TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions

Zhihao Jia
Stanford University
zhihao@cs.stanford.edu

Oded Padon
Stanford University
padon@cs.stanford.edu

James Thomas
Stanford University
jjthomas@stanford.edu

Todd Warszawski
Stanford University
twarszaw@stanford.edu

Matei Zaharia
Stanford University
matei@cs.stanford.edu

Alex Aiken
Stanford University
aiken@cs.stanford.edu

Abstract

Existing deep neural network (DNN) frameworks optimize the computation graph of a DNN by applying graph transformations manually designed by human experts. This approach misses possible graph optimizations and is difficult to scale, as new DNN operators are introduced on a regular basis.

We propose TASO, the first DNN computation graph optimizer that *automatically* generates graph substitutions. TASO takes as input a list of operator specifications and generates candidate substitutions using the given operators as basic building blocks. All generated substitutions are formally verified against the operator specifications using an automated theorem prover. To optimize a given DNN computation graph, TASO performs a cost-based backtracking search, applying the substitutions to find an optimized graph, which can be directly used by existing DNN frameworks.

Our evaluation on five real-world DNN architectures shows that TASO outperforms existing DNN frameworks by up to 2.8×, while requiring significantly less human effort. For example, TensorFlow currently contains approximately 53,000 lines of manual optimization rules, while the operator specifications needed by TASO are only 1,400 lines of code.

CCS Concepts • **Computing methodologies** → **Neural networks**; • **Computer systems organization** → *Neural networks*; • **Software and its engineering** → *Formal software verification.*

Keywords deep neural network, computation graph substitutions, superoptimization, formal verification

SOSP '19, October 27–30, 2019, Huntsville, ON, Canada

ACM ISBN 978-1-4503-6873-5/19/10…$15.00
https://doi.org/10.1145/3341301.3359630

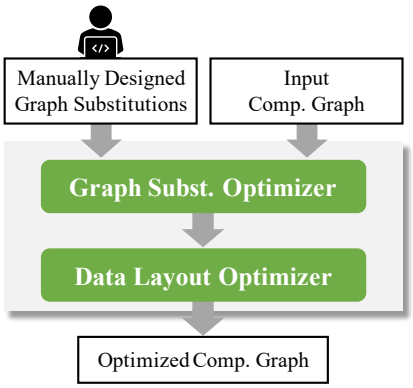
ACM Reference Format:
Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. In *ACM SIGOPS 27th Symposium on Operating Systems Principles (SOSP '19), October 27–30, 2019, Huntsville, ON, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3341301.3359630

1 Introduction

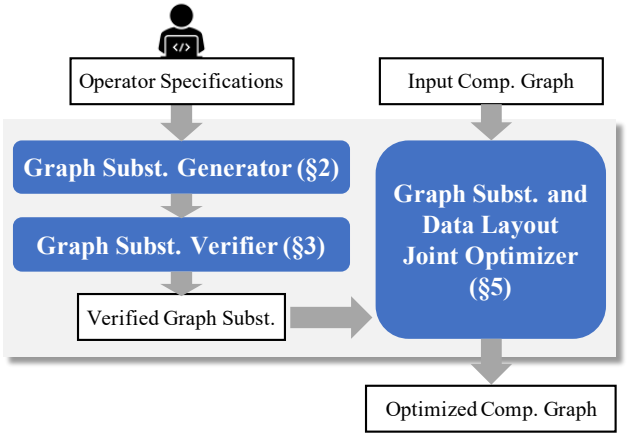
Deep neural network (DNN) frameworks represent a neural architecture as a *computation graph*, where each node is a mathematical tensor operator (e.g., matrix multiplication, convolution, etc.). To improve the runtime performance of a computation graph, the most common form of optimization is *graph substitutions* that replace a subgraph matching a specific pattern with a functionally equivalent subgraph with improved performance.

Existing DNN frameworks optimize a computation graph by applying graph substitutions that are manually designed by domain experts, as depicted in Figure 1a. For example, TensorFlow, PyTorch, TensorRT, and TVM use a greedy rule-based optimization strategy and directly perform all applicable substitutions (i.e., rules) on an input computation graph [6, 8, 31, 36]. MetaFlow [21] allows substitutions that may either increase or decrease performance to enable a larger search space of equivalent computation graphs and uses back-tracking search to explore this space, but it still requires manually specified substitutions. Although manually designed substitutions improve the performance of DNN computations, they fall short in several respects.

Maintainability. Hand-written graph substitutions require significant engineering effort. For example, TensorFlow r1.14 includes 155 substitutions implemented in approximately 53K lines of C++ code. The maintenance problem is aggravated by the fact that new operators are continuously introduced; for example, recent work has proposed depthwise [19], grouped [38], and transposed convolutions [16] for different image classification tasks. TensorFlow r1.14 currently includes 17 graph substitutions (written in 4K lines of code) to optimize ordinary convolution (e.g., fusing it with different types of operators). With the existing approach,



(a) Existing DNN frameworks.



(b) TASO.

Figure 1. Comparing computation graph optimization in existing DNN frameworks with TASO.

supporting each new convolution variant would require a similar implementation effort, as each has slightly different semantics and cannot be directly optimized using existing substitutions.

Data layout. Tensor data can be stored in memory in various layouts, and this choice has a high impact on runtime performance. The best layout depends on both the operator and the hardware. For example, on a P100 GPU, convolution performs best with row-major layout (i.e., the inner-most dimension is contiguously stored), while matrix multiplication performs best with column-major layout (i.e., the outer-most dimension is contiguously stored). On a Tesla V100 GPU with tensor cores [5] supporting 4×4 matrix operations, optimal performance may require tiling tensors into 4×4 chunks. However, considering layout transformations together with graph substitutions adds another level of complexity. For example, a graph substitution may only improve performance if it is combined with a particular layout transformation (see Section 7.5). Current frameworks avoid this complexity by treating data layout and graph substitution as separate optimization problems and solve them sequentially [8, 27], as shown in Figure 1a, but this separation misses many possible optimization opportunities.

Correctness. Hand-written graph substitutions are error-prone, and a bug in graph substitutions can lead to incorrect computation graphs [2, 4]. The same issue arises in compiler optimization, where an incorrect optimization leads to incorrect programs. In the compiler literature, significant effort has been devoted to formally verifying optimizations [7, 11, 13, 23, 29, 30, 33, 35]. However, to the best of our knowledge, such techniques have not been applied to graph substitution optimizations performed by DNN frameworks.

1.1 Our Approach

In this paper, we present TASO (Tensor Algebra SuperOptimizer), the first DNN computation graph optimizer that *automatically generates* graph substitutions. Figure 1b shows an overview of TASO, which differs from existing frameworks in three aspects. First, TASO only requires operator definitions and specifications, and automatically generates graph substitutions, reducing manual effort. Second, TASO employs formal verification to ensure correctness of the generated graph substitutions. Finally, TASO jointly optimizes graph substitution and data layout, achieving significantly better runtime performance.

Generating substitutions. TASO's *graph substitution generator* enumerates all possible computation graphs over a given set of DNN operators (e.g., the cuDNN kernels [10]) up to a fixed size, and executes them on a set of random input tensors. Any pair of computation graphs that have identical results on the random inputs are considered as a candidate substitution. To efficiently find all such pairs, TASO constructs a hash table where computation graphs are stored based on the hash of their outputs for the random inputs.

Formal verification. TASO's *graph substitution verifier* is used to ensure correctness of the generated graph substitutions, relying on user provided *operator properties*. Operator properties capture mathematical properties of operators, e.g., linearity of convolution. The full list of 43 operator properties we used appears in Table 2. As our evaluation shows, a small set of properties for each operator suffices to prove the correctness of complex substitutions.

Formally, we model tensor operators using a symbolic representation based on first-order logic that is agnostic to the size of the underlying tensors, and can succinctly express operator properties. The verifier uses the specified properties to check the correctness of all generated graph substitutions using an automated theorem prover.

We also present a methodology for developing operator properties, which assists the developer in two ways: (1) discovery of required properties is guided by the graph substitution generator, and (2) operator properties are subject to further validation using symbolic execution on tensors of small sizes. During the development process, we found that the our verification methodology uncovered several bugs, both in the operator specifications and in the implementation of the graph substitution generator.

Joint optimization. TASO jointly optimizes graph substitutions and data layout transformations by integrating them into a common representation. TASO uses the cost-based backtracking search algorithm of MetaFlow [21] and extends its cost model to also capture performance differences that arise from different data layouts. During the search, TASO measures the performance of a proposed DNN operator with a specific proposed data layout on the hardware. These individual measurements are used to predict the performance of an entire computation graph with specific data layouts.

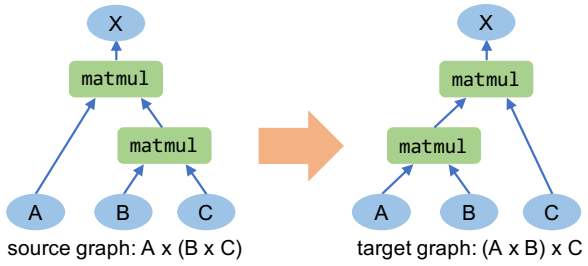
Evaluation. We evaluate TASO on five real-world DNN architectures. For widely used DNNs optimized by existing frameworks, such as ResNet-50 [18], TASO matches the performance of these frameworks with hand-written rules by using operator definitions and specifications 1,400 lines long.

For new DNN architectures such as ResNeXt-50 [38], NasRNN [39], NasNet-A [40], and BERT [15], TASO is up to 2.8× faster than state-of-the-art frameworks, by automatically discovering novel graph substitutions to optimize these architectures. Compared to sequentially optimizing graph substitutions and data layout, we show that the joint optimization can further improve performance by 1.2×. In all experiments, TASO discovered an optimized graph in less than ten minutes, making it feasible to use when optimizing a DNN architecture before large-scale deployment.

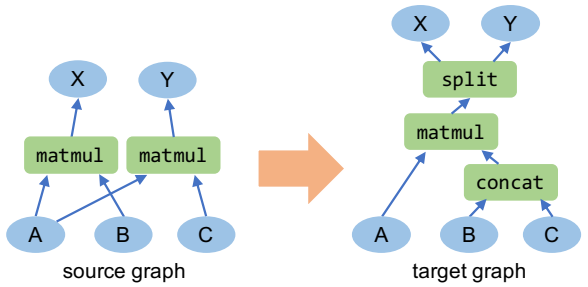
2 Graph Substitution Generator

This section describes the TASO substitution generator that automatically generates potential substitutions given a list of primitive operators. The generation algorithm finds all valid substitutions up to a certain size.

To find all potential substitutions, a straightforward approach is to test all pairs of graphs for equivalence, which requires a quadratic number of tests between graphs. We adopt an idea from compiler superoptimization [7] and compute a *fingerprint* for each graph, which is a hash of the graph outputs on some specific inputs. Two graphs are certainly not equivalent if they have different fingerprints, and so by only comparing graphs with the same fingerprint, TASO significantly reduces the number of equivalence tests. In the experiments, we observe that all graphs with the same fingerprint are verified equivalent by TASO.



(a) Associativity of matrix multiplication.



(b) Fusing two matrix multiplications using concatenation and split.

Figure 2. Graph substitution examples.

2.1 Graph Substitution Definition

A *graph substitution* consists of three components: (1) a *source graph* that is matched to subgraphs in a computation graph; (2) a *target graph*[1] that defines a functionally equivalent new subgraph to replace the matched subgraph; and (3) a *mapping* relation between input/output tensors in the source and target graphs. Figure 2a shows an example graph substitution using the associativity of matrix multiplication. Figure 2b fuses two matrix multiplications into one using concatenation and split along the row dimension. A, B, C, X, and Y identify the mapping between input and output tensors in the source and target graphs.

A graph substitution is specified independently of the concrete tensor shapes. For example, the substitutions of Figure 2 can be applied to tensors A, B, and C of any concrete shape. Some operators also depend on *configuration parameters* to determine the behavior of the operator. For example, the parameters of convolution determine the strides, padding, and activation (e.g., applying the `relu` function [28] as part of convolution); and the parameters of split or concatenation determine the axis along which to apply the operator.

Concatenation and split operators. Concatenation and split operators are commonly used in fusing operators with shared inputs, as illustrated in Figure 2b. A split operator partitions a tensor into two disjoint sub-tensors along a

[1]In some of the superoptimization literature, what we call the *source* is called the *target*, and what we call the *target* is called the *rewrite*.

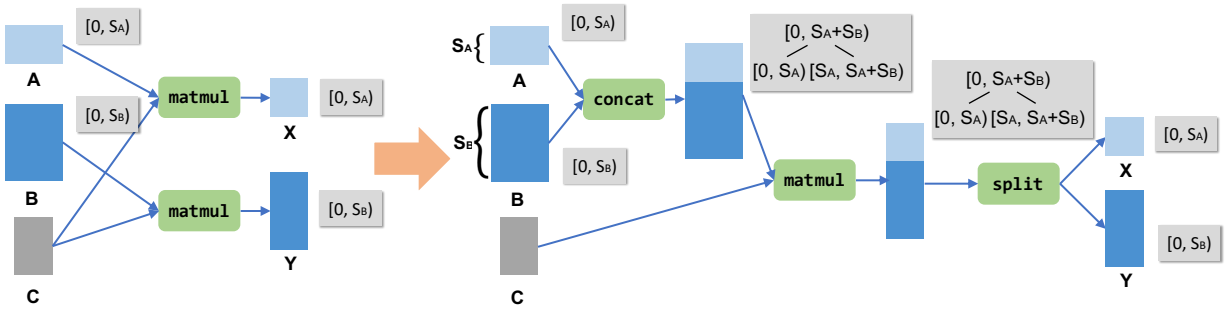


Figure 3. A graph substitution for fusing matrix multiplications with a shared input. The target graph has a `concat` and a `split` operator, both of which are performed along the row dimension of a matrix. The split tree of the row dimension for each tensor is shown in a gray box.

dimension determined by its parameter. This presents a complication, as the split point cannot be inferred from the input tensors or the parameter. To solve this problem, we observe that a split operator always partitions a tensor at previous concatenation points to "undo" the most recent concatenation operator. We use this fact to define a suitable semantics for the split operator.

Formally, we maintain a *split tree* for each dimension of a tensor to track the concatenation history. Figure 3 shows the split trees of the row dimension for all tensors in Figure 2b. The split trees allow the substitution to recover the split point without introducing any additional parameters. Our approach also supports multi-way concatenation and split by nesting of concatenation and split operators.

2.2 Generation Algorithm

For a given set of operator specifications, TASO generates potential graph substitutions in two steps, as shown in Algorithm 1.

Step 1: Enumerating potential graphs and collecting their fingerprints. TASO first enumerates all potential graphs up to a certain size by using a given set of operators. To construct a graph, TASO iteratively adds an operator in the current graph by enumerating the type of the operator and the input tensors to the operator. The input tensors can be initial input tensors to the graph (e.g., A, B, and C in Figure 2) or the output tensors of previous operators (e.g., the output of the `matmul` and `concat` operators in Figure 2).

Algorithm 1 (line 7-18) shows a *depth-first search* algorithm for constructing all acyclic computation graphs that do not contain duplicated computation. We say a graph contains duplicated computation if it has two operators performing the same computation on the same input tensors. The generator ignores such graphs as they do not represent useful computation graphs.

For each graph, we collect its fingerprint, which is a hash of the output tensors obtained by evaluating the graph on some input tensors. TASO uses both randomly initialized tensors and a number of constants as inputs to allow finding substitutions involving constant tensors, such as the identity matrix (see examples in Section 7.3). To avoid floating-point errors in computing a fingerprint, all tensors are represented with integers, following the method introduced in [37].

Since a graph can have an arbitrary number of output tensors, the hash function must ensure the fingerprint is independent of any permutation of the output tensors. To guarantee this property, TASO employs a two-step hash

Algorithm 1 Graph substitution generation algorithm.

```
1:  Input: A set of operators 𝒫, and a set of input tensors ℐ.
2:  Output: Candidate graph substitutions 𝒮.
3:
4:  // Step 1: enumerating potential graphs.
5:  𝒟 = {} // 𝒟 is a graph hash table indexed by their fingerprints.
6:  Build(1, ∅, ℐ)
7:  function Build(n, 𝒢, ℐ)
8:      if 𝒢 contains duplicated computation then
9:          return
10:     𝒟 = 𝒟 + (FingerPrint(𝒢), 𝒢)
11:     if n < threshold then
12:         for op ∈ 𝒫 do
13:             for i ∈ ℐ and i is a valid input to op do
14:                 Add operator op into graph 𝒢.
15:                 Add the output tensors of op into ℐ.
16:                 Build(n + 1, 𝒢, ℐ)
17:                 Remove operator op from 𝒢.
18:                 Remove the output tensors of op from ℐ.
19:
20: // Step 2: testing graphs with identical fingerprint.
21: 𝒮 = {}
22: for 𝒢1, 𝒢2 ∈ 𝒟 with the same FingerPrint(·) do
23:     if 𝒢1 and 𝒢2 are equivalent for all test cases then
24:         𝒮 = 𝒮 + (𝒢1, 𝒢2)
25: return 𝒮
```

function to compute fingerprints as follows.

$$\textsc{FingerPrint}(\mathcal{G}) = hash_2\left(\{hash_1(t_i) \mid i \in \textsc{Outputs}(\mathcal{G})\}\right)$$

where t_i are the output tensors of graph $\mathcal{G}$. $hash_1$ computes the states and content of an output tensor, including the size, shape, and content of the tensor. $hash_2$ is a symmetric hash function applied to an unordered set of hash values.

Step 2: Testing graphs with identical fingerprint. For graphs with the same fingerprint, TASO further examines their equivalence on a set of test cases. Similar to collecting fingerprints, each test case contains a set of randomized input tensors, and two graphs pass if they produce equivalent output tensors for all test cases. Unlike the fingerprints, these tests use floating point numbers ranging between −1 and 1, and classify two output tensors as equivalent if their outputs differ by no more than a small threshold value, which is 10^{-5} in the evaluation. For this threshold, we observed no discrepancy from the integer tests. However, it is possible to use a smaller threshold to filter out substitutions that are valid for real numbers but result in floating point errors.

Each pair of graphs passing the random testing becomes the source and target graphs of a candidate graph substitution, and the mapping relation between the input/output tensors in the source and target graphs can be automatically inferred from the test cases. All candidate graph substitutions are then sent to the substitution verifier to check their correctness (Section 3), and later pruned to eliminate redundant substitutions (Section 4).

The algorithm described so far is generic, in the sense that it does not depend on the specific tensor operators used. However, we observed that for DNN applications, there are two operators that require special handling. The `relu` operator [28], which is commonly used in DNN applications, returns 0 for all negative inputs. As `relu` often returns 0, it results in many superfluous substitutions being valid. To prevent these substitutions from being generated, the generator replaces `relu` by an arbitrary non-linear function (our implementation uses $x \mapsto x(x+1)+1$). The `enlarge` operator increases the size of a tensor by padding it with zeros, which is useful for fusing convolutions with different kernel sizes [21]. However, the presence of zeros also results in many superfluous substitutions. To overcome this, the generator only considers computation graphs in which `enlarge` is applied to an input tensor, i.e., not to the output of another operator. This restriction captures the intended use of `enlarge` for fusing convolutions, while avoiding the superfluous substitutions.

It is worth noting that prior work [7] reported false positives in using random testing to examine code transformations in compiler superoptimization. They observed that a number of incorrect code transformations passed a set of test cases. We have not observed any false positive cases in all the experiments. We use a single test case to examine all graph pairs with the same fingerprint, and all substitutions passing the test case are correct and verified by the substitution verifier. This is likely due to the high arithmetic density of DNN operators and the lack of branching (if statements) in computation graphs. As a reference, [17] shows that for programs with only linear operators, the probability that two nonequivalent programs produce identical output on a random input is at most $\frac{1}{d}$, where d is the number of possible values for a variable (i.e., $d = 2^{32}$ in TASO).

Table 1. Tensor operators and constant tensors included in TASO. Similar to existing DNN frameworks [6, 31], pooling and convolution operators support different strides and padding modes (i.e., $\mathsf{P}_{\mathtt{same}}$ and $\mathsf{P}_{\mathtt{valid}}$); convolution supports different activation functions (i.e., $\mathsf{A}_{\mathtt{none}}$ and $\mathsf{A}_{\mathtt{relu}}$). Section 6 provides more details on the usage of the constants.

Name	Description	Parameters
Tensor Operators		
`ewadd`	Element-wise addition	
`ewmul`	Element-wise multiplication	
`smul`	Scalar multiplication	
`transpose`	Transpose	
`matmul`	Batch matrix multiplication[#]	
`conv`	Grouped convolution[%]	stride, padding, activation
`enlarge`	Pad conv. kernel with zeros[†]	kernel size
`relu`	Relu operator	
$\mathtt{pool}_{\mathtt{avg}}$	Average pooling	kernel size, stride, padding
$\mathtt{pool}_{\mathtt{max}}$	Max pooling	kernel size, stride, padding
`concat`	Concatenation of two tensors	concatenation axis
$\mathtt{split}_{\{0,1\}}$	Split into two tensors	split axis
Constant Tensors		
$\mathsf{C}_{\mathtt{pool}}$	Average pooling constant	kernel size
$\mathsf{I}_{\mathtt{conv}}$	Convolution id. kernel	kernel size
$\mathsf{I}_{\mathtt{matmul}}$	Matrix multiplication id.	
$\mathsf{I}_{\mathtt{ewmul}}$	Tensor with 1 entries	

[#] Normal matrix multiplication is considered as batch size equals 1.
[%] Normal and depth-wise conv. are special cases of grouped conv.
[†] Increase the size of a conv. kernel, restricted to operate on input tensors.

3 Graph Substitution Verifier

The key idea behind our approach to formally verifying substitutions is to use a small set of *operator properties* expressed in first-order logic. These properties are manually written and reviewed, and are further validated by symbolically executing operators on tensors of small sizes and confirming that the operator properties are satisfied for these tensor sizes. Development of operator properties is guided by the substitutions discovered by the substitution generator.

For purposes of verification, we model tensor operators using first-order logic, where operators are represented using functions of both their parameters and their input tensors. For example $\mathtt{conv}(s, p, c, x, y)$ represents the convolution operator applied to tensors x and y, where the parameter s determines the stride, p determines padding mode, and c determines the activation mode, e.g., applying a `relu` activation function as part of the convolution operator kernel. For example, the fact that convolution without activation

Table 2. Operator properties used for verification. The operators are defined in Table 1, and the properties are grouped by the operators they involve. Logical variables w, x, y, and z are of type tensor, and variables a, c, k, p, and s are of type parameter. The variable a is used for the axis of concatenation and split, c for the activation mode of convolution, k for the kernel shape of pooling, p for the padding mode of convolution and pooling, and s for the strides of convolution and pooling.

Operator Property	Comment
$\forall x, y, z.\ \mathsf{ewadd}(x, \mathsf{ewadd}(y, z)) = \mathsf{ewadd}(\mathsf{ewadd}(x, y), z)$	ewadd is associative
$\forall x, y.\ \mathsf{ewadd}(x, y) = \mathsf{ewadd}(y, x)$	ewadd is commutative
$\forall x, y, z.\ \mathsf{ewmul}(x, \mathsf{ewmul}(y, z)) = \mathsf{ewmul}(\mathsf{ewmul}(x, y), z)$	ewmul is associative
$\forall x, y.\ \mathsf{ewmul}(x, y) = \mathsf{ewmul}(y, x)$	ewmul is commutative
$\forall x, y, z.\ \mathsf{ewmul}(\mathsf{ewadd}(x, y), z) = \mathsf{ewadd}(\mathsf{ewmul}(x, z), \mathsf{ewmul}(y, z))$	distributivity
$\forall x, y, w.\ \mathsf{smul}(\mathsf{smul}(x, y), w) = \mathsf{smul}(x, \mathsf{smul}(y, w))$	smul is associative
$\forall x, y, w.\ \mathsf{smul}(\mathsf{ewadd}(x, y), w) = \mathsf{ewadd}(\mathsf{smul}(x, w), \mathsf{smul}(y, w))$	distributivity
$\forall x, y, w.\ \mathsf{smul}(\mathsf{ewmul}(x, y), w) = \mathsf{ewmul}(x, \mathsf{smul}(y, w))$	operator commutativity
$\forall x.\ \mathsf{transpose}(\mathsf{transpose}(x)) = x$	transpose is its own inverse
$\forall x, y.\ \mathsf{transpose}(\mathsf{ewadd}(x, y)) = \mathsf{ewadd}(\mathsf{transpose}(x), \mathsf{transpose}(y))$	operator commutativity
$\forall x, y.\ \mathsf{transpose}(\mathsf{ewmul}(x, y)) = \mathsf{ewmul}(\mathsf{transpose}(x), \mathsf{transpose}(y))$	operator commutativity
$\forall x, w.\ \mathsf{smul}(\mathsf{transpose}(x), w) = \mathsf{transpose}(\mathsf{smul}(x, w))$	operator commutativity
$\forall x, y, z.\ \mathsf{matmul}(x, \mathsf{matmul}(y, z)) = \mathsf{matmul}(\mathsf{matmul}(x, y), z)$	matmul is associative
$\forall x, y, w.\ \mathsf{smul}(\mathsf{matmul}(x, y), w) = \mathsf{matmul}(x, \mathsf{smul}(y, w))$	matmul is linear
$\forall x, y, z.\ \mathsf{matmul}(x, \mathsf{ewadd}(y, z)) = \mathsf{ewadd}(\mathsf{matmul}(x, y), \mathsf{matmul}(x, z))$	matmul is linear
$\forall x, y.\ \mathsf{transpose}(\mathsf{matmul}(x, y)) = \mathsf{matmul}(\mathsf{transpose}(y), \mathsf{transpose}(x))$	matmul and transpose
$\forall s, p, c, x, y, w.\ \mathsf{conv}(s, p, c, \mathsf{smul}(x, w), y) = \mathsf{conv}(s, p, c, x, \mathsf{smul}(y, w))$	conv is bilinear
$\forall s, p, x, y, w.\ \mathsf{smul}(\mathsf{conv}(s, p, \mathsf{A}_{\mathsf{none}}, x, y), w) = \mathsf{conv}(s, p, \mathsf{A}_{\mathsf{none}}, \mathsf{smul}(x, w), y)$	conv is bilinear
$\forall s, p, x, y, z.\ \mathsf{conv}(s, p, \mathsf{A}_{\mathsf{none}}, x, \mathsf{ewadd}(y, z)) = \mathsf{ewadd}(\mathsf{conv}(s, p, \mathsf{A}_{\mathsf{none}}, x, y), \mathsf{conv}(s, p, \mathsf{A}_{\mathsf{none}}, x, z))$	conv is bilinear
$\forall s, p, x, y, z.\ \mathsf{conv}(s, p, \mathsf{A}_{\mathsf{none}}, \mathsf{ewadd}(x, y), z) = \mathsf{ewadd}(\mathsf{conv}(s, p, \mathsf{A}_{\mathsf{none}}, x, z), \mathsf{conv}(s, p, \mathsf{A}_{\mathsf{none}}, y, z))$	conv is bilinear
$\forall s, c, k, x, y.\ \mathsf{conv}(s, \mathsf{P}_{\mathsf{same}}, c, x, y) = \mathsf{conv}(s, \mathsf{P}_{\mathsf{same}}, c, x, \mathsf{enlarge}(k, y))$,	enlarge convolution kernel
$\forall s, p, x, y.\ \mathsf{conv}(s, p, \mathsf{A}_{\mathsf{relu}}, x, y) = \mathsf{relu}(\mathsf{conv}(s, p, \mathsf{A}_{\mathsf{none}}, x, y))$	conv with $\mathsf{A}_{\mathsf{relu}}$ applies relu
$\forall x.\ \mathsf{relu}(\mathsf{transpose}(x)) = \mathsf{transpose}(\mathsf{relu}(x))$	operator commutativity
$\forall s, p, x, k.\ \mathsf{conv}(s, p, \mathsf{A}_{\mathsf{none}}, x, \mathsf{C}_{\mathsf{pool}}(k)) = \mathsf{pool}_{\mathsf{avg}}(k, s, p, x)$	pooling by conv. with $\mathsf{C}_{\mathsf{pool}}$
$\forall k, x.\ \mathsf{conv}(1, \mathsf{P}_{\mathsf{same}}, \mathsf{A}_{\mathsf{none}}, x, \mathsf{I}_{\mathsf{conv}}(k)) = x$	identity kernel
$\forall x.\ \mathsf{matmul}(x, \mathsf{I}_{\mathsf{matmul}}) = x$	identity matrix
$\forall x.\ \mathsf{ewmul}(x, \mathsf{I}_{\mathsf{ewmul}}) = x$	ewmul identity
$\forall a, x, y.\ \mathsf{split}_0(a, \mathsf{concat}(a, x, y)) = x$	split definition
$\forall a, x, y.\ \mathsf{split}_1(a, \mathsf{concat}(a, x, y)) = y$	split definition
$\forall x, y, z, w.\ \mathsf{concat}(0, \mathsf{concat}(1, x, y), \mathsf{concat}(1, z, w)) = \mathsf{concat}(1, \mathsf{concat}(0, x, z), \mathsf{concat}(0, y, w))$	geometry of concatenation
$\forall a, x, y, w.\ \mathsf{concat}(a, \mathsf{smul}(x, w), \mathsf{smul}(y, w)) = \mathsf{smul}(\mathsf{concat}(a, x, y), w)$	operator commutativity
$\forall a, x, y, z, w.\ \mathsf{concat}(a, \mathsf{ewadd}(x, y), \mathsf{ewadd}(z, w)) = \mathsf{ewadd}(\mathsf{concat}(a, x, z), \mathsf{concat}(a, y, w))$	operator commutativity
$\forall a, x, y, z, w.\ \mathsf{concat}(a, \mathsf{ewmul}(x, y), \mathsf{ewmul}(z, w)) = \mathsf{ewmul}(\mathsf{concat}(a, x, z), \mathsf{concat}(a, y, w))$	operator commutativity
$\forall a, x, y.\ \mathsf{concat}(a, \mathsf{relu}(x), \mathsf{relu}(y)) = \mathsf{relu}(\mathsf{concat}(a, x, y))$	operator commutativity
$\forall x, y.\ \mathsf{concat}(1, \mathsf{transpose}(x), \mathsf{transpose}(y)) = \mathsf{transpose}(\mathsf{concat}(0, x, y))$	concatenation and transpose
$\forall x, y, z.\ \mathsf{concat}(1, \mathsf{matmul}(x, y), \mathsf{matmul}(x, z)) = \mathsf{matmul}(x, \mathsf{concat}(1, y, z))$	concatenation and matrix mul.
$\forall x, y, z, w.\ \mathsf{matmul}(\mathsf{concat}(1, x, z), \mathsf{concat}(0, y, w)) = \mathsf{ewadd}(\mathsf{matmul}(x, y), \mathsf{matmul}(z, w))$	concatenation and matrix mul.
$\forall s, p, c, x, y, z.\ \mathsf{concat}(0, \mathsf{conv}(s, p, c, x, z), \mathsf{conv}(s, p, c, y, z)) = \mathsf{conv}(s, p, c, \mathsf{concat}(0, x, y), z)$	concatenation and conv.
$\forall s, p, c, x, y, z.\ \mathsf{concat}(1, \mathsf{conv}(s, p, c, x, y), \mathsf{conv}(s, p, c, x, z)) = \mathsf{conv}(s, p, c, x, \mathsf{concat}(0, y, z))$	concatenation and conv.
$\forall s, p, x, y, z, w.\ \mathsf{conv}(s, p, \mathsf{A}_{\mathsf{none}}, \mathsf{concat}(1, x, z), \mathsf{concat}(1, y, w)) = \mathsf{ewadd}(\mathsf{conv}(s, p, \mathsf{A}_{\mathsf{none}}, x, y), \mathsf{conv}(s, p, \mathsf{A}_{\mathsf{none}}, z, w))$	concatenation and conv.
$\forall k, s, p, x, y.\ \mathsf{concat}(1, \mathsf{pool}_{\mathsf{avg}}(k, s, p, x), \mathsf{pool}_{\mathsf{avg}}(k, s, p, y)) = \mathsf{pool}_{\mathsf{avg}}(k, s, p, \mathsf{concat}(1, x, y))$	concatenation and pooling
$\forall k, s, p, x, y.\ \mathsf{concat}(0, \mathsf{pool}_{\mathsf{max}}(k, s, p, x), \mathsf{pool}_{\mathsf{max}}(k, s, p, y)) = \mathsf{pool}_{\mathsf{max}}(k, s, p, \mathsf{concat}(0, x, y))$	concatenation and pooling
$\forall k, s, p, x, y.\ \mathsf{concat}(1, \mathsf{pool}_{\mathsf{max}}(k, s, p, x), \mathsf{pool}_{\mathsf{max}}(k, s, p, y)) = \mathsf{pool}_{\mathsf{max}}(k, s, p, \mathsf{concat}(1, x, y))$	concatenation and pooling

(denoted by $\mathsf{A}_{\mathsf{none}}$) is linear in its first argument is captured by the following operator property (where ewadd represents element-wise tensor addition):

$$\forall s, p, x, y, z.\ \mathsf{conv}(s, p, \mathsf{A}_{\mathsf{none}}, \mathsf{ewadd}(x, y), z) = \mathsf{ewadd}(\mathsf{conv}(s, p, \mathsf{A}_{\mathsf{none}}, x, z), \mathsf{conv}(s, p, \mathsf{A}_{\mathsf{none}}, y, z))$$

Table 1 lists all operators and tensor constants used in our evaluation, and Table 2 shows the full list of operator properties used in our evaluation to verify graph substitutions.

Given the operator properties, we use a first-order theorem prover—our implementation uses Z3 [14]—to verify all

generated substitutions. This verification amounts to entailment checking in first-order logic, checking that the operator properties entail functional equivalence of the source and target graphs of each generated substitution.

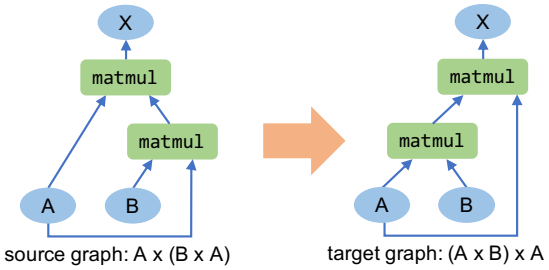
Modeling the operators using first-order logic involves a degree of abstraction (e.g., the shapes of tensors are not modeled). We found this level of abstraction to be suitable for verifying graph substitutions. We also note that the data layout is abstracted for verification purposes—layout does not affect operator semantics, and the optimizer (Section 5) ensures that layouts are used consistently.

Methodology for developing operator properties. We developed operator properties as needed to determine the correctness of generated graph substitutions using an iterative process. During the development process, we ran the substitution generator and tried to verify all discovered substitutions. If a substitution could not be verified and appeared correct, we added an appropriate property (or properties). To safeguard against mistakes in operator properties, we used further *validation* steps.

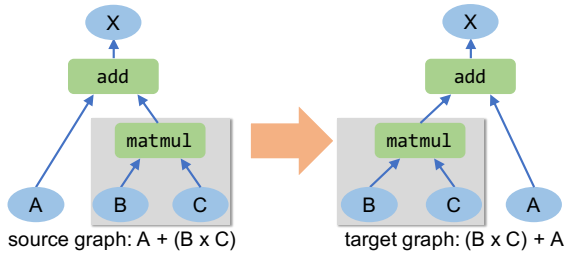
To validate operator properties, TASO verifies the operator properties themselves for all combinations of parameter values and tensor sizes up to a small bound—in our evaluation the bound was 4×4×4×4. For this, TASO requires a basic symbolic implementation of each tensor operator in Python. TASO symbolically executes this implementation for tensors of small size, effectively elaborating the tensor operations into symbolic real arithmetic expressions, where activation functions (e.g., `relu`) are modeled using uninterpreted functions. TASO then uses Z3, here as an SMT solver for the theory of real arithmetic, to verify the operator properties. For example, if a user would try to add the (wrong) property stating the convolution operator is linear for all activation modes (including `relu` activation), then this check would show that this property is not satisfied by the actual operators.

As an additional validation step that assists the development process, TASO checks that the set of operator properties is consistent and does not contain redundancies (i.e., a property entailed by other properties), which amounts to first-order entailment checks. These checks are also useful for discovering erroneous properties, and are cheaper to perform than the verification for small tensor sizes.

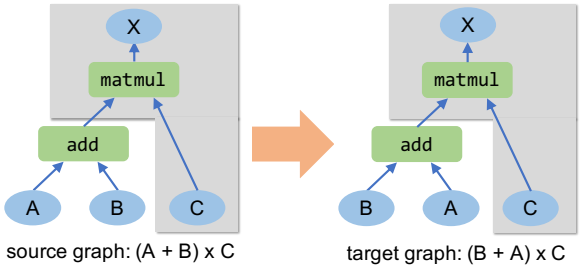
During our development process, the verification methodology revealed several subtle bugs. Some bugs in the graph substitution generator were found when it generated substitutions that could not be verified, and the validation steps described above revealed several bugs in candidate operator properties. In our experience, a new operator can be supported with a small amount of effort, usually a few hours of work by an expert. Typically a few properties must be written for each operator. In our evaluation, we were able to verify all 743 generated graph substitutions using 43 operator properties (see Table 2).



(a) A redundant substitution that is equivalent to Figure 2a by renaming input tensor C with A.



(b) A redundant substitution with a common subgraph.



(c) A redundant substitution with a common subgraph.

Figure 4. Example redundant substitutions pruned by TASO. `Matmul` and `Add` refer to matrix multiplication and element-wise addition, respectively. For each subgraph, A, B, and C refer to its input tensors, while X refers to the output tensor.

4 Pruning Redundant Substitutions

A graph substitution is redundant if it is subsumed by a more general valid substitution. This section describes the pruning techniques used by TASO to eliminate redundant graph substitutions. All pruning steps preserve *all* optimization opportunities: if graph $\mathcal{G}$ can be transformed into graph $\mathcal{G}'$ using a sequence of substitutions, then $\mathcal{G}$ can always be transformed into $\mathcal{G}'$ after pruning (possibly using a different set of transformations).

Input tensor renaming. TASO eliminates graph substitutions identical to other substitutions modulo input tensor

Table 3. The number of remaining graph substitutions after applying the pruning techniques in order.

Pruning Techniques	Remaining Substitutions	Reduction v.s. Initial
Initial	28744	1×
Input tensor renaming	17346	1.7×
Common subgraph	743	39×

renaming. For example, Figure 4a shows a redundant substitution equivalent to Figure 2a by renaming input tensor C with A. For substitutions that are equivalent through input tensor renaming, TASO prunes all but a single most general substitution.

Common subgraph. TASO also tries to eliminate substitutions whose source and target graphs have a common subgraph. TASO identifies two forms of common subgraphs that can lead to pruning.

The first form of common subgraph is illustrated in Figure 4b. Here, the source and target graphs both contain a common operator with the same input tensors (highlighted in gray boxes). The common subgraph represents an input to other operators in both the source and target graphs. Therefore, we can obtain a more general substitution by replacing the common subgraph with a fresh input tensor. If this more general substitution is indeed valid, then TASO prunes the less general substitution.

The second form of common subgraph is demonstrated in Figure 4c. Here, the common subgraph (highlighted in gray boxes) includes all the outputs in both the source and target graphs. In this case, a more general substitution can be obtained by completely removing the common subgraph, making its inputs new outputs of the source and target graphs. TASO prunes the less general substitution if the more general one is valid.

Table 3 shows the effect of the TASO pruning techniques on the number of substitutions. We observe that both pruning techniques play an important role in eliminating redundant substitutions and their combination reduces the number of substitutions TASO must consider by 39×.

5 Joint Optimizer

We now describe the TASO optimizer for jointly optimizing data layout and graph substitution. The optimizer uses the MetaFlow [21] cost-based backtracking search algorithm to search for an optimized computation graph by applying verified substitutions. TASO extends MetaFlow's search algorithm to also consider possible layout optimization opportunities when performing substitutions.

When applying a substitution on a matched subgraph, based on the data layouts of tensors in the source graph and the layouts supported by the operators, TASO enumerates possible layouts for tensors in the target graph. As a result, applying a substitution on a matched computation graph may result in multiple graphs with identical graph structure but different data layouts.

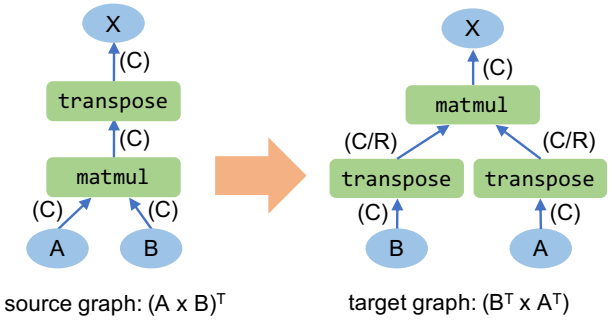


Figure 5. A graph substitution using the transpose of matrix multiplication. `matmul` and `transpose` indicate matrix multiplication and transpose, respectively. The parentheses show the potential layouts for each tensor in the source and target graphs, where `C` and `R` indicate the column-major and row-major layouts of a tensor.

Algorithm 2 Cost-Based Backtracking Search

Input: an input graph $\mathcal{G}_{in}$, verified substitutions $\mathcal{S}$, a cost model $Cost(\cdot)$, and a hyper parameter α.
Output: an optimized graph.

$\mathcal{P} = \{\mathcal{G}_{in}\}$ // $\mathcal{P}$ *is a priority queue sorted by Cost.*
while $\mathcal{P} \neq \{\}$ **do**
  $\mathcal{G} = \mathcal{P}$.dequeue()
  for substitution $s \in \mathcal{S}$ **do**
    // Layout$(\mathcal{G}, s)$ *returns possible layouts applying* s *on* $\mathcal{G}$.
    for layout $l \in$ Layout$(\mathcal{G}, s)$ **do**
      // Apply$(\mathcal{G}, s, l)$ *applies* s *on* $\mathcal{G}$ *with layout* l.
      $\mathcal{G}' =$ Apply$(\mathcal{G}, s, l)$
      if $\mathcal{G}'$ is valid **then**
        if $Cost(\mathcal{G}') < Cost(\mathcal{G}_{opt})$ **then**
          $\mathcal{G}_{opt} = \mathcal{G}'$
        if $Cost(\mathcal{G}') < \alpha \times Cost(\mathcal{G}_{opt})$ **then**
          $\mathcal{P}$.enqueue($\mathcal{G}'$)
return $\mathcal{G}_{opt}$

For example, Figure 5 shows the potential computation graphs that can be derived by applying the transpose of matrix multiplication on a source graph with a default column-major layout (shown as `C`). Both the matrix multiplication and transpose operators also support an alternative row-major layout (shown as `R`). The data layouts for all mapped tensors in the target graph (i.e., A, B, and X) must match the layouts in the source graph. The two intermediate tensors in the target graph can have either a row-major or a column-major layout, therefore TASO considers four different computation graphs (i.e., `CC`, `CR`, `RC`, and `RR` for the two intermediate tensors) when applying this substitution. This allows TASO to

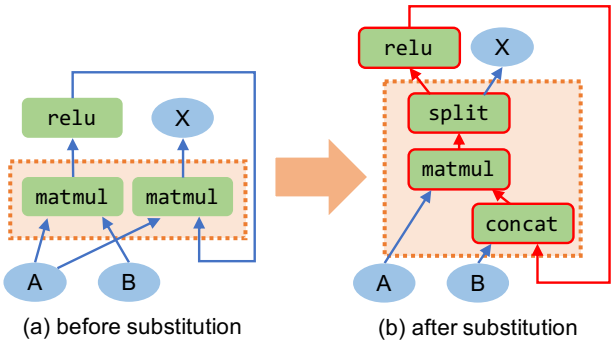


Figure 6. A graph substitution example that introduces a cycle into a computation graph, where A, B are the inputs, and X is the output. The original graph computes $A \times \texttt{relu}(A \times B)$, and the new graph is the result of applying the substitution shown in Figure 2b that fuses the two matrix multiplications using concatenation and split. The source and target graphs of the substitution are shown in the dotted boxes. Both the original graph and the substitution are acyclic. However, the resulting graph contains a cycle (highlighted in red).

capture potential layout transformation opportunities when performing graph substitutions.

Algorithm 2 shows our *cost-based backtracking search* algorithm for jointly optimizing substitution and data layout. The cost model is motivated by the fact that DNN operators perform dense linear algebra with no branches, and therefore their performance on hardware is highly consistent and predictable given the same data layouts and configuration parameters (e.g., the strides and padding of a convolution). Similar to MetaFlow [21], TASO measures the execution time of a DNN operator once for each configuration and data layout, and estimates the performance of a graph by summing up the measured execution time of its operators.

To search for an optimized graph, all candidate graphs are maintained in a priority queue $\mathcal{P}$ and are dequeued in increasing order of cost. For each dequeued graph $\mathcal{G}$, TASO considers each verified substitution and possible layouts applicable to the substitution, and applies them to obtain functionally equivalent new graphs $\mathcal{G}'$.

A non-obvious property of graph substitutions is that applying them can introduce cycles into a graph. Figure 6 shows one example where applying a valid substitution results in a cyclic graph. Since computation graphs must be acyclic, TASO checks the acyclicity of $\mathcal{G}'$ (line 12 of Algorithm 2) before enqueuing it in $\mathcal{P}$.

Finally, the best discovered graph $\mathcal{G}_{opt}$ is returned by the search algorithm. The search space is pruned by a hyper parameter α, which directly eliminates all graphs whose cost is α times worse than the best discovered graph. The parameter α trades off between the search time and the best discovered graph. Setting $\alpha = 1$ reduces the search to a simple greedy algorithm without backtracking, and a high value for α makes the search explore more possible candidates and causes more backtracking. We observe that $\alpha = 1.05$ achieves good performance in our evaluation.

6 Implementation

TASO is designed and implemented as a generic and extensible computation graph optimizer for tensor computations, such that new tensor operators can be easily added. Table 1 lists the tensor operators included in the current implementation of TASO. Some operators also depend on additional parameters to determine the behavior of the operator, such as the strides, padding, and activation of a convolution. In addition to operators, TASO also includes four types of constant tensors that are useful in substitutions. In particular, $\texttt{I}_{\texttt{ewmul}}$, $\texttt{I}_{\texttt{matmul}}$, and $\texttt{I}_{\texttt{conv}}$ are identity tensors for element-wise multiplication, matrix multiplication, and convolution, respectively. $\texttt{C}_{\texttt{pool}}$ allows converting an average pooling operator to a depth-wise convolution (see examples in Section 7.3).

As explained in Section 3, TASO uses operator properties specified by the user to verify the generated graph substitutions. Table 2 lists the 43 properties used to verify all substitutions in our evaluation.

TASO can easily be extended to include new tensor operators. For each operator, TASO requires two forms of input: (1) reference implementations for the operator, and (2) specifications of operator properties. (1) consists of a concrete implementation (in C++) used by the substitution generator and a symbolic implementation (in Python) used to validate the operator specifications. In our experience, adding a new operator requires a few hours of work by an expert.

For a new operator whose specifications are currently missing, TASO treats it as an opaque operator and can still optimize the rest of the graph using verified substitutions.

TASO is implemented on top of MetaFlow, and reuses the MetaFlow cost-based backtracking search [21]. Overall, our implementation of TASO contains around 8,000 lines of code for the core components (i.e., the substitution generator, verifier, and optimizer), and 1,400 lines of code for the operator reference implementations, including the 43 operator properties.

TASO is framework-agnostic and can be plugged in to existing DNN frameworks such as TensorRT and TVM by simply emitting the optimized graph in the target framework's input format. In the evaluation, we demonstrate this portability on TensorRT and TVM, and show that they can directly use TASO's optimizations to improve performance.

7 Evaluation

In this section we aim to evaluate the following points:

- Can TASO automatically generate and verify graph substitutions in acceptable run time?

- Can TASO improve the end-to-end performance of real-world DNN architectures, especially for emerging architectures with recently introduced operators?
- Can TASO's joint optimization of computation graphs and data layouts achieve better performance than separate optimizations?

7.1 Experimental Setup

DNNs. We use five real-world DNN architectures to evaluate TASO. ResNet-50 [18] is a widely used convolutional neural network for image classification and achieved the best classification performance in the ILSVRC [32] competition. ResNeXt-50 [38] improves the model accuracy and runtime efficiency of ResNet-50 by introducing a new grouped convolution operator. NasNet-A [40] and NasRNN [39] are two DNN architectures automatically discovered by machines through neural architecture search. NasNet-A and NasRNN exceed the best human-designed DNN architectures for image classification and language modeling tasks, respectively. Finally, BERT [15] is a new language representation architecture that obtained the state-of-the-art model accuracy on a spectrum of language tasks.

All experiments were performed on an Amazon p3.2xlarge instance [1] with an 8-core Intel E5-2600 CPU, 64 GB DRAM, and one NVIDIA Tesla V100 GPU.

To generate candidate graph substitutions, TASO enumerates all potential graphs with up to four operators by using all DNN operators listed in Table 1. TASO generated 743 candidate substitutions in around 5 minutes.

In the cost-based backtracking search for optimized DNN graphs, we set the hyperparameter α to be 1.05, which is identical to the value used in MetaFlow [21]. In all experiments, the end-to-end search time to discover an optimized computation graph is less than ten minutes.

7.2 End-to-End Evaluation

We first compare the end-to-end inference performance among TensorFlow [6], TensorFlow XLA [3], TensorRT [36], TVM [8], MetaFlow [21], and TASO on a V100 GPU. Figure 7 shows the results. TensorFlow, TensorFlow XLA, TensorRT, and MetaFlow use the highly-engineered cuDNN and cuBLAS libraries [10, 12] to perform DNN operators on GPUs, while TVM generates customized GPU kernels for the DNN operators. To eliminate the impact of different operator libraries, we evaluate the performance of TASO on both backends.

To generate GPU kernels in TVM, we allow the auto tuner [9] to run 2000 trials and use the best discovered configuration for each DNN operator. It takes 2 hours on average to tune a GPU kernel for each DNN operator. The TASO graph optimizer needs to query the execution time of hundreds of DNN operators for its cost model, therefore, for the TVM backend, we reuse the best discovered computation graph for the cuDNN backend, assuming the cost of an operator in cuDNN is a reasonable estimate for its cost in TVM.

Among the five DNN architectures, ResNet-50 has been commonly used and heavily optimized by existing DNN frameworks. TASO achieves on-par performance for ResNet-50 with existing frameworks, showing that TASO is able to automatically discover graph substitutions manually designed by domain experts. For the remaining four DNN architectures with new operators and graph structures, TASO outperforms existing DNN frameworks with speedups ranging from 1.3× to 2.8× on the cuDNN backend and 1.1× to 1.8× on the TVM backend. The speedup is achieved by (1) automatically discovering optimizing substitutions for the new operators and (2) jointly optimizing graph substitution and data layout. We analyze the substitutions discovered by TASO in Sections 7.3 and 7.4, and the joint optimization of substitution and data layout in Section 7.5.

7.3 Substitution Case Study

To understand how the substitutions generated and verified by TASO improve runtime performance, we study a few graph substitution examples in detail.

NasNet-A is the best discovered CNN architecture for the CIFAR-10 dataset, obtained by neural architecture search. Figure 8a shows a convolutional cell in NasNet-A. Unlike human-designed architectures, NasNet-A contains unconventional graph structures, making it hard to optimize with manual substitutions designed for more standard DNN architectures. To illustrate how TASO optimizes this architecture, we show two example substitutions discovered by TASO; neither is present in any existing DNN framework.

Figure 8b shows graph substitutions that transform two average pooling operators followed by element-wise addition to a single depth-wise convolution, by using a constant tensor $\mathsf{C}_{\mathsf{pool}}$ defined in Table 1. The mathematical formula for average pooling is:

$$o(n, c, x, y) = \frac{1}{K_X \times K_Y} \sum_{k_x} \sum_{k_y} i(n, c, x + k_x, y + k_y)$$

where, K_X and K_Y are the height and width of the pooling filter. Similarly, the formula for depth-wise convolution is:

$$o(n, c, x, y) = \sum_{k_x} \sum_{k_y} i(n, c, x + k_x, y + k_y) \times w(c, k_x, k_y)$$

which produces mathematically equivalent result as an average pooling if we have $w(c, k_x, k_y) = 1/(K_X \times K_Y)$. In addition, TASO also fuses the two depth-wise convolutions into one using its linearity.

A second new sequence of substitutions for NasNet-A is shown in Figure 8c, which fuses two depth-wise convolutions and two convolutions followed by addition to a depth-wise convolution followed by a standard convolution. This substitution increases the operator granularity and reduces the operator launch overhead by using larger operators.

For inference workloads, the weights in DNN architectures (e.g., W_i and $\mathsf{C}_{\mathsf{pool}}$ in Figure 8) are fixed and independent of

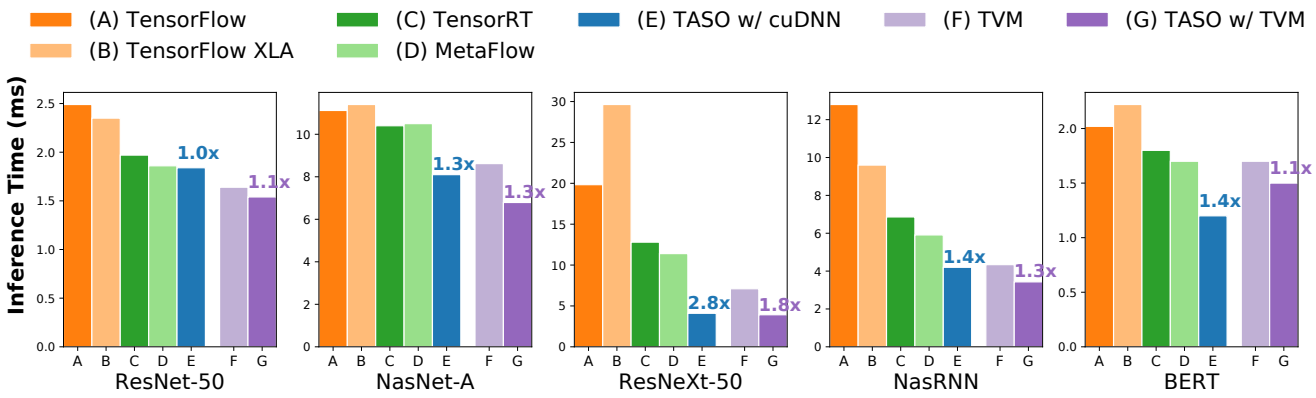
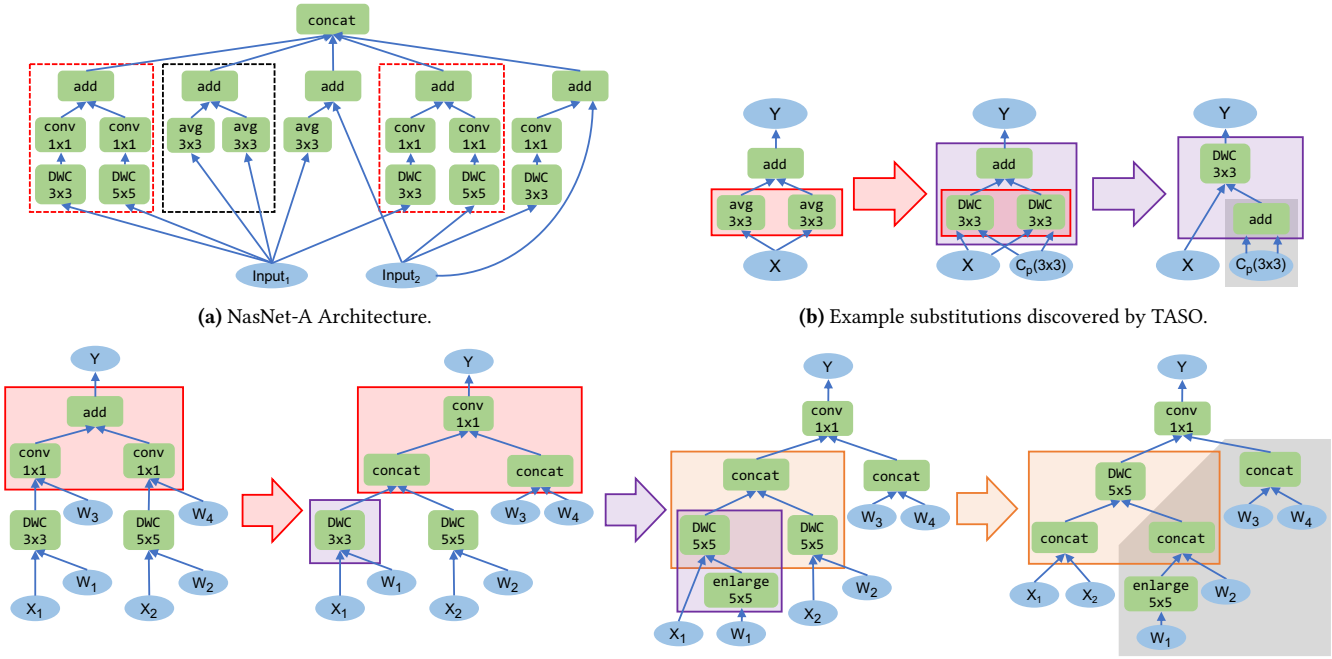


Figure 7. End-to-end inference performance comparison among existing DNN frameworks and TASO. The experiments were performed using a single inference sample, and all numbers were measured by averaging 1,000 runs on a NVIDIA V100 GPU. We evaluated the TASO's performance with both the cuDNN and TVM backends. For each DNN architecture, the numbers above the TASO bars show the speedup over the best existing approach with the same backend.



(a) NasNet-A Architecture.

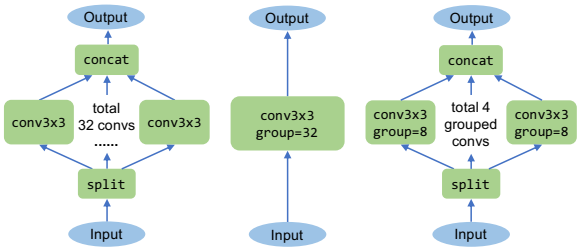
(b) Example substitutions discovered by TASO.

(c) A sequence of substitutions discovered by TASO.

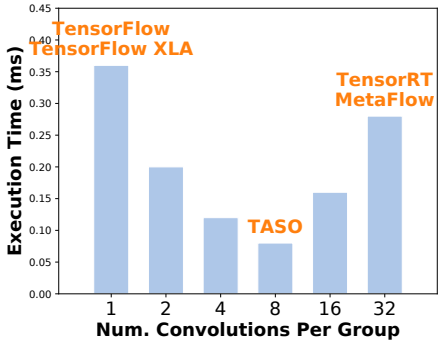
Figure 8. The NasNet-A architecture [40] and substitutions discovered by TASO to optimize NasNet-A. Figure 8a shows the architecture, where avg, conv, and DWC refer to average pooling, convolution, and depth-wise convolution, respectively. The weight tensors are eliminated for simplicity. Figures 8b and 8c shows two sequences of substitutions discovered by TASO that are used to optimize subgraphs marked in the black and red boxes in Figure 8a. In Figures 8b and 8c, each arrow refers to a substitution, and the subgraphs in the same color are the source and target graphs of the substitution. $C_{pool}(3 \times 3)$ in Figure 8b is a constant matrix whose entries are 1/9, as defined in Table 1. The enlarge operator in Figure 8c increases a convolution's kernel size by padding the weight (i.e., W_1) with extra 0's. For inference workloads, operators in the gray areas in Figures 8b and 8c only depend on pre-trained weights (i.e., W_i), and therefore can be pre-computed.

the inputs. TASO preprocesses operators whose inputs are all pre-trained weights (e.g., the gray areas in Figure 8) to further reduce the inference time.

ResNeXt-50 replaces large convolutions in ResNet-50 with multiple branches of much smaller convolutions to improve both model accuracy and runtime efficiency, as shown



(a) Multi-branch convolution. **(b)** Grouped convolution. **(c)** Multi-branch grouped convolution.



(d) Performance comparison.

Figure 9. Different approaches to perform multi-batch convolutions in ResNeXt-50 and their performance comparison. TensorFlow and TensorFlow XLA launch the 32 convolutions separately (Figure 9a). TensorRT and MetaFlow launch a single grouped convolution kernel that computes all 32 convolutions in parallel (Figure 9b). The best graph discovered by TASO uses 4 grouped convolutions, each of which computes 8 convolutions (Figure 9c).

in Figure 9a. However, directly launching these small convolutions incurs high kernel launch overhead. The cuDNN library has recently introduced grouped convolution kernels that perform multiple convolutions in parallel using a single CUDA kernel [10]. TensorFlow and TensorFlow XLA (r1.14 as of August 2019) currently do not support grouped convolution, so the fastest available ResNeXt-50 implementation in TensorFlow launches convolutions in multiple branches separately with the resulting high kernel launch overhead. TensorRT and MetaFlow use a single grouped convolution kernel that computes a group of 32 convolutions in parallel. While grouped convolution enables additional parallelism and reduces kernel launch overhead, it also requires a larger cache to save intermediate states for all convolutions, which results in decreased runtime performance when too many convolutions are aggregated in a single kernel. Figure 9d gives the time to run all 32 convolutions using different group sizes (i.e., the number of convolutions in a group), showing that neither launching individual convolutions nor grouping all 32 convolutions is the best option.

Existing frameworks either launch 32 individual convolutions or a single grouped convolution, both of which result in suboptimal performance. For ResNeXt-50, TASO uses a mixture of previous approaches and launches multiple grouped convolutions, as shown in Figure 9c. TASO discovered this mixture automatically, resulting in a speedup of 2.8× compared to the best existing approach.

7.4 Analysis of Used Substitutions

We now present a detailed analysis of how the graph substitutions discovered by TASO impact the performance of the optimized graphs. Figure 10 shows a heat map of the substitutions used to optimize each of the five DNN architectures. Each DNN uses 4-10 different substitutions to achieve optimized performance, and different DNNs require different sets of substitutions. This shows the difficulty of manually designing a few core substitutions to optimize today's DNN architectures with increasingly high diversity. TASO is better positioned for optimizing new DNNs by automatically discovering performance critical substitutions.

Additionally, we evaluate the scalability of TASO by considering substitutions with different size limitations, and measuring the runtime performance of the optimized graphs. Figure 11 shows the results. For all three DNN architectures, performance improvement is consistently achieved by using larger substitutions up to size 3. ResNeXt-50 and BERT do not obtain additional speedups by using substitutions with 4 operators, while NasNet-A achieves 1.2× by considering larger substitutions. Our current implementation of TASO does not scale to generate all substitutions with 5 or more operators, since the generator is limited by the memory needed to hold the fingerprints of all potential graphs, which scales exponentially with graph size. A distributed fingerprint generator could potentially handle graphs of size 5 and even more, which we leave as future work.

7.5 Joint Optimization of Graph Substitutions and Data Layout

To evaluate the performance of the joint optimization in TASO, we compare the joint optimization with three baseline strategies: (1) performing only graph substitution optimizations; (2) performing only data layout optimizations; and (3) performing the two optimizations sequentially.

Figure 12 shows the comparison results among the four strategies on BERT. TASO outperforms the three baseline strategies by 1.2-1.3×. We observe that the speedup is achieved by using graph substitutions that transform both graph structure and data layout. One example is depicted in Figure 5. The most time consuming operation in BERT is matrix multiplication $A \times B$, where A is 64 by 1024 and B is 1024 by 4096. In cuBLAS, the transposed version of this matrix multiplication (i.e., $(B^T \times A^T)^T$) achieves 1.5× speedup when B^T and A^T are in the column-major and row-major layout, respectively.

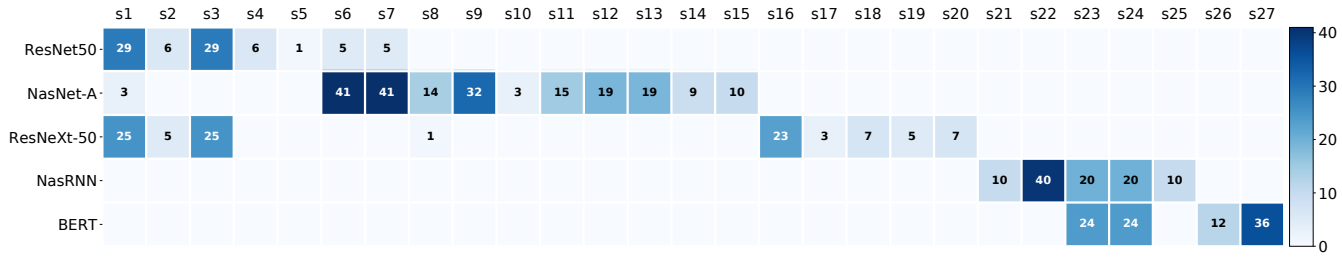


Figure 10. A heat map of how often the verified substitutions are used to optimize the five DNN architectures. Only substitutions used in at least one DNN are listed. For each architecture, the number indicates how many times a substitution is used by TASO to obtain the optimized graph.

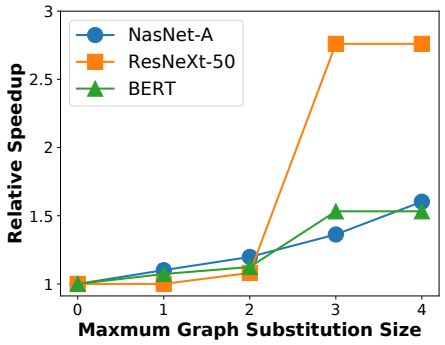


Figure 11. Performance comparison by using graph substitutions with different size limitations. The y-axis shows the relative speedups over the input computation graphs.

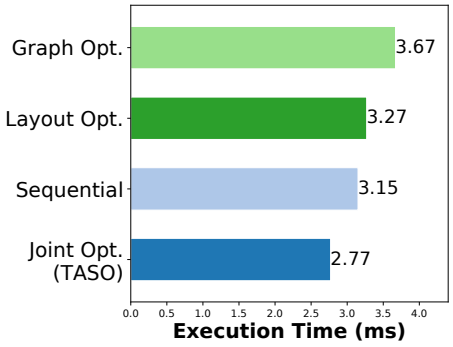


Figure 12. End-to-end inference performance comparison on BERT using different strategies to optimize graph substitution and data layout.

This graph optimization can only be captured when graph substitution and data layout are jointly considered.

7.6 Graph Substitution Verifier

We evaluate the performance of the graph substitution verifier for its two key tasks: verifying generated substitutions against operator specifications, and validating the operator specifications themselves to aid in the development process (Section 3). Our implementation uses Z3 [14] to automatically discharge all proof obligations, and our experiments were performed with Z3 version 4.8.5.

Generating the 743 graph substitutions takes around five minutes, and verifying them against the 43 specified operator properties takes less than 10 minutes. When checking the specification for redundancies we use Z3 to search for a proof of an invalid formula (stating that a specified property is entailed by the rest of the specification). This search can continue indefinitely, and in our evaluation we used a timeout of 10 seconds per query, resulting in a run time of less than 10 minutes (for 43 axioms). During the development process, when we had some redundant specifications they were discovered in a few seconds.

The validation check that verifies the operator specification for all combinations of parameter values and tensor sizes up to 4×4×4×4 is more computationally expensive, with roughly one million proof obligations. We parallelized it using 128 CPU cores, which resulted in a run time of roughly one hour. During the development process, we also found it useful to verify the operators for more restricted combinations. For example, verifying the specification for tensors of size *exactly* 4×4×4×4 (rather than *all* tensors up to that size) takes under 10 minutes using a single CPU core.

8 Related Work

Manually designed graph substitutions are used in existing DNN frameworks to optimize DNN architectures. For example, TensorFlow, TensorRT, and TVM use a rule-based strategy and directly perform all applicable substitutions on an input graph [6, 8, 36]. MetaFlow [21] allows users to define performance-decreasing substitutions to obtain a larger space of potential graphs. The key difference between TASO and these frameworks is that TASO can automatically generate candidate substitutions, and also provides semi-automatic support for verifying their correctness. In the evaluation, we also show that existing frameworks can directly use TASO's optimized graphs to improve performance.

Automated DNN code generation. Recent work has proposed various approaches to generate hardware-specific

code for DNN operators. For example, TVM [8, 9] uses a learning-based approach and automatically generates low-level optimized code for a diverse set of hardware backends. Astra [34] optimizes DNN computation by exploring the optimization space of multi-version complication during training. Compared to these approaches, TASO aims at optimizing DNN computation at a higher graph level, and therefore TASO's optimizations are orthogonal and can be combined with code generation techniques. It still remains an open problem of how to jointly optimize DNN computation at both graph-level and operator-level.

Automated DNN parallelization. ColocRL [26] uses reinforcement learning to automatically discover an efficient device placement for parallelizing DNN training across multiple GPUs. FlexFlow [20, 22] introduces a comprehensive search space of parallelization strategies for DNN training, and uses a randomized search algorithm to find efficient strategies in the search space. These frameworks optimize distributed DNN training assuming a fixed computation graph. We believe it is possible to combine TASO's graph optimizations with training parallelization techniques.

Superoptimization is a compiler optimization technique that was originally designed to find the optimal code for a sequence of instructions [25]. TASO's approach to identifying potential substitutions via enumeration of graphs and fingerprinting is similar to work in automatically generating peephole optimizers using superoptimization techniques [7]. TASO's approach to verification, however, is significantly different. Verification in superoptimization typically relies on "bit blasting", that is, modeling every bit in a computation explicitly in a logical formula (e.g., as a boolean variable). This approach is possible only when all aspects of a program transformation, including the computation and the data, can be expressed using a known number of bits. For TASO, where the input tensor sizes for graph substitutions are unknown, we must take a different approach. While not fully automatic like verification via bit blasting, our methodology based on writing operator specifications is much more flexible in being able to model future operators with almost arbitrary semantics, in addition to smoothly handling the issue of unknown tensor dimensions and split points.

Data layout optimizations. Existing DNN frameworks that support data layout optimizations treat data layouts and graph transformations as separate optimization problems [8, 24, 27]. TASO formulates the problem of performing graph substitutions and deciding the data layout of each DNN operator as a joint optimization problem and considers layout conversions as a part of graph substitutions. As a result, TASO can automatically generate graph substitutions that optimize both graph structures and data layouts, and our evaluation shows that jointly optimizing the two tasks can significantly improve the end-to-end performance, compared to optimizing the them separately.

9 Limitations and Future Work

One limitation of TASO is the reliance on user provided operator properties. While our experience has been that the required effort is manageable, it would be better to eliminate it altogether. One possible approach is to automatically verify substitutions directly against the implementations of the operators, e.g., cuDNN kernels.

Another limitation of TASO is the scalability of the generator, which requires saving the fingerprints of all computation graphs up to a fixed size. This approach currently does not scale beyond graphs of size 4. One possible approach to scale to larger graphs is to implement a distributed generator. A second possibility is to replace the brute-force enumeration with more efficient algorithms or heuristics.

An additional avenue for future research is combining graph-level and operator-level optimizations. This joint optimization is challenging as both problems involve large and complex search spaces, and optimizations at one level affect the search space of the other.

10 Conclusion

TASO is the first DNN computation graph optimizer that automatically generates graph substitutions. TASO formally verifies the substitutions, and considers graph substitutions and layout transformations together as a joint optimization problem, exploiting more optimization opportunities. TASO matches the performance of existing frameworks on DNNs for which these frameworks have been heavily optimized such as ResNet-50, and outperforms existing frameworks by up to 2.8× on other DNNs, finding novel optimizations not present in the hundreds of optimization rules in existing frameworks. TASO achieves these results with dramatically less human effort than existing frameworks, and provides a higher level of correctness guarantees.

Acknowledgments

We thank Nikolaj Bjørner, Mingyu Gao, Vinod Grover, Sina Lin, Feng Ruan, Xi Wang, the anonymous SOSP reviewers, and our shepherd, Joey Gonzalez, for their helpful feedback. This work was supported by NSF grant CCF-1409813, the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, and is based on research sponsored by DARPA under agreement number FA84750-14-2-0006. This research was supported in part by affiliate members and other supporters of the Stanford DAWN project—Ant Financial, Facebook, Google, Infosys, Intel, Microsoft, NEC, SAP, Teradata, and VMware—as well as Cisco and the NSF under CAREER grant CNS-1651570. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] 2017. Amazon EC2 P3 Instances. https://aws.amazon.com/ec2/instance-types/p3/.

[2] 2017. Tensorflow graph transform creates corrupted graph. https://github.com/tensorflow/tensorflow/issues/7523.

[3] 2017. XLA: Optimizing Compiler for TensorFlow. https://www.tensorflow.org/xla.

[4] 2018. Graph transform: fold constant with invalid graph. https://github.com/tensorflow/tensorflow/issues/16545.

[5] 2018. Tensor Cores in NVIDIA Volta Architecture. https://www.nvidia.com/en-us/data-center/tensorcore/.

[6] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning.. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*.

[7] Sorav Bansal and Alex Aiken. 2006. Automatic Generation of Peephole Superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*.

[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: End-to-End Optimization Stack for Deep Learning. *CoRR* abs/1802.04799 (2018). http://arxiv.org/abs/1802.04799

[9] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to Optimize Tensor Programs. In *Advances in Neural Information Processing Systems 31*.

[10] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *CoRR* abs/1410.0759 (2014). http://arxiv.org/abs/1410.0759

[11] Berkeley R. Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic Program Alignment for Equivalence Checking. In *Proceedings of the 2019 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Phoenix, AZ, USA, June 22-26, 2019*. https://doi.org/10.1145/3314221.3314596

[12] cuBLAS 2016. Dense Linear Algebra on GPUs. https://developer.nvidia.com/cublas.

[13] Manjeet Dahiya and Sorav Bansal. 2017. Black-Box Equivalence Checking Across Compiler Optimizations. In *Programming Languages and Systems*, Bor-Yuh Evan Chang (Ed.). Springer International Publishing, Cham.

[14] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*.

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018).

[16] Vincent Dumoulin and Francesco Visin. 2016. A guide to convolution arithmetic for deep learning. *CoRR* (2016).

[17] Sumit Gulwani and George C. Necula. 2003. Discovering Affine Equalities Using Random Interpretation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*. ACM.

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

[19] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* abs/1704.04861 (2017).

[20] Zhihao Jia, Sina Lin, Charles R. Qi, and Alex Aiken. 2018. Exploring Hidden Dimensions in Accelerating Convolutional Neural Networks. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Vol. 80. PMLR.

[21] Zhihao Jia, James Thomas, Todd Warzawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2019. Optimizing DNN Computation with Relaxed Graph Substitutions. In *Proceedings of the 2nd Conference on Systems and Machine Learning (SysML'19)*.

[22] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. In *Proceedings of the 2nd Conference on Systems and Machine Learning (SysML'19)*.

[23] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 216–226. https://doi.org/10.1145/2594291.2594334

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

[25] Henry Massalin. 1987. Superoptimizer: a look at the smallest program. In *ACM SIGARCH Computer Architecture News*, Vol. 15.

[26] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. 2017. Device Placement Optimization with Reinforcement Learning. (2017).

[27] MKLDNN 2016. Intel Math Kernel Library for Deep Neural Networks. https://01.org/mkl-dnn.

[28] Vinod Nair and Geoffrey E. Hinton. 2010. Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning (ICML'10)*. Omnipress, USA, 807–814. http://dl.acm.org/citation.cfm?id=3104322.3104425

[29] George C. Necula. 2000. Translation validation for an optimizing compiler. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*. 83–94. https://doi.org/10.1145/349299.349314

[30] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation. In *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*. 151–166. https://doi.org/10.1007/BFb0054170

[31] PyTorch 2017. Tensors and Dynamic neural networks in Python with strong GPU acceleration. https://pytorch.org.

[32] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[33] Rahul Sharma, Eric Schkufza, Berkeley R. Churchill, and Alex Aiken. 2013. Data-driven equivalence checking. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. 391–406. https://doi.org/10.1145/2509136.2509509

[34] Muthian Sivathanu, Tapan Chugh, Sanjay S. Singapuram, and Lidong Zhou. 2019. Astra: Exploiting Predictability to Optimize Deep Learning.

In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, New York, NY, USA.

[35] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2011. Equality Saturation: A New Approach to Optimization. *Logical Methods in Computer Science* 7, 1 (2011). https://doi.org/10.2168/LMCS-7(1:10)2011

[36] TensorRT 2017. NVIDIA TensorRT: Programmable Inference Accelerator. https://developer.nvidia.com/tensorrt.

[37] Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. 2018. Training and Inference with Integers in Deep Neural Networks. In *International Conference on Learning Representations*.

[38] Saining Xie, Ross B. Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. 2016. Aggregated Residual Transformations for Deep Neural Networks. *CoRR* abs/1611.05431 (2016).

[39] Barret Zoph and Quoc V. Le. 2016. Neural Architecture Search with Reinforcement Learning. *CoRR* abs/1611.01578 (2016).

[40] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. 2018. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*.