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

Zhihao Jia, Oded Padon, James Thomas,

Todd Warszawski, Matei Zaharia, Alex Aiken

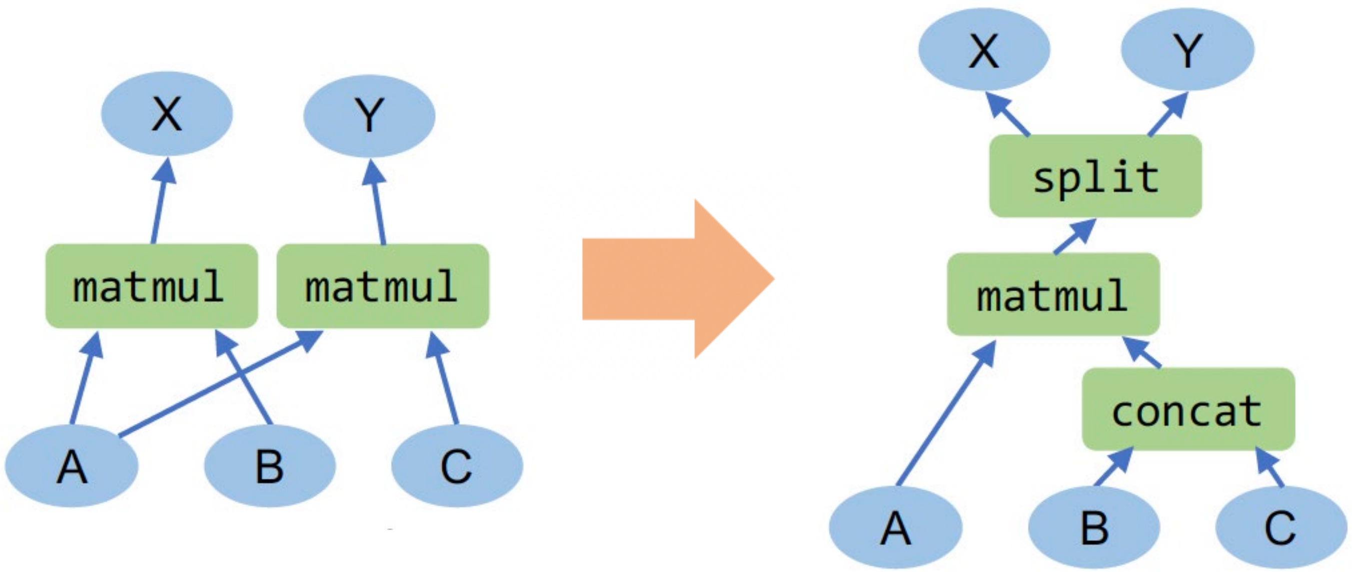
Presented By

Pranav Talluri

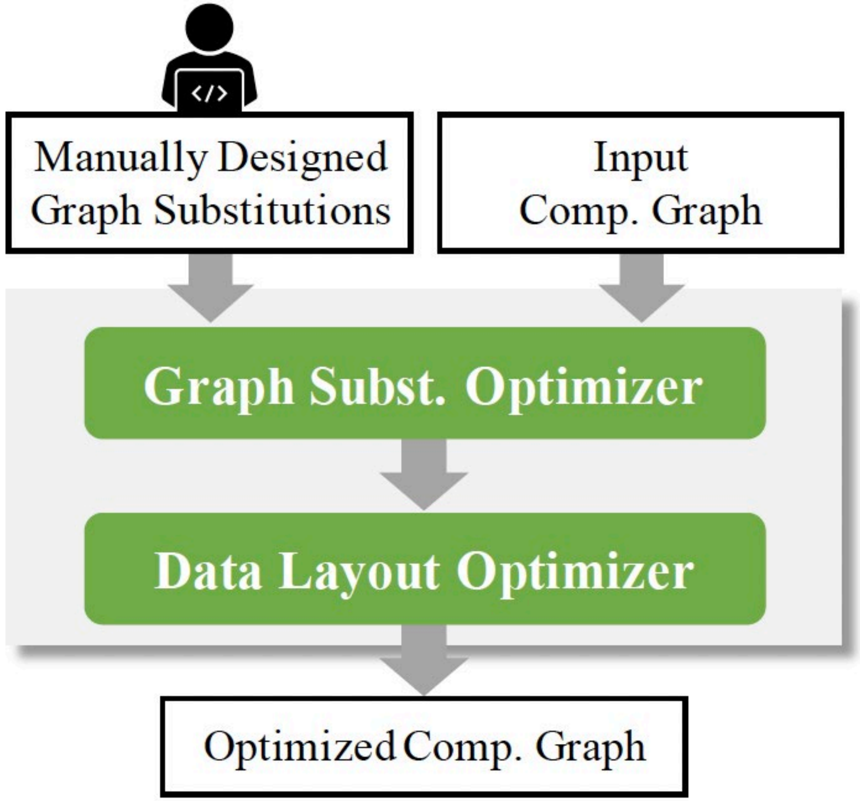
Background

- DNNs are expressed as computation graphs
- Multiple formulations can achieve the same goal, with differing costs
- Introduces the desire to optimise DNN computation graphs
- Before TASO, the specific optimisations were manually designed by human experts
- TASO automates the generation of graph substitutions in order to programmatically optimise DNN graphs

Background



Background

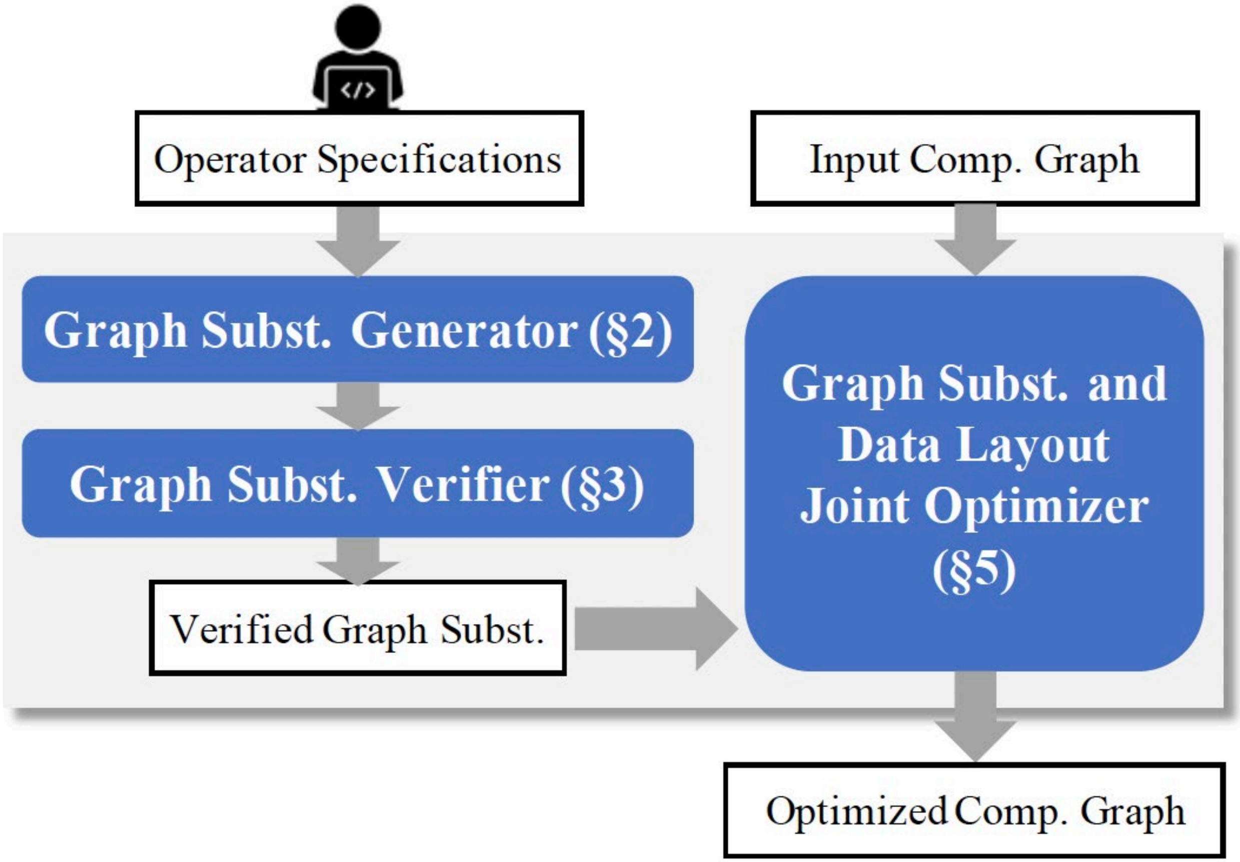


</>
Manually Designed
Graph Substitutions
Input
Comp. Graph
Graph Subst. Optimizer
Data Layout Optimizer
Optimized Comp. Graph

Overview

- TASO automates generation of graph substitutions
- Framework agnostic (cuDNN + TVM)
- Takes operator specifications as an input
- Does so in a few stages:
  - Programmatically generate candidate graph substitutions
    - Generate
    - Quick test to prune impossible substitutions
  - Formally verify validity
  - Cost-based backtracking search to find an optimised graph
    - Includes co-optimisation of data locality

Overview


Operator Specifications
Input Comp. Graph
Graph Subst. Generator (§2)
Graph Subst. Verifier (§3)
Verified Graph Subst.
Graph Subst. and Data Layout Joint Optimizer (§5)
Optimized Comp. Graph

Approach: Generate Substitutions

- Substitution = source, target, mapping
- Configuration parameter dependent operators
- Generation algorithm
  - Enumerate potential graphs
  - Create graphs iteratively
  - Collect fingerprints
  - Test graphs with identical fingerprints
- Special Cases

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.
$\textsc{Build}(1, \emptyset, \mathcal{I})$
function $\textsc{Build}(n, \mathcal{G}, \mathcal{I})$
    if $\mathcal{G}$ contains duplicated computation **then**
        return
    $\mathcal{D} = \mathcal{D} + (\textsc{FingerPrint}(\mathcal{G}), \mathcal{G})$
    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 $\mathcal{I}$.
                $\textsc{Build}(n + 1, \mathcal{G}, \mathcal{I})$
                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 $\textsc{FingerPrint}(\cdot)$ **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 $\mathcal{S}$

Approach: Formal Verification

- Verify generated substitutions
- Operator properties expressed in FOL
    - Manually written and reviewed
    - Further validated using symbolic execution
    - Properties are added when required
    - Checked for consistency and redundancies are removed
- Uses Z3 (SMT Solver)
- Shapes of tensors are not modelled
- Data layout not included

Approach: Formal Verification

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

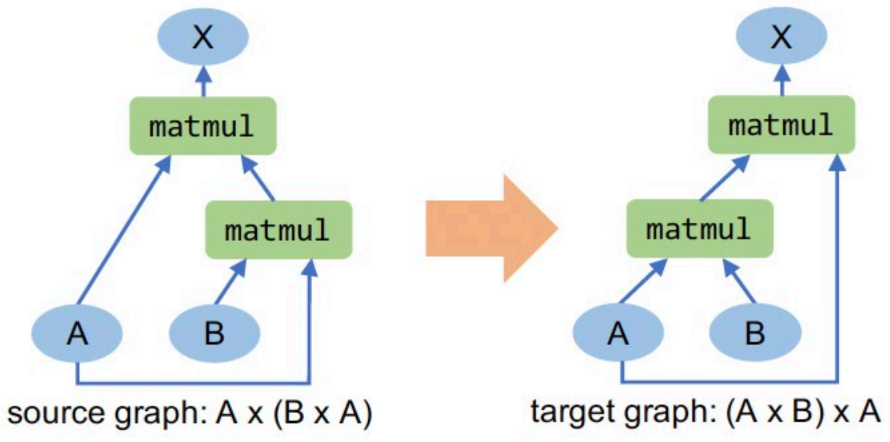
Approach: Formal Verification

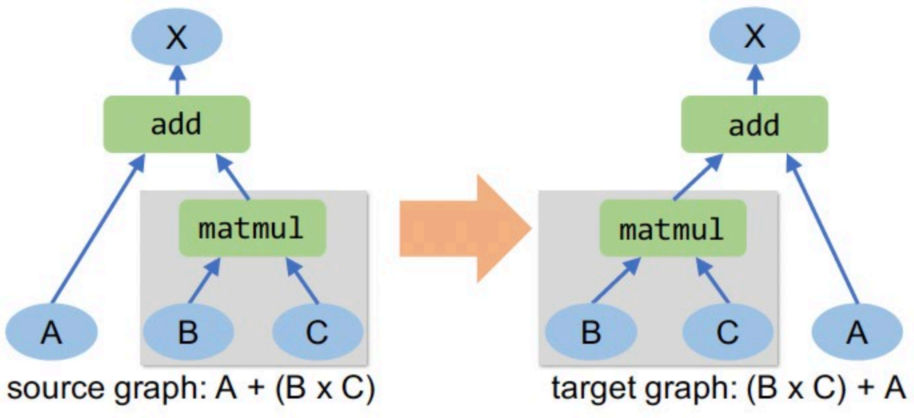
Operator Property	Comment
$\forall x, y, z.\ \mathtt{ewadd}(x, \mathtt{ewadd}(y, z)) = \mathtt{ewadd}(\mathtt{ewadd}(x, y), z)$	ewadd is associative
$\forall x, y.\ \mathtt{ewadd}(x, y) = \mathtt{ewadd}(y, x)$	ewadd is commutative
$\forall x, y, z.\ \mathtt{ewmul}(x, \mathtt{ewmul}(y, z)) = \mathtt{ewmul}(\mathtt{ewmul}(x, y), z)$	ewmul is associative
$\forall x, y.\ \mathtt{ewmul}(x, y) = \mathtt{ewmul}(y, x)$	ewmul is commutative
$\forall x, y, z.\ \mathtt{ewmul}(\mathtt{ewadd}(x, y), z) = \mathtt{ewadd}(\mathtt{ewmul}(x, z), \mathtt{ewmul}(y, z))$	distributivity
$\forall x, y, w.\ \mathtt{smul}(\mathtt{smul}(x, y), w) = \mathtt{smul}(x, \mathtt{smul}(y, w))$	smul is associative
$\forall x, y, w.\ \mathtt{smul}(\mathtt{ewadd}(x, y), w) = \mathtt{ewadd}(\mathtt{smul}(x, w), \mathtt{smul}(y, w))$	distributivity
$\forall x, y, w.\ \mathtt{smul}(\mathtt{ewmul}(x, y), w) = \mathtt{ewmul}(x, \mathtt{smul}(y, w))$	operator commutativity
$\forall x.\ \mathtt{transpose}(\mathtt{transpose}(x)) = x$	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
$\forall x, y, z.\ \mathtt{matmul}(x, \mathtt{matmul}(y, z)) = \mathtt{matmul}(\mathtt{matmul}(x, y), z)$	matmul is associative
$\forall x, y, w.\ \mathtt{smul}(\mathtt{matmul}(x, y), w) = \mathtt{matmul}(x, \mathtt{smul}(y, w))$	matmul is linear
$\forall x, y, z.\ \mathtt{matmul}(x, \mathtt{ewadd}(y, z)) = \mathtt{ewadd}(\mathtt{matmul}(x, y), \mathtt{matmul}(x, z))$	matmul is linear

...

Approach: Pruning Redundant Substitutions

- Redundant substitutions are subsumed by more general, valid substitutions
- Input tensor renaming
- Common subgraph





Approach: Joint Optimisation

- Utilises MetaFlow cost-based backtracking search algorithm
- Considers data layout optimisation opportunities
- Joint optimisation uncovers otherwise impossible optimisations
- Costs are given by execution times of specific operators
- Cycle removal
- Alpha parameter prunes search space

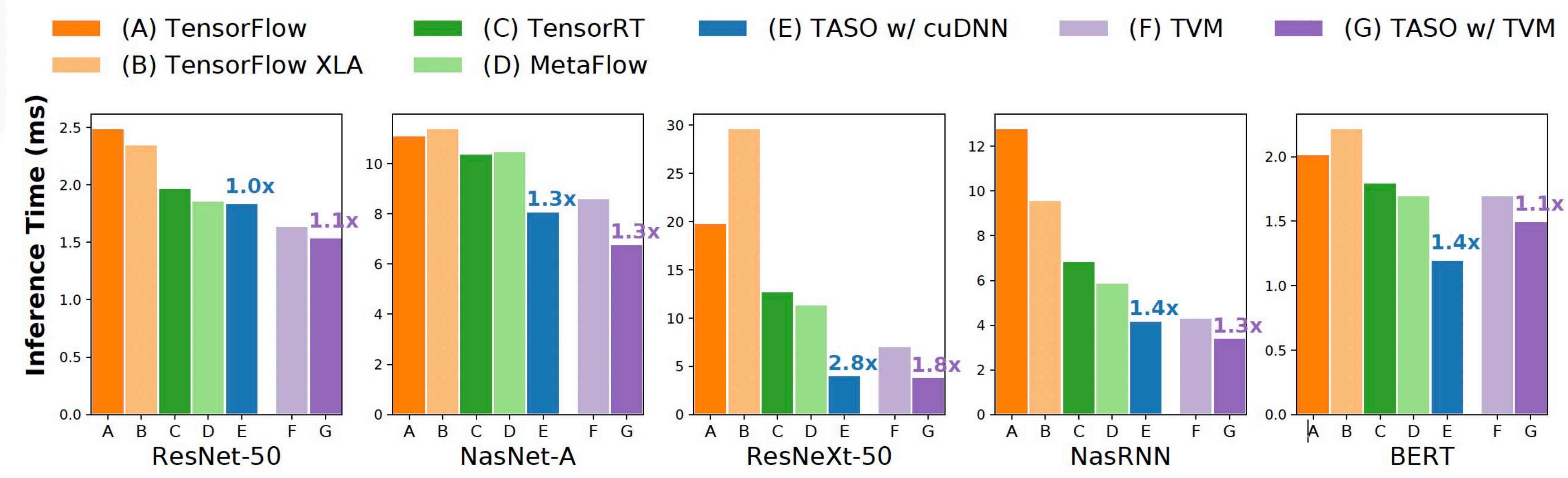
Algorithm 2 Cost-Based Backtracking Search

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

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

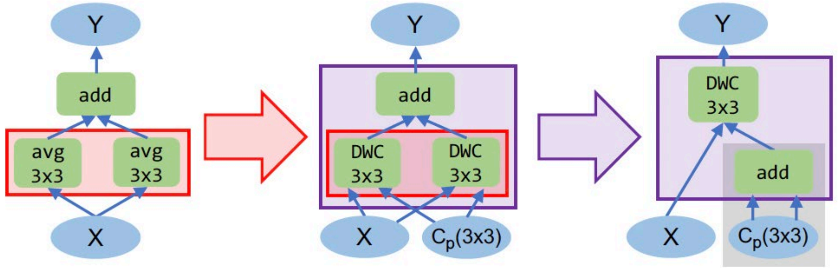
Evaluation: Optimisation

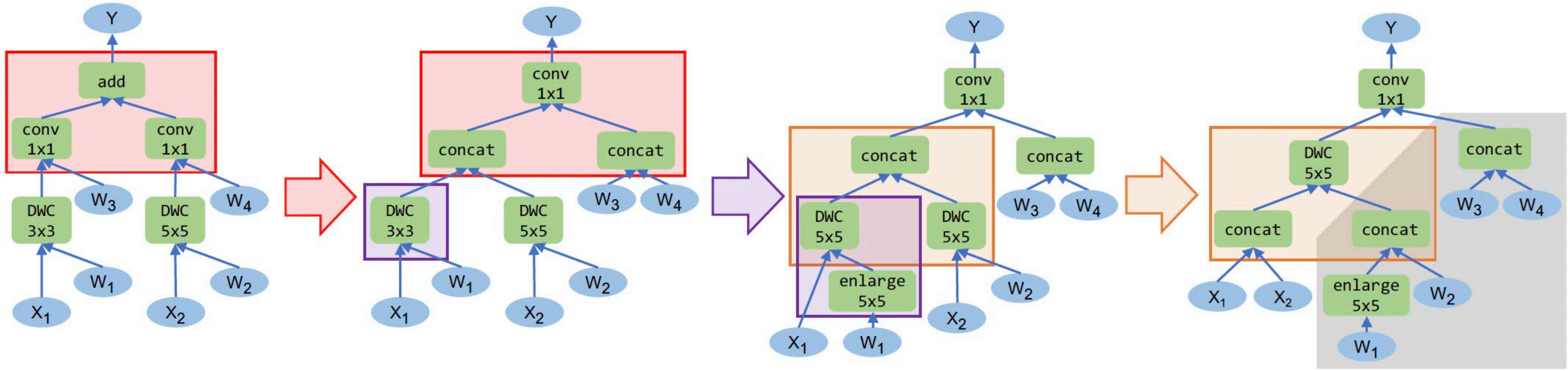
- Setup – tested on 5 DNNs
- Successful automatic optimisation – inference time reduction
  - cuDNN: 1.3x to 2.8x
  - TVM: 1.1x to 1.8x



Evaluation: Substitutions

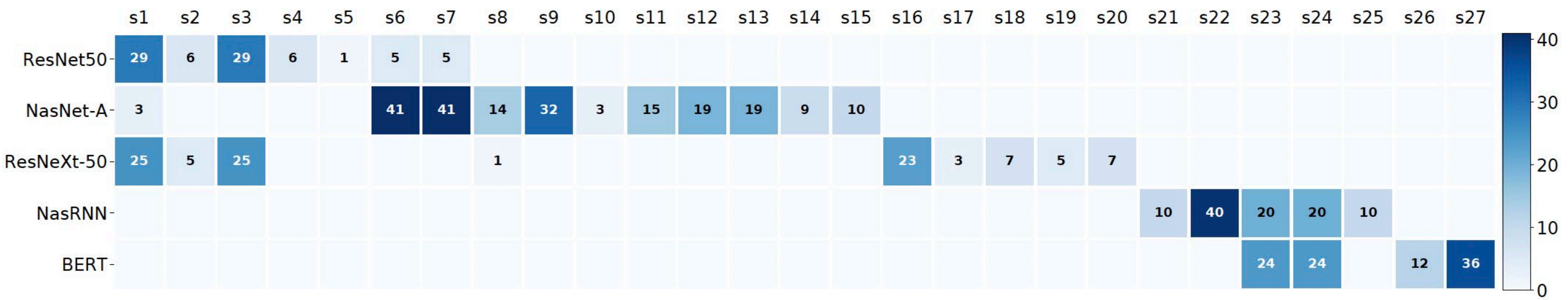
- NasNet was produced using neural architecture search
- Unconventional optimisations were discovered

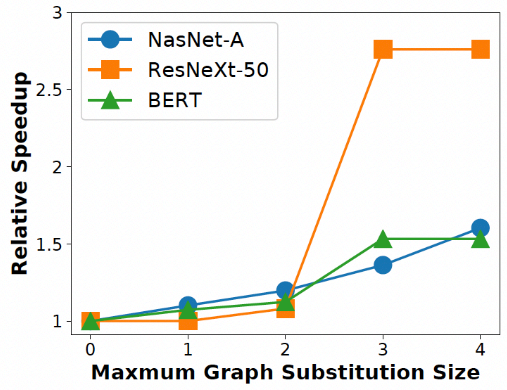




Evaluation: Substitutions

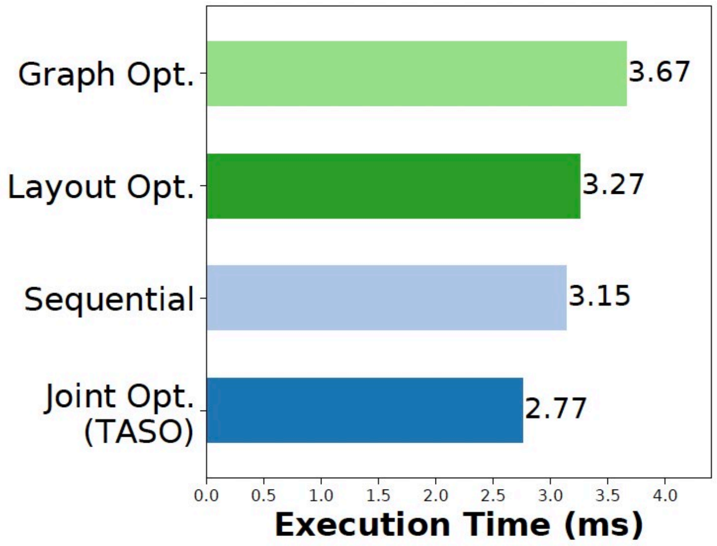
- Different DNNs used different optimisations, showing usefulness of TASO
- Scalability
  - Larger operator substitutions could be useful





Evaluation: Substitutions

- Joint optimisation
  - Better than individual or sequential
  - $(A \times B)$ -> $((B^T \times A^T)^T)$ with B^T in row-major and A^T in column-major
  - Phase ordering?
- Relatively quick - <10 minutes for each DNN



Review

Positives

- Novel idea
- Successful execution
    - Improves DNN performance
    - Reduces human effort
    - Extensible framework
- Seminal work in an exciting research area
    - Graph transformation backend still in use

Negatives

- Reliant on user provided operator properties
- Scalability of generator
- Phase ordering problem + search procedure
- Cost model has issues

Future Works

- Future works have built on this approach
- PET
  - Partially equivalent optimisations
- TENSAT
  - Equality saturation
- X-RLflow
  - RL approach to searching optimisation space
- REGAL
  - Transfer knowledge