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

By Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, Alex Aiken

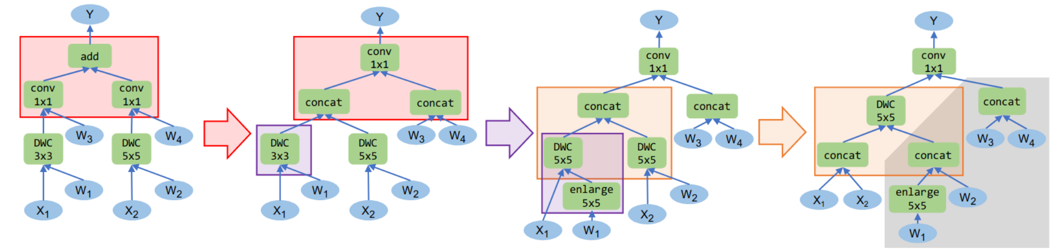
Samuel Stark

22/11/2021

University of Cambridge

Background

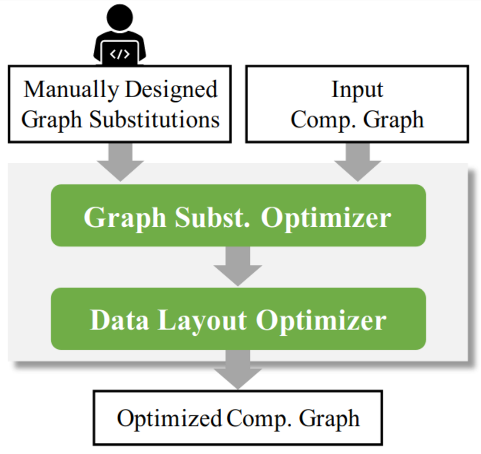
- Deep Neural Networks can be expressed as a computational graph
- A fresh DNN may not be very performant
- DNNs can be optimized by substituting subgraphs for equivalent, faster ones



Example substitution chain on NasNet-A[1, Fig 7]

TASO Concept

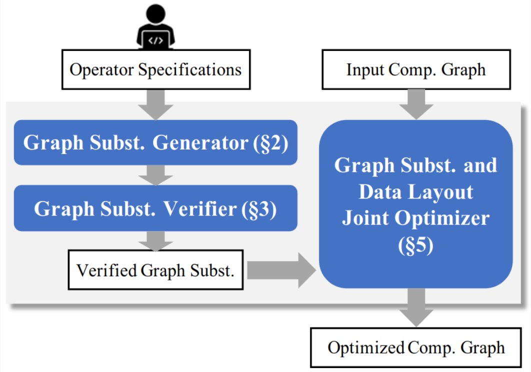
- Previous work used *manual* substitutions
- 155 substitutions = 53KLoC in TensorFlow
  - Especially bad when new operators are created
- Substitutions are not verified, may be buggy
- The graph and the data layouts are optimized separately



Previous DNN optimization flow[1, Fig 1]

TASO Concept

- TASO *automatically* generates substitutions
- 743 substitutions = 1KLoC in TASO
- Substitutions are formally proven to be correct
- The graph and data layouts are optimized *together*



TASO optimization flow[1, Fig 1]

Graph Substitution Generator

Goal: Find Equivalent Subgraphs

1. Enumerate potential graphs
   - Depth-first search, excluding duplicated computation
2. Compute Fingerprint for each graph
   - Hash outputs for constant integer input
3. Test matching-Fingerprint pairs with more data
   - Check with floating-point input, $\epsilon = 10^{-5}$

[image not found]

Graph Substitution Generator

Goal: Find Equivalent Subgraphs

1. Enumerate potential graphs
   - Depth-first search, excluding duplicated computation
2. Compute Fingerprint for each graph
   - Hash outputs for constant integer input

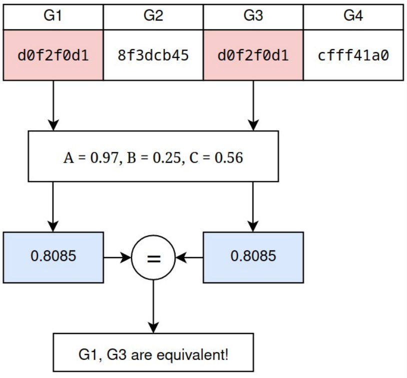
$$hash_{sym}(\{hash_{tensor}(t_i) \quad | \quad i \in \text{Outputs}\})$$

3. Test matching-Fingerprint pairs with more data
   - Check with floating-point input, $\epsilon = 10^{-5}$

Graph Substitution Generator

Goal: Find Equivalent Subgraphs

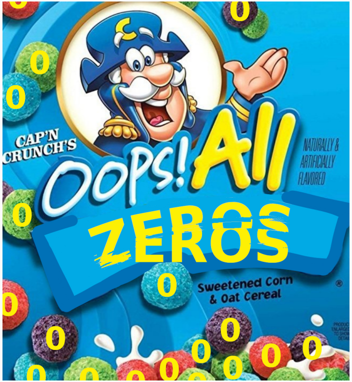
1. Enumerate potential graphs
   - Depth-first search, excluding duplicated computation
2. Compute Fingerprint for each graph
   - Hash outputs for constant integer input
3. Test matching-Fingerprint pairs with more data
   - Check with floating-point input, $\epsilon = 10^{-5}$



Goal: Find Equivalent Subgraphs

Operations that produce zeroes need more special handling:

- `relu` often returns 0 for -ve units
  - Use a diferent non-linear function
- `enlarge` literally pads with 0
  - Only allow `enlarge` on inputs, not intermediate values



Removing Redundancies

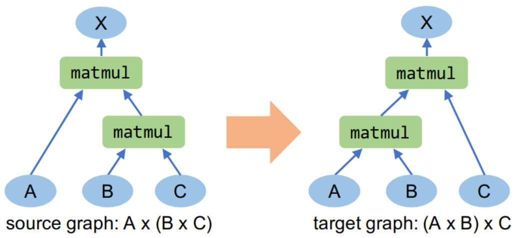
Goal: Remove Redundant/Overly Specific Substitutions

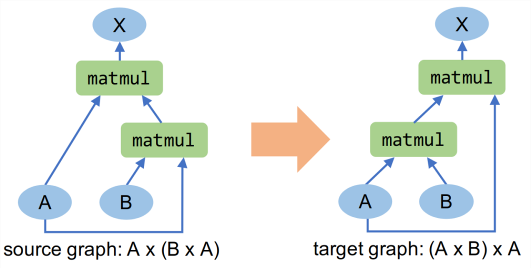
Pruning Techniques	**Remaining Substitutions**	**Reduction v.s. Initial**
Initial	28744	1×
Input tensor renaming	17346	1.7×
Common subgraph	743	39×

Table 3 from [1]

Goal: Remove Redundant/Overly Specific Substitutions

1. Remove substitutions that are identical other than input names



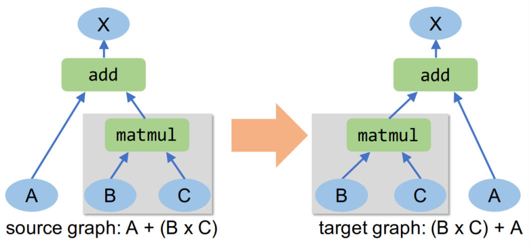


Figs 2a, 4a, b, c from [1]

Removing Redundancies

Goal: Remove Redundant/Overly Specific Substitutions

2. Remove substitutions with common subgraphs

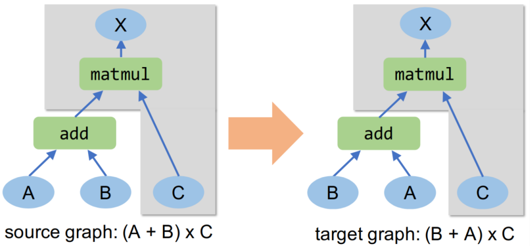


Figs 2a, 4a, b, c from [1]

Removing Redundancies

Goal: Remove Redundant/Overly Specific Substitutions

2. Remove substitutions with common subgraphs



Figs 2a, 4a, b, c from [1]

Graph Substitution Verifier

Goal: Formally Prove Substitutions are Equivalent

- Define a set of logical properties for each operator
  - 43 operators total
- Verify the operator properties hold
  - Use an SMT solver to verify the properties hold for a Python version
- Use properties to prove substitutions are equivalent
  - Use a theorem solver (Z3)

$\forall x.\ \mathtt{transpose}(\mathtt{transpose}(x)) = x$	transpose is its own inverse
$\forall x, y.\ \mathtt{transpose}(\mathtt{ewadd}(x, y)) = \mathtt{ewadd}(\mathtt{transpose}(x), \mathtt{transpose}(y))$	operator commutativity
$\forall x, y.\ \mathtt{transpose}(\mathtt{ewmul}(x, y)) = \mathtt{ewmul}(\mathtt{transpose}(x), \mathtt{transpose}(y))$	operator commutativity
$\forall x, w.\ \mathtt{smul}(\mathtt{transpose}(x), w) = \mathtt{transpose}(\mathtt{smul}(x, w))$	operator commutativity

Table 2 from [1]

Substitution + Layout Joint Optimizer

Goal: Find Optimal Graph with Substitutions

- Cost-Based Backtracking Search
  - Based on MetaFlow[2]

1. Pop graph off of priority queue
2. Try applying substitutions
3. Check costs of results
4. Push results onto queue
5. Repeat until queue is empty

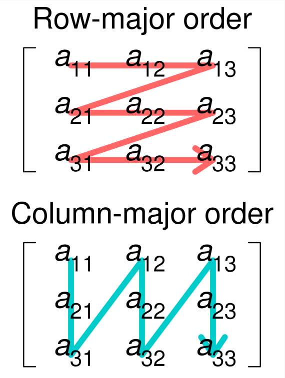
- Hyperparameter α tunes backtracking
  - 1 = No backtracking
  - 1.05 chosen for evaluation

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*.
        $\mathcal{G}' = \text{Apply}(\mathcal{G}, s, l)$
        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($\mathcal{G}'$)

Algorithm 1 from [1], based on [2]

TASO Cost Function

- TASO improves the cost function to include data layout
- $\boldsymbol{Cost(Operator, Layout)}$ measured on-device
- Data Layout = Column-Major or Row-Major
- Consider each permutation of data layouts
- $\boldsymbol{Cost(G) = \sum Cost(o_i, l_i)}$



Column/Row-Major Order
Cmglee, CC BY-SA 4.0, via Wikimedia Commons

TASO Cost Function

- TASO improves the cost function to include data layout
- $Cost(Operator, Layout)$ measured on-device
- Data Layout = Column-Major or Row-Major
- Consider each permutation of data layouts
- $Cost(G) = \sum Cost(o_i, l_i)$

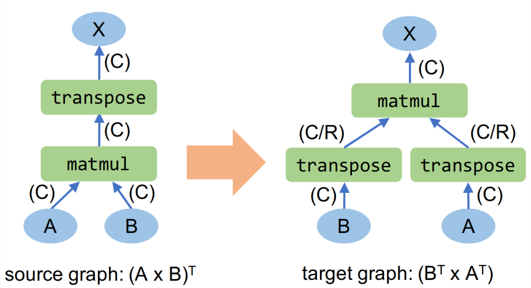


Fig 5 from [1]

TASO Cost Function

- TASO improves the cost function to include data layout
- $\mathit{Cost}(\mathit{Operator}, \mathit{Layout})$ measured on-device
- Data Layout = Column-Major or Row-Major
- Consider each permutation of data layouts
- $\mathit{Cost}(G) = \sum \mathit{Cost}(o_i, l_i)$

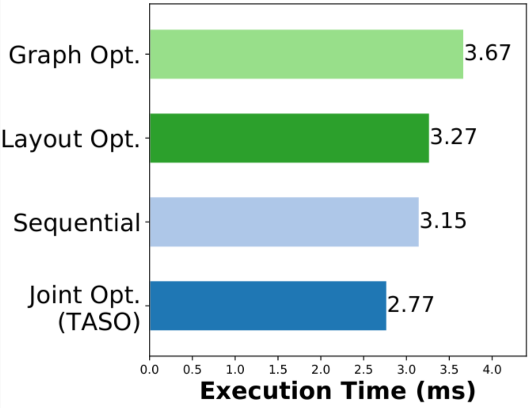
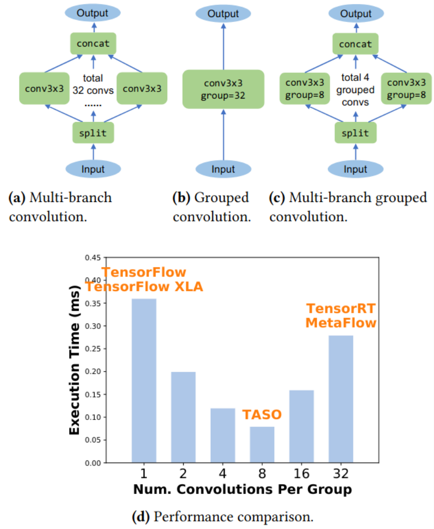


Fig 12 from [1]

Evaluation - Interesting Note

- TASO evaluates cost from real-world performance
- This allows it to find optimal strategies which might be device-specific
- But this might prevent it from mapping to distributed computing



(d) Performance comparison.

Fig 9 from [1]

Evaluation - Overall Optimization

- Consistently better performance than alternatives
  - Although they don't specify alternative optimization configs
- Only 27/743 optimizations actually used...

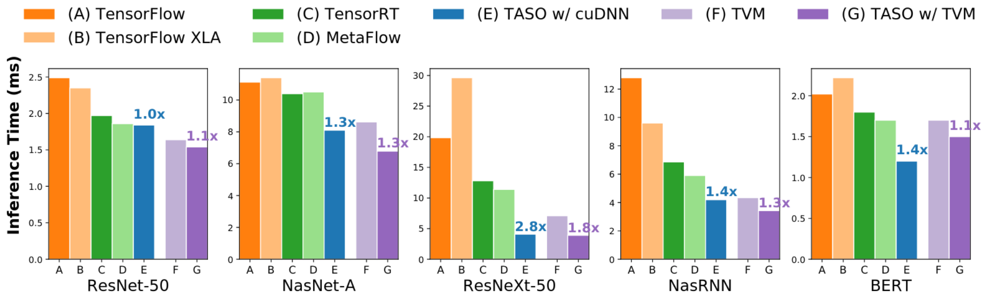


Fig 7 from [1]

Evaluation - Overall Optimization

- Consistently better performance than alternatives
  - Although they don't specify alternative optimization configs

- Only 27/743 optimizations actually used...

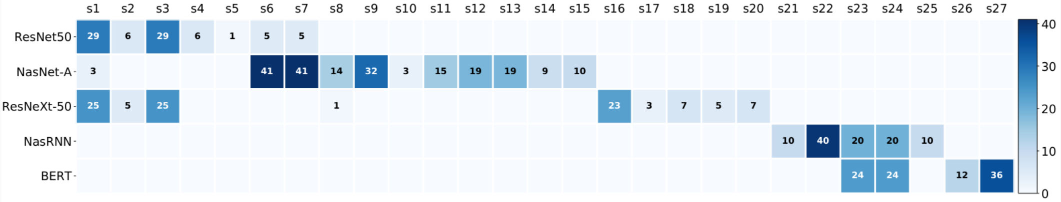


Fig 10 from [1]

Where are they now?

- Repo has 480+ GitHub stars!
- Repo is basically dead.
    - Only bugfixes since 2019
- Paper has 42 citations, was directly followed up by first author
    - Pet (next presentation!) relaxes the need for completely equivalent transformations, and then strengthens it again.
- TensorFlow has stuck with Grappler[3]
    - Applies generic optimizations
    - e.g. constant folding
    - Similar to how compilers work

Summary

Pros

- Formal verification of substitutions
- Optimizing Layout + Graph together is very cool
- Low ratio of code/optimizations
- Produces good results!

Cons

- Lots of redundancy in generated substitutions
  - Only 27 end up used at all!
- Substitutions limited to size=4
- Doesn’t evaluate time taken to optimize
- Cost model = Sum, no parallelization

Summary

Pros

- Formal verification of substitutions
- Optimizing Layout + Graph together is very cool
- Low ratio of code/optimizations
- Produces good results!

Cons

- Lots of redundancy in generated substitutions
  - Only 27 end up used at all!
- Substitutions limited to size=4
- Doesn't evaluate time taken to optimize
- Cost model = Sum, no parallelization

Questions?

References

[1] Zhihao Jia et al. "TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP '19. New York, NY, USA: Association for Computing Machinery, 27th October 2019, pp. 47–62. ISBN: 978-1-4503-6873-5. DOI: 10/gg6c64.

[2] Zhihao Jia et al. "Optimizing DNN Computation with Relaxed Graph Substitutions". In: (2019), p. 13. URL: https://cs.stanford.edu/~zhihao/papers/sysml19b.pdf.

[3] Rasmus Munk Larsen and Tatiana Shpeisman. *TensorFlow Graph Optimizations*. 2019. URL: https://research.google/pubs/pub48051.pdf.