NeoFlow: A Flexible Framework for Enabling Efficient Compilation for High Performance DNN Training

Size Zheng, *Student Member, IEEE*, Renze Chen, Yicheng Jin, Anjiang Wei, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang, *Senior Member, IEEE*

Abstract—Deep neural networks (DNNs) are increasingly deployed in various image recognition and natural language processing applications. The continuous demand for accuracy and high performance has led to innovations in DNN design and a proliferation of new operators. However, existing DNN training frameworks such as PyTorch and TensorFlow only support a limited range of operators and rely on hand-optimized libraries to provide efficient implementations for these operators. To evaluate novel neural networks with new operators, the programmers have to either replace the holistic new operators with existing operators or provide low-level implementations manually. Therefore, a critical requirement for DNN training frameworks is to provide high-performance implementations for the neural networks containing new operators automatically in the absence of efficient library support. In this article, we introduce NeoFlow, which is a flexible framework for enabling efficient compilation for high-performance DNN training. NeoFlow allows the programmers to directly write customized expressions as new operators to be mapped to graph representation and low-level implementations automatically, providing both high programming productivity and high performance. First, NeoFlow provides expression-based automatic differentiation to support customized model definitions with new operators. Then, NeoFlow proposes an efficient compilation system that partitions the neural network graph into subgraphs, explores optimized schedules, and generates high-performance libraries for subgraphs automatically. Finally, NeoFlow develops an efficient runtime system to combine the compilation and training as a whole by overlapping their execution. In the experiments, we examine the numerical accuracy and performance of NeoFlow. The results show that NeoFlow can achieve similar or even better performance at the operator and whole graph level for DNNs compared to deep learning frameworks. Especially, for novel networks training, the geometric mean speedups of NeoFlow to PyTorch, TensorFlow, and CuDNN are 3.16X, 2.43X, and 1.92X, respectively.

Index Terms—Deep learning, training, code generation, compiler optimization, automatic differentiation

1 Introduction

Deep neural networks (DNNs) have gained great breakthroughs in various domains including image processing [1], [2], [3] and natural language processing [4], [5], [6]. The excellent performance of these DNNs often comes with a long training time. Commonly, training a model may take hours or even days. Today, the best practice of training DNN models is to use frameworks such as PyTorch [7], TensorFlow [8], and MxNet [9]. The users implement their DNN models in these frameworks by defining the DNN graph, In general, a DNN graph is composed of a series of tensors and operators, where a tensor is regarded as a multidimensional array and an operator is a function applied to the tensors (e.g., convolution). After that, the deep learning frameworks rely on high-performance hand-optimized libraries to accelerate the execution of DNN graphs. For example, on Nvidia GPUs, PyTorch and TensorFlow leverage CuDNN library [10] to accelerate DNN operators such as convolution and batch normalization by mapping them to the corresponding APIs of CuDNN.

The dramatic growth in use has bolstered new DNN models such as CapsuleNet [11], [12], [13], MI-LSTM [14], and SC-RNN [15]. However, existing DNN training frameworks heavily rely on hand-optimized libraries and thus lack support for these emerging DNN models with new operators. For example, there is currently no implementation of capsule convolution [11], [12], [13] in CuDNN, and as a result, deep learning frameworks can't provide an efficient implementation for this operator. There are two possible approaches to deal with this problem. One approach is to use existing operators to assemble a computation graph that is functionally equivalent to the new operator. For example, the users can use several 2D convolutions to substitute a capsule convolution by assembling the results of these convolutions. However, this approach breaks a holistic operator into small operators and introduces additional

- *Size Zheng, Renze Chen, Yicheng Jin, Anjiang Wei, Bingyang Wu, and Yun Liang are with the School of EECS, Peking University, Beijing 100871, China. E-mail: {zhengsz, crz, yicheng.jin, weianjiang, bingyangwu, ericlyun}@pku.edu.cn.*
- *Xiuhong Li is with SenseTime Research & Shanghai AI Lab, Beijing 100080, China. E-mail: lixiuhong@sensetime.com.*
- *Shengen Yan is with SenseTime Research, Beijing 100080, China. E-mail: yanshengen@sensetime.com.*

Manuscript received 19 Aug. 2021; revised 10 Nov. 2021; accepted 23 Dec. 2021. Date of publication 28 Dec. 2021; date of current version 23 May 2022.
This work was supported in part by the National Natural Science Foundation of China (NSFC) under Grant U21B2017, and in part by the Shanghai Committee of Science and Technology, China under Grant 20DZ1100800. (Corresponding author: Yun Liang.)
Recommended for acceptance by A. J. Peña, M. Si and J. Zhai.
Digital Object Identifier no. 10.1109/TPDS.2021.3138862

tensor transformation overhead such as slicing and reshaping, resulting in a complex implementation and low performance. More importantly, this manual process requires significant effort and scales poorly as the number of new operators increases. Another approach is to implement the low-level kernels (e.g., CUDA kernels) manually for the new operator and register them into the deep learning frameworks. But the low-level kernel programming and optimization is time-consuming and error-prone. A less optimized implementation often results in low performance. Besides, supporting new operators in training requires extensive modification to the deep learning framework as the users have to infer the gradients of new operators and implement them manually.

Recently, deep learning compilers have gained more and more attention, which can generate an efficient implementation for new DNN operators without hand-optimized library support. The users just need to write a short expression in domain-specific languages (DSLs), and these compilers will generate low-level code for the users. For example, Halide [16] and TVM [17] require the users to write *compute* language and *schedule* language to generate low-level code for operators. PlaidML [18] and Tensor Comprehensions [19] take high-level expressions as inputs and generate high-performance code via polyhedral model. Some frameworks [20], [21], [22] have integrated these compilers for DNNs. PyTorch can redirect inference tasks to TVM [20]; JAX [23] supports fine-grained autodiff and code generation with the support of XLA [24]; nGraph [22] leverages PlaidML to generate code for training, but the users have to use existing primitives to describe their DNNs in JAX and nGraph; Relay [21] only has very rudimentary support for training and is still not usable for real training tasks.

Although the integration of deep learning compilers into deep learning frameworks is appealing, several challenges remain. First, to enable new operators for training, the users have to manually implement the computation of gradients for the new operators. Second, the unoptimized implementation of new operators often results in low performance and sophisticated optimizations that elaborately optimize the new operators globally at the DNN graph level are necessary. Third, the compilation and code generation could be prohibitively long, which may delay the whole process of training. Therefore, the runtime system needs to balance the compilation overhead and execution gain for training.

In this paper, we introduce NeoFlow, a flexible framework for enabling efficient compilation for high-performance DNN training. NeoFlow provides both high programming productivity and high performance through innovations in DNN model representation and optimization, compilation and code generation, and efficient runtime system. For DNN model representation, NeoFlow proposes to treat expressions written by the users as operators and constructs DNN graphs by directly linking these operators with tensors. NeoFlow applies automatic model transformations to optimize the DNN model and develops an expression-based automatic differentiation flow. For DNN model compilation, NeoFlow partitions the DNN graph into subgraphs and explores optimization schedule spaces for each subgraph. The optimized schedules are used in the code generation process to generate a high-performance library for each subgraph.

For DNN graph execution, NeoFlow proposes a runtime system that seamlessly integrates the compilation and training. The runtime can use the optimized library generated by the compilation in either static or dynamic mode. In static mode, the subgraphs are compiled ahead of time and the libraries are invoked at runtime. This mode is normally useful when the DNN models are stabilized. In dynamic mode, the subgraphs are compiled dynamically to generate libraries. The code generation at the subgraph level helps to mitigate the compilation overhead and we carefully design the runtime system to allow the subgraphs to be compiled incrementally so that the optimized libraries can be used in the subsequent iterations of mini-batch training. The DNN model execution at early iterations also provides feedback to guide the selection of optimization schedules for the later iterations. This mode is normally useful when the DNN models are evolving and the users desire interactive feedback during training.

In summary, this paper makes the following contributions:

- We propose NeoFlow, a flexible framework for enabling efficient compilation for high-performance DNN training. NeoFlow can support DNN models containing new operators automatically and efficiently.
- We design an expression-based automatic differentiation for DNNs with novel operators. We also propose automatic model transformation techniques to optimize DNN models.
- We develop a DNN compilation and runtime system that seamlessly integrate incremental subgraph compilation and code generation for high-performance training.

Experiments using various novel DNN models including CapsuleNet [12], MI-LSTM [14], and SC-RNN [15] show that NeoFlow can achieve better performance. The geometric mean speedups to PyTorch, TensorFlow, and CuDNN are 3.16X, 2.43X, and 1.92X, respectively.

2 Background and Motivation

In this section, we first introduce the background of deep learning frameworks and compilers. Then, we present the motivation of NeoFlow.

2.1 Deep Learning Frameworks

Deep learning frameworks [7], [8], [9] are complex systems that perform diverse functions including tensor management, automatic differentiation, library invocation, etc. To use these frameworks, the users construct a DNN graph by writing a high-level language (e.g., in Python) where the graph nodes are operators and graph edges are tensors. The users define a forward part of the DNN graph, and the frameworks will automatically figure out the gradient calculation for each operator, forming a backward graph for training. In Fig. 1, we show a simple example of DNN graph for training. It is composed of three parts. The forward and loss parts are provided by users, while the backward part is automatically generated by the framework. The *func1* to

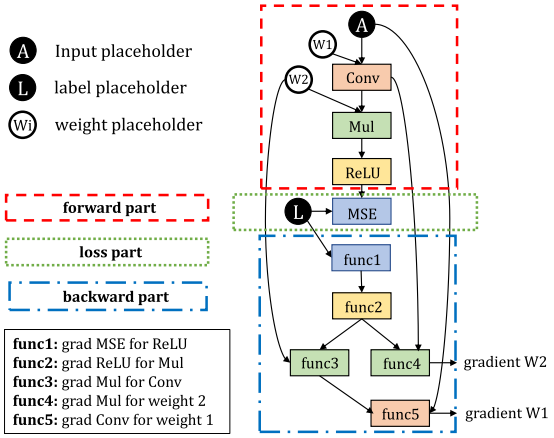


Fig. 1. An example DNN graph with forward part, loss part, and backward part.

func5 are also operators, which are automatically appended to the graph.

To accelerate DNN training, deep learning frameworks map the training graphs to hand-optimized libraries. The libraries include CuDNN [10] and CuBLAS [25] for Nvidia GPUs, MKL [26] and OneDNN [27] for Intel CPUs, etc. Based on these libraries, frameworks such as PyTorch [7] can support thousands of operators.

2.2 Deep Learning Compilers

Recently, important practical advances have occurred in deep learning compilers [16], [17], [18], [19], which take high-level expressions as inputs and generate low-level code such as C and CUDA automatically. The input expression describes the mathematical logic of a calculation. To generate high-performance code, compilers such as Tensor Comprehensions [19] and PlaidML [18] rely on cost models to guide code generation, while Halide [16] and TVM [17] employ writing schedule languages to control sophisticated loop transformations and hardware-specific optimizations, such as loop tiling, unrolling, and vectorization. To find the optimal combination of schedules is hard and requires a long time of exploration in the huge schedule space during compilation. Previous work proposes to use machine learning techniques to guide the schedule search. Typical frameworks include Halide autoscheduler [28], AutoTVM [29], Chameleon [30], FlexTensor [31], and Ansor [32]. But they all focus on inference scenarios and rely on other graph-level frameworks such as Relay [21] to handle the DNN model representation and optimization.

To compile a whole DNN model, existing compilers employ two levels of representation: graph IR and operator IR. Graph IR is composed of various nodes that represent different operators. For example, Relay [21] uses specific nodes to represent 2D convolution and matrix multiplication. During compilation, these nodes in graph IR are lowered to operator IR which is composed of loop nests. This two-level IR infrastructure makes it hard to support training and code generation within the same framework. First of all, if the compilers employ automatic gradient algorithm that is widely used in current deep learning frameworks such as PyTorch [7] and TensorFlow [8], it is likely to generate unrecognized operators that can't be lowered by the compiler. For

Shape	Batch	In_C	Out_C	Height	Width	Capsules
	1	64	256	28	28	8

	PyTorch	TensorFlow	NeoFlow
Latency	1.529 ms	4.192 ms	0.451 ms
Launch Overhead	0.473 ms	0.717 ms	0.007 ms
Kernel Overhead	0.899 ms	2.532 ms	0.390 ms
Utilization	65.5%	77.9%	98.2%

Fig. 2. Performance of capsule convolution.

instance, the gradient operator of capsule convolution [11], [12], [13] is another new operator that is not supported by any framework or compiler. Second, if the compilers choose to implement automatic differentiation at the operator level, then they can hardly apply graph optimization such as operator fusion because the existing automatic differentiation algorithm based on loop-level IR restricts the operators to be represented by perfect loop nests [33], [34], but fusion tends to produce imperfect loop nests.

2.3 Motivation

Here we use an example of capsule convolution [11], [12], [13] to illustrate the challenges of accelerating the training of novel DNNs with new operators. Capsule convolution's calculation expression is

$$C[b,k,p,q,i,j] = A[b,c,p*2+r,q*2+s,i,k] * B[k,c,r,s,k,j],$$

where A, B, C are tensors. Capsule convolution is a core operator in CapsuleNet [11], [12], [13], which is designed to better model hierarchical relationships. However, there is currently no library support for capsule convolution. To support this operator, one choice is to split it into a series of small 2D convolutions and then produce output tensor by concatenating the partial results of these 2D convolutions.

We present the performance of capsule convolution in PyTorch and TensorFlow on Nvidia Tesla V100 GPU in Fig. 2. The problem size, kernel launch overhead, and device utilization are presented. The implementation in PyTorch and TensorFlow uses 8 2D convolution to assemble the results of capsule convolution, which is inefficient due to frequent kernel launch and low resource utilization. NeoFlow's capsule convolution achieves high speedup by implementing the computation within one kernel. To use capsule convolution in training tasks without code generation, we have to manually infer the gradient operators for capsule convolution and implement the low-level code, which requires great expertise and is time-consuming. In this paper, we propose NeoFlow leverage code generation techniques to support high-performance training with new operators automatically.

3 Overview of NeoFlow

Fig. 3 shows the overview of NeoFlow, which includes three parts: model definition, graph compilation, and graph execution. In the model definition part, NeoFlow defines a DNN model using expressions. The expressions are regarded as operators, while the symbolic inputs and outputs of the expressions are tensors. The defined model is then optimized by NeoFlow's automatic model transformations including layout transformation for better data access pattern and

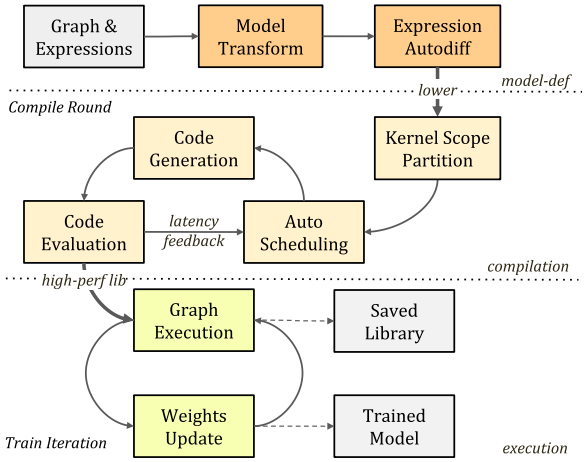


Fig. 3. Overview of NeoFlow.

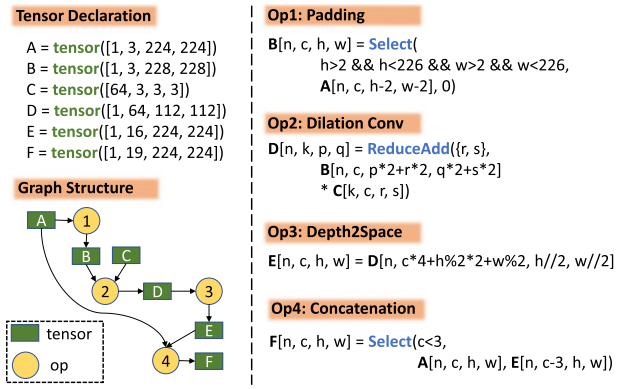


Fig. 4. Expression-based representation examples.

parallel operator fusion for simple graph structure. After model transformation, NeoFlow performs automatic differentiation (autodiff) to obtain the gradient graph, forming a new training graph. In the graph compilation part, NeoFlow partitions the entire DNN graph into subgraphs. To compile the subgraphs into efficient libraries, NeoFlow applies sophisticated optimizations including expression fusion, block/thread decomposition, data buffer configuration, loop unrolling, and vectorization within the subgraph. In the graph execution part, NeoFlow invokes the libraries from the compilation part in topological order to perform graph execution and weight updating. The final outputs of NeoFlow are the generated high-performance library and a trained model. NeoFlow provides both programming productivity and high performance. To encourage the innovation of new operators in DNNs and promote the development of ground-breaking DNNs, NeoFlow allows users to design new operators by writing mathematical expressions in Python.

4 Model Representation and Optimization

Existing tensor compilers such as TVM [17] and PlaidML [18] are limited in training because of their two-level IR infrastructure. NeoFlow uses a single-level expression-based IR system to represent graphs and operators, which makes it possible to define a DNN model by writing simple expressions. Then, we will introduce automatic differentiation and model optimization using the proposed expression-based representation.

4.1 Expression-Based Representation

As introduced in Section 2.2, existing compilers [17], [18], [19] use two levels of IR for compilation, which limits their support for training. In NeoFlow, we use a single level of IR for the compilation of both graph and operator. We name the IR as expression-based representation because the IR depicts the mathematical computation definition of the whole DNN model through simple arithmetic expressions. To capture all the information of a DNN model within a single level of IR, we focus on two aspects: how many data elements each layer produces and how each data element is produced by the layer. The first aspect is captured by tensor declaration, while the second aspect is captured by expression. In NeoFlow, we use

$$
\begin{aligned}
&B[x_1, \ldots, x_N] = \mathbf{F}_{R=\{r_1,\ldots,r_L\}} \\
&(A_1[f_1^1(x_1, ..., x_N, r_1, ..., r_L), \ldots, f_{M_1}^1(x_1, ..., x_N, r_1, ..., r_L)], \\
&A_2[f_1^2(x_1, ..., x_N, r_1, ..., r_L), \ldots, f_{M_2}^2(x_1, ..., x_N, r_1, ..., r_L)], \\
&\ldots, \\
&A_K[f_1^K(x_1, ..., x_N, r_1, ..., r_L), \ldots, f_{M_K}^K(x_1, ..., x_N, r_1, ..., r_L)])
\end{aligned} \tag{1}
$$

to represent the computation within one layer, where B is a N-dim output tensor of the layer, A_i $(1 \leq i \leq K)$ is an M_i-dim input tensor for this layer, and $\mathbf{F}$ is calculation operation (such as *ReduceAdd* for reduction sum, *Select* for conditional operation, etc.). $R = \{r_1, \ldots, r_L\}$ are index variables that represent reduction loops (empty if no reduction), and f_j^i $(1 \leq i \leq K, 1 \leq j \leq M_i)$ is a function on $x_1, x_2, \ldots, x_N$ and $r_1, \ldots, r_L$ that produces the corresponding access index for input tensors. In most cases, f_j^i is a (quasi-)affine transformation of index variables. The expression-based representation provides great flexibility for defining new operators as well as graph structures but introduces huge challenges for high-performance training. First, deep learning frameworks organize data arrays, gradient calculation, and graph optimization with respect to operators and define a list of operators, each with special treatments. However, this case-by-case design principle can't handle various arbitrary expressions. Second, the hand-optimized library such as CuDNN [10] often follows the fixed computation pattern such as loop order, data access pattern, etc., and thus fails to support arbitrary expressions. We address these challenges in NeoFlow by optimizations based on expressions as illustrated in the following sections.

In Fig. 4, we show a graph example using expression-based representation. The tensor declaration in Fig. 4 shows the number of elements each layer produces. There are four operations within the graph, and each operation corresponds to one layer. The four layers include both compute-intensive and memory-intensive operations. The first layer (*Op1*) is padding, whose padding border width is 2. The second layer (*Op2*) is dilation convolution [35], of which the stride and dilation factors are all 2. For this layer, the $\mathbf{F}$ in Equation 1 is *ReduceAdd*, and the reduction loops are $\{c, r, s\}$. The third layer (*Op3*) is depth2space [36]. The last layer (*Op4*) is concatenation of two tensors with a shortcut link to

the input tensor of the graph. This example graph shows the generality of expression-based representation.

4.2 Expression-Based Autodiff

Gradient calculation is an indispensable part of DNN training. Deep learning frameworks [7], [8], [9] usually adopt automatic gradient (autograd) [37] to compute gradients. For each operator, there exists a gradient operator in the operator inventory. The autograd algorithm traverses the DNN graph and finds all the backward paths from loss function to input tensors. Each path is composed of gradient operators from the operator inventory. Such an autograd algorithm requires the gradient operators to be prepared ahead of time. For new operators that can't be represented by existing operators in the inventory, the users must implement corresponding kernels manually by inferring the gradient formulas and writing low-level code. Although how to automatically get the gradient formulas for scientific computing expressions is well known, there still lacks a good automatic algorithm designed for deep learning expressions where operands are all tensors and computations are all deeply nested loops [34]. In NeoFlow, we develop an automatic differentiation (autodiff) algorithm to compute the gradients for expressions and whole DNN. Our gradient algorithm is a hybrid of symbolic gradient algorithm [34] and autograd algorithm [37]. For expressions, we use symbolic differentiation, and for the whole DNN, we use autograd algorithm.

In the following, we focus on the symbolic part of our algorithm and explain the calculation process in detail. Formally, for a given expression in Equation 1, if we want to calculate gradient to A_i, the autodiff algorithm should generate an expression

$$\begin{aligned}
&dA_i[z_1^i, \ldots, z_{M_i}^i] = \mathbf{H}_{R'=\{r'_1,...,r'_P\}} \\
&(dB[g_1(z_1^i, \ldots, z_{M_i}^i, r'_1, ..., r'_P), \ldots, g_N(z_1^i, \ldots, z_{M_i}^i, r'_1, ..., r'_P)], \\
&A_1[h_1^1(z_1^i, \ldots, z_{M_i}^i, r'_1, ..., r'_P), \ldots, h_{M_1}^1(z_1^i, \ldots, z_{M_i}^i, r'_1, ..., r'_P)], \\
&\ldots, \\
&A_K[h_1^K(z_1^i, \ldots, z_{M_i}^i, r'_1, ..., r'_P), \ldots, h_{M_K}^K(z_1^i, \ldots, z_{M_i}^i, r'_1, ..., r'_P)]),
\end{aligned} \tag{2}$$

where dA_i is the gradient tensor of A_i, dB is the gradient tensor of B, $\mathbf{H}$ is the gradient function corresponding to $\mathbf{F}$, $R' = \{r'_1, \ldots, r'_P\}$ are reduction axes, g_p and h_t^s $(1 \le p \le N, 1 \le s \le K, 1 \le t \le M_s)$ are all functions of $z_1^i, \ldots, z_{M_i}^i$ and $r'_1, \ldots, r'_P$ that produce indices for data access. The core problem of autodiff is to figure out the index function g_p and h_t^s $(1 \le p \le N, 1 \le s \le K, 1 \le t \le M_s)$ and determine the iteration domains of $r'_1, \ldots, r'_P$. We point out that this problem can be solved by linear algebra when the index functions used in the forward expression (the f_j^i in Equation (1)) are (quasi-)affine. Formally, the problem is expressed as the following linear equation system

$$\begin{aligned}
&f_1^i(x_1, ..., x_N, r_1, ..., r_L) = z_1, \\
&\ldots, \\
&f_{M_i}^i(x_1, ..., x_N, r_1, ..., r_L) = z_{M_i},
\end{aligned} \tag{3}$$

where $x_1, \ldots, x_N$ and $r_1, \ldots, r_L$ are considered as unknowns and $z_1, \ldots, z_{M_i}$ are considered as constants. If we can solve the unknowns in Equation 3, then we can easily substitute the unknowns in Equation 1 with the solutions and reorganize it to the form of Equation 2. Previous work [34] solves this problem with a prerequisite that index transformations are pure affine, that is, the f_j^i is always linear combination of $x_1, \ldots, x_N$ and $r_1, \ldots, r_L$. This seriously limits the scope of the operators of the DNNs. Important operators such as depth2space [36] shown in Fig. 4 are not supported, because there exist integer division and modular operations, which are not pure affine.

To address this problem, we substitute all the non-affine sub-expressions in $f_1^i, \ldots, f_{M_i}^i$ with new variables so that the remaining expressions are still pure affine. Then we solve this linear system and finally take the substituted sub-expressions back to the results by sepcially resolving quasi-affine operations such as division and modular. In detail, if an integer division sub-expression x/V (V is constant) is substituted by s_1, to solve this substitution, we first find corresponding substitutions for modular sub-expression $s_2 = x\%V$ because they together can solve $x = s_1 * V + s_2$. If we can't find such a pair of division and modular, we then introduce new free variables as reduction axes, such as $x = s_1 * V + r$ where r is a reduction axis. At last, we infer iteration domains for the reduction axes to finish our symbolic gradient process. With the gradient expressions for all the forward operators, we can construct the training graph by autograd algorithm as previous frameworks [7], [8], [9].

4.3 Automatic Model Transformation

Based on the above-proposed representation, we can perform automatic model transformations to further improve model performance at at a high-level. The transformations we consider include layout transformation, tensor swapping, and parallel fusion.

Layout Transformation. Layout is critical to the performance of DNNs for three reasons. First, coalesced data access can be enabled for proper layout and used to improve memory access efficiency. Second, memory address calculation can be reduced through vector loads with regular address intervals. Third, hardware-specific instructions such as AVX for CPU and Tensor Core for GPU can be leveraged to gain additional speedup.

Previous compilers such as TVM require a manual specification for layout transformation, and only provide limited choices for layout (e.g., NCHW, NHWC, etc.), which cannot fully exploit the speedup provided by layout transformation. Instead of requiring manual specifications, NeoFlow automatically applies transformation steps to change the tensor layout and corresponding expressions. To do this, NeoFlow first performs pattern matching for a vector or matrix within the expression and then split the matched dimensions of tensors to produce a packed innermost vector or matrix. In addition, NeoFlow is able to pack data into a small vector or matrix with a tunable vector or matrix size for performance tuning.

In Fig. 5 we show two examples for layout transformation. In part a) is a convolution expression with NCHW layout. Code generated from this expression will load input data and compute output data in the unit of scalar, so data load and address calculation overhead may slow down the

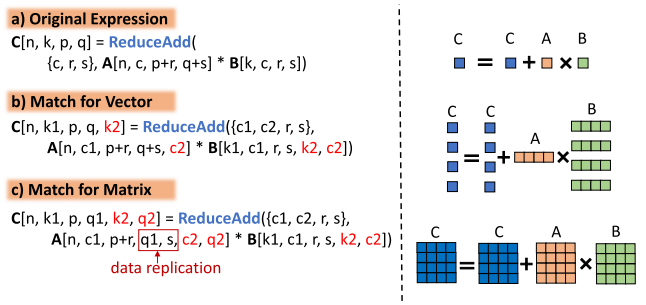


Fig. 5. Layout transformation examples.

program. In part b), the channel dimension of convolution is split into two parts and the inner part is moved to the innermost dimension. Such layout transformation allows vector load of tensor A and tensor B and reduces address calculation overhead. In part c), we show an additional dimension split for convolution to produce an innermost GEMM computation within the expression. This transformation enables the usage of Tensor Core instructions but increases data load overhead because tensor A is replicated s times (s is the kernel width). There is a tradeoff between layout transformation overhead and the efficiency of special load/compute instructions. Layout transformation brings additional data transfer overhead and requires additional fusion steps to achieve implicit transformation. And the split factors used when packing data into vector or matrix also influence performance. To obtain good performance, NeoFlow will tune the related factors (vector/matrix size) during code generation.

Tensor Swapping. Layout transformation alone is insufficient to optimize the program as it can't change the order of tensors within an expression. For most DNN models, the computation within one layer is commutative. For example, if we swap tensor A and tensor B in Fig. 5, the results remain unchanged. By changing the order of tensors, we have the chance to exploit better vectorization acceleration. For example, for matrix multiplication in DNN models, the expression is written as $C[i,j] = ReduceSum(\{k\}, A[i,k] * B[k,j])$, and thus vectorization is applicable to dimension j for tensor C and B. But when the j dimension is small and insufficient to use vector instructions, we can change the order of tensor A and B and rewrite the expression as $C[j,i] = ReduceSum(\{k\}, B[j,k] * A[k,i])$ so that dimension i is exposed for vectorization, which can potentially bring better performance. In NeoFlow, tensor swapping is enabled for binary operations such as matrix multiplication. Whether to apply tensor swapping is also a tunable factor during code generation.

Parallel Fusion. Parallel fusion [38], [39] is to merge two operators with no direct data dependency but share at least one common input tensor as one operator. Parallel fusion can help to improve parallelism. Using expression-based representation, we can easily detect such cases. For example, two expressions $A_1[i,k] * A_2[k,j_1]$ and $A_1[i,k] * A_3[k,j_2]$ can be merged as $A_1[i,k] * A_4[k,j]$ where A_4 is the concatenation of A_2 and A_3 along the second dimension (called a fusible dimension) (to get the outputs of original expressions, an additional tensor slice operation should also be appended). Our parallel fusion algorithm works as follows. First, for each operator (defined by an expression with output B, so we use B to denote an operator for simplicity), we first find fusible operators by enumerating all the operators that share input tensors with B and check if there is direct data dependency. If not, we record this fusible operator. Then, for each fusible operator B_1 and each dimension x_p of B_1, we check if the shape of B_1 except dimension x_p is the same as that of B, if so, we call x_p a fusible dimension. There may be many fusible dimensions, but we only take the first one and fuse B with B_1 along this dimension.

5 Compilation System

To generate efficient code for DNN models, NeoFlow adopts various optimization techniques including fusion and tuning. For training tasks, fusion and tuning are challenging because of the non-trivial tradeoff between recomputation and data transfer overhead. We first introduce subgraph partition and fusion and then explain the tuning of NeoFlow for code generation.

5.1 Subgraph Partition and Fusion

One DNN graph can contain tens or hundreds of operators. For some large networks such as ShuffleNet [40], there could be as many as 946 operators. Exploration-based compilers [17], [19] would take hours or days to compile and generate code for the entire training graph. Therefore, such a compilation manner will incur prohibitively high overhead and seriously delay the training process. NeoFlow addresses this problem by partitioning the original graph into small subgraphs and fuse the subgraphs into larger subgraph. Finally, NeoFlow generates one kernel for each fused subgraph.

To partition the graph, NeoFlow traverses the expression IR from output to input to capture producer-consumer relationships for the whole graph. For each edge (tensor) between two operations, NeoFlow creates a new placeholder to substitute the original tensor so that the two operations are separated from each other and form independent subgraphs. As a result, the whole graph is partitioned into a list of subgraphs, with each subgraph containing one layer of operation. The list of subgraphs will then be passed to the fusion process for further optimization.

To efficiently fuse these subgraphs, NeoFlow needs to address the tradeoff between recomputation and data transfer overhead. Such a challenge comes from the training scenario and is not revealed in previous fusion frameworks such as Relay [21] in TVM [17] and TASO [39] because they only focus on inference tasks. For training tasks, we are faced with two kinds of computation: forward and backward. Optimizing forward graph and optimizing backward graph are dual problems. If there exists a tensor in the forward graph, there will be an operation to compute the gradient for this tensor in the backward graph and vice versa. Moreover, the forward graph and backward graph are connected by lots of intermediate tensors. For example, to calculate the gradient for the weight of a convolution layer, the original input for this convolution layer is required, which is probably the intermediate result of previous layers (such as a ReLU [41] layer or another convolution layer). So if we fuse two layers in the forward graph and eliminate one intermediate tensor (reduce data transfer overhead), we will need to reproduce this tensor for the corresponding operations in the backward graph that require this

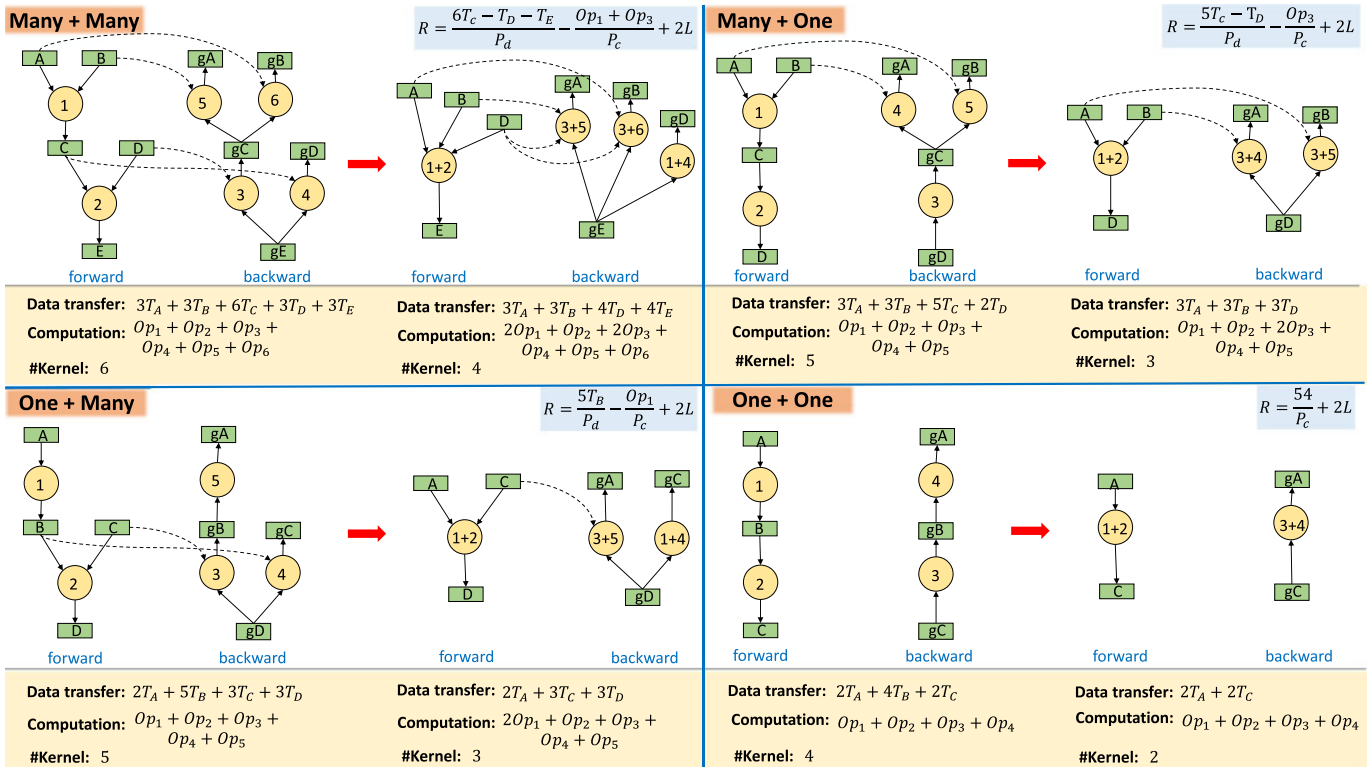


Fig. 6. Analysis of the results for different subgraph fusion decisions in NeoFlow.

intermediate tensor as input (bring recomputation overhead). At the same time, the gradient operation for this eliminated tensor should also be fused because there is no need to calculate the gradient for this tensor. Such connections between the forward graph and the backward graph make fusion for the training task more difficult.

NeoFlow addresses this challenge by analyzing the gains and costs for each fusion decision. To simplify the problem, we consider fusing two operations at a time with a single output. The analysis for multiple outputs is a dual problem, and we omit the discussion here. We divide operators into two categories according to the number of inputs (fan-in). For operators with only one input, we call them *One*, and for operators with multiple inputs, we call them *Many*. There are four combinations between *One* and *Many*, as listed in Fig. 6. We use $T_A, T_B, ...$ to represent the data transfer amount, and use $Op_1, Op_2, ...$ to represent the amount of computation. gA is the gradient for tensor A and other tensors are similar. #Kernels shows the number of generated kernels. By comparing the data transfer amount, computation amount, and kernel numbers for these cases, we have the following observations:

1) *One* + *One* fusion is always beneficial. We can reduce $4 \times T_B$ data transfer overhead without increasing any computation. Such a fusion can be applied to layers such as ReLU [41], Padding, Reshape, etc.
2) *One* + *Many* and *Many* + *One* fusion decisions are likely to be profitable because they reduce 4-5 tensor transfer requirements at the cost of increasing computation for one operation. However, whether such fusion decisions are profitable should be judged by precise calculations.
3) *Many* + *Many* fusion (such as convolution + convolution) increases computation for one forward operation and one backward operation, and whether the data transfer amount is reduced is up to the intermediate tensor size (if $6T_C > T_D + T_E$, then data transfer amount is reduced). So we still need further calculation to tell whether this fusion is beneficial.

During compilation, NeoFlow employs a simple reward function to judge whether a fusion decision is profitable. To predict the real runtime behavior is hard, but the principle of reward function is to filter out fusion decisions that are destined to be of low performance. So the reward function uses the peak performance to give an optimistic evaluation for each fusion decision, and if the reward under the optimistic assumption is still negative, NeoFlow can then safely refuse the fusion plan. We use P_d to denote peak data transfer bandwidth between off-chip memory and on-chip memory, and use P_c to denote the peak calculation performance of the device. We also use L to denote the average kernel launch overhead. P_d, P_c, and L are device-specific and can be measured using benchmarks [42]. The reward function is written as following.

$$R = \frac{\Delta T}{P_d} + \frac{\Delta C}{P_c} + \Delta K \times L, \tag{4}$$

where ΔT is the reduced amount of data transfer, ΔC is the reduced amount of computation, and ΔK is the number of eliminated kernel launches. These values can be negative if the corresponding metric is increased. We show the reward R for the four fusion decisions in Fig. 6. The reward for *One* + *One* is always positive, while for other cases, it depends on the concrete problem size. NeoFlow calculates these

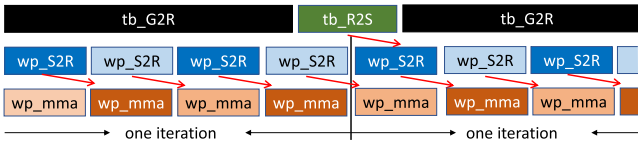


Fig. 7. Pipeline in the kernel generated by NeoFlow. *tb* represents threadblock. *wp* represents warp. *G2R* is data load from global to register. *R2S* is data store from register to shared memory. *S2R* is data load from shared memory to register. *mma* represents the Tensor Core instruction.

rewards during compilation and chooses the decision with the best reward greedily.

5.2 Tuning and Library Generation

In order to provide high-performance, it is necessary to select good tuning factors within the optimization space and generate high-performance libraries. NeoFlow explores several critical optimizations including layout-transformation factors, swapping factors, tiling factors, unrolling and pipelining factors, etc. All these optimization schedules together with their parameters form a large schedule space. NeoFlow explores the schedule space using a genetic algorithm. It first generates a set of initial parameters and then evaluates the performance on hardware. According to the evaluated performance, NeoFlow mutates the parameters to produce new factors. This process is repeated thousands of times and finally, a set of good parameters is found. On GPU, NeoFlow uses TVM to generate CUDA code and uses NVCC [43] to generate executable binary code. NeoFlow can also generate LLVM IR [44] to directly generate PTX code through LLVM backend. In addition, other platforms can be supported as LLVM supports multiple hardware devices. To use special instructions such as Tensor Core in GPU, NeoFlow uses CUDA WMMA instructions to perform matrix load, calculation, and store. Additional optimizations on Tensor Core such as register blocking and pipelining are supported. In detail, NeoFlow loads 1 to 4 fragments (tunable) within one warp before calculation, and these fragments are stored in registers. The load and computation are overlapped by double buffering, which forms a 3-level pipeline (shown in Fig. 7).

6 Runtime System

6.1 System Design

The runtime system can use the optimized library generated by the compilation system in either static or dynamic mode.

Static Mode. In static mode, the subgraphs are compiled ahead of time and the libraries are invoked at runtime. However, compiling and generating libraries for the entire DNN graph usually takes hours or days, but the benefit is that the generated libraries can be reused in later execution with no compilation overhead during execution. This mode is normally useful when the DNN models are stabilized.

Dynamic Mode. In dynamic mode, the subgraphs are compiled dynamically to generate the libraries. Our insight is that we can start training immediately with a moderate schedule and iteratively improve the schedules as the training proceeds. The model execution at early iterations also provides feedback to guide the selection of subgraph and optimization schedules for later iterations. Based on this insight, we develop a runtime that invokes the compilation incrementally. When the training starts to execute the generated libraries and update the weight, meanwhile we start the next round of compilation of subgraphs, overlapping the compilation and execution.

6.2 Implementation Details

For static mode, we can easily replace the subgraph with the generated library call. For dynamic mode, we need to mitigate the compilation overhead at runtime. The compilation is further divided into exploration and code generation parts. We use three threads at runtime, one for exploration, one for code generation, and one for execution. To communicate between threads, we use one message queue called *schedule-queue*. We also allocate an intermediate buffer called *function-cache* to cache the generated code, which is shared by the code generation thread and execution thread. The workflow of these three threads is as follows. The exploration thread explores optimization schedule space and at the end of each round, it puts the current optimal schedule into the schedule-queue. The code generation thread waits for schedules at the schedule-queue, when a new schedule comes, it generates code according to this schedule. Each generated code corresponds to a subgraph. If there is no record for the subgraph in *function-cache*, the generated code is immediately saved in *function-cache*. Otherwise, its performance is compared with the previously saved code, and only the better one is saved. To get the performance evaluation, the code generation thread runs the generated code on the target device and collects performance numbers. These performance numbers are sent back to guide the searching for better schedules. As for the execution thread, it executes the whole graph in order by using the functions in *function-cache*.

Fig. 8 presents an example of dynamic mode. In this example, we have four subgraphs to run, and we split the exploration for each subgraph into four rounds (each round contains multiple steps of exploration). Solution 1 is the static mode that does the four-round exploration in one shot and generates a library for each subgraph. After all the libraries are generated, the training can get started in the end. Solution 2 is an implementation of dynamic mode without *function-cache*, we overlap the exploration, code generation, and training threads (t1, t2, and t3). The optimal schedules of each round are used for code generation and training. But they are not stored for later usage, so the execution thread has to wait for the exploration thread for new schedules. Solution 3 is our final implementation of dynamic mode, we improve solution 2 by using *function-cache*, so the execution doesn't wait for exploration except for the first iteration. For training tasks, the dynamic mode can efficiently decrease the waiting time. We use multi-threading to accelerate compilation and evaluation in our runtime. For GPUs, we also support the usage of CUDA Graph [45] to further improve performance.

7 Experiments

7.1 Experiments Setup

We evaluate NeoFlow using a variety of DNNs. The benchmarks include both CNNs and RNNs. Some DNNs contain

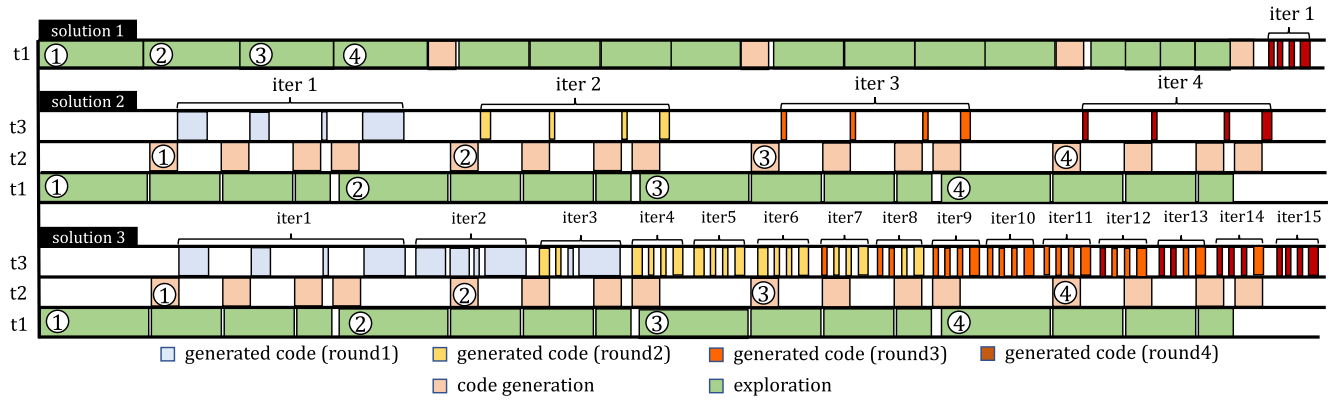


Fig. 8. Illustration of different runtime designs. Solution 1 is static mode and solution 3 is dynamic mode in NeoFlow.

new operators, posing challenges to deep learning frameworks. Among them, CapsuleNet [11], [12], [13] is implemented with two layers of capsule convolutions and three iterations of dynamic routing [12] between them; ShuffleNet [40] contains not only depthwise convolutions but also channel shuffle operators; MI-LSTM [14], SCRNN [15], subLSTM [47], and LLTM [48] are variants of LSTM [4]. We also evaluate common models such as ResNet [2], MobileNet [49], and Bert [5]. All performance experiments are done on Nvidia Tesla V100 GPU. We compare NeoFlow with PyTorch [7], TensorFlow [8], CuDNN [10], and AutoTVM [29]. The setup of our baselines is shown in Table 1. In the following, we evaluate both training and inference performance with different batch sizes. All the networks use FP32 data type except for Tensor Core. On Tensor Core, we use FP16 data type. Each benchmark takes about 8-10 hours (around 60-260 rounds) for compilation. We also discuss the benefit of our runtime in Section 7.3.

7.2 Performance Results

Training Performance. Fig. 9 compares NeoFlow with PyTorch for the seven models by varying the batch size from 1 to 64. We also show the performance of ResNet-50. For batch size 1, NeoFlow is $1.61\times$ the performance of PyTorch. But for larger batch sizes, NeoFlow cannot exceed PyTorch because the transposed convolution operator in backward graph is specially optimized in PyTorch with implicit GEMM algorithm, which is not implemented in NeoFlow currently. But for other models that are composed of new operators, the speedup of NeoFlow is significant. For CapsuleNet, PyTorch is slow because of the frequent kernel launch and low device utilization. For ShuffleNet, NeoFlow can explore deeper fusion such as fusing ReLU [41] with channel shuffle operator and convolution layer with batch normalization layer. The depthwise convolution [50] of ShuffleNet is slow in libraries due to poor optimization. But in NeoFlow, the operator is specially optimized by exploring various tiling, binding, and unrolling schedules. For MI-LSTM, SCRNN, subLSTM, and LLTM, their computations are variants of LSTM so their PyTorch implementations have to use multiple steps of GEMM, addition, and activation to obtain the final results. But in NeoFlow, we generate an efficient library by fusing the GEMM and elementwise operations together. Overall, NeoFlow achieves a geometric mean speedup of $3.16\times$ for these DNNs. When the CuDNN is enabled, the speedup to PyTorch is $1.92\times$.

Fig. 10 shows the comparison result with TensorFlow. Similar to PyTorch implementations, we manually implement baselines by replacing the new operators with multiple existing operators. We show the performance of TensorFlow with and without XLA. The average speedup to TensorFlow with XLA is $2.43\times$. XLA is a code generation compiler that generates efficient code for the new models. But the performance of XLA is inferior to NeoFlow because XLA can't tune optimization parameters to gain higher performance. NeoFlow can explore a schedule space to find better optimization choices during code generation.

Inference Performance With LLVM Backend. We show the inference performance of NeoFlow in Fig. 11 part a). Specially, we use LLVM [44] backend in NeoFlow to generate PTX code. The results show that for batch size 1 and 16,

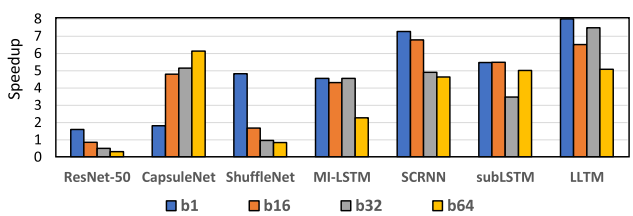


Fig. 9. Training speedup of NeoFlow compared to PyTorch.

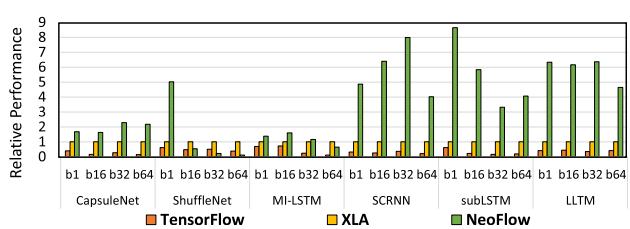


Fig. 10. Training speedup of NeoFlow compared to TensorFlow.

TABLE 1
Baseline setup

Name	Configuration
PyTorch	Version 1.10 with CUDA Graph [45] support.
TensorFlow	Version 2.4 with XLA [24]. Eager mode enabled.
CuDNN	Version 8.0. CUDA toolkit version is 11.0.
AutoTVM	Version 0.8, with XGBoost [46] tuner.

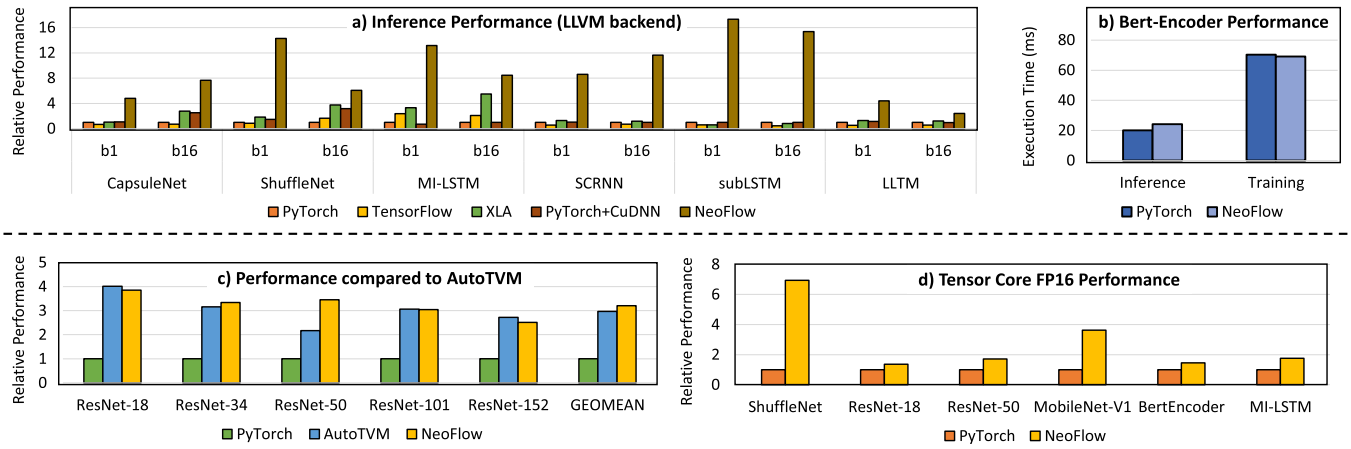


Fig. 11. Part a): Inference performance of NeoFlow using LLVM backend. Part b): Bert-Encoder performance. Part c): Performance comparison with AutoTVM. Part d): Performance of NeoFlow using FP16 Tensor Core.

NeoFlow can exceed PyTorch (with CuDNN) and TensorFlow (with XLA). The geometric mean speedup to PyTorch with CuDNN is $6.72\times$ and the speedup to TensorFlow with XLA is $4.96\times$.

Large Model Evaluation. We also evaluate performance using large models. We show the performance for Bert [5] encoder in Fig. 11 part b). The decoder of Bert can't be supported currently because the decoder has dynamic loops, but code generation compilers require a static loop structure. We use Bert-base configuration, which can saturate V100's device memory with batch size 1 (batch size 16 will exceed the memory capacity). According to the results, the inference and training execution time of NeoFlow is comparable to PyTorch.

Comparison With AutoTVM. We compare with AutoTVM for inference and show the results in Fig. 11 part c). AutoTVM requires Relay [21] to handle the graph-level code generation. Relay's IR is different from the IR used by AutoTVM, and the autograd module of Relay is not compatible with AutoTVM. As a result, we can't obtain the training performance using AutoTVM. For inference performance, NeoFlow is comparable with AutoTVM (geometric mean speedup is $1.08\times$).

Code Generation With FP16 Tensor Core. On V100 GPU, NeoFlow can also exploit the FP16 Tensor Core acceleration by generating CUDA WMMA instructions. We show the performance of NeoFlow using Tensor Core in Fig. 11 part d). The geometric mean speedup to PyTorch is $2.31\times$.

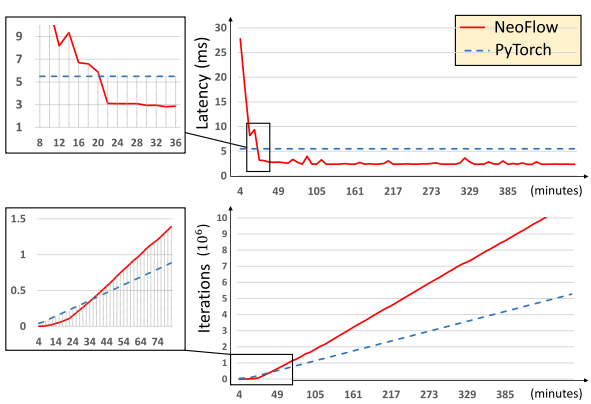


Fig. 12. Training performance of MI-LSTM.

7.3 Case Study: MI-LSTM

To evaluate the runtime of NeoFlow, we use the MI-LSTM [14] benchmark as a case study. MI-LSTM contains a hundred of subgraphs (35 unique). It replaces the original equation $\phi(Wx + Uz + b)$ in LSTM [4] with $\phi(Wx \odot Uz + b)$, where $\odot$ is Hadamard product [14]. Such a new equation is not well supported in deep learning frameworks but NeoFlow can support it using one expression. We use dynamic mode of runtime to train MI-LSTM and show how dynamic mode allows training to start early and improves overall system efficiency.

The results are shown in Fig. 12. We set the batch size to 64 and run the training tasks for 10 hours. The upper part of Fig. 12 shows the latency of each iteration (int milliseconds). The blue dashed line is our baseline (PyTorch) and the red solid line is NeoFlow. The training of NeoFlow starts 3.96 minutes late due to compilation, and the initial latency is 27.791 ms. But after 19.11 minutes, the latency drops to 5.35 ms, which is better than that of PyTorch (5.49ms). The fluctuation in Fig. 12 is caused by the evaluation noise on target device, but the latency becomes stable after about 1.5 hours. The dynamic mode allows the users to get the training started immediately (in minutes) without waiting for the entire DNN graph compilation to finish (in hours). On the other hand, we show the number of iterations done as time goes in the lower half of Fig. 12 (10^6 iterations as a unit). PyTorch proceeds at a steady speed while NeoFlow surpasses it within the first hour. Although fallen behind at the beginning, the number of iterations done by NeoFlow exceeds that of PyTorch after 41 minutes and the final number of iterations done by NeoFlow within 10 hours is about 2.22x than that of PyTorch.

7.4 Model Transformation Results

NeoFlow uses three important optimizations: layout transformation, tensor swapping, and parallel fusion. We use 2D convolution, GEMM, and capsule convolution to evaluate their effects. For layout transformation, we test a 2D convolution (feature size 14×14, channel size 1024, kernel size 1×1) with different batch sizes. Fig. 13 shows the effect of layout transformation. By default, we use NCHW layout, which is always slower than PyTorch when batch size is larger than 1. NeoFlow automatically transforms the

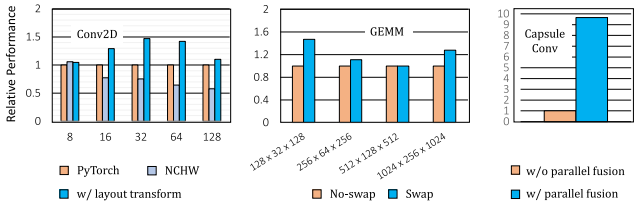


Fig. 13. Results of layout transformation and parallel fusion.

convolution to CHWN layout and utilizes vectorization optimization, leading to 1.25X speedup than PyTorch. For tensor swapping, we test four GEMM shapes. As shown in Fig. 13, the x-axis is the shape $M \times N \times K$. By swapping the order of operands, the performance is improved by 20.19% on average. For parallel fusion, we test a capsule convolution. We deliberately implement it with 8 2D convolutions along with tensor concatenation in NeoFlow. We compare the performance with and without parallel fusion. The parallel fusion fuses 8 2D convolutions together. With parallel fusion, we can improve the performance to 9.66X.

8 Related Work

Deep learning frameworks such as PyTorch [7], TensorFlow [8], and MXNet [9] rely on hand-optimized libraries including CuDNN [10] and MKL [26] to accelerate DNN inference and training. The libraries are specially optimized for a narrow range of operators, so the frameworks are restricted to an opaque and inflexible operator inventory, making it hard to accelerate novel operators and network architectures. On the other hand, Deep learning compilers such as Halide [16], TACO [51], TVM [17], PlaidML [18], and Tensor Comprehensions [19] provide high flexibility for deep learning code generation. They produce low-level code for customized operators through high-level expressions and compiler optimizations. Latte [52] and SWIRL [53] can generate high-performance code on CPU for inference and training, but they focus on typical layers such as convolution and pooling, while we focus on novel models. To get high-performance for DNN operators via these compilers, Halide autoscheduler [28] uses tree search and random programs; Chameleon [30] and FlexTensor [31] proposes to use reinforcement learning methods, while Ansor [32] uses an evolutionary search algorithm. However, these optimization frameworks focus on either single operator or inference scenarios, not applicable to DNN training acceleration.

Integrating code generation techniques to DNN frameworks for the acceleration of DNNs has gained more and more attention. PyTorch [7] and Relay [21] use TVM to generate code for DNN graphs but are restricted to inference tasks. Relay provides limited training support and is still not usable. nGraph [22] takes a model definition written in deep learning frameworks (such as TensorFlow) and generates low-level code via PlaidML. It has no direct support for novel DNNs due to the rigid interface and limited operator support. TensorFlow [8] and JAX [23] use XLA [24] to generate code for training, but the users can only define DNNs with supported (or traceable) primitives. To support autodiff for expressions, [33], [54] provide expression-based autodiff, but their optimizations on graph-level are limited.

9 Conclusion

Accelerating training for novel DNNs is in great demand with the rapid development of deep learning. In this paper, we introduce a flexible framework called NeoFlow for enabling efficient compilation for high performance DNN training. NeoFlow supports defintion of a model directly by expressions and provides expression-based autoddiff to support training tasks. Experiments for novel DNNs training on GPU show that NeoFlow can achieve better performance. The geometric mean speedups to PyTorch, TensorFlow, and CuDNN are 3.16X, 2.43X, and 1.92X, respectively.

Acknowledgments

We are immensely grateful to all the reviewers for their comments, which are helpful for the improvement of this paper.

References

[1] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. 3rd Int. Conf. Learn. Representations*, 2015. [Online]. Available: http://arxiv.org/abs/1409.1556

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778, doi: 10.1109/CVPR.2016.90.

[3] S. Xie, R. B. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 5987–5995, doi: 10.1109/CVPR.2017.634.

[4] H. Sak, A. W. Senior, and F. Beaufays, "Long short-term memory recurrent neural network architectures for large scale acoustic modeling," in *Proc. 15th Annu. Conf. Int. Speech Commun. Assoc.*, 2014, pp. 338–342. [Online]. Available: http://www.isca-speech.org/archive/interspeech_2014/i14_0338.html

[5] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," 2018, *arXiv:1810.04805v2*.

[6] T. B. Brown *et al.*, "Language models are few-shot learners," 2020, *arXiv:2005.14165*.

[7] A. Paszke *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Annu. Conf. Neural Inf. Process. Syst.*, 2019, pp. 8024–8035. [Online]. Available: http://papers.nips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library

[8] M. Abadi *et al.*, "Tensorflow: A system for large-scale machine learning," in *Proc. 12th USENIX Symp. Oper. Syst. Des. Implementation*, 2016, pp. 265–283. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[9] T. Chen *et al.*, "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," 2015, *arXiv:1512.01274v1*.

[10] S. Chetlur *et al.*, "cuDNN: Efficient primitives for deep learning," 2014, *arXiv:1410.0759v3*.

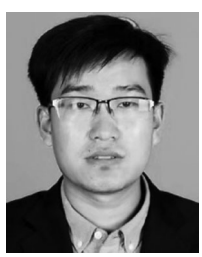
[11] G. E. Hinton, A. Krizhevsky, and S. D. Wang, "Transforming auto-encoders," in *Proc. 21st Int. Conf. Artif. Neural Netw. Artif. Neural Netw. Mach. Learn.*, 2011, pp. 44–51, doi: 10.1007/978–3-642-21735-7_6.

[12] S. Sabour, N. Frosst, and G. E. Hinton, "Dynamic routing between capsules," in *Proc. Annu. Conf. Neural Inf. Process. Syst.*, 2017, pp. 3856–3866. [Online]. Available: http://papers.nips.cc/paper/6975-dynamic-routing-between-capsules

[13] G. E. Hinton, S. Sabour, and N. Frosst, "Matrix capsules with EM routing," in *Proc. Int. Conf. Learn. Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=HJWLfGWRb

[14] Y. Wu, S. Zhang, Y. Zhang, Y. Bengio, and R. Salakhutdinov, "On multiplicative integration with recurrent neural networks," in *Proc. Annu. Conf. Neural Inf. Process. Syst.*, 2016, pp. 2856–2864. [Online]. Available: http://papers.nips.cc/paper/6215-on-multiplicative-integration-with-recurrent-neural-networks
[15] T. Mikolov, A. Joulin, S. Chopra, M. Mathieu, and M. Ranzato, "Learning longer memory in recurrent neural networks," in *Proc. 3rd Int. Conf. Learn. Representations*, 2015.
[16] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. P. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2013, pp. 519–530, doi: 10.1145/2491956.2462176.
[17] T. Chen *et al.*, "TVM: An automated end-to-end optimizing compiler for deep learning," in *Proc. 13th USENIX Symp. Oper. Syst. Des. Implementation*, 2018, pp. 578–594. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/chen
[18] Intel, "PlaidML," 2020, Accessed: Jul. 10, 2020. [Online]. Available: https://ai.intel.com/plaidml
[19] N. Vasilache *et al.*, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," 2018, *arXiv:1802.04730v3*.
[20] Facebook, "PyTorch TVM integration," 2020, Accessed: Aug. 05, 2020. [Online]. Available: https://github.com/pytorch/tvm
[21] J. Roesch *et al.*, "Relay: A high-level IR for deep learning," 2019, *arXiv:1904.08368v2*.
[22] S. Cyphers *et al.*, "Intel ngraph: An intermediate representation, compiler, and executor for deep learning," 2018, *arXiv:1801.08058v2*.
[23] R. Frostig, M. J. Johnson, and C. Leary, "Compiling machine learning programs via high-level tracing," *Syst. Mach. Learn.*, pp. 23–24, 2018.
[24] Google, "TensorFlow XLA," 2020, Accessed: Aug. 05, 2020. [Online]. Available: https://www.tensorflow.org/xla
[25] NVIDIA, "CUBLAS," 2021, Accessed: Jun. 21, 2021. [Online]. Available: https://docs.nvidia.com/cuda/cublas/index.html
[26] Intel, "MKL," 2020, Accessed: Jul. 10, 2020. [Online]. Available: https://software.intel.com/en-us/mkl
[27] Intel, "Onednn," 2020, Accessed: Aug. 08, 2020. [Online]. Available: https://github.com/oneapi-src/oneDNN
[28] A. Adams *et al.*, "Learning to optimize halide with tree search and random programs," *ACM Trans. Graph.*, vol. 38, no. 4, pp. 121:1–121:12, 2019, doi: 10.1145/3306346.3322967.
[29] T. Chen *et al.*, "Learning to optimize tensor programs," in *Proc. Annu. Conf. Neural Inf. Process. Syst.*, 2018, pp. 3393–3404. [Online]. Available: http://papers.nips.cc/paper/7599-learning-to-optimize-tensor-programs
[30] B. H. Ahn, P. Pilligundla, A. Yazdanbakhsh, and H. Esmaeilzadeh, "Chameleon: Adaptive code optimization for expedited deep neural network compilation," in *Proc. 8th Int. Conf. Learn. Representations*, 2020.
[31] S. Zheng, Y. Liang, S. Wang, R. Chen, and K. Sheng, "FlexTensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system," in *Proc. Int. Conf. Architectural Support Program. Languages Oper. Syst.*, 2020, pp. 859–873, doi: 10.1145/3373376.3378508.
[32] L. Zheng *et al.*, "Ansor: Generating high-performance tensor programs for deep learning," 2020, *arXiv:2006.06762*.
[33] T.-M. Li, M. Gharbi, A. Adams, F. Durand, and J. Ragan-Kelley, "Differentiable programming for image processing and deep learning in halide," *ACM Trans. Graph.*, vol. 37, no. 4, pp. 1–13, 2018.
[34] S. Urban and P. van der Smagt, "Automatic differentiation for tensor algebras," 2017, *arXiv:1711.01348v1*.
[35] F. Yu and V. Koltun, "Multi-scale context aggregation by dilated convolutions," in *Proc. 4th Int. Conf. Learn. Representations*, 2016.
[36] W. Shi *et al.*, "Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 1874–1883, doi: 10.1109/CVPR.2016.207.
[37] D. Maclaurin, D. Duvenaud, and R. P. Adams, "Autograd: Effortless gradients in numpy," in *Proc. Int. Conf. Mach. Learn. Workshop*, 2015, Art. no. 5.
[38] Z. Jia, J. J. Thomas, T. Warszawski, M. Gao, M. Zaharia, and A. Aiken, "Optimizing DNN computation with relaxed graph substitutions," in *Proc. Mach. Learn. Syst*, 2019. [Online]. Available: https://proceedings.mlsys.org/book/276.pdf
[39] Z. Jia, O. Padon, J. J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, "TASO: Optimizing deep learning computation with automatic generation of graph substitutions," in *Proc. 27th ACM Symp. Oper. Syst. Princ.*, 2019, pp. 47–62, doi: 10.1145/3341301.3359630.
[40] X. Zhang, X. Zhou, M. Lin, and J. Sun, "ShuffleNet: An extremely efficient convolutional neural network for mobile devices," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 6848–6856.
[41] A. F. Agarap, "Deep learning using rectified linear units (ReLU)," 2018, *arXiv:1803.08375v2*.
[42] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the NVIDIA volta GPU architecture via microbenchmarking," 2018, *arXiv:1804.06826v1*.
[43] "CUDA NVCC compiler," 2020. [Online]. Available: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/
[44] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. 2nd IEEE/ACM Int. Symp. Code Gener. Optim.*, 2004, pp. 75–88, doi: 10.1109/CGO.2004.1281665.
[45] Nvidia, "CUDA grah," 2021, Accessed: Nov. 05, 2021. [Online]. Available: https://developer.nvidia.com/blog/cuda-graphs/
[46] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2016, pp. 785–794, doi: 10.1145/2939672.2939785.
[47] R. P. Costa, I. A. M. Assael, B. Shillingford, N. de Freitas, and T. P. Vogels, "Cortical microcircuits as gated-recurrent neural networks," in *Proc. Annu. Conf. Neural Inf. Process. Syst.*, 2017, pp. 272–283. [Online]. Available: http://papers.nips.cc/paper/6631-cortical-microcircuits-as-gated-recurrent-neural-networks
[48] Facebook, "PyTorch LLTM Implementation," 2020, Accessed: Aug. 16, 2020. [Online]. Available: https://pytorch.org/tutorials/advanced/cpp_extension.html
[49] A. G. Howard *et al.*, "MobileNets: Efficient convolutional neural networks for mobile vision applications," 2017, *arXiv:1704.04861v1*.
[50] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," 2016, *arXiv:1610.02357v3*.
[51] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. P. Amarasinghe, "The tensor algebra compiler," *Proc. ACMon Program. Languages*, vol. 1, pp. 77:1–77:29, 2017, doi: 10.1145/3133901.
[52] L. Truong *et al.*, "Latte: A language, compiler, and runtime for elegant and efficient deep neural networks," in *Proc. 37th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2016, pp. 209–223, doi: 10.1145/2908080.2908105.
[53] A. Venkat, T. Rusira, R. Barik, M. Hall, and L. Truong, "SWIRL: High-performance many-core CPU code generation for deep neural networks," *Int. J. High Perform. Comput. Appl.*, vol. 33, no. 6, pp. 1275–1289, 2019, doi: 10.1177/1094342019866247.
[54] R. Wei, L. Schwartz, and V. Adve, "DLVM: A modern compiler infrastructure for deep learning systems," 2017, *arXiv:1711.03016*.

Size Zheng (Student Member, IEEE) received the BS degree from the Department of Computer Intelligence Science, Peking University, Beijing, China, in 2019. He is currently working toward the PhD degree with the Department of Computer Science, School of EECS, Peking University. His research interests include deep learning compilers, software optimization for GPU architectures, and code generation.



Xiuhong Li received the BS and PhD degrees from Peking Univiersity, China. He is currently a senior researcher with Deep Learning Platform Department, Sensetime. His research interests include deep learning compiler, distributed parallel computing, and high-performance computation on heterogeneous architectures.

Shengen Yan received the BS degree from the Harbin Institute of Technology, Harbin, China, in 2009, and the PhD degree from the Institute of Software, Chinese Academy of Sciences, Beijing, China, in 2014. He was a visiting student with NC State University, Raleigh, North Carolina from June 2013 to February 2014. He was a postdoctoral researcher with Multimedia Lab, Chinese University of Hong Kong, Hong Kong, from 2015 to 2017. He is currently the executive research director of Data and Computing Platform Department, SenseTime. He has authored or coauthored about 30 papers in the area of parallel computing and deep learning. His research interests include large scale deep learning and high performance computing.

Yun Liang (Senior Member, IEEE) is currently an associate professor (with tenure) with the School of EECS, Peking University, China. His research interests include computer architecture, compiler, electronic design automation, and embedded system. He has authored more than 90 scientific publications in premier international journals and conferences in related domains. He was the recipient of the best paper awards at FCCM 2011 and ICCAD 2017 and best paper nominations at PPoPP 2019, DAC 2017, ASPDAC 2016, DAC 2012, FPT 2011, and CODES+ISSS 2008 for his research. He is an associate editor for *ACM Transactions in Embedded Computing Systems* (TECS), *ACM Transactions on Reconfigurable Technology and Systems* (TRETS), and *Embedded System Letters* (ESL). He is also with the program committees in the premier conferences in the related domain including MICRO, DAC, HPCA, FPGA, ICCAD, FCCM, ICS, etc.