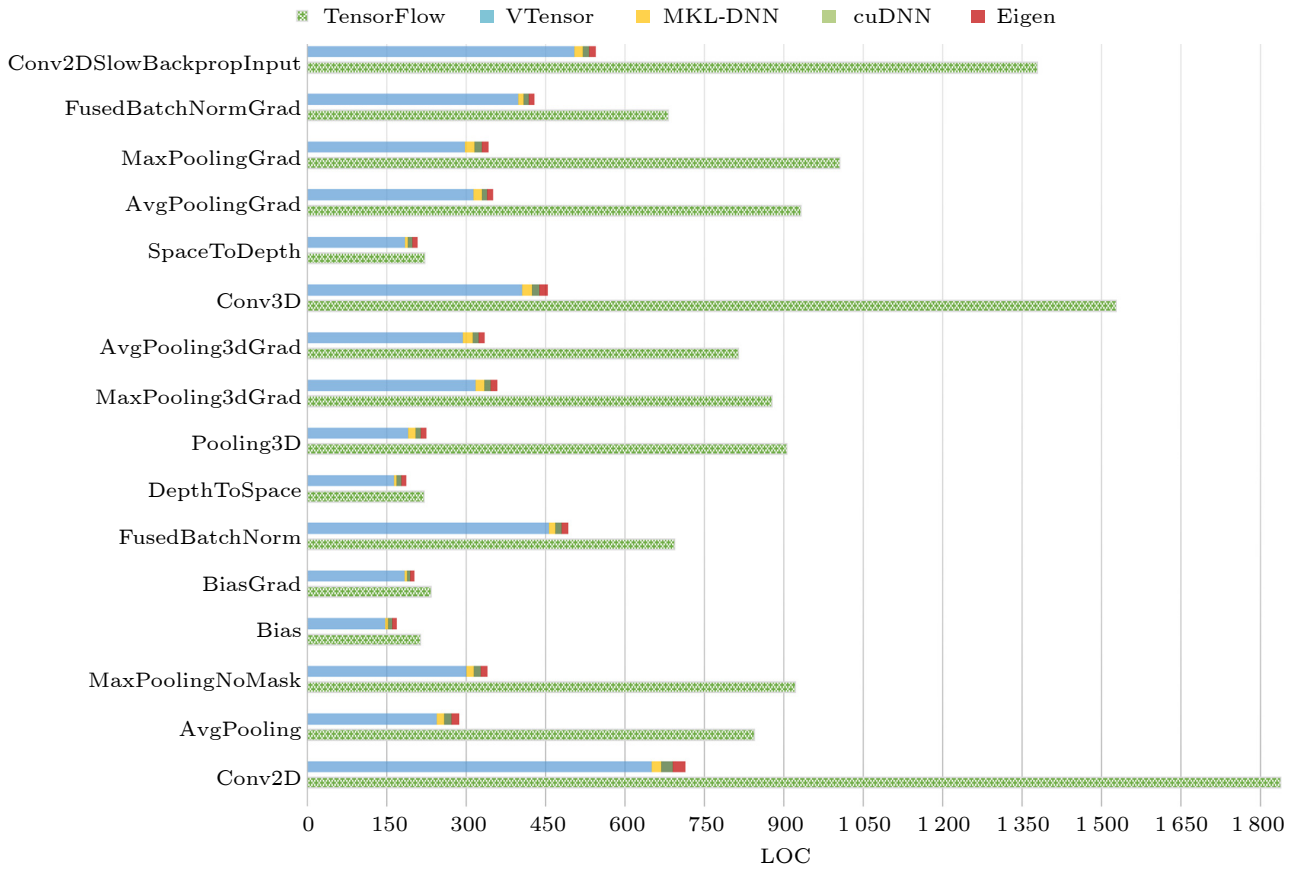

Fig.10. Comparison of LOC when writing an operator using VTensor/TensorFlow framework.

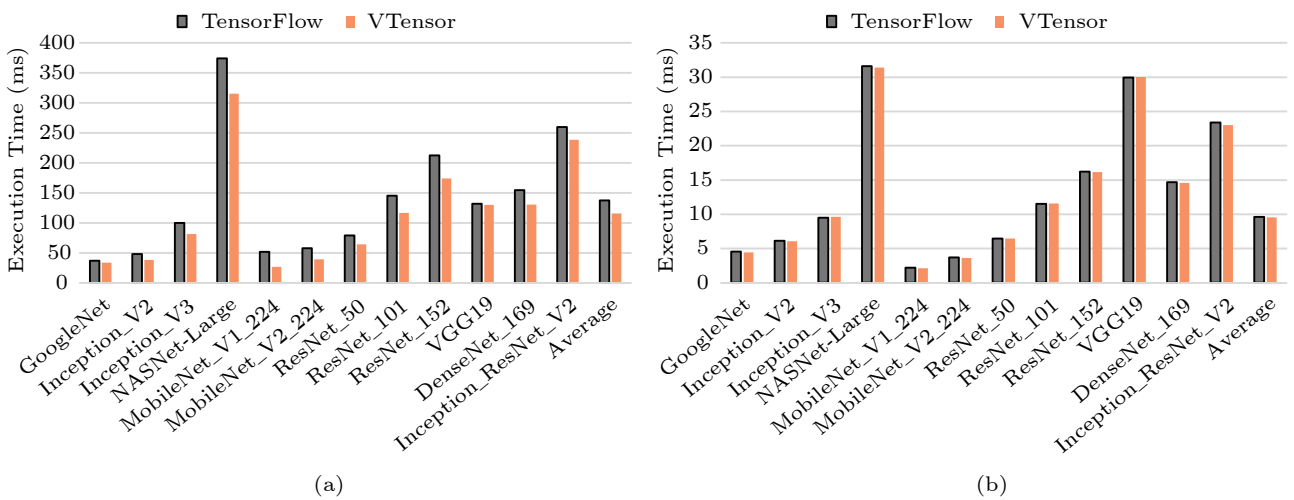

Fig.11. Inference latency of TensorFlow/VTensor. (a) CPU platform. (b) GPU platform.

horizontal axis represents different networks, while the vertical axis represents inference latency. The results reveal that VTensor achieves a notable performance improvement, ranging from 1.37% to 48.27%, with an average improvement of 18.65% compared with TensorFlow.

The CPU results are particularly impressive as they involve the utilization of both MKL-DNN and Eigen in the same network, each employing different layouts. It is worth noting that networks with a substantial number of element-wise nodes, such as ResNet networks, exhibit even greater performance enhancements with VTensor. These networks incorporate numerous data conversion nodes, offering ample opportunities for layout-oriented optimization to enhance global layout transformation. Consequently,

VTensor effectively leverages DLR to determine the layout for each tensor, thereby exposing additional optimization prospects.

In Fig.11(b), VTensor exhibits a slight performance advantage over TensorFlow on the GPU platform, with an improvement of merely 0.31%. The GPU results may appear somewhat underwhelming, given that cuDNN is utilized for all operators. However, cuDNN exclusively adheres to the NCHW layout, limiting VTensor's capacity to identify opportunities for layout optimizations.

6.4 Standard Deviation of Execution Time

Fig.12 illustrates the standard deviation resulting from 1 000 executions of various networks using VTensor and TensorFlow with a batch size of 1. The horizontal axis represents distinct networks, while the vertical axis represents the standard deviation. Fig.12 demonstrates that VTensor's standard deviation on the CPU/GPU platform closely aligns with TensorFlow's.

The significantly higher standard deviation in the network execution time on the CPU platform can be attributed to thread over-subscription. This phenomenon is particularly pronounced when compared with the GPU platform. It is important to note that Eigen and MKL-DNN each employ their own thread pools, lacking a coordination mechanism between them. Consequently, thread over-subscription may occur when multiple operators run in parallel or when one operator executes without an immediate thread sleep.

6.5 Reduced Data Conversions (LOO)

Fig.13 illustrates the performance contribution of LOO on the CPU platform with TensorFlow as the

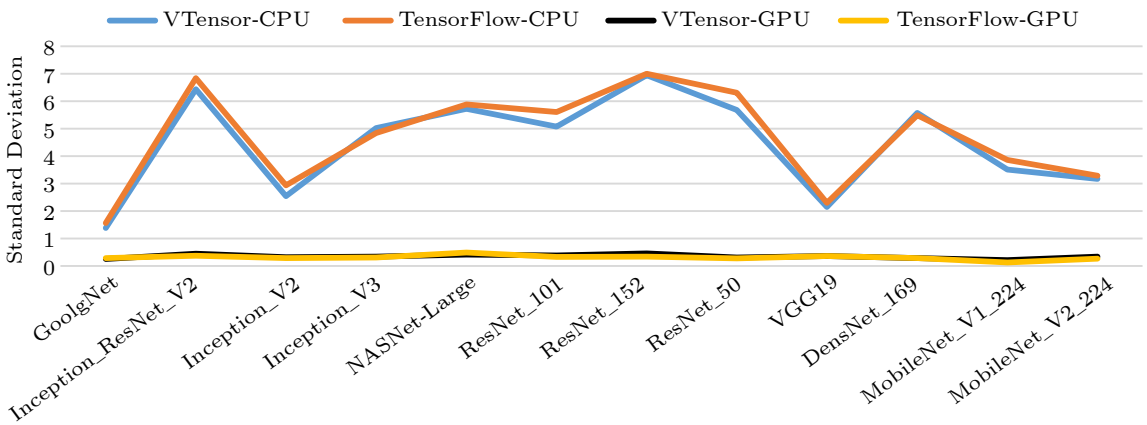

Fig.12. Standard deviation of VTensor/TensorFlow when the batch size is 1.

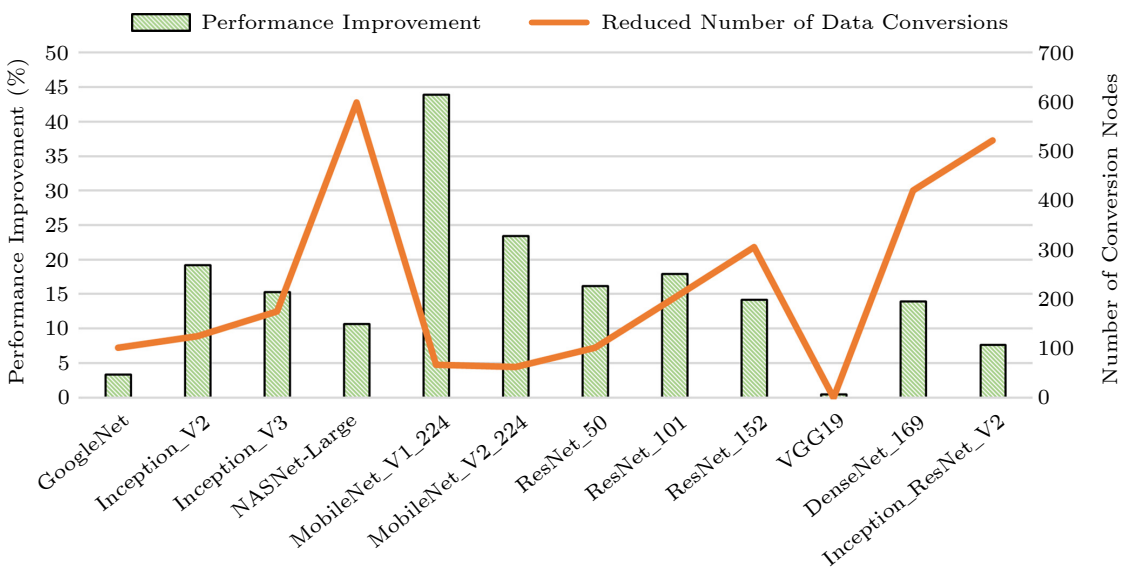

Fig.13. Reduced number of data conversions and performance benefit of LOO.

baseline. The horizontal axis represents different networks, the left vertical axis represents the performance improvement over TensorFlow, and the right vertical axis represents the reduced number of data conversions.

As shown in Fig.13, LOO can enhance performance from 0.46% to 43.92%, with an average improvement of 15.54%. LOO achieves this performance boost by reducing data conversions. For instance, in the case of ResNet_152, LOO reduces the number of data conversions from 317 to 18, resulting in a performance gain of 14.18%. It is important to note that for networks without branches and element-wise nodes, such as VGG19, LOO cannot achieve performance improvement.

The extent of performance improvement through layout optimization relies on the percentage of time saved by eliminating data conversion operations during the entire network's execution. This clarifies why NASNet-Large, which experiences the most significant reduction in data conversion operations, does not exhibit the largest performance improvement.

6.6 Overhead of DLR and EWO

Fig.14 displays the distribution of execution time for each optimization module, normalized to TensorFlow Grappler's total execution time. The yellow, green, and gray bars represent the percentages of the execution time consumed by VTensor framework's DLR, LOO, and other optimizations, respectively, also normalized to TensorFlow Grappler's total time. The red bar signifies the time spent on TensorFlow's layout optimization, while the blue bar accounts for the overall execution time of TensorFlow optimizations, excluding layout optimization.

From Fig.14, it is evident that the proportion of VTensor's layout optimization (the green bar) is smaller than that of TensorFlow's layout optimization (the red bar). Specifically, the execution time of VTensor's LOO is shorter than that of TensorFlow's LOO. This discrepancy arises because the most time-intensive aspect of layout optimization involves creating and inserting data conversion operators. VTensor LOO inserts significantly fewer operators than TensorFlow (as shown in Fig.13), resulting in shorter execution time for VTensor LOO compared with TensorFlow. Additionally, VTensor's Grappler consumes less execution time when compared with TensorFlow.

6.7 Optimization for Different Batch Sizes

Since VTensor cannot identify further optimization opportunities on the GPU platform, we exclusively opt for the CPU platform to evaluate the VTensor network's performance across various batch sizes. As depicted in Fig.15, LOO consistently exhibits performance improvements across diverse batch sizes. We calculate the network's execution time based on the average of 1 000 iterations, with batch size samples being executed in each iteration. The horizontal axis of Fig.15 denotes different batch sizes, while the bars of distinct colors represent various net-

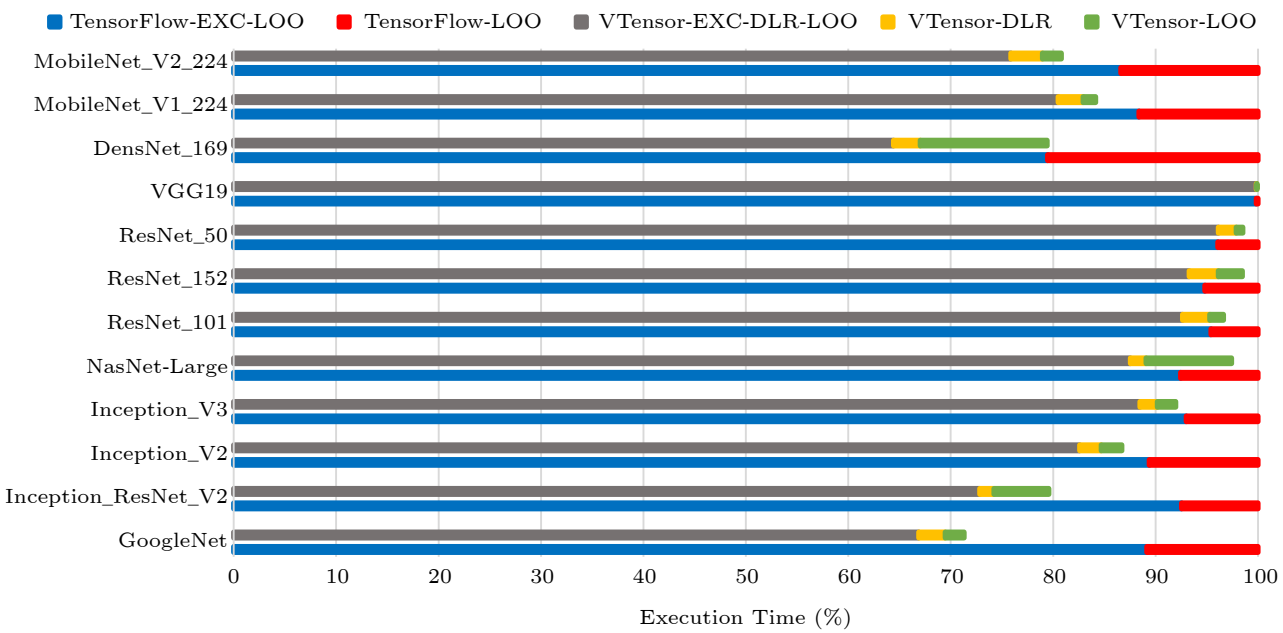

Fig.14. Breakdown of Grappler time under VTensor/TensorFlow.

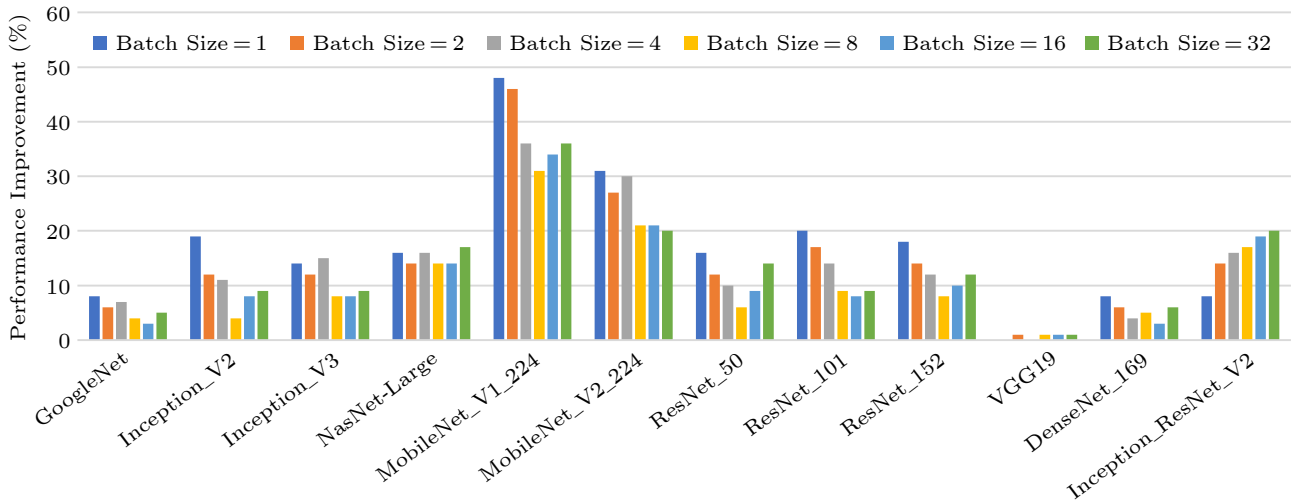

Fig.15. Percentage of performance improvement of different networks under the VTensor framework.

works. The vertical axis represents the percentage of performance enhancement achieved by VTensor compared with TensorFlow.

Fig.15 illustrates that LOO remains effective for different batch sizes. Nevertheless, the LOO performance enhancement percentage for the same network fluctuates across different batch sizes without a discernible pattern. For the sake of clarity, we employ ResNet-50 as an exemplar to elucidate the variations in LOO performance enhancement. This is accomplished by analyzing the breakdown diagram of network execution time across different batch sizes, as seen in Fig.16. Notably, the percentage of the operator's execution time relative to the total network time does not increase with larger batch sizes. The precise reasons for this phenomenon are beyond the scope of this paper. Based on our analysis, we discern that one factor contributing to LOO performance improvement hinges on the proportion of data conversion operations within the entire network.

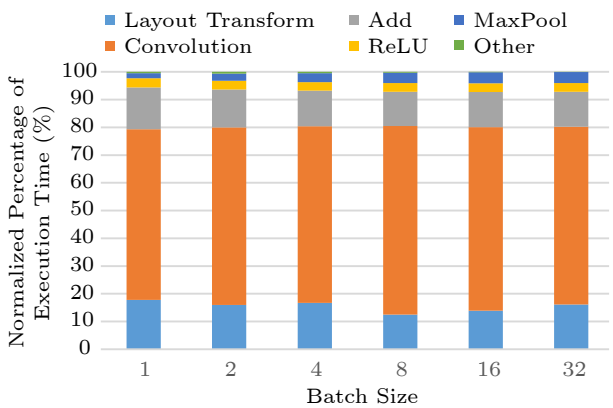

Fig.16. Time breakdown of ResNet-50 under different batch sizes under TensorFlow.

7 Discussion

VTensor's idea of layout decoupling is not limited to a particular AI framework, such as TensorFlow. In this section, we discuss how to apply the ideas of VTensor to Pytorch, and provide an initial assessment of the migration effort required for developers. Furthermore, we examine the integration of VTensor's concept with machine learning compilers like XLA.

7.1 Migration to PyTorch

TensorFlow constructs a data flow graph prior to executing a neural network, while PyTorch employs an imperative execution model, bypassing a separate graph construction phase. These differing execution modes lead to distinct VTensor runtime implementations. Firstly, the DLR module's PE algorithm processes individual nodes rather than the entire graph. Secondly, as PyTorch lacks access to the entire computational graph, layout optimization necessitates either a heuristic or a greedy approach. In the greedy approach, one approach is to maximize the tensor layout's lifecycle. To implement this method, the layout guideline function first includes the use of *produced_layout* as one of the criteria for selecting the layout. Secondly, after the PE algorithm resolves the *required_layout* attribute, it determines whether to insert the data conversion operation immediately. A commonality between PyTorch and TensorFlow is that neither framework has a mechanism for preserving layout information. Consequently, operator developers must remain cognizant of the layout and manually manage layout information.

The implementation of the VTensor programming framework in PyTorch proceeds as follows. Since the library description pertains to the specific library and is framework-independent, it can be directly reused. Other than the library description, we can implement the remaining VTensor framework APIs through the following steps. Initially, we extract the portion of the *at::Tensor* class that is independent of the layout as a virtual tensor class. Subsequently, we extract the layout-related segment of the *at::Tensor* class as a physical tensor class. We then introduce the *produced_layout* attribute to the virtual tensor class and the *required_layout* attribute to the physical tensor class. Finally, we divide each operation into a computation and a set of invokers (with one invoker per library). We employ the PTensor API and the *libTagger* API to encapsulate data/parameter preparation, library function calls, and other operations into the invoker. For libraries already supported by the VTensor framework, the invoker/library description can be reused. Ultimately, we register each invoker using the *libraryInvoker* API in VTensor.

7.2 Programming Efforts

It is required approximately 10 000 LOC to implement the VTensor idea within the TensorFlow framework. This includes 4 300 LOC for the revision of 16 operators using VTensor framework's APIs, around 3 700 LOC for developing the VTensor runtime, and roughly 600 LOC for crafting the VTensor framework's APIs.

Here, we conduct an analysis to determine which modules necessitate adjustments when transitioning the VTensor framework, originally built on TensorFlow, to alternative frameworks. If the current framework supports the acquisition of the entire data flow graph, there is no need for modifications to the ILP solver within the LOO module. However, due to disparities between TensorFlow and PyTorch in the graph data structure and layout transfer mechanisms, we must rewrite the DLR module and the layout optimizer. For frameworks that do not require the construction of computational graphs, a complete rewrite of the VTensor runtime becomes imperative. Additionally, the API of the VTensor framework must be reworked, as runtime and tensor data structures differ across frameworks. Concerning operators, framework-specific components encompass constructors, class definitions, and parameters tied to the framework. For instance, the *OpKernelContext* class is utilized by the TensorFlow framework to record the execution context of the current operator. The remaining operators can be reused directly. Moreover, the library description serves solely to document information about the library and its functions, making it suitable for direct reuse.

7.3 Interaction with XLA

The aim of XLA is to combine numerous small operators and automatically generate the fused code. When dealing with large operators, like convolution, XLA still invokes the kernel code that is implemented based on the library. However, for operators based on libraries, the VTensor framework can still offer a layout-agnostic programming diagram to assist operator developers. Consequently, VTensor remains independent of compilers such as XLA.

8 Related Work

Layout Optimization. A substantial body of research has addressed the significance of layout tuning and selection[8, 9, 24–30]. Specifically, Kim *et al.*[24] analyzed the performance of five AI frameworks with different convolution algorithms and found that layout is a performance-critical factor. Li *et al.*[8] investigated the memory efficiency of various convolutional neural network layers and unveiled performance implications arising from both data layout and memory access patterns. Anderson and Gregg[9] abstracted the layout and primitive selection problem into a PBQP problem from a graph-level perspective. Wen *et al.*[25, 26] introduced the ILP technology to address the limited memory resource selection for the optimal combination of primitives and layouts. Zhang *et al.*[27] proposed a decision tree-based approach to select suitable layouts for a network in DSP. Zheng *et al.*[28] observed that matrix multiplication constitutes the performance bottleneck for LSTM RNN on GPU and introduced EcoRNN for automatic library selection and layout. TASO[29] considers layout transformations in conjunction with graph substitutions. NeoCPU[30] alters the layout of all convolution operations to NCHW[x] and globally tunes the parameter x when providing hardware details. These approaches still rely on traditional layout-aware programming interfaces, which place a significant maintenance and conversion burden on developers. Furthermore, the afore-

mentioned layout optimization methods are only applicable in scenarios where the layout can be determined at compile time. In contrast, VTensor dynamically selects layouts at runtime thanks to the DLR. Ould-Ahmed-Vall *et al.*[11] divided the data flow graph into multiple subgraphs based on whether the operator is implemented using MKL-DNN. Data conversion operations within each subgraph are performed by the operator. Although this heuristic method can be executed at runtime, the layout choice represents only a local optimal solution. In contrast to their approach, VTensor acquires layout information for each node in the data flow diagram through the DLR at runtime and then selects the layout. Consequently, our approach yields superior performance improvements.

Tensor Processing. There has been extensive research on tensor processing, spanning from domain-specific languages to optimized compilers[1, 31–36]. Most existing work in this area requires programmers to be aware of tensor layouts. Ragan-Kelley *et al.*[31] introduced the concept of separating computation from scheduling, enabling the efficient generation of image processing pipelines. Chen *et al.*[32, 33] employed a domain-specific language based on tensor expressions, along with a comprehensive compilation stack, to facilitate efficient tensor operator generation on heterogeneous architectures. FlexTensor[34] and Ansor[35] focus on automatic schedule space exploration. TACO[36] offers an alternative approach through the generation of dense/sparse kernels from tensor algebra expressions. Additionally, researchers have proposed a series of methods for dealing with sparse tensors. Given the diversity of sparsity in sparse tensors, researchers[37–39] suggested using machine learning to analyze non-zero layouts and select the optimal storage format. Nisa *et al.*[40] proposed a mixed-mode storage format for sparse tensors of arbitrary dimensions, enabling efficient memory access across different dimensions. Dong *et al.*[41] introduced a new data layout to optimize DNNs with input sparsity. To enhance the performance of tensor contraction operators and utilize hardware resources efficiently, various techniques, such as tiling and data reorganization, have been proposed to improve data reuse[42–46] and manage data movement[47]. The approaches mentioned above primarily focus on exploiting hardware-specific properties to enhance tensor processing performance, whereas VTensor takes an orthogonal approach to these methods.

9 Conclusions

In this paper, we observed that developers employ mathematical semantics for layouts at the application layer, while adopting physical semantics for layouts at the neural network library layer. Based on these observations, we proposed a novel programming abstraction and a layout resolution mechanism. These innovations aim to bridge the gap between the application layer's arbitrary layout utilization and the layout conventions of high-performance libraries. Notably, as layout resolution occurs at runtime in the VTensor framework, we uncovered two new opportunities for layout optimization: the elimination of redundant layout transformation operations in the computational graph, and the enhancement of layout selection for element-wise operators with broadcast semantics. Our experimental results, driven by typical networks, demonstrated significant benefits. In comparison to TensorFlow, using the VTensor framework reduces the LOC required to write operators by 47.8%. Furthermore, the layout optimization techniques presented in this paper enhance the performance of the entire network by 18.6%. Thus, VTensor exhibits great potential for utilization in operator development on emerging accelerators or hardware, such as Cambricon's MLU and Huawei's Ascend, effectively boosting network performance and improving operator development efficiency.

However, it is essential to note that VTensor is currently optimized for the layout decoupling of dense tensors. The abstract representation of sparse tensors remains an open challenge. In the future, our plan is to integrate VTensor's concepts into the compiler's intermediate representation to support the abstract representation of sparse tensors.

Conflict of Interest The authors declare that they have no conflict of interest.

References

[1] Abadi M, Barham P, Chen J M, Chen Z F, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M, Kudlur M, Levenberg J, Monga R, Moore S, Murray D G, Steiner B, Tucker P, Vasudevan V, Warden P, Wicke M, Yu Y, Zheng X Q. TensorFlow: A system for large-scale machine learning. In *Proc. the 12th USENIX Symposium on Operating Systems Design and Implementation*, Nov. 2016, pp.265–283. DOI: 10.5555/3026877.3026899.

[2] Chen T Q, Li M, Li Y T, Lin M, Wang N Y, Wang M J, Xiao T J, Xu B, Zhang C Y, Zhang Z. MXNet: A flexible

and efficient machine learning library for heterogeneous distributed systems. arXiv: 1512.01274, 2015.https://doi.org/10.48550/arXiv.1512.01274, Sept. 2023.

[3] Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin Z M, Gimelshein N, Antiga L, Desmaison A, Köpf A, Yang E, DeVito Z, Raison M, Tejani A, Chilamkurthy S, Steiner B, Fang L, Bai J J, Chintala S. PyTorch: An imperative style, high-performance deep learning library. In *Proc. the 33rd International Conference on Neural Information Processing Systems*, Dec. 2019, Article No. 721. DOI: 10.5555/3454287.3455008.

[4] Jia Y Q, Shelhamer E, Donahue J, Karayev S, Long J, Girshick R, Guadarrama S, Darrell T. Caffe: Convolutional architecture for fast feature embedding. In *Proc. the 22nd ACM International Conference on Multimedia*, Nov. 2014, pp.675–678. DOI: 10.1145/2647868.2654889.

[5] Barham P, Isard M. Machine learning systems are stuck in a rut. In *Proc. the 2019 Workshop on Hot Topics in Operating Systems*, May 2019, pp.177–183. DOI: 10.1145/3317550.3321441.

[6] Langtangen H P. Numerical computing in Python. In *Python Scripting for Computational Science*, Langtangen H P (ed.), Springer, 2004, pp.131–188. DOI: 10.1007/978–3-662–05450-5_4.

[7] Goldsborough P. A tour of TensorFlow. arXiv: 1610.01178, 2016. https://doi.org/10.48550/arXiv.1610.01178, September 2023.

[8] Li C, Yang Y, Feng M, Chakradhar S, Zhou H Y. Optimizing memory efficiency for deep convolutional neural networks on GPUs. In *Proc. the 2016 International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2016, pp.633–644. DOI: 10.1109/SC.2016.53.

[9] Anderson A, Gregg D. Optimal DNN primitive selection with partitioned Boolean quadratic programming. In *Proc. the 2018 International Symposium on Code Generation and Optimization*, Feb. 2018, pp.340–351. DOI: 10.1145/3168805.

[10] Chetlur S, Woolley C, Vandermersch P, Cohen J, Tran J, Catanzaro B, Shelhamer E. cuDNN: Efficient primitives for deep learning. arXiv: 1410.0759, 2014. https://doi.org/10.48550/arXiv.1410.0759, September 2023.

[11] Ould-Ahmed-Vall E, Abuzaina M, Amin M, Bobba J, Dubtsov R, Fomenko E, Gangadhar M, Hasabnis N, Huang J, Karkada D, Kim J Y, Makineni S, Mishura D, Raman K, Ramesh A, Rane V, Riera M, Sergeev D, Sripathi V, Subramanian B, Tokas L, Valles A. Accelerating TensorFlow on modern Intel architectures. In *Proc. the 1st International Workshop on Architectures for Intelligent Machines*, Sept. 2017.

[12] van der Walt S, Colbert S C, Varoquaux G. The NumPy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, 2011, 13(2): 22–30. DOI: 10.1109/MCSE.2011.37.

[13] Ioffe S, Szegedy C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proc. the 32nd International Conference on Machine Learning*, Jul. 2015, pp.448–456. DOI: 10.5555/3045118.3045167.

[14] Szegedy C, Ioffe S, Vanhoucke V, Alemi A A. Inception-v4, inception-ResNet and the impact of residual connections on learning. In *Proc. the 31st AAAI Conference on Artificial Intelligence*, Feb. 2017, pp.4278–4284. DOI: 10.5555/3298023.3298188.

[15] Szegedy C, Liu W, Jia Y Q, Sermanet P, Reed S, Anguelov D, Erhan D, Vanhoucke V, Rabinovich A. Going deeper with convolutions. In *Proc. the 2015 IEEE Conference on Computer Vision and Pattern Recognition*, Jun. 2015. DOI: 10.1109/CVPR.2015.7298594.

[16] Szegedy C, Vanhoucke V, Ioffe S, Shlens J, Wojna Z. Rethinking the inception architecture for computer vision. In *Proc. the 2016 IEEE Conference on Computer Vision and Pattern Recognition*, Jun. 2016, pp.2818–2826. DOI: 10.1109/CVPR.2016.308.

[17] He K M, Zhang X Y, Ren S Q, Sun J. Deep residual learning for image recognition. In *Proc. the 2016 IEEE Conference on Computer Vision and Pattern Recognition*, Jun. 2016, pp.770–778. DOI: 10.1109/CVPR.2016.90.

[18] Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition. arXiv: 1409.1556, 2014. http://arxiv.org/abs/1409.1556, September 2023.

[19] Huang G, Liu Z, Van Der Maaten L, Weinberger K Q. Densely connected convolutional networks. In *Proc. the 2017 IEEE Conference on Computer Vision and Pattern Recognition*, Jul. 2017, pp.2261–2269. DOI: 10.1109/CVPR.2017.243.

[20] Howard A G, Zhu M L, Chen B, Kalenichenko D, Wang W J, Weyand T, Andreetto M, Adam H. MobileNets: Efficient convolutional neural networks for mobile vision applications. arXiv: 1704.04861, 2017. https://doi.org/10.48550/arXiv.1704.04861, Sept. 2023.

[21] Sandler M, Howard A, Zhu M L, Zhmoginov A, Chen L C. MobileNetV2: Inverted residuals and linear bottlenecks. In *Proc. the 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, Jun. 2018, pp.4510–4520. DOI: 10.1109/CVPR.2018.00474.

[22] Zoph B, Vasudevan V, Shlens J, Le Q V. Learning transferable architectures for scalable image recognition. In *Proc. the 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, Jun. 2018, pp.8697–8710. DOI: 10.1109/CVPR.2018.00907.

[23] Wang Y E, Wu C J, Wang X D, Hazelwood K, Brooks D. Exploiting parallelism opportunities with deep learning frameworks. *ACM Trans. Architecture and Code Optimization*, 2021, 18(1): Article No. 9. DOI: 10.1145/3431388.

[24] Kim H, Nam H, Jung W, Lee J. Performance analysis of CNN frameworks for GPUs. In *Proc. the 2017 IEEE International Symposium on Performance Analysis of Sys-*

tems and Software (*ISPASS*), Apr. 2017, pp.55–64. DOI: 10.1109/ISPASS.2017.7975270.

[25] Wen Y, Anderson A, Radu V, O'Boyle M F P, Gregg D. TASO: Time and space optimization for memory-constrained DNN inference. In *Proc. the 32nd International Symposium on Computer Architecture and High Performance Computing* (*SBAC-PAD*), Sept. 2020, pp.199–208. DOI: 10.1109/SBAC-PAD49847.2020.00036.

[26] Wen Y, Anderson A, Radu V, O'Boyle M F P, Gregg D. POSTER: Space and time optimal DNN primitive selection with integer linear programming. In *Proc. the 28th International Conference on Parallel Architectures and Compilation Techniques* (*PACT*), Sept. 2019, pp.489–490. DOI: 10.1109/PACT.2019.00059.

[27] Zhang X Y, Xiao J M, Zhang X B, Hu Z Z, Zhu H R, Tian Z B, Tan G M. Tensor layout optimization of convolution for inference on digital signal processor. In *Proc. the 2019 IEEE International Conference on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking* (*ISPA/BDCloud/SocialCom/SustainCom*), Dec. 2019, pp.184–193. DOI: 10.1109/ISPA-BDCloud-SustainCom-SocialCom48970.2019.00036.

[28] Zheng B J, Vijaykumar N, Pekhimenko G. Echo: Compiler-based GPU memory footprint reduction for LSTM RNN training. In *Proc. the 47th Annual International Symposium on Computer Architecture* (*ISCA*), May 30–Jun. 3, 2020, pp.1089–1102. DOI: 10.1109/ISCA45697.2020.00092.

[29] Jia Z H, Padon O, Thomas J, Warszawski T, Zaharia M, Aiken A. TASO: Optimizing deep learning computation with automatic generation of graph substitutions. In *Proc. the 27th ACM Symposium on Operating Systems Principles*, Oct. 2019, pp.47–62. DOI: 10.1145/3341301.3359630.

[30] Liu Y Z, Wang Y, Yu R F, Li M, Sharma V, Wang Y D. Optimizing CNN model inference on CPUs. In *Proc. the 2019 USENIX Conference on Usenix Annual Technical Conference*, July 2019, pp.1025–1040. DOI: 10.5555/3358807.3358895.

[31] Ragan-Kelley J, Barnes C, Adams A, Paris S, Durand F, Amarasinghe S. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proc. the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 2013, pp.519–530. DOI: 10.1145/2491956.2462176.

[32] Chen T Q, Moreau T, Jiang Z H, Zheng L M, Yan E, Cowan M, Shen H C, Wang L Y, Hu Y W, Ceze L, Guestrin C, Krishnamurthy A. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proc. the 13th USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2018, pp.579–594. DOI: 10.5555/3291168.3291211.

[33] Chen T Q, Zheng L M, Yan E, Jiang Z H, Moreau T, Ceze L, Guestrin C, Krishnamurthy A. Learning to optimize tensor programs. In *Proc. the 32nd International Conference on Neural Information Processing Systems*, Dec. 2018, pp.3393–3404. DOI: 10.5555/3327144.3327258.

[34] Zheng S Z, Liang Y, Wang S, Chen R Z, Sheng K W. FlexTensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proc. the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2020, pp.859–873. DOI: 10.1145/3373376.3378508.

[35] Zheng L M, Jia C F, Sun M M, Wu Z, Yu C H, Haj-Ali A, Wang Y D, Yang J, Zhuo D Y, Sen K, Gonzalez J E, Stoica I. Ansor: Generating high-performance tensor programs for deep learning. In *Proc. the 14th USENIX Symposium on Operating Systems Design and Implementation* (*OSDI 20*), Nov. 2020, Article No. 49. DOI: 10.5555/3488766.3488815.

[36] Kjolstad F, Kamil S, Chou S, Lugato D, Amarasinghe S. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 2017, 1(OOPSLA): Article No. 77. DOI: 10.1145/3133901.

[37] You Y, Demmel J. Runtime data layout scheduling for machine learning dataset. In *Proc. the 46th International Conference on Parallel Processing* (*ICPP*), Aug. 2017, pp.452–461. DOI: 10.1109/ICPP.2017.54.

[38] Li J J, Tan G M, Chen M Y, Sun N H. SMAT: An input adaptive auto-tuner for sparse matrix-vector multiplication. *ACM SIGPLAN Notices*, 2013, 48(6): 117–126. DOI: 10.1145/2499370.2462181.

[39] Zhao Y, Li J J, Liao C H, Shen X P. Bridging the gap between deep learning and sparse matrix format selection. In *Proc. the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb. 2018, pp.94–108. DOI: 10.1145/3178487.3178495.

[40] Nisa I, Li J J, Sukumaran-Rajam A, Rawat P S, Krishnamoorthy S, Sadayappan P. An efficient mixed-mode representation of sparse tensors. In *Proc. the 2019 International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2019, Article No. 49. DOI: 10.1145/3295500.3356216.

[41] Dong X, Liu L, Zhao P, Li G L, Li J S, Wang X Y, Feng X B. Acorns: A framework for accelerating deep neural networks with input sparsity. In *Proc. the 28th International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2019, pp.178–191. DOI: 10.1109/PACT.2019.00022.

[42] Hildebrand M, Khan J, Trika S, Lowe-Power J, Akella V. AutoTM: Automatic tensor movement in heterogeneous memory systems using integer linear programming. In *Proc. the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2020, pp.875–890. DOI: 10.1145/3373376.3378465.

[43] Hong C W, Sukumaran-Rajam A, Nisa I, Singh K, Sa-

dayappan P. Adaptive sparse tiling for sparse matrix multiplication. In *Proc. the 24th Symposium on Principles and Practice of Parallel Programming*, Feb. 2019, pp.300–314. DOI: 10.1145/3293883.3295712.

[44] Jiang P, Hong C W, Agrawal G. A novel data transformation and execution strategy for accelerating sparse matrix multiplication on GPUs. In *Proc. the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb. 2020, pp.376–388. DOI: 10.1145/3332466.3374546.

[45] Li R, Sukumaran-Rajam A, Veras R, Low T M, Rastello F, Rountev A, Sadayappan P. Analytical cache modeling and tilesize optimization for tensor contractions. In *Proc. the 2019 International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2019, Article No. 74. DOI: 10.1145/3295500.3356218.

[46] Peng X, Shi X H, Dai H L, Jin H, Ma W L, Xiong Q, Yang F, Qian X H. Capuchin: Tensor-based GPU memory management for deep learning. In *Proc. the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2020, pp.891–905. DOI: 10.1145/3373376.3378505.

[47] Kim J, Sukumaran-Rajam A, Thumma V, Krishnamoorthy S, Panyala A, Pouchet L N, Rountev A, Sadayappan P. A code generator for high-performance tensor contractions on GPUs. In *Proc. the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, Feb. 2019, pp.85–95. DOI: 10.1109/CGO.2019.8661182.

Feng Yu is a Ph.D. candidate in the State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing. He received his Bachelor's degree in computer science from Henan University, Kaifeng, in 2016. His research interests include artificial intelligence compiler.

Jia-Cheng Zhao received his B.S. degree in computer science from Tianjin University, Tianjin, in 2012, and his Ph.D. degree in computer science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences, Beijing, in 2017. He is currently an associate professor at ICT, Chinese Academy of Sciences (CAS), Beijing. His research interests include compiler optimizations, programming languages, and programming environments.

Hui-Min Cui received her B.S. and M.S. degrees in computer science from Tsinghua University, Beijing, in 2001 and 2004, respectively, and her Ph.D. degree in computer science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, in 2012. She is currently a professor at ICT, CAS, Beijing. Her research interests include compiler optimizations, programming languages, and programming environments.

Xiao-Bing Feng received his B.S. degree in computer science from Tianjin University, Tianjin, in 1992, his M.S. degree in computer science from Peking University, Beijing, in 1996, and his Ph.D. degree in computer science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, in 1999. He is currently a professor at ICT, CAS and the University of Chinese Academy of Sciences, Beijing. His research interests include compiler optimizations and binary translation.

Jingling Xue received his B.Eng. and M.Eng. degrees in computer science and engineering from Tsinghua University, Beijing, in 1984 and 1987, respectively, and his Ph.D. degree in computer science and engineering from Edinburgh University, Edinburgh, in 1992. He is currently a scientia professor with the School of Computer Science and Engineering at the University of New South Wales, Sydney. His research interests include programming languages, compiler technology, and program analysis.