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

Tudor Andrei
Ana Batrineanu

# Background: DNN

- Deep neural networks (DNNs) are widely used today in AI models.
- It is common to view a DNN as a computation graph, which describes the operations that the model applies on the input tensors.

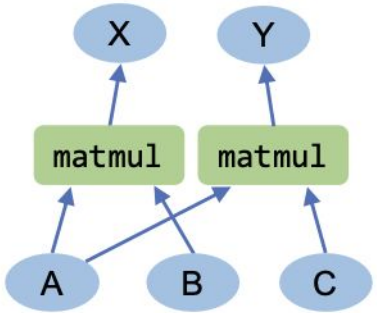



Figure 1: Computation graph
In blue: tensors
In green: operators

# Background: Optimizing DNN

- A fresh DNN may not be very performant
- DNNs can be optimized by substituting subgraphs for equivalent, faster ones

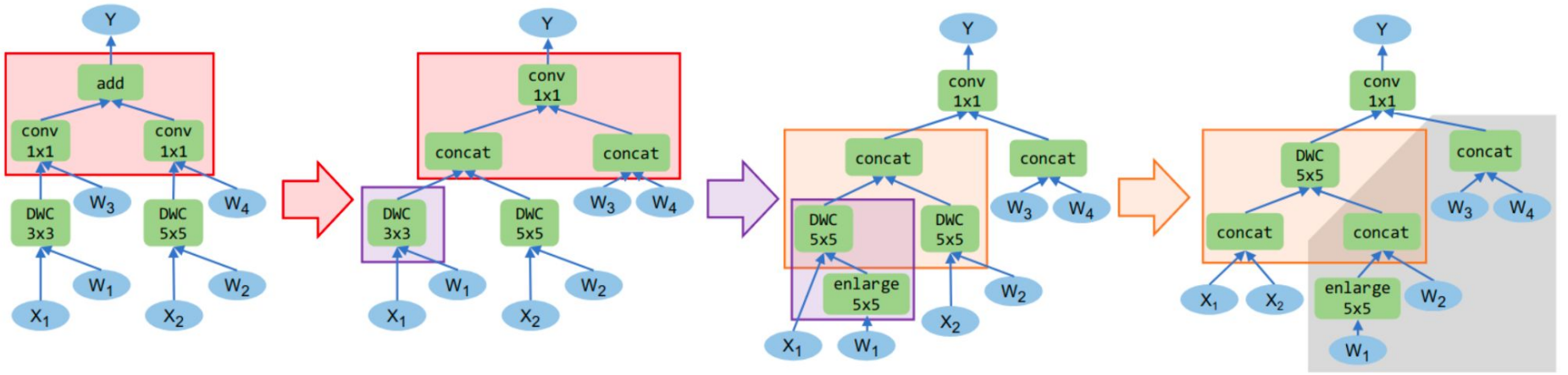



Figure 2

# Background

- There is a lot of effort put into optimizing deep neural network (DNN) architectures.
- Common frameworks like TensorRT, PyTorch or TensorFlow do this by manually applying graph transformations, which are designed by human experts.
- This approach misses potential graph optimizations and it is difficult to scale.
- Handwritten graph substitutions requires significant engineering effort.
- Moreover, addition of new operators requires manual effort for finding out suitable substitutions.

# TASO

- TASO is the first DNN computation graph optimizer that automatically generates graph substitutions in an offline manner.
- A substitution is represented as a source and target graph. The source graph can be replaced with the target, preserving the same operational outcome but with improved runtime performance.

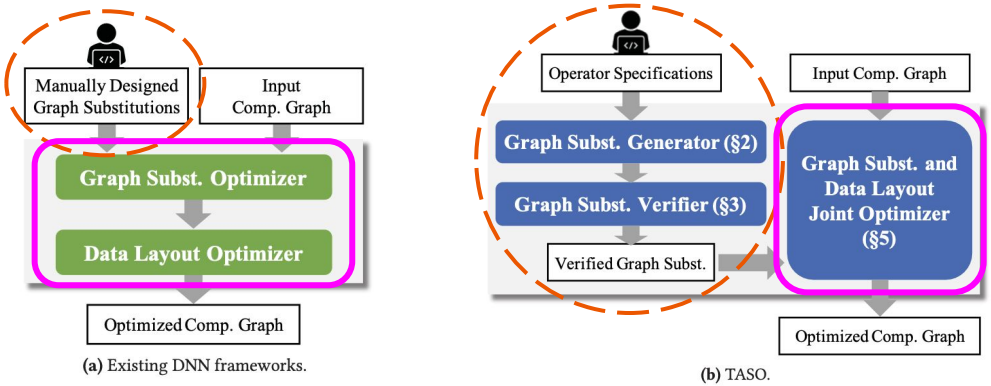



Figure 3

# Operators

- TASO already supports operators described in cuDNN
- However, it can extended with new operators. TASO requires a concrete implementation in C++ and symbolic implementation in Python (to validate operator specifications)

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

Table 1

# Graph Substitution Generator

- To find equivalent graphs, TASO uses **fingerprints**: each graph has a fingerprint (a hash of the mapping from input tensors to output tensors)
- Two graphs are certainly not equivalent if they have different fingerprints

**Algorithm 1** Graph substitution generation algorithm.

**Input:** A set of operators $\mathcal{P}$, and a set of input tensors $\mathcal{I}$.
**Output:** Candidate graph substitutions $\mathcal{S}$.

*// Step 1: enumerating potential graphs.*
$\mathcal{D} = \{\}$ *// $\mathcal{D}$ is a graph hash table indexed by their fingerprints.*
BUILD$(1, \emptyset, \mathcal{I})$
**function** BUILD$(n, \mathcal{G}, \mathcal{I})$
    **if** $\mathcal{G}$ contains duplicated computation **then**
        **return**
    $\mathcal{D} = \mathcal{D} +$ (FINGERPRINT$(\mathcal{G}), \mathcal{G})$
    **if** $n <$ *threshold* **then**
        **for** $op \in \mathcal{P}$ **do**
            **for** $i \in \mathcal{I}$ and $i$ is a valid input to $op$ **do**
                Add operator $op$ into graph $\mathcal{G}$.
                Add the output tensors of $op$ into $\mathcal{I}$.
                BUILD$(n + 1, \mathcal{G}, \mathcal{I})$
                Remove operator $op$ from $\mathcal{G}$.
                Remove the output tensors of $op$ from $\mathcal{I}$.

*// Step 2: testing graphs with identical fingerprint.*
$\mathcal{S} = \{\}$
**for** $\mathcal{G}_1, \mathcal{G}_2 \in \mathcal{D}$ with the same FINGERPRINT$(\cdot)$ **do**
    **if** $\mathcal{G}_1$ and $\mathcal{G}_2$ are equivalent for all test cases **then**
        $\mathcal{S} = \mathcal{S} + (\mathcal{G}_1, \mathcal{G}_2)$
**return** $\mathcal{S}$

Algorithm 1

# Graph Substitution Generator

Goal: Find equivalent subgraphs

- Enumerate computation graphs up to a certain size, using depth-first search and random tensors
  - Generate a subgraph using cuDNN operators as blocks and run the tensors on it
  - Exclude graphs that contain duplicated computation
  - Besides the random tensors, TASO also tests with some constant tensors, in order to allow finding substitutions involving constant tensors (e.g. identity matrix)

**Algorithm 1** Graph substitution generation algorithm.

**Input:** A set of operators $\mathcal{P}$, and a set of input tensors $\mathcal{I}$.
**Output:** Candidate graph substitutions $\mathcal{S}$.

*// Step 1: enumerating potential graphs.*
$\mathcal{D} = \{\}$ // $\mathcal{D}$ is a graph hash table indexed by their fingerprints.
BUILD$(1, \emptyset, \mathcal{I})$
**function** BUILD$(n, \mathcal{G}, \mathcal{I})$
  **if** $\mathcal{G}$ contains duplicated computation **then**
    **return**
  $\mathcal{D} = \mathcal{D} +$ (FINGERPRINT$(\mathcal{G}), \mathcal{G})$
  **if** $n <$ *threshold* **then**
    **for** $op \in \mathcal{P}$ **do**
      **for** $i \in \mathcal{I}$ and $i$ is a valid input to $op$ **do**
        Add operator $op$ into graph $\mathcal{G}$.
        Add the output tensors of $op$ into $\mathcal{I}$.
        BUILD$(n + 1, \mathcal{G}, \mathcal{I})$
        Remove operator $op$ from $\mathcal{G}$.
        Remove the output tensors of $op$ from $\mathcal{I}$.

*// Step 2: testing graphs with identical fingerprint.*
$\mathcal{S} = \{\}$
**for** $\mathcal{G}_1, \mathcal{G}_2 \in \mathcal{D}$ with the same FINGERPRINT$(\cdot)$ **do**
  **if** $\mathcal{G}_1$ and $\mathcal{G}_2$ are equivalent for all test cases **then**
    $\mathcal{S} = \mathcal{S} + (\mathcal{G}_1, \mathcal{G}_2)$
**return** $\mathcal{S}$

Algorithm 1

# Graph Substitution Generator

Goal: Find equivalent subgraphs

- For each graph, collect its fingerprint
- All equivalent computation graphs have the same fingerprint
- Run more test cases on the pairs found with matching fingerprints
- Each test case contains a set of randomized input tensors
- The two graphs are equivalent if they produce equivalent output tensors for all test cases (they differ by no more than a very small threshold value)
- All candidate graph substitutions that passed this stage are sent to the substitution verifier to check their correctness.
- Later we can apply substitutions discovered by TASO on real models.

**Algorithm 1** Graph substitution generation algorithm.

**Input:** A set of operators $\mathcal{P}$, and a set of input tensors $\mathcal{I}$.
**Output:** Candidate graph substitutions $\mathcal{S}$.

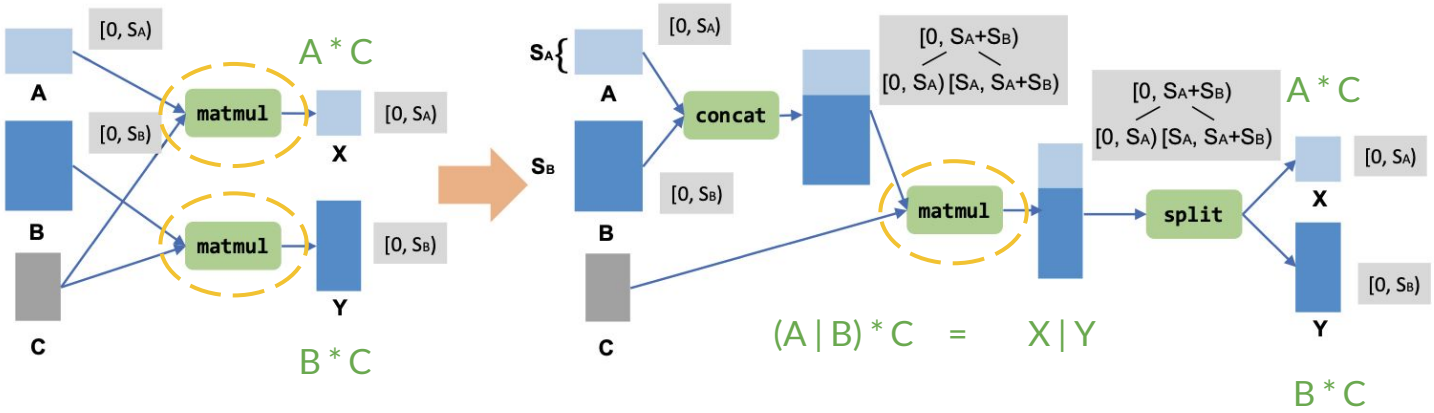
*// Step 1: enumerating potential graphs.*
$\mathcal{D} = \{\}$ *//* $\mathcal{D}$ *is a graph hash table indexed by their fingerprints.*
BUILD$(1, \emptyset, \mathcal{I})$
**function** BUILD$(n, \mathcal{G}, \mathcal{I})$
  **if** $\mathcal{G}$ contains duplicated computation **then**
    **return**
  $\mathcal{D} = \mathcal{D} +$ (FINGERPRINT$(\mathcal{G}), \mathcal{G})$
  **if** $n <$ *threshold* **then**
    **for** $op \in \mathcal{P}$ **do**
      **for** $i \in \mathcal{I}$ and $i$ is a valid input to $op$ **do**
        Add operator $op$ into graph $\mathcal{G}$.
        Add the output tensors of $op$ into $\mathcal{I}$.
        BUILD$(n + 1, \mathcal{G}, \mathcal{I})$
        Remove operator $op$ from $\mathcal{G}$.
        Remove the output tensors of $op$ from $\mathcal{I}$.

*// Step 2: testing graphs with identical fingerprint.*
$\mathcal{S} = \{\}$
**for** $\mathcal{G}_1, \mathcal{G}_2 \in \mathcal{D}$ with the same FINGERPRINT$(\cdot)$ **do**
  **if** $\mathcal{G}_1$ and $\mathcal{G}_2$ are equivalent for all test cases **then**
    $\mathcal{S} = \mathcal{S} + (\mathcal{G}_1, \mathcal{G}_2)$
**return** $\mathcal{S}$

Algorithm 1

# Graph Substitution Generator

Example of substitution




source graph

target graph

Figure 4

# Graph Substitution Verifier

**How does TASO prove the equivalence of substitutions?**

- We know some mathematical and logical properties for operators: e.g: convolution without activation is linear in its arguments: conv(x+y, k) = conv(x,k) + conv(y,k).
- TASO uses 43 operators properties in total.
- As evaluation shows, a small set of properties for each operator suffices to prove the correctness of complex substitutions.

| | |
|---|---|
| $\forall x.\ \texttt{transpose}(\texttt{transpose}(x)) = x$ | `transpose` is its own inverse |
| $\forall x, y.\ \texttt{transpose}(\texttt{ewadd}(x, y)) = \texttt{ewadd}(\texttt{transpose}(x), \texttt{transpose}(y))$ | operator commutativity |
| $\forall x, y.\ \texttt{transpose}(\texttt{ewmul}(x, y)) = \texttt{ewmul}(\texttt{transpose}(x), \texttt{transpose}(y))$ | operator commutativity |
| $\forall x, w.\ \texttt{smul}(\texttt{transpose}(x), w) = \texttt{transpose}(\texttt{smul}(x, w))$ | operator commutativity |
| $\forall x, y, z.\ \texttt{matmul}(x, \texttt{matmul}(y, z)) = \texttt{matmul}(\texttt{matmul}(x, y), z)$ | `matmul` is associative |
| $\forall x, y, w.\ \texttt{smul}(\texttt{matmul}(x, y), w) = \texttt{matmul}(x, \texttt{smul}(y, w))$ | `matmul` is linear |
| $\forall x, y, z.\ \texttt{matmul}(x, \texttt{ewadd}(y, z)) = \texttt{ewadd}(\texttt{matmul}(x, y), \texttt{matmul}(x, z))$ | `matmul` is linear |
| $\forall x, y.\ \texttt{transpose}(\texttt{matmul}(x, y)) = \texttt{matmul}(\texttt{transpose}(y), \texttt{transpose}(x))$ | `matmul` and `transpose` |

Table 2: Indicates some of the operator properties that are used to formally verify graph substitutions

# Graph Substitution Verifier

**How does TASO prove the equivalence of substitutions?**

- For this stage TASO uses the *symbolic* implementation of operators.
- A symbolic implementation isn't concerned with producing values, but rather expressing what the operator should do:

```
def pointwise(x, y):
    return [xi + yi for xi, yi in zip(x,
y)]
```

- In practice a symbolic implementation should work with Z3 variables as input.
- TASO will enumerate all possible combinations of operator parameters (strid, padding, etc..) and tensor input sizes (up to 4x4x4x4). For each of these combinations, it obtains a Z3 formula for a graph computation which can be verified against other formulas for equality.

# Pruning Redundant Substitutions

**A graph substitution is redundant if it is subsumed by a more general valid substitution**

Two types of pruning:

1. Tensor renaming
   - Consider the two substitutions on the right.
   - The second substitution has the same effect as the first (matrix associativity), but it is less general.
   - The less general substitution (obtained through tensor renaming) is dropped.
   - The more general substitution (in this case the first) is kept

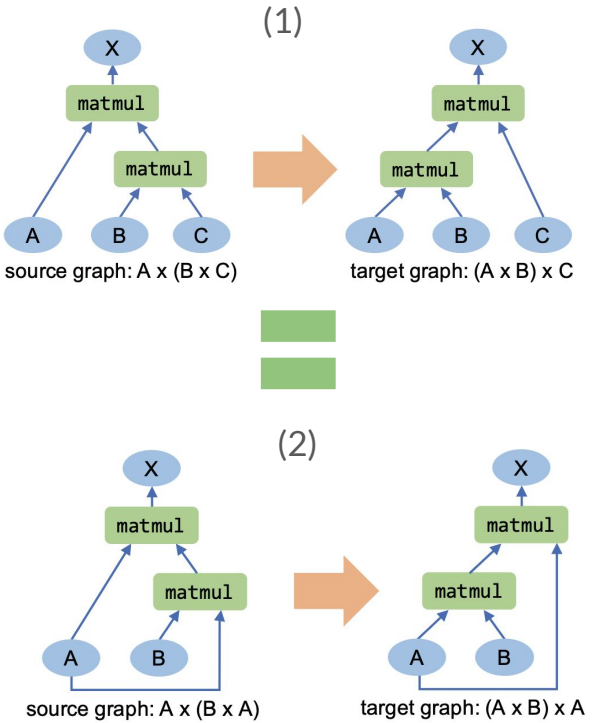


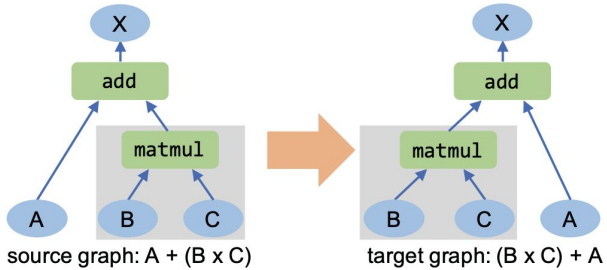

Figure 5: Pruning

(a) A redundant substitution. The two figures are equivalent by renaming input tensor C with A

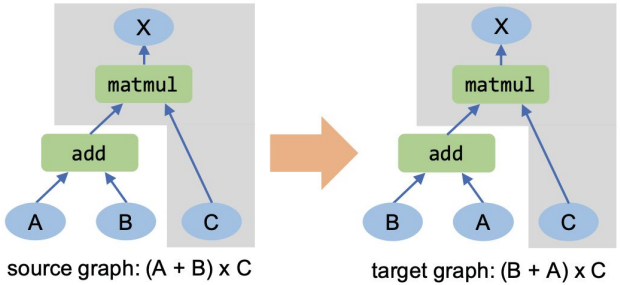
# Pruning Redundant Substitutions

2. Common subgraph

- In a substitution, the source and target may contain an identical subgraph (same operator and same input tensors). Hence, we can obtain a smaller and more general substitution by replacing the common subgraph in both the source and the target with another input tensor.




**(b)** A redundant substitution with a common subgraph.

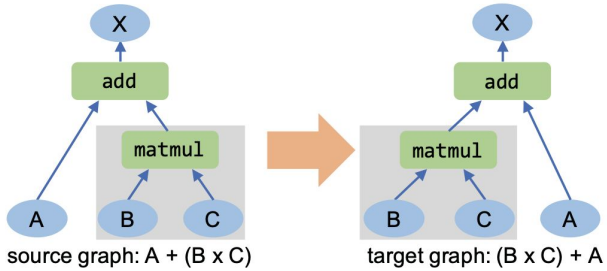



**(c)** A redundant substitution with a common subgraph.

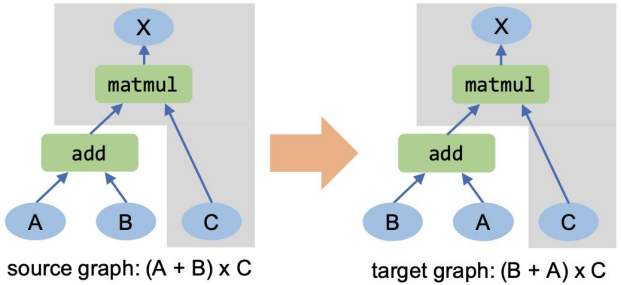
# Pruning Redundant Substitutions

2. Common subgraph

- The common subgraph includes **all the outputs** in both the source and target graphs. In this case, a more general substitution can be obtained by completely removing the common subgraph, making its inputs new outputs of the source and target graphs. We should get a smaller and more general substitution.




**(b)** A redundant substitution with a common subgraph.




**(c)** A redundant substitution with a common subgraph.

# Pruning Redundant Substitutions

- Table 2 shows the effects of TASO pruning techniques on the number of substitutions.
- We observe that both pruning techniques play an important role in eliminating redundant substitutions and their combination reduces the number of substitutions TASO must consider by 39x.

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

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

# Joint Optimizer: Graph Substitutions & Data Layout

**Aim: Find optimal graph with substitutions**

- The optimizer uses the MetaFlow cost-based backtracking search algorithm to search for an optimized computation graph by applying verified substitutions
- TASO extends MetaFlow's search algorithm to also consider possible layout optimization opportunities when performing substitutions.

**Algorithm 2** Cost-Based Backtracking Search

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

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

Algorithm 2

# Joint Optimizer: Graph Substitutions & Data Layout

**Aim: Find optimal graph with substitutions**

- Start with an initial computation graph you wish to optimize
- At each step, take the optimal graph and **apply** all possible substitutions / data layouts
- If the cost is better (<**α** ***x Cost*($G_{opt}$)**) than what the best graph so far, enqueue it
- Hyperparameter **α** tunes the backtracking:
  - 1 = search is reduced to a simple greedy without backtracking
  - 1.05 chosen for evaluation

**Algorithm 2** Cost-Based Backtracking Search

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

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

Algorithm 2

# Joint Optimizer: Graph Substitutions & Data Layout

- A non-obvious property of graph substitutions is that applying them can introduce cycles into a graph.
- **TASO checks for acyclity before enqueuing the graph.**

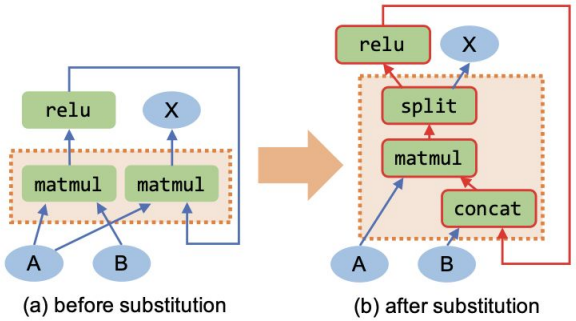



Figure 6

# Joint Optimizer: Graph Substitutions & Data Layout

## TASO Cost Function

- TASO improves MetaFlow's cost function to include data layout.
- **Cost(Operator, Layout)**
- Data Layout = Column major or Row major
- **TASO differs from current frameworks by being able to optimize over both the structure of the computation graph and data layout at the same time** (usually data layout optimization is performed after the graph optimization).

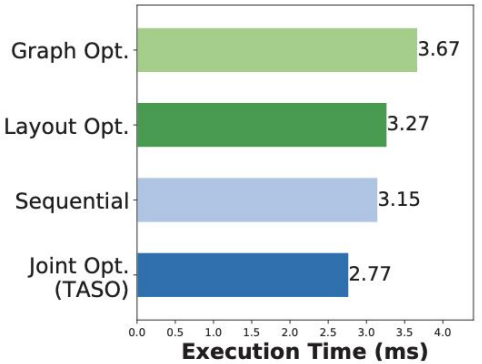



Figure 7: Performance comparison on BERT using different strategies to optimize graph substitutions and data layout. TASO outperforms the three baseline strategies by 1.2-1.3x.

# Evaluation

- Comparison of end-to-end inference performance
- Consistently better performance than alternatives

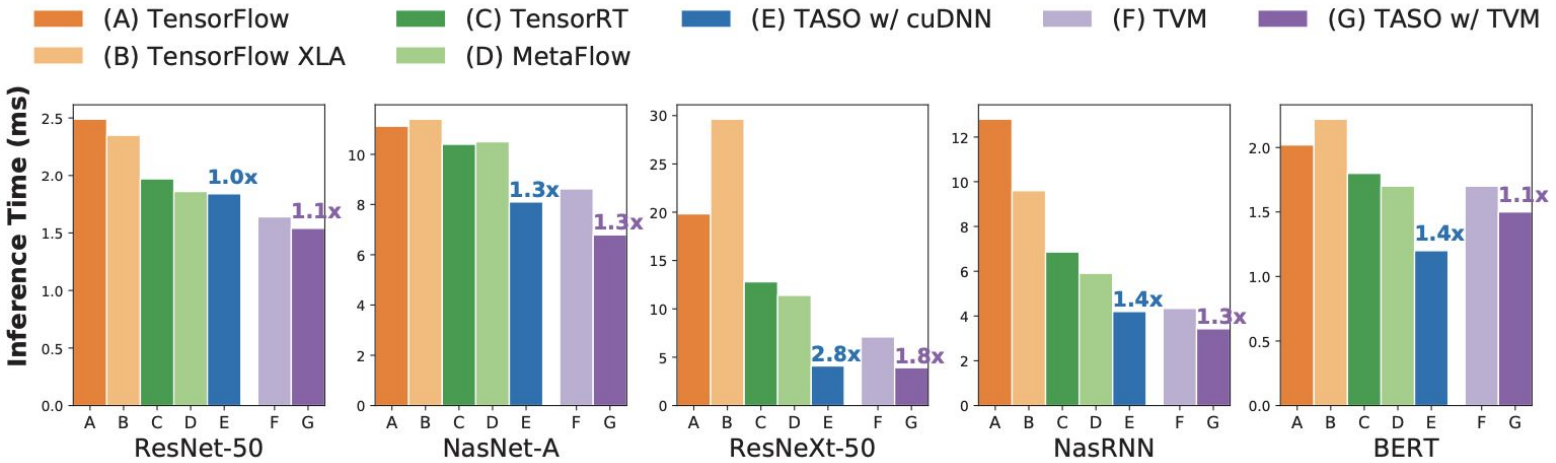



Figure 8

# Evaluation

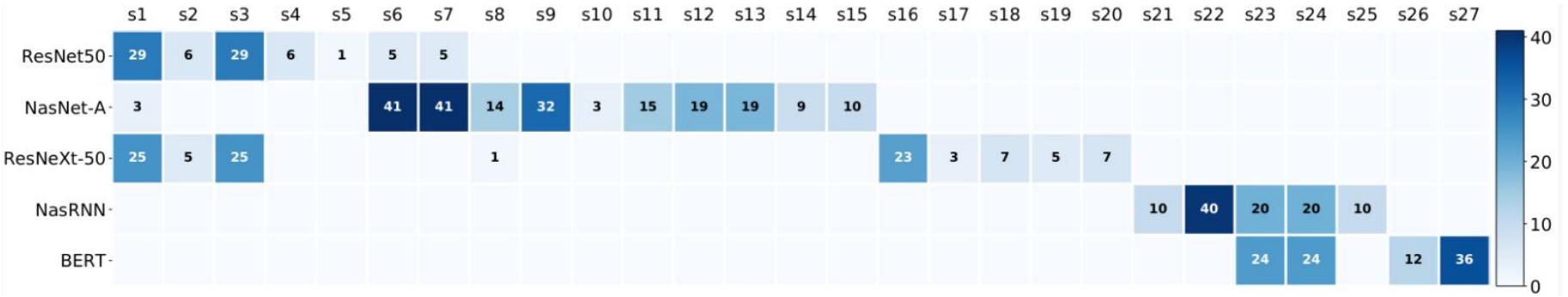



Figure 9: A heat map of how often the verified substitutions are used to optimize the five DNN architectures. Only substitutions used in at least one DNN are listed. For each architecture, the number indicates how many times a substitution is used by TASO to obtain the optimized graph.

# Summary

## Pros

- Formal verification of substitutions
- Cost-based backtracking search algorithm
- Cost function jointly optimizes graph substitutions and data layouts
- Great results: TASO outperforms existing DNN frameworks by up to 2.8x
- Low ratio of code (e.g. TensorFlow currently contains approximately 53,000 lines of manual optimization rules, while the operator specifications needed by TASO are only 1,400 lines of code).

## Cons

- From the 20k original substitutions that TASO found, after pruning and optimization it obtains 743 substitutions
- Out of the 743, only 27 ended up being used on real world models
- Project on github hasn't been maintained since 2021
- Enumeration algorithm for generating computation graphs cannot exceed graphs with size > 4 due to poor scalability.