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

- Submitted conference/journel name/years: Published as a conference paper at OSDI, 2021.
- Author: Haojie Wang, Jidong Zhai, Mingyu Gao et al.
- Presented date: 2022.03.29.

Presented by: JunYeong Park

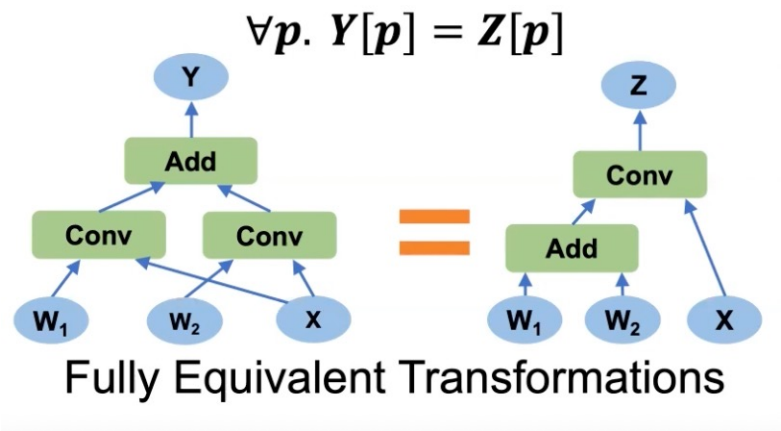
Student ID: 72220216

Content

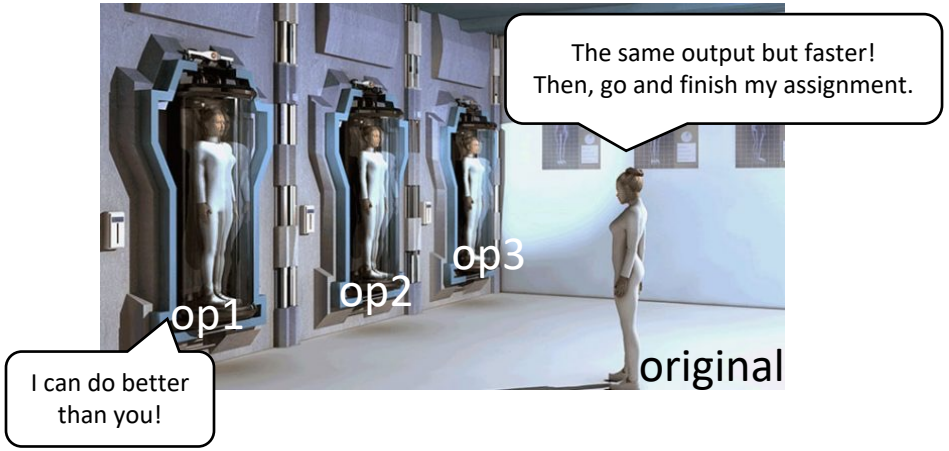
Introduction

1. Limitation of existing frameworks

- Optimize the tensor programs applying 'Fully Equivalent Transformation'
- Tensor Program: direct acyclic computation graphs describing the operations applied to a set of tensors
- Fully Equivalent Transformation: the new subprogram is mathematically equivalent to the original subprogram for arbitrary inputs



➔ Miss opportunities to optimize tensor programs using transformation method



Introduction

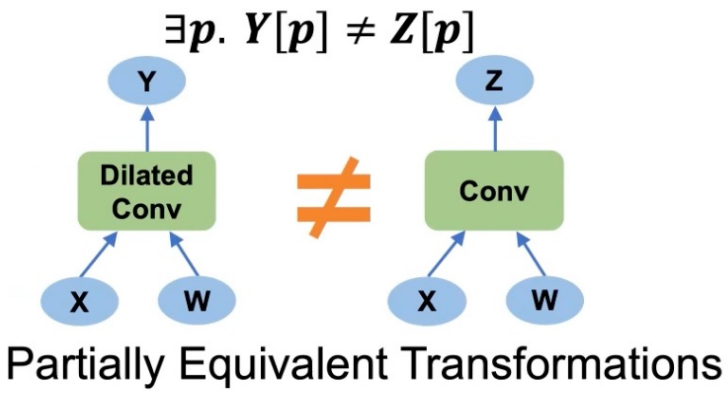
2. Partially Equivalent Transformation

- Transformation that do not preserve full equivalence on all elements of the output tensors.
- Example:

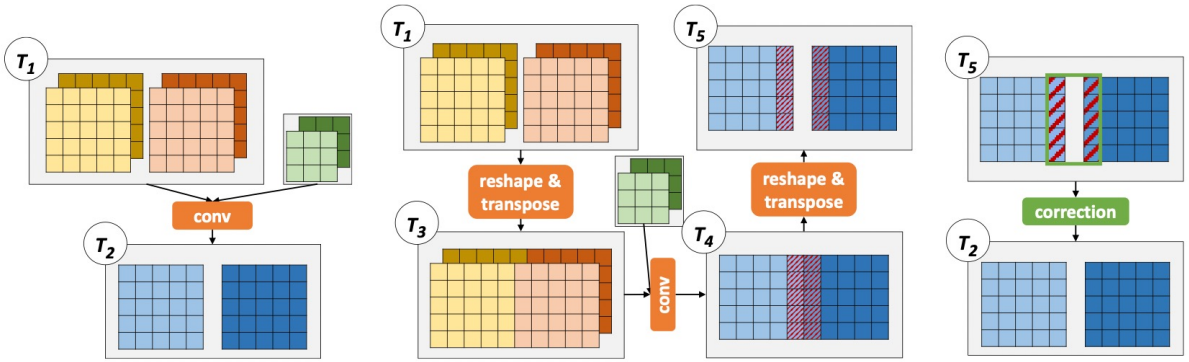
  Changing the shape or linearization ordering of input tensors

  Replacing less efficient operators with more optimized operators with similar mathematical behavior

  Transforming the graph structure of a program
- Improve computational efficiency and optimize the performance



Introduction



(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.

3. Challenges

- Directly applying partially equivalent transformation makes different output tensors from output tensors of original input program(applying only fully equivalent transformation).
- Different output tensors may adversely affect performance.
- So applying correction kernels is necessary. But it is difficult to examine the equivalence and generate correction kernels.

Introduction

4. Contribution

- First attempt in tensor program optimization to exploit partially equivalent transformation with automated corrections.
- Theoretical foundations that simplify the equivalence examination and correction kernel generation.
- Efficient generation and optimization approaches to explore the large design space automatically with both fully and partially equivalent transformations.
- Implementation of new framework PET. (2.5x speedup)

5. PET architecture component

- Mutation Generator: generate *mutants*
- Mutation Corrector: examine the equivalence and automatically generates correction kernel
- Program Optimizer: identify mutant candidates with high performance and post-optimize them

Design Overview

1. MLTP (Multi-linear Tensor Program)

- A program p, in which all operators op are multi-linear.

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

- DNN subprogram

  = Multi-linear tensor operator + non-linear operator

2. PET Overview

- Split input program to be optimized based on a non-linear operator.
- Mutation generator discovers partially equivalent transformation by generating mutants for MLTPs in the subprogram.
- Mutation corrector examines the equivalence between a mutant and its original MLTP, and generates correction kernels.
- Corrected mutants are sent to program optimizer.
- Program optimizer combines f.e.t with p.e.t and post-optimize it.

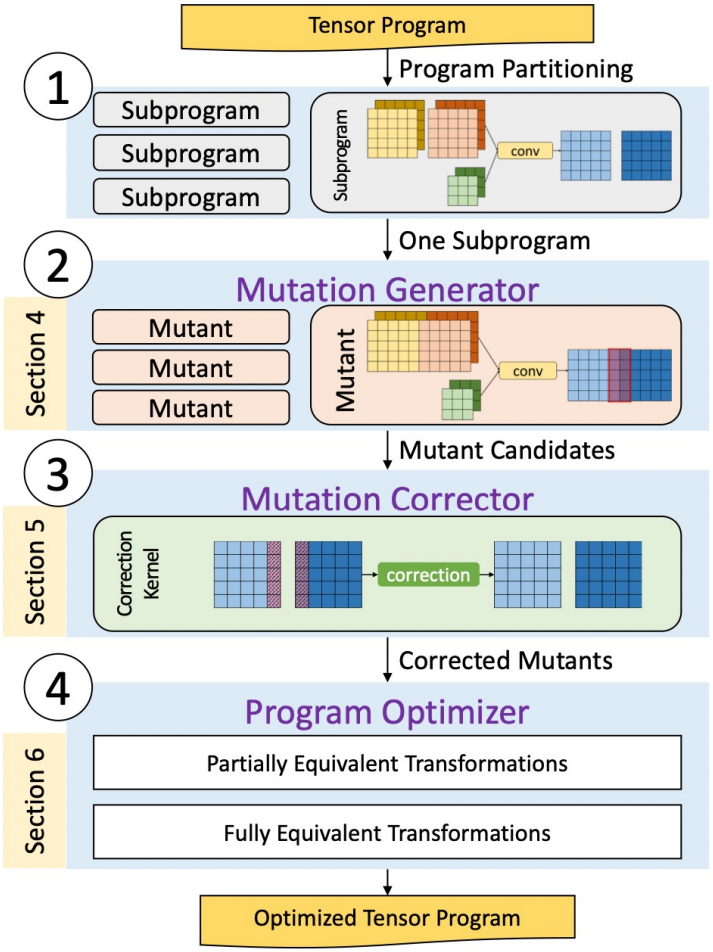


Figure 2: PET overview.

Mutation Generator

Depth-first search algorithm & check the shape

- PET iteratively adds a new operator to current program *p* by enumerating the type of operator from *O* and the input tensors to the operator.

- Enumerates all potential MLTPs up to *depth*.
  For each mutant *p*, PET checks whether *p* and *p_0* have the same number and shapes of inputs/outputs.
  p is a valid mutant if it passes this test.

Algorithm 1 MLTP mutation generation algorithm.

Input: A set of operators O; an input MLTP $\mathcal{P}_0$
Output: A set of valid program mutants $\mathcal{M}$ for $\mathcal{P}_0$
I_0 = the set of input tensors in $\mathcal{P}_0$
$\mathcal{M} = \varnothing$
BUILD(1, $\varnothing$, I_0)
// Depth-first search to construct mutants
function BUILD(n, $\mathcal{P}$, I)
    if $\mathcal{P}$ and $\mathcal{P}_0$ have the same input/output shapes **then**
        $\mathcal{M} = \mathcal{M} + \{\mathcal{P}\}$
    if $n <$ *depth* **then**
        for $op \in O$ **do**
            for $i \in I$ and i is a valid input to op **do**
                Add operator op into program $\mathcal{P}$
                Add the output tensors of op into I
                BUILD($n+1$, $\mathcal{P}$, I)
                Remove operator op from $\mathcal{P}$
                Remove the output tensors of op from I
return $\mathcal{M}$

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

Mutation Corrector

1. Why needs?

- Mutations resulted from partially equivalent transformations potentially leads to accuracy loss.
- Therefore, PET preserve the statistical behavior and guarantees the same model accuracy.

2. Specifically what does it do?

- Input: an MLTP *p_0*, mutant *p*
- Automatically generates correction kernels and applies them to the output tensors to maintain equivalence.

3. Challenges

- The output tensors may be very large. → infeasible to verify every single elements of the outputs.
- Numerically enumerating all possible values for many input variables is impractical.

Mutation Corrector

4. Proposal

- Only needs to verify a few representative output positions with a small number of randomly generated input values.

5. Theoretical Foundations

- Assumption: input MLTP *p_0* and its mutant *p* each has one output.

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

- $I = (I_1, \ldots, I_n)$: n input tensors
- $I_j[\vec{u}]$: input value at position $\vec{u}$ of I_j
- $p(I)$: output tensor of running p on *n* input tensors
- $p(I)[\vec{v}]$: output value at position $\vec{v}$
- $R(\vec{v})$: the summation interval of $\vec{v}$
  ($p(I)[\vec{v}]$ 이거 계산할때 반복될 간격)
- $\vec{u} = \mathbf{L}_j(\vec{v}, \vec{r})$: linear mapping from $(\vec{v}, \vec{r})$ to a position $\vec{u}$ of the j-th iput

Mutation Corrector

5-1. Example (3x3 convolution, zero padding)

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

(D: the number of channels, H: height, W: width, I_1: input image, I_2: kernel)

- Group the output positions with identical summation interval into a box

➔ A box is a region of an output tensor whose elements all have the same summation intervals (9 boxes)

- All output positions in the same box have an identical summation interval and share similar mathematical properties.

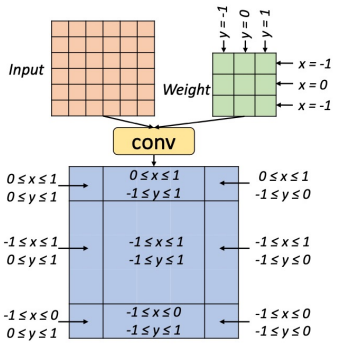


Figure 4: The nine boxes of a convolution with a 3×3 kernel and zero padding, as well as their summation intervals. A convolution has three summation dimensions (i.e., d, x, and y in Equation (1)). The channel dimension (i.e., d) has the same internal in all boxes and is thus omitted.

Mutation Corrector

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 $\forall I, \vec{v} \in \mathcal{B}$, $\mathcal{P}_1(I)[\vec{v}] = \mathcal{P}_2(I)[\vec{v}]$.

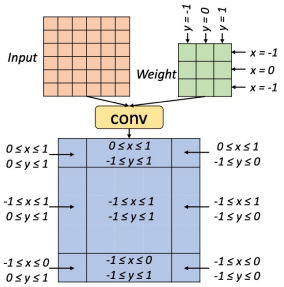
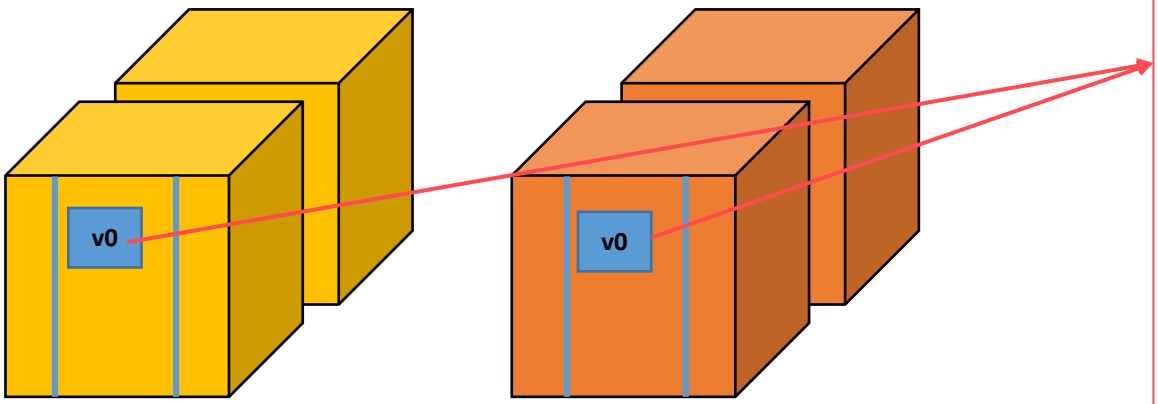


Figure 4: The nine boxes of a convolution with a 3×3 kernel and zero padding, as well as their summation intervals. A convolution has three summation dimensions (i.e., d, x, and y in Equation (1)). The channel dimension (i.e., d) has the same internal in all boxes and is thus omitted.

- We want to check the equivalence of two output tensors

  → choose one random position for each box (v0) → compare p_0(v0~v(m+1)) == p(v0~v(m+1))

- If all position in the If the values in the m+1 positions are all the same, the values in the overlapping box ensure equivalence.



If 4 dimensional tensor

(batch, channel, width, height)

If,

One random position v0 = (1, 1, 3, 4)

v1 = (2, 1, 3, 4) v2= (1, 2, 3, 4)

v3 = (1, 1, 4, 4) v4 = (1, 1, 3, 5)

So check 5 position v0~v4 in 4 dim tensor

Mutation Corrector

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 $\mathbb{F}$.*

- n/p = $(\frac{1}{2^{31}})^t$

  (in t random integer)

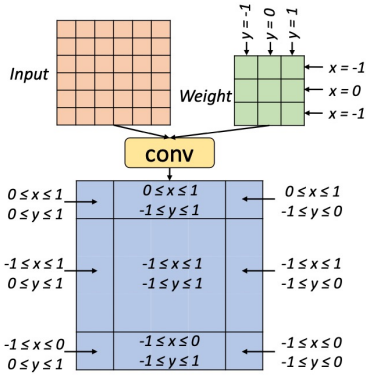


Figure 4: The nine boxes of a convolution with a 3×3 kernel and zero padding, as well as their summation intervals. A convolution has three summation dimensions (i.e., d, x, and y in Equation (1)). The channel dimension (i.e., d) has the same internal in all boxes and is thus omitted.

- If program *p* and *p_0* are not equivalent,

  the probability that the output values of the two programs are the same when random inputs are inserted is very low.

- Examine the equivalence for each pair of **overlapped boxes** from the two MLTPs, **using a small number of random tests**

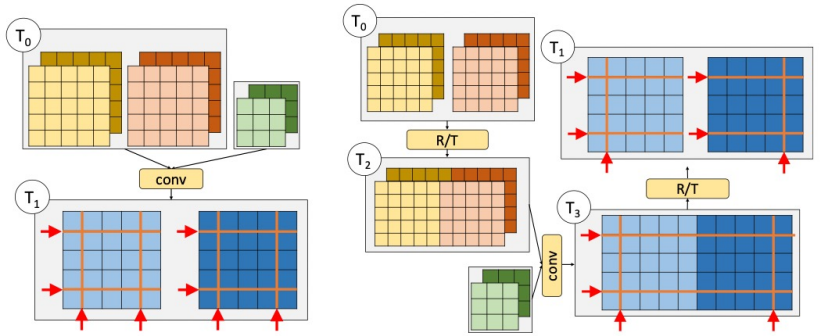
Mutation Corrector

6. Mutation Correction Algorithm

(1) Box Propagation

- Infer the split points of its output tensors based on the split points of tis input tensors and the operator type and hyper-parameter.

- a set of split points for each dimension of a tensor
  → identify the boundaries of its boxes.



(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.

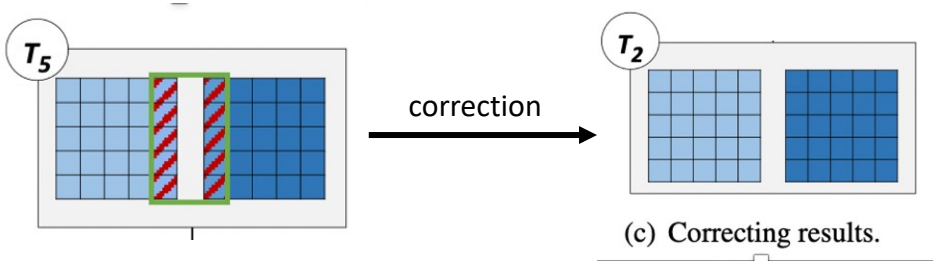
(2) Random testing for each box pair

- If two boxes do not have any overlapped region, the can be skipped. (not equivalent)
- **Examines the equivalence of the two programs on m+1 positions identified by Theroem1**
  (4 dimensional output tensor → v0 ~ v5)
- 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 F.
- caching optimization: all random inputs for the tests are the same.

Mutation Corrector

6. Mutation Correction Algorithm

(3) Correction kernel generation



(c) Correcting results.

- For each box failing the random tests, PET generates correction kernels to maintain original statistical behavior (mathematical equivalence)
- 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.
- Kernel generation leverage existing libraries and techniques.
- To reduce the correction overhead, PET opportunistically fuses the correction kernels with existing tensor operators.

Mutation Corrector

7. Fusing Correction Kernels

- Launching the correction kernels → substantial overhead
  → eliminate the advantage of partially equivalent transformation (performance gain)

- Opportunistically fuses correction kernels with other tensor operators.

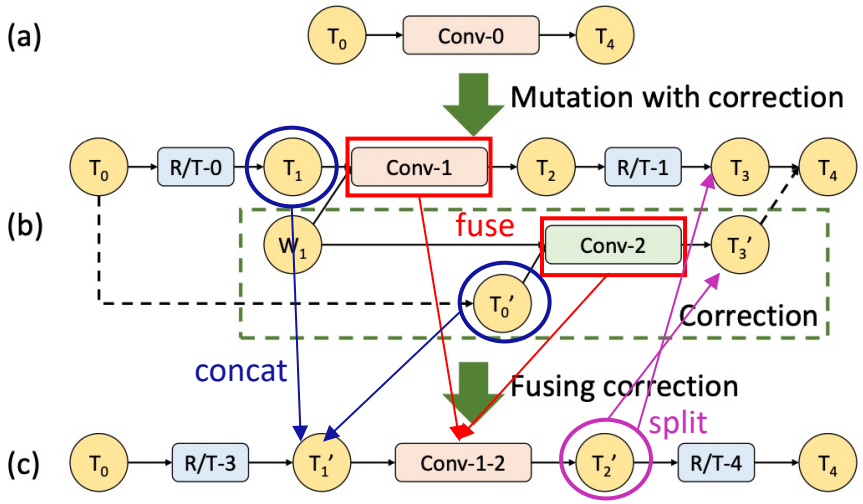


Figure 6: Fusing correction kernels with DNN kernels.

Program Optimizer

1. Program splitting

- Splits an input program into multiple disjoint subprograms with smaller sizes. (larger program → high complexity)
- Split point: non-linear operators (activation layers)

2. Subprogram optimization for best mutant

- Call mutation generator → keeping the top-K candidates
- Larger K → more memory and higher computation cost
- At each step, each of the obtained mutants replace its corresponding subprogram in each of the current candidates(*p*) to generate a new candidate(*p_new*)
- Apply a series of post-optimization

Algorithm 2 Program optimization algorithm.

Input: An input tensor program $\mathcal{P}_0$
Output: An optimized tensor program $\mathcal{P}_{\text{opt}}$

Split $\mathcal{P}_0$ into a list of subprograms
Initialize a heap $\mathcal{H}$ to record the top-K programs
$\mathcal{H}$.insert($\mathcal{P}_0$)
// Greedily mutate each subprogram
for each subprogram $\mathcal{S} \in \mathcal{P}_0$ **do**
    mutants = GETMUTANTS($\mathcal{S}$)
    Initialize a new heap $\mathcal{H}_{\text{new}}$
    for $\mathcal{P} \in \mathcal{H}$ **do**
        for $\mathcal{M} \in$ *mutants* **do**
            $\mathcal{P}_{\text{new}}$ = replace $\mathcal{S}$ with $\mathcal{M}$ in $\mathcal{P}$
            Apply post-optimizations on $\mathcal{P}_{\text{new}}$
            $\mathcal{H}_{\text{new}}$.insert($\mathcal{P}_{\text{new}}$)
    $\mathcal{H} = \mathcal{H}_{\text{new}}$
$\mathcal{P}_{\text{opt}}$ = the program with the best performance in $\mathcal{H}$
return $\mathcal{P}_{\text{opt}}$

function GETMUTANTS($\mathcal{S}_0$)
    $\mathcal{O}$ = the set of mutant operators for $\mathcal{S}_0$
    $\mathcal{Q} = \{\mathcal{S}_0\}$, *mutants* = $\{\mathcal{S}_0\}$
    for r rounds **do**
        $\mathcal{Q}_{\text{new}} = \{\}$
        for $\mathcal{S} \in \mathcal{Q}$ **do**
            for each subset of operators $\mathcal{S}' \in \mathcal{S}$ **do**
                for $\mathcal{M}' \in$ MUTATIONGENERATOR($\mathcal{O}, \mathcal{S}'$) **do**
                $\mathcal{M}$ = replace $\mathcal{S}'$ with $\mathcal{M}'$ in $\mathcal{S}$
                Add $\mathcal{M}$ to $\mathcal{Q}_{\text{new}}$ and *mutants*
        $\mathcal{Q} = \mathcal{Q}_{\text{new}}$
    return *mutants*

Program Optimizer

3. Post Optimizations

- The optimized mutant for all subprograms need to be stitched together.
- Connecting their input and output tensors
  → fuse R/T operators across subprograms
  → fuse the non-linear operators
- [fuse R/T operators] Figure 7(b): Group reshape/transpose operators between subprograms by reordering the R/T operators with non-linear activations.
- [fuse non-linear operators] Figure 7(c): Conv+ReLU = Conv-ReLU

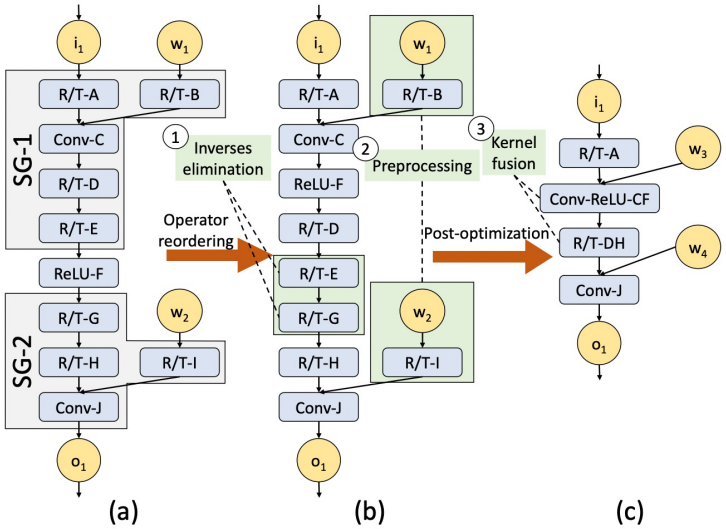


Figure 7: Post-optimizations applied when stitching two sub-programs `SG-1` and `SG-2`. `R/T` refers to a `reshape` followed by a `transpose`. `Conv` and `ReLU` denote a convolution and a ReLU operator, respectively.

[3 post optimization]

- Inverse elimination: eliminate any pairs of R/T operators(called inverse group) that can be cancel out each other.
- Operator fusion:
  Fuses remaining R/T operators into a single operator to reduce the kernel launch cost.
  non-linear activations in a tensor program are fused with R/T or other linear operations.
- Preprocessing: preprocess any operator if all its input tensors are statically known.

Conclusion

- 5 DNN Architectures: ResNet-18 (Image Classification), CSRNet (semantic segmentation), Inception-v3, BERT (language representation), Resnet3D-18 (video processing)
- Default mutation generation depth = 4 (Algorithm 1. DFS)
  search rounds = 4 (Algorithm 2, Greedy search)

1. End-to-end inference performance

- Batch size under 1 and 16
- Larger batch
  → relatively higher performance

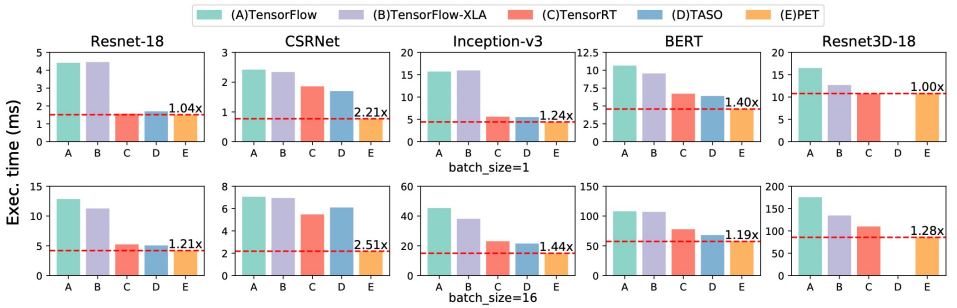


Figure 8: End-to-end performance comparison between PET and existing frameworks. For each DNN, the numbers above the PET bars show the speedups over the best baseline. TASO does not support the 3D convolution operators in Resnet3D-18.

2. Partial equivalent transformations discovered by PET

- Manually modify TASO as adding partial equivalent transformations and corresponding correction kernels.
- Evaluate the end-to-end inference performance of TASO compared to original TASO and PET.
- 'TASO + PET' transformations and corrections' is relatively faster than original TASO.

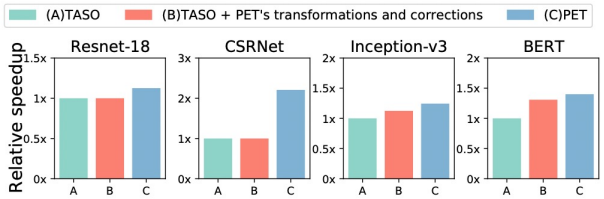


Figure 9: Performance benefits after adding PET's partially equivalent transformations into TASO.

Thank You For Listening