

Equality Saturation for Tensor Graph SuperOptimization

Authors: Y. Yang, P.M. Phothilimthana, Y. R. Wang, M. Willsey, S. Roy, J. Piennar

MLSys'21

Presenter: Grant Wilkins (gfw27)



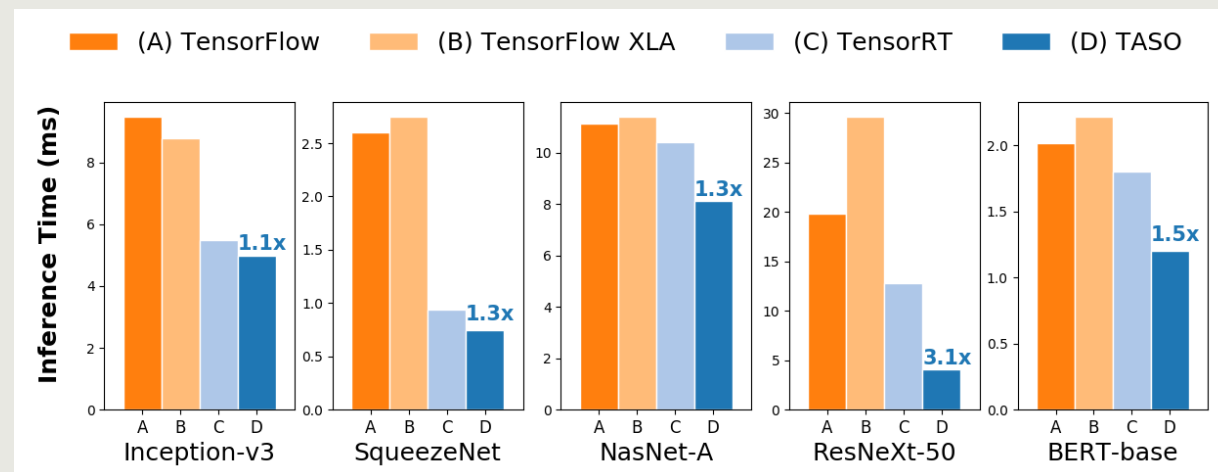
Situating this Study

- Neural networks have tensor representations programmatically
- Tensors can be *rewritten* to optimize compilation
- State of the art: TASO
- Improve on previous methods by attempting to consider many more possible representations



TASO (*Tensor Algebra SuperOptimizer*)

- Takes arbitrary DNN model
- Auto-generates graph transformations to build a large search space of potential computation graphs.
- Employs cost-based search algorithm to explore the space
- Automatically discovers highly optimized computation graphs.



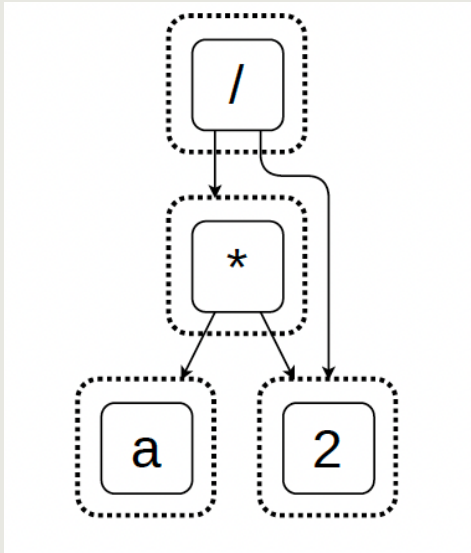
What this paper adds?

- **Tensat:** Optimization of search for equivalent optimal graphs.
- *Equality saturation*
- Exploration phase: generating and rewriting all possibilities
- Extraction phase: selects graph with lowest “cost”
- Authors find 16% faster than TASO, and reduces optimization by 300x

	Search time (s)		Runtime speedup (%)	
	TASO	TENSAT	TASO	TENSAT
NasRNN	177.3	0.5	45.4	68.9
BERT	13.6	1.4	8.5	9.2
ResNeXt-50	25.3	0.7	5.5	8.8
NasNet-A	1226	10.6	1.9	7.3
SqueezeNet	16.4	0.3	6.7	24.5
VGG-19	8.9	0.4	8.9	8.9
Inception-v3	68.6	5.1	6.3	10.0

Table 1. Comparison of optimization time and runtime speedup of the optimized computation graphs over the original graphs, TASO (Jia et al., 2019a) v.s. TENSAT.

Equality Saturation

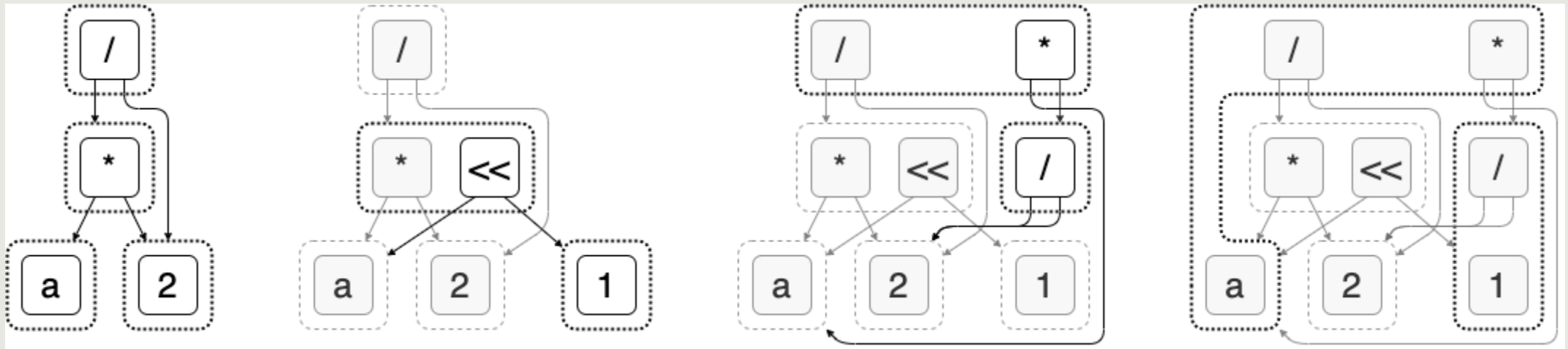


Dotted: e-class
Blocks: nodes

- *What is an e-graph?*
 - *Just set of equivalence relations with e-nodes*
 - *Represents many terms to optimize*
- *What is an e-class?*
 - *Represents terms under rewrite*
 - *Groups equivalent terms*
- *What is an e-node?*
 - *In e-classes, contains e-classes as children*
 - *Terms built out of applied operators*

Example: Exploration

Original Problem: $(a * 2) / 2$

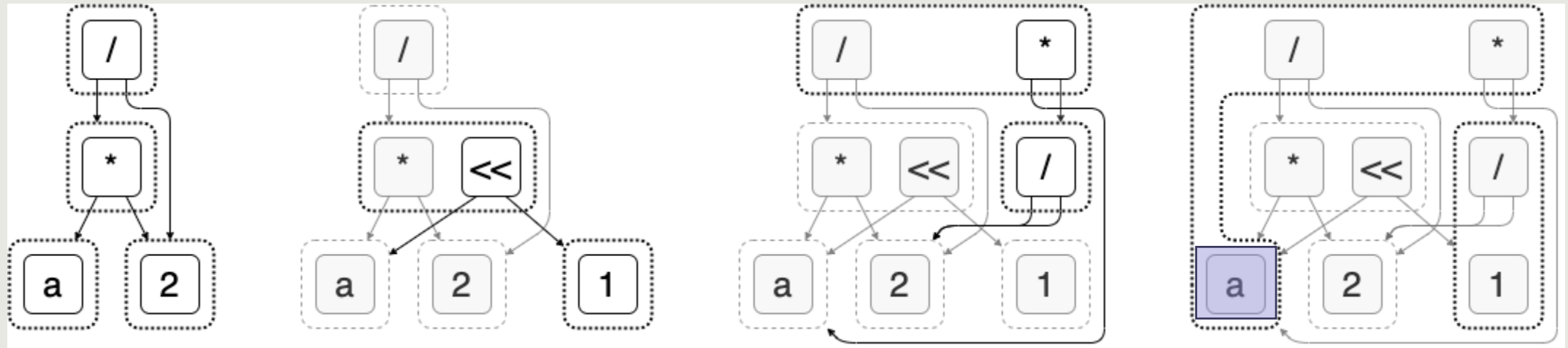


According to Tensat and rewrite rules, the e-graph expands and you find all equivalent programs.

1. $x * 2 \rightarrow x \ll 1$
2. $(x * y) / z \rightarrow x * (y / z)$
3. $x / x \rightarrow 1$
4. $x * 1 \rightarrow x$

Example: Extraction

Define a cost function, then pick least costly representation



$$(a * 2) / 2 \rightarrow a$$

How are we going to extract?

$$\text{Minimize: } f(x) = \sum_i c_i x_i$$

Subject to:

$$x_i \in \{0, 1\}, \quad (1)$$

$$\sum_{i \in e_0} x_i = 1, \quad (2)$$

$$\forall i, \forall m \in h_i, x_i \leq \sum_{j \in e_m} x_j, \quad (3)$$

$$\forall i, \forall m \in h_i, t_{g(i)} - t_m - \epsilon + A(1 - x_i) \geq 0, \quad (4)$$

$$\forall m, 0 \leq t_m \leq 1, \quad (5)$$

- Old: Greedy extraction:
 - Calculate all of the costs and then take the minimum.
- NEW: Integer Linear Programming (ILP) extraction
 - Binary x_i for each node
 - c_i cost for each node
 - Constraint 2: Must have a root node
 - Constraint 3: Requires a child in each e-class to be selected
 - Constraints 4 and 5: Keeps a proper Topological order of selection
 - Without proof: always gives a valid graph with lowest cost.

Cycles

Solve cycles through efficient filtering by keeping a log of the previous graphs and noting if there's a loop introduced from new rewrite.

At the end backtrack depth-first and remove last node that caused the cycle.

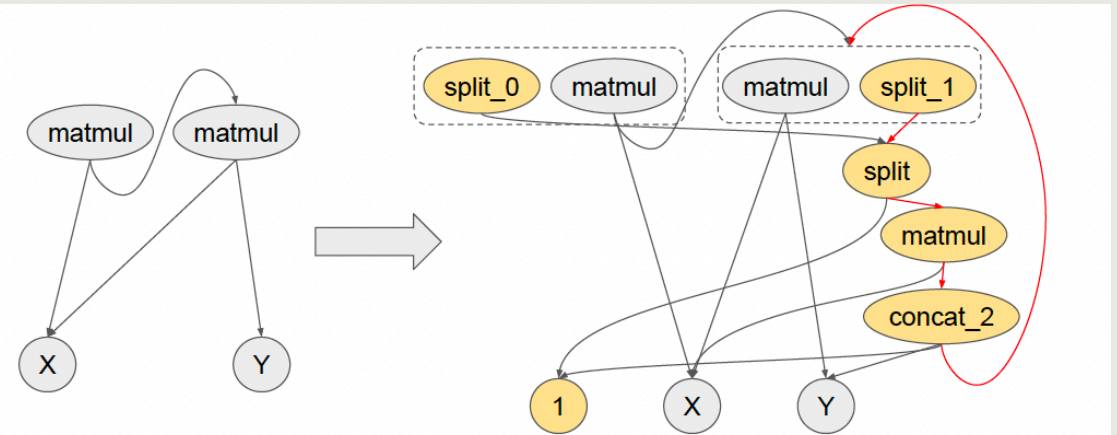


Figure 3. Example on how a valid rewrite can introduce cycles into the e-graph. RHS is the resulting e-graph after applying the rewrite rule from Figure 2 to the LHS. Dotted lines circles the e-classes. We omit the e-classes with a single node for clarity. If the node split_1 is picked in the right e-class, then the resulting graph will have a cycle (indicated by the red edges).

How does Tensat actually perform?



Finding: Tensat is more performant than TASO across the board for tensor optimization

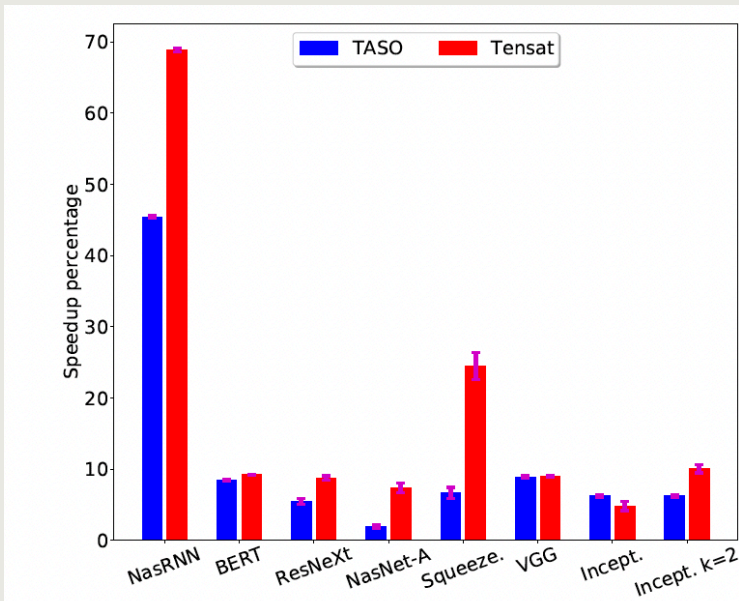


Figure 4. Speedup percentage of the optimized graph with respect to the original graph: TASO v.s. TENSAT. Each setting (optimizer \times benchmark) is run for five times, and we plot the mean and standard error for the measurements.

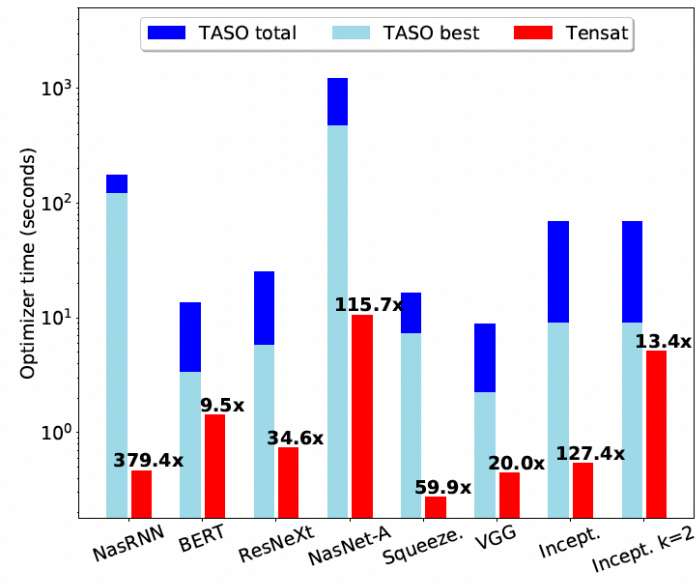


Figure 5. Optimization time (log scale): TASO v.s. TENSAT. “TASO total” is the total time of TASO search. “TASO best” indicates when TASO found its best result; achieving this time would require an oracle telling it when to stop.

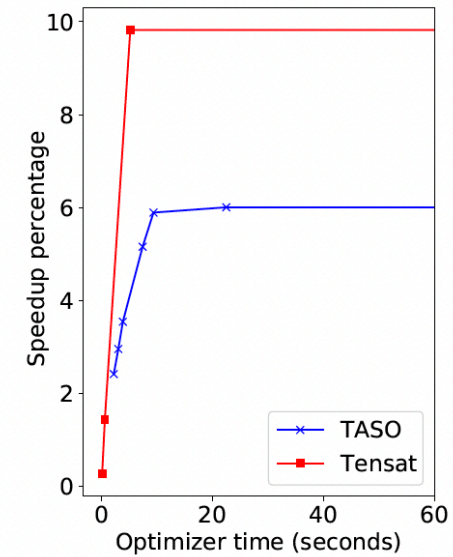


Figure 6. Speedup over optimization time for TASO and TENSAT, on Inception-v3. We use a timeout of 60 seconds.

Finding: There is a tradeoff for how much you can grow your search space in memory and runtime.

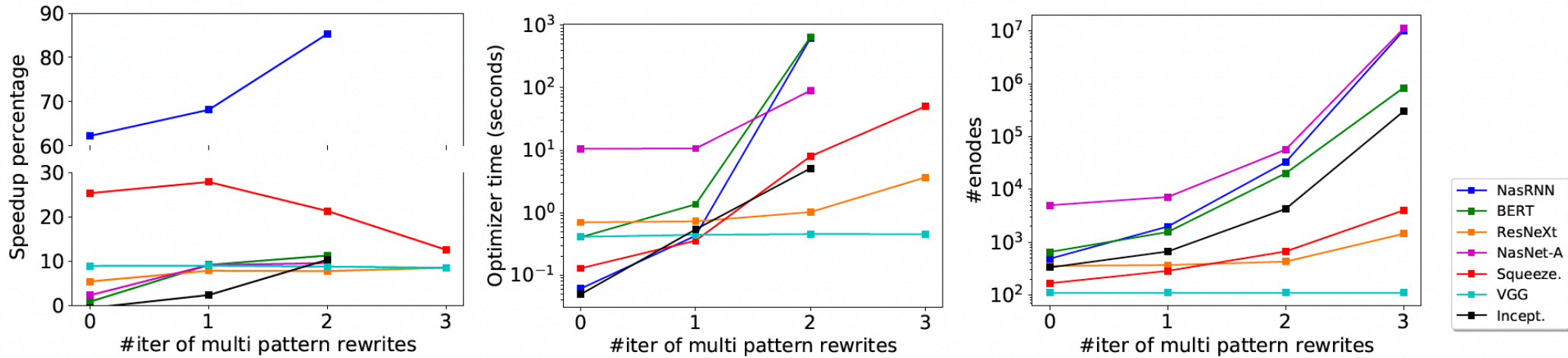



Figure 7. Effect of varying the number of iterations of multi-pattern rewrites k_{multi} . For BERT, NasNet-A, NasRNN, Inception-v3, the ILP solver times out at one hour for $k_{\text{multi}} = 3$. Left: speedup of the optimized graphs (the y -axis is split for clarity). Middle: time taken by TENSAT. Right: final e-graph size (number of e-nodes). The middle and right figures are in log scale.

Big Picture Summary

- Tensat is a tool to optimize tensor rewrites
 - Through compact e-graph representation, exponentially generates alternative rewrites of tensor programs
 - Performs equality saturation: two-phase approach to expand search space and reduce runtime for selection for rewrites
 - Achieves significant run-time and optimization advances over TASO
- 

My critique

Cons:

- Unclear that cost function used is optimal for the e-graph selection and the authors should experiment with different ones.
- Author is quick to assume relevance of work without clear motivation beyond speedup.
- Memory overhead of solution discussed but not quantified.

Pros:

- Very lucid explanation of equivalence graphs.
- Demonstrates dominant results over TASO.
- Clearly documented ideas behind methodology and sourcing of different portions of solution.

Questions?



Table 2. Operators supported by TENSAT. There are four types for the nodes in our representation: tensor type (T), string type (S), integer type (N), and tensor tuple type (TT). The integer type is used to represent parameters of the operators, such as stride, axis, and also padding and activation modes (by representing different modes using different integers). The more complex, variable-length parameters (e.g. shape, axes permutation) are represented using the string type according to the specified formats.

Operator	Description	Inputs	Type signature
ewadd	Element-wise addition	input ₁ , input ₂	(T, T) → T
ewmul	Element-wise multiplication	input ₁ , input ₂	(T, T) → T
matmul	Matrix multiplication	activation, input ₁ , input ₂	(N, T, T) → T
conv ^a	Grouped convolution	stride _h , stride _w , pad., act., input, weight	(N, N, N, N, T, T) → T
relu	Relu activation	input	T → T
tanh	Tanh activation	input	T → T
sigmoid	Sigmoid activation	input	T → T
poolmax	Max pooling	input, kernel _{h,w} , stride _{h,w} , pad., act.	(T, N, N, N, N, N, N) → T
poolavg	Average pooling	input, kernel _{h,w} , stride _{h,w} , pad., act.	(T, N, N, N, N, N, N) → T
transpose ^b	Transpose	input, permutation	(T, S) → T
enlarge ^c	Pad a convolution kernel with zeros	input, ref-input	(T, T) → T
concat _n ^d	Concatenate	axis, input ₁ , ..., input _n	(N, T, ..., T) → T
split ^e	Split a tensor into two	axis, input	(N, T) → TT
split ₀	Get the first output from split	input	TT → T
split ₁	Get the second output from split	input	TT → T
merge ^f	Update weight to merge grouped conv	weight, count	(T, N) → T
reshape ^g	Reshape tensor	input, shape	(T, S) → T
input	Input tensor	identifier ^h	S → T
weight	Weight tensor	identifier ^h	S → T
noop ⁱ	Combine the outputs of the graph	input ₁ , input ₂	(T, T) → T

Graph Runtime (ms)	Original	Greedy	ILP
BERT	1.88	1.88	1.73
NasRNN	1.85	1.15	1.10
NasNet-A	17.8	22.5	16.6

Table 4. Comparison between greedy extraction and ILP extraction, on BERT, NasRNN, and NasNet-A. This table shows the runtime of the original graphs and the optimized graphs by greedy extraction and ILP extraction. The exploration phase is run with $k_{\text{multi}} = 1$.

Extraction time (s)	k_{multi}	With cycle		Without cycle
		real	int	
BERT	1	0.96	0.98	0.16
	2	>3600	>3600	510.3
NasRNN	1	1116	1137	0.32
	2	>3600	>3600	356.7
NasNet-A	1	424	438	1.81
	2	>3600	>3600	75.1

Table 5. Effect of whether or not to include cycle constraints in ILP on extraction time (in seconds), on BERT, NasRNN, and NasNet-A. For the cycle constraints, we compare both using real variables and using integer variables for the topological order variables t_m .

Exploration time (s)	k_{multi}	Vanilla	Efficient
BERT	1	0.18	0.17
	2	32.9	0.89
NasRNN	1	1.30	0.08
	2	2932	1.47
NasNet-A	1	3.76	1.27
	2	>3600	8.62

Table 6. Comparison between vanilla cycle filtering and efficient cycle filtering, on the exploration phase time (in seconds) for BERT, NasRNN, and NasNet-A.