ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

PET: Optimizing Tensor Programs with Partially Equivalent Transformations and Automated Corrections

Haojie Wang Jidong Zhai Mingyu Gao Zixuan Ma Shizhi Tang
Liyan Zheng Yuanzhi Li[†] Kaiyuan Rong Yuanyong Chen Zhihao Jia[†‡]
Tsinghua University *Carnegie Mellon University*[†] *Facebook*[‡]

Abstract

High-performance tensor programs are critical for efficiently deploying deep neural network (DNN) models in real-world tasks. Existing frameworks optimize tensor programs by applying fully equivalent transformations, which maintain equivalence on every element of output tensors. This approach misses possible optimization opportunities as transformations that only preserve equivalence on subsets of the output tensors are excluded.

We propose PET, the first DNN framework that optimizes tensor programs with partially equivalent transformations and automated corrections. PET discovers and applies program transformations that improve computation efficiency but only maintain partial functional equivalence. PET then automatically corrects results to restore full equivalence. We develop rigorous theoretical foundations to simplify equivalence examination and correction for partially equivalent transformations, and design an efficient search algorithm to quickly discover highly optimized programs by combining fully and partially equivalent optimizations at the tensor, operator, and graph levels. Our evaluation shows that PET outperforms existing systems by up to 2.5×, by unlocking previously missed opportunities from partially equivalent transformations.

1 Introduction

Existing deep neural network (DNN) frameworks represent DNN computations as *tensor programs*, which are direct acyclic computation graphs describing the operations applied to a set of tensors (i.e., *n*-dimensional arrays). The operators in tensor programs are mostly linear algebra computations such as matrix multiplication and convolution. Although tensor programs are specified based on the high-level insights of today's DNN algorithms, such constructions do not necessarily offer the best runtime performance. Current practice to optimize tensor programs in existing DNN frameworks is to leverage *program transformations*, each of which identifies a subprogram that matches a specific pattern and replaces it with another subprogram that offers improved performance.

To preserve the statistical behavior of DNN models, existing frameworks only consider *fully equivalent* program transformations, where the new subprogram is mathematically equivalent to the original subprogram for *arbitrary* inputs. For example, TensorFlow, PyTorch, TensorRT, TVM, and Ansor all use rule-based optimization strategies that directly apply manually designed program transformations whenever applicable [3, 6, 26, 32, 34]. TASO automatically generates and verifies transformations by taking operator specifications as inputs, but is still limited to fully equivalent transformations [15].

Despite the wide use of equivalent program transformations in conventional compilers and modern DNN frameworks, they only exhibit limited opportunities for performance optimization, especially for tensor programs. Unlike traditional programs whose primitives are scalars or simple arrays of scalars, tensor programs operate on high-dimensional tensors with up to millions of elements. Many transformations can improve the runtime performance of a tensor program but do not preserve full equivalence on *all* elements of the output tensors. We call such transformations *partially equivalent*. Examples of performance-optimizing partially equivalent transformations include (1) changing the shape or linearization ordering of input tensors to improve computational efficiency, (2) replacing less efficient operators with more optimized operators with similar mathematical behavior, and (3) transforming the graph structure of a program to enable subsequent performance optimizations.

Partially equivalent transformations, despite their high potential, are not exploited in existing DNN frameworks due to several challenges. First, directly applying partially equivalent transformations would violate the functional equivalence to an input program and potentially decrease the model accuracy. It is necessary to correct any non-equivalent regions of output tensors, to preserve transparency to higher-level algorithms. However, quickly *examining equivalence* to identify these regions and effectively generating the required *correction kernels* are difficult tasks. Second, when partially equivalent transformations are applied, the design space is substantially enlarged compared to existing frameworks under equivalence constraint. Theoretically, any program transformation, regardless of how different the result is from the original one, becomes a potential candidate. The *generation algorithm* for partially equivalent transformations should carefully manage its computational complexity. The *optimizer* must balance the benefits and overhead and be able to combine fully and partially equivalent transformations to obtain performant tensor programs.

In this paper, we explore a radically different approach to optimize tensor programs, by exploiting partially equivalent transformations. We develop rigorous theorems that simplify equivalence examination and correction kernel generation, allowing us to easily restore functional equivalence and provably preserve the DNN models' statistical behavior. With a significantly larger search space of program optimizations that includes both fully and partially equivalent transformations, our approach can discover highly optimized tensor programs that existing approaches miss. Based on these techniques, we propose PET, the first DNN framework that optimizes tensor programs with partially equivalent transformations and automated corrections. PET consists of three main components:

Mutation generator. To discover partially equivalent transformations automatically for an input subprogram, PET uses a *mutation generator* to construct potential program *mutants*. Each mutant takes the same input tensors as in the original subprogram and produces output tensors with the same shapes. This ensures that a mutant can replace the input subprogram and therefore constitutes a potential transformation.

Mutation corrector. The generated mutants of an input subprogram may produce different results on some regions of the output tensors, thus affecting the model accuracy. To preserve its statistical behavior, PET's *mutation corrector* examines the equivalence between an input subprogram and its mutant and automatically generates *correction kernels*. These are subsequently applied to the output tensors to maintain an end-to-end equivalence to the input subprogram. To reduce the overhead and heterogeneity introduced by the correction kernels, PET opportunistically *fuses* the correction kernels with other tensor computation kernels.

Examining and correcting a partially equivalent transformation is difficult, since the output tensors of a program include up to millions of elements, and each one must be verified against a large number of input elements. A key contribution of PET is a set of rigorous theoretical foundations that significantly simplify this verification process. Rather than examining program equivalence for *all* positions in the output tensors, PET needs to test only a few representative positions.

Program optimizer. PET uses a *program optimizer* to identify mutant candidates with high performance, by effectively balancing the benefits from using better mutants and the overheads of extra correction kernels. We first split an arbitrarily large input program into multiple small subprograms at the positions of non-linear operators. Each subprogram then contains only linear operators and can be independently mutated. We support mutations on various subsets of operators in the subprogram, and can iteratively apply mutations to obtain mutants that are more complex. Finally, we apply a series of post-optimizations across subprogram boundaries, including redundancy elimination and operator fusion.

We evaluate PET on five real-world DNN models. Even for common and heavily optimized models in existing frameworks such as Resnet-18 [14], PET can still improve the performance by 1.2×. For new models such as CSRNet [20] and BERT [12], PET is up to 2.5× faster than the state-of-the-art frameworks. The significant performance improvement is enabled by combining fully and partially equivalent transformations at the tensor, operator, and graph levels.

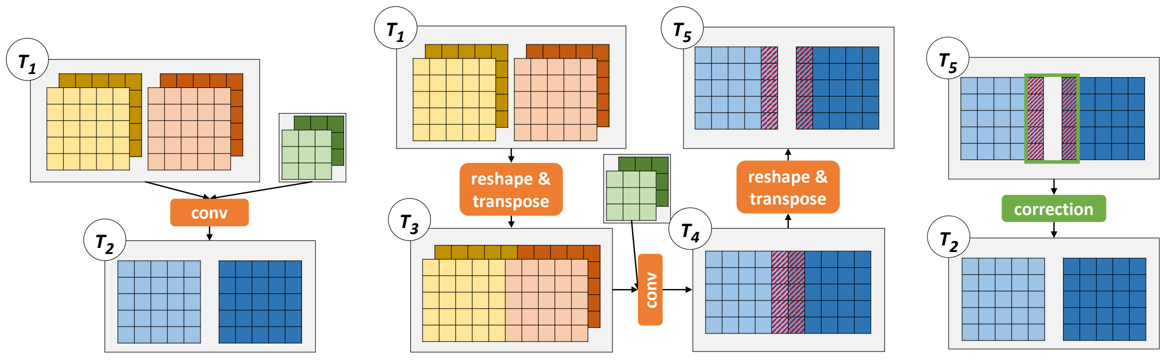
This paper makes the following contributions.

- We present the first attempt in tensor program optimization to exploit partially equivalent transformations with automated corrections. We explore a significantly larger search space than existing DNN frameworks.
- We develop rigorous theoretical foundations that simplify the equivalence examination and correction kernel generation, making it practical to preserve statistical behavior even with partially equivalent transformations.
- We propose efficient generation and optimization approaches to explore the large design space automatically with both fully and partially equivalent transformations.
- We implement the above techniques into an end-to-end framework, PET, and achieve up to 2.5× speedup compared to state-of-the-art frameworks.

2 Background and Motivation

To generate high-performance tensor programs, a common form of optimization in existing DNN frameworks (e.g., TensorFlow [3], TensorRT [32], and TVM [6]) is fully equivalent transformations that improve the performance of a tensor program while preserving its mathematical equivalence. Examples of current fully equivalent transformations include operator fusion [2, 6], layout transformations [18], and automated generation of graph substitutions [15]. Though effective at improving performance, fully equivalent transformations explore only a limited space of program optimizations.

In contrast, Figure 1 shows an example of a partially equivalent transformation for a convolution operator. It concatenates two individual images into a larger one along the width dimension to improve performance. This is because a larger width, which is typically the innermost dimension for convolution on modern accelerators like GPUs, provides more parallelism and improves computation locality. However, the new program after this transformation (shown in Figure 1(b)) produces different results on a sub-region of the output tensor along the boundary of the concatenation (shown as the shaded boxes in Figure 1(b)), resulting in partial non-equivalence.



(a) Input program. (b) A partially equivalent transformation. (c) Correcting results.

Figure 1: A partially equivalent transformation that improves the performance of convolution by manipulating tensor shape and linearization. The shaded boxes in (b) highlight non-equivalent elements between two programs in the transformation. The correction kernel in (c) is applied to these elements to recover the functional equivalence of the input program.

In addition to the above example that optimizes a tensor program by changing the shape and linearization of its tensors, partially equivalent transformations also include replacing less efficient operators with more optimized ones with similar semantics, and modifying the graph structure of a tensor program to enable additional optimizations. We provide more such examples in §4.2 and evaluate them in §8.3.

Although partially equivalent transformations exhibit high potential for performance improvement, they are not considered in current DNN frameworks due to their possible impact on model accuracy. Manually implementing such partially equivalent transformations is prohibitive. First, it requires evaluating a large amount of potential partially equivalent transformations to discover promising ones. Second, to apply partially equivalent transformations while preserving model accuracy, we need correction kernels to fix the results for non-equivalent parts (see Figure 1(c)). Overall, more *automated* approaches are needed to discover performance-optimizing partially equivalent transformations and correct the results, which are the main focus of this work.

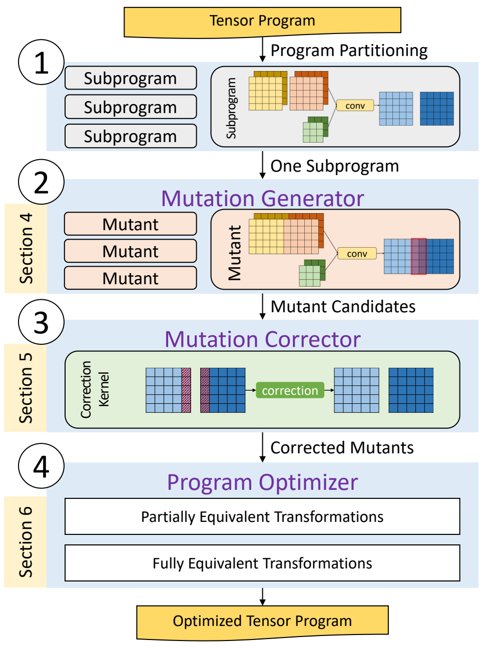


Figure 2: PET overview.

3 Design Overview

PET is the first framework to optimize tensor programs by exploiting partially equivalent transformations and correcting their results automatically. To realize this, PET leverages the multi-linearity of tensor programs.

Multi-linear tensor programs (MLTPs). We first define multi-linear tensor operators. An operator *op* with *n* input tensors $I_1,\dots,I_n$ is *multi-linear* if *op* is linear to all inputs I_k:

$$op(I_1,\dots,I_{k-1},X,\dots,I_n)+op(I_1,\dots,I_{k-1},Y,\dots,I_n)$$
$$=op(I_1,\dots,I_{k-1},X+Y,\dots,I_n)$$
$$\alpha\cdot op(I_1,\dots,I_{k-1},X,\dots,I_n)=op(I_1,\dots,I_{k-1},\alpha\cdot X,\dots,I_n)$$

where X and Y are arbitrary tensors with the same shape as I_k, and α is an arbitrary scalar. DNN computation generally consists of multi-linear tensor operators (e.g., matrix multiplication, convolution) and element-wise non-linear operators (e.g., ReLU [23] and sigmoid). The linear operators consume the majority of the computation time, due to their high computational complexity. A program $\mathcal{P}$ is a *multi-linear tensor program* (MLTP) if all operators $op \in \mathcal{P}$ are multi-linear.

PET overview. Figure 2 shows an overview of PET. The input to PET is a tensor program to be optimized. Similar to prior work [6,34], PET first splits an input program into smaller subprograms to reduce the exploration space of each subprogram without sacrificing performance improvement opportunities. For each subprogram, PET's *mutation generator* discovers partially equivalent transformations by generating possible *mutants* for MLTPs in the subprogram. Each mutant has the

Table 1: Multi-linear tensor operators used in PET.

Operator	Description
add	Element-wise addition
mul	Element-wise multiplication
conv	Convolution
groupconv	Grouped convolution
dilatedconv	Dilated convolution
batchnorm	Batch normalization
avgpool	Average pooling
matmul	Matrix multiplication
batchmatmul	Batch matrix multiplication
concat	Concatenate multiple tensors
split	Split a tensor into multiple tensors
transpose	Transpose a tensor's dimensions
reshape	Decouple/combine a tensor's dimensions

same input and output shapes as the original MLTPs, thus constitutes a partially equivalent transformation (§4).

To maintain the end-to-end equivalence to an input program, PET's *mutation corrector* examines the equivalence between a mutant and its original MLTP, and automatically generates *correction kernels* to fix the outputs of the mutant. PET leverages rigorous theoretical foundations to simplify such challenging tasks (§5).

The corrected mutants are sent to PET's *program optimizer*, which combines existing fully equivalent transformations with partially equivalent ones to construct a comprehensive search space of program optimizations. The optimizer evaluates a rich set of mutants for each subprogram and applies post-optimizations across their boundaries, in order to discover highly optimized candidates in the search space (§6).

4 Mutation Generator

This section describes the mutation generator in PET, which takes an MLTP as input and automatically generates possible *mutants* to replace the input MLTP. The generation algorithm discovers valid mutants up to a certain size. Each generated mutant does not necessarily preserve mathematical equivalence to the input program on the entire output tensors. To restore functional equivalence, the mutation corrector (§5) automatically generates correction kernels.

4.1 Mutation Generation Algorithm

We call an MLTP $\mathcal{P}_1$ a *mutant* of another MLTP $\mathcal{P}_0$ if $\mathcal{P}_1$ and $\mathcal{P}_0$ have the same number of inputs (and outputs) and each input (and output) has the same shape. The computations of $\mathcal{P}_0$ and $\mathcal{P}_1$ are not necessarily equivalent. Intuitively, if $\mathcal{P}_0$ is a subprogram in a tensor program, then replacing $\mathcal{P}_0$ with $\mathcal{P}_1$ yields a valid but potentially non-equivalent tensor program.

For a given MLTP $\mathcal{P}_0$, PET generates potential mutants of $\mathcal{P}_0$ using a given set of multi-linear operators O as the basic building blocks. Table 1 lists the operators used in our evaluation. The list covers a variety of commonly used tensor operators, including compute-intensive operators (conv, matmul, etc.), element-wise operators (add, mul, etc.), and tensor manipulation (split, transpose, etc.). This set can also be extended to include new DNN operators.

Algorithm 1 MLTP mutation generation algorithm.

```
1: Input: A set of operators $O$; an input MLTP $\mathcal{P}_0$
2: Output: A set of valid program mutants $\mathcal{M}$ for $\mathcal{P}_0$
3: $I_0$ = the set of input tensors in $\mathcal{P}_0$
4: $\mathcal{M} = \varnothing$
5: BUILD(1, $\varnothing$, $I_0$)
6: // Depth-first search to construct mutants
7: function BUILD($n$, $\mathcal{P}$, $I$)
8:     if $\mathcal{P}$ and $\mathcal{P}_0$ have the same input/output shapes then
9:         $\mathcal{M} = \mathcal{M} + \{\mathcal{P}\}$
10:    if $n < depth$ then
11:        for $op \in O$ do
12:            for $i \in I$ and $i$ is a valid input to $op$ do
13:                Add operator $op$ into program $\mathcal{P}$
14:                Add the output tensors of $op$ into $I$
15:                BUILD($n+1$, $\mathcal{P}$, $I$)
16:                Remove operator $op$ from $\mathcal{P}$
17:                Remove the output tensors of $op$ from $I$
18: return $\mathcal{M}$
```

Algorithm 1 shows a *depth-first search* algorithm for constructing potential mutants of an MLTP $\mathcal{P}_0$. PET starts from an empty program with no operator and only the set of original input tensors to $\mathcal{P}_0$. PET iteratively adds a new operator to the current program $\mathcal{P}$ by enumerating the type of operator from O and the input tensors to the operator. The input tensors can be the initial input tensors to $\mathcal{P}_0$ (i.e., I_0 in Algorithm 1) or the output tensors of previous operators. The depth-first search algorithm enumerates all potential MLTPs up to a certain size (called the mutation *depth*). For each mutant $\mathcal{P}$, PET checks whether $\mathcal{P}$ and $\mathcal{P}_0$ have the same number and shapes of inputs/outputs. $\mathcal{P}$ is a valid mutant if it passes this test.

4.2 Example Mutant Categories

While the above mutation generation algorithm is general enough to explore a sufficiently large design space, we emphasize that several mutant categories are of particular importance to PET and lead to mutants with improved performance. Note that PET does not rely on manually specified categories. Rather, these categories are discovered by PET automatically.

Reshape and transpose. It is widely known that the in-memory *layouts* of tensors play an important role in optimizing tensor programs [6]. PET leverages the reshape and transpose operators to transform the shapes of input tensors and the linearization ordering of tensor dimensions to generate mutants with better performance. A reshape operator changes the shape of a tensor by decoupling a single dimension into multiple ones or combining multiple dimensions into one. E.g., a `reshape` can transform a vector with four elements into a 2×2 matrix. A `transpose` operator modifies the linearization ordering of a tensor's dimensions, such as converting a row-major matrix to a column-major one.

`Reshape` and `transpose` are generally applied jointly to transform the tensor layouts. For example, Figure 1 shows a potential mutant of a convolution operator that concatenates two separate images (i.e., $T_1 \rightarrow T_3$ in Figure 1(b)) along the width dimension to improve the performance of convolution: typically a larger width exhibits more parallelism to be exploited on modern accelerators such as GPUs. This concatenation involves a combination of three `reshape` and `transpose` operators. First, a `reshape` operator splits the batch dimension of T_0 into an inner dimension that groups every two consecutive images, and an outer dimension that is half the size of the original. Then, a `transpose` operator moves the newly created inner dimension next to the width dimension and updates the tensor's linearization ordering accordingly, so each row of the two images in the same group is stored consecutively in memory. Finally, another `reshape` operator combines the two images.

The mutation generator usually *fuses* multiple consecutive `reshape` and `transpose` operators into a single compound operator, namely `reshape & transpose`. This fusion reduces the size of the generated mutants and allows for exploring much larger and more sophisticated mutants.

Single-operator mutants. PET can also generate mutants that replace an inefficient operator in a tensor program with a different and more performant operator. Several standard tensor operators, such as convolution and matrix multiplication, have been extensively optimized either manually or automatically on modern hardware backends. In contrast, their variants, such as strided or dilated convolutions [20], are not as efficiently supported. There are performance-related benefits to mutating them into their standard counterparts with highly optimized kernels. As an example, Figure 3 shows a mutant that transforms a dilated convolution into a regular convolution by reorganizing the linearization ordering of the input tensor based on the given dilation. However, the mutant is not fully equivalent to the input program and requires corrections afterward to restore functional equivalence.

Multi-operator mutants. PET also supports substituting a subgraph of multiple operators with another more efficient set of operators. For example, a few independent convolutions with similar tensor shapes may be combined into a single larger convolution to improve GPU utilization and reduce kernel launch overhead. This requires manipulating tensor shapes and adding proper padding (see the examples in §8.3.3).

5 Mutation Corrector

While the mutants generated by PET have potentially higher performance than the original programs, they may produce different mathematical results on some regions of the output tensors, potentially leading to accuracy loss. To maintain transparency at the application level, PET chooses to preserve the statistical behavior of the input program and guarantees the same model accuracy, with the help of a *mutation corrector*. Specifically, the mutation corrector takes as inputs an MLTP $\mathcal{P}_0$ and one of its mutants $\mathcal{P}$, and automatically generates *correction kernels* that are applied to the output tensors of $\mathcal{P}$ to maintain functional equivalence to $\mathcal{P}_0$.

The goal of the mutation corrector is twofold. First, for any given MLTP and its mutant, the corrector analyzes the two programs and identifies all the regions of the output tensors on which the two programs provide identical results and therefore do not need any correction. Second, for the remaining regions where the two outputs are different, the corrector automatically generates kernels to fix the output of the mutant and preserve functional equivalence.

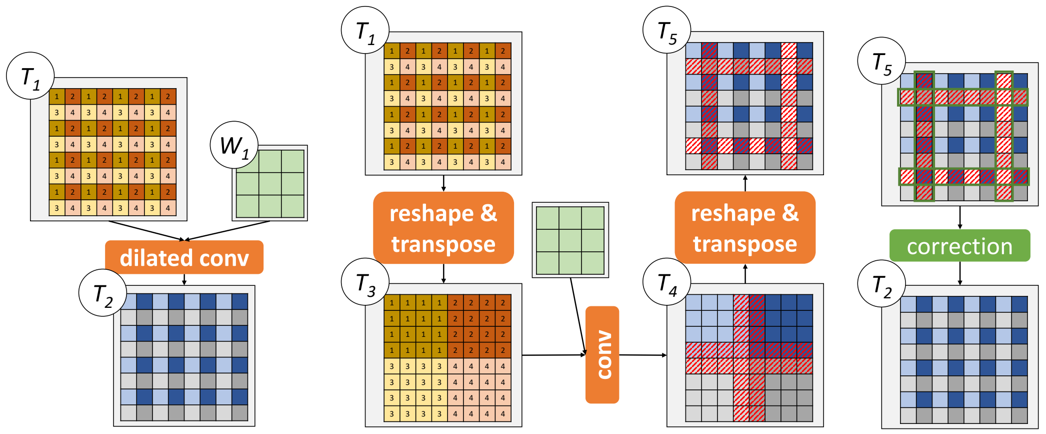
Designing the mutation corrector requires addressing two challenges. First, the output tensors may be very large, involving up to many millions of elements that all require equivalence verification. It is infeasible to verify *every* single element of the output tensors individually. Second, the verification of each output element may depend on a large number of input variables in many tensor operators. For example, each output element of a matrix multiplication is the inner product of one row and one column of the two input matrices, both with sizes up to several thousand. Numerically enumerating all possible values for this many input variables is impractical.

Two theorems that significantly simplify the verification tasks are central to the PET mutation corrector. Rather than verifying all output positions with respect to all input value combinations, PET only needs to verify a few representative output positions with a small number of randomly generated input values. This dramatically reduces the verification workload. We describe these theoretical foundations in §5.1 and introduce our mutation correction algorithm in §5.2.

5.1 Theoretical Foundations

To simplify our analysis, we assume an input MLTP $\mathcal{P}_0$ and its mutant $\mathcal{P}$ each has one output. Our results can be generalized to programs with multiple outputs by sequentially analyzing each one. Let $\mathcal{P}(I)$ denote the output tensor of running $\mathcal{P}$ on n input tensors $I = (I_1, ..., I_n)$. Let $\mathcal{P}(I)[\vec{v}]$ denote the output value at position $\vec{v}$, and let $I_j[\vec{u}]$ denote the input value at position $\vec{u}$ of I_j. With these definitions, the computation for a single output position of an MLTP $\mathcal{P}$ is represented as

$$\mathcal{P}(I_1, ..., I_n)[\vec{v}] = \sum_{\vec{r} \in \mathcal{R}(\vec{v})} \prod_{j=1}^{n} I_j[\mathbf{L}_j(\vec{v}, \vec{r})]$$



(a) Input program. (b) A partially equivalent transformation. (c) Correcting results.

Figure 3: An example mutant that transforms a dilated convolution to a standard convolution. The red-shaded boxes in (b) highlight non-equivalent elements between the two programs, which are fixed by the correction kernel in (c).

where $\mathcal{R}(\vec{v})$ is the *summation interval* of $\vec{v}$, which is iterated over when computing $\mathcal{P}(I)[\vec{v}]$, and $\vec{u} = \mathbf{L}_j(\vec{v},\vec{r})$ is a linear mapping from $(\vec{v},\vec{r})$ to a position $\vec{u}$ of the j-th input tensor I_j. For example, a convolution with a kernel size of 3×3 and zero padding is defined as

$$conv(I_1,I_2)[c,h,w] = \sum_{d=0}^{D-1} \sum_{x=\max(-1,-h)}^{\min(H-1-h,1)} \sum_{y=\max(-1,-w)}^{\min(W-1-w,1)} I_1[d,h+x,w+y] \times I_2[d,c,x,y] \tag{1}$$

where D, H, and W refer to the number of channels, height, and width of the input image I_1, respectively. The numbers below and above the summation symbols respectively denote the lower and upper bounds of the summation interval. The two linear mappings can be represented as $\mathbf{L}_1(\vec{v},\vec{r}) = (d,h+x,w+y)$ and $\mathbf{L}_2(\vec{v},\vec{r}) = (d,c,x,y)$, where $\vec{v} = (c,h,w)$ and $\vec{r} = (d,x,y)$.

Different positions of an output tensor may have different summation intervals. For the convolution operator defined above, computing the top-left output position (i.e., $h = 0$, $w = 0$) only involves a 2×2 kernel (i.e., $0 \le x \le 1, 0 \le y \le 1$) since that position does not have a left or top neighbor, as shown in Figure 4. We group the output positions with an identical summation internal into a *box*. Formally, a box is a region of an output tensor whose elements all have the same summation internal. This convolution has nine boxes overall, which are depicted in Figure 4.

All output positions in the same box have an identical summation internal and share similar mathematical properties, which are leveraged by PET when examining program equivalence. Instead of testing the equivalence of two MLTPs on all individual positions, PET only needs to verify their equivalence on $m+1$ specific positions in each box, where m is the number of dimensions of the output tensor.

Theorem 1 *For two MLTPs $\mathcal{P}_1$ and $\mathcal{P}_2$ with an m-dimension output tensor, let $\vec{e}_1, \ldots, \vec{e}_m$ be a set of m-dimension base vectors. That is, $\vec{e}_i = (0,\ldots,0,1,0,\ldots,0)$ is an m-tuple with all coordinates equal to 0 except the i-th.*

Let $\mathcal{B}$ be a box for $\mathcal{P}_1$ and $\mathcal{P}_2$, and let $\vec{v}_0$ be an arbitrary position in $\mathcal{B}$. Define $\vec{v}_j = \vec{v}_0 + \vec{e}_j, 1 \le j \le m$. If $\forall I, 0 \le i \le m$, $\mathcal{P}_1(I)[\vec{v}_i] = \mathcal{P}_2(I)[\vec{v}_i]$, then $\forall I, \vec{v} \in \mathcal{B}$, $\mathcal{P}_1(I)[\vec{v}] = \mathcal{P}_2(I)[\vec{v}]$.

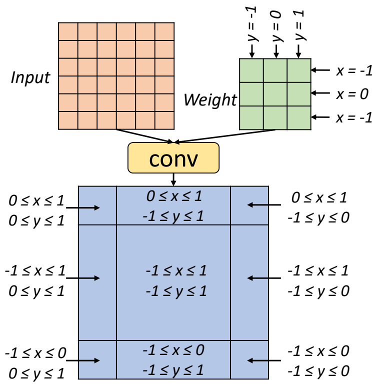


Figure 4: The nine boxes of a convolution with a 3×3 kernel and zero padding, as well as their summation intervals. A convolution has three summation dimensions (i.e., d, x, and y in Equation (1)). The channel dimension (i.e., d) has the same internal in all boxes and is thus omitted.

Proof sketch. The proof uses a lemma whereby if $\mathcal{P}_1$ and $\mathcal{P}_2$ are equivalent for positions $\vec{v}_0$ and $\vec{v}_0 + \vec{e}_i$, then the equivalence holds for $\vec{v}_0 + k \cdot \vec{e}_i$, where k is an integer. We prove this lemma by comparing the coefficient matrices of $\mathcal{P}_1$ and $\mathcal{P}_2$ with respect to the input variables. Using this lemma, we show that $\mathcal{P}_1$ and $\mathcal{P}_2$ are equivalent for the entire box $\mathcal{B}$, since any $\vec{v} \in \mathcal{B}$ can be decomposed to a linear combination of $\vec{v}_0$ and $\vec{e}_0, \ldots, \vec{e}_m$. $\square$

Theorem 1 shows that, if $\mathcal{P}_1$ and $\mathcal{P}_2$ are equivalent for $m+1$ specific positions in a box, identified by $\vec{v}_0, \ldots, \vec{v}_m$, then the equivalence holds for all other positions in the same box. This theorem significantly reduces the verification workload:

Table 2: Reducing verification workload in PET.

Methods	Output positions	Input combinations
Original	all	all
+ Theorem 1	a few positions	all
+ Theorem 2	a few positions	a few random inputs

instead of examining all positions of an output tensor, PET only needs to verify $m+1$ specific positions in each box.

The verification of a single position remains challenging, nevertheless, as each MLTP generally involves a large number of input variables. Proving the equivalence of two MLTPs requires examining all possible combinations of value assignments to these input variables. We further address this challenge using the following theorem.

Theorem 2 *For two MLTPs $\mathcal{P}_1$ and $\mathcal{P}_2$ with n input tensors, let $\vec{v}$ be a position where $\mathcal{P}_1$ and $\mathcal{P}_2$ are not equivalent, i.e., $\exists I$, $\mathcal{P}_1(I)[\vec{v}] \neq \mathcal{P}_2(I)[\vec{v}]$. Let I' be a randomly generated input uniformly sampled from a finite field $\mathbb{F}$. The probability that $\mathcal{P}_1(I')[\vec{v}] = \mathcal{P}_2(I')[\vec{v}]$ is at most $\frac{n}{p}$, where p is the number of possible values in $\mathbb{F}$.*

Proof sketch. This is a corollary of the Schwartz–Zippel Lemma [28, 35]. □

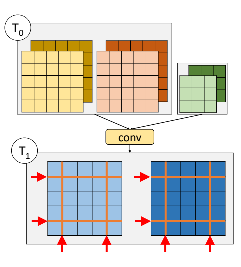
Theorem 2 shows that if two MLTPs with n inputs are not equivalent on a specific position $\vec{v}$, then the probability that they produce an identical result on this position with a random input sampled from a finite field $\mathbb{F}$ is low (i.e., at most $\frac{n}{p}$, where p is the number of possible values in $\mathbb{F}$). This theorem shows the sufficiency and effectiveness of random testing for examining the equivalence of two MLTPs.

Theorem 2 relies on the fact that $\mathbb{F}$ is a finite field, from which the random inputs are sampled, but MLTPs operate on the infinite field of real numbers. To apply Theorem 2, we choose $\mathbb{F}$ to be a field of integers modulo p, where p is a large prime number ($p = 2^{31}-1$ in our evaluation). The arithmetic operations in random testing are performed on integers and calculated modulo the prime number p. Working with a finite field provides another desirable property that applying arithmetic operators does not involve integer overflow.

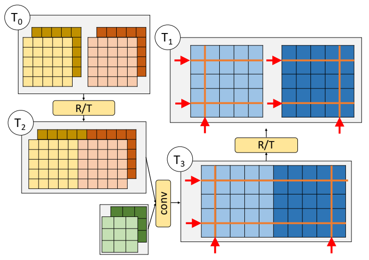
By combining Theorems 1 and 2, PET reduces the original verification task of examining all output positions with respect to all input value combinations to a much more lightweight task that only requires testing a few representative positions using several randomly generated inputs, as shown in Table 2.

5.2 Mutation Correction Algorithm

The PET mutation correction algorithm exploits the theorems in §5.1 to calculate which regions of the output tensors in a mutant are not equivalent to the input MLTP and, therefore, need additional correction. In particular, it suffices to examine the equivalence for each pair of overlapped boxes from the two MLTPs, using a small number of random tests. The overall algorithm works in the following three steps.



(a) Input program $\mathcal{P}_0$.



(b) A potential mutant $\mathcal{P}_1$.

Figure 5: Box propagation for the example in Figure 1. The red arrows indicate the split points of each tensor dimension.

Step 1: Box propagation. First, we calculate the boxes of a given MLTP through *box propagation*. The idea of box propagation is similar to forward and backward propagation in deep learning: we compute the boxes of an operator's output tensors based on the boxes of its inputs, and the computation is conducted following the operator dependencies in a program. We maintain a set of *split points* for each dimension of a tensor to identify the boundaries of its boxes. For a multi-linear operator, we infer the split points of its output tensors based on the split points of its input tensors and the operator type and hyper-parameters. Figure 5 shows the box propagation procedure for the mutation example in Figure 1.

Step 2: Random testing for each box pair. After obtaining all boxes of an input MLTP $\mathcal{P}_1$ and its mutant $\mathcal{P}_2$, PET leverages the theorems in §5.1 to examine the intersected regions of each pair of boxes from $\mathcal{P}_1$ and $\mathcal{P}_2$. If two boxes do not have any overlapped region, they can be skipped. For each box intersection, PET examines the equivalence of the two programs on $m+1$ positions identified by Theorem 1, where m is the number of output tensor dimensions (e.g., $m=4$ in Figure 5, since the output of a convolution has four dimensions).

For each of these $m+1$ positions, PET runs a set of random tests by assigning input tensors with values uniformly sampled from a finite field $\mathbb{F}$ containing all integers between 0 and $p-1$, where $p = 2^{31}-1$ is a prime number. As a result, the probability that two non-equivalent MLTPs produce identical outputs on a random input is at most $\frac{n}{p}$, where n is the number of inputs to the MLTPs. Finally, two non-equivalent MLTPs pass all tests with a probability lower than $(\frac{n}{p})^t$, where t is the number of test cases and a hyper-parameter in PET that serves as a tradeoff between the speed of the corrector and the error probability that non-equivalent MLTPs pass all random tests.

Our approach introduces an extremely small and controllable probability of error that we have to tolerate. That is, non-equivalent programs may pass random testing with probability $(\frac{n}{p})^t$. We argue that this is an example of how random testing can enable a tradeoff between the cost of program verification and a small probability of unsoundness for verifying

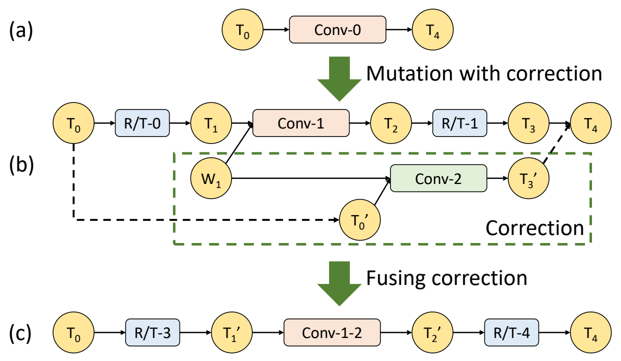


Figure 6: Fusing correction kernels with DNN kernels.

tensor program transformations.

To further reduce the verification workload, PET includes a *caching optimization*: the tests for all boxes share the same set of random inputs, and PET caches and reuses all intermediate results to avoid redundant computations.

Step 3: Correction kernel generation. For each box failing the random tests, PET generates *correction kernels* to fix its outputs and restore the mathematical equivalence between the original MLTP and its mutant. To fix the outputs, the correction kernel performs the same set of operations as the original MLTP but only on those boxes where the two input programs are not equivalent (shown as the red shaded boxes in Figure 1). These boxes are regular cubes in the multi-dimensional space and can be viewed as sub-tensors of the original ones but with much smaller sizes. Therefore, PET directly leverages existing DNN libraries [8, 10] or kernel generation techniques [6, 34] to generate correction kernels. To reduce the correction overhead, PET opportunistically fuses the correction kernels with existing tensor operators (§5.3).

5.3 Fusing Correction Kernels

Correction kernels may introduce non-trivial overheads due to the cost of launching the correction kernels and their limited degrees of parallelism. For example, some correction kernels may have similar execution time compared to the corresponding full-size tensor operators. This may eliminate the performance gains from applying partially equivalent transformations. To reduce the correction overhead, PET opportunistically fuses correction kernels with other tensor operators.

For example, Figure 6(b) shows the tensor program after applying the partially equivalent transformation in Figure 1. `Conv-2` is the correction kernel for fixing the output of `Conv-1`. Since the two convolution operators share the same weights (i.e., W_1), PET fuses them into a single convolution, shown as `Conv-1-2` in Figure 6(c). This fusion requires concatenating T_1 and T_0' into a single tensor and splitting the output of `Conv-1-2` into T_2 and T_3'. The concatenation and split only involve direct memory copies and can be fused with the `reshape` and `transpose` operators.

6 Program Optimizer

In this section, we describe the program optimizer in PET, which explores a large search space of program optimizations, combining fully and partially equivalent transformations, and quickly discovers highly optimized programs. The program optimizer first splits an input program into multiple subprograms with smaller sizes to allow efficient mutation generation (§6.1). Second, to optimize each individual subprogram, PET searches for the best mutants in a rich candidate space by varying both the subsets of operators to mutate together and the number of iterative rounds of mutation (§6.2). Finally, when stitching the optimized subprograms back together, PET applies additional post-optimizations across the boundaries of the subprograms, including redundancy elimination and operator fusion (§6.3). The overall program optimization algorithm is summarized in Algorithm 2.

Algorithm 2 Program optimization algorithm.

```
1: Input: An input tensor program $\mathcal{P}_0$
2: Output: An optimized tensor program $\mathcal{P}_{opt}$
3:
4: Split $\mathcal{P}_0$ into a list of subprograms
5: Initialize a heap $\mathcal{H}$ to record the top-$K$ programs
6: $\mathcal{H}$.insert($\mathcal{P}_0$)
7: // Greedily mutate each subprogram
8: for each subprogram $\mathcal{S} \in \mathcal{P}_0$ do
9:     $mutants$ = GETMUTANTS($\mathcal{S}$)
10:    Initialize a new heap $\mathcal{H}_{new}$
11:    for $\mathcal{P} \in \mathcal{H}$ do
12:        for $\mathcal{M} \in mutants$ do
13:            $\mathcal{P}_{new}$ = replace $\mathcal{S}$ with $\mathcal{M}$ in $\mathcal{P}$
14:            Apply post-optimizations on $\mathcal{P}_{new}$
15:            $\mathcal{H}_{new}$.insert($\mathcal{P}_{new}$)
16:    $\mathcal{H} = \mathcal{H}_{new}$
17: $\mathcal{P}_{opt}$ = the program with the best performance in $\mathcal{H}$
18: return $\mathcal{P}_{opt}$
19:
20: function GETMUTANTS($\mathcal{S}_0$)
21:    $\mathcal{O}$ = the set of mutant operators for $\mathcal{S}_0$
22:    $\mathcal{Q} = \{\mathcal{S}_0\}$, $mutants = \{\mathcal{S}_0\}$
23:    for $r$ rounds do
24:        $\mathcal{Q}_{new} = \{\}$
25:        for $\mathcal{S} \in \mathcal{Q}$ do
26:            for each subset of operators $\mathcal{S}' \in \mathcal{S}$ do
27:                for $\mathcal{M}' \in$ MUTATIONGENERATOR($\mathcal{O}, \mathcal{S}'$) do
28:                    $\mathcal{M}$ = replace $\mathcal{S}'$ with $\mathcal{M}'$ in $\mathcal{S}$
29:                    Add $\mathcal{M}$ to $\mathcal{Q}_{new}$ and $mutants$
30:        $\mathcal{Q} = \mathcal{Q}_{new}$
31:    return $mutants$
```

6.1 Program Splitting

The complexity of the mutation generation grows rapidly with the input program size, as explained in §4. It is nearly impossible to directly mutate a large tensor program with many hundreds of operators. Instead, PET splits an input program into multiple disjoint subprograms with smaller sizes.

It is crucial to properly select the split points for an input program, to effectively reduce the mutation complexity while still preserving most program optimization opportunities. More split points lead to smaller subprograms with fewer mutant candidates to be explored. As an extreme case, by constraining each subprogram to have only a small constant number of operators, the overall complexity scales linearly with the program size, rather than the naive exponential trend. However, an overly aggressive split may result in locally optimized mutants that are limited within subprograms, missing optimization opportunities across subprogram boundaries.

We use *non-linear operators* in tensor programs as the split points. First, non-linear operators such as the activation layers in DNNs are widely used in tensor programs. Typically, each one or a few linear operators are followed by a non-linear activation (e.g., ReLU or sigmoid). This effectively limits the split subprograms to the small sizes we expect. Second, as §5 explains, PET's mutation only applies to MLTPs; any non-linear operators must be excluded from the mutation. This makes splitting at the points of non-linear operators a natural choice for the partially equivalent mutation in PET. Third, our design is also motivated by an observation that most existing tensor program transformations [2, 6, 15] also do not include non-linear operators in their substitution patterns (except for fusion, which we handle in §6.3).

PET further adjusts the subprogram sizes after splitting an input program at the non-linear operators. For multiple individual subprograms without any data dependency, PET considers the possibility of combining them into a single subprogram using grouped or batched operators. Examples include fusing the standard convolutions on different branches of an Inception network [31] into a grouped convolution, as shown in Figure 10. On the other hand, if a subprogram is still too large, PET will only query the mutation generator with a subset of operators each time (see §6.2).

6.2 Subprogram Optimization

After splitting an input program into multiple individual subprograms, PET mutates each subprogram by querying the mutation generator in §4.1 and keeps the top-K candidates with the best estimated performance in a heap structure $\mathcal{H}$, as shown from Lines 7 to 16 in Algorithm 2. A larger K allows PET to tolerate intermediate performance decreases during the search but requires more memory to save all K candidates and involves higher computation cost. At each step, each of the obtained mutants replaces its corresponding subprogram in each of the current candidates (i.e., $\mathcal{P}$ in Algorithm 2) to generate a new candidate (i.e., $\mathcal{P}_{\text{new}}$), which is then applied a series of post-optimizations (see §6.3).

PET estimates the performance of each new candidate $\mathcal{P}_{\text{new}}$

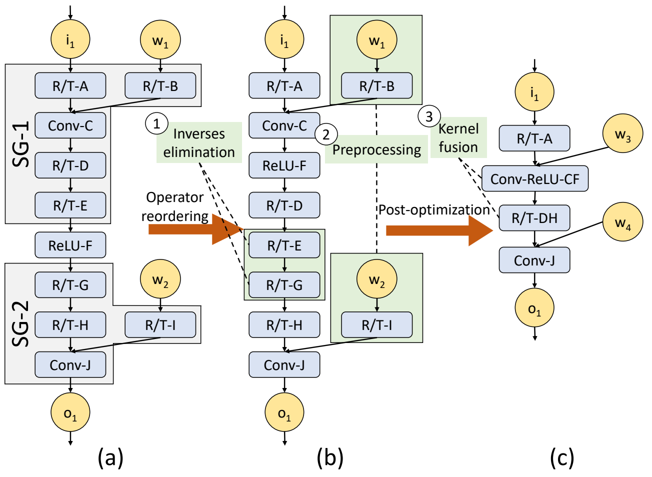


Figure 7: Post-optimizations applied when stitching two subprograms `SG-1` and `SG-2`. `R/T` refers to a `reshape` followed by a `transpose`. `Conv` and `ReLU` denote a convolution and a ReLU operator, respectively.

using a cost model adapted from TASO [15]. The cost model measures the execution time of each tensor operator once for each configuration (e.g., different strides and padding of a convolution), and estimates the performance of a new program candidate $\mathcal{P}_{\text{new}}$ by summing up the measured execution time of its operators. The top-K program candidates with the best performance thus far are kept in $\mathcal{H}$.

To explore a sufficiently large space of possible mutants for each subprogram within reasonable time and space cost, we manage the mutation process with several key features. First, when the number of operators in a subprogram exceeds a threshold d (our evaluation uses $d = 4$), PET breaks the subprogram into smaller subsets of operators by enumerating all possible combinations with up to d operators, and only queries the mutation generator on the subset, while keeping the remaining operators unchanged (Algorithm 2 Line 26). Second, we allow iterative mutation on a subprogram for up to r rounds (Algorithm 2 Line 23), which significantly enlarges the search space of possible mutants and allows PET to discover more optimized mutants. All generated mutants in all rounds are returned to the optimizer as potential candidates.

It is worth noting that PET's optimizer is compatible with and can incorporate existing fully equivalent transformations [2, 15] besides PET's mutations. Doing so merely requires enhancing the mutation generator to explore and return fully equivalent transformations as well, which are directly applicable to the input subprograms in the same way as the mutations. By combining fully and partially equivalent transformations, PET explores a significantly larger search space of program optimizations and discovers highly optimized programs that existing optimizers miss.

6.3 Post-Optimizations

Finally, the optimized mutants for all subprograms need to be stitched together. In addition to connecting their input and output tensors, we also perform several post-optimizations across the subprogram boundaries to further improve the overall performance. We observe that the mutation generator in PET introduces a large number of `reshape` and `transpose` (`R/T`) operators, especially at the beginning and the end of each subprogram. There are opportunities to fuse these `R/T` operators across subprograms and further fuse the non-linear operators that are excluded from the above subprogram optimizations.

Figure 7 shows an example with two optimized subprograms. To optimize the boundaries between subprograms, PET first groups together all `R/T` operators between subprograms by reordering the `R/T` operators with element-wise non-linear activations (e.g., ReLU and sigmoid), as shown in Figure 7(b). This reordering is functionally correct, since both `reshape` and `transpose` are commutative with element-wise operators. The reordering also allows PET to fuse the non-linear activations with other linear operators, such as fusing a `Conv` and a subsequent `ReLU` into a `Conv-ReLU`, as shown in Figure 7(c). We then apply the following three post-optimizations.

Inverses elimination. We eliminate any pairs of `R/T` operators that can cancel out each other and therefore are equivalent to a no-op. We call each such pair an inverse group and directly remove them as part of the post-optimization. An example of an inverse group is `R/T-E` and `R/T-G` in Figure 7(b).

Operator fusion. As shown in Figure 7(c), PET fuses the remaining consecutive `R/T` operators into a single operator (e.g., `R/T-DH`) to reduce the kernel launch cost. The non-linear activations in a tensor program are also fused with an `R/T` or with other linear operators. Note that operator fusion is the most commonly used, if not the only, program optimization for non-linear operators. PET is able to recover most of the efficiency that was lost when splitting the tensor program.

Preprocessing. We preprocess any operator if all its input tensors are statically known. For example, in Figure 7(b), both `R/T-B` and `R/T-I` can be preprocessed on the convolution weight tensors w_1 and w_2.

7 Implementation

PET is implemented as an end-to-end tensor program optimization framework, with about 13,000 lines of C++ code and 1,000 lines of Python code. This section describes our implementation of the PET mutation generator and corrector.

Mutation operators. Table 1 lists the tensor operators included in the current implementation of PET. We use cuDNN [8] and cuBLAS [10] as our backend operator libraries. PET can also be extended to include other libraries, such as TVM [6] and Ansor [34]. In our evaluation, we demonstrate this extensibility on TVM and Ansor, and show that they can directly benefit from PET's partially equivalent optimizations and automated corrections.

`Reshape` and `transpose` are two frequently used operators in partially equivalent transformations. Our implementation includes a series of optimizations on them, including eliminating inverse groups of `R/T` operators and fusing consecutive `R/T` operators, as described in §6.3. Since both `reshape` and `transpose` are multi-linear operators, PET directly uses the random testing method introduced in §5 to examine whether a sequence of `R/T` operators forms an inverse group and therefore can be eliminated.

Correction kernels. §5.2 describes a generic approach to generate correction kernels by directly running the original program on the positions with incorrect results. To reduce the correction overhead, PET fuses the correction kernels with other tensor operators, as described in §5.3. The correction kernel fusion introduces additional memory copies, which are also fused with the `R/T` operators during post-optimizations.

8 Evaluation

8.1 Experimental Setup

Platforms. We use a server equipped with two-socket, 28-core Intel Xeon E5-2680 v4 processors (hyper-thread enabled), 256 GB of DRAM, and one NVIDIA Tesla V100 GPU. All experiments use CUDA 10.2 and cuDNN 7.6.5 except for those with TVM and Ansor, which directly use the best kernels generated by these backends.

PET preserves an end-to-end equivalence between the original and optimized programs, same as all the baselines. PET takes ONNX models as input. TensorRT and TASO directly support the ONNX format. For TensorFlow and TensorFlow-XLA, we use the onnx-tensorflow tool [25] for format conversion.

Workloads. We use five DNN architectures. Resnet-18 [14] is a widely used convolutional network for image classification. CSRNet [20] is a dilated convolutional network used for semantic segmentation. Its sampling rate can be arbitrarily adjusted to enlarge the receptive field for higher accuracy. Inception-v3 [31] is an improved version of GoogLeNet [30] with carefully designed Inception modules to improve accuracy and computational efficiency. BERT [12] is a language representation architecture that obtains state-of-the-art accuracy on a wide range of natural language tasks. Resnet3D-18 [13] is a 3D convolutional network for video processing.

Unless otherwise stated, in all experiments, we use CUDA events to measure the elapsed time from launching the first CUDA kernel in a tensor program to receiving the completion notification of the last kernel. We set the default mutation generation depth to 4 (i.e., $depth = 4$ in Algorithm 1) and the search rounds to 4 (i.e., $r = 4$ in Algorithm 2). We further evaluate the scalability of the mutation generator and the program optimizer in §8.5.

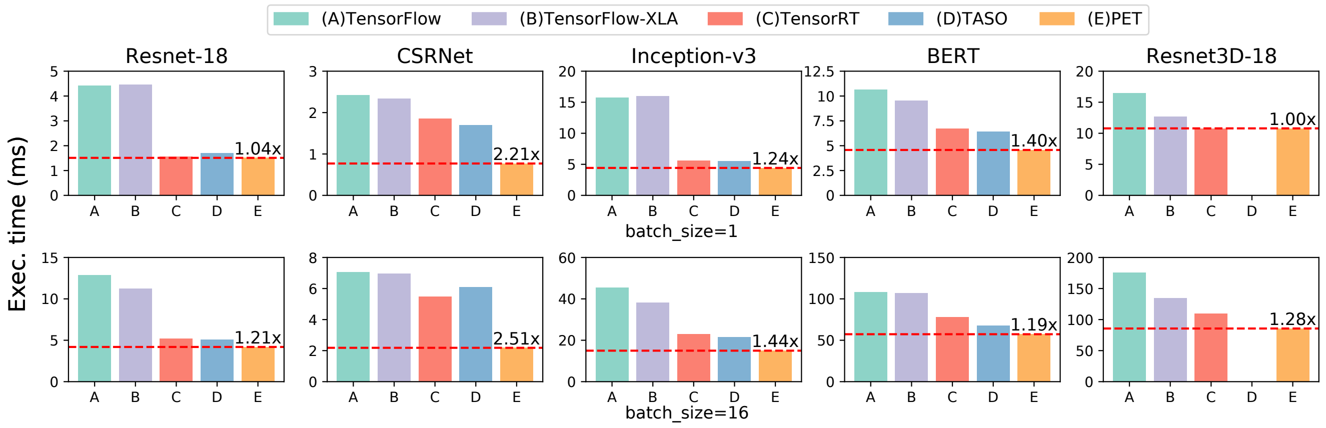


Figure 8: End-to-end performance comparison between PET and existing frameworks. For each DNN, the numbers above the PET bars show the speedups over the best baseline. TASO does not support the 3D convolution operators in Resnet3D-18.

8.2 End-to-End Evaluation

We first compare the end-to-end inference performance between PET and existing tensor program optimizers, including TensorFlow [3], TensorFlow XLA [1], TensorRT [32], and TASO [15]. Figure 8 shows the results under batch sizes of 1 and 16. To eliminate the impact of using different operator libraries, all optimizers use the same cuDNN [8] and cuBLAS [10] libraries as the backend. Therefore, the performance differences only come from different optimized tensor programs produced by PET and the baselines. §8.4 further evaluates PET with existing kernel generation techniques, such as TVM [6] and Ansor [34].

Among the five DNN architectures, Resnet-18 and Resnet3D-18 are commonly used and heavily optimized in existing DNN frameworks. However, PET is still able to improve their performance by up to 1.21× and 1.28×, respectively, by discovering new partially equivalent transformations not considered by existing optimizers. For Resnet-18, CSRNet, and Inception-v3, PET achieves higher speedups with a batch size of 16. This is because a larger batch size offers more mutation opportunities across different tensor dimensions for PET to exploit. Overall, PET outperforms existing DNN frameworks by up to 2.5×.

To further evaluate the partially equivalent transformations discovered by PET, we manually add them and corresponding correction kernels as additional graph substitutions into TASO, and measure by how much these new transformations improve TASO's performance. As shown in Figure 9, the enhanced version of TASO further improves the inference performance of Inception-v3 and BERT by 1.12× and 1.31×, respectively. This demonstrates that partially equivalent transformations indeed enlarge the design space of graph transformations, and PET unleashes these benefits automatically. Some non-trivial partially equivalent transformations are not leveraged by TASO, due to substantial correction overhead, while PET is able to avoid this overhead through correction kernel fusion (§5.3) and post-optimization (§6.3).

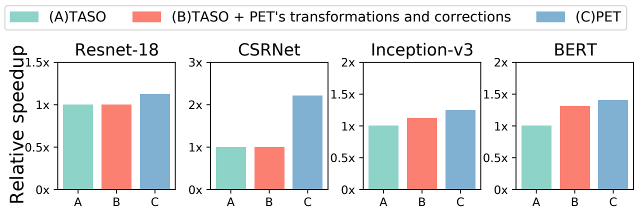


Figure 9: Performance benefits after adding PET's partially equivalent transformations into TASO.

Table 3: Operator benchmark list.

Operator	Input	Weight	#Op
`conv`	[1, 48, 38, 38]	[64, 48, 5, 5]	1
`dilatedconv`	[1, 512, 14, 14]	[256, 512, 3, 3]	1
`groupconv`	[1, 768, 18, 18]	[192,768, 1, 1]	2
	[1, 768, 18, 18]	[160,768, 1, 1]	2
`batchmatmul`	[512, 768]	[768, 768]	3

8.3 Case Studies

To understand how partially equivalent transformations discovered by PET optimize DNN computation, we study four optimization categories in detail.

8.3.1 Tensor-Level Optimization

PET discovers many partially equivalent transformations that improve DNN computation by optimizing the shapes or linearization of tensors. We evaluate a convolution operator in Inception-v3, whose configuration is depicted in Table 3 `conv`. PET transforms the input tensor shape from [1, 48, 38, 38] to [16, 48, 10, 10] by splitting both the height and width dimensions each into four partitions. `IGEMM` and `FFT` are the most efficient convolution algorithms before and after the optimization, respectively. Using the transformed input tensor

Table 4: Case studies on the performance of the `conv` and `dilatedconv` operators in Table 3. `IGEMM`, `FFT`, and `WINO` refer to implicit GEMM, Fast Fourier Transform, and Winograd convolution algorithms, respectively. For `conv`, the optimized program transforms the input tensor shape from [1, 48, 38, 38] to [16, 48, 10, 10]. For `dilatedconv`, the optimized program replaces the `dilatedconv` with a regular convolution with the same input and kernel sizes.

		Algo	Time (us)	# GPU DRAM	# GPU L2	FLOP
`conv`	**Original**	`IGEMM`	90	1.51×10^4	2.80×10^6	2.26×10^8
		`FFT`	352	1.06×10^8	1.15×10^8	8.75×10^7
	Optimized	`IGEMM`	90	1.52×10^4	1.46×10^6	2.46×10^8
		`FFT`	51	1.09×10^6	7.44×10^6	1.26×10^8
`dilated conv`	**Original**	`IGEMM`	153	1.06×10^5	2.46×10^6	1.32×10^8
		`WINO`	N/A	N/A	N/A	N/A
	Optimized	`IGEMM`	153	8.54×10^4	1.80×10^6	1.32×10^8
		`WINO`	79	2.23×10^6	6.36×10^6	7.20×10^7

reduces the GPU DRAM and L2 accesses by 100× and 15×, respectively, and thus reduces the run time by 7× (Table 4).

As another example of tensor-level optimization, for `conv` with a stride size larger than 1 (i.e., the output tensor is a down-sample of the input tensor), PET can reorganize the linearization of the tensors and reduce the stride size to 1, which improves the computation locality.

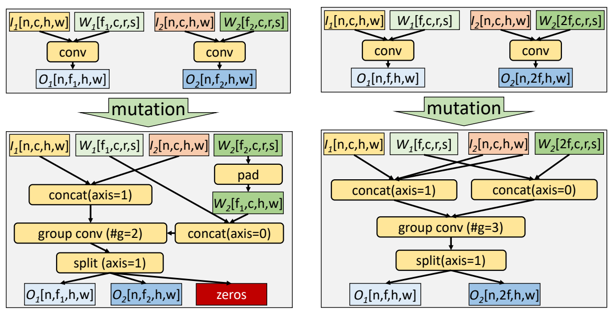
8.3.2 Operator-Level Optimization

For operators with less efficient implementations on specific hardware backends, PET can opportunistically replace them with semantically similar ones with more optimized implementations. We study the performance of a dilated convolution in CSRNet [20], whose configuration is shown in Table 3 `dilatedconv`. PET replaces it with a regular convolution operator (as shown in Figure 3) to enable more efficient algorithms on GPUs such as Winograd [17]. This reduces the execution time by 1.94× (Table 4).

Other examples of operator-level optimizations include replacing a batch matrix multiplication with a standard matrix multiplication, a group convolution with a convolution, and an average pooling with a group convolution or a convolution if the replacement leads to improved performance, even when including the correction cost.

8.3.3 Graph-Level Optimization

PET also discovers graph-level optimizations. Figure 10 shows two graph transformations discovered by PET to optimize Inception-v3 [31]. For two parallel `conv` operators with different numbers of output channels, Figure 10(a) shows a non-equivalent transformation that fuses the two `conv` operators into a `groupconv` by padding W_2 with zeros, so that the output of `pad` has the same shape as W_1. The correction splits and discards the *zeros* tensor at the end (shown in red).



(a) (b)

Figure 10: Mutants discovered by PET for Inception-v3. `axis` denotes the dimension on which to perform `concat` and `split`.

PET also discovers fully equivalent transformations that are missed by existing frameworks. The mutation corrector can successfully verify the equivalence for all output elements, in which case no correction is needed. Figure 10(b) shows a new equivalent transformation discovered by PET that optimizes two `conv` operators by duplicating the input tensors (i.e., I_1 and I_2) and fusing the two `conv` operators into a `groupconv`. Note that Figure 10 shows two different mutants of the same input program. PET's program optimizer can automatically select a more efficient one based on the performance of these mutants on specific devices.

8.3.4 Kernel Fusion

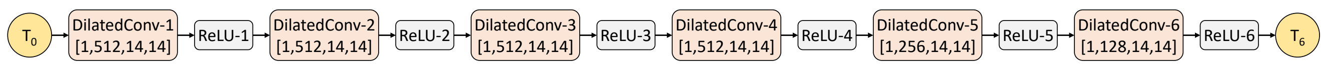
We use CSRNet [20] as an example to study the effectiveness of PET's kernel fusion optimization. Figure 11(a) and Figure 11(b) show the original and optimized model architectures of CSRNet. The numbers in each operator denote the input tensor shape. To demonstrate the correction kernel fusion and post-optimization in PET, Figure 11(c) shows the subprogram of a single dilated convolution before post-optimization, which contains three correction kernels and six `R/T` (i.e., `reshape` and `transpose`) operators. These correction kernels are fused with `Conv-4`, as described in §5.3. In addition, the multiple `R/T` operators between convolutions are fused into a single one during post-optimization (§6.3).

Fusing correction kernels and `R/T` operators is critical to PET's performance. In an ablation study, disabling kernel fusion in PET decreases the performance of the final program by 2.9×, making it even slower than the original one.

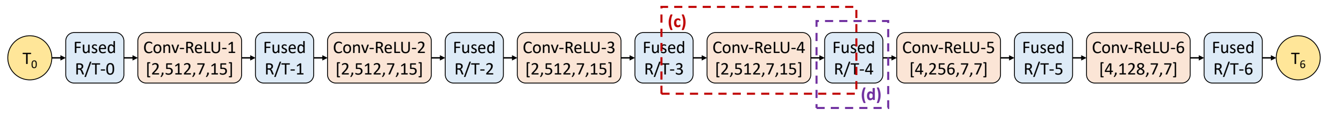
8.4 TVM and Ansor

PET improves tensor computations by generating and correcting partially equivalent transformations and is therefore orthogonal to and can potentially be combined with recent kernel generation techniques, such as TVM [6] and Ansor [34].

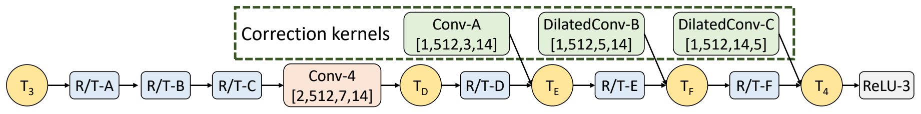
We evaluate PET on TVM and Ansor with a set of commonly used DNN operators, including `conv`, `dilatedconv`,


T0
DilatedConv-1 [1,512,14,14]
ReLU-1
DilatedConv-2 [1,512,14,14]
ReLU-2
DilatedConv-3 [1,512,14,14]
ReLU-3
DilatedConv-4 [1,512,14,14]
ReLU-4
DilatedConv-5 [1,256,14,14]
ReLU-5
DilatedConv-6 [1,128,14,14]
ReLU-6
T6

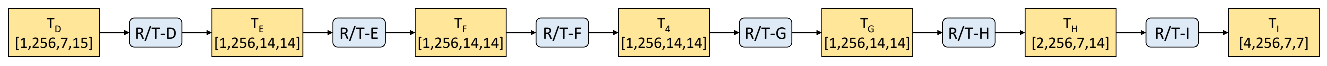
(a) CSRNet before optimization.



(b) CSRNet after optimization.



(c) `DilatedConv-4`'s subprogram after subprogram optimization but before post-optimization.



(d) Unfused R/T operators and corresponding tensors' shape of F R/T-4.

Figure 11: Optimization details in PET for CSRNet.

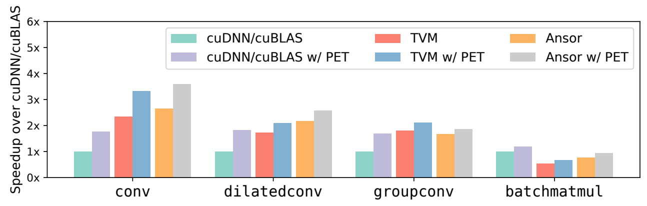


Figure 12: Performance comparison of PET on the cuDNN/cuBLAS, TVM, and Ansor backends. The performance is normalized to cuDNN/cuBLAS without PET.

`groupconv`, and `batchmatmul`, which are obtained from Resnet-18, CSRNet, Inception-v3, and BERT, respectively. Their shape configurations are listed in Table 3. To generate kernels for potential mutants during the search, we allow TVM and Ansor to run 1024 trials and use the best discovered kernels to measure the cost of the mutants.

As Figure 12 shows, when combining PET with TVM and Ansor, PET can improve the performance of the evaluated operators by up to 1.23× and 1.21×, respectively, compared to directly generating kernels for these operators. Beyond such simple combinations, joint optimization of PET and existing kernel generation techniques would uncover more benefits, which we leave as future work.

8.5 Ablation and Sensitivity Studies

The key insight of PET is to explore partially equivalent program mutants, while state-of-the-art frameworks only capture fully equivalent transformations [15, 34]. We run several variants of PET to evaluate the benefits of considering either fully or partially equivalent program transformations, or both of them, as PET does. Figure 13 shows the results. When restricting PET to consider only equivalent transformations, it achieves similar performance gains as previous work such as

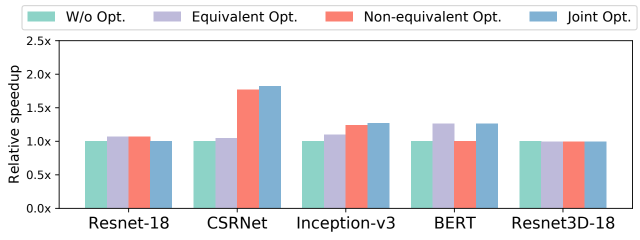


Figure 13: Performance comparison of tensor program optimizations using only (fully) equivalent transformations, only partially equivalent transformations, and both (as in PET).

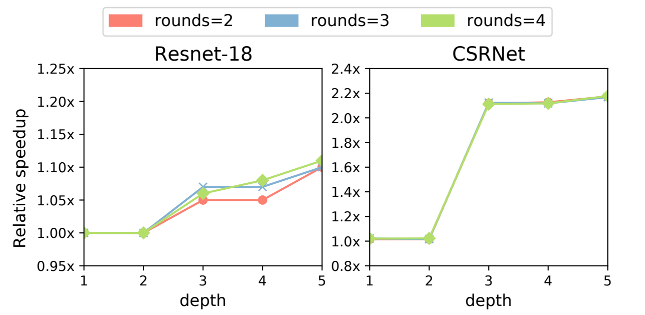


Figure 14: Performance comparison by using PET with different mutation depths (§4.1) and rounds (§6.2).

TASO. Partially equivalent transformations, by themselves, enable noticeable benefits but also miss significant potential. Finally, PET achieves the highest performance by jointly considering both fully and partially equivalent transformations.

Finally, PET relies on several heuristic parameters to balance the search time and the resultant program performance. The mutation depth in Algorithm 1 limits the maximum number of operators in a program mutant; the mutation round in Algorithm 2 specifies the maximum number of iterations to apply mutations. Larger values of these thresholds allow larger design spaces of potential mutants but also require more time to search. Figure 14 compares the performance of the optimized programs under different searching depths and rounds for Resnet-18 and CSRNet. The performance gains keep increasing with larger rounds values for Resnet-18, due to the generation of more optimized mutants, while for CSRNet, the performance improvement mainly comes from larger mutation depth. On the other hand, increasing the mutation depth from two to three improves the performance for both models significantly, since many mutations PET finds are subprograms with three operators. In summary, the key takeaway is that PET has only moderately high search complexity yet achieves significant performance gains.

8.6 Searching Time

PET uses a program optimizer to explore the search space of possible mutants and discover highly optimized candidates. Typically, it takes under 3 minutes (89 seconds, 88 seconds, 91 seconds, and 165 seconds on Resnet-18, CSRNet, BERT, and Resnet3D-18, respectively) for PET to find highly optimized program mutants with a batch size of 1. However, PET spends about 25 minutes optimizing Inception-v3, due to the multiple branches in the Inception modules [31]. Although their search spaces are not directly comparable, PET's search time is on par with state-of-the-art DNN optimization frameworks such as TASO [15] and Ansor [34], and is acceptable because it is a one-time cost before stable deployment. We leave any further search optimizations, such as aggressive pruning and parallelization, to future work.

9 Related work

Graph-level optimizations. TensorFlow [3], TensorRT [32], TVM [6], and MetaFlow [16] optimize tensor programs by applying substitutions that are manually designed by domain experts. TASO [15] generates graph substitutions automatically from basic operator properties, which significantly enlarges search space and reduces human effects. The key difference between PET and these frameworks is that PET can generate and correct partially equivalent transformations, enabling a significantly larger space of program optimizations.
Program mutation is a program testing technique designed to evaluate the quality of existing test cases [11]. By randomly mutating the input program and running the generated mutants on existing test cases, the technique can quickly estimate the coverage of these test cases. PET generates mutants for a different purpose. Instead of testing an input tensor program, the mutants generated by PET are used for performance optimizations on the program.
Code generation. Halide [27] is a programming language designed for high-performance tensor computing, and several works are proposed based on its scheduling model [4, 19, 22]. TVM [6, 7] uses a similar scheduling language and a learning-based approach to generate highly optimized code for different hardware backends. Ansor [34] explores larger search spaces than TVM and finds better optimized kernels. TensorComprehensions [33] and Tiramisu [5] use polyhedral compilation models to solve code generation problems in deep learning. As shown in §8.4, PET's program-level optimizations are orthogonal and can be combined with these code generation techniques.
Data layout optimization. NeoCPU [21] optimizes CNN models by changing the data layout and eliminating unnecessary layout transformations on CPUs, while Li et al. [18] explore the memory efficiency for CNNs on GPUs. Chou et al. [9] introduce a language to describe the different sparse tensor formats and automatically generate code for converting data layouts. Many transformations discovered by PET also involve layout conversions. However, the key differences between PET and prior work are that PET considers more complicated layouts and combines tensor layout optimizations with operator- and graph-level optimizations.
AutoML. Recent work has proposed approaches to search for accurate neural architectures by iteratively proposing modifications to the models' architectures and accepting proposals with the highest accuracy gain. Examples include automatic statistician [29] and TPOT [24]. These approaches apply non-equivalent transformations to a model architecture and rely on expensive retraining steps to evaluate how each transformation affects model accuracy. On the contrary, PET leverages performance optimizations in non-equivalent transformations and applies automated corrections to preserve an end-to-end equivalence. As such, PET does not require retraining.

10 Conclusion

We present PET, the first DNN framework that optimizes tensor programs with partially equivalent transformations and automated corrections. PET discovers program transformations that improve DNN computations with only partial functional equivalence. Automated corrections are subsequently applied to restore full equivalence with the help of rigorous theoretical guarantees. The results of our evaluation show that PET outperforms existing frameworks by up to 2.5× by unlocking partially equivalent transformations that existing frameworks miss. PET is publicly available at https://github.com/thu-pacman/PET.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Behnaz Arzani, for their valuable comments and suggestions. This work is partially supported by National Natural Science Foundation of China (U20A20226, 62072262) and Beijing Natural Science Foundation (4202031). Jidong Zhai is the corresponding author of this paper (zhaijidong@tsinghua.edu.cn).

References

[1] Xla: Optimizing compiler for tensorflow. https://www.tensorflow.org/xla, 2017.

[2] TensorFlow Graph Transform Tool. https://github.com/tensorflow/tensorflow/tree/master/tensorflow/tools/graph_transforms, 2018.

[3] 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. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2016.

[4] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, et al. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)*, 38(4):1–12, 2019.

[5] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 193–205. IEEE, 2019.

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: end-to-end optimization stack for deep learning. *CoRR*, abs/1802.04799, 2018.

[7] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems 31*, NeurIPS'18. 2018.

[8] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014.

[9] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Automatic generation of efficient sparse tensor format conversion routines. *arXiv preprint arXiv:2001.02609*, 2020.

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

[11] Richard A DeMillo, Edward W Krauser, and Aditya P Mathur. Compiler-integrated program mutation. In *1991 The Fifteenth Annual International Computer Software & Applications Conference*, pages 351–352. IEEE Computer Society, 1991.

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[13] Kensho Hara, Hirokatsu Kataoka, and Yutaka Satoh. Learning spatio-temporal features with 3d residual networks for action recognition. In *Proceedings of the IEEE International Conference on Computer Vision Workshops*, pages 3154–3160, 2017.

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

[15] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62, 2019.

[16] Zhihao Jia, James Thomas, Todd Warzawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. Optimizing dnn computation with relaxed graph substitutions. In *Proceedings of the 2nd Conference on Systems and Machine Learning*, SysML'19, 2019.

[17] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4013–4021, 2016.

[18] Chao Li, Yi Yang, Min Feng, Srimat Chakradhar, and Huiyang Zhou. 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, 2016.

[19] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. Differentiable programming for image processing and deep learning in halide. *ACM Transactions on Graphics (TOG)*, 37(4):1–13, 2018.

[20] Yuhong Li, Xiaofan Zhang, and Deming Chen. Csrnet: Dilated convolutional neural networks for understanding the highly congested scenes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1091–1100, 2018.

[21] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. Optimizing {CNN} model inference on cpus. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 1025–1040, 2019.

[22] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics (TOG)*, 35(4):1–11, 2016.

[23] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pages 807–814, USA, 2010. Omnipress.

[24] Randal S Olson and Jason H Moore. Tpot: A tree-based pipeline optimization tool for automating machine learning. In *Workshop on automatic machine learning*, pages 66–74. PMLR, 2016.

[25] TensorFlow Backend for ONNX. `https://github.com/onnx/onnx-tensorflow`.

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

[27] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, 2013.

[28] Jacob T Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM (JACM)*, 27(4):701–717, 1980.

[29] Christian Steinruecken, Emma Smith, David Janz, James Lloyd, and Zoubin Ghahramani. The automatic statistician. In *Automated Machine Learning*, pages 161–173. Springer, Cham, 2019.

[30] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.

[31] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[32] NVIDIA TensorRT: Programmable inference accelerator. `https://developer.nvidia.com/tensorrt`, 2017.

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

[34] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: generating high-performance tensor programs for deep learning. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 863–879, 2020.

[35] Richard Zippel. Probabilistic algorithms for sparse polynomials. In *International symposium on symbolic and algebraic manipulation*, pages 216–226. Springer, 1979.

A Artifact Appendix

A.1 Abstract

This artifact appendix helps the readers reproduce the main evaluation results of the OSDI' 21 paper: PET: Optimizing Tensor Programs with Partially Equivalent Transformations and Automated Corrections.

A.2 Pet Usage

PET provides C++ API to build the input tensor program, and also supports importing input tensor program from ONNX[1] model. For each input tensor program, PET generates a mathematically equivalent executable that includes the performance optimizations described in this paper. PET uses cuDNN/cuBLAS as backend by default, but users can also export the mutation subprograms with their corresponding input/output tensor shapes to use different backends like TVM and Ansor.

A.3 Scope

The artifact can be used for evaluating and reproducing the main results of the paper, including the end-to-end evaluation, the operator-level evaluation, the performance comparison across different optimization policies and heuristics parameters, and the searching time.

A.4 Contents

The artifact evaluation includes the following experiments:
E1: An end-to-end performance comparison between PET and other frameworks. (Figure 8)
E2: An operator-level performance comparison on different backends, including cuDNN/cuBLAS, TVM, and Ansor. (Figure 12)
E3: A performance comparison across different optimization policies, including fully-equivalent transformations, partially-equivalent transformations, and joint optimization using both. (Figure 13)

[1]`https://onnx.ai/`

E4: A performance comparison using different heuristics. (Figure 14)
E5: Searching time. (Section 8.6)

A.5 Hosting

The source code of this artifact can be found on GitHub: `https://github.com/whjthu/pet-osdi21-ae`, master branch, with commit ID: 9e07cb1.

A.6 Requirements

Hardware dependencies

This artifact depends on an NVIDIA V100 GPU.

Software dependencies

This artifact depends on the following software libraries:

- PET uses cuDNN and cuBLAS libraries as backend. Our evaluation uses CUDA 10.2 and cuDNN 7.6.5.
- TensorFlow, TensorRT, TASO, TVM and Ansor are used as baseline DNN frameworks in E1 and E2. Our evaluation on these baseline uses TensorFlow 1.15, TensorRT 7.0.0.11, TASO with commit ID f11782c (we add some minor fixes for TASO to support the tested models), and TVM with commit ID 3950639.

A.7 Installation

A.7.1 Install Pet from source

- Clone code from git
- Install PET
  - `mkdir build; cd build; cmake ..`
  - `make -j`
- Set the environment for evaluations
  - `export PET_HOME=path_to_pet_home`

A.7.2 Install other frameworks

Please refer to the artifact evaluation instruction (README.pdf in the git repo `https://github.com/whjthu/pet-osdi21-ae`) or the installation instructions provided by the frameworks.

A.8 Experiments workflow

The following experiments are included in this artifact. All DNN benchmarks use synthetic input data in GPU device memory to remove the side effects of data transfers between CPU and GPU. The detailed running instruction can be found in the artifact evaluation instruction (README.pdf in the git repo `https://github.com/whjthu/pet-osdi21-ae`).

A.8.1 End-to-end performance (E1)

This experiment reproduces Figure 8 in the paper. Prerequisite: generate ONNX models

- `cd $PET_HOME/models-ae`
- `./generate_onnx.sh`

TensorFlow & TensorFlow XLA. The TensorFlow & TensorFlow XLA results of the 4 models are available in the `tensorflow_ae` folder. The following command lines measure the inference latency of TensorFlow and TensorFlow XLA, respectively:

- `cd $PET_HOME/tf-ae`
- `./run.sh`

TensorRT. The TensoRT results of the 4 models are available in the `tensorrt_ae` folder. The following command lines measure the inference latency of TensorRT:

- Load TensorRT environment (add library path to `LD_LIBRARY_PATH`)
- `cd $PET_HOME/trt-ae`
- `./run.sh`

TASO. The TASO results of the 4 models are available in the `taso_ae` folder. The following command lines measure the inference latency of TASO:

- Load TASO environment
- `cd $PET_HOME/taso-ae`
- `./run_e2e.sh`

PET. The PET results of the 4 models are available in the `pet_ae` folder. The following command lines measure the inference latency of PET:

- `cd $PET_HOME/pet-ae`
- `./run_e2e.sh`

A.8.2 Operator-level performance (E2)

This experiment reproduces Figure 12 in the paper. The scripts are available in the `operator_ae` folder. The experiments of TVM and Ansor will take a very long time to search different mutation kernels.

cuDNN/cuBLAS. The following command lines measure cuDNN/cuBLAS results for the 4 operator-level benchmarks:

- `cd operator_ae/cudnn`
- `./run.sh`

TVM & Ansor. The scripts in `operator_ae/autotvm` and `operator_ae/ansor` search the kernels for the 4 operator-level benchmarks using TVM and Ansor, respectively.

A.8.3 Different optimization policy (E3)

This experiment reproduces Figure 13 in the paper. The scripts are available in the `pet-ae` folder. The following command lines measure the results:

- `cd $PET_HOME/pet-ae`
- `./run_policy.sh`

A.8.4 Different heuristic parameters (E4)

This experiment reproduces Figure 14 in the paper. The scripts are available in the `pet-ae` folder. The following command lines measure the results:

- `cd $PET_HOME/pet-ae`
- `./run_param.sh`

A.8.5 Searching time (E5)

This experiment reproduces Section 8.6 in the paper. The scripts are available in the `pet-ae` folder. The same commands for PET in E1 print the searching time at the same time.

- `cd $PET_HOME/pet-ae`
- `./run_e2e.sh`

Note that our evaluation platform for AE has different CPUs from the platform we used for the paper so that the searching time could be different. Nevertheless, they should be within the same scale.

Theorems Appendix

1 Tensor and tensor programming

We define a L-order tensor T as a multi-array $T \in \mathbb{R}^{d_1 \times d_2 \times \cdots d_m}$, where each $d_i \in \mathbb{N}$. We call $(d_1, d_2, \cdots, d_m)$ the dimension of the tensor T.

A multi-linear tensor program P is defined as a mapping that takes inputs (1). Tensors $\vec{T} = (T_1, T_2, T_3, \cdots, T_J)$ where each T_j has dimension $(d_{j,1}, d_{j,2}, \cdots, d_{j,m_j})$, (2). Positions $\vec{v} = (v_1, v_2, \cdots v_m) \in \mathcal{V} \subseteq \mathbb{N}^m$, such that for a finite set $\mathcal{B} \subsetneq \mathbb{Z}^S$ and linear functions $L_{j,k} : \mathcal{V} \times \mathcal{B} \to [d_{j,k}]$ where $j \in [J], k \in [m_j]$, we have:

$$P(T_1, T_2, \cdots, T_L)[v_1, v_2, \cdots, v_m] \tag{1.1}$$

$$= \sum_{\vec{r} \in \mathcal{B}} \prod_{j \in [J]} [T_j]_{L_{j,1}(\vec{v},\vec{r}), L_{j,2}(\vec{v},\vec{r}), \cdots, L_{j,m_j}(\vec{v},\vec{r})} \tag{1.2}$$

Now, we have the following main theorem:

Theorem 1.1 (Main). *Suppose $P^{(1)}, P^{(2)}$ are two multi-linear tensor programs that both take inputs as tensors $\vec{T} = (T_1, T_2, T_3, \cdots, T_J)$ and positions $\vec{v} = (v_1, v_2, \cdots v_m) \in \mathcal{V} \subseteq \mathbb{N}^m$, and each defined by sets $\mathcal{B}^{(1)}, \mathcal{B}^{(2)}$ and linear functions $L^{(1)}_{j,k}, L^{(2)}_{j,k}$ respectively. Suppose there exists m'-many ($m' \leq m+1$) $\vec{v}^{(i)} \in \mathcal{V}, i \in [m]$ such that: $\left(\mathcal{V} - \vec{v}^{(1)}\right) \subseteq$ **span**$\{\vec{v}^{(i)} - \vec{v}^{(1)}\}_{i=2,3,\cdots,m'}$ and:*

$$\forall i \in [m'], \forall \vec{T} : \; P^{(1)}(\vec{T})[\vec{v}^{(i)}] = P^{(2)}(\vec{T})[\vec{v}^{(i)}] \tag{1.3}$$

Then we must have:

$$\forall \vec{T}, \forall \vec{v} \in \mathcal{V} : \; P^{(1)}(\vec{T})[\vec{v}] = P^{(2)}(\vec{T})[\vec{v}] \tag{1.4}$$

In order to prove this theorem, we need the following Lemma:

Lemma 1.2 (Finite Set). *For every $D \in \mathbb{N}$, let $\mathcal{E}$ be a finite set in $\mathbb{R}^D$, suppose there exists a vector $x \in \mathbb{R}^D$ such that*

$$\mathcal{E} + x = \mathcal{E} \tag{1.5}$$

Then we must have that either $x = 0$ or $\mathcal{E} = \varnothing$.

Proof of Lemma 1.2. Let us only focus on the case when $\mathcal{E} \neq \varnothing$. Let us denote $S = |\mathcal{E}|$ and

$$\mathcal{E} = \{g_i\}_{i \in [S]} \tag{1.6}$$

Since $\mathcal{E} + x = \mathcal{E}$, we know that there is a permutation $\pi : [S] \to [S]$ such that:

$$\forall i \in [S], \; g_i + x = g_{\pi(i)} \tag{1.7}$$

For every $j \geq 2$, let us denote $\pi^{(j)}(i) = \pi(\pi^{(j-1)}(i))$ and $\pi^{(1)}(i) = \pi(i)$. By the basic property of permutation, we know that there exists $j^* \in [S]$ such that $\pi^{(j^*)}(1) = 1$. Hence, we have:

$$g_1 = g_{\pi^{(j^*)}(1)} = g_1 + j^* x \tag{1.8}$$

This implies that $x = 0$.

$\square$

Proof of Theorem 1.1. We will show that the two multi-linear tensor programs are equal by comparing the coefficients of the tensors. In particular, for $h_{j,k} \in [d_{j,k}]$, we have that the coefficient in $P(\vec{T})[\vec{v}]$ associated with $\prod_{j \in [J]} [T_j]_{h_{j,1}, h_{j,2}, \cdots, h_{j,m_j}}$ is given by:

$$\mathcal{C}(\{h_{j,k}\}_{j \in [J], k \in [m_j]}) = \sum_{\vec{r} \in \mathcal{B}} \mathbb{I}[\forall j \in [J], k \in [m_j] : L_{j,k}(\vec{v}, \vec{r}) = h_{j,k}] \tag{1.9}$$

Since $L_{j,k}$ is a linear function, we can decompose it as: for linear functions $L_{1,j,k}, L_{2,j,k}$:

$$L_{j,k}(\vec{v}, \vec{r}) := \langle w_{1,j,k}, \vec{v} \rangle + \langle w_{2,j,k}, \vec{r} \rangle + c_{j,k} \tag{1.10}$$

Hence we can re-write the coefficient $\mathcal{C}$ as:

$$\mathcal{C}(\{h_{j,k}\}_{j \in [J], k \in [m_j]})[\vec{v}] = \sum_{\vec{r} \in \mathcal{B}} \mathbb{I}[\forall j \in [J], k \in [m_j] : \langle w_{1,j,k}, \vec{v} \rangle + \langle w_{2,j,k}, \vec{r} \rangle + c_{j,k} = h_{j,k}] \tag{1.11}$$

Now, let us define $\mathbf{W}_1 := (w_{1,j,k})_{j \in [J], k \in [m_j]}$, $\mathbf{W}_2 := (w_{2,j,k})_{j \in [J], k \in [m_j]}$, $\vec{c} = (c_{j,k})_{j \in [J], k \in [m_j]}$ and $\vec{h} := (h_{j,k})_{j \in [J], k \in [m_j]}$, we have that:

$$\forall j \in [J], k \in [m_j] : \langle w_{1,j,k}, \vec{v} \rangle + \langle w_{2,j,k}, \vec{r} \rangle + c_{j,k} = h_{j,k} \iff \mathbf{W}_1 \vec{v} + \mathbf{W}_2 \vec{r} + \vec{c} = \vec{h} \tag{1.12}$$

Hence can get:

$$\mathcal{C}(\vec{h})[\vec{v}] = \left| \left\{ r \in \mathcal{B} \mid \mathbf{W}_2 \vec{r} + \mathbf{W}_1 \vec{v} + \vec{c} = \vec{h} \right\} \right| \tag{1.13}$$

Now, for every $s \in \mathbb{N}$, let us denote

$$\mathcal{E}_s := \left\{ \vec{g} \in \mathbb{Z}^M \mid |\{ \vec{r} \in \mathcal{B} \mid \mathbf{W}_2 \vec{r} + \vec{c} = \vec{g} \}| = s \right\} \tag{1.14}$$

Using the assumption that $\forall \vec{T} : \ P^{(1)}(\vec{T})[\vec{v}^{(i)}] = P^{(2)}(\vec{T})[\vec{v}^{(i)}]$, let us denote $M = \sum_{j \in [J]} m_j$, we can conclude that:

$$\forall \vec{h} \in \mathbb{Z}^M, \ \left| \left\{ \vec{r} \in \mathcal{B}^{(1)} \mid \mathbf{W}_2^{(1)} \vec{r} + \mathbf{W}_1^{(1)} \vec{v}^{(i)} + \vec{c}^{(1)} = \vec{h} \right\} \right| = \left| \left\{ r \in \mathcal{B}^{(2)} \mid \mathbf{W}_2^{(2)} \vec{r} + \mathbf{W}_1^{(2)} \vec{v}^{(i)} + \vec{c}^{(2)} = \vec{h} \right\} \right| \tag{1.15}$$

This implies that for every $s \in \mathbb{N}$:

$$\mathcal{E}_s^{(1)} + \mathbf{W}_1^{(1)} \vec{v}^{(i)} = \mathcal{E}_s^{(2)} + \mathbf{W}_1^{(2)} \vec{v}^{(i)} \tag{1.16}$$

Since $\mathcal{B}^{(1)}, \mathcal{B}^{(2)}$ are both finite, we know that for every $s \geq 1$, $\mathcal{E}_s^{(1)}$ and $\mathcal{E}_s^{(2)}$ are both finite.

Let us first consider $\vec{v}^{(1)}$. For each $s \geq 1$, for $a := \mathbf{W}_1^{(1)} \vec{v}^{(1)} - \mathbf{W}_1^{(2)} \vec{v}^{(1)}$, let us denote $|\mathcal{E}_s^{(1)}| = S_s$ and we can write $\mathcal{E}_s^{(1)}$, $\mathcal{E}_s^{(2)}$ as:

$$\mathcal{E}_s^{(1)} = \{ \vec{g}_{s,j} \}_{j \in [S_s]}, \quad \mathcal{E}_s^{(2)} = \{ \vec{g}_{s,j} + a \}_{j \in [S_s]} \tag{1.17}$$

Now, for other $\vec{v}^{(i)}$, let us write $b := \mathbf{W}_1^{(1)} \vec{v}^{(i)} - \mathbf{W}_1^{(2)} \vec{v}^{(i)}$, we know that:

$$\{ \vec{g}_{s,j} \}_{j \in [S_s]} + b = \{ \vec{g}_{s,j} + a \}_{j \in [S_s]}, \quad \{ \vec{g}_{s,j} \}_{j \in [S_s]} + b - a = \{ \vec{g}_{s,j} \}_{j \in [S_s]} \tag{1.18}$$

Apply Lemma 1.2, we know that for every $s \geq 1$, either $S_s = 0$ or $b - a = 0$. We consider these two cases separately:

1. When $S_s > 0$, we know that

$$\forall i \in [m'], \quad \mathbf{W}_1^{(1)} \vec{v}^{(i)} - \mathbf{W}_1^{(2)} \vec{v}^{(i)} = \mathbf{W}_1^{(1)} \vec{v}^{(1)} - \mathbf{W}_1^{(2)} \vec{v}^{(1)} \tag{1.19}$$

By our assumption, we know that: $\left(\mathcal{V} - \vec{v}^{(1)}\right) \subseteq \mathbf{span}\{\vec{v}^{(i)} - \vec{v}^{(1)}\}_{i=2,3,\cdots,m'}$. This implies that:

$$\forall \vec{v} \in \mathcal{V}, \quad \mathbf{W}_1^{(1)}\vec{v} - \mathbf{W}_1^{(2)}\vec{v} = \mathbf{W}_1^{(1)}\vec{v}^{(1)} - \mathbf{W}_1^{(2)}\vec{v}^{(1)} \tag{1.20}$$

This gives us that

$$\forall \vec{v} \in \mathcal{V}, \quad \mathcal{E}_s^{(1)} + \mathbf{W}_1^{(1)}\vec{v} = \mathcal{E}_s^{(2)} + \mathbf{W}_1^{(2)}\vec{v} \tag{1.21}$$

Which further implies that $\forall \vec{v} \in \mathcal{V}, \forall \vec{h} \in \mathcal{Z}^M$,

$$\left|\left\{\vec{r} \in \mathcal{B}^{(1)} \mid \mathbf{W}_2^{(1)}\vec{r} + \mathbf{W}_1^{(1)}\vec{v} + \vec{c}^{(1)} = \vec{h}\right\}\right| = s \iff \left|\left\{\vec{r} \in \mathcal{B}^{(2)} \mid \mathbf{W}_2^{(2)}\vec{r} + \mathbf{W}_1^{(2)}\vec{v} + \vec{c}^{(2)} = \vec{h}\right\}\right| = s \tag{1.22}$$

2. When $S_s = 0$, we know that $\forall \vec{v} \in \mathcal{V}, \forall \vec{h} \in \mathcal{Z}^M$,

$$\left|\left\{\vec{r} \in \mathcal{B}^{(1)} \mid \mathbf{W}_2^{(1)}\vec{r} + \mathbf{W}_1^{(1)}\vec{v} + \vec{c}^{(1)} = \vec{h}\right\}\right|, \left|\left\{\vec{r} \in \mathcal{B}^{(2)} \mid \mathbf{W}_2^{(2)}\vec{r} + \mathbf{W}_1^{(2)}\vec{v} + \vec{c}^{(2)} = \vec{h}\right\}\right| \neq s \tag{1.23}$$

Together, this completes the proof.

□

Now we show a theorem for the randomized testing:

Theorem 1.3. *Suppose $P^{(1)}, P^{(2)}$ are two multi-linear tensor programs that both take inputs as tensors $\vec{T} = (T_1, T_2, T_3, \cdots, T_J)$ and positions $\vec{v} = (v_1, v_2, \cdots v_m) \in \mathcal{V} \subseteq \mathbb{N}^m$, and each defined by sets $\mathcal{B}^{(1)}, \mathcal{B}^{(2)}$ and linear functions $L_{j,k}^{(1)}, L_{j,k}^{(2)}$ respectively. Suppose there exists an $\vec{v}^\star \in \mathcal{V}$ such that*

$$\exists \vec{T}^\star: \; P^{(1)}(\vec{T}^\star)[\vec{v}^\star] \neq P^{(2)}(\vec{T}^\star)[\vec{v}^\star] \tag{1.24}$$

Then for every integer $K \geq 1$, let $\vec{T} = (T_1, T_2, T_3, \cdots, T_J)$ be an array of random tensors where each entry of the tensors is sampled i.i.d. uniformly at random from $[K]$, then we must have:

$$\mathop{\mathbf{Pr}}_{\vec{T}}\left[P^{(1)}(\vec{T})[\vec{v}^\star] \neq P^{(2)}(\vec{T})[\vec{v}^\star]\right] \geq 1 - \frac{J}{K} \tag{1.25}$$

The theorem is an immediate corollary of the Schwartz-Zippel Lemma:

Lemma 1.4 (Schwartz, Zippel)**.** *For any field $\mathbb{F}$, let P be a non-zero polynomial over $\mathbb{F}^n$ with total degree at most d. Let $\mathcal{S} \subseteq \mathbb{F}$ be a finite set, then we have: let $x_1, x_2, \cdots, x_n$ i.i.d. sampled uniformly at random from $\mathcal{S}$, then*

$$\mathbf{Pr}[P(x_1, x_2, \cdots, x_n) = 0] \leq \frac{d}{|\mathcal{S}|} \tag{1.26}$$